Suitability of Genetic Algorithms for solving Flexible Job Shop Problems

Marko Ivanov
Supervisor(s): Kim van den Houten, Mathijs de Weerdt
EEMCS, Delft University of Technology, The Netherlands

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Abstract

The aim of this research paper is to present two genetic algorithms targeted at solving the Flexible Job Shop Problem (FJSP). The first one only tackles a single objective - the schedule makespan, while the second one takes into account multiple objectives for the problem. Each schedule is represented by two integer vectors - one for the machine assignments and one for the operation sequence. Special care is taken to only produce valid schedules when generating the starting population and applying the mutation and crossover operations for further populations. A Mixed Integer Linear Programming (MILP) solution to the FJSP is presented and used as a benchmark for the feasibility of the genetic algorithms. The algorithms are tested on a set of 13 provided problem instances. The results showcase that genetic algorithms outperform the MILP implementation for large problem instances and produce solutions much faster.

1 Introduction

In a DSM production line there are multiple enzymes with different recipes which need to be produced. There is an order book of enzymes which need to be produced, along with a soft deadline for each one. Each recipe is an ordered series of operations along with their durations, which need to be done in order to produce the enzyme. Furthermore, for each operation, there are multiple machines in the facility which can execute it. If a machine is used for the production of one enzyme, it has to be cleaned before an operation with another enzyme begins.

There is a vital question which arises here - what is the best way to schedule the production of the enzymes given these constraints? This can be modelled as a special form of the Flexible Job Shop Problem. The jobs in this case are the completion of enzyme recipes. The special part is the existence of cleaning times between consecutive uses of the same machine, which need to be taken into account in our models.

The Flexible Job Shop Problem is NP-hard [1]. Exploring the entire search space is infeasible even for problem instances with very few machines and jobs. On top of that, there are multiple ways to measure the fitness of a schedule - the makespan of the schedule, idle time of the machines, total lateness, etc.

There are a plethora of ways this problem has been addressed in scientific literature. A literature review is presented in section two. We focus on two approaches for solving FJSP - Mixed Integer Linear Programming and Genetic Algorithms. We have been provided with an implementation of a MILP solver, as well as a set of 13 problem instances.

The aim of this research paper is to investigate whether a Genetic Algorithm can produce feasible solutions to the given Flexible Job Shop problem instances for the DSM plant. A Genetic Algorithm was chosen in order to be able to address the problem of having more than a single objective. A Genetic Algorithm can produce a set of solutions, instead of a single one. This means we can have solutions which perform very well on some objectives but poorly on others. This allows the scheduling of the production plant to be more flexible based on different conditions.

In section 3 two genetic algorithms are presented. In section 4 an evaluation of the results is made in comparison to the results produced by the provided MILP implementation on the same problem instances. Multiple configurations of the genetic algorithm parameters are taken into account and their performance is measured. In the next sections, some notes on responsible research and a discussion of the process are presented. In section 7 the concluding remarks and future work are presented.

2 Literature review

In this section a brief introduction of the Flexible Job Shop Problem is given, as well as a review of the literature on using Mixed Integer Linear Programming and Genetic Algorithms to solve it.

2.1 Introduction to Flexible Job Shop Problems

In the classical version of the Job-shop Scheduling Problem (JSP), there are n jobs which need to be completed. Each job has a set of operations which need to be completed in order to finish the job. Each operation of each job has a specific machine on which it needs to be executed. The extended version of this problem - the Flexible Job Shop Problem (FJSP) - each operation can be performed by a subset of the machines, rather than only one. Research into FJSP began in the early 90s by Brucker and Schlie [2], where they gave a solution to the problem for 2 jobs in polynomial time. It has been proven that the general JSP is NP-hard [1]. FJSP, being an extension and a more difficult version of the classical JSP, is also NP-hard.

There are many approaches to solving the FJSP which use different algorithms and combinations between them. A comprehensive overview of the literature is presented in [3]:

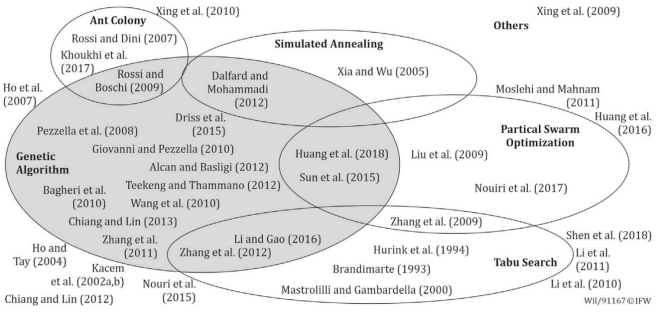

Figure 1: Overview of literature on FJSP
Source: [3]

There are a few main groups present in the literature - Tabu Search, Ant Colony Optimisation, Simulated Annealing, Partical Swarm Optimisation, Mixed Integer Linear Programming (MILP) and Genetic algorithms. Combinations between

different approaches are also used to try to achieve better results. In this paper we will be focusing on MILP and Genetic algorithm approaches.

