Barge processing policies in container terminals: A multi-agent simulation study

Author
Sabri Bouzidi
307228

Supervisor
Dr. Yingqian Zhang
Co-reader
Qing Chuan Ye

MSc Thesis
Economics and Informatics
Computational Economics
2014

ERASMUS UNIVERSITEIT ROTTERDAM
ERASMUS SCHOOL OF ECONOMICS

Barge processing policies in container terminals:

A multi-agent simulation study

Student: Sabri Bouzidi, 307228

Supervisor: Dr. Yingqian Zhang

Co-reader: Qing Chuan Ye

MSc Thesis

Economics and Informatics

Computational Economics

Erasmus University Rotterdam

August 1, 2014

Abstract

In this thesis, we develop a simulation program which can be used to evaluate policies in a port. We use this program to examine the use of a multi-agent system for making appointments between barge operators and container terminal operators. In an existing model, terminals reserve their quay cranes for specific barges at specific times based on appointments. Since it is hard to predict the arrival times of barges at terminals, it often occurs that terminals keep their quay cranes reserved, while it could already handle other barges. We suggest that this results in more total waiting time for the whole system. Therefore, we propose an unreserved policy, where appointments are still made, but are less strict than with the reserved policy. The simulation experiments show that using an unreserved policy results in significant lower average sojourn times in the port, especially in busy ports. The explanation for the lower sojourn times is that the terminals are less idle, which makes the throughput higher. In addition, we examine the implications for individual barges by measuring their satisfaction with the waiting time and the provided information. The results suggest that barges are more satisfied with the waiting time in case of using an unreserved policy, and that barges are only somewhat satisfied with the provided information for both policies, since the predicted sojourn times in the port lack precision.

Keywords: Multi-agent simulation, Barge, Terminal, Scheduling, TDTSP

Contents

Chapter 1

Introduction

1.1 Relevance and motivation

Over the last decade container port traffic has tripled from 200 million TEU to 600 million TEU per year [24]. Such an immense growth forces terminal and ship operators to think more about efficiency and costs savings. In the field of operations research a lot of diverse studies are conducted that investigate ways to improve efficiency and save costs [22].

Mathematical programming models are used mainly for scheduling and allocating resources, where simulation models are used to evaluate various policies and control rules. Most studies introduce methods designed as optimization tool for terminal logistics, including:

- The ship planning process, for example berth allocation, stowage planning and quay crane handling;
- Storage and stacking logistics, for example container stacking strategies;
- Transport optimization, for example quayside transport routing and fleet of trucks scheduling;
- Integrative approaches that argue that improved terminal performance can only be obtained by solving various operations connected to each other,

for example cooperative scheduling of quay cranes and container vehicles. There are approaches that are analytical, use simulation, and ones that are based on multi-agent systems (i.e., distributed artificial intelligence).

One of the logistic problems concerns the handling of barges by container terminals in ports with high demanding hinterland regions, such as the Port of Rotterdam. Container ships that carry 1,000 to 18,000 TEU are too big to go through a river or canal, therefore they use a port nearby the hinterland as a hub. The container ship unloads containers at the hub port, from there trucks, trains and barges take over. In this thesis the focus is on the barges. Barges are defined as flat-bottomed boats used for transport of heavy goods and containers through a river or canal. They have carrying capacities of a few dozen TEU. The facility where the cargo containers are transshipped is referred to as container terminal. The port's hinterland is the area that it serves for import and export. Transport by barges is also called inland shipping.

Figure 1.1: A barge is loading containers at a container terminal in the Port of Rotterdam

The use of barges to transport containers is becoming more important, because it replaces many trucks on the road, it is cost efficient, and it is also a green way to transport containers (i.e., better for the environment). Ports encourage the use of

barges, including the Port of Rotterdam. Partly as a result of that, the expectation is that the use of barges will increase. However, the process of transporting containers by barges is not yet optimal, since barges often spend more time in the port than necessary.

1.2 Problem description

When a barge enters a port, it often needs to visit several terminals. This involves making appointments between terminals and barges. A port often contains multiple terminals owned by different companies, which do not share their schedules. The same applies to the barge companies. In the current situation appointments between terminal and barge operators are made manually. Whether it is by phone, email, or more advanced computer programs such as Portbase [20]. The result is a loss in efficiency for both barges and terminals [11].

In this thesis, we study the use of a multi-agent system, which can assist in making appointments between barge operators and terminal operators. In the system, the computer agents of the terminal operators provide the computer agents of the barge operators with information about the waiting times in the form of a waiting profile. The agents of the terminals add slack to the waiting profiles to increase planning flexibility. Using the waiting profiles, expected sailing times between terminals, and expected handling times at terminals, the agent of the barge operator solves a time dependent traveling salesman problem, which returns the route in the port with the least expected sojourn time. Based on the route, the agent of the barge operator makes appointments with the agents of the terminal operators. A barge operator uses the appointments to determine in which sequence it has to visit the terminals. For the terminals, the consequence of an appointment is that it has to reserve its quay cranes to handle a barge with which it has an appointment at specified times. Since it is hard to predict the exact arrival times of barge at terminals, it happens that terminals keep their quay crane reserved, while there are other barges

already present, which in turn have to wait for their appointment. In our view, this leads to more total waiting time for all barges. We investigate the effect of a less strict reservation policy of the quay cranes, which we will refer to as an unreserved policy.

In addition, we link the service quality (i.e., acceptability of the waiting time and correctness of the information provision) of these policies with the customer satisfaction. We consider the barge operators as customers of the port. If the customer satisfaction is low, then the customer will look for other service providers in the future [9]. In this case, that would mean that barge operators will look for other ports. Therefore, it is essential for ports to keep the visiting barges satisfied with the service. The goal of including the customer satisfaction is to get insight in the implications for individual barges, instead of measuring only averages.

1.3 Research question and structure of the thesis

This thesis investigates the impact of the terminal queue processing policy and slack policy on the sojourn time of barges in the port. The objective of the queue processing policy is to influence the idle time of terminals. The objective of the slack policy is to create more planning flexibility in the schedule of terminal operators, while also influencing the route of barges in the port by increasing the promised expected waiting time. The research questions are formulated as follows:

1. What is the impact of the terminal queue processing policy and slack policy on the sojourn time of barges in the port?

2. What is the impact of the terminal queue processing policy and slack policy on predictive power of the sojourn time of barges in the port?

3. What is the impact of the terminal queue processing policy and slack policy on the customer satisfaction?

To tackle these research questions, we first develop a software program which can be used to evaluate policies that involve multi-agent systems in a port. We use an object-oriented approach in building the software, also known as object-oriented programming (OOP). This makes it possible to understand and extend the program with relatively little effort. We include the software design choices, documentation and source code in the thesis.

Second, we conduct several simulation experiments to evaluate the influence of the policies on the sojourn time of barges in the port and the correctness of the expected sojourn time in the port (i.e., predictive power). The first policy concerns the way a terminal operator handles barges that arrive at their terminal. We evaluate an unreserved policy relative to a reserved policy. The second policy concerns the amount of slack added to the waiting profiles.

Finally, we assess the barge operator (or customer) satisfaction with the policies. We develop two satisfaction measures. The first measure is used to assess the satisfaction with the information correctness. This is determined on the basis of the difference between the expected and actual sojourn time. The second measure is used to assess the satisfaction with the waiting time with respect to the service time.

We include the following chapters. Chapter 2 studies the multi-agent based barge and terminal operations, the time dependent traveling salesman problem, and other related studies. Chapter 3 describes the methodology, including the policies, the scenario settings, simulation program design, sample simulations, and satisfaction levels. Chapter 4 presents the results of the simulation experiments. Chapter 5 discusses the findings of the research, answers the research questions, and provides suggestions for further research. Appendix A shows the data used in the analysis of the results. Appendix B provides the source code of the simulation program.

Chapter 2

Literature review

2.1 Multi-agent based barge and terminal operations

This section discusses three studies by Douma et al. [11, 12, 13] conducted between 2009 and 2012. The research deals with the deployment of multi-agent systems in the barge terminal scheduling problem. The authors consider the problem of aligning barge rotations with quay schedules of terminals in the port of Rotterdam. The purpose of the multi-agent system is to let barge operators and terminal operators make appointments with each other in a more efficient way.

The problem considered is related to the berth allocation problem [7] and the ship scheduling and routing problem [5]. Also, it is related to the attended home delivery problem [3] and the hospital patient scheduling problem [10]. For a discussion about the differences we refer to [11]. Contrary to the just mentioned problems, for the new problem a central solution (i.e., know all information in advance) is not feasible because of several reasons, such as autonomy of the barge companies, the lack of contractual relations between barge companies and terminals, and the lack of sharing information because of competitive reasons. In addition, there is also a complexity in mathematical terms, including dealing with different interests of different players, the dynamic environment, and the low structured and loosely coupled network. This is why the authors propose a decentralized, multi-

agent based approach, since it can mirror the market structure and can provide a solution that is acceptable to both the barge and the terminal companies.

2.1.1 Waiting profiles

The use of waiting profiles to support the alignment of barge and terminal operations is introduced in [11]. When using waiting profiles, the terminal operators provide the barge operators with information about the maximum amount of time a barge has to wait until containers will be starting to (un)load after it has arrived at the terminal. The barge operator can determine the rotation with the smallest sojourn time based on the waiting profiles, sailing times in the port, and handling times at the terminals. The authors add slack to the waiting profiles in order to increase the planning flexibility of terminals. One of the important assumptions in the study is that the sailing and handling times are deterministic. Simulation results indicate that using waiting profiles is a promising control structure to enhance barge terminal operations' efficiency. In this thesis, we build further on the idea of using waiting profiles to improve the alignment between barge and terminals. The rest of this section describes the initial model and the mathematical models for the barge and terminal agent, which are acquired from [11]. Section 3.4 shows examples of applications of the model.

Initial model

The waiting profile concept lays the foundation of the use of multi-agent systems for the barge handling problem. It introduces two types of agents, one for barge operators and one for terminal operators. In addition, it introduces the waiting profile, which includes information about maximum waiting times for every possible arriving time at a terminal. A terminal operator agent is reactive, which means that it responds to requests of barges. The barge operator agent is proactive, which means that it can anticipate on the situations. The agents use a direct communication mechanism, which allows agents to contact each other directly. The goal of the

agents is to make appointments. An appointment is defined as an agreement from both the terminal side and the barge side. The operation sequence is as follows.

1. The barge agent requests waiting profiles from the terminals it has to visit when its corresponding barge enters the port.
2. The terminal agents react to the requests by providing the barge agent with waiting profiles.
3. The barge agent determines its optimal port rotation based on the waiting profiles and on network information (i.e., sailing times).
4. The barge agent makes appointments with terminal agents based on the rotation. An appointment contains the following information:
   (a) The barge agrees to be at the terminal at a certain time, which is called the latest arrival time (LAT).
   (b) The terminal agrees to start processing the barge at the latest starting time (LST).

If the barge is later than the latest starting time, then it has to make a new appointment. There are no further consequences such as penalty costs. The assumption is made that there are no disturbances, and that sailing, mooring, and handling times are deterministic. This allows a barge to plan exactly when it arrives, and thus it will not arrive later than the latest starting time.

Information exchange levels In order to test the waiting profile concept, it is compared to two other levels of information exchange, i.e., the extent to which terminals provide barges with information about their schedule. The idea is that with more information barges can determine rotations with less sojourn time in the port. The levels of information exchange are as follows.

1. No information. Terminals do not provide barges with information. The barge operator determines the rotation with only a shortest path algorithm.

2. Yes/no. Barge operators can repeatedly request whether certain times are available, then the terminal operator responds with yes or no. The fastest rotation with this information level often is not the shortest path.

3. Waiting profiles. Terminal operators provide barge operators with information about the maximum amount of time a barge has to wait after it has arrived for every possible arrival moment. The barge will then determine the rotation with the smallest sojourn time and make the appointments with the terminals.

The comparisons show that using waiting profiles is the superior information exchange level [11]. Therefore, this thesis adapts the waiting profile, and dismisses the other two information exchange levels.

Waiting profile The authors define the waiting profile of terminal i as a t-parameter family of pairs (t, w_{it}), where w_{it} is the maximum waiting time when the barge arrives at terminal i at time t, for all t during time period $[0, T]$. The waiting profile is specific for every barge and time and is generated after a request of a barge. The maximum waiting time is guaranteed making it possible for the barge to arrive at the agreed time at their next destination. The waiting profile should be beneficial for both terminal and barge. Terminals can use the profile to get more flexibility in their schedule, by increasing the waiting times during busy hours. Barges can use the profile to determine a rotation with minimum sojourn time in the port.

Barge operator agent

The barge operator agent has to decide the rotation in the port. The mathematical model is described below. Note that the model does not consider capacity and stowage constraints.

Model and notation The barge operator agent makes two decisions: (1) in which sequence the barge visits the terminals, and (2) the specific time each terminal is

visited. It is assumed that the decisions are made when a barge enters the port, the information of the terminals is reliable, and that barge operators make decisions in real-time.

N is a set of terminals that has to be visited by a barge, it is a subset of all terminals in the port. For every $i \in N$ the agent knows the handling time h_i. For every $(i, j) \in N$ the agent knows the sailing time s_{ij}. Assume that h_i and s_{ij} are deterministic. A barge has to pass the port entrance and exit point. The primary objective is to finish all activities in the port as soon as possible. The secondary objective is to minimize the sailing time. Assume the agent knows the maximum waiting time w_{it} for every $i \in N$ at every arrival time t. The agent assumes it has to wait w_{it}. The time dependent travel time $\tau_{ij}(d_i)$ is defined as the sum of sailing time s_{ij}, handling time h_i, and maximum waiting time w_{it} from i to j, where the barge departs at time d_i at terminal i. Let the arrival time at terminal j be denoted by $a_j(d_i)$, thus $a_j(d_i) = d_i + s_{ij}$. Then $\tau_{ij}(d_i) = s_{ij} + w_{j,a_j(d_i)} + h_j$.

It follows that the maximum amount of time between i and j is given by $\tau_{ij}(d_i)$. The objective of the barge is to find the rotation which minimizes the sum of the time dependent travel times. Assume that the terminals in N are visited once, and assume that the handling process is never interrupted. The model is solved using a TDTSP, see section 2.2. The notations are summarized as follows:

N	set of terminals that a barge has to visit
h_i	handling time at i, $i \in N$
s_{ij}	sailing time from i to j, $(i, j) \in N$
w_{it}	maximum waiting time at i, $i \in N$ at arrival time t
d_i	departure time from i, $i \in N$
$a_j(d_i)$	arrival time at j, $j \in N$, $a_j(d_i) = d_i + s_{ij}$
$\tau_{ij}(d_i)$	time dependent travel time $\tau_{ij}(d_i) = s_{ij} + w_{j,a_j(d_i)} + h_j$

How decisions are made by the barge operator The decisions of barge operators depend on the information exchange of the terminals. The information

exchange level *waiting profile* means that all terminals $i \in N$ provide barge operators with information about the maximum waiting times. A barge assumes it has to wait the indicated maximum amount of time. Based on the maximum waiting times, sailing times and handling times the barge solves the TDTSP. Based on the outcome of the TDTSP the barge makes its decisions.

Terminal operator agent

The terminal operator agent has to respond to requests from barges. The information must contain available times barges can be processed. When the terminal has more information about incoming barges, then the information is of higher quality. Assume that terminals only have information about barges that are already in the port. The model and the way terminals operate is described below.

Model and notations Every terminal has an agent to make appointments with barges. Let Q be the set of berth-crane-team combinations. Every $q \in Q$ represents the resources required to handle one barge. Every barge is assigned to one $q \in Q$ and every $q \in Q$ has the capacity to process one barge at a time. Assume the handling time of each container is deterministic.

The first task of the agent is to make appointments with barge operator agents. The second task is to keep these appointments. When making appointments, the agent has to generate a waiting profile using the handling times of the barge. The agent will generate this when a barge requests it. Keeping appointments implies that barges are scheduled without violating existing appointments.

An appointment is defined as an agreement from both the terminal side and the barge side. The barge agrees to be at the terminal at the latest arrival time (LAT). The terminal then specifies the maximum waiting time (MWT) for this LAT and agrees that the barge will be processed at a latest starting time (LST). The latest starting time is the sum of the latest arrival time and the maximum waiting time, LST = LAT + MWT. When the appointment is made, the terminal schedules it such

that no other appointments are violated. The schedule, which is only visible to the terminal agent, contains a planned starting time (PST) and an expected departure time (EDT) for every barge. The expected departure time is the sum of planned starting time and the processing time (PT), this is expressed as EDT = PST + PT. The PT is the handling time h_j of the barge.

Keep appointments The terminal agent has to ensure that all appointments are met (i.e., no barge starts later than the LST). However, it is possible that rescheduling appointments may be advantageous, e.g., when a barge arrives earlier than its LAT, which can happen when a barge does not have to wait the MWT at a previous terminal. Rescheduling means that a barge can start handling before a barge that is scheduled earlier. When a barge arrives that is not next in the schedule, the terminal will check whether the handling can start without crossing the LST of the next barge in the schedule. If this is true, the handling can start, otherwise the barge has to wait.

Waiting profile construction A waiting profile is constructed at the moment a barge requests it. The waiting profile contains the MWT until the handling of the barge starts, for every arrival moment during the planning horizon. The terminal determines all possible start intervals in its current schedule. A start interval is a time interval in which the barge can start without violating any appointments. All possible insertion points i are considered. Insertion point i is the point after the ith barge, where $i = 0$ means the barge is scheduled as first. All barges before every insertion point are planned as early as possible (i.e., at LAT), while barges after the insertion point are planned as late as possible (i.e., at LST). The sequence of already planned barges does not change this way. The starting time of the barge is then equal to the EDT of the preceding barge, if $i = 0$ then the starting time is the actual time. The end time is the PST of the next barge, minus the PT of the concerned barge. If there is no next barge, then the end time is set to infinity. The feasible start intervals are added to the list. The waiting time is zero if the arrival

time is between the start time and end time of a start interval. Else, the waiting time is calculated by subtracting the next start time from the arrival time. If the outcome is a negative number, then the waiting time is zero.

2.1.2 Service-time profiles

In [12] the concept of service-time profiles is introduced. This is an extension on the waiting profiles. Not only the waiting time until the start of handling is considered, but also the handling time itself. This is referred to as the service time, which is defined as the sum of the handling and waiting time. The advantage of the service-time profile compared to the waiting profile is that it can take into consideration closing times of terminals. The results show that service-time profiles perform slightly better than waiting time profiles, although the advantages are minor. The value of adding slack to appointments also becomes clear in this study. It improves the performance of the system, because it increases the flexibility of the planning, which leads to lower sojourn time for the barges. The relation between slack, terminal utilization degree, and barge waiting time is not clear yet. The authors suggest further research on that.

2.1.3 Usefulness and acceptance

In [13] the authors develop a real-time multi-player game which simulates the multi-agent system for the barge handling problem. The acceptance of a system by users depends on perceived usefulness and perceived ease of use [8]. In this case, the perceived usefulness is the degree to which a barge or terminal operator believes the multi-agent system is improving the appointment making process. The perceived ease of use is the degree to which a barge or terminal operator believes the use of the multi-agent system is effortless. The simulation game was developed to test these two acceptance factors.

The game consists of a port with a number of terminals, and barges that have to visit a subset of these terminals. The barges enter and leave the port via the

same start and end point. The players of the game are barge operator planners. The terminal operator is controlled by the computer. The player has to plan the sequence in which the terminals are visited. The goal is to minimize the time a barge is in the port, i.e., minimize the turn-around time. Since it is a multi-player game, and thus players make decisions concurrently, they influence each other's possibilities. The game proceeds as follows. The barges start at the start point, which represents the entrance of the port. All players know which terminals to visit before the beginning of the game. Players can make their rotation plan as soon as the game starts. Players are allowed to change their plans at every time in the game. The handling time is equal for each barge at each terminal. Terminals will answer information requests with a random delay, which simulates the time an operator agent needs to respond. The user interface contains a map with the location of the barges and terminals, a tab with statistics of the barge and a rotation planning tab.

The authors reduce the complexity of the design in order to keep the program understandable for the players. They argue that when including more details and making the game more complex, the perceived usefulness and user satisfaction may decrease [16]. The following assumptions are made. Terminals have no restricted opening times, therefore the less complex waiting profiles are used instead of the service-time profiles. Experiments with different groups, including barge operators, show that the game is a useful tool to give people a clear understanding about the system. Also, the game shows what the impact was of having more information. Players changed the way of decision making based on information they got on the terminal. The game also has an additional purpose, which is making the system tangible and explainable to potential practitioners. The authors argue that without the game it would not be possible to get support to make further steps to implementation. The authors conclude that the game is a step forward in the acceptance of the system.

2.2 Time dependent traveling salesman problem

The barge agent computes a rotation by solving a time dependent traveling salesman problem (TDTSP). In this section the TDTSP is further explored. The TDTSP is an extension of the traveling salesman problem (TSP) and is a special case of the time dependent vehicle routing problem (TDVRP). The solution (i.e., the rotation) depends on the sailing time between terminals and the time of the day.

2.2.1 Mixed integer programming

The following is a mixed integer programming formulation of the TDTSP acquired from [18].

Constants

n	number of nodes including the depot
M	number of time intervals considered for each link
c_{ij}^m	travel time from i to j if started at i during time interval m; $c_{ii}^m = \infty, \forall i, m$
c_i	service time at node i; $c_i = 0$ for $i = 1, n+1$
T_{ij}^m	upper bound for time interval m for link (i, j)
t	the starting time from the depot node 1
B_1	a large number
B_2	a large number
L_i	earliest time to arrive at node i
U_i	latest time to arrive at node i

Decision variables

x_{ij}^m	1 if there is a travel from node i to j during time interval m; 0 otherwise
t_j	departure time from node j

Minimize

$$t_{n+1} \tag{2.1}$$

subject to

$$\sum_{i=1,i\neq j}^{n}\sum_{m=1}^{M} x_{ij}^{m} = 1 \qquad (j = 2,...,n+1) \tag{2.2}$$

$$\sum_{j=2,j\neq i}^{n+1}\sum_{m=1}^{M} x_{ij}^{m} = 1 \qquad (i = 2,...,n) \tag{2.3}$$

$$\sum_{j=2}^{n}\sum_{m=1}^{M} x_{ij}^{m} = 1 \tag{2.4}$$

$$t_1 = t \tag{2.5}$$

$$t_j - t_i - B_1 x_{ij}^{m} \geq c_{ij}^{m} + c_j - B_1 \qquad (i = 1,...,n; j = 2,...,n+1; i \neq j; m = 1,...,M) \tag{2.6}$$

$$t_i + B_2 x_i j^{m} \leq {T_{ij}}^{m} + B_2 \qquad (i = 1,...,n; j = 2,...,n+1; i \neq j; m = 1,...,M) \tag{2.7}$$

$$t_i - {T_{ij}}^{m-1} x_{ij}^{m} \geq 0 \qquad (i = 1,...,n; j = 2,...,n+1; i \neq j; m = 1,...,M) \tag{2.8}$$

$$L_i + c_i \leq t_i \leq U_i + c_i \qquad (i = 1,...,n+1) \tag{2.9}$$

$$x_{ij}^{m} \in \{0,1\} \qquad \forall i,j,m \tag{2.10}$$

$$t_i \geq 0 \qquad \forall i \tag{2.11}$$

The objective function (2.1) minimizes the total route time of the barge, which is sailing time + handling time + waiting time. Constraints (2.2) and (2.3) ensure that each terminal is visited exactly once. Constraint (2.4) ensures that the terminals are visited by the one corresponding barge. Increasing this number means increasing the number of barges concerned by one agent, however this is disregarded in this thesis. Constraint (2.5) sets the starting time from the port entrance to t. Constraint (2.6) computes the departure time at terminal j. Constraints (2.7)

and (2.8) ensure that the proper parallel link m is chosen between terminals i and j according to the departure time from node i. Constraint (2.9) imposes the time windows that are defined in terms of the arrival times at the nodes while the variables t_i for $i = 1,...,n+1$ represent the departure times from the terminals. Constraint (2.10) ensures that x_{ij}^m is a binary variable. Constraint (2.11) ensures that the departure time is non-negative.

2.2.2 Dynamic programming exact algorithm

The following algorithm is acquired from [19]. A directed graph $G(V,E)$ is given with V the set of nodes and E the set of directed links. A directed link is represented by an ordered pair of nodes (i,j) in which i is called the origin and j is called the destination of the link. The graph is assumed complete and an $n \times n$ time dependent matrix $C(t) = [c_{ij}(t_i)]$ is also given representing the travel times on every link $(i,j) \in E$, where $c_{ij}(t_i)$ is a function of the departure time t_i from the origin node i of the link.

A dynamic programming exact algorithm that finds a TDTSP tour with the earliest return time to the depot for a given starting time from the depot is described next. There are $n-1$ customers to be visited, represented by nodes $1,...,n$–1, and the depot is node 0. Given a set of customers $S \subseteq \{1,...,n-1\}$ and $k \in S$, let $T(S,k)$ be the minimum time needed to start from the depot, visit all the nodes in S and end at node k. The first step is to find $T(S,k)$ for $|S| = 1$ which is

$$T(\{k\},\ k) = T_0 + c_0 + c_{0k}(t_0) \qquad \forall k = 1...,n-1 \qquad (2.12)$$

where T_0 is the starting time from the depot, c_0 is the serving (or preparation time) at the depot, and $c_{0k}(t_0)$ is the travel time from node 0 to k directly as a function of the departure time from node 0, $t_0 = T_0 + c_0$. Hence $T(k,k)$ represents the arrival time at node k.

For $|S| > 1$ and k the last node visited, we consider visiting k immediately after p (for all p) and look up $T(S-k,p)$ from the previous calculations. So we have

$$T(S,k) = \min_{p \in S-k} [T(S-\{k\},p) + c_p + c_{pk}(t_p)] \qquad \forall k \in S \tag{2.13}$$

where c_p is the service time at node p and $c_{pk}(t_p)$ is the travel time from node p to k directly as a function of the departure time $t_p = T(S-\{k\},p) + c_p$ from node p. For the complete tour the minimum return time to the depot is
$T* = \min_{p \in \{1,...,n-1\}} [T(\{1,...,n-1\},p) + c_p + c_{p0(t_p)}]$.

2.2.3 Computational time

It is important that a barge agent is able to provide a solution to the TDTSP algorithm in a short amount of time, because it has to make a decision about where to go after it arrives at the port. The authors of [11] use a depth-first search (DFS) algorithm for instances with up to seven terminals, and the dynamic programming heuristic for instances with more than seven terminals. The heuristic is a modified variant of the dynamic programming exact algorithm, where only a fixed number of partial tours is retained at each stage.

We conduct experiments using the dynamic exact algorithm described in Section 2.2.2. The results show that the amount of time required to solve the problem is acceptable (i.e., less than a few seconds) for up to eight terminals to visit per barge. We assume a barge does not visit more than this number of terminals. Therefore, we choose only to utilize the dynamic exact algorithm in order to always obtain the optimal solution.

2.3 Other related papers

The authors of [22] discuss the importance of optimizing logistic operations at container terminals by reviewing theoretically and practically oriented papers. The authors describe that many problems addressed relate to vehicle routing problems. The authors review papers related to operation research at container terminals. Two topics relate to our topic. First, the berth allocation problem, which is interpreted in several ways. There are variations in the constraints and in the algorithms that are used for solving the problem, such as tabu search and CPLEX algorithms. Second, the multi-agent approaches. The authors of [23] introduce an agent architecture focusing on the quayside operations in order to reduce the ship handling time. This architecture uses cooperative scheduling of quay cranes and container vehicles.

In [2] a tactical planning model for service network design in barge transportation is proposed. The model is intended as a decision support tool for barge operators and shipping lines that want to offer roundtrip services. A roundtrip is defined as a shipping route that starts at a port in the hinterland, then proceeds to a major port and finally returns to the same hinterland port. The model determines the optimal route for a given capacity and roundtrip frequency. The results suggest that when shipping lines plan barge services and empty container repositioning jointly it will reduce costs. The primary assumption made in the model is that demand is known beforehand. To deal with this assumption, a part of the vessel capacity can be reserved. The amount to be reserved depends on the variability of demand, which is comparable to the safety stock in inventory theory.

In [4] an overview is provided of planning decisions in intermodal freight transport and solution methods proposed in the scientific literature. The authors indicate that the main attention of the literature has been given to intermodal transport by rail. It is suggested that future attention can be given to operations in intermodal barge transport. For regions with extensive waterway networks (e.g., Western Europe) scientific development in this area is necessary.

Chapter 3

Methodology

We use the multi-agent model from Section 2.1.1 to test our policies. We use the agent types and waiting profile concept. This concept is chosen because it contains the necessary elements to do the research, which are software agents with their accompanying behavior rules. This thesis omits service-time profiles, because taking into account terminal closing times will unnecessarily make the model more complex. By doing this the focus is kept on the policies. We build a computer simulation program to run the model. The programming language we use is Java, in combination with Repast Symphony. Repast is an open source agent-based modeling and simulation platform. The advantage of using Repast is that it has a fully concurrent discrete event scheduler.

Section 3.1 describes the policies, Section 3.2 describes the scenarios in which the policies are tested, Section 3.3 describes the software implementation, Section 3.4 shows two example simulation runs, and describes an example of how a single barge is modeled, and Section 3.5 describes the satisfaction measures.

3.1 Policies

We investigate two policy settings. The first policy setting relates to the terminal logic. The second policy setting relates to the slack added to the waiting profiles.

This section discusses the policy settings in more detail.

