universität wien

MASTERARBEIT | MASTER'S THESIS

Titel | Title

Impact of different levels of solution-representations on the performance of a hybrid genetic algorithm for a dual-resource constrained re-entrant flow shop problem

verfasst von | submitted by

Philipp Kerl

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien | Vienna, 2025

Studienkennzahl lt. Studienblatt | Degree programme code as it appears on the student record sheet:	UA 066 915
Studienrichtung lt. Studienblatt | Degree programme as it appears on the student record sheet:	Masterstudium Betriebswirtschaft
Betreut von | Supervisor:	emer. o. Univ.-Prof. Dipl.-Ing. Dr. Richard Hartl
Mitbetreut von | Co-Supervisor:	Dr. Yannick Scherr B.Sc. M.Sc.

Abstract- German

In dieser Masterarbeit wird untersucht, welchen Einfluss die Wahl der Lösungsrepräsentation auf die Leistungsfähigkeit hybrider genetischer Algorithmen (HGAs) am Beispiel des Dual Resource Constrained Re-entrant Flexible Flow Shop Problems (DRCRFFSP) hat. Das zugrunde liegende Problem ist durch das reale Siebdruck-Produktionssystem motiviert, mit Re-Entrant-Routing, parallelen Maschinen und teilweise flexiblen, qualifizierten Mitarbeitern, welches auf die Minimierung des Makespans abzielt. Ausgehend von einem veröffentlichten Single-Level-HGA mit indirekter Kodierung wird eine neue Multi-Level-Repräsentation entwickelt, die Operationsreihenfolge, Maschinenzuordnung und Mitarbeiterzuordnung in drei aufeinander abgestimmten Chromosomen abbildet. Im Rahmen eines gemeinsamen C++-Frameworks werden vier Varianten implementiert und verglichen: SL-GA, SL-HGA, ML-GA und ML-HGA. Zusätzlich wird eine Random-Crossover-Baseline für die Multi-Level-Variante berücksichtigt. Alle Varianten werden auf einem standardisierten DRCRFFSP-Benchmark getestet, wobei für kleine Instanzen Optimierungsergebnisse aus Constraint Programming (CP) herangezogen werden und für große Instanzen identische Zeitlimits gelten. Auf kleinen Instanzen erzeugt die Single-Level-HGA bessere Lösungen als ihre Multi-Level-Gegenstücke, wobei SL-HGA im Durchschnitt weniger als ein Prozent vom Optimum abweicht. Bei größeren Problemen sind die Multi-Level-Varianten deutlich schneller in der Lösungsfindung, weisen jedoch weiterhin eine spürbare Qualitätslücke gegenüber den Single-Level-Lösungen auf. Für große Instanzen liegen die Multi-Level-Kodierungen typischerweise um etwa 30 % über den Single-Level-Lösungen in Bezug auf den Makespan, sowohl für den Standard-GA als auch für den Hybrid-GA. Insgesamt zeigen die Ergebnisse, dass die Lösungsrepräsentation den stärksten Einfluss auf die HGA-Performance hat, gefolgt von Local Search und der Gestaltung der Operatoren. Bei der Suche nach der besten Qualität der Pläne sind Single-Level HGAs zu bevorzugen. Wo es darauf ankommt, schnell zu einer Lösung zu kommen, sind Multi-Level HGAs eine gute Wahl, auch wenn dadurch die Lösungsqualität leiden muss.

Abstract- English

This thesis investigates how solution representation affects the performance of hybrid genetic algorithms (HGAs) for the Dual Resource Constrained Re-entrant Flexible Flow Shop Problem (DRCRFFSP). The problem is motivated by a real screen-printing production system with re-entrant routing, parallel machines and partially flexible, skilled workers, where the objective is to minimize makespan. Building on a published single-level HGA with an indirect encoding, a new multi-level representation is designed that stores operation sequence, machine assignment and worker assignment in three aligned chromosomes. Within a common C++ framework, four variants are implemented and compared: SL-GA, SL-HGA, ML-GA and ML-HGA, Additionally, a random-crossover ML baseline was included. All variants were evaluated on a standard DRCRFFSP benchmark, utilizing optimization results from constraint programming for smaller instances and equal time limits for larger ones. On small instances, the single level HGA produced better solutions than their multi-level counterpart, with SL-HGA being on average less than one percent from the optimal solution. On larger problems, the multi-level versions were much quicker at producing solutions. However, they still had a noticeable quality gap compared to the single level solutions. On large problems, it was common for the multi-level encodings to be approximately 30 percent worse than single-level in terms of makespan, for both standard and hybrid GA. Overall, these studies indicated that how a problem is defined as a representation is by far the most significant influence on HGA performance, and that local search and the choice of operators can provide additional performance advantages. Thus, single-level HGAs are the preferred option if the primary concern is the quality of the schedules generated, while multi-level HGAs will be a better option if there is a need to generate solutions quickly and compromises on solution quality are acceptable.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

CP	Constraint Programming
DRC	Dual Resource Constraints
DRCRFFSP	Dual Resource-Constrained Re-entrant Flexible Flow Shop Problem
FSP	Flow Shop Problem
GA	Genetic Algorithm
HGA	Hybrid Genetic Algorithm
JS	Job Shop
ML	Multi-Level (representation)
ML-GA	Multi-Level Genetic Algorithm
ML-HGA	Multi-Level Hybrid Genetic Algorithm
MIP	Mixed-Integer Programming
OS	Open Shop
RFSP	Re-entrant Flow Shop Problem
RFFSP	Re-entrant Flexible Flow Shop Problem
SL	Single-Level (representation)
SL-GA	Single-Level Genetic Algorithm
SL-HGA	Single-Level Hybrid Genetic Algorithm
Cmax	Makespan

1. Introduction

The complexity of scheduling today's production systems is increasing because of several factors: The shortening of the cycle length of products, an increase in customization, and an ever-increasing demand to use available resources as efficiently as possible. Many modern manufacturing systems have products visiting each processing station in a production system more than once, competing for the same limited amount of machine time, and requiring skilled laborers who are unavailable when needed. As a result, these characteristics create constrained planning problems, which are too complex to solve optimally using optimization techniques or even simple heuristics for realistically sized problem instances (Garey et al., 1976). This has led to an increased interest in advanced heuristics and metaheuristics including hybrid genetic algorithms (HGA), for both research and application purposes.

1.1 Problem Context and Motivation

The problem considered in this thesis is rooted in a real industrial production system for multi-layer products with several printing and processing steps, as described by Mlekusch & Hartl (2025). In this industrial process, jobs go through a sequence of stages but have to travel back to previous stages for the application of additional layer or treatment. This type of routing behavior can be modeled using a re-entrant flow shop, where the jobs will re-enter a stage previously visited, before being completed. These types of re-entrant structures are also typical of other manufacturing processes such as semiconductor manufacturing (Vargas-Villamil et al., 2003), where wafers may be returned to photolithography or etching tools to add multiple layers. As a result, the re-entrant flow shop scheduling problem (RFSP) is a particular version of shop scheduling which extends the classic flow shop problem by allowing the jobs to re-enter the same stage more than once. Due to the complexity of the interactions between the jobs, machines and re-entries, the RFSP is classified as difficult to solve optimally and exact solutions are generally limited to small size problems. Over recent years, a significant amount of research work has been dedicated to investigating the properties of the different versions of the RFSP and similar models, with meta-heuristic techniques (such as genetic algorithms, tabu search and hybrid approaches) becoming increasingly popular to solve these problems (Danping & Lee, 2011).

However, machine availability is not the only crucial limitation in many actual production systems. Additional resources, most notably human labor with specialized skills, are often necessary for production. Workers and machines must be scheduled together in these dual-resource-constrained (DRC) environments, and each operation needs an appropriate mix of workers and machines for the duration of its processing time. The importance and complexity of such DRC systems have already been noted by Treleven (1989). According to more recent surveys, research on DRC issues and their adaptable variations is still ongoing and difficult (Dhiflaoui et al., 2018).

A variety of factors are present in the production environment analyzed by Mlekusch & Hartl (2025): re-entering routing, machine flexibility (specifically parallel machines at some process steps), labor resource limitations and worker skill. The DRCRFFSP (dual-resource-constrained re-entrant flexible flow-shop-problem) was introduced as a methodological way to model these constraints to minimize the total processing time (makespan). In order to obtain better solutions than those obtained with the pure genetic algorithm, Mlekusch & Hartl (2025) suggested a hybrid genetic algorithm with a one level encoding for the solution and a constraint programming model. While their study has shown that a hybrid approach may be a good alternative when exact methods fail to provide high-quality solutions for realistic size problems, it also highlights that there is still much to do to improve both the algorithmic performance and the scalability of the hybrid approach to large and complex problems. In the same direction, Lin et al. (2013) proposed a multi-level genetic algorithm that separates the decision concerning the sequencing of jobs from other resource allocation decisions in different chromosome levels. The multi-level encodings were compared to each other with respect to solution quality, and the results showed that the multi-level encodings are competitive regarding the solution quality and allow for an explicit modeling of the interplay between the routing and the resource constraints.

1.2 Research Gap

Existing research on re-entrant flow shops, dual-resource-constrained scheduling and their combination in the DRCRFFSP indicates that metaheuristic approaches, and in particular genetic algorithms and hybrid GAs, are a promising option for realistically sized instances (Chamnanlor et al., 2014; J.-S. Chen et al., 2008; Han et al., 2021; Mlekusch & Hartl, 2025).

However, within this stream of work, comparatively little attention has been paid to the role of solution representation as a design choice in hybrid GAs for the DRCRFFSP.

In the broader RFSP and DRC literature, different genetic algorithm designs have been proposed that encode scheduling decisions either in a single chromosome (e.g., a sequence of operations) or in a multi-level structure with separate chromosomes for sequencing and resource allocation decisions (Gong et al., 2018; Lin et al., 2013; Mlekusch & Hartl, 2025). Each type has demonstrated its capabilities within its own problem domain and thus, for each type, there exists empirical data showing the potential to produce high-quality solutions when paired with effective local search and appropriate decoding techniques. These results, however, have generally been presented in isolation. As a result, they were generated using different problem formulations, instance sets, and evaluation metrics.

The lack at present is an empirical study comparing a single level and a multi-level representation for Genetic Algorithms (GAs) in a common hybrid GA architecture using a common DRCRFFSP scenario. More specifically, it is unknown how different single-level and multi-level representations may compare with respect to solution quality, computation required, and robustness across problem instance size as well as the degree of sensitivity to key genetic search parameter settings when used to solve the same instances of the DRCRFFSP and when optimized for the same objective function. This means that there are very few resources available to guide practitioners and researchers about whether a single or a multi-level representation will be better suited for solving problems associated with the DRCRFFSP.

This thesis aims to address this void by developing and empirically analyzing both a single-level and a multi-level (hybrid) Genetic Algorithm for the DRCRFFSP on a set of identical benchmark instances. Therefore, the main goal is to develop a comprehensive comparative analysis of how the above-mentioned levels of solution representation affect the efficiency of hybrid GAs in a difficult scheduling environment (i.e., the DRCRFFSP).

1.3 Research Question and Hypotheses

Building on the described literature and industrial motivation, this thesis analyzes and compares alternative solution representations within a hybrid genetic algorithm for the DRCRFFSP. The thesis specifically tackles the following research question:

How does a multi-level solution representation in hybrid genetic algorithms improve the algorithmic performance for a dual resource-constrained re-entrant flow shop scheduling problem compared to single-level representations?

To answer this research question, the thesis pursues two main objectives:

1. To design and implement a multi-level solution representation for a hybrid genetic algorithm tailored to the DRCRFFSP, based on and extending the existing single-level HGA of Mlekusch & Hartl (2025).
2. To conduct a systematic computational study comparing the performance of single-level and multi-level representations in terms of solution quality and runtime on a set of benchmark instances.

The investigation is directed by the following hypotheses, which are empirically examined through computational experiments:

1. Multi-level representations yield better optimization results than single-level representations by an improved fitness evaluation, thereby improving the solution quality of the model.
2. Multi-level representations yield more efficient results than single-level representations by reducing the runtime of the model.

The primary objective of the thesis is to elucidate the function of solution representation design in hybrid genetic algorithms for intricate production scheduling challenges and to formulate pragmatic recommendations for researchers and practitioners engaged in DRCRFFSP-type environments.

1.4 Methodological Approach

To address the research question and test the hypotheses, this thesis follows an experimental, model-based research design. As a starting point, the dual-resource-constrained re-entrant flexible flow shop problem (DRCRFFSP) and the single-level Hybrid Genetic Algorithm (HGA) implementation in C++ by Mlekusch & Hartl (2025) are adopted as the underlying problem setting, including their assumptions, constraints and makespan minimization objective. This ensures that all comparisons are carried out within a consistent framework.

On this basis, the original single-level HGA is analyzed and modularized with respect to its main components (chromosome structure, decoding, selection, crossover, mutation and local search). A multi-level solution representation is then designed that explicitly encodes operation sequence, machine assignment and worker assignment in separate chromosomes. Within a shared C++ framework, a family of algorithm variants is implemented, including single-level GA/HGA (SL-GA, SL-HGA) and multi-level GA/HGA (ML-GA, ML-HGA), as well as an alternative crossover operator as a control variant to isolate the effects of operator design. The variants share as many components as possible so that observed differences can be attributed mainly to representation and hybridization choices.

Benchmark instances for empirical research were obtained from Mlekusch (2024) for the DRCRFFSP. For small instances, the GA/HGA variants are compared against a constraint programming (CP) model to compare against optimal solutions. This is only available for the small instance set, as the large set is too complex to find optimal solutions. For larger instances, therefore, the focus lies on relative performance between single-level and multi-level variants. All instance types, as well as various configurations of the algorithms being studied, had multiple independent executions with identical parameters between the variants tested to allow evaluation of comparative performance through comparison of best (and average) makespan values as well as execution time. This allows me to evaluate the trade-offs associated with increased representation complexity vs. algorithmic efficiency and test the previously-stated hypotheses.

1.5 Structure of the Thesis

The remainder of this thesis will be organized in the following chapters. Chapter 2 provides an overview of the theoretical basis for re-entrant flow shop production systems, dual resource constrained problems, and genetic algorithms and references existing studies. Chapter 3 gives the formal definition of the DRCRFFSP, along with a mixed-integer programming formulation of it. Chapter 4 describes the methodological approach to solving the DRCRFFSP using both single-level and multi-level Genetic Algorithms (GAs) and Hybrid Genetic Algorithms (HGA), which include descriptions of how each solution representation and component of each algorithm was implemented. Chapter 5 describes the experimental design that was used for testing, such as the specifics of the implementation of the GAs/HGAs, specific characteristics

of the benchmark data sets that were created and how each instance of the data sets were evaluated. Chapter 6 presents the computational results from running the algorithms on both small and large instance sets. The final chapter, Chapter 7, summarizes the major contributions of the study, provides information about the potential real-world applications of the model and solution approaches described, and identifies areas for further research.

2. Theoretical Background

This section gives an overview of relevant related work and addresses the theoretical background of this thesis. It begins by discussing the core principles of flow shop and re-entrant flow shop scheduling and the specific challenge introduced by dual resource constraints. Next, solution methods frequently used in the literature for such scheduling problems, as well as the proposed Genetic Algorithm and its variants, will be introduced. By covering the key ideas and methods that will be developed further in the upcoming sections, this chapter provides valuable insight into the state of the current research on such scheduling problems and their solution methods.

2.1 Review of Flow Shop and Re-entrant Flow Shop Scheduling

Scheduling in the context of manufacturing and production systems is a decision-making process routinely used to allocate resources to operations in order to optimize said schedules. It typically involves deciding on the timing and sequence of tasks across available machines, with the aim of optimizing a given performance criterion, such as makespan, cost or tardiness. Depending on how the tasks of each job are routed through the manufacturing system, different types of scheduling systems can occur (Pinedo, 2016):

- **Flow Shops (FS):** All jobs go through the same sequence of machines.
- **Job Shops (JS):** Each job has a different processing sequence.
- **Open Shops (OS):** No fixed order for job processing.

The scheduling system this thesis focuses on is flow shop scheduling. Pinedo (2016) further describes flow shops as m machines in series, where each job n has to be processed on each of the m machines. Furthermore, all jobs need to follow the same order of machines, typically denoted M_1 -> M_2 -> … -> M_m. The main objective according to the author is to find the sequence in which the n jobs are completed in a way that the overall makespan C_{max} is minimized. Similar to this definition, Emmons & Vairaktarakis (2013) describe flow shops as a system that processes jobs with the same number and sequence of operations, called a chain precedence. These are performed on the same machines with the same processing time for each operation. An example of such systems is that of assembly lines, in which products have

to go through the same steps every time along the assembly line, as seen in automobile manufacturing. A review of flow shop scheduling research by Cheng et al. (2000) aligns with these definitions and characteristics, demonstrating a common understanding in academic literature and indicating a clear consensus on what a flow shop is. On top of varying scheduling types, there are also variations of flow shop scheduling (Emmons & Vairaktarakis, 2013):

Flow Shop Type	Job Routing	Common Examples
Simple Flow Shop	Same linear sequence	Auto assembly lines
Hybrid Flow Shop	Same sequence, but parallel stages possible	Electronics, component assembly
No-Wait Flow Shop	Immediately on next machine without waiting time	Heat-sensitive or time-critical processes
Re-Entrant Flow Shop	Same sequence, but with re-entrant loops	Semiconductor, Integrated Circuit manufacturing

Table 1: Types of Flow Shops

This thesis will focus on the Re-Entrant Flow Shop Scheduling Problem (RFSP). Graves et al. (1983) were among the first authors to introduce the concept of the RFSP. They developed a model to describe production systems where there are jobs that revisit one or more machines before they complete their production cycle. Integrated Circuit (IC) Manufacturing is a typical example of such a system. Silicon Wafers go through many Photolithography Processes that need to be repeated for every layer of the IC. Due to the lack of re-entering in basic flow shop models, these systems cannot be modeled with flow shop models. Therefore, the RFSP is viewed in a different way from the flow shop models. In addition to the complexity introduced by the non-linearity of the sequence of the jobs, the RFSP also includes additional complexities related to scheduling. Graves et al. pointed out that due to the non-linear nature of the sequence of the jobs, the RFSP looks similar to a Job Shop at first glance. However, when you take a closer look at the RFSP, it is clear that the RFSP has much more structure than a classic Job Shop. That's why the Re-entrant Flow Shop can be seen as a combination of both the structure of a flow shop and the flexibility of a Job Shop.

