Bachelor Degree Project

# Approaches to Multi-Constraint Job Order Balancing

A comparison between Constraint Programming and the Genetic Algorithm for schedule generation

Bachelor Degree Project in Information Technology
Basic level 30 ECTS
Spring 2024

Kurt Areskoug, Jean-Paul Hanna

Supervisor: Simon Butler
Examiner: Joe Steinhauer

# Summary

In scheduling, not all processes can be scheduled equally and may present their own unique set of constraints. Solution approaches include meta-heuristics and exact methods.

Two different approaches were chosen to generate schedules with constraints and compare their performance when implemented for a scheduling activity; Constraint Programming and the Genetic Algorithm. Quasi-experiments were conducted to evaluate the execution time and accuracy score of each solution using a dataset of 50 jobs. The baseline includes a completed scheduling of the jobs.

The results indicate that the Genetic Algorithm solution offers the best results in terms of execution time and accuracy, exhibiting results comparable to the baseline. The Constraint Programming solution failed to find any optimal results, demonstrating lower accuracy compared to the Genetic Algorithm and the baseline.

With the foundation laid by this study, further work may improve each model to a more usable degree.

**keywords:** Constraint Programming, Genetic Algorithm, Scheduling, Assembly Line

# Contents


**Contents** **3**

**1 Introduction** **1**

**2 Background** **3**
2.1 Approximate methods . . . 4
2.1.1 Metaheuristic Algorithms . . . 4
2.1.2 Physics-based methods . . . 4
2.1.3 Swarm-based methods . . . 5
2.1.4 Evolutionary-based methods . . . 5
2.2 Genetic Algorithm . . . 6
2.2.1 Chromosome . . . 6
2.2.2 Population . . . 6
2.2.3 Fitness Function . . . 7
2.2.4 Selection . . . 7
2.2.5 Reproduction - Mutation . . . 8
2.2.6 Reproduction - Crossover . . . 8
2.3 Exact methods . . . 9
2.3.1 Linear Programming . . . 10
2.3.2 Constraint Programming . . . 10
2.4 Similar problems . . . 13

**3 Problem** **15**
3.1 Aim . . . 15
3.2 Problem description . . . 16
3.3 Constraints . . . 16
3.4 Dependent and Independent Variables . . . 17
3.5 Research Questions . . . 18
3.6 Hypotheses . . . 18
3.7 Objectives . . . 19

**4 Method** **20**
4.1 Alternative Methods . . . 20
4.1.1 Case Study . . . 21
4.1.2 Controlled Experiment . . . 21
4.1.3 Systematic Literature Review . . . 21
4.1.4 Survey . . . 21

4.2 Dataset . . . 21
4.2.1 Examples . . . 23
4.3 Penalty System . . . 24
4.4 Tools . . . 24
4.5 Genetic Algorithm - Implementation . . . 24
4.5.1 Framework . . . 24
4.5.2 Representation . . . 25
4.5.3 Initial Population . . . 25
4.5.4 Fitness Evaluation . . . 25
4.5.5 Selection . . . 27
4.5.6 Crossover . . . 27
4.5.7 Mutation . . . 27
4.5.8 Stopping criteria - Stagnation . . . 27
4.5.9 Configuration . . . 28
4.6 Constraint Programming - Implementation . . . 28
4.6.1 Decision Variables . . . 29
4.6.2 Constraints . . . 29
4.6.3 Tests . . . 34
4.7 Comparison Between Solutions . . . 34

**5 Results** **35**
5.1 Genetic Algorithm . . . 35
5.1.1 Testing conditions . . . 35
5.1.2 Execution Time and Accuracy . . . 35
5.1.3 Manual inspection . . . 37
5.1.4 Answering RQ1 and RQ2 . . . 37
5.2 Constraint Programming . . . 39
5.2.1 Soft Constraint Combinations . . . 39
5.2.2 All Constraints . . . 41
5.2.3 Answering RQ3 and RQ4 . . . 43
5.3 Comparison of CP and GA . . . 44
5.3.1 Research Question 5 . . . 44

**6 Discussion** **46**
6.1 Sustainability - Economical, Ecological . . . 46
6.2 Societal Considerations . . . 47
6.3 Genetic Algorithm Discussion . . . 47
6.4 Constraint Programming Discussion . . . 48
6.5 Comparison Discussion . . . 49
6.6 Threats to Validity . . . 50

**7 Conclusions** **53**

**Bibliography** **55**

**A Appendix A** **I**
A.1 Configuration - tests and conclusions . . . I
A.1.1 Population size . . . I
A.1.2 Mutation rate . . . I
A.1.3 Crossover rate . . . VIII

A.1.4 Stagnation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . VIII

**B Appendix B** **XIII**

**C Appendix C** **XIX**

# 1 | Introduction

Scheduling plays a crucial role in various industrial processes, especially in assembly lines (Boysen et al., 2022). When devising schedules for assembly lines, schedulers must prioritize productivity. Within a schedule, not all jobs are prioritized equally. Furthermore, they are likely subjected to constraints that the schedule must adhere to, necessitating tailored solutions to optimize production efficiency. These constraints can be aimed at preventing congestion or solving logistical problems (Gao et al., 2019). Workload balancing emerges as a fundamental aspect because of this, as it is intended to maintain a consistent workflow and prevent issues such as bottlenecks (Sotskov, 2023). With optimal scheduling, higher throughput can be achieved (Taillard, 2023).

Scheduling problems are usually combinatorial and, depending on their complexity can be defined as NP-hard (Guzman et al., 2022). Various forms of combinatorial problems have been defined in the literature in the context of scheduling, such as the job shop scheduling problem (Drótos et al., 2009), assembly line balancing problem (Boysen et al., 2022), and nurse scheduling/rostering problem (Burke et al., 2008).

In this study, the aim is to investigate a problem consisting of $n$ jobs, each categorized by one or more batch groups. These batch groups impose constraints on the scheduling of jobs, with some being required (hard constraints) and others optional (soft constraints). The goal is to create a schedule that meets all hard constraints while minimizing the number of soft constraint violations.

These types of combinatorial problems can be addressed using either an exact or approximate method (Talbi, 2009). One approach from each category is selected to compare the practicality of these approaches in a scheduling context. Constraint Programming (CP) is chosen as an exact method. A meta-heuristic approach, the Genetic Algorithm (GA), is chosen as an approximate method. These methods are then evaluated based on their execution time (the time taken to generate a valid schedule) and accuracy (how closely the schedule adheres to the specified constraints).

The solutions obtained from both methods are compared against a predefined baseline to assess their performance. The baseline comprises a dataset of 50 jobs categorized into 11 different batch groups, already arranged according to the given constraints. Finally, the results obtained from both methods are compared with each other.

Both implementations are based on the C# language. The CP approach utilized the Google OR-Tools library (Google, 2023), whereas the GA was implemented using the Genetic Sharp library (Giacomelli, 2023b). Subsequently, tests were conducted to collect data on accuracy and execution time.

The results indicate that the CP solution falls short of achieving the same level of accuracy and time efficiency as the GA solution. Additionally, it does not closely approach the baseline values, rendering it currently unsuitable for practical use. However, the implementation lays the foundation for future enhancements within the specified problem domain.

On the other hand, the GA solution shows promise, as it can occasionally achieve results

comparable to or even better than the baseline in terms of accuracy score. Furthermore, it maintains reasonable execution times when run on a personal computer. However, it should be noted that the solution is not currently stable, and it tends to produce highly irregular results.

# 2 | Background

Throughout the life-cycle of an assembly line, it is necessary to balance the workload to increase productivity (Sotskov, 2023). Re-balancing is the most important short- to mid-term planning task on an assembly line and has to be performed whenever production processes or production rates change (Walter et al., 2021). Commonly assembly lines are balanced by considering a set of assembly operations and their allocation across available workstations while adhering to precedence constraints. One of its objectives is to minimize or maintain the cycle time within a specified limit. This type of balancing is called, Simple Assembly Line Balancing Problem (SALBP) (Boysen et al., 2022). The problem being studied in this work involves production planning, focusing on scheduling the start of each job on an assembly line to balance the workload and mitigate the impact of bottlenecks while adhering to additional constraints. Although the objectives are similar to those of SALBP, only one machine or operation is considered—specifically, the first. Scheduling optimization problems are frequently combinatorial and can be classified as NP-Hard problems depending on their complexity and size (Guzman et al., 2022). In other words, these problems belong to a category where the optimal solution cannot be achieved in polynomial time. When devising a solution for such problems, it is essential to comprehend the problem and its context. This entails understanding factors such as the time constraints imposed on the solution, the frequency at which the problem will be solved, and the required accuracy level. These considerations are crucial for designing an effective and practical solution that meets the specific needs and constraints of the problem at hand. For instance, in the design of a telecommunication network, the problem typically needs to be solved only once. In such cases, the quality of the solution holds greater significance, as any flaws in implementation could result in long-term costs. Conversely, consider finding the shortest path between two locations using a GPS, where users request real-time routes. Utilizing an exact algorithm like Dijkstra's algorithm to find the optimal solution would be time-consuming. Hence, a compromise must be struck, sacrificing some accuracy for a quicker result (Talbi, 2009).

Optimization methods can be categorized into two main groups: exact methods and approximate methods (Talbi, 2009). The primary distinction between the two lies in their objectives: exact methods aim to identify the global optimum[1] of the problem, while approximate methods employ various rules (such as heuristics) to narrow the search space in the hope of finding the best possible solution. Hence, approximate methods may only provide an approximation of the best solution.

[1]The optimal value of the problem space

## 2.1 Approximate methods

### 2.1.1 Metaheuristic Algorithms

Methods that can be applied to achieve near-global optimal results to optimization problems exist, including Heuristic Algorithms (HA) and Meta-Heuristic (MH). HAs, include algorithms such as Minimum Spanning Tree and Nearest Neighbour (Horowitz and Sahni, 1978). These types of algorithms were used to provide the best and fastest results (Van den Bergh et al., 2013). However, as mentioned the drawback was that the optimum solution could not be ensured. Furthermore, they are not general-purpose algorithms but are designed to solve specific problems (Talbi, 2009).

Since the 1980s, MHs have been demonstrated to be versatile problem-solving methods, offering effective solutions to various challenging combinatorial optimization problems. MHs are general-purpose algorithms designed with three key aims: quick problem-solving, addressing large-scale challenges, and developing resilient algorithms (Talbi, 2009). This is achieved by focusing on a smaller problem space and exploring it efficiently.

Two contradictory criteria must be considered in an MH: diversification and intensification. With diversification, territories within the search space that have not been visited must be traversed to ensure a comprehensive exploration of all areas, resulting in a higher probability of not getting stuck on a local optimum solution. With intensification, promising regions are explored deeper for a better solution.

MHs usually start with either a single or a group of feasible solutions (referred to as single or population-based[2]). Through iterative steps within a predefined search space, typically employing deterministic or stochastic optimization techniques such as random numbers[3], this solution is refined until a predetermined stopping criterion is satisfied (Heil et al., 2020; Talbi, 2009). MHs can be divided into three classes, the Physics-based, Swarm-based, and Evolutionary-based methods (Gharehchopogh, 2023).

### 2.1.2 Physics-based methods

The Physics-based methods take inspiration from physics rules such as inertia, gravitational force, and electromagnetism to search the parameter space[4] (Gharehchopogh, 2023). One such algorithm is Simulated Annealing (SA), which is a single solution-based MH (Heil et al., 2020) that draws inspiration from the annealing process used in metallurgy where a metal is heated to a high temperature and then subjugated to controlled cooling. With SA, changes that lower the cost function are always accepted. If not, the neighbor with the highest probability based on the current temperature and energy degradation level is chosen. The analogy here is thus that on each iteration whenever a new neighbor is introduced the hotter the "search" is, the higher the probability of that move being accepted as the acceptance criteria are low. Still, the cooler it gets the harsher those criteria get. This allows the algorithm to have a wider search area within the parameter space to avoid getting stuck on local optimum solutions, before it narrows down to a possible global optimum solution (Kirkpatrick et al., 1983; Talbi, 2009).

[2]With a single-based solution, only one solution is being worked on and manipulated, while in population-based an unspecified number of solutions are worked on (Talbi, 2009)

[3]Using a deterministic method on a solution will always produce the same answer for a specific problem. However, because stochastic methods involve random aspects, the same does not apply for when those methods are used (Talbi, 2009).

[4]The parameter space encompasses every conceivable combination of values for all the various parameters within a specific mathematical model.

Fattahi et al. (2009) proposed a hierarchical approach that uses SA called *overlapping in operations*, which tackles a combinatorial problem, the flexible job shop scheduling problem (FJSP) with overlapping operations, where the overlap is limited by constraints such as box dimensions and container capacity. A comparison of the results produced by SA and an exact solution (Branch and Bound) was conducted, which validated the algorithm's efficiency and effectiveness.

### 2.1.3 Swarm-based methods

The Swarm-based methods draw inspiration from the collective behavior of social creatures such as flocks, herds, and colonies (Gharehchopogh, 2023). The primary feature of swarm-based methods is the use of simple particles that collaborate through an indirect communication medium while navigating within the decision space (Talbi, 2009). An example of such an algorithm is Ant Colony Optimization, a population-based MH that emulates the foraging for food conducted by ants, where pheromones are left for the colony by the individual to promising sources of nourishment. However, pheromones evaporate over time resulting in less valuable routes disappearing. In this analogy, when a single particle discovers a promising solution, the particle memorizes its position and the quality of the path. Each time a particle takes that path the pheromone value increases. While less promising paths are ignored and in time disappear, enabling the colony to uncover superior solutions in subsequent iterations (Wari and Zhu, 2016; Talbi, 2009).

### 2.1.4 Evolutionary-based methods

Evolutionary-based methods excel at approximating near-optimal solutions for various problems because they operate without presumptions about the underlying fitness landscape, leveraging the theory of natural selection and biological evolution, such as reproduction and mutation (Gharehchopogh, 2023; Talbi, 2009). An example of such an algorithm is the population-based, GA. In this context, an initial population of candidate solutions is randomly created, with each individual having a fitness score. Individuals then go through a selection phase to determine which individual's genes are to be passed to a new generation. The chosen individuals form pairs and produce two offspring, where elements from the parents are passed over to the children, using a crossover method[5] (see Section 2.2.6) and a mutation[6] method (see Section 2.2.5). The children then replace the parents as current individuals in the population. The process ensures that traits that produce better results survive the evolution process (Talbi, 2009).

Burke et al. (2008) proposed an approach that hybridizes heuristic ordering with Variable Neighbourhood Search (VNS) to solve the nurse rostering problems, with a penalty system to handle constraints. The results are later compared with those produced by commercial software, ORTEC's Harmony, which includes a GA component. They concluded that VNS was superior in instances with fewer than 20 nurses. However, the GA produced on average better results with instances of more than 20 nurses.

Zhang et al. (2020) employs a variation of the GA called the Cellular Genetic Algorithm to solve an energy-oriented balancing and sequencing problem of mixed-model assembly lines. It is suggested that employing the cellular strategy will help the algorithm avoid getting stuck on local optimum[7] solutions by maintaining a proper balance between individual diversity and convergence.

[5]One child inherits some parts from one parent and the other parts from the other parent, the other child receives the remaining parts

[6]A small chance that part of the child is altered, examples of which are swapping, inverting, or inserting values

[7]Best solution within a region of the search space.

## 2.2 Genetic Algorithm

Table 2.1: Terminology used in evolutionary algorithms (Talbi, 2009).

| **Metaphor** | **Optimization** |
|---|---|
| Environment | Optimization problem |
| Evolution | Problem solving |
| Individual/Chromosome | Solution |
| Population | Group of Solutions |
| Fitness | Objective function/Fitness score |
| Locus/Gene | Element of the solution |
| Allele | Value of the element (locus/gene) |

Seeing as Evolutionary Algorithms (EA) borrow concepts from natural evolution, it should come as no surprise that the terminology is similar, as can be seen in Table 2.1. The most important terms are chromosome, population, locus/gene, and allele which will be used heavily in the coming section.