2.2 MILP approaches for FJSP

Mixed Integer Linear Programming is a technique which provides a mathematical model for the problem. The model is a set of linear constraints of variables and the aim is to find a solution by optimising the value of an objective function. In most MILP formulations which aim to tackle the FJSP, the objective function is to minimise the makespan of the schedule. The first time an Integer Linear Programming model was applied to a scheduling problem was in 1959 by Harvey M. Wagner [4]. In more recent literature, multiple different mathematical formulations have been used for the FJSP. In [5] five different mathematical formulations are examined.

In this section we first give some mathematical notations for the FJSP which are later used in the paper. They are also the basis of what a MILP formulation of the problem looks like.

Definitions for the FJSP

In this particular scenario, we have a set of jobs J, which need to be completed on a set of available machines M. Each job represents the production of one of the enzymes from set E. Each job i has an associated sequence of operations O_i, where O_{ij} is the j-th operation of job i. Each operation has an associated set of machines $M_{i,j} \subseteq M$, which can perform the j-th operation of job i. Furthermore, there is a changeover time between two consecutive operations on the same machine based on the enzymes in the two operations. $CO_{me_1e_2}$ is the changeover time on machine m where the first product is e_1 and the next one is e_2. Let S_{ijk} be the starting time of j-th operation of job i on machine k and C_{ijk} be the completion time of j-th operation of job i on machine k.

There are two important precedence constraints:

1. The j-th operation of job i needs to be finished before the $j+1$-st operation can begin. In other words:
$$\forall i \in J, j \in O_i, k \in M_{ij} : C_{ijk} \leq S_{ij+1k}$$
2. Two operations on the same machine cannot overlap. Furthermore, the changeover time constraint between two consecutive operations needs to be satisfied. Usually there would also be a mathematical formulation for these constraints as well, but in the context of the genetic algorithm it is unnecessary.

2.3 Genetic algorithm approaches for FJSP

Overview of genetic algorithms

Genetic algorithms are used in a wide variety of problems, including different variations of the Travelling Salesman problem, real world scheduling problems, including (flexible) job shop problems, as well as a multitude of other optimisation problems [6]. This method is inspired by the processes of natural selection, as first described in [7]. The main idea of the genetic algorithm is that over a number of generations, a population of individuals evolves in such a way that the individuals in it try to optimise a given fitness function.

There are a few parameters in a typical Genetic Algorithm:

- Population Size - the number of individuals in the population
- Max Generation Count - for how many generations the population will evolve
- Fitness function - the way in which individuals are ranked. Fitter individuals have a higher chance to reproduce and keep their characteristics into the next generation.

There is also a set of genetic operators which vary significantly across different implementations:

- Population Initialisation - The way the individuals in the initial population are constructed.
- Selection - The way in which the parents of are chosen, as well as which individuals to keep to the next generation.
- Crossover - The way in which two parents are combined to produce an offspring which keeps both parents' characteristics.
- Mutation - The way in which we mutate an individual's genome to achieve genetic diversity in the population.

Genetic algorithms for FJSP

Genetic algorithms are used both for single objective FJSP, usually considering makespan, as well as for the multi-objective case, considering objectives such as total and maximum machine load.

In FJSP each possible schedule is modelled as an individual chromosome from a population. The main representation used in literature is the two vector representation first proposed in [8] and used in [9], [10]. In this representation, each schedule is represented by a machine assignment vector, as well as an operation sequence vector. More information on this can be found in section 3.2.

When it comes to population initialisation, it can be split up into two categories: random initialisation and heuristic initialisation. In [9] a totally random initialisation is used. In [10] the operation sequence is generated at random, while the machine assignment is based on the lowest processing time for an operation across all available machines. In [11] operations are assigned to machines "taking into account the processing times and workloads of machines on which we have already assigned operations".

Mutation is a necessary part of a Genetic Algorithm, which ensures that we preserve sufficient genetic diversity in the population and do not reach premature convergence.

The fitness function in a Genetic Algorithm aims to give higher scores to individuals from the population, which have certain desirable qualities. In the FJSP case, these qualities are the different possible objectives - makespan, total machine workload, maximum machine workload, lateness, etc.

When it comes to crossover, there are a multitude of different operators which are typically used in genetic algorithms, such as N-point crossover, Uniform crossover, Partially mapped crossover, Cycle crossover and many custom crossover operators which fit the specific representation for each individual algorithm.

3 Methodology

In this section two genetic algorithms are presented. The first one aims to tackle the single objective FJSP considering makespan. The second algorithm aims to serve as a proof of concept of the suitability of genetic algorithms based on non-dominated sorting for solving FJSP. It is based on the NSGA-II [12] algorithm, but does not include the crowded distancing function. Firstly, an overview of the algorithms is presented, followed by a description of the schedule representation. Then, each step in the algorithms is described in detail.

