University of Stavanger

Faculty of Science and Technology

MASTER'S THESIS

Study program/ Specialization: Industrial Economics/ Project Management	Autumn semester, 2016 Open access
Writer: Håkon Hapnes Strand	……………………………………… (Writer's signature)
Faculty supervisor: Tomasz Wiktorski External supervisor(s): Rolf Thu	
Thesis title: Open shop scheduling in a manufacturing company using machine learning	
Credits (ECTS): 30	
Key words: Open shop scheduling, machine learning, reinforcement learning, metaheuristics, ant colony optimization, genetic algorithms, discrete-event simulation, manufacturing	Pages: 100 + enclosure: 10 Stavanger, December 12, 2016

Truth is much too complicated

to allow anything but

approximations.

John von Neumann (1903-1957)

Abstract

Scheduling jobs in a manufacturing company that delivers custom products is challenging. Aarbakke is a company that manufactures advanced assemblies for the oil and gas industry. Its existing resource planning tool frequently produces unrealistic job schedules, leading to substantial delays. In this thesis, we describe a reinforcement learning agent that optimizes scheduling by utilizing historical data. Our aim is to minimize the time spent processing jobs past their deadlines - the tardiness. The problem can be modelled as an open shop scheduling problem. Existing research in this area has only looked at other performance measures, such as makespan. Due to the NP-hardness of the problem, we used ant colony optimization and genetic algorithms to produce heuristic scheduling algorithms that can make efficient decisions under uncertainty. Based on cross-validation of 18 algorithms, a candidate model was selected and tested in a hypothesis test. It reduced the mean tardiness by 2.6 % compared to the scheduling algorithm currently in use when testing on historical processing times and by 14.6 % when testing on sets of forecasted processing times. Implementing it in production can potentially lead to savings in manufacturing cost. The approach can be applied to similar problems in other custom job shops.

Preface

At the start of 2016, I had been juggling with the idea of using machine learning in an industrial setting for my master's thesis for some time, when I became aware of a project my employer, F5 IT, had initiated with manufacturing company Aarbakke.

I knew this was a unique opportunity to test my ideas in a real-world environment. Not only did Aarbakke have a large database of information collected from the entire domain of their business. Its management also had great ambitions about how to use it. There had been little in terms of progress at the time, and I took it upon myself to rejuvenate the project. After a successful proof of concept phase, we landed on a problem statement that was aligned with the company's strategy as well as my own interests.

The subsequent work has taken me on an interesting journey of data analysis, research and software development which has culminated in this thesis.

Håkon Hapnes Strand

Stavanger, December 2016

Acknowledgements

I would like to thank my supervisor Rolf Thu, Kjartan Gausland and the good people at Aarbakke for letting a master's student tackle some of the most interesting problems of their business. Rolf and Kjartan have dedicated a significant portion of their valuable time to supporting the project over a period of more than half a year. Without their efforts, this thesis would not have been possible.

I would like to thank my co-supervisor, Tomasz Wiktorski, at the Department of Electrical Engineering and Computer Science, for supervising the work of a student from another department and for pushing me to improve the quality of my work.

I would like to thank my manager at F5 IT, Jo Arild Tønnessen, for giving me the freedom to combine academic pursuits with my daily work as a software engineer. In addition, I thank Lene Andrea Knutsen and Kåre Plahte, who helped putting me in contact with the right people at Aarbakke.

Last, but not least, I thank my girlfriend, Camilla Madsen, for her love and support.

Contents

List of Figures

List of Tables

Abbreviations

ACO	Ant Colony Optimization
BM	Benchmark
CSV	Comma-Separated Values
DES	Discrete-Event Simulation
EDD	Earliest Due Date
ERP	Enterprise Resource Planning
GA	Genetic Algorithm
GB	Gigabyte
JIT	Just-In-Time manufacturing
JSP	Job Shop Problem/Job Shop Scheduling
LDD	Latest Due Date
LNC	Largest Number of Children
LNO	Largest Number of Operations
LPT	Longest Processing Time
MDP	Markov Decision Process
NOK	Norwegian kroner
OS	Operating System
OSSP	Open Shop Scheduling Problem
PC	Personal Computer
RAM	Random Access Memory
SNC	Smallest Number of Children
SNO	Smallest Number of Operations
SPT	Shortest Processing Time
SQL	Structured Query Language
SWB	Scheduling Workbench
TSP	Travelling Salesman Problem
WO	Work Order

Chapter 1

Introduction

1.1 Background

Aarbakke is a Norwegian manufacturing company that produces advanced equipment and assemblies for the oil and gas industry. To stay competitive in a toughening market, the company has implemented an ERP system that controls and monitors every operation in its manufacturing process.

An integral part of the ERP system is a scheduling tool that we will refer to as the *Scheduling Workbench* (SWB). Throughout the company, all operations are performed according to schedules that this tool outputs daily.

The operations are part of work orders that are inserted into the schedules based on a deterministic dispatch algorithm. Although the system logs a large amount of trace information to a database, the scheduling algorithm does not take this historical knowledge into account. Nor does it consider the stochastic nature of processing times. As a consequence, the company frequently experiences a substantial disparity between scheduled activities and actual deliveries. Adding to the difficulty, Aarbakke's work orders don't follow a regular pattern. Due to the customized nature of the equipment the company delivers, orders are often very different.

To improve upon the existing scheduling framework, this research suggests an alternative approach which utilises machine learning and computational statistics to discover and exploit patterns contained in the historical data.

1.2 Scope

The primary aim of the research is to reduce the mean tardiness in the manufacturing environment at Aarbakke by improving the scheduling algorithm used in the Scheduling Workbench.

In order to achieve that, we use machine learning techniques and computational statistics to infer optimal scheduling strategies. More specifically, strategies are learned using implementations of two families of metaheuristic search algorithms, ant colony optimization and genetic algorithms, in combination with reinforcement learning.

It is also paramount to the research to validate and test the results using statistically rigorous evaluation criteria.

1.3 Significance

The alternative scheduling approach presented in this thesis can reduce the tardiness at Aarbakke's machine shop with a noticeable percentage, potentially leading to significant savings in manufacturing cost.

The proposed framework can be implemented by integrating the software developed during this research with the already existing Scheduling Workbench, with minimal need for adjustments.

While this research only looks for scheduling strategies that generalize well to the manufacturing environment at Aarbakke specifically, the general approach could be applied to manufacturing environments of a similar type, where the scheduling process can be modelled as an open shop scheduling problem.

Much of the existing literature on job scheduling, some of which is reviewed in chapter 2, only consider job shops with regular job patterns. Scheduling in a custom job shop is significantly more difficult (Lang, 2011).

Open shop scheduling is given less attention than other scheduling problems in literature (Naderi, Ghomi, Aminnayeri & Zandieh, 2010). As we shall see in chapter 2, evaluating the performance of metaheuristic algorithms in minimizing tardiness in open shop scheduling is an open research problem.

1.4 Problem statement

How can Aarbakke utilize historical data to improve the scheduling algorithm in its scheduling tool?

1.5 Assumptions

Throughout the thesis, we assume that the reader is familiar with the basic theory of statistics and elementary computer science. The definitions in 1.8 should provide sufficient information for readers with a scientific background, but explanations of the most rudimentary concepts are omitted. The algorithms are either presented in an informal pseudocode that imitates the Python programming language that was used extensively throughout the research, or in actual Python code. Python has a comparatively simple syntax that is sometimes called "executable pseudocode" (Krol, 2014). Little or no prior knowledge of computer science should be required to grasp the logic of the algorithms.

It is assumed that the information contained in the database provides an accurate account of historical events and that all work has been logged correctly. [1]

It is assumed that operations that are planned with zero duration don't act as constraints.

[1]An overview of the database is given in appendix B.

1.6 Limitations

The time series data that the research is based on dates back to December 12, 2015, when Aarbakke started logging operational information in a database. The entire data set spans approximately one year at the time of thesis submission. Hence, there could be long-term trends that analysis of the current data will not uncover. The calendar year of 2016 has been one of great turmoil in the Norwegian oil and gas industry (NTB, 2016). The activity level at the company this year does not necessarily reflect what has been normal in previous time periods.

1.7 Delimitations

Aarbakke's database contains more than 20 GB of tabular data spread across 23 tables, most of which has a relevance to the problem statement. Some of the data has been excluded from the scope so that a practical solution to the problem can be implemented within the given time frame, leading to the following delimitations.

- While the research has modelled statistical distributions of processing times connected to machines and resources, it would be possible to use the same approach on other entities that are linked to work orders and operations, such as employees, customers and machine parts. The research has been purposely limited to machines and resources.

- Operations are constrained by the availability of parts. The database keeps track of the inventory, but this information is not considered in the simulation models.

- The machines are modelled as ideal machines that never experience downtime due to failure.

- The arrivals of new work orders are stochastic by nature and affect future schedules. When simulating the outcome of a schedule, future orders are not considered.

- On some occasions, planned operations are moved from one machine to another. This is not modelled in simulations.

- Schedules are exported from the Scheduling Workbench and effectuated on a daily basis. When simulating the outcome of a schedule, the fact that the schedule is subject to daily change is ignored. To incorporate future schedules in the model, there would have to be simulations within simulations, vastly increasing the computational complexity.

- All statistical models are defined according to the principles of frequentist probability. Aarbakke has several senior employees with decades of experience within the company, whose domain knowledge could be exploited to construct Bayesian probability models.

- External factors that impact the activities at Aarbakke could be used as inputs to the machine learning algorithm, but have been excluded from the scope.

1.8 Definitions

Given the interdisciplinary nature of the thesis, this section provides definitions of concepts specific to computer science and computational statistics.

Antifragility: Antifragile systems have the property of improving when facing risk and uncertainty, as opposed to resilient systems which simply resist shock (Taleb, 2012, p. 2).

Big O notation: A way of describing the asymptotic upper bound running time of an algorithm as a function of its input size n, often by inspecting the algorithm's overall structure (Cormen, Leiserson, Rivest and Stein, 2002, pp. 41-44).

Cross-validation: A model-validation technique for assessing how the results of a statistical analysis will generalize to an independent data set ("Cross-validation", n.d.).

Discrete-event simulation: Modelling of systems in which the state variables change only at discrete points in time (Banks, Carson, Nelseon and Nicol, 2009, p. 12). In discrete-event simulation, an artificial history of a system is generated based on model assumptions and the performance of the system is estimated using numerical methods.

Dispatch rule: A simple, rule-based algorithm that dispatches jobs in order by evaluating a single metric.

Feature: A term commonly used in machine learning for input variables to the machine learning algorithm.

Forward chaining: An alternative approach to cross-validation for time-series data. As with k-fold cross-validation, the training subset of the dataset is split into folds, but the model is always tested on data that succeeds the training data in time (Williams, 2011).

Hyperparameter: Hyperparameters (in the context of machine learning) are parameters that define higher level concepts about models and cannot be learned directly from the data (Amatriain, 2016).

Kernel density estimation: An estimation of the probability density function of a variable, using sums of symmetrical probability densities. The estimation is defined by the integral $\int_{-\infty}^{\infty} K(t)dt = 1$, where $K(t)$ is a kernel function (Rizzo, 2008). In this research, we use the standard normal density, or the Gaussian kernel, as the kernel function.

K-fold cross-validation: A machine learning technique in which the model training subset of the dataset is split into *k* folds and the model is successively trained on *k-1* folds and tested on the remaining fold. The process is repeated *k* times so that every fold is used for testing exactly once (Raschka, 2015, p. 175).

Lazy evaluation: Evaluation of a programming expressing only when it is needed, to avoid repeated evaluations (”Lazy evaluation”, n.d.).

Machine learning: The science of getting computers to act without being explicitly programmed (Ng, 2011). Also known as *data mining*.

Machine learning model: A mathematical function inferred from data using a machine learning algorithm (Amatriain, 2016).

Markov decision process: A discrete time stochastic control process that is partly random and partly under the control of a decision maker (”Markov decision process”, n.d.).

Metaheuristics: A high-level problem-independent algorithmic framework that provides a set of guidelines or strategies to develop heuristic optimization algorithms (Sörensen & Glover, 2013).

NP-complete: A decision problem is said to be NP-complete if it cannot be solved in polynomial-time. That is, no solution with a time complexity of, using big-O notation, $O(n^k)$ has ever been found for an NP-complete problem (Cormen et al., 2002).

NP-hard: The hardest complexity class in computational complexity theory. An NP-hard problem is at least as hard as an NP-complete problem.

Overfitting: When a model captures the training data well, but fails to generalize well to unseen data (Raschka, 2015), often as a result of prolonging the training experiment longer than necessary or because of biases in the validation and testing phases of machine learning.

Pseudorandom: Computer-generated numbers that seem random but aren't in a strict sense ("Can a computer generate a truly random number?", 2011). A constant number, known as a random seed, can be used to reconstruct the same sequence of pseudorandom numbers.

Reinforcement learning: A type of machine learning where a software agent improves its performance by interacting with an environment and receiving rewards for favourable outcomes. (Raschka, 2015, p. 6)

Search space: In this context, the set of possible solutions to a decision problem.

Tardiness: A measure of delay that ignores early completions ("Tardiness (scheduling)", n.d.).

1.9 Summary

In this chapter, we have defined the problem statement and scope of the research. Important limitations, delimitations and assumptions that was made during the research were highlighted. The reader has been provided with a background explaining the underlying motivation for undertaking the research as well as definitions of key concepts used throughout the thesis.

Chapter 2

Literature review

2.1 Scheduling

In job scheduling, the optimization problem is defined by the structure of the work process. We define any work process as a set of *jobs* to be performed on *machines* using a number of *operations*. Each operation can only be performed on one machine, while the machines could have specific or generic capabilities depending on the problem. A machine does not have to be a piece of physical equipment. It could also be an abstraction of a group of employees or other entities that can execute work. The term *resource* is used interchangeably, but in our case most of the resources are physical machines.

In the operations research literature on scheduling problems, different terms refer to similar problems with small alterations. To avoid confusion, we reproduce the definitions described in Anand and Panneerselvam (2015):

- **Single machine scheduling**: The problem consists of arranging a number of jobs to be performed on one machine that can handle one job at a time.
- **Flow shop scheduling**: The problem consists of scheduling n jobs on m machines, each with m operations to be performed in the same order.
- **Job shop scheduling**: The problem consists of scheduling n jobs on m machines, each with m operations to be performed in varying order.
- **Open shop scheduling**: The problem consists of scheduling n jobs on m machines, each with 1, 2,...,m operations to be performed in varying order.

2.2 Open shop scheduling

In this research, we are dealing with a set of jobs that may require processing on any number of machines from 1 to m. The order of processing can vary freely. According to the definitions in the previous section, we can model this as an open shop scheduling problem (OSSP).

Anand and Panneerselvam (2015) provide a comprehensive study of the current state of open shop scheduling problems. Their review of the literature group the different approaches by their optimization objectives:

- Minimizing the makespan.

- Minimizing the sum of completion time of all the jobs.
- Minimizing the weighted sum of completion time of all the jobs.
- Minimizing the total tardiness of all jobs.
- Minimizing the weighted total tardiness of all jobs.
- Minimizing the number of tardy jobs.
- Minimizing the number of weighted sum of tardy jobs.
- Minimizing the maximum lateness.

