Controlling Robotic Mobile Fulfillment Systems

and further topics in decision support

Marius Merschformann

Paderborn 2018

University of Paderborn
Faculty of Business Administration and Economics
Department of Business Information Systems
Warburger Straße 100
33098 Paderborn, Germany
www.uni-paderborn.de

Abstract

This thesis focuses on *Robotic Mobile Fulfillment Systems* (RMFS), which are a new type of parts-to-picker order fulfillment system, designed especially for e-commerce warehouses. The mobile robots or automated guided vehicles (AGV) bring movable shelves, called pods, to workstations where inventory is put on or removed from the pods. The main focus of this thesis is on decision problems arising in the control of RMFS and methods for studying them. A detailed discrete event simulation model is proposed and made available as open source software for the purpose of studying dynamic effects occurring in RMFS. The simulation model is used to analyze multiple path planning algorithms with kinematic constraints as well as decision rules for the core control problems occurring in RMFS operation. Furthermore, a novel queuing theory approach is used to study efficient control mechanisms for systems facing changing conditions. Another part of the work investigates new control opportunities that arise from the dynamic inventory behavior when complete inventory racks can be moved freely within the system. Finally, as a result of collaborative opportunities, a second part of this thesis covers further topics on decision support. These address further applications and supporting methods in algorithm configuration and heuristic decision algorithm topics for crew rostering and container loading applications.

Abstract (German)

Diese Arbeit beschäftigt sich vorwiegend mit *Robotic Mobile Fulfillment Systems* (RMFS), welche eine neue Art speziell für den E-Commerce Bereich entwickeltes Ware-zu-Mann System darstellen. Die mobilen Roboter oder auch Fahrerlose Transportfahrzeuge (FTF) bringen transportfähige Regale, sogenannte Pods, zu den Arbeitsstationen, an welchen Material ein- bzw. ausgelagert wird. Diese Arbeit beschäftigt sich mit den Entscheidungsproblemen, welche für den effizienten Einsatz eines RMFS bewältigt werden müssen, sowie Methoden zum Studium dieser. Zu diesem Zweck wird eine detaillierte, ereignisorientierte Simulation beschrieben und zur weiteren Verwendung als Open Source Software zur Verfügung gestellt. Weiterhin wird das entwickelte Simulationsmodell verwendet um neuartige Pfadplanungsverfahren unter Berücksichtigung kinematischer Anforderungen sowie Entscheidungsmechanismen für Kernplanungsprobleme im Betrieb von RMFS zu analysieren. Darüber hinaus wird eine neuartige Methode aus dem Bereich Warteschlangentheorie entwickelt und angewandt um effiziente Regeln für den Betrieb eines RMFS unter sich verändernden Bedingungen abzuleiten. Ein weiterer Teil dieser Arbeit untersucht den Nutzen neuer Steuerungsmöglichkeiten, welche sich aus sich ständig bewegenden Regalen (und damit auch Inventar) ergeben. Über den Bereich RMFS hinaus enthält diese Arbeit weitere aus Kollaborationsmöglichkeiten entstandene Arbeiten zum übergeordneten Thema Entscheidungsunterstützung. Diese beschäftigen sich mit weiteren Anwendungen und unterstützenden Methoden im Bereich Algorithm Configuration, sowie heuristischen Entscheidungsverfahren für die Crew Rostering und Container Loading Probleme.

Preface

This thesis focuses on methods and applications for and in the field of decision support. The main focus is on *Robotic Mobile Fulfillment Systems* (RMFS), which are a new type of parts-to-picker order fulfillment system. However, during the work on this thesis collaboration opportunities on further methods and applications in decision support presented themselves and have been conducted alongside the main project. While the topics across this thesis are quite diverse, they all share the common objective of improving efficiency by applying novel methodology and algorithms to the respective application areas.
To provide a clear structure to the reader this thesis consists of two parts. The RMFS part (part I) is kept self-contained and may also be considered as standalone. It consists of a short introduction to the RMFS topic, summaries of the research papers and the 5 research papers themselves. The next part (part II) contains a combination of different research topics in the Operations Research & Artificial Intelligence fields. This part consists of a brief motivation for the research and how it relates to the main topic, the summaries of the research papers and the 3 research papers themselves.

Dortmund, 2018

(Marius Merschformann)

Acknowledgements

This work originated from research I conducted as a member of the International Graduate School of Dynamic Intelligent Systems and the Decision Support & Operations Research Lab at the University of Paderborn. I am very grateful for the opportunity to work on this thesis as well as the fun and pain I had that goes along with it. Both institutions were the organizational backbone of this thesis. I am still thankful for the long opening times of the office complex enabling big leaps in short time frames.
Foremost, I like to thank my supervisor Prof. Dr. Leena Suhl for drawing my attention to the field of Operations Research in the first place as well as all the interesting projects I was able to work on during my student assistant and PhD time. She inspired me a lot and was very supportive at all times. I also enjoyed the atmosphere she created at our group greatly. The environment was very beneficial for a creative and fruitful working process. For this, I also want to thank all of my colleagues. In particular, I would like to thank Daniela Guericke, Stefan Guericke, Daniel Müller, Lars Beckmann and Christoph Weskamp for all the long discussions and other fun times we had. Furthermore, I would like to thank Prof. Dr. Kevin Tierney for his great feedback and support. It has been a great time working with all of you and I very much like that lifelong friendships originated from it. Further on, I enjoyed all the discussions with students I supervised during the past years and I want to thank them for their efforts. Notably, I want to thank Daniel Erdmann and Jonas König who helped kick-starting the RMFS project during the early phase a lot.
I would like to thank Prof. Dr. Lin Xie for her early advances which were a starting point for much of my work, as well as Hanyi Li who supported the RMFS project greatly. They were also responsible for bringing up the funding for most of my PhD time via the Ecopti GmbH by supporting the IGS scholarship. Without them I would not have had this interesting opportunity of researching such a modern and dynamic system which I enjoyed a lot and what influenced my ongoing path heavily.

It was even a chance for me to test my tinkering driven self by integrating physical components with my remaining research.
I am very grateful for the collaboration with Tim Lamballais and Prof. Dr. René de Koster of the Erasmus University Rotterdam. What initially was planned as a short collaborative effort lead to a long partnership which broadened my view on modern logistics and improved my take on complicated projects in many aspects.
I like to thank my family who always supported me and have been very patient with me for so long now. I also like to thank all of my friends, especially, Julian Drücker who sparked and pushed my urge for research from the very beginning.
Most of all I want to thank my wife Sina, who endured all of these years and backed me up even in moments of most grief. Thank you so much for everything!

List of publications

Scientific papers on Robotic Mobile Fulfillment Systems included in this thesis:

Chapter 2: Marius Merschformann, Lin Xie, Hanyi Li: *RAWSim-O: A Simulation Framework for Robotic Mobile Fulfillment Systems*, Logistics Research (2018), Volume 11, Issue 1

Chapter 3: Marius Merschformann, Lin Xie, Daniel Erdmann: *Multi-Agent Path Finding with Kinematic Constraints for Robotic Mobile Fulfillment Systems*, submitted to Journal of Artificial Intelligence Research

Chapter 4: Marius Merschformann, Tim Lamballais, René de Koster, Leena Suhl: *Decision Rules for Robotic Mobile Fulfillment Systems*, submitted to European Journal of Operational Research

Chapter 5: Tim Lamballais, Marius Merschformann, Debjit Roy, Leena Suhl, René de Koster: *Dynamic Policies for Resource Reallocation in a Robotic Mobile Fulfillment System with Time-Varying Demand*, submitted to Transportation Science: Focused Issue on Urban Freight Transportation & Logistics

Chapter 6: Marius Merschformann: *Active repositioning of storage units in Robotic Mobile Fulfillment Systems*, Selected Papers of the Annual International Conference of the German Operations Research Society (GOR), Freie Universität Berlin, Germany, September 6-8, 2017

Scientific papers on other topics included in this thesis:

Chapter 8: Yuri Malitsky, Marius Merschformann, Barry O'sullivan, Kevin Tierney: *Structure-Preserving Instance Generation* (2016), LION 10 - Learning and Intelligent Optimization, At Ischia, Italy, 29.05.-01.06.2016

Chapter 9: Marius Merschformann: *Algorithm Configuration Applied to Heuristics for Three-Dimensional Knapsack Problems in Air Cargo* (2015), Algorithm Configuration: Papers from the 2015 AAAI Workshop

Chapter 10: Lin Xie, Marius Merschformann, Natalia Kliewer, Leena Suhl: *Meta-heuristics approach for solving personalized crew rostering problem in public bus transit*, Journal of Heuristics (2017), Volume 23, Issue 5, pp 321–347

Contents

Part I

Controlling Robotic Mobile Fulfillment Systems

Chapter 1

Synopsis on RMFS

From 2008 to 2017 e-commerce market penetration in Europe did more than double, i.e., European citizens with at least one order per three months increased from 23% to 48% (see [8]). Similar trends can be observed worldwide. There seems to be no indication for a decline of this development, instead, even more products become available online (e.g., groceries). Furthermore, a trend to offer more variants of products can be observed in many industries. Such developments imply a challenge for distribution centers (DC) enabling the sortation of the growing SKU (stock-keeping unit) numbers for the increasing count of small customer orders seen in e-commerce. Customer orders in e-commerce usually consist of only a few lines, i.e., often only one or two products are ordered at a time. Moreover, todays e-commerce customers expect fast deliveries on the next or even the same day. This reduces the time that can be spent on actually picking the order at the DC.

The DC, which is a part of virtually every logistics chains, is responsible for temporarily storing products and sorting them according to given orders. This means that the products received in larger homogeneous packages need to be separated, stored, picked, packed and send out. In Figure 1.1 an abstract outline of the DC within the logistics chain is shown. For the sake of clarity it is assumed that no further DCs exist between suppliers and downstream operations. The description aims to address the differences in e-commerce and retail logistics, while it should be noted that these are not mutually exclusive. Nowadays, many retail companies are looking for ways of combining their retail with e-commerce business, thus, often requiring combined operations at their DCs.

At the inbound side of the DC products brought by trucks arrive in homogeneous packagings, i.e., all same products on a pallet or in a carton. In addition to the ho-

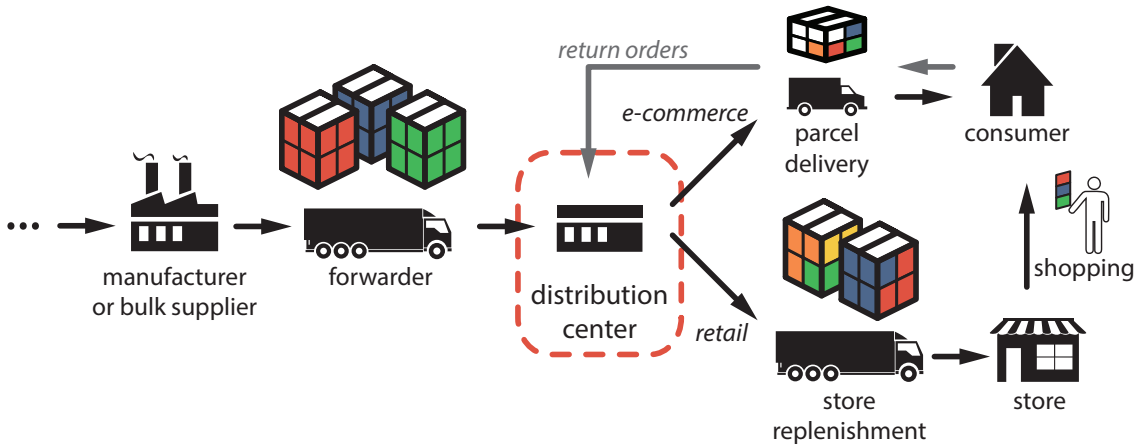


Figure 1.1 The basic logistics chain (split by retail vs. e-commerce)

mogeneous products, especially in e-commerce logistics, often return orders need to be handled. These may require refurbishing and result in a more heterogeneous inbound material flow. After the processing at the DC the prepared orders are picked up and shipped at the outbound side. In the case of e-commerce the parcels are sent to the customer directly, while in the case of retail pallets or movable roll cages (trolleys) are shipped to the stores. The customers then pick up the products at the store themselves. The key difference considering the DC operations challenge is the severely smaller number of lines and units overall in an order, while the number of orders (destinations) is much higher. From this point of view the store may be seen as an additional storage and sortation instance where the customers assist the sortation task themselves. In contrast, this effort must be handled by the DC alone in case of e-commerce.

In the same period in which the e-commerce market penetration doubled, the cost of labor in the transportation and logistics sector in Europe increased by 14.8 % (see [7]). Since DC operations are not exactly a value adding process, costs are usually kept minimal. Therefore, more automated solutions, so-called parts-to-picker systems, become more relevant. According to a study of [18] in classical manual order picking systems the tasks of search and travel account for 70 % of the time a picker spends per shift (see [18], p. 434). Both processes are virtually eliminated in a parts-to-picker system. However, not only the growing e-commerce market and increased labor costs drive the challenge for intralogistics operations, but also the quicker product innovations in combination with the higher number of product variants. This and nowadays generally fast-paced markets increase the volatility of customer demands which in turn increases the demand for more flexible intralogistics solutions. While increasing labor costs are already a driver for more automated logistics systems, many of these aren't flexible due to long design and installation

phases and immovable parts. The Robotic Mobile Fulfillment System (RMFS) is one parts-to-picker system aiming to bridge this gap, i.e., enabling automated processes in a more mobile solution. This system is the central focus of this work.
The demand for more mobile solutions in logistics did not go unnoticed. According to a market analysis of Interact Analysis, a market research consulting agency, the market of "Mobile Robots in Logistics Centers" is forecast to surpass three billion $ in 2022 (see [17]). The mobile robots focused by the analysis are AGVs (automated guided vehicle driving on pre-defined paths) and AMRs (autonomous mobile robots capable of free-roaming). The former also include the robots used in an RMFS.

1.1 Distribution center process

After the previous description on processes before and after the DC in the e-commerce and retail industry, this section gives a short outline about typical processes inside of the DC. In general a multitude of systems work together to handle the storage and sortation task at a DC, because there is hardly ever one system capable of handling all different product attributes (e.g., shapes & sizes; normal, chilled or frozen; etc.) and throughput requirements a DC is confronted with. Additionally, some intermediate processes may be required at one DC, but can be eliminated at another. However, in the following the typical process steps that are very common, especially in more automated DCs, are identified and described in sequence of material flow to supply background information to the reader.
This material flow is outlined in Figure 1.2. The process at a DC starts with the arrival of new material which is typically delivered by trucks either of third-party logistics companies or suppliers themselves. These products stored on pallets or in cartons are first handled in the receiving area where they are shortly buffered, checked for damaging, booked and pre-sorted for the subsequent processes. Next, there may be a manual (e.g.: forklift truck based) or automated (e.g.: crane-based) pallet warehouse in which most of the pallets are temporarily stored until they are needed to replenish following processes. The pallets need to be depalletized for further processing, either manually or automatically. The degree of automation depends on the handled material characteristics and the invest in automation equipment. Often manual and automated depalletizing is done concurrently. If no pallet warehouse is part of the facility but pallets need to be processed, they are depalletized immediately after arrival. The depalletized cartons or boxes may now be stored directly in the case buffer, a system usually designed for shorter storage periods and smaller loads than pallet warehouse systems. Additionally or alternatively, the products may be singulated further on, e.g., single mobile phones are put into bins (or so-called totes) instead of storing the complete carton (bins and totes are homogeneous plastic boxes in which the actual products are stored and

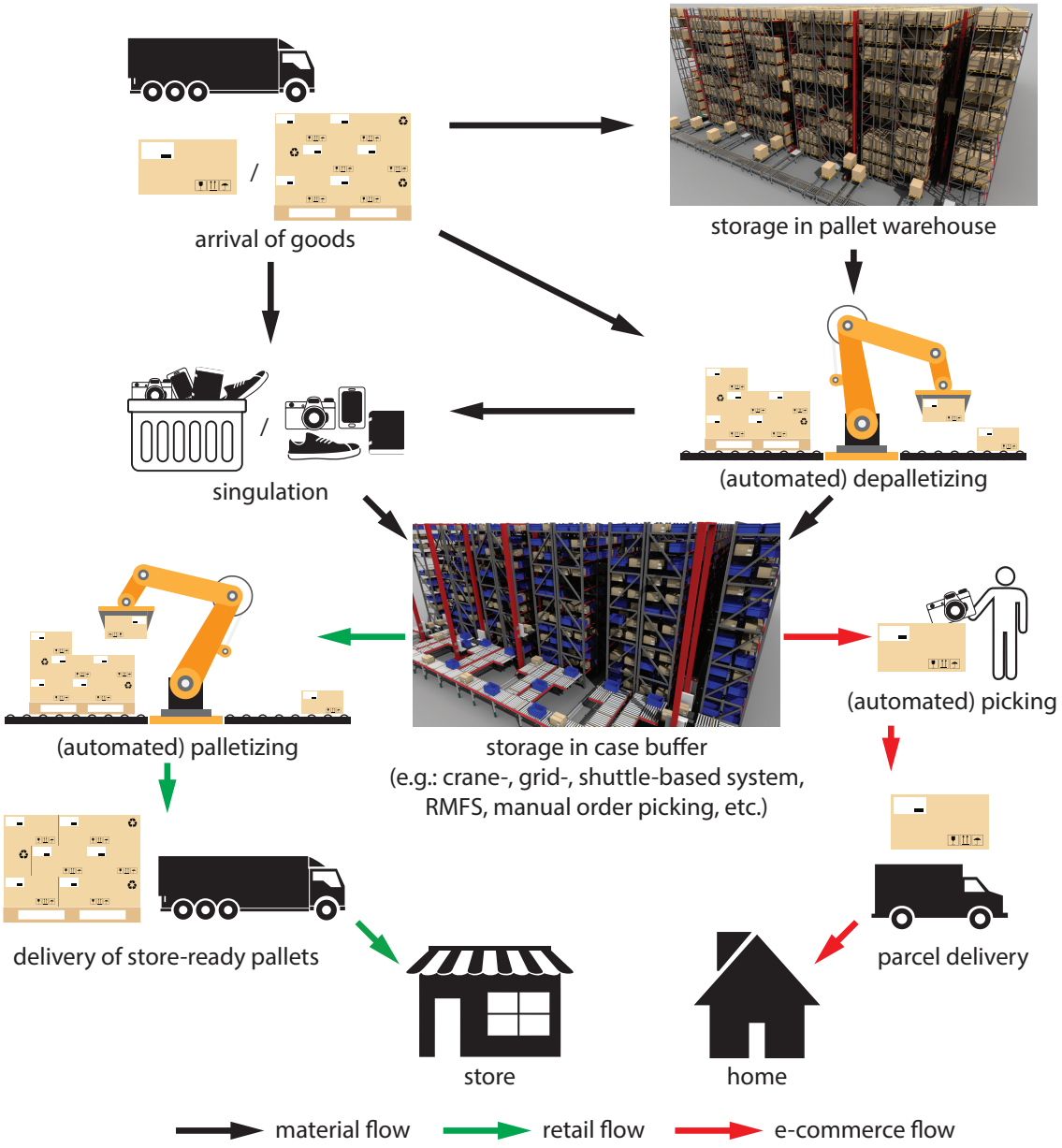


Figure 1.2 Typical material flow in retail and e-commerce distribution centers

transported within the DC). Further material flows are possible at the inbound side, like pallets that are only stored temporarily in the pallet warehouse and send to the store later on as a whole. This is typically done for bulk-ware and more common in the discounter retail business.
Especially, in e-commerce DCs pallet warehouses are less prominent and most stor-

age is handled by case buffer systems directly. One of the reasons for this is that higher numbers of SKUs are accessible more quickly. As a case buffer system classical approaches employ pickers walking the aisles of the warehouse picking the products from the shelves according to the orders they have been assigned. More automated approaches involve crane-, shuttle- or grid-based systems (see [1] for a comprehensive overview) delivering bins filled with the products to the pickers. In these approaches some pickers may also be robotic arms picking the material from the bin. An RMFS, the system in focus of this work, is another approach for this role. The reader may note that different systems are often combined to handle the overall material flow due to their characteristics. For example, a shuttle-based solution usually allows the highest throughput and may therefore contain the most popular products, while a less-expensive manual order picking system may host products that are ordered at lower rates. However, if suitable for the requirements, the case buffer may be the only storage system at the DC. I.e., only one system and no pallet warehouse handles the complete material flow.
For many case buffer systems bin handling and direct handling need to be distinguished depending on the stored products and system characteristics. For example, at a DC where full cartons are shipped to stores these may be handled directly, if the case buffer system supports this. This may be the case for crane- and shuttle-based solutions that can grasp cartons or shrink-foiled product bundles of reasonable quality. In contrast, for very small sized products or when single products are shipped, it is usually necessary to store them in bins.
Up to this point the solutions for the processes before the case buffer of retail and e-commerce operations may be quite similar, although singulation is more relevant in e-commerce and a pallet warehouse more typical in retail. However, the processes after the case buffer system, except for possible additional sortation processes, are quite different. The main reason for this are the different container used for shipping the products. In retail operations pallets or roll cages are shipped to the stores, while in e-commerce parcels are sent to the customer.
In retail operations most products are retrieved as cartons or shrink-foiled product bundles and palletized or put in a roll cage directly. Only for small sized products an additional picking process may exist in between both. The sequence in which products arrive at the palletizing station is often already prepared so that removing the goods at the corresponding store can be done quickly (e.g.: based on the aisles of the store). While palletizing is often already partially automated by using multi-axes robots and shrink-foil wrapping machines, the assembly of products in a roll-cage is typically still a manual process. The last step of the retail process is to load the pallets to a truck and deliver them to the stores.
In e-commerce operations single products are picked directly from the case buffer system and put into cartons that, after carton sealing and labeling, are then sent

to the customer. In large facilities or in case of multiple case buffer systems an additional consolidation process is required between picking and the actual sealing of the package. It should be noted that both operations may be employed at the same DC, if the company trades to both channels.

1.2 Robotic Mobile Fulfillment System

(a) Amazon system (© Amazon Robotics)

(b) Swisslog CarryPick (© Swisslog)

(c) GreyOrange Butler (© GreyOrange)

(d) Scallog System (© Scallog)

(e) Bleum Warehouse Robots (© Bleum)

(f) Alibaba system (© Alibaba Group)

Figure 1.3 Mobile robots of the current top RMFS integrators

In 2006 Kiva Systems, the company developing and selling the first RMFS, put their initial system into live operations at a Staples (an American company in the retail

industry) distribution center (see [20]). After six years selling the system as a parts-to-picker solution for distribution centers mostly to e-commerce retail companies the company was bought by Amazon.com, Inc in 2012. From then on the system (see Fig. 1.3a) was exclusively improved for and integrated into Amazon's distribution centers world-wide. The company which is nowadays called Amazon Robotics has more than 100,000 robots in operation since mid of 2017 and is planning to further expand the RMFS based operations (see [16]). However, since the system is not sold to other companies anymore, more logistics automation companies started developing their own RMFS. Some of the more prominent examples include Swisslog's CarryPick system (see Fig. 1.3b), the GreyOrange Butler (see Fig. 1.3c), the Scallog System (see Fig. 1.3d) and the Bleum Warehouse Robots (see Fig. 1.3e). All of these and more are aiming for the market gap left by Kiva Systems after it was bought by Amazon. Furthermore, direct competitors of Amazon started developing their own RMFS solutions. For example, the Alibaba Group, a competitor of Amazon in China, started the development for their distribution center operations (see Fig. 1.3f). All of these systems may technically differ in certain aspects, e.g., lifting mechanism, navigation technique, etc. However, they all employ the same basic principle of moving rack-like storage units to the pickers by using small mobile robots that move underneath them. Furthermore, all systems employ path systems that are (almost) entirely grid-like. This is not surprising, since rectangular storage units in a grid-based inventory area offer the highest storage density and allow the application of less complex planning algorithms. The term Robotic Mobile Fulfillment System is used to refer to the abstract concept of all of the prior systems. While the practitioners' systems may deviate in certain aspects (e.g., structure of the navigation graph) from the model representation chosen for the studies in this work, the key principle stays the same. Thus, results may serve as indications for practitioners too. The definition of the RMFS model is explained in more detail in the following.

In an RMFS mobile robots are used to bring rack-like storage units (so-called "pods") to the replenishment and pick stations (see Fig. 1.4). These workstations act like an interface of the RMFS in terms of material flow. Inventory is replenished into the pods (i.e., put away in the pods) at replenishment stations and picked from the pods at pick stations. This work assumes that one workstation solely takes one of these roles. Although it is technically possible to combine both roles for one workstation, it is often constrained by external DC processes. I.e., there is typically an overall material flow planned for a DC, as a result of truck bay locations, staging for shipping or other storage systems. Thus, if a pick station is not physically connected to the replenishment feed, it cannot serve as a replenishment station. Vice versa, a replenishment station not connected to the shipping process cannot serve as a pick station. However, processes enabling a combined role for stations

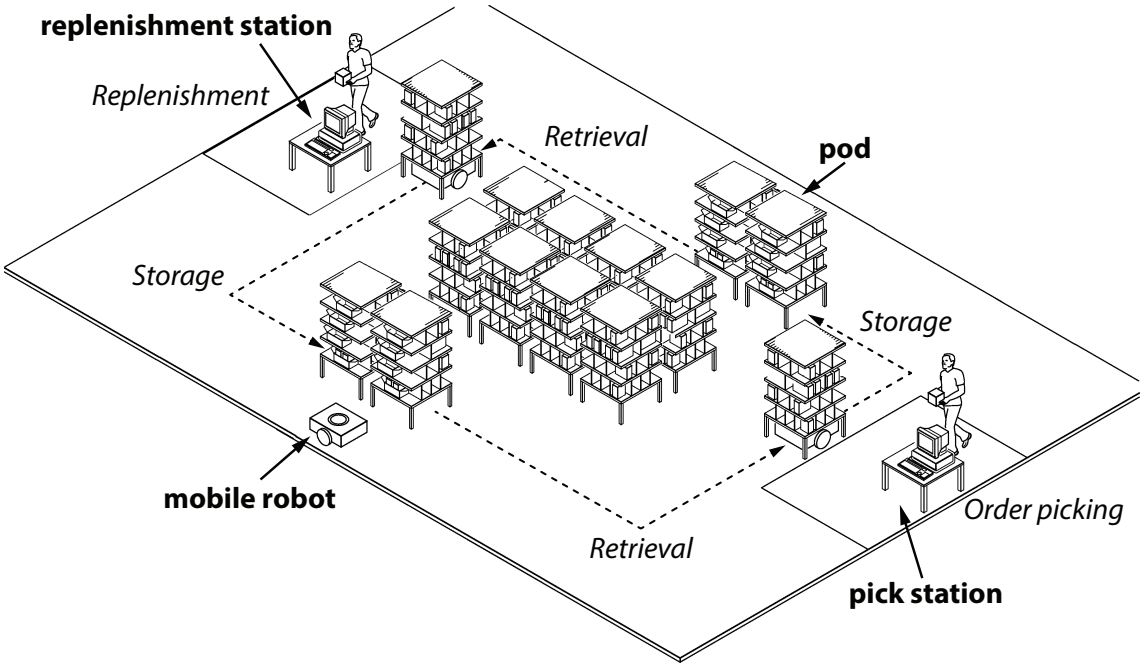


Figure 1.4 The central process and elements of an RMFS. Basic processes are indicated by italic descriptions and dashed lines; main elements are marked by bold descriptions and straight lines (diagram based on a Kiva Systems patent, see [10])

exist. For example, it is possible to temporarily store finished orders in a pod and eventually bring them to a shipping station or the shipping area itself.
The pods offering the storage space for inventory are mobile racks that can be picked up and set down by the robots. The pods are usually segmented into small compartments containing units of only one SKU each. The pods can be configured for the inventory handled at the DC and may yet include hangers for cloths. At the pick station the compartments to pick from are typically marked by laser indicators or similar techniques to reduce search time and also the chance of picking errors.
The mobile robot or drive unit used to bring the pods to the workstations can turn on the spot and independently from the pod it is carrying. The latter enables higher speeds in a more narrow storage area, since curve movement can be avoided. The robot can still change orientation while keeping the pod in it's original one. If instead the robot would turn the pod while turning around itself, wider aisles in between stored pods would be required (reducing storage density) or the pod needs to be set down before and picked up after turning (reducing average traveling speed). While driving the robot uses a waypoint graph for navigation. The waypoints are typically marked by QR-codes and identified by a downwards facing camera mounted within the robot.

The process within an RMFS is a sequence of storage and retrieval trips where robots bring pods to workstations and back to the storage area (see Fig. 1.4). The main process can be described as the following procedure:

1. The robot drives to the requested pod and picks it up
2. The robot transports the pod to the requesting workstation
3. The robot queues at the workstation
4. The pod is handled at the workstation:
   - Replenishment station: New inventory is put in the pod
   - Pick station: Products are picked from the pod
5. If the pod is requested by another station, go to 2.
6. The pod is brought to a selected available storage location
7. The robot requests the next assignment; go to 1.

It is interesting to note that the storage location from which the pod was retrieved and the one it is stored at again does not necessarily match. This allows a continuous improvement of inventory placement. This means pods can be stored at locations matching their current content. For example, a pod that is needed for other customer orders present in the backlog can be stored near to the pick stations to enable shorter retrieval times. Nowadays, the workstations are typically operated by human workers, however, ongoing research supports the development of robots capable to pick from the pods (see [5]). While such picking robots may be mobile and move to storage racks by themselves, they may also be stationary and integrated at the workstations of an RMFS. The benefit over a mobile picking robot would be that the most expensive component of the resulting system, the robotic arm (including end effector and sensors), can be maximally utilized, because no time needs to be spent on moving it to the inventory. Furthermore, it is possible to combine human and robotic work stations in one system, if for example only a subset of the SKUs can be picked by the picking robot.

In the following a brief comparison regarding the advantages and drawbacks of RMFS as a parts-to-picker system is discussed. The assessment is done qualitatively taking features of other typical parts-to-picker systems (e.g., crane-, shuttle-, or grid-based systems) into account. For a comprehensive overview of automated picking systems and further discussion on the need for flexible and scalable solutions the reader may refer to [1].

Advantages:

Flexibility One of the advantages of RMFS over other parts-to-picker systems is that it can be installed very quickly. The main reason for this are the few fixed components. While other systems often involve time intense installations for high rack storage or conveyors, the most time intense installation for an RMFS are the markers for robot navigation. Furthermore, the homogeneous and independent components allow the extension of systems in operation. Vice versa, it is also possible to remove components from the system again. The latter allows the exchange of components between multiple DCs employing an RMFS or even the complete disassembly of the system in order to rebuild it at another facility. Hence, this flexibility can help the company to react to changing throughput demands more quickly and with less costs.

Adaptability Pods can be segmented into compartments of different size and even diverse pods can be handled within one system. This allows RMFS to fit a wider variety of products than most other parts-to-picker systems while still conserving a similar space utilization. In a shuttle-based solution for example the storage height is the same at least along one shuttle track. Thus, the height implied by the largest product to be stored also specifies the storage height for all other compartments the same shuttle needs to access. Furthermore, most other systems require some containing transport unit like a bin to handle small items, which is usually uniform throughout the complete system. Hence, the size of the largest inventory that can be stored in such a system is determined by the size of the container. Moreover, the pods can be adjusted to changing product shapes which reduces the risk for the company to face additional investments in order to keep the system adjusted to the demands.

Scalability In an RMFS all stations can be supplied with all pods via every robot. Furthermore, inventory is usually available on multiple pods. This means that the dependency on bottleneck resources is very low. The multiple aisles that are available for the robots to travel along support this even more. Thus, transport tasks are very independent from each other. All of this enables high scalability of the system, i.e., adding more workstations increases the throughput of the system. An indication of the scalability of RMFS is supported by the results observed in Chapter 4. The scalability of RMFS in conjunction with it's flexibility and adaptability render it a very agile parts-to-picker solution. In contrast, other parts-to-picker systems may rely on conveyors that have a maximal throughput capacity and, due to conveyor layout practicability, may not easily be scalable beyond some point in terms of throughput.

Fault tolerance As a result of the only homogeneous components and independent processes an RMFS has no single point of failure. If a robot fails, it can be re-

placed by another one without stopping the system, thus, keeping throughput up. Most crane- or shuttle-based approaches rely on conveyors connecting the aisles to the pick stations. In such systems one or more complete aisles are inaccessible, if a transport load gets stuck or a conveyor breaks down. In an RMFS a dynamic sector can be blocked and an operator safely guided there to fix it leaving the remaining system in operation.

Building requirements The building hosting the RMFS itself does only face few requirements. The system is very flat and therefore also fits quite low buildings, where other systems may require certain height to achieve an economic throughput. The typical navigation techniques do allow quite some tolerance in terms of an even floor. Other systems may require a very flat floor or custom installations to correct deviations. Furthermore, obstacles like pillars introduced by the building can be easily incorporated in the system layout.

Drawbacks:

Ergonomics No study analyzing the health effects for the pickers is available so far, but the continuous stretching and bending down to reach all compartments of the pod may lead to back trouble. The score when calculating the NIOSH method (a method aiming to quantify the risk of low-back pain, see [19]) for a picker working at a station in an RMFS is expected to be fairly poor. In contrast, other parts-to-picker systems like crane-, shuttle- and grid-based ones deliver material to the pickers in a position that is easy to reach.

Storage density While storage density is a minor issue for horizontal space utilization it is a larger issue when it comes to vertical space utilization. Since pickers need to be able to reach all compartments of a pod when it is presented at the station, an RMFS is merely two-dimensional. Hence, it is only possible to utilize vertical space in warehouses by installing mezzanine floors on which further pods are stored or complete independent systems are installed. As these mezzanine floors need to carry all the material stored in the pods and the system components itself, heavy payload floors are required, which drive the investment costs. Thus, the RMFS may not be a good fit in regions with high land costs. In contrast, other concepts like crane- and shuttle-based parts-to-picker systems can easily utilize the vertical space up to and above 20 meters.

Picking performance As a side effect of the ergonomics at the pick station the picks per hour may stay behind the picks achievable in other systems. For example, systems integrating conveyor based inventory transport can integrate so-called high speed workstations. Such workstations present the source and

destination bins right next to each other, which reduces the distance to cover by the picker and also eliminates any stretching or bending down effort. Furthermore, the bin exchange at such a station is so quick that it happens 'in the shadow' of the actual pick move.

Inefficient transports In an RMFS the throughput performance mainly relies on two factors: First, the time it takes to bring pods to and from the workstations and second, the number of products that can be picked from the pod during one presentation at the workstation (the so-called "pile-on" or "hitrate"). For the latter it is trivial to see that the less products are picked from a pod during one presentation the more pods need to be transported overall. While a low pile-on negatively impacts the overall throughput (the reader may refer to Chapter 4 for more detailed information about on pile-on), it also means that more inventory is transported unnecessarily. For example, if a pod needs to be brought to a pick station for only one product (e.g., a card game) to be picked from it, in todays RMFS it may require the transport of half a ton of material and equipment along 50 meters or more. This may lead to higher energy consumption and robot wear compared to other parts-to-picker systems which usually only transport small bins or bins. However, the negative effect is not studied and may be negligible, because only few components taking part in the transport are powered and subject to wear (essentially, only the robots).

Integratability Compared to other parts-to-picker systems the integration of an RMFS is more complicated. This is a result of the inventory 'loosely' stored in mobile racks which still requires manual put and pick operations at the workstations (or a very complex robotic solution). Hence, an RMFS does not immediately interface with conventional conveyor systems typically found in distribution centers which makes it's integration in the complete DC process more complicated.

From all listed advantages and drawbacks of the RMFS it is still challenging to draw a concise conclusion of when it is a good fit, since many different parameters from a potential scenario need to be taken into account. However, the flexibility and adaptability of the concept are the most intriguing advantages considering todays fast-paced markets. For companies most affected by these conditions the RMFS can be a method for mitigating investment risks, if the listed drawbacks are no criterion for exclusion.

1.3 Decision problems in RMFS control

Many decision problems are involved in the planning and operation of an RMFS. In Figure 1.5 an outline structured by the time-horizon is given. Mechanical and engi-

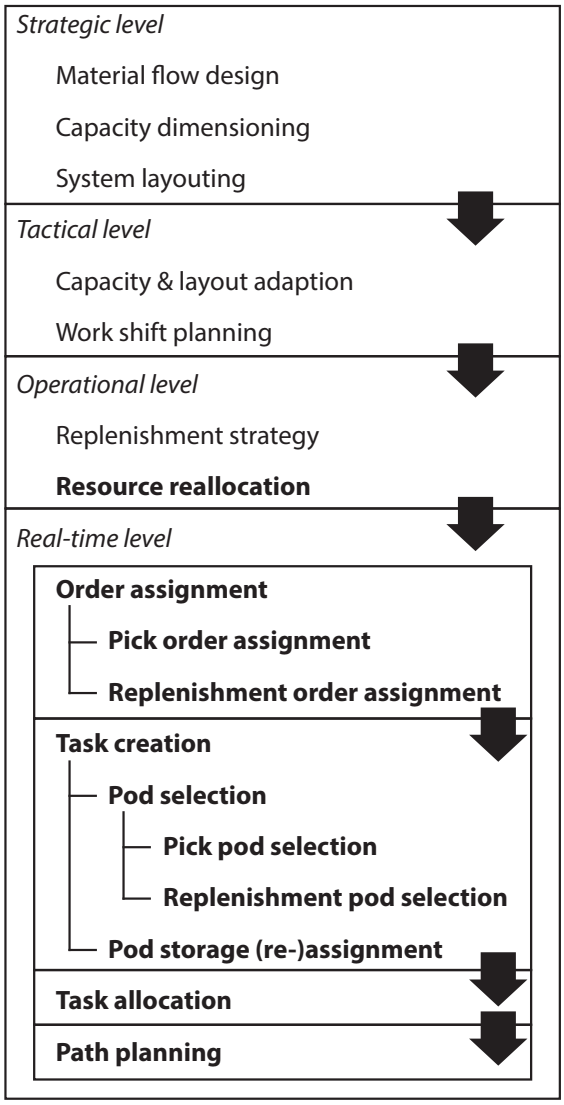


Figure 1.5 Decision problems associated with the planning and control of an RMFS (decision problems relevant to this work in bold, arrows indicate the implications of previously met decisions on subsequent stages)

neering related design and decision problems are not part of this, but are considered an input in terms of parameters like the robot's acceleration and drive speed.
On the strategic level, at first, the material flow handled by the RMFS is decided, i.e., which products need to be handled, how many of them need to be stored and how much throughput of these must be supplied. This decision needs to be aligned with all other processes of the DC. In conjunction with this the system's capacity needs to be dimensioned. I.e., how many workstations are built, how many pods are

used and how many robots are necessary. This is mainly driven by the requirements of the company using the RMFS. Capacity planning is furthermore intertwined with designing the system layout. For this, the space provided by the facility as well as the possible areas for inbound and outbound inventory flow need to be taken into account. Furthermore, the waypoint system for robot navigation, the storage locations for the pods, the workstation locations and queue designs are specified. The latter may be subject to the design of the workstation itself, if unusual configurations like two operator windows per workstation are chosen.
After the installation of the RMFS and on a more tactical level it is still possible to revise previous decisions, as a result of the system's flexibility. For example, the throughput capacity may be increased by extending the RMFS at one side, including additional workstations, pods and robots. Potentially this could even be done only temporarily for high-peak throughput periods like the Christmas season. Another possibility to readjust throughput capacity to the required level is the planning of work shifts. In an RMFS the workstations are very independent and if preceding and subsequent processes are planned accordingly, it is possible to only have one station in operation up to all stations in a 24 hour setting.
On a operational level some strategies for replenishment may be set. For example, the number of pods across which SKUs shall be distributed can be predefined. Furthermore, it can be controlled when and to what storage fill level the RMFS is replenished. Depending on the average storage time per product the chosen replenishment strategy may affect weeks of RMFS operation. A reason for temporarily choosing a different strategy may be for preparation of peak days like black Friday or the Christmas season. For these special occasions more work intense replenishment strategies that distribute the same products across more pods may be invoked so that picking processes are sped up or extra workstations can be supplied without increasing the number of robots. Moreover, it is also possible to reallocate resources of the system while it is in operation, e.g., operators working at pick stations may temporarily help out replenishment operations or other processes at the DC (see Chapter 5 for a first work on this topic).
At the lowest level the real-time decision and control problems have to be handled. This is the main focus in terms of decision problems of this work. One of these is the assignment of orders. The problem can be subdivided into *pick order assignment*, the assignment of customer orders to pick stations, and *replenishment order assignment*, whose task is to decide from which replenishment station new inventory is stored in pods. Next, these assignments imply demands for products, respectively, empty compartments to be brought to the workstations. During the creation of tasks these demands are matched by selecting pods of the system to fulfill them. *Pick pod selection* means the selection of a pod to pick one or more products from in order to fulfill customer orders at a pick station. *Replenishment pod selection* is the

selection of pods to store new products in. Both selections next demand a robot to actually transport the selected pod to the associated workstation. After the pod was handled a storage location needs to be selected to return the pod to. This selection is called *pod storage assignment*. A special case of this decision is when a pod is actively repositioned, i.e., a pod is brought from one storage location to another without bringing it to a workstation. All of these implicitly created tasks next need to be assigned to a robot. This is the result of conducting the *task allocation*. Since these tasks involve transporting a pod and/or driving empty they result in trips that need to be executed by the robots. For every trip *path planning* needs to be done to guide the robot without a collision or deadlock from it's location to the destination in a time-efficient way.
The main objective of all of the decision problems above can be stated as "using minimal resources to satisfy the target throughput and storage capacity requirements while adhering to all given constraints". Although there may be more individual requirements and objectives depending on the industry, the company or the products, since DC operations are not exactly a value-adding process and the pressure on the price per picked order line is increasingly high the demand for a low-cost RMFS will remain a core interest. Hence, the resources of an RMFS should be used as effectively as possible to allow a reduction of equipment. The effective use, i.e., achieving high throughput with a given set of equipment, is the main driver for the research done in this work.

1.4 Simulation

There are different opportunities for scientifically studying the behavior of real-world systems (see [14] and Fig. 1.6). First, it is possible to study the actual system itself or a physical model of it (e.g., clay car in wind tunnel). Both approaches are not suited for studying the control of the RMFS, since they are too expensive, too inflexible in terms of changing parameters and too slow (they cannot be artificially accelerated) in returning measurements. Hence, it is necessary to draw certain assumptions and find a mathematical representation of the RMFS. While finding an analytical solution representing the complete RMFS that provides exact answers to multiple questions arising in the control of the system is virtually impractical, a simulation model enables detailed insights by allowing a quite accurate and holistic representation of it. With a simulation model it is for example possible to observe and measure effects like congestion and track the inventory stored in the pods for layout sizes matching the ones seen in practice. Capturing all of such effects and behaviors easily overstrains the complexity of analytical solutions. Thus, although it is possible to analytically model certain decision problems and processes of an RMFS, this work approaches the RMFS concept in a very detailed and holistic

manner by using simulation. In this way the simulation model (RAWSim-O[1], see Chapter 2) introduced by this work may also serve as a validation tool for future analytical research on RMFS which uses simplified representations or models only partial processes.

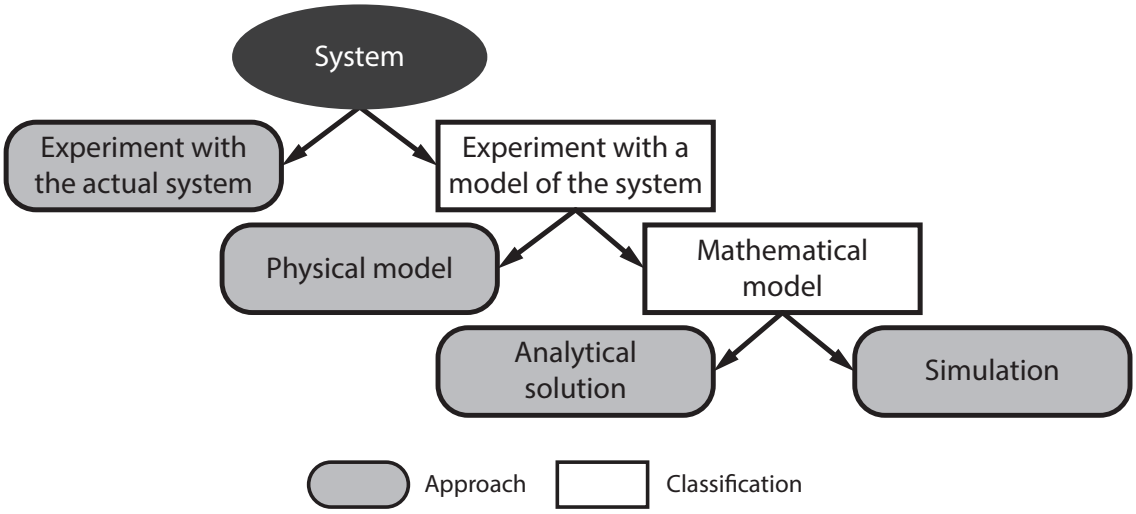


Figure 1.6 Ways to study a system (own representation based on [14])

One of the most popular methods for simulating complex systems is discrete-event simulation (DES). In a DES the state of the system is tracked over time by jumping from one discrete event to the next one until some termination criterion is met, e.g., the simulation duration was reached or all processes of interest stopped. At each of these events the complete state of the system is realized, i.e., all potential changes to the variables from thee former event to the current one are checked and committed. Additionally, two types of time-advance mechanisms are distinguished: the *fixed-increment time advance* and the *next-event time advance* (see [14]). The former uses the same time-delay when updating from one event to the next one. This is a time-advance routine which is rarely used, since it may increase execution time by introducing unnecessary events to be simulated while potentially causing some measurement error by conducting state changes at the wrong times. In contrast, the latter mechanism simulates all events at which a state change can happen, but also not any further event.
Another special type of DES is an agent-based simulation (ABS). In an ABS entities like robots are agents that update their state according to the simulated events, expose functionality of the corresponding entity (e.g.: movement behavior) and calculate their individual next event. Therefore, the time advance mechanism in an ABS always skips to the earliest next event of all agents of the simulation. During

[1]source code available at https://github.com/merschformann/RAWSim-O

each event the agents may also adjust their next event. While using agents to constitute physical entities seems more obvious, controllers or statistical tracking entities can be seen as agents of the same simulation framework too. This divide and conquer approach also makes the implementation of a simulation more modular and maintainable. As an example, a statistical agent tracking the positions of all robots over time in order to supply a heatmap of the movement behavior after the simulation finished could take a snapshot of the robot locations every 60 seconds. At each of the events introduced by this statistical agent all robot agents need to update their position accordingly which allows the retrieval of the exact positions of the robots, even though the driving part does not incur further events before completion. This clearly requires the robot agents to be updated before the statistical agent is, hence, it is noteworthy that the sequence in which agents in an ABS are updated is not necessarily arbitrary.

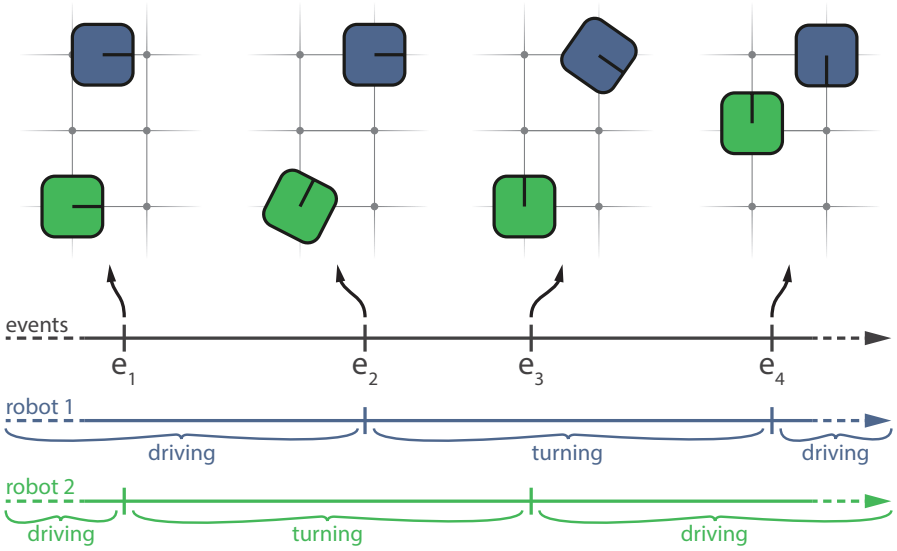


Figure 1.7 Example of events occurring for two moving robots (top-line: all events, mid- and bottom-line: events and movement states of robot 1 & 2; the situation at all four implied events is visualized at the top)

In Figure 1.7 an example of the next-event time advance technique in an agent-based setting is given. In the example two robots drive along a predefined path. The first robot's (blue) path includes a right-turn at the top-right node of the graph (indicated by the gray lines) while the second robot does a left-turn at the bottom-left node before continuing to drive. The first event marks the second robot's movement type changing from driving to turning. During the state change the robot agent calculates the time for turning by 90 degrees and marks the time as it's next event. During

the same event the position of the first robot is updated to the one it would be located at at the given event time. This way the complete simulation world is set to the state of the given event. Next, the simulation skips to the earliest next event marked by the first robot. This time it is the other way around: the first robot stops and starts turning, while the second robot's position is adjusted. This process is repeated until the simulation stops. The reader may note that another logical agent keeping track of the robots' trajectories and potential collisions is needed to sense robots driving through each other for situations where no event occurs during the drive. If such an agent is not given, the simulation would return incorrect results.

1.5 Research gap and scope

Although the RMFS is in operation at many facilities world-wide for approximately a decade now, scientific research is still very limited. In contrast, research on other automated warehouse systems has been thriving for several decades. Many of the problems faced in other systems share similarities with the ones arising in RMFS control, but a successful adaption of the methods is unclear. However, this section draws an outline of related work, but limited to work focusing RMFS directly. For further related work the reader may refer to the corresponding sections of the included papers themselves. For the sake of clarity related work is divided in three categories, i.e., research on path planning in RMFS, research on decision support for the design and control of RMFS and a last short note on detailed simulations for RMFS.

Path planning for RMFS Multi-agent pathfinding is a well studied problem with applications from video games to robotics. In this field research exists that lies a foundation for methods applicable to RMFS. The works of [4], [15] and [3] provide first results and insights about algorithms applied to MAPF instances with RMFS layout characteristics. However, physical properties like momentum (leading to finite acceleration / deceleration values) and that a robot can usually only drive in one direction (requiring adherence to rotational speeds) are not considered in the classical MAPF problem. To overcome this, [11] propose post processing of solutions obtained by state-of-the-art MAPF algorithms. This thesis takes a different approach by incorporating the kinematic constraints in the problem formulation and adapting the methods to the new model (see Chapter 3).

Decision support research for RMFS In a paper by the inventors of the RMFS (see [6] & [21]), the authors describe a number of operational problems they encountered in practice. One of the very rare works on these problems is done by [2]. The authors propose methods for optimally aligning the sequences of pick orders and

pods at one station such that the number of products picked from each pod is maximized. Thus, less pods need to be moved to fulfill the pick orders. The authors observe a significant improvement when compared to a random assignment. However, the studied system sizes are minimalistic and the scalability of the approach when also incorporating multiple stations accessing the same pods is uncertain.
Further research on RMFS bases on queueing networks aiming to provide analytical answers to questions emerging for tactical decisions in design and control of RMFS. [12] create a queueing network for single- and multi-line orders, with and without zoning in the storage area, that captures only the pick operations, but that does include robot movement behavior. Their model can quickly provide valuable estimates about the expected order cycle time, workstation utilization and robot utilization. The model is used to determine how the storage area dimensions and the workstation placement around the storage area affect the maximum order throughput, by evaluating a large number of possible designs. The work of [13] uses a similar approach that addresses control decisions on a tactical level. The authors show the effect of the number of pods per SKU and of the replenishment level of a pod on order throughput, and they show what the optimal ratio of the number of pick stations to the number of replenishment stations is. In [22] semi-open queueing networks are used to analyze the policy for assigning robots to pick stations. The results show that the random policy is significantly outperformed by the proposed handling-speeds-based assignment rule when facing varying service rates of the pickers. The authors of [23] build a semi-open queueing network for evaluating the effects of battery management in RMFS. The strategies of battery swapping, automated plug-in charging and inductive charging at the pick station are compared. The authors come to the conclusion that battery swapping is generally more expensive than plug-in charging while inductive charging outperforms both in throughput and costs, if robot prices and retrieval times are low. Further descriptions of above work can be found in Chapter 4 & 5.

Simulations for RMFS The foundation for the simulation framework described in this thesis is done by [9]. The authors introduce a DES called Alphabet Soup. The presented simulation incorporates robot agents carrying buckets in which colored letters are stored. The colored letters are then brought to and picked at stations to complete colored words. The rationale behind this is to capture the basic behavior of an RMFS. However, certain movement characteristics of the robots, the structure of customer orders & inventory, the order release process and more are inaccurate for a realistic warehouse emulation and have been addressed in the simulation framework introduced by this work (see Chapter 2).

As research on RMFS is sparse in general and virtually non-existent on real-time control related aspects (see Chapter 1.3 for an overview), this work focuses specifically on resulting research gap. This also means that the given system design and layout is considered 'frozen' in all aspects not control related, e.g., improvements of the layout like the graph structure or how inventory is replenished. This work focuses on the study of control mechanisms while considering many real-world features to allow detailed and holistic insights about RMFS. Therefore, the main research question can be outlined as follows:

What is an efficient approach for controlling a Robotic Mobile Fulfillment System in order to maximize order throughput and resource utility?

In order to answer this question the following subgoals are addressed:

1. Devise a realistic simulation framework for the study of RMFS
2. Develop algorithms capable of controlling mobile robots of the RMFS
3. Analyze effects of control strategies for all core decision problems in RMFS
4. Study control strategies for an RMFS underlying changing conditions
5. Analyze special RMFS features introducing novel control opportunities

1.6 Paper summaries & contributions

Summaries of all included papers on the topic of RMFS follow in the order they appear.

Chapter 2: RAWSim-O: A Simulation Framework for Robotic Mobile Fulfillment Systems

This paper introduces RAWSim-O - a simulation framework for RMFS. The simulation framework allows the study of the most prominent decision problems faced when operating an RMFS. For this, an agent-based discrete-event simulation (DES) was developed and made available online for other researchers to use. The framework allows the modular integration and combination of control mechanisms to study the system performance and behavior effected by them. Furthermore, the detailed nature of the framework was used to conduct a small application experiment using vacuum cleaning robots to confirm the applicability of algorithms to the real-world.
Main contributions:

- A publicly available realistic simulation framework supporting RMFS research
- A real-world application for the demonstration of RMFS

Chapter 3: Multi-Agent Path Finding with Kinematic Constraints for Robotic Mobile Fulfillment Systems

A collection of path planning algorithms is studied in this paper. A reformulation of the multi-agent pathfinding problem is introduced, which extends classic MAPF to a formulation integrating RMFS path planning characteristics. A set of state-of-the-art algorithms from the field of MAPF is revised and adapted to the new formulation. Furthermore, the performance of these algorithms is compared using the RAWSim-O simulation framework.
Main contributions:

- A novel mathematical formulation for path planning integrating kinematic constraints suitable for RMFS
- Five adapted path planning algorithms for solving the problem
- A benchmark study providing insights about the performance of the algorithms

Chapter 4: Decision rules for Robotic Mobile Fulfillment Systems

In this work, strategies for multiple operational decision problems are studied, namely, pick order assignment, replenishment order assignment, pick pod selection, replenishment pod selection and pod storage assignment. For each decision problem a set of decision rules is introduced. All of the possible rule combinations for the control of RMFS are then analyzed in a large simulation study using the RAWSim-O framework.
Main contributions:

- A definition of the core RMFS control decision problems
- A number of decision rules suitable for controlling an RMFS
- A large experiment providing insight about the performance of the studied rules and importance of the decision problems

Chapter 5: Optimal policies for resource reallocation in a Robotic Mobile Fulfillment System

The work focuses on RMFS facing time-varying demand and resource reallocation policies to mitigate negative effects of the fluctuations. A queuing network modeling the RMFS is integrated with a Markov Decision Process (MDP) aiming to allocate robots to pick and replenishment operations according to current workloads. The derived policies are compared with benchmark

policies from practice.
Main contributions:

- A novel integration of a queuing network modeling RMFS with a resource allocating MDP
- Insights about effective control policies for RMFS underlying fluctuating demand
- Result validation via a detailed RMFS simulation model

Chapter 6: Active repositioning of storage units in Robotic Mobile Fulfillment Systems
This paper studies the effects of active storage unit repositioning done in RMFS. Two techniques (active and passive repositioning) are distinguished and their effects studied by applying mechanisms which exploit them. Results obtained using the RAWSim-O simulation framework suggest a positive effect of preparing storage unit positions, if customer orders are known a reasonable time in advance.
Main contributions:

- Methods for active repositioning in RMFS
- A metric for determining the 'well-sortedness' of inventory
- An analysis of whether active repositioning may contribute to the overall throughput for different scenarios

1.7 Conclusion

This work addressed the efficient control of the Robotic Mobile Fulfillment System. The conducted research focuses on a realistic and holistic view of the RMFS. Therefore, a detailed simulation framework called RAWSim-O was introduced, which comprises features like physical momentum of the robots and realistic order process emulation. Moreover, the framework including the source-code is made available to support other researchers in the field by eliminating the need for developing a complete simulation framework in order to analyze new control algorithms and strategies. To enable this the framework is kept modular. Furthermore, algorithms from the multi agent pathfinding field of research are extended to respect kinematic constraints so that these can be used for the control of the robots in an RMFS. Moreover, the proposed algorithms may have applications in similar environments where simple robots navigate by using navigation graphs with the same characteristics. Next, decision rules for multiple decision problems occurring in the control of RMFS have been analyzed in a large two-staged experiment to give first insights

about the relevance of each one of them and first successful strategies to apply for practitioners. The proposed decision mechanisms scale well and can be applied for systems of practical size. Moreover, an RMFS queuing model integrated with an MDP for resource reallocation is proposed for RMFS underlying fluctuating demands. The derived control policies perform well when compared to benchmark policies from pratice and the results are validated using RAWSim-O. At last, the opportunity of re-sorting inventory during RMFS operation is studied in a first experiment. The results reveal potential for decision mechanisms taking current order backlog situations into account in order to improve inventory 'well-sortedness' accordingly. Moreover, formulas for quantifying the current 'well-sortedness' of the inventory are proposed.
Current research on RMFS is still very limited, although an increased application of the concept can be observed in practice. The RMFS combines a unique set of decision problems like storage assignment and path planning in one self-contained system. Thus, there is high potential for further methods from the fields of operations research and artificial intelligence to be successfully applied to it. With the available RAWSim-O framework a direct comparison of new methods against methods proposed by this work can be done more easily.

References

[1] Kaveh Azadeh, M. B. M. de Koster, and Debjit Roy. "Robotized Warehouse Systems: Developments and Research Opportunities". In: *SSRN Electronic Journal* (2017). ISSN: 1556-5068. DOI: 10.2139/ssrn.2977779.

[2] Nils Boysen, Dirk Briskorn, and Simon Emde. "Parts-to-picker based order processing in a rack-moving mobile robots environment". In: *European Journal of Operational Research* 262.2 (2017), pp. 550–562.

[3] L. Cohen et al. "Rapid Randomized Restarts for Multi-Agent Path Finding Solvers". In: *ArXiv e-prints* (2017).

[4] Liron Cohen, Tansel Uras, and Sven Koenig. "Feasibility Study: Using Highways for Bounded-Suboptimal Multi-Agent Path Finding". In: *Eighth Annual Symposium on Combinatorial Search*. 2015.

[5] Nikolaus Correll et al. "Analysis and Observations From the First Amazon Picking Challenge". In: *IEEE Transactions on Automation Science and Engineering* 15.1 (2018), pp. 172–188. ISSN: 1545-5955. DOI: 10.1109/TASE.2016.2600527.

[6] J. J. Enright and P. R. Wurman. "Optimization and Coordinated Autonomy in Mobile Fulfillment Systems". In: *Working paper* (2011).

[7] Eurostat. *Indizes von Arbeitskosten nach NACE Rev. 2*. 2018. URL: http://appsso.eurostat.ec.europa.eu/nui/show.do?dataset=lc_lci_r2_q&lang=de (visited on 05/24/2018).

[8] Eurostat. *Internet-Käufe durch Einzelpersonen*. 2018. URL: http://appsso.eurostat.ec.europa.eu/nui/show.do?dataset=isoc_ec_ibuy&lang=de (visited on 05/24/2018).

[9] Christopher J. Hazard, Peter R. Wurman, and Raffaello D'Andrea. "Alphabet Soup: A Testbed for Studying Resource Allocation in Multi-vehicle Systems". In: *Proceedings of AAAI Workshop on Auction Mechanisms for Robot Coordination*. Citeseer, 2006, pp. 23–30.

[10] A. E. Hoffman et al. "System and method for inventory management using mobile drive units". US20130103552 A1. 2013. URL: https://www.google.com/patents/US20130103552.

[11] Wolfgang Hönig et al. "Multi-Agent Path Finding with Kinematic Constraints". In: *ICAPS*. 2016, pp. 477–485.

[12] T. Lamballais, D. Roy, and M. B. M. de Koster. "Estimating Performance in a Robotic Mobile Fulfillment System". In: *European Journal of Operations Research* 256 (2017), pp. 976–990.

[13] T. Lamballais, D. Roy, and M. B. M. de Koster. "Inventory Allocation in Robotic Mobile Fulfillment Systems". Working Paper, available at SSRN. 2017.

[14] Averill M. Law. *Simulation modeling and analysis*. Fifth edition, international edition. McGraw-Hill Education. 2015. ISBN: 978-1-259-01071-2.

[15] Hang Ma et al. "Overview: Generalizations of Multi-Agent Path Finding to Real-World Scenarios". In: *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI) Workshop on Multi-Agent Path Finding*. 2016.

[16] Nick Wingfield. "As Amazon Pushes Forward With Robots, Workers Find New Roles". In: *The New York Times* (2017). URL: https://nyti.ms/2xUhVgM.

[17] Ash Sharma. *Mobile Robots in Logistics Centers Forecast to Surpass $3bn in 2022*. 2018. URL: https://www.interactanalysis.com/logistics-robots-forecast-growth/.

[18] James A. Tompkins. *Facilities planning*. 4th ed. Hoboken, NJ and Chichester: John Wiley & Sons, 2010. ISBN: 0470444045.

[19] T. R. Waters et al. "Revised NIOSH equation for the design and evaluation of manual lifting tasks". In: *Ergonomics* 36.7 (1993), pp. 749–776. ISSN: 0014-0139. DOI: 10.1080/00140139308967940.

[20] Marc Wulfraat. *Is Kiva Systems a Good Fit for Your Distribution Center? An Unbiased Distribution Consultant Evaluation.* 2012. URL: http://www.mwpvl.com/html/kiva_systems.html (visited on 01/14/2018).

[21] P. R. Wurman, R. D'Andrea, and M. Mountz. "Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses". In: *AI Magazine* 29.1 (2008), pp. 9–19.

[22] Bipan Zou et al. "Assignment rules in robotic mobile fulfilment systems for online retailers". In: *International Journal of Production Research* 55.20 (2017), pp. 6175–6192.

[23] Bipan Zou et al. "Evaluating battery charging and swapping strategies in a robotic mobile fulfillment system". In: *European Journal of Operational Research* 267.2 (2018), pp. 733–753.

Chapter 2

RAWSim-O: A Simulation Framework for Robotic Mobile Fulfillment Systems

Marius Merschformann[1] Lin Xie[2] Hanyi Li[3]

[1] *University of Paderborn, Paderborn, Germany*
[2] *Leuphana University of Lüneburg, Lüneburg, Germany*
[3] *Beijing HANNING ZN Tech Co.,Ltd, Beijing, China*
marius.merschformann@uni-paderborn.de, lin.xie@leuphana.de

Logistics Research (2018), Volume 11, Issue 1

Abstract

This paper deals with a new type of warehousing system, Robotic Mobile Fulfillment Systems (RMFS). In such systems, robots are sent to carry storage units, so-called "pods," from the inventory and bring them to human operators working at stations. At the stations, the items are picked according to customers' orders. There exist new decision problems in such systems, for example, the reallocation of pods after their visits at work stations or the selection of pods to fulfill orders. In order to analyze decision strategies for these decision problems and relations between them, we develop a simulation framework called "RAWSim-O" in this paper. Moreover, we show a real-world application of our simulation framework by integrating simple robot prototypes based on vacuum cleaning robots.

2.1 Introduction

Due to the rise of e-commerce, the traditional manual picker-to-parts warehousing systems no longer work efficiently, and new types of warehousing systems are required, such as automated parts-to-picker systems. For details about the classification of different warehousing systems we refer to [8]. This paper studies one of the parts-to-picker systems, a so-called Robotic Mobile Fulfillment Systems (RMFS), such as the Kiva System ([4], nowadays Amazon Robotics), the GreyOrange Butler or the Swisslog CarryPick. The approach of an RMFS completely eliminates the need for travel within the warehouse, which accounts for approximately 50 % of a picker's time in manual warehouse operations according to [14]. [15] indicates that the Kiva System increases the productivity two to three times, in contrast to a traditional manual picker-to-parts system. Compared to other kinds of warehousing systems, the biggest advantages of an RMFS are its flexibility as a result of having virtually no fixed installations, the scalability due to accessing the inventory in a parallel manner, and the reliability due to the use of only homogeneous components, i.e., redundant components may compensate for faulty ones (see [5] and [16]).
The first framework, "Alphabet Soup," was published by [5], and is a first simulation of the RMFS concept. In this work, we extend the work of [5] to develop our simulation framework, called "RAWSim-O" (Robotic Automatic Warehouse Simulation (for) Optimization). Similarly to "Alphabet Soup," we use an agent-based and event-driven simulation focusing on a detailed view of the system, but extend our simulation framework with the cases of multiple floors, which are connected by elevators. The reason for this extension is to avoid the lack of utilization of vertical space in an RMFS compared to other parts-to-picker systems, such as shuttle-based solutions (see [13]). Moreover, we integrate a more realistic robot movement emulation by considering the robot's turning time and adjusted acceleration or deceleration formulas. This enables a real world application of our simulation framework by using simple robot prototypes based on vacuum cleaning robots (iRobot Create 2). The implementation of the framework was done in C# and is compatible with the Mono project to allow execution on high throughput clusters. All source codes are available at https://github.com/merschformann/RAWSim-O.
There exist many decision problems in an RMFS, and they influence each other. With the publication of RAWSim-O we provide a tool for analyzing effects of decision mechanisms for these problems and synergies of decision strategies. The framework enables a holistic look at RMFS which helps uncovering side-effects of strategies, e.g. deciding tasks for robot that cause congestion issues for path planning methods. Hence, RAWSim-O enables a more reliable assessment of the system's overall efficiency, e.g., in terms of customer order throughput. Thus, we hope to further support and push the research on RMFS with our work. We describe the RMFS in more detail and point out the decision problems in focus in Section 2.2. Next, our

simulation framework is described in Section 2.3. After that, we show a demonstration application of our simulation in Section 2.4. Finnaly, we conclude our work in Section 2.5.

2.2 The Robotic Mobile Fulfillment System

Instead of using a system of shelves and conveyors as in traditional parts-to-picker warehouses, the central components of an RMFS are:

- movable shelves, called *pods*, on which inventory is stored
- *storage locations* denoting the inventory area where the pods are stored
- workstations, where the pick order items are picked from pods (*pick stations*) or replenishment orders are stored to pods (*replenishment stations*)
- mobile *robots*, which can move underneath pods and carry them to workstations.

The pods are transported by robots between the inventory area and workstations. Figure 2.1 shows the storage and retrieval process: after the arrival of a replenishment order (consisting of a number of physical units of one stock keeping unit (SKU)), robots carry selected pods to a replenishment station to store units in pods. Similarly, after receiving a pick order (including a set of order lines, each for one SKU, with corresponding units necessary to fulfill the line), robots carry selected pods to a pick station, where the units for the order lines are picked. Note that, in order to fulfill pick orders, several pods may be needed, since orders may have multiple lines. Although pods typically contain multiple SKUs with many SKUs in stored in the system it is very unlikely that a pick order can be completed with only one pod. Then, after a pod has been processed at one or more stations, it is brought back to a storage location in the inventory area.

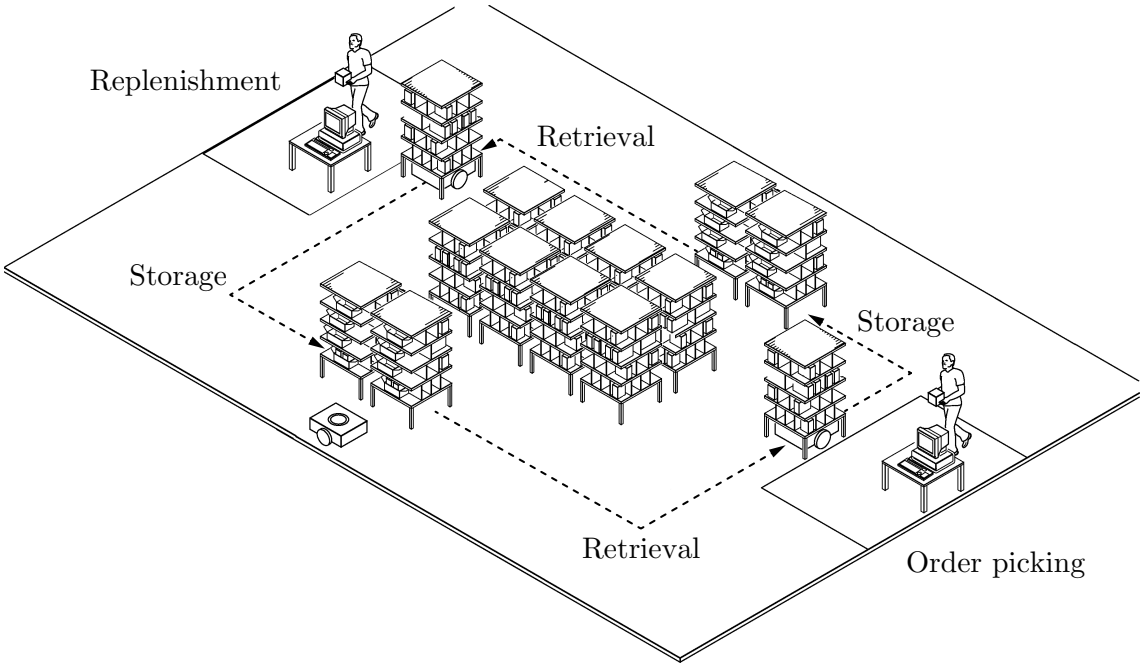


Figure 2.1 The central process of an RMFS (see [6])

2.2.1 Decision Problems

In an RMFS environment, various optimization and allocation problems have to be solved in real time. The system aims at keeping human workers at the stations busy while minimizing the resources (e.g. robots) to fulfill the incoming pick orders. These problems were first described by [16]:

- For the bots, the planning of their tasks, paths, and motions
- For the stations, the management of the workflow for each station
- For the resources of the system, their planning, provisioning, and allocation to other components of the system, usually called the Resource Allocation Problem

As [16] note, the Resource Allocation Problem cannot be treated as a global optimization problem, but rather as a set of subproblems that should be solved using specialized methods to enable the decisions in an online e-commerce environment. We reiterate these decision problems briefly in the context of the simulation framework developed in this work:

- *Order Assignment*

  - **Replenishment Order Assignment (ROA)**: assignment of replenishment orders to replenishment stations
  - **Pick Order Assignment (POA)**: assignment of pick orders to pick stations

- *Task Creation*
  - *Pod Selection*
    - **Replenishment Pod Selection (RPS)**: selection of the pod to store one replenishment order in
    - **Pick Pod Selection (PPS)**: selection of the pods to use for picking the pick orders assigned at a pick station
  - **Pod Storage Assignment (PSA)**: assignment of an available storage location to a pod that needs to be brought back to the inventory area

- **Task Allocation (TA)**: assignment of tasks from *Task Creation* and additional support tasks like idling to robots

- **Path Planning (PP)**: planning of the paths for the robots to execute

- **Station Activation (SA)**: controlling active times of the stations (e.g. switching some off based on work shifts or for emulation of "jumper" pickers that can be assigned to allow for temporarily increased throughput)

- **Method Management (MM)**: exchange of controlling mechanisms in a running system (e.g. replacing the PSA controller with a different one during runtime in order to adapt to changing conditions)

The decisions for the aforementioned operational problems influence each other. Here we sketch two relationships between decision problems that may exploit synergy effects or sabotage each others success:

- POA and PPS: one objective is to maximize the average number of picks per handled pod (called pile-on); in other words, the higher the pile-on is, the fewer pods are needed at pick stations. Therefore, the selection of a pod for an order in PPS is dependent on the selection of the pick station of other orders. If similar orders are assigned to the same pick station, a pod can be selected to maximize the number of picks.