3.1 Algorithm Overview

An overview of both algorithms is as follows:

1. Generate a starting population of schedule representations
2. For each generation:
   (a) Optional: Select the fittest individuals to keep as elites. Elites are kept unchanged in the next generation.
   (b) Use selection to choose parents
   (c) Split the parents into pairs and apply crossover
   (d) Apply mutation to children and non-elite members of the population
   (e) Add elite members to the next generation's population
   (f) Fill the rest of the population size with the fittest individuals from the mutated non-elites and children
3. Repeat either until we have reached the maximum number of generations or a set time limit

In the following sections, each of these steps is examined in detail. Population initialisation, mutation and crossover are the same for both algorithms, while there are differences in the selection step and the fitness function.

3.2 Schedule Representation

A typical human-readable schedule representation is a table, where each row contains the information for the execution of a single operation. This includes the Machine, Start time, End time, Product, etc. An example can be seen in figure 2.

In order to be able to use a Genetic Algorithm, a representation of the schedules as chromosomes is needed, in which the traditional genetic operators - selection, mutation and crossover can be applied. Traditionally, in genetic algorithms each chromosome is either a binary or an integer vector. A binary representation is not suitable for FJSP, as most of the choices which need to be made are non-binary, e.g. choosing which machine to assign an operation to or determining the sequence of operations. The following representation is presented in [9] and was first proposed by [8]. Each schedule is represented by two integer vectors. The length of both vectors is the total number of operations across all jobs $\sum_{j \in J} |O_j|$. An example can be seen in figure 3. The first vector contains the machine assignments for each operation. In this example operation O_{00} is performed on machine 2, operation O_{01} is performed on machine 5, etc. The second

Machine	Job	Product	Operation	Start	Duration	Completion
1	0	enzyme0	0	0	8	8
3	0	enzyme0	1	8	4	12
7	0	enzyme0	2	18	4	22
0	1	enzyme1	0	9	3	12
4	1	enzyme1	1	12	2	14
3	2	enzyme2	0	0	3	3
8	2	enzyme2	1	3	3	6
2	3	enzyme3	0	0	4	4
5	3	enzyme3	1	4	6	10
8	3	enzyme3	2	10	6	16
0	4	enzyme4	0	0	5	5
6	4	enzyme4	1	5	4	9
7	4	enzyme4	2	9	7	16
4	5	enzyme5	0	0	8	8
8	5	enzyme5	1	19	3	22

Figure 2: Table representation of schedule

vector shows the order of execution of the operations. For the operation sequence vector, each entry represents the job, whose operation needs to be performed next. So the order in which operations are scheduled starts with O_{50}, then O_{40} is scheduled and so on. In the next section the encoding and decoding process is showcased.

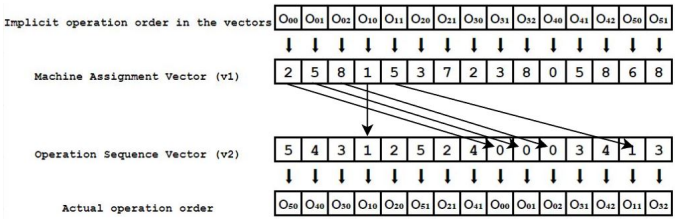

Figure 3: Two-vector schedule encoding

This representation allows us to perform the genetic operations both for the machine assignments and order of operations. This representation also does not include the cleaning time between consecutive operations. This is accounted for in the decoding phase.

Encoding and Decoding

Let's assume there is a two-vector representation of a schedule. It needs to be decoded into a more useful representation in order to be able to calculate the fitness function for example. The basic idea is that the operation sequence vector is traversed from left to right. For each operation, the machine we need to assign the operation to is retrieved from the machine assignments vector and the operation is assigned in the earliest possible time slot for that machine. The change-over times are included in the decoding step - each time an operation is assigned to a machine, with the respective change-over constrains being satisfied. The decoding works in the following way:

1. Go through the operation sequence vector from left to right

2. Look up the corresponding machine for this operation from the machine assignment vector
3. Look at all the previous operation assignments for this particular machine
4. If there is a large enough gap between two operations on the machine, including the 2 potential change-over cleaning times, assign the operation in the beginning of that gap. If not, assign it after the last scheduled operation for this machine.

Encoding a schedule from tabular form into a two-vector representation is straightforward. To construct the machine assignment vector, the entries are sorted by Job and Operation. The resulting Machine column is the needed machine assignment vector. For the operation sequence vector, the entries are sorted by Start time. The resulting Job column is the needed operation sequence vector.

