SOLVING LARGE JOB SHOP SCHEDULING PROBLEMS: USING GRAPH CLASSIFICATION VIA GRAPH NEURAL NETWORKS TO PRE-SEED A GENETIC ALGORITHM FOR MACHINE DISPATCHING RULE OPTIMIZATION

by

Isaac Schwab

A Dissertation Submitted in

Partial Fulfilment of the

Requirements of the Degree of

Doctor of Philosophy

in Engineering

at

The University of Wisconsin-Milwaukee

August 2024

ABSTRACT

SOLVING LARGE JOB SHOP SCHEDULING PROBLEMS: USING GRAPH CLASSIFICATION VIA GRAPH NEURAL NETWORKS TO PRE-SEED A GENETIC ALGORITHM FOR MACHINE DISPATCHING RULE OPTIMIZATION

by

Isaac Schwab

The University of Wisconsin-Milwaukee, 2024
Under the Supervision of Professor Dr. Matthew Petering

The job shop scheduling problem is a difficult problem to solve, and it is also difficult to implement solutions found in research into real shops. In this research, a methodology is proposed to develop schedules for real shops. The methodology utilizes a genetic algorithm to select dispatching rules for each machine cell and accesses these schedules through a simulation optimization framework. The simulation framework allows for the study of random elements including variable job processing times and random machine breakdowns. This creates a robust schedule that is easy to understand, and therefore implement, while scaling to large, real-world job shops. To gain additional efficiencies, a novel methodology is proposed to classify the graphs which represent different types of shop environments, with a graph neural network, to pre-seed the initial population of the genetic algorithm. This process allows the system to leverage pre-existing knowledge of similar shops to reduce the number of generations required to reach a reasonable solution.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

My first job after college was as a mechanical engineer where I designed jet engine components. My specialty was in optimizing the heat treat process to deliver parts more quickly. One day, I presented to the director of engineering some computational fluid dynamic modeling I was doing focusing on the heat treat process. I was convinced we could shave off 3 minutes on quench times. I projected this out, outlining a savings in capacity of the heat treat cell of around 2%, which over the year more than paid for my salary. He informed me that a large portion of our machine shop sat idle 40% of the time, yet was projected at being near capacity due to poor planning. If I could fix that, there would be real savings.

This problem got me interested in the job shop scheduling problem, which seemed as simple as enumerating each possible dispatching option and seeing which was best. As I continued to explore this space and the work, I realized this wasn't as simple as I initially thought. This inspired me to explore various projects. I worked as our new part introduction coordinator where I helped over 200 new parts through engineering, the first production run, and addressing initial quality issues. I was a process leader in continuous improvement, where I worked to improve operational efficiency using Lean manufacturing principles and optimize scheduling. Ultimately, I left the company as the challenge became how to convince management, and less on how to best schedule.

I then started working at the IRS as an operations research analyst in Research Analytics and Applied Statistics (RAAS). From there, I have worked my way to my current role as Chief of the Asset Discovery Lab. My team identifies large-scale tax avoidance, criminal activity, and money laundering, and supports examinations of the largest financial institutions entrusted to

stop this activity. This work, although in a completely different domain than my previous work, has required many of the same techniques identified in this work. We often look at large financial structures, and I have started to take some of the techniques I applied in this work to help our teams identify similar networks through graph neural networks. Simulation optimization can be used to simulate tax schemes, and identify optimal strategies to understand and identify if they involve illegal maneuvers. These are often optimized with genetic algorithms.

A key throughline between this research and my current work is the appreciation I have for finding a process that works within real-world constraints. At the IRS, we must find ways of identifying cases, providing the relevant information to our talented revenue agents, and ensuring they have the correct information to build out a case. In manufacturing, one must balance the needs of the business and operations and create a system that handles the variability of machines breaking down and process variations. A key item in all my work is trying to make sure I achieve real world results, and I have tried to continue that focus in this research. I believe this leverages many interesting theoretical concepts in unique ways, but ultimately it is about creating a process that can be understood by both management and operations and implemented in a real shop.

This work has been a passion of mine, and I am glad to have had the opportunity to fully think this through. I want to thank my advisor, Dr. Matthew Petering, for his thoughts and guidance on this process. He has helped me understand the nuances of the job shop scheduling problem. I would also like to thank Dr. Jaejin Jang, whose scheduling course helped me solidify my ideas about utilizing dispatching rules. Dr. Hamid Seifoddini whose course flexible

manufacturing systems helped me think through how a system like this would be implemented in a variety of shop environments. Dr. Christine Cheng's courses on algorithm design and analysis of algorithms gave me the skillsets to adequately access the code I needed to write for this work and allowed me to optimize it to run in a reasonable timeframe. Thanks also to Dr. Kaan Kuzu who along with Dr. Cheng introduced to me the idea that there may be different types of shops and that these similarities between shop types could potentially be leveraged. Their guidance during my proposal hearing led me to explore graph neural networks to classify different job shop environments.

Finally, and most importantly, I want to thank my wife McKenzie who has supported me throughout my PhD studies. She has constantly encouraged, supported, and helped me think through and discuss the many issues I ran into throughout this process. I truly could not have completed this work without her support.

1. Introduction

1.1 The Job Shop Problem

Manufacturing is the backbone of the modern economy. Whether it is the steel in our houses providing a secure place to live, the engines in our cars, trains, and planes allowing us to easily travel where we want to go, or the semiconductors allowing me to write these sentences on a computer, manufacturing is what drives our world. As the world gets more complex, so does manufacturing. Modern manufacturers have a variety of ways to set up their shop, the most adaptable of which is the job shop. A job shop consists of different pieces of equipment, each with their own tasks, capable of manufacturing a variety of similar, yet non-identical, parts. As the needs of the modern world continue to grow, so does the diversity of work that a job shop must handle. With this diversity comes a scheduling challenge. Which operations and which workpieces should be done when? These scheduling decisions can be the difference between fully utilizing equipment and having long downtimes. If jobs are scheduled efficiently, manufacturers can minimize downtime which either increases profits or reduces costs for consumers. This also reduces energy consumption, helping the environment. These issues have made the job shop scheduling problem a popular topic of investigation for university researchers, since the real-world benefits of improving these schedules can be tremendous.

The standard job shop scheduling problem (JSSP) is a mathematical optimization problem that represents a shop with multiple jobs which must be processed across a set of machines, with each job consisting of multiple operations that each utilize a different machine.

1.2 JSSP formulation

There are a variety of objective functions that can be analyzed for the JSSP. A common objective is to minimize the makespan, the total length of the schedule. This is the time between the start of the very first operation and the end of the very last operation. This is a straightforward objective function. It avoids potential issues with due dates or weights, which can make the problem easier to solve. This can, however, also lead to uninteresting comparisons between methodologies as a machine can dominate the schedule if it is overloaded, resulting in a schedule where only the outcome of that machine matters to makespan.

Another common objective is to minimize total completion time, which looks at the sum of the completion times of all the jobs. This objective allows for useful comparisons between different schedules as all scheduling decisions matter in the final solution.

A third objective that should be mentioned is the minimization of total tardiness, which equals the sum of the tardiness of all the jobs where each job's tardiness equals the completion time minus its due date (if the job is not completed before its due date). Minimizing total tardiness, and the variation of total weighted tardiness, are the most common objectives used in industry. Tardiness is similar to completion time; however, the jobs are compared to a due date rather than a start time. Weighted tardiness includes weights for each job so that high-priority parts can be accounted for in the schedule. Tardiness can be difficult to consider using synthetic data since if the due dates are not set aggressively, the comparisons are uninteresting. To address this issue, the due dates of all jobs can be set to the start of the

planning horizon. But this leads to the total completion time metric. Therefore, in this work, total completion time will be utilized as it allows for useful comparisons between scheduling techniques and provides a transferable process to industry practice.

To solve the JSSP, the primary problem is to determine the order in which the operations assigned to the same machine need to be scheduled. This is known as the *disjunctive constraint*. There are two possible scheduling decisions for each pair of operations on the same machine, and resolving each of these disjunctive constraints creates the schedule. The disjunctive constraint can be resolved through a variety of graph-based methods, most often utilizing a branch-and-bound algorithm to resolve each of these disjunctive constraints.

Another standard method of solving a job shop problem is using mathematical programming or constraint programming using commercial solvers such as CPLEX or Gurobi to obtain optimal or near-optimal solutions. The standard mathematical formulation of the JSSP is shown below.

Mathematical Formulation of the Job Shop Scheduling Problem (JSSP)

Indices

i,j refers to jobs

k,n refers to operations where (i,k) or (j,n) refer to the job and operation

Parameters

- *J = Number of Jobs*
- *Ops_i = Number of operations for job i (integer) The operations for each job are numbered. Operations with lower numbers must be completed before those with higher numbers begin.*
- *M = Number of machines (integer)*
- *M_{ik} = The machine where job i operation k must be scheduled*
- *t_{ik} = Processing time of job i operation k (integer, ≥ 1)*
- *W = Large positive number*

Decision Variables

- X_{ikjn} = *1 if job i operation k is scheduled before job j operation n on the same machine, else 0 (binary)*
- S_{ik}= *Start time of job i operation k (integer, ≥ 0)*
- C_i = *Completion time of job i*

Objective – Minimize Total Completion Time

$$\min\left(\sum_{i=1}^{J} C_i\right) \tag{1}$$

Constraints

$$S_{i,k+1} \geq S_{ik} + t_{ik}\ \forall i\ from\ 1\ to\ J,\ \forall k\ from\ 1\ to\ Ops_i - 1 \tag{2}$$

$$S_{jn} \geq S_{ik} + t_{ik} - W \cdot (1 - X_{ikjn})\ \forall i\ from\ 1\ to\ J,\ \forall k\ from\ 1\ to\ Ops_i, \tag{3}$$

$$\forall j\ from\ 1\ to\ J,\ \forall n\ from\ 1\ to\ Ops_j,\ such\ that\ (i,k) \neq (j,n)\ and\ M_{ik} = M_{jn}$$

$$S_{ik} \geq S_{jn} + t_{jn} - W \cdot (X_{ikjn})\ \forall i\ from\ 1\ to\ J,\ \forall k\ from\ 1\ to\ Ops_i, \tag{4}$$

$$\forall j\ from\ 1\ to\ J,\ \forall n\ from\ 1\ to\ Ops_j,\ such\ that\ (i,k) \neq (j,n)\ and\ M_{ik} = M_{jn}$$

$$C_i \geq S_{ik} + t_{ik}\ \forall i\ from\ 1\ to\ J,\ \forall k\ from\ 1\ to\ Ops_i \tag{5}$$

The job shop problem is subject to the constraints shown above. The problem starts with a list of *J* jobs with each job *i* having Ops_i different operations. Job *i*'s k^{th} operation must be scheduled on a machine (M_{ik}) and will take a given amount of time (t_{ik}) to process. The starting time of each operation must be greater than or equal to 0. Equation 1 states the problem's objective, which is to minimize the sum of the completion times of the jobs. Equation 2 shows the *conjunctive constraint*. This states that, for all jobs, the next operation must begin after the previous operation has completed. Equation 3 and Equation 4 enforce the *disjunctive constraint*. The disjunctive constraint states that, for each pair of operations (*i,k*) and (*j,n*) that

are scheduled on the same machine, either (*j,n*) must begin after (*i,k*) has completed (Equation 3) or (*i,k*) must begin after (*j,n*) has completed (Equation 4). This order on each machine is controlled through the binary variable X_{ikjn}, which equals 0 or 1. Variable X_{ikjn} is the primary variable whose value must be determined for each combination of jobs assigned to the same machine. Finally, Equation 5 ensures that the completion time of each job is properly computed to be after the job's final operation is completed.

The flow shop, which is a variation on the job shop was shown to be NP-Complete by Garey et. Al (1976) when solving for makespan and when $M>2$. The JSSP is classified as an NP-hard problem for many of the different variations of makespan a shown by Mastrolilli and Svensson (2011), and Chen et. al. (1998). When solving for total completion time as well as variations on completion time the JSSP was shown to be NP-hard by Bruker (1997).

Another way to represent the JSSP is through a graph with a series of *conjunctive arcs*, representing the order of operations within each job, and a series of *disjunctive arcs* between all jobs processed on the same machine. These represent both the conjunctive and disjunctive constraints shown above.

The graph for a small instance of the job shop problem with three jobs, three operations per job, and three machines is shown below in Figure 1.

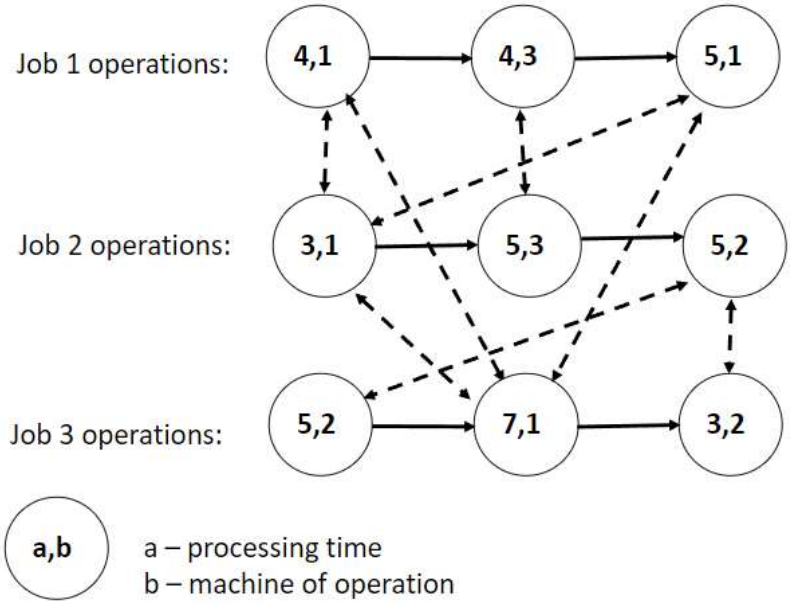


Figure 1: Example Job Shop Problem Disjunctive Graph

Each node shows the processing time of the operation and the machine it is assigned to. The operations on the same job are linked by (solid) conjunctive arcs, and the operations assigned to the same machine are linked by bidirectional (dashed) disjunctive arcs.

We now show the importance of scheduling decisions on the end solution. Figure 2 shows two possible solutions for the JSSP instance depicted in Figure 1. Each solution is shown in the form of a detailed machine schedule. Each row in the schedule represents a different machine, and the columns represent discrete time steps. The top schedule in Figure 2 is the optimal schedule in terms of both total completion time and the makespan of the schedule. By a simple choice of swapping the order of the first two operations on the first machine, the resulting schedule (shown on the bottom of Figure 2) and solution becomes non-optimal. These decisions can compound for larger scheduling problems, making the JSSP extremely difficult to solve in real-world scenarios.

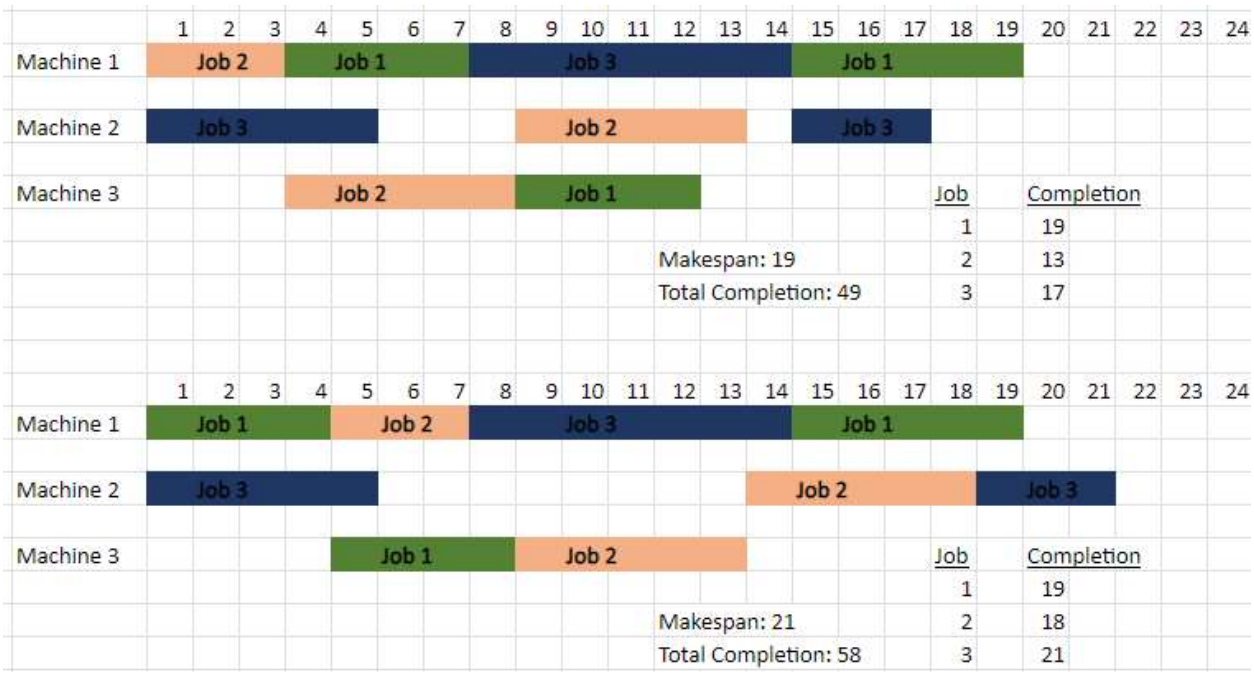


Figure 2: Example Job Shop Problem with optimal (top) and non-optimal (bottom) solutions

The mathematical formulation of the JSSP can be solved via standard integer programming software programs such as CPLEX and Gurobi. However, this approach has some major drawbacks. First, large shop formulations can lend themselves to long computation times to identify an optimal solution, owing to the NP-hardness of the JSSP. Additionally, optimal solutions may require holding a machine idle to wait for another machine to finish running an operation even though the machine could be put to work immediately. The rigidity of an optimal schedule is unintuitive to shop managers and makes the shop vulnerable to variability in processing times. Thus, it is often difficult to implement a mathematically optimal schedule in practice. The work presented in this dissertation avoids the inflexibility of an optimal schedule and focuses on developing scheduling solutions for large job shops in which processing times may be variable.

1.3 Real-World Requirements

The goal of this work is to create a scheduling method and the associated scheduling algorithms that can be applied to real-world job shops. Part of what makes real-world job shops difficult to solve is their scale and variable processes. Specifically, the scheduling method we seek must meet the following four requirements:

1. The proposed method must be able to scale to *large* job shops to be used in a real setting.
2. The proposed method must be able to run in a relatively short time frame so updated schedules can be run overnight. Schedules need to be able to readjust to new demands.
3. The proposed method must account for the real conditions of high-mix job shops, including random machine breakdowns and variable processing times.
4. The proposed method must allow for intuitive implementation in a real shop. This precludes systems that require the precise scheduling of operations and/or that keep machines idle even though operations could be immediately performed on them.

1.4 Stochastic Job Shops

As outlined, a major requirement for representing a real-world job shop is to account for the stochastic elements of the job shop. This modification to job shop scheduling is called the *stochastic job shop scheduling problem* (SJSSP) which includes stochastic elements such as variable processing times (t_{ik}). Authors who have considered variable processing times include Yan and Wang (2007), Jin and Branke (2005), and Vázquez-Rodríguez and Petrovic (2010). Another way to introduce variability is to have random machine breakdowns (RMBs). Many

authors consider job shop scheduling problems with RMBs to be a type of SJSSP. Yan and Wang (2007) and some others consider this to be a *dynamic job shop scheduling problem* (DJSSP). Other authors, such as Sharma and Jain (2015) and Liang et al. (2020) only consider a job shop scheduling problem to be dynamic if new jobs suddenly appear while jobs are being processed. In this research, we consider variable processing times and RMBs but not dynamically appearing jobs.

1.5 Large Job Shops

Much of the academic research focuses on optimal scheduling for small instances of the JSSP, for example with 10 jobs and 10 machines (notated as a 10x10 instance). (If operations are included, 10x10x20 represents a JSSP instance with 10 jobs, 10 machines, and 20 operations per job). Such research is useful in exploring different techniques and can be well suited to compare different algorithms against common benchmarks, especially since there are many 10x10 instances with known optimal solutions to compare against. However, for a technique to have true utility, it must be scalable so it can be used in larger real-world shops.

Based on the author's experience working for a large forging supplier and his frequent discussions with others in industry, it is not uncommon for a shop to have 200 jobs in the system at once, with each job having between 10 and 30 operations. The size of these shops limits the types of solutions that can be used, especially if schedules need to be generated in a limited time frame and/or schedules must be revised due to large changes such as machine breakdowns, the existence of high priority parts, or major due date changes. Another feature of large job shops is that often machines are grouped into machine cells that have many identical

and redundant machines that work to process parts. This means that random machine breakdowns do not stop an entire machine cell from processing parts but do reduce a cell's efficiency.

In a large job shop, there may be scores of machines and hundreds of jobs, each involving multiple operations. Furthermore, each job may consist of hundreds of individual identical parts and each machine may be a machine cell with several identical machines. Research in this space is diverse, with different approaches having different advantages. Much of the research maintains a theoretical bent, with less attention given to the ease of implementation of the algorithm, as this often falls outside the role of the academic researcher. Furthermore, there are often hurdles that must be addressed both in convincing management and the shop floor to use an advanced scheduling approach and ensuring that a computer-generated schedule can be executed properly. These constraints are why many job shops stick to manual scheduling methods. Creating a scheduling process that can be implemented in any large shop is the goal of this research. Simply put, we aim to bridge the gap between research and practice to enable improved scheduling for large, real-world job shops.

1.6 Machine Dispatching Rules

Job shops often rely on one of two techniques to handle these complexities: manual scheduling or using dispatching rules. Manual scheduling can be extremely responsive to the needs of the business, especially when it comes to prioritizing certain jobs to speed them through the system. However, the extra tinkering can often reduce the schedule's efficiency, sometimes making it perform worse than random scheduling. Manual scheduling is also

susceptible to issues of natural favoritism of certain processes over others, leading to the bullwhip effect. The bullwhip effect is a well-known phenomenon in supply chain management in which small changes in a process (demand, supply, schedule, etc.) are not reacted to quickly and the effect compounds, with each stage of the process reacting poorly causing greater and greater impacts over time. The bullwhip effect has been shown to be applicable to job shops as outlined by Stockheim et al. (2005) and Franke et al. (2004).

The alternative is to use dispatching rules. A dispatching rule is a rule for determining which job, among a set of jobs waiting to be processed by a machine, is the next job processed by that machine. Dispatching rules are often applied in an overly rigid manner, such as using "first in, first out" (FIFO) for all machines in the shop, or for part of the shop, allowing manual scheduling for the other parts. However, dispatching rules can also vary by machine and vary at different times on the same machine.

There are some obvious limitations to dispatching rules, the primary being their rigidity. A dispatching rule will always select an available job to schedule from those waiting and will not hold a machine idle in order to wait for a different, often high priority, job to arrive. This means that dispatching rules will not provide the global optimum solution. This may not be as significant a downside as it first appears. For these large, stochastic job shops, globally optimal solutions may be functionally impossible to identify due to the facility's size and complexity. Also due to the stochastic elements and delays in communication that can occur, keeping machines open can lead to additional problems if parts are delayed or there are issues transporting them to start when planned.

Dispatching rules provide another value to job shops; they allow for decentralized control over operations. Each machine or functional area can know what the dispatching rule is and select the next job to run based on that dispatching rule. This makes it easy to implement in practice, limits the impact of variable processes and poor communication, and allows the system to scale to large job shops. To improve the value of these dispatching rules, a framework will be proposed which allows each machine to use its own dispatching rule. This allows for different machines to react to different situations. If a machine is a bottleneck, its dispatching rule can be different than the machines that feed into the bottleneck. By allowing for this flexibility, good non-optimal schedules can be generated that are easy to implement and easy to manage.

2. Literature Review

To understand the approaches taken within this research, it is important to place them in the context of the academic literature. The literature review presented in this section considers methods for solving large job shop scheduling problems, followed by research on dispatching rules. This provides a background on the difficulties of these problems and standard methodologies used to solve them. Next, literature on the following topics related to the solution framework proposed in this research will be introduced: genetic algorithms, simulation optimization, and neural networks. Finally, real-world scheduling will be reintroduced along with lean manufacturing principles to show how this work fits into the broader research landscape.

2.1 Solving Large Job Shop Scheduling Problems

A variety of methods are proposed in the literature for solving large JSSPs. Standard mixed integer programming (MIP) solvers such as CPLEX or Gurobi, and other exact approaches that try to find provably optimal solutions often do not converge in a timely manner when solving large problem instances. Therefore, to solve these large instances, approximation algorithms and metaheuristics are used to seek reasonable, but not necessarily optimal, solutions. Another approach is to use decomposition or relaxation methods to reduce the size of the problem being solved. Often these techniques are combined to both shrink the problem space and increase the ease finding a good solution.

A common method is to use a decomposition method with rolling horizons or time windows to reduce the problem's size. From here, a variety of optimization methods can then be used. These methods reduce the scheduling problem into smaller time horizons and schedule each sub-problem independently, often with more focus given to the first-time horizon. A rolling horizon method has been used with a shifting bottleneck procedure to solve 40-job 20-operation problems in the work by Wang and Li (2007). Another proposed approach by Zhai et al. (2013) used decomposition focusing on the bottleneck machine or machines. They created the schedules of the bottleneck machines via a genetic algorithm and let all other machines be scheduled via dispatching rules. Instances with up to 100 jobs with 50 operations per job were considered. Other approaches have been proposed to use dispatching rules to reduce the problem size. In their paper, Schewnke et al. (2018) proposed using dispatching rules to determine the direction of the disjunctive edges. Then, this solution is refined using

tabu search to alter the disjunctive edges. This created a solution for a problem with 55,917 total operations (about 550 jobs and 100 operations per job). Another paper by Teppan (2018) used a set of dispatching rules, each weighted by a factor to create novel dispatching rules, to schedule problems with 100,000 total operations (1000 jobs and 100 operations per job). Teppan's work will be discussed in more detail later.

As seen in the literature, truly large job shop scheduling problems are often solved via dispatching rules as either a driver of the scheduling algorithm or to help decompose the problem. Dispatching rules have many advantages. They are linear heuristics that when applied always result in reasonable schedules, with no extreme inefficiencies that can occur from poor schedules created by MIP solvers that were provided with insufficient computation time. They also are easy to implement due to shop management's familiarity with them, since they are often used in the absence of other scheduling approaches. Another point of interest from the research when looking at large scheduling problems is the use of evolutionary algorithms as the preferred meta-heuristic for optimization.

Large, high-mix job shops present a unique challenge. The largest job shops in terms of the number of jobs and operations are semiconductor manufacturing facilities in which it is common to have between 242 to 583 operations per job according to Kopp et al. (2020). However, the product mix at such facilities is typically low. The largest high mix-job shops often approach the complexity of semiconductor plants but have less precise processing times and less machine reliability. In high-mix job shops, there is crossover between product lines and therefore multiple overlapping value streams must be created. To address this, equipment

must be shared across many value streams rather than being dedicated to one or a few. This leads to clashes and sequencing issues that have been previously discussed.

2.2 Dispatching Rules

As seen previously, scheduling large-scale job shops often requires dispatching rules, as direct scheduling becomes too intense. Dispatching rules can be easy to implement at the shop floor level. Dispatching rules are predefined rules that limit the solution space of the problem, making it easier to create schedules for large job shops. Additionally, they can be unique to every machine, allowing for different dispatching rules to be used by different machines. The benefits of dispatching rules are that there are predefined sets of rules that are followed in all instances, as well as a requirement that machines run what is at the machine and immediately available to be processed. This eliminates the ability to hold a machine open to wait for additional work to come. This means that only jobs currently in queue when the machine becomes free can be assigned as the next operation.

In general, dispatching rules come in a few varieties. The first kind are defined by the current state of the system. These are based on a property such as the arrival time of each job at the machine, or each job's due date. A systematic look at a variety of dispatching rules was conducted by Panwalkar and Iskander (1977). These dispatching rules represent a variety of standard approaches, and most additions to this list are based on combinations of many of these existing rules. The other variety are custom dispatching rules determined by an algorithm that creates a custom priority for sequencing operations.

A list of common dispatching rules is provided by Park et al. (2018). These rules are shown in Table 1.

Table 1: Dispatching Rules according to Park et al. (2018)

Dispatching Rule	Description
First In First Out (FIFO):	The first job to arrive at the machine is processed first.
Last In First Out (LIFO)	The last job to arrive at the machine is processed first.
Shortest Processing Time (SPT)	The job with the current operation having the shortest processing time is processed first.
Longest Processing Time (LPT)	The job with the current operation having the longest processing time is processed first.
Shortest Total Processing Time (STPT)	The job with the shortest total processing time across all operations is processed first.
Longest Total Processing Time (LTPT)	The job with the longest total processing time across all operations is processed first.
Least Operations Remaining (LOR)	The job with the least number of remaining operations is processed first.
Most Operations Remaining (MOR)	The job with the greatest number of remaining operations is processed first.
Least Work in Next Queue (LWINQ or LQNO)	The job whose next operation is at a machine that has the least amount of waiting jobs, is processed first.
Most Work in Next Queue (WINQ or MQNO)	The job whose next operation is at a machine that has the most waiting jobs is processed first.
Random	Each job is equally likely to be processed next on the machine.

To illustrate how a dispatching rule works in practice, consider machine 1 in the example JSSP instance depicted in Figures 1 and 2. In this instance, jobs 1 and 2 are both available to be scheduled on Machine 1 at time 0. If we were to use the SPT dispatching rule, the processing time of the first operation for both jobs would be compared and the job with the shorter first operation (job 2) would be scheduled first, followed by the job with the longer first operation (job 1).

