Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Economics and Management
Dept. of Industrial Economics and Technology Management

Henrik Irgens Gravdal
Jørgen Weidemann

An Adaptive Genetic Algorithm for the Hybrid Flexible Flowshop Scheduling Problem With Sequence-Dependent Set-up Times

Master's thesis in Industrial Economics and Technology Management
Supervisor: Henrik Andersson
Co-supervisor: Anders Gullhav and Maryna Waszak
June 2022

NTNU
Kunnskap for en bedre verden

Henrik Irgens Gravdal
Jørgen Weidemann

An Adaptive Genetic Algorithm for the Hybrid Flexible Flowshop Scheduling Problem With Sequence-Dependent Set-up Times

Master's thesis in Industrial Economics and Technology Management
Supervisor: Henrik Andersson
Co-supervisor: Anders Gullhav and Maryna Waszak
June 2022

Norwegian University of Science and Technology
Faculty of Economics and Management
Dept. of Industrial Economics and Technology Management

NTNU
Norwegian University of
Science and Technology

Preface

This master's thesis is part of our Master of Science at the Norwegian University of Science and Technology, Department of Industrial Economics and Technology Management. The thesis is written as a part of the course TIØ4905 Managerial Economics and Operations Research and builds on the work of our project's thesis in Gravdal and Weidemann (2021).

We want to give our sincere thanks to our supervisor, Professor Henrik Andersson and co-supervisor, Associate Professor Anders Gullhav, from The Department of Industrial Economics and Technology Management. Also, a special thanks to our co-supervisor, Maryna Waszak from The Department of Software and Service Innovation at SINTEF. We are grateful for the insights and feedback you have provided us with while researching, coding, and writing this thesis.

Henrik Irgens Gravdal and Jørgen Weidemann

June 8, 2022

Abstract

Over the last decades, manufacturing businesses have seen mass outsourcing of production to low-cost countries. Furthermore, larger manufacturers have enjoined reduced costs per produced unit due to their streamlined factories and production volume. The small production facilities in high-cost countries are losing the war on efficiency and need to improve at a lower cost than their larger competitors. One promising area of improvement is within production planning. This area concerns the generation of effective production schedules in a reasonable time on affordable hardware.

This thesis examines a variant of the *flowshop scheduling problem* (FSP), inspired by the production environment of a small Norwegian manufacturer, Haugstad Furniture Factory (Haugstad). The machines in the factory are grouped by function into stages, and all products, also called *jobs*, follow the same flow through these stages. At least one stage contains more than one machine, making it a *hybrid* flowshop. The machines of a given stage are assumed to be identical. Not all products need to be processed in every stage, which also makes it *flexible*. Moreover, machines need to be prepared before processing each job, and the set-up time depends on the machine's previously processed product. This final extension is called *sequence-dependent set-up times*. Set-up is also non-anticipatory, meaning that it cannot start before the product arrives physically at the machine. This combination of flowshop extensions is called the hybrid flexible flowshop scheduling problem with sequence-dependent set-up times (HFFSP SDST). The objective is to find production schedules minimising the total time it takes to produce a known quantity of products, also known as *makespan*.

Out of 172 research papers on the hybrid flowshop scheduling problem (HFSP) in the last decade, only three papers use genetic algorithms (GA) to solve the HFFSP SDST with minimising makespan as the objective. We propose a new GA that finds production schedules for problem instances containing up to 120 jobs, eight stages, and four machines in each stage in less than two minutes. The GA consists of several operators yet to be tested for the FSP. The *Best Cost Block Crossover* (BCBX) is a crossover operator adapted from the vehicle routing problem. The *Greedy Construction Heuristic* (GCH) is a new construction heuristic used to create the initial population. Also, an adaptive choice of crossovers is introduced.

The GA performs better than all benchmark algorithms across all sizes of problem instances, both in the average relative distance from the best solution found by any method and the number of best solutions found. Furthermore, BCBX is the best crossover for smaller problem instances, GCH is the dominant initialisation method, and the adaptive choice of crossover operator shows superior to each operator's single performance. In conclusion, the GA can be said to provide high-quality production schedules for production environments of different characteristics, including those of Haugstad.

Sammendrag

Over de siste tiårene har produksjonsbedrifter flyttet store deler av sin produksjon til lavkostnadsland. I tillegg har store produsenter benyttet seg av strømlinjeformede produksjonsinstallasjoner og høye produksjonsvolum for å senke enhetskostnadene sine. Mindre produsenter i høykostnadsland sliter med å få ned sine enhetskostnader og må konkurrere med lavere budsjett enn sine større konkurrenter. Et lovende område for kostnadseffektivisering for slike bedrifter er datadrevet produksjonsplanlegging. Dette innebærer metoder for å effektivt lage gode produksjonsplaner ved hjelp av rimelige datamaskiner.

Dette studiet undersøker en variant av *flowshop planleggingsproblemet*, inspirert av produksjonsmiljøet til en liten norsk produsent, Haugstad Møbel (Haugstad). Maskinene i fabrikken er gruppert etter funksjonalitet i ulike *trinn* og alle *produkter* følger samme flyt gjennom trinnene. Minst ett trinn har mer enn en maskin tilgjengelig, noe som gjør produksjonsmiljøet til en *hybrid* flowshop. Videre antas det at alle maskinene er identiske. Ikke alle produkter behøver å prosesseres i alle trinn. Dette gjør at produksjonsmiljøet betegnes som *fleksibelt*. Maskinene må klargjøres før de kan prosessere et nytt produkt, og tiden det tar avhenger også av hvilket produkt som sist ble prosessert. Dette kalles *sekvensavhengig klargjøringstid*. Klargjøringen er ikke mulig å starte før produktet som skal prosesseres ankommer maskinen. Denne kombinasjonen av utvidelser kalles et hybrid fleksibelt flowshop problem med sekvensavhengige klargjøringstider. Målet er å finne produksjonsplaner som minimerer den totale tiden det tar å produsere en kjent mengde produkter.

Av 172 vitenskapelige artikler publisert om det hybride flowshop planleggingsproblemet det siste tiåret, er det bare tre som bruker genetiske algoritmer til å løse den ovennevnte varianten og med minimering av total produksjonstid som mål. I denne oppgaven introduserer vi en ny genetisk algoritme som på under to minutter finner gode produksjonsplaner for opptil 120 produkter, åtte trinn, og fire maskiner i hvert trinn. Dette oppnås ved å benytte operatorer som aldri har blitt brukt for å løse problemet før. *Best Cost Block Crossover* (BCBX) er en krysningsoperator adaptert fra et ruteplanleggingsproblem. *Greedy Construction Heuristic* (GCH) er en ny konstruksjonsheuristikk for å initialisere den første populasjonen. I tillegg brukes et adaptivt valg av krysningsoperatorer.

Den genetiske algoritmen gjør det bedre enn tre implementerte referansealgoritmer for alle problemstørrelser, både i relativt avvik fra beste løsning og antall beste løsninger funnet. Videre er BCBX den beste krysningsoperatoren for små probleminstanser, GCH er den dominerende metoden for å initialisere populasjonen, og det adaptive valget av krysningsoperatorer gjør det bedre enn noen enkelt operator for seg selv. Oppsummert er den nye genetiske algoritmen effektiv til å finne gode produksjonsplaner til fabrikker av ulike karakteristikker, inkludert karakteristikker som kjennetegner Haugstad.

Table of Contents

List of Tables

List of Figures

1 Introduction **1**

2 Literature Review **4**
2.1 Production Scheduling 4
2.2 Literature Search Strategy 5
2.3 The Flowshop Scheduling Problem 6
2.4 Solution Methods 8
2.5 Contribution 11

3 Problem Description **13**

4 Mathematical Model **15**
4.1 Modelling Assumptions 15
4.2 Mathematical Formulation 15

5 Solution Methods **19**
5.1 Solution Representation 19
5.2 Makespan Calculation 20
5.2.1 First-In, First-Out 21
5.2.2 First-Completion 22
5.3 Construction Heuristic 22
5.3.1 NEH 23
5.3.2 Modified Dynamic Dispatching Rule 24
5.4 Improvement Heuristics 24
5.4.1 Iterated Greedy 24
5.4.2 Genetic Algorithm 25

6 Genetic Algorithm **27**
6.1 Objective 27
6.2 Solution Representation 28
6.3 Initialisation 28
6.4 Selection 28

6.5 Crossover . . . 29
6.5.1 Similar Job Order Crossover and Similar Block Order Crossover . . . 30
6.5.2 Best Cost Block Crossover . . . 31
6.5.3 Partially Mapped Crossover . . . 32
6.5.4 Adaptive Choice of Crossover Operators . . . 33
6.6 Mutation . . . 34
6.6.1 Shift . . . 34
6.6.2 Swap . . . 34
6.6.3 Reversal . . . 35
6.6.4 Greedy . . . 35
6.7 Local Search . . . 36
6.8 Generational Scheme . . . 36
6.8.1 Generational Genetic Algorithm . . . 36
6.8.2 Steady-State Genetic Algorithm . . . 36
6.9 Crowding . . . 37
6.9.1 Implicit Crowding . . . 38
6.9.2 Explicit Crowding . . . 39
6.9.3 Replacement Rules . . . 39
6.10 Replacement . . . 41

7 Computational Study **42**
7.1 Problem Instances . . . 43
7.2 Tuning of Genetic Algorithm . . . 43
7.2.1 Methodology . . . 43
7.2.2 Tuning . . . 45
7.3 Tuning of Iterated Greedy . . . 55
7.4 Convergence Tests . . . 56
7.5 Performance Tests . . . 57
7.5.1 Comparison with Mathematical Model . . . 57
7.5.2 Comparing Solution Methods . . . 59

8 Concluding Remarks **62**

9 Future Work **64**

Bibliography **66**

Appendix **i**
A Tuning of Genetic Algorithm . . . i
B Tuning of Iterated Greedy . . . xvii

List of Tables

2.1 Literature search keywords 5
2.2 Literature classification 12

7.1 Technical specifications for testing of non-exact methods 42
7.2 Preliminary parameter values 44
7.3 Parameter selection based on tuning 55
7.4 Technical specifications for running the mathematical model 58
7.5 Mathematical model and adaptive genetic algorithm comparison 58
7.6 Comparison of best solutions found 60

List of Figures

5.1 Search space and solution space connection 20
5.2 Example of the FIFO makespan calculation in the first stage 21
5.3 Example of the second phase of NEH 23
5.4 Flowchart of the genetic algorithm 26

6.1 Example of tournament selection 29
6.2 Example of the SJOX crossover operator 30
6.3 Example of the BCBX crossover operators 31
6.4 Example of the PMX crossover operator 32
6.5 Example of the shift mutation operator 34
6.6 Example of the swap mutation operator 35
6.7 Example of the reversal mutation operator 35
6.8 Example of the greedy mutation operator 35
6.9 Example of the exact match similarity metric 37
6.10 Example of the deviation distances similarity metric 38
6.11 Example of explicit crowding 39

7.1 Example of first-completion makespan calculation procedure assignment 45
7.2 Results from testing generational schemes 46
7.3 Results from testing initialisations 47
7.4 Results of testing convergence with random and GCH initialisations 48
7.5 Results of testing population sizes 49
7.6 Results of testing crossover choices 50
7.7 Results of testing mutation operator choices 51
7.8 Results of including a local search in the genetic algorithm 52
7.9 Results of testing crowding 53
7.10 Results of testing replacement schemes 54
7.11 Results of tuning iterated greedy 55
7.12 Results of convergence tests on the adaptive genetic algorithm and iterated greedy 56
7.13 Results of mathematical model converging 59
7.14 Results of testing the adaptive genetic algorithm, iterated greedy, MDDR, and NEH 59
7.15 Results of testing the adaptive genetic algorithm, iterated greedy, MDDR, and NEH for different sized problem instances 61

1 Extended results of testing makespan calculation procedures ii
2 Extended results of testing generational schemes iii
3 Extended results of testing initialisations iv
4 Extended results of testing population sizes v
5 Extended results of testing tournament sizes vi
6 Extended results of testing the choice of crossover operators vii
7 Extended results of testing Q-learning crossover parameters viii
8 Extended results of testing the choice of mutation operators ix
9 Extended results of testing mutation probabilities x
10 Extended results of including a local search in the adaptive genetic algorithm . . xi
11 Extended results of testing deviation distance crowding xii
12 Extended results of testing exact match crowding xiii
13 Extended results of testing crowding xiv
14 Extended results of testing replacement schemes xv
15 Extended results of testing replacement rates xvi
16 Extended results of tuning iterated greedy xvii

Chapter 1

Introduction

Over the last decades, industrialised countries have moved an ever-increasing portion of their production to previously non-industrialised countries. The wealth of the former is soaring to new heights, and with it, the service sector is growing and putting pressure on wages. With the cost of machines and other means of production decreasing and wages increasing, the difference between producing in high-cost contra low-cost countries keeps growing.

The large manufacturers are streamlining their production, using entire production facilities to produce only a few products. Producing only a handful of products in a facility saves time on logistics and set-up. This is possible because of their scale and the demand for their products. Smaller manufacturers are losing out in this war on costs. They cannot reduce costs by specialising entire facilities, as the demand for their products is insufficient. They inherently experience more downtime on machines due to the need for switching production more often and special-purpose machines only used for a few products. To stay profitable, they need to charge higher prices for their products. If this trend continues, they will eventually be uncompetitive, as there is a limit to the consumer's willingness to pay premium prices for similar products.

Norway is losing manufacturing jobs every year (SSB, 2020) and small local manufacturers are looking for ways to reduce costs without having to reduce their differentiation (Gravdal and Weidemann, 2021). They usually differentiate themselves by quality and location, providing jobs where they operate. In parallel, the Norwegian Government is convinced that many of the production facilities are about to move back from low-cost to high-cost countries (Ministry of Trade Industry and Fisheries, 2017). The rationale is that superior technology will reduce the need for manual labour and increase throughput. These new technologies include advances in hardware, which allows for the adaption of automatised machinery with sensor-based decision systems, and software for control and planning. In short, new machines can speed up production and software can increase their utilisation.

Machines are necessary to manufacture products, and hence all manufacturers have them. Upgrades in machinery provide a tangible impact that is often denoted by an exact increase in throughput. However, increases in efficiency through advanced planning software are less tangible, as its effect is not constant nor well defined. Therefore, it is seldom given the same priority. Production planning is, in fact, often still a human domain.

In the fall of 2021, we met with Haugstad Furniture Factory (Haugstad). Haugstad is a small and local furniture factory in Norway currently investing heavily in machinery fitted with sensors to keep track of production. The factory is looking into software for controlling and understanding production but has not yet adapted software for production planning.

Advances in technology require substantial investments, and there are reasons to believe scale will be a deciding factor for what the companies can afford. Large manufacturing corporations can utilise their scale to buy customised machines and develop software customised to their needs. Furthermore, they can acquire state of the art hardware to run their powerful software. Smaller businesses will struggle to finance the same investments. If they invest heavily in machinery, there will be little left to invest in other technologies. They have to rely on standardised software and slower hardware, making them susceptible to falling even further behind. This thesis aims to decrease the deficit small manufacturers struggle with within production planning.

The term *production planning* refers to deciding which machine in a production line should perform which task and at what time. Most often, this is done in relation to maximising the throughput of the manufacturers in some form. Production planning is a complex and time-consuming task. As it is still often handled manually, sub-optimal plans are made, and the task requires specialised personnel. Research has shown substantial potential gains by adapting more sophisticated and automated approaches (Ruiz and Maroto, 2006).

The desire to create efficient production schedules has existed for a long time. It was introduced along with the first factories in the late 1700s and renewed when electricity enabled the factories to be even larger (Herrmann, 2006). One of the earliest attempts was made by Gantt (1903), with the *Gantt chart* as a graphical tool to display and analyse production schedules. However, he introduced no principles for creating nor improving the plans themselves. In 1919, Taylor argued that the planning of production and execution should be separate matters and considered the scheduling problems formally for the first time.

At the time of Taylor and Gantt's publications, all production facilities were considered to function like workshops, where each employee created a product from start to finish. It was not until Johnson (1954) introduced the flowshop that factories with machines organised according to function and a unidirectional product flow were considered with regards to formal planning methods (Herrmann, 2006). There has been a tremendous amount of papers on the topic in the decades since. To begin with, they primarily introduced assignment rules or principles to design good schedules, and with the emergence of computer science, more sophisticated algorithms were introduced.

Haugstad is the main inspiration for this thesis, and the problem is modelled according to their production environment. We aim to introduce a solution method to a production scheduling problem sufficiently similar to the one Haugstad faces every day. There are a few criteria to consider for a scheduling procedure to be relevant to them. Although they manage to invest substantial capital in new machinery, it is doubtful that they can invest the same amount in pure production planning software and hardware. The methods for finding production schedules need to run on affordable hardware and produce schedules sufficiently fast. Finding the optimal schedule is worthless if it takes days or even hours to find it.

In an effort to increase the effectiveness and quality of Haugstad's production planning, we examine the *hybrid flexible flowshop scheduling problem with sequence-dependent set-up times* (HFFSP SDST). In this flowshop variant, the machines are grouped in stages according to their function. All products follow the same unidirectional flow through the stages. Not all products need to be acted upon by all functions, but since a single machine can process a range of products, they must be calibrated before most jobs. In the case of Haugstad, 10 000 to 20 000 unique components are manufactured each week, making this a challenging problem to solve manually.

We have previously tested exact methods, which are too slow, even on sophisticated hardware (Gravdal and Weidemann, 2021). Researchers have implemented several non-exact methods to solve this and similar problems. However, the publications on the HFFSP SDST are few. Genetic algorithms (GAs) are the most common solution methods for similar extensions of the flowshop scheduling problem (FSP) and have shown promising results. However, few studies into the HFFSP SDST test GAs as a solution method.

This thesis is organised as follows: Chapter 2 introduces literature about the problem described above and methods others have used to solve it. A formal description of the problem is given in Chapter 3 and based on it, a mathematical model is formulated and introduced in Chapter 4. In Chapter 5, all solution methods implemented and tested in this thesis are described, and the description of our GA is extended in Chapter 6. All tests and results are described in Chapter 7. Finally, Chapter 8 sums up the findings of this study, and Chapter 9 looks at a few areas that could be interesting to examine further in the future.

Chapter 2

Literature Review

This chapter introduces relevant literature in the field of *scheduling*. Scheduling is defined as "the allocation of one or more resources over time to perform a collection of tasks", with a focus on doing so efficiently (Chen et al., 1998). The main concepts of the large field of scheduling are introduced with a focus on planning in industry settings and production scheduling in Section 2.1. Section 2.2 describes the overall search procedure and relevant search words. Then, the focus is narrowed to the flowshop scheduling problem (FSP) in Section 2.3. Finally, Section 2.4 review the solution methods used to solve complex versions of the FSP. Note that Section 2.1 and the beginning of Section 2.3 are revised versions of the literature review of Gravdal and Weidemann (2021).

2.1 Production Scheduling

Production scheduling concerns the planning of production items, sequencing, and time allocation of operations to complete the items in time (Stoop and Wiers, 1996). Thus, it can be said to lie at the very heart of the performance of manufacturing organisations. In the last decades, efficient scheduling has become vital due to increased demands for quality, flexibility, and order lead times. An example of such an emerging concept is mass customisation. It aims to provide customised products or services through flexible processes in high volumes and at reasonably low costs (Suzić et al., 2018). It makes for a very complex scheduling problem, but scheduling is still typically a human domain. This motivates the development of models and algorithms for tackling such complex scheduling problems.

In the field of production planning, there are two main types of production systems (Fan et al., 2018): *single-stage* and *multi-stage*. A *stage* is a collection of related *machines* that can process *jobs*. A job is any physical product produced during manufacturing. The single-stage system is characterised by jobs that only need processing in one stage, and hence one machine. The multi-stage production system consists of several stages, characterised by jobs needing to be

processed by many machines.

Within the multi-stage production system, there are three distinct production subsystems: *open shop scheduling*, *jobshop scheduling*, and *flowshop scheduling*. In open shop scheduling, jobs have to undergo a set of operations in different stages, but the order to carry them out is not predetermined (Hosseinabadi et al., 2019). The real-life equivalent could be a car repair shop where there makes no difference if the headlights are changed before the engine is examined. Jobshop scheduling extends open shop scheduling by having a predetermined order that operations are performed in (Fan et al., 2018). This resembles a high-end furniture workshop where some furniture is sanded before it is coated, while others are coated before sanding, and the order is crucial. In flowshop scheduling, the jobshop scheduling is extended by making the flow of products unidirectional (Ruiz and Vázquez-Rodríguez, 2010). This is exemplified through a traditional mass production facility where all products follow the same assembly line. This thesis examines the flowshop scheduling problem (FSP) and ways to construct its production schedule.

2.2 Literature Search Strategy

A systematic literature search is carried out to find papers on the FSP and the relevant parts of its extensions and solution methods. In this search, terminology established and structured by Pinedo (2012) on scheduling is used to discover relevant literature. These categories' terms are combined to find publications capturing relevant aspects of the problem. All searches are conducted on Google Scholar. Table 2.1 presents an overview of the terms used in combination to find relevant literature. Note that not all publications agree on the names of certain extensions, while the meaning and changes to the FSP are generally agreed upon. An example is the flexible extension, also denoted as the stage skip extension. Although they have different names, both extend the FSP by some jobs not needing processing in all stages. The search resulted in numerous articles filtered again for relevance in titles and abstracts. The most relevant ones are selected and included in the subsequent sections.

Table 2.1: Keywords used in the literature search.

General	**Problem extensions**	**Solution method**	**Objective**
Flowshop	Flexible	Heuristic	Makespan
Scheduling	Hybrid	Meta-heuristic	Completion time
Planning	Sequence dependent setup times	Genetic algorithm	

2.3 The Flowshop Scheduling Problem

The first paper on the FSP was published in 1954. In the following 50 years, more than 1200 papers were published on this topic (Gupta and Stafford, 2006), and yet more since then. A reason for its popularity may be that flowshops arise in most standardised manufacturing and assembly facilities where a set of jobs has to undergo a series of operations (Pinedo, 2012). The versatility and applicability of flowshops have been shown for a wide variety of areas such as textile, electronics, automobile manufacturing, and chemical industries (González-Neira et al., 2017). Numerous variants of the FSP have been formulated for such applications.

For a manufacturing facility to classify as a flowshop, all jobs must visit the stages in the same order, and a job can never be processed more than once in any stage. In its simplest form, the flowshop scheduling problem consists of determining the uninterrupted flow of jobs through multiple machines in series (Gupta and Stafford, 2006). More formally, the flowshop scheduling problem is defined by Gupta and Stafford (2006) as follows:

> Given a set of jobs $\mathcal{N} = 1, \ldots, n$ to be processed on a set of machines $\mathcal{M} = 1, \ldots, m$ in the same technological order; the processing time of job i on machine j being p_{ij} $(i = 1, 2, \ldots, n; j = 1, 2, \ldots, m)$; it is desired to find the schedule in which these n jobs should be processed on each of the m machines to minimise a well defined measure of production cost.

The flowshop scheduling problem was first introduced by Johnson (1954) in a two-machine environment. He studied a simplification named the *permutation flowshop scheduling problem* (PFSP), which assumes the equal ordering of jobs in every stage. In this paper, Johnson proposes an algorithm for optimally scheduling jobs to minimise the total elapsed time for the entire operation in $\mathcal{O}(n \log n)$ time. He proves that the algorithm solves the PFSP to optimality for two stages, with one machine in each. However, already for a flowshop with three or more stages, the problem is NP-complete (Garey et al., 1976). It implies that no algorithm can solve the problem to optimality in polynomial time in the number of jobs. Rossit et al. (2018) state that the optimal objective value to a PFSP, as opposed to a comparable FSP, is anywhere between 0.5% higher to 40% higher, depending on the size of the problem instance and the objective function. So, the extra complexity of the FSP might be worthwhile. In fact, more than 65% of the papers on non-PFSP have been published after 2006, showing a substantial popularity increase lately (Rossit et al., 2018).

Although well-studied in literature, the classical flowshop formulation has limited application in real production environments. In the last couple of decades, there has been a trend to more realistic modelling (Wang, 2005) and a great many extensions have been introduced. Colak and Keskin (2022) identifies 26 categories of extensions from 172 studies, excluding an "other" category of 38 papers with yet more extensions. The 38 uncategorised papers indicate a long tail of extensions relevant to the industry yet to receive attention among researchers.

The *hybrid* extension might be the most important and common extension, with 172 articles studying it between 2010-2020 (Colak and Keskin, 2022). In a hybrid flowshop, at least one stage consists of parallel machines where each job visits exactly one machine. There are three different configurations to parallel machines in a stage: (1) the machines are identical, (2) the machines operate with different speeds, and (3) the machines are unrelated, meaning the speed of each machine is dependent on the job (Pinedo, 2012). A prevalent way to increase throughput in production environments is to install several similar machines, so one could argue that the extension increases the realism of the problem. This increase in realism comes at the cost of increased complexity. To be specific, even the case with two stages, one with a single machine and one with two, is NP-hard, according to Fattahi et al. (2013). Note that some publications denote the hybrid extension as flexible instead.

Another common extension is the *sequence-dependent set-up times* (SDST) extension. This extension is combined with the hybrid extension in 51 of the 172 articles Colak and Keskin (2022) review. SDST make the set-up time for each job dependent on the preceding job and the stage in which it is to be processed. A well-known justification for this extension is made by Ruiz and Maroto (2006) in relation to ceramic tile manufacturing. The set-up is highly dependent on the final product visiting the machine because of different moulds, temperatures, and glaze. A final variation for set-up, is the *anticipatory* distinction. It allows set-up to start before jobs physically arrive at the machine (Colak and Keskin, 2022).

The three next most common extensions Colak and Keskin (2022) finds, are *batch processing*, *flexibility*, and *machine eligibility*. Batch processing groups similar jobs to be acted on as a collection. A batch can then be handled as a single job. Doing so can reduce the complexity of the problem if the batches are fixed early in the search. Flexibility allows jobs to skip stages; for example, not all tiles need all layers of glaze. Machine eligibility extends the hybrid extension, prohibiting certain machines from processing certain jobs. Besides the one exception of batch processing in heuristics, all these extensions increase the complexity of the problem.

The most common assumption in flowshop scheduling is that buffers are unlimited. Colak and Keskin (2022) find that only 5% of modern papers on hybrid flowshops leave out this assumption. However, in the case of limited buffers between successive machines, *blocking* may occur (Pinedo, 2012). Blocking happens when a buffer is full such that the upstream machine is prevented from releasing its current job. Although realistic, it is challenging to solve and is strongly NP-hard (Papadimitriou and Kanellakis, 1980).

Although there are many objectives to solve all combinations of flowshops for, Rossit et al. (2018) find that 73% of the papers they review on the non-PFSP use a form of completion time objective. The reason can be that it is tightly coupled to the efficiency of the production schedule (Pinedo, 2012). Among them, minimisation of *makespan*, the total completion time of all jobs, dominates as the objective of 53% the total papers, against 17% other completion time objectives. For the rest of this literature study, it is assumed that makespan is used as objective unless otherwise specified. Other objectives consider *due-dates*, finishing jobs in time, and *cost*, often related to energy or raw materials. They represent 8% and 10% of the papers.

2.4 Solution Methods

This section covers the development of solution methods to solve variations of the FSP. Exact methods are covered first, along with complexity. Then, problem-specific heuristics are reviewed. GAs are the most used meta-heuristics for the FSP, so it is extensively covered before the focus shifts to an iterated local search called Iterated Greedy.