3.3 Initial population

The quality of individuals in the initial population is of vital importance for the performance of the genetic algorithm. In this paper, a randomised initial population is used. The way each member in the initial population is constructed is the following:

1. Construct the machine assignment vector by randomly picking one of the available machines for each operation
2. Construct a vector which contains each job number $j, |O_j|$ times.
3. Construct the operation sequence vector by randomly shuffling the just constructed vector.

Random initialisation ensures that we have enough genetic diversity at the start. Other options, such as using heuristics to construct the schedules such that they have lower makespan in the start were tested but random initialisation outperformed them. More on this can be found in the discussion section.

3.4 Selection Operator

Single objective algorithm

For the single objective genetic algorithm roulette wheel selection is used. The individuals in the population are ranked according to their fitness and the scores are normalised (the lowest makespan member's is set to 1 and the highest to 0). Then a probability vector is constructed by normalising the fitness score vector (dividing by the sum of the elements, making sure it's a unit vector). This vector represents the probability of each individual to be chosen as a parent. A random individual is picked from the population according to the probability distribution described by the vector until as many parents as the population size are chosen. They are then randomly split into pairs and crossover is performed on each pair.

To determine which individuals are chosen for the next generation, the elites, non-elites and children are ranked based on their makespan. The top $|population_size|$ schedules are chosen for the next generation.

Multi-objective algorithm

Roulette wheel selection is difficult to be translated into the multi-objective case, as we would have to combine all the objectives into a single value in order to be able to use it. This defeats the purpose of having multiple objectives - there is no need implement a more advanced algorithm if all the objectives can be reduced into a single one.

Instead of roulette wheel selection, for the multi-objective genetic algorithm tournament selection is used. To select one parent, ten individuals from the population are drawn at random. The one with the best fitness is chosen to be a parent. How the fitness of two schedules is compared in the multi-objective case will become more clear in section 3.7.This procedure is repeated, until we have chosen as many parents as the population size. They are then split into pairs and crossover is applied.

After the children are generated, non-dominated sorting is applied to a joint population of the elites, non-elites and children. As with the single objective case, the top $|population\_size|$ schedules are chosen for the next generation. More information on non-dominated sorting in 3.7.

3.5 Crossover Operator

In this algorithm, two crossover operators are used - one for the machine assignment vector and one for the operation order vector. The two operators were used in [10]. In their work, two children are produced from each parent pair, but in our implementation only one child is produced. This reduces the actual runtime of the the non-dominated sorting.

The machine assignment vectors are crossed using Uniform crossover. We generate a random binary string with length equal to the length of the vectors. At the positions, which are 1 in the binary string, the machines from the first parent are transferred to the child, while where it is 0 - the machine assignments from the second parent are chosen. You can see a visual representation in figure 4.

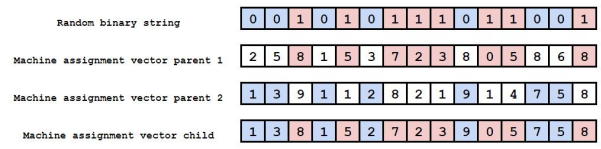

Figure 4: Machine assignment vector crossover

Cycle crossover is used for the operation sequence vector. It works in the following way:

- Split the jobs into two groups.
- For the jobs in the first group, transfer the operations for those jobs from the first parent's sequence vector to the same positions in the child operation sequence vector
- For the jobs in the second group, look up their operations in the sequence vector of the second parent from left to right and insert them into the empty slots in the operation sequence vector of the child.

A visual representation can be seen in figure 5.

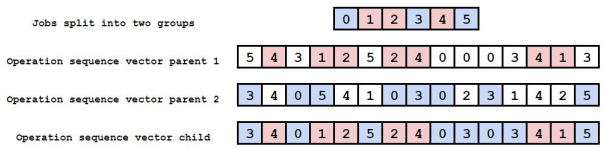

Figure 5: Operation sequence vector crossover

3.6 Mutation Operator

Two mutation operators are used :

1. For the machine assignment vector swap a single machine assignment with another valid machine for the operation
2. For the operation sequences - chose an operation and insert it at another random place in the operation sequence vector

The first operation is used as presented in [9], while the second one is a variation of a mutation operator used in the same paper. In their paper the order of two random entries in the sequence vector is swapped. In this approach, a random operation is moved a random amount earlier or later. This preserves the order of all operations except the one moved. Both of the mutation operators produce valid schedules without the need to apply additional procedures.

The mutation coefficient is a parameter which determines how often a mutation happens. When we want to perform mutation on a schedule we draw two random numbers $r1$ and $r2$ from a uniform distribution $U(0, 1)$. If $r1 < mutation_coefficient$, mutation of type 1. is performed on the schedule with a random machine. If $r2 < mutation_coefficient$, mutation of type 2. is performed on the schedule with a random operation. It could be the case that both, only one, or none of the mutation operations are preformed on a schedule based on the random draws.

3.7 Fitness function

Single objective algorithm

For the single objective implementation of the algorithm, only the makespan is used as a fitness function.