In subsequent years other researchers such as Kumar (1993) and Danping & Lee (2011) established the Re-entrant Flow Shop Scheduling Problem as a distinct area of study within the

field of scheduling. Some areas that exhibit the same characteristics as those of the RFSP include semiconductor manufacturing, textile dyeing, chemical processing and screen printing. All of these industries have products made using a sequence of operations that are followed in a predetermined order. Therefore, the RFSP is well suited to use in these industries. For instance, the following could be considered as an example of an RFSP:

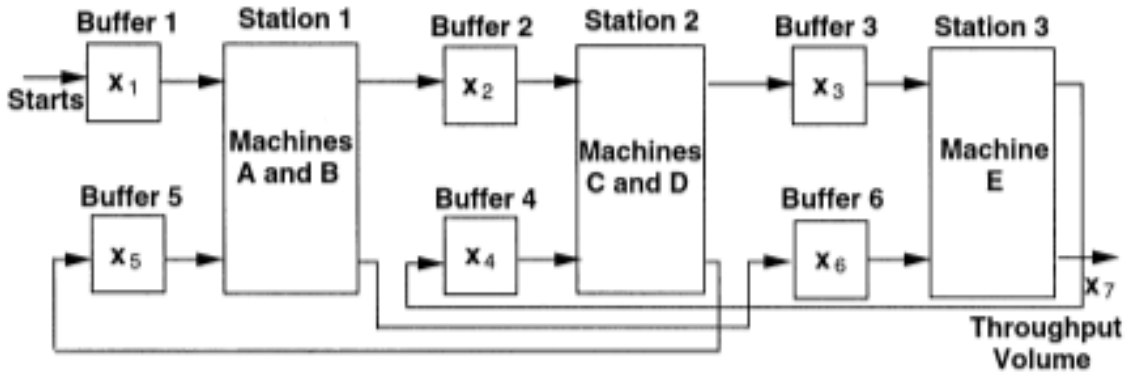

Figure 1: RFSP based on an Intel case study (Vargas-Villamil et al., 2003)

Figure 1 illustrates a production system with three stations (S_t) as part of a production workflow that consists of six individual steps. For this production system, the production job passes through the stations sequentially in the order of: S_1 -> S_2 -> S_3 -> S_2 -> S_1 -> S_3, thereby revisiting stations that have already processed the job previously rather than continuously processing the job from start to finish. The RFFSP model provides an illustration of how most of these production workflows are re-entrant, i.e., they pass by previous stages more than one time. If production systems produce multiple products and those products revisit a particular station more than one time, then the complexity of the manufacturing system increases significantly. As shown in Figure 1, if a station such as Station 2 processes the same production item at two or more different stages in the production workflow, the station will be subjected to increased levels of resource utilization as well as overlap.

Building on this concept, the Re-entrant Flexible Flow Shop Scheduling Problem (RFFSP) extends the RFSP by incorporating machine flexibility, meaning certain stages may have multiple parallel machines available to process operations. This means that at some stages, there may be more than one machine available to process operations at the same time. This version is more like what happens in real-life factories, where some steps in the process need more capacity than others. This scheduling problem is harder because it needs to be scheduled

not only in the order of each job, but also on which machine each stage of the job needs to be done. Wang et al. (2025) and Mlekusch & Hartl (2025), are two recent studies that address this increasing complexity in their works.

The classic Flow Shop Scheduling Problem (FSSP) itself is a difficult problem to solve, and it becomes NP-hard with more than two machines (Garey et al., 1976). Once you add re-entrant loops and flexibility (RFSP & RFFSP) the difficulty of solving this type of scheduling problem increases significantly. The problem can also be classified as strongly NP-hard for many real-world scenarios (Danping & Lee, 2011; Pan & Chen, 2003). Therefore, the use of the standard sequencing rules that have been developed for FSSPs does not always lead to the desired results and additional strategies need to be developed to meet goals such as reducing the makespan or maintaining machine utilization over time. The complexity of these problems is further exacerbated by the presence of dual resource constraints (DRCs) - i.e. both equipment and personnel are constrained. DRCs are a significant addition to the classic flow shop environment and will be the focus of the subsequent section.

2.2 Review of Dual Resource Constraints in Scheduling

The re-entrant aspect of the FSP makes it more complicated, but many manufacturing systems also have to deal with another important constraint: they have to not only assign operations to machines, but also workers to each operation. The traditional FSP and its extension, the RFFSP, only look at the machine as a resource constraint. The Dual Resource Constraints (DRC) add to this by also looking at the people who are needed in these manufacturing systems.

Nelson (1967) was among the first researchers to explore both machine and worker limitations as constraints in scheduling. He was also one of the first to conduct comparative studies of these two classes of resource constraint. Treleven (1989) extended Nelson's earlier research on the coordination of machine and worker resources within the context of manufacturing systems through an empirical analysis of how these resources interact in actual production situations. In addition, Treleven has categorized manufacturing systems based upon the degree of labor flexibility in task assignments, ranging from fixed assignments of tasks to workers to completely flexible task-worker assignments. Treleven also emphasizes the interdependence of machine and worker resources stating that operational activities cannot

occur until both of these resources are simultaneously available. Finally, Treleven discusses the need for training employees in multiple jobs as a means of improving their ability to be moved between tasks and locations as needed by the organization (e.g., when product mix changes or there is insufficient material). The comprehensive literature review conducted by Hottenstein & Bowman (1998), which documented and synthesized results from over a dozen discrete-event simulation-based DRC studies, provided a foundation for this research effort and reinforced the importance of organizing the workforce and providing cross-training for employees.

Subsequent simulation studies explored how workforce flexibility and staffing levels affect performance. For example, Frye (1974) and Park (1991) showed that even limited cross-training can improve overall throughput and reduce bottlenecks, particularly in dual-constrained job shops. Kher & Malhotra (1994) found that flexible labor can partially compensate for less efficient layout configurations. A central takeaway from these studies is the trade-off between the number of available workers and their level of flexibility. Felan, Fry, & Philipoom (1993) demonstrated that a small group of broadly skilled workers can sometimes outperform a larger, specialized team. Kher & Fry (2001) extended this by showing that partial flexibility, where only some workers are cross-trained, can offer most of the performance benefits of full flexibility at much lower cost. On top of that, Felan & Fry (2001) suggest that a homogenous workforce with uniform cross-training, where every worker is trained to do every task, can lead to lower system performance in certain contexts. Instead, based on their simulation results, they found that systems with a mix of highly flexible and specialized workers can perform better in terms of overall utilization and scheduling efficiency. Additionally, Daniels et al. (2004) show that increasing flexibility beyond a certain point can lead to diminishing returns, particularly when the cost of training is factored in or when coordination complexity increases.

Although a completely cross-trained workforce would eliminate many of the scheduling complexities associated with machine and worker constraint assignment restrictions because any worker could potentially run any machine, this degree of flexibility is generally unattainable in most production environments. In most production environments, employees are trained in a particular set of skills that limits their potential utilization on certain machinery or at specific times, with training costs representing an additional limitation to employee

placement. In addition to the aforementioned skill constraints, employee availability during multiple shifts, labor regulations (such as employee work hour restrictions) and the potential for worker fatigue create additional restrictions to feasible assignments and make the scheduling process even more complex as the scheduler will have to take into consideration both machine and employee availability.

Dhiflaoui et al. (2018) present a more detailed examination of dual resource constraints when reviewing scheduling problems within these environments. The authors categorize DRC scheduling problems into distinct categories as follows: shop types (flow shop, job shop, flexible job shop), flexibility levels of workers, and problem objectives with a focus on static, deterministic models. The authors' work contrasts prior DRC research by focusing on static, deterministic models while the majority of previous DRC research examined dynamic shop-floor systems through the use of simulations. They highlight the increased computational complexity introduced into DRC environments and indicate that even simple scheduling problems can become strongly NP-hard due to the need to assign both the required machinery and the qualified workers.

The authors further categorize the DRC into three distinct problems:

- **Machine Assignment**: Determining which machine each operation should be processed on.
- **Worker Assignment**: Allocating workers to operations based on their qualification levels and time constraints.
- **Operation Sequencing**: Creating the order of operations with regard to the availability of machines and qualified workers in order to optimize the performance objective

The proposed formal problem formulation is defined over a set of independent jobs $J = \{J_1, J_2,..., J_n\}$, machines $M = \{M_1, M_2, ..., M_m\}$, and workers $W = \{W_1, W_2,..., W_l\}$. Each job J_i comprises a sequence of operations $\{O_{i1}, O_{i2},..., O_{ir}\}$ that must adhere to precedence constraints and be executed on a compatible machine operated by a suitably skilled worker. The processing time of each operation is a function of both the assigned machine and worker. To organize and evaluate the growing body of research in this area, Dhiflaoui et al. introduce a six-dimensional classification schema. This schema classifies existing DRC studies based on: (1) the type of solution method employed (e.g., exact, heuristic, or metaheuristic), (2) the extent of machine

flexibility, (3) the degree of worker flexibility, (4) the optimization objectives (e.g., makespan, cost, tardiness), (5) the implemented algorithmic approaches (e.g., genetic algorithms, mixed-integer programming, tabu search), and (6) the structure of the approach, distinguishing between hybrid and non-hybrid models. This classification helps when comparing different DRC scheduling approaches and points to gaps, particularly in more complex scenarios like re-entrant systems.

This thesis will build on this foundation by looking at the DRC environments that were shown. In these environments, workers can use more than one machine, but they are only experts at a few. This limited flexibility creates realistic limits that are often seen in industrial systems and makes the RFFSP even more complicated. The Dual Resource Constrained Re-Entrant Flexible Flow Shop Problem (DRCRFFSP) is the name of this combined problem. The proposed method for solving it will be used. Finding the best solutions for the real-world DRCRFFSP is impossible using exact methods once the problem gets too big. This is because the problem is too hard to solve. This has led to the use of heuristics and metaheuristics, like genetic algorithms and tabu search, for problems that are this hard. While the DRCRFFSP literature remains sparse, the same methodological justification is applicable.

2.3 Review of Solution Methods for the DRCRFFSP

This chapter has given an organized overview of all the solution strategies offered in literature for the Re-entrant Flexible Flow Shop Problem (RFFSP) and the Dual Resource Constraints (DRC) which complicate the RFFSP. The following section is organized from less to more difficult problems based on the problems complexity. First, there are traditional solutions approaches developed for flow shop and re-entrant flow shop scheduling. Following these approaches, I have identified the solution methodologies which solve DRC's or other related constraints which lead to the Dual Resource Constrained Re-entrant Flexible Flow Shop Problem (DRCRFFSP). In addition to evaluating all of the solution approaches discussed above using the classification framework provided by Dhiflaoui et al. (2018), as well as described in Chapter 2.2, the solution approaches were also classified according to solution approach (i.e., exact, heuristic, metaheuristics), degrees of machine and worker flexibility, objective function, algorithmic structures, and hybridization. Using this process will allow to identify how

advances in scheduling theory occur in a specific area and what areas of complex, real-world manufacturing systems need additional research.

2.3.1 Solution Methods for Re-entrant Flow Shop Problems (RFSP)

The authors first discussing the term "RFSPs" were Graves et al. (1983). They stated the problems inherent with scheduling jobs which have to go back to equipment in order to be completed. Later research on the RFSP further confirmed that even a simple version of it is an NP-hard problem.

The early studies using exact models to solve the RFSP used small problem instances to solve them. An example of this is Pan & Chen (2003) that had created 3 different extended mixed integer programming (MIP) versions of the Re-entrant Permutation Flow Shop (RPFSP) to create the shortest makespan possible. They found that although their method could optimize job sequences for very small job quantities and equipment quantity, as the quantity of jobs and equipment increased, so did the amount of time required to process the jobs, and thus the efficiency of their model decreased dramatically. Similarly, Yang et al. (2008) and Choi & Kim (2009) both developed Branch-and-Bound algorithms to optimize for two machine RFSP problems for Makespan and Total Tardiness, respectively. Although both of the exact algorithms mentioned above will produce optimal job sequences in smaller examples, they too have shown significant limitations in being scalable enough to handle much larger job and equipment examples. To summarize, the use of exact optimization techniques such as MIP and Branch-and-Bound models has been very successful at finding optimal job sequences in relatively small-scale examples of the RFSP. However, their use cases are significantly limited by the computational complexities involved and therefore become less efficient to use on medium to large scale examples of the RFSP. This serves to motivate the development of additional solution techniques that can accommodate larger scale examples.

Researchers employed heuristic methods to address larger RFSPs, prioritizing practicality over guaranteed optimality in their solutions. Bertel & Billaut (2004) made early attempts to come up with constructive heuristics and simple genetic algorithms for a re-entrant hybrid flow shop (RHFSP) with jobs that go around and around. They tried to cut down on lateness by using rules that were specific to the problem to guide the search. Choi & Kim (2008) also came up with a number of heuristic methods just for the RFSP. Their work presents two heuristics: a

lower-bound-based algorithm (LBB) and an idle-time-based algorithm (ITB). These put sub-jobs in order based on either their estimated makespan bounds or their machine idle time. They built on these and suggested two hybrid versions (HLI1 and HLI2) that use both strategies. The authors also changed the well-known NEH constructive heuristic by Nawaz et al. (1983) to fit the re-entrant setting. This resulted in three modified variants (MN1 - MN3) that consider re-entrant precedence constraints during insertion. These heuristic methods were useful for finding good solutions to problems that exact methods could not solve.

As re-entrant flow shop problems got bigger, metaheuristic algorithms became the best way to solve them. Genetic algorithms (GA), tabu search, and other algorithms strike a balance between exploration and exploitation, allowing them to solve complex problems in ways that simple heuristics cannot. Chen et al. (2008) were among the first to use a GA on the RFSP, which was a change from exact methods to heuristic and metaheuristic methods. As time went on, more advanced and hybrid metaheuristics were developed. These included new heuristics that made the solutions even better. Chamnanlor et al. (2014) were the first to suggest an adaptive hybrid GA (HGA) for an RFSP with time-window constraints. They did this by using fuzzy logic control and a left-shift local search to improve the quality of the solutions. Their HGA was able to solve bigger problems that were too hard for exact methods to do perfectly. Subsequent metaheuristics include an Adaptive Large Neighborhood Search (ALNS) for distributed RFSP (Rifai et al., 2016), a hybrid particle swarm optimization algorithm for RFFSP (Sangsawang et al., 2015), and a hybrid tabu search for RPFSP (J.-S. Chen et al., 2007), each demonstrating substantial enhancements over fundamental heuristics. As these solution methods matured and demonstrated efficacy for progressively intricate scenarios, researchers commenced investigations not only into problem-solving techniques but also into optimization criteria. This resulted in an expanded emphasis on objectives beyond makespan, aligning with the requirements of contemporary production settings. Ying et al. (2014) tackled a bi-objective RHFSP by presenting an Iterated Pareto Greedy (IPG) algorithm, concentrating on the minimization of both makespan and total tardiness. Their method found a middle ground between conflicting goals and high scores on certain benchmark tests.

The inclusion of additional objectives reflects a research trend toward more realistic evaluations of schedule quality. Notably, metaheuristics have proven capable of accommodating these complex objectives. Studies report that GAs and related methods

perform well on large re-entrant flow shop instances (Chamnanlor et al., 2014; J.-S. Chen et al., 2008), where exact methods usually fail to find any feasible schedule. In summary, the literature on RFSP and related versions shows a clear evolution: Exact methods established foundational results on small cases, but heuristic and metaheuristic approaches now dominate practical solution techniques, enabling near-optimal scheduling for large-scale re-entrant flow shops.

2.3.2 Solution Methods for Dual Resource Constraints (DRC) in Scheduling

The previous section discussed solution methods for flow shops with re-entrant structures. However, many actual scheduling situations are made even more difficult by the presence of dual-resource constraints (DRCs). Consequently, a substantial body of literature has developed concerning the resolution of DRC issues, incorporating both analytical and simulation-based approaches. The next part gives an overview of DRC scheduling solution methods, from exact methods and constraint programming models to metaheuristics.

Because even simple DRC problems are NP-hard, exact methods for DRC scheduling have mostly only been used for small cases or for benchmarking. Several researchers have suggested MIP formulations to encapsulate these dual resource constraints regarding machine and worker assignments. For instance, Chu et al. (2018) developed a chain RFSP incorporating multi-resource qualification matching via MIP, stipulating that specific jobs commence and conclude on the same machine with compatible workers. Andrade-Pineda et al. (2020) also created a dual-resource flexible job shop with processing times and due-date goals that depended on the workers, but they found that pure MIPs did not work well for anything bigger than small cases. In general, these kinds of studies show that MIP models have a hard time even finding solutions that work for medium-sized DRC cases. Constraint Programming (CP) is another exact method that is better at dealing with resource limits that come with workers and machines. Kress & Müller (2019) created a CP model for a flexible job shop where the length of each operation depends on the specific machine-worker pairing. Their CP did better than a similar MIP, showing that it works better for scheduling problems with limited workers. Müller & Kress (2022) showed again that an optimized CP model can be a tough test for metaheuristics on small to medium-sized instances, in addition to their earlier work. These precise and constraint-based techniques yield significant optimal solutions for minor instances

and assist in the validation of heuristics. However, they are usually not very useful for real-world situations because they are getting more complicated.

Researchers used simulation and heuristic methods to find more useful solutions to realistic DRC systems. In the early days of DRC scheduling, simulation was used a lot to test basic dispatch rules and workforce policies. Treleven (1989) and Hottenstein & Bowman (1998) looked at basic simulation studies and talked about how to decide when and where to send workers and how to dispatch jobs. Early DRC problems were often solved using heuristics like First-Come-First-Served (FCFS), Shortest Processing Time (SPT), and Modified Due Date (MOD). Park (1991) and Frye (1974) found that cross-training workers made them work much better, even when they were using simple dispatching rules. Over time, more specific rules came out, like centralized vs. decentralized worker transfer. With centralized rules, idle workers are moved to other tasks right away, but with decentralized rules, they have to wait until local queues are clear (H. Kher & Malhotra, 1994). Another heuristic strategy was to change the way workers were set up. Kher & Fry (2001) and Felan et al. (1993) showed that teams with a mix of workers who are good at a lot of things and workers who are experts in one thing work better than teams that are either fully flexible or fully specialized. Felan & Fry (2001) suggested using a few very flexible "utility" workers to fix problems while other workers stay focused on their main tasks. Bokhorst et al. (2004) backed this up by showing that different worker skills make dispatching rules like the "assign-best-worker" policy work better. Based on this, simulation studies showed that system performance gets better when dispatch rules are used with skill-based worker allocation and transfer policies. Most DRC methods used in real life are based on these heuristic combinations.