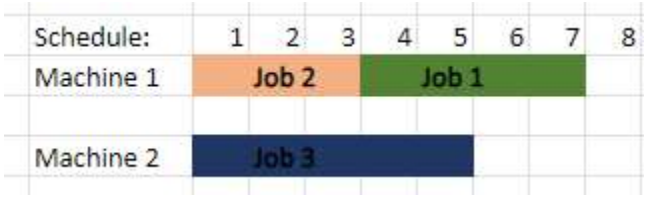


Figure 3: Initial portion of the schedule for the instance shown in Figures 1-2 if the SPT dispatching rule is used at Machine 1.

Meanwhile, machine 2 starts to process job 3's first operation begins at time 0 on machine 2 because that is the only operation available to be performed on machine 2 at that time. Note that only jobs 1 and 2 are considered for scheduling on Machine 1 at time 0, and in no case will the machine be left open at time 0 waiting for job 3.

These common dispatching rules have been used extensively in research. For example, Teppan (2018) analyzes a large job shop to replicate the semiconductor industry, where there can be 100,000 weekly operations processed on 1,000 different machines. To optimize the schedules for these job shops, sets of dispatching rules were analyzed. In particular, 12 different dispatching rules are combined to provide a normalized score which makes the scheduling decision for the next operation. This allows schedules to be calculated in linear time through a single schedule and yields results for very large problem sizes. The approach taken was non-dynamic with 33,936 dispatching rule permutations tested, all explicitly. This work

shows that even with limited variety, dispatching rules can be effective for large problem instances.

A more customizable approach to creating dispatching rules was taken by Hildebrandt et al. (2014) where they used a genetic algorithm (GA) to build custom dispatching rules for a large shop. (A general introduction to GAs is provided later in this dissertation). This methodology relied on using sets of parameters commonly used in dispatching rules such as the processing time of the current operation, average processing time of waiting jobs, time already spent in the system, and time already spent in the queue. These parameters were then weighted to varying degrees to create a custom dispatching rule for the exact problem instance.

Xu et al. (2021) created a methodology in which a unique dispatching rule for each individual machine was evolved using a GA. In Xu et al. (2021) sets of features from various dispatching rules were combined to evolve a unique dispatching rule. This approach allowed for multiple comparisons and ways to combine the various features beyond simple weightings, such as using addition, multiplication, and minimum and maximum selectors. For example, using the terminology in Table 2 the next job processed by a machine could be the job with the minimum value of the function min{AT, PT}*PT + min{MRT-NRT, (PT+1)*min{AT, PT}} + NPT*PT*NINQ.

Table 2: Dispatching rule abbreviations from Xu et al. (2021)

Feature	Description
AT	Arrival time of job
PT	Processing time of the operation
MRT	Machine ready time
NRT	Ready time of the next machine
NPT	Processing time of the job's next operation
NINQ	Number of jobs queuing at the job's next machine

Another popular set of evolutionary algorithms that have been used to build unique dispatching rules are pheromone-based algorithms. One example of these algorithms is ant-colony optimization (ACO). ACO consists of multiple artificial agents (ants) that iteratively traverse the solution space, depositing pheromones based on the quality of the solution. In the JSSP, these pheromone trails represent machine paths that parts can take to be scheduled, and the paths that result in better schedules end up with a stronger pheromone trail.

The ACO algorithm has been used in simulation optimization frameworks to build custom dispatching rules. The work by Gohareh and Mansouri (2022), utilizes a Markov decision process to model a SJSSP. Due to the large number of states that can exist, several simulations are run and pheromones deposited along the solution space. Their work broke the simulation optimization ant colony framework (SOACF) into two steps. The first was used for data

gathering: exploring the solution space through a set of heuristics that select which job to process when and depositing pheromones along the favorable paths. The next phase involved selecting jobs through a hybrid of heuristic methods and the weight of the pheromone trails. By using the initial heuristics, which were dispatching rules, the authors found faster convergence. Next, jobs were selected via a roulette wheel corresponding to pheromone levels. The resultant simulation ended with a database of different states along with the pheromone levels for those states. This pheromone database could then be used as a specified dispatching rule to determine which jobs to run at which point in the schedule.

Another perspective introduced by Yan and Wang (2007) is to define dispatching rules over time to the entire shop instead of to applying them to individual machines. In their work, Vazquez-Rodriguez and Petrovic (2010) use a GA to determine sets of dispatching rules and the number of operations that should use these dispatching rules. For example, the algorithm may determine that the first five operations on a machine will use FIFO, the next twelve use shortest processing time, etc. A different perspective on this is shown in work by Rolf et al. (2020). In their work, the same dispatching rule is applied to all machines in a shop, but the rule may change over time. Therefore, the schedule may start with earliest due date, then move to shortest processing time at a set time and continue with this method. The number of switching points was defined up front, but the time of each switching point and the dispatching rule used during each time window was decided by the GA.

Due to the varying nature of job shops no one dispatching rule will always perform best, and dynamically updating rules can yield better results as shown in the work by Grady and Lee (1988), Pierreval (1992), and Pierreval and Mebarki (1997). There are two main methodologies

proposed in the literature to update dispatching rules dynamically: real-time simulation and artificial intelligence methods. Dispatching rules may be dynamically updated either with the arrival of new parts, or after enough time has passed to better process the later operations in the schedule.

Real-time simulation methodologies have been proposed to simulate a given time window for the current state of the shop such as in the work by Ishii and Talavage (1991). However, these methodologies have two shortcomings as identified by Fan et al. (2015). The first is that the simulation takes time, making the system slow to respond if updates are needed. Second, the methodology has no way to learn more about the given shop and does not account for knowledge gained over time.

The alternative is to utilize artificial intelligence methods, such as expert systems, machine learning methods, or artificial neural networks. Expert systems are rule-based systems that try to approximate the decisions scheduling experts would normally make. Once these rules have been validated, the system can behave like a scheduling expert, making decisions in real-time as shown by Zhao et al. (2015). Machine learning methods utilize simulations to create a large training dataset to refine the machine learning algorithm, as shown by Li and Jiang (2007), Wang (2021), and Priore et al. (2001). Finally, neural networks are trained using optimal schedules and allowing the trained neural network to select the sequencing of the jobs. These approaches allow for quick creation of schedules, allowing for online or real-time scheduling.

Since artificial intelligence methods rely on previous schedules, dynamic shops with different part profiles may have issues adapting these artificial intelligence methods. In dynamic shops, many different machines may end up becoming bottleneck machines depending on the profile of jobs running through the shop at any given time. If there is time to re-optimize schedules, a simulation-based approach provides a schedule tailored to the exact profile of the shop. On the other hand, if the training dataset does not capture the full variability of the shop, the machine-learning-based dispatching rules are likely to perform poorly, especially with variability such as from RMBs. Ultimately, there are many ways that dispatching rules can be selected to generate useful schedules, especially for large, high-mix job shops.

2.3 Applications of Genetic Algorithms for Job Shop Scheduling

Evolutionary algorithms such as genetic algorithms (GAs) or ant-colony optimization (ACO) are well suited for solving large NP-hard problems like the JSSP and its many variations. These algorithms can find solutions quickly, while avoiding local optima that many other metaheuristic algorithms have trouble avoiding. When employing these algorithms, there are a few approaches discussed in the research. These involve either directly creating a detailed schedule, selecting from a set of dispatching rules, or creating a unique dispatching rule to use for future scheduling. Each of these approaches has advantages and disadvantages.

The most common evolutionary algorithm employed is the GA. The GA is a global search technique which is outlined extensively by Eiben and Smith (1998). The algorithm starts by encoding a set of potential solutions to a problem instance into *chromosomes*, with each potential solution represented by a different chromosome. The detailed pieces of information

within each chromosome are called *genes*. The chromosome is a way of encoding a solution to the problem instance. Multiple chromosomes are created for a single generation, and over multiple generations, these chromosomes are modified to improve their performance, also known as *fitness*. The chromosomes are manipulated through a combination of crossover and mutation operations between generations. The algorithm is flexible and can be designed in different ways based on the problem domain. Crossover operations are conducted by selecting parents based on a weighted random number such that fitter parents are more likely to be selected. Each crossover operation takes information from two different parent chromosomes to create two child chromosomes. In a crossover operation no new information is created. Instead, sections of both parents are used. This is often done by selecting a breakpoint. Data prior to the breakpoint from the first parent chromosome is combined with data after the breakpoint from the second parent chromosome to form the first child chromosome. Also, data prior to the breakpoint from the second parent chromosome is combined with data after the breakpoint from the first parent chromosome to form the second child chromosome. For mutation, random genes within a selected chromosome are modified. This mutation operation avoids local optima when searching by continually adding new information. In order to keep the best options available for crossover and mutation operations, the fittest N ($N \geq 1$) chromosomes in each generation are typically copied into the next generation. The algorithm executes for a set number of generations and the best chromosome is returned when it terminates.

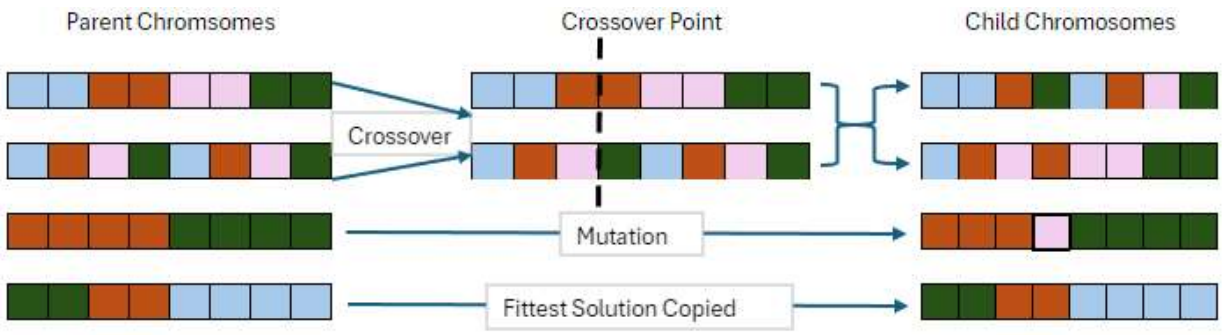


Figure 4: Outline of Genetic Algorithm Population Change

The basic functioning of a GA is depicted in Figure 4. In this example, a population with four chromosomes exists in the first generation. For the next generation, the last chromosome is the fittest and is copied in its entirety into the next generation. Chromosomes 1 and 2 in the next generation are created through a crossover between the first two parent chromosomes. Child chromosome 1 consists of the first three genes from parent chromosome 1 and the last five genes from parent chromosome 2. Child chromosome 2 consists of the opposite portions of the two parent chromosomes. For chromosome 3, the original is passed to the next generation except that the fourth gene is mutated. In practice, mutation is often combined with crossover as well, but is separated here for clarity. These four chromosomes are now part of the next generation, and the process continues.

One approach to using a GA for job shop scheduling is to directly create a detailed schedule. For this, each chromosome directly encodes the solution. There are multiple ways to do this. The most common are to either utilize a representation of a run order as conducted by Ye and Chen (2010), or assign each job operation a priority order, then schedule jobs based on the priority order as conducted by Czerwinski (1997). Due to the precedence constraints of the JSSP, it is often necessary to add an additional step after the construction of a new solution to

repair the chromosome to ensure none of the constraints are violated. This correction can lead to both long computation times as well as issues with local optima as the genetic algorithm continues to find similar solutions when correcting the encoded schedule.

To address the correction of these schedules, various methodologies have been proposed. In the work by Ali et al. (2020), a genetic algorithm is created with two breakpoints for the crossover operation. The total processing time of each of the three segments is calculated and the child chromosome is created through the swapping of the high, medium, and low processing time segments. The authors claim this helps pass job precedence information on, which reduces the modifications performed when recombining chromosomes and passes more valuable information on to the next generation.

The other main approach for using a GA for job shop scheduling is to encode machine dispatching rules in the chromosomes rather than a detailed schedule. This can be done through either selecting from a set of dispatching rules, often allowing the use of different dispatching rules at different times or machines, or by creating a new, custom dispatching rule and having this rule drive decisions. These unique dispatching rules can perform well for a given problem; however, they may be more difficult to execute at the shop floor level due to the inherent difficulty in explaining how the dispatching rule is selecting the next job to process, resulting in the need for the creation of priority lists for each machine. They also do not provide flexibility and may not result in optimal decisions for all machines.

Another variation utilizing an evolutionary algorithm is introduced by Zhou et al. (2019). In that research, a GA with generative expression programming (GEP) is used to store

chromosomes in a tree structure. The chromosomes in these trees store different criteria for scheduling and different mathematical operators. The resultant gene expression creates a unique dispatching rule combining a variety of mathematical operators and scheduling criteria.

2.4 The Stochastic Job Shop Scheduling Problem

Real-world, stochastic job shop scheduling problems (SJSSPs) are not only dynamic regarding the job profiles that can exist at a given time, but also in the uncertainty that can occur within a shop. As previously mentioned, this uncertainty includes both variability in machine processing times and random machine breakdowns (RMBs).

Adding the uncertainty of variable processing times and machine breakdowns to the problem can create issues for the algorithms that are traditionally used to solve the (non-stochastic) JSSP. To understand this variability and how it can impact solutions, we must first understand the variability that exists within an algorithm. Jin and Branke (2005), provide a comprehensive overview of how uncertainty can exist within an evolutionary algorithm. They define four classes of uncertainty:

1. Noise in fitness evaluation. In the SJSSP, the expected fitness of a given chromosome is approximated by simulating its performance in a stochastic job shop environment. By limiting the number of simulation replications used to evaluate each chromosome, noise is introduced to the fitness evaluation.
2. Chromosome robustness. Stochastic events like RMBs can occur after a schedule is issued which can affect the fitness of a given chromosome, and/or its feasibility. For example, the fitness of a chromosome that defines a detailed schedule for a

deterministic JSSP may not be easy to compute in the context of a SJSSP because the chromosome may not even be feasible for the SJSSP. A more robust chromosome design – for example, using dispatching rules – leads to less uncertainty in fitness evaluation for SJSSPs.

3. Fitness approximation. If the fitness function takes too much time to calculate, it must be approximated. Depending on the accuracy of the approximated fitness function, checks against the true fitness function must be computed, and these function checks also take time.
4. Time-varying fitness functions. In some cases, re-optimizations must be completed due to changes in the system. These variations can be done more quickly if the results of previous solutions are accounted for during these runs, such as by using previous high performing chromosomes as a starting point to optimize for new chromosomes.

These issues can be addressed through various means. The first two issues can be addressed by increasing the number of simulation replications. The increased number of simulation replications means more robust solutions can be evaluated; this, however, comes at the cost of increased computation time. The issues around noise and robustness can also be addressed by ensuring that the system is monitored and if too much variability is encountered, the process is re-optimized to generate a new schedule. Fitness approximations can be an issue with complex fitness functions; however, the fitness functions we consider – total completion time, tardiness, and weighted tardiness are all easy to calculate. By sticking to these simpler fitness functions and ensuring the scheduling algorithm can be re-run easily, we can address both issue 3 and issue 4.

2.5 Simulation Optimization

As discussed, a common way to address variability, and design a system that responds to changes, is to run sets of simulations which represent this variability. The system can then be optimized using the fitness values calculated by the simulation runs. Simulation also provides a way to translate a set of dispatching rules into a detailed schedule. If there is no variability in the model, only one simulation is needed to calculate the fitness of a solution. However, if variable processing times and/or RMBs are included, multiple simulations are needed, and the results are averaged to calculate the expected fitness of the schedule. This technique – of using simulation as a subroutine for computing the objective value of each proposed solution within an overall optimization routine which seeks to find the best solution – is called *simulation optimization*.

A simulation optimization approach for the SJSSP is proposed by Yan and Wang (2007). They outline how improvements can be made to previous simulation optimization approaches by allowing for optimization at the machine level. Yan and Wang (2007) build on previous analysis by replacing the JSSP with the SJSSP. The SJSSP more accurately represents real-world scheduling problems where job operation times are not fixed. However, their work does not consider RMBs.

Due to the computational burden of simulation, other measures have been proposed by Wu et al. (2018) using slack-based methodologies. Simulation model robustness is especially an issue in the context of RMBs. To account for this, Wu et al. (2018) propose a way to account for RMBs through reactive scheduling where a new schedule is created when a RMB occurs.

Accounting for RMBs directly within a simulation can lead to solutions which better respond to these breakdowns, as shown by Xu et al. (2021).

Altering a problem to fit a specific use case can give an advantage over more generalized methods. In their work, Golenko-Ginzburg and Gonik (2002) create a methodology to represent costs in their scheduling problem. This creates a penalty for each job that represents the lateness cost of the job. This lateness cost captures the costs to hold inventory. The lateness cost of the job includes one-time lateness penalties and time-based lateness penalties. Methodologies such as these represent the creation of dispatching rules to directly represent the optimization goals, in this case minimizing penalty costs. When the objective function is less linear than makespan, more intricate dispatching rules can account for this lack of linearity. This work identifies unique dispatching rules for a particular domain, showing that a good understanding of the problem can yield positive results.

Simulation optimization can also make use of commercial simulation software. These software packages allow the user to approximate many parts of the shop using a low-code/no-code interface. In their paper, Li and Qu (2009) showed the feasibility of these methods by creating a specific simulation model of the shop being optimized. This can allow less technical users to add their expertise to the problem to create a more accurate simulation, tailored to a specific shop environment. However, using commercial simulation software can increase computation times.

There has also been extensive work using simulation optimization methodologies in other domains outside of job shop scheduling. For example, Gholami and Zandish (2009) use a

GA to optimize a scheduling method in a flexible job shop with RMBs. Sajadi et al. (2019) uses a GA to select job priorities while accounting for RMBs. Similarly, Gupta and Jain (2021) identify how preventative maintenance can impact machine scheduling through extensive simulations. Finally, Gohareh and Mansouri (2022) present a simulation optimization approach for the SJSSP using Markov decision processes and an ant colony optimization algorithm. As seen in the literature, the simulation optimization framework is well established.

2.6 Classifying the Job Shop: Introduction to Neural Networks

One way that a GA's performance can be improved is through pre-seeding. Pre-seeding a GA is when the initial population of chromosomes include high quality known solutions to similar problem instances. This allows the GA to look for solutions similar to these high quality solutions when solving, reducing the overall computation time. Work by Oman and Cunningham (2001) show that pre-seeding was extremely beneficial to solving the Traveling Salesman Problem, which is similar to the JSSP but with fewer constraints, however, did not provide any benefits to the JSSP. Their findings highlighted that the pre-seeded solutions were too dissimilar from those being solved and this was why the benefits were not seen with the JSSP. Later work by Walsh and Fenton (2004) found that pre-seeded dispatching rules provided overall benefits, but there was a limit on how providing to many initial solutions which could reduce overall performance. Other work by Zhao et. al. (2005) showcased how a Neural Network could be used to estimate an initial dispatching order, which could then be refined using a GA, using these initial estimates as a pre-seed to the GA. In their work the GA is

designed to optimize directly for the run sequence for all jobs, as compared to creating a dispatching rule.

The disjunctive graph of the job shop problem (see Figure 1) is unique to its problem. A technique explored in this dissertation is to see if classifying these disjunctive graphs can provide useful information when optimizing the solution. To classify job shop problem instances into categories, a type of neural network architecture known as a graph neural network (GNN) will be utilized. To appropriately understand how GNNs are useful in these studies, an introduction to general neural network architecture is appropriate.

Neural network architecture is outlined by Aggarwal (2017). Neural networks leverage networks of artificial neurons. An example of this is the perceptron. The perceptron was developed by Frank Rosenblatt in the 1950s and 1960s. Perceptrons are not used as often in modern neural network architectures; however, they provide an introductory foundation to the concept.

A perceptron takes several binary inputs and produces a single binary output as outlined in Figure 5.

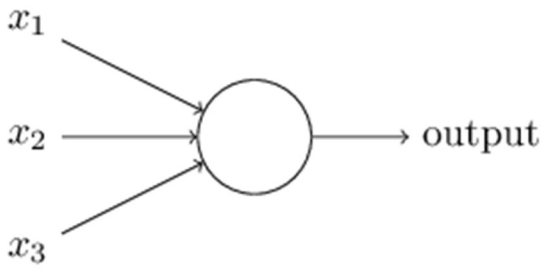


Figure 5: Perceptron as outlined by Aggarwal (2017)

Each of the inputs is assigned a weight which represents the importance of the input. The output of the neuron, either 0 or 1, is based on the sum of the inputs and their weights. If this sum is higher than a predetermined threshold, the output is 1; otherwise, it is 0.

Neurons come in a variety of formats. A more complex example is the sigmoid perceptron. Standard perceptrons, when combined into a neural network, can result in situations where small changes in the inputs cause major changes in the output as the values move above or below the threshold. To address this, sigmoid perceptrons are used. With sigmoid perceptrons, the output of the neuron is no longer binary but instead a continuous value based on the sigmoid function defined below in equation 6 where *z* equals the sum of the inputs times their weights and any bias *b (z=w*x+b)*. The plot of this function is shown in Figure 6.

$$\boldsymbol{\sigma(z) = \frac{1}{1 + e^{-z}}} \tag{6}$$

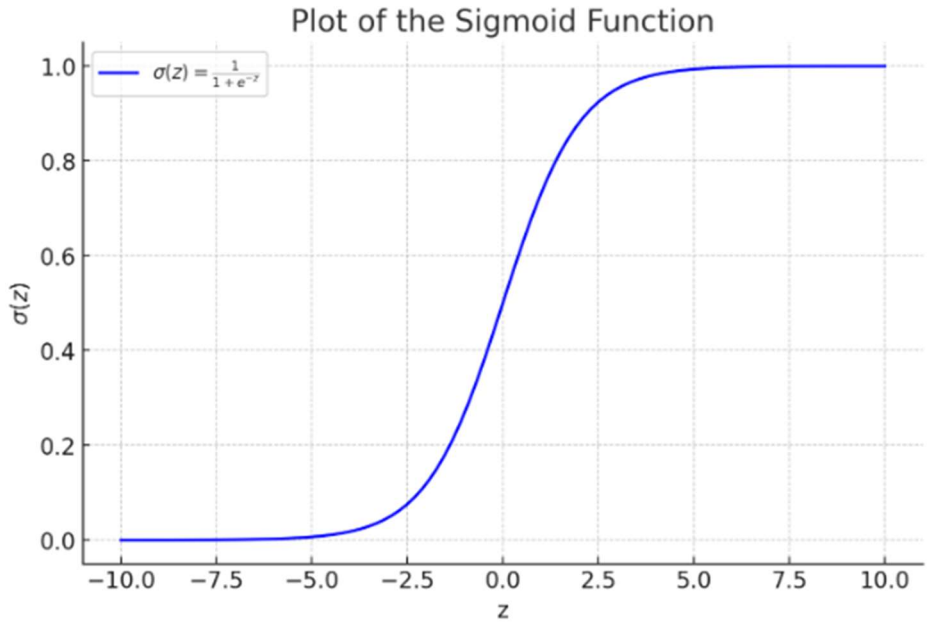


Figure 6: Sigmoid Function

Sigmoid neurons are built into networks and, through an iterative process of training, the weights and bias are refined. These are often built into multiple layers. The first is the input layer, which feeds into middle layers referred to as hidden layers since these neurons are not directly viewable albeit important to the overall performance of the network. The final is the output layer, which in the case of a binary classification problem would be a single neuron. This results in the network decision being 0 or 1, outlined in Figure 7. These types of networks are sometimes referred to as feed-forward networks since the information from one-layer feeds forward to the next without any recursion, looping, or complex interactions.

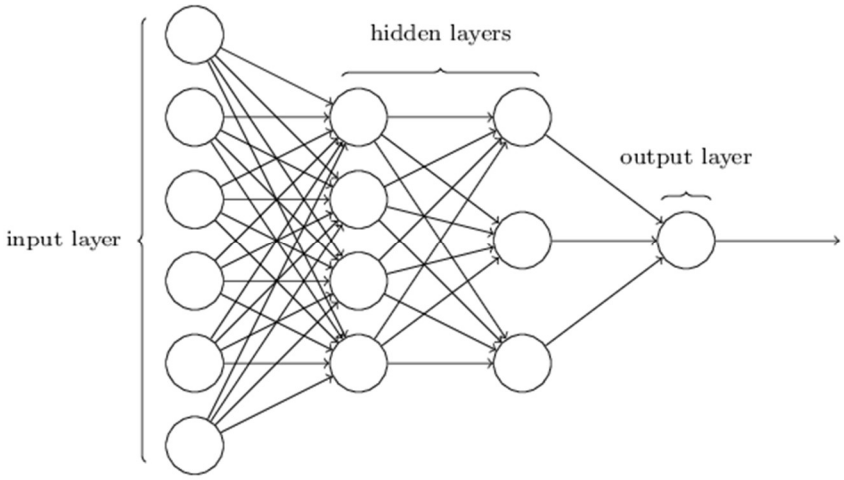


Figure 7: Simple neural network as outlined by Aggarwal (2017)

These networks are trained using sets of examples along with the associated known output values of those examples. For example, the training set could be a set of digital images along with the known output values 1 (image shows an apple) and 0 (image does not show an apple). Initially, the neural network weights and biases are randomized and the performance of this network is measured using a loss function which defines the difference, or loss, between the known output values of the training set and the output values generated by the neural

network. This can then be solved like any optimization problem, including using GAs, where the values of the weights and biases are encoded and modified over time to improve performance as shown by Ding et al. (2011).

One way that neural networks aggregate information is through convolution layers. Combining convolution layers with a standard neural network creates a convolutional neural network (CNN). CNNs are mainly used in image recognition tasks and have applications for graph problems. CNNs create a structure for information aggregated as outlined by O'Shea and Nash (2015). In the previous apple example, a CNN allows the NN to aggregate information from nearby pixels together, so the model may look for many red pixels surrounded by other red pixels to help identify the apple. CNNs will be discussed in more detail in the following section.

2.7 Classifying with Graph Neural Networks

Graph classification is a powerful technique to help identify inherent structures within graphs. The process of graph classification looks at either full or sub-graphs and identifies other graphs that share those structures. An example of this is outlined in Figure 8 where the first two graphs are similar enough and considered to be of the same general graph class, whereas the final graph is unique enough to be a separate graph class.

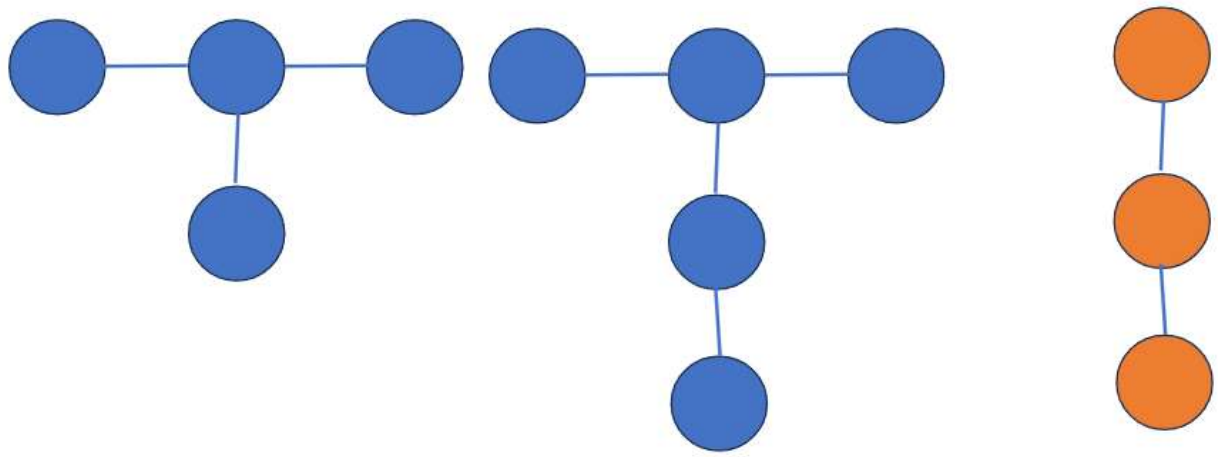


Figure 8: Graph Classification Example

There are standard approaches to classify graph structures, such as looking for cliques or other specific graph substructures. Due to the variability of job shops, these approaches are not always useful as similarities in job shop environments are driven not only by how parts are assigned to machines (the graph structure) but also by part processing times which are represented by node labels. In addition, the variety of graph structures that can exist for the disjunctive graph of the large JSSP is so great that they cannot be fully defined.

However, it is possible to understand the general topology of the disjunctive graph and classify these graphs. If these graphs can be classified, understanding which solutions perform well for one class of job shop problem instances may help in tackling other classes of instances that have similar topology. For example, a shop with a single bottleneck machine will require a different approach to scheduling than one in which three different machines act as bottlenecks throughout the process, and both require different approaches than if all machines are equally loaded. The goal of graph classification is not to fit each shop instance into a perfect category

but to group instances based on their general qualities to learn from the uniqueness of their topology.

To group these problem instances, graph neural networks (GNNs) can be used for the task of graph classification. A graph neural network is one whose input includes the graph's structure representing the underlying problem being solved. For example, the disjunctive graph that represents an instance of the JSSP (see Figure 1); Thus, a GNN's structure is typically more complex than the feed-forward network shown in Figure 7.

GNNs effectively model and process graph-structured data. For example, the topology of the disjunctive graph that represents a JSSP can easily be handled by a GNN (Zhou et al., 2020). In traditional feed-forward or convolutional neural networks, data is typically represented as vectors or multi-dimensional arrays. However, this form of representation does not encapsulate the relational structure present in graph data. The unique aspect of GNNs is their ability to consider this relational information, allowing for the modelling of complex systems that involve interactions and relationships between the components of a system (i.e., between the nodes in a graph).