The underlying goal of scheduling optimization is roughly the same regardless of objective function. The aim is to execute as much work as possible in as little time as possible, maximizing efficiency.

As stated in section 1.2, the specific aim of this research is to minimize mean tardiness as defined in 1.8 and 3.3. Very little research has been carried out to minimize total tardiness [1] of all jobs in the open shop scheduling problem (Anand & Panneerselvam, 2015, p. 45). The mentioned paper suggests development of heuristic, metaheuristic and hybrid algorithms for this problem. It is in this exact direction our research turns its attention.

2.3 Time complexity

Determining the asymptotic running time, or time complexity, of scheduling problems has long been a research topic of interest in computer science. The

[1] Section 3.3 explains why we use mean tardiness as the objective function instead of total tardiness and why it only makes a minor difference.

Travelling Salesman Problem (TSP) is a well-known special case of the Job Shop Problem (JSP) that was implicitly proven to be NP-hard in the early 1970s (Karp, 1972). By extension, JSP is also NP-hard, given that TSP can be considered a simplification of JSP with $m = 1$.

The time complexity of the Open Shop Scheduling Problem (OSSP) has also been devoted considerable study. The two-machine OSSP with different resources has been shown to be NP-hard when the objective is minimizing the makespan, that is, the total time elapsed from the beginning of the first job to the end of the last (Gonzalez & Sahni, 1976). Other, similar results are listed in Anand and Panneerselvam (2015) for various objective functions.

Considering the NP-hardness of general open shop scheduling problems, this research focuses on heuristics, algorithms that find approximations rather than exact solutions.

2.4 Metaheuristics

A heuristic algorithm is characterized by its aim of finding a solution that is ”good enough”, when finding the global optimum through exact means would be impractical. Metaheuristics is a class of algorithms that generate heuristic algorithms.

Formally, metaheuristics is a high-level problem-independent algorithmic framework that provides a set of guidelines or strategies to develop heuristic optimization algorithms (Sörensen & Glover, 2013). In practice, the term is also used to refer to a problem-specific implementation of a heuristic optimization algorithm according to the guidelines expressed in the metaheuristic framework (Bolufé-Röhler, 2014).

Yang (2011) provides an overview of the most commonly used metaheuristic algorithms. Many of them are inspired by phenomena occurring in nature, leading to a subset of metaheuristics called *nature-inspired algorithms*. This research focuses on two classes of nature-inspired algorithms that have been proven successful on a wide variety of optimization problems through decades of research.

2.4.1 Ant colony optimization

Ant colony optimization was introduced in the PhD thesis of Marco Dorigo (Dorigo, 1992). It is inspired by the foraging behaviour of social ants. Ants communicate by excreting a chemical substance called pheromones. Ants are attracted to its scent, so that an ant is more likely to follow a trail with a high concentration of pheromones. Since pheromones gradually evaporate over time, ants will tend to prefer the shortest route from the ant heap to the food source. More and more ants will eventually follow the same trail, and the colony will discover the shortest route from A to B. The behaviour of the colony is a positive feedback mechanism, where the colony converges to a self-organized state, which is the defining characteristic of ant algorithms (Yang, 2011).

Ant colony optimization can be used both for discrete and continuous optimization problems (Heaton, 2014). In the discrete version, the goal is to traverse a graph, a structure of connected nodes, by travelling the shortest distance. The "distance" in this case is the objective function, the total tardiness in the machine shop.

The pheromones are added to the edges between nodes, so that the probability of an ant moving from node i to node j is

$$P(i,j) = \frac{\tau_{ij}}{\sum_{k=1}^{m} \tau_k}, \tag{2.1}$$

where τ_{ij} is the amount of pheromones at edge ij, τ_k is the amount of pheromones at edge k and m is the total number of edges.

The implementation in this research is based on Heaton's approach. In our case, each node in the graph corresponds to a work order, and the path through the graph is the order in which work orders are scheduled. Pheromones are deposited and evaporated after each iteration when all the ants in the colony have traversed the graph, thus reinforcing the trail. Several update rules are possible (Blum, 2005):

- Elitist Ant System
- Rank-Based Ant System (ASRank)
- Max-Min Ant System (MMAS)
- Ant Colony System (ACS)
- Hyper-Cube Framework (HCF)

This research implements variations of the Elitist and ASRank systems.

2.4.2 Genetic algorithms

Genetic algorithms are inspired by the evolutionary process of natural selection and were first devised by John Holland in the 1960s and 1970s (Yang, 2011). The following overview is based on Heaton (2014).

In genetic algorithms, a population of candidate solutions evolve through a series of *generations*. The members of the population are called *phenotypes*, while their actual solutions are called *genotypes*. The genotype is an array of numbers representing the model's weights for each input variable. These resemble the chromosomes in the DNA of living organisms. Hence, the name *genetic* algorithms. The output of the model is the scalar product of the input variables and the genotype. Let $\mathbf{x}$ be the input vector, $\mathbf{w}$ be the genotype and n be the size of each vector. Then the output, y, is calculated as:

$$y = \sum_{i=1}^{n} x_i w_i \tag{2.2}$$

In our implementation, the inputs are a series of dispatch rules further described in chapter 5.

After each generation, the population evolves through natural selection. The fittest phenotypes, according to some fitness function, breed new phenotypes, while the least fit phenotypes are removed from the population. There are three main selection methods:

- **Truncation selection**: The population is sorted by fitness and the fittest proportion p of the population is selected and reproduced $1/p$

times. The rest of the phenotypes are removed from the population.

- **Fitness proportionate selection**: Each phenotype is selected with a probability proportionate to its fitness.

- **Tournament selection**: A number of phenotypes, k, are selected to participate in a tournament. Two and two phenotypes are paired up in knockout rounds, where the fittest advance to the next round. The tournament ends when only one winner is left.

Tournament selection has several advantages over the other methods. Most importantly, it bypasses the concept of generations. For each tournament, only its participants need to be scored, instead of the entire population. This becomes very convenient when the scoring procedure is computationally expensive, as is the case in this research. Thus, we will be using tournament selection exclusively.

Once phenotypes have been selected for breeding, new members of the population are generated by utilizing two genetic operators, *crossovers* and *mutation*. These serve to balance exploitation and exploration of the search space, much like in real-world biological evolution. Crossover works by taking parts of the genotype of each parent solution when creating a new candidate solution, exploiting the fit traits of the selected parents. Mutation works by slightly altering parts of the parent genotype that is transferred to the child genotype. This ensures that the population does not get stuck in a local optimum by forcing exploration.

As with ant colony optimization, genetic algorithms can be applied to both discrete and continuous problem. In this research, we use a continuous implementation to learn a set of composite dispatch rules. This approach has been used successfully in job shop scheduling (Shahzad & Mebarki, 2010).

2.5 Reinforcement learning

Machine learning is the science of getting computers to act without being explicitly programmed (Ng, 2011). This field of computer science can further be divided into three broad categories, depending on the problem type: Supervised learning, unsupervised learning and reinforcement learning (Raschka, 2015). Supervised learning deals with labelled data, when certain inputs are established to correspond with known outputs. Unsupervised learning discovers hidden structure in unlabelled data. In the problem at hand, our aim is to devise an algorithm that learns optimal scheduling strategies when outcomes are highly uncertain. One of the main methods to tackle such problems is reinforcement learning (Kochenderfer, 2015, pp. 4-5).

Sutton and Barto (2012) define reinforcement learning as learning how to map situations to actions, so as to maximize a numerical reward signal. In mathematical optimization, this reward signal is referred to as the *objective function.* In recent literature on machine learning, the term *cost function* - or *loss function* - is used if the objective is minimization, and *fitness function* if the objective is maximization. A more informal definition focusing on computational application is given by Raschka (2015, p. 6), who describes reinforcement learning as a type of machine learning where a software agent improves its performance by interacting with an environment and receiving rewards for favourable outcomes.

The process in which scheduling takes place is essentially a Markov decision process, a discrete-time stochastic control process that is partly random and partly under the control of a decision-maker, in this case the Scheduling Workbench. Reinforcement learning can solve Markov decision processes without explicit specification of transition probabilities. Instead, the transition probabilities are accessed through a simulator ("Markov decision process", n.d.). A more extensive discussion of the theory behind this can be found in Sutton and Barto (2012, pp. 58-73). This research implements a reinforcement learning agent in a discrete-event simulation environment.

2.6 Discrete-event simulation

Discrete-event simulation (DES) models systems in which the state variables change at discrete points in time (Banks et al., 2009, p. 12). This is a flexible approach with several advantages (Sharma, 2015), some of which are:

- DES allows the study of complex systems.
- DES enables feasibility testing of any hypothesis regarding the system.
- Designing the simulation model might help in increasing knowledge about the system.

- DES aids the formulation and verification of analytical solutions.

A discrete-event simulator consists of the following components:

- **System state**: A set of variables that capture the properties of the system.
- **Simulation clock**: An internal integer counter that keeps track of simulation time.
- **Pending events**: A list of events that are going to be simulated at specified simulation times.
- **Random number generator**: One or several pseudorandom number generators simulate the random behaviour of the system.
- **Statistics**: The simulator keeps track of quantitative performance measures that describe the result of the simulation.
- **Ending condition**: The condition in which the simulation is terminated. In this case, after time t has elapsed.

2.7 Summary

In this chapter, we have reviewed the existing literature on open shop scheduling and established that our problem is an open research problem that is NP-hard. The concept of metaheuristics has been defined and a description has been given of the nature-inspired algorithms implemented in this research, ant colony optimization and genetic algorithms. Finally, we have discussed the machine learning task of reinforcement learning as a way of solving Markov decision processes and given a brief overview of discrete-event simulation.

Chapter 3

Methodology

3.1 Research methodology

To address the problem statement from 1.4, we use the scientific method, which involves developing a testable hypothesis, gathering data and making observations about the results. The problem can be divided into three central questions that the research experiment must answer:

- Is it possible to use historical data to improve the scheduling algorithm?
- If it is, which model should be used?
- How well does the selected model perform?

A machine learning model applied to a practical problem should be validated on the basis of three measures: *Accuracy*, *reliability* and *usefulness* (Lukawiecki, 2016).

In the case of reinforcement learning, there is no meaningful way of measuring predictive accuracy, since the exact values of optimal outcomes are not known. Instead, the performance of the model is measured by some objective function, or cost function. This metric is discussed in section 3.3.

The reliability of a model describes how well it generalizes to unseen data. This is assessed through cross-validation and subsequent hypothesis testing, described in sections 3.4 and 3.5 respectively.

Finally, the usefulness of a model is a qualitative assessment performed by a domain expert. The expert's experience in the domain in which the model is to be applied will help quantify what would be a significant result in the research experiment. This is discussed in section 3.5.1.

3.2 Simulation

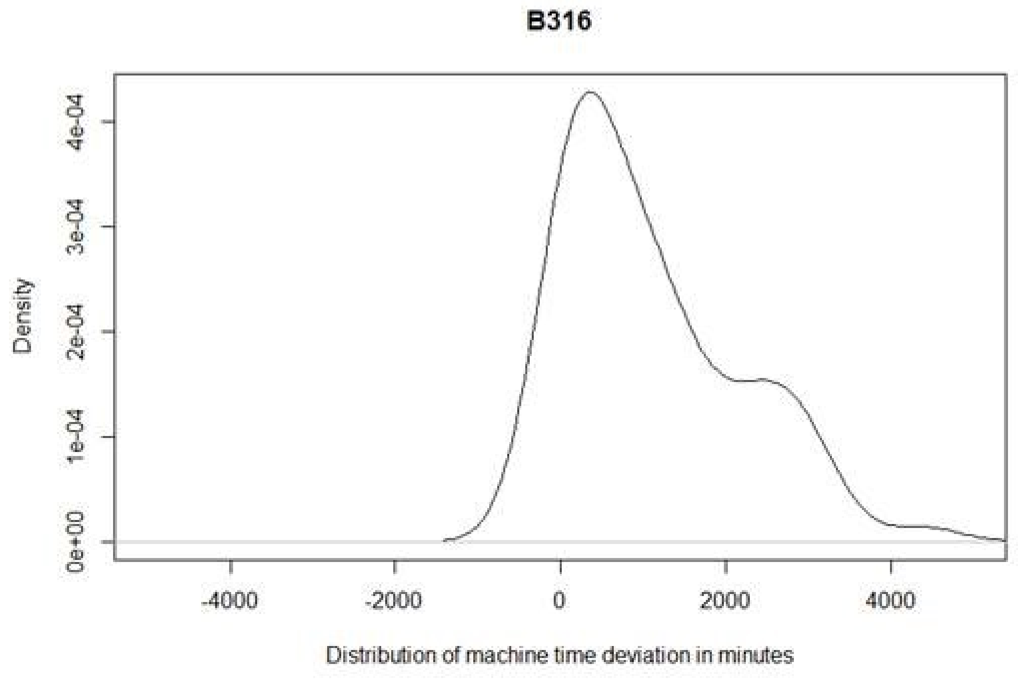
To train, validate and test machine learning models against historical data, a discrete-event simulator has been implemented. The technical implementation is described in chapter 4.

The simulator contains a model of the entire machine shop as well as a framework for machine learning, cross-validation and two-sample hypothesis testing. It is also capable of forecasting realistic operation processing times based on historical observations, using kernel density estimation.

3.2.1 Kernel density estimation

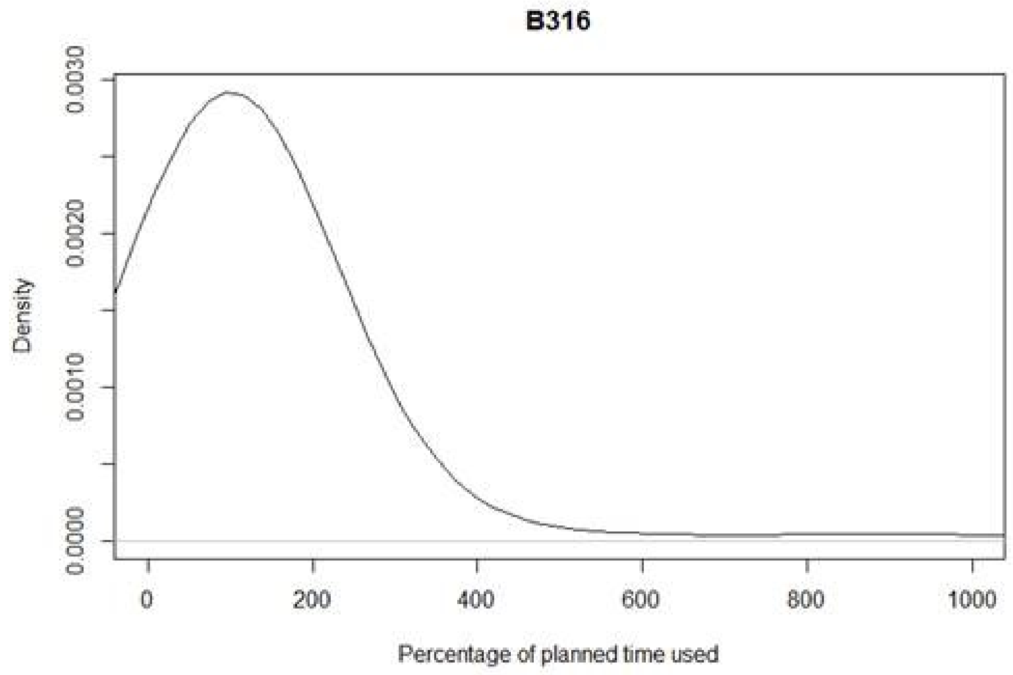
When forecasting the processing times of operations, the simulator samples from probability density functions of processing times on each machine in the machine shop which were estimated by kernel density estimation on historical data. Figure 3.1 shows an example of such an estimate.