According to Bäck et al. (2000), John Henry Holland proposed the first version of the GA in the book "*Adaptation in Natural and Artificial Systems*" published in 1975. Three features set it apart from other EAs, it is represented as bit-strings, uses proportional selection (see Equation 2.1 )[8], and that which makes it most distinct, is the use of crossover as the primary reproduction method. The GA is arguably the most widely recognized EA and Holland's idea is the foundation for most known implementations of GA (Bäck, 1996).

$$Pr(i) = (\frac{x(i)}{\sum_{i=1}^{\mu} x(i)}) \tag{2.1}$$

### 2.2.1 Chromosome

Every chromosome that makes up the population is a solution modeled in some form of structure (Talbi, 2009). These structures which are commonly represented as bitstrings are the genotypes[9] that are manipulated by the GA each generation. The value at each locus (decision variable) in this structure is called an allele. Depending on the problem being modeled individual loci can be called genes, though there are times when groups of loci form genes if they have phenotypical[10] meaning (Bäck et al., 2000). The concept is illustrated in Figure 2.1.

### 2.2.2 Population

Population-based MHs are naturally more explorative search algorithms than single solution-based meta-heuristics. The reason for this is their large initial population that introduces diversity, which widens the search space. The creation of the initial population is an important step within the algorithm as it significantly influences both the effectiveness and efficiency of the algorithm. Were there to be insufficient diversity in the initial population it could lead to premature convergence, and getting stuck in a local optimum solution (Talbi, 2009).

[8]The chance of being selected for reproduction is proportionate to the individual's fitness Bäck et al. (2000)
[9]The genetic makeup of an organism.
[10]Phenotype refers to the observable physical properties of an organism

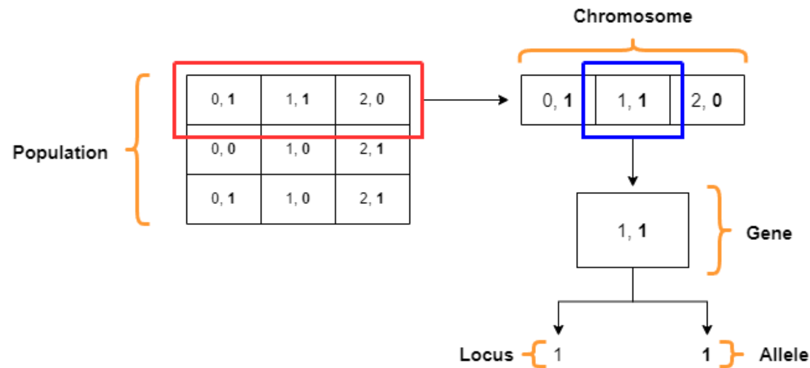


Figure 2.1: Terminology

Finding the optimum diversification of the population can in itself be categorized as an optimization problem (Talbi, 2009), which is why heuristics such as greedy algorithm have been used to help achieve a more diverse population. However, the drawbacks include the high cost associated with this method and the potential loss of valuable traits within the population due to the forced selection. A commonly used alternative is to generate random values for the population within a set range, with the most popular method for achieving this being pseudo-random number generation (Talbi, 2009).

### 2.2.3 Fitness Function

Every chromosome is linked to a fitness score to guide the algorithm towards better solutions. This fitness value denotes the chance for the individual's characteristics to endure through the evolutionary process. The higher the value the greater the chance of the chromosome's traits to be passed on to another generation. The fitness score is derived from an objective function that is used to define the objective to be achieved and is by far the most expensive step in the process. Objective functions are tailored toward specific problems, meaning that if it is not accurately defined it could lead to infeasible solutions (Talbi, 2009).

### 2.2.4 Selection

This operator selects the better solutions from a population while discarding the weaker ones, combining reproduction and selection concepts. Reproduction involves replicating one or more good solutions, while selection entails placing multiple copies of an individual into a population and removing inferior ones (Bäck et al., 2000).

The selection process is one of the main operators and favors individuals who have shown to be of better fit for the problem being solved so that they can pass on their advantageous traits to their offspring (Taillard, 2023). This inclination towards higher fitness levels generates a selective pressure that drives the population towards superior solutions (Talbi, 2009). However, there is still a small chance that individuals with high fitness values get rejected in favor of a less fit individual, as they still hold the potential of providing valuable genetic material that can lead to better solutions (Bäck et al., 2000). Many selection operators exist, too many for them to be mentioned, but two of the most well-known will be briefly introduced.

#### Proportional Selection

Proportional selection also know as Roulette Wheel Selection, involves creating offspring proportionally to the individual's fitness, giving each individual a probability (see Equation 2.1) of being chosen for reproduction (Bäck et al., 2000). However, according to Talbi (2009) this selection strategy lacks the necessary pressure to choose the best individual when all individuals have similar fitness.

#### Tournament selection

Tournament selection randomly chooses a group of individuals of a predetermined size from the population. The tournament winner, the one with the highest fitness value, is then chosen for reproduction, while the rest are discarded. This is repeated until a predetermined amount of individuals are chosen (Bäck et al., 2000). However, there is a small chance that the fittest individual will not be chosen. The tournaments can be of any size, though they usually consist of two individuals. The reasoning behind this is that if tournament sizes are larger, more genetic material gets discarded at the end of each tournament, leading to higher selective pressure (Talbi, 2009).

### 2.2.5 Reproduction - Mutation

All evolutionary algorithms combine selection with some means to introduce variations, with the best-known being the mutation operation (Bäck et al., 2000). Mutations act on a single individual where genes chosen at random have the allele substituted with another value. The goal of mutation is to increase the chance of reaching all solutions within the search space, and not getting stuck in a local optima (Talbi, 2009). The way mutations occur varies depending on the structure of the chromosome. For example, the flip operator mutation can be used in a binary representation, while order-based representation favors operations such as inversion, swapping, or insertion (Talbi, 2009).

The mutation rate governs the likelihood of a gene getting mutated. The literature claims that the most common mutation rate is one where all decision variables have a chance to mutate, which can be calculated using Equation 2.2, where $i$ is the number of decision variables (Bäck et al., 2000; Talbi, 2009). Furthermore, according to Bäck (1996) studies show, that large population sizes paired with high mutation rates and small population sizes paired with low mutation rates negatively impact the results

$$Pm(i) = (\frac{1}{i}) \tag{2.2}$$

### 2.2.6 Reproduction - Crossover

Crossover is the most important search operation of GAs. Unlike mutations which are used to reintroduce "lost alleles" to chromosomes that would otherwise be impossible through recombination, crossover simply recombines genes from existing chromosomes (Bäck, 1996; Talbi, 2009). The basic idea of a crossover is to combine different positive traits of two individuals with high fitness to create a new individual that performs better than the parent chromosomes in terms of fitness. Of course, this is not possible with the GA, as the algorithm does not understand which parts of the chosen individual harbor the positive traits. What happens is instead that features of the chromosomes recombine at random, and the hope is that this produces superior offspring. Unsurprisingly, the recombination could result in ill-fitted solutions. However, those individuals will not have a high chance of surviving the evolutionary process (Bäck et al., 2000). As with

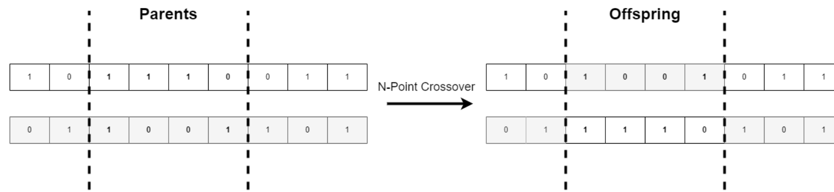


Figure 2.2: 2-Point Crossover

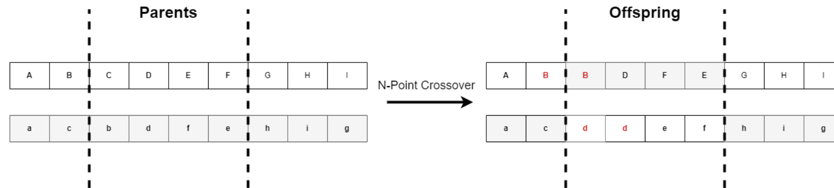


Figure 2.3: 2-Point Crossover on permutation

mutation, crossover is associated with a probability of execution. The literature mentions that the most common crossover probability lies within the interval of 0.45 to 0.95. However, there is no consensus on how to decide upon this value, rather the crossover probability is related to the chosen selection operator, population size, and mutation rate (Talbi, 2009; Bäck et al., 2000)

The choice of crossover operation depends on how the data is represented. For example in cases of binary representations $n$-point crossover can be used. In this case, $n$ crossover-points are selected at random and the genes between the chosen points crossover to form the new offspring. Figure 2.2 illustrates an example, where a 2-point crossover is performed (Talbi, 2009). However, if the solution is represented as a permutation with no repetition, using an $n$-point crossover would most likely result in infeasible solutions, Figure 2.3 illustrates this scenario. Therefore another crossover operation that respects these constraints must be chosen, such as the Order Crossover (OX1). Just as with $n$-point crossover OX1 starts by picking two crossover points. Every gene in between those points on the first parent is then carried over to the child. The corresponding genes in the second parent are then removed from the crossover process, after which the remaining genes starting from the second crossover point are crossed over in the order found, filling up the empty locus of the child. This process is conducted twice, with the parents changing their position to produce two offspring (Talbi, 2009). An illustration of the OX1 is given in Figure 2.4.

## 2.3 Exact methods

Exact methods are used to find optimal solutions and are also able to confirm their optimality (Talbi, 2009). The exact methods are split into different classical algorithms: Dynamic Programming, Branch and X family of algorithms, Constraint Programming and A* family of search algorithms. The different algorithms focuses on splitting the problem up into less complex subproblems and also exploring the relevant parts of the search space (Talbi, 2009). In scenarios where accuracy is important or the problem size is relatively small, exact Optimiser Algorithms (OA) like Branch and Bound, and Branch and Cut can be employed. These algorithms offer the advantage of not evaluating every possible scenario of a problem. Instead, they prune branches

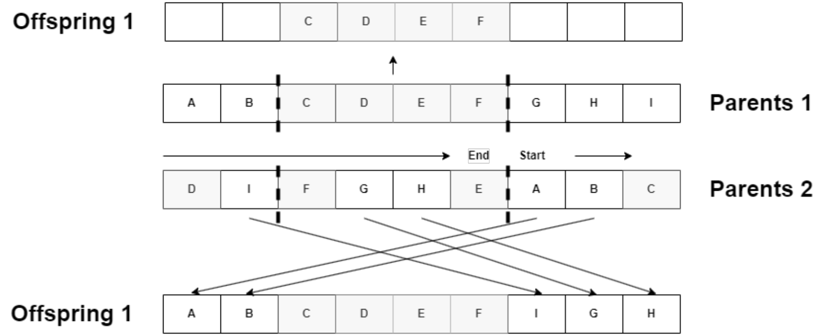


Figure 2.4: OX1 applied on a representation with ordered permutation.

of a tree where it becomes apparent that a node violates a placed constraint, leading to an infeasible solution (Talbi, 2009). While these algorithms are capable of finding the optimal solution for problems, they can be slow and time-consuming when dealing with larger problems (Andres et al., 2016).

### 2.3.1 Linear Programming

Linear Programming (LP) is an optimization model, derived from mathematical programming models, where a problem is defined as a set of different linear functions (Talbi, 2009). LP optimization problems are solved by having an objective function to be optimized, as well as constraints that are applied to the model. Both the objective function and constraints are represented as linear functions (Talbi, 2009). The objective function is represented as a vector of decision variables, multiplied by a constant vector of coefficients. The constraints are represented as a relation between the decision variables and a limit value i.e an inequality for $x * A \neq b$ where $x$ is the vector of decision variables, $A$ is a constant vector of coefficients and $b$ is the constant limit value. In the previous example, the constraint does not allow results where $x * A$ equals the value of $b$. LP consisting of only decision variables with discrete values, is called Integer Linear Programming (ILP). When the decision variables are both continuous and discrete the problem is considered to be a Mixed Integer Problem (MIP), that can then be solved with Mixed Integer Linear Programming (MILP) techniques.

### 2.3.2 Constraint Programming

Constraint Programming (CP) is used to solve Constraint Satisfaction Problems (CSP) by applying a general declarative methodology (Guns et al., 2011). The focus of CP is, instead of defining how a solution is to be solved, modeling what the problem is. A *model* is separate from a *solver*, where the *solver* provides solutions (if possible) to the modeled problem. While CP may be used to find the optimal solution, it may also give the *feasible* solutions that satisfy all constraints (but does not necessarily give the best solution available). CP is considered an exact optimization method (Talbi, 2009), and is used to model a problem according to a set of global constraints that are then applied to a set of variables. A CSP, $P$, is defined as a triple $P = [X, D, C]$, where $X$ is an $n$-tuple of variables $X = [x_1, x_2, ..., x_n]$, $D$ is an $n$-tuple of domains $D = [D_1, D_2, ..., D_n]$, and $C$ is a $m$-tuple of different constraints $C = [C_1, C_2, ..., C_m]$ (Freuder and Mackworth, 2006).

$X$ and $D$ have corresponding variables, where the domain of $x_1$ is $D_1$, the domain of $x_2$ is $D_2$, and so on. Constraints are pairs $[R_{S_j}, S_j]$ containing a relation $R_{S_j}$, which are relations to the variables in $S_i = scope(C_i)$. A solution to a CSP is an $n$-tuple $A = [a_1, a_2, ..., a_n]$, where $a_i \epsilon D_i$, as well as each constraint $C_j$ is satisfied such that each relation $R_{S_j}$ finds a value within the scope $S_j$ to apply to $A$. A given solution is either a feasible or optimal solution, if all constraints can be applied. If the set of solutions to the CSP is empty it means that no feasible solutions have been found, and conversely, if the set of solutions contains multiple solutions that means that there are multiple feasible solutions to the CSP.

CP is mainly suited for combinatorial problems with a highly constrained nature, like time tabling and scheduling (Russell and Urban, 2006). The goal is to produce a CP formulation containing the variables and constraints to model the problem, and proceed to use CSP techniques to solve it. That is to say: model the problem according to a set of formalized constraints. While CP may have some similarities with LP and MILP, a significant difference is that CP does not require an objective function. CP may, however, have one or more objectives as well.

There are different ways to represent decision variables in a model. For scheduling, a common approach is to use binary variables representing the assignment in the schedule for each job (Naderi et al., 2023). Different possible models can be used:

- The position-based model represents the decision variables in a matrix with the size of amount of jobs times the amount of positions. A job $j$ is scheduled in position $i$ if $x_{i,j} = 1$ (Naderi et al., 2023).
- The sequence-based model let the decision variables determine the immediate preceding job. The matrix for the decision variables has the same of decision variables of amount of jobs squared. In the end, each job has a immediate preceding and immediate succeeding job, with the exception of the first and last jobs in the sequence (Naderi et al., 2023).
- The Manne-based model uses a relative sequence to determine if a job precedes another job, but compared to the sequence based model it does not necessarily mean it is preceded immediately. Manne-based modelling requires fewer decision variables than the previous models (Naderi et al., 2023).

When considering mathematical programs, MIP has historically been the first choice for many researchers. However, recent studies show that CP has seen an increase in use in the area, as well as recent developments mitigating known drawbacks with the methodology (Naderi et al., 2023).

**Solver techniques**

A CP solver may use a propagation-search technique to traverse the search space of a CSP, pruning the search space to avoid non-usable subtrees (van Hoeve and Katriel, 2006). *Constraint propagation* is the idea of removing values (or their combinations), caused by the given constraints from the domain of a problem (Bessiere, 2006). To avoid searching through unusable paths, the solver applies the propagation. An example: Given a CSP with the constraint $x + y = 3$. In the case that both $x$ and $y$ are set to 1, it is possible to conclude that no feasible solution can be found, regardless of the values of any other variables (van Hoeve and Katriel, 2006). The solver can then skip checking any part of the search space that uses $x = y = 1$ since that is an infeasible solution. This type of search and propagation is commonly used by commercial constraint solvers and influences the modelling decisions made with regards to the constraint propagation (Smith, 2006).