Figures 9-12 illustrate how a GNN can be can be used to classify disjunctive graphs that represent JSSP instances. Figure 9, which is identical to Figure 1, shows the disjunctive graph of a small JSSP instance. The first number in each node is the processing time of the job and the second number is the machine where the given operation takes place. Figure 10 shows the same shop as represented in the GNN. Here, the second label on each node is the number of incoming edges into the node. Finally, Figure 11 represents the same information except that

the nodes are sized based on their processing times and colored based on the number of incoming edges. Nodes with longer processing times are larger and those with more incoming edges (representing how many scheduling conflicts the operation needs to resolve) are darker. The node labels (i.e., *node features*) and loops within the graph make it significantly more complex than the simple feed-forward graph shown in Figure 7, making a GNN approach necessary for the JSSP.

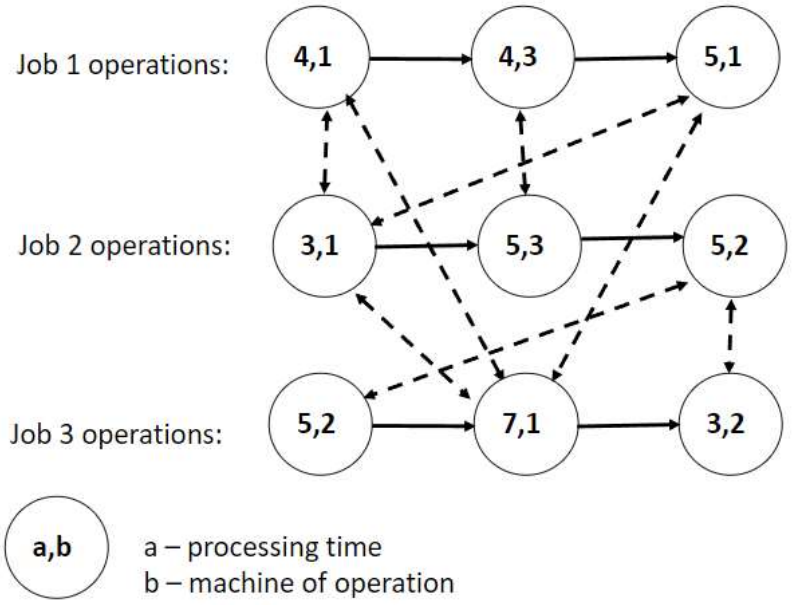


Figure 9: Disjunctive graph of a small JSSP instance

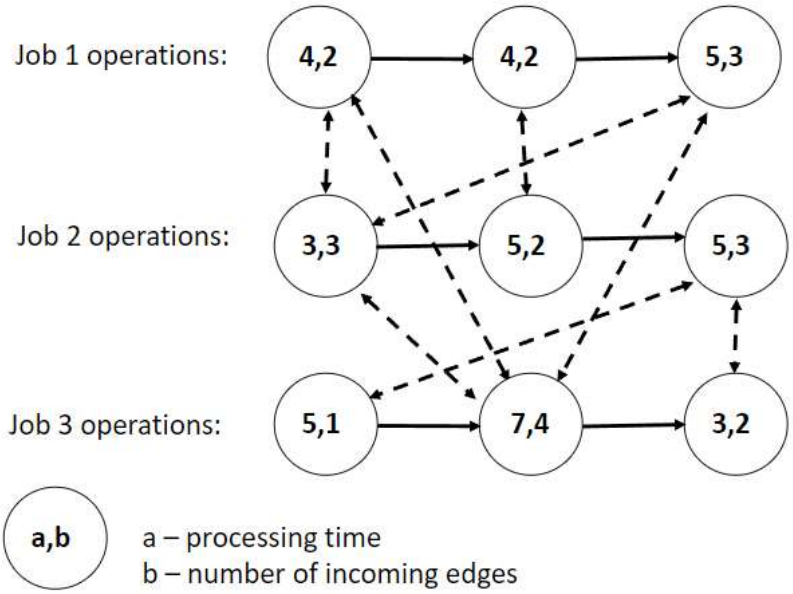


Figure 10: JSSP with mapping for GNN

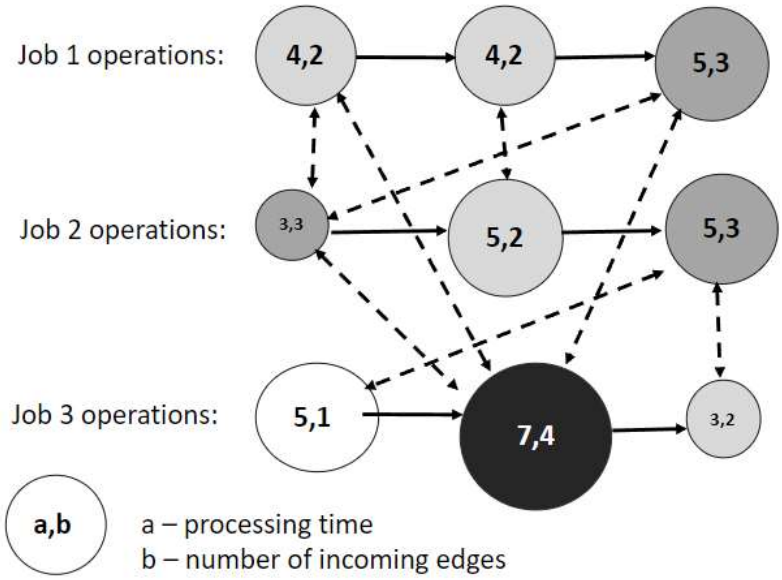


Figure 11: JSSP with mapping and visualization for GNN

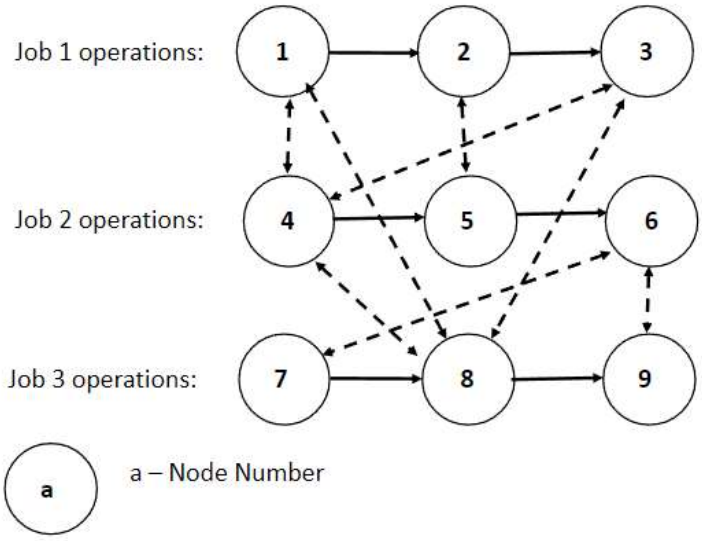


Figure 12: JSSP node labels

Figure 13 provides an overview of the GNN methodology. The GNN takes the node labels in Figures 10-11 represented above and the list of edges as inputs (the model takes the list of edges which it converts into an adjacency matrix as part of the graph representation). These are then processed through different types of convolutions to generate representations of the node information. Then, these are processed through a feed-forward process like a basic neural network to determine an output representing the predicted class of the graph. Examples of a graph class are: one bottleneck machine, three bottleneck machines, no bottleneck machine.

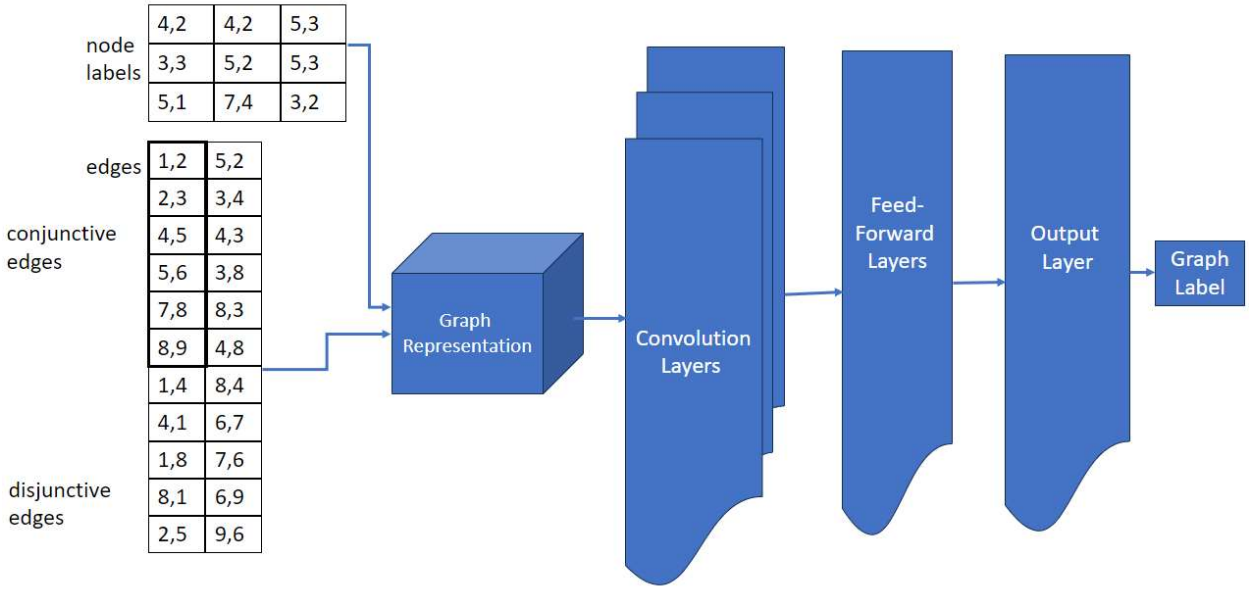


Figure 13: GNN Architecture

Details regarding feed-forward layers and the output layer were covered in Section 2.6, but more explanation is needed regarding convolution layers. There are two common variants of convolution layers used for graph classification: graph convolution and graph attention. Both are applied as specific layers in the neural network and help aggregate network features. When applied, the resulting neural networks are broadly referred to as graph convolutional neural networks (GCNN). When only graph attention is used, they are called graph attention networks (GAT). Understanding how these layers function provides insight into how a GNN can be used for graph classification.

Convolutional layers in neural networks are classically applied to grid-like structured data such as images, but they can also be applied to the more general graph structures found in GNNs. A GNN with at least one convolutional layer is called a graph convolutional network (GCN). As explained by Zhang et al. (2019), this is done through neighborhood aggregation

where features of adjacent nodes are aggregated together. The features are then transformed and a non-linear activation function is applied to the transformed features to better enable the GCN to capture complex features. A common transformation, outlined by Veličković et al. (2017), is shown in Equation 7.

$$\overrightarrow{h_i'} = \sigma\left(\sum_{j \in N_{ij}^{\alpha}} \overrightarrow{g_j}\right) \tag{7}$$

In equation 7, a new node feature *h'* is calculated for each node *i* in the graph based on an activation function *σ* which could be the sigmoid function shown in Equation 6 or some other activation function. The term inside the parentheses is the sum of feature vectors for the node's adjacent nodes (g_i) times the specific weighting factor (α_{ij}). N_i is the set of nodes adjacent to node *i.*

This is shown visually in Figure 14, where the node feature *h'* for the upper-right node (i.e., node 2) is calculated based on the node features g_1, g_3, and g_4 of its three adjacent nodes.

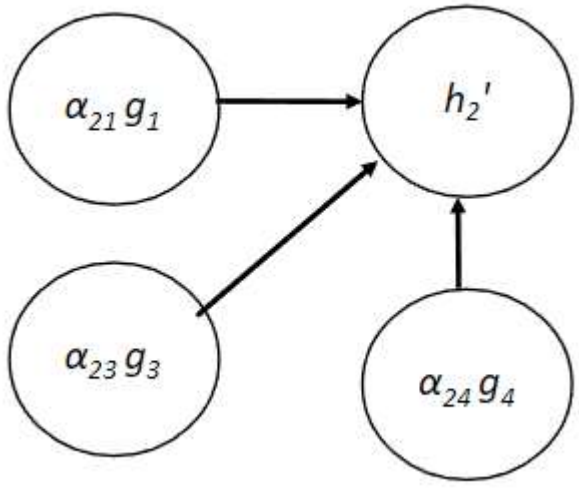


Figure 14: Representation of Convolution in a Graph

The α_{ij} values are either defined explicitly through structural properties of the graph or implicitly by leaving them as learnable weights. This means that each node *i* will gain a new node feature (h_i') that represents the factors of its nearby nodes. If this is run a single time, *h'* will include the influence of the directly adjacent nodes. Adding a second convolution layer will increase the influence to those nodes two hops away, and so on as more layers are added. This results in nodes including information from their immediate and non-immediate neighbors, effectively encoding the graph structure into the nodes themselves.

Graph attention networks (GATs) are a type of convolutional network created to address some shortcomings of traditional convolutional neural networks. GATs incorporate an attention mechanism which computes *attention coefficients* (e_{ij}) that adaptively signify the importance of neighboring nodes during feature aggregation. This allows the model to focus more on relevant neighbor features, as discussed by Veličković et al. (2017). Attention coefficients determine the influence of neighboring nodes which are parametrized through a weight matrix for each node. The weighted features are then aggregated for each node. Another key feature of GATs is the use of multi-head attention, where multiple independent attention mechanisms are applied to provide multiple feature representations for each node.

GATs are differentiated from convolutional layers by implicitly defining the α_{ij} in equation 7 using self-attention. Equations 8-9 show the GAT process. First, in equation 8, e_{ij} is calculated for each pair of nodes *i,j* based on the feature vectors of those nodes (h_i, h_j). The value of e_{ij} is then compared to the values of e_{ik} for all nodes *k* that are adjacent to node *i* to compute α_{ij}.

$$e_{ij} = a(\overrightarrow{h_i}, \overrightarrow{h_j}) \tag{8}$$

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k \in N_i} \exp(e_{ik})} \tag{9}$$

This is often replicated K times through a process known as multi-head attention. This is outlined by Veličković et al. (2017) in their seminal work:

> *A GAT layer with multi-head attention. Every neighbour i of node 1 sends its own vector of attentional coefficients, $\vec{\alpha}_{1i}$ one per each attention head α_{1i}^k. These are used to compute K separate linear combinations of neighbours' features $\vec{h}_i$, which are then aggregated (typically by concatenation or averaging) to obtain the next-level features of node 1, $\vec{h}_1'$.*

This multi-head attention provides additional flexibility compared to standard convolution. The attention feature can be learned, allowing additional flexibility for the model to increase the importance of certain relationships over others, highlighting the attention of those relationships in the network. The way attention networks are applied is like convolution networks except that there is added flexibility from the multi-head attention.

2.8 Applications of GNNs to the JSSP

With the growth of machine learning, there has been some exploration into how to best use GNNs to solve the large disjunctive graph problems that represent instances of the JSSP. The research to date has aimed at using GNNs for edge prediction to resolve edges to directly solve the disjunctive graph problem, or to use these predicted edges as a way of creating custom dispatching rules. However, an area that has not been explored, to this author's knowledge, is using a GNN to classify problem instances to aid in pre-seeding the initial

solutions for an algorithm such as a GA. Additionally, there has not been exploration into whether solutions for small problem instances are helpful in tackling larger problems.

The use of GNNs to solve the JSSP has broadly been explored. One approach was used by Park et al. (2021). In their work, they propose using a GNN to build a custom priority dispatching rule. They built their GNN with node features that represent the status of the operation, whether it was completed or not, the processing time, the number of succeeding operations, waiting time, and remaining time. With this information, the current state of the shop can be accounted for and an applicable dispatching policy determined.

A similar study was conducted by Zhang et al. (2020) in which a priority dispatching rule was developed via a GNN. One difficulty they found was in creating a large enough training sample. They limited their input data graphs to 10x10 instances due to the number of these instances with known optimal solutions. They found that training on instances of this size could still find favorable results for larger instances of size 50x20. One of the limitations the authors highlighted was the lack of interpretability in their analysis, a limitation which traditional dispatching rules do not have.

GCNs and GATs have also been used to address scheduling problems. In their work, Wang and Gombolay (2020) create node features using a GAT to create a custom dispatching rule. As discussed, the key difference between GCNs and GATs is that GATs adaptively determine the significance of the neighboring nodes in the graph. This can be especially useful in complex graphs such as the disjunctive graph model, which was found to be true in their work. There has been some limited work in combining these approaches to gain further

benefits, but these are generally applied in other domains. For example, Dai et al. (2022) explore the application of both GAT and GCN layers to learn the features of circRNAs and diseases where the different layers allowed the learning of both the circRNAs and diseases simultaneously to help with drug discovery. For both the circRNA and diseases, the models were setup with GCN layers, GAT layers, followed by additional GCN layers.

The classification of JSSP instances is a task well suited to a GNN. The use of GNNs so far in the research have been used primarily to build specific dispatching rules. To this author's knowledge, they have not been used to classify similar disjunctive graphs to leverage their features to pre-seed the initial solutions used in an optimization algorithm.

Traditional graph classification is often a supervised task as shown by Zhang et al. (2018). To successfully implement a supervised classification system would require full knowledge and definition of different shop structures. In theory, this can be accomplished. As theoretical disjunctive graphs are constructed, their criterion for construction allows them to be classified. For example, a disjunctive graph where most parts are processed on a single machine could be grouped with another disjunctive graph where only 40% of the parts are processed on a single machine. In this example, similarities would be grouped based on the definitions of the user.

There has been recent work in classifying graphs with GCNs using unsupervised and semi-supervised techniques. Hui et al. (2020) propose a graph clustering method using a Gaussian mixture model to cluster and label data as an input to a supervised GCN classification

model. Caron et al. (2018) uses pseudo-labels generated from deep learning for unsupervised learning of visual features contained in digital images.

There has been some limited work to classify job shop problems, such as that by Nouri et al. (2016) where the job shop problem with transportation resources is classified manually. However, this was done to understand the types of problems and the standard approaches to solve them. This is the case in the literature reviews by Bergamashi et al. (1997) and Abdolrazzagh-Nezhad and Abdullah (2017) in which the job shop problem instances are classified to better understand the variations of the instances being studied, instead of looking at how a shop may change over time. Work by Su et al. (2023) used a GNN to create a graph level embedding, or mathematical representation, to feed into a policy network to make scheduling decisions. Their work used the disjunctive graphs of the JSSP in their current state (i.e. partially solved) to then determine the next job to select via an action selection policy network, this method was used to solve a variety of job shop instances with the largest having 100 jobs and 20 operations while accounting for both variation in processing times and RMBs.

As seen, the literature (i) on classifying JSSP instances to improve scheduling and (ii) using GCNs for JSSP graph classification is very limited. To this author's knowledge a combination of these techniques has yet to be implemented.

2.9 Real-World Scheduling and Lean Manufacturing

While research into how to solve the JSSP has progressed toward more complicated solutions, industry has moved somewhat in the opposite direction, striving for process simplification and a desire to drive out waste. These manufacturing process management

systems have evolved over time and were driven from various sources, none more influential than the Toyota Production System. To determine how a scheduling system would be implemented in a real-world shop, these approaches must be understood first.

The introduction of the Toyota Production System (TPS) marked a major evolution in thought regarding how to run a manufacturing environment. TPS was developed by Toyota in the 1940s-1970s and published in a guide by Taiichi Ohno (1988). TPS introduced principles centering on a system of continuous improvement and created a focus on the worker and removing worker obstacles. TPS also popularized a system of production sequencing known as *Kanban*, also known as a *pull system* or *just-in-time manufacturing*. The Kanban system had been in use previously but was popularized and refined by Ohno. A pull system uses a Kanban or signboard to track part inventory at each production step. The system uses a set of name cards to communicate to upstream operations. When the supply of a given part falls below a certain level, a card is sent upstream. This card tells the upstream system to create or deliver the given part. Each machine then processes requests in the order they are received. This effectively eliminates the need for scheduling, as scheduling is managed through creating demand signals based on what is needed. This system works well for assembly type job shops where a huge variety of parts are combined to deliver a relatively small variety of finished products, such as the automotive industry.

Krafcik (1988) outlined the triumph of the Lean Production System. In his article, he generalizes some of the features of TPS and rebrands them as *Lean*. From these, lean manufacturing continued to evolve, and it has been applied generally to many manufacturing environments. Lean's goal of continuing to improve processes is not necessarily antagonistic to

job shop scheduling optimization. However, due to its desire for continuous improvement and to empower shop floor personnel, Lean can create pressure for simplified scheduling methodologies. This can be done through leveraging techniques proposed by Ohno such as Kanban or moving toward one-piece flow.

In job shops there are often difficulties implementing Lean solutions. The variety of parts running through the shop and the potential for long setup times make one-piece flow impractical. Lean looks for ways to minimize setup times to get closer to one-piece flow. However, in practice, there is still often a need for setups. One practical way this problem is solved is using value streams.

Rother and Shook (1998) published their seminal work outlining the practice of value stream mapping (VSM) through studying Toyota's practice of material and information flow mapping. In VSM, a model is created of the production process, outlining each step involved in creating the product including both value-added activities and non-value-added activities. The practice is designed to identify where waste exists within the process and help eliminate it. One part of this practice is to identify product families. Product families are products that have similar processing steps and therefore can be captured in a single VSM.

Once VSMs are introduced, Rother and Shook recommend implementing continuous flow by combining processes. Where this is not possible, they suggest processing the jobs at each machine in a first in first out (FIFO) manner. The recommendation is then to implement virtual supermarkets. Like physical supermarkets the goal is to make sure all parts are available; to maintain a standard level of inventory the supermarkets utilize a Kanban system to send

signals to restock these supermarkets as they are depleted. Rother and Shook specifically advise against utilizing scheduling software or methodologies to control dispatching. They advise against scheduling software to keep the system simple, flexible, and to make sure there is manual engagement. These are all good goals, but they can still be met while using advanced scheduling methods.

There are still many benefits to lean manufacturing, especially in terms of processes and gaining efficiencies within them. However, there is a lack of research on how to bridge the gap between large-scale job shops and Lean methodologies. Irani (2020) provided general guidance for introducing Lean to large job shops. When it comes to how to handle sequencing of operations, a generalized scheduling software is recommended by Irani. This software, such as Opcenter by Siemens (2022) allows for flexibility and direct schedule optimization. Commercial software uses a variety of heuristics and direct optimization techniques. They can be good options for many companies; however, they are still limited by the same challenges previously outlined. For large complex shops, the optimization of the schedules is still a challenge, and the implementation of this software may not provide the desired outcome.

To avoid these issues, Lean practitioners often try to force an adherence to value streams. This can add unneeded costs through additional capacity, extra downtime, or extra inventory while reducing processing times and lowering WIP inventory. There has been extensive research on this as shown by the work of Sharma and Virmani (2020), Lian and Landeghem (2002), and Yu et al. (2009).

The creation of value streams allows for streamlined use of resources, but as seen in complex shops, the overlapping flow of different jobs across machines makes the creation of true value streams impossible. One other issue with large high-mix job shops is that, as jobs move through their routings and new jobs are released, the problem evolves causing the definition of value streams to evolve over time.

In an analysis that looked broadly at work on value stream mapping, Singh et al. (2011) considers value stream mapping's impact on Indian manufacturing. Their survey covers a range of works, finding that a popular technique utilized simulation to model and understand the implications of introducing value streams. Much of the work does not directly address the issue of scheduling and instead assumes the use of FIFO to schedule the shop. It then uses simulation to understand how to better set up the shop.

Simulation is a method used to understand how the implementation of a VSM could be beneficial to a job shop. McDonald et al. (2002) run simulation experiments to expand on the standard, static view of value stream mapping, allowing for a deeper understanding of how material flows in the shop. A similar approach was utilized by Gopakumar et al. (2008) where simulation is used to understand production with VSMs in order to identify operational inefficiencies in the process and address them through standard Lean methodologies. Pekarcikova et al. (2021) investigate how digitization can provide feedback from the shop floor and discuss how to implement these digital systems with Lean methodologies.

Dispatching rules are also considered in lean manufacturing research. In their work, Gracanin et al. (2013) analyzes a simple job shop with seven jobs and five machines with

multiple schedule variants analyzed. They look at a set of different dispatching rules applied to the entire shop to understand the total cost of different rules, including costs due to wait times. Their work ties scheduling impacts to decisions made based on Lean principles.

Many case studies show the value of VSM techniques and the improvements possible. These techniques are often applied to smaller job shops such as in the work by Simons (2006) at a Bosch factory. According to the study, VSM techniques reduced cycle times by 33% and changeover times by 81%. Work such as this makes it clear that meaningful improvements are possible for reasonably sized job shops and that the general techniques can be applied more broadly to improve job shops.

Many Lean strategies are applicable to smaller job shops. But issues may arise in large job shops where it is difficult to apply VSM techniques, such as the use of a simple scheduling rule, as seen in the previous research.

Managers of large job shops who look at the academic research are therefore left in an odd place. On one side, there is technical research that tries to fully understand the complex manufacturing environment and solve the associated specific JSSP applicable to a given manufacturing environment. This research has been hugely beneficial in certain industries and is used extensively in highly complex job shops such as silicon chip fabrication as discussed by Li et al. (2023). The other side recommends a simplification of the process, using basic techniques to manage flow as shown by Rother and Shook (1998). Sometimes this includes more modern technologies such as real-time feedback from various sensors as part of Industry 4.0. However, the scheduling remains simplistic as put forward by Valamede and Akkari (2020). Overall, Lean

has been successful in simpler job shops with less variety in the types of jobs processed. As complexity increases in a job shop, the implementation of Lean becomes more difficult.

2.10 Research Contributions

The research in this dissertation tries to solve a specific type of JSSP, specifically one represented by real-world conditions: the large, stochastic job shop problem with random machine breakdowns (RMBs). This is done through a few novel techniques. The first is the overall solution framework. This framework uses a GA to select machine dispatching rules from a list of known dispatching rules and determine the best rule for each machine through a simulation optimization framework that accounts for stochasticity in machine processing times and RMBs. The goal of this approach is to create a solution that scales to large shops and is easy to implement. This is a novel framework to this author's knowledge. The framework allows for new ways to leverage techniques. A key technique is to look for similarities in job shop graph topologies to pre-seed the GA. Most of the existing work using GNNs for JSSPs use them to create dispatching rules. Instead, this work uses a GNN to create graph embeddings to classify JSSP instances in order to identify good initial solutions used by a GA. There has been work in other domains using graph embeddings to learn precursor information for optimization problems. However, both within the domain of the JSSP and within the framework of pre-seeding a GA, this appears to be a novel technique.

This methodology proposed here learns and can be continually refined for a given job shop. As time passes, more schedules can be added to the training set for the GNN, which provides updated and improved initial solutions for the GA. The system is also robust. As

variability occurs, the dispatching rules remain constant, allowing the shop floor to implement the solution. Additional simulations can be run to monitor the system, and if variability becomes too large, the dispatching rules can be easily reoptimized. The use of dispatching rules makes the system easy to understand and easy to implement. They also fit within a job shop's existing lean manufacturing processes.

This work is differentiated from the current literature through the combination of both the way the GA is applied to solve the problem, and the use of the GNN to pre-seed the GA. There has been other work to use GA's to select dispatching rules, the most similar was by Yan and Wang (2007) where a JSSP problem was solved with machine specific dispatching rules, this work however did not account for RMBs in the simulation optimization framework, nor did it do any pre-seeding of the GA. The use of a GA to select dispatching rules has been explored subsequently by Vasquez-Rodriguez and Petrovic (2010) or by Rolf et, al. (2020) where a GA is used to modify a single shop level dispatching rule over time, alternating between different dispatching rules. Generally, when dispatching rules are studied, they are studied at the shop level, often through customized dispatching rules (Hildebrandt et. al. (2014)), or with standard dispatching rules are generally solved for smaller shops, such as the 200 job with 5 operation (1000 scheduled operations) solved by Yan and Wang (2007). When larger shops are solved for, such as by Teppan (2018) where 1000 job 100 operation (100,000) total operations were solved, only a single dispatching rule was used at the entire shop level, and uncertainty/stochasticity was not considered.

Another area this work tries to address is to design a process to pre-seed the GA. There has been some minimal work to pre-seed GA's through various techniques, some highlighting

the issues that can arise leading to poor solutions from pre-seeding as shown by (Oman and Cunningham (2001) as well as some cases where it can be successful as shown by Walsh and Fenton (2004) and Zhao et. al. (2005). Additionally, there has been some work done to analyze the entire disjunctive graph of the JSSP using GNN's such as the work by Su et. al. (2023), however in this work the graph was not fully classified and instead the representation of the graph was fed into another model. Generally, GNN's have been used for edge prediction to help build dispatching rules directly as shown by Wang and Gombolay (2020). To this author's knowledge the direct use of graph classification to pre-seed a GA has not been attempted.

Many real-world job shops experience problems when trying to implement a new system. Managers often require an understanding of how a new system will behave before they are willing to take the risk of implementation. Much of the previous literature ignores these business constraints and focuses on solving the abstract JSSP or focuses too heavily on these business constraints and proposes an overly simple solution. This dissertation, on the other hand, proposes an advanced solution that still allows everyone in an organization – managers, workers on the shop floor, and engineers – to understand how jobs are sequenced on the machines.