The makespan of a schedule is the latest completion time $\max_{i\in J, i\in O_i, k\in M_{ij}} C_{ijk}$. It measures the amount of time it takes to complete all orders. Lower makespan means better utilisation of the available machines.

The ranking of the individuals is done by comparing their makespans. Lower makespan schedules are considered better than higher makespan ones. This ranking can be achieved by a simple sorting operation which takes $O(nlog(n))$ time where n is the number of schedules to be sorted.

Multi-objective algorithm

In the multi-objective case we use two objectives: makespan and lateness. The makespan is defined in the same way as in the single objective case.

The lateness of the schedule is the total amount of time a job is completed after its deadline across all jobs. If the deadline for job i is D_i, then the lateness of a schedule is $\sum_{i\in J} \max(0, \max_{j\in O_i, k\in M_{ij}} C_{ijk} - D_i)$.

If the latest completion of a job is before or at the deadline, the sum is not increased. If it is after the deadline, the sum is increased by the amount of time the job is delayed. As with the makespan, a lower value for lateness means a better schedule.

Other objectives, such as total and maximum machine load were also considered for inclusion into the algorithm. For the 13 provided DSM problem instances, the total machine workload is a constant, as the processing times for each operation are the same across all machines. Tests with maximum machine workload were performed, but the results showed no overall improvement.

Ranking the individuals in the multi-objective algorithm differs significantly from the single objective case. It is based on the non-dominated sorting presented in [12].

In this approach, schedule A dominates schedule $B \iff$

$$\forall i : O_i^A \geq O_i^B$$

$$\exists i : O_i^A > O_i^B$$

where O_i^A is the value for the i-th objective for schedule A. When using the non-dominated sorting approach to evaluate schedules, we are not looking for a specific schedule, but rather a set of non-dominated schedules, for which we can compare the values for each separate objective.

With this definition of domination the schedules in the population are split into frontiers. The optimal frontier contains the schedules which are not dominated by any other schedule. These are the elite members of the population, which are kept into the next generation unchanged. The second frontier contains schedules which are only dominated by schedules from the optimal frontier. All other frontiers contain contain schedules which are dominated only by those in previous frontiers. The algorithm by which this ranking is performed is presented in [12] and its runtime is $O(mn^2)$, where m is the number of objectives and n is the number of schedules which are ranked. This generally means that for larger population sizes, the ranking is quite slow in practical terms.

All schedules in the same frontier are considered as equal. When performing tournament selection or determining which schedules to keep in the next generation, schedules with smaller frontier numbers are considered superior.

4 Experimental Setup and Results

To test the performance of the algorithms, the 13 provided instances are used. In all of the instances there are 9 machines and 3 different types of operations - preparation, filtering and reception. Each machine can only do one type of operation - machines [0, 1, 2] can perform preparation, [3, 4, 5, 6] - filtering, [7, 8] - reception. The processing times per operation per job are the same across all machines. In each consecutive instance there is an increasing amount of jobs - from 6 jobs in instance 0 to 78 jobs in instance 12 in increments of 6 additional jobs per instance.

The algorithms are implemented in Python 3.8 and ran on a personal computer. All of the tests are preformed on the same machine in order to be able to make a fair comparison of the results.

Firstly, the performance of the provided MILP implementation is measured and later used as a benchmark for the results of the genetic algorithms. For the single objective genetic algorithm the same measurements are performed with a time limit and the results are averaged over multiple runs. Different values for the genetic algorithm parameters are tested to determine which works best. The MILP results are presented in 4.1, the genetic algorithm results in 4.2 and a comparison between the two in 4.3.

4.1 MILP Results

The MILP implementation uses the gurobi Mixed-Integer Programming solver, which uses the Branch-and-Bound approach [13]. To obtain results about from the provided MILP implementation, we measure its performance for all instances with different time limits. For lower time limits the algorithm doesn't produce results for the more complex instances. We can clearly see in the figure 6 that the more time the algorithm has to explore the search space, the better solutions it produces.

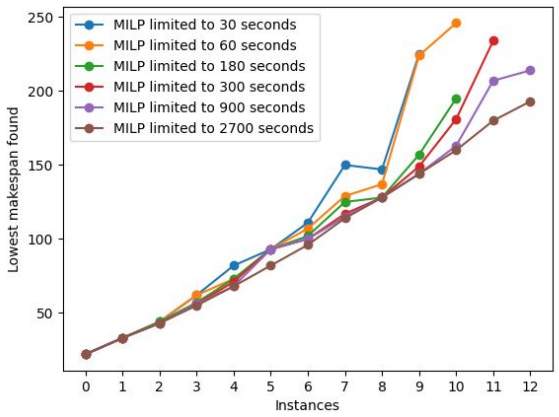

Figure 6: MILP results

4.2 Genetic algorithm Results

In this subsection the results of the genetic algorithms are showcased. Firstly, the results of the single objective algorithm are presented. Different combinations of the parameters: population size, mutation coefficient and time limit were tested. In the next section we make a comparison between the results of the MILP and the genetic algorithms.

