Flexible Job-Shop Scheduling under heavy constraints using AI-based Algorithms

van Straaten, Kjell

Award date:
2023

Department of Industrial Engineering & Innovation Sciences
Department of Mathematics & Computer Science

Flexible Job-Shop Scheduling under heavy constraints using AI-based Algorithms.

In collaboration with

Author:

Kjell van Straaten (1252712)

Supervision:

Dr. Yingqian Zhang, Eindhoven University of Technology

Dr. Meng Fang, Eindhoven University of Technology

Tobias Scheepers, WeFabricate Best

Eindhoven, June 2023

Abstract

In this research, we explore different types of flexible job-shop scheduling problems (FJSP) and compare the performance of two advanced algorithms. Our goal is to determine which algorithm is better suited for scheduling automation in large-scale manufacturing processes. Through a series of experiments, we have concluded that our proposed self-learning effective genetic algorithm (SLEGA) outperforms the end-to-end deep reinforcement learning (E2E-DRL) approach in most scenarios.

The SLEGA combines traditional genetic algorithms with a self-learning module that can automatically adjust important parameters during the optimization process. On the other hand, the E2E-DRL approach treats scheduling as a graph problem, where a reinforcement learning agent selects which operations should be performed on which machines based on graph representations.

We evaluated both algorithms using various job scheduling problems, including standard FJSP, FJSP with sequence-dependent setup times, and a highly-constrained FJSP scenario inspired by real-world manufacturing situations. This scenario involved additional factors such as release dates, order deadlines, night times, and sequence-dependent setup times between operations.

Our findings indicate that the SLEGA algorithm is more adaptable to different types of scheduling problems and performs better overall compared to the E2E-DRL approach. Additionally, the SLEGA algorithm can be parallelized, which means that it can handle computations more efficiently and scale well with larger problem sizes. E2E-DRL is more usable in scenarios where scheduling decisions are required to be available in real-time (i.e., online scheduling).

In conclusion, our research provides valuable insights into the automation of scheduling in manufacturing. The SLEGA algorithm shows great potential for optimizing job-shop scheduling and improving manufacturing efficiency.

Executive Summary

This summary serves as a brief overview of the executed research project. It describes the research problem at hand, the research approach taken to tackle the problem, the identified solution methods, the results obtained with these solution methods, and the conclusions and recommendations.

Research Problem

We are collaborating with Wefabricate on a research project aimed at addressing machine scheduling problems in the milling process of their factory. Specifically, we are focusing on the highly constrained flexible job-shop scheduling problem (FJSP), with the goal of selecting the optimal operation and machine pair at each point in time to minimize the total makespan of the schedule and minimize associated costs. In literature, the vanilla FJSP with minimal additional characteristics is usually investigated. Furthermore, problem instance sizes usually do not scale up to industry scenarios.

The environment under which this problem arises is the milling process of Wefabricate. The raw material that arrives at the factory is first sawed, then milled, cleaned inspected and finally packaged. The milling process itself is executed in one or more stages. Each stage can be executed by different machines and could require an operator for setup and execution, which are not available during the night. For each stage, several different resources are required. Resources required are both consumed (i.e., raw material) or renewed (i.e., machining tools). Raw material stock is not infinite, and thus jobs are released at a certain point in time. The customer also prescribes a product quantity and delivery deadline as well. Furthermore, the setup between different operations is sequence-dependent, furthermore increasing the complexity of finding an efficient schedule. Wefabricate desires an algorithm that could compute these schedules automatically.

The performance of this schedule is generally described using the makespan. The makespan is defined as the moment at which the last operation is finished. Besides makespan, we also optimize for the cost incurred by the schedule. The total cost consists of six different components: (1) missing deadlines, (2) finished good inventory, (3) work-in-progress inventory, (4) logistic movements, (5) manual operations and (6) addition of resources. Finding the Pareto front along these two objectives should enable Wefabricate to select highly efficient and effective schedules.

Research Approach

This thesis aims to create a flexible planning algorithm to solve the FJSP. The algorithm should be generally applicable, to current manufacturing processes and future manufacturing processes. Besides solution quality, scalability and generalizability play an important role for Wefabricate, as the set of input jobs to schedule will change over the coming years. In the literature, we have already seen various reinforcement learning-based job scheduling algorithms. More specifically, we have seen a state-of-the-art end-to-end deep reinforcement learning (E2E-DRL) approach based on graph neural networks, and a self-learning genetic algorithm (SLGA) approach in order to tackle the vanilla FJSP. However, the literature does not test these solutions on industry problems such as the highly constrained FJSP that Wefabricate is dealing with. This insight led to the formulation of the following main research question, which is central to this master thesis:

How can we integrate machine learning into job-scheduling algorithms for a highly constrained FJSP?

In order to explore the different approaches for solving the vanilla FJSP problem, we conduct an in-depth analysis by proposing a self-learning effective genetic algorithm (SLEGA) and by benchmarking this against the E2E-DRL approach. Our methodology involves implementing these algorithms and training them on various instance sizes. We evaluate the trained models using several benchmark literature

datasets, including *mkdata*, *edata*, *rdata*, *vdata* and *ftdata*. Additionally, we test the models on a custom FJSP benchmark dataset that spans job and machine sizes from 5x5 to 100x100. This allows us to compare the performance of the two approaches and identify their strengths and weaknesses on the vanilla FJSP. To further examine the algorithms' capabilities, we introduce a variant of the FJSP problem that includes sequence-dependent setup times. This enables us to assess how each model performs when faced with a new problem characteristic. Moving on to a real-world application, we consider the highly constrained FJSP of Wefabricate and compare the performance of E2E-DRL and the implemented SLEGA. Finally, we explore the flexibility and performance of the SLEGA in a multi-objective setting. Overall, our analysis provides a comprehensive understanding of the SLEGA and E2E-DRL approaches in the context of the FJSP problem and sheds light on their suitability for real-world applications.

Solution Methods

To compare our two solution methods, we developed a single schedule evaluation function that returns the two main objectives of interest for Wefabricate: the makespan and the cost of the schedule. We use this function to evaluate both the highly-constrained instances of Wefabricate and the vanilla FJSP instances from the literature. The function takes as input a schedule representation and a problem instance. A schedule is represented using the double-layer encoding technique, consisting of an operation sequence string and a machine allocation string. Using the problem instance, we decode the schedule representation into a feasible schedule that takes into account various constraints, including release dates, precedence constraints, setup times, night times, and backfilling. Backfilling allows for operations to be scheduled in gaps between already scheduled operations. We designed the evaluation function to be executed in parallel, enabling fast evaluations. This allows us to properly compare the performance of our two algorithms on both types of instances.

The SLEGA implementation consists of two components, the underlying genetic algorithm, and a self-learning module to adjust the hyperparameters of the genetic algorithm during execution. The genetic algorithm works in the following way. Individuals are represented following the double-layer encoding approach mentioned above. Individuals are evaluated following the evaluation function defined in the previous paragraph. For initialization, the population size is set to 100 and initialized using global selection (60%), local selection (30%) and random selection (10%). The evolutionary algorithm is then generally executed for 100 generations, with the following genetic operations. For selection, we use tournament selection (k=3) when considering the single-objective setting or NSGA-II in the multi-objective setting. For crossover, we use a 2-point crossover (p=0.5) and uniform crossover (p=0.5) for the machine allocation string, and precedence preserving crossover for the operation sequence string. For mutation, we use greedy mutation for the machine allocation string, and swap mutation for the operation sequence string.

This genetic algorithm is then wrapped in a DRL environment to create the self-learning module. In every generation, a PPO agent is fed the following state: (1) average normalized mean fitness of objectives of the current population, (2) average normalized best fitness of objectives of the current population, (3) average normalized standard deviation of fitness of the current population, (4) normalized remaining budget, (5) normalized stagnation count, (6) normalized hypervolume indicator and (7) normalized Pareto size. Note that (6) and (7) are only emitted by the environment for the multi-objective setting. This state space is then embedded with an MLP policy. From this embedding, the PPO agent then picks the crossover probability, individual mutation probability and gene mutation probability. The next iteration of the genetic algorithm is then executed. The agent then receives a reward or penalty for each objective value. For the single-objective setting, we use absolute increase/decrease as the reward function. While for the multi-objective setting, we use binary increase/decrease as a reward function. Overall, this approach allows the genetic algorithm to self-learn and improve over time by incorporating feedback from the PPO agent.

In the E2E-DRL approach, we formulate an FJSP instance as a heterogeneous graph. This graph consists of operation and machine nodes. Bi-directional arcs represent operation-machine pairs, while directional arcs represent the order of the jobs. The graph represents the state space of an environment and contains different features. Operations nodes contain (1) scheduling status, (2) the number of neighbouring machines, (3) processing time, (4) the number of unscheduled operations in the job, (5) start time, (6) setup time, (7) night schedule flag, (8) night setup flag, (9) deadline flag and (10) time to release. The machine nodes contain (1) the number of neighbouring operations, (2) available time, (3) utilization, (4) time of scheduling, (5) remaining time until night. O-M pairs contain (1) processing time, (2) sequence-dependent setup time (3) time running at night. This graph is then embedded using a two-stage embedding scheme. First, the machine nodes are embedded using several graph attention networks (GATs). GATs are selected as they automatically learn the importance of different nodes by applying the attention mechanism. Next, operation nodes are embedded using several MLPs. After stacking and pooling, the critic network computes the value of the state with an additional MLP, whereas the actor network computes the probability of selecting a certain action. This probability distribution can then be used to greedily select actions or selection actions on a sampling basis. The sampling approach is usually deployed, but for testing purposes, results of greedily selecting actions are reported too.

Both the SLEGA and E2E-DRL implementations are trained using a sampling-based actor-critic approach. The PPO loss is considered as the loss function during training. The trained policies are evaluated every x timesteps on a validation dataset. The model with the highest validation score is then saved and used during testing.

Results

In Experiment One, we have seen that in the vanilla FJSP, the instance size is of importance when selecting a scheduling approach according to its performance (i.e., makespan). More specifically, we saw that E2E-DRL outperformed SLEGA for instances which have relatively few good solutions and a larger search space. In those cases, the makespan was reduced by up to 30%. For instances where the number of good solutions was relatively higher, the SLEGA outperformed E2E-DRL and reduced makespan up to 20%. In terms of execution times, E2E-DRL was faster by 20 to 90%, where execution time gain was less for larger instances. Hence, for the vanilla FJSP problem, we consider both algorithms to be scalable and efficient, where the E2E-DRL approach is more efficient than the SLEGA approach.

In Experiment Two, we compared the performance of two approaches for solving the FJSP with SDSTs. Our results showed that the SLEGA approach outperformed E2E-DRL in terms of makespan, with a 5% improvement. Moreover, we found that retraining improved the performance of E2E-DRL, while the SLEGA approach demonstrated better generalization to unseen characteristics, making it more flexible. In terms of inference time, E2E-DRL was significantly faster, although the execution time of the SLEGA approach was still within acceptable limits. Overall, our findings suggest that the SLEGA approach is a more reliable and flexible option for solving the FJSP with SDSTs, whereas E2E-DRL is a faster but less adaptable approach.

In Experiment Three, we investigated the performance of two approaches for solving the highly constrained FJSP of Wefabricate. Our results demonstrated that the SLEGA approach outperformed E2E-DRL in terms of makespan, even when considering new and previously unseen instance sizes. This suggests that the SLEGA approach is more flexible and robust to changes in problem characteristics. Moreover, we found that the performance of the SLEGA approach could be further improved by increasing the population size and the number of generations, although this trade-off with efficiency should be considered. Nevertheless, our results showed that the SLEGA approach was able to outperform traditional heuristics while still achieving acceptable execution times. Overall, these findings indicate that the SLEGA approach is a reliable and effective option for solving the highly constrained FJSP of Wefabricate.

In Experiment Four, we tested the performance of the SLEGA approach on the highly-constrained FJSP in a multi-objective setting. Our results demonstrated that the SLEGA approach was able to handle the multi-objective nature of the problem. E2E-DRL in the multi-objective context was not attempted, due to the fact that there is no E2E-DRL approach in literature for this problem. These findings further emphasize the flexibility of the SLEGA scheduling approach. Our experiments also revealed that incorporating a self-learning module into the SLEGA approach improved its performance when compared to a vanilla genetic algorithm. Moreover, the algorithm remained efficient enough to be used in production. Overall, these results suggest that the SLEGA approach is a robust and adaptable option for handling the highly-constrained FJSP in a multi-objective setting, and its performance can be further enhanced through the incorporation of self-learning modules.

Conclusion and Recommendations

Based on our findings in the previous paragraphs, we recommend that Wefabricate adopt the SLEGA algorithm as a standard scheduling approach for various operation scheduling problems when integrating machine learning into job scheduling. This is because the SLEGA approach outperforms E2E-DRL in terms of flexibility, performance, and generalization. Specifically, the SLEGA approach can handle various constraints and objectives more effectively, achieve lower makespan and cost, and require less retraining than E2E-DRL. Additionally, the SLEGA approach is efficient enough to achieve acceptable execution times, is highly scalable through parallelization, and can adapt to dynamic events during schedule execution. The evaluation function can be controlled to adjust the preferences of the company, such as buffers and working hour capacity, and the SLEGA approach can learn the setting of hyperparameters based on the given evaluation function. Overall, the SLEGA approach is a generic and reliable option for machine learning-based job scheduling at Wefabricate. This answers our main research question.

With the SLEGA scheduling approach, onboarding a new process only requires creating an evaluation function and potentially genetic operations. The algorithm remains the same, unlike the E2E-DRL approach, which would require significant effort. Additionally, the SLEGA approach can be applied to various optimization problems as long as a solution can be represented as an individual and this individual can be evaluated given the business context.

Potential areas for future research on the E2E-DRL approach include investigating its graph formulation and modelling setup times in different ways, such as by adding them to processing time. Additional features could also be added to improve its performance, and hyperparameter tuning should be explored to improve the training procedure and objective values. Warm starts could be investigated to optimize existing schedules in case of dynamic events. Furthermore, the E2E-DRL approach could be used for multi-objective optimization, which would be an interesting area to explore.

Regarding Wefabricate, we recommend two specific actions. Firstly, fine-tuning the evaluation function is crucial to getting accurate schedules, and it should be of the highest quality possible. Secondly, we suggest adopting the SLEGA approach for scheduling, which is a generic approach that can be easily extended to new production processes and other business problems, provided that a solution representation and evaluation function can be created. By splitting jobs into multiple ones and spreading out the quantity, optimization becomes more flexible and can lead to better results.

Preface

I am excited to present this thesis, the culmination of my dual master's degree in Operations Management and Logistics and Data Science in Engineering at the Eindhoven University of Technology. Over the past eight months, I have been immersed in the world of job scheduling optimization at the industry scale, working in collaboration with Wefabricate.

I cannot express enough gratitude to everyone who has supported me on this journey. In particular, I owe a debt of thanks to Yingqian Zhang, who mentored me during my master's program and served as my supervisor during this thesis. Without your guidance and support, I would not have been able to achieve this result. I would also like to express my gratitude to my second supervisor, Meng Fang, for entrusting me with the freedom to execute my work as I saw fit.

I also want to thank Tobias Scheepers and Jeroen Raaijmakers from Wefabricate, whose expertise and dedication were invaluable in tackling the challenging topic at hand. Your contributions were instrumental in the success of this work.

Finally, I want to express my appreciation to Robbert Reijnen, whose insights and feedback were a valuable sounding board throughout my research. Your willingness to engage with high- and low-level problems made you a perfect sparring partner.

Kjell van Straaten
Eindhoven, June 2023

Contents

Acronyms

BB	branch and bound
CAD	computer-aided design
CO	combinatorial optimization
CP	constraint programming
DRL	deep-reinforcement learning
E2E	end-to-end
EDF	earliest deadline first
FFSP	flexible flow-shop scheduling problem
FIFO	first-in first-out
FJSP	flexible job-shop scheduling problem
GS	global selection
HG	heterogeneous graph
HGNN	heterogeneous graph neural network
IP	integer programming
JSP	job-shop scheduling problem
LS	local selection
MDP	Markov decision process
MILP	mixed-integer linear programming
MIP	mixed integer programming
MLP	multi-layer perceptron
MOO	multi-objective optimization
MOR	most operations remaining
MTO	make-to-order
MWKR	most work remaining
PPO	proximal policy optimization
RS	random selection
SDST	sequence-dependent setup-times
SLEGA	self-learning effective genetic algorithm
SLGA	self-learning genetic algorithm
SOO	single-objective optimization
SPT	shortest processing time
TRPO	trust region policy optimization
TWK	total work content
WF	Wefabricate
WIP	work-in-progress

List of Figures

List of Tables

Chapter 1

Introduction

This Chapter introduces the business context of the company and introduces the problem by providing an overview of the problem characteristics at play. Then, the research is introduced by formulating the research questions and research approach.

1.1 Problem Context

Wefabricate (WF) is a startup company in the manufacturing industry focused on value-added manufacturing, products (parts or final products) that leave the production lines are worth more than the value that was put into the product throughout the manufacturing process. Wefabricate currently has four locations, three locations near Eindhoven (Best, Duizel, and Budel) and one more location in China. Since the company was founded about three years ago, Wefabricate grew a lot through the customers they have been able to add to their sales funnel. Wefabricate has significantly large customers for parts and finished products across different industries. Examples of industries customers operate in are healthcare, automotive, and semiconductors.

The added value manufacturing materializes through the different production cells displayed in Figure 1.1. Examples of production cells are injection moulding, machining, and assembly. Outside of the added-value manufacturing, WF also sells a couple of in-house products. Fyllar, for example, is a subsidiary company focused on creating refilling solutions for customers like supermarkets. Finally, to support the goal of Wefabricate, WeAutomate was founded and focused on the automation of all operations.

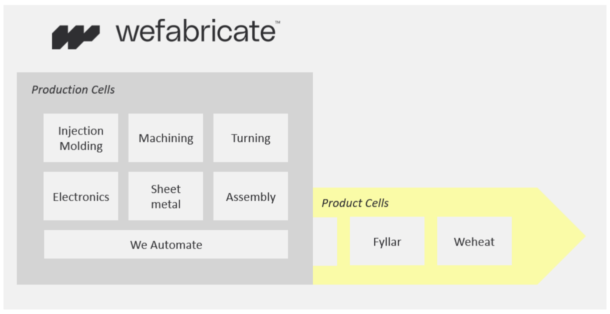


Figure 1.1: Business setup of Wefabricate.

1.1.1 Automation

WeAutomate is a very important department as it is a major goal of Wefabricate to automate all of the operations within the factory, which would enable the so-called “lights out factory”, where there are no operators required in order to fulfil customer demands. In the long term, the entire factory will be automated such that customers are only required to upload a design of a product/part to the website. Then, the customers will receive pricing and design for manufacturing feedback. If customers accept the quotation, the rest of the operations within the factory are kick-started, and the product is produced without human touch (from input raw material to packaged product).

Full automation will enable customization for production orders with a quantity of a single item. This would make ordering a fully custom product as easy as the purchase of a new device or book from e-commerce websites. An example use case is the purchase of a new bicycle. Right now, customers can select a predefined frame size with standard conventional sizes (e.g., S, M, or L). Through automation, the customer would be prompted to fill in their length per body part, after which the perfect frame size would be calculated and produced for the customer. When this make-to-order (MTO) manufacturing setting is fully automated, the cost and speed would not differ from traditional make-to-stock manufacturing.

MTO combined with decentralized manufacturing (i.e., producing close to the customer) would disrupt the manufacturing market. First of all, products are only created as required by customers, no resources are wasted on products which never seen the light of day. Furthermore, raw materials are the only items that are required to be shipped to and from the factory. This required significantly less capacity than shipping boxes of final products. This again can only be realized when the factory is completely automated.

1.1.2 Machining

Products that are machined generally follow the process illustrated in Figure 1.2. Input raw material is sawed such that we remain with a “blank” that has dimensions the milling machine can work with. The blank is then milled until the required features are achieved. Then, products have to be cleaned as various substances (e.g., oil) is used for cooling in the milling process. Next, the quality of features is checked (i.e. ensuring specified tolerances are met) and products are packed and now considered finished products. This marks the last step of the end-to-end (E2E) machining process. In case machined parts are required for assembly, they would not have been packaged but transferred to the assembly process instead.

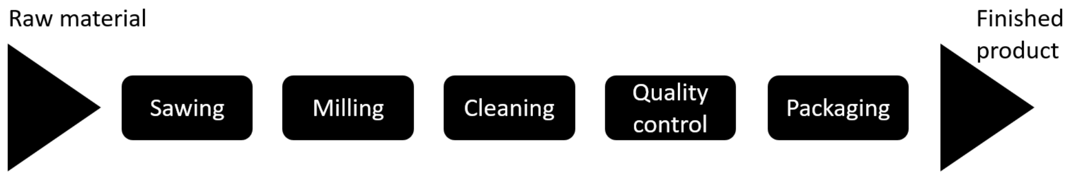


Figure 1.2: E2E machining process

In this research we will mainly address the milling operation of the machining process, hence this process is further illustrated below. As can be seen in Figure 1.3 below, a block of raw material is held by a pair of clamps. Then, a milling tool moves across the piece of raw material. This is repeated at several different angles using various tools until the desired shape and features of the required part are achieved. Since this is done using robotics, the precision is extremely high, and the features of the parts are very accurate. The type of raw material can be different depending on what is required by the customer (e.g., copper, steel, or aluminium). Within each milling operation, a product requires one or multiple set-ups of a machine in order to arrive at the required output part. This process is illustrated in Figure 1.4.

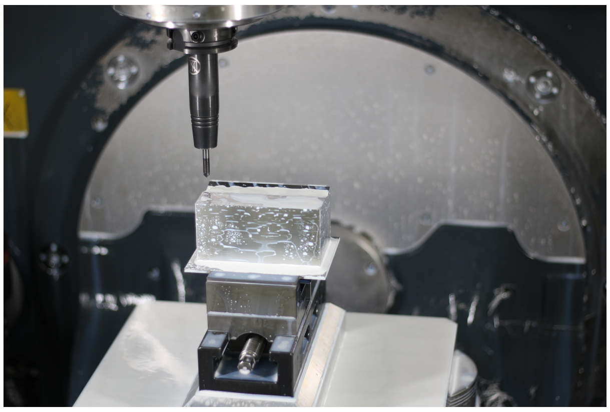
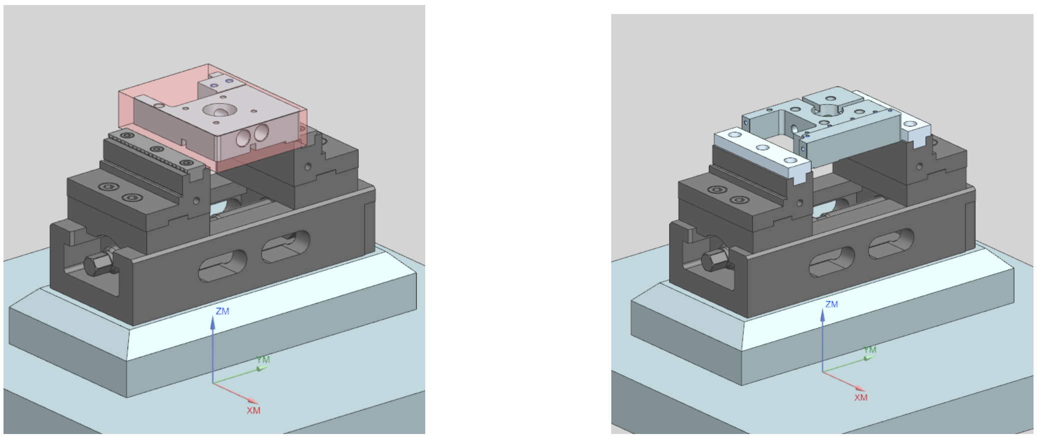


Figure 1.3: Close-up of raw material in a milling machine.

There are over 20.000 different parts for which Wefabricate has sets of machining instructions available. Furthermore, machining instructions can be generated automatically using the computer-aided design

(CAD) of a part. Hence, the company can technically produce an infinite amount of different parts. Each milling operation has its own required production time, tools, pallet configuration, fixtures, tops, and type of input raw material. Furthermore, there are currently multiple different machines that can handle the execution of these milling operations. However, the production times for each step can differ per machine. Furthermore, some operations require an operator to pay close attention to the production process (and interrupt where necessary). Finally, the setup time between different milling operations is sequence-dependent, since the raw material, pallet, fixture, clamp tops, and required tools could have to be changed depending on what was milled previously.



(a) Setup 1 (b) Setup 2

Figure 1.4: Illustration of different stages of the milling process.

1.1.3 Scheduling

Following the setting outlined in the paragraph above, it is not hard to see that the scheduling of the jobs on the different machines isn't particularly straightforward. Especially since each customer order also has additional requirements such as a deadline and order quantity. The order quantity can vary significantly, whereas series jobs could have more than a hundred units, and prototype or one-off jobs only would have a quantity of one. While production scheduling is difficult, it is key for manufacturing systems to improve efficiency and utilization rate of resources (R. Chen et al., 2020).

Besides the job-specific characteristics described above, the schedule is also influenced by the environment. For example, express jobs could arrive which have to be prioritized or a machine could break down. While all of these characteristics are at play, Wefabricate would like to find the schedule that optimizes for the makespan and the total incurred cost associated with the schedule. The total cost is made up of multiple factors, such as missing deadlines, inventory cost, required physical labour, and more. These are further described in Chapter 2.

While a (near) optimality of the scheduling algorithm is important, the algorithm should also be flexible, robust and efficient. The flexibility of the scheduling algorithm would allow Wefabricate to extend to the algorithm by including different production cells such as sawing or cleaning. The robustness of the scheduling algorithm would allow a near-optimal solution to be found if the set of input job drastically changes. Efficiency, or fast computation, would mean that in case of any dynamic events (like an operator that doesn't follow the planning), a new optimal schedule could be computed relatively fast. Furthermore, creating a scheduling algorithm would also enable full factory automation, the most important goal of Wefabricate. Right now, the schedule is created by hand using the earliest deadline first (EDF) heuristic. This indicates that there is a huge amount of improvements to be made in the scheduling field.

1.2 Research Questions

The scheduling problem outlined in the previous section is generally described as flexible job-shop scheduling problem (FJSP) in literature. The FJSP is usually decomposed into two sub-problems. A machine allocation problem, and a job sequencing problem. Literature shows that using AI-based optimization methods, such as E2E deep-reinforcement learning (DRL) and self-learning genetic algorithms (SLGAs), minimizes schedule computation time. Based on these findings, the main research question is defined as followed:

How can we integrate machine learning into job scheduling algorithms for a highly constrained FJSP?

To answer this question, we define the following sub-questions:

1. How can the scheduling problem at Wefabricate be formulated according to the literature?
2. What traditional- and learning-based approaches exist for the scheduling problem of Wefabricate?
3. How do the identified methods compare on different scheduling problems and in different scheduling scenarios?

Answering the sub-questions would allow us to formulate an answer to the main research question at hand.

1.2.1 Scope

As previously mentioned, the goal of this project is to generate an optimal schedule of jobs for the products created on the milling machines. These jobs consist of one or more stages. We plan to solve this problem by creating two AI-based solutions, an self-learning effective genetic algorithm (SLEGA) and an E2E-DRL framework. We propose these methods after the literature review and problem formulation are conducted. Furthermore, generating interesting datasets that could describe future scenarios for the company is also considered in scope.

In order to evaluate the performance of different methods, we conduct four different types of benchmarking. First, we benchmark the algorithms on non-company related existing benchmark datasets, in order to compare our performance against the performance of literature on standard datasets. Next, we benchmark our algorithms across these three different datasets against other existing (meta)-heuristics, in order to identify performance gain.

The first company-specific benchmark executed is on actual datasets of Wefabricate. Here, we derive performance differences between executed schedules versus proposed schedules by our algorithms, giving an idea of how optimal the current schedule is and what improvements can be made. Finally, we benchmark our implemented algorithms on different scenario datasets in order to measure the flexibility, performance, robustness and scalability of the algorithms.

For Wefabricate, the main goal is to integrate the best methods then in day-to-day operations in order to improve the quality of the existing job scheduling algorithm. This would enable their operations to run more efficiently, thus increasing performance and eventually profitability of operations.

What is considered out of scope, for now, is the following: The addition of other stages (cleaning, quality control, sawing) of the production process. The integration of logistic resources (e.g., schedule jobs such that they finish right as AGV is available), and job scheduling of other production cells of WF (e.g. injection moulding) is also out of scope.

1.2.2 Research Setup

To address the research gap identified in the previous section, we formulate the research setup as visualized in Figure 1.5 below. It consists of five sections: (1) Literature Review (2) Problem Formulation, (3) Methods (4) Data Generation, and (5) Results.

For each category, we define one or more research objectives. For the literature review, we define three topics that have to be investigated. The first objective (Q1) is to investigate existing methods in literature used to formulate and solve FJSP's. Next, we investigate how the FJSP can be approached in the context of multi-objective optimization (MOO) (Q2) and AI-based optimization (Q3). In parallel to the previous two objectives, we can define the FJSP of Wefabricate (Q4) using the results from Q1. Next, to solve the problem defined in Q4, we develop different frameworks. These frameworks are an SLEGA (Q5) and E2E-DRL (Q6). In parallel to all of this, we have to generate scenario datasets (Q7) to benchmark (Q8) the frameworks defined in Q5 and Q6 in terms of quality (flexibility, scalability, performance, robustness).

For Q7, it is important to acknowledge that the existing data available to Wefabricate does not encompass all potential scenarios that may arise in the future. Hence it is important to generate datasets that would cover various edge cases. Example edge cases are situations where the list of jobs only consists of series or only consists of prototype jobs. The optimal execution of operations is very different in that case. Hence it is interesting to identify the performance of the created framework under different scenarios, to evaluate the robustness and flexibility of the framework.

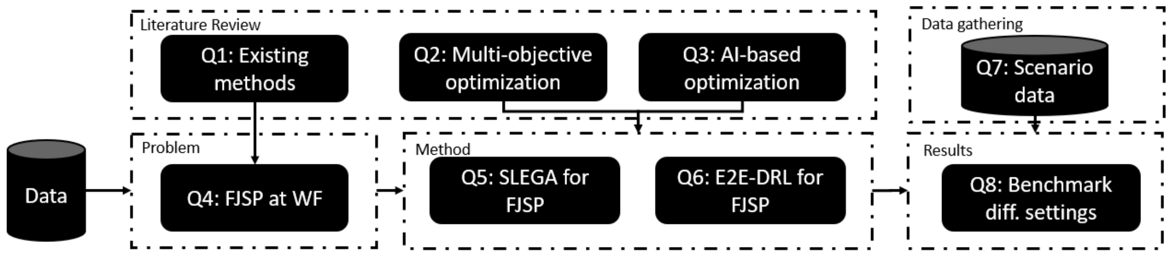


Figure 1.5: Research setup

For Q5 and Q6, we implement two AI-based solutions to solve the FJSP problem at WF. The goal also is to benchmark these implemented algorithms against the simpler heuristics identified in Q1.

1.2.3 Outline

The rest of the document is structured as follows. Chapter 2 thoroughly describes the problem at hand, by listing all existing characteristics and by formulating the problem as an mixed-integer linear programming (MILP). Chapter 3 covers relevant literature from previous work that has addressed similar problems. More specifically, we describe exact, heuristic, and reinforcement learning-based solutions. Chapter 4 covers the solutions methods proposed to solve the problem at hand using the reviewed literature. The implemented solution will then be taken through various experiments, which are described in Chapter 5. Chapter 6 covers the results of these experiments. Finally, this research is concluded in Chapter 7, where recommendations for the company are made.

1.2.4 Contributions

The research aims to enhance our understanding of job scheduling by applying AI-based optimization algorithms to specific job-scheduling scenarios. This approach provides new insights into job scheduling and represents a novel contribution to the field. More specifically:

1. We propose SLEGA, a hybrid heuristic based on the work of G. Zhang et al. (2011) and R. Chen et al. (2020).
2. We compare the performance of the SLEGA and E2E-DRL approaches for the vanilla FJSP, filling the gap in the existing literature that lacks such a comparison.
3. We illustrate how the FJSP instance size and complexity influence the difference in the makespan performance of these algorithms, showing that there is no “one-size-fits-all” for this type of problem.
4. We show that to effectively tackle the FJSP with SDSTs using the E2E-DRL approach, the setup times can be represented as an additional feature on the arcs of the heterogeneous graph. This is a novel extension to the work of Song et al. (2022) which has not been done before, showing competitive results in this particular context.
5. We show that the SLEGA approach is better at generalizing to new instance sizes and more flexible in adopting new problem characteristics.
6. We show that a self-learning module helps guide optimization for highly-constraint FJSPs, both in a single- and multi-objective setting.

Chapter 2

Problem Formulation

In this chapter, we outline the problem that is being studied. More specifically, we first introduce the general problem setting. Then, we deep dive into different problem characteristics within the scheduling environment of Wefabricate, formalizing the problem as a linear programming model. Finally, we detail how the objective values are calculated.

2.1 Job Scheduling

Job scheduling, also known as job sequencing, is a well-known combinatorial optimization problem (Potts, 1980). Combinatorial optimization (CO) problems are usually situated in discrete space. Korte and Vygen (2012) defined combinatorial optimization more formally as such:

Definition 2.1.1 (Combinatorial Optimization)**.** Let V be a set of elements and $f : V \longmapsto \mathbb{R}$ be a cost function. Combinatorial optimization problems aim to find an optimal value of the function f and any corresponding optimal element that achieves the optimal value on the domain V.

In the case of job scheduling, V is the set of all feasible sequences of jobs, and the makespan of a solution usually represents the function f. The goal of this CO problem thus is to find the solution which minimizes the makespan. The size of the set V is of size $O(n!)$ in the simplest variant of the job scheduling problems, where n represents the number of jobs to schedule. More difficult variants will thus have an even larger search space.

Job scheduling is especially popular in industrial engineering, and the problem exists in various types with different characteristics. Graham et al. (1979) introduced a convenient notation to indicate the type of job scheduling problem at hand. The notation is as follows: (α, β, γ), where α indicates the machining environment, β the job characteristics and γ the objective function.

2.1.1 Environment

For the machining environment (α), there are 7 different options. Single-machine scheduling (**1**), Identical-machine scheduling (**P**), Uniform-machine scheduling (**Q**), Unrelated-machine scheduling (**R**), Open-shop scheduling (**O**), Flow-shop scheduling (**F**) and Job-shop scheduling (**J**). Job-shop scheduling refers to the situation where there exist j jobs consisting of n_j operations which have to be executed in the given order (Graham et al., 1979). Figure 2.1 illustrates the job-shop problem with six jobs, six stages and six machines. The length of the bar reflects the processing time of scheduling a stage of a job on a certain machine. Notice that each job has a different order of machines.

In case there exist multiple types of machines capable of executing each operation, the definition of job-shop scheduling is generally extended to flexible job-shop scheduling (FJSP) (Chan et al., 2006). This extension is displayed in Figure 2.2. Note how stage 2 of job 2 is moved from machine 1 to machine 3, indicating that both machines 1 and 3 are capable of executing this operation.

This definition is further extended to account for parallel machines, which means that for each type of machine, there are multiple machines available. These available machines share processing times and setup times for each operation. Figure 2.3 below illustrates this new setting, where jobs are allocated to machine groups rather than to machines (J. C. Chen et al., 2012).

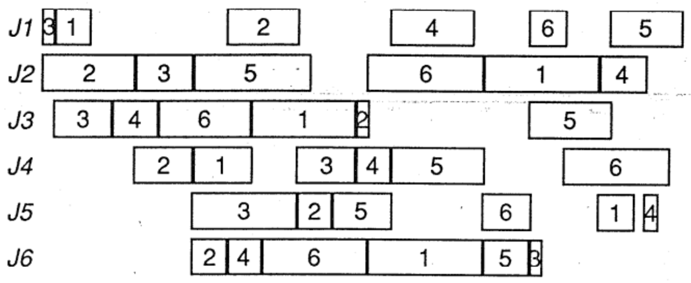


Figure 2.1: Job-Shop Scheduling, adopted from Yamada and Nakano (1992)

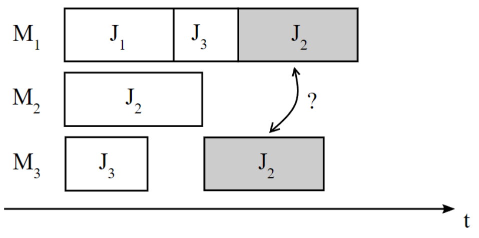


Figure 2.2: Flexible Job-Shop Scheduling

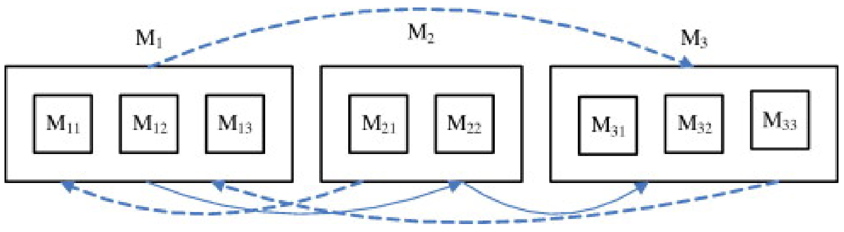


Figure 2.3: FJSP with parallel machines, adopted from J. C. Chen et al. (2012)

2.1.2 Characteristics

The job characteristics (β) are diverse and likely different for each scheduling environment. Example characteristics are given in Table 2.1 below. In Section 2.2.1 we display the job characteristics at play at Wefabricate, and indicate how these characteristics can be modelled according to literature.