3. Methodology

3.1 The Setting

Every manufacturing environment is different, and these differences affect how a model of it is constructed. In this work, we consider a large manufacturing facility with considerable variability. This is based on the author's observations working for a large forged products

supplier. In a shop of this nature, many jobs exist at once, with each job consisting a minimum of 30 pieces. These pieces are processed by machines that may exist in cells where each cell has one to five identical machines. By having a high ratio of pieces to machines it allows calculating processing times for each operation based on the number of pieces in the job and the number of machines at the workstation. The large number of pieces in the job also means process times can be assumed to scale linearly. So, if one machine in a cell goes down, the total processing time for that a job would increase proportionally with the loss of operational capacity. For example, if one machine in a five-machine cell goes down, the cell would be at 4/5 capacity and any jobs run in the cell would take 5/4 as long to complete. These assumptions will be built implicitly into our modeling framework. All examples will operate under the assumption of large jobs and machine cells instead of individual machines processing single-piece jobs, though the size and scope of the shop may vary from this example.

In this research, time is discretized into *time steps*. Therefore, the processing time of each operation is rounded up to the nearest integer. Although in the examples the time for a given operation is unitless, in practice the time would be represented in minutes. This simplification was done due to the size and scale of the machine shop. Providing a schedule with accuracy to the nearest minute would be sufficient and therefore, any losses due to rounding would be lost in the variability that naturally exists within the job shop.

3.2 Defining Problem Instances

A standard methodology was developed to build job shop problem instances by constructing different disjunctive graphs. An algorithm was devised which creates randomized

graph structures based on various criteria. The algorithm builds out lists of machines that the operations in each job are assigned to and the processing times of those operations.

The creation of a JSSP instance depends on two key parameters. The first is the *job density*, which refers to density of operations for each machine. Let JobDensity_m be the relative likelihood that a random operation will take place on machine *m,* i.e. the job density of machine *m*. A machine with higher job density will have more operations assigned to it. The second is *processing time density*, which represents the average processing time of operations performed on a given machine. Let $PT\_density_m$ be the processing time density on machine *m*, i.e., the average processing time of operations on machine *m* relative to the average processing time of operations on other machines. For example, a processing time density of 2 will double the average processing time for operations assigned to that machine. These factors are referred to as density, as they increase the density of work at a given machine. Job density increases the number of operations at that machine, increasing the likelihood it is a bottleneck. Processing time density works similar in that the processing time for jobs assigned to those machines is longer, and therefore increases the total workload on those machines. When looking at the schedules, machines with higher density (job density and/or processing time density) are apparent as their schedules are more full (i.e., denser).

The process of creating a new problem instance is shown in Table 3. Table 4 shows the implementation of this process for the 3x3x3 problem instance shown in Figures 1-2.

Table 3: Job Shop Generator Algorithm

Job Shop Problem Generator	
1	**Input** the number of jobs J
2	**Input** the number of machines *Mach*
3	**Input** the number of operations Ops_i for each job i
4	**If** $Ops_i = Ops_j$ for all $i \neq j$, **Define** problem size as JxMxO.
5	**Let** $JobDensity_m$ be the job density for machine m
6	**Let** $PT\_Density_m$ be the processing time density for machine m
7	**Let** PT_Base be the base processing time of all operations for all jobs
8	**Let** PT_var be the variability of processing times where it is ± this number
9	**For** each i in jobs J:
10	**For** each k in operation Ops_i:
11	**Determine** M_{ik} as the machine operation (i,k) is assigned to, with a probability of that M_{ik} = m = $\frac{JobDensity_m}{\sum_{m=1}^{M} JobDensity_m}$
12	**Determine** t_{ik} as the processing time for operation (i,k), based on $t_{ik} = (PT\_Base)\left(PT\_Density_{M_{ik}}\right) + PT\_Perturbation$ where PT_Pertubbation is a uniformly distributed discrete random variable based on PT_var so PT_Pertubation ~ DU(-PT_var,PT_var)

Table 4: Build Graph Algorithm

Job Shop Problem Generator	
1	**Input** $J = 3$
2	**Input** $Mach = 3$
3	**Input** $Ops_i = 3$
4	**Define** as 3x3x3
5	**Let** $(JobDensity_1, JobDensity_2, JobDensity_3) = (2,2,1)$
6	**Let** $(PT\_Density_1, PT\_Density_2, PT\_Density_3) = (1,1,1)$
7	**Let** $PT\_Base = 4$
8	**Let** PT_var = 2
9	**For** each i in jobs J:
10	**For** each k in operation Ops_i:
11	$M_{1,1}$ = Random Choice(1,1,2,2,3) = 1 $M_{1,2}$ = Random Choice(1,1,2,2,3) = 3 $M_{1,3}$ = Random Choice(1,1,2,2,3) = 1 $M_{2,1}$ = Random Choice(1,1,2,2,3) = 1 $M_{2,2}$ = Random Choice(1,1,2,2,3) = 3 $M_{2,3}$ = Random Choice(1,1,2,2,3) = 2 $M_{3,1}$ = Random Choice(1,1,2,2,3) = 2 $M_{3,2}$ = Random Choice(1,1,2,2,3) = 1 $M_{3,3}$ = Random Choice(1,1,2,2,3) = 2
12	PT_Pertubation = (-2,2) $t_{1,1} = (4)(1)+DU(-2,2) = 5(1)+(-1) = 4$ $t_{1,2} = (4)(1)+DU(-2,2) = 5(1)+(-1) = 4$ $t_{1,3} = (4)(1)+DU(-2,2) = 5(1)+(0) = 5$ $t_{2,1} = (4)(1)+DU(-2,2) = 5(1)+(-2) = 3$ $t_{2,2} = (4)(1)+DU(-2,2) = 5(1)+(0) = 5$ $t_{2,3} = (4)(1)+DU(-2,2) = 5(1)+(0) = 5$ $t_{3,1} = (4)(1)+DU(-2,2) = 5(1)+(0) = 5$ $t_{3,2} = (4)(1)+DU(-2,2) = 5(1)+(2) = 7$ $t_{3,3} = (4)(1)+DU(-2,2) = 5(1)+(-2) = 3$

In Table 4, each job has 3 operations. In step 5 the job density was set to [2,2,1] which results in a higher likelihood that a job would be assigned to machine 1 or machine 2. This can be seen in the machine assignment where each operation had a random choice of being assigned to each machine, randomly selecting from a list of [1,1,2,2,3]. As seen, machine 1 and machine 2 are twice as likely to be selected as machine 3. Processing times work similarly.

According to step 6 the processing time density was identical for all machines, but if this were not the case, the processing time would be scaled based on that number.

To create a test problem set, a default job shop size of 10 jobs, 10 machines, 10 operations per job (10x10x10) was used. A variety of different criteria were chosen to create a set of unique job shop scenarios. For job density, four different loading levels were used of even, one bottleneck machine, two bottleneck machines, and ramped loading with a distribution of progressively loaded machines. A similar distribution was considered for processing times: even, one bottleneck machine, two bottleneck machines, and progressive loading. There are different ways in which processing time densities can align with the job densities (e.g., the bottleneck machine for processing time density could be the same as the bottleneck machine for job density or it could be different from it). The various permutations of how these could align created 24 different density *scenarios* shown in Table 5. Each of these scenarios were considered for three levels of variability low, medium, and high in the number of operations per job (Ops_i). Thus, a total of 72 problem instance *categories*, or *classes,* for 10x10x"10" instances were considered. (Parentheses have been placed around the final 10 to indicate that this is the expected number of operations per job, not the actual number of operations for every job.)

Table 5: Job and processing time density scenarios for the 10-machine instances considered in the experiments

Density Sequence	**Job Density**	**Processing Time Density**	**Correlation**	**Job Density, Processing Time Density for Machine Cell X**									
Density Sequence	Even, Ramping or 1 or 2 bottleneck Machines (bot-M)	Even, Ramping or 1 or 2 bottleneck Machines (bot-M)	Correlation between Job Density and Processing Time Density	1	2	3	4	5	6	7	8	9	10
1	Even	Even	N/A	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1
2	Even	1 Bot-M	N/A	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	**1,5**
3	Even	2 Bot-M	N/A	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	**1,5**	**1,5**
4	Even	Ramping	N/A	1,1	1,1	1,1	1,1	**1,2**	**1,2**	**1,2**	**1,3**	**1,3**	**1,4**
5	1 Bot-M	Even	N/A	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	**5,1**
6	1 Bot M	1 Bot-M	High	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	**5,5**
7	1 Bot-M	1 Bot-M	Low	1,1	1,1	1,1	1,1	1,1	1,1	1,1	**5,1**	1,1	**1,5**
8	1 Bot-M	2 Bot-M	High	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	**1,5**	**5,5**
9	1 Bot-M	2 Bot-M	Low	1,1	1,1	1,1	1,1	1,1	1,1	1,1	**5,1**	**1,5**	**1,5**
10	1 Bot-M	Ramping	High	1,1	1,1	1,1	1,1	**1,2**	**1,2**	**1,2**	**1,3**	**1,3**	**5,4**
11	1 Bot-M	Ramping	Medium	1,1	1,1	1,1	1,1	**1,2**	**1,2**	**1,2**	**5,3**	**1,3**	**1,4**
12	2 Bot-M	Even	N/A	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	**5,1**	**5,1**
13	2 Bot-M	1 Bot-M	High	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	**5,1**	**5,5**
14	2 Bot-M	1 Bot-M	Low	1,1	1,1	1,1	1,1	1,1	1,1	**5,1**	**5,1**	1,1	**1,5**
15	2 Bot-M	2 Bot-M	High	1,1	1,1	1,1	1,1	1,1	1,1	1,1	1,1	**5,5**	**5,5**
16	2 Bot-M	2 Bot-M	Medium	1,1	1,1	1,1	1,1	1,1	1,1	1,1	**5,1**	**5,5**	**1,5**
17	2 Bot-M	2 Bot-M	Low	1,1	1,1	1,1	1,1	1,1	1,1	**5,1**	**5,1**	**1,5**	**1,5**
18	2 Bot-M	Ramping	Very High	1,1	1,1	1,1	1,1	**1,2**	**1,2**	**1,2**	**1,3**	**5,3**	**5,4**
19	2 Bot-M	Ramping	High	1,1	1,1	1,1	1,1	**1,2**	**1,2**	**1,2**	**5,3**	**5,3**	**1,4**
20	2 Bot-M	Ramping	Medium	1,1	1,1	1,1	1,1	**1,2**	**1,2**	**5,2**	**5,3**	**1,3**	**1,4**
21	Ramping	Even	N/A	1,1	1,1	1,1	1,1	**2,1**	**2,1**	**2,1**	**3,1**	**3,1**	**4,1**
22	Ramping	1 Bot-M	High	1,1	1,1	1,1	1,1	**2,1**	**2,1**	**2,1**	**3,1**	**3,1**	**4,5**
23	Ramping	2 Bot-M	High	1,1	1,1	1,1	1,1	**2,1**	**2,1**	**2,1**	**3,1**	**3,5**	**4,5**
24	Ramping	Ramping	High	1,1	1,1	1,1	1,1						

3.3 Accounting for Uncertainty

Another aspect of real-world scheduling that is considered in our framework is uncertainty. In real-world scheduling, variability can come in two major forms. The first, and most common, is variability in processing times. The second, and more impactful, form of uncertainty is machine downtime. In this work we remove variability in our first two sets of experiments to simplify comparisons when fine-tuning our methodology. After that, we consider variability to see how the methodology performs for real-world problems.

There are a few types of variability that will be assessed. Regarding variable processing times, every machine operator processes parts at a different pace. Even for highly automated operations, variabilities can occur due to handling, loading time, setup time, and for variability in other operations such as extra time to fix defects. All these result in uncertain processing times. To account for this, every machine in our modeling framework is given a predefined process time variability. This variability defines a range of possible processing times. For example, variability could be represented as a uniformly distributed ratio of processing times: if there is 10% variability for an operation, the actual processing time will be between 90% and 110% of the expected processing time t_{ik}.

The second variability comes from RMBs. The RMBs within each machine cell and the subsequent machine repairs in the cell resemble a birth and death process in which births correspond to machine breakdowns and deaths correspond to machines being repaired. A standard birth and death process is a Markov process in which the state of the system is a non-negative integer that equals the number of living entities in the system. The process is defined

by two characteristics. The first, λ is the Poisson rate at which births occur. The second, μ is the Poisson rate at which deaths occur. The standard birth and death process can be seen below in Figure 16.

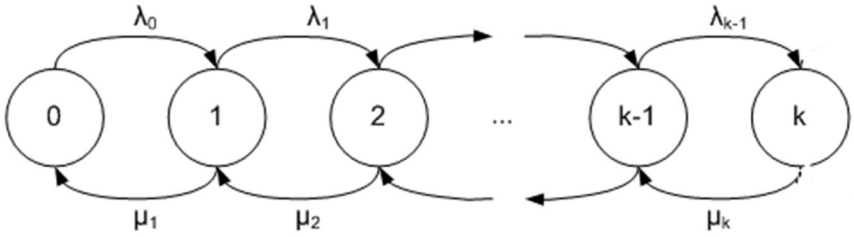


Figure 16: Standard birth and death process

The RMB assumptions in this work differs from those of a birth and death process in order to replicate a real job shop as best as possible. For each machine cell, there is a constant, non-zero probability that one machine in the cell will break down. (This probability is zero if all machines are broken.) This assumption resembles a birth and death process. However, repair time cannot be accurately modeled as a Markov process. Repair time is better represented as a fixed time, with variation applied for each machine that is broken. For example, the repair time for a broken machine could be modeled as taking between 3 and 7 time steps instead of an equal probability of repair within the cell as a whole at each timestep. This is because repairs take a minimum amount of time due to the need to assess the problem and get the right resources in place to address the issue. The types of breakdowns are normally well defined, so they will fall within a general range of expected downtime values. There may be chances of catastrophic failures that cause extreme downtime which are not accounted for in our modeling framework. In these cases, the situation has drastically changed, and the problem instance should be completely re-solved to account for these changes.

Our overall method for modeling RMBs is as follows. Each machine cell is assumed to have a set number of machines *k*, and initially all machines are operational. At each timestep of the simulation, each machine cell has a probability of experiencing a machine breakdown. (We do not allow two machines to simultaneously breakdown in the same cell.) If a machine breaks down, the repair time is calculated at that point based on a uniformly distributed repair time. The states of the machines are calculated in advance for the entire time horizon. These calculations are performed in advance to improve computational efficiency.

The impact of RMBs and variable process times can be seen in the examples below. In Figure 17, an example of a schedule is shown with no variability. In this instance, which is identical to Figure 2, there are three jobs processing across three machine cells, and each machine cell contains two machines.

Figure 17: Detailed schedule for a 3-job, 3-machine, 3-operation instance with two machines per cell and no machine breakdowns (same as Figure 2)

With the introduction of RMBs, suppose that one of the machines in cell 2 breaks down at time 3 and takes 8 time steps to repair. Additionally, cell 3 has two breakdowns; the first at time 10 which takes 3 time steps to repair, and the second at time 12 which also takes 3 time steps to repair. The effect of these breakdowns on the schedule can be seen in Figure 18.

Figure 18: Detailed schedule for a 3-job, 3-machine, 3-operation instance with two machines per cell and three machine breakdowns

Finally, random variations in processing times are introduced. In this instance, operation 1 of job 2 and operation 2 of job 3 each take one extra time step to complete, and operation 3 of job 3 takes one less time step to complete. The impacts can be seen in Figure 19.

Figure 19: Detailed schedule for a 3-job, 3-machine, 3-operation instance with two machines per cell, three machine breakdowns, and processing time variability

These variabilities can cause a certain scheduling order that was optimal to become non-optimal, especially in larger problem instances. By studying these variabilities, the robustness of a scheduling rule can be determined.

3.4 Dispatching Rules

The literature shows that for large stochastic problem instances, dispatching rules are a promising method of finding schedules. Dispatching rules reduce the problem space, allowing for larger job shops to be scheduled. Additionally, by using dispatching rules, the implementation of the schedule becomes easier. Instead of needing a tight schedule with start

and stop times for all jobs, a simple rule can be followed by the machine cell, allowing the schedule to be easily implemented. When variability is introduced, the dispatching rule may choose a different sequencing order and result in a different schedule. This allows the schedule to adapt naturally to changes in the environment.

In the literature, there are two popular ways to address dispatching rules. The first is through the introduction and implementation of a specialized dispatching rule trained for the entire shop; the second is through careful use of standard dispatching rules. For this work, standard dispatching rules will be explored. There is little research on this, none of which examines the use of machine-specific dispatching rules for problems with RMB. This research will explore novel ways of implementing standard dispatching rules.

To improve the model's performance, we allow each machine to have its own dispatching rule. Since a supervisor will oversee one or only a few machine cells, there is reduced overhead managing these rules and this avoids extra complexity when implementing the schedules. This ease of implementation will be discussed in more depth in the discussion. Dispatching rules considered in this research are outlined in Table 6.

Table 6: Dispatching Rules

Dispatching Rule	Description
First In First Out (FIFO):	The first job to arrive at the machine is processed first.
Last In First Out (LIFO)	The last job to arrive at the machine is processed first.
Shortest Processing Time (SPT)	The job with the current operation having the shortest processing time is processed first.
Longest Processing Time (LPT)	The job with the current operation having the longest processing time is processed first.
Shortest Total Processing Time (STPT)	The job with the shortest total processing time across all operations is processed first.
Longest Total Processing Time (LTPT)	The job with the longest total processing time across all operations is processed first.
Least Operations Remaining (LOR)	The job with the least number of remaining operations is processed first.
Most Operations Remaining (MOR)	The job with the greatest number of remaining operations is processed first.
Least Work in Next Queue (LWINQ or LQNO)	The job whose next operation is at a machine that has the least amount of waiting jobs, is processed first.
Most Work in Next Queue (WINQ or MQNO)	The job whose next operation is at a machine that has the most waiting jobs is processed first.
Ratio of remaining time (Ratio RT)	Calculate the ratio of the remaining processing time on the job to the job's total processing time, scheduling the largest ratios first (those with the most relative amount of time to complete first).

3.5 Simulation Optimization

The overall methodology for solving this problem uses a GA and a simulation optimization framework. Each chromosome in each generation of the GA is a set of dispatching rules for the machines. The fitness of each chromosome is evaluated by running one or more simulation replications using a custom-built discrete event simulation model. The simulation model automatically converts (i) the parameters of a problem instance such as those shown in Table 5, (ii) the set of dispatching rules for the machines (Section 3.4), and (iii) any additional uncertainty (Section 3.3) into a detailed job shop schedule. The use of simulations allows for the introduction and accounting of randomness without the need for completely calculating the distribution of events, at the outset. Simulation constrains the methods used for optimization. Since each simulation replication presents a different realization of a problem instance, simulation helps to assess the impacts of different types of variability in the SJSSP with RMB.

In the simulation models, whenever a machine cell becomes idle and jobs are waiting in its queue, a priority array is calculated to determine which operation is performed next on the machine. The values in the priority array are based on the machine's dispatching rule as specified by the current chromosome. These values are sorted from smallest to largest, and the highest priority job is performed next on the machine. Then the simulation model advances to the next clock time when a machine cell becomes idle and jobs are waiting in its queue, and the process repeats. Importantly, when a scheduling decision is made, only the jobs in a machine's queue can be scheduled.

Once the entire schedule is created, the fitness of the schedule is calculated (e.g., using the total completion time metric). This is done for each chromosome in the current generation using the above methodology. At this point, the next generation is created as follows. First, the population is sorted by fitness. The fittest F% of the chromosomes are directly copied into the next generation. The remaining members of the next generation are created using a random combination of two of the existing chromosomes. Any of the best B% of the chromosomes in the current generation has a chance to be selected, with the more fit entities being more likely to be selected. Two such individuals are selected and a random crossover point is determined. The chromosomes are both cut at this point, and the end of one chromosome is combined with the beginning of the second. There is a C% chance this individual will also mutate. The mutation will randomly select several genes (i.e., machines) in the chromosome and modify their values, altering the dispatching rules for these machines. Thus, the mutation keeps part of the original set of genes intact, while randomizing the others. The mutation works by randomly selecting a set of genes in the chromosome and randomizing them. (i.e., a subset of the machines is selected, these are given a new, random, dispatching rule). Each gene is randomly given a value of 0 or 1, where 1 represents that it will be mutated, if it is mutated it is randomly assigned a new value, with all values being equally likely. The values of F, B, and C were determined after conducting tests at various levels to maximize the rate at which the solution improves. The GA procedure is shown in Table 7. The complete process is shown in Figure 20.

Table 7: Genetic Algorithm

Genetic Algorithm	
1	**Let** *machs* be a list of lists containing the machine each operation is scheduled on such that for job *i*, operation *j machs(i,j)* = machine *m*
2	**Let** *ptime* be a list of lists containing the time t each operation takes to complete the operation such that for job *i*, operation *j ptime(i,j)* = *t*
3	**Let** *generation* be a list of N chromosomes, each chromosome has a random dispatching rule assigned to each machine
4	**Let N***umGenerations* be the total number of generations the GA runs
5	**For** each dispatching rule: update one chromosome in *generation* so that every machine uses that dispatching rule in that chromosome
6	**For** *i* = 1 to *NumGenerations*:
7	**For** each chromosome in *generation i*:
8	**For S** simulation replications:
9	Run the simulation model to translate the parameters in the problem instance, the chromosome at hand, and any uncertainty into a detail schedule to get *Starts*=**Simulate**(chromosome,$J,Ops_i,Mach,M_{ik}.t_{ik}$)
10	*fitness* = average objective value for all replications
11	*breeding_chromosomes* = set of the fittest (F%)(N) chromosomes in generation *i*
12	*next_generation* = list containing top 1/10 of population by fitness from generation
13	**While** length(*next_generation*) < N:
14	*g1* = a random element in *breeding_chromosomes* based on fitness
15	*g2* = a random element in *breeding_chromosomes* based on fitness
16	*break_pt* = random crossover point
17	*new_chromosome*= portion of g1 prior to break_pt + portion of g2 after break_pt
18	**If** random uniform number between 0 and 1 < C/100:
19	*new_chromosome* = mutated by randomly selecting a random number of genes (DU(0,1)) in the chromosome and setting them to new random genes (dispatching rules)
20	Append *new_chromosme* to *next_generation*
21	**Let** the set of chromosomes in generation *i*+1 = *next_generation*

The Simulate algorithm, which is called in like 9 of the GA procedure are shown in Table 8. This algorithm involves modifications to include both the RMB and random variations in

processing time. The Simulate algorithm is more verbose to ensure the nuances of these calculations are clear.

Table 8: Simulate Algorithm

Simulate (chromosome,J,Ops_i,$Mach$,M_{ik}.t_{ik})

Let *random_time* = a random amount that each operations time is adjusted for
Let each machine have an expected rate that it will break down, and expected amount of time to repair, as well as an expected amount of variability in that time.
Let *total_time* be equal to all processing times *t* representing the max length of the schedule
Let *number_machine* be list that updates over time keeping track of the number of machines available where *number_machine*[*m*][0] = the number of total machines in the machine cell
Let *time_machine* track the time that a change in the number of machines available, initialize with *time_machine*[*m*][0] = 0
Let *repair_time* track the time a machine breakdown will be repaired, initialize the first instance with all machines available *repair_time*[*m*][0] = 0
Let *machine_available_time* be a list for all machines representing when the last operation finished running, initialize to 0 for all machines.
For *time* in *total_time*:
    For *m* in *Mach*:
        n = 0
        Let *last_time_checked* = 0 where it tracks the last instance of n that was checked
        Let *machine_list* be an empty list to track the number of machines who's repairs need to be accounted for
        If probability of *m* breaking down ≥ random uniform number between 0 and 1:
            If *number_machine*[*m*][*n*] < machines in machine cell:
                While *machine_list*:
                    k = pop(*machine_list*)
                    If *repair_time*[*m*][*k*] ≤ *time*:
                        n = *n*+1
                        number_machine[*m*][*n*]= *number_machine*[*m*][*n*-1] +1
                        time_machine[*m*][*n*] = *time*
                    Else:
                        Break While Loop
            n=*n*+1
            number_machine[*m*][*n*] = *number_machine*[*m*][*n*-1] -1
            time_machine[*m*][*n*] = *time*
            Append *n* to *machine_list*
            repair_time[*m*][*n*] = *time* + expected repair time + variability of repair time
For each *job* and *operation*: Update t_{ik} based on variations in t_{ik}

```
29  For each job:
30      Calculate priority value based on dispatching rule of machine from gene
31      Append priority_list with priority value, job and operation
32  Let current_break = 0 for all machines, to track the place in the
    number_machine/time_machine list for current work
33  Select next machine that is available for scheduling as m
34  While priority_list:
35      Pop highest priority job and operation on machine m from priority_list
36      Let consumed_time be 0
37      Let current_break =0
38          If starts[job][operation]+t[job][operation]
            >Machine_Available_Time[M[job][operation]]:
39                  starts[job][operation] = starts[job][operation-1]+t[job][operation-1]+1
40          Else:
41                  starts[job][operation] = machine_available_time[M[job][operation] + 1
42      Else:
43          starts[job][operation] = machine_available_time[M[job][operation]] + 1
44      Let break_time = starts[job][operation]
45      While consumed_time < t[job][operation]
46          efficiency = (total_machines[[M[job][operation]]  -
            number_machines[M[job][operation]][current_break] )/
            total_machines[[M[job][operation]]
47          If efficiency = 0:
48                  consumed_time = consumed_time
49          Else:
50                  If consumed_time +
                    (time_machine[machs[job][operation]][current_break+1] -
                    time_machine[machs[job][operation]][current_break])/efficiency <
                    t[job][operation]:
51                          consumed_time = consumed_time +
                            (time_machine[machs[job][operation]][current_break+1] -
                            time_machine[machs[job][operation]][current_break])/efficiency
52                  Else:
53                          endtime = (t[job][operation]-consumed_time)/efficiency +
                            time_machine[M[job][operation]][current_break]]
54          current_break = current_break +1
55      t[job][operation] = round(endtime – starts[job][operation])
56      Select next machine that is available for scheduling as m based on time_machine
57      Update priority for machine m by referencing priority equation based on
        dispatching rule assigned to machine per the chromosome passed in gene
58  Return(starts)
```

The simulation is a highly complex process. The key items are the process of scheduling and updating the list starting at line 34. This process starts with a machine identified and looks for the job with the highest priority to be scheduled. It then schedules that job; this includes updating the time the machine is available and the processing time of the job and passing that job to the next machine. After this is complete, the next machine is identified based on availability. This process checks each machine and moves to the next time a machine is available, without the need to simulate each timestep.

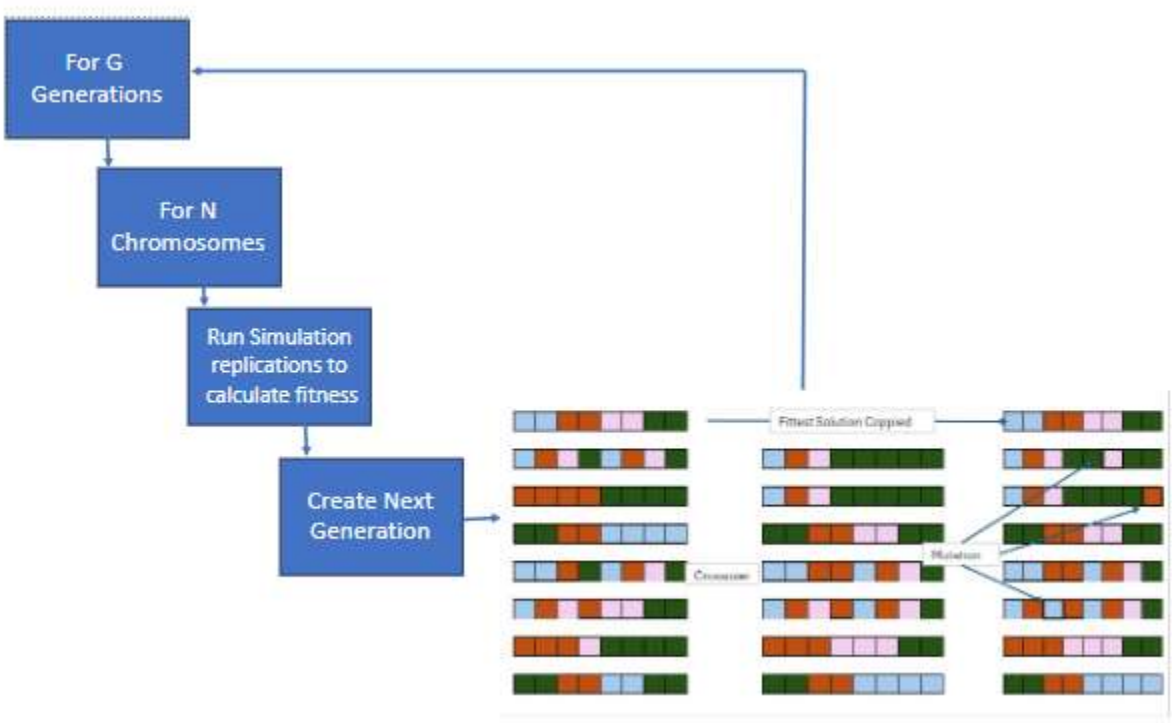


Figure 20: Overall simulation optimization framework

Clearly, the simulation procedure is the most computationally intensive part of the process. Each operation must be scheduled. For a 10 job, 10 machine, 10 operation instance, 100 total operations must be scheduled. Each scheduling step requires iterating through multiple lists. The priority lists of jobs queuing at the machines can vary in length. The longest

possible priority list has all *J* jobs in it. The lists maintaining priority are not sorted. Instead, the next operation to be scheduled is swapped to the back of the list then it is removed through a pop operation, this becomes a negligible amount of time, O(1) for the swap, O(1) for the pop.