Single objective genetic algorithm results

The results for figure 7 were obtained in the same fashion as with the MILP ones - by setting a time limit. For each instance the genetic algorithm was ran 5 times. The average minimum makespan is shown in the figure.

There are a few things to note from the results shown here. First of all, the performance of the algorithm with mutation coefficient 0.2 is better than with 0.4. Tests with other mutation coefficients were also conducted, but these were the two most performant ones. Secondly, while the runtime of the genetic algorithm is increased, there is virtually no difference in its performance. Experiments were ran with time limits larger than 90 seconds, but by that stage the algorithm has pretty much converged around a few solutions. Early convergence, or even convergence around a single solution and its neighbourhood are well known issues of genetic algorithms. Possible causes of this early convergence are:

- The process of keeping the most highly ranked individuals as elites and unchanged onto the next generation.
- Inability of crossover operator to pass on beneficial parent traits to the children.
- Mutation coefficient is either too high and introduces too much randomness in the population or too low and it does not ensure enough genetic diversity in the population.
- Insufficient representation of the overall search space in the initial population.

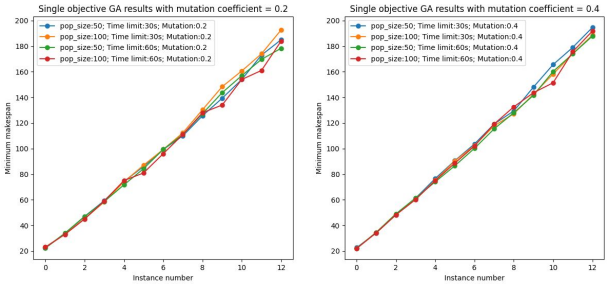

Figure 7: Single objective genetic algorithm results

Results from multi-objective genetic algorithm

The two used objectives are makespan and lateness. In all of the experiments the time limit was set to 90 seconds, which is fair when taking into account the additional complexity of the non-dominated sorting. In figure 8 we can see the results of running the algorithm with different mutation coefficients. The population size was the same across all runs - 100. We can see that even with multiple objectives, the genetic algorithm is able to find schedules which have makespan comparable to or better than that of the results of the single objective genetic algorithm.

Another interesting graphic to observe is the convergence rate of both objectives in a single run of the algorithm. In figure 9 it can be seen how the average and minimum makespan and lateness of the population change over the generations. This also showcases one of the strengths of genetic algorithms - the ability to escape from local optima. Between generations 60 and 85 the minimum and average makespans have converged. Either through a favourable mutation or crossover, a new minimum is found and the algorithm is able to improve its solution.

4.3 Comparison between MILP and GA

In figure 10 we can see the comparison between results from the three algorithms. Both genetic algorithms were ran with population size 100 and mutation coefficient 0.2. The single objective genetic algorithm was ran 5 times and the average

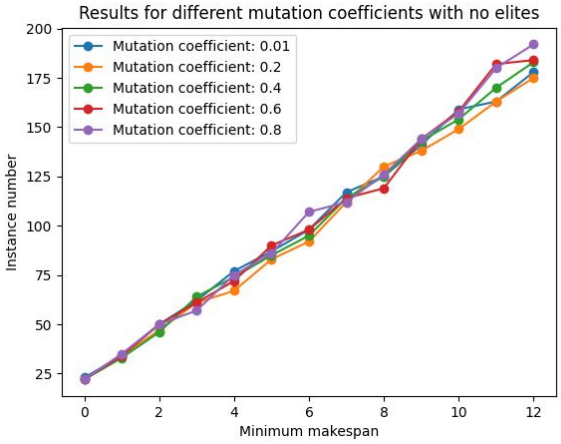

Figure 8: Multi-objective GA; Population size: 50, Time limit: 90s

results are shown. The other two algorithms were ran only once to produce these results.

The MILP implementation performed the best in instances 0-5. The most likely reason is that it was able to explore most of the search space and find solutions close to the optimum. Then in instances 6 and 7, the three algorithms had comparable performance. From instance 8 onward, however, clearly the best solution was the multi-objective genetic algorithm using makespan and lateness. A hypothesis for the good performance of the algorithm is that these two objectives are complementary - a lower makespan usually leads to a decrease in lateness and lower lateness usually leads to a lower makespan. This interplay between the two ensures more diverse solutions are explored compared to the single objective algorithm, increasing the probability of finding a lower makespan solution.

5 Responsible Research

For any scientific work it is very important that all the information is properly referenced and all the results from the experiments are reproducable. Also, that the conclusions which follow from the research can be regarded as accurate within reasonable doubt. It is also vital that when working with sensitive data such as personally identifiable information, safety precautions are taken. To aid these aims, the following measures were taken during the course of the research.