As these simulation-based heuristics got better, people started to pay more attention to metaheuristics because they work well with large instance sizes. GAs were one of the first ways to schedule DRC (ElMaraghy et al., 2000). They did this by encoding job-machine-worker sequences and changing them to improve goals like makespan or tardiness. Over time, a lot of different metaheuristic methods have been used, such as Tabu Search, Iterated Local Search, and Scatter Search. These methods have been shown to be able to deal with the combinatorial complexity of DRC problems, which is something that exact methods like Mixed Integer Programming (MIP) often cannot do on bigger instances (Andrade-Pineda et al., 2020; Chu et al., 2018).

The majority of research studies regarding DRC has been on job-shop problems, however, researchers have also applied meta-heuristics to optimize Flow Shop Problems. Han et al. (2021) combined the concept of rigidly assigned workers in an HFSP model with MOEAH utilizing many heuristic decoding methods, while producing better results than MILP on the larger models. Gong et al. (2018) developed a Memetic Algorithm to address worker flexibility by implementing genetic operators along with local search, which demonstrated strong performance on extensive models. In previous years, Mehravaran & Logendran (2013) researched this area of study in a DRCPFSP and produced three Tabu Search algorithms to address the extremely difficult NP-Hard problem. Similarly, Ruiz-Torres et al. (2022) modeled a totally flexible workforce environment and indicated that through the application of flexibility along with powerful meta-heuristics, it is possible to achieve significant improvements in the quality of solutions. The most commonly used scheduling method for DRC has been evolutionary algorithms, including genetic algorithms (GAs) and memetic algorithms (MAs). These types of algorithms have proven useful for finding near-optimal schedules in large, NP-hard problems where classical optimization does not function well.

2.3.3 Solution Methods for DRCRFFSP

While there is a large body of research addressing either the Re-entrant Flow Shop Problem (RFSP) or Dual Resource Constraints (DRC) individually, only a limited number of studies have looked at their combination. The Dual Resource Constrained Re-entrant Flow Shop Problem (DRCRFSP) incorporates the looped routing feature of re-entrant systems along with the increased difficulty of coordinating schedules for both machines and qualified workers. Because of this combination, the problem is hard to solve with computers and has only recently gotten more attention in the scheduling literature.

Lin et al. (2013) was among the first researchers to consider the above combined system. Lin et al. proposed a Multi-Level Genetic Algorithm (MLGA), that can address both re-entrant job flow and worker-machine dependency. Worker flexibility is a part of their model as well. Flexibility allows certain workers to perform tasks on multiple machines. They link this flexibility to the re-entrant processing route of the jobs. The MLGA has the potential to provide improved models of realistic factory systems, particularly the types found in the semiconductor or electronic industry, where both re-entrancy and worker limitations

commonly exist. Another study that addresses both resource complexity and routing complexity is the one by Chu et al. (2018). This study presents an approach to solve the multi-resource qualification matching problem in a chain-type RFSP system. Specifically, their model requires jobs to begin and finish at the same machine. As such, they reinforce the re-entrant characteristic of the system. Their algorithm seeks to match worker qualifications with job requirements for operations as those require looped paths, demonstrating the practical relevance of the integration of re-entrance with DRC in both modeling and optimization. One of the most recent and extensive studies on the DRCRFSP was conducted by Mlekusch & Hartl (2025). Their research provides the most detailed and expansive approach to date to the DRCRFSP. Their study utilizes a single-level HGA to address the DRCRFSP, including benchmark cases and the application of exact solutions using CP, along with information regarding scalability of various solution strategies. Additionally, their research examines partial worker flexibility which is akin to the majority of manufacturing systems where workers are trained for fewer than all of the stages of the process. A study by Dong & Ye (2022) does not specifically relate to human resources. However, their study is related to the topic. They studied energy-aware constraints in an RHFSP. The authors show that similar to the inclusion of multiple resource levels to flow-shop models, other layers of resources can be added to the model in similar ways, although worker layers were not the subject of the study. Their study demonstrates how to model hybrid constraints within production systems that have the ability to be re-entered.

There is a clear trend in the literature: DRCRFSP is rarely used with exact methods. Chu et al. (2018) and Mlekusch & Hartl (2025) suggest exact methods like MIP or CP, but these are mostly used for small problems or to set standards. Exact methods quickly stop working for bigger, industrial-scale cases. As a result, many studies use metaheuristics like GAs or hybrid strategies because the problem is NP-hard. Another limitation is that there are not any public benchmark datasets for DRCRFSP. There are datasets for standard RFSP, like the ones mentioned by Chamnanlor et al. (2014), but they are not specifically designed for dual resource constraints. Mlekusch & Hartl (2022) were the first to publish clear DRCRFSP benchmarks. This makes their work very useful for future research. These differences make it hard to compare results from different papers and show that we need to use more standardized problem definitions in this field.

In the literature, it appears that DRCRFSP is almost never solved using an exact solution method. Chu et al. (2018) and Mlekusch & Hartl (2025) recommend exact solution methods such as MIP or CP. However, exact solution methods are generally limited to smaller problems and/or providing performance bounds for larger problems. For larger, industrial-sized problems, exact solution methods typically fail. Therefore, most of the literature uses heuristics such as Genetic Algorithms or Hybrid Strategies since the problem is NP-hard. A second limitation of the literature is that no public benchmark datasets exist for the DRCRFSP. While some benchmark datasets exist for the Standard RFSP, e.g., those listed by Chamnanlor et al. (2014), none of them are designed for the specific dual-resource constraint of the DRCRFSP. Mlekusch & Hartl (2022) were the first to develop and publish benchmark data sets for the DRCRFSP and therefore provide valuable contributions to the literature for future research. The lack of common problem definitions between publications creates difficulties when comparing results across the various publications in the field and demonstrates the necessity to adopt more standard problem definitions in this area of study.

2.4 Review of Genetic Algorithms in DRC and RFSP

Within metaheuristics, evolutionary algorithms, particularly those with GAs, are the most widely used methods for scheduling problems with dual resource constraints. This is due to their ability to efficiently encode machine and worker assignments and balance multiple objectives such as makespan and tardiness simultaneously. Their specific structure also makes it easy to hybridize with other heuristics, which is why GAs have become a staple in the evolution of scheduling algorithms from early adoptions to recent advancements (Dhiflaoui et al., 2018).

GAs have been used in the late 1990s as solutions to scheduling problems that could be proven NP-hard. There were many researchers including Chen et al. (1995) and Reeves (1995), who developed genetic algorithms for the PFSP. Following these researchers' work, Murata et al. (1996), developed a GA for the FSP which they tested with a variety of crossover and mutation operator types. In comparing this GA, and others, Murata et al. (1996) determined that their hybrid GA, when paired with local search or simulated annealing, was superior to all others. Their results clearly demonstrated that GAs can produce near optimal schedules for difficult

scheduling problems, and that using GAs in combination with heuristics is often an effective method of solving such scheduling problems.

Researchers in the literature were able to experiment with many other problem types. Chen et al. (2008) demonstrated a Hybrid Genetic Algorithm (HGA) with a Local Improvement (LI) component was able to find near-optimal makespans for RFSPs, and was able to beat a Plain Genetic Algorithm (PGA), and classical heuristics in performance. Sun (2009) subsequently investigated a RFSP with sequence dependent setup times, by developing a GA that obtained good makespan results compared to other heuristics. Researchers investigating semiconductor manufacturing contexts may encounter additional constraints to flow shop type problems including time windows, so in order to investigate how to solve such problems, Chamnanlor et al. (2014) studied an RFSP with per-stage time windows, where their proposed HGA used active repair to address any time window constraint violations, and then applied a local search to improve upon the solution. Alongside research into the area of RFSPs, there has been significant advances in GAs solving DRC scheduling problems, which are among the most researched areas within DRC scheduling problems. The primary advantage of GAs is their ability to easily encode both resources and production schedules into a single chromosome, thereby providing the possibility of evolving both feasible schedules and resource allocations. A number of studies have utilized this fact, typically encoding schedules with resource assignment genes, and utilizing specialized crossover and repair operators to maintain schedule feasibility under dual constraints. One example of the above is the study completed by Lin et al. (2013), who investigated a DRCRFSP, and therefore combined both problems. They developed an ML-GA that captured re-entrant production stages, and co-allocated labor resources, utilizing a repair operator to resolve any conflicts. Using minimization of makespan as the objective function, their GA performed better than a comparable Simulated Annealing (SA) algorithm for both small and large instances, thus demonstrating the ability of GAs to perform well on such cases.

Over the last few years there have been advancements in GA literature through hybridizing with specialized methodologies as well as adding additional objectives to be solved within a GA framework. A prime example of this is Mlekusch & Hartl (2025). They developed a Hybrid Genetic Algorithm (HGA), based on a DRCRFFSP related to screen printing production, by combining a Constraint Programming Model with both a single level GA and Intelligent

Decoding Heuristic. The HGA was able to produce high quality schedules for large problem sizes, while also improving upon existing benchmark solutions. This illustrates the current performance of GA's in comparison to other methodologies (especially when used in combination with problem specific heuristics or used with exact methodologies for comparing a GA's solution against the optimal solution for small instances). Modern GAs can do more than just minimize makespan. They can also handle multiple goals. For instance, Chen et al. (2025) came up with a quantum-tunneling mutation for a dual-resource flexible job shop that would help meet three goals at once. Their algorithm concentrates on late jobs, overall tardiness, and makespan, yielding almost optimal solutions within feasible runtime constraints, thereby showcasing GA's flexibility in multi-objective scheduling contexts.

The development of this new generation of evolutionary methods shows how clearly these new advanced hybrid versions of GAs dominate scheduling research today compared to the very early implementations that were first developed in the 1990s. GAs have shown themselves to be able to be applied to complex production environments such as re-entrant manufacturing and dual-resource shops with great success and produce nearly optimal solutions. This demonstrates that GAs remain one of the main tools for scheduling optimization research, since, as long as they are continually adapted, they can handle the growing complexity of real-world production scheduling problems.

The progression of GA-based models demonstrates an increase in complexity from simpler implementations developed in the 1990s to today’s more advanced hybrid GAs. Successful use of GAs has been shown in difficult environments such as re-entrant manufacturing and dual resource shops and have achieved solutions that closely approximate optimal solutions. These studies indicate that GAs remain one of the key technologies for improving the science of scheduling as well as for handling the increasing complexities of real-world production scheduling problems.

2.5 Summary

In the literature on re-entrant flow shop scheduling (RFSP), dual resource-constrained scheduling (DRC), and their combination (DRCRFFSP), there is a clear consensus on effective solution strategies. Exact methods such as MIP or CP can guarantee optimal solutions, but only

for very small problem instances, since their runtime grows exponentially with problem size. Heuristic dispatching rules, such as FCFS or SPT, are fast and can generate decent solutions on medium-sized problems, but their performance decreases as problem size and complexity increase. In contrast, metaheuristic algorithms have emerged as the dominant approach for large and complex scheduling scenarios, especially for problems with re-entrant flows and dual resource constraints. This is especially due to their ability to balance thorough exploration of the search space with exploitation of high-quality solutions, as well as to be tailored to multiple objectives and constraints.

Within metaheuristics, GAs are most common in scheduling settings due to flexible encoding and easy hybridization with local search. Recent studies reported strong results from advanced GA variants for DRC and RFSP settings, including both multi and single-level GAs and HGAs. However, there is also a clear gap in the literature: There have been very few systematic comparisons between different GAs for DRC and RFSP scheduling environments. Particularly, a comparison between single-level GAs (SL-GA) and multi-level GAs (ML-GA) and their hybrid versions has not yet been made. Such a comparison can provide valuable insights into how different GA structures behave under the same problem setting. This lack of direct comparative analysis under standardized conditions motivates the methodology of this thesis.

This thesis therefore addresses this identified gap by:

- Implementing an ML-GA for the DRCRFFSP to test it against an SL-GA.
- Using a common set of benchmark instances and a consistent evaluation for fair performance comparisons.
- Conducting a comparative analysis of both versions and analyzing their performance, including sensitivity to key GA parameters to identify which version is more suitable for the DRCRFFSP.

3. Problem Formulation

The following section formulates the DRCRFFSP as a mixed-integer programming (MIP) model based on the formulation by Mlekusch & Hartl (2025), which in turn builds on the model of Ku & Beck (2016). The model is tailored to the specific structure of the DRCRFFSP with parallel machines, worker skills and possible re-entrances. In the following, I briefly summarize the main decision variables and explain the purpose of the constraints that define the model.

Decision variables

For each job $j \in J$, let O_j denote the ordered set of its operations, and let M and W be the sets of machines and workers, respectively. Operations are processed on predefined stages s_{ji}. The model uses the following decision variables:

- x_{jimw} binary variable equal to 1 if operation *i* of job *j* is processed on machine *m* with worker *w* on the corresponding stage $s\hat{}_{ij}$.
- $y_{jij'i'm}$ binary variable equal to 1 if operation *i* of job *j* precedes operation *i'* of job *j'* $\forall j \neq j'$ on machine *m* of stage $s\hat{}_{ij}$.
- $z_{jij'i'w}$ binary variable equal to 1 if operation *i* of job *j* precedes operation *i'* of job *j'* $\forall j \neq j'$ and both are operated by worker *w*.
- S_{jimw} continuous variable for the starting time of operation *i* of job *j* on machine *m* with worker *w* on the corresponding stage $s\hat{}_{ij}$.
- C_{jimw} continuous variable for the completion time of operation *i* of job *j* on machine *m* with worker *w* on the corresponding stage $s\hat{}_{ij}$.

MIP Model

$$\min \quad C\max \tag{A1}$$

$$Subject \quad to:$$

$$\sum_{m\in M_s}\sum_{w\in W_s} x_{jimw} = 1 \quad \forall j \in J, i \in O_j, s = \hat{s_{ji}} \tag{A2}$$

$$C_{jimw} \geq S_{jimw} + p_{ji} - H(1 - x_{jimw}) \quad \forall j \in J, i \in O_j, s = \hat{s_{ji}}, m \in M_s, w \in W_s \tag{A3}$$

$$S_{jimw} + C_{jimw} \leq H(x_{jimw}) \quad \forall j \in J, i \in O_j, s = \hat{s_{ji}}, m \in M_s, w \in W \tag{A4}$$

$$\sum_{m\in M_s}\sum_{w\in W_s} S_{jimw} \geq \sum_{m\in M_{s'}}\sum_{w\in W_{s'}} C_{ji-1mw} \quad \forall j \in J, i = 2, \ldots, k_j, s = \hat{s_{ji}}, s' = \hat{s_{ji-1}} \tag{A5}$$

$$\sum_{w\in W_s} S_{jimw} \geq \sum_{w\in W_s} C_{j'i'mw} - H(y_{jij'i'm}) \tag{A6}$$

$$\sum_{w\in W_s} S_{j'i'mw} \geq \sum_{w\in W_s} C_{jimw} - H(1 - y_{jij'i'm}) \quad \forall j, j' \in J, i \in O_j, i' \in O_{j'}, (j,i) \neq (j',i'), s = \hat{s_{ji}} = \hat{s_{j'i'}}, m \in M_s \tag{A7}$$

$$\sum_{m\in M_s} S_{jimw} \geq \sum_{m'\in M_{s'}} C_{j'i'm'w} - H(z_{jij'i'w}) \tag{A8}$$

$$\sum_{m'\in M_{s'}} S_{j'i'm'w} \geq \sum_{m\in M_s} C_{jimw} - H(1 - z_{jij'i'w}) \quad \forall j \in J, i \in O_j, i' \in O_{j'}, (j,i) \neq (j',i'), s = \hat{s_{ji}}, s' = \hat{s_{j'i'}}, w \in W_s \cap W_{s'} \tag{A9}$$

$$\sum_{j\in J}\sum_{i\in O_j}\sum_{m\in M_{\hat{s_{ji}}/\hat{s_{ji}}=s}} x_{jimw} = 0 \quad \forall s \in S \forall w \notin W_s \tag{A10}$$

$$C\max \geq \sum_{m\in M_s}\sum_{w\in W_s} C_{jk_jmw} \quad \forall j \in J, s = \hat{s_{jk_j}} \tag{A11}$$

$$S_{jimw}, C_{jismw} \geq 0 \, \forall j \in J, i \in O_j, s = \hat{s_{ji}}, m \in M_s, w \in W \tag{A12}$$

$$x_{jimw} \in \{0,1\} \, \forall j \in J, i \in O_j, s = \hat{s_{ji}}, m \in M_s, w \in W \tag{A13}$$

$$y_{jij'i'm} \in \{0,1\} \, \forall j, j' \in J, i \in O_j, i' \in O_{j'}, (j,i) \neq (j',i'), s = \hat{s_{ji}} = \hat{s_{j'i'}}, m \in M_s \tag{A14}$$

$$z_{jij'i'w} \in \{0,1\} \, \forall j, j' \in J, i \in O_j, i' \in O_{j'}, (j,i) \neq (j',i'), s = \hat{s_{ji}}, s' = \hat{s_{j'i'}}, w \in W_s \cap W_{s'} \tag{A15}$$

Constraints (A1)-(A15) formulate the DRCRFFSP as follows. The objective in (A1) minimizes C_{max}, as defined in (A11), the maximum completion time over all operations, and therefore the makespan of the schedule. The constraint (A2) guarantees that each operation is assigned to a unique feasible pair of machines and workers at the corresponding production step. The constraints (A3) and (A4) link the decision-making process with respect to the timing. While (A3) assures that the completion time of all operations will be greater than or equal to the sum of their start time and their processing time, constraint (A4) imposes that the completion time

of the jobs assigned to the other machines, which are processed by the same workers, will be zero. Finally, the constraint (A5) defines the earliest possible start time of each operation *i* of job *j*, as the completion time of the immediately preceding operation *i* -1 of job *j*. Constraints (A6) and (A7) prevent machine conflicts by requiring each machine to process only one operation at a time. Similarly, constraints (A8) and (A9) require that workers can work on only one machine at any moment. Constraint (A10) restricts assignments of workers to only machines for which they are skilled and trained for. Constraint (A11) defines the makespan by ensuring that C_{max} is not smaller than the completion time of the final operation of each job. Finally, constraints (A12)-(A15) define the decision variables. Start and completion times are non-negative, and all assignment and sequencing variables are binary.