### Constraint Modeling

Modelling a problem using CP is of high importance to define a CSP that is able to be solved quickly, with the instances to be solved having acceptable runtime behaviour. A common approach is to use matrix models to model problems (Smith, 2006).

### Viewpoints

A viewpoint offers a way to view or model a problem (Smith, 2006) that includes how the decision variables are chosen and what they represent for the given problem. In an example problem where it is required to find a permutation of $n$ values where no value is repeated. One way is to use an array of decision variables where the array index determines the position, and its value is determined by the value of the decision variable. Another way to represent the model is by having a boolean two dimensional array of decision variables. The decision variables are all combinations of positions and values in the problem space, where a decision variable is either true if a specific value is placed at a specific position otherwise false. Difficulties may arise when trying to express some constraints with a poorly chosen viewpoint (Smith, 2006), which may also increase the execution time since it makes the propagation step more complex. Another consideration is that in some cases, it may allow non-desirable partial solutions that may lead to unnecessary work performed by the solver.

### Implied Constraints

Considerations can be made for implied constraints. Implied constraints are constraints that do not alter the solutions, but help in limiting the search space and propagation by explicitly disallowing searches where forbidden assignments are known beforehand, but are still allowed by the original constraints (Smith, 2006).

### Permutation Problems

A common case of problems are *permutation problems*. For permutation problems, the domain's union equals the number of variables, as well as each of the variables having different values. Solutions are a permutation of the different values applied to the decision variables. Depending on the constraints on the problem, different permutations may not be acceptable solutions (Smith, 2006).

### Symmetry

Some problems may have symmetric solutions when using CP (Gent et al., 2006). If a given problem has symmetric solutions, then the solver may be wasting time in searching for something it has already found. For example, a problem where the goal is finding a chess position where nine queens and a king of the same colour, in such a way that no piece is on the same row, column, or diagonal line as another queen of any other colour. When one solution is found, there are other symmetric solutions such as: keeping the same positions but switching the colours or rotating the chess board among others (Gent et al., 2006). There are different methods for breaking the symmetry, to attempt to speed up the search as to not search for the same symmetric solution multiple times. Some methods are: reformulating the problem in an effort to eliminate symmetries, introducing symmetry breaking constraints, and using dynamic symmetry breaking (Gent et al., 2006).

### Soft Constraints

A CP model that contains constraints such that no feasible solution can be found is said to be *over-constrained*. A common occurrence is when trying to use constraints as a definition for desired properties rather than non-violable requirements (Meseguer et al., 2006). While not desired, these properties should be violated if no other feasible option is available. One way to handle desired properties is via soft constraints. If soft constraints handle desired (or optional) properties of a model, hard constraints handle non-violable requirements. In some cases, many solutions satisfy the given soft constraints, but some are more desirable than others. By applying a weight or penalty to the soft constraint, a solution will have a value with which one can measure the desirability of the given solution. The optimal solution is the solution with the lowest penalty (Meseguer et al., 2006).

With the use of soft constraints the notion of feasible solutions is introduced, where a solution is feasible if it adheres to all hard constraints, but not necessarily all soft constraints. The extent of the violations of the soft constraints is measured by an objective value or penalty score, to then be able to compare the correctness of the different solutions. A solution is optimal if it is feasible and produces the least violations of soft constraints possible for the given model. Should a solution not be able to conform to the hard constraints, then that solution is considered infeasible.

### Related Work

Garrido et al. (2009) uses CP to formulate many different constraints together with CPT (an optimal temporal planner), to find a CP formulation to represent highly-constrained planning and scheduling problems. Yuraszeck et al. (2023) proposes a CP formulation to solve a Flexible Job Shop Scheduling Problem (FJSSP) with the goal of minimizing the makespan, testing the formulation on 271 classical FJSSP instances and 50 FJSSP instances with sequencing flexibility. Naderi et al. (2023) compare Mixed Integer Programming and CP models for a set of 12 different Shop Scheduling Problems: Eight variations of the Flow Shop Problem, two variations of the Job Shop Problem, the Open Shop Problem, and the Parallel Machine Scheduling Problem. Metivier et al. (2009) explore the use of soft global constraints on a nurse scheduling problem, comparing results against ad hoc Operational Research methods.

## 2.4 Similar problems

There are similar classical problems such as the Assembly Line Balancing Problem, Flow Shop, Open Shop, and Job Shop Scheduling problems. The problems focus on finding an optimal job scheduling considering different processes and their processing times.

The job shop scheduling problem (JSSP) is a problem where there are $n$ jobs to be processed by $m$ machines (without preemption) (Yuraszeck et al., 2023). The jobs contain a sequence of operations that need to be processed at different machines (Drótos et al., 2009), where the goal is finding a sequence of jobs that allows for the optimal use of processing time for all machines fulfilling the objective function. The objective function is usually to minimize the makespan of the schedule, that is to say, to minimize the time it takes from start to finish of the entire schedule. Similar to the JSSP is the flow shop problem (FSP), which has a set of jobs to be processed in stages, each stage is processed on only one machine (Naderi et al., 2023). For FSP, each jobs stages are processed in the same sequence, all starting from the first stage and ending in the last. The sequence of scheduled jobs holds true even for the stages between different jobs, where the first job will reach the last stage before the second job in the schedule, and so on. The

open shop problem (OSP) is another variation in the "shop problem"-family, where the tasks of each job may be processed in any order (Malapert et al., 2012).

Another similar problem is the knapsack problem, which searches for an optimal combination of objects to fit in a given space. The knapsack poses a fixed capacity, and there is a set of items to be put into the knapsack. The items have known values and sizes. The goal of the problem is to find the collection of items that provide the highest value while not exceeding the size of the knapsack (Kleywegt and Papastavrou, 1998).

The nurse scheduling problem seeks to schedule staff over a period of time, filling all different shifts with their respective requirements. All while considering different constraints such as staff preferences, personnel regulations, and hospital policy regulations amongst other (Weil et al., 1995). The nurse scheduling problem calls "optional" constraints *soft constraints* (or *soft rules*). The purpose of the soft constraints is to avoid undesirable schedules, but in cases where it is impossible, the solution may break one or more of the soft constraints. One example of a soft constraint is that a nurse should not work for two consecutive weekends. While a nurse may be scheduled two weeks in a row in some cases, it should be avoided when possible.

# 3 | Problem

Planning, scheduling, and sequencing are widespread practices in various industries and are usually aimed at reducing unproductive time and maximizing profits. It is common knowledge that wasted time directly correlates with reduced productivity, underscoring the importance of efficient planning. An example of this is bottlenecks which pose a significant concern. When certain processes have longer completion times than others, the imbalance may lead to throughput issues. However, if these processes are spread out over an allotted time it can potentially alleviate these concerns and enhance overall process completion. Which is particularly true in assembly lines, where maintaining a consistent workflow throughout production is essential as delays in one part of the production chain will inevitably impact the workload further along the chain. Ultimately, optimizing scheduling can result in higher throughput (Taillard, 2023).

The reality is often more complex due to constraints that a schedule must adhere to. These increase the complexity of the process significantly, making it hard and time-consuming to perform by hand. The constraints are designed to keep the assembly line running at a steady production flow and avoid congestion, while considering customer priorities and needs, such as the order in which the products are to be packed or if some orders have higher priority than others.

## 3.1 Aim

One of the aims of the work conducted in this study is to produce better schedules that incur as little waiting time as possible, thus minimizing the duration during which machinery remains idle while consuming energy. The motivation for this is that, according to Zhang et al. (2020), the manufacturing sector accounts for roughly one-third of global energy consumption. An extreme example is China, where the manufacturing industry alone is responsible for 55.6% of the country's total energy use. They further state that this puts a considerable strain on the environment and challenges power providers to meet the growing demand. This is in line with one of the targets of the seventh UN sustainability goal, target 7.3, which states that by 2030, the global rate of improvement in energy efficiency needs to be doubled (UNG, 2023).

The other aim is to automate as much as possible of the scheduling process. The reason for this is that manual scheduling can be slow and repetitive, especially when managing constraints, which relies heavily on the expertise and experience of the scheduler. A more efficient approach involves initially generating a schedule with a predefined set of constraints and then having a domain expert review and adjust it as necessary. This approach accelerates the scheduling process and reduces the potential for human errors inherent in repetitive tasks, thereby improving schedule accuracy. To implement this, well-defined constraints must be established, with some flexibility allowed within those constraints.

## 3.2 Problem description

The problem consists of production planning, aiming to schedule the start of each job in an assembly line that balances the workload to minimize the effects of bottlenecks while respecting further constraints.

The problem studied is: A set of $n$ jobs are to be scheduled over a set of $n$ time slots, where each job is classified by one or more *batch groups*. Each time slot has only one job linked to it, and each job can only be associated with one time slot where each time slot and job's execution time is considered as one singular time unit. Each job has a set of $c$ constraints applied to it, these sets depend on the batch groups by which the job is classified. The set of constraints is divided into subsets consisting of a subset of *hard constraints* and a subset of *soft constraints*, where heeding hard constraints is mandatory and soft constraints are optional. The main concern of the problem is not to consider the entire production chain of a job with all its subtasks, but instead the order in which to start the jobs. Considerations for bottlenecks are built into the batch group definitions. The objective is to find a permutation of all the jobs (schedule) that violates as few soft constraints as possible, while still adhering to all hard constraints. Another aspect to consider is that the production of this schedule must not exceed a reasonable time frame.

In general, when designing a schedule, the primary concern isn't always achieving a perfect schedule. Which at times, because of the given constraints, can be impossible. Rather, the aim is to create a good enough or feasible schedule, considering the constraints, if there are conflicts, and finding compromises (Burke et al., 2008).

A concrete example of the problem would be a factory that offers different products to different customers. Each order is classified by the batch groups corresponding to: the customer or product, other possible modification, and eventual logistical requests. Such as, Customer A orders 10 of the same product with a specific configuration and requests them to be delivered in quantities of 5. Customer A's orders will be described by a batch group, let us call it BG99, were each product is counted as a job. The factory may have logistical constraints, meaning that only a few products can be stored in the facility. Therefore, products need to be packed straight away with as few other products being produced at the same time as possible. BG99 will thus have the "keep together" soft constraint to allow efficient use of facility storage.

Some orders may present conflicting constraints. For example, Customer B orders seven products belonging to batch group BG50, which has a slower production rate and introduces bottlenecks in the factory. To address this issue, BG50 has the "spread out" constraint attached to it, which helps to even out the workflow in the factory. However, Customer B can only receive five products at a time. Therefore, all orders placed by Customer B are assigned to batch group BG40, which has the "keep together" constraint applied to it and a limit of five products produced at a time. These two batch groups, BG40 and BG50, are then applied to the order made by Customer B, resulting in two separate conflicting constraints being applied to the jobs in the order. These conflicting constraints may require a compromise to satisfy the higher-weighted constraint while minimizing violations to the lesser-weighted constraint

## 3.3 Constraints

The jobs are classified by one or more batch groups. The batch groups contain classification data: *rule*, *batch size*, *sorting*, *weight*, *batch order*, *cooldown*, and *priority*. The *rule* describes the rule the batch group has to follow, the two possible rules is either *keep together* or *spread out*. *Batch size* determines how many jobs of a certain batch group is considered together as a *batch*.

The *sorting* given to the batch group determines what value in the job is used to sort the jobs in a batch, if no sorting is necessary then a *random* sorting is used. Each batch group has a *weight* that determines the importance of the rule given to the batch group. In the case that a job has multiple batch groups and the rules of these batch groups conflict, the rule of the batch group with the highest weight should take precedence. Some batch groups interleave their batches in a specific pattern, where each batch group places one batch each in a repeating pattern until no more batches are left. This is given by the *batch order* value in the batch group description. Some batch groups require a *cooldown* before a job of the same batch group may be scheduled. Finally, batch groups may have a *priority.* The possible priorities are [*low, normal, high*]. The batch groups with *low* or *high* priorities have their own constraints, which **must** take precedence over any other rule imposed by any potential additional batch group on the same job.

Hard constraints are non-breakable and **must** be adhered to at all times. The following are the hard constraints placed to the jobs:

- Each job must be placed in only one time slot, not allowing the same job to be placed multiple times. The same applies for time slots, where they may only have at most one job scheduled to them.
- Jobs with a batch group having a high priority are to be placed as early as possible in the schedule, jobs with a low priority are to be placed as late as possible in the schedule.

The problem also includes requests that are not always possible to satisfy. These requests are considered *soft constraints.* When a soft constraint can not be satisfied a penalty will be applied. The penalties shows the level of satisfiability of the entire solution in regards to the constraints.

The following constraints are the soft constraints applied to the jobs:

- The *cooldown* constraint requires that a job of a batch group should not be scheduled after previous job of the same batch group, before the cooldown has expired.
- Batch groups with the *keep together* rule request that the batches (determined by the batch size) are scheduled as close together as possible. A batch group with multiple batches will only require that the jobs in the batch are "kept together" instead of all the batches being kept together.
- The *spread out* rule requests that a batch groups batches are spread out as evenly as possible over the schedule.
- *Batch order* requests that different batches are placed in a set sequence (the batch groups *batch order*).

Some of the soft constraints are directly conflicting with each other, such as *keep together* and *spread out.* In the case of conflicts, the batch group with the highest weight will take precedence.

## 3.4 Dependent and Independent Variables

The independent variable used in this study is an unsorted list of jobs where the start sequence is to be ordered in specific ways while avoiding the creation of bottlenecks further down the assembly line. This data will be used in a quasi-experiments where two algorithms will try to order the list. The data will be created by professionals working in the field, they will also sort this list based on the rules and constraints that were provided. This will be the base used as a comparison point to the algorithms.

To be able to quantify the performance of the given solutions for this study, two different dependent variables are introduced: the *time efficiency* to produce a schedule and the *accuracy* it has with respect to the given constraints. The *time efficiency* is defined as the process time it takes from starting a scheduling activity to producing a finished schedule. *Accuracy* is defined as a value given to quantify the violation of constraints for a produced schedule, giving a penalty score. In this report, the term accuracy is used to refer to the penalty score of a schedule, where the penalty amount is calculated by the number of constraints that are broken, as well as the type of constraint that is broken. Accuracy is represented as a negative number, where 0 is the most accurate.

The work conducted in this study focuses on two different solutions using a GA approach and a CP approach to find a suitable sequence scheduling that considers the given constraints.

## 3.5 Research Questions

***RQ1:*** - How accurately can the implemented Genetic Algorithm solve a combinatorial problem of sequencing jobs with constraints?

***RQ2*****:** - How fast does the implemented Genetic Algorithm solve a combinatorial problem of sequencing jobs with constraints?

***RQ3:*** - How accurately can the implemented Constraint Programming solution solve a combinatorial problem of sequencing jobs with constraints?

***RQ4:*** - How fast does the implemented Constraint Programming solution solve a combinatorial problem of sequencing jobs with constraints?

***RQ5:*** - Which differences can be observed between the implemented Genetic Algorithm and the implemented Constraint Programming solutions?

## 3.6 Hypotheses

Within this study, the following hypotheses are presented:

*Hypothesis 1* aims to handle the accuracy of the Genetic Algorithm solution.

**HP $1_0$:** *There is no accuracy advantage to the Genetic Algorithm solution.*

**HP $1_1$:** *The Genetic Algorithm solution is more accurate when producing a schedule than the schedule sorted by scheduling personnel.*

*Hypothesis 2* aims to handle the execution time of the Genetic Algorithm solution.