3.1.1 Terminal logic

We define terminal logic as the way a terminal processes barges, i.e., how a terminal determines if a barge can start with (un)loading containers. A distinction is made between two types: reserved and unreserved.

Reserved terminal logic

The reserved terminal logic is based on the appointment based method introduced in [11]. In this method, a barge promises a latest arrival time to the terminal, and the terminal promises the barge a latest starting time. In the reserved terminal logic, we translate this as follows: when a barge arrives at a terminal, it can start handling if it is the next barge in the terminal schedule and the terminal is idle. If the barge is not the next barge in the schedule, but the terminal is idle, it is possible to start handling if the time of arrival at the terminal plus the handling time at the terminal (i.e., the expected end time of handling) is smaller than the latest starting time of the next barge in the schedule. If the handling cannot start, the barge waits in the queue, until the terminal decides that the handling can start. Figure 3.1 shows the activity diagram of the reserved terminal logic.

We expect that this method works best if it is easier to predict arrival times at terminals. This is the case with a deterministic model, waiting profiles with no slack or a low amount of slack. We expect that this method works less well when there is more uncertainty in the model, i.e., when using a stochastic model and adding slack to the waiting profiles. The advantage of this system is that it is clear for barges when they will be handled, and it is also clear for terminals when barges will arrive. A disadvantage could be that when a barge arrives at an idle terminal, it still has to wait.

It is possible that a terminal cannot start handling before the promised latest starting time. In that case, a barge has to wait until the terminal is ready. This can

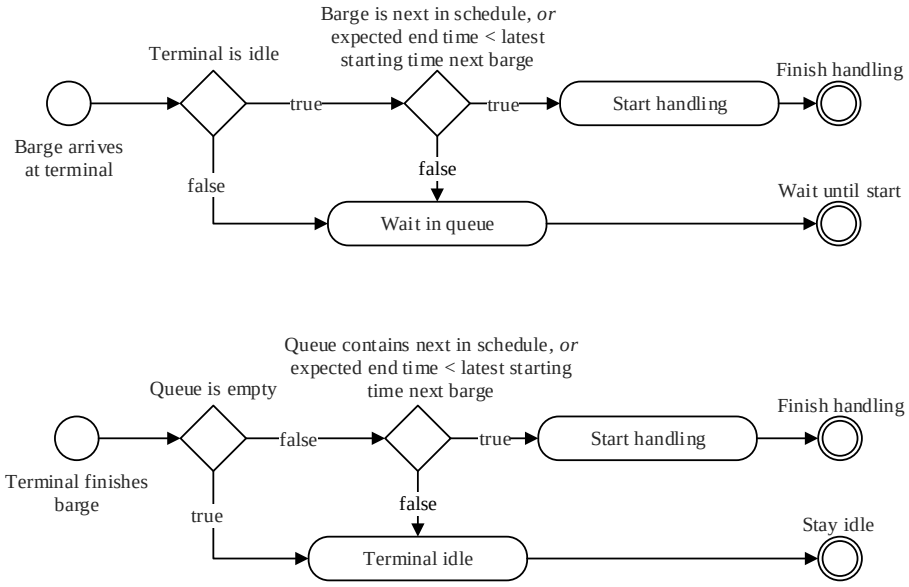


Figure 3.1: Reserved terminal logic. The upper diagram shows the activities that occur when a barge arrives at a terminal. The bottom diagram shows the activities that occur when a terminal finishes the handling of a barge.

happen when using stochastic values for the handling and sailing times, whereby the handling time takes longer than planned, and the difference between the expected end time of handling and the latest starting time of the next barge in the schedule is smaller than the extra time the handling takes. When a barge arrives later than the promised latest arrival time, it has to wait in the queue until the terminal has time to start handling. This policy will be the same as in [11], which makes a comparison possible.

Unreserved terminal logic

The unreserved terminal logic is based on the well known first in, first out (FIFO) method, in combination with the appointment based method. This means that the terminal handles barges in the order in which they arrive, and the barges still make appointments with the terminals based on waiting profiles. The appointments are less strict than with the reserved method, and can be seen more as an indication

of the arrival time and start of handling time. This means that a barge can start handling, even if it is not the next barge in the schedule, or when the expected end time of handling is larger than the latest starting time of the next barge in the schedule. Figure 3.2 shows the activity diagram of the unreserved terminal logic.

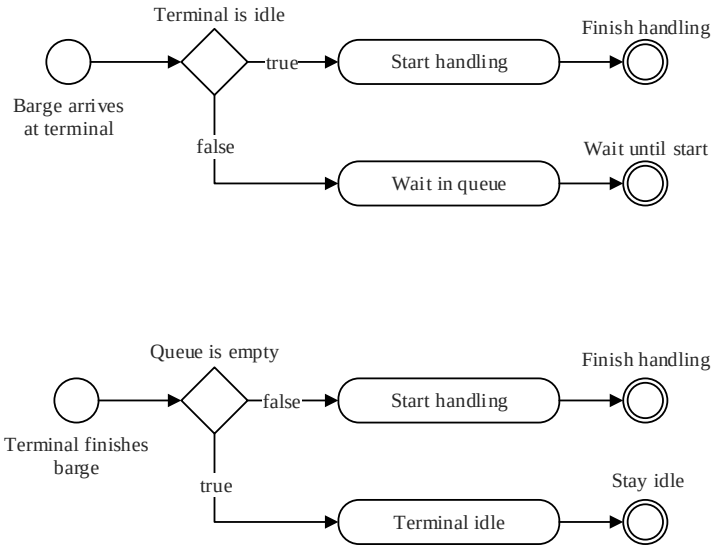


Figure 3.2: Unreserved terminal logic. The upper diagram shows the activities that occur when a barge arrives at a terminal. The bottom diagram shows the activities that occur when a terminal finishes the handling of a barge.

The difference between the reserved and unreserved terminal logic is that with the reserved terminal logic, the terminal uses its schedule to determine whether or not a barge can start handling, while with the unreserved terminal logic the appointments schedule will not be considered. In either case, the appointments are made. In the unreserved method, the appointments are used as guideline for the barges to determine their route, rather than a strict appointment between barge and terminal.

The advantage of the unreserved terminal logic is that when barges arrive at an idle terminal, they do not need to wait. In addition, it allows easier operations at the terminals. A disadvantage could be, that when a barge arrives at the appointed time, it still has to wait because another barge arrived one moment earlier. We

expect that this policy can result in less waiting time, in particular in scenarios in which it is harder to predict the arrival times at terminals, i.e., with a stochastic model, and waiting profiles with slack.

3.1.2 Slack policy

A terminal has to decide how much slack to add to a waiting profile. The constant slack is described below. Adding slack to a waiting profile means that the expected maximum waiting time increases. Barges use the maximum waiting time to compute their rotation in the port, and make appointments based on the outcome of that computation. The aim of adding slack is to create more planning flexibility in the schedule of the terminal operators. Previous research [11] performed experiments in which a slack of 0, 30 and 60 minutes was added to the waiting profiles. These values are estimations and are not determined with a particular method. The research showed that the benefit of the amount of slack relate to the arrival rate of barges. Low arrival rates requires a slack between 0 and 30 minutes, while higher arrival rates require a slack between 30 and 60 minutes. In our scenarios, we test 30, 60 and 90 minutes of slack for each arrival rate. Also, we use a slack of 0 minutes (i.e., no slack). We refer to this method as *constant slack*.

3.2 Scenarios

We evaluate the policies in a stochastic and deterministic model. The handling and sailing times are fixed in the deterministic model, while they are randomized in the stochastic model. Furthermore, we evaluate five barge arrival rates. The settings are discussed in more detail below.

3.2.1 Deterministic and stochastic model

When moving through the port, there are two parameter settings that influence how long the actual sojourn time is. The first setting is to use deterministic values, this means that the actual sailing and handling times are the same as the ones used to determine the route. The second setting is to use stochastic values, this means that the actual sailing and handling times are not fixed. When using stochastic values, the sailing and handling times are calculated by means of a Gaussian distributed random number generator. The mean is the value used by the barge to determine the route, and the standard deviation is set to 3 minutes. We choose this number, because it ensures that the deviation seems realistic. We round the outcome of the random number generator to the nearest integer to obtain a discrete value.

3.2.2 Port settings

The number of terminals in the port is derived from the sailing times table. We obtained a distance matrix from a paper of the Port of Rotterdam. We convert the distance matrix to a sailing times table by assuming a barge sails at a speed of 12 km/hr. The table contains the sailing times between 14 terminals, see Table 3.1. Each terminal has the capacity to handle one barge at a time in our model. The sailing times table and the capacity do not necessarily correspond to the actual port. Nevertheless, we find that these values are a good indication for testing our policies.

The handling times are obtained through a Gaussian distributed random num-

Table 3.1: Sailing times in minutes between terminals and port entrance/exit. The port entrance/exit is referred to by t0. Terminal 1 to 14 are referred to by t1 to t14.

	t0	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12	t13	t14
t0	0	42	32	40	46	36	31	28	35	32	28	31	35	18	28
t1	42	0	30	38	44	34	30	26	10	13	17	20	24	27	26
t2	32	30	0	19	25	15	11	6	24	20	16	19	23	17	11
t3	40	38	19	0	17	6	12	15	32	28	24	27	31	25	12
t4	46	44	25	17	0	13	18	21	38	34	30	33	37	31	18
t5	36	34	15	6	13	0	3	11	28	24	20	23	27	21	8
t6	31	30	11	12	18	3	0	7	23	20	16	19	23	17	3
t7	28	26	6	15	21	11	7	0	20	16	12	15	19	13	3
t8	35	10	24	32	38	28	23	20	0	7	11	14	18	21	20
t9	32	13	20	28	34	24	20	16	7	0	7	10	14	17	16
t10	28	17	16	24	30	20	16	12	11	7	0	6	10	13	12
t11	31	20	19	27	33	23	19	15	14	10	6	0	7	16	15
t12	35	24	23	31	37	27	23	19	18	14	10	7	0	20	19
t13	18	27	17	25	31	21	17	13	21	17	13	16	20	0	13
t14	28	26	11	12	18	8	3	3	20	16	12	15	19	13	0

ber generator, with a mean of 30 minutes and a standard deviation of 10 minutes, which is the same as in [11]. We add to this a discretization of the value by means of rounding to the nearest integer, and that the handling time at a terminal cannot be lower than 10 minutes.

3.2.3 Number of terminals to visit

The number of terminals a barge visits is determined by means of a Gaussian distributed random number generator, with a mean of 5 terminals and a standard deviation of 1 terminal. The maximum terminals a barge visits is set to 8. These values are estimations, which we assume corresponds well with the actual situation.

3.2.4 Simulation length

The length of each simulation is 10 days plus a 2 day warm-up period, i.e., 17,280 minutes. We add the warm-up period, since the first barges arrive in the simulations at an empty port, giving them almost no waiting time. That results in unbalanced statistics. Therefore, barges that leave the port within the warm-up period are not included in the statistics.

3.2.5 Barge arrival rate

The barges arrive in the port according to an exponential distributed random number generator. The goal is to run scenarios which vary between quiet and busy. Quiet scenarios are those with few queues and thus where barges have little waiting time, while busy scenarios are the opposite. To determine the arrival rates, we ran test simulations which showed that the port is very busy when using an arrival rate of 1 barge every 12 minutes on average according to an exponential distribution. With an arrival rate of 1 barge every 15 minutes, we observe a quiet port. Therefore, we will run scenarios with the arrival rates 12, 13, 14, and 15 minutes per barge on average according to an exponential distribution.

We make a calculated guess on the capacity of the port, based on the parameter settings. There are 14 terminals, which are available 14,400 minutes. Each terminal has the capacity to handle one barge at a time. So the capacity in minutes is $(14 \times 14,400 =)$ 201,600. A barge visits 5 terminals on average, and has a handling time of 30 minutes on average at each terminal. So each barge requires 150 minutes of capacity. If we neglect the sailing and waiting times, then the port can handle $\frac{201,600}{150} = 1,344$ barges. The corresponding arrival rate is $\frac{14,400}{1,344} = 10.7$ minutes per barge. If we take into account the sailing and waiting times, then we expect that the utilization of the terminals will be high with an arrival rate of 12 minutes per barge. This is consistent with the observations in the test simulation, and confirms that we choose correct barge arrival rates.

3.2.6 Pseudo randomness

The simulation program uses fixed seeds to initialize the (pseudo) random number generators. This is necessary to make fair comparisons between policies. We need to investigate the influence of the random seed in order to rule out that there is not an arbitrary preference for a policy. We will average the outcomes of the simulations over the random seeds, in order to level the randomness of the distributions.

The random seed is used when generating the time of arrival of a barge at the port, the number of terminals to visit by a barge, the handling time at each terminal, and to deviate the sailing and handling times in the stochastic model. We use four seeds: 489258742, 651517117, 284324984, and 166267832. The simulation program uses the Apache Commons random data generator [6].

3.3 Software design

To design the software, we first draft the requirements. The requirements are as follows. The program needs to simulate the process of barges visiting terminals in a port. We do this by using discrete-event simulation, i.e., we model the operations in the port as a discrete sequence of events in time. The port contains multiple terminals. The terminals can construct waiting profiles on the request of barges. The barges can determine their route in the port by solving a TDTSP. The program needs to collect statistics of the simulation.

In Section 3.3.1, we describe the events of the simulator. In Section 3.3.2, we briefly describe the implementation by showing the classes of the program.

3.3.1 Discrete event simulation

The simulation program contains a schedule, we refer to this as event schedule. All the events that occur at a particular time are registered in the event schedule. From a technical point of view, the simulation program goes from event to event. At one event a new event is scheduled, which keeps the program going.

Events

The events are briefly described below. Figure 3.3 shows the sequence of events of a barge in the port. The simulation starts by scheduling the arrival of the first barge in the event schedule.

1. Barge arrives at the port A subset of the terminals in the port is assigned to the list of terminals to visit, along with the corresponding handling times. The barge agent requests the waiting profiles from the terminals. The terminal agents react to this request, by constructing the waiting profiles and sending them to the barge agent. The barge computes its best route with the TDTSP algorithm. Based on the outcome, the appointments are made with the terminals. The arrival at the first terminal is scheduled in the event schedule. Finally, the arrival of a new barge is scheduled in the event schedule, which is necessary to keep the simulation going.

2. Barge arrives at a terminal The barge is added to the queue. At this point, the terminal logic is going to do its job, see Section 3.1.1. The terminal agent schedules the start of the handling in the event schedule.

3. Terminal starts handling barge (Un)loading of containers begins. The terminal schedules when the handling is finished in the event simulation schedule.

4. Terminal finishes handling barge The barge is released from the terminal. It has to decide what to do. It has two options: (1) if it has to visit more terminals, then it schedules the next arrival at the next terminal in the event schedule; (2) else it schedules to leave the port in the event schedule.

5. Barge leaves port The system registers the statistics of the barge. The barge is removed from the port.

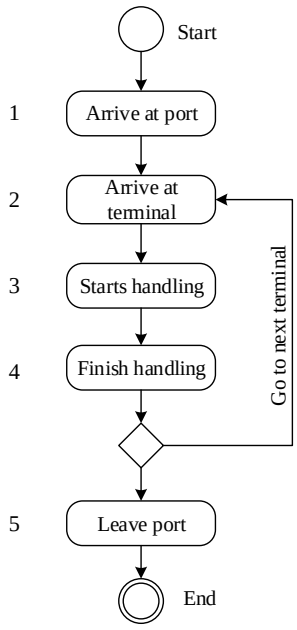


Figure 3.3: The sequence of events that occur in the port for every barge.

Scheduling events

An event is scheduled by implementing the Repast ISchedule interface. The Repast ISchedule implementation manages the execution of events according to the simulation clock. An event is scheduled by specifying a time, a target object on which to call a method, the method to call, and the parameters for the method. When the simulation clock is at the specified time, the method is called on the specified object. For more information about the interface, we refer to the Repast API [21].

3.3.2 Class overview

This section gives an overview of the program. Based on the requirements we design a program. An overview of the classes with their relations are shown in Figure 3.4. The rest of this section describes the classes briefly. We have also included the source code with documentation in Appendix B.

Port

The simulation context is specified in the Port class. The Port class implements the ContextBuilder interface from Repast Symphony. The interface makes possible that agents interact inside a context, and also that simulations can be configured from a user interface. This makes it more convenient to run multiple simulations. The implementation is available in appendix B.1.

The purpose of the Port class is to build the context, and to keep a schedule of the simulation events. Building the context is done in a few steps. First, the Repast ISchedule is initialized. Then the parameters are read from the user interface. The parameters that require configuration through the user interface are identified by us, and are specified in the parameters.xml file, a file from the Repast suite.

We choose to make the following parameters configurable: random seed, barge arrival rate, model type (i.e, stochastic or deterministic actual sailing and handling times), terminal logic, and slack setting. The terminal logic and slack setting are part of the terminal policy, see Section 3.1. The rest of the parameters are part of the scenarios, see Section 3.2. Next, the random generators are set up, the terminal objects are created, the sailing times are loaded from a spreadsheet, and a statistics object is initialized. The last step of building the context is to schedule the first arrival of a barge, which makes the simulation start.

Barge

The Barge class represents the barge operator agent and the barge itself. It is an implementation of the mathematical model described in Section 2.1.1. The barge

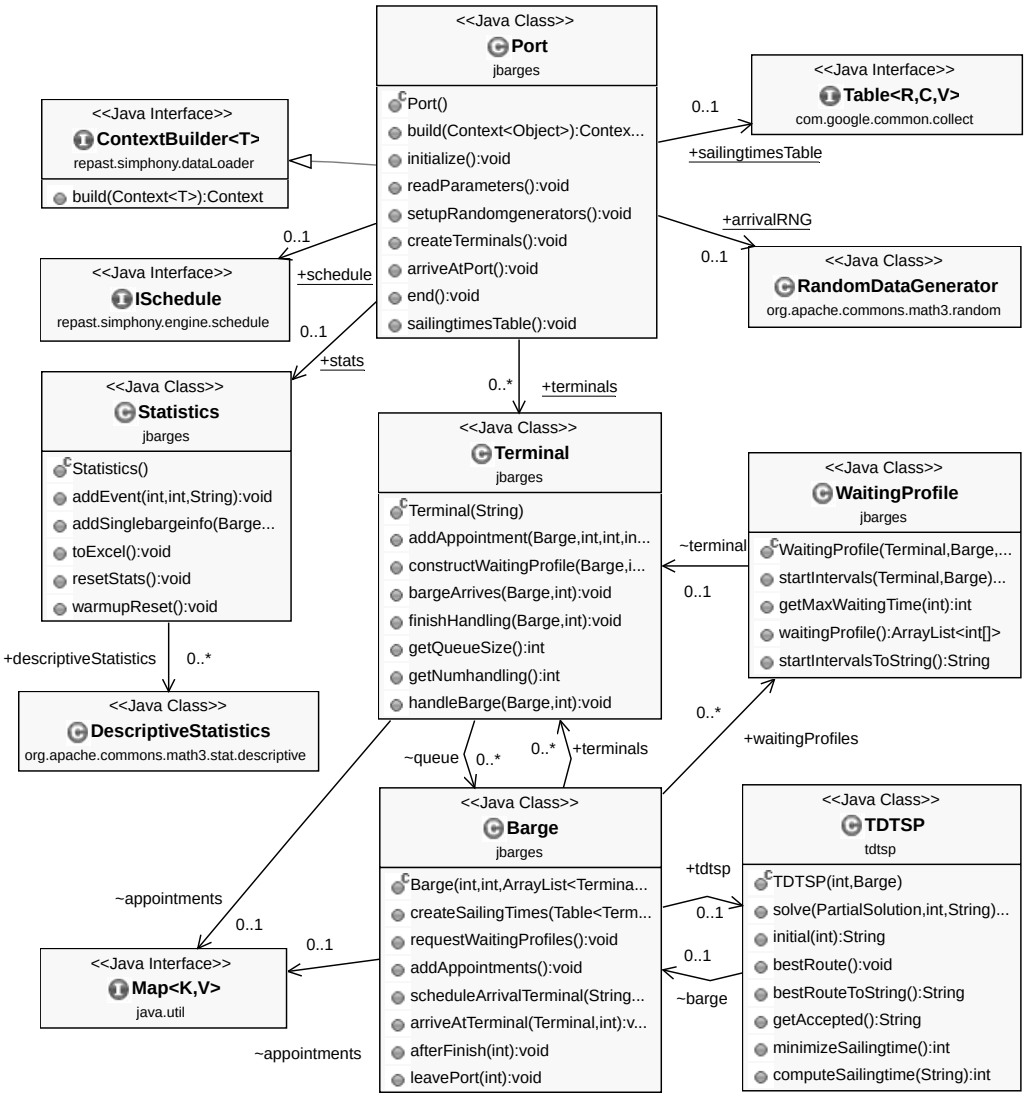


Figure 3.4: This compact version of the class diagram describes the structure of the program by showing the classes, its methods, and the relationships between classes. The Port class serves as the context for the simulation. The barge and terminal agents interact in this context. The Barge class simulates the operations of the barge. The Terminal class simulates the operations of the terminal. The Waiting profile class contains the waiting profile. The TDTSP class is used by a barge to determine the route. The Statistics class manages the statistics of the simulation. A full version of the class diagram is available online: `http://goo.gl/OIIrRZ`

is the entity that moves through the system. The purpose of the barge is to execute the operations of the barge. The implementation is available in appendix B.2. A barge has the following attributes: barge number to identify the barge, arrival time at the port, a list of terminals to visit and the associated handling times, a sailing times table that only contains the terminals this barge has to visit, a map to store the waiting profiles, a TDTSP object that handles the computation of the route and also stores all information associated with it, a map to store appointments with terminals, and integers to store the actual sailing, handling and waiting time and expected sojourn time.

Figure 3.5 shows the sequence of actions of the barge. In contrast to the sequence of events in Figure 3.3, here the interaction between classes is shown. The parameters for constructing a barge are determined in a method called *arriveAtPort* in the Port class. The parameters include the barge number, terminals to visit and associated handling times. When the barge is created, it immediately requests the waiting profiles from the terminals it has to visit. Also, it immediately creates the sailing times table with only the terminals it has to visit. It does this by taking the complete sailing times table from the port, and filters out all terminals it does not visit. Then, it solves the TDTSP algorithm by using the information from the waiting profiles, sailing times table, and handling times table. The TDTSP returns the best route, meaning the route with the least expected sojourn time in the port. Based on the best route, it makes the appointments with the terminals. After that it visits all terminals in the sequence of the appointments list. The last actions of the barge is that it registers the statistics. Recall that the statistics include the actual handling, sailing and waiting times and the expected sojourn time. Finally, the barge leaves the port, which removes the barge from the system.

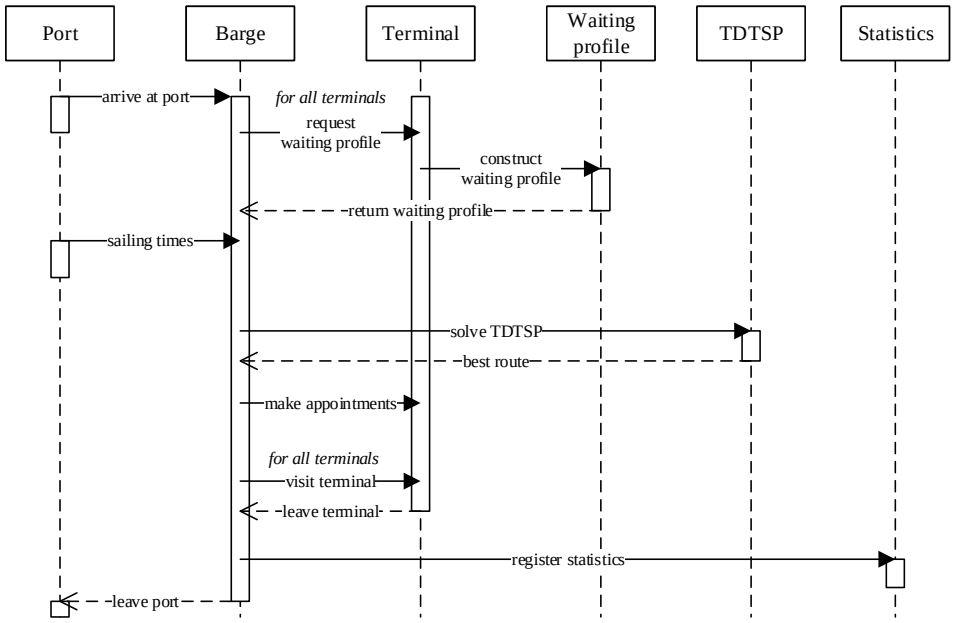


Figure 3.5: Sequence diagram of the barge.

Terminal

The Terminal class represents the terminal operator agent and the terminal itself. It is an implementation of the mathematical model described in Section 2.1.1. We add to the model different queue processing policies. The terminals are created when the context is built in the Port class. A terminal agent has to construct waiting profiles on the request of barges, and it has to handle barges that arrive at the terminal. The implementation is available in appendix B.3. The actual construction of a waiting profile takes place in the WaitingProfile class, see Section 3.3.2. However, the construction is triggered by the terminal.

The terminal stores appointments in a map with the following values: latest arrival time, latest starting time, planned starting time, processing time, and expected departure time. When an appointment is added, the program makes sure that the appointment list is sorted on the latest starting time. This is necessary in order to construct waiting profiles. Furthermore, the terminal has a queue. Every barge that

arrives is added to the queue, even if the queue is empty. The terminal decides when a barge is removed from a queue and starts the handling process. The terminal has two policies of processing the queue. We refer to this as reserved and unreserved terminal logic. The terminal logics are further explained in Section 3.1.1.

Waiting profile

Terminals provide barges with information about the maximum amount of waiting time until the processing is started. This information is provided for every possible arrival moment during a certain time horizon in the from of a waiting profile. The WaitingProfile class is an implementation of the mathematical description of the construction of the waiting profile, see also Section 2.1.1. The class contains two tables, one to store the start intervals and one to store the actual waiting profile. We create methods to compute the content of these tables. The implementation is available in appendix B.5.

When the barge has the waiting profile of a terminal, it can get the maximum waiting time for any given time. We create a method called *getMaxWaitingTime*, which has as parameter the time on which the barge needs to know the maximum waiting time. It returns the maximum waiting time. This method contains an important part of the program, because in this method the slack is added. First, the method uses the waiting profile table to compute the maximum waiting time. Then, the slack is added to the waiting time.

Statistics

The Statistics class stores and manages the statistics of the simulation. The implementation is available in appendix B.4. The key statistics to collect are as follows: the number of barges that enter the port, the number of barges that leave the port, the expected sojourn time of each barge, and the actual sailing, waiting, and handling time of each barge. The program computes the mean, standard deviation, maximum and minimum of the expected sojourn times, actual sailing, waiting, and

handling times. The program stores the values in a table together with all parameter settings. This makes it possible to analyze which parameters are responsible for which results.

The Statistics class can optionally save all events of a simulation. The feature creates a table with three columns. The first column contains the time of the event, the second column contains the barge to which the event applies, and the third column contains a description of the event. We make this feature optional, because of the large number of events per simulation. When we run single simulations, it is interesting to observe the actual events that take place. When we run a large amount of simulations, we are more interested in the overall statistics.

The Statistics class can also optionally save all information from a barge. The feature creates a table containing the barge number, arrival time, terminals to visit with associated handling times, the route returned by the TDTSP, the waiting profiles, the appointment list, the appointment lists of the terminals at the moment of requesting the waiting profiles, the expected sojourn time, and the actual sailing, waiting, and handling times. We make the feature optional for the same reason as why the events are saved optionally.

After each simulation the program stores the key statistics and parameters, and optionally events and barge information, in a spreadsheet. This makes it easy to analyze the results. We present a part of the output in Section 3.4. A full output in form of a spreadsheet is available online via `http://goo.gl/OIIrRZ`.

TDTSP

The TDTSP is implemented in a seperate package. The classes are shown in Figure 3.6. The algorithm is described in Section 2.2.2. The implementation uses a recursive algorithm / backtracking technique and is partly based on an implementation of the eight queens problem as shown in [15]. In particular, the solve method in the TDTSP class and the use of a PartialSolution are inspired by [15].

Next, we describe the classes briefly. Since the classes are quite sophisticated,

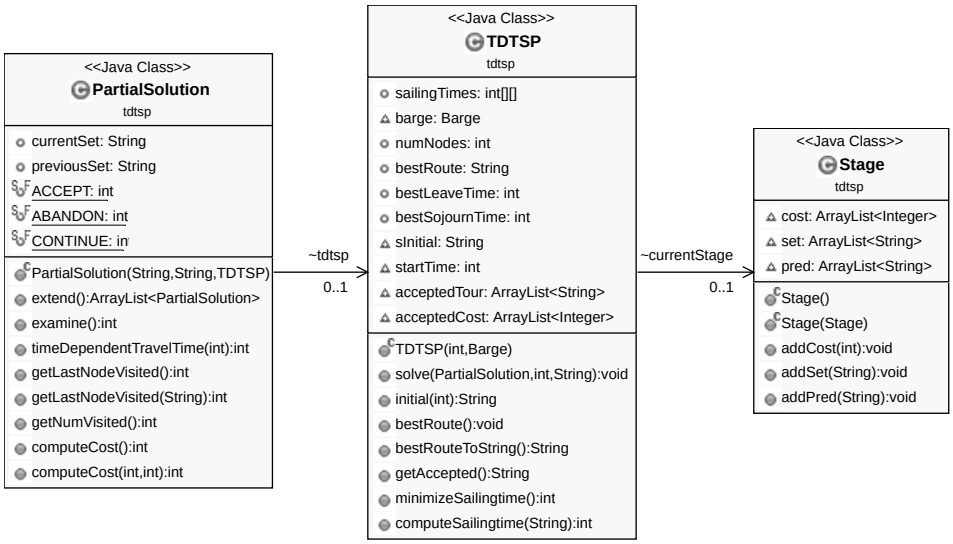


Figure 3.6: Class diagram of TDTSP package. The main class is the TDTSP class, which contains a Stage object to store the sets of the partial solutions together with the cost and the previous sets. The PartialSolution class contains a TDTSP object. We point out the extend method in the PartialSolution class, which adds nodes to current sets, and the solve method in the TDTSP class, which contains the recursive function.

we also refer to Appendix B.6 for the TDTSP class, Appendix B.7 for the Stage class, and Appendix B.8 for the PartialSolution class. Also, we provide an example in Section 3.4.3.