Johnson (1954) introduces an algorithm for scheduling jobs to optimality in a two-machine environment in $\mathcal{O}(n \log n)$ time. For a set of jobs $\mathcal{N} = 1, ..., n$ where the processing times are known for each job on both machines, the rule works as follows: (1) Select the job which has the shortest processing time in either of the machines. (2) If the shortest processing time is for the first machine, place the job first in the order; otherwise, place it last. (3) Repeat for all remaining jobs until all jobs are placed in order. This algorithm is referred to as Johnson's rule. Johnson also proves that the algorithm works for a particular case of three-machine environments. Still, as shown in Garey et al. (1976), the general case of any production environment with three machines or more is NP-complete. There have been few successful enquiries into exact algorithms after this.

Ignall and Schrage (1965) use *branch and bound* (B&B) to solve the PFSP and find optimal solutions with ten jobs in a three machine environment with makespan as the objective. Along with the development in B&B methods, there has been a revolution in computational power (Moore, 1965) and today, B&B methods can solve problems of 500 jobs and 20 machines in 20 minutes for PFSPs (Gmys et al., 2020). Furthermore, while exact algorithms have received little attention lately, mathematical models in the form of *mixed integer programs* (MIPs) are used to solve smaller instances for a great variety of extensions and to benchmark approximation methods (Colak and Keskin, 2022). Ruiz et al. (2008) were among the first to mathematically model and implement a hybrid flexible flowshop in terms of a MIP to reduce the gap between the flowshops in the literature and real production environments. Many have since followed.

The search for heuristics has often been focused on the PFSP because of its simplicity and solutions having some merit for the FSP as well. Campbell et al. (1970) seek to expand the use of Johnson's rule to m-machine environments. They introduce the CDS heuristic, which clusters the m original machines into two virtual machines and repeatedly uses Johnson's rule to produce $m-1$ schedules. The authors initially made CDS not require a computer to find a good schedule, and since, better heuristics have been introduced. NEH, proposed by Nawaz et al. (1983), is a two-phase heuristic sorting jobs according to processing times and then inserting them greedily into the job permutation. Although many other problem-specific construction and improvement heuristics have been proposed (Dannenbring, 1977; Ho and Chang, 1991; Suliman, 2000), Ruiz and Maroto (2005) finds that NEH is by far the dominant one.

A genetic algorithm (GA) is a meta-heuristic which differs from others by keeping several solutions and combining them in a fashion inspired by evolution (Ruiz and Maroto, 2006). It is commonly used to approximate the Pareto front as the solutions often are spread among several local optima. Today, it is the most common meta-heuristic to use when solving the FSP. It is

used in 16% of the studies Rossit et al. (2018) reviews. This was not always the case. Before the 2000s, many believed GAs to be inferior to the likes of tabu search (TS) and simulated annealing (SA) (Murata et al., 1996). GAs did, however, show promise when solving similar problems to the FSP, like the travelling salesman problem (TSP), and when the available computational power increased substantially. Here follows a study into the recent developments of GAs used to solve a variety of FSPs. For a full description of the GA, see Section 5.4.2 and Chapter 6.

Murata et al. (1996) use a simple GA to solve the classic PFSP. So do Ruiz et al. (2006), but with a significantly more complex algorithm. In parallel, Ruiz and Maroto (2006) explore the effectiveness of a GA on the hybrid, SDST, and machine eligibility extensions, while still keeping the permutation simplification. Several researchers follow by applying GAs to complex versions of the FSP: Gómez-Gasquet et al. (2012) use an agent-based genetic algorithm to solve the FSP with both the hybrid, flexible, and SDST extensions without any simplifications like permutation. Today this is the standard complexity of FSPs to solve with GAs (Sioud et al., 2013, 2014; Farahmand-Mehr et al., 2014; Yu et al., 2018). Furthermore, eight out of the nine most relevant articles for this thesis use makespan as the objective and also the fitness criteria as evident by Table 2.2.

As for all meta-heuristics, the encoding of solutions dictates the search space. Many different encodings have been used for the FSP. There are some main categories, where the simplest one is the permutation simplification as introduced in Section 2.3. For this representation, a solution contains the production order of all jobs, and it is the same in every stage. This representation was popular around the 2000s (Murata et al., 1996; Ruiz and Maroto, 2006; Ruiz et al., 2006). Combined with the hybrid extension, Ruiz and Maroto (2006) use assignment rules to determine the machine allocation in each stage. Gómez-Gasquet et al. (2012) and Sioud et al. (2013, 2014) also use only one permutation of all jobs but apply assignment rules for the processing order in all stages after the first stage and in all machines. Some researchers opt for encoding all job orders in all stages and assigning every job to machines (Engin et al., 2011; Farahmand-Mehr et al., 2014).

For initialisation of populations, the standard approach is to use entirely random job permutations, as seen in Murata et al. (1996), Engin et al. (2011), and Farahmand-Mehr et al. (2014). Since this is likely to start the algorithm without any high-quality solutions, Ruiz and Maroto (2006) modify the NEH heuristic also to solve extensions of FSP and use it to ensure initialising one good solution. Ruiz et al. (2006) and Gómez-Gasquet et al. (2012) use simple greedy algorithms to create several better solutions, while Yu et al. (2018) use two sorting algorithms to create two high-quality solutions, with the rest entirely random.

Crossovers have received immense attention, and the simple operators used 20 years ago are now discarded for more complex alternatives. Murata et al. (1996) adopt operators that were standard at the time. They are based on one- and two-point crossovers, keeping blocks from one parent and the order of the rest of the jobs from the other. Ruiz and Maroto (2006) introduce modified operators comparing the parents and keeping jobs that occur in the same position for both parents, and using one- and two-point crossover afterwards. Engin et al. (2011) use similar

crossovers but also introduce operators to copy blocks from a parent and insert it in a different position for a child. To keep as much of a parent as possible, Sioud et al. (2013) use operators that copy whole blocks from a parent and use only the order for the remaining blocks from the other. They also introduce an operator with a greedy choice, approximating the makespan of several options and returning the best one. Finally, Gómez-Gasquet et al. (2012) use previously introduced crossovers and an adaptive method to choose which to use in every instance. The population consists of several agent solutions, an agent part and a solution part. The agent contains logic to select a crossover operator and learn from the choice.

Most papers set a low probability of mutations and implement either a swap of two or three jobs (Gómez-Gasquet et al., 2012; Sioud et al., 2013, 2014; Farahmand-Mehr et al., 2014), a shift of jobs (Ruiz and Maroto, 2006), or both (Murata et al., 1996; Ruiz et al., 2006; Yu et al., 2018). Most apply a mutation operator to a child that a crossover operator has just produced, but Sioud et al. (2013, 2014) use only a crossover operator or mutation operator to create the next generation.

A final thing that all GAs need to incorporate is a generational scheme, defining the transition from one generation to the next. The classic approach is to create all-new individuals in every generation (Murata et al., 1996; Engin et al., 2011; Sioud et al., 2013, 2014; Farahmand-Mehr et al., 2014). Elitism is a possible addition, involving carrying a certain percentage of the best solutions from the previous generation to the next, with Farahmand-Mehr et al. (2014) keeping as much as 20% of the best solutions. Farahmand-Mehr et al. (2014) also use immigration, adding randomly generated solutions to the population in every generation to induce diversity. Another generational scheme is *steady-state*, generating a small number of children in every generation and only adding them to the population if they are better than some parent. This generational scheme is also used by several authors (Ruiz and Maroto, 2006; Ruiz et al., 2006; Yu et al., 2018).

To enhance performance and avoid preemptive convergence in a local optimum, Ruiz and Maroto (2006) introduce a replacement mechanism. Suppose the algorithm does not improve over some generations. In that case, the 80% least fit individuals are replaced with a mix of mutations of the best solutions and new randomly created solutions. Yu et al. (2018) use the same idea but with an extra mutation operator changing more significant parts of the solution. Another module to improve the GAs are local searches, making them *memetic* algorithms. Murata et al. (1996) argue this is the only way to make GAs better than TS and SA. Ruiz and Maroto (2006); Yu et al. (2018) also use local searches to improve the performance.

Ruiz and Stützle (2007) argue that many of the complicated methods of SA, TS, and GA have yet to show better results than simple local search meta-heuristics. This is tested again in Naderi et al. (2010) with the same result. More straightforward methods have fewer parameters to tune, are quick to implement, and are transferable to other problems and hence extensions. Iterated Greedy (IG) shows real promise in this area, and there are examples of it outperforming more complex alternatives. It is extended by several researchers, for example, Ozsoydan and Sağir (2021).

Several other meta-heuristics are also used to solve versions of the FSP. Rossit et al. (2018) finds that after GA, TS is most applied, as it is the primary solution method in 12% of the papers on hybrid flowshops. Then follow SA and ant colony optimisation, each the topic of 8% of the papers. Other meta-heuristics account for only 6%.

2.5 Contribution

Of all the 172 articles on the HFSP between 2010 and 2020, only 20 include the hybrid, flexible, and sequence-dependent set-up times extensions (Colak and Keskin, 2022). Of these, 13 use makespan as their objective, and three use GA for their primary solution method. We have yet to find any paper implementing crossovers and mutations using domain knowledge to the extent this thesis does, nor implementations of crowding. Furthermore, we introduce a new randomised NEH heuristic and adopt Q-learning to choose crossover adaptively. The complete classifications of reviewed literature and overlap with this thesis are found in Table 2.2.

Table 2.2: Classification of most relevant literature for genetic algorithms for the HFFSP SDST. All classifications covered by this thesis is coloured green.

Article	Problem	Objective	Encoding	Initiation	Selection	Crossover	Mutation	Crowding	Replacement	Generational Scheme	Other
Murata et al. (1996)	PFSP	Makespan	Permutation	Random	Ranking	OP, TP, PBX	Swap and shift	No	No	Generational	Include local search
Ruiz and Maroto (2006)	HPFSP SDST Machine eligibility	Makespan	Permutation	NEH and random	Ranking and tournament	OP, TP, PMX, OX, UOB, SJOX, SBOX, SJ2OX, SB2OX	Shift	No	Random and mutate	Generational and steady-state	Include local search
Ruiz et al. (2006)	PFSP	Makespan	Permutation	NEH, random, and randomised NEH	Ranking and tournament	OP, TP, PMX, OX, UOB, SJOX, SBOX, SJ2OX, SB2OX	Swap and shift	No	Random, mutate, and randomised NEH	Generational and steady-state	Include local search
Engin et al. (2011)	HFSMT	Makespan	Direct encoding	Random	Ranking	PBX, PMX, OX, CX, LOX, OBX	Neighbourhood search	No	No	Steady-state	
Gómez-Gasquet et al. (2012)	HFFSP SDST	Makespan	Job order in first stage	Greedy algorithm	-	SB2OX, NCO	Swap	No	No	Generational	Agent-solutions as chromosomes
Sioud et al. (2013)	HFFSP SDST	Makespan	Job order in first stage	NEH and random	Tournament	RMPX, ARMPX, LJMPX, MPBOX, UOBX	Swap	No	No	Generational with elitism	Either crossover or mutation
Sioud et al. (2014)	HFFSP SDST	Makespan	Job order in first stage	NEH and random	Tournament	RMPX, ARMPX, LJMPX, MPBOX	Swap	No	No	Generational with elitism	Either crossover or mutation
Farahmand-Mehr et al. (2014)	HFSP SDST	Makespan	Direct encoding	Random	Random	OX	Swap	No	No	Generational with elitism	Use immigration
Yu et al. (2018)	HFSP SDST	Tardiness	Priority queue	Sorting heuristics and random	Roulette wheel and tournament	OP, TP, PMX, SB2OX, OBX	Swap and shift	No	Random, mutate, and large mutation	Steady-state	Include local search
This thesis	HFFSP SDST	Makespan	Job order in first stage	NEH, random, and randomised NEH	Tournament	PMX, SJOX, SBOX, BCBX	Swap, shift, reversal and greedy	Yes	Random, mutate best solutions, randomised NEH	Generational and steady-state	Adaptive crossover choice, include local search

Chapter 3

Problem Description

This chapter describes the hybrid flexible flowshop scheduling problem with sequence-dependent set-up times (HFFSP SDST) in detail. The problem is mainly associated with factories having a streamlined production environment and usually concerns production over a limited time horizon. More specifically, operational planning on a daily to weekly basis. It is a deterministic problem where the objective is to minimise the total production time, the completion of the last product, and indirectly maximise the factory's output.

The production environment of a factory consists of *stages*. In every stage, there is a known number of *machines*. Due to the hybrid extension, at least one of the stages consists of more than one machine. The machines in each stage are identical. The stages are ordered according to the flow of the production line, and *jobs* visit the stages in the given order. Jobs are physical products produced by the factory. Although all jobs go through the stages in the same predefined order, each job does not necessarily have to be processed in every stage, as the flexible extension dictates. That is, a job may skip one or more stages. A job visits each stage at most once, and only one machine can process the job in each stage. Accordingly, every job has a predefined path through the stages where the stages may differ, but the order is the same.

Machines process jobs, and each machine can only process one job at a time. Before processing, the machines need to be set up for that particular job. The *set-up time* depends on the current job and the previously processed job and is the same for all machines in the same stage, meaning they have sequence-dependent set-up times with identical machines. In addition, set-up is *non-anticipatory*, meaning it cannot start before the job which is to be processed arrives at the machine. During set-up, the machine cannot process any job. The machines are identical, so processing times are equal in all machines in the same stage for a given job. Moreover, once a machine starts processing a job, it cannot be interrupted.

Even though the jobs are physical objects and the factory is of finite size, the *buffers* between stages are assumed to be non-restrictive. This entails there is no limit on the number of jobs that can wait between two adjacent stages.

A solution to the problem determines what machines are to process which jobs and the internal job order in every machine. For the solution to be feasible, every job must be processed in every stage required. The objective is to minimise *makespan*; the total time it takes to process all jobs. This is equivalent to minimising the completion time of the job with the latest completion time.

Chapter 4

Mathematical Model

In this chapter, we present a mathematical model for the hybrid flexible flowshop scheduling problem with sequence-dependent set-up times (HFFSP SDST) as defined in Chapter 3. In Section 4.1, all modelling assumptions are stated and Section 4.2 introduces the mathematical model with its sets, parameters, and constraints.

4.1 Modelling Assumptions

The mathematical model assumes production can be left at any time and resumed. This assumption implies that no notions of days or weeks are defined, and thus no bookkeeping constraints are needed to connect those. Accordingly, the mathematical model schedules the entire production period without breaks. In addition, we introduce the term *machine run* to denote the position a job has in the queue of a machine. If a job is assigned to machine run 4 of machine 2 in stage 3, the job is to be the fourth job to visit machine 2 in stage 3.

4.2 Mathematical Formulation

The following mathematical model is inspired by the model introduced in Defersha and Chen (2012). The model is originally developed for the HFFSP SDST with batch processing, anticipatory set-up times and machine eligibility extensions. The notion of batches and machine eligibility constraints are removed to make the model fit the HFFSP SDST described in Chapter 3. In addition, all set-up times are made non-anticipatory. Finally, a procedure for calculating values of M in the Big M-constraints is proposed.

Indices

$i,\ l$	stages
$m,\ k$	machines
$n,\ p$	jobs
$r,\ u$	machine runs

Sets

$\mathcal{I}$	set of stages
$\mathcal{M}_i$	set of machines at stage i
$\mathcal{N}$	set of jobs
$\mathcal{E}_n$	set of pairs of stages (l, i), where stage l precedes stage i, for job n
$\mathcal{R}_{im}$	set of machine runs for machine m at stage i

Parameters

T_{in}	processing time of job n at stage i
T^S_{ipn}	set-up time in stage i for job n succeeding job p
P_{in}	1 if product type n needs processing on stage i, 0 otherwise
M	upper bound on total production time

Decision Variables

t^N_{in}	completion time of job n in stage i
t^M_{imr}	completion time of machine run r on machine m in stage i
c_{max}	makespan, completion time of the last operation

$$x_{imrn} = \begin{cases} 1 & \text{if machine run } r \text{ on machine } m \text{ in stage } i \text{ processes job } n, \\ 0 & \text{otherwise.} \end{cases}$$

$$z_{imr} = \begin{cases} 1 & \text{if machine run } r \text{ on machine } m \text{ at stage } i \text{ has been assigned to a job,} \\ 0 & \text{otherwise.} \end{cases}$$

Objective Function and Constraints

$$\min z = c_{max} \tag{4.1}$$

s.t.

$$t^M_{imr} \geq t^N_{in} + M(x_{imrn} - 1) \qquad i \in \mathcal{I}, m \in \mathcal{M}_i, r \in \mathcal{R}_{im}, n \in \mathcal{N} \tag{4.2}$$

$$t^M_{imr} \leq t^N_{in} + M(1 - x_{imrn}) \qquad i \in \mathcal{I}, m \in \mathcal{M}_i, r \in \mathcal{R}_{im}, n \in \mathcal{N} \tag{4.3}$$

$$t^M_{im1} \geq T_{in} + T^S_{i0n} + M(x_{im1n} - 1) \qquad i \in \mathcal{I}, m \in \mathcal{M}_i, n \in \mathcal{N} \tag{4.4}$$

$$t^M_{imr} - t^M_{im(r-1)} \geq T_{in} + T^S_{ipn} + M(x_{im(r-1)p} + x_{imrn} - 2) \qquad i \in \mathcal{I}, m \in \mathcal{M}_i, r \in \mathcal{R}_{im}, n, p \in \mathcal{N} | r > 1 \tag{4.5}$$

$$t^M_{im1} - t^M_{lku} \geq T_{in} + T^S_{i0n} + M(x_{lkun} + x_{im1n} - 2) \qquad i, l \in \mathcal{I}, k \in \mathcal{M}_l, m \in \mathcal{M}_i, u \in \mathcal{R}_{lk}, n \in \mathcal{N} | (l, i) \in \mathcal{E}_n \tag{4.6}$$

$$t^M_{imr} - t^M_{lku} \geq T_{in} + T^S_{ipn} + M(x_{lkun} + x_{im(r-1)p} + x_{imrn} - 3) \qquad i, l \in \mathcal{I}, m \in \mathcal{M}_i, k \in \mathcal{M}_l, r \in \mathcal{R}_{im}, u \in \mathcal{R}_{lk}, n, p \in \mathcal{N} | (l, i) \in \mathcal{E}_n, r > 1 \tag{4.7}$$

$$\sum_{n \in \mathcal{N}} x_{imrn} = z_{imr} \qquad i \in \mathcal{I}, m \in \mathcal{M}_i, r \in \mathcal{R}_{im} \tag{4.8}$$

$$z_{imr} \leq z_{im(r-1)} \qquad i \in \mathcal{I}, m \in \mathcal{M}_i, r \in \mathcal{R}_{im} | r > 1 \tag{4.9}$$

$$\sum_{m \in \mathcal{M}_i} \sum_{r \in \mathcal{R}_{im}} x_{imrn} = P_{in} \qquad i \in \mathcal{I}, n \in \mathcal{N} \tag{4.10}$$

$$c_{max} \geq t^N_{in} \qquad i \in \mathcal{I}, n \in \mathcal{N} \tag{4.11}$$

$$x_{imrn} \in \{0, 1\} \qquad i \in \mathcal{I}, m \in \mathcal{M}_i, r \in \mathcal{R}_{im}, n \in \mathcal{N} \tag{4.12}$$

$$z_{imr} \in \{0, 1\} \qquad i \in \mathcal{I}, m \in \mathcal{M}_i, r \in \mathcal{R}_{im}, \tag{4.13}$$

$$t^N_{inj} \geq 0 \qquad i \in \mathcal{I}, n \in \mathcal{N}, j \in \mathcal{J}_n \tag{4.14}$$

$$t^M_{imr} \geq 0 \qquad i \in \mathcal{I}, m \in \mathcal{M}_i, r \in \mathcal{R}_{im} \tag{4.15}$$

The objective function (4.1) specifies the minimisation of makespan. Constraints (4.2) and (4.3) define the relationships between the t^N_{in}, t^M_{imr}, and x_{imrn} variables. That is, when a job is scheduled for a run in a machine in a given stage (x_{imrn}), the time for the completion of the job (t^N_{in}) and the time the machine is free again (t^M_{imr}) have to equal. Constraints (4.4) make sure the first job processed in every machine does not start before the set-up is completed and constraints (4.5) ensure that the succeeding jobs aren't processed before set-up, nor that the machines process more than one job at a time.

Constraints (4.6) disallow the first job to arrive at any machine to start setting up before it finishes in the previous stage or processing before the set-up is complete. Constraints (4.7) have the same effect on all succeeding jobs and ensure sequence-dependent set-up times are complied with.

The x_{imrn} and z_{imr} variables are connected by constraints (4.8) such that every machine can at most process one job at a time. Constraints (4.9) squeeze machine runs, so machine run r is only used if machine run $r-1$ is used. Constraints (4.10) ensure that every job that needs processing in a given stage is processed. Constraints (4.11) make the variable for total makespan equal to the finishing time of the last job. Finally, constraints (4.12) and (4.13) state the binary variables, whereas (4.14) and (4.15) are non-negativity constraints.

Big M-constraints are applied in constraints (4.2) through (4.7). The value of M is related to the time variables t^N_{in} and t^M_{imr}. A conservative value for M can be calculated when all jobs go through only one machine in each stage, and the worst case sequence-dependent set-up times are realised. Equation (4.16) gives the worst set-up time for every job in every stage, and equation (4.17) finds the value for M as the sum of processing time and set-up time for all jobs in all stages.

$$\overline{T}^S_{in} = \max_{p \in \mathcal{N}} \left\{ T^S_{ipn} \right\} \qquad i \in \mathcal{I}, n \in \mathcal{N} \tag{4.16}$$

$$M = \sum_{i \in \mathcal{I}} \sum_{n \in \mathcal{N}} (T_{in} + \overline{T}^S_{in}) \tag{4.17}$$

The set of machine runs, $\mathcal{R}_{im}$, is a set with a size that is difficult to define optimally. It dictates how many jobs a machine can process. The larger the set, the more decision variables, t^M_{imr}, x_{imrn}, and z_{imr}, are created. The smaller the set $\mathcal{R}_{im}$ is, the fewer constraints and variables are defined, and the faster the search for the optimal solution. However, it is important not to make the optimal solution infeasible by restricting $\mathcal{R}_{im}$ too much. The theoretical maximum of machine runs on any single machine is the total number of jobs. However, as not all jobs visit every stage, the number may be reduced to the number of jobs that has to visit a particular stage. Also, with more than one machine in a stage, this theoretical maximum is likely an overestimation. Without any way of telling for certain the optimal number of machine runs for a machine, the theoretical maximum is used.

Chapter 5

Solution Methods

This thesis aims to solve the hybrid flexible flowshop scheduling problem with sequence-dependent set-up times (HFFSP SDST) and introduces a new genetic algorithm (GA) to accomplish it. In addition, some already known algorithms are implemented as benchmarks. However, before introducing the different solution methods, Section 5.1 looks at what constitutes a solution and how it affects the search and solution spaces. Section 5.2 continues by bridging the gap between encoded solutions and their real-life interpretations. Then, Section 5.3 explains how reasonably good feasible solutions can be constructed, while Section 5.4 introduces two methods for improvement of feasible solutions.

5.1 Solution Representation

This section explores two ways of representing a solution to the HFFSP SDST and motivates which is more advantageous for this study. There are two common approaches: (1) A solution can be a complete representation of the production schedule. That is, the solution contains the allocation of jobs to machines in every stage and the internal orderings of jobs in machines. (2) A solution can be a partial representation of a production schedule. In this case, assignment rules are needed to transform the partial schedule into a complete one. One example of a partial schedule is the order of jobs in the first stage. This representation is the most common partial representation in flowshop scheduling and is the one considered here.

The solution representation defines how the solution and search spaces are tied together. The solution space contains all possible complete production schedules and is defined by the problem instance alone. The search space consists of all possible production schedules an algorithm can create. It is defined by the problem instance and the solution representation. All candidates in the search space map to exactly one solution in the solution space, as illustrated in Figure 5.1. This mapping may be one-to-one, implicating that there is a unique solution in the solution space for each candidate in the search space. Alternatively, it may be many-to-one, implicating

that several candidates in the search space map to the same solution in the solution space.

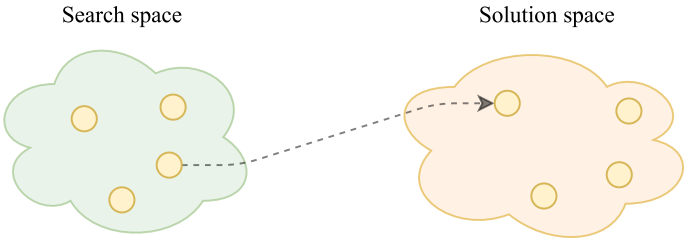


Figure 5.1: The connection between search space and solution space.

With a complete solution representation, there is a one-to-one mapping where every solution in the solution space is mapped to by a candidate from the search space. It makes the spaces the same size, and often it induces a vast search space. There are as many solutions for each stage as there are permutations of jobs. This amounts to $(|\mathcal{N}|!)^{|\mathcal{I}|}$ possible solutions, where $\mathcal{N}$ is the set of jobs and $\mathcal{I}$ the set of stages. Note that this underestimates the search space as the assignment of jobs to machines is not considered. For a problem instance with 120 jobs and eight stages, there are at least $4.0 * 10^{1590}$ solutions.

For the partial solution representation, the search space is significantly reduced and smaller than the solution space. The mapping might be one-to-one or many-to-one, and there will be solutions in the solution space that cannot be found through the mapping from the search space. With only a permutation of the jobs in the first stage, there are $|\mathcal{N}|!$ different representations in the search space. For the same problem instance as above, this amounts to $6.7 * 10^{198}$ representations to search. This significant decrease contributes to a faster search. However, this also likely removes several high-quality solutions. Nonetheless, previous research has shown that the removal of high-quality solutions is by far outweighed by the search space being easier to navigate (Naderi et al., 2010). Therefore, this study uses partial representations of production schedules and assignment rules to create the complete schedules. The assignment rules are described in the following section.

5.2 Makespan Calculation

Two different procedures using assignment rules to define the mapping between the search space of partial solution representations and the solution space are presented. They are used to determine the order of jobs in subsequent stages and assign them to machines. This section explains how this is done, thus providing the schedule's makespan. The first procedure is a *first-in, first-out* (FIFO) based approach. It schedules jobs into stages according to the order they finished in the previous stage. The second method we refer to as *first-completion*. This method uses the order of the jobs in the given permutation in the first stage, but in later stages prioritises scheduling jobs according to which may be completed first.

5.2.1 First-In, First-Out

The partial solution representation provides the order for the jobs to be processed in the first stage. The remaining decisions are to allocate each job to a machine in every stage. The allocation of jobs to machines is equal in the first stage for both makespan calculation methods. This is done by scheduling each job, in order, to the machine that can finish it with the lowest completion time. The completion time is calculated as in Equation 5.1. To get the completion time, the time the machine becomes available after processing the previous job is added with the sequence-dependent set-up time for the job. Finally, the processing time of the job is added. The completion time of each job is recorded in the production schedule and used for scheduling jobs in the succeeding stages.

$$Completion\_time = Machine\_available\_time + Setup\_time + Processing\_time \tag{5.1}$$

An example with seven jobs and two machines is illustrated in Figure 5.2 to clarify how the FIFO scheduling process works in the first stage. *Machine 1* is scheduled to process job 2 before job 4, while *Machine 2* is scheduled to process job 5 before job 6. The next job to be assigned to a machine from the ordered list that the solution provides is job 3. Currently, *Machine 1* is done processing its last job before *Machine 2* is. However, the sequence-dependent set-up time for job 3 following job 4 is higher than for job 3 following job 6. This results in *Machine 2* being able to finish processing job 3 first out of the two machines. Therefore, job 3 is assigned to *Machine 2*. Then, the new available time for *Machine 2* becomes equal to that of the completion time for job 3 in this machine, whereas it remains unchanged for *Machine 1*. Then, the process is repeated with the next job from the ordered list of remaining jobs, namely job 1.