**HP $2_0$:** *There is no efficiency advantage to the Genetic Algorithm solution.*

**HP $2_1$:** *The Genetic Algorithm solution is faster to produce a schedule than the schedule sorted by scheduling personnel.*

*Hypothesis 3* aims to handle the accuracy of the Constraint Programming solution.

**HP $3_0$:** *There is no accuracy advantage to the Constraint Programming solution.*

**HP $3_1$:** *The Constraint Programming solution is more accurate when producing a schedule than the schedule sorted by scheduling personnel.*

*Hypothesis 4* aims to handle the execution time of the Constraint Programming solution.

**HP $4_0$:** *There is no efficiency advantage to the Constraint Programming solution.*

**HP $4_1$:** *The Constraint Programming solution is faster to produce a schedule than the schedule sorted by scheduling personnel.*

*Hypothesis 5* aims to handle the execution time between both solutions.

**HP $5_0$:** *The Genetic Algorithm solution has equal or worse execution time than the Constraint Programming solution.*

**HP $5_1$:** *The Genetic Algorithm solution has a shorter execution time than the Constraint Programming solution.*

*Hypothesis 6* aims to handle the accuracy between both solutions.

**HP $6_0$:** *The Constraint Programming solution has equal or worse accuracy than the Genetic Algorithm solution.*

**HP $6_1$:** *The Constraint Programming solution achieves higher accuracy than the Genetic Algorithm solution.*

## 3.7 Objectives

To ensure that the research questions are answered the following objectives must be met:

- Identify two candidate solutions for the scheduling problem, one exact and the other approximate.
- Define the hard and soft constraints.
- Implement the solutions with the given constraints.
- Run tests on the implementations to evaluate the accuracy and execution time of the solutions.
- Compare the results of the solutions against each other and the baseline (see Section 4.2).

# 4 | Method

In this study, the execution time and accuracy of an exact method and an approximate method are evaluated, when applied to a combinatorial problem.

The GA was chosen as the approximate solution as it is one of the most widely used MH in the literature (Guzman et al., 2022) and has been successfully applied in numerous combinatorial problems. Of particular interest are personnel scheduling (Burke et al., 2008) and the assembly line balancing problem (Boysen et al., 2022), as the problem under consideration incorporates elements of both. GA is not a new concept; hence, other versions of GA have been studied in the literature, such as the *Cellular Genetic Algorithm* (Zhang et al., 2020). However, to comprehend how these versions are implemented, an understanding of the base GA is necessary. Due to the time limitations of this study, the base version of the GA was chosen.

In the case of the exact method CP was chosen, due to it being used successfully within scheduling problems (see 2.3.2). CP has seen an increase in use in recent studies, as well as performance improvements (Naderi et al., 2023), making it a methodology with promising future prospects.

The problem studied is to assign the start order of jobs while considering constraints, such as the *spread out* and *keep together* rules. This data needs to be collected from actual implementations of said algorithms that tackle the proposed problem. The reason for this is that each combinatorial problem, even though similar, has distinctions that make them unique, such as different constraints and variables. This is especially true when it comes to assembly lines where bottlenecks, packing orders, and sizes of batches vary between each assembly line and must be taken into consideration. Therefore, the research questions can be answered through experiments where implementations of the algorithms will use a dataset created by a domain expert. The dataset simulates a possible, simplified situation on an assembly line. The method chosen for this experiment was a quasi-experiment (Wohlin et al., 2012). Although a full experiment may have been possible, the added workload to generate randomized data to be scheduled could exceed the available time allotted for this study.

## 4.1 Alternative Methods

Exploring alternative methods may lead to the development of alternative studies that can be utilized to gather relevant findings within the problem area. Relevant alternative methods for this study include literature reviews, case studies, controlled experiments, and surveys.

While some alternative methods may be employed to gather additional information in the area, they do not inherently lead to the production of actual solutions, with the exception of controlled experiments. A quasi-experiment measuring the performance of a given solution is still necessary to assess its applicability in test scenarios before considering implementation in a real-life setting.

### 4.1.1 Case Study

A case study aims to study a singular case in its real-life context, gathering data and information over time (Wohlin et al., 2012). This information can then be utilized to draw conclusions, with clear deductions derived from the collected evidence (Wohlin et al., 2012). Conducting a case study on the scheduling process for a large assembly line can offer valuable insights into various aspects to consider during scheduling. These insights could include avoiding bottlenecks, ensuring even production flow, and addressing logistical challenges, among others. Such findings may assist in designing a tailored system to a factory's specific needs. Considering that many factories have different requirements and needs it may, however, prove difficult to generalize the findings in a case study.

### 4.1.2 Controlled Experiment

Compared to a quasi-experiment, a controlled experiment introduces randomness to the equation. The goal is to evaluate whether a statistically significant cause-effect relationship can be observed within the experiment results (Wohlin et al., 2012). The random aspect could involve utilizing a larger dataset of jobs and batch groups and randomly selecting combinations to include in the test. However, due to the small size of the provided dataset, creating a larger version would have been necessary. Additionally, the implementation of the experiment and its randomness would have to be implemented and tested, increasing the time needed to complete the study. These in combination were considered too time-consuming considering the limited time of this study.

### 4.1.3 Systematic Literature Review

A systematic literature review may serve as a valuable tool for identifying suitable solution approaches to the problem domain. The primary aim of a systematic literature review is to present a coherent analysis of all current information relevant to the problem area (Wohlin et al., 2012). By meticulously examining studies and existing evidence within the problem domain, such a review could yield well-defined methods and approaches applicable to the problem at hand. Selecting the most appropriate approaches, grounded in the available evidence, enables the formulation of generalizable conclusions regarding the problem area.

### 4.1.4 Survey

Surveys serve as tools for collecting information about the current situation, typically employed when introducing a new tool or shortly before, to assess the prevailing conditions (Wohlin et al., 2012). Survey methods include questionnaires as well as interviews. In gathering information pertinent to the problem area, surveys may present a viable alternative as they enable the direct collection of information from workers and experts in the field. The resulting insights can highlight areas more susceptible to problems, thereby guiding the study towards finding solutions where actual issues have been identified, rather than focusing on areas currently unaffected by problems.

## 4.2 Dataset

The dataset utilized for the study encompasses a set of *batch groups*, each characterized by distinct requirements. The *batch groups' patterns* are subsequently used to generate the batches utilized in the scheduling process. Within the dataset, an unsorted schedule comprising 50 jobs

Table 4.1: Number of jobs in each batch group.

| **BG**: | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **No of jobs**: | 1 | 3 | 25 | 11 | 7 | 8 | 10 | 5 | 5 | 2 | 1 |

and a sorted rendition of the same schedule are present. A domain expert produced the sorted schedule, adhering to the established rules and patterns outlined by the *batch pattern*. The sorted list exhibits violations of certain constraints, illustrating the precedence of certain constraints over others. The sorted list mirrors the complexities inherent in real-world job scheduling, where achieving a flawless schedule may not always be feasible, necessitating compromises. The sorted schedule is considered to be the baseline for the accuracy of a produced schedule.

The baseline schedule has a fitness value of -141.75, determined using the fitness checker of the GA implementation (see Section 4.5.4). A rough estimate of the time it takes for a scheduler to sort this schedule was set at around 12 minutes. This estimation was made after consulting with a domain expert, who suggested that a schedule of similar complexity and reasonable size for the domain, between 300 to 500 jobs, may take between two to four hours to sort. Since automated solutions are intended to either relieve personnel workload or be utilized in the absence of personnel, the best-case scenario with the highest number of jobs was chosen, 2 hours and 500 jobs. For the presented solutions, the goal is to produce better or similar results than the baseline, that is to say: better or comparable results than scheduling personnel. With a combinatorial problem, the complexity rises with the problem size and the number of decision variables involved. However, because there is only a single small dataset to work with and a lack of additional datasets to take inspiration from it is difficult to speculate on the composition of a hypothetical larger dataset. Deciding on which decision variables to include would be too complex of a task. Therefore, it is generously assumed that the complexity scales linearly with completion times. Hence, the estimated completion time was derived by multiplying the best-case completion time with the percentage results of 50/500, resulting in approximately 12 minutes.

A person with expertise in a particular area often approaches problems differently, relying on intuition to generate results quickly (Kahneman, 2017). In the context of scheduling, an experienced scheduler may rapidly produce schedules based on intuition developed over years of practice. For example, they may intuitively recognize which groups of jobs are less desirable when placed next to each other. However, quantifying the time required to produce a schedule is challenging, as it is not linear. While the time to produce 50 jobs may be relatively short, an experienced scheduler might quickly identify a usable schedule with a glance. Furthermore, scheduling problems are combinatorial in nature, making it difficult to predict how changes in the number of jobs will affect the time required to create a schedule.

For the purpose of this study, a linear simplification is used to estimate the time required to produce a schedule due to the complexity of factors influencing this measure. Although an exponential measure may more accurately reflect reality, determining the appropriate values for such a measure is challenging. Additionally, the lack of access to schedules produced within the industry prevents the derivation of an average completion time to serve as a baseline. Despite the potential unreliability of this measure, it is the best available option for this study's purposes and is deemed sufficient for the intended analysis.

The 50 jobs are categorized into 11 distinct batch groups. These batch groups adhere to the naming convention *"BGXY"*, where *"XY"* represents two digits. Specifically, there is one job assigned to BG01, three jobs to BG02, 25 jobs to BG03, 10 jobs to BG04, seven jobs to BG05, eight jobs to BG06, 10 jobs to BG07, six jobs to BG08, five jobs to BG09, two jobs to BG10, and one job to BG11, as illustrated in the table 4.1.

Each job has various fields. The *Job ID* holds the unique identifier of a job. The *Customer*

Table 4.2: The batch pattern for the dataset, sorting is performed on delivery Sequence (DS)

| **Batch Group** | **Quantity** | **Rule** | **Sorting** | **Weight** | **Cooldown** | **Batch Order** |
|---|---|---|---|---|---|---|
| BG01 | All | Keep together | Random | 100 | 1 | 1 |
| BG02 | 1 | Spread out | Random | 99 | 10 | 2 |
| BG03 | All | Spread out | Random | 75 | 2 | 3 |
| BG04 | 5 | Keep together | DS ASC | 20 | 1 | 3.1 |
| BG05 | 4 | Keep together | DS ASC | 20 | 1 | 3.2 |
| BG06 | 4 | Keep together | DS ASC | 20 | 1 | 3.3 |
| BG07 | 2 | Keep together | DS ASC | 20 | 1 | 4 |
| BG08 | 2 | Keep together | DS ASC | 20 | 1 | 5 |
| BG09 | 2 | Keep together | DS ASC | 20 | 1 | 6 |
| BG10 | All | Keep together | Random | 99 | 1 | 7 |
| BG11 | All | Keep together | Random | 100 | 1 | 8 |

*ID* holds the identifier for the customer who has requested a job. The *Delivery Sequence* (DS) is a value used to determine the order of the jobs in the schedule. The *Matching Batch Group ID's* contain the different identifiers for the batch groups that a given job matches.

The dataset encompasses eleven distinct *Batch groups*, denoted as BG01 through BG11. BG01 represents the *highest-priority* batch group, whereas BG11 denotes the *lowest-priority* batch group. BG10 serves as a *"fallback"* batch group, accommodating jobs not aligning with any other batch group designation.

In scenarios where certain jobs are associated with multiple batch groups, conflicts may arise among the constraints specified by these batch groups. In such cases, precedence is determined by the weight assigned to each batch group. For instance, consider the conflicting constraints between BG03 and BG04 regarding the *"keep together"* and *"spread out"* constraints. BG03 requires an even dispersion of all matching jobs throughout the day, assigning every alternate job in the schedule to BG03, given 25 jobs spread over 50 slots. However, BG04 has the "keep together" constraint, requiring batches of five jobs to remain as close to each other as possible. Given that BG03 holds greater weight than BG04, the "spread out" constraint of BG03 takes precedence. Consequently, the batch assigned to BG04 should first adhere to BG03's "spread out" constraint before addressing its own "keep together" constraint.

Each batch group can be subdivided into subsets. The size of these subsets is variable and may differ among the batch groups. These subsets are treated as separate smaller batches, each adhering to the same rule sets as the main batch group.

All resulting schedules from both the tests conducted on the implementations and the baseline are built from the same dataset and have the same constraints applied to them when producing schedules.

The different batch groups have a priority field. The priority ranges from the following priority values: $\{low, normal, high\}$.

### 4.2.1 Examples

Example 1: BG04 which contains 11 jobs, with only 10 of them being linked to BG03. Job #5 is of interest for this example, because BG04 has a batch size of 5. These jobs are divided into groups of 5, where job #5 is placed in its first batch. This means that job #5 must follow the rules associated with both BG03 and BG04. All the jobs of group BG03 should be placed every second job in the schedule (spread-evenly, 50%), while all the jobs in each batch of BG04 are to be kept as close together to the other jobs in the batch as possible (keep-together).

Table 4.3: Computer hardware used for testing.

| Computers | OS | CPU | Speed | Cores | Thread | RAM |
|---|---|---|---|---|---|---|
| **Computer A** | Win 11 | AMD Ryzen 7 6800H | 3.20 GHz | 8 | 16 | 32 GB |
| **Computer B** | Win 11 | Intel Core i7 11700KF | 3.6 GHz | 8 | 16 | 32 GB |

Example 2: Job #10 is part of BG07, which contains a total of 10 jobs. This batch group has a batch size of two, meaning that the jobs must be placed in one of 5 batches. The only rule that is applied in this case is the "keep-together" rule, meaning that each job in the batch must be scheduled as close together as possible.

## 4.3 Penalty System

The accuracy of the schedule is assessed by employing a penalty system, detailed in Table 4.4, which tallies the number of constraint violations and computes a penalty score. The details of the penalty system, which is shared with the genetic algorithm implementation, are further elaborated upon in Section 4.5.4

## 4.4 Tools

The implementations has been done using the .NET 7.0 framework and C# as the programming language, as well as the following tools:

*Google OR-Tools - CP-SAT,* OR-Tools (Google, 2023) are a set of libraries provided by Google to solve optimization problems. The CP-SAT solver is one of the libraries offered by the OR-Tools suite, which is catered to integer programming and CP problems.

*GeneticSharp* (Giacomelli, 2023b), the GA used in this study was implemented using C# with the GeneticSharp library, which is an open-source C# based, multi-threading GA library. This library has successfully been used in studies such as Leiber et al. (2022), where it is used on an assembly line optimization problem, and Iannino et al. (2019), where it is applied to production optimization systems for long production facilities.

*ClosedXML*, this library has been used for importing the dataset from an *Excel* spreadsheet format to be used with the programmatic representation of the dataset to be used with the separate solutions. The export of the resulting schedule is also using this library. This library does, however, not interfere with the different solutions. Instead, the library is used before and after the solutions are executed.

The specifications for the computers used in the study can be found in Table 4.3

## 4.5 Genetic Algorithm - Implementation

### 4.5.1 Framework

GeneticSharp comes with many pre-made classes for the operations and features that comprise a GA, such as genes, chromosomes, population, crossover, selection, and mutation. These pre-made classes were used whenever possible, but there were instances where that was not possible. Two classes had to be created to replace pre-existing ones. To be able to represent an ordered sequence of jobs, the new implementation of the classes, chromosome, and subsequently population, had

to be created using the interfaces available. The fitness class had to be created from the ground up.

### 4.5.2 Representation

The chromosome representation of the imported data consists of an array of genes. Each locus in the chromosome facilitates a timeslot, the gene array index, and a job object. Together they form a gene, where the first index, *0*, contains the first job to be performed and index $n$, the last job. Each job can only be linked to a single locus, thus forming a permutation with a unique set of jobs.

$$\begin{array}{r} \text{Time Slot. (s)} \begin{pmatrix} 1 & 2 & 3 & \cdots & s \end{pmatrix} \\ \text{Job No. (j)} \begin{pmatrix} 1 & 2 & 3 & \cdots & j \end{pmatrix} \end{array}$$

Figure 4.1: Chromosome representation

Furthermore, the chromosome contains two additional imported lists. They are however static interpretations of the rules and constraints applied on the chromosome, and will not be subjected to the evolutionary process.