The start time of each operation also depends on machine breakdowns (see Figures 18-19). To improve simulation efficiency, we precompute for each machine cell the times when machines break down and are repaired during the first 1000 time-steps prior to the beginning of each simulation run. If more than 1000 time-steps need to be simulated, the breakdown and repair events during time steps 1-1000 are repeated in time steps 1001-2000 and every thousand steps thereafter. This allows sufficient variability without overwhelming the computing resources. Nevertheless, the process of deciding which event occurs next in the simulation is still computationally non-trivial. For example. The simulation model must iterate through all machines to identify which is the next available. Then that machine's priority list must be updated by iterating through the number of jobs waiting at that machine.

With this process outlined, the time complexity for the simulation process can be defined a JxMxO job shop instance. During the simulation every operation needs to be scheduled so this takes J*Ops_i updates. At each scheduling step, the next job is determined to be scheduled, this requires looking at each machine, taking *M* time, then that priority list must be updated since there are some dispatching rules that change over time (such as WINQ – work in next queue). Worst case there is a machine with every job waiting to be scheduled, so therefore checking the priority list takes up to *J* time. Once these are updated the job with the highest priority is scheduled, which requires checking this priority list, taking *J* time steps. The priority list doesn't need to be sorted, as mentioned, and instead relies on swapping list indices

and then popping from the end of the list updates on priority. Additionally, there are multiple fixed time operations. In other words, the simulation algorithm, takes O($J*O*(J+J+M+C$)) where C is a non-insignificant constant. The total computation time of the simulation model is O($J^2O+J*O*M$). This results in the simulation being the most significant computation time section of the code.

4. Preliminary Results: Comparing the GA to Mixed Integer Programming

4.1 Experimental Setup

A set of preliminary trials were run to verify the feasibility of the proposed methodology. Five different problem instances were analyzed, each representing job shop sizes ranging from small to medium. The GA simulation-optimization framework's performance was then compared against a mixed-integer programming solver that was given a computation time limit of 10,000 seconds. The comparison between these two methods was used to determine the feasibility of the proposed algorithm.

One of the goals of the proposed framework is to outperform a MIP solver. If the preliminary results show improvements in the solution over MIP for medium-sized problems, then it is clear that for even larger problems, the proposed GA simulation optimization framework (GASO framework) will also outperform MIP. Both the GASO framework and the MIP framework were implemented in Python. To reduce overall computation time, this section of the code was implemented in C and called via Python during the running of the GA. Through

a combination of minimizing time complexity and implementing this in C, the resulting algorithm was 20 times faster than the original implementation written in Python.

The MIP framework used the open-source package MIP, which is part of the COIN OR Project and the MIP that was solved is shown in equations 1-5 in Section 1.2.

Table 9: GA settings for preliminary experiments

Problem Setup

GA

Number of Chromosomes	100
Number of Generations	100
Percent of Chromosomes copied to next Generation (F)	10
Percent of Chromosomes Participating in Crossover (B)	88
Percent of Chromosome that are Mutated (C)	35
Simulations of each Chromosome (S)	10

The GA settings are shown in Table 9. The GASO framework was implemented with 100 chromosomes that were allowed to run for a total of 100 generations. Each chromosome contained a set of dispatching rules. To ensure all dispatching rules were given a chance, the initial population was pre-seeded with one chromosome for each dispatching rule in which every machine used that dispatching rule. The remaining chromosomes were generated randomly. This gave a baseline performance for that schedule, and allowed for comparison to the MIP formulation, which does not account for any randomness. The experiments were run on an Intel i7-3770K 3.50 GHz processor with 16 GB of RAM.

4.2 Experimental Results and Discussion

The five instances that are considered, and the experimental results for each, are summarized in Table 10. Each instance has a set number of jobs, operations, and machines. The first instance was non-symmetric, with a few jobs having three operations and a few having four operations. The remaining instances each have a standard number of operations per job as outlined below. In instances 1-4, all machines are randomly loaded, with an equal chance of any operation being performed on any given machine. For instance 5, 80% of the operations are randomly assigned to the first four machines and 20% of the operations are randomly split amongst the remaining six machines; this represents a case with multiple bottleneck machines.

Table 10: Initial Problems Analyzed to Compare to MIP

Instance	No. Jobs	No. Machines	No. Operations per job	Total No. operations	GA total completion time	GA CPU time (sec)	MIP total completion time	MIP CPU time (sec)
1	6	4	3-4	21	128	11	126	6
2	10	10	10	100	7,622	33	7,288	10,000
3	50	9	10	500	10,325	208	NA	10,000
4	30	30	30	900	41,081	300	NA	10,000
5	30	10	10	300	45,534	159	169,815	10,000

There is an interesting inflection point within the analysis, where instances 1 and 2 result in the MIP solver performing better. However, beyond this, the GA performs better with lower objective values and computation times. In all cases, the GA found a solution in well under 10,000 seconds, but in many cases the MIP solver reached its time limit. In two cases no feasible solution was found by the MIP solver within 10,000 seconds.

A few items of particular interest can be observed in Table 10. First, the GA does not find an optimal solution. Due to the reliance on dispatching rules and variability considered by the simulation model, it is extremely unlikely that the GA will find an optimal solution, regardless of the amount of computation time given. However, it does find a good solution. This can be seen in instance 1 which MIP easily finds an optimal solution, but the GA does not. However, for the instances in which MIP computation time becomes an issue, the GA finds better performing solutions.

Second, the performance of the GA scales well with instance complexity. As seen in the examples, the GA maxed out at 300 seconds for computation time. The GA has a higher computation time than MIP for the smallest instance, but since it scales better, it can be useful for very large job shops. The GA reached similar solutions for the first two problems, showing that while not optimal, it can find good solutions.

Figures 21 and 22 show the best solutions found by the MIP solver and the GA for the smallest instance. The optimal solution found by MIP, which has a total completion time of 126, is shown in Figure 21. The problem was then analyzed with the GA which resulted in a completion time of 128 as seen in Figure 22.

Due to the lack of options, many dispatching rules will lead to the same or similar results for small instances like the one shown here. The GA schedule was generated using the following dispatching rules respectively in machine order:

Machine 1: Least Work in Next Queue (LWINQ)

Machine 2: Shortest Total Processing Time (STPT)

Machine 3: Longest Processing Time (LPT)

Machine 4: Shortest Total Processing Time (STPTP)

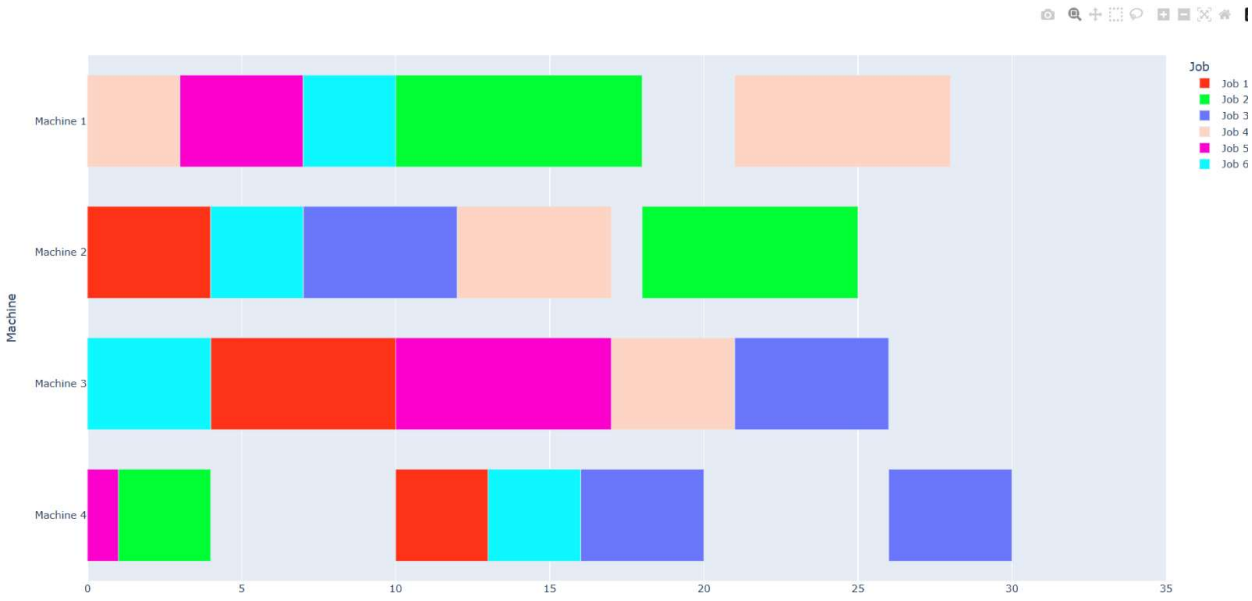


Figure 21: MIP Output of Problem 1

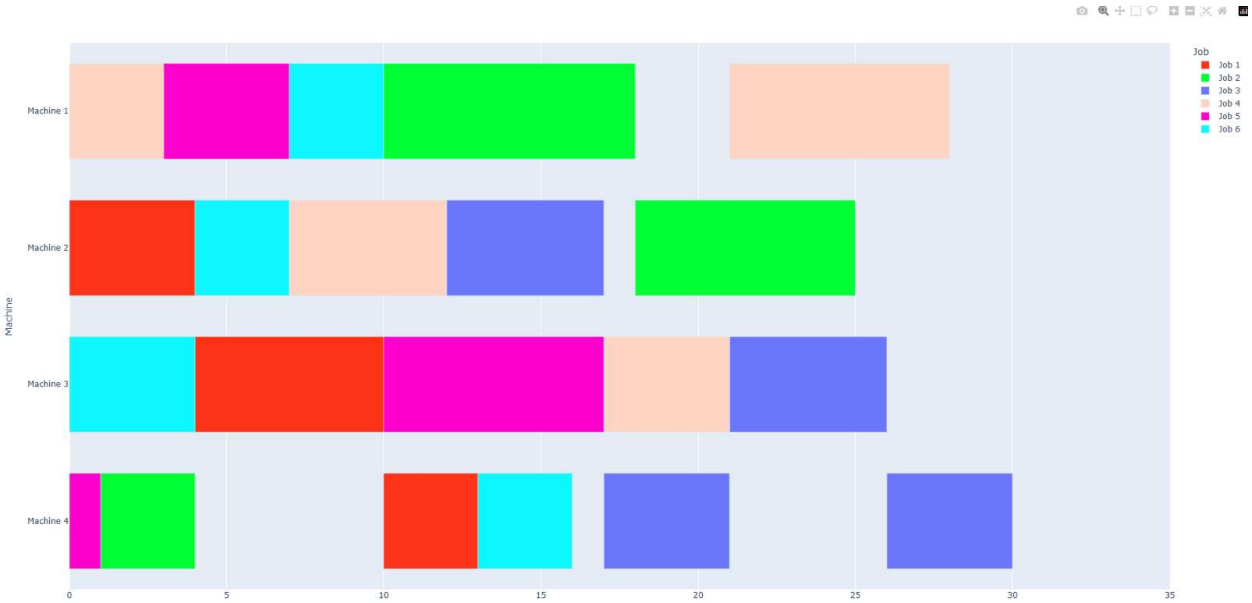


Figure 22: GA Output of Problem 1

Figures 23-24 show the stark difference between the MIP and GA solutions for instance 5. In this instance, there are four bottleneck machines, adding extra complexity due to the number of disjunctive edges the MIP solver needs to resolve. The MIP solver found a feasible solution, but as seen in Figure 23, the solution is not dense. This can be compared to Figure 24 where the GA solution's schedule is much denser, providing significant reduction in the overall completion time.

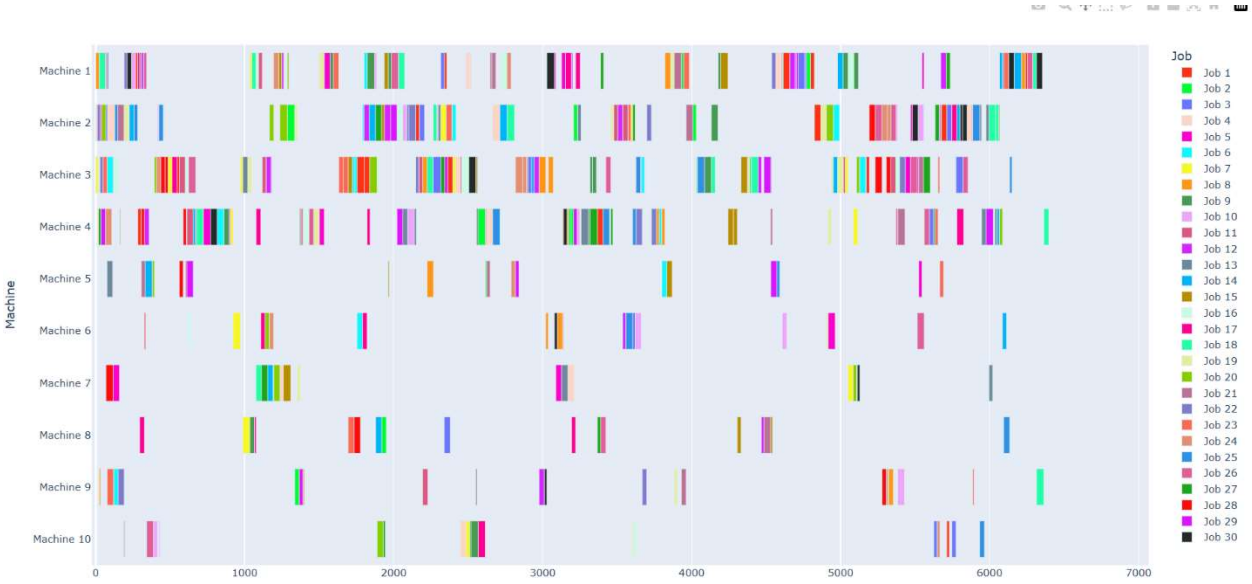


Figure 23: MIP Output of Problem 5

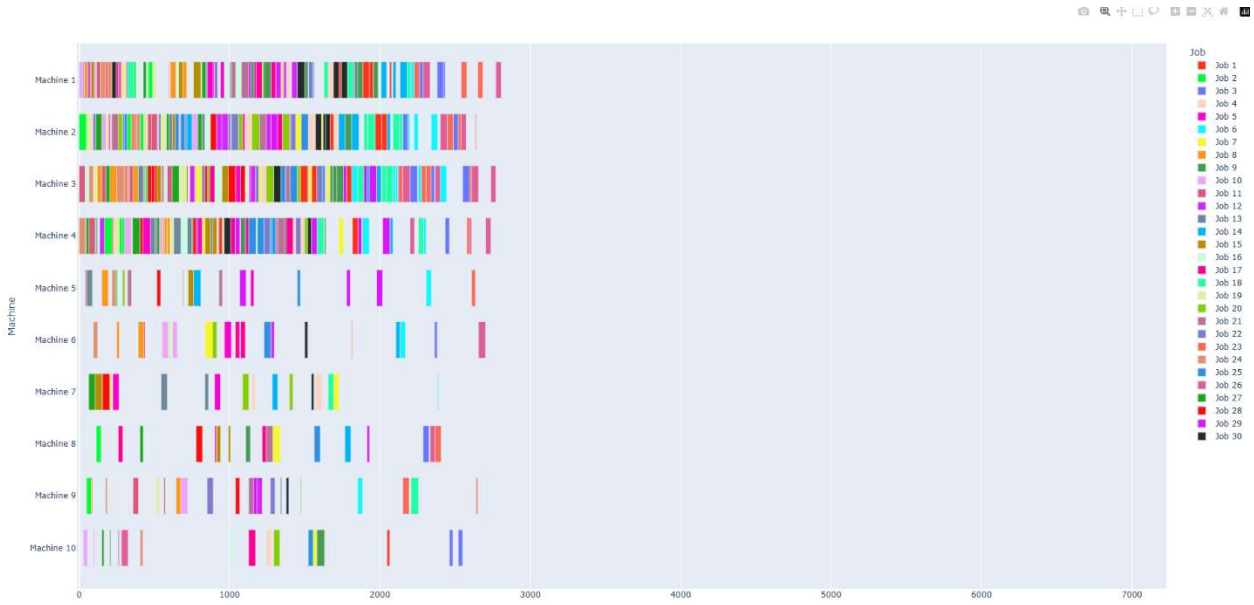


Figure 24: GA Output of Problem 5

Overall, the GA significantly outperforms a classic MIP approach while still not reaching a true optimal solution. This allows the approach to scale to large real-world job shops.

5. Refined Methodology

5.1 Motivation for a Refined Methodology

As demonstrated above, the general framework has allowed for the creation of schedules that perform well for large, complex shops. However, the results might be highly driven by the initial population, i.e., the set of (random) chromosomes in the first generation. In addition, for very large problems, the simulation time may become cumbersome, causing further reliance on the initial seeded population.

An open question within the study of job shops is whether similar patterns within problem instances can be used to identify favorable solutions for new problem instances that

share the pattern. Specifically, can solutions for small problem instances be scaled to larger problem instances? If there are enough similarities in the disjunctive graph between two instances, will similar dispatching rules provide similar levels of performance? This work will attempt to answer these questions while proposing a solution to solve larger problems.

Our overall motivation for developing a refined methodology is to test several hypotheses. Our first hypothesis is that it is possible to train a GNN to successfully classify JSSP instances of any size. The second hypothesis that if two problem instances belong to the same graph class, then similar solutions will work well for both. For example, if a job shop has a single bottleneck machine and a with similar distribution of work on the other machines, the types of dispatching rules that work well for the bottleneck machine and the non-bottleneck machines will be the same whether the shop has 10 machines or 100. The third hypothesis assumption is that if we initialize the GA with solutions that resemble elite solutions that have been identified for other instances belonging to the same graph class, the GA will converge to a good solution more quickly.

Applying the known solutions can be easily done if the class of the problem is known, and if the problem is the same size as the training set. In many cases, neither of these will be true so a process must be designed to address both. Scaling the shop is relatively straightforward. If the class is known, then the variability in the operations worked on each machine can be calculated and a mapping of each machine can be created. This mapping will determine which machine from the smaller 10x10x10 job shop the machine from the larger job shop corresponds to. To classify the different types of graphs, a GNN will be used.

5.2 Updated Solution Framework: Overview

The preceding discussion leads to an updated, three-phase solution methodology shown in Figure 25. In the first phase, a set of 10x10x10 problem instances – 150 for each of the 72 graph classes mentioned in section 3.2 – are generated; the GASO framework is run on these instances; and the best 20 solutions for each graph class are stored. Next, a GNN is trained to classify which of the 72 graph classes a problem instance belongs to. The training set consists of 150 10x10x10 instances, ten 20x20x20 instances, ten 30x30x30 instances, and ten 40x40x40 instances for each of the 72 job shop classes. In the third and final phase, experiments are conducted to test the effectiveness of the methodology on JSSP instances of any size. Each problem instance is first classified into one of the 72 graph classes by a trained GNN. Then the first generation of the GA is initialized with chromosomes that resemble elite solutions that were identified for the 10x10x10 instances belonging to the same graph class as in phase 1. Finally, the GA is run with these elite initial chromosomes and the experimental results are recorded. This process is shown in Figure 25.

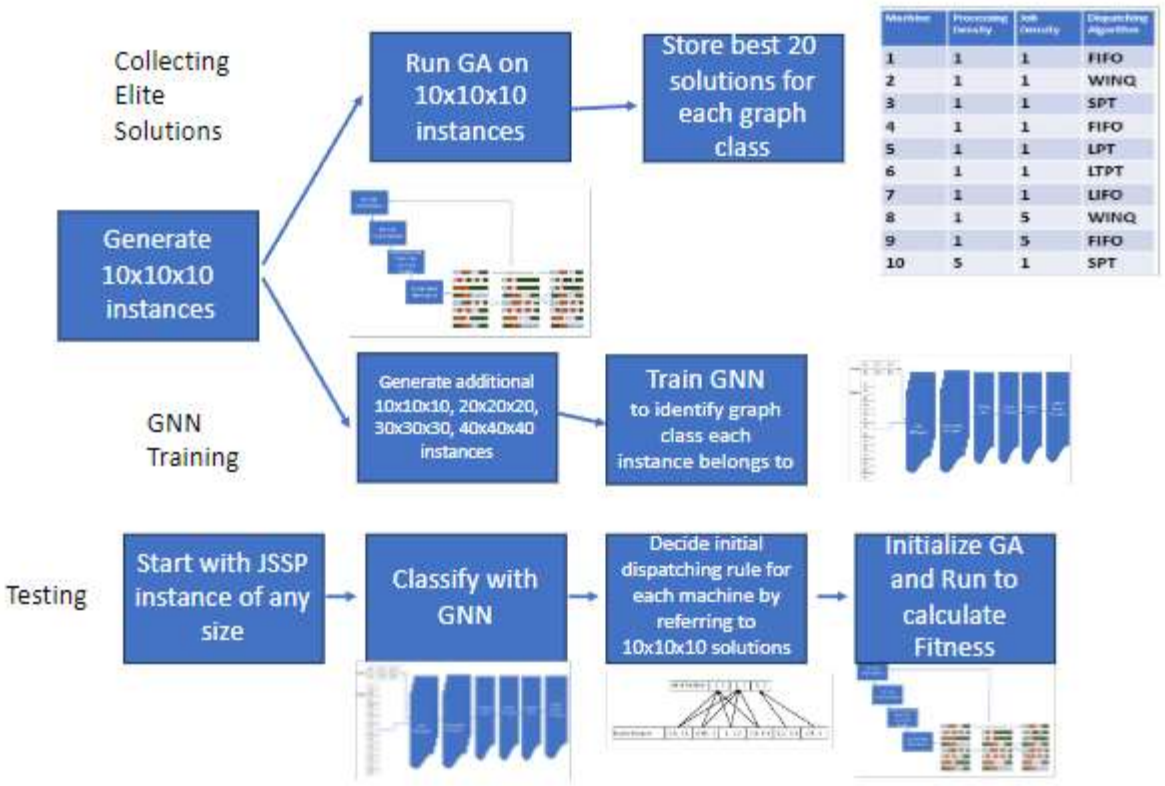


Figure 25: Complete Process Flow

5.3 Updated Solution Framework: Details

We now discuss phases 2 and 3 of the updated solution framework in detail. Phase 2 is the training of the GNN to classify problem instances into one of 72 graph classes. The 10x10x10 instances used to train the GNN are created as described in section 3.2. The larger problem instances (20x20x20, 30x30x30, 40x40x40) use the same methodology as the 10x10x10 instances except they are the probabilities outlined in Table 5 are doubled, tripled, or quadrupled. This means that for a one bottleneck problem, instead of one bottleneck and nine non-bottlenecks, there are two bottlenecks and eighteen non-bottlenecks. This is the same methodology used to generate the larger problems for testing that will be discussed in Section 6.

For each training instance, the GNN takes the node labels shown in Figure 11 and the adjacency matrix of the problem instance's disjunctive graph inputs. The GNN is trained using a combination of both graph attention (GAT) and Graph Convolutional (GCN) layers, which both aggregate the nearby graph features. Next, the model uses a pooling layer to aggregate the node representations which results in a coarser structure. The model next contains a set of dense layers to capture the complex relationships within the graph. Then, a dropout layer is used to reduce overfitting by randomly dropping 10% of the node features during training to build more robust representations. The results are passed through a final set of dense layers to reach the final result. The overall structure of the GNN is shown below in Figure 26.

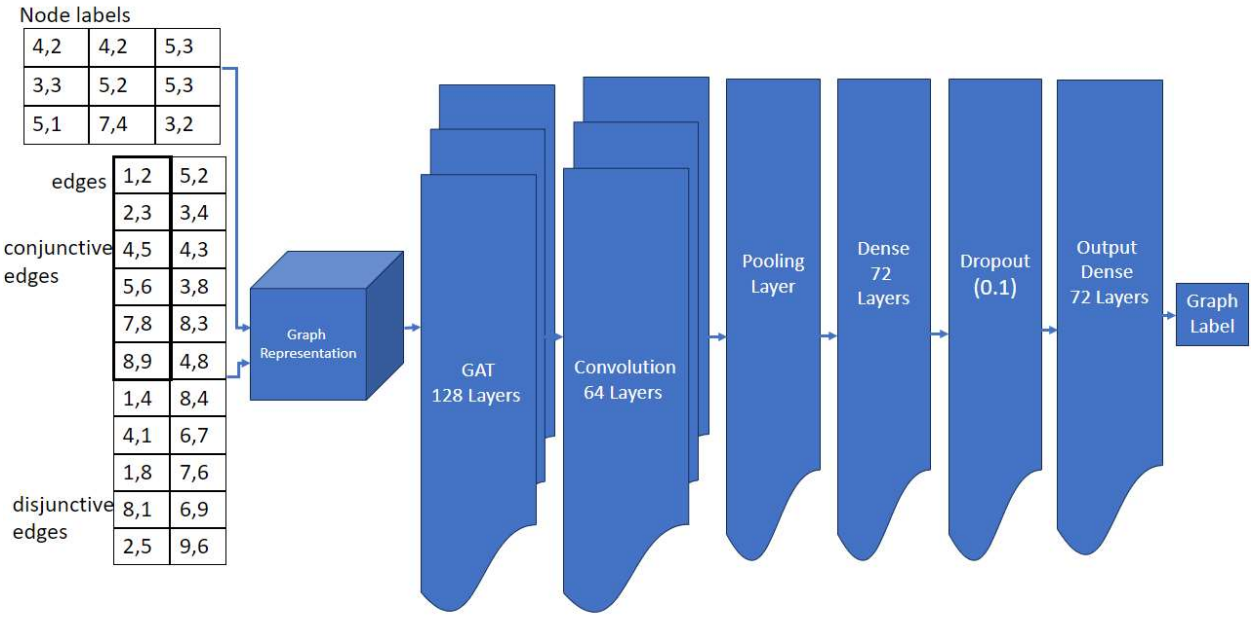


Figure 26: GNN Architecture

The GNN was constructed using TensorFlow and was framed in a Google Colab notebook to leverage an A100 Nvidia GPU. GNNs can use the architecture of GPUs to drastically improve performance, reducing training time and allowing for the training of larger neural

networks. A key inclusion that improved performance of the model was convolutional layers, both traditional and graph attention layers. As previously discussed, these layers add information from adjacent nodes to better capture network features. The theory in constructing the GNN with both layers was that the GAT layers, through the attention mechanism, highlighted the key features of the graph relevant to its specific class, while the standard convolution layers helped the model generalize. The model was trained using the Nadam optimizer with a learning rate of 0.0065, and was optimized for accuracy. Some of the key code snippets for building the model are shown below in Figure 27. The end result was a model with 52% accuracy in predicting the correct graph class. With 72 cases, this represents a significant improvement over random selection.

```
for learn in [ 0.0065]:
    with tf.device('/device:GPU:0'):
       model = create_gat_model(num_classes)
       model.compile(optimizer=Nadam(learning_rate=learn),
loss='categorical_crossentropy', metrics=['accuracy'])
       model.fit([nf, am], lbl, epochs=100)
```

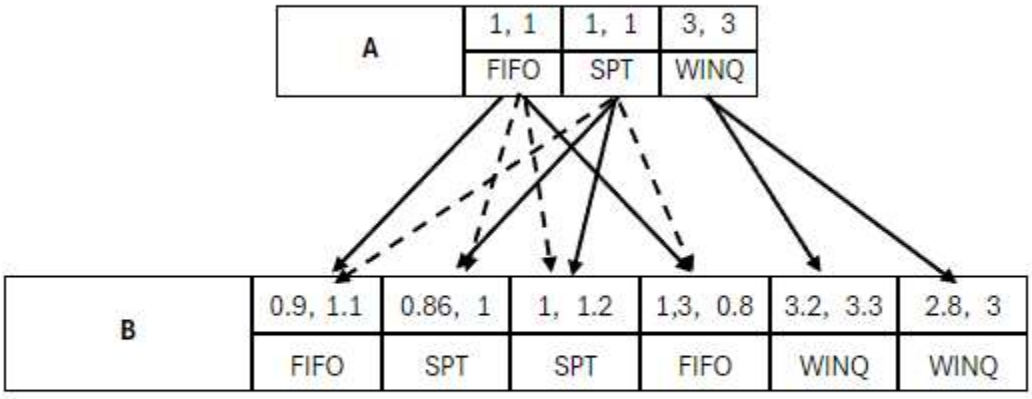
```
def create_gat_model(num_classes, dropout_rate=0.1, l1_reg=0.001, F=2,
n_attn_heads=8):
    node_features_input = Input(shape=(None, 2), name="node_features")
      # F is the number of features per node
    adjacency_input = Input(shape=(None, None), dtype='float32',name="adjacency")
    # GAT layer
    gat_1 = GATConv(
        128,
        activation='relu',
        kernel_regularizer='l1',
        attn_heads=n_attn_heads,
        concat_heads=True
    )([node_features_input, adjacency_input])
    graph_conv2 = GraphConvolution(64, activation='relu', l1_reg=l1_reg)([gat_1,
adjacency_input])
    pooling = GlobalAveragePooling1D()(graph_conv2)
    dense = Dense(num_classes, activation='relu')(pooling)
    dropout_final = Dropout(0.1)(dense)
    output_layer = Dense(num_classes, activation='sigmoid')(dropout_final)

    model = Model(inputs=[node_features_input, adjacency_input],
outputs=output_layer)
```

Figure 27: Key Code Snippets for GNN

We now discuss the third phase of the updated solution framework: testing. When a new instance is encountered, the instance is first classified into a graph class by the GNN. Then the best solutions that were identified for the 10x10x10 graph class are used to construct chromosomes in the GA's initial population. In particular, 20 of the chromosomes in the GA's initial generation came from the 20 elite chromosomes that were identified for 10x10x10 instances in the same graph class. The process of *mapping* the information in one elite chromosome for a small problem instance to create a chromosome for a larger problem

instance that is classified in the same graph class is shown in Figure 28. The top of the figure shows an elite chromosome for a small, 3-machine instance. The dispatching rule and known values of (*JobDensity, PT_Density*) for each machine are shown. The bottom of the figure shows the construction of a chromosome for a larger 6-machine instance that the GNN assigns to the same graph class. First, the values of (*Job,Density, PT_Density*) are observed for each machine. This information is shown in the top half of the chromosome. Next, the dispatching rule for a machine in the small chromosome that has similar values for (*JobDensity, PT_Density*). In other words, a 1-to-2 mapping of machines in the small chromosome is created using the values of (JobDensity, PT_Density), and the dispatching rules are copied into the large chromosome accordingly.