The TDTSP class is used by a barge to compute the best rotation and it also stores information about the rotation. As input it uses the barge and the arrival time in the port. From the barge object, it takes the sailings time table, which is the smaller version with only the terminals the barge has to visit. By using the sailing times table it deduces how many terminals the barge has to visit, thus for how many nodes the TDTSP needs to be solved. By using that information, it constructs an initial binary string. The initial binary string contains for each node a 0, plus five 0's to store the last node visited. For example, the initial binary string 0000*00000* is created for a barge that visits five terminals. The italic 0's can be converted to a decimal number to get the last node visited. The first five 0's represent five unvisited nodes, where the 0 on the first position on the left represents node 5.

When a node is visited, the 0 is set to a 1. The initial binary string is used as input for creating the first PartialSolution.

The PartialSolution class is used by the TDTSP class to store a part of the solution. It contains a binary string with the current set, and with the previous set. It also has a method to extend the partial solution, and a method to examine whether or not the current solution should be further extended, or that it has to be accepted.

The extend method adds all unvisited nodes to the current set. For example, if the current set is 0011100011 (i.e., node 4 and 5 are unvisited, last node visited is node 3), it adds the nodes 4 and 5. Adding node 4 means the new binary string becomes 0111100100 (i.e., node 5 is unvisted, last node visited is node 4). Adding node 5 means the new binary string becomes 1011100101 (i.e., node 4 is unvisted, last node visited is node 5). The two new binary strings are used to create a new PartialSolution. From there the process of extending repeats by using the recursive function inside the solve method of the TDTSP class. The solve method is recursive, which means that it calls itself. The function stops when the examine method in the PartialSolution class concludes that the PartialSolution can be accepted as final solution. This happens when the digits in the binary string representing nodes are all 1's.

The cost of the route, which is the time in the port, is recorded and stored during the extending of the routes inside the Stage class. The accepted routes are added to a list in the TDTSP class, together with the cost of the routes. When all accepted routes are in the list, it selects the route with the lowest cost. It then checks if there is more than one route with the lowest cost. If that is the case, then it selects the route with the least sailing time. In the exceptional case that there are routes with the same cost and the same sailing time, it selects the first one in the list, since both routes are equally good.

3.4 Sample simulations

The purpose of the simulation examples is to demonstrate what happens in a simulation. The purpose is not to give in-depth analysis of the results. This is done in Chapter 4. We describe two simulation runs in this section. First, we configure a deterministic model and add no slack to the waiting profile. In this way, the expected times of events should be the same as the actual times of events. Next, we configure a stochastic model and add slack to the waiting profiles. This creates more scheduling challenges, which means that the expected times of events are different from the actual times.

Next, we focus on a single barge. We show the actions and events of a barge in chronological order. We show what information a barge has, how it solves the TDTSP, how the terminals provide the waiting profiles, and how the barge moves through the system. After that, we show an example of how a terminal determines which barge to handle, by using both the reserved and unreserved terminal logic.

3.4.1 Sample overview

Table 3.2 shows the parameter settings and the descriptive statistics of both simulations. Note that the time deviation only applies to the stochastic model. As anticipated, we observe in simulation 1 that the expected sojourn time is equal to the actual sojourn time, and that in simulation 2 there is a difference. The progress of simulation 1 is shown is Figure 3.7, and the progress of simulation 2 is shown in Figure 3.8. The figures show the number of barges in the port, the number of barges in a queue, the number terminals busy (i.e., the number of barges handling), and the number of terminals in the port. Although the barges that arrive in both simulations are exactly the same, it is interesting to see that the progress of the simulations differ. Chapter 4 explains which parameters are responsible for the difference.

Table 3.2: The parameter settings and the descriptive statistics of the two example simulations.

Parameter	Simulation 1	Simulation 2
Random seed	651517117	651517117
Terminal logic	Reserved	Unreserved
Actual sailing and handling times	Deterministic	Stochastic
Run time	17280	17280
Warm-up time	2880	2880
Time deviation	0	3
Slack method	Constant	Constant
Slack constant	0	30
Mean terminals to visit	5	5
SD terminals to visit	1	1
Mean handling time	30	30
SD handling time	10	10
Arrival rate	14	14
Statistic	Simulation 1	Simulation 2
Barges entered the port after warm-up	1030	1030
Barges left the port after warm-up	1027	1034
Barges in port after warm-up	23	31
Mean expected sojourn time	365	353
SD expected sojourn time	69	65
Min expected sojourn time	112	120
Max expected sojourn time	561	559
Mean actual sojourn time	365	400
SD actual sojourn time	69	93
Min actual sojourn time	112	129
Max actual sojourn time	561	738
Mean waiting time	80	116
SD waiting time	58	65
Min waiting time	0	0
Max waiting time	308	390
Mean handling time	152	153
SD handling time	38	39
Min handling time	46	43
Max handling time	262	261
Mean sailing time	132	132
SD sailing time	27	26
Min sailing time	59	57
Max sailing time	219	211

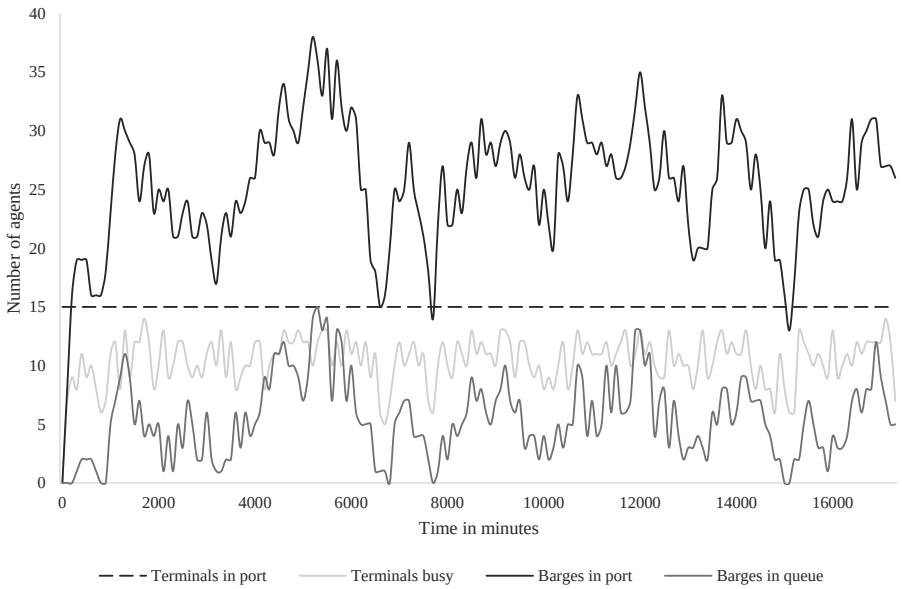


Figure 3.7: Progress by time of simulation 1.

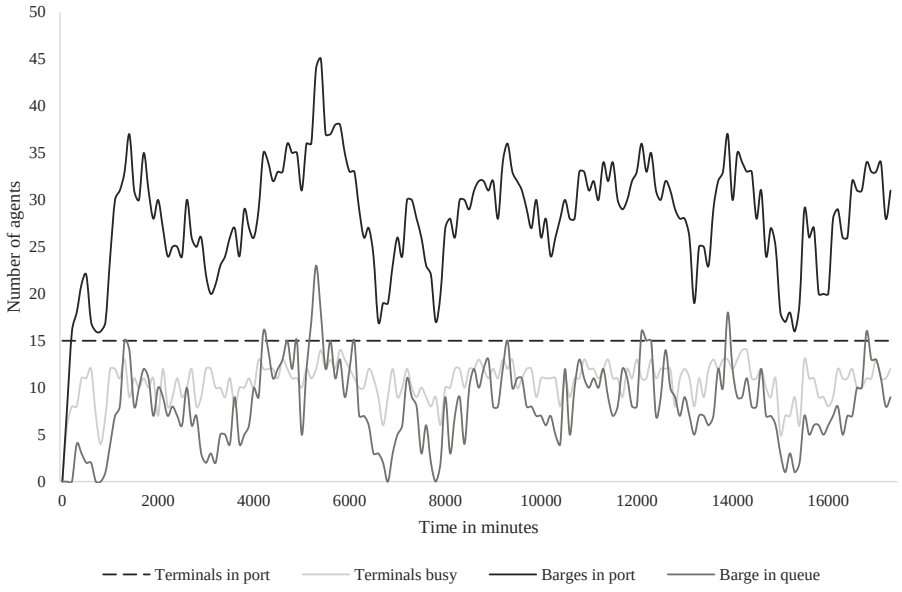


Figure 3.8: Progress by time of simulation 2.

3.4.2 Sample events

Since the number of events in only simulation 1 is already 17,561, we choose to only present a sample of the events in Table 3.3. It is a sample of 12 minutes of simulation 1. All events of both simulations are available in a spreadsheet online via `http://goo.gl/OIIrRZ`. Since simulation 1 uses deterministic values for the actual handling and sailing times, and adds no slack to the waiting profiles, there is no difference between the expected time of events and the actual time events occur. The arrival at the port and the finish of handling do not have expected times, therefore these events show no difference behind the event description.

Table 3.3: A 12 minute events sample of simulation 1. In this sample all 5 events are shown as described in Section 3.3.1: arrives at port, arrives at terminal, starts handling at terminal, finishes handling at terminal, and leaves port.

Time	Barge number	Event description
8,046	565	Arrived at Port
8,048	548	Finished handling at Terminal t8
8,048	562	Arrived at Terminal t7. Expected - actual = 8048 - 8048 = 0
8,050	555	Arrived at Terminal t6. Expected - actual = 8050 - 8050 = 0
8,050	555	Started handling at Terminal t6. Expected - actual = 8050 - 8050 = 0
8,051	559	Arrived at Terminal t4. Expected - actual = 8051 - 8051 = 0
8,051	545	Arrived at Terminal t13. Expected - actual = 8051 - 8051 = 0
8,051	545	Started handling at Terminal t13. Expected - actual = 8051 - 8051 = 0
8,052	549	Arrived at Terminal t3. Expected - actual = 8052 - 8052 = 0
8,052	549	Started handling at Terminal t3. Expected - actual = 8052 - 8052 = 0
8,052	563	Arrived at Terminal t2. Expected - actual = 8052 - 8052 = 0
8,053	543	Finished handling at Terminal t1
8,053	554	Arrived at Terminal t1. Expected - actual = 8053 - 8053 = 0
8,053	554	Started handling at Terminal t1. Expected - actual = 8053 - 8053 = 0
8,055	561	Arrived at Terminal t1. Expected - actual = 8055 - 8055 = 0
8,055	548	Arrived at Terminal t9. Expected - actual = 8055 - 8055 = 0
8,055	552	Left port. Expected - actual = 8055 - 8055 = 0
8,056	564	Arrived at Terminal t5. Expected - actual = 8056 - 8056 = 0
8,057	556	Finished handling at Terminal t11
8,058	548	Started handling at Terminal t9. Expected - actual = 8058 - 8058 = 0

3.4.3 Sample barge

To give a better understanding of how a barge is modeled, we provide information from a single barge in this section. All barges from both simulations are also available online in a spreadsheet. For each barge, we saved the barge number to keep it recognizable, the arrival time, the terminals to visit with corresponding handling times, the route, all waiting profiles and start interval tables, the list of appointments, the appointments of the terminals at the time of requesting the waiting profiles, and the statistics regarding the expected sojourn time, actual sojourn time, actual waiting, handling and sailing time.

In this section, we provide the information of one barge. We choose a barge that only visits two terminals, to keep the example relatively small. The barge is from simulation 2 and has barge number 875. We choose simulation 2, because it uses stochastic values for the actual handling and sailing times, and it also contains slack. The following data was generated for this barge. The arrival time at port is 12,203. The terminals to visit are t10 and t2 with expected handling times of 33 and 17 minutes respectively.

Creating waiting profiles

The first action of the barge is to request waiting profiles. The terminals respond to this request by creating the waiting profiles. Terminal t10 creates the waiting profile as follows. It first constructs a start intervals table containing all intervals in which the barge can start handling. The start intervals define where in the schedule of the terminal the handling time of the barge fits. The table contains 3 columns: the first column contains the start interval number, the second column contains the start time of the interval, and the last column contains the end time of the interval. If a barge arrives within the start interval, then the expected waiting time is zero. To compute the start and end times, the terminal agent uses its current appointments. The appointments of terminal t10 and t2 are given in Table 3.4.

A start interval is only saved if it is feasible, i.e., the start time is smaller than

Table 3.4: Appointments of terminal t10 and t2 at time 12,203 in simulation 2. Note that barge 875 is shown in this table, but in the actual situation it is added after solving the TDTSP, which we show in the next section. The time units are minutes. LAT=latest arrival time, LST=latest starting time, PST=planned starting time, PT=processing time, EDT=expected departure time.

Terminal	Barge	LAT	LST	PST	PT	EDT
t10	851	12,152	12,152	12,152	39	12,191
	854	12,186	12,191	12,191	47	12,238
	861	12,196	12,238	12,238	45	12,283
	862	12,291	12,291	12,291	51	12,342
	870	12,365	12,365	12,365	24	12,389
	869	12,416	12,416	12,416	40	12,456
	875	**12,364**	**12,456**	**12,456**	**33**	**12,489**
t2	863	12,190	12,190	12,190	13	12,203
	869	12,166	12,203	12,203	12	12,215
	865	12,251	12,251	12,251	42	12,293
	871	12,176	12,293	12,293	38	12,331
	875	**12,235**	**12,331**	**12,331**	**17**	**12,348**

the end time and the start time is larger than the arrival time of the barge at the port. The start time of start interval 1 is equal to the arrival time of the barge in the port. The start time of a start interval 2 is equal to the expected departure time (EDT) of the appointment on row 1. The start time of a start interval 3 is equal to the EDT of the appointment on row 2, and so on. For terminal t10 this means that the start time of start interval 1 is 12,203, and the start times of interval 2 to 10 are the EDT's from Table 3.4 (12,191 to 12,489).

The end time of start interval 1 equals the planned starting time (PST) of the appointment on row 1, which is 12,152, minus the handling time at t10, which is 33 minutes. So the end time of start interval 1 equals 12,119. This is not feasible, therefore it is not added to the table. The end time of start interval 2 equals the planned starting time (PST) of the appointment on row 2, which is 12,191, minus the handling time at t10. So the end time of start interval 2 equals 12,158. This continues for all start intervals, except for the last. The end time is set to infinite

for the last start interval (we use the maximum integer value in our program). This means that the barge can make an appointment at any time after the EDT of the last barge in the current schedule, with an expected waiting time of zero. The last step of creating the start intervals table is checking if start intervals are disjoint, i.e., the end time is larger than the start time of the next start interval. If that is the case, the intervals are merged. Terminal t10 has only one feasible start interval for barge 875.

Table 3.5: Start intervals of Terminals t10 and t2. Note that only the feasible start intervals are shown (i.e., the start time is smaller than the end time and the start time is larger than the arrival time of the barge at the port). The only possible slot where a barge with handling time of 33 minutes can be added to the schedule of terminal t10 is after the last appointment. That is why it only shows 1 start interval for t10. For terminal t2 there are four slots possible.

Terminal	Start interval	Start time	End time
t10	7	12,456	∞
t2	3	12,215	12,234
	5	12,331	∞

Terminal t2 does the same operations as terminal t10, only then it uses its own appointments table. The start intervals for both terminals are shown in Table 3.5. Next, the terminal uses the start intervals to construct the waiting profile. A waiting profile table contains two columns. The first column contains the start time, and the second column contains the waiting time.

The start time of the first row of the waiting profile equals the arrival time of the barge. The start time of the second row of the waiting profile equals the end time of the first start interval. The start time of the third row of the waiting profile equals the end time of the second start interval, and so on. Only the last row of the start intervals table is not used in the waiting profile, because the end time of the last row of the start intervals table is never reached.

The waiting time of the first row equals the start time of the first row of the start interval table, minus the arrival time. The waiting time of the second row of the

waiting profile, equals the start time of the second row of the start interval table, minus end time of the first row of the start interval table. The row is only added to the waiting profile, if the waiting time is larger than 0. The waiting profile table is shown in Table 3.6.

Table 3.6: Waiting profile of Terminals t10 and t2.

Terminal	Start time	Waiting time
t10	12,203	253
t2	12,203	12
	12,234	97

Terminal t10 only has one start interval. It shows that the earliest time of handling is on time 12,456. This means that the waiting time is $(12,456 - 12,203 =)$ 253 minutes. Terminal t2 has four start intervals. The first arrival time in the waiting profile should be the arrival time in the port, which is 12,203. The waiting time equals the start time of the first start interval, which is also 12,203, minus the arrival time. This means that waiting time is 0, that is why it is not added to the waiting profile. The second arrival time in the waiting profile should be the end time of the first start interval, which is 12,203. The waiting time equals the start time of the second start interval, which is 12,331, minus the end time of the first start interval, which is 12,203. So the waiting time is $(12,331 - 12,234 =)$ 97 minutes. The process repeats for the rest of the start intervals.

Solving TDTSP

After receiving the waiting profiles of the terminals, the barge constructs the smaller version of the sailing times table with only the terminals it has to visit. This is the last input required for solving the TDTSP. The sailing times table also contains the entrance/exit point of the port, which is t0. The sequence of the terminals in the sailing times table is important, because it is used to trace back the solution of the TDTSP, which uses node numbers instead of terminal numbers. The sequence is the same as the sequence in which the terminals are assigned to the barge. For

barge 875 the sequence is: t10, t2. Therefore, node 1 represents t10, and node 2 represents t2. Node 0 always represents t0. The sailing times table is shown in Figure 3.7.

Table 3.7: Sailing times between the terminals barge 875 has to visit in minutes.

	t0	t10	t2
t0	0	28	32
t10	28	0	16
t2	32	16	0

The TDTSP object of the barge manages the computations. First, it creates the initial set, consisting of the number of terminals to visit in 0's, plus five 0's to keep track of the last node visited. For barge 875 it is 0000000 (meaning node 1 and 2 unvisited, last node visited node 0). The initial set is used to create the first partial solution, where the initial set is assigned to both the current set and previous set. The first partial solution is used, together with the arrival time at the port (recall that it is 12,203), as input for the solve method, which contains the recursive function.

The solve method first computes the time dependent travel time (TDTT) of the first partial solution. Since both the current and previous set are the same, the TDTT is 0. The solve method keeps track of the cost of a route by using a current time variable. The TDTT is added to the current time variable, which thus stays 12,203. The time variable and the set (which is still the initial set) are stored. Next, the solve calls itself, with the partial solutions that are created by calling the extend method on the initial partial solution as input.

By calling the extend method on the initial partial solution, the current set is extended with one node in every direction. This means that it goes from node 0 to node 1, and from node 0 to node 2. This creates two new sets: 0100001 (meaning node 1 visited, node 2 unvisited, last node visited node 1) and 1000010 (meaning node 2 visited, node 1 unvisited, last node visited node 2), respectively. For each new set a new partial solution is created, where the new set is assigned to the current

set, and the initial set is assigned to the previous set. We refer to this as the second and third partial solution.

Starting with the second partial solution, the solve method computes the TDTT of going from node 0 to node 1. Recall that the TDTT is the sum of the sailing time, handling time, and waiting time. The sailing time is 28 minutes, see the Table 3.7. The handling time at node 1 (which is terminal t10) is 33 minutes. The waiting time is computed by using the waiting profile. For this it needs as input the arrival time. The arrival time at the terminal is the departure time plus the sailing time, which is $(12,203 + 28 =)$ 12,231. Starting from the bottom of the waiting profile, it checks if the arrival time is larger then the start time of the waiting profile. Since the waiting profile of t10 only has one row with start time the arrival time at the port, this is the case at the first check. The corresponding waiting time is 253 minutes. Since the barge arrives later than the start time of the waiting profile, the waiting time should be adjusted. The adjusted waiting time is the waiting time from the waiting profile plus the start time of the waiting profile minus the arrival time, which is $(253 + 12,203 - 12,231 =)$ 225 minutes. At this point, the slack is added. Recall that simulation 2 uses a constant slack of 30 minutes. The adjusted waiting time becomes 255 minutes. Therefore, the TDTT of going from node 0 to node 1 is $(28 + 33 + 255 =)$ 316 minutes. The TDTT is added to the time variable, which becomes $(12,203 + 316 =)$ 12,519. The time variable and the set are stored. The set cannot be accepted yet, because there is one more unvisited node. So it continues, by calling its own method, with the partial solution (only one this time) that is created by calling the extend method on the second partial solution.

It goes from node 1 to node 2. This creates one new set: 1100010 (meaning node 1 and 2 visited, last node visited node 2). A new partial solution is created, where the new set is assigned to the current set, and the current set is assigned to the previous set. The new partial solution is used as input for the solve method, we refer to this as the fourth partial solution. Note that the algorithm starts working with the fourth partial solution, before it starts working with the third partial

solution. This is because the solve function first needed to finish the second partial solution, from which arose the fourth partial solution, i.e., the second partial solution includes the fourth partial solution.

The solve method computes the TDTT of the fourth partial solution. The sailing time is 16 minutes. The handling time at node 2 (which is terminal t2) is 17 minutes. The waiting time is computed by using the waiting profile of t2. The arrival time at t2 is the departure time plus the sailing time, which is $(12,554+16=)$ 12,570. Starting from the bottom of the waiting profile, it checks if the arrival time is larger than the start time of the waiting profile. The bottom start time in the waiting profile table is 12,234, so it will use this to compute the waiting time. The corresponding waiting time is 97 minutes. The adjusted waiting time $(97+12,234-12,570)$ is smaller than 0. If the waiting time is smaller than 0, it becomes 0. The slack will also be 0, since it is computed by a multiplication of the adjusted waiting time. The TDTT of going from node 0 to node 1 is $(16+17=)$ 33 minutes. It adds the TDTT to the time variable, which becomes $(12,570+33=)$ 12,603. Then, it checks if the route is complete, i.e., all nodes are visited. Both node 1 and 2 are visited, so this is true. Now it returns to the port entrance/exit, by adding the sailing time from node 2 to node 0 (which is 32 minutes) to the time variable. The return time of the route becomes $(12,603+32=)$ 12,635. To get the cost (or expected sojourn time) of the route, the return time is subtracted from the arrival time in the port. The cost is $(12,635-12,203=)$ 432 minutes.

After the route was accepted, the algorithm continues with the third partial solution. The process of the third partial solution is the same as the second partial solution. It goes from node 2 to node 1. The process of the fifth partial solution is the same as that of the fourth. When all accepted routes are stored, the one with the lowest expected cost is chosen. The route is: 0 2 10, with expected cost of 314 minutes. The barge makes appointments on basis of this route. The appointments are shown in Table 3.4.

To give more insight in how the partial solutions develop, we have added Ta-

ble 3.8. A sample is shown of the progress of a TDTSP of a barge that visits 6 terminals. The initial node went from node 0 to node 1, 2, 3, 4, 5, and 6. The output shows that the algorithm is a depth-first search algorithm. The full table has 1,957 entries and is available online. The purpose of the table is to show how the algorithm develops.

Table 3.8: TDTSP sample of a random barge that visits 6 terminals.

Set	Route	Cost
00000000000	0	0
00000100001	0 1	47
00001100010	0 1 2	76
00011100011	0 1 2 3	110
00111100100	0 1 2 3 4	163
01111100101	0 1 2 3 4 5	230
11111100110	0 1 2 3 4 5 6	287
10111100110	0 1 2 3 4 6	203
11111100101	0 1 2 3 4 6 5	272
01011100101	0 1 2 3 5	167
01111100100	0 1 2 3 5 4	238
11111100110	0 1 2 3 5 4 6	278
11011100110	0 1 2 3 5 6	224
11111100100	0 1 2 3 5 6 4	280
10011100110	0 1 2 3 6	149

Moving through the port

Next, we discuss how barge 875 moves through the port by describing each event. We filter the spreadsheet discussed in Section 3.4.2 on barge 875. The result is shown in Table 3.9.

After arriving at the port, the barge visits terminal t2. Since simulation 2 uses stochastic values for the sailing time and handling time, the actual sailing time is different from the sailing time stated in the sailing times table. The expected sailing

time is 32 minutes, while the actual sailing time is 26 minutes. The handling could start at 12,296. This means that the terminal was busy at arrival. Still, the handling could start 35 minutes before the latest starting time. The expected handling time at t2 is 17 minutes, while the actual handling time is $(12,308 - 12,296 =)$ 12 minutes. After finishing at t2, the barge visits t10. The expected sailing time is 16 minutes, while the actual sailing time is $(12,322 - 12,308 =)$ 14 minutes. Since the handling does not start at the arrival time, we know that this terminal is busy. The barge has to wait $(12,435 - 12,322 =)$ 113 minutes before the handling starts. The expected handling time at t10 is 33 minutes, while the actual handling time is $(12,466 - 12,435 =)$ 31 minutes. After finishing at t2, the barge sails to the port entrance/exit point. The expected sailing time is 32 minutes, while the actual sailing time is $(12,493 - 12,466 =)$ 27 minutes.

Table 3.9: The events of barge 875 in simulation 2. With *expected* we mean the expected time in the appointments table, and the with *actual* we mean the actual time of the event. The difference between the expected and actual sailing and handling times are given in the paragraph above the table.

Time	Event description
12,203	Arrived at Port
12,229	Arrived at Terminal t2. Expected - actual = 12235 - 12229 = 6
12,296	Started handling at Terminal t2. Expected - actual = 12331 - 12296 = 35
12,308	Finished handling at Terminal t2
12,322	Arrived at Terminal t10. Expected - actual = 12364 - 12322 = 42
12,435	Started handling at Terminal t10. Expected - actual = 12456 - 12435 = 21
12,466	Finished handling at Terminal t10
12,493	Left port. Expected - actual = 12517 - 12493 = 24

Because the barge did not have to wait for its appointment, it was possible that the handling could start earlier than planned at both terminals. The barge leaves the port 24 minutes earlier than expected. This means that barge 875 benefits from the unreserved policy. Note that this does not mean that the policy is beneficial for all barges, as pointed out in Section 3.1.1. In the next chapter, we analyze the results

of the whole system more in-depth.

3.5 Information correctness and waiting time satisfaction

According to [1] waiting time has four aspects: objective, subjective, cognitive, and affective. The objective aspect is the actual elapsed time. The subjective aspect is the estimated time of waiting by a customer, which is related to the objective time. The cognitive aspect is how the customer evaluates the waiting time. For example, it can be acceptable, reasonable, or tolerable. The affective aspect consists of the emotional response, such as irritation, boredom, frustration, stress, pleasure, or happiness. In [17] the authors propose a formula to measure satisfaction of a service: $S = P - E$, where S is satisfaction, P is perception and E is expectation. This means that if the client expects a certain level of service, and the client perceives the service as higher, then the client is satisfied. We use the information from [1] and [17] to measure the information correctness and waiting time satisfaction.

3.5.1 Information correctness satisfaction

We translate this as follows in the measurement of the information correctness satisfaction. We are dealing with the sojourn time of barges in the port, instead of only the waiting time. We argue that this is justified, because the sojourn time in the port is related to the waiting time. We examine the information correctness satisfaction by comparing the expected and actual sojourn time of barges in the port. The expected sojourn time is based on the information provided by terminals. We treat the actual sojourn time as the perception of the service, and the expected sojourn time as the expectation of the service. In contrast to the formula in [17], a lower number of the perception (i.e., actual sojourn time) means a higher service. Therefore, the formula becomes: satisfaction = expected sojourn time - actual sojourn time. We use the objective aspect of the sojourn time, since we are conducting a simulation study. We link the outcome of the formula to the cognitive and affective aspects of

the sojourn time. We do this by using the utility function shown in Figure 3.9.

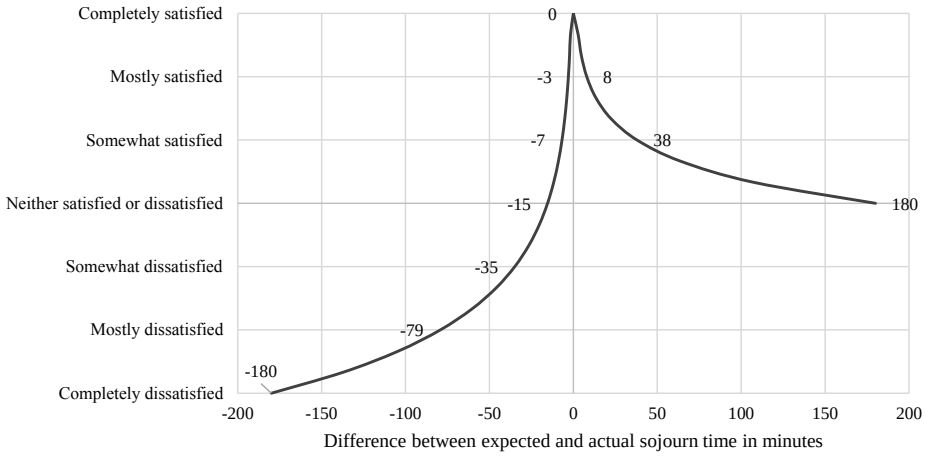


Figure 3.9: Utility graph used for measuring the satisfaction level of the information correctness regarding the expected sojourn time.