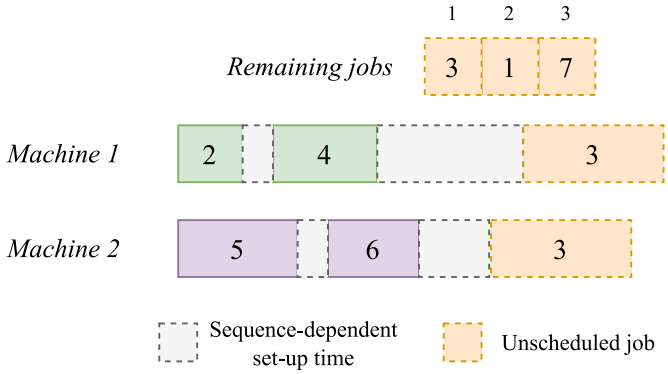


Figure 5.2: An example situation that could arise in the first stage while using FIFO.

For FIFO, the order of jobs in succeeding stages is given by the order completed in the previous stage. Ties in completion times are given to the job that first started processed in the previous stage. For stages after the first one, Equation 5.1 is altered to ensure no job starts before it finishes in the previous stage, nor before the machine is ready. This is reflected in the first term of Equation 5.2. The remaining terms are equal to those of Equation 5.1. When all jobs are

assigned to machines in all stages required, makespan is calculated as the maximum completion time of any job.

$$Completion\_time = \max(Machine\_available\_time, Completion\_time\_from\_previous\_stage) + Setup\_time + Processing\_time \quad (5.2)$$

5.2.2 First-Completion

FIFO is still used for scheduling jobs in the first stage. However, in the succeeding stages, the strategy is rather different. Instead of following the order of jobs finishing in the previous stage, all jobs are considered at once. The job that can be completed first by any of the machines in the current stage calculated by Equation 5.2 is assigned to the corresponding machine. This process continues until all jobs are assigned to machines in every stage. To clarify, we again look at the example in Figure 5.2. Instead of considering only job 3, jobs 1 and 7 are also considered for both machines. The job-machine combination that has the lowest completion time is selected. Ties in completion time are given to the job completed earliest in the previous stage.

Both makespan calculation procedures have their advantages and drawbacks. The FIFO procedure is faster as only one job is considered at a time. Moreover, in the FIFO procedure, fewer changes are made to the order of jobs since the order of completion from the previous stage is used. Thus, the initial order from the search algorithm has a greater chance of manifesting itself through the stages. More significant changes are likely made to the order of jobs in each stage for the first-completion procedure. This procedure finds quite different solutions than the FIFO procedure. For SDST problems, in particular, this may be beneficial as it allows for choosing job sequences with low set-up times. However, the changes in one stage may not be the best in the longer term, as this procedure does not consider what happens in future stages. In order to find out which procedure performs the best, both are tested in Chapter 7.

5.3 Construction Heuristic

This section introduces two construction heuristics for two reasons. First, they are straightforward to implement and serve as benchmarks for the more complicated methods. If the more complicated methods cannot produce better solutions, there is no reason to complicate matters. Second, one of them provides an excellent starting point for the search by the improvement heuristics covered in Section 5.4.

5.3.1 NEH

As mentioned in Chapter 2, NEH is considered state of the art among construction heuristics for the permutation flowshop scheduling problem (Ruiz and Maroto, 2006). It has been used extensively and later expanded for various extensions of the flowshop scheduling problem (FSP), including the HFFSP SDST.

The expanded NEH algorithm consists of two main phases (Ruiz and Maroto, 2006): (1) For every job, the sum of processing times for the job in all stages is found, and the list of jobs is ordered in descending order of total processing times. (2) The jobs are inserted into a new list in the order found in the previous stage. To decide a job's placement in the new list, makespan is calculated for every possible position for the job. The one yielding the lowest makespan is chosen. This entails that the first job has only one possible position to be put in, the second has two, and so on. The runtime complexity of NEH is $\mathcal{O}(|\mathcal{N}|^3 \sum_{i=1}^{|\mathcal{I}|} m_i)$, where $\mathcal{N}$ is the set of jobs and m_i the number of machines in stage $i \in \mathcal{I}$ (Fernandez-Viagas and Framinan, 2015). Note that NEH operates with an encoded solution representation of only the jobs' order in the first stage. Ties go to the lowest index position. An example is provided in Figure 5.3 to clarify how the second phase of the algorithm works.

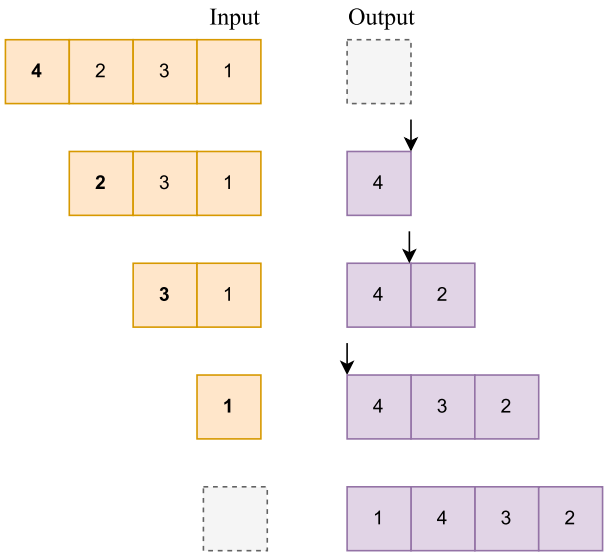


Figure 5.3: Visualisation of NEHs second phase in an environment with four jobs. The next job to be placed in the output list is highlighted (bold) in the input permutation. The arrows in each row indicate the best found position for the current job.

Figure 5.3 illustrates the second phase of NEH. Assume four jobs are ordered by descending total processing time in the first phase: [4,2,3,1]. Job 4 can only be placed in the one position available in the new list. After calculating the makespan of [2,4] and [4,2], the latter is found to have the lowest, placing job 2 after job 4. While inserting job 3, [4,3,2] is found to have a smaller makespan than [3,4,2] and [4,2,3]. Finally, job 1 is placed in the first position by the same arguments. This simple procedure has several times shown to produce high-quality

schedules for different FSPs.

5.3.2 Modified Dynamic Dispatching Rule

The *modified dynamic dispatching rule* (MDDR) heuristic, introduced by Naderi et al. (2010), can be described by using the logic from the makespan calculation procedure first-completion, introduced in Section 5.2.2, in every stage. The first stage uses Equation 5.1 for all combinations of jobs and machines and schedules the sequence of jobs with the lower completion time. Ties are decided in favour of the lower job number. All succeeding stages use Equation 5.2 in the same manner as in the first stage.

With a runtime complexity of $\mathcal{O}(|\mathcal{N}|^2 \sum_{i=1}^{|\mathcal{I}|} m_i)$, MDDR is faster than NEH as the number of jobs increase (Naderi et al., 2010). Here, $\mathcal{N}$ refers to the set of jobs and m_i the number of machines in stage $i \in \mathcal{I}$. Moreover, while NEH has shown to provide good solutions for smaller instances of the HFFSP SDST, MDDR performs better as the instances grow in size, especially for large SDST problems. Since it is a bit different, fast, and has shown to provide good solutions in the literature, it is considered a good benchmark for more complicated methods on the larger problem instances.

5.4 Improvement Heuristics

Having introduced two methods to create initial solutions, we now introduce two iteration-based meta-heuristics for improving existing solutions. Iterated Greedy is a well known, highly randomised local search. GAs, on the other hand, are complex methods made to search more expansive areas of the solution space. They are also based on randomisation but are less likely to be stuck in a single local optimum. These meta-heuristics are chosen along the construction heuristics to test diametrically different approaches.

5.4.1 Iterated Greedy

Iterated Greedy (IG) is a simple and effective algorithm introduced by Ruiz and Stützle (2007). The algorithm iterates an encoded solution representation of an ordered list of jobs going into the first stage. This means that it needs a feasible initial solution. NEH is used to provide this.

Each iteration of IG starts by destroying the *current solution* by removing d jobs at random. Then, in the order they were removed, jobs are inserted back into the *destroyed solution* by the same method NEH uses to construct the schedules. Makespan is calculated for all possible positions to place the job in, and the position yielding the lowest makespan is used. The lowest index decides position in case of ties.

Once the solution is fully reconstructed, its makespan is compared to the current solution. If it is better, the *current* and *best* solutions are updated. If it is not, the best solution stays

unchanged, and the current solution is updated with a probability given by Equation 5.3, where $C_{max}(\pi'')$ is the makespan of the new solution, and $C_{max}(\pi)$ is the makespan of the current solution. Note that the temperature in this equation is constant never to restrict the search too much. This works well when the solution space has many local optima, and finding a solution close to one of those is more important than finding the local optimum. Notice also that IG has only two parameters, d and T, that need tuning.

$$prob = e^{-\frac{C_{max}(\pi'') - C_{max}(\pi)}{Temperature}} \tag{5.3}$$

$$Temperature = T * \frac{\sum_{i=1}^{m} \sum_{j=1}^{n} p_{ij}}{n * m * 10} \tag{5.4}$$

5.4.2 Genetic Algorithm

Another popular improvement heuristic in the field of flowshop scheduling, and the main focus of this thesis, is the *genetic algorithm* (GA). It is a population-based algorithm working on a pool of individuals, where each individual is a solution in itself. The difference between a GA and other searches done in parallel is that the GA combines the different solutions throughout the search. GAs aim to make incremental changes to the individuals over generations, guided by the objective function. Figure 5.4 shows a flow chart of the main components of a GA.

First, an *initial population* of a given population size is created. Then, each individual in the population undergoes a *fitness evaluation*, evaluating its quality. Based on the resulting fitness values, individuals are chosen for reproduction through a *parent selection* mechanism with a bias toward choosing individuals with better fitness values. The selected individuals, namely the *parents*, are combined two and two in *crossover* operators to generate *offspring*. The crossover operators merge information from two parents into one or two offspring with a degree of stochasticity (Eiben and Smith, 2015). The idea behind these operators is to combine individuals with different but desired features to produce offspring who inhibit traits from both parents.

Following crossover, the newly created offspring might undergo *mutation*. Mutations are stochastic operators that, with a small probability, modify the offspring to preserve and introduce diversity.

The offspring, or *children*, can be further improved with a *local search*. This is a common way to exploit problem-specific knowledge. Finally, the new population, or *generation*, needs to be compiled with children, parents, or a combination of the two. This selection mechanism is what is called a *generational scheme*. The generational scheme determines how many children to create and how they are selected at the cost of their parents or discarded. The schemes commonly use fitness to rank the candidates. They can, however, be extended further by a technique called *crowding* to ensure the selection also considers diversity.

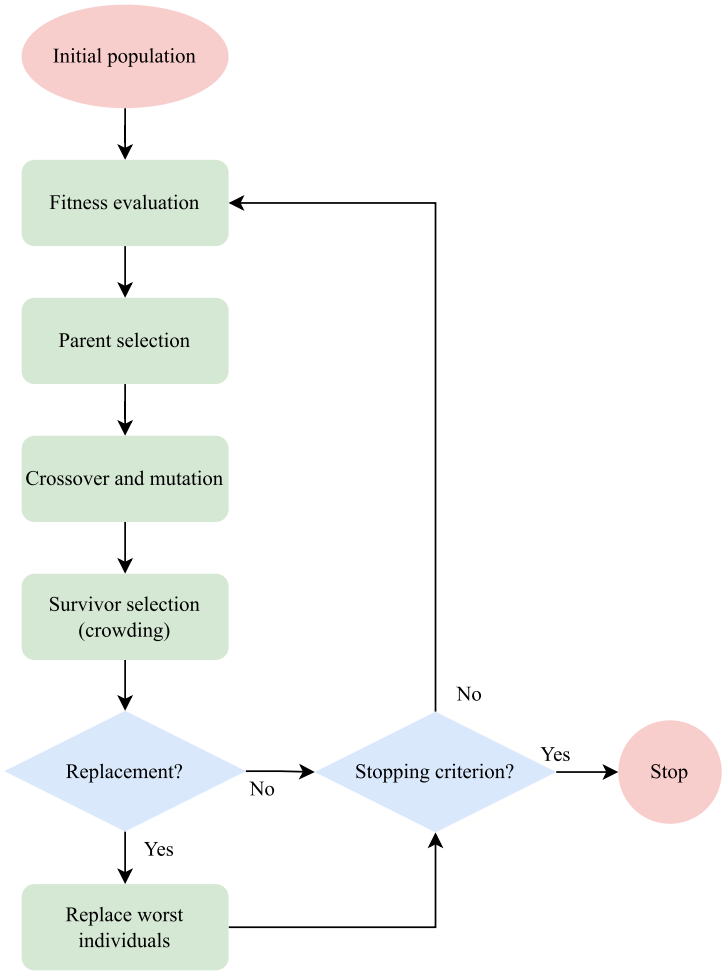


Figure 5.4: Overview of components of a genetic algorithm and the flow between them.

The GA either proceeds with a new generation or stops based on some termination criteria. It is common to stop the genetic algorithm after a fixed number of generations, fitness calculations, after a given time, or when one of the individuals is considered sufficiently good. Further details on GAs and the GA implemented in this thesis are given in Chapter 6.

Chapter 6

Genetic Algorithm

This chapter describes the genetic algorithm (GA) introduced in this thesis in terms of the steps and terminology introduced in Section 5.4.2. First, Section 6.1 defines the objective of the GA. Then, Section 6.2 describes the representation of the individuals and how these map to real-world solutions. Then follows the selection procedure in Section 6.4. Section 6.5 and Section 6.6 describe crossover and mutation operators, respectively, while Section 6.7 explains the local search. Finally, Section 6.8 considers generational schemes, whereas Section 6.9 and Section 6.10 introduce crowding and replacement.

6.1 Objective

The objective of a GA is a function used to evaluate the individuals in the population. It is often referred to as the *fitness function* since it takes an individual as input and yields a *fitness value*; a measure of the quality of the solution. The fitness value is used as the basis of the search procedure, as it is paramount when choosing parents for the next generations and offspring to keep. In other words, the better the fitness value, the more likely the traits of an individual are to be included in later generations. So, the objective must inhibit the requirements the population should adapt to meet (Eiben and Smith, 2015).

The objective of the HFFSP SDST is an integer corresponding to the makespan of the individual's schedule. The calculation of makespan is performed as described in Section 5.2 and is subject to minimisation.

6.2 Solution Representation

Each individual in the population is made up of a set of properties that defines its *chromosome*. However, the chromosome is a *genotype*, an encoded version of the real-life interpretation, the *phenotype*. The choice of encoding to use for the chromosome lays the foundation for the search space of the genetic algorithm and its variation operators. This relation may be represented through the invertible mapping $f : G \mapsto P$, where G is the genotype, and P is the phenotype. This was covered in depth in Section 5.1, so with that reference: An ordered list of jobs to be processed in stage 1 is chosen as the genotype, and the phenotype is found by the makespan calculations introduced in Section 5.2.

6.3 Initialisation

The initialisation is kept simple in most GAs; the first population often contains only randomly generated individuals (Eiben and Smith, 2015). Alternatively, problem-specific construction heuristics can be used to provide better initial individuals. This, however, may come at the cost of the population's diversity, which could, in turn, reduce the breadth of the search. It is common to use a mix of random and heuristic solutions in the first generation to resolve this problem. The hope is thus to have the beneficial traits of high-quality solutions disseminated to later generations without them dominating the whole search. Since this choice is highly problem-specific, the initialisation is tested both as a mix and fully random.

The random initialisation is simple and consists of randomly ordering a list containing all the different jobs. As for high-quality starting individuals, the NEH heuristic is used as a basis. The NEH algorithm is described in Section 5.3.1 and constructs a high-quality individual. However, as NEH provides only one unique solution, we changed the NEH procedure to include randomisation and produce several high-quality individuals. Instead of using the sorted list of jobs according to total production time as input in the second stage of the algorithm, a random permutation of the jobs is used as input. Then, the algorithm proceeds to iteratively insert the jobs in the best possible location in the output permutation. This allows for several high-quality individuals to be created. We categorise the altered NEH algorithm as *Greedy Construction Heuristic* (GCH).

6.4 Selection

There exist several selection mechanisms for selecting the parents in GAs, and *tournament selection* is used in this thesis due to its simplicity and performance (Ruiz and Maroto, 2006). Tournament selection is a selection mechanism in which a local tournament is set up between a given number of individuals selected randomly from the population. The number of individuals in each tournament, k, is referred to as the *tournament size*. Then the fittest of them is chosen. The procedure selects one parent and is repeated to select as many parents as is required. An

example of tournament selection with a population size of six is shown in Figure 6.1. Here, individuals are represented as circles, with their fitness values denoted. Individuals with fitness values 8, 2 and 5 are selected randomly to participate in the tournament. Then, the individual with a fitness value of 8 is selected as the winner, as it has the highest fitness value.

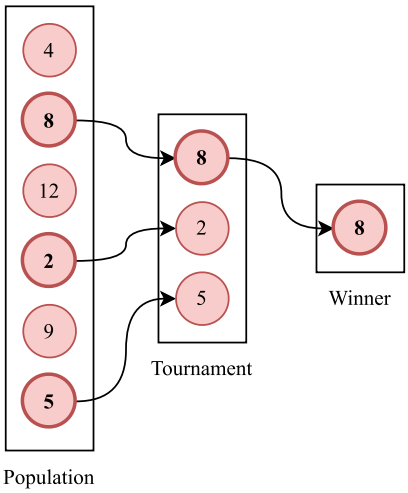


Figure 6.1: An example of tournament selection for a population with six individuals, and tournament size $k = 3$. Maximisation of fitness is assumed.

Tournament selection emphasises the fitter individuals. However, varying the tournament size, k, impacts the selection pressure in the population. When k is set high, a fit individual likely dominates the population. Lower values of k mean higher chances for less fit individuals to reproduce, thus maintaining diversity and increasing chances of escaping local optima. Therefore, as with most other parameters, choosing a k is a trade-off.

6.5 Crossover

Crossover operators are stochastic operators merging information from two parents' genotypes into either one or two offspring genotypes. The choice of which parts of each parent are combined and how this is done is subject to randomisation. Crossover operators aim to combine the beneficial traits of two fit but different parents to create new, fit individuals. Permutation representations, as found in this implementation, require special crossover operators to sustain feasibility. In this thesis, four different crossover operators are implemented and tested. The first two crossover operators are adapted from Ruiz and Maroto (2006), whereas the third is a crossover operator originally developed for a vehicle-routing problem by Ombuki-Berman and Hanshar (2009) but modified in this implementation to fit the HFFSP SDST. The final crossover was first presented in Goldberg et al. (1985) and is a popular operator for many permutation problems. Finally, we also examine options for alternating crossovers.

6.5.1 Similar Job Order Crossover and Similar Block Order Crossover

The first implemented crossover operator is the *Similar Job Order Crossover* (SJOX). SJOX examines two parents' genotypes, *Parent 1* and *Parent 2*, on a position-by-position basis. Wherever the two parents have the same job in the same position, the job is directly copied into two new offspring, *Offspring 1* and *Offspring 2*. Then, a random cut point is chosen between two indices. Each job in front of the cut point of *Parent 1* is copied directly over to *Offspring 1* and similarly for *Parent 2* and *Offspring 2*. Finally, the remaining jobs which are not yet scheduled in *Offspring 1* are inserted into the open positions in the order they appear in *Parent 2*, and similarly for *Offspring 2* and *Parent 1*. An example of this crossover for a problem instance of 10 jobs is shown in Figure 6.2.

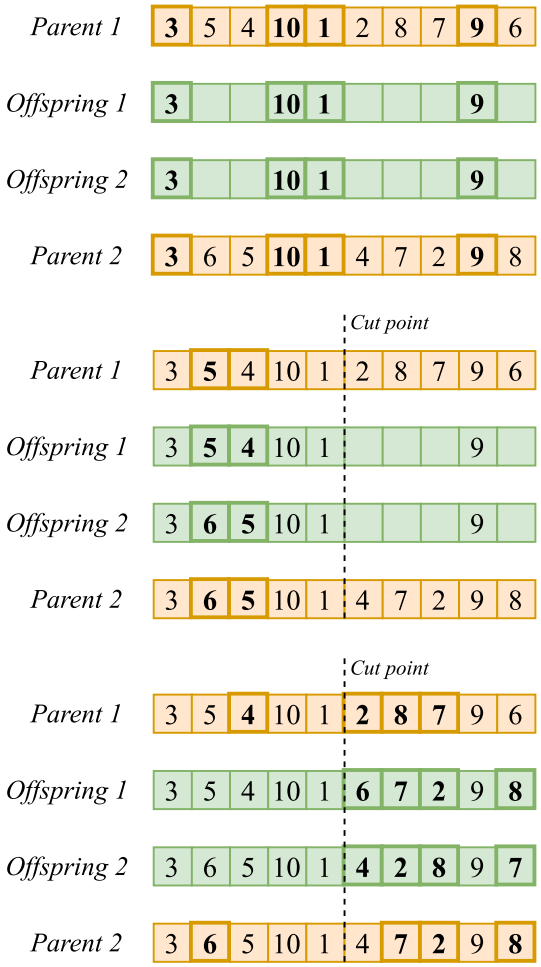


Figure 6.2: The dynamics of SJOX for a problem with 10 jobs.

The second crossover operator is the *Similar Block Order Crossover* (SBOX). This operator is similar to SJOX, as it operates on a positional basis. However, SBOX copies blocks of two or more similar jobs in the same positions succeeding each other, rather than single jobs. This is based on the assumption that it is the *sequence* of jobs, rather than their absolute position, that impacts the makespan the most. The rest of the procedure is the same for SJOX and SBOX.

6.5.2 Best Cost Block Crossover

The third crossover operator is adapted from a GA solving the vehicle routing problem. This is a more problem-specific crossover which uses domain knowledge to generate good offspring.

The *Best Cost Block Crossover* (BCBX) takes two parents, *Parent 1* and *Parent 2*, and creates two offspring, *Offspring 1* and *Offspring 2*. The offspring are initially created as direct copies of the parents. Then, a sequence, or *block*, of jobs of a parameterised length is randomly selected in each of the parents. Following this, all jobs which are present in the block from *Parent 1* are removed from *Offspring 2*, and similarly for *Parent 2* and *Offspring 1*. Finally, the block from *Parent 1* is inserted into the position in *Offspring 2* yielding the lowest makespan value, whereas the same is done for *Parent 2* and *Offspring 1*. This operator enables potentially beneficial sequences of jobs to be carried on through generations and inserts them into the best possible locations. In turn, this may lead to higher quality solutions than solely random operators do. However, the operator is more computationally heavy than the others, as more makespan calculations are needed. An example of this operator applied to a problem instance of ten jobs is illustrated in Figure 6.3.

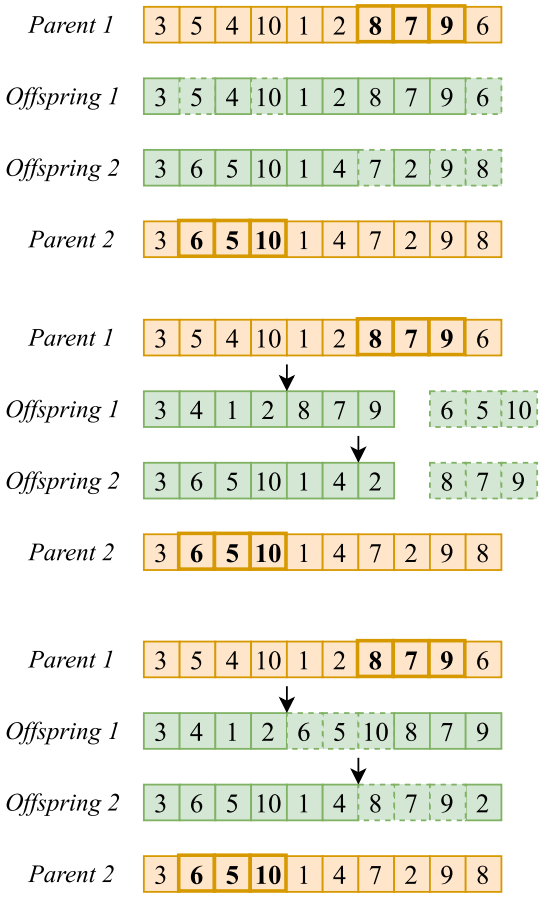


Figure 6.3: The dynamics of BCBX for a problem with 10 jobs.

6.5.3 Partially Mapped Crossover

The last implemented crossover operator is the *Partially Mapped Crossover* (PMX). In PMX, two positions are selected at random. Then, the sequence of jobs between these positions is swapped between the parents to create *Offspring 1* and *Offspring 2*. However, this operation alone may break the permutation as a job may appear more than once, whereas others are entirely removed.

Therefore, a *partial map* is identified between the blocks of jobs that are swapped. This map consists of mappings between the pairs of jobs that occupy the same position in the two parents. That is, if the first job in the block of swapped jobs is job 6 in *Parent 1* and job 3 in *Parent 2*, $6 \leftrightarrow 3$ appears as a *mapping* in the partial maps. These mappings are then used to fix the broken permutations by updating the conflicting jobs in each offspring outside the swapped blocks. If a job outside the swapped blocks exists in the partial map, it is changed to the job it maps to. This mapping restores the permutations.

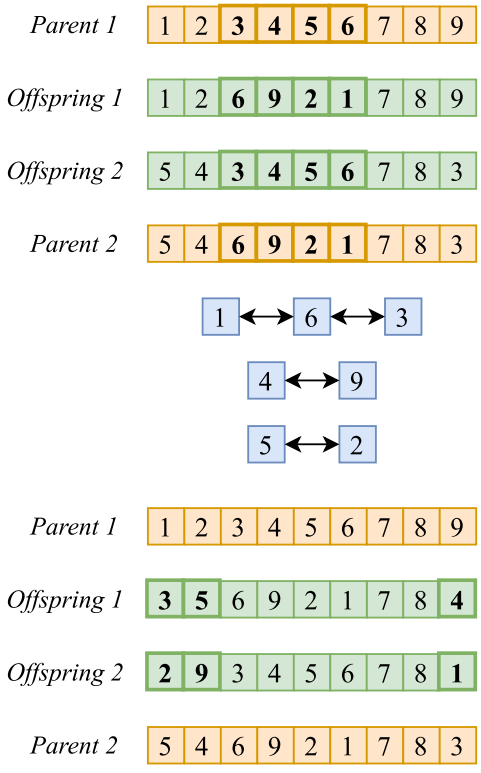


Figure 6.4: The dynamics of PMX for a problem with 9 jobs.

Figure 6.4 illustrates PMX for a problem instance with ten jobs. The sequences of jobs from location 3 to location 6 are selected and swapped between the two parents to generate *Offspring 1* and *Offspring 2*. Note that this operation breaks the permutation. For example, job 1 appears twice in *Offspring 1*, and never in *Offspring 2*. Therefore, the pairs of jobs that are swapped are recorded as mappings in the partial map (blue). Since, for instance, job 6 and job 3 are in the same location in the selected sequence, they appear in the mapping. Because job 6 appears in the selected sequence in both parents, this job maps to job 1 as well, as shown in the figure.

Finally, the jobs outside the selected sequence are mapped to their respective values for the two offspring, and feasible permutations are restored.

6.5.4 Adaptive Choice of Crossover Operators

The most intuitive way of choosing a crossover operator in any GA is to use the same every time. However, the different operators have unique traits, making it likely they perform well in different search stages and for problem instances of particular characteristics. A low-effort way of testing if a combination of crossover operators provides a better search than a single one is to choose the operator randomly. In addition, it serves as a good benchmark for any adaptive method of choosing. A random crossover operator choice mechanism is implemented with all aforementioned crossovers, and they are all given an equal chance of being picked every time the crossover procedure is called.