The first list contains all batch group objects, Where each batch group contains metadata such as batch group identification, rules, constraints, and all jobs linked to the batch group. The other list contains all batch objects and their metadata such as which batch group or groups it is linked to, which jobs are associated with the batch, and the best-expected distance from the first job in the batch, to the last.

### 4.5.3 Initial Population

As has been stated utilizing a heuristic, like a greedy algorithm, to generate the initial population often yields higher-quality initial solutions. However, this approach also entails increased computational cost and a heightened risk of premature convergence. Sufficient diversity must be introduced into the population to benefit from the crossover operation. The initial populations are usually generated with a pseudo-random number generator (Bäck et al., 2000; Jalilvand-Nejad and Fattahi, 2015; Talbi, 2009). The literature mentions that there exist approaches that provide better diversity, such as quasi-random sequence, and parallel and Sequential diversification (Talbi, 2009). However, because of time constraints placed on this study, the ease of implementation, and its prevalent use in the literature, a pseudo-random number generator is used to shuffle the genes of each chromosome upon its creation.

### 4.5.4 Fitness Evaluation

The chromosome fitness value is the sum of all penalties sustained due to constraint violations. Every constraint is evaluated in isolation, in which the number of violations is recorded, and the subsequent penalty calculated. To ensure that the chromosome fitness values do not result in a numeric overflow, the weight value associated with a batch group is multiplied by a fraction, giving an additional method in which to adjust the penalty score. These calculations differ depending on what constraints are broken, and in some cases, the severity of the violation. Where lenient violations are calculated linearly, that being: the *weight fraction* or a *preset weight value* times the *number of recorded violations*, and harsher violations are calculated exponentially in a similar manner, except it is subjected to an exponent as well.

Table 4.4: Penalty calculation, with $n$ being the number of violations.

| Constraint | Penalty function | Violation calculation |
|---|---|---|
| High priority job placements | $((0.01{\times}Weight) \times n)^8$ | The number of timeslots from the last found instance of high priority job to the index of the expected last index of high priority job. |
| Low priority job placements | $((0.01{\times}Weight) \times n)^8$ | The number of steps from the first instance of a low priority job to the expected index of the first instance of a low priority job. |
| Spread out, within the set index range. | $(0.01{\times}Weight) \times n$ | The number of additional steps from the best placement of a job, in regards to the minimum wait time until jobs from the same batch can be placed. This applies when the second job is within the accepted index range. |
| Spread out, outside the set index range. | $((0.02{\times}Weight) \times n)^2$ | The number of steps from the best placement of a job, in regards to the minimum wait time until jobs from the same batch can be placed. This applies when the index of the second job is found outside the acceptable range. |
| Keep together constraint | $(0.01{\times}Weight) \times n$ | The number of timeslots that exceed the expected batch spread. This includes the number of jobs in the batch and the expected wait time. |
| Repeating pattern | $((1{\times}0.40) \times n)^2$ | Every instance of a job that breaks the expected batch pattern. |
| Batches placed in ASC order | $((1{\times}0.40) \times n)^2$ | The number of time slots between the indexes of the first instance of the second batch and the last instance of the first batch. |
| Jobs in batches sorted | $((1{\times}Weight) \times n)^2$ | Every instance of a job that is not placed in the correct order of that specific batch |

The idea is that constraints that are not supposed to be violated such as the placements of high- and low-priority jobs, and the spread of jobs that produce bottlenecks, generate a higher penalty. On the other hand, constraints that are desired but not as important, such as how far from the best job placement - while still being placed within the acceptable range, generate less penalty. Thus a few instances of expansive violation influence the score more at the start of the evolutionary process than a high amount of minor violations, while the minor violations become more important at the end (This can be observed in Appendix B). This should hopefully mean that none of the important constraints are broken, while all or only a few of the minor constraints are violated. A complete list containing the constraints and their penalty functions can be seen in Table 4.4.

This approach is heavily inspired by work conducted by Burke et al. (2008), which uses different penalty functions and weight values depending on the broken constraint. Unlike their work, however, a static exponent is supplied here instead of changing it depending on the severity of the violation.

### 4.5.5 Selection

Tournament Selection with a tournament size of two, was chosen as the selection operator. The rationale behind using a small tournament size is to maintain the population diversity for as long as possible, minimizing the number of discarded individuals at the end of each tournament. By doing so, it reduces the likelihood of premature convergence. The selection operator, Roulette Wheel selection, was also evaluated and tested. However, it was promptly dismissed due to its failure to converge, regardless of the duration it was left to run.

### 4.5.6 Crossover

When choosing a crossover operator, how the data is represented had to be considered, and considering that it is represented as a permutation without repetition, the chosen operator had to respect the ordering of the genes whenever a crossover is performed. For this reason, Order-based crossover (OX2) was chosen. According to the documentation supplied by GeneticSharp (Giacomelli, 2023a), OX2, which is a modified version of OX1 was created specifically with scheduling in mind. Unlike OX1, OX2 does not choose entire sections to crossover between the two parents. Rather it chooses $n$ random genes from the second parent and then locates the corresponding genes in the first parent. All genes in the first parent that did not get chosen are copied as is to the offspring, while the chosen genes on the first parent are copied over to the offspring, having their allele rearranged to maintain the same relative order that they had in the second parent. Figure 4.2 illustrates the crossover process of OX2.

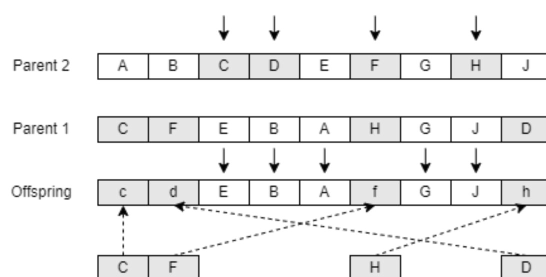


Figure 4.2: Crossover operation OX2.

### 4.5.7 Mutation

Similar to the selection of the crossover operation, the choice of the mutation operator also needs to consider the data representation. The Twors mutation was chosen for this implementation. It simply chooses two loci at random and switches their allele.

### 4.5.8 Stopping criteria - Stagnation

Two stopping criteria were considered: *Time Evolving* and *Fitness Stagnation*. The former halts the process after a specific time duration, while the latter stops it after a specified number of generations pass without any changes to the fittest individual's fitness.

To avoid prematurely terminating the evolutionary process, the stopping operator *Fitness Stagnation* was selected. This criterion allows the algorithm to continue exploring the search space in pursuit of potentially better solutions. However, it should be noted that *Fitness Stagnation* has a significant drawback in testing scenarios, as it introduces fluctuations in execution

Table 4.5: The settings chosen to be used when testing the Genetic Algorithm

| **Population size** | **Tournament size** | **Mutation rate** | **Crossover rate** | **Stagnation level** |
|---:|---:|---:|---:|---:|
| 5000 | 2 | 0.75 | 0.70 | 100 |

time. This occurs because whenever the fitness of the best chromosome changes, either due to finding a chromosome with better fitness or due to modifications that lower the fitness while still producing the best result, the stagnation counter resets. Consequently, this can unpredictably extend the execution time.

### 4.5.9 Configuration

The selected configurations for population size, tournament size, mutation rate, crossover rate, and stagnation level are detailed in Table 4.5. All selections, except for the tournament size, were made following tests conducted with various values. These tests were primarily conducted within specified ranges for the said parameters, as indicated in the literature, with the exception of the mutation rate as pilot tests revealed that better results could be achieved with higher mutation rates, leading to their inclusion in the study. The conducted test and their results can be viewed in Appendix A.1.

## 4.6 Constraint Programming - Implementation

To implement the Constraint Programming model, the CP-SAT solver from the OR-Tools library (Google, 2023) has been used. The OR-Tools come with documentation containing examples for different types problems. Some of the problems included in these examples are the nurse scheduling problem (with and without requests/soft constraints)[1] [2], the jobshop scheduling problem[3] [4] and the knapsack problem[5], among others.

A few helper functions have been extracted from the *ShiftSchedulingExample.cs* example in the repository (Google, 2023), to aid in the implementation of the soft constraints. One is a function to implement soft limits on a sum (*AddSoftSumConstraint()*), while the other is a utility function used inside the soft limit function (*Range()*). The soft constraint implementations work by passing the CP-SAT model, an array of decision variables as well as giving the upper and lower limits (both hard and soft) together with their respective penalties (as well as a prefix to name the variables for the model)[6]. These limits are called *hardMin, softMin, softMax, hardMax.* Each soft limit has a corresponding penalty, *softMin* has *minCost* and *softMax* has *maxCost.* Each hard limit is enforced.

To implement soft limits on a sum, *AddSoftSumConstraint()* function is used. This function limits the total sum of true variables within the given decision variables. The total sum of true values must not be less than *hardMin* or exceed *hardMax.* If the sum of true values is less than *softMin* then the *minCost* is applied as a penalty. If the sum of true values exceeds *softMax* then the *maxCost* is applied as a penalty.

[1] *NursesSat.cs*, commit hash: a76bf1c
[2] *ShiftSchedulingSat.cs*, commit hash: a76bf1c
[3] *JobshopFt06Sat.cs*, commit hash: a76bf1c
[4] *JobshopSat.cs*, commit hash: a76bf1c
[5] *knapsack_2d_sat.py*, commit hash: a76bf1c
[6] However, this does not affect the functionality and can be disregarded. It is merely mentioned for the sake of completeness

$$\text{Slots}\overset{\text{Jobs}}{\begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1m} \\ x_{21} & x_{22} & \cdots & x_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nm} \end{pmatrix}}$$

Figure 4.3: Matrix representation of the decision variables in the CP model

The final one is *Range()* which is a C# implementation of *Python*'s *Range* functionality. It returns a sequence of integers within the given range (*start, stop*)[7].

### 4.6.1 Decision Variables

The decision variables are represented as boolean variables in a matrix (Figure 4.3) where each job with a timeslot has their own decision variable. $x_{slot,job}$ has the domain $\{0, 1\}$. If a specific job is scheduled on a specific timeslot the decision variable will be true, otherwise false. The representation chosen for the decision variables is the position-based representation (Naderi et al., 2023). A multidimensional matrix representation for the decision variables is present in several of the examples found in the *OR-Tools* (Google, 2023) repository.

For the given dataset the amount of jobs and timeslots $|Jobs| = |Slots| = 50$, making the decision variable matrix's size $|Jobs| * |Slots| = 2500$.

### 4.6.2 Constraints

The constraints for the model are the following:

1. One job per slot
2. One slot per job
3. High-priority jobs first
4. Low-priority jobs last
5. Keep together
6. Spread out
7. Cooldown
8. Batch Order (not implemented)

These are either hard or soft constraints. For hard constraints, it means that they must not be broken, while soft constraints will be penalized if broken. The model then seeks to minimize the penalties by searching for the solution with the least violations.

[7] There is also an overloaded implementation only taking a *stop* argument that counts from 0 up to *stop*

### Hard Constraints

The constraint *One job per slot* (*Algorithm* 1) enforces that no slot may have any other numbers of jobs than 1. Constraining the sum of a single slot (with all jobs) to 1, is achieved by collecting all jobs for a single slot and applying the linear constraint that the sum of all of those decision variables is exactly 1. The same process is then repeated for every slot.

**Algorithm 1** One job per slot

1: $slots \leftarrow$ all time slots
2: $jobs \leftarrow$ all jobs
3: **for** $i = 0$ **to** length($slots$) **do**
4:     **for** $j = 0$ **to** length of $jobs$ **do**
5:         $x \leftarrow$ job$_j$ at slot$_i$
6:     **end for**
7:     $Add(Sum(x) == 1)$
8: **end for**

Similar to the previous constraint, *One slot per job* (*Algorithm* 2) is enforced by summing all decision variables for a single job and constraining the sum to be 1 or less.

**Algorithm 2** One job at most

1: $slots \leftarrow$ all time slots
2: $jobs \leftarrow$ all jobs
3: **for** $i = 0$ **to** length ($jobs$) **do**
4:     **for** $j = 0$ **to** length($slots$) **do**
5:         $x \leftarrow$ job$_i$ at slot$_j$
6:     **end for**
7:     $Add(Sum(x) <= 1)$
8: **end for**

*High-priority jobs first* (*Algorithm* 3), it is required that jobs with a high-priority batch group are placed as early as possible in the schedule. The first step is to find all jobs with a high-priority batch group, as well as counting the amount of jobs with a high-priority batch group. Finally, by iterating through the first slot up to the same amount of high-priority jobs $[0, |highPrioJobs|]$, forcing the sum of all high-priority jobs on one slot to be 1.

**Algorithm 3** High Priority First

1: $highPrioJobs \leftarrow$ all high priority jobs
2: **for** $i = 0$ **to** length($highPrioJobs$) **do**
3:     **for** $j = 0$ **to** length($highPrioJobs$) **do**
4:         $x \leftarrow$ job at $highPrioJobs$[j] at slot$_i$
5:     **end for**
6:     $Add(Sum(x) == 1)$
7: **end for**

The same idea applies to *Low-priority jobs last* (*Algorithm* 4). However, instead of high-priority jobs, it is the jobs with a low-priority batch group that are gathered and counted. The other difference is that instead of starting from the first timeslot counting up, the jobs are scheduled from the last slot decreasing by one until the slot in the place $|timeslots| - |lowPrioJobs|$ is found.

**Algorithm 4** Low Priority last

```
1: lowPrioJobs ← all low priority jobs
2: for i = length(slots) to length(slots) - length(lowPrioJobs) do
3:     for j = 0 to length(lowPrioJobs) do
4:         x ← job at lowPrioJobs[j] at slot_i
5:     end for
6:     Add(Sum(x) == 1)
7: end for
```

**Soft Constraints**

The soft constraints of this model are penalized by the weight given in the dataset. This is done so that when constraints conflict, such as jobs with multiple batch groups, the solver will prioritize the batch group rules that are more heavily weighted. Some soft constraints may in isolation not consider some of the more "obvious" constraints, such as ensuring only one job per slot. However, since the hard constraints handle these it is not necessary to double them in the soft constraints.

Batch groups have one rule out of a selection of rules. One of them is *Keep together* (*Algorithm* 5), where the goal is to have these jobs in a batch as close together as possible. Going through each batch that has a batch group with the *Keep together* rule all jobs in the batch are gathered into an array of decision variables. The array contains all decision variables that match the jobs in the batch as well as sequential timeslots for all jobs where the timeslots used equal the batch size. The array of decision variables is then used to limit the sum of true variables between 0 and the batch size. The goal is to have the same amount of true variables as the size of the batch, which implies that all jobs in the batch are scheduled after each other. The sum of true variables is collected and are used to calculate the deficit of true variables. That is, $batchSize - amountOfTrue$, which would give the amount of penalties to apply to the constraint. If the number of true variables match the batch size then no penalty is applied. On the other hand, if there is a deficit of true variables compared to the batch size, then the penalty is multiplied by the amount of "missing" true variables. Iterating through the same process to eventually go through all jobs in a batch on all timeslots, as well as all batches in the schedule.

Another rule is the *Spread out* rule (*Algorithm* 6), which is considered for batch groups that are to be spread out over the day such that each job of the same batch group is equally spaced out. Batches with 1 job are not considered, since there is nothing to *"spread out"*. To find the optimal spread, a distance between jobs of the same batch group is calculated by taking the amount of timeslots divided by the batch size - 1 (considering the job to check). The algorithm goes through each job in each batch group, considering one job at a time at each timeslot as well as the following jobs in the following slots up to the spread size. Collecting these decision variables gives an array with a job at a certain timeslot as well as the rest of the jobs in the batch group at timeslots $slot + spreadSize$. These decision variables are passed to *AddSoftSumConstraint()* forbidding sums of true values of 0 and less as well as larger than the batchSize. Penalties are applied to sums of true values larger than 1. The result is that a job is only allowed to be placed within the spread size, and if the amount of jobs within that spread size exceeds one, a penalty will be applied based on the batch groups weight. The same procedure is repeated for each job in the batch, each slot, each batch, and each batch group.