- PSA, TA and PP: One objective is to minimize the travel times of robots to complete all orders, thus, reducing the number of robots needed to achieve a certain throughput. For this, the selected storage locations for pods impact the performance of the assignment of tasks and the pathfinding, because the trip destinations change.

Some of the operational decision problems described above have been addressed in the context of RMFS in previous publications. First, the sequencing of pick orders and pick pods for an integration of PPS and POA is studied in [1]. The authors propose methods for obtaining sequences minimizing the number of pod visits at one station. Second, the planning of paths in an RMFS is subject to the studies in [3], [2], and [10]. The state-of-the-art multi-agent path planning algorithms were implemented in [10] to suit PP in an RMFS. Besides the operational problems, we would like to note that more literature exists on more tactical to strategic decision problems, like layout planning, for example, in [9].

Furthermore, many of the problems discussed above share similarities with decision problems well-studied in the context of other warehouse systems or in theory itself. However, the successful application of policies or algorithms of other applications to an RMFS often is unclear. For an overview of literature about decision problems occurring when controlling other Automated Storage & Retrieval Systems' processes, we refer to [12]. The authors provide an overview about literature on decision problems, such as storage assignment (sharing similarities with RPS and PSA) and order batching (sharing similarities with POA). Another decision problem related to one occurring in RMFS is multi robot task allocation (sharing similarities with TA). Similar to TA in RMFS, a set of homogeneous robots needs to be matched with a set of tasks to maximize the overall system performance. For an overview of literature on the subject we refer to [7]. Furthermore, the problem of multi agent path planning (sometimes referred to as route planning or pathfinding) is a subject of research in multiple applications and theories. The studied formulations often differ in the way whether kinematic constraints and further additional constraints from the real-world applications are modeled. We refer to [11] for a detailed explanation of the problem and an overview of existing literature.

2.3 RAWSim-O

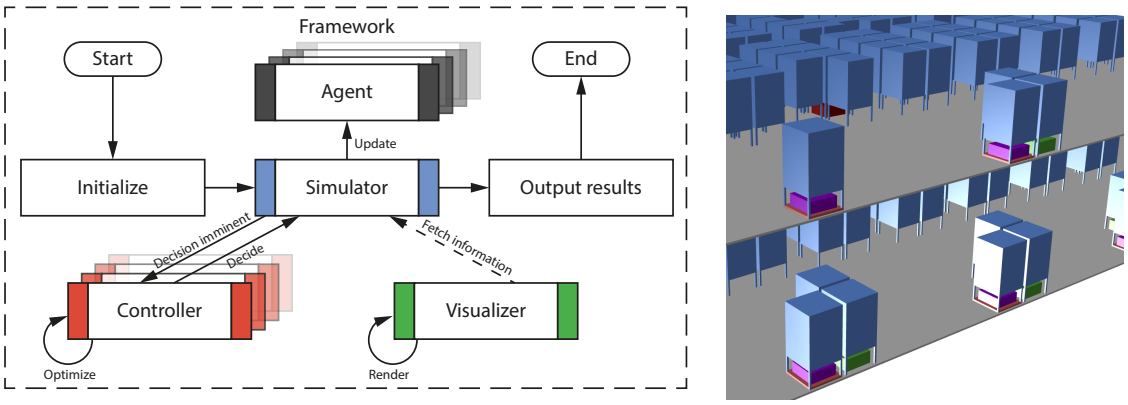
RAWSim-O is an agent-based discrete-event simulation framework. It is designed to study the context of an RMFS while considering kinematic behavior of the mobile robots and evaluating multiple decision problems jointly. Hence, the main focus of the framework is the assessment of new control strategies for RMFS and their mutual effects. In the following, we describe the RAWSim-O simulation framework in more detail. At first, we describe the general structure of the framework, and then we give detailed information about how the robotic movement is emulated.

2.3.1 Simulation Process

Figure 2.2a shows an overview of our simulation process, which is managed by the core *simulator* instance. The tasks of the simulator contain:

- Updating *agents*, which can resemble either real entities, such as robots and stations, or virtual entities like managers, e.g. for emulating order processes.
- Passing decisions to *controllers*, which can either decide immediately or buffer multiple requests and release the decision later.
- Exposing information to a *visualizer*, which allows optional visual feedback in 2D or 3D. Figure 2.2b illustrates a screenshot of our simulation in 3D.

As mentioned before, RMFS usually lack utilization of vertical space when compared to other parts-to-picker systems, for example shuttle-based solutions (see [13]). Although the flexibility of the system due to the mobile components is an advantage over the many fixed components of a shuttle-based solution, in areas where land costs are high the lack of vertical space utilization can be a deal-breaker for the RMFS concept. To help mitigate this disadvantage RAWSim-O allows the study of multi-floor systems for applications where the integration of mezzanine floors is possible (see Fig. 2.2b for example). The floors are connected by elevators that can be used by one robot at a time. One elevator connects at least two waypoints of the underlying waypoint-system, which is used for guiding the robots. For each pair of connected waypoints a constant time is specified to capture the travel time of the lift transporting a robot.
Conceptually, the waypoint system itself is a directed graph which robots travel along while adhering to their kinematic behavior. I.e., turning is only allowed on-the-spot at waypoints (essentially nodes of the graph) and is executed by a specified angular speed. Furthermore, straight travel along the edges of the graph involves acceleration and deceleration, which is discussed in more detail in Sec. 2.3.2. To avoid blocked destination waypoints for path planning (many robots share the same destination when approaching stations or elevators) waypoints may form a queuing zone. After a robot reaches such a zone guided by a path planning engine a queue manager takes over handling of the robots paths within the zone. For this, the manager may make use of shortcuts definable within the queue, but mainly lets the robots move up until they reach the end waypoint (e.g., a pick station). On leaving the queuing zone again the defined path planning engine takes over again.
The framework allows easy exchange of controllers by implementing a base structure for all mentioned decision problems. For this, all controllers are also agents of the simulation that may expose a next event to jump to, and also get updated at each event. This enables the mechanisms to react to every change occurring in the simulated world as well as to steer it. As a shortcut, controllers may also subscribe



(a) Overview of the simulation process. **(b)** Visualization screenshot

Figure 2.2 RAWSim-O simulation framework

to events that fire, if certain situations occur.
Additionally, the controllers are called whenever a new decision needs to be made, e.g. the POA controller is called when a new pick order is submitted to the system, and the TA controller is called when a robot finished its task. In Tab. 2.1 we include a short outline of the calls to the event-driven decision controllers. The SA and MM controllers are not event-driven, because they are considered like management mechanisms. However, all controllers may subscribe to simulation-wide events.

Table 2.1 Overview of the base events causing calls to the controller per decision problem and the corresponding assignment responsibility

Problem	Decision	Main triggers
ROA	repl. order → repl. station	New repl. order, repl. order stored in pod
POA	pick order → pick station	New pick order, pick order completed, repl. order stored in pod
RPS	repl. order → pod	New repl. order, SKU unit picked
PPS	pod → pick order(s)	New task is assigned to robot
PSA	pod → storage location	Pod needs to be brought back to inventory
TA	task → robot	Robot needs a new task
PP	robot → robot	Robot has a new destination

In addition to acting ad-hoc according to the triggers mentioned in Tab. 2.1 strategies may plan ahead, e.g., by sequencing tasks for robots instead of greedily selecting them as soon as a robot is done with its previous one. For this example it can be done by preparing tasks ahead and assigning them to the chosen robot as soon as it becomes ready. Furthermore, the framework also allows some controllers to run optimization algorithms in parallel. This means that the controllers are allowed to buffer certain decisions (like the assignment of pick orders), run an optimization procedure in parallel, and submit the decision later on. In this case the simulation is paced until the optimization algorithm returns a solution in order to synchronize the wall time of the algorithm with the simulation time. I.e., the simulation will only continue for the wall time that already passed for the optimization algorithm converted to simulation time (while it is still running), but not beyond it.

2.3.1.1 Core Decision Hierarchy

In the following, we describe the hierarchy of all core decision problems after new replenishment or pick orders are submitted to the system (see Figure 2.3). For this, the SA and MM decision problems are neglected, since they have a more supportive role and can even be replaced with default mechanisms that keep the status quo. If a new replenishment order is received, first the controllers of ROA and RPS are responsible for choosing a replenishment station and a pod. This technically results in an insertion request, i.e. a request for a robot to bring the selected pod to the given workstation. A number of these requests are then combined in an insertion task and assigned to a robot by a TA controller. Similarly, after the POA controller selects a pick order from the backlog and assigns it to a pick station, an extraction request is generated, i.e. a request to bring a suitable pod to the chosen station. Up to this point, the physical units of SKUs for fulfilling the pick order are not yet chosen. Instead, the decision is postponed and taken just before PPS combines different requests into extraction tasks and TA assigns these tasks to robots. This allows the implemented controllers to exploit more information when choosing a pod for picking. Hence, in this work we consider PPS as a decision closely interlinked with TA. Furthermore, the system generates store requests each time a pod is required to be transported to a storage location, and the PSA controller decides the storage location for that pod. The idle robots are located at dwelling points, which are located in the middle of the storage area to avoid blocking prominent storage locations next to the stations. Another possible type of task is charging, if the battery of a robot runs low; however, for this work we assume the battery capacity to be infinite. All of the tasks result in trips, which are planned by a PP algorithm. The only exception is when a pod can be used for another task at the same station, thus, not requiring the robot to move.

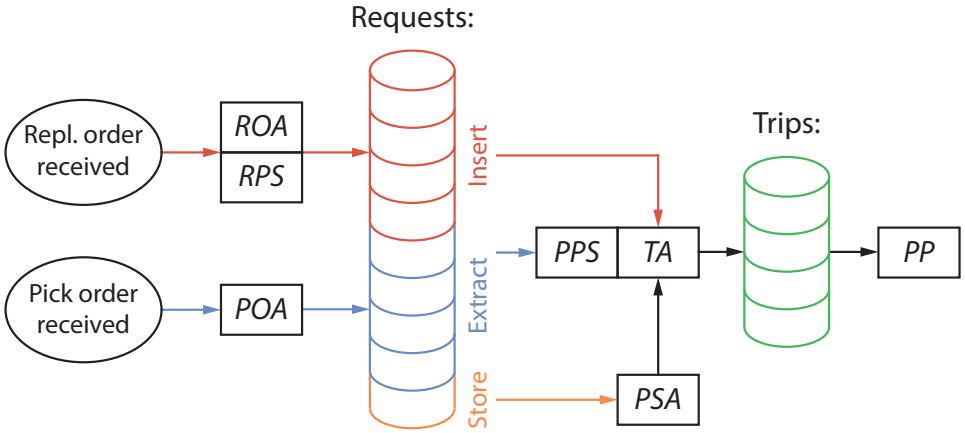


Figure 2.3 Order of decisions to be done triggered by receiving a pick or replenishment order

2.3.1.2 Input Information

The simulation framework conceptually consists of three different inputs. First, a layout configuration specifies the characteristics and dimensions of the system layout itself. Second, a scenario configuration describes how orders are generated and further settings of the system's surroundings. Finally, a controller configuration is given to specify the decision mechanisms for all previously described decision problems. We distinguish them as three different input files to enable easier assessment of control methods for different systems under diverse scenarios.

Layout Specification The layout specification can be either an explicit file specifying the exact positions and individual characteristics for all stations, the waypoint system, and the robots; or a file providing specifications leading to a default layout based on the concepts of [9]. This default layout can be seen in Fig. 2.4, with pick stations as red circles, replenishment stations as yellow circles, robots in green, and the pod storage locations in the middle as blue squares. It is based on the idea of circular flows around storage location blocks that also align with the entrances and exits of the stations and their queuing areas (dotted lines). Between the queuing area and the storage locations, a hallway area (dashed lines) allows the robots to cross between the stations. In order to generate such a layout, the numbers of pick and replenishment stations on the north, south, east, and west sides need to be set. Furthermore, the numbers of vertical and horizontal aisles and the block size need to be specified.

Scenario Specification The next input file specifies information about the scenario to simulate. This includes the duration of the simulation and settings about

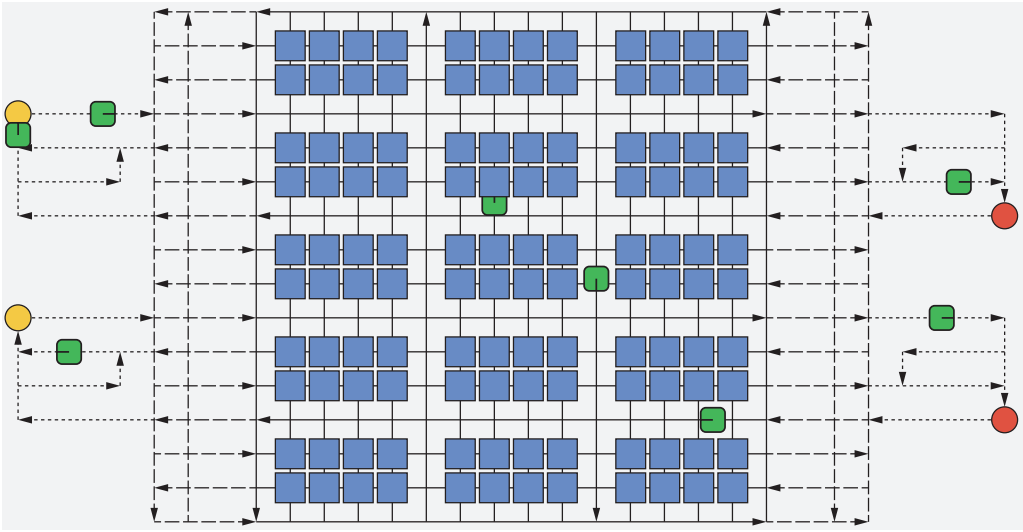


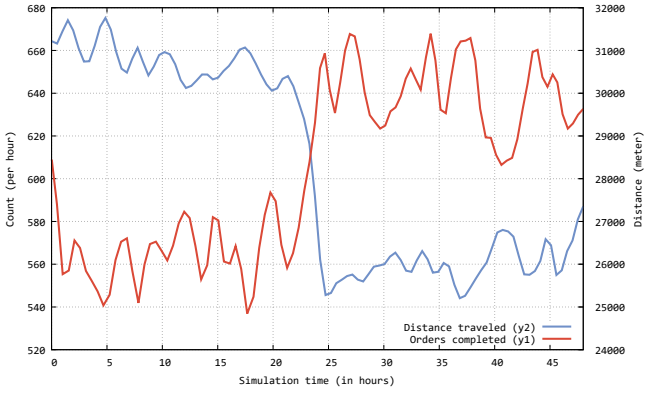
Figure 2.4 The default layout, with two replenishment and two pick stations

the inventory and order emulation. In contrast with the concept of colored word order emulation of "Alphabet Soup" in [5], we use a concept for SKU and pick order generation based on typical random distributions. For the generation of SKU popularity information, we implemented constant, uniform, normal, and gamma distributions to emulate simple scenarios up to more ABC-like popularity curves. The SKU for each pick order line is then selected using the chosen distribution, but limiting the choice to only in-stock products. This is done to avoid stock-out situations, which are not useful when simulating the control of an RMFS for such situation, i.e. we assume that no unfulfillable pick orders are submitted to the system. The number of order lines and the number of units per line are also chosen from a distribution previously specified by the user. For choosing the SKU for each replenishment order, we use the same popularity distribution connected to information about the order size in which a certain product is replenished. For this, we also allow setting a parameterized amount of replenishment orders to be return orders, i.e. single line and single unit replenishment orders. The space consumption of one unit of a SKU on a pod is a one-dimensional factor which is also set according to a user-specified distribution. This is necessary to enable the simulation of both pick and replenishment operations at the same time, while emulating the inventory situation in our system.

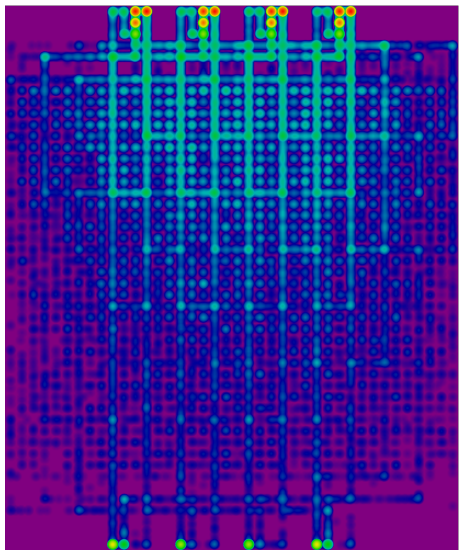
Moreover, we implement three procedures for generating new replenishment and pick orders. First, a constant order backlog scenario can be selected. This means that a completed order is immediately replaced by a new one. This is done to keep the system under constant pressure. To avoid completely overfilling or draining the inventory over time, we allow the specification of replenishment and pick order

generation pauses according to given inventory level thresholds. E.g., if the inventory reaches a 95% filling, replenishment order generation is paused until it drops below 85% again. Second, we generate orders according to arrivals of a configurable Poisson process. This Poisson process can be inhomogeneous to capture order peak situations during a day or match certain patterns observed at distribution centers. Finally, we allow the input of files to specify which orders are generated during the simulation horizon. Additionally, we allow a combined setting of one of the order generation scenarios above with the submission of order batches at given periodic time points. This is done to allow emulation of batch operations or hybrid scenarios. With all of these options, we aim to resemble most artificial and realistic scenarios.

Controller Configuration The last input file specifies the controller to use for each decision problem as well as the parameters. This enables a flexible configuration due to the modular controller concept. Controllers integrating multiple decision problems can be configured using dummy controllers for the other components. With the help of reflection most parameter structures only need to be defined once. Without additional implementation effort new parameters are serializable and configurable in the graphical user interface.



(a) Sample progression plot of picked orders (red) and distance traveled (blue) per hour



(b) Example of a heatmap showing robot movement behavior over time (red ≡ high, purple ≡ low)

Figure 2.5 Examples of the system's output

2.3.1.3 Output

In order to investigate the system's behavior in more detail, different output measures are tracked and logged over time. For this, we distinguish three main measure types. First, a footprint is written for each simulation execution to allow for an easy comparison of multiple executions. This footprint contains most basic performance measures for the simulation run like the order throughput rate or the distance traveled by the robots. Second, time-based information is logged to generate plots after execution of the simulation. These are useful for post-experiment analysis of the simulation processes, especially if no visualization was attached. As an example, Figure 2.5a shows the pick order throughput and overall distance traveled per hour for a simulation in which the PSA strategy is switched after half of the simulation horizon. For this experiment, a random PSA strategy was replaced with a strategy selecting the nearest available storage location for the pod. This is reflected by the decrease in distance to cover by the robots, which immediately leads to an increased throughput in pick orders, because more pods are available at the pick stations. Finally, location-based information is logged to conduct heatmap analyses. This is useful, for example, to get insights about congestion effects when looking at robot movement behavior over time (see Fig. 2.5b). In this example, we can see the queuing happening at the pick stations at the top and the replenishment stations at the bottom. Furthermore, we can identify highly frequented areas which are prone to congestion. Note that the logarithm was applied to the heat values in order to increase the contrast in color in the resulting heatmap.

2.3.2 Robot Movement Emulation

Table 2.2 Symbol definitions

Symbol	Description
d	Drive distance
v_t	The speed at time t
s_t	The position at time t
$\overrightarrow{a}$	Acceleration in $\frac{m}{s^2}$
$\overleftarrow{a}$	Deceleration in $\frac{m}{s^2}$ (negative)
$\overline{v}$	Top-speed in $\frac{m}{s}$

In order to accurately capture the movement behavior of the robots we consider linear acceleration and deceleration for straight robot movement. Furthermore, we allow continuous turning of the robots while adhering to a certain turning speed. Thereby, turning can only be done on the spot while standing still. Hence, we assume that moving along a curve is not possible. Next, we describe the necessary

calculations for modeling the described movement.
The computation of traveling times and distances for straight movement is based on uniform acceleration and deceleration. For this, the velocity of a robot has to be considered to determine its arrival time at a destination node. The symbols used in the following description are defined in Tab. 2.2. The fundamental formulas for all remaining definitions are given by the resulting speed v_t after accelerating by $\overrightarrow{a}$ for time t from initial speed v_0 (see Eq. 2.1), the new position s_t compared to the initial position s_0 after driving while accelerating for time t (see Eq. 2.2), and the new position s_t compared to the initial position s_0 after driving at top-speed for time t (see Eq. 2.3).

$$v_t = \overrightarrow{a}t + v_0 \tag{2.1}$$

$$s_t = \frac{\overrightarrow{a}}{2}t^2 + v_0 t + s_0 \tag{2.2}$$

$$s_t = \overline{v}t + s_0 \tag{2.3}$$

For the implementation of the simulation framework, it is required to not only calculate the time for covering a distance when initially standing still, but also to calculate distances and times based on an initial speed $v_0 > 0$, because simulation events may occur at any time during robot travel. For this, the time ($t_{v_0 \to \overline{v}}$) and distance ($s_{t_{v_0 \to \overline{v}}}$) to reach the top-speed are needed (see Eq. 2.4 & 2.5). This can analogously be defined for full deceleration (see Eq. 2.6 & 2.7).

$$\overline{v} = \overrightarrow{a}t_{v_0 \to \overline{v}} + v_0 \Leftrightarrow t_{v_0 \to \overline{v}} = \frac{\overline{v} - v_0}{\overrightarrow{a}} \tag{2.4}$$

$$s_{t_{v_0 \to \overline{v}}} = \frac{\overrightarrow{a}}{2}\left(t_{v_0 \to \overline{v}}\right)^2 + v_0 t_{v_0 \to \overline{v}} \tag{2.5}$$

$$0 = \overleftarrow{a}\,t_{v_0 \to 0} + v_0 \Leftrightarrow t_{v_0 \to 0} = \frac{0 - v_0}{\overleftarrow{a}} \tag{2.6}$$

$$s_{t_{v_0 \to 0}} = \frac{\overleftarrow{a}}{2}\left(t_{v_0 \to 0}\right)^2 + v_0 t_{v_0 \to 0} \tag{2.7}$$

For the calculation of time for traveling distance d when starting at an initial speed of v_0, four cases need to be considered. In the first case, only deceleration is possible. In the second case, cruising at top-speed and deceleration are possible. In the third case, acceleration up to top-speed, cruising at top-speed and deceleration are possible. In the fourth case, the distance is so short that only acceleration and deceleration phases are possible. The function defined in Alg. 2.1 calculates the remaining cruise time for all of the cases. Line 7 uses the time at which acceleration switches to deceleration, which is described in more detail below.

Algorithm 2.1: $CruiseTime(\overrightarrow{a}, \overleftarrow{a}, \overline{v}, v_0, d)$

if $d \leq s_{t_{v_0 \to 0}}$ **then**
  return $t_{v_0 \to 0}$
if $v_0 = \overline{v}$ **then**
  return $\frac{d - s_{t_{v_0 \to 0}}}{\overline{v}} + t_{v_0 \to 0}$
if $s_{t_{v_0 \to 0}} + s_{t_{v_0 \to 0}} \leq d$ **then**
  return $t_{v_0 \to \overline{v}} + \frac{d - s_{t_{v_0 \to \overline{v}}} - s_{t_{v_0 \to 0}}}{\overline{v}} + t_{v_0 \to 0}$
return $\sqrt{\frac{d + \frac{\overrightarrow{a}}{2}\left(\frac{v_0}{\overrightarrow{a}}\right)^2}{\frac{\overrightarrow{a}}{2} + \frac{\overrightarrow{a}}{2\overleftarrow{a}}}} + \sqrt{\frac{d + \frac{\overrightarrow{a}}{2}\left(\frac{v_0}{\overrightarrow{a}}\right)^2}{\frac{\overleftarrow{a}}{2} + \frac{\overleftarrow{a}}{2\overrightarrow{a}}}} - \frac{v_0}{\overrightarrow{a}}$

Let d' be the full distance from the start node to the destination node, thus, starting with zero speed at the beginning of d' and stopping with zero speed at the end of d'. For this, the time of switching from acceleration to deceleration is given by Eq. 2.8, i.e., the time for driving while accelerating. To calculate this time we make use of the fact that the speed at which we switch from acceleration to deceleration must match ($\overrightarrow{a} t_1 = \overleftarrow{a} t_2$), hence, we can substitute t_2 with $\frac{\overrightarrow{a} t_1}{\overleftarrow{a}}$.

$$
\begin{aligned}
d' &= \frac{\overrightarrow{a}}{2} t_1^2 + \frac{\overleftarrow{a}}{2} t_2^2 \\
\Leftrightarrow d' &= \frac{\overrightarrow{a}}{2} t_1^2 + \frac{\overleftarrow{a}}{2} \left(\frac{\overrightarrow{a} t_1}{\overleftarrow{a}}\right)^2 \\
\Leftrightarrow d' &= \frac{\overrightarrow{a}}{2} t_1^2 + \frac{\overleftarrow{a}}{2} \frac{(\overrightarrow{a} t_1)^2}{2\overleftarrow{a}} \\
\Leftrightarrow t_1^2 &= \frac{d'}{\frac{\overrightarrow{a}}{2} + \frac{\overrightarrow{a}^2}{2\overleftarrow{a}}} \\
\Leftrightarrow t_1 &= \sqrt{\frac{d'}{\frac{\overrightarrow{a}}{2} + \frac{\overrightarrow{a}^2}{2\overleftarrow{a}}}}
\end{aligned}
\tag{2.8}
$$

For the calculation, we assume that speed is currently zero and the movement just starts at the start node. If $v_0 > 0$ and $d < d'$, then d' has to be calculated by using d (see Eq. 2.9).

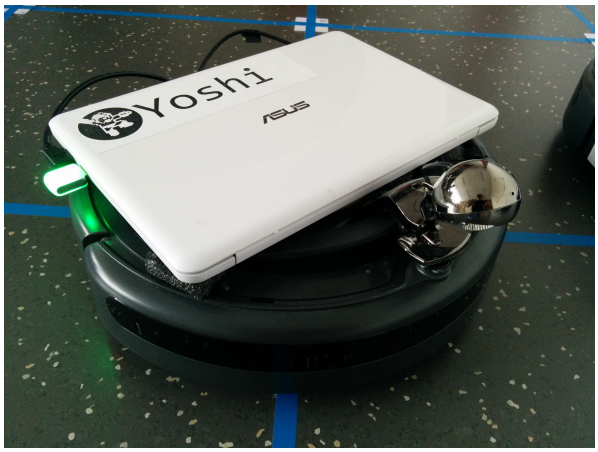
$$
d' = d + \frac{\overrightarrow{a}}{2} \left(\frac{v_0}{\overrightarrow{a}}\right)^2 \tag{2.9}
$$

Analogously to Eq. 2.8, it is also possible to solve for t_2, i.e., the time for driving while decelerating. The sum of t_1 and t_2 is the complete time for the cruise from

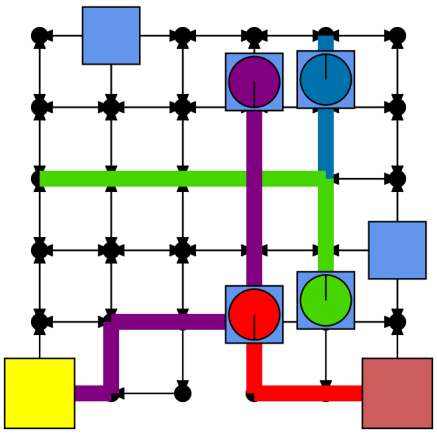
start node to destination node. The time for accelerating to v_0 is being subtracted, such that the remaining time can be expressed as in line 7 of Alg. 2.1.

2.4 Demonstrator

We implemented a demonstrator functionality to investigate how our algorithms developed within RAWSim-O work with real robots. For example with the help of the iRobot Create 2, a mobile robot platform based on the Roomba vacuum cleaning robot. There are several reasons we choose the iRobot Create 2: ease of programming, low complexity, similar movement behavior like typical RMFS robots and low costs. The robots are equipped with ASUS Eee PCs through serial-to-USB cables for processing capabilities, webcams for line-following, blink(1) for visual feedback, and RFID tag readers mounted inside the former vacuum cleaning compartment for waypoint recognition (see Figure 2.6a). Although we cannot emulate the transportation of pods with the robots, we are still able to study the overall movement behavior in a real situation, which is more prone to errors and noise than a simulation. Technically, the demonstrator robots replace the simulated robots, hence, a hybrid of a real system and a simulation is built.
We demonstrate our RMFS with a simple running example of four robots in a grid-world with $0.45m \times 0.45m$ cells (totally 36 cells), see Figure 2.6b for the view from RAWSim-O's visualization for this example and Figure 2.6c for the view of the demonstration. One replenishment station is set on the left bottom side and one pick station is set on the right bottom side. In total, there are six pods (blue rectangles) available. The maximum velocity limit of each robot is $0.21\frac{m}{s}$, while the time it takes for each robot to do a complete turn is set to $5.5s$. And the maximum acceleration and deceleration of each robot are set to $0.5\frac{m}{s^2}$ and $-0.5\frac{m}{s^2}$. To adhere to the kinematic constraints (such as turning times and acceleration) in continious time, we use the MAPFWR-solver from [10] to generate time-efficient collision- and deadlock-free paths. Those paths are converted to a sequence of go straight, turn left, and turn right commands and sent to the robot via WiFi. The robot then executes these commands and sends back the RFID waypoint tag of each intersection it comes across. By doing this, the movement of the real robot and the expected movement are synchronized. This needs to be done in order to avoid errors from the noisy real-world setting to add up. Similarly to the simulated environment, pick and replenishment operations are emulated by blocking the robot at the station for a fixed time. With the published source code and similar components the demonstrator can be rebuild.
The robot line-following is implemented using a common PID (Proportional Integration Differential) controller method. The center of the line is extracted by using the OpenCV library. The line recognition first translates the image to a binary one



(a) Close-up of a demonstrator robot

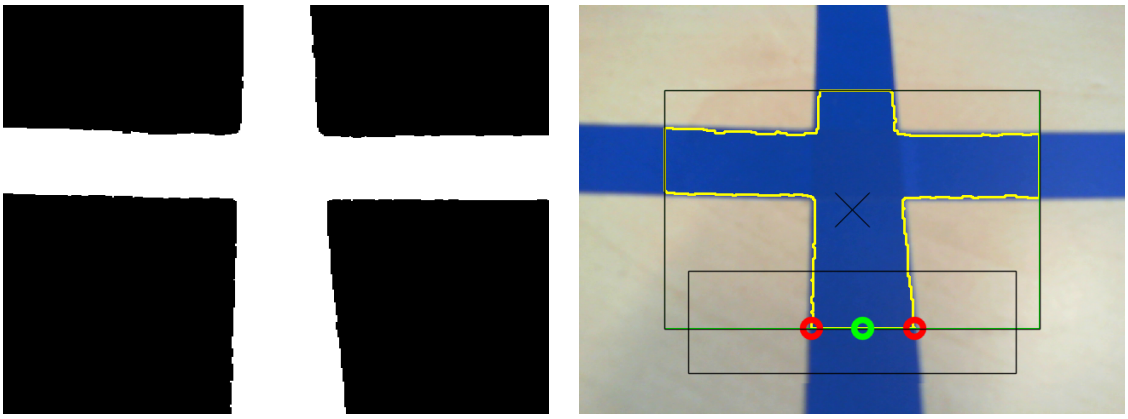


(b) The view from RAWSim-O

(c) Multiple robots in an emulated RMFS

Figure 2.6 Demonstrator example of four robots running in an emulated RMFS

by applying a range filter, and then applies the basic morphological operation "erosion" for removing noise (see resulting image in Figure 2.7a). Afterwards, contour extraction is used to find the line's borders within the gray image. Using these lines, the bottom center is estimated to allow a stable target (see green ring in Figure 2.7b) for the PID controller.
Using the demonstrator, we are able to show the successful application of the path planning algorithms developed within RAWSim-O in a real-world situation. Furthermore, it can be used to demonstrate the basic idea of an RMFS at a small scale.



(a) Filtered image showing the line **(b)** Processing result on original image

Figure 2.7 Line-following example using a blue line

2.5 Conclusion

In this work, we outlined core real-time decision problems occurring when operating an RMFS. To investigate different solution approaches for those decision problems, we introduce the RMFS simulation framework RAWSim-O and describe its functionality and capabilities. Alongside this publication, we also publish the source code of the framework and already present controllers and algorithms at https://github.com/merschformann/RAWSim-O to support future research on RMFS. With this work we aim to support future research on RMFS. For example, a study of how to control systems involving multiple mezzanine floors can be done using RAWSim-O. For these, an efficient storage strategy and task allocation method is expected to be crucial in order to mitigate the bottle-neck effect introduced by the elevators. Furthermore, we have shown demonstrator application capabilities of the framework that enable the integration of simple robots for real world demonstrations.

2.6 Acknowledgements

We would like to thank Tim Lamballais for providing us with the concepts and implementation of the default layout generator used in RAWSim-O.

References

[1] Nils Boysen, Dirk Briskorn, and Simon Emde. “Parts-to-picker based order processing in a rack-moving mobile robots environment”. In: *European Journal*

of Operational Research 262.2 (2017), pp. 550–562. ISSN: 03772217. DOI: 10.1016/j.ejor.2017.03.053.

[2] L. Cohen et al. "Rapid Randomized Restarts for Multi-Agent Path Finding Solvers". In: *ArXiv e-prints* (2017).

[3] Liron Cohen, Tansel Uras, and Sven Koenig. "Feasibility Study: Using Highways for Bounded-Suboptimal Multi-Agent Path Finding". In: *Eighth Annual Symposium on Combinatorial Search*. 2015.

[4] John Enright and Peter R. Wurman. "Optimization and Coordinated Autonomy in Mobile Fulfillment Systems". In: *Automated Action Planning for Autonomous Mobile Robots*. Ed. by Sanem Sariel-Talay, Stephen F. Smith, and Nilufer Onder. 2011.

[5] Christopher J. Hazard, Peter R. Wurman, and Raffaello D'Andrea. "Alphabet Soup: A Testbed for Studying Resource Allocation in Multi-vehicle Systems". In: *Proceedings of AAAI Workshop on Auction Mechanisms for Robot Coordination*. Citeseer, 2006, pp. 23–30.

[6] A. E. Hoffman et al. "System and method for inventory management using mobile drive units". US20130103552 A1. 2013. URL: https://www.google.com/patents/US20130103552.

[7] Alaa Khamis, Ahmed Hussein, and Ahmed Elmogy. "Multi-robot Task Allocation: A Review of the State-of-the-Art". In: *Cooperative robots and sensor networks 2015*. Ed. by Anis Koubâa. Vol. 604. Studies in Computational Intelligence. Cham: Springer, 2015, pp. 31–51. ISBN: 978-3-319-18298-8. DOI: 10.1007/978-3-319-18299-5_2.

[8] René de Koster, Tho Le-Duc, and Kees Jan Roodbergen. "Design and control of warehouse order picking: A literature review". In: *European Journal of Operational Research* 182.2 (2007), pp. 481–501. ISSN: 03772217. DOI: 10.1016/j.ejor.2006.07.009.

[9] T. Lamballais, D. Roy, and M.B.M. de Koster. "Estimating performance in a Robotic Mobile Fulfillment System". In: *European Journal of Operational Research* (2016). ISSN: 03772217. DOI: 10.1016/j.ejor.2016.06.063.

[10] M. Merschformann, L. Xie, and D. Erdmann. "Path planning for Robotic Mobile Fulfillment Systems". In: *ArXiv e-prints* (2017).

[11] Adriaan Willem ter Mors. *The world according to MARP: Multi-Agent Route Planning*. Delft: Technische Universiteit Delft, 2010. ISBN: 9085599377.

[12] Kees Jan Roodbergen and Iris F.A. Vis. "A survey of literature on automated storage and retrieval systems". In: *European Journal of Operational Research* 194.2 (2009), pp. 343–362. ISSN: 03772217. DOI: 10.1016/j.ejor.2008.01.038.

[13] Elena Tappia et al. "Modeling, Analysis, and Design Insights for Shuttle-Based Compact Storage Systems". In: *Transportation Science* 51.1 (2017), pp. 269–295. ISSN: 0041-1655. DOI: 10.1287/trsc.2016.0699.

[14] James A. Tompkins. *Facilities planning*. 4th ed. Hoboken, NJ and Chichester: John Wiley & Sons, 2010. ISBN: 0470444045.

[15] Marc Wulfraat. *Is Kiva Systems a Good Fit for Your Distribution Center? An Unbiased Distribution Consultant Evaluation*. 2012. URL: http://www.mwpvl.com/html/kiva_systems.html (visited on 01/14/2018).

[16] Peter R. Wurman, Raffaello D'Andrea, and Mick Mountz. "Coordinating hundreds of cooperative, autonomous vehicles in warehouses". In: *AI Magazine* 29.1 (2008), p. 9. ISSN: 0738-4602.

Chapter 3

Multi-Agent Path Finding with Kinematic Constraints for Robotic Mobile Fulfillment Systems

Marius Merschformann[1] Lin Xie[2] Daniel Erdmann[1]

[1] *University of Paderborn, Paderborn, Germany*
[2] *Leuphana University of Lüneburg, Lüneburg, Germany*
marius.merschformann@uni-paderborn.de, lin.xie@leuphana.de

Submitted to Journal of Artificial Intelligence Research

Abstract

This paper presents a collection of path planning algorithms for real-time movement of multiple robots across a Robotic Mobile Fulfillment System (RMFS). In such a system, robots are assigned to move storage units to pickers at working stations instead of requiring pickers to go to the storage area. Path planning algorithms aim to find paths for the robots to fulfill the requests without collisions or deadlocks. The robots are fully centralized controlled. The traditional path planning algorithms do not consider kinematic constraints of robots, such as maximum velocity limits, maximum acceleration and deceleration, and turning time. This work aims at developing new multi-agent path planning algorithms by considering kinematic

constraints. Those algorithms are based on some exisiting path planning algorithms in literature, including WHCA*, FAR, BCP, OD&ID and CBS. Moreover, those algorithms are integrated within a simulation tool to guide the robots from their starting points to their destinations during the storage and retrieval processes. Ten different layouts with a variety of numbers of robots, floors, pods, stations and the sizes of storage areas were considered in the simulation study. Performance metrics of throughput, path length and search time were monitored. Simulation results demonstrate the best algorithm based on each performance metric.

3.1 Introduction

Due to the increasingly fast-paced economy, an efficient distribution center plays a crucial role in the supply chain. From the logistics perspective the main task is to turn homogeneous pallets into ready-to-ship packages that will be sent to the customer. Traditionally, as some customers' orders are received, pickers in different zones of a warehouse are sent to fetch the products, which are parts of several different customers' orders. After that, the products should be sorted and scanned. Once all parts of an order are complete, they are sent to packing workers to finish packaging. An extensive overview of manual order picking systems can be found in [1]. As shown in [41], 50% of pickers' time in these systems is spent on traveling around the warehouse. To ensure that the orders are shipped as fast as possible, automated storage and retrieval systems were introduced. An extensive literature review is provided by [29]. While using these systems offers a high potential throughput they also face certain drawbacks such as high costs, long design cycles, inflexibility and lack of expandability as pointed out by [47]. In order to improve or eliminate those disadvantages, automated Robotic Mobile Fulfillment Systems (RMFS), such as the Kiva System ([9], nowadays Amazon Robotics), have been introduced as an alternative order picking system in recent years. Robots are sent to carry storage units, so-called "pods", from the inventory and bring them to human operators, who work at picking stations. At the stations, the items are packed according to the customers' orders. [47] indicate that this system increases the productivity two to three times, compared with the classic manual order picking system. Moreover, the search and travel tasks for the pickers are eliminated.
The applications in a similar RMFS have been paid more attention recently (see [26] for a better overview). An efficient path-planning algorithm is important, since it aims at finding paths for robots to fulfill the requests without collisions or deadlocks, which is considered a major aspect of automation for storage and retrieval in an RMFS. The existing publications formulate path planing in such system as the classic Multi-Agent Pathfinding problem (MAPF) and this problem is solved with a bounded sub-optimal solver (see [7] and [6]). MAPF is a challenging problem with

many applications in robotics, disaster rescue and video games (see [45]). However, MAPF solvers from AI typically do not work with real-world mobile robots, since they do not work for agents considering kinematic constraints, such as maximum velocity limits, maximum acceleration and deceleration, and turning times, in continuous environments. The authors of [18] suggested a postprocessing to create a schedule that can be executed by robots, based on the output of a MAPF solver. Differ to that, we introduce a novel mathematical formulation of MAPF, called Multi-Agent Pathfinding Problem for mobile Warehousing Robots (MAPFWR for short), that takes some of the kinematic constraints of real robots as well as the structure of the simulated warehouse environment into account during the path-planning phase. The existing MAPF-algorithms, including WHCA*,FAR, BCP, OD&ID and CBS, are modifed for this purpose. There are two existing open-source frameworks to simulate an RMFS-System, the first framework, "Alphabet Soup" was published by [17], and the framework "RawSim-O" ([26]). This latter one is based on the former one with extending cases of multiple floors and they consider different robot movement emulations. In this work, we test different MAPFWR-algorithms in the framwork "RawSim-O", since it has realistic robot movement emulation by considering the robot's turning time and adjusted acceleration or deceleration formulas.
The first part of the paper gives a mathematical description of the MAPFWR problem and a literature review of the related problem MAPF. Next, we investigate the properties of the search space of MAPFWR. After that, we describe the path-planning algorithms that we implement. Finally, we present experimental results obtained by the simulation framework "RawSim-O".

3.2 Background

This section first describes RMFS and the inherent decision problems in more detail. Next, the MAPFWR problem and its related problem, the MAPF problem, is defined. At last, a literature review of MAPF is presented.

3.2.1 Robotic Mobile Fulfillment System

According to [16] the "(...) basic requirements in warehouse operations are to receive Stock Keeping Units (SKUs) from suppliers, store the SKUs, receive orders from customers, retrieve SKUs and assemble them for shipment, and ship the completed orders to customers." Using this definition an RMFS provides the temporal storage of SKUs, and their retrieval to fulfill incoming customer orders. The approach for this requirement is to use pods (shelf-like storage units) to store the inventory and bring these to replenishment and pick stations as they are required for insert and extract transactions of items. The basic layout of one floor in a RMFS is illustrated

in Fig. 3.1. The pods are located at the storage area in the middle of the layout. The robot carries a pod by following waypoints to the replenishment station on the left-hand side, where new items are inserted into the pod. After that, the pod is carried back to the storage area. Similar operations are done at the order picking station on the right-hand side, where items are picked to fulfill orders. The robots are guided by waypoints, which are typically connected by a grid-like graph (but not limited to it).

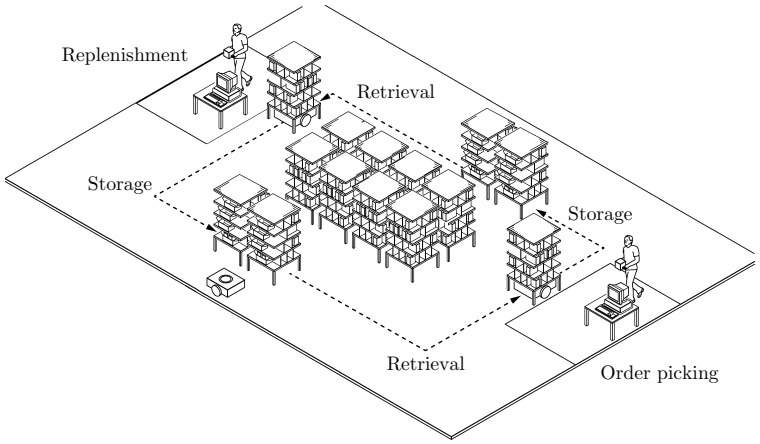


Figure 3.1 A basic layout of a RMFS based on the Kiva patent [19]

In the control of such a system many different decision problems need to be overcome to sustain an overall efficiency (see [9]). Limiting the view only at the core of the operational decision problems we propose the abstract structure given by Figure 3.2 to support a better understanding of the problem interactions in a RMFS. As new item bundles shall be stored in the inventory a *replenishment assignment* controller needs to choose from which replenishment station these shall be inserted and a *bundle storage assignment* controller needs to determine a suitable pod to store it on. For a new customer order first only a pick station to process it is chosen by an *order assignment* controller. The former process leads to insertion requests, while the latter leads to extraction requests. These can now be combined to tasks that shall be executed by a robot. While insert tasks can simply be determined by combining all requests with matching station and pod to a task, we need to select a pod to fulfill the extraction requests using a *pick pod selection* controller first to create extract tasks. The decision about the right pod to bring to a pick station is postponed, because it allows us to exploit more information that becomes available over time. In addition to the insertion and extraction requests that resemble the systems main purpose, park requests are the result of pods that need to be brought back to the inventory. For these a *pod storage assignment* controller determines a suitable storage location. The combination of requests to tasks and the allocation

of them to robots is done by a *task allocation* controller. Most of the mentioned tasks require the robot to go from one location to another, which leads to trips from one waypoint in the system to another. For these trips a *path planning* controller is needed to ensure time-efficient, collision- and deadlock-free paths. As the last is the main focus of this work, fixed controllers are used for all other problem components to enable a working system while allowing a fair comparison of the path planning methods (see Section 3.5). The existing researches of each decision problem can be found in [5] and [26]. We concentrate in this work only the path planning algorithms.

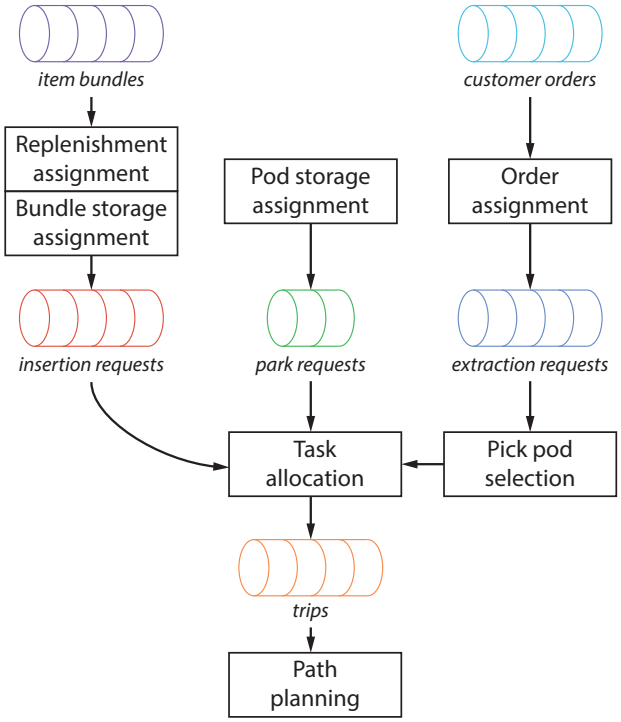


Figure 3.2 Overview of the core decision problems of a RMFS at operational scope

3.2.2 MAPFWR

As mentioned in Section 1, we formulate path planning as the MAPFWR problem, which will be described in this section.

3.2.2.1 Problem input

The main resources are given by the sets of pods $b \in \mathcal{B}$ and stations $m \in \mathcal{M} := \mathcal{M}^I \cup \mathcal{M}^O$ (replenishment- and pick-stations). The agents in the described system

are mobile robots $r \in \mathcal{R}$. Therefore, robots and agents are used synonymously in this work. These robots use a multi-layer planar graph $\mathcal{G}$ composed of waypoints $w \in \mathcal{W}$ as its vertices and edges $e \in \mathcal{E}$ connecting them. The edges of the transpose graph $\mathcal{G}^{-1}$ are denoted by $\mathcal{E}^{-1}$. Every layer of the graph represents one tier $h \in \mathcal{H}$ of an instance. All positions across the tiers are denoted by $\left(p_{it}^H, p_{it}^X, p_{it}^Y\right) \in \bigcup_{h \in \mathcal{H}} \{h\} \times \left[0, L_h^X\right] \times \left[0, L_h^Y\right]$ (i.e. the current tier the robot is on, the x- and the y-coordinate) for movable entities $i \in \mathcal{R} \cup \mathcal{B}$ (robots and pods) depending on the current time $t \in \mathbb{T}$ and bounded by the length L_h^X and width L_h^Y of the respective tier h. The time-independent position $\left(p_i^H, p_i^X, p_i^Y\right)$ of immovable entities $i \in \mathcal{W} \cup \mathcal{M}$ is defined analogously. At this, the time-horizon is continuous ($\mathbb{T} := \mathbb{R}_0^+$). At the starting time t_0 every robot, pod and station is located at an initial position $V_i^I \in \mathcal{W}$. The time-dependent distance between two entities i and j is defined as the euclidean norm and denoted by $d^E(i, j, t)$ (analogously the time-independent distance is $d^E(i, j)$). Additionally, the speed of the robots is expressed by $v_{rt} \in \mathbb{R}_0^+$ and orientation by $o_{rt} \in \{0, 2\pi\}$. The shape of the robot is abstracted by a circle with radius $L_r^R \in \mathbb{R}^+$. The angular velocity the robot is turning by at time t is indicated by $\omega_{rt} \in \mathbb{R}$. Furthermore, every robot has a constant acceleration rate $\overrightarrow{A}_r \in \mathbb{R}^+$, deceleration rate $\overleftarrow{A}_r \in \mathbb{R}^+$ and maximal velocity $\overline{V}_r \in \mathbb{R}^+$. The robot can change its orientation clockwise and counterclockwise by the maximal angular velocity of $\Omega_r \in \mathbb{R}^+$. This work abstracts from the rotational momentum.

The multiple tiers of an instance can be connected by elevators $l \in \mathcal{L}$, which are able to transport a robot between two waypoints w and w' in the constant time $T_{lww'}^L$, which depends on the height between the two waypoints to cover and the elevator's speed. During this time all waypoints associated with the elevator are blocked. When a robot stops at a station the time for handling one item is assumed to be constant. We distinguish the time for handling one physical item to fulfill an order at a pick station $T_m^O \in \mathbb{R}_0^+$ with $m \in \mathcal{M}^O$ and the time to handle an incoming bundle of items at a replenishment station $T_m^I \in \mathbb{R}_0^+$ with $m \in \mathcal{M}^I$. The time for picking up and setting down a pod is given by the constant value T^B. Analogous to the robot a pod's shape is abstracted by a circle with the radius L_b^R. A pod not currently carried by a robot can be stored at one of the parking positions $w \in \mathcal{W}^{SL} \subset \mathcal{W}$. The choice of the parking position is determined by another planning component that is not discussed any further in this work.

The length of the edges of the graph is set a priori and has a lower bound determined by the maximal radii of the robots and pods (see Eq. 3.1).

$$\forall (w_1, w_2) \in \mathcal{E} : d^E(w_1, w_2) \geq \max_{i \in \mathcal{R} \cup \mathcal{B}} \max_{i' \in \mathcal{R} \cup \mathcal{B} \setminus \{i\}} L_i^R + L_{i'}^R \tag{3.1}$$

With this requirement, all robots and pods can be arbitrarily distributed among the waypoints.

During runtime a robot can be asked to bring a pod to a station (to insert an item or extract it), to park a pod or to rest at a certain waypoint (these are called as tasks). Every one of these tasks can be decomposed into subtasks as shown in Tab. 3.1. Note that the first two subtasks of an insert or extract task can be skipped, if the robot already carries the right pod at the time the task is assigned to it. All subtasks, except for the move subtask, block the robot at its current position for the constant period of time described above. Only the move subtask has to be immediately considered here as it requires the generation of a path respecting the movement of other robots in the system. This path starts at the current location of the robot and ends at the destination waypoint specified by the move subtask.

Table 3.1 The task types and their subtask types

Task type	Subtask types
Insert	$(move, pickup, move, put)$
Extract	$(move, pickup, move, get)$
Park	$(move, setdown)$
Rest	$(move)$

3.2.2.2 Constraints during path generation

In the following, constraints that have to be adhered during path generation are described. At first, robots always have to be located either on a node or an edge (see Eq. 3.2).

$$\begin{aligned} \forall r \in \mathcal{R}, t \in \mathbb{T} : \exists w \in \left\{ \mathcal{W} \mid p_w^H = p_{rt}^H \right\} : p_w = p_{rt} \vee \\ \exists \left(w_1, w_2\right) \in \left\{ \mathcal{E} \mid p_{w_1}^H = p_{w_2}^H = p_{rt}^H \right\} : \\ d^E\left(w_1, r, t\right) + d^E\left(r, w_2, t\right) = d^E\left(w_1, w_2\right) \end{aligned} \tag{3.2}$$

Next, the robots cannot overlap with each other at any time (see Eq. 3.3) as this would immediately result in a collision. This is analogously defined for the pods in Eq. 3.4.

$$\forall r_1, r_2 \in \mathcal{R}, t \in \mathbb{T} : p_{r_1 t}^H = p_{r_2 t}^H \implies d^E\left(r_1, r_2, t\right) \geq L_{r_1}^R + L_{r_2}^R \tag{3.3}$$

$$\forall b_1, b_2 \in \mathcal{B}, t \in \mathbb{T} : p_{b_1 t}^H = p_{b_2 t}^H \implies d^E\left(b_1, b_2, t\right) \geq L_{b_1}^R + L_{b_2}^R \tag{3.4}$$

The function $f^O\left(b, t\right) : \mathcal{B} \times \mathbb{T} \to \mathcal{R} \times \mathcal{W}^{SL}$ determines the current owner of a pod, which can be either a robot carrying it or a storage location it is stored at. For this

we also require the injectivity of the function, because a pod can only be carried or stored by one entity (see Eq. 3.5).

$$\forall b_1, b_2 \in \mathcal{B}, t \in \mathbb{T} : f^O(b_1, t) = f^O(b_2, t) \implies b_1 = b_2 \tag{3.5}$$

Using this constraint, the position of the pod can be inferred by the position of either the robot or the storage location (see Eq. 3.6).

$$\begin{aligned} \forall b \in \mathcal{B}, t \in \mathbb{T} : \big(\exists r \in \mathcal{R} : f^O(b, t) = r \implies p_{bt} = p_{rt}\big) \\ \vee \big(\exists w \in \mathcal{W}^{SL} : f^O(b, t) = w \implies p_{bt} = p_{wt}\big) \end{aligned} \tag{3.6}$$

This also infers that a robot carrying a pod cannot pass waypoints at which a pod is stored. Conversely, a robot not carrying one can move beneath pods, thus, using waypoints at which a pod is stored. Furthermore, the turning of a robot is only allowed on a waypoint while it is not moving (see Eq. 3.7). Hence, a robot moving along an edge has to be oriented towards the same direction as the edge (see Eq. 3.8). These constraints limit the movement of the robots to the graph similarly to MAPF, but with the difference that a robot can be located between two waypoints at a time t.

$$\forall r \in \mathcal{R}, t \in \mathbb{T} : \omega_{rt} \neq 0 \implies v_{rt} = 0 \wedge \exists w \in \mathcal{W}^{SL} : p_w = p_{rt} \tag{3.7}$$

$$\begin{aligned} \forall r \in \mathcal{R}, t \in \mathbb{T}, (w_1, w_2) \in \left\{\mathcal{E} \mid p^H_{w_1} = p^H_{w_2} = p^H_{rt}\right\} : \\ p_{rt} \neq p_{w_1} \wedge p_{rt} \neq p_{w_2} \wedge d^E(w_1, r, t) + d^E(r, w_2, t) = d^E(w_1, w_2) \\ \implies o_{rt} = \operatorname{atan2}\left(p^Y_{w_1} - p^Y_{rt}, p^X_{w_1} - p^X_{rt}\right) \\ \vee o_{rt} = \operatorname{atan2}\left(p^Y_{w_2} - p^Y_{rt}, p^X_{w_2} - p^X_{rt}\right) \end{aligned} \tag{3.8}$$

Under these principles the movement of a robot can be described as a sequence of an optional rotation, an acceleration, an optional top-speed and a deceleration phase. The time it takes the robot to rotate depends on the rotation angle φ and is defined in Eq. 3.9.

$$t^R(r, \varphi, \Omega_r) := \varphi \Omega_r^{-1} \text{ with } r \in \mathcal{R}, \varphi, \Omega_r \in \mathbb{R}^{\geq 0} \tag{3.9}$$

Respecting the robots constant acceleration rate and the following equation expresses the time that is consumed while a robot is moving straight (according to [28]). At this, two cases have to be distinguished. In the first case the driving

distance d is sufficiently long to fully accelerate the robot. This is the case, if the following is true (see Eq. 3.10).

$$d \geq \tilde{d} \text{ with } \tilde{d} := \frac{\overrightarrow{A}_r}{2}\left(\frac{\overline{V}_r}{\overrightarrow{A}_r}\right)^2 + \frac{\overleftarrow{A}_r}{2}\left(\frac{\overline{V}_r}{\overleftarrow{A}_r}\right)^2 \tag{3.10}$$

This leads to the time being a sum of the acceleration phase, full speed phase and deceleration phase time. If the distance is too short to fully accelerate, the time only depends on the acceleration and deceleration phases. Equation 3.11 comprises both cases.

$$t^D\left(r, d, \overrightarrow{A}_r, \overline{V}_r, \overleftarrow{A}_r\right) := \begin{cases} \frac{\overline{V}_r}{\overrightarrow{A}_r} + \frac{d - \frac{\overrightarrow{A}_r}{2}\left(\frac{\overline{V}_r}{\overrightarrow{A}_r}\right)^2 - \frac{\overleftarrow{A}_r}{2}\left(\frac{\overline{V}_r}{\overleftarrow{A}_r}\right)^2}{\overline{V}_r} + \frac{\overline{V}_r}{\overleftarrow{A}_r} & \text{if } d \geq \tilde{d} \\ \sqrt{\frac{d}{\frac{\overrightarrow{A}_r}{2} + \frac{\overrightarrow{A}_r^2}{2\overleftarrow{A}_r}}} + \sqrt{\frac{d}{\frac{\overleftarrow{A}_r}{2} + \frac{\overleftarrow{A}_r^2}{2\overrightarrow{A}_r}}} & \text{if } d < \tilde{d} \end{cases} \tag{3.11}$$

Every replenishment station, pick station and elevator can only be used by one robot at a time. For storing bundles at an replenishment station m the robot is blocked for a constant time T_m^I during the execution of subtask *put* at the station's position. Analogously, the robot waits a constant time T_m^O during the subtask *get* at pick stations. For traveling with elevator l from w to w' the robot is blocked for $T_{lww'}^L$ time-units at w with w and w' contained in $\mathcal{W}_l$. During execution of the system every robot that has just finished a task requests a new one. The decision of which task the robot has to execute next is determined by another component and given by $\alpha^{TA}(r,t)$ with all possible tasks denoted by Tab. 3.1. For all robots with the subtask *move*, a path must be planned. A path π_{rt} of robot r consists of a sequence of triples $(w, stop, wait)$. The triple describes the actions of the robot with $w \in \mathcal{W}$ as the waypoint to go to next, $stop \in \{true, false\}$ denoting whether the robot stops at the waypoint and $wait \in \mathbb{R}_0^+$ determining the time the robot waits at the waypoint if $stop = true$ before executing the next action. Let $\pi'_{rt} \subseteq \pi_{rt}$ be a subpath for which the first and last triples denote $stop = true$ and for all others $stop = false$. Let $(w_1, \ldots, w_n)$ be the node sequence of the subpath π'_{rt}, then two subsequent nodes have to be connected by an edge and all edges have to be parallel to each other. Equations 3.12 and 3.13 comprise this requirement.

$$\forall i = 0, \ldots, n-1 : (w_i, w_{i+1}) \in \mathcal{E} \tag{3.12}$$

$$\begin{aligned}\forall i = 0, \ldots, n-2 : \operatorname{atan2}\left(p^X_{w_i t} - p^Y_{w_{i+1} t}, p^Y_{w_i t} - p^X_{w_{i+1} t}\right) \\ = \operatorname{atan2}\left(p^Y_{w_{i+1} t} - p^Y_{w_{i+2} t}, p^X_{w_{i+1} t} - p^X_{w_{i+2} t}\right)\end{aligned} \tag{3.13}$$

Table 3.2 shows the different conditions that need to be valid for completing a subtask for robot r at time t. A change of the value for function $f^O(b,t)$ is thereby only possible at time $t = t'$ at which either the subtask *pickup* or *setdown* is completed. The most relevant subtask is depicted by *move*, because it depicts the operation of going from one waypoint to another. Furthermore, the subtask *pickup* lifts a pod such that the robot can carry it. The reverse operation of setting down a pod is depicted by *setdown*. At last, the subtask *put* stores a number of item bundles in a pod while the subtask *get* picks a number of items from a pod.

Table 3.2 Conditions for switching from the respective subtask to a succeeding one.

Subtask	Condition
move	$p_{rt'} = p_w \wedge v_{rt'} = 0$
pickup	$\exists w \in \mathcal{W}^{SL} \forall t \in \left[t' - T^B, t'\right] : f^O(b,t) = w \wedge p_{rt} = p_{bt} = p_w$
setdown	$\exists b \in \mathcal{B} \forall t \in \left[t' - T^B, t'\right] : f^O(b,t) = r \wedge p_{rt} = p_{bt} = p_w$
put	$\forall t \in \left[t' - T^O_m, t'\right] : f^O(b,t) = r \wedge p_{rt} = p_{bt} = p_m$
get	$\forall t \in \left[t' - T^I_m, t'\right] : f^O(b,t) = r \wedge p_{rt} = p_{bt} = p_m$

3.2.2.3 Objectives

One of the main objectives for an RMFS is a high throughput of customer orders. Since we are also considering replenishment operations, we combine the number of picked items with the number of stored bundles to our main metric of handled units overall. This is supported by the robots bringing suitable pods to the stations to complete the aforementioned insert and extract requests. Hence, more time-efficient paths for the robots increase the system's throughput by decreasing the waiting times at the stations. Note that we assume that the subtasks *pickup*, *setdown*, *get* and *put* underlie constant times, therefore the minimization can be achieved by the time-efficient paths, except for possible queuing time at stations. It is similar to the function *sum-of-cost* in [11], where driving and waiting times are aggregated for the finding paths. Therefore, we are also especially interested in the average time for completing a trip. At last, the average distance covered per trip is important when considering wear of the robots and as an indicator for energy consumption. This is also similar to the fuel function in [11].

3.2.3 Related work

The similar problem MAPF is widely discussed in the literature, with applications from video games (see [22]) to exploration of three-dimensional environments with quadrotor drones (see [42]). In this problem, a set of agents is given, and each of them has its start and end positions. It aims at finding the path for each agent without causing collisions. In many cases minimizing the sum of the timesteps that are required for every agent to reach its goal is also considered as an additional goal. [33] describe this system as generally consisting of a set of agents, each of which has a unique start state and a unique goal state. Moreover, the time is discretized into timepoints and the time for rotation of each agent is set to zero. Also, it is assumed that the time for crossing each arc is constant. These differ to our problem, that means our problem MAPFWR considers continious timepoints and each agent requires time to rotate and the time for crossing each arc might be differed for each agent. Therefore, MAPF can be considered as a special case of our problem.
In the literature, there are a number of sub-optimal/optimal MAPF solvers (see Sections 3.2.3.1 and 3.2.3.2). In this paper, we modify some of them to solve MAPFWR, especially search-based optimal/sub-optimal MAPF solvers, since they are ususally designed for the sum-of-costs objective function and they support the massive search. The details of the modifications can be found in Section 4. The state-of-the-art MAPF path planning algorithms can be found in [33].

3.2.3.1 Sub-optimal algorithms

Finding an optimal solution for the MAPF problem is proven to be NP-hard (see [48]). The standard admissible algorithm for solving the MAPF problem is the A*-algorithm, which uses the following problem representation. A state is a n-tuple of grid locations, one for each of n agents. The standard algorithm considers the moves of all agents simultaneously at a timestep, so each state potentially has b^n legal operators (b is the number of possible actions). With the increasing number of agents the state space grows exponentially, therefore, there are many sub-optimal solvers in the literature for solving this problem more quickly.

Search-based Solvers These solvers are usually designed for the sum-of-cost objective function. Hierarchical Cooperative A* (HCA*), introduced by [35], plans the agents one at a time according to some predefined orders. A global space time reservation table is used to reserve the path for each agent. That ensures the path for actual agent do not include collisions with the path of previous agent. Windowed-HCA* (WHCA*) in [35] enhances HCA* to apply the reservation table within a limit time window. There are some other enhancements of HCA* shown in [3], [37] and [30]. [13] introduce a simple meta-algorithm called Biased Cost Pathfinding

(BCP), which assigns a priority to each agent. For each agent, A* is used to find the optimal path without considering collisions. This algorithm extends $h(n)$ with an additional virtual cost to control collisions.

Rule-based Solvers These solvers include specific agent-movement rules for different scenarios, but they are usually do not include massive search. Therefore, they are not adopted in this paper. These solvers include TASS ([21]), Push-and-Swap ([25] and [32]), Push-and-Rotate ([46]), BIBOX ([38]) and diBOX ([4]).

Hybrid Solvers These solvers include both movement rules and massive search. For example, Flow Annotation Replanning (FAR), introduced by [44] and [45], adds a flow restriction to limit the movement along a given row or column to only one direction, which avoids head-to-head collisions. The shortest path for each agent is implemented by the A*-algorithm independently, and a heuristic procedure is used to repair plans locally, if deadlocks occur. This method shows similar results to WHCA*, but with lower memory capacities and shorter runtime. A similar approach was proposed in [20].

3.2.3.2 Optimal algorithms

There are some optimal solvers for MAPF discussed in the literature, such as the algorithm of [36] and Conflict Based Search (CBS) introduced by [33]. It is still possible to find the optimal solutions for real-world problems within an acceptable time: for example, if the paths found by an A*-based algorithm do not contain any collisions. Moreover, it is easier to find optimal solutions if the number of agents is small compared to the size of the graph.

Reduction-based Solvers Some optimal solvers in the literature try to reduce MAPF to standard known problems, such as SAT ([15] and [40]), ASP ([10]) and CSP ([31]), since they have existing high-quality solvers. However, they were not designed to solve the sum-of-cost objective function; therefore, it is not easy to modify them for our purpose. Still, A first reduction-based SAT solver for sum-of-costs objective function was introduced in [39].

Search-based Solvers Some search-based solvers are based on A* algorithm and try to overcome the drawbacks of A*, such as large open-list of successors and large number of neighbors during branching. In order to reduce the number of nodes, which are generated but never expanded, Standley introduced an operator decomposition (OD). An intermediate node is introduced to ensure only the moves of a single agent are considered when a regular A* node is expanded. The authors

in [14] introduce another method, called a priori domain knowledge to reduce the number of nodes. Moverover, Standley introduced the Independence Detection (ID) framework to reduce the effective number of agents. The idea behind this is to detect independent groups of agents. Two groups of agents are independent, if an optimal solution can be found by a low-level solver (such as OD) for each group and no conflict occurs between them. Initially, each agent is a group and an optimal solution is found for it. The agents, which have conflicts with each other, are merged into one group and new solutions are found for them, and so on. This method ends if one solution without collision is found, or all agents belonging to one group. This algorithm is called OD&ID in the rest of this paper. Another optimal MAPF solver based on A* is M* (see [43]). There are some search-based solvers are not based on A* algorithm, such as Conflict Based Search (CBS) introduced by [33] and Increasing Cost Treee Search (ICTS) by [34]. The idea of CBS is similar to that of the branch-and-bound algorithm (see [24]), where MAPF is decomposed into a large number of constrained single-agent pathfinding problems. It aims at finding a minimum-cost constraint tree without collisions. This algorithm works on two levels, namely high level and low level. At the high level, conflicts are found and constraints are added, while an optimal path is found for each agent at the low level, which is consistent with the new constraints. The low-level and high-level searches are best-first searches; however, [2] introduced Enhanced Conflict-Based Search (ECBS), which uses focal searches (see [27]) for both levels. This algorithm is sub-optimal, but with shorter runtime compared with CBS, since the focal search considers a subset of the best search and expands a node with $f(n) \leq \epsilon f_{min}$. The parameter ϵ is defined by the user. Cohen et al. [7] combine *highways* with ECBS for solving the Kiva system. According to the authors, the combination has been demonstrated to decrease computational runtime and costs compared with ECBS; moreover, ECBS alone is slow for a large number of agents in the Kiva system. ICTS works with discrete increasing cost tree at high level, while at the low level a non-conflicing complete solution is found. We do not adopt ICTS since MAPFWR works in continuous enviornments. More about search-based optimal solvers for MAPF can be found in [12].

3.3 Search space

In this section we discuss the properties of search space $\mathcal{S}$ as a state space. Note that $\mathcal{S}$ is different from the graph $\mathcal{G}$. The state in search space $\mathcal{S}$ is noted as n, while the node in Graph $\mathcal{G}$ is noted as w. As described in Section 3.2.3 for MAPF, there are four possible actions for agents in the grid graph and one action for waiting. Therefore, the grade of the search space is $O(5^k)$, where k is the number of agents. However, the grade of the search space of MAPFWR can be infinite, since each

agent can get any orientation and can wait for any interval. Therefore, we assume that the waiting time is limited to a given interval $T^W \in \mathbb{R}^+$. So the cost of the arc (n_1, n_2) for a waiting agent r is $c_r^W(n_1, n_2) = T^W$, while the cost of the arc for a moving agent is $c_r^M(n_1, n_2) = t^R(r, \varphi, \Omega_r) + t^D\left(r, s, \overrightarrow{A}_r, \overline{V}_r, \overleftarrow{A}_r\right)$. At this, the times for rotation and driving are considered, and s is the Euclidean distance.
With the discreted waiting time, the grade of the search space in MAPFWR is similar to the grade of the search space in MAPF, but the time for each action differs from one agent to another. An example illustrated in Fig. 3.3 shows the reservation for a path from w_1 to w_3 with waiting time 0 at w_2. The marked area (blue) means that agents cannot go through this node for the given time, e.g., because another agent already has an ongoing reservation for the node at the time. For MAPF in the left-hand graph, two actions are required, namely "move to w_2" and "move to w_3". However, there are two possible cases for MAPFWR. For the case in the central graph (where a quick stop at w_2 is done) we get the path: $(w_1, true, 0), (w_2, true, 0), (w_3, true, 0)$. And for the case in the right-hand graph (without stopping at w_2) we get the path: $(w_1, true, 0), (w_2, false, 0), (w_3, true, 0)$. w_2 in the central graph is blocked for a longer time due to the times for acceleration and deceleration necessary for the stop; instead, the reservation of w_2 in the right-hand graph is possible without overlapping. The same problem can also appear in the reservation for a sequence of nodes.

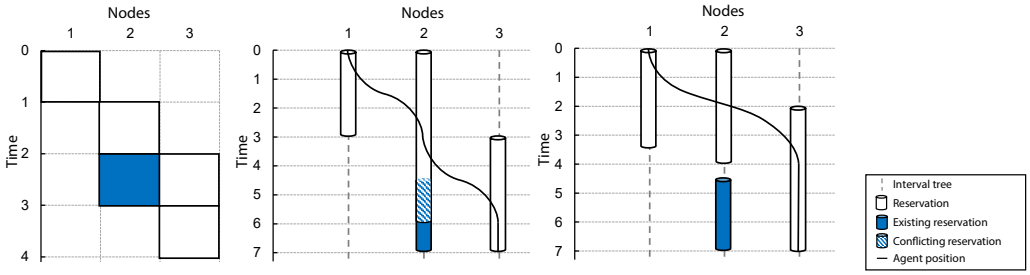


Figure 3.3 The reservation for the path from w_1 to w_3 with waiting time 0 at w_2. From left to right: MAPF, MAPFWR with stopping at w_2, MAPFWR without stopping at w_2

Therefore, we provide two solutions. The first solution is that we generate a move action only for connected nodes and a wait action with a fixed given time. Once a conflict occurs, we search the move actions in the same direction until we get the first node with a move action without conflict, or no further nodes exist in that direction. Therefore, the maximum grade of states in the search space for an agent is equal to the grade of nodes in the graph. The second solution is extending reservations to the next node in the same direction by assuming a continuous drive. Fig. 3.4 shows an example to generate the successors for the node w_4, namely w_5, w_6, w_7. If we

ignore the wait actions, the number of nodes corresponds to the number of states. The cost for w_5 is equal to the sum of the drive time until w_4, the time for rotating by the angle β and the moving time between w_4 and w_5. This is analogously done for w_7 with the angle γ. For the calculation of the cost of w_6, a backward search is until the last rotation is done (in this example: w_1). We can infer that the agent had to stop at this node. Thus, the cost of w_6 is determined by adding up the sum of cost up to w_1, the time for rotating by the angle α, and the drive time from w_1 to w_6 without stopping. The backward search can be done in $O(1)$, if we keep track of the last rotation or waiting action for each node.