Figure 3.1: Probability density function of deviations on machine B316.



The operations that are scheduled on a single machine vary greatly in planned duration. The magnitude of duration deviations increase as the planned duration increases. Thus, it makes more sense to sample from the density of percentage deviations instead of absolute time deviations. In the example, we can see that the density of percentage deviations has a smoother curve, which will typically be the case.

Figure 3.2: Probability density function of deviations (%) on machine B316.



The procedure for sampling from the density of machine m is as follows:

1. Generate a random uniform variable U on [0,1].

2. Find an index $i = \lfloor U * n \rfloor$, where n is the size of the sample vector $\mathbf{x}$ containing historical duration deviations on machine m.

3. Generate a sample from the Gaussian kernel of x_i.

The sample is then multiplied with the planned duration to get a forecasted processing time for the operation on machine m. The Python code for this procedure is listed in appendix A.1.

3.3 Cost function

The objective of the machine learning model is to minimize mean tardiness. If we let d_i be the deadline of a job i and C_i be the time of completion, then the tardiness of the job is defined as

$$T_i = max\{0, C_i - d_i\} \tag{3.1}$$

The total tardiness of a system is

$$T = \sum_{i=0}^{n} T_i \tag{3.2}$$

and the mean tardiness is

$$\overline{T} = \frac{1}{N} \sum_{i=0}^{n} T_i \tag{3.3}$$

where N is the number of jobs.

By jobs, we are referring to work orders in this context. Since all the discrete-event simulations are stopped by an ending condition, not every work order will be completed during the simulation. Using total tardiness as the cost function could cause the learning algorithm to prefer orders with shorter completion time, causing unwanted bias in the model. Thus, we will perform the research experiments using mean tardiness as the measure of performance instead. In practice, this computational trick makes a minor difference, as the number of completed jobs in the typical time windows vary little compared to the variance of total tardiness.

3.4 Data partitioning

In machine learning, it is common to divide datasets into *partitions* and subsets in order to separate the data used for model training and model testing.

3.4.1 Cross-validation

When choosing between predictive models, one must consider how well the model performs on data that was not used in building the candidate models. The models should be subjected to cross-validation (Walpole, Myers, Myers, & Ye, 2012, p. 487).

There are two main cross-validation techniques in machine learning. The simplest is *holdout cross-validation*, in which the dataset is split into a training set and a test set. This approach is problematic, since the same partition that is used to select the best model is also used to evaluate the performance of the model. This means that evaluation of model performance is subject to selection bias (Najafzadeh, 2014) and the model will be more likely to overfit (Raschka, 2015, p. 173). A better method is to separate the data into three parts: a training set, a validation set and a test set.

The other technique is called *k-fold cross-validation.* The training set of the dataset is split into *k* folds and the model is successively trained on *k-1* folds and tested on the remaining fold. The process is repeated *k* times so that every fold is used for testing exactly once (Raschka, 2015, p. 175).

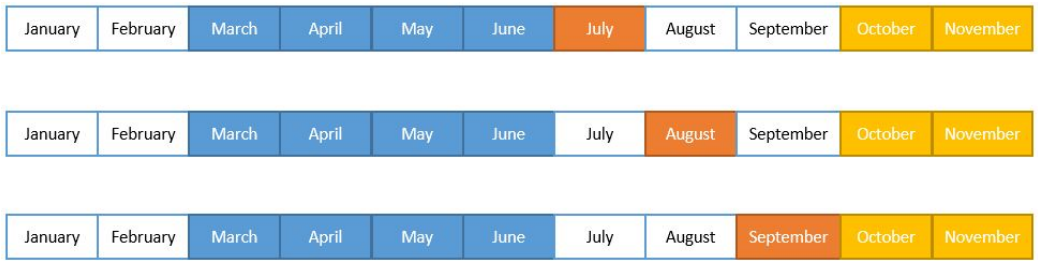
Neither of these techniques are particularly suitable for time series data, which we are dealing with in this research. A more principled approach for time series data based on k-fold cross-validation is *forward chaining* (Williams, 2011).

3.4.2 Forward chaining

The idea of forward chaining is to preserve the most important principle of cross-validation for time series data: Chronology. The model should only be exposed to knowledge during training that would have been available at the time, and validated on data that was collected later. As an example, if a machine learning algorithm is trained on historical data from July and observes that job j was delayed by a certain amount of time, the trained model will be better able to schedule j in June. Yet, we obviously don't have exact knowledge about the future in the real world. Thus, during training we only expose the learning algorithm to data that was collected before the validation and test data. Figure 3.3 shows three configurations that are used for several models in the research experiments [1].

[1] Blue = training, orange = cross-validation, yellow = testing, white = not used.

Figure 3.3: Forward chaining on 2016 data from Aarbakke's database.



In our research, this is sometimes done slightly different, because we are also training the model on *forecasted* data. This means that the model can be trained on forecasted July data and validated on data from the same month, because the forecasted data was constructed using information that was available before July 1, again preserving the principle of chronology.

Models are also cross-validated against forecasted, and not just historical, numbers. The reasoning behind that is to evaluate the models on a wider range of scenarios. The actual events that took place in the few months of data that we're testing against is simply one scenario out of many likely scenarios that could have occurred in a highly uncertain environment. We want to create an antifragile agent that improves its performance in the face of uncertainty.

The only concern with the forward chaining approach on Aarbakke's dataset is that the most extensive work orders span many months and will overlap the different partitions. A way to counter that would be to intentionally remove some work orders from the dataset to avoid any overlap. However, that would cause a bigger problem, since we are interested in the dynamics of the entire system, and removing work orders from the experiments would create unrealistic scenarios. With the data available, it is simply not possible to create an experiment that covers the dynamics of the entire system without any overlap.

On the positive side, the work orders are comprised of operations with durations that are measured in the hours and minutes. These will very rarely overlap partitions. The forward chaining process aims at minimizing the effects of overlapping work orders.

3.5 Hypothesis testing

Once a candidate model has been selected, a statistical hypothesis test will establish whether the following null hypothesis (H_0) can be rejected: *It is impossible to achieve a significant reduction in mean tardiness by altering the scheduling algorithm in the Scheduling Workbench using historical data.*

If the null hypothesis is rejected, its mutually exclusive logical negation, the alternative hypothesis (H_1), must be accepted: *It is possible to achieve a significant reduction in mean tardiness by altering the scheduling algorithm in the Scheduling Workbench using historical data.*

If the null hypothesis can't be rejected, the hypothesis test has proved inconclusive.

3.5.1 Significance

In order to be able to test the hypothesis, we need to clarify what constitutes a significant reduction in mean tardiness.

It is clear that late work orders cause a multitude of negative consequences for Aarbakke, including:

- Increased resource use.
- Loss of future orders.
- Penalties in the form of fines.
- Compounding effect of late orders causing more late orders.

Each customer order has a huge impact on company revenue, often in the millions of NOK. (R. Thu, personal communication, November 4, 2016).

Comparing these impacts with the relatively low cost of altering the scheduling algorithm, we will consider a reduction of at least 1.0% in mean tardiness significant, given that a p-value of less than 0.05 can be obtained in a Student's t-test when running multiple experiments against the benchmark algorithm.

3.6 Benchmark algorithm

The benchmark algorithm in the Scheduling Workbench is a variation of the dispatch rule Earliest Due Date (EDD), with slight modifications. The calculated priority number given to a work order is a function of the current day, using a relative time scale. We will denote it as $\Pi(n)$. Let n be a relative integer representation of the current day, n_{start} a relative integer representation of the deadline to start work on the order and n_{end} a relative integer representation of the deadline to finish work on the order, all using the same zero reference. The priority function is then:

$$\Pi(n) = c_1 * (n + 777 - n_{start}) + c_2 * (n + 777 - n_{end}) + u(n) \tag{3.4}$$

where c_1 and c_2 are arbitrary constants. Throughout this research, the following values were used: $c_1 = 0.8$, $c_2 = 0.2$.

$u(n)$ is a step function, defined as follows:

$$u(n) = \begin{cases} 100, & \text{for } n_{end} - n < 7 \\ 92, & \text{for } 7 \leq n_{end} - n \leq 13 \\ 80, & \text{for } 14 \leq n_{end} - n \leq 20 \\ 72, & \text{for } 21 \leq n_{end} - n \leq 30 \\ 40, & \text{for } 31 \leq n_{end} - n \leq 90 \\ 12, & \text{for } 91 \leq n_{end} - n \leq 120 \\ 0, & \text{for } n_{end} - n > 120 \end{cases}$$

3.7 Summary

We started this chapter with an outline of the research methodology. We described the simulation approach used to train, validate and test models. We defined the cost function used to measure model performance. We discussed how the original dataset was split into three partitions, using the concept of forward chaining. We described the details of the benchmark algorithm. Finally, the null hypothesis of the research experiment was defined, and we quantified what we would be a statistically significant improvement in the cost function, compared to the benchmark algorithm.

Chapter 4

Design and implementation

4.1 System specifications

4.1.1 Computer specifications

All development and testing was performed on a PC with these specifications:

Model:	ASUS N552VX
Processor:	Intel Core i7-6700HQ
Clock rate:	2.60 GHz
Memory:	16 GB RAM
System type:	64-bit
OS:	Microsoft Windows 10.0.14393

4.1.2 Software versions

The Anaconda data science distribution of the Python programming language was used for developing the simulation software along with the following packages (version number on the right):

Anaconda	4.0.0
conda	4.2.7
Jupyter	1.0.0
matplotlib	1.5.1
NumPy	1.11.1
pandas	0.18.0
pyodbc	3.0.10
Python	3.5.2
SciPy	0.18.0
SimPy	3.0.8
SQLAlchemy	1.0.12

Other software tools used for purposes such as statistical analysis, data storage, automation, version control and report writing include:

EmEditor	16.0.2
Git	2.8.1
MiKTeX	2.9
Windows Powershell	5.1.14393.206
Powershell ISE	3.0
R	3.3.0
RStudio	0.99.902
SQL Server	13.0.15000.23
Sublime Text	3.3126
TeXStudio	2.11.0
Visual Studio Code	1.6.0

4.2 Data preparation and automation

At the start of every simulation run in the discrete-event simulator, three tables are generated in the SQL database based on historical data. Each table is created based on multiple joins from tables in the original database.

- **Capacity**: This table contains all allocated capacities and constraints on each machine.
- **Schedule**: This table contains the entire schedule in the Scheduling Workbench at the historical point in time when the simulation is started.

- **Statistics**: This table contains statistics about processing times on each machine.

The capacity and schedule tables are used to generate schedules in the simulator, while the statistics table is used to simulate forecasted processing times.

The entire dataset is automatically refreshed every hour with dumps from the Scheduling Workbench using a series of Powershell scripts [1].

4.3 Discrete-event simulator

The discrete-event simulator (DES) is at the core of technical solution to the optimization problem presented in this thesis. It consists of a console application written in Python that models and simulates the machine shop in Aarbakke's factory.

The main components of the DES are implemented using SimPy, a discrete-event simulation library for Python. SimPy keeps track of the simulation clock as well as access rights for operations to be performed on machines through queueing.

The library is largely built around the *yield* statement, which is Python's internal implementation of the generator pattern. The advantage of this is that generators use lazy evaluation, which can reduce the running time of the program drastically.

[1]Powershell is an automation tool for Windows.

Built on top of the DES is an optimization and statistical hypothesis testing framework. It is capable of constructing candidate models from two types of metaheuristic optimization algorithms (ACO and GA). In addition, it can cross-validate the results using standardized two-sample hypothesis tests to assess the performance of models. We will henceforth refer to the entire system as the *simulator*.

Figure 4.1: Simulator workflow.

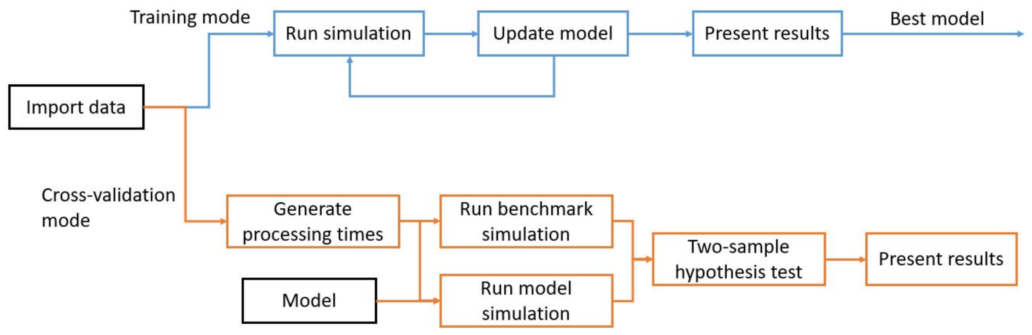


Figure 4.1 shows a flow diagram of the simulator workflow. Every run starts by importing the relevant data based on a set of input parameters.

If training mode is specified (blue track), the simulator will iteratively run simulations and update the model according to the update rules in the metaheuristic algorithm until the specified number of iterations are completed. At the end, the cost function of the best candidate solution and average cost function of the population of solutions will be plotted per iteration. The model parameters that define the best model are given as output.

If cross-validation mode is specified (orange track), the simulator will generate an array of either planned, forecasted, random or historical processing times. The benchmark algorithm and candidate model are then simulated using the same numbers in parallel for a specified number of times. After the simulations, a two-sample hypothesis test is performed, comparing the candidate model to the benchmark, and the results are presented as output.

4.3.1 Parameters

When discussing simulation parameters, it is important to distinguish between the three different types of parameters present within the simulator.

The machine learning *model parameters* define the mathematical function that the algorithm has learned implicitly from the data. The machine learning *hyperparameters* are a set of explicitly stated properties of the machine learning algorithm that can be tuned to alter its behaviour. *The parameters of the simulator* are inline arguments to the console application that define the experiment. In this subsection, we will describe the latter.

- **Granularity**: This parameter refers to how often a machine updates its capacity. By default, it is set to 15 time steps (minutes), which offers a significant improvement in execution speed over updating every time step.

- **Import type**: This parameter determines where the simulator imports the data from, either from a CSV file, a local SQL database or a cloud-based SQL database.