First of all, no sensitive data is used in this paper. Proper citations and references have been provided for the information used from literature sources where possible. Being completely honest, there are still places where more references are needed, a glaring one being the literature review on genetic algorithms.

The source code used to implement the genetic algorithm is available in a public GitHub repository [1] and can be freely accessed by anyone interested. It contains the provided MILP implementation, the implementation of the genetic algorithms, the 13 problem instances, as well as the scripts used to obtain the results. This information should be enough to

[1] https://github.com/whodatbo1/research_project_dsm_enzymes/tree/marko_GA

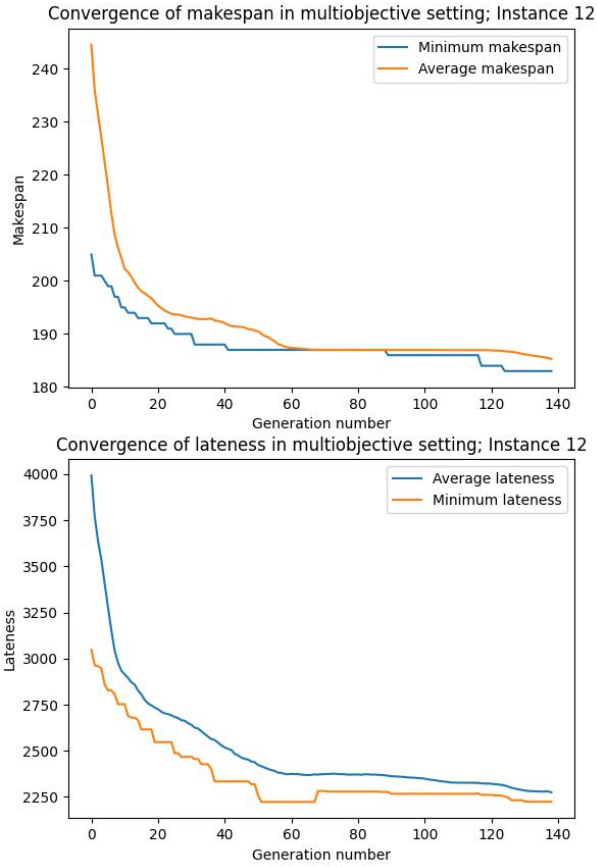

Figure 9: Convergence of multi-objective GA on instance 12; Mutation coefficient: 0.2

reproduce the results in this paper, provided the genetic algorithm results are averaged over multiple runs, as the randomness involved in its implementation can lead to outliers if ran only once.

Something else which needs to be taken into account when reproducing the results is the hardware, which the code is ran on. Time limits are used for the termination of both MILP and GA. This means that with better equipment, one can achieve better results in the same amount of time. All of the experiments were performed on the same hardware, so as to be able to make a fair comparison. If ran on different hardware, the absolute results may vary significantly.

6 Discussion

In this section some general remarks about the process of developing the algorithms are presented, as well as thoughts on future versions of genetic algorithms.

Process

Over the course of the implementation of the genetic algorithms there were many versions which did not make the final paper. Combinations of different genetic operators and parameters were tested. Here is a brief review of the experiments which did not make it to the final paper.

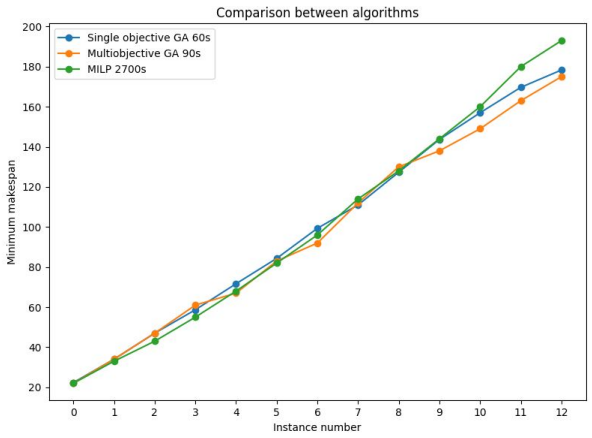

Figure 10: Comparison between the three algorithms

Two other initialisation schemes were tested during the research. Both of them initially construct the operation sequence vector at random. Then, for each operation a machine is assigned. The first one prioritises machines with lower cleaning times for this assignment, while the second one prioritises machines with lower processing times for the operation. In the 13 provided DSM instances the processing times per job and operation are the same across all machines. This rendered the second method useless. The performance of the first one was better for the initial population, but the convergence rate of the algorithm was slower and better solutions were found with random initialisation. A possible explanation to this is that random initialisation provides more genetic diversity, while the prioritisation scheme "locks" the schedules into a more specific type which more likely than not is not close to the optimum.