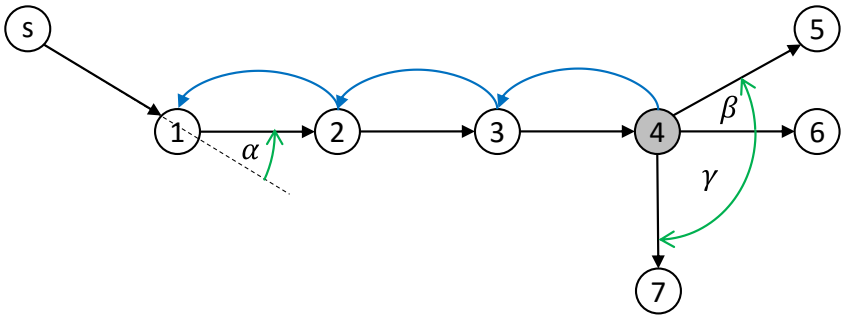


Figure 3.4 One example for the calculation for the costs of the successors of w_4

In the second solution above, we consider only times (costs), so we can abandon the wait actions. Moreover, we don't need to generate and check the reservations. In this case, the grade of the search space in our problem is equal to the grade of search space in MAPF.

Now we have to prove that the A*-Algorithm is complete and admissible in the search space of our problem, because some algorithms we discuss in the next section are based on the A*-Algorithm. An algorithm is complete if it terminates with a solution in case one exists, while an algorithm is admissible if it is guaranteed to return an optimal solution whenever a solution exists. According to the properties shown in [27], when the search space is a tree, there is a single beginning state and a set of goal states, the cost of the path is the sum of the costs of all arcs in this path, and the heuristic function h meets the following conditions: $h(n) \geq 0\ \forall n$ and $h(n) = 0$ if n is a goal state. Moreover, each state should have a finite number of successors and the cost $c(n_1, n_2)$ of each arc (n_1, n_2) in the search space should meet the condition $c(n_1, n_2) \geq \delta \geq 0$. According to the problem description in Section 3.2 and this section, the search space of our problem fulfills most of the conditions, except the last one. Therefore, we only have to prove that the cost of each arc in S has a finite lower bound δ (see the proof in Section 3.7). So the A*-Algorithm is complete and admissible in the search space of our problem.

In the next section, we discuss the path planning algorithms; most of them use the A*-algorithm for searching a path from actual location w_0 to goal location w_e for an agent r. The heuristic function $h(n)$ is defined as:

$$h(n) = t^D\left(r, d^E\left(w_0, w_e\right), \overrightarrow{A}_r, \overline{V}_r, \overleftarrow{A}_r\right)$$

where $d^E\left(w_0, w_e\right)$ is the Euclidean distance from w_0 to w_e. Since the moving time increases monotonically during the trip and the Euclidean distance satisfies triangle inequality, hence, the heuristic function $h(n)$ is consistent.

3.4 Algorithm design

In this section we describe our implemented algorithms, which mainly differ to the original ones for the MAPF with reservation table considering continuous time (see Section 3.4.1) and considering kinematic constraints in continuous time. That is followed by the description of our implemented path-planning algorithms in terms of their differences from the existing ones for the MAPF problem, including WHCA*, FAR, BCP, OD&ID and CBS. Finally, a method for resolving deadlocks is described.

3.4.1 Data structure

We describe in this subsection the reservation table, which is introduced in [35] for solving the MAPF problem. According to Silver, the impassable space-time regions are marked in the reservation table with the form (x, y, t) and they are stored in a hash table with random keys. Note that x, y are coordinates and t is a timepoint. The left-hand graph in Fig. 3.5 illustrates the reservation table used in [35], where the reservations of robot 1 and 2 are marked with green and blue colors. These impassable regions should be avoided during searches of the next robots. The node w is blocked if one robot waits at it. If one robot goes through the arc (w_1, w_2), then its corresponding nodes w_1 and w_2 are blocked until the robot goes past the arc. Hence, the distance for robots traveling in a convoy is at least one arc.
We adopt the reservation table to store occupied regions, because it is a sparse data structure, which considers space and time. Since our problem does not consider discrete and constant timepoints, a different implementation compared with [35] is illustrated in the right-hand graph of Fig. 3.5. For each node in the graph, one interval tree (see [8], pp. 350–357) is generated, if a robot waits at it or goes through it (this is also called lazy initialization). Also, a set of intervals is included in each interval tree and each interval has its starting time t_s and ending time t_e. The search for one interval (t_1, t_2) at node w and the test of overlapping can be realized using a binary search with runtime $O(\log n)$.

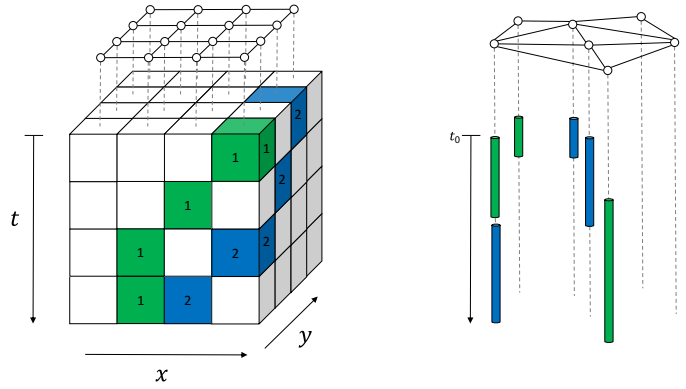


Figure 3.5 Data structure for reservations in MAPF of [35] vs. MAPFWR path planning

An example of the reservation for a robot in the reservation table is shown in Fig. 3.6. The robot begins with node w_2 at time t_0 and ends with node w_6. Once the robot travels between two nodes, both nodes are blocked (vertical bars). Moreover, it takes longer time to speed up and slow down. This data structure is used in all path-planning algorithms in the following subsections.

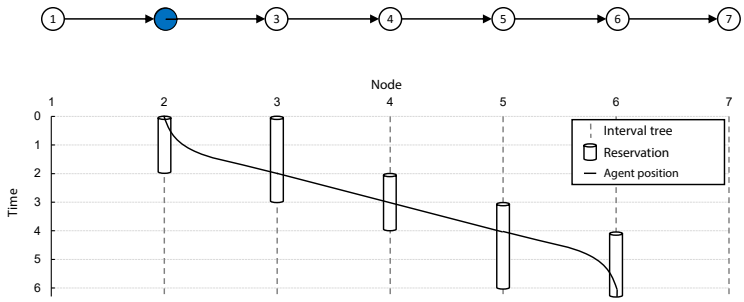


Figure 3.6 An example in the reservation table for the path from w_2 to w_6

3.4.2 WHCA*

The main difference to the existing WHCA*, applied in MAPF in [35], is the modified reservation table (see Section 3.4.1). Moreover, we use another heuristic to calculate costs individually for our problem. We describe below the adjusted WHCA* for the MAPFWR problem.

We have an arbitrary robot r, which is located on the node w_s on time t in the graph, and its goal is the node w_e. The A*-algorithm finds an optimal path for this robot from w_s to w_e, if the heuristic function $h(w_s)$ is admissible. For now, we assume

that the nodes in the graph are equal to the states in the search space; moreover, we do not consider the other robots and wait actions. $h^{2D}(w_s)$ is calculated with Eq. (3.14).

$$h^{2D}(w_s) := t^D\left(r, d^E(w_s, w_e), \overrightarrow{A}_r, \overline{V}_r, \overleftarrow{A}_r\right) \tag{3.14}$$

where the time is estimated by calculating the driving time for the euclidean distance while considering acceleration and deceleration times. Thus, it is impossible for a robot to cover the distance between the two waypoints in shorter time. We use this heuristic function $h^{2D}(w_e)$ in the Reverse Resumable A* (RRA*) algorithm, which searches the path from w_e to w_s in $\mathcal{G}^{-1}$; therefore the heuristic function $h^{RRA*}(w_e)$ is equal to $h^{2D}(w_e)$. $g^{RRA^*(w_e)}$ is the function that calculates the sum of time for rotation and moving from w_e to w_s like discussed above. This algorithm ends when w_s enters the closed set C^{RRA^*} and we do not consider the rotations of w_s and w_e here. As explained in [35], the reason why we use RRA* in WHCA* is that the result of RRA* provides better assumption for the paths without collisions. It is a lower bound of the solutions found in WHCA*, because the solution found in RRA* is optimal for a single agent. If w_s enters the closed set C^{RRA^*} then the value of the heuristic function is equal to the value of the cost function of RRA* to the goal node; otherwise the search continues until $w_i \in C^{RRA^*}$. The heuristic function $h^{WHCA^*}(w_s)$ used by WHCA* can now be defined like in Eq. (3.15).

$$h^{WHCA^*}(w_s) = \begin{cases} g^{RRA^*}(w_s) & \text{if } w_s \in C^{RRA^*} \\ RRA^*(w_e, E^{-1}, w_s, g^{RRA^*}, h^{RRA^*}) & \text{other cases} \end{cases} \tag{3.15}$$

In the following, we describe two variants of WHCA*, namely volatile and non-volatile WHCA*. The first one was used in [35] with some modifications. However, this variant is time-consuming, since the calculated paths are not stored for the next execution and it is suitable for the case where the moving obstacles change the graph in the next execution, such as in a computer game. In our case, the obstacles are deterministic and predictable. Therefore, we develop the latter, non-volatile WHCA*, in Section 3.4.2.2.

3.4.2.1 Volatile WHCA*

Algorithm 3.1 shows how volatile WHCA* works. Let $\mathcal{B}' \subset \mathcal{B}$ be a subset of pods, which are not carried by robots, while $\mathcal{R}' \subset \mathcal{R}$ is a subset of robots, which have subtask *move*. Also, the priority p_r^O for each robot r is set to 0 at the beginning of the algorithm (line 1). The search repeats until a path π_{rt} is found for each robot r at time t or the number of iterations i reaches the given iteration limit I. In each iteration, the reservation table r^T is initialized with fixed reservations (using

the function $f^R(\mathcal{R}')$). These are called "fixed", because for each already moving robot $r \in \mathcal{R}'$ there are nodes that are required to be reserved until r reaches its next planned stop. Then, the robots are sorted based on three criteria (see the sort-function $f^S\left(\mathcal{R}', p_r^O\right)$). The first one is the priority of the robot, while the second one is to check whether a robot carries a pod. These robots are preferred because they cannot move beneath other pods, i.e. they have fewer paths available to get to their goals. The last criterion is the distance towards the goal, preferring robots nearer to their goal. The set $\mathcal{J}$ contains all blocked nodes. The positions of the robots $r \in \mathcal{R} \setminus \mathcal{R}'$ are blocked and stored in the set $\mathcal{J}$. Additionally, if the robot r is carrying a pod (i.e., the function $f^C(r,t)$ is true), then the nodes all other pods $b \in \mathcal{B}'$ are stored at are blocked and stored in $\mathcal{J}$ as well. Once the destination of the robot r has been changed, RRA* should be recalculated with the actual set $\mathcal{J}$ (see line 9). Also, wait actions are added based on its priority (through the function $f^W\left(r, \lfloor 2^{p_r^O - 1} \rfloor\right)$). After that a new path is expanded using A^*_{ST}, which considers space and time (see Section 3.3). If a path was found ($\pi_r \neq \emptyset$), then the reservation table r^T is updated ($f^{AR}\left(\pi_r, r^T\right)$), otherwise the priority of this robot r should be increased (see line 12).

Algorithm 3.1: $WHCA^*_v(t, w, \mathcal{R}', \mathcal{B}', I)$

```
1  foreach r ∈ R' do p_r^O ← 0
2  i ← 1
3  while i ≤ I ∨ ∃r ∈ R' : π_r = null do
4      r^T ← f^R(R'), f^S(R', p_r^O)
5      foreach r ∈ R' do
6          J ← f^P(R \ R')
7          if f^C(r,t) then J ← J ∪ f^P(B')
8          if f^DC(r) then
9              RRA*(r, J), π_r = f^W(r, ⌊2^{p_r^O − 1}⌋)
10             π_r = π_r ⋃ A*_ST(t, w, r, r^T, J, g^{WHCA*}, h^{WHCA*})
11         if π_r ≠ ∅ then f^{AR}(π_r, r^T)
12         else p_r^O ← p_r^O + 1
13     i ← i + 1
14 return π
```

For each robot, we search a sequence of actions in the search graph to get to the goal without a collision. We assume that a robot stops at node w_i and then we generate the following states in the search graph. They are only generated, if they do not block any robots. Moreover, the states, which are in conflict with existing

reservations in the reservation table, cannot be generated as well. But we consider the reservation table only within time window w; after that time window, the rest of the paths calculated by RRA^* are used. We search a sequence of actions for each robot iteratively; if we cannot find a sequence of actions for a robot without conflicts, then the priority of that robot is increased and we restart the search.
Fig. 3.7 shows an example of evasion, where robot r_1 tries to move from s_1 to g_1, while robot r_2 tries to move from s_2 to g_2. Also, we follow the criteria of sorting that we discussed above. Moreover, we assume in this example that the time to go through an arc is one timepoint. The time window w is 10 in this example. We begin with the robot r_1, and we get the following reservation with the form [beginning time, ending time]: [0,1] for w_3; [0,2] for w_2; [1,3] for w_1. For the robot r_2 we do not get any reservations without collision, since [0,1] should be reserved for it. Therefore, we increase the priority of r_2. Here, we get a deadlock, because each robot tries to reach another's beginning node. To solve this deadlock, the robot with the higher priority should at first choose the wait action. According to line 9 in Algorithm 3.1, the length of the wait robot grows exponentially with the priority. In this example, r_1 in the fourth iteration gets the priority 1 and r_2 gets the priority 2. Therefore, the reservation of r_2 is done as follows: [0,2] for w_2; [1,3] for node w_3; [2,3] for w_4. After that, r_1 can find a path without collisions, namely the path 3-5-3-2-1. In another case, if we try the robot r_2 in the first iteration, then the deadlock can be solved as well.

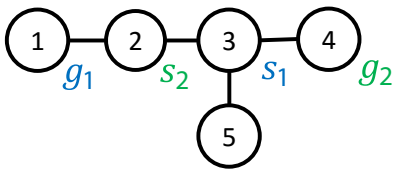


Figure 3.7 An example of evasion, where robot r_1 tries to move from s_1 to g_1, while robot r_2 tries to move from s_2 to g_2

3.4.2.2 Non-volatile WHCA*

As mentioned before, the volatile variant in the previous subsection has a problem in that the existing path and reservation for each robot are recalculated in each execution of the algorithm. This occurs already if only one robot with a move action does not have a path. Instead of that, the non-volatile variant of WHCA* stores the existing path and reservation for each robot, which brings a much shorter runtime. However, reusing the existing paths might cause a problem for generating new paths, since we cannot do any modification of existing paths to adopt the new ones. A comparison between the volatile and non-volatile variants of WHCA* can

be found in the next section. Alg. 3.2 shows the process of the non-volatile variant. The set $\mathcal{R}' \subset \mathcal{R}$ is a subset of robots, which has subtask *move* and does not have a path yet. First of all, they are sorted analogously to how they are sorted in the volatile variant (function $f^S(\mathcal{R}')$). Also, the reservation table r^T is reorganized with the function $f^{RR}(r^T)$. This means that all future reservations of the robots in need for a new path are dropped (see Section 3.4.1). Then, we find a path for each robot iteratively (similar to the volatile variant). Moreover, we check at the end of each path search for the robot r whether a final reservation is possible (function $f^{AFR}(r)$). A final reservation blocks the nodes, where the last action occurs. Hence, a path with only a sequence of wait actions is always possible. By doing this, we can find a solution without prioritizing.

Algorithm 3.2: $WHCA^*_n(w, \mathcal{R}', \mathcal{B}', I)$

```
f^S(R'), f^RR(r^T)
foreach r ∈ R' do
    J ← f^P(R \ R')
    if f^C(r, t) then J ← J ∪ f^P(B')
    if f^DC(r) then RRA*(r, J)
    π_r = π_r ⋃ A*_ST(t, w, r, r^T, J, g^WHCA*, h^WHCA*), f^AR(π_r, r^T), f^AFR(r^T)
return π
```

Now we apply this algorithm to the example in Fig. 3.7. The final reservation does not guarantee to find a path for the robot r_1, since w_2 is blocked for r_2. Such conflict might often occur if we consider a large number of robots. Therefore, we increase the heuristic costs of the node, which is on the shortest path of another robot, for robot r (see Eq. 3.16). So the robot r_1 does not stay at w_3 but at w_5, since w_3 is on the shortest path of r_2, and a higher heuristic cost is considered on w_3.

$$h_p^{WHCA^*} = h^{WHCA^*}(w_i) + c_p|\{r' \in \mathcal{R}' | r' \neq r \wedge w_i \in \pi^{RRA^*}(r', w_r)\}| \tag{3.16}$$

3.4.3 FAR

The FAR algorithm from [44] was used to calculate paths in a flow-annotated search graph for the MAPF problem. Each grid graph can be converted to a flow-annotated graph, but some rules should be held to ensure the connectivity of the graph (see [44]). The flow-annotated graph is necessary for the FAR algorithm to detect deadlocks and to resolve them. We use a warehouse layout as in [23], which contains some properties of a flow-annotated graph (see Section 3.5.1.1). The contribution here is different evasion strategies designed for the MAPFWR, which will be described latter in this section.

Wang and Botea's idea in [44] was to use the A*-algorithm to find paths for robots for as long as possible without collisions. They modify the reservation table of [35] to reserve a number of steps ahead von the generated paths. Instead, we use the reservation table shown in Section 3.4.1. Algorithm 3.3 shows how FAR works. Similarly to WHCA*, the paths of all robots $r \in \mathcal{R}'$ with subtask *move* are planned iteratively. Recall that $\mathcal{B}'$ is the set of pods, which are not being carried by robots. Initially, the reservation table r^T stores all existing reservations of all robots to their next nodes and final reservations for these (returning from the function $f^{FR}(\mathcal{R}')$). As soon as a path is generated for a robot r, the final reservations in r^T will be deleted (function $f^D\left(r, r^T\right)$). The algorithm RRA^* is restarted with the actual block nodes in $\mathcal{J}$. From the resulting path the first *Hop* is extracted. The *Hop* is a sequence of actions, which include *rotate*, *move* from a starting node and *stop* at an ending node. This ending node is considered as the next beginning node for the recalculation of the path. This differs from the original idea of FAR, which stops at each node and recalculates the path there. Since we consider the physical properties of robots, it makes more sense to use the *Hop*. The function $f^{HOP}\left(RRA^*(r, \mathcal{J}), r^T\right)$ returns the *Hop* (line 5). If the *Hop* does cause collisions, it is shortened until it is possible to submit it to the reservation table. If the *Hop* does not contain any nodes, then the returning value r' is the robot who blocks the next node. If no path is found in RRA* ($\pi_r = \emptyset$), then the robot r' is considered as the next r. If it is the first time, then the robot r will wait for a given time interval (a wait action is added to the path π_r through the function $f^{AW}(\pi_r)$). The relation (r, r') means that a robot r waits for another robot r'. Such a relation is established if no *Hop* is found for the robot r and removed again as soon as one is found. If there is a circle of the transitive closure of the relation from r (returning from the function $f^{RC}(r, r')$) or the robot r waits more than the given waiting time T^W, then an evasion strategy $f^E(r)$ should be called. The evasion strategy returns an alternative path without wait actions. In the following subsections, two new evasion strategies are described. At the end of the FAR algorithm, the final reservations will be stored in the reservation table again (using function $f^{AFR}\left(r^T\right)$).

3.4.3.1 Evasion strategy 1: rerouting

In this strategy we try to find a new path for a robot r to escape from a deadlock. The node, at which the next blocked robot stands, is stored to $\mathcal{J}$. Then, a new path is generated again. If another robot is blocked on this new path, then this rerouting starts again. The number of calls for rerouting is limited by a given number. If the number of calls exceeds that given number or no path is found due to the blocked nodes, then this robot r has to wait for a given time period. This method is called FAR_r in the remainder of this paper.

Algorithm 3.3: $FAR(t, w, \mathcal{R}', \mathcal{B}', T^W)$

```
1  r^T ← f^FR(R')
2  foreach r ∈ R' do
3      f^D(r, r^T), J ← f^P(R \ R')
4      if f^C(r, t) then J ← J ⋃ f^P(B')
5      (π_r, r') ← f^HOP(RRA*(r, J), r^T)
6      if π_r = ∅ then
7          if f^GA(π_r) ≠ wait then f^AW(π_r)
8          else
9              if f^RC(r, r') ∨ t > T^W then f^E(r)
10     f^AFR(r^T)
11 return π
```

3.4.3.2 Evasion strategy 2: evasion step

Wang and Botea explain that a deadlock occurs if all four nodes of a square are occupied by robots, and each robot wants to exchange nodes with each other. In the flow-annotated graph, each robot has two arcs to leave the actual node. If one arc causes the deadlock with other robots, then another arc should be chosen. In our case, we choose a random arc, which does not cause any deadlocks. After the robot has gone through that selected arc, it should wait for a random time between 0 and T^W. If that arc does not exist, then the robot should simply stay and wait for a given time. This strategy is called FAR_e in the remainder of this paper.

3.4.4 BCP

Geramifard et al. describe BCP as a meta-algorithm for the path planning of each robot. The main difference to the original algorithm is the reservation table shown in Section 3.4.1. In the original algorithm, a hash table is used to detect collisions of a discrete time-horizon. In our modified version the reservation table stores continuous time intervals per robot and searches for collisions using binary search. As shown in Algorithm 3.4, each robot with the subtask *move* is initialized with a heuristic function h as shown in Eq. (3.14). BCP changes the function until a runtime limit T^R is reached or paths without any collisions are found. In each iteration, the reservation table is initialized with fixed reservations. A path is determined by A^*_S for each robot in the two-dimensional search space (without considering time) while considering blocked nodes. If there is no path found for a robot, then the robot has to wait for a given time period (line 8). As long as a path is found for the robot r, the reservations will be stored in the reservation table. If the reservation is not possible,

since another robot r' already has a reservation of this node w, then the heuristic function will be modified to penalize the value of robot r on w. This and causes the search to consider alternative paths. If there are multiple collisions (detected through the function $f^{DC}\left(r, r^T\right)$), then the first one is always chosen by function $f^{GC}\left(\pi_r, r\right)$. Also, the cost of h_r^{BCP} is updated (function $f^C\left(c, h_r^{BCP}\right)$). If BCP stops due to a reached runtime limit, at least one robot's path contains a collision. This collision is detected during simulation, further execution of the path is stopped and BCP is called again. Hence, we need a lower bound of runtime between two calls, such that the robot can wait at the node. The deadlock can be solved with the method shown in Section 3.4.7.

Algorithm 3.4: $BCP(t, w, \mathcal{R}', \mathcal{B}', T^R)$

```
1  foreach r ∈ R' do h_r^BCP ← h
2  while t ≤ T^R do
3      r^T ← f^R(R'), c ← ∅
4      foreach r ∈ R' do
5          J ← f^P(R \ R')
6          if f^C(r, t) then J ← J ∪ f^P(B')
7          π_r ← A*_S(r, J, g, h_r^BCP)
8          if π_r = ∅ then f^AW(π_r)
9          else
10             if f^DC(π_r, r) then c ← f^GC(π_r, r), f^C(c, h_r^BCP)
11     if c = ∅ then return π
12 return π
```

3.4.5 OD&ID

When determining the next action for all agents the grade of the search space is $O(b^k)$, where k is the number of robots and b is the number of actions. As mentioned in Section 3.2, intermediate states are introduced in [36] to reduce the number of generated states in this algorithm. We describe in the following subsections the modified operator decomposition and independence detection for MAPFWR.

3.4.5.1 Operator decomposition (OD)

One of the main modifications of OD for MAPFWR is to consider continuous times. Hence, full-value states cannot be reached in OD. Each action of a robot requires an individual time interval. Therefore, it is possible that from one state to its following

state, all robots are located on different timepoints. So we define for our problem that each robot r has an action in state n before a time stamp t_{rn} is reached. In the initial state n_0, all robots have a set of empty actions and the time stamp t_{r0} for each robot r is equal to 0. From one state to its following state, the time stamp is increased with the time of the action, which the robot selects, while the time stamps of the other robots remain unchanged. $g(n)$ is the sum of times from n_0 to n, while $h(n)$ is the sum of estimated values for all robots to reach their goal nodes. These values are calculated by RRA* without considering collisions (similar to WHCA* in Section 3.4.2).

In the initial state, a robot is chosen randomly to generate the next state, since the time stamps of all robots are set to 0. In other cases, the robot with the lowest time stamp is chosen to generate the next state. The following state is generated for each action that does not cause any collisions. The reservation table is also used here to detect collisions. The reservation table is initially empty and gets populated each following state of this robot by the reservations of all other robots that end after t_{rn}. In order to generate paths without collisions, a final reservation is required after the last reservation. For each possible action of a robot, the resulting reservation is tested for collisions. If there is no collision, then the time stamp is increased correspondingly and the following state is generated. Finally, the reservation table is emptied and the following state is considered. If all states are generated for a robot, then the A*-algorithm searches for the next state without any successors. After this selection, the robot with the lowest time stamp will be selected and the following state is generated, and so on.

The OD is not suitable for real-time problems, therefore, several stopping criteria are defined to limit the runtime. First of all, the OD stops if a state is reached where all robots arrive at their goal nodes. It also means that all paths are free of collisions. Secondly, the OD also stops if a path is found for each robot within a predefined time limit. This is similar to the time limit of WHCA*, but the searches of following states in OD are simultaneous, not iterative. The third stopping criterion is that the maximum number of expanded states is reached. In the second and third stopping criteria, the state is selected from a search space $\mathcal{S}' \subset \mathcal{S}$, which is defined in Eq. 3.17,

$$\mathcal{S}' = \left\{ n \in \mathcal{S} \mid \max_{r \in \mathcal{R}} t_{rn} \geq \frac{\max_{n' \in \mathcal{S}} \max_{r \in \mathcal{R}} t_{rn'}}{2} \right\} \tag{3.17}$$

where the set $\mathcal{S}'$ includes all states, whose selected paths reach into the second half of the expanded time interval.

For better understanding of OD, the search space of A* with OD is shown in Fig. 3.8 for the same example that is illustrated in Fig. 3.7. Recall that a robot r_1 begins with w_3 and ends with w_1, while a robot r_2 begins with w_2 and ends with w_4. Let $\overrightarrow{A}_r$, $\overleftarrow{A}_r$ and $\overline{V}_r$ be 1 for all $r \in \mathcal{R}$. Moreover, the length of each arc is also equal

to 1. Therefore, the moving time can be calculated as the number of the arcs plus 1. The time of a rotation by 360° also is 1. The length of a wait action here is 5. The final reservation is used in this example as well. In the initial state n_0, the moving time of both robots is 3, and r_1 has to rotate by 90°. Therefore, we get the value of $f(n_0)$ 6.25. There, the time stamps of r_1 and r_2 are equal to 0, so we can select one of them randomly. In this example we firstly choose r_1, which has three possible actions, moving to w_5 (n_1) or w_4 (n_2) or w_5 (n_3). However, r_1 cannot reach w_2 through the final reservation. Also, the cost of n_2 is cheaper than n_1. Thus, n_2 is selected as the expanded state. In this state, $t_{r_2 n_2}$ is equal to 0, therefore the actions for r_2 will be selected. For this robot, the reservation of the move action on w_3 is $[0, 2]$ and the final reservation on w_4 is $[0, \infty)$. Thus, r_2 can either move to w_1 or wait. In this example, we can see the influence of the length of the wait action on the solution. If the length is too small, then there are too many states and the second stopping criterion is reached without generating a good solution. On the contrary, the expansion of state n_1 can happen.

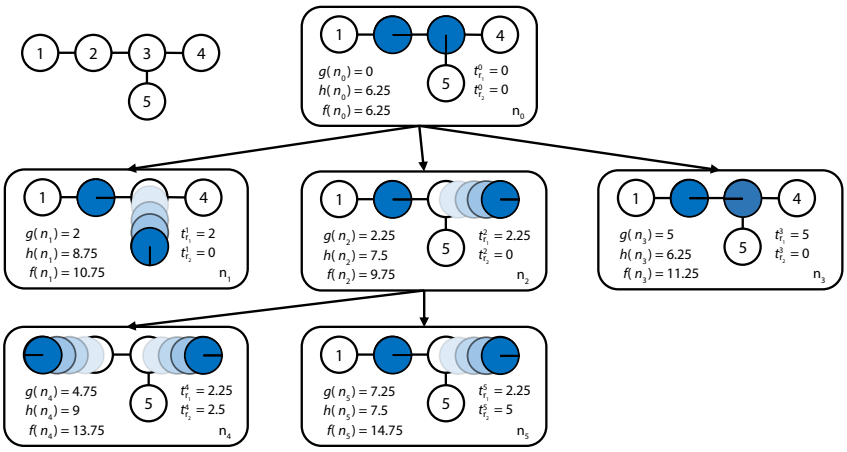


Figure 3.8 The search space of A* with OD for the case where a robot r_1 begins with w_3 and ends with w_1, while a robot r_2 begins with w_2 and ends with w_4

3.4.5.2 Independence detection (ID)

The runtime of OD increases exponentially with the number of robots (see [36]). Therefore, Standley introduced ID, which considers disjoint subsets of robots. This means the paths can be planned for each subset of robots independently. Initially, each robot is considered as its own group and for each robot a path is generated with OD. In each iteration, the generated paths of all groups are tested to determine whether they are free of collision. If a collision occurs between two groups, these two groups are merged into one group. Then, new paths are generated for the new

groups. This repeats until either all groups can be combined without any collisions or there is only one group remaining.

3.4.6 CBS

The algorithm CBS is introduced by [33], and uses the constraint tree to dissolve conflicts gradually. Each state in the constraint tree has three properties: one constraint, a solution including a set of paths for all robots and the cost. CBS is a meta-algorithm, which uses the components of a path-planning algorithm to find paths in three dimensional search space, while some nodes are blocked for a given time. Therefore, the algorithm A^*_{ST} is used as in WHCA*. The initial state of the CBS is that a path is found for each robot. There, no restrictions about the conflicts, i.e. collisions, are considered. The corresponding cost of such a state is the sum of costs of all paths of all robots. A state is considered as generated, if a path for each robot is found. Such generated states will be selected for expansion, if they contain conflicts between their paths. I.e., a conflict occurs for a path, if the reservation of a robot r_1 for w overlaps with that of a robot r_2 for the same node. A new interval for w is calculated based on the maximum ending time and the minimum beginning time of both reservations. Now, two successor states are generated and restricted using this interval. In the first, a new path is searched for robot r_1, which is free of overlaps with the new calculated interval. Then, the same is done for robot r_2.
It is efficient that the paths and their corresponding costs are calculated once for the initial state, since only the cost of a new path for a robot is updated in each following state, and the paths of other robots remain. The cost of a state is calculated based on a δ-evaluation. And only the new restriction is stored to each following state. In order to hold all restrictions for a state, the tree is traversed from the parent states to the initial state. The same is done to get the path of a state. Recall that, a path for each robot is found in the initial state.
The CBS method is similar to the branch-and-bound algorithm, which includes some selection strategies (see [24]). The best-first strategy was selected in [33]. Additionally, we apply breadth-first and depth-first strategies to find feasible solutions faster, because MAPFWR has to be solved in a real-time environment.
Moreover, we use different stop criterium for MAPFWR. The original CBS algorithm stops, if a collision-free solution is found. A solution for MAPF is optimal if an optimal path-planning solver is used and the best-first search is applied as well. However, such a statement is no longer applicable for the CBS algorithm used to solve our problem, since the restrictions include an arbitrary limit to the length of reservations. Moreover, an early termination is required for the application in a real-time environment, because an optimal solution is rarely found for a large number of potential collisions. Therefore, the runtime of this algorithm is limited. It is possible that no feasible solution is found before termination. In this case,

we return the solution with the longest time interval until the first collision. Right before the collision would occur CBS is called again to determine a new solution.

3.4.7 Resolving deadlocks

All the path-planning algorithms we have developed have a runtime limit and time-out is required between two executions. Therefore, it is possible that no path can be found or the path has only wait actions. Moreover, most of the algorithms above, except FAR_e, are deterministic, thus, these results are repeated at each call, which might cause an interruption of the system. In order to resolve deadlocks, we keep track of the time each robot approached its current node. If the robot does not leave the node within a given time, then we randomly choose a neighboring node that is not blocked or reserved. If such a node exists, it is assigned to the robot as its next goal. Moreover, it is possible that the robot has only one node to select and the method produces a path that includes the original node again. In this case, both nodes are blocked, and other robots do not have a chance to go through them. Therefore, a random waiting time is required, which is evenly distributed in $[0, T^W]$. Thereby, the robots, which stand closely to each other, can move apart.

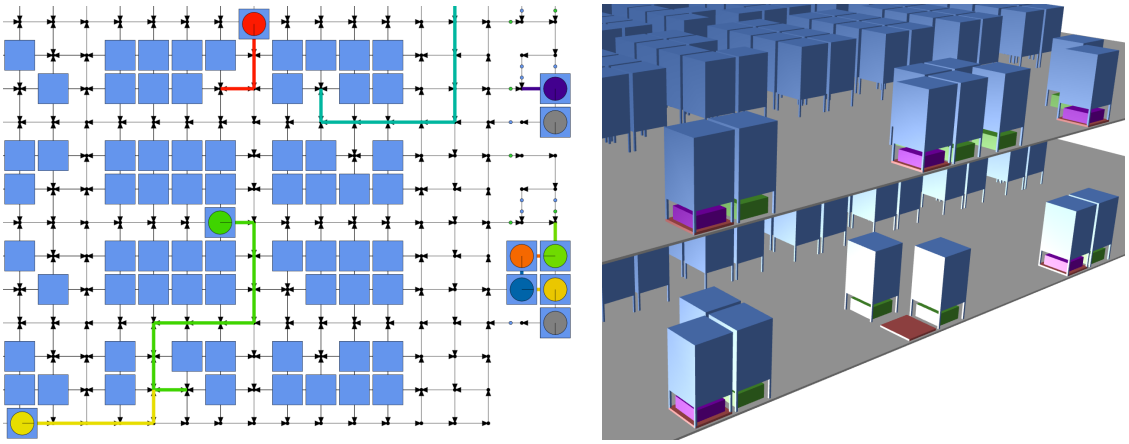
3.5 Simulation study

In this section we first describe our simulation framework while also defining parameters of the experimental setup. Furthermore, we discuss the computational results and compare the different applied methods.

3.5.1 Simulation framework

We use an event-driven agent-based simulation to capture the behavior of RMFS. The 2D- und 3D-visualizations of the simulation are shown on the left and right sides of Fig. 3.9 respectively, where 3D-visualization is shown for two tiers. The details of the layout will be described later in this section. To investigate the effectiveness in conjunction with the path planning methods described in this work we fix the methods for the other decision problems to the following simple policies. This means that orders are assigned to pick stations randomly (*order assignment*), bundles are assigned to replenishment stations randomly (*replenishment assignment*), bundles are assigned to pods randomly (*bundle storage assignment*), pods are send to random free storage locations (*pod storage assignment*), pods are selected for picking by the number of requests that can be completed with them (*pick pod selection*) and the robots work for all stations equally, but with a preference for pick stations (*task allocation*). As mentioned in Section 3.2.1, all of these decisions ultimately result in

simple requests for the robots to complete; e.g., an order present at a station results in a demand for a certain item. This requires a suitable pod to be brought to this station. Thus, a path from its position to the pod and further on to the station needs to be generated for the robot executing the task.



(a) Detailed 2D view **(b)** 3D overview

Figure 3.9 Screenshots of the simulation visualization

Due to the focus on path planning a backlog of orders of constant length is available at all times, i.e., a new customer order is generated as soon as another one is finished. Hence, the system is being kept under pressure and robots may always have a task to execute. The same is done for generating new bundles of items such that the inventory does not deplete. The number of SKUs is also kept low (100 SKUs) to reduce the risk of stock-outs and to maintain a more stable system during the simulation horizon.

For each of the methods mentioned above a path planning engine is implemented as a wrapper that acts like an agent of the simulation. It is responsible for passing necessary information about the current state of simulation to the methods and coordinates the calls to the planning algorithms. In the update routine of the path planning agent (see Alg. 3.5) first the reservation table is reorganized to remove past reservations and then the embedded path planning algorithm is executed. This is only done, if there is no ongoing timeout and there is at least one robot requesting a new path. A robot will request a new path, if it is assigned to a new task with a new destination or the execution of its former path failed. Like mentioned before, the execution of a path may fail, if paths that are not collision-free are assigned to the robots. In this case the path planning engine will abort the execution of the path to avoid an imminent collision. For this reassurance again the reservation table is used.

Algorithm 3.5: *Update*$(t, \mathcal{R}, r^T, T^T)$

```
1 f^RR(r^T)
2 if (t' + T^T < t) ∧ (∃r ∈ R : π(r) = ∅) then
3     ExecutePathPlanner()
4     t' ← t
```

3.5.1.1 Layout

The layout of the instances used for the experiment are built by a generator based on the work of [23]. Mainly three different areas can be identified within the layout (see Fig. 3.10). First, an inventory area is built, which contains all pod storage locations $\mathcal{W}^{SL}$ (indicated by blue squares). These are created in blocks of eight waypoints and connected by other waypoints by bidirectional edges. The waypoints of the aisles in between are connected by directional edges such that a cycle emerges around each block up to the complete inventory area. The directions of the edges are denoted by arrows. Hence, a robot, which is not carrying a pod, can almost freely navigate below stored pods, while a robot carrying a pod must adhere to a certain cyclic flow. This area is surrounded by a hall-area (long dashes) that serves as a highway for robots traveling from storage locations to stations. That offers some space to reduce congestion effects in front of the stations. The last area (short dashes) is used for buffering robots inbound for the stations. This area also contains all replenishment (yellow circles) and pick stations (red-circles). Each station has its own queue that is managed by a queue manager instead of path planning; i.e., as soon as a robot enters this area path planning gets deactivated and instead a queue manager is responsible for moving up robots towards the station while also exploiting shortcuts, if possible.

3.5.2 Experiment setup

For the experiment we use 10 instances with different layouts (see Table 3.3). The names of the instances are derived from the number of tiers, replenishment stations, pick stations, robots and pods. All of them adhere to the layout described above, but with minor modifications. The instance 1-12-20-128-1965 uses a layout that surrounds the inventory area with the hall and buffer areas leading to more stations compared to the storage locations. Furthermore, instances 2-8-8-64-1100 and 3-12-12-96-1650 contain two, respectively, three floors connected by elevators. The elevators are positioned similar to the stations and transport one robot at a time from one floor to the next one in 10 seconds. Instance 1-6-14-106-1909 is shaped like a 'L' while 1-6-16-146-2726 contains 'holes' emulating obstacles of a warehouse

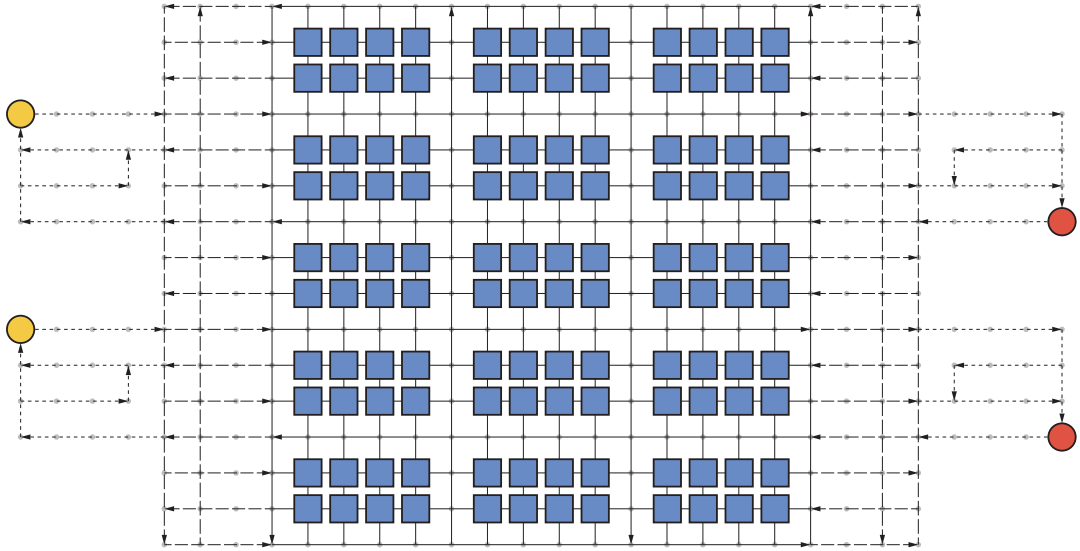


Figure 3.10 Basic layout of instances built by the generator

building structure in its waypoint graph. The instances 1-4-16-144-1951 and 1-1-3-48-795 have all their stations positioned at only one side. The other instances use a form of the default layout described above in different sizes. For all instances roughly 85% pods are used when compared to available storage locations. This allows for more options when dynamically determining a storage location each time a pod is brought back to the inventory. Additionally, we provide the ratio of robots per station. This is useful as a first intuition for the potential of congestion effects, i.e.: the more robots are used in less space the more conflicts may occur. Furthermore the number of waypoints in the graph is shown. Note that these are all waypoints of the system, including the storage locations shown separately as well as other special purpose waypoints (e.g. the ones used for stations and their queues).

All robots of the experiment share the same acceleration and deceleration rate of $0.5\frac{m}{s^2}$ and a top-speed of $1.5\frac{m}{s}$. The time needed for a full rotation is set to $2.5s$. Robots and pods are emulated as moving circles with a diameter of $70cm$, respectively $90cm$. The times for picking up a pod, setting down a pod, storing a bundle of items (T_m^I) and picking a single item (T_m^O) are all constant and set to $3s$, $3s$, $10s$ and $10s$, respectively. The following default parameters are used for the different methods, which are chosen according to the results of a preceding grid search on a small subset of possible parameter values. For all methods the length of a wait step is set to $2s$ while the timeout for a single path planning execution is set to $1s$, i.e., the path planning algorithm is called at most once per second. For WHCA$^*_v$ and WHCA$^*_n$ the time window is set to $20s$ and $30s$, respectively. For BCP the biased

Table 3.3 Characteristics of the instances used in the experiment

Name	tiers	bots	bots per station	pods	storage locations	way-points	repl. stations	pick stations
1-1-3-48-795	1	48	12.0	795	936	2112	1	3
1-4-4-32-550	1	32	4.0	550	648	1640	4	4
1-4-16-144-1951	1	144	7.2	1951	2296	5528	4	16
1-6-14-106-1909	1	106	5.3	1909	2248	5654	6	14
1-6-16-146-2726	1	146	6.6	2726	3208	8120	6	16
1-8-8-64-1040	1	64	4.0	1040	1224	3064	8	8
1-8-8-96-1502	1	96	6.0	1502	1768	4104	8	8
1-12-20-128-1965	1	128	4.0	1965	2312	6088	12	20
2-8-8-64-1100	2	64	4.0	1100	1296	4144	8	8
3-12-12-96-1650	3	96	4.0	1650	1944	6216	12	12

cost is set to 1. The search method used for CBS is best first. The maximal node count for OD&ID is set to 100.
For the assessment of performance we first look at the sum of item bundles stored and units picked at the replenishment and pick stations (handled units). This metric also resembles the work that is done by the system during simulation horizon, which relates to the throughput being a typical goal for such a parts-to-picker system. Hence, the implied goal for path planning is to generate paths that can be executed very fast such that the stations wait for robots bringing pods as little as possible. For more detailed insights we added the average length of the computed paths and the average time it took the robots to complete them (trip length & trip time). At last we look at the wall-clock time consumed by the different methods (wall time).

3.5.3 Simulation results

In the following we discuss the computational results of the experiment described above. Each combination of method and instance is simulated for 24 hours with 10 repetitions to lessen the effect of randomness caused by other controllers and simulation components. In table 3.4, the arithmetic mean of the proposed metrics is given per method and across all instances. Additionally, the timeout of $1s$ per path planning execution implies that the overall wall-clock time usable by a method is limited by the simulation horizon of $86.400s$. In table 3.5 the handled units of the methods in average per instance are shown. This is done to allow further insights about the performance of the method related to the instance characteristics.
In terms of handled units, WHCA$^*_v$ is the most successful one, followed by BCP. We can also see that the trip time highly relates to the number of handled units, i.e. the

Table 3.4 Performance results for the different methods and metrics (averages across repetitions and instances, ordered by handled units)

Method	Handled units	Trip length (m)	Trip time (s)	Wall time (s)
WHCA$^*_v$	81011.43	52.76	67.61	9555.28
BCP	80469.76	53.69	69.53	81027.86
FAR$_r$	78260.82	54.17	71.07	1272.06
WHCA$^*_n$	77326.95	53.58	71.27	2184.29
OD&ID	75380.12	52.48	72.94	14780.25
FAR$_e$	71942.03	54.91	77.51	473.72
CBS	60205.38	53.31	90.21	65560.65

Table 3.5 Handled units per instance and method (averages across repetitions, green ≡ best / red ≡ worst per row)

Instance	WHCA$^*_v$	BCP	FAR$_r$	WHCA$^*_n$	OD&ID	FAR$_e$	CBS
1-1-3-48-795	30332	30186	30955	30359	29641	30763	29437
1-4-4-32-550	29820	28787	28230	28835	28357	27481	29669
1-4-16-144-1951	142482	139722	136946	124404	118436	94683	53156
1-6-14-106-1909	126878	124571	124391	124412	122263	121788	86965
1-6-16-146-2726	149142	146119	147903	147188	144540	145789	112812
1-8-8-64-1040	50313	49174	47456	47767	47299	44742	48334
1-8-8-96-1502	62120	61939	57862	58693	56857	53700	46633
1-12-20-128-1965	85463	82024	80497	80870	77644	74796	60705
2-8-8-64-1100	54383	53026	51944	52907	52021	50756	54188
3-12-12-96-1650	79183	78940	76425	77836	76744	74923	78968

shorter the time is for completing a trip the more units are handled overall. This does not hold for the trip length, i.e. the length of the trips lies in a close range, but a shorter one is not necessarily faster. This is mainly impacted by the more important coordination of the robots. Thus, shorter trips may cause more congestion and longer waiting times for the robot while it is executing a path. The wall time consumed by the methods differs significantly. While FAR$_e$ in average uses less than ten minutes to plan the paths for all robots for 24 hours, going from 85.57s for the 1-4-4-32-550 layout up to 1008.59s for the 1-6-16-146-2726 layout. In contrast, BCP almost always uses the complete allowed runtime. This means that BCP does plan reasonably efficient paths, but is not able to completely resolve all conflicts up until the destination of all of them. This can also be seen when looking at the percentage of executions that ended in a timeout (see Tab. 3.6). It is almost impossible for BCP to generate completely conflict-free paths for the layout instances of quite realistic size before the timeout of 1s. We can only observe this for the smallest instances

of the set. However, the paths successfully planned by BCP until the timeout takes effect are competitive. The FAR methods cause the longest trip lengths, but the FAR_r variant can still compete with the others in terms of handled units and trip time. This is especially interesting when looking at the wall time consumed by it. Hence, the FAR method is a candidate to consider when controlling instances much larger than the ones considered in this work. Furthermore, the strategy of FAR to avoid head to head collisions is working well (in comparison) for instances that are more crowded with robots, i.e. have a higher robot to station ratio (see Tables 3.5). The rather poor performance of CBS is a reason of the method not being able to generate efficient paths within the time limit for the quite large instances of the pool. This even causes cascading congestion effects, if a robot is not assigned any path and will block others even longer. In contrast, we see a good performance of CBS for the smaller instances with a lower robot to station ratio (see Tab. 3.5). The WHCA^*_n variant is still performing well while only consuming roughly 23 % wall-clock time of WHCA^*_v. However, a longer trip time is the result of the robots being forced to plan their trips based on the existing ones without the possibility to find overall improved paths. The method OD&ID achieves reasonable performance while consuming acceptable wall-clock time across all instances.

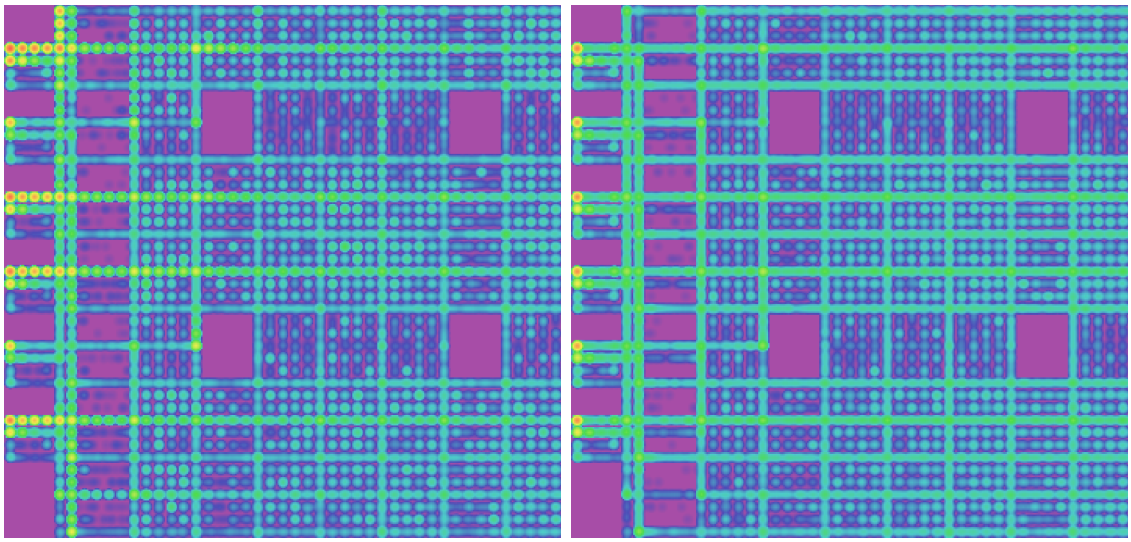
Table 3.6 Extended method comparison

Method	Station idle time	Timeouts	Memory used (MB)	
			average	maximum
WHCA^*_v	44.8 %	7.0 %	120.63	227.62
BCP	46.2 %	97.1 %	86.01	147.76
FAR_r	46.4 %	0.0 %	105.43	193.88
WHCA^*_n	47.0 %	0.0 %	128.98	254.78
OD&ID	48.2 %	0.2 %	99.22	183.13
FAR_e	50.1 %	0.0 %	101.26	193.11
CBS	56.5 %	65.9 %	77.00	125.60

Furthermore, we can observe that the idle time of the station, i.e. the time the station is not busy picking items, respectively not busy storing bundles, is another metric that is closely related to the trip time of the methods (see Tab. 3.6). To some extent, this means that stations will idle less, if robots reach them faster. Regarding the fairly high idle times note that the experiment is designed in a way that increases pressure on the path planning components, i.e. the controllers for the other components are causing longer trips and reasonable times for handling pods at the stations. This is done, because there is a natural upper bound for handling items and item bundles at the stations given by the constant time it needs to process one unit of each. Hence, it is not possible to process more units than given by the following

simple upper bounds, which would lead to a bottleneck obscuring the impact of the path planning components when reached. For pick stations the time for picking one item limits the throughput per hour ($UB^O_m := \frac{3600}{T^O_m}$) while for replenishment stations it is limited by the time for storing one item bundle ($UB^I_m := \frac{3600}{T^I_m}$). Summing up upper bounds of the stations leads to an overall upper bound for handled units for the system ($UB := \sum_{m \in \mathcal{M}^O} UB^O_m + \sum_{m \in \mathcal{M}^I} UB^I_m$). Like mentioned before some of the methods reach the given runtime timeout much more than others. Looking at the average timeouts of the different methods across all instance we can observe that BCP almost always uses its complete runtime given. However, it still obtains results of reasonable quality. In contrast, CBS also reaches it's runtime very often but is not able to obtain efficient paths within time except for the small instances of the set. For the latter also less timeouts occur for CBS. Except for WHCA$^*_v$ all other methods virtually never run into a timeout. Interestingly OD&ID consumes more wall time than WHCA$^*_v$ but faces less timeouts. Since the timeouts per instance are very similar for WHCA$^*_v$ this suggests that certain states during the simulation horizon cause spikes in the wall time consumed that lead to these timeouts. At last, we show the maximal memory consumed by the different method. This is the maximum across all instances. It is obtained by executing a reference simulation for each instance applying a random walk method and subtracting the resulting memory consumption from the memory consumed by the respective method for the same instance. Note that the measurement of the memory underlies inaccuracy caused by technical influences like the garbage collection. However, overall we can see that the memory consumed only differs between the methods in reasonable absolute numbers. Especially when looking at the maximal memory consumption across all instances we can observe that memory is not the limiting factor for the proposed methods considering todays typical hardware.

The impact of path planning on the systems overall performance can also be seen in more detail when comparing the movement of the robots between WHCA$^*_v$ and CBS for a large instance by using the heatmaps shown in Fig. 3.11. The heatmaps show the positions of all robots that are periodically polled throughput the simulation horizon. A logarithmic scale from purple as the coldest color to red as the hottest is used to render the values. Hence, warmer colors indicate areas where robots have been observed more frequently. In the case of CBS the highest conflicting area in terms of congestion are the exits of the stations (stations are positioned all around the inventory area for this instance). When applying WHCA$^*_v$ to the same instance robots are able to leave the stations quickly and spent more time within the inventory area storing and retrieving pods. This is specifically indicated by less “hot dots”, which can especially be observed at intersection waypoints. Overall the movement behavior of WHCA$^*_v$ is more 'fluent' along the main axes from the inventory area towards the stations and back. We can observe similar effects for the other methods



(a) CBS **(b)** WHCA$^*_v$

Figure 3.11 Heat map of the robot positions over time for a part of instance 1-6-16-146-2726 (purple ≡ low, red ≡ high)

and instances. Thus, ensuring a fluent robot movement is an objective of efficient path planning.

3.6 Conclusion

In this work we proposed a generalized problem definition of the MAPF problem that allows the application of methods for the field of Robotic Mobile Fulfillment Systems. Additionally, we have proven that A^* is complete and admissible in the search space of our problem. Further on, we describe necessary modifications to the A^*-based algorithms previously applied to MAPF. Based on the computational results from our simulation framework, WHCA$^*_v$ performed the best according to the proposed metrics, but does not scale as well as the FAR methods in terms of wall-clock time. Similarly, the time consumption of CBS negatively impacted the performance for the large instances in our set. In contrast, FAR is applicable even for very large instances.
In future we want to investigate the combination of the proposed methods with control mechanisms for the other decision components, such as *task allocation*. At this, we expect that methods have mutual dependencies. Hence, effective methods have to be evaluated that cooperate best to achieve a globally efficient system. Furthermore, we observe the impact of the layout characteristics and size on the performance, which suggests a more detailed investigation of the dependencies of these.

Acknowledgements

We would like to thank Tim Lamballais for providing us with the layout concept used for the instances of the experiments. Additionally, we thank the Paderborn Center for Parallel Computing (PC^2) for the use of their HPC systems for conducting the experiments. Marius Merschformann is funded by the International Graduate School - Dynamic Intelligent Systems, Paderborn University.

References

[1] Tuan Le-Anh and René de Koster. "A review of design and control of automated guided vehicle systems". In: *European Journal of Operational Research* 171.1 (2006), pp. 1–23. ISSN: 03772217. DOI: 10.1016/j.ejor.2005.01.036.

[2] Max Barer et al. "Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem". In: *Seventh Annual Symposium on Combinatorial Search*. 2014.

[3] Zahy Bnaya and Ariel Felner. "Conflict-Oriented Windowed Hierarchical Cooperative A∗". In: *IEEE International Conference on Robotics and Automation (ICRA), 2014*. Piscataway, NJ: IEEE, 2014, pp. 3743–3748. ISBN: 978-1-4799-3685-4. DOI: 10.1109/ICRA.2014.6907401.

[4] Adi Botea and Pavel Surynek. "Multi-Agent Path Finding on Strongly Biconnected Digraphs". In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pp. 2024–2030.

[5] Nils Boysen, Dirk Briskorn, and Simon Emde. "Parts-to-picker based order processing in a rack-moving mobile robots environment". In: *European Journal of Operational Research* 262.2 (2017), pp. 550–562. ISSN: 03772217. DOI: 10.1016/j.ejor.2017.03.053.

[6] L. Cohen et al. "Rapid Randomized Restarts for Multi-Agent Path Finding Solvers". In: *ArXiv e-prints* (2017).

[7] Liron Cohen, Tansel Uras, and Sven Koenig. "Feasibility Study: Using Highways for Bounded-Suboptimal Multi-Agent Path Finding". In: *Eighth Annual Symposium on Combinatorial Search*. 2015.

[8] Thomas H. Cormen and Paul Molitor. *Algorithmen - eine Einführung*. 3., überarb. und erw. Aufl. München: Oldenbourg, 2010. ISBN: 9783486590029.

[9] John Enright and Peter R. Wurman. "Optimization and Coordinated Autonomy in Mobile Fulfillment Systems". In: *Automated Action Planning for Autonomous Mobile Robots*. Ed. by Sanem Sariel-Talay, Stephen F. Smith, and Nilufer Onder. 2011.

[10] Esra Erdem et al. "A General Formal Framework for Pathfinding Problems with Multiple Agents". In: *AAAI 2013*.

[11] Ariel Felner et al. "PHA*: Finding the Shortest Path with A* in An Unknown Physical Environment". In: *Journal of Artificial Intelligence Research* 21 (2004), pp. 631–670.

[12] Ariel Felner et al. "Search-based Optimal Solvers for the Multi-agent Pathfinding Problem: Summary and Challenges". In: *AAAI Publications, Tenth Annual Symposium on Combinatorial Search*.

[13] Alborz Geramifard, Pirooz Chubak, and Vadim Bulitko. "Biased Cost Pathfinding". In: *AIIDE*. 2006, pp. 112–114.

[14] M. Goldenberg et al. "Enhanced Partial Expansion A*". In: *Journal of Artificial Intelligence Research* 50 (2014).

[15] Anna Gorbenko and Vladimir Popov. "Multi-agent Path Planning". In: *Applied Mathematical Sciences* Vol. 6, no. 135 (2012), pp. 6733–6737.

[16] Jinxiang Gu, Marc Goetschalckx, and Leon F. McGinnis. "Research on warehouse operation: A comprehensive review". In: *European Journal of Operational Research* 177.1 (2007), pp. 1–21. ISSN: 03772217. DOI: 10.1016/j.ejor.2006.02.025.

[17] Christopher J. Hazard, Peter R. Wurman, and Raffaello D'Andrea. "Alphabet Soup: A Testbed for Studying Resource Allocation in Multi-vehicle Systems". In: *Proceedings of AAAI Workshop on Auction Mechanisms for Robot Coordination*. Citeseer, 2006, pp. 23–30.

[18] Wolfgang Hoenig et al. "Multi-Agent Path Finding with Kinematic Constraints". In: *ICAPS 2016*, pp. 477–485.

[19] A. E. Hoffman et al. "System and method for inventory management using mobile drive units". US20130103552 A1. 2013. URL: https://www.google.com/patents/US20130103552.

[20] R. Jansen and N. R. Sturtevant. "A new approach to cooperative pathfinding". In: *AAMAS '08 Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, pp. 1401–1404.

[21] Mokhtar M. Khorshid, Robert C. Holte, and Nathan Sturtevant. "A Polynomial-Time Algorithm for Non-Optimal Multi-Agent Pathfinding". In: ().

[22] Athanasios Krontiris, Ryan Luna, and Kostas E. Bekris. "From feasibility tests to path planners for multi-agent pathfinding". In: *Sixth Annual Symposium on Combinatorial Search*. 2013.

[23] T. Lamballais, D. Roy, and M.B.M. de Koster. "Estimating performance in a Robotic Mobile Fulfillment System". In: *European Journal of Operational Research* (2016). ISSN: 03772217. DOI: 10.1016/j.ejor.2016.06.063.

[24] Ailsa H. Land and Alison G. Doig. "An Automatic Method of Solving Discrete Programming Problems". In: *Econometrica,* Vol. 28, No. 3 (1960), pp. 497–520.

[25] Ryan Luna and Kostas E. Bekris. "Efficient and complete centralized multi-robot path planning". In: *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2011)*. 2011, pp. 3268–3275. DOI: 10.1109/IROS.2011.6095085.

[26] M. Merschformann, L. Xie, and H. Li. "RAWSim-O: A Simulation Framework for Robotic Mobile Fulfillment Systems". In: *ArXiv e-prints* (2017).

[27] Judea Pearl. *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Pub. Co., Inc., 1984.

[28] Hans Albert Richard and Manuela Sander. *Technische Mechanik - Dynamik: Grundlagen - effektiv und anwendungsnah*. 1. Aufl. Viewegs Fachbücher der Technik. Wiesbaden: Vieweg, 2008. ISBN: 9783528039950.

[29] Kees Jan Roodbergen and Iris F.A. Vis. "A survey of literature on automated storage and retrieval systems". In: *European Journal of Operational Research* 194.2 (2009), pp. 343–362. ISSN: 03772217. DOI: 10.1016/j.ejor.2008.01.038.

[30] M.m Ryan. "Exploiting Subgraph Structure in Multi-Robot Path Planning". In: *Journal of Artificial Intelligence Research* 31 (2008), pp. 497–542.

[31] Malcolm Ryan. "Constraint-based multi-robot path planning: 3 - 7 [i.e. 3 - 8] May 2010, Anchorage, Alaska, USA". In: *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 922–928.

[32] Qandeel Sajid, Ryan Luna, and Kostas E. Bekris. "Multi-Agent Pathfinding with Simultaneous Execution of Single-Agent Primitives". In: *Proceedings of the Fifth Annual Symposium on Combinatorial Search*.

[33] Guni Sharon et al. "Conflict-based search for optimal multi-agent pathfinding". In: *Artificial Intelligence* 219 (2015), pp. 40–66. ISSN: 00043702. DOI: 10.1016/j.artint.2014.11.006.

[34] Guni Sharon et al. "The Increasing Cost Tree Search for Optimal Multi-Agent Pathfinding". In: *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence*, pp. 662–667.

[35] David Silver. "Cooperative Pathfinding". In: *AIIDE*. 2005, pp. 117–122.

[36] Trevor Scott Standley. "Finding Optimal Solutions to Cooperative Pathfinding Problems". In: *AAAI*. Vol. 1. 2010, pp. 28–29.

[37] Nathan Sturtevant and Michael Buro. "Improving Collaborative Pathfinding Using Map Abstraction". In: *AIIDE 2006*, pp. 80–85.

[38] P. Surynek. "A novel approach to path planning for multiple robots in bi-connected graphs". In: *IEEE International Conference on Robotics and Automation*, pp. 3613–3619. DOI: 10.1109/ROBOT.2009.5152326.

[39] Pavel Surynek. *Makespan Optimal Solving of Cooperative Path-Finding via Reductions to Propositional Satisfiability*. 2016. URL: http://arxiv.org/pdf/1610.05452.

[40] Pavel Surynek. "Towards Optimal Cooperative Path Planning in Hard Setups through Satisfiability Solving". In: *PRICAI 2012: trends in artificial intelligence*. Ed. by Patricia Anthony, Mitsuru Ishizuka, and Dickson Lukose. Vol. 7458. Lecture notes in computer science Lecture notes in artificial intelligence. Berlin: Springer, 2012, pp. 564–576. ISBN: 978-3-642-32694-3. DOI: 10.1007/978-3-642-32695-0_50.

[41] James A. Tompkins. *Facilities planning*. 4th ed. Hoboken, NJ and Chichester: John Wiley & Sons, 2010. ISBN: 0470444045.

[42] M. Turpin, N. Michael, and V. Kumar. "CAPT: Concurrent assignment and planning of trajectories for multiple robots". In: *The International Journal of Robotics Research* 33.1 (2014), pp. 98–112. ISSN: 0278-3649. DOI: 10.1177/0278364913515307.

[43] Glenn Wagner and Howie Choset. "Subdimensional expansion for multirobot path planning". In: *Artificial Intelligence* 219 (2015), pp. 1–24. ISSN: 00043702. DOI: 10.1016/j.artint.2014.11.001.

[44] Ko-Hsin Cindy Wang and Adi Botea. "Fast and Memory-Efficient Multi-Agent Pathfinding". In: *ICAPS*. 2008, pp. 380–387.

[45] Ko-Hsin Cindy Wang and Adi Botea. "MAPP: a scalable multi-agent path planning algorithm with tractability and completeness guarantees". In: *Journal of Artificial Intelligence Research* 42 (2011), pp. 55–90.

[46] Boris de Wilde, Adriaan W. ter Mors, and Cees Witteveen. "Push and Rotate: Cooperative Multi-Agent Path Planning". In: *AAMAS 2013 Proceedings of the 12th International Conference on Autonomous Agents and Multiagent Systems*, pp. 87–94.

[47] Peter R. Wurman, Raffaello D'Andrea, and Mick Mountz. "Coordinating hundreds of cooperative, autonomous vehicles in warehouses". In: *AI Magazine* 29.1 (2008), p. 9. ISSN: 0738-4602.

[48] Jingjin Yu and Steven M. LaValle. "Planning Optimal Paths for Multiple Robots on Graphs". In: URL: http://arxiv.org/pdf/1204.3830v4.

Appendix

3.7 Theorem

Theorem 1. *Let $c(n_1, n_2)$ be the cost of the arc (n_1, n_2) in search space S of MAPFWR. Then it holds a finite and positive lower bound δ, $c(n_1, n_2) \geq \delta > 0$.*

Proof. Let r be $argmax_{r \in \mathcal{R}} \overline{V}_r$, T^R be $\frac{\max_{i \in \mathcal{R} \cup \mathcal{B}} \max_{i' \in \mathcal{R} \cup \mathcal{B} \setminus \{i\}} L_i^R + L_{i'}^R}{\overline{V}_r}$ and δ be $min(T^W, T^R)$. So $\delta > 0$ since $T^W \in \mathbb{R}^+$, $\forall r \in \mathcal{R} : L_r^R, \overline{V}_r \in \mathbb{R}^+$ and $\forall b \in \mathcal{B} : L_b^R \in \mathbb{R}^+$. Now we look at the possible actions occurring on the arc (n_1, n_2).

1. If there is a wait action then $c(n_1, n_2) = T^W \geq \delta > 0$
2. If there is a move action through the arc (w_1, w_2) and a rotation occurs on w_1 with angle φ then

$$c(n_1, n_2) = t^R\left(r, \varphi, \Omega_r\right) + t^D\left(r, d^E\left(w_1, w_2\right), \overrightarrow{A}_r, \overline{V}_r, \overleftarrow{A}_r\right)$$
$$\geq \frac{\varphi}{\Omega_r} + \frac{d^E\left(w_1, w_2\right)}{\overline{V}_r} \geq \frac{d^E\left(w_1, w_2\right)}{\overline{V}_r} \geq T^R \geq \delta > 0$$

3. If there is a move action through the arc (w_1, w_2) and no rotation occurs on w_1, moreover, w_0 is the last stopping node where a rotation occurs with angle φ then
$$c(n_1, n_2) = t^R\left(r, \varphi, \Omega_r\right) + t^D\left(r, d^E\left(w_0, w_2\right), \overrightarrow{A}_r, \overline{V}_r, \overleftarrow{A}_r\right)$$
$$- t^D\left(r, d^E\left(w_0, w_1\right), \overrightarrow{A}_r, \overline{V}_r, \overleftarrow{A}_r\right)$$
$$\geq \frac{\varphi}{\Omega_r} + \frac{d^E\left(w_1, w_2\right)}{\overline{V}_r} \geq \frac{d^E\left(w_1, w_2\right)}{\overline{V}_r} \geq T^R \geq \delta > 0$$

Thus, $c(n_1, n_2) \geq \delta > 0$. □

In the proof, δ is selected as the smallest cost of the changed state for each robot, which is either the waiting time T^W or the time T^R for the robot with highest speed to go through the shortest arc. According to Eq. (3.1), $\max_{i \in \mathcal{R} \cup \mathcal{B}} \max_{i' \in \mathcal{R} \cup \mathcal{B} \setminus \{i\}} L_i^R + L_{i'}^R$ is the lower bound of the length of $(w1, w2)$. In the first case, there is no action, which is shorter than δ, since δ is $min(T^W, T^R)$. In the second case, the time for going through an arc is longer than T^R, and likewise for the third case. Note that in the third case, the time for the path from $w1$ to $w2$ is calculated through the path from $w0$ to $w2$ minus the time for the path from w_0 to w_1. Therefore, δ is the finite and positive lower bound for all actions in search space $\mathcal{S}$.

-

Chapter 4

Decision rules for Robotic Mobile Fulfillment Systems

Marius Merschformann[1] Tim Lamballais[2] René de Koster[2] Leena Suhl[1]

[1] *University of Paderborn, Paderborn, Germany*
[2] *Rotterdam School of Management, Erasmus University Rotterdam, Rotterdam, Netherlands*
marius.merschformann@upb.de, lamballaistessensohn@rsm.nl, r.koster@rsm.nl, leena.suhl@upb.de

submitted to European Journal of Operational Research

Abstract

The Robotic Mobile Fulfillment Systems (RMFS) is a new type of robotized, parts-to-picker material handling system, designed especially for e-commerce warehouses. Robots bring movable shelves, called pods, to workstations where inventory is put on or removed from the pods. This paper simulates both the pick and replenishment process and studies the order assignment, pod selection and pod storage assignment problems by evaluating multiple decision rules per problem. The discrete event simulation uses realistic robot movements and keeps track of every unit of inventory on every pod. We analyze seven performance measures, e.g. throughput capacity and order due time, and find that the unit throughput is strongly correlated with the other performance measures. We vary the number of robots, the number of pick stations, the number of SKUs (stock keeping units), the order size and whether

returns need processing or not. The decision rules for pick order assignment have a strong impact on the unit throughput rate. This is not the case for replenishment order assignment, pod selection and pod storage. Furthermore, for warehouses with a large number of SKUs, more robots are needed for a high unit throughput rate, even if the number of pods and the dimensions of the storage area remain the same. Lastly, processing return orders only affects the unit throughput rate for warehouse with a large number of SKUs and large pick orders.

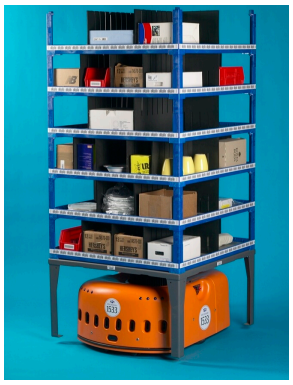
4.1 Introduction

The rise of e-commerce has created the need for new warehousing systems. Traditional, manual picker-to-parts systems work best when orders are large, i.e. consist of many SKUs so that consolidation has to be organized well. However, e-commerce orders are typically small and e-commerce warehouses are often large as they need to contain large assortments of products, which results in long walking distances for the pickers. In contrast to manual picker-to-part systems, automated parts-to-picker systems eliminate the time pickers spend traveling. Thus, they can achieve higher pick rates.
The Robotic Mobile Fulfillment System (RMFS) is an automated parts-to-picker system. Robots transport movable shelves, called "pods", that contain the inventory, back and forth between the storage area and the workstations. As RMFSs eliminate picker walking time, high pick rates can be expected. The systems are mainly used by Amazon, which bought the company that invented the RMFS, Kiva Systems, and has since deployed more than 100,000 robots in its warehouses (see [12]). Recently, competitors such as Swisslog, Interlink, GreyOrange, Mobile Industrial Robots and Scallog have been rolling out their versions of an RMFS.
The RMFS is described in more detail in [5] and [16]. They mention that numerous operational decision problems are yet to be examined in depth, for example the assignment of customer orders to workstations or of pods to storage locations. Each of these decision problems comes with a trade-off. An order may be assigned to a workstation if it is nearing its due time, but assigning another order that has lines in common with other orders assigned to that workstation may result in more picks per pod and hence a reduction in the number of pod trips. Furthermore, assigning a pod to a storage location that is close to the workstation reduces travel time, but keeping the inventory sorted by assigning pods to favorable storage location if they are likely to be needed in the near future may reduce travel times more.
These trade-offs are linked to the number of robots in the system. As an example, with more robots, more trips can be done and hence the order due times can become a more important criterion than the number of picks per pod when selecting a pod to be transported to a workstation. The trade-offs are also linked to the resources and

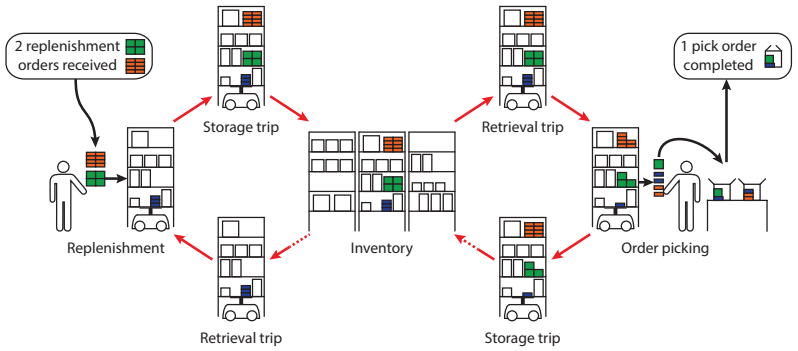
conditions in the warehouse. For example, the more SKUs a warehouse contains, the more difficult it becomes to assign orders to pick stations in such a way that multiple products can be picked from a single pod.
As these examples indicate, a need exists for finding methods to address the decision problems in an RMFS, for research on the performance of RMFSs across performance measures, and for examining performance while varying aspects like the number of robots. This paper addresses this need. We study the pick order assignment, replenishment order assignment, pick pod selection, replenishment pod selection, and pod storage assignment decision problems and propose several decision rules for each. To see which trade-offs in performance may exist, we use different performance measures. Furthermore, we vary three aspects of the RMFS, namely whether or not return orders need to be processed, the size of the orders, and the number of SKUs in the warehouse. This study focuses on both the pick process and the replenishment process, because a more efficient replenishment process frees up robots for pick tasks. Lastly, the number of pick stations and the number of robots per pick station is varied. Varying these numbers shows how many pick stations and robots are needed to provide pickers with a near continuous supply of pods.
Section 4.2 describes the RMFS in more detail, Section 4.3 points out related work, Section 4.4 the decision problems, Section 4.5 the decision rules, and Section 4.6 describes the realistic simulation built for evaluating the decision rules, while Section 4.7 explains the evaluation framework, Section 4.8 shows the results of the analysis, and Section 4.9 provides conclusions and directions for future research.