Symbol	Name
l_{ij}	Sequence Dependent Setup Times between job i and job j
r_j	Release Date of job j
Various (e.g. chains)	Precedence Constraints between job/operation i, j
Due date (d_j) or strict deadline ($\bar{d}_j$)	Time Windows of job j
t_{mn}	Transportation Delays between machines m and n
DJA	Dynamic Job Arrival
PM	Planned Maintenance
O	Outsourcing
RR	Renewable Resources
CR	Consumable Resources
M_j	Multi-purpose Machines (set of purposes of machine j)
$PaMa$	Parallel Machines

Table 2.1: Example characteristics of job scheduling environment.[1]

2.1.3 Objectives

The most common objective function (γ) in job scheduling is the minimization of the maximum completion time (makespan). Besides this objective function, various other objectives are considered within job scheduling. Other example objectives are listed in Table 2.2 below.

Symbol	Name	Objective
C_{max}	Maximum completion time, or makespan	Minimize
$\sum_{j \in J} C_j$	Total completion time of all jobs J.	Minimize
L_{max}	Minimize maximum lateness	Minimize
$\sum_{j \in J} U_j$	Number of tardy jobs (J)	Minimize
$\sum_{i \in J} \sum_{j \in J, i \neq j} l_{ij}$	Setup time between two jobs i and j	Minimize
P	Profit	Maximize
T	Throughput	Maximize

Table 2.2: Example objectives in job scheduling.

Job scheduling can also be considered in a multi-objective setting. For example, Zhu and Tianyu (2019) tried to minimize total energy consumption while also minimizing the total weighted completion time. Wang and Pan (2019) tried to minimize the makespan while also minimizing the total tardiness of jobs. Sekkal and Belkaid (2020) also tried to minimize the makespan, but rather also tried to minimize the consumed resources rather than the tardiness of a job. In the next chapter, we illustrate how multi-objective problems can be tackled.

2.2 Job Scheduling at Wefabricate

Now that we have described the general problem, we describe the specific scheduling problem at play at Wefabricate in this section. Using the notation from Graham et al. (1979) outlined in the previous section, the specific scheduling problem could be formulated as such:

$$(J, \{d_j, l_{ij}, r_j, DJA, RR, PM, PaMa, NT\}, MOO) \tag{2.1}$$

As we have a job-shop scheduling problem (JSP) problem (symbolized with J, abbreviated as JSP) with the following characteristics: Due dates (d_j), sequence-dependent setup times (l_{ij}), release dates (r_j), dynamic job arrival (DJA), renewable resources (RR), planned maintenance (PM), Night Times (NT) Identical Machines (IM). Furthermore, we are dealing with a multi-objective optimization (MOO) problem, for which the multiple objectives are to minimize the makespan, while also accounting for the costs incurred in doing so. Similar to (Chan et al., 2006), we extend the definition of JSP to FJSP. We do so as there exist multiple types of machines capable of executing each stage per job. Furthermore, looking towards the future setting of Wefabricate, an FJSP setting is even more flexible than a flexible flow-shop scheduling problem (FFSP) setting (Lunardi, 2020), hence an FJSP framework is further explored. The next section further details the specific problem characteristics.

2.2.1 Problem Characteristics

Table 2.3 below displays all of the characteristics of the FJSP environment at Wefabricate, and indicates how those characteristics can be modelled. The characteristics are the following. Each machining job (creating a part) consists of one or more stages (i.e., milling steps, also referred to as operation). Each of these stages can be executed on a different type of machine, for which the processing time differs per machine. This characteristic is known as "flexible machines" in literature. Furthermore, for some machines, we have more than one machine available, which is known as "identical machines". Setups required between each operation are referred to as sequence-dependent setup-times (SDST). For each job, customers define an order quantity and deadline. Internally, a bill of materials is also defined.

The machines on which the operations are scheduled, also have standard maintenance jobs (single stage) which have to be executed. Furthermore, each operation requires a set of resources, both consumable (tools, raw material) and renewable (operator, clamps, pallet). For a created schedule, we require a set of resources (e.g., pallets and tools). The inventory of resources can be increased by means of purchasing (i.e., incurring cost). The schedule also results in an inventory level, which also has a cost attached. Next, dynamic job arrival or machine breakdowns could occur throughout the execution of a schedule.

Since Wefabricate would like to control how the algorithms select the planning horizon, and how the algorithm weights certain factors, this is also included in the job characteristic list. Finally, the problem at hand is considered to be a multi-objective optimization problem as we are optimizing for both the makespan and operational costs incurred because of the schedule. An overview of these characteristics that play a role in the is given in Table 2.3.

We note that most characteristics are static, but that there also exists some dynamic characteristics within the problem (job arrival, tool/machine breakdown). Furthermore, we note that MOO is required. Planning decisions are made at two different moments in time. The first decision is a weekly recurring decision, where the operations for the configured timeframe are scheduled. The next moment is when any dynamic events (like machine breakdowns) occur. At such events, the activities should be rescheduled in order to re-optimize for the new situation. Finally, the created algorithm should be flexible, robust and efficient as explained in Section 1.1.3.

Name	Type	Nature	Applicable to	Description	Modeling Approach
Multi-Stage Milling Process	Characteristic	Static	Jobs	Each milling job is executed in one or more stages (i.e., products can have multiple production steps).	Inherent to FJSP.
Flexible Machines	Characteristic	Static	Jobs	Each stage of a job can be executed on different types of machines.	Inherent to FJSP.
Identical Machines	Characteristic	Static	Machine	For each type of machine, we have 1 or more machines available.	Introduce a variable M_i, which indicates the number of machines that are available for type machine type i (J. C. Chen et al., 2012).
Sequence-Dependent Setup Times	Characteristic	Static	Machine	In between jobs, material, waste flow, pallets, clamps, tops and tools might have to be adjusted for the next job.	Introduce a variable s_{ij} which represents the set-up time for immediately scheduling job j after job i, use this variable when calculating performance (Ying and Cheng, 2010)
Job Deadline	Characteristic	Static	Jobs	A job needs to be finished before a set moment in time.	Model deadline as a soft or hard constraint (Kaplan and Rabadi, 2012).
Job Release Date	Characteristic	Static	Jobs	A job cannot start before a certain moment.	Release dates can be modelled by fixing proposed solutions (Al-harkan and Qamhan, 2019).
Job Quantity	Characteristic	Static	Jobs	Multiple products are made under a single job	Quantity can be modelled by splitting jobs into multiple jobs, or by increasing operation duration.
Maintenance jobs	Constraint	Static	Machine	Machines need an hour a week of maintenance and a longer session of 4 hours per month, which can be scheduled as preferred.	Introduce the notion of timeslots, jobs have to be allocated to a timeslot (Lei and Yang, 2022). Timeslots are created to ensure planned maintenance times aren't crossed. For flexible maintenance jobs, add to the list of jobs to plan.
Resource Required	Constraint	Static	Jobs	An operation requires different resources, both consumable (bill of material and tool lifetime) and renewable (operator, tools, pallet, clamps, tops)	Fix created solutions based on resource constraints (Al-harkan and Qamhan, 2019).
Cost of Resources	Characteristic	Static	Jobs	Resources can be added at a cost (e.g., extra tool costs $300).	Add to the cost function, applied when soft resource constraint is broken.
Cost of Inventory	Characteristic	Static	Jobs	Inventory in the factory could lead to a certain cost. (e.g., when jobs finished early or WIP inventory is generated).	Add to cost function.
Night times	Characteristic	Static	Environment	There are no operators available at night.	Similar to maintenance jobs, timeslots are blocked for when scheduling is unavailable.
Dynamic Job Arrival	Characteristic	Dynamic	Jobs	Customers might send express orders which might have to be prioritized.	Complete reschedule or warm-start (Hamzadayi and Yildiz, 2016).
Breakdowns	Characteristic	Dynamic	Machine	Machines and tools might break down and go in to error state and require unforeseen maintenance.	Complete reschedule or warm-start (Hamzadayi and Yildiz, 2016).
Planning Horizon	Characteristic	-	Environment	The planning horizon should be controllable.	Input to scheduling.
Controls	Characteristic	-	Environment	The company would like to be able to control parameters of the objective function and decision points.	Adjust objective function.
Multi-Objective Optimization	Objective	-	Environment	For now, the goal is to optimize for both 1) the makespan and 2) cost incurred due to operations. I.e., there is a trade-off between speed and cost.	Multiple objective values.
Flexibility, Robustness and Efficiency	Algorithm reqs.	-	Environment	The created algorithm should be robust to different lists of jobs, flexible to include other production cells, and efficient for fast re-planning.	-

Table 2.3: Problem characteristics applicable in the business context.

2.2.2 Mathematical model

To illustrate the problem at hand, we proceed to formulate the problem as a linear programming model while making the following simplifications: (1) Dynamic job characteristics are not included within the model. (2) Maintenance jobs are excluded. (3) An infinite number of tools, materials, clamps, clamp tops and operators is assumed. (4) Identical machines and order quantity are ignored. Furthermore, we only consider costs incurred by missing deadlines. We define the following formulation for the remainder of the problem characteristics. The formulation is based on the work conducted by Özgüven et al. (2010).

Type	Variable	Description
Set	$J = \{J_1, J_2, \ldots, J_n\}$	Set of jobs, where n is the total number of jobs
Set	$O = \{O_{1,1}, O_{1,2}, \ldots, O_{i,j}\}$	Set of operations, where i is the job number and j is the stage number
Set	O_i	Set of ordered operations, for job Ji.
Set	$M = \{M_1, M_2, \ldots, M_m\}$	Set of machines, where m is the machine number
Set	Mij	Set of alternative machines where operation O_{ij} can be processed.
Constant	t_{ijm}	Processing time for operation O_{ij} on machine M_m
Constant	$p_{iji'j'}$	Sequence-dependent setup times for scheduling operation O_{ij} after operation $O_{i',j'}$ on machine M_m.
Constant	d_i	Deadline of job J_i.
Constant	e_i	Cost of missing deadline of job J_i.
Constant	n_i	Length of job J_i
Constant	H	A huge number.
Decision variable	S_{ijk}	Starting time of operation $O_{i,j}$ on machine k.
Decision variable	X_{ijk}	Decision to schedule operation $O_{i,j}$ on machine Mk.
Decision variable	$Y_{iji'j'}$	Decision indicating whether operation O_{ij} preceeds operation $O_{i'j'}$.
Decision variable	C_i	Completion time of job J_i.
Decision variable	C_{max}	Maximum completion time over all jobs (makespan).
Decision variable	z_i	Variable indicating whether job J_i is missed the deadline or not.

Table 2.4: Variable definition

With this notation, we can now introduce the mathematical model and we define the following objectives. 2.2 is the first objective and covers the goal of minimizing the completion of the whole schedule (i.e., the makespan), 2.3 covers minimization of the cost incurred due to missing deadlines.

$$\min \quad C_{max} \tag{2.2}$$

$$\min \quad \sum_{i \in J} z_i \times e_i \tag{2.3}$$

Subject to:

$$\sum_{k \in M_{ij}} X_{i,j,k} = 1 \qquad \forall i, j \in O \tag{2.4}$$

$$S_{ij} \geq S_{i(j-1)} + \sum_{k \in M_{i(j-1)}} t_{i(j-1)k} \times X_{i(j-1)k} \quad \forall j \in \{2, \ldots, , n_i\}, \forall i \in J \tag{2.5}$$

$$S_{ij} \geq S_{i'j'} + t_{i'j'k} + p_{i'j'ijk} - (2 - X_{ijk} - X_{i'j'k} + Y_{i'j'ij}) \times H \forall O_{ij}, O_{i'j'} \in O \times O, O_{ij} \neq O_{i'j'}, k \in M_{ij} \cap M_{i'j'} \tag{2.6}$$

$$S_{i'j'} \geq S_{ij} + t_{ijk} + p_{i'j'ijk} - (3 - X_{ijk} - X_{i'j'k} + Y_{i'j'ij}) \times H \forall O_{ij}, O_{i'j'} \in O \times O, O_{ij} \neq O_{i'j'}, k \in M_{ij} \cap M_{i'j'} \tag{2.7}$$

$$C_i \geq S_{in_i} + \sum_{k \in M_{in_i}} t_{in_ik} \times X_{in_ik} \quad \forall i \in J \tag{2.8}$$

$$C_i + z_i \times H \leq d_i \quad \forall i \in J \tag{2.9}$$

$$C_{max} \geq C_i \quad \forall i \in J \tag{2.10}$$

and

$$\begin{aligned}
&X_{i,j,k} \in 0,1 \quad \forall i,j \in O, \forall k \in M \\
&z_i \in 0,1 \quad \forall i \in J \\
&Y_{i,j,i',j'} \in 0,1 \quad \forall i,j \in O, \forall i',j' \in O \\
&S_{ijk} \geq 0 \quad \forall i,j \in O, \forall k \in M \\
&C_i \geq 0 \quad \forall i \in J
\end{aligned}$$

In words, Constraint 2.4 ensures all operations are allocated to one and only one eligible machine. Constraint 2.5 ensures the presence of relations between jobs. Constraints 2.6 and 2.7 help avoid overlapping operations on the same machine M_k. Constraint 2.8 sets the completion time of each job. Constraint 2.9 then indicates whether this completion time met the deadline or not. Finally, Constraint 2.10 calculates the total makespan. The remaining equations ensure the decision variables take values on a feasible space.

This formulation encompasses several key problem characteristics, including milling processes, flexible machines, sequence-dependent setup times, and order deadlines. The remaining problem characteristics will be modelled as such. Identical machines can be included by adding additional machines to the set of machines with identical processing times. The Bill of Materials of jobs will be modelled as a release date, this date represents the moment that all input materials are available for the job to be executed.

Order quantity can be modelled by creating an individual job for every item in an order. Since this might explode the number of jobs (i.e., an order size of 100 would result in 100 separate jobs), a batch size will be used for grouping identical products. A limitation here is that the precedence constraints for the different stages limit the search space. For example, if we assume a batch size of 25, the second stage can only be started when all 25 products have finished stage 1. While in practice, stage 2 could already start when the first product of the batch is finished. The batch size will be tuned and adjusted later.

Regarding resources, we will assume the infinite capacity of tool lifetimes, as tools can be created in parallel to the execution of jobs. Tools are created on a different machine with a very low utilization so this assumption is quite reasonable. For renewable resources, we will assume that the number of resources available can be added at a certain price, and thus it will be modelled as a soft constraint. E.g., exceeding resource capacity will result in a cost (e.g., purchasing extra clamps), rather than an unfeasible schedule. Furthermore, during operating hours, infinite operators will be assumed available. At night, no operators are available. The operating hours are to be defined and can be adjusted for every scheduling instance.

Next, inventory costs are also added as a soft constraint, and will be included within the schedule evaluation function. Finally, dynamic characteristics (breakdowns, express orders) happen on a very infrequent basis. Hence they are ignored in the model but are tackled by complete rescheduling when a dynamic event occurs.

2.2.3 Objective Values

The linear program defined in the previous section relaxes the cost objective. This section further details the actual cost considered at Wefabricate which will be considered in the experiments of this research. In case time units are considered below, they are given in seconds.

The first objective, makespan, is minimized. The makespan is given by the latest moment in time a job is finished, or:

$$C_{max} = max_{\forall i \in J} C_i \tag{2.11}$$

The second objective, operational cost, is a little more complicated and consists of the components outlined below. We follow the same notation as in the previous section, defining new notation where necessary.

Missed Deadlines

Missing deadlines damages customer satisfaction and customer goodwill towards Wefabricate, which will eventually lead to decreased customer retention. In order to account for this in the scheduling algorithm, a missed deadline is considered to cost 50% of the product price (p_j). The total cost for missing deadlines (c_{dl}) is thus calculated as such:

$$c_{dl} = \sum_{j \in J} \mathbb{1}\{C_j > d_j\} \times 0.5 p_j \tag{2.12}$$

Finished Goods Inventory

If products are finished before the deadline, they are stored in the warehouse until they can be shipped to the customer. Wefabricate has the possibility to rent out pallet slots to third parties. This is considered an opportunity cost when utilizing its own storage capacity. The opportunity cost comes at a value of 5 euros per pallet per week. For simplicity, we consider that 25 units can be placed on each pallet, whereas the quantity per job (q_j) varies per job. The total finished goods inventory cost (c_{fgi}) is thus calculated as such:

$$c_{fgi} = \frac{5}{(7 \times 24 \times 60 \times 60 \times 25)} \sum_{j \in J} (d_j - C_j) \times q_j \tag{2.13}$$

where the fraction of the formula covers the cost of a pallet per week, given the right part of the formula is given in seconds.

Work-In-Progress Inventory

Work-in-progress (WIP) inventory arises because of a potential slack between the completion of a certain stage, and the start of the consecutive stage. As this WIP inventory is placed close to the machines, a high amount of inconvenience is experienced by operations. More specifically, the threshold for the wip at any moment in time (wip_t) considered inconvenient is set at 25 units. The cost associated then is 10€ per minute this threshold is exceeded, hence the fraction divides by 60.

$$c_{wip} = \sum_{t=0}^{C_{max}} \mathbb{1}\{wip_t > 25\} \times \frac{10}{60} \tag{2.14}$$

Addition of Resources

In order to execute the schedule, various different renewable resources are used. Resources (R) consist of pallets, clamp tops and tools. During scheduling, these resources might be allocated to different machines at the same time. As the amount of inventory for that resource (I_r) might not meet the required quantity (Q_r), resources might have to be added at a certain cost. The cost considered here is the value v_r of the resource.

$$c_{ar} = \sum_{r \in R} max(Q_r - I_r, 0) \times v_r \tag{2.15}$$

Manual Operations

Even though switching materials and tools might be parallelizable, they still require an operator to execute this manual labour. Manual labour is valued at 1.5 € per minute and is used as a proxy to compute total manual cost (c_{mo}). The formula below also indicates the number of minutes each manual operation takes. E.g., an orientation change takes 3 minutes.

$$c_{mo} = 1.5 \times (3 \times |OrientationChanges| + 8 \times |PalletChanges| + 5 \times |WasteFlowChanges| + 1 \times |ToolChanges| + 5 \times |NewTools| + 3 \times |TopChanges|) \tag{2.16}$$

Logistic Operations

As we have multiple different machine options per stage, it could occur that a second stage is executed on a different machine, while it also could have been executed on the machine that handled the first stage. In this case, we require extra logistic operations (i.e., moving the product from machine A to machine B). For each mandatory logistical operation, a cost of 50 € is considered as we require the products to travel a certain distance. Logistic operations are minimized in a subroutine by delaying all logistic operations as long as possible while grouping those operations that could be executed at the same time (i.e., both preceding stages are completed before the start time of either following stages). The total cost of logistics (c_{lo}) are calculated as such:

$$c_{lo} = 50 \times |LogisticOperations| \tag{2.17}$$

The total cost of schedule execution (c_{tot}) is then simply given by taking the sum of the above cost. Or:

$$c_{tot} = c_{dl} + c_{fgi} + c_{wip} + c_{ar} + c_{mo} + c_{lo} \tag{2.18}$$

Now that the problem formulation is complete, we explore solution directions according to the literature in the next chapter.

Chapter 3

Literature Review

In this chapter, the existing literature is reviewed to identify solution directions for the problem that is formulated in the previous chapter. This chapter consists of the following sections. First, we describe how multi-objective optimizations are generally tackled in Section 3.1. Then, we explore solutions directions. According to Lunardi (2020), combinatorial optimization problems can be approached using either exact or heuristic methods. Table 3.1 below displays the prevalence in literature for each class of optimization technique. As can be seen, FJSPs have been more frequently solved using heuristic methods rather than exact methods. Exact and approximation solutions are covered in Section 3.2, heuristic solutions are covered in Section 3.3 and reinforcement learning-based approaches in Section 3.4. Finally, we draw conclusions based on the overview of the current state of the literature on solving the problem at hand in Section 3.5.

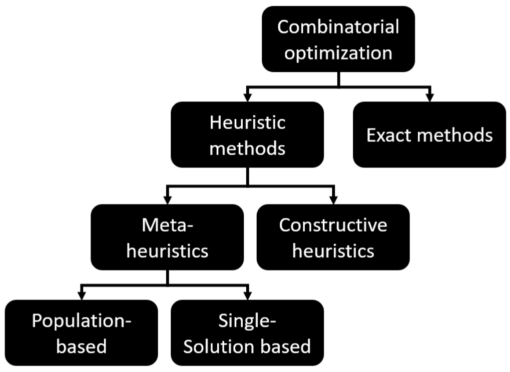


Figure 3.1: Overview of optimization methods

Technique	Class	Number of papers	Percentage
Hybrid	Mix of heuristic methods	69	35.03%
Evolutionary algorithm	Population-Based	47	23.86%
Constructive algorithm	Heuristic methods	19	9.64%
Tabu search	Single solution based	12	6.09%
Integer linear programming	Exact methods	10	5.08%

Table 3.1: Techniques applied to solve FJSP, 1990 - 2016 (Lunardi, 2020).

3.1 Multi-Objective Optimization

Multi-objective optimization originally grew out of 3 main areas; economic equilibrium and welfare theories, game theory, and mathematical optimization (Marler and Arora, 2004). In this field, we try to optimize for not one but two or more objectives. In order to solve a multi-objective optimization problem, one can take several approaches. The book of Burke and Graham (2014) gives an overview of what approaches could work. Generally, these approaches follow two steps; (1) Find multiple trade-off optimal solutions with a wide range of values for objectives, and (2) Choose one of the obtained solutions using higher-level information. From step 1, a Pareto front follows (see Figure 3.2). This front contains all non-dominated solutions. A dominated solution is defined as such:

Definition 3.1.1 (Domination)**.** A solution j is said to dominate a solution i if the following two conditions are met (Burke and Graham, 2014).

$$f_a(x_j) \geq f_a(x_i) \quad \text{for all objective functions } a$$

$$f_a(x_j) > f_a(x_i) \quad \text{for at least one objective functions } a$$

In words, the objective values for solution j are no worse than those of solution i, and solution j has at least one objective value where solution j is better than solution i. Such a solution is also called Pareto optimal (Marler and Arora, 2004).

In case the Pareto front only exists of a single point (i.e. this point has optimized values for all objectives), then this point is called a utopia point (Marler and Arora, 2004).

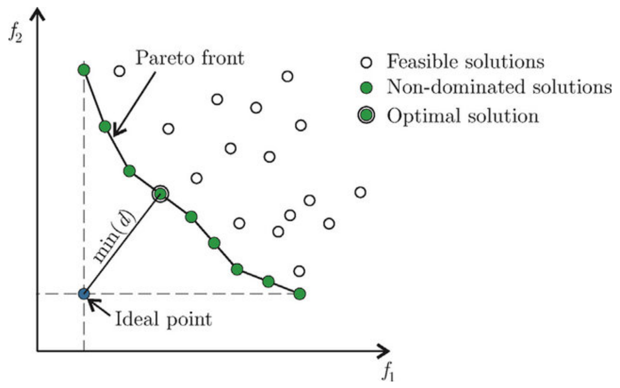


Figure 3.2: Pareto front illustrated for a two objective minimization problem.

To determine the Pareto front, one simple approach would be to deploy an iterative procedure that retains a set of non-dominated solutions and checks whether any of these solutions is dominated by other potential solutions. After doing so, the set of non-dominated solutions remains which resembles the Pareto front. This procedure runs in $O(MN^2)$, where M is the number of objectives and N is the number of possible solutions (Burke and Graham, 2014). The computational cost thus increases rapidly when the number of possible solutions does too.

In order to evaluate the quality of the approximated Pareto front (PF), a popular measure is the front's hypervolume (Cao et al., 2015). The hypervolume measures the size of the space enclosed by all solutions on the Pareto front and a user-defined reference point. More formally, the hypervolume ($I_H(PF, r)$) is defined as such:

$$I_H(PF, r) = \lambda(\cup_{\mathbf{s} \in PF} space(\mathbf{s}, \mathbf{r})) \tag{3.1}$$

where s and r are vectors with the length equal to the number of dimensions (in this case, objectives). λ is the Lebesgue measure. The hypervolume for a Pareto front with two objectives is visualized in Figure 3.3 below.

Cao et al. (2015) mention there are multiple drawbacks of evaluating the Pareto front using the hypervolume. The most important drawback is the reference point selection. The reference point is usually set to the nadir point (or slightly worse) of the Pareto front under investigation when the true Pareto front is unknown. The nadir point is defined as the point which has the worst criteria in all objectives

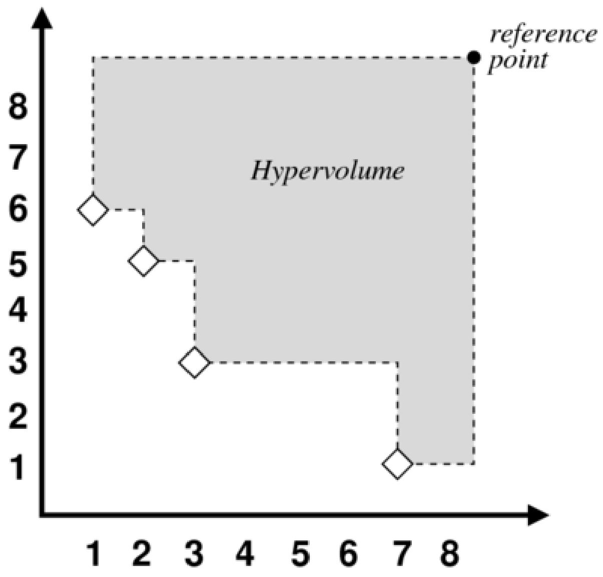


Figure 3.3: Hypervolume of Pareto front in 2D space, from Fonseca et al. (2017)

(point $(7, 6)$ in Figure 3.3). Because of the fixed reference point, hypervolume measures can decrease in magnitude (i.e., the magnitude of improvement) whilst the Pareto front keeps evolving in quality.

Now we have a basic understanding of multi-objective optimization, we dive into flexible job-shop scheduling solutions in the next sections.

3.2 Exact Computation and Approximation Schemes

The JSP problem is well-known to be NP-hard, the problem is even considered one of the hardest CO problems by many (Lunardi, 2020). Since FJSP is a combination of an assignment problem and a classical JSP, the problem is at least as hard as the classic JSP. Since the assignment problem is NP-hard as well (Özbakir et al., 2010), it can be safely concluded that the FJSP is very complicated to solve too. Nonetheless, there exist different approaches to solving the FJSP. These are further outlined below.

3.2.1 Mathematical Programming

Mathematical programming is the first mathematical approach outlined here. Within mathematical programming, linear programming (LP) is one of the most satisfactory models for solving optimization problems (Talbi and El-Ghazali, 2009). LP models can be solved using efficient exact algorithms such as the simplex-type method or inferior-point method within continuous decision space (Lunardi, 2020). These methods are efficient since the feasible region of the problem is a convex set.

Within discrete decision space, on the other hand, efficient exact algorithms are less developed (Talbi and El-Ghazali, 2009). For example, in integer programming (IP) decision variables are discrete (i.e. $\in \mathbb{Z}$). There are also mixed integer programming (MIP) models. Both MIP and IP models can be solved using the branch and bound algorithm.

The branch and bound (BB) algorithm searches a dynamically constructed tree representing all feasible solutions (i.e. exhaustive search). According to Burke and Graham (2014), BB reduces the number of alternatives that need to be considered by partitioning possible solutions into a smaller set of sub-problems which can then be shown to be sub-optimal and thus eliminated from the set. Even though this procedure has evolved to exclude large proportions of the tree that is searched, the algorithm is still not capable of handling the explosive growth of the solution space. That is, the complete enumeration of allocating n operations to n machines would result in $n!$ terminal nodes. Which approximates to 2.4×10^{18} terminal nodes for n = 20 (Lunardi, 2020).

3.2.2 Constraint Programming

Constraint programming (CP) builds upon mathematical programming in a smart way Talbi and El-Ghazali, 2009. For example, let's assume we are trying to model the constraint that all jobs (I) should be allocated to a different machine within all machines (M). Rather than introducing a binary variable

x_im for allocating a job i to a certain machine m, we can create an integer variable $x_i, x_i \in 1, \ldots, |M|$. Thus the number of decision variables is reduced from $O(|I| \times |M|)$ to $O(|I|)$, allowing for more efficient computation and optimization (Lunardi, 2020).

With numerical experiments, Lunardi (2020) shows that a CP Optimizer is able to find feasible solutions to large-scale instances (e.g. 50 machines 100 jobs, 1000 operations), unlike the MIP solver. However, the author also mentioned that ad-hoc heuristics most likely outperform the CP optimizer, as they are able to fully exploit the specificities of the problem.

Approximation algorithms

A way to go about NP-hard problems is to deploy approximation algorithms. Approximation algorithms are defined as such:

Definition 3.2.1 (Approximation algorithm). $f(x)$ is a $p-$approximation algorithm if:

$$\begin{cases} OPT \leq f(x) \leq p \times OPT, & \text{if} \quad p > 1. \\ p \times OPT \leq f(x) \leq OPT, & \text{if} \quad p < 1. \end{cases}$$

Where $p < 1$ is used for maximization and $p > 1$ is used for minimization problems.

The goal of the approximation algorithm is to create an algorithm which runs in polynomial time and has a bound on the error margin (denoted with ϵ). These algorithms are called polynomial time approximation schemes, or PTAS in short. For example, Arora (1998) designed a $1 + \epsilon$ approximation for the Euclidean TSP, which had a prohibitive running time of $n^{O(\frac{1}{\epsilon})}$. This was improved later to run in near-linear time.

Regarding the FJSP, we have seen that each of its components has a PTAS available for a specific case. Generalized assignment problems can be approximated using LP-relaxation, resulting in a $(1 - \frac{1}{\varepsilon})$-approximation (Fleischer et al., 2006). Furthermore, standard JSPs can be approximated using a PTAS which gives a $(1 + O(\epsilon))$ bound on the makespan (Jansen et al., 2000). Jansen et al. even finds a $(1 + O(\epsilon))$ bound on discrete problem examples. However, they assume that the processing times of jobs are controllable.

Regarding the direct approximation of the FJSP, Jansen and Mastrolilli (2004) present a $(2 + \epsilon)$ linear time approximation scheme for the situation where the number of machines m and number of operators u is fixed. Besides the author's work, there has not much work been conducted on approximation schemes for FJSP.

All in all, we conclude that it is not possible to efficiently or effectively solve a heavy constraint, multi-objective, FJSP situated in discrete decision space, mathematically. Furthermore, there are few to no approximation schemes available which provided a good bound on the solution. Particularly for the FJSP with the characteristics of Wefabicate, no approximation scheme exists. Thus we further investigate solving the FJSP of Wefabicate using machine learning and other AI-based algorithms and heuristics in the next chapter.

3.3 Heuristic Solutions

According to Lunardi (2020), heuristic methods can be divided into three categories; Constructive heuristics and single-solution or population-based metaheuristics (Figure 3.1). Each of these categories is further explored in the following sections. Furthermore, hybrid heuristics are also investigated as they have been proven to be most popular for the FJSP (see Table 3.1).

3.3.1 Constructive Heuristics

In this section, constructive heuristic solutions are covered which aim at solving FJSPs. Constructive heuristics generally start with an empty solution, and repeatedly extend the current solution until a complete solution is obtained (Lunardi, 2020). Koulamas (1998) were one of the first to solve a job scheduling problem using a constructive heuristic. In their work, the authors solved a non-constraint flow shop scheduling problem using a constructive heuristic. This constructive heuristic consists of two phases. In phase one a permutation sequence is created as input for phase two. Subsequently, phase two then has the capability of consuming this permutation sequence and can produce a non-permutation sequence if it is found appropriate. The created algorithm still had a computational complexity of $O(m^2n^2)$, and only worked for non-permutational schedules.

For FJSP specifically, Liang et al. (2021) use a constructive heuristic from Abreu et al. (2020) to create the initial population used for their genetic algorithm. The constructive heuristic (called BICH-MIH) works in the following way: A performance indicator (Ψ_{ij}) is calculated for the insertion of operation π_{ij}. Then, while all jobs are not allocated yet, find the machine that has the earliest finishing time. Next, based on the performance indicator, find the remaining operation which is best suited for this machine, and add the operation to the machine. In the paper, the performance indicator is designed to optimize for a combination of idleness and completion time minimization (based on weight α).

3.3.2 Metaheuristics

In this section, we cover metaheuristics used for solving the FJSP at hand. These metaheuristics can be both single-solution and population-based. Metaheuristics are defined as high-level problem-independent frameworks that provide a set of guidelines or strategies to develop heuristic optimization algorithms (Sörensen et al., 2012). There are many types of metaheuristics applied in the literature to the FJSP at hand. The most prevalent algorithms are outlined below.

3.3.3 Local Search

Similar to Tabu search, Local Search (LS) is a heuristic method to solve computationally hard problems. Local search algorithms move from solution to solution within a set of candidate solutions by applying local changes, until a solution deemed optimal is found.

Bissoli and Amaral (2018) solve an FJSP using a hybrid iterated local search (HILS) heuristic, using simulated annealing as a local search. After generating an initial solution, HILS works in the following way: a perturbation in the best solution is applied, which is then submitted to a local search algorithm and then evaluated by an acceptance criterion. The local search algorithm consists of the best improvement method, simulated annealing and neighbourhood structures (Bissoli and Amaral, 2018). With empirical research on benchmark datasets (HUdata, BRdata, DPdata and BCdata), the authors show that the approach is robust and competitive when compared to state-of-the-art FJSP algorithms. HILS solves an FSJP with 15 machines, 15 jobs, and 2-3 possible machines per job in 800 seconds.

3.3.4 Tabu Search

Tabu search (TS) uses a local or neighbourhood search procedure to iteratively move from one potential solution x to an improved solution x' in the neighbourhood of x, until some stopping criterion has been satisfied (Glover, 1986). Tabu search works with a memory structure, i.e., a tabu list, in order to keep track of what solutions are not to admit into the neighbourhood anymore. The memory structure contains both short-term and long-term memory components. The tabu list helps with escaping local minima, which is one of the difficulties search algorithms have.

Tabu search has been widely used to find optimal solutions for combinatorial optimization methods (Saidi-Mehrabad and Fattahi, 2007). The authors specifically use it for finding a solution to the FJSP optimizing for the makespan in two steps. The first step finds the best operations sequence and the second step finds the best choice of the machine's alternative. It was found that the tabu search algorithm performs significantly better for larger problem instances than the exact branch and bound technique. The authors are namely able to solve a 3 job, 3 machines, 9 operation problem in 2 minutes whereas branch and bound took a full hour. The method takes 30 minutes to solve a 15 job 6 machine 88 operations problem.

3.3.5 Genetic Algorithms

Genetic Algorithms (GA) belongs to the class of Evolutionary Algorithms (EA) and its development was inspired by the process of natural genetic evolution (Holland, 1992). Within genetic algorithms, an initial population is evolved over several generations until certain termination criteria are met. Within evolution, individuals are **selected**, **crossed-over** and **mutated** for the next generation, based on the "survival of the fittest" principle. This process is displayed in Figure 3.4 below. In this figure, elitism is also displayed. Elitism is implemented to ensure that during selection, the most promising solution is not discarded, but kept in the loop for further evolution.