Q-learning is a machine learning and, more specifically, a reinforcement learning technique that keeps track of and updates the chances of choosing an action based on previous success (Watkins and Dayan, 1992). Q-learning is based on an agent-environment interaction where the *agent* can choose among available *actions* in any state of the *environment* it finds itself in. An agent is defined as an entity sensing its environment, making autonomous decisions of its actions, in this case, which crossover operator to use, and a manipulator of its surroundings. In this case, the environment is *known* and *deterministic*, meaning that the agent can perceive the state of the environment correctly and knows exactly how its actions will alter it.

At the heart of Q-learning is the *Q-table* of *state-action pairs* and *Q-values*. A state-action pair is a combination of a state and an action, and a Q-value is a measure of success the agent has had executing this action given the state. The agent chooses to execute the action with the highest Q-value for a given state. However, for Q-learning to work in this context, the state cannot be a representation of the population, as this is very unlikely to repeat, rendering the system of learning ineffective. The Q-table is reduced to actions with a Q-value, making it a total of four choices in every iteration.

The Q-values are all initiated to zero. Whenever an operator is used, the performance is evaluated and given a *reward*, a measure of the operator's success. The reward is set to the improvement the best child has over the best parent. Crossovers are highly random, and the parents are expected to be relatively fit. The goal of crossovers is not to produce improvements every time but to induce diversity and find local optima. In addition, when using elitism or steady-state, there is no difference in a crossover producing a horrible or slightly inferior offspring. Offspring being better than the parents is the only thing that matters. For these reasons, when failing to produce better offspring, crossovers are not punished proportionally to the difference in makespan between offspring and parent, but with a zero reward.

Once the reward is determined, the Q-value is updated with Equation 6.1. $Q_n(a)$ is the Q-value of action a after the n^{th} iteration, $R_n(a)$ is the reward of action a in the n^{th} iteration and α is the learning rate.

$$Q_n(a) = (1 - \alpha)Q_{n-1}(a) + \alpha R_n(a) \tag{6.1}$$

There is a constant dilemma of *exploring* if there exist any more suitable crossovers than the one with the current highest Q-value and *exploiting* that the agent possibly knows what the current best choice is. This is called the *explore/exploit dilemma* and is solved by an *epsilon*-value denoting the chance of making a random choice among the available actions. Epsilon needs to be tuned along with the learning rate.

6.6 Mutation

Mutations are variation operators usually working on only one individual at a time, making slight modifications to chromosomes called *mutants* (Eiben and Smith, 2015). Mutation takes place after parents are selected and crossover is performed. It is relatively rare in biology, and the same is seen in most GAs, with a mutation probability around 2% (Simon, 2013). Nevertheless, mutation is important, allowing the evolutionary process to explore new areas of the search space and increasing diversity. For this thesis, four different mutation operators are implemented and tested.

6.6.1 Shift

The first mutation operator is the *Shift* mutation. The Shift mutation selects a job at a random position in the permutation and inserts it into another randomly selected position. The jobs between these two positions are shifted one position forwards or backwards, without changing their internal order, to accommodate the move. For example, if the job at location 3 is moved to location 7, the jobs that occupied positions 4 to 7 are moved to occupy positions 3 to 6. This is illustrated in Figure 6.5.

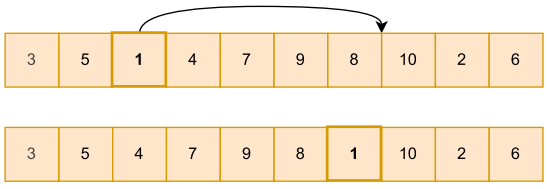


Figure 6.5: The dynamics of the Shift mutation.

6.6.2 Swap

The second mutation operator is the *Swap* mutation. Similarly to the Shift mutation, two positions are selected at random. However, instead of moving only one job and shifting the rest, the two jobs that occupy the two random locations swap places. This is illustrated in Figure 6.6, where job 5 and job 9 are selected to swap positions.

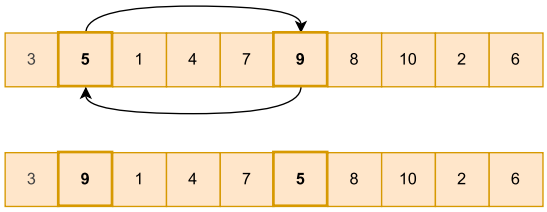


Figure 6.6: The dynamics of the Swap mutation.

6.6.3 Reversal

In the third mutation operator, the *Reversal* mutation, one position is selected at random. Then, the sequence of jobs within a fixed range behind this position, including the position, is reversed. For example, if the selected position is 2, and the range we want to reverse is of length 4, then position 2 is swapped for position 5 and position 3 is swapped with position 4, as visualised in Figure 6.7.

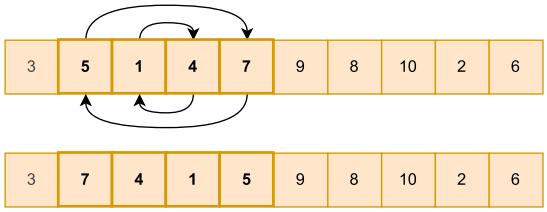


Figure 6.7: The dynamics of the Reversal mutation.

6.6.4 Greedy

The final implemented mutation operator, *Greedy*, is problem-specific and uses domain knowledge to create higher-quality offspring. This operator selects a job at random and removes it from the sequence of jobs. Then it checks all possible positions in the sequence for the position yielding the lowest makespan. If there are several positions with equal lowest makespan, one of them is picked at random. Figure 6.8 displays an example of the Greedy mutation operator for a problem instance with 10 jobs. Job 2 is selected for re-positioning. All possible positions are evaluated, and there are two positions with equally low makespan (arrows). The arrow marked with an asterisk (*) is randomly picked between these two superior positions.

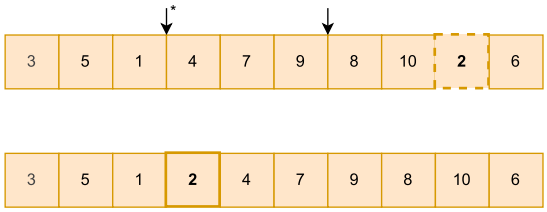


Figure 6.8: The dynamics of the Greedy mutation.

6.7 Local Search

After new individuals have been created and potentially mutated, they could be further improved by a local search. Though this is not something every GA uses, IG is implemented as described in Section 5.4.1. Each iteration of IG is very time-consuming compared to crossover and mutation operators. Therefore, to make sure the local search does not occupy the larger part of the run-time of the GA, the number of iterations is limited by a parameter.

A GA enhanced with a local search is in the literature typically referred to as a *memetic algorithm*. However, as the local search is an extension that will be tested and not an integral part of the GA itself, we will continue using the term GA for our implementation.

6.8 Generational Scheme

The generational scheme in a GA dictates the change in population from generation to generation. Two typical generational schemes are *generational* GAs and *steady-state* GAs. Both are implemented and described in this section. Their performances are highly problem dependent and have to be tested.

6.8.1 Generational Genetic Algorithm

A generational GA generates as many offspring in each generation as there are parents in the population. In each generation, all parents in the current population are discarded to make space for the newly created individuals, which then constitute the new population. This is done by filling up a *mating pool* with the selection procedure, tournament selection, as described in Section 6.4. Since fitter individuals are more likely to be selected, they often appear more than once. When the mating pool is filled up, two and two parents, starting with the first and second parent, are paired together to create offspring through crossover and mutation. This generational scheme changes the whole population every generation and thus makes large changes every iteration. Generational GAs often implement *elitism*. Elitism involves keeping a small number of the best-known parents instead of replacing all of them each generation.

6.8.2 Steady-State Genetic Algorithm

With a steady-state generational scheme, only one or two new offspring are created in each generation. For each generation, two parents are selected. Crossover is used to generate new offspring, and these are potentially mutated. Their fitness is compared to the current least fit individuals in the population. If the offspring is fitter, they replace the least fit individuals; else, they are discarded. As opposed to a generational GA, the steady-state GA only makes small changes to the population in every generation.

6.9 Crowding

Niching algorithms are used to maintain diversity and are an important research area in genetic and evolutionary computing (Mengshoel and Goldberg, 2008). The two main objectives of niching algorithms are to (1) converge the population to multiple, highly fit, and significantly different solutions and (2) slow down convergence in applications where only one solution is needed. *Crowding* is, together with other methods such as fitness sharing and clustering, a common niching approach. It is important to note that these methods change the generational schemes rather significantly by altering the methods of selecting the new generation.

It is paramount to define what *similarity* between individuals means when considering diversity. For permutations, like job sequences, several *distance metrics* exist for defining similarity, and a comprehensive list is provided by Cicirello (2019). We focus on algorithmic and run-time simplicity, and the two chosen metrics are introduced below.

Exact Match

The *exact match* metric counts the positions where two permutations differ. This is done by looping through the two job permutations parallelly and increasing a counter for every position with different jobs. For this implementation, the metric runs in $\mathcal{O}(n)$ time, where n is the number of jobs. Figure 6.9 shows an example of a problem with ten jobs. The jobs in six positions differ, and thus, the exact match distance is six.

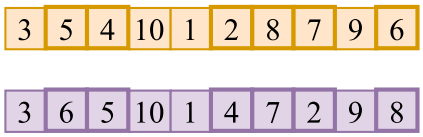


Figure 6.9: An example of the exact match distance metric. The distance in this example is six.

Deviation Distance

The *deviation distance* metric is the sum of all index deviations between the same jobs in two job permutations. This metric is implemented by running through the permutations, p_1 and p_2, and storing the elements' positional indexes in separate arrays, i_1 and i_2. Then the difference in each job's indices is calculated and summed. This gives a run-time complexity of $\mathcal{O}(n)$, n being the number of jobs. This is exemplified in Figure 6.10 with a problem instance of ten jobs. Since job 1 is found in the 5^{th} position in p_1, the 1^{st} entry of i_1 is 5, and so on. In d_{12}, the absolute values of the differences between the entries in i_1 and i_2 are recorded. The final metric is the sum of these values, and the distance is found to be 18.

With well-defined similarity measures and diversity in mind, the initial single focus on fitness in the generational schemes is changed. Instead of always considering the entire population while evaluating children, crowding uses smaller tournaments among similar individuals. This is to avoid replacing a solution in a very different part of the search space because it has a lower fitness value. Here, we propose the famous allegory of comparing apples to oranges. Two solutions can have very different contributions to the process and should not be put up against each

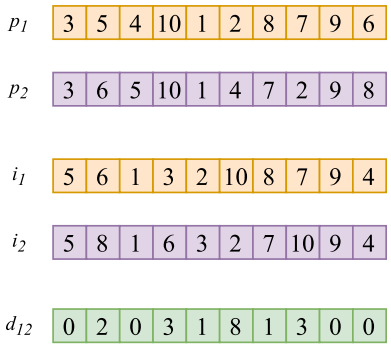


Figure 6.10: An example of the deviation distance metric. The distance in this example is 18.

other. Note that generational GAs and steady-state GAs are different and have different ways of determining what individuals are to compete in a tournament. However, the replacement rules for deciding which individual to keep once the tournament is set are the same. The two crowding algorithms are explained in Section 6.9.1 and Section 6.9.2, while the replacement rules are introduced in Section 6.9.3.

6.9.1 Implicit Crowding

Implicit crowding is used for generational GAs and implements tournaments between children and their direct parents. This changes the implementation of generational GAs by allowing parents to survive at the cost of their children. It is done by comparing the individuals most alike. The approach works as follows (Galan and Mengshoel, 2010):

1. Parents (p_1, p_2) are selected through the given selection mechanism and paired for crossover to generate children (c_1, c_2). Mutation is applied with a probability P_m.
2. Children (c_1, c_2) compete with one of their parents depending on the chosen distance metric. Let $d(i_1, i_2)$ denote the distance between two individuals, i_1 and i_2:

   If $d(p_1, c_1) + d(p_2, c_2) < d(p_1, c_2) + d(p_2, c_1)$
   $w_1 \leftarrow$ winner of competition between p_1 and c_1
   $w_2 \leftarrow$ winner of competition between p_2 and c_2
   else
   $w_1 \leftarrow$ winner of competition between p_1 and c_2
   $w_2 \leftarrow$ winner of competition between p_2 and c_1

In the tournament above, w_1 and w_2 denote the winners of the two local competitions, respectively. Put in words, implicit crowding pairs parents and children based on their similarities. If the sum of the distances between (p_1, c_1) and (p_2, c_2) is less than that of (p_1, c_2) and (p_2, c_1), then p_1 competes with c_1 and p_2 competes with c_2. Else, p_1 competes with c_2 and p_2 competes with c_1. The winners of the pairs are kept for the new generation. In general, the fittest individual triumphs, but more on this in Section 6.9.3.

Using implicit crowding in a steady-state GA does not have the intended effect. The fitter parents have a higher probability of being selected for reproduction, and these would also have a higher probability of being replaced when offspring compete with their direct parents. The less fit individuals, which are more seldomly selected, would then be included in the population for a longer time, even though they are very similar to the fitter children.

6.9.2 Explicit Crowding

Explicit crowding is used for steady-state GAs and implements tournaments between a child and a number of the most similar parents. To initiate a tournament using explicit crowding, the k individuals most similar to a new child are identified. The child and the least fit individual among the ones identified compete to decide who proceeds to the next generation. The *k-nearest* approach that is used here is very sensitive to the parameter k, so this should be chosen carefully.

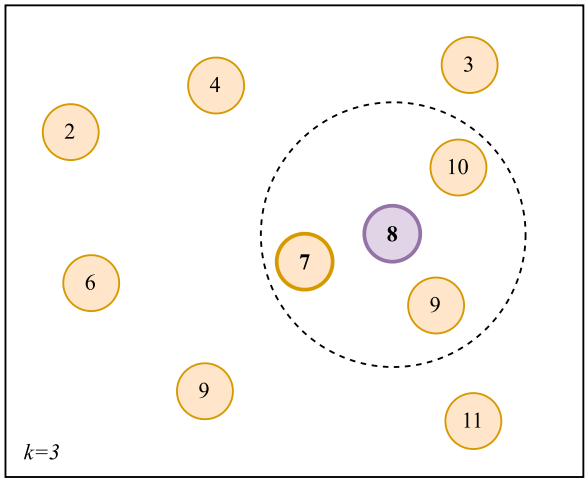


Figure 6.11: The *k-nearest* procedure for selecting who will compete with the newly created child (purple), when $k = 3$. Individuals are represented as circles, with their respective fitness values denoted. The fitness is assumed maximised in this example.

In Figure 6.11, explicit crowding with the k-nearest approach is visualised. The individuals are represented as circles with respective fitness values denoted (fitness is assumed maximised). The newly created child is marked in purple, and its three nearest neighbours are found inside the dotted circle. Of these three individuals, the one with a fitness value of 7 is the least fit. Therefore, this is the individual selected to compete with the offspring for survival. The internals of the actual competition is described in the following section.

6.9.3 Replacement Rules

As described above, local competitions are created in both implicit and explicit crowding. These competitions consist of *replacement rules* that are responsible for deciding which of the two competing individuals wins. Typical replacement rules are *Boltzmann replacement*, *deterministic crowding*, *probabilistic crowding* and *generalised crowding* (Galan and Mengshoel,

2010). In the next paragraphs, the latter three will be briefly explained. In these paragraphs, the fitness function, $f : \mathbb{Z}_+^n \mapsto \mathbb{Z}_+$, is assumed to be maximised, without loss of generality. For minimisation problems, the function f can be substituted as $1/f$ instead.

Deterministic Crowding

In *deterministic crowding*, the survivor of the competition between parent p and child c is simply the individual with the highest fitness. Let P_c denote the probability that child c replaces parent p in the population. Then it is given by the following expression:

$$P_c = \begin{cases} 1 & \text{if } f(c) > f(p), \\ 0.5 & \text{if } f(c) = f(p), \\ 0 & \text{if } f(c) < f(p). \end{cases} \tag{6.2}$$

Probabilistic Crowding

In contrast to deterministic crowding, *probabilistic crowding* uses a non-deterministic rule to choose a survivor between parent p and child c. The probability depends on the ratio of the fitness between the two individuals. The probability that the child survives, P_c, is defined as follows:

$$P_c = \frac{f(c)}{f(c) + f(p)} \tag{6.3}$$

Generalised Crowding

In *generalised crowding*, the probability is scaled by a factor of ϕ. This scaling factor offers a broad range of replacement rules by simply adjusting ϕ. In generalised crowding, the probability that child c survives the tournament, P_c, is defined as follows:

$$P_c = \begin{cases} \frac{f(c)}{f(c)+\phi \times f(p)} & \text{if } f(c) > f(p), \\ 0.5 & \text{if } f(c) = f(p), \\ \frac{\phi \times f(c)}{\phi \times f(c)+f(p)} & \text{if } f(c) < f(p). \end{cases} \tag{6.4}$$

Equation 6.4 assumes maximising f, as well as $\phi \in \mathbb{R}^+ \cup \{0\}$. Notice that for $\phi = 0$, the probability for generalised crowding becomes equal to that of deterministic crowding. Moreover, for $\phi = 1$, generalised crowding turns into probabilistic crowding. For $0 < \phi < 1$, it is possible that the least fit of the competing individuals wins, but less likely than in probabilistic crowding. Finally, for $\phi > 1$, the probability that the least fit individual wins the tournament is greater than that of probabilistic crowding. Generalised crowding is implemented, as this implicitly also leaves the option to test both deterministic and probabilistic crowding.

6.10 Replacement

A *replacement scheme* is used to avoid stagnation and replaces the least fit portion of the population, P_r, after a given number of iterations, t, without improvement in the fittest individual. We propose three replacement schemes:

1. **Random**: Replace P_r of the population with randomly generated individuals.
2. **GCH**: Replace P_r of the population with individuals generated by the GCH described in Section 6.3.
3. **Mutate**: Replace P_r of the population where half of the new individuals are made by repeatedly choosing one of the individuals from the $(1-P_r)$ fittest portion of the population and mutating it once. The second half is randomly generating new individuals.

Chapter 7

Computational Study

This chapter describes the testing methodology, presents and discusses the results from tuning, and concludes on the performance of the implemented algorithms. The focus is on tuning the genetic algorithm (GA) and comparing it with the benchmark algorithms and the mathematical model. Section 7.1 introduces the problem instances used for testing. Section 7.2 presents and discusses the results of tuning the GA, whereas Section 7.3 comments briefly on tuning the Iterated Greedy (IG) algorithm. Finally, the algorithms' convergence and performance are examined in Section 7.4 and Section 7.5, respectively.

All algorithms are implemented in the programming language *Rust*. The complete list of technical specifications is given in Table 7.1. A random seed is set to assure reproducible results. In Rust, this is done by initiating a random number generator with the given seed and then passing this generator to every operator in the GA that depends on random drawings. This is equivalent to keeping a list of random numbers and drawing the next one every time any random choice is made. The generator is re-initiated for the solution process of every problem instance, which is equivalent to starting from the beginning of the list of random numbers. It should be noted that the sequence of random numbers in Rust is system dependent, so to reproduce our exact results, the tests would need to be performed in a similar environment.

Table 7.1: Hardware and software specifications used in the computational studies.

Processor	2.90 GHz Intel Core i7-10700
Memory	16 GB RAM
CPU Cores	8
Operating System	Windows 10 Education
Rust version	1.59.0
Random seed	123

7.1 Problem Instances

To compare the algorithms, we use a set of 960 problem instances developed by Naderi et al. (2010). The problem instances range between 20 and 120 jobs, 2 and 8 stages, and there are between 1 and 4 machines in each stage. More specifically, there is a uniform distribution of instances of 20, 50, 80, and 120 jobs and instances with 2, 4, and 8 stages. Moreover, the processing times are generated from a uniform distribution [1, 99]. For the smaller problem instances of 20 to 50 jobs, half of the instances have their set-up times drawn from a distribution which is 25% of the processing time distribution. The other half has set-up times from the processing distribution, meaning that set-up times and processing times are, on average, the same size. The larger instances, 80 to 120 jobs, scales the set-up-to-processing distribution ratio in four equally large levels: 25 %, 50 %, 100 % and 125 %. The variation in ratios ensures the problem instances are of diverse characteristics, so the algorithms are evaluated for a wide range of production situations. All instances are available for download from `http://soa.iti.es`. The set of problem instances is called "Instances for hybrid flexible flowshop problems with setups", and the download also includes a detailed description.

7.2 Tuning of Genetic Algorithm

This section presents results from tuning the GA, which is vital to its performance. Since there are many parameters and combinations, the approach needs to be systematic. Section 7.2.1 describes the testing methodology, and in Section 7.2.2, the results are presented and discussed, and the parameters are determined.

7.2.1 Methodology

Ninety-six problem instances are chosen to test set-ups of the GA, which is a representative selection. The reasoning is twofold. (1) Limiting the test set helps to avoid *overfitting*, which is a problem when tuning and testing on the same data set. Overfitting makes the resulting algorithm adept at solving the problem instances it is tuned for but leads to a loss of generality. (2) It saves time and simplifies testing. A representative selection is achieved by selecting eight instances for each combination of jobs and stages. In addition, instances are selected with representative set-up-to-processing time ratios.

A reasonable set of parameters that receive good results is found through preliminary tests. These parameters are used as the *base case* and listed in Table 7.2. With the base case established, several values are tested at a time for one or two parameters. If a parameter value performs better than the parameter value in the base case, its value is updated. Then, the updated base case is used in the next test. This process repeats until all parameters are tested with the ever-updating base case. We have not opted to test every parameter combination as some do in the literature. It would be too time-consuming with the number of parameters

included in this study. Not testing all instances and all combinations of parameters might not yield the optimal set-up, but meta-heuristics are not about absolute precision. As will be evident in Section 7.2.2, we do not consider the statistical significance of the performance of any parameters for the same reason.

Table 7.2: The parameter values in the base case resulting from preliminary tests.

Parameter	**Value**
Makespan calculation	FIFO
Generational scheme	Steady-state
Initialisation	GCH (100 %)
Population size	150
Tournament size	2
Crossover	PMX
Mutation	Shift
Mutation probability	5 %
Local search	No
Crowding	No
Replacement	No

During tuning, the GA is allocated time based on the size of the problem instances it solves. The relevant size indicators are the number of jobs and stages, which determine the solution and search space. This is in line with the literature as seen in, for instance, Naderi et al. (2010). The exact time allocated is given by Equation 7.1, $\mathcal{N}$ being the set of jobs, and $\mathcal{I}$ the set of stages.

$$\text{Time } (ms) = |\mathcal{N}|^{1.7} \cdot |\mathcal{I}| \cdot 1.5 \tag{7.1}$$

This equation results in 488 ms of run-time for the smallest problem instances consisting of 20 jobs and two stages. On the other end of the spectrum, the largest instances consisting of 120 jobs and eight stages are allocated about 41 s. The time starts at the first line of code, being initialisation, and interrupts the algorithm during the final generation.

Several parameter values or algorithms are used to solve the same problem instances during testing. To determine which performs better, they need to be compared with a suitable measure. Using the average makespan is a poor measurement as the larger problem instances or those with longer production or set-up times will dominate the smaller ones. We opt, therefore, to use the relative percentage deviation (RPD) as defined by Equation 7.2. Alg_{sol} refers to the makespan of a given algorithm, whereas $Best_{sol}$ refers to the best makespan achieved by any of the competing algorithms. Using the average RPD makes it possible to compare algorithms despite varying problem sizes.

$$RPD = \frac{Alg_{sol} - Best_{sol}}{Best_{sol}} \cdot 100\% \tag{7.2}$$

7.2.2 Tuning

This section presents the results from the tuning of the implemented GA. In total, 11 different tests have been carried out. Only the most prominent results are discussed as this is a comprehensive study. The results from all tests are found in Appendix A. The second axis of all graphs in this sub-section is the average RPD, as described in Section 7.2.1. The vertical bars extending from the average RPD are the *standard error of the mean* (SEM) of the RPDs. SEM measures how different the population mean likely is from a sample mean. This is calculated as per Equation 7.3. Here, σ refers to the standard deviation of the RPDs and n to the number of instances.

$$SEM = \frac{\sigma}{\sqrt{n-1}} \tag{7.3}$$

Makespan calculation

It appears to be significant performance deviations between the two makespan calculation procedures presented in Section 5.2. In fact, the FIFO procedure results in a better makespan value in 90 out of the 96 problem instances. The average RPD for the FIFO procedure is $< 1\%$, whereas it is 15 % for the first-completion procedure.

While examining the solutions made by the first-completion procedure, it becomes apparent that the reason for the performance gap might be more intuitive than first anticipated. The first-completion procedure always assigns machines the job that can be completed first. It does not consider what jobs are currently in the buffer, ready to start processing. Therefore, machines could end up idle, waiting for a job to finish in the previous stage. This defect is illustrated with an example in Figure 7.1

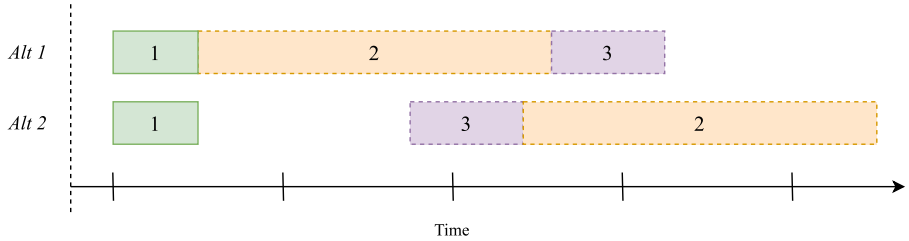


Figure 7.1: The two alternatives of assigning job 2 and job 3 to a machine in an arbitrary stage.

Figure 7.1 is an example of the issue the first-completion procedure experiences when assigning jobs to machines. For simplicity, the example only considers the assignment of two jobs on one machine in an arbitrary stage. Job 1 is already assigned, whereas jobs 2 and 3 are not. Job 2 is available in the machine's buffer at the moment when job 1 is done processed, whereas job

3 arrives at a later time. Moreover, job 2 has a considerably longer processing time than job 3 in this stage. The figure illustrates the schedules of processing job 2 before job 3 and vice versa. With the first-completion procedure, the machine waits for job 3 rather than processing job 2 first since it can complete job 3 the earliest. This corresponds to *Alt 2*, with idle time the machine could have avoided with *Alt 1*.

We find the phenomenon to occur the most when the differences in processing times are extensive. The jobs with the shorter processing times will often be prioritised, leaving the jobs with the longer processing times to be processed last. Another effect, not evident from this example, is that schedules derived from the first-completion procedure are very similar for different input permutations. Similar schedules would not have been a problem had they been satisfactory. However, they are, in general, worse than the schedules provided by the FIFO procedure. The dismal performance of the GA could also be due to the symmetry the first-completion procedure introduces. With a many-to-one mapping between the solution- and search space, the search space has redundancy, as it operates with solutions without any practical significance. It also flattens the search space, giving more solutions the same objective value making it difficult for search algorithms to manoeuvre. For these reasons, the FIFO procedure is chosen as the makespan calculation procedure.

Generational scheme

The second entry in the base case is the generational scheme. Figure 7.2 shows the GA with a steady-state generational scheme outperforming the GA with a generational set-up. Although it is difficult to say why this is for certain, one explanation could be the complexity of the search and the lack of fit children. If, on average, the new generation is worse than the old, besides the few solutions kept due to elitism, then using a GA changing the entire population in every generation might not provide the same progression that a steady-state GA does. This reasoning comes from the idea that, given enough diversity, changing an individual for a less fit one will not take the search in the right direction. It is also in accordance with literature that has tested both schemes (Ruiz and Maroto, 2006; Ruiz et al., 2006). For these reasons, steady-state is included in the base case.