4.2 The Robotic Mobile Fulfillment System

An RMFS consists of shelves on which products are stored (called pods), robots that can move underneath and also carry them (see Figure 4.1a), and work stations. After handling a pod at a station it can be returned to a different storage location than where it was retrieved from, hence, inventory can be sorted continuously throughout the day.
Figure 4.1b shows the storage and retrieval processes, where the robots transport pods between the workstations and the storage area. Starting at the replenishment station, in the example, two replenishment orders with 4 and 8 units of two SKUs (green & orange) are stored on a pod that was retrieved from the inventory by a robot. Some units of the blue SKU, also relevant to the process example, have already been available on the pod at this point. After the pod was handled at the station it is stored in inventory again. Next, if the pod is selected for picking at a pick station, it is brought to that station. The operator at the station then picks the units matching the open order lines at the station from the pod and puts them into the bins for the respective pick orders. As soon as a pick order is completed



(a) Robot carrying a pod (see [5])



(b) The internal storage / retrieval process in RMFSs (red: robot & pod movement

Figure 4.1 The essential elements of an RMFS

it leaves the pick station and is handled by further warehouse systems. If zoning is in place at the warehouse, the pick order may only be a part of a larger customer order and must be consolidated further with the other partial pick orders in a following sortation process. If the customer order is already completely fulfilled at the pick station, it may be packed into a carton and prepared for shipping immediately with no further handling. The latter may only be possible in e-commerce operations where lines per order are small.

Each pair of storage and retrieval trip is one robot cycle in an RMFS. During one cycle the robot does not set-down or leave the rack until it is returned to a storage location. Note that, the pod may be brought to further replenishment or pick stations between the retrieval and the storage trip, if further replenishment or immediate picking can be done with it. For the sake of clarity we limited the visits per cycle to one station in the example above. While the operation of the robot is cyclic the flow of the inventory units through the system starts at a replenishment station (by storing a replenishment order) and exits at a pick station (by fulfilling a pick order). However, in contrast to other systems there is quite some overhead inventory movement, because all contained units, not only needed ones, are moved when a pod is brought to a station. The same happens during replenishment operations, if non-empty pods are moved to a replenishment station.

Robots navigate their paths through the warehouse using a waypoint system, which is laid out as a grid. A path is a sequence of connected waypoints and all robots have to be guided concurrently along their paths while avoiding collisions and deadlocks. Robots that are not carrying a pod can move underneath stationary pods and hence take other paths than robots that do carry pods, because the latter cannot use occupied storage locations. The system layout is depicted in Figure 4.2 and

consists of a storage area where the pods are stored, pick and replenishment stations grouped around the storage area, maneuvering areas between the storage area and the workstations, and per workstation a buffer area. A robot carries a pod from the storage area, via the maneuvering area, to the buffer area of the destination workstation. Pods are picked or replenished one at a time per station. Workers at the replenishment stations replenish the pods with new inventory. In contrast, workers at the pick stations pick product units to fulfill orders. A picker picks for multiple unfinished/incomplete pick orders at the same time. For both operations the robots need to stop with a pod at a waypoint representing the access point of the respective station. In the buffer area next to each workstation, robots carrying pods can wait for their turn. In the middle of the layout a number of waypoints is used as possible storage locations where pods can be put when they are not used. Every storage location is directly reachable from an aisle and access to a storage location cannot be blocked by stored pods. Travel in the aisles is single-directional to avoid gridlock and reduce congestion.
The system has the ability to adapt to changing demand conditions. E.g., if order arrival rates of some SKUs drop, pods containing those SKUs can be relocated further away from the pick stations. This relocation frees up storage locations near the pick stations for pods containing SKUs with high order arrival rates. Pods can be relocated when returning from a workstation, hence the inventory can be continually sorted in response to changing demand.

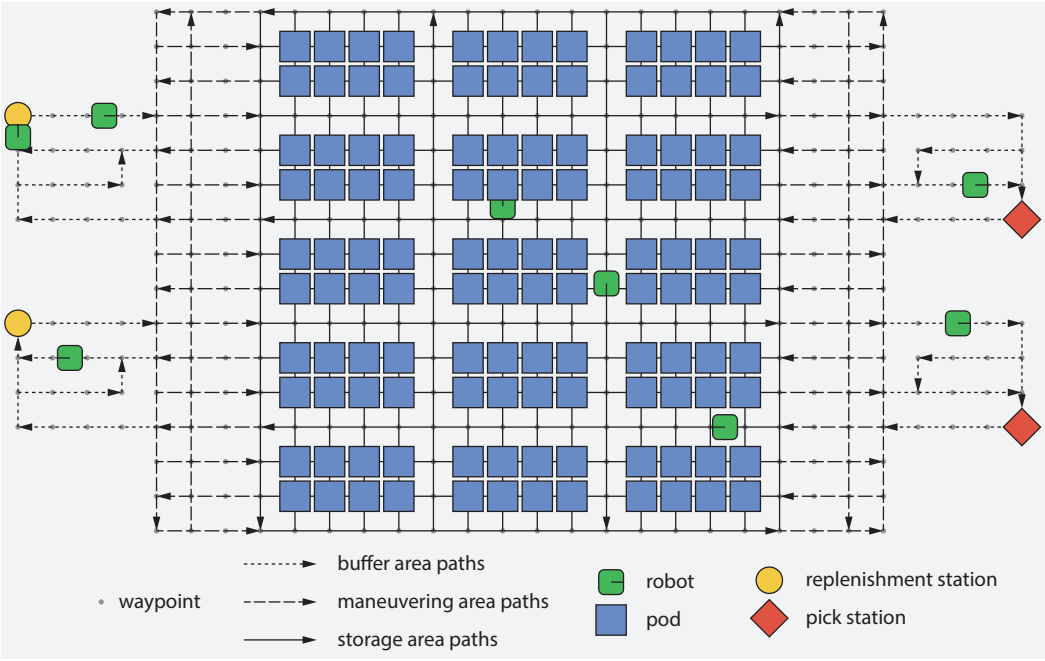


Figure 4.2 A top view of an RMFS layout

4.3 Related Work

To this date no detailed discrete event simulation based research on control topics has been done for RMFS. Moreover, most research on RMFSs to date uses queueing networks to study design questions on the strategic level. This work aims to close the gap by delivering insights about RMFS using a very detailed simulation framework that integrates most dynamic effects an operator faces. Next, we first outline the queuing network based research and close this section with simulation based work.

[13] create queueing networks similar to earlier queueing networks used for autonomous vehicle storage and retrieval systems (AVS/RS) and automated storage and retrieval systems (AS/RS) (see [6] and [15]). Their queueing networks capture both pick and replenishment operations but cannot model robot movement realistically. They estimate the order throughput time for single-line orders. [7] create a different queueing network for both single- and multi-line orders, with and without zoning in the storage area, that captures only the pick operations, but that does include realistic robot movement. Their model can accurately estimate the expected order cycle time, workstation utilization and robot utilization. [7] determine how the storage area dimensions and the workstation placement around the storage area affect the maximum order throughput, by evaluating a large number of possible designs. [8] develop a queueing network that addresses problems on a tactical level. They show the effect of the number of pods per SKU and of the replenishment level of a pod on order throughput, and they show what the optimal ratio of the number of pick stations to the number of replenishment stations is. They find that it is better to replenish pods before they are entirely empty, even with multiple pods per SKU. [17] use semi-open queueing networks to analyze the policy for assigning robots to pick stations. The authors find that the random policy is significantly outperformed by the proposed handling-speeds-based assignment rule when facing varying service rates of the pickers. [18] build a semi-open queueing network for evaluating the effects of battery management in RMFS. The strategies of battery swapping, automated plug-in charging and inductive charging at the pick station are compared. The authors come to the conclusion that battery swapping is generally more expensive than plug-in charging while inductive charging outperforms both in throughput and costs, if robot prices and retrieval times are low.

[5] and [16] mention several decision problems on the operational level that they encountered in practice. One of the few studies that address decision problems on the operational level is by [2]. They provide methods for optimally batching the pick orders and sequencing both the pick orders and the pods transported to the stations. They show that an optimized pick order processing requires only half the number of robots that a pick order process based on simple decision rules would need. [3] devise a simulation study to compare the RMFS to a miniload order picking system.

The authors find that for the assumed scenarios and parameters a miniload system with four aisles and one conveyor loop yields approximately the same performance as an RMFS with 50 AGVs. In the experiments one fixed control logic is used. [14] utilize a simulation based approach in order to optimize the warehouse layout of a manual order picking system for an industrial partner. The authors devise an integrated approach taking on certain design decisions as well as selecting control policies. The simulation is thereby used "as a solution tool and an evaluation system" (see [14]). [4] use a simulation based approach for evaluating the performance of policy sets for manual order picking systems. The authors make use of DEA as a tool for obtaining a comparable performance indicator among the policy sets. [1] use a discrete event simulation approach similar to this work for assessing storage policies for Automated Grid-based Storage systems. The authors find that even simple strategies improve the system efficiency, which encourages research on more complex strategies. [9] develop a Markow decision process (MDP) model for addressing the resource reallocation problem, i.e., the problem of deciding how many workers and robots to allocate to the pick process and replenishment process continually throughout time. The assumptions related to replenishment differ strongly across the papers mentioned above, and the number of approaches to replenishment in practical applications is diverse as well.

4.4 Decision Problems

This section introduces the decision problems considered in this paper and places them within the context of other decision problems in an RMFS. Requests to the system occur via pick orders or replenishment orders. Upon receipt, pallets are broken up into smaller parts consisting of multiple units of one SKU. A replenishment order is a request to place one such part, i.e. a number of units of one specific product, on a pod.
We structure the decisions at the operational level in four steps: (1) Order Assignment (OA), the assignment of pick or replenishment orders to workstations, (2) Task Creation (TC), the creation of tasks for the robots, (3) Task Allocation (TA), the allocation of tasks to robots, and (4) Path Planning (PP), the creation of paths along which the robots will move. There are two kinds of Order Assignment decisions: the assignment of pick orders to pick stations, called the Pick Order Assignment (POA) problem, and the assignment of replenishment orders to replenishment stations, called the Replenishment Order Assignment (ROA) problem. In the second step, a task is defined as transporting a specific pod to a specific workstation and back to a specific storage location. Therefore, for each workstation, the Task Creation decision problem includes the two subproblems of (2.1) deciding which pod to select for transportation, the Pod Selection (PS) decision problem, and (2.2) decid-

Figure 4.3 Hierarchical overview of the decision problems and their relations

ing at which storage location to return the pod, the Pod Storage Assignment (PSA) decision problem. The Pod Selection (PS) decision problem differs for the pick and replenishment process, because for the pick process the due times of the pick orders is important in selecting a pod. Pod selection in the pick process is called Pick Pod Selection (PPS) and pod selection in the replenishment process is called Replenishment Pod Selection (RPS). Task Creation uses the pick order and replenishment order assignments to select suitable pods and subsequently converts the requests for the selected pods into tasks for pod transportation between the workstations and the storage area. Task Allocation creates a trip by building a sequence of tasks for the robots to execute. These sequenced tasks implicitly define trips and serve as input for the Path Planning algorithms, where a path is generated for a robot to follow.

Figure 4.3 shows an overview of the decision problems at the strategic, tactical and operational level in an RMFS, with the problems addressed in this paper in bold.

As can be seen in Figure 4.3, this paper focuses on decision problems at the operational level. We use the term "decision rule" to refer to a fairly simple method to solve a decision problem. The aim of this paper is to evaluate several decision rules per decision problem. Some decision rules may closely resemble common best practices, whereas others may be more specific to RMFS. The Task Allocation decision problem is intertwined with the Path Planning decision problem, which has been addressed by [10]. Therefore we do not consider the Task Allocation and Path Planning decision problems. We do address Pick Order Assignment (POA), Replenishment Order Assignment (ROA), Pick Pod Selection (PPS), Replenishment Pod Selection (RPS), and Pod Storage Assignment (PSA). For Pick Order Assignment, we assume there is a constant backlog, and the pick stations are always filled to full capacity with pick orders. Whenever a pick order is fulfilled and leaves its pick station, a pick order has to be selected from the backlog and assigned to the pick station. For replenishment orders, we assume that the sequence of replenishment orders inbound to the system cannot be altered anymore. This assumption resembles the situation in conventional conveyor-based material handling components that do not allow sequence modification but only load routing. Moreover, we aim to avoid taking decision problems outside of the system's boundaries into account, e.g., different dispatching rules of preceding systems. The replenishment stations have a finite capacity. If a replenishment order arrives and multiple replenishment stations have capacity left, the ROA decision rule determines to which replenishment station the replenishment order is assigned. If no place is available, replenishment orders are put in a replenishment order backlog. When a replenishment order is fulfilled at one of the replenishment stations, a new replenishment order is chosen from the replenishment order backlog according to the FCFS rule. Table 4.1 summarizes the decision problems addressed in this paper.
At this point we also introduce the concept of "pile-on" (sometimes also called "hit-rate"). Pile-on as a concept refers to the average number of units that are picked from a pod every time a pod is presented to a picker at a pick station. Pile-on as a metric measures the number of units (across all SKUs) picked from a pod when presented to a picker at a pick station, averaged across every visit of a pod to a pick station during the entire time horizon. In other words, pile-on is measured in "units picked per pod visit to a pick station". The higher the pile-on is, the fewer pods need to be transported between the pick stations and the storage area, which may reduce the number of robots needed.

Table 4.1 Decision Problems

Abb.	Name	Description	Trigger
POA	Pick Order Assignment	Choosing a pick order from the backlog	When another pick order is fulfilled and leaves the pick station, creating room for the next pick order to be assigned
ROA	Replenishment Order Assignment	Selecting the replenishment station for the next replenishment order	When a replenishment order arrives at the system and one or more replenishment stations have capacity left
PPS	Pick Pod Selection	Selecting a pod to transport to a pick station	When a robot working for a pick station needs a new task
RPS	Replenishment Pod Selection	Select a pod for the next replenishment order	Depends on the ROA decision rule
PSA	Pod Storage Assignment	Choosing a storage location for a pod	When a pod leaves a workstation

4.5 Decision Rules

To solve the operational problems, we define several decision rules per decision problem that are evaluated in a realistic simulation. Several Path Planning algorithms for the RMFS are compared in [10], therefore this decision problem will not be addressed in this paper. Thus, we selected WHCA$^*_v$, one of the best performing algorithms from the paper, as the path planning engine for this work. Additionally, we fix the Task Allocation algorithm to a simple method that first assigns two-thirds of the robots to pick operations and the rest to replenishment operations. Then, it aims to equally distribute the robots across the respective stations. This means a robot will only do tasks related to the station it is assigned to. This section will therefore only describe decision rules for the Pick Order Assignment, Replenishment Order Assignment, Pick Pod Selection, Replenishment Pod Selection and Pod Storage Assignment decision problems.
While replenishment and pick operations are similar in the sense that high throughput should be achieved with few resources, the main asymmetry between both is that for the former the goal is to fill the inventory as quickly as possible and for the latter to empty it as quickly as possible. This means that for replenishment operations we aim to replenish pods fast to have them available for pick operations early while preparing pod content such that it allows for a high pile-on during pick operations. For pick operations we aim to achieve a high pile-on and keep trips short to fulfill as many orders as possible while also considering due times of the pick orders. Furthermore, we do not allow the sequence of replenishment orders to be modified. In contrast, for pick orders we allow to arbitrarily choose one order from the backlog. Lastly, pick orders have due times. All of this leads to different

strategies we focus on per decision problem, instead of fully symmetric rules between pick and replenishment decision problems.
For a more precise description of some of the rules we introduce the notation shown in Table 4.2.

Table 4.2 Overview of the symbols used in the rule descriptions

Symbol	Explanation
$\mathcal{P}$	Set of all pods
$\mathcal{P}_s^I$	Set of pods heading to station s
$\mathcal{I}$	Set of all SKUs
$\mathcal{O}^B$	Set of pick orders in backlog
$\mathcal{O}_s^S$	Set of pick orders assigned to station s
$C(p,i)$	Number of units of SKU i contained in pod p
$L(o,i)$	Required units necessary to fulfill line i of order o
$D(o,i)$	Remaining units necessary to fulfill line i of order o
t_o^D	Due time of order o
t_o^S	Time of assignment to the station of order o
t	Time of decision rule invocation (moment of decision)

4.5.1 Pick Order Assignment Rules

A pick station has to be chosen for every pick order submitted to the system and the pick order itself has to be chosen from the order backlog. In this work, we consider a pick order backlog of constant size, i.e., as soon as an order is removed from the backlog a new one is generated to replace it. This and the immediate replacement of orders completed at a station lead to only one option available to assign any pick order to: the slot of the just completed order. Hence, the choice of station is not a degree of freedom in this work. The rare occasions of multiple orders to be completed at the same time are handled by assigning the orders to the pick stations randomly. Hence, we only investigate rules for selecting the next pick order from the backlog to fill the only open slot at a station. We devise six rules to solve this problem: “Random”, “FCFS”, “Due-Time”, “Fast-Lane”, “Common-Lines” and “Pod-Match”:

Random The Random rule randomly selects a next pick order from the backlog and is used as a benchmark.

FCFS The FCFS rule assigns the pick order that was first received. The rationale behind this is to keep pick order throughput times short.

Due-Time The Due-Time rule selects the pick order with the earliest due time from the backlog and assigns it to a station. This is a greedy approach aiming to finish the pick orders before their deadline.

Fast-Lane The Fast-Lane rule randomly selects a pick order from the backlog like the Random rule, but keeps one slot at each pick station open for immediately completeable pick orders. I.e., only pick orders (o), for whom all lines and all units of inventory are available on the next pod (p_n) will be assigned to this station's "fast-lane" order slot (see Equation (4.1)). Thus, orders assigned to the "fast-lane" slot are processed shortly after assignment. The next pod of the station is either a not completely processed pod the picker is currently working on or the next pod in the station's queue, if no such pod is available. In cases where no pod reached the station's queue yet, we consider the pod with the shortest remaining path to estimate the next pod. When facing multiple options we use a random tie-breaker. Note that this rule can be combined with any other proposed POA rule. The reason we combine it with random selection is to better assess the impact of the idea itself.

$$\forall i \in \mathcal{I} : L(o,i) \leq C(p_n,i) \tag{4.1}$$

Common-Lines The Common-Lines rule compares the station's (s) currently assigned pick orders with all orders from the backlog and selects the one with most lines in common for assignment (see Equation (4.2)). The rationale behind this is to increase pile-on by exploiting synergies among the pick orders. When facing multiple options we use a random tie-breaker.

$$\operatorname{argmax} o \in \mathcal{O}^B \sum_{o' \in \mathcal{O}_s^S} \sum_{i \in \mathcal{I}} y_{oo'i} \quad \text{with } y_{oo'i} = \begin{cases} 1 & L(o,i) > 0 \wedge L(o',i) > 0 \\ 0 & otherwise \end{cases} \tag{4.2}$$

Pod-Match The Pod-Match rule selects the pick order from the backlog that matches best the pods heading to the station (s) at the moment of assignment best. I.e., the more units of the pick order are already available in the pods the better the match (see Equation (4.3)). When facing multiple options we use a random tie-breaker.

$$\operatorname{argmax} o \in \mathcal{O}^B \sum_{p \in \mathcal{P}_s^I} \sum_{i \in \mathcal{I}} \left(\min\left(C(p,i), D(o,i)\right)\right) \tag{4.3}$$

4.5.2 Replenishment Order Assignment Rules

As a result of the assumptions that replenishment orders arrive in a fixed sequence, we investigate only two different approaches for assigning replenishment orders to the stations, i.e., immediate Random assignment and batching of customer orders that go on the same pod. Hence, we construct two rules for replenishment assignment: "Random" and "Pod-Batch":

Random The Random rule randomly selects a next station with sufficient remaining capacity to allocate incoming replenishment orders to. If no such station is available, the order will wait until one becomes available again.

Pod-Batch The Pod-Batch rule tries to use a pod already selected to go to a replenishment station for assigning the next replenishment order. In other words, the Pod-Batch rule first waits for the Replenishment Pod Selection (Section 4.5.4) rule to decide which orders are assigned to which pod, and then uses the same replenishment station for the orders of one pod. If the replenishment orders do not fit one station, they wait until a station with sufficient capacity becomes available. During this time all consecutive orders are also blocked, because the sequence cannot be altered.

4.5.3 Pick Pod Selection Rules

Every time a robot working for a pick station s requests a next task, a pod suitable for picking at pick station s must be selected. We require for all rules that at least one unit can be picked from the pod. This means that no pod is brought to a station completely in vain and additionally it implies a pile-on of at least 1. The six PPS rules used in this paper are the "Random", "Nearest", "Pile-on", "Demand", "Lateness", and "Age" rules:

Random The Random rule randomly selects a pod that offers at least one useful unit for picking.

Nearest The Nearest rule selects the pod which has the least estimated path time towards the station according to the path planning algorithm and that offers at least one useful unit for picking.

Pile-on The Pile-on rule selects the pod that offers most units necessary to fulfill the orders at the station (see Equation (4.4)). Ties are broken by favoring pods with which more orders can be completed. If ties still persist, they are broken randomly.

$$\operatorname{argmax} p \in \mathcal{P} \sum_{i \in \mathcal{I}} \sum_{o \in \mathcal{O}_s^S} \left(\min\left(C(p,i), D(o,i)\right)\right) \tag{4.4}$$

Demand The Demand rule selects the pod whose content is most demanded considering the current pick order backlog situation, i.e. the pod with most units demanded in the backlog is chosen (see Equation (4.5)). Ties are broken randomly.

$$\operatorname{argmax} p \in \mathcal{P} \sum_{i \in \mathcal{I}} \sum_{o \in \mathcal{O}^B} \min\left(C(p,i), D(o,i)\right) \tag{4.5}$$

Lateness The Lateness rule aims to finish late pick orders by selecting a pod that offers units needed to fulfill open order lines with most lateness at the station, i.e., for one order the time the order is late is summed as fractions of the open picks (see Equation (4.6)). If no order is late, the resulting ties are broken by using the same metric but replacing $\max\left(t - t_o^D, 0\right)$ with t_o^D, thus, selecting pods for orders whose due times are most imminent.

$$\operatorname{argmax} p \in \mathcal{P} \sum_{i \in \mathcal{I}} \sum_{o \in \mathcal{O}_s^S} \left(\frac{\min\left(C(p,i), D(o,i)\right)}{\sum_{i' \in \mathcal{I}} D(o,i')} \max\left(t - t_o^D, 0\right) \right) \tag{4.6}$$

Age The Age rule aims to finish the oldest pick orders of a station by selecting a pod that offers units needed to fulfill the oldest open order lines, i.e. for one order the time the order spent assigned to the station is summed as fractions of the open picks (see Equation (4.7))

$$\operatorname{argmax} p \in \mathcal{P} \sum_{i \in \mathcal{I}} \sum_{o \in \mathcal{O}_s^S} \left(\frac{\min\left(C(p,i), D(o,i)\right)}{\sum_{i' \in \mathcal{I}} D(o,i')} \left(t - t_o^S\right) \right) \tag{4.7}$$

4.5.4 Replenishment Pod Selection Rules

For every replenishment order, a suitable pod with sufficient remaining storage capacity needs to be chosen. The decision is taken right before the replenishment order is assigned to a replenishment station. Depending on the selected ROA and RPS rules both are either invoked simultaneously or, if there is a dependency between the two, one after the other. An example for the latter case is the combination of the PodBatch ROA rule with the Emptiest RPS rule, because the PodBatch rule relies on an already selected pod for the replenishment order. Since Replenishment Pod Selection determines the composition of the pods, it offers many possibilities to create pods with different features, e.g. high frequency pods that combine frequently ordered products, or family-based pods combining products that are often ordered together. If all replenishment orders assigned to the same pod are assigned to the same replenishment station, only one trip is necessary to place all replenishment orders on the pod, which reduces the number of robot movements.

The five RPS rules used in this paper are the “Random”, “Emptiest”, “Nearest”, “Least-Demand” and “Class” rules:

Random The Random rule selects a random pod with sufficient remaining capacity.

Emptiest The Emptiest rule assigns replenishment orders to the emptiest pod and reuses the same pod for subsequent replenishment orders until it is full or used at a station.

Nearest The Nearest rule assigns an incoming replenishment order to the nearest pod with sufficient remaining capacity.

Least-Demand With the Least-Demand rule an incoming replenishment order is assigned to the pod currently offering the least demanded inventory, i.e. the pod with the least units offered when compared to the aggregated demand by assigned and backlogged pick orders is selected. Thus, this pod is not useful for pick-operations at the time of selection and by this it is not disadvantageous to block it for replenishment operations.

Class The Class rule assigns incoming replenishment orders to a pod of the same class as the replenishment order, i.e. fast moving SKUs to pods with other fast moving SKUs. The classes are built by a background mechanism for which the cumulative relative amount of pods per class are given. For this work we use “0.1, 0.3, 1.0”, i.e., three classes where the first class holds 10 % of the pods for the highest frequency SKUs, the second class holds 20 % and the last class holds the remaining ones, which are the ones with the lowest frequency SKUs. To assign a replenishment order of a certain class, the emptiest pod is selected from the pods of that particular class. Similar to the Emptiest rule, a selected pod is used for the subsequent incoming replenishment orders of the same class until no more replenishment orders fit the pod or until the respective pod completes its visit to a replenishment station.

4.5.5 Pod Storage Assignment Rules

For each pod an unoccupied storage location has to be selected, every time after visiting a pick or replenishment station. PSA is an important aspect of the RMFS, because being able to change the storage location of pods after every visit to a workstation is what makes continuous automatic sorting possible. For PSA, five decision rules are examined, namely the “Random”, “Fixed”, “Nearest”, “Station-Based” and “Class” rules.

Random The Random rule chooses a random free storage location.

Fixed The Fixed rule maintains the initially assigned storage location for all pods.

Nearest The Nearest rule stores pods at the nearest unoccupied storage location in terms of shortest estimated path time. This path time is determined using an A* algorithm that takes the time needed for turning the robot (with or without pod) into account.

Station-Based The Station-based rule is a variant on the Nearest rule, i.e. instead of bringing the pod to a storage location that is nearest to the robot's position the storage location with shortest path time to a pick station is selected. The greatest difference with the Nearest rule is in the storage locations chosen for pods returning from a visit to a replenishment station.

Class The Class rule brings pods back to storage locations of the same class, where classes are constructed in a similar fashion as in the RPS decision problem, but based on the shortest path time to a pick station. Within a class, a storage location for a pod is selected analogously to the Nearest rule.

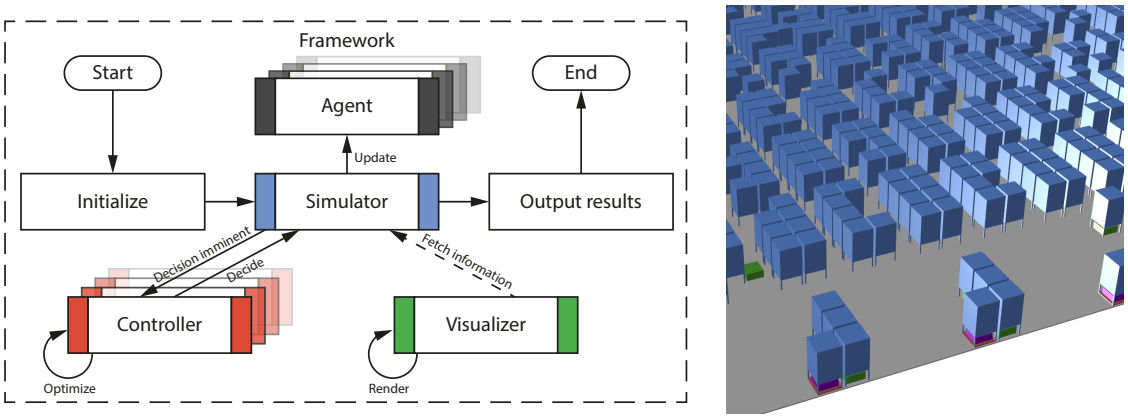
Table 4.3 provides an overview of the decision rules per decision problem and shows how the decision rules are labeled across decision problems. Note that choosing a rule for one decision problem may jeopardize strategies chosen for others. For example, a random Pick Order Assignment may have a negative impact on a Class-based approach for assigning replenishment orders to storage locations, because it does not respect the units currently positioned near the pick station while assigning orders to it. Hence, a selection respecting mutual influences has to be done to provide an efficient compilation of rules that is able to adequately overcome the planning problems in such a system.

Table 4.3 Overview of the Decision Rules per Decision Problem

Decision Problem	Decision Rules
POA	Random, FCFS, Due-Time, Fast-Lane, Common-Lines, Pod-Match
ROA	Random, Pod-Batch
PPS	Random, Nearest, Pile-on, Demand, Lateness, Age
RPS	Random, Emptiest, Nearest, Least-Demand, Class
PSA	Random, Fixed, Nearest, Station-Based, Class

4.6 Simulation Framework

In this work we use the simulation framework "RAWSim-O". A more detailed description of it can be found in [11] while the source code is available at https:



(a) Overview of the simulation process. **(b)** Visualization screenshot

Figure 4.4 RAWSim-O simulation framework

//github.com/merschformann/RAWSim-O. "RAWSim-O" is an agent-based and event-driven simulation focusing at a detailed view of an RMFS. The basic simulation process is managed by the core simulator instance (see Figure 4.4a), which is responsible for obtaining the next event and updating the agents. Agents can either represent real entities like robots and stations or virtual entities like process managers, e.g. for emulating order processes. Every decision that has to be made is passed to the corresponding controller. The controller can either immediately decide or can buffer multiple requests in order to optimize and release the decision later on. However, in this work we only consider ad-hoc decision rules with the former approach. To allow visual feedback, the ongoing simulation can optionally be rendered in 2D and 3D. The implementation was done in C#.

The level of detail of the simulation is especially high for the simulated movement behavior of the robots. We consider the robot's momentum by emulating acceleration and deceleration behavior, collision avoidance and turning speed (see Table 4.5). The emulation employs a continuous time-horizon. The times for activities other than robot movement, e.g. lifting or storing a pod, or picking one unit at a pick station, are constant (see Table 4.5). The waypoints allow the emulated robot behavior to match real robot behaviour. Robots that do not carry a pod can traverse underneath stored pods by using the waypoints at which the pods are stored. Furthermore, in the buffers of the workstations, robots can take short-cuts if the buffer is (partially) empty.

Information about the system's state is tracked in a high level of detail, because some decision rules differ with regard to the information they require. For example, all pods and all units on all pods are tracked exactly. Incoming information is divided into a static and a dynamic category. Static information includes everything

describing a system instance and is completely given at the start. Static information therefore includes the number and composition of pick stations and replenishment stations, the pods, the robots, and the waypoint system used for robot navigation. All of the decision rules proposed in this work differ in their computational complexity and therefore also in the computational time they require to reach a decision. They are, however, simple enough to be considered as ad-hoc decisions even for large system sizes.
In contrast to static information, the dynamic information is not completely known beforehand, but becomes available over time. This is the case for incoming pick orders and replenishment orders submitted to the system over time by external processes. While each replenishment order consists of a number of physical units of one SKU, each pick order consists of a set of order lines, each for one SKU, with corresponding units necessary to fulfill the line. We assume for both pick and replenishment orders, that there is a constant order backlog. A constant order backlog means that when an order from the backlog is assigned to a workstation, it is immediately replaced by a newly generated order. By keeping the order backlogs constant, we aim to analyze the system's behavior under constant pressure. However, it also leads to the phenomenon that the system's storage space utilization (utilized space divided by total space available) in the storage area is affected by the performance of the decision rules controlling it, because no further virtual manager steers the process. E.g., if a combination of rules is leading to quick replenishment, the storage space utilization will increase. In contrast, it will decrease, if the rules are replenishing slowly. Situations in which the storage space utilization is nearing 100%, and only few storage places for new replenishment orders are available, lead to an inefficient replenishment process. To avoid such situations, we pause replenishment order generation, if storage space utilization exceeds 85 % and it is continued after it drops below 65 % again. Analogously, we pause the pick order generation, if storage space utilization drops below 10 % and resume after it exceeds 60 % again. The latter is done to avoid draining the inventory completely. Since in both cases either the replenishment stations or the pick stations will become inactive due to no further orders to process, the robots will be reassigned to the remaining active stations. This redistribution of robots across the active stations is done at any time a station becomes active or inactive, i.e. at the beginning and end of order generation pauses.

If a new replenishment order is received, first the rules for ROA and RPS are responsible for choosing a replenishment station and a pod (see Figure 4.5). The time the decision is taken depends on the active rules. The execution of the assignment can earliest be done as soon as there is sufficient capacity on a pod and a station available. Technically, it results in an insertion request (shown as red cylinders), i.e. a request that requires a robot to bring the pod to the workstation. Multiple

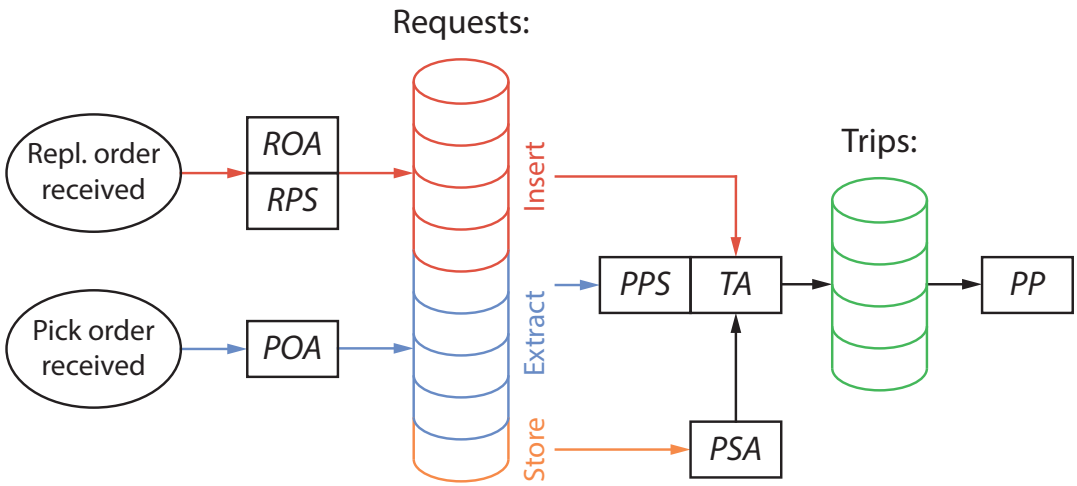


Figure 4.5 Order of decisions to be done induced by receiving pick and replenishment orders

of these requests are then combined to an insertion task and assigned to a robot by a TA rule. Similarly, after the POA rule selects a pick order from the backlog and the assignment is committed to a pick station, an extraction request (shown as blue cylinders) is generated, i.e. a request that requires bringing a suitable pod to the chosen station. Up to this point, the physical units of SKUs for fulfilling the pick order are not yet chosen. Instead, the decision is postponed and taken right before combining different requests to extraction tasks by PPS and assigning them to robots by TA. This allows the implemented rules to exploit more information when choosing a pod for picking. Hence, in this work we consider PPS as a decision closely interlinked with TA. Furthermore, the system generates store requests (shown as orange cylinders) each time a pod has to be transported to a storage location. The PSA rule only decides the storage location for a pod that is not needed anymore and has to be returned to the storage area. If all requests are already being handled by other robots, the robot will be assigned an idle task, thus, the robot dwells at a dwelling point until needed. Dwelling points can be used to reduce congestion effects if there are only a few active stations compared to the number of robots, e.g. robots waiting at a storage location block others that try to pass by. For this, the robot will park at a free storage location to avoid causing conflicts with other robots. The dwell point policy uses locations in the middle of the storage area to avoid blocking prominent storage locations next to the stations. Another type of task would be charging, which is necessary when robots run low on battery, however, in this work we assume the battery capacity to be infinite, so this type of task is ignored. All of the tasks result in trips (shown as green cylinders), which are planned by a path planning algorithm and executed by the robots. The only exception is when a pod

can be used for another task at the same station. The trips are planned by a PP algorithm and the resulting paths are executed by the robots. Figure 4.5 shows an abstract overview of these dependencies. The exact times at which the decisions are taken depend on the respective rules, e.g. the Pod-Batch ROA rule assigns a batch of replenishment orders to the first pick station offering sufficient space while the Random ROA rule immediately assigns single replenishment orders to the first station with sufficient capacity available. However, all of the rules have in common that they make assignments greedily while adhering to certain capacity constraints (station capacity, pod capacity, etc.).

4.7 Evaluation Framework

This section describes the evaluation framework used to carry out the research in this paper. Two central concepts to the evaluation framework are the Rule Configuration (RC) and the Warehouse Scenario (WS). The RC specifies for each decision problem, which decision rule is used. The WS specifies the warehouse layout, number of robots, number of workstations, number of SKUs, whether or not return orders are part of the operations of the warehouse, and pick order size. During one simulation run the RC and WS do not change, so they can be seen as an input to a simulation run.

The evaluation framework consists of two phases, one varying the RCs, the other varying the WSs. Phase 1 evaluates all 1620 possible RCs on one WS. For phase 1, we compare eight performance measures: (1) unit throughput rate, (2) pick order throughput rate, (3) order turnover time, (4) distance traveled per robot, (5) order offset, (6) fraction of orders that are late, (7) pile-on (8) the pick station idle time. Unit throughput rate is the number of picked units of all SKUs per hour. Pick order throughput rate is the number of pick orders fulfilled per hour. Order turnover time is the average time between submitting a pick order to the backlog and fulfilling it. Order offset is the average time between the due time and the completion time of the pick orders. Thus, a value smaller than zero shows how much in advance pick orders are completed. The rationale behind this is that follow-up processes at the distribution center are not deterministic, hence, pick orders completed earlier may improve the overall service level. The pick station idle time is measured as an average across all pick stations in the system.

Phase 1 selects the RCs with the highest unit throughput rate. However, among these selected best RCs, the variety in the decision rules may be low. For a particular decision problem, all of the selected RCs may use the same decision rule. To ensure more diversity in the RCs in phase 2, we define 6 so-called “benchmark RCs”, see Table 4.4. The benchmark RCs were chosen such, that all decision rules across all decision problems appear in at least one of the benchmark RCs. Each benchmark

RC has been given a name that reflects a characteristic that the decision rules have most in common. Phase 2 evaluates the selected RCs from phase 1 and the

Table 4.4 Benchmark RCs

Benchmark RC	POA	ROA	PPS	RPS	PSA
Demand	Due-Time	Pod-Batch	Demand	Least-Demand	Fixed
Speed	Fast-Lane	Pod-Batch	Lateness	Emptiest	Nearest
Nearest	FCFS	Random	Nearest	Nearest	Nearest
Class	Common-Lines	Pod-Batch	Age	Class	Class
Greedy	Pod-Match	Pod-Batch	Pile-on	Emptiest	Station-Based
Random	Random	Random	Random	Random	Random

benchmark RCs, while varying the warehouse scenarios. Since we are specifically interested in efficiency of RCs we neglect layout decisions for this work. Thus, we choose one specific layout, using the style described in Section 4.2. The concrete layout instance comprises 1149 pods and 1352 storage locations (85% filled) and is shown in Figure 4.6. When varying the number of pick stations during phase 2 we add workstations in the order given in Figure 4.6.

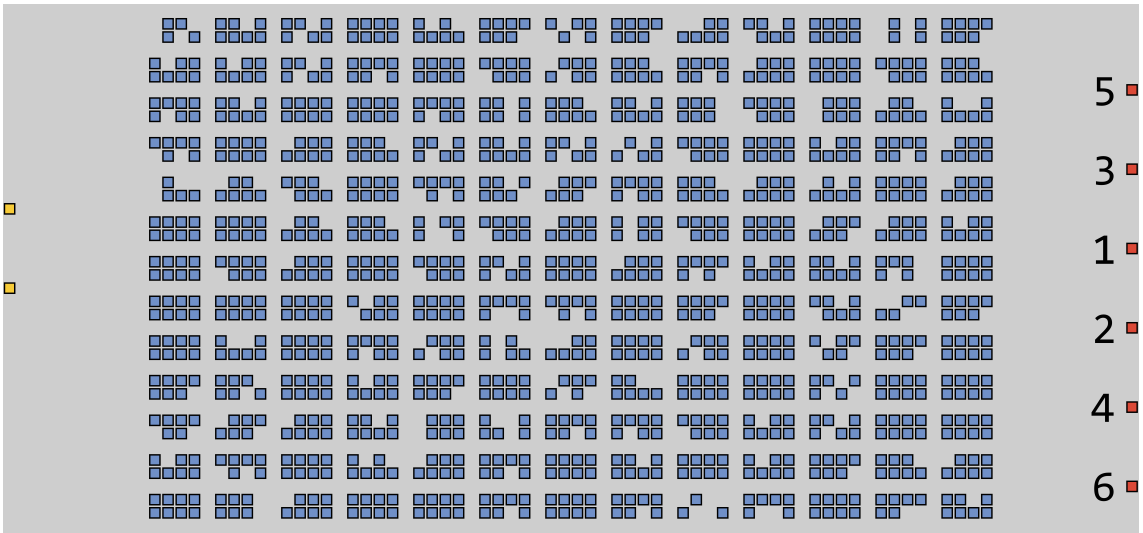


Figure 4.6 Top view of the layout, including pick station indices, with the storage area in the middle, replenishment stations to the left, and pick stations to the right

4.7.1 Parameters

In the following we describe the used parameters in more detail. The parameters shared for both phases are outlined in Table 4.5. We set a continuous simulation horizon of 48 hours in order to decrease the impact of side effects like recurring replenishment overflows, which cause replenishment pauses described previously. Within a duration of 48 hours we observe sufficient repetitions of such patterns to achieve a reasonable mitigation of these side effects.
Furthermore, for each RC and WS combination in phase 1 and in phase 2 we conduct 10 runs to lessen the effect of randomness. To keep the system under continuous pressure, like described above, we keep a constant pick and replenishment order backlog of 200 orders each. At simulation start inventory is generated until 70 % overall storage utilization to avoid cold starting the system. This is done using the same process used for generating replenishment orders during simulation and using assignment rules suiting the respective RPS rule in place. The storage capacity of a pod is set to 500 slots while the storage consumption of one SKU unit is drawn from a uniform distribution between 2 and 8 slots, thus, a full pod contains 100 units in average. The popularity of the SKUs is determined by drawing a value from an exponential distribution with parameter $\lambda = \frac{1}{2}$ for each SKU to emulate a typical ABC curve in e-commerce. This popularity is the relative frequency parameter between all SKUs, thus, the frequency (if divided by the sum of all frequencies) is the probability of choosing a particular SKU when generating an order line for both replenishment and pick orders. One replenishment order restocks between 4 and 12 units of one SKU following a uniform distribution. To emulate due times we distinguish between priority and normal orders that have to be completed in 30 minutes respectively 120 minutes. This reflects the need for preferring important orders.
The movement behavior of the robots is emulated by using a maximum velocity of $1.5\frac{m}{s}$ with acceleration and deceleration rates of $0.5\frac{m}{s^2}$. We set the rotational speed to $\frac{4}{5}\pi\frac{rad}{s}$, i.e., $2.5s$ for a full turn. Turning takes the same amount of time regardless of whether a robot is carrying a pod. The time for lifting and setting down a pod is set to $3s$. This should reflect the capabilities of mobile robots used in similar industry applications reasonably close. For the actual pick operation of one unit at a pick station we assume a constant time of $8s$. The complete time for handling one unit including additional operations, like putting the product unit in the correct pick order tote, is set to $15s$. This distinction is considered to allow for an early release of the robot, such that no unnecessary robot waiting times are caused. This is not distinguished for replenishment operations, since we assume that a robot can only leave after fully completing the put operation to the pod. The time of a put operation of one replenishment order is set to $20s$.

Table 4.5 Parameters shared across all simulations

Parameter	**Value**
Simulation	
Simulated duration of warehouse operations	48 hours
Number of simulation repetitions	10 repetitions
Size of pick order backlog	200 pick orders
Size of repl. order backlog	200 repl. orders
Layout	1149 pods, 1352 storage locations in 2×4 blocks, 12 aisles and 12 cross-aisles
Orders	
Number of units per repl. order	uniform distribution between 4 and 12 units
Amount of priority orders in pick orders	20 %
Priority pick order due time	backlog submission time + 30 min.
Normal pick order due time	backlog submission time + 120 min.
Threshold when pick order generation starts	60% of inventory capacity of the storage area
Threshold when pick order generation stops	10% of inventory capacity of the storage area
Threshold when repl. order generation starts	65% of inventory capacity of the storage area
Threshold when repl. order generation stops	85% of inventory capacity of the storage area
Inventory	
Initial inventory in the storage area	70% of the inventory capacity of the storage area
Space on a pod	500 slots
SKU frequency / popularity	Exponential distribution, $\lambda = \frac{1}{2}$
SKU size	uniform distribution between 2 and 8 slots
Robot movement	
Robot acceleration/deceleration	$0.5 \frac{m}{s^2}$
Robot maximum velocity	$1.5 \frac{m}{s}$
Time needed for a full turn of a robot	$2.5s$
Time needed for lifting and storing a pod	$3s$
Time needed for picking a unit	$8s$
Time needed for handling a unit at pick station	$15s$
Time needed for putting a repl. order on a pod	$20s$
Stations	
Repl. station capacity	two times pod capacity
Pick station capacity	8 pick orders

The parameters in Table 4.5 are shared across all conducted experiments, while the parameters in Table 4.6 depend on phase and scenario. For the first phase we assess all possible RCs for one fixed warehouse scenario. Note that the RPS rule Nearest and the ROA rule Pod-Batch rely on each others assignments for taking their decisions (since they are using them as inputs), which leads to no decision at all. Hence, the combination of these rules is forbidden. For the fixed warehouse scenario we set the number of robots to 4 per pick station, i.e. 8 robots in the system at whole. Furthermore, we set the number of pick stations to 2, the number of SKUs to 1000 and exclude the processing of return orders. The order setting

is set to Mixed. This means the number of lines per pick order and the number of units per order line are generated following truncated normal distributions with parameters shown in Table 4.6. This is done to resemble e-commerce pick order characteristics of generally small orders with occasional larger ones in between. Equation (4.8) shows that phase 1 has 1620 RCs, and since phase 1 has 1 WS and 10 runs are conducted per RC and WS combination, this results in 16200 simulation runs for phase 1. Phase 2 has 10 RCs (see Table 4.6) and Equation (4.8) shows that it has 360 WSs, which together with 10 runs per RC and WS combination leads to 36000 simulation runs for phase 2.

$$\begin{aligned}\#RC \text{ in phase } 1 = \big| \{ROA\} \times \{POA\} \times \{RPS\} \times \{PPS\} \times \{PSA\} \setminus \\ \{(roa, poa, rps, pps, psa) \mid (roa = \text{Pod-Batch}) \wedge (rps = \text{Nearest})\} \big| = 1620\end{aligned} \tag{4.8}$$

$$\begin{aligned}\#WS \text{ in phase } 2 = \big| \{\text{number pick station}\} \times \{\text{robots per pick station}\} \\ \times \{\text{number of SKUs}\} \times \{\text{return orders}\} \times \{\text{pick order size}\} \big| = \\ 6 \times 5 \times 2 \times 2 \times 3 = 360\end{aligned} \tag{4.9}$$

For phase 2 we limit the RCs to the 6 benchmark RCs and the 4 best ones from phase 1, i.e., the 4 RCs with highest throughput rate. Moreover, we vary the number of pick stations from 1 through 6 and the number of robots per pick station from 2 through 6. This leads to a range from 2 robots in the system to 36 robots across all WSs. In addition to WSs with 1000 SKU, we also assess WSs with 10000 SKUs stored in the system. For the order size we define two additional settings of small and large orders. For the Small pick order size, only single line / single unit pick orders are generated. For the Large pick order size, the distributions from the Mixed order setting are used but the min parameter for both is set to 2. Lastly, in WSs where we emulate the processing of return orders, 30 % of the generated replenishment orders are single unit. The total number of RC and WS combinations for the phase 2 is therefore 3600, which leads to 36000 simulation runs.

Table 4.6 Varied parameters for phase 1 and 2

Parameter	Phase 1 values	Phase 2 values
Rule configurations (RCs)	1620 RCs	6 Benchmark RCs + 4 best RCs from phase 1
# pick stations	2	1, 2, 3, 4, 5, 6
Robots per pick station	4	2, 3, 4, 5, 6
# SKUs	1000	1000, 10000
Return orders	0 %	0 %, 30 %
Pick order size	*Mixed* - line & unit dist.: $\mu = 1, \sigma = 1, \min = 1, \max = 4$ $\mu = 1, \sigma = 0.3, \min = 1, \max = 3$	*Small* - line & unit dist.: $\min = 1, \max = 1$ $\min = 1, \max = 1$ *Mixed* - line & unit dist.: $\mu = 1, \sigma = 1, \min = 1, \max = 4$ $\mu = 1, \sigma = 0.3, \min = 1, \max = 3$ *Large* - line & unit dist.: $\mu = 1, \sigma = 1, \min = 2, \max = 4$ $\mu = 1, \sigma = 0.3, \min = 2, \max = 3$
# RC	1620	10
# WS	1	360
# RC×WS	1620	3600
# simulation runs	16200	36000

4.8 Computational Results

This section shows the results from phase 1 and phase 2 of the evaluation framework. Throughout this section, the unit throughput rate is presented as a percentage of the upper bound on the unit throughput rate. The unit throughput rate is presented in this way to facilitate interpretation and comparison of results across experiments. Moreover, the RMFS is supposed to have high pick rates as it eliminates the need of walking for the workers, while the robots are supposed to supply the pickers with a constant stream of pods to pick from. Presenting the unit throughput rate as a percentage shows clearly to what extent these aims are achieved. The upper bound is discussed in more detail in Appendix 4.10. The length of the confidence intervals is always less than 1% of the mean, based on 10 runs per RC and WS combination, and therefore does not add much information.

4.8.1 Phase 1

The first phase aims to investigate throughput performance and the impact per decision problem of decision rules on throughput. Furthermore, we assess the behavior of the different output measures depending on decision rule selection. For this, Table 4.7 shows how across these simulations the eight previously introduced performance measures correlate with each other. At first, we can observe that as the unit throughput rate score improves, the other performance measures improve

Table 4.7 Correlations between the different performance measures for first phase

	Unit throughput	Order throughput	Order turnover time	Distance traveled	Order offset	Late orders	Pile-on	Station idle time	μ	σ
Unit throughput	-	-	-	-	-	-	-	-	0.556	0.189
Order throughput	1.000	-	-	-	-	-	-	-	241.963	82.234
Order turnover time	-0.950	-0.950	-	-	-	-	-	-	3549.625	1220.445
Distance traveled	-0.952	-0.952	0.880	-	-	-	-	-	122598.768	19433.860
Order offset	-0.950	-0.950	1.000	0.880	-	-	-	-	-2565.458	1224.985
Late orders	-0.590	-0.591	0.685	0.549	0.684	-	-	-	0.187	0.115
Pile-on	0.899	0.899	-0.802	-0.796	-0.802	-0.448	-	-	2.438	1.450
Station idle time	-1.000	-1.000	0.950	0.952	0.950	0.591	-0.899	-	0.450	0.186

as well. As the unit throughput rate score increases, pick order throughput rate and pile-on increase as well, whereas the order turnover time, the distance that robots travel, the order offset, the fraction of orders that miss their due time, and the station idle time decreases. Although it is not clear what the exact causal relationships are, the correlations suggest that pile-on and the distance traveled by the robots are the main drivers behind these improvements. With higher pile-on, more units are picked per pod, so order lines are fulfilled more quickly and fewer trips are needed to fulfill the pick orders. This also causes longer processing times for each pod at the pick station, which in turn increases the time for the next robot to queue and become ready at the station. In other words: a more continuous input of inventory at the pick station is achieved. Additionally, fewer trips for the pick process free up robots to do more replenishment tasks. With less distance traveled by the robots we expect pods to be presented at the pick stations more continuously. Similar to the pile-on this effect enables more continuous picking, which in turn increases the overall unit throughput rate. Both measures, pile-on and the traveled distance, are intermediate measures affected by the choice of strategy for the different decision problems, i.e., a better score in both decrease the idle time at the stations, which in turn increase the throughput. An increased throughput, in the constant pick order backlog setting of this work, also decreases the turnover time of pick orders and the due time offset. Only the number of orders being late is not strongly correlated with the two main throughput drivers. The two main throughput drivers can also be observed when looking at a scatter plot of all simulation runs of the first phase (see Figure 4.7). Here we can see the best results in unit throughput rate score are achieved with a high pile-on and less distance traveled per robot. The group of simulation runs with least distance traveled per bot and a pile-on around 4 are RCs involving the Nearest PPS rule, while the simulation runs with highest

pile-on (greater 5) at the top of the plot are RCs involving the Demand PPS rule. In both groups we find runs with the highest unit throughput rate score, hence, a higher throughput is not only achieved by a high pile-on. In particular within the top ten RCs in terms of unit throughput rate score the pile-on ranges between 3.84 and 6.36, while the distance traveled per bot ranges between 68.04 km to 80.36 km. Hence, pile-on and the traveled distance enable higher throughput, but may also compensate for each other. This is particularly interesting, because both come at operational costs. For traveled distance this is energy consumption and robot wear, while for pile-on it may be costs arising from potentially more complex replenishment processes. Furthermore, within both groups better results are obtained with RCs also involving the Pod-Match POA rule, which causes an additional boost in pile-on.
In Figure 4.7 we also observe a 'cutoff' of simulation runs in the upper right and bottom left areas. This can be explained by the longer handling time at the station resulting from a higher pile-on. I.e., the longer a robot needs to wait at a station for the picking to finish the less it can travel in the meantime. Thus, rules increasing pile-on may help reducing the necessary travel distance, and by this also robot wear and energy consumption.

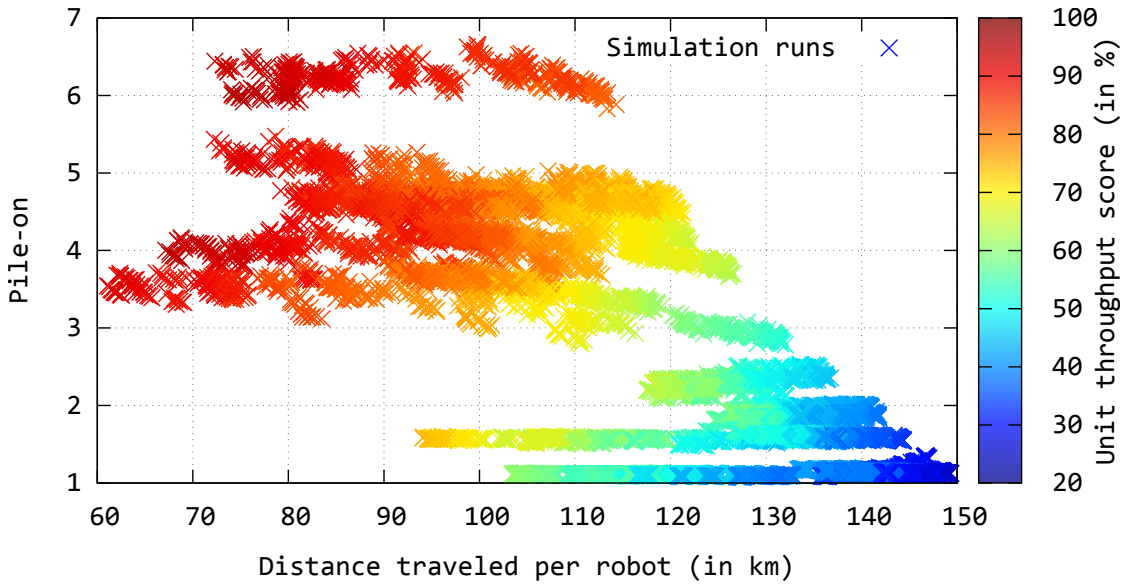


Figure 4.7 Scatter plot for pile-on vs. traveled distance per robot colored by the achieved throughput rate score for all simulation runs of the first phase

The pick order throughput rate is neglected completely in the remainder of this work, because it almost completely aligns with the unit throughput rate score. The reason for this is the constant backlog of 200 pick orders over 48 hours: with a pick order

throughput rate of 241.963 completed orders per hour in average, omitting certain pick orders is almost impossible. Hence, we cannot observe a potential temporary throughput gain by preferring smaller or larger orders. In order to investigate the trade-off between picking many units and completing more pick orders an experiment with a fixed set of backlogged pick orders over a fixed period of time should be devised. For this, the possibly tedious processing of leftover pick orders, which are presumably harder to pick quickly, needs to be investigated. We leave this work for future research.

Table 4.8 Average unit throughput rates as percentages of the upper bound for all rules, together with the **best** / **worst** performance multiplier per decision problem

							Mult. $\left(\frac{\text{best}}{\text{worst}}\right)$
POA	Common-Lines	Due-Time	Fast-Lane	FCFS	Pod-Match	Random	
	50.93%	41.93%	76.13%	41.81%	**81.18%**	**41.71%**	1.946
ROA	Random	Pod-Batch					
	53.71%	**57.99%**					1.080
PPS	Age	Demand	Lateness	Nearest	Pile-on	Random	
	61.50%	52.70%	**48.63%**	**62.16%**	59.82%	48.88%	1.278
RPS	Class	Nearest	Emptiest	Least-Demand	Random		
	56.16%	58.42%	**59.63%**	57.71%	**47.56%**		1.254
PSA	Class	Fixed	Nearest	Random	Station-Based		
	55.91%	54.08%	**58.79%**	**53.60%**	55.70%		1.097

Table 4.8 shows for each decision problem the unit throughput rate score for each of the decision rules, averaged across all simulations in phase 1. We calculate the multiplier by dividing the highest unit throughput rate by the lowest. As the multiplier in unit throughput rates is rather large for the POA decision problem, system integrators and RMFS suppliers may benefit from carefully selecting a POA decision rule and from investigating better decision rules for this decision problem. The multiplier for the Replenishment Order Assignment is near 1, indicating that using a different decision rule does not offer much performance improvements. However, we note that we keep the sequence of incoming replenishment orders fixed at all times in this work, which limits improvement potential. Nevertheless, we expect limited degrees-of-freedom in replenishment operations to be more realistic, because the sequence will typically be a result of preceding operations or systems. Moreover, the limited number of replenishment stations diminishes the impact of ROA decision rules even more. Furthermore, the impact of the Pod Storage Assignment selection rule seems to be fairly low. This may be a reason of the quite small layout. We expect the impact of PSA decision rules to increase with the size of the instance layout, because the effect on the traveled distance would grow by a large amount.
In the following we analyze the achieved throughput performance per decision rule. For this, Figure 4.8 shows the box-plots of unit throughput rate scores for each deci-

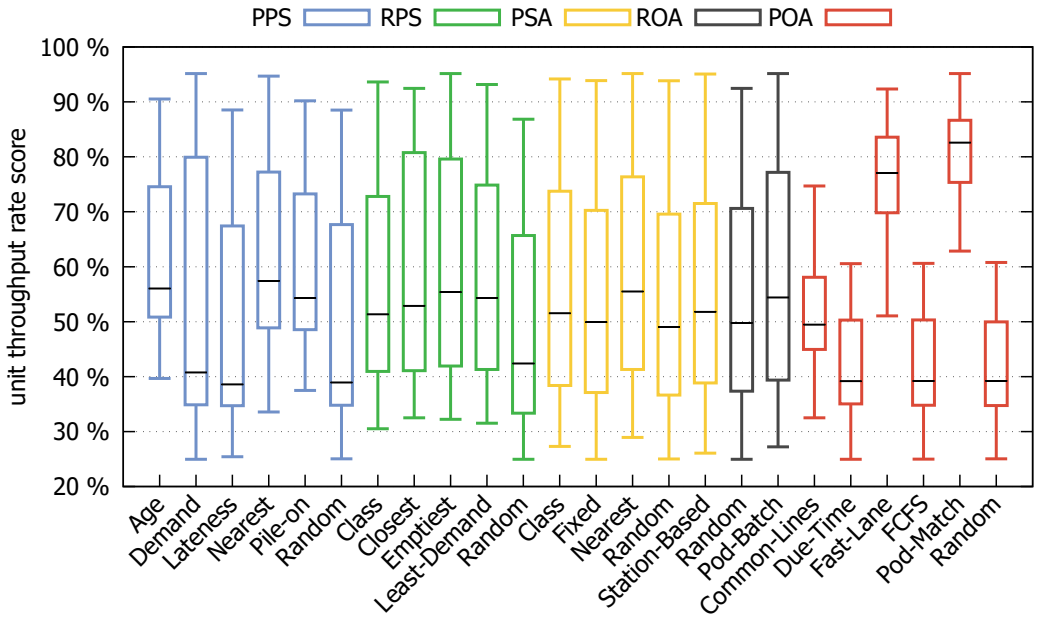


Figure 4.8 Unit throughput rate performance of all runs involving the given rule

sion rule colored per decision problem. The boundaries of the boxes are determined by the upper and lower quartile while the line in the middle indicates the median value. The whiskers extend from the boxes to the minimum and maximum values. The first observation is that throughput performance of the RMFS is most sensitive to the choice of POA decision rule among the defined decision rules. This aligns with the previously observed correlations, because the choice of POA immediately affects the pile-on, which is identified as a major performance driver. The best performing POA strategies are FastLane and PodMatch, which both look at the incoming pods at a pick station when assigning new pick orders from the backlog. This suggests that a strategy aligning pick orders with the content of incoming pods seems most promising for throughput efficiency. This backs up the findings of [2]. Although the Common-Lines rule exploits a similar greedy strategy, it achieves substantially less throughput. Hence, only matching pick orders to each other but not to the content of the pods squanders throughput capabilities of the system. All other POA decision rules achieve similar throughput performance, since they do not consider order characteristics that would affect pile-on or traveled distance.
When looking at the PPS rule box-plots the average best throughput performance with least variance is achieved by the Age, Nearest and Pile-on rules. All of them focus either on maximizing the pile-on or minimizing the traveled distance. Although the Age rule does only indirectly maximize pile-on, it achieves a higher average pile-on of 2.92 among all RCs containing it than the actual Pile-on rule, which achieves

an average pile-on of 2.79. The Demand rule has the highest spread across PPS rules with a very low median, but also provides some top performing RCs (see Table 4.9). This suggests that the throughput performance of the rule has a higher dependency on the selection of other rules.
Although the variation among the ROA decision rules is small, we observe a slightly better throughput performance by the Pod-Batch rule. This is a reason of the smaller number of trips necessary when batching replenishment orders.
Many of the top performing RCs contain the Emptiest or Nearest RPS decision rule. The main reason for the good throughput performance again seems to rely on fewer and shorter trips. The Emptiest rule decreases the number of trips, because more replenishment orders are stored in pods at once until it is full. E.g., only 31.03 % pods need to be brought to replenishment stations in average when compared to the Random rule. The Nearest rule benefits from a similar effect since the same (nearest) pod is used for further replenishment orders even while it is already approaching. Furthermore, Nearest decreases the distance per replenishment trip, because nearer pods are used. The Random rule performs worst for RPS. The main reason for this is that too many trips are caused by randomly selecting pods while only storing few replenishment orders per trip.
Among the PSA decision rules we observe the best throughput performance for the Nearest strategy. This is again mainly caused by the shorter trips for the robots. When comparing the Nearest and the Station-based rule we see the benefit from shorter trips for replenishment operations increasing throughput of pick operations. However, this depends on the queue length at stations and the distribution of robots between replenishment and picking. I.e., if longer queue times are expected at replenishment stations than in our devised scenarios, moving pods nearer to the pick stations when returning them to the inventory may improve overall throughput performance. The Fixed and Random decision rules differ little in their performance. The main reason for this is that the storage location per pod in the Fixed rule is randomly selected. Thus, leading to a very similar behavior.
Due to the large sample sizes, the results of ANOVA and Tukey's range tests rejected the hypotheses that the means were equals at the 0.05 significance level within groups and pair-wise, with five exceptions. The null hypothesis of equal means was not rejected at the 0.05 significance level for POA rules FCFS and Due-Time, for Random and Due-Time, and for Random and FCFS. Furthermore, for PPS rules Random and Lateness the hypothesis of equal means could not be rejected, and for PSA rules Station-Based and Class.

4.8.2 Phase 2

From the 1620 RCs in phase 1, the four with the highest unit throughput rate (see Table 4.9) together with the benchmark RCs form the set of ten RCs used in phase

Table 4.9 RCs with best throughput score selected from first phase (performance is unit throughput rate score)

RC rank	POA	ROA	PPS	RPS	PSA	performance
1	Pod-Match	Pod-Batch	Demand	Emptiest	Nearest	94.81 %
2	Pod-Match	Pod-Batch	Demand	Emptiest	Station-Based	94.63 %
3	Pod-Match	Pod-Batch	Nearest	Emptiest	Nearest	94.43 %
4	Pod-Match	Pod-Batch	Demand	Emptiest	Class	94.00 %

2. The main purpose of phase 2 is to examine how well the RCs perform under different circumstances. In the following we analyze the results obtained for the 12 warehouse scenarios and 30 resource settings described before (see Section 4.7.1).

Table 4.10 Best unit throughput rate score for all scenarios, robots per pick station and numbers of pick stations. Scenario abbreviations: [SKU count: 1000 (1K), 10000 (10K)]-[Order size: Small (S), Medium (M), Large (L)]-[Return orders: yes (R), no (N)]

Stations	1					2					3					4					5					6				
Robots	2	3	4	5	6	2	3	4	5	6	2	3	4	5	6	2	3	4	5	6	2	3	4	5	6	2	3	4	5	6
1K-S-N	44	82	91	97	97	59	89	94	97	98	64	90	95	97	98	60	87	93	97	98	57	87	93	97	98	59	87	94	97	98
1K-S-R	46	82	92	97	97	59	89	94	97	98	63	90	95	97	98	61	88	93	97	98	56	87	93	97	98	55	86	93	97	98
1K-M-N	45	83	92	97	98	60	90	95	98	98	64	90	95	98	98	60	88	93	98	98	57	88	94	98	98	59	88	94	98	98
1K-M-R	45	82	92	97	98	59	89	95	98	98	63	91	96	98	98	62	89	93	98	98	56	88	94	98	98	56	87	94	98	98
1K-L-N	54	83	93	99	99	66	90	97	99	99	68	91	96	99	99	67	88	94	99	99	63	86	94	98	99	63	87	94	99	99
1K-L-R	50	80	92	99	99	64	88	96	99	99	65	90	97	99	99	67	88	94	99	99	62	86	93	98	99	62	84	94	98	99
10K-S-N	21	39	55	68	78	27	44	59	72	81	28	46	61	73	80	29	47	59	69	77	30	47	57	68	77	31	45	58	68	76
10K-S-R	20	40	56	70	80	27	45	61	74	82	29	47	62	74	82	30	48	61	70	78	31	48	58	67	76	31	46	57	66	74
10K-M-N	23	41	58	71	81	28	46	61	75	84	29	48	64	76	83	30	49	61	71	80	31	48	60	71	80	32	47	60	71	79
10K-M-R	21	41	59	73	83	28	48	63	76	85	30	49	65	77	84	31	50	63	72	81	32	49	60	70	79	32	47	59	69	77
10K-L-N	36	63	84	94	98	44	71	89	96	99	46	73	87	94	98	47	69	83	92	97	46	67	84	91	97	45	68	83	91	97
10K-L-R	30	52	76	89	96	37	62	81	92	97	40	64	83	91	96	41	64	76	87	94	42	61	74	84	93	41	59	73	84	92

Table 4.10 shows the results, with the entries being the unit throughput rate as a percentage of the upper bound. In each cell the result of the best performing RC for the respective scenario and station / robot configuration is shown. The unit throughput rate scales well when adding more pick stations, the scaling is (almost) completely independent of the scenario characteristics. However, the necessary number of robots to achieve a given unit throughput rate greatly depends on the scenario characteristics, e.g., for more SKUs more robots are necessary to achieve a high unit throughput rate. The number of SKUs, does have a major impact on performance overall, where the main reason is that pile-on is considerably lower for the 10000

SKU scenarios. A reason for this is the lower likeliness to have a pod with a good combination of SKUs matching the orders of the pick stations available. Thereby, if larger orders have to be processed with the system, this helps mitigating the negative effect of handling lots of SKUs. The main reason for this are the larger number of order lines active at a station when picking larger orders. I.e., more open order lines increase the likeliness of having a well matching pod available for the inventory required at a pick station. Processing return orders has an increased negative effect, if the order size of customer orders is large. However, in general, whether return orders are processed has a lesser effect on throughput performance than the other warehouse scenario variations. The reason behind this may be that even though approximately 19.76 % more time is spent on replenishment operations by the robots when compared to the scenarios without return order processing, replenishment operations are overall quick enough to mitigate the effect. Replenishment operations only consume 20.29 % out of the overall time consumed by the robots in average across all phase 2 simulation runs. Furthermore, we can conclude that with 1000 SKUs, the unit throughput rates are close to their theoretical maximum even with relatively few robots per stations. Table 4.11 shows the unit throughput rate score

Table 4.11 Unit throughput rate scores for the RCs in phase 2 (green ≡ best, red ≡ worst)

Stations	1					2					3					4					5					6				
Robots	2	3	4	5	6	2	3	4	5	6	2	3	4	5	6	2	3	4	5	6	2	3	4	5	6	2	3	4	5	6
RC #1	31	61	77	87	92	43	68	81	89	93	45	70	82	89	93	46	69	79	87	91	44	68	78	86	91	44	67	78	86	90
RC #2	29	59	76	86	91	41	67	80	89	93	44	69	81	88	92	44	68	78	86	91	42	66	77	85	90	42	65	76	85	89
RC #3	33	61	76	86	91	44	69	81	88	93	47	70	82	88	92	47	69	79	86	91	45	68	78	85	90	46	67	78	85	90
RC #4	29	58	75	85	90	40	65	79	87	91	42	67	79	87	91	43	66	77	85	90	41	65	75	83	89	41	64	75	83	88
Demand	14	25	37	48	58	18	29	41	52	62	20	31	42	53	62	20	32	42	52	60	21	32	41	50	58	21	31	40	49	56
Speed	24	40	57	70	80	29	48	63	75	83	32	51	65	76	83	34	52	64	74	81	35	52	63	72	80	35	51	63	72	79
Nearest	19	34	49	62	73	26	41	56	68	78	28	43	57	69	78	29	44	57	67	75	30	44	55	64	72	30	43	54	63	70
Class	26	43	56	67	75	32	47	59	69	77	33	48	61	70	76	34	50	60	68	75	35	49	59	67	73	35	48	58	66	72
Greedy	34	57	72	82	88	43	64	77	85	89	44	66	77	85	89	45	65	75	83	88	44	63	74	81	87	43	62	73	81	86
Random	12	23	23	34	45	12	23	29	37	47	15	23	32	41	51	15	24	34	44	53	15	25	35	45	54	16	26	36	45	53

for the RCs for all combinations of number of robots (n_r) and number of stations (n_s), averaged across WSs and presented as whole percentages. From Table 4.11 we can see that the Ranked RCs from phase 1 perform similarly and better than the benchmark RCs. Among the benchmark RCs, the Greedy benchmark outperforms the others consistently across all settings and is the only one whose unit throughput rate scores approached those of the ranked RCs.