- **Iterations**: Number of iterations in the experiment. For each iteration, at least one simulation is run.
- **k**: Number of discrete time steps in the simulation. Each step represents a minute.
- **Mode**: Algorithmic mode. See 4.3.2.
- **Month**: This parameter defines the starting point, t_0, of a simulation in simulated time, which is always at 5:00 on the first weekday of a month.
- **Time mode**: See 4.3.2.
- **Verbosity**: This parameter controls the amount of text the application will output to the console. It has five levels (0-4).

4.3.2 Modes

The simulator can be run in four different algorithmic modes, depending on the aim of the experiment:

- **Benchmark**: This mode is used to assess the performance of the scheduling algorithm in the Scheduling Workbench. It can also be used to assess alternative algorithms based on deterministic dispatch rules.

- **Ant colony optimization**: This mode infers optimal permutations of work order priorities using an implementation of the ant colony optimization algorithm. The implementation is described in detail in 4.4.

- **Genetic algorithm**: This mode infers a generalized linear model of composite dispatch rules using a genetic algorithm. The implementation is described in detail in 4.5.

- **Cross-validation**: This mode is used to run a two-sample hypothesis test on a trained model against the benchmark algorithm used in the Scheduling Workbench.

Four different time modes are available:

- **Historical time** (deterministic): When using this mode, the processing times of operations are the historical times that were recorded after each operation finished.

- **Planned time** (deterministic): When using this mode, the processing times of operations are the durations as they were planned in the original schedule.

- **Forecasted time** (stochastic): This time mode is mainly used in conjunction with the cross-validation mode to compare a model against the benchmark. For each iteration of the experiment, a set of forecasted processing times are generated and used in consecutive simulations for both the benchmark and the model.

- **Random time** (stochastic): When using this mode, all processing times are random, although algorithmically generated numbers are never random in the strictest sense of the word (”Can a computer generate a truly random number?”, 2011).

4.3.3 Machine shop model

The machine shop is modelled through four classes that simulate the workflow at Aarbakke.

- The *MachineShop* class holds an array of the available machines and keeps track of all the processing time delays that have been registered on every machine. At the end of a simulation, the class's evaluate() method outputs a summary of the time delays.

- The *Machine* class models processing machines in the factory. Every instance of this class corresponds to an actual machine. The main responsibility of the class is to grant or deny operations access to the machine. This is accomplished with the class's capacity_controller() method, which uses data from the imported Capacity table to control access to the machine and handle the queue of operations. The full Python code of the capacity_controller() method is listed in appendix A.9.

- The *WorkOrder* class models work orders. It holds arrays of its associated operations and children work orders, both of which could be empty. Its calculate_features() method calculates the feature vector used in the genetic algorithm, while its set_priority() method assigns the work order a priority number equal to the scalar product of the feature vector and the weight vector learned by the genetic algorithm.

- The *Operation* class holds metadata about operations through its class members. The class has one method, forecast(), which preassigns the operation with a forecasted processing time that can be reused in multiple simulations.

In addition, a set of separate functions built on top of SimPy link the data model to the simulations.

The operation() function models the actual execution of operations. It extends SimPy's process() function, which is the basic generator upon which the rest of the library is built. We will elaborate on the implementation of the operation() function, as it is the central component that drives the simulator.

The pseudocode in algorithm 1 presents a simplified, high-level version of the operation() function. The function takes a list of arguments as input, listed in table 4.1.

Table 4.1: operation() arguments

Argument	**Description**
environment	The SimPy simulation environment the operation is a part of.
machine	The *Machine* object the operation is scheduled on.
time_mode	The time mode of the current simulation (see 4.3.2).
historical_time	Historical processing time (see 4.3.2).
planned_time	Planned processing time (see 4.3.2).
forecasted_time	Forecasted processing time (see 4.3.2).
planned_start	The simulation time at which the operation is scheduled to start.

Algorithm 1: operation(arguments)

```
1  begin
2      yield until planned_start
3      if time_mode is 'historical' then
4          time_left = historical_time
5      else if time_mode is 'deterministic' then
6          time_left = planned_time
7      else if time_mode is 'forecast' then
8          time_left = forecasted_time
9      else if time_mode is 'random' then
10         time_left = planned_time * sample_from_density(machine)
11     request_time = environment.current_time
12     yield machine.access_controller.request() /* wait for access */
13     while time_left do
14         if machine has capacity then
15             time_left = time_left - max(machine.capacity, 1)
16     time_spent = environment.current_time - request_time
17     machine.delay = machine.delay + time_spent - planned_time
```

First, the function sleeps until the scheduled start of operation using the yield statement (line 2).

Then, a processing time is assigned to the *time_left* variable depending on the time mode of the current simulation (lines 3-10). If the *random* time mode is enabled, a random processing time is sampled from the machine's kernel density estimation (line 10).

Next, the operation requests access to the machine it is scheduled on (line 12). If the machine has a queue of operations, the function sleeps until access is granted.

Once access is granted, the operation starts consuming minutes from the *time_left* variable (lines 13-15). For every minute, a number equal to the machine's capacity is consumed, for a maximum of one minute (line 15).

Being part of the simulation *environment* means that the function knows the state of the simulation clock and is able to record the simulated delay once it is finished (line 17), which is its main purpose.

4.3.4 Scheduling

Scheduling at Aarbakke follows the principles of Just-In-Time (JIT) manufacturing, also known as lean manufacturing. Every work order is given a priority number and the order with the highest number is scheduled until all the orders have been assigned to the schedule. When a work order is being scheduled, its operations are prioritized in numerical order, starting with the smallest. An operation is always assigned to exactly one machine (or resource).

The scheduling algorithm always looks for the latest possible starting time for an operation where there is enough capacity on the machine to finish the operation just before its deadline if it uses the planned amount of time. If there is not enough capacity before the deadline, the algorithm will look for a later starting time as close to the deadline as possible. These scheduling rules are universally true for all the algorithms used in this research, both the deterministic and heuristic ones. The only differences between the algorithms are the order in which work orders are scheduled.

Scheduling an operation is a computationally intensive procedure that is invoked often during the course of a simulation. At the early stages of development, this was a major bottleneck in the execution time of the simulator. Algorithm 2 contains two lines of code that optimizes the procedure that finds the starting point of the schedule using Python's NumPy library. Note that this is actual Python code, not pseudocode.

Algorithm 2: Vectorized Python code

```
1 reverse = capacities[0:latest_start][::-1]
2 scheduled_start = len(reverse) -
    np.amin(np.where(np.cumsum(reverse)>consumption))
```

The *capacities* array contains the capacity on a machine for every minute (or, with a granularity of 15, every 15 minutes) of the simulation. The *consumption* variable represents the number of minutes the operation is planned to consume on the machine. Finally, *np.amin*, *np.where* and *np.cumsum* are NumPy functions for absolute minimum, where clause and cumulative sum. The algorithm finds the latest index in the the capacities array where the operation can be scheduled and still have enough capacity left before the deadline to finish the operation in time if it takes as long time as planned. Using this vectorized implementation improved the performance by an order of 750 times over a standard loop implementation.

4.4 Ant colony optimization

For the purpose of ant colony optimization, a custom, but fairly straightforward, algorithm was implemented.

The colony maintains a matrix $\mathbf{T}$ with each element τ_{ij} representing the edge between node i and node j. The ants in the colony traverse this graph and deposit pheromones depending on the cost (mean tardiness) their candidate solutions achieve.

In the implementation of the Elitist Ant System, the ant with the best path through the graph in the current iteration, as well as the ant with the best global path, both deposit pheromones to the edges along the paths with the following update equation:

$$Q/C \tag{4.1}$$

Here, Q is an arbitrary parameter that is selected to fit the data while C is the value of the cost function in the path.

The implementation of the Rank-Based Ant System (ASRank) follows this procedure:

1. Sort the colony by cost function starting with the smallest.

2. Select the first half of the ants for pheromone deposits.

3. Update edges using equation 4.1, where C is the each ant's cost.

Ant colony optimization depends on the following hyperparameters:

- **Colony size**: The number of ants in the colony.
- **Evaporation rate (ρ)**: The ratio in which pheromones disappear from the edges.
- **Initial pheromone level (τ_0)**: The initial pheromone level in all edges at the start of optimization.
- **Added pheromone level per iteration ($\Delta\tau$)**: The amount of pheromones deposited to each edge per iteration.
- **Q**: An arbitrary parameter that controls the rate of learning. It should be selected based on the magnitude of the input variables and size of the dataset.

The full Python code of the function that implements the ants' graph traversal is listed in appendix A.2, while the function that implements pheromone deposits is listed in appendix A.3.

4.5 Genetic algorithm

The genetic algorithm also follows a custom, but fairly straightforward, implementation. It really only has one hyperparameter, which is the population size, or the number of candidate solutions. The main concepts of genetic algorithms were described in 2.4.2.

Each training cycle, or iteration, consists of the following procedure:

1. Select two parents from the population using tournament selection.
2. Create two child phenotypes using crossover operations. Let the first child inherit three chromosomes from the first parent and one from the second parent. Let the other child inherit three chromosomes from the second parent and one from the first parent.
3. Add the children to the population.
4. Score the children by running simulations with their genotypes.
5. Select two victims from the population using a reverse tournament.
6. Remove the victims from the population.
7. Select a random phenotype from the population and mutate it using shuffle mutation.
8. Select a random phenotype from the population and mutate it using perturb mutation.

Each phenotype's genotype is a vector $\mathbf{w}$ containing the weights of four dispatch rules:

- **EDD:** Earliest Due Date. Work orders are prioritized according to the due date of their last operations. Earlier due dates have priority. Conversely, if the weight becomes a negative number, it represents the inverse rule, Latest Due Date (LDD).

- **LPT:** Longest Processing Time. Work orders are prioritized according to the sum of their planned processing times. More time-consuming orders have priority. Conversely, if the weight becomes a negative number, it represents the inverse rule, Shortest Processing Time (SPT).
- **LNO:** Largest Number of Operations. Work orders are prioritized according to the total number of operations within the order. Work orders with more operations have priority. Conversely, if the weight becomes a negative number, it represents the inverse rule, Smallest Number of Operations (SNO).
- **LNC:** Largest Number of Children. Work orders are prioritized according to their total number of child work orders. Work orders with more children have priority. Conversely, if the weight becomes a negative number, it represents the inverse rule, Smallest Number of Children (SNC).

Using four signed numbers as weights, the genetic algorithm can optimize scheduling with combinations of eight different dispatch rules.

The full Python code of the functions that implement crossovers, perturb mutations, shuffle mutations, tournament selection and reverse tournament selection are listed in appendices A.4, A.5, A.6, A.7 and A.8 respectively.

4.6 Summary

In this chapter, we have described the implementation of the discrete-event simulator and its components in detail, including an overview of its different parameters and metaheuristic optimization algorithms. The core algorithms that control operation processing and scheduling were also presented.

Chapter 5

Results and analysis

5.1 Algorithm overview

In this chapter, we evaluate and compare the following scheduling algorithms, which are abbreviated accordingly in figures and tables:

- **BM:** The benchmark algorithm used in the Scheduling Workbench. All the two-sample hypothesis tests are carried out by comparing the performance of each algorithm to this one.
- **EDD:** Earliest Due Date. Work orders are prioritized according to the due date of their last operations. Earlier due dates have priority.
- **LDD:** Latest Due Date. Work orders are prioritized according to the due date of their last operations. Later due dates have priority.
- **LPT:** Longest Processing Time. Work orders are prioritized according to the sum of their planned processing times. More time-consuming orders have priority.
- **SPT:** Shortest Processing Time. Work orders are prioritized according to the sum of their planned processing times of their operations. Less time-consuming orders have priority.

- **LNO:** Largest Number of Operations. Work orders are prioritized according to the total number of operations within the order. Work orders with more operations have priority.
- **SNO:** Smallest Number of Operations. Work orders are prioritized according to the total number of operations within the order. Work orders with fewer operations have priority.
- **LNC:** Largest Number of Children. Work orders are prioritized according to their total number of child work orders. Work orders with more children have priority.
- **SNC:** Smallest Number of Children. Work orders are prioritized according to their total number of child work orders. Work orders with fewer children have priority.
- **GA1:** Genetic Algorithm trained on historical processing times on data spanning March through June.
- **GA2:** Genetic Algorithm trained on forecasted processing times on data spanning March through June.
- **GA3:** Genetic Algorithm trained on historical processing times on the June dataset.
- **GA4:** Genetic Algorithm trained on forecasted processing times on the July dataset.
- **GA5:** Genetic Algorithm trained on planned processing times on data spanning July through September.
- **GA6:** Genetic Algorithm trained on forecasted processing times on data spanning July through September.
- **ACO1:** ASRank Ant Colony Optimization trained on historical processing time for each month.
- **ACO2:** ASRank Ant Colony Optimization trained on forecasted processing times for each month.
- **ACO3:** Elitist Ant Colony Optimization trained on historical processing time for each month.

- **ACO4:** Elitist Ant Colony Optimization trained on forecasted processing times for each month.

5.2 Metaheuristic search

Training experiments using metaheuristic reinforcement learning were intentionally restricted to keep execution time to a maximum of about 15 minutes, which is approximately the time window available in the Scheduling Workbench before a new schedule is forwarded to the ERP system.

For the training experiments, a partition of the dataset containing the first half of 2016 was used. In some experiments, data was forecasted for later months, though only using information that would have been available before July 1, 2016.

All the genetic algorithms were trained with a population size of 20 phenotypes. Similarly, all the ant colony optimization algorithms were trained with a colony size of 10 ants. All algorithms were trained for 50 iterations.

The hyperparameters were kept constant in order to give all algorithms the same prerequisites. In ant colony optimization, the following hyperparameters were used:

- Evaporation rate, ρ: 0.5
- Initial pheromone level, τ_0: 1
- Added pheromone level per iteration, $\Delta\tau$: 0.05
- Q: 1000000

All the ant colony optimization algorithms were discrete, whereas the genetic algorithms were continuous.

5.2.1 Ant colony optimization

The ant colony optimization experiments show poor convergence. Figures 5.2.1 and 5.2 present similar patterns for training experiments with the Rank-based ant system (ASRank) and Elitist ant system. The best solution improves in both cases, but the average tardiness does not decrease throughout the experiment, indicating that the algorithm has not learned a path through the solution graph that generalizes well to the problem of reducing tardiness. The same pattern appeared when using both historical and forecasted processing times.

Figure 5.1: ASRank ACO training experiment on the July dataset.