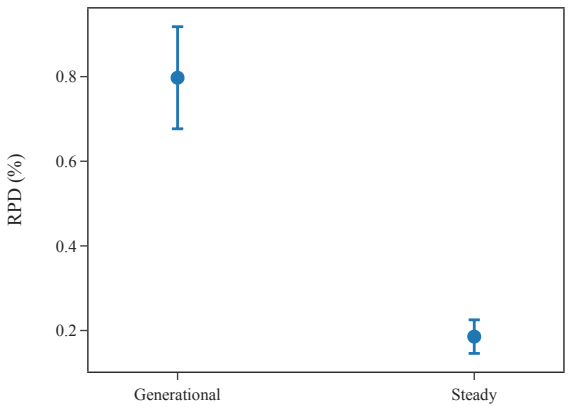


Figure 7.2: Average RPD for the generational and steady-state generational schemes.

Initialisation

Five different methods for initialising the population are considered. Like in most literature, an entirely randomly generated population is tested. With a slight tweak, the population is also initialised with randomly generated individuals and one individual created by the adapted NEH heuristic. Finally, three different alternatives are tested with individuals created by the Greedy Construction Heuristic (GCH). These alternatives include initialising 20 %, 50 %, and 100 % of the initial population with solutions from the GCH. The remaining individuals that are not created by GCH are created randomly. From Figure 7.3, it is evident that initialising the population with solutions solely from the GCH performs the best.

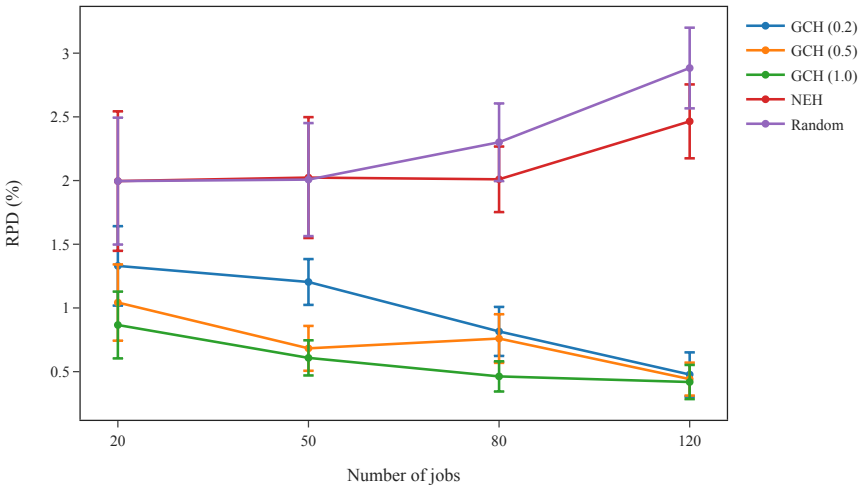


Figure 7.3: Average RPD for the different initialisation methods, grouped by the number of jobs.

It is not surprising that starting with reasonable solutions is better than the alternative. Especially knowing that there is no reason to believe these solutions to be less diverse than what randomly generated solutions are. Figure 7.3 shows the gap from random and NEH to GCH increases as the number of jobs increases. The increasing gap shows how more than one fit solution grows in importance as the number of jobs increases.

Another observation is that the spread among 20%, 50%, and 100% fit solutions decreases as the number of jobs increases. A reason could be that for larger problem instances, the extra search time outweighs the gains of an initially all fit population. Figure 7.4 shows that creating 100% fit individuals takes up a significant amount of the time for larger instances with 41 s run-time. With the smaller instances, it is, to the contrary, insignificant. These results indicate that the initial population benefits from a critical mass of fit individuals with a steady-state generational scheme. However, increasing it further gives limited benefit, especially for the larger problem instances. With 100% fit individuals being best for all instances, it is included in the base case.

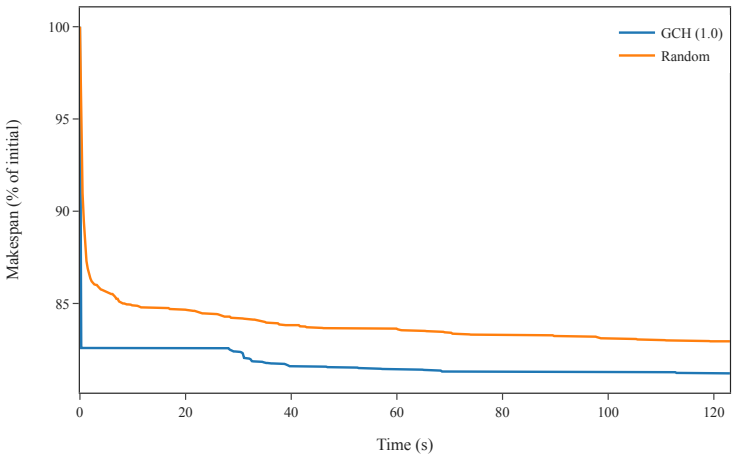


Figure 7.4: Convergence of the average of ten problem instances with 120 jobs and 8 stages.

Figure 7.4 shows how the GA with random initialisation converges quickly for the first generations but never reaches the best initial solution created with GCH within the 41 s. This phenomenon does not change when running them in parallel for a while longer. However, it is possible that the random initialisation eventually would reach the level of a 100% GCH initialisation. Note that the graph does not include the progression of GCH improvements, and the entire initialisation of *GCH (1.0)* is set to the best solution found at any time during initialisation. Since 100% GCH initialisation works best in the time frame considered in this thesis, it is added to the base case.

Population size

Preliminary testing found a population size of about 150 to be best. Thus, test values are chosen based on this and what others have used previously, which is generally a bit below 150. The results broken down by the number of jobs can be found in Figure 7.5. Note that a population size of 50 is the worst option for all but 120 jobs and is significantly better for 80 than 20 and 50 jobs. This behaviour could be connected to the initialisation of the population. The GA spends a large amount of its available time generating the initial population for the larger instances. A smaller population size allows for more time to perform crossover and mutation, whereas a population size of 300 probably does not have any time for this. That could explain why the performance of the GA for the larger problem instances is inversely proportional to the population size. Nevertheless, this study looks to maximise the performance of the GA for all problem instances. With a population size of 150 having the lowest average RPD, it is included in the base case.

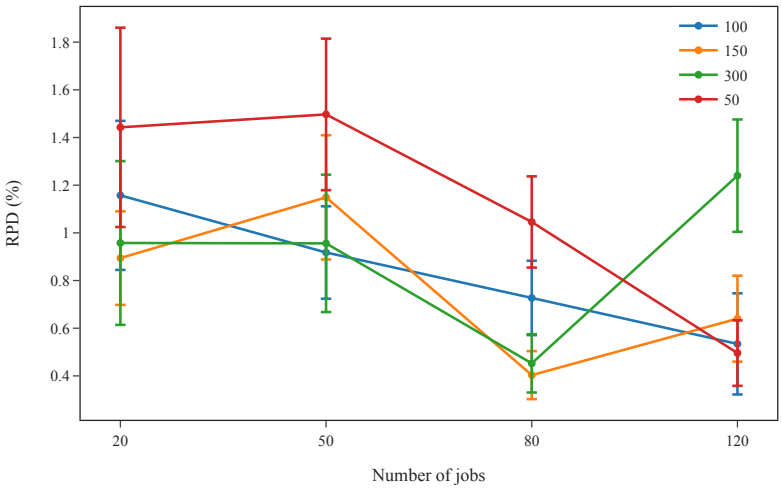


Figure 7.5: Average RPD for different population sizes, grouped by the number of jobs of problem instances.

Tournament size

A tournament size of 2 is best on average and close to the best for all problem instance sizes in the number of jobs and stages. The second best option is 3, indicating that tournaments among a few individuals are the most effective. As the steady-state generational scheme prioritises elitism over diversity, a small tournament size gives the opposite effect. All the graphs containing the results of this test are included in the Appendix A. A tournament size of 2 is kept in the base case.

Crossover

The fifth entry in the GA's base case is the crossover. The results from this test are plotted in Figure 7.6. BCBX is the best operator out of the four implemented crossover operators. This operator finds excellent solutions for the smaller problem instances of 20 and 50 jobs. However, for the larger instances, other crossover operators perform better. This is likely because BCBX operates similarly to GCH and spends more time generating children as the problem instances, and hence job permutation sizes increase. SBOX and SJOX do not have this disadvantage for the larger problem instances and outperform the other crossovers for 80 and 120 job instances.

When looking at performance categorised by the stages in problem instances, the BCBX is again superior out of the standard crossover operators. The difference is most prominent in problem instances with eight stages. The outperformance in many stages can be due to choosing blocks that work well for more than the first stages. BCBX uses domain knowledge and tests several alternative solutions. This search is more complicated and might draw more use of the extra complexity of the BCBX operator.

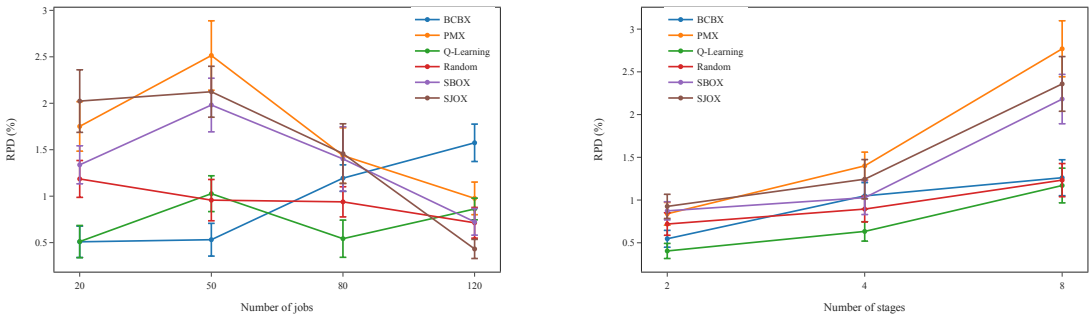


Figure 7.6: Average RPD for the crossover operators, grouped by the number of jobs (left) and the number of stages (right).

Choosing a random crossover operator is never the best option out of all the crossover methods tested here, but it performs well overall. It has the second-lowest average RPD, indicating that varying the choice of crossover has some merit. With a random choice, the BCBX is chosen sufficiently often for a random choice to beat the other standard crossovers in BCBX's optimal range. In addition, it is punished less severely for the larger instances, using enough of the quicker crossover operators to ensure sufficiently many iterations.

The best choice of an operator is to choose it adaptively, as here implemented by a version of Q-learning. It gives, by far, the lowest average RPD and is among the best alternatives across all sized problem instances. After running the algorithm, inspecting the Q-table shows that BCBX is chosen more often for the smaller problem instances, and PMX, SJOX, and SBOX are chosen more for the larger ones. Before this test, however, the internal parameters of Q-learning are tuned. The results can be found in Appendix A. The best options are a learning rate of 0.2 and an epsilon of 0.25. These parameters emphasise that although the algorithm should be able to learn, it should also be able to reconsider rather quickly and keep choosing a large portion of the crossover operators at random. Since it performs best overall, Q-learning is adopted into the base case.

Mutation

Out of the four mutations, Shift and Greedy perform nearly equally well. They receive an average RPD well below Swap and Reverse. Possible reasons for this could be that Shift mostly maintains blocks of jobs in the job permutations while giving only one job a new relative order, whereas most of the other jobs keep their immediate predecessor and successor. This is one of the aspects described thoroughly in the literature. On the other hand, Greedy breaks down the solutions more but uses domain knowledge to build them up again. It may not keep the blocks but ensures the creation of some that are probably just as good or better. These mutations cater well to the sequence-dependent set-up times, which Reverse and Swap do not. Reverse changes the whole order of a section, potentially increasing set-up times, and Swap breaks up twice as much of the order as Shift does.

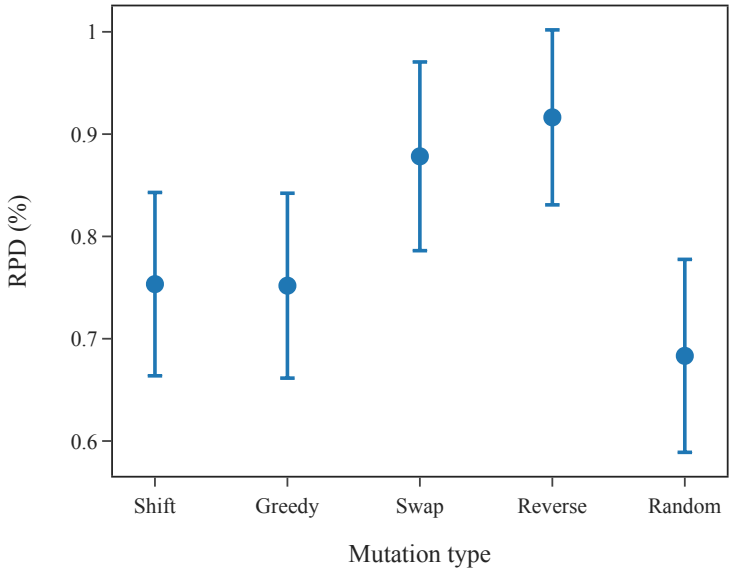


Figure 7.7: Average RPD for the different mutation operators.

The best option is to use them all by choosing a random one each time. This sits well with the intention of using mutations in the first place, which is to induce diversity. Potentially, more of the search space will be covered by switching the mutator. For these reasons, the random choice is included in the base case. Note also that the performance of mutations is tightly coupled with the mutation rate. The mutation rate is tested, and the results can be found in Appendix A. While the literature commonly uses mutation rates around 2%, these tests with a random mutation operator find the best choice to be 10%. It is the dominant option in most categorisations according to size.

An adaptive approach with Q-learning could have been applied for mutation operators as well. However, there is less difference in which mutation operator performs best for the different sized problem instances than for crossover operators. Since there is less difference, there is likely less of a performance enhancement to gain from an adaptive choice. Therefore, and to limit the number of parameters, Q-learning is not tested for mutations.

Local search

Contrary to the results of Murata et al. (1996); Ruiz and Maroto (2006); Ruiz et al. (2006); Yu et al. (2018), a local search does not improve the performance of our GA. The results are displayed in Figure 7.8. Here, 5, 50, and 500 iterations of IG have been tested after offspring is created through crossover and mutation. The lack of improvement could be caused by the local search seizing time from the other operators of the GA that use the time better. One iteration of IG takes significantly more time than one iteration of the crossover and mutation operators, except for BCBX and Greedy. The time complexity is similar to that of these operators. Another coinciding reason could be how IG destroys and rebuilds job permutations, similarly to what

BCBX and Greedy do. The effect of an extra search with the same concept as these operators is probably limited. None of the four aforementioned papers uses BCBX nor Greedy, which fits well with the hypothesis.

There appears to be a trend in Figure 7.8 indicating that more iterations of IG are better than fewer. Nevertheless, we refrain from increasing the number of iterations further as it could potentially dominate the GA. With 500 iterations, IG is already the most time-consuming operator of the GA. Notice that IG is tuned, and the results are presented in Section 7.3. Since the local search does not improve the GA, it is not included in the base case.

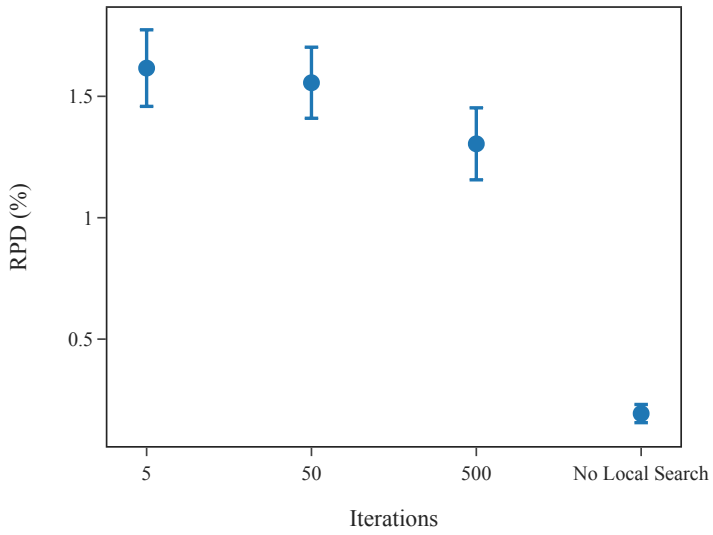


Figure 7.8: Average RPD for different number of iterations for the local search and no local search.

Crowding

The crowding methods are tuned before testing their impact on the GA. The results from each of these tests are found in Figure 11 and Figure 12 in Appendix A. It shows that both deviation distance crowding and exact match crowding are best with a crowding scale of zero and k-nearest of 20. This implies deterministic crowding and comparison to 20 neighbours. The results further indicate that increasing the number of neighbours could provide even better results. It is not tested as it would increase the selection pressure and undermine the rationale for crowding in the first place.

With tuning revealing that deterministic crowding with many neighbours dominates the alternatives, it is no surprise that no crowding is, in fact, better. This is the equivalent of increasing the neighbourhood size to equal the population size. Figure 7.9 shows how no crowding is better than the tuned crowding methods for all categories of problem instance sizes grouped by the number of jobs. It can be seen in Appendix A, Figure 13, that this is true with regards to the number of stages as well.

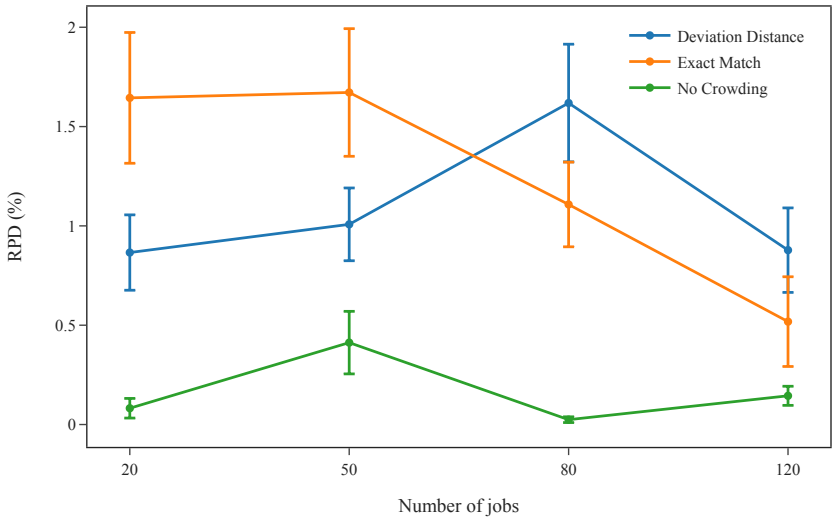


Figure 7.9: Average RPD for crowding with the two different distance metrics compared to no crowding, grouped by the number of jobs.

There are several reasons why crowding might not have the intended effect on the GA. One could be how crowding affects the replacement in the steady-state GA. Without crowding, the offspring replace the worst individuals in the population. With crowding, fitness is no longer the sole criteria for keeping offspring. Furthermore, tournament selection has a bias towards selecting fitter individuals as parents. The parents are likely to be similar to their offspring, and with probabilistic crowding, with few neighbours, the parents could be replaced by a less fit offspring. In every generation where this happens, the search can, in theory, regress. While this is the intention of crowding, it is built on the assumption of lacking diversity. If the diversity is satisfactory, there is no good reason to replace fit solutions with less fit ones.

Although crowding does not generally provide better results, it is apparent from Figure 7.9 that exact match crowding provides better results as the problem instances grow in the number of jobs. This might be due to the increase in the search space, for it might also increase the number of local optima. Then, crowding is likely to be more helpful in spreading the solutions across these local optima.

An important note regarding crowding is the testing conditions. The tests are run for an amount of time found to allow convergence without crowding. As crowding is used to delay convergence, it is natural that it does not improve the performance with limited run-time. These results could be very different with increased run-time. However, this thesis aims to solve the problem with limited time, so no changes are made, and crowding is not included in the base case.

Replacement

The final test examines if new individuals should replace a given portion of the population upon stagnation. The three different methods, *mutate*, *random*, and *GCH* are tested with a

replacement rate of 20%, as they are described in Section 6.10. That means the least fit 20% is replaced in case of stagnation. From Figure 7.10, it is apparent that the mutation scheme performs quite well compared to the others and is better than having no replacement scheme at all. This result aligns with many of the earlier tests indicating sufficient diversity. This could explain why introducing new randomly generated solutions has a negative impact. If the diversity is satisfactory, the only effect is to replace solutions close to local optima. Introducing new individuals with GCH could alleviate the problem by providing solutions close to local optima. However, creating new solutions with this heuristic is very time-consuming. The results are clear; the extra time it takes to generate fit replacement individuals is not worth it.

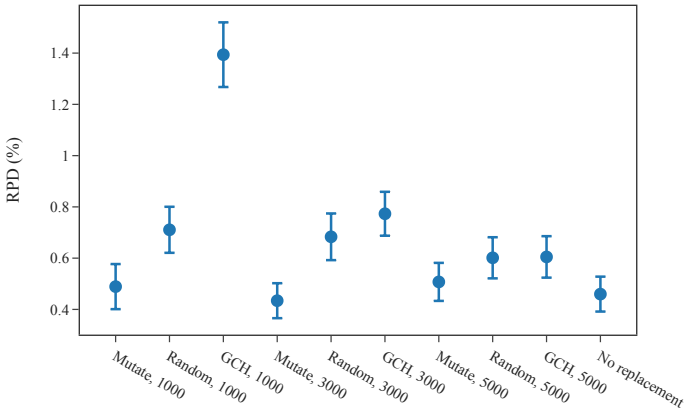


Figure 7.10: Average RPD for different replacement schemes compared to no replacement. The first axis shows the replacement scheme, as well as number of iterations without improvement in the best solution before replacement.

In order to determine how large the portion of the population should be replaced, a separate test is performed. The results from this test is found in Figure 15 in Appendix A. The best results are found by replacing 20 % of the population with the *mutate* scheme in the case of 3000 iterations without improvement to the best solution. Here, replacing a higher percentage of the population might introduce too many random solutions. In addition, it might remove too many of the high-quality individuals the GA has spent a considerable amount of time generating.

Summary

The final values for all parameters after tuning the GA are summarised in Table 7.3. These parameters are used to compare the GA with its benchmarks in the following sections. As previously mentioned, there might be a bias towards the values we introduced as the base case at the start of tuning. An example of this is the replacement rate of 20% used when testing replacement type. Other replacement types might have proven more effective if a replacement rate of 80 % was used. This is listed as a weakness of the experimental design of these tests. Nevertheless, running the GA with a combination of all parameter values is considered too time-consuming and makes it more challenging to analyse and discuss results for the individual parameters. The limited time also favours GAs that converge within the allocated time, and changing the time might give other results.

Table 7.3: Summary of the final parameter values in the base case for the GA.

Parameter	**Value**
Makespan calculation	FIFO
Generational scheme	Steady-state
Initialisation	GCH (100 %)
Population size	150
Tournament size	2
Crossover	Q-learning
Learning rate	0.2
Epsilon	0.25
Mutation	Random
Mutation probability	10 %
Local search	No
Crowding	No
Replacement type	Mutate
Replacement iterations	3000
Replacement rate	20 %

7.3 Tuning of Iterated Greedy

In order to make a fair comparison, the IG algorithm is also tuned. IG consists of two parameters, making for a less comprehensive tuning. The parameters are the temperature and the number of jobs removed from the sequence of jobs for each iteration. The results are displayed in Figure 7.11. It is clear that setting $d = 2$ consistently performs better than all other values for d. The temperature is less important, but a value of $T = 0.5$ has the lowest RPD and is carried forward.

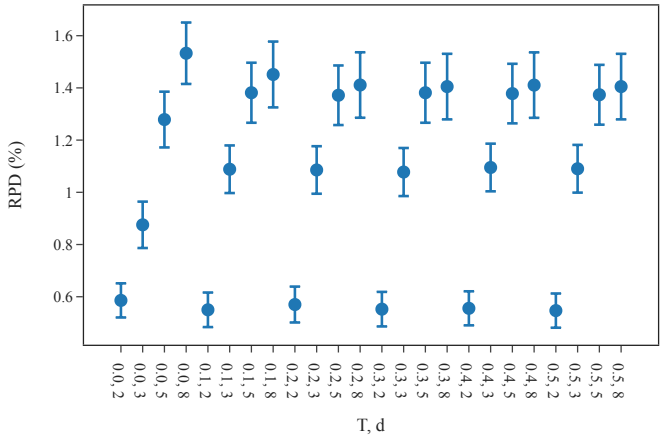


Figure 7.11: Average RPD for different values of temperature and number of jobs in IG.

7.4 Convergence Tests

Before comparing GA to IG on all 960 problem instances, it is vital to understand how much time it takes for them to converge. Knowing their convergence allows allocating a reasonable amount of time for a fair comparison. It is also interesting to see how the best solutions improve over time, as it indicates how the stages of the algorithms operate.

The GA and IG solve ten problem instances of 20, 50, 80, and 120 jobs, all with eight stages to test maximum run time. The makespan value of the best solution in every iteration is recorded for both algorithms. Note that the GA performs no iterations before the initial population is generated. All makespan values before the first iteration of the GA are set to the makespan of the best schedule of the initial population. We do this to indicate what time is spent initialising and what is spent iterating. The results are averaged over the ten instances and displayed in Figure 7.12. The second axis is denoted by the percentage of the makespan of any of the two algorithms' first, worst, solution. With its NEH starting position, IG's first iteration is the worst starting point in all cases.

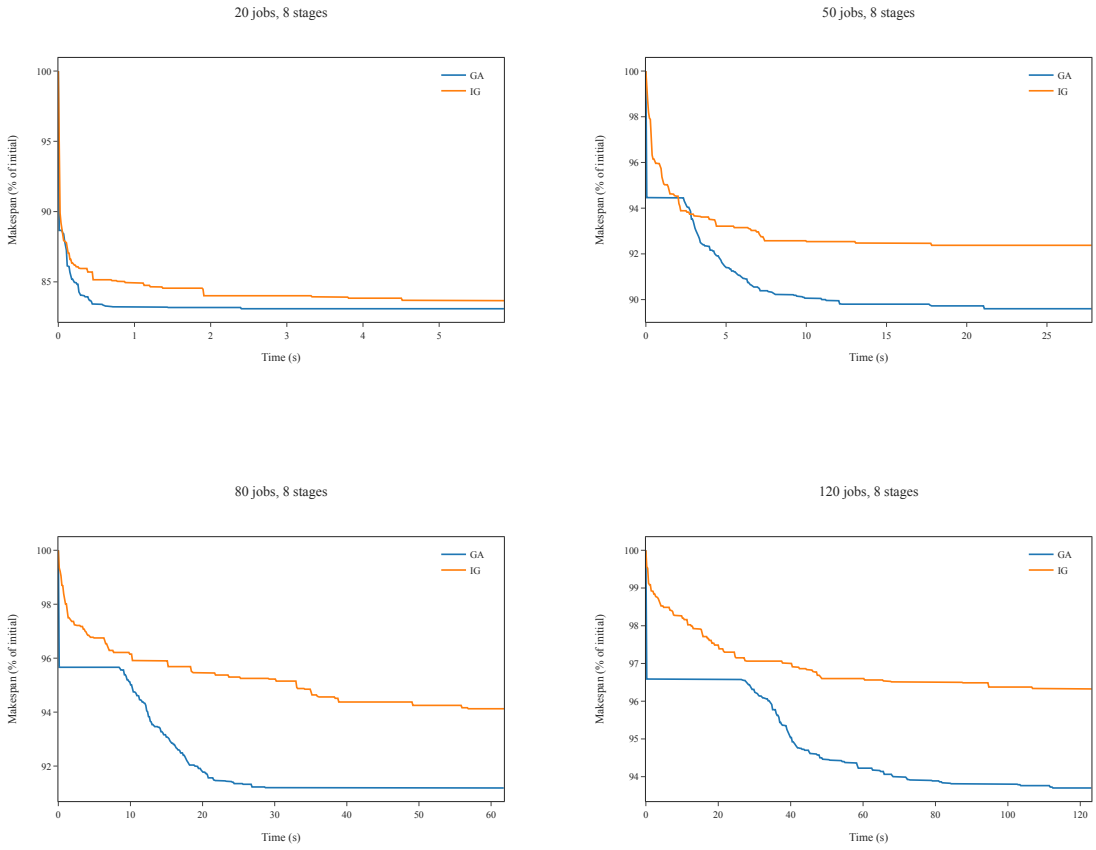


Figure 7.12: Convergence tests for the GA and IG for different number of jobs.