4.9 Conclusion

In this work we studied the throughput performance of decision rules for multiple decision problems occurring in the control of RMFS. By analyzing a total of eight output measures for a total of 1620 RCs, we found strong correlations between these. Most interestingly a high pile-on and a short distance traveled by the robots together almost immediately account for the success of a decision rule applied to RMFS. Hence, we propose using these two output measures as the key tactics when designing decision strategies for RMFS that aim to achieve high throughput. In the investigated high pressure situation further performance measures like the turnover time of pick orders were also highly correlated with the unit throughput rate, which is why we focused on the throughput itself as the main metric for a successful RMFS. Furthermore, we found that varying the decision rule used for solving the Pick Order Assignment affected the unit throughput rate the most. The average unit throughput rate was twice as high for the best decision rule as it was for the worst. This finding indicates that system engineers and warehouse operators should pay most attention to the Pick Order Assignment decision problem. Moreover, the unit throughput rate score ranges from 25.24% for the worst RC assessed in phase 1 to 94.81% for the best scoring RC. Hence, the right combination of decision rules plays a crucial role when controlling an RMFS. We propose that future research may assess how to scale beyond the throughput performance of the merely simple decision rules investigated in this work. However, we observe some cross-dependencies between different strategies for the core decision problems featured in this paper, e.g., the Demand PPS rule is part of the best performing and the worst performing RC. Thus, an integrated and realistic evaluation or validation of new decision methods for RMFS is highly important, since dependencies exist and side-effects should not be neglected. Additionally, we found that the number of different SKUs in the system has a strong impact on the unit throughput rate. This finding is probably due to a decrease in pile-on for a higher number of SKUs. This effect is considerably less for larger orders, presumably because for larger pick orders pile-on tends to be higher. Having to process return orders seems to affect the unit throughput rate more, if the pick orders are large. Moreover, we found that the performance of the "greedy" benchmark consistently came close to the best ranked configurations of decision rules.

This paper has studied solutions to several operational problems, which lead towards promising directions for future research. Each decision rule in this study has looked at an operational problem in isolation, but heuristics that try to integrate multiple operational problems and optimize these problems jointly could achieve substantial increases in order throughput or reductions in resources used. Investigating rules and heuristics that increase pile-on, i.e. the number of picks per handled pod, would also be of great use to practitioners.

While many decision rules and parameters were varied to deliver insightful results we expect even more insight when varying the layout itself. For example, we expect a larger impact of the PSA rule selection when facing huge layout instances. This was not done in this work in order to keep a certain focus and to keep computational resource utilization for the conducted experiments tractable. RMFSs are a new category of automated systems and concepts specific to RMFSs have not received much scholarly attention. An example would be cache zoning / priority zoning, that is the implementation of special zones near the workstations where pods are stored that will be needed in the near future. Another example would be a study of the automatic sorting of the system without explicit zones. Since pods can be relocated to another storage location each time they are transported to and from a workstation, the inventory can be sorted automatically to some degree during operations. It is not clear at which speed automatic sorting takes place or how much performance benefits from it. Automatic sorting is a unique feature of RMFSs, but as with so many other aspects of RMFSs, it remains to be explored.

Acknowledgements

We would like to thank the Paderborn Center for Parallel Computing (PC^2) for the use of their HPC systems for conducting the experiments. Marius Merschformann is funded by the International Graduate School - Dynamic Intelligent Systems, University of Paderborn.

References

[1] Michaela Beckschäfer et al. “Simulating Storage Policies for an Automated Grid-Based Warehouse System”. In: *Computational logistics*. Ed. by Tolga Bektas et al. Lecture Notes in Computer Science. Cham: Springer, 2017, pp. 468–482. ISBN: 9783319684963. URL: https://doi.org/10.1007/978-3-319-68496-3_31.

[2] Nils Boysen, Dirk Briskorn, and Simon Emde. “Parts-to-picker based order processing in a rack-moving mobile robots environment”. In: *European Journal of Operational Research* 262.2 (2017), pp. 550–562.

[3] Yavuz A. Bozer and Francisco J. Aldarondo. “A simulation-based comparison of two goods-to-person order picking systems in an online retail setting”. In: *International Journal of Production Research* 29.1 (2018), pp. 1–21. ISSN: 0020-7543. DOI: 10.1080/00207543.2018.1424364.

[4] C. M. Chen et al. “A Flexible Evaluative Framework for Order Picking Systems”. In: *Production and Operations Management* 19.1 (2010), pp. 70–82.

[5] John Enright and Peter R. Wurman. "Optimization and Coordinated Autonomy in Mobile Fulfillment Systems". In: *Automated Action Planning for Autonomous Mobile Robots*. Ed. by Sanem Sariel-Talay, Stephen F. Smith, and Nilufer Onder. 2011.

[6] S. S. Heragu et al. "Analytical models for analysis of automated warehouse material handling systems". In: *International Journal of Production Research* 49.22 (2011), pp. 6833–6861.

[7] T. Lamballais, D. Roy, and M. B. M. de Koster. "Estimating Performance in a Robotic Mobile Fulfillment System". In: *European Journal of Operations Research* 256 (2017), pp. 976–990.

[8] T. Lamballais, D. Roy, and M. B. M. de Koster. "Inventory Allocation in Robotic Mobile Fulfillment Systems". Working Paper, available at SSRN. 2017.

[9] T. Lamballais et al. "Optimal policies for Resource Reallocation in a Robotic Mobile Fulfillment System". Working paper. 2017.

[10] M. Merschformann, L. Xie, and D. Erdmann. *Path planning for Robotic Mobile Fulfillment Systems*. Working paper, available at arXiv, https://arxiv.org/abs/1706.09347. 2017.

[11] Marius Merschformann, Lin Xie, and Hanyi Li. "RAWSim-O: A Simulation Framework for Robotic Mobile Fulfillment Systems". Working Paper, available at arXiv, https://arxiv.org/abs/1710.04726. 2017.

[12] Nick Wingfield. "As Amazon Pushes Forward With Robots, Workers Find New Roles". In: *The New York Times* (2017). URL: https://nyti.ms/2xUhVgM.

[13] S. Nigam et al. "Analysis of Class-based Storage Strategies for the Mobile Shelf-based Order Pick System". In: *Progress in Material Handling Research: 2014*. 2014.

[14] Kees Jan Roodbergen, Iris F.A. Vis, and G. Don Taylor. "Simultaneous determination of warehouse layout and control policies". In: *International Journal of Production Research* 53.11 (2014), pp. 3306–3326. ISSN: 0020-7543. DOI: 10.1080/00207543.2014.978029.

[15] D. Roy et al. "Performance analysis and design trade-offs in warehouses with autonomous vehicle technology". In: *IIE Transactions* 44 (2012), pp. 1045–1060.

[16] P. R. Wurman, R. D'Andrea, and M. Mountz. "Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses". In: *AI Magazine* 29.1 (2008), pp. 9–19.

[17] Bipan Zou et al. "Assignment rules in robotic mobile fulfilment systems for online retailers". In: *International Journal of Production Research* 55.20 (2017), pp. 6175–6192.

[18] Bipan Zou et al. "Evaluating battery charging and swapping strategies in a robotic mobile fulfillment system". In: *European Journal of Operational Research* 267.2 (2018), pp. 733–753.

Appendix

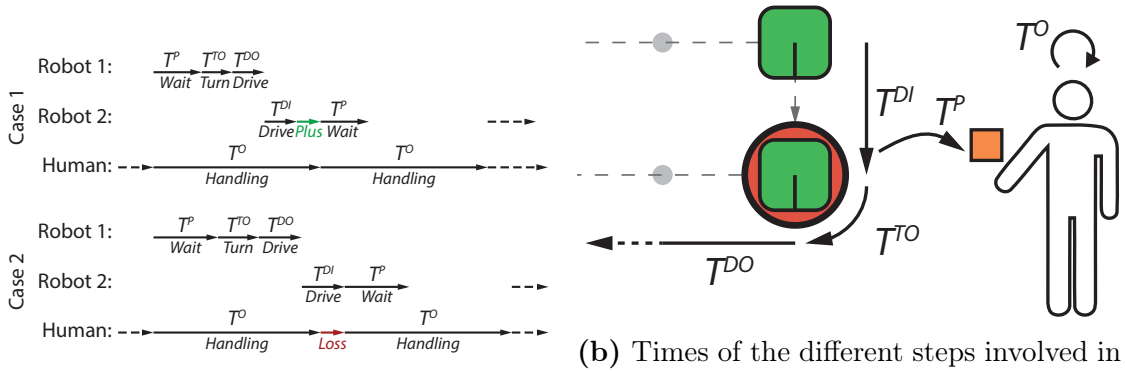
4.10 Upper bound on the unit throughput rate

Table 4.12 Times for determining the upper bound on unit throughput (all times in seconds)

Symbol	Explanation
T^P	Time for picking one unit from the pod (after which the robot can be released, if no further picks are necessary)
T^O	Time for handling one unit at a station (including picking, putting, packing, etc.)
T^{DI}	The time for the robot to move up within the queue to the pick station's waypoint
T^{TO}	The time for the robot to prepare for leaving the station (turning towards exit)
T^{DO}	The time for the robot to clear the station's waypoint (time to cover the minimal distance)

In the following we introduce an upper bound for the number of units picked per hour. This can be done by considering the constant time for picking (T^P) and the constant time for handling (T^O) a unit at a pick station. For an overview of all necessary times see Table 4.12. If a robot is queueing in the buffer of a pick station, it is assumed that it already turns the right pick face of the pod towards the side where the picker will be. Since the robot is waiting in the queue, this happens in the best case without any additional loss of time. During the actual pick process, the robot is occupied for T^P seconds. After this time the robot is allowed to leave the station while the overall handling time for one unit at a station of T^O can be longer.

There are two cases to distinguish for obtaining a performance upper bound. First, if the time for picking a unit from the pod plus the time for moving up the next robot (i.e. the time to turn and drive away from the station and the time for the next robot to approach the station from the queue area) is smaller than the overall



(a) Sample times for both mentioned cases

(b) Times of the different steps involved in picking units

Figure 4.9 Illustrations of the relevant times

handling time of a unit at the station, there is a surplus of time available on the system side and the performance is limited by the handling time of the picker (see case 1 in Figure 4.9a). In the second case we face a longer time for moving up the next robot, hence, in this case we have a loss of time on the system's side and the system is limiting the throughput performance of the picker (see case 2 in Figure 4.9a). For the sake of clarity we define the time for moving up the next robot in queue as $T^{MU} := T^{DI} + T^{TO} + T^{DO}$.

$$UB := \begin{cases} \left|\mathcal{M}^O\right| \dfrac{3600}{T^O} & T^P + T^{MU} \leq T^O \\ \left|\mathcal{M}^O\right| \mathrm{IPO} \dfrac{3600}{T^P + T^{MU} - T^O + \mathrm{IPO} T^O} & \text{else} \end{cases} \tag{4.10}$$

Considering both cases we can determine an upper bound on the unit throughput rate, i.e. the number of units picked per hour (see Equation 4.10). For the first case we only need to consider the unit handling time of the picker to determine the maximum throughput rate of one station and multiply it with the overall count of pick stations $\left|\mathcal{M}^O\right|$. The second case is slightly more complicated, because we also need to consider the pile-on. In the denominator we first calculate the loss of time seen in Figure 4.9a and add it to the average handling time of a pod based on the estimated number of picks from it (IPO). We calculate how many pods are handled in one hour and multiply this by the IPO and the number of stations overall to get the overall upper bound.

We recognize that this upper bound on the unit throughput rate relies on some heavy assumptions for real systems, but still propose it as a rule-of-thumb for practitioners, since it is useful for implementations where the time for moving the next robot, the handling times and the pile-on can be estimated. It is a natural limit of the system's performance that the system cannot exceed, even if the number of robots is more

than sufficient to supply a continuous stream of pods and all rules are performing well. For this work, the upper bound is correct, because all mentioned times can be accurately determined or are constant and are not subject to random influence within the simulation.

Chapter 5

Dynamic Policies for Resource Reallocation in a Robotic Mobile Fulfillment System with Time-Varying Demand

Tim Lamballais[1] Marius Merschformann[2] Debjit Roy[3] Leena Suhl[2]
René de Koster[1]

[1] *Rotterdam School of Management, Erasmus University Rotterdam, Rotterdam, Netherlands*
[2] *University of Paderborn, Paderborn, Germany*
[3] *Indian Institute of Management Ahmedabad, Ahmedabad, India*
lamballaistessensohn@rsm.nl, marius.merschformann@upb.de, debjit@iimahd.ernet.in, leena.suhl@upb.de, r.koster@rsm.nl

Submitted to Transportation Science: Focused Issue on Urban Freight Transportation & Logistics

Abstract

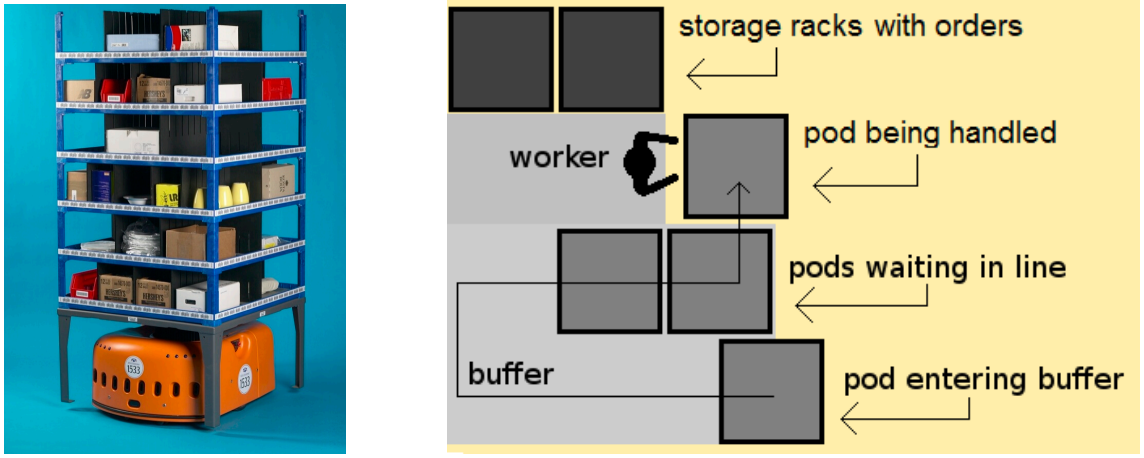
The Robotic Mobile Fulfillment System (RMFS) is an automated part-to-picker material handling system, in which robots carry pods with products to the order pickers. It is particularly fit for e-commerce order fulfillment. An RMFS can quickly and frequently reallocate workers and robots across the picking and replenishment processes to respond to strong demand fluctuations. More resources for the picking

process means less customer waiting time, whereas more resources for the replenishment process means a higher inventory level and product availability. This paper models the RMFS as a queueing network and integrates it within a Markov Decision Process (MDP), that aims to allocate robots across the pick and replenishment processes during both high and low demand periods, based on the workloads in these processes. We do so by extending existing MDP models with one resource type and one process to an MDP model for two resources types and two processes. The policies derived from the MDP are compared with benchmark policies from practice. The results show that the length of the peak demand phase and the height of the peak affects the optimal policy choice. In addition, policies that continually reallocate resources based on the workload outperform benchmark policies from practice. Moreover, if the quantity of robots is limited, continual resource reallocation can reduce costs sharply.

5.1 Introduction

Handling peak demand for order fulfillment is a challenge for e-commerce warehouses. According to [18], e-commerce is challenging for three reasons. First, it requires additional services such as packaging following the picking operations. Second, e-commerce warehouses usually store a very large number of stock keeping units (SKUs), or products, that are picked as individual units rather than as cartons or pallets, making operations complex and labor intensive. Third, e-commerce creates a high pressure on reducing the customer waiting time, with tight delivery schedules, such as same-day delivery, becoming increasingly common. This overlaps with findings of [27] and [28], who identify high customer demand for faster response times and rising expectations of customer service as important current trends and note that the material handling industry is increasingly adopting robotics and automation to create competitive advantage (see [3]).
In e-commerce, demand fluctuations tend to be strong and the influx of new products and the outflux of outdated products are large. The Christmas season is an absolute peak demand season that requires hiring temporary workers to expand the capacity of the warehouse from November to December. Other important peak demand periods are the evening and weekend, when consumers have spare time to shop online, see also [1] and [31].
This paper focuses on handling short term peak demand for a warehouse operating a Robotic Mobile Fulfillment System (RMFS), with rates alternating between high and low levels. We look at two short term peak demand situations, (1) the "daily" situation where high demand occurs in the evening and low demand during the rest of the day, and (2) the "week" situation, where high demand occurs during the weekend and low demand during working days. The RMFS is a parts-to-picker au-

tomated material handling system particularly suited for e-commerce warehouses. A parts-to-picker system prevents the pickers from having to walk long distances and it accommodates integrating packaging and other services within the fulfillment process. Moreover, it lends itself well to picking individual units rather than cases and pallets. By letting the worker focus on handling units and eliminating walking time, high pick rates can be achieved with an RMFS in an e-commerce environment, see also [38].
An RMFS works as follows. Robots bring pods, i.e. shelves containing inventory, to workstations. At a workstation, the robot with pod queues until its turn to be handled. A workstation is either a pick station, where the worker retrieves inventory from the pod to fulfill a customer order, or a replenishment station, where the worker places inbound inventory on the pod. Each time a pod is brought back to the storage area, it can be stored at a different location. This makes it possible to continually and automatically sort the inventory during operations and to keep the pods with popular SKUs close to the pick stations and nearly empty pods close to the replenishment stations. Figure 5.1a shows a robot carrying a pod with items, while Figure 5.1b shows a schematic top view of a workstation, with robots with pods queueing in the buffer.



(a) Robot carrying a pod [16]

(b) Top view of a workstation

Figure 5.1 A robot and a workstation in the RMFS

The two main processes in an RMFS are the picking and the replenishment processes. When an order arrives, it is assigned to a pick station, where it waits until all the items it needs are picked from the pods brought to the pick station. Whereas picking involves handling one or a few products per pod each in a small quantity, replenishment involves multiple units per product. When suppliers deliver goods at

the warehouse they are usually first stored in a reserve area elsewhere in the warehouse. Only when the system needs units of a product, a pallet with that product is sent to a replenishment station, where a worker can then put individual units on the pods [38].
One of the main characteristics of an RMFS is the ability to reallocate workers from pick stations to replenishment stations and reallocate robots from working on picking tasks to working on replenishment tasks. This characteristic is known as the dynamic reallocation of resources across picking and replenishment activities, where the workers and robots are the resources. Dynamic reallocation of resources can be achieved quickly and in real time in an RMFS. The workstations in an RMFS are suitable for both picking and replenishment. Pick stations can be converted to replenishment stations in a short amount of time and vice versa. Alternatively, it is possible to have additional pick and replenishment stations that can be opened and closed on demand. For example, if the number of pick stations needs to be decreased by one and the number of replenishment stations increased by one, a worker at a pick station can close that station, move to a closed replenishment station, and open that replenishment station. A robot can be assigned to either a pick task or a replenishment task, after it returns a pod to a storage location. In other words, robots can in principle be reallocated continually. Therefore, set-up or switching costs due to resource reallocation are relatively low in an RMFS, and can occur frequently to maximize performance under changing order arrival rates.
Resource reallocation does influence other costs, in particular those related to the customer waiting time and inventory availability, which depend on both the picking and the replenishment process. Too few resources in the picking process means that the customer waiting time will become too long, a metric that e-commerce companies are keen on minimizing. Inventory availability, however, increases as the replenishment process has more resources and pods are quickly replenished. With more resources allocated to the picking process, customer waiting time should decrease as products are fulfilled more quickly. However, if too few resources are allocated to the replenishment process, the inventory level in the storage area becomes low. Inventory availability in an RMFS is tied to the inventory held on the pods; RMFSs typically have only a few days worth of inventory and a stock-out in the storage area may mean unavailability of a product for at least a day ([39] and [38]). In e-commerce, low inventory availability may lead to lost sales as customers will place their order with a competitor that does have stock immediately available. The resources are the robots and the workers, the reallocation happens between the picking and replenishment process, the costs are those related to customer waiting time and lost sales, and strong demand fluctuation means alternating between a high and a low order arrival rate. This paper focuses on using resource reallocation to minimize costs under strong demand fluctuation, modeled as an Markov

Modulated Poisson Process (MMPP). The number of workers is kept fixed, so that wages do not factor into the costs. Furthermore, resource reallocation can happen frequently as set-up or switching costs are considered negligible. The goal is to find an optimal policy for allocating robots and workstations to picking and replenishment activities given the number of unfulfilled orders and the number of pods that need replenishment. The optimal policy must balance the cost of customer waiting time with the cost of unavailability of inventory. The optimal policy is compared to two benchmark policies used commonly in practice. The contributions of this paper are threefold. (1) We show how to extend the queueing networks used for modeling an RMFS in the literature, to include both robots and inventory pods rather than either one of these two. We also show how to integrate the extended queueing networks into an MDP model. (2) We consider two resources to be used in two processes, whereas previous work studies how to optimize the use of only one resource in only one process. This generalization allows us to study the trade-off of reallocating resources between the picking process and the replenishment process, which influences customer waiting time and inventory availability. This trade-off is particularly relevant for e-commerce warehouses, as they need to deliver quickly while also maintaining sufficient inventory to prevent lost sales. (3) Our approach shows the benefits of being able to reallocate resources quickly and frequently. The cost reductions achieved in an RMFS, where resources can be reallocated quickly and frequently, are compared with cost reductions under benchmark policies where resources are reallocated less often. (4) We obtain optimal policies while modeling time-varying demand using a two-phase MMPP.
This paper is structured as follows: Section 5.2 discusses the literature, Section 5.3 develops a queueing network to model the performance of the system given a fixed sets of resources, Section 5.4 develops a Markov Decision Process (MDP) model for finding the optimal policy of allocating resources, Section 5.5 gives the stability conditions for this MDP and the structure for the optimal policy, Section 5.6 validates the performance of the queueing network by comparing it with a detailed, realistic simulation of an RMFS, and shows the results of applying the MDP model to several case studies, and Section 5.7 draws conclusions and explores promising avenues for further research.

5.2 Literature

This paper studies the RMFS under strong demand fluctuations and resource reallocation. We therefore first examine the RMFS literature, especially existing queueing networks for the RMFS, second we discuss how demand fluctuations are typically modeled in stochastic models as a Markov Modulated Poisson Process (MMPP), thirdly we look at MDPs have been used in warehouse situations, and finally we re-

view how MDPs have been used for resource allocation in manufacturing situations. The RMFS is described generally and compared with other robotized warehousing systems in [3], and described in more detail by [16] and [39], the developers of the first RMFS. They identify numerous, challenging operational decision problems, such as path planning, task allocation, order assignment to pick stations, pod storage allocation, and inventory pod selection. Path planning in an RMFS has been studied by [25], while [26] address the above-mentioned decision problems together, except for path planning. [10] provide methods for optimally batching and sequencing picking orders, and for sequencing the pods transported to a pick station. In contrast, both [21], [22], [40], [41] and [30] focus on the warehouse design aspects of an RMFS. They model the operations in an RMFS using queueing networks. [30] build queueing network models to analyze the throughput time of single-line orders in an RMFS. [21] design queueing models that incorporate realistic movement of the robots, storage zones and multi-line orders in an RMFS. These models provide accurate estimates for workstation and robot utilization and order throughput time. They find how the dimensions for the storage area and placement of workstations around the storage area impact the throughput times of single- and multi-line orders. [40] examine how to determine the routing probabilities, i.e. the probability with which a robot chooses a certain destination workstation for transporting the pod. They analytically derive the best size of a storage block. Moreover, they find that an assignment rule incorporating the different handling speeds of the workers leads to significantly better routing probabilities, and they provide a neighborhood search algorithm that performs close to optimal.
[41] study battery management in RMFSs, and compare three policies: battery swapping, automated plug-in charging, and inductive charging. They build a semi-open queueing network to evaluate these policies and conclude that, if robot prices are low and required retrieval transaction throughput time are small, inductive charging outperforms the other two policies in annual costs. They also find that ignoring battery recovery underestimates the system costs and the number of robots required.
[22] study three key variables, namely the number of pods per product, the ratio of the number of pick stations to replenishment stations, and the replenishment level. They find that these variables interact and together affect the order throughput time and the stability of the system in the long run. Their model expands the work by [21] by including replenishment in the queueing network. They build a new type of Semi-Open Queueing Network (SOQN), the cross-class matching SOQN, to analyze the pick and replenishment operations simultaneously.
In e-commerce fulfillment applications, the RMFS has to deal with strong demand fluctuations and periods of peak demand. [36] show that assuming that demand is stationary when it is not, can lead to high total expected costs, even when the

optimal stationary inventory policy is used. Peak demand or seasonal demand can be incorporated in stochastic models by modeling the order arrival process as a Markov Modulated Poisson Process (MMPP) [17]. [32] and [19] explore the effect of using an MMPP arrival process for queueing systems and derive the properties of the waiting time, idle time and busy period. [13] construct fast algorithms for solving queueing systems with MMPP inputs and [15] solve a Semi-Open Queueing Network (SOQN) with MMPP input using the Matrix Geometric Method.
In other words, demand fluctuates and orders arrive at the warehouse stochastically. It is therefore often modeled as a Markov Decision Process (MDP), where the problem is to optimize the flow of goods. [35] and [2] use an MDP to optimally control transshipments between warehouses. The paper of [24] focuses more on optimal replenishment. The author studies an inventory system with two arrival processes: the standard order arrival process and the return of goods. The author uses an MDP to model the dynamics of the returns of goods and to derive optimal replenishment policies.
The resource allocation problem has been studied in the context of manufacturing. The resources are a number of identical machines, which can be allocated production, and the problem is to find the optimal production rate while keeping inventory (costs) under control. [14] shows how to optimally control a production system with variable service rates. [7] derive the structure of the optimal production and inventory policies for a manufacturing system subject to MMPP arrivals. [4] model the problem of finding optimal production rates for a manufacturer with one product serving multiple classes of customers under seasonal demand as an MDP. They demonstrate how to calculate the optimal production rates and show that the optimal rates are season-dependent. [8] show the value of having flexibility in both the production rates and the inventory levels in a manufacturing system subject to seasonal demand. [5] look at a similar environment but focus on the characteristics of the seasonal demand. They examine how the differences in average demand, in average season duration and duration variability, in the randomness of the sequence of seasons and in skewed seasonality have an effect on a manufacturing system where production rates and inventory levels can be flexibly controlled. [6] build on the previous work by adding service level constraints to each season. In all of these studies, the model includes one type of resource and one type of process to allocate the resource to.
The queueing networks in [21] and [22] model either robots or pods, but the research in this paper needs to model both resource types. The robots need to be included in the model as they are reallocated between the picking and replenishment process, and the pods are needed to model inventory availability. In addition, existing resource reallocation MDP models, such as the models of [8], [5] and [6] in a manufacturing settings, only model one process and one resource, whereas we examine

a problem with two processes, picking and replenishment, and two resource types, workers and robots. Section 5.3 describes the queueing network used in this paper, and section 5.4 describes the MDP for two processes and two resources and how to integrate the queueing network in that MDP.

5.3 Queueing Models

This section develops a queueing network model to estimate the system performance when the resources allocated to pick and replenishment activities are fixed. This work builds on and extends the queueing networks in [21] and [22].

5.3.1 Assumptions

The model makes the following assumptions. (1) The orders arrive according to a Markov Modulated Poisson Process (MMPP). We capture the time-varying nature of demand with different demand seasons. Within each demand season, the orders arrive according to a Poisson process. As [29] point out, a Poisson process is a reasonable assumption if a large customer base makes ordering decisions independently and under similar conditions. (2) During a period of high demand, the system can be temporarily unstable, with queues building up, as long as together with periods of low demand the system is stable in the long run. (3) We aggregate the SKUs to one SKU. We are only interested in the total workloads for the picking and replenishment process, so this simplifying assumption allows us to focus on the behavior of the system as a whole and not on individual products. We elaborate more on this assumption at the end of this subsection. (4) At the replenishment stations there are always pallets waiting with goods that need to be placed on the inventory pods. (5) Robot congestion or gridlock does not occur. Unloaded robots can move underneath parked pods, which gives them more possible routes and options to adjust their chosen route than robots carrying pods, which have to travel in the aisles. We assume single-directional travel in the aisles, which strongly decreases the congestion effects and the likelihood of gridlock. (6) The robot dwelling policy is the Point Of Service Completion (POSC) policy.
Two assumptions need further elaboration, namely assumption (1) about MMPP arrivals, and assumption (3) requires more discussion on the probability that a pod needs to go to a replenishment station after visiting a pick station. This probability is denoted by q. In an MMPP arrival process, the external demand is modeled as an irreducible continuous time Markov Chain, where the order arrival rate depends on the state of the external demand. As long as external demand is in a state s, orders arrive to the system according to a Poisson process with rate λ_s. When external demand moves from state s to another state s', orders start to arrive according to a

Poisson process with rate $\lambda_{s'}$ instead of rate λ_s. External demand moves from state s to another state according to an exponential process with average time ν_s^{-1} [17]. In e-commerce applications, demands varies throughout the year, with many peaks and troughs. [38] shows a graph illustrating such a demand pattern. A MMPP distribution with two phases seems to fit the data well, namely low demand with arrival rate λ_l and average duration ν_l^{-1}, and high demand with arrival rate λ_h and average duration ν_h^{-1}. The latter phase of high demand models periods of peak demand.
Assumption (3) means that we treat all SKUs as if they were the same, so in our model we effectively have only SKU. The advantage of aggregating all SKUs into one SKU is that fluctuations of individual SKUs are canceled out. One SKU may experience a high demand while another is experiencing a low demand, and by aggregating all SKUs such individual differences are averaged out. One, aggregated SKU is more realistic in conjunction with our MMPP distribution, which models periods of high and low demand for the warehouse as a whole. A consequence of aggregating the SKUs to one SKU in the model, is that an order can be immediately matched with a pod when it arrives at the system. This is realistic as in practical implementations of an RMFS, it is unlikely that an order would be released for which no stock is available in the storage area. The main disadvantage of modeling only one SKU rather than all products separately, is that we cannot keep track of the number of units per product on each pod, which would have allowed us to accurately model when a pod needs to go to replenishment. However, as long as we can accurately estimate the probability q that a pod needs to go for replenishment after visiting a picking station, the average customer waiting time and the availability of inventory are not affected. The probability q is exogenous to our model and can be observed in real RMFS applications. It is not the probability that a pod directly goes to a replenishment station after picking, since the pod may spend some time in storage before a robot becomes available to collect it from storage and to transport it. Rather, q is the probability that the pod will be needed for replenishment, either because it has run low on inventory itself, or because a product is running low on inventory and the pod is needed for storing the units of that product.

5.3.2 Description of the Complete Queueing Network

The complete queueing network is shown in Figure 5.2. Our network is a special class of queueing networks known as semi-open queueing networks (SOQN)(see [34]). SOQNs are particularly useful to capture external transaction waiting times and capture the synchronization between the orders and the resources. It is open with respect to the orders and closed with respect to the resources. Each queue in Figure 5.2 is labeled and in the following, queue $[x]$ will refer to queue number x in Figure 5.2. Three types of entities move through this network: orders, robots, and

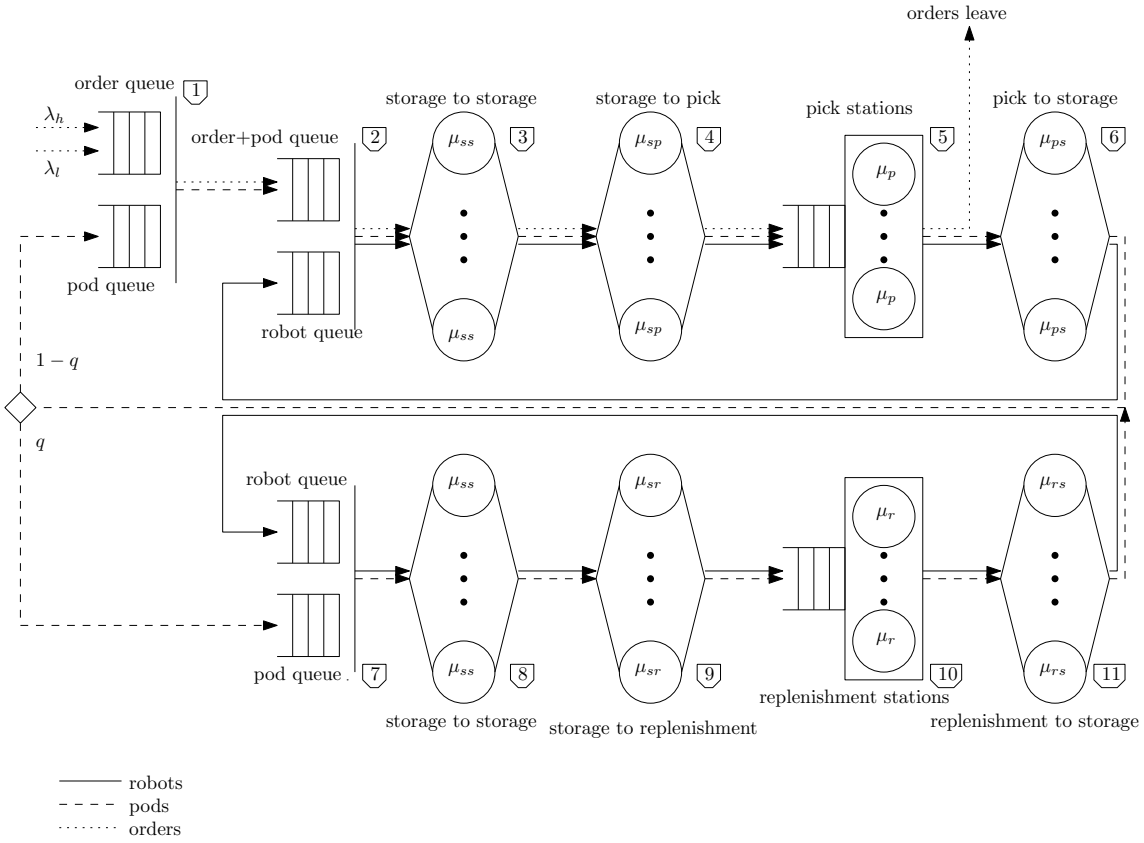


Figure 5.2 The Complete Semi-Open Queueing Network

pods. The robots are dedicated to either picking or replenishment tasks and cannot switch to a different task.

The picking process works as follows, see Figure 5.2. Orders arrive at a synchronization station, queue [1], at either a low arrival rate λ_l or a high arrival rate λ_h. The time that a period of low demand or of high demand lasts is exponentially distributed with mean ν_l^{-1} and mean ν_h^{-1}, respectively. An order is matched at queue [1] with a suitable pod, after which the pod with assigned order is matched with an idle picking robot at the next synchronization station, queue [2]. The picking robot with assigned order and pod then moves through two Infinite Server (IS) queueing stations, queues [3] and [4], that model ([3]) the travel from the dwell location of the picking robot to the storage location of the pod and ([4]) the travel from the storage location of the pod to a pick station. The service times of the IS queueing stations correspond to the respective traveling times. The distributions can be calculated by assigning probabilities to every storage location, that indicate the likelihood of pod retrieval from that storage location for transport. As the travel times between any

two points in the warehouse can be calculated using closed form expressions [21], these probabilities and travel times together completely describe the distribution of the travel times, without the need for any further assumptions. In other words, the travel times follow an empirical distribution.
The RMFS has W_p pick stations, which are modeled as queue [5], a queueing station with W_p servers. Each server operates with a service rate μ_p. The picking robot with assigned order and pod arrives at queue [5], which corresponds to the robot arriving at the buffer of a pick station. After the picker has retrieved a unit from the pod, the order assigned to that pod leaves the system. The picking robot and its assigned pod proceeds through an IS queueing station, queue [6], which models travel from the pick station to the storage location where the pod must be returned. After storing the pod, the picking robot becomes idle and joins a synchronization station, queue [2] where it can be matched with a new pod and order coming from queue [1]. The picking robot will continue this cycle indefinitely, but the pod only continues this cycle until it has to visit a replenishment station. After every cycle, the pod has to visit a replenishment station with a probability q. In that case it joins a synchronization queue, namely queue [7], which is associated with the replenishment process. Otherwise, it goes to the synchronization station where it can be matched with orders, queue [1].
The replenishment process works as follows. At queue [7], a pod that needs replenishment is matched with an idle replenishment robot. The replenishment robot with its assigned pod then moves through four queueing stations, queues [8], [9], [10] and [11], that are similar to queues [3], [4], [5] and [6] in the picking process. The pod is replenished at queue [10]. The replenishment robot continues this cycle indefinitely, but the pod moves to a synchronization station, queue [1], where it can be matched with an assigned order.
In Figure 5.2, μ_{ss}^{-1} is the average robot travel time from a random storage location to another random storage location, μ_{sp}^{-1} is the average robot travel time from a random storage location to a random pick station, μ_p^{-1} is the average time that a picking operation takes, μ_{ps}^{-1} is the average robot travel time from a random pick station to a random storage location, μ_{sr}^{-1} is the average robot travel time from a random storage location to a random replenishment station, μ_r^{-1} is the average time that a replenishment operation takes and μ_{rs}^{-1} is the average robot travel time from a random replenishment station to a random storage location. As mentioned earlier, the travel times follow an empirical distribution. The pick times at the pick stations follow an exponential distribution with mean μ_p^{-1}, and replenishment times at replenishment stations an exponential distribution with mean μ_r^{-1}. However, these distributions can also be empirical distributions if there is empirical data available on pick and replenishment times.

5.3.3 The Compact Network

Section 5.4 develops an MDP model for analyzing dynamic resource reallocation. This MDP model needs to incorporate the service rates of the picking process and the service rates of the replenishment process. Incorporating the complete queueing network in Figure 5.2 would lead to a state space that even for small instances becomes too large for analysis, It would have to keep track of the MMPP phase and all the orders, pods and robots at the 11 queueing stations, leading to a high dimensional state space. To keep the size of the state space sufficiently small, this section transforms the complete queueing network in Figure 5.2 to a more compact queueing network that makes it amenable to analysis. Note that the network dynamics are still captured in the compact network model.
The transformation from the complete to the compact network is done in two steps and works as follows. The first step is transforming the complete network to an intermediate network. From the complete queueing network, the queues related to the replenishment process, namely queues [8], [9], [10] and [11], are used to form a Closed Queueing Network (CQN), as shown in Figure 5.3. In this CQN, let the throughput rate with i robots in the CQN be equal to $\hat{\mu}_r(i)$. Using Norton's theorem, we can transform this CQN into a single equivalent load-dependent queue with service rates $\hat{\mu}_r(i)$ (see [37]). The service rates $\hat{\mu}_r(i)$ for $1 \leq i \leq R$ can be found by varying the number of robots i in the CQN from 1 to R and calculating the resulting throughput rates. By convention, the service rate of an empty load-dependent queue is zero, so $\hat{\mu}_r(0) = 0$.

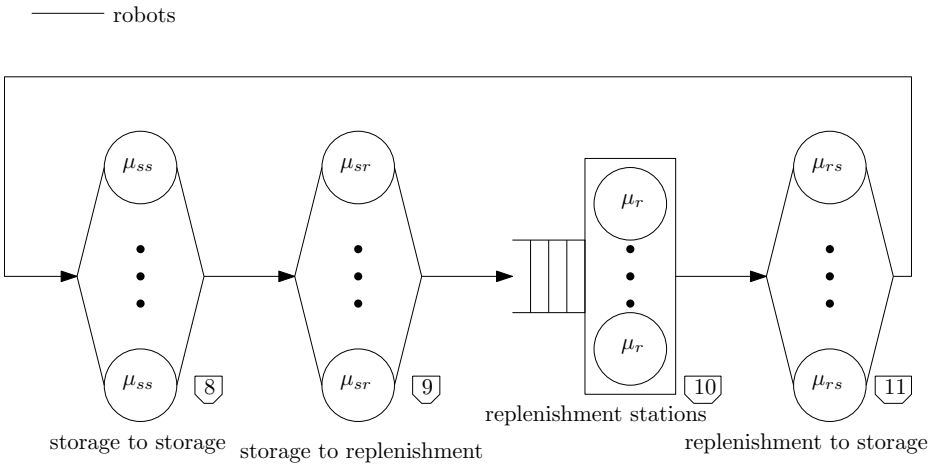


Figure 5.3 Part of the complete queueing network associated with the replenishment process

The first step, where the complete network in Figure 5.2 is transformed to the intermediate network in Figure 5.4, is an approximation, because the travel times follow an general distribution (estimated empirically), hence reduction of the network into a compact network using Norton's theorem does not result in exact results (see also [37] and [23]). We use the AMVA algorithm of [11] to calculate the throughput times of the CQNs, $\hat{\mu}_r(i)$, see also Appendix 5.8.

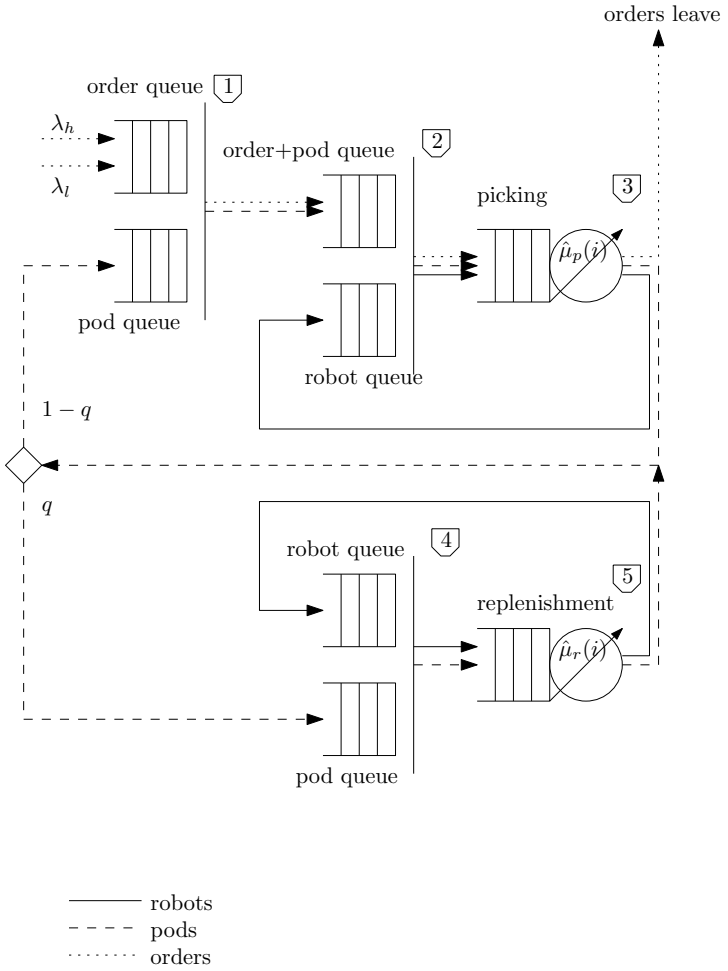


Figure 5.4 The Intermediate Queueing Network

The resulting load-dependent queue representing the replenishment process is shown in Figure 5.5, where λ_p is the arrival rate of robots with pods to the replenishment process. In the same way, we can create a load-dependent queue with service rate $\hat{\mu}_p(i)$ representing the pick process.

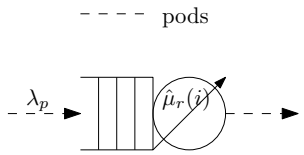


Figure 5.5 A load-dependent queue representing the replenishment process in Figure 5.3

We then create an intermediate queueing network by replacing queues [8], [9], [10] and [11] in the complete queueing network by the load-dependent queue representing the replenishment process, and by replacing queues [3], [4], [5] and [6] with the load-dependent queue representing the pick process. The intermediate queueing network is shown in Figure 5.4.
In Figure 5.4, $\hat{\mu}_p(i)$ is the service rate of the picking process when i pick robots are busy with activities related to picking, and $\hat{\mu}_r(i)$ is the service rate of the replenishment process when i replenishment robots are busy with activities related to replenishment.
The second step is to simplify the intermediate queueing network further in order to arrive at the compact network. A complication is that the arrival rate of robots with pods, λ_p, is unknown for both the pick and replenishment process. In other words, in Figure 5.4, we do not know the arrival rate or robots carrying pods for queues [3] and [5], representing the picking and replenishment process respectively. Therefore, we use an approximation to simplify the intermediate network in Figure 5.4. Appendix 5.9 shows how to transform a queueing network consisting of a synchronization station connected to a load-dependent queue into a queueing network consisting only of a load-dependent queue. It makes it possible to eliminate two of the three synchronization stations in the intermediate queueing network. Queues [2] and [3] in the intermediate queueing network are replaced by a load-dependent queue with the same service rates $\hat{\mu}_p(i)$ as queue [3], while queues [4] and [5] in the intermediate queueing network are replaced by a load-dependent queue with the same service rates $\hat{\mu}_r(i)$ as queue [5]. The resulting queueing network is the compact queueing network shown in Figure 5.6. We do not need to actually solve this queueing network, since we are only interested in calculating $\hat{\mu}_p(i)$ and $\hat{\mu}_r(i)$.

The validation of the compact queueing network with a realistic and detailed simulation of an RMFS is shown in Section 5.6.
The next section introduces the MDP model, which entails adopting new notation. For the sake of convenience, Table 5.1 contains an overview of the notation used throughout the main text.

Table 5.1 Notation overview

Symbol(s)	Meaning
a, A	a is an allocation of workstations and robots to the pick process, and A is the set of all allocations, so $a \in A$
C_{cwt}, C_{ls}	the cost of customer waiting time, and of lost sales, respectively
C_p, C_r	the cost associated with picking and replenishment, respectively
f^*, f^F, f^{DD}, f^{SD}, f^{CD}	the optimal, fixed, demand dependent, state dependent, and cost dependent policies, respectively
M, M_p, M_r	the number of pods in total, in the pick process, and in the replenishment process, respectively
O_σ, O_t	the number of orders at the synchronization station, and the total number of orders in the system, respectively
q	the probability for a pod to go to replenishment after picking
R, R_p, R_r	the number of robots in total, in the pick process, and in the replenishment process, respectively
s, $\mathcal{S}$	s is a state in the MDP and $\mathcal{S}$ is the set of all states, so $s \in \mathcal{S}$
W, W_p, W_r	the number of stations in total, allocated to the pick process, and to the replenishment process, respectively
γ_s	the fraction of orders that are rejected from the system
λ_Δ, λ_h, λ_l	the arrival rate in phase Δ, during a period of high demand, and during a period of low demand, respectively
$\bar{\lambda}$, λ_e	the long term average order arrival rate to the system, and the effective arrival rate (excludes rejected orders), respectively
$\bar{\rho}$	the (long term) average utilization of the systems
μ_p, μ_r	the avg. pick rate and the avg. repl. rate, respectively
μ_{ss}^{-1}, μ_{sp}^{-1}, μ_{ps}^{-1}, μ_{sr}^{-1}, μ_{rs}^{-1}	the avg. robot travel time from storage to storage, storage to pick station, pick station to storage, storage to replenishment station,and replenishment station to storage, respectively
$\hat{\mu}_p^a(i)$, $\hat{\mu}_r^a(i)$	the rates at which a pod leaves the picking process, and the replenishment process, respectively, given that there are i pods and given an allocation a
ν_Δ^{-1}, ν_h^{-1}, ν_l^{-1}	the average duration time of a phase Δ, of a period of high demand, and of a period of low demand, respectively
ψ_s	the stock-out probability for a SKU
ω	the order cap, $O_\sigma \leq \omega$
ζ	parameter used for approximating the stock-out probability

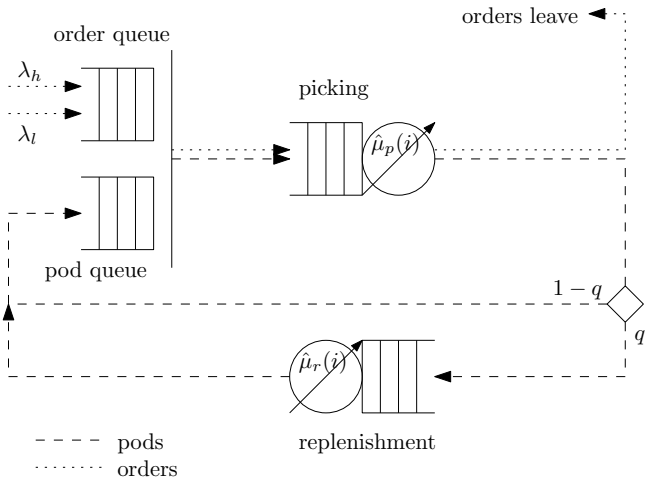


Figure 5.6 The Compact Queueing Network

5.4 The Markov Decision Process

This section explains the MDP model used to find the optimal policies for resource reallocation of the workers and robots. It is a discretized MDP with a discrete action space embedded in continuous time with an infinite time horizon. The two resources that can be reallocated are workstations and robots, which can be allocated to either picking or replenishment activities. We will assume that the number of workstations, W, and the number of robots, R, are fixed, so that determining the number of pick stations implicitly determines the number of replenishment stations, and determining the number of pick robots implicitly determines the number of replenishment robots. The number of pick stations is denoted by W_p, the number of replenishment stations by W_r, the number of pick robots by R_p, and the number of replenishment robots by R_r. A resource allocation a describes the number of pick stations, W_p, and the number of pick robots, R_p, that will be used and thus implicitly the number of replenishment stations, W_r, and the number of replenishment robots, R_r. In other words, an allocation a is given by $a = (W_p, R_p)$, and implicitly sets W_r and R_r as $W_r = W - W_p$ and $R_r = R - R_p$.

5.4.1 A State in the MDP

A state s in the MDP is constructed as follows. The first element that describes a state s is the demand phase Δ, where $\Delta = l$ in the low demand phase of the MMPP and $\Delta = h$ in the high demand phase of the MMPP. The order arrival rate is denoted by λ_Δ, and becomes λ_l and λ_h in the low and high demand phase, respectively. The second element of a state s is O_σ, the number of orders in the order

queue that have not been finished yet. The total number of pods in the system is denoted by M, the number of pods in the pick process by M_p, and the number of pods in the replenishment process by M_r. The third element of a state s is M_p and the fourth element is M_r. M_p and M_r indicate the workload of the picking and replenishment process respectively and therefore contain vital information for resource reallocation decisions. The number of pods stored in the storage area not involved in any activities is implicitly given by $M - M_p - M_r$. Taken together, a state s in the MDP is given by $s = (\Delta, O_\sigma, M_p, M_r)$.

In a state s, five different types of transitions are possible with reference to the compact queueing network in Figure 5.6. These transitions are denoted by letters A to E: (A) an order arrives to the order queue, (B) the picking of a pod finishes, the order assigned to that pod leaves the system, and the handled pod moves from the picking queue to the pod queue, (C) the picking of a pod finishes, the order assigned to that pod leaves the system, and the handled pod moves from the picking queue to the replenishment queue, (D) the replenishment of a pod finishes and the handled pod moves from the replenishment queue to the pod queue, and (E) the demand phase rate Δ changes. After a transition, if Δ was h, it is l, and if Δ was l, it is h.

In an MDP, at each transition from a state s, an action has to be chosen from the action space $A = \{a_1, \ldots, a_i, \ldots, a_n\}$. Here n is the number of possible resource allocations. The set A of possible resource reallocations is the same for all states.

Theoretically, the total number of possible resource allocations is $n = (W + 1) \times (R + 1)$, as between 0 and W workstations can be allocated to picking and between 0 and R robots can be allocated to picking. However, not every possible allocation need to be included in A. It is not necessary, for example, to include an allocation with one pick robot and multiple pick stations, as that is quite clearly inefficient. In addition, the action space A should be limited to only a few possible actions for the sake of solving the MDP in numerical experiments.

The compact queueing network in Figure 5.6 assumes that the number of pick stations, replenishment stations, pick robots, and replenishment robots is fixed. Therefore, for every action $a \in A$, we need to solve a different queueing network to find the rates at which pods complete the picking and replenishment processes. Given a resource allocation a, the service rates for the picking process are denoted by $\hat{\mu}_p^a(M_p)$ and the service rates for the replenishment process by $\hat{\mu}_r^a(M_r)$.

The transition probabilities can be constructed as follows. For a transition of type (A), the system moves out of a state s with rate λ_Δ, the rate at which orders arrive at the system. For transitions (B) and (C), given a resource allocation a, the rate is $\hat{\mu}_p^a(M_p)$, the rate at which the picking of a pod finishes given that M_p pods are involved in the picking process. Similarly for transition (D), the rate is $\hat{\mu}_r^a(M_r)$, the rate at which replenishment of a pods finishes, given that M_r pods are involved in the replenishment process. Transition (E) occurs with rate ν_Δ, the rate at which

the order arrival rate changes in state s. For any state s, the total rate at which the system moves out of s is thus $\lambda_\Delta + \hat{\mu}_p^a(M_p) + \hat{\mu}_r^a(M_r) + \nu_\Delta$ The transition probability $P^a(s'|s)$ denotes the probability to move from a state s to a state s' when choosing resource allocation a.
To ensure that the queueing network and the system are both stable, we will assume that there is a cap ω on the number of orders at the synchronization station. The number of orders in the system, O_t, consists of two components: (1) the number of orders waiting at the synchronization station in the compact queueing network, O_σ, and (2) the number of orders being processed, which equals the number of pods involved in picking activities, M_p, i.e., pods being transported to or from the pick stations or being handled at the pick stations. In other words, in any state s, $O_t = O_\sigma + M_p$ and $O_\sigma \leq \omega$. If $O_\sigma = \omega$, no orders can enter the system and thus transition (A) cannot occur. If $M_p = 0$, transitions (B) and (C) cannot occur and their transition probability is zero. Similarly, if $M_r = 0$ transition (D) cannot occur and the corresponding transition probability is zero. The transitions and their probabilities are shown in Table 5.2. In Table 5.2, $\tilde{\Delta}$ is as follows: if $\Delta = h$, then $\tilde{\Delta} = l$, and if $\Delta = l$, then $\tilde{\Delta} = h$. The MMPP arrival process has thus been explicitly woven into the transition probabilities of the MDP.

Table 5.2 Possible transitions from a state $s = (\Delta, O_\sigma, M_p, M_r)$, with $\Delta \in \{l, h\}$, and $a \in A$

Type	Condition(s)	Transition probability		
(A)	$O_\sigma = \omega, M_p + M_r = M$	$P^a(\Delta, O_\sigma, M_p, M_r \mid s)$	$=$	$\frac{\lambda_\Delta}{\lambda_\Delta + \hat{\mu}_p^a(M_p) + \hat{\mu}_r^a(M_r) + \nu_\Delta}$
(A)	$O_\sigma < \omega, M_p + M_r = M$	$P^a(\Delta, O_\sigma + 1, M_p, M_r \mid s)$	$=$	$\frac{\lambda_\Delta}{\lambda_\Delta + \hat{\mu}_p^a(M_p) + \hat{\mu}_r^a(M_r) + \nu_\Delta}$
(A)	$O_\sigma < \omega, M_p + M_r < M$	$P^a(\Delta, O_\sigma, M_p + 1, M_r \mid s)$	$=$	$\frac{\lambda_\Delta}{\lambda_\Delta + \hat{\mu}_p^a(M_p) + \hat{\mu}_r^a(M_r) + \nu_\Delta}$
(B)	$O_\sigma = 0, M_p > 0$	$P^a(\Delta, O_\sigma, M_p - 1, M_r \mid s)$	$=$	$\frac{(1-q)\hat{\mu}_p^a(M_p)}{\lambda_\Delta + \hat{\mu}_p^a(M_p) + \hat{\mu}_r^a(M_r) + \nu_\Delta}$
(B)	$O_\sigma > 0, M_p > 0$	$P^a(\Delta, O_\sigma - 1, M_p, M_r \mid s)$	$=$	$\frac{(1-q)\hat{\mu}_p^a(M_p)}{\lambda_\Delta + \hat{\mu}_p^a(M_p) + \hat{\mu}_r^a(M_r) + \nu_\Delta}$
(C)	$M_p > 0$	$P^a(\Delta, O_\sigma, M_p - 1, M_r + 1 \mid s)$	$=$	$\frac{q\hat{\mu}_p^a(M_p)}{\lambda_\Delta + \hat{\mu}_p^a(M_p) + \hat{\mu}_r^a(M_r) + \nu_\Delta}$
(D)	$O_\sigma = 0, M_r > 0$	$P^a(\Delta, O_\sigma, M_p, M_r - 1 \mid s)$	$=$	$\frac{\hat{\mu}_r^a(M_r)}{\lambda_\Delta + \hat{\mu}_p^a(M_p) + \hat{\mu}_r^a(M_r) + \nu_\Delta}$
(D)	$O_\sigma > 0, M_r > 0$	$P^a(\Delta, O_\sigma - 1, M_p + 1, M_r - 1 \mid s)$	$=$	$\frac{\hat{\mu}_r^a(M_r)}{\lambda_\Delta + \hat{\mu}_p^a(M_p) + \hat{\mu}_r^a(M_r) + \nu_\Delta}$
(E)	-	$P^a(\tilde{\Delta}, O_\sigma, M_p, M_r \mid s)$	$=$	$\frac{\nu_\Delta}{\lambda_\Delta + \hat{\mu}_p^a(M_p) + \hat{\mu}_r^a(M_r) + \nu_\Delta}$

5.4.2 Solving the MDP Model

The goal is to allocate resources in such a way that the cost associated with customer waiting time and lost sales are minimized. Each workstation has exactly one worker, and since the number of workstations is fixed, wages can be excluded from the analysis. An important cost factor is the customer waiting time, which in an e-commerce setting should be kept as low as possible. Another import cost factor is the availability of inventory, as not having products in stock results in a long delay in the fulfillment of an order, or possibly lost sales. [33] argues that the average cost criterion is suitable when decisions are made frequently, because then the discount factor of future costs is close to one. For the MDP in this paper that is arguably the case, as state transitions typically happen within seconds of each other. Therefore, the optimality criterion used in this paper is the average costs per time unit. Minimizing the average customer waiting time can be done by minimizing the average number of orders in the system, as the arrival process is given. The order cap ω leads to lost sales when $O_\sigma = \omega$, which creates an additional cost C_{ls} per lost order (lost sales). When $O_\sigma = \omega$, lost sales occur at the order arrival rate λ_Δ. During the high demand phase, the order arrival rate exceeds the system capacity, meaning that some of orders need to be processed during the low demand phase. Another interpretation for ω is that it is the maximum number of orders that can be transferred from the high demand phase to the low demand phase (or vice versa, although that situation should be rather unlikely).
Let $\mathbb{1}^s_{[O_\sigma=\omega]}$ be a function that equals 1 if $O_\sigma = \omega$ in state s of the MDP and 0 otherwise, and let C_{cwt} be the customer waiting time cost per time unit per order. The unavailability of inventory is approximated by using $\frac{M_r}{M}$ since this is the fraction of empty pods relative to the total number of pods. The cost of unavailability is thought of as lost sales: a fraction ψ_s of incoming orders finds that the storage area does not contain the desired product and the customers buy the product elsewhere. In a typical RMFS, each SKU tends to be spread across multiple pods, therefore ψ_s increases nonlinearly in the number of empty pods. Appendix 5.10 discusses in more detail how ψ_s can be derived based on realistic data with multiple SKUs. Appendix 5.10 shows that $\left(\frac{M_r}{M}\right)^\zeta$ offers a reasonable approximation of the stock-out probability, with ζ a parameter. The formula is shown in Equation (5.1).
The order cap together with inventory unavailability mean that the fraction of incoming orders that are rejected from the system, denoted by γ_s, is as given in Equation (5.2). The total cost $C_T(s)$ per time unit to be in state s is given by Equation (5.3). The total cost C_T can be partitioned in a pick component and a replenishment component. Let the costs associated with picking be denoted with C_p and the costs associated with replenishment as C_r. Delays in the picking process causes customer waiting costs, and it may cause orders to be rejected if $\mathbb{1}^s_{[O_\sigma=\omega]} = 1$.

Delays in the replenishment process causes lost sales. Hence C_p and C_r are as given in Equations (5.4) and (5.5).

$$\psi_s = \left(\frac{M_r}{M}\right)^{\zeta} \tag{5.1}$$

$$\gamma_s = \max\left(\mathbb{1}^s_{[O_\sigma=\omega]}, \psi_s\right) \tag{5.2}$$

$$C_T(s) = O_t C_{cwt} + \gamma_s \lambda_\Delta C_{ls} \tag{5.3}$$

$$C_p = O_t C_{cwt} + \mathbb{1}^s_{[O_\sigma=\omega]} \lambda_\Delta C_{ls} \tag{5.4}$$

$$C_r = \psi_s \lambda_\Delta C_{ls} \tag{5.5}$$

A policy f describes for every state s which allocation a should be chosen when a transition occurs. For a state s, let π^f_s be the stationary probability to be in state s under a policy f. We want to solve the MDP for an infinite horizon, as we want to analyze the performance of the RMFS in the long run, and hence choose the expected cost minimization criterion. It therefore makes sense to try to minimize the expected costs. If unique stationary state probabilities π^f_s exist, the expected cost C^f_A per time unit under policy f can be expressed as:

$$C^f_A = \sum_{s\in\mathcal{S}} \pi^f_s C_T(s) \tag{5.6}$$

In Equation (5.6), $\mathcal{S}$ is the set of all states, and $C_T(s)$ is the total cost as described in Equation (5.3). We can now show that the average cost C^f_A per time unit under any policy f exists and is a finite quantity as follows. We have that $O_\sigma \leq \omega$, $M_r \leq M$, $O_t \leq \omega + M$, and $\lambda_\Delta \leq \lambda_h$. Furthermore, the number of states $s = (\Delta, O_\sigma, M_p, M_r)$ is finite as Δ can take two values, $O_\sigma \leq \omega$, and both M_p and M_r can take integer values from 0 to M. Therefore, the total cost $C_T(s)$ and the number of states are bounded as follows:

$$C_T(s) \leq (\omega + M) C_{cwt} + \lambda_h C_{ls} < \infty \tag{5.7}$$

$$|\mathcal{S}| \leq 2 \times \omega \times (M+1) \times (M+1) < \infty \tag{5.8}$$

Equation (5.8) shows that the number of states is finite, which means that the sum in Equation (5.6) is over a finite number of terms. As Equation (5.7) shows that $C_T(s) < \infty$, and since π^f_s is a probability, each of the summation terms in Equation (5.6) is finite. As a finite sum of finite terms is finite, it must hold that $C^f_A < \infty$. In other words, for each policy f the expected cost per time unit is finite.

An optimal policy f^* is a policy that minimizes the average cost per time unit. In other words, the optimality criterion is:

$$f^* \in \operatorname{argmin} f \; C_A^f \tag{5.9}$$

Unique stationary state probabilities π_s^f exist under policy f, if two conditions are met [9]. First of all, the Markov Chain describing the system must be aperiodic. Secondly, given a reference state s_0, the expected time $\tau(s, s_0)$ needed to go from any state s to the reference state s_0, must be bounded by a number $T < \infty$. In other words, $\tau(s, s_0) \leq T < \infty \quad \forall s \in \mathcal{S}$. In addition, the reference state s_0 must be positive recurrent.
For the MDP described above, we can define a reference state $s_0 = (\lambda_l, 0, 0, 0)$. In s_0 the MMPP phase is low, the system contains no orders, and no pods are involved in picking or replenishment. Given this definition of the reference state s_0, the conditions for the existence of the unique stationary state probabilities π_s^f coincide with the assumption that the system is stable under policy f. Therefore, an optimal policy f^* is unique and therefore we can refer to *the* optimal policy f^*. Let X be the Continuous Time Markov Chain describing the system and let X_n be the state of the Markov Chain at time n. The stationary state probabilities π_s^f, independent of the initial state i, are given by:

$$\pi_s^f = \lim_{n \to \infty} P^f \{X_n = s | X_0 = i\} \quad \forall s \in \mathcal{S} \tag{5.10}$$

The stationary state probabilities $\pi_s^{f^*}$ of the optimal policy f^* can be calculated by solving the following Linear Program:

$$\min_{x_{s,a}} \sum_{s \in \mathcal{S}} \sum_{a \in A} x_{s,a} C_T(s) \tag{5.11}$$

subject to

$$\sum_{a \in A} x_{s,a} = \sum_{i \in \mathcal{S}} \sum_{a \in A} x_{i,a} P^a(s|i) \qquad \forall s \in \mathcal{S} \tag{5.12}$$

$$\sum_{i \in \mathcal{S}} \sum_{a \in A} x_{i,a} = 1 \tag{5.13}$$

$$x_{s,a} \geq 0 \qquad \forall s \in \mathcal{S}, \forall a \in A \tag{5.14}$$

We have that $\pi_s^f = \sum_{a \in A_s} x_{s,a}$. In other words $x_{s,a}$ is the steady state fraction of time the system spends in state s and allocation a is chosen. Equation 5.11 is the objective function and minimizes the total costs of the system, which can also be written as $\sum_{s \in \mathcal{S}} \pi_s^f C_T(s)$. Constraint 5.12 incorporates the transitions between states. Constraint 5.13 ensures that in every state an allocation is chosen and con-

straint 5.14 ensures that allocations cannot be chosen a negative number of times. By solving the linear program, we obtain $x^*_{s,a}$, the values for $x_{s,a}$ that minimize the costs. From $x^*_{s,a}$ we can calculate $\pi_s^{f^*} = \sum_{a \in A_s} x^*_{s,a}$, and we can also construct the optimal policy f^*. The optimal policy f^* is to choose allocation a in state s with a probability equal to $x^*_{s,a}$.
Instead of using Linear Programming, it is also possible to use a Dynamic Programming approach. The Bellman equation corresponding to the MDP in this paper is given in Equation (5.15):

$$v_t^*(s) = \min_{a \in A} \left\{ C_T(s) + \sum_{s' \in \mathcal{S}} P^a(s|s') v^*_{t-1}(s') \right\} \tag{5.15}$$

In Equation (5.15), t indicates the time period and $v_t^*(s)$ is the expected total cost of the system, if the system is in state s between at time point t. The Dynamic Program can be solved using well-known methods such as policy iteration or value iteration.

5.4.3 Benchmark Policies

To understand the benefits of using the optimal policy f^*, we will compare the minimal costs under policy f^* with four benchmark policies. Two of these benchmark policies are derived from practice, and two are customized for the MDP in this paper. The four benchmark policies are: (1) a fixed policy f^F that always uses the same allocation a^F regardless of the state s, (2) a demand-dependent policy f^{DD} that always uses an allocation a_h^{DD} for states with a high arrival rate and an allocation a_l^{DD} for states with a low arrival rate, (3) a cost-dependent policy f^{CD} that always uses an allocation a_p^{CD} if $C_p \geq C_r$, and a_r^{CD} if $C_p < C_r$, and (4) a state-dependent policy f^{SD}, where for each allocation we evaluate expected total cost after a transition. More specifically, given the current state s, let the set $\mathcal{S}'$ be the set of all possible states after a transition, and let $P^a(s'|s)$ be the probability to transit from the current state s to another state s'. The expected total cost $\tilde{C}(a)$ of an allocation a is then as given in Equation (5.16). The state-dependent policy f^{SD} chooses the allocation a^{SD} that results in the lowest expected total costs after the next transition, as shown in Equation (5.17).

$$\tilde{C}(a) = \sum_{s' \in \mathcal{S}'} C_T(s') P^a(s'|s) \tag{5.16}$$

$$a^{SD} = \arg\min_{a \in A} \tilde{C}(a) \tag{5.17}$$

The fixed policy f^F is similar to an RMFS where workstations and robots are dedicated to either picking or replenishment, whereas the demand-dependent policy f^{DD} is similar to an RMFS where workstations and robots are only dedicated to picking or replenishment during work shifts, but can be reallocated between shifts. The state-dependent policy f^{SD} chooses an allocation where the ratio between pick rates and replenishment rates matches the ratio between the costs associated with picking and the costs associated with replenishment as closely as possible. Some allocations a will reduce the number of waiting orders O_t by allocating the majority of workstations and robots to the picking workload, whereas other allocations a will primarily reduce the number of pods waiting for replenishment, M_r. Naturally, some allocations a will strike a balance between the picking and replenishment workloads. The fixed policy f^F always chooses a balanced allocation, whereas the demand-dependent policy chooses an allocation that mainly reduces O_t when the MMPP phase is high and an allocation that mainly reduces M_r when the MMPP phase is low. The main benefit of policy f^* may lie in the fact that it can choose to spend most workers and robots on picking when O_t is high even when the MMPP phase is low.

5.5 Stability Conditions

This section establishes necessary stability conditions for the system studied in this paper. The MMPP contains two phases, (1) low and (2) high, with respective stationary probabilities $\hat{\pi} = (\hat{\pi}_l, \hat{\pi}_h)$. The duration of these phases are exponentially distributed, with ν_l^{-1}, the average time for the low phase, and ν_h^{-1}, the average time for the high phase. The corresponding generator matrix is:

$$Q = \begin{bmatrix} -\nu_l & \nu_l \\ \nu_h & -\nu_h \end{bmatrix} \tag{5.18}$$

It must hold that $\hat{\pi}Q = 0$ and $\hat{\pi}\mathbf{1} = 1$. Solving the corresponding system of equations leads to Equation (5.19).

$$\hat{\pi}_h = \frac{\nu_h^{-1}}{\nu_h^{-1} + \nu_l^{-1}}, \qquad \hat{\pi}_l = \frac{\nu_l^{-1}}{\nu_h^{-1} + \nu_l^{-1}} \tag{5.19}$$

The average arrival rate $\bar{\lambda}$ is the average order arrival rate in the long run across all demand periods. The effective arrival rate λ_e is the long term average rate at which the orders are actually admitted to the system. The effective arrival rate λ_e takes into account that orders can become lost sales and do not enter the system, and it therefore depends on the chosen policy. The average arrival rate $\bar{\lambda}$ and the effective

arrival rate λ_e are defined in Equation (5.20) and (5.21), respectively. It naturally holds that $\bar{\lambda} \geq \lambda_e$.

$$\bar{\lambda} = \hat{\pi}_h \lambda_h + \hat{\pi}_l \lambda_l \tag{5.20}$$

$$\lambda_e = \sum_{s \in \mathcal{S}} \pi_s^f (1 - \gamma_s) \lambda_\Delta \tag{5.21}$$

For assessing upfront whether the system can be stable, λ_e cannot be used as it depends on the operational policy. Instead, we use $\bar{\lambda}$ to assess whether the system can be stable. Naturally, if $\bar{\lambda}$ is smaller than the order throughput capacity, TH of the system, $\bar{\lambda} \leq TH$, then the stability of the system is assured. However, the throughput of the system depends on the allocation a of both types of resources. Let TH^a be the system throughput if the allocation of resources is fixed to allocation a and the allocation does not change. In order to find TH, we eliminate the synchronization station from the compact queueing network in Figure 5.6 to arrive at the Closed Queueing Network in Figure 5.7. In the latter queueing network, pods do not have to wait for orders and hence the throughout is higher than in the former queueing network. Given an allocation a and M, let $TH^a(M)$ denote the throughout rate of the Closed Queueing Network in Figure 5.7. The network in Figure 5.7 can be analyzed with a single-class exact Mean Value Analysis (MVA) for Load-dependent queues as can be found in [20] and [11], and this yields $TH^a(M)$.

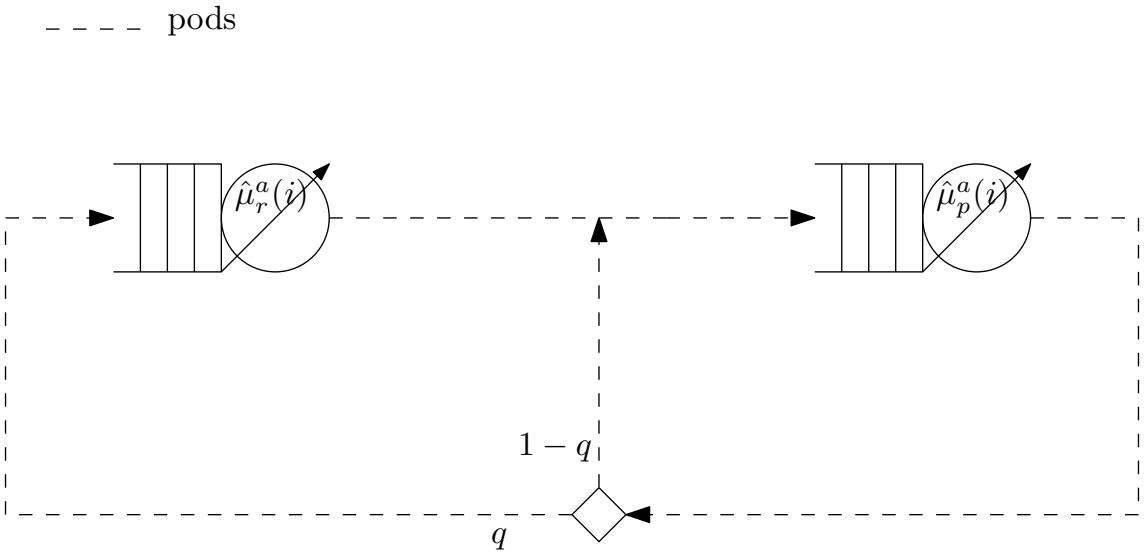


Figure 5.7 Closed queueing network based on the compact queueing network in Figure 5.6

TH_{max}, as given in Equation (5.22), is the maximum throughput the system can achieve if a fixed resource allocation is chosen, and TH_{min} is the minimum.

$$TH_{max} = \max_{a \in A} TH^a(M) \tag{5.22}$$

$$TH_{min} = \min_{a \in A} TH^a(M) \tag{5.23}$$

The system is stable under any resource allocation if $\bar{\lambda} \leq TH_{min}$. If it is possible to choose the resource allocation upfront, then a resource allocation can be chosen such that the system is stable if $\bar{\lambda} \leq TH_{max}$. It is therefore possible to calculate for any fixed policy f^F, whether the system will be stable. Moreover, we can draw further conclusions with regards to existence of a demand-dependent policy that results in a stable system, as shown in Theorem 2.