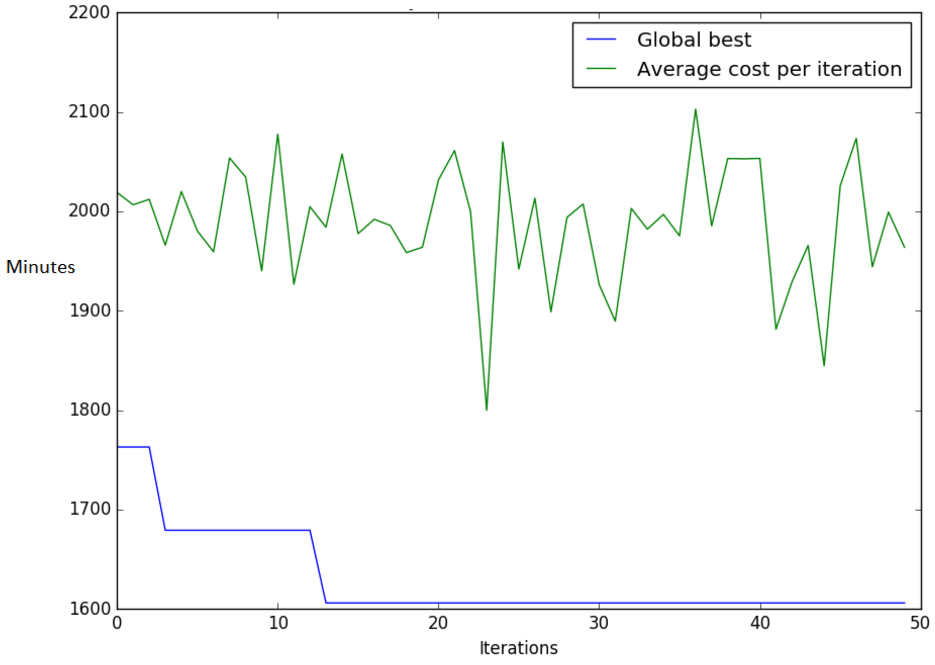
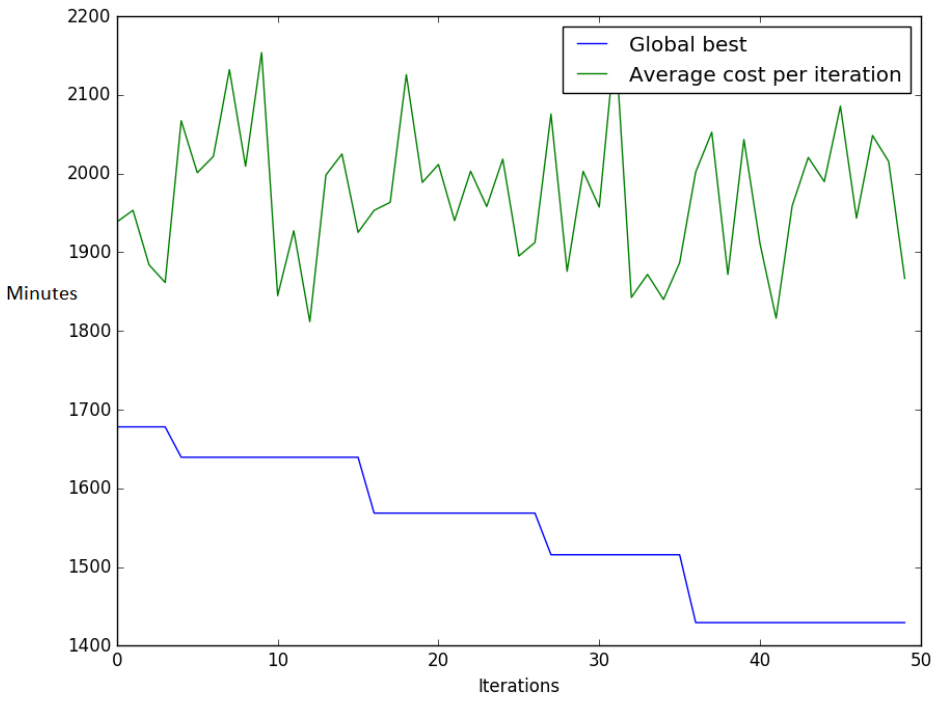


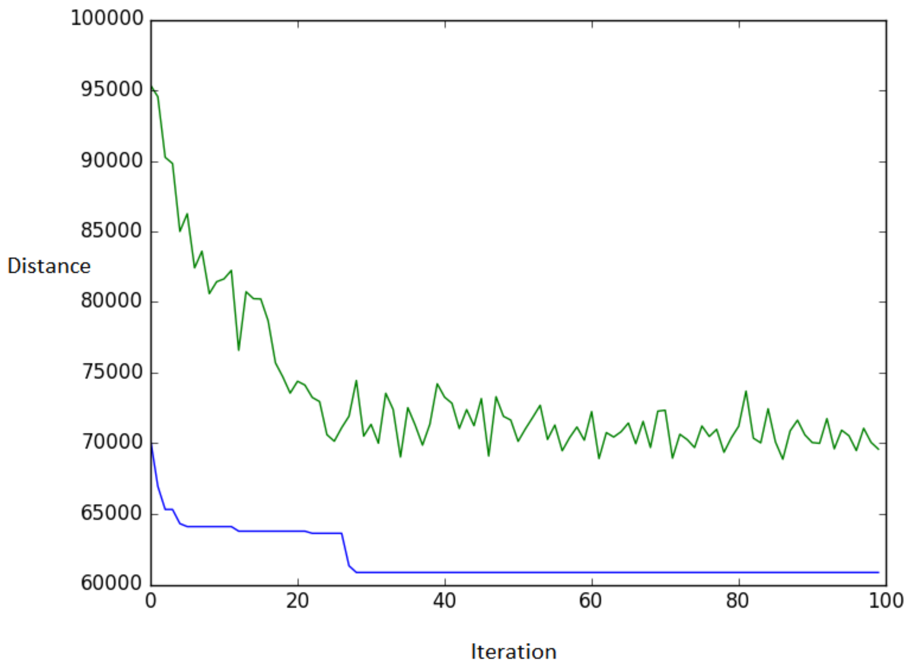
Figure 5.2: Elitist ACO training experiment on the August dataset.



The lack of convergence is mainly caused by the size of the graph that represents the search space. For a scheduling problem with N number of work orders, the size of the graph is its number of edges, $|E| = N^2$. In the datasets used in this research, the work orders number approximately 10^3, leading to graphs with sizes in the order of 10^6. The number of distinct paths through such a graph is $N!$, which becomes an exceptionally vast number in this context.

For comparison, Figure 5.3 shows the result of the very same implementation of the algorithm benchmarked against the Travelling Salesman Problem with $N = 10$ and $|E| = 100$.

Figure 5.3: ACO training experiment on the Travelling Salesman Problem.



Here, we can clearly see that the colony converges towards the global optimum, which it finds in less than 30 iterations.

Attiratanasunthron and Fakcharoenphol (2007) show that the expected number of iterations required for an ACO-based algorithm with n ants and m edges is, using big-O notation,

$$O(\frac{1}{\rho}n^2 m \log n), \tag{5.1}$$

where ρ is the evaporation rate. In our case, this would amount to 460517019 iterations, or, with an average of half a minute per iteration, an execution time of 438 years.

5.2.2 Genetic algorithms

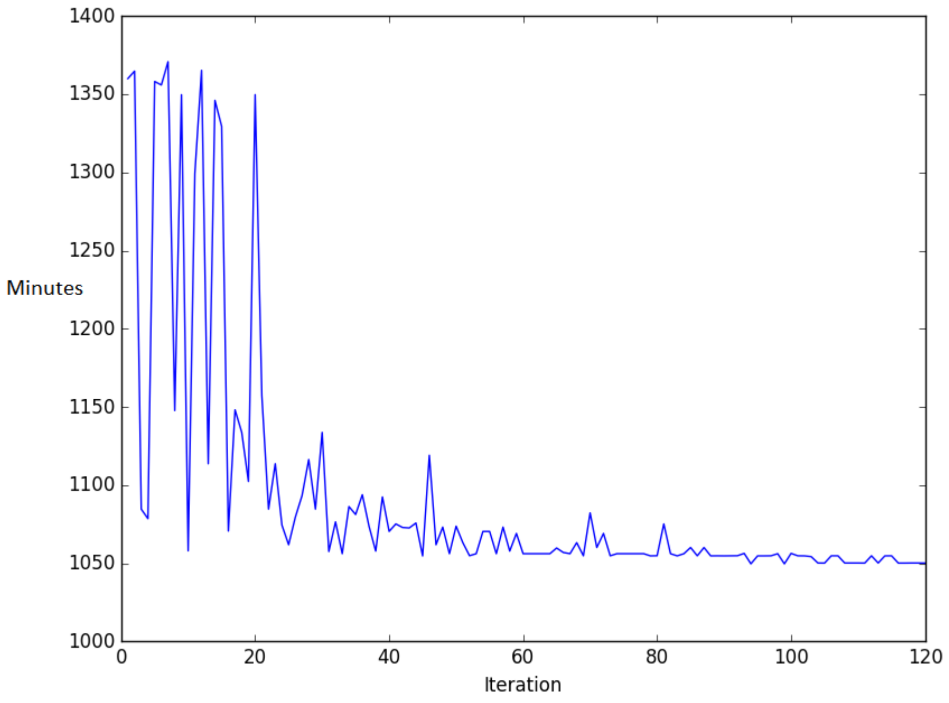
Composite dispatch rules were introduced in an effort to reduce the search space. To optimize the weights of the rules, a continuous algorithm was needed. A genetic algorithm was implemented for this purpose.

Priority numbers were calculated based on four pairs of dispatch rules: LNC/SNC, LNO/SNO, LDD/EDD and LPT/SPT (see 5.1 for an explanation of each). With this configuration, a weight vector of [0, 0, -1, 0] would correspond to the EDD rule, and conversely, [0, 0, 1, 0] to the LDD rule. Table B.3 shows the feature weights for the dispatch rules as well as the composite dispatch rules found by the genetic algorithms.

By greatly simplifying the search problem, the genetic algorithms were able to converge within the restrictions of the experiments. Figure 5.4 shows the mean tardiness of each phenotype in an experiment of 50 iterations. [1] The algorithm quickly converges, with all new members of the population producing good scores halfway into the experiment.

[1] The x-axis of the plot extends to 120 because two phenotypes are scored each iteration and the experiment starts with all 20 phenotypes in the original population being scored.

Figure 5.4: Genetic Algorithm training experiment on the September dataset with historical processing times. Individual phenotype tardiness.



Figures 5.5 and 5.6 show the total mean tardiness of the entire population as it evolves through each iteration in two different experiments with historical and forecasted processing times, respectively. As we can see, the algorithm converges much more smoothly when trained on the historical processing times found in the schedule. Whether this means that the algorithm has simply found a poor local optimum can be revealed by how well it performs. Next, we will use cross-validation through forward chaining to select a candidate model.

Figure 5.5: Genetic Algorithm training experiment on the June dataset with historical processing times. Mean population tardiness.

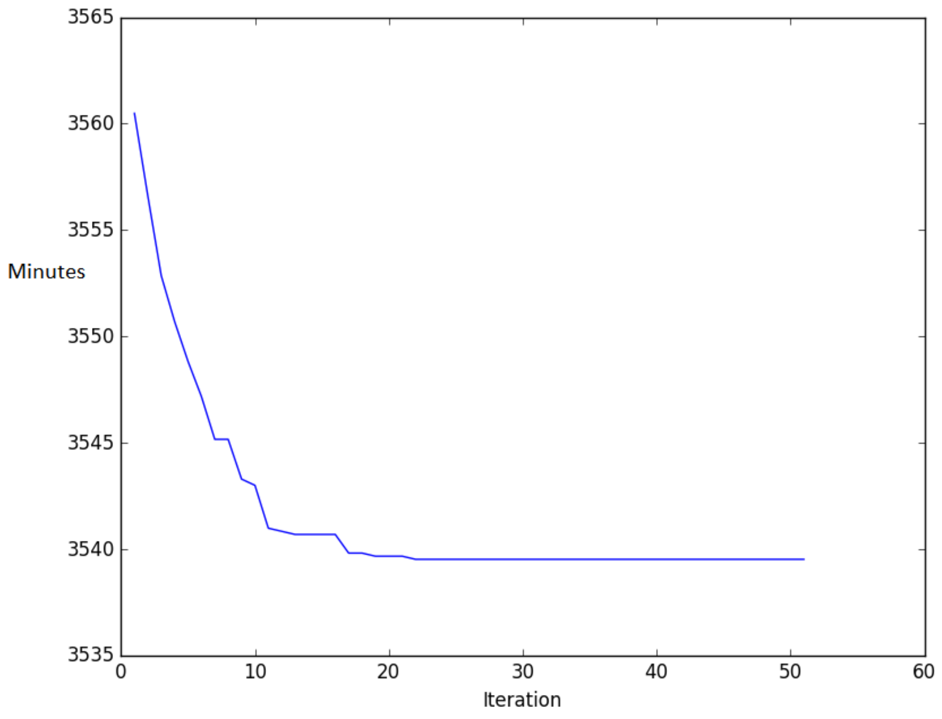
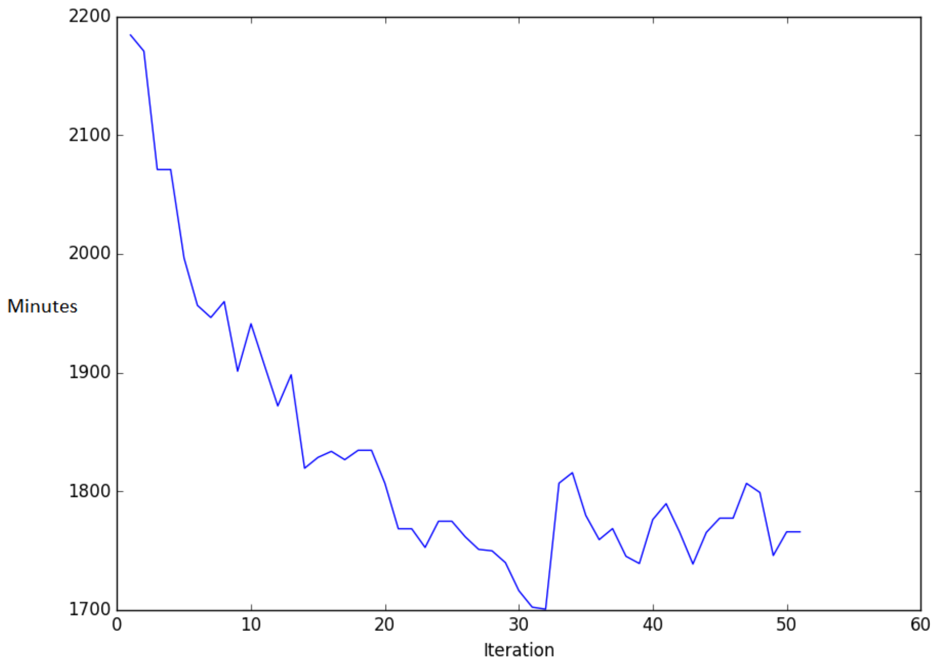


Figure 5.6: Genetic Algorithm training experiment on the July dataset with forecasted processing times. Mean population tardiness.



5.3 Cross-validation

18 different scheduling algorithms (see 5.1) were cross-validated against the benchmark on three different subsets of the validation partition of the dataset, spanning the months of July, August and September of 2016, respectively. Both historical and planned processing times retrieved from the database and forecasted processing times generated by sampling from kernel density estimations were used, leading to a total of 108 experiments. Whenever fore-

casted numbers were used, each algorithm went through 50 iterations with 50 different sets of forecasted processing times. A pseudorandom seed was implemented, ensuring that all algorithms were exposed to the same samples, providing a fair comparison. Performing Student's t-tests on the results, we were able to calculate p-values and confidence intervals that indicate whether there was a statistically significant difference between the benchmark and the algorithm under validation.