If the expected sojourn time is the same as the actual sojourn time, then a barge is completely satisfied. This would be the ideal scenario, since a barge can schedule his next activities more precisely after its visit to the port. If a barge finishes earlier than expected, it will still be satisfied with the information correctness, since it would still be in time for its activities after the visit to the port. However, if the difference increases, then the satisfaction will decrease. Otherwise, adding a huge amount of slack would always result in a satisfaction with the information correctness. We argue that a barge will not be dissatisfied with the information correctness when it finishes earlier than expected, because it will not have negative consequences, such as being late for a next appointment. This in contrast to the scenario in which a barge finishes later than expected. If a barge leaves later, it will be more dissatisfied, because the probability that he misses a next appointment (outside the port) increases.

We determined the points in the utility graph by setting a threshold value for which the satisfaction level reaches the lowest point. We set this at 3 hours (180 minutes), because in our opinion if the expected sojourn time differs more than 3

hours from the actual sojourn time, then the information correctness is poor. We use this estimation, since no previous literature has investigated this. With this estimation, we draw a convex utility curve, on basis of which we determine the values that belong to each rating. The ratings are acquired from a 7-point Likert scale that measures customer satisfaction. Table 3.10 summarizes the ratings with their corresponding intervals.

3.5.2 Waiting time satisfaction

To measure the waiting time satisfaction, we compare the waiting time with the handling time. We assume a barge is completely satisfied if its total waiting time is not more than 60% its total handling, and is completely dissatisfied if its total waiting is twice its total handling. Between *completely satisfied* and *completely dissatisfied* we draw a convex utility curve, on basis of which we rate the waiting time. Figure 3.10 shows the utility curve. Table 3.10 summarizes the ratings with their corresponding intervals.

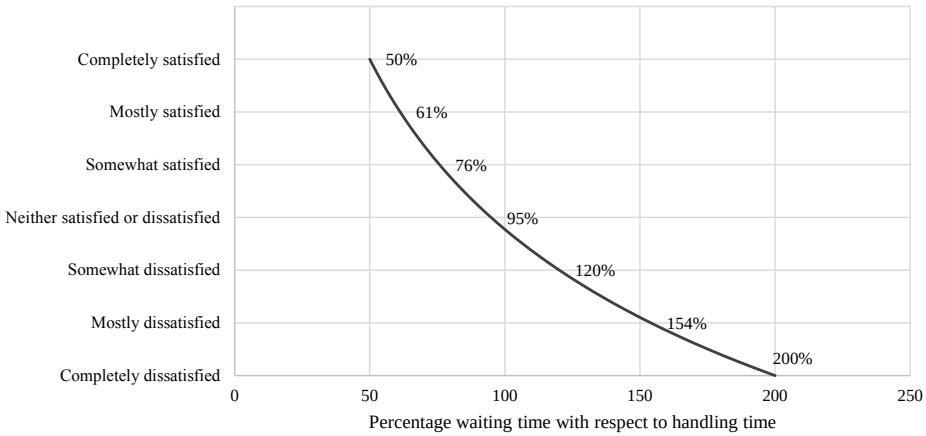


Figure 3.10: Utility graph used for measuring the satisfaction level of the waiting time.

We link the satisfaction level with the cognitive and affective aspects of the waiting and sojourn time. We use a straightforward method, which means that we replace *satisfied* with *acceptable* for the cognitive aspect. For the affective aspect

Table 3.10: Satisfaction levels with corresponding intervals. *Minutes earlier* is used in case the actual time in shorter than the expected time. *Minutes later* is used in case the actual time is longer than the expected time.

	Information correctness		Percentage waiting time
Rating	Minutes earlier	Minutes later	with respect to the handling time
Completely satisfied	0 – 7	0 – 2	0 – 60
Mostly satisfied	8 – 37	3 – 6	61 – 75
Somewhat satisfied	38 – 179	7 – 14	76 – 94
Neither satisfied or dissatisfied	>180	15 – 34	95 – 119
Somewhat dissatisfied	–	34 – 78	120 – 153
Mostly dissatisfied	–	79 – 179	154 – 199
Completely dissatisfied	–	>180	>200

we use a happiness scale, which goes from completely happy, to completely upset. It is important to have happy customers, since it affects their future behavior [9]. In this case, this means that barges probably would look for other ports if they are not satisfied with the service.

Chapter 4

Results

This chapter analyzes the results of the simulation experiments. In order to answer the research question, we have to examine which policy works best with which slack settings. After that, we compare the policies on the impact on the sojourn time, predictive power, and the influence on barge (or customer) satisfaction with regard to the correctness of the information and the waiting time.

4.1 Simulation statistics

We conduct a total of 256 simulations, which are made by combining the policy settings with the scenario settings. Table 4.1 gives a summary of the settings. We collect statistics from barges that left the port after the warm up period. The number of barges per simulation that left the port varies between 846 and 1,230, with a total of 264,816 barges in all simulations, a mean of 1,034 per simulation and a standard deviation of 94 barges. We collect the average, standard deviation, minimum and maximum of the expected sojourn time, actual sojourn time, actual waiting time, actual handling time, and actual sailing time. The statistics of all simulations are available online in a spreadsheet. Next, we explain what we do with the random seeds and deterministic/stochastic values for the actual sailing and handling time.

Table 4.1: Simulation overview. The total number of simulations is obtained by multiplying the number of settings per parameter.

Parameter	Number	Settings
Terminal logic	2	reserved, unreserved
Slack method	4	constant slack (0, 30, 60, 90)
Sailing and handling times	2	deterministic, stochastic
Arrival rate	4	12, 13, 14, 15 minutes per barge
Random seed	4	4 different random seeds
Total simulations	256	

4.1.1 Random seeds

Table 4.2: Average sojourn time in minutes and average number of barges per simulation. Averaged over 64 simulations per random seed. The actual and expected sojourn time is averaged over the barged that left the port. With *barges left the port* we mean barges that left the port after the warm-up period. With *barges entered* we mean barges that entered the port after the warm-up period. With *starting barges* we mean barges that are in the port at the end of the warm-up period.

Random seed	Expected sojourn time	Actual sojourn time	Barges left the port	Barge entered	Starting barges
166267832	485	523	10,41	1,099	24
284324984	460	475	976	988	30
489258742	548	691	1,069	1,110	26
651517117	500	570	1,052	1,086	27
Average	499	567	1,034	1,071	27

The random seed influences the expected and actual sojourn, as shown in Table 4.2. An explanation for this is that the number of barges in the port depends on a random number generator, and that barges arrive according to an exponential distribution. That results in different peak times in the port. In our analysis, we compare the policies on all four seeds together. That leads in our opinion to the

best comparison, where the randomness of the distribution is leveled.

4.1.2 Deterministic and stochastic

We conduct a test to compare the use of deterministic values and stochastic values for the actual sailing and handling times in our simulations. We use a two-sample t-test with a significance level of 5%. Table 4.3 shows the input for the t-tests, where n is sample size (i.e., number of barges), $\bar{x}$ is the sample mean (i.e., mean sojourn time), and σ is the standard deviation.

First, we test if the difference in actual sojourn time is significant. The null and alternative hypotheses are as follows: $H_0 : \mu_{deterministic} = \mu_{stochastic}$; $H_a : \mu_{deterministic} \neq \mu_{stochastic}$. The two-sample t-test shows that there is a significant difference in actual sojourn time when using deterministic values ($\bar{x}_{actual} = 512$, $\sigma_{actual} = 228$) and stochastic values ($\bar{x}_{actual} = 623$, $\sigma_{actual} = 288$); t-difference= -155.368; df-t= 499873.6; p-value= 0. We reject H_0, since the p-value is smaller than the significance level.

Next, we test the significance for the expected sojourn time. The null and alternative hypotheses look the same as the previous. The two-sample t-test shows that there is a significant difference in expected sojourn time when using deterministic values ($\bar{x}_{expected} = 496$, $\sigma_{expected} = 120$) and stochastic values ($\bar{x}_{expected} = 502$, $\sigma_{expected} = 124$); t-difference= -17.891; df-t= 528427.5; p-value= 0. We reject H_0, since the p-value is smaller than the significance level.

Table 4.3: To get the sample size n we take the sum of the number of barges that left the port over the simulations (256 simulations each). To get the sample mean $\bar{x}$, we divide the total sojourn of the barges by n. To get the standard deviation σ, we take the square root of the average variance.

	n	$\bar{x}_{actual}$	σ_{actual}	$\bar{x}_{expected}$	$\sigma_{expected}$
Deterministic	133,396	512	228	496	120
Stochastic	131420	623	288	502	124
Total	264,816	567	259	499	122

We can conclude that the use of stochastic values for the actual handling and sailing times results in significant differences in the expected and actual sojourn in the port for barges. In the rest of our analysis, we only focus on the stochastic values, because it is believed to be more similar to the real situation. We also think that it creates more scheduling challenges, making the policies more interesting. For example, when using deterministic values and a policy that adds no slack to the waiting profiles, and uses reserved terminal logic for processing barges, then the actual time of events is exactly the same as planned. This is shown in an example in previous chapter. In our opinion, this makes the simulation less interesting and perhaps unnecessary. Therefore, we choose to further neglect the simulations that use deterministic values, and only analyze the simulation that use stochastic values.

4.2 Minimal sojourn time and predictive power

We investigate which policy results in the least average actual sojourn time for each arrival rate. We also investigate which policy leads to the best prediction of the sojourn time, i.e, which policy leads to the least average difference between the actual and expected sojourn time. First, we search for the best settings for each policy at each arrival rate. After that, we compare the best settings of each policy using a two-sample t-test in order to choose the best policy. We have included Table A.1 in Appendix A.1, which contains the sample mean, standard deviation minimum and maximum of the expected and actual sojourn time. It also contains the information about the actual waiting, handling and sailing time. We use this information in our analysis.

For each arrival rate, we select the two slack settings that result in the lowest sample mean for the actual sojourn time and compare them with a two sample t-test. Next, we select the two slack settings that result in the least difference between the actual and expected sojourn time, and also compare them with a two sample t-test. We get the input for the t-tests from the table in the Appendix A.1. Table 4.4

shows the input and the outcome of the t-tests. For the setup and explanation of the t-test, with hypotheses and an example, we refer to Appendix A.1. Next, we discuss the preferred slack setting per policy.

Table 4.4: Two-sample t-test input and output. We use this to choose the best slack setting for each policy and arrival rate. We use a significance level of 5%. The sample mean, standard deviation and sample size are extracted from Table A.1. With left parameters is meant the left slack setting in the compare column; with right parameter is meant the right slack setting in the compare column. Recall that constant slack has the parameters 0, 30, 60, and 90. The p-values are calculated with an online tool [14].

				Left parameter			Right parameter				
	Arrival rate	Test	To compare	$\bar{x}$	σ	n	$\bar{x}$	σ	n	p	Conclusion
Reserved and constant slack	12	Actual time	0 and 90	941	274	4507	721	316	4580	0.0001	choose 90
		Best predictor	30 and 90	269	486	4404	-40	349	4580	0.0001	choose 90
	13	Actual time	30 and 90	516	177	4303	513	180	4359	0.4342	indifferent
		Best predictor	30 and 60	1	210	4303	-66	216	4292	0.0001	choose 30
	14	Actual time	30 and 60	399	103	4043	426	123	4040	0.0001	choose 30
		Best predictor	30 and 60	-78	136	4043	-161	167	4040	0.0001	choose 30
	15	Actual time	30 and 60	374	94	3759	394	111	3758	0.0001	choose 30
		Best predictor	30 and 60	-87	128	3759	-179	159	3758	0.0001	choose 30
Unreserved and constant slack	12	Actual time	0 and 30	582	211	4707	607	222	4709	0.0001	choose 0
		Best predictor	30 and 60	62	248	4709	5	269	4694	0.0001	choose 60
	13	Actual time	0 and 30	470	150	4374	467	149	4366	0.3483	indifferent
		Best predictor	0 and 30	74	177	4374	-35	176	4366	0.0001	choose 30
	14	Actual time	0 and 30	406	112	4047	414	113	4040	0.0014	choose 0
		Best predictor	0 and 30	51	134	4047	-66	142	4040	0.0001	choose 0
	15	Actual time	0 and 30	368	94	3756	375	91	3767	0.001	choose 0
		Best predictor	0 and 30	36	114	3756	-88	124	3767	0.0001	choose 0

4.2.1 Reserved policy

From the t-tests, we conclude that the best slack settings for this policy is 90 minutes for arrival rate 12, and 30 minutes for arrival rate 14 and 15. These slack settings result in both the least actual sojourn time, as well as the least difference between the actual and expected sojourn time. For arrival rate 13 the difference in using a slack of 30 and 90 minutes is not statistical significant. Since using a slack of 30 minutes results in the best prediction, it can be argued that using 30 minutes is preferred for arrival time 13. The results for this policy match the results of [11], which uses this policy solely. The only difference is that their model is deterministic, and they do not include the slack of 90 minutes.

The explanation for the benefit of using slack is that it increases the planning flexibility. With higher arrivals, more flexibility is required. The difference between the actual and expected sojourn time is explained by the way barges compute their expected time; they assume they have to wait the maximum amount of time, which includes the slack. The reason why a constant slack of 0 (i.e., no slack) still results in a difference between the actual and expected sojourn time is the use of stochastic values for the actual sailing and handling time.

4.2.2 Unreserved policy

From the t-tests, we conclude that the best slack settings for this policy is to use a slack of 0 minutes. For arrival rate 12, we observe an advantage for using 60 minutes as predictor. For arrival rate 13, we observe that using 0 and 30 minutes results in an statistical insignificant difference, and that using 30 minutes results in the best predictions. However, the predictions are still poor, and since a slack of 0 minutes results in the least actual sojourn time for arrival rates 12, 14 and 15, we argue that this should also be used for arrival rate 13.

That adding no slack to the waiting profiles for this policy leads to good results, can be explained by the fact that terminals process queues on basis of the FIFO system. Therefore, they do not have to create more planning flexibility, since

appointment are not strict. In addition, we see small differences in the actual sojourn time when adding more slack to the waiting profile. The reason for this also that barges are processed on basis of the FIFO system. The only large consequence of adding slack is that the expected sojourn time increases.

In Figure 4.1, the expected and the actual sojourn time per policy is shown. Overall, we can conclude that unreserved terminal logic with no slack added to the waiting profiles has the most potential. Next is the reserved terminal logic with a constant slack of 30 minutes added to the waiting profiles.

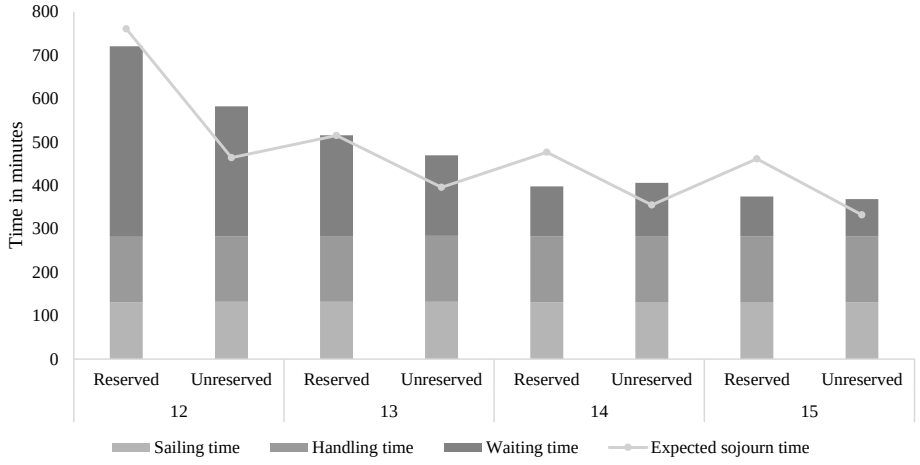


Figure 4.1: Comparison per policy per arrival rate. Note that higher arrival rates mean a more busy port. This is clearly visible in this graph, where arrival rate 12 (which means a barge arrives every 12 minutes on average) has the highest waiting times, while arrival rate 15 has the lowest waiting times.

4.2.3 Comparing the best policies

We compare the reserved and unreserved policy on their average expected and actual sojourn, while also considering the maximum sojourn time. On basis of the previous results, we select the best slack settings for each arrival rate policies to analyze further. A reserved policy with a slack 90 minutes for arrival rate 12, and 30 minutes for arrival rate 13, 14, and 15. An unreserved policy with a constant slack of 0 minutes (i.e., no slack) for arrival rate 12 to 14. Figure 4.2 shows the

mean and maximum of the expected and actual sojourn time per arrival rate per policy.

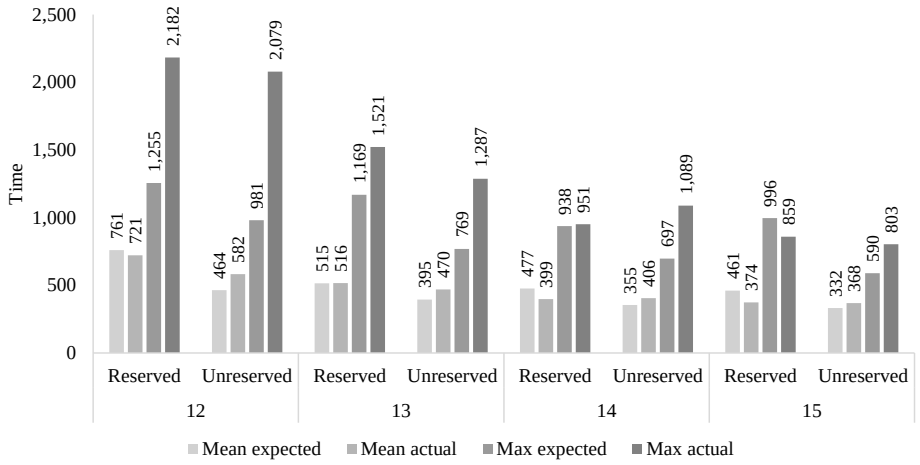


Figure 4.2: Expected and actual sojourn time. This graph contains the mean and maximum values. The more busy scenarios (12, 13) have higher sojourn times, then the more quiet scenarios (14, 15).

For arrival rate 12 and 13 the best predictor is to use reserved terminal logic, while the least sojourn time will be achieved when using unreserved terminal logic. The worst-off barge has a lower actual sojourn time when using a unreserved policy. For arrival rate 14, the unreserved policy, while the reserved policy leads to a slightly better actual sojourn time. Also, the worst-off barge has a lower sojourn time in case of using a reserved policy. For arrival rate 15, the unreserved policy has a slightly advantage. Since this is the most quiet scenario, there will be less waiting, making the impact of a policy change less visible. Note that the means are same as used in previous significance tests, and that we cannot conduct a significance test for the worst-off situation, because those are maximum values of the samples, instead of a sample means.

The intuitive reason why unreserved terminal logic works better than reserved terminal logic is that barges do not have to wait at a terminal when the terminal is idle. This leads to less waiting time on average for the whole system.

4.3 Individual barge satisfaction

This section presents the results of the satisfaction with the information correctness and waiting time. Recall that the satisfaction measures are described in Section 3.5. For the examination of the barge satisfaction, we run 32 new simulations, which makes it possible to collect the information correctness satisfaction and waiting time satisfaction of every individual barge. The following settings are used in the simulation. All simulations use the random seed (284324984). The actual sailing and handling times are stochastic. The arrival rates are 12 to 15 minutes per barge. The slack added to the waiting profile is 0, 30, 60, and 90 minutes. The policies are reserved and unreserved.

In each simulation, we collect the statistics of exactly 500 barges. Since the simulations use the same random seed, a barge with a specific number visits the same terminals and has the same handling times in every simulation. Also, in case of the same arrival rate , the barge has the same arrival time in the port. This makes it possible to evaluate the policy on individual barge level. The data is available online in a spreadsheet and in a SPSS data document.

4.3.1 Information correctness satisfaction

We select the slack setting that results in the most satisfied customers with regard to the information correctness for both the reserved and the unreserved policy for each arrival rate. We did a comparison from which the results suggest that a slack of 30 minutes leads to the most satisfied customers for both the reserved and unreserved policy for each arrival rate. The comparison is shown in Appendix A.2. Figure 4.3 shows the number of barges per satisfaction level. It shows that there are slightly more barges satisfied with the information correctness in the reserved policy cases, but overall it looks similar. Furthermore, we observe a large majority that is *somewhat satisfied*. The reason for this is that the expected sojourn time differs too much from the actual sojourn time, and in a large majority of cases

the expected sojourn time is higher than the actual sojourn time. Since a higher expected sojourn time could not lead to dissatisfied customers, as argued in Section 3.5.1, it often results in somewhat satisfied. The reason why the expected time is often higher than the actual time, is because in all cases there is slack added to the waiting profiles, which makes the expected sojourn time higher.

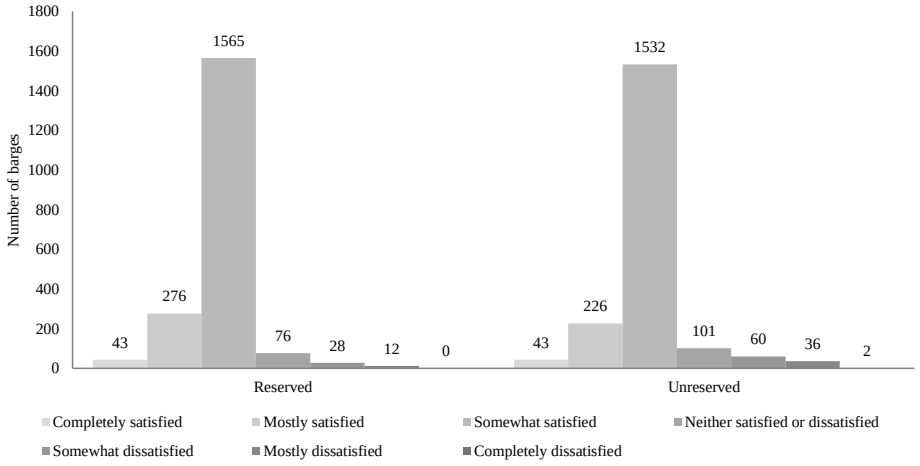


Figure 4.3: The results of the information correctness satisfaction. The figure contains the sum of the four arrival rates. This means that for each policy there are 2,000 barges observed. The reason why *somewhat satisfied* has a high frequency, is because the most common interval points to that rating. This is shown in the next figure.

We also analyze the difference between the expected and actual sojourn time of barges. We do this because the information correctness is derived from that difference. Figure 4.4 shows frequencies of the differences. It shows the majority of barges have a difference in the interval of *somewhat satisfied* (38 to 180 minutes). This is an indication that the predictive power of both policies lack precision.

We conduct an unpaired t-test to examine the significance of the difference between the information correctness satisfaction in the reserved and unreserved policy cases, on a 95% confidence interval. The statistics used are described in Table 4.5. The two-tailed P value is less than 0.0001. By conventional criteria, this difference is considered to be extremely statistically significant [14].

We also conduct an unpaired t-test on the underlying values that lead to the sat-

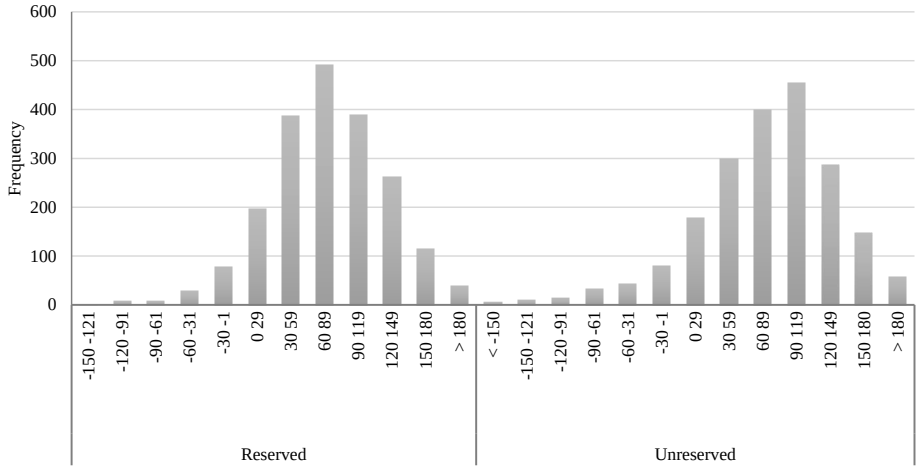


Figure 4.4: Frequency charts of *the difference between expected and actual sojourn time* divided into groups of 30 minutes, combined over the four arrival rates.

Table 4.5: Information correctness satisfaction statistics and the difference between the expected and actual sojourn time, from which the satisfaction level is derived.

		Reserved difference	Reserved satisfaction	Unreserved difference	Unreserved satisfaction
N		2000	2000	2000	2000
Mean		77.78	5.10	79.12	4.99
Median		78.00	5.00	85.50	5.00
Mode		89	5	96	5
Std. Deviation		52.014	.602	61.639	.740
Range		531	5	536	6
Minimum		-133	2	-205	1
Maximum		398	7	331	7
Sum		155566	10194	158236	9975
Percentiles	25	44.25	5.00	43.25	5.00
	50	78.00	5.00	85.50	5.00
	75	113.00	5.00	119.00	5.00

isfaction levels (i.e., the difference between the expected and actual sojourn time). We find the two-tailed P value to be equal to 0.4575. By conventional criteria, this difference is considered to be not statistically significant. This means that reserved policy is preferred in terms of information correctness satisfaction. However, the underlying values that lead to the satisfaction levels are not significantly different.

The reason for this is the aggregating in the satisfaction groups. If you only have a few groups, it is much easier to have something significantly different. Furthermore, in these groups, every member is treated the same, regardless of whether it was on the low end of the interval, or the high end. For example, an interval of group being [38,180], a difference of 40 minutes is treated the same as a difference of 180 minutes, and in addition a difference of 181 minutes would be completely different from 180 minutes, but ”as different” from 40 minutes. The latter ”borderline cases” are prominent when the differences change only by a few minutes, which means they suddenly belong to a completely different group, hence explaining why the information correctness satisfaction is significantly different, but the underlying differences are not.

4.3.2 Waiting time satisfaction

We select the slack setting that results in the most satisfied customers with regard to the waiting time for both the reserved and the unreserved policy for each arrival rate, based on the utility curve described in 3.5.2. The comparison in Appendix A.3 suggests that a slack of 30 minutes results in the most satisfied customers, except for the unreserved policy with arrival rate 12 and 15, for which no slack results in more satisfied customers. Figure 4.5 shows the number of barges per satisfaction level. There are slightly more satisfied barges in case of using the unreserved policy.

The percentage waiting with respect to service time is the underlining value for the satisfaction levels. Figure 4.6 shows the distribution of the percentages. It shows that the distribution is right skewed. It also shows that most barges are in the group below 60%. This explains why most barges are satisfied with the waiting time.

We conduct an unpaired t-test to examine the significance of the difference between the waiting time satisfaction in the reserved and unreserved policy cases, on a 95% confidence interval. Table 4.6 shows the statistics used in the t-test. The

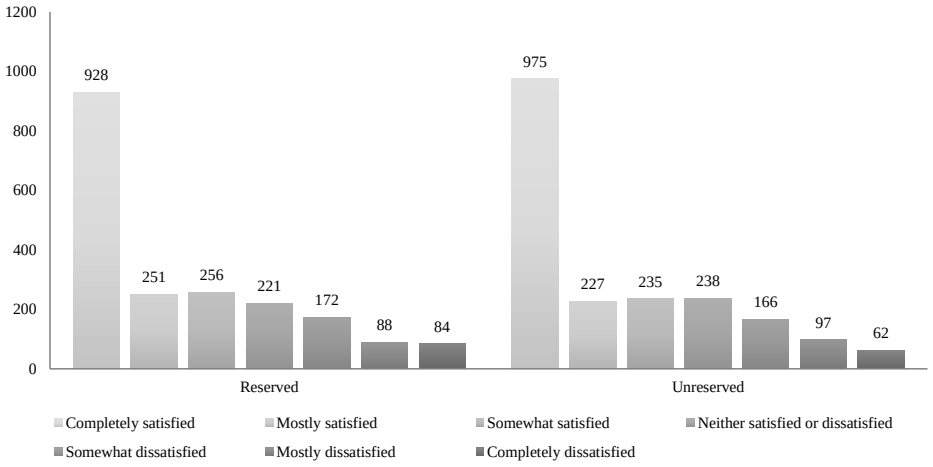


Figure 4.5: The results of the waiting time satisfaction. The figure contains the sum of the four arrival rates. This means that for each policy there are 2,000 barges tested. The reason why *completely satisfied* has a high frequency, is because of the right skewness of the distribution. Every barge with a percentage below 60% is completely satisfied. This is shown in the next figure.