*Cooldown* (*Algorithm* 7) is considered for batch groups where a job should not be scheduled after a job of the same type, before the cooldown has passed. For each batch group, collect all jobs that have a matching batch group. Each batch group has a specified cooldown that tells

**Algorithm 5** Keep Together

```
1: for each b ∈ 𝓑 do
2:     for each bg ∈ 𝓑𝓖 do
3:         for each s = 0 to length(slots) - spread do
4:             for each job ∈ GetJobs(b)  do
5:                 for i = 0 to length(GetJobs(b)) do
6:                     x ← job at s + i
7:                 end for
8:             end for
9:             min = 1
10:            max = length(GetJobs(b))
11:            penalty ← bg.Weight
12:            Add(Sum(x) == trueValues)
13:            Add(delta == max − trueValues)
14:            objective.Add(delta ∗ penalty)
15:        end for
16:    end for
17: end for
```

**Algorithm 6** Spread Out

```
1: for each b ∈ 𝓑 do
2:     for each bg ∈ 𝓑𝓖 do
3:         spread ← length(slots) / length(b.jobs) − 1
4:         for each job ∈ GetJobs(b) do
5:             for each s1 = 0 to length(slots) - spread do
6:                 x ← job at s1
7:                 for each job2 ∈ GetJobs(bg) ∧ job ≠ job2  do
8:                     for k = 1 to spread do
9:                         x ← job2 at s1 + k
10:                    end for
11:                end for
12:                min = 1
13:                max = length(GetJobs(b))
14:                penalty ← bg.Weight
15:                AddSoftSumConstraint(min, max, penalty)
16:            end for
17:        end for
18:    end for
19: end for
```

how many slots in between there needs to be for two different jobs of the same batch group. From the jobs of a batch group, pick a job at a specific slot and then gather the following jobs at the following slots from $slot + cooldown$ which produces an array of decision variables. Using *AddSoftSumConstraint()* limits the amount of true variables to being less than 0, as well as penalizing sums of true values larger than 1 and forbidding sums larger than the amount of jobs in that batch group. The result is that the constraint favors sums of true values that equal to 1, which means that only one job is scheduled within the cooldown time frame. Otherwise penalizing sums of true values that exceed one (as long as they do not exceed the number of jobs in the batch group). The same procedure is then repeated for every timeslot, every job in the batch group as well as every batch group[8].

**Algorithm 7** Cooldown

```
1: for each bg ∈ BG do
2:     c ← GetCooldown(bg)
3:     for each job ∈ GetJobs(bg) do
4:         for each s1 = 0 to length(slots) - c do
5:             x ← job at s1
6:             for each job2 ∈ GetJobs(bg) ∧ job ≠ job2 do
7:                 for k = 1 to c do
8:                     x ← job2 at s1 + k
9:                 end for
10:            end for
11:            min = 0
12:            max = length(GetJobs(bg))
13:            penalty ← bg.Weight
14:            AddSoftSumConstraint(min, max, penalty)
15:        end for
16:    end for
17: end for
```

### Job Ordering

The final step of the algorithm is to use the resulting schedule and place the jobs in sorted order according to their *Customer Delivery Sequence ID*. The ordering is not implemented as a constraint but instead as a procedure to be executed after the solver finds a feasible or optimal solution. The procedure checks each job in the schedule from the beginning, gathering all unique batch group combinations. Then it is followed by adding the jobs in the schedule by their respective batch group combinations, in an ordered queue sorted by *Customer Delivery Sequence ID*. Lastly, with an iteration through the schedule, each batch group combination is used to put the jobs in ascending order without modifying the order of the batch groups in the schedule.

### Batch Order

The batch order constraint could not be implemented because of the limited time available.

[8] Every batch group with a normal priority, that is. Since both high and low priority batch groups take precedence in the hard constraints.

### 4.6.3 Tests

Early runs of the solution found that the execution time was unreasonably long, where one execution was allowed to run on Computer A for 6 days. Tests were conducted using the different soft constraints in combinations, to evaluate how the different soft constraints affect the accuracy and the execution time of the solution. All hard constraints were always present, as they lay the foundation for the solutions. The different combinations tested where *keep together, spread out, keep together & spread out, cooldown, cooldown & keep together, cooldown & spread out* and finally *no soft constraints*. To compare the produced schedules, comparisons were made between the execution time, objective value, and fitness value. The execution time will show how fast a schedule can be produced, depending on the combination of soft constraints. The objective value will only be relevant for schedules with the same soft constraints since the value is dependent on the number of violations of the soft constraints. Having multiple soft constraints will in turn increase the risk of having penalties (higher objective value) to a produced schedule. The fitness value is used to test the accuracy of the schedule with respect to the baseline.

The soft constraint *cooldown* increased the execution time significantly. In early tests it was allowed to run without any other soft constraint for 100 hours without finding an optimal solution. Instead, the approach taken was to try executions with different lengths of time, for all constraints, to see if the solution produced better results the longer it runs. The tests include all soft constraints (and hard constraints) and measure in one hour increments in an effort to find a point where more time does not equal better results. The execution time, fitness value and objective value are used in these tests as well. The execution time is static for these tests, since the running time is set before the test is run. The fitness value tests the accuracy of the schedule with respect to the baseline (as written above). The objective value helps finding the solution that best respects the implementation of all the soft constraints. However, the better objective value may not always produce the best fitness.

## 4.7 Comparison Between Solutions

To address the final research question, RQ5, a comparison has been conducted between the results obtained for both solutions. The comparison primarily focuses on two aspects: the execution time required to find a solution and the accuracy of the produced solution. Execution time is measured from the start to the end of implementation execution, while accuracy is assessed by comparing the fitness of the produced solution with the Fitness Checker from the GA.

# 5 | Results

## 5.1 Genetic Algorithm

### 5.1.1 Testing conditions

Ninety runs of the GA were executed on Computer A, following the established settings outlined in Table 4.5. Two evaluation methods were employed: the fitness checker used within the algorithm and a manual inspection of the schedules. However, due to time constraints, only three constraints were checked during the manual inspection, as a complete check was deemed too time-consuming. The constraints include *High-priority*, *Low-priority*, and the *spread of the batch group BG10*, which dictates that the entire batch group must be placed within an interval of at most 4 indexes. All these constraints carry high weight values and are expected to have a low probability of being violated. The purpose of the manual check is to assess the scoring system utilized by the fitness checker. A further thirty runs were performed on Computer B to increase the sample size of the manual inspection.

### 5.1.2 Execution Time and Accuracy

Similar to the test carried out in the configuration test section in A.1, the accuracy scores exhibit a wide distribution, spanning from -350 to -57, with the majority concentrated within the -200 to -70 range as illustrated by Figure 5.1. The data reveals a discernible trend where longer execution times typically lead to higher scores, indicating a correlation between improved results and extended execution times. This relationship is illustrated by the trendline shown in Figure 5.2. The algorithm's execution time typically lies within the range of 450 000 to 800 000 milliseconds, with a mean value of approximately 686 047 milliseconds. However, numerous outliers exist with these outliers usually receiving higher accuracy scores (see Figure 5.2). Such patterns are likely shaped by factors like the mutation rate in tandem with the stagnation level, which gives the algorithm a chance to shift the search when encountering a local optimum. Regardless, the considerable spread observed and the inconsistency in accuracy scores in these and prior tests hint at the algorithm's tendency to prematurely converge towards local optimal solutions.

Table 5.1: Descriptive data of the Genetic Algorithm execution time and accuracy score.

| Sample Size | Mean Score | Score Std | Score S.E. | Mean time | Time Std (ms) | Time S.E. (ms) |
|---|---|---|---|---|---|---|
| 90 | -153.21 | 66.42 | 7.00 | 686047.02 | 265397.25 | 27975.33 |

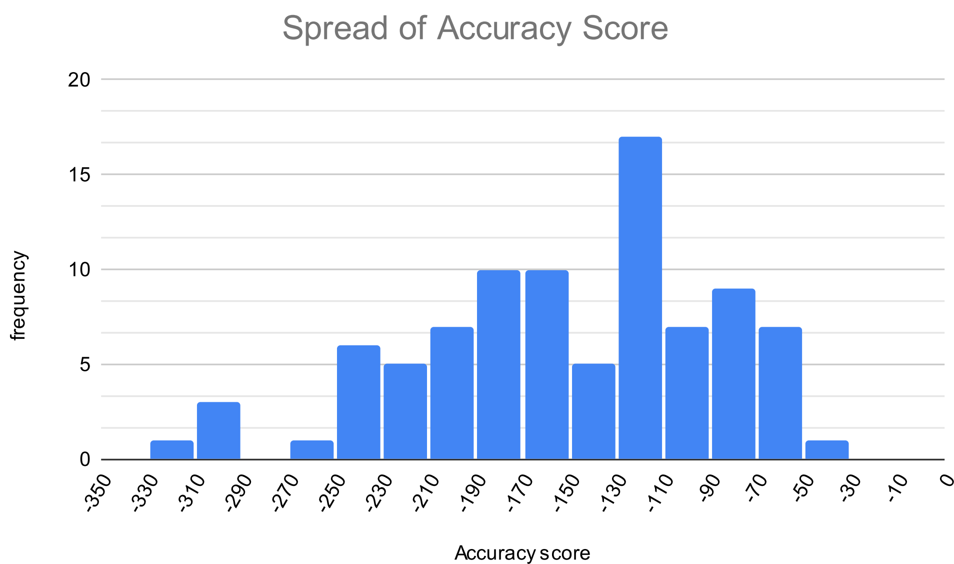


Figure 5.1: Display the distribution of scores within designated ranges following 90 test runs.

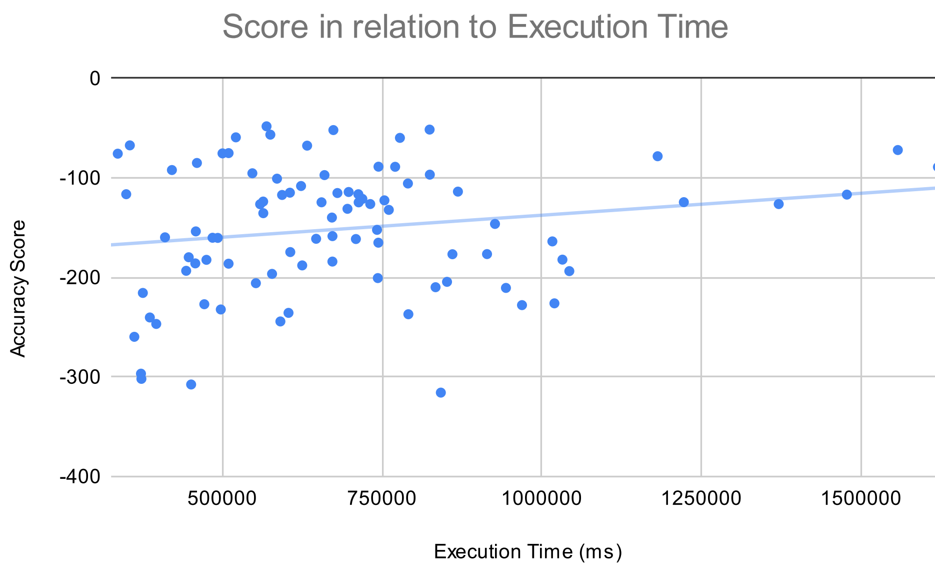


Figure 5.2: The execution time in relation to the accuracy score of the implemented version of Genetic Algorithms.

Table 5.2: Faults found in test results.

| Total schedules | High-Priority Miss | BG10 Keep-Together miss | Low-Priority miss |
|---|---|---|---|
| 120 | 44 | 64 | 87 |

### 5.1.3 Manual inspection

During the manual inspection of the 120 schedules, 44 displacements within the high-priority batch group and 87 displacements within the low-priority batch group were discovered. Nevertheless, it should be noted that in these instances, they were positioned at most one index away from their optimal position. However, no clear reason was identified for the disproportionately higher number of displacements within the low-priority batch group compared to the high-priority group, given that both factors carry equal weight and are subject to identical calculation functions. When analyzing BG10 it appears that over half of the schedules display faults related to the distance between the two members of the batch group. However, despite this prevalence, there doesn't seem to be a discernible pattern in these faults.

The inspection reveals that the penalty system partially achieves its intended purpose. This is evident in the fact that high and low-priority jobs are typically displaced by at most one index, despite a high rate of displacement overall. Furthermore, in at least half of the schedules, members of BG10 are positioned correctly. Unfortunately, the results indicate that the current state of the penalty system negatively impacts the generated schedules.

### 5.1.4 Answering RQ1 and RQ2

**Research Question 1:**

- **HP** $1_0$: *There is no accuracy advantage to the Genetic Algorithm solution.*
- **HP** $1_1$: *The Genetic Algorithm solution is more accurate when producing a schedule than the schedule sorted by scheduling personnel.*

Conducting a Z-test (Boslaugh, 2012) with a confidence level of 0.95 on the collected accuracy scores reveals that **HP** $1_0$, cannot be rejected. This conclusion was drawn by utilizing the fitness score of the presorted schedules, -141.75, as the hypothesized mean. It is important to note that the difference in mean scores was not found to be significant, as the mean score of the algorithm is only 11.46 points lower than that of the presorted schedules, which Figure 5.3 illustrates. However, in its current state, the algorithm yields highly irregular results due to premature convergence, resulting in twice as high or low occasional scores. Consequently, although the results show promise, the answer to RQ1 is that this implementation of the genetic algorithm, in its current state, produces irregular results with accuracy scores ranging from -350 to -57.

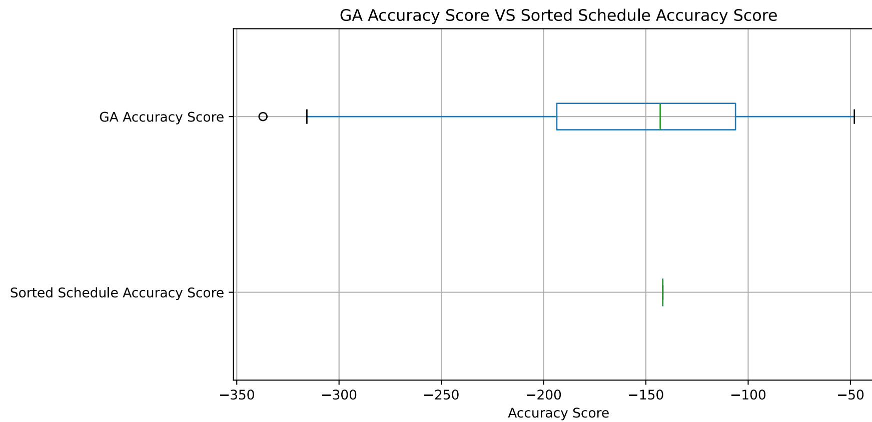


Figure 5.3: A boxplot illustrating the results of the accuracy score compared to the sorted schedule result

**Research Question 2:**

- **HP** $2_0$: *There is no efficiency advantage to the Genetic Algorithm solution.*
- **HP** $2_1$: *The Genetic Algorithm solution is faster to produce a schedule than the schedule sorted by scheduling personnel.*

Performing a Z-test (Boslaugh, 2012) on the collected execution times, with an estimated scheduler completion time of approximately 12 minutes as the hypothesized mean, shows that **HP** $2_0$ could not be rejected. It should be noted that the difference between the hypothesized mean and the algorithm mean execution time is small, as seen in Figure 5.4. Typically, the algorithm provides a schedule within the range of 7.5 to 13.33 minutes on Computer A, with a mean execution time of approximately 11.43 minutes. However, it can occasionally produce significantly longer execution times due to the combination of mutation rate and stagnation level, with results as long as 27 minutes; although these instances are not the norm.