First, we will look at the experiments run on the subset of the validation partition containing forecasted data for July 2016. The data was constructed using statistical information available before July 1. Table 5.1 and figure 5.7 summarize the results.

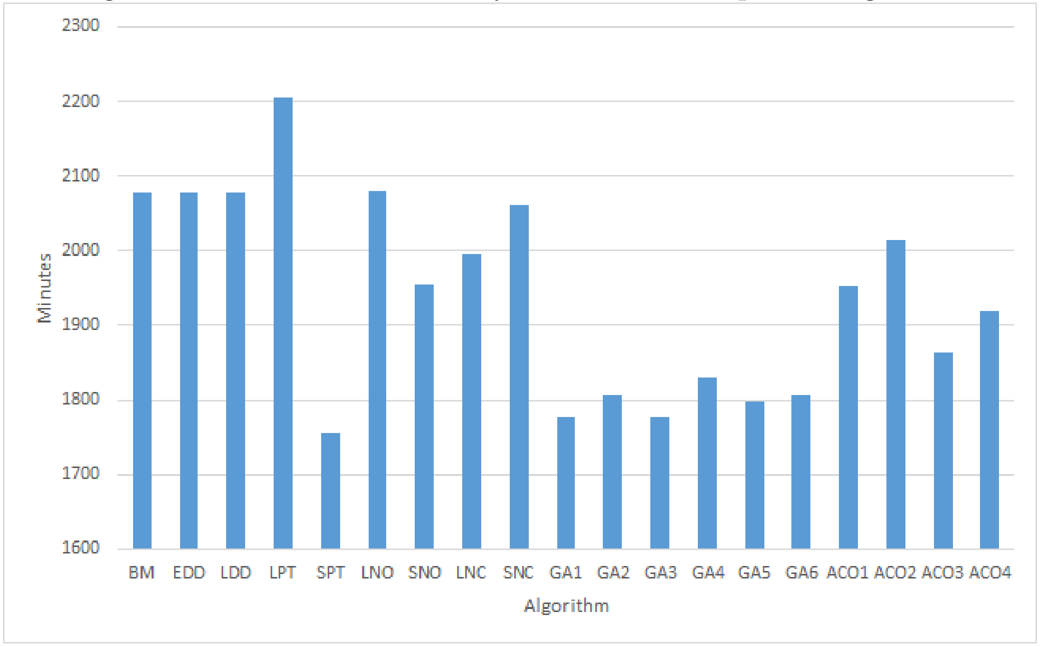
Throughout the 50 iterations, the benchmark incurred a mean tardiness of 2078 minutes. This was exactly the same as the EDD rule, which is a close approximation of the benchmark algorithm. Among the other deterministic dispatch rules, the most striking result was achieved by the SPT rule. In fact, it obtained the lowest mean tardiness estimate out of all 18 algorithms on forecasted July data, 1755 minutes, which is a 15.5 % reduction compared to the benchmark. Based on the experiment, the SPT rule reduces the mean tardiness with between 242 minutes and 404 minutes with 95 % confidence. The Student's t-test yielded a very small p-value (less than 0.001), which proves the statistical significance of the result. The only other dispatch rule to achieve a significant improvement was the SNO, with a mean reduction in tardiness of 122 minutes and a p-value of 0.005.

Table 5.1: July cross-val. with forecasted processing times (n=50)

Algorithm	$\overline{T}$	Δ	Δ **(%)**	**L. bound**	**U. bound**	**p-value**
BM	2078	0	0.00	0	0	1.000
EDD	2078	0	0.00	0	0	1.000
LDD	2077	-1	-0.05	-87	85	0.878
LPT	2206	128	6.16	24	232	0.014
SPT	1755	-323	-15.54	-404	-242	0.000
LNO	2080	2	0.10	-95	99	0.969
SNO	1956	-122	-5.87	-208	-35	0.005
LNC	1996	-82	-3.95	-187	24	0.120
SNC	2061	-17	-0.82	-105	72	0.706
GA1	1776	-302	-14.53	-389	-216	0.000
GA2	1807	-271	-13.04	-365	-177	0.000
GA3	1776	-302	-14.53	-388	-216	0.000
GA4	1830	-248	-11.93	-320	-176	0.000
GA5	1799	-279	-13.43	-370	-188	0.000
GA6	1806	-272	-13.09	-353	-191	0.000
ACO1	1952	-126	-6.06	-218	-34	0.006
ACO2	2014	-64	-3.08	-146	18	0.117
ACO3	1864	-214	-10.30	-309	-119	0.000
ACO4	1920	-158	-7.60	-246	-17	0.000

All the six genetic algorithms were able to perform better than every dispatch rule but the SPT. They achieved reductions in mean tardiness from 11.9 % (GA4) to 14.5 % (GA1 and GA3) compared to the benchmark. All were significant reductions within a 95 % confidence interval with very small p-values.

Figure 5.7: Mean tardiness July with forecasted processing times.



The solutions found by the ant colony optimization algorithms performed significantly better than the benchmark with the exception of ACO2. With a rather high p-value of 0.117, there is statistically a 11.7 % chance that the true performance of ACO2 would be no better or worse than the benchmark. ACO1, ACO3 and ACO4 all achieved significant reductions in mean tardiness, by 6.1 %, 10.3 % and 7.6 %, respectively. The best ant colony solution was still worse than the worst genetic solution on this subset.

Using the forward chaining technique, we roll forward to the subset of the validation partition containing forecasted data for August 2016. Table 5.2 and figure 5.8 summarize the results of the experiments.

The mean tardiness incurred by the benchmark algorithm through 50 cycles of forecasted processing times was 2014 minutes. Again, the EDD rule replicates the results of the benchmark. The SPT rule performs significantly better than the benchmark again, with a p-value of 0.006 and a mean tardiness estimate of 1838, which is an 8.5 % reduction. The SNO rule achieved results on the borderline of what can be considered statistically significant with a p-value of 0.052, although we can not say with a 95 % confidence that it improves upon the benchmark.

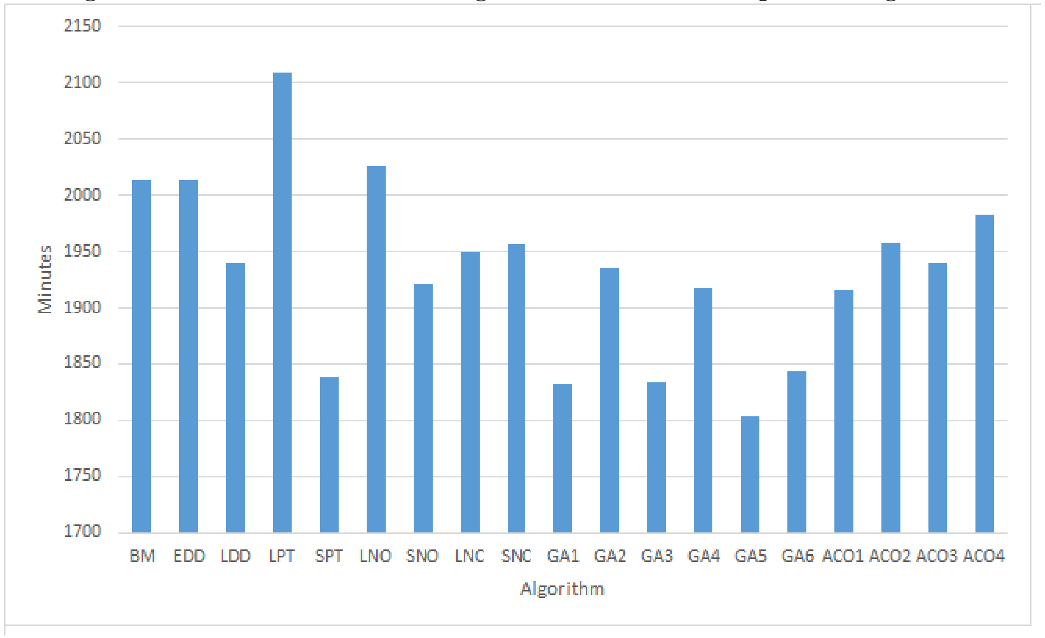
Looking at the genetic algorithms we find the best results. With the exceptions of GA2 and GA4, they achieved statistically significant reductions in mean tardiness, with estimates ranging from 8.2 % (GA6) to 10.2 % (GA5).

The solutions provided by ant colony optimization gave noticeably uneven results. Although their mean estimates are between 1.5 % (ACO4) to 4.7 % (ACO1) lower than the benchmark, the variances in the results are high enough that the p-values lie between 0.1 and 0.6.

Table 5.2: August cross-val. with forecasted processing times (n=50)

Algorithm	$\overline{T}$	Δ	Δ **(%)**	**L. bound**	**U. bound**	**p-value**
BM	2014	0	0.00	0	0	1.000
EDD	2014	0	0.00	0	0	1.000
LDD	1940	-74	-3.56	-188	39	0.191
LPT	2109	95	4.57	-67	257	0.242
SPT	1838	-176	-8.47	-303	-49	0.006
LNO	2026	12	0.58	-115	138	0.853
SNO	1922	-92	-4.43	-187	3	0.052
LNC	1950	-64	-3.08	-184	56	0.287
SNC	1957	-57	-2.74	-163	49	0.285
GA1	1832	-182	-8.76	-307	-58	0.001
GA2	1935	-79	-3.80	-193	36	0.169
GA3	1834	-180	-8.66	-294	-67	0.001
GA4	1917	-97	-4.67	-221	27	0.118
GA5	1803	-211	-10.15	-325	-97	0.000
GA6	1843	-171	-8.23	-280	-62	0.002
ACO1	1916	-98	-4.72	-215	20	0.096
ACO2	1958	-56	-2.69	-185	73	0.381
ACO3	1939	-75	-3.61	-232	82	0.337
ACO4	1983	-31	-1.49	-150	88	0.601

Figure 5.8: Mean tardiness August with forecasted processing times.



The last subset of forecasted data in the cross-validation partition contains data for September 2016. Table 5.3 and figure 5.9 summarize the results of the experiments.

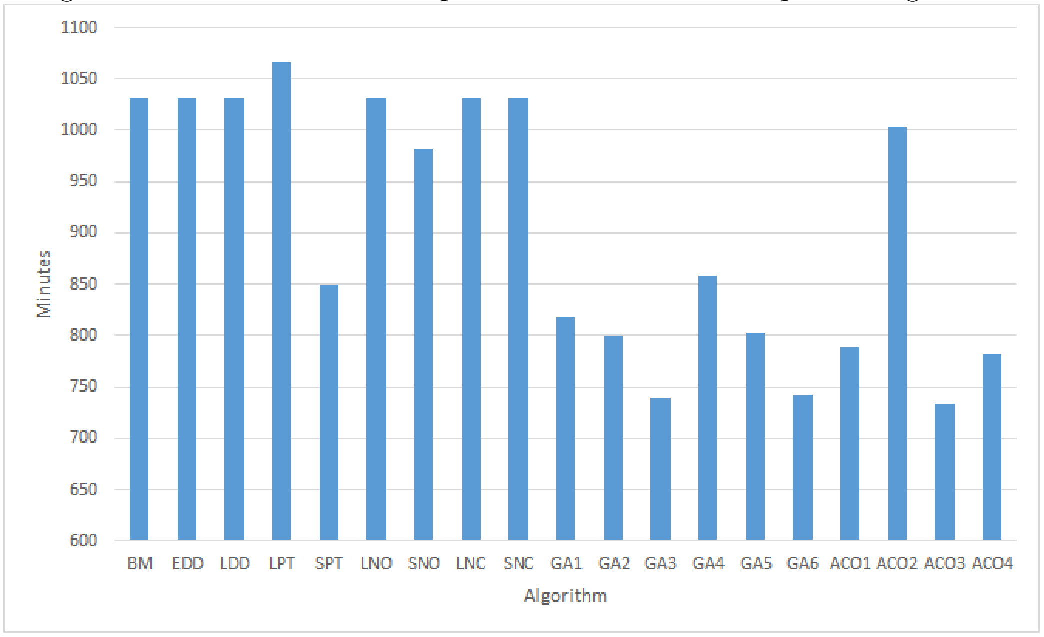
The benchmark and the EDD rule both incurred a mean tardiness of 1032 minutes in the experiments. Curiously enough, the inverse LDD rule is very close with a mean tardiness of 1031 minutes. The SPT rule performs better than the benchmark with a mean tardiness estimate of 850 minutes, but with a higher p-value of 0.075 this time. Other dispatch rules do not offer significant improvements in these experiments.

Table 5.3: September cross-val. with forecasted processing times (n=50)

Algorithm	$\overline{T}$	Δ	Δ **(%)**	**L. bound**	**U. bound**	**p-value**
BM	1032	0	0.00	0	0	1.000
EDD	1032	0	0.00	0	0	1.000
LDD	1031	-1	-0.03	-300	299	0.997
LPT	1066	34	1.64	-212	280	0.784
SPT	850	-182	-8.76	-387	23	0.075
LNO	1032	0	0.00	-234	234	1.000
SNO	982	-50	-2.41	-326	225	0.715
LNC	1032	0	0.00	-251	251	1.000
SNC	1032	0	0.00	-231	231	1.000
GA1	818	-214	-10.30	-421	-8	0.038
GA2	799	-233	-11.21	-431	-36	0.018
GA3	740	-292	-14.05	-491	-93	0.003
GA4	858	-174	-8.37	-372	23	0.077
GA5	802	-230	-11.07	-467	6	0.051
GA6	743	-289	-13.91	-505	-73	0.007
ACO1	789	-243	-11.69	-451	-34	0.020
ACO2	1003	-29	-1.40	-252	185	0.789
ACO3	734	-298	-14.34	-492	-103	0.002
ACO4	781	-251	-12.08	-504	1	0.047

The best results are more unevenly distributed between the two families of metaheuristic algorithms on the September subset. The very best result is achieved by ACO3, with a mean tardiness estimate of 734 minutes, a 14.3 % reduction, and a p-value of 0.002. It is closely followed by GA3 and GA6 with mean tardiness estimates of 740 minutes (14.1 % reduction) and 743 minutes (13.9 % reduction), respectively.

Figure 5.9: Mean tardiness September with forecasted processing times.



Most of the solutions produced by metaheuristic means offer statistically significant performance improvements on the September subset, with the notable exception of ACO2, which is no better than the benchmark with a p-value of 0.789.