A- Elite chromosome for small, 3-machine problem instance
B- Initial chromosome for large, 6-machine problem instance that the GNN assigned to the same graph class

Figure 28: Shop Scaling and Mapping

In this example, there are many machines in the large problem instance which have approximate relative values of 1 for the job density and processing time density that could be mapped to either machine 1 or machine 2 of the initial problem. The bold arrows in Figure 28 show the actual mapping in this case, and the dashed arrows show other possible mappings. Details of the mapping process are shown in the "Shop-Scale" algorithm shown in Table 11.

Table 11: Shop Scaling algorithm to map smaller job shops to larger shops

Shop_Scale(*processtime, operations, graph_class)*	
1	**Let** *JobDensity, length_variablity, PT_Density* be matrices with all possibilities outlined in table 5
2	**Lookup** the *JobDensity*, *length_variability*, and *PT_Density* using the *graph_class* for the problem to be mapped
3	**Calculate** the total processing time and number of operations assigned to each machine
4	**Normalize** the processing time and operations at each machine to 1-10 scale to apply to the 10x10x10 problem which has a base of 10 operations on 10 machines with a base processing time of 10.
5	**For each** *JobDensity* and *PT_Density:*
7	**Calculate** the difference between the normalized value and each value in *JobDensity* and *PT_Density*
8	*Job_map* = the machine number that has the closest (minimum) *JobDensity* value
9	*PT_map* = the machine number that has the closest (minimum) *PT_Density* value
	Initialize a list called *mapping* to store mapped values
10	**For** each *i* in *PT_map:*
11	**For** each *j* in *PT_density:*
12	**If** *PT_map[i] = PT_density[j]*
13	**If** *job_map[i] = JobDensity[j]:* **append** *j* to *mapping[i]* **(Check If** the machine in the larger problem is mapped to the 10x10x10 instance for both *PT_Density* and *JobDensity*)
14	**For each** list in *mapping:*
15	**Set** *mapping* to a random value in the list of mapping (When there are more than one mapping, randomly select which machine to map to)
16	**Return** *mapping*

A key feature in improving performance using the shop scaling algorithm function is the randomization at step 15, which meant that there was additional variety in the initial population. This is especially visible when looking at mappings to the *base* machines (i.e., those with a base *PT_Density* and *JobDensity* of 1). This was seen previously in Figure 28, where there were numerous ways the 1,1 instances of job and processing time density can map create many possible variations to the problem. If a single mapping was used, this would reduce the number of unique instances in the initial population and limit the performance of the GA, since a more diverse initial population in the GA increases the speed of change from generation to generation.

6. Results

6.1 GA Results for the 10x10x10 problem instances

A total of 10,800 instances of size 10x10x10 – 150 for each of the 72 problem classes mentioned in Section 3.2 were created, of which ten were solved (720 total) using the GASO framework described in Section 3.5 to get a set of elite solutions. No uncertainty was considered in the simulation model, and the GA settings shown in Table 9 were used except that additional simulations were not run so the simulations (S) was set to 1. The solutions to these instances provide a useful set of dispatching rules for pre-seeding the GA when solutions to larger problem instances are considered.

To evaluate the effectiveness of scheduling techniques, comparisons are made both between methodologies and to random dispatching. Random dispatching can be considered a unique dispatching rule, where each time a scheduling decision needs to be made, a random

job is chosen as the next job on that machine. All the previously introduced rules for dispatching continue to apply, this includes only scheduling jobs available to be scheduled. Random dispatching acts as a stand in for manual scheduling done by human experts as when a shop is complex enough dispatching decisions cannot be adequately assessed and end up similar to random scheduling.

The results from these 10x10x10 instances showed positive results compared to random dispatching. The GASO procedure took an average of 33.5 seconds to compute per instance. For 91% of the instances, the best solution identified by the GASO procedure performed better than random dispatching. Also, the average objective value of the solutions found by the GASO procedure was 6% better than that for random dispatching. Depending on the instance, the GASO resulted in an objective value that was between 31% better and 12% worse than random dispatching. The best performing chromosomes for these instances are later were used to determine the dispatching rules for the pre-seeding operation of the GA after classification.

As part of the training of the GNN the 150 10x10x10, ten 20x20x20, ten 30x30x30, and ten 40x40x40 instances were used to train the model. The model was trained for 100 epochs, and ended up with an accuracy of 0.47. Accuracy is simply the number of correct decisions over the total number of predictions, so our model can correctly identify the correct class out of the 72 different classes 47% of the time based on the training data that was used. Since this is a multi-class classification problem this is a good outcome.

6.2 GNN Framework: Experimental Setup and Results

To study whether the above methodology performs as desired, a set of experiments were run. A set of instances were generated for five different shop sizes: 20x20x20m 30x30x30, 50x20x20, 200x30x20, and 40x10x20.

To easily compare results across methodologies, randomness in RMBs and variation in processing time were not considered, so each chromosome is simulated only once (S=1). For each shop size, three problem instances for each of the 72 problem classes (Section 3.2) were created using the procedure described in Section 3.2. Each instance was solved using the GASO method described in Section 3.5 with the GA settings as shown in Table 9 (except that S=1). Furthermore, the experiments considered three different *methodologies* for classifying the disjunctive graph of the underlying problem instance:

1. No classification: The standard GASO framework is applied to the instance without any graph classification in advance.
2. Perfect classification: The GASO framework is applied after the GA's first generation is pre-seeded with 20 elite solutions that are created with perfect knowledge of the instance's graph class using the Shop_Scale algorithm.
3. GNN classification: The GASO framework is applied after the GA's first generation is pre-seeded with 20 elite solutions that are created based on the trained GNN's classification of the graph class using the Shop_Scale algorithm.

The 5 problem sizes, 72 problem classes, 3 instances per class for each problem size, 100 GA generations, 100 chromosomes per generation, 1 simulation run per chromosome and 3

methods of classification give rise to a total of (5)(72)(3)(100)(100)(3) = 32,400,000 individual simulation runs. The goal of these experiments is to understand the impact of these proposed methodologies and measure whether knowledge of the graph can improve performance.

Depending on the problem's size, the number of scheduling operations for each simulation varied, which could affect run time. For a 10 job 10 machine 10 operation problem, 100 operations need to be simulated. These operations require analyzing the priority lists of the queues of jobs to be scheduled at each machine, meaning these longer queues can impact computation time. For shop layouts where there is consistently a bottleneck, the computation time can be longer than those with a more even distribution of scheduling operations. This can be seen most evidently in the 200 job 20 machine 30 operation problem instance, where computation time varied between 1,789 seconds (about 30 minutes) on the short end, and 3,503 seconds (about 58 minutes) on the long end. This computation time is nearly doubled.

As discussed, using total completion time as the objective to be minimized results in a more interesting comparison between methods. As will be shown, the results are similar between the different classification methods. This is due to the fundamental nature of the dispatching rules, which generally perform well. However, improvements are still possible, made by careful selection of dispatching rules by the GA over multiple generations.

The first problem size analyzed was the 20 jobs, 20 machines, and 20 operations problem. The results from these trials are shown numerically in Table 12 and graphically in Figure 29. The average, maximum, and minimum computation time for the 216 instances that are considered (3 for each of the 72 graph classes) from the start of GNN classification (if

performed) to the end of the 100th generation is shown in Table 13. Figure 30 sows the final schedule for the best chromosome identified by the GA without graph classification for an instance with density scenario 1 (Table 5). In this scenario, the workload is evenly balanced among the machine cells. Figure 31 shows the final result for the same instance with perfect classification of the underlying graph. Figure 31 shows the best solution identified by the GA without graph classification for an instance with density scenario 15 (Table 5). In this scenario, there are two bottlenecks machine cells having both a high concentration of jobs assigned as well as long processing times.

When looking at the results, there are a variety of cases that can occur. Looking at Case 0, a balanced shop environment, the no classification process creates a schedule as shown below in Figure 30. The difference between this result and the perfect classification solution can be seen when compared to Figure 31.

Table 12: 20x20x20 Results – fitness over 100 generations

Generation	1	10	20	30	40	50	60	70	80	90	100
No Classification	38,613	37,819	37,654	37,600	37,577	37,566	37,555	37,542	37,534	37,519	37,509
Perfect Classification	38,546	37,813	37,652	37,597	37,567	37,551	37,542	37,532	37,522	37,515	37,510
GNN Classification	38,577	37,786	37,662	37,603	37,580	37,561	37,543	37,527	37,521	37,516	37,511

Table 13: 20x20x20 Results – average computation time

	Average Computation Time (sec)	Minimum Computation Time (sec)	Maximum Computation Time (sec)
No Classification	130	104	151
Perfect Classification	139	113	251
GNN Classification	157	110	347

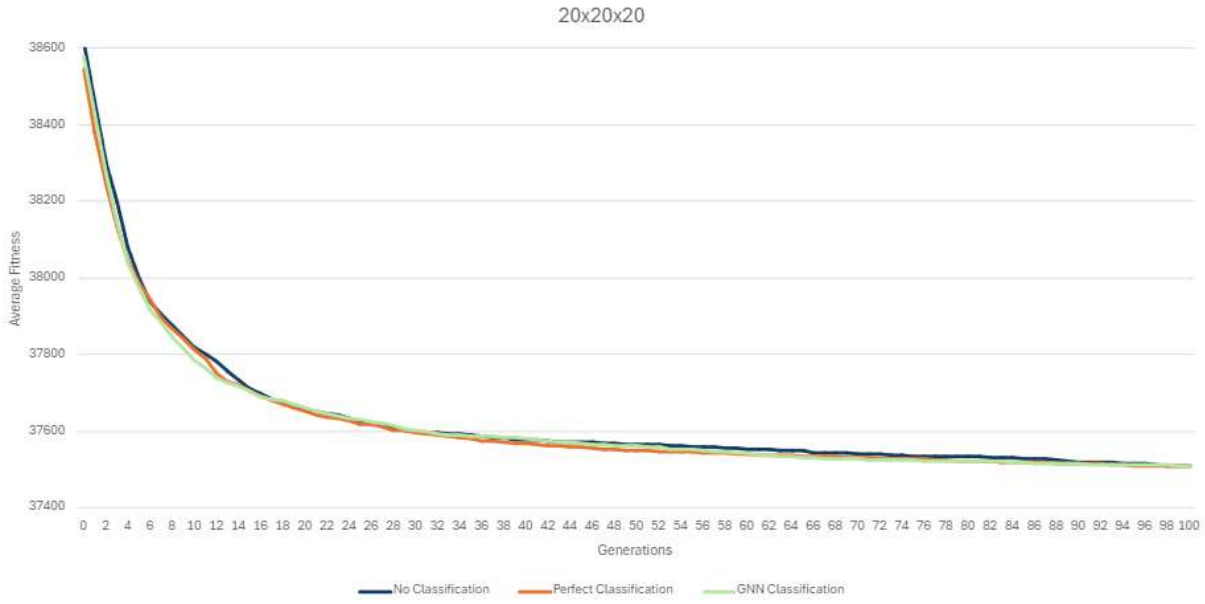


Figure 29: 20x20x20 Methodology Comparison

According to Table 12, the average fitness of a chromosome is about 38,000. Dividing this by 20 gives an average job completion time of about 1900 across all 216 instances. However, as Figures 31-32 show, the average job completion time can vary dramatically among instances with some instances much lower (Figure 31) and some much higher (Figure 32) than 1900. As shown, the perfect classification methodology performs best at the start of the problem, followed by GNN classification, then no classification. The final results after 100 generations are close, and the differences between the solutions basically disappear by the 20th generation.

When comparing the various methodologies, it is important to note that the GA improves quickly in the first few generations, however the differences between later generations can still be impactful. If the goal is to create the best schedule possible (as is normally the goal), then marginal improvements can result in 10 or more generations in computation time saved, by reducing the number of generations the GA runs.

Figure 30: 20x20x20 original methodology – Even distribution of jobs and machines

Figure 31: 20x20x20 perfect classification – Even distribution of jobs and machines

There are slight differences in the schedules shown in Figures 30-31. The perfect classification schedule in Figure 31 has a total completion time of 12,584 whereas the no classification methodology in Figure 30 produced a total completion time of 12,731. Another benchmark to compare these solutions to is random schedules, where the scheduling decisions are random. In all cases, these random schedules are beaten, but they serve as a benchmark for human decision making. When this instance is solved with random dispatching over 10 different schedules, an average total completion time of 14,450 was found.

There are not always major differences between the three classification methods, and there are instances where the different methodologies converged to the exact same solution. This can be seen in Figure 32, which has two bottleneck machine cells for every ten machine cells, each having a high concentration of jobs assigned as well as long processing times, and no variation in the number of operations on the jobs. For this instance, both the “no classification” and “perfect classification” methodologies terminated with the same solution shown in Figure 32.

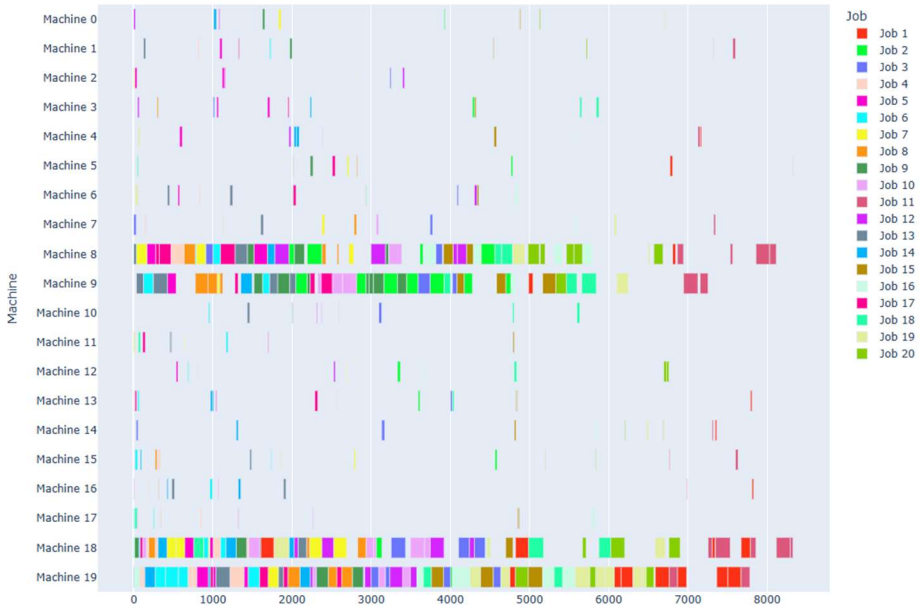


Figure 32: 20x20x20 no classification methodology – 2 bottlenecks per 10 machines with increased machine assignment and increased processing times

In this case, although the methodologies converged to a total completion time of 85,673, they significantly outperformed random dispatching which had a total completion time of 114,810.

The next problem size considered is 30x30x30. The results for these can be seen below in Table 14 and Figure 33. The average, maximum, and minimum computation time for these 216 problem instances is shown in Table 15.

Table 14: 30x30x30 Results – fitness over 100 generations

Generation	**1**	**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**
No Classification	92,740	90,870	90,466	90,289	90,172	90,098	90,028	89,980	89,953	89,922	89,881
Perfect Classification	92,748	90,828	90,435	90,238	90,122	90,043	89,996	89,957	89,927	89,899	89,879
GNN Classification	92,760	90,783	90,421	90,257	90,185	90,090	90,044	90,001	89,965	89,937	89,902

Table 15: 30x30x30 Results – computation time

	No Classification	Perfect Classification	GNN Classification
Average Computation Time	328	324	322
Minimum Computation Time	254	264	267
Maximum Computation Time	519	529	470

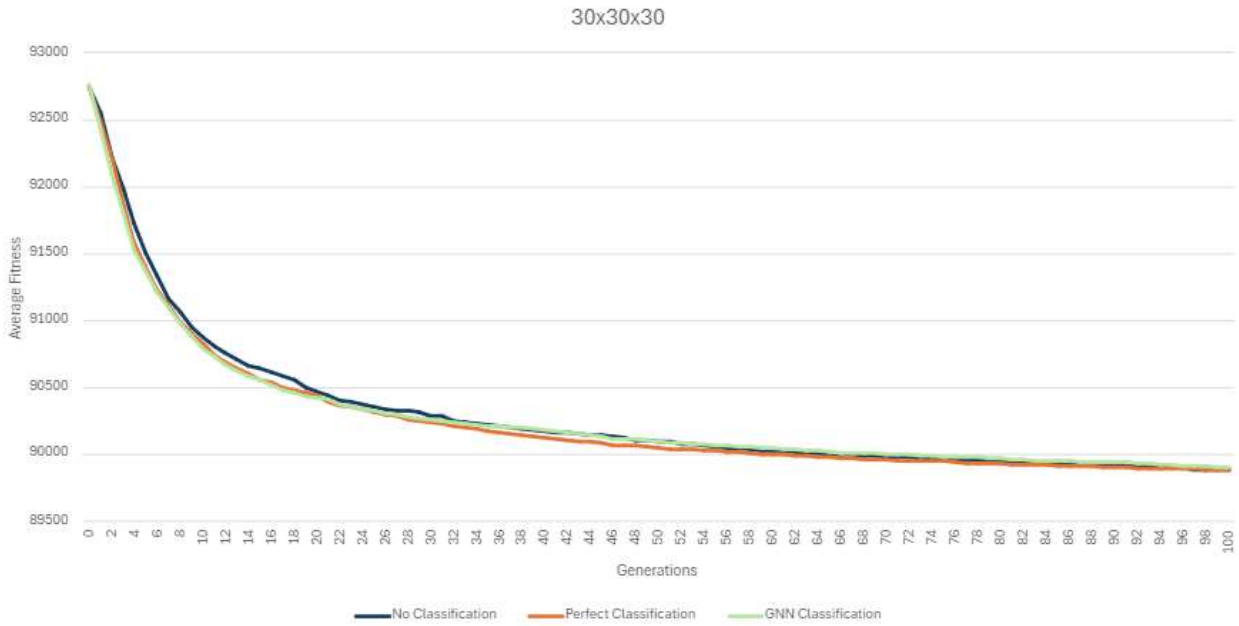


Figure 33: 30x30x30 Methodology Comparison

There are some similarities between these results and those for the 20x20x20 instances. Although the no classification starts with overall better fitness, the perfect classification values improve faster, resulting in better solutions after a few generations and maintaining a small lead after generation 10. Although the values are close, the perfect classification values are similar to those for the no classification solution with an additional 10 generations (compare generation 70 for perfect classification with generation 80 for no classification). This shows the potential value if that level of refinement is desired.

It is interesting to note that with the 30x30x30 instances there is less overall benefit from perfect classification. Total fitness is roughly equal to the no classification values, starting only marginally worse, then performing only slightly better by the end. This indicates that the initial seeding from classification still creates minor improvements for the 30x30x30 problem, but these improvements decrease over time. Another interesting note is that the GNN classification starts out worse than no classification (in generation 1). This is due to a few outliers from the initial seeding that perform poorly. After 10 generations this is no longer the case as the GA has had time to refine the results. Generally, the initial results for generation 1 are interesting, but the trends from them should not be given too much weight as the GA has not had time to refine any of the chromosomes.

One interesting comparison that can be made is between perfect classification and random dispatching for one of the instances with density scenario 2 (see Table 5). In this scenario, there is one bottleneck cell per 10 cells and the bottleneck is caused by longer processing times at the bottleneck cell. The best schedule created with perfect classification is shown in Figure 34 followed by the schedule with random dispatching in Figure 35.

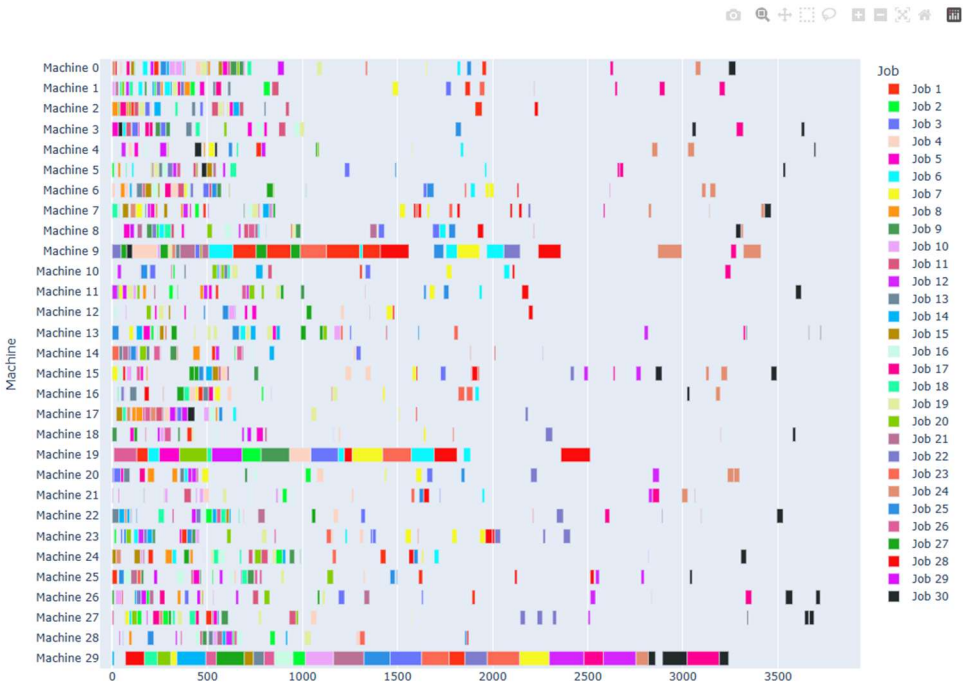


Figure 34: 30x30x30 perfect classification – 1 bottleneck cell per 10 cells based on longer processing times

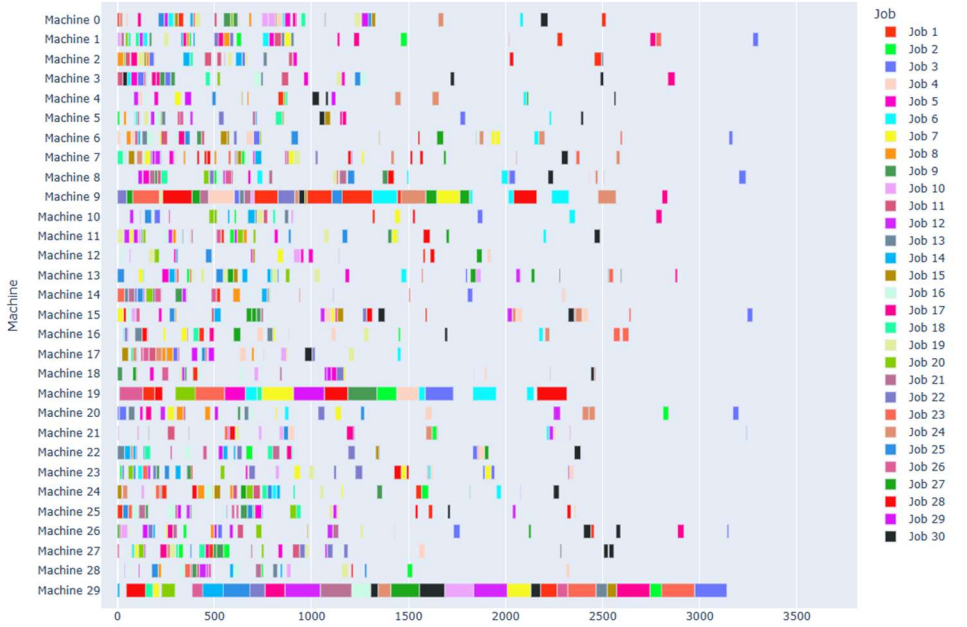


Figure 35: 30x30x30 random dispatching – 1 bottleneck cell per 10 cells based on longer processing times

At first glance it may seem that the second schedule is better. However, the schedule with perfect classification has a higher density of parts scheduled within the first quarter of the

makespan, which results in an improved fitness of 48,376 for the schedule with perfect classification versus 57,389 for the schedule with random dispatching.

The next problem set is 50x20x20: 50 jobs, 20 machines, and 20 operations per job. This configuration increases the amount of competition for each scheduling operation as the number of jobs is higher than the number of machines. The results can be seen below in Table 16 and Figure 36. The average, maximum, and minimum computation time for these problem instances is shown in Table 17.

Table 16: 50x20x20 Results – fitness over 100 generations

Generation	1	10	20	30	40	50	60	70	80	90	100
No Classification	186,097	184,055	183,570	183,348	183,223	183,144	183,088	183,049	183,015	182,980	182,955
Perfect Classification	186,078	184,004	183,584	183,384	183,246	183,155	183,112	183,074	183,053	183,018	183,001
GNN Classification	186,057	184,056	183,545	183,324	183,168	183,083	183,047	182,993	182,958	182,934	182,912

Table 17: 50x20x20 Results – computation time

	Average Computation Time	Minimum Computation Time	Maximum Computation Time
No Classification	399	315	507
Perfect Classification	409	319	717
GNN Classification	459	317	605

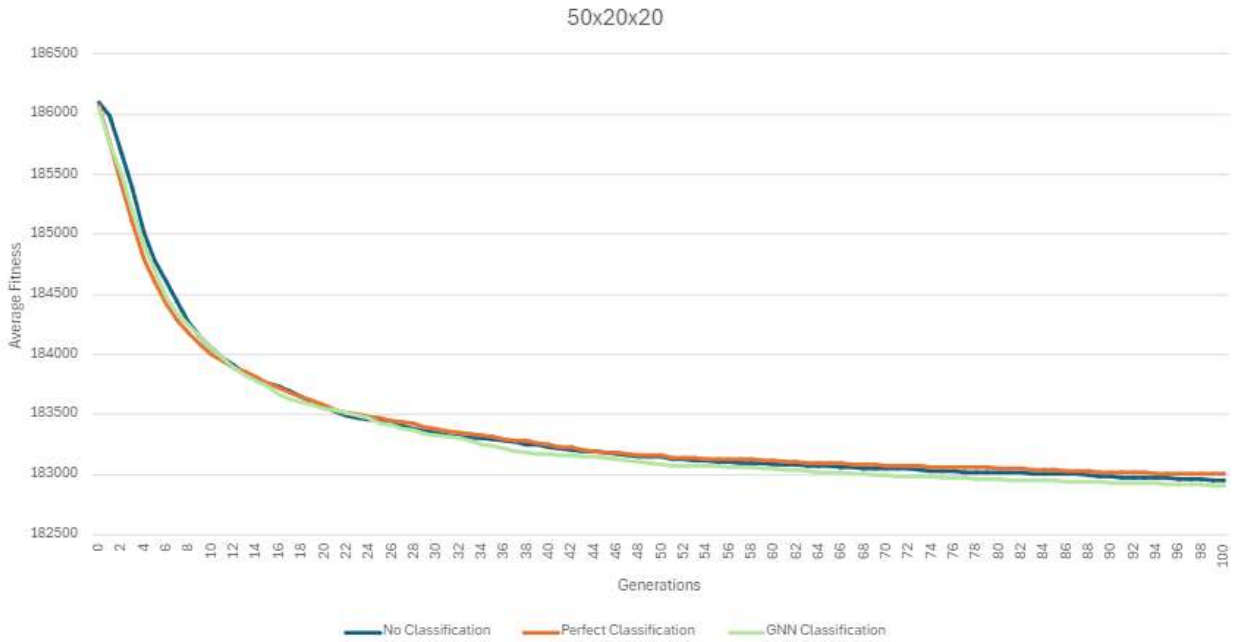


Figure 36: 50x20x20 Methodology Comparison

For these instances, classifying with the GNN results in the fittest average solutions at generation 1. Perfect classification improves the most, performing best after 10 generations. This pattern is then inverted at the 20th generation with the GNN classification performing the best of any methodology. The problem ends with GNN classification performing best, reaching similar solutions after 80 generations to what no classification reached at 100, while perfect classification performed the worst of any methodology. This performance difference might be explained by the differences in the 50x20x20 shop and the 10x10x10 shop and how the mapping algorithm (Shop_Scale Table 11) scaling works. In particular, the mapping from a 10x10x10 instance to a 50x20x20 instance that is properly classified may not be beneficial owing to the extra bottlenecks introduced in a 20-machine instance versus a 10-machine instance and how those bottlenecks are loaded due to the higher ratio of jobs to machines (50 jobs to 20 machines versus 10 jobs to 10 machines). The GNN may be able to identify a better

overall match than this perfect classification, which in this case results in a solution of comparable quality to no classification.

An example of this can be seen with instance 16. This instance has two bottleneck cells for every ten cells where half of the bottleneck cells have roughly five times as many jobs assigned to them and the other half of the bottleneck cells experience processing times that are roughly five time longer. This instance was classified by the GNN as having three bottleneck cells for very ten cells, one due to the number of operations and two due to higher processing times. In other words, the GNN thought there was an additional bottleneck cell due to processing times for every ten cells. When comparing the performance between perfect classification, where the GA was pre-seeded with density scenario 7 chromosomes (see Table 5), to GNN classification, with density scenario 9 chromosomes, the GNN classification resulted in a total completion time that was more than 1,000 lower (around 1% better). This is not a major difference, but these minor differences can result in meaningful savings for a large manufacturer.