Therefore, the answer to RQ2 is that the implemented version of the GA exhibits irregular execution times, typically ranging between 5 and 17 minutes, with the highest concentration of collected times falling within the range of 7.5 to 13.34 minutes.

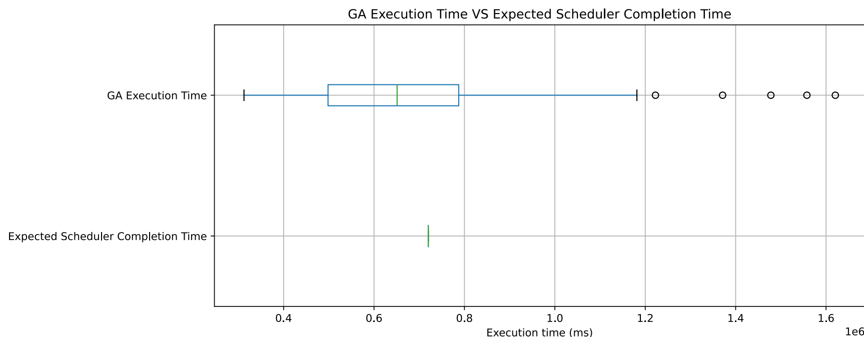


Figure 5.4: A boxplot illustrating the results of the execution time of the algorithm compared to the expected scheduler completion time.

## 5.2 Constraint Programming

The following tests have been conducted on Computer B.

### 5.2.1 Soft Constraint Combinations

The tests for any combination of the soft constraint *cooldown* did not produce any results. Since the execution time for only *cooldown* exceeded reasonable limits (no optimal value found after 100 hours of execution), the combination of *cooldown* with any other soft constraint would also have unreasonably long execution time.

The results gathered were on the following soft constraints: *keep together, spread out, keep together & spread out, no soft constraints.* Each set of the preceding soft constraints were run 30 times. The data collected was the *execution time*, *objective value*, *penalty score* and *solution status.* The execution time was collected by starting the built-in C# `Stopwatch` class, starting it right before running the CP modelling and ending after the solver and job ordering has finished executing. The *OR-Tools* (Google, 2023) *CP-SAT* solver prints out the objective value for the solution, as well as the solution status which may be: `FEASIBLE` if the constraints are adhered to but the solution is not optimal, `OPTIMAL` if the solution is optimal with respect to the given constraints, or `INFEASIBLE` if no solution can be found with the given constraints. The penalty score is collected by sending the produced schedule to the GA solution's Fitness Checker.

#### Results

The following soft constraints combinations are present in the results: *keep together & spread out (KT + SO), keep together (KT), spread out (SO)* and *no soft constraints (NS).* All soft constraints combinations produced solutions with the status `OPTIMAL`, meaning that the lowest possible objective value was found for all of the above given combinations.

KT+SO has a an average run time of 40 226 milliseconds, with the highest standard deviation and standard error in the table (table 5.3). It has the highest maximum time at 75 220 milliseconds as well as the highest minimum time at 13 613 milliseconds. KT has an average time of 508 milliseconds and NS has an average time of 229 milliseconds, both with low values of standard deviation and standard error. KT has maximum time of 520 milliseconds and a minimum time of 496 milliseconds. NS has a maximum time of 229 milliseconds and minimum

Table 5.3: Execution time statistics for the different soft constraint combinations

| Constraint | Avg | Std Dev | Std Err | Max | Min |
|---|---|---|---|---|---|
| **KT+SO** | 40226 | 12995 | 2373 | 75220 | 13613 |
| **KT** | 508 | 5 | 1 | 520 | 496 |
| **SO** | 1125 | 121 | 22 | 1432 | 906 |
| **NS** | 117 | 22 | 4 | 229 | 109 |

Table 5.4: Fitness value statistics for the different soft constraint combinations

| Constraint | Avg | StdDev | StdErr | Max | Min |
|---|---|---|---|---|---|
| **KT+SO** | -3644.74 | 821.80 | 150.04 | -2723.03 | -6703.29 |
| **KT** | -3183.24 | 0.00 | 0.00 | -3183.24 | -3183.24 |
| **SO** | -2955.50 | 514.97 | 94.02 | -1931.40 | -4295.70 |
| **NS** | -7863.97 | 0.00 | 0.00 | -7863.97 | -7863.97 |

time of 109 milliseconds. SO has an average time of 1 125 milliseconds with a standard deviation of 121 milliseconds and a standard error of 22 milliseconds. SO has a maximum time of 1 432 milliseconds and minimum time of 906 milliseconds.

The objective values for SO and NS where all 0. KT and KT+SO have both an objective value of 51 366 for each data point.

The fitness values vary for both KT+SO and SO. NS has the same fitness value repeated for each data point, KT also repeats the same fitness value for each data point (table 5.4). KT+SO has a maximum value of -2 723.03 and a minimum value of -6 703.29 SO has a maximum value of -1931.40 and a minimum value of -4295.70.

#### Analysis

All different combinations of soft constraints produce `OPTIMAL` results, however, it does not necessarily mean that they are optimal with respect to the study's problem. They are optimal within their given constraints. That is to say that the more soft constraints, the more likely it is to have a higher objective value. In the case of *no soft constraints* it is expected to find a optimal result with no objective value, since there are no soft constraints to adhere to.

As seen in the results KT+SO had the highest execution times, both in average as well as the highest minimum and maximum execution times. The results for KT and SO both run in significantly less time compared to KT+SO, as well as varying less per data point. NS show the lowest values of both average, maximum and minimum execution times. One finding of these results is that the execution time is clearly increased with the addition of soft constraints. While somewhat less between no soft constraint and one soft constraint, it still shows an increase in execution time. The increase in execution time is not constant per soft constraint, but differs depending on the type of soft constraint. The combination of soft constraints (KT+SO) do not show a linear increase in execution time, but an exponential increase in execution time.

The objective values do not differ within the different soft constraint combinations, which shows that when an `OPTIMAL` solution is found, it is indeed optimal (with respect to the given constraints). For SO, the solver found solutions that did not violate the soft constraint and as such could produce a solution with an objective value of 0. NS could not have any other objective value than 0, since it contained no soft constraints. KT and KT+SO both had an objective value of 51366 which show that the result is a compromise since the optimal result could not have an objective value of 0.

Table 5.5: Results for the tests with all constraints, sorted by amount of hours run. The best values for Objective and Fitness are marked in bold text.

| Hours | Objective | Fitness |
|---|---|---|
| 1 | 55556 | -877.24 |
| 2 | 55741 | -809.03 |
| 3 | 55331 | -788.59 |
| 4 | **55291** | -806.68 |
| 5 | 55441 | -822.43 |
| 6 | 55461 | -721.05 |
| 7 | 55686 | -808.67 |
| 8 | 55406 | -816.77 |
| 9 | 55311 | -735.73 |
| 10 | 55461 | **-612.71** |
| 11 | 55406 | -777.37 |
| 12 | 55441 | -754.91 |
| 24 | 55461 | -800.12 |

The fitness values show that the resulting schedules for KT and NS have no variation, and always produce the same (or equivalent) result. However the fitness values for KT+SO and SO show variances, which means that the produced schedules are different despite always producing a constant objective value.

The different combinations of constraints show that all combinations, without the *cooldown* constraint, produce fast results. The slowest execution clocks in at 1 minute and 15 seconds, (KT+SO) which is faster than the baseline.

### 5.2.2 All Constraints

Because of the problems regarding the *cooldown* soft constraint, no optimal solutions with all soft constraints could be found. Instead, the approach taken was to try different lengths of time to see if the solution produces better results the longer it runs. 13 different tests were run at different times, where the test executed for different amounts of time. The times used were 1 hour, 2 hours, 3 hours, 4, hours, 5, hours, 6, hours, 7 hours, 8 hours, 9 hours, 10 hours, 11 hours, 12 hours and 24 hours[1].

#### Results

The results are presented in table 5.5. The best fitness value was found after 10 hours of run time. The best objective value was found after 4 hours. The 24 hour result has neither the best nor the worst of either objective and fitness values.

#### Analysis

The results show that there is a trend for better performing fitness values and objective values the longer time the solver gets to work, as shown in Figure 5.5 and Figure 5.6. The charts do, however, show inconsistencies within the results. It is possible for the solver to find better results with less execution time, which is clearly shown when comparing the fitness value for 24 hours

[1] Due to time constraints data points for hours 13 to 23 inclusive had to be omitted.

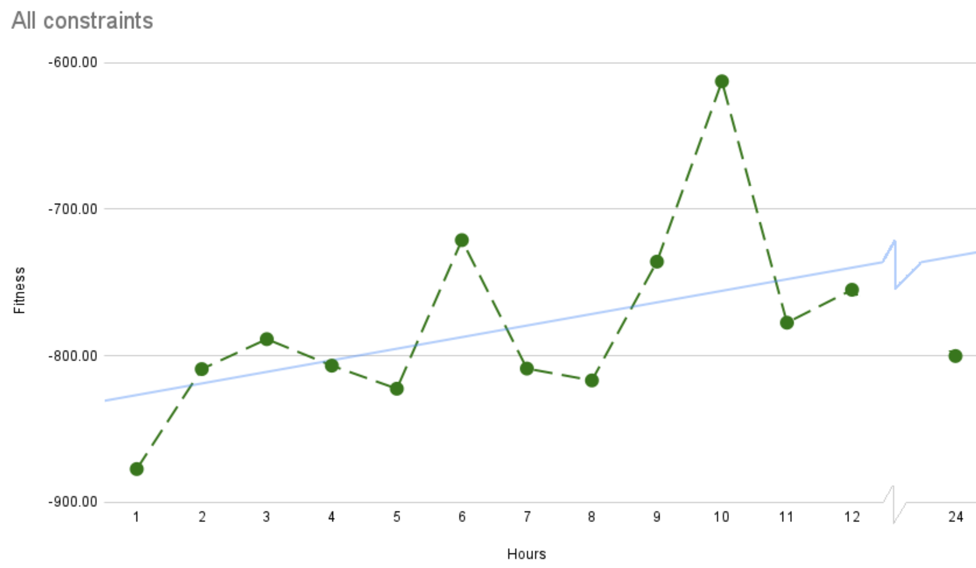


Figure 5.5: Fitness values per hour

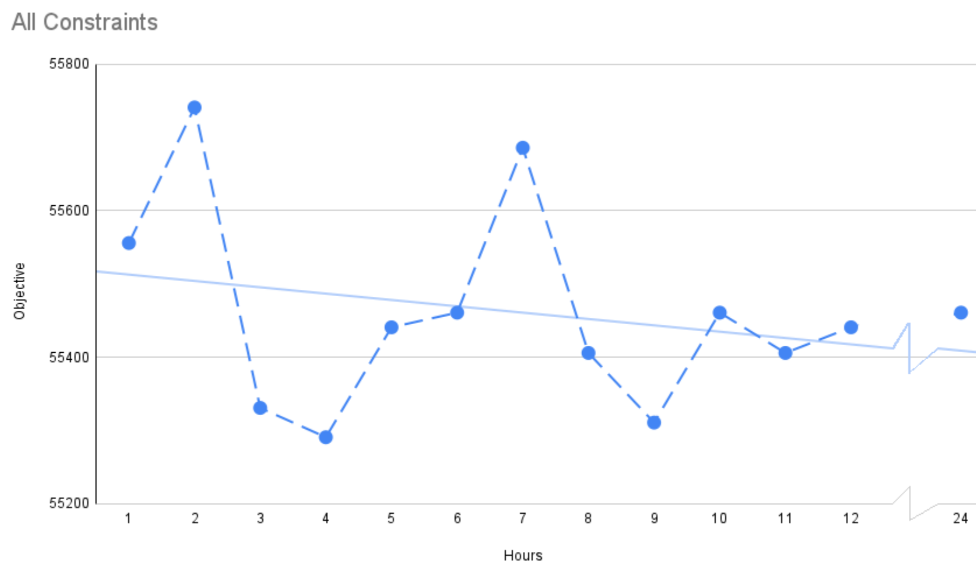


Figure 5.6: Objective values per hour

and 10 hours, as well as the objective values for 24 hours and 4 hours. Depending on what path the solver takes when finding solutions, it may find a better objective value earlier but that is up to chance. The results show that despite running for a significantly longer time, better results may sometimes be found with less execution time.

The results using all constraints also show better fitness values than for any of the tests in the soft constraint combination tests. Despite not displaying any optimal results, the fitness values are significantly better, implying that the combination of all soft constraints help in finding better fitness values. The objective values are lower in the soft constraint combination tests, but that is due to having less soft constraints that penalize. The more soft constraints are used in combination, the higher the risk is to violate them (due to the tighter constraints), which in turn leads to higher objective values.

The results also show that having a better objective value does not necessarily cause a better fitness value, which may be something that requires further investigation.

### 5.2.3 Answering RQ3 and RQ4

The gathered results from the CP solution are insufficient to answer **HP 3** and **HP 4** with any suitable statistical method. In the case of the research questions, they are answered via conclusions using observations gathered from the results. No conclusive evidence can be drawn from these results, however, the results show some tendencies around the behaviour of the solution which may be used to answer the research questions.

**Research Question 3:**

- **HP** $3_0$: *There is no accuracy advantage to the Constraint Programming solution.*
- **HP** $3_1$: *The Constraint Programming solution is more accurate when producing a schedule than the schedule sorted by scheduling personnel.*

In the case of **HP** $3_1$, a statistical test cannot be performed due to insufficient data points. The solution has in general a low accuracy with the produced schedules. In the case of the soft constraint combinations tests and the test with all constraints, some correlation can be found with having more constraints in the model increase the resulting fitness values. The best value found was with all soft constraints at -612.71. The baseline fitness is still significantly higher, with 470.96 in difference. The average fitness value for all constraints is -779.33, which is still far below the baseline (a difference of 637,58). The fitness values range from -877.24 to -612.71. The fitness values are far below the desired results. The solution, in its current state, does not produce schedules with satisfying accuracy. To answer RQ3, the accuracy for the CP solution ranges from -877.24 to -612.71 (for all constraints) with varying regularity. The fitness values have a trend of increasing with longer execution times, but it is too irregular to say conclusively from the current results as well as the required execution time to find the optimal result exceeds reasonable execution times.

**Research Question 4:**

- **HP** $4_0$: *There is no efficiency advantage to the Constraint Programming solution.*
- **HP** $4_1$: *The Constraint Programming solution is faster to produce a schedule than the schedule sorted by scheduling personnel.*

In the case of **HP** $4_1$, a statistical test cannot be performed due to insufficient data points. The soft constraint combination tests have shown that the soft constraint that slows down the solution significantly is the *cooldown* soft constraint. Any combination of soft constraints without *cooldown* range between 109 milliseconds to 75 220 milliseconds, which all are considerably faster results than the baseline value. However, none of these results are very usable from the perspective of accuracy, which means that the results found with all constraints need to be in focus.

In the case of all constraints, no `OPTIMAL` result could be found since the execution time far exceeded reasonable limits. That leaves the tests with `FEASIBLE` resulting schedules, but with the problem of not being the best possible results from the model. The tested execution times range from 1 hour to 24 hours[2]. However, all these time values exceed the baseline's execution time of 12 minutes, well beyond reasonable proximity. The 1 hour test shows the worst fitness value, which makes it a even less viable result. Answering RQ4, the execution times of the CP solution do not produce desirable results. With no `OPTIMAL` results found no conclusive time values can be gathered. The results with fitness values above the average (-779.33) of the all constraint test, occur at the earliest after 6 hours, but there are still values that are below the average that occur after 6 hours as well. The best found fitness value was found after 10 hours. To answer RQ4, the results show that the accuracy of the CP solution offers an average accuracy score of -779.33 which is well beyond the baseline.