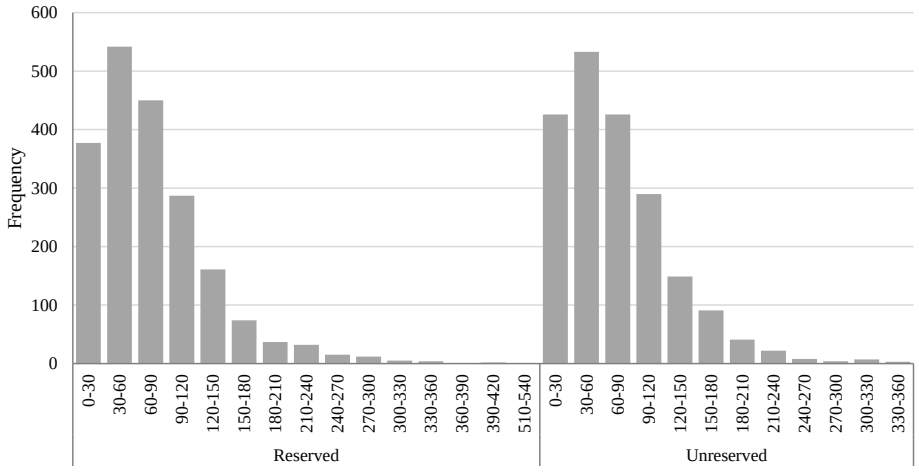


Figure 4.6: Frequency charts of *the percentage waiting time with respect to service time* divided into groups of 20%, combined over the four arrival rates.

two-tailed P value equals 0.2090. By conventional criteria, this difference is considered to be not statistically significant. The P value equals 0.0798 in the t-test on the difference between the reserved and unreserved percentages. By conventional criteria, this difference is considered to be not quite statistically significant. This

Table 4.6: Waiting time satisfaction statistics and the percentage of waiting time with respect to service time, on which the satisfaction levels are based.

		Reserved percentage	Reserved satisfaction	Unreserved percentage	Unreserved satisfaction
N		2000	2000	2000	2000
Mean		76,3940	5,47	73,3038	5,53
Median		64,7529	6,00	62,1582	6,00
Mode		0,00	7	0,00	7
Std. Deviation		57,77664	1,812	53,68400	1,779
Range		512,82	6	359,38	6
Minimum		0,00	1	0,00	1
Maximum		512,82	7	359,38	7
Sum		152787,90	10942	146607,58	11068
Percentiles	25	36,4676	4,00	34,2191	4,00
	50	64,7529	6,00	62,1582	6,00
	75	101,1409	7,00	101,6653	7,00

means that there is no preference for one of the two policies, if assessed on the waiting time satisfaction. However, we do observe a small advantage towards the unreserved policy.

4.4 Distinguished cases

Since the arrival rate and the slack setting have a considerable impact on the satisfaction levels, we select a few distinguished cases to examine separately. On the basis of the means of the satisfaction levels, we select (1) a case where unreserved is better than reserved, (2) another case where reserved is better than unreserved, (3) another case in which one policy is preferred for information correctness and one for waiting time satisfaction, and (4) another case where there is no difference. We explain why that is, based on the arrival rate and slack time. For the selected cases, we compare the means with t-tests and also analyze the underlying values. Table 4.7 shows the means of the satisfaction levels, which we use to select the cases.

Table 4.7: Satisfaction mean per arrival rate per policy.

Slack	Policy	Information provision				Waiting time			
		12	13	14	15	12	13	14	15
0	Reserved	1.44	1.188	1.368	1.398	1.762	1.754	2.414	2.62
	Unreserved	3.392	3.542	3.734	4.282	4.292	5.24	5.836	6.356
	Difference	-1.952	-2.354	-2.366	-2.884	-2.53	-3.486	-3.422	-3.736
30	Reserved	5.062	5.14	5.112	5.074	4.236	5.466	5.91	6.272
	Unreserved	4.826	5.01	5.046	5.068	4.132	5.634	5.854	6.192
	Difference	0.236	0.13	0.066	0.006	0.104	-0.168	0.056	0.08
60	Reserved	4.696	4.588	4.546	4.458	2.408	4.856	5.374	5.908
	Unreserved	4.602	4.552	4.444	4.4	3.72	4.782	5.492	5.804
	Difference	0.094	0.036	0.102	0.058	-1.312	0.074	-0.118	0.104
90	Reserved	4.318	4.166	4.144	4.112	3.73	4.836	5.346	5.768
	Unreserved	4.226	4.13	4.112	4.106	3.734	4.684	5.378	5.938
	Difference	0.092	0.036	0.032	0.006	-0.004	0.152	-0.032	-0.17

Case 1

Observation The cases in which the slack is 0 show much difference in favor of the unreserved policy in the satisfaction of both the information provision and waiting time. We select the cases with arrival rate 15, since these have the largest difference. The t-tests show that the unreserved policy is indeed significantly better, as shown in Table 4.8.

Table 4.8: T-tests for case 1.

		Mean	Std. deviation	P value	Conclusion
Information	Reserved	1.398	0.498	< 0.0001	Unreserved better
correctness	Unreserved	4.282	1.594		
Difference expected	Reserved	-205.200	68.000	< 0.0001	Unreserved better
and actual sojourn time	Unreserved	-26.742	53.300		
Waiting time	Reserved	2.620	1.590	< 0.0001	Unreserved better
satisfaction	Unreserved	6.356	1.238		
Percentage	Reserved	180.207	86.684	< 0.0001	Unreserved better
waiting time	Unreserved	45.980	38.040		

Explanation Clearly, the unreserved policy results in more satisfied customers, as shown in Figure 4.7. The explanation for the information correctness satisfaction lies in the difference between the expected and actual sojourn time. The differences in case of using the reserved policy are much higher than when using the unreserved policy. Also, most of them are negative values, which explains that the completely dissatisfied category has such a high frequency. Furthermore, the expected time is set equal to the best time that follows from the outcome of the TDTSP for each barge, since no slack is added to the waiting profiles. Because we are dealing with uncertainties (i.e., stochastic model), it is difficult to determine the arrival times at the terminals. Therefore, the progress of a barge in the port will not be as planned. As a result, there will be unexpected queues, which are efficiently processed with the unreserved policy, while the reserved policy will make

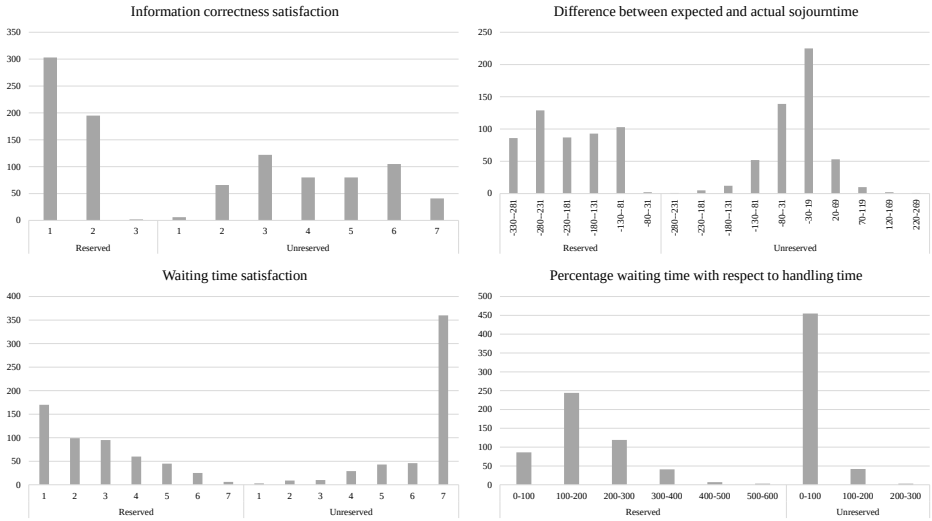


Figure 4.7: Figure for case 1. The top left chart shows frequencies in *information correctness satisfaction levels*, where 1 is completely dissatisfied, and 7 is completely satisfied. The top right chart shows the frequencies of *the difference between expected and actual sojourn time*, which are the underlying values of the information correctness satisfaction levels. The bottom left chart chart shows the *waiting time satisfaction* frequencies. The bottom right shows the frequencies of *the percentages waiting time with respect to handling time*, which are the underlying values for the waiting time satisfaction levels.

terminals wait until a barge with which it has an appointment has arrived (unless it could start handling a barge in the queue without violating an appointment). This makes the time in port many times higher in the case of the reserved policy. This is also reflected in the waiting time satisfaction and the percentage waiting time with respect to the handling time, where the unreserved policy also scores much better.

Case 2

Observation The cases in which the slack is 30 show much smaller differences, than the cases in which the slack is 0. We select the cases with arrival rate 12, since these have the largest difference in favor of the reserved policy. However, the t-tests show that the reserved policy is only significantly better in terms of information correctness satisfaction, as shown in Table 4.9.

Table 4.9: T-tests for case 2.

		Mean	Std. deviation	P value	Conclusion
Information	Reserved	5.062	0.948660061	0.0003	Reserved better
correctness	Unreserved	4.826	1.092754421		
Difference expected	Reserved	50.184	62.62189031	0.5959	Indifferent
and actual sojourn time	Unreserved	52.55	77.62617565		
Waiting time	Reserved	4.236	2.106461098	0.4203	Indifferent
satisfaction	Unreserved	4.132	1.970365621		
Percentage	Reserved	116.3194733	74.72264688	0.8704	Indifferent
waiting time	Unreserved	115.6112687	61.94302353		

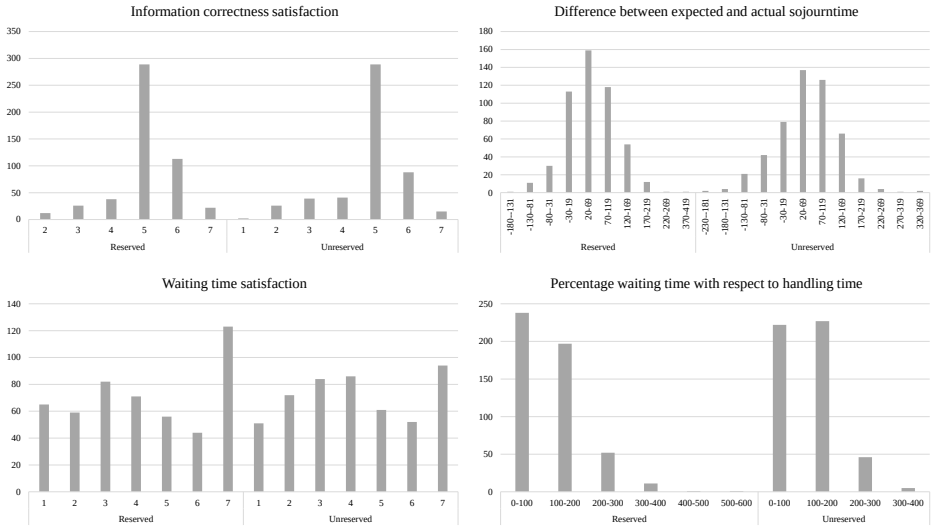


Figure 4.8: Figure for case 2.

Explanation The reason why the information correctness satisfaction is significantly better and the underlying values are not, is because of the aggregating in the satisfaction groups, as explained in Section 4.3.1. When looking at the underlying values in Figure 4.8, we see that the unreserved policy results in somewhat larger differences between the expected and actual sojourn. This makes the derived satisfaction levels just significantly. However, also as argued in Section 4.3.1, the expected sojourn times are not accurate in both cases.

In contrast with case 1, here is slack added to the waiting profiles, which makes the reserved policy as good as the unreserved policy in terms of waiting time satisfaction in this particular case.

Case 3

Observation The cases in which the slack is 60 show again smaller differences. We select the arrival rate 12. This case is interesting, since it has a favor towards the reserved policy with regard to the information provision, and a favor towards the unreserved policy with regard to the waiting time. The t-test show that unreserved policy indeed is better in terms of waiting time satisfaction. However, the unreserved policy only scores significantly better for the underlying values of the information correctness satisfaction, as shown in Table 4.10.

Table 4.10: T-tests for case 3.

		Mean	Std. deviation	P value	Conclusion
Information	Reserved	4.696	1.076057526	0.1096	Indifferent
correctness	Unreserved	4.602	0.751482169		
Difference expected	Reserved	71.64	94.24467368	< 0.0001	Reserved better
and actual sojourn time	Unreserved	141.48	94.0834363		
Waiting time	Reserved	2.408	1.789956393	0.0011	Unreserved better
satisfaction	Unreserved	3.72	1.980263337		
Percentage	Reserved	189.3895369	86.15213566	< 0.0001	Unreserved better
waiting time	Unreserved	131.5271795	71.16521913		

Explanation The reason why the unreserved policy works best in this case is because the queues are more efficiently processed. The reserved policy, however, predicts the sojourns times just somewhat better, but still inaccurate. The reason why the sojourn times are predicted better by the reserved policy has likely something to do with the keeping of appointments.

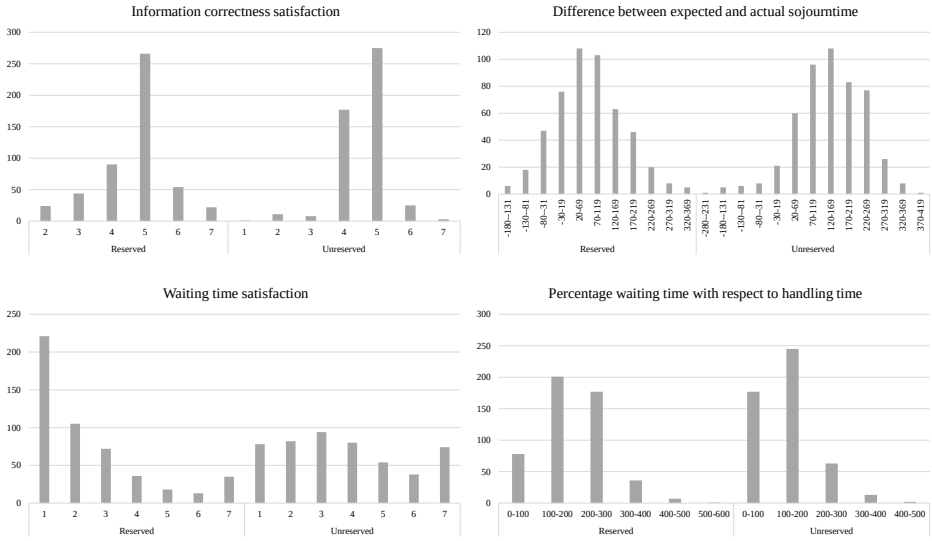


Figure 4.9: Figure for case 3.

Case 4

Observation The cases in which the slack is 90 show the smallest differences. We select arrival rate 14, since the difference is the smallest and we aim for a case for which there is actually no significant difference. As shown in Table 4.11, only the difference between the underlying values of the information correctness satisfaction is significant.

Table 4.11: T-tests for case 4.

		Mean	Std. deviation	P value	Conclusion
Information	Reserved	4.144	0.357098079	0.1438	Indifferent
correctness	Unreserved	4.112	0.334184618		
Difference expected	Reserved	288.994	102.0606423	0.0014	Reserved better
and actual sojourn time	Unreserved	310.444	109.7782738		
Waiting time	Reserved	5.346	1.978411541	0.7925	Indifferent
satisfaction	Unreserved	5.378	1.866033269		
Percentage	Reserved	79.83075919	61.4407829	0.6604	Indifferent
waiting time	Unreserved	78.22145361	54.11285143		

Explanation The reason for the indifferences has likely something to do with the relative high amount of slack in combination with the relatively low arrival rate. With this arrival rate, there are overall less queues present. Due to the high amount of slack, barges are not scheduled in rapid succession. As a result, barges often have to wait the same amount of time in both policies.

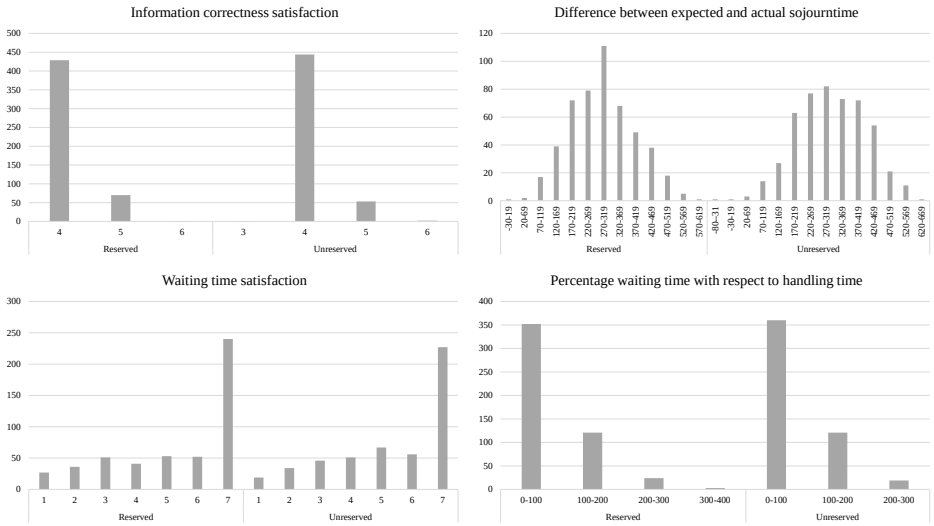


Figure 4.10: Figure for case 4.

4.5 Discussion

The experiments show that there is a connection between the scenario settings and performance in terms of sojourn time and customer satisfaction. We can not identify a single policy and setting that works best in all scenarios. Therefore, we refer to different settings to discuss the findings.

To begin with sojourn time, the experiments show that especially in busy scenarios there is a difference between the policies. An unreserved policy generally works best when there is no slack added to the waiting profiles. The reason for this is that the additional scheduling flexibility gained by adding slack is not necessary, since the queues are not processed on the basis of appointments but on the basis of

FIFO. If one chooses for a reserved policy, then one does well to add a moderate amount of slack to the waiting profiles, in our experiments this was 30 minutes.

When we compare the unreserved policy in combination with no slack with the reserved policy in combination with moderate slack, then the experiments show that, especially for arrival rate 12 and 13, there is an advantage in the direction of unreserved no slack. In the quiet scenarios, arrival rate 13 and 14, the experiments show that there is no clear winner.

When it comes to accuracy of the predictions, then the experiments show that the two policies do not perform well. However, there is a slight advantage toward the reserved policy. This is particularly due to keeping appointments. However, we believe that the accuracy should be improved before there can be talked about predictive value. For clarity, this applies to both policies.

Regarding customer satisfaction, we found the same pattern as in the sojourn time. Reserved with a slack of only about 30 minutes leads to slightly better customer satisfaction with regard to the provision of information. The unreserved policy leads to a slightly better customer satisfaction with respect to the waiting time. This follows logically from the lower sojourn times.

Table 4.12: Where = stands for no significant difference, res and unr stand for a significant difference, where res means that it is in favor of the reserved policy and unr is in favor of the unreserved policy (in favor meaning higher satisfaction).

	information correctness				waiting time			
	0	30	60	90	0	30	60	90
12	unr	res	=	=	unr	=	unr	=
13	unr	=	=	=	unr	=	=	=
14	unr	=	=	=	unr	=	=	=
15	unr	=	=	=	unr	=	=	=

If the same amount of slack is added to the reserved and unreserved policy, then the experiments show that the difference in customer satisfaction is as shown in Table 4.12. Note that the preference for *unr* at arrival rate 12 and slack 60 looks

like a deviating preference due to the fact the res does performs well at slack 30.

The table makes clear that the unreserved policy works well without slack, and can form a good alternative to the reserved policy that was used in the base model. A reserved policy only leads to significantly improved customer satisfaction with regard to the provision of information in the busiest scenario. And yet it has no predictive value in our opinion, since the differences between the expected and actual time are too high.

Summarizing, the experiments have shown that the unreserved policy is simple but effective. It is not perfect, and there is room for improvement, in particular in the form of better predictions. However, this also applies to the alternative.

Chapter 5

Conclusion

5.1 Answer to the research questions

The terminal queue processing policy has a significant impact on the sojourn time in the port. The simulation experiments show that using an unreserved policy with no slack results in the least actual sojourn time in the port on average. This is the most essential finding of the simulation experiments. The explanation for this is that terminals are less idle, giving the port an overall higher throughput. In addition, the experiments show that when one would use a reserved policy, it is best to add constant slack to the waiting profiles.

When it comes to the predictive power, the experiments show an advantage towards the reserved policy. With regard to the satisfaction with the information correctness, we found also that using the reserved policy results in overall more satisfied customers (i.e., barge operators). However, a large majority is only somewhat satisfied with the information correctness in both the policies. In other words, the predictive power does lack precision for both policies. In the reserved policy this is due to the slack added to the waiting profiles, while in the unreserved policy, it is due to not strictly keeping to appointments.

With regard to the satisfaction with the waiting time, we found that the unreserved policy performed slightly better. However, the difference is not very signifi-

cant. This is surprising, since the unreserved policy results in significant less actual sojourn time in the port. The reason why the difference in waiting time satisfaction is not significant, may possibly be attributed to the way in which the measurement is implemented.

In addition to the answers to the research questions, the experiments show that the differences between the reserved policy and unreserved policy are less significant in more quiet scenarios. The explanation for this is that the queues and waiting times are smaller in the more quiet scenarios. Because the policies primarily affect the queues and waiting times, those differences are less reflected in scenarios where these are less common.

5.2 Summary

We develop a simulation program to evaluate two policies for processing queues by container terminals. In the reserved policy, queues are processed on the basis of their appointment, while in the unreserved policy queues are processed on the basis of the first-in first-out method. What makes the unreserved policy different from a regular first-in first-out method system, is that terminals and barges still make their appointments. However, the appointments are less strict than with the reserved policy. In this way, the appointments influence the route of the barges.

We run multiple simulation experiments in which we evaluate the policies in busy scenarios and quiet scenarios. The results indicate that using an unreserved policy results in significantly less sojourn time in the port in the busy scenarios. In the most busy scenario the sojourn time in the port dropped by 24% (about 140 minutes) on average per barge. In the second most busy scenario the sojourn time in the port dropped by 9% (about 46 minutes) on average per barge. In the quiet scenarios the difference is less significant. In the most quiet scenario the sojourn time in the port dropped by 1.3% (about 6 minutes) on average per barge. In the second quiet scenario the sojourn time in the port even increases with 1.7% (about

7 minutes) on average per barge.

The reason why it works better in the busy scenarios is that there are usually queues at the terminals. Therefore, the terminals are almost always handling barges. In other words, terminals are most of the time not idle. This results in a higher throughput of the whole system.

A drawback of the unreserved policy is that the predicted (or expected) sojourn time is less accurate than when using the reserved policy. The reason for this is that terminals do not consider the planned times to determine whether or not a barge can start handling. Therefore, the handling often starts at another time than planned. This is also reflected in the satisfaction of barges, since there are more barges satisfied with the information correctness in case of using a reserved policy.

With regards to the slack method, the experiments indicate that no slack should be added to the waiting profiles when using the unreserved policy. When using the reserved policy, a slack of 90 minutes should be used in the most busy scenarios, while a slack of 30 minutes should be used in the more quiet scenarios. The reason why more busy scenarios perform better with more slack, is because it gives the terminals more planning flexibility.

5.3 Future research

The simulation experiments show that using an unreserved policy results in significant lower sojourn times in the port, especially in busy ports. However, the expected times lack precision. Future research can focus on increasing the predictability of the sojourn time. For example, the expected time can be modified with a fixed number or percentage. Also, the deviation in the actual sailing and handling time can be predicted and then used in the TSP.

We also suggest further research on the measurement of the waiting time satisfaction. The experiments show that a majority of barges are completely satisfied with the waiting time. In our measurement, we made assumptions on the values

that are used to assign a certain waiting time to certain a rating. In our opinion, these values are still correct. However, future research may focus on creating more supported values, for example by doing a survey research. The same applies to the measurement of the information correctness satisfaction, where we found a large majority in the *somewhat satisfied* category. However, this rating may also be true, since it is rare that the expected sojourn time is close the actual sojourn time. In order to improve the information correctness satisfaction, the precision of the expected sojourn time should improve.

Other future research can examine the practical implications of using an unreserved policy. Since appointments are not fixed, and terminals process queues on basis of first-in first-out, then barges and terminals could dismiss the system. Perhaps an incentive or reward system can be created.

Finally, a more intelligent way of determining the amount of slack added to the waiting profiles can be investigated in case of using a reserved policy. Although the reserved policy results in worse actual sojourn time than the unreserved policy, we still believe that in the end it is important to have appointments and also to comply with appointments. In the ideal situation, one would have the results of the unreserved policy, but with kept appointments. This requires a more accurate prediction of the time of events in the port.

Acknowledgments

I would like to thank my supervisor Yingqian and co-reader Charlie. Thank you very much for your time and for the meetings. Without your knowledge and skills this thesis would never have been completed in this form. I liked working with you and I hope to hear from you in the future.

Bibliography

[1] Frédéric Bielen and Nathalie Demoulin. Waiting time influence on the satisfaction-loyalty relationship in services. *Managing Service Quality*, 17(2):174 – 193, 2007.

[2] Kris Braekers, An Caris, and Gerrit K. Janssens. Optimal shipping routes and vessel size for intermodal barge transport with empty container repositioning. *Computers in Industry*, (0):–, 2012.

[3] Ann Melissa Campbell and Martin Savelsbergh. Incentive schemes for attended home delivery services. *Transportation Science*, 40(3):327–341, 2006.

[4] An Caris, Cathy Macharis, and Gerrit K. Janssens. Planning problems in intermodal freight transport: Accomplishments and prospects. *Transportation Planning and Technology*, 31(3):277–302, 2008.

[5] Marielle Christiansen, Kjetil Fagerholt, and David Ronen. Ship routing and scheduling: Status and perspectives. *Transportation Science*, 38(1):1–18, 2004.

[6] Apache Commons. Random data generator. `http://commons.apache.org/proper/commons-math/apidocs/org/apache/commons/math3/random/RandomDataGenerator.html`, 2014. [Online; accessed 23-April-2014].

[7] Jean-Francois Cordeau, Gilbert Laporte, Pasquale Legato, and Luigi Moccia. Models and tabu search heuristics for the berth-allocation problem. *Transportation Science*, 39(4):526–538, 2005.

[8] Fred D. Davis, Richard P. Bagozzi, and Paul R. Warshaw. User acceptance of computer technology: A comparison of two theoretical models. *Management Science*, 35(8):pp. 982–1003, 1989.

[9] Mark Davis and Thomas Vollmann. A framework for relating waiting time and customer satisfaction in a service operation. *Journal of Services Marketing*, 4(1):61 – 69, 1990.

[10] Keith Decker and Jinjiang Li. Coordinating mutually exclusive resources using gpgp. *Autonomous Agents and Multi-Agent Systems*, 3(2):133–157, June 2000.

[11] Albert Douma, Marco Schutten, and Peter Schuur. Waiting profiles: An efficient protocol for enabling distributed planning of container barge rotations along terminals in the port of rotterdam. *Transportation Research, Part C: Emerging technologies*, 17(2):133–148, 2009. Selected papers from the Sixth Triennial Symposium on Transportation Analysis (TRISTAN VI).

[12] Albert Douma, Marco Schutten, and Peter Schuur. Aligning barge and terminal operations using service-time profiles. *Flexible Services and Manufacturing Journal*, 23(4):385–421, 2011. Open access article.

[13] Albert Douma and Peter Schuur. Design and evaluation of a simulation game to introduce a multi-agent system for barge handling in a seaport. *Decision Support Systems*, 53(3):465 – 472, 2012.

[14] Graphpad. Quickcalcs t test calculator. `http://www.graphpad.com/quickcalcs/ttest1/?Format=SD`, 2014. [Online; accessed 2-June-2014].

[15] Cay Horstmann. *Java for Everyone*. Wiley, 2e edition, 2013. [Section 13.7 Backtracking].

[16] Tal Len-Zvi. The efficacy of business simulation games in creating decision support systems: An experimental investigation. *Decision Support Systems*, 49(1):61 – 69, 2010.

[17] David Maister. *The psychology of waiting lines*. Lexington Books/DC Heath, London, 1985.

[18] Chryssi Malandraki and Mark S. Daskin. Time dependent vehicle routing problems: Formulations, properties and heuristic algorithms. *Transportation Science*, 26(3):185–200, 1992.

[19] Chryssi Malandraki and Robert B. Dial. A restricted dynamic programming heuristic algorithm for the time dependent traveling salesman problem. *European Journal of Operational Research*, 90(1):45–55, 1996.

[20] Portbase. Home of logistics intelligence. `http://data.worldbank.org/indicator/IS.SHP.GOOD.TU/`, 2014. [Online; accessed 10-Juni-2014].

[21] Repast Simphony. Interface ischedule. `http://repast.sourceforge.net/docs/api/repast_simphony/repast/simphony/engine/schedule/ISchedule.html`, 2014. [Online; accessed 9-June-2014].

[22] Robert Stahlbock and Stefan Voß. Operations research at container terminals: a literature update. *OR Spectrum*, 30:1–52, 2008. 10.1007/s00291-007-0100-9.

[23] Tom Thurston and Huosheng Hu. Distributed agent architecture for port automation. In *26th International Computer Software and Applications Conference, IEEE Computer Society*, pages 819–0. IEEE Computer Society, 2002.

[24] Worldbank. Container port traffic. http://data.worldbank.org/indicator/IS.SHP.GOOD.TU/, 2013. [Online; accessed 16-September-2013].

Appendix A

Data

A.1 Significance test

This appendix describes an example of a significance test as conducted in the results chapter. Table A.1 contains the table with the data used for the tests. The table contains the the standard deviations, minimum, maximum of the expected/actual times, the sample size and unfinished barges (i.e., barges that are in the port at the end of the simulation). We use this information to conduct the tests.

As an example, we show how we test which slack settings results in the lowest sojourn time, and which slack setting is the best predictor for the sojourn time for arrival rate 12. The test is repeated for the other arrival rates, terminal logic and slack settings. The results of all tests are summarized in the results section of the thesis.