The next problem size analyzed is 200x30x20L 200 jobs, 30 machines, 20 operations per job. This represents a real-world sized shop where the number of machines is varied but limited, and there are many different types of jobs in the system at any given time. These results are displayed in Table 18 and Figure 37. The average, maximum, and minimum computation time for these problem instances are shown in Table 19. Notice the huge range of computation time due to the queues that form at various machines, resulting in much longer computation times to create schedules for many instances.

Table 18: 200x30x20 Results – fitness over 100 generations

Generation	1	10	20	30	40	50	60	70	80	90	100
No Classification	1,698,049	1,691,940	1,689,026	1,687,860	1,687,235	1,686,935	1,686,706	1,686,564	1,686,448	1,686,287	1,686,183
Perfect Classification	1,697,840	1,691,640	1,688,864	1,687,845	1,687,399	1,687,063	1,686,771	1,686,528	1,686,468	1,686,334	1,686,238
GNN Classification	1,697,926	1,691,190	1,688,664	1,687,573	1,687,078	1,686,802	1,686,568	1,686,383	1,686,224	1,686,078	1,686,038

Table 19: 200x30x20 Results – computation time

	Average Computation Time	Minimum Computation Time	Maximum Computation Time
No Classification	2350	363	3503
Perfect Classification	2364	363	5342
GNN Classification	2332	363	6430

Figure 37: 200x20x20 Methodology Comparison

The results show a similar trend as the 50x20x20 shop. Perfect classification performs well at first, but performs the worst of any methodology by generation 100. GNN classification

consistently outperforms the no classification methodology. The largest disparity between the GNN classification and the no classification methodologies can be seen in the instances with density scenario 21 (Table 5). In this scenario, job density is ramped among the machines and average processing time is evenly distributed. The results for one such instance with no classification and with GNN classification are shown below in Figure 38 and Figure 39.

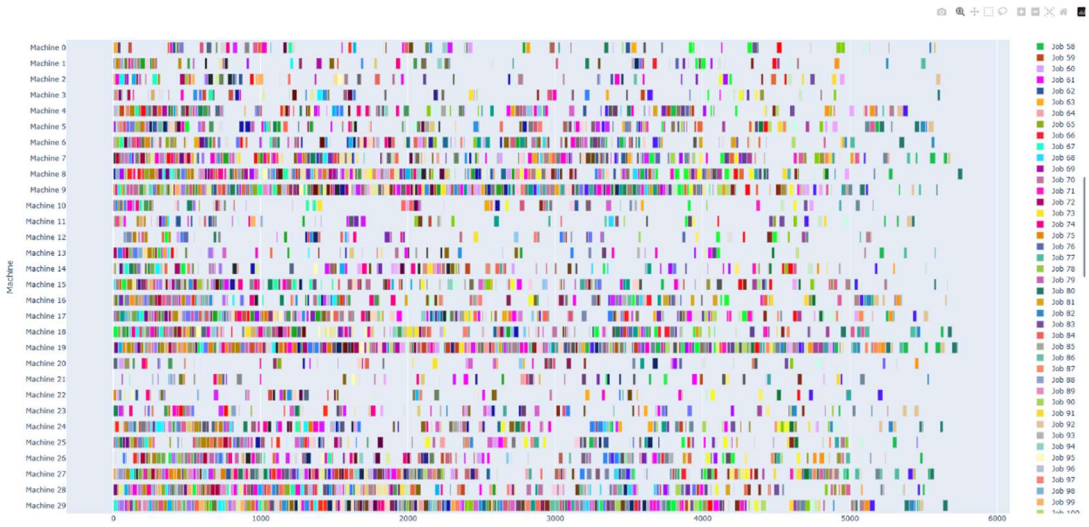


Figure 38: 200x30x20 no classification – ramped job density and even processing time density

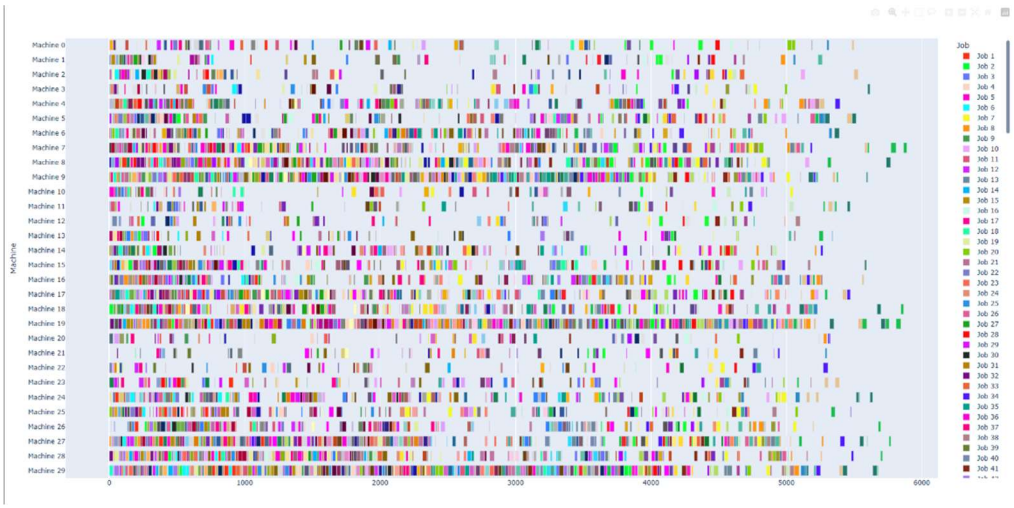


Figure 39: 200x30x20 GNN classification – ramped job density and even processing time density

The schedule created without classification (Figure 35) has a fitness of 513,776, while the schedule created using GNN classification has a fitness of 512,937. With such marginal

differences and the large problem size, the results remain consistent. It is difficult to visually determine the differences in performance. However, GNN classification performs marginally better than no classification.

The final problem size analyzed was 40x10x20, a 40 jobs, 10 machines, 20 operations per job. The goal of considering these 10-machines instances is to understand whether performance improves when mapping from 10x10x10 instances to other instances with 10 machines compared to instances with more than 10 machines. The results for these instances can be seen below in Table 20 and Figure 40. The average, maximum, and minimum computation time for these problem instances is shown in Table 21.

Table 20: 40x10x20 Results – fitness over 100 generations

Generation	1	10	20	30	40	50	60	70	80	90	100
No Classification	213,745	212,730	212,489	212,408	212,352	212,310	212,286	212,265	212,245	212,227	212,214
Perfect Classification	213,795	212,703	212,468	212,369	212,326	212,300	212,268	212,242	212,229	212,215	212,204
GNN Classification	213,795	212,721	212,490	212,403	212,359	212,327	212,297	212,264	212,249	212,238	212,235

Table 21: 40x10x20 Results – computation time

	Average Computation Time	Minimum Computation Time	Maximum Computation Time
No Classification	332	228	760
Perfect Classification	319	245	424
GNN Classification	315	233	549

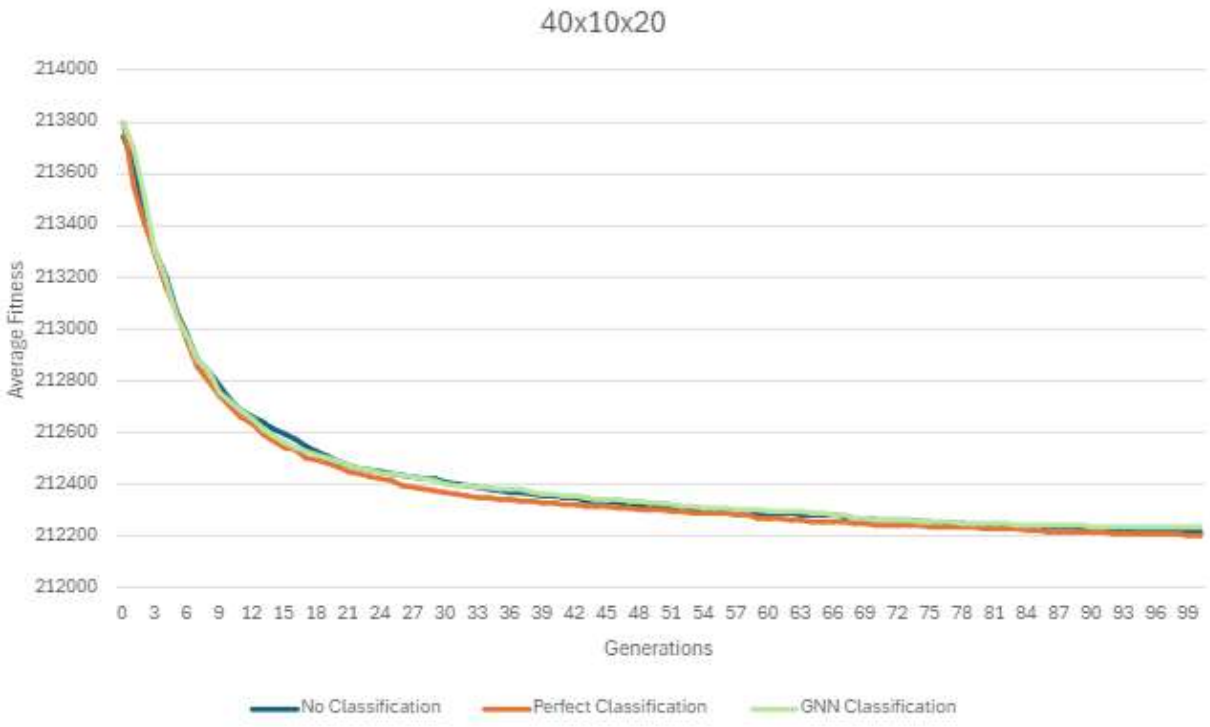


Figure 40: 40x10x20 Methodology Comparison

The results for this problem show the perfect classification methodology provides the best schedule at generation 10, followed by no classification, with GNN classification performing the worst. This shows a reversal of the rankings for the 200x30x20 instances. Thus, the difficulties in translating solutions for smaller problem instances into solutions for larger instances exists for all five problem sizes that are considered, including one with only 10 machines. Overall, perfect classification generally performed better than no classification, for the 20x20x20, 30x30x30, and 40x10x20 instances but worse than no classification for the 50x20x20 and 200x30x20 instances. It is interesting to note that GNN classification performs better for the 50x20x20 and 200x20x20 problems, indicating that graph classification is beneficial, but the best mapping may be to a different graph class than that used to generate the instance.

In the above experiments, solutions were compared by removing variability, which allowed for direct comparisons. Based on the results from these trials, graph classification is marginally beneficial. This is the case in the 20x20x20 and 30x30x30 shops where having perfect knowledge of the graph class and using this knowledge to pre-seed the GA leads to improved solution quality. This is also the case the larger instances where perfectly classifying instances did not improve performance, but GNN classification did improve performance. These results might be explained as follows. First, the job shop problem instance classification system may start to deteriorate as larger instances are considered. The 200x30x20 shop has more jobs available to be scheduled at any given time, resulting in longer queues at each machine cell. Therefore, the types of relevant dispatching rules may be different than what they would be for a 10x10x10 shop.

The likely cause of this is that as instances become larger, their graphs continue to diverge from the training set. This appears to be especially true as additional machines are added. The difficulties involved in graph classification might be addressed through a more nuanced approach. In a real-world shop, the job profiles will almost certainly be different than the ones proposed here, and they will be unique to the specific shop. By using historic schedules, classification might be done with more specific graph classes that are unique to that job shop, and then the model can be trained on these classifications. Overall, tailoring the graph classes may eliminate the phenomenon seen here in which performance for the largest instances is better when they are classified into a graph class that is different than the one used to create them.

6.3 Introducing Uncertainty: Experimental Setup and Results

As previously discussed, uncertainty in the form of variation in processing time and RMBs is an important item to address to accurately represent real-world shops. Now that a framework has been identified, how well it holds up to the introduction of random elements must be investigated. In this section, all experiments are run using GNN classification to identify the graph class and then the GA to find the best solution for the given instance.

Overall, we consider the four uncertainty (i.e. randomness) scenarios outlined in Figure 41. In the first scenario, no uncertainty is considered. In the second scenario, only variation in processing times is applied. In the third scenario, only RMBs are considered. In the final scenario, both forms of randomness are applied.

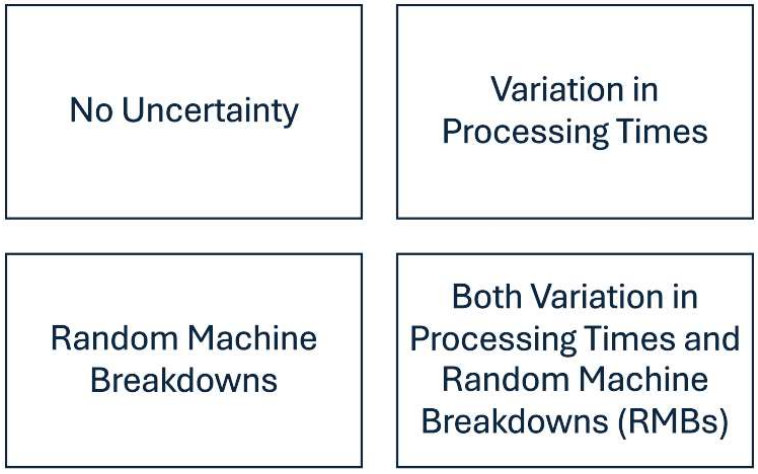


Figure 41: Uncertainty Scenarios

In these experiments, we assume the following.

- There are 5 machines per machine cell.
- There is a 5% probability that one machine in a particular cell will break down at any timestep. (This becomes 0% if all machines in the cell have broken down)

- The repair time for an individual machine ranges from 4 to 6 timesteps based on a uniform distribution.
- For every operation, actual processing time varies uniformly between 20% below and 20% above the expected processing time t_{ik}.

The amount of variability for the relevant cases was set to have a marginal impact so one variation in the schedule did not majorly impact the results.

The experiments consider three different variations of a shop using scenario 1 (see Table 5) this scenario has an even distribution in both job density and processing time density. These shops were created using the process previously outlined, which does impart some variability into the machine assignment and the processing times, however, these instances should be generally evenly loaded. All instances have size 20x20x20. To assess the effectiveness of the GNN+GA solution framework, the results are compared against random dispatching for five simulation replications.

Importantly, every chromosome in every generation of the GA is evaluated by conducting five simulation replications (S=5), and the average fitness across these replications was used to determine the fitness for that chromosome. This increased the number of simulations required by a factor of 5, while also increasing the computation time due to the need to calculate the RMBs, as previously discussed. The results of these experiments are shown in Table 22.

Table 22: 20x20x20 Uncertainty Study

Problem	Average Total Completion Time for GA + GNN	Average Total Completion Time for Random Dispatching	Ratio of GNN + GA Methodology to Random Trials
No Uncertainty – Instance 1	12672	13737	0.9225
No Uncertainty - Instance 2	32644	36842	0.8861
No Uncertainty - Instance 3	26814	28093	0.9545
Variation in Processing Time - Instance 1	13110	13286	0.9868
Variation in Processing Time - Instance 2	33426	35639	0.9379
Variation in Processing Time - Instance 3	27222	26514	1.0267
RMB - Instance 1	12614	13497	0.9346
RMB - Instance 2	31612	33357	0.9477
RMB - Instance 3	25809	26098	0.9889
RMB and Processing Time - Instance 1	12745	13554	0.9403
RMB and Processing Time - 2	32024	35414	0.9043
RMB and Processing Time - 3	25476	26168	0.9736

As seen in the study, there is some variability in the results. The key metric is the ratio between the average objective value using the GNN + GA and that using random dispatching. The low ratios show the utility of the GA method. The average ratio is 0.92 when there is no uncertainty. This can be compared to a ratio of 0.98 when there is variability in processing times, the 0.96 when there are RMBs, and 0.94 with both types of uncertainty.

The goal of these trials is to quantify the impacts of randomness on scheduling and identify the impact the proposed GNN + GA methodology has on the results compared to random scheduling. The exact numbers from these trials could be further refined with additional trials. However, these results are sufficient to determine some trends. As seen in the results, the benefits of using the GNN + GA is smallest when there is randomness in processing

times only. This benefit increases with the inclusion of RMBs. However, the benefits are greatest when there is no randomness followed by both types of randomness. This shows that when both forms of randomness, from variation in processing times and RMBs, the GNN + GA methodology provides a tangible benefit in improving the quality of the machine dispatching rules.

An important aspect to consider with RMBs is that when a machine breaks down the drop in efficiency can create a local bottleneck. This bottleneck requires more precision in scheduling to find a useful set of dispatching rules. On the other hand, that variation in processing time can go in both directions, reducing the time a job takes as well as increasing it. This helps explain why the performance difference between the GNN + GA and random dispatching is small when only processing times are variable and is larger when there are RMBs.

To gain insights in the effects of uncertainty on system performance, we now consider two schedules created by the GNN + GA methodology for the first problem instance. Figure 42 shows the schedule with no uncertainty; Figure 43 shows one of the five simulated realizations of the shop schedule when both types of uncertainty are present.

Figure 42: Base case – No uncertainty

Figure 43: Base case – RMB and variable processing times

In the figures, the impacts are subtle but they show that variability can impact the overall results slightly, causing different dispatching rules to be selected. The impacts of variability are seen in the delays experienced on the less loaded machines, including delays in machine cells 7 and 18. As seen in the results, as randomness is introduced the impacts of proper dispatching rules remain important. It should be noted that the dispatching rules used in

Figure 42-43 are optimized for total completion time, so even though the “no randomness” schedule has a higher total makespan it has a lower total completion time than the scenario with both types of randomness (12,672 vs 12,745). This is expected since the introduction of randomness will impact the schedule slightly. Overall, these experiments show that the most improvement provided by the proposed GNN + GA methodology is when no randomness is present, and the second most improvement is when there are both types of randomness.

6.4 Scaling for Large Job Shops: Experimental Setup and Results

As the goal of this work focuses on addressing real-world problems, the methodology must scale to create schedules for large job shops. To do this, a set of four very large problem instances with more than 10,000 total operations were examined. The goals of this section are to test whether large-scale job shops can be scheduled within a reasonable timeframe, how quickly the GA converges, and whether the results provide utility in a real setting. Our experiments consider both variability in processing times and RMB’s and utilize the graph classification as previously outlined. The GA settings match those in Table 10 (with S= 5).

The instances we consider, and the experimental results are shown in Table 23. Note that the GA + GNN generally converged in less than the standard 100 generations.

Table 23: Large Job Shop Problem Study

Problem Size	GA Total Completion Time (TCT)	Total Random Dispatching Completion Time	GA Ratio to Random Dispatching	Ratio of Final TCT to Start TCT	Ratio of Final TCT to gen. 10 TCT	Ratio of Final TCT to gen. 20 TCT	Ratio of Final TCT to gen. 50 TCT	Ratio of Final TCT to gen. 80 TCT	Total CPU Time (hrs)
100x100x100	64,755	67,762	0.956	0.954	0.983	0.990	0.999	1.000	2.01
150x150x150	97,447	104,806	0.930	0.970	0.980	0.989	0.995	1.000	3.07
200x200x200	131,624	139,361	0.944	0.974	0.981	0.982	0.995	1.000	4.66
300x300x300	200,703	211,733	0.948	0.979	0.999	0.999	0.999	1.000	10.02

As seen, the GA + GNN outperformed random dispatching for all four problem instances. The problems all ran within reasonable timeframes using a personal computer. The final 300x300x300 problem took slightly over 8 hours to run, which would allow the schedule to be recreated within the timeframe of a standard work shift. The output for the 300x300x300 problem can be seen in Figure 44.

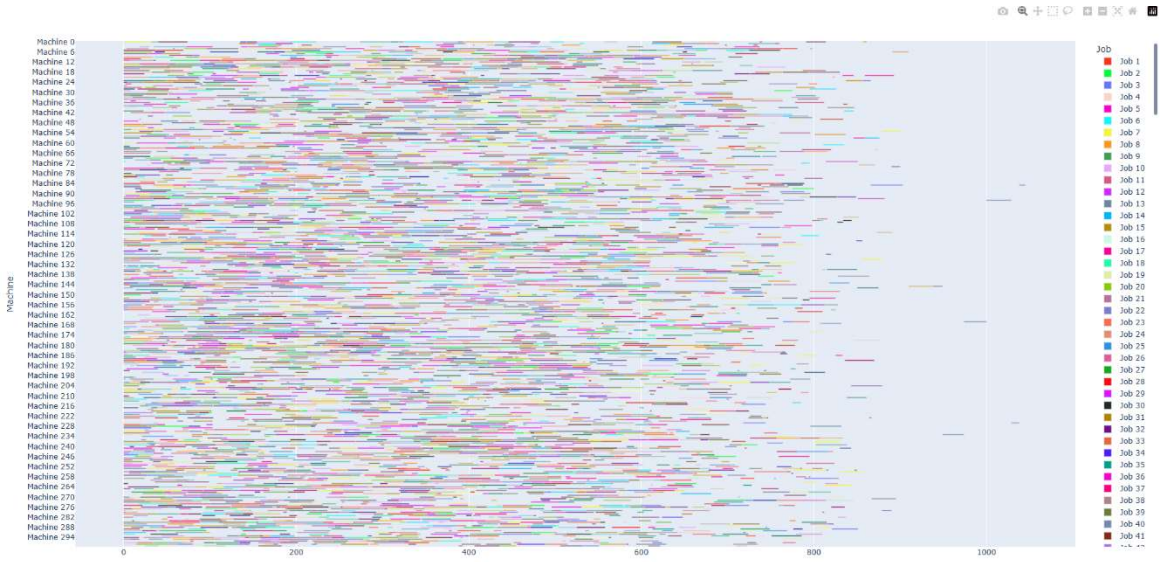


Figure 44: 300x300x300 problem – Base case instance RMB and variable processing times

The GA converges quickly, reaching solutions within 20 generations that are only 1-2% worse than the final solution. This indicates that 100 GA generations may be more than is

needed and 20 generations may be enough for most cases. This would cut the total computation time by a factor of 5 making the solution scale exceptionally well with large problems.

6.5 Scaling for Large Job Shops: Discussion of Results

As seen in the results, the GNN GA methodology scales well. The solution can be run for up to 300x300x300 problems. If we look at processing time, the problem scales from 6.9 schedules simulated per second for the 100x100x100 problem (100 chromosomes, 5 variations per chromosome, 100 generations for 50,000 simulations in 2.01 hours) to 1.38 schedules simulated per second for the 300x300x300 problem. This uses overall time to include all the overhead of the GA. These values have an obvious logarithmic relationship, so if we use a log transformation then use a linear regression to approximate this trend, we get the following Equation 11, which can be seen visually in Figure 45. As can be seen, the operations simulated per second decrease with the problem size, which is expected as the problem gets larger, the number of schedules that can be completed per second will decrease as there are more operations to schedule per schedule.

$$\boldsymbol{time} = \mathbf{6301} \cdot (\boldsymbol{problem\ size})^{-\mathbf{0.731}} \quad (10)$$

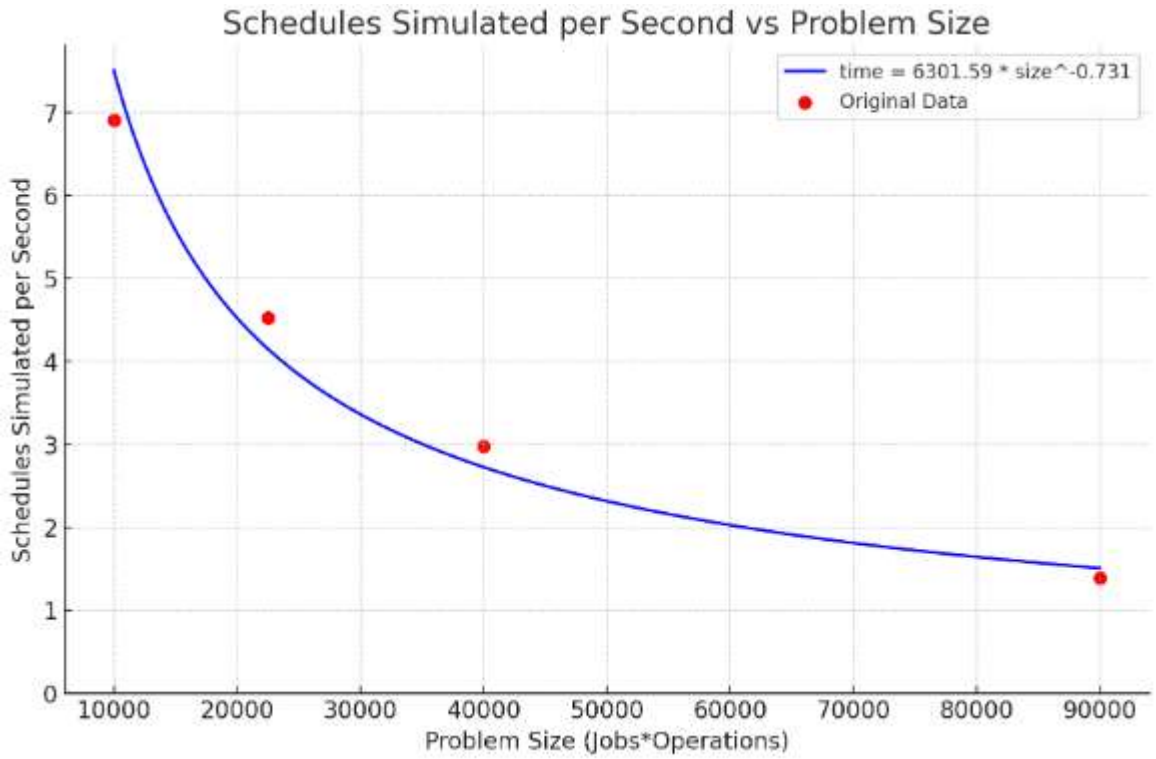


Figure 45: Relationship of problem size to time per simulation

Using this equation, we approximate that a 1000x1000x1000 instance could complete 0.25 schedule simulations per second (i.e, 4 seconds per simulation) and would therefore take 53 hours to complete 100 generations. If we allowed the process to stop after 20 generations, this would take 10 hours to complete and could be run overnight. One of the benefits of the GA is that it is easy to parallelize. Each of the chromosomes in a generation could be ran in parallel since the chromosome is known upfront. In theory, each simulation replication for that chromosome could also be run in parallel. This could lead to a 500 fold increase in computational speed if the computational resources were available. If all these improvements were implemented, not assuming any speed-up due to faster CPUs, the 1000x1000x1000 instance could be completed in 1.2minutes. Even if this was parallelized at the chromosome level, this would result in a 6-minute computation time. Therefore, if we kept our constraint of

being able to run an instance overnight and added a 4-hour buffer to give it 12 hours to run, our theoretical ceiling is $5.78*10^9$ operations to simulate per simulation run or a 76,000x76,000x76,000 shop, assuming a sufficiently strong computing environment is available.

Overall, the method presented in this work shows how to improve job shop schedules in real facilities in a way that can be easily scaled to even the largest shops, run in a reasonable timeframe, and provide measurable improvements over manual schedules and random dispatching rules.

7. Discussion

7.1 Implementation of Framework: Priorities

This solution framework ties in well with Lean manufacturing. It allows for the involvement of shop floor management, since the use of dispatching rules can be easily understood, and their implementation can be easily handled. This allows shop floor management to focus on improving their environment and streamlining their own operation, while allowing easy flow management through the shop. This approach takes inspiration from value stream mapping as the proposed process tries to leverage a classification technique similar to how value stream mapping would, to improve flow. It avoids the rigidity that value stream mapping often relies on in practice and instead uses technology to keep the system dynamic.

As a manager of a job shop, there are many competing priorities that must be considered when determining how to implement a scheduling regime. These problems are not

as simple as identifying an optimal way of scheduling and forcing the shop to run those jobs at the exact time specified by the schedule states. Key stakeholders' interests also need to be balanced. These are primarily the needs of the business, often represented by sales and logistics departments, and the needs of operations departments, primarily represented by shop floor management but also by engineering and quality functions.

Business needs require the shop to deliver to constantly changing requirements. Even if a shop is impeccably managed, it's customer's will not be. Therefore, in response to the natural bullwhip that occurs with improper planning, requested delivery dates will always be in flux. This can be mitigated through communication, but if the ability to adjust schedules is provided to the business, it represents a lever that can help win more business in the future. Therefore, there will always be shifting requirements for delivery. On top of this, new business usually comes with unforeseen issues during development that need to be addressed in production, such as the introduction of high priority jobs that need to be rushed.

Operations departments have different constraints. Depending on how the shop floor is managed, both an area manager and the operator on the machine need to know what part is running next. In practice, if a machine is sitting idle and there are parts available to start running, there may be a push to start processing what is available. Regardless of scheduling, depending on how well operations are managed and how easily communication flows, machine-level decisions will always need to be made at the machine. Therefore, creating simpler sets of rules for generating schedules is one way to ensure the machine-level decisions are made as desired by the business. Engineering and quality can also affect operations through last-minute changes to processes, unexpected delays, or forcing the operation to wait so that

the operation can be witnessed at a certain time. These issues can often be managed; however, this can cause extra variation in processing time as delays are added at the last minute.