Altogether, the constraints and variables form a compact reference model for the DRCRFFSP that I use as a basis for the rest of the thesis. In the next chapter, I build on this formulation and describe the genetic algorithm variants that are designed to find good schedules for small and large-sized instances.

4. Methodology

After discussing the relevant literature on GAs and DRCRFFSP and the problem formulation, the next chapter will discuss the approach used to solve this scheduling problem. The main aim is to measure the impact of the solution representation on solving the DRCRFFSP, comparing two solution representations, single-level and multi-level, under a common GA structure. To do so, I developed a new multi-level (ML) solution representation, built on top of an existing single-level (SL) version from Mlekusch & Hartl (2025), which serves as the baseline. This includes two different variants, a standard GA (SL-GA) and an HGA (SL-HGA), including a local search heuristic. My contribution is an equivalent ML version (ML-GA/HGA) that keeps the basic GA loop and evaluation settings the same, while introducing multi-level encoding and adjusted genetic operators needed to handle it. This version is then used to compare the newly generated results to the results obtained from the single-level version. A comparative analysis like this has not been done before for DRCRFFSP specifically. The results, therefore, contribute to the field by providing valuable insight into the choice of solution representation suitable for such practical applications. The chapter is structured as follows: Section 3.1 gives an overview of the GA concept and how it applies here. Section 3.2 details the two solution

representations and their decoding procedures, which form the core of this comparison. Section 3.3 then describes the overall algorithmic framework, including GA, HGA and all variants considered. Section 3.4 will examine the adaptations to the genetic operators for the ML-GA/HGA, and Section 3.5 closes with a brief summary that bridges into the experimental setup in the next chapter.

4.1 Genetic Algorithm Overview

This thesis uses a standard GA structure. As explained previously, the baseline for this is the structure used by Mlekusch & Hartl (2025). This algorithm iteratively improves a population of individual solutions, in this case a feasible schedule of operations with assigned machines and workers. This is done by applying genetic operators to find these improved individuals. Figure 2 demonstrates the basic GA loop:

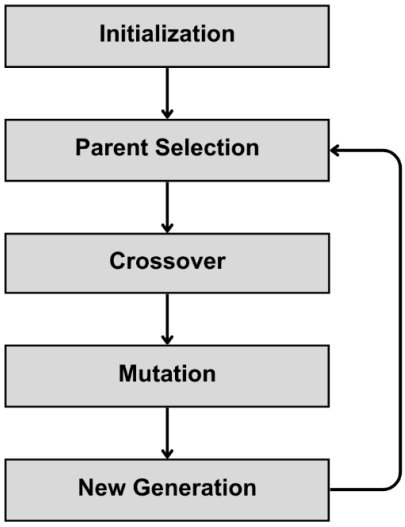

Figure 2: Basic GA structure

The loop begins by generating an initial population of individual solutions. Each GA iteration, called a generation, then proceeds through the following steps:

1. Parent Selection: A set of two parent solutions is chosen from the current population, with higher-fitness individuals more likely to be selected.
2. Crossover: Each pair of parents is combined to produce an offspring by using parts of their chromosomes.

3. Mutation: With a certain probability, each offspring goes through a random mutation to introduce additional diversity, in this case a swap of two operations.

After the mutation, each offspring is decoded into a full schedule, including a timetable for each operation. This schedule is then evaluated to calculate its fitness, which is the inverse of the makespan. So, the higher the fitness, the better the solution quality. From these offspring solutions, the best are then copied into the population of the new generation. This loop repeats until a termination criterion is met, in this case the number of generations.

In addition to this loop, Mlekusch & Hartl (2025) introduce two further characteristics, elitism and hybridization. Elitism, including elite-crossover, is a small set of individuals forming the elite population, which are the best-found solutions in the population. These elite individuals are then copied directly to the new generation as survivors in order to preserve their schedules. Furthermore, if an elite individual is selected during parent selection, an elite-crossover is applied as a modified version of the normal crossover. To hybridize the existing algorithm and create the HGA, a local search heuristic was added. This heuristic is applied to promising individuals, e.g., elite individuals, to explore their neighborhood for better schedules. The newly found schedule is then evaluated and the change will be kept if the fitness improves. This is done repeatedly for a certain number of times. With these additions, the algorithm is able to further improve solutions found by the standard GA. The following pseudocode shows the general GA loop used in both the single-level and multi-level approaches, not including the hybrid versions. The main loop remains the same, only the chromosome structure and the details of the genetic operators differ between the two variants to maintain a fair comparison.

```
1:  population_size = N
2: survivors = crossover_probability * N
3:  oversampling factor = p
4:  generation = 0
5:  population = generate_initial_population(p * N + survivors)
6:  new_generation = generate_initial_population(p * N + survivors)
7:  calculate_fitness(population)
8:  best_fitness = get_best_fitness(population)
9:
10:  while generation < generation_limit do
11:    if generation % 2 == 0 then                // switch between population and new_generation
12:       copy_elite(population, new_generation, survivors)
```

```
13:       while size(new_generation) < p * N + survivors do          // fill remaining with offspring
14:          parent1, parent2 = select_parents(population)
15:          if is_elite(parent1) or is_elite(parent2) then
16:             child = elite_crossover(parent1, parent2)
17:          else
18:             child = crossover(parent1, parent2)
19:          if should_mutate() then                                 //based on mutation probability
20:             mutation(child)
21:          new_generation.append(child)
22:       calculate_fitness(new_generation)
23:       sort_by_fitness(new_generation)
24:
25:    if stopping_criteria_met() then
26:       break
27:    generation = generation + 1
28:
29: if generation % 2 == 0 then
30:    return best_individual(new_generation)
31: else
32:    return best_individual(population)
```

Figure 3: Pseudocode for the basic GA structure

Figure 3 shows the basic GA structure. The initialization step is represented in lines [1] to [10]. It starts by setting population size and how many survivors (elite individuals) are kept each generation, as well as an oversampling factor $p > 1$, which controls how many additional individuals are generated in the population per generation. Then it creates individuals for the population/new generation and evaluates and sorts them based on their fitness. The main loop then starts in line [12] with line [13] setting up the switch between population and new generation for each iteration. Line [14] copies the elite individuals into the next generation. The remaining spots are filled in [15]-[23]: parents are selected in line [16], crossover is applied, with elite-crossover if at least one parent is an elite individual in lines [17] to [20]. Then, the mutation follows in lines [21] to [22], and the child is added [23]. Next, the new generation is evaluated and sorted in lines [24] to [25]. Stopping and generation update are handled in lines [27]-[29]. Finally, the best solution from the last-updated generation is returned in lines [31]-[34].

To compare the effect of representation under the same search process, four variants are used that all follow the loop above and share the same parameters:

- **SL-GA:** single-level representation, no local search
- **SL-HGA:** single-level representation with local search
- **ML-GA:** multi-level representation, no local search
- **ML-HGA:** multi-level representation with local search

The GA settings are consistent across all of the different implementations. It is the method in which an applicant solution is encoded (one level or many levels) and the decision on whether to apply a local search algorithm that changes. Because of this, I will detail both representations in the subsequent section as well as how a valid schedule can be generated using either one.

4.2 Solution Representations

A solution representation for a GA specifies how a solution or chromosome will be encoded and decoded into a solution. A good representation is important in the scheduling problem since both the search space and the definition of genetic operators depend on this representation. Indirect encoding and direct encoding are two basic representations used in GAs to solve scheduling problems (Topcuoglu et al., 2007):

- **Direct encoding:** all the needed information to create a full schedule is already stored in the chromosome.
- **Indirect encoding:** only part of the information is stored in the chromosome, e.g. only operations.

For DRCRFFSP specifically, a direct representation encodes all scheduling decisions. This means the sequence of operations and the assignment of each operation to a specific machine and a specific worker are all contained in the chromosome. Direct encodings can represent any feasible schedule, even poor ones with low fitness values, due to a larger search space. This can lead to crossover and mutation producing equally low-quality or infeasible offspring. By contrast, an indirect representation encodes only part of the solution. For the DRCRFFSP, this means only the operations are encoded in a chromosome and an additional decoding heuristic is used to construct a complete feasible schedule from that information. Due to machines and workers being assigned based on certain rules, the search space gets intentionally restricted. This leads to some solutions being left out compared to direct encoding, biasing a more

promising region of solutions (Mlekusch & Hartl, 2025). The chromosome is therefore simpler, and feasibility is typically guaranteed by the decoding step. This approach has been applied in different scheduling GAs, where a decoder finds good assignments for a given order of operations. For instance, Chamnanlor et al. (2014) used an operation-sequence encoding with a decoding algorithm for a re-entrant flow shop, whereas Han et al. (2021) applied a decoding heuristic for dual resource scheduling.

Beyond the choice of direct/indirect encoding, solution representations can also be classified based on their chromosome design. In a one-string design, the chromosome is a single sequence. While this is typical for indirect encoding, direct encoding can also be designed as a single chromosome, as shown by Lin et al. (2013). They presented two different levels of chromosome encodings. The first is a one-level chromosome design in which all information needed for decoding is stored inside. A problem with this one-layer design is that it mixes different kinds of information in a single sequence. This can lead to abnormal offspring during crossover. Therefore, a multi-level chromosome design is said to be more appropriate for direct encodings, as it avoids this by storing each information type in its own layer.

For dual-resource settings, a three-layer structure is common, with operations, machines and workers sequences as well as layer-specific crossovers/mutations (Gong et al., 2018). Below is an example of three jobs to demonstrate this structure:

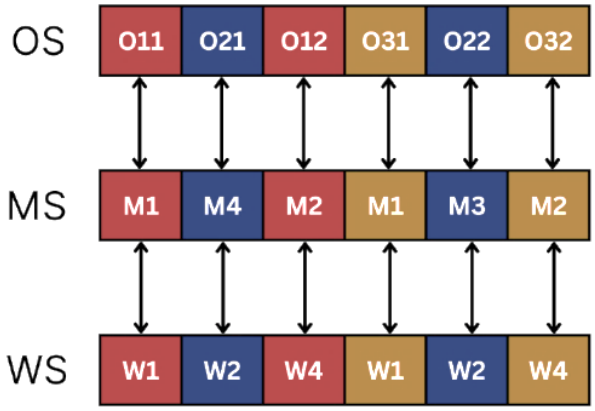

Figure 4: Three-layer chromosome structure

The figure shows three chromosomes: OS (operation sequence), MS (machine sequence), WS (worker sequence). The top row (OS) is the randomly generated operation sequence to be processed: O11, O21, O12, O31, O22. Below each operation is where the MS (machine) and

WS (worker) rows are recorded, with the MS row being the machine chosen for the same location, and the WS row being the worker. This creates an ordered set of complete decision triples, as each column will be the same size and indexed identically. Decoding thus merely positions each set of triples along the time line to produce a viable schedule.

From this, I can describe the particular solutions that were analyzed in the comparative analysis:

Baseline: single-level, indirect representation (SL)

- Chromosome: One string that stores only the operation sequence. No machine or worker choices are encoded.
- Decoding: An earliest start heuristic constructs a feasible schedule by decoding the chromosome for the operations. For each operation, the following process is applied:

    1. Compute the earliest time allowed by precedence (including re-entry).
    2. Select an available machine with the first idle interval that can fit the processing time.
    3. Pick a qualified worker whose idle interval overlaps that machine slot. If several workers are available for the same time slot, a tie-breaking rule is applied to pick the worker.
    4. Assigns start/finish and update machine/worker calendars. If there is no worker that can fit into the time slot, it will move the start time to the next available idle worker and then re-select the machine. This ensures feasibility for every individual by design while maintaining the chromosome as short and simple as possible.

Adaptation: multi-level, direct representation (ML)

- Chromosome: Three strings of equal length are chained together to store three types of information: Operations, Machines and Workers.
- Decoding: Since the decoder uses information stored in the chromosome to determine start times, no resource selection heuristics are required.

4.3 Adaptations to the Single-Level Algorithm

With the two encodings defined, the next step is to align the components of the GA from the SL to the ML version. In developing the ML-GA and HGA, I began with the SL-GA by Mlekusch and Hartl (2025) as a baseline. Since the ML version encodes solutions in three layers rather than a single sequence, several parts of the algorithm had to be adapted. While some remain unchanged, components that manage or manipulate chromosomes must be compatible with multiple layers. I therefore re-implemented these steps of the SL-GA to handle the new chromosome structure. The components in question include population generation, parent selection, crossover and elite crossover, mutation, and the local search heuristic, as well as fitness calculation and decoding. Below, each of them is described together with the differences between the two versions. Additionally, pseudocode is provided to illustrate how they work in the adapted design.

4.3.1 Population Generation

As a first step of the GA, the initial population as well as the new generation of individuals are generated. This step serves as the basis for all subsequent steps that follow. The population size as well as the number of survivors are determined based on the parameters defined in Chapter 4. The way this population is created is by randomly generating each individual, including its operations, machines and workers. Only the first individual is created differently.

In the SL version, only an operation chromosome is created. The first individual follows a chronological job order, meaning that it starts with all operations from the first job, then all operations from the second job until all jobs are filled in. All other individuals are random shuffles of that operation list. Machines and workers are not encoded and are assigned later by the decoder during fitness evaluation. In the ML version, three chromosomes are built: operation_chromosome, machine_chromosome and worker_chromosome. The first individual again keeps a chronological operations order, but adds random valid machines and workers per stage. The remaining individuals also shuffle operations, but again assign valid machines and workers already during this step.

```
1: population = []
2: operation_chromosome = []
3: machine_chromosome = []
4: worker_chromosome = []
5:
6: Generate first individual:
7:     For each operation in job order do:
8:         Assign job ID -> operation_chromosome
9:         Assign random valid machine -> machine_chromosome
10:        Assign random valid worker -> worker_chromosome
11:    Add first individual to population
12:
13: For i from 1 to population_size - 1 do:
14:    Shuffle job IDs of first individual
15:    Track stage progress per job
16:    For each job ID in shuffled list:
17:        Determine current operation based on stage progress
18:        Randomly assign valid machine for stage
19:        Randomly assign valid worker for stage
20:    Add random individual to population
21:
22: For each individual in population:
23:    Calculate fitness
24:    Sort population based on fitness
```

Figure 5: Pseudocode of Population Generation

Figure 5 shows the pseudocode for the population generation. The initialization begins by preparing three empty chromosomes [1]-[4]. The first individual is then constructed in a fixed, non-random way in [5]-[10]. In [6]-[7], operations are written in chronological job order (all operations from job 1, then all operations from job 2, …). For each operation, the algorithm takes a valid machine and worker from the list [8]-[9]. Valid in this case means the resource is allowed for that stage, as every stage has a list of valid machines and workers that are skilled for it. The finished individual is then added to the population [10]. To create the remaining individuals, randomness is introduced in [12]-[19]. First, the operations chromosome from the first individual is shuffled to diversify said chromosome [13]. Because jobs are re-entrant and go through multiple stages, the algorithm tracks per-job stage progress [14], ensuring that when a job ID appears in the shuffled list, the next expected operation for that job is selected [15]-[16]. For each selected operation, the procedure then takes a machine from the stage's valid machine set and a worker from its worker set [17]-[18]. The newly created individual is

also added to the population [19]. This loop then repeats until the population size is reached. Once all individuals are created, they are evaluated in [21]-[22] by scheduling their operations at their earliest feasible start times, while respecting stage order as well as machine and worker availability, to obtain the makespan. Finally, the population is sorted by fitness [23], so that the best individuals appear first and can be preserved later in the GA cycle. Contrary to this, in the single-level variant, steps [8]-[9] and [17]-[18] are absent, as machines/workers are not part of the chromosome. The first individual is still chronological, the rest are random shuffles of the operation chromosomes. During fitness calculation [22], the decoder assigns a feasible resource for each operation before computing the makespan. The final sort [23] is identical.

4.3.2 Parent Selection

Next follows the selection of individuals as parents for the crossover. The selection method used in this thesis is roulette-wheel selection. Roulette-wheel, also called fitness proportionate selection, is a widely used GA method that assigns probabilities proportional to an individual's fitness, so higher-quality individuals are more likely to be picked (Whitley, 1994). The idea behind this is that two parent solutions with a low makespan will produce offspring with an equally low or even lower makespan. How this works is that each parent is given a certain probability, similar to an area on a roulette wheel. The higher the fitness, the bigger the area. The wheel is then spun twice to select the two parents. Because this step depends only on fitness values, not on how chromosomes are encoded, the same routine is used for both SL and ML variants. The pseudocode below displays how this component works.

```
1: selected_parents = []
2: total_fitness = 0
3: for each individual in population do
4:     total_fitness = total_fitness + individual.fitness
5:
6: for i from 0 to 1 do     //to get two parents
7:     random_value = random number between 0 and 1
8:     if random_value == 1,0 then
9:         selected_parents.append(last index of population)
10:    else
11:        partial_sum = 0
12:        for j from 0 to population_size - 1 do
13:            partial_sum = partial_sum + (population[j].fitness / total_fitness)
```

```
14:         if partial_sum ≥ random_value then
15:             selected_parents.append(j)
16:             break
17:
18: return selected_parents
```

Figure 6: Pseudocode of Parent Selection

Figure 6 shows the parent selection before each crossover. The algorithm starts by preparing an empty list for the two selected parents [1]. In the next step, the total fitness of the current population is calculated by summing the fitness values of all individuals [2]-[4]. This value is then used to normalize probabilities in the following loop. The actual selection of parents takes place in [6]-[16]. Since two parents are needed for each crossover, the loop is executed twice. Each time, a random number between 0 and 1 is generated [7]. The algorithm then creates a partial sum [11] and the population is traversed [12]-[15], adding each individual's probability share (fitness / total fitness) to a cumulative sum until that sum meets or exceeds the random value. The index where this happens is appended as the selected parent [15]-[16]. Once both iterations are complete, the two chosen parents are returned [18] and can then be used in the crossover operator.

4.3.3 Crossover & Elite-Crossover

The next GA step after the parent selection is the crossover, which forms the core of the algorithm. It is the process in which new and ideally better solutions are created, and also the part of the code that differs the most between the two solution representations, which is why more emphasis will be laid on it. In this thesis, two main crossover versions are used: a standard and an elite crossover. Additionally, a random crossover is implemented in the ML-GA as a baseline control to test whether observed improvements come from operator design rather than chance.