G. Zhang et al. (2011) applied a genetic algorithm to an FJSP as such. In order to represent a schedule, individuals are encoded using the double-layer encoding technique. The first layer is a process-based coding scheme (also called job sequence) and the second layer is based on machine allocation. In order to compute the fitness (makespan in this case) of a solution, the individuals are decoded by iterating over the job sequence and machine allocation. During decoding, potential opportunities for backfilling

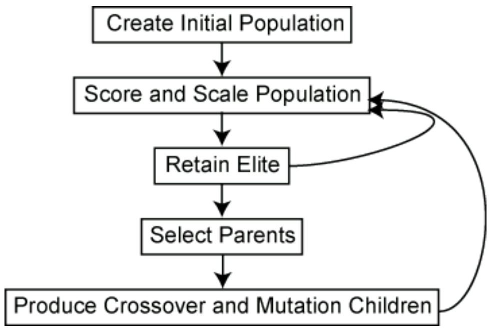

Create Initial Population
Score and Scale Population
Retain Elite
Select Parents
Produce Crossover and Mutation Children

Figure 3.4: GA process with elitism

are checked in order to improve the solution. The first generation is initialized using local, global, and random selection. Throughout the evolutionary process, the authors use tournament selection. Mutation and selection operators occur for each coding layer separately. For the machine allocation layer, two-point crossover and flip mutation are applied. For the job sequence layer, precedence preserving order-based crossover (POX) and swap mutation are applied (G. Zhang et al., 2011). This approach manages to obtain near-optimal solutions for instances containing 20 jobs and 10 machines in about 12 minutes. However, the authors do not indicate to what instance size they could scale.

In the field of multi-objective optimization, a non-dominated sorting genetic algorithm (NSGA) is an algorithm capable of dealing with multiple objectives. The algorithm has a second version too; NSGA-II. NSGA-II was first proposed in Deb et al. (2002) and is still relevant today. As explained in Section 2.4.2, the algorithm is able to find a set of Pareto optimal solutions in $O(MN^2)$ time, where M is the number of objectives and N is the number of individuals. The algorithm is the first multi-objective genetic algorithm able to handle constraints and elitism (Deb et al., 2002).

NSGA-II has the following procedure: (1) Perform a non-dominated sorting in the combination of parent and offspring populations and classify them by fronts. (2) Fill the new population according to front raking. (3) perform a Crowding-sort that uses crowding distance that is related to the density of solutions around each solution. (4) Create an offspring population from this new population using crowded tournament selection, crossover and mutation operators. This technique has been applied extensively to FJSPs. Since genetic algorithms are usually the basis for hybrid heuristics, they are further introduced and described in the next section.

In order to measure the quality of an iteration of a genetic algorithm, there exist multiple solutions. The first one was to measure the hypervolume, which was explained in Section 2.4. Another approach is to compare the input and output generation to a reference set. Liu et al. (2019) compare two generations using the generational distance (GD) and inverse generational distance (IGD). Both IGD and GD calculate the distance between the Pareto front (PF), and the objective vector set (V). The IGD and GD are calculated as such:

$$IGD = \frac{\sum_{p \in PF} dist(p, V)}{|PF|} \tag{3.2}$$

$$GD = \frac{\sum_{v \in V} dist(v, PF)}{|V|} \tag{3.3}$$

These performance indicators are most common as they are simple and have low computational cost (Santos and Xavier, 2018).

3.3.6 Hybrid Heuristics

Hybrid heuristics combine two or more (meta-)heuristics in order to find a solution to the problem at hand. These hybrid heuristics could be combinations of any existing heuristics and thus are very diverse. An example of a hybrid solution is the combination of differential evolution followed by local search used in Yuan and Xu (2013). Since these hybrid approaches are very diverse, they will be covered paper by paper.

In order to apply NSGA to the FJSP, Liang et al. (2021) propose an improved and adaptive non-dominated sorting genetic algorithm with an elitist strategy based on NSGA-II. It consists of 9 steps: (1) set up basic parameters in the algorithm, (2) perform two types of population initialization; random and using a constructive heuristic, (3) generate machine and operation sequence code, calculate fitness values, (4) create child generation using NSGA adaption, (5) merge child and parent generation using simulated annealing strategy, (6) perform non-dominated sorting for operation population, (7) calculate crowding distance and congestion degree of previously created operation population, (8) screen out better Pareto set, (9) check termination criteria, if no termination, return to step 5.

Dai et al. (2019) use a similar approach, where they enhance a genetic algorithm using simulated annealing as mutation operator, and particle swarm optimization (PSO) as crossover operator. More specifically, the same double-layer encoding method is used as in Liang et al. (2021). During crossover, both the machine and operation string are crossed over in a similar fashion. The authors are able to solve problems that consist of 80 jobs and 50 machines. Shahsavari-Pour and Ghasemishabankareh (2013) created a novel hybrid genetic algorithm and simulated annealing (NHGASA) algorithm in order to solve the FJJS problem. Here, SA is used to improve individuals. NHGASA outperforms 3 other methods on popular benchmarks and is able to solve a 10-machine and 10-job problem in 18 seconds.

Rooyani and Defersha (2019) propose a two-stage genetic algorithm to solve an FJSP. Here, the authors also consider sequence-dependent setup times, machine release dates, and time lag. As seen in Figure 4.2, an FJSP individual is usually encoded by means of an order sequence string and a machine allocation string. During the evolutionary process, both of these strings are then manipulated simultaneously. Rooyani and Defersha (2019) propose a different approach, introducing a two-stage genetic algorithm (2SGA) instead of simultaneous gene manipulation. Here, the first stage of the genetic algorithm covers the operation sequence, similar to the operation sequence string in the double-layer encoding. While optimizing the order sequence, the machine that would process an operation the fastest is always selected. The second stage starts from the solutions of the first stage and then follows the common approach of genetic algorithms to search the entire solution space.

In their work, the authors show that the algorithm is applicable to solve many other variants of the FJSP, given the general scheme of the algorithm. More specifically, the algorithm is able to tackle an FJSP with additional constraints, such as sequence-dependent setup times, release dates, and time lag, effectively. Moreover, the authors show that the 2SGA can scale to instance sizes of 100x50 and 140x80, reaching good solutions in 0.34h and 2.50h respectively. A regular genetic algorithm did not converge to a solution while having searched 5h and 70h respectively.

3.4 Reinforcement Learning-Based Approaches

Within reinforcement learning-based Approaches, we distinguish between end-to-end reinforcement learning solutions and reinforcement learning-based heuristics. These are outlined below. However, we introduce reinforcement learning first.

3.4.1 Reinforcement Learning

In Reinforcement Learning (RL), there exists an agent which performs an action on the environment based on its current policy, for these actions the agent receives rewards and updates its policy accordingly. To illustrate, let's take the simple example of a game of pong [1]. The action the agent can take is moving the paddle. The location of the paddles, the location of the ball, and the direction of the ball make up the state of the environment is in.

Starting off, the agent randomly moves the paddle until a reward is received. The reward can be positive (scored a point) or negative (opponent scored a point). The reward allocation function then adjusts the policy of the agent for the states which led up to the reward. For example, when a point was scored, the actions taken by the agent (moving the paddle) in the states leading up to the point are more likely to happen in the next iteration. Eventually, the policy of the agent is optimal and no more updates are required.

The reinforcement learning approach can be used to solve the combinatorial problems outlined above. In order to apply RL to CO, the problem is modelled as a Markov decision process (MDP). An MDP is defined as such (Sutton and Barto, 2018):

[1] https://en.wikipedia.org/wiki/Pong

Definition 3.4.1 (Markov decision process)**.** An MDP is defined as tuple $M = (S, A, R, T, \gamma, H)$ where

- S - state space
- A - action space
- R - reward function, mapping states and actions into real numbers ($R : S \times A \longmapsto \mathbb{R}$)
- T - transition function
- γ - scalar discount factor, $0 < \gamma \leq 1$
- H - horizon defining the length of episodes

In the MDP, the agent performs a sequence of actions on the environment based on its current policy to find a solution. The states of the environment are encoded such that they can be consumed by the RL algorithm. The encoder is required to map the state S to d-dimension space $\mathbb{R}^d$. For example, the connected nodes in a network of a TSP are encoded into a vector. The RL algorithm then determines how the agent learns (i.e. updates its policy $\pi(s)$) and makes decisions for a given MDP.

3.4.2 Deep Reinforcement Learning for FJSP

Since job scheduling can be formulated as a sequential decision-making process, it can also be solved using RL (Aydin and Öztemel, 2000). Chang et al. (2022) show that they are able to solve a dynamic FJSP (DFSJP) with random job arrival using a double deep queue network (DDQN) with a soft ϵ-greedy policy, outperforming other state-of-the-art RL approaches. Here, the authors optimized for penalties incurred due to earliness and tardiness. The authors show that the DDQN outperforms five other methods in terms of solution quality and generalization.

First of all, the DFJSP is modelled as MDP as such. The production state reflects the state the environment is in and contains information like the number of jobs, the number of operations, the number of machines, the remaining working hours, the number of remaining operations, the load of machine tools and the total processing time. This production state is then reduced to four production state features between 0 and 1. The features represent the objectives based on the input features and are used to represent the production state. The four production state features are (1) Average utilization rate, (2) Estimated earliness and tardiness rate, (3) Actual earliness and tardiness rate (4) Actual earliness and tardiness penalty.

The action set then covers both sub-problems FJSPs entail, operation sequencing and machine selection. The action set is built out of four comprehensive dispatching rules, for which we refer back to Chang et al. (2022). Actions are selected according to a soft ϵ-greedy policy, which was designed to adapt to flexible scheduling problems of different scales. Since ϵ represents the probability of exploration over exploitation, it should decay relative to the scale of the scheduling problem. This is exactly what the soft ϵ-greedy policy achieves.

The DQQN architecture (see Figure 3.5) consists of two networks, a target and an online network. These networks are decoupled in order to split the action selected from the action evaluation network. The Q-value from the online network is provided as a basis for action selection, whereas the Q-value from the target network is provided for the evaluation of the selected action. This approach reduces over-estimations of evaluations. Both the target network and online network have the same architecture, such that the weights and biases of the target network can be replaced by the online network's weights and biases every C step.

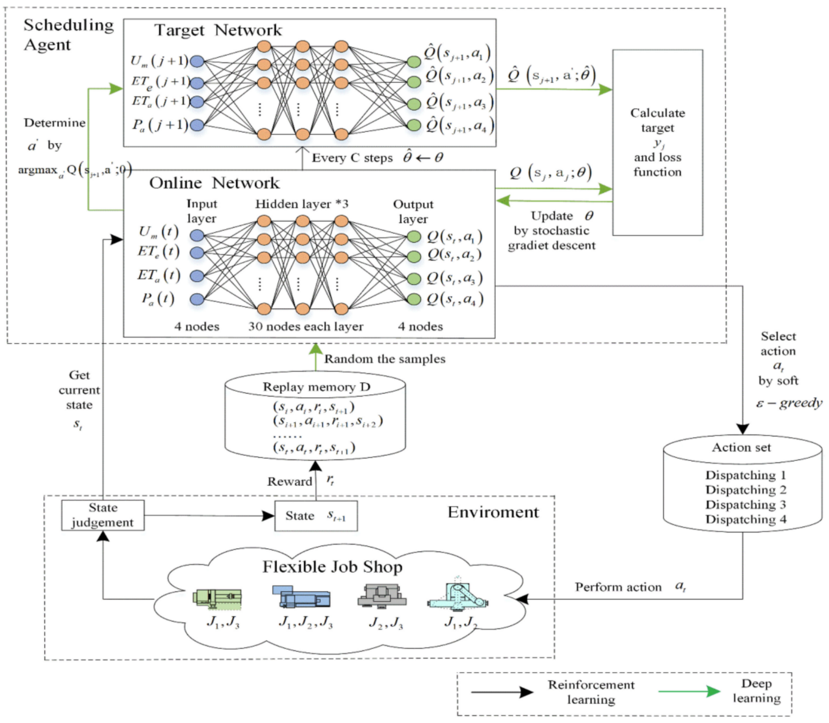


Figure 3.5: DDQN for DFJSP, adopted from (Chang et al., 2022)

In order to learn the DDQN makes use of experience replay. State transition tuples (s_t, a_t, r_t, a_{t+1}) are stored in a replay memory D. This tuple data is then randomly sampled and the target and loss functions are calculated according to the online and target network, to adjust the parameters (θ_t) of the online network.

Song et al. (2022) on the other hand, solves the FJSP problem using a heterogeneous graph neural network (HGNN) combined with deep reinforcement learning. In this paper, the authors also compare results on the MK benchmark dataset with genetic algorithm approaches. The results show that their DRL algorithm can solve 20-job, and 10-machine instances within 3.5 seconds. However, the solution has an optimality gap of over 20%. The SLGA approach on the other hand solves the instance after 280 seconds, with an optimality gap of 6.21%. Google's OR-tools (a CP solver) solves the instance in 900 seconds with an optimality gap of less than 2%. In the next Chapter, we will describe the work of Song et al. (2022) more thoroughly as it serves as basis for one of our algorithms.

3.4.3 Heuristics Controlled by Reinforcement Learning

Even though we saw that DRL can help solve the FJSP at hand, there are other possibilities to leverage the strengths of machine learning. As explained in Section 4.2.2., different parameters needed to be initialized for the GA approach to FJSP. Moreover, the objective function weighting and parameters of other algorithms also need tuning. Hence it is important this tuning is executed in a proper way, as sub-optimal parameter configuration might lead to sub-optimal results and execution speed. Machine learning became key in optimal parameter configuration for meta-heuristics (Karimi-Mamaghan et al., 2022). ML can serve a broader purpose beyond parameter setting. It can be leveraged for tasks such as algorithm selection, fitness evaluation, initialization, evolution, and cooperation. How ML can be used is given in Table 3.2 below.

R. Chen et al. (2020) created a self-learning genetic algorithm (SLGA) for the FJSP by intelligently adjusting the optimization method and key parameters using RL. More specifically, the authors try to optimize for makespan under a standard FJSP setting. The same double-layer coding representation for individuals is used as in Section 4.3. The initial population is initialized with two priority rules: (1) (CMO) chooses the job that has the greatest number of operations remaining and (2) (HCMS) chooses the machine that has the shortest processing time for the job with high probability. Regarding genetic operations, precedence preserving order-based crossover (POX) and swap mutation are applied. Elitist retention is taken as a strategy for selection. A flowchart of the SLGA is given in Figure 3.6 below.

Where?	How?
Algorithm selection	ML can predict the performance of metaheuristics for solving problems.
Fitness evaluation	ML techniques can estimate computationally expensive fitness functions.
Initialization	ML can help generate good initial solutions.
Evolution	ML can help select search strategies.
Parameter setting	ML techniques can control parameter setting before or during evolution.
Cooperation	ML can adjust the behaviour of metaheuristics.

Table 3.2: Where and how ML is used at the service of metaheuristics.

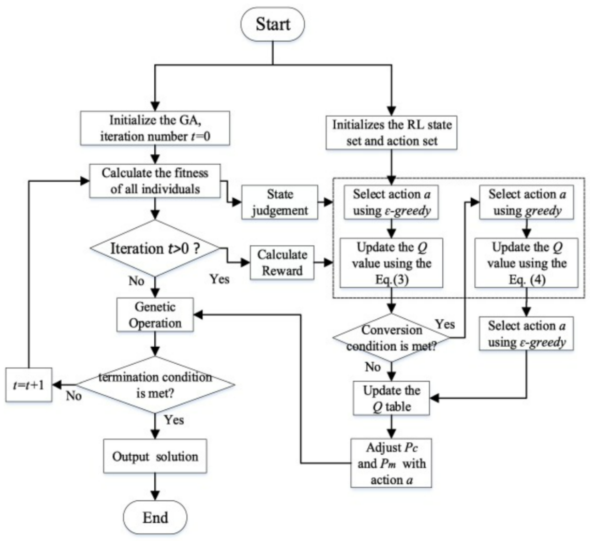

Start
Initialize the GA, iteration number t=0
Initializes the RL state set and action set
Calculate the fitness of all individuals
State judgement
Select action a using ε-greedy
Select action a using greedy
Iteration t>0 ?
Calculate Reward
Update the Q value using the Eq.(3)
Update the Q value using the Eq. (4)
No
Yes
Genetic Operation
Conversion condition is met?
Yes
Select action a using ε-greedy
No
t=t+1
termination condition is met?
No
Update the Q table
Yes
Output solution
Adjust Pc and Pm with action a
End

Figure 3.6: SLGA flowchart, adopted from (R. Chen et al., 2020)

These design choices are then combined with RL in the following way: rather than setting a fixed probability of mutation and crossover (Pm and Pc), the values are set intelligently according to the current population, previous and future states (see Figure 3.7). From an RL point of view, the state of the GA is considered the state s_t of the environment, and the actions a_t are adjusting and updating (Pm and Pc). The GA then takes these values to compute the new state s_{t+1} and emits rewards to the agent. Meanwhile, valuation calculation is performed with the value function, and the Q-value is updated in the Q-table using the value and emitted reward. The state (i.e. population of a GA) is valued with three factors: (1) average fitness of the population, (2) population diversity and (3) fitness of the best individual. Regarding rewards, there is a reward for changing crossover (change in max. fitness) and a reward for changing the mutation probability (change in average fitness). In experimental results, the authors find that SLGA has a good performance on small-scale problems, even better than traditional GA. Time consumption is reduced by 10%, and maximum and average RPD decrease by 20 and 30% respectively.

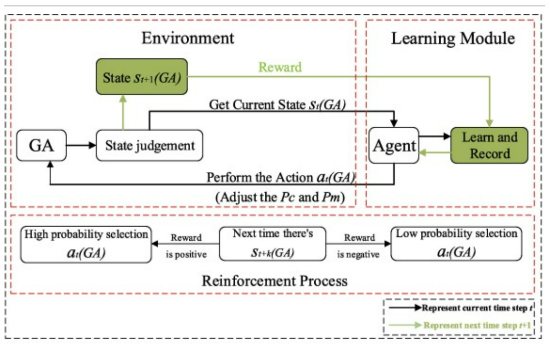

Environment
Learning Module
State $S_{t+1}(GA)$
Reward
Get Current State $S_t(GA)$
GA
State judgement
Agent
Learn and Record
Perform the Action $a_t(GA)$
(Adjust the Pc and Pm)
High probability selection $a_t(GA)$
Reward is positive
Next time there's $S_{t+k}(GA)$
Reward is negative
Low probability selection $a_t(GA)$
Reinforcement Process
Represent current time step t
Represent next time step t+1

Figure 3.7: Reinforcement process within GA, adopted from (R. Chen et al., 2020)

Visutarrom et al. (2020) on the other hand, combine RL and differential evolution (DE) to propose RL-DE. Again, RL is used to set the parameters cross-over rate (CR) and scaling factor (F). This is conducted in a similar fashion as was done in R. Chen et al. (2020); In every step of the differential evolution algorithm, the agent can select a sub-range the parameters CR and F can value in. The authors conclude that RLDE shows competitive results.

To summarize this Section, we conclude that Dai et al. (2019) and Rooyani and Defersha (2019) tested their algorithm on the largest instances of size 80x50 and 140x80 (jobs times machines) respectively. All other listed algorithms aren't tested on instances with (near) similar sizes. We have also seen that for single objective optimization (usually makespan) local search and genetic algorithms achieve the highest results. For MOO, on the other hand, genetic algorithms are mainly used. We have also seen that extra constraints are rarely taken into account when investigating the FJSP. It is mainly Rooyani and Defersha (2019) that have shown the generalizability of their 2SGA. We have seen that end-to-end DRL approaches work well in case a solution needs to be obtained very fast (a couple of seconds of inference time). However, models need to be trained beforehand in this case. Self-learning algorithms on the other hand have also proven to be useful in order to speed up optimization, here also requiring training time to learn an agent how to set the search parameters (such as crossover rate in genetic algorithms). However, an overview of how well the solutions tend to generalize to unseen environments is missing. When should each of these solutions be used? In the work of Song et al. (2022), generalization is touched upon slightly, but this could use much more exploration. Furthermore, these approaches take the classic FJSP scenario, whereas industry problems tend to have significantly more constraints than currently considered.

3.5 Conclusion

In this Chapter, we have reviewed the state-of-the-art literature regarding FJSP solutions. More specifically, we have seen the various solution directions, among which are mathematical approaches, heuristic solutions, meta-heuristics and AI-based solutions.

To conclude, we have seen that in order to solve the FJSP efficiently, the problem should not be approached mathematically. More specifically, we have seen that AI-based solutions offer the most promising results considering their results in terms of computation time and objective value. What is missing in the literature however is knowledge of when to select which solution approach. For example, in case job instances characteristics are volatile, would an SLGA or DRL approach perform better? Which of the two approaches is more generalizable? Which of the two algorithms would perform better when the instance scales up to industry-size datasets? Or should we then simply rely on heuristics? Which approach should one take if fast schedule computation is required?

Furthermore, what is also missing is the performance of the algorithms in case additional constraints are added. Which algorithm is most flexible to adapt and learn the behaviour of new constraints? Can we easily adapt existing models? Or do we have to retrain in order for performance to be acceptable? Similar to how some of these questions are addressed for heuristics in the 2SGA work of Rooyani and Defersha (2019), they should be addressed for the AI-based approaches.

Identifying which algorithms are most modular is an important step in order to push towards more generalizable job scheduling. These questions are the basis of the research questions defined in this work, stated in Section 1.2.

Chapter 4

Solution Methods

In this chapter, the proposed solution method will be outlined. These solutions methods are designed to create a schedule of jobs, which optimizes for the objectives discussed in Chapter 2. Furthermore, the solution methods are selected based on flexibility, efficiency, and robustness. The chapter covers the two proposed algorithms, the self-learning effective genetic algorithm and the end-to-end deep reinforcement learning algorithm. Both algorithms are designed to tackle the specific FJSP with sequence-dependent setup times, release dates and night times. The first algorithm is a reinforcement learning-based heuristic that combines the work of G. Zhang et al. (2011) and R. Chen et al. (2020), which can be customized to operate in a single- (SO-SLEGA) and multi-objective (MO-SLEGA) setting. For benchmarking purposes, the SLEGA will also be tested without the learning module. In this case, we refer to the algorithm as SO-EGA and MO-EGA. The SO-EGA is following the exact implementation of G. Zhang et al. (2011), where the MO-EGA is slightly different as described in Section 4.1.3. The second proposed algorithm builds on the work of Song et al. (2022) to tackle this specific FJSP using E2E-DRL in a single-objective setting which can be deployed using a greedy (DRL-G) or sampling (DRL-G) action strategy.

4.1 Self-Learning Effective Genetic Algorithm

In this section, we outline our first solution method, the self-learning effective genetic algorithm (SLEGA). First, we describe the complete framework, after which we dive into individual components of the framework. Finally, we describe how the algorithm is trained.

Figure 4.1 below illustrates the SLEGA framework. As can be seen, it consists of an initialization, a decoder (i.e., the evaluation function), a genetic algorithm, and a deep reinforcement learning (DRL) agent. First, the initial population is created. Each individual is then evaluated. This evaluated population is then fed into a DRL agent, which determines the parameters of the genetic algorithm based on the current state of the optimization process. After the parameters are set, one iteration of the genetic algorithm is executed, and the new offspring population is sent back to the decoder for evaluation. Once the termination criteria are met, the final population and hall of fame (i.e., best-found individuals) are returned. The pseudocode for this algorithm is given at the end of Section 4.1.

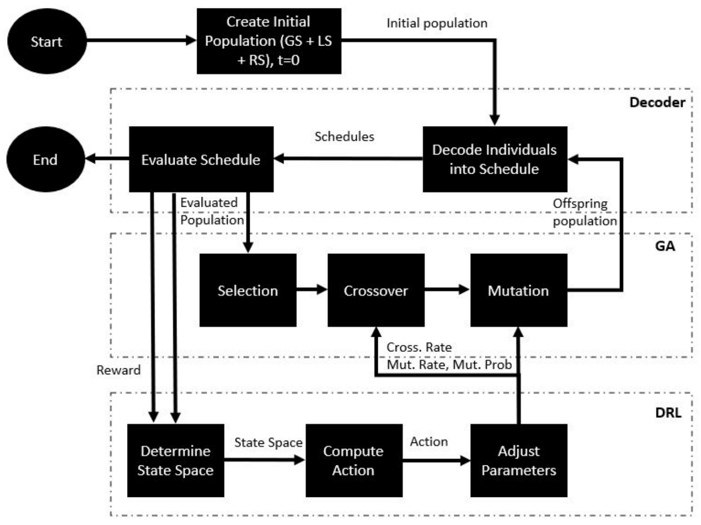


Figure 4.1: SLEGA framework

The advantages of including a self-learning module in a genetic algorithm are as follows. As the agent learns to make effective decisions on what value to set the mutation and crossover rates to, the search space is traversed much more efficiently. More specifically, the SLEGA is able to converge faster to an optimal solution, and the solution can even be better than the solution without this learning module.

Throughout the optimization process, schedules are encoded using the double-layer encoding method as seen in the literature (Liang et al., 2021). This encoding represents a complete schedule as decisions made in flexible job-shop scheduling can be divided into two categories; the operation sequence, and the machine allocation.

Figure 4.2 displays an example encoding of a schedule. This schedule consists of three input jobs, J_1, J_2 and J_3. J_1 has two operations $O_{1,1}$ and $O_{1,2}$, whereas J_2 and J_3 only have one operation each; $O_{2,1}$ and $O_{3,1}$ respectively. A feasible schedule can be represented as given in the encoding below. In the encoding, the operation sequence is given by $[J_1, J_2, J_1, J_3]$ and the machine allocation is given by $[M_1, M_3, M_1, M_2]$. The operation sequence determines the order in which operations should be scheduled. In this case, this is $[O_1, O_3, O_2, O_4]$, which is based on the occurrence of the job number. The machine allocation determines on which an operation is scheduled. This layer of the encoding is given in order of the operations of the FJSP instance, this thus means that operation (1,1) is scheduled on machine 1, operation (1,2) on machine 3, operation (2,1) on machine 1 and operation (3,1) on machine 2.

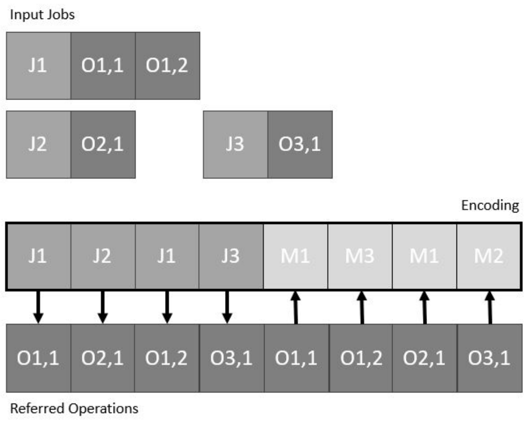


Figure 4.2: Example encoding format of a job schedule.

4.1.1 Population Initialization

The initialization of the population is a crucial task, as the initial population determines the quality and convergence speed of the algorithm (Rahnamayan et al., 2007). In order to do so, we follow the approach proposed by G. Zhang et al. (G. Zhang et al., 2011). More specifically, to initialize our population, we combine global selection (GS) (60%), local selection (LS) (30%) and random selection (RS) (10%). How each initialization algorithm generates machine allocation and operation sequence strings is explained below.

Global selection is focused on balancing the load across all machines. More specifically, an array of size $|M|$ is initialized. Then, random jobs (J) are selected and their operations (O_{ij}) are scheduled on the machine with the least load so far. After scheduling an operation, the duration is added to the total load of the selected machine. The pseudo-code for this algorithm is given in Algorithm 1.

Local selection is similar to the global selection, except for one important detail. Rather than optimizing machine load across all jobs, in the local selection, the machine load is optimized per job specifically. The pseudo-code for this variant is given in Algorithm 2. Note the initialization of the loaded array.

Finally, random selection is pretty straightforward. Both the machine allocation string and operation sequence are randomly initialized.

Algorithm 1 Global Selection

```
    Input
    Job Instance Information ($I$)

    Output
    Machine allocation ($allocation$)
    Operation sequence ($sequence$)

1: procedure GlobalSelection($I$)
2:     $J \leftarrow random.sample(J, |J|)$
3:     $load \leftarrow []$
4:     $sequence \leftarrow []$
5:     $allocation \leftarrow []$
6:     for $J_i \in J$ do
7:         for $O_{ij} \in J_i$ do
8:             $tempLoad \leftarrow []$
9:             for $M_k \in opeMas(O_{ij})$ do                    ▷ for all eligible machines
10:                $tempLoad[M_k] \leftarrow load[M_k] + duration(O_{ij}, M_k)$     ▷ load if schedule Oij on Mk.
11:            end for
12:            $M^* \leftarrow argMin(tempLoad)$                  ▷ determine machine with least load
13:            $load[M^*] \leftarrow load[M^*] + duration(o, M^*)$     ▷ add load to machine
14:            $allocation[O_{ij}] \leftarrow M^*$                ▷ allocate machine to operation
15:            $sequence[O_{ij}] \leftarrow J_i$                  ▷ add operation to sequence
16:        end for
17:    end for
18: end procedure
```

4.1.2 Schedule Decoding and Evaluation

Now that our population is initialized, we proceed to evaluate the population. The evaluation environment used here is Python 3, version 3.9. This evaluation environment is selected as it allows for parallel evaluation of different schedules. This will be used in the experiments later, to test the scalability of the execution times of the proposed algorithms.

Since our population consists of encoded individuals, this evaluation function is also referred to as the "decoder". Decoding individuals basically means that for each operation a start and completion time are allocated. The main scheduling algorithm is illustrated in Algorithm 3 and consists of three subroutines, CheckBackFill, NightPush and CalcSDST. Note that in all algorithms descriptions, very specific implementation details (e.g., start-of-day timestamp configuration) are omitted. Besides the operation sequence and machine allocation, the algorithm requires job instance information (I) as input as well.

Algorithm 2 Local Selection

```
    Input
    Job Instance Information (I)

    Output
    Machine allocation ($allocation$)
    Operation sequence ($sequence$)

1: procedure LocalSelection($I$)
2:     $J \leftarrow random.sample(J, |J|)$
3:     $sequence \leftarrow []$
4:     $allocation \leftarrow []$
5:     for $J_i \in J$ do
6:         $load \leftarrow []$                                          ▷ reinitialize load array every job
7:         for $O_{ij} \in J_i$ do
8:             $tempLoad \leftarrow []$
9:             for $M_k \in opeMas(O_{ij})$ do                           ▷ for all eligible machines
10:                $tempLoad[M_k] \leftarrow load[M_k] + duration(O_{ij}, M_k)$    ▷ load if schedule O on M.
11:            end for
12:            $M^* \leftarrow argMin(tempLoad)$                         ▷ determine machine with least load
13:            $load[M^*] \leftarrow load[M^*] + duration(O_{ij}, M^*)$        ▷ add load to machine
14:            $allocation[O_{ij}] \leftarrow M^*$                       ▷ allocate machine to operation
15:            $sequence[O_{ij}] \leftarrow J_i$                         ▷ add operation to sequence
16:        end for
17:    end for
18: end procedure
```

This dictionary contains all the required information for scheduling the operations. For example the required resources, sequence-dependent setup times, and release dates.

Subroutine CheckBackFill looks at all pairs of subsequent operations A and B which have already been scheduled, and checks whether it is feasible to schedule the current operation C at the end of operation A, or before the start of operation B. Note that in algorithmic implementation, we guarantee the correctness of the schedule in case operations are scheduled through backfilling. This is required as backfilling operations adjusts the required setups before and after the scheduled operation.

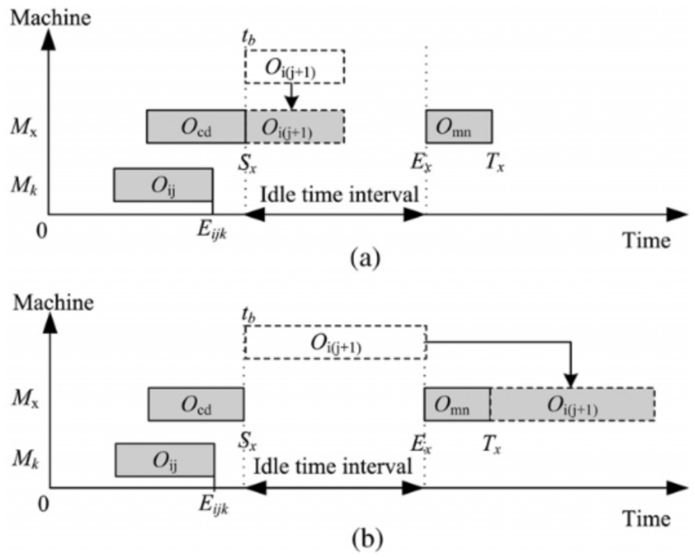


Figure 4.3: Backfilling example, adopted from G. Zhang et al. (2011)

Algorithm 3 Schedule decoding algorithm

```
Input:
Operation Ordering Encoding (O)
Operation Machine Allocation (M)
Job Instance Information (I)

1: procedure ScheduleOperations($O, M, I$)
2:     for $O_{ij} \in O$ do
3:         $M_k \leftarrow M(O_{ij})$    ▷ Extract machine from operation machine allocation.
4:         $a \leftarrow max(O_{ij}.preConstrSatis, O_{ij}.releaseDate)$    ▷ Check min. start time of operation
5:         $b \leftarrow max(M_k.timeState, a)$    ▷ Align start with machine
6:         $c \leftarrow CheckBackFill(O_{ij}, M_k, a, b)$    ▷ Final start time, potentially backfilled
7:         if $c$ is feasible then    ▷ Starts during day, only end during night without operator
8:             $O_{ij}$.startTime ← c
9:         else
10:            $O_{ij}$.startTime ← NightPush($O_{ij}, M_k$)    ▷ Move start time to start of next day
11:        end if
12:        $O_{ij}$.completionTime ← $c + O_{ij}.processingTime +$ CalcSDST($O_{ij}, M_k$)
13:        $M_k.timeState \leftarrow O_{ij}.completionTime$
14:    end for
15: end procedure
```

Subroutine NightPush sets the start time of a schedule to the start of the next day. Subroutine CalcSDST takes the current state of the machine and the operation. Using that information in determines the required set-ups that the machine has to go through in order to start the operation.

When a schedule is decoded and timestamps are allocated to all individuals, the schedule can be replayed in order to determine the makespan and cost of operations. This is conducted through simulation, resulting in the objective values defined in Chapter 2.

4.1.3 Genetic Algorithm

In this section, we outline the effective genetic algorithm (G. Zhang et al., 2011) that is part of the SLEGA framework shown in Figure 4.1. More specifically, here we address the genetic operations and the parallelization framework used within the genetic algorithm. The population initialization is already defined in Section 4.1.1.

The genetic algorithm has the following procedure. First, a population is initialized with a given size. This population is initially evaluated given the evaluation function above, i.e., we are minimizing the makespan and cost of the schedule. This is considered as the fitness of each individual. Then, the population enters a generational process. The most promising individuals are filtered with a *selection* function for a set number of generations. These selected individuals are then mated using a *crossover* function. Finally, *mutation* is applied to improve exploration. Note that with our double-layer encoding, we have different mutation and crossover functions for the operation sequence and machine allocation string. The selection operator remains the same. Throughout the evolutionary process, individuals evolve using different genetic operators. The three types of genetic operators are *selection*, *crossover*, and *mutation.*

Selection

Selection is the first genetic operation. In our implemented algorithm we use tournament selection with a tournament size of 3. This is similar to the implementation of G. Zhang et al. (2011). Note that tournament selection can only take into account a single objective, in this case, the makespan is used to select the individuals.

In order to deal with selection in a multi-objective setting, we deploy NSGA-II from Deb et al. (2002). In the literature review, we have seen that this was the first multi-objective genetic algorithm that can handle constraints and elitism. Since it is still very relevant today, we use it here.

Crossover

Next, two individuals are taken and mated as explained below. As mentioned before, this process is different for the machine allocation and operation sequence string.

The operation sequence string is crossed over using Precedence preserving order-based crossover (POX) Lee et al. (1998). POX works as such. (1) Two sub-job sets Js1 and js2 are generated from all jobs and two parent individuals p1 and p2 are randomly selected. (2) Any gene in p1/p2 that belong to Js1/Js2 are copied into child individuals c1/c2, and retain in the same position. (3) All genes that are already in sub-job Js1/Js2 are deleted from p2/p1. (4) The empty positions in c1/c2 are orderly filled with genes of p2/p1 that belong to their previous sequence. This approach is visualized in Figure 4.4 below.

The machine allocation string is crossed over using both two-point crossover and uniform crossover. For two-point crossover, two random points in the individual are selected. Then, individuals are generated by combining the genes between the two points of individual 1, with the genes outside the two points of individual 2, and vice-versa. In uniform crossover, every gene of the new individual has an equal probability of being selected from individual 1 or 2. These operators are further illustrated in Figure 4.5.