Again, we vary the allocated time according to the number of jobs and stages of problem instances. Thus, the values along the first axis vary in the different graphs in Figure 7.12. The

GA and IG seem to converge well within the time provided for all problem instance sizes. IG generally improves rapidly over the first iterations and converges quicker than the GA for 20, 50, and 120 jobs. For problem instances with 80 jobs, IG seems to converge slower than the GA. Ultimately, the results show the algorithms have minor improvements after about 2/3s of the time. For this reason, Equation 7.4, allocating about 2/3s of the time used in these tests, are used for their final comparison.

$$\text{Time } (ms) = |\mathcal{N}|^{1.7} \cdot |\mathcal{I}| \cdot 3.0 \tag{7.4}$$

Another observation is how the GCH consistently provides better initial solutions to the GA than NEH provides IG. However, NEH only provides one solution. GCH has 150 chances of producing a better solution, as there are 150 individuals in the initial population. Nevertheless, this emphasises the importance of generating quality initial solutions in the GA.

7.5 Performance Tests

The algorithms are compared performance-wise with the tuned parameter values and the appropriate run-time. First, however, the GA is compared with the results from the mathematical model in Section 7.5.1. Then, in Section 7.5.2, the performance of the four solution methods are evaluated.

7.5.1 Comparison with Mathematical Model

In this section, the solutions from the GA are compared to the solutions from the mathematical model introduced in Chapter 4. Since the mathematical model is not the main focus of this thesis, not much effort has been put into further improving the model by introducing symmetry-breaking constraints, lower-bound calculations, or any other enhancements.

As the mathematical model requires significantly more memory than the GA, it is run on *Solstorm*. Solstorm is a shared cluster provided by the Department of Industrial Economics and Technology Management at the Norwegian University of Science and Technology. The specifications of the hardware and software are listed in Table 7.4. The model is implemented in Gurobi through their Python interface. The problem is very complex, so only the smallest problem instances of 20 jobs and two stages are used. Five problem instances are tested, and each of those is run for 48 hours.

Table 7.4: Hardware and software specifications for the implementation of the mathematical model.

Processor	2.2GHz AMD Opteron
Memory	128GB RAM
CPU Cores	16
Operating System	CentOS 7.9
Python version	3.9.6
Gurobi version	9.1.2

In Table 7.5, information on the running of the mathematical model is listed along with comparable information on the GA. These results show that the GA outperforms the mathematical model in solution quality and efficiency. The GA sees an average improvement of 8.2 % over the mathematical model, with a fraction of the time. Moreover, the mathematical model cannot close the gap between the lower bound and the best integer solution. In the end, the average gap is still 95.7 %.

Table 7.5: Comparison between the mathematical model and GA.

	Mathematical model			**Genetic algorithm**	
Problem (#)	Makespan	Gap (%)	Time (s)	Makespan	Time (s)
1	651	95.4 %	172 800	612	0.5
2	641	95.5 %	172 800	627	0.5
3	612	97.5 %	172 800	568	0.5
4	556	95.0 %	172 800	522	0.5
5	484	95.2 %	172 800	375	0.5
Average	**588.8**	**95.7 %**	**172 800**	**540.8**	**0.5**

The convergence of the mathematical model's upper and lower bound is abysmal, as Figure 7.13 displays. The results of the five instances, also displayed in Table 7.5, are averaged for every second and plotted.

The results support the findings from Gravdal and Weidemann (2021). The authors found that the mathematical model of a similar problem can only solve tiny problem instances to optimality in a reasonable amount of time. A high number of constraints, considerable amounts of symmetry in solutions, and slack due to several Big M-constraints were pointed out as the most prominent limitations. This is also likely to be the reason for this implementation's poor performance. The lower performance of exact methods emphasises the use of approximate methods in this field of research.

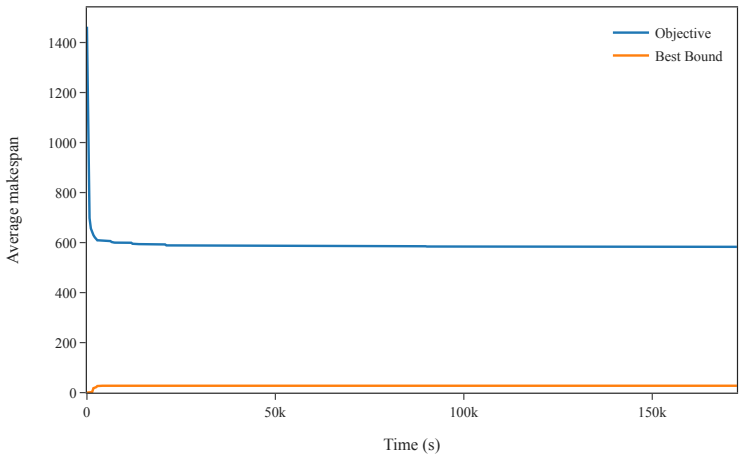


Figure 7.13: Improvement in best integer solution and lower bound averaged over the five problem instances for the mathematical model.

7.5.2 Comparing Solution Methods

This section presents the results from running each of the four solution methods introduced in Chapter 5 for all 960 instances. Once again, the average RPD is used to compare the algorithms. The results are displayed as a scatter plot in Figure 7.14. The average RPDs indicate that the construction heuristics perform notably worse than the meta-heuristics, which are relatively equal in performance compared to the construction heuristics. It is important to note that both meta-heuristics start with initialising solutions comparable to the construction heuristics and then perform an additional search.

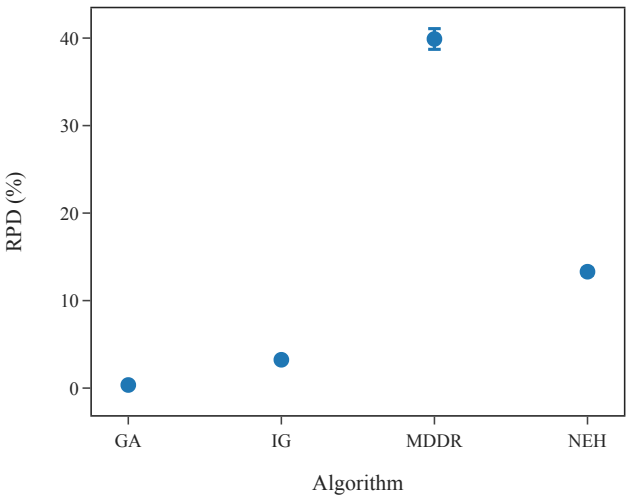


Figure 7.14: Comparison of all the algorithms implemented, showing the average RPD for each of them.

MDDR displays worse performance than what the literature suggests it would (Naderi et al., 2010). The reason for the deviation can be twofold. First, computers have improved significantly. The meta-heuristics might get to perform more iterations in the same amount of time, meaning that implementing a meta-heuristics is getting more powerful. The second reason could be that the GA proposed in this thesis performs better than the peers used in Naderi et al. (2010). Thus, MDDR likely performs worse relative to the algorithms it is compared to in this thesis, and not in general to how it performed before. The beginning of Section 7.2.2 discusses why the first-completion makespan calculation procedure is dominated performance-wise by FIFO. The same reasoning can explain the lack of performance in MDDR. The schedules it provides often contain too much idle time in the machines. However, this is primarily a problem when production and set-up times vary widely in the problem instances, and this will not always be the case. Table 7.6 gives a different picture than the scatter plot altogether. MDDR finds the best solutions for 30 problem instances out of any method, while NEH finds only one. This can be explained by the increasing search space affecting MDDR to a lesser degree, as it is a simple assignment rule. On the other hand, the meta-heuristics will perform worse in comparison.

Table 7.6: Number of best solutions found by each algorithms, grouped by number of jobs.

Number of jobs	GA	IG	MDDR	NEH
20	143	140	0	1
50	207	29	2	0
80	215	25	9	0
120	211	10	19	0
Total	**776**	**204**	**30**	**1**

Looking at the performance grouped by categories based on the size of the problem instances in Figure 7.15 provides more insight. The difference in performance between the construction and improvement heuristics decreases as the number of jobs increases. This can be due to the iteration-based improvement heuristics running for fewer iterations. These are more affected by the problem instances increasing in size than the construction heuristics. Therefore, it could be assumed that the meta-heuristics perform worse, not the other way around. With an increase in stages, the opposite trend can be observed. With more stages, the initial choices made in the first stage have more time to manifest themselves. As the meta-heuristics experiment with different orders, they can potentially repair initially bad choices that the construction heuristics cannot. In addition, since the solution representation only consists of the order of jobs in the first stage, the number of stages likely affects the search less than the number of jobs for the meta-heuristics. To sum up, an increase in the number of jobs is to the relative advantage of the construction heuristics, whereas an increase in the number of stages is to the relative advantage of the meta-heuristics.

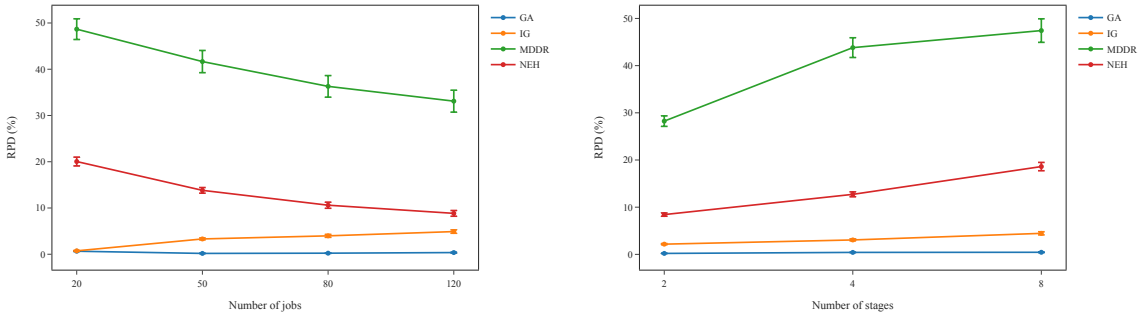


Figure 7.15: Average RPD for all algorithms for different number of jobs and stages, respectively.

Although the two meta-heuristics, IG and GA, perform quite similarly relative to the construction heuristics, there are notable differences between these. IG has an average RPD of 3.24%, whereas the GA has an average RPD of 0.35%. Moreover, the proposed GA has the lowest average RPD in all size categories of the problem instances. It also finds the best solutions in over 80% of the problem instances and, as discussed in Section 7.4, the convergence of the two meta-heuristics is also very similar. The GA dominates the other solution methods except for problem instances with 20 jobs.

In conclusion, it is not surprising that the more sophisticated improvement heuristics provide the best results. The MDDR and NEH heuristics find better solutions than the two improvement heuristics a few times but perform considerably worse overall. Of the two improvement heuristics, the GA finds the best solutions, particularly when the instances grow in size.

Chapter 8

Concluding Remarks

This thesis aims to introduce a solution method for finding high-quality production schedules for small manufacturing businesses. The production environment considered is inspired by Haugstad Furniture Factory (Haugstad). The scheduling problem considers two major decisions. First, each product needs to be allocated to a machine for every required process. Second, every process's start and finishing times need to be determined. All production is assumed to be deterministic and according to the flowshop production environment. The objective is to minimise the makespan; the total time of production.

The flowshop scheduling problem (FSP) is well-studied, but this is not the case for many of its extensions and the combinations of those. This includes the hybrid flexible flowshop scheduling problem with sequence-dependent set-up times (HFFSP SDST). This flowshop variant resembles the characteristics of Haugstad. The variant has the limitations of any flowshop; a unidirectional flow of products through stages where machines are grouped by function. Moreover, being both *hybrid* and *flexible*, there are several machines in certain stages and products are allowed to skip stages, respectively. Because of differences in products, all machines need to be prepared before processing. This operation is dependent on the current settings on the machine, making the set-up sequence-dependent.

A mathematical model and several non-exact methods are implemented to solve the problem. Two construction heuristics are implemented as benchmarks. These also provide starting solutions for the meta-heuristics. Iterated Greedy is implemented as it has proved effective in literature and represents a simple meta-heuristic. Finally, we introduce a genetic algorithm (GA) inspired by methods used for similar FSPs, other permutation problems, and popular methods for GAs in general. All solution methods are tested on a set of publicly available problem instances also used by other researchers.

Few studies have used GAs as a solution method for the HFFSP SDST. The GA proposed in this thesis introduces several new components to the problem and combinations yet to be tested. This includes two novel crossover and mutation operators using domain knowledge to improve

the individuals, an altered construction heuristic for generating fit initial populations, and an adaptive crossover selection mechanism based on Q-learning.

The mathematical model introduced in this thesis is dominated by approximation methods in terms of both run-time and absolute performance, which is in line with the previous study of Gravdal and Weidemann (2021).

Among the non-exact solution methods, the GA finds the best schedules in 776 out of 960 problem instances, dominating the benchmarks. Moreover, several of the introduced components outperform those presented in previous studies. While some are more effective in niche cases, others improve the algorithm across all problem instance sizes. Most notably, the adaptive selection of crossover operators takes advantage of the former, as the different operators have specific problem instance sizes at which they excel. The adaptive selection found the most suitable operator and performed better than applying one specific or randomly choosing one every iteration.

We have reasons to believe the GA introduced in this thesis is suited well to Haugstad. The GA finds efficient production schedules for problem instances of 120 products, eight stages, and up to four machines in each stage in less than two minutes. These tests are performed on inexpensive hardware that Haugstad can access.

Finally, we note that the introduced GA is likely to have some merit for other variants of the flowshop scheduling problem. As long as the solution representation can be represented as the order of jobs in the first stage and suitable assignment rules are established, the operators introduced in this thesis are likely to produce good solutions. The GA could probably work well for other similar factories and larger problem instances.

Chapter 9

Future Work

The genetic algorithm (GA) shows promising results for the hybrid flexible flowshop scheduling problem with sequence-dependent set-up times (HFFSP SDST). However, there is still room for improvement. These areas include the operators of the GA, the problem it is applied to, and the data used to test it. In the following paragraphs, aspects of each of these areas are discussed.

Concepts that did not have the intended effect could be looked into and possibly improved. For instance, the local search was intended to improve the new offspring but proved ineffective. While it was dropped from further consideration in this thesis, applying it with more care might prove more successful. A potential drawback with our implementation is that it spends the same amount of time on all new solutions, including the unfit. The local search could be applied only to new individuals with makespan values within a certain percentage of the current best individual's makespan. Moreover, it could be interesting to apply more local searches at the start or end phase of the search. Applying local search at the start could provide the GA with an even larger head start than the Greedy Construction Heuristic can alone. Applying it at the end could improve the best solutions to increase the GA's absolute performance.

Another concept that failed to show the intended results is crowding. A significant drawback with the k-nearest implementation is that individuals similar to fit offspring are likely also to be fit. Thus, fit solutions are replaced, whereas low-quality solutions that are different are kept. An alternative idea is: Instead of including the k nearest individuals to the offspring in the tournament, it is possible to include the k least fit individuals. Then, let the most similar among these k individuals compete for its place in the next generation.

Introducing a meta-model to replace the makespan procedure and quickly approximate an individual's fitness could speed up the search. The rationale for such a component is that determining fitness is the procedure that takes up the most time for both the GA and Iterated Greedy (IG). Furthermore, the number of iterations is tightly coupled with the aptness of the resulting schedule. Hopefully, by speeding up the fitness evaluation and performing more

iterations, the schedules will be better. A neural network could be a suitable structure for a meta-model as it is quick once trained and does not have to be 100% accurate.

The assignment of jobs to machines is another area of potential improvement. As of now, jobs are assigned to the machine that can finish the job first. Such a simple dispatching rule can miss out on many high-quality solutions. Instead, an agent could be used to assign jobs to machines and improve to find a more effective assignment rule. This agent could be implemented with a reinforcement learning framework. The reward could be the makespan, which could be discounted for every stage.

The HFFSP SDST incorporates extensions to make the modelling more realistic, but several aspects are still not covered. Examples are uncertainty in processing times, machine breakdowns, deadlines, and multiple objectives. Uncertainty in processing times could be handled by working with distributions, whereas machine breakdowns could be modelled by a predetermined chance for every machine to break down during production. These changes would require a change in objective to *expected* makespan. Deadlines are also very relevant and probe yet another change in the objective. Tardiness, or total delay of products, is a better performance measure in such a scenario. Finally, GAs are potent in finding the Pareto front in multiple objective problems. Potentially interesting additional objectives are power consumption, raw material use, and employee welfare.

All tests are performed on artificially generated data. It should be tested in the production environment with real data to determine the actual value of the algorithm to a manufacturer. Such data is difficult to collect and systematise, but tuning the GA to the new requirements is a relatively small task. As a final test, schedules from manual planning or other existing planning systems could be used as benchmarks to determine the potential improvement.

Bibliography

Campbell, H. G., Dudek, R. A., and Smith, M. L. (1970). A heuristic algorithm for the n job, m machine sequencing problem. *Management Science*, 16(10):B–630.

Chen, B., Potts, C. N., and Woeginger, G. J. (1998). A review of machine scheduling: Complexity, algorithms and approximability. *Handbook of Combinatorial Optimization*, pages 1493–1641.

Cicirello, V. A. (2019). Classification of permutation distance metrics for fitness landscape analysis. In *International Conference on Bio-inspired Information and Communication*, pages 81–97. Springer.

Colak, M. and Keskin, G. A. (2022). An extensive and systematic literature review for hybrid flowshop scheduling problems. *International Journal of Industrial Engineering Computations*, 13:185–222.

Dannenbring, D. G. (1977). An evaluation of flow shop sequencing heuristics. *Management Science*, 23(11):1174–1182.

Defersha, F. M. and Chen, M. (2012). Mathematical model and parallel genetic algorithm for hybrid flexible flowshop lot streaming problem. *The International Journal of Advanced Manufacturing Technology*, 62(1-4):249–265.

Eiben, A. and Smith, J. (2015). Evolutionary computing: The origins. In *Introduction to Evolutionary Computing*, pages 13–24. Springer.

Engin, O., Ceran, G., and Yilmaz, M. K. (2011). An efficient genetic algorithm for hybrid flow shop scheduling with multiprocessor task problems. *Applied Soft Computing*, 11(3):3056–3065.

Fan, K., Zhai, Y., Li, X., and Wang, M. (2018). Review and classification of hybrid shop scheduling. *Production Engineering*, 12(5):597–609.

Farahmand-Mehr, M., Fattahi, P., Kazemi, M., Zarei, H., and Piri, A. (2014). An efficient genetic algorithm for a hybrid flow shop scheduling problem with time lags and sequence-dependent setup time. *Manufacturing Review*, 1:21.

Fattahi, P., Hosseini, S. M. H., and Jolai, F. (2013). A mathematical model and extension algorithm for assembly flexible flow shop scheduling problem. *The International Journal of Advanced Manufacturing Technology*, 65(5):787–802.

Fernandez-Viagas, V. and Framinan, J. M. (2015). Neh-based heuristics for the permutation flowshop scheduling problem to minimise total tardiness. *Computers & Operations Research*, 60:27–36.

Galan, S. F. and Mengshoel, O. J. (2010). Generalized crowding for genetic algorithms. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 775–782.

Gantt, H. L. (1903). A graphical daily balance in manufacture. *ASME Transactions*, 24:1322–1336.

Garey, M. R., Johnson, D. S., and Sethi, R. (1976). The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1(2):117–129.

Gmys, J., Mezmaz, M., Melab, N., and Tuyttens, D. (2020). A computationally efficient branch-and-bound algorithm for the permutation flow-shop scheduling problem. *European Journal of Operational Research*, 284(3):814–833.

Goldberg, D. E., Lingle, R., et al. (1985). Alleles, loci, and the traveling salesman problem. In *Proceedings of an international conference on genetic algorithms and their applications*, volume 154, pages 154–159. Lawrence Erlbaum Hillsdale, NJ.

Gómez-Gasquet, P., Andrés, C., and Lario, F.-C. (2012). An agent-based genetic algorithm for hybrid flowshops with sequence dependent setup times to minimise makespan. *Expert Systems with Applications*, 39(9):8095–8107.

González-Neira, E., Montoya-Torres, J., and Barrera, D. (2017). Flow-shop scheduling problem under uncertainties: Review and trends. *International Journal of Industrial Engineering Computations*, 8(4):399–426.

Gravdal, H. I. and Weidemann, J. (2021). Hybrid flexible flowshop scheduling in a norwegian manufacturing factory (unpublished). Technical report, Norwegian University of Science and Technology.

Gupta, J. N. and Stafford, E. F. (2006). Flowshop scheduling research after five decades. *European Journal of Operational Research*, 169(3):699–711.

Herrmann, J. W. (2006). A history of production scheduling. In *Handbook of production scheduling*, pages 1–22. Springer.

Ho, J. C. and Chang, Y.-L. (1991). A new heuristic for the n-job, m-machine flow-shop problem. *European Journal of Operational Research*, 52(2):194–202.

Hosseinabadi, A. A. R., Vahidi, J., Saemi, B., Sangaiah, A. K., and Elhoseny, M. (2019). Extended genetic algorithm for solving open-shop scheduling problem. *Soft Computing*, 23(13):5099–5116.

Ignall, E. and Schrage, L. (1965). Application of the branch and bound technique to some flow-shop scheduling problems. *Operations Research*, 13(3):400–412.

Johnson, S. M. (1954). Optimal two-and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly*, 1(1):61–68.

Mengshoel, O. J. and Goldberg, D. E. (2008). The crowding approach to niching in genetic algorithms. *Evolutionary Computation*, 16(3):315–354.

Ministry of Trade Industry and Fisheries (2017). Industrien – grønnere, smartere og mer nyskapende.

Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117.

Murata, T., Ishibuchi, H., and Tanaka, H. (1996). Genetic algorithms for flowshop scheduling problems. *Computers & Industrial Engineering*, 30(4):1061–1071.

Naderi, B., Ruiz, R., and Zandieh, M. (2010). Algorithms for a realistic variant of flowshop scheduling. *Computers & Operations Research*, 37(2):236–246.

Nawaz, M., Enscore Jr, E. E., and Ham, I. (1983). A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *Omega*, 11(1):91–95.

Ombuki-Berman, B. and Hanshar, F. T. (2009). Using genetic algorithms for multi-depot vehicle routing. In *Bio-inspired algorithms for the vehicle routing problem*, pages 77–99. Springer.

Ozsoydan, F. B. and Sağir, M. (2021). Iterated greedy algorithms enhanced by hyper-heuristic based learning for hybrid flexible flowshop scheduling problem with sequence dependent setup times: a case study at a manufacturing plant. *Computers & Operations Research*, 125:105044.

Papadimitriou, C. H. and Kanellakis, P. C. (1980). Flowshop scheduling with limited temporary storage. *Journal of the ACM (JACM)*, 27(3):533–549.

Pinedo, M. (2012). *Scheduling*, volume 29. Springer, 5. edition.

Rossit, D. A., Tohmé, F., and Frutos, M. (2018). The non-permutation flow-shop scheduling problem: a literature review. *Omega*, 77:143–153.

Ruiz, R. and Maroto, C. (2005). A comprehensive review and evaluation of permutation flowshop heuristics. *European Journal of Operational Research*, 165(2):479–494.

Ruiz, R. and Maroto, C. (2006). A genetic algorithm for hybrid flowshops with sequence dependent setup times and machine eligibility. *European Journal of Operational Research*, 169(3):781–800.

Ruiz, R., Maroto, C., and Alcaraz, J. (2006). Two new robust genetic algorithms for the flowshop scheduling problem. *Omega*, 34(5):461–476.

Ruiz, R., Şerifoğlu, F. S., and Urlings, T. (2008). Modeling realistic hybrid flexible flowshop scheduling problems. *Computers & Operations Research*, 35(4):1151–1175.

Ruiz, R. and Stützle, T. (2007). A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. *European Journal of Operational Research*, 177(3):2033–2049.

Ruiz, R. and Vázquez-Rodríguez, J. A. (2010). The hybrid flow shop scheduling problem. *European Journal of Operational Research*, 205(1):1–18.

Simon, D. (2013). *Evolutionary optimization algorithms*. John Wiley & Sons, 2. edition.

Sioud, A., Gagné, C., and Gravel, M. (2014). Metaheuristics for solving a hybrid flexible flowshop problem with sequence-dependent setup times. In *International Conference on Swarm Intelligence Based Optimization*, pages 9–25. Springer.

Sioud, A., Gravel, M., and Gagné, C. (2013). A genetic algorithm for solving a hybrid flexible flowshop with sequence dependent setup times. In *2013 IEEE Congress on Evolutionary Computation*, pages 2512–2516. IEEE.

SSB (2020). Arbeidskraftundersøkelsen, tabell 09788: Sysselsatte. Årsgjennomsnitt, etter statistikkvariabel, kjønn, yrke og år.

Stoop, P. P. and Wiers, V. C. (1996). The complexity of scheduling in practice. *International Journal of Operations & Production Management*, 16(10):37–53.

Suliman, S. (2000). A two-phase heuristic approach to the permutation flow-shop scheduling problem. *International Journal of Production Economics*, 64(1-3):143–152.

Suzić, N., Forza, C., Trentin, A., and Anišić, Z. (2018). Implementation guidelines for mass customization: current characteristics and suggestions for improvement. *Production Planning & Control*, 29(10):856–871.

Taylor, F. W. (1919). *The principles of scientific management*. Harper & brothers.

Wang, H. (2005). Flexible flow shop scheduling: optimum, heuristics and artificial intelligence solutions. *Expert Systems*, 22(2):78–85.

Watkins, C. J. and Dayan, P. (1992). Q-learning. *Machine Learning*, 8(3):279–292.

Yu, C., Semeraro, Q., and Matta, A. (2018). A genetic algorithm for the hybrid flow shop scheduling with unrelated machines and machine eligibility. *Computers & Operations Research*, 100:211–229.

Appendix

In this appendix, the results from all tests performed in the computational study is found. There are three different types of graphs for each test. All of them display the *relative percentage deviation* (RPD) averaged over all problem instances tested. The RPD for the test run for a specific problem instance is defined as in Equation 1:

$$RPD = \frac{Alg_{sol} - Best_{sol}}{Best_{sol}} \cdot 100\% \tag{1}$$

In Equation 1, Alg_{sol} refers to the makespan of a given algorithm or parameter combination, whereas $Best_{sol}$ refers to the best makespan achieved by any of the competing algorithms. The error bars in the graphs are the *standard error of the mean* (SEM), as defined in Equation 2. In this equation, σ refers to the standard deviation of the RPDs and n to the number of instances.

$$SEM = \frac{\sigma}{\sqrt{n-1}} \tag{2}$$

For each set of graphs from a particular test, the first graph shows the average RPD and SEM across all sized problem instences. The second graph shows how the response variables vary with the number of jobs. Finally, the third graph shows variations across the number of stages.