The standard crossover is used on regular parent individuals. In this, two parents are combined, with both getting parts of their chromosomes copied into the newly created offspring. The deciding factor on what parts of the chromosome are getting copied are two randomly selected numbers r_1 and r_2, between 0 and the number of operations. These numbers, representing the crossover points, define the section to be inserted into the offspring from parent 1. The rest of the offspring is then filled with the remaining data from

parent 2. The other version is an elite crossover. This version is applied if one of the selected parents is an elite individual. The main difference is the way the crossover points are calculated. Instead of selecting two random numbers as before, r1 and r2 are selected from the first and last third, respectively. This is done to retain promising solutions from elite individuals. Otherwise, scenarios in which only fractions of elite individuals are copied to the offspring could occur. This is not ideal, as good solutions should be preserved and more exploration of them encouraged. Therefore, the introduction of an elite crossover proves beneficial for generating better solution quality when a promising solution is selected as a parent. Below is an example of this crossover.

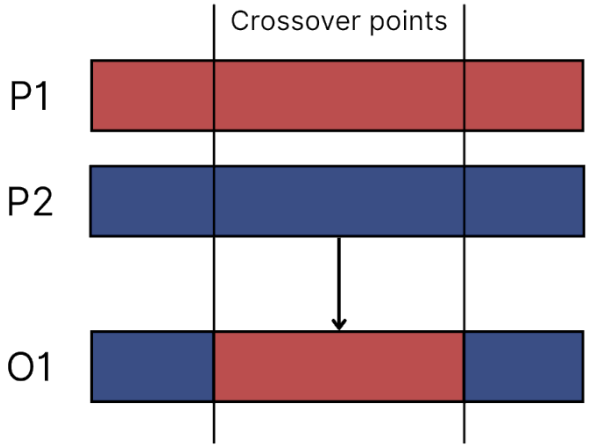

Figure 7: Crossover operator

In the single-level version, the crossover operates only on the operation chromosome, which is a simple permutation of job IDs. A slice of job IDs is copied from one parent, and the remaining positions are filled with the corresponding IDs from the other parent. This process can be seen in Figure 6, where a portion of parent 1 gets copied into the offspring and the rest is filled with operations from parent 2. Similar to the other operators, machines and workers are not part of the operator and are later assigned by the decoding heuristic during fitness calculation. This makes the operator simple and fast, but it pushes more computational load onto the decoder.

In the multi-level version, the crossover must handle three aligned chromosomes (operations, machines, and workers). This means that when a slice is copied, the job IDs as well as their corresponding machine and worker assignments are transferred together. Due to stage constraints, a job occurrence tracker was added for the ML version to know which operation

is being added. The filling process is then similar to the SL version in that the remaining operations come from the second parent, but here the corresponding machine and worker assignments must also be copied, done with the help of the occurrence tracker. For the random crossover version, the procedure is identical up to copying parent 2 assignments into the offspring. Instead of filling in from parent 2, this version takes random machines and workers for the remaining operations yet to be scheduled. This ML version adds complexity, since the feasibility of machine and worker assignments must already be preserved during the crossover itself. While this makes the operator more computationally demanding, it guarantees that the child produced is already a fully specified and schedulable solution.

```
Input: parent1, parent2
Determine better parent by comparing makespan
Select parent1 as fitter, parent2 as the other

Extract operation, machine, and worker chromosomes from both parents

For each gene in parent1.operation_chromosome:
    Track occurrence number of job IDs → parent1_occurrence
For each gene in parent2.operation_chromosome:
    Track occurrence number of job IDs → parent2_occurrence

Initialize array positions[job] = 0 for all jobs

Select two random crossover points rand_1 and rand_2 (ensure rand_1 < rand_2)

Copy segment [rand_1, rand_2) from parent1 into child (all three chromosomes):
    child.operations[i] = parent1.operations[i]
    child.machines[i] = parent1.machines[i]
    child.workers[i] = parent1.workers[i]

Create dummy_parent2 ← copy of parent2.operations

For each copied gene in segment [rand_1, rand_2):
    Identify its job ID and occurrence number from parent1_occurrence
    Find and remove matching occurrence in dummy_parent2 (set to 0)

Initialize:
    front_counter = 0
    back_counter = rand_2

For each gene in dummy_parent2:
    If gene == 0: skip
```

```
33:    target_job = gene, target_occurrence = parent2_occurrence[i]
34:
35:    Try FRONT insertion (before rand_1):
36:       Search parent1 for matching gene and occurrence before rand_1
37:       If found:
38:          Insert gene + parent2's machine/worker at front_counter
39:          Update front_counter and positions[job]
40:          Continue to next gene
41:
42:    If not inserted:
43:       Try BACK insertion (from rand_2 to end):
44:          Search parent1 for matching gene and occurrence
45:          If found:
46:             Insert gene + parent2's machine/worker at back_counter
47:             Update back_counter and positions[job]
48:
49: Output: fully constructed child chromosomes (operations, machines, workers)
```

Figure 8: Pseudocode of Crossover

The pseudocode of the crossover operator is depicted in Figure 8. It begins by taking two parents as input, selecting the fitter one based on makespan, and then extracting the three synchronized chromosomes of both parents [5]. As jobs may occur multiple times, an occurrence tracker is used for each parent's operation chromosome [7]-[10]. The occurrence tracker allows the crossover to know not only which job is represented, but also which occurrence (for example, the second or third time that job occurs). To maintain the proper number of occurrences of each job, the crossover operator tracks both the job ID and the occurrence of the job when the genes are copied and removed. The next step is to initialize a small helper array that will be used to keep track of the number of occurrences of each job that has been placed into the child [12]. The crossover operator selects two random crossover points that determine the segment of parent 1 to be included in the child [14]. This segment is then copied into the child in all three layers so that the alignment of the chromosomes remains intact [16]-[19]. The crossover operator creates a working copy of the operation chromosome of parent 2 [21] and removes the occurrence of each operation copied from parent 1 in the working copy (sets it to zero), so the operation will not be duplicated from parent 2 [23]-[26]. The child is then built in two stages using two pointers. A front pointer is used to fill in the space before the first crossover point, while a back pointer is used to fill in the space after the second crossover point [27]-[30]. The working copy of parent 2's operation

chromosome is scanned in order. When a zero is encountered, the scanning stops and continues at the next non-zero value [31]-[32]. For each gene read from the working copy, the crossover operator checks if there is a valid position available in the front part of the child to insert the gene (i.e., the occurrence of the gene in the fitter parent appears prior to the first crossover point) [35]-[40]. If there is no valid position available in the front part of the child, the crossover operator attempts to insert the gene in the back part of the child [42]-[47]. Each time a gene is inserted, the operation along with the machine and worker entries from parent 2 are copied into the child. Once all of the genes have been inserted, the three chromosomes of the child are complete [49], the middle segment of parent 1 has been preserved, and the remaining segments have been filled in using the genes of parent 2.

4.3.5 Mutation

After the crossover is done, the mutation operator then provides small, targeted changes to maintain diversity and escape local optima. This way, new areas of the solution space can be explored that would likely not be accessible otherwise. The applied mutation method in this algorithm is a two-point swap. For this, two operations from differing jobs get swapped with each other. This way, new changed offspring get created to generate a diverse set of individuals. Though this is not applied to all individuals, only selected individuals are determined by the mutation probability.

As in the other components, the mutation for the SL version only touches the operation chromosome by swapping two operations with each other. The decoder later re-assigns machines and workers during fitness calculation to generate a feasible solution again. The ML version does the same swap of operations, but machine and worker assignments have to be swapped as well. Similar to the crossover, this proves difficult, as the new chromosome sequence must adhere to the stage constraints. It can happen, for example, that the 6th operation of a job gets inserted between the 2nd and 3rd, thus needing a repair of the whole job sequence between the 2nd and the 6th operation. Therefore, a repair mechanism is added to the mutation to ensure the feasibility of the mutated individuals.

```
1:  Input: individual (operations, machines, workers)
2:  o = number of operations
3:  n = number of jobs
4:
5:  Select two distinct random positions first and second
6:  While jobs at both positions are the same:
7:      Reselect second position
8:
9:   j1 = job at position first
10: j2 = job at position second
11: p1 = min(first, second), p2 = max(first, second)
12:
13: Snapshot original machine and worker assignments for j1 and j2:
14:     For each operation:
15:         If job is j1 → store machine & worker in origMach1, origWork1
16:         If job is j2 → store machine & worker in origMach2, origWork2
17:
18: Swap job IDs in operation chromosome at first and second
19:
20: Define repair procedure fixSegment(job, origMach, origWork):
21:     For p = p1 to p2:
22:         If operation_chromosome[p] == job:
23:             Count occurrence (occ) of job up to position p
24:             Replace machine_chromosome[p] = origMach[occ - 1]
25:             Replace worker_chromosome[p] = origWork[occ - 1]
26:
27: Apply fixSegment for j1 using origMach1 and origWork1
28: Apply fixSegment for j2 using origMach2 and origWork2
29:
30: Recalculate fitness of the mutated individual
```

Figure 9: Pseudocode of Mutation

Figure 9 shows the mutation of the individuals during the algorithm. The operator starts off by selecting two distinct positions in the operation chromosome and ensures they belong to different jobs, so the move is meaningful [5]-[10]. Similar to the occurrence tracker for the crossover, a snapshot is created of the original machine/worker assignments for every occurrence of the two affected jobs, j_1 and j_2 [13]-[16]. It then swaps the job IDs at the chosen positions in the operation chromosome [18]. This swap can desynchronize the three layers within the segment between the two cut points, so a targeted repair is applied only between p_1 and p_2, which are the two selected positions [20]-[26]. For each position p in that segment where the gene equals $j \in \{j_1, j_2\}$, the procedure counts how many times job j has appeared up

to p (its occurrence index) and restores the corresponding machine and worker from the snapshot [23]-[25]. This keeps the original assignments intact, while reflecting the new chromosome sequences created by the swap. The repair is executed once for j_1 and once for j_2 [27]-[28], leaving all other jobs untouched. Finally, the individual's fitness is recomputed [30].

4.3.6 Local Search Heuristic

To hybridize the standard GA, a local search heuristic is added to the algorithm. This local search is added only in the HGA as an intensification step at the beginning of each generation. Similar to the mutation operator, the idea is to take already good individuals and try small, targeted changes in order to improve them even further. Following the baseline SL-HGA, this step is restricted to the top 10% of the elite population. In each generation, individuals are ranked by their fitness and the subset of elite individuals is taken to apply the local search heuristic. Each solution is then explored with a bounded number of swap moves (Mlekusch & Hartl, 2025). The way it works, similarly to the mutation operator, is that it applies a two-swap heuristic to the chromosome to find new improvements. The difference is that it does not necessarily keep this new solution, but re-evaluates it. If the swapped solution improves fitness, it replaces the current one. Otherwise, another swap is done on the individual until it reaches a predefined number of moves. For this component, the same differences between SL and ML-GA apply as for mutation.

```
Input: individual (operations, machines, workers)
o = number of operations
maxMoves = 10% of o

Store current best fitness and makespan
Store current scheduled_jobs as prefix

For k = 1 to maxMoves do:
    Pick two random positions: first, second
    While jobs at both positions are the same:
        Reselect second

    j1 = job at position first
    j2 = job at position second
    p1 = min(first, second), p2 = max(first, second)

    Save full copies of operation, machine, and worker chromosomes
```

```
18:
19:    Snapshot original machines and workers of j1 and j2
20:       For each operation:
21:          If job is j1 → save in origM1, origW1
22:          If job is j2 → save in origM2, origW2
23:
24:    Swap job IDs at first and second in operation chromosome
25:
26:    Define repair procedure fixSegment(job, origMach, origWork):
27:       For p = p1 to p2:
28:          If operation_chromosome[p] == job:
29:             Count occurrence occ of job up to p
30:             machine_chromosome[p] = origMach[occ - 1]
31:             worker_chromosome[p] = origWork[occ - 1]
32:
33:    Apply fixSegment for j1 and j2
34:
35:    Recalculate fitness starting from p1 with current prefix
36:
37:    If new fitness ≥ previous best:
38:       Accept solution → update best fitness, makespan, and prefix
39:    Else:
40:       Revert chromosomes and restore previous prefix, fitness, and makespan
41:
42: End For
```

Figure 10: Pseudocode of Local Search Heuristic

The local search heuristic in Figure 10 is first used as an initial improvement for each generation. Similar to the mutation, this process employs the swap-and-repair process, however, the algorithm will continue to use this procedure to generate potential moves until it has reached a predetermined move limit. Once the number of moves (which is set to be 10% of the total operations), the current best fitness, and a portion of previously scheduled jobs [2]-[6] are saved, the algorithm will enter into a loop that will evaluate the possible moves [8]. In the evaluation loop, the algorithm will select two random position values from jobs which are not related [9]-[15], save the previous chromosome state [17], capture the machine and worker assignments for both jobs that were affected by the previous move [19]-[23], swap the job ID's on the operation layer [24] and apply the segment repair (similar to the mutation operator) between p1 and p2 to align the machines and workers with the swapped operations [26]-[34]. Next, the algorithm will recalculate the fitness of the current solution starting at p1 using the saved prefix (all jobs prior to p1 will not be changed) [35]. If the newly calculated

fitness is greater than the current best fitness, the new move is accepted and the best fitness is updated [37]-[39]. If the newly calculated fitness is less than or equal to the current best fitness, all changes made during the last move will be undone [40]. The loop will repeat itself until the number of swap moves is met [42].

4.3.7 Fitness Calculation & Decoding

A central step of the GA is the evaluation of individuals, where each chromosome is decoded into a concrete schedule and its fitness is calculated as the inverse of the makespan. It is also the component that changed the most when moving from the single-level to the multi-level GA together with the crossover.

Both versions use an occurrence counter to map the kth occurrence of a job in the operation chromosome to its stage k. Following each scheduling operation, the decoded operation will be added to scheduled_jobs_ along with the start time of the job, the machine it was processed on, and the worker who processed the job. The decoder will also add the earliest start time for the job's successor and update the current makespan.

Both versions will allow for prefix replay. Prefix replay allows the user to re-schedule from the middle of the chromosome. This is particularly beneficial when performing a local search, as the only segment of the sequence that will be modified is a small portion near the end of the chromosome. Once the decoder has identified the stopping index for the perturbation, all of the operations before the stopping index may be re-scheduled by simply replaying them using the schedule_already_scheduled method. The schedule_already_scheduled method is a very low overhead function that will copy the previous start times and assignments for the operations being re-scheduled. Scheduling may then continue in the normal manner from the stopping index forward.

Ultimately, the main computational trade-off between the two models is that the SL model will spend more time evaluating each operation as it is forced to evaluate multiple machines and workers for each job, whereas the ML model will evaluate each operation quickly since no evaluation need take place. However, if the chromosome is inconsistent, the ML model will incur the penalty of generating infeasible solutions due to mis-allocated resources.

In short, the way fitness was calculated changed from "the algorithm determines resource needs when it decodes" in the SL version to "the algorithm respects resource needs that are encoded in the chromosome" in the ML version. Both versions of the algorithm use the same main components, but the ML version moves the complexity of determining resource feasibility from the decoder to the genetic operators, who are responsible for ensuring that the chromosomes they create will be valid.

4.4 Summary of Methodology

In this chapter I will explain the methodologies used in the study to analyze how the manner in which solutions are represented would affect GAs for the DRCRFFSP. In addition to using the single level baseline of Mlekusch & Hartl (2025), I developed a multi-level (ML) version that retains the entire GA loop as well as all parameters of the original GA, however the encoding method was changed and, therefore, many parts of the GA were modified. The comparisons of these four "like for like" versions of GAs were made based upon the SL-GA, SL-HGA, ML-GA, and ML-HGA.

Although the main loop (population → selection → crossover → mutation → evaluation) is the same for each of the versions of GAs studied here, the primary difference among them is their representation of the solution.

- SL: one chromosome stores only the operation order. Machines and workers are assigned by a decoding heuristic during fitness calculation.
- ML: Operations, machines, and workers are all encoded into an aligned chromosome. Thus, individuals are completely defined before scheduling occurs.

Because of this change, several components were adapted:

- Population generation: SL shuffles operations to obtain initial schedules and saves them to the chromosome. ML does the same while also assign valid machines/workers per stage for each operation.
- Parent selection: Remains unchanged between SL and ML (roulette wheel).

- Crossover: SL operates on a single chromosome. ML operates on three simultaneously and maintains occurrence tracking to place the remaining genes. A random-assignment crossover was added in ML to observe effectiveness of operator design.
- Mutation: SL swaps two operations. ML performs the same swap but repairs machines/workers within the affected segment keep stage constraints intact and ensure feasibility.
- Local search (HGA only): Uses the same swap operator from Mutation and has therefore the same changes from SL to ML.
- Fitness & decoding: SL chooses resources during decoding (heavier per operation). ML respects preselected resources and only finds start times (lighter per operation), shifting responsibility for feasibility into the operators.

In total, this chapter defined an experimental environment: the same genetic algorithm (GA) configuration and loop, where the difference was restricted to the representation and the least amount of operator modification necessary to enable the new representation. It provides the basis for the experimental method used in Chapter Three.

5. Experimental Design

The research conducted here utilizes an experimentally-controlled design. Each representation runs with the same parameters, stopping rules, and objective function (makespan minimization). For each of these, the total runtime is measured across all runs. The benchmark problems used have realistic variability in the attributes of jobs, the re-entrance level of jobs, available machines and labor flexibility, thus allowing us to assess how well the efficiency of schedules has been improved when compared to real-world applications. The SL version will serve as the control group to provide a solid base-line to help allow me to test if the ML based framework delivers superior performance for the DRCRFFSP through the use of direct multi-level chromosomes.

This chapter describes the experimental setup for this comparative study. In the first part, it explains implementation details. These include: programming environment, hardware specifications, and parameter settings that will allow reproduction of results. In the second

part, benchmark instances are presented. They cover design, size categories, and levels of complexity related to the DRCRFFSP. In the third part, evaluation criteria and stopping rules are explained as a way of measuring and comparing Performance between the GA variants. Finally, Chapter 5 ends with an overview of the experiment design. This defines scenarios and comparisons that guide the analysis of results in the next chapters.