Theorem 2. *If a fixed policy f^F exists, such that $\bar{\lambda} \leq TH_{max}$, meaning that the system is stable, then there also exists a demand-dependent policy f^{DD} for which the system is stable.*

Proof. For a fixed policy f^F, a^F is the allocation used and $TH^{a^F}(M)$ is the order throughput capacity of the system under f^F. Let f^F be such that $TH^{a^F}(M) = TH_{max}$ and let the system be stable under f^F, i.e. $\bar{\lambda} \leq TH_{max}$. For a demand-dependent policy f^{DD}, let the order throughput capacity be denoted with $TH^{(f^{DD})}$. If we choose the demand-dependent policy f^{DD} to be such that $a_h^{DD} = a^F$ and $a_l^{DD} = a^F$, then $TH^{(f^{DD})} = TH^{a^F}(M)$ as the same action would be chosen in each state for both f^{DD} and f^F. We can therefore choose a demand-dependent policy f^{DD} such that $TH^{(f^{DD})} = TH_{max}$, and since $\bar{\lambda} \leq TH_{max}$ the system would be stable under f^{DD}. □

5.6 Validation & Numerical Experiments

This section describes the numerical experiments. Subsection 5.6.1 describes the first numerical experiment, called the "*small instance experiment*", where the optimal policy is compared with the benchmark policies on a set of small instances. An explanation of the policies can be found in Section 5.4.3. The aim of this experiment is to be able to compare the benchmark policies with an optimal policy to obtain an impression of their performance. The optimal policy can only be determined if the number of states is small, which limits on the number of pods M and the order cap ω. Both M and ω are small as a result.
Subsection 5.6.2 describes the second numerical experiment, a comparison of the benchmark policies on a set of large instances. This experiment is called the "*large*

instance experiment", because the size of the systems in terms of number of pods, number of workstations, and size of the layouts, are sufficiently large to represent real systems. For these large instances, the optimal policy could not be determined, but nevertheless the experiment leads to some new insights due to the benchmark policies.

Throughout all experiments, the same layouts, parameters, and resource allocations are used. Three layouts are used in total, a small, a medium, and a large layout. The layouts are described in Table 5.3 and depicted in Figures 5.8 and 5.9. In Figure 5.8, the numbers next to the workstations indicate the order of activation. For example, if there are two replenishment stations in our MDP model, that means that on the left side of Figure 5.8 the workstations with a 1 and 2 next to them will be used as replenishment stations, while with three replenishment stations, the replenishment station with a 3 next to it would also be used as a replenishment station.

In Table 5.3, μ_{ss}^{-1}, μ_{sp}^{-1}, μ_{ps}^{-1}, μ_{sr}^{-1}, and μ_{rs}^{-1} are the average travel times, as described in Section 5.3.2. The corresponding travel time variances are denoted by $\hat{V}_{ss}$, $\hat{V}_{sp}$, $\hat{V}_{ps}$, $\hat{V}_{sr}$, and $\hat{V}_{rs}$. We have that $\mu_{sp}^{-1} = \mu_{ps}^{-1} = \mu_{sr}^{-1} = \mu_{rs}^{-1}$ and that $\hat{V}_{sp} = \hat{V}_{ps} = \hat{V}_{sr} = \hat{V}_{rs}$, due to the symmetry of the layouts, e.g. the placement of the pick stations around the storage area mirrors the placements of the replenishment stations around the storage area in all layouts. The order arrival rates differ across layouts and across experiments and are shown in Table 5.4.

Table 5.3 Layouts variables (with block size: 2x4)

Name	M	W	Aisles	×	Cross-aisles	μ_{ss}^{-1}	$\hat{V}_{ss}$	$\mu_{sp}^{-1} = \mu_{ps}^{-1} = \mu_{sr}^{-1} = \mu_{rs}^{-1}$	$\hat{V}_{sp} = \hat{V}_{ps} = \hat{V}_{sr} = \hat{V}_{rs}$
Small	550	4	8	×	8	18.4 s	90.0 s^2	34.5 s	118.8 s^2
Medium	1149	6	12	×	12	26.6 s	188.3 s^2	45.3 s	250.0 s^2
Large	1965	8	16	×	16	34.8 s	322.5 s^2	56.1 s	429.2 s^2

Table 5.4 shows the parameters that are used throughout the numerical experiments. It includes the parameter needed for calculating the empirical travel time distributions, i.e. to calculate μ_{ss}^{-1}, μ_{sp}^{-1}, μ_{ps}^{-1}, μ_{sr}^{-1}, μ_{rs}^{-1}, $\hat{V}_{ss}$, $\hat{V}_{sp}$, $\hat{V}_{ps}$, $\hat{V}_{sr}$, and $\hat{V}_{rs}$. These parameters are the robot acceleration, robot deceleration, the average robot speed, the maximum robot speed, the time needed to lift and to store a pod, and the time a robot needs to make a 90 degree turn. We estimated these parameters by observing real implementations of RMFSs.

In addition, Table 5.4 also includes the other parameters needed for the queueing networks, namely the probability q that a pod needs to go for replenishment after visiting a pick station, the average pick time (μ_p^{-1}) and the average replenishment time (μ_r^{-1}) needed for the complete queueing network. The q is exogenous to the

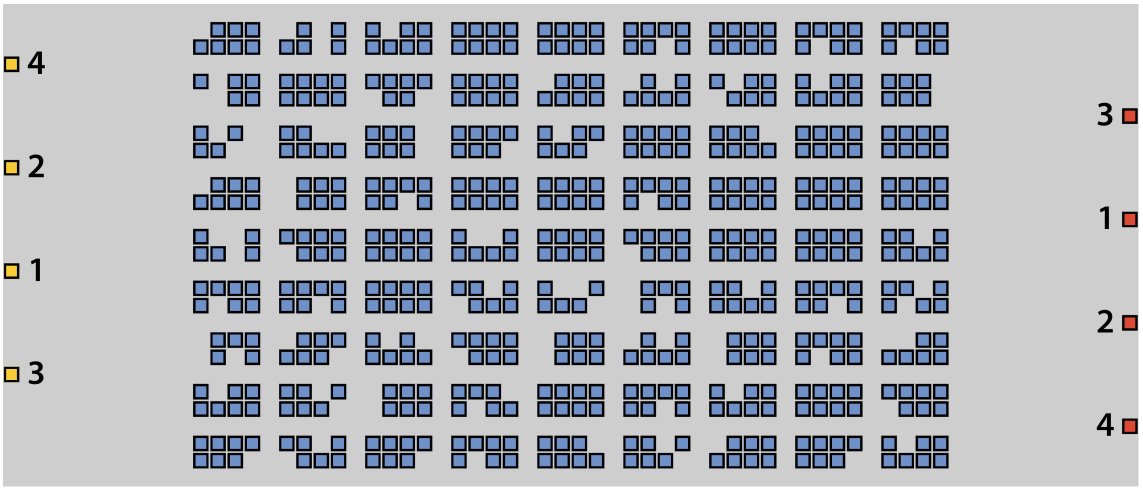


Figure 5.8 Top view of the small layout, including station activation sequence numbering, with the storage area in the middle, replenishment stations to the left, and pick stations to the right

(a) Medium layout **(b)** Large layout

Figure 5.9 Top view of the medium and large layout, with the storage area in the middle, replenishment stations to the left, and pick stations to the right

model we present, and can be observed in existing implementations. It is, however, not the probability to directly go to a replenishment station after picking. A pod that has entered the replenishment process, shown in Figure 5.3, may spend a considerable amount of time in store before it is transported to a replenishment station by a robot. In other words, q is the probability that a pod will need replenishment, or can be needed to replenish a product running low on inventory. We estimated these parameters from observations at an RMFS at a Dutch retailer and found that $q = 20\%$.

The parameters in Table 5.4 that are related to the costs are C_{cwt}, C_{ls}, and ζ. The stock-out probability ζ is derived in Appendix 5.10. For the cost of a lost sale, we estimate that in an E-commerce environment the cost is on average 20 € per order. The customer waiting in an E-commerce environment stems from the idea that if a customer has to wait too long, the customer may cancel the order and buy the product(s) elsewhere, either online or in a retail store, or may be less inclined to place a future order with the company. Assuming that a waiting time of 24 hours is equivalent to a lost sale, the cost of customer waiting time is roughly 0.00023 € per order per second.

The results for both the large instance experiment and the small instance experiment are generated by running a discrete event simulation of the corresponding Markov Decision Process. The state in this simulation is s and the transition to another state occurs as detailed in Table 5.2. Table 5.4 show the simulation time per simulation run and the number of simulation runs. The simulation time is 31536000 seconds, which equals one year and is sufficiently long for the system to have reached a long-run equilibrium.

Table 5.4 also contains parameters that differ between the large instance experiment and the small instance experiment, namely the order cap ω and the average arrival rate $\bar{\lambda}$. As mentioned earlier, in the small instance experiment, the order cap ω needs to be kept small. However, in real systems, an order is not rejected due to some order cap, but is instead temporarily put in a backlog until the system can process it. Therefore, in the large instance experiment the order cap should have a minimal influence, and consequently is set to a large value.

The order arrival rates differ across the layouts, because the travel times differ strongly across the layouts and this affects the order throughput capacity of the system. The order arrival rates also differ between the two experiments, because in the small instance experiment the number of pods M is small and this reduces the order throughput capacity. The order arrival rates were chosen such that the number of picking completion events (transitions (B) and (C)) where (nearly) equal to the number of replenishment completion events (transition (D)) under the state-dependent policy.

Lastly, Table 5.5 shows the resource allocations, labeled a_1 to a_{11}, and how these

resource allocations translate to number of pick stations and pick robots for each layout. The ratios and fractions used in these resource allocations are based on our observations at an RMFS implementation at a Dutch retailer.

Table 5.4 Parameters used throughout the experiments

Parameter	Value	Parameter	Value
Pod storage time	3.0 s	Pod lift time	3.0 s
Avg. robot speed	1.3 $\frac{m}{s}$	Max. robot speed	1.5 $\frac{m}{s}$
Robot acceleration	0.5 $\frac{m^2}{s}$	Robot deceleration	0.5 $\frac{m^2}{s}$
Robot turn time	2.5 s	Prob. repl. after picking (q)	20.0%
Avg. pick time (μ_p^{-1})	10.0 s	Avg. repl. time (μ_r^{-1})	30.0 s
Number of simulation runs	10 runs	Simulation time	31536000 s
Cost of waiting time (C_{cwt})	0.00023 €/order/s	Cost per lost sale (C_{ls})	20.0 €/order
Stock-out probability parameter ζ	7		
Small instance experiment		Large instance experiment	
ω	20 orders	ω	2000000 orders
$\bar{\lambda}$, small layout	360.0 orders/h	$\bar{\lambda}$, small layout	468.0 orders/h
$\bar{\lambda}$, medium layout	432.0 orders/h	$\bar{\lambda}$, medium layout	540.0 orders/h
$\bar{\lambda}$, large layout	504.0 orders/h	$\bar{\lambda}$, large layout	612.0 orders/h

Table 5.5 Resource allocations a per layout, $a = (W_p, R_p)$

		Small Layout ($W = 4$)			Medium Layout ($W = 6$)			Large Layout ($W = 8$)		
		$R = 16$	$R = 24$	$R = 32$	$R = 24$	$R = 36$	$R = 48$	$R = 32$	$R = 48$	$R = 64$
$a_1 =$	$(W_p = 0, R_p = 0)$	(0, 0)	(0, 0)	(0, 0)	(0, 0)	(0, 0)	(0, 0)	(0, 0)	(0, 0)	(0, 0)
$a_2 =$	$(W_p = \lfloor 0.45W \rfloor, R_p = \lfloor 0.45R \rfloor)$	(1, 7)	(1, 10)	(1, 14)	(2, 10)	(2, 16)	(2, 21)	(3, 14)	(3, 21)	(3, 28)
$a_3 =$	$(W_p = \lfloor 0.45W \rfloor, R_p = \lfloor 0.60R \rfloor)$	(1, 9)	(1, 14)	(1, 19)	(2, 14)	(2, 21)	(2, 28)	(3, 19)	(3, 28)	(3, 38)
$a_4 =$	$(W_p = \lfloor 0.45W \rfloor, R_p = \lfloor 0.75R \rfloor)$	(1, 12)	(1, 18)	(1, 24)	(2, 18)	(2, 27)	(2, 36)	(3, 24)	(3, 36)	(3, 48)
$a_5 =$	$(W_p = \lfloor 0.60W \rfloor, R_p = \lfloor 0.45R \rfloor)$	(2, 7)	(2, 10)	(2, 14)	(3, 10)	(3, 16)	(3, 21)	(4, 14)	(4, 21)	(4, 28)
$a_6 =$	$(W_p = \lfloor 0.60W \rfloor, R_p = \lfloor 0.60R \rfloor)$	(2, 9)	(2, 14)	(2, 19)	(3, 14)	(3, 21)	(3, 28)	(4, 19)	(4, 28)	(4, 38)
$a_7 =$	$(W_p = \lfloor 0.60W \rfloor, R_p = \lfloor 0.75R \rfloor)$	(2, 12)	(2, 18)	(2, 24)	(3, 18)	(3, 27)	(3, 36)	(4, 24)	(4, 36)	(4, 48)
$a_8 =$	$(W_p = \lfloor 0.75W \rfloor, R_p = \lfloor 0.45R \rfloor)$	(3, 7)	(3, 10)	(3, 14)	(4, 10)	(4, 16)	(4, 21)	(6, 14)	(6, 21)	(6, 28)
$a_9 =$	$(W_p = \lfloor 0.75W \rfloor, R_p = \lfloor 0.60R \rfloor)$	(3, 9)	(3, 14)	(3, 19)	(4, 14)	(4, 21)	(4, 28)	(6, 19)	(6, 28)	(6, 38)
$a_{10} =$	$(W_p = \lfloor 0.75W \rfloor, R_p = \lfloor 0.75R \rfloor)$	(3, 12)	(3, 18)	(3, 24)	(4, 18)	(4, 27)	(4, 36)	(6, 24)	(6, 36)	(6, 48)
$a_{11} =$	$(W_p = W, R_p = R)$	(4, 16)	(4, 24)	(4, 32)	(6, 24)	(6, 36)	(6, 48)	(8, 32)	(8, 48)	(8, 64)

5.6.1 Small Instance Experiment Results

Table 5.7 shows a comparison of the optimal policy with the four benchmark policies for small instances, i.e. instances with low M and small ω. The M and ω are low to keep the number of states in the MDP sufficiently small that it can be solved to optimality. The optimal policy was determined with the open source MDP toolbox from INRA, which is described in [12].

For each policy f in Table 5.6, results were generated via a simulation of the Markov Decision Process, where allocations were chosen according to the policy f. For each

policy f, 10 simulation runs were performed and the average costs C_A^f are an average across these 10 simulation runs. For each of the five policy categories, namely F, DD, CD, SD, and O (Optimal), we examine the average costs of the policies in that category and report the lowest cost among these policies. For example, Table 5.6 shows five fixed policies, namely F_1 to F_5, where the expected costs per second are $C_A^{F_1}, \ldots, C_A^{F_5}$ respectively. We denote the minimal cost in a policy category X as $C_A^{X^*}$. For the F, DD, and CD policy categories, the minimal cost is given in Equations (5.24), (5.25), and (5.26), respectively. The SD and O categories only have one policy each, so the minimal cost in that category equals the average cost per second C_A of the one policy in that category.

$$F^* = \arg\min \left(C_A^{F_1}, \ldots, C_A^{F_5}\right) \tag{5.24}$$
$$DD^* = \arg\min \left(C_A^{DD_1}, \ldots, C_A^{DD_9}\right) \tag{5.25}$$
$$CD^* = \arg\min \left(C_A^{CD_1}, \ldots, C_A^{CD_9}\right) \tag{5.26}$$

The different settings in Table 5.7 include the three layouts (with corresponding number of robots) and three MMPP scenarios, creating nine settings in total. In the first MMPP scenario, the low demand phase lasts on average 4 times as long as the high demand phase, while during the high demand phase 4 times as many orders arrive compared to the low demand phase. It is a MMPP scenario where the differences between the high and the low phase are moderate. In the second MMPP scenario, the ratio is further skewed to 1:7. This MMPP scenario aims to represent a normal workday, where customers have three hours of spare time in the evening during which they shop online. In contrast, the rest of the day is the low demand phase. The third MMPP scenario does not aim to represent one day, but rather one week. The high demand phase represents the weekend, when customers have more time to pursue online shopping, whereas the low demand phase represents the work week.

The results in Table 5.7 show the 95% confidence intervals across the 10 simulation runs of the best policies per category. The confidence intervals are the smallest for MMPP scenario 1, and the largest for MMPP scenario 3. The best cost-dependent policy leads to higher costs on average than the best policies from the other categories. The higher costs are caused by the small value for order cap ω; the ω should be set to a high number as discussed earlier, and as is done for the large instance experiment, but is set to a small value here to limit the number of states. The small value for ω has a disproportionate effect on the picking costs that the cost-dependent policy uses, which means it will favor allocating resources to the picking process over replenishment process. However, with just $M = 30$ pods, replenishment costs due to stock-out will consequently become larger. In other words, the small value for ω

distorts the cost-dependent policies.
Also, for the small layout, the optimal policy does seem to clearly outperform the best policies from the other categories. However, for the other layouts, the confidence intervals of the optimal policy and the best F, DD, and SD policies mostly overlap. Figure 5.10 provides an overview of the average cost across all policies. There is a group of policies that lead to low costs and another group that leads to high costs, with a large gap in between the two groups. However, across MMPP scenarios the performance does not vary much, indicating that the policies are robust against time-varying demand.
The main observation from the results in Table 5.7 is that confidence intervals of the benchmark policies mostly overlap with the confidence intervals from the optimal policies. In other words, the benchmark policies perform similarly to the optimal policy. We can therefore focus on the benchmark policies for real-life larger instances.

Table 5.6 Overview of policies used in the small instance experiment. The SD policy is given in Equation 5.17, and allocations $a_1, \ldots, a_{11}$ can be found in Table 5.5.

Policy	Allocation	Policy	Allocation	Policy	Allocation
F_1	$a^F = a_1$	DD_1	$a_l^{DD} = a_1$, $a_h^{DD} = a_6$	CD_1	$a_p^{CD} = a_1$, $a_r^{CD} = a_6$
F_2	$a^F = a_2$	DD_2	$a_l^{DD} = a_1$, $a_h^{DD} = a_{10}$	CD_2	$a_p^{CD} = a_1$, $a_r^{CD} = a_{10}$
F_3	$a^F = a_6$	DD_3	$a_l^{DD} = a_1$, $a_h^{DD} = a_{11}$	CD_3	$a_p^{CD} = a_1$, $a_r^{CD} = a_{11}$
F_4	$a^F = a_{10}$	DD_4	$a_l^{DD} = a_2$, $a_h^{DD} = a_6$	CD_4	$a_p^{CD} = a_2$, $a_r^{CD} = a_6$
F_5	$a^F = a_{11}$	DD_5	$a_l^{DD} = a_2$, $a_h^{DD} = a_{10}$	CD_5	$a_p^{CD} = a_2$, $a_r^{CD} = a_{10}$
		DD_6	$a_l^{DD} = a_2$, $a_h^{DD} = a_{11}$	CD_6	$a_p^{CD} = a_2$, $a_r^{CD} = a_{11}$
		DD_7	$a_l^{DD} = a_6$, $a_h^{DD} = a_6$	CD_7	$a_p^{CD} = a_6$, $a_r^{CD} = a_6$
		DD_8	$a_l^{DD} = a_6$, $a_h^{DD} = a_{10}$	CD_8	$a_p^{CD} = a_6$, $a_r^{CD} = a_{10}$
		DD_9	$a_l^{DD} = a_6$, $a_h^{DD} = a_{11}$	CD_9	$a_p^{CD} = a_6$, $a_r^{CD} = a_{11}$

Table 5.7 Results for the small instance experiment: The 95% confidence intervals of the costs for the benchmark policies and optimal policy (costs in € per second), with $M = 30$, and $\omega = 20$, # states = 2232

Layout	R	MMPP	ν_h^{-1}	ν_l^{-1}	λ_h:λ_l	$C_A^{F^*}$	$C_A^{DD^*}$	$C_A^{SD^*}$	$C_A^{CD^*}$	$C_A^{O^*}$
Small	16	1	3 h	12 h	1:4	[0.50, 0.52]	[0.50, 0.52]	[0.50, 0.52]	[0.61, 0.64]	[0.45, 0.47]
Medium	24	1	3 h	12 h	1:4	[0.61, 0.64]	[0.61, 0.65]	[0.58, 0.60]	[0.72, 0.76]	[0.56, 0.60]
Large	32	1	3 h	12 h	1:4	[0.78, 0.82]	[0.76, 0.79]	[0.79, 0.81]	[0.88, 0.91]	[0.77, 0.79]
Small	16	2	3 h	21 h	1:7	[0.67, 0.73]	[0.64, 0.69]	[0.67, 0.72]	[0.70, 0.74]	[0.63, 0.66]
Medium	24	2	3 h	21 h	1:7	[0.80, 0.87]	[0.80, 0.87]	[0.79, 0.85]	[0.90, 0.94]	[0.79, 0.86]
Large	32	2	3 h	21 h	1:7	[1.02, 1.09]	[1.02, 1.08]	[0.98, 1.05]	[1.08, 1.18]	[1.00, 1.09]
Small	16	3	36 h	132 h	3:11	[0.47, 0.54]	[0.49, 0.55]	[0.44, 0.50]	[0.57, 0.72]	[0.41, 0.47]
Medium	24	3	36 h	132 h	3:11	[0.54, 0.61]	[0.53, 0.65]	[0.53, 0.58]	[0.70, 0.80]	[0.51, 0.56]
Large	32	3	36 h	132 h	3:11	[0.69, 0.84]	[0.69, 0.81]	[0.72, 0.87]	[0.81, 0.96]	[0.71, 0.86]

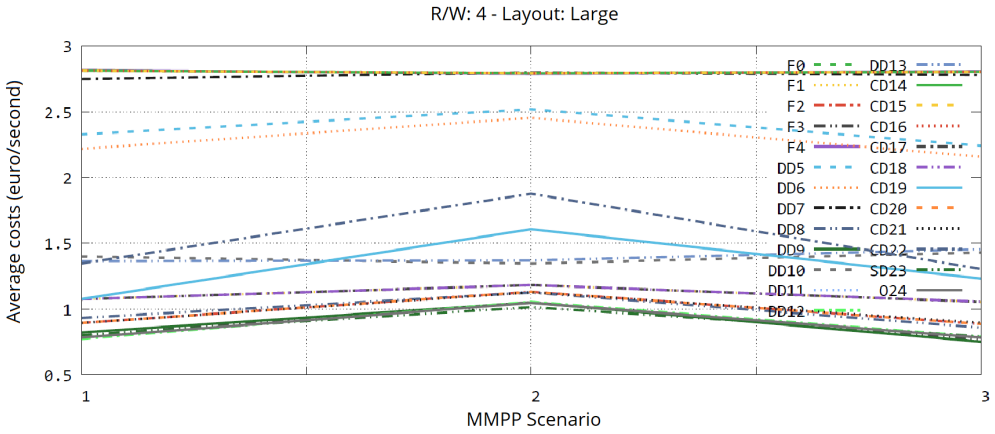


Figure 5.10 Average costs for policies in the small instance experiment across MMPP Scenarios

5.6.2 Large Instance Experiment Results

Table 5.8 shows the results for the four benchmark policies on large instances. For the sake of brevity, no confidence intervals are shown. The number of states for these instances is so large that solving the MDP optimally is not computationally feasible. The policies used are described in Table 5.9. The policies vary widely in their costs and can become as large as 350 € per second, which indicates an unsustainable, large order backlog. Under such policies, the system is clearly unstable.
The best policies in each policy category, namely F^*, DD^*, CD^*, and SD^* and the associated minimal costs for each policy category, namely $C_A^{F^*}$, $C_A^{DD^*}$, $C_A^{CD^*}$, and $C_A^{SD^*}$, are calculated as described in Section 5.6.1 for the small instance experiment.

Besides a larger ω and a higher number of pods M in each layout, namely the values as shown in Table 5.3, the large instance experiment contains more MMPP scenarios than the small instance experiment. Also, the number of robots per workstation, denoted by R/W, is varied. MMPP scenarios 8, 9, and 10 are meant to represent a typical week, with the weekend represented by the high demand phase as customers shop more in the weekend, and the work week represented by the low demand phase. First of all, we can see that, for the same layout and policy category, the average costs differ widely across the MMPP scenarios. This cost difference shows that not only the average order arrival rate matters, but also the length of a period of peak demand and the height of the peak affect costs. For different MMPP scenarios, different policies and types of policies lead to the lowest cost. Secondly, we can see that with four robots per workstation the costs are much higher, indicating that when only four robots are present, orders have to wait much longer to be fulfilled. However, for the best CD policy, the costs are typically only a fraction of the costs under the F and DD policies, showing that resource reallocation can reduce costs sharply when the number of robots is limited. In other words, even when the number of robots is sufficient to process all orders on average, the number of robots and the policy employed can have a strong, non-linear impact on the customer waiting time. Lastly, in all cases either the state-dependent policy or a cost-dependent policy deliver the lowest average costs. The state-dependent policy and the cost-dependent policies exploit the fact that they can dynamically reallocate resources, whereas the fixed policies and the demand-dependent policies do not. Therefore, dynamic resource reallocation can contribute to lowering costs in an RMFS. Moreover, the fixed policy and the demand-dependent policy can be applied in other contexts, but the state-dependent policy and cost-dependent policy are specific for the RMFS. An RMFS may therefore provide value to E-commerce companies by lowering costs when deploying the state-dependent or a cost-dependent policy.

Table 5.8 Results for the large instance experiment (costs in € per second)

					Small Layout				Medium Layout				Large Layout			
MMPP	ν_h^{-1}	ν_l^{-1}	λ_h:λ_l	R/W	$C_A^{F^*}$	$C_A^{DD^*}$	$C_A^{SD^*}$	$C_A^{CD^*}$	$C_A^{F^*}$	$C_A^{DD^*}$	$C_A^{SD^*}$	$C_A^{CD^*}$	$C_A^{F^*}$	$C_A^{DD^*}$	$C_A^{SD^*}$	$C_A^{CD^*}$
1	3 h	3 h	1:1	4	26.35	22.80	0.03	0.05	21.92	0.86	0.01	0.01	47.28	2.18	0.05	0.08
1	3 h	3 h	1:1	6	< 0.01	0.12	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01
1	3 h	3 h	1:1	8	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01
2	3 h	6 h	1:2	4	25.56	52.62	2.06	0.33	23.17	23.98	0.72	0.29	50.30	54.26	3.51	0.41
2	3 h	6 h	1:2	6	0.02	0.47	< 0.01	0.02	0.02	0.02	< 0.01	0.01	0.03	0.04	< 0.01	0.02
2	3 h	6 h	1:2	8	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01
3	3 h	9 h	1:3	4	28.67	68.35	4.09	0.68	29.35	47.64	1.87	0.52	49.54	82.73	5.85	0.68
3	3 h	9 h	1:3	6	0.15	0.59	0.09	0.13	0.13	0.25	0.06	0.11	0.19	0.22	0.07	0.13
3	3 h	9 h	1:3	8	0.09	0.07	0.03	0.04	0.02	0.02	< 0.01	0.02	0.03	0.03	< 0.01	0.02
4	3 h	12 h	1:4	4	31.39	77.75	7.51	5.10	32.94	62.88	3.45	4.81	54.69	98.75	9.35	2.57
4	3 h	12 h	1:4	6	0.30	0.63	0.22	0.27	0.29	0.42	0.16	0.23	0.40	0.34	0.20	0.27
4	3 h	12 h	1:4	8	0.20	0.18	0.10	0.12	0.08	0.13	0.04	0.08	0.11	0.15	0.05	0.10
5	3 h	15 h	1:5	4	33.59	86.01	7.49	6.76	31.49	73.78	6.53	7.15	56.39	> 100	11.64	7.36
5	3 h	15 h	1:5	6	0.42	0.73	0.39	0.39	0.47	0.49	0.31	0.47	0.64	0.48	0.39	0.64
5	3 h	15 h	1:5	8	0.33	0.36	0.17	0.22	0.16	0.29	0.11	0.16	0.20	0.31	0.12	0.18
6	3 h	18 h	1:6	4	29.81	93.24	7.70	8.11	32.13	79.84	6.38	8.69	48.69	> 100	13.89	9.66
6	3 h	18 h	1:6	6	0.56	0.92	0.55	0.52	0.71	0.59	0.49	0.71	0.93	0.72	0.53	0.93
6	3 h	18 h	1:6	8	0.40	0.60	0.28	0.29	0.26	0.53	0.19	0.26	0.31	0.39	0.21	0.31
7	3 h	21 h	1:7	4	34.78	96.49	11.28	9.47	33.26	82.01	9.54	10.50	49.63	> 100	12.89	11.49
7	3 h	21 h	1:7	6	0.61	1.15	0.75	0.60	0.98	0.75	0.61	0.98	1.04	1.05	0.70	1.04
7	3 h	21 h	1:7	8	0.48	0.77	0.40	0.38	0.35	0.57	0.29	0.35	0.44	0.45	0.27	0.44
8	36 h	132 h	3:11	4	36.50	20.10	37.13	5.24	52.89	1.58	21.18	5.89	64.04	3.16	24.04	6.66
8	36 h	132 h	3:11	6	0.75	1.23	2.18	0.88	2.60	1.40	2.06	2.60	3.73	1.46	2.67	3.73
8	36 h	132 h	3:11	8	0.51	0.53	1.07	0.48	0.65	0.96	0.57	0.33	0.88	1.07	0.50	0.35
9	48 h	120 h	2:5	4	34.42	32.89	18.63	3.13	27.00	1.42	14.89	1.06	69.79	2.86	26.84	1.39
9	48 h	120 h	2:5	6	0.38	0.46	0.79	0.38	0.98	1.33	0.57	0.29	1.33	1.34	0.71	0.32
9	48 h	120 h	2:5	8	0.28	0.28	0.06	0.06	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01
10	60 h	107 h	5:9	4	28.68	47.88	11.61	0.59	34.47	12.32	10.00	0.66	46.36	18.81	19.71	0.81
10	60 h	107 h	5:9	6	0.11	0.13	< 0.01	0.01	< 0.01	< 0.01	< 0.01	< 0.01	0.09	0.09	< 0.01	0.01
10	60 h	107 h	5:9	8	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01

Table 5.9 Overview of policies used in the large instance experiment. The SD policy is given in Equation 5.17, and allocations $a_1, \ldots, a_{11}$ can be found in Table 5.5.

Policy	Allocation	Policy	Allocation	Policy	Allocation
F_1	$a^F = a_2$	DD_1	$a_l^{DD} = a_1,\ a_h^{DD} = a_7$	CD_1	$a_p^{CD} = a_7,\ a_r^{CD} = a_1$
F_2	$a^F = a_3$	DD_2	$a_l^{DD} = a_1,\ a_h^{DD} = a_9$	CD_2	$a_p^{CD} = a_7,\ a_r^{CD} = a_2$
F_3	$a^F = a_4$	DD_3	$a_l^{DD} = a_1,\ a_h^{DD} = a_{10}$	CD_3	$a_p^{CD} = a_7,\ a_r^{CD} = a_3$
F_4	$a^F = a_5$	DD_4	$a_l^{DD} = a_1,\ a_h^{DD} = a_{11}$	CD_4	$a_p^{CD} = a_7,\ a_r^{CD} = a_5$
F_5	$a^F = a_6$	DD_5	$a_l^{DD} = a_2,\ a_h^{DD} = a_7$	CD_5	$a_p^{CD} = a_9,\ a_r^{CD} = a_1$
F_6	$a^F = a_7$	DD_6	$a_l^{DD} = a_2,\ a_h^{DD} = a_9$	CD_6	$a_p^{CD} = a_9,\ a_r^{CD} = a_2$
F_7	$a^F = a_8$	DD_7	$a_l^{DD} = a_2,\ a_h^{DD} = a_{10}$	CD_7	$a_p^{CD} = a_9,\ a_r^{CD} = a_3$
F_8	$a^F = a_9$	DD_8	$a_l^{DD} = a_2,\ a_h^{DD} = a_{11}$	CD_8	$a_p^{CD} = a_9,\ a_r^{CD} = a_5$
F_9	$a^F = a_{10}$	DD_9	$a_l^{DD} = a_3,\ a_h^{DD} = a_7$	CD_9	$a_p^{CD} = a_{10},\ a_r^{CD} = a_1$
		DD_{10}	$a_l^{DD} = a_3,\ a_h^{DD} = a_9$	CD_{10}	$a_p^{CD} = a_{10},\ a_r^{CD} = a_2$
		DD_{11}	$a_l^{DD} = a_3,\ a_h^{DD} = a_{10}$	CD_{11}	$a_p^{CD} = a_{10},\ a_r^{CD} = a_3$
		DD_{12}	$a_l^{DD} = a_3,\ a_h^{DD} = a_{11}$	CD_{12}	$a_p^{CD} = a_{10},\ a_r^{CD} = a_5$
		DD_{13}	$a_l^{DD} = a_5,\ a_h^{DD} = a_7$	CD_{13}	$a_p^{CD} = a_{11},\ a_r^{CD} = a_1$
		DD_{14}	$a_l^{DD} = a_5,\ a_h^{DD} = a_9$	CD_{14}	$a_p^{CD} = a_{11},\ a_r^{CD} = a_2$
		DD_{15}	$a_l^{DD} = a_5,\ a_h^{DD} = a_{10}$	CD_{15}	$a_p^{CD} = a_{11},\ a_r^{CD} = a_3$
		DD_{16}	$a_l^{DD} = a_5,\ a_h^{DD} = a_{11}$	CD_{16}	$a_p^{CD} = a_{11},\ a_r^{CD} = a_5$

5.7 Conclusions

Warehouses have to balance resources between order picking and inventory replenishment tasks. This is particularly important in online retail environments, which suffer from high demand peaks and where customers require very short response times. Too many resources in replenishment may lead to delays in fulfilment. However, too many resources allocated to picking, may deplete inventory and ultimately lead to even larger delays in fulfilment. We model this problem for robotic mobile fulfillment systems, which are popular in online retail warehouses. In such a system two different resources, robots and workers, work together to fill orders and replenish inventory. Resources can rapidly switch between tasks (with no or little setup), based on demand and inventory levels. We build an MDP model embedded in a queuing network model that allows to take optimal decisions, for small instances, minimizing operational, customer wait, and lost sales cost. For larger, real-life size instances we find heuristic allocation policies that are close to optimal for small instances. Four heuristic policies are evaluated. The fixed policy keeps the resource allocation fixed, whereas the demand-dependent policy has two different resource allocations, one for high demand and one for low demand. The cost-dependent policy also has two different resource allocations, depending on the ratio of picking to replenishment costs. Finally, the state-dependent policy uses the transition probabilities of the MDP model to estimate the expected cost of an allocation and chooses the one with lowest cost. The cost-dependent and state-dependent policies outperform the fixed and the demand-dependent policies. Continually reallocating resources based on the state of the system (order demand rate, allocation, and number of waiting orders) appears to bring substantial cost savings. The benefits are even more pronounced when the number of robots is small, because then the cost-dependent and/or state-dependent policies can sharply reduce costs compared to the fixed and demand-dependent policies. We also show that the characteristics of peak demand have a strong effect on the costs. Given a fixed average order arrival rate across all demand phases, we varied the length of the peak demand phase and the height of the peak, and found that this influences both the average costs but also the type of policy that minimizes costs. To minimize costs, a system should deploy different resource allocation policies depending on the duration and height of peak demand. The method developed in this paper to cope with non-stationary demand and dynamic reallocation of resources may be deployed rapidly to other handling systems. Automated systems, such as automated-guided vehicle systems and robotic systems may be the most suitable, as these resources can continually switch between different processes without setup cost, based on software control. For robotic mobile fulfillment systems, an interesting area of future research would be to include the repositioning process in the method. Robots may reposition pods stored within the storage area in order to sort the inventory. Repositioning may

reduce the time needed to retrieve pods, and hence reduce the customer waiting time, but repositioning tasks also add to the workload of the robots.

References

[1] N. A. H. Agatz, M. Fleischmann, and J. A. E. E. van Nunen. "E-fulfillment and multi-channel distribution: a review". In: *European Journal of Operational Research* 187 (2008), pp. 339–356.

[2] T. W. Archibald. "Modelling Replenishment and Transshipment Decisions in Periodic Review Multilocation Inventory Systems". In: *The Journal of the Operational Research Society* 58 (2007), pp. 948–956.

[3] K. Azadeh, M. B. M. de Koster, and D. Roy. "Robotized Warehouse Systems: Developments and Research Opportunities". Available at SSRN: https://ssrn.com/abstract=2977779. 2017.

[4] S. Bhat and A. Krishnamurthy. "Flexible Policies for Multi-Class Systems with Seasonal Demands". In: *Proceedings of the 2013 Industrial and Systems Engineering Research Conference*. 2013, pp. 2846–2854.

[5] S. Bhat and A. Krishnamurthy. "Interactive effects of seasonal-demand characteristics on manufacturing systems". In: *International Journal of Production Research* 54.10 (2016), pp. 2951–2964.

[6] S. Bhat and A. Krishnamurthy. "Production control policies to maintain service levels in different seasons". In: *Journal of Manufacturing Systems* 41 (2016), pp. 31–44.

[7] S. Bhat and A. Krishnamurthy. "Production rate strategies for manufacturing systems with seasonal demands". In: *Proceedings of the 2012 Industrial and Systems Engineering Research Conference*. 2012.

[8] S. Bhat and A. Krishnamurthy. "Value of capacity flexibility in manufacturing systems with seasonal demands". In: *IIE Transactions* 47 (2015), pp. 693–714.

[9] G. Bolch et al. *Queueing Networks and Markov Chains*. Wiley Publishing Inc., 2006.

[10] Nils Boysen, Dirk Briskorn, and Simon Emde. "Parts-to-picker based order processing in a rack-moving mobile robots environment". In: *European Journal of Operational Research* 262.2 (2017), pp. 550–562.

[11] R. Buitenhek, G.-J. Van Houtum, and H. Zijm. "AMVA-based solution procedures for open queueing networks with population constraints". In: *Annals of Operations Research* 93 (2000), pp. 15–40.

[12] Iadine Chadès et al. "MDPtoolbox: a multi-platform toolbox to solve stochastic dynamic programming problems". In: *Ecography* 37.9 (2014), pp. 916–920.

[13] W. K. Ching, R. H. Chan, and X. Y. Zhou. "Circulant preconditioners for Markov-modulated Poisson processes and their applications to manufacturing systems". In: *Siam Journal of Matrix Analytical Applications* 18.2 (1997), pp. 464–481.

[14] T. B. Crabil. "Optimal Control of a Maintenance System with Variable Service Rates". In: *Operations Research* 22.4 (1974), pp. 736–745.

[15] Vibhuti Dhingra et al. "Solving semi-open queuing networks with time-varying arrivals: An application in container terminal landside operations". In: *European Journal of Operational Research* (2017).

[16] J. J. Enright and P. R. Wurman. "Optimization and Coordinated Autonomy in Mobile Fulfillment Systems". In: *Working paper* (2011).

[17] W. Fischer and K. Meier-Hellstern. "The Markov-modulated Poisson process (MMPP) cookbook". In: *Performance Evaluation* 18 (1992), pp. 149–171.

[18] K. Gue et al. *Material Handling & Logistics: US Roadmap*. http://www.mhlroadmap.org/roadmap.html. [Accessed January 10th, 2017]. 2014.

[19] M. H. van Hoorn and L. P. Seelen. "The SPP/G/1 queue: a single server queue with a switched poisson process as input process". In: *O.R. Spektrum* 5 (1983), pp. 207–218.

[20] J. Jia. "Solving Semi-Open Queuing Networks". PhD thesis. Rensselaer Polytechnic Institute, 2005.

[21] T. Lamballais, D. Roy, and M. B. M. de Koster. "Estimating Performance in a Robotic Mobile Fulfillment System". In: *European Journal of Operations Research* 256 (2017), pp. 976–990.

[22] T. Lamballais, D. Roy, and M. B. M. de Koster. "Inventory Allocation in Robotic Mobile Fulfillment Systems". Working Paper, available at SSRN. 2017.

[23] Edward D Lazowska et al. *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., 1984.

[24] X. Li. "Managing dynamic Inventory Systems with Product Returns: A Markov Decision Process". In: *Journal of Optimization Theory and Applications* 157 (2013), pp. 577–592.

[25] M. Merschformann, L. Xie, and D. Erdmann. *Path planning for Robotic Mobile Fulfillment Systems*. Working paper, available at arXiv, https://arxiv.org/abs/1706.09347. 2017.

[26] M. Merschformann et al. “Investigating Decision Mechanisms for Robotic Mobile Fulfillment Systems”. Working paper. 2017.

[27] MHI and Deloitte. *The 2015 MHI Annual Industry Report: Supply chain innovation - Making the impossible possible*. Tech. rep. [Available at https://www.mhi.org/publications/report, Accessed January 10th, 2017]. MHI, 2015.

[28] MHI and Deloitte. *The 2016 MHI Annual Industry Report: Accelerating change: How innovation is driving digital, always-on supply chains*. Tech. rep. [Available at https://www.mhi.org/publications/report, Accessed January 10th, 2017]. MHI, 2016.

[29] I. van Nieuwenhuyse and M. B. M. De Koster. “Evaluating order throughput time in 2-block warehouses with time window batching”. In: *International Journal of Production Economics* 121 (2009), pp. 654–664.

[30] S. Nigam et al. “Analysis of Class-based Storage Strategies for the Mobile Shelf-based Order Pick System”. In: *Progress in Material Handling Research: 2014*. 2014.

[31] H. Patil and B. R. Divekar. “Inventory management challenges for B2C e-commerce retailers”. In: *Procedia Economics and Finance* 11 (2014), pp. 561–571.

[32] N. U. Prabhu and Y. Zhu. “Markov Modulated Queueing Systems”. In: *Queueing Systems* 5 (1989), pp. 215–246.

[33] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Publishing Inc., 1994.

[34] D. Roy. “Semi-open queuing networks: a review of stochastic models, solution methods and new research areas”. In: *International Journal of Production Research* 54.6 (2016), pp. 1735–1752.

[35] A. Seidscher and S. Minner. “A Semi-Markov decision problem for proactive and reactive transshipments between multiple warehouses”. In: *European Journal of Operational Research* 230 (2013), pp. 42–52.

[36] H. Tunc et al. “The cost of using stationary inventory policies when demand is non-stationary”. In: *Omega* 39 (2011), pp. 410–415.

[37] N Viswanadham and Y Narahari. *Performance modeling of automated manufacturing systems*. Prentice Hall Englewood Cliffs, NJ, 1992.

[38] M. Wulfraat. *Is Kiva Systems a Good Fit for Your Distribution Center? An Unbiased Distribution Consultant Evaluation*. http://www.mwpvl.com/html/kiva_systems.html. 2012.

[39] P. R. Wurman, R. D'Andrea, and M. Mountz. "Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses". In: *AI Magazine* 29.1 (2008), pp. 9–19.

[40] Bipan Zou et al. "Assignment rules in robotic mobile fulfilment systems for online retailers". In: *International Journal of Production Research* 55.20 (2017), pp. 6175–6192.

[41] Bipan Zou et al. "Evaluating battery charging and swapping strategies in a robotic mobile fulfillment system". In: *European Journal of Operational Research* 267.2 (2018), pp. 733–753.

Appendix

5.8 AMVA Algorithm

This appendix shows the single class AMVA algorithm used for evaluating a CQN. It is the AMVA algorithm from Appendix A.2 in [11]. The Infinite Servers queueing stations are modeled by setting the number of servers c_s equal to the number of tokens N. The notation is explained in Table 5.10. Visit ratios are calculated as explained in [9].
For example, for the CQN associated with the replenishment process, shown in Figure 5.3, the parameters are: $S = 4$, $N = R_r$, $ES_1 = \mu_{ss}$, $ES_2 = \mu_{sr}$, $ES_3 = \mu_r$, $ES_3^2 = 2\mu_r^2$, $ES_4 = \mu_{rs}$, $c_1 = c_2 = c_4 = N$, $c_3 = W_r$, $v_1 = v_2 = v_3 = v_4 = 1$. The travel times follow empirical distributions, from which the second moments ES_1^2, ES_2^2, and ES_4^2 are calculated. The resulting throughput of the CQN, $\tau(n)$, form the required load-dependent queue service rates, also shown in Figure 5.5, i.e. $\tau(n) = \hat{\mu}_r(n)$ for $n = 1, \ldots, N$.

Table 5.10 Notation used in the AMVA

Symbol	Meaning
S	the total number of queueing stations in the CQN
N	the total number of tokens in the CQN
$ES_{rem,s}$	the expected time remaining until the first departure at station s
ES_s	the first moment of the service time of station s
ES_s^2	the second moment of the service time of station s
$\tilde{L}_s(n)$	the expected queue length excluding tokens in service at station s given that the CQN contains n tokens
$Q_s(n)$	the probability that all servers are busy at station s when the CQN contains n tokenss
$p_s(i \mid n)$	the probability that there are i tokens at station s given that the CQN contains n tokens
$\tau(n)$	the throughput given that the CQN contains n tokens
$ET_s(n)$	the lead time at station s when the CQN contains n tokens
c_s	the number of servers at station s
v_s	the visit ratio of station s

Step 1: Initialize:

$$p_s(0 \mid 0) = 1, \qquad s = 1, \ldots, S \tag{5.27}$$
$$Q_s(0) = 0, \qquad s = 1, \ldots, S \tag{5.28}$$
$$\tilde{L}_s(0) = 0, \qquad s = 1, \ldots, S \tag{5.29}$$

Step 2: Preprocessing. For $s = 1, \ldots, S$

$$ES_{rem,s} = \frac{c_s - 1}{c_s + 1}\frac{ES_s}{c_s} + \frac{2}{c_s + 1}\frac{1}{c_s}\frac{ES_s^2}{2ES_s} \tag{5.30}$$

Step 3: Iteration. For $n = 1, \ldots, N$

(a) For $s = 1, \ldots, S$

$$ET_s(n) = Q_s(n-1)ES_{rem,s} + \tilde{L}_s(n-1)\frac{ES_s}{c_s} + ES_s \tag{5.31}$$

(b)

$$\tau(n) = \frac{n}{\sum_{s=1}^{S} v_s ET_s(n)} \tag{5.32}$$

(c) For $s = 1, \ldots, S$ and for $b = 1, \ldots, \min(c_s - 1, n)$

$$p_s(b \mid n) = \frac{ES_s}{b} v_s \tau(n) p_s(b-1 \mid n-1) \tag{5.33}$$

(d) For $s = 1, \ldots, S$, if $n < c_s, Q_s(n) = 0$, otherwise,

$$Q_s(n) = \frac{ES_s}{c_s} v_s \tau(n)[Q_s(n-1) + p_s(c_s - 1 \mid n-1)] \tag{5.34}$$

(e) For $s = 1, \ldots, S$

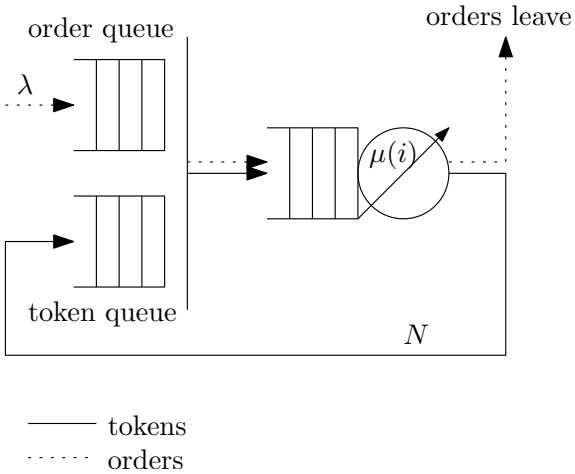
$$p_s(0 \mid n) = 1 - \sum_{b=1}^{\min(c_s-1,n)} p_s(b \mid n) - Q_s(n) \tag{5.35}$$

(f) For $s = 1, \ldots, S$, if $n < c_s, \tilde{L}_s(n) = 0$, otherwise,

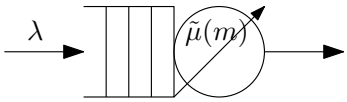
$$\tilde{L}_s(n) = \frac{ES_s}{c_s} v_s \tau(n)[\tilde{L}_s(n-1) + Q_s(n-1)] \tag{5.36}$$

5.9 Synchronization with a Load-dependent Queue

This section shows how to transform a queueing network consisting of a synchronization station with a load-dependent queue into a queueing network consisting of only a load-dependent queue. The transformation is a rather accurate approximation, but it is not exact. A queueing network consisting of a synchronization station and a load-dependent queue is shown in Figure 5.11a. Figure 5.11b shows a queueing network consisting of only a load-dependent queue.



(a) A queueing network with a synchronization station and a load-dependent queue



(b) A queueing network with only a load-dependent queue

Figure 5.11 Queueing Networks

In the queueing network in Figure 5.11a, orders arrive with a rate λ at the synchronization station, where the orders are matched with so-called tokens. After being matched with a token the order-token pair goes to a load-dependent queue. If there are i order-token pairs at the load-dependent queue, then the service rate is $\mu(i)$. When service at the load-dependent queue finishes, an order-token pair leaves the load-dependent queue. The order of that order-token pair leaves the system and the token itself returns to the synchronization station. The system contains a total of N tokens. The state space of this system can be described as (j, i), where j represents the number of orders waiting at the synchronization station and i represents the number of order-token pairs at the load-dependent queue. The number of tokens waiting at the synchronization station is denoted by k, with $k = N - i$. If $j > 0$ then it follows that $k = 0$, otherwise an order and token can be matched and go to the load-dependent queue. Similarly, if $k > 0$ then it follows that $j = 0$. Table 5.11 shows the transitions from a state (j, i) to another state. Time is considered to be

continuous, so it is not possible to increase or decrease j at the exact same moment that i is increased or decreased, and vice versa, nor can the state remain the same after a transition. At each transition either j changes or i changes but not both simultaneously. However, the system is not a Markov chain, because the memoryless property does not apply. If an order arrives at the synchronization station and matches with a token, it moves to the load-dependent queue, where service time is reset / redrawn. However, if an order arrives at the synchronization station and no token awaits, the order stays at the synchronization station and the service time at the load-dependent queue is not reset / redrawn, so the memoryless property does not apply in this situation. Therefore, when $j > 0, i = N$, the transition rate is approximately $\mu(N)$ rather than exactly $\mu(N)$.
Figure 5.11b shows a queueing network with only a load-dependent queue, where the service rate is $\tilde{\mu}(m)$ if m orders are waiting at the queue. If we would set the service rates μ as $\mu = \tilde{\mu}$, then the two networks are equivalent when $i \leq N$, but diverge when $i > N$, because the queueing network in Figure 5.11a does not have the memoryless property in that case, whereas the queueing network in Figure 5.11b does.

Table 5.11 Transition rates from state (j, i) to another state x

State (j, i)	$x = (j+1, i)$	$x = (j-1, i)$	$x = (j, i+1)$	$x = (j, i-1)$
$j = 0, i = 0$	0	0	λ	0
$j = 0, 0 < i < N$	0	0	λ	$\mu(i)$
$j = 0, i = N$	λ	0	0	$\mu(N)$
$j > 0, i = N$	λ	(approx.) $\mu(N)$	0	0

5.10 Stockout Probability

The stockout probability ψ_s is the probability that a SKU is not in the storage area when an order arrives at the warehouse. In this paper we aggregate all SKUs into one SKU, so technically a stockout only happens when there is not a single unit left in the storage area, but this is not realistic. Therefore this appendix shows how to calculate the stockout probability ψ_s given a set of SKUs, in a more realistic fashion. [22] show that it is beneficial to spread the inventory of a SKU across multiple pods, as this decreases the average travel time of a SKU to a workstation. Let SKU c be distributed across m_c different pods, then that SKU is stocked out if all m_c pods on which the SKU is located are empty. The number of pods is M, and the number of empty pods is M_r. If $m_c > M_r$, then the stockout probability for SKU c is zero, since there is at least one pod left in the inventory, with the SKU. If $m_c \leq M_r$, then the

stockout probability for SKU c is calculated as follows. If $m_c = 1$, then a stockout happens with probability $\frac{M_r}{M}$. If $m_c = 2$, then a stockout happens with probability $\frac{M_r}{M} \times \frac{M_r-1}{M-1}$, because the probability that the second pod does not contain SKU c given that the first does not contain SKU c is $\frac{M_r-1}{M-1}$. More generally, a stockout for SKU c occurs with probability:

$$\prod_{i=0}^{m_c-1} \frac{M_r - i}{M - i}, \qquad m_c \leq M_r \tag{5.37}$$

Let $\mathcal{C}$ be the set of all SKUs, let c indicate an SKU, let ϕ_c be the probability that an order needs a unit of SKU c, and let SKU c be distributed across m_c number of pods, and let $\mathbb{1}_{m_c \leq M_r}$ be an indicator function that is one if $m_c \leq M_r$ and zero otherwise. Across all SKUs, the stockout probability ψ_s in state s is then given by Equation (5.38).

$$\psi_s = \sum_{c \in \mathcal{C}} \phi_c \mathbb{1}_{m_c \leq M_r} \left(\prod_{i=0}^{m_c-1} \frac{M_r - i}{M - i} \right), \text{ with } M_r \in s \tag{5.38}$$

Figure 5.12 shows five different stock-out probability curves as a function of the fraction of pods to be replenished, $\frac{M_r}{M}$. The first stock-out probability curve, $\psi^{(1)}$, shows what happens if we have two classes of SKUs that are somewhat dissimilar, with $\phi_c = [0.6, 0.4]$ and $m_c = [40, 60]$. The second stock-out probability curve, $\psi^{(2)}$, models the ABC curve, where 20% of the products account for 70% of demand ("A" class products), 30% of products account for 25% of demand ("B" class products), and the remaining 50% of products accounts for 5% of demand ("C" class products). We model this by setting $\phi_c = [0.7, 0.25, 0.05]$ and $m_c = [20, 30, 50]$. For the third stock-out probability curve, $\psi^{(3)}$, we created 10 classes that are quite different from one another, for the sake of variety. We set $\phi_c = [0.01, 0.03, 0.05, 0.07, 0.09, 0.11, 0.13, 0.15, 0.17, 0.19]$ and $m_c = [19, 17, 15, 13, 11, 9, 7, 5, 3, 1]$. To see whether we can capture create a similar curve with a simpler function, we also show two curves that are a simply a power of $\frac{M_r}{M}$, namely $\tilde{\psi}^{(4)} = \left(\frac{M_r}{M}\right)^3$ and $\tilde{\psi}^{(5)} = \left(\frac{M_r}{M}\right)^7$. Figure 5.12 shows that $\tilde{\psi}^{(5)}$ provides a compromise between $\psi^{(1)}$ and $\psi^{(2)}$ on the one hand and $\psi^{(3)}$ on the other hand. Figure 5.12 shows $\psi^{(1)}$, $\psi^{(2)}$, and $\psi^{(3)}$ with continuous lines, because they have been calculated with Equation (5.38), whereas $\tilde{\psi}^{(4)}$ and $\tilde{\psi}^{(5)}$ are shown with dotted lines, because they were calculated with $\left(\frac{M_r}{M}\right)^\zeta$ instead, where ζ is a parameter. We therefore posit that, without any further, specific information about ϕ_c and m_c, $\tilde{\psi}^{(5)}$ offers a reasonable approximation of the stock-out probability as a function of the fraction of pods needing replenishment, i.e. where $\zeta = 7$.

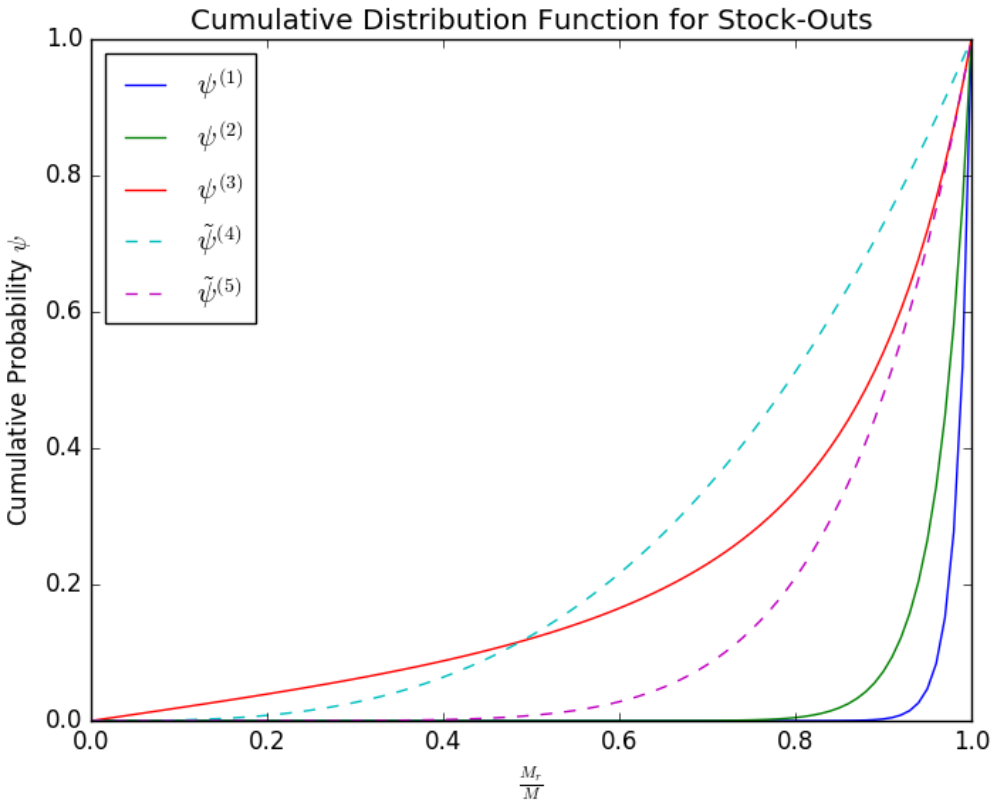


Figure 5.12 Cumulative Distribution of the stock-out probability ψ_s, as a function of the fraction of empty pods, $\frac{M_r}{M}$

For systems with a large number of SKUs, that are all equally frequently ordered, we propose the following approximation shown Proposition 3. Since in E-commerce environments the order frequency of fast movers is quite different from the order frequency of slow movers, Proposition 3 does not hold for the warehouse environment in the current study.

Proposition 3. *Given that $m_c = m$, $\phi_c = \phi \forall c \in \mathcal{C}$, and that $m \leq M_r$, it holds for single-line orders that $\psi_s = \binom{M_r}{m} / \binom{M}{m}$, where $|\mathcal{C}|$ is the number of SKUs.*

Proof. Equation (5.37) can be rewritten as shown in Equation (5.39), and Equation (5.38) as Equation (5.41). If $m > M_r$ then $\psi_s = 0$. However, if $m \leq M_r$, then Equation (5.41) can be written as Equation (5.42). Since ϕ_c is the probability that an order needs a unit of SKU, and orders are assumed to be single-line orders, we have that $\sum_c \phi_c = 1$. Moreover, since ϕ_c is the same probability ϕ, we have that $\sum_c \phi_c = \sum_c \phi = |\mathcal{C}|\phi = 1$. Therefore, Equation (5.42) can be written as Equation (5.43).

$$\prod_{i=0}^{m_c-1} \frac{M_r - i}{M - i} = \frac{M_r!(M - m_c)!}{M!(M_r - m_c)!} = \frac{M_r!(M - m)!m!}{M!(M_r - m)!m!} = \frac{\binom{M_r}{m}}{\binom{M}{m}} \tag{5.39}$$

$$\psi_s = \sum_{c \in \mathcal{C}} \phi_c \mathbb{1}_{m_c \leq M_r} \left(\prod_{i=0}^{m_c-1} \frac{M_r - i}{M - i} \right) \tag{5.40}$$

$$= \sum_{c \in \mathcal{C}} \phi \mathbb{1}_{m \leq M_r} \left(\frac{\binom{M_r}{m}}{\binom{M}{m}} \right) \tag{5.41}$$

$$= \frac{\binom{M_r}{m} |\mathcal{C}| \phi}{\binom{M}{m}} \tag{5.42}$$

$$= \frac{\binom{M_r}{m}}{\binom{M}{m}} \tag{5.43}$$

□

Chapter 6

Active repositioning of storage units in Robotic Mobile Fulfillment Systems

Marius Merschformann[1]

[1] *University of Paderborn, Paderborn, Germany*
marius.merschformann@upb.de

Selected Papers of the Annual International Conference of the German Operations Research Society (GOR), Freie Universität Berlin, Germany, September 6-8, 2017

Abstract

In our work we focus on Robotic Mobile Fulfillment Systems in e-commerce distribution centers. These systems were designed to increase pick rates by employing mobile robots bringing movable storage units (so-called pods) to pick and replenishment stations as needed, and back to the storage area afterwards. One advantage of this approach is that repositioning of inventory can be done continuously, even during pick and replenishment operations. This is primarily accomplished by bringing a pod to a storage location different than the one it was fetched from, a process we call *passive pod repositioning*. Additionally, this can be done by explicitly bringing a pod from one storage location to another, a process we call *active pod repositioning*. In this work we introduce first mechanisms for the latter technique and conduct a simulation-based experiment to give first insights of their effect.

6.1 Introduction

In today's increasingly fast-paced e-commerce an efficient distribution center is one crucial element of the supply chain. Hence, new automated parts-to-picker systems have been introduced to increase throughput. One of them is the Robotic Mobile Fulfillment System (RMFS). In a RMFS mobile robots are used to bring rack-like storage units (so-called pods) to pick stations as required, thus, eliminating the need for the pickers to walk and search the inventory. A task which can take up to 70 % of their time in traditional picker-to-parts systems (see [6]). This concept was first introduced by [7] and an earlier simulation work by [2]. The first company implementing the concept at large scale was Kiva Systems, nowadays known as Amazon Robotics.
One of the features of RMFS is the continuous resorting of inventory, i.e. every time a pod is brought back to the storage area a different storage location may be used. While this potentially increases flexibility and adaptability, rarely used pods may block prominent storage locations, unless they are moved explicitly. This raises the question whether active repositioning of pods, i.e. picking up a pod and moving it to a different storage location, can be usefully applied to further increase the overall throughput of the system. In order to address this issue we focus on two approaches for active repositioning. First, we look at repositioning done in parallel while the system is constantly active and, second, we look at repositioning during system downtime (e.g. nightly down periods). While the assignment of storage locations to inventory is a well studied problem in warehousing (see [1]) the repositioning of inventory is typically not considered for other systems, because it is usually very expensive.

6.2 Repositioning in RMFS

In a RMFS *passive repositioning* of pods is a natural process, if the storage location chosen for a pod is not fixed. For example, in many situations using the next available storage location is superior to a fixed strategy, because it decreases the travel time of the robots and by this enables an earlier availability for their next tasks. However, this strategy might cause no longer useful pods to be stored at very prominent storage locations, which introduces the blocking problem discussed earlier. There are two opportunities to resolve this issue: on the one hand it is possible to already consider characteristics of the pod content while choosing an appropriate storage location, while on the other hand it is possible to actively move pods from inappropriate storage locations to better fitting ones. We call the latter approach *active repositioning* of pods. Both repositioning approaches are shown in Fig. 6.1. Additionally, the figure shows an excerpt of the basic layout used for

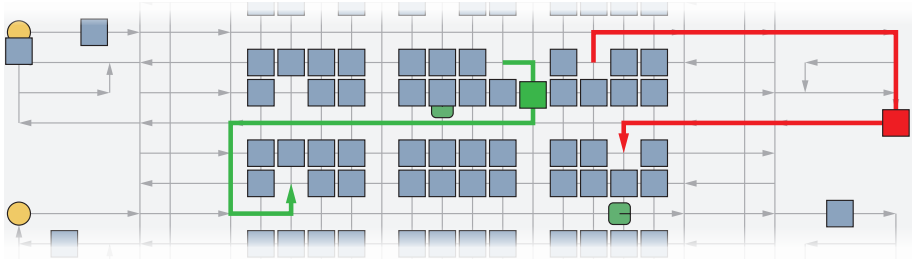


Figure 6.1 Active repositioning move (green arrow) vs. passive repositioning (red arrow)

the experiments, i.e. the replenishment stations (yellow circles), pick stations (red circles), pods (blue squares), the storage locations (blocks of 2 by 4) and the directed waypoint graph used for path planning. This layout is based on the work by [3].

In order to assess the value of storage locations and pods we introduce the following metrics. First the *prominence* F^{SL} of a storage location $w \in \mathcal{W}^{SL}$ is determined by measuring the minimum shortest path time to a pick station $m \in \mathcal{M}^{O}$ (see Eq. 6.1). The shortest path time $f^{A_t^*}$ is computed with a modified A* algorithm that considers turning times to achieve more accurate results. The storage location with the lowest $F^{SL}(w)$ is considered the most prominent one, since it offers the shortest time for bringing the pod to the next pick station. In order to assess the value of a pod b at time t we introduce the pod-speed (F^{PS}) and pod-utility (F^{PU}) measures. The *speed* of a pod (see Eq. 6.2) is calculated by summing up (across all SKUs) the units of an SKU contained (f^{C}) multiplied with the frequency of it (f^{F}). This frequency is a relative value reflecting the number of times a SKU is part of a customer order compared to all other SKUs. By using the minimum of units of an SKU contained and the demand for it (f^{D}), the *utility* of a pod (see Eq. 6.3) sums the number of potential picks when considering the customer order backlog. Thus, it is a more dynamic value. Both scores are then combined in the metric F^{PC} (see Eq. 6.4). For our experiments we consider the weights $w^{S} = w^{U} = 1$ to value both characteristics equally.

$$F^{SL}(w) := \min_{m \in \mathcal{M}^{O}} f^{A_t^*}(w, m) \tag{6.1}$$

$$F^{PS}(b,t) := \sum_{d \in \mathcal{D}} \left(f^{C}(b,d,t) \cdot f^{F}(d) \right) \tag{6.2}$$

$$F^{PU}(b,t) := \sum_{d \in \mathcal{D}} \left(\min \left(f^{C}(b,d,t), f^{D}(d,t) \right) \right) \tag{6.3}$$

$$F^{PC}(b,t) := \frac{F^{PS}(b,t)}{\max_{b' \in \mathcal{B}} F^{PS}(b',t)} \cdot w^{S} + \frac{F^{PU}(b,t)}{\max_{b' \in \mathcal{B}} F^{PU}(b',t)} \cdot w^{U} \tag{6.4}$$

Algorithm 6.1: CalculateWellsortednessCombined

1 $\mathcal{L} \leftarrow$ `Sort(` $\mathcal{W}^{SL}, i \Rightarrow F^{SL}(i)$ `)` , $r' \leftarrow 1$, $f' \leftarrow \min_{i \in \mathcal{W}^{SL}} F^{SL}(i)$
2 **foreach** $i \in \{0, \ldots,$ `Size(`$\mathcal{L}$`)`$-1\}$ **do**
3 **if** $F^{SL}(i) > f'$ **then** $r' \leftarrow r' + 1$, $f' \leftarrow F^{SL}(i)$
4 $r_i \leftarrow r'$
5 $c \leftarrow 0$, $d \leftarrow 0$
6 **foreach** $i_1 \in \{0, \ldots,$ `Size(`$\mathcal{L}$`)`$-1\}$ **do**
7 **foreach** $i_2 \in \{i_1 + 1, \ldots,$ `Size(`$\mathcal{L}$`)`$-1\}$ **do**
8 **if** `IsPodStored(`$\mathcal{L}[i_1]$`)` $\wedge$ `IsPodStored(`$\mathcal{L}[i_2]$`)` $\wedge\ r_{\mathcal{L}[i_1]} \neq r_{\mathcal{L}[i_2]}$ **then**
9 $b_1 \leftarrow$ `GetPod(`$\mathcal{L}[i_1]$`)`, $b_2 \leftarrow$ `GetPod(`$\mathcal{L}[i_2]$`)`
10 **if** $F^{PC}(b_1, t) < F^{PC}(b_2, t)$ **then** $c \leftarrow c + 1, d \leftarrow d + \left(r_{\mathcal{L}[i_2]} - r_{\mathcal{L}[i_1]}\right)$
11 **return** $a \leftarrow \frac{d}{c}$

For evaluation purposes we can use these measures to determine an overall "well-sortedness" score for the inventory. The procedure for calculating the well-sortedness score is described in Alg. 6.1. At first we sort all storage locations by their prominence score in ascending order (see line 1 f.). Next, ranks r_i are assigned to all storage locations $i \in \mathcal{W}^{SL}$, i.e., the best ones are assigned to the first rank and the rank is increased by one each time the prominence value increases (see line 2 f.). Then, we assess all storage location two-tuples and count misplacements, i.e., both storage locations are not of the same rank and the score of the better placed pod at i_1 is lower than the worse placed pod at i_2 (see line 5 f.). In addition to the number of misplacements we track the rank offset. From this we can calculate the average rank offset of all misplacements, i.e., the well-sortedness. Hence, a lower well-sortedness value means a better sorted inventory according to the given combined pod-speed and pod-utility measures.
In this work we investigate the following repositioning mechanisms:

Nearest (N) For passive repositioning this mechanism always uses the nearest available storage location in terms of estimated path time ($f^{A^*_t}$). This mechanism does not allow active repositioning.

Cache (C) This mechanism uses the nearest 25 % of storage locations in terms of estimated path time ($f^{A^*_t}$) as a cache. During passive repositioning pods with combined score (F^{PC}) above a determined threshold are stored at a cache storage location and others are stored at one of the remaining storage locations. In its active variant it swaps pods from and to the cache.

Utility (U) This mechanism matches the pods with the ranks of the storage locations (see Alg. 6.1) on the basis of their combined score (F^{PC}). A nearby

storage location with a close by rank is selected during passive repositioning. In its active variant pods with the largest difference between their desired and their actual storage location are moved to an improved one.

6.3 Computational results

For capturing and studying the behavior of RMFS we use an event-driven agent-based simulation that considers acceleration / deceleration and turning times of the robots (see [5]). Since diverse decision problems need to be considered in an RMFS we focus the scope of the work by fixing all remaining mandatory ones to simple assignment policies and the FAR path planning algorithm described in [4]. A more detailed overview of the core decision problems of our scope are given in [5]. For all experiments we consider a simulation horizon of one week, do 5 repetitions to reduce noise and new customer and replenishment orders are generated in a random stream with a Gamma distribution ($k = 1, \Theta = 2$) used for the choice of SKU per order line from 1000 possible SKUs. Furthermore, we analyze repositioning for four layouts. The specific characteristics are set as follows:

Layout	Small	Wide	Long	Large
Stations (pick / replenish)	4/4	8/8	4/4	8/8
Aisles (hor. x vert.)	8x10	16x10	8x22	16x22
Pods	673	1271	1407	2658

For the evaluation of active repositioning effectiveness we consider two scenarios. At first, we look at a situation where the system faces a nightly down period (22:00 - 6:00) during which no worker is available for picking or replenishment, but robots can be used for active repositioning. In order to keep the replenishment processes from obscuring the contribution of nightly inventory sorting, replenishment orders are submitted to the system at 16:00 in the afternoon in an amount that is sufficient to bring the storage utilization back to 75 % fill level. For pick operations we keep a constant backlog of 2000 customer orders to keep the system under pressure. Additionally, we generate 1500 orders per station at 22:00 in the evening to increase information for the pod utility metric about the demands for the following day. Secondly, we look at active repositioning done in parallel in a system that is continuously in action. For this, we consider three subordinate configurations distributing the robots per station as following:

R1P3A0: $\frac{1}{4}$ replenishment, $\frac{3}{4}$ picking, no active repos.

R1P2A1: $\frac{1}{4}$ replenishment, $\frac{2}{4}$ picking, $\frac{1}{4}$ active repos.