We compare the averages of the two settings which presumably are the best, i.e., with the lowest sojourn. This is a constant slack of 0 and 90. The null and alternative hypotheses are as follows: $H_0 : \mu_0 = \mu_{90}$; $H_a : \mu_0 \neq \mu_{90}$. A two-sample t-test shows that there is a significant difference in actual sojourn time when using a constant slack of 0 minutes ($\bar{x}_0 = 941$, $\sigma_0 = 274$, $n_0 = 4,507$) and a constant slack of 90 minutes ($\bar{x}_{90} = 721$, $\sigma_{90} = 316$, $n_{90} = 4,580$); t-difference= -8.126; df-t= 17955.1; p-value= 0. We use a significance level of 5%. We reject H_0, since

the p-value is smaller than the significance level. Therefore, we can conclude that when the arrival rate is 12 minutes per barge the best constant slack is 90 minutes.

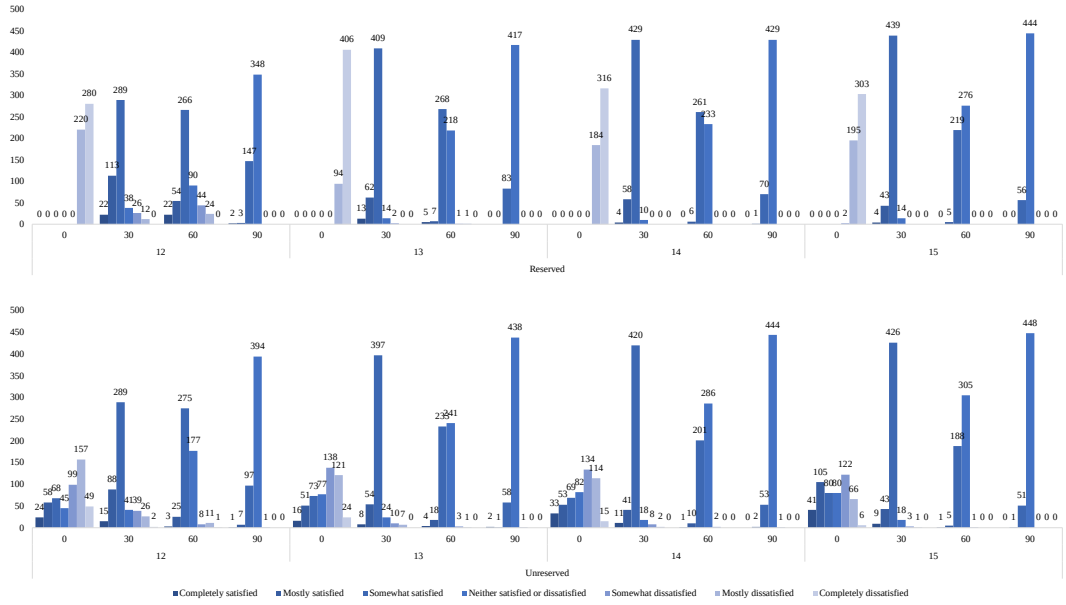
Next, we test which slack constant gives the best prediction for the sojourn time for arrival rate 12. We suspect that this is also slack constant 90. We test it against the slack constant 90, since that appears to be the second best. First, we need the difference in means and standard deviations. We compute this as follows: $\bar{x} = \bar{x}_{actual} - \bar{x}_{expected}$; $\sigma = \sqrt{\sigma^2_{actual} + \sigma^2_{expected}}$.

The null and alternative hypotheses are as follows: $H_0 : \mu_0 = \mu_{90}$; $H_a : \mu_0 \neq \mu_{90}$. A two-sample t-test shows that there is a significant difference in predictability of the slack constant when using a constant slack of 0 minutes ($\bar{x}_0 = 302$, $\sigma_0 = 330$, $n_0 = 4,507$) and a constant slack of 90 minutes ($\bar{x}_{90} = -40$, $\sigma_{90} = 349$, $n_{4,580} = 9,475$); t-difference= -11.344; df-t= 18921.6; p-value= 0. We use a significance level of 5%. We reject H_0, since the p-value is smaller than the significance level. Therefore, we can conclude that when the arrival rate is 12 minutes per barge the best predictor for the actual sojourn time is to use constant slack is 90 minutes.

A.2 Information provision satisfaction

We choose the slack settings which result in the most satisfied customers with regard to the information provision. These are used to compare the reserved and unreserved policy in Chapter 4. On the y-axis we set the number of observation. On the x-axis we set the arrival rate (12 to 15) and the amount of slack (0, 30, 60, 90).

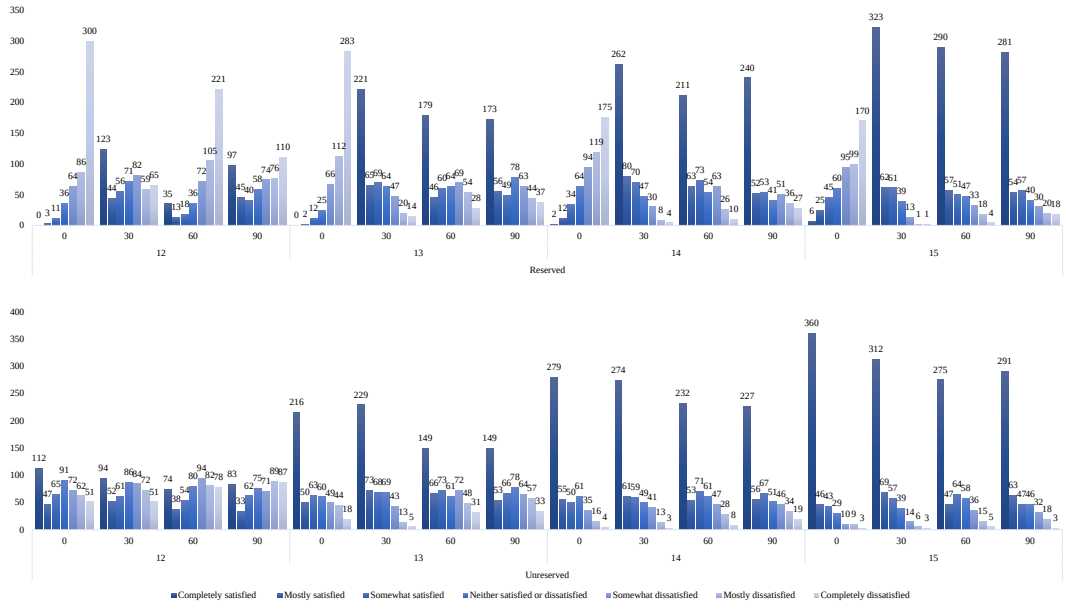
We found that using a slack of 30 minutes results in the overall most satisfied customers in all cases. Although a slack of 0 minutes in case of using an unreserved policy results is more completely satisfied customers, it also results in a higher degree of customer dissatisfaction. Therefore, we argue that using 30 minutes is preferred.



A.3 Waiting time satisfaction

We choose the slack settings which result in the most satisfied customers with regard to the waiting time. These are used to compare the reserved and unreserved policy in Chapter 4. On the y-axis we set the number of observation. On the x-axis we set the arrival rate (12 to 15) and the amount of slack (0, 30, 60, 90).

We found that using a slack of 30 minutes results in the overall most satisfied customers in all cases, except for the cases with the unreserved policy and arrival rate 12 and 15, where a slack of 0 minutes works slightly better.



policy	arrival rate	parameter	expected sojourn time				actual sojourn time				waiting time				handling time				sailing time				actual - expected $\bar{x}$	actual- expected σ	sample size n	unfinished barges
			$\bar{x}$	σ	*min*	*max*	$\bar{x}$	σ	*min*	*max*	$\bar{x}$	σ	*min*	*max*	$\bar{x}$	σ	*min*	*max*	$\bar{x}$	σ	*min*	*max*				
Reserved terminal logic and constant slack	12	0	638	184	182	1,428	941	274	245	1,785	658	274	78	1,499	151	39	30	315	131	28	56	225	302	330	4507	464
		30	676	241	218	1,891	946	422	147	2,693	664	416	2	2,391	151	39	40	304	130	27	52	224	269	486	4404	556
		60	637	109	191	988	1,011	488	110	2,630	730	480	0	2,244	151	39	38	311	130	26	54	205	375	500	4371	603
		90	761	147	202	1,255	721	316	110	2,182	439	302	0	1,845	151	40	34	314	131	26	55	224	-40	349	4580	381
	13	0	455	123	124	1,002	746	198	227	1,370	462	196	45	1,092	152	39	31	312	132	28	56	222	291	233	4221	338
		30	515	113	172	1,169	516	177	121	1,521	232	161	0	1,196	151	39	37	306	132	27	54	231	1	210	4303	244
		60	608	112	188	978	542	185	122	1,383	260	165	0	955	151	39	31	300	130	26	52	221	-66	216	4292	259
		90	731	146	206	1,240	513	180	113	1,481	231	158	0	1,089	151	39	35	317	131	27	51	217	-218	232	4359	195
	14	0	376	84	116	755	654	161	226	1,080	370	157	20	846	151	39	32	313	132	27	53	223	278	182	3912	277
		30	477	89	172	938	399	103	113	951	116	74	0	707	151	39	33	310	131	26	55	229	-78	136	4043	136
		60	587	113	172	949	426	123	117	911	144	94	0	518	152	39	33	311	131	26	52	221	-161	167	4040	145
		90	704	147	202	1,239	433	134	109	1,036	151	106	0	647	152	39	36	302	131	27	53	220	-271	199	4033	152
	15	0	341	69	112	647	578	141	131	960	296	135	0	700	151	39	35	308	131	27	47	213	237	157	3662	209
		30	461	87	154	996	374	94	117	859	91	63	0	611	151	39	36	311	132	27	55	234	-87	128	3759	110
		60	573	113	179	946	394	111	109	899	112	81	0	482	151	39	33	307	130	26	50	217	-179	159	3758	120
		90	689	147	214	1,246	400	118	112	998	118	89	0	632	151	39	35	316	130	27	52	230	-289	188	3764	115
Reserved terminal logic and factor slack	12	10	773	298	194	4,281	2,223	1,069	406	6,906	1,934	1,071	123	6,753	151	40	31	311	139	30	53	251	1450	1110	3587	1416
		20	603	192	155	2,667	1,489	797	245	4,750	1,201	795	78	4,612	152	39	37	315	137	30	54	259	886	820	3924	1048
		30	604	190	147	1,870	1,106	543	245	3,685	820	539	78	3,495	151	39	30	315	134	29	42	244	502	575	4216	755
		40	626	188	182	2,097	987	361	245	2,927	702	359	78	2,693	151	39	30	315	133	28	56	242	361	408	4392	579
	13	10	589	195	140	1,882	1,577	781	266	4,186	1,288	780	49	3,885	151	39	38	314	138	30	55	237	988	805	3629	950
		20	466	131	124	1,739	956	468	227	3,297	669	465	45	3,154	151	39	33	312	135	29	54	236	490	486	3957	602
		30	456	128	117	1,256	795	258	227	1,971	510	255	45	1,621	152	39	33	312	133	29	56	240	339	289	4144	415
		40	455	123	124	1,002	746	198	227	1,370	462	196	45	1,092	152	39	31	312	132	28	56	222	291	233	4221	338
	14	10	460	137	140	1,434	1,088	510	200	3,025	802	509	10	2,792	151	39	36	302	136	29	57	228	629	528	3621	573
		20	387	96	116	1,152	711	238	226	1,720	425	234	20	1,506	151	39	36	313	134	28	50	220	324	257	3844	345
		30	376	84	116	755	654	161	226	1,080	370	157	20	846	151	39	32	313	132	27	53	223	278	182	3912	277
		40	376	84	116	755	654	161	226	1,080	370	157	20	846	151	39	32	313	132	27	53	223	278	182	3912	277
	15	10	388	100	124	1,434	771	346	130	2,183	486	343	0	2,032	151	39	34	306	135	28	51	229	383	360	3540	332
		20	344	75	124	868	588	151	131	1,201	306	145	0	935	151	39	37	308	132	27	47	224	245	168	3655	216
		30	341	69	112	647	578	141	131	960	296	135	0	700	151	39	35	308	131	27	47	213	237	157	3662	209
		40	341	69	112	647	578	141	131	960	296	135	0	700	151	39	35	308	131	27	47	213	237	157	3662	209
Unreserved terminal logic and constant slack	12	0	464	122	112	981	582	211	115	2,079	300	191	0	1,668	151	39	28	317	132	28	49	231	118	244	4707	243
		30	545	113	172	1,179	607	222	135	1,928	323	199	0	1,527	151	39	33	306	133	28	47	218	62	248	4709	247
		60	637	108	190	980	642	246	147	2,022	361	227	0	1,739	151	40	32	305	130	26	53	205	5	269	4694	264
		90	779	143	202	1,250	625	234	126	2,025	344	212	0	1,678	151	40	33	317	130	26	50	207	-154	274	4702	263
	13	0	395	94	112	769	470	150	129	1,287	186	128	0	933	151	39	36	312	132	28	51	233	74	177	4374	175
		30	502	94	142	1,116	467	149	116	1,557	183	124	0	1,184	151	39	36	304	133	27	51	217	-35	176	4366	184
		60	613	111	188	989	482	150	122	1,322	201	126	0	973	151	39	35	315	130	26	56	220	-131	187	4374	187
		90	744	147	202	1,246	487	158	110	1,389	206	134	0	1,046	151	39	37	308	130	26	52	246	-257	215	4358	185
	14	0	355	74	119	697	406	112	112	1,089	123	87	0	746	152	39	30	313	132	27	54	233	51	134	4047	136
		30	480	86	142	898	414	113	114	1,092	130	86	0	749	152	39	36	315	132	27	57	216	-66	142	4040	140
		60	593	112	184	976	420	116	131	1,011	138	88	0	607	152	39	36	316	130	26	53	208	-173	161	4032	153
		90	720	149	202	1,202	428	121	108	961	146	93	0	632	151	39	30	301	130	27	55	222	-292	192	4030	153
	15	0	332	65	112	590	368	94	118	803	86	65	0	459	151	39	33	306	131	27	53	212	36	114	3756	108
		30	462	84	154	756	375	91	121	928	92	59	0	563	151	39	38	308	131	26	55	222	-88	124	3767	107
		60	580	114	172	983	392	103	116	915	110	74	0	588	151	39	31	323	130	26	54	227	-188	154	3759	116
		90	701	149	202	1,239	397	108	111	920	116	77	0	514	151	39	34	314	130	27	51	235	-303	184	3753	121
Unreserved terminal logic and factor slack	12	10	478	135	128	1,688	616	234	135	2,005	327	211	0	1,626	151	40	37	308	138	30	53	265	138	270	4691	266
		20	463	121	124	1,279	592	218	115	1,908	308	194	0	1,568	151	40	34	317	134	29	49	247	129	249	4695	255
		30	463	120	112	1,125	580	210	115	1,873	297	189	0	1,421	151	39	34	317	133	29	49	231	118	242	4701	249
		40	464	122	112	981	582	211	115	2,079	300	191	0	1,668	151	39	28	317	132	28	49	231	118	244	4707	243
	13	10	399	95	114	1,021	475	149	133	1,261	188	125	0	934	151	39	35	307	136	28	54	239	75	177	4370	185
		20	394	93	112	1,146	469	156	129	1,605	185	132	0	1,209	151	39	34	312	133	28	50	229	75	181	4374	175
		30	395	94	112	769	470	150	129	1,287	186	128	0	933	151	39	36	312	132	28	51	233	74	177	4374	175
		40	395	94	112	769	470	150	129	1,287	186	128	0	933	151	39	36	312	132	28	51	233	74	177	4374	175
	14	10	363	82	124	941	412	118	105	1,202	127	91	0	806	152	39	39	313	134	28	51	217	49	143	4035	148
		20	355	75	119	899	408	113	112	1,099	125	88	0	767	152	39	32	313	132	27	54	220	53	135	4046	137
		30	355	74	119	697	406	112	112	1,089	123	87	0	746	152	39	30	313	132	27	54	233	51	134	4047	136
		40	355	74	119	697	406	112	112	1,089	123	87	0	746	152	39	30	313	132	27	54	233	51	134	4047	136
	15	10	335	68	112	778	373	95	112	876	89	67	0	517	151	39	33	310	132	27	51	226	38	117	3757	110
		20	332	65	112	590	368	94	118	803	86	65	0	459	151	39	33	306	131	27	53	212	36	114	3756	108
		30	332	65	112	590	368	94	118	803	86	65	0	459	151	39	33	306	131	27	53	212	36	114	3756	108
		40	332	65	112	590	368	94	118	803	86	65	0	459	151	39	33	306	131	27	53	212	36	114	3756	108

Table A.1: Results per policy per arrival rate. Each row is an average of four simulations, consisting of the four random seeds.

Appendix B

Source code

Building the simulation program is a large part of the project. This appendix contains the source code of the program. The source code is also available online via `http://goo.gl/OIIrRZ`. An archive file which can be imported into Eclipse/Repast Symphony is also available on that address. The project name is *jbarges*, wherein the *j* stands for Java, and *barges* stands for the entity that moves through the system. The sequence of the classes is as follows: the Port class, the Barge class, the Terminal class, the WaitingProfile class, the Statistics class, the TDTSP class, the Stage class, and the PartialSolution class. The purpose of providing the source code is to give better understanding on how the program is build. In addition, to make it possible for future studies to use the whole program, or parts of it.

```java
package jbarges;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.Random;

import org.apache.commons.math3.random.RandomDataGenerator;
import org.apache.commons.math3.util.Precision;
import org.apache.poi.ss.usermodel.Cell;
import org.apache.poi.ss.usermodel.Row;
import org.apache.poi.ss.usermodel.Sheet;
import org.apache.poi.ss.usermodel.Workbook;
import org.apache.poi.xssf.usermodel.XSSFWorkbook;

import com.google.common.collect.HashBasedTable;
import com.google.common.collect.Table;

import repast.simphony.context.Context;
import repast.simphony.dataLoader.ContextBuilder;
import repast.simphony.engine.environment.RunEnvironment;
import repast.simphony.engine.schedule.ISchedule;
import repast.simphony.engine.schedule.ScheduleParameters;
import repast.simphony.parameter.Parameters;
import repast.simphony.util.ContextUtils;


/**
 * Builds the context for the simulation.
 */
public class Port implements ContextBuilder<Object> {

    /**
     * The Repast ISchedule manages the execution of events according to the simulation clock.
     */
    public static ISchedule schedule;

    /**
     * The sailing times in a (Guava) Table object where the keys are the Terminals and the value is the sailing time.
     */
    public static Table<Terminal,Terminal,Integer> sailingtimesTable;

    /**
     * The Terminal objects are stored in this ArrayList.
     */
    public static ArrayList<Terminal> terminals;

    /**
     * The String parameters.Model stands for stochastic or deterministic sailing and handling times.
     */
    public static String sheetName, model, terminalLogic, slackMethod, eventsToExcel, bargesDetailsToExcel;

    /**
     * The RandomDataGenerators.
     */
    public static RandomDataGenerator arrivalRNG, numTerminalRNG, handlingTimeRNG, timeRNG;

    /**
     * The random data seed parameter.
     */
    public static long seed;

    /**
     * The Integer parameters.
     */
    public static int timeSigma, slack, slackDenominator, numNodes;

    /**
     * The Double parameters.
     */
    public static double endTime, warmup, arrivalRate, numTerminalMean, numTerminalStd, handlingTimeMean, handlingTimeStd;

    /**
     * The Statistics object that manages the statistics.
     */
    public static Statistics stats;

    /**
     * Keeps track of the scenarios. Especially useful when running simulations in batches. We set this to 0,
     * and plus 1 in the build method. When a new context is build, the scenario number will be updated.
     */
    public static int scenarioCount=0;

    /**
     * Used to randomly select terminals to visit. This is used in the arriveAtPort() method.
     */
    ArrayList<Integer> terminalList;