5.1 Implementation Environment

The algorithms developed in this thesis were implemented in C++. The reason for the implementation of the algorithms in C++ was to ensure consistency in comparison with the baseline SL-GA of Mlekusch & Hartl (2025) by using the same programming language as the baseline SL-GA. The only differences between the two versions should be the chromosome design and the adapted operators.

The ML-GA and HGA were compiled in Microsoft Visual Studio (C++ 17) following the same project structure and coding conventions as the SL code base. Time taken at run-time was captured using the C++ clock, which captures both total run time and component-level timings (e.g., crossover, mutation, fitness calculation). This ensures consistency in environment and thereby ensuring that any observed performance differences can be attributed to changes in representation and operator, rather than hardware or tools used.

All experiments were conducted on a Lenovo Legion workstation with an AMD Ryzen 7 5800H processor (8 cores, 16 threads, 3.2 GHz base clock), 16 GB RAM and running the Windows 11 operating system.

5.2 Parameter Settings

To enable a comparable evaluation of both versions of the algorithms, the ML-GA has been set up using parameters that have been chosen according to the ones used in the SL-GA by Mlekusch & Hartl (2025) in order to keep a consistent methodology and allow the differences in performance to be assigned mainly to the solution representation instead of the external parameters. Therefore, the parameters described below represent the reference point for all experiments conducted in the scope of this thesis.

Population size and number of elite individuals, as well as mutation and crossover rate and the termination condition are the most important parameters of a genetic algorithm. Therefore, the following parameters have been applied for the ML-GA: Population size = 50, offspring generated in each generation = 1.5 × population size, mutation rate = 0.05, crossover rate = 0.1 (same as in the SL-Baseline) and 5 elite individuals reserved per generation to secure the survival of the best solutions.

Stopping criteria follows a dual approach, with a time limit as well as a limit of generations during the run. For the runtime limit, there is a difference between small and large instances. Small instances are capped at 300 seconds, whereas large instances are capped at 600 seconds. For the second stopping criteria, a number of generations without no further improvements ends the run. This is set at 3000 generations for both instance sets before the run stops.

Baseline parameters used here are derived from implementation of SL and prior GA studies in scheduling (Mlekusch & Hartl, 2025; Murata et al., 1996; Reeves, 1995) with additional testing done with very small problem instances to test feasibility:

Parameter	Small Instances	Large Instances
Population size	50	50
Elite individuals per generation	5	5
Mutation probability (%)	5	5
Individuals created by crossover	75	75
Number of runs per instance	30	30
Stopping criterion (generations)	3000	3000
Stopping criterion (time, sec)	300	600

Table 2: Parameter settings for the GA/HGA

5.3 Benchmark Instances

Benchmark problems for the evaluation of the performance of the proposed Meta-Heuristics based Genetic Algorithm (ML-GA), as well as for a comparison of its performance with that of the established Single Level (SL) baseline, are represented by a collection of standard problems, referred to as Benchmark Problems. These Benchmark Problems, as described above, were developed by Mlekusch (2024) and are publicly accessible at https://data.mendeley.com/datasets/73hn9bxxvv/2. They represent a real-world screen-printing environment with varying numbers of jobs, stages, machines and workers. The objective across all tests is to minimize the makespan. By including both small and large problem sets, the benchmark covers realistic variation in job numbers, stages, machine availability, re-entry depth, and worker flexibility, making it well-suited for testing the performance of metaheuristic algorithms in industrially motivated settings.

The instances follow a structured design, with each being defined by four main parameters:

- **Jobs:** number of jobs to be scheduled.
- **Stages:** number of processing stages in the re-entrant flow.
- **Machines per stage:** for each stage, between 1 and 5 parallel machines are available.
- **Workers:** workforce size and skill distribution. Each worker is assigned to a subset of stages, reflecting partial flexibility.

Each name of an instance contains the primary characteristics of that instance. For example, the name "Instance_10_5_2_2" indicates that this instance includes ten jobs, five stages, two levels of re-entry and two versions of this instance. The final number in each instance name denotes its version. Every single instance has a standardized format. Each instance starts with one header-line providing the major parameters: the number of jobs, the number of stages, the number of machines, the number of workers, the number of re-entrances, the total number of operations, and the sum of all the processing times. Following the header-line, there is a part of the instance describing the stages: in particular, how many machines and how many workers have been assigned to each stage. Then follows the jobs section where each job is defined as being composed of a number of operations and for each operation the stage number and its processing time are given. At the end of the jobs section the worker-section

describes the qualification matrix. In other words, it indicates for which stages, a specific worker is qualified to work.

The above-mentioned structure makes sure that every single instance represents the three main features of the DRCRFFSP:

1. Job routing, including re-entrant flows, where operations may revisit the same stage multiple times.
2. Machine flexibility, since stages can contain parallel machines.
3. Worker flexibility, with each worker qualified for a subset of stages.

5.4 Evaluation Metrics

There are two key metrics to use to evaluate how well the methods developed for scheduling performed. The first was identified previously as the main method, the second is another (secondary) method. For this thesis, the primary metric to be utilized to determine how well the schedules perform is the makespan, or Cmax, which defines the completion time of the last operation in the schedule. Additionally, the amount of time it takes to compute each run is also measured as a secondary metric. Because scheduling heuristics can typically be found in many real-world applications that require timely answers, measuring runtime provides an alternate way of assessing the operational effectiveness of the algorithms.

The GA's fitness function is defined as the inverse of the makespan as follows for s, where s is the individual to whom the fitness is being assigned:

$$fitness(s) = \frac{1}{c_{max}(s)}$$

Using this formulation, individuals in a population which have a lower total makespan will tend to have a higher fitness value, thus they are more likely to be selected for continuation through the evolutionary cycle of the GA.

5.5 Comparative Analysis Procedure

For the comparative analysis, five algorithm variants are tested:

- **SL-GA**: single-level representation, no local search.

- **SL-HGA**: single-level representation with local search.
- **ML-GA**: multi-level representation, no local search.
- **ML-HGA**: multi-level representation with local search.
- **ML-GA (random crossover)**: multi-level representation with a randomized crossover operator. This variant serves as a control baseline to isolate the contribution of the problem-specific crossover.

Three main comparison points provide structure to the evaluation:

1. **SL vs. ML:** This will demonstrate how well the multi-level adaptation of the representation performed against the base-line Single-Level (SL) implementation.
2. **GA vs. HGA:** This allows the added intensity provided by the local search to be isolated as a means to improve the representation.
3. **Small vs. Large:** This will illustrate how scalable both the representations were on two different sets of problem sizes.

The same parameter settings and termination conditions were utilized across all variations of the algorithm to allow for a fair basis for comparing the effects of varying the representation and operators of the algorithm. The ML-GA with random crossover was only tested with the small instance set and only within the Standard GA framework, since it serves solely as a base line for measuring how much performance is improved by using the random crossover operator in conjunction with the tailored crossover operator in the ML-GA. All evaluations of runtime and makespan were made for each run of all five algorithmic variations. A detailed presentation of the analysis' results is found in Chapter 6.

6. Results and Discussion

Building on the controlled setup from Chapter 5, this chapter now presents and discusses the performance of the algorithm variants stated in Chapter 5.5. The goal is to understand whether a multi-level encoding (ML) leads to better schedules and/or quicker results than a single-level encoding (SL) under the same GA loop. Furthermore, it is important to find out what drives the differences between the variants. For this, I compare the four main variants SL-GA & SL-

HGA / ML-GA / ML-HGA and include a random-crossover ML-GA for the small instance set as a control.

The analysis is guided by four questions:

4. Can the algorithm variants reach the optimum on small instances where an exact benchmark is available?
5. How close are the variants when they do not reach the optimum?
6. Do representation choices (single-level vs. multi-level) and local search make a practical difference?
7. How do these effects change as instances grow larger and more complex?

To achieve this, I proceed in two steps. First, Section 6.1 shows the small-instance results against optimal results. For each instance, I show the best and average makespan together with the runtime and display the solution quality as the gap to the optimum. Additional graphs are provided to further demonstrate the differences in performance between the variants. Second, Section 6.2 covers the large-instance set, where no exact benchmark exists. Here, results are not compared to the optimal solution, but between the variants, using the same metrics for better comparison. Section 6.3 then summarizes the findings in order to highlight the contribution made by the choice of representation (SL vs. ML) and hybridization (GA vs. HGA).

6.1 Results for Small Instances of the DRCRFFSP

To begin with, the performance of all four algorithm variants is first evaluated on the small instance set. For this comparison, optimal solutions obtained from the CP model by Mlekusch & Hartl (2025) are available as benchmarks. This allows for a direct assessment of how close each of the approaches comes to the optimal solutions and how efficiently they reach them. Tables 3 and 4 present the results for the SL and ML variants, respectively, showing both the average and best makespan gaps to the CP benchmark as well as the average computation time per run for 30 runs per instance.

	CP	SL-GA			ML-GA			ML-GA (random crossover)		
Instance	Optimum	Gap to CP (Avg %)	Gap to CP (Best %)	Time (s) per run	Gap to CP (Avg %)	Gap to CP (Best %)	Time (s) per run	Gap to CP (Avg %)	Gap to CP (Best %)	Time (s) per run
10_5_2_2	156	4%	1.9%	7.3	13.6%	10.9%	3.3	15.3%	11.9%	3.8
10_5_2_4	165	3.3%	0.6%	7.0	11.5%	7.8%	3.4	12.4%	7.5%	3.8
10_5_3_3	194	10.7%	8.5%	12.0	17.9%	15.3%	4.8	20.7%	18.0%	5.3
10_10_2_1	321	3.0%	1.5%	18.9	12.9%	9.1%	9.2	14.4%	11.9%	9.2
10_10_2_3	301	6.9%	3.5%	18.9	20.6%	17.8%	8.8	23.6%	20.0%	9.0
20_5_2_0	290	0.0%	0.0%	15.9	6.6%	4.0%	9.1	7.9%	5.4%	10.2
20_5_2_1	268	4.6%	1.8%	20.9	12.9%	10.1%	9.0	15.8%	12.1%	10.2
20_5_3_1	404	0.1%	0.0%	24.8	1.8%	0.2%	12.8	1.9%	0.5%	13.6
30_5_2_1	389	0.0%	0.0%	27.6	1.7%	0.3%	16.8	3.2%	1.1%	17.3
30_5_3_2	545	6.3%	5.2%	53.2	11.7%	7.5%	26.7	14.8%	10.9%	29.5
40_5_2_4	519	0.0%	0.0%	41.7	5.5%	3.0%	26.9	7.1%	4.9%	26.4
50_5_3_2	1011	0.0%	0.0%	87.4	1.0%	0.2%	53.6	1.4%	0.4%	60.9
Average		3.2%	1.9%	27.9	9.8%	7.2%	15.3	11.5%	8.7%	16.6

Table 3: Results of SL-GA and ML-GA for small DRCRFFSP instances.

	CP	SL-HGA			ML-HGA		
Instance	Optimum	Gap to CP (Avg %)	Gap to CP (Best %)	Time (s) per run	Gap to CP (Avg %)	Gap to CP (Best %)	Time (s) per run
10_5_2_2	156	1%	0.0%	12.1	13%	9.3%	5.2
10_5_2_4	165	0.0%	0.0%	9.6	10.8%	6.3%	5.3
10_5_3_3	194	4.9%	3.5%	21.7	17.9%	13.8%	8.4
10_10_2_1	321	1.2%	0.0%	39.4	12.2%	9.1%	20.1
10_10_2_3	301	2.0%	0.3%	40.3	19.5%	15.7%	19.1
20_5_2_0	290	0.0%	0.0%	32.4	5.4%	2.4%	19.2
20_5_2_1	268	0.5%	0.0%	43.4	12.3%	9.2%	18.6
20_5_3_1	404	0.0%	0.0%	58.0	1.4%	0.2%	29.0
30_5_2_1	389	0.0%	0.0%	67.9	1.7%	0.0%	40.8
30_5_3_2	545	1.7%	1.3%	180.0	10.6%	6.7%	78.0
40_5_2_4	519	0.0%	0.0%	137.2	5.4%	1.3%	78.5
50_5_3_2	1011	0.0%	0.0%	347.8	0.8%	0.1%	139.1
Average		0.9%	0.4%	82.5	9.3%	6.2%	38.4

Table 4: Results of SL-HGA and ML-HGA for small DRCRFFSP instances.

On all of the small test cases, the single level versions of the algorithm produce better quality solutions than the ML versions. For example, on 5 out of 12 test cases, SL-GA found the optimal solution for the CP problem. On the remaining 7 test cases, it was able to find a solution that was within an average of 3.2% of the optimum solution (with the best-of-run results being 1.9%). Using local search with the SL-GA algorithm was shown to be very helpful, as it produced optimal solutions on 8 of the 12 test cases and reduced the average gap between the solution produced by the SL-HGA and the optimum solution to 0.9% (with the best-of-run results being 0.4%). However, running the SL-HGA algorithm is significantly slower than

running the SL-GA version, requiring approximately 82.5 seconds of computational time per run (as opposed to 28 seconds for SL-GA). The multi-level versions of the algorithms are considerably faster than their single level counterparts, but they do not provide as good of solution quality as the single level versions of the algorithms on the small test cases. ML-GA averages 9.8% (best-of-run at 7.2%) at 15.4s per run, and ML-HGA improves this slightly to 9.3% (best-of-run 6.2%) with a per-run time of 38.4s. The ML-GA with random crossover performs worst among all GA variants, with an average gap of 11.5% (best-of-run at 8.7%) and a similar runtime of 16.6s per run and thus remains behind both the SL and ML variants. Notably, of the ML variants, only the ML-HGA gets close (≤ 0.3% gap) on a few instances (e.g., 20_5_3_1 / 30_5_2_1 / 50_5_3_1) and matches CP exactly once (30_5_2_1), but it does not dominate overall on this set.

The SL representation generally has a better schedule for small instances because it has to search for resource pairings during decoding. It does so at a cost of longer evaluation time. On the other hand, the ML representation is faster at evaluating. However, the difference in solution quality remains consistent over all instance sets. These findings are consistent with the trade-offs associated with designing a SL representation as discussed in Chapter 4.2, where the GA spends additional time finding a good pairing of resources during the decoding process that results in better solutions on smaller problems. Additionally, the random-crossover ML-GA indicates that this gap is not only due to the representation itself, but also due to operator design. When a purely random recombination replaces the ML crossover, the average gap increases from 9.8% to 11.5% while the runtime remains similar.

The hybridization in form of a local search operator seems to help both representations, but its impact is stronger for SL. Moving from GA to HGA cuts the average gap from 3.2% to 0.9% for SL, versus 9.8% to 9.3% for ML. In other words, local search appears to enhance the SL advantage for small instances, whereas for ML it yields slight quality improvements while keeping the runtime well below that of SL-HGA.

To better understand how these quality differences arise, Figures 7 and 8 illustrate the convergence behavior of the GA variants on a representative small instance (10_5_2_2). For each generation, the best, average and worst makespan in the population are recorded for SL-GA and ML-GA over the first 200 generations. Similar curves were obtained for several other small instances and random seeds, thus demonstrating a representative pattern.

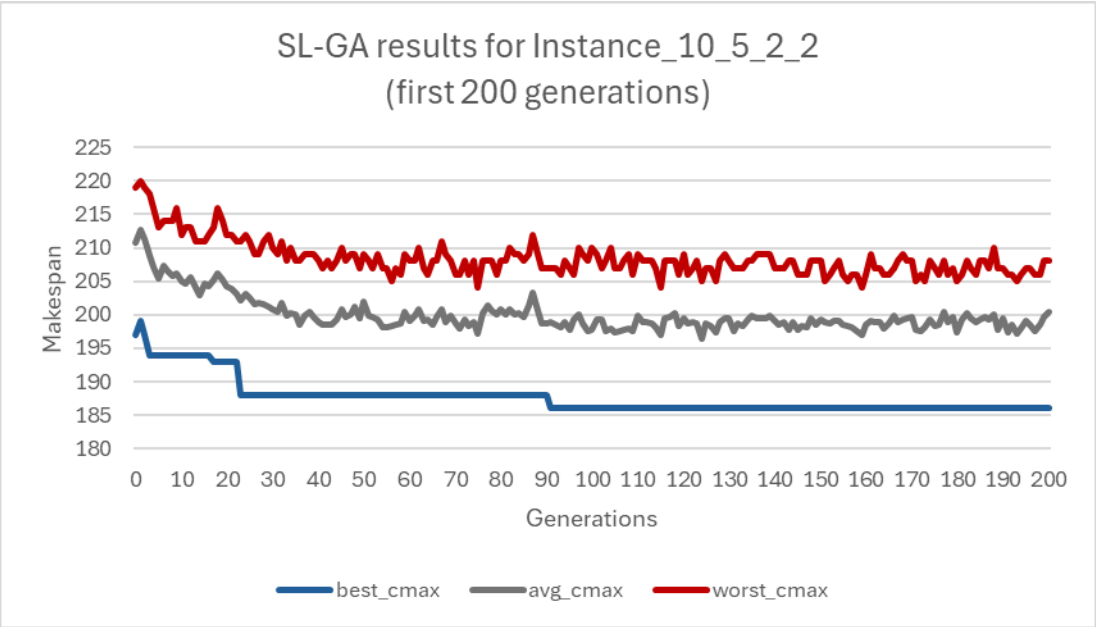

Figure 11: Convergence plot of SL-GA

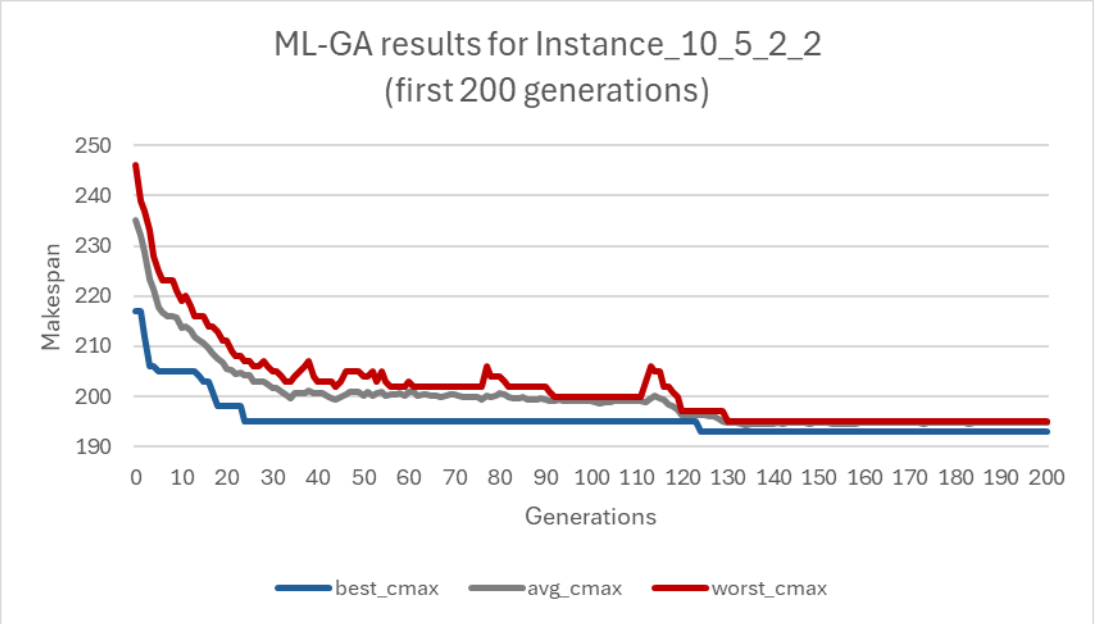

Figure 12: Convergence plot of ML-GA

The SL version's population exhibits an obvious distribution between the best (bottom), average (middle) and worst (top) individuals as far as the best makespan is concerned. The best makespan grows very quickly in the first generations and then with slightly slower increases until approximately generation 200. Both the average and worst distributions are noticeably higher than the best distribution showing that there is still considerable diversity in the population and that the GA is exploring new areas of the search space.