## 5.3 Comparison of CP and GA

In the current state of the algorithms, a notable difference in results is apparent, with the GA approach demonstrating superior performance compared to the CP approach in both accuracy and execution time. The highest fitness value obtained from the CP solution is still significantly lower than the lowest fitness value from the GA solution. The CP solution failed to complete its execution with all constraints applied within a reasonable time frame. However, the best result was obtained after 10 hours (36 000 seconds) of execution.

Conversely, the GA recorded its longest execution time of 1 621 seconds, or approximately 27 minutes, which is considerably longer than its typical duration. Nevertheless, the execution time remains considerably faster than what could be achieved by the CP approach, while also yielding substantially better fitness values. The standard deviation for fitness in the CP solution is slightly better at 64.24, whereas the GA solution has a standard deviation of 66.42. However, the limited number of samples in the CP solution makes it challenging to draw confident conclusions.

### 5.3.1 Research Question 5

- **HP** $5_0$: *The Genetic Algorithm solution has equal or worse execution time than the Constraint Programming solution.*

- **HP** $5_1$: *The Genetic Algorithm solution has a shorter execution time than the Constraint Programming solution.*

- **HP** $6_0$: *The Constraint Programming solution has equal or worse accuracy than the Genetic Algorithm solution.*

- **HP** $6_1$: *The Constraint Programming solution achieves higher accuracy than the Genetic Algorithm solution.*

[2]Excluding times within 13-23 inclusive

Similar to **HP** $3_1$ and **HP** $4_1$, in the case of **HP** $5_1$ and **HP** $6_1$, a statistical test cannot be performed due to the CP solution's insufficient number of data points.

As the results demonstrate, the GA solution outperforms the CP solution in terms of both speed and accuracy. Although there is a slight advantage in standard deviation for the CP solution, the limited sample size in CP tests complicates drawing any clear conclusions. To answer RQ5, the observed differences indicate significantly better outcomes for the GA solution across both dependent variables. Table 5.6 presents the best, worst, and mean values for all generated schedules in terms of accuracy score for both the GA and CP solutions. The execution time required to reach these schedules are shown as well.

Table 5.6: Accuracy scores for the best, worst, and mean values for the generated schedules per solution, and their associated execution times

| | **Best** | | **Mean** | | **Worst** | |
|---|---|---|---|---|---|---|
| **Solution** | **GA** | **CP** | **GA** | **CP** | **GA** | **CP** |
| **Time(seconds)** | 569 | 36000 | 686 | 28246 | 842 | 3600 |
| **Accuracy** | -48.10 | -612.71 | -153.21 | -779.33 | -316 | -877.24 |

# 6 | Discussion

## 6.1 Sustainability - Economical, Ecological

An effective scheduling system can alleviate the workload of scheduled personnel while also generating a tight schedule, thereby reducing idle time for machines and lessening energy usage. Consequently, personnel can be utilized more efficiently, and the resulting reduction in energy consumption may lead to a more environmentally friendly operation and lower costs overall.

The paragraph above describes one of the goals attempted to be addressed with this study, thereby contributing to the achievement of UN sustainability goal seven, target 7.3. This goal has been partially achieved with the assistance of the GA implementation, which occasionally produces very good schedules, and does so in a reasonable time on higher-end hardware. However, due to its unstable state, it does not do so reliably, which diminishes its contribution. Regarding the CP solution, the energy consumption goals could not be met. In its current and unfinished state, it takes far too long for it to generate a solution that may not even be optimal seeing as not all constraints are included, thereby wasting energy and time instead of saving it. Consequently, it offers no realistic benefits compared to manual scheduling activities, as a scheduler could perform the work faster and more accurately than the CP solution.

The work conducted here has demonstrated that the GA implementation shows promise, as it yielded results closest to what was expected. With more time dedicated to adjusting the penalty system, the GA implementation should be capable of delivering more reliable results that may align more closely with the aims. An example is the adjustment of the weight for the high and low priority constraints from 100 to 1000. Preliminary tests shows that no instances of these constraints are being broken with the new weights.

For the current problem, the goal has been to find a solution that can generate sufficiently accurate schedules within a reasonable time. In this aspect, the CP solution falls short, due to exceedingly long execution times with poor accuracy scores. Compared to the GA solution, the CP solution is not close to producing reasonable results. However, in the possible situation where the problem area requires more accurate and consistent schedules, there may be a future for the CP solution. Improvements to the CP solution may make it viable. Reformulating the CSP, such as modifying the present viewpoint, using another perspective entirely, and using combinations of viewpoints (Smith, 2006), may provide new insights while providing a potential impact to the performance. Partial filtering may be used when perfect filtering proves too costly. It is weaker than perfect filtering but may be used to filter out domain values to limit the search space (van Hoeve and Katriel, 2006). While alterations to the model may reduce the search space, the same alterations may not necessarily reduce execution time. Therefore it is important to run empirical tests to ensure that the alterations affect the performance in a desired way (Smith, 2006).

## 6.2 Societal Considerations

A risk associated with introducing a system like the ones proposed in this study is the temptation to replace human workers with a fully autonomous system. However, it is crucial to recognize that there will always be a need for experts in the field to ensure that constraints are clearly defined and that desired outcomes are achieved. There is a necessity for someone to verify that the results produce the desired outcomes, especially when considering that production requirements change over time. New products or legislative requirements may change the business needs with short notice. Therefore, experts within various scheduling areas must be retained to ensure smooth execution.

## 6.3 Genetic Algorithm Discussion

Using the stagnation level as a stopping criterion aimed to allow the algorithm to thoroughly explore the search space before terminating prematurely. Some instances corroborate this rationale. Consider the run, R2:13, which had among the longest execution times and yielded one of the lower scores (see Appendix B.1). However, ironically, it also led to some of the worst results, presumably because the algorithm got trapped in a local optimum and was unable to continue its search once it reached a certain number of generations without any changes. Such occurrences include runs R2:27 and R2:24, which exhibited some of the lowest execution times and accuracy scores. With the benefit of hindsight, an alternative stopping criterion would have been more suitable for these experiments, such as timed evolving (terminates the algorithm after a specified time) or generation number (terminates the algorithm after a set number of generations). These might introduce greater stability to the tests, however, this conclusion could not be corroborated with the literature and thus has to be researched and tested. Furthermore, that would make answering the RQs related to execution time unanswerable.

The current state of the penalty system may have influenced the distribution of the results, both in time and score. One theory is that the penalties might not offer enough differentiation for the algorithm to distinguish between a highly-weighted constraint and multiple low-weighted constraints, potentially leading the algorithm to incorrect decisions. This could stem from the use of different fraction values in the penalty calculation, impacting the weights of the various constraints differently, or it could be that the weights were initially too similar. The research conducted by Burke et al. (2008) demonstrates that a GA when paired with a penalty system that introduces sufficient distinction, can effectively address a scheduling problem. They utilize a weight system including values such as 1, 5, 10, 100, and 1000, where the most critical constraints receive the highest value of 1000, and constraints that are highly desired get a weight of 100, making the distinctions clearer. Additionally, their approach differs in their use of the exponential function as they do not use fractions and squares the number of violations before multiplying by the corresponding weight. This shows that the implemented version of the penalty system in this study is flawed and requires improvement.

After conducting some testing, a decision was made to deviate from the norm of utilizing a low mutation rate. The results from those tests were quite surprising, considering that most literature consulted during this research advocated for a low mutation rate (Bäck et al., 2000; Talbi, 2009). Further exploration of the literature revealed instances where high mutation rates had been employed. One example is the work of Zhang et al. (2020) where they conducted numerous tests, ultimately determining a mutation rate between 0.85 and 0.95, similar to the tests conducted for this study. However, it is worth mentioning that they employed a GA that restricts interactions among individuals within the population, potentially influencing the obtained results. Leiber

et al. (2022), however, contradicts this assertion. They suggest that mutation probability has the most significant impact on the quality of the results, but they continue to state that high mutation rates are closer to a random search. Settling for a mutation probability of 0.25, they propose that a balance must be struck. It should be noted that they still use a much higher probability than the literature suggests.

## 6.4 Constraint Programming Discussion

The main problem with the CP implementation is the lack of results. The unexpected and long execution times made it difficult to gather enough results within the allotted time for the study. A best effort has been made in trying to gather and analyse results to find decent numbers pertaining to fitness and execution time, but the lack of samples made it difficult to be able to test properly. Further issues with not being able to find any optimal solution within a reasonable time frame render this solution unusable at the current state. An attempt to find the optimal result for all constraints was made with Computer A, but after 6 days of runtime without results it could not be considered usable.

Due to the time constraints, one soft constraint is missing from the CP solution, the batch order soft constraint. Addition of the batch order constraint would most likely introduce longer execution times, evidenced by the soft constraint combination test results where more soft constraints in combination seem to imply longer execution times. Although, it is hard to prove this conclusively due to the small result size. The all constraint test results seem to imply that the more soft constraints, the better the fitness of the schedule regardless if the found result has been optimal or not. While it may be naive to assume that the introduction of any new constraints will improve the fitness of a produced schedule, the fitness score is dependent on all the soft constraints in combination. As such, it would seem that implementing the final soft constraint may in the best case improve the fitness value of a resulting schedule, or at the very least warrant an investigation as to how the final constraint may affect the accuracy of the model.

Smith (2006) mention the importance of how a model of a problem is formulated affects the simplicity of finding a solution. That is to say, the given CP model for the problem may be improved or re-formulated in such a way that its shortcomings may have diminished impact on the results. The solution provided followed the ideas behind the examples provided in the OR-tools repository Google (2023).

While the CP solution in this study settled for a Boolean model representation, Smith (2006) mention that *"in practice, it is often more useful to try to convert an initial Boolean model into one with integer or set variables"*. They also go on to write about how different viewpoints may open up for new insights regarding a given problem. Another way to consider the given problem is to not consider each job to be scheduled, but rather considering the problem to be: mixing together a given set of batch groups (with given amounts per batch groups) and finding a solution that follows the constraints while only considering the batch groups. After that it is trivial to place the different jobs either in order if the batch group requires it, otherwise randomly in the places of the batch groups. That may, however, introduce symmetry to the problem formulation which in turn may require some considerations for symmetry breaking methods (Gent et al., 2006).

Garrido et al. (2009) introduce two heuristics to prune the search space for planning and scheduling tasks. The results of Metivier et al. (2009) show that neighborhood heuristics may improve performance in a CP solution for a variety of different problems, including the Nurse Rostering Problem (NRP), compared to ad hoc methods. Another direction for simplifying the search space may then be to apply the ideas found in Garrido et al. (2009) and Metivier et al.

(2009) to find suitable heuristics that can simplify the search space for the CP solution.

Metivier et al. (2009) use relaxations for their CP model of a NRP. Using relaxations for the soft constraints help in decreasing the search space, and may be a suitable direction for optimizing this study's CP solution. Metivier et al. (2009) also mention that global constraints traditionally do not intercommunicate, such that findings from one global constraint do not affect filtering in the search space of following global constraints. They go on to propose a solution that has communication between global constraints that mostly performs better than the separate filtering solution, for small to medium NRP instances. The introduction of heuristics may, however, limit the results to near-optimal results considering the use of heuristics limits the approach from being an exact method.

Combining two mutually redundant viewpoints (viewpoints that represent the same problem) may also produce beneficial results (Smith, 2006). While changes to the model may improve its runtime or its accuracy. Something Smith (2006) points out is that changes may not always reduce search time and instead urges to actually perform empirical tests to ensure the results are satisfactory.

In the study by Yuraszeck et al. (2023) there appears to be a non-linear relationship in the increase in time between different instances. There seems to be an implication that depending on the complexity of the structure of an instance, there may be a significant increase in time to find a solution, when comparing to similar instances with a slight increase in one parameter. There are, however, instances with smaller amount of combinations that may still be slower than similar problems with a greater amount of combinations. Implying that the increase in time may not always be the case in some edge cases. The same trend is shown in results of Metivier et al. (2009), in the filtering comparison, with more complex instances of NRP generally taking significantly longer time. The findings in the CP solution for this study seems to echo the idea that increased model complexity increases the execution time significantly.

## 6.5 Comparison Discussion

When deciding to compare an exact and approximate solution, the outcome was never in doubt. Exact solutions are designed to find the optimal solution; therefore, given sufficient time, the result achieved would always be optimal. The drawback of exact solutions lies in their potential time consumption, especially in larger instances. MH approaches solve this drawback as they draw inspiration from various scientific principles to shorten the search process and achieve results faster. However, the approximate approach cannot guarantee an optimal solution, only a near-optimal one. Hence, the expected outcome was that the CP solution would be accurate but slow, while the GA solution would be semi-accurate but fast.

These expectations are not unwarranted, as many studies have conducted comparisons between exact and approximate solutions or have utilized both to solve problems of varying sizes. For instance, Fattahi et al. (2009) compared a hybrid algorithm using Simulated Annealing with the traditional optimization technique, branch and bound. They found that the exact approach could provide an optimum result with small instances but took too long to complete on larger instances. In contrast, the MH approach found near-optimal to optimal results but did so slower than the exact approach on very small instances while faster on medium-sized ones. Similarly, Jalilvand-Nejad and Fattahi (2015) utilized two different approaches to solve the cyclic flexible job shop scheduling problem: MILP model for smaller instances and GA for larger instances, stating the necessity of using an approximate approach for real-size problems due to the NP-hard class of the problem. Additionally, Woo and Kim (2018) mentioned that the MILP model is not a reasonable approach for large-sized problems to be solved within a reasonable time frame.

Therefore, they used the MILP model for small instances while GA and SA were employed for larger ones when solving a parallel machine scheduling problem.

As demonstrated in the results, this assertion could not be fully confirmed. The CP solution was incomplete and unable to perform a full run within a reasonable time when all but one constraint was enabled. Although the GA solution could provide somewhat accurate results within a reasonable time frame, its results were inconsistent. This suggests that the results, particularly regarding the execution time, were in line with expectations. However, these results do not provide a fair comparison between an exact and approximate solution, as neither of these solutions are in a complete or reliable state.

## 6.6 Threats to Validity

### Construct Validity

Other dependent variables may be relevant to the study, i.e. solution specific ones such as; the number of branches searched or nodes visited in CP. However, a choice was made to keep a small amount of comparable dependent variables, such that simple comparisons could be made between the two approaches.

The presented solutions may create unrepresentative results of either approach (Genetic Algorithm vs Constraint Programming). That is to say that the solutions do not necessarily describe whether one is more performant than the other, in the given problem area, but rather if our implementation of them is. Someone with more experience in the given approaches may find more optimized ways of developing better solutions to produce better results.

Considering the wider problem area, the size of the dataset is significantly smaller than what a real-world problem would be, which may affect the resulting conclusions. A study with a larger dataset may have provided better insights into the performance of solutions but would introduce further problems with long execution times. Another approach would be to have many different similar datasets that can statistically represent the wider problem area, to gather significant averages for both the problem area and the solutions' results.

### Conclusion Validity

A larger sample size of baseline values would have been preferable to draw reasonable conclusions regarding the problem area. Currently, the baseline is only representative of a single instance. Therefore, drawing statistical conclusions related to the solutions' usability in a wider area becomes limited.

The derived execution time for the baseline may not accurately represent the completion time for scheduling personnel to produce a schedule of 50 jobs. Several factors can influence the scheduling speed and how efficiently a human handles the workload. Variations in the workload, such as the number of batch groups available for each job and the quantity of jobs within each batch group, can significantly impact scheduling efficiency. However, it is challenging to establish a reliable metric without further investigation into the specific operational context. Therefore, a linear simplification has been employed, albeit with the caveat that it may not fully capture the complexities of scheduling in practice.

A stable testing environment could not be attained in time, for this study. Many attempts were made but failed due to time constraints, hardware issues, and unreasonable execution times. Instead, the tests were made on two different game-based, personal computers, without any proper way to isolate the tests' execution. The results, more specifically: the execution

times, should not be considered as a general reproducible result. Instead, the results are analyzed comparatively, since the used computers