R1P3A1: $\frac{1}{5}$ replenishment, $\frac{3}{5}$ picking, $\frac{1}{5}$ active repos. (+1 robot per station)

Table 6.1 Unit throughput rate score of the different scenarios, layouts and mechanisms (values in percent (%), read columns as [passive mechanism-active mechanism])

Layout	Small				Wide				Long				Large			
Setup/Mech.	C-C	U-U	N-C	N-U	C-C	U-U	N-C	N-U	C-C	U-U	N-C	N-U	C-C	U-U	N-C	N-U
Deactivated	52.9	53.6	55.0	55.0	50.6	50.5	52.7	52.7	43.3	46.0	47.8	47.8	42.6	44.8	46.3	46.3
Activated	52.9	53.5	55.5	55.9	50.5	50.4	53.1	53.5	42.8	46.3	49.0	49.7	42.2	45.1	47.6	48.2
R1P3A0	52.6	51.7	55.6	55.6	50.3	48.9	53.3	53.3	43.3	43.4	48.2	48.2	42.4	42.4	46.6	46.6
R1P2A1	40.9	40.7	44.6	43.8	39.7	38.1	43.3	42.3	31.7	33.8	39.1	38.1	31.5	32.8	38.8	37.8
R1P3A1	51.9	51.5	55.3	54.5	50.3	48.5	53.7	53.0	41.6	44.0	49.3	48.1	41.2	42.6	48.9	47.6

This scenario is kept under continuous pressure by keeping a backlog of constant size for both: replenishment (200) and customer orders (2000).
The main performance metric for the evaluation is given by the unit throughput rate score (UTRS). Since we use a constant time of $T^P = 10s$ for picking one unit an upper bound for the number of units that can possibly be handled by the system during active hours can be calculated by $UB := \left|\mathcal{M}^O\right| \frac{3600}{T^P}$ with the set of all pick stations $\mathcal{M}^O$. Using this we can determine the fractional score by dividing the actual picked units per hour in average by this upper bound.

The results of the experiment are summarized in Tab. 6.1. For the comparison of resorting the inventory during the nightly down period (line: Activated) vs. no active repositioning at all (line: Deactivated) we can observe an advantage in throughput. However, for the parallel active repositioning it is not possible to observe a positive effect. When moving one robot per pick station from pick operations to active repositioning (lines: R1P3A0 and R1P2A1) we observe a loss in UTRS, because less robots bring inventory to the pick stations. Even with an additional robot per pick station (line: R1P3A1) we cannot observe a substantial positive effect. For most cases, the effect is rather negative as a result of the increased congestion potential for robots moving within the storage area.

In the following we take a closer look at the nightly down period scenario. If we keep the system sorted with the passive repositioning mechanism (C-C and U-U), nightly active repositioning does not have a noticeable positive effect, because the passive repositioning mechanism already keeps the inventory sorted for the most part. However, the Nearest mechanism which has a better overall performance, can benefit from a nightly active repositioning (N-C and N-U). Especially for the Large and Long layouts we can observe a reasonable boost in UTRS. The greater merit for layouts with more vertical aisles suggest that shorter trip times of the robots are the reason. This can also be observed when looking at the detailed results of a run with and without active repositioning (see Fig. 6.2). First, more orders can

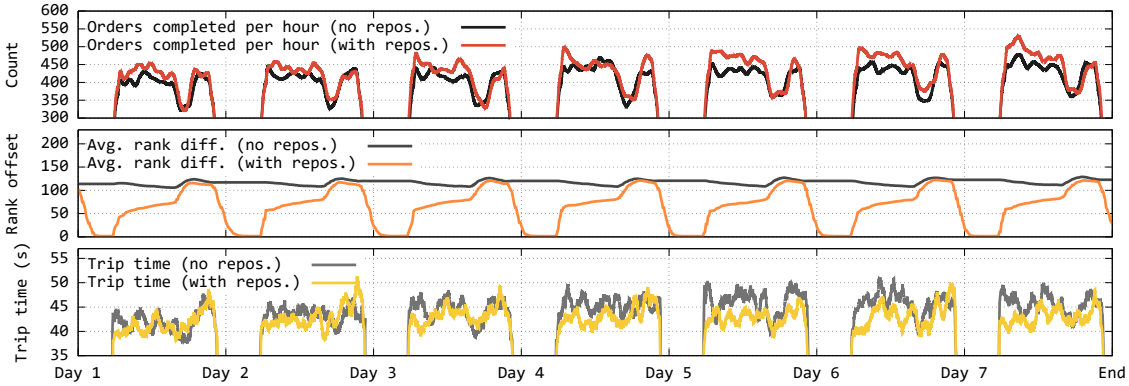


Figure 6.2 Time-wise comparison of layout Long and mechanisms N-U with (colored lines) and without (gray lines) active repositioning at night

be completed per hour after the inventory was sorted over night (first graph). This boost is eliminated as soon as replenishment operations begin. Thus, when and how replenishment is done is crucial to the benefit of resorting during down times, because the effect may be lost quite quickly. In the third graph the shorter times for completing trips to the pick stations after sorting the inventory support the assumption that these are the the main reason for the boost. Lastly, the second graph provides the well-sortedness measure and shows that sorting the inventory can be done reasonably fast. The situation before and after sorting is shown in Fig. 6.3. In this heatmap the combined score ($F^{PC}(b,t)$) is visualized with one colored tile of the size of a storage location. Here the most useful and just replenished pods can be seen positioned next to the replenishment stations (left side) before sorting. After sorting most useful pods are positioned on the far right side of the horizontal aisles inbound to a pick station. Thus, these pods offering a high potential hit-rate (number of picks from a pod) can be fetched most quickly.

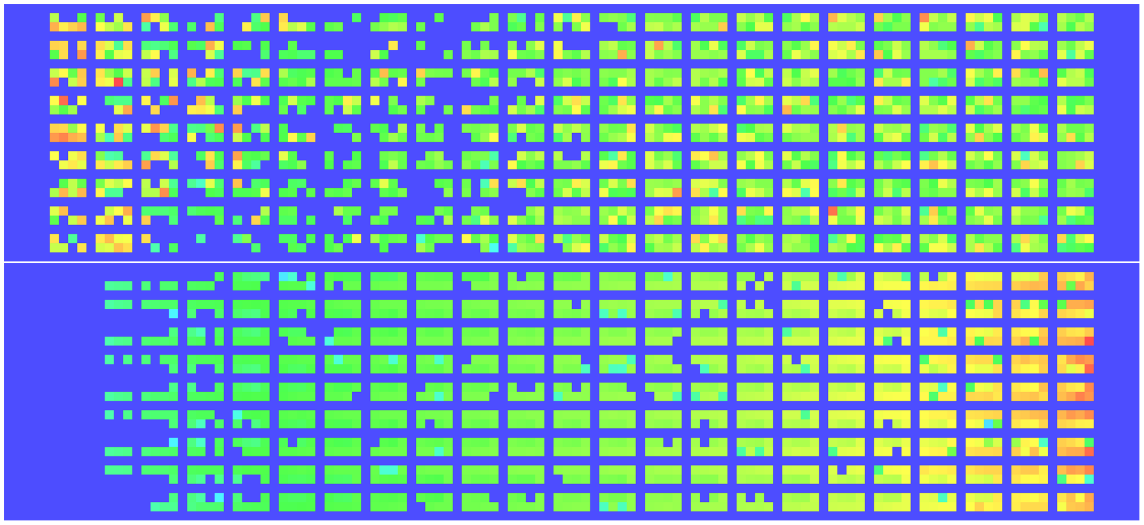


Figure 6.3 Comparison of inventory situation before and after nightly active repositioning (Long layout, N-U mechanisms, top: day 4 22:00, bottom: day 5 06:00)

6.4 Conclusion

The results suggest that active repositioning may boost throughput performance of RMFS. If the system faces regular down periods, costs for repositioning (energy costs, robot wear) are reasonable and charging times allow it, active repositioning can make a reasonable contribution to a system's overall performance. Since the introduced mechanisms greedily search for repositioning moves, more moves are conducted than necessary to obtain a desired inventory well-sortedness. For future research we suggest to predetermine moves before starting repositioning operations, e.g. by using a MIP formulation matching pods with storage locations and selecting the best moves. The source-code of this publication is available at https://github.com/merschformann/RAWSim-O.

References

[1] Jinxiang Gu, Marc Goetschalckx, and Leon F. McGinnis. "Research on warehouse operation: A comprehensive review". In: *European Journal of Operational Research* 177.1 (2007), pp. 1–21. ISSN: 03772217. DOI: 10.1016/j.ejor.2006.02.025.

[2] Christopher J. Hazard, Peter R. Wurman, and Raffaello D'Andrea. "Alphabet Soup: A Testbed for Studying Resource Allocation in Multi-vehicle Systems".

In: *Proceedings of AAAI Workshop on Auction Mechanisms for Robot Coordination*. Citeseer, 2006, pp. 23–30.

[3] T. Lamballais, D. Roy, and M.B.M. de Koster. "Estimating performance in a Robotic Mobile Fulfillment System". In: *European Journal of Operational Research* (2016). ISSN: 03772217. DOI: 10.1016/j.ejor.2016.06.063.

[4] M. Merschformann, L. Xie, and D. Erdmann. "Path planning for Robotic Mobile Fulfillment Systems". In: *ArXiv e-prints* (2017).

[5] M. Merschformann, L. Xie, and H. Li. "RAWSim-O: A Simulation Framework for Robotic Mobile Fulfillment Systems". In: *ArXiv e-prints* (2017).

[6] James A. Tompkins. *Facilities planning*. 4th ed. Hoboken, NJ and Chichester: John Wiley & Sons, 2010. ISBN: 0470444045.

[7] Peter R. Wurman, Raffaello D'Andrea, and Mick Mountz. "Coordinating hundreds of cooperative, autonomous vehicles in warehouses". In: *AI Magazine* 29.1 (2008), p. 9. ISSN: 0738-4602.

Part II

Further topics on decision support

Chapter 7

Synopsis on further topics

During the writing of this thesis further projects in Operation Resaerch & Artificial Intelligence have been engaged in collaboration with other researchers. Although these are on different topics, they, and the ones of the previous part, all share the aim to support smart decision making. In this section all of the resulting papers are outlined to allow the reader an overview of the second part of this thesis.

A problem often faced in academic decision support research is the lack of real-world problem instances. For example, the development of optimization algorithms is often driven by solving industrial problem instances. However, the access to these is typically limited. Hence, developing optimization algorithms by testing them on such small instance sets runs the risk of over-fitting the procedure to these few instances. Another example is the research on algorithm configuration and selection, which aims to improve performance of algorithms (typically execution time or solution quality) by finding improved parameter sets or selecting the right portfolio of algorithms for specific problem instances. For these typically machine learning driven methods large instance sets are mandatory to avoid over-fitting the resulting algorithm portfolio. These needs motivated the first of the following works, which introduces a method called "Structure-Preserving Instance Generation". The novel approach alters real-world instances by combining and varying them instead of artificially generating them from scratch (see Chapter 8).

The next work applies algorithm configuration methods to analyze potential improvements for the chosen parameter sets of multiple heuristics for the container loading problem faced in air cargo transport. In contrast to typical liner shipping containers with their cuboid shapes and larger size compared to the pieces they have to contain, the containers in air cargo (so-called Unit Load Devices (ULD)) are ir-

regularly shaped and the density of the packing requires the more detailed modeling of the shapes of the pieces to load. The solutions calculated by the heuristics serve as build-up plans for packers at an airport cargo hub. Thus, the computational time available is very limited, since material may arrive late and the plane cannot wait. However, unfortunately the applied algorithm configuration methods were unable to improve on the initial manually chosen set of parameters.
The last work introduces a set of metaheuristic algorithms for the crew rostering problem in public bus transit. The rationale behind the addressed problem is to assign bus drivers to work shifts, meeting many legal and work contract related requirements while considering the preferences of the drivers themselves as well as company interests. The studied algorithms base on the Simulated Annealing, Tabu Search, Ant Colony System and Max-Min Ant System metaheuristic concepts. Such metaheuristic algorithms are used to achieve solutions of proper quality in reasonable time by exploiting domain based knowledge in combination with suitable structural problem representations. This is required, if exact optimization techniques struggle to find proper solutions to real-world problem instances in reasonable time.

7.1 Paper summaries & contributions

Summaries of all included papers on further topics follow in the order they appear.

Chapter 8: Structure-Preserving Instance Generation

Since the access to real-world instances is often limited in research, but crucial to a goal-oriented development of state-of-the-art algorithms, this paper introduces a new technique for the structured generation of instances. The generator applies a large neighborhood search-like method which combines components of instances to new ones. The procedure is tested by training algorithm portfolios (using algorithm configuration and selection methods) for SAT and MaxSAT problem instances. The results suggest that the performance achieved by training on generated instances is similar to training on real-world instances. In some cases the performance even exceeds the one of the portfolios trained on real-world instances.

Main contributions:

- A novel instance generation method applied to the satisfiability problem (SAT)
- Experimental results suggesting that portfolios trained on a broader instance set obtained by the technique outperform the ones trained on a limited set

Chapter 9: Algorithm Configuration Applied to Heuristics for Three-Dimensional Knapsack Problems in Air Cargo

This paper includes a study of a algorithm configuration technique applied to heuristics for the three-dimensional knapsack problem arising in air cargo. The studied heuristics include form approximation techniques that allow the packing of non-cuboid objects of theoretically any form. However, the results of the study show that no improvement over the original parameter set could be achieved by the algorithm configuration method.

Main contributions:

- Results of an algorithm configuration experiment on a practical problem from the air cargo industry

Chapter 10: Metaheuristics approach for solving personalized crew rostering problem in public bus transit

This paper addresses the crew rostering problem in public bus transit, which aims to construct personalized monthly schedules for all drivers. The problem is formulated as a multi-objective optimization problem, since it considers the interests of both the management of bus companies (e.g., cost of caused overtime) and the drivers (e.g., preferred days off). The different objectives are combined in a weighted sum to determine the overall solution quality. Ant colony optimization, simulated annealing, and tabu search methods are then proposed to solve the multi-objective personalized crew rostering problem in public bus transit. The developed algorithms are tested on real-world instances, and the results are compared to solutions obtained by commercial solvers.
Main contributions:

- Four metaheuristic algorithms in six variants designed for the personalized crew rostering problem in public bus transit
- A detailed experiment comparing the solution quality obtained by the introduced algorithms on a real-world instance set

Chapter 8

Structure-Preserving Instance Generation

Yuri Malitsky[1] Marius Merschformann[2] Barry O'Sullivan[3] Kevin Tierney[2]

[1]*IBM T.J. Watson Research Center, New York, USA*
[2]*Decision Support & Operations Research Lab, University of Paderborn, Germany*
[3]*Insight Centre for Data Analytics, University College Cork, Ireland*
yuri.malitsky@gmail.com, merschformann@dsor.de, b.osullivan@insight-centre.org, tierney@dsor.de

LION 10 - Learning and Intelligent Optimization, At Ischia, Italy, 29.05.-01.06.2016

Abstract

Real-world instances are critical for the development of state-of-the-art algorithms, algorithm configuration techniques, and selection approaches. However, very few true industrial instances exist for most problems, which poses a problem both to algorithm designers and methods for algorithm selection. The lack of enough real data leads to an inability for algorithm designers to show the effectiveness of their techniques, and for algorithm selection it is difficult or even impossible to train a portfolio with so few training examples. This paper introduces a novel instance generator that creates instances that have the same structural properties as industrial instances. We generate instances through a large neighborhood search-like method that combines components of instances together to form new ones. We test our

approach on the MaxSAT and SAT problems, and then demonstrate that portfolios trained on these generated instances perform just as well or even better than those trained on the real instances.

8.1 Introduction

One of the largest problems facing algorithm developers is a distinct lack of industrial instances with which to evaluate their approaches. Yet, it is the use of such instances that helps ensure the applicability of new methods and procedures to the real-world. Algorithm configuration and selection techniques are particularly sensitive to the lack of industrial instances and are prone to overfitting, as it is difficult to build valid learning models when little data is present. Although a plethora of random instance generators exist, the structure of industrial instances tends to be different than that of randomly generated instances, as has been shown for the satisfiability (SAT) problem [2, 1, 20].

In this work, we therefore present a novel framework for instance generation that creates new instances out of existing ones through a large neighborhood search-like iterative process of destruction and reconstruction [28] of structures present in the instances. Specifically, given an instance to modify, m, and a pool of similar instances, P, we destroy elements of m that fit certain properties (such as variable connectivity) and merge portions of the instances in P into m, to create a set of new instances. We compute the features of each new generated instance and accept the instance if it falls into the cluster of instances defined by P. To the best of our knowledge, our framework is the first approach able to generate industrial-like instances directly from real data.

Aside from the immediate benefits of providing a good training set for portfolio techniques, this type of instance generation has the potential of opening new avenues for future research. In particular, the underlying assumption of most portfolio techniques is that a representative feature vector can be used to identify the best solver to be employed on that instance. Techniques like ISAC [19] take this idea further by claiming that instances with similar features are likely to have the same underlying structure, and can therefore be solved using the same solver. Structure-preserving instance generation uses and furthers this notion, that in order to predict the best solver for a given instance, one should create a plethora of instances with very similar features, train a model on them, and then make a prediction for the original instance. Additionally, given recent results regarding the benefits of having multiple, correlated instances for CSPs [14], our instance generator may be able to help solvers more quickly find solutions, as it can provide such correlated instances. In theory, structure-preserving instance generation can even be used to generate instances significantly different from those observed before. This could in turn allow

the targeted creation of portfolios that can anticipate any novel instances not part of the original training set. It would also allow for a systematic way of studying the problem space to identify regions of hard and easy problems, as in [31] but with a stronger basis in real instances. This would allow algorithm designers to create new approaches to specifically target challenging problems.
In this work we primarily focus on the well-known SAT problem, as well as its optimization version, maximum SAT (MaxSAT). These two problems pose ideal test beds for our approach, as although the number of industrial instances is low, it is still larger than what is available in most other domains. For example, the MaxSAT competition in 2013 [7] had only 55 industrial instances in the unweighted category, as opposed to 167 crafted and 378 random instances. We show that the instances generated by our method have similar runtime profiles as the industrial instances they are based on, that they have similar features, and that they can be used in algorithm selection techniques with no loss of performance and, in some cases, even provide small gains. We then show that our technique will even help for problems with larger available datasets, as is the case with SAT and the 300 available industrial instances from the 2013 SAT Competition [8]. We also evaluate our generator using the Q-score from [11], which is specifically designed for evaluating instance generators, and receive near perfect scores. Finally, our source code is available in a public repository at: https://bitbucket.org/eusorpb/spig.

8.2 Related Work

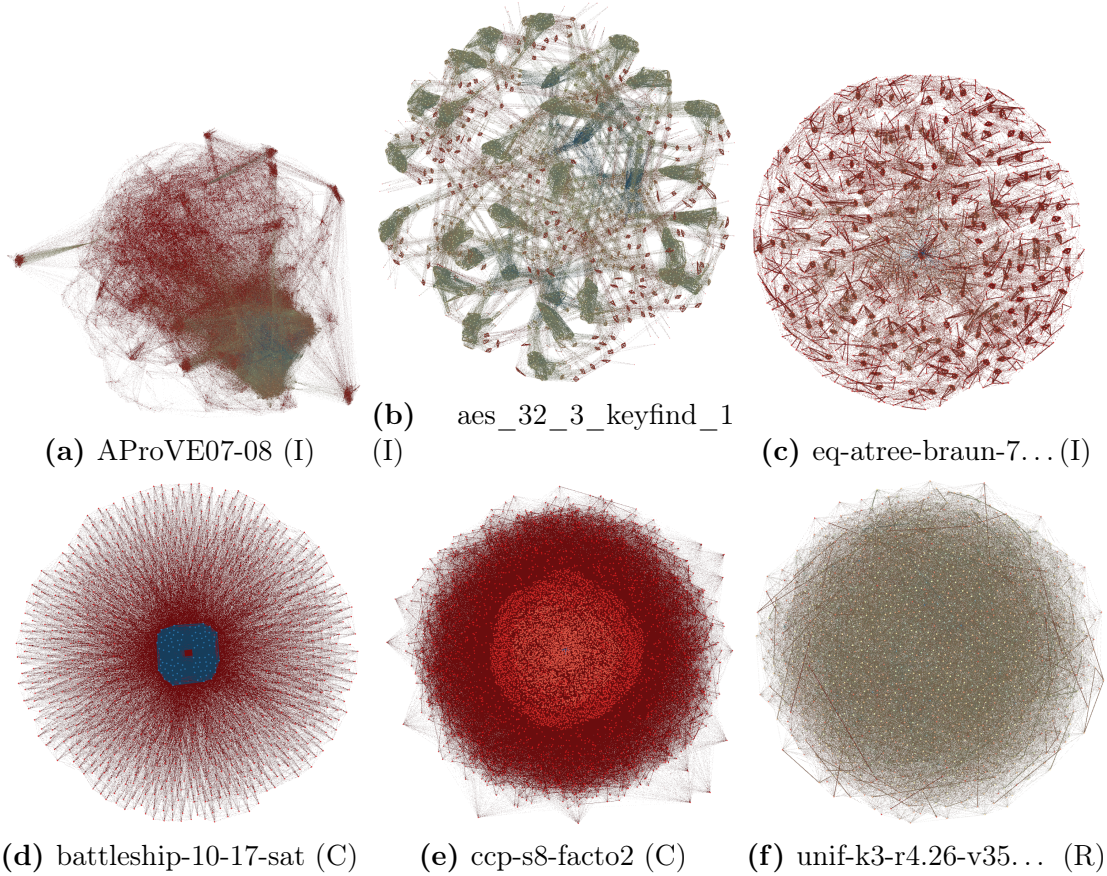
Numerous random instance generators exist for SAT/CSP problems, such as [16, 9, 32], to name a few[1]. Some generators try to hide solutions within the problem or generate a specific number of solutions ([21] and [27], respectively), whereas others convert problems from other fields to SAT/CSP problems (e.g., [5]). The approach of Slater (2002) [29] creates instances by connecting "modules" of 3SAT instances with a shared component, a structure that is often present in industrial instances. For MaxSAT, several generators exist, e.g. [12], which generates bin packing-like problems. The generator from Motoki [24] can create MaxSAT problems with a specific number of unsatisfied clauses. However, in all of these generators the structures inherent in industrial problems are not present.
The most industry-like SAT/MaxSAT instances are generated by Ansótegui et al. [4] through the modification of a random instance generator to use a power-law distribution. In contrast, our framework is able to specifically target certain types of industrial instances. Our approach is similar to instance morphing [15], the primary difference being our focus on instance features and a destroy-repair paradigm. Fur-

[1] An extended version of this work provides a more extensive literature review; see: https://bitbucket.org/eusorpb/spig/

thermore, morphing is meant to "connect" the structures of instances, while our goal is to also find new combinations of structures leading to new areas of the instances' feature space.
Burg et al. propose a way of generating SAT instances by clustering the variables based on their degree in a weighted variable graph in Burg et al. [13]. Several approaches are tested to try to "re-wire" the instance by adding and removing connections between the variables. The authors compute the features from Nudelman et al. [26] for the original instances and the generated ones, noting that several features of the generated instances no longer resemble the original instances. In contrast, the instances we generate have similar features to their original instances and stay in the same cluster as the original instance pool.
An evolutionary algorithm approach is used by Smith-Miles and van Hemert [31] to generate traveling salesman problem instances that are uniquely hard or easy for a set of algorithms. This approach starts from a random instance under some assumptions about the size and structure of the resulting instance. In Lopes and Smith-Miles [22], real-world-like instances for a timetabling problem are generated using an existing instance generator. While similar to our approach, their work focuses mainly on generating instances that are able to discriminate between solvers in terms of performance. Furthermore, our approach does not require an existing instance generator. TSP instances are also evolved in Nallaperuma et al. [25] for a parameter prediction model for an ant colony optimization model. However, all these works focus on creating hard instances for particular solvers, rather than instances that resemble industrial instances.
The most similar work to ours is from Smith-Miles and Bowly [30], in which instances are generated for the graph coloring problem targeting specific instance features. The authors project their features into a two dimensional space with a principal component analysis, and then check if the instance features to be generated are in a feasible region of the space. A genetic algorithm is then used to try to find an instance matching the input features. This approach is more general than ours, but it is not known how well it works with industrial instances, or other problem types.

8.3 Structure-preserving Instance Generation

Our instance generation algorithm is motivated by the differing structures found in SAT instances, especially between the industrial, crafted and random categories of instances. Figure 8.1 provides some visualizations of SAT instances based on their clause graphs. In these graphs, each node represents a clause in the formula, with an edge specifying that the two clauses share at least one variable. The nodes are also color coded from red to blue, where nodes with only a few edges are colored red and those with the most edges are colored blue. A force-based algorithm is used to



(a) AProVE07-08 (I) **(b)** aes_32_3_keyfind_1 (I) **(c)** eq-atree-braun-7... (I)

(d) battleship-10-17-sat (C) **(e)** ccp-s8-facto2 (C) **(f)** unif-k3-r4.26-v35... (R)

Figure 8.1 Visualizations of industrial (I), crafted (C) and random (R) instances made with Gephi [10]. Nodes are clauses, and an edge exists if the corresponding clauses share a variable.

spread nodes apart. In this way, nodes that share many edges between each other are pulled together into clumps, while the others are pushed away.

Note that the industrial instances, which in Figure 8.1 are a, b and c, tend to contain a core set of clauses that share at least one variable with many other clauses. In addition, a large number of small subsets of clauses are built on the same set of variables. A few variables link the subsets of clauses to the common core. In contrast, instances d and e, which are from the crafted category from the SAT competition, and f, which is from the random category, show significantly more connectivity between clauses and less modules or groupings of nodes within the graph.

Given a pool of instances, P, and an instance to modify, m, structure-preserving instance generation works as shown in Algorithm 8.1 to create a set of generated instances *gen*. The instance pool should be a set of homogeneous instances, such

Algorithm 8.1: Structure-preserving instance generation algorithm.

```
1  Function SpIG(m, P, α, β)
2      gen ← ∅, m′ ← m
3      repeat
4          do
5              m′ ← DESTROY(m′, SELECT-STRUCT(m′))
6          while SIZE(m′) > β · SIZE(m)
7          do
8              i ← random instance in P
9              d ← max(0, VARS(m) − VARS(m′))
10             m ← REPAIR(m′, SELECT-STRUCT(i), d)
11         while SIZE(m′) < α · SIZE(m)
12         if ACCEPT(m′, P ∪ {m}) then
13             gen ← gen ∪ {m′}
14     until TERMINATE
15     return gen
```

as the instances in a particular cluster from the ISAC method [19]. While using a heterogeneous pool would still result in instances, recall that in this work we aim to create instances with similar properties. In cases where an industrial instance has no similar instances, our method can still be used with a pool consisting of only the instance to modify. The parameters $\alpha \in [0, 1]$ and $\beta \in [1, \infty]$ define the minimum and maximum of the size of the generated instances in proportion to m, respectively. We use these values as general guidelines rather than hard constraints in order to prevent the instance from growing too large or too small.

Our proposed algorithm can be thought of as a modified large neighborhood search [28], in which the incumbent solution, in this case the instance to modify, is iteratively destroyed and repaired. The DESTROY function identifies and removes a particular structure or component of m. The destroy process is run at least once, and is continued until the instance drops below its maximum size. The REPAIR function then identifies and extracts structures from one or more randomly chosen instances from P and inserts them into m. This is repeated until m is larger than the minimum instance size. The d parameter taken by the repair method makes sure that the total number of variables in the problem also stays constant. An acceptance criterion determines whether or not the modified instance should be added to the dataset of instances being built. We base this acceptance on the features of the instance and check whether each individual feature is close to the features of the cluster formed by $P \cup \{m\}$. For problems like SAT or CSP, where an unsatisfiable

component or tautology could be introduced, a check can be performed to ensure that the instance did not become trivial to solve. The algorithm terminates when enough instances are generated.

Here it may be argued that an alternate search strategy may also work as well or even better than the one outlined in this section. While alternatives are certainly possible, they are beyond the scope of this work (although some have been tried). For example, one can imagine a combination of local search strategies where each method adjusts an instance to match some combination of features. This modifies the internals of an instance while keeping the instance features relatively unchanged, or moves them back if they change too much. The issue with this method is that the features of interest for problems like SAT are highly interdependent, making any fine grained control over them an arduous task at best. Furthermore, it is frequently very easy to make an instance trivial to solve by introducing an infeasibility, a position that is very difficult to remedy.

Alternatively, one can argue that as long as the provided acceptance criteria is utilized as is, it is possible to employ a local search to just try a number of instantiations. While possible in theory, this approach can take a considerable amount of time before stumbling over even a single seemingly useful instance. The problem space of instance generation is simply too vast. Therefore, while we do not claim that the approach presented here is the only way of generating instances or even the best way, it is a systematic procedure that allows rich datasets to be quickly generated that we can empirically demonstrate works well in practice.

8.4 Application to SAT and MaxSAT

We present an instantiation of the structure-preserving instance generation framework on the NP-complete SAT problem and NP-hard MaxSAT problem. A SAT problem consists of a propositional logic formula F given in conjunctive normal form. The goal of the SAT problem is to find an assignment to the variables of F such that F evaluates to true. MaxSAT is the optimization version of SAT, in which the goal is to find the largest set of clauses of F that have some satisfying assignment. A version of MaxSAT can also have a weight associated with each clause, with the objective then being to maximize the sum of satisfied clauses. In this work, however, we concentrate only on the unweighted variant of MaxSAT and describe our structure identification routines (SELECT-STRUCT), destroy, repair and acceptance operators. Due to the similarity of the SAT and MaxSAT problems, our instance generation procedure is the same with the exception of the acceptance criteria, which we modify to avoid trivial SAT instances.

Algorithm 8.2: Variable based structure selection heuristic.

```
1 Function select-struct-var(i)
2     E ← ∅, f ← 0
3     a ← MEAN-VAR-IN-CLAUSES(i)
4     while E = ∅ do
5         E ← {v ∈ VARS(i) | |CLAUSES(v) − a| ≤ f}
6         f ← f + 1
7     return IN-CLAUSES(random variable in E)
```

8.4.0.1 Structure Identification

Many industrial SAT/MaxSAT instances consist of a number of connected components that are bound together through a core of common variables (see Figure 8.1). Our goal is to identify one of these components in an instance at random and remove it. To this end, we propose two heuristics for identifying such structures that we use in both the destroy and repair functions with 50% probability in each iteration. The first heuristic identifies a set of clauses shared by a particular variable, whereas the second identifies a clause and selects all of the clauses it shares a variable with.

Our goal in the *variable-based selection* heuristic is to identify components of instances with a shared variable, as shown in Algorithm 8.2. We first calculate the mean number of clauses that a variable is in. Variables in many clauses are likely to be a part of the "core" of an instance that connects various sub-components, whereas variables in the average number of clauses are more likely to be part of the sub-components themselves. The algorithm selects a set of variables, E, in the average number of clauses, if there are any. If E is empty, the algorithm relaxes its strictness of how many clauses a variable should be in until some clauses are found. Finally, the algorithm selects a variable from E at random and returns all of the clauses that variable is present in.

In contrast to our previous heuristic, the *clause-based selection* heuristic focuses on clauses with an average out degree. The out degree of a clause is defined as the number of clauses sharing at least one variable in common with the clause. This corresponds to the out degree of the clause's node in the clause-variable graph. Algorithm 8.3 accepts an instance i and a parameter σ, described below. The heuristic selects a random clause and compares its out degree to the average out degree of all the clauses. If the clause's out degree is within σ standard deviations of the average clause out degree, we accept the clause and return all of the clauses it is connected to in the clause-variable graph.

Algorithm 8.3: Clause based structure selection heuristic.

```
1  Function select-struct-clause(i, σ)
2      C = ∅
3      mcod ← MEAN-CLAUSE-OUT-DEGREE(i)
4      scod ← STD-CLAUSE-OUT-DEGREE(i)
5      while C = ∅ do
6          c ← random clause in i
7          cod ← CLAUSE-OUT-DEGREE(c)
8          if |cod − mcod| < σ · scod then
9              C ← {c′ ∈ CLAUSES(i) | c and c′ share at least one variable.}
10     return C
```

8.4.0.2 Destroy

Our destroy function accepts an instance i and a set of clauses C selected by SELECT-STRUCT-VAR or SELECT-STRUCT-CLAUSE that are to be removed from the instance. First, all clauses in C are removed, i.e., $\text{CLAUSES}(i) \leftarrow \text{CLAUSES}(i) \setminus C$, and then all variables that no longer belong to any clause are removed from $\text{VARS}(i)$.

8.4.0.3 Repair

Our repair procedure maps the variables contained within a previously selected set of clauses to the variables present in the instance to modify, and adds new variables with some probability. To avoid confusion, we refer to the instance being modified as the *receiver*, and the instance providing clauses as the *giver*. Algorithm 8.4 shows the repair process, which is initialized with the receiving instance r, the set of clauses to add, C, and some number of variables to add to the instance, d. The parameter d is used to increase the size of the receiver if too many variables are deleted during the destruction phase. Additionally, we note that C contains clauses from the giver, meaning the variables in those clauses do not match those in the receiving instance. Thus, the main action of the repair method is to find a mapping, M, that allows us to convert the variables in C into similar variables in the receiver.
We map the variables in C into the variables of r by computing the following three features for each variable in the VAR-FEATURES function. We use these features because our goal is to map variables with similar connectivity to other parts of the instance with each other and they are easy to compute.

1. Number of clauses the variable is in divided by the total number of instance clauses.

Algorithm 8.4: SAT/MaxSAT instance repair procedure.

```
1  Function Repair(r, C, d)
2      V_g ← ⋃_{c∈C} VARS(c)
3      F_r ← VAR-FEATURES(r)
4      F_g ← VAR-FEATURES(V_g)
5      M ← ∅
6      for v_g ∈ V_g do
7          if ¬TRIVIAL(v_g) and RND(0,1) < d/|V_g| then
8              v' ← new variable
9              VARS(r) ← VARS(r) ∪ {v'}
10             M ← M ∪ {v_g ↦ v'}
11         else
12             dists ← {‖F_r(v) − F_g(v_g)‖², ∀v ∈ VARS(r)}
13             v' ← argmin_{v∈VARS(r)}{dists(v) | v ∉ M}
14             M ← M ∪ {v_g ↦ v'}
15     CLAUSES(r) ← CLAUSES(r) ∪ MAP-VARS(C, M)
16     return r
```

2. Percent of clauses the variable is in, in which the variable is positive.

3. Average of the number of variables of each clause the variable v is in.

On Line 7 of Algorithm 8.4 we decide whether to map v_g to an existing variable in r or to a new variable. The function $\textsc{trivial}(v_g)$ returns true if v_g is not (i) positive in at least one clause in C, and (ii) negated in at least one clause C. This ensures that if we map v_g to a new variable, a valid assignment of v_g is not entirely obvious. We assign v_g to a new variable with probability $d/|V_g|$, as with this probability we add roughly d new variables in the absence of trivial variables.

We compute the $L2$ norm between the giver variables and the receiver's variables on line 12, and should we decide not to add a new variable to r, we map v_g to the variable in r that most closely resembles its features that is not yet assigned to a different variable. This is a greedy procedure, that finds the best match for each variable individually. Finally, the algorithm performs the variable mapping and merges the clauses of C into r. We omit the details of the merging process as it is straight forward.

8.4.0.4 Acceptance Criteria

We compute a set of well known features for SAT and MaxSAT[2] problems from [26] in order to determine whether to accept a modified instance. We compute the average and standard deviation for each feature across the entire pool of instances (including the instance to modify). An instance is accepted if all of its features are within three standard deviations of the mean. That is, we compare a feature to the cluster center on a feature by feature basis. However, some features do not vary at all in a cluster, meaning they have a standard deviation of 0. In such cases, even small changes to an instance can result in a rejection of all generated instances, although the instance is for the most part within the cluster. Thus, when absolutely no instance could be generated we relax the conditions for features that do not vary within the cluster, and allow them to vary by some epsilon value. We note that in our experiments such instances were still well situated within clusters when measured with the Euclidean distance to the cluster center.
For SAT problems, we extend this acceptance criteria with an execution of the instance with a SAT solver. If the instance is solvable in under 30 seconds it is discarded. We do this because our generation procedure sometimes introduces unsatisfiable components to satisfiable problems that are easily found and exploited by solvers. This clearly breaks the structure of the instance that we are striving to preserve, thus the instance must be discarded. Note that this does not guarantee that the instance will be satisfiable, all we are checking is that the generated instance is not trivially solvable. This issue generally only happens to a couple of instances per generation procedure.

8.5 Computational Evaluation

We perform an evaluation of structure-preserving instance generation on instances from the MaxSAT and SAT competitions. We show that the instances generated using our method preserve structure well enough such that they are effectively solved using the same algorithms as the original instances. To evaluate this, we train an algorithm selection approach on the generated data and evaluate it on the subset of original test instances that were neither part of the training nor the generation. It is assumed that if our generated instances can allow us to train a portfolio to identify the most appropriate solver for the instance at hand, then they successfully embody the same key structures as the original industrial data. For SAT, we also use the Q-score method of [11] to show the quality of our instance generator. All experiments were performed on a cluster of Intel Xeon E5-2670 processors with 4GB of RAM

[2]We do note use local search probing features in this work.

for random instances and 12GB for industrial instances for MaxSAT (as industrial MaxSAT instances are very large), and 4GB of RAM for all SAT instances.

8.5.1 MaxSAT

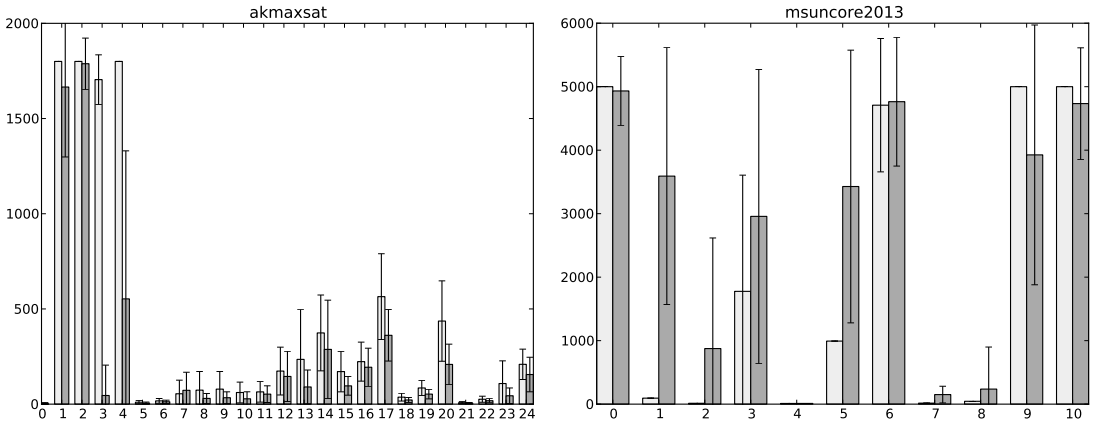
For MaxSAT, we evaluate our technique on both the random as well as industrial instances from the MaxSAT 2013 competition [7]. Using the random dataset in addition to the industrial dataset shows that our method can be used for any group of similar instances, even though our main target is industrial instances. Here we generate our datasets according to a manually established similarity measure based on the filenames of the instances. For each pool of instances, we perform 10 instance generations for each instance of the pool with a different random seed. For the experiments presented in this section, we limit each generation attempt to 25 destroy/repair iterations. This process generated 5,306 instances based off of 378 random instances, and 2,606 instances based off of 42 industrial instances[3]

One measure to ensure the quality of the generated instances is to compare the runtime of a solver on both the original and the new dataset. Should the runtime performance of an algorithm be similar on both the original and the new dataset, we can conclude that the new dataset is similar to the old one. This is a desirable property for our instance generator, and is based on a fundamental argument on which algorithm portfolios are built and what makes them so successful in practice: that a solver/algorithm performs analogously well or poorly on instances that are similar.

Figure 8.2 shows the average solution time of the original instances and their generated counterparts for the akmaxsat and msuncore2013 solvers for several clusterings of instances on the random and industrial datasets, respectively. The clusters were generated based on the categories of the instances. The solutions times are comparable for both random and industrial instances, with the exception of clusters 2 and 5 on the industrial dataset in which the generated instances are too hard. This runtime performance similarity strongly indicates that our generator preserves the structure of instances during generation.

Note that by comparable runtimes, we do not mean identical, which would be an undesirable quality since generated instances should be slightly different from the originals. Furthermore, even when running a solver on the same instance runtimes can vary. The results displayed for the industrial dataset are somewhat noisy, due to the fact that very few original instances exist as a basis for comparison. For example, if we had more instances for cluster 2 (in reality we only have a single instance), it could very well be the case that they are hard to solve as well, but the

[3]The MaxSAT 2013 dataset contains 55 instances, but we remove instances over 110 MB after performing unit propagation, as SpIG cannot fit them in RAM.



(a) Akmaxsat on the random dataset **(b)** Msuncore2013 on the industrial dataset

Figure 8.2 The average solution time in CPU seconds and standard deviation for each cluster in terms of original (left bar, light gray) and generated (right bar, dark gray) instances.

one instance we have turned out to be solved through a smart (or lucky) branching decision by msuncore2013.

Another important result of our CPU runtime experiments is that "industrial" solvers perform well on our generated industrial instances, whereas "random" solvers tend to timeout. The opposite is also true; when we run an industrial solver on our generated random instances the industrial solvers tend to timeout, but the random solvers perform well. This means that our instance generation framework is able to preserve instance structure nearly regardless of what type of instance it is used on. We note, however, that we do not intend for our instance generator to be used on random or crafted instances, as perfectly good generators already exist for these categories of instances. We show results from these categories only to serve as an evaluation of the overall approach.

One might not even expect our generator to work at all on random instances, as they tend to have little structure. We believe the effectiveness of our approach for such instances is simply due to random changes to a random instance not having a huge effect. Industrial instances (or any instance with some kind of global/local structure), however, require an approach like the one we provide so that generated instances do not get malformed through completely random changes.

Our final experimental comparison on the MaxSAT dataset observes the effect of training a simple portfolio on only the generated instances as opposed to the original ones. Table 8.1 shows the performance of a portfolio trained and evaluated using leave-one-out cross validation. Note here that for evaluating the generated dataset, none of the instances generated from the test instances were included in

Table 8.1 Comparison of a portfolio trained (leave-one-out) on only the original MaxSAT instances, and one that is trained on the generated instances. The average time is given in seconds.

	Original		Generated	
Model	Average time	Unsolved	Average time	Unsolved
Best Single	735	2	735	2
Random Forest	988	5	599	2
SVM (radial)	734	2	591	1
VBS	184	0	184	0

the training set. We compare the performance of only using the overall best solver to a portfolio that uses either a random forest or a support vector machine (SVM) to predict the runtime of each solver, selecting the solver with the best expected performance. There are of course a plethora of other popular and more powerful portfolio techniques that could be used and compared, but random forests and SVMs are readily available to anyone and have been previously shown to be effective for runtime prediction, and here we show that even they are able to perform well in our case. The virtual best solver (VBS) gives the performance of an oracle that always picks the fastest solver. Due to the limited training set, the best the portfolio can do is match the performance of a single solver. However, if we train the same solvers on the generated instances and evaluate on the original instances, we are able to see improvements over the best solver, meaning the extra generated instances provide value to the portfolio approach. Our generator is able to help fill in gaps between training instances in the feature space, allowing learning algorithms to avoid having to make a guess as to which algorithm will work best within such gaps. Instead, learning approaches have data on the instances in these gaps and can make informed decisions for their portfolio.

8.5.2 SAT

We next use the instances from the 2013 SAT competition [8] to conduct further experiments. For the competition, these instances are split into three categories each consisting of 300 instances: application (industrial), crafted, and random. And although this competition has taken place annually for the last decade, it is important to note that the majority of the industrial instances repeat each year, which means our experiments use most of the instances available.We first evaluate our generated dataset using the Q-score technique from [11]. We then use an algorithm selection approach to confirm the usefulness of our generator.

Table 8.2 Comparison of a portfolio evaluated on 195 Industrial SAT instances when trained on either 300 randomly generated instances, 300 crafted instances, a subset of 46 industrial instances, 300 generated instances, or 1500 generated instances.

	Average	PAR10	Solved
Best Single	453	3,872	157
Random (300)	541	5,107	144
Crafted (300)	386	4,090	154
Industrial (46)	348	3,426	161
Generated (300)	502	3,463	162
Generated (1,500)	437	3,049	166
VBS	364	364	195

8.5.2.1 Algorithm selection with CSHC

Due to the increased number of available instances over the Max-SAT scenario, we apply a more automated technique for grouping SAT instances for generation. In MaxSAT, the instances were grouped based on a manually defined similarity metric associated with the filenames. For SAT, however, the industrial instances are clustered based on their features using the g-means algorithm from [17], an approach that automatically determines the best number of clusters based on how Gaussian distributed each cluster is. Limiting the minimum size of a cluster to 50 resulted in a total of 7 clusters. We typically use 50 instances for a cluster to ensure we have a reasonable statistical evidence that a particular solver works better than another. We sample 15% of the instances from each cluster to compose our training set, and to form the subsets of similar instances for generation.

We perform a more standard portfolio evaluation of the generator for SAT since so many instances are available. For this evaluation we use the top solvers from the 2013 SAT Competition: glucose, glue bit, lingeling 587f, lingeling aqw, MIPSat, riss3g, strangenight, zenn, CSHCapplLC, and CSHCapplLG.

Our test set for all the subsequent experiments is the collection of 254 industrial instances remaining after the subset of training instances is removed. This list is then further reduced by removing those instances for which no solver finds a solution within 1,800 seconds. This leaves a total of 195 instances. We compare the portfolios based on three metrics: average time without timeouts (Average), PAR10, and number of instances solved (Solved). PAR10 is a penalized average where a timeout counts as having taken 10 times the timeout time.

For our underlying portfolio technique, we utilized CSHC [23], the technique that won the "Open Track" at the 2013 SAT Competition and was behind the portfolio of ISAC++ [6] that won the MaxSAT Evaluation in 2013 and 2014. The core premise of this portfolio technique is a branching criteria for a tree that ensures that after

each partition, the instances assigned to sub-node maximally prefer a particular solver other than the one used in the parent node. Training a forest of such trees then ensures the robustness of the algorithm.
The results of the experiments are presented in Table 8.2. The best standalone solver is Lingeling aqw, which solves a total of 157 instances. We trained the portfolio on a variety of training sets: 300 random instances, 300 crafted instances, 46 industrial instances, 300 generated (industrial) instances and 1500 generated (industrial) instances. Not surprisingly, training on random or crafted instances does not perform well. In both cases, less instances can be solved than just using the best single solver, and the PAR10 scores are significantly higher. This further confirms a well-known result in the algorithm selection literature that training and test sets need to be similar in order for the learning algorithm to be successful. We include these results to emphasize the fact that if our generated instances were significantly different from the datasets they were generated from, we would expect similarly bad performance on the test set. Indeed, using even just 46 industrial instances already results in better performance than the best single solver in terms of average time, PAR10 and the number of instances solved.
Using 300 generated industrial instances shows similar performance to the original industrial instances in terms of the number of the PAR10 score and number of instances solved, although the average runtime is higher. This is already enough evidence to further confirm that our instance generation routine is successful at preserving instance structures in SAT, as in Max-SAT. However, because we are not limited by the number of instances we generate, we can create much larger training samples. We therefore evaluate our portfolio trained on 1,500 generated instances and observe that 166 instances can be solved, 5 more than with the original training set. In a competition setting, this improvement is often the difference between first place and finishing outside the top three solvers. This provides further support that our approach fills in gaps in the instance feature space, and that this provides critical information to selection algorithms that improves their performance.

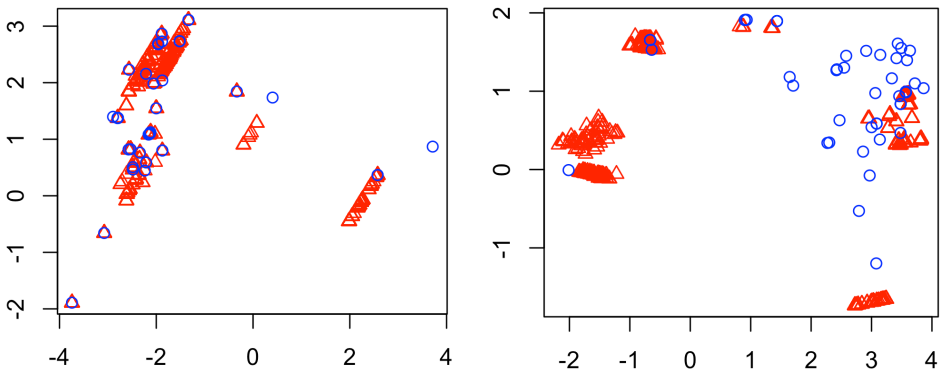
8.5.2.2 Q-score

The Q-score, introduced by Bayless et al. in [11], provides a mechanism for assessing whether or not a dataset of instances can act as a proxy for some other set of instances. In other words, using the Q-score we can check whether the instances we generate share similar properties with the original dataset of industrial instances. The score is based on the performance of parameters found through algorithm configuration using a method like [3] or [18]. We use SMAC [18] as it was previously used for calculating the Q-score in [11]. We configure the Lingeling and Spear solvers each three times for five days on the same 1500 generated instances used in our algorithm selection experiments and all 300 original industrial instances,

which we label S and T, respectively. Adopting the notation of [11] (which we refer to for full details), the Q-score is computed by $c(A(\theta'_T), T)/c(A(\theta'_S), T)$, where c is the PAR10 score of a parameterization on the specified dataset, A specified an algorithm configuration, and θ'_T and θ'_S are the best performing configurations (on the test set) of all tuned configurations and on the generated set, respectively.
We found the Q-score 0.9177 for lingeling and 0.9978 for Spear on our generated instances. We note that 1.0 is the best possible score. This indicates that the datasets we generate lead to high quality parameter configurations that generalize to the original instances. Interestingly the best parameter configuration for Lingeling on the test set was one of the parameterizations trained on the generated instances. However, its training set evaluation was beaten by another parameterization, thus we do not use it in the calculation of the Q-score for the set S. This is especially noteworthy given that our generated instance set is not even based on all of the industrial instances, but is nonetheless being compared to parameters specifically tuned on all 300 industrial instances.

8.5.3 Structure Comparison

As a final evaluation of our instance generation methodology we present a comparison of the original and generated instances when their features are projected into a two dimensional space. We do this using a standard principal component analysis (PCA). Figure 8.3 presents the results for both the MaxSAT and SAT datasets. The figure shows that there is not a perfect matching between the generated and original instances. While future work can focus on reducing the spread between these instances, we note that a perfect matching is not desirable as we do not want exact replicas of our instance pool. Instead, we want to cover a range of scenarios of similar instances, which can be seen in many parts of the projection. This subsequently leads to a better trained portfolio. Furthermore, note that the generated instances tend to be close to their original counterparts in this projected space. This means that although they are not completely identical, the generated instances are still fairly representative of their originals.



(a) MaxSAT **(b)** SAT

Figure 8.3 Projection of the instances into 2D using PCA on their features. Original training instances are represented as blue circles, the generated instances are represented by red triangles.

8.6 Conclusion and Future Work

One of the current key problems in solver development is the limited number of instances on which algorithms can be compared. This is especially the case for industrial instances, where datasets are extremely limited and difficult to expand. To remedy this, this paper presented a novel methodology for generating new instances with structures similar to a given dataset. We then demonstrated the quality of the generated datasets by training portfolios on them and evaluating them on the original instances. This showed that not only do the instances have similar structures as the originals, but that those structures also allow a portfolio (and algorithm configuration) to correctly learn the best solver for provided instances. For future work, will evaluate our instance generation framework on other types of problems, such as CSPs and MIPs, as well as explore how to improve the generated instances' coverage of the feature space.

Acknowledgements We thank the Paderborn Center for Parallel Computing for the use of the OCuLUS cluster for the experiments in this paper.

References

[1] C. Ansótegui, J. Giráldez-Cru, and J. Levy. "The Community Structure of SAT Formulas". In: *Theory and Applications of Satisfiability Testing (SAT 2012)*. Ed. by A. Cimatti and R. Sebastiani. LNCS 7317. Springer, Jan. 2012, pp. 410–423. ISBN: 978-3-642-31611-1, 978-3-642-31612-8.

[2] C. Ansótegui and J. Levy. "On the Modularity of Industrial SAT Instances". In: *CCIA*. Ed. by C. F., H. Geffner, and F. Manyà. Vol. 232. FAIA. IOS Press, 2011, pp. 11–20. ISBN: 978-1-60750-841-0.

[3] C. Ansotegui, M. Sellmann, and K. Tierney. "A Gender-Based Genetic Algorithm for the Automatic Configuration of Algorithms". In: *Principles and Practice of Constraint Programming (CP-09)*. Ed. by I.P. Gent. Vol. 5732. LNCS. Springer, 2009, pp. 142–157.

[4] C. Ansótegui et al. "Analysis and Generation of Pseudo-Industrial MaxSAT Instances". In: *CCIA*. Vol. 248. FAIA. IOS Press, 2012, pp. 173–184. ISBN: 978-1-61499-138-0.

[5] C. Ansótegui et al. "Edge Matching Puzzles as Hard SAT/CSP Benchmarks". In: *CP*. Vol. 5202. LNCS. 2008, pp. 560–565.

[6] Carlos Ansotegui, Yuri Malitsky, and Meinolf Sellmann. "MaxSAT by Improved Instance-Specific Algorithm Configuration". In: *AAAI* (2014).

[7] J. Argelich et al. *Eighth Max-SAT Evaluation*. 2013.

[8] A. Balint et al. *Proceedings of SAT Competition 2013; Solver and benchmark descriptions*. Tech. rep. University of Helsinki, 2013.

[9] R. Barták. "On Generators of Random Quasigroup Problems". In: *Recent Advances in Constraints*. Vol. 3978. LNCS. Springer, 2006, pp. 164–178. DOI: 10.1007/11754602_12.

[10] M. Bastian, S. Heymann, and M. Jacomy. "Gephi: An Open Source Software for Exploring and Manipulating Networks". In: *AAAI Conference on Weblogs and Social Media*. 2009.

[11] S. Bayless, D.A.D. Tompkins, and H.H. Hoos. "Evaluating Instance Generators by Configuration". In: *LION 8*. Ed. by Panos M. Pardalos et al. Vol. 8426. LNCS. Springer, 2014, pp. 47–61. ISBN: 978-3-319-09583-7. DOI: 10.1007/978-3-319-09584-4.

[12] R. Bejar et al. "Generating Hard Instances for MaxSAT". In: *International Symposium on Multiple-Valued Logic (ISMVL 2009)*. May 2009, pp. 191–195. DOI: 10.1109/ISMVL.2009.58.

[13] S. Burg, S. Kottler, and M. Kaufmann. "Creating industrial-like SAT instances by clustering and reconstruction". In: *SAT 2012*. Springer, 2012, pp. 471–472.

[14] I. Dinur, S. Goldwasser, and H. Lin. "The computational benefit of correlated instances". In: *Proceedings of the 2015 Conference on Innovations in Theoretical Computer Science*. ACM. 2015, pp. 219–228.

[15] I. P. Gent et al. "Morphing: Combining Structure and Randomness". In: *AAAI*. Ed. by J. Hendler and D. Subramanian. 1999, pp. 654–660. ISBN: 0-262-51106-1.

[16] C. P. Gomes and B. Selman. "Problem Structure in the Presence of Perturbations". In: *AAAI*. Ed. by B. Kuipers and B. L. Webber. 1997, pp. 221–226. ISBN: 0-262-51095-2.

[17] Greg Hamerly and Charles Elkan. "Learning the K in K-Means". In: *Neural Information Processing Systems (NIPS)*. 2003.

[18] F. Hutter, H.H. Hoos, and K. Leyton-Brown. "Sequential model-based optimization for general algorithm configuration". In: *LION 5*. Springer, 2011, pp. 507–523.

[19] S. Kadioglu et al. "ISAC – Instance-Specific Algorithm Configuration". In: *ECAI*. Vol. 215. FAIA. IOS Press, 2010, pp. 751–756.

[20] G. Katsirelos and L. Simon. "Eigenvector Centrality in Industrial SAT Instances". In: *CP*. Ed. by M. Milano. Vol. 7514. LNCS. Springer, 2012, pp. 348–356. ISBN: 978-3-642-33557-0.

[21] F. Krzakala and L. Zdeborová. "Hiding quiet solutions in random constraint satisfaction problems". In: *Physical review letters* 102.23 (2009), p. 238701.

[22] L. Lopes and K. Smith-Miles. "Generating applicable synthetic instances for branch problems". In: *Operations Research* 61.3 (2013), pp. 563–577.

[23] Yuri Malitsky et al. "Algorithm Portfolios Based on Cost-Sensitive Hierarchical Clustering". In: *IJCAI* (2013).

[24] M. Motoki. "Test Instance Generation for MAX 2SAT". In: *CP05*. Ed. by P. van Beek. LNCS 3709. Springer, Jan. 2005, pp. 787–791. ISBN: 978-3-540-29238-8, 978-3-540-32050-0.

[25] S. Nallaperuma, M. Wagner, and F. Neumann. "Parameter Prediction Based on Features of Evolved Instances for Ant Colony Optimization and the Traveling Salesperson Problem". In: *PPSN XIII*. Vol. 8672. LNCS. Springer, 2014, pp. 100–109. ISBN: 978-3-319-10761-5.

[26] E. Nudelman et al. "Understanding random SAT: Beyond the clauses-to-variables ratio". In: *CP 2004*. Springer, 2004, pp. 438–452.

[27] P. R. Pari et al. "Generating 'Random' 3-SAT Instances with Specific Solution Space Structure". In: *AAAI*. Ed. by D. L. McGuinness and G. Ferguson. 2004, pp. 960–961. ISBN: 0-262-51183-5.

[28] P. Shaw. "Using constraint programming and local search methods to solve vehicle routing problems". In: *CP 1998*. Springer, 1998, pp. 417–431.

[29] A. Slater. "Modelling More Realistic SAT Problems". In: *Australian Joint Conference on AI*. Ed. by B. McKay and J. K. Slaney. Vol. 2557. LNCS. Springer, 2002, pp. 591–602. ISBN: 3-540-00197-2.

[30] K. Smith-Miles and S. Bowly. "Generating new test instances by evolving in instance space". In: *Computers & Operations Research* 63 (2015), pp. 102–113. ISSN: 0305-0548. DOI: http://dx.doi.org/10.1016/j.cor.2015.04.022.

[31] K. Smith-Miles and J. van Hemert. "Discovering the suitability of optimisation algorithms by learning from evolved instances". In: *Ann Math Artif Intell* 61.2 (2011), pp. 87–104. ISSN: 1012-2443.

[32] A. Van Gelder and I. Spence. "Zero-One Designs Produce Small Hard SAT Instances". In: *SAT 2010*. Ed. by O. Strichman and S. Szeider. LNCS 6175. Springer, Jan. 2010, pp. 388–397. ISBN: 978-3-642-14185-0, 978-3-642-14186-7. (Visited on 01/31/2014).

Chapter 9

Algorithm Configuration Applied to Heuristics for Three-Dimensional Knapsack Problems in Air Cargo

Marius Merschformann[1]

[1] *University of Paderborn, Paderborn, Germany*
marius.merschformann@upb.de

Algorithm Configuration: Papers from the 2015 AAAI Workshop

Abstract

The problem of efficiently packing items into containers is of great importance in the air cargo industry. Hence, the algorithms used to solve the corresponding problem should also be efficient, including their configurations. We present an algorithm configuration scenario using a state-of-the-art algorithm from this area.

9.1 Introduction

Efficiently packing items into a set of containers is a basic process in many different applications. Especially in air cargo logistics, storage volume is expensive and the handling times at cargo hubs are short. Thus, there is demand for a decision support system capable of generating efficient so-called "build-up plans" that can be quantified and immediately executed, instead of planning the load allocation manually. As a special extension, more complex forms are handled through an approximation using "Tetris"-shapes (see Figure 9.1 and [2]). The resulting problem is called three-dimensional Tetris Multiple Heterogeneous Knapsack Problem according to [5]. The algorithms of this paper aim to generate such plans to solve such problems in a reasonable amount of time.
The problem can be decomposed into three decisions. First, items must be allocated to available containers. Every item may be assigned to at most one suitable container. Secondly, the positions of items within containers must be determined. Finally, the orientation of the item is chosen from 24 distinctive 90° rotations for "Tetris"-shapes. These decisions are bounded by certain basic requirements like the non-overlapping of items and containers and more application specific ones like forbidden orientations and incompatibilities when handling dangerous goods. The objective of this problem is to utilize the available space as effectively as possible, similar to the classic knapsack problem.
We present an algorithm configuration scenario using SMAC [3] to configure the parameters of a GRASP-like Greedy Adaptive Search Procedure (GASP) that solves the packing problem at hand.
A valid solution to this problem may look like the one depicted in Figure 9.2.

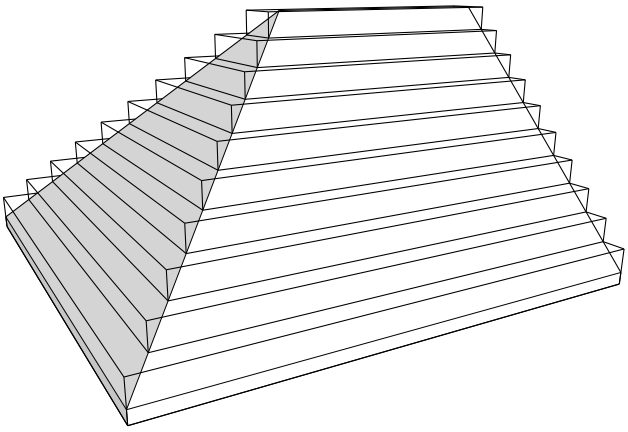


Figure 9.1 "Tetris"-approximation.

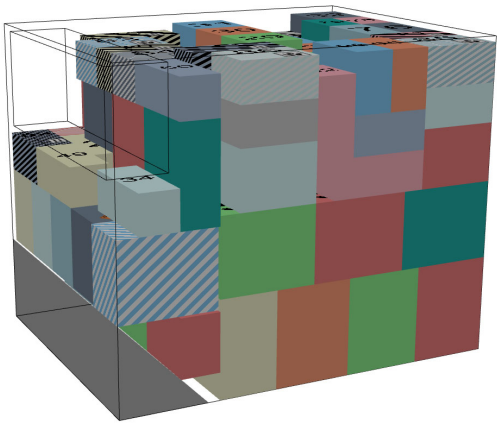


Figure 9.2 Build-up plan for one container.

9.2 Solution approach

In order to solve the previously described problem, researchers have created several heuristic construction algorithms based on GASP [4]. In this approach solutions are iteratively generated in parallel threads by inserting the items in an order corresponding to a score per item. This score is updated in each iteration. Three different techniques are used, in which two of them are also capable of handling "Tetris"-items. Since the handling of such items alters the behavior of the algorithm significantly, those techniques are split into two separate algorithms for tuning. The first technique is called Extreme Point Insertion (EPI) and uses so-called Extreme Points (EP) (see [1]) at which new items are positioned. The list of available EPs is updated every time a new item is inserted. EPI-t refers to the "Tetris" variant of this approach. The second technique is called Push Insertion (PI), which uses fixed insertion points to initially allocate the items and subsequently pushes them towards the container's origin. PI-t refers to the "Tetris" variant of this approach. The last technique is called Space Defragmentation (SD) (see [6]) and is an extension of the first approach. In addition to using EPs for insertion the technique focuses on increasing the density of the packing by pushing items away from newly allocated ones and subsequently all of them towards the containers origin. Thus, the items change their position constantly which makes an efficient integration of "Tetris" items difficult.

9.3 Algorithm configuration approach

The heuristic approaches described have a number of configurable parameters. We use the SMAC algorithm proposed by [3]. This system utilizes random forests, a standard machine learning technique for regression and classification. We perform configuration for the EPI, EPI-t, SD, PI and PI-t methods. While the basic algorithms are tuned on 100 instances only containing cuboid items, the "Tetris"-variants are tuned on 100 instances which also contain "Tetris"-shaped items. Both of the sets are split into a training set of 60 instances and a test set of 40 instances.

We tune each of the five algorithms for five days. Each single execution is limited with a five minute cutoff. The objective is set to solution quality, instead of time, because the time available for the solution process does not vary in the application field. The objective function value of the solution is given by the utilized volume of the containers. The objectives are combined by the mean metric of SMAC. Also, due to the parallelization of the algorithms, determinism can not be guaranteed. We describe the parameters of the algorithms along with their domains as follows. Parameters not used by a specific algorithm are ignored for the corresponding run.

- *ItemOrder*: The initial order of the items when inserting.
  $[0, \ldots, 23] \subset \mathbb{Z}$, (EPI, EPI-t, PI, PI-t, SD)

- *BestFit*: A boolean parameter defining whether to use a merit-function or not. If the value is false, every item is inserted at the first valid insertion point, otherwise the best available one is used as indicated by the merit-function.
  $\{true, false\}$, (EPI, EPI-t)

- *MeritType*: A conditional parameter identifying the specific merit-function to use, if *BestFit* is activated.
  $[0, \ldots, 6] \subset \mathbb{Z}$, (EPI, EPI-t)

- *InflateAndReplaceInsertion*: Defines whether to use or skip the inflate-and-replace strategy of the SD technique.
  $\{true, false\}$, (SD)

- *NormalizationOrder*: The order by which the container is normalized, i.e. items are pushed to the corresponding directions consequentially until no further push is possible.
  {xyz,zyx,zxy,yzx,xzy,yxz}, (PI, PI-t, SD)

- *ScoreBasedOrder*: Deactivates the score-based sorting of items and substitutes it with a complete random approach.
  $\{true, false\}$, (EPI, EPI-t, PI, PI-t, SD)

- i^{RD}: The maximal distance without an improvement on the objective value before the score is reinitialized.
  $[50, \ldots, 2500] \subset \mathbb{Z}$, (EPI, EPI-t, PI, PI-t, SD)

- r^S: A "salt"-value applied randomly for every item, hence, increasing this value increases a certain random influence on the method.
  $[0, \ldots, 0.9] \subset \mathbb{R}$, (EPI, EPI-t, PI, PI-t, SD)

- m^I: The initial score modification.
  $[0.01, \ldots, 1] \subset \mathbb{R}$ (EPI, EPI-t, PI, PI-t, SD)

- m^{max}: The maximum score modification
  $[1, \ldots, 5] \subset \mathbb{R}$, (EPI, EPI-t, PI, PI-t, SD)

- s^P: The number of minimal swaps
  $[1, \ldots, 4] \subset \mathbb{Z}$, (EPI, EPI-t, PI, PI-t, SD)

- s^{max}: The number of maximal swaps
  $[4, \ldots, 8] \subset \mathbb{Z}$, (EPI, EPI-t, PI, PI-t, SD)

9.4 Computational results

Overall, the configuring of the parameters did not obtain significantly improved configurations. As depicted by the following graphs for the EPI algorithm, the tuning has a slight positive (but also negative impact) depending on the particular instances. This can be seen by comparing the result of the default parameters depicted on the x-axis with the tuned configuration depicted on the y-axis. The respective axes identify the obtained evaluation values (relative volume utilization) by the two configurations. For both the training set (Figure 9.3) and the test set (Figure 9.4) only a very slight positive trend is seen. The results for the other algorithms are very similar. Thus, the trend is too small to verify a positive effect of the configuration. This might be due to the very heterogeneous instances requiring different algorithm settings. Hence, in future it may be useful to integrate instance features to find better configurations depending on those. The result would be an automatic algorithm configuration routine before solving the respective instance exploiting the results of the tuning.

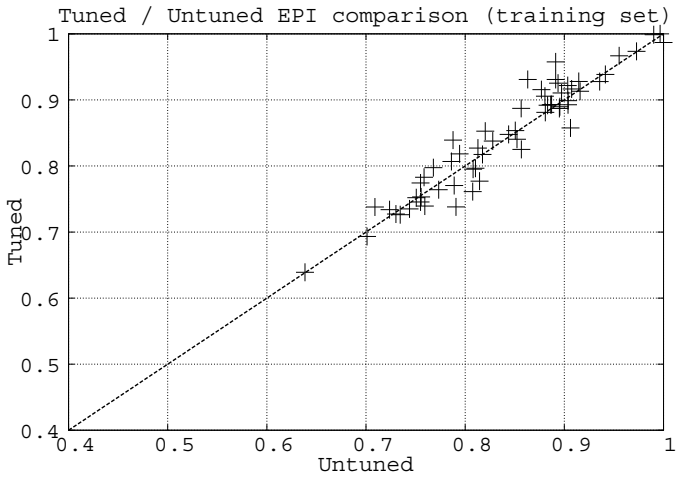


Figure 9.3 Results for EPI (training set)

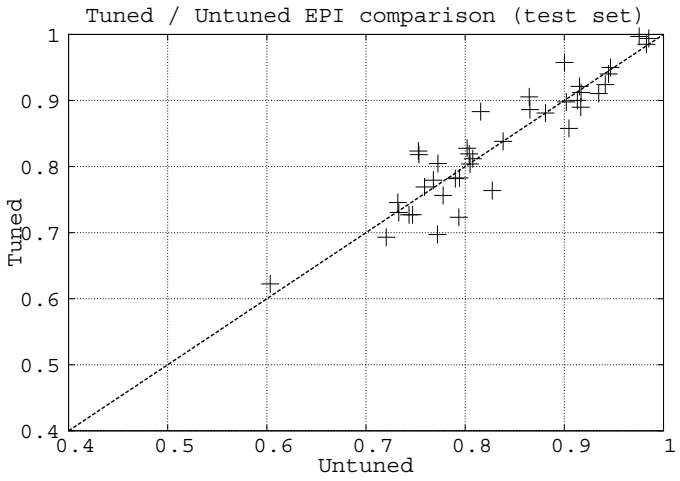


Figure 9.4 Results for EPI (test set)

9.5 Conclusion

We presented an algorithm configuration scenario for three dimensional knapsack problems in the air-cargo industry. Although the results of the configuration were not significantly better than the default configuration, there is hope that other strategies for configuring these algorithms could provide better performance.

References

[1] T. G. Crainic, G. Perboli, and R. Tadei. "Extreme Point-Based Heuristics for Three-Dimensional Bin Packing". In: *INFORMS Journal on Computing* 20.3 (2008), pp. 368–384. DOI: 10.1287/ijoc.1070.0250. eprint: http://pubsonline.informs.org/doi/pdf/10.1287/ijoc.1070.0250. URL: http://pubsonline.informs.org/doi/abs/10.1287/ijoc.1070.0250.

[2] G. Fasano. "A global optimization point of view to handle non-standard object packing problems". English. In: *Journal of Global Optimization* 55.2 (2013), pp. 279–299. ISSN: 0925-5001. DOI: 10.1007/s10898-012-9865-8. URL: http://dx.doi.org/10.1007/s10898-012-9865-8.

[3] F. Hutter, H. H. Hoos, and K. Leyton-Brown. "Sequential Model-Based Optimization for General Algorithm Configuration". In: *Proceedings of LION-5*. 2011, pp. 507–523.

[4] G. Perboli, T. G. Crainic, and R. Tadei. "An efficient metaheuristic for multi-dimensional multi-container packing". In: *Automation Science and Engineering (CASE), 2011 IEEE Conference on*. 2011, pp. 563–568. DOI: 10.1109/CASE.2011.6042476.

[5] G. Wäscher, H. Haußner, and H. Schumann. "An improved typology of cutting and packing problems". In: *European Journal of Operational Research* 183.3 (2007), pp. 1109–1130. ISSN: 0377-2217. DOI: http://dx.doi.org/10.1016/j.ejor.2005.12.047. URL: http://www.sciencedirect.com/science/article/pii/S037722170600292X.

[6] W. Zhu et al. "Space defragmentation for packing problems". In: *European Journal of Operational Research* 222.3 (2012), pp. 452–463. ISSN: 0377-2217. DOI: http://dx.doi.org/10.1016/j.ejor.2012.05.031. URL: http://www.sciencedirect.com/science/article/pii/S037722171200389X.

Chapter 10

Metaheuristics approach for solving personalized crew rostering problem in public bus transit

Lin Xie[1] Marius Merschformann[2] Natalia Kliewer[3] Leena Suhl[2]

[1] *Leuphana University of Lüneburg, Lüneburg, Germany*
[2] *University of Paderborn, Paderborn, Germany*
[3] *Freie Universität Berlin, Berlin, Germany*
lin.xie@leuphana.de, marius.merschformann@upb.de, natalia.kliewer@fu-berlin.de, leena.suhl@upb.de

Journal of Heuristics (2017), Volume 23, Issue 5, pp 321–347

Abstract

The crew rostering problem in public bus transit aims at constructing personalized monthly schedules for all drivers. This problem is often formulated as a multi-objective optimization problem, since it considers the interests of both the management of bus companies and the drivers. Therefore, this paper attempts to solve the multi-objective crew rostering problem with the weighted sum of all objectives using ant colony optimization, simulated annealing, and tabu search methods. To the best of our knowledge, this is the first paper that attempts to solve the personalized crew rostering problem in public transit using different metaheuristics, especially the ant

colony optimization. The developed algorithms are tested on numerical real-world instances, and the results are compared with ones solved by commercial solvers.