A Tuning of Genetic Algorithm

Makespan Procedures

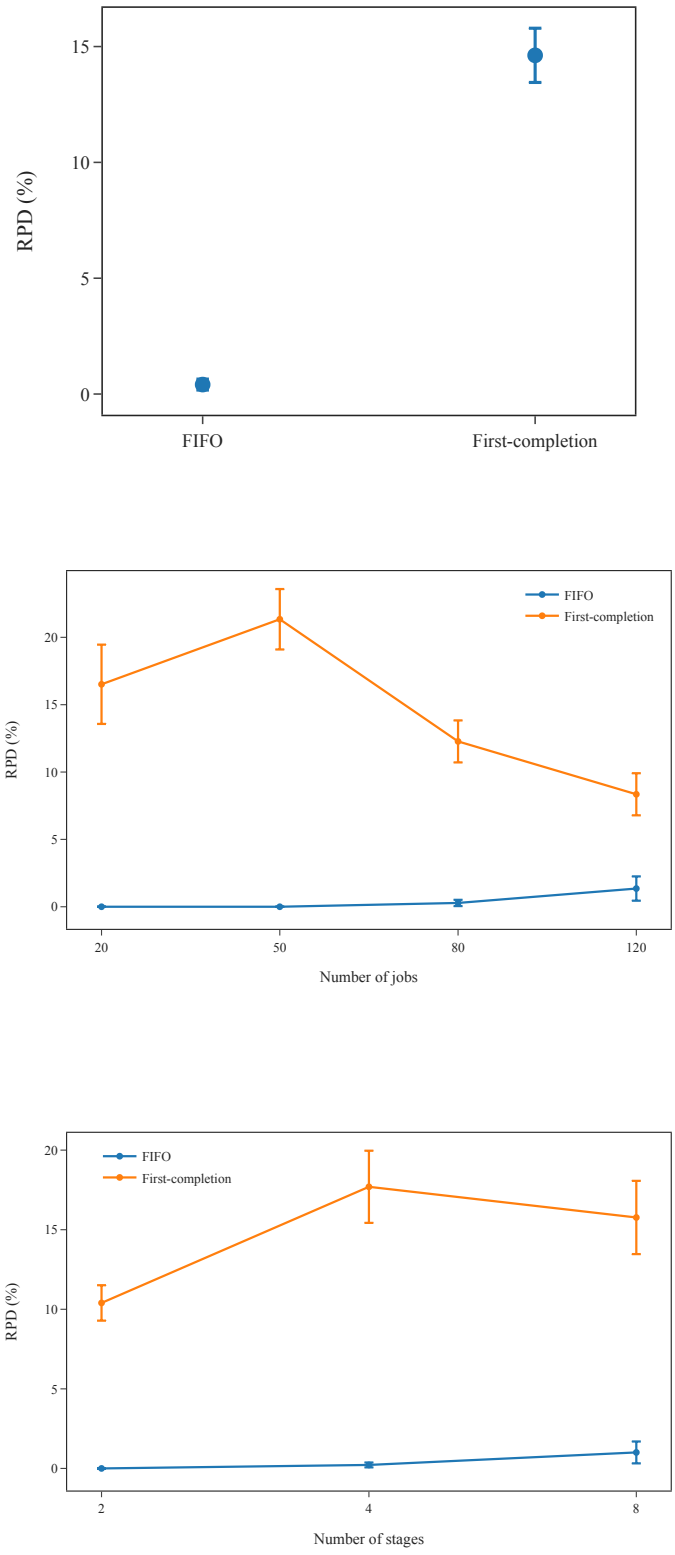


Figure 1: Makespan procedures.

Generational Scheme

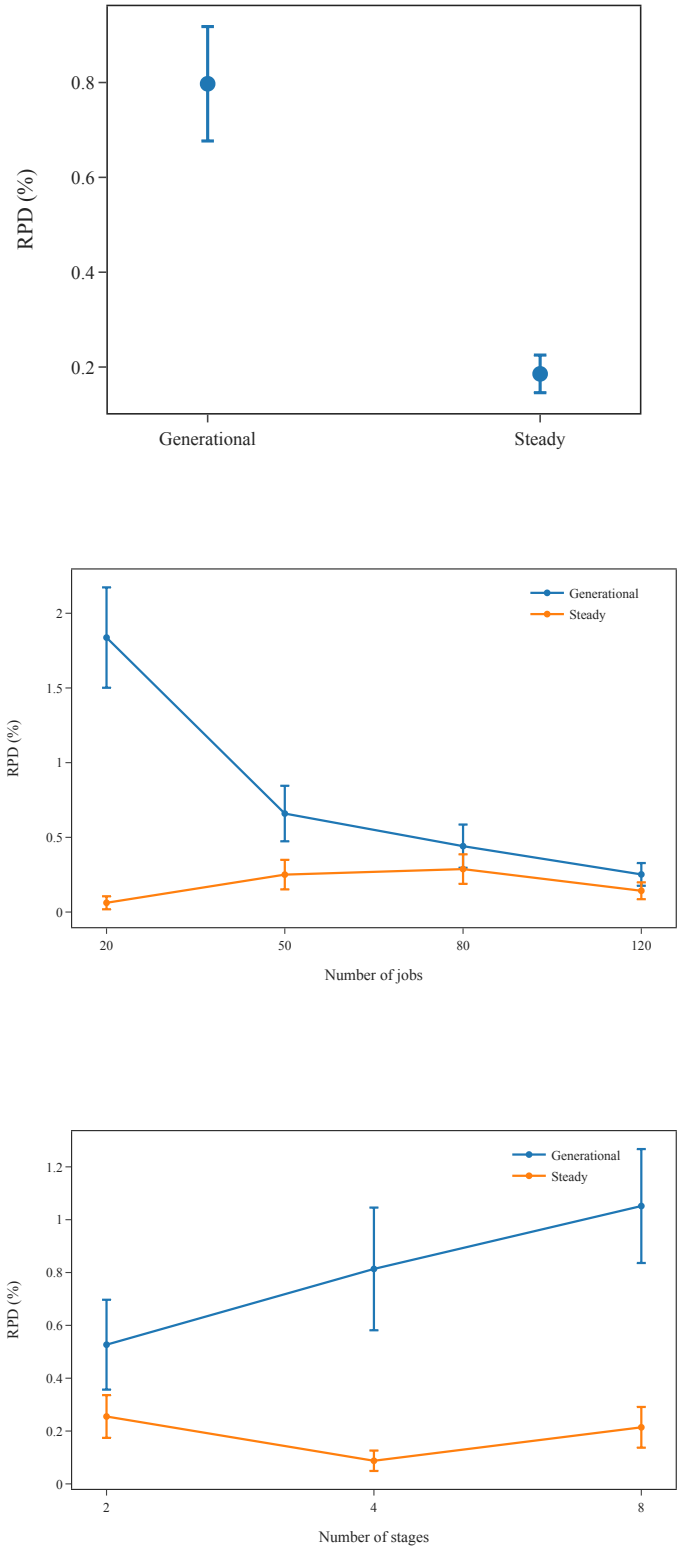


Figure 2: Generational scheme.

Initialisation

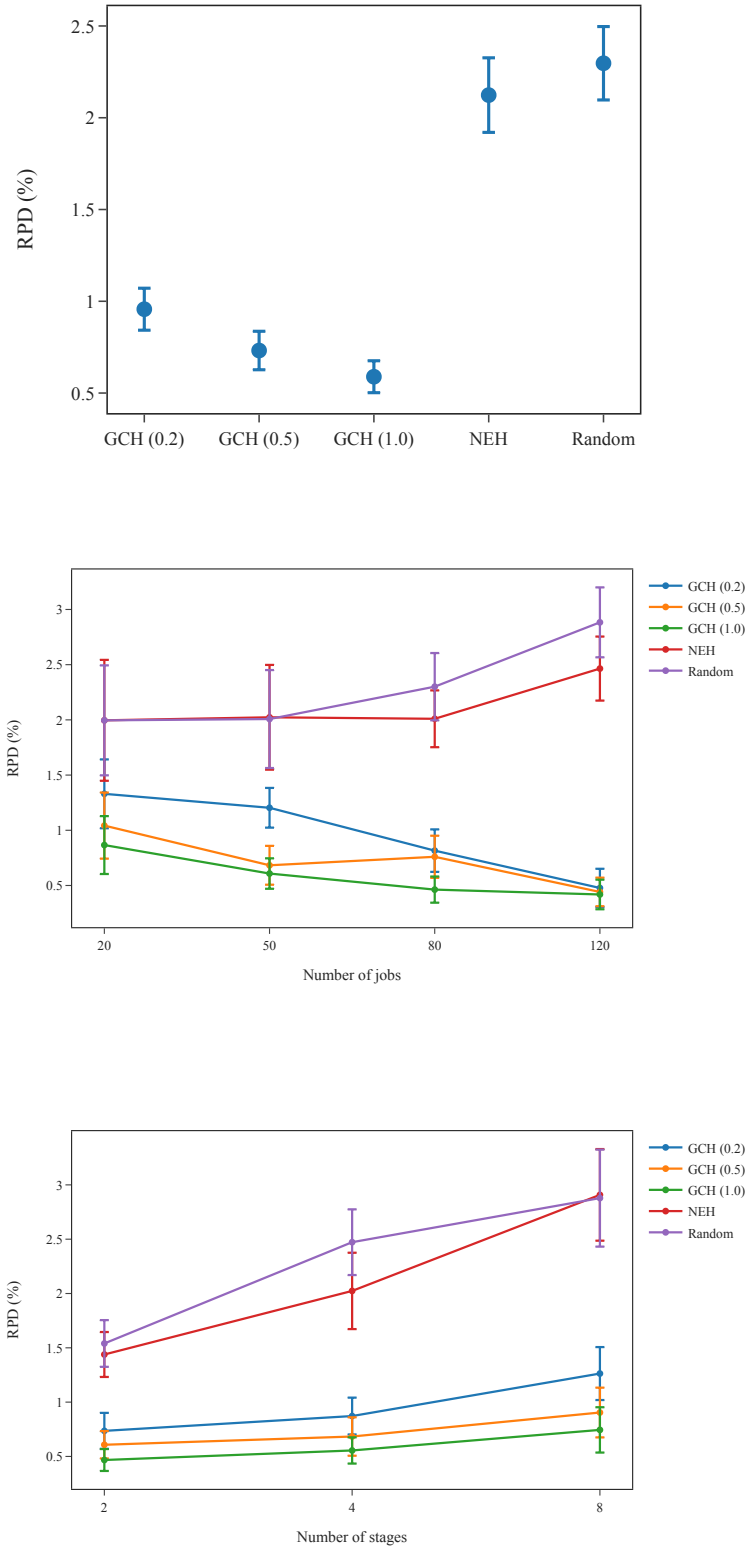
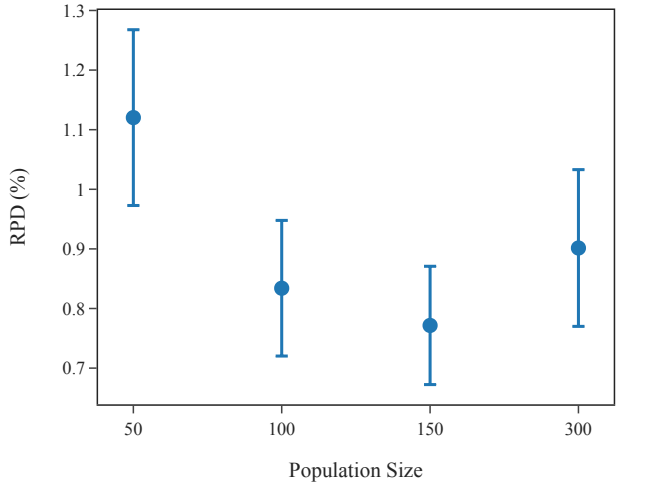
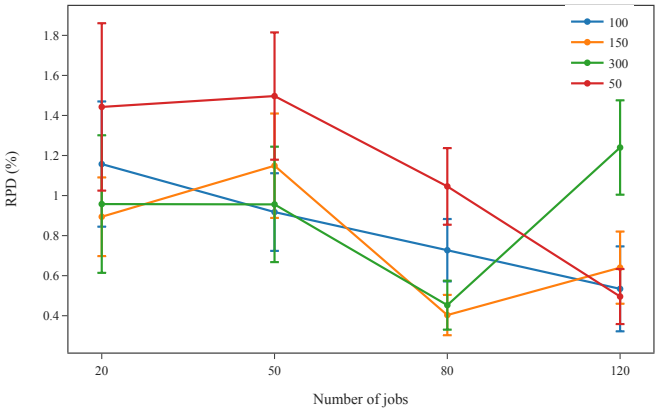


Figure 3: Initialisation.

Population Size





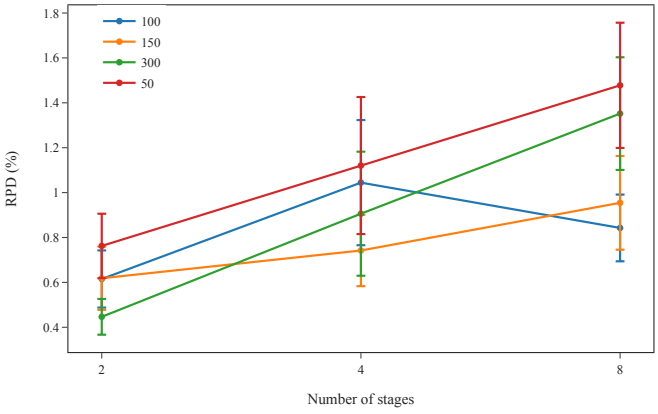


Figure 4: Population size.

Tournament Size

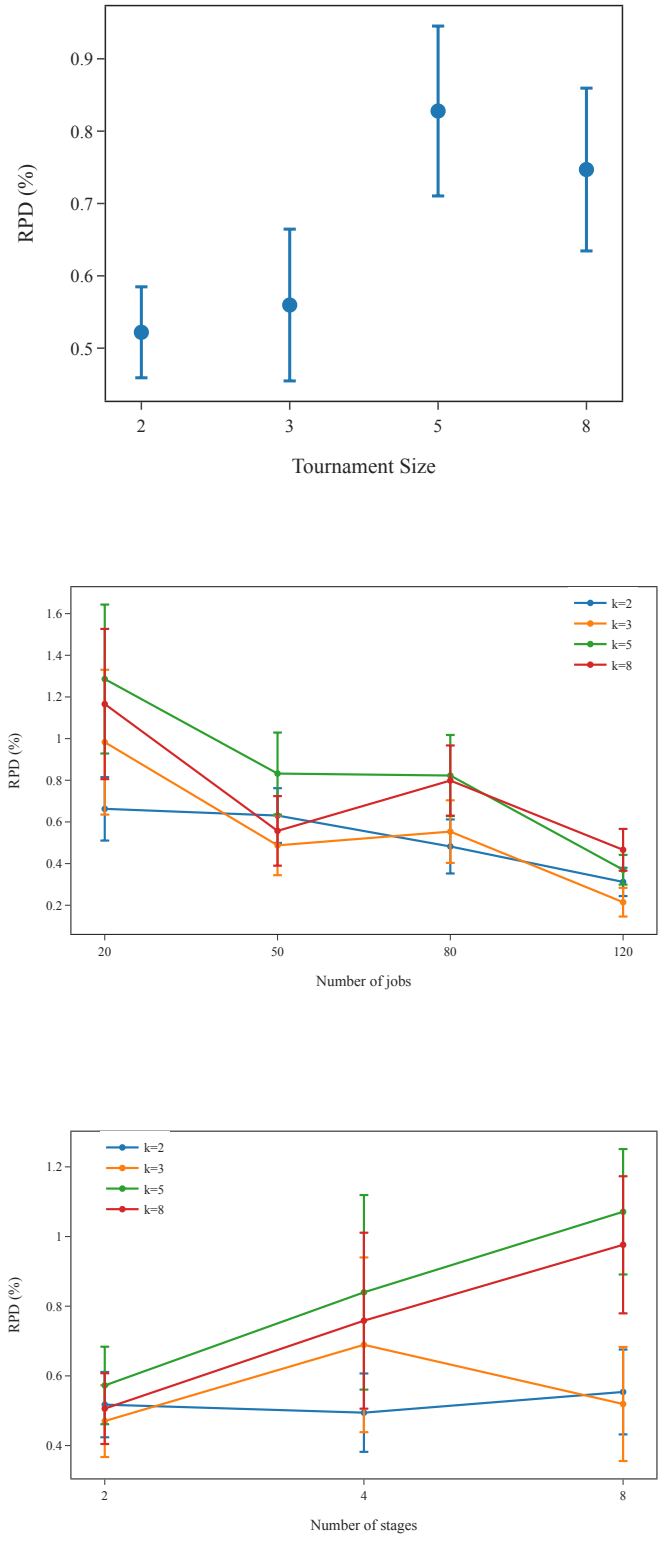
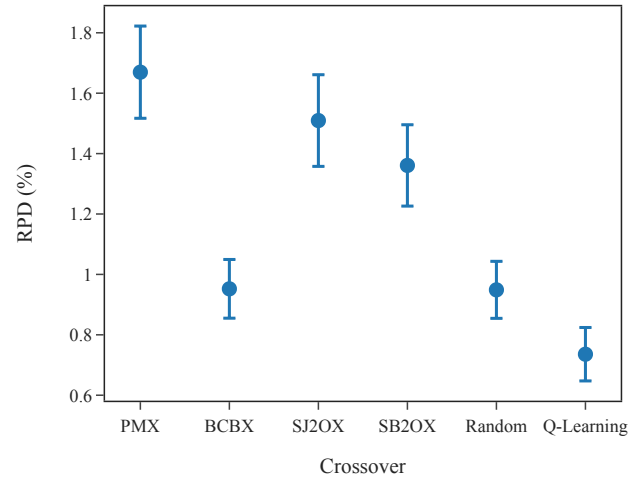
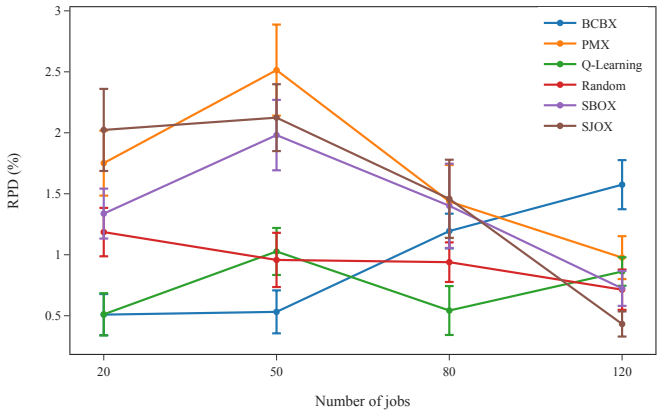


Figure 5: Tournament size.

Crossover





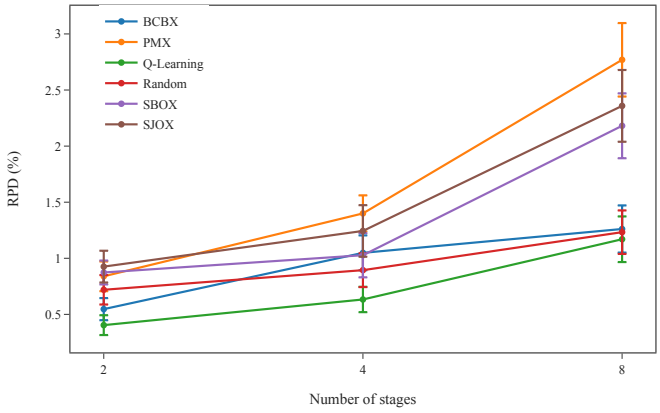


Figure 6: Crossover operators.

Q-learning Crossover Parameters

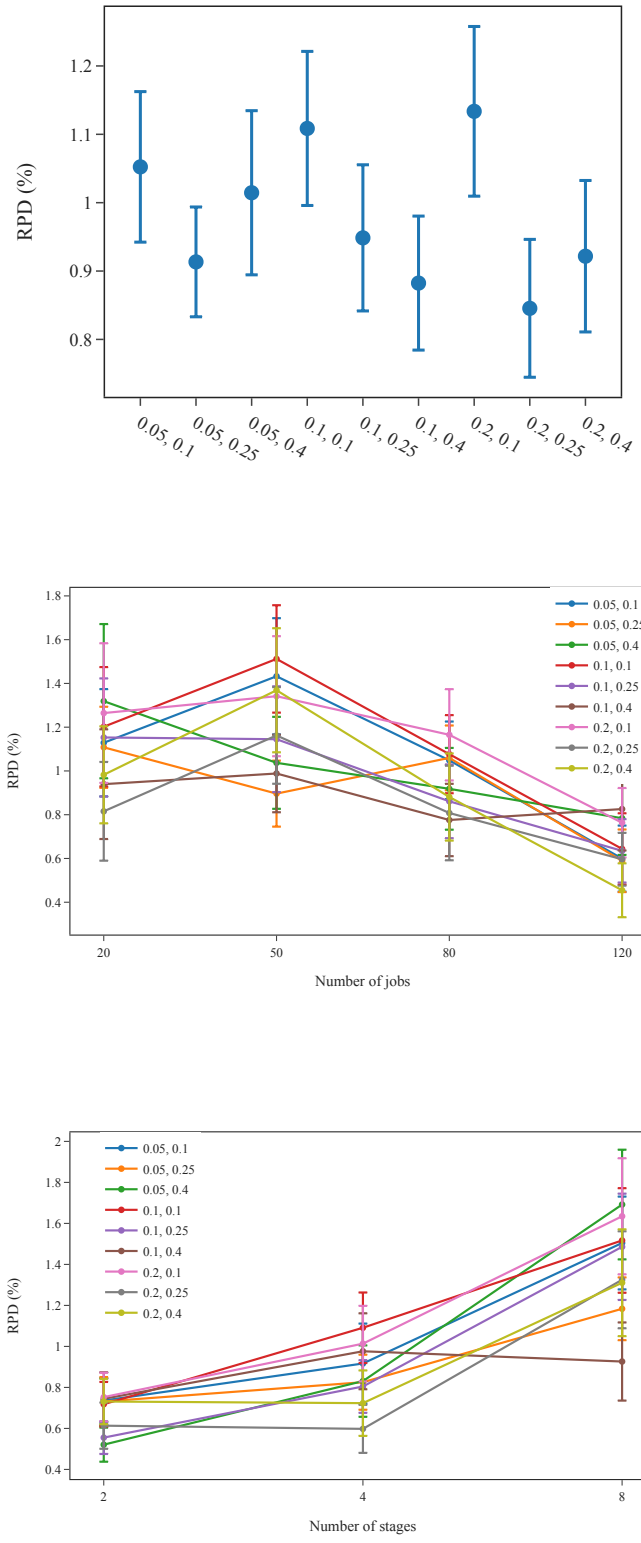
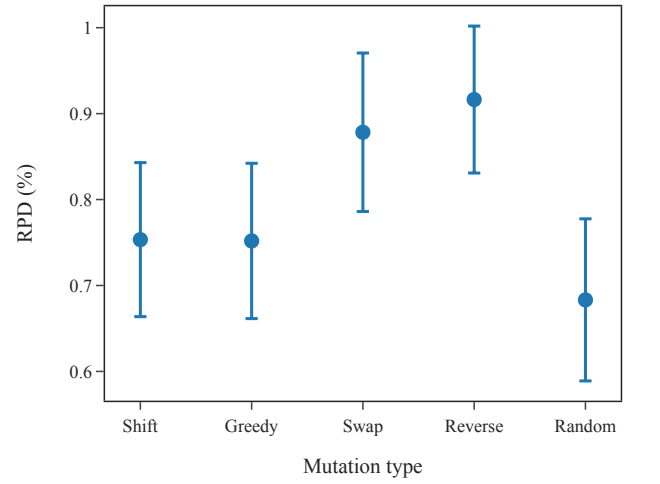
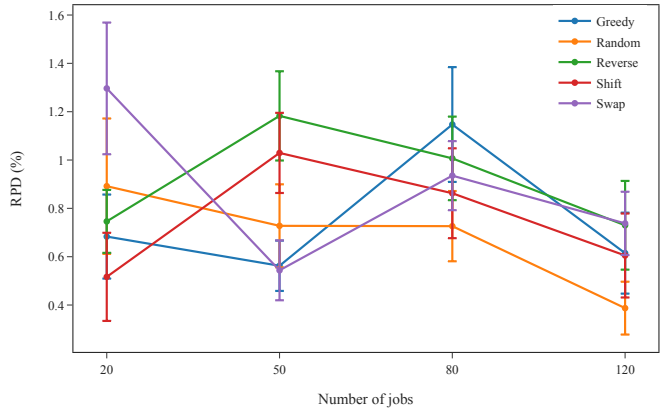


Figure 7: Q-learning crossover parameters (learning rate, epsilon).

Mutation





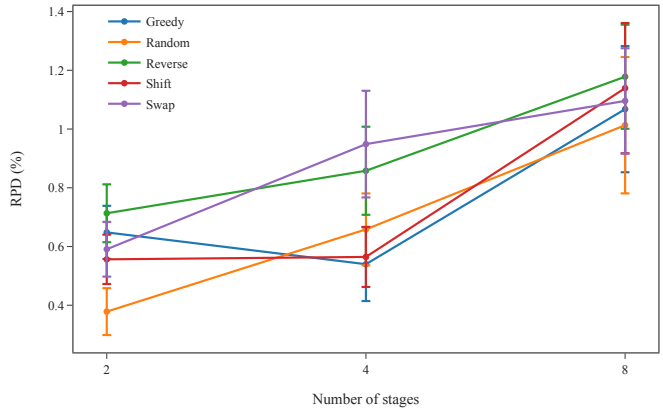


Figure 8: Mutation types.