```

```
 93
 94      /**
 95       * This method build the Context.
 96       * The Context is the core concept and object in Repast Simphony.
 97       * It provides a data structure to organize agents.
 98       */
 99      @Override
100      public Context<Object> build(Context<Object> context) {
101
102          schedule = RunEnvironment.getInstance().getCurrentSchedule();
103
104          //read parameters, setup random data generators, construct terminal agents, and read sailing times table.
105          this.readParameters();
106          this.setupRandomgenerators();
107          this.createTerminals();
108          this.sailingtimesTable();
109
110          //add terminal agents
111          context.addAll(terminals);
112
113          //construct and add statistics object
114          stats = new Statistics();
115          context.add(stats);
116
117          //plus 1 the scenario number
118          scenarioCount++;
119
120          //schedule the first event in the ISchedule schedule
121          schedule.schedule(ScheduleParameters.createOneTime(0), this, "initialize"); // schedule first event
122
123          return context;
124      }
125
126
127      /**
128       * The first event of the simulation. Schedules the first arrival.
129       */
130      public void initialize() {
131          int arrivalTime = (int)schedule.getTickCount()+(int) Precision.round(arrivalRNG.nextExponential(Port.arrivalRate),0);
132          schedule.schedule(ScheduleParameters.createOneTime(arrivalTime, 1), this, "arriveAtPort");
133          schedule.schedule(ScheduleParameters.createOneTime(warmup, ScheduleParameters.LAST_PRIORITY), stats, "warmupReset");
134          schedule.schedule(ScheduleParameters.createOneTime(endTime, ScheduleParameters.LAST_PRIORITY), this, "end");
135      }
136
137
138      /**
139       * Reads in all the parameters and initializes the corresponding objects.
140       * Note that the parameters that require configuration from that GUI was changed,
141       * therefore there are some fixed parameters below.
142       */
143      public void readParameters(){
144          Parameters params = RunEnvironment.getInstance().getParameters();
145          sheetName = "14 terminals";
146          seed = params.getInteger("randomSeed");
147          warmup = 2880;
148          endTime = 17280;
149          arrivalRate = params.getDouble("arrivalRate");
150          numTerminalMean = 5;
151          numTerminalStd = 1;
152          handlingTimeMean = 30;
153          handlingTimeStd = 10;
154          eventsToExcel = params.getString("eventsToExcel");
155          bargesDetailsToExcel = params.getString("bargesDetailsToExcel");
156          model = params.getString("model");
157          timeSigma = 3;
158          terminalLogic = params.getString("terminalLogic");
159          slack = params.getInteger("slack");
160          slackMethod = params.getString("slackMethod");
161          slackDenominator = params.getInteger("slackDenominator");
162      }
163
164      /**
165       * Setup the random data generators.
166       */
167      public void setupRandomgenerators(){
168          arrivalRNG = new RandomDataGenerator();
169          numTerminalRNG = new RandomDataGenerator();
170          handlingTimeRNG = new RandomDataGenerator();
171          timeRNG = new RandomDataGenerator();
172          arrivalRNG.reSeed(seed);
173          numTerminalRNG.reSeed(seed);
174          handlingTimeRNG.reSeed(seed);
175          timeRNG.reSeed(seed);
176      }
177
178
179
180      /**
181       * Create terminal object/agents.
182       * The number of rows in sailing times is equal to the number of terminals.
183       * (Note: the port entrance is seen terminal object)
184       */
```

```
185     public void createTerminals(){
186         numNodes=15;
187         terminals = new ArrayList<Terminal>();
188         for(int i=0; i<numNodes; i++){
189             terminals.add(new Terminal("t"+i));
190         }
191 
192         terminalList = new ArrayList<Integer>();
193         for(int i = 1; i<terminals.size(); i++){ //exclude i = 0 (= port entrance)
194             terminalList.add(i);
195         }
196     }
197 
198 
199     /**
200      * This method creates all parameters to construct a barge. After that the barge is constructed.
201      * It also schedules the arrival of the next barge.
202      */
203     public void arriveAtPort(){
204 
205         // update statistic
206         Port.stats.bargesEnteredPort++;
207 
208         // declare the input for the barge constructor
209         ArrayList<Terminal> terminalsToVisit = new ArrayList<Terminal>();
210         ArrayList<Integer> handlingTimes = new ArrayList<Integer>();
211 
212         // add depot node with handling time 0
213         terminalsToVisit.add(terminals.get(0));
214         handlingTimes.add(0);
215 
216         // number of terminals to visit, normal distribution
217         int numberToVisit = (int) Precision.round(numTerminalRNG.nextGaussian(Port.numTerminalMean, Port.numTerminalStd),0);
218 
219         // max 8 terminals to visit
220         if(numberToVisit>8){
221             numberToVisit=8;
222         }
223 
224         //this could happen in ports with less than 8 terminals.
225         if(numberToVisit>Port.terminals.size()-1){
226             numberToVisit=Port.terminals.size()-1;
227         }
228 
229         // select terminals randomly, add handling times and add to lists
230         Collections.shuffle(terminalList, new Random(1));
231         for(int i = 0; i<numberToVisit;i++){
232             int terminalNumber = terminalList.get(i);
233             Terminal terminal = terminals.get(terminalNumber);
234             Integer handlingTime = (int) Precision.round(numTerminalRNG.nextGaussian(Port.handlingTimeMean, Port.handlingTimeStd),
235             // the minimum handling time is 10
236             if(handlingTime<10){
237                 handlingTime=10;
238             }
239             terminalsToVisit.add(terminal);
240             handlingTimes.add(handlingTime);
241         }
242 
243         // arrival time of the barge in the port as an integer
244         int arrivalTime = (int) Math.round(schedule.getTickCount());
245 
246         // create the new barge agent
247         Barge barge = new Barge(stats.bargeCount, arrivalTime, terminalsToVisit, handlingTimes);
248 
249         // add to context: first get the context by using one of the terminals, then use it to add the barge
250         @SuppressWarnings("unchecked")
251         Context<Object> context = ContextUtils.getContext(terminals.get(0));
252         context.add(barge);
253 
254         if(Port.eventsToExcel.equals("Yes")){
255             Port.stats.addEvent(barge.arrivalTime, barge.bargeNumber, "Arrived at Port");
256         }
257 
258         //update barge count so the next barge will get a new bargeNumber.
259         stats.bargeCount++;
260 
261         // if the next barge arrives before the end time, then schedule the next arrival
262         int nextArrivalTime = (int)schedule.getTickCount()+(int) Precision.round(arrivalRNG.nextExponential(Port.arrivalRate),0);
263         if (nextArrivalTime < endTime){
264             schedule.schedule(ScheduleParameters.createOneTime(nextArrivalTime, 1), this, "arriveAtPort");
265         }
266     }
267 
268     /**
269      * The statistics are written to excel after which they are cleared. The simulation ends.
270      */
271     public void end(){
272             stats.toExcel();
273             Port.stats.resetStats();
274             RunEnvironment.getInstance().endRun(); // end the simulation
275             System.out.println("Scerario "+Port.scenarioCount+" completed.");
276     }
```

```
277
278
279     /**
280      * Put the sailing times from the spreadsheet into a HashBasedTable. The table is used by a
281      * barge to construct a smaller sailing times array with only the relevant terminals. The barge
282      * will use the smaller version of the table as input for the TDTSP.
283      */
284     public void sailingtimesTable() {
285
286         String filePath; //the file path depends on whether the simulation is part of a batch run
287         if(RunEnvironment.getInstance().isBatch()==true){
288             filePath = "C:/jbarges/data.xlsx";
289         }
290         else{
291             filePath = "src/data.xlsx";
292         }
293
294         int[][] table=null;
295         try {
296             FileInputStream file = new FileInputStream(new File(filePath));
297
298             // get the file
299             Workbook workbook = new XSSFWorkbook(file);
300
301             // get the sheet
302             Sheet sheet = workbook.getSheet(sheetName);
303
304             // create iterator
305             Iterator<Row> rowIterator = sheet.iterator();
306
307             // create an array to store the content. the array will later in this method be converted to a HashBasedTable
308             int noOfColumns = sheet.getRow(0).getPhysicalNumberOfCells();
309             table = new int[noOfColumns][noOfColumns];
310
311             // store the content in the array
312             for(int i = 0; i<noOfColumns; i++){
313                 Row row = rowIterator.next();
314                 Iterator<Cell> cellIterator = row.cellIterator();
315                 for (int j = 0; j<noOfColumns; j++){
316                     Double cell = cellIterator.next().getNumericCellValue();
317                     table[i][j]= cell.intValue();
318                 }
319             }
320             file.close();
321
322         } catch (FileNotFoundException e1) {
323             e1.printStackTrace();
324         } catch (IOException e) {
325             e.printStackTrace();
326         }
327
328         // convert to a HashBasedTable
329         sailingtimesTable = HashBasedTable.create();
330         for(int i=0; i<terminals.size(); i++){
331             for(int j=0; j<terminals.size();j++){
332                 sailingtimesTable.put(terminals.get(i), terminals.get(j), table[i][j]);
333             }
334         }
335     }
336 }
337
```

```java
package jbarges;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.Map;

import repast.simphony.context.Context;
import repast.simphony.engine.schedule.ScheduleParameters;
import repast.simphony.util.ContextUtils;
import tdtsp.TDTSP;

import com.google.common.collect.Table;

/**
 * The Barge object represents the barge agent and everything that is associated with a barge.
 */
public class Barge{

    /**
     * The barge number is used to identify the barge. The first barge gets the number 0 so
     * that it is the same as the index number of the barge list in the simulation class.
     */
    public int bargeNumber;

    /**
     * The arrival time at the port. This is derived from the tick count of the simulation
     * schedule and is used as input for the TDTSP.
     */
    public int arrivalTime;

    /**
     * The terminals this barge has to visit. This list is parallel to the handlingTimes List.
     * At index 0 is the port entrance.
     */
    public ArrayList<Terminal> terminals;

    /**
     * The handling time at each terminal. This list is parallel to the terminals List.
     */
    public ArrayList<Integer> handlingTimes;

    /**
     * This is a smaller version of the sailing times table in the Simulation class.
     * It contains only the terminals this barge has to visit and is used as input for the TDTSP.
     */
    public int[][] sailingTimes;

    /**
     * Stores the waiting profile for each terminal. It is used as input for the TDTSP.
     */
    public Map<Terminal, WaitingProfile> waitingProfiles;

    /**
     * This TDTSP object handles the computation of the route and also stores all
     * information associated with it.
     */
    public TDTSP tdtsp;

    /**
     * This map stores the appointments. The key of the map is the Terminal. The value is
     * an integer array with the following indexes: 0=LAT, 1=LST. LAT=Latest arrival time,
     * LST=Latest starting time.
     */
    Map<Terminal,int[]> appointments;

    /**
     * The remaining route is derived from the bestRoute in the TDTSP object.
     * It is used as input for scheduleArrivalTerminal method.
     * Every time a barge is finished at a terminal a node is removed from the remaining route.
     */
    String remainingRoute;

    /**
     * To store the appointments of the terminals at the time of constructing the waiting profile.
     * This is only used to print the appointments together with the barge info. This is needed when
     * the correctnes of the waiting profiles is checked.
     */
    public String terminalAppointments;

    /**
     * Barge state.
     */
    int state;
    public static final int SAILING = 1;
    public static final int WAITING = 2;
    public static final int HANDLING = 3;

    /**
     * Barge statistic.
     */
    public int actualSojourntime, totalSailingTime, totalWaitingTime, totalHandlingTime, differenceExpectedActual;
```

```
93
94      /**
95       * Stores the satisfaction on a scale from 1 to 7, where 1 is completely dissatisfied and 7 completely satisfied.
96       * The informationSatisfaction relates to the difference between the expected and actual sojourn time.
97       * The waitingtimeSatisfaction relates to the waiting time with respect to the service time.
98       */
99      public int informationSatisfaction, waitingtimeSatisfaction;
100
101     /**
102      * Fraction waiting time with respect to handling time. Used to determine the waitingtimeSatisfaction.
103      */
104     public double fraction;
105
106     /**
107      * The constructor of class Barge. This constructor assigns all attributes of a barge.
108      * It also runs the commands that requests the waiting profiles from the terminals,
109      * computes the best route using TDTSP and it schedules the arrival at the
110      * first terminal.
111      * @param bargeNumber The barge number is used to identify the barge
112      * @param arrivalTime The arrival time at the port
113      * @param terminals The terminals this barge has to visit
114      * @param handlingtimes The handling time at each terminal
115      */
116     public Barge(int bargeNumber, int arrivalTime, ArrayList<Terminal> terminals,
117             ArrayList<Integer> handlingtimes) {
118
119         this.bargeNumber = bargeNumber;
120         this.arrivalTime = arrivalTime;
121         this.terminals = terminals;
122         this.handlingTimes = handlingtimes;
123
124         this.totalSailingTime=0;
125         this.totalWaitingTime=0;
126         this.totalHandlingTime=0;
127
128         this.createSailingTimes(Port.sailingtimesTable);
129         this.requestWaitingProfiles();
130
131         // start time for TDTSP = arrival time in port
132         this.tdtsp = new TDTSP(arrivalTime, this);
133
134         // make appointments
135         this.addAppointments();
136
137         // save the appointments of terminals so that they can be written to the spreadsheet
138         if(Port.eventsToExcel.equals("Yes")){
139             this.saveAppointmentsterminals();
140         }
141
142         this.state = SAILING;
143
144         // to remove the first space of bestRoute we take this substring of bestRoute.
145         this.remainingRoute = this.tdtsp.bestRoute.substring(1);
146
147         // schedule the arrival at the first terminal.
148         this.scheduleArrivalTerminal(this.remainingRoute, this.arrivalTime);
149     }
150
151     /**
152      * Creates the two dimensional sailing times array that only contains the terminals this
153      * barge has to visit. It is stored in sailingTimes.
154      * @param sailingTimeTable The sailing times table containing all terminals in the port
155      */
156     public void createSailingTimes(Table<Terminal,Terminal,Integer> sailingTimeTable){
157         int n = terminals.size();
158         this.sailingTimes = new int[n][n];
159
160         for(int i=0; i<n; i++){
161             for(int j=0; j<n; j++){
162                 Terminal ti = terminals.get(i);
163                 Terminal tj = terminals.get(j);
164                 this.sailingTimes[i][j] = sailingTimeTable.get(ti,tj);
165             }
166         }
167     }
168
169     /**
170      * Requests waiting profiles from terminals and stores them in waitingProfiles.
171      */
172     public void requestWaitingProfiles(){
173         this.waitingProfiles = new HashMap<Terminal, WaitingProfile>();
174
175         for(Terminal terminal : terminals){
176             WaitingProfile waitingProfile = terminal.constructWaitingProfile(this,
177                     this.arrivalTime);
178             waitingProfiles.put(terminal, waitingProfile);
179         }
180     }
181
182     /**
183      * Makes appointments with terminals using the best rotation.
184      */
```

```
185     public void addAppointments(){
186
187         this.appointments = new LinkedHashMap<Terminal, int[]>();
188
189         String bestRoute = tdtsp.bestRoute;
190         int departureTime = this.arrivalTime;
191         int departureNode = 0;
192         for(int i=0; i<bestRoute.length(); i++){
193             String c = ""+bestRoute.charAt(i);
194             // check if the character is a " ". example of bestRoute: " 1 2 3 4 0"
195             if(!c.equals(" ")){
196                 // return terminal object at index c from the terminals list
197                 int destination = Integer.parseInt(c);
198                 Terminal ter = this.terminals.get(destination);
199
200                 // compute latest arrival time (LAT) and latest starting time (LST)
201                 int LAT = departureTime + this.sailingTimes[departureNode][destination];
202                 int LST = LAT + this.waitingProfiles.get(ter).getMaxWaitingTime(LAT);
203
204                 // add the appointment to the schedule of this barge
205                 appointments.put(ter, new int[]{LAT,LST});
206
207                 // send the LAT, LST and handling time to the terminal
208                 int handlingTime = this.handlingTimes.get(destination).intValue();
209                 ter.addAppointment(this, LAT, LST, handlingTime);
210
211                 //update departureTime and departureNode for next iteration
212                 departureTime = LST + handlingTime;
213                 departureNode = destination;
214             }
215         }
216     }
217
218     /**
219      * Schedules the arrival at the next terminal.
220      * @param route the (remaining) route. Example of the String: "0 1 2 3 4 5 0"
221      * @param time departure time from the previous point
222      */
223     public void scheduleArrivalTerminal(String route, int time){
224
225         // determine sailing time
226         int sailingTime=0;
227         Terminal terminalDestin = null;
228         for(int i=0; i<route.length()-2; i++){
229             String c = ""+route.charAt(i);
230             if(!c.equals(" ")){
231                 int origin = Integer.parseInt(c);
232                 // the destination node is 2 indexes further in String route
233                 String d = ""+route.charAt(i+2);
234                 int destin = Integer.parseInt(d);
235
236                 // get sailing time between the depot node and the first terminal
237                 sailingTime=this.sailingTimes[origin][destin];
238
239                 // set terminal to this terminal, this is input for the scheduled action below
240                 terminalDestin = this.terminals.get(destin);
241
242                 // end loop
243                 break;
244             }
245         }
246
247         if(Port.model.equals("Stochastic")){
248             sailingTime = (int) Math.round(Port.timeRNG.nextGaussian(sailingTime, Port.timeSigma));
249             if(sailingTime<1){
250                 sailingTime=1;
251             }
252         }
253
254         // add to statistic
255         this.totalSailingTime+=sailingTime;
256
257         // arrival time at the terminal
258         int arrivalTimeTerminal = time + sailingTime;
259
260         // schedule the arrival in the simulation schedule
261         Port.schedule.schedule(
262                 ScheduleParameters.createOneTime(arrivalTimeTerminal), this, "arriveAtTerminal",
263                 terminalDestin, arrivalTimeTerminal);
264
265     }
266
267     /**
268      * [ISchedulableAction] Arrive at Terminal. The terminal decides when the handling starts.
269      * @param terminal The terminal at which the barge arrives
270      * @param time The arrival time at the terminal
271      */
272     public void arriveAtTerminal(Terminal terminal, int time){
273
274         // add to queue, even if the terminal is idle. the terminal agent will take care of the rest
275         terminal.queue.add(this);
276
```

```
277         // set state of barge to waiting
278         this.state = Barge.WAITING;
279 
280         if(Port.eventsToExcel.equals("Yes")){
281                 Port.stats.addEvent(time, this.bargeNumber, ("Arrived at Terminal " + terminal.toString()+
282                         ". Expected - actual = " + this.appointments.get(terminal)[0]+" - " + time  + " = "
283                         + (this.appointments.get(terminal)[0]-time)));
284         }
285 
286         // schedule the start handling of this barge at the terminal in the simulation schedule
287         Port.schedule.schedule(
288                 ScheduleParameters.createOneTime(time), terminal, "bargeArrives", this, time);
289     }
290 
291     /**
292      * Actions for the barge after it finished handling at a terminal.
293      * @param time the time it finished handling at the last visited terminal
294      */
295     public void afterFinish(int time){
296 
297         this.state=Barge.SAILING;
298 
299         // if the barge visits more than 1 terminal
300         // remove a node from remainingRoute (example: "1 2 3 4 0" ---> "2 3 4 0")
301         if(this.remainingRoute.length()>3){
302             this.remainingRoute = this.remainingRoute.substring(2);
303         }
304 
305         // check if there is another terminal to visit
306         if(this.remainingRoute.length()>3){
307             this.scheduleArrivalTerminal(this.remainingRoute, time);
308         }
309         else{
310             // sail to port exit point, i.e., schedule leaving the port in the simulation schedule
311             int lastTerminal = Integer.parseInt(""+this.remainingRoute.charAt(0));
312             int exitPoint = Integer.parseInt(""+this.remainingRoute.charAt(2));
313             int sailingTime = this.sailingTimes[lastTerminal][exitPoint];
314             if(Port.model.equals("Stochastic")){
315                 sailingTime = (int) Math.round(Port.timeRNG.nextGaussian(sailingTime, Port.timeSigma));
316                 if(sailingTime<1){
317                     sailingTime=1;
318                 }
319             }
320             // add to statistic
321             this.totalSailingTime+=sailingTime;
322             // schedule arrival at exit point
323             int arrivalTimeAtExit = time + sailingTime;
324             Port.schedule.schedule(ScheduleParameters.createOneTime(arrivalTimeAtExit,ScheduleParameters.LAST_PRIORITY)
325                     , this, "leavePort", arrivalTimeAtExit);
326         }
327     }
328 
329     /**
330      * [ISchedulableAction] Leaves the port.
331      * @param time the time the barges leaves the port
332      */
333     public void leavePort(int time){
334 
335         //only save statistics of barge 301 to 800.
336         if(this.bargeNumber>300 && this.bargeNumber<=800){
337 
338             Port.stats.bargesLeftPort++;
339 
340             Port.stats.descriptiveStatistics[0].addValue(this.tdtsp.bestSojournTime);
341 
342             this.actualSojourntime = time-this.arrivalTime;
343             Port.stats.descriptiveStatistics[1].addValue(this.actualSojourntime);
344 
345             Port.stats.descriptiveStatistics[3].addValue(this.totalHandlingTime);
346             Port.stats.descriptiveStatistics[4].addValue(this.totalSailingTime);
347 
348             this.totalWaitingTime = time-this.arrivalTime - this.totalHandlingTime - this.totalSailingTime;
349             Port.stats.descriptiveStatistics[2].addValue(this.totalWaitingTime);
350 
351             differenceExpectedActual = this.tdtsp.bestLeaveTime-time;
352 
353             this.determineSatisfaction();
354 
355             if(Port.eventsToExcel.equals("Yes")){
356                 Port.stats.addEvent(time, this.bargeNumber, ("Left port. Expected - actual = " +
357                         this.tdtsp.bestLeaveTime + " - "+ time + " = " + differenceExpectedActual));
358             }
359 
360             //add to barge stats
361             if(Port.bargesDetailsToExcel.equals("Yes")){
362                 Port.stats.addSinglebargeinfo(this);
363             }
364         }
365 
366         // remove from context: first get the context by using one of the terminals, then use it to remove the barge
367         @SuppressWarnings("unchecked")
368         Context<Object> context = ContextUtils.getContext(Port.terminals.get(0));
```

```
369         context.remove(this);
370     }
371 
372     public String toString(){
373         return "Barge "+ this.bargeNumber;
374     }
375 
376     /**
377      *
378      * @return The appointments in a String
379      */
380     public String appointmentsToString(){
381         String str = "Schedule of barge "+ this.bargeNumber + "\n";
382         for(Terminal t : this.appointments.keySet()){
383             str += "Terminal " + t.name + " ";
384             for(int i : this.appointments.get(t)){
385                 str += i +" ";
386             }
387             str += "\n";
388         }
389         return str+"\n";
390     }
391 
392     /**
393      * Saves the appointment of the terminals at the time of constructing the waiting profile.
394      * This is required to save the appointments to barge_info in the spreadsheet.
395      *
396      */
397     public void saveAppointmentsterminals(){
398         //the appointments of the terminals this barge has to visit
399         terminalAppointments="";
400         for(int i=1; i<this.terminals.size(); i++){
401             terminalAppointments+=this.terminals.get(i).appointmentsToString();
402         }
403     }
404 
405     /**
406      * Checks the satisfaction/happiness with (1) the information providing regarding the expected sojourn time
407      * and (2) the waiting time with respect to the service time.
408      */
409     public void determineSatisfaction(){
410 
411         //checks if the difference between the expected and actual sojourn time is acceptable.
412         if(this.differenceExpectedActual > 179){
413             this.informationSatisfaction = 4;
414         }
415         else if(this.differenceExpectedActual > 37){
416             this.informationSatisfaction = 5;
417         }
418         else if(this.differenceExpectedActual > 7){
419             this.informationSatisfaction = 6;
420         }
421         else if(this.differenceExpectedActual > -3){
422             this.informationSatisfaction = 7;
423         }
424         else if(this.differenceExpectedActual > -7){
425             this.informationSatisfaction = 6;
426         }
427         else if(this.differenceExpectedActual > -15){
428             this.informationSatisfaction = 5;
429         }
430         else if(this.differenceExpectedActual > -34){
431             this.informationSatisfaction = 4;
432         }
433         else if(this.differenceExpectedActual > -79){
434             this.informationSatisfaction = 3;
435         }
436         else if(this.differenceExpectedActual > -180){
437             this.informationSatisfaction = 2;
438         }
439         else{
440             this.informationSatisfaction = 1;
441         }
442 
443         //checks if the waiting time is proportional to the service time
444         fraction = (double)this.totalWaitingTime / (double)this.totalHandlingTime;
445 
446         if(fraction < 0.61){
447             this.waitingtimeSatisfaction = 7;
448         }
449         else if(fraction < 0.76){
450             this.waitingtimeSatisfaction = 6;
451         }
452         else if(fraction < 0.95){
453             this.waitingtimeSatisfaction = 5;
454         }
455         else if(fraction < 1.2){
456             this.waitingtimeSatisfaction = 4;
457         }
458         else if(fraction < 1.54){
459             this.waitingtimeSatisfaction = 3;
460         }
```

```
461         else if(fraction < 2){
462             this.waitingtimeSatisfaction = 2;
463         }
464         else{
465             this.waitingtimeSatisfaction = 1;
466         }
467     }
468 }
469
```

```java
package jbarges;

import java.util.LinkedHashMap;
import java.util.LinkedList;
import java.util.Map;
import java.util.Queue;

import repast.simphony.engine.schedule.ScheduleParameters;

/**
 * Represents the terminal agent and everything that is associated with a terminal.
 */
public class Terminal {

    /**
     * The name of the terminal. This is used to identify the terminal.
     */
    String name;

    /**
     * The appointments this terminal made with barges are stored in this
     * Map. The key is the barge. The value is an integer array. The meaning
     * of the indexes in the integer array are as follows:
     * 0 = LAT, 1 = LST, 2 = PST, 3 = PT, 4 = EDT
     */
    Map<Barge,int[]> appointments;

    /**
     * The number of barges handling at the terminal.
     */
    int numHandling;

    /**
     * The queue at this terminal.
     */
    Queue<Barge> queue;

    /**
     * States
     */
    public int state;
    public static final int HANDLING = 1;
    public static final int IDLE = 2;

    /**
     * @param name the name of the terminal. This is used to identify the terminal.
     */
    public Terminal(String name) {
        this.name = name;
        this.appointments = new LinkedHashMap<Barge,int[]>();
        this.queue = new LinkedList<Barge>();
        this.state = IDLE;
        this.numHandling = 0;
    }

    /**
     * Add a barge to the schedule. The schedule should be sorted on the
     * latest starting time in order to construct waiting profiles. Because
     * LinkedHashMap (in which schedule is stored) does not provide this
     * functionality out of the box, we implemented that in this method as well.
     * @param barge barge
     * @param LAT latest arrival time
     * @param LST latest starting time
     * @param PST planned starting time
     * @param PT processing time (handling time)
     * @param EDT expected departure time
     */
    public void addAppointment(Barge barge, int LAT, int LST, int PT){
        // determine position where the appointment should be inserted.
        // the schedule should be sorted on the LST in order to create
        // a right waiting profile
        // to keep track of the position we use integer i.
        int i = 0;
        for(Barge b : this.appointments.keySet()){
            //compare LAT of Barge b with the LAT of the new barge
            if(LST > this.appointments.get(b)[1]){
                i++;
            }
            else{
                break;
            }
        }

        //create a new LinkedHashMap that replaces appointments at the end of this method
        LinkedHashMap<Barge,int[]> newAppointments = new LinkedHashMap<Barge,int[]>();

        for(int j=0; j<appointments.size()+1; j++){

            if(j<i){
                //add the appointment from appointments to newAppointments
                Barge key = (Barge) appointments.keySet().toArray()[j];
                int[] value = appointments.get(key);
```

```
93                  newAppointments.put(key, value);
94              }
95              else if(j==i){
96                  //add the new appointment at this position
97                  int PST = LST;
98                  int EDT = PST + PT;
99                  newAppointments.put(barge, new int[]{LAT,LST, PST, PT, EDT});
100             }
101             else{
102                 //add the remaining appointment after the new appointment
103                 Barge key = (Barge) appointments.keySet().toArray()[j-1];
104                 int[] value = appointments.get(key);
105                 newAppointments.put(key, value);
106             }
107         }
108         //replace appointments with newAppointments
109         this.appointments = newAppointments;
110     }
111
112     /**
113      * Construct and send the waiting profile
114      */
115     public WaitingProfile constructWaitingProfile(Barge barge, int currentTime){
116         return new WaitingProfile(this, barge, currentTime);
117     }
118
119     /**
120      * [ISchedulableAction] Starts the handling of a barge
121      * @param barge the barge that arrived at the terminal
122      * @param time the time at which the barge tries to start handing. this is the arrival time at the terminal at the first try
123      */
124     public void bargeArrives(Barge barge, int time){
125
126         if(Port.terminalLogic.equals("Unreserved") && this.state==Terminal.IDLE){
127             Port.schedule.schedule(ScheduleParameters.createOneTime(time,ScheduleParameters.LAST_PRIORITY), this, "handleBarge", I
128         }
129         else if(Port.terminalLogic.equals("Reserved") && this.state==Terminal.IDLE){
130             //we need the following information to check whether or not the barge can start handling
131             Barge nextBargeInSchedule = (Barge)this.appointments.keySet().toArray()[0];
132             int expectedEndTimeThisBarge = time+barge.handlingTimes.get(barge.terminals.indexOf(this));
133             int lstNextAppointment = this.appointments.get(nextBargeInSchedule)[1];
134
135             //is this barge the next barge in the schedule, then start handling
136             if(barge.equals(nextBargeInSchedule)==true || expectedEndTimeThisBarge<=lstNextAppointment){
137                 Port.schedule.schedule(ScheduleParameters.createOneTime(time,ScheduleParameters.LAST_PRIORITY), this, "handleBarge
138             }
139         }
140     }
141
142     /**
143      * [ISchedulableAction] Finish handling of a barge
144      * @param barge barge that finished
145      * @param time finish time
146      */
147     public void finishHandling(Barge barge, int time){
148
149         // remove the barge from appointments of terminal
150         this.appointments.remove(barge);
151         this.numHandling--;
152
153
154         if(this.queue.size()==0){
155             this.state=Terminal.IDLE;
156         }
157         else if(Port.terminalLogic.equals("Unreserved")){
158             Barge nextBarge = this.queue.peek();
159             Port.schedule.schedule(ScheduleParameters.createOneTime(time,ScheduleParameters.LAST_PRIORITY), this, "handleBarge", I
160
161         }
162         else if(Port.terminalLogic.equals("Reserved")){
163             Barge nextBargeInSchedule = (Barge)this.appointments.keySet().toArray()[0];
164             if(this.queue.contains(nextBargeInSchedule)==true){
165                 Port.schedule.schedule(ScheduleParameters.createOneTime(time,ScheduleParameters.LAST_PRIORITY), this, "handleBarge
166             }
167             else{
168                 int lstNextAppointment = this.appointments.get(nextBargeInSchedule)[1];
169                 boolean startNextBarge=false;
170                 for(Barge nextBarge : this.queue){
171                     int expectedEndTimeThisBarge = time+nextBarge.handlingTimes.get(nextBarge.terminals.indexOf(this));
172                     if(expectedEndTimeThisBarge<=lstNextAppointment){
173                         Port.schedule.schedule(ScheduleParameters.createOneTime(time,ScheduleParameters.LAST_PRIORITY), this, "har
174                         startNextBarge=true;
175                         break;
176                     }
177                 }
178                 if(startNextBarge==false){
179                     this.state=Terminal.IDLE;
180                 }
181             }
182         }
183
184         if(Port.eventsToExcel.equals("Yes")){
```

```
185             Port.stats.addEvent(time, barge.bargeNumber, ("Finished handling at Terminal "+this.toString()));
186         }
187 
188         //let the barge decide what to do after it finished handling
189         barge.afterFinish(time);
190     }
191 
192     /**
193      * used to display the queue size in the chart in the GUI
194      */
195     public int getQueueSize(){
196         return this.queue.size();
197     }
198 
199     /**
200      * used to display the number of busy terminals in the GUI
201      */
202     public int getNumhandling(){
203         return this.numHandling;
204     }
205 
206     /**
207      * Start handling
208      * @param barge
209      * @param currentTime
210      */
211     public void handleBarge(Barge barge, int currentTime){
212         // start handling
213         // set the states of the barge and terminal to handling
214         barge.state=Barge.HANDLING;
215         this.state=Terminal.HANDLING;
216         this.numHandling++;
217 
218         if(Port.eventsToExcel.equals("Yes")){
219             Port.stats.addEvent(currentTime, barge.bargeNumber, ("Started handling at Terminal "+this.toString()+
220                     ". Expected - actual = " + barge.appointments.get(this)[1]+ " - " + currentTime + " = "
221                      + (barge.appointments.get(this)[1]-currentTime)));
222         }
223 
224         //remove the barge from the queue
225         this.queue.remove(barge);
226 
227         //get handling time
228         int handlingTime = barge.handlingTimes.get(barge.terminals.indexOf(this));
229         if(Port.model.equals("Stochastic")){
230             handlingTime = (int) Math.round(Port.timeRNG.nextGaussian(handlingTime, Port.timeSigma));
231             if(handlingTime<10){
232                 handlingTime=10;
233             }
234         }
235 
236         // add handling time to total (actual) handling time statistic of the barge
237         barge.totalHandlingTime += handlingTime;
238 
239         //schedule finishHandling. at start time handling + handling time
240         int finishTime = currentTime + handlingTime;
241         Port.schedule.schedule(ScheduleParameters.createOneTime(finishTime,ScheduleParameters.FIRST_PRIORITY), this, "finishHandli
242     }
243 
244     /**
245      * @return A String representation of the appointments this terminal has with barges
246      */
247     public String appointmentsToString(){
248         String str = "Schedule of terminal "+ this.name + "\n";
249         for(Barge b : this.appointments.keySet()){
250             str += "Barge " + b.bargeNumber + " ";
251             for(int i : this.appointments.get(b)){
252                 str += i +" ";
253             }
254             str += "\n";
255         }
256         return str+"\n";
257     }
258 
259     /**
260      * @param time
261      * @return A String representation of the queue at this terminal
262      */
263     public String queueToString(int time){
264         String str = "Queue of terminal "+ this.name + "at time "+time+ "\n";
265         for(Barge b: this.queue){
266             str+="Barge "+b.bargeNumber+"\n";
267         }
268         return str;
269     }
270 
271     public String toString(){
272         return name;
273     }
274 }
275 
```

```java
package jbarges;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Date;

import org.apache.commons.math3.stat.descriptive.DescriptiveStatistics;
import org.apache.commons.math3.util.Precision;
import org.apache.poi.ss.usermodel.Row;
import org.apache.poi.ss.usermodel.Sheet;
import org.apache.poi.ss.usermodel.Workbook;
import org.apache.poi.xssf.usermodel.XSSFWorkbook;

import repast.simphony.engine.environment.RunEnvironment;

/**
 * Manages the statistics.
 */

public class Statistics {

    public int bargesEnteredPort, bargesLeftPort, bargesInPort, bargesInPortafterWarmup, bargeCount;


    /**
     * Construct DescriptiveStatistics from org.apache.commons.math3.
     * Array indexes: 0=expectedSojournTime, 1=actualSojournTime, 2=totalWaitingTime,
     * 3=totalHandlingTime, 4=totalSailingTime
     */
    public DescriptiveStatistics[] descriptiveStatistics;

    public ArrayList<Object[]> singleBargeinfo, events;


    public Statistics(){


        bargesEnteredPort=0;
        bargesLeftPort=0;
        bargesInPort=bargesEnteredPort-bargesLeftPort;

        bargeCount = 0;

        descriptiveStatistics = new DescriptiveStatistics[5];
        for(int i = 0; i<5; i++){
            descriptiveStatistics[i]= new DescriptiveStatistics();
        }

        singleBargeinfo = new ArrayList<Object[]>();
        events = new ArrayList<Object[]>();
    }


    public void addEvent(int time, int bargeNumber, String description){

        Object[] event = new Object[]{
            time,
            bargeNumber,
            description
        };

        this.events.add(event);
    }

    public void addSinglebargeinfo(Barge barge){

        //terminals to visit
        String terminalsToVisit="";
        for(int i=1; i<barge.terminals.size();i++){
            Terminal t = barge.terminals.get(i);
            terminalsToVisit+= t+" ";
        }

        //handling time (parallel to the terminal to visit)
        String handlingtimes="";
        for(int i=1; i<barge.handlingTimes.size();i++){
            Integer htime = barge.handlingTimes.get(i);
            handlingtimes+=htime+" ";
        }

        //waiting profiles
        String wprofiles="";
        for(int i=1; i<barge.terminals.size(); i++){
            Terminal t = barge.terminals.get(i);
            WaitingProfile wp = barge.waitingProfiles.get(t);
            wprofiles += "Waiting Profile of " + t.toString()+ "\n"+ wp.toString() + "\n";
            wprofiles += "Start invervals of " + t.toString()+ "\n" + wp.startIntervalsToString() + "\n";
        }
```

```
 93
 94
 95          //sailing times
 96          String sailingTimes="";
 97          for(int[] i : barge.sailingTimes){
 98              for(int j : i){
 99                  sailingTimes+=j+" ";
100              }
101              sailingTimes+="\n";
102          }
103
104
105          Object[] bargeInfo = new Object[]{
106              Port.scenarioCount,
107                  barge.bargeNumber,
108                  barge.arrivalTime,
109                  barge.terminals.size()-1,
110                  terminalsToVisit,
111                  handlingtimes,
112                  barge.tdtsp.bestRouteToString(),
113                  //wprofiles,
114                  //sailingTimes,
115                  //barge.appointmentsToString(),
116                  //barge.terminalAppointments,
117                  barge.tdtsp.bestSojournTime,
118                  barge.actualSojourntime,
119                  barge.totalWaitingTime,
120                  barge.totalHandlingTime,
121                  barge.totalSailingTime,
122                  barge.differenceExpectedActual,
123                  barge.fraction,
124                  barge.informationSatisfaction,
125              barge.waitingtimeSatisfaction
126          };
127
128          this.singleBargeinfo.add(bargeInfo);
129      }
130
131
132      public void toExcel(){
133
134          String filePath;
135          if(RunEnvironment.getInstance().isBatch()==true){
136              filePath = "C:/jbarges/output.xlsx";
137          }
138          else{
139              filePath = "output/output.xlsx";
140          }
141
142
143          try {
144
145              //check if file exists, else create the file
146              File f = new File(filePath);
147              FileInputStream file;
148              if(f.exists() && !f.isDirectory()){
149                  file = new FileInputStream(new File(filePath));
150              }
151              else{
152                  FileOutputStream out =
153                          new FileOutputStream(new File(filePath));
154                  Workbook workbook = new XSSFWorkbook();
155
156                  //create sheet with average statistics
157                  Sheet sheet = workbook.createSheet("Simulations");
158                  //add headings in an array
159                  Object[] heading = new Object[]{
160                          "Run date",
161                          "Random seed",
162                          "Terminal logic",
163                          "Actual sailing and handling times",
164                          //"Run time",
165                          //"Warm-up time",
166                          //"Time deviation (in case of stochastic)",
167                          "Slack method",
168                          "Slack constant",
169                          "Slack denominator",
170                          //"Mean terminals to visit",
171                          //"SD terminals to visit",
172                          //"Mean handling time",
173                          //"SD handling time",
174                          "Arrival rate",
175                          "Barges entered the port after warm-up",
176                          "Barges left the port after warm-up",
177                          "Barges in port after warm-up",
178                          "Mean expected sojourn time",
179                          "SD expected sojourn time",
180                          "Min expected sojourn time",
181                          "Max expected sojourn time",
182                          "Mean actual sojourn time",
183                          "SD actual sojourn time",
184                          "Min actual sojourn time",
```

```
                    "Max actual sojourn time",
                    "Mean waiting time",
                    "SD waiting time",
                    "Min waiting time",
                    "Max waiting time",
                    "Mean handling time",
                    "SD handling time",
                    "Min handling time",
                    "Max handling time",
                    "Mean sailing time",
                    "SD sailing time",
                    "Min sailing time",
                    "Max sailing time",

                    "# Completely dissatisfied with info",
                    "# Mostly dissatisfied with info",
                    "# Somewhat dissatisfied with info",
                    "# Neither satisfied or dissatisfied with info",
                    "# Somewhat satisfied with info",
                    "# Mostly satisfied with info",
                    "# Completely satisfied with info",

                    "# Completely dissatisfied with waiting time",
                    "# Mostly dissatisfied with waiting time",
                    "# Somewhat dissatisfied with waiting time",
                    "# Neither satisfied or dissatisfied with waiting time",
                    "# Somewhat satisfied with waiting time",
                    "# Mostly satisfied with waiting time",
                    "# Completely satisfied with waiting time"
                };
            //add the heading to the sheet
            Row row = sheet.createRow(sheet.getLastRowNum());
            int cellIndex=0;
            for(Object cellContent : heading){
            row.createCell(cellIndex).setCellValue(cellContent.toString());
            cellIndex++;
            }

            //create the sheet with the single barge information
            Sheet sheet2 = workbook.createSheet("Barge info");
            //add headings
            Object[] heading2 = new Object[]{
                "Scenario",
                    "Barge number",
                    "Arrival time",
                    "Number of terminals to visit",
                    "Terminals to visit",
                    "Handling times",
                    "Best route",
                    //"Waiting profiles",
                    //"Sailing times table",
                    //"Appointments",
                    //"Appointments of terminals",
                    "Expected sojourn time",
                    "Actual sojourn time",
                    "Total waiting time",
                    "Total handling time",
                    "Total sailing time",
                    "Difference between expected and actual sojourntime",
                    "Fraction waiting time with respect to handling time",
                    "Information provision satisfaction",
                "Waiting time satisfaction"
            };
            Row row2 = sheet2.createRow(sheet.getLastRowNum());
            int cellIndex2=0;
            for(Object cellContent : heading2){
            row2.createCell(cellIndex2).setCellValue(cellContent.toString());
            cellIndex2++;
            }


            //create the sheet with the events
            Sheet sheet3 = workbook.createSheet("Events");

            //add headings
            Object[] heading3 = new Object[]{
                "Time",
                    "Barge",
                    "Event",
            };
            Row row3 = sheet3.createRow(sheet.getLastRowNum());
            int cellIndex3=0;
            for(Object cellContent : heading3){
            row3.createCell(cellIndex3).setCellValue(cellContent.toString());
            cellIndex3++;
            }

            //save the file
            workbook.write(out);
            out.close();
            file = new FileInputStream(new File(filePath));
        }
```

```
277
278             Workbook workbook = new XSSFWorkbook(file);
279             Sheet sheet = workbook.getSheetAt(0);
280
281
282             //count satisfaction levels. satisfaction rating 1 is located at index 0, rating 2 is located at index 1, etc.
283             int infoSatisfaction[] = new int[]{0,0,0,0,0,0,0};
284             int waitSatisfaction[] = new int[]{0,0,0,0,0,0,0};
285             for(Object[] o : this.singleBargeinfo){
286             //the information satisfaction rating is located at index 14, waiting time satisfaction rating at index 15
287             int info = (int) o[14];
288             int wait = (int) o[15];
289             infoSatisfaction[info-1]++;
290             waitSatisfaction[wait-1]++;
291             }
292
293             // this array is used to fill the row
294             Object[] rowContent = new Object[]{
295             new Date(System.currentTimeMillis()),
296             Port.seed,
297                 Port.terminalLogic,
298                 Port.model,
299                 //Port.endTime,
300                 //Port.warmup,
301                 //Port.timeSigma,
302                 Port.slackMethod,
303
304                 Port.slack,
305
306                 Port.slackDenominator,
307
308                 //Port.numTerminalMean,
309                 //Port.numTerminalStd,
310                 //Port.handlingTimeMean,
311                 //Port.handlingTimeStd,
312                 Port.arrivalRate,
313
314                 this.bargesEnteredPort,
315                 this.bargesLeftPort,
316                 this.bargesInPortafterWarmup,
317
318                 Precision.round(this.descriptiveStatistics[0].getMean(),0),
319                 Precision.round(this.descriptiveStatistics[0].getStandardDeviation(),0),
320                 Precision.round(this.descriptiveStatistics[0].getMin(),0),
321                 Precision.round(this.descriptiveStatistics[0].getMax(),0),
322
323                 Precision.round(this.descriptiveStatistics[1].getMean(),0),
324                 Precision.round(this.descriptiveStatistics[1].getStandardDeviation(),0),
325                 Precision.round(this.descriptiveStatistics[1].getMin(),0),
326                 Precision.round(this.descriptiveStatistics[1].getMax(),0),
327
328                 Precision.round(this.descriptiveStatistics[2].getMean(),0),
329                 Precision.round(this.descriptiveStatistics[2].getStandardDeviation(),0),
330                 Precision.round(this.descriptiveStatistics[2].getMin(),0),
331                 Precision.round(this.descriptiveStatistics[2].getMax(),0),
332
333                 Precision.round(this.descriptiveStatistics[3].getMean(),0),
334                 Precision.round(this.descriptiveStatistics[3].getStandardDeviation(),0),
335                 Precision.round(this.descriptiveStatistics[3].getMin(),0),
336                 Precision.round(this.descriptiveStatistics[3].getMax(),0),
337
338                 Precision.round(this.descriptiveStatistics[4].getMean(),0),
339                 Precision.round(this.descriptiveStatistics[4].getStandardDeviation(),0),
340                 Precision.round(this.descriptiveStatistics[4].getMin(),0),
341                 Precision.round(this.descriptiveStatistics[4].getMax(),0),
342
343                 infoSatisfaction[0],
344                 infoSatisfaction[1],
345                 infoSatisfaction[2],
346                 infoSatisfaction[3],
347                 infoSatisfaction[4],
348                 infoSatisfaction[5],
349                 infoSatisfaction[6],
350
351                 waitSatisfaction[0],
352                 waitSatisfaction[1],
353                 waitSatisfaction[2],
354                 waitSatisfaction[3],
355                 waitSatisfaction[4],
356                 waitSatisfaction[5],
357                 waitSatisfaction[6]
358             };
359
360             Row row = sheet.createRow(sheet.getLastRowNum()+1);
361
362             int cellIndex=0;
363             for(Object cellContent : rowContent){
364             if(cellContent instanceof Integer){
365                     row.createCell(cellIndex).setCellValue((int) cellContent);
366                 }
367             else if(cellContent instanceof Double){
368                     row.createCell(cellIndex).setCellValue((double) cellContent);
```

```
369                 }
370             else if(cellContent instanceof String){
371                     row.createCell(cellIndex).setCellValue(cellContent.toString());
372                 }
373             else if(cellContent instanceof Date){
374                     row.createCell(cellIndex).setCellValue((Date) cellContent);
375                 }
376             else if(cellContent instanceof Long){
377                     row.createCell(cellIndex).setCellValue((long) cellContent);
378                 }
379             cellIndex++;
380             }
381
382
383
384             //add the ArrayList<Object[]> singleBargeinfo to sheet 2 in the excel file.
385             Sheet sheet2 = workbook.getSheet("Barge info");
386
387             for(int i=0; i<singleBargeinfo.size(); i++){
388             Row row2 = sheet2.createRow(sheet2.getLastRowNum()+1);
389             int cellIndex2=0;
390                 for(Object cellContent : singleBargeinfo.get(i)){
391                 if(cellContent instanceof Integer){
392                         row2.createCell(cellIndex2).setCellValue((int) cellContent);
393                     }
394                 else if(cellContent instanceof Double){
395                         row2.createCell(cellIndex2).setCellValue((double) cellContent);
396                     }
397                 else if(cellContent instanceof String){
398                         row2.createCell(cellIndex2).setCellValue(cellContent.toString());
399                     }
400                 else if(cellContent instanceof Date){
401                         row2.createCell(cellIndex2).setCellValue((Date) cellContent);
402                     }
403                 cellIndex2++;
404                 }
405
406             }
407
408
409             //add the ArrayList<Object[]> events to sheet 3 in the excel file.
410             Sheet sheet3 = workbook.getSheet("Events");
411
412             for(int i=0; i<events.size(); i++){
413             Row row3 = sheet3.createRow(sheet3.getLastRowNum()+1);
414             int cellIndex3=0;
415                 for(Object cellContent : events.get(i)){
416                 if(cellContent instanceof Integer){
417                         row3.createCell(cellIndex3).setCellValue((int) cellContent);
418                     }
419                 else if(cellContent instanceof Double){
420                         row3.createCell(cellIndex3).setCellValue((double) cellContent);
421                     }
422                 else if(cellContent instanceof String){
423                         row3.createCell(cellIndex3).setCellValue(cellContent.toString());
424                     }
425                 else if(cellContent instanceof Date){
426                         row3.createCell(cellIndex3).setCellValue((Date) cellContent);
427                     }
428                 cellIndex3++;
429                 }
430
431             }
432
433             file.close();
434
435             FileOutputStream outFile =new FileOutputStream(new File(filePath));
436             workbook.write(outFile);
437             outFile.close();
438
439         } catch (FileNotFoundException e) {
440             e.printStackTrace();
441         } catch (IOException e) {
442             e.printStackTrace();
443         }
444
445     }
446
447
448     public void resetStats(){
449         //reset stats
450         this.bargesInPortafterWarmup=0;
451         this.bargesEnteredPort=0;
452         this.bargesLeftPort=0;
453         for(int i=0; i<5;i++){
454             this.descriptiveStatistics[i].clear();
455         }
456         this.singleBargeinfo.clear();
457         this.events.clear();
458     }
459
460     public void warmupReset(){
```

```
461         //register current barges in port
462         this.bargesInPortafterWarmup=this.bargesEnteredPort-this.bargesLeftPort;
463 
464         //reset stats
465         this.bargesEnteredPort=0;
466         this.bargesLeftPort=0;
467         for(int i=0; i<5;i++){
468             this.descriptiveStatistics[i].clear();
469         }
470         this.singleBargeinfo.clear();
471         this.events.clear();
472     }
473 }
474 
```

```java
package jbarges;

import java.util.ArrayList;

/**
 * Waiting profiles. Terminals provide barges information about
 * the maximum amount of time a barge has to wait until its processing
 * is started after it has arrived. This information is provided for
 * every possible arrival moment during a certain time horizon in the
 * form of a waiting profile.
 */
public class WaitingProfile {

    /**
     * Indexes of integer array: 0=startInterval, 1=startTime, 2=endTime, 3=insertionPoint
     */
    ArrayList<int[]> startIntervals;

    /**
     * Indexes of integer array: 0=Time, 1=Maximum waiting time, 2=Insertion Point
     */
    ArrayList<int[]> waitingProfile;

    int currentTime;

    Terminal terminal;

    /**
     * construct the waiting profile
     * @param terminal
     * @param barge
     * @param currentTime the arrival time in the port
     */
    public WaitingProfile(Terminal terminal, Barge barge, int currentTime){
        this.currentTime=currentTime;
        this.terminal=terminal;
        // there is no waiting at the port entrance, therefore we set all the values to 0
        if(terminal.toString().equals("t0")){
            waitingProfile = new ArrayList<int[]>();
            this.waitingProfile.add(new int[]{0,0,0});
            startIntervals = new ArrayList<int[]>();
            this.startIntervals.add(new int[]{0,0,0,0});
        }
        else{
            this.startIntervals(terminal,barge);
            this.waitingProfile();
        }
    }


    /**
     * Determine start intervals
     * @param terminal
     * @param barge
     * @return the start interval
     */
    public ArrayList<int[]> startIntervals(Terminal terminal, Barge barge){

        // every integer array in this list contains: startInterval, startTime, endTime
        startIntervals = new ArrayList<int[]>();

        for(int i=0; i<=terminal.appointments.size(); i++){
            // declare the values to compute for each interval
            int startInterval, startTime, endTime;

            // start interval and insertion point
            startInterval = i+1;

            // start time
            if(i==0){
                startTime = currentTime;
            }
            else{
                // start time is equal to the EDT of the last planned barge before insertion point i
                Barge key = (Barge) terminal.appointments.keySet().toArray()[i-1];
                int[] appointment = terminal.appointments.get(key);
                startTime = appointment[4]; //index 4 is EDT
            }

            // end time
            if(i==terminal.appointments.size()){
                endTime = Integer.MAX_VALUE; //used as infinity
            }
            else{
                // The end time of the start interval is equal to the
                // PST of the first planned barge after insertion point i,
                // minus the processing time of barge b. (They take LST in the example,
                // because they plan barges after i as late as possible)
                Barge key = (Barge) terminal.appointments.keySet().toArray()[i];
                int PST_i = terminal.appointments.get(key)[1];//index LST is 1
                int indexTerminal = barge.terminals.indexOf(terminal);
                int handlingTime = barge.handlingTimes.get(indexTerminal);
```

```
                endTime = PST_i - handlingTime;
            }

            //add to startInterval ArrayList
            //if feasible (?). i.e., startTime < endTime
            if(startTime<endTime && startTime>=this.currentTime){
                startIntervals.add(new int[]{startInterval, startTime, endTime});

            }

        }

        //check if intervals are disjoint
        for(int i=0; i<startIntervals.size()-1; i++){
            int end_i = startIntervals.get(i)[2];
            int start_i1 =  startIntervals.get(i+1)[1];
            if(end_i > start_i1){
                int[] startInterval = startIntervals.get(i);
                startInterval[2] = start_i1;
                startIntervals.set(i, startInterval);
                end_i = start_i1;
            }
        }

        return startIntervals;
    }

    /**
     * Slack is added to maximum waiting time (mwt). Slack is a parameter which can be configured in the repast GUI.
     * Furthermore, the slack method and slack denominator can also be configured in the GUI.
     * @param t arrival time
     * @return maximum waiting time
     */
    public int getMaxWaitingTime(int t){

        //search index of waiting profile to use
        for(int i=(waitingProfile.size()-1); i>=0; i--){

            if(t>=waitingProfile.get(i)[0]){

                int mwt;

                // mwt maximum waiting time + current time - arrival time
                int time = waitingProfile.get(i)[0];
                int maxWaitingTime = waitingProfile.get(i)[1];
                mwt = maxWaitingTime + time - t;

                if(mwt<0){
                    mwt=0;
                }
                if(Port.slackMethod.equals("Constant")){
                    return (mwt + Port.slack);
                }
                else if(Port.slackMethod.equals("Factor")){
                    //get the number of appointments in the terminal schedule
                    int numAppointments = terminal.appointments.size();
                    int slack = numAppointments / Port.slackDenominator * mwt;
                    return (mwt + slack);
                }
            }
        }
        return 0; // this will never be returned, see above
    }

    public ArrayList<int[]> waitingProfile(){
        // every integer array in this list contains: Time, Maximum waiting time
        waitingProfile = new ArrayList<int[]>();
        int time, maximumWaitingTime;

        for(int i=-1; i<startIntervals.size()-1; i++){
            if(i==-1){
                time=currentTime;
            }
            else{
                time=startIntervals.get(i)[2];
            }
            maximumWaitingTime=startIntervals.get(i+1)[1]-time;

            if(maximumWaitingTime>0){
                waitingProfile.add(new int[]{time, maximumWaitingTime});
            }
        }
        return waitingProfile;
    }

    /**
     *
     * @return String with the start intervals.
     */
    public String startIntervalsToString(){
        String s = "";
        for(int[] i: this.startIntervals){
```

```
185             for(int j:i){
186                 s+=j +" ";
187             }
188             s+="\n";
189         }
190 
191         return s;
192     }
193 
194     /**
195      * @return String with the waiting profile.
196      */
197     public String toString(){
198 
199         String s = "";
200         //add the table with the time, max waiting time
201         for(int[] i: this.waitingProfile){
202             for(int j:i){
203                 s += j +" ";
204             }
205             s += "\n";
206         }
207         return s;
208     }
209 }
210 
```

```java
package tdtsp;

import java.util.ArrayList;
import java.util.Collections;

import jbarges.Barge;

/**
 * An implementation of the TDTSP. This object is used by a barge to compute
 * the best rotation and it also stores information about the rotation.
 * It is based on the algorithm developed by Malandraki and Dial (1996).
 * The implementation uses a recursive algorithm / backtracking technique
 * and is partly based on an implementation of the eight queens problem as shown
 * in "Java for Everyone, 2e by Cay Horstmann.
 */
public class TDTSP{

    /**
     * The sailing times. This is a smaller version of the sailing times table in the simulation class. It
     * only contains the terminals that this barge has to visit.
     */
    public int[][] sailingTimes;

    /**
     * The barge for which the TDTSP is solved.
     */
    Barge barge;

    /**
     * Number of nodes in the graph, excluding the depot (support up to 27 nodes excluding depot, can easily be extended if neces
     */
    public int numNodes;

    /**
     * The best route is the route with the lowest sojourn time. If there is more than one route, then it is the route with the l
     */
    public String bestRoute;

    /**
     * The best leave time is the time in of leaving the port. The best sojourn time is
     * the time of leaving the port minus the time of arrival at the port.
     */
    public int bestLeaveTime, bestSojournTime;

    /**
     * The number of 0's is the number of nodes excluding the depot node 0, plus 5 zeros to store the last node visited.
     * This is created by the method initial()
     */
    String sInitial;

    int startTime;

    //the current stage and the previous stage is retained.
    Stage currentStage;

    //Accepted tours with corresponding cost
    ArrayList<String> acceptedTour;
    ArrayList<Integer> acceptedCost;

    public TDTSP(int startTime, Barge barge){

        this.barge = barge;
        this.startTime = startTime;
        this.sailingTimes = barge.sailingTimes;
        this.numNodes = sailingTimes.length-1;
        this.sInitial= this.initial(numNodes);

        this.acceptedTour = new ArrayList<String>();
        this.acceptedCost = new ArrayList<Integer>();
        this.currentStage = new Stage();

        PartialSolution start = new PartialSolution(sInitial,sInitial, this);
        this.solve(start,this.startTime, "");

        this.bestRoute();

        this.currentStage.cost.clear();
        this.currentStage.pred.clear();
        this.currentStage.set.clear();
    }

    public void solve(PartialSolution sol, int time, String pred){



        //compute cost to cost (cost = arrival time)
        int tdtt = sol.timeDependentTravelTime(time);
        time += tdtt;

        this.currentStage.addCost(time);
        //add to set
        this.currentStage.addSet(sol.currentSet);
```

```
93          //add to route
94          pred = pred +" "+ sol.getLastNodeVisited();
95          this.currentStage.addPred(pred);
96
97          int exam = sol.examine();
98          if (exam == PartialSolution.ACCEPT){
99
100             //return to the depot
101             int lastNode = sol.getLastNodeVisited();
102             time+=sol.computeCost(lastNode, 0);
103             pred = pred +" "+ 0;
104
105             //add solution to list
106             this.acceptedTour.add(pred);
107             this.acceptedCost.add(time);
108
109         }
110         else if (exam == PartialSolution.CONTINUE){
111             for(PartialSolution p : sol.extend()){
112                 solve(p, time, pred);
113             }
114         }
115     }
116
117     /**
118      * Initialize the first set binary string.
119      * Number of 0's is the number of nodes excluding the depot node 0, plus 5 zeros to store the last node visited.
120      * Nodes visited and last node node visited (does not include depot), also known as set S and node k.
121      * @param numNodes
122      * @return
123      */
124     public String initial(int numNodes){
125         String s= "00000";
126         for(int i = 0; i<numNodes; i++){
127             s+="0";
128         }
129         return s;
130     }
131
132     /**
133      * Set the best route to the route with the least cost from the accepted tours list.
134      */
135     public void bestRoute(){
136         int index = this.minimizeSailingtime();
137         this.bestRoute = this.acceptedTour.get(index);
138         this.bestLeaveTime = this.acceptedCost.get(index);
139         this.bestSojournTime = this.bestLeaveTime - this.startTime;
140     }
141
142     /**
143      * This method converts the best route to a printable String for the console
144      * @return a string with the best route and associated cost
145      */
146     public String bestRouteToString(){
147         String best = this.bestRoute;
148         String converted = "";
149         for(int i=0; i<best.length(); i++){
150             String c = ""+best.charAt(i);
151             if(c.equals(" ")){
152                 converted += " ";
153             }
154             else{
155                 int ter = Integer.parseInt(c);
156                 converted += barge.terminals.get(ter).toString();
157             }
158         }
159         return converted;
160     }
161
162     /**
163      * Puts all the accepted tours with associated cost in a String that is made for the console
164      * @return
165      */
166     public String getAccepted(){
167
168         String acceptedTours = "";
169
170         for(int i=0; i<this.acceptedTour.size(); i++){
171             //get tour at index i
172             String tour = acceptedTour.get(i);
173
174             //convert tour
175             String convTour = "";
176             for(int j=0; j<tour.length(); j++){
177                 String c = ""+tour.charAt(j);
178                 if(c.equals(" ")){
179                     convTour += " ";
180                 }
181                 else{
182                     int ter = Integer.parseInt(c);
183                     convTour += barge.terminals.get(ter).toString();
184                 }
```

```
            }

            //get associated costs
            int cost = this.acceptedCost.get(i);

            //add to the string
            acceptedTours += "Tour: " + convTour + "\t Cost: " + cost + "\n";
        }

        return acceptedTours;

    }

    /**
     * If there are more than 1 best tour this method will select the tour
     * with the least sailing time. A solution with less sailing time is
     * preferred because it will cost less fuel cost.
     * @return the index in acceptedTours of the tour with the least sailingTime
     *
     */
    public int minimizeSailingtime(){
        //a list to store all indexes with the lowest costs
        ArrayList<Integer> indexes = new ArrayList<Integer>();

        // find the lowest cost
        int minCost = Collections.min(this.acceptedCost);
        int indexBestTour = this.acceptedCost.indexOf(minCost);

        // find all tours with that cost
        for(int i=0; i<this.acceptedCost.size(); i++){
            //check if tours has the same cost as minCost
            if(this.acceptedCost.get(i) == minCost){
                //add the index
                indexes.add(i);
            }
        }

        //check if there is more than 1 best tours
        if(indexes.size()>1){
            //a list to save the sailing times
            ArrayList<Integer> sTimes = new ArrayList<Integer>(indexes.size());

            //compute the sailing times for each of these tours
            for(int i : indexes){
                //get the tour and compute its sailing time
                String tour = this.acceptedTour.get(i);
                int sailingTime = this.computeSailingtime(tour);
                //add the sailing to the list
                sTimes.add(sailingTime);
            }

            // index of the minimum sailing time. if there are more than 1 than the
            // first occurrence in the list is sufficient.
            int minSailingTime = Collections.min(sTimes);
            int indexSTimes = sTimes.indexOf(minSailingTime);
            indexBestTour = indexes.get(indexSTimes);
        }
        return indexBestTour;
    }

    /**
     * Compute the total sailing time of a specific tour. This method is used
     * in the minSailingTime() method.
     * @param route a String representation of the tour. e.g., "0 1 5 9 0"
     * @return the sailing time of a tour
     *
     */
    public int computeSailingtime(String route){
        int sailingTime=0;

        // iterate over the String
        // -2 because the last destination node is already reached at tour.length()-2
        for(int i=0; i<route.length()-2; i++){
            String c = ""+route.charAt(i);
            if(!c.equals(" ")){
                int origin = Integer.parseInt(c);
                //the destination node is 2 indexes further in String tour
                String d = ""+route.charAt(i+2);
                int destin = Integer.parseInt(d);

                //add sailing time between this node and the next node
                sailingTime+=this.sailingTimes[origin][destin];
            }
        }
        return sailingTime;
    }
}
```

```java
package tdtsp;

import java.util.ArrayList;

/**
 * Stage of the TDTSP.
 */
public class Stage {

    // arrival times at last node
    ArrayList<Integer> cost;

    // S and k are stored in parallel in an array
    ArrayList<String> set;

    //predecessor nodes
    ArrayList<String> pred;

    public Stage() {
        this.cost = new ArrayList<Integer>();
        this.set = new ArrayList<String>();
        this.pred = new ArrayList<String>();
    }

    public Stage(Stage stage){
        this.cost = new ArrayList<Integer>();
        this.set = new ArrayList<String>();
        this.pred = new ArrayList<String>();


        for(int c : stage.cost){
            this.cost.add(c);
        }

        for(String s : stage.set){
            this.set.add(s);
        }

        for(String p : stage.pred){
            this.pred.add(p);
        }
    }

    public void addCost(int cost){
        this.cost.add(cost);
    }

    public void addSet(String set){
        this.set.add(set);
    }

    public void addPred(String pred){
        this.pred.add(pred);
    }
}