No changes are made to crossover for the multi-objective setting. Even though these crossover functions are makespan focused, they offer sufficient exploration capabilities.

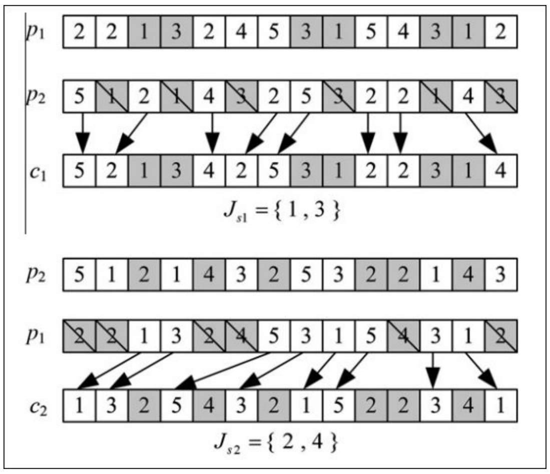


Figure 4.4: Operation sequence crossover, adopted from G. Zhang et al. (2011)

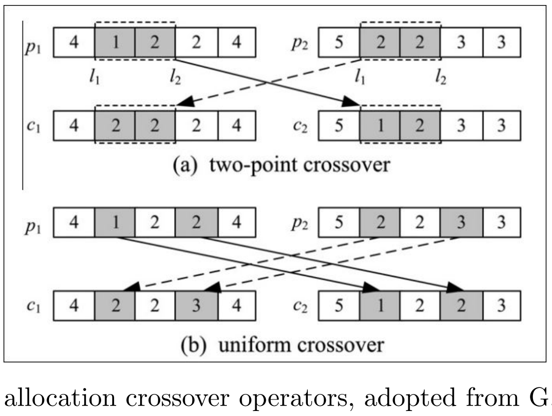


Figure 4.5: Machine allocation crossover operators, adopted from G. Zhang et al. (2011)

Mutation

The mutation operators for the single-objective setting are quite straightforward. For the machine allocation string, we select the machine which for which the operation-machine pair has the lowest operating time in case of mutation. In the case of sequence-dependent setup times, the required setup between the current operation of the machine and eligible operations is added to the operating time before the operation-machine pair is picked. For the operation sequence string, we generate a new index $0 \leq i \leq |OS|$, and swap the gene under consideration with the gene in the generated index.

For mutation, we add four additional mutation functions to deal with the multi-objective setting. Two for the operations sequence string, and two for the machine allocation string. These operations are given in Algorithm 4, and are the following four. (1) greedy deadline mutation: swap operation that misses the deadline to the start of the operation sequence. (2 and 3) greedy WIP mutation: operations from the same job are scheduled sequentially and on the same machine (4) greedy cost of addition mutation: an operation is allocated on the machine on which the operation with the most similar required resources is allocated. The similarity is determined through cosine similarity, which is given in Equation 4.1 below. In this Equation, A and B are boolean vectors containing whether each item (i.e., the indices) is required by operations a and b respectively.

$$S_C(A,B) := \cos(\theta) = \frac{A \cdot B}{||A||||B||} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}} \tag{4.1}$$

Algorithm 4 MO-SLEGA Mutation

```
    Input
    Job Instance Information ($I$)
    Machine allocation ($allocation$)
    Operation sequence ($sequence$)
    Individual mutation probability ($indpb$)

    Output
    Machine allocation ($allocation$)
    Operation sequence ($sequence$)

1:  procedure MUTATION($I$)
2:      $p1 \leftarrow random.random()$
3:      $p2 \leftarrow random.random()$
4:      if $p1 < \frac{1}{3}$ then
5:          sequence ← WIPMutOS($sequence$, $I$, $indpb$)      ▷ schedule operations from same job sequentially.
6:      else if $p1 < \frac{2}{3}$ then
7:          sequence ← SwapMut($sequence$, $I$, $indpb$)      ▷ swap order of two operations.
8:      else
9:          sequence ← DeadlineMut($sequence$, $I$, $indpb$)      ▷ move operation that missed deadline to the front.
10:     end if
11:     if $p2 < \frac{1}{3}$ then
12:         allocation ← WIPMutMS($allocation$, $I$, $indpb$)      ▷ schedule operation on the same machine as another operation of the same job.
13:     else if $p2 < \frac{2}{3}$ then
14:         allocation ← ProcTimeMut($allocation$, $I$, $indpb$)      ▷ schedule operation on machine with lowest processing time.
15:     else
16:         allocation ← AddMut($allocation$, $I$, $indpb$)      ▷ schedule operation on machine that processes most similar operation.
17:     end if
18: end procedure
```

Figure 4.6 below displays an example Pareto front which is obtained after a single run of the MO-GA. On the Pareto front, there are 9 points. From these 9 points, the company would have to select the schedule seen as most fit with regards to the makespan/cost trade-off. That schedule can then be executed in practice.

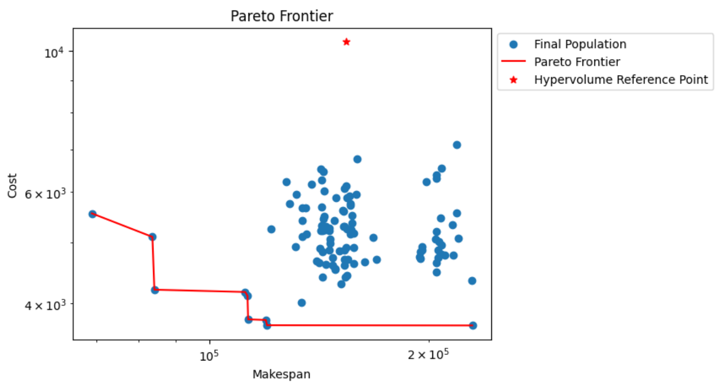


Figure 4.6: Example Pareto front of company instance.

Parallelization

In order to speed up the aforementioned algorithm, we make use of parallelization. Parallelization is possible for genetic algorithms as the fitness evaluations are independent, and can be executed at the same time. This allows us to take advantage of the multiple cores that computational machines have. Parallelization is realized by the use of a multiprocessing pool in Python, which is easily integrated with the genetic algorithm framework. Figure 4.7 below displays the speed of running the genetic algorithm for 100 generations. As can be seen from the figure, when multiprocessing is enabled, the algorithm completes much faster. At first, the algorithm lacks behind as in the first 10 seconds the algorithm is preparing the parallel workers (i.e., little overhead), shuffling around data where necessary. Then once the workers are initialized, we can see that the algorithm with parallelization completes 100 generations more than twice as fast (47 seconds or 101 seconds respectively).

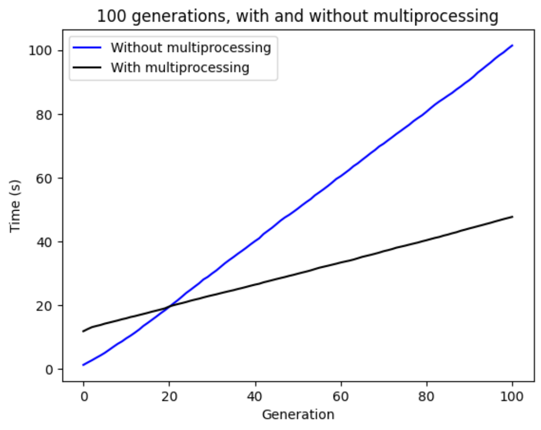


Figure 4.7: Advantage of multiprocessing

4.1.4 Self-Learning Module

This section describes the third and final component of the proposed SLEGA, the self-learning module. For this component, there are a couple of differences between the single-objective and multi-objective settings. These differences are highlighted below. Note that the SLEGA is referred to as SO-SLEGA in the single-objective setting and MO-SLEGA in the multi-objective setting during experiments.

To allow for parameter control by a deep reinforcement learning agent, we define the following Markov decision process for the SO-SLEGA. In each generation, an agent is given the following state space (S_t):

- Normalized mean fitness of the current population.
- Normalized the best fitness of the current population.
- Normalized standard deviation of fitness of current population
- Normalized remaining budget.

- Normalized stagnation count.

The fitness (f) values are normalized by scaling the fitness values on the range of worst (f_{max}) and best-seen fitness (f_{min}) values so far. More specifically:

$$\hat{f} = (f - f_{min})/(f_{max} - f_{min}) \tag{4.2}$$

Following this formula, a value of 0 would indicate optimal fitness and a value of 1 would indicate worst fitness.

Moreover, the budget (i.e., number of generations left, b) and stagnation count (s_c) are normalized by dividing the value by the total number of generations (n_gen). Equations for the normalized stagnation count and budget are given in Equations 4.3 and 4.4 respectively.

$$\hat{b} = b/n\_gen \tag{4.3}$$

$$\hat{s_c} = s_c/n\_gen \tag{4.4}$$

The normalized standard deviation is clipped to 1, to reduce to impact of extreme observations. All other values within the state space are clipped to 1 by definition as well.

The action space of the agent then consists of setting (1) the mutation probability (probability each individual mutates), (2) the crossover probability, and (3) the mutation rate (probability each gene mutates). More specifically, the agent can set these values between 0 and 1. This allows for radical changes from one generation to the other.

Based on the selected probabilities (A_t), the algorithm then executes the next set of genetic operations. If the makespan (T_c or f) of the output population has increased, the agent is rewarded with the absolute increase of this makespan. If the makespan has decreased, the agent is punished accordingly.

$$R_t = T_{c_{t-1}} - T_{c_t} \tag{4.5}$$

This reward (R_t) is sent back together with the state space (S_{t+1}) based on the freshly created population (Pt).

The Markov-decision process for the MO-SLEGA is slightly different and formulated as such. Note that these changes are only implemented in Experiment 4. In each generation, we now have the following state space (S_t):

- Normalized remaining budget.
- Normalized stagnation count.
- Average normalized best fitness of the current population.
- Average normalized mean fitness of the current population.
- Average normalized standard deviation of fitness of the current population.
- Normalized hypervolume indicator.
- Normalized Pareto size.

The normalized remaining budget and stagnation count are similar to before. For the fitness values, we now average over the two objective values as given in Equation 4.6 below.

$$\hat{f} = \frac{(f^1 \_ f^1_{min})/(f^1_{max} - f^1_{min}) + (f^2 - f^2_{min})/(f^2_{max} - f^2_{min})}{2} \tag{4.6}$$

In this case, f^1 refers to the makespan and f^2 refers to the cost of a schedule. Note that the hypervolume indicator and Pareto size are new features, to represent the quality of the found Pareto front. The hypervolume indicator (Equation 3.1) is the most used set-quality indicator for the assessment of multi-objective optimizers where the actual Pareto front is unknown (Guerreiro et al., 2021). In order to normalize the hypervolume and objective values, we keep track of the best value that was found and divide the value of the current value by the best value of that objective. In order to normalize the Pareto set (PS), we divide the size of the Pareto set by the population size (Equation 4.8). We do so as the actual Pareto set is unknown. This will ensure that all features are between 0 and 1.

$$\hat{hv} = (hv - hv_{min})/(hv_{max} - hv_{min}) \tag{4.7}$$

$$\hat{ps} = \frac{|PS|}{n\_pop} \tag{4.8}$$

The action space is the same as in the setting before. However, the reward is now given as such (Equation 4.9). For each objective value that has improved, a reward of 1 is received (usually called binary objective value increase). For example, if makespan has improved, a reward of 1 is allocated. If the cost also improved, another reward of 1 is allocated. The total reward thus equals 2. Similarly, in case of a decrease in objective value, a penalty of 1 is also allocated. We change to this approach as both objectives are of equal importance, meaning that increases and decreases should be rewarded on the same scale.

$$Rt = \mathbb{1}\{f_t^1 < f_{t-1}^1\} - \mathbb{1}\{f_t^1 > f_{t-1}^1\} + \mathbb{1}\{f_t^2 < f_{t-1}^2\} - \mathbb{1}\{f_t^2 > f_{t-1}^2\} \tag{4.9}$$

Now that all components of the SLEGA are outlined, we proceed to give the full pseudocode in Algorithm 5 below. Note that *selection* , *crossover*, *getstate reward* refer to different subroutines for the SO and MO setting. The differences of these procedures are outlined in the previous paragraphs.

Algorithm 5 Self-Learning Effective Genetic Algorithm

Input
Job Instance Information (I)
DRL policy (π)
Population size (n_pop)
Number of generations (n_gen)

Output
Pareto Front (hof)

```
1:  procedure SLEGA($I$)
2:      $pop \leftarrow []$
3:      $hof \leftarrow []$
4:      for $x \in range(n\_pop)$ do                         ▷ Generate population
5:          $rnd \leftarrow random(0, 1)$                     ▷ Random value between 0 and 1
6:          if $rnd < 0.6$ then
7:              $Indv \leftarrow GlobalSelection$
8:          else if $rnd < 0.9$ then
9:              $Indv \leftarrow LocalSelection$
10:         else
11:             $Indv \leftarrow RandomSelection$
12:         end if
13:         $pop[x] \leftarrow Indv$
14:     end for
15:     for $gen \in range(n\_gen)$ do
16:         $pop \leftarrow evaluate(pop)$                    ▷ Evaluate population
17:         $S \leftarrow getstate(pop)$                      ▷ Determine state space
18:         $A \leftarrow \pi(S)$                             ▷ Get actions from agent
19:         $ofsp \leftarrow selection(pop)$                  ▷ Compute offspring
20:         $ofsp \leftarrow crossover(ofsp)$
21:         $new\_pop \leftarrow mutation(ofsp)$              ▷ Compute new population
22:         $R \leftarrow reward(new\_pop, pop)$              ▷ Compute reward (only for training).
23:         $pop \leftarrow new\_pop$                         ▷ Update population.
24:         $hof \leftarrow update(pop)$                      ▷ Update pareto front.
25:     end for
26: end procedure
```

4.1.5 Training

The DRL agent is trained using proximal policy optimization (PPO). PPO is a policy-based reinforcement learning algorithm that was first introduced by Schulman et al. (2017). The original motivation for selecting this algorithm was the ease of implementation. Furthermore, this agent is also utilized in our E2E-DRL solution, as described in Section 4.2. Selecting a single training policy makes for a more equitable benchmark when experimenting. RL classically suffers from the fact that it's training data is generated by the agent on-the-fly, rather than relying own a static training set. As a result, the parameters of the neural network are updated based off the data gathered with an not updated neural network. trust region policy optimization (TRPO) is the first algorithm that attempts to overcome this problem (Schulman et al., 2015). TRPO applies a surrogate objective function for updating the policy network subject to a constraint on the size of the policy network, known as the Kullback-Leibler constant. This idea ensures that the jump in old and updated policy is limited, enabling stability in the learning process.

PPO essentially is a simplified version of TRPO and PPO exists in two main forms, PPO penalty, and PPO clip. In this research on scheduling, clipped PPO is used as this is considered the most prevalent. Clipped PPO simply restricts the range in which the policy can change. More specifically, the clipped PPO loss is given in Equation 4.10 below. The clipped loss is then used to update the policy as given in Equation 4.11.

$$\mathcal{L}_{\theta_k}^{CLIP}(\theta) = E_{\tau \sim \pi_k}[\sum_{t=0}^{T}[min(r_t(\theta)\hat{A}_t^{\pi_k}, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t^{\pi_k})] \tag{4.10}$$

where $\hat{A}_t^{\pi_k}$ indicates the estimated advantages for of a policy π_k at time step t. Within our SLEGA, this is interpreted as the advantage of selecting specific values for the crossover probability, mutation rate, and mutation probability compared to the average value obtained when using different settings for these parameters. $[1-\epsilon, 1+\epsilon]$ indicates the clipping range, $r_t(\theta)$ represents the reward obtained at a particular time step t when using the hyperparameters θ selected in the genetic algorithm. $r_t(\theta)$ contains the clipped rewards obtained by the agent through setting the genetic operation parameters. The clipped loss then is a function of the advantages and rewards, computed by tacking the expected value over multiple trajectories, i.e., one or more optimization runs of the genetic algorithm.

$$\theta_{k+1} = argmax_{\theta}\mathcal{L}_{\theta_k}^{CLIP}(\theta) \tag{4.11}$$

The policy is updated in an actor-critic setting, where we have N actor networks and a critic network which all share the same network architecture. The actors run the old policy $\pi_{\theta_{old}}$ in an environment for T timesteps. Within the SLEGA, this means that multiple genetic algorithms are executed before the policy θ is updated. After the T timesteps, $\mathcal{L}_{\theta_k}^{CLIP}(\theta)$ is optimized for K epochs given a mini-batch size $M \leq N \times T$. This will result in the new policy which is copied back to the actors.

In order to embed state information, the agent makes use of a standard multi-layer perceptron (MLP) policy from stable-baselines3. This policy and value network both consist of two linear layers. The first layer has 5 or 7 input features (i.e., our defined state space) and 64 output features. The second layer has 64 in- and out features. Both layers have a Tanh activation function. This activation function is given in Equation 4.12 below. Tanh became preferred over the Sigmoid activation function, as it gave better performance for multi-layer neural networks (Schulman et al., 2017). Before the output layer, the 64 hidden features represent the embedding of the current state of the optimization algorithm. To estimate the value and actions of a state, we employ an additional output layer with 64 input features and 1 output feature for the value network, and 3 output features for the action network. The value network here outputs the expected value of a state, which is used during training. The action network here outputs the crossover rate, mutation rate and mutation probability which are used in the genetic algorithm.

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{4.12}$$

To avoid overfitting, we validate the policy every 1000 time steps (i.e., 10 SLEGA runs) on a validation environment. The model with best performance on validation environment is then saved. Furthermore, The pseudocode for this training algorithm is given in Algorithm 6 below. The algorithm already has various hyperparameters filled in (e.g., validation every 1000 timesteps).

Algorithm 6 SLEGA training procedure

```
    **Input**
    MLP policy and genetic algorithm

 1: procedure TrainPPOModelSLEGA
 2:     best reward ← 0
 3:     while $t < \mathcal{T}$ do                          ▷ until total training time steps are reached
 4:         if $t$ mod 1000 == 0 then
 5:             sample new batch $\mathcal{B}$ of FJSP instances
 6:         end if
 7:         for $b \in \mathcal{B}$ do                              ▷ in parallel
 8:             initialize environment $\mathcal{E}$ based of $b$
 9:             for $iter \in \mathcal{I}$ do                       ▷ one run of SLEGA
10:                 compute mutation rate, crossover, and mutation probability from MLP policy
11:                 apply selection, crossover, and mutation
12:                 receive reward $r_t$ and next state $s_{t+1}$.
13:                 $s_t \leftarrow s_{t+1}$                        ▷ update state of environments
14:             end for
15:         end for
16:         compute PPO loss
17:         update policy
18:         $t \leftarrow t + |\mathcal{B}| \times \mathcal{I}$           ▷ update number of executed time steps
19:         if $t$ mod 1000 == 0 then
20:             valid reward ← validate policy
21:             if valid reward ≥ best reward then
22:                 save model
23:                 best reward ← valid reward
24:             end if
25:         end if
26:     end while
27: end procedure
```

4.2 End-to-End Deep Reinforcement Learning

For our second solution method, we make use of end-to-end deep reinforcement learning (E2E-DRL). More specifically, we adopt the implementation of Song et al. (2022) and make adjustments to tailor it specifically to our FJSP instances. These proposed adjustments introduce the novelty in our work. The section is divided in the following way. First we describe the general framework used to train the E2E-DRL solution. Then, we dive deeper into the state space, the GNN used for state embedding, and the action space. Finally, we describe the training procedure.

Figure 4.8 below displays the E2E-DRL framework. As can be seen, the scheduling state is formulated as heterogeneous graph. This graph is then embedded using a graph-neural network. After which a policy network consumes the embedding to action probabilities. Based on this action probability, actions can then either be sampled (DRL-S) or greedily selected (DRL-G).

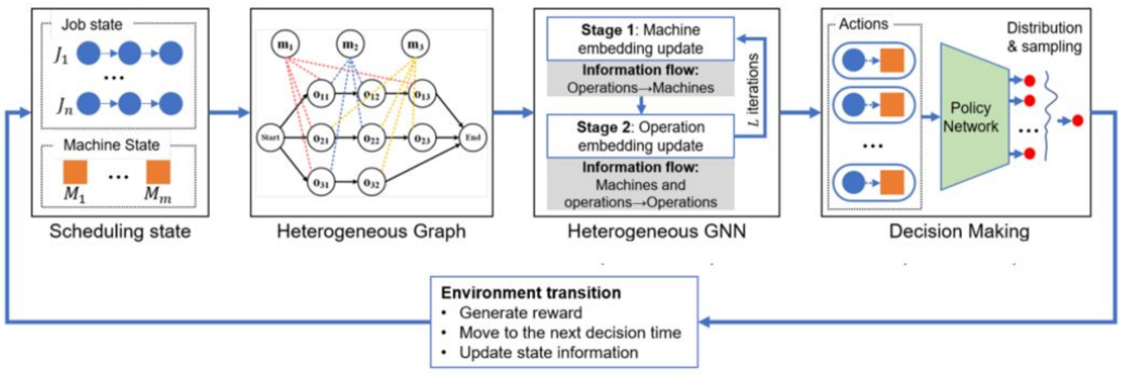


Figure 4.8: E2E-DRL Framework, adopted from Song et al. (2022)

The Markov decision process (MDP) used by the authors to model the FJSP is as such. The scheduling environment in the context of the Flexible Job-Shop Scheduling Problem (FJSP) operates based on discrete events. These events represent specific points in time when certain changes occur. For example, an event can occur when an operation is completed or when a machine becomes available. The environment keeps track of the current moment in time during the scheduling process. The moment at which an event occurs is also directly the new state of the environment. In this state, the agent decides which operation-machine (O-M) pair to execute, taking into account the current various features described below. The selected machine is then marked as unavailable for the duration of time it takes to process the chosen operation.

If there are no eligible O-M pairs available for selection at a given state, the environment progresses to the next point in time until a new O-M pair becomes available. This iterative process continues until all operations are successfully scheduled. The makespan, which represents the total duration of the scheduling process, is determined by identifying the first moment when all machines are available, and all operations have been executed.

4.2.1 State space

To spice things up, the state space of the FJSP environment is represented as heterogeneous graph (HG). Such representation is given in Figure 4.9 below. The graph is defined as such $\mathcal{H} = (\mathcal{O}, \mathcal{M}, \mathcal{C}, \mathcal{E}_t)$. The machine nodes ($\mathcal{M}$) are given on the top side of the graph, and the operation nodes ($\mathcal{O}$) are connected from left to right to represent a job. The graph also has dummy start and end nodes. For example, job 1 has three operations, o_{11}, o_{12}, and o_{13}, these are connected from left to right to indicate the order in the operations are to be processed. The directional arcs ($\mathcal{C}$) between these nodes thus indicate the precedence constraints exciting between the operations. Furthermore, there exist bi-directional arcs ($\mathcal{E}_t$) between the machine and operations nodes. These arcs represent either possible combinations of O-M pairs (dotted lines) or scheduled O-M pairs (solid lines). This heterogeneous graph representation is used over the original FJSP disjunctive graph (C. Zhang et al., 2020) as (1) the graph density is significantly reduced, (2) information on O-M combinations can easily be represented and (3) information on machines can easily be represented. The bi-directional arc set $\mathcal{E}_t$ changes over time as operations are scheduled, hence the full graph representation $\mathcal{H}_t$ is also dependent on time t.

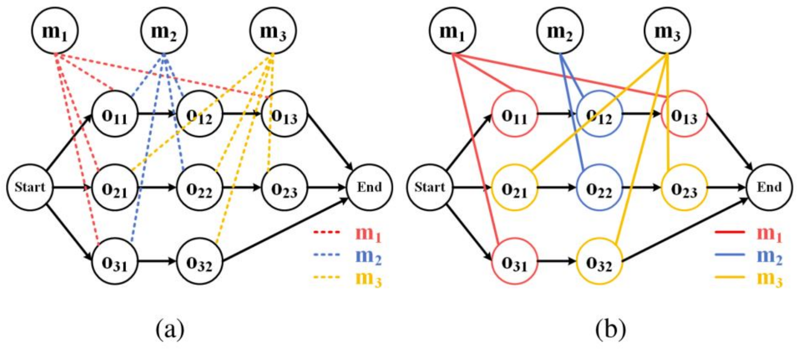


Figure 4.9: Heterogeneous Graph representation of FJSP. A dotted line means processable, while a solid line means scheduled. Adopted from Song et al. (2022)

In all of our experiments outlined in the next chapter, the nodes and arcs have the features outlined below at each timestep during scheduling. These features are either taken from the work of Song et al. (2022), or are custom-made extensions to tackle our specific problem at hand. Most features are dynamic and thus change over time, some features are static and are set during initialization. To compute some of the features, we need to track the current partial schedule of operations and the partial schedule of machines. To clarify computation of the features at state s_t, we define some new notation below.

For the partial operation schedule (SO) we track whether an operation O_{ij} is scheduled ($SO[0]_{ij}$), the allocated machine ($SO[1]_{ij}$), the calculated or actual start time ($SO[2]_{ij}$), the expected or actual end time ($SO[3]_{ij}$). For the partial machine schedule (SM), we track whether a machine Mk is idle ($SM[0]_k$), when the machine is next available ($SM[1]_k$), the amount of time a machine is operating ($SM[2]_k$) and the operation that is running ($SM[3]_k$). These variables are dynamic and dependent on state s_t.

Furthermore, static information at the start is collected from the FJSP environment. This static information consists of (1) the processing time p_{ijk} of scheduling a operation O_{ij} on machine M_k, (2) the eligibility el_{ijk} of scheduling an operation O_{ij} on machine M_k (binary indicator), (3) the precedence constraints $c_{iji'j'}$ that indicates whether an operation O_{ij} preceeds operation $O_{i'j'}$ (binary indicator), (4) reverse precedence constraint $c^r_{iji'j'}$ that indicates whether and operation O_{ij} is a successor of $O_{i'j'}$ (binary indicator), (5) oj_{ij} represents the job number of operation O_{ij}, (6) fo_j marking the first operation of job J_j, (7) no_j indicating the number of operations of job J_j, (8) the remaining number of operations r_ij after completing operation O_ij, (9) the sequence-dependent setup-times $s_{iji'j'}$ required between operation O_{ij} and operation $O_{i'j'}$. (10) no_{ij} indicating whether operation O_{ij} can be processed at night (binary indicator), (11) ns_{ij} indicating whether operation O_{ij} can be started at night (binary indicator), (12) the release date r_{ij} of operation O_{ij}. Given this information and the current state S_t at time unit t_t, we then can calculate the features of the nodes and arcs as such.

Operation nodes ($\forall O_{ij} \in \mathcal{O}$)

1. $I(O_{ij}, s_t)$: Binary indicator that equals 1 when an operation O_{ij} is scheduled in state St. This is directly equal to $SO[0]_{ij}$.
2. $NC(O_{ij})$: Number of connected machines. This attribute indicates how many possible machine options an operation O_{ij} has. This is set during initialization as such $\sum_{k\in M} el_{ijk}$.
3. $\hat{P}(O_{ij}, s_t)$: Average or actual processing time in state s_t. This attribute indicates the average processing time over the connected machines in case an operation is not scheduled. $\frac{1}{\sum_{k\in M} el_{ijk}} \sum_{k\in M} el_{ijk} \times p_{ijk}$. In case an operation is scheduled on any machine M_k, this feature represent the actual processing time p_{ijk}.
4. $NU(O_{ij}, s_t)$: The number of unscheduled jobs at state s_t. This is computed using $SO[0]_{ij}$ and oj_{ij}, by aggregating over the job instance J_j found in $SO[0]_{ij}$.

5. $\hat{S}(O_{ij}, s_t)$: Potential or actual start time. This is calculated by recursively computing start times for all preceding operation $O_{i'j'}$. This can be computed as such: $\hat{S}(O_{ij}, s_t) = \max[\hat{S}(O_{i'j'}, s_t) + \hat{P}(O_{ij}, s_t), r_ij]$. The precedence constraints are given in $c_{iji'j'}$.

6. $\hat{C}(O_{ij}, s_t)$: Expected or actual operation completion time. This is computed in the following way: $\hat{C}(O_{ij}, s_t) = \hat{S}(O_{ij}, s_t) + \hat{P}(O_{ij}, s_t)$.

7. $no(O_{ij})$: Binary indicator that equals one when an operation can be processed at night. This is static and comes directly from input information.

8. $ns(O_{ij})$: Binary indicator that equals one when an operation can be started at night. This is static and comes directly from input information.

9. $ttr(O_{ij}), s_t)$: Time until the release date of an operation is satisfied. Computed by taking subtracting the current time unit at state s_t from the release date of operation. $ttr(O_{ij}, s_t) = r_ij - tt_t$.

Machine nodes $(\forall M_k \in \mathcal{M})$

1. $NC(M_k)$: Number of connected operations. This attribute indicates how many possible operations a machine has. This is set during initialization as such $\sum_{O_{ij} \in O} el_{ijk}$.

2. $A(M_k, s_t)$: Available time of machine M_k at state s_t. The time unit at which the machine becomes available again. This is maintained in $SM[1]$.

3. $U(M_k, s_t)$: Utilization of machine M_k at in state s_t. This is calculated as such: $U(M_k, s_t) = SM[2]_k / tt_t$.

4. $TS(s_t)$: Time of scheduling, directly taken from tt_t.

5. $RTN(s_t)$: Remaining time until night. Calculated using: $\max[50400 - (tt_t \mod 86400), 0]$. Note that this variable is equal to 0 when it is already night, 86400 covers a full day in time units, and night starts at 50400 time units. Note that we emit some logic here in case the night is about to end.

O-M pairs $(\forall (O_{ij}, M_k) \in \mathcal{E}_t)$

1. $PT(O_{ij}, M_k)$: Processing time of operation O_{ij} on machine M_k, directly taken from p_{ijk}.

2. $ST(O_{ij}, M_k, s_t)$: Sequence-dependent setup time for scheduling operation O_{ij} on machine M_k at state s_t. Since this variable is dependent on the previous operation $O_i'j'$, we extract $O_i'j'$ from $SM[3]_k$. Then, $s_iji'j'$ indicates the required setup time for scheduling operation O_ij on machine M_k.

3. $TRN(O_{ij}, M_k, s_t)$: The number of time units an operation O_{ij} would be running at night on machine M_k at state s_t. Calculated by as such: $TRN(O_{ij}, M_k, s_t) = PT(O_{ij}, M_k) + ST(O_{ij}, M_k, s_t) - RTN(s_t)$

Compared to the original work of Song et al. 2022, seven new features were added. The motivation for the features is given below. For operation nodes, three features were added. The first added feature is **night schedule** ($no(O_{ij})$), which represents whether operations can be completed or executed during the night. The second added feature is **night setup**, which represents whether features can be started at night ($ns(O_{ij})$. These two features should allow the agent to learn that certain operations can be executed or started during the night, when no operators are available. This thus it should be scheduled then to minimize makespan. The last feature is **time to release** ($ttr(O_{ij}, s_t)$). Time until release allows agents to understand the behaviour of when important operations will become available.

For machine nodes, we added information on **current time of scheduling** ($TS(S_t)$), and the **remaining time until night** ($RTN(s_t)$). This should represent the logic required for understanding the time of day, in order to be able to decide whether an operation that can run during the night should be considered.

For O-M pairs, we have two additional features: **sequence-dependent setup times** ($ST(O_{ij}, M_k, s_T)$) and **time running at night** ($TRN(O_{ij}, M_k, s_t)$). The first feature shows the setup time incurred when an O-M pair is selected, whereas the second feature covers the time the operation would be running at night. These features are interesting for the agent to select the operation with the least setup time, and the most time running at night. All other features are directly taken from Song et al. (2022).

4.2.2 State Embedding

Now, in order to allow a DRL agent to consume this graph as information, we embed the graph using the HGNN proposed by Song et al. (2022). GNNs are size-agnostic, meaning that they can handle graphs of varying sizes. This allows us to deploy the trained network on FJSP instances of different sizes than the ones trained on. This property is achieved through their graph-based message-passing mechanism, where information for each node is extracted from the neighbours of a node. The HGNN consists of three parts, (1) machine node embedding, (2) operation node embedding and (3) stacking and pooling. Similar to the authors work, we do not use the pooling layer (denoted with L) in our experiments. Hence only the process of stacking is described in step 3 of the embedding.

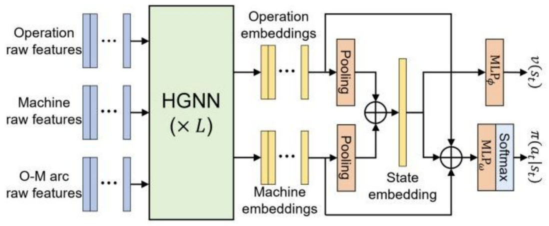


Figure 4.10: HGNN architecture, adopted from Song et al. (2022)

.

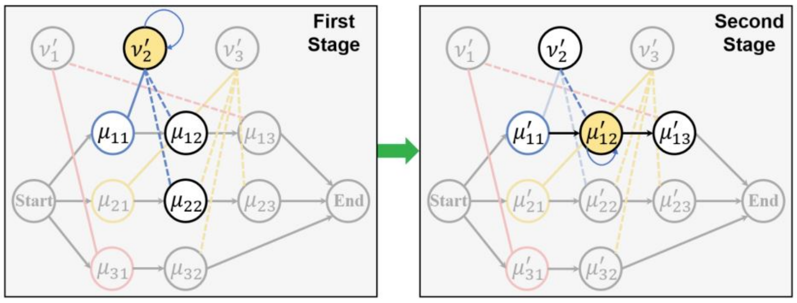


Figure 4.11: Two-stage embedding scheme of the heterogeneous graph, adopted from Song et al. (2022)

.

Like in the work of Song et al. (2022), we use the graph attention network for machine node embedding, as it able to learn the importance of different operation nodes. For example, operations expected to start sooner might be more important than operations expected to start later. Under the same logic, the network should also be able to learn that operations that run almost entirely during the night, are more critical than operations that would finish just after the start of the night. The message-passing step here is quite simple, a machine node aggregates information from only the first-order neighbourhood (only the eligible operations on that machine). More specifically, the Song et al. (2022) propose the following steps to embed the machine nodes M in the setting of a heterogeneous graph.

For a machine M_k, the attention coefficients e_{ijk} can be calculated as given in Equation 4.13. This attention coefficient represents the importance of each neighboring operation $O_{ij} \in N_t(M_k)$.

$$e_{ijk} = \text{LeakyReLU}\left(a^\top \left[W^M \nu_k || W^O \mu_{ijk}\right]\right) \tag{4.13}$$

where e_{ijk} represents the attention coefficient between machine M_k and operation O_{ij}, LeakyReLU denotes the Leaky Rectified Linear Unit activation function, $a\top$ represents the weight parameter, W^M and W^O denote the linear transformations that are used, ν_k represents the machine embedding, and μ_{ijk} represents the extended feature vector of the operation-machine arc. Here, only the neighborhood of the machine node is considered. I.e., the attention coefficient does not include the attention coefficient of the machine M_k to itself. In order to account for that, we calculate the attention coefficient e_{kk} as listed in Equation 4.14

$$e_{kk} = \text{LeakyReLU}\left(a\,[W^M \nu_k \,||\, W^M \nu_k]\right) \tag{4.14}$$

where e_{kk} represents the attention coefficient for machine M^k to itself, LeakyReLU denotes the activation function, a represents the weight, and W^M denotes the weight matrix for machine nodes. Then, all attention coefficients $e_{ijk} \forall O_{ij} \in N_t(M_k)$ are normalized together with e_{kk} using a softmax to obtain the normalized attention coefficients a_{ijk} and a_{kk}. The final machine embedding ν'_k is then calculated using equation 4.15.

$$\nu'_k = \sigma\left(\sum_{O_{ij} \in N_t(M_k)} \alpha_{ijk}\, W^O \mu_{ijk} + \alpha_{kk}\, W^M \nu_k\right) \tag{4.15}$$

where ν'_k represents the machine embedding for machine M_k, σ denotes the activation function, μ_{ijk} represents the raw feature vector of the arc between machine M_k and operation O_{ij}.