Elitism in the genetic algorithms led to mixed results. Tests were conducted both with and without keeping the most fit part of the population as elites. It seemed that the end result was very similar between the two approaches. The difference is that when using elites, convergence was slower. This might be due to the fact the there is less diversity in the population. Also, even if we do not explicitly keep the elites unchanged into the next population, they still have a very high chance of being present in the next generation, as they are the most fit individuals in the previous one. A quantitative evaluation is necessary to determine whether elites increase the performance of the algorithm.

Forward-looking remarks

During the implementation of the algorithm, seeing intermediate results lead to the conclusion that the parameters of the algorithm are of great importance to the results. It also became evident that over the course of a single run of the algorithm the parameters had different influence over the population. For example, in the first few generation mutation is not that important - there is enough genetic diversity in the population already, thanks to the random initialisation. However, in later stages, when the algorithm has more or less converged, mutation greatly increases in importance, as it allows for potentially favourable traits to enter the gene pool.

These points raise a forward-looking question. Can a Genetic Algorithm be constructed, such that it adapts to the problem and sets the parameters to their optimal values?

7 Conclusion and Future work

In this research paper two versions of a genetic algorithm were presented - one dealing with the single objective case and one dealing with the multi-objective case. Their results were compared to the MILP implementation on the 13 provided problem instance. It is clear that both of these algorithms can outperform the MILP implementation on larger problem instances. If given enough time and computing power a MILP implementation will usually reach a solution close the optimal. However, when it comes to practical applications, usually a solution which is good enough and is found in a reasonable amount of time is sufficient. The two presented genetic algorithms showcase that in just a few minutes, they can produce better results for large problem instances.

The results are enough to suggest that these genetic algorithms can perform well on problem instances, which are not part of the 13 provided ones. This, of course cannot be proven unless experiments with known problem instances in the literature are conducted.

The key strengths and weaknesses of the genetic algorithm approach can be summarised as follows: Strengths:

- Produce multiple feasible solutions.
- Incorporate multiple objective functions.
- Reach good solutions faster than MILP implementation for larger instances.

Weaknesses:

- Difficulty in setting parameters - it is difficult to judge which combination of population size, mutation coefficient and genetic operators will perform well on a problem instance. Multiple different combinations need to experimented with to judge the best performance.
- Rarely reaches the optimal solutions.
- Premature convergence.
- Randomness in results. Running the algorithm only once may not lead to a good solution.

To further add to the quality of the research, there are a few steps which can be taken:

- Test the algorithms on problem instances which are used as benchmarks in the literature.
- Use problem specific traits as objectives. The unique part about the DSM problems is the presence of cleaning times. A possible objective function is the minimisation of total cleaning time or maximum machine cleaning time.
- Test the multi-objective approach on more than two objectives, as well with objectives which might be conflicting with one another. This would really add to the ability to generalise the results for more practical applications, where flexibility is needed.

References

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

[2] P. Brucker and R. Schlie. Job-shop scheduling with multi-purpose machines. *Computing*, 45(4):369–375, 1990.

[3] Berend Denkena, Fritz Schinkel, Jonathan Pirnay, and Sören Wilmsmeier. Quantum algorithms for process parallel flexible job shop scheduling. *CIRP Journal of Manufacturing Science and Technology*, 33:100–114, 2021.

[4] Harvey M. Wagner. An integer linear-programming model for machine scheduling. *Naval Research Logistics Quarterly*, 6(2):131–140, 1959.

[5] Yunus Demir and S. Kürşat İşleyen. Evaluation of mathematical models for flexible job-shop scheduling problems. *Applied Mathematical Modelling*, 37(3):977–988, 2013.

[6] Shabnam Sangwan. Literature review on genetic algorithm. *International Journal of Research*, 5:1142, 06 2018.

[7] Charles Darwin. *On the Origin of Species by Means of Natural Selection*. Murray, London, 1859. or the Preservation of Favored Races in the Struggle for Life.

[8] M. Gen, Y. Tsujimura, and E. Kubota. Solving job-shop scheduling problems by genetic algorithm. 2:1577–1582 vol.2, 1994.

[9] Jie Gao, Linyan Sun, and Mitsuo Gen. A hybrid genetic and variable neighborhood descent algorithm for flexible job shop scheduling problems. *Computers amp; Operations Research*, 35(9):2892–2907, 2008.

[10] Xiaojuan Wang, Liang Gao, Chaoyong Zhang, and Xinyu Shao. A multi-objective genetic algorithm based on immune and entropy principle for flexible job-shop scheduling problem. *The International Journal of Advanced Manufacturing Technology*, 51(5-8):757–767, 2010.

[11] I. Kacem, S. Hammadi, and P. Borne. Approach by localization and multiobjective evolutionary optimization for flexible job-shop scheduling problems. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 32(1):1–13, 2002.

[12] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii, 2002.

[13] Mixed-integer programming (mip) – a primer on the basics. https://www.gurobi.com/resource/mip-basics/. Accessed: 2022-06-19.