UNIVERSITY OF MANNHEIM

Three Essays on Algorithmic Approaches for Operations Management

Inaugural Dissertation to Obtain the Academic Degree of a Doctor in Business Administration at the University of Mannheim

Submitted by

Amir Foroughi
Mannheim

Dekan:	*Joachim Lutz*
Referent:	*Prof. Dr. Raik Stolletz*
Korreferent:	*Prof. Dr. Michael Schneider*

Day of Oral Examination: *08.07.2022*

To my parents

Reza and Zahra

Acknowledgements

This dissertation represents the culmination of years of dedication and hard work. However, it is not solely the result of my efforts alone. This dissertation would not have been possible without the support of many people that have accompanied me during this period, starts from Darmstadt and ends in Mannheim. Their guidance, encouragement, and, unwavering support have been instrumental in shaping my research, and I am deeply grateful for their contributions.

First and foremost, I would like to extend my sincere gratitude to all of my doctoral advisors (chronically). To start with, I would like to acknowledge and express my appreciation to Professor Michael Schneider. I am forever thankful to Michael for his constant encouragement and supervision throughout my academic journey, and I always owe him for his assistance. It was a great opportunity for me to learn from his expertise. The knowledge and skills I gained while working under Michael's supervision have been extremely valuable in my academic and professional development. I am also indebted to Professor Simon Emde and for his knowledge and thoughtful feedback that have helped me to improve my work and grow as a professional. Although our time working together was brief, he provided valuable insights that helped shape the direction of my research. His intelligence and experience have been an invaluable asset to my research and I have learned a great deal from working under Simon's supervision. I believe we were bad luck not to publish more when I was in TU Darmstadt, however, the knowledge I gained from Michael's and Simon's expertise proved to be indispensable. Last but not least, I am deeply grateful to Professor Raik Stolletz for his guidance throughout the course of this thesis. His understanding during the time when I struggled with the research was very useful, and it helped me to persevere and overcome the challenges that I faced. He has helped me to better structure my thoughts and approach my work with a clearer perspective. The strong motivation I received from him gave me the chance to make this thesis possible.

I was very lucky to meet wonderful people during my Ph.D. studies, starting from Darmstadt and ending in Mannheim. I would like to express my heartfelt gratitude to my wonderful friends for their companionship. Their friendship and positive influence have been a constant source of strength and motivation throughout this journey, especially my lovely friends (in no particular order) Hossein, Mohammad, Hamed, Luca, Matthew, Marius, Jannik, Katharina, Alison, Clemens, Santanu, Hala, Theresa, Chia-Yi Yen, Mengnan, Gia, Lenty, Swetlana, Emily, Johannes, Matteo, Eva, Ömer, Regina, Jonas, Emilia, Sarah, Justus, Syamak, Niloofar, Müge, Katrin, Dagmar, Sebastian, Simon, Mona, Biggie, Max, Maryam, Martin, Ivan, Hamid, Viviane, Georgie, Tobias, Konstantin, Ehsan, Mohammad N., Milad, Sarah V., Mr. Hahn, and many others. I am grateful for the laughter, fun, and good times we shared together, which helped to ease the stress and challenges. They have been my sounding board, my confidants, and my cheerleaders all rolled into one. I cannot imagine what this experience would have been like without them by my side. Their love and support have been invaluable and I am so grateful to have them in my life. I love you all.

Finally, I would like to express my deepest appreciation to my parents and my brothers (Mohammad and Omid) for their infinite love throughout my academic journey. Their belief in me and my abilities have been a constant source of motivation and has helped me to push through the challenging times. I am grateful for their sacrifices and the efforts they have made to ensure that I have had the resources necessary to complete my studies. I could not have done this without their support and I will be forever thankful for all that they have done for me. Thank you, Mom and Dad.

Mannheim, July 2022

Amir Foroughi

Contents

List of Figures

List of Tables

1. Introduction

Operations management is an area of business concerned with producing goods and services. It manages the conversion of inputs (such as materials, labor, and energy) into outputs (in goods and services). It ensures that business operations are efficient in using as few resources as required and effective in meeting consumer needs (Heizer et al., 2017). Operations management is responsible for a wide range of decisions, such as planning capacity for manufacturing plants, designing the structure of supply chain networks, and developing and operating service systems.

Decision-making in operations management is often reinforced by analytical and algorithmic tools which are based on logic and consider all available data and possible alternatives. This dissertation focuses on designing different algorithmic approaches supporting decision-making in operations management. It consists of three articles that design, implement, and analyze one or several algorithms for distinct practical problems in operations management. The papers are co-authored by Professor Dr. Schneider, Professor Dr. Emde, Professor Dr. Boysen, and Professor Dr. Stolletz. Each article describes the potential motivation of the research, positioning in the literature, and a detailed problem/model description. In addition, numerical results of each, generate insights into problems described in every chapter.

The first paper (Chapter 2) presents algorithmic approaches for addressing order picking in a warehouse setting. The practice of retrieving inventory products from their storage locations to fulfill customer orders is known as order picking (De Koster et al., 2007). Order picking is a critical factor for a supply chain's competitiveness since poor order picking performance (e.g., long delivery times) leads to customer discontent and high warehousing expenses (e.g., labor costs) (Wäscher, 2004). The focus of the first paper is high-density storage. The paper describes a storage setting where mobile racks are mounted on rails. A strong engine moves whole blocks of racks aside, opening an aisle and enabling access to a specific rack. Moving the heavy racks, takes a considerable amount of time and, consequently, strongly affects the total pick time. We formalize the resulting operational picker-routing problem, which aims at routing the picker such that as few aisles as possible need to be accessed and the total distance covered is minimal. We also address the interdependent tactical product-location problem, which deals with deciding on the storage locations of the items to be picked. We propose dynamic programming as an exact solution approach and several heuristics for both problems. In a comprehensive computational study, we apply these algorithms to explore the interdependencies between product location and picker routing and derive recommendations for suitable storage policies in a mobile rack warehouse. The computational tests demonstrate that the proposed heuristics for the routing problem perform pretty well. In addition, the priority rule proposed to address the product-location problem delivers better results than Chaotic storage policies, i.e., assigning random locations to SKUs.

Similar to the first Chapter, in the second one we also design an optimization algorithm for a practical application in operation management. The second article (Chapter 3) designs a solution procedure for the appointment scheduling problem. In many service systems such as outpatient clinics, customers must make an appointment before receiving services. Decision-makers set the appointments such that there is a tradeoff between customers' interest which is to have a short waiting time and servers' preference which is to have a short overtime (Gupta and Denton, 2008). The focus of the second paper is to make scheduling decisions while considering a system's priority rules. Appointment systems usually use a particular priority rule to control the patients' access waiting for being served when no emergency patients are waiting. We identify two classes of priority rules used in appointment systems: Static and dynamic priority rules. In static priority rules, the priority of patients does not change depending on the system's state; conversely, in a dynamic setting, the priority of a patient depends on the system's state. The decision is to optimally schedule outpatients if there are arrivals of emergency patients and inpatients, given that the system uses either of the priority rules mentioned above. We propose a simulation optimization approach based on tabu search with a neighborhood reduction technique to address this problem. The solution approach uses low-fidelity simulation to rank and pick neighboring solutions. The findings demonstrate the effectiveness and efficiency of the proposed algorithm in solving the outpatient scheduling problem. Furthermore, we study the structure of the optimal schedule of the outpatients when the system has different priority rules. The results suggest that the optimal solutions differ dependent on priority rules, and thus decision-makers should consider which priority rules the appointment system applies to schedule outpatients.

An appointment system can be represented with a time-dependent queueing system, as we see in Chapter 3. We used the simulation to evaluate the appointment scheduling problem's schedules (solutions) because of the lack of approximation methods. The last article (Chapter 4) proposes an approximation method to analyze time-dependent queueing systems; time-dependent queues are used to represent real-world systems, such as appointment systems in healthcare. In such systems, the rate of arrivals may surpass the service capacity for some time, leading to overloading. We analyze a single server time-dependent queue with exponentially distributed inter-arrival and service time and propose a hybrid approximation method based on the Stationary Backlog Carryover (SBC) and the Point-wise Stationary Fluid Flow Approximation (PSFFA) approaches. A mechanism is applied to adjust the parameters of the hybrid proposed approximation method when the system transfers from an overloaded period to an underloaded one or the other way around. A simulation benchmark confirms that the proposed hybrid method approximates the performance measures of the time-dependent queueing system for different parameter configurations accurately. Numerical experiments show the quality of the proposed approximation method in comparison with the SBC, the PSFFA, and the Fluid approximation approaches. The proposed hybrid method outperforms them for sinusoidal arrival functions and benchmark instances from the literature for queueing systems.

Chapter 5 discusses some concluding remarks for the dissertation as a whole. The references for all chapters are listed in a joint bibliography.

2. High-density storage with mobile racks: Picker routing and product location

Co-authors:

Nils Boysen
Friedrich-Schiller-Universität Jena, Lehrstuhl für Operations Management

Simon Emde
Technische Universität Darmstadt, Fachgebiet Management Science / Operations Research

Michael Schneider
RWTH Aachen University, Deutsche Post Chair – Optimization of Distribution Networks

Published in:

Journal of the Operational Research Society,
Volume 72 (2021), pages 535-553, DOI: 10.1080/01605682.2019.1700180

Abstract:

In response to the scarce space in many urban areas, high-density storage has become a widely discussed topic in warehousing research and practice. We investigate a storage setting where mobile racks are mounted on rails, so that a strong engine moves whole blocks of racks aside, opening an aisle and enabling access to a specific rack. Moving the heavy racks takes a considerable amount of time and, consequently, it strongly affects the total pick time. We formalize the resulting operational picker-routing problem, which aims at routing the picker such that as few aisles as possible need to be accessed and the total distance covered is minimal. Moreover, we address the interdependent tactical product-location problem, which deals with deciding on the storage locations of the items to be picked. We present suitable exact and heuristic solution procedures for both problems. In a comprehensive computational study, we

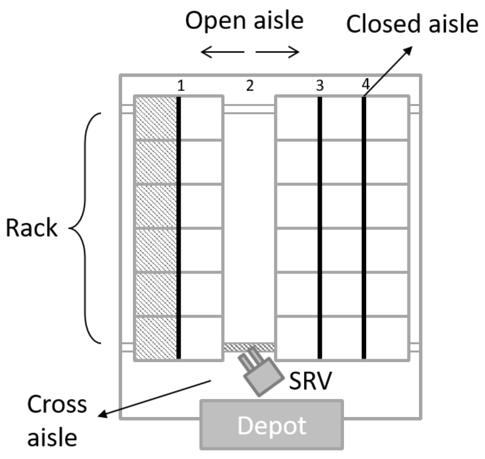
apply these algorithms to explore the interdependencies between product location and picker routing, and derive recommendations as to suitable storage policies in a mobile rack warehouse.

2.1. Introduction

Two recent trends in particular have made high-density storage a hot topic among both researchers and practitioners alike. First, the rising population density in many metropolitan areas has made space for roomy conventional warehouses, e.g., with single-deep storage racks each separated by a wide aisle, scarce and expensive. For instance, in Korea even underground food storage is explored due to the limited space (Choi et al., 2000). Furthermore, increasing ecological awareness of customers exerts pressure on supply chains to reduce their carbon footprints. Storing inventory more densely reduces the need for energy to heat, cool, light, and ventilate excess storage space. Especially in refrigerated warehouses, e.g., for frozen food, considerable savings in cooling costs of more than 30% can be realized by high-density storage according to the Material Handling Industry of America Material Handling Industry of America (MHIA) (2009) because less warehouse space needs to be cooled to refrigerate the same amount of product.

In this context, different systems like deep-lane storage (Stadtler, 1996), puzzle-based storage (Gue and Kim, 2007) or the so-called live-cube system (Zaerpour et al., 2015) have been investigated. A survey of these systems is provided by Azadeh et al. (2019). This paper treats an alternative high-density storage system based on mobile racks. Here, parallel racks are mounted on rails, so that a strong engine can move them sidewards. Because the length of the rails is only slightly longer than the total width of the racks, a sidewards movement is required to open an aisle for accessing the stock-keeping units (SKUs) of a specific rack as depicted in Figure (2.1a). Manually driven, e.g., with a star handle, mobile racks are often applied for high-density storage of rarely accessed documents in archives or libraries (see Figure 2.1b). However, engine-driven mobile shelving accessed with some human-operated (man-aboard) or un-manned storage-and-retrieval vehicle (SRV) are, for instance, often applied in refrigerated warehouses for frozen food, flowers, or pharmaceutical products.

Because fully-loaded racks may become very heavy, it takes a considerable amount of time to move the racks and open an aisle. In a specific system implementation (for instance, see (Schäfer, 2018)), the racks move only with 4 m/min, which may cause considerable waiting time for the SRV having a much higher velocity of 115 m/min. Thus, the effort of order picking is mainly influenced by the number of aisle changes of the SRV. To keep this effort low, it seems promising to apply a shared storage policy, which allows storing unit loads of the same SKU in multiple locations (Bartholdi and Hackman, 2017). In this way, frequently requested SKUs can be stored in multiple racks, so that the probability of having to move the aisle when retrieving a specific picking order is reduced. Clearly, there is a basic trade-off between the total space re-



(a) schematic warehouse layout

(b) mobile shelving in archives

Figure 2.1.: Mobile shelving[1]

quirement, which increases with a higher multiplicity of SKUs, and the resulting picking effort, which is reduced the more often SKUs are placed in different racks. It is the basic aim of this paper to explore this trade-off in our mobile rack setting. To do so, the following two interdependent decision problems need to be solved (for a survey on interdependent decision making in warehouses see (van Gils et al., 2018)):

- In a pick-by-order environment, *order picking* is the short-term task of successively retrieving all items of a picking order by moving the SRV between the locations the respective SKUs are stored in and transporting them to the depot (or I/O point). In our specific setting, where products may be stored in multiple locations, the main decision is whether a specific aisle is opened, and which SKUs of the accessed aisle should be retrieved. The optimization goal of this operational problem is to reduce the total pick time, consisting both of the time for opening aisles and travelling to storage locations.
- Over a mid-term planning horizon, it has to be decided where each SKU is located: the so-called *product-location problem* deals with assigning SKUs to storage positions. To avoid frequent rack movements when picking orders, SKUs frequently ordered together should be stored in racks accessible via the same aisle. Furthermore, our shared storage policy requires to simultaneously define the number of storage locations that each SKU is allowed to occupy. The goal of this tactical optimization problem is to allocate space to SKUs such that the (anticipated) routing problem becomes easier.

[1] Picture 2.1b is published under the Creative Commons License and its author is AndreasPraefcke.

Besides modelling and solving these two optimization problems, we also aim to derive some general insight into what storage policies are particularly promising for mobile rack warehouse operators who want to improve pick times.

After a review of the related literature (Section 2.2), we formalize both decision problems for our mobile rack setting and provide different exact and heuristic solution procedures. Sections 2.3 and 2.4 are dedicated to the picker-routing and product-location problem, respectively. In Section 2.5, we explore the computational performance of the developed algorithms and investigate the basic trade-off between space requirement and picking effort depending on different levels of product multiplicity. Finally, Section 2.6 concludes the paper.

2.2. Literature review

Numerous papers on warehousing in general and on optimizing the routing of pickers and the location of products in racks in particular exist. Instead of trying to summarize this vast body of literature, we refer to the survey papers published on these topics over the past years, e.g., (de Koster et al., 2007; Gu et al., 2007, 2010; Boysen et al., 2019). To emphasize the importance of simultaneously planning picker routes and product locations, we refer to a recent case study at a distribution center for alcoholic beverages in Canada: Renaud and Ruiz (2008) calculate an impressive reduction rate of 27% on the total picking effort when addressing both problems jointly.

In the following, we only survey those studies, which share at least one of the four elementary characteristics which mainly affect the structure of our two optimization problems: (i) the application of mobile racks, (ii) a shared storage policy, (iii) the product location for reducing the picking effort of a given, deterministic order set, and (iv) interdependent decision making in warehouses combining multiple decision tasks.

i) Literature on mobile rack storage is rare. This is probably due to the low performance gains promised by computerized decision support for manual mobile racks like they are often applied in libraries (see (Russon et al., 1982)). However, in recent years, more and more automated mobile rack systems have been erected mainly for refrigerated warehouses (see (Schäfer, 2018)). To the best of our knowledge, the only three papers having an operational research focus on automated mobile racks with an SRV are the papers of Chang et al. (2007), Hu et al. (2009), and Boysen et al. (2017). The former two studies also treat picker routing for isolated picking orders and make similar assumptions to ours. They allow only a one-sided access of the SRV into an aisle, which leads to routing solutions with a fairly simple structure. On the other hand, neither a shared storage policy nor the product-location problem is treated, and only very basic priority-rule-based heuristics are presented. Given that only one-sided access to the aisles is possible and the conventional policy of storing each SKU at a single storage position is used, the picker-routing problem is trivial. To formulate non-trivial problems, Chang et al. (2007) assume that a product stored in a rack is accessible from both neighboring aisles, and Hu et al. (2009) additionally introduce a middle (cross-)aisle. Unlike our paper, the study of Boysen et al. (2017) does not consider the routing of an isolated order, but treats the scheduling of successive orders. Their aim is to reuse aisles left open when finishing the previous order and starting the next one.

ii) A shared storage policy allows SKUs to be stored at and picked from multiple locations. This policy receives considerable attention in real-world warehousing, because online retailer Amazon applies an extreme form of this policy in many of its distribution centers. They break down unit loads and spread isolated items all over the shelves of their warehouse. This policy is also denoted as mixed-shelves or scattered storage and especially suited if orders demand just a few items, e.g., in business-to-consumer online retailing (Boysen et al., 2019; Weidinger and Boysen, 2018). In refrigerated warehouses where mobile racks are predominately applied, however, the low temperatures require SRV support of pickers to quickly handle SKUs. SRVs are, typically, not able to handle isolated items but only complete unit loads (e.g., pallets), so that they are kept together in mobile rack warehouses. Here, shared storage is realized by storing multiple unit loads of the same SKU in multiple storage positions that are accessible via different aisles. Shared storage has rarely been treated in previous research. Daniels et al. (1998), for instance, were the first to consider the picker-routing problem if multiple locations per SKU are available. They consider traditional warehouse layouts having fixed racks and multiple cross aisles, so that the problem is modeled as an enriched traveling salesman problem. The product-location problem is also combined with shared storage. However, previous research either considers automated storage and retrieval systems (ASRS, i.e., high-bay racks accessed with a crane) (Goetschalckx and Ratliff, 1990; Kulturel et al., 1999; Chen et al., 2010) or scattered storage (Weidinger and Boysen, 2018; Weidinger et al., 2019; Boysen et al., 2019). Shared storage for mobile rack systems has neither been considered for picker routing nor for the product-location problem.

iii) With regard to the product-location problem, it is clearly advantageous to store products often ordered jointly at nearby positions to reduce the necessity of moving racks out of the way and to reduce the travel distance when picking the orders. However, the problem of anticipating reliable (short-term) picking orders when planning the product locations over a mid-term planning horizon remains. In general, there are three alternatives of how to deal with this problem: (i) In highly volatile environments, no information on SKUs ordered jointly may be available, so that only product specific information is at hand. In this case, for instance, the famous cube-per-order index (see (Heskett, 1963)) can be applied, to store frequently demanded products closer to the depot. (ii) Given enough adequate historical data, product correlations defining the probability of a product pair being jointly ordered may be obtainable (see (Frazelle and Sharp, 1989; Brynzér and Johansson, 1996)). These correlations allow for a grouping of product families, so that similar SKUs can be located in the same region of the storage area. In this study, we follow van Oudheusden and Zhu (1992), Boysen and Stephan (2013) and Kress et al. (2017) and (iii) presuppose deterministic knowledge on the order set to be retrieved. For instance, this setting arises in intermediate warehouses where materials and parts need to be picked for recurrently manufactured production lots (Boysen and Stephan, 2013). Note that even in a truly stochastic environment, the deterministic problem can be used in a scenario-based approach. To explore the limits of our deterministic approach, we will investigate the robustness of our results in the face of forecast errors in Section 2.5.

iv) Our research combines the product-location problem, which decides on where to store the SKUs, and picker routing, which decides on the way of a picker through the warehouse. The recent survey paper of van Gils et al. (2018) identifies interdependent decision making as a very promising avenue for future research and reviews the existing combined models in the warehousing literature. They identify 24 papers also combining product location and picker routing. However, the vast majority of them combine these two problems in traditional picker-to-parts warehouses with stationary racks (e.g., (Theys et al., 2010; Chan and Chan, 2011)); a mobile rack warehouse has not been treated from this perspective. The main distinction to the traditional warehouse settings is that large parts of the picking effort is caused by the rack movement, so that picker and racks need to be synchronized both in product location and picker routing.

It can be concluded that our mobile rack setting combining all four aforementioned characteristics constitutes a novel application, which has yet not been investigated.

2.3. Picker routing

2.3.1. Problem definition

In our mobile rack setting, we have shelves storing SKUs, which are only accessible if the racks are moved such that the adjacent aisle is opened. We assume that there is only enough space for a single open aisle at a time. Therefore, the total set $J = \{1, \ldots, n\}$ of SKUs can be subdivided into m subsets $J_i \subset J$ defining the products stored in those two racks accessible via an opened aisle $i = 1, \ldots, m$. Note that for a single-picker routing problem, only those SKUs occurring in the current pick list are relevant. Without loss of generality, we assume that the parallel racks are arranged from left to right with the (potential) aisles numbered in increasing order, while the SKUs $J_i = \{j_1^i, \ldots, j_{|J_i|}^i\}$ in each aisle i are numbered in non-decreasing distance from the cross-aisle, i.e., the SRV has to move no farther up aisle i to retrieve SKU j_k^i than to retrieve SKU j_{k+1}^i, $\forall k = 1, \ldots, |J_i| - 1$. The depot where each tour of the picker starts and ends is located somewhere along the cross aisle, as depicted in Figure (2.1a). Furthermore, we assume that the aisles are only accessible by the SRV from the side facing the depot, so that we have only a single cross aisle at the front, i.e., the picker has to enter and leave each aisle via the same entrance point (also denoted as the "out and back" approach or the "return policy"). Especially in refrigerated warehouses, where space and energy reduction is a pressing concern, making do with the least number of cross aisles is a widespread layout choice.

For this warehouse setting, we investigate a picker-to-parts, pick-by-order organization of the picking process. This means that a picker receives a picking order at the depot, successively moves to rack positions where the SKUs defined on the current list are retrieved, and finally returns to the depot where the items are further processed. In our setting, opening a specific aisle takes considerable time, so that a shared storage policy is applied. This means that SKUs of the same type may be stored in multiple aisles, so that there is a higher probability of SKUs ordered together being jointly accessible via the same aisle without requiring a rack move. We assume that each location where a unit load, e.g., a pallet, of a specific SKU is stored contains enough units to satisfy any demanded number of products defined in a picking order. On the one hand, this allows us to exclude special cases (which considerably complicate the description) where multiple product locations of a specific SKU need to be accessed in order to gather enough items in exceptionally high demand. On the other hand, we can define a picking order as a set of different SKUs $P \subseteq J$ to be retrieved without having to record the respective number of units demanded.

Given a specific storage layout (defined by the sets J_i of SKUs per aisle $i = 1, \dots, m$), our picker-routing problem aims to minimize the total retrieval time for a given picking order P. Because the pick time to retrieve the defined number of items once the picker has reached a specific shelf does not depend on the routing decision there remain three parts to be considered:

- The waiting time w of the SRV while an aisle is opened: In the most widespread system implementation, there is some mechanism, e.g., a light barrier, button or ripcord, at the front of each rack to initiate the rack movement and open the neighboring aisle. In this setting, the SRV has to wait for w time units once it reaches its target rack. As a matter of convenience, we will not consider the position of the open aisle at the beginning of the routing, so that each aisle movement requires w time units. Certainly, this is a good approximation of reality if there are many racks and few items on the pick list, because then the probability that an item can be retrieved from the currently open aisle becomes negligible. Note that fully automated rack shifts that are automatically synchronized with SRV movements are rare in current industrial practice although they are technically possible. In such an automated system, a rack move is initiated once the SRV leaves its previous aisle. Consequently, a waiting time only occurs if w is larger than the time it takes the SRV to move from the previous rack to the current one. As the alterations required for adapting our solution procedures to this setting are quite straightforward, we abstain from a detailed description.
- The driving time of the SRV in an aisle: If we assume a constant velocity of the vehicle, it is the one SKU to be retrieved stored farthest from the depot that determines the driving effort per aisle. Thus, if d_{ji} defines the time to drive from the frontend of aisle i to the storage location of SKU j (stored in this aisle), then the time in aisle i to receive a subset of items $P' \subseteq P \cap J_i$ from this aisle requires the travel to and from the farthest location, i.e., $2 \cdot \max_{j \in P'}\{d_{ji}\}$. Note that we assume that vertical movement times (if any) are negligible.
- The driving time of the SRV in the cross aisle: The routing aspect in our picker-routing problem is rather trivial to solve. In the cross aisle, it is obviously an optimal policy to start with the aisle farthest left and then successively visit all other aisles from left to right until the aisle farthest right is reached from which the SRV travels back to the depot. Let A be the set of aisles to be accessed, Δ_i the driving time between a specific aisle and the depot, and $\delta_{ii'}$ the driving time between two aisles i and i', then the total driving time of the SRV in the cross aisle amounts to $\Delta_{\underline{i}} + \delta_{\underline{i},\overline{i}} + \Delta_{\overline{i}}$ with $\underline{i} = \min\{i \in A\}$ (leftmost aisle) and $\overline{i} = \max\{i \in A\}$ (rightmost aisle).

Given these explanations, our specific picker-routing problem can be formally defined as follows. A route Ω is a set of tuples $(i,S) \in \Omega$, defining the set S of SKUs to be retrieved from the racks accessed via aisle i. We say Ω is feasible if

1. for each $i = 1,\dots,m$, there is at most one $(i,S) \in \Omega$, i.e, each aisle is accessed either once or not at all,
2. for each tuple $(i,S) \in \Omega$, we have $S \subseteq J_i$, i.e, a set S of SKUs can only be retrieved from an aisle i if these SKUs are actually available there,
3. for each pair of tuples $(i,S) \in \Omega$ and $(i',S') \in \Omega$ with $i \neq i'$, we have $S \cap S' = \varnothing$, i.e., each SKU is retrieved from a unique shelf (and not via multiple aisles), and
4. we have $\bigcup_{(i,S)\in\Omega} S = P$, i.e., all items of picking order P are retrieved.

Among all feasible routes, we seek one Ω which minimizes the total picking time $T(\Omega)$ defined as follows

$$T(\Omega) = w \cdot |\Omega| + \sum_{(i,S)\in\Omega} 2 \cdot \max_{j\in S}\{d_{ji}\} + \Delta_{\underline{i}} + \delta_{\underline{i},\bar{i}} + \Delta_{\bar{i}}, \tag{2.1}$$

with the leftmost aisle $\underline{i} = \min_{(i,S)\in\Omega}\{i\}$ and the rightmost aisle $\bar{i} = \max_{(i,S)\in\Omega}\{i\}$. The first, second, and third term define the time required for moving the racks, moving inside the aisles, and between aisles, respectively.

Note that to reduce our problem setting to the most relevant core, i.e., the rack movement and its synchronization with the picker movement, we exclude further warehousing peculiarities that may extend the problem setting in real life. For instance, interdependencies among pickers, e.g., blocking each other in narrow aisles (Hong et al., 2012) or the assignment of order batches to pickers (Matusiak et al., 2014), are excluded. Otherwise a much more complex problem setting having to plan multiple pickers in parallel would arise. Further note that energy savings when moving the heavy racks can also be an important optimization objective. Since, however, large parts of our picking effort are caused by the movement of racks, we indirectly also support this aim with our objective.

Example: Consider a picking order $P = \{A,B,C,D\}$ to be retrieved from a mobile rack setting with three aisles as is depicted in Figure 2.2. Given the travel times defined there and waiting time $w = 2$ for opening an aisle, the left-hand and right-hand solution with $\Omega = \{(1,\{A,B\}),(3,\{C,D\})\}$ and $\Omega' = \{(1,\{B\}),(2,\{A,C,D\})\}$ amount to $T(\Omega) = 26$ and $T(\Omega') = 21$, respectively.

Theorem 1. *Our picker-routing problem is strongly NP-hard.*

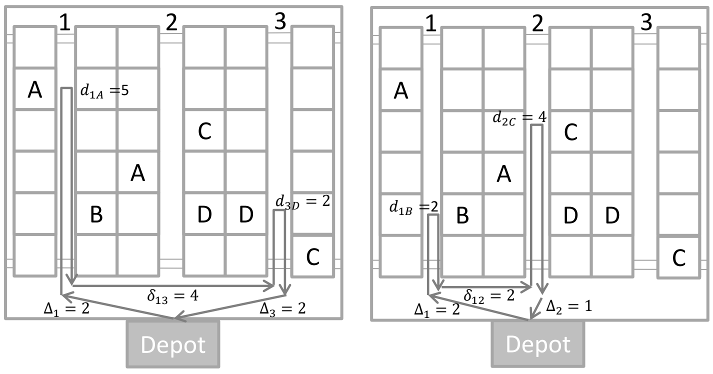


Figure 2.2.: Two alternative solutions for the example

Proof. If we consider an infinite velocity of the SRV, so that the total picking time is only determined by the number of aisles accessed, then the transformation from the minimum set-covering problem, which is well known to be strongly NP-hard (see (Garey and Johnson, 1979)) is readily available. Given a collection C of subsets of a finite set Γ, the minimum set-covering problem aims at a subset $C' \subseteq C$ such that every element in Γ belongs to at least one member of C', and $|C'|$ is minimal. We simply have to transfer all subsets $c \in C$ to a unique aisle i with stored items $J_i = c$, set $P = \Gamma$, and we have a one-to-one mapping of both problems. $\square$

2.3.2. Algorithms for the picker-routing problem

In this section, we present a dynamic programming (DP) scheme, which can be applied to solve our picker-routing problem to optimality. However, due to the NP-hard nature of the problem, it is unlikely that large instances can be solved to optimality in a reasonable amount of time. Therefore, we also extend the DP to a heuristic beam search (BS) approach.

Our DP is divided into $m + 1$ stages, each stage $i = 0, \ldots, m$ containing states (Q, i), where $Q \subseteq P$ and $i \in \{0, \ldots, m\}$. Set Q contains the remaining SKUs still to be retrieved and i is the last aisle visited by the SRV. The initial state is $(P, 0)$, because initially all items listed in the picking order are still to be picked, and the SRV is in the leftmost dummy aisle 0. A transition from state (Q, i) to successor states (Q', i') implies that the SRV drives from aisle i straight to aisle i', where it picks the SKUs $Q \setminus Q'$. Such a transition exists if the following conditions hold:

- $i' > i$, i.e., aisles are explored from left to right, and the SRV does not take "zigzag" routes.
- $Q' = Q \setminus R_k^{i'}$, where $R_k^{i'} = \{j_l^{i'} : l = 1, \ldots, k\}$, for some $k \in \{1, \ldots, |J_{i'}|\}$. $R_k^{i'}$ is the set of SKUs in aisle i' up to the storage position of the SKU that is the

k-farthest from the depot in that aisle. Note that we consider the SKUs of an aisle according to non-decreasing distance from the depot, i.e., $j_1^{i'}, \dots, j_{|J_{i'}|}^{i'}$ (without loss of generality, we assume that $d_{j_k^{i'},i'} \leq d_{j_{k+1}^{i'},i'}$, $\forall k = 1, \dots, |J_{i'}| - 1$, holds). Therefore, if the SRV traverses aisle i' until the position of SKU $j_k^{i'}$, it can pick all the SKUs on the way, i.e., $j_1^{i'}, \dots, j_k^{i'}$.

- $Q' \subseteq \bigcup_{i''=i'+1}^{m} J_{i''}$ because a successor state (Q', i') and the corresponding transition from predecessor state (Q, i) should only be generated if the remaining SKUs Q' can actually be picked from the remaining aisles.

The partial objective value $f((Q, i), (Q', i'))$ associated with a transition $((Q, i), (Q', i'))$ from predecessor state (Q, i) to successor state (Q', i') is calculated as follows:

$$f((Q,i),(Q',i')) = \begin{cases} \Delta_{i'} + w + 2 \cdot \min_{k=1,\dots,|J_{i'}|}\{d_{j_k^{i'},i'} : Q' = Q \setminus R_k^{i'}\} & \text{if } Q = P \\ \delta_{i,i'} + w + 2 \cdot \min_{k=1,\dots,|J_{i'}|}\{d_{j_k^{i'},i'} : Q' = Q \setminus R_k^{i'}\} & \text{otherwise,} \end{cases}$$

where the top equation is relevant only for the first transition when no item has been picked yet, i.e., $Q = P$. In this case, the travel time from the depot to the first visited aisle $\Delta_{i'}$ must be taken into consideration. For all other transitions, the bottom expression is relevant.

In this way, we generate transitions and successor states from any state previously generated and for any SKU $j_1, \dots, j_{|J_{i'}|}$ successively reached in current aisle i'. Clearly, identical states must not be duplicated, so that the respective transition has to be redirected to the existing state. Let $\Gamma(Q', i')$ be the set of states from which a transition to state (Q', i') exists, then the Bellman equation calculating the picking time $Z(Q', i')$ for state (Q', i') is defined as follows:

$$Z(Q', i') = \min_{(Q,i) \in \Gamma(Q',i')} \{Z(Q, i) + f((Q, i), (Q', i'))\}.$$

If we initialize start state $(P, 0)$ with $Z(P, 0) := 0$, then solving our picker-routing problem is equivalent to determining the final state $(\varnothing, i^*)$ with $i^* = \operatorname{argmin}_{i=1,\dots,m} \{Z(\varnothing, i) + \Delta_i\}$, and the optimal solution can be determined by a simple backward recovery along the optimal path to that state.

Example (cont.): For our example in Figure 2.2, the resulting DP graph is depicted in Figure 2.3. The bold faced optimal path represents the right-hand solution within Figure 2.2 and, thus, results in a total picking effort of $Z(\varnothing, 2) + \Delta_2 = 20 + 1 = 21$.

The total number of states in our DP is in $\mathcal{O}(2^n \cdot m)$, and the number of transitions is bounded by $\mathcal{O}(m^2 \cdot n \cdot 2^n)$ because the total number of SKUs n is an upper bound of the

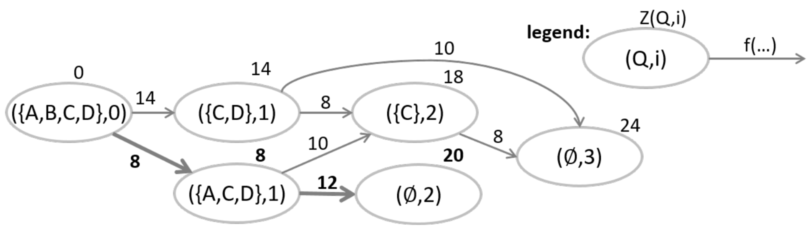


Figure 2.3.: DP graph for the example

SKUs stored per aisle. Because the effort per transition is in $\mathcal{O}(1)$ time, our DP runs in $\mathcal{O}(m^2 \cdot n \cdot 2^n)$ time.

The runtime of the algorithm should be acceptable for small problems with few racks and short pick lists. However, for larger instances, we extend the DP scheme to a beam search (BS) heuristic in the following.

2.3.3. Beam search

BS is a search technique which heuristically estimates the θ best nodes on each stage of a search tree. Only those best nodes are further developed on each stage; all others are fathomed, making the solution method fast but approximate. This heuristic was first used in artificial intelligence for the speech recognition problem by Lowerre (1976). Since then, many applications of BS have been reported in the literature.

In this paper, we turn our DP into BS by calculating the lower bound for each node of each stage of the DP graph and pruning the ones whose calculated costs are not among the θ lowest. The lower bound consists of two terms: the first calculates the minimum number of racks to be moved by dividing the number of remaining SKUs to be picked by the maximum number of items from the picklist on the racks yet to be visited. The second calculates the minimum distance that the SRV must travel to pick all the remaining items by adding the distance to the SKU yet to be picked that is farthest from the current aisle, i.e.,

$$LB(Q,i) = w \cdot \left\lceil \frac{|Q|}{\max_{i'=i+1,\dots,m}\{|J_{i'}|\}} \right\rceil + \max_{j\in Q}\left\{ \min_{i'=i+1,\dots,m: j\in J_i} \{\delta_{i,i'} + 2\cdot d_{ji'} + \Delta_{i'}\} \right\}.$$

In each stage, we add the calculated lower bound to the current cost of the node at that stage and then keep the θ best nodes and discard the others.

To make the algorithm more efficient, we also use a modified version of DP and BS by adding bounds. In this setting, first we obtain an upper bound by solving the problem by using the set-covering heuristic described below. Then, in the course of search, we use the upper bound to prune nodes whose lower bound is not less than that upper bound.

Example (cont.): Consider the DP graph in Figure 2.3. In state $(\{C, D\}, 1)$, the lower bound is

$$LB(\{C, D\}, 1) = 2 \cdot \lceil 2/\max\{3, 2\} \rceil + \max\{\min\{11, 8\}, \min\{7, 10\}\} = 2 + 8 = 10,$$

where the first summand 2 denotes the minimum effort to move racks, and the second summand 8 is the minimum travel time of the SRV to retrieve the item farthest from the current position (aisle 1) and return to the depot. In this example, SKU C is the farthest: the quickest way to retrieve it is from the first slot of aisle 3, which would incur a total travel time of 8 (including the return trip to the depot). Seeing that it already took an effort of $Z(\{C, D\}, 1) = 14$ to pick items A and B, no schedule reached from state $(\{C, D\}, 1)$ can have a lower objective value than $Z(\{C, D\}, 1) + LB(\{C, D\}, 1) = 14 + 10 = 24$. If an upper bound of 24 or better is already known, the state can be fathomed.

2.3.4. Set-covering heuristic

Apart from the DP-based schemes described above, we also propose a heuristic that makes use of the similarity of our problem to set covering (see Theorem 1). We dub this approach the set-covering heuristic (SCH). The rationale is the following: we assume that the rack movement time dominates the travel time of the SRV, and therefore, getting a solution with the minimum number of aisles to open should be close to optimal. If each aisle is considered as a set of SKUs, covering order set P with as few rack sets J_i as possible also minimizes the number of rack moves. Moreover, to take the travel time into consideration, the procedure can be started multiple times, where each set gets a different weight every time based on the distance from the cross aisle to a randomly selected SKU in the respective aisle, plus a small fraction (1%) of the distance from the depot to the aisle. The resulting problem is a weighted set-covering problem (WSCP). More precisely, we first randomly choose a SKU $j_i^* \in J_i$, $\forall i = 1, \ldots, m$. Then, we set the weight of set i in the objective function of the WSCP to $2 \cdot d_{j_i^*,i} + 0.01 \cdot \Delta_i$. The first part equals the time it takes for the SRV to reach the randomly chosen SKU from the cross aisle. The second part is used to differentiate the aisles which are close to the depot from those which are farther. This way, aisles in which SKU j_i^* is close to the entrance and aisles which are closer to the depot receive a lower weight and are thus preferred in the set covering solution. By randomly choosing j_i^*, we ensure that the weights (and solutions) are different every time the procedure is executed, encouraging heuristic exploration. To this end, we restart this procedure 1000 times, each time with different random weights, i.e., random j_i^*, and finally return the best found solution. Note that 1000 iterations turned out to yield a good compromise between runtime and solution quality in preliminary tests.

Although the WSCP is NP-hard, there are several algorithms and tools which can solve it quite efficiently for practical problem sizes, see, e.g., (Lan et al., 2007). We solve the problem using the commercial MIP solver CPLEX 12.6 in its default setting. The resulting set-covering solution dictates which aisles are to be visited to retrieve which SKUs. It is thus easy to derive a corresponding schedule Ω.

Example (cont.): Using SCH on the example in Figure 2.2, for each aisle, a set is created which contains the items in the respective aisle, i.e., there are 3 sets in total. In each aisle, a SKU is randomly selected; the chosen SKUs in the example are marked by a star sign in Figure 2.4. Then, each set is assigned a weight by calculating the distance which the SRV has to cover to retrieve the selected SKU from the cross aisle, i.e., twice the distance from the cross aisle to the selected SKU. In addition, to take the distance from the depot into consideration, SCH adds $0.01 \cdot \Delta_i$ to the weight of each set. In this example, the first set has a weight of 4.02 units, the second has a weight of 8.01 units, and finally the last one has 2.02 units of weight. In the next step, a weighted set-covering model is created with the corresponding weights for each set. The following 0-1 integer linear model is generated for this example:

$$\min z = 4.02 \cdot x_1 + 8.01 \cdot x_2 + 2.02 \cdot x_3$$

$$\begin{aligned}
\text{s.t.:} \qquad x_1 + x_2 &\geq 1 && \text{for SKU } A\\
x_1 &\geq 1 && \text{for SKU } B\\
x_2 + x_3 &\geq 1 && \text{for SKU } C\\
x_2 + x_3 &\geq 1 && \text{for SKU } D\\
x_i &\in \{0,1\} && \forall i = 1,2,3,
\end{aligned}$$

where x_i is the decision variable to use set (aisle) i. In the optimal solution, $x_1 = x_2 = 1$ and $x_3 = 0$, which means that SKU B is picked from aisle 1 and SKUs D, A, and C are picked from the second aisle, i.e., $\Omega = \{(1, \{B\}), (2, \{A, C, D\})\}$. This corresponds to the solution depicted on the right-hand side of Figure 2.2 with an objective value of $T(\Omega) = 21$.

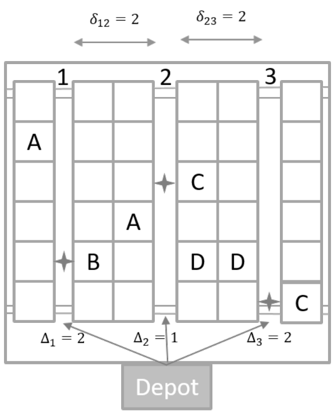


Figure 2.4.: SCH in the example; the SKUs which are selected randomly are marked with a star.

2.4. Product location

2.4.1. Problem description

The picker-routing problem is obviously heavily dependent on the location of the SKUs to be picked: If the SKUs to be accessed are in the most remote parts of the warehouse, then even the best route will take a long time to complete, and a lot of unnecessary rack moves may be incurred. A clever SKU placement, on the other hand, can help to significantly reduce the operational picking and rack moving effort later on. Today, many warehouses that employ a shared or scattered storage policy use an entirely random assignment of SKUs to racks (online retailers like Amazon are a prime example, see Weidinger and Boysen (2018)). In this section, we investigate the mid-term product-location problem in a mobile-rack warehouse with the goal of deriving a better storage policy than purely chaotic storage.

The product-location problem deals with assigning SKUs to storage positions. In our specific case a SKU of a single type can be stored in multiple storage positions, so that we additionally have to select how often each SKU should be stored without exceeding the available storage space. The aim of the product-location problem is to assign SKUs such that the effort for processing picking orders is minimized. Unfortunately, exact information on the (short-term) picking orders to be retrieved is typically not available when assigning the storage locations over a mid-term horizon. Nonetheless, it may be possible to assemble a representative deterministic set of orders O, e.g., from historical data, and aim to place the SKUs such that the total picking effort for retrieving set O is minimized. Certainly, in many industries it is hard to anticipate a representative order set, so that set O and the resulting product-location plan will be prone to forecast errors. We will investigate the robustness of our deterministic approach in the face of uncertain picking orders in Section 2.5.

Furthermore, we presuppose a unit-load warehouse, i.e., the given mobile rack system provides a given number of storage positions, e.g., for pallets, at which the given set of SKUs $J = \{1, \dots, n\}$ can be stored. Thus, each SKU can be stored at any storage position. Let $S(i)$ be the set of storage positions in aisle $i = 1, \dots, m$. Because we consider a shared storage policy, we have $|\bigcup_{i=1}^{m} S(i)| \geq n$. We seek an assignment function $g : S(i) \rightarrow \{1, 2, \dots, n\}, \forall i = 1, \dots, m$, i.e., an assignment of SKUs to storage positions, such that each SKU receives at least one position, that is,

$$\left| \left\{ u \in \bigcup_{i=1}^{m} S(i) : g(u) = j \right\} \right| \geq 1 \quad \forall j \in J,$$

must hold, and the total picking effort for processing all picking orders $o \in O$ is minimized.

Let $\Omega^*(o, g)$ be the optimal schedule for processing order o given the storage plan defined by assignment function g. We seek a feasible assignment function g that minimizes objective value $W(g)$:

$$W(g) = \sum_{o \in O} T(\Omega^*(o, g)),$$

where $T(\Omega^*(o, g))$ is the optimal total picking effort for processing order o given location assignment g (see Equation (2.1)).

Example: Consider the example depicted in Figure 2.5, where $n = 8$ SKUs have to be located in 12 storage positions accessible via $m = 2$ aisles. If the waiting time for opening aisles amounts to $w = 3$, then the total picking effort for retrieving the given order set O given the depicted product-location plan amounts to $W(g) = 58$. More precisely, the first order set includes $\{A, B, D\}$, which is retrieved from the first aisle, i.e., $\Omega^*(\{A, B, D\}, g) = \{(1, \{A, B, D\})\}$. The picking time to retrieve this order set is the sum of the waiting time to open aisle 1, the picking effort from the cross aisle, and the travel time it takes for the SRV to travel from the depot to the aisle and back, which is $T(\Omega^*(\{A, B, D\}, g)) = 9$ (see Equation (2.1)). Similarly, the picking time of the others are $T(\Omega^*(\{B, F\}, g)) = 11$, $T(\Omega^*(\{D, C\}, g)) = 9$, $T(\Omega^*(\{E, G, H\}, g)) = 9$, $T(\Omega^*(\{F, H\}, g)) = 9$, and $T(\Omega^*(\{A, G, H\}, g)) = 11$.

Intuitively speaking, the product-location problem also seems to be strongly NP-hard because it contains our picker-routing problem as a subproblem. However, with the additional flexibility of selecting the storage locations of SKUs, it may be possible that

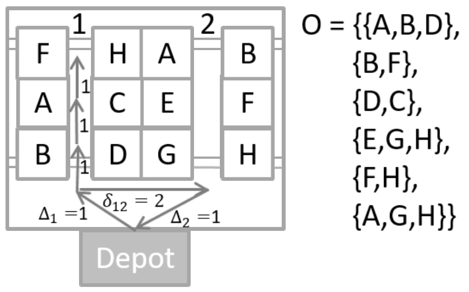


Figure 2.5.: Example for the product-location problem

only trivial routing problems remain. However, this is in fact not the case, as is shown by Theorem 2.

Theorem 2. *Our product-location problem is strongly NP-hard.*

Proof. Placing unit loads in racks such that the picking effort for a deterministic order is minimized has been shown to be strongly NP-hard by Boysen and Stephan (2013) even if only a single aisle exists, and a shared storage is not allowed. Clearly our problem setting with multiple aisles operated under a shared storage policy is a generalization of this problem and, thus, also strongly NP-hard. □

2.4.2. Policies

We investigate two product-location policies: chaotic storage and a priority rule. As mentioned earlier, chaotic storage is a storage policy in which positions are randomly assigned to each SKU. Usually, this policy is implemented as a shared storage policy.

Our priority-rule-based approach, outlined in Algorithm 1, is built on the following considerations. First, SKUs that are often ordered together are put in the same aisle. Second, the most frequently ordered SKUs should be kept close to the cross-aisle and to the depot.

Algorithm 1 Priority rule to locate products

```
 1: procedure PRIORITY_RULE_HEURISTIC ($O$)
 2:     for each aisle $i$ do
 3:         for each potential location $h \in S(i)$ do
 4:             $C(i,h) \leftarrow \Delta_i + d_{hi}$
 5:     $S^* \leftarrow$ Sort($\bigcup_{i=1}^{m} S(i)$)                ▷ based on $C(i,h)$, non-decreasing
 6:     $J^* \leftarrow$ Sort($J$)           ▷ based on their order frequency, non-increasing
 7:     $J' \leftarrow J^*$                                  ▷ SKUs yet to be assigned
 8:     for $h \in S^*$ do
 9:         $i^* \leftarrow$ aisle that location $h$ belongs to
10:         if no SKU has yet been assigned to aisle $i^*$ then
11:             $g(h) \leftarrow j^*$, such that $j^*$ is the most frequently ordered SKU in $J'$
12:         else
13:             for $j \in J'$ do
14:                 $G_j^* \leftarrow |\{(h',o) \mid g(h') \in o \wedge j \in o, h' \in S(i^*), o \in O\}|$      ▷ number of
    times the SKUs stored in aisle $i^*$ and SKU $j$ are ordered together
15:             $g(h) \leftarrow \underset{j \in J'}{\operatorname{argmax}}\{G_j^*\}$
16:         $J' \leftarrow J' \setminus \{j\}$
17:         if $J' = \varnothing$ then
18:             $J' \leftarrow J^*$
19:     End procedure
```

In the first loop, each location j in each aisle i is associated with a cost $C(i,j)$. This cost is the time it takes the SRV to travel to location j in aisle i starting from the depot. We first sort the slots non-decreasingly based on the calculated cost, i.e., locations closer to the depot have more priority. SKUs are sorted depending on their order frequency (in representative order set O) and are kept in J^* (Line 6). For each SKU in J', we calculate G_j^* and then put the SKU with the largest G_j^* in the next best position of S^* (lines 8-13). G_j^* is the number of times SKUs from the current aisle i^* are ordered jointly in the current order j in the given order set O.

Effectively, we put the most frequently ordered items at the nearest positions to the depot and then remove them from J'. Finally, if there is no SKU left in J', the set is refilled, and the search procedure continues until all storage locations are filled.

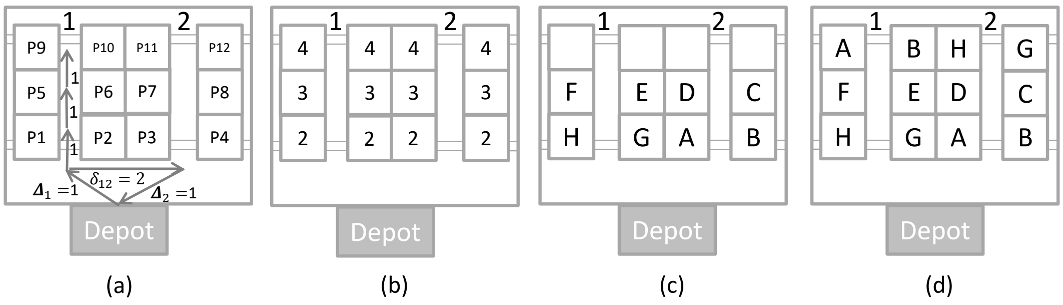


Figure 2.6.: Illustration of the priority rule. (a) Labeling each slot, (b) calculating the costs, (c) locating products in the first iteration, and (d) locating all products

Example (cont.): Consider the example in Figure 2.5. First, the heuristic calculates $C(i, j)$ for all potential locations in all the aisles, which is illustrated in Figures 2.6a and 2.6b. Numbers within each slot in Figure 2.6b show the calculated costs (lines 2-4). Consequently, the slots are sorted as $S^* := \langle P_1, P_2, P_3, \ldots, P_{12}]$ (line 5), and the SKUs are sorted based on their order frequency as $J^* := J' := \{H, A, B, D, F, G, C, E\}$. The heuristic starts placing the first element of set J', which is H, in the first element of S^*, which is P_1. For position P_2, for each SKU $j \in J'$, the heuristic calculates the number of times that SKU j and other SKUs which are already put in aisle 1 (so far only SKU H) are ordered together. In position P_2, G^*_A equals one because in only one order A and H are ordered together. For the other SKUs, this number is as follows: $G^*_B = 0$, $G^*_D = 0$, $G^*_F = 1$, $G^*_G = 2$, $G^*_C = 0$, and $G^*_E = 1$. Consequently, SKU G is selected and put in position P_2 because G^*_G is the maximum. After 7 more repetitions, Figure 2.6c is reached (lines 8-13). Note that when multiple G^*_j for some SKUs in a certain location are equal, a SKU is selected randomly. The final configuration is represented in Figure 2.6d. After solving the picker-routing problem for the given order set on this configuration, the total picking effort for retrieving the order set O is $W(g) = 58$.

2.5. Computational experiments

In this section, we present the numerical experiments to evaluate the proposed algorithms. In Section 2.5.1.1, we discuss how we generate instances for the picker-routing problem, and Section 2.5.1.2 concentrates on the computational results. Finally, we generate the instances for the product-location problem (Section 2.5.2.1) and perform computational tests to analyze the storage rules (Section 2.5.2.2).

2.5.1. Picker-routing algorithms

2.5.1.1. Benchmark generation and testing environment

In this section, we outline the generation of test data to evaluate the solution methods proposed in this paper. Because no adequate benchmark instances for the picker-routing problem in mobile rack warehouses exist in the literature, we generate a set of instances with different sizes and different features. Generally speaking, we assume the basic warehouse layout that is introduced in Section 2.3.1.

The set of instances consists of three main groups, which differ with regard to the number of aisles and SKUs. Within each group, instances are divided into two subgroups based on the density of SKUs in the warehouse. Specifically, we have two classes called *medium* and *high*. In class *medium*, 50% of the slots in all racks are occupied by SKUs of the order set. The other remaining locations either are not available, empty, or belong to other SKUs which are not part of the order set. Class *high* indicates that 75% of all locations include SKUs of the order set.

Rack and SRV movement times are uniformly distributed in the following intervals: according to (Schäfer, 2018), the reported velocity of a fully-loaded rack is 4 m/min. Allowing for different types of equipment and loads, this corresponds to a rack movement time within the interval of $[30, 90]$ seconds. We generate three classes of instances which differ with regard to the ratio of the time of rack movement and the travel speed. To this end, the rack movement time for each subgroup is in $\{30, 60, 90\}$ seconds. The comparatively fast-moving SRV can travel from one aisle to its neighbor with the average speed of $1\frac{m}{s}$. For every subgroup, we create 20 instances, which are formed by randomly distributing the SKUs in the racks following a uniform distribution. Note that this corresponds to a chaotic storage policy. The distance between two consecutive aisles is a random number in $\{0.5, 1, 1.5, 2\}$ meters. In the process of the generation of the test data, to select the location of the depot, first, an aisle is selected randomly and then it is assumed that the depot is immediately underneath it. Therefore, the distance between that aisle and the depot is zero. The distance from the other aisles to the depot can be calculated accordingly.

Thus, we have 3 (aisle counts) $\times$ 2 (densities) $\times$ 3 (rack movement times) $\times$ 20 (generated instances per group) = 360 benchmark instances in total. Table 2.1 summarizes the parameter ranges for the different instance sets. The latter are named in a specific way such that the first number shows the number of aisles, the second number states the number of SKUs in the order set, and the third part is the density of the problem.

To make the instance design more tangible, consider instance group 30-30-high as an example. In this group, there are 20 different instances. Figure 2.7 depicts the graphical

Instance set	no. aisles	no. different SKUs	no. locations in each aisle	time rack movement	density (%)
10-30-medium	10	5	10	30	50
10-30-high	10	5	10	30	75
10-60-medium	10	5	10	60	50
10-60-high	10	5	10	60	75
10-90-medium	10	5	10	90	50
10-90-high	10	5	10	90	75
30-30-medium	30	20	20	30	50
30-30-high	30	20	20	30	75
30-60-medium	30	20	20	60	50
30-60-high	30	20	20	60	75
30-90-medium	30	20	20	90	50
30-90-high	30	20	20	90	75
100-30-medium	100	70	30	30	50
100-30-high	100	70	30	30	75
100-60-medium	100	70	30	60	50
100-60-high	100	70	30	60	75
100-90-medium	100	70	30	90	50
100-90-high	100	70	30	90	75

Table 2.1.: Characteristics of generated benchmark instances.

layout of one of the instances in this group. There are 30 aisles with 20 locations in each aisle, and the order set to retrieve is $O = \{1, \ldots, 20\}$. The time it takes for an aisle to be opened is 30 seconds, and the driving time between two consecutive aisles is 1 second. In this instance group, almost 75% of all locations includes SKUs in set order O. Even so, rack movement times can be expected to dominate SRV travel times: even going to the farthest location inside an aisle and back takes at most 20 seconds, whereas opening the aisle in the first place takes 30 seconds.

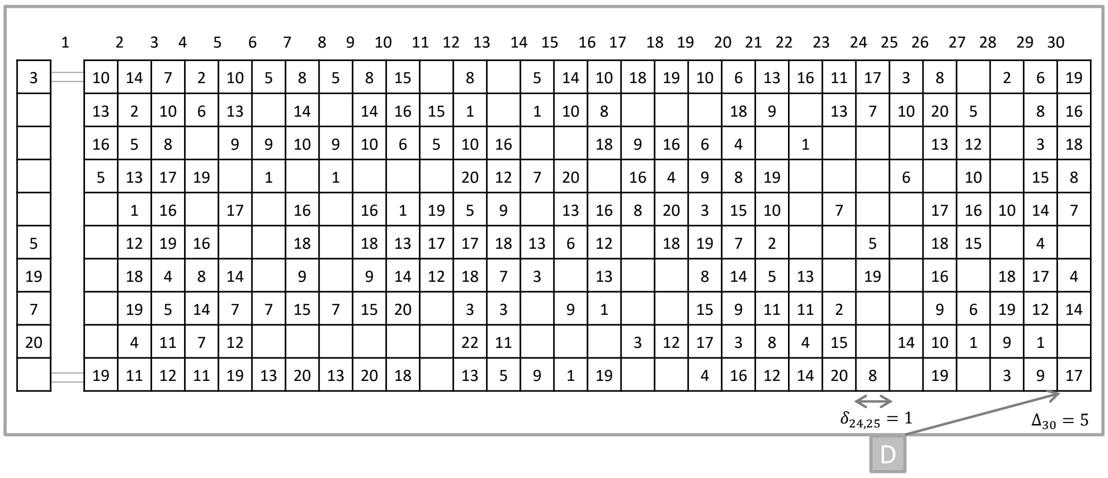


Figure 2.7.: The illustrative example of an instance in instance group 30-30-high.

All solution methods are coded in Java. The commercial solver CPLEX 12.6 is used to solve the mathematical models of the set-covering heuristic. All tests are performed on a standard desktop computer with an Intel Core 2 Quad Processor at 2.83 GHz, using 16 GB of RAM, and running Windows 7 Professional.

2.5.1.2. Performance of picker-routing algorithms

The results obtained for our solution methods on the small, medium-sized and large instances described above are presented in Table 2.2. The table reports the name of the instance set and the best known solution (BKS) as the average of the best objective function value found for the individual instances in the set by any of the solution methods. In case all instances in a set could be solved to optimality, we indicate this using an asterisk. The solution methods are abbreviated as follows: *SCH* stands for the set-covering heuristic, *BBS* for the bounded BS heuristic, and *DP* for our DP approach given a 6-hour time limit to solve each instance. In case of *BBS*, the number after the dash denotes the beam width. For each method, we report the average of the relative gaps of the objective value found by the respective solution method to the best-known solution in column "Gap". "CPU" reports the rounded up, average CPU time in seconds. In case DP cannot solve all instances of a set within the given time limit, no gap or run-time values are reported for DP. This is the case for the medium-sized and the large instances. Note that the CPU time for BBS includes the time it takes to calculate the initial upper bound via SCH.

BBS-1000 performs remarkably well in comparison with other solution methods with respect to solution quality. It finds the best solution for all but four instance sets, and its average gap to the BKS is 0.00% for the small and medium instances, and 0.52% for the large ones. However, especially for the large instances, run-times are quite long, exceeding three hours on average. *SCH* is able to obtain solutions of decent quality much quicker, taking less than four minutes of CPU time on average even for the large instances. It finds the best solution for only 5 out of 45 instances, and the average gaps are 2.16%, 2.85%, and 1.07% for the small, medium, and large instances, respectively. Finally, the performance of *BBS-100* and *BBS-10* are not convincing: on the small instances, no run-time benefits can be obtained, on the medium-sized instances, it is roughly faster by a factor of three compared to *BBS-1000* but with a significantly reduced solution quality, and on the large instances, it is dominated by *SCH*, the latter achieving better solution quality within shorter run-times.

Since the total pick times consist of travel time, when the picker moves from aisle to aisle or inside the aisles, and waiting time, when the picker has to wait for an aisle to open, we are interested in analyzing the proportion of waiting and travel times to the

		SCH		BBS-10		BBS-100		BBS-1000		DP	
Instance set	BKS	Gap (%)	CPU (s)	Gap (%)	CPU (s)	Gap (%)	CPU (s)	Gap (%)	CPU (s)		
10-30-medium	53.55*	4.38	5.4	0.19	5.4	0.00	5.3	0.00	5.3	0.00	0.0
10-60-medium	96.35*	2.77	5.8	0.26	5.8	0.00	5.7	0.00	5.6	0.00	0.0
10-90-medium	121.65*	1.42	5.0	0.00	5.0	0.00	4.9	0.00	4.9	0.00	0.0
10-30-high	39.30*	2.36	5.4	0.00	5.4	0.00	5.2	0.00	5.2	0.00	0.0
10-60-high	68.70*	1.36	4.9	0.15	4.9	0.00	4.7	0.00	4.7	0.00	0.0
10-90-high	99.10*	0.65	5.3	0.00	5.3	0.00	5.0	0.00	5.0	0.00	0.0
Avg. Small		2.16	5.3	0.10	5.3	0.00	5.1	0.00	5.1	0.00	0.0
30-30-medium	161.80	3.78	13.1	5.25	15.5	3.80	20.4	0.00	136.8	–	–
30-60-medium	259.70	3.49	13.2	4.56	15.2	2.07	15.5	0.00	129.0	–	–
30-90-medium	353.20	1.94	13.6	1.94	16.1	2.05	15.6	0.00	127.7	–	–
30-30-high	110.45	4.12	24.2	3.49	25.2	2.08	29.9	0.00	219.0	–	–
30-60-high	168.20	2.27	22.4	2.20	24.6	2.41	27.0	0.00	204.3	–	–
30-90-high	236.05	1.48	22.2	1.46	24.0	1.19	26.3	0.00	211.0	–	–
Avg. Med.		2.85	18.1	3.15	20.1	2.27	22.5	0.00	171.3	–	–
100-30-medium	615.40	0.00	150.1	0.89	153.6	0.67	243.1	0.20	7729.0	–	–
100-60-medium	872.10	1.71	148.5	3.69	155.7	3.14	243.8	0.00	8289.1	–	–
100-90-medium	1182.90	0.00	144.1	3.78	150.7	2.07	236.5	0.75	8253.7	–	–
100-30-high	441.10	0.00	269.1	1.47	306.4	2.80	434.3	1.70	14758.8	–	–
100-60-high	618.70	0.00	292.9	2.31	335.6	4.78	463.0	0.44	15193.1	–	–
100-90-high	781.90	4.73	280.5	6.86	319.4	6.09	447.4	0.00	14851.5	–	–
Avg. Large		1.07	214.2	3.17	236.9	3.26	344.7	0.52	11512.5	–	–
Total Avg.		2.03	79.2	2.14	87.4	1.84	124.0	0.17	3896.3	–	–

Table 2.2.: Comparison of solution methods on small, medium-sized, and large instances.

total order pick times for the best found solutions in each instance group. Table 2.3 reports the percentage shares of travel times and waiting times of the total pick times. The waiting time dominates the travel time in 16 out of 18 instance groups. On average, 71% of the total pick time is spent waiting for aisles to open.

Instance set	BKS	Waiting time Share %	Travel time Share %
10-30-medium	53.55*	73	27
10-60-medium	96.35*	84	16
10-90-medium	121.65*	89	11
10-30-high	39.30*	76	24
10-60-high	68.70*	87	13
10-90-high	99.10*	91	09
Avg. Small		83	17
30-30-medium	161.80	58	42
30-60-medium	259.70	72	28
30-90-medium	353.20	80	20
30-30-high	110.45	54	46
30-60-high	168.20	71	29
30-90-high	236.05	78	22
Avg. Med.		69	31
100-30-medium	615.40	47	53
100-60-medium	872.10	64	36
100-90-medium	1182.90	72	28
100-30-high	441.10	44	56
100-60-high	618.70	59	41
100-90-high	781.90	73	27
Avg. Large		60	40
Total Avg.		71	29

Table 2.3.: The share of waiting and travel time of the total pick time.

In Table 2.4, we investigate the effect of adding weights to each aisle (set) to improve the performance of *SCH* as proposed in Section 2.3.4. More precisely, we compare the performance of *SCH* with that of a variant of *SCH* using uniform weights (i.e., every set has the same weight of 1), denoted as *SCH-uniform*. The results show that adding weights dramatically improves the solution quality. On the other hand, the CPU times for *SCH-uniform* are much shorter because the set-covering problem only needs to be solved once. However, this speed-up cannot compensate for the significantly reduced solution quality.

		SCH		SCH-uniform	
Instance set	BKS	Gap (%)	CPU (s)	Gap (%)	CPU (s)
10-30-medium	53.55*	4.38	5.4	25.63	0.0
10-60-medium	96.35*	2.77	5.8	16.65	0.0
10-90-medium	121.65*	1.42	5.0	11.72	0.0
10-30-high	39.30*	2.36	5.4	34.50	0.0
10-60-high	68.70*	1.36	4.9	28.03	0.0
10-90-high	99.10*	0.65	5.3	17.07	0.0
Avg. Small		2.16	5.3	22.27	0.0
30-30-medium	161.80	3.78	13.1	34.90	0.0
30-60-medium	259.70	3.49	13.2	23.77	0.0
30-90-medium	353.20	1.94	13.6	17.47	0.0
30-30-high	110.45	4.12	24.2	36.52	0.0
30-60-high	168.20	2.27	22.5	26.81	0.0
30-90-high	236.05	1.48	22.2	19.31	0.0
Avg. Med.		2.85	18.1	26.46	0.0
100-30-medium	615.40	0.00	150.1	27.81	0.2
100-60-medium	872.10	1.71	148.5	15.87	0.1
100-90-medium	1182.90	0.00	144.1	7.84	0.2
100-30-high	441.10	0.00	269.1	26.90	0.2
100-60-high	618.70	0.00	292.9	19.11	0.2
100-90-high	781.90	4.73	280.5	14.90	0.2
Avg. Large		1.07	214.2	18.74	0.2
Total Avg.		2.03	79.2	22.49	0.1

Table 2.4.: Effect of adding weights to SCH

Finally, Table **??** studies the effect of using upper bounds obtained via *SCH* to prune states in BS as explained in Section 2.3.2. We report aggregated values over the groups of instance sets for different versions of BS using bounds (called *BBS*) or not using bounds (called *BS*). As for the results, using bounds improves the solution quality in most cases, especially when low beam widths are used. On the downside, CPU times increase to some extent. However, the impact is not huge compared to the often significantly improved solution quality.

	BBS-100		BS-100		BBS-1000		BS-1000	
Instance set	Gap (%)	CPU (s)	Gap (%)	CPU (s)	Gap (%)	CPU (s)	Gap (%)	CPU (s)
Avg. Small	0.00	5.1	0.01	0.0	0.00	5.1	0.00	0.0
Avg. Med.	2.27	22.5	14.92	3.3	0.00	171.3	2.30	164.7
Avg. Large	3.26	344.7	15.40	133.8	0.52	11512.5	3.45	11529.7

Table 2.5.: Effect of using bounds in beam search

2.5.2. Product-location problem

2.5.2.1. Testing environment

In this part of our computational study, we investigate the product-location problem introduced in Section 2.4. Specifically, we seek to answer the question which of the discussed product-location policies is most suitable for facilitating the subsequent picker-routing problem. Clearly, both problems are interdependent: depending on how SKUs to be picked are distributed in the warehouse, even optimal picker routes can be longer or shorter.

For this series of tests, we generate a new data set with 16 instances. The layout of the warehouse and the distances between aisles and depot are exactly as described in Section 2.5.1.1. We investigate four different warehouse settings and generate four instances for each setting. A setting is defined by a tuple $(m, k, |J|, |O|, M)$ with m the number of aisles in the warehouse, k the number of locations in each aisle, $|J|$ the maximum number of different SKUs to store in the warehouse, $|O|$ the number of orders in the main order set, and M the maximum number of SKUs in each single order in the order set. The settings $(5, 5, 10, 5, 5)$, $(10, 10, 20, 10, 7)$, $(20, 15, 40, 15, 15)$, and $(20, 15, 40, 15, 15)$ are studied. In the order set O, we have $|O|$ different individual orders, and each order can consist of a maximum of M different SKUs, which are drawn randomly from an exponential distribution with a mean of 10. The exponential distribution is chosen because in many real-world warehouses a small set of SKUs are substantially more popular than others, e.g., (Bartholdi and Hackman, 2008). Note that some SKUs might not be in the order set at all, e.g., if an instance is supposed to contain 80 different SKUs, but only 78 are actually part of any order (due to the exponential distribution). These SKUs are still considered part of the problem, however, and will be assigned to storage locations.

For each instance, to assign the SKUs to storage locations, we employ two policies: chaotic storage, where SKUs are assigned randomly to storage locations, as well as the priority rule we proposed in Section 2.4. To evaluate the quality of a given location assignment, we subsequently solve the picker-routing problem on one of two order sets. The first one is called the *historical order set*, which means that the exact same orders that were used to determine product locations are also actually realized. Recall that this setting comes close to reality in intermediate warehouse where recurrent part and material demands for cyclically repeated production lots need to be satisfied. In other warehouse settings, however, it is not necessarily realistic to assume that the *exact* composition of future orders is already known when the location problem is solved. To emulate such a setting we also use an *anticipated order set* to solve the picker-routing problem. This order set is randomly generated as described in the previous paragraph, using the same probability distribution (exponential distribution with a mean of 10) but using a new random seed. In other words, this order set obeys the anticipated distribution, but it is not exactly equal to the original order set used to solve the product-location problem.

Finally, we also propose an integrated MIP model in the appendix. Using a default solver (CPLEX 12.6), the model solves the joint product location and picker routing problem to optimality. In other words, this model operates under the assumption that the exact pick list is already known at the time the shelves are stocked, which is not realistic in most cases but yields the very best objective value under perfect information, which may serve as a bound to compare the other product location policies against.

Due to the randomness of the chaotic storage policy, we re-assign storage locations 100 times and only the best objective value out of these runs is reported. CPU times are cumulative for all runs, however. Moreover, there is a time limit of 3 hours for the policies (including both the location as well as routing phases). After exceeding the time limit, the best found solution is returned. To solve the routing problem, we use DP for the smaller warehouse settings (instances 1 through 8). For the larger instances, we employ SCH (instances 9 through 16). Note that both chaotic assignment as well as the priority rule heuristic take negligible CPU time for all our test problems, i.e., substantially less than one second in all cases. We therefore do not explicitly mention CPU times in the following tables. Solving the subsequent picker routing problems can take a significant amount of time. As the CPU time is independent of the product locations, however, we refer the reader to Section 2.5.1.2 for details.

2.5.2.2. Performance of location assignment policies

Table 2.6 illustrates the results obtained with the proposed storage policies, i.e., the priority rule (*P-R*) and chaotic storage (*Chaotic*). Column *BKS* reports the best known solution, columns *Gap* the percentage gap of the total pick time for all orders in the order set for both historical and anticipated (columns labeled *(antc)*) order sets. Finally, the average relative gaps of the integrated MIP model to the proposed priority rule and chaotic storage policy are reported in column *Int. MIP*. Since the integrated MIP optimizes product locations and picker routes jointly, whereas the other policies optimize both problems successively, the gaps cannot be positive – the integrated solution must be at least as good as the best successive solution. Note that the integrated MIP model can only solve the small instances for the *historical order set* within a 3-hour time limit.

Warehouse setting							P-R	P-R (antc)	Chaotic	Chaotic (antc)	Int. MIP
Instance	m	k	$\|J\|$	$\|O\|$	M	BKS	Gap (%)	Gap (%)	Gap (%)	Gap (%)	Gap (%)
1	5	5	10	5	5	301.00	1.00	1.99	0.00	1.00	-1.33
2						300.00	4.33	1.33	4.33	0.00	-1.92
3						301.00	2.99	0.00	4.32	1.00	-0.97
4						306.00	0.00	1.31	0.33	0.98	-1.63
Avg. Small							2.08	1.16	2.24	0.74	-1.46
5	10	10	20	10	7	872.00	0.00	3.44	16.51	9.86	–
6						856.00	0.00	6.54	7.01	14.72	–
7						842.00	1.43	1.43	9.26	0.00	–
8						858.00	0.00	1.86	7.93	4.43	–
Avg. Mid.							0.36	3.32	10.18	7.25	–
9	20	15	40	15	15	1437.00	0.00	33.33	21.99	42.38	–
10						1660.00	0.00	3.61	13.31	3.55	–
11						1500.00	0.00	13.40	17.67	18.33	–
12						1383.00	0.00	19.45	32.10	27.40	–
Avg. Large							0.00	17.45	21.27	22.92	–
13	40	20	80	20	30	3219.00	0.00	16.43	30.41	28.18	–
14						3502.00	0.00	20.99	22.96	27.41	–
15						3682.00	0.00	15.15	26.18	21.94	
16						3171.00	0.00	24.06	26.58	35.26	–
Avg. Very large							0.00	19.16	26.53	28.20	–
Total Avg.							0.61	10.27	15.06	14.78	–

Table 2.6.: Comparing priority rule, chaotic storage policy and integrated storage and routing.

Comparing the priority rule and the chaotic storage policy, the priority rule-based product location policy fails to produce the shortest pick time only in one single instance. In all other cases, it is clearly superior to chaotic storage. This holds true regardless of whether the historical or the anticipated order set is used to evaluate product locations. Regarding this distinction, unsurprisingly, the priority rule works best if the historical data is also used for evaluation. In practice, this would, however, imply that the planner already knows the precise customer orders when the product-location problem is solved. Due to the different planning horizon of these two steps, this is unlikely to be the case in most branches of industry. It is therefore encouraging to see that the priority rule still delivers substantially better results than chaotic storage even if merely the probability distribution of future orders is known, i.e., the anticipated order set is used in the routing phase.

So far, we assumed that shared storage of SKUs is allowed, meaning that the same SKU may be put in multiple locations. However, for organizational reasons and to make routes less confusing for human pickers, it may be expedient for warehouse operators to assign fixed (or dedicated, see (Bartholdi and Hackman, 2017)) storage locations to SKUs, i.e., each SKU is assigned to exactly one location. To compare the shared and fixed storage schemes, we employ the priority rule and the chaotic storage policy using both schemes. Specifically, under a fixed storage policy, a SKU is removed from consideration once it has been assigned to a storage position by either the priority rule of the random number generator (in case of chaotic storage). Under a shared storage policy, the same SKU can be assigned multiple times as long as there is space left and each SKU is in at least one position. Table 2.7 compares the total pick time, calculated as before. The column headers are the same as in Table 2.6, but in the columns labeled*fixed* the results obtained using fixed assignment of each SKU to one unique location are listed; the other columns contain the results under the shared storage scheme, as in Table 2.6.

Regarding the results, expectedly, fixed assignment performs worse than shared. However, the size of the gap is astonishing: the average pick time is almost 3 times as large under a fixed assignment for the chaotic strategy. If the more sophisticated priority rule is used, the gap is clearly lower, at about 33% additional picking effort over the best known (shared) solution. The size of the gap is probably due to the slow movement speed of the racks: if SKUs are only stored in a single location, it becomes much more likely that a greater number of aisles must be opened, which is heavily penalized by long waiting times. Since our priority rule tends to cluster SKUs that are often ordered together in the same aisles, the penalty is substantially lower. This strongly suggests that chaotic storage paired with fixed storage locations is not a good fit at all for mobile rack warehouses.

Warehouse setting							P-R	P-R (fixed)	Chaotic	Chaotic (fixed)
Instance	m	k	$\|J\|$	$\|O\|$	M	BKS	Gap (%)	Gap (%)	Gap (%)	Gap (%)
1	5	5	10	5	5	301.00	1.00	36.54	0.00	2.66
2						300.00	4.33	41.33	4.33	47.00
3						301.00	2.99	39.20	4.32	48.50
4						306.00	0.00	17.97	0.33	27.78
Avg. Small							2.08	33.76	2.24	31.49
5	10	10	20	10	7	872.00	0.00	0.00	16.51	239.68
6						856.00	0.00	0.00	7.01	154.21
7						842.00	1.43	1.43	9.26	183.73
8						858.00	0.00	0.00	7.93	149.07
Avg. Med.							0.36	0.36	10.18	183.73
9	20	15	40	15	15	1437.00	0.00	68.06	21.99	258.32
10						1660.00	0.00	56.99	13.31	234.28
11						1500.00	0.00	64.00	17.67	228.73
12						1383.00	0.00	55.60	32.10	260.09
Avg. Large.							0.00	61.16	21.27	245.35
13	40	20	80	20	30	3219.00	0.00	33.15	30.41	317.46
14						3502.00	0.00	35.55	22.96	302.14
15						3682.00	0.00	45.82	26.18	286.69
16						3171.00	0.00	25.70	26.58	285.59
Avg. Very large							0.00	35.05	26.53	297.97
Total Avg.							0.61	32.58	15.06	189.12

Table 2.7.: Comparing shared and fixed policies for the priority rule and the chaotic strategy

2.6. Conclusion

In this paper, we investigated the picker routing as well as the interrelated product-location problem as it arises in warehouses with moveable racks, designed for space efficiency. The routing problem is made more complicated in such warehouses than in many classic warehouses, especially in case of scattered storage, due to the added difficulty of having to move racks out of the way to open aisles. To solve this problem, we proposed a dynamic programming and a beam search scheme, as well as a set-covering-based heuristic. For the product-location problem, which concerns itself with placing SKUs in the storage area such that subsequent pick times are as short as possible, we investigated two different policies, and compared the performance of shared as well as fixed storage space allocation.

Our computational tests reveal that the proposed heuristics for the routing problem perform quite well, although there is a distinct tradeoff regarding solution time and quality. The dynamic programming scheme cannot reasonably solve larger instances, whereas both beam search as well as the set-covering heuristic have proven quite adept. Beam search, however, while delivering good results, may take a long time to reach that solution in many instances. The set-covering heuristic is very fast but exhibits non-negligible gaps regarding the best solutions known.

Regarding managerial insights, our tests reveal the following take-home messages:

- Chaotic storage, i.e., assigning random locations to SKUs, delivers suboptimal results, and leads to unnecessarily long pick times. Even with the comparatively simple priority rule presented in this paper, significantly better results are possible, i.e., up to 15%.
- Fixed storage, i.e., assigning a single storage location to each SKU, is not advisable in a mobile rack warehouse. It leads to a much greater number of rack movements, which should be avoided due to the low movement speed. Savings upwards of 30% are possible.
- The priority rule for the location problem introduced in this paper works quite well even if the exact composition of future orders is not known. It suffices to have an idea about the distribution of future SKU demands. Significant savings compared to chaotic storage are possible.

Future research should focus on designing more sophisticated solution methods for the product-location problem. Seeing that mobile racks are often used in refrigerated warehouses, it may also be relevant to explicitly consider perishable products requiring just-in-time operations. Furthermore, it may make sense to schedule replenishment activities concurrently with retrievals, seeing that the rack movements are so slow; this can also be incorporated in future models. Since our computational experiments show that a substantial share of the total pick time consists of waiting for the racks to move, investigating systems with more than one open aisle may also be a promising avenue of future research. Finally, in our paper we assume that each unit load contains enough items to satisfy each customer order. In a business-to-business (B2B) setting, however, extraordinarily large orders may occur, so that multiple storage positions of the same SKU have to be visited to gather enough items. In this case, our dynamic programming scheme no longer works, because the remaining items per storage position have to be recorded in the states. Designing algorithms, e.g., integer programming approaches, for this extended problem setting is also a challenging task for future research.

3. Outpatient appointment scheduling with priority rules

Co-author:

Raik Stolletz
Chair of Production Management, Business School, University of Mannheim, Germany

Working paper.

Abstract:

Many service providers that use appointment systems, such as radiology departments in hospitals, experience walk-ins in addition to scheduled appointments. Among the former, emergency patients must be served as soon as possible, and other types of patients are mostly given lower priority to be served. The system usually uses a certain priority rule to control the access of the patients waiting for being served when no emergency patients are waiting. Two groups of priority rules, namely, static and dynamic priority rules, are used in appointment systems. In static priority rules, the priority of patients does not change depending on the state of the system; conversely, in dynamic priority rules, the priority of a patient depends on the state of the system. Using a specific priority rule may lead to a disruption of the initially scheduled appointments and affect the system's performance. In this paper, we determine when outpatients should be scheduled if there are arrivals of emergency patients and inpatients and a certain priority rule is used by the system.

To address this problem, a simulation optimization approach based on tabu search with a neighborhood reduction technique is proposed. It uses low-fidelity simulation to rank and pick neighboring solutions. The findings demonstrate the effectiveness and efficiency of the proposed algorithm in solving the outpatient scheduling problem with walk-ins and no-shows. Furthermore, the structure of the optimal schedule of the outpatients when the system has different priority rules is studied. The results suggest that to schedule outpatients, decision-makers should consider which priority rules the appointment system applies.

keywords: Outpatient appointment scheduling & Priority rules & Simulation optimization & Multifidelity simulation

3.1. Introduction

Outpatient service providers face a rapidly increasing demand to satisfy within a limited time frame or with a limited capacity of facilities (Froehle and Magazine (2013)). Therefore, improving access to health care services is an important goal for these providers. Timely access influences the quality of the service encounters of patients and health care providers (Cayirli and Veral (2003)). It also improves health care providers' contentment and earnings and patients' satisfaction (Gupta and Denton (2008)). Scheduling the appointments of outpatients trades off patients' long waiting times and technicians' overtime (Ahmadi-Javid et al. (2017)). Outpatient appointment scheduling consists of determining the arrival times of scheduled outpatients on a particular day, wherein some patients can share the same arrival time. This task may be further complicated by outpatient no-shows, the stochastic nature of arrivals of walk-ins, and the use of a particular priority rule in the system.

Health care systems consider different classifications of patients who are served according to priority rules. Jaiswal (1968) defined the priority as the "measure of importance" that distinguishes the different groups of arrivals. For example, in radiology departments in hospitals, three patient classes–emergency patients, outpatients, and inpatients–receive services. If there is an emergency demand request, it receives the highest priority. Since inpatients are available for a greater period, they are given a lower priority than outpatients (Green et al. (2006)).

Jaiswal (1968) also explained that each priority rule always specifies which unit is served once the server is free and whether to continue or discontinue the service of the unit being served. Priority rules provide real-time control of access to the facility for potentially competing patient classes (Green et al. (2006)). For example, in radiology departments in hospitals, at each point in time, there may be waiting patients from more than one class. In the absence of emergency patients, the predetermined priority rule decides whether an outpatient or an inpatient is served next. When an outpatient appointment system has a specific priority rule, the selection of an appointment schedule affects the likelihood and timing of the delays of different patient classes, which impacts the overall performance of the service facility. Therefore, it is important and challenging to make scheduling decisions while considering a system's priority rules.

When a system is remarkably underloaded, the optimal schedule for outpatients does not depend on the priority rule. This exists because the probability of having different types of patients waiting at a certain point in time is low regardless of the system's priority rule. The question that arises is when the load of the system is high, does the optimal schedule differ with different priority rules?

To address how the priority rules affect the optimal appointment schedule of outpatients, a simulation optimization approach with tabu search enhanced with a multifidelity simulation-based neighborhood reduction method is proposed. Simulation provides a highly accurate estimate of an expected performance measure in a system and has high flexibility in modeling and analyzing complex systems. Tabu search guides local search heuristics to escape from local optima and provides optimal or near-optimal solutions by applying an adaptive form of memory (Glover (1986)). Tabu search is a strong metaheuristic that is used in many studies in several applications, as shown in Klassen and Yoogalingam (2009) and Schneider et al. (2017).

The proposed tabu search is enhanced by a neighborhood reduction technique. Neighborhood size reduction techniques have been widely studied in the literature. For example, Schneider et al. (2017), Toth and Vigo (2003), and Potvin et al. (1996) consider the limited list of promising neighbors in their proposed solution methods. In another example, Xiao et al. (2011) consider a reduced distance-based neighborhood structure for the proposed variable neighborhood search procedure. In our study, a neighborhood reduction method based on the study of Xu et al. (2014) for multifidelity approximation methods is used. There are several studies related to simulation optimization approaches that apply frameworks to use multifidelity approximation methods, such as Lin et al. (2019, 2020).

The reminder of this paper is organized as follows: Section 3.2 explains priority rules and provides an overview of the literature on appointment scheduling for these rules. Section 3.3 describes the studied problem. The proposed simulation optimization with tabu search is explained in Section 3.4. The numerical studies are described in Section 3.5. Finally, the last section concludes this paper and presents possible future research directions.

3.2. Literature review

Appointment scheduling has received considerable attention from academics, as shown in the literature reviews Cayirli and Veral (2003), Gupta and Denton (2008), and Ahmadi-Javid et al. (2017). This section reviews the papers on appointment systems with heterogeneous patient groups, where access to service is controlled by priority rules. Priority rules have been studied extensively due to their numerous applications in many service facilities, such as telecommunications (Stidham Jr (2002)) and health care (Lakshmi and Iyer (2013)).

Subsection 3.2.1 reviews the appointment systems with static priority rules, and Subsection 3.2.2 describes the appointment scheduling studies considering dynamic priority rules.

3.2.1. Appointment systems with a static priority rule

In appointment systems having a static priority rule, the priority of patient groups does not change depending on the state of the system. Different priorities are given to patient groups based on the urgency of their treatment. The server always picks the waiting patient with the highest priority to serve. When the priority rule is preemptive, the service of low priority patients is interrupted to serve an arriving higher priority patient. For example Luo et al. (2012) address this issue with two patient groups. The remaining papers in appointment systems that are reviewed here consider nonpreemptive service, where the server always finishes the current treatment.

Many papers in the appointment scheduling literature studying two patient groups have a static priority rule, such as Koeleman and Koole (2012) and Begen et al. (2012). In this subsection, we focus on papers that consider at least three classes of patients. Rising et al. (1973) study a university health service clinic. They model three patient classes, including emergency, scheduled, and walk-in patients, where emergency patients have the highest priority and walk-ins have the lowest priority. They use simulation and historical data from the university clinic to evaluate different scheduled appointments. In Peng et al. (2014), the appointment system in a clinic with prebooked, open-access, and walk-in patients was analyzed. Prebooked patients already have an appointment and have the highest priority. The second priority belongs to open-access patients who ask for same-day appointments. The model optimizes the schedule of prebooked and open-access patients while minimizing the patients' expected waiting times and the health care provider's idle time and overtime. They propose simulation optimization using a genetic algorithm solution approach. The study of Bhattacharjee and Ray (2016) is a case study of the radiology department of a hospital in India. The authors model four different patient classes where the highest priority is emergency patients and the lowest priority is walk-ins. The priorities of the other patient groups are the same, and the first-come-first-served discipline is applied to control their access to service. The simulation is used to examine different appointment rules with data drawn from the studied hospital.

3.2.2. Appointment systems with a dynamic priority rule

Dynamic priority rules depend on the state of the system. There are two different state-dependent priority rules that are considered in the literature. We call the first group the substitutive priority rule, and the second group is called the threshold priority rule.

In the substitutive priority rule, if a scheduled patient does not show up for her appointment (at a certain point in time), the priority of the patient from a lower priority class, such as a nonurgent walk-in, which has the largest waiting time becomes higher than other scheduled patients who enter the system after that time. In other words, the position of the no-show in the queue is given to that nonurgent patient and she is substituted with the no-show patient.

Liu and Ziya (2014) discuss a queueing model ($M(t)/D/1$) to examine an appointment system with scheduled patients and walk-ins, where scheduled patients have higher priority. The model optimizes the total number of patients treated and the level of overbooking. The authors analytically solve the problem and gain insights into the structure of the optimal solution. Kortbeek et al. (2014) propose two models to evaluate the schedule of patients to design an appointment system for a clinic. The proposed models optimize the appointment day and time of each scheduled patient while simultaneously maximizing number of nonurgent patients served. The optimal schedules of their models have to meet a predefined target for the access time service level. Nonurgent patients are served in free appointment slots and in reopened appointment slots of no-show patients. This study employs a full enumeration procedure and a heuristic solution approach to numerically solve the problems.

In the threshold priority rule, when a low priority patient waits for a certain amount of time or the queue length reaches a threshold, she obtains a higher priority compared to that of some or all other patient groups.

Borgman et al. (2018) study an appointment system in a radiology department in a Dutch hospital with scheduled patients and walk-ins with a threshold priority rule. If the waiting time of each walk-in patient reaches its threshold, their priority becomes higher than that of scheduled patients. Their model minimizes the expected waiting time, idle time, and overtime costs. They apply a simulation optimization approach using a local search to explore the solution space. Green et al. (2006) examine the appointment scheduling problem in a radiology department with emergency patients, scheduled outpatients, and inpatients. The authors model the problem as a Markov decision process and propose a linear capacity allocation heuristic. Their model schedules outpatients for each period and dynamically selects the next patient to be served at the beginning of every period. The model maximizes the total expected profits of serving inpatients and outpatients. Gocgun et al. (2011) extend the study of Green et al. (2006) and consider more patient types in a radiology department with two scanners. They also apply a Markov decision process to obtain the optimal policy of selecting the next patient to be served at the beginning of each period. In Kolisch and Sickinger (2008), an appointment system in a radiology department with two CT scanners in which three patient types (emergency, scheduled outpatients, and inpatients) were served was examined. The problem is modeled as a Markov decision process. The proposed model decides which patient type to choose to serve in each period. They compare the obtained optimal solution with given schedules as appointment rules over data from their field studies.

To summarize, all the studies considering an appointment system with distinguished patient groups, whether optimization or simulation studies, consider only one priority rule. No paper studies the importance of the priority rules used in the system while optimizing a decision variable or evaluating performance measures. Moreover, appointment systems with at least three patient groups have received limited coverage in the research literature. We fill this gap by analyzing the structure of the optimal schedule of an appointment system with three patient groups and comparing the schedules for priority rules.

3.3. Problem description

In this section, we describe the studied system and the optimization problem. We analyze static (P_1), substitutive (P_2), and threshold ($P_3(\omega)$) priority rules with ω as the predefined threshold for waiting time.

The appointment system is represented as a single server queue with three patient groups, as shown in Figure 3.1. Let us assume that the treatment room is operational in periods $t \in \{1, \cdots, T\}$ with a length of d each. The service time is generally distributed with a service rate of μ and a coefficient of variation of cv_s. In radiology departments, service is usually nonpreemptive. If a patient is not yet served during the planning horizon, she can be served using overtime.

We distinguish between three independent heterogeneous patient groups. Let i be the index of each patient type. The first queue, which represents the arrival of emergency patients ($i = 1$), has the highest priority regardless of the system's priority rule. It has a given distribution for the interarrival time with an arrival rate of λ_1 and a related coefficient of variation of cv_{a_1}. The second arrival process ($i = 2$) represents the arrival of the scheduled outpatients with a rate of $\lambda_2(t) = X(t)$ at the beginning of each period. The aim of the studied optimization problem is to schedule N outpatients.

The actual number of outpatients is stochastic since each scheduled patient has a probability of ρ of not showing up to her appointment or canceling late. In the reminder of the paper, we use the term scheduled patients and outpatients interchangeably. The last queue represents the arrival of inpatients ($i = 3$) with an arrival rate of λ_3 and a coefficient of variation of cv_{a_3}.

In our study, when the static priority rule is applied for the system, the priority of scheduled outpatients is higher than that of inpatients. Moreover, while using the substitutive priority rule in the system, inpatients have lower priority than outpatients, and they are replaced by no-show outpatients. Finally, in the threshold priority rule, when the waiting time of an inpatient reaches her threshold, her priority increases compared to that of outpatients, and she will be served as soon as the server is empty and no emergency patients are waiting.

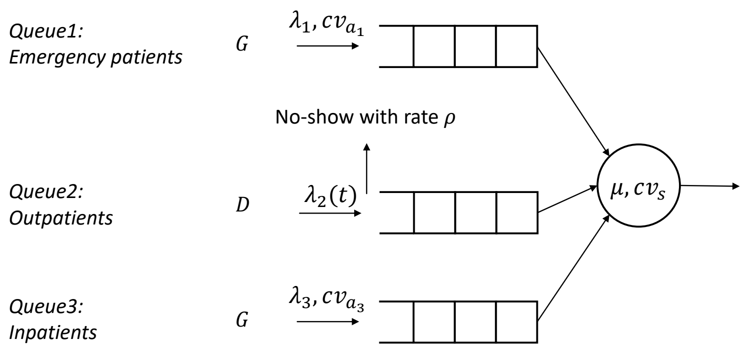


Figure 3.1.: Queueing system description

$X(t)$ is the main decision variable of the optimization problem. The objective function consists of the expected overtime costs $E[\mathbf{O}(X(t))]$ and the aggregate expected waiting costs related to each type of patient $E[\mathbf{W}_i(X(t))]$. Let c^{over} per hour be the overtime cost coefficient. The patients' waiting cost coefficients for each type i are c_i^w. The optimization problem for a given priority rule is stated as follows:

$$\text{Min } Z = \underbrace{c^{over} E[\mathbf{O}(X(t))]}_{\text{overtime cost}} + \sum_{i \in \{1,2,3\}} \left(\underbrace{c_i^w E[\mathbf{W}_i(X(t))]}_{\text{waiting costs}} \right) \tag{3.1}$$

s.t:

$$\sum_{t=1}^{T} X(t) = N \tag{3.2}$$

$$X(t) \in Z^+ \qquad \forall t \tag{3.3}$$

The objective function (3.1) minimizes the expected costs. The first part is the expected overtime costs of the server. The reminder is the expected costs caused by waiting for different patient types. Constraint (3.2) assures that exactly N outpatients are scheduled. The last constraint defines the domain of the variables.

The used notation is summarized in Table 3.1.

Table 3.1.: Table of notations for the problem description

Notation	Type	Description
t	Index	Index of periods
i	Index	Type of patients
T	Parameter	Number of periods
d	Parameter	Intervals length
N	Parameter	Number of scheduled patients to be served
ω	Parameter	Threshold for changing the priority in threshold priority
μ	Parameter	Service rate
cv^2	Parameter	Coefficient of variation of service time
λ_1	Parameter	Arrival rate of emergency patients
$\lambda_2(t)$	Parameter	Arrival rate of scheduled patients at period t
λ_3	Parameter	Arrival rate of inpatients
cv_{a_1}	Parameter	Coefficient of variation of emergency patients' inter-arrival time
cv_{a_3}	Parameter	Coefficient of variation of inpatients' inter-arrival time
ρ	Parameter	No-show probability of each outpatient
c_i^w	Parameter	Waiting time cost coefficient for patient type i
c^{over}	Parameter	Overtime cost coefficient
$X(t)$	Decision	Number of scheduled patients at the beginning of period t
Z	Decision	Objective value
$E[\mathbf{W}_i]$	Decision	Aggregate expected waiting time of patients of type i
$E[\mathbf{O}]$	Decision	Expected server's overtime

3.4. Simulation optimization with tabu search

Simulation optimization has been used in a wide range of optimization studies, as shown in the review of Amaran et al. (2016). We propose simulation optimization with tabu search to solve the appointment scheduling problem presented in Section 3.3. As shown previously, the studied system has high complexity, and thus simulation is a proper choice to accurately evaluate an incumbent solution's performance measures. However, simulation is time-consuming. Therefore, applying a local search that only can evaluate one solution per iteration will slow down the optimization algorithm dramatically. Therefore, a tabu search is used which guides the search for an optimum. To more speed up the proposed algorithm, we implement multifidelity simulations (with a high and low number of replications) within a neighborhood reduction method.

A solution $X(t) = (X(1), X(2), \cdots, X(T))$ represents the number of outpatients scheduled at the start of each period. To construct an initial solution, each outpatient is assigned to a random slot t. Then, the expected waiting time of each patient group $E[\mathbf{W}_i(X(t))]$ and the expected overtime $E[\mathbf{O}(X(t))]$ are evaluated by high-fidelity simulation (simulations with a high number of replications).

In every iteration, the neighborhood $\mathcal{N}(S)$ is built by applying a set of neighborhood operators or moves to the incumbent solution S. We discuss the structure of the neighborhood for the search in Section 3.4.1. Section 3.4.2 introduces a neighborhood reduction scheme in which low-fidelity simulation (simulations with a low number of replications) is applied and a subset $\mathcal{N}_r(S) \subseteq \mathcal{N}(S)$ of neighboring solutions is selected. $\mathcal{N}_r(S)$ is evaluated by high-fidelity simulation, and the best nontabu solution is chosen.

To diversify the search, two mechanisms called the “probabilistic phase” and “randomly chosen phase” are applied. Each mechanism is activated when a certain criterion is fulfilled. The diversification methods and the criteria in which they are activated are explained in Section 3.4.3. After selecting the new incumbent solution, the tabu list and the best-so-far solution (BSF) are updated; and the decision of whether to start, continue, or terminate a diversification mechanism is made. Whenever the algorithm finds a new BSF solution, an aspiration criterion permits clearing the tabu list. Finally, when the BSF is not improved during the last η_{max} iterations, the search terminates.

3.4.1. Neighborhood structure and tabu list

The improvement part of the algorithm uses a composite neighborhood $\mathcal{N}(S)$ obtained by combining left-shift and right-shift operators. The left-shift operator is derived from the study of Kaandorp and Koole (2007), in which an outpatient moves from time slot t to time slot $t-1$. If the patient is at the first time slot of the planning horizon, i.e., $t = 1$, she moves to the last time slot ($t = T$). For the right-shift operator, an outpatient is moved from time slot t to time slot $t+1$. Both moves are executed only if there is at least one patient scheduled at time slot t. Applying these two moves results in at most $2 \cdot T$ different neighboring solutions to S.

Regarding tabu list management, we define the solution that is once selected as an incumbent solution as a tabu for a tabu tenure of ϑ iterations. Therefore, the moves that result in the solution in the tabu list are forbidden from being executed. The tabu tenure is set to a uniform random value in $[\vartheta_{min}, \vartheta_{max}]$ for every solution inserted into the tabu list. The tabu list is cleared as an aspiration criterion every time the algorithm finds a new BSF solution.

3.4.2. Neighborhood reduction method

Given a list of all neighboring solutions $\mathcal{N}(S)$, a restricted neighborhood $\mathcal{N}_r(S)$ with only $\theta\%$ of all neighboring solutions is selected.

To define $\mathcal{N}_r(S)$, we apply a framework similar to that of Xu et al. (2014) for a multifidelity simulation optimization algorithm. First, we evaluate all the neighboring solutions in $\mathcal{N}(S)$ by low-fidelity simulation and rank them according to their objective values. They are partitioned into a given number of groups (K) based on their relative ranks. Because of the potentially significant errors in low-fidelity models and the possibility that the best nontabu neighboring solution exists in a nontop group, a certain number of best solutions is selected from each group.

The iterative procedure to pick solutions from each group is based on the mean and variance of objective values within a group and based on the distance in the mean between two groups, see Appendix B.

3.4.3. Diversification mechanism

A crucial component in the design of an efficient solution method is the inclusion of a diversification mechanism that helps to explore the unvisited solution space. We use two main diversification methods, which are called the “probabilistic phase” and “randomly chosen phase” (Schneider et al., 2017).

For the probabilistic phase, a nontabu solution among reduced neighboring solutions is randomly chosen. The probabilistic phase is started as soon as there is no improvement for the BSF solution for the past η_{div} iterations. Once started, the probabilistic phase runs for at most η_{prob} iterations or it is stopped earlier if a new BSF solution is found.

In the randomly chosen phase, the next solution to visit is constructed randomly. The tabu list is restarted, and thus, it may help to escape situations in which the algorithm gets stuck in local optima. This diversification mechanism terminates after one iteration. Note that the search is very likely to take a different path than before because it is restricted with different tabu list entities.

In summary, Algorithm 2 shows the pseudocode of the proposed algorithm to solve the optimization problem for a given priority rule.

Algorithm 2 Pseudocode Overview of Tabu Search

```
1: procedure TABU SEARCH
2:     Initialize $tabuList$
3:     $S \leftarrow$ initial solution created by RANDOM PROCEDURE
4:     $S \leftarrow S^*$
5:     $probabilisticPhase \leftarrow$ false
6:     $randomlyChosenPhase \leftarrow$ false
7:     while termination criteria not satisfied do
8:         if $randomlyChosenPhase$ then
9:             $S \leftarrow$ solution created by RANDOM PROCEDURE
10:        else if $probabilisticPhase$ then
11:            $S \leftarrow$ select a random solution in $\mathcal{N}_r(S) \subset \mathcal{N}(S)/tabuList$
12:        else
13:            $S \leftarrow$ select best solution in $\mathcal{N}_r(S) \subset \mathcal{N}(S)/tabuList$
14:        update $tabuList$, $S^*$, $probabilisticPhase$ , and $randomlyChosenPhase$
15:    Return $S^*$
```

3.5. Numerical study

In this section, we present numerical experiments. In Section 3.5.1, we first explain how the analytical instances are generated and how the algorithm parameters are selected. In Section 3.5.2, the performance of the algorithm compared with a full enumeration procedure is discussed. In Section 3.5.3, sensitivity analyses for arrival parameters are conducted to study the structure of the solution for the three priority rules.

3.5.1. Instance generation and parameter selection

We generate instances with one service provider. The planning horizon is divided into periods with length of $d = 30\ min$. In the following, we summarize the parameter choices in the base instance benchmark based on case studies reported in the literature.

We use a gamma distribution with a given service rate of $\mu \in \{1, 2\}$ per hour and $cv_s^2 = 0.4$ based on Yang et al. (1998) and Cayirli and Veral (2003). The interarrival time is assumed to be exponentially distributed with rates of $\lambda_1 = 0.1\ \frac{1}{h}$ and $\lambda_3 = 0.2\ \frac{1}{h}$ for arrival of emergency patients and inpatients according to Green et al. (2006), Peng et al. (2014), and Rising et al. (1973) . We choose $\rho = 0.2$ for the no-show rate based on Peng et al. (2014). According to Cayirli and Veral (2003), the waiting cost ratio of emergency patients over that of outpatients is set as three ($\frac{c_1^w}{c_2^w} = 3$), that for inpatients is 0.7 ($\frac{c_3^w}{c_2^w} = 0.7$), and that for overtime costs is 15 ($\frac{c^{over}}{c_2^w} = 15$). Therefore, we select $c_1^w = 90$, $c_2^w = 30$, $c_3^w = 21$, and $c^{over} = 450$.

For the threshold priority rule (P_3), the threshold parameter $\omega = 1\ h$ is selected based on Borgman et al. (2018). Instances are distinguished from each other by the number of total outpatients to schedule (N) and the number of periods in the planning horizon (T).

Table 3.2.: Overview of the parameter setting of the proposed algorithm

Tabu Search			Diversification			Neighborhood reduction		
Parameter	Range	Final choice	Parameter	Range	Final choice	Parameter	Range	Final choice
η_{max}	$[100, 1500]$	500	η_{div}	$[10, 500]$	25	θ	$[0.3, 0.5]$	0.4
ϑ_{min}	$[1, 5]$	2	η_{prob}	$[5, 300]$	10	K	$[1, 10]$	3
ϑ_{max}	$[3, 10]$	5	η_{reset}	$[50, 100]$	50			

To specify the parameters for the algorithm, an extensive prenumerical study results in the parameters shown in Table 3.2. For each parameter, the analyzed range and the final choice are given.

To choose the appropriate combination of the number of replications for high- and low-fidelity simulation, we analyze instances as described above with a service rate of $\mu = 1$ ($\frac{1}{h}$), $N \in \{5, \cdots, 10\}$ patients, and $T = 10$ periods (i.e., six distinguished instances). We test nine different combinations of the number of replications for high-fidelity simulation ($HF \in \{1000, 2000, 5000\}$) and for low-fidelity simulation ($LF \in \{50, 100, 250\}$). The number of replications for high-fidelity simulation is based on the literature on simulation optimization in appointment systems, such as Klassen and Yoogalingam (2019). Each instance is solved with the proposed algorithm using the respective combination of HF and LF. Each instance's obtained solution is finally evaluated by simulation with 150000 replications.

Table 3.3 indicates the results. For each instance, "BKS" is the objective value of the best-know solution, and the "Gap" columns show the rounded-up percentages of the deviation from BKS for the objective value of the solution with the respective combination of LF and HF. Please note that the best-know solution is the final schedule found by the algorithm for each instance, where its objective value is reported in the table. The proposed algorithm's run time in seconds is reported in "CPU (s)." Finally, "Pri." displays the priority rule used.

In Table 3.3, when the algorithm uses $HF = 5000$, it is very accurate but needs considerable CPU time. In general, the CPU time increases in LF. The combination of $HF = 2000$ and $LF = 100$ also shows only minor differences in the objective value compared to BKS, and it is fast. To conclude, we use the combination of $HF = 2000$

Table 3.3.: Comparison of the relative gaps in the percentage and CPU time of the solutions obtained by the algorithm with respective replications for high- and low-fidelity simulation, evaluated by simulation with 150000 replications

Pri.	N	T	BKS	$HF = 1000$						$HF = 2000$						$HF = 5000$					
				$LF = 50$		$LF = 100$		$LF = 250$		$LF = 50$		$LF = 100$		$LF = 250$		$LF = 50$		$LF = 100$		$LF = 250$	
				Gap($\%$)	CPU (s)	Gap($\%$)	CPU (s)	Gap($\%$)	CPU (s)	Gap($\%$)	CPU (s)	Gap($\%$)	CPU (s)	Gap($\%$)	CPU (s)	Gap($\%$)	CPU (s)	Gap($\%$)	CPU (s)	Gap($\%$)	CPU (s)
P_1	5	10	727.63	1.50	28.4	0.00	30.6	0.00	53.5	1.50	57.7	0.00	48.9	0.00	93.3	0.00	146.2	0.00	144.2	0.00	169.3
	6	10	1035.96	0.00	36.9	0.00	45.5	0.00	70.7	0.00	82.1	0.00	80.5	0.00	114.6	0.00	162.6	0.00	199.2	0.00	203.1
	7	10	1409.35	0.07	46.4	0.07	55.8	0.07	108.4	0.00	114.7	0.00	111.6	0.00	159.9	0.00	267.2	0.00	222.1	0.00	316.0
	8	10	1819.36	0.01	72.6	0.01	78.2	0.01	126.9	0.01	163.0	0.01	151.2	0.01	203.9	0.00	317.0	0.00	335.1	0.00	346.2
	9	10	2253.45	0.03	81.8	0.03	103.8	0.03	213.4	0.03	205.8	0.03	195.2	0.03	243.4	0.00	494.8	0.00	467.8	0.00	511.8
	10	10	2714.53	1.06	117.6	0.00	163.5	0.00	329.3	0.05	196.4	0.04	279.4	0.04	328.4	0.00	543.5	0.00	683.6	0.00	628.9
	Ave.			0.44	64.0	0.02	79.6	0.02	150.4	0.26	136.6	0.01	144.5	0.01	190.6	0.00	321.8	0.00	342.0	0.00	362.5
P_2	5	10	729.56	0.00	27.2	0.00	32.0	0.00	50.9	0.57	57.3	0.00	48.4	0.00	80.3	0.00	125.4	0.00	142.8	0.00	149.3
	6	10	1039.71	0.00	36.6	0.00	48.6	0.00	105.9	0.00	82.6	0.00	86.9	0.00	118.1	0.00	197.1	0.00	195.0	0.00	217.2
	7	10	1415.30	0.07	54.8	0.07	69.8	0.07	104.4	0.00	153.6	0.00	124.3	0.00	176.9	0.00	331.9	0.00	310.9	0.00	356.4
	8	10	1828.16	0.01	74.6	0.01	100.8	0.01	190.6	0.00	139.0	0.01	161.9	0.01	216.0	0.00	411.2	0.00	418.3	0.00	437.1
	9	10	2265.85	0.03	108.9	0.03	106.9	0.03	211.9	0.30	182.8	0.03	207.1	0.03	305.8	0.00	511.8	0.00	548.5	0.00	579.0
	10	10	2731.35	0.00	128.3	0.00	136.1	0.00	286.3	0.00	191.9	0.00	236.9	0.00	355.0	0.00	780.7	0.00	631.8	0.00	783.1
	Ave.			0.02	71.8	0.02	82.4	0.02	158.3	0.15	134.5	0.01	144.3	0.01	208.7	0.00	393.0	0.00	374.6	0.00	420.3
$P_3(1)$	5	10	736.40	1.55	31.4	0.00	30.3	0.00	50.6	0.54	40.7	0.00	49.2	0.00	75.3	0.00	150.0	0.00	133.0	0.00	134.6
	6	10	1049.61	0.55	39.4	0.00	43.8	0.00	83.7	0.00	85.3	0.00	75.9	0.00	110.1	0.00	192.7	0.00	199.2	0.00	211.1
	7	10	1428.24	0.48	54.1	0.05	62.7	0.05	125.9	0.00	111.1	0.00	106.3	0.00	170.0	0.00	283.1	0.00	324.7	0.00	263.8
	8	10	1844.09	0.03	64.0	0.03	77.4	0.03	172.2	0.03	163.6	0.03	156.2	0.03	232.6	0.00	329.4	0.00	436.5	0.00	382.9
	9	10	2284.41	0.06	94.1	0.06	114.2	0.06	179.9	0.06	192.8	0.06	182.6	0.06	291.8	0.00	500.3	0.00	481.2	0.00	541.4
	10	10	2752.40	0.01	135.6	0.01	110.9	0.01	256.4	0.01	212.8	0.00	225.9	0.00	343.6	0.00	602.7	0.00	667.9	0.00	748.0
	Ave.			0.44	69.8	0.02	73.2	0.02	144.8	0.11	134.4	0.01	132.7	0.01	203.9	0.00	343.0	0.00	373.7	0.00	380.3

and $LF = 100$ in the upcoming analyses because it is fast, and, on average, the relative gap is very low.

It should be noted that the seed values for all high-fidelity simulation runs are equal. In addition, all low-fidelity simulation runs have the same seed value. The proposed solution approach is coded in Java. All tests are performed on a standard desktop computer with an Intel Core(TM) i7 Processor at 3.33 GHz, 32 GB of RAM, and Windows 7 Professional.

3.5.2. Performance of the algorithm

In this section, we analyze the reliability of the proposed algorithm by comparison with a full enumeration procedure combined with simulation with 2000 replications. The full enumeration procedure terminates if all feasible schedules ($\binom{N+T-1}{N}$) are evaluated or the time limit is reached. The best solution is reported as its outcome. Note that both solution approaches use simulation with 2000 replications, and simulation with 150000 replications are not applied due to comparison issues.

We test small and large instances with a service rate of $\mu = 1$ $(\frac{1}{h})$ and all of the other parameters as described above. For small instances, $N \in \{5, \cdots, 10\}$ patients and $T = 10$ periods are used; and for large instances, $N \in \{10, 15, 20\}$ patients and $T \in \{10, 15, 20\}$ periods are employed.

The results related to the comparison of the full enumeration procedure and the proposed algorithm are reported in Table 3.4 for small instances and in Table 3.5 for large instances. For each instance, the tables report the objective value of the best known solution found by any of the solution methods (in column BKS). For each solution method, the running time in seconds is reported in “CPU (s).” The labels “Alg.” and “F_E” refer to the proposed algorithm and the full enumeration procedure, respectively. The time limit for solving each instance is 24 hours. For large instances, the percentages of deviation from the BKS for each solution method are reported in “Gap.”

Table 3.4 shows that the proposed algorithm can find the optimal solutions for all small instances. The CPU time of the algorithm is significantly smaller than that of full enumeration. It shows a good first sign of the solution quality.

In Table 3.5, the full enumeration approach cannot finish within 24 hours except for one instance with $N = 10$ and $T = 10$. For the first instance, the solution of the proposed algorithm and the full enumeration approach is equal. For the other instances, the proposed algorithm finishes with better solutions than the full enumeration procedure after 24 hours, whose average relative gap is more than 38%. The CPU time for the

Table 3.4.: The evaluation of the algorithm performance- small instances

Pri.	N	T	BKS	Alg.	F_E
				CPU (s)	CPU (s)
P_1	5	10	705.46	48.9	125.4
	6	10	1019.93	80.5	357.7
	7	10	1409.40	111.6	921.8
	8	10	1807.47	151.2	1832.7
	9	10	2231.84	195.2	4766.2
	10	10	2734.95	279.4	9396.7
	Ave.			144.5	2900.1
P_2	5	10	707.28	48.4	154.3
	6	10	1023.58	86.9	435.7
	7	10	1414.65	124.3	1127.3
	8	10	1815.55	161.9	2099.4
	9	10	2245.87	207.1	5535.5
	10	10	2752.13	236.9	11372.4
	Ave.			144.3	3454.1
$P_3(1)$	5	10	712.11	49.2	138.2
	6	10	1033.22	75.9	389.2
	7	10	1427.42	106.3	997.2
	8	10	1830.07	156.2	1921.1
	9	10	2265.09	182.6	4978.9
	10	10	2773.73	225.9	9978.4
	Ave.			132.7	3067.2

Table 3.5.: The evaluation of the algorithm performance- large instances

Pri.	N	T	BKS	Alg.		F_E	
				Gap($\%$)	CPU (s)	Gap($\%$)	CPU (s)
P_1	10	10	2734.95	0.00	262.4	0.00	9396.7
	15	10	5309.48	0.00	631.4	10.15	86400.0
	20	10	8231.87	0.00	1337.3	11.38	86400.0
	10	15	1971.69	0.00	473.8	43.98	86400.0
	15	15	4335.63	0.00	968.7	36.08	86400.0
	20	15	7068.09	0.00	2622.7	30.65	86400.0
	10	20	1421.79	0.00	416.4	103.87	86400.0
	15	20	3487.02	0.00	1836.5	69.90	86400.0
	20	20	6192.29	0.00	3500.7	52.51	86400.0
	Ave.			0.00	1338.9	39.84	77844.1
P_2	10	10	2752.13	0.00	222.9	0.00	11372.4
	15	10	5346.05	0.00	725.9	9.97	86400.0
	20	10	8296.22	0.00	1488.1	11.20	86400.0
	10	15	1987.65	0.00	504.7	44.06	86400.0
	15	15	4380.08	0.00	929.0	35.40	86400.0
	20	15	7153.33	0.00	2366.1	29.81	86400.0
	10	20	1435.24	0.00	634.1	103.15	86400.0
	15	20	3534.01	0.00	2095.3	68.42	86400.0
	20	20	6283.27	0.00	3533.3	51.21	86400.0
	Ave.			0.00	1388.8	39.25	78063.6
$P_3(1)$	10	10	2773.73	0.00	206.2	0.00	9978.5
	15	10	5375.40	0.00	566.8	9.86	86400.0
	20	10	8326.95	0.00	1433.2	11.10	86400.0
	10	15	2016.50	0.00	529.2	43.28	86400.0
	15	15	4419.08	0.00	1439.6	34.80	86400.0
	20	15	7207.70	0.00	2991.7	29.27	86400.0
	10	20	1467.43	0.00	603.2	100.48	86400.0
	15	20	3585.28	0.00	1685.6	66.73	86400.0
	20	20	6348.95	0.00	3622.1	50.17	86400.0
	Ave.			0.00	1453.1	38.41	77908.7

proposed solution method is on average 1393.9 seconds and increases in N and T. However, an increase in N has a stronger impact than an increase in T.

To summarize, the proposed algorithm is highly efficient and can find a good-quality solution (optimal for all small instances) faster than full enumeration.

3.5.3. Sensitivity analysis of the arrival parameters of patient classes

In this section, we compare the obtained schedule of outpatients for three priority rules. The instance benchmark is based on Section 3.5.1, with a service rate of $\mu = 2\ \frac{1}{h}$, $T = 16$ periods, and $N = 11$ outpatients to schedule. We conduct sensitivity analysis for one arrival parameter of the patient classes (either λ_1, N, or λ_3) by increasing them while keeping the other values constant.

The expected load of the system $\widehat{\rho}$ is calculated by Equation 3.4.

$$\widehat{\rho} = \frac{N \cdot (1-\rho)}{\mu \cdot T \cdot d} + \frac{\lambda_2 + \lambda_3}{\mu} \tag{3.4}$$

, where the first part is the mean load generated by serving all outpatients in a planning horizon. The second part is the load generated by the arrival of emergency patients and inpatients together. Tables 3.6 to 3.8 display the results for varying the arrival parameters of inpatients, emergency patients, and outpatients. In all tables, the changes in the arrival parameters result in $\widehat{\rho}$ ranging from 0.7 to 1.35 with a step size of 0.05. In those tables, the last columns show the obtained cumulative number of outpatients in each period. The numbers in white highlight the difference in the structure of the solution in the associated period, where the appointment system uses different priority rules.

In Table 3.6, in the slots where the cumulative scheduled outpatients are different (the white numbers), the following findings are noted. When the threshold priority rule $P_3(1)$ is used, outpatients are scheduled closer to the end of the planning horizon compared to the case where the static priority rule P_1 is applied. The main reason is that the former is likely to have waiting inpatients who reach their threshold. Therefore, it is cost-beneficial to schedule outpatients later and incur lower costs for a waiting outpatient. In addition, the table clearly shows that priority rule P_2 sometimes has the same effect as priority rule P_1 on the obtained outpatient schedule and sometimes affects them in a manner similar to priority rule $P_3(1)$. The reason is the similarity of the substitutive priority rule and the threshold priority rule for giving an inpatient the priority higher than that of outpatients.

Table 3.6.: The effects of inpatient arrival rates on the structure of the optimal schedule where the system uses different priority rules

λ_3	$\widehat{\rho}$	Pri.	Cumulative number of patients in each period															
0.2	0.70	P_1	1	2	3	4	5	5	6	7	8	8	9	10	10	11	11	11
		P_2	1	2	3	4	5	5	6	7	8	8	9	10	10	11	11	11
		$P_3(1)$	1	2	3	4	5	5	6	7	8	8	9	10	10	11	11	11
0.3	0.75	P_1	2	3	3	4	5	6	6	7	8	8	9	10	10	11	11	11
		P_2	2	3	3	4	5	6	6	7	8	8	9	10	10	11	11	11
		$P_3(1)$	2	3	3	4	5	6	6	7	8	8	9	10	10	11	11	11
0.4	0.80	P_1	2	3	4	4	5	6	7	7	8	9	9	10	10	11	11	11
		P_2	2	3	4	4	5	6	7	7	8	9	9	10	10	11	11	11
		$P_3(1)$	2	3	4	4	5	6	7	7	8	9	9	10	10	11	11	11
0.5	0.85	P_1	2	3	4	4	5	6	6	7	8	8	9	10	10	11	11	11
		P_2	2	3	4	4	5	6	6	7	8	8	9	10	10	11	11	11
		$P_3(1)$	2	3	4	4	5	6	6	7	8	8	9	10	10	11	11	11
0.6	0.90	P_1	2	3	4	5	5	6	7	7	8	8	9	10	10	11	11	11
		P_2	2	3	4	4	5	6	6	7	8	8	9	10	10	11	11	11
		$P_3(1)$	2	3	4	4	5	6	6	7	8	8	9	10	10	11	11	11
0.7	0.95	P_1	2	3	4	4	5	6	6	7	8	8	9	10	10	11	11	11
		P_2	2	3	4	4	5	6	6	7	8	8	9	10	10	11	11	11
		$P_3(1)$	2	3	4	4	5	6	6	7	8	8	9	10	10	11	11	11
0.8	1.00	P_1	2	3	4	5	5	6	7	7	8	8	9	9	10	10	11	11
		P_2	2	3	4	5	5	6	6	7	8	8	9	9	10	10	11	11
		$P_3(1)$	2	3	4	5	5	6	6	7	8	8	9	9	10	10	11	11
0.9	1.05	P_1	2	3	4	5	5	6	7	7	8	8	9	10	10	11	11	11
		P_2	2	3	4	4	5	6	6	7	7	8	9	9	10	10	11	11
		$P_3(1)$	2	3	4	4	5	6	6	7	7	8	9	9	10	10	11	11
1.0	1.10	P_1	2	3	4	5	5	6	7	7	8	8	9	9	10	10	11	11
		P_2	2	3	4	5	5	6	7	7	8	8	9	10	10	11	11	11
		$P_3(1)$	2	3	4	5	5	6	6	7	7	8	9	9	10	10	11	11
1.1	1.15	P_1	2	3	4	5	5	6	6	7	7	8	8	9	9	10	11	11
		P_2	2	3	4	5	5	6	6	7	7	8	8	9	9	10	11	11
		$P_3(1)$	2	3	4	5	5	6	6	7	7	8	8	9	9	10	11	11
1.2	1.20	P_1	2	3	4	5	5	6	6	7	7	8	8	9	9	10	11	11
		P_2	2	3	4	5	5	6	6	7	7	8	8	9	9	10	10	11
		$P_3(1)$	2	3	4	5	5	6	6	7	7	8	8	9	9	10	10	11
1.3	1.25	P_1	2	3	4	4	5	5	6	6	7	7	8	8	9	9	10	11
		P_2	2	3	4	4	5	5	6	6	7	7	8	8	9	9	10	11
		$P_3(1)$	2	3	4	4	5	5	6	6	6	7	7	8	8	9	10	11
1.4	1.30	P_1	2	3	4	5	5	5	6	6	7	7	8	8	9	9	10	11
		P_2	2	3	4	4	5	5	6	6	7	7	8	8	9	9	10	11
		$P_3(1)$	2	3	4	4	5	5	5	6	6	6	7	7	8	8	9	11
1.5	1.35	P_1	2	3	4	4	5	5	6	6	6	7	7	8	8	9	10	11
		P_2	2	3	4	4	5	5	5	6	6	7	7	8	8	9	9	11
		$P_3(1)$	2	3	4	4	5	5	5	6	6	6	7	7	8	8	9	11

There is one counterexample, the instance with $\widehat{\rho} = 1.1$, in which for the appointment system with priority rule P_2, the white numbers are larger than those for P_1 and $P_3(1)$, meaning that the outpatients are scheduled one slot sooner compared to when other priority rules are applied. This is because of the simulation and the proposed algorithm errors, which can be solved by increasing the number of replications for low-fidelity and high-fidelity simulations, as described in Section 3.5.1. We reran that instance with $LF = 500$ and $HF = 150000$ to check this; and we observed that the same structure occurred as for the effects of the priority rules P_1, P_2, and $P_3(1)$ on the obtained solution which are explained sooner.

Tables 3.7 and 3.8 show the results of the sensitivity analyses of λ_1 and N, respectively. The findings are similar to what we explain for Table 3.6.

Comparing all tables together, we detect that a high inpatient arrival rate impacts the schedule more than a high emergency arrival rate or a high number of outpatients to schedule for different priority rules. When the inpatient rate increases, more inpatients wait. In $P_3(1)$, inpatients reach the threshold more often. Additionally, when the system has priority rule P_2, in the case of a no-show, it is more likely that an inpatient exists and waits in the system so that she can be substituted.

In summary, we can conclude that to optimize the outpatients' schedule, a manager has to consider which priority rule the system uses because different priority rules result in having different schedules. We also show that at higher loads, the differences between the schedules are more vivid, specifically when the change in the load is caused by inpatients. Generally, when the threshold priority rule is applied, the solution tends to schedule one outpatient closer to the end of the planning horizon, to incur less costs for outpatient waiting even though penalizing overtime costs more.

3.6. Conclusion and further research

In this paper, we investigate the outpatient scheduling problem when the system has a specific priority rule as it arises in radiology departments. The priority rules are used to control the access of patients who are waiting to be served. When the appointment system uses a specific priority rule, it may disrupt the initial schedule and affect the system's performance. To study the effects of the priority rules on the schedule of outpatients, a simulation optimization approach with a tabu search is designed. The proposed algorithm uses a neighborhood reduction technique in which low-fidelity simulation is applied to speed up the procedure of finding the optimal solution. In every iteration, the neighborhood is built by applying a set of moves to the incumbent solution. Then, low-fidelity simulation is applied to select a subset of neighboring solutions

Table 3.7.: The effects of emergency arrival rates on the structure of the optimal schedule where the system uses different priority rules

λ_1	$\widehat{\rho}$	Pri.	Cumulative number of patients in each period															
0.1	0.70	P_1	1	2	3	4	5	5	6	7	8	8	9	10	10	11	11	11
		P_2	1	2	3	4	5	5	6	7	8	8	9	10	10	11	11	11
		$P_3(1)$	1	2	3	4	5	5	6	7	8	8	9	10	10	11	11	11
0.2	0.75	P_1	2	3	3	4	5	6	6	7	8	8	9	10	10	11	11	11
		P_2	2	3	3	4	5	6	6	7	8	8	9	10	10	11	11	11
		$P_3(1)$	2	3	3	4	5	6	6	7	8	8	9	10	10	11	11	11
0.3	0.80	P_1	2	3	4	4	5	6	7	7	8	8	9	10	10	11	11	11
		P_2	2	3	3	4	5	6	6	7	8	8	9	10	10	11	11	11
		$P_3(1)$	2	3	3	4	5	6	6	7	8	8	9	10	10	11	11	11
0.4	0.85	P_1	2	3	4	5	5	6	6	7	8	8	9	10	10	11	11	11
		P_2	2	3	4	5	5	6	6	7	8	8	9	10	10	11	11	11
		$P_3(1)$	2	3	4	5	5	6	6	7	8	8	9	10	10	11	11	11
0.5	0.90	P_1	2	3	4	4	5	6	6	7	8	8	9	10	10	11	11	11
		P_2	2	3	4	5	5	6	7	7	8	8	9	10	10	11	11	11
		$P_3(1)$	2	3	4	5	5	6	7	7	8	8	9	10	10	11	11	11
0.6	0.95	P_1	2	3	4	5	5	6	6	7	8	8	9	9	10	11	11	11
		P_2	2	3	4	5	5	6	7	7	8	9	9	10	10	11	11	11
		$P_3(1)$	2	3	4	5	5	6	6	7	8	8	9	9	10	11	11	11
0.7	1.00	P_1	2	3	4	5	5	6	6	7	8	8	9	9	10	10	11	11
		P_2	2	3	4	5	5	6	6	7	8	8	9	9	10	10	11	11
		$P_3(1)$	2	3	4	5	5	6	6	7	8	8	9	9	10	10	11	11
0.8	1.05	P_1	2	3	4	5	5	6	7	7	8	8	9	9	10	10	11	11
		P_2	2	3	4	5	5	6	7	7	8	8	9	9	10	10	11	11
		$P_3(1)$	2	3	4	5	5	6	7	7	8	8	9	9	10	10	11	11
0.9	1.10	P_1	2	3	4	5	5	6	6	7	8	8	9	9	10	10	11	11
		P_2	2	3	4	5	5	6	6	7	8	8	9	9	10	10	11	11
		$P_3(1)$	2	3	4	5	5	5	6	7	7	8	8	9	10	10	11	11
1.0	1.15	P_1	2	3	4	4	5	6	6	7	7	8	8	9	9	10	11	11
		P_2	2	3	4	4	5	5	6	6	7	7	8	8	9	9	10	11
		$P_3(1)$	2	3	4	4	5	5	6	6	7	7	8	8	9	9	10	11
1.1	1.20	P_1	2	3	4	4	5	5	6	6	7	7	8	8	9	9	10	11
		P_2	2	3	4	4	5	5	6	6	7	7	8	8	9	9	10	11
		$P_3(1)$	2	3	4	4	5	5	5	6	6	7	7	8	8	9	10	11
1.2	1.25	P_1	2	3	3	4	4	5	5	6	6	7	7	8	8	9	9	11
		P_2	2	3	4	4	5	5	5	6	6	7	7	8	8	9	9	11
		$P_3(1)$	2	3	3	4	4	5	5	6	6	7	7	8	8	9	9	11
1.3	1.30	P_1	2	3	4	4	5	5	5	6	6	6	7	7	8	8	9	11
		P_2	2	3	4	4	5	5	5	6	6	6	7	7	8	8	9	11
		$P_3(1)$	2	3	4	4	5	5	5	6	6	6	7	7	8	8	9	11
1.4	1.35	P_1	2	3	4	4	5	5	5	6	6	6	7	7	8	8	9	11
		P_2	2	3	4	4	5	5	5	6	6	6	7	7	8	8	9	11
		$P_3(1)$	2	3	4	4	5	5	5	6	6	6	7	7	8	8	9	11

Table 3.8.: The effects of the number of outpatients to schedule on the structure of the optimal schedule where the system uses different priority rules

N	$\widehat{\rho}$	Pri.	Cumulative number of patients in each period															
11	0.70	P_1	1	2	3	4	5	5	6	7	8	8	9	10	10	11	11	11
		P_2	1	2	3	4	5	5	6	7	8	8	9	10	10	11	11	11
		$P_3(1)$	1	2	3	4	5	5	6	7	8	8	9	10	10	11	11	11
12	0.75	P_1	2	3	4	4	5	6	7	8	8	9	10	11	11	12	12	12
		P_2	2	3	4	4	5	6	7	8	8	9	10	11	11	12	12	12
		$P_3(1)$	2	3	4	4	5	6	7	8	8	9	10	11	11	12	12	12
13	0.80	P_1	2	3	4	5	6	7	7	8	9	10	11	11	12	13	13	13
		P_2	2	3	4	5	6	7	7	8	9	10	11	11	12	13	13	13
		$P_3(1)$	2	3	4	5	6	7	7	8	9	10	11	11	12	13	13	13
14	0.85	P_1	2	3	4	5	6	7	8	**9**	9	10	11	12	13	13	14	14
		P_2	2	3	4	5	6	7	8	**8**	9	10	11	12	13	13	14	14
		$P_3(1)$	2	3	4	5	6	7	8	**8**	9	10	11	12	13	13	14	14
15	0.90	P_1	2	3	4	5	6	7	8	9	10	11	12	13	13	14	15	15
		P_2	2	3	4	5	6	7	8	9	10	11	12	13	13	14	15	15
		$P_3(1)$	2	3	4	5	6	7	8	9	10	11	12	13	13	14	15	15
16	0.95	P_1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	16
		P_2	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	16
		$P_3(1)$	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	16
17	1.00	P_1	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	17
		P_2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	17
		$P_3(1)$	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	17
18	1.05	P_1	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
		P_2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
		$P_3(1)$	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
19	1.10	P_1	3	4	5	6	7	8	9	10	11	12	13	14	16	17	18	19
		P_2	3	4	5	6	7	8	9	10	11	12	13	14	16	17	18	19
		$P_3(1)$	3	4	5	6	7	8	9	10	11	12	13	14	16	17	18	19
20	1.15	P_1	3	4	5	6	7	8	9	10	11	13	14	15	16	17	18	20
		P_2	3	4	5	6	7	8	9	10	11	13	14	15	16	17	18	20
		$P_3(1)$	3	4	5	6	7	8	9	10	11	13	14	15	16	17	18	20
21	1.20	P_1	3	4	5	6	8	9	10	11	12	13	14	15	16	**18**	19	21
		P_2	3	4	5	6	8	9	10	11	12	13	14	15	16	**18**	19	21
		$P_3(1)$	3	4	5	6	8	9	10	11	12	13	14	15	16	**17**	19	21
22	1.25	P_1	3	4	5	**7**	8	9	10	11	12	13	14	15	16	18	19	22
		P_2	3	4	5	**6**	8	9	10	11	12	13	14	15	16	18	19	22
		$P_3(1)$	3	4	5	**6**	8	9	10	11	12	13	14	15	16	18	19	22
23	1.30	P_1	3	4	5	6	8	9	10	11	12	13	14	16	17	18	19	23
		P_2	3	4	5	6	8	9	10	11	12	13	14	16	17	18	19	23
		$P_3(1)$	3	4	5	6	8	9	10	11	12	13	14	16	17	18	19	23
24	1.35	P_1	3	4	6	7	8	9	10	11	12	13	14	15	16	18	19	24
		P_2	3	4	6	7	8	9	10	11	12	13	14	15	16	18	19	24
		$P_3(1)$	3	4	6	7	8	9	10	11	12	13	14	15	16	18	19	24

to finally be evaluated by high-fidelity simulation. Two diversification mechanisms, the probabilistic phase and randomly chosen phase, are applied to explore the unvisited solution space.

Our computational tests reveal the effectiveness and efficiency of the proposed algorithm to solve the problems by comparing the solutions obtained from a full enumeration procedure. The numerical results indicate that when decision-makers want to optimize the timely schedules of outpatients, they must consider what priority rules the system uses. Specifically, when the system experiences a high load, the optimal schedule of outpatients differs depending on the priority rule used by the system. More precisely, for the higher load triggered by inpatients, the optimal schedule is more likely to be different. The results also show that if the threshold priority rule is used, in the obtained solution, outpatients are scheduled near the end of the planning horizon, resulting in lower outpatient waiting costs, even when incurring further overtime costs.

Further research may study other system assumptions, such as patient abandonment. Some studies indicate that patients are not eager to wait too long for their service, and they might leave the system without being served. Another issue to analyze is how to include patients who arrive earlier or later than their appointment times. Unpunctuality of patients is a major problem as it disrupts the smooth running of various clinical specialties.

From the method perspective, one should focus on designing alternative solution methods and alternative evaluation methods. For the latter, low- or high-fidelity simulations to evaluate the performance of the system can be replaced by approximation methods.

4. A Stationary-backlog Carryover and Fluid Approximation for Time-dependent Queues with Overloading

Co-authors:

Raik Stolletz
Chair of Production Management, Business School, University of Mannheim, Germany

Working paper.

Abstract:

Time-dependent queues are used to represent real-world systems, such as appointment systems in healthcare or runway systems in airports. In these systems, the rate of arrivals may temporarily surpass the service capacity, leading to overloading. This paper analyzes a single server time-dependent queue with exponentially distributed inter-arrival and service time. We propose a hybrid approximation method which is on basis of the Stationary Backlog Carryover (SBC) and the Point-wise Stationary Fluid Flow Approximation (PSFFA) approaches. A mechanism is applied to adjust the parameters of the proposed approximation method when the system transfers from an overloaded period to an underloaded one or the other way around.

A simulation benchmark confirms that the proposed hybrid method approximates the performance measures of the time-dependent queueing system for different parameter configurations well. The results suggest tuning the parameters of the approximation method dramatically increases its accuracy. Numerical experiments show the quality of the proposed approximation method in comparison with the SBC, the PSFFA, and the Fluid approximation approaches. The proposed hybrid method outperforms them for queueing systems with sinusoidal arrival functions and for benchmark instances from the literature.

keywords: Stationary Backlog Carryover & Point-wise Stationary Fluid Flow Approximation & Time-dependent Queues with Overloading

4.1. Introduction

Queueing systems with time-dependent parameters are used for modeling real-world systems containing queues. For example, runway systems in airports or appointment systems in outpatient clinics can be modeled by time-dependent queueing systems (Stolletz, 2008b; Brahimi and Worthington, 1991). In such systems, the arrival rates may exceed the service capacity (rate) for some extended time, which leads to overloading. For a runway system in an airport, redirecting the flights from a neighboring airport or changing the weather conditions may cause temporarily overloading. In an outpatient clinic, the arrival rate of patients may rise more than the provided capacity during winters or pandemics. When the system is in overload, queues form fast, and arrivals are served after waiting for a potentially long wait.

Time-dependent queues are difficult to analyze, even for the Markovian setting. The closed-form and explicit solutions for them are only obtained for special cases (Schwarz et al., 2016). Simulation provides a highly accurate estimate of the expected performance measures in time-dependent queueing systems, but its computational time is long. Compared with simulation, approximation methods are more computationally efficient (Moore, 1975). Furthermore, fast and accurate performance approximations for decision support are needed. They can be integrated into optimization approaches, where a solution is evaluated. If the queueing model is just one component of a larger model (*e.g.*, dynamic control) or is being used as part of an iterative computation system, it might be desirable to have an approximation approach that is fast enough to keep the cost of computation within reasonable bounds (Rider, 1976).

The analysis of time-dependent queueing systems has a long tradition dating back to the paper of Kolmogrov (1931). Since then, it has received considerable attention from academics, as shown in the literature reviews such as Ingolfsson et al. (2007) and Schwarz et al. (2016). It is demonstrated that only a limited subset of approximation approaches address time-dependent queues with overload, where they need special conditions held to approximate accurately.

The purpose of this paper is to design an approximation method that fast and accurately analyzes an $M(t)/M/1$ queue where overloading situations occur. A hybrid approximation method that combines the Stationary Backlog Carryover (SBC) and the Point-wise Stationary Fluid Flow Approximation (PSFFA) approaches is proposed. The literature demonstrates that the SBC approach approximates the time-dependent queues in underload accurately, for example see Stolletz (2008a); Selinka et al. (2016). The PSFFA approach is based on the Fluid approximation method. The Fluid approximation performs very well for approximating the queues with overloading, for example see Wang et al. (1996); Mandelbaum et al. (1998); Jiménez and Koole (2004). In this paper, we improve the PSFFA approach within the proposed hybrid approximation method to increase its effectiveness. The approximated expected utilization used in the core of the algorithm is modified. The proposed hybrid approximation method applies the SBC approach for underloaded periods and the modified PSFFA approach for overloaded ones. A mechanism is also used to adjust the approximation parameters when a transition from an overloaded period to underloaded one, or the other way around, happens. More precisely, in transitions, the output of one of the approximation method adjusted such that it becomes the input of the another approximation method.

The quality of the proposed hybrid method is evaluated by comparing the resulting performance measures from simulation. A simulation benchmark confirms that the proposed method approximates the performance measure of the time-dependent queueing system with overload situations very well. We analyze the quality of the proposed hybrid method for different parameter configurations and compare them with simulation. The results suggest tunning the parameters of the approximation method dramatically increases its accuracy. In addition, numerical experiments indicate the high quality of the proposed hybrid approximation method in comparison with the SBC, the PSFFA, and the Fluid approximation approaches in isolation.

The remaining sections of this paper are structured as follows: Section 4.2 reviews the literature. In Section 4.3, we describe the SBC, the PSFFA, and the Fluid approximation approaches. We also comprehensively explain the proposed hybrid approximation method. Numerical studies are performed and the results are presented in Section 4.4. Finally, Section 4.5 draws some conclusions from this study.

4.2. Literature review

In this section, we review the literature on performance approximation approaches for time-dependent queues. We consider the approximation methods that can evaluate the queues in which overloading occurs in some extended time. The papers reviewed consider a single-stage queue that features: 1- unlimited waiting room capacity, 2- stochastic arrivals with time-dependent rates, 3- homogeneous arrivals (*i.e.*, no different classes of arrivals and no priority queues), and 4- the server(s) serving all arrivals (*i.e.*, no retrial or abandonment). We apply a structure similar to what is introduced in

Schwarz et al. (2016) to categorize the papers that fit the previous selecting criteria. They classify the literature for time-dependent queues into three main categories:

- Analytical solution approaches
- Approaches based on modified system characteristics
- Approaches based on models with piecewise constant parameters

1- *Analytical solution approaches* are based on ordinary differential equations (ODE), which can be numerically solved for Markovian systems (Kleinrock, 1975). The approach forms the Chapman-Kolmogorov equations (CKEs), and it solves them. Kolmogrov (1931) proposed the approach for the $M(t)/M/c$ queue. It is usually used as a benchmark to compare the approximation method, such as in Rider (1976); Green and Kolesar (1991, 1995); Ingolfsson et al. (2007). The computational time of ODE solvers is long.

2- *Approaches based on modified system characteristics* replace a deterministic continuum with arriving discrete jobs. The Fluid approximation proposed by Newell (1971) is a prime example. Jung and Lee (1989) model repair facilities in a military context by a $G(t)/G/c(t)$ queue and use the Fluid approximation. Jiménez and Koole (2004) analyze the $M(t)/M/c$ queues and improve the approximation limits proposed by Newell (1971) and Mandelbaum et al. (1998). The main advantage of the Fluid approximation is to provide a good approximation for the performance measure when the system is in overload. However, when the system is underloaded or critically loaded, it performs poorly, see Mandelbaum et al. (1998) for more detailed discussion.

The Fluid approximation also gains additional relevance as an integral part of other approximation approaches, namely, the PSFFA approach and Coordinate Transformation Technique (CTT), which will be described next.

The PSFFA approach coined by Wang et al. (1996) is a similar approach to the Fluid approximation. It is modified such that the outflow from the system depends on the server utilization. The PSFFA approach combines the deterministic Fluid approximation with the steady-state expected utilization formula to integrate random variation. Agnew (1976) reports the first ideas of relating the outflow of a fluid queue to the expected number of jobs in the system for an $G(t)/G/1$ queue. Filipiak (1984) considered a traffic assignment problem with a $M(t)/M/1$ queue and applies the PSFFA approach in the proposed optimization approach. It is demonstrated that it performs well for systems in underload and it becomes similar to the Fluid approximation in overloaded periods. However, it reaches the steady-state too rapidly. This leads to an overestimation of peaks and underestimation of valleys for quickly varying input rates.

3- One of the successful algorithms among *approaches based on models with piecewise constant parameters* is the SBC approach. It is developed by Stolletz (2008a) for the $M(t)/M/c(t)$ queues. It divides the planning horizon into non-overlapping equal-sized intervals. The SBC approach consists of two steps. In the first step, the backlog of unserved arrivals based on the stationary loss system is calculated and the expected utilization is approximated. In the second step, the approach uses the formula for the stationary approximation by employing a modified arrival rate which results from the expected utilization from the first step. Stolletz (2008b) models runway systems in airports by a $M(t)/G(t)/1$ queueing system and extends the SBC approach to analyze it. $M(t)/G/c(t)$ queueing systems are considered by Stolletz (2011). The SBC approach considers the dependencies between successive intervals. It also performs well for the underloaded and temporarily overloaded queues. However, it does not perform remarkably when the overloaded periods are extended or overloading starts at the beginning.

The discrete-time methods (DTMs) are based on the idea of exchanging continuous time with discrete points in time. In these approaches, the planning horizon is split into equal-sized intervals. They allow various arrival processes, *e.g.* time-dependent or batch arrivals, as long as the probability of the number of arrivals can be calculated for each time interval. Alfa (1990) and Alfa and Chen (1991) apply the DTM for $M(t)/D/1$ and $M(t)/G/1$ queues, respectively. They modify the DTM to avoid using computationally expensive auxiliary variables to speed up the approximation approach. The main advantages of DTMs are their accuracy and their flexibility. However, their state space rapidly grows with an increasing waiting room capacity. It is time-consuming and it incurs approximation errors if the system does not operate with time slots.

CTT is proposed by Kimber and Hollis (1977) for an $M(t)/M/1$ system. As mentioned previously, the core idea of CTT is to combine the Fluid approximation and the Pointwise Stationary Approximation (PSA) approaches. This approach fits this group better based on its features, rather than *approaches based on modified system characteristics*. In CTT, a transformation of a steady-state queueing formula is used to calculate an interval's performance. The transformation converges to the steady-state formula using the PSA approach for decreasing traffic intensities and to a deterministic Fluid approximation for increasing traffic intensities. Catling (1977) studies an $M(t)/G/1$ system using CTT and in Kimber and Daly (1986), CTT is used to analyze a $G(t)/G(t)/1$ queueing system. The PSA worsens as the maximum traffic intensity increases, which impacts the quality of CTT, see Green and Kolesar (1991).

It is observed in the relevant literature (meeting the features introduced at the beginning of this section) on the performance approximation methods for time-dependent queueing systems with overload have limited coverage. The exact and accurate approximation methods, such as analytical approaches and DTMs, require a long time to execute. The fast approximation approaches require certain criteria to hold in order to accurately perform. This leads to a reduction of the quality of the approximated solution when those criteria are not met, such as the Fluid approximation in systems with underload and the SBC approaches for queueing systems where overloading starts at the beginning. The literature lacks an accurate and fast approximation method that is flexible enough to be used in different configurations. This paper aims at filling this gap by introducing the approximation method that combines the SBC approach with the modified PSFFA approach.

4.3. Performance approximation method

In this section, we describe the studied $M(t)/M/1$ queueing system and explain the proposed method to approximate the performance measures. The planning horizon is considered between $[0, T]$ and it is divided into a set of non-overlapping periods $i \in \{1, \cdots, I\}$ with a length of $d = \frac{T}{I}$ each, meaning that period i starts from time t_{i-1} to time $t_i = t_{i-1} + d$. The arrival rates are time-dependent and are assumed to be piecewise constant. More precisely, during period i, the inter-arrival time is exponentially distributed with a known rate of $\lambda_i \geq 0$ per time unit. The service time is exponentially distributed (i.i.d) with a rate of $\mu > 0$ per time unit. Items are served based on a first-come-first-served discipline. If the server is busy, arrivals wait in the waiting room with unlimited capacity. We refer to period i where $\frac{\lambda_i}{\mu} < 1$ as “underloaded,” and accordingly, period i as “overloaded” if $\frac{\lambda_t}{\mu} \geq 1$.

Subsection 4.3.1 explains the SBC approach to approximate the $M(t)/M/1$ queues. Subsection 4.3.2 presents the core ideas of the Fluid approximation and the PSFFA approaches. The modifications for the PSFFA approach are described in this subsection. Finally, Subsection 4.3.3 comprehensively explains the proposed hybrid method that combines the SBC and modified PSFFA approaches to approximate time-dependent queues when overload situations occur during some periods.

4.3.1. The core ideas of the SBC approach

In the SBC approach, periods are further divided into smaller intervals $j \in J$ of an equal length of l and with constant arrival rates of λ_i (Stolletz, 2008a). We use the average processing time ($l = \mu^{-1}$) for the interval length, as originally proposed by Stolletz (2008a). The SBC approach consists of two steps: In Step 1, the expected utilization is approximated. In Step 2, the waiting-based performance measures are calculated based on the approximated expected utilization:

Step 1: In each interval (j), the SBC approach applies a stationary loss model, *i.e.*, $M/M/1/1$, with an artificial arrival rate $\widetilde{\lambda_{i,j}}$. This yields an artificial probability of blocking ($P_{i,j}^{BL}$). A backlog rate ($b_{i,j}$) is generated from (artificially) blocked jobs. These artificially blocked jobs are carried over as additional arrivals in the subsequent interval $(i, j+1)$ or the first interval of the next period $(i+1, 1)$ if j is the last interval of period i ($j = |J|$).

The artificial arrival rate is calculated by summing the original arrival rate in period i (λ_i) and the backlogged rate from previous interval (Equation (4.1)).

$$\widetilde{\lambda_{i,j}} = \lambda_i + b_{i,j-1} \qquad \forall i,j,\ b_{1,0} = 0, b_{i>1,0} = b_{i-1,|J|} \tag{4.1}$$

Then, the blocking probability of an arriving item for an $M/M/1/1$ queue is approximated via Equation (4.2) (Gross et al., 2008).

$$P_{i,j}^{BL} = \frac{\widetilde{\lambda_{i,j}}}{\widetilde{\lambda_{i,j}} + \mu} \qquad \forall i,j \tag{4.2}$$

Based on the blocking probability, the backlog rate ($b_{i,j}$) of (artificially) blocked arrivals in the loss model is approximated (Equation (4.3)). This value will be carried over as an additional arrival rate into the subsequent interval.

$$b_{i,j} = \widetilde{\lambda_{i,j}} P_{i,j}^{BL} \qquad \forall i,j \tag{4.3}$$

Finally, the expected utilization $E[U_{i,j}]$ for a stationary loss model $M/M/1/1$ is approximated through Equation (4.4).

$$E[U_{i,j}] = \frac{\widetilde{\lambda_{i,j}}(1 - P_{i,j}^{BL})}{\mu} \qquad \forall i,j \tag{4.4}$$

Step 2: A stationary waiting model (*i.e.*, $M/M/1/\infty$) is used in each interval to approximate the waiting-based performance measures. To obtain approximations of the performance measures of interest, the modified arrival rates ($\lambda_{i,j}^{MAR}$) serve as input to the stationary queueing model. $\lambda_{i,j}^{MAR}$ is set such that the approximated expected utilization ($E[U_{i,j}]$) of Step 1 is reached (Equation 4.5).

$$\lambda_{i,j}^{MAR} = E[U_{i,j}]\mu \qquad \forall i,j \tag{4.5}$$

This modified arrival rate is used to approximate the expected work-in-process ($E[L_{i,j}]$) and the expected waiting time ($E[W^q_{i,j}]$) of a stationary $M/M/1$ queue via Equations (4.6) and (4.7).

$$E[L_{i,j}] = \frac{\lambda^{MAR}_{i,j}}{(\mu - \lambda^{MAR}_{i,j})} \qquad \forall i,j \tag{4.6}$$

$$E[W^q_{i,j}] = \frac{\lambda^{MAR}_{i,j}}{\mu(\mu - \lambda^{MAR}_{i,j})} \qquad \forall i,j \tag{4.7}$$

One of the advantages of the SBC approach is that it can be combined with any performance evaluation approach for stationary queues in both steps. It also approximates the performance measures very well for systems in underload, provided that the “right” length for intervals is chosen (Stolletz and Lagershausen, 2013). Choosing a too short interval length results in underestimation of the performance measures in transient phases and choosing a too long value will result in overestimation.

4.3.2. The core ideas of the Fluid approximation and the PSFFA approaches

The Fluid approximation and the PSFFA approaches replace discrete jobs with a continuum. Their core idea is based on “fluid dynamic flow equation,” which states that according to the flow conservation principle, the rate of change equals the difference between aggregate inflow and outflow in every point of time (Newell, 1971; Wang et al., 1996).

Let t be a point of time in period i (*i.e.*, $t_{i-1} \le t < t_i$). We define $x(t)$ as the ensemble expected number of items in the system at time t. Given two functions $f_{in}(t)$ and $f_{out}(t)$, the ensemble average flow in and flow out of the system at time t, respectively, the fluid flow equation can be shown by:

$$\frac{dx(t)}{dt} = f_{in}(t) - f_{out}(t), \tag{4.8}$$

where $\frac{dx(t)}{dt}$ is the rate of change of the number of items in the system with respect to time.

Given the fluid flow Equation (4.8), Newell formulated the deterministic Fluid approximation for a single server queue as follows (Newell, 1971):

$$\frac{dE[L_i(t)]}{dt} = \begin{cases} \lambda_i - \mu & E[L_i(t)] > 0 \\ \max\left(0, \lambda_i - \mu\right) & E[L_i(t)] = 0 \end{cases} \qquad \forall i, t_{i-1} \leq t < t_i, \tag{4.9}$$

where $\frac{dE[L_i(t)]}{dt}$ is the rate of change of the number of items in the system with respect to time in period i. It shows clearly that in underloaded periods, the rate of change becomes negative. This results in only serving the delayed jobs in those periods. After serving all of them, the expected number of items in the system becomes zero. The differential Equation (4.9) needs to be solved numerically to approximate the expected number of items at each instant of time.

The PSFFA approach combines the deterministic Fluid approximation with the steady-state expected utilization formula to integrate random variation (Wang et al., 1996). Concerning Equation (4.8), when the queueing system has infinite queue length capacity, the inflows are only shaped by arrivals with a rate of λ_i. The outflow function is expressed through the approximated expected utilization at point time t (*i.e.*, $E[U_i(t)]$) multiplied by the service rate μ. The general idea is to determine the values for $E[U_i(t)]$ at particular instants of time by a point-wise mapping from the current value of $E[L_i(t)]$ into $E[U_i(t)]$ using the steady state queue. Explicitly, the expected utilization is approximated by the inverse of stationary queueing formulas such that the expected utilization becomes a function of the expected number of jobs in the system. Hence, the PSFFA approach models the fluid flow equation for an $M(t)/M/1$ queue as follows:

$$\frac{dE[L_i(t)]}{dt} = \lambda_i - \mu E[U_i(t)] = \lambda_i - \mu \frac{E[L_i(t)]}{E[L_i(t)] + 1} \qquad \forall i, t_{i-1} \leq t < t_i \tag{4.10}$$

To approximate the expected number of jobs in the system at each point in time, Equation (4.10) has to be solved numerically.

Comparing Equations (4.9) and (4.10), we simply identify that the difference between the fluid flow equations formulated by the Fluid approximation and the PSFFA approach is the inclusion of the expected utilization function. More precisely, it can be expressed by the function $\frac{E[L_i(t)]}{E[L_i(t)] + \omega}$. The PSFFA approach uses the parameter $\omega = 1$, and in the Fluid approximation, $\omega = 0$ is applied.

The PSFFA approach reaches the steady-state too rapidly. This leads to an overestimation of peaks and underestimation of valleys for quickly varying input rates. Accordingly, in overload situations, it fails to exactly approximate the expected utilization. The approximated utilization is underestimated and it results in the overestimation in approximating the expected number of items in the system (see Equation (4.10)). On the other hand, the Fluid approximation underestimates the approximated performance, because it does not capture the randomness. See Mandelbaum et al. (1998); Altman et al. (2001); Jiménez and Koole (2004) for more detailed explanations.

We believe that different values of ω can result in better approximation in overload situations. Therefore, we propose a modified PSFFA approach with different values of ω to be used in overload periods.

4.3.3. The hybrid approximation method

In this subsection, we comprehensively explain the proposed hybrid approximation method. Generally speaking, the hybrid approximation method combines the SBC approach with the modified PSFFA approach to approximate the performance measure of the queues where overloading occurs. The SBC approach is used in underloaded periods and the modified PSFFA approach is used to approximate the performance measure when the period is overloaded. A mechanism for adjusting the SBC and the modified approaches is designed when a transition from underload to overload (or the other way around) happens.

The underloaded period i is further divided into non-overlapping intervals ($j \in J$) with equal sizes ($l = \mu^{-1}$). Equations from (4.1) to (4.4) are applied to approximate the expected utilization in the j^{th} interval of period i. The value of the modified arrival rate ($\lambda_{i,j}^{MAR}$) is calculated by Equation (4.5). The proposed hybrid approximation method uses the Equations (4.6) and (4.7) to approximate the expected number of items in the system and the expected waiting time at the end of interval j in period i, respectively.

At the beginning of each period, three scenarios may happen. For period $i = 1$, the initial amount of the backlog rate at the beginning of this period ($b_{1,0}$) equals zero. If period $i > 1$ and period $i - 1$ are both underloaded, these artificially blocked jobs are carried over from the last interval of the previous period, *i.e.*, $b_{i,0} = b_{i-1,|J|}$. Finally, if $i > 1$ is an underloaded period and $i - 1$ is an overloaded period, an adjustment to backlog rate is needed. The initial backlog rate ($b_{i,0}$) is modified such that the expected number of items in the end of the previous period is resulted. Considering Equations (4.1) to (4.4) for each j, one can find a closed-form solution for the expected number of items in the system represented by artificial arrival rate and the service rate as:

$$\widetilde{\lambda_{i,j}} = E[L_{i,j}]\mu \qquad \forall i, j \tag{4.11}$$

Therefore, the backlog rate which is carried over into the first interval of period i (*i.e.*, $j = 1$) is calculated by:

$$b_{i,0} = E[L_{i-1}]\mu - \lambda_i \qquad \forall i > 1 | \frac{\lambda_i}{\mu} < 1 \& \frac{\lambda_{i-1}}{\mu} \geq 1, \tag{4.12}$$

where $E[L_{i-1}]$ is the expected number of items at the end of period $i-1$ and is calculated by the modified PSFFA approach, which will be explained next.

Let t be a point of time in period i (*i.e.*, $t_{i-1} \leq t < t_i$). If period i is overloaded, then the approximation method uses the modified PSFFA to approximate the performance measures. The approximation method models the fluid flow equation as follows:

$$\frac{dE[L_i(t)]}{dt} = \lambda_i - \mu \frac{E[L_i(t)]}{E[L_i(t)] + \omega} \qquad \forall i, t_{i-1} \leq t < t_i \tag{4.13}$$

The Euler method is applied to solve the differential Equation (4.13). The Euler method divides period i into very small time steps with the length of Δt. The initial value to plug in the Euler method is given as the expected number of items at the end of previous period $(i-1)$ (*i.e.*, $E[L_{i-1}]$). More precisely, in the first interval $\left[t_{i-1}, t_{i-1} + \Delta t\right)$, the expected number of items in the system at the end of the time interval is as follows: $E[L_i(t_{i-1} + \Delta t)] = E[L_{i-1}] + (\lambda_i - \mu \frac{E[L_{i-1}]}{E[L_{i-1}]+\omega})\Delta t$. The amount of $E[L_i(t_{i-1} + \Delta t)]$ then becomes the initial value for the next time step, *i.e.*, $[t_{i-1} + \Delta t, t_{i-1} + 2\Delta t]$, and then $E[L_i(t_{i-1} + 2\Delta t)] = E[L_i(t_{i-1} + \Delta t)] + (\lambda_i - \mu \frac{E[L_i(t_{i-1}+\Delta t)]}{E[L_i(t_{i-1}+\Delta t)]+\omega})\Delta t$. This procedure is repeated for each time step (Δt) in period i.

For period $i = 1$, the initial value to plug in the Euler method is a small value of ϵ. If there is a transition from an underloaded period to an overloaded one, the initial value to use in the Euler method is the expected number of items calculated by the SBC approach in the last interval of the previous period as $E[L_{i-1,|J|}]$.

To summarize, the pseudocode of the proposed hybrid approximation method is given by Algorithm 3, see the source codes in Appendix D. First, the parameters of the algorithm are initialized (line 3). The algorithm iterates over periods (line 4). If period i is underloaded (line 5), the hybrid approximation method uses the SBC approach to approximate the expected work-in-process and the expected waiting time (lines 6 to 19). If there is a transition from an overloaded period to an underloaded one, to (re-)start the SBC approach, the backlog rate is derived from the expected number of items in the previous items (line 7). If period i is overloaded (line 21), the modified PSFFA approach is used by the proposed hybrid approximation method (lines 22 to 30). The Euler method solves the differential Equation (4.13) (lines 25 to 29). If there is a transition from an underloaded period to an overloaded one, the expected number of items in the last interval of the previous period is used as the initial value to plug in the Euler method to solve the differential Equation (4.13) (line 23). The outcome of the algorithm is the approximated expected work-in-process and the approximated expected waiting time at the end of each period. Additionally, all the notation used in this section are collected in Table 4.1.

Algorithm 3 Pseudocode Overview of proposed hybrid approximation method

```
 1: procedure HYBRID APPROXIMATION METHOD
 2:     Input: $\lambda_i, \mu, \omega$
 3:     Initialize $b_{1,0}, \epsilon, \Delta t$
 4:     for each period $i$ do
 5:         if $\frac{\lambda_i}{\mu} < 1$ then
 6:             if from overloaded to underloaded period then
 7:                 adjust $b_{i,0}$                                        ▷ Equation (4.12)
 8:             for each interval $j$ do
 9:                 $\widetilde{\lambda_{i,j}} = \lambda_i + b_{i,j-1}$
10:                 procedure $M/M/1/1(\widetilde{\lambda_{i,j}}, \mu)$          ▷ Step 1 of the SBC approach
11:                     return $P_{i,j}^{BL}, E[U_{i,j}]$
12:                 $b_{i,j} = P_{i,j}^{BL}\widetilde{\lambda_{i,j}}$
13:                 $\lambda_{i,j}^{MAR} = E[U_{i,j}]\mu$
14:                 procedure $M/M/1/\infty(\lambda_{i,j}^{MAR}, \mu)$          ▷ Step 2 of the SBC approach
15:                     return $E[L_{i,j}], E[W_{i,j}^{Q}]$
16:         if $\frac{\lambda_i}{\mu} \geq 1$ then
17:             if from underloaded to overloaded period then
18:                 adjust $E[L_i(t_{i-1})]$                       ▷ $E[L_i(t_{i-1})] := E[L_{i-1,|J|}]$
19:             $t := t_{i-1}$
20:             while $t < t_i$ do
21:                 $E[L_i(t+\Delta t)] = E[L_i(t)] + (\lambda_i - \mu\frac{E[L_i(t)]}{E[L_i(t)]+\omega})\Delta t$     ▷ Euler method
22:                 $t := t + \Delta t$
23:     return $E[L_i(t)]$
```

Table 4.1.: The summary of the notation used for the system description and proposed hybrid approximation method

Notation	Description
t	($t \in [0, T]$) time
i	($i \in \{1, \cdots, I\}$) the index of periods
j	($j \in J = \{1, \cdots, \lceil \mu \rceil\}$) the index of intervals
d	Period length
l	Interval length (in the SBC approach)
λ_i	Arrival rate in period i
μ	Service rate
$b_{i,j}$	Backlog rate for arrivals in period i interval j
$P_{i,j}^{BL}$	Probability of items being backlogged in period i interval j
$\widetilde{\lambda_{i,j}}$	Artificial arrival rate in period i interval j
$\lambda_{i,j}^{MAR}$	Modified arrival rate in period i interval j
$E[U_{i,j}]$	Expected utilization in period i interval j
$E[L_{i,j}]$	Expected number of items in period i interval j
$E[W_{i,j}^{Q}]$	Expected waiting time in period i interval j
$E[U_i(t)]$	Expected utilization in period i at time t ($t_{i-1} \leq t < t_i$)
$E[L_i(t)]$	Expected number of items in period i at time t ($t_{i-1} \leq t < t_i$)
$x(t)$	Ensemble expected number of items in the system in period i at time t ($t_{i-1} \leq t < t_i$)
ϵ	Small pre-defined value to plug in the Euler approach
Δt	Time step length in the Euler approach
ω	Pre-defined parameter used in modified PSFFA approach

4.4. Numerical study

The numerical studies are explained in this section. Subsection 4.4.1 studies the impacts of parameter ω used in the proposed hybrid approximation method in $M(t)/M/1$ queues. Subsection 4.4.2 evaluates the quality of the proposed hybrid approximation method in comparison with simulation and three performance approximation approaches from the literature.

The proposed hybrid approximation method is coded in Java and all the experiments are performed on an Intel core i5-6300U CPU.

4.4.1. The effects of parameter ω in time-dependent queues

In this subsection, we examine the performance of the proposed hybrid approximation method for different values of parameter ω. In the modified PSFFA approach, ω plays a crucial role. If $\omega = 0$, it underestimates the expected performance measure and if it equals one (*i.e.*, $\omega = 1$), overestimation happens, see Section 4.3.2. We compare the performance of the proposed hybrid approximation method when $\omega \in \{0.1, 0.3, 0.7\}$. In addition to this, all the numerical studies to analyze the transient behavior of the proposed hybrid approximation method and the effects of parameter ω on its performance are presented in Appendix C.

To that end, instances with sinusoidal arrival rates are generated. Sinusoidal arrival rates are commonly used in the literature of the performance approximation approaches for time-dependent queues, for example see Green et al. (1991); Eick et al. (1993); Green and Kolesar (1998); Whitt (2014). One of the main features of queueing systems with such arrival rates is their ongoing changes. The queueing system with such arrival rates can alternate between underloaded and overloaded periods. We examine two sinusoidal arrival functions for an $M(t)/M/1$ queueing system with the service rate of one per time unit. The first function is $\lambda_t = \sin(0.2(t + 20)) + 1.1$. The system starts with underloaded periods and transfers to overload after several periods. The second arrival function is $\lambda_t = \sin(0.2(t + 5)) + 1.1$, in which the system starts in overload and it later shifts to underloaded periods. They both are transferred to a piecewise arrival rate for one time unit by taking the value of the sinus equations at the beginning of each interval. Figures 4.1 and 4.2 show the arrival rates and the service rate of these two instances.

We use another instance, where an $M(t)/M/1$ queueing system is modeled with 15 periods ($i \in \{1, \cdots, 15\}$) with a length of one time unit and the average service time of $\mu^{-1} = 3$ time unit. The arrival rates are adjusted such that the system has a traffic

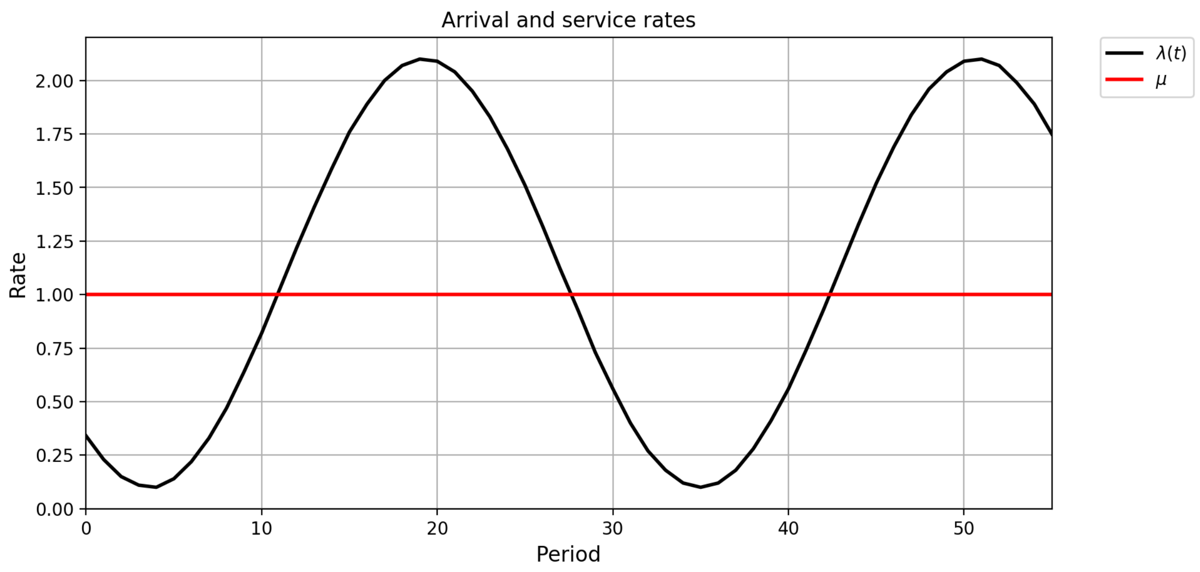


Figure 4.1.: The arrival rate and the service rate of instance with sinusoidal arrival rate with function $\lambda_t = \sin(0.2(t+20)) + 1.1$

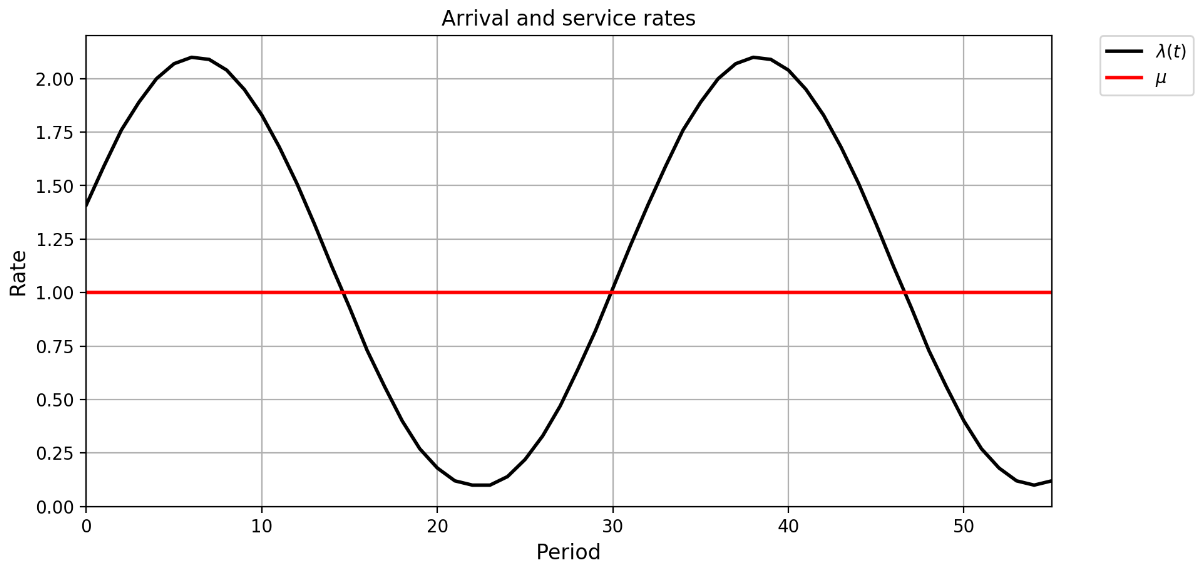


Figure 4.2.: The arrival rate and the service rate of instance with sinusoidal arrival rate with function $\lambda_t = \sin(0.2(t+5)) + 1.1$

intensity of 0.5 in the first five periods. The peak occurs in the sixth period with a traffic intensity of 1.2 and lasts until the end of period 10. Finally, the system comes back to underload with the traffic intensity of 0.8. The arrival rates and the service rate of this instance are depicted in Figure 4.3.

Simulation is applied as a benchmark. Figures 4.4 to 4.6 depict the value of the excepted work-in-process from the proposed hybrid approximation method with different values of ω.

The proposed hybrid approximation method with $\omega = 0.3$ works very well when the system starts from an underloaded period. In Figure 4.4, when the system transfers into the overloaded period, the hybrid approximation method with $\omega = 0.7$ always overestimates the expected value of the work-in-process. $\omega = 0.1$ always results in an underestimation of the proposed hybrid approximation method for the expected work-in-process.

In Figure 4.5, the hybrid approximation method with $\omega = 0.1$ is better than others in the first half of the planning horizon. The hybrid approximation method with $\omega = 0.7$ is the worst in comparison with others. The proposed hybrid approximation method with $\omega = 0.3$ overestimates the performance measure up to period 30. However, after the transition into an underloaded period, the approximation errors of the proposed hybrid approximation method reduce and it outperforms others. In Figure 4.6, $\omega = 0.3$ results the best in comparison with others. The high value of the expected number of jobs in the system when the transition to an overloaded period happens causes the differences to disappear after a small amount of time.

In summary, the proposed hybrid approximation method performs very well, given that the “right” value of ω is used. Based on the results presented in this subsection and in Appendix C, the value is recommended to be $\omega = 0.3$, because it increases the accuracy of the proposed hybrid approximation method.

4.4.2. Evaluation of the approximation method

As mentioned in previous sections, our proposed hybrid approximation method combines two approximation methods, namely, the SBC approach and the modified PSFFA approach. Each of these approximation methods performs well in different situations. In this subsection, we benchmark our approach against the SBC and the PSFFA approaches in isolation. The Fluid approximation is also selected because of its high performance when the system is in overload. Simulation results act as benchmarks for comparison. The first instance with the Sinusoidal arrival rates introduced in the previ-

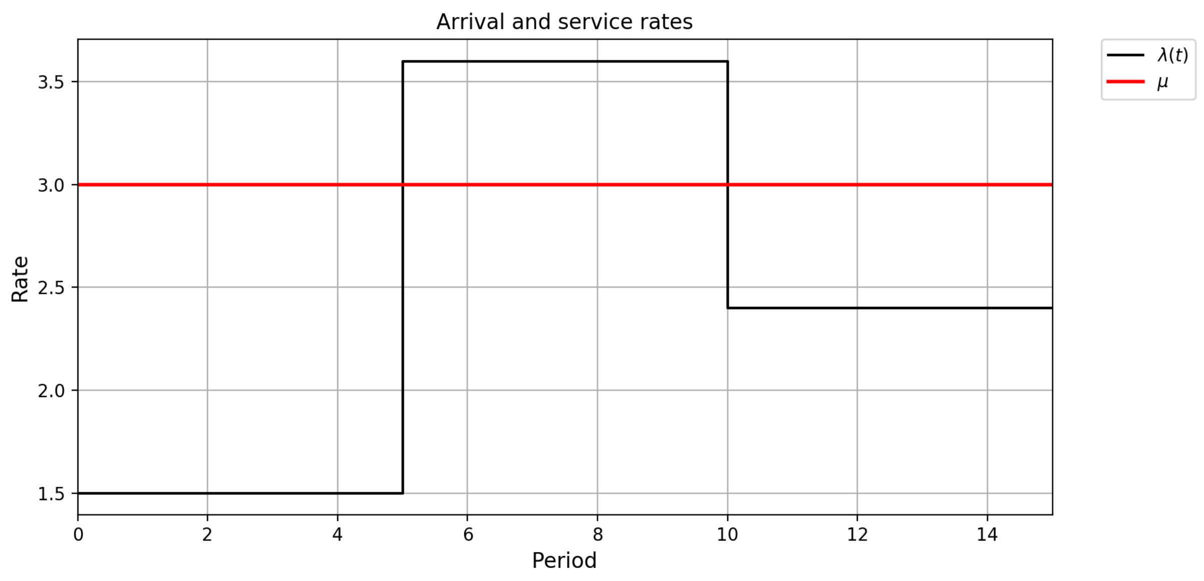


Figure 4.3.: The arrival rate and the service rate of instance with one peak

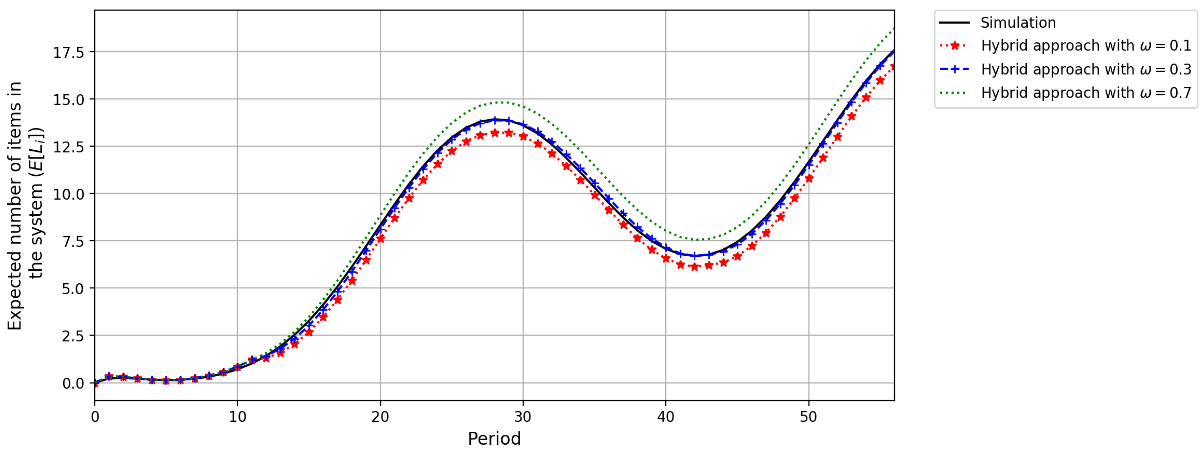


Figure 4.4.: The effects of ω on the quality of the approximation method for instance with sinusoidal arrival rate with function $\lambda_t = \sin(0.2(t + 20)) + 1.1$

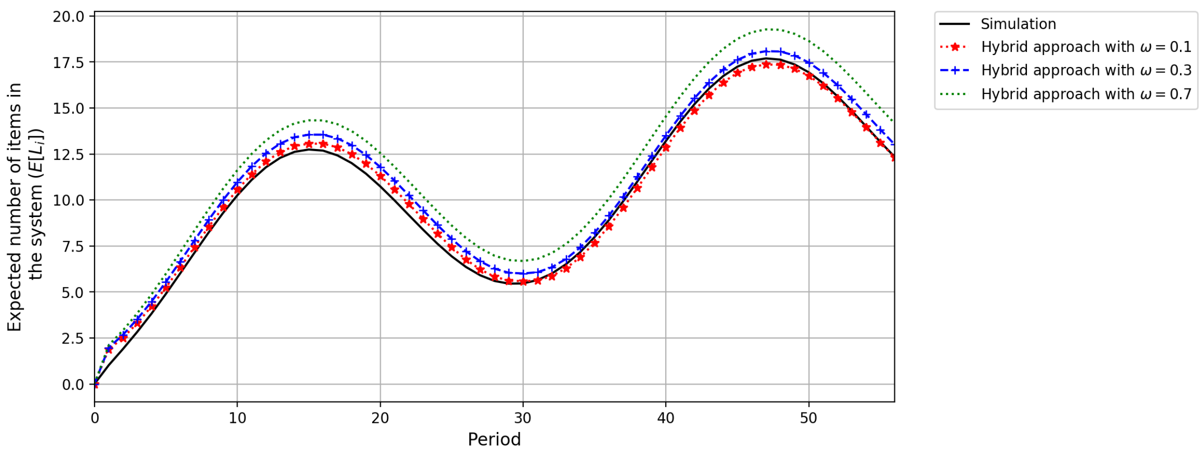


Figure 4.5.: The effects of ω on the quality of the approximation method for instance with sinusoidal arrival rate with function $\lambda_t = \sin(0.2(t + 5)) + 1.1$

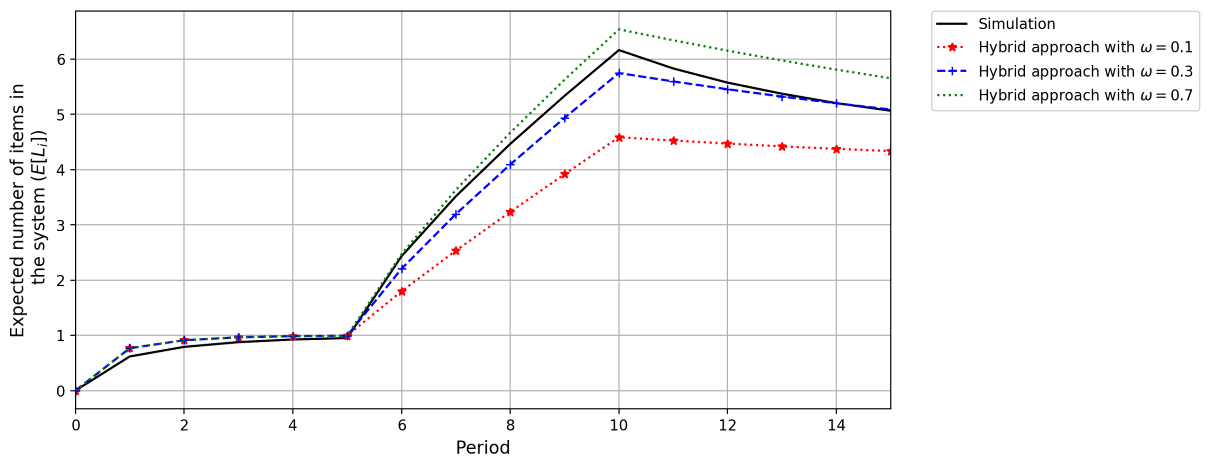


Figure 4.6.: The effects of ω on the quality of the approximation method for instance with one peak

ous subsection (Figure 4.1) is chosen. Figures 4.7 and 4.8 illustrate the results for the expected waiting time and the expected number of items in the system, respectively.

The performance of the approximation method is well in comparison with simulation. It also outperforms the SBC, the PSFFA, and the Fluid approximation approaches in isolation. The SBC and the PSFFA approaches overestimate the performance measure of the overloaded periods. This causes further approximation errors to occur when the system transfers back to the underloaded periods. The rationale for that is in the SBC approach, the artificial arrival rates grow very fast in overloaded periods and it causes overestimation. In the PSFFA approach, the expected utilization is not approximated well when parameter ω equals one, also see 4.4.1 and Appendix C. The Fluid approximation method underestimates the performance measure when the system transfers into an underloaded period. In general, the Fluid approximation acts as a lower bound for the performance measure, also see Jiménez and Koole (2004). Further investigations are depicted in Figures 4.9 and 4.10. We show the absolute value of the deviation from simulation and all approximation methods in each period by a box-plot diagram. The proposed hybrid approximation method outperforms others.

In the next experiments, we analyze systems introduced in the literature for time-dependent queues with overload situations. We generate the instances by keeping the shape of the arrival rates the same as they are introduced, and we adjust the service rates.

The first instance is derived from the study of Jiménez and Koole (2004). They analyze a customer service system (call center) having exponential service time and a time-dependent exponential inter-arrival time. In the first 30 minutes, the arrival rate is 11.03 per minute. It becomes 7.43 per minute for the second and 5.4 per minute for the last 30 minutes. The system has the traffic intensity of $\rho = 1.37$ for the first, $\rho = 0.92$ for the

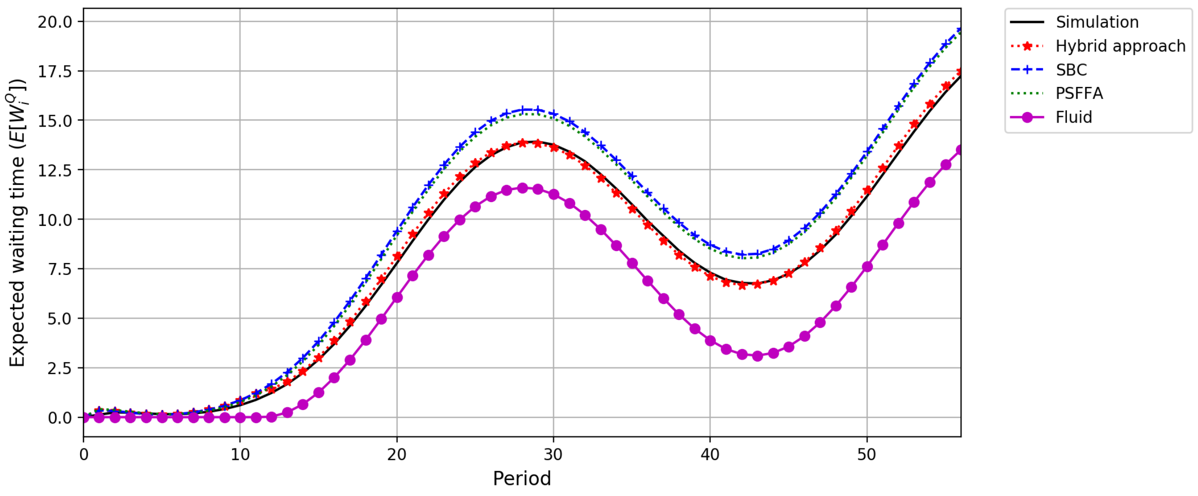


Figure 4.7.: The comparison of the proposed hybrid approximation method with simulation, SBC, PSFFA, and Fluid for the expected waiting time

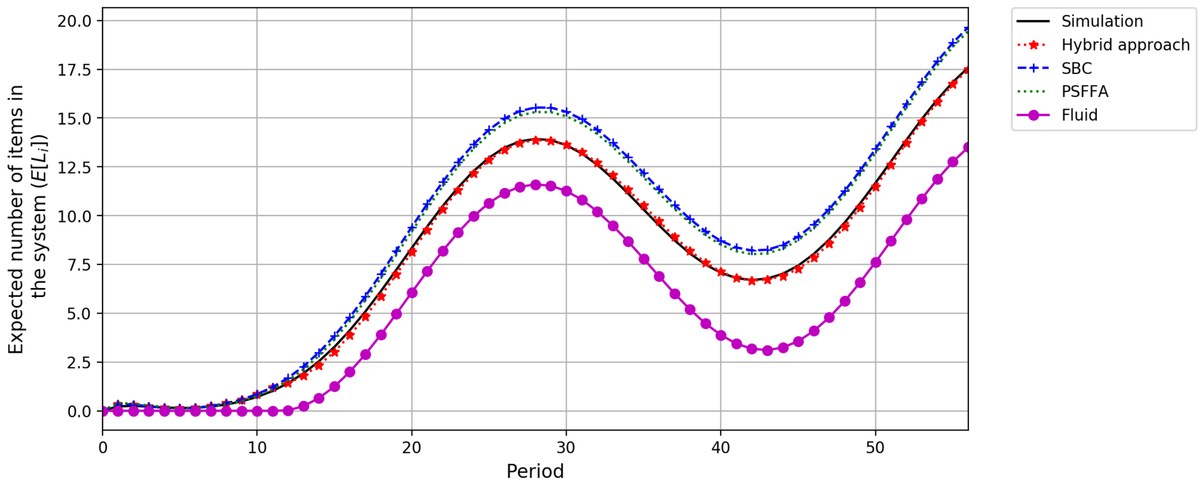


Figure 4.8.: The comparison of the proposed hybrid approximation method with simulation, SBC, PSFFA, and Fluid for the expected number of items in the system

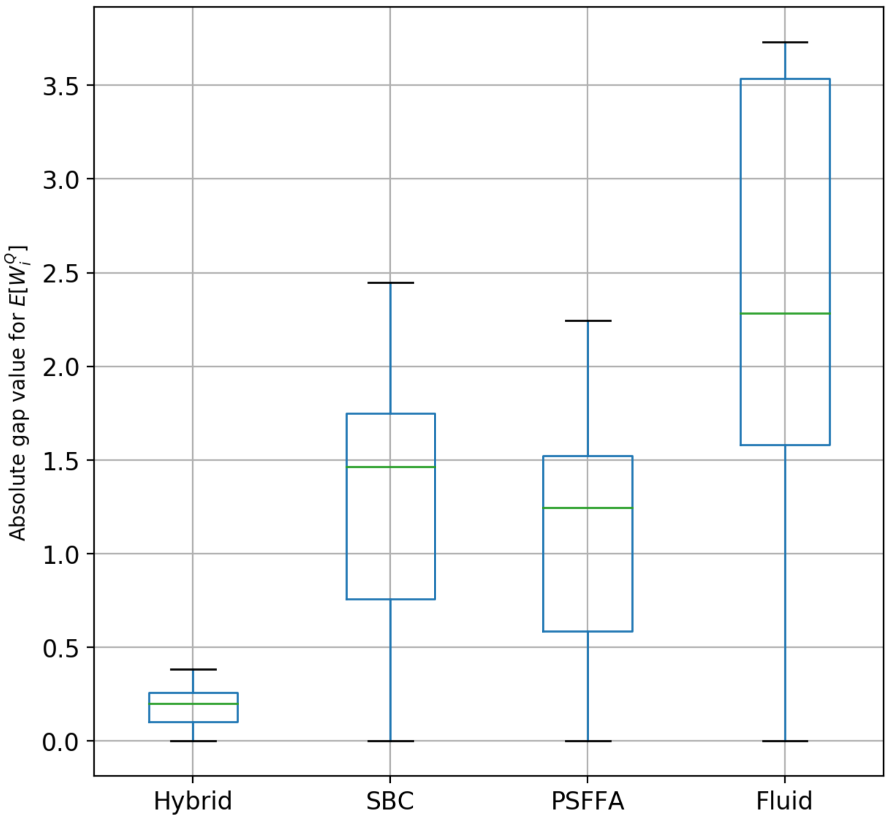


Figure 4.9.: The box-plot of the absolute values of deviation from simulation for the expected waiting time

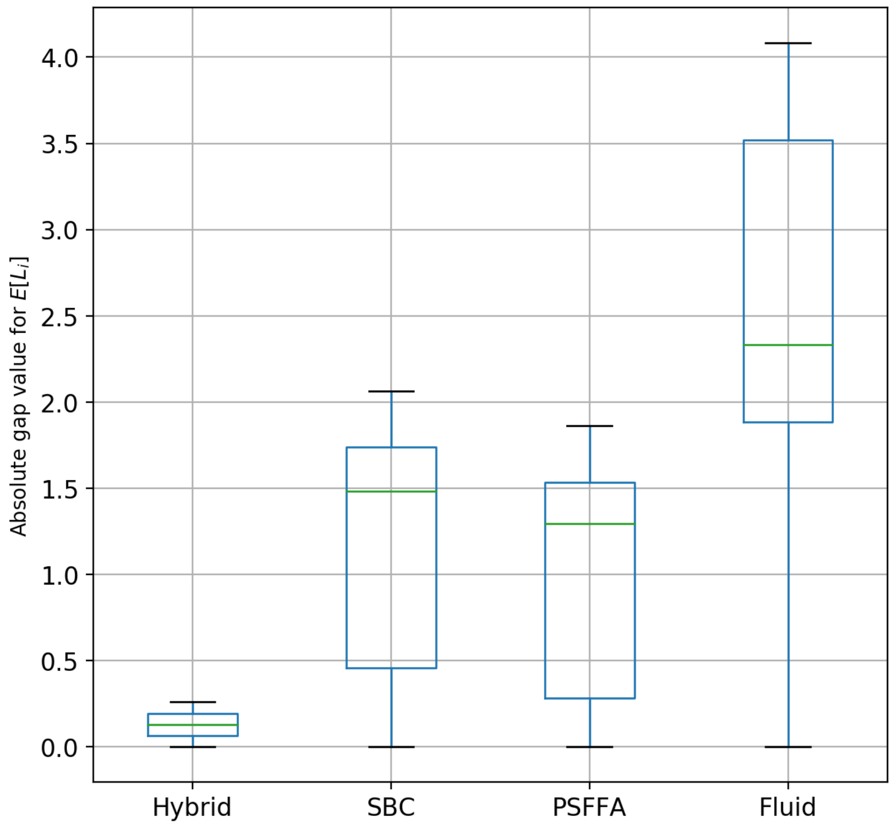


Figure 4.10.: The box-plot of the absolute values of deviation from simulation for the expected number of items

second, and $\rho = 0.67$ for the last 30 minutes. The system starts in an overload situation and it transfers to the underloaded periods after 30 minutes. We create an instance out of it. The instance consists of a single server queue with exponentially distributed service time with a rate of eight per minute ($\mu = 8\frac{1}{min}$) and the same arrival rates. Therefore, the traffic intensity of the system is the same as the introduced instance. The rate of arrivals as well as service are shown in Figure 4.11.

The results of the approximation method, simulation, the SBC, the PSFFA, and the Fluid approaches for the expected waiting time and the expected work-in-process are illustrated in Figures 4.12 and 4.13 for modified instances from Jiménez and Koole (2004).

The second instance is derived from Stolletz (2008a). We create an instance as an $M(t)/M/1$ queue such that the shape of the arrival rates is kept the same as it is introduced in Stolletz (2008a). The service rate is chosen to be five (*i.e.*, $\mu = 5\frac{1}{min}$). It results in overload situations occurring in the system. The system starts from the underloaded period. In the eighth period ($i = 8$), there is a "jump" from an underloaded period to an overloaded one. The system is overloaded at several periods with a maximum traffic intensity of 1.64. From periods $i = 25$ to 28, the load of the system is 0.95; ultimately, it reduces more. The shape of the arrival rates, service rate, and the evaluated performance of the system by the proposed hybrid approximation method, simulation, the SBC, the PSFFA, and the Fluid approaches are depicted in Figures 4.14 to 4.16.

The proposed hybrid approximation method performs very well in comparison with simulation. It also outperforms the SBC, the PSFFA, and the Fluid approximation approaches. It is demonstrated that in all of the instances when there is a shift from an overloaded period to an underloaded one, the proposed hybrid approximation method overestimates it. However, this approximation error can be neglected, as they are low in value.

In summary, the quality of the proposed hybrid approximation method for the performance evaluation of time-dependent queues is high. It can be used as a substitute for simulation since it is accurate and it is fast because it is the combination of the SBC approach, PSFFA, and a calibration procedure.

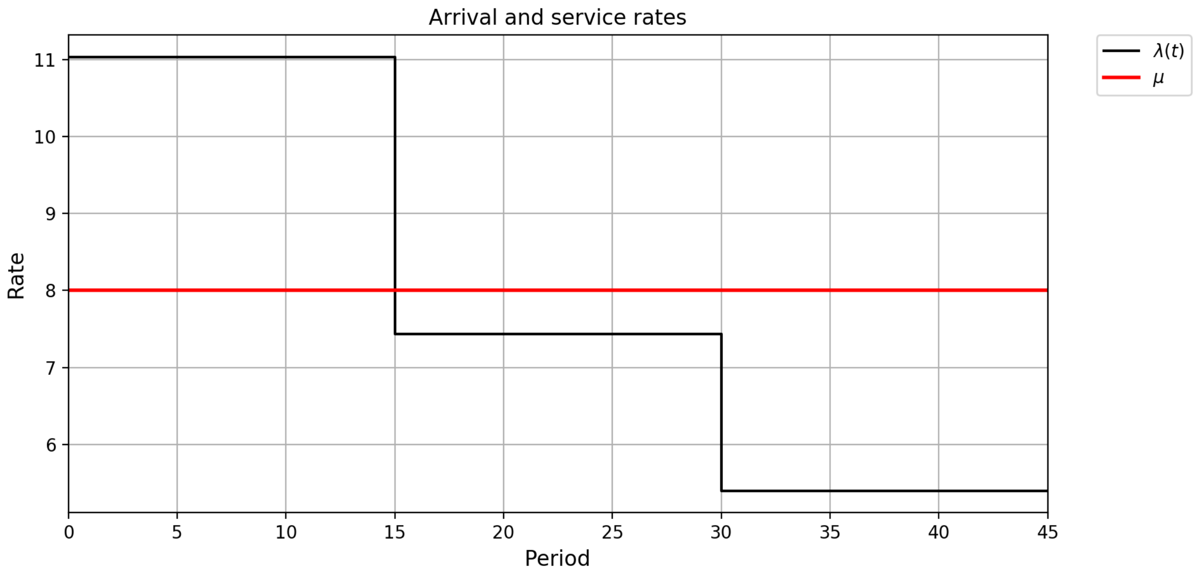


Figure 4.11.: The rate of arrivals and the service derived from Jiménez and Koole (2004)

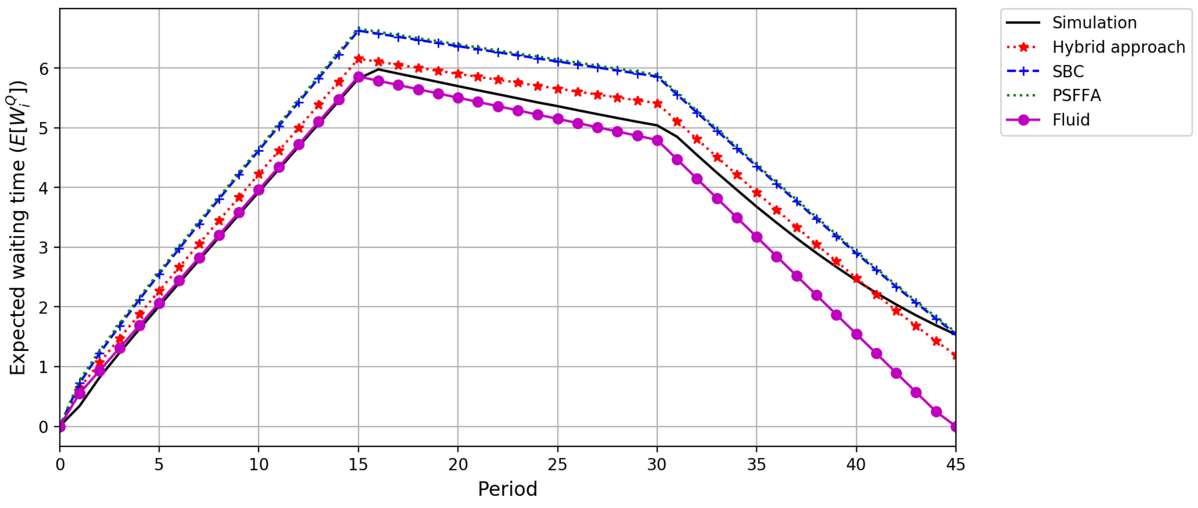


Figure 4.12.: The comparison of the proposed hybrid approximation method with simulation, SBC, PSFFA, and Fluid over modified benchmark from Jiménez and Koole (2004) for expected waiting time

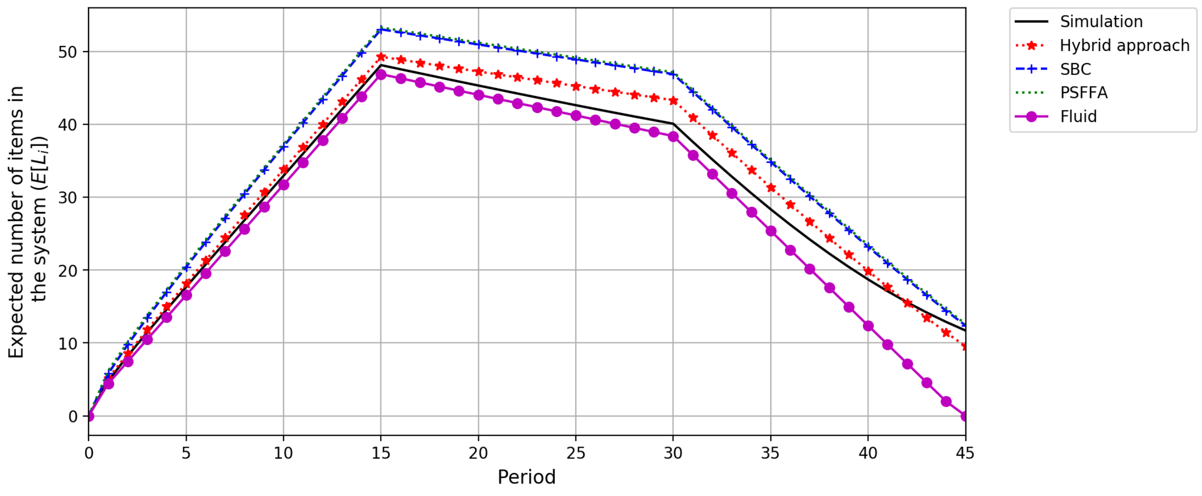


Figure 4.13.: The comparison of the proposed hybrid approximation method with simulation, SBC, PSFFA, and Fluid over modified benchmark from Jiménez and Koole (2004) for the expected number of items in the system

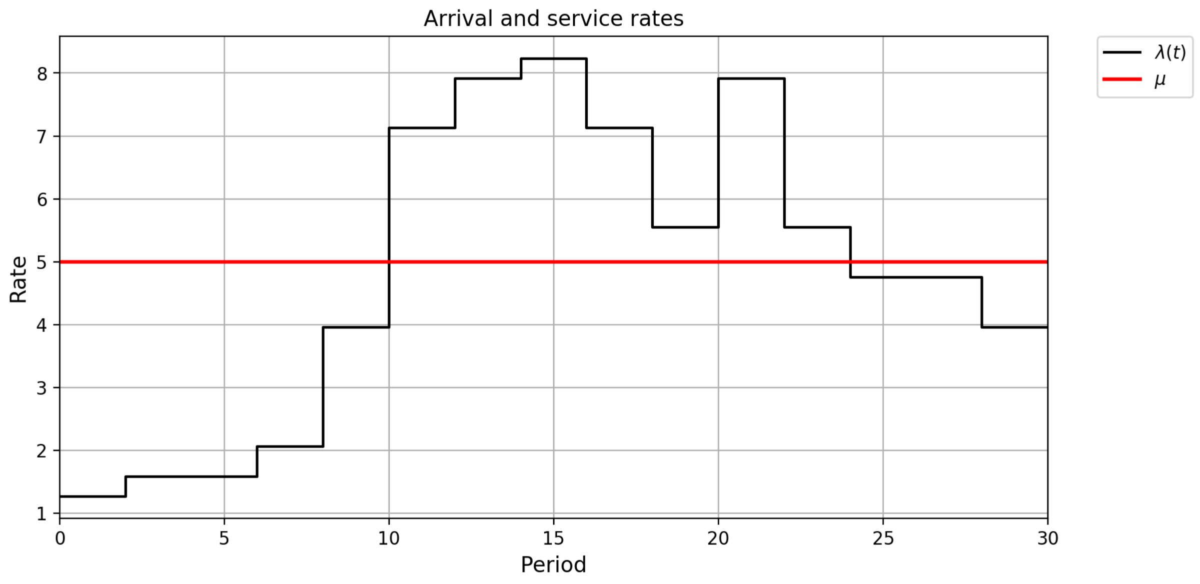


Figure 4.14.: The rate of arrivals and the service derived from Stolletz (2008a)

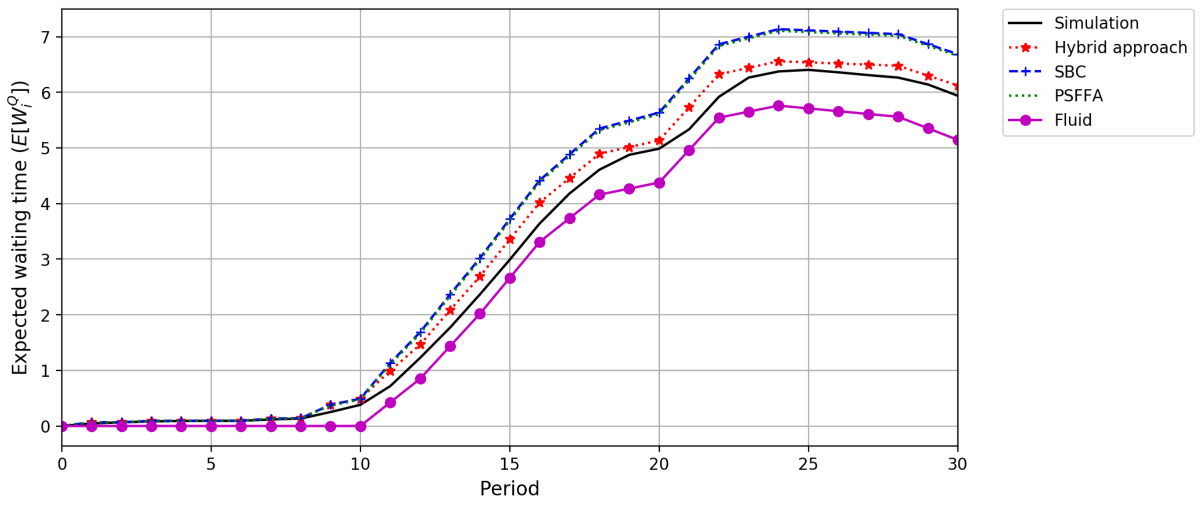


Figure 4.15.: The comparison of the proposed hybrid approximation method with simulation, SBC, PSFFA, and Fluid over modified benchmark from Stolletz (2008a) for expected waiting time

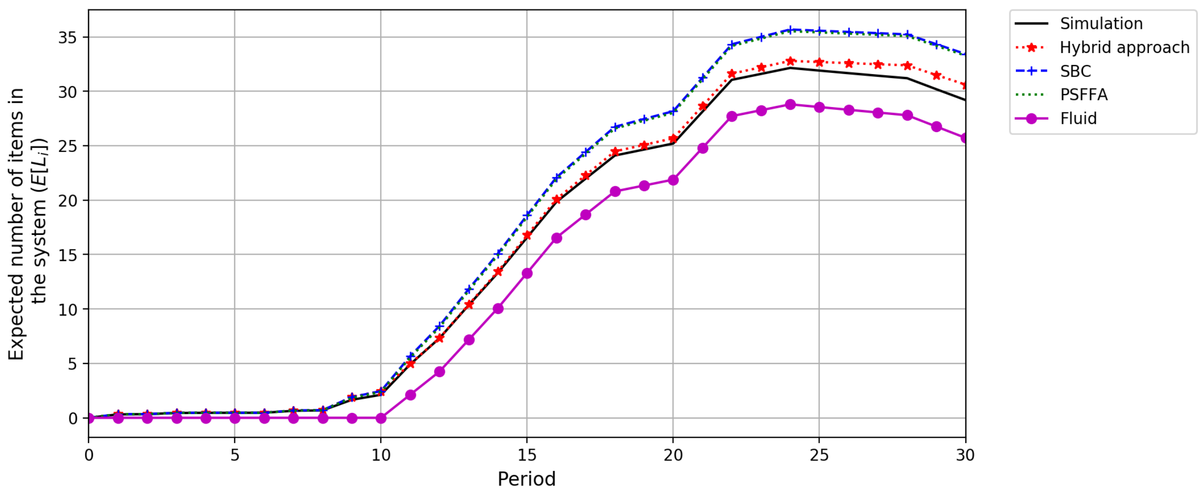


Figure 4.16.: The comparison of the proposed hybrid approximation method with simulation, SBC, PSFFA, and Fluid over modified benchmark from Stolletz (2008a) for the expected number of items in the system

4.5. Conclusion and further research

Many real-world systems can be represented by time-dependent queues. In such systems, the rate of arrivals may surpass the service capacity for some time periods, which causes the system to be in overload. In this paper, we analyze a single server time-dependent queue with exponentially distributed inter-arrival and service time. The hybrid approximation method combines the SBC approach with the PSFFA approach to approximate the performance measure of the queues where overloading occurs. Some modifications are made to the PSFFA approach within the proposed hybrid method. The SBC approach is used in underloaded periods and the modified PSFFA approach is used to approximate the performance measure when the period is overloaded. A mechanism for adjusting the parameters of the SBC and the modified approaches is designed when a transition from underload to overload (or the other way around) happens.

The effects of the proposed hybrid approximation method's parameters are numerically examined. Numerical experiments show that the quality of the approximation method in comparison with simulation is very high. In addition, it outperforms the SBC approach, the PSFFA, and the Fluid approximation approaches in isolation for different time-dependent queueing instances.

We believe the proposed approximation method is flexible enough to be adjusted for the evaluation of different time-dependent queues, specifically for those queueing systems where the SBC approach and the Fluid approximation are employed successfully. For example, considering the time-dependent multiple servers cannot dramatically change the proposed algorithm. The only measure to perform is to modify the SBC- and Fluid approximation-related equations from a single server to multiple servers. Likewise, queueing systems with general distributions for service and inter-arrival times can easily be approximated by the proposed approximation method, only if the performance evaluation approaches related to general distribution are plugged into the SBC approach, and the expected utilization function of the modified PSFFA has changed accordingly. These ideas are suitable to study more in the future. In addition, applying the presented idea to the other queueing systems such as queueing systems with heterogeneous arrivals, those with a retrial, those with limited waiting room capacity, and those with batch arrivals is left for future research.

The proposed hybrid approximation method also can be used for decision support since it is fast and accurate. It can be integrated into optimization approaches, where a solution is evaluated. In addition, in simulation-optimization approaches, simulation can be substituted for the proposed hybrid approximation method. Further research has to be conducted on applying different period length adjustments in $M(t)/M/1$ to improve the quality of the approximation method, specifically when it transfers from an overloaded system to an underloaded one.

5. Conclusions and outlooks

5.1. Conclusion

This dissertation proposes several algorithmic methods to reinforce operations management decision-making. It includes three papers that design and analyze analytical methods to address practical applications in operations management.

In the first paper, we investigate the picker routing problem and the interrelated product-location problem as they arise in warehouses with mobile racks designed for space efficiency. We present dynamic programming, beam search algorithm, and set-covering-based heuristics to solve the picker routing problem. We evaluate two solution approaches and compare the product-location problem's shared and fixed storage space allocation performance. Our computational tests reveal that the proposed heuristics for the picker routing problem perform pretty well, although there is a distinct tradeoff regarding solution time and quality. The dynamic programming scheme cannot reasonably solve larger instances, whereas beam search and the set-covering heuristic have proven quite adept. Beam search, however, while delivering good results, may take a long time to reach that solution in many instances. The set-covering heuristic is very fast but exhibits non-negligible gaps regarding the best-known solutions. The managerial insights obtained are as follows: 1- Chaotic storage, i.e., assigning random locations to SKUs, delivers suboptimal results and leads to unnecessarily long pick times. With the proposed priority rule, significantly better results are achievable. 2- Fixed storage, i.e., assigning a single storage location to each SKU, is not advisable in a mobile rack warehouse. It leads to a much greater number of rack movements, which should be avoided due to the low movement speed. 3- The priority rule suffices to know the distribution of future SKU demands. Significant savings compared to chaotic storage are possible.

The second paper analyzes the outpatient scheduling problem with priority rules. The priority rules control the access of patients waiting to be served. We design a simulation optimization approach with a tabu search to study the effects of the priority rules on the schedule of outpatients. The proposed algorithm uses a neighborhood reduction technique in which low-fidelity simulation is applied to speed up searching for the optimal solution. Our computational tests reveal the effectiveness and efficiency of the proposed algorithm to solve the problems by comparing the solutions obtained from a full enumeration procedure. The numerical results indicate that when decision-makers attempt to optimize the timely schedules of outpatients, they must consider what priority rules the system uses. Specifically, when the system experiences a high load, the optimal schedule of outpatients differs depending on the priority rule used. Furthermore, the optimal schedule is more likely to differ for the higher load triggered by inpatients. Additionally, the results demonstrate that if the threshold priority rule is used, the outpatients are scheduled near the end of the planning horizon, resulting in lower outpatient waiting costs, even while incurring further overtime costs.

The last paper considers a time-dependent single server queueing system in which the rate of arrivals surpasses the service capacity for some periods, resulting in the system being overloaded. The hybrid approximation method combines the SBC approach with the modified PSFFA approach to approximate the performance measure of the queues where overloading occurs. Numerical experiments show that the quality of the approximation method in comparison with simulation is very high. It outperforms the SBC approach, the PSFFA, and the Fluid approximation approaches in isolation for different time-dependent queueing instances. The proposed hybrid approximation method can be used for decision support since it is fast and accurate. It also can be integrated into optimization approaches, where a solution is evaluated.

5.2. Further research directions

In this subsection, we summarize the suggestions for further research in each chapter and, as a whole, give several new directions for research.

In Chapter 2, for the proposed high-density storage setting, we assume the aisles are approachable only from one side. However, one can analyze other settings where the aisles are approachable from both sides, and thus different tour strategies (starting from and ending at a central depot), such as S-shape and midpoint, can be applied. From a practical standpoint, it is probably desirable to research the usage of mobile racks in refrigerated warehouses. It is interesting to investigate perishable items that require just-in-time operations. It can be embodied in future models by adding constraints (such as including the due date for storing goods) and objective functions (such as including penalties for deviation from the due date). In addition, scheduling the replenishment and retrievals activities is another direction of research in such a storage setting. Finally, a dynamic order picking setting in warehouses with mobile racks, where the information about the incoming orders is unknown at the beginning of the planning period, is another direction for research. As for the analytical approaches, an interesting topic for future research is incorporating neighborhood reduction techniques in the solution approaches. We already showed how to benefit from the inclusion of the neighborhood reduction techniques in a solution approach in Chapter 3. There is still room for investigating the exact solution approaches for the order picking in high-density storage with mobile racks. Considering the product-location problem, one can incorporate accurate statistical approaches and solve the problem precisely because the order sets are, in reality, stochastic. Machine learning models embodied in optimization tools and stochastic optimization approaches can be regarded as a good option in such settings.

In Chapter 3, considering other assumptions such as patient abandonment, patient's unpunctuality, and patient's preferences are exciting topics to further research. In reality, the arrival of emergency patients and inpatients are time-dependent. In addition, the service time distribution differs for patient types. Incorporating these practical assumptions is one direction for future research. In some hospitals, there is tension between serving inpatients and outpatients. Simultaneously scheduling inpatients and outpatients to resolve this tension, apart from prioritizing them, would be a potential research outlook. From the method standpoint, as appointment scheduling can be represented by time-dependent queueing systems, integrating the approximation methods for time-dependent queues into solution approaches would be an interesting topic to investigate. Machine learning models can be embodied to design more practical solution approaches. Various stochastic optimization techniques (such as 1- variance reduction techniques and 2- parallel computing techniques for optimization and evaluation) can be applied and compared to obtain analytical insights.

Finally, in Chapter 4, one can investigate and modify the proposed approximation method for other queueing systems such as queues with time-dependent number of servers, queues with abandonment and retrials, queues with generally-distributed rates, and queues with limited capacity. In addition, as explained above, the proposed approximation method for time-dependent queueing systems can be integrated into solution methods where the system experience overloading periods, such as the solution method to solve the proposed appointment scheduling. Analytically interpreting the parameter ω in the proposed approximation method can be another direction for research.

To summarize, we believe that our dissertation sheds light on the importance of algorithmic approaches for supporting decision-making in operations management. With the aid of analytical methods used to analyze practical applications in our dissertation, we have made some headway toward a better understanding of routing decisions for warehouses with mobile racks, timing decisions for hospitals, and approximation of time-dependent queues with overloading.

Appendices

A. Mathematical model (Chapter 2)

For the MIP model, let the set of all locations be denoted by $\mathcal{V}$. Furthermore, $\mathcal{B}$ is the set of orders and $\mathcal{S}$ is the set of all SKUs. Each shelf can be accessed from two sides, depending on which aisle is open, i.e., every location can be reached by the picker either from the left aisle or the right aisle. Let set $\mathcal{K} = \{\text{left}, \text{right}\}$ indicate the possible sides. In the following, the indices of the sets, the parameters, and variables are summarized.

Indices:

$i, j \in \mathcal{V}$	Locations with $i = 1$ as the depot
$b \in \mathcal{B}$	Orders
$s \in \mathcal{S}$	SKUs
$k, t \in \mathcal{K} = \{\text{left}, \text{right}\}$	Accessible sides

Parameters:

$t_{(i,k)(j,t)}$	The distance from location i side k to location j side t
p_b	Number of SKUs in order b
$q_{b,s}$	1 if order b includes SKU s; 0, otherwise

Variables:

$x^b_{(i,k)(j,t)}$	1 if order b is picked via the path from location i side k to location j side t
$\alpha_{b,s,i}$	1 if SKU s is in location i for order b
$u_{b,i}$	Auxiliary variables for subtour elimination

The optimization model is defined as follows.

$$\text{Min} \sum_{b \in \mathcal{B}} \sum_{i \in \mathcal{V}} \sum_{\substack{j \in \mathcal{V}: \\ j \neq i}} \sum_{k \in \mathcal{K}} \sum_{t \in \mathcal{K}} x^b_{(i,k)(j,t)} t_{(i,k)(j,t)} \tag{A.1}$$

s.t.

$$\sum_{k \in \mathcal{K}} \sum_{\substack{j \in \mathcal{V}: \\ j \neq i}} \sum_{t \in K} x^b_{(i,k)(j,t)} = \sum_{s \in \mathcal{S}} \alpha_{b,s,i} \qquad \forall i \in \mathcal{V}, b \in \mathcal{B}, i > 1 \tag{A.2}$$

$$\sum_{\substack{i \in \mathcal{V}: \\ j \neq i}} \sum_{k \in \mathcal{K}} \sum_{t \in \mathcal{K}} x^b_{(i,k)(j,t)} = \sum_{s \in \mathcal{S}} \alpha_{b,s,j} \qquad \forall j \in \mathcal{V}, b \in \mathcal{B}, j > 1 \tag{A.3}$$

$$\sum_{\substack{i \in \mathcal{V}: \\ j \neq i}} \sum_{k \in \mathcal{K}} x^b_{(i,k)(j,t)} = \sum_{\substack{i \in \mathcal{V}: \\ j \neq i}} \sum_{k \in \mathcal{K}} x^b_{(j,t)(i,k)} \qquad \forall j \in \mathcal{V}, b \in \mathcal{B}, t \in \mathcal{K} \tag{A.4}$$

$$\sum_{\substack{i \in \mathcal{V}: \\ i > 1}} \sum_{k \in \mathcal{K}} \sum_{t \in \mathcal{K}} x^b_{(i,k)(1,t)} = 1 \qquad b \in \mathcal{B} \tag{A.5}$$

$$\sum_{k \in \mathcal{K}} \sum_{\substack{j \in \mathcal{V}: \\ j > 1}} \sum_{t \in \mathcal{K}} x^b_{(1,k)(j,t)} = 1 \qquad b \in \mathcal{B} \tag{A.6}$$

$$u_{b,i} - u_{b,j} + \left(p_b \sum_{k \in \mathcal{K}} \sum_{t \in \mathcal{K}} x^b_{(i,k)(j,t)} \right) \leq p_b - 1 \qquad \forall i, j \in \mathcal{V}, b \in \mathcal{B}, i > 1, j > 1, i \neq j \tag{A.7}$$

$$0 \leq u_{b,i} \leq p_b - 1 \qquad \forall i \in \mathcal{V}, b \in \mathcal{B} \tag{A.8}$$

$$\sum_{s \in \mathcal{S}} \alpha_{b,s,i} \leq 1 \qquad \forall i \in \mathcal{V}, b \in \mathcal{B}, i > 1 \tag{A.9}$$

$$\alpha_{b,s,i} + \alpha_{b',s',i} \leq 1 \qquad \forall b, b' \in \mathcal{B}, s, s' \in \mathcal{S}, i \in \mathcal{V}, i > 1, s \neq s', b \neq b' \tag{A.10}$$

$$\sum_{i \in \mathcal{V}} \alpha_{b,s,i} = q_{b,s} \qquad \forall b \in \mathcal{B}, s \in \mathcal{S} \tag{A.11}$$

$$x^b_{(i,k)(j,t)} \in \{0, 1\} \qquad \forall i, j \in \mathcal{V}, k, t \in \mathcal{K}, i \neq j \tag{A.12}$$

$$\alpha_{b,s,i} \in \{0, 1\} \qquad \forall i \in \mathcal{V}, s \in \mathcal{S}, b \in \mathcal{B} \tag{A.13}$$

Objective function (A.1) minimizes the total time for picking orders. Note that, to construct contiguous tours, we must keep track of whether a given location is accessed from the left or the right aisle; hence the need to sum over $k \in \mathcal{K} = \{\text{left}, \text{right}\}$. However, this is immaterial for the distances. Our assumption is that aisles are sufficiently narrow that the picker can pick from either side without traversing the aisle. Consequently, the distance from the right side of some shelf to the left side of the shelf opposite it in the same aisle can be set to 0. Also note that the distance matrix $\boldsymbol{t}$ contains the rack movement time when the picker has to switch aisles. Constraints (A.2) and (A.3) ensure that if a SKU s is part of order set b and actually at a given location i (i.e., $\alpha_{b,s,i} = 1$), the location is visited. Constraint (A.4) enforces that if location j is visited from side t, it has to be exited from the same side. Constraints (A.5) and (A.6) show that all tours start from the depot and end at the depot. Constraints (A.7) and (A.8) are subtour elimination constraints for each order in the order set. Constraints (A.9), (A.10), and (A.11) ensure that each shelf location contains at most one SKU and that one specific SKU location is assigned to an order if that order actually contains the SKU (i.e., $q_{b,s} = 1$). Note that this does not preclude multiple copies of the same SKU to be placed in different locations (i.e., shared storage). Constraints (A.11) only enforce that one specific location must be chosen for each order b. Different orders may take the same SKU from different locations, however. The remaining constraints define the domain of the variables.

B. Simulation-optimization tabu search (Chapter 3)

B.1. Multi-fidelity simulations in the neighborhood reduction method

Xu et al. (2014) propose a framework called “ordinal transformation and optimal sampling” within a simulation-optimization algorithm. They use a low-fidelity model to rank all the solutions into several groups. Then, a so-called “optimal sampling” procedure is applied to choose a certain number of candidate solutions from each group for conducting high-fidelity simulation on them. The best-obtained result from high-fidelity simulation is reported as the (near) optimal solution for the studied problem.

In our proposed neighborhood reduction method, given $\mathcal{N}(S)$, the list of all unique neighboring solutions of an incumbent solution S, a low-fidelity simulation run is conducted to evaluate the cost of all neighboring solutions based on objective function 3.1, Section 3.3. Let $j \in J = \{1, \cdots, |\mathcal{N}(S)|\}$ be the index of all neighboring solutions ranked after the low-fidelity simulation run. The solution deemed to be the best receives a rank of one ($j = 1$), and the worst receives the rank of $|\mathcal{N}(S)|$ ($j = |\mathcal{N}(S)|$). The sorted $\mathcal{N}(S)$ is partitioned into a given number of $|K|$ groups ($k \in K = \{1, \cdots, |K|\}$) of size $m = \lceil \frac{|\mathcal{N}(S)|}{|K|} \rceil$, except the last group with the size of m or fewer than m. Let $\overline{Z}_k$ be the mean value and σ_k^2 the variance of the objective values of solutions ($Z'(X_j)$) in group k calculated by Equations B.1 to B.4. The group distance in mean value between two groups (δ_{k_1,k_2}) is calculated by Equation B.5.

$$\overline{Z}_k = \frac{1}{m} \sum_{j=(k-1)m+1}^{km} Z'(X_j) \qquad k \neq |K| \in K \quad \text{(B.1)}$$

$$\overline{Z}_k = \frac{1}{|\mathcal{N}(S)| - (m-1)|K|} \sum_{j=(k-1)m+1}^{|\mathcal{N}(S)|} Z'(X_j) \qquad k = |K| \quad \text{(B.2)}$$

$$\sigma_k^2 = \frac{1}{m-1} \sum_{j=(k-1)m+1}^{km} \left(Z'(X_j) - \overline{Z}_k\right)^2 \qquad k \neq |K| \in K \quad \text{(B.3)}$$

$$\sigma_k^2 = \frac{1}{|\mathcal{N}(S)| - (m-1)|K| - 1} \sum_{j=(k-1)m+1}^{|\mathcal{N}(S)|} \left(Z'(X_j) - \overline{Z}_k\right)^2 \qquad k = |K| \quad \text{(B.4)}$$

$$\delta_{k_1,k_2} = \left|\overline{Z}_{k_1} - \overline{Z}_{k_2}\right| \qquad k_1, k_2 \in K \quad \text{(B.5)}$$

Xu et al. (2014) recommend avoiding selecting solutions only from the best group, the one with the index of $k = 1$ which contains lower ranked solutions. The rationale behind this is the unknown and potentially significant errors in low-fidelity models. In addition, the optimal solution (and for our case, the best non-tabu neighboring solution) might have a higher rank or might be in a non-top group. Therefore, they use a selecting procedure called “optimal sampling” to choose the candidate solutions from each group for conducting high-fidelity simulation on them. We modify their optimal sampling procedure and proposed an iterative algorithm depicted by Algorithm 4 to calculate the number of top solutions from each group.

Algorithm 4 Pseudocode Overview for modified selecting procedure

```
1:  procedure MODIFIED SELECTING PROCEDURE
2:      $B \leftarrow 0$
3:      $B^{past} \leftarrow 0$
4:      Initialize $N_{|K|}$
5:      Calculate $N_k|(k \in K \& k \neq 1, |K|)$ based on $N_{|K|}$
6:      Calculate $N_1$
7:      Calculate $B$
8:      if $B < \theta$ then
9:          update $B^{past}$ and $N_k^{past}$
10:         Increase $N_{|K|}$ and go to $3$
11:     else
12:         if $|\theta - B^{past}| \leq |\theta - B|$ then
13:             create $\mathcal{N}_r(S)$ based on $\lceil N_k^{past} \rceil$
14:         else
15:             create $\mathcal{N}_r(S)$ based on $\lceil N_k \rceil$
16:     return $\mathcal{N}_r(S)$
```

The goal of the algorithm is to select θ neighboring solutions from $\mathcal{N}(S)$. Let B and B^{past} be the total number of solutions chosen in the current iteration and the previous iteration, respectively. They are initialized on lines 2 and 3. Also, let N_k and N_k^{past} be the number of solutions chosen from each group $k \in K$, in the current and previous iteration. To calculate N_k, first, we set $N_{|K|} = 1$ (line 4). Then, at line 5, the number of solutions from each group, except for the best group, is calculated by Equation B.6. In Equation B.6, the larger the group distance between the best group and group k, the smaller the number of high-fidelity evaluations allocated to group k. It is because such a group is unlikely to contain better solutions. In addition, if the group variance is high, that group receives more high-fidelity evaluations because there is more uncertainty about the objective value of the solution in this group.

On line 6, the number of solutions from the best group ($k = 1$) is calculated by Equation B.7. Subsequently, at line 7, the total number of solutions picked is calculated using Equation B.8. If this number is less than the aimed number of θ, then the algorithm increases the initial number of solutions of the last group ($N_{|K|}$) and does the whole steps for calculation of N_k again. However, it memorizes the currently obtained results and updates the values of B^{past} and N_k^{past} (line 9). If the total number of solutions is higher than θ, the algorithm stops. Then, the algorithm reports the values of N_k if $|\theta - B| < |\theta - B^{past}|$; otherwise, the values of N_k^{past} are reported (lines 12-15).

$$N_k = \left\lceil N_{|K|} \cdot \left(\frac{\delta_{1,|K|}/\sigma_{|K|}}{\delta_{1,k}/\sigma_k}\right)^2 \right\rceil \qquad k \neq 1 \,\&|K| \in K \tag{B.6}$$

$$N_1 = \left\lceil \sigma_1 \sqrt{\sum_{k=2}^{|K|} \frac{N_k^2}{\sigma_k^2}} \right\rceil \tag{B.7}$$

$$B = \sum_{k=1}^{|K|} N_k \tag{B.8}$$

B.2. Performance of the algorithm

B.2.1. Value of moves

In this subsection, the value of moves, as an improvement part of the algorithm, is studied. In Section 3.4.1, two moves, namely right-shift and left-shift are introduced to create the proposed composed neighborhood structure. In the literature related to the tabu search algorithm, when using the similar structure to represent the solution, see Section 3.4.1, another move called “exchange” can be used, in which the amount of $X(t)$ and $X(T - t)$ are swapped, see Cordeau et al. (2001).

We apply seven different combinations of right-shift, left-shift, and exchange moves in the proposed algorithm and compare their performance with each other for small instances from 3.5.2. In Table B.1, columns containing “E” show that the exchange

Table B.1.: The comparison of the effects of the moves in the algorithm's performance- medium instances

Pri.	N	T	BKS	R-L-E		R-L		L-E		R-E		R		L		E	
				Gap (%)	CPU (s)	Gap (%)	CPU (s)	Gap (%)	CPU (s)	Gap (%)	CPU (s)	Gap (%)	CPU (s)	Gap (%)	CPU (s)	Gap (%)	CPU (s)
P_1	5	10	705.46	0.00	67.7	0.00	48.9	0.00	63.4	0.00	84.2	0.70	27.5	0.24	30.6	1.09	30.9
	6	10	1019.93	0.00	100.7	0.00	80.5	0.00	77.5	1.75	80.3	0.00	69.7	0.00	42.5	0.63	34.7
	7	10	1409.40	0.00	115.1	0.00	111.6	0.00	99.4	0.00	122.1	0.00	131.3	0.00	66.0	0.27	29.1
	8	10	1807.47	0.00	190.2	0.00	151.2	0.47	135.1	0.02	232.0	0.00	129.7	0.00	146.0	0.38	37.3
	9	10	2231.84	0.00	219.4	0.00	195.2	0.00	214.9	0.14	193.4	0.17	150.4	0.17	62.8	0.41	55.4
	10	10	2734.95	0.00	264.0	0.00	279.4	0.36	278.5	0.00	230.9	0.54	167.3	0.49	143.7	0.71	64.4
	Ave.			0.00	159.5	0.00	144.5	0.14	144.8	0.32	157.2	0.23	112.7	0.15	81.9	0.58	42.0
P_2	5	10	707.28	0.00	70.0	0.00	48.4	0.00	67.1	0.00	90.9	0.67	33.6	0.37	36.3	1.09	26.3
	6	10	1023.58	0.00	95.0	0.00	86.9	0.00	84.9	0.63	93.5	0.33	54.4	0.00	45.1	0.59	38.6
	7	10	1414.65	0.00	112.7	0.00	124.3	0.07	106.8	0.00	165.6	0.24	164.2	0.00	74.6	0.25	33.4
	8	10	1815.55	0.00	226.1	0.00	161.9	0.46	150.9	0.09	199.6	0.09	136.6	0.12	105.9	0.24	51.4
	9	10	2245.87	0.00	265.6	0.00	207.1	0.26	308.3	0.08	280.2	0.00	156.9	0.34	64.8	0.50	77.4
	10	10	2752.13	0.00	433.2	0.00	236.9	1.02	243.4	0.09	259.2	0.00	222.5	0.20	108.1	1.73	40.0
	Ave.			0.00	200.4	0.00	144.3	0.30	160.2	0.15	181.5	0.22	128.0	0.17	72.5	0.74	44.5
$P_3(1)$	5	10	712.11	0.00	65.1	0.00	49.2	0.00	63.1	0.00	76.4	0.65	31.0	0.25	32.4	1.11	38.9
	6	10	1033.22	0.00	80.1	0.00	75.9	0.00	81.9	1.73	79.8	0.00	72.1	0.00	43.8	2.25	21.3
	7	10	1427.42	0.00	129.3	0.00	106.3	0.00	176.0	0.00	198.2	0.00	123.8	0.00	68.4	0.72	57.2
	8	10	1830.07	0.00	222.0	0.00	156.2	0.28	154.7	0.01	117.6	0.00	126.4	0.19	109.8	0.28	50.1
	9	10	2265.09	0.00	280.2	0.00	182.6	0.27	171.0	0.12	215.8	0.16	142.1	0.00	130.0	0.89	34.6
	10	10	2773.73	0.00	294.3	0.00	225.9	0.68	180.6	0.00	237.1	0.00	163.9	0.54	91.0	1.45	34.9
	Ave.			0.00	178.5	0.00	132.7	0.21	137.9	0.31	154.2	0.13	109.9	0.16	79.2	1.12	39.5

move is included in the set of moves. Columns “L” and “R” show right-shift and left-shift are included in the move set, respectively.

As shown in Table B.1, when these two moves are used together, the algorithm can find all the BKSs. It occurs even when exchange move is included in the set of moves with them. Additionally, when we use only right-shift together with left-shift, the CPU time decreases in comparison with when exchange move is included, for all instances except two. Apart from that, Table B.1 shows that for some instances, if right-shift and left-shift are not included in the move set together, the algorithm gets stuck in a local optima even with larger CPU time. To sum up, the importance of using moves right-shift and left-shift together is vivid.

To more investigate the effects of the exchange move specifically, we test only two sets of moves, namely right-shift together with left-shift (R-L) and all of them together (R-L-E), for large instances. It should be noted that we investigated the effects of other combinations on solving the large instances in the pre-numerical studies and we observed the similar results and conclusions which are explained for Table B.1. Table B.2 presents the obtained results.

We find that using only right-shift and left-shift moves together can result in finding the best-known solution for all cases but one. In instance with $N = 10$, $T = 20$, and $\alpha = P_1$, the relative gap with the BKS is 0.01%. On the other hand, while using all moves together, still the algorithm performs very well and can find the BKS for all instances, except for only one. The relative gap to the BKS for instance with $N = 20$, $T = 20$, and $\alpha = P_3(1)$, is 0.03%. It also shows that using all of the moves together is more time-costly than when only right-shift and left-shift moves are used.

Considering all tables together, we can conclude that using the exchange move is not beneficial for all cases but one. Even, in some instances, using the exchange move misleads the search procedure to a local optima with low quality. This is because the number of different new neighboring solutions from the incumbent solution while using exchange move is less than that of right-shift as well as left-shift moves and it causes less intensifying the search.

It should be noted that there is no direct link between the number of moves and the quality of the final obtained solution, however, for our specific studied appointment scheduling problem, the importance of right-shift and left-shift moves together is clear.

Table B.2.: The comparison of the effects of the moves in the algorithm's performance- large instances- only for sets {Right-shift, left-shift, exchange} and {Right-shift, left-shift}

Pri.	N	T	BKS	R-L		R-L-E	
				Gap($\%$)	CPU (s)	Gap($\%$)	CPU (s)
P_1	10	10	2734.95	0.00	262.4	0.00	251.8
	15	10	5309.48	0.00	631.4	0.00	833.1
	20	10	8231.87	0.00	1337.3	0.00	1120.1
	10	15	1971.69	0.00	473.8	0.00	659.5
	15	15	4335.63	0.00	968.7	0.00	1680.7
	20	15	7068.09	0.00	2622.7	0.00	2477.4
	10	20	1421.58	0.01	416.4	0.00	542.2
	15	20	3487.02	0.00	1836.5	0.00	1611.3
	20	20	6192.29	0.00	3500.7	0.00	4289.2
	Ave.			0.00	1338.9	0.00	1496.1
P_2	10	10	2752.13	0.00	222.9	0.00	429.1
	15	10	5346.05	0.00	725.9	0.00	727.4
	20	10	8296.22	0.00	1488.1	0.00	1375.6
	10	15	1987.65	0.00	504.7	0.00	448.6
	15	15	4380.08	0.00	929.0	0.00	1441.6
	20	15	7153.33	0.00	2366.1	0.00	2720.5
	10	20	1435.24	0.00	634.1	0.00	882.6
	15	20	3534.01	0.00	2095.3	0.00	2313.6
	20	20	6283.27	0.00	3533.3	0.00	4447.5
	Ave.			0.00	1388.8	0.00	1643.0
$P_3(1)$	10	10	2773.73	0.00	206.2	0.00	281.5
	15	10	5375.40	0.00	566.8	0.00	908.0
	20	10	8326.95	0.00	1433.2	0.00	1156.8
	10	15	2016.50	0.00	529.2	0.00	426.1
	15	15	4419.08	0.00	1439.6	0.00	1344.7
	20	15	7207.70	0.00	2991.7	0.00	2801.3
	10	20	1467.43	0.00	603.2	0.00	694.3
	15	20	3585.28	0.00	1685.6	0.00	2532.6
	20	20	6348.95	0.00	3622.1	0.03	4080.4
	Ave.			0.00	1453.1	0.00	1580.6

C. Effects of the parameters of the approximation method: Effects of ω (Chapter 4)

In the underloaded periods, the parameters related to SBC for different queues are well-studied in the literature, for example see Stolletz (2008a,b); Stolletz and Lagershausen (2013). For overloaded periods and for when there is a transition from an overloaded period to an underloaded period, we analyze the effects of the parameters of the proposed hybrid approximation method on the quality of the approximated performance measure in this Section.

In the proposed hybrid approximation method for overloaded periods, the approximated expected utilization is multiplied by the service rate to calculate the amount of flow out in the fluid flow formula, see Section 4.3 Equation (4.8). In the PSFFA approach, for an $M(t)/M/1$ queue, the value of the ω is one. However, the pretests show that $\omega = 1$ is not a proper choice when the system is in overload situation. In this subsection, we analyze parameter ω for different traffic intensity values equal and larger than one on the performance of the proposed hybrid approximation method. We increase the value of parameter ω step by step and compare the performance of the approximation method with that of simulation for the expected work-in-process ($E[L]$), while the queueing system starts empty. The period length (d) is selected to be one time unit and the service rate is $\mu = 1$ per time unit. Figures C.1 to C.6 show the transient behavior of the proposed hybrid approximation method in overloaded situations. The absolute value of the relative gaps between $E[L]$ from the proposed hybrid approximation method and that from the simulation are also depicted.

When the load of the system is one ($\rho = 1$), $\omega = 0.7$ outperforms other values of ω. For $\rho = 3.5$, on the other hand, a range of values of ω from 0.1 to 0.6 makes the approximation method perform well. We can observe that in the high traffic intensities, the larger range of values of ω results well in terms of the absolute value of the relative gap. The reason is in high loads, the queues form faster and when the expected number of jobs in the system is very high, the value of the approximated expected utilization in Equation (4.13) for an arbitrary t becomes very close to each other for

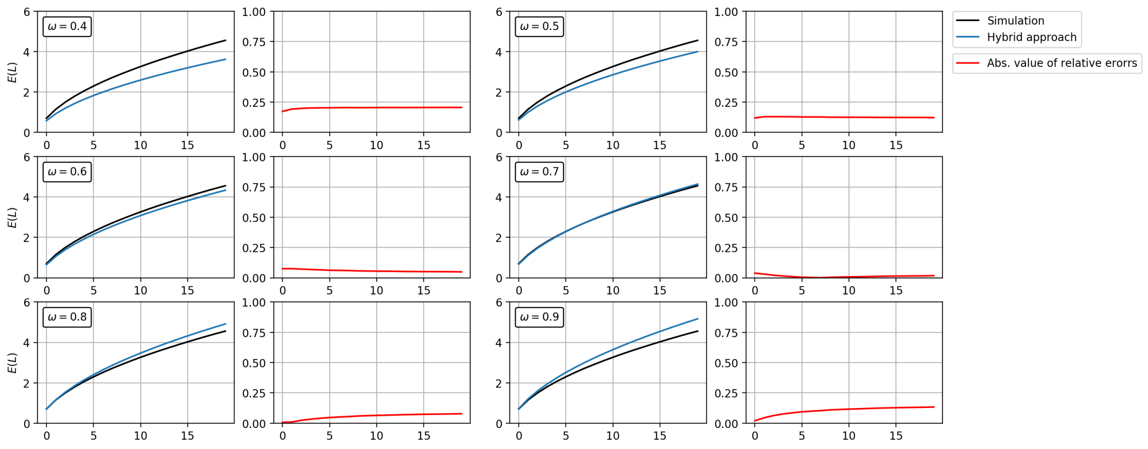


Figure C.1.: The effects of ω on the quality of the approximation method in overloaded systems with $\rho = 1$ and $\mu = 1$

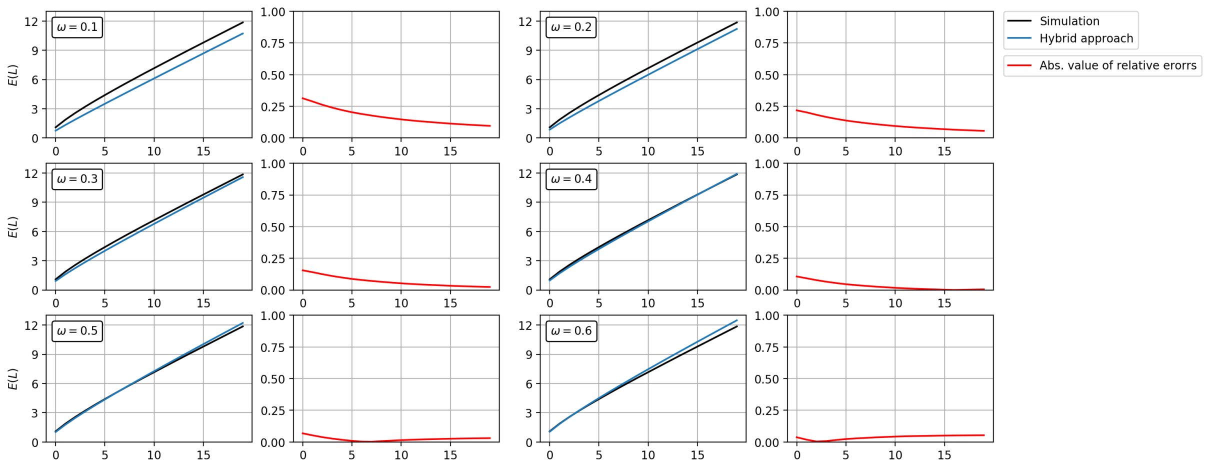


Figure C.2.: The effects of ω on the quality of the approximation method in overloaded systems with $\rho = 1.5$ and $\mu = 1$

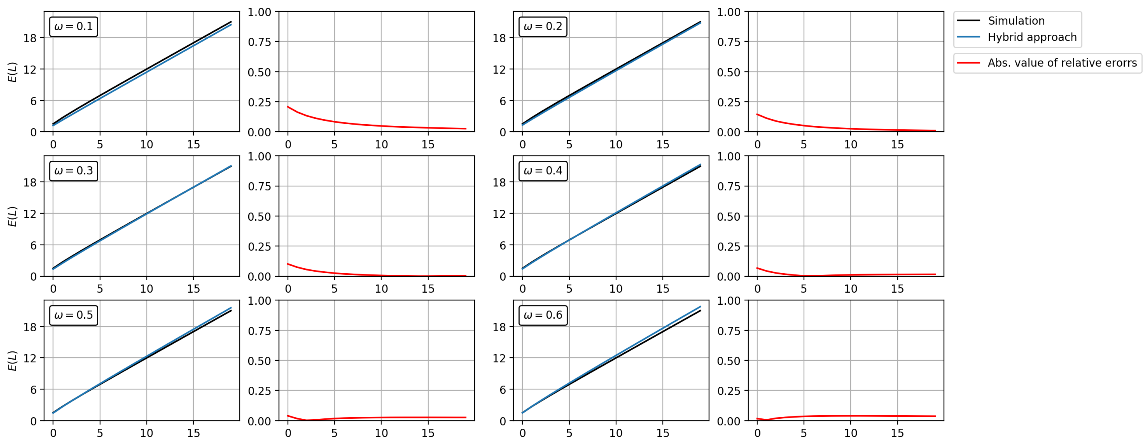


Figure C.3.: The effects of ω on the quality of the approximation method in overloaded systems with $\rho = 2$ and $\mu = 1$

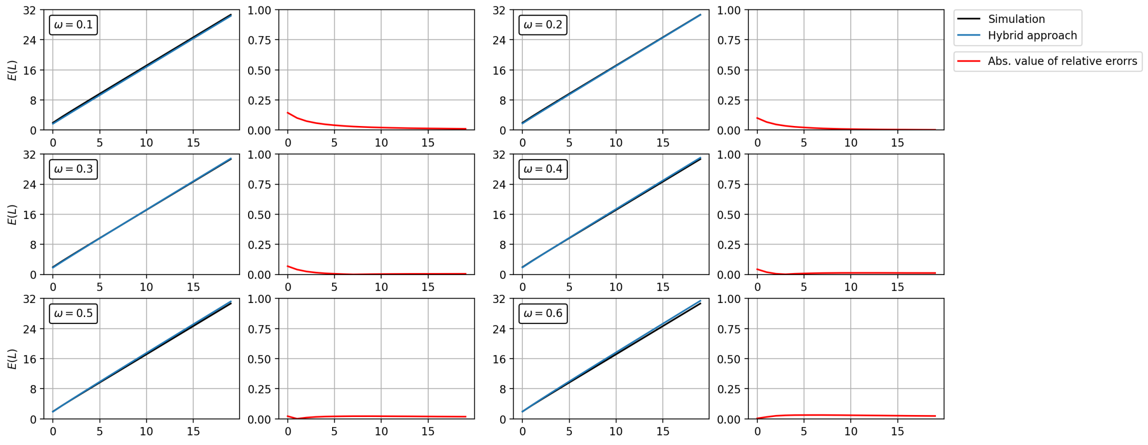


Figure C.4.: The effects of ω on the quality of the approximation method in overloaded systems with $\rho = 2.5$ and $\mu = 1$

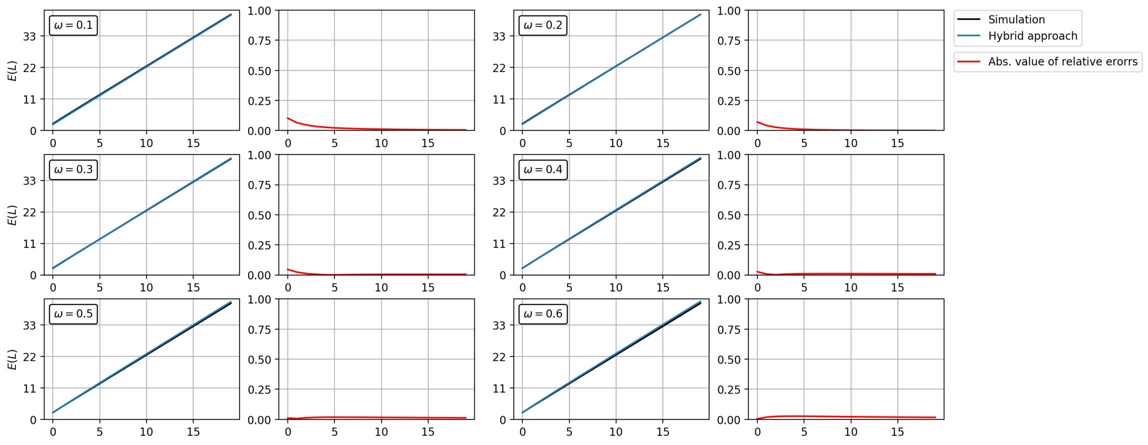


Figure C.5.: The effects of ω on the quality of the approximation method in overloaded systems with $\rho = 3$ and $\mu = 1$

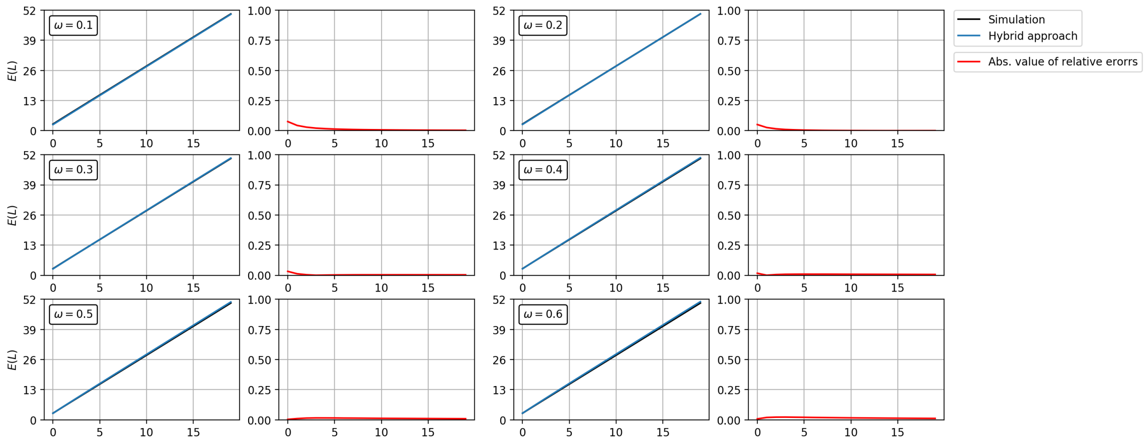


Figure C.6.: The effects of ω on the quality of the approximation method in overloaded systems with $\rho = 3.5$ and $\mu = 1$

more values of ω, and then the approximated $E[L]$ by the proposed method is near the approximated $E[L]$ by simulation.

When the values of ω are smaller than one, the performance of the approximation method for overloaded periods with a high value of ρ is well. If the system is critically loaded and starts empty, the value of ω should be 0.7. In this experiment, a range of values from $\omega = 0.3$ to $\omega = 0.5$ outperform other values of ω when ρ is larger than one.

D. Java codes of the hybrid approximation method (Chapter 4)

```java
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package Mannheim.Approach;

import Mannheim.Instance.Instance;
import Mannheim.core.Solution;

/**
 *
 * @author Amir
 */
abstract public class Approach
{
   public Instance instance;
   public Solution solution;

   public double[] length_period_app;



   public void setInstance(Instance instance)
   {
      this.instance = instance;
   }

   public Instance getInstance()
   {
      return this.instance;
   }

   public void setSolution(Solution solution)
   {
      this.solution = solution;
   }

   public Solution getsolution()
   {
      return this.solution;
   }

   abstract public void run();

}
```

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package Mannheim.Approach;

import Mannheim.Queue.MMc_over;

/**
 *
 * @author Amir
 */
public class Fluid_M_M_c extends Approach
{

   @Override
   public void run()
   {
      MMc_over mmc_over = new MMc_over(0.00001);
      mmc_over.setOmega(1);

      //constant mu is assumed
      double mu = instance.ser_rate;
      double lambda;
      int c;
      int k;
      double timeStep = 1.0 / mu;
      //the time where the last phase ends
      double endTime = instance.pla_hor;
      //the number of time steps to initialize the arrays with the correct length
      int nrTimeSteps = (int) (Math.ceil(endTime / timeStep));
      //the current time considered in the loop
      double currentTime = 0;

      double x_0 = 0;
      double x = 0;
      double y = 0;
      double timeStep_psfa = 0.0001;
      double util = 0;
      int nrTimeStepsPSFA = (int) Math.ceil(timeStep / timeStep_psfa);

      //the arrays are initialized
      solution.utilization = new double[nrTimeSteps];
      solution.expWaitingTime = new double[nrTimeSteps];
      solution.expCycleTime = new double[nrTimeSteps];
      solution.expNoWaiting = new double[nrTimeSteps];
      solution.expNoInSystem = new double[nrTimeSteps];
      solution.time = new double[nrTimeSteps];

      //Loop: for each station and time the performance values are calculated
      for (int t = 0; t < solution.time.length; t++) {

         //the current time is saved
         if (Math.ceil(currentTime) - currentTime <= 0.0001) {
            solution.time[t] = Math.ceil(currentTime);
         } else {
            solution.time[t] = currentTime;
         }
```

```
        //time bigger then the end time of the last phase should not be considered.
        //Therefore we set the last time considered as end time. After this run the loop
            will stop.
        if (solution.time[t] > endTime) {
           solution.time[t] = endTime;
        }

        //updating the parameters
        int currentPhase = (int) Math.min(Math.floor(solution.time[t] /
            instance.len_period), instance.arr_rate.length - 1);

        lambda = instance.arr_rate[currentPhase];
        c = instance.ser_num[currentPhase];
        // the max queuing length is equal then the server number because we use the mmcc
            model
        k = c;

        //using stationary MMcc (phase 1)
        if (t == 0) {
           x_0 = lambda / (mu * c);
           mmc_over.resetQueue(x_0);
           double rho = 0;
           for (int i = 0; i < nrTimeStepsPSFA; i++) {
              mmc_over.resetQueue(x_0);
              rho = 1;
              x = x_0 + (lambda - mu * rho * c) * timeStep_psfa;
              x = Math.max(0, x);
              x_0 = x;

           }
        } else {
           mmc_over.resetQueue(x_0);
           double rho = 0;
           for (int i = 0; i < nrTimeStepsPSFA; i++) {
              mmc_over.resetQueue(x_0);
              rho = 1;
              x = x_0 + (lambda - mu * rho * c) * timeStep_psfa;
              x = Math.max(0, x);
              x_0 = x;
              y = x/mu;

           }

        }
        solution.expNoInSystem[t] = x;
        solution.expWaitingTime[t] = y;
//                     System.out.println("t: " + t);
//                     System.out.println("time: " + solution.time[t]);
//                     System.out.println("artificialLambda " + artificialLambda);
//                     System.out.println("modifiedLambda " + modifiedLambda);
//                     System.out.println("backlogRate(in t-1): " + backlogRate);
//                     System.out.println("WIP:" + solution.expNoInSystem[t]);
//
    System.out.println("__________________________________________________________________________________________
        currentTime = currentTime + timeStep;
        /**
         * Normal PSFA.
         */
        }
    }

}
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package Mannheim.Approach;

import Mannheim.Queue.MMc;
import Mannheim.Queue.MMcc;
import Mannheim.Queue.MMcK;
import Mannheim.Queue.MMc_over;
import Mannheim.Run.Runner;

/**
 *
 * @author Amir Foroughi <aforough at mail-mannheim.de>
 */
public class Hybrid_M_M_c_BacklogColib extends Approach
{

//   double WIP_from_sa = Runner.WIP_from_run_sa;
    @Override
    public void run()
    {

        //The mmcc is used to calculate the expected utilization ( k = c)
        MMcc mmcc = new MMcc(instance.arr_rate[0], instance.ser_rate, instance.ser_num[0],
            instance.ser_num[0]);
        //The mmc is used to calculate the expected waiting time
        MMc mmc = new MMc(instance.arr_rate[0], instance.ser_rate, instance.ser_num[0]);
        MMc_over mmc_over = new MMc_over(0.1);

        double initial_WIP = 0.00001;
        /**
         * OMEGA HERE.
         */
        mmc_over.setOmega(0.3);

        int counter_of_iteration = 0;
        double parameters = 0.999;

        double backlogRate = 0;
        double artificialLambda = 0;
        double modifiedLambda = 0;
        //constant mu is assumed
        double mu = instance.ser_rate;
        double lambda;
        int c;
        int k;
        double timeStep = 1.0 / mu;
        //the time where the last phase ends
        double endTime = instance.pla_hor;
        //the number of time steps to initialize the arrays with the correct length
        int nrTimeSteps = (int) (Math.ceil(endTime / timeStep));
```

```
      //the current time considered in the loop
      double currentTime = 0;

      double x_0 = 0;
      double x = 0;
      double y_0 = 0;
      double y = 0;
      double timeStep_psfa = 0.0001;
      double util = 0;
      int nrTimeStepsPSFA = (int) Math.ceil(timeStep / timeStep_psfa);

      //the arrays are initialized
      solution.utilization = new double[nrTimeSteps];
      solution.expWaitingTime = new double[nrTimeSteps];
      solution.expCycleTime = new double[nrTimeSteps];
      solution.expNoWaiting = new double[nrTimeSteps];
      solution.expNoInSystem = new double[nrTimeSteps];
      solution.time = new double[nrTimeSteps];

      boolean if_prv_period_overloaded = false;

      //Loop: for each station and time the performance values are calculated
      for (int t = 0; t < solution.time.length; t++) {

         //the current time is saved
         if (Math.ceil(currentTime) - currentTime <= 0.0001) {
            solution.time[t] = Math.ceil(currentTime);
         } else {
            solution.time[t] = currentTime;
         }

         //time bigger then the end time of the last phase should not be considered.
         //Therefore we set the last time considered as end time. After this run the loop
             will stop.
         if (solution.time[t] > endTime) {
            solution.time[t] = endTime;
         }

         //updating the parameters
         int currentPhase = (int) Math.min(Math.floor(solution.time[t] /
             instance.len_period), instance.arr_rate.length - 1);

         lambda = instance.arr_rate[currentPhase];
         c = instance.ser_num[currentPhase];
         // the max queuing length is equal then the server number because we use the mmcc
             model
         k = c;

         //using stationary MMcc (phase 1)
         artificialLambda = lambda + backlogRate;
         mmcc.resetQueue(artificialLambda, mu, c, c);
         util = artificialLambda * (1 - mmcc.getPn(k)) / (c * mu);

         if (lambda / (mu * c) < 1) {
//             if (true) {
            if (!if_prv_period_overloaded) {
               //using stationary MMc (phase 2)
               solution.utilization[t] = util;
               modifiedLambda = solution.utilization[t] * k * mu;
               mmc.resetQueue(modifiedLambda, mu, c);

               //saving the results of the calculation in two dimensional arrays
               solution.expWaitingTime[t] = mmc.getExpWaitingTime();
               solution.expCycleTime[t] = mmc.getExpCycleTime();
               solution.expNoWaiting[t] = mmc.getExpNoWaiting();
               solution.expNoInSystem[t] = mmc.getExpNoInSystem();

               x_0 = solution.expNoInSystem[t];
               y_0 = solution.expNoWaiting[t];
               if_prv_period_overloaded = false;
//                System.out.println("t: " + t);
//                System.out.println("time: " + solution.time[t]);
//                System.out.println("artificialLambda " + artificialLambda);
//                System.out.println("modifiedLambda " + modifiedLambda);
//                System.out.println("backlogRate(in t-1): " + backlogRate);
//                System.out.println("pn " + mmcc.getPn(k));
//                System.out.println("Util: " + solution.utilization[t]);
//                System.out.println("WIP:" + mmc.getExpNoInSystem());
//
    System.out.println("__________________________________________
            } else {
               if (counter_of_iteration % 50 == 0) {
                  parameters = parameters - 0.005;
               }
               modifiedLambda = util * k * mu;
               mmc.resetQueue(modifiedLambda, mu, c);
               t = t - 1;
               currentTime = currentTime - timeStep;
//                if_change = true;

               if (x >= mmc.getExpNoInSystem()) {
                  if_prv_period_overloaded = false;
                  counter_of_iteration = 0;
                  parameters = 0.999;
               }

               artificialLambda *= parameters;
//                System.out.println("x = " + x);
//                System.out.println(mmc.getExpNoInSystem());
//                System.out.println("artificialLambda: " + artificialLambda);
//                System.out.println("modifiedLambda: " + modifiedLambda);
//                System.out.println("util: " + util);
               counter_of_iteration++;
            }

            //updating the backlog rate
//             backlogRate = artificialLambda * mm11.getPn(k);
            //updating the current time
//             currentTime = currentTime + timeStep;
         } /**
          * Normal PSFA.
          */
         else {
            if (t == 0) {
               x_0 = lambda / (mu * c);
//                x_0 = initial_WIP;
              y_0 = 0.005;
```

```
                mmc_over.resetQueue(x_0);

//              double omegaa = 0.7 * c;
//              mmc_over.setOmega(omegaa);

                double rho = 0;
                for (int i = 0; i < nrTimeStepsPSFA; i++) {
                   mmc_over.resetQueue(x_0);
                   rho = mmc_over.getRho();
                   x = x_0 + Math.max(0, lambda - mu * rho * Math.min(c, x)) * timeStep_psfa;
                   x_0 = x;

                   mmc_over.resetQueue(y_0);
                   rho = mmc_over.getRho();
                   y = y_0 + (lambda - mu * rho * c) * timeStep_psfa;
                   y_0 = y;

                }
             } else {
                mmc_over.resetQueue(x_0);

//              double omegaa;
//              if (mu == 1) {
//                 omegaa = 0.7;
//              } else {
//                 omegaa = (0.2 * Math.exp(-.8 * (lambda / (c * mu))) + 0.1) * mu / 4;
//              }
//              mmc_over.setOmega(omegaa);

                double rho = 0;
                for (int i = 0; i < nrTimeStepsPSFA; i++) {
                   mmc_over.resetQueue(x_0);
                   rho = mmc_over.getRho();
                   x = x_0 + Math.max(0, lambda - mu * rho * Math.min(c, x)) * timeStep_psfa;
                   x_0 = x;

                   mmc_over.resetQueue(y_0);
                   rho = mmc_over.getRho();
                   y = y_0 + (lambda - mu * rho * c) * timeStep_psfa;
                   y_0 = y;
                }

             }
             solution.expNoInSystem[t] = x;
             solution.expNoWaiting[t] = y;

//           System.out.println("t: " + t);
//           System.out.println("time: " + solution.time[t]);
//           System.out.println("artificialLambda " + artificialLambda);
//           System.out.println("modifiedLambda " + modifiedLambda);
//           System.out.println("backlogRate(in t-1): " + backlogRate);
//           System.out.println("WIP:" + solution.expNoInSystem[t]);
//
     System.out.println("____________________________________________________________
             if_prv_period_overloaded = true;

          }
//        if (!if_prv_period_overloaded && !if_change) {
//           backlogRate = artificialLambda * mm11.getPn(k);
//          }
//          if_change = false;
         backlogRate = artificialLambda * mmcc.getPn(k);
         currentTime = currentTime + timeStep;

      }
   }
}
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package Mannheim.Instance;

import Instances.P;
import Mannheim.core.Phase;

/**
 *
 * @author Amir
 */
public class Instance
{

   public String name;

   //<editor-fold defaultstate="collapsed" desc="Important info">
   public String queue;
//</editor-fold>

   //<editor-fold defaultstate="collapsed" desc="Session">
   public int num_period;
   public double len_period;
   public double pla_hor;
//</editor-fold>

   //<editor-fold defaultstate="collapsed" desc="Service distribution info">
   public String ser_dist;
   public double ser_mean;
   public double ser_var;
   public double ser_st;
   public double ser_cv2;
   public double ser_rate;
   public double[] rho;
   public int[] ser_num;
//</editor-fold>


   //<editor-fold defaultstate="collapsed" desc="Arrival distribution info">
   public String arr_dist;
   public String batch_dist;

   public double[] arr_rate;
   public double[] batch_cv2;
   public double[] batch_mean;
   public double[] batch_var;
   public double[] distance_batch;
   /**
```

```
     * P(X = 1).
     */
    public double[] g_1;
//</editor-fold>

    void makeInstance()
    {
        //<editor-fold defaultstate="collapsed" desc="Service distribution info">
        pla_hor = num_period * len_period;
//</editor-fold>

        //<editor-fold defaultstate="collapsed" desc="Service distribution info">
        ser_mean = (ser_rate > 0 ? 1 / ser_rate : 99999999);
        ser_cv2 = 1;
        ser_var = ser_cv2 * ser_mean * ser_mean;
        ser_st = Math.sqrt(ser_var);
        rho = new double[arr_rate.length];
        if (queue.equalsIgnoreCase("M_M_1")) {
            for (int i = 0; i < arr_rate.length; i++) {
                rho[i] = arr_rate[i] / ser_rate;
            }
        } else {
            for (int i = 0; i < arr_rate.length; i++) {
                rho[i] = (arr_rate[i] * batch_mean[i]) / ser_rate;
            }
        }

//</editor-fold>
        //<editor-fold defaultstate="collapsed" desc="Arrival">
        if (queue.equalsIgnoreCase("M_M_1_batch")) {
            batch_cv2 = new double[batch_mean.length];
            batch_var = new double[batch_mean.length];
            g_1 = new double[batch_mean.length];

            for (int i = 0; i < arr_rate.length; i++) {
                /**
                 * Because in Geometric, we use the probability rate, not average.
                 * https://en.wikipedia.org/wiki/Geometric_distribution
                 */
                batch_mean[i] = 1.0/ batch_mean[i];
                if (batch_dist.equalsIgnoreCase("Geo")) {
                    g_1[i] = 1 / batch_mean[i];
                    batch_var[i] = (1 - g_1[i]) / (Math.pow(g_1[i], 2));
                    batch_cv2[i] = batch_var[i] / (Math.pow(batch_mean[i], 2));
                } else if (batch_dist.equalsIgnoreCase("Det")) {
                    g_1[i] = (batch_mean[i] == 1 ? 1 : 0);
                    batch_var[i] = 0;
                    batch_cv2[i] = 0;
                } else {
                    batch_var[i] = (Math.pow(2 * distance_batch[i] + 1, 2) - 1) / 12;
                    batch_cv2[i] = batch_var[i] / Math.pow(batch_mean[i], 2);
                    if (batch_mean[i] - distance_batch[i] <= 1) {
                        g_1[i] = 1.0 / (2 * distance_batch[i] + 1);
                    } else {
                        g_1[i] = 0;
                    }
                }
            }
        }
```

```
//</editor-fold>

   }

   public void showInstance()
   {
      System.out.println("Name: "+ name);
      if (queue.equalsIgnoreCase("M_M_1_batch")) {
         System.out.println("Queue:\tM(t)^{X(t)}/M/1");
         System.out.println("Sesison:\tLength: " + len_period + ",\tNumber of periods: " +
             num_period + ",\tPlanning horizon: " + pla_hor);
         System.out.println("Arrivals:\tBatch_dist:" + batch_dist);
         System.out.println("Rates:");
         P.write(arr_rate);
         P.write(batch_mean);
         System.out.println("Other info:");
         P.write(batch_cv2);
         System.out.println("Service:\tRate:" + ser_rate);
         P.makeItDotted('-');
      } else {
         System.out.println("Queue:\tM(t)/M/1");
         System.out.println("Sesison:\tLength:" + len_period + ",\tNumber of periods:" +
             num_period + ",\tPlanning horizon:" + pla_hor);
         System.out.println("Arrivals:");
         System.out.println("Rates:");
         P.write(arr_rate);
         System.out.println("Other info:");
         System.out.println("Service:\tRate:" + ser_rate);
         P.makeItDotted('-');
      }

   }


}
package Mannheim.Instance;

import java.io.BufferedReader;
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.InputStreamReader;
import java.util.HashMap;

public class MapClassInstance
{
      private final HashMap<String, String> data;

      private MapClassInstance()
      {
            data = new HashMap<String, String>();
      }

      public void init(String path)
      {
            try {
                  DataInputStream in = new DataInputStream(new FileInputStream(path));
                  BufferedReader br = new BufferedReader(new InputStreamReader(in));
                  String line;
            while ((line = br.readLine()) != null) {
                if (line.indexOf('#') != -1) {
                    line = line.substring(0, line.indexOf('#'));
                }
                line = line.trim();
                if (!line.isEmpty()) {
                    parseProperty(line);
                }
            }
            in.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
}

private static class InstanceHolder
{
        private static final MapClassInstance INSTANCE = new MapClassInstance();
}


public static MapClassInstance getInstance()
{
        return InstanceHolder.INSTANCE;
}

public static double getDouble(String key)
{
        return Double.valueOf(getInstance().data.get(key));
}

public static int getInt(String key)
{
        return Integer.valueOf(getInstance().data.get(key));
}

public static boolean getBoolean(String key)
{
        return getInstance().data.get(key).equals("true");
}

public static String getString(String key)
{
        return getInstance().data.get(key);
}

public static String[] getArray(String key)
{
        return getString(key).split("\\s*,\\s*");
}

public static int getArrayLength(String key)
{
        return getArray(key).length;
}

public void parseProperty(String property)
{
        String[] tokens = property.split("=");
```

```
            set(tokens[0], tokens[1]);
      }

      public void set(String key, String value)
      {
            data.put(key.trim(), value.trim());
      }

      public void set(String key, Number value)
      {
            data.put(key.trim(), value.toString());
      }

      public boolean hasProperty(String key)
      {
            return data.containsKey(key);
      }

}/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package Mannheim.Queue;

/**
 *
 * @author Amir Foroughi <aforough at mail-mannheim.de>
 */
public class MMcc extends Queue
{

   public int k;

   public MMcc(double lambda, double mu, int c, int k)
   {
      this.lambda = lambda;
      this.mu = mu;
      this.c = c;
      this.k = k;
      this.lambdaeff = getLambdaeff();

   }
   /**
    * This method calculates the value of the expected number waiting.
    * @return Returns <code>expNoWaiting</code>
    */
   @Override
   public double getExpNoWaiting()
   {
      double expNoWaiting = 0;

      for (int n = c; n <= k; n++) {
         expNoWaiting = expNoWaiting + (n - c) * getPn(n);
      }
      return expNoWaiting;

   }
   /**
 * This method calculates the value of the effective arrival rate.
 * @return Returns <code>lambdaeff</code>
 */
public double getLambdaeff()
{
   lambdaeff = (1 - getPn(k)) * lambda;
   return lambdaeff;
}

/**
 * This method calculates the value of the probability p0 of an empty system.
 * @return Returns <code>p0</code>
 */
public double getP0()
{

   double p0 = 0;
   double a = lambda / mu;
   double rho = lambda / (c * mu); //this is not the effective utilization of the system!
   double sum1 = 1;

   //sum a^n / n!
   for (int n = 1; n < c; n++) {
      // a^n / n!
      double prod = 1;
      for (int j = 0; j < n; j++) {
         prod = prod * (a / (n - j));
      }

      sum1 = sum1 + prod;
   }

   // a^c / c!
   double prod = 1;
   for (int j = 0; j < c; j++) {
      prod = prod * (a / (c - j));
   }

   // sum_{n=c}^{K} rho^(n-c)
   double sum2 = 0;
   for (int n = c; n <= k; n++) {
      sum2 = sum2 + Math.pow(rho, (n - c));
   }

   p0 = 1 / (sum1 + (prod * sum2));

   return p0;
}

/**
 * This method calculates the value of the probability of n jobs in system.
 * @return Returns <code>pn</code>
 */
public double getPn(int n)
{
   double pn = 0;
   double a = lambda / mu;
   double prod;
```

```
        if (n == 0) {
           pn = getP0();
        }

        if (1 <= n && n < c) {

           // a^n / n!
           prod = 1;
           for (int j = 0; j < n; j++) {
              prod = prod * (a / (n - j));
           }

           pn = prod * getP0();

        }

        if (c <= n && n <= k) {

           // a^c / c!
           prod = 1;
           for (int j = 0; j < c; j++) {
              prod = prod * (a / (c - j));
           }

           // a^(c-n) / c^(n-1)
           for (int j = 0; j < c - n; j++) {
              prod = prod * (a / c);
           }

           pn = ((prod * Math.pow(a, (n - c))) / Math.pow(c, (n - c))) * getP0();

        }

        return pn;
     }


     public double getProbBlock()
     {
        double probBlock = 0;

        probBlock = getPn(k);

        return probBlock;
     }
     public void resetQueue(double lambda, double mu, int c, int k)
     {
        this.lambda = lambda;
        this.mu = mu;
        this.c = c;
        this.k = k;
        this.lambdaeff = getLambdaeff();
     }
  }
  /*
   * To change this license header, choose License Headers in Project Properties.
   * To change this template file, choose Tools | Templates
   * and open the template in the editor.
   */
```

```java
package Mannheim.Queue;

/**
 *
 * @author Amir Foroughi <aforough at mail-mannheim.de>
 */
public class MMcK extends Queue
{
   public int k;

   public MMcK(double lambda, double mu, int c, int k)
   {
      this.lambda = lambda;
      this.mu = mu;
      this.c = c;
      this.k = k;
      this.lambdaeff = getLambdaeff();

   }
   /**
    * This method calculates the value of the expected number waiting.
    * @return Returns <code>expNoWaiting</code>
    */
   @Override
   public double getExpNoWaiting()
   {
      double expNoWaiting = 0;

      for (int n = c; n <= k; n++) {
         expNoWaiting = expNoWaiting + (n - c) * getPn(n);
      }
      return expNoWaiting;

   }
   /**
    * This method calculates the value of the effective arrival rate.
    * @return Returns <code>lambdaeff</code>
    */
   public double getLambdaeff()
   {
      lambdaeff = (1 - getPn(k)) * lambda;
      return lambdaeff;
   }

   /**
    * This method calculates the value of the probability p0 of an empty system.
    * @return Returns <code>p0</code>
    */
   public double getP0()
   {

      double p0 = 0;
      double a = lambda / mu;
      double rho = lambda / (c * mu); //this is not the effective utilization of the system!
      double sum1 = 1;

      //sum a^n / n!
      for (int n = 1; n < c; n++) {
         // a^n / n!
```

```
            double prod = 1;
            for (int j = 0; j < n; j++) {
               prod = prod * (a / (n - j));
            }

            sum1 = sum1 + prod;
         }

         // a^c / c!
         double prod = 1;
         for (int j = 0; j < c; j++) {
            prod = prod * (a / (c - j));
         }

         // sum_{n=c}^{K} rho^(n-c)
         double sum2 = 0;
         for (int n = c; n <= k; n++) {
            sum2 = sum2 + Math.pow(rho, (n - c));
         }

         p0 = 1 / (sum1 + (prod * sum2));

         return p0;
      }

      /**
       * This method calculates the value of the probability of n jobs in system.
       * @return Returns <code>pn</code>
       */
      public double getPn(int n)
      {
         double pn = 0;
         double a = lambda / mu;
         double prod;

         if (n == 0) {
            pn = getP0();
         }

         if (1 <= n && n < c) {

            // a^n / n!
            prod = 1;
            for (int j = 0; j < n; j++) {
               prod = prod * (a / (n - j));
            }

            pn = prod * getP0();

         }

         if (c <= n && n <= k) {

            // a^c / c!
            prod = 1;
            for (int j = 0; j < c; j++) {
               prod = prod * (a / (c - j));
            }
            // a^(c-n) / c^(n-1)
            for (int j = 0; j < c - n; j++) {
               prod = prod * (a / c);
            }

            pn = ((prod * Math.pow(a, (n - c))) / Math.pow(c, (n - c))) * getP0();

         }

         return pn;
      }


      public double getProbBlock()
      {
         double probBlock = 0;

         probBlock = getPn(k);

         return probBlock;
      }
      public void resetQueue(double lambda, double mu, int c, int k)
      {
         this.lambda = lambda;
         this.mu = mu;
         this.c = c;
         this.k = k;
         this.lambdaeff = getLambdaeff();
      }
   }
   /*
    * To change this license header, choose License Headers in Project Properties.
    * To change this template file, choose Tools | Templates
    * and open the template in the editor.
    */
   package Mannheim.Queue;

   /**
    *
    * @author Amir Foroughi <aforough at mail-mannheim.de>
    */
   public class MMc_over
   {
      private double omega;
      double wip;

      public MMc_over(double wip)
      {
         this.wip = wip;
      }
      /**
       * @param omega the omega to set
       */
      public void setOmega(double omega)
      {
         this.omega = omega;
      }

      public double getRho()
      {
         return (wip / (omega + wip));
      }

      public void resetQueue(double wip)
      {
         this.wip = wip;
      }
   }
   /*
    * To change this license header, choose License Headers in Project Properties.
    * To change this template file, choose Tools | Templates
    * and open the template in the editor.
    */
   package Mannheim.Approach;

   import Mannheim.Queue.MMc_over;

   /**
    *
    * @author Amir
    */
   public class PSFFA_M_M_c extends Approach
   {

      @Override
      public void run()
      {
         MMc_over mmc_over = new MMc_over(0.001);
         /**
          * OMEGA HERE.
          */
         mmc_over.setOmega(1);

         //constant mu is assumed
         double mu = instance.ser_rate;
         double lambda;
         int c;
         int k;
         double timeStep = 1.0 / mu;
         //the time where the last phase ends
         double endTime = instance.pla_hor;
         //the number of time steps to initialize the arrays with the correct length
         int nrTimeSteps = (int) (Math.ceil(endTime / timeStep));
         //the current time considered in the loop
         double currentTime = 0;

         double x_0 = 0;
         double x = 0;
         double y = 0;
         double timeStep_psfa = 0.0001;
         double util = 0;
         int nrTimeStepsPSFA = (int) Math.ceil(timeStep / timeStep_psfa);

         //the arrays are initialized
         solution.utilization = new double[nrTimeSteps];
         solution.expWaitingTime = new double[nrTimeSteps];
         solution.expCycleTime = new double[nrTimeSteps];
         solution.expNoWaiting = new double[nrTimeSteps];
```

```
        solution.expNoInSystem = new double[nrTimeSteps];
        solution.time = new double[nrTimeSteps];

        //Loop: for each station and time the performance values are calculated
        for (int t = 0; t < solution.time.length; t++) {

           //the current time is saved
           if (Math.ceil(currentTime) - currentTime <= 0.0001) {
              solution.time[t] = Math.ceil(currentTime);
           } else {
              solution.time[t] = currentTime;
           }

           //time bigger then the end time of the last phase should not be considered.
           //Therefore we set the last time considered as end time. After this run the loop
               will stop.
           if (solution.time[t] > endTime) {
              solution.time[t] = endTime;
           }

           //updating the parameters
           int currentPhase = (int) Math.min(Math.floor(solution.time[t] /
               instance.len_period), instance.arr_rate.length - 1);

           lambda = instance.arr_rate[currentPhase];
           c = instance.ser_num[currentPhase];
           // the max queuing length is equal then the server number because we use the mmcc
               model
           k = c;

           //using stationary MMcc (phase 1)
           if (t == 0) {
              x_0 = lambda / (mu * c);
              mmc_over.resetQueue(x_0);
              double rho = 0;
              for (int i = 0; i < nrTimeStepsPSFA; i++) {
                 mmc_over.resetQueue(x_0);
                 rho = mmc_over.getRho();
                 x = x_0 + (lambda - mu * rho * c) * timeStep_psfa;
                 x_0 = x;

              }
           } else {
              mmc_over.resetQueue(x_0);
              double rho = 0;
              for (int i = 0; i < nrTimeStepsPSFA; i++) {
                 mmc_over.resetQueue(x_0);
                 rho = mmc_over.getRho();
                 x = x_0 + (lambda - mu * rho * c) * timeStep_psfa;
                 x_0 = x;
                 y= x/mu;

              }

           }
           solution.expNoInSystem[t] = x;
           solution.expWaitingTime[t] = y;

//                      System.out.println("t: " + t);
  //                      System.out.println("time: " + solution.time[t]);
  //                      System.out.println("artificialLambda " + artificialLambda);
  //                      System.out.println("modifiedLambda " + modifiedLambda);
  //                      System.out.println("backlogRate(in t-1): " + backlogRate);
  //                      System.out.println("WIP:" + solution.expNoInSystem[t]);
  //
      System.out.println("________________________________________________________________________________________
           currentTime = currentTime + timeStep;
           /**
            * Normal PSFA.
            */

        }
     }

  }
  /*
   * To change this license header, choose License Headers in Project Properties.
   * To change this template file, choose Tools | Templates
   * and open the template in the editor.
   */
  package Mannheim.Queue;

  /**
   *
   * @author Amir Foroughi <aforough at mail-mannheim.de>
   */
  abstract public class Queue
  {
     public int c;
     public double lambda;
     public double lambdaeff;
     public double mu;
     /**
      * This method calculates the value of the expected cycle time.
      *
      * @return Returns <code>expCycleTime</code>
      */
     public double getExpCycleTime()
     {
        return (getExpNoWaiting() / lambdaeff) + (1 / mu);
        // return getWaitingTime()+1/mu;
     }


     /**
      * This method calculates the value of the expected number in system.(WIP)
      *
      * @return Returns <code>expNoInSystem</code>
      */
     public double getExpNoInSystem()
     {
        return lambdaeff * getExpCycleTime();
     }


     /**
      * This method calculates the value of the expected number waiting.
      *
```

```
     * @return Returns <code>expNoWaiting</code>
     */
    public double getExpNoWaiting()
    {
       double expNoWaiting = 0;
       expNoWaiting = lambdaeff * getExpWaitingTime();
       return expNoWaiting;
    }

    /**
     * This method calculates the value of the expected waiting time.
     *
     * @return Returns <code>waitingTime</code>
     */
    public double getExpWaitingTime()
    {
       return (getExpNoInSystem() / lambdaeff) - (1 / mu);
       // return getExpNoWaiting()/lambdaeff;
    }
    /**
     * This method calculates the value of utilization.
     *
     * @return Returns <code>utilization</code>
     */
    public double getTrafficIntensity()
    {
       return lambda / (c * mu);
    }
    /**
     * This method calculates the value of utilization.
     *
     * @return Returns <code>utilization</code>
     */
    public double getUtilization()
    {
       return lambdaeff / (c * mu);
    }


}
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package Mannheim.Approach;

import Mannheim.Instance.Instance;
import Mannheim.Queue.*;
import Mannheim.core.Solution;

/**
 *
 * @author Amir
 */
public class SBC_M_M_c extends Approach
{

    /**
     * The current phase
     */
    @Override
    public void run()
    {
       //The mmcc is used to calculate the expected utilization ( k = c)
       MMcc mmcc = new MMcc(instance.arr_rate[0], instance.ser_rate, instance.ser_num[0],
           instance.ser_num[0]);
       //The mmc is used to calculate the expected waiting time
       MMc mmc = new MMc(instance.arr_rate[0], instance.ser_rate, instance.ser_num[0]);
       double backlogRate = 0;
       double artificialLambda = 0;
       double modifiedLambda = 0;
       //constant mu is assumed
       double mu = instance.ser_rate;
       double lambda;
       int c;
       int k;
       double timeStep = 1.0 / mu;
       //the time where the last phase ends
       double endTime = instance.pla_hor;
       //the number of time steps to initialize the arrays with the correct length
       int nrTimeSteps = (int) (Math.ceil(endTime / timeStep));
       //the current time considered in the loop
       double currentTime = 0;

       //the arrays are initialized
       solution.utilization = new double[nrTimeSteps ];
       solution.expWaitingTime = new double[nrTimeSteps ];
       solution.expCycleTime = new double[nrTimeSteps ];
       solution.expNoWaiting = new double[nrTimeSteps ];
       solution.expNoInSystem = new double[nrTimeSteps ];
       solution.time = new double[nrTimeSteps ];


       //Loop: for each station and time the performance values are calculated
       for (int t = 0; t < solution.time.length; t++) {

          //the current time is saved
          if (Math.ceil(currentTime) - currentTime <= 0.0001) {
             solution.time[t] = Math.ceil(currentTime);
          } else {
             solution.time[t] = currentTime;
          }

          //time bigger then the end time of the last phase should not be considered.
          //Therefore we set the last time considered as end time. After this run the loop
              will stop.
          if (solution.time[t] > endTime) {
             solution.time[t] = endTime;
          }

          //updating the parameters
          int currentPhase = (int) Math.min(Math.floor(solution.time[t] /
              instance.len_period), instance.arr_rate.length - 1);

          lambda = instance.arr_rate[currentPhase];
          c = instance.ser_num[currentPhase];
```

```
            // the max queuing length is equal then the server number because we use the mmcc
                model
            k = c;

            //using stationary MMcc (phase 1)
            artificialLambda = lambda + backlogRate;
            mmcc.resetQueue(artificialLambda, mu, c, c);
            solution.utilization[t] = artificialLambda * (1 - mmcc.getPn(k)) / (c * mu);

            //using stationary MMc (phase 2)
            modifiedLambda = solution.utilization[t] * k * mu;
            mmc.resetQueue(modifiedLambda, mu, c);

//          System.out.println("artificialLambda " + artificialLambda);
//          System.out.println("modifiedLambda " + modifiedLambda);
//          System.out.println("backlogRate(in t-1): " + backlogRate);
//          System.out.println("pn " + mmck.getPn(k));
//          System.out.println("utilization " + utilization[noElements][t] + " " +
    mmck.getUtilization());
//          System.out.println("k "+k );
//
    System.out.println("________________________________________________________________________________
            //saving the results of the calculation in two dimensional arrays
            solution.expWaitingTime[t] = mmc.getExpWaitingTime();
            solution.expCycleTime[t] = mmc.getExpCycleTime();
            solution.expNoWaiting[t] = mmc.getExpNoWaiting();
            solution.expNoInSystem[t] = mmc.getExpNoInSystem();

            //updating the backlog rate
            backlogRate = artificialLambda * mmcc.getPn(k);

            //updating the current time
            currentTime = currentTime + timeStep;
        }

    }
}
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package Mannheim.Instance;

import java.io.File;

/**
 *
 * @author Amir
 */
public class SimpleLoader
{
    /**
     * @param path
     * @return
     */
    public static Instance Load(String filename)
    {
        Instance instance = new Instance();
```

```
   try {
      initializeMapClassInstance(filename);
      instance.name = filename.substring(filename.lastIndexOf(File.separatorChar) + 1,
          filename.lastIndexOf('.'));

      instance.queue = MapClassInstance.getString("Queue");

      instance.num_period = MapClassInstance.getInt("period");
      instance.len_period = MapClassInstance.getDouble("period.length");

      /**
       * Arrival.
       */
      instance.arr_rate = new double[instance.num_period];
      String[] st = MapClassInstance.getArray("arr.rate");
      String[] st2 = MapClassInstance.getArray("arr.cv2");

      for (int i = 0; i < st.length; i++) {
         instance.arr_rate[i] = Double.valueOf(st[i]);
      }

      if (instance.queue.equalsIgnoreCase("M_M_1_batch")) {
         instance.batch_mean = new double[instance.num_period];
         instance.distance_batch = new double[instance.num_period];
         st = MapClassInstance.getArray("arr.num");
         st2 = MapClassInstance.getArray("batch.dist");
         for (int i = 0; i < instance.batch_mean.length; i++) {
            instance.batch_mean[i] = Double.valueOf(st[i]);
            instance.distance_batch[i] = Double.valueOf(st2[i]);
         }
         instance.batch_dist = MapClassInstance.getString("dist.batch");
      }
      instance.arr_dist = MapClassInstance.getString("dist.arr");
      instance.ser_dist = MapClassInstance.getString("dist.ser");

      /**
       * Service.
       */
      instance.ser_rate = MapClassInstance.getDouble("ser.rate");
      st = MapClassInstance.getArray("ser.num");
      instance.ser_num = new int[st.length];
      for (int i = 0; i < st.length; i++) {
         instance.ser_num[i] = Integer.valueOf(st[i]);
      }


   } catch (Exception e) {
      e.printStackTrace();
      System.err.println("Problems in instances!");
      return null;
   }
   instance.makeInstance();
   return instance;

}

private static void initializeMapClassInstance(String propertiesPath)
{
   MapClassInstance mapClassInstance = MapClassInstance.getInstance();
```

```
      mapClassInstance.init(propertiesPath);
      if (!mapClassInstance.hasProperty("threads")) {
         mapClassInstance.set("threads", Runtime.getRuntime().availableProcessors());
      }
   }

}
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package Mannheim.core;

import Mannheim.Instance.Instance;

/**
 *
 * @author Amir
 */
public class Solution
{
   private Instance instance;
   public double[] expCycleTime;
   public double[] expNoInSystem;
   public double[] expNoWaiting;
   public double[] expWaitingTime;
   public double[] time;
   //the arrays are initialized
   public double[] utilization;

   public Solution(Instance instance)
   {
      setInstance(instance);
   }

   /**
    * @return the instance
    */
   public Instance getInstance()
   {
      return instance;
   }

   /**
    * @param instance the instance to set
    */
   public void setInstance(Instance instance)
   {
      this.instance = instance;
   }

}
```

Bibliography

Agnew, C. E. (1976). Dynamic modeling and control of congestion-prone systems. *Operations Research 24*(3), 400–419.

Ahmadi-Javid, A., Z. Jalali, and K. J. Klassen (2017). Outpatient appointment systems in healthcare: A review of optimization studies. *European Journal of Operational Research 258*(1), 3–34.

Alfa, A. S. (1990). Approximating queue lengths in M (t)/D/1 queues. *European Journal of Operational Research 44*(1), 60–66.

Alfa, A. S. and M. Chen (1991). Approximating queue lengths in M (t)/G/1 queue using the maximum entropy principle. *Acta Informatica 28*(8), 801–815.

Altman, E., T. Jiménez, and G. Koole (2001). On the comparison of queueing systems with their fluid limits. *Probability in the Engineering and Informational Sciences 15*(2), 165.

Amaran, S., N. V. Sahinidis, B. Sharda, and S. J. Bury (2016). Simulation optimization: a review of algorithms and applications. *Annals of Operations Research 240*(1), 351–380.

Azadeh, K., R. De Koster, and D. Roy (2019). Robotized and automated warehouse systems: Review and recent developments. *Transportation Science 53*(4), 917–945.

Bartholdi, J. J. and S. T. Hackman (2008). Allocating space in a forward pick area of a distribution center for small parts. *IIE Transactions 40*(18), 1046–1053.

Bartholdi, J. J. and S. T. Hackman (2017). Warehouse & distribution science: Release 0.98. Retrieved from: https://www2.isye.gatech.edu/~jjb/wh/book/index.html.

Begen, M. A., R. Levi, and M. Queyranne (2012). A sampling-based approach to appointment scheduling. *Operations Research 60*(3), 675–681.

Bhattacharjee, P. and P. K. Ray (2016). Hospital appointment scheduling in presence of walk-ins and emergency arrivals. In *Smart Technologies for Smart Nations*, pp. 175–194. Springer.

Borgman, N. J., I. M. Vliegen, R. J. Boucherie, and E. W. Hans (2018). Appointment scheduling with unscheduled arrivals and reprioritization. *Flexible Services and Manufacturing Journal 30*(1-2), 30–53.

Boysen, N., D. Briskorn, and S. Emde (2017). Sequencing of picking orders in mobile rack warehouses. *European Journal of Operational Research 259*(1), 293–307.

Boysen, N., R. de Koster, and F. Weidinger (2019). Warehousing in the e-commerce era: A survey. *European Journal of Operational Research 277*(2), 396–411.

Boysen, N. and K. Stephan (2013). The deterministic product location problem under a pick-by-order policy. *Discrete Applied Mathematics 61*(18), 2862–2875.

Brahimi, M. and D. Worthington (1991). Queueing models for out-patient appointment system–A case study. *Journal of the Operational Research Society 42*(9), 733–746.

Brynzér, H. and M. I. Johansson (1996). Storage location assignment: using the product structure to reduce order picking times. *International Journal of Production Economics 46*, 595–603.

Catling, I. (1977). A time-dependent approach to junction delays. *Traffic Engineering & Control 18*(11), 520–526.

Cayirli, T. and E. Veral (2003). Outpatient scheduling in health care: a review of literature. *Production and Operations Management 12*(4), 519–549.

Chan, F. T. S. and H. K. Chan (2011). Improving the productivity of order picking of a manual-pick and multi-level rack distribution warehouse through the implementation of class-based storage. *Expert Systems with Applications 38*(3), 2686–2700.

Chang, T. H., H. P. Fu, and K. Y. Hu (2007). A two-sided picking model of m-as/rs with an aisle-assignment algorithm. *International Journal of Production Research 45*(17), 3971–3990.

Chen, L., A. Langevin, and D. Riopel (2010). The storage location assignment and interleaving problem in an automated storage/retrieval system with shared storage. *International Journal of Production Research 48*(4), 991–1011.

Choi, S. O., H. D. Park, Y. J. Park, H. Y. Kim, and H. D. Jang (2000). Test running of an underground food storage cavern in korea. *Tunnelling and Underground Space Technology 15*(1), 91–95.

Cordeau, J.-F., G. Laporte, and A. Mercier (2001). A unified tabu search heuristic for vehicle routing problems with time windows. *Journal of the Operational research society 52*(8), 928–936.

Daniels, R. L., J. L. Rummel, and R. Schantz (1998). A model for warehouse order picking. *European Journal of Operational Research 105*(1), 1–17.

De Koster, R., T. Le-Duc, and K. J. Roodbergen (2007). Design and control of warehouse order picking: A literature review. *European Journal of Operational Research 182*(2), 481–501.

de Koster, R., T. Le-Duc, and K. J. Roodbergen (2007). Design and control of warehouse order picking: A literature review. *European Journal of Operational Research 182*(2), 481–501.

Eick, S. G., W. A. Massey, and W. Whitt (1993). Mt/G/∞ queues with sinusoidal arrival rates. *Management Science 39*(2), 241–252.

Filipiak, J. (1984). Dynamic routing in a queueing system with a multiple service facility. *Operations Research 32*(5), 1163–1180.

Frazelle, E. A. and G. P. Sharp (1989). Correlated assignment strategy can improve order-picking operation. *Industrial Engineering 4*(9), 33–37.

Froehle, C. M. and M. J. Magazine (2013). Improving scheduling and flow in complex outpatient clinics. In *Handbook of healthcare operations management*, pp. 229–250. Springer.

Garey, M. R. and D. S. Johnson (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: Freeman.

Glover, F. (1986). Future paths for integer programming and links to ar tifi cial intelli g en ce. *Computers operations research 13*(5), 533–549.

Gocgun, Y., B. W. Bresnahan, A. Ghate, and M. L. Gunn (2011). A markov decision process approach to multi-category patient scheduling in a diagnostic facility. *Artificial Intelligence in Medicine 53*(2), 73–81.

Goetschalckx, M. and H. D. Ratliff (1990). Shared storage policies based on the duration stay of unit loads. *Management Science 36*(9), 1120–1132.

Green, L. and P. Kolesar (1991). The pointwise stationary approximation for queues with nonstationary arrivals. *Management Science 37*(1), 84–97.

Green, L., P. Kolesar, and A. Svoronos (1991). Some effects of nonstationarity on multiserver markovian queueing systems. *Operations Research 39*(3), 502–511.

Green, L. V. and P. J. Kolesar (1995). On the accuracy of the simple peak hour approximation for markovian queues. *Management science 41*(8), 1353–1370.

Green, L. V. and P. J. Kolesar (1998). A note on approximating peak congestion in Mt/G/∞ queues with sinusoidal arrivals. *Management Science 44*(11), 137–144.

Green, L. V., S. Savin, and B. Wang (2006). Managing patient service in a diagnostic medical facility. *Operations Research 54*(1), 11–25.

Gross, D., J. F. Shortle, J. M. Thompson, and C. M. Harris (2008). *Fundamentals of queueing theory*, Volume 4. John Wiley & Sons, NY, United States.

Gu, J., M. Goetschalckx, and L. F. McGinnis (2007). Research on warehouse operation: A comprehensive review. *European Journal of Operational Research 177*(1), 1–21.

Gu, J., M. Goetschalckx, and L. F. McGinnis (2010). Research on warehouse design and performance evaluation: A comprehensive review. *European Journal of Operational Research 203*(3), 539–549.

Gue, K. R. and B. S. Kim (2007). Puzzle-based storage systems. *Naval Research Logistics 54*(5), 556–567.

Gupta, D. and B. Denton (2008). Appointment scheduling in health care: Challenges and opportunities. *IIE transactions 40*(9), 800–819.

Heizer, J., B. Render, and C. Munson (2017). *Operations management: sustainability and supply chain management*, Volume 12. New Jersey, USA: Pearson Education.

Heskett, J. L. (1963). Cube-per-order index - a key to warehouse stock location. *Transport and Distribution Management 3*(1), 27–31.

Hong, S., A. L. Johnson, and B. A. Peters (2012). Batch picking in narrow-aisle order picking systems with consideration for picker blocking. *European Journal of Operational Research 221*(3), 557–570.

Hu, K. Y., T. H. Chang, H. P. Fu, and H. Yeh (2009). Improvement order picking in mobile storage systems with a middle cross aisle. *International Journal of Production Research 47*(4), 1089–1104.

Ingolfsson, A., E. Akhmetshina, S. Budge, Y. Li, and X. Wu (2007). A survey and experimental comparison of service-level-approximation methods for nonstationary M(t)/M/s(t) queueing systems with exhaustive discipline. *INFORMS Journal on Computing 19*(2), 201–214.

Jaiswal, N. K. (1968). *Priority queues*, Volume 50. Academic Press.

Jiménez, T. and G. Koole (2004). Scaling and comparison of fluid limits of queues applied to call centers with time-varying parameters. *OR Spectrum 26*(3), 413–422.

Jung, M. and E. Lee (1989). A multi-echelon and multi-indenture repairable item queuing model during emergencies. *Mathematical and Computer Modelling 12*(7), 851–864.

Kaandorp, G. C. and G. Koole (2007). Optimal outpatient appointment scheduling. *Health Care Management Science 10*(3), 217–229.

Kimber, R. and P. Daly (1986). Time-dependent queueing at road junctions: observation and prediction. *Transportation Research Part B: Methodological 20*(3), 187–203.

Kimber, R. and E. Hollis (1977). Flow/delay relationships for major/minor priority junctions. *Traffic Engineering & Control 18*(11), 516–519.

Klassen, K. J. and R. Yoogalingam (2009). Improving performance in outpatient appointment services with a simulation optimization approach. *Production and Operations Management 18*(4), 447–458.

Klassen, K. J. and R. Yoogalingam (2019). Appointment scheduling in multi-stage outpatient clinics. *Health Care Management Science 22*(2), 229–244.

Kleinrock, L. (1975). *Queueing systems: theory*, Volume 1. John Wiley, NY, United States.

Koeleman, P. M. and G. M. Koole (2012). Optimal outpatient appointment scheduling with emergency arrivals and general service times. *IIE Transactions on Healthcare Systems Engineering 2*(1), 14–30.

Kolisch, R. and S. Sickinger (2008). Providing radiology health care services to stochastic demand of different customer classes. *OR spectrum 30*(2), 375–395.

Kolmogrov, A. (1931). Sur le problème d'attente. *Matematicheskii Sbornik 38*(1-2), 101–106. (in French).

Kortbeek, N., M. E. Zonderland, A. Braaksma, I. M. Vliegen, R. J. Boucherie, N. Litvak, and E. W. Hans (2014). Designing cyclic appointment schedules for outpatient clinics with scheduled and unscheduled patient arrivals. *Performance Evaluation 80*, 5–26.

Kress, D., N. Boysen, and E. Pesch (2017). Which items should be stored together? a basic partition problem to assign storage space in group-based storage systems. *IIE Transactions 49*(1), 13–30.

Kulturel, S., N. E. Ozdemirel, C. Sepil, and Z. Bozkurt (1999). Experimental investigation of shared storage assignment policies in automated storage/retrieval systems. *IIE Transactions 31*(8), 739–749.

Lakshmi, C. and S. A. Iyer (2013). Application of queueing theory in health care: A literature review. *Operations Research for Health Care 2*(1-2), 25–39.

Lan, G., G. W. DePuy, and G. E. Whitehouse (2007). An effective and simple heuristic for the set covering problem. *European Journal of Operational Research 176*(3), 1387–1403.

Lin, Z., N. Frigerio, A. Matta, and S. Du (2020). Multi-fidelity surrogate-based optimization for decomposed buffer allocation problems. *OR Spectrum*, 1–31.

Lin, Z., A. Matta, and J. G. Shanthikumar (2019). Combining simulation experiments and analytical models with area-based accuracy for performance evaluation of manufacturing systems. *IISE Transactions 51*(3), 266–283.

Liu, N. and S. Ziya (2014). Panel size and overbooking decisions for appointment-based services under patient no-shows. *Production and Operations Management 23*(12), 2209–2223.

Lowerre, B. (1976). *The Harpy Speech Recognition System*. Ph. D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA.

Luo, J., V. G. Kulkarni, and S. Ziya (2012). Appointment scheduling under patient no-shows and service interruptions. *Manufacturing & Service Operations Management 14*(4), 670–684.

Mandelbaum, A., W. A. Massey, and M. I. Reiman (1998). Strong approximations for markovian service networks. *Queueing Systems 30*(1), 149–201.

Material Handling Industry of America (MHIA) (2009). Automated storage systems make a play for sustainability. Retrieved from http://www.mhia.org/news/industry/9141/as-rs-industry-group-releases-fall-2009-quarterly-report.

Matusiak, M., R. de Koster, L. Kroon, and J. Saarinen (2014). A fast simulated annealing method for batching precedence-constrained customer orders in a warehouse. *European Journal of Operational Research 236*(3), 968–977.

Moore, S. C. (1975). Approximating the Behavior of Nonstationary Single-Server Queues. *Operations Research*.

Newell, C. (1971). *Applications of queueing theory*, Volume 1. Chapman and Hall, London.

Peng, Y., X. Qu, and J. Shi (2014). A hybrid simulation and genetic algorithm approach to determine the optimal scheduling templates for open access clinics admitting walk-in patients. *Computers & Industrial Engineering 72*, 282–296.

Potvin, J.-Y., T. Kervahut, B.-L. Garcia, and J.-M. Rousseau (1996). The vehicle routing problem with time windows part i: tabu search. *INFORMS Journal on Computing 8*(2), 158–164.

Renaud, J. and A. Ruiz (2008). Improving product location and order picking activities in a distribution centre. *Journal of the Operations Research Society 59*(12), 1603–1613.

Rider, K. L. (1976). A simple approximation to the average queue size in the time-dependent M/M/1 queue. *Journal of the ACM (JACM) 23*(2), 361–367.

Rising, E. J., R. Baron, and B. Averill (1973). A systems analysis of a university-health-service outpatient clinic. *Operations Research 21*(5), 1030–1047.

Russon, D., V. Kútik, and A. Clarke (1982). Future storage requirements at the british library lending division: Fixed or mobile shelving? *Interlending & Document Supply 10*, 54–58.

Schäfer, S. (2018). Mobile pallet racks. Retrieved from https://www.ssi-schaefer.com/en-ca/products/storage/pallet-rack-systems/mobile-pallet-racks-58156.

Schneider, M., F. Schwahn, and D. Vigo (2017). Designing granular solution methods for routing problems with time windows. *European Journal of Operational Research 263*(2), 493–509.

Schwarz, J. A., G. Selinka, and R. Stolletz (2016). Performance analysis of time-dependent queueing systems: survey and classification. *Omega 63*, 170–189.

Selinka, G., A. Franz, and R. Stolletz (2016). Time-dependent performance approximation of truck handling operations at an air cargo terminal. *Computers & Operations Research 65*, 164–173.

Stadtler, H. (1996). An operational planning concept for deep lane storage systems. *Production and Operations Management 5*(3), 266–282.

Stidham Jr, S. (2002). Analysis, design, and control of queueing systems. *Operations Research 50*(1), 197–216.

Stolletz, R. (2008a). Approximation of the non-stationary M(t)/M(t)/c(t)-queue using stationary queueing models: The stationary backlog-carryover approach. *European Journal of Operational Research 190*(2), 478–493.

Stolletz, R. (2008b). Non-stationary delay analysis of runway systems. *OR Spectrum 30*(1), 191–213.

Stolletz, R. (2011). Analysis of passenger queues at airport terminals. *Research in Transportation Business & Management 1*(1), 144–149.

Stolletz, R. and S. Lagershausen (2013). Time-dependent performance evaluation for loss-waiting queues with arbitrary distributions. *International Journal of Production Research 51*(5), 1366–1378.

Theys, C., O. Bräysy, W. Dullaert, and B. Raa (2010). Using a tsp heuristic for routing order pickers in warehouses. *European Journal of Operational Research 200*(3), 755–763.

Toth, P. and D. Vigo (2003). The granular tabu search and its application to the vehicle-routing problem. *Informs Journal on Computing 15*(4), 333–346.

van Gils, T., K. Ramaekers, A. Caris, and R. B. de Koster (2018). Designing efficient order picking systems by combining planning problems: State-of-the-art classification and review. *European Journal of Operational Research 267*(1), 1–15.

van Oudheusden, D. L. and W. Zhu (1992). Storage layout of as/rs racks based on recurrent orders. *European Journal of Operational Research 58*(1), 48–56.

Wang, W.-P., D. Tipper, and S. Banerjee (1996). A simple approximation for modeling nonstationary queues. In *Proceedings of IEEE INFOCOM'96. Conference on Computer Communications*, Volume 1, pp. 255–262. IEEE.

Wäscher, G. (2004). Order picking: a survey of planning problems and methods. In *Supply Chain Management and Reverse Logistics*, pp. 323–347. Springer.

Weidinger, F. and N. Boysen (2018). Scattered storage: How to distribute stock keeping units all around a mixed-shelves warehouse. *Transportation Science 52*(6), 1412–1427.

Weidinger, F., N. Boysen, and M. Schneider (2019). Picker routing in the mixed-shelves warehouses of e-commerce retailers. *European Journal of Operational Research 274*(2), 501–515.

Whitt, W. (2014). The steady-state distribution of the Mt/M/∞ queue with a sinusoidal arrival rate function. *Operations Research Letters 42*(5), 311–318.

Xiao, Y., I. Kaku, Q. Zhao, and R. Zhang (2011). A reduced variable neighborhood search algorithm for uncapacitated multilevel lot-sizing problems. *European Journal of Operational Research 214*(2), 223–231.

Xu, J., S. Zhang, E. Huang, C.-H. Chen, L. H. Lee, and N. Celik (2014). An ordinal transformation framework for multi-fidelity simulation optimization. In *2014 IEEE International Conference on Automation Science and Engineering (CASE)*, pp. 385–390. IEEE.

Yang, K. K., M. L. Lau, and S. A. Quek (1998). A new appointment rule for a single-server, multiple-customer service system. *Naval Research Logistics (NRL) 45*(3), 313–326.

Zaerpour, N., Y. Yu, and R. B. de Koster (2015). Small is beautiful: A framework for evaluating and optimizing live-cube compact storage systems. *Transportation Science 51*(1), 34–51.

Curriculum vitae

Personal Information

Name	Amir Foroughi

Education

2017—2022	Doctoral Candidate Business School, University of Mannheim, Germany
2015—2017	External PhD student School of Business, Law and Economics, Technical University of Darmstadt
2011—2013	Industrial Engineering: Socio-Economic Systems Engineering (M. Sc.) Sharif University of Technology, Iran
2006—2011	Railway Operation and Management Engineering (B. Sc.) Iran University of Science and Technology, Iran

Professional Experience

2017—2021	Research Assistant Chair of Production Management, Business School, University of Mannheim
2015—2017	Research Assistant Management Science and Operations Research, Logistics Planning & Information Systems department School of Business, Law and Economics, Technical University of Darmstadt
2010—2012	Research Assistant Research Laboratory of Intelligent Computations in Rail Transportation (ICRT), Iran University of Science and Technology, Iran