The operation node is then embedded given the eligible machines, the previous node, the next node and the node itself. Here, we are also able to learn the features of operation nodes such as the average processing time, or the time until the release date (Song et al., 2022). more specifically, we generate the operation node embedding μ'_{ij} using five different MLPs. Each of these networks has two hidden dimensions with 64 features and uses ELU activation. These networks together are responsible for the final projection of the operation node embedding. Since an operation can have multiple machines in its first-order neighborhood an element wise sum is applied to obtain the aggregated machine node embedding ($\overline{\nu}'_{ij}$). This is calculated as such: $\overline{\nu}'_{ij} = \sum_{k \in Nt(O_{ij})} \nu'_k$. The final operation node embedding is then given in Equation 4.16 below.

$$\mu'_{ij} = \text{MLP}_{\theta_0}\left(\text{ELU}\left[\text{MLP}_{\theta_1}(\mu_{i,j-1}) \,||\, \text{MLP}_{\theta_2}(\mu_{i,j+1}) \,||\, \text{MLP}_{\theta_3}(\overline{\nu}'_{ij}) \,||\, \text{MLP}_{\theta_4}(\mu_{ij})\right]\right) \tag{4.16}$$

where μ_{ij} represents the embedding for operation O_{ij}, MLP_{θ_i} denotes the MLP with parameters θ_i, ELU denotes the activation function, $\mu_{i,j-1}$ and $\mu_{i,j+1}$ represent the embeddings of the immediate predecessor and successor of O_{ij}, respectively, $\overline{\nu}'_{ij}$ represents the aggregated embedding of the neighboring machines, and μ_{ij} represents the original embedding of operation O_{ij}.

Once both the machine node and operation node embeddings are computed, they are stacked in order to arrive at the final state embedding (h_t). The two embeddings are concatenated ($||$) as given in Equation 4.17 below.

$$h_t = \left[\frac{1}{|O|} \sum_{O_{ij} \in O} \mu'_{ij} || \frac{1}{|M|} \sum_{M_k \in M} \nu'_k\right] \tag{4.17}$$

Action Space and Reward Function

Now that we have our state embedding (h_t), we need to convert the state embedding into a O-M allocation. We do so using the actor-critic set-up proposed by Song et al. (2022). Similar to the authors, we only use a single actor network.

The actor network computes a probability distributed across the available actions given the current state ($\pi(a_t|s_t)$). The actor consists of two MLP layers of 64 features with tanh activation (denoted as MLP_ω). The MLP layers are used to calculate the preference ($P(a_t, s_t)$ for selecting action a_t in state s_t (Equation 4.18). Then, a softmax is applied to convert the features into a probability density across the possible different O-M allocations. The softmax is given in Equation 4.19 below.

$$P(a_t, s_t) = \text{MLP}_\omega(\mu'_{ij} \,||\, \nu'_k \,||\, h_t) \tag{4.18}$$

$$\pi_\omega(a_t|s_t) = \frac{\exp(P(a_t, s_t))}{\sum_{a'_t \in \mathcal{A}_t} \exp(P(a'_t, s_t))} \quad \forall a_t \in \mathcal{A}_t \tag{4.19}$$

where $\pi_\omega(a_t|s_t)$ denotes the policy for selecting action a_t given state s_t, $P(a_t, s_t)$ represents the preference or score associated with action a_t and state s_t, $\mathcal{A}_t$ is the set of all possible actions at time t, and $\sum_{a'_t \in \mathcal{A}_t}$ denotes the summation over all actions in $\mathcal{A}_t$.

The critic network on the other hand computes the value of the current state ($\nu(s_t)$). The critic network is a simple MLP network (MLP_θ) consisting of two hidden layers of 64 features with tanh activation. The output layer of this network has a single feature, which represents the value of state s_t.

In order for the actions to be eligible given the constraints of Wefabricate. We make the following adjustments. First, we ensure that the agent can only select feasible actions by removing all actions that are not eligible from the decision set, similar to how this was implemented in Song et al. (2022). This is implemented by masking the decision set ($\mathcal{A}_t$) such that the agent will always select an available decision (i.e., setting probabilities to invalid numbers).

The following extra checks are executed on the action set. Only operation-machine allocations which (1) will finish during the day, or can run at night, (2) can start and run during the night or start during the day, and (3) release dates have passed. The same addition is made to the transition function, where in case the environment has no operation-machine pairs available, the environment will transition to the nearest (1) new day-start, (2) release date, or (3) time at which the first machine comes available. After this transition, the new state space s_t is computed.

Now, in order for the agent to learn, the reward function given in 4.20 is implemented. This reward function compares the expected makespan ($\hat{C}_{\max,t}$) in state s_t against the makespan in the previous state s_{t-1}. If the makespan has increased, we get a penalty (negative reward r_t). The estimated makespan is recomputed after every decision made by the agent to calculate the reward. The estimated makespan is calculated recursively as given in equation 4.21.

$$R_t = \hat{C}_{\max,t-1} - \hat{C}_{\max,t} \tag{4.20}$$

$$\hat{C}_{\max} = \max_{\forall O_{ij} \in O} [\max_{O_{i'j'} \in prec(O_{ij})} C_{i'j'} + \hat{p}_{ij}] \tag{4.21}$$

where $\max_{O_{i'j'} \in prec(O_{ij})} C_{i'j'}$ is the first potential starting moment of operation O_{ij} (i.e., maximum completion of all precedence constraints $prec(O_{ij})$), and $\hat{p_{ij}}$ is the average processing time of this operation.

4.2.3 Training

Similar to our proposed SLEGA, we make use of PPO to train our DRL agent. For a general introduction on this approach please refer back to Section 4.1.5. Again, we use the clipped PPO loss function given in Equation 4.10 to train our agent. The loss function can now be interpreted as such. $\hat{A}_t^{\pi_k}$ indicates the advantage of selecting a specific O-M pair over the average value of selecting any O-M pair. $r_t(\theta)$ represents the reward obtained at a particular time step t when selecting the O-M pair the agent θ prescribed $r_t(\theta)$ contains the clipped rewards obtained by the agent through setting certain parameters. The clipped loss then is a function of the advantages and rewards, computed by tacking the expected value over multiple trajectories, i.e., one or more scheduling environments solved with the DRL agent.

Since the actor returns a probability density function that a certain action should be selected, we can take several approaches for testing/training purposes. During training, actions are always picked by sampling the probability density function in order to encourage exploration. During testing, actions can either be taken based on sampling actions according to the distribution function (DRL-S), or by greedily picking the action with the highest probability (DRL-G). Sampling actions will result in different schedules, whereas greedily selecting actions will always result in the same schedule. Because sampling results in different schedules, more optimal schedules can potentially be found. This is further described in the next chapter.

The pseudocode for how the DRL agent is trained is given in Algorithm 7 below. Note that the algorithm for testing is a subset of this algorithm. More specifically, lines 8-14 represent the procedure used for testing. In this procedure, the sampled actions can be replaced by selecting actions greedily (i.e., change line 11). Every 20 iterations, we validate the policy on validation instances to avoid overfitting. Furthermore, we notice that training occurs in parallel. This allows us to take advantage of the multi-processing capabilities of modern computers, which significantly speeds up the process.

Algorithm 7 E2E-DRL training procedure

Input
HGNN network, policy network, and critic network with trainable parameters θ, ω and ϕ

```
1: procedure TrainPPOModelE2E-DRL
2:     for $iter \in \mathcal{I}$ do
3:         best reward $\leftarrow -$infinite
4:         if $iter\ mod\ 10 == 0$ then
5:             sample new batch $\mathcal{B}$ of FJSP instances
6:         end if
7:         for $b \in \mathcal{B}$ do                                        ▷ in parallel
8:             initialize environment $\mathcal{E}$ based of $b$
9:             while $\mathcal{E}$ not terminal do              ▷ until all schedules are done
10:                extract embeddings using HGNN.
11:                sample $a_t$ $\pi_\omega(\cdot \| s_t)$.
12:                receive reward $r_t$ and next state $s_{t+1}$.
13:                $s_t \leftarrow s_{t+1}$                         ▷ update state of environments
14:            end while
15:        end for
16:        compute the PPO loss $\mathcal{L}$, and optimize the parameters $\theta, \phi$ and $\omega$ for $\mathcal{R}$ epochs.
17:        update network parameters
18:        if $iter\ mod\ 20 == 0$ then
19:            valid reward $\leftarrow$ validate policy
20:            if valid reward $\geq$ best reward then
21:                save model
22:                best reward $\leftarrow$ valid reward
23:            end if
24:        end if
25:    end for
26: end procedure
```

Chapter 5

Experimental Setup

Now that we have outlined two novel algorithms for tackling the FJSP problem of Wefabricate in the previous chapter, we continue defining the experiments in this chapter. More specifically, four experiments will be executed. The experiments are outlined in Table 5.1 below. Note that experiments are done both in a single-objective optimization (SOO) as MOO setting. For experiment four, only the SLEGA is considered the E2E-DRL algorithm is not designed for the MOO setting. The proposed experiments allow us to highlight performance differences between the two algorithms described in the previous section across various scenarios. This benchmark provides novelty in the field of flexible job-shop scheduling algorithms.

Exp.	Setting	Dataset	Problem Type[1]	Description	Purpose	Using results from
1	SOO	*mkdata* *edata* *rdata* *vdata* *cudata*	FJSP	Evaluate scheduling algorithms on traditional FJSP setting.	Identify performance, scalability, and flexibility of implemented algorithms.	-
2	SOO	*ftdata*	1 & SDST	Evaluate scheduling algorithms on traditional FJSP with SDST.	Benchmark flexibility of algorithms to new characteristics.	1
3	SOO	*WFdata*	2 & Release Dates & Night Times	Evaluate scheduling algorithms on company-specific datasets.	Evaluate the performance of implemented algorithms for the company, analyzing robustness, flexibility and scalability.	1
4	MOO	*WFdata*	3	Evaluate scheduling algorithms in a multi-objective setting.	Benchmark algorithm flexibility and scalability when dealing with multiple objectives.	-

[1] Building on previous experiments by adding constraints.

Table 5.1: Experiment design

5.1 Performance Indicators

In Chapter 6, we report various performance indicators during training and testing. During training, we generally report the rewards obtained by the DRL agent, and the makespan. Furthermore, we define the following performance indicators which will be used to benchmark the algorithms on top of the makespan and operations cost objective. These performance indicators are the optimality gap, the ranking score, the win count and computation time.

The reward functions are different for each algorithm and are explained in Chapter 4. When rolling averages of rewards are shown, they are calculated according to Equation 5.1.

$$\hat{r}_t = \frac{r_t + \hat{r}_{t-1}(t-1)}{t} \tag{5.1}$$

Where $\hat{r}_t$ is the rolling average at timestep t, and r_t the reward at timestep t. The initial rolling average reward is set to 0 ($\hat{r}_0 = 0$).

The optimality gap (in percentage) is calculated as given in Equation 5.2.

$$G_{opt} = \frac{\hat{C}_{\max}}{LB - 1} * 100 \tag{5.2}$$

where $\hat{C}_{\max}$ is the average makespan for an instance set, and LB the average lower bound of this set. In case no lower bound is available for a certain problem instance, we consider our best-found solution to be the lower bound of this instance. Note that this LB thus does not have to be equal to the optimal solution of the instance.

We also introduce a ranking score and notion of wins in experiments 2 and 3. An algorithm is said to have won an instance if it has (or ties with) the best makespan for that specific instance. It thus could occur that multiple algorithms have "won" an instance. The win count is interesting as it helps identify which algorithm performs best overall test instances, as the optimality gap is influenced significantly by performance outliers in test instances. Table 5.2 shows an example of how the average ranking score and win count are computed given 3 algorithms and 3 instances. As can be seen, algorithm A reaches an average ranking score of $1\frac{1}{3}$ as it manages to obtain the lowest makespan in 2 out of 3 instances. Hence, the win count is also 2 for this algorithm.

Algorithm	Instance 1	Instance 2	Instance 3	Ranking Score	Win Count
A	**100 (1)**	**500 (1)**	80 (2)	$1\frac{1}{3}$	2
B	200 (2)	900 (3)	**20 (1)**	2	1
C	300 (3)	700 (2)	**20 (1)**	2	1

Table 5.2: Example win count and ranking score calculation.

Models are compared against benchmark information available from the literature. Benchmark approaches considered are the implemented GA (SO-EGA/MO-EGA), random scheduling (RANDOM), greedy scheduling (GREEDY) and the following dispatching rules: shortest processing time (SPT), most operations remaining (MOR), most work remaining (MWKR), first-in first-out (FIFO). Random scheduling, greedy scheduling and most work remaining are also implemented for benchmarking purposes on custom instances. Greedy scheduling is a constructive approach, where in each iteration, the operation-machine combination which increases the total makespan the least is selected.

5.2 Training Approach

To train the models introduced in the coming sections, we use a standardized approach. All models are trained on an AIME A4000 - Multi GPU HPC rack server with 96GB of GPU memory, 132GB of RAM, and 48 cores. Throughout all of our experiments, we use the training algorithms defined in the previous chapter. Note that little effort is taken to tune the hyperparameters of each training approach. The training approach is different for the E2E-DRL and SLEGA solutions and is further outlined below separately. Note that pseudocode of the training algorithms is given in Chapter 4.

5.2.1 SLEGA

The self-learning effective genetic algorithm (SLEGA) approach is trained using a custom OpenAI gym environment, together with the PPO agent from Stable Baselines3[1]. This implementation also uses PyTorch in the back end. During training, the batch size was set to 1, and the number of generations and population size per run were set to 100. Multi-processing was enabled as we saw that this was faster than increasing the batch size. The number of steps for updating was set to 100 (i.e., update every GA completion). For the updates, a mini-batch size for updates of 50 is used. The other parameters of the PPO policy are unchanged, which results in a learning rate of 0.0003, a discount rate of 0.99, and a clip range of 0.2. For each run of the genetic algorithm, a different FJSP instance is taken from the training instances. Every 1000 steps, a new batch of FJSP instances is sampled to train the model on. The best

[1]https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html

model found during validation is saved. Validation is executed on 50 independent instances, every 1000 steps, by computing the total reward over these 50 instances.

5.2.2 E2E-DRL

The end-to-end deep reinforcement learning approach is trained in PyTorch 1.13.1, using a custom OpenAI Gym environment[2]. For the PPO agent, the implementation from Song et al. (2022) is followed. During training, the batch size is set to 100, which means that a hundred scheduling instances are solved at the same time. During each selected action, memories are collected. When all instances are finally solved, the PPO agent will sample actions from memory with a mini-batch size set to 512, and optimize its parameters θ, ϕ, and ω for 3 epochs. In every iteration, the PPO policy is optimized. Each model is trained for 1000 iterations. The discount factor is set to 1 which means that no discounting is applied to the obtained rewards. This discount factor is set as the distant reward is based on the final makespan, hence the distance reward should be valued more and no discounting should be applied. The clip range is set to 0.2, and the learning rate is set to 0.0002 as this worked well in the paper of Song et al. (2022).

Every 20 iterations a new training batch of FJSP instances is sampled, in order to avoid overfitting. Furthermore, every 20 iterations, the model is also validated on 50 validation instances. The model with the lowest validation makespan is saved and returned at the end of the training procedure. The size of the training instances is different per experiment and will be further elaborated on below. Note that the validation reward (i.e., makespan) is thus different from the training reward (i.e., decrease in makespan). When testing and validating the performance of the E2E-DRL approach, we consider both a greedy action selection strategy (DRL-G) and a sampling action selection strategy (DRL-S). For the sampling approach, a sample size of 20 is considered.

5.3 Literature Benchmark Instances

This section describes the literature benchmark datasets used in testing. These are the Brandimarte dataset (*mkdata*), Hurink dataset (*edata*, *rdata* and *vdata*) and Fattahi dataset (*ftdata*). For literature benchmark FJSP datasets, flexibility is an important consideration. The level of flexibility discussed below considers the average amount of machines which are able to process a said action.

To describe the benchmark instances, we use the following formulation. n is the number of jobs, m is the number of machines, h_i indicates the number of operations per job i, $|M_{i,k}|$ indicates the average number of machines that can handle an operation, $p_{i,j,k}$ indicates the processing time range for operation-machine pairs, and $s_{i,j,k,l}$ indicates the sequence-dependent setup time range. LB indicates the instance lower bound. E.g., the value that can not be outperformed by any algorithm as this is infeasible given the instance. UB indicates the instance's upper bound. E.g., the best-known feasible solution. CP indicates the result found when constraint programming is executed for 10 minutes. The CP result supports identifying the complex problem instances.

Brandimarte

Brandimarte (1993) introduced the general FJSP and provided a set of 15 problem instances with medium flexibility (Behnke and Geiger, 2012). The parameters here are configured in the following way:

instance	n	m	h_i	$\lvert M_{i,k}\rvert$	$p_{i,k,j}$	LB	UB	CP
mk01	10	6	[5, 7]	3	[1,7]	36	39	40
mk02	10	6	[5, 7]	6	[1,7]	24	26	27
mk03	15	8	10	5	[1,20]	204	204	204
mk04	15	8	[3,10]	3	[1,10]	48	60	60
mk05	15	4	[5,10]	2	[5,10]	168	172	174
mk06	10	15	15	5	[1,10]	33	58	59
mk07	20	5	5	5	[1,20]	133	139	143
mk08	20	10	[5,15]	2	[5,20]	523	523	523
mk09	20	10	[10,15]	5	[5,20]	299	307	307
mk10	20	15	[10,15]	5	[5,20]	165	197	214

Table 5.3: Brandimarte FJSP instances description

[2]https://www.gymlibrary.dev/

Hurink

Hurink et al. (1994) created the *sdata* dataset based of the instances *mt06*, *mt10* and *mt20* datasets from Crowston et al. (1963), and the instances *la01* to *la40* generated by Lawrence (1984). For the *sdata* set, every operation is assigned to exactly one particular machine. The *sdata* set was adjusted by enlarging the respective set of adjustable machines to create three more datasets: *edata*, *rdata* and *vdata* (Behnke and Geiger, 2012). The final datasets (4 datasets with 43 instances each) are described in Table 5.4 and 5.5 below. The average lower bound, average upper bound and average constraint programming results equal (981.7, 1004.0, 1009.1) for *edata*, (899.6, 909.9, 917.9) for *rdata* and (894.8, 895.6, 895.9) for *vdata* respectively.

instance	n	m
mt06	6	6
mt10	10	10
mt20	20	5
la01-05	10	5
la06-10	15	5
la11-15	20	5
la16-20	10	10
la21-25	15	10
la26-30	20	10
la31-35	30	10
la36-40	15	15

Table 5.4: Hurink instance description

	avg $\vert M_{i,k}\vert$	max $\vert M_{i,k}\vert$	min $\vert M_{i,k}\vert$
sdata	1	1	1
edata	1.15	3	1
rdata	2	3	1
vdata	$\frac{1}{2}m$	$\frac{4}{5}m$	1

Table 5.5: Hurink dataset modification description

Fattahi

The instances from Saidi-Mehrabad and Fattahi (2007) are described in Table 5.6. Because of their relatively small size, they are generally used in mathematical programming benchmarking. Instances 1-10 are considered small, and instances 11-20 are considered of medium size. Partial flexibility here includes that operations can be executed on a subset of machines, whereas total flexibility indicates that all machines are able to process a said operation.

Instance	n	m		s,i,j,k,l	Flexibility	LB	UB	CP
1	2	2	2	[3,8]	Total	66	66	66
2	2	2	2	[3,10]	Partial	107	107	107
3	3	2	2	[6,15]	Partial	212	221	221
4	3	2	2	[8,21]	Partial	331	355	355
5	3	2	2	[3,6]	Total	107	119	119
6	3	2	2	[5,18]	Partial	310	320	320
7	3	5	3	[8,23]	Total	397	397	397
8	3	4	3	[4,13]	Total	216	253	253
9	3	3	3	[4,11]	Total	210	210	210
10	4	5	3	[10,28]	Partial	427	516	516
11	5	6	3	[8,24]	Partial	403	468	468
12	5	7	3	[9,26]	Partial	396	446	446
13	6	7	3	[9,30]	Partial	396	466	466
14	7	7	3	[10,31]	Partial	496	554	554
15	7	7	3	[10,30]	Partial	414	514	541
16	8	7	3	[10,30]	Partial	614	635	634
17	8	7	4	[10,31]	Partial	764	879	931
18	9	8	4	[10,30]	Partial	764	884	884
19	11	8	4	[10,30]	Partial	807	1088	1070
20	12	8	4	[10,33]	Partial	944	1267	1208

Table 5.6: Fattahi dataset description

Based on the dataset descriptions above, the most complicated instances to solve are mk04, mk06 and mk10 from *mkdata*, the *edata* from Hurink et al. (1994) and instances 11-20 from *ftdata*. This conclusion is based on the CP result.

5.4 Custom FJSP instances

This section describes the datasets custom generated for this research. These are the custom vanilla FJSP instances (*cudata*) and the constraint Wefabricate instances (*WFdata*).

Besides these instances, we also use the instances of Song et al. (2022) for testing purposes (*sodata*). The data of Song et al. (2022) consists of 100 instances for sizes 10x05, 15x10, 20x10 and 20x05. A copy of the *sodata* is made for Experiment 2, where a setup time between 1 and 15 is uniformly generated for each possible combination of a machine and two operations (*stdata)*. This range was selected as it is quite similar to the testing dataset.

5.4.1 Vanilla FJSP instances

In order to further investigate the performance differences between the two algorithms on the vanilla FJSP, we generate custom vanilla FJSP instances. These FJSP instances are referred to as *cudata.* These instances have varying sizes between 5x5 and 95x95 and are generated using the FJSP case generator from Song et al. (2022). In this case generator, we define a minimum and maximum number of operations per job. These values are set to 5 and 10 respectively. The minimum and maximum number of machines per operation is not set, but this will be between 1 and the total number of machines by definition. The number of machine options is also uniformly randomly generated. The processing time is set uniformly between 1 and 20 for each operation. This dataset thus includes extreme instances in terms of the jobs-to-machine ratio. I.e., there are instances with 0.05 jobs per machine and instances with 19 jobs per machine. Analysing the performance of the two proposed algorithms for these edge cases illustrates how scalable and flexible the algorithms are.

5.4.2 Wefabricate instances

The Wefabricate-specific instances (*WFdata*) are generated using a set of products that customers can potentially order. This set consists of 75 different jobs, each consisting of 1-5 operations. The jobs can have a quantity between one and five. The processing time is given per operation per product, whereas the total processing time for an operation is then calculated by multiplying the processing time by the quantity. Figure 5.1 below displays the distribution of processing times for the different operations. As can be seen, the duration of operations ranges anywhere from 0 to 5000. Because quantities are between 1 and 5, there exist 375 different jobs which can exist within an instance. The number of machines and jobs is set before generation. The instance is considered completely flexible, which means that all operations can be scheduled on all machines. Furthermore, the operations have equal processing time on each of the available machines.

Sequence-dependent setup times are based on the pallet of an operation, in case the pallet is different for two operations, a setup time of 10000-time units is considered. We consider this significant set-up time as we would like to discourage any scheduling algorithm to schedule operations with different pallets back-to-back. Deadlines are generated following the total work content (TWK) method of Blackstone et al. (1982). More specifically, in order to generate deadlines, we initialize our deadlines to the processing times of each operation. Then in order to calculate the actual deadline, we increase a tightness factor by 10%. With this tightness factor, the new deadline is calculated by multiplying this with the processing time per operation. This process is repeated until 75% of jobs can finish before their deadline when a shortest processing time approach is taken. Release dates are then calculated by subtracting a constant value of 15000 from this deadline. Whether operations can run at night, or whether they can be started at night is uniformly generated. A maintenance job per machine is added to the set of jobs. The hypervolume reference point for the instance is set by computing 10 schedules and considering the highest cost and highest makespan.

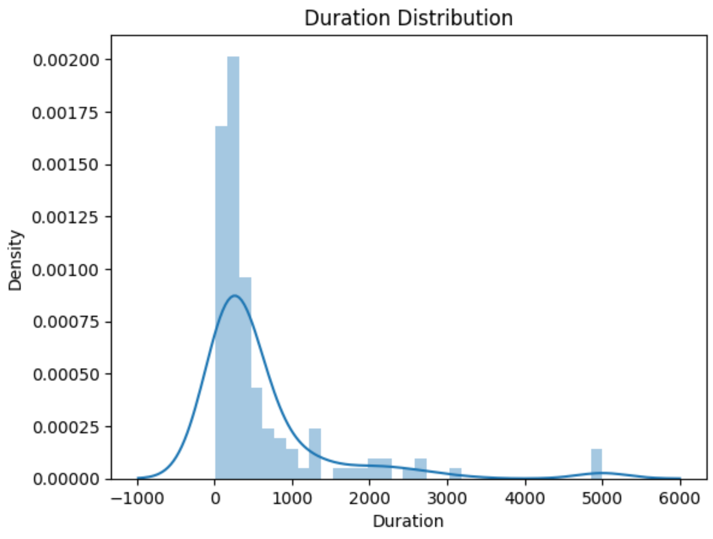


Figure 5.1: Example instance processing time distribution.

In order to create a reference point for the hypervolume, we create 10 random schedules. The 10 schedules are then evaluated, and the highest makespan and highest cost are taken as the fixed reference point for this company instance. The reference points are only used in the last experiment.

Chapter 6

Results

This chapter covers the results of the experiments outlined in the previous chapter. The results are discussed per experiment separately. In this Chapter, we mainly make observations of what we have seen throughout the experiments. The conclusions for the company are made in the next chapter.

6.1 Experiment 1. E2E-DRL vs. SO-SLEGA for Traditional FJSP

In the first experiment, we benchmark the implemented algorithms on traditional FJSP instances. We also compare the implemented algorithms against simpler heuristics (Table 6.2). Furthermore, literature and custom datasets of various sizes are considered. Finally, we also dive deep into the policy learned by the SLEGA. Results for these sizes will be discussed separately.

6.1.1 Training and Validation

For Experiment 1, we dive into the traditional FJSP problem. We do so to bridge unexplored areas in literature. For this, we train both our SO-SLEGA and E2E-DRL approaches on 100 different FJSP instances. We do so for 5 different FJSP sizes from *sodata*, 10x05, 15x10, 20x05, 20x10, and a mix of the aforementioned instance sizes. The E2E-DRL training algorithm is executed for a thousand iterations.

The resulting models are then validated on 10 FJSP instances with of similar size as the models are trained on. The model that is trained on mixed instances is validated on 10 FJSP instances of size 20x10. Validation is the first moment where we deploy both DRL-G and DRL-S for the E2E-DRL solution. Each of the trained models is executed once on the validation set, where the sample size for DRL-S is set to 20. The result for training and validation is given in Table 6.1. As can be seen, training times are quite similar for the E2E-DRL and SO-SLEGA approaches. For inference time, we see that our E2E-DRL approach is much faster, both for sampling and greedily selecting actions. This makes sense as the SLEGA conducts many more computations throughout the optimization process. From the validation performance, we notice that the DRL-S performs better than DRL-G as expected. Furthermore, we notice that the DRL-S and SO-SLEGA approaches are quite similar in terms of performance.

Furthermore, Figure 6.1 covers the train and validation curves for the E2E-DRL model trained on FJSP instances of size 15x10. As can be seen, the reward curve is converging to approximately -30. This means the average makespan across the training instances increased by 30 compared to the estimated makespan from the start. We also notice that the validation makespan decreases quite rapidly and that it already is close to optimal around 150 iterations.

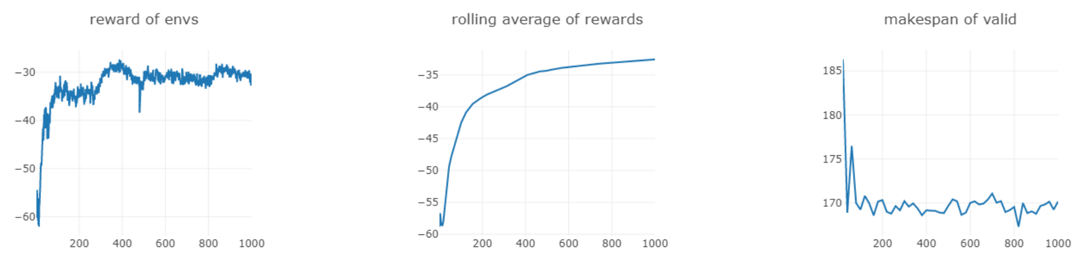
Figure 6.2 display the first and best schedule from the validation dataset for the 15x10 FJSP instances. The colors and numbers indicate the different jobs, the machines are on the y-axis, and the moment in time is given on the x-axis. As can be seen, the makespan of the first created schedule on the dataset is equal to 162, while the makespan of the best-created schedule is equal to 149. In the first schedule, multiple gaps cause the difference in makespan.

Figure 6.3 displays the learning curves of the SO-SLEGA of instance sizes 20x10. The raw reward shows a stagnation around a value of 80. The rolling average also shows how improvements are stagnating.

Instances	Training				Validation					
	E2E-DRL		SO-SLEGA		Inference Time			Average Makespan		
	Duration	Iteration[1]	Duration	Timestep[1]	DRL-G	DRL-S (20x)	SO-SLEGA	DRL-G	DRL-S (20x)	SO-SLEGA
10x05	0.67h	940	1.70h	31000	0.8s	2.6s	21.0s	116.1	111.0	111.2
15x10	2.96h	820	3.44h	34800	3.0s	15.8s	100.1s	183.8	178.7	174.4
20x05	1.59h	240	2.68h	50000	1.5s	7.2s	108.5s	242.1	228.7	214.4
20x10	4.58h	40	5.22h	21600	4.8s	30.8s	145.8s	278.8	224.8	219.6
Mixed	2.73h	200	3.18h	25000	2.3s	6.8s	36.0s	211.8	210.1	223.7

[1] The iteration and timestep of the algorithms where the validation performance was highest.

Table 6.1: Training and Validation of Experiment 1.



(a) Reward per iteration. (b) Rolling average of reward. (c) Validation curve

Figure 6.1: Training and Validation of 10x15 E2E-DRL model for experiment 1.

6.1.2 Testing

Now, in order to test these models, we proceed with the following experiments. Each trained model is tested on three datasets. The first two are the Brandimarte (*mkdata*) (Brandimarte, 1993) and Hurink (*e*-, *r*- and *v-data*) datasets (Hurink et al., 1994). From the Hurink datasets, we only use the *la* instances, 1-40 for *e*- and *r-data*, and 1-30 for *v-data*. The third is the *cudata* dataset explained in the previous Chapter. The first and second datasets are used to benchmark my implementation against existing AI-based algorithms, heuristic solutions, and dispatching rules. Furthermore, the two implementations are compared in order to identify key differences. Especially on the third dataset, key differences will be investigated in order to analyze the scalability, flexibility, and generalizability of the algorithms.

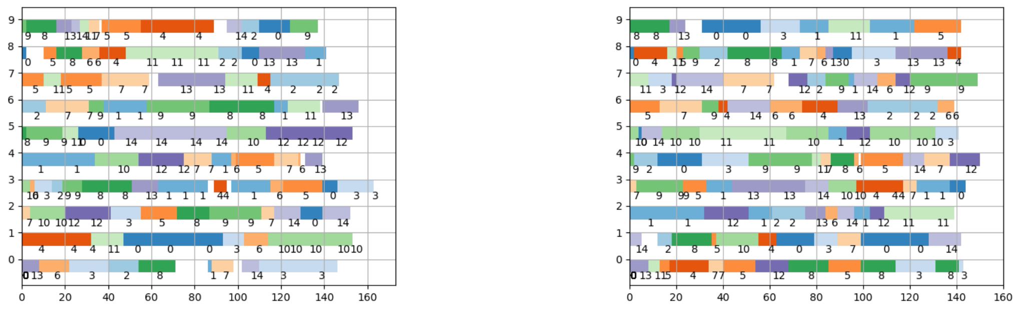
In order to test the algorithms properly, we deploy both DRL-G and DRL-S. DRL-S is used with a sample size of 20. For the *mkdata*, the SO-SLEGA will be executed 10 times and report the average and best makespan, while running this algorithm only once on the *e*-, *r*- and *v-data*. Furthermore, we execute random scheduling, greedy scheduling, and the SO-EGA implementation to compare against. We report both average algorithm duration and objective values, to identify the trade-off between time and optimality. The main results are given in Table 6.2.

Literature Datasets

On the *mkdata* of Brandimarte (1993), we note that the best optimality gap is obtained by the EGA of G. Zhang et al. (2011) (6.11%). Since our SO-EGA is following this EGA implementation exactly, similar results are expected. However, we only obtain an optimality gap of 17.09%. This is partially because of our lower number of instances ($n = 100$, as to $n = 300$ for example for mk06). However, after trying various parameter settings, we still do not manage to reach a makespan better than 67 for mk06, whereas the EGA reached a makespan of 58. The authors did share the final schedule for this instance. This is given in Figure 6.4 below. As can be seen, only three operations are scheduled on machine 4 (index 3). Where G. Zhang et al. (2011) scheduled 9 operations on machine 4. Perhaps this is where a difference in optimal schedules is coming from. This is not further investigated.

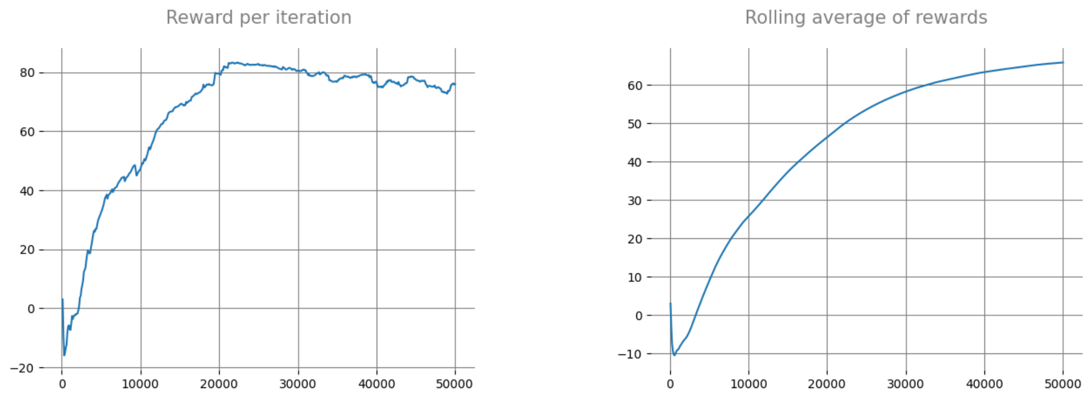
Next, we notice that the 2SGA of Rooyani and Defersha (2019) performs very well on this dataset (optimality gap of 7.29%). However, in this paper, the population size and number of generations range from 1000 to 3000. Hence the performance gap is considered fair, as the number of schedules that are evaluated is much larger.

We notice that the performance of DRL-S is significantly better than the performance of DRL-G, for all literature datasets. More specifically, performance gaps decrease by 2-10 percentage points depending on



(a) Iteration 20 (b) Iteration 820 (best)

Figure 6.2: First and best schedule for 15x10 E2E-DRL model of experiment 1 on the validation set.



(a) Reward per iteration. (b) Rolling average of rewards.

Figure 6.3: Training and Validation of 10x15 SO-SLEGA model for experiment 1.

the trained instance size. Mixed instance sizes seem to be the way to go for DRL-G, as the performance is the highest for this training dataset. For DRL-S, performance is the highest for 15x10 training instances. We also note that DRL-S is approximately three times as slow while solving 20 times as many instances as DRL-G. Compared to the dispatching rules (i.e., SPT, MOR, MWKR and FIFO), DRL-S outperforms all of them, except for when trained on 20x05 instances. The model does not generalize well to the test dataset in this case.

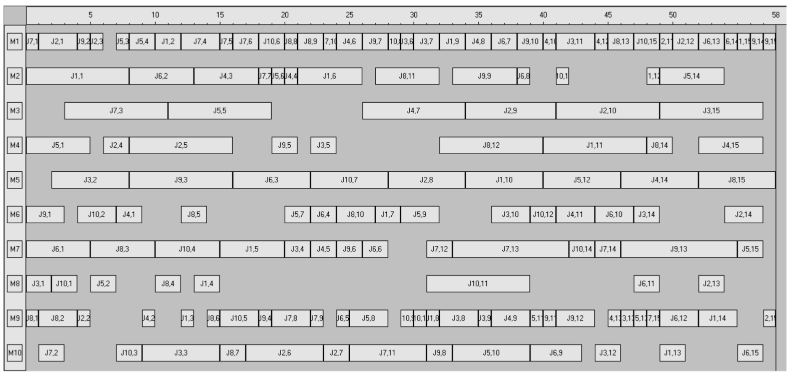
For the MK datasets, the E2E-DRL approach is faster (6x as fast), but performance is still lacking compared to the SO-SLEGA and SO-EGA implementations. The SO-SLEGA models namely reach an optimality gap of around 10%, which is similar to the SLGA of R. Chen et al. (2020). The DRL-S model only reaches an optimality gap of 18.25% at best. For the *rdata* and *vdata* however, we note that the DRL-S approach reaches higher performance, with the lowest optimality gap of 7.26% and 2.87% respectively. The SO-SLEGA only reaches an optimality gap of 9.93% and 12.71%. Training on mixed instances appears to offer the highest benefits for these instances. The root cause of why the E2E-DRL approach outperforms the SO-SLEGA approach will be further investigated in the next subsection.