In contrast, the multi-level GA converges much more quickly. After an initial improvement phase in the first 120 to 130 generations, the average and worst makespan curves collapse close to the best curve and all three remain almost flat afterwards. This means that the population becomes homogeneous early on and further generations produce marginal changes in makespan. The reduced variance in the ML population can be explained by the design of the representation and operators. For each gene, not only is the operation order fixed, but also its machine-worker assignment, and the crossover tends to preserve long segments of these assignments. Beneficial resource patterns, therefore, spread rapidly through the population, and most individuals share very similar machine-worker choices as a result. As a result, ML shifts from exploration to exploitation earlier than SL, which helps runtime but can also explain why ML tends to get trapped at a higher makespan on these small instances.

Following the analysis of the convergence behavior, I have provided two figures (one for GA and one for HGA) showing per-instance run times for both variants. The figures emphasize computational effort instead of solution quality and are designed to provide a clear view of the time required for each variant to solve each instance thereby providing the opportunity to easily compare the performance of all variants over all instances.

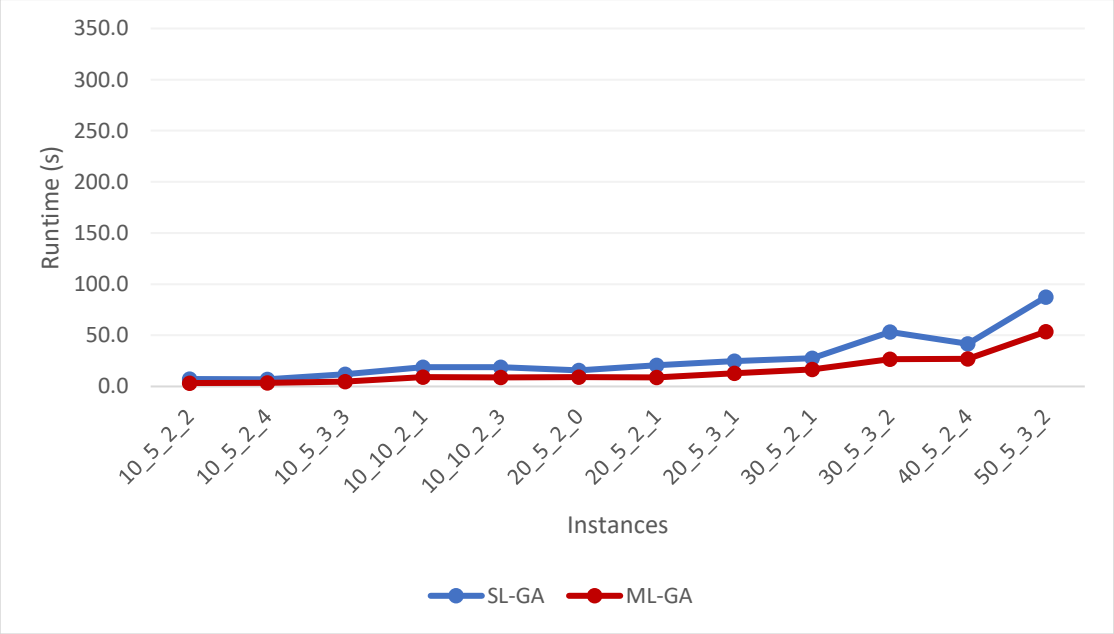

Figure 13: Runtime comparison between SL-GA and ML-GA

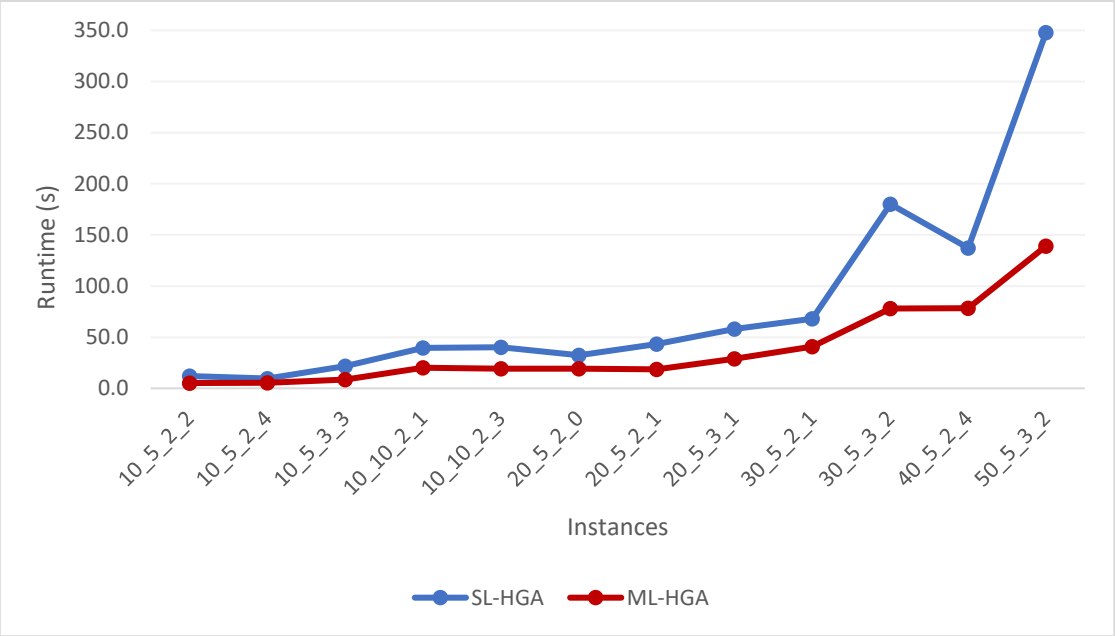

Figure 14: Runtime comparison between SL-HGA and ML-HGA

Figures 9 and 10 show how much time is required by the two types of GAs (SL vs.ML). One can see that the ML-GAs (red) are always faster than the SL-GAs (blue), particularly on larger instances. The primary explanation for this speedup comes from the fact that the decoding process in the SL versions is expensive because they assign machines and workers during the fitness evaluation process. However, the ML versions have the machine and worker assignments in their chromosomes and therefore do not require additional computation during fitness evaluation except for the feasible start time calculation for each operation. While both types of GAs scale similarly, there is clearly an advantage of the ML version, particularly for the HGA version where local search further increases the cost of evaluation.

For the small test cases where the optimal solutions were obtained via CP, the highest solution quality was delivered by the single level representation (SL-HGA: average gap = 0.9%, best-of-run = 0.4%) while the ML representation evaluated faster, but had a larger average gap to the optimum. Local search improved the quality of both representations. However, it demonstrated a stronger improvement for the SL versions.

I will next turn to the large-instance test case to determine if the gap in solution quality between the SL and ML representations continues to exist as problem size grows, and if the faster computation of ML versions translate into higher-quality solutions at larger scales.

6.2 Results for Large Instances of the DRCRFFSP

For the large instance set, no optimal solutions through CP are available. The comparison, therefore, shifts to a performance comparison across algorithm variants only. With the large amount of data to be processed for each instance, the stopping criteria shift to time limits now instead of the number of generations. All runs use the same time cap of approximately 600s to ensure equal search time. Therefore, runtime will not be considered anymore in this section. For each instance and variant, I report the average and best makespan over 30 repeated runs, together with a relative gap, shown for both averages and best result values. Tables 5 and 6 summarize these results for GA and HGA, respectively.

	SL-GA		**ML-GA**		**ML vs. SL gap (Δ%)**	
Instance	Avg. Makespan	Best Makespan	Avg. Makespan	Best Makespan	Δ% Avg	Δ% Best
60_20_4_0	1 636	1 624	2 146	2 115	31%	30%
60_25_4_0	1 731	1 717	2 933	2 860	69%	67%
60_30_4_0	1 959	1 934	3 025	2 979	54%	54%
70_20_4_0	1 959	1 942	2 738	2 630	40%	35%
70_25_4_0	1 981	1 967	2 771	2 736	40%	39%
70_30_4_0	2 122	2 106	3 094	3 051	46%	45%
80_20_4_0	2 086	2 074	2 442	2 418	17%	17%
80_25_4_0	2 085	2 070	2 490	2 467	19%	19%
80_30_4_0	2 195	2 172	3 033	3 000	38%	38%
90_20_4_0	2 295	2 286	2 679	2 655	17%	16%
90_25_4_0	2 265	2 261	2 594	2 557	15%	13%
90_30_4_0	2 405	2 392	2 999	2 958	25%	24%
100_20_4_0	2 417	2 406	2 798	2 759	16%	15%
100_25_4_0	2 721	2 712	3 256	3 218	20%	19%
100_30_4_0	2 811	2 790	3 727	3 639	33%	30%
Average	2 178	2 164	2 848	2 803	31%	30%

Table 5: Results of SL-GA and ML-GA for large DRCRFFSP instances.

Δ% = (ML − SL) / SL × 100%

	SL-HGA		ML-HGA		ML vs. SL gap (Δ%)	
Instance	Avg. Makespan	Best Makespan	Avg. Makespan	Best Makespan	Δ% Avg	Δ% Best
60_20_4_0	1 640	1 620	2 140	2 103	30%	30%
60_25_4_0	1 725	1 709	2 916	2 867	69%	68%
60_30_4_0	1 946	1 929	3 015	2 970	55%	54%
70_20_4_0	1 953	1 941	2 727	2 642	40%	36%
70_25_4_0	1 969	1 953	2 765	2 723	40%	39%
70_30_4_0	2 099	2 076	3 086	3 037	47%	46%
80_20_4_0	2 082	2 070	2 433	2 403	17%	16%
80_25_4_0	2 077	2 063	2 480	2 437	19%	18%
80_30_4_0	2 175	2 161	3 026	2 980	39%	38%
90_20_4_0	2 297	2 279	2 666	2 637	16%	16%
90_25_4_0	2 268	2 261	2 582	2 556	14%	13%
90_30_4_0	2 385	2 367	2 985	2 952	25%	25%
100_20_4_0	2 420	2 406	2 796	2 736	16%	14%
100_25_4_0	2 699	2 672	3 241	3 225	20%	21%
100_30_4_0	2 783	2 763	3 728	3 667	34%	33%
Average	2 168	2 151	2 839	2 796	31%	30%

Table 6: Results of SL-HGA and ML-HGA for large DRCRFFSP instances.

$\Delta\% = (ML - SL) / SL \times 100\%$

The SL approach is clearly superior to the ML approach in terms of quality of solutions produced over all tested problem sizes. When comparing the results from both approaches using the standard GA without local search (as shown in table 5), SL-GA achieved a make span of 2,178 while ML-GA was at 2,848. That shows there is an average of 31% difference between the two methods (30% for BO-runs). This trend also occurred with the addition of local search (table 6), as SL-HGA had an average makespan of 2,168 while ML-HGA averaged 2,839, or another 31% average difference (30% for BO-runs). Therefore, with larger problems, the "best" solution generated by the multi-level variant is always inferior to the "best" schedule developed by the single level.

The relative difference varies across the instances. Grouping by the number of stages shows that the ML-SL gap widens as the stage count increases. At 20 stages (e.g., instances like 60_20_4), the difference between ML and SL-GA is about 24%. At 25 stages, the number rises to 33% and up to 39% for 30 stages for both GA and HGA. By contrast, increasing the number of jobs tends to narrow the gap. While being at a 52% and 42% difference for 60 and 70 jobs, respectively, the number decreases to 25% and 19% for 80 and 90 jobs, before going up again to 23% for 100 jobs. This mirrors the small-set results. As the number of stages rises, together

with an increase in re-entrance, the direct ML encoding seems to struggle more. By contrast, when the job count increases, both representations have more opportunities to improve within the same time limit, so the gap between them narrows. This stems from the ML variants being more time-efficient, thus going through more generations in the same runtime of 600s.

The large dataset results indicate that local search has a negligible effect on the overall performance. On average, SL-HGA outperformed SL-GA by only a little (2 178 vs 2 168) while ML-HGA was just slightly better than ML-GA (2 848 vs 2 839) as well. The average ML-SL difference did not change much at 31% when using either a genetic algorithm or hybrid genetic algorithm. This is completely different from the smaller data sets, where there was a considerable improvement for SL through the use of local search. Since there was a time limit placed on the larger data set, local search resulted in an insignificant improvement and did not affect the outcome regarding which approach performed best.

Figure 9 provides an illustration of the ML-SL difference for each individual instance. It shows the values in the last two columns of Tables 5 and 6 and displays how the ML-SL difference changed for all instances.

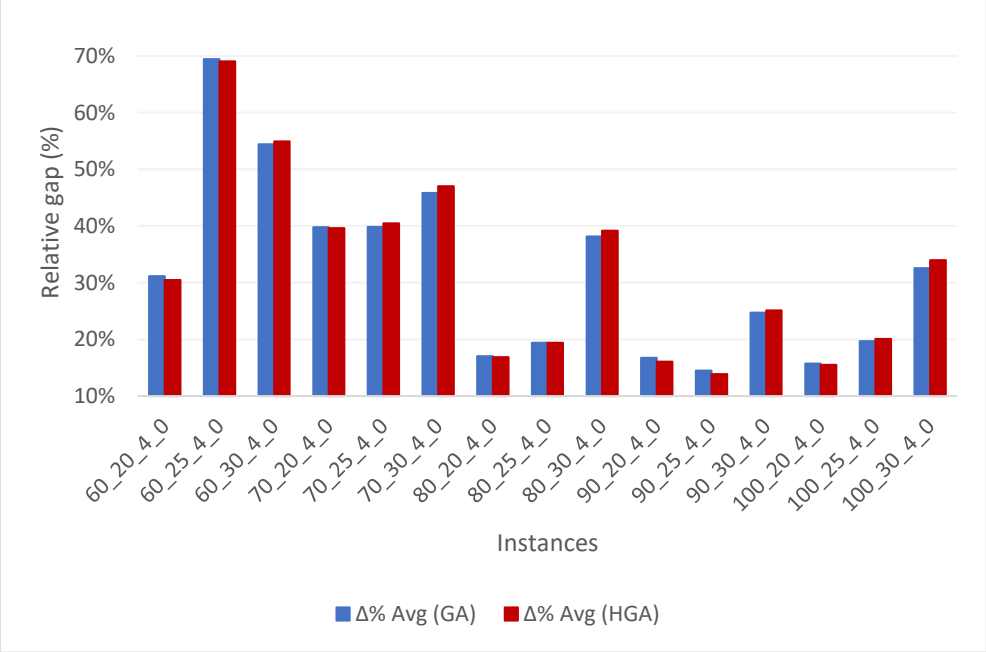

Figure 15: Relative makespan gap of ML to SL across large instances.

The data indicate a similar relative gap (Δ%) for the average results from both the ML and SL algorithms on all large instances as shown by the two nearly identical bars. Both are almost identical, showing that the representation is mainly driving the difference for these instances.

Adding local search changes individual results but not the pattern. When further analyzing this graph, two trends crystallize:

- With the same number of jobs, a higher number of stages increases the ML-SL gap.
- With the same number of stages, a higher number of jobs decreases the ML-SL gap.

In summary, the ML version performs worst in terms of the gap from SL when there are many stages to be coordinated and when the system is lightly loaded with jobs. Under these conditions, restricting the search to the fixed machine and worker choices in the chromosome appears to increase the gap from SL. However, as the number of jobs in the system increases, the ML-SL gap decreases as the algorithm has more opportunity to execute its search.

6.3 Summary of Results

Across all the tests, representation was the most important factor. In terms of quality of the best solution found, using a single level representation worked best in all test cases. When tested with an optimal solution, SL-HGA had an average gap of 0.9% and SL-GA had an average gap of 3.2%. As can be expected with faster evaluation times for the multi-level representations, this representation produced the worst makespans in all of the test cases. These results carried forward into the larger test case set. The multi-level representation performed on average 31% worse than the single-level representation for both the GA and HGA under the same runtime conditions. While local search does help improve the performance of the algorithms, its effectiveness is dependent upon the test case conditions. On the small test cases, the addition of local search improved the solution quality for the single-level representation from 3.2% to 0.9% on average. However, in the fixed-time test cases, the addition of local search resulted in very small absolute improvements. In addition to demonstrating the effectiveness of local search as a means to improve the performance of the algorithms, the use of random crossover instead of crossover in the multi-level representation demonstrates that the operator used within the representation will affect performance. In the same representation, replacing crossover with random crossover resulted in decreased solution quality, and did so at the same runtime as crossover. The decrease in solution quality caused by the use of random crossover in place of crossover is less than the effects of either the type of representation or whether or not local search was included.

However, it does indicate that if more refined or alternative operators are developed specifically to take advantage of the properties of the multi-level representation, they may result in improved performance of the multi-level representation.