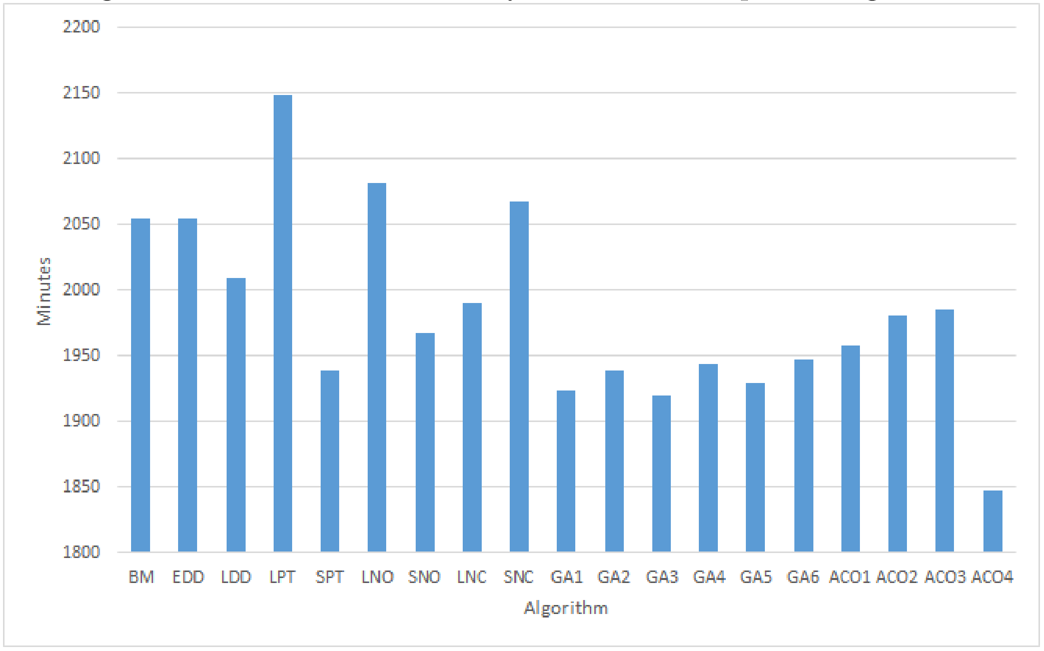
We now move over to the cross-validation experiments performed with historical data.

Table 5.4: Cross-validation with historical processing times

Algorithm	$\overline{T}$ **July**	$\overline{T}$ **August**	$\overline{T}$ **Sept.**	Δ	Δ (%)
BM	2055	2280	609	0.00	0.00
EDD	2055	2280	609	0.00	0.00
LDD	2009	2167	609	-53	-2.40
LPT	2148	2242	592	13	0.02
SPT	1939	2416	502	-29	-5.75
LNO	2082	2221	609	-11	-0.42
SNO	1968	2386	609	6	0.14
LNC	1990	2158	609	-62	-2.84
SNC	2067	2287	609	6	0.30
GA1	1924	2287	502	-77	-7.88
GA2	1939	2452	502	-17	-5.22
GA3	1920	2284	502	-79	-7.99
GA4	1944	2467	502	-10	-4.92
GA5	1930	2259	502	-84	-8.19
GA6	1947	2332	502	-54	-6.85
ACO1	1958	2321	502	-54	-6.83
ACO2	1981	2351	530	-27	-4.49
ACO3	1986	2314	502	-47	-6.48
ACO4	1847	2369	502	-75	-7.93

The cross-validation partition follows the same calendar months, but this time using data that was actually produced at Aarbakke in the respective months instead of forecasted numbers. All the 54 tests are presented in table 5.4. The historical data will always yield the same deterministic results as long as the same heuristic algorithm is used to schedule work orders. Because of this, the results are only calculated once, and it wouldn't make sense to talk about statistical metrics like confidence intervals and p-values. We evaluate the results simply by observing the values obtained in the cost function, the mean tardiness.

Figure 5.10: Mean tardiness July with historical processing times.

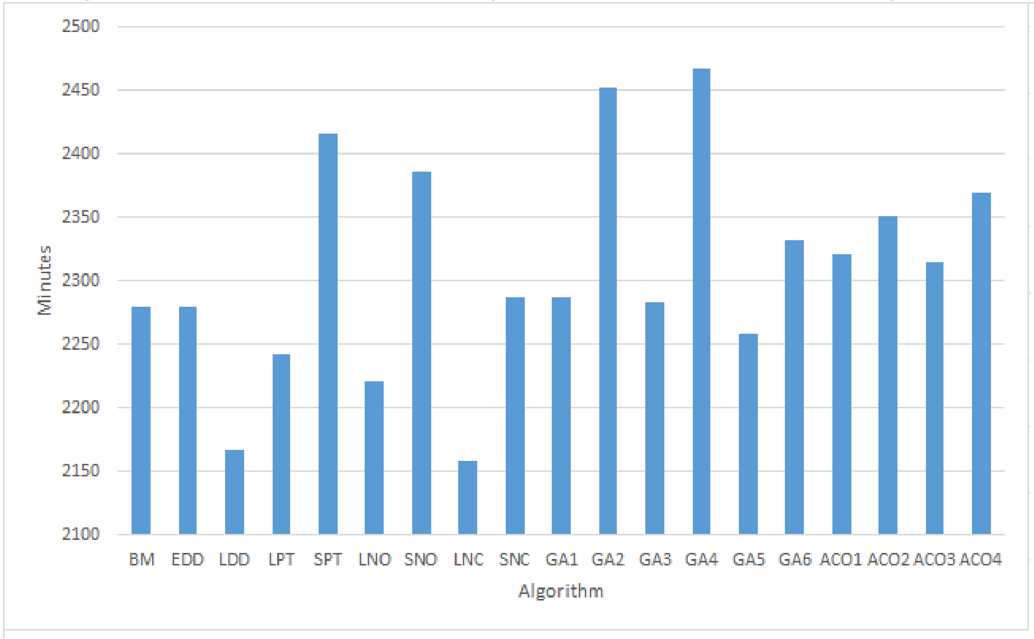


In July (figure 5.10), the benchmark incurred a mean tardiness of 2055 minutes, close to what was forecasted. The SPT rule, which has performed best among the simple dispatch rules, averaged out to 1939 minutes of tardiness.

The genetic algorithms came pretty close to the SPT rule in performance, ranging narrowly from 1920 minutes (GA3) to 1947 minutes (GA6). Results like this could partly be explained by the fact that the SPT rule is weighted positively in all six candidate models, although it is not the dominating rule in all of them (see table B.3).

The results from the experiments with historical data from August (figure 5.11) differ drastically from the other results. The benchmark incurred a mean tardiness 2280 minutes, higher than what was forecasted. What is more interesting, however, is that the performance of the other algorithms are practically reversed when compared with other experiments.

Figure 5.11: Mean tardiness August with historical processing times.

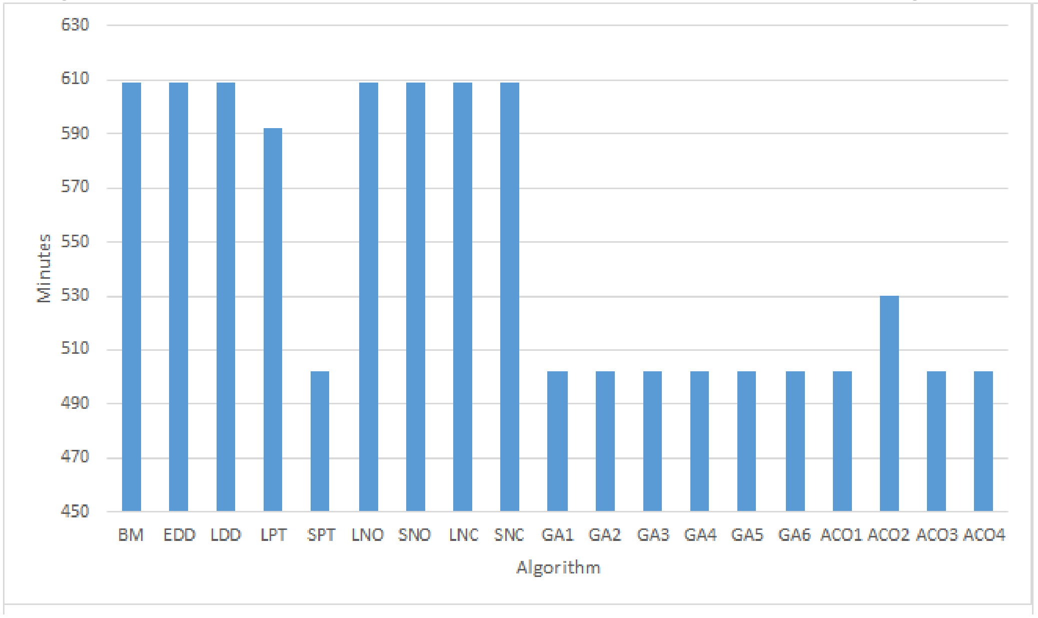


The otherwise poorly performing LNC and LDD rules achieved the best results with a mean tardiness of 2158 minutes and 2167 minutes, respectively. The metaheuristic algorithms show poor results, the worst being GA4 with a mean tardiness of 2467 minutes, and GA2 with a mean tardiness of 2452 minutes. GA5 is the only metaheuristic algorithm to improve slightly over the benchmark with a mean tardiness of 2259 minutes. These numbers show that the uncertainty associated with just one month of data can generate somewhat unexpected results.

Finally looking at historical data from September, we see an altogether different pattern. With fewer work orders and lower tardiness, the algorithms seem to have hit upon the same scheduling paths that produce either one of two results. The benchmark incurred a mean tardiness of 609 minutes, which is also true for most of the other simple dispatch rules.

The SPT rule, as well as almost all of the metaheuristic algorithms obtained a mean tardiness of 502 minutes, the only excetion being ACO2, which is close at 530 minutes.

Figure 5.12: Mean tardiness September with historical processing times.



5.3.1 Model evaluation

Comparing the overall results of the algorithms in all experiments, GA3 performed best, scoring on average 10.20 % better than the benchmark, 12.42 % with forecasted numbers and 7.99 % with historical numbers. The next three ranks also belong to genetic algorithms, with GA5, GA1 and GA6 all scoring more than 9 % better than the benchmark on average.

The genetic algorithms that were trained on deterministic numbers generally perform better than the ones trained on forecasted numbers, with the top three overall algorithms all belonging to the former group.

The fifth best algorithm is a deterministic dispatch rule, SPT. It achieves the lowest mean tardiness on the July dataset with forecasted processing times, but does not perform as well as the metaheuristic algorithms on the experiments with historical processing times. The impressive performance by this dispatch rule, which favours work orders with short processing times, could probably be explained by the limited time windows of the data subsets used for validation. This dispatch rule will naturally tend to give the heavy duty work orders low priority, which could simply serve to postpone the problems with tardiness until later. In the metaheuristic algorithms, the effect of this tendency is reduced by also training on longer time windows.

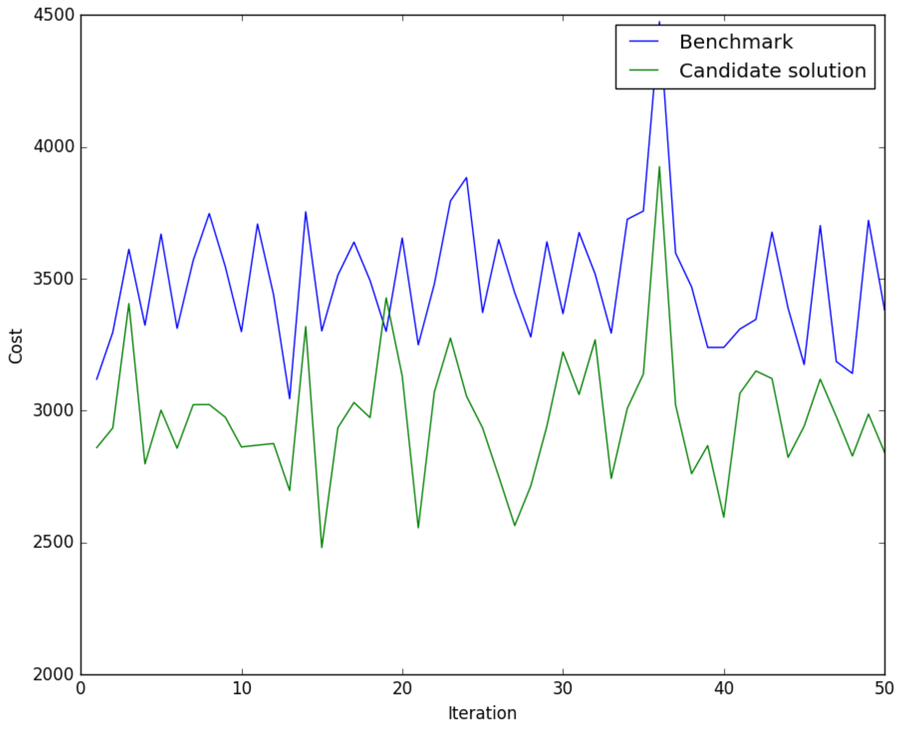
The ant colony optimization algorithms generally outperform the deterministic dispatch rules, but are more uneven than their genetic counterparts. The p-values from the Student's t-tests are well above what can be considered statistically significant for some of the experiments. ACO3 achieves the lowest mean tardiness on the September dataset with forecasted processing times while ACO4 achieves the lowest mean tardiness on the July dataset with historical processing times. The sporadic nature of these results suggest that the algorithms have stumbled upon good solutions by chance.

Based on the results and analysis, we select GA3 as the candidate model.

5.4 Hypothesis test

The candidate model (GA3) was first tested on 50 iterations of forecasted data on the October and November dataset in a two-sample hypothesis test against the benchmark algorithm. The benchmark achieved a mean tardiness of 3412 minutes while the candidate solution achieved a mean tardiness of 2913 minutes, a reduction of 14.6 %. The standard error in the hypothesis test was 39 minutes, leading to a Z-score of -12.4 and a very small p-value on a Z-test. Similarly, the p-value from the Student's t-test was smaller than 0.001 with three digits of precision. The mean tardiness using the candidate solution was between 577 minutes and 421 minutes lower than when using the benchmark with 95 % confidence. Figure 5.13 shows the consistency in which the candidate solution outperforms the benchmark on forecasted data.

Figure 5.13: Hypothesis test on forecasted October and November data.



The test with actual, historical data from October and November provided less dramatic, but still significant, results.

Table 5.5: Test on historical October and November data

Metric	**Benchmark**	**Candidate model**
Total tardiness (mins.)	613119	597283
Mean tardiness (mins.)	955	930
Percentage of benchmark	100%	97.4%
Improvement over BM	0.0%	2.6%
Finished work orders	642	642
Total machine delay	295058	295058

The candidate model achieved a 2.6 % reduction in mean tardiness. The number of finished work orders was the same for both algorithms. Thus, the candidate model also achieved a 2.6 % reduction in total tardiness. Around half of the tardiness (295058 minutes) was a result of delay that was incurred when processing operations on machines, while the rest was a result of operations waiting for machine access.

The two-sample hypothesis test of the candidate model satisfied the criteria for significance specified in 3.5.1. The null hypothesis can therefore be rejected.

5.5 Summary

In this chapter, we have presented and analysed the results of the machine learning experiments that were performed to optimize schedules on mean tardiness. The performance of 18 different scheduling algorithms based on both metaheuristics and deterministic dispatch rules were cross-validated against the benchmark by running a total of 108 experiments on both forecasted and historical data, using the forward chaining technique. The family of genetic algorithms outperformed all other algorithms. The candidate model (GA3) was selected based on cross-validation and evaluated in a statistical hypothesis test. It performed drastically better (14.6 %) than the benchmark when testing on forecasted data and significantly better (2.6 %) when testing on historical data. We ended the chapter by concluding that the null hypothesis can be rejected.