When comparing our algorithms against more simple heuristics. We note that the SO-SLEGA outperforms all traditional heuristics. The optimality gap of the SLEGA has an improvement of 6.86 percentage points over the traditional genetic algorithm (SO-EGA). Figure 6.5 illustrates this improvement. As can be seen, the average makespan of the population for the SLEGA decreases more steadily than the average makespan for the population of the traditional GA. Furthermore, the final average makespan is significantly lower for the SLEGA than this of the GA. The traditional GA seems to converge too early to a sub-optimal solution.

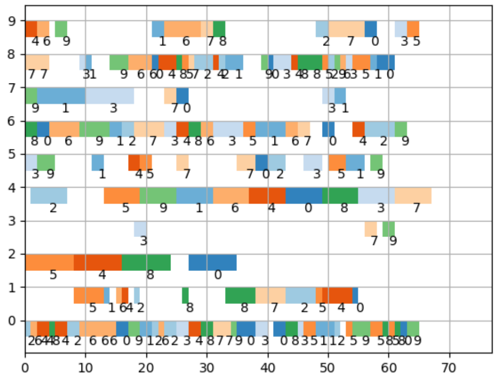
Within the heuristics, the MWKR heuristic appears to work best next to the traditional genetic algorithm.

Approach		mkdata			edata		rdata		vdata	
		$\hat{C}_{\max}$	*Gopt*	$\hat{t}(s)$	$\hat{C}_{\max}$	*Gopt*	$\hat{C}_{\max}$	*Gopt*	$\hat{C}_{\max}$	*Gopt*
OPT*		163.3	-	-	1005	-	923	-	807.9	-
SPT		283.2	73.48%	-	1305.35	29.89%	1184.80	28.36%	-	-
MOR		202.3	23.89%	-	1211.2	20.52%	1064.0	15.28%	-	-
MWKR		200.17	22.58%	-	1179.9	17.40%	1046.2	13.35%	-	-
FIFO		206.1	26.20%	-	1255.46	24.92%	1082.1	17.24%	-	-
RANDOM		637.5	290.39%	60.00	1225.9	21.98%	1212.9	31.41%	1041.7	28.94%
GREEDY		484.9	196.94%	1.30	1290.5	28.41%	1150.4	24.64%	894.5	10.72%
SO-EGA		192.2	17.09%	30.02	1144.6	13.89%	1105.3	19.75%	953.4	18.01%
EGA (G. Zhang et al., 2011)		173.3	6.11%	-	-	-	-	-	-	-
SLGA (R. Chen et al., 2020)		181.3	11.02%	-	-	-	-	-	-	-
2SGA (Rooyani and Defersha, 2019)		175.2	7.29%	-	-	-	-	-	812.20	0.53%
	10x05	200.1	22.54%	1.22	1193.1	18.72%	1049.7	13.73%	856.1	5.97%
	15x10	200.3	22.66%	1.23	1197.5	19.15%	1054.3	14.23%	858.0	6.20%
DRL-G	20x05	220.1	34.78%	1.22	1269.5	26.32%	1125.1	21.90%	897.4	11.08%
	20x10	199.3	22.05%	1.21	1192.5	18.66%	1046.2	13.35%	841.3	4.13%
	Mixed	198.0	21.25%	1.29	1218.0	21.19%	1056.3	14.44%	845.0	4.59%
	10x05	194.6	19.16%	4.60	1139.5	13.38%	1009.0	9.32%	827.6	2.44%
	15x10	193.1	18.25%	4.32	1144.9	13.92%	1008.2	9.23%	828.8	2.59%
DRL-S	20x05	208.1	27.37%	4.25	1176.0	17.01%	1049.1	13.66%	857.8	6.18%
	20x10	195.2	19.53%	4.24	1152.85	14.71%	1015.5	10.02%	830.9	2.85%
	Mixed	196.8	20.51%	5.11	1107.5	10.20%	990.0	7.26%	831.1	2.87%
	10x05	180.7	10.66%	29.83	1103.5	9.80%	1047.8	13.52%	894.5	10.72%
	15x10	184.5	12.98%	29.62	1110.3	10.48%	1061.0	14.95%	914.7	13.22%
SO-SLEGA	20x05	180.1	10.29%	34.91	1097.7	9.22%	1040.3	12.71%	895.4	10.83%
	20x10	180.5	10.53%	33.23	1099.5	9.40%	1034.9	12.12%	888.13	9.93%
	Mixed	179.6	10.23%	28.18	1111.8	10.63%	1069.3	15.85%	955.8	18.31%

Table 6.2: Results of different algorithms on literature test data.

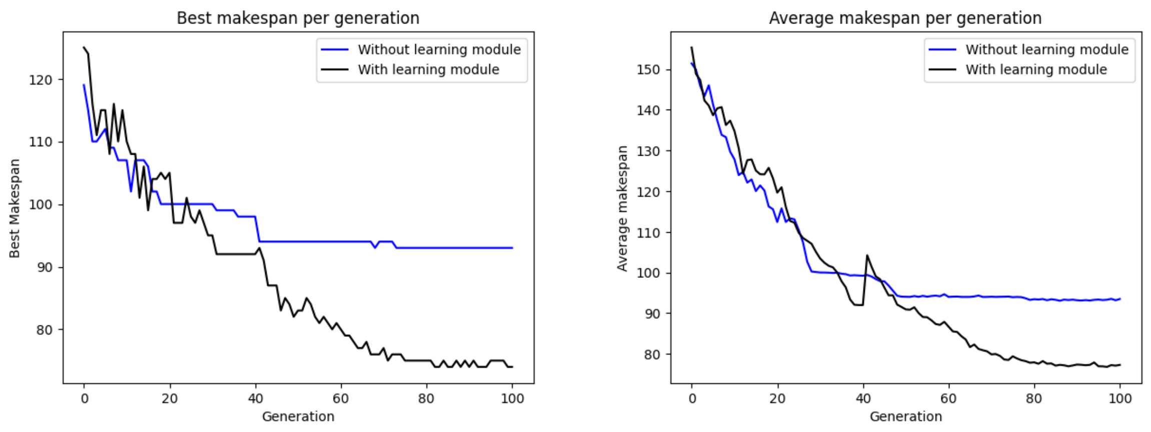


(a) EGA (58)



(b) SO-SLEGA (20x10) (67)

Figure 6.4: mk06 Schedule for EGA (G. Zhang et al., 2011) and SO-SLEGA (20x10).



(a) Best makespan

(b) Average makespan

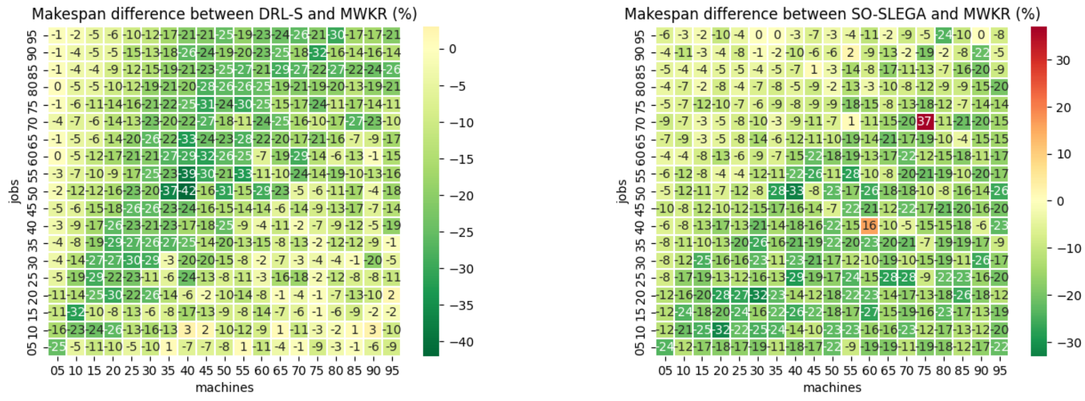
Figure 6.5: Advantage of learning module

Custom Instances with Varying Sizes

In order to further explore the root cause of the performance difference between the E2E-DRL and SO-SLEGA approaches, we test the E2E-DRL and SO-SLEGA models (both trained on 15x10 instances) on the *cudata*. We also compare these approaches against the MWKR dispatching rule. Figure 6.6 below displays two heatmaps, showcasing the percentual performance difference between SO-SLEGA/DRL-S and the MWKR rule for various instance sizes. A negative value of e.g., -10% indicates a makespan improvement of the first mentioned algorithm. For example, in Figure 6.6a, DRL-S has a 25% improvement on MWRK for the instance with 5 jobs and 5 machines.

As can be seen, DRL-S can improve the makespan of a schedule by up to 40%, whereas SO-SLEGA only improves performance by up to 30%. We note that there appears to be a relationship between the number of machines, jobs, and performance increase. More specifically, when the number of machines and jobs (e.g., 50x50) increases simultaneously, the performance increase is much higher than when only one of the two characteristics increases (e.g., 80x20). This relationship is not identified for the SO-SLEGA (Figure 6.6b).

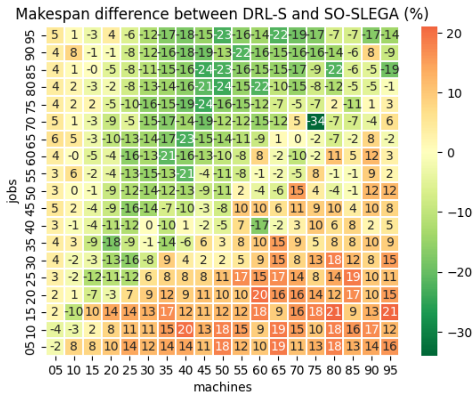
Now, comparing DRL-S against the SO-SLEGA directly, we note that this relationship exists here as well. However, now the performance gain of the DRL-S approach only occurs within this same region (up until 30%), whereas outside this region the SO-SLEGA outperforms DRL-S (up until 20%) makespan increase. with regard to computation time, we note that DRL-S is always faster, but that the percentage computation time difference is much lower for larger instances. This is most likely due to the genetic algorithm approach's large overhead, whereas this is not the case for E2E-DRL.



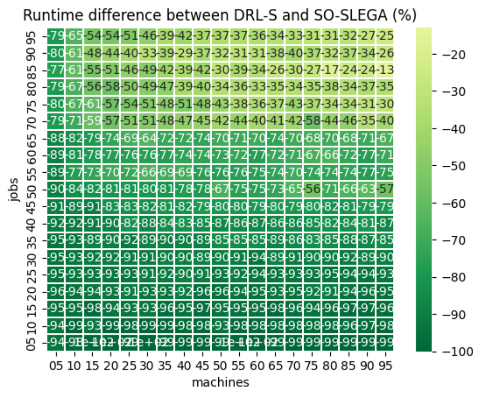
(a) DRL-S (15x10)

(b) SO-SLEGA (15x10)

Figure 6.6: Makespan difference between SO-SLEGA (15x10), DRL-S (15x10), and MWKR.



(a) Makespan



(b) Computation time

Figure 6.7: Difference between DRL-S (15x10) and SO-SLEGA (15x10).

Figure 6.8 further investigates the relationship between the instance size and increase in performance by plotting the performance gain against the job-to-machine (J-M) ratio. As can be seen, E2E-DRL performs much better when instances have job-to-machine ratios around 2, whereas SO-SLEGA performs better when instances have a very small ratio, or a ratio higher than 5. This is likely due to the fact that deep reinforcement learning approaches aren't as beneficial when dealing with very easy (low number of jobs per machine) or very complicated (high number of per machine) instances.

Table 6.3 deep dives on three instances along different J-M ratios of Figure 6.8. More specifically, three instances with J-R ratios of 0.2, 1.6 and 9.0 are taken. As these instances are custom generated, the optimal performance is unknown. In order to compute a lower bound, we make use of the constraint programming (CP) solver of OR-tools[1]. This solver finds a good lower bound for the 15x80 and 80x50 FJSP instance. The quality of the lower bound for the complex 90x10 instance (146) is uncertain, as it is far from the best solution found (604). We follow these bounds nonetheless.

For the FJSP instance with a J-M ratio of 0.2, we notice that the optimal solution is found by the CP model of OR tools and by the (self-learning effective) genetic algorithm. For this instance, the search space is quite big and there also are a high number of good solutions as there are a lot of machines available. The E2E-DRL approach only reaches a solution with a makespan of 120 (optimality gap of 21%). Thus we can conclude for such a simple instance, a search or constraint programming approach is better.

For the FJSP instance with a J-M ratio of 1.6. For this instance, the solution space is relatively big, and the fraction of good solutions is relatively small. This means that it is harder to make good decisions. We note that the best solution is found by the constraint programming approach. This solution takes 6.5 hours to reach, however. The E2E-DRL approach performs better than the SO-SLEGA approach in this case, as it reaches an optimality gap of 33% (153) whereas SO-SLEGA reaches an optimality gap of 63% (188).

The last investigated FJSP instance has a J-M ratio of 9.0. Here, the instance is typically considered a difficult instance to solve as the number of machines is relatively low. This also means that the solution space is relatively small. The fraction of good solutions is relatively high here because of the small solution space. Here we notice that the CP approach finds the best solution with an optimality gap of 314% (604). SO-SLEGA outperforms DRL-S as the algorithms reach an optimality gap of 371% (687) and 336% (637) respectively.

We thus conclude that for instances where the jobs are balanced with the resources, an E2E-DRL approach performs better. This follows from Figure 6.8 and Table 6.3. Because of the balance, the fraction of good solutions is rather small. It thus will be harder for constraint programming or search approach (like SO-SLEGA) to find a good solution within the search space, while the E2E-DRL approach is able to select the correct actions sequentially.

[1]https://github.com/google/or-tools/blob/stable/examples/python/flexible_job_shop_sat.py

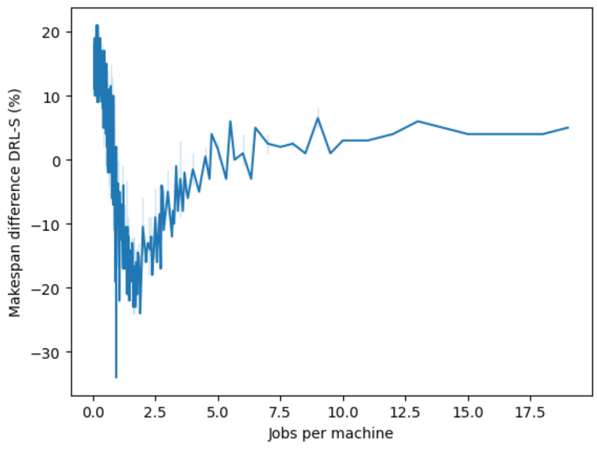


Figure 6.8: Makespan difference (%) between DRL-S (15x10) and SO-SLEGA (15x10) over job-to-machine ratios.

Instance	J-M Ratio	OR-Tools LB	OR-Tools Computation Time	*Gopt*			
				OR-Tools	DRL-S 15x10	SO-SLEGA 15x10	SO-EGA
15x80	0.2	99[1]	30s	0% (99[1])	21% (120)	0% (99[1])	0% (99[1])
80x50	1.6	115	6.5h	6% (122)	33% (153)	63% (188)	75% (203)
90x10	9.0	146	7.5h	314% (604)	371% (687)	336% (637)	476% (695)

[1]Optimal solution found. [2] DRL-S computes a solution within 1-60 seconds, whereas the (SLE)GAs take from 1-5 minutes.

Table 6.3: Deep dive on optimality gap of three different instances.

In order to validate this performance improvement, we take a single instance (50x50 in this case) and run each algorithm 20 times. Figure 6.9 displays the resulting distributions of those 20 runs. As can be seen, the DRL-S distribution is centered around 117, whereas the SO-SLEGA is centered around 127. In order to validate this difference in distribution, the Wilcoxon rank-sum test is conducted. From the test, it is concluded that both distributions are significantly different ($p = 1.91\text{e}{-6}$) and that thus you are more likely to get a lower makespan while using the DRL-S approach in this instance.

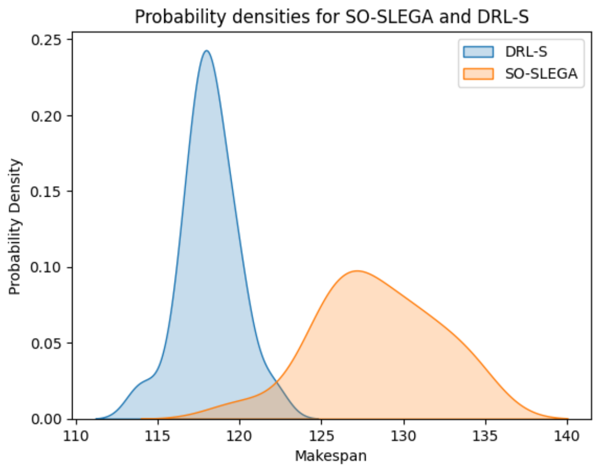


Figure 6.9: Distribution of makespan for 50-job, 50-machine instances for DRL-S(15x10) and SO-SLEGA (15x10).

6.1.3 SLEGA Policy Deep Dive

In this section, we dive deep into the SLEGA policy to understand what actions are taken and why, in order to explain the performance difference.

An example of how the mutation rate, mutation probability, and crossover probabilities are set during a run of the SLEGA is given in Figure 6.10 below. As can be seen, the agent generally sets the crossover rate as high as possible, with few exceptions throughout the generational process. The individual mutation probability is almost always set to 0, whereas there are few spikes to overcome stagnation seen in the blue line.

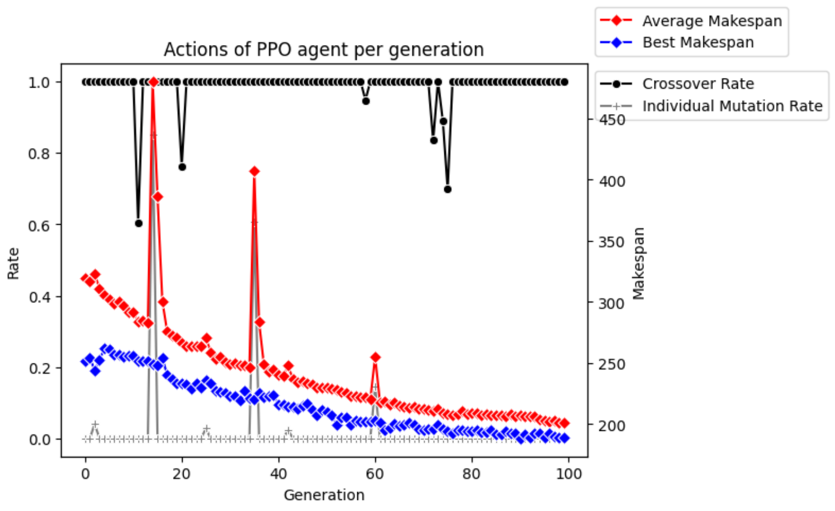


Figure 6.10: SO-SLEGA: Actions of PPO agent per generation

Figure 6.11 displays the state space under which mutation rates were chosen for a different run. As can be seen from the graph, the mutation rate is set larger than 0 for 4 different points in time. From the state space, we can see that the agent might base the decision to set the mutation rate high on the stagnation count; as the stagnation count goes up, the agent seems more likely to set the mutation rate higher. This seems quite logical as this means the agent overcomes early convergence. After closer investigation through a Pearson correlation test, the correlation between none of the state features and mutation action was found to be significant. This was to be expected given the complex MLP policy of the PPO agent. Figure 6.12 show a decision tree which tries to predict the agent's decision of exploration (i.e., setting an individual mutation rate bigger than 0) or exploitation given the current state space. From the tree, we note that the remaining budget and stagnation count are of high importance. If the population strayed away from the highest fitness seen so far, the agent also selects exploration. Hence, this verifies our conclusions drawn from Figure 6.10.

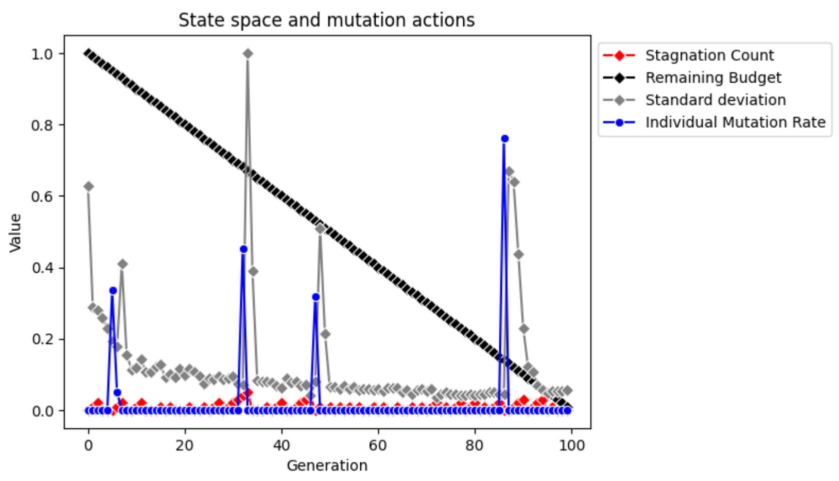


Figure 6.11: SO-SLEGA: Partial state space and mutation rate action per generation.

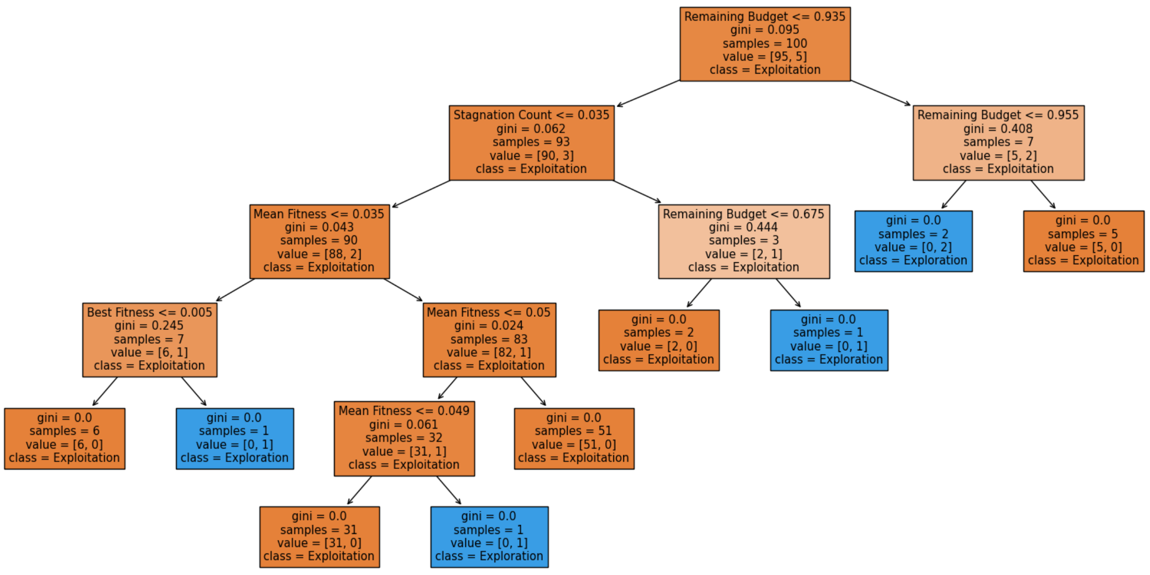


Figure 6.12: SO-SLEGA policy explanation in a decision tree.

From Figure 6.10 it appears that the crossover and mutation rates are almost constant. Learning would thus not add any value, except the removal of the parameter tuning step. Hence we test whether learning adds significant value using a Wilcoxon rank-sum test. This test checks whether two samples are taken from the same population. For our use case, we take 20 solutions of a single instance from a trained SLEGA model, and 20 solutions from the regular GA with a crossover rate set to 1, and a mutation rate set to 0. The distributions of results are given in Figure 6.13. As can be seen from the figure, the distributions overlap, but yet seem different. The Wilcoxon rank-sum test confirms this hypothesis, as the null hypothesis of equal distributions is rejected ($p \leq 0.001$). Hence we consider that learning still adds significant value.

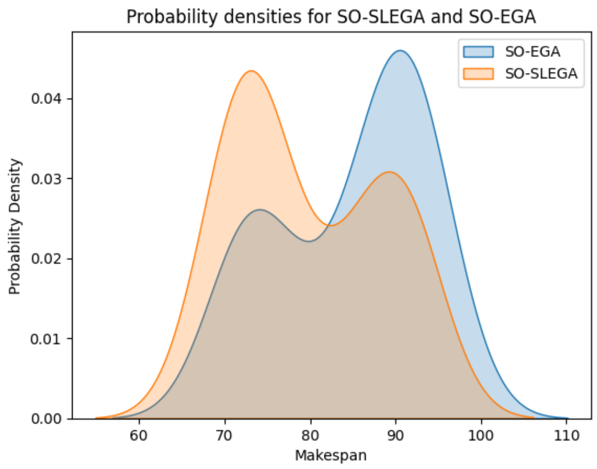


Figure 6.13: Makespan distribution of mk06 for SO-EGA and SO-SLEGA.

To summarize the first experiment, we conclude that it depends on the instance size and training approach which algorithm performs best in terms of generalizability and performance in the vanilla FJSP setting. More specifically, we have seen that on the benchmark datasets, E2E-DRL outperformed the SO-SLEGA on the rdata and vdata FJSP instances, while SO-SLEGA outperformed E2E-DRL on the mkdata and edata FJSP instances. E2E-DRL is much faster (approximately 6 times as fast) than the SO-SLEGA approach. We have seen that the performance (i.e., makespan) difference between E2E-DRL and SO-SLEGA depends on the type of instances. When instances have relatively few good solutions and a bigger search space (job-machine ratio of about 2.0), E2E-DRL can reduce makespan up to 30%. While if instances have a higher/lower job-machine ratio, the SO-SLEGA approach performs up to 20% better. SO-SLEGA and E2E-DRL outperform simple dispatching rules, while still being outperformed by constraint programming. This is acceptable as constraint programming is generally much slower. In terms of scalability, we have seen that E2E-DRL is generally much faster than SO-SLEGA, hence is considered better.

6.2 Experiment 2. E2E-DRL vs. SO-SLEGA for FJSP with Sequence-Dependent Setup-Times

For our second experiment, we extend the classical FJSP definition with sequence-dependent setup times. Again, we separate the section into training and validation, and testing.

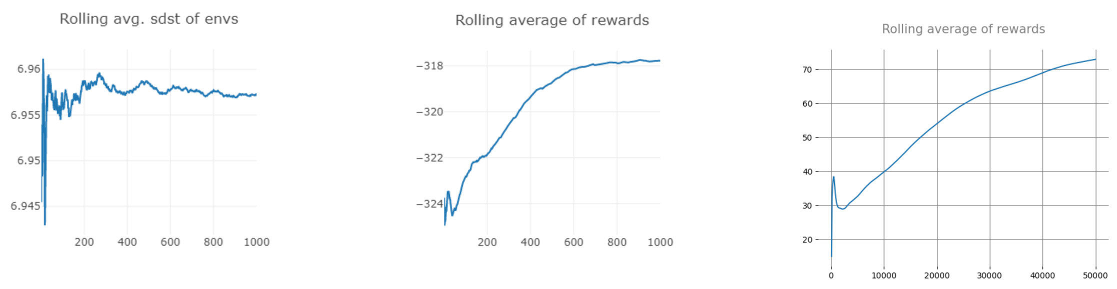
6.2.1 Training and Validation

The training and validation data is similar to experiment 1. Again, both the SO-SLEGA and E2E-DRL approaches are trained on instances of size 10x05, 15x10, 20x05, and 20x10. Mixed instances are no longer considered here. Now, the instances are taken from the *stdata*.

Table 6.4 shows training and validation results for Experiment Two. We note that now the SO-SLEGA performs slightly better on validation instances, except for the 15x10 SO-SLEGA. It appears the performance here is lacking behind after closer inspection. From the learning curve, it appears that the model had difficulty understanding how to interact with the environment. As this is only one of the models, we proceed either way. It is also noted that training times have increased slightly, this could have been because multiple algorithms were running simultaneously on the machine. Figure 6.14 displays the training curves for the 20x05 instance. As can be seen, the rolling average of the mean sequence-dependent setup time per environment stabilizes and decreases only slightly. This indicates that in order to get the highest reward, (i.e., the lowest makespan increase) it does not necessarily mean that sequence-dependent setup times should be minimized. Furthermore, we note that both E2E-DRL and SO-SLEGA implementation seem to manage to learn how to create schedules to optimize for the reward.

Instances	Training				Validation					
	E2E-DRL		SO-SLEGA		Inference Time			Average Makespan		
	Duration	Iteration	Duration	Timestep	DRL-G	DRL-S (20x)	SO-SLEGA	DRL-G	DRL-S (20x)	SO-SLEGA
10x05	1.1h	260	1.3h	46500	0.3s	0.7s	13.1s	199.3	190.1	168.0
15x10	4.5h	580	6.1h	31900	1.1s	3.6s	78.4s	283.8	284.9	342.3
20x05	2.6h	660	4.0h	40400	0.6s	1.7s	36.1s	394.6	380.1	340.8
20x10	6.8h	80	6.2h	23200	1.3s	6.3s	39.8s	365.6	378.2	375.2

Table 6.4: Training and Validation of Experiment 2.

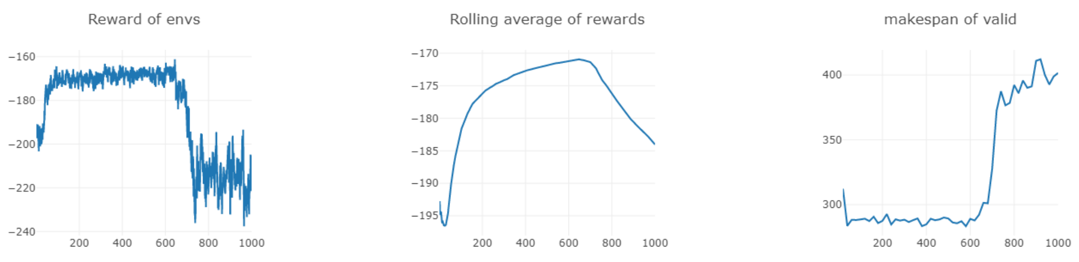


(a) Rolling average SDST, E2E-DRL. (b) Learning curve, E2E-DRL. (c) Learning curve, SO-SLEGA.

Figure 6.14: Training curves for Experiment 2.

Catastrophic Interference

While training the models on the FJSP instances with the addition of sequence-dependent setup times, we ran into the issue known as catastrophic interference. Catastrophic interference occurred when a model loses the knowledge it has learned about a particular environment all of a sudden. As can be seen from Figure 6.15 below, this is the case for our E2E-DRL model. Between iterations 600 and 800, the reward the model obtains per run decreases significantly. The model doesn't relearn any behaviour afterwards. Trying several approaches, no solution was found to avoid this behaviour from occurring. To proceed, we decided to save the model where the validation makespan was the lowest as we did in the previous experiment.



(a) Reward per iteration (b) Rolling average of reward (c) Validation curve

Figure 6.15: Catastrophic interference of E2E-DRL model on 15x10 FJSP instances with SDSTs.

6.2.2 Testing

For testing purposes, we again benchmark trained models against traditional heuristics and dispatching rules. We do so on the FJSP instances with SDSTs of Fattahi (Saidi-Mehrabad and Fattahi, 2007). These instances range in size from 2x2 to 12x8. Furthermore, we compare our trained models against our models that were trained on instances without SDSTs, to identify whether retraining is necessary.

As can be read from the table, we see that the SO-SLEGA approach outperforms E2E-DRL significantly. SO-SLEGA manages to reach an optimality gap of 1.29%, whereas DRL-S only reaches an optimality gap of 6.80% at best. Again, the computation time of E2E-DRL is much faster (below a second) than SO-SLEGA which can take anywhere between 42 and 70 seconds. This difference is much larger than in the first experiment, but this is mainly explained by the fact that this test dataset has smaller instances (max. 12x8) than the MK dataset (max. 20x15) used in the previous experiment.

The win count and average rank also confirm that SO-SLEGA outperforms E2E-DRL for FJSP with SDSTs. The win count of SO-SLEGA is 15 and 16 for the retrained and vanilla FJSP models respectively, while DRL-S only reaches a win count of 7. The average rank of SO-SLEGA equals 1.20 and 1.25, which is the highest performance across all algorithms. E2E-DRL only reaches an average rank of 2.70 and a win count of 7.

We note that DRL-S still outperforms the traditional genetic algorithm, random scheduling, and simple dispatching rules. This is confirmed by the optimality gap, where DRL-S reaches a value of 6.80%. The vanilla genetic algorithm follows with an optimality gap of 13.28%. Random scheduling does have a win count of 10, which is higher than the win count of DRL-S (7). However, the average rank (3.75) of random scheduling is worse than the rank of DRL-S (2.70). This is due to the fact that random scheduling is able to brute-force the first 10 instances which are relatively small (4 jobs 5 machines at most). For larger instances, DRL-S achieves a much better makespan and thus ends up with a better average rank.

When comparing the newly trained algorithms versus the ones from the previous experiment, we note that retraining allows for a better makespan to be reached. SO-SLEGA improves from an optimality gap of 1.71% to 1.29%. The average rank however decreased from 1.20 to 1.25. This could have happened due to chance. Since this improvement is considered insignificant, we conclude that retraining doesn't add any extra benefits for SO-SLEGA, while it does for DRL-S, as here the best optimality gap decreases from 8.04% (10x05) to 6.80% (15x10), and the average rank decreases from 3.10 to 2.70. This difference between the increase in re-training makes sense, as for the SO-SLEGA we are simply learning how to adjust the parameters of our search. In the E2E approach, on the other hand, we are required to extract information from the environment in order to select the best action directly, and thus nothing is learned on sequence-dependent setup times for E2E-DRL in Experiment 1.

To summarize, we conclude that the SO-SLEGA approach is better at solving the FJSP setting with SDSTs than the E2E-DRL approach. We have seen that the average rank for SO-SLEGA (1.20/1.25) is much higher than that of E2E-DRL (2.70/3.10/5.25). We have also seen that DRL-S still outperforms vanilla SO-EGA and simple dispatching rules, while DRL-G is beaten by random scheduling and by the vanilla SO-EGA. We concluded that the performance increases for E2E-DRL when models are retrained on datasets with SDSTs, while performance does not increase for SO-SLEGA. Hence we conclude that SO-SLEGA generalizes better to FJSP instances with unseen characteristics. Generalizability is further explored in the next couple of experiments.

		ftdata				
		$\hat{C}_{\max}$	*Gopt*	$\hat{t}(s)$	Win Count	Average Rank
LB[2]		536.4	-	-	-	-
MWKR		787.4	46.81%	0.73	0	8.85
RANDOM		638.6	19.06%	60.00	10	3.75
GREEDY		667.5	24.45%	0.92	0	7.30
SO-EGA		607.6	13.28%	59.24	4	5.10
DRL-G	10x05	643.2	19.91%	0.08	3	5.25
	15x10	634.9	18.37%	0.08		
	20x05	681.3	27.03%	0.08		
	20x10	648.4	20.89%	0.08		
DRL-S	10x05	577.0	7.57%	0.23	7	2.70
	15x10	572.8	6.80%	0.22		
	20x05	594.6	10.86%	0.21		
	20x10	576.3	7.45%	0.24		
DRL-S[1]	10x05	580.0	8.04%	0.22	7	3.10
	15x10	580.3	8.18%	0.22		
	20x05	589.9	9.97%	0.25		
	20x10	583.3	8.74%	0.22		
SO-SLEGA	10x05	543.3	1.29%	56.60	15	1.25
	15x10	568.4	5.98%	42.22		
	20x05	556.4	3.74%	66.08		
	20x10	549.4	2.42%	70.22		
SO-SLEGA[1]	10x05	547.7	2.12%	68.83	16	1.20
	15x10	547.1	1.99%	55.21		
	20x05	552.2	2.96%	66.92		
	20x10	545.5	1.71%	68.20		

[1]Models trained on vanilla FJSP (Experiment 1), [2]lower bound calculated using best makespan found over various algorithms.