These findings demonstrate how problem structure impacts the gap between models. As you increase the number of stages for a given number of jobs, you will increase the depth of coordination by means of re-entrance, thus widening the ML-SL gap. Conversely, as you increase the number of jobs for a given number of stages, you will decrease the ML-SL gap, implying that ML will benefit more from an increased number of search opportunities when the system is more heavily loaded. A possible explanation is that constraining machine and worker choices in the chromosome limits the ability of the ML to search when the routing depth is high. However, when there are many jobs, the algorithm may still be able to make headway via sequencing.

In practice, if schedule quality is your main concern and you are willing to accept moderate computation time, then SL-HGA would be your best choice with SL-GA being a reasonable second. However, if quick evaluation time is important, then the ML versions will provide faster turnaround time, but at the expense of a consistent quality gap.

7. Conclusion and Future Research

The Dual-Resource-Constrained Re-entrant Flexible Flow Shop Problem (DRCRFFSP) combines standard flow shops with re-entrant routing and labor constraints, making it difficult to obtain fast, high-quality schedules for industrial settings such as screen printing and similar multi-stage manufacturing systems. Building on an existing single-level Hybrid Genetic Algorithm (HGA) implementation in C++ by Mlekusch & Hartl (2025), the goal of this thesis was to investigate how different genetic algorithm representations compare to each other in this setting. The work was motivated by the observation of a clear gap in the literature regarding comparisons between single-level GAs (SL-GA), multi-level GAs (ML-GA), and their hybrid versions.

To address this gap, a multi-level solution representation was designed that explicitly encodes operation sequence, machine assignment and worker assignment within separate chromosomes. Within a common GA/HGA framework, the single-level (SL) and multi-level (ML) variants were then systematically compared on small instances with CP benchmarks and on larger instances without exact optima. Finally, the study aimed to identify whether the main drivers of performance differences can be attributed to the choice of representation (SL vs. ML), hybridization (GA vs. HGA) or operator design (standard vs. random crossover).

7.1 Main Findings

The thesis has shown that performance mainly depends upon the choice of the solution representation across all the experiments. The multi-level versions perform significantly faster in terms of evaluations, allowing for a greater number of generations to be processed within the same time frame than their single-level counterparts, although the single-level versions generally have an advantage when it comes to producing solutions of higher quality. This trade-off was observed both in smaller problem instances that are solvable via constraint programming (CP) as well as in larger, more realistic instances for which there were no exact reference values.

Hybridization through local search further improves the results, in particular for the single-level representation, but its effect depends on instance size and time limits. The experiments also indicate that problem structure matters. Instances with many stages and re-entrant visits

tend to favour the single-level encoding, whereas instances with many jobs offer more opportunities for the faster multi-level variants. Finally, the comparison with a random-crossover ML-GA confirms that, within a fixed representation, the design of genetic operators also influences performance and thus constitutes a promising direction for further improving multi-level approaches.

Ultimately, the main practical implication from this work is that, for the current configuration, single level variants will continue to be the most desirable if one's top priority is to produce high-quality schedules. If acceptable computational times are needed, then SL-HGA will be the preferred option, and as an even more efficient version, SL-GA would be suitable. As well as providing faster convergence, multi-level encoding methods have the benefit of being capable of producing good-quality solutions quickly. However, their advantage may be offset by the persistent quality gap between these and the best solution produced by the single-level variant. In the end, it was found that multi-level encoding methods can only take full advantage of their potential if they are used in combination with carefully designed operators.

7.2 Limitations

Conclusions drawn from this study were based on just one specific multi-level encoding along with a unique collection of operators and parameters. Another multi-level design, a different neighborhood in the local search space, or additional, even more comprehensive parameter optimization may alter the tradeoff between solution quality and run time for the problem. Furthermore, the experiments relied upon one class of test case families that contained deterministically specified service times as well as a single main objective (makespan). These results therefore cannot be generalized to other settings with differing routing configurations, greater variability in service times or multiple, potentially conflicting objectives.

7.3 Future Research

The results of this thesis suggest several directions for further work on multi-level genetic algorithms for the DRCRFFSP. A natural next step is to refine the multi-level genetic operators. This includes designing alternative crossover and mutation schemes that are better suited to multi-level chromosomes. Additional operators aimed at maintaining diversity, such as a

function that creates a new population after a certain number of stagnant generations, could help to counteract the early convergence observed in the ML variants.

Beyond operator design, future research could extend the setting in several ways. Multi-objective versions of the DRCRFFSP, including criteria such as energy consumption, worker utilization or tardiness, would better reflect real production trade-offs. Likewise, incorporating uncertainty (e.g., variable processing times or machine failures) would increase practical relevance. Finally, combining the metaheuristic framework with learning-based components or exact models, for instance, to generate initial solutions, tune parameters or evaluate resource patterns, offers further potential to improve performance and to deepen the understanding of when single-level or multi-level encodings are most appropriate.

References

Andrade-Pineda, J. L., Canca, D., Gonzalez-R, P. L., & Calle, M. (2020). Scheduling a dual-resource flexible job shop with makespan and due date-related criteria. *Annals of Operations Research*, *291*(1–2), 5–35. https://doi.org/10.1007/s10479-019-03196-0

Bertel, S., & Billaut, J.-C. (2004). A genetic algorithm for an industrial multiprocessor flow shop scheduling problem with recirculation. *European Journal of Operational Research*, *159*(3), 651–662. https://doi.org/10.1016/s0377-2217(03)00434-x

Bokhorst, J. A. C., Slomp, J., & Gaalman, G. J. C. (2004). On the who-rule in Dual Resource Constrained (DRC) manufacturing systems. *International Journal of Production Research*, *42*(23), 5049–5074. https://doi.org/10.1080/00207540410001733878

Chamnanlor, C., Sethanan, K., Chien, C.-F., & Gen, M. (2014). Re-entrant flow shop scheduling problem with time windows using hybrid genetic algorithm based on auto-tuning strategy. *International Journal of Production Research*, *52*(9), 2612–2629. https://doi.org/10.1080/00207543.2013.861949

Chen, C.-A., Kuo, H.-A., & Chien, C.-F. (2025). Dual-Resource Constrained Flexible Job Shop Scheduling for SMT Back-End Production and an Empirical Study of Wearable Devices. *IEEE Transactions on Automation Science and Engineering*, *22*, 9230–9239. https://doi.org/10.1109/TASE.2024.3503412

Chen, C.-L., Vempati, V. S., & Aljaber, N. (1995). An application of genetic algorithms for flow shop problems. *European Journal of Operational Research*, *80*(2), 389–396. https://doi.org/10.1016/0377-2217(93)e0228-p

Chen, J.-S., Pan, J. C.-H., & Lin, C.-M. (2008). A hybrid genetic algorithm for the re-entrant flow-shop scheduling problem. *Expert Systems with Applications*, *34*(1), 570–577. https://doi.org/10.1016/j.eswa.2006.09.021

Chen, J.-S., Pan, J. C.-H., & Wu, C.-K. (2007). Minimizing makespan in reentrant flow-shops using hybrid tabu search. *The International Journal of Advanced Manufacturing Technology*, *34*(3–4), 353–361. https://doi.org/10.1007/s00170-006-0607-2

Cheng, T. C. E., Gupta, J. N. D., & Wang, G. (2000). A REVIEW OF FLOWSHOP SCHEDULING RESEARCH WITH SETUP TIMES. *Production and Operations Management*, *9*(3), 262–282. https://doi.org/10.1111/j.1937-5956.2000.tb00137.x

Choi, S.-W., & Kim, Y.-D. (2008). Minimizing makespan on an -machine re-entrant flowshop. *Computers & Operations Research*, *35*(5), 1684–1696. https://doi.org/10.1016/j.cor.2006.09.028

Choi, S.-W., & Kim, Y.-D. (2009). Minimizing total tardiness on a two-machine re-entrant flowshop. *European Journal of Operational Research*, *199*(2), 375–384. https://doi.org/10.1016/j.ejor.2008.11.037

Chu, F., Liu, M., Liu, X., Chu, C., & Jiang, J. (2018). Reentrant Flow Shop Scheduling considering Multiresource Qualification Matching. *Scientific Programming*, *2018*, 1–8. https://doi.org/10.1155/2018/2615096

Daniels, R. L., Mazzola, J. B., & Shi, D. (2004). Flow Shop Scheduling with Partial Resource Flexibility. *Management Science*, *50*(5), 658–669. https://doi.org/10.1287/mnsc.1040.0209

Danping, L., & Lee, C. K. M. (2011). A review of the research methodology for the re-entrant scheduling problem. *International Journal of Production Research*, *49*(8), 2221–2242. https://doi.org/10.1080/00207541003720350

Dhiflaoui, M., Nouri, H. E., & Driss, O. B. (2018). Dual-Resource Constraints in Classical and Flexible Job Shop Problems: A State-of-the-Art Review. *Procedia Computer Science*, *126*, 1507–1515. https://doi.org/10.1016/j.procs.2018.08.123

Dong, J., & Ye, C. (2022). Green scheduling of distributed two-stage reentrant hybrid flow shop considering distributed energy resources and energy storage system. *Computers & Industrial Engineering*, *169*, 108146. https://doi.org/10.1016/j.cie.2022.108146

ElMaraghy, H., Patel, V., & Abdallah, I. B. (2000). Scheduling of manufacturing systems under dual-resource constraints using genetic algorithms. *Journal of Manufacturing Systems*, *19*(3), 186–201. https://doi.org/10.1016/s0278-6125(00)80011-4

Emmons, H., & Vairaktarakis, G. (2013). *Flow shop scheduling: Theoretical results, algorithms, and applications*. Springer Verlag.

Felan, J. T., & Fry, T. D. (2001). Multi-level heterogeneous worker flexibility in a Dual Resource Constrained (DRC) job-shop. *International Journal of Production Research*, *39*(14), 3041–3059. https://doi.org/10.1080/00207540110047702

Felan, J. T., Fry, T. D., & Philipoom, P. R. (1993). Labour flexibility and staffing levels in a dual-resource constrained job shop. *International Journal of Production Research*, *31*(10), 2487–2506. https://doi.org/10.1080/00207549308956871

Frye, J. S. (1974). Labor Flexibility in Multiechelon Dual-Constraint Job Shops. *Management Science*, *20*(7), 1073–1080. https://doi.org/10.1287/mnsc.20.7.1073

Garey, M. R., Johnson, D. S., & Sethi, R. (1976). The Complexity of Flowshop and Jobshop Scheduling. *Mathematics of Operations Research*, *1*(2), 117–129. https://doi.org/10.1287/moor.1.2.117

Gong, G., Deng, Q., Gong, X., Liu, W., & Ren, Q. (2018). A new double flexible job-shop scheduling problem integrating processing time, green production, and human factor indicators. *Journal of Cleaner Production*, *174*, 560–576. https://doi.org/10.1016/j.jclepro.2017.10.188

Graves, S. C., Meal, H. C., Stefek, D., & Zeghmi, A. H. (1983). Scheduling of re-entrant flow shops. *Journal of Operations Management*, *3*(4), 197–207. https://doi.org/10.1016/0272-6963(83)90004-9

Han, W., Deng, Q., Gong, G., Zhang, L., & Luo, Q. (2021). Multi-objective evolutionary algorithms with heuristic decoding for hybrid flow shop scheduling problem with worker constraint. *Expert Systems with Applications*, *168*, 114282. https://doi.org/10.1016/j.eswa.2020.114282

Hottenstein, M. P., & Bowman, S. A. (1998). Cross-training and worker flexibility: A review of DRC system research. *The Journal of High Technology Management Research*, *9*(2), 157–174. https://doi.org/10.1016/s1047-8310(98)90002-5

Kher, H., & Malhotra, M. (1994). Acquiring and operationalizing worker flexibility in dual resource constrained job shops with worker transfer delays and learning losses. *Omega*, *22*(5), 521–533. https://doi.org/10.1016/0305-0483(94)90032-9

Kher, H. V., & Fry, T. D. (2001). Labour flexibility and assignment policies in a job shop having incommensurable objectives. *International Journal of Production Research*, *39*(11), 2295–2311. https://doi.org/10.1080/00207540110036704

Kress, D., & Müller, D. (2019). Mathematical Models for a Flexible Job Shop Scheduling Problem with Machine Operator Constraints. *IFAC-PapersOnLine*, *52*(13), 94–99. https://doi.org/10.1016/j.ifacol.2019.11.144

Ku, W.-Y., & Beck, J. C. (2016). Mixed Integer Programming models for job shop scheduling: A computational analysis. *Computers & Operations Research*, *73*, 165–173. https://doi.org/10.1016/j.cor.2016.04.006

Kumar, P. R. (1993). Re-entrant lines. *Queueing Systems*, *13*(1–3), 87–110. https://doi.org/10.1007/bf01158930

Lin, D., Lee, C. K. M., & Ho, W. (2013). Multi-level genetic algorithm for the resource-constrained re-entrant scheduling problem in the flow shop. *Engineering Applications of Artificial Intelligence*, *26*(4), 1282–1290. https://doi.org/10.1016/j.engappai.2012.10.006

Mehravaran, Y., & Logendran, R. (2013). Non-permutation flowshop scheduling with dual resources. *Expert Systems with Applications*, *40*(13), 5061–5076. https://doi.org/10.1016/j.eswa.2013.03.007

Mlekusch, J. (2024). *Instances for the Dual-Resource-Constrained Re-entrant Flexible Flow shop scheduling problem* [Dataset]. Mendeley Data. https://doi.org/10.17632/73HN9BXXVV.2

Mlekusch, J., & Hartl, R. F. (2022). Solution Methods for the Dual-Resource-Constrained Re-Entrant Flexible Flow Shop. *SSRN Electronic Journal*. https://doi.org/10.2139/ssrn.4120101

Mlekusch, J., & Hartl, R. F. (2025). The dual-resource-constrained re-entrant flexible flow shop a constraint programming approach and a hybrid genetic algorithm. *International Journal of Production Research*, *63*(5), 1803–1824. https://doi.org/10.1080/00207543.2024.2392198

Müller, D., & Kress, D. (2022). Filter-and-fan approaches for scheduling flexible job shops under workforce constraints. *International Journal of Production Research*, *60*(15), 4743–4765. https://doi.org/10.1080/00207543.2021.1937745

Murata, T., Ishibuchi, H., & Tanaka, H. (1996). Genetic algorithms for flowshop scheduling problems. *Computers & Industrial Engineering*, *30*(4), 1061–1071. https://doi.org/10.1016/0360-8352(96)00053-8

Nawaz, M., Enscore, E. E., & Ham, I. (1983). A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *Omega*, *11*(1), 91–95. https://doi.org/10.1016/0305-0483(83)90088-9

Nelson, R. T. (1967). Labor and Machine Limited Production Systems. *Management Science*, *13*(9), 648–671. https://doi.org/10.1287/mnsc.13.9.648

Pan, J.-H., & Chen, J.-S. (2003). Minimizing makespan in re-entrant permutation flow-shops. *Journal of the Operational Research Society*, *54*(6), 642–653. https://doi.org/10.1057/palgrave.jors.2601556

Park, P. S. (1991). The examination of worker cross-training in a dual resource constrained job shop. *European Journal of Operational Research*, *52*(3), 291–299. https://doi.org/10.1016/0377-2217(91)90164-q

Pinedo, M. L. (2016). *Scheduling*. Springer International Publishing. https://doi.org/10.1007/978-3-319-26580-3

Reeves, C. R. (1995). A genetic algorithm for flowshop sequencing. *Computers & Operations Research*, *22*(1), 5–13. https://doi.org/10.1016/0305-0548(93)E0014-K

Rifai, A. P., Nguyen, H.-T., & Dawal, S. Z. M. (2016). Multi-objective adaptive large neighborhood search for distributed reentrant permutation flow shop scheduling. *Applied Soft Computing*, *40*, 42–57. https://doi.org/10.1016/j.asoc.2015.11.034

Ruiz-Torres, A. J., Paletta, G., & Adenso-Díaz, B. (2022). Hybrid two stage flowshop scheduling with secondary resources based on time buckets. *International Journal of Production Research*, *60*(6), 1954–1972. https://doi.org/10.1080/00207543.2021.1880656

Sangsawang, C., Sethanan, K., Fujimoto, T., & Gen, M. (2015). Metaheuristics optimization approaches for two-stage reentrant flexible flow shop with blocking constraint. *Expert Systems with Applications*, *42*(5), 2395–2410. https://doi.org/10.1016/j.eswa.2014.10.043

Sun, J. U. (2009). A genetic algorithm for a re-entrant job-shop scheduling problem with sequence-dependent setup times. *Engineering Optimization*, *41*(6), 505–520. https://doi.org/10.1080/03052150802613335

Topcuoglu, H. R., Demiroz, B., & Kandemir, M. (2007). Solving the Register Allocation Problem for Embedded Systems Using a Hybrid Evolutionary Algorithm. *IEEE Transactions on Evolutionary Computation*, *11*(5), 620–634. https://doi.org/10.1109/TEVC.2007.892766

Treleven, M. (1989). A Review of the Dual Resource Constrained System Research. *IIE Transactions*, *21*(3), 279–287. https://doi.org/10.1080/07408178908966233

Vargas-Villamil, F. D., Rivera, D. E., & Kempf, K. G. (2003). A hierarchical approach to production control of reentrant semiconductor manufacturing lines. *IEEE Transactions on Control Systems Technology*, *11*(4), 578–587. https://doi.org/10.1109/tcst.2003.813368

Wang, X., Liu, C., Wang, R., Yu, Z., & Yang, S. (2025). Improved Genetic Algorithm Using Reinforcement Learning to Solve the Re-entrant Flexible Flow Shop Scheduling Problem. *2025 IEEE Congress on Evolutionary Computation (CEC)*, 1–8. https://doi.org/10.1109/cec65147.2025.11043111

Whitley, D. (1994). A genetic algorithm tutorial. *Statistics and Computing*, *4*(2). https://doi.org/10.1007/BF00175354

Yang, D.-L., Kuo, W.-H., & Chern, M.-S. (2008). Multi-family scheduling in a two-machine reentrant flow shop with setups. *European Journal of Operational Research*, *187*(3), 1160–1170. https://doi.org/10.1016/j.ejor.2006.06.065

Ying, K.-C., Lin, S.-W., & Wan, S.-Y. (2014). Bi-objective reentrant hybrid flowshop scheduling: An iterated Pareto greedy algorithm. *International Journal of Production Research*, *52*(19), 5735–5747. https://doi.org/10.1080/00207543.2014.910627