Mutation Probability

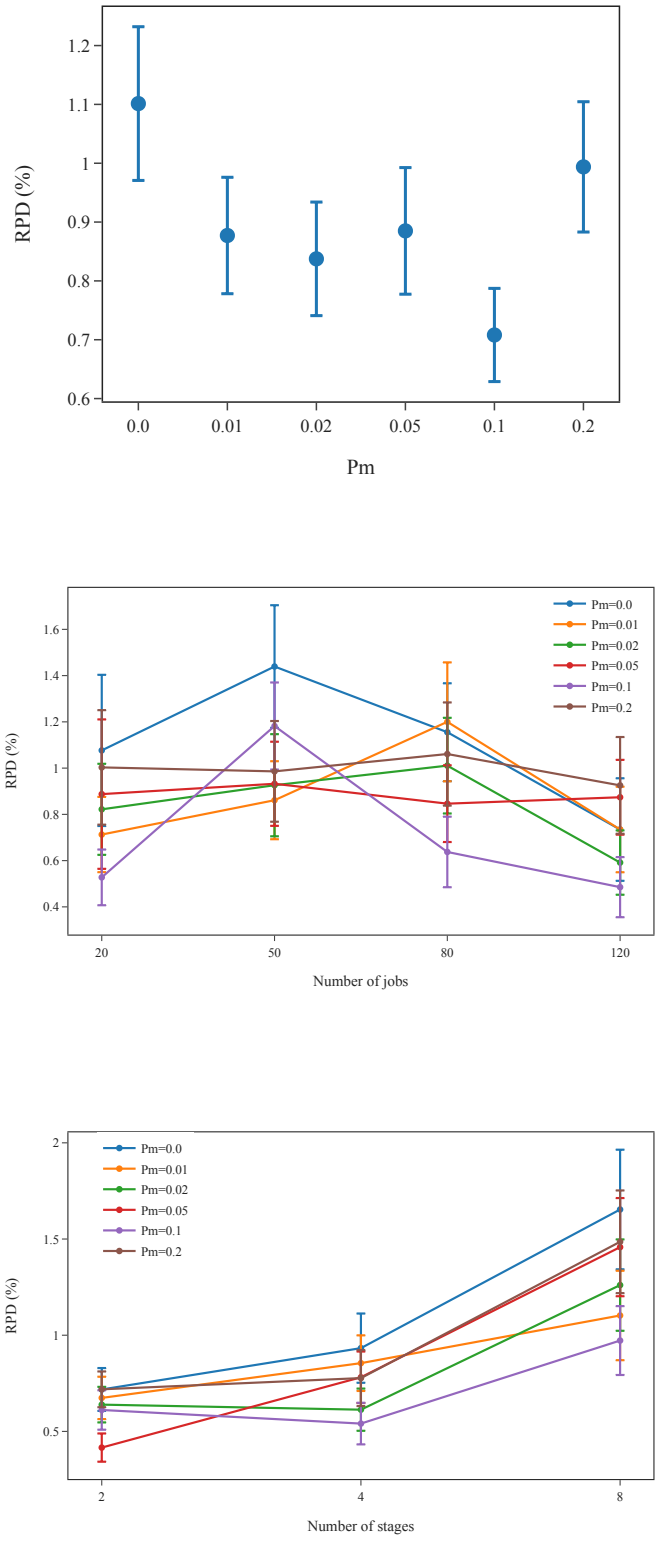
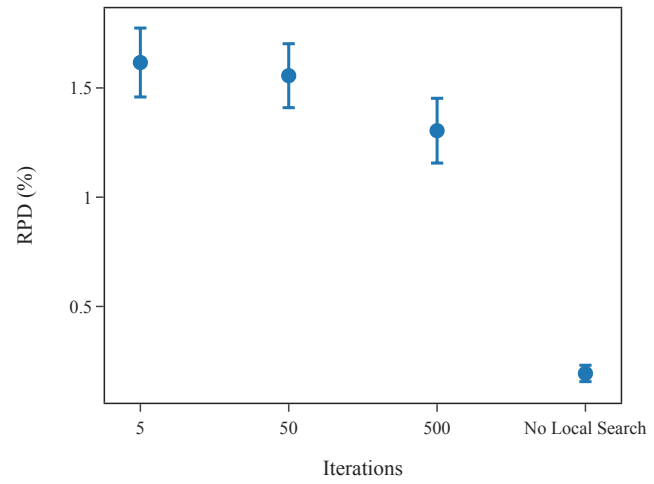
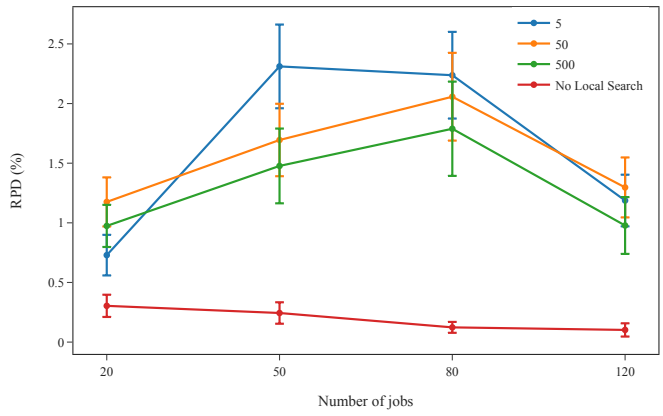


Figure 9: Mutation probability.

Local Search





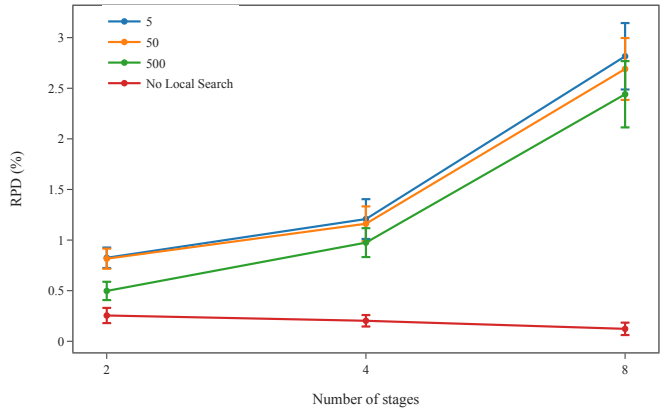


Figure 10: Local search.

Deviation Distance Crowding

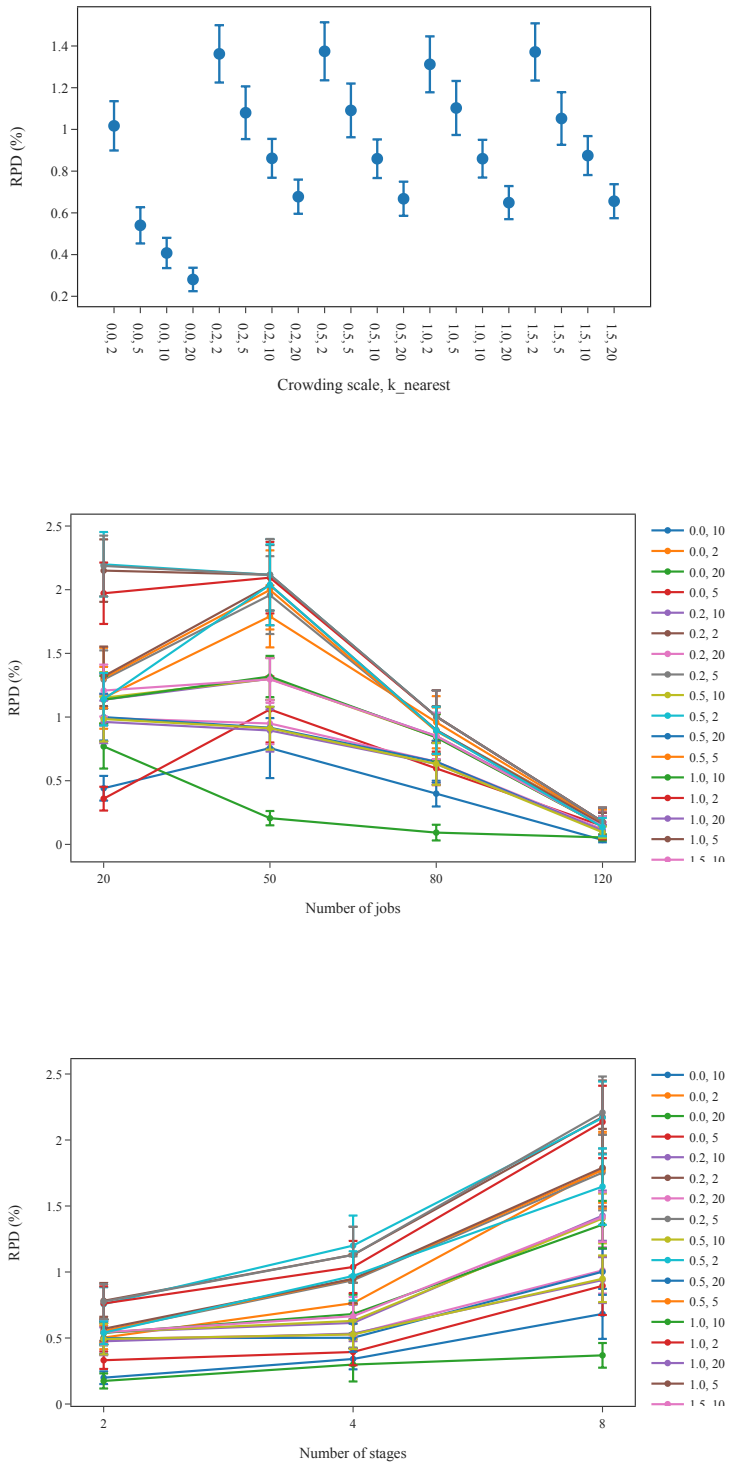


Figure 11: Deviation distance crowding.

Exact Match Crowding

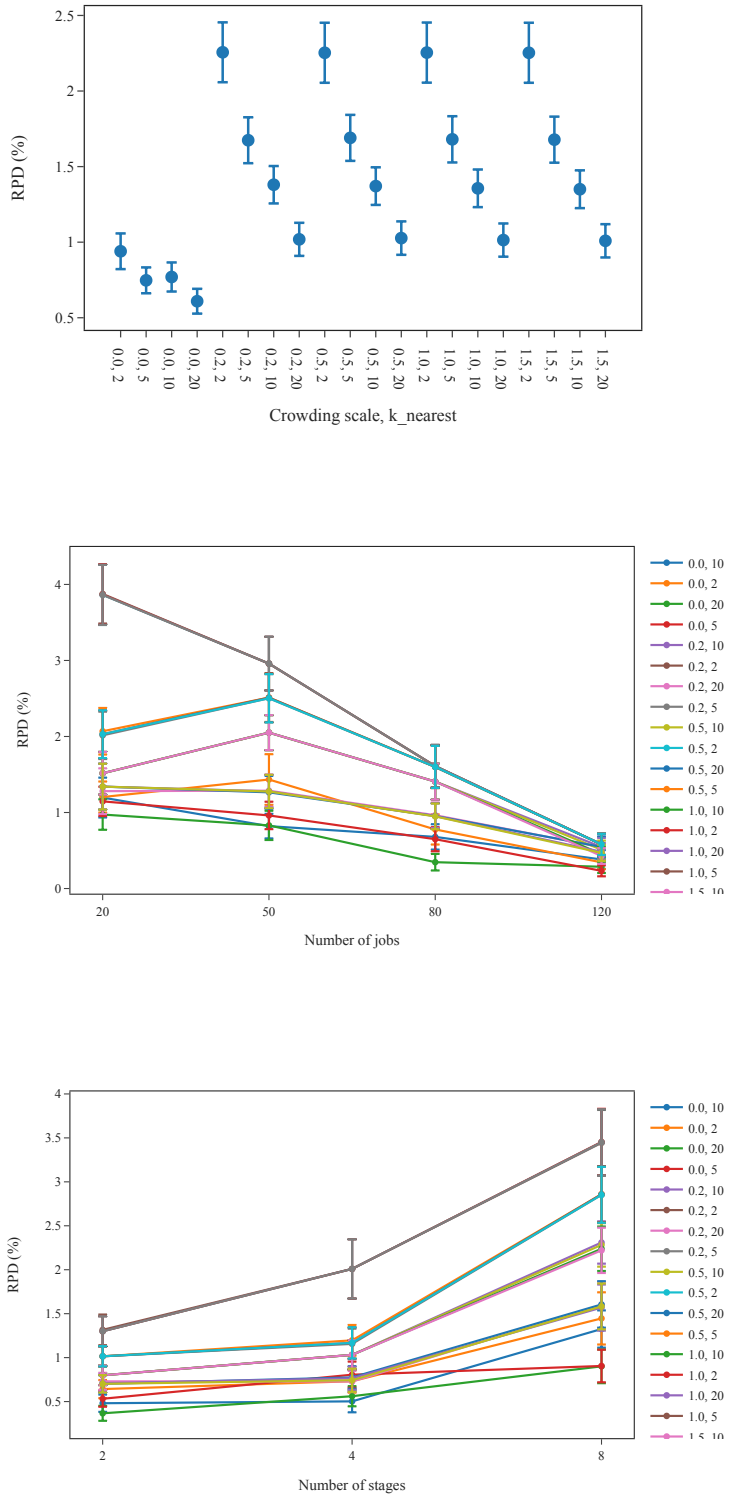
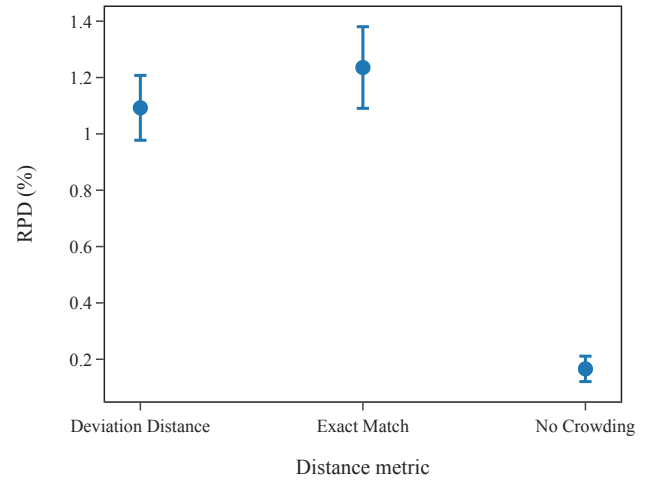
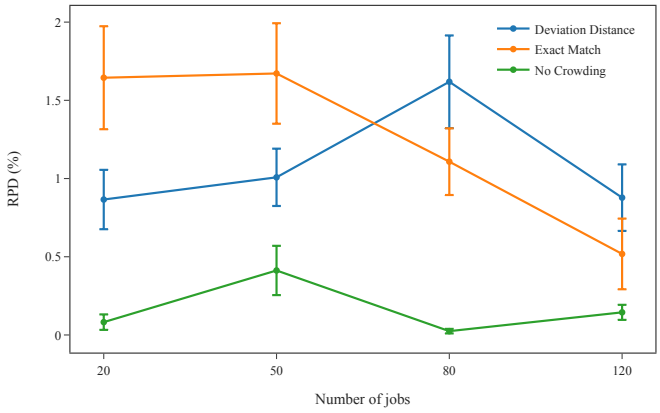


Figure 12: Exact match crowding.

Crowding Versus no Crowding





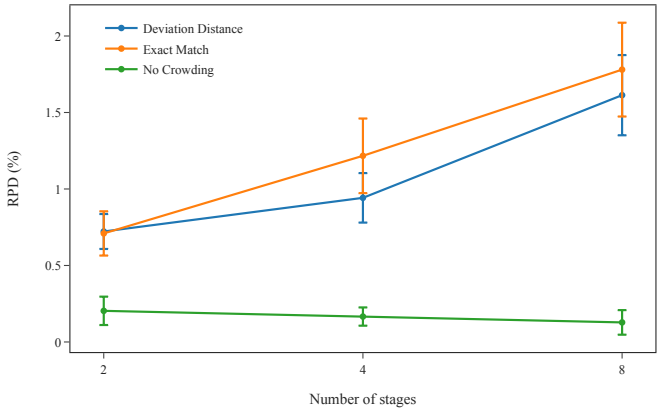


Figure 13: Crowding.

Replacement

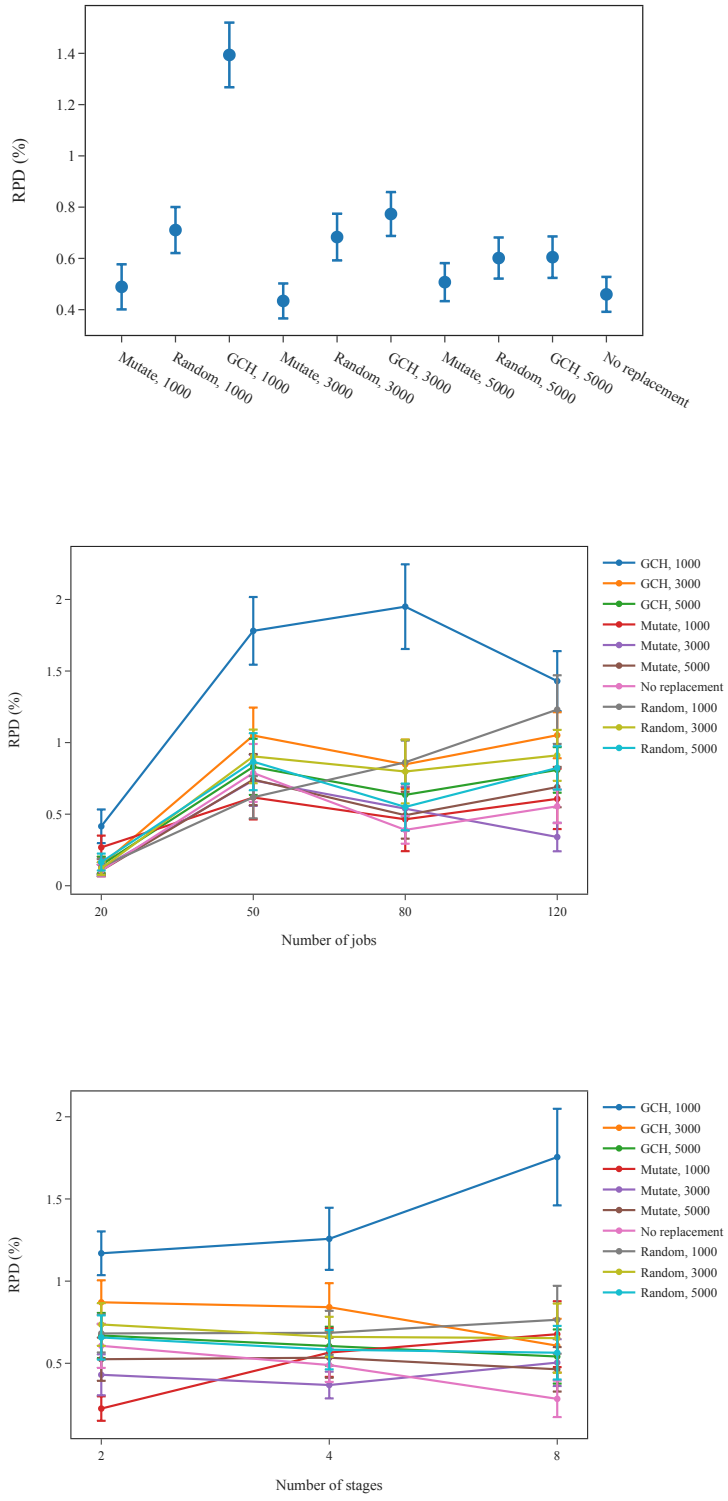
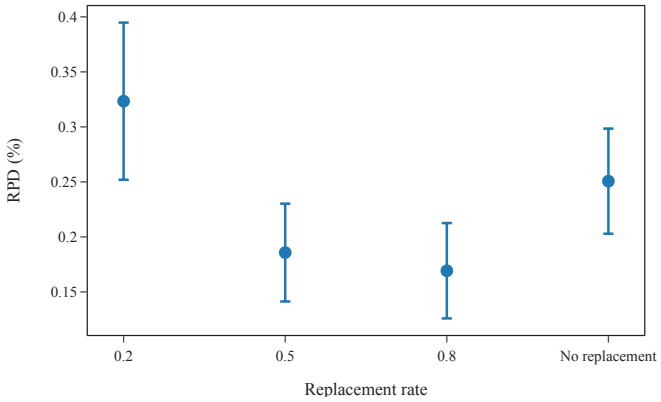
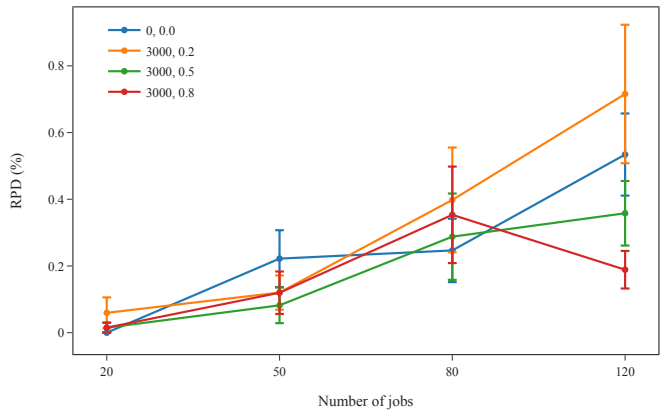


Figure 14: Replacement types.

Replacement Rate





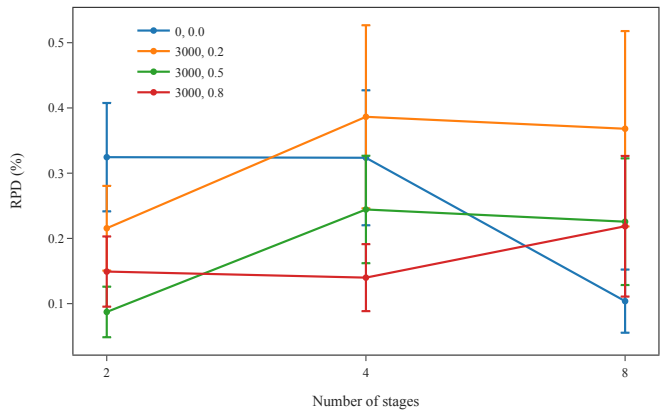


Figure 15: Replacement rates.

B Tuning of Iterated Greedy

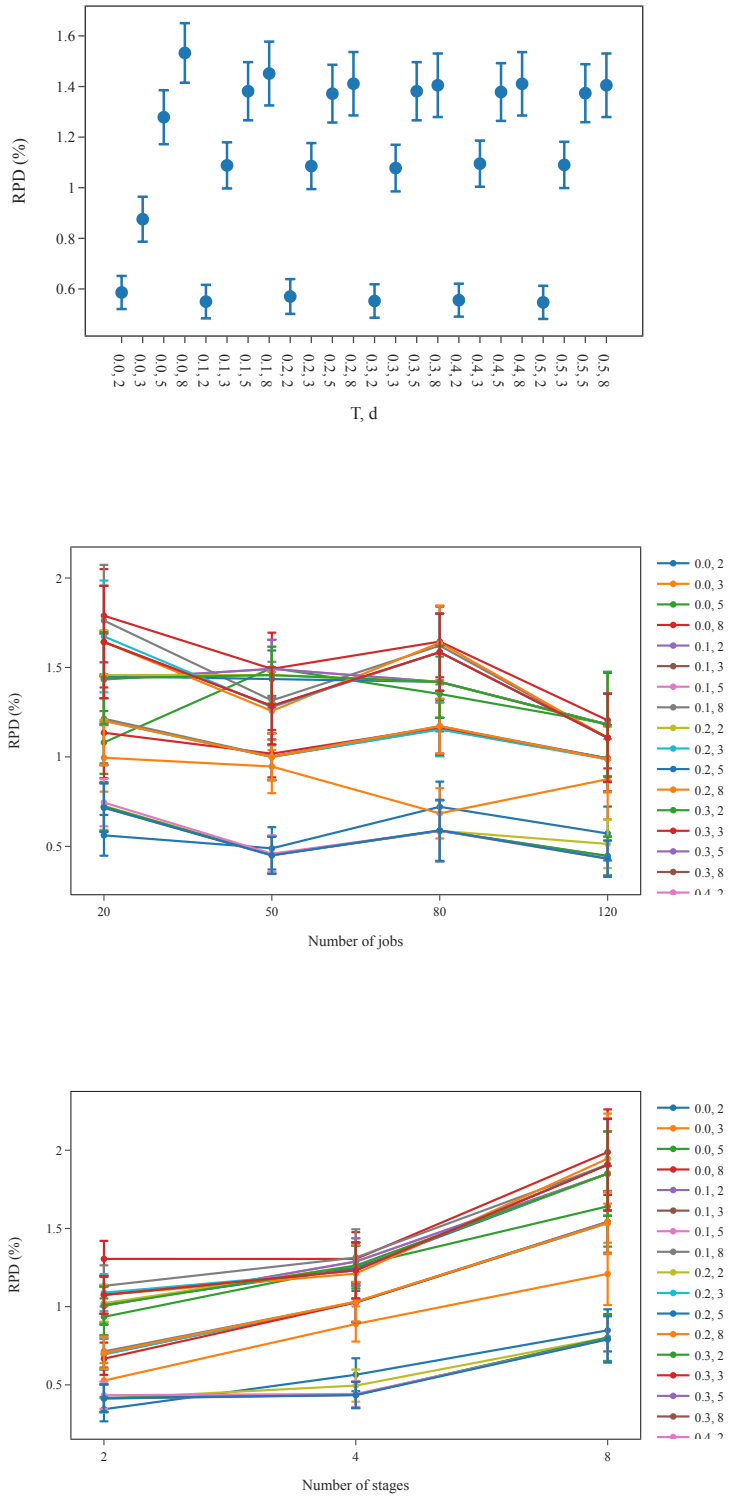
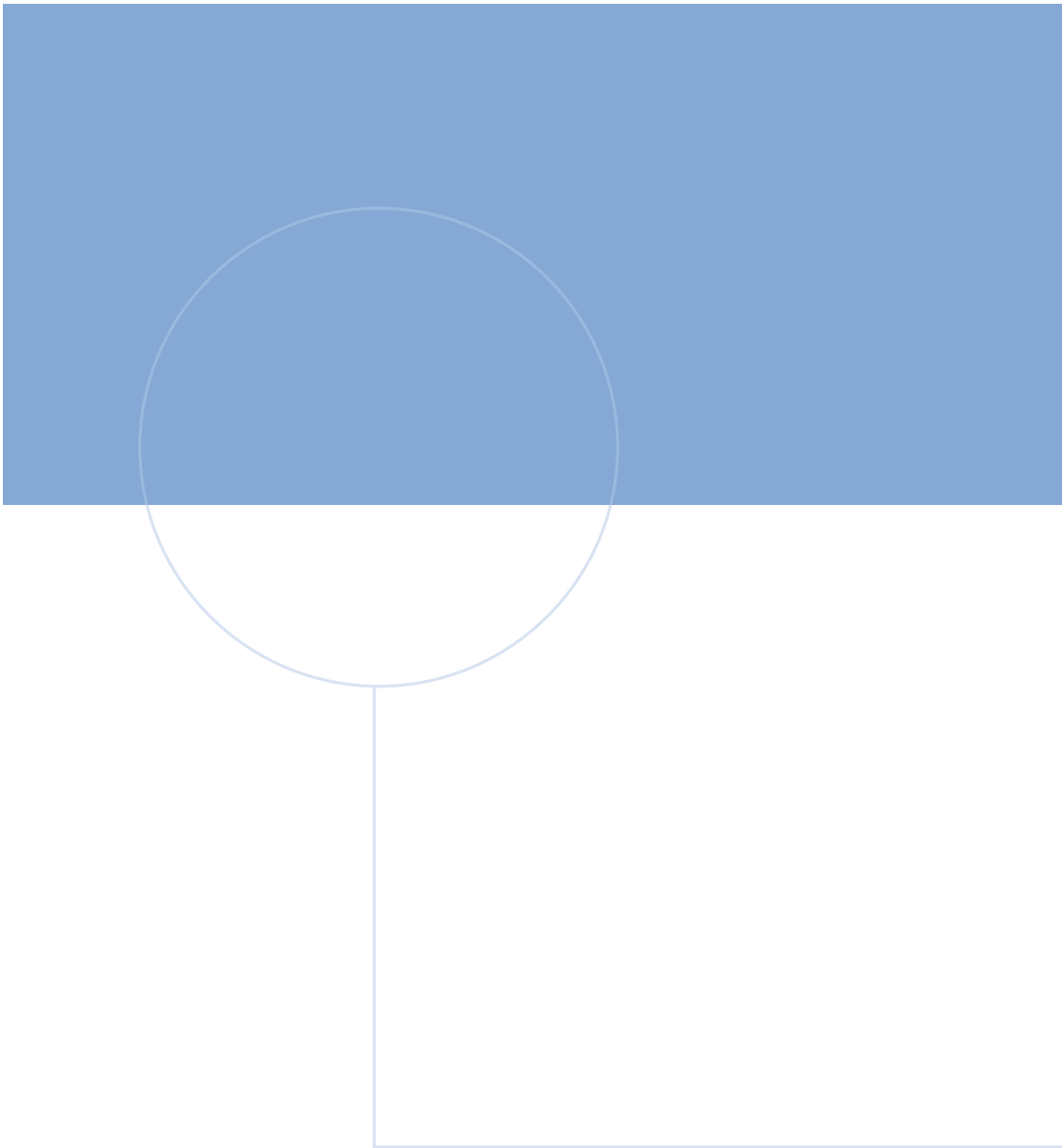


Figure 16: Iterated greedy tuning.



NTNU

Kunnskap for en bedre verden