Table 6.5: Results of different algorithms on SDST literature test data.

6.3 Experiment 3. E2E-DRL vs. SO-SLEGA for Company-Specific Instances

We proceed to benchmark our E2E-DRL and SO-SLEGA implementations on the company-specific instances. This section is separated into training and validation, and testing. Furthermore, we add an ablation study in order to deep dive into why the performance of E2E-DRL degrades. Note that we do not use OR-tools to benchmark here, as it is complicated to configure for our custom instances.

6.3.1 Training and Validation

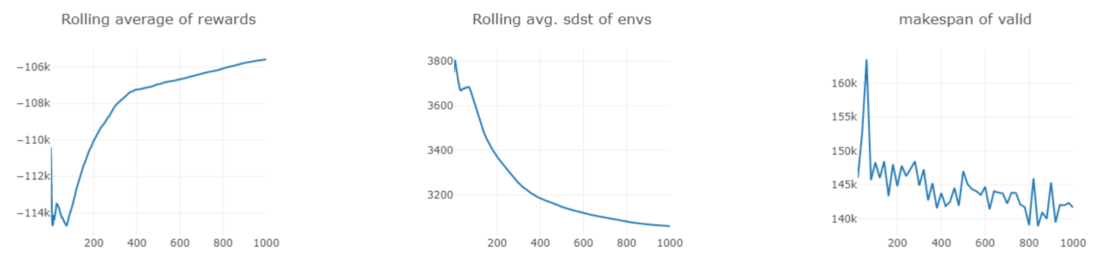
Similar to Experiments 1 and 2, we train the E2E-DRL and SO-SLEGA approaches on instance sets of different sizes. Training instances with sizes of 17x02, 42x02, 64x04, and 88x08 are selected. Results are given in Table 6.6 below.

Instances	Training				Validation					
	E2E-DRL		SO-SLEGA		Inference Time			Average Makespan		
	Duration	Iteration	Duration	Timestep	DRL-G	DRL-S (20x)	SO-SLEGA	DRL-G	DRL-S (20x)	SO-SLEGA
17x02	0.8h	840	3.3h	14000	0.2s	2.8s	48.9s	142787	127219	101122
42x02	2.4h	460	6.2h	24000	0.5s	3.3s	93.8s	281604	282146	223935
64x04	4.8h	160	7.5h	16000	0.8s	3.8s	111.3s	243945	230984	192755
88x08	7.4h	460	10.9h	11000	1.1s	4.3s	142.1s	202454	185219	165779

Table 6.6: Training and Validation of Experiment 3.

From Table 6.6, we can see that training on these specific instances takes significantly longer than on literature instances (Table 6.1). This is mostly because of the additional logic that is introduced in the environment by characteristics such as release dates, deadlines, and night times. From the save iteration/timestep, we can see that the lowest validation makespan is achieved on various points throughout learning. Sometimes early, such as on the 64x04 instance for E2E-DRL (iteration 160). Looking at the inference times, we note that E2E-DRL again is very fast at obtaining a solution. Even with sampling (20x), all solutions are obtained within 5 seconds. On validation makespan, we note that SO-SLEGA outperforms E2E-DRL for each instance set. A sampling approach tends to be better than a greedy approach, except for the 42x02 instances. This is probably due to randomness.

Learning and validation curves for SO-SLEGA and E2E-DRL are given in Figures 6.16 and 6.17. For the E2E-DRL curve, we see that the validation makespan decreases quite rapidly and reaches a low at iteration 840. Whereas the SO-SLEGA validation reward increases slightly and then appears quite random. SO-SLEGA reaches the highest validation performance on timestep 14000. The reward does seem to stagnate around 14000.



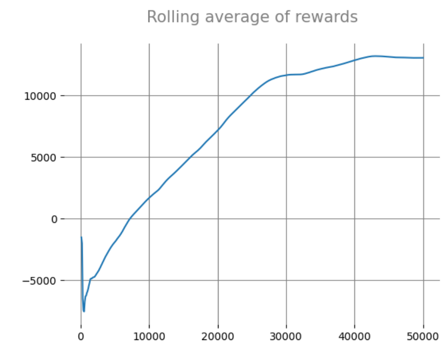
(a) Rolling average rewards (b) Rolling average SDSTs (c) Validation makespan

Figure 6.16: Training and validation curves of E2E-DRL model on 17x02 instances.

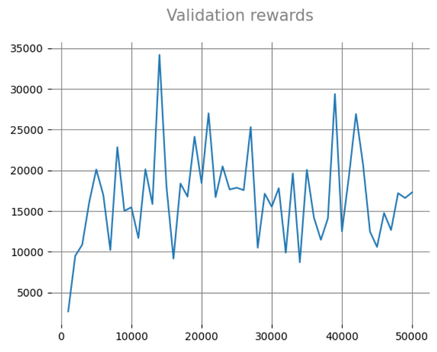
6.3.2 Testing

To test these models, we consider a diverse set of instance sizes. More specifically, instances where the number of jobs ranges between 5 and 100 and the number of machines ranges between 2 and 10 are considered. An instance of size 100x10 should match the industry scale. We use these instances in order to test generalizability and scalability. Furthermore, we compare the algorithms against simple heuristics, random scheduling and the best-performing models from Experiment 1.

Table 6.7 below displays the results of the tested algorithms on the company-specific FJSP instances.



(a) Rolling average of rewards.



(b) Validation rewards per timestep.

Figure 6.17: Training and validation curves of SO-SLEGA model on 17x02 instances.

Note that the makespan is significantly larger than for literature instances. This is because the makespan is given in seconds for these instances, whereas in literature there is usually no unit specified. From the table, we notice that the SO-SLEGA approach trained on 17x02 instances is performing best, with an optimality gap of 4.96%. We note that when training on different instance sizes, the performance decreases significantly. Especially for the model that was trained on 42x02 where the optimality gap is 21.60%. Close inspection of this trained model shows that the model did not converge properly, which might cause this performance issue. The models trained on 64x04 and 88x08 instances have an optimality gap of 7.69% and 9.63% respectively. The average rank achieved by SO-SLEGA is 1.21, with a total win count of 81 (81%).

DRL-S tends to outperform DRL-G, which is explained by the fact that DRL-S samples actions, whereas DRL-G picks actions in a deterministic fashion. More specifically, when training on instances of sizes 17x02, 62x04 and 88x08, sampling actions improve performance by 1.5, 11.5 and 3.2 percentage points respectively. Performance when training on 42x02 instances does decrease when sampling actions, but this is less than 1 percentage point so considered insignificant. Where DRL-S outperformed greedy scheduling and dispatching rules in experiments 1 and 2, it fails to outperform greedy scheduling and the vanilla SO-EGA for company-specific instances. Greedy scheduling and vanilla SO-EGA achieve an average rank of 3.16 and 3.14 respectively, which is better than DRL-S (3.51) and DRL-G (3.96). This is further investigated below. Retraining seems necessary as the performance of both SO-SLEGA and E2E-DRL increases significantly when models are retrained. The SO-SLEGA and E2E-DRL models increase in performance by 3 and 10 percentage points respectively when retraining. Figure 6.18 displays the approach that found the best solution for each test instance. From this figure, we note that greedy tends to outperform the other tested algorithms when instances are larger in size (top right corner) and more complicated (higher jobs/machines ratio). This is most likely due to the fact that all algorithms do not perform well relative to the actual optimal solution, but this is not further explored. We see that E2E-DRL obtains the best solution in the lower right corner. SO-SLEGA generally performs well.

		WFdata				
		$\hat{C}_{\max}$	*Gopt*	$\hat{t}(s)$	Win Count	Average Rank
LB[2]		132173	-	-	-	-
MWKR		266848	101.89%	6.22	1	6.85
RANDOM		195869	46.95%	10.00	3	5.29
GREEDY		164040	23.07%	15.63	24	3.16
SO-EGA		161651	22.31%	37.45	13	3.14
DRL-G	17x02	177421	34.23%	1.68	1	3.96
	42x02	181551	37.44%	1.47		
	64x04	192354	45.53%	2.32		
	88x08	187690	42.00%	2.20		
DRL-S	17x02	175477	32.76%	5.46	5	3.51
	42x02	182804	38.31%	4.35		
	64x04	177147	34.03%	2.87		
	88x08	183488	38.82%	3.35		
	15x10[1]	188850	42.88%	2.88		
SO-SLEGA	17x02	138733	4.96%	35.92	81	1.21
	42x02	160726	21.60%	33.96		
	64x04	142333	7.69%	29.65		
	88x08	144898	9.63%	30.45		
	15x10[1]	143992	8.03%	39.54		

[1]Models trained on vanilla FJSP (Experiment 1), [2]lower bound calculated using best makespan found per instance across all algorithms.

Table 6.7: Results of different algorithms on WF-specific instances.

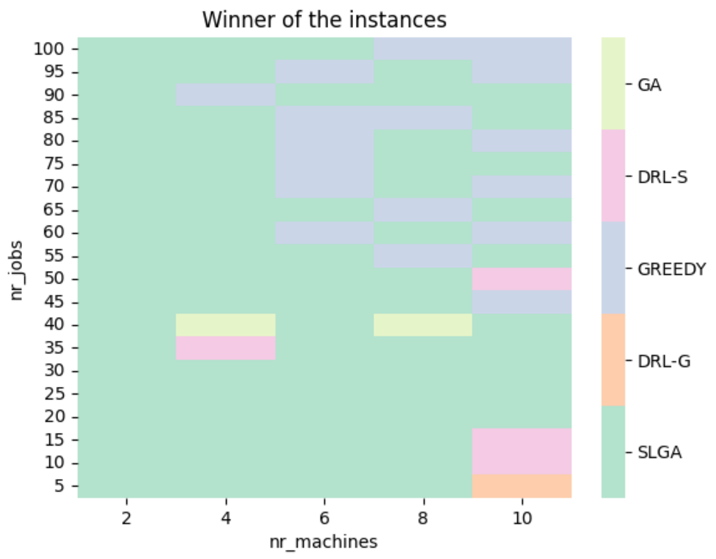
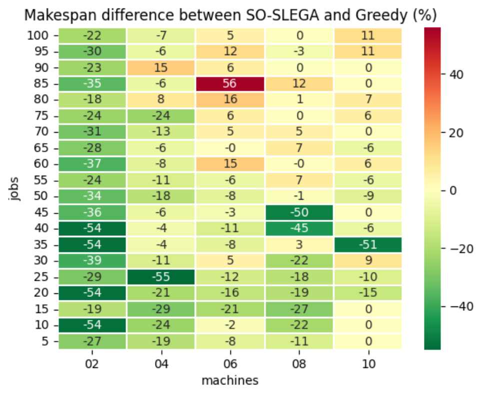


Figure 6.18: Approach with the best solution per test instance.

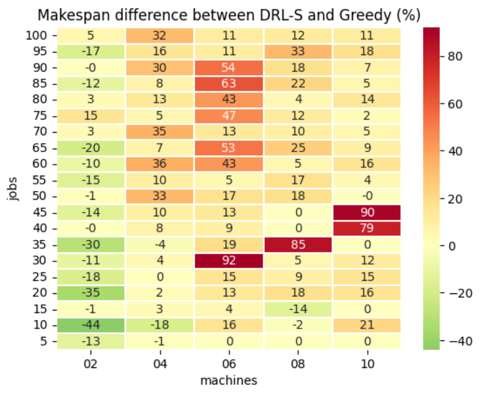
Table 6.8 displays the number of wins of the DRL-S and SO-SLEGA models (trained on 17x02) when comparing them against the greedy scheduling. As can be seen, the SO-SLEGA model outperforms the greedy scheduling model 76 times, ties 7 times, and loses 17 times. DRL-S only outperforms greedy scheduling 35 times, ties 5 times and loses 60 times. To further identify why these differences occur, we display heatmaps in makespan differences in Figure 6.19 for the models trained on 17x02 FJSP instances. Figure 6.19a shows that greedy scheduling is better for instances in the top right corner. For the instance of size 85x06, greedy scheduling is even 56% better than the genetic algorithm. While for the instance of size 25x04, SO-SLEGA outperforms greedy scheduling by 55%. This is further investigated below. Figure 6.19b displays the differences between DRL-S and greedy scheduling. DRL-S tends to outperform greedy scheduling for instances with only 2 machines, whereas greedy scheduling significantly outperforms DRL-S for instances with 30 jobs or more, and 4 machines or more.

Algorithm	Wins	Losses	Ties	Win Rate (%)
SO-SLEGA	76	17	7	76%
DRL-S	35	60	5	35%

Table 6.8: Algorithms compared against greedy heuristic, performance aggregated by taking best performance over the trained models.



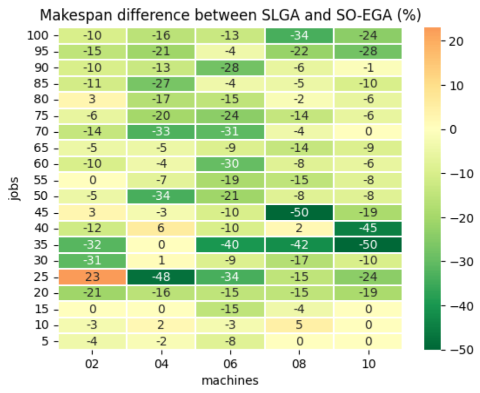
(a) SO-SLEGA (17x02)



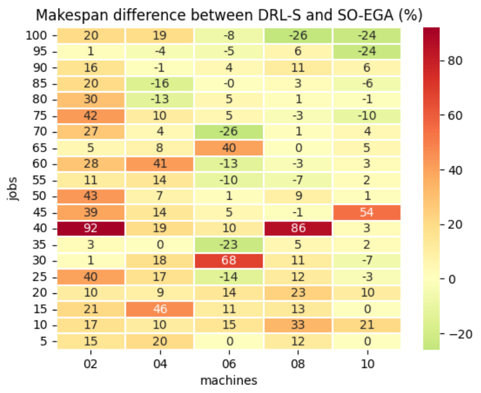
(b) DRL-S (17x02)

Figure 6.19: Makespan difference between SO-SLEGA (17x02), DRL-S (17x02), and Greedy scheduling.

When comparing the algorithm to the vanilla SO-EGA (Figure 6.20), we notice that SO-SLEGA generally outperforms SO-EGA, improving up to 50%. For only a single instance (25x02), SO-SLEGA is significantly outperformed (23%). This could be due to randomness. When comparing DRL-S against the vanilla SO-EGA, we note that DRL-S can outperform SO-EGA in the top-right corner, which is because of the increased complexity. Because of this complexity, the vanilla SO-EGA most likely did not converge yet. Besides that, no relationships are identified between the performance difference and instance characteristics.



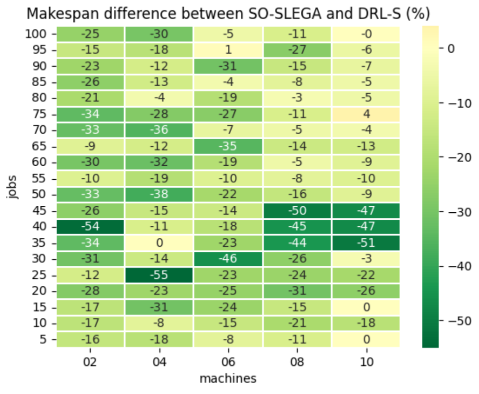
(a) SO-SLEGA (17x02)



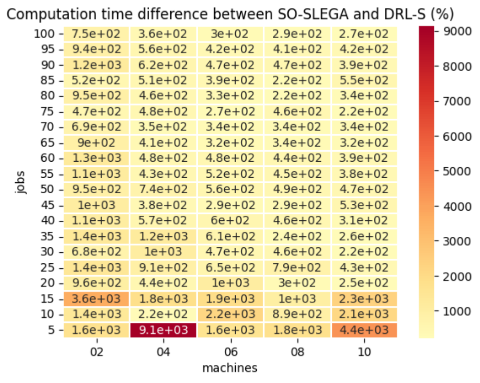
(b) DRL-S (17x02)

Figure 6.20: Makespan difference between SO-SLEGA (17x02), DRL-S (17x02), and SO-EGA.

When comparing SO-SLEGA against DRL-S directly, we note that SO-SLEGA almost always outperforms DRL-S. The performance increase for SO-SLEGA is up to 55%, while only losing 4% of performance at most. Regarding Computation time, DRL-S is much faster. The computation time gain is between 1000 and 9000% (10x and 90x as fast). These results are shown in Figure 6.8.



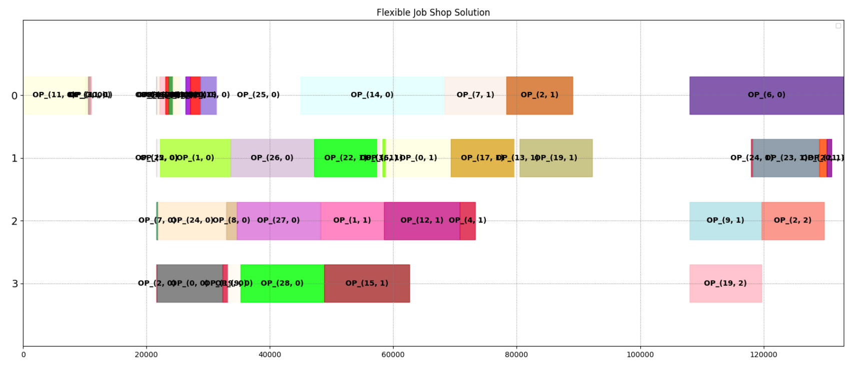
(a) Makespan



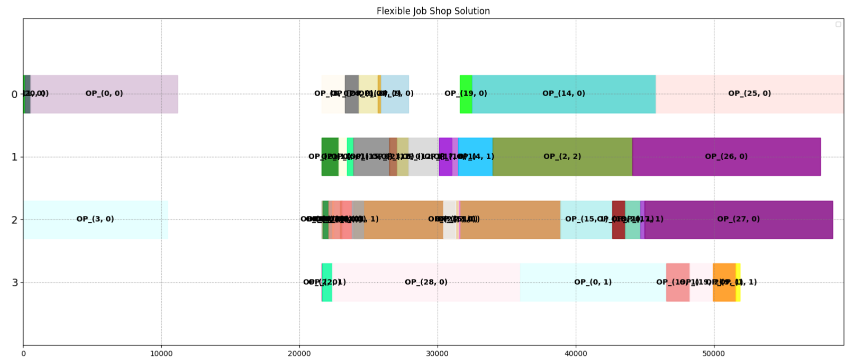
(b) Computation time

Figure 6.21: Comparison between SO-SLEGA (17x02) and DRL-S (17x02).

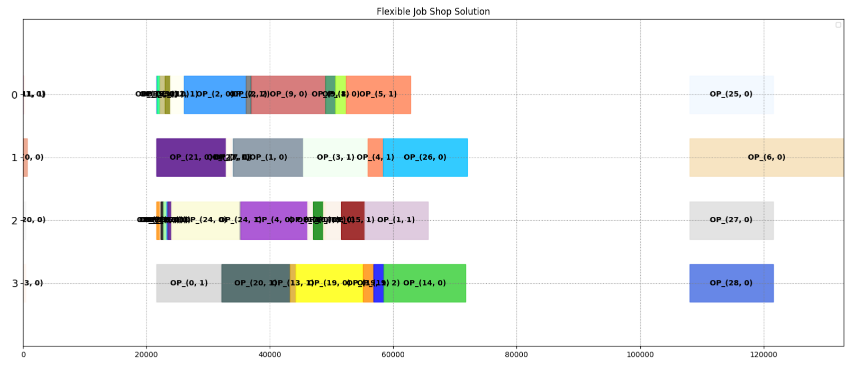
To further investigate why the performance difference between SO-SLEGA, DRL-S and greedy scheduling is caused, we look at two specific instances. The 25x04 instance and the 85x06 instance. For the first instance, SO-SLEGA performs best with a makespan of 59410, while DRL-S and greedy scheduling lack behind with both a makespan of 133000. From figure 6.22 it can be seen that SO-SLEGA manages to schedule everything within the first day. DRL-S and greedy scheduling do not; 4 and 8 operations remain to be processed on the second day respectively. Generally, we see that SO-SLEGA manages to make the schedule much tighter than either one of the other approaches. This could be because the SO-SLEGA optimizes over different schedules with the notion of backfilling, whereas E2E-DRL does not have this information available, and has to build up a schedule from scratch. The large gap in performance is explained because if a slight mistake is made during scheduling, the operation is to be executed on the second day, which includes the overhead of night times.



(a) Greedy Scheduling (133000)



(b) SO-SLEGA (17x02) (59410)

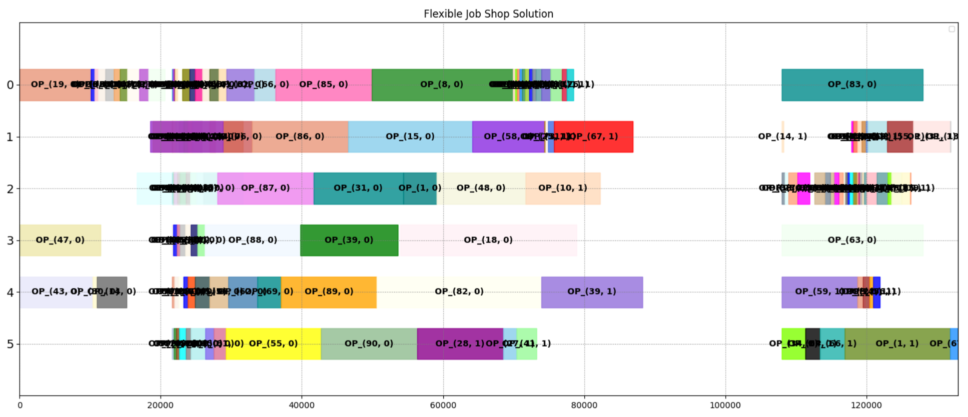


(c) DRL-S (17x02) (133000)

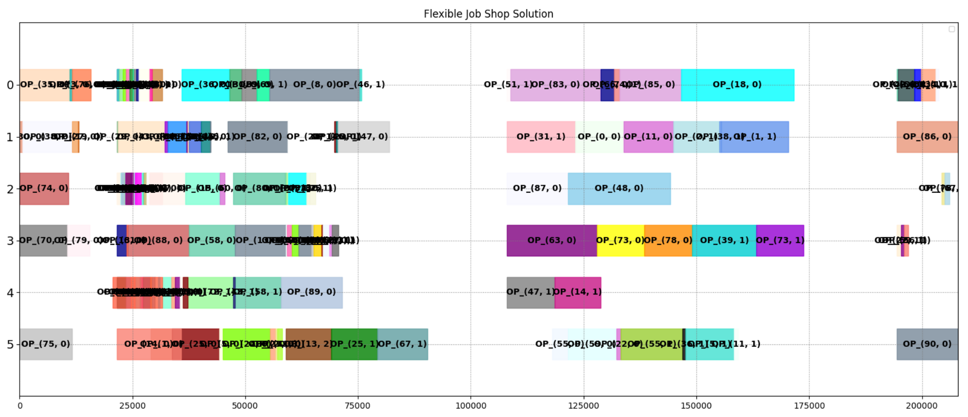
Figure 6.22: Test instance (25x04) schedules.

Figure 6.23 displays the schedules computed for the different algorithms for the company-specific FJSP instance of size 85x06. Here we note that greedy scheduling manages to obtain the best solution. As the instance has much more operations than the previous one, the self-learning genetic algorithm does not arrive at a good solution within the search process. Hence it arrives at a makespan of 208000. Greedy scheduling manages to fit everything in two days this time around, managing a total makespan of 132980. DRL-S ends at a total duration of 217050, just barely requiring to use of 3 days as well. After closer inspection, greedy scheduling introduces a total setup time of 260000 within the instance, whereas the SO-SLEGA and DRL-S introduce a total setup time of 460000 and 550000 respectively. This could cause this difference in performance. This is most likely due to the fact that the genetic algorithm does not mutate individuals well for setup times, as it greedily assigns a machine for which the processing time and setup time are minimal, disregarding the setup that is required after the mutated operation. The DRL agent did not learn enough about setup times generally speaking.

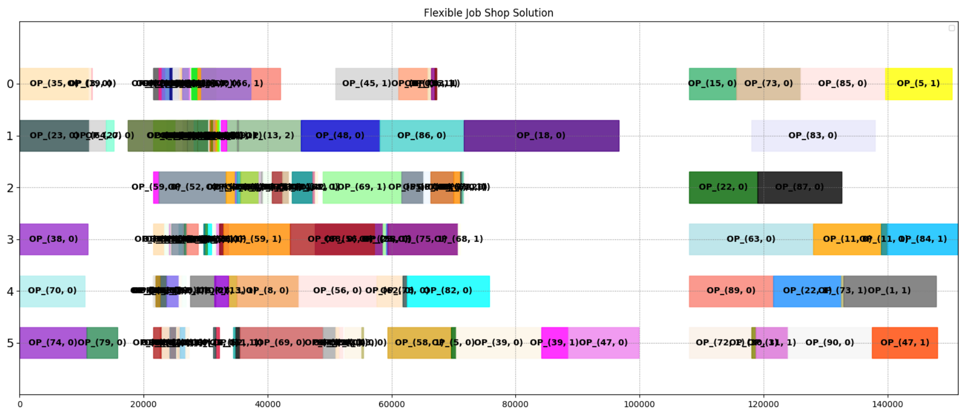
However, when looking at the SO-SLEGA trained on 64x04 (6.23c) and the SO-SLEGA trained on 88x08 (6.23d) FJSP instances, we note that it could occur that all operations are scheduled within a single day (makespan of 151365). However, this seems to happen because of randomness, not because of the training instances sizes. This is further investigated below.



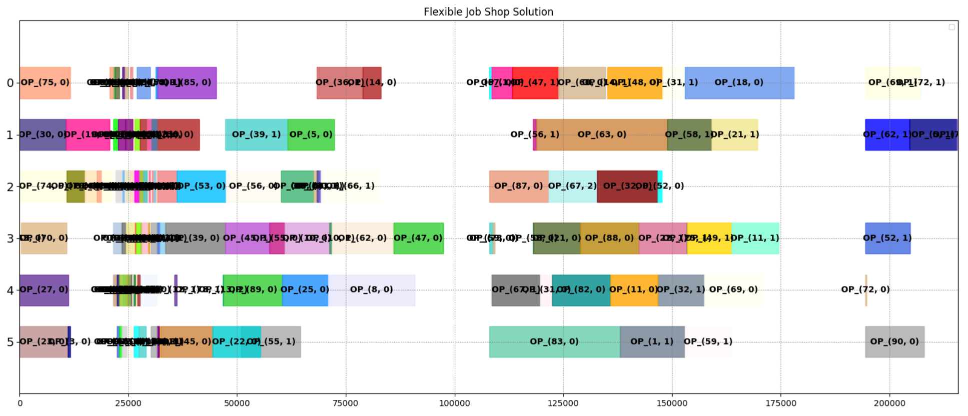
(a) Greedy Scheduling (132980)



(b) SO-SLEGA (17x02) (208000)



(c) SO-SLEGA (64x04) (151365)



(d) SO-SLEGA (88x08) (217050)

Figure 6.23: Test instance (85x06) schedules.

We deep-dive further on the instance of size 85x06 to investigate the generalizability of the algorithms and see if the performance of greedy scheduling can be equalled. When increasing the number of generations and population size from 100 to 500 for the SO-SLEGA, we manage to obtain a makespan of 131336 on the 85x06 instance. The algorithm took 232 seconds to execute. This increase in duration would still be acceptable for the company, as schedules can be created on an ad-hoc basis, and are rarely required to be available instantly. Figure 6.24 below shows the schedule that was found when increasing the number of generations and population size. We conclude that SO-SLEGA still outperforms greedy scheduling.

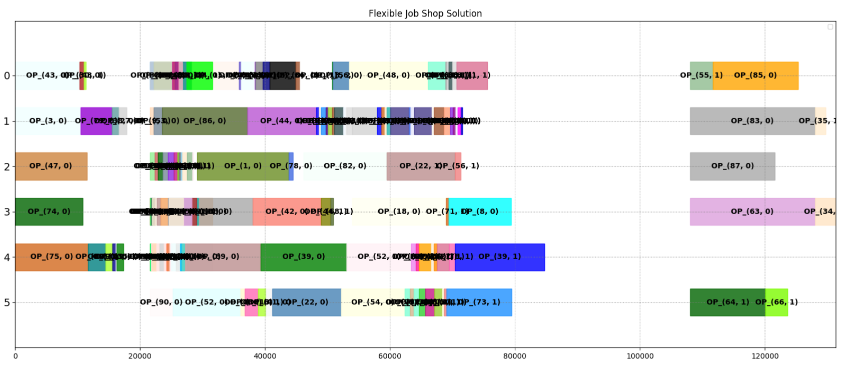


Figure 6.24: Schedule (131336) for SO-SLEGA (17x02) with 500 individuals and 500 generations.

6.3.3 Ablation Study

To validate the features on the heterogeneous graph proposed in Chapter 4, we conduct a small ablation study to identify the impact of adding these features. Table 6.9 below displays the results of this ablation study. Here, we show the average makespan for 5 different types of FJSP instances, from the vanilla to the company instances. For each of the instances, we report the average makespan across the company test set. Note that for each situation, some characteristics are disabled, hence the same could be dataset is used. We report the average makespan for (a) greedy scheduling, (b) the SO-SLEGA, (c) DRL-S without any added features, (d) DRL-S with the features for the instance type and (e) DRL-S with all proposed features. For example, in the instance of type 2 (FJSP with SDSTs), model (d) would only have the sequence-dependent setup times as an extra feature on the arcs, while model (e) would also have the night schedule and setup times. Δ displays the performance difference between the SO-SLEGA and DRL-S with stacking features.

Index	Instance Type	Average makespan GREEDY	SO-SLEGA	DRL-S	DRL-S stack. feat.	DRL-S all feat.	Δ
1	FJSP	48170	45051	45821	45821	46556	770 (1.7%)
2	(1) with SDSTs	102778	93992	119522	118148	128335	25530 (20.5%)
3	(2) with Release Dates	115135	103773	129749	126371	132535	22598 (17.9%)
4	(3) with Night Times	164040	138733	184627	175447	175447	36714 (20.9%)
5	(1) with Night Times	71760	64261	68276	67727	73552	3466 (5.4%)

Table 6.9: Ablation study for FJSP characteristics and E2E-DRL features.

From the table, we note that for the vanilla FJSP, the SO-SLEGA only outperforms DRL-S by 1.7%. Both DRL-S and SO-SLEGA outperform greedy scheduling for this instance type. As soon as sequence-dependent setup times are added, the DRL-S approach fails to outperform greedy scheduling for any other instance type. We note that adding features sequentially adds value for the agent to perform better at creating schedules. For example, adding night times reduces the average makespan from 68.3k to 67.7k for the instance types with index 5. For type 4, the average makespan is even decreased from 184.6k to 175.4k.

DRL-S still outperforms greedy scheduling for the instance type without SDSTs but with the addition of night times (index 5). However, the performance gain for the additional features is not significant. This is most likely due to the fact that most instances are solved within a day, not crossing the situation where this agent would be most beneficial. Adding all proposed features from the start is not reasonable, as the total performance is always lower than selecting features carefully based on the type of FJSP instance.

Since the SDSTs are higher than the processing times, they influence the final makespan significantly. So far, no discounting has been applied to the received rewards. This means the agent cares more for the distant reward (i.e., total makespan) than the immediate reward (mistake of setup times). This design choice made sense initially, as we purely focused on makespan and the impact of individual decisions was not as big. However, given the environment where immediate decisions are more important (high setups could be resulting), considering reward discounting might be actually beneficial. We test this hypothesis by retraining the DRL-S model with stacking features under the environment of type 2, with the discount rate reduced to 0.95, 0.5 and 0.25. The resulting average makespan for these settings was 116758, 109068 and 105151. This shows that increasing the discount rate can be beneficial, indicating the need to pay attention to immediate rewards rather than distant rewards. However, the performance is still not as good as the performance of the SO-SLEGA or the performance of greedy scheduling.

To summarize this experiment, we conclude that SO-SLEGA performs better than E2E-DRL for all company instances. SO-SLEGA outperforms most traditional heuristics too. However, for larger instances, a bigger population size and a higher number of generations are required to do so. This should not be a problem as the computation time is this acceptable. Hence, we also conclude that SO-SLEGA is better at generalizing to instances with unseen characteristics as well, as the SO-SLEGA model in Experiment 1 outperforms retrained DRL-S models. This indicates that in order to tackle different types of scheduling problems, a SO-SLEGA approach is more general and will reach higher performance. No conclusions are drawn on which model generalizes better to unseen instance sizes. From the ablation study, we have seen that E2E-DRL stops performing well as soon as sequence-dependent setup times are added. This is most likely because the immediate reward should be considered more important than the distance reward, and can be improved by considering a discounting rate.

6.4 Experiment 4. MO-GA vs. MO-SLEGA for Company-Specific Instances

For the fourth experiment, we transition towards a MOO setting. More specifically, we use the same training and testing instances generated in Experiment 3, but now instead of only considering the makespan, we also consider the cost of schedule incurred. The allocated cost of the schedule is calculated given the evaluation function described in Chapter 4.

6.4.1 Training and Validation

We follow the same training procedure for the SLEGA but now use the MOO state space, action space, reward function, and genetic operations for this setting are described in Section 4.1. Table 6.10 below displays the results of training. Please note again that MO-GA refers to the SLEGA without the self-learning module, as explained in Chapter 4.

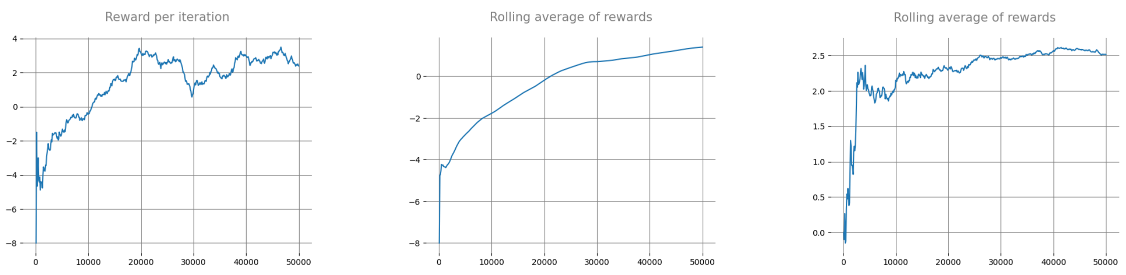
From the first three experiments, we have seen that the instance size on which the model was trained was important. More specifically, in Experiment 3, the models trained on the 17x02 instances performed best. Hence in this experiment, we train a single model only using this type of instance. We do so as training duration for MOO instances took a significant amount of time longer than training in an SOO setting. This execution time increase is explained by the following. The evaluation function in MOO-setting is harder to execute, as computing the cost is much more complex than the makespan given the different components. Furthermore, rather than maintaining a single individual in the hall of fame, we need to maintain the Pareto front as well. Furthermore, the state space is harder to compute, especially the calculation of the hypervolume.

Instances	Training		Validation			
	Duration	Timestep	Inference Time	Best Makespan	Best Cost	Hypervolume
MO-SLEGA	24.01h	14800	22.60s	95105	3173	$1.8e10$
MO-GA	-	-	7.82s	105324	3772	$1.6e10$

Table 6.10: Training and Validation of Experiment 4.

From Table 6.10, we see that training of the MO-SLEGA model took just over a full day. The best validation performance was found in timestep 14800. Figure 6.25 displays the learning and validation curves. As can be seen, the model is indeed learning how to set the hyperparameters in order to obtain a higher reward. The validation performance follows the training curve and stabilises at a reward of about 2.5.

On the validation dataset, the MO-SLEGA outperforms MO-GA. More specifically, the average best makespan found is 9.5% better, the average best cost is 15.9% better, and the hypervolume is 12.5% better for the MO-SLEGA. Hence we conclude that the model has learned well. Figure 6.26 below displays an example Pareto front found by MO-SLEGA and MO-GA. From the Figure, we can indeed conclude the front found by MO-SLEGA is better than the Pareto front found by the MO-GA.



(a) Training rewards. (b) Training reward rolling average. (c) Validation reward rolling average.

Figure 6.25: Training and validation curves of MO-SLEGA model on 17x02 instances.

For testing purposes, we take the same testing dataset as in the Experiment. We now compare the trained MO-SLEGA against the trained MO-SLEGA from Experiment 1, the vanilla GA implementation (i.e., described in Section 4.2.4). Note that we consider hyperparameter values (crossover rate etc.) from G. Zhang et al. (2011).