10.1 Introduction

In public bus transit, the labor costs for the drivers and other personnel represent a significant portion of bus operators' budgets (more than 50%). The drivers are considered as one of the most important resources, therefore more attention should be paid to the optimization process of generating schedules for the drivers. The process of generating schedules for drivers is normally divided into two sequential steps, i.e. first, crew scheduling, and second, crew rostering.
The *crew scheduling problem* aims at generating anonymous duties while some work regulations have to be satisfied, such as maximum duration of a duty and minimal break times during a duty. A *duty* is a sequence of tasks within one day that is performed by a driver who leaves and returns to the same depot in accordance with the work regulations. The generated duties and some other activities are assigned to drivers in the *crew rostering problem*, while following the complex law and labor union rules, such as the minimum rest period between two duties or the maximum length of consecutive working days. The preferences of drivers and companies are also considered during the optimization. The generated schedule for each driver is called a *roster*. Optimal rosters are characterized by maximum satisfaction of drivers and minimal operational costs.
The generated duties in the crew scheduling problem does not include any information about drivers (anonymous duties). Therefore, it causes difficulties generating schedules of drivers: for example, some duties cannot be covered, while some drivers do not get any jobs on some days. The minimization of costs is still important in the crew rostering problem, and the preferences of drivers are considered during the optimization as well. The rosters which are generated by considering preferences of drivers bring higher acceptance than rosters that ignore individual wishes (see [17]), which might cause fewer exchanges and less absence in operational days. Therefore, fewer recovery activities are expected, which implies lower operational costs, and better services are expected. Therefore, the crew rostering problem is a multi-objective optimization problem, since it considers the interests of both the management and the drivers. However, considering both interests increases the difficulties in solving this problem. Thus, there are some publications which concentrate on sub-problems of personalized crew rostering (see [5], [3], [6], [20], and [26]). The definition of sub-problems can be found in Section 10.2.3.
Still, there are some heuristics or exact approaches that have been published in the last ten years to solve the personalized crew rostering problem in public transit; however, they are only applied to small test instances. A Strength Pareto Evolution-

ary Algorithm (SPEA2) is applied in [25], which aims at minimizing the maximum overtime for drivers while minimizing the number of drivers. The instances were designed to cope with the case of a bus transport company in Portugal, which considers 45 drivers within four weeks. Based on the work of [25], a Strength Pareto Utopic and Memetic Algorithm (SPUMA) is adopted in [28], which was proved to outperform the result of SPEA2. A column generation approach is developed for urban transit crew rostering in [33], which aims at minimizing the number of crews for the given periods. This approach is based on real-world instances with up to 40 crews; however, the planning horizon is limited to one week. In [23], a MIP model is developed for solving crew rostering, which aims at the same objectives as in [25]. This bi-objective problem is solved by the goal programming approach, and the solving instances are limited to 1,300 duties and about 70 drivers.

Until last year a MIP model based on the multicommodity flow network was implemented for different crew rostering problems in [32], including personalized crew rostering. A set of real-world instances from different German bus companies were tested; however, some large real-world instances do not generate feasible or good solutions within the given running times (24 hours). Therefore, the goal of this paper is to extend implementations in [32] with a novel comparison of state-of-the-art metaheuristics. That includes ant colony metaheuristics, simulated annealing, and tabu search. Existing applications of ant colony optimization algorithms for solving train and airline crew scheduling problems can be found in [21] and [18]. In [22], simulated annealing and tabu search are applied for solving the air crew rostering problem. As described in [2], the sizes of real-world instances of the crew rostering problem in public bus transit are larger than those for the air/rail rostering problem. A duty (known as pairing) in air/rail rostering can span up to several days, while a duty in public bus transit is a daily job for a driver. This creates a large number of duties to be considered in rosters in public bus transit.

The rest of the paper is structured as follows. In Section 10.2, the crew rostering problem is introduced, followed by a brief description of the implemented metaheuristics in Section 10.3. Section 10.4 shows the experimental results. Finally, Section 10.5 concludes this paper.

10.2 Problem description

In this section the necessary information of crew rostering is described, which includes the problem input in Section 10.2.1, network design in Section 10.2.2, and a short description of the sequential and integrated approaches in Section 10.2.3.

10.2.1 Problem input

A roster is considered as a schedule of combinations of work-related activities $\mathcal{D}_w$ (shift/duty $\mathcal{D}_s \subseteq \mathcal{D}_w$, standby $\mathcal{D}_b \subseteq \mathcal{D}_w$), day off activities $\mathcal{D}_f$ (single-off $\mathcal{D}_{sf} \subseteq \mathcal{D}_f$, double-off $\mathcal{D}_{df} \subseteq \mathcal{D}_f$), and others (such as leaves and training periods) within a given time period $\mathcal{T}$ (such as one month). A duty activity $d \in \mathcal{D}_s$ is defined as the daily job for a driver, which includes a start time, an end time, the duration, a depot (where the driver begins and ends), a shift type (for example, one duty beginning between 3 am and 6 am is an early shift), and the calendar date that the duty belongs to. *Standby* activities are planned to cover the absences of drivers, while *days off* consist of a couple of rest days between working days. Days off include a *single off* (a single day off between two work-related activities) and *double off* (at least two consecutive days off). *Leaves* are defined as vacations. The training periods and leaves are preassigned and fixed for each driver and cannot be changed in the optimization.

Also, in the personalized crew rostering problem each roster is generated for a driver $m \in \mathcal{M}$. The data about drivers includes not only the depot (where the driver begins and ends) and vehicle types (depending on the capacity, speed, or equipment of each vehicle), but also the number of days off F_m, the target working days T_d and working hours R_m for the current planning period, the target number of standby activities V_m, and the required training periods and leaves. All of these depend on the work contracts and the drivers's current work-accounts. That work-account of each driver includes their current overtime and their number of days off from previous periods. It describes the driver's credit, e.g. a driver with more overtime in previous periods can get more jobs with shorter working time to reduce overtime gradually. Moreover, the driver can express their preferences, including their daily desired activities and their possible combination of activities. The daily desires of a driver mean that the driver wishes to get one activity on one day, while the possible combination of activities can be, for instance, similar shifts within a working week, and/or not to get an early shift after a day off.

During the generation of rosters, a set of rules and regulations should be considered, including horizontal rules, vertical rules, and quality rules. Horizontal rules are rules that depend only on one roster, while vertical rules combine the information among all rosters. *Horizontal rules* consist of compatibility, working block, and days off block. The incompatible connections of activities are gathered in a list of forbidden sequences according to the working regulation. For example, it is forbidden to follow an early duty with a late duty because of the short rest time between them. The forbidden sequences are provided by the bus companies. The length of the forbidden sequences is not limited to two, but the forbidden sequences with the length two and the definition of single off and double off can be implicitly considered during the network generation (see Section 10.2.2). Moreover, the maximum consecutive

number of working days L^W and days off L^R (working/days off block) are stricted based on working regulations. For example, the number of consecutive working days is limited to five, while the number of consecutive days off is restricted to three. *Vertical rules* consider restricted resources among all rosters. Not only are the limits on the assignment of the amount of a duty/shift during each day considered, but also the limits on the available numbers of days off and duties. *Quality rules* affect the quality of the rosters, but without them the generated rosters are still legal. These rules include preferences of drivers, real costs, and fairness among all drivers. As mentioned before, the daily desired activities and the possible combination of activities are given by drivers. Moreover, due to the limited number of drivers that may simultaneously take a day off, some desired days off (on weekends) cannot be satisfied, but can be moved to within a couple of days earlier or later. There are called moved days off (on weekends). However, the number of them is minimized due to their unpopularity. Similarly, the number of single-off activities is limited to E_m for the driver m due to unpopularity. The distance between two double-off acitivities D is defined in the company rules to check whether a driver gets at least two double-offs within $D + 4$ days. If that is not the case, then the number of such violations should be minimized. The insufficient assignment of standbys and duties/shifts might cause additional personnel costs or additional costs for the overtimes of all planned drivers. The same is true for open days, on which drivers do not get any activities. Therefore, the number of unassigned standbys and duties/shifts should be minimized and the open days should be minimized as well. An additional situation that should be avoided is one where, for example, some drivers work overtime while others work substantially less than the regular working time. This can be avoided by minimizing the maximum overtime for all drivers, which results in less payment for overtime. In the mathematical model of [32], the horizontal and vertical rules are formulated as hard constraints while the quality rules are formulated as soft constraints. A counter is utilized for the violation of each quality rule and each of the violation is punished with a different penalty cost in the objective function.

10.2.2 Network design

The personalized crew rostering problem is formulated in this paper as a multicommodity flow network as in [32], where a network layer represents the valid activities for a driver m, and possible connections between them. On the network layer of the driver m, only the possible activities $d \in \mathcal{D}$ on day $t \in \mathcal{T}$ are illustrated as activity arcs $e \in \mathcal{A}_{m,t,d}$, whose cost C_e is defined to reflect the daily desire of the driver m. We also use nodes in this paper to illustrate the activity arcs for better illustration. Moreover, only the compatible activity arcs are connected by compatibility arcs $e \in \mathcal{E}$. That means that the forbidden sequences with length two and the definitions of single off and double off can be considered during the generation of the network.

Also, the cost C_e of each compatibility arc $e \in \mathcal{E}$ is used to reflect the combination of desires of activities for the driver m. The less-desired activity arcs and less-desired compatibility arcs have higher costs. In order to ensure the network's feasibility, flow conservation, demand satisfaction, and flow capacity constraints are required. More details about the network design can be found in [32].

10.2.3 Sequential vs. integrated approach

Due to the high complexity of the crew rostering problem, it is usually divided into several sequential sub-problems (see [32]). The sub-problems are *days off scheduling*, *shift assignment*, and *duty sequencing*, which will assign days off to drivers by considering fair distribution of days off, assign shifts to drivers, and assign duties to drivers based on the result of shift assignment, respectively. The problem of integrated days off scheduling and shift assignment is named *rota scheduling* in [12] and [31]. Metaheuristics, such as simulated annealing, tabu search, and ejection-chains, are applied to solve days off scheduling and shift scheduling in [20] and [26]. The integrated approach means that all activities (days off, duties, standbys, leaves, training periods) are assigned to drivers without any intermediate steps, while considering horizontal, vertical, and quality rules.
In [32] several examples are shown to demonstrate the drawbacks of the sequential approach compared with the integrated one. Moreover, according to [32], the integrated problem can be easily extended by the sub-problem *rota scheduling*, in both the network design and the mathematical model. For simplicity's sake, we develop different metaheuristics firstly for rota scheduling in Section 10.3. Then, the best metaheuristic is chosen to be applied to solve the intergrated problem in Section 10.4.

10.3 Algorithm design

10.3.1 Information about heuristic

First of all, we will show the method of updating meta-information (Section 10.3.1.1), the description of different local search operators (Section 10.3.1.2), and the procedure for generating an initial solution (Section 10.3.1.3).

10.3.1.1 Update of meta-information

Meta-information includes the actual information about the horizontal and vertical rules, such as block length, forbidden sequences, and capacity of activities, as well as the counters of all quality rules. Additionally, the costs of the personal preferences of the selected activities and actual overtimes are stored. Since each quality rule can

be managed with a counter or a storage value, they will not be explained in more detail. The capacity of each activity is updated for all drivers on each calendar date. For better understanding, the updates of block lengths and forbidden sequences will be described with examples as follows. Note that, the update of meta-information occurs not only for the generation of a feasible initial solution but also for the δ-evaluation using local search operators.
Figure 10.1 shows two examples of tracking block lengths of work-related and days-off activities. For each node (activity), the accumulated lengths of consecutive work-related (E, M, and L lettered on nodes mean an early, a midday, and a late duty, respectively) and days-off activities (1 and 2 labelled on nodes means a single off and one day of double off, respectively) are stored during the generation process (see left-hand side of Figure 10.1), while they are updated for each switch during backtracking or local search in the right-hand side of Figure 10.1.

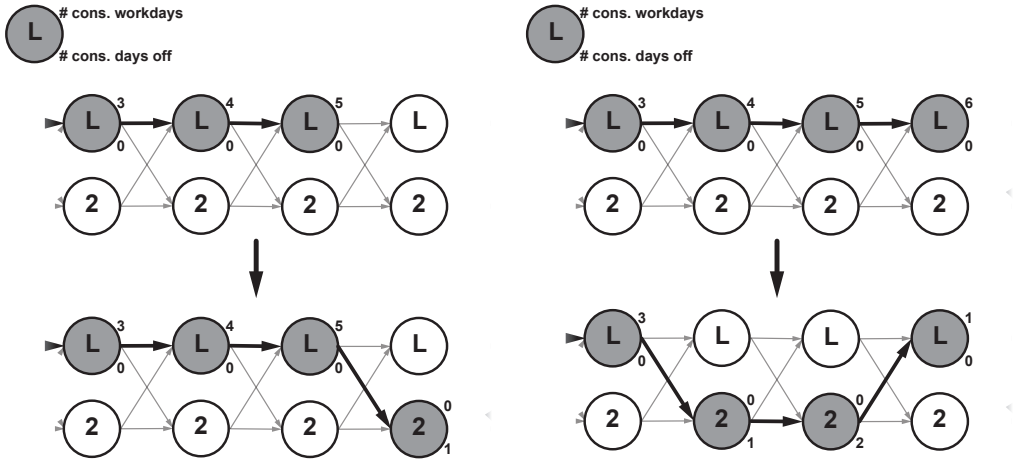


Figure 10.1 Update of block length during the generation of a solution (left) and during a switch in local search operator or backtracking (right).

We build all forbidden sequences as branches of trees, in order to avoid unnecessary searches for all forbidden sequences during the generation of a solution or when switching activities. In the example shown in Figure 10.2, three forbidden sequences are considered, which are illustrated as branches of trees, i.e. (L, E), (L, M, E), and (E, L). A check on the validity of adding an activity can be done by checking its corresponding node in the tree up to the root nodes and down to all leaves, if necessary. The reference between each activity and itself in each forbidden tree is attached during the generation of a solution. Each reference is illustrated by a dotted line. In this example, if the early duty (E) is added to the actual solution, then this duty is attached to the leaf of the tree (by a dotted line). That means this early duty is forbidden to be added to the actual solution, since the late and midday duties

on the previous two days are selected sequentially (the forbidden sequence (L, M, E)). In this case, the check for all forbidden sequences can be avoided. For the same reason, the structure of the tree can speed up the validation of switching activities. When backtracking occurs, the state of the previous activity can be restored by saving the related nodes of the trees to the activities.

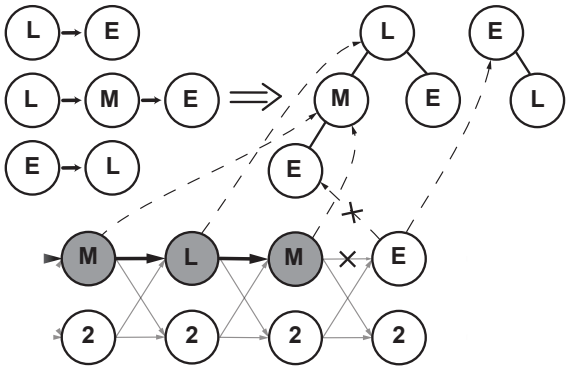


Figure 10.2 Tracking of forbidden sequences during generation of a solution.

10.3.1.2 Local search operators

In this subsection we introduces the different operators that will be used by local search-based methods, i.e. the simulated annealing and tabu search algorithms in Sections 10.3.3 and 10.3.4. In [22], a simple operator is applied to exchange duties between two crews. However, we develop different operators based on the properties of our crew rostering problem, such as the Crawl operator, which is defined to cope with the definition of double-offs. It is worth noting that each switch should satisfy all hard constraints to generate a feasible solution.

The *Simple-Switch* operator is the most primitive operator in terms of the changes made to the current solution. With this operator, usually only one activity for a specific driver on a specific day is substituted by a new activity. The new activity has to satisfy all hard constraints; this is checked before the switch is performed. However, one special case has to be coped with here. Since a double-off activity has to be linked to at least one other double-off activity, exchanging one of them would obviously lead to invalid solutions. If this case occurs, the two double-off activities are exchanged with the other two activities. This case is illustrated in Figure 10.3, where double-offs on days 10 and 11 are exchanged with an early duty and a single-off, respectively.

The *Multi-Switch* operator performs a limited number of Simple-Switches in a row, while the affected days are consecutive. An example of exchanges with a depth of

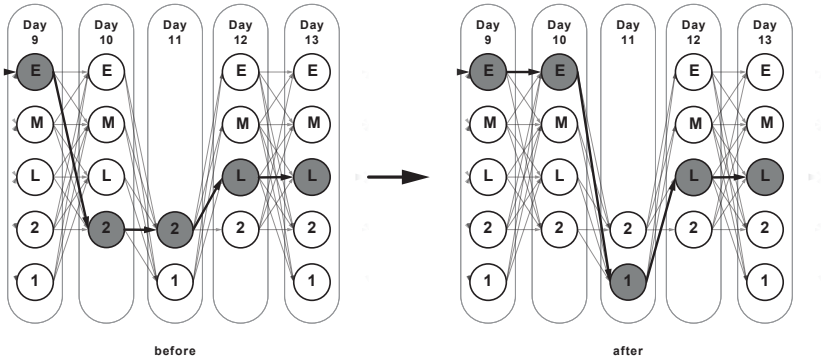


Figure 10.3 An example of the Simple-Switch operator performing a two-day exchange.

three is shown in Figure 10.4. It begins on day 9, and a Simple-Switch has occurred. On days 10 and 11 the same Simple-Switch involves two double-offs; this is shown as in Figure 10.3. After that, a Simple-Switch is applied on day 12. All possible exchanges are tested, and the best is chosen. This operator is implemented to allow bigger leaps when modifying a solution.

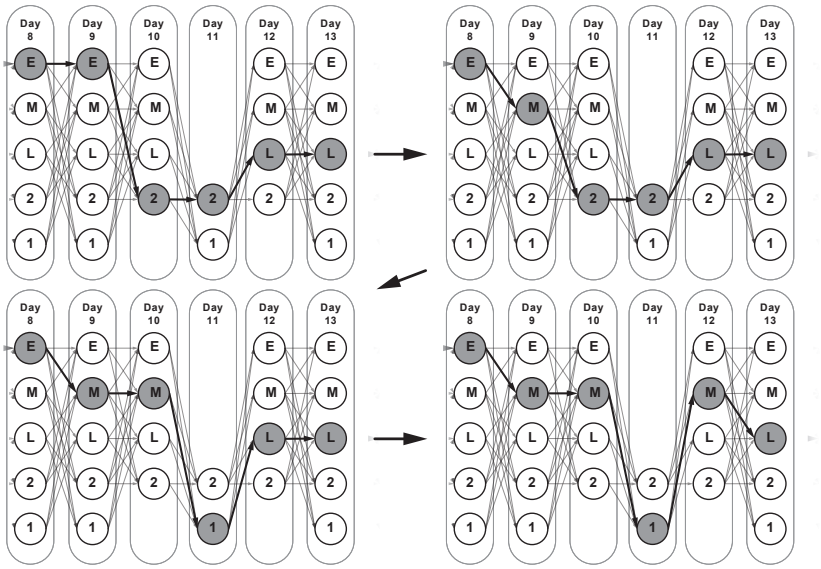


Figure 10.4 An example of the Multi-Switch operator.

The *Crawl* operator is used to shift a sequence of double-offs to the left or the right. This is required because removing a sequence of double-offs often leads to a violation of the given distance between two subsequent double-offs. Therefore, it causes penalty costs in the evaluation value. For that reason, the exchange should be

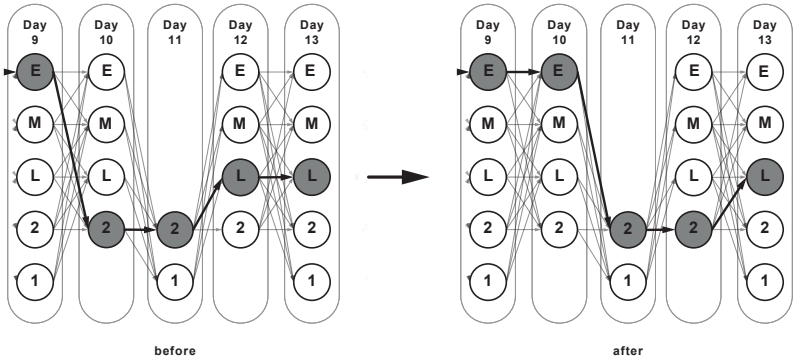


Figure 10.5 An example of the Crawl operator.

removed in the local search. To overcome this, the complete sequence of double-offs is moved by using the Crawl operator. Figure 10.5 illustrates an example where two double-offs on days 10 and 11 are shifted to days 11 and 12. Technically, the Crawl operator can be considered as two Simple-Switches. A Simple-Switch is applied on day 10 to change double-off to another activity, while another Simple-Switch is applied to change the activity on day 12 to a double-off.

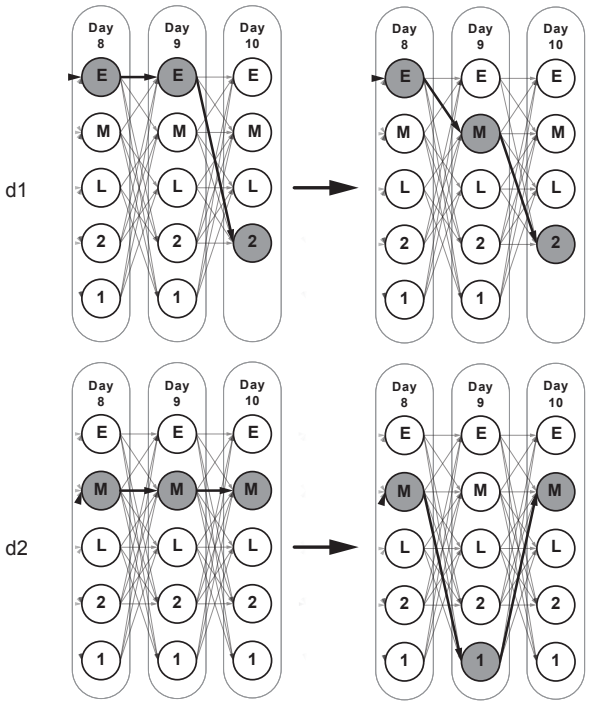


Figure 10.6 An example of the Duty-Switch operator.

The last operator that is implemented is the *Duty-Switch*. This operator is used to move a selected duty from one driver to another. It works as follows, with an example shown in Figure 10.6. Firstly, an early duty (E) for a driver $d1$ on day 9 is substituted with a randomly selected valid activity (in this case: a midday duty M). Secondly, the selected duty on day 9 (M) for the second driver ($d2$) is changed to another activity. The aims of this operator could include, for example, reducing the request for early duties on day 9 without changing the actual assigned amount for the midday duty, since the early duties on day 9 are over-requested.

10.3.1.3 Initial solution

An initial solution $\mathcal{X}$ can be achieved by a construction heuristic *AntWalk* with *BacktrackRepair* (see Algorithms 10.1 and 10.2). $\mathcal{X}$ is defined as a set of all activity arcs in a feasible solution for all drivers. An artificial ant walks through the randomly ordered network layers and stops when reaching the sink-node of the last network layer. Recall that each network layer is generated for a driver $m \in \mathcal{M}$. A feasible solution is achieved, since the horizontal and vertical rules are considered in the construction heuristic. Let e^s_m be the first node of the network layer of the driver m. In $\mathcal{F}'_n$ the compatibility arcs e are collected such that adding their corresponding activity arcs e' to the actual solution does not violate any maximum block lengths or generate any forbidden sequences. One of them is chosen based on the weighted probabilities p_e, which will be described in Section 10.3.2. Let a parameter A_e store the outgoing activity arc of the compatibility arc $e \in \mathcal{E}$. Additionally, the capacity of the activity d of e' does not exceed I_d (vertical rules). In order to determine valid activities all meta-information should be tracked. In the function *UpdateMetaInfo*() in Algorithm 10.1, the actual block length of working days and days off as well as the selected capacity of the activity d of e' are updated. An example illustrated in the left-hand side of Figure 10.1 shows the update of block length of working days and days off. The capacity of each activity is updated for all drivers on each day. If no valid choice is generated, backtracking (Algorithm *BacktrackRepair*) is activated until a valid choice is available or a source-node is reached.
In the backtracking algorithm, $\mathcal{K}$ is defined as a set of activity arcs (similar to a tabu list), which are stored during the backtracking to ensure that the same path will not be selected again. It is done by removing the last added activity k from the solution, and the link to k is marked as forbidden (see line 1). Then the valid options outgoing from the new node are evaluated. Since the last-used link is forbidden, there is no possibility to use the same path again. If there are still no other options left, the last activity is removed and the forbidden link is updated to the new one, since the first forbidden link could possibly belong to an alternate path. This is repeated until no valid options are available or a sink-node of a previous driver is the current last node (see lines 2–4). The algorithm gives activity arc k as its output.

Algorithm 10.1: *AntWalk*()

$\mathcal{X} \leftarrow \emptyset$
$\mathcal{Q} \leftarrow$ Randomly ordered queue of sub-graph start-nodes e^s_m for all employees $m \in \mathcal{M}$
forall $n \in \mathcal{Q}$ **do**
  $n \leftarrow e^s_m$
  while $n \neq nil$ **do**
    $\mathcal{F}'_n \leftarrow$ Subset of feasible outbound edges of $\mathcal{F}_n$
    if $\mathcal{F}'_n = \emptyset$ **then** $a \leftarrow BacktrackRepair(\mathcal{X})$
    Probabilistically choose $e \in \mathcal{F}'_a$ depending on p_e, $e' \leftarrow A_e$
    $\mathcal{X} \leftarrow \mathcal{X} \cup \{e'\}$, *UpdateMetaInfo*(), $n \leftarrow e'$
return $\mathcal{X}$

Algorithm 10.2: *BacktrackRepair*($\mathcal{X}$)

$k \leftarrow Pop(\mathcal{X})[0]$, *UpdateMetaInfo*(), $\mathcal{K} \leftarrow \mathcal{K} \cup k$
while $\mathcal{F}'_k = \emptyset$ **do**
  if $k \neq nil$ **then** $\mathcal{K} \leftarrow \mathcal{K} \setminus \{k\}$
  $k \leftarrow Pop(\mathcal{X})[0]$,*UpdateMetaInfo*(),$\mathcal{K} \leftarrow \mathcal{K} \cup k$
return k

10.3.2 Solving rota scheduling by ant colony optimization

Ant colony optimization (ACO) is a technique for optimization, which was introduced in the early 1990s by Dorigo and his colleagues (see [8], [10]). It belongs to swarm intelligence, which applies the behavior of real swarms or insect colonies to efficient computational methods (see [4] for an overview). ACO was inspired by the foraging behavior of real ant colonies, where ants can find the shortest paths between their nest and food sources. Initially, ants search randomly near their nest to find food. While searching, they leave a chemical pheromone trail on the ground. Ants which choose the shorter path arrive earlier at the food source, and prefer this path on their way back to the nest. The other ants will choose the paths with stronger pheromone concentration to find the food source. The pheromone will accumulate faster on the shorter path than on the others, therefore, after some time nearly all ants take the shorter path.

The basic ACO algorithm for combinatorial optimization problems is shown in Algorithm 10.3 (from [11]). After parameters are initialized, a main loop including three steps is repeated until a termination condition is met. First, feasible solutions are constructed by ants, then they are (optionally) improved by using a local search, and

finally the pheromones are updated. During the update all pheromone values are decreased by a certain percentage (evaporation), while the pheromone values corresponding to good solutions are increased (intensification). The most commonly used stopping criteria (possibly used in combination) are that a predefinied maximum number of iterations has been reached, a time limit has been exceeded, or the best solution has not changed over a certain number of iterations.

Algorithm 10.3: ACO algorithm for combinatorial optimization problems

```
1 Initialization
2 while termination condition not met do
3 |   ConstructAntSolutions
4 |   ApplyLocalSearch      %optional
5 |   UpdatePheromones
```

Algorithm 10.4 shows the new structure of ACO applied to the crew rostering problem. The main differences from the basic ACO algorithm shown in Algorithm 10.3 are the construction of ant solutions in parallel (see line 3) and the updating of heuristic information for every h-th iteration (see line 6).

Algorithm 10.4: *ACO*()

```
1 Initialization, i ← 0
2 while termination condition not met do
3 |   forall a ∈ Colony in parallel do Colony[a].X ← AntWalk()
4 |   a_best ← argmin_{a∈Colony} f(Colony[a].X)
5 |   SA(Colony[a_best].X, t_0 ← 0), UpdatePheromones
6 |   if i mod h = 0 then UpdateHeuristicInformation
7 |   i ← i + 1
8 Return Colony[a_best].X
```

Initialization At the beginning of the ACO algorithm, parameters (such as h, t_0) are set and all pheromone variables are set to a value τ_0.

ConstructAntSolutions In line 3 of the ACO algorithm, an AntWalk algorithm is applied to generate a solution for each ant a (see Algorithm 10.1). Recall that $\mathcal{X}$ is the set of all activity arcs in a feasible solution for all drivers. A feasible outgoing edge $e \in \mathcal{F}'_n \subset \mathcal{F}_n$ is chosen based on the weighted probability p_e (in line 8 of

Algorithm 10.1). We assume an ant chooses a node n on day i. From day i to day $i+1$, the subsequent construction step is done with probability

$$p_e = \begin{cases} \frac{\tau_e^\alpha \cdot \eta_e^\beta}{\sum_{e' \in \mathcal{F}'_n} \tau_{e'}^\alpha \cdot \eta_{e'}^\beta} & e \in \mathcal{F}'_n \\ 0 & otherwise \end{cases} \tag{10.1}$$

where τ_e is defined as *pheromone value* of $e \in \mathcal{F}'_n$, while η_e is the *heuristic information* of e. This is called a *state transition rule*. The exponents α and β are positive parameters, which indicate the relationship between the pheromone value and heuristic information. If $\alpha = 0$, then an arc $e \in \mathcal{F}'_n$ is chosen dependent only on the heuristic information. If $\beta = 0$, then the choice depends only on the pheromone value of that arc e.

In this step, *parallelization* is implemented to gain a speed-up. Instead of the parallel independent execution of multiple colonies introduced by [27] and [29], we adopt another approach described in [27], where the construction and the evaluation of ants within a colony are implemented in parallel. On the one hand, this is due to memory limitations. On the other hand, the influence of randomness is quite low. Thus, multiple executions will not have a significant effect on the quality of the solution. This is one of the extensions we implemented in order get a better solution quickly compared with the basic ACO algorithm. Another one is the update of heuristic information, which we will describe later.

ApplyLocalSearch In line 5 of the ACO algorithm, the value obtained by the function $f(\mathcal{X})$ is defined as the evaluation value of a solution $\mathcal{X}$. And the best-so-far solution Colony$[a_{best}].\mathcal{X}$ is improved by a local search technique, which is based on the simulated annealing described in Section 10.3.3. However, the initial temperature t_0 is set to be zero, i.e. a simple local search with limited iterations is applied to improve the solution.

UpdatePheromones The pheromone τ_e on each edge e is updated in line 5 of the ACO algorithm as follows:

$$\tau_e = (1-\rho) * \tau_e + \rho * \Delta\tau_e \tag{10.2}$$

where

$$\Delta\tau_e = \begin{cases} \frac{F^{base} * f(\mathcal{X}^1)}{f(\text{Colony}[a_{best}].\mathcal{X})} & e \in \text{Colony}[a_{best}].\mathcal{X} \\ 0 & \text{otherwise} \end{cases} \tag{10.3}$$

That is, we treat the update of the pheromones on the edges that are included in the best solution, and those that are included in other solutions, differently. The parameter $\rho \in (0, 1)$ is defined as a pheromone decay parameter, while F^{base} is a coefficient for pheromone update.

UpdateHeuristicInformation The *heuristic information* is usually problem-dependent and static throughout the algorithm. In our problem, the generation of feasible solutions succeeds due to the enforcement of feasibility; however, the quality of the generated solutions is usually poor. Therefore, the heuristic values η_e need to be updated by every h iteration for each edge e of the best-so-far solution Colony$[a_{best}].\mathcal{X}$ (see Constraint (10.6)). The parameter h is defined to control how often the update of heuristic information occurs. As mentioned before, the violation of each quality rule is associated with a penalty cost, therefore we define $\mathcal{C}$ as a set of cost categories. The value δ_e in Constraint (10.6) can be a positive or negative update for an edge e depending on whether the edge is causing high costs or not. The influencing data on the value δ_e of an edge e consists of the related weight w_i of each cost category $i \in \mathcal{C}$ in Constraint (10.4), and the impact of the edge e for the cost category i, $\xi_{e,i}$. We define $\xi_{e,i}$ as a dynamic value, which depicts the positive or negative influence of the cost category i on the edge e. For example, a violation of the minimal number of days off might exist in the incumbent solution. During the update of heuristic values, a slight increase of η_e is performed for each edge e, which leads to more days off for drivers with insufficient days off. The calculation of the value δ_e is shown in Constraint (10.5).

To avoid extreme heuristic values, which might drain the influence of the pheromone values, a normalization is required after each update to transform all heuristic values into a predetermined range $[0, 1]$ (see Constraint (10.7)). Let N_{min} and N_{max} be parameters, which define the minimum and maximum of normalized heuristic values. η^{min} and η^{max} are defined as the updated minimum and maximum of heuristic values of all edges.

$$w_i := \frac{C_i}{\sum_{j \in \mathcal{C}} C_j} \quad \text{with } \mathcal{C} \text{ as the set of cost-indices and the costs } C \tag{10.4}$$

$$\delta_e = \sum_{i \in \mathcal{C}} \xi_{e,i} * w_i \tag{10.5}$$

$$\eta_e = \eta_e + \delta_e \tag{10.6}$$

$$N^{min} + (N^{max} - N^{min}) * \frac{\eta_e - \eta^{min}}{\eta^{max} - \eta^{min}} \quad \forall \eta_e > 0 \tag{10.7}$$

Termination The whole procedure is iterated until one of the specified termination criteria is met. In particular, the termination criteria are a given CPU-time limit, a certain number of solution constructions or when an evaluation value stagnates over limited iterations (stored in parameter I^{stag}).

ACS and MMAS In this paper, two ant colony optimization variants are implemented. The first one is derived from the Ant Colony System (ACS) introduced by [9], but we do not implement the local pheromone updating, since the length of the network layers is static for all ants. Thus, no differential length effect can be exploited. Algorithm 10.5 shows the procedure of the ACS algorithm. The second variant is based on the *Max-Min* Ant System (MMAS) introduced by [30], where the limit to the range of pheromone values is set to $[\tau_{min}, \tau_{max}]$ to prevent the search from stagnating.

Algorithm 10.5: ACS()

```
1 initialization
2 while termination condition not met do
3     foreach ant do
4         applies an extended state transition rule (see constraint (10.8)) to
          incrementally build a solution
5     A pheromone updating rule is applied
```

Algorithm 10.6: MMAS()

```
1 initialization, τ_0 ← τ_max
2 while termination condition not met do
3     foreach ant do
4         applies a state transition rule (see constraint (10.1)) to incrementally
          build a solution
5     A pheromone updating rule is applied (limit to [τ_min, τ_max])
```

According to Algorithms 10.5 and 10.6, the difference between ACS and MMAS lies in the transition rule and the update of pheromone. For MMAS, a classic stochastic variant is chosen as shown above in (10.1). However, the transition rule of ACS extends the transition rule in (10.1) in the following way:

$$e = \begin{cases} \underset{e \in \mathcal{F}_n}{\operatorname{argmax}} \left\{ \tau_e^{\alpha} \cdot \eta_e^{\beta} \right\} & \text{if } q \leq q_0 \\ X & otherwise \end{cases} \tag{10.8}$$

where q is a random number uniformly distributed in [0...1], and q_0 is a parameter between 0 and 1. If $q \leq q_0$ then the best edge is chosen, otherwise the probabilistic decision according to (10.1) is made. $X : \Omega \to \mathcal{F}_n$ is defined as a random variable with $\mathcal{P}(\{x = e\}) = p_e$. The heuristic values η_e are initiated as described previously in this section, considering dynamic weights w_i.

For the pheromone values, the MMAS algorithm uses an explicit lower bound $\tau_{min} > 0$ and an upper bound $\tau_{max} > 0$ to limit the pheromone values. If a new best-so-far solution is found, then the bounds are dynamically updated (for more detail see [30]):

$$\tau_{max} = 1/((1-\rho) * F(\text{Colony}[a_{best}].\mathcal{X})) \tag{10.9}$$

$$\tau_{min} = (\tau_{max} * (1 - \sqrt[n]{p_{best}}))/((avg - 1) * \sqrt[n]{p_{best}}) \tag{10.10}$$

where p_{best} is defined as the probability of finding the best solution. The parameter avg is defined as the mean of all possible solutions at each choice point of an ant, i.e. the average number of outgoing arcs for each compatibility arc.

10.3.3 Solving rota scheduling by simulated annealing (SA)

Simulated annealing (SA) was inspired by cooling in metallurgy. This algorithm is independently described by [19] and [7], who developed an algorithm based on the physical process described in [24].

The algorithm used here is depicted in Algorithm 10.7. First, an initial solution is generated using Algorithm 10.1 (*AntWalk*), and f^{best} is chosen based on fitness $f(\mathcal{X})$. Then an operator is selected by a weighted random choice from the available set of local search operators described in Section 10.3.1.2 (called *RandomlySelectOperator()*). The weights are given by the parameter values $W^{simple}, W^{multi}, W^{crawl}, W^{shift}$. After that, an activity is chosen randomly based on the initial solution $\mathcal{X}$ and the selected operator *op* (called *RandomlyChooseActivity*($\mathcal{X}$, *op*)). For the Crawl operator, a double-off will be chosen, and for the Duty-Switch operator, a duty activity is necessary, while for the other operators, no particular activity type for the selected edge has to be considered. After applying the selected operator, a new fitness $f(\mathcal{X}')$ will be evaluated for the modified solution $\mathcal{X}'$. If it is better than the global best one f^{best}, the new solution replaces the old one. Otherwise, in order to overcome local optima, it is still possible to select a poorer solution that depends on a random variable y between 0 and 1, the current temperature t, and the deterioration of the evaluation values $f(\mathcal{X}') - f^{best}$. If the following formulation is true, then the solution $\mathcal{X}'$ is selected.

$$y < e^{-\frac{f(\mathcal{X}') - f^{best}}{t}} \tag{10.11}$$

Algorithm 10.7: $SA()$

initialization, $t \leftarrow t_0$, $i \leftarrow 0$, $\mathcal{X} \leftarrow AntWalk()$, $f^{best} \leftarrow f(\mathcal{X})$
while *termination condition not met* **do**
  if $i \leq I^{max}$ **then**
    $op \leftarrow RandomlySelectOperator()$,
     $e \leftarrow RandomlyChooseActivity(\mathcal{X}, op)$
    if $op =$ *Simple-Switch* **then** $\mathcal{X}' \leftarrow$ Simple-Switch$(\mathcal{X}, e)$
    if $op =$ *Multi-Switch* **then** $\mathcal{X}' \leftarrow$ Multi-Switch$(\mathcal{X}, e)$
    if $op =$ *Crawl* **then** $\mathcal{X}' \leftarrow$ Crawl$(\mathcal{X}, e)$
    if $op =$ *Duty-Switch* **then** $\mathcal{X}' \leftarrow$ Duty-Switch$(\mathcal{X}, e)$
    $y \leftarrow random(0, 1)$
    if $(f(\mathcal{X}') \leq f^{best}) \vee (t > 0 \wedge y < e^{-\frac{f(\mathcal{X}') - f^{best}}{t}})$ **then**
      $f^{best} \leftarrow f(\mathcal{X}')$, $\mathcal{X} \leftarrow \mathcal{X}'$
    $i \leftarrow i + 1$
  else
    $i \leftarrow 0$, $t \leftarrow \alpha \cdot t$
Return $\mathcal{X}^{best}$

Let I^{max} be the parameter defining the number of iterations on a temperature level: the temperature will be cooled down if this number of iterations is achieved. The geometric cooling schedule ($t \leftarrow \alpha \cdot t$) is used to reduce the temperature t gradually, where α is defined between $[0.8, 0.99]$.
This procedure is repeated while the temperature is reduced. The termination condition is met when the temperature is equal to zero or when the time limit is reached.
As mentioned above, this procedure is used as the local search sub-routine in the ACO approach (see Algorithm 10.4). In this case the initial solution is provided rather than generating a new one. Since the temperature t is already set to 0, only improved solutions will be accepted, such that the SA procedure in this case is considered as a simple local search algorithm.

10.3.4 Solving rota scheduling by tabu search (TS)

Another local search method we use in this paper is tabu search (TS), which was proposed in [13]. The basic idea is to collect the subset of the moves in a forbidden neighborhood (tabu) that prevents cycling when moving away from local optima through non-improving moves. The search space is the space of all feasible solutions that can be visited during the search. More details can be found in [14], [15], [16].

Algorithm 10.8: $TS()$

initialization, $i \leftarrow 0$, $\mathcal{X} \leftarrow AntWalk()$, $\mathcal{X}^{storage} \leftarrow \{\mathcal{X}\}$
while *termination condition not met* **do**
  forall $e \in \mathcal{X}^{S=2}$ **do** $\mathcal{X}' \leftarrow \text{Crawl}(\mathcal{X}, e)$, $\mathcal{X}^{storage} \leftarrow \mathcal{X}^{storage} \cup \mathcal{X}'$
  forall $e \in \mathcal{X}$ **do**
    $\mathcal{X}' \leftarrow \text{Simple-Switch}(\mathcal{X}, e)$, $\mathcal{X}^{storage} \leftarrow \mathcal{X}^{storage} \cup \mathcal{X}'$
    $\mathcal{X}' \leftarrow \text{Multi-Switch}(\mathcal{X}, e)$, $\mathcal{X}^{storage} \leftarrow \mathcal{X}^{storage} \cup \mathcal{X}'$
  $\mathcal{X}^{best} \leftarrow \underset{\mathcal{X}' \in \{\mathcal{X}' \in \mathcal{X}^{storage} | i - rec(\mathcal{X}') > d^{rec}\}}{\text{argmin}} f(\mathcal{X}')$, $\mathcal{X} \leftarrow \mathcal{X}^{best}$, $\mathcal{X}^{storage} \leftarrow \{\mathcal{X}\}$,
  $i \leftarrow i + 1$
Return $\mathcal{X}^{best}$

A simple tabu search is developed in this work (see Algorithm 10.8). Some of the operators described in Section 10.3.1.2 are applied here as well. Similarly to SA, the initial solution is also generated by Algorithm 10.1. The implementation of the tabu search algorithm is similar to the one described in [4]. In each iteration, every possible switch based on the Crawl, Simple-Switch, and Multi-Switch operators is used to generate a new solution $\mathcal{X}'$, which is stored in a set $\mathcal{X}^{storage}$. At the beginning of Algorithm 10.8, every double-off is shifted to the left and the right, where $\mathcal{X}^{s=2}$ is defined as the set of active double-off activities. Then all available Simple-Switch and Multi-Switch exchanges are evaluated. At the end of each iteration, the best available exchange is executed. Exchanges unavailable to this selection are the ones that are tabu. An exchange is tabu if it affects an employee on a specific day that has been exchanged during the last d^{rec} iterations. This is repeated until no further exchanges are available due to the *recency*, or either the time limit or maximum number of iterations is reached.

10.4 Computational results

We begin this section with the detail of our results for solving the rota scheduling problem, which is followed by the short description of the results for solving the integrated problem. In this paper, we test our solution approaches on the same set of real-world instances as shown in [32], i.e. 15 real-world instances of personalized crew rostering for which the number of assigning duties varies between 1,313 and 19,486. The number of drivers in the instances varies between 48 and 629. The name of each instance includes the number of drivers, the number of planning days, and the number of shift types. As mentioned in [32], these instances are available at the web page `http://dsor.upb.de/crewrostering`. All experiments, including parameter tunings for all metaheuristics and the results solved by solvers (from [32]),

were performed on six technically identical machines with 8 GB of main memory and Intel Core2Quad CPUs at 2.83 GHz running Microsoft Windows 7.

10.4.1 Results of solving the rota scheduling problem

Nowadays, algorithm configuration approaches are used to obtain an efficient set of parameters for a given set of instances. According to [1], a large set of instances is required for an algorithm configuration to avoid over-tuning effects. Due to our limited number of real-world instances, such an algorithm configuration method is not adopted in this paper. Instead we conduct a limited full factorial design experiment for each method with 10,368 combinations of parameters assessed overall and two repetitions done for each combination. The best parameters were then selected for the remaining experiments and are shown below:

ACO

1. number of ants within a colony $|Colony| = 10$.
2. influence of pheromone value $\alpha = 1$.
3. influence of heuristic information $\beta = 1$.
4. pheromone decay parameter $\rho = 0.7$.
5. coefficient for pheromone update $F^{base} = 10$.
6. number of iterations after which the ACO is ended due to stagnation $I^{stag} = 100$.
7. interval of updating heuristic information $h = 1$.
8. minimum of normalized heuristic value $N^{min} = 0.1$.
9. maximum of normalized heuristic value $N^{max} = 1$.
10. ACS
    (a) the probability of exploitation $q_0 = 0.01$.
11. MMAS
    (a) parameter to influence τ_{min}, $P_{best} = 0.05$.
    (b) lower bound of pheromone value $\tau_{min} = 0.1$.
    (c) upper bound of pheromone value $\tau_{max} = 1$

SA

1. initial temperature $t_0 = 1e7$.
2. maximum iterations $I^{max} = 1e5$.

3. temperature reduction $\alpha = 0.9$.

TS

1. number of iterations of recency $d^{rec} = 50$.

The number of ants is limited to 10. Based on our experiment, this restriction does not limit the overall performance of the ACO algorithm, since there is few difference between the evaluation values obtained by 10 ants and those obtained by 15, 30, and 50 ants within the 30 minute timelimit.
In this paper, we develop three additional functionalities compared with the classic ACO algorithm, i.e. parallelization, updating heuristic information, and embedded local search. We test the benefits of those developments, including ACS, ACS with parallelization (in short: ACSPara), MMAS, and MMAS with parallelization (in short: MMAPara).
During the evaluation, two physical cores are available; the parallelized variant gains an average speed-up of about 1.80 for the ACS compared with the original variant and about 1.82 for the MMAS (see Table 10.1). Beside that, the structure of the problem influences the time for an artificial ant to generate a feasible solution. Furthermore, some times might be lost because of the updates of the heuristic values and pheromones. Anyway, a speed-up is achieved across all instances. Note that, the same running time (30 minutes are considered as acceptable time limit for our real-world instances) is set to ACS, ACSPara, MMAS and MMASPara to see a speed-up during the same time, for example Figure 10.7 shows a speed-up for the ACS and MMAS for the instance 629-46-26. The shaded area represents the standard deviation of all runs and shows the distribution of different methods in all runs. Similar results can be obtained for any other instances.
Moreover, changing heuristic information periodically implies a speed-up. Although the same initial solution and the same values of initial heuristic information are used, periodic update of the heuristic information brings an improvement to the solution at an early stage, for both ACS and MMAS. After that, the algorithm is able to exploit a superior neighborhood of solutions and terminates with a much better one. For example, we show distributions with the activated vs. deactivated update of heuristic information for the large instance 629-46-26 in Figure 10.8. Similar results can be obtained for all other instances.
However, the local search method was deactivated during both evaluations above, since its great impact on the performance overshadows the two aspects above. Moreover, the local search algorithm consumes much time, but it is necessary in order to achieve good solutions for solving the driver rostering problem; for example, Figure 10.9 shows the example of all ACO methods with activated and deactivated local search for the instance 629-46-26. The local search improves the solutions from the

Table 10.1 Results of the parallelization evaluation (runtime in seconds and the relative speed-up).

Instance	ACS	ACSPara	Speed-up	MMAS	MMASPara	Speed-up
48-75-6	69.09	32.99	2.09	69.44	33.05	2.10
52-73-6	77.54	36.86	2.10	77.25	36.42	2.12
52-75-6	75.48	36.92	2.04	76.20	36.37	2.09
393-45-37	343.56	180.95	1.90	345.77	168.25	2.06
392-45-37	336.89	170.56	1.98	339.21	167.74	2.02
397-40-37	303.58	152.61	1.99	316.10	160.17	1.97
96-70-8	330.53	193.91	1.70	334.48	194.22	1.72
87-70-8	308.89	180.50	1.71	306.06	177.99	1.72
89-70-8	356.31	207.96	1.71	347.45	208.78	1.66
221-45-30	727.75	464.35	1.57	716.28	455.44	1.57
211-45-34	445.10	272.97	1.63	438.79	272.26	1.61
214-45-34	570.70	335.44	1.70	575.64	338.52	1.70
629-46-26	1,362.00	794.77	1.71	1,393.75	815.15	1.71
606-70-26	2,816.89	1,777.88	1.58	2,844.88	1,728.66	1.65
607-70-26	2,805.16	1,705.26	1.65	2,762.45	1,719.46	1.61
Average			1.80			1.82

beginning. In this test, the update of heuristic information is not included. Similar results can be obtained using all ACO methods for all instances.

In Table 10.2, the results of ACO methods (including ACS, ACSPara, MMAS, and MMASPara with update of heuristic information as well as applying the local search) are compared with those obtained by SA, TS, the Gurobi solver, and the CPLEX solver. In the table, the relative relation between each evaluation value and the best one is shown for each instance. The time limit is set to 30 minutes. Furthermore, 10 repetitions were done for all metaheuristic methods to minimize the influence of randomness, thus this results in 900 performed tests. For some instances, the speed-up by parallelization in ACO methods is not beneficial, because the methods converge or terminate before the timeout, such as 397-40-37 and 211-45-34. Moreover, all ACO methods tend to generate a very good solution at the early stage, while SA and TS have poorer starts, although their start solutions are generated by the same algorithm (Algorithm 10.1). TS generally produces weak solutions, so the main competitors among our implemented metaheuristics are the SA and ACO methods.

It is obvious that the ACO methods can compete with SA only on small instances with about 50 drivers. Our SA obtains generally better or equal solutions for all instances compared with all ACO methods, except for the instance 48-75-6. The

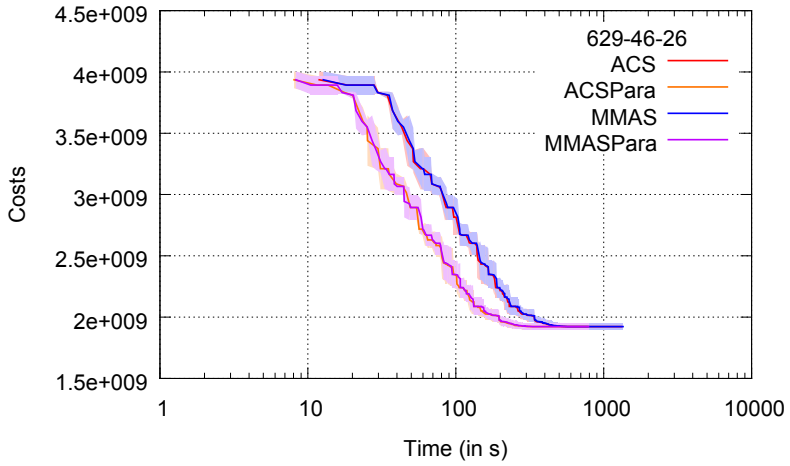


Figure 10.7 Behavior of the different ACO methods for instance 629-46-26 (logarithmic x-scale).

Table 10.2 Evaluation overview – the obtained evaluation values relative to the particular best value of each instance at the end of the 30-minute time limit. "n.a." means that no solution is obtained.

Instance	ACS	ACSPara	MMAS	MMASPara	SA	TS	Gurobi	CPLEX
48-75-6	1.02	1.02	1.02	1.02	1.25	1.26	1.00	1.00
52-73-6	1.00	1.00	1.00	1.00	1.00	1.01	1.00	1.00
52-75-6	1.00	1.00	1.00	1.00	1.00	1.01	1.00	1.00
393-45-37	1.30	1.29	1.30	1.29	1.19	2.81	1.00	1.00
392-45-37	1.08	1.08	1.08	1.08	1.05	1.57	1.00	1.00
397-40-37	1.19	1.19	1.19	1.19	1.13	2.57	1.00	1.00
96-70-8	1.27	1.24	1.27	1.24	1.16	1.66	1.00	1.00
87-70-8	3.25	3.08	3.26	3.08	1.81	15.49	1.00	1.00
89-70-8	1.31	1.26	1.31	1.27	1.00	4.34	2.57	1.00
221-45-30	1.17	1.16	1.17	1.16	1.00	1.31	n.a.	6.99
211-45-34	1.90	1.89	1.91	1.89	1.70	2.27	1.00	24.64
214-45-34	2.13	2.11	2.13	2.11	2.01	2.52	1.00	15.60
629-46-26	1.61	1.55	1.62	1.55	1.00	3.07	n.a.	7.13
606-70-26	1.59	1.56	1.58	1.55	1.00	2.50	n.a.	n.a.
607-70-26	1.65	1.62	1.64	1.61	1.00	2.40	n.a.	n.a.

possible explanation for this result is the agility of the SA approach. As shown in Table 10.3, the number of iterations per second is higher with SA than with any other approaches, since only some parts of solutions might be altered while using the δ-evaluation instead of creating a new one. This probably enables SA to dive

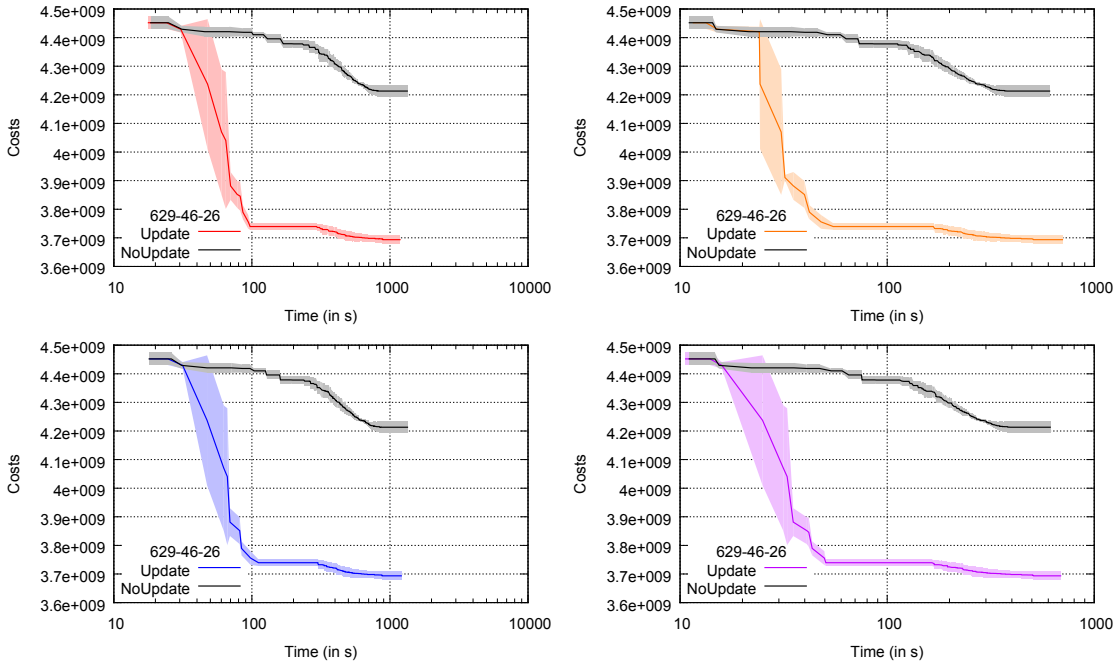


Figure 10.8 Behavior of different ACO methods (from top left to bottom right: ACS, ACSPara, MMAS, MMASPara) with the activated update of the heuristic information vs. the deactivated update for instance 629-46-26 (logarithmic x-scale).

into good solutions very fast, while the ACO approaches need more time on each iteration to generate new solutions for most of the instances.

In order to give some insight about the significance of the obtained results we conducted a paired two-sample t-test on all method combinations. The resulting t-values are shown in Table 10.4. With a significance level of 1% the absolute test value has to be greater than 2.609. We color all significantly better values in terms of the average objective value in green, while all significantly worse are colored in red. These results show that our SA performs significantly better than all other methods evaluated in this paper, while our TS is significantly worse than all other methods. In contrast, the ACO based methods are almost insignificantly different to each other. The main reason for this is that we use the same local search algorithm for those methods, which has a notable impact on the performance (see Fig. 10.9).

Moreover, the solutions obtained by solvers perform well on smaller instances (such as instances 48-75-6 to 89-70-8), but are unable to generate one feasible solution within the time limit for a larger instance, such as 606-70-26 or 607-70-26. In such cases, the SA method brings good solutions within 30 minutes; for instance, the CPLEX solver obtains a solution seven times larger than the one obtained by the SA method for the instance 629-46-26. According to our experiments, longer

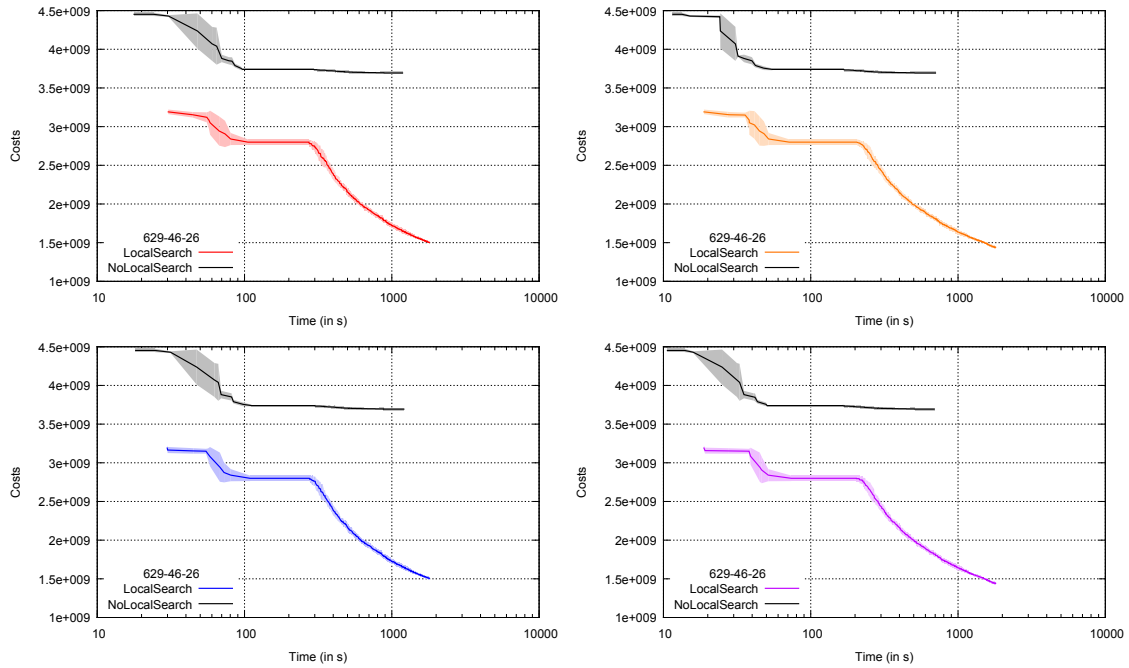


Figure 10.9 Behavior of different ACO methods (from top left to bottom right: ACS, ACSPara, MMAS, MMASPara) with activated local search vs. deactivated local search for instance 629-46-26 (logarithmic x-scale).

running times (the time limit is not set) for all metaheuristics approaches do not bring significantly better solutions.

The SA method is the best metaheuristic we implemented; therefore, it is worth taking a deeper look at the local search operators used in the SA method (see Section 10.3.1.2), which are different from the existing operators used in [22]. In order to see the influence of each operator on the quality of the solution, different combinations of their weights are tested (the order: $W^{simple} - W^{multi} - W^{crawl} - W^{duty}$). In the left-hand side of Figure 10.10, one of the local search operators is activated. The Crawl operator brings the worst solutions (the normalized evaluation values) for all instances. We know this in advance, since this operator is only an assistant operator for the others. The second worst operator is the Duty-Switch operator, but for some small instances, such as 52-73-6 and 52-75-6, this operator alone provides good solutions. The classical Simple- and Multi-Switch operators generally bring good solutions. Therefore, we would like to see the influence of deactivating one of the operators in the right-hand side of Figure 10.10. Without Simple-Switch and Crawl operators the solutions for very large instances are worse, while the solutions for medium-sized and large instances are worse without the

Table 10.3 Overview of the average iterations per second of the different approaches.

Instance	ACS	ACSPara	MMAS	MMASPara	SA	TS
48-75-6	0.37	0.41	0.38	0.41	11,438.29	17.32
52-73-6	1.16	1.68	1.17	1.67	18,650.84	18.63
52-75-6	1.19	1.72	1.20	1.66	18,802.83	18.01
393-45-37	0.31	0.42	0.30	0.42	7,389.54	5.03
392-45-37	0.33	0.46	0.32	0.47	9,358.61	3.63
397-40-37	0.35	0.50	0.35	0.50	8,630.59	5.12
96-70-8	0.35	0.44	0.35	0.44	4,328.90	1.26
87-70-8	0.35	0.44	0.35	0.44	4,371.06	1.33
89-70-8	0.33	0.41	0.33	0.41	4,397.49	1.21
221-45-30	0.18	0.22	0.19	0.22	3,102.91	0.59
211-45-34	0.21	0.26	0.21	0.26	3,255.13	0.61
214-45-34	0.20	0.25	0.21	0.26	2,912.51	0.51
629-46-26	0.07	0.09	0.07	0.09	1,647.82	0.15
606-70-26	0.05	0.05	0.05	0.05	1,461.38	0.11
607-70-26	0.05	0.05	0.05	0.05	1,585.37	0.12
Overall	0.44	0.55	0.43	0.54	6,493.02	10.52

Table 10.4 t-values obtained by a paired two-sample t-test (row is first sample and column second sample) for each method pair rounded to three decimal places. Results with a significance level of 1% are colored.

	ACS	ACSPara	MMAS	MMASPara	SA	TS
ACS		4.537	-0.213	0.444	8.800	-11.124
ACSPara	-4.537		-1.298	-0.647	8.727	-11.161
MMAS	0.213	1.298		1.790	8.618	-11.069
MMASPara	-0.444	0.647	-1.790		8.809	-10.971
SA	-8.800	-8.727	-8.618	-8.809		-10.590
TS	11.124	11.161	11.069	10.971	10.590	

Duty-Switch operator. The activating of all operators brings the best or second-best solution (the difference to the best one is less than 1%) for all instances.

10.4.2 Results of solving the integrated problem

Based on the previous parameter setting and results, the simulated annealing method is the best choice among several metaheuristics for solving our rota schedul-

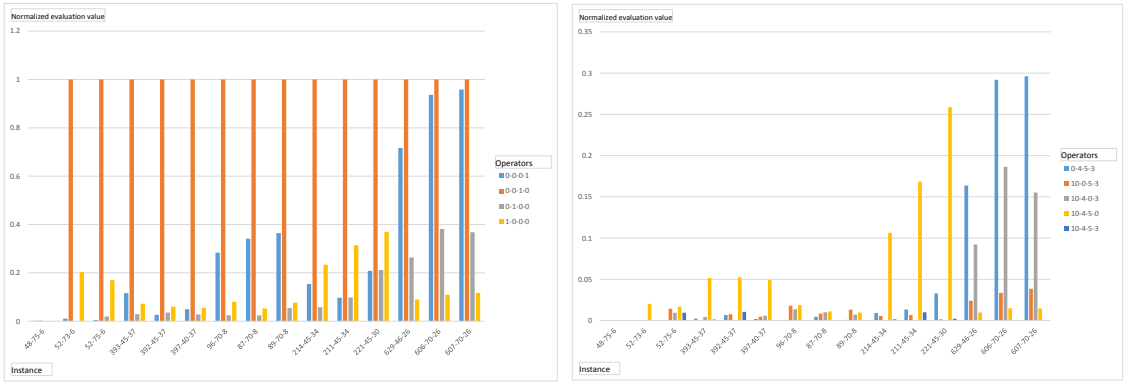


Figure 10.10 The results obtained for all instances using SA by activating one of the local search operators (left) and after deactivating one of the local search operators and activating all operators (right).

ing problem. Therefore, we use the simulated annealing to solve the integrated problem. The simulated annealing approach described in Section 10.3.3 remains; however, the constraints about the duties, such as the relation between a shift and a duty with that shift type, the capacity of each duty, and impossible combinations of duties, should be checked during the generation of the initial solution and during the switching.

For solving the integrated approach, the advantage of applying the SA method is more obvious compared to solving the rota scheduling problem (see Table 10.5). The instances that are larger than 397-40-37 solved by the SA can outperform the results of the CPLEX solver with the same running time of as SA. Moreover, the SA method produces solutions for the very large instances, which cannot be solved by the CPLEX solver due to it running out of memory. Note that here we compare the results of SA only with those produced by the CPLEX solver, since the CPLEX solver is proved to be more suitable for the integrated problem than the Gurobi solver in [32].

Table 10.5 Evaluation overview – the obtained evaluation values relative to the particular best value of each instance at the end of time limit of the SA. "n.a." means that no solution is obtained.

Instance	SA	CPLEX
48-75-6	2.1	1
52-73-6	1.4	1
52-75-6	1.4	1
393-45-37	1.3	1
392-45-37	1.1	1
397-40-37	1	3.7
96-70-8	1	14.6
87-70-8	1	131.6
89-70-8	1	9.7
221-45-30	1	9.7
211-45-34	1	12.1
214-45-34	1	n.a.
629-46-26	1	n.a.
606-70-26	1	n.a.
607-70-26	1	n.a.

10.5 Conclusions

The goal of this work was to explore novel metaheuristics, including ant colony algorithms, simulated annealing, and tabu search, to solve the personalized crew rostering problem in public bus transit. We demonstrate that our different metaheuristics are capable of solving large scale real-world rostering problems. Moreover, results from the simulated annealing method outperform those obtained with the CPLEX-solver within the same running time of simulated annealing.
In Future, we want to test our algorithms with more real-world instances, but not limited to instances in German bus companies. We consider in this work only the law and labor union rules of German bus companies, but our algorithms can be extended to adopt the other rules, for example the percentage of weekends-off. Moreover, we are able to do better algorithm configurations for the metaheuristics implemented in this work due to more real-world instances.

References

[1] Carlos Ansótegui, Meinolf Sellmann, and Kevin Tierney. "A Gender-Based Genetic Algorithm for the Automatic Configuration of Algorithms". In: *Principles and Practice of Constraint Programming - CP 2009: 15th International Conference, CP 2009 Lisbon, Portugal, September 20-24, 2009 Proceedings*. Ed. by Ian P. Gent. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 142–157. ISBN: 978-3-642-04244-7. DOI: 10.1007/978-3-642-04244-7_14.

[2] C. Barnhart and G. Laporte. *Handbooks in Operations Research & Management Science*. Vol. 14. Elsevier, 2006.

[3] L. Bianco et al. "A heuristic procedure for the crew rostering problem". In: *European Journal of Operational Research* 58.2 (1992), pp. 272–283.

[4] E.K. Burke and G. Kendall. *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*. Springer, 2005.

[5] P. Carraresi and G. Gallo. "A multi-level bottleneck assignment approach to the bus drivers' rostering problem". In: *European Journal of Operational Research* 16.2 (1984), pp. 163–173.

[6] F. Catanas and J. Paixão. "A New Approach for the Crew Rostering Problem". In: *Computer-Aided Transit Scheduling*. Ed. by J. Daduna, I. Branco, and J. Paixao. Vol. 430. Lecture Notes in Economics and Mathematical Systems. Springer, Berlin & Heidelberg, 1995, pp. 267–277.

[7] V. Černý. "Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm". In: *Journal of Optimization Theory and Applications* 45.1 (1985), pp. 41–51.

[8] M. Dorigo. "Optimization, Learning and Natural Algorithms". PhD thesis. Dipartimento di Elettronica, Politecnio di Milano, Italy, 1992.

[9] M. Dorigo and L.M. Gambardella. "Ant colony system: A cooperative learning approach to the traveling salesman problem". In: *IEEE Transactions on Evolutionary Computation* 1.1 (1997), pp. 53–66.

[10] M. Dorigo, V. Maniezzo, and A. Colorni. "Ant system: optimization by a colony of cooperating agents". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics* 26.1 (1996), pp. 29–41.

[11] M. Dorigo and T. Stützle. "Ant colony optimization: overview and recent advances". In: *Handbook of Metaheuristics*. Springer, 2010, pp. 227–263.

[12] T. Emden-Weinert, H.G. Kotas, and U Speer. *DISSY – A driver scheduling system for public transport*. Tech. rep. Bremen, Germany: VSS GmbH and Bremer Straßenbahn AG, 2000.

[13] F. Glover. "Future paths for integer programming and links to artificial intelligence". In: *Computers & Operations Research* 13.5 (1986), pp. 533–549.

[14] F. Glover. "Tabu search - part I". In: *ORSA Journal on Computing* 1.3 (1989), pp. 190–206.

[15] F. Glover. "Tabu search - part II". In: *ORSA Journal on Computing* 2.1 (1990), pp. 4–32.

[16] F. Glover. "Tabu search: A tutorial". In: *Interfaces* 20.4 (1990), pp. 74–94.

[17] T. Hanne, R. Dornberger, and L. Frey. "Multiobjective and preference-based decision support for rail crew rostering". In: *IEEE Congress on Evolutionary Computation CEC'09*. 2009, pp. 990–996.

[18] S.H. Huang, T.H. Yang, and R.T. Wang. "Ant colony optimization for railway driver crew scheduling: from modeling to implementation". In: *Journal of the Chinese Institute of Industrial Engineers* 28.6 (2011), pp. 437–449.

[19] S. Kirkpatrick, C. D. Gelatt Jr, and M.P. Vecchi. "Optimization by simulated annealing". In: *Science* 220.4598 (1983), pp. 671–680.

[20] J. Kyngäs and K. Nurmi. "Days-off scheduling for a bus transportation company". In: *International Journal of Innovative Computing and Applications* 3.1 (2011), pp. 42–49.

[21] C.C. Lo and G.F. Deng. "Using ant colony optimization algorithm to solve airline crew scheduling problems". In: *Third International Conference on Natural Computation, ICNC 2007*. Vol. 4. IEEE. 2007, pp. 797–804.

[22] P. Lučić and D. Teodorović. "Metaheuristics approach to the aircrew rostering problem". In: *Annals of Operations Research* 155.1 (2007), pp. 311–338.

[23] M. Mesquita et al. "A new model for the integrated vehicle-crew-rostering problem and a computational study on rosters". In: *Journal of Scheduling* 14.4 (2011), pp. 319–334.

[24] N. Metropolis et al. "Equation of state calculations by fast computing machines". In: *The Journal of Chemical Physics* 21 (1953), pp. 1087–1092.

[25] M. Moz, A. Respício, and M.V. Pato. "Bi-objective evolutionary heuristics for bus driver rostering". In: *Public Transport* 1.3 (2009), pp. 189–210.

[26] K. Nurmi, J. Kyngäs, and G. Post. "Driver Rostering for a Finnish Bus Transportation Company". In: *IAENG Transactions on Engineering Technologies — Special Edition of the International Multiconference of Engineers and Computer Scientists 2011*. Vol. 7. World Scientific. 2011, p. 15.

[27] M. Randall and A. Lewis. "A parallel implementation of ant colony optimization". In: *Journal of Parallel and Distributed Computing* 62.9 (2002), pp. 1421–1432.

[28] A. Respício, M. Moz, and M.V. Pato. *A Memetic Algorithm for a Bi-objective Bus Driver Rostering Problem*. Centro de Investigação Operacional, Universidade de Lisboa. 2007.

[29] T. Stützle. "Parallelization strategies for ant colony optimization". In: *Proceedings of the 5th International Conference on Parallel Problem Solving from Nature, PPSN V*. Springer. 1998, pp. 722–731.

[30] T. Stützle and H. H. Hoos. "MAX–MIN ant system". In: *Future Generation Computer Systems* 16.8 (2000), pp. 889–914.

[31] L. Xie, M. Naumann, and L. Suhl. "A stochastic model for rota scheduling in public bus transport". In: *Proceedings of 2nd Stochastic Modeling Techniques and Data Analysis International Conference*. 2012, pp. 785–792.

[32] L. Xie and L. Suhl. "Cyclic and non-cyclic crew rostering problems in public bus transit". In: *OR Spectrum* (2014). DOI: 10.1007/s00291-014-0364-9.

[33] T.H. Yunes, A.V. Moura, and C.C. De Souza. "Hybrid column generation approaches for urban transit crew management problems". In: *Transportation Science* 39.2 (2005), pp. 273–288.