The GNN GA framework introduced in this dissertation addresses some of these issues naturally. First, the dispatching rules are easy to implement and manage at the machine level. In practice, each area supervisor will need training on the meaning of the rules and will be given enough information to make these decisions. Ideally, this would be implemented through a job board, preferably an electronic board, that collects real-time information on downstream backlogs, arrival times, processing times, etc. When provided with a dispatching rule, the dashboard could then populate an order to be processed by the machine cell, which could then be easily followed. Training is still important here, as it is essential management understands how decisions are being made, so they can explain them to machine operators. This type of system integrates naturally with Industry 4.0 where real-time data on operations can be made both to the shop floor and to the business allowing real-time updates to schedules and delivery.

The process also works well with Lean manufacturing principles. By keeping scheduling rules simple, the control of the workspace is entrusted to operations, allowing empowerment of the shop floor. Through continuous improvement and Lean departments, improvements at the shop floor level can be made, reducing processing times and improving machine uptime. Investigations of value streams may be useful, but depending on the job mix, they may not be necessary. Value streams, when implemented, often require either dedicated machinery or a dedicated share of machinery. This can work in low-mix shops. However, in a high-mix job shop, more flexibility is needed. The proposed GNN GA methodology from this work helps address the scheduling regularity that value streams bring. Their study though could be very useful. By

identifying value streams, similar jobs can be grouped, and these value streams can be mapped to a shop environment to help with classification. To ensure there is adequate training data for the GNN, these value streams might be utilized to synthesize additional data.

It should also be noted that the proposed methodology can help inform Lean manufacturing decisions. Part of Lean implementation involves an understanding of machine breakdowns and processing time variability. However, knowing these numbers by themselves does not necessarily translate into better managerial decisions. On the other hand, the proposed methodology allows the impacts of variability on different machines to be measured to make sure resources are devoted to areas with the highest payback. There may be a low-utilized area that is highly variable with many machine breakdowns. This may not matter due to the low utilization, or it may be hugely impactful because these variations may negatively impact downstream operations. By using the proposed methodology to study these impacts over time, engineering, Lean and continuous improvement efforts can focus on the areas that matter most to the shop environment.

7.2 Implementation of Framework: Guidelines

The preceding section discussed general priorities to be addressed. However, there are a few concrete steps that need to be in place to utilize the proposed methodology. The first is to ensure there are enough computing resources available in an environment for both the instance classification and simulation. Classification requires access to GPUs, which can be achieved through many cloud providers; Google Colab was used for this work. The CPU environment needed for the simulation optimization depends on the maximum size of the job

shop. If the shop size stays below roughly 100,000 total operations, then a high-end but normal personal computer can be used. If the job shop is larger, parallelization will be needed. The number of cores will be based on the number of operations. This can be calculated as previously outlined in Section 6.5 based on the needs to reschedule.

Once a computing environment is setup, the objective function needs to be decided. The objective function used in this research was total completion time. As discussed, total completion time creates a useful metric to compare schedules. However, real-world scheduling is often based on a combination of priority level and due date. One option for handling this is to use a weighted due-date function to calculate fitness when determining the best set of dispatching rules for a given job shop instance. Generally, giving certain jobs higher priority should be avoided. However, this functionality should be added to the framework. High priority jobs can be given a weighting function that the different dispatching rules can use when calculating priority order. Again, management should avoid using these except for extreme cases, as relying only on priority will create highly non-optimal schedules from a machine utilization and due date perspective.

The next modification is to collect data for the simulation of the operations. Data concerning the randomness for both machine breakdowns and variable processing times should be collected as accurately as possible, either through direct studies or analysis of historical data. The more accurately these variabilities are represented, the more accurate the simulation model will be. Also, information for each machine cell must be contained along with the number of individual machines at that cell. Furthermore, the facility must be able to generate a standard report from the Enterprise Resource Planning (ERP) system to retrieve a list of current

jobs in the system and their routings, along with the number of pieces and their processing times. A computer program must then be written to convert this into an organized list of jobs and operations to be scheduled. Finally, the simulation must account for delayed starts so in-process machines continue working on their current operations at time 0. This information must therefore also be included in the ERP report.

The GNN will need to be set up. For this, a set of historical problem instances must be generated. This can be done through a lookback. As discussed, value streams can help create classes of shops, but ultimately a variety of methods can be used. These include scientific methods such as conducting various unsupervised machine learning approaches to group the disjunctive graphs of historic problem instances via clustering based on job routings and product mix, or through manual labeling of data. Once a method is designed, it is unlikely that sufficient data will be in place, so additional synthetic problem instances may need to be created. All those instances should be scheduled utilizing the GASO framework to create a set of elite solutions for pre-seeding the GA. Once these are in place, the GNN can be trained to classify instances for future production use.

The system can then be used for scheduling. At the start of each week, the schedule should be re-optimized, allowing for more generations than the minimum. If the minimum is set for 20 generations, then at least 40 should be used. This is important, because the schedules created during these runs will be saved for further refinement. If a large change happens during the week, such as a huge downtime event, a new high-priority job, or a large change in priorities, the GASO framework can be re-run overnight using the minimum number of generations. In either case, the schedule created is not important; what is important is the

dispatching rules identified for each machine. These dispatching rules are communicated to the shop floor so they know how they are running for the week. Limiting re-optimizing to a weekly activity keeps the changes at the shop floor level minimal, allowing for easier implementation. As technology is integrated and real-time data is updated, the scheduling window can be adjusted. However, during implementation it is recommended to keep optimization runs to a minimum as buy-in from the shop floor is a key driver in whether the system will be adopted.

Once the system is live, a database should be created to keep track of the different problem instances encountered and the schedules created, and it should be continually updated over time. Each weekly schedule should be saved along with the graph class the GNN assigned it to. In addition, the set of elite solutions used for pre-seeding the GA and the disjunctive graph of the problem instance should be stored for future training. With weekly updates, retraining the GNN is a lower priority, but is recommended yearly. It should also be reevaluated when anything occurs that changes the overall form of the job shop. This could be the purchasing of new equipment where machines are added to create new machine cells or winning a large new order or set of orders that changes the dynamics of the shop. In these cases, time should be spent creating new synthetic schedules and determining if new classes of problem instances should be added to the training data. Additional studies should be done using methods similar to those presented in this research to measure the impacts of classification. With limited training data, it is possible that classification with the GNN does not provide value, so it should be ignored until more training data is collected. As shown, classification with a GNN could improve the schedules and should still be considered.

At this point, the system should be functional and create value for the facility. As with any process, monitoring is important. The predicted schedules should be compared to those achieved to measure effectiveness.

7.3 Looking to the Future

A key piece of this research involves making sure the proposed methodology can be implemented in a real-world shop. This means that the processes must be easy to understand and resilient to change. Dispatching rules meet both these requirements. As manufacturing continues to automate, real-time data will be more readily available. In a shop with real-time parts tracking, equipment monitoring, and electronic dashboards for shop floor management and/or automated selection of the next part, the requirements for the system change. In this case, explainability is less essential, and the level of robustness is lower since the system can be monitored, and the process can be automatically re-optimized if there are significant changes. Theoretically, it could also be in a constant state of re-optimization, where every time a schedule is created, the latest data is considered and GASO process starts again. With each iteration automatically flowing information to the shop floor. If communication sequencing is electronic, the priority list can also be electronic. In this case, the dispatching rule can become more complicated. This continues to add complexity as parts often need to be staged requiring a way to lock them in place to ensure that when a new priority list comes out, it does not affect the next part to be scheduled and changes the downstream operations. This information also needs to flow back to the scheduling system so it can accurately represent the shop floor when conducting simulation experiments.

In this case a more complex set of dispatching rules can be used and alternatives should be explored. The research shows how many different custom dispatching rules provide useful results utilizing ant-colony optimization or other pheromone-based systems. These are often treated uniformly, creating a master dispatching rule for the entire shop. However, keeping with the theme of this research, specific dispatching rules at each machine allow for additional flexibility. These custom dispatching rules could be determined as a pre-step to the GNN+GA framework, then including them as additional options for each chromosome to select for a given machine. These more complex dispatching rules will be more difficult for shop floor management to inherently understand, and so should not be considered as the system is first introduced. However, if the GASO framework is well adopted, and it is integrated with real-time monitoring and automatic job boards, custom dispatching rules could be adopted since the order of jobs to run could be displayed without the need to explain why the order was created.

The system should not be rigid as what works well in one shop may not work well in all shops. Through proper management, these systems can evolve over time and be optimized for the unique environment of each shop that is encountered.

There are a few other areas for possible improvements. One of which is improving how re-optimization or re-scheduling works. When there are major issues and a new schedule needs to be created, the proposal in this work is to re-run the schedule and have the shop update all dispatching rules to accommodate. This can be disruptive to the shop. To minimize these disruptions the schedule change can be focused on the impacted area keeping dispatching rules constant throughout the rest of the shop. This can be added as a constraint where the GASO only optimizes the impacted machine cell, and any highly dependent machine cells. This

reduction in the problem size would also allow for quick re-optimizations. Additional flexibility could be added to break jobs apart, this can be especially important during a breakdown. By allowing jobs to start processing on the next operation once a certain percentage of the job is complete, performance of the schedules can be improved by reducing potential downtime. These rules would need to be defined based on expected variability in the process to ensure that the downstream operation is not interrupted. This would require modifications to the simulation to make jobs available earlier in the downstream queues.

8. Conclusion

The goal of this research was to develop a methodology that can be used to schedule a real-world job shop. This meant the scheduling decisions need to be easily explainable to shop floor management, that the proposed methodology can handle large job shops, and that it can create schedules for these large job shops in a reasonable timeframe. The use of a genetic algorithm (GA) to select dispatching rules through simulation optimization proved to meet these requirements. Using a graph convolutional neural-network, a problem instance can be classified to allow a GA to be pre-seeded with elite dispatching rules for the machines, allowing for improvements to the solution. This methodology proved to be scalable to a job shop with up to 90,000 total operations, with theoretical scalability up to 5,780,000,000 total operations (76000x76000x760000) if additional computing power and parallelization is available. The proposed methodology beat a random schedule by 5% for job shops with significant variability and uncertainty in their operations, so it is robust. The proposed methodology is also designed

to be implemented into a real-world shop and easily explained. Overall, this research creates an intersection between theoretical job shop scheduling and the needs of actual job shops.

Work Cited

Abdolrazzagh-Nezhad, M. & S. Abdullah. (2017). “Job Shop Scheduling: Classification, Constraints and Objective Functions.” World Academy of Science, Engineering and Technology, International Journal of Computer and Information Engineering 11: 429-434.

Aggarwal, C. C. (2017). “Neural Networks and Deep Learning: A Textbook.” Springer. ISBN: 978331994647

Ali, K. B., A. J. Telmoudi, & S. Gattoufi. (2020). "Improved Genetic Algorithm Approach Based on New Virtual Crossover Operators for Dynamic Job Shop Scheduling." in IEEE Access, vol. 8, pp. 213318-213329, 2020, doi: 10.1109/ACCESS.2020.3040345.

Bergamaschi, D., R. Cigolini, M. Perona, & A. Portioli. (1997). “Order review and release strategies in a job shop environment: A review and a classification.” International Journal of Production Research, 35, 399-420. doi: 10.1080/002075497195821

Brucker, P., S.A. Kravchenko & Y.N. Sotskov. (1997). On the complexity of two machine job-shop scheduling with regular ob jective functions, OR Spektrum 19 (1997), 5-10. doi: 10.1007/BF01539799

Caron, M., P. Bojanowski, A. Joulin, & M. Douze. (2018). “Deep clustering for unsupervised learning of visual features.” In ECCV, 132–149. doi: 10.48550/arXiv.1807.05520

Chen, B., C.N. Potts, and G. J. Woeginger. (1998). A review of machine scheduling: Complexity, algorithms and approximability. Handbook of Combinatorial Optimization, 3:21–169. doi: 10.1007/978-1-4613-0303-9_25

Czerwinski, S.E. (1997). “Exploring the Job-Shop Search Space with Genetic Algorithms.” MIT September 1997, Master Thesis.

Dai, Q., Z. Liu, Z. Wang, X. Duan, & M. Guo. (2022). “Graph CDA: a hybrid graph representation learning framework based on GCN and GAT for predicting disease-associated circRNAs.” Brief Bioinform. 2022 Sep 20;23(5):bbac379. doi: 10.1093/bib/bbac379. PMID: 36070619.

Ding, S., X. Xu, H, Zhu, J. Wang, & F. Jin. (2011). “Studies on optimization algorithms for some artificial neural networks based on genetic algorithm (GA).” J. Comput., 6(5), 939-946. doi: 10.4304/jcp.6.5.939-946

Eiben, A. & J. Smith. (2003). “Introduction to Evolutionary Computing. Springer.” ISBN: 3-540-40184-9

Fan, H. -L., H. -G. Xiong, G. -Z. Jiang & G. -F. Li. (2015). "Survey of the selection and evaluation for dispatching rules in dynamic job shop scheduling problem." 2015 Chinese Automation Congress (CAC), 2015, pp. 1926-1931, doi: 10.1109/CAC.2015.7382819.

Franke J., T. Stockheim, & W. Koenig. (2004). “The impact of reputation on supply chains: An analysis of permanent and discounted reputation,” To appear in Journal of Information Systems and e-Business Management. doi: 10.1007/s10257-005-0007-4

Garey, M. R., D. S. Johnson, & R. Sethi. (1976).” The Complexity of Flowshop and Jobshop Scheduling.” Mathematics of Operations Research, 1(2), 117–129. http://www.jstor.org/stable/3689278

Gholami, M. & M. Zandieh. (2009). “Integrating simulation and genetic algorithm to schedule a dynamic flexible job shop.” J Intell Manuf 20, 481–498. doi: 10.1007/s10845-008-0150-0

Gohareh, M. M., & E. Mansouri. (2022). “A simulation-optimization framework for generating dynamic dispatching rules for stochastic job shop with earliness and tardiness penalties.” Computers & Operations Research, Volume 140, 1-14, doi: 10.1016/j.cor.2021.105650

Golenko-Ginzburg, D. & A. Gonik. (2002). “Optimal job-shop scheduling with random operations and cost objectives.” International Journal of Production Economics, Volume 76, Issue 2, 2002, Pages 147-157, ISSN 0925-5273, doi: 10.1016/S0925-5273(01)00140-2.

Gopakumar, B., Sundaram, S., Wang, S., Koli, S., & Srihari, K. (2008). “A simulation based approach for dock allocation in a food distribution center.” 2008 Winter Simulation Conference, pp. 2750-2755. doi: 10.1109/WSC.2008.4736393

Gracanin, D., L. Bojan, B. Ivan, L. Danijela, & B. Borut. (2013). “Cost-Time Profile Simulation for Job Shop Scheduling Decisions.” International Journal of Simulation Modelling. 12. 213-224. 10.2507/IJSIMM12(4)1.237. doi: 10.2507/IJSIMM12(4)1.237

Grady, O.P. & K. H. Lee. (1988). “An intelligent cell control system for automated manufacturing.” International Journal of Production Research, 1988,26: pp. 845-861. doi: 10.1080/00207548808947906

Gupta, S., & Jain, A. (2021). “Analysis of Integrated Preventive Maintenance and Machine Failure in Stochastic Flexible Job Shop Scheduling with Sequence-dependent Setup Time.” Smart Science, 10, 175 - 197. doi: 10.1080/23080477.2021.1992823

Hildebrandt, T., D. Goswami & M. Freitag. (2014). "Large-scale simulation-based optimization of semiconductor dispatching rules." Proceedings of the Winter Simulation Conference 2014, 2014, pp. 2580-2590, doi: 10.1109/WSC.2014.7020102.

Hui, B., Zhu, P., & Hu, Q. (2020). “Collaborative Graph Convolutional Networks: Unsupervised Learning Meets Semi-Supervised Learning.” AAAI Conference on Artificial Intelligence. doi: 10.1609/aaai.v34i04.5843

Irani, S.A. (2020). “Job Shop Lean: An Industrial Engineering Approach to Implementing Lean in High-Mix Low-Volume Production Systems (1st ed.).” Productivity Press. https://doi.org/10.4324/9781003034186

Ishii, N., & J. J. Talavage. (1991). “A transient-based real-time scheduling algorithm in FMS.” International Journal of Production Research, 29, pp. 2501-2520. doi: doi.org/10.1080/00207549108948099

Jin, Y. & J. Branke. (2005). "Evolutionary optimization in uncertain environments-a survey." in IEEE Transactions on Evolutionary Computation, vol. 9, no. 3, pp. 303-317, June 2005, doi: 10.1109/TEVC.2005.846356.

Kopp, D., M. Hassoun, A. Kalir, & L. Mönch. (2020). “SMT2020—A semiconductor manufacturing testbed”. IEEE Transactions on Semiconductor Manufacturing, 33(4), pp. 522-531. Doi: /10.1109/TSM.2020.3001933

Krafcik, J. F. “Triumph of the Lean Production System.” (1988) Sloan Management Review vol 30, 1 International Journal of Advanced Manufacturing Technology. Vol 53: pages 799-809. doi: 10.1007/s00170-010-2860-7

Li, L., Q. Yu, K. Lin, Y. Ma, & F. Qiao. (2023). “Scheduling of Semiconductor Manufacturing System. In: Data-Driven Scheduling of Semiconductor Manufacturing Systems”. Advanced and Intelligent Manufacturing in China. Springer, Singapore. https://doi.org/10.1007/978-981-19-7588-2_1

Li, L., & Z. Jiang. (2007). “Self-adaptive dynamic scheduling of virtual production systems.” International Journal of Production Research, 45(9), 1937–1951. doi: 10.1080/00207540600791640

Li, Q. & D. Qu (2009). “Simulation and Optimization of Discrete Customized Job-Shop Scheduling.” In Proceedings of the 2009 Second International Conference on Intelligent Computation Technology and Automation - Volume 02 (ICICTA '09). IEEE Computer Society, USA, pp. 213–215. doi: 10.1109/ICICTA.2009.289

Lian, Y., & H.V. Landeghem. (2002). “An application of simulation and value stream mapping in lean manufacturing.”

Liang, X., Y. Huang & M. Huang. (2020). "Prediction of Optimal Rescheduling Mode of Flexible Job Shop Under the Arrival of a New Job." 2020 IEEE 8th International Conference on Computer Science and Network Technology (ICCSNT), 2020, pp. 55-58, doi: 10.1109/ICCSNT50940.2020.9304997.

Mastrolilli, M. & O. Svensson. (2011). “Hardness of Approximating Flow and Job Shop Scheduling Problems.” J. ACM 58, 5, Article 20 (October 2011), 32 pages. doi: 10.1145/2027216.202721

McDonald, T., E.M. Van Aken, & A.F. Rentes. (2002). “Utilizing Simulation to Enhance Value Stream Mapping: A Manufacturing Case Application.” International Journal of Logistics Research and Applications, 5, pp. 213 - 232.

Nouri, H.E., O. Belkahla Driss, & K. Ghédira. (2016). “A Classification Schema for the Job Shop Scheduling Problem with Transportation Resources: State-of-the-Art Review.” Computer Science On-line Conference. doi: 10.1007/978-3-319-33625-1_1

Ohno, T. (1988). “Toyota Production System.” New York, NY: Productivity Press. pp. XV. ISBN 0-915299-14-3.

Oman, S. & P. Cunningham. (2001). “Using Case Retrieval to Seed Genetic Algorithms. International Journal of Computational Intelligence and Applications.” 1. 71-82. doi:10.1142/S1469026801000056.

O’Shea, K. & R. Nash (2015). “An Introduction to Convolutional Neural Networks.” doi: 10.48550/arXiv.1511.08458

Panwalkar, S. S., & W. Iskander. (1977). A Survey of Scheduling Rules. Operations Research, 25(1), 45–61. http://www.jstor.org/stable/169546

Park, J., J. Chun, S. Kim, Y. Kim, & J. Park. (2021). “Learning to schedule job-shop problems: representation and policy learning using graph neural network and reinforcement learning.” International Journal of Production Research, 59, pp. 3360 - 3377. Doi: 10.1080/00207543.2020.1870013

Park, J. Y. Mei, S. Nguyen, G. Chen, & M. Zhang, (2018). ‘‘An investigation of ensemble combination schemes for genetic programming based hyperheuristic approaches to dynamic job shop scheduling.’’ Appl. Soft Comput., vol. 63, pp. 72–86, Feb. 2018, doi: 10.1016/j.asoc.2017.11.020.

Pekarcikova, Miriam, P. Trebuna, M. Kliment, S. Kral, & M. Dic. (2021). “Modelling and Simulation the Value Stream Mapping – Case Study.” Management and Production Engineering Review: Vol 12 Iss 2. doi: 10.24425/mper.2021.137683

Pierreval, H. (1992). “Expert system for selecting priority rules in flexible manufacturing systems.” Expert Systems with Applications, 1992, 5: pp. 51 -57.

Pierreval, H. & NN Mebarki. (1997). “A real-time scheduling approach based on a dynamic selection of dispatching rules.” International Journal of Production Research, 1997,35(6): 1 pp. 575- 1 591.

Priore, P, D. de la Fuente D, A. Gomez, & J. Puente. (2001). “A review of machine learning in dynamic scheduling of flexible manufacturing systems.” AI EDAM, 2001 ,1 5(03):251-263

Rolf, B., T. Reggelin, A. Nahhas, M. Müller & S. Lang. (2020). "Scheduling Jobs in a Two-Stage Hybrid Flow Shop with a Simulation-Based Genetic Algorithm and Standard Dispatching Rules." 2020 Winter Simulation Conference (WSC), 2020, pp. 1584-1595, doi: 10.1109/WSC48552.2020.9384067.

Rother, M. & Shook, J. (1998). “Learning to See: Value Stream Mapping to Create Value and Eliminate Muda.” Lean Enterp. Inst. Brookline. Lean Enterprise Institute, Cambridge.

Sajadi, S. M., A. Alizadeh, M. Zandieh, & F. Tavan. (2019). “Robust and stable flexible job shop scheduling with random machine breakdowns: multi-objectives genetic algorithm approach.” International Journal of Mathematics in Operational Research Vol. 14 No. 2. doi: 10.1504/IJMOR.2019.097759

Schwenke, C., H. Blankenstein & K. Kabitzsch. (2018). "Large-Scale Scheduling with Routing, Batching and Release Dates for Wafer Fabs using Tabu Search." 2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA), 2018, pp. 472-479, doi: 10.1109/ETFA.2018.8502606.

Sharma, P. & A. Jain. (2015). “Stochastic Dynamic Job Shop Scheduling with Sequence-Dependent Setup Times: Simulation Experimentation.” International Journal of Engineering and Technology. 5. Pp. 19-25. doi: 10.4103/0976-8580.149475.

Sharma, V. & N. Virmani, (2020). "Development of lean production system using value stream mapping approach: A case study." International Journal of Productivity and Quality Management, Inderscience Enterprises Ltd, vol. 30(2), pp. 168-185.

Siemens Digital Industries Software. (2022). “Optimizing and managing the complexity of your PCB production schedule.” [White paper]. https://static.sw.cdn.siemens.com/siemens-disw-assets/public/45njDBvo27cduFneZwJrMo/en-US/Optimizing-and-managing-the-complexity-of-your-PCB-production-schedule_tcm27-105743.pdf

Simons D. & D. Taylor (2006). “Lean thinking in the UK red meat industry: a systems and contingency approach.” Int J Prod Econ 106: pp. 70–81. doi: 10.1016/j.ijpe.2006.04.003.

Singh, B., S.K. Garg, & S.K. Sharma. (2011). “Value stream mapping: literature review and implications for Indian industry.” Int J Adv Manuf Technol 53, 799–809 doi: 10.1007/s00170-010-2860-7

Stockheim T., O. Wendt, & M. Schwind, (2005). "A Trust-based Negotiation Mechanism for Decentralized Economic Scheduling." Proceedings of the 38th Annual Hawaii International Conference on System Sciences, Big Island, HI, USA, pp. 35a-35a. doi: 10.1109/HICSS.2005.59.

Su, C., C. Zhang, D. Xia, B. Han, C. Wang, G. Chen, & L. Xie. (2023). “Evolution strategies-based optimized graph reinforcement learning for solving dynamic job shop scheduling problem,” Applied Soft Computing, Volume 145, 110596, ISSN 1568-4946, doi: 10.1016/j.asoc.2023.110596.

Teppan, E. C. (2018). "Dispatching Rules Revisited-A Large Scale Job Shop Scheduling Experiment." 2018 IEEE Symposium Series on Computational Intelligence (SSCI), 2018, pp. 561-568, doi: 10.1109/SSCI.2018.8628827.

Valamede, L. S., A. C. Santos Akkari. (2020). "Lean 4.0: A New Holistic Approach for the Integration of Lean Manufacturing Tools and Digital Technologies." International Journal of Mechanical Engineering and Materials Sciences, 5(5), pp. 851-868. Retrieved from https://www.ijmems.in/volumes/volume5/number5/66-IJMEMS-20-73-5-5-851-868-2020.pdf

Vázquez-Rodríguez, J.A., S. Petrovic. (2010). "A new dispatching rule based genetic algorithm for the multi-objective job shop problem." J Heuristics 16, pp. 771–793 (2010). doi: 10.1007/s10732-009-9120-8

Veličković, P., G. Cucurull, A. Casanova, A. Romero, P. Lio, & Y. Bengio (2017). "Graph Attention Networks". ICLR 2018. https://arxiv.org/pdf/1710.10903v3.pdf doi: 10.48550/arXiv.1710.10903

Walsh, P. & P. Fenton. (2004). "A high-throughput computing environment for job shop scheduling (JSP) genetic algorithms." 1554 - 1560 Vol.2. doi:10.1109/CEC.2004.1331081.

Wang, B. & Q. Li. (2007). "Rolling Horizon Procedure for Large-scale Job-shop Scheduling Problems." 2007 IEEE International Conference on Automation and Logistics, 2007, pp. 829-834, doi: 10.1109/ICAL.2007.4338679.

Wang, Y. (2021). "Flexible Job Shop Scheduling Rules Mining Based on Random Forest." 2021 2nd International Conference on Computing and Data Science (CDS), 2021, pp. 220-226, doi: 10.1109/CDS52072.2021.00045.

Wang, Z. & M. Gombolay, (2020) "Learning Scheduling Policies for Multi-Robot Coordination With Graph Attention Networks," in IEEE Robotics and Automation Letters, vol. 5, no. 3, pp. 4509-4516, July 2020, doi: 10.1109/LRA.2020.3002198.

Wu, Z., S.Sun, & S. Xiao. (2018). "Risk measure of job shop scheduling with random machine breakdowns." Computers & Operations Research, Volume 99, 2018, pp. 1-12, ISSN 0305-0548, doi: 10.1016/j.cor.2018.05.022.

Xu, B., L. Tao, X. Deng & W. Li. (2021). "An Evolved Dispatching Rule Based Scheduling Approach for Solving DJSS Problem." 2021 40th Chinese Control Conference (CCC), 2021, pp. 6524-6531, doi: 10.23919/CCC52363.2021.9549754.

Yan, Y. & G. Wang. (2007). "A job shop scheduling approach based on simulation optimization." 2007 IEEE International Conference on Industrial Engineering and Engineering Management, 2007, pp. 1816-1822, doi: 10.1109/IEEM.2007.4419506.

Ye, L. & Y., Chen. (2010). "A Genetic Algorithm for Job-Shop Scheduling." Journal of Software. 5. 10.4304/jsw.5.3. pp. 269-274.

Yu, H., T. Tweed, M. Al-Hussein, & R. Nasseri. (2009). "Development of Lean Model for House Construction Using Value Stream Mapping." Journal of Construction Engineering and Management-asce, 135, pp. 782-790. doi: 10.1061/(ASCE)0733-9364(2009)135:8(78

Zhai, Y. & D. Zhaoyang, C. Wei & L. Changjun. (2013). “Decomposition-based scheduling algorithm for large-scale job shop.” 2013 IEEE International Conference on Information and Automation, ICIA 2013. 124-127. doi: 10.1109/ICInfA.2013.6720282.

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.” ArXiv, doi: 10.48550/arXiv.2010.12367

Zhang, M., Z. Cui, M. Neumann, & Y. Chen. (2018). “An End-to-End Deep Learning Architecture for Graph.” Classification. AAAI Conference on Artificial Intelligence. doi: 10.1609/aaai.v32i1.11782

Zhang, S., H. Tong, J. Xu, & R. Maciejewski. (2019). “Graph Convolutional Networks: A comprehensive review.” Computational Social Networks, 6(1). doi: 10.1186/s40649-019-0069-y

Zhao, F., Y. Hong, D. Yu, & Y. Yang. (2005). “A Hybrid Approach Based on Artificial Neural Network and Genetic Algorithm for Job-shop Scheduling Problem.” Proceedings of 2005 International Conference on Neural Networks and Brain Proceedings, ICNNB'05. 3. 1687 - 1692. doi:10.1109/ICNNB.2005.1614954.

Zhao, F., J. Zhang, C. Zhang, & J. Wang. (2015). ‘‘An improved shuffled complex evolution algorithm with sequence mapping mechanism for job shop scheduling problems." Expert Syst. Appl., vol. 42, no. 8, pp. 3953–3966, May 2015. doi: 10.1016/j.eswa.2015.01.007

Zhou J., C. Ganqu, H. Shengding, Z. Zhengyan, Y. Cheng, L. Zhiyuan, W. Lifeng, L. Changcheng, S. Maosong. (2020). "Graph neural networks: A review of methods and applications" AI Open 1 57-81. doi: 10.1016/j.aiopen.2021.01.001

Zhou, Y.,J. -J. Yang & L. -Y. Zheng. (2019). "Hyper-Heuristic Coevolution of Machine Assignment and Job Sequencing Rules for Multi-Objective Dynamic Flexible Job Shop Scheduling." in IEEE Access, vol. 7, pp. 68-88, 2019, doi: 10.1109/ACCESS.2018.2883802.