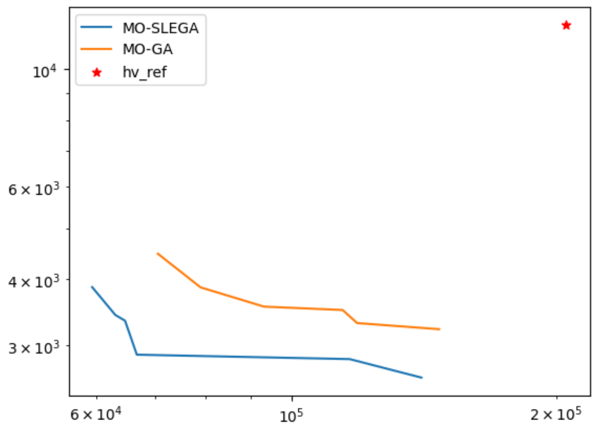


Figure 6.26: Example Validation Pareto front difference between MO-SLEGA and MO-GA.

6.4.2 Testing

Now our MO-SLEGA is trained on company-specific instances, we put it to the test. Table 6.11 below contains the results for the various algorithms on the test instances. For each approach, we present the average duration, the average hypervolume, the mean and median makespan and cost (i.e., mean of best makespan/cost of the Pareto front), and the optimality gap for the average makespan. The median is presented as the mean is sensitive to outliers, which were present in the case of the MO-GA for the test instances.

Approach	Duration	Hypervolume	Makespan			Cost	
			Mean	Median	Gopt	Mean	Median
RANDOM	20.00	3.5e11	192144	152982	44.25%	746934	244174
MO-GA	34.45	5.2e11	184684	152445	38.65%	171733	11648
MO-SLEGA	89.46	6.8e11	158348	139793	19.78%	14094	8835
SO-SLEGA[1]	35.92	-	138733	130482	4.96%	-	-

[1] Models trained Experiment 3.

Table 6.11: Results of models on company test set in MOO setting.

From the table, we note the following. First, we note that the average inference duration of the SLEGA has increased from 35.92 to 89.46 (147%) when switching from a single- to multi-objective setting. This is still acceptable to the company so this is not a big deal. Furthermore, we note that the MO-SLEGA achieves the highest average hypervolume of $6.8e11$. The vanilla MO-GA and random scheduling only achieve a hypervolume of $5.2e11$ (-24%) and $3.5e11$ (-48%) respectively.

In terms of makespan, the MO-SLEGA degrades in performance by approximately 15 percentage points when comparing it against the SO-SLEGA. Of course, when considering multiple objectives, the algorithm won't be as good at finding the best makespan. For example, various mutation actions are taken to improve the cost, which will reduce the makespan on the other hand. Compared to the vanilla genetic algorithm, adding a self-learning module allows for makespan to be improved from 184684 to 158358 (17% improvement). In Section 4.3.2 we have seen that the agent sets the individual and gene mutation rates quite high, which is not the case for the vanilla genetic algorithm. Random scheduling is slightly worse than the genetic algorithm with regard to makespan so the same conclusions hold.

In terms of cost, the MO-SLEGA improves performance by 92% when looking at the averages. When looking at median cost, this is only an improvement of 24%. This large difference is explained by the outliers in the best cost performance found by the genetic algorithm. The outliers in cost are caused by the exponentially increasing WIP cost.

Figure 6.27 shows the results in terms of hypervolume over the different instances. The results are pretty self-explanatory. When instance sizes are not small (i.e., more than 10 jobs and 2 machines), MO-SLEGA achieves the highest hypervolume at all times. For smaller instances, the MO-GA might outperform the MO-SLEGA. This difference might be explained by the low mutation rate and higher crossover rate. The performance difference is minimal (2-3% at most). Even though this small difference is caused by a loose reference point, we consider it negligible. From Figure 6.28c we note that improvements in hypervolume when considering the MO-SLEGA over MO-GA can be up to 170%. Generally speaking, improvements are much higher when instance sizes are larger.

Looking at Figures 6.28a and 6.28b, we note that makespan performance can increase up until 51% and cost improvements can be up until 99%. For cost improvements, we note that they generally occur for larger instances, while for makespan improvements this isn't necessarily true. The large cost improvements again are explained by the exponential cost incurred due to WIP inventory.

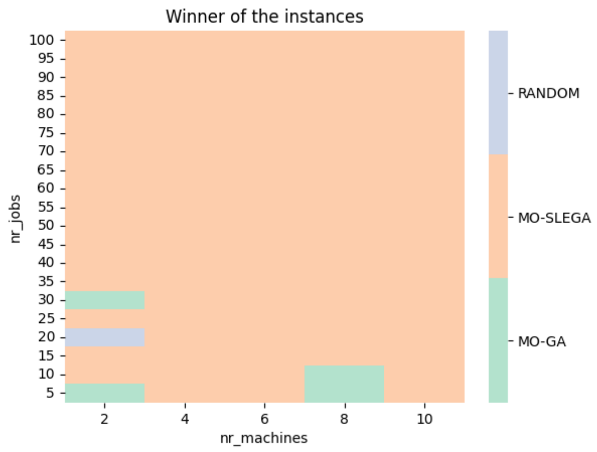
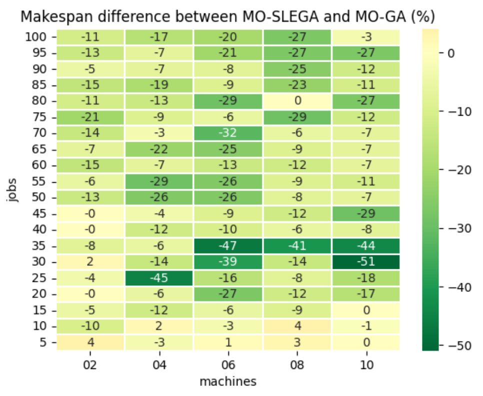
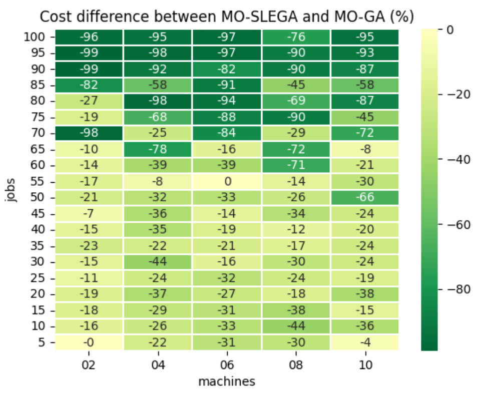


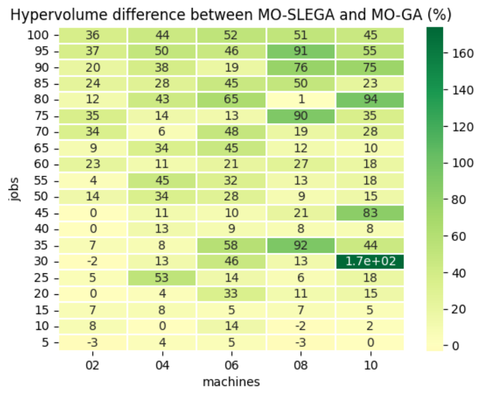
Figure 6.27: Approach with the highest hypervolume per test instance.



(a) Makespan



(b) Cost



(c) Hypervolume

Figure 6.28: Comparison between MO-SLEGA and MO-GA

Figure 6.29 displays an example of found Pareto fronts by the different benchmarked approaches. As can be seen, the performance displayed in Table 6.11 is confirmed. More specifically, we note that the MO-SLEGA finds points where the cost is lower while obtaining a lower makespan as well. We note that the reference point is not picked properly, as the lowest cost achieved goes beyond the reference point. This is not considered problematic as from the Figure we note that MO-SLEGA still outperforms the other algorithms for this region, thus it would not change the observations.

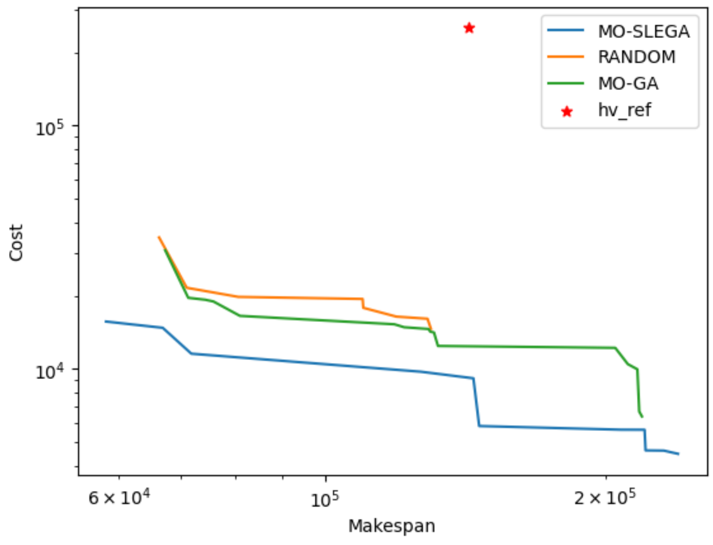


Figure 6.29: Pareto front of 30x08 FJSP instance.

Now we displayed how and where the MO-SLEGA performs better, we dive into why this has happened. Table 6.12 below displays the breakdown of incurred cost of 4 schedules created on the 85x06 instance. The 4 schedules are created using (1) the schedule on the bottom-left (eyeballed) of the Pareto front found using MO-SLEGA (Figure 6.30a), (2) the schedule with the lowest cost found using MO-SLEGA (Figure 6.30b, (3) the schedule found using SO-SLEGA with 500 generations and 500 individuals (Figure 6.24) and (4) the schedule found using the original SO-SLEGA train on instances of size 64x04 (Figure 6.23c).

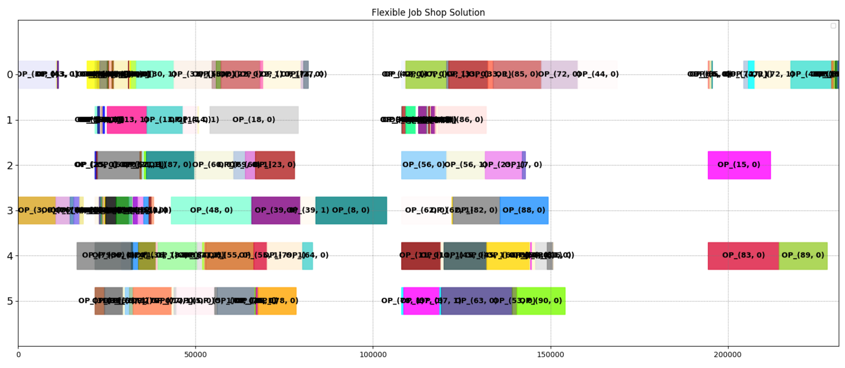
From the table, we note that the main cost incurred in schedules is due to WIP inventory. This makes sense as this wasn't penalized at all in the SOO setting. The cost incurred for deadlines is much higher though for the MOO setting. This also makes sense, because when optimizing for the complete cost picture, we increase the makespan, and miss more deadlines. The cost of deadlines in this case did not weigh up against the cost of WIP inventory. More specifically, the cost of missing deadlines did not weigh up against the cost of addiction and manual labour too, as in the worst case it only makes up 2.8% of the total cost.

For the situation with the best cost, we also notice that we manage to decrease the cost of resource addition through the created mutation function. This cost namely decreases from about 23k to 18k for the cheapest schedule. The finished product costs are negligible. We note that for the MOO setting, the setup total setup times are extremely high. This means that the algorithm makes the additional setups required to decrease the costs.

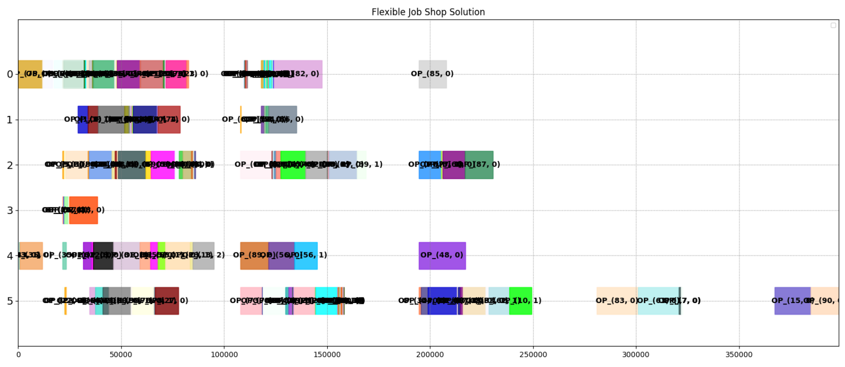
To summarize this experiment, we conclude that for a MOO setting, the MO-SLEGA performs better than a vanilla MO-GA and random scheduling. We also see that makespan performance did not decrease by more than 15 percentage points, while also accounting for the cost objective in the meantime. We also note that the cost function could require a redesign in order to account for the various cost components more equally.

Component	Schedule			
	MO-SLEGA (ideal point)	MO-SLEGA (best cost)	SO-SLEGA (17x02)	SO-SLEGA (64x05)
Figure	6.30a	6.30b	6.24	6.23c
Makespan	231278	398390	131336	151365
Setup Time	460000	460000	220000	370000
Cost	27056	22142	593282	390997
Addition	22850	17610	23390	33082
FP	34	32	36	36
WIP	201	0	565714	352430
Deadlines	350	600	75	150
Manual	3572	3900	3068	3299

Table 6.12: Schedule objective breakdowns.



(a) Best Schedule (231278, 27056)



(b) Cheapest Schedule (39890, 22142)

Figure 6.30: Test instance (85x06) schedules created using MO-SLEGA.

Chapter 7

Conclusion and Recommendations

This research project analyzes various scheduling algorithms on the flexible job-shop scheduling problem (FJSP), which consists of machine allocation and operation sequencing sub-problems. In addition to the vanilla FJSP, we investigate two variants: one with sequence-dependent setup times (SDSTs) and another customized for Wefabricate, including SDSTs, night times, maintenance jobs, resources, release dates, deadlines, inventory cost, and dynamic events. A literature review highlights previous approaches, their limitations, and gaps, leading to our benchmarking study.

This research, commissioned by Wefabricate, aims to find scalable, flexible, efficient, generalizable, and robust scheduling solutions. The approach should quickly find a near-optimal schedule and maintain performance when input instances change. Additionally, it should adapt to new environments and processes. This chapter presents conclusions drawn from the experiments conducted and provides recommendations for Wefabricate.

7.1 Conclusion

In this research, we use a single evaluation function to assess the schedules produced by the algorithms. This function is implemented twice to simulate schedules, one for decoding individuals of the (SLE)GA, and one for E2E-DRL. Both implementations follow the same logic and return objective values that are considered the standard for evaluating any schedule executed within the factory. All the conclusions drawn in this study are based on the results obtained using this evaluation function.

We propose two algorithms for scheduling: a self-learning effective genetic algorithm (SLEGA) approach and an end-to-end deep reinforcement learning (E2E-DRL) approach. In the SLEGA approach, individuals are formulated using a double-layer encoding of operation sequence and machine allocation strings, which are then decoded into feasible schedules. Population initialization involves local, global, and random selection, while selection uses tournament selection (SO-SLEGA) and NSGA-II (MO-SLEGA). Crossover is performed using precedence-preserving order-based crossover (POX), two-point crossover, and uniform crossover, while mutation employs various greedy makespan (SO-SLEGA) and cost (MO-SLEGA) mutations. To speed up the search process, individual evaluations are parallelized. The hyper-parameters of search (crossover rate, individual mutation rate, gene mutation rate) are then optimized by a PPO agent during the evolutionary process. The E2E-DRL approach formulates a schedule instance as a heterogeneous graph with operation and machine nodes, where the arcs represent operation-machine allocation decisions. Node and arc features are embedded using a heterogeneous graph neural network, and the embedding is used by a PPO agent to select an operation-machine pair for scheduling.

In Experiment One, we have seen that in the vanilla FJSP, the instance size is of importance when selecting a scheduling approach according to its performance (i.e., makespan). More specifically, we saw that E2E-DRL outperformed SO-SLEGA for instances which have relatively few good solutions and a larger search space. In those cases, the makespan was reduced by up to 30%. For instances where the number of good solutions was relatively higher, the SO-SLEGA outperformed E2E-DRL and reduced makespan up to 20%. In terms of execution times, E2E-DRL was faster by 20 to 90%, where execution time gain was less for larger instances. Hence, for the vanilla FJSP problem, we consider both algorithms to be scalable and efficient, where the E2E-DRL approach is more efficient than the SO-SLEGA approach.

In Experiment Two, we compared the performance of two approaches for solving the FJSP with SDSTs. Our results showed that the SO-SLEGA approach outperformed E2E-DRL in terms of makespan, with a 5% improvement. Moreover, we found that retraining improved the performance of E2E-DRL, while the SO-SLEGA approach demonstrated better generalization to unseen characteristics, making it more flexible. In terms of efficiency, E2E-DRL was significantly faster, although the execution time of the SO-SLEGA approach was still within acceptable limits. Overall, our findings suggest that the SO-SLEGA approach is a more reliable and flexible option for solving the FJSP with SDSTs, whereas E2E-DRL is a faster but less adaptable approach.

In Experiment Three, we investigated the performance of two approaches for solving the highly constrained FJSP of Wefabricate. Our results demonstrated that the SO-SLEGA approach outperformed E2E-DRL in terms of makespan, even when considering new and previously unseen instance sizes. This suggests that the SO-SLEGA approach is more flexible and robust to changes in problem characteristics. Moreover, we found that the performance of the SO-SLEGA approach could be further improved by increasing the population size and the number of generations, although this trade-off with efficiency should be considered. Nevertheless, our results showed that the SO-SLEGA approach was able to outperform traditional heuristics while still achieving acceptable execution times. Note that in this experiment, E2E-DRL did not manage to outperform random scheduling. Overall, these findings indicate that the SO-SLEGA approach is a reliable and effective option for solving the highly constrained FJSP of Wefabricate.

In Experiment Four, we tested the performance of the MO-SLEGA approach on the highly-constrained FJSP in a multi-objective setting. While we did not attempt to test E2E-DRL in this context due to the extensive work required, our results demonstrated that the MO-SLEGA approach was able to handle the multi-objective nature of the problem. These findings further emphasize the flexibility of the MO-SLEGA scheduling approach. Our experiments also revealed that incorporating a self-learning module into the MO-SLEGA approach improved its performance when compared to a vanilla genetic algorithm. Moreover, the algorithm remained efficient enough to be used in production. Overall, these results suggest that the MO-SLEGA approach is a robust and adaptable option for handling the highly-constrained FJSP in a multi-objective setting, and its performance can be further enhanced through the incorporation of self-learning modules.

Based on our conclusions from the previous paragraphs, we recommend that Wefabricate adopt the SLEGA algorithm as a standard scheduling approach for various operation scheduling problems when integrating machine learning into job scheduling. This is because the SLEGA approach outperforms E2E-DRL in terms of flexibility, performance, and generalization. Specifically, the SLEGA approach can handle various constraints and objectives more effectively, achieve lower makespan and cost, and require less retraining than E2E-DRL. Additionally, the SLEGA approach is efficient enough to achieve acceptable execution times, is highly scalable through parallelization, and can adapt to dynamic events during schedule execution. The evaluation function can be controlled to adjust the preferences of the company, such as buffers and working hour capacity, and the SLEGA approach can learn the setting of hyperparameters based on the given evaluation function. Overall, the SLEGA approach is a generic and reliable option for machine learning-based job scheduling at Wefabricate. This answers our main research question.

With the SLEGA scheduling approach, onboarding a new process only requires creating an evaluation function and potentially genetic operations. The algorithm remains the same, unlike the E2E-DRL approach, which would require significant effort. Additionally, the SLEGA approach can be applied to various optimization problems as long as a solution can be represented as an individual and this individual can be evaluated given the business context.

7.2 Limitations and Recommendations

In this work, we investigated two main algorithms for solving the flexible job shop scheduling problem. Although this work laid a foundation towards generic job scheduling, this work is not complete. In this work, we took various shortcuts which future researchers are recommended to look into:

In our implementation of E2E-DRL, sequence-dependent setup times were represented as a distinct feature on the arcs of a heterogeneous graph. Alternatively, it may be possible to combine the setup time and processing time of each O-M pair into a single feature. This approach may prevent catastrophic interference during the learning process. Additionally, it would be beneficial to investigate the root cause of the catastrophic interference and identify potential solutions. Furthermore, the E2E-DRL approach needs a lot more exploration and fine-tuning for the company-specific instances.

During our comparison, we did not extensively tune the hyperparameters of the learning algorithms (specifically PPO), which could potentially improve training times and outcomes. Additionally, we did not tune the hyperparameters of the genetic algorithm, which can also have a significant impact on the learning process. The number of generations or the population size could be interesting to look at.

To address dynamic events, it would be valuable to investigate the impact of a "warm start" on the optimization speed of the SLEGA algorithm. This would enhance the flexibility of SLEGA and reduce the time needed to recompute solutions, ultimately leading to faster execution times.

Our multi-objective approach revealed that the best makespan identified on the Pareto front was not as optimal as the one obtained through single-objective optimization. To address this, it would be worthwhile to investigate potential modifications to the multi-objective optimization search, with the goal of also identifying the solutions found through single-objective optimization.

To evaluate the performance of the multi-objective approach more accurately, it would be worthwhile to compute several actual Pareto fronts from real-world instances provided by the company. This would provide a more accurate measure of how closely the found Pareto fronts align with the optimal solutions and would offer a more realistic assessment of algorithm performance.

Our approach involved taking a fixed set of jobs as input, with a fixed execution quantity equal to the total amount required by the customer. This meant that during the optimization process, there was no consideration given to the possibility of splitting a single job into smaller ones. It would be worthwhile to investigate how job-splitting decisions can be incorporated into the optimization process, to introduce greater flexibility and improve scheduling outcomes.

For Wefabricate, it is important to fine-tune the evaluation function. Following the garbage-in garbage-out principle, the evaluation function is of uttermost importance in order to arrive at proper schedules. Hence the cost function should be carefully investigated and improved where possible. This could be done by including the dimensions of products in order to represent the actual WIP cost for example.

Bibliography

Abreu, L. R., Cunha, J. O., Prata, B. A., & Framinan, J. M. (2020). A genetic algorithm for scheduling open shops with sequence-dependent setup times. *Computers & Operations Research*, *113*, 104793. https://doi.org/https://doi.org/10.1016/j.cor.2019.104793

Al-harkan, I. M., & Qamhan, A. A. (2019). Optimize unrelated parallel machines scheduling problems with multiple limited additional resources, sequence-dependent setup times and release date constraints. *IEEE Access*, *7*, 171533–171547. https://doi.org/10.1109/ACCESS.2019.2955975

Arora, S. (1998). Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems. *J. ACM*, *45*(5), 753–782. https://doi.org/10.1145/290179.290180

Aydin, M., & Öztemel, E. (2000). Dynamic job-shop scheduling using reinforcement learning agents. *Robotics and Autonomous Systems*, *33*(2), 169–178. https://doi.org/https://doi.org/10.1016/S0921-8890(00)00087-7

Behnke, D., & Geiger, M. J. (2012). Test instances for the flexible job shop scheduling problem with work centers.

Bissoli, D., & Amaral, A. R. S. (2018). A hybrid iterated local search metaheuristic for the flexible job shop scheduling problem. *2018 XLIV Latin American Computer Conference (CLEI)*, 149–157. https://doi.org/10.1109/CLEI.2018.00026

Blackstone, J. H., Philips, D. T., & Hogg, G. L. (1982). A state-of-the-art survey of dispatching rules for manufacturing job shop operations. *International Journal of Production Research*, *20*(1), 27–45. https://doi.org/10.1080/00207548208947745

Brandimarte, P. (1993). Routing and scheduling in a flexible job shop by tabu search. *Annals of Operations Research*, *41*, 157–183.

Burke, E. K., & Graham, K. (2014). *Search methodologies: Introductory tutorials in optimization and decision support techniques, second edition.* https://doi.org/10.1007/978-1-4614-6940-7

Cao, Y., Smucker, B. J., & Robinson, T. J. (2015). On using the hypervolume indicator to compare pareto fronts: Applications to multi-criteria optimal experimental design. *Journal of Statistical Planning and Inference*, *160*, 60–74. https://doi.org/https://doi.org/10.1016/j.jspi.2014.12.004

Chan, F. T. S., Wong, T. C., & Chan, L. Y. (2006). Flexible job-shop scheduling problem under resource constraints. *International Journal of Production Research*, *44*(11), 2071–2089. https://doi.org/10.1080/00207540500386012

Chang, J., Yu, D., Hu, Y., He, W., & Yu, H. (2022). Deep reinforcement learning for dynamic flexible job shop scheduling with random job arrival. *Processes*, *10*(4). https://doi.org/10.3390/pr10040760

Chen, J. C., Wu, C.-C., Chen, C.-W., & Chen, K.-H. (2012). Flexible job shop scheduling with parallel machines using genetic algorithm and grouping genetic algorithm. *Expert Systems with Applications*, *39*(11), 10016–10021. https://doi.org/https://doi.org/10.1016/j.eswa.2012.01.211

Chen, R., Yang, B., Li, S., & Wang, S. (2020). A self-learning genetic algorithm based on reinforcement learning for flexible job-shop scheduling problem. *Computers & Industrial Engineering*, *149*, 106778. https://doi.org/https://doi.org/10.1016/j.cie.2020.106778

Crowston, W. B. S., Glover, F. W., Thompson, G. L., & Trawick, J. D. (1963). Probabilistic and parametric learning combinations of local job shop scheduling rules.

Dai, M., Tang, D., Giret, A., & Salido, M. A. (2019). Multi-objective optimization for energy-efficient flexible job shop scheduling problem with transportation constraints. *Robotics and Computer-Integrated Manufacturing*, *59*, 143–157. https://doi.org/https://doi.org/10.1016/j.rcim.2019.04.006

Deb, K., Pratap, A., Agarwal, S., & Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, *6*(2), 182–197. https://doi.org/10.1109/4235.996017

Fleischer, L., Goemans, M., Mirrokni, V., & Sviridenko, M. (2006). Tight approximation algorithms for maximum general assignment problems. *Mathematics of Operations Research*, *36*, 611–620. https://doi.org/10.1145/1109557.1109624

Fonseca, C. M., Lopez-Ibanez, M., Paquete, L., & Guerreiro, A. P. (2017). Computation of the hypervolume indicator.

Glover, F. (1986). Future paths for integer programming and links to artificial intelligence [Applications of Integer Programming]. *Computers & Operations Research*, *13*(5), 533–549. https://doi.org/https://doi.org/10.1016/0305-0548(86)90048-1

Graham, R. L., Lawler, E. L., Lenstra, J. K., & Kan, A. H. (1979). Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, *5*(100). https://doi.org/10.1016/S0167-5060(08)70356-X

Guerreiro, A. P., Fonseca, C. M., & Paquete, L. (2021). The hypervolume indicator. *ACM Computing Surveys*, *54*(6), 1–42. https://doi.org/10.1145/3453474

Hamzadayi, A., & Yildiz, G. (2016). Event driven strategy based complete rescheduling approaches for dynamic m identical parallel machines scheduling problem with a common server. *Computers and Industrial Engineering*, *91*, 66–84. https://doi.org/10.1016/j.cie.2015.11.005

Holland, J. H. (1992). *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control and artificial intelligence*. MIT Press.

Hurink, J., Jurisch, B., & Thole, M. (1994). Tabu search for the job-shop scheduling problem with multi-purpose machines. *OR Spectrum = OR Spektrum*, *15*(4), 205–215. https://doi.org/10.1007/BF01719451

Jansen, K., & Mastrolilli, M. (2004). Approximation schemes for parallel machine scheduling problems with controllable processing times. *Computers and Operations Research*, *31*(10), 1565–1581. https://doi.org/10.1016/S0305-0548(03)00101-1

Jansen, K., Mastrolilli, M., & Solis-Oba, R. (2000). Approximation algorithms for flexible job shop problems. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *1776 LNCS*, 68–77. https://doi.org/10.1007/10719839{_}7

Kaplan, S., & Rabadi, G. (2012). Exact and heuristic algorithms for the aerial refueling parallel machine scheduling problem with due date-to-deadline window and ready times. *Computers and Industrial Engineering*, *62*(1), 276–285. https://doi.org/10.1016/j.cie.2011.09.015

Karimi-Mamaghan, M., Mohammadi, M., Meyer, P., Karimi-Mamaghan, A. M., & Talbi, E.-G. (2022). Machine learning at the service of meta-heuristics for solving combinatorial optimization problems: A state-of-the-art. *European Journal of Operational Research*, *296*(2), 393–422. https://doi.org/https://doi.org/10.1016/j.ejor.2021.04.032

Korte, B. H., & Vygen, J. (2012). *Combinatorial optimization: Theory and algorithms*. Springer-Verlag. https://doi.org/10.1007/978-3-642-24488-9

Koulamas, C. (1998). A new constructive heuristic for the flowshop scheduling problem. *European Journal of Operational Research*, *105*(1), 66–71. https://doi.org/https://doi.org/10.1016/S0377-2217(97)00027-1

Lawrence, S. (1984). Resouce constrained project scheduling : An experimental investigation of heuristic scheduling techniques (supplement). *Graduate School of Industrial Administration, Carnegie-Mellon University*. https://cir.nii.ac.jp/crid/1571980073974705920

Lee, K.-M., Yamakawa, T., & Lee, K.-M. (1998). A genetic algorithm for general machine scheduling problems. *1998 Second International Conference. Knowledge-Based Intelligent Electronic Systems. Proceedings KES'98 (Cat. No.98EX111)*, *2*, 60–66 vol.2. https://doi.org/10.1109/KES.1998.725893

Lei, D., & Yang, H. (2022). Scheduling unrelated parallel machines with preventive maintenance and setup time: Multi-sub-colony artificial bee colony. *Applied Soft Computing*, *125*, 109154. https://doi.org/10.1016/J.ASOC.2022.109154

Liang, X., Chen, J., Gu, X., & Huang, M. (2021). Improved adaptive non-dominated sorting genetic algorithm with elite strategy for solving multi-objective flexible job-shop scheduling problem. *IEEE Access*, *9*, 106352–106362. https://doi.org/10.1109/ACCESS.2021.3098823

Liu, Y., Wei, J., Li, X., & Li, M. (2019). Generational distance indicator-based evolutionary algorithm with an improved niching method for many-objective optimization problems. *IEEE Access*, *7*, 63881–63891. https://doi.org/10.1109/ACCESS.2019.2916634

Lunardi, W. T. (2020). A real-world flexible job shop scheduling problem with sequencing flexibility: Mathematical programming, constraint programming, and metaheuristics.

Marler, R. T., & Arora, J. S. (2004). Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization*, *26*(6), 369–395. https://doi.org/10.1007/s00158-003-0368-6

Özbakir, L., Baykasoğlu, A., & Tapkan, P. (2010). Bees algorithm for generalized assignment problem. *Applied Mathematics and Computation*, *215*(11), 3782–3795. https://doi.org/https://doi.org/10.1016/j.amc.2009.11.018

Özgüven, C., Özbakır, L., & Yavuz, Y. (2010). Mathematical models for job-shop scheduling problems with routing and process plan flexibility. *Applied Mathematical Modelling*, *34*(6), 1539–1548. https://doi.org/https://doi.org/10.1016/j.apm.2009.09.002

Potts, C. N. (1980). An algorithm for the single machine sequencing problem with precedence constraints. In V. J. Rayward-Smith (Ed.), *Combinatorial optimization ii* (pp. 78–87). Springer Berlin Heidelberg. https://doi.org/10.1007/BFb0120909

Rahnamayan, S., Tizhoosh, H. R., & Salama, M. M. (2007). A novel population initialization method for accelerating evolutionary algorithms. *Computers and Mathematics with Applications*, *53*(10), 1605–1614. https://doi.org/https://doi.org/10.1016/j.camwa.2006.07.013

Rooyani, D., & Defersha, F. M. (2019). An efficient two-stage genetic algorithm for flexible job-shop scheduling ∗∗this research is funded by the natural science and engineering research counsel (nserc) of canada [9th IFAC Conference on Manufacturing Modelling, Management and Control MIM 2019]. *IFAC-PapersOnLine*, *52*(13), 2519–2524. https://doi.org/https://doi.org/10.1016/j.ifacol.2019.11.585

Saidi-Mehrabad, M., & Fattahi, P. (2007). Flexible job shop scheduling with tabu search algorithms [Copyright - The International Journal of Advanced Manufacturing Technology is a copyright of Springer, (2006). All Rights Reserved]. *The International Journal of Advanced Manufacturing Technology*, *32*(5-6), 563–570. https://www.proquest.com/scholarly-journals/flexible-job-shop-scheduling-with-tabu-search/docview/2262523871/se-2

Santos, T., & Xavier, S. (2018). A convergence indicator for multi-objective optimisation algorithms. *TEMA*, *19*, 437–448. https://doi.org/10.5540/tema.2018.019.03.0437

Schulman, J., Levine, S., Abbeel, P., Jordan, M., & Moritz, P. (2015). Trust region policy optimization. In F. Bach & D. Blei (Eds.), *Proceedings of the 32nd international conference on machine learning* (pp. 1889–1897). PMLR. https://proceedings.mlr.press/v37/schulman15.html

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms.

Sekkal, N., & Belkaid, F. (2020). A multi-objective simulated annealing to solve an identical parallel machine scheduling problem with deterioration effect and resources consumption constraints. *Journal of Combinatorial Optimization*, *40*(3), 660–696. https://doi.org/10.1007/s10878-020-00607-y

Shahsavari-Pour, N., & Ghasemishabankareh, B. (2013). A novel hybrid meta-heuristic algorithm for solving multi objective flexible job shop scheduling. *Journal of Manufacturing Systems*, *32*(4), 771–780. https://doi.org/https://doi.org/10.1016/j.jmsy.2013.04.015

Song, W., Chen, X., Li, Q., & Cao, Z. (2022). Flexible job shop scheduling via graph neural network and deep reinforcement learning. *IEEE Transactions on Industrial Informatics*, 1–11. https://doi.org/10.1109/TII.2022.3189725

Sörensen, K., Duque, P., Vanovermeire, C., & Castro, M. (2012). Metaheuristics for the multimodal optimization of hazmat transports. https://doi.org/10.1002/9783527664818.ch10

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

Talbi & El-Ghazali. (2009). An effective genetic algorithm for the flexible job-shop scheduling problem. *Design to Implementation*, *74*. https://doi.org/https://doi.org/10.1016/j.eswa.2010.08.145

Visutarrom, T., Chiang, T.-C., Konak, A., & Kulturel-Konak, S. (2020). Reinforcement learning-based differential evolution for solving economic dispatch problems. *2020 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*, 913–917. https://doi.org/10.1109/IEEM45057.2020.9309983

Wang, M., & Pan, G. (2019). A Novel Imperialist Competitive Algorithm with Multi-Elite Individuals Guidance for Multi-Object Unrelated Parallel Machine Scheduling Problem. *IEEE Access*, *7*, 121223–121235. https://doi.org/10.1109/ACCESS.2019.2937747

Yamada, T., & Nakano, R. (1992). A genetic algorithm applicable to large-scale job-shop problems. *Parallel Problem Solving from Nature*, *2*, 283–292.

Ying, K. C., & Cheng, H. M. (2010). Dynamic parallel machine scheduling with sequence-dependent setup times using an iterated greedy heuristic. *Expert Systems with Applications*, *37*(4), 2848–2852. https://doi.org/10.1016/j.eswa.2009.09.006

Yuan, Y., & Xu, H. (2013). Flexible job shop scheduling using hybrid differential evolution algorithms. *Computers & Industrial Engineering*, *65*, 246–260. https://doi.org/10.1016/j.cie.2013.02.022

Zhang, C., Song, W., Cao, Z., Zhang, J., Tan, P. S., & Xu, C. (2020). Learning to dispatch for job shop scheduling via deep reinforcement learning.

Zhang, G., Gao, L., & Shi, Y. (2011). An effective genetic algorithm for the flexible job-shop scheduling problem. *Expert Systems with Applications*, *38*(4), 3563–3573. https://doi.org/https://doi.org/10.1016/j.eswa.2010.08.145

Zhu, W., & Tianyu, L. (2019). A novel multi-objective scheduling method for energy based unrelated parallel machines with auxiliary resource constraints. *IEEE Access*, *7*, 168688–168699. https://doi.org/10.1109/ACCESS.2019.2954601