Chapter 6

Conclusion

The aim of this research was to reduce the problem with work order delays at Aarbakke by altering the algorithm that schedules work. At the start of chapter 3, we broke the problem into three questions that we wanted the research to answer. We have shown that it is indeed possible to improve on the current scheduling regime by introducing algorithms that exploit patterns in historical data. The research has highlighted the importance of reducing the search space and aiming for approximate solutions when dealing with the NP-hard problem of open shop scheduling. Among the models we evaluated, the best was a genetic algorithm that simplified the problem by optimizing the composition of a set of dispatch rules. It achieved significant improvements when testing on a subset of historical time series data containing processing times. It also generalized well when exposed to a wider range of scenarios by simulating processing times from statistical distributions.

With the adverse effects of late work orders and low cost of implementing the proposed solution, the alternative scheduling algorithm could prove beneficial for the manufacturing company to gain an extra edge in a competitive market. Furthermore, the approach could be applied to similar problems, where custom jobs are scheduled with high degrees of uncertainty.

6.1 Suggestions for future research

Future research could look into the effects of using other metaheuristic algorithms for tardiness optimization in the open shop scheduling problem, such as simulated annealing, tabu search or particle swarm optimization. Researchers could also implement Q-learning in combination with artificial neural networks.

It is likely that the results achieved in this particular problem domain could be improved by making the machine shop model more accurate. A good start would be to eliminate some of the delimitations listed in section 1.7. Another suggestion is to add more dispatch rules to the feature vector. One could also study the effects of changing the underlying scheduling algorithm by sidestepping the principle of just-in-time or using statistical estimates of processing times when determining how much machine capacity is needed in the schedule.

The candidate model should be subjected to further testing in a production environment to make a more precise assessment of its performance.

List of references

Amatriain, X. (2016). What are hyperparameters in machine learning? Message posted to https://www.quora.com/What-are-hyperparameters-in-machine-learning

Anand, E., & Panneerselvam, R. (2015). Literature Review of Open Shop Scheduling Problems. *Intelligent Information Management*. Retrieved from http://file.scirp.org/pdf/IIM_2015012815594820.pdf

Attiratanasunthron, N., & Fakcharoenphol, J. (2007). A running time analysis of an Ant Colony Optimization algorithm for shortest paths in directed acyclic graphs. *Information Processing Letters*. Retrieved from https://ai2-s2-pdfs.s3.amazonaws.com/fb04/ee79e626385cf8bfb9968f93cdf5c3091d16.pdf

Banks, J., Carson, J. S., Nelseon, B. L., & Nicol, D. M. (2009). *Discrete-Event System Simulation* (5th ed.). London, UK: Pearson.

Blum, C. (2005). Ant colony optimization: Introduction and recent trends. *Physics of Life Reviews 2*. Retrieved from https://www.ics.uci.edu/˜welling/teaching/271fall09/antcolonyopt.pdf

Bolufé-Röhler, A. (2014). What are the differences between heuristics and metaheuristics? Message posted to https://www.researchgate.net/post/What_are_the_differences_between _heuristics_and_metaheuristics

Can a computer generate a truly random number? (2011). Retrieved from http://engineering.mit.edu/ask/can-computer-generate-truly-random-number

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2002). *Introduction to Algorithms* (2nd ed.). Cambridge, MA: The MIT Press.

Cross-validation. (n.d.). In *Wikipedia.* Retrieved October 12, 2016, from https://en.wikipedia.org/wiki/Cross-validation_(statistics)

Dorigo, M. (1992). Optimization, Learning and Natural Algorithms (PhD thesis). Politecnico di Milano, Italy.

Gonzalez, T., & Sahni, S. (1976). Open Shop Scheduling to Minimize Finish Time. *Journal of the Association for Computing Machinery.* Retrieved from https://www.cise.ufl.edu/ sahni/papers/openShopFinishTime.pdf

Heaton, J. (2014). *Artificial Intelligence for Humans Volume 2: Nature-Inspired Algorithms.* St. Louis, MO: Heaton Research, Inc.

Karp, R. M., (1972). Reducibility among combinatorial problems. *Complexity of Computer Computations.* Retrieved from http://cgi.di.uoa.gr/ sgk/teaching/grad/handouts/karp.pdf

Kochenderfer, M. J. (2015). *Decision Making Under Uncertainty: Theory and Application.* Cambridge, MA: The MIT Press.

Krol, M. (2014). Is it accurate to describe Python as 'executable pseudocode'? Retrieved from https://www.quora.com/Is-it-accurate-to-describe-Python-as-executable-pseudocode

Lang, L. (2011). Job Shop Scheduling: The Secret to Getting On Time & Lead Times. Retrieved from http://www.fabricatingandmetalworking.com/2011/12/job-shop-scheduling-the-secret-to-getting-on-time-reducing-lead-times/

Lazy evaluation (n.d.). In *Wikipedia.* Retrieved November 8, 2016, https://en.wikipedia.org/wiki/Lazy_evaluation

Lukawiecki, R. (2016). Lecture on Practical Data Science. Personal collection of R. Lukawiecki, Project Botticelli, Dublin, Ireland.

Markov decision process (n.d.). In *Wikipedia*. Retrieved October 12, 2016, from https://en.wikipedia.org/wiki/Markovdecisionprocess

Naderi, B., Ghomi, S. M. T. F., Aminnayeri, M., & Zandieh, M. (2010). A study on open shop scheduling to minimise total tardiness. *International Journal of Production Research*. Retrieved from http://dx.doi.org/10.1080/00207543.2010.497174

Najafzadeh, M. (2014). What is the difference between test set and validation set? Message posted to http://stats.stackexchange.com/questions/19048/what-is-the-difference-between -test-set-and-validation-set

Ng, A. (2011). Machine Learning [Online course]. Retrieved from https://www.coursera.org/learn/machine-learning

NTB. (2016). Oljekrisen har ført til 25.000 færre arbeidsplasser. *Sysla*. Retrieved from http://sysla.no/2016/05/18/oljeenergi/oljekrisen-har-fort-til-25-000-faerre-arbeidsplasser_105227/

Raschka, S. (2015). *Python Machine Learning*. Birmingham, UK: Packt Publishing.

Rizzo, M. L. (2008). *Statistical Computing with R*. Boca Raton, FL: Chapman & Hall/CRC.

Shahzad, A., & Mebarki, N. (2010). Discovering dispatching rules for job shop scheduling problem through data mining. *8th International Conference of Modeling and Simulation*. Hammamet, Tunisia.

Sharma, P. (2015). Discrete-Event Simulation. *International Journal of Scientific & Technology Research*, 4. Retrieved from http://www.ijstr.org/final-print/apr2015/Discrete-event-Simulation.pdf

Sutton, R. S., & Barto, A. G. (2012). *Reinforcement Learning: An Introduction*. Cambridge, MA: The MIT Press.

Sörensen, K., & Glover, F. (2013). Metaheuristics. In *Scholarpedia.* Retrieved from http://www.scholarpedia.org/article/Metaheuristics

Taleb, N. (2012). *Antifragile: Things That Gain From Disorder.* New York, NY: Random House.

Tardiness (scheduling) (n.d.). In *Wikipedia.* Retrieved November 8, 2016, from https://en.wikipedia.org/wiki/Tardiness_(scheduling)

Walpole, R. E., Myers, R. H., Myers, S. L., & Ye, K. (2012). *Probability & Statistics For Engineers and Scientists* (9th ed.). Boston, MA: Pearson.

Williams, K. (2011). Using k-fold cross-validation for time-series model selection. Message posted to http://stats.stackexchange.com/questions/14099/using-k-fold-cross-validation-for-time-series-model-selection

Yang, X. (2011). Metaheuristic Optimization. In *Scholarpedia.* Retrieved from http://www.scholarpedia.org/article/Metaheuristic_Optimization

Appendices

Appendix A

Source code

This appendix lists the source code of some of the more important functions.

A.1 sample_from_density function (kde.py)

```
def sample_from_density(series, n=1):
        series = np.array(series)
        u = np.random.uniform(0,1,n)
        i = u*len(series)
        j = np.floor(i).astype(int)
        rand_samples = series[j]
        sigma = np.std(series)
        rand_kernel = np.random.normal(rand_samples, sigma/2, n)
        sample = np.floor(rand_kernel).astype(int)
        if(len(sample)==1):
                sample = sample[0]

        return sample
```

A.2 step function (ant.py)

```
def step(self):
    transition_probabilities = self.matrix[self.location]
    step = -1 # Intentionally crash the program if something is wrong.

    # Weighted random sampling:
    goal = np.random.uniform()*sum(transition_probabilities)
    cumulative = 0
    for i in range(0,len(transition_probabilities)):
        cumulative += transition_probabilities[i]
        if(cumulative>goal):
            step = i
            break

    self.location = step # Move ant to the new location.
    self.matrix[:,step] = 0
    # Ensure ant will not visit this place again.
    self.path.append(step)
```

A.3 deposit_pheromones function (colony.py)

```
def deposit_pheromones(self, algorithm):
    if algorithm == 'asrank':
        for ant in self.get_top_ants(int(self.colony_size/2)):
            for i in range(0,len(ant.path)-1):
                self.grid[ant.path[i],ant.path[i+1]] += \
                self.Q/ant.cost # Update rule

    if algorithm == 'elitist':
        # Best ant in this iteration deposits pheromones
        for i in range(0, len(self.best_path)-1):
            self.grid[self.best_path[i],self.best_path[i+1]] \
            += self.Q/self.global_best_path_cost
        # Global best ant deposits pheromones
        for i in range(0, len(self.global_best_path)-1):
            self.grid[self.global_best_path[i], \
            self.global_best_path[i+1]] \
            += self.Q/self.global_best_path_cost

    self.grid += self.pr
    # All edges receive a small amount of
    # pheromones to ensure some randomness
    self.grid = self.grid * self.inverse_identity
    # Remove self-loops to nodes
```

A.4 crossover function (genetic.py)

```
def crossover(mother, father):
    daughter = Phenotype()
    daughter.chromosomes = np.empty_like(mother.chromosomes)
    daughter.chromosomes[:] = mother.chromosomes

    son = Phenotype()
    son.chromosomes = np.empty_like(father.chromosomes)
    son.chromosomes[:] = father.chromosomes

    index = np.random.randint(len(mother.chromosomes))
    # Only swap one gene
    daughter.chromosomes[index], son.chromosomes[index] \
    = son.chromosomes[index], daughter.chromosomes[index]
    # Crossover

    return son, daughter
```

A.5 perturb_mutate function (genetic.py)

```
def perturb_mutate(self):
    perturb_amount = 0.1*max(abs(self.chromosomes))
    child = Phenotype()
    child.chromosomes = np.empty_like(self.chromosomes)
    child.chromosomes[:] = self.chromosomes
    child.chromosomes \
    += np.random.uniform(-perturb_amount, perturb_amount)

    return child
```

A.6 shuffle_mutate function (genetic.py)

```
def shuffle_mutate(self):
    child = Phenotype()
    child.chromosomes = np.empty_like(self.chromosomes)
    child.chromosomes[:] = self.chromosomes
    index1 = np.random.randint(len(self.chromosomes))
    index2 = index1
    while index2==index1:
        index2 = np.random.randint(len(self.chromosomes))
    child.chromosomes[index1], child.chromosomes[index2] \
    = child.chromosomes[index2], child.chromosomes[index1]

    return child
```

A.7 tournament_select function (genetic.py)

```
def tournament_select(self, rounds):
    champ = None
    for i in range(rounds):
        contender = \
        self.phenotypes[np.random.randint(len(self.phenotypes))]
        if not champ or contender.fitness < champ.fitness:
            champ = contender

    return champ
```

A.8 reverse_tournament function (genetic.py)

```
def reverse_tournament(self, rounds):
    victim = None
    for i in range(rounds):
        contender = \
        self.phenotypes[np.random.randint(len(self.phenotypes))]
        if not victim or contender.fitness > victim.fitness:
            victim = contender

    return victim
```

A.9 capacity_controller function (machine_shop.py)

```
def capacity_controller(self, env):
    while True:
        if self.capacities[env.now] == 0:
            if self.access_controller.capacity>1:
                self.access_controller = \
                simpy.Resource(self.env, capacity=1)
                # Set machine capacity to 1

            if self.VERBOSITY>=4:
                print("[", env.now, "]: Deactivating", self.name, \
                "capacity:", self.access_controller.capacity)
                self.active = False
        else:
            if not self.active:
                self.active = True
                cap = max(self.capacities[env.now],
                self.granularity)
                self.access_controller = \
                simpy.Resource(self.env, capacity=cap)
                # Alter machine capacity

            if self.VERBOSITY>=4:
                print("[", env.now, "]: Activating", self.name, \
                "with capacity", self.access_controller.capacity)

        yield self.env.timeout(self.granularity)
```

Appendix B

Tables

Table B.1: Overview of Aarbakke's database

Table	**Description**
Customer	A list of Aarbakke's customers
CustomerOrder	Single orders by customer
CustomerOrderLine	Customer orders split into manufacturing lines
Employee	Employees at Aarbakke
ExternalOrderPriority	Work order priorities generated by machine learning
ImportRef	An index of all the data import times by datetime
Part	A list of parts used in manufacturing
PartStock	The inventory of available parts
PlanningGroup	A list of resources/machines

Table B.2: Overview of Aarbakke's database (continued)

ProcurementOrder	Single procurement orders
ProcurementOrderLine	Procurement orders split into manufacturing lines
ProcurementOrderLineTransaction	Changes in procurement order lines
ProductStructure	Overviews of operations needed to manufacture products
ProductStructureOperation	Planned time and cost of manufacturing operations
ProductStructureOperationMaterial	Planned material use of manufacturing operations
ResourceCapacity	Calendar of schedules for resources
StockTransaction	Changes in inventory
Supplier	A list of Aarbakke's vendors
SWBWorkOrder	Dumped simulation data on work orders
SWBWorkOrderOperation	Dumped simulation data on single operations
WorkOrder	Logged work order data
WorkOrderOperation	Logged single operation data
WorkOrderOperationMaterial	Logged material usage in operations
WorkOrderOperationTransaction	Logged changes in single operations

Table B.3: Feature representation of composite dispatch rules.

Algorithm	**LNC**	**LNO**	**LDD**	**LPT**
EDD	0	0	-1	0
LDD	0	0	1	0
LPT	0	0	0	1
SPT	0	0	0	-1
LNO	0	1	0	0
SNO	0	-1	0	0
LNC	1	0	0	0
SNC	-1	0	0	0
GA1	0.03546921	0.41632695	-0.6430961	-0.57688912
GA2	0.08808435	0.80908903	0.85178828	-0.71719603
GA3	0.01251136	0.11287817	-0.81582536	-0.8730170
GA4	0.16040067	-0.06205546	0.80051283	-0.21281097
GA5	0.51273901	0.60069229	-0.4295663	-0.94386541
GA6	-0.3709734	0.12062362	-0.67238265	-0.49071046