```

```java
package tdtsp;

import java.util.ArrayList;

import jbarges.Terminal;

/**
 * Partial solution of TDTSP.
 */
public class PartialSolution{

    public String currentSet, previousSet;

    public static final int ACCEPT = 1;
    public static final int ABANDON = 2;
    public static final int CONTINUE = 3;

    TDTSP tdtsp;

    /**
     * Constructor of PartialSolution
     * @param currentSet    example 000000110001011
     * @param previousSet   example 000000110001101
     * @param tdtsp the     TDTSP object
     */
    public PartialSolution(String currentSet, String previousSet, TDTSP tdtsp){
        this.currentSet = currentSet;
        this.previousSet = previousSet;
        this.tdtsp=tdtsp;
    }

    /**
     * add node
     * @return
     */
    public ArrayList<PartialSolution> extend(){
        ArrayList<PartialSolution> set = new ArrayList<PartialSolution>();
        for(int i = tdtsp.numNodes-1; i>=0; i--){
            String node = ""+currentSet.charAt(i);
            String newNode = "";
            if(node.equals("0")){

                //set unvisited node from 0 to 1
                String newSet = currentSet.substring(0,i) + '1' + currentSet.substring(i+1, tdtsp.numNodes) ;

                //set the last visited to the new node visited and get the corresponding 5 bit notation
                newNode = Integer.toBinaryString(tdtsp.numNodes-i);
                String zeros="";
                for(int u=0; u< 5-newNode.length(); u++){
                    zeros+="0";
                }
                newNode=zeros+newNode;

                //add the last node visited to the new set
                newSet+=newNode;

                //create the partial solution
                PartialSolution newSol = new PartialSolution(newSet, currentSet, tdtsp);

                //add the partial solution to set
                set.add(newSol);
            }
        }
        return set;
    }

    /**
     * examine whether to accept, or continue
     * @return
     */
    public int examine(){
        if(this.getNumVisited() == tdtsp.numNodes){
            return ACCEPT;
        }
        else{
            return CONTINUE;
        }
    }


    /**
     * get the time dependent travel time between i (origin) and j (destination)
     * as a function of the departure time from the origin node i of the link.
     * @param departureTime
     * @return
     */
    public int timeDependentTravelTime(int departureTime){

        int originIndex = getLastNodeVisited(previousSet);
        int destinIndex = getLastNodeVisited(currentSet);

        Terminal destinTerminal = tdtsp.barge.terminals.get(destinIndex);
```

```
 93
 94         int sailingTime = tdtsp.sailingTimes[originIndex][destinIndex];
 95         int waitingTime = tdtsp.barge.waitingProfiles.get(destinTerminal).getMaxWaitingTime(departureTime+sailingTime);
 96         int handlingTime = tdtsp.barge.handlingTimes.get(destinIndex);
 97
 98         int tdtt = sailingTime + waitingTime + handlingTime;
 99
100         return tdtt;
101     }
102
103     /**
104      * get last node from the current set
105      * @return
106      */
107     public int getLastNodeVisited(){
108         return getLastNodeVisited(currentSet);
109     }
110
111     /**
112      * get last node from binary string s
113      * @param s example 000000110001011 (the last five digits contains info about the last visited node)
114      * @return the index of the the terminal in barge.terminals
115      */
116     public int getLastNodeVisited(String s){
117         String last="";
118         for(int i = tdtsp.numNodes; i<tdtsp.numNodes+5; i++){
119             last=last+s.charAt(i);
120         }
121         int b = Integer.parseInt(last,2);
122         return b;
123     }
124
125     /**
126      * get number of visited nodes
127      * @return the number of visited nodes
128      */
129     public int getNumVisited(){
130         int numVisited = 0;
131         for(int i = tdtsp.numNodes-1; i>=0; i--){
132             String node = ""+currentSet.charAt(i);
133             if(node.equals("1")){
134                 numVisited++;
135             }
136         }
137         return numVisited;
138     }
139
140     /**
141      * get the cost of adding a node. si is the origin set, sj is the destination set.
142      * note: only used to return to exit point
143      * @return
144      */
145     public int computeCost(){
146         int origin = getLastNodeVisited(previousSet);
147         int destination = getLastNodeVisited(currentSet);;
148         return tdtsp.sailingTimes[origin][destination];
149     }
150
151     /**
152      * get the cost of adding a node.
153      * still used for getting the cost of the return to the depot, which is not time dependent
154      * @param origin       example 000000110001011
155      * @param destination  example 000000110001011
156      * @return
157      */
158     public int computeCost(int origin, int destination){
159         return tdtsp.sailingTimes[origin][destination];
160     }
161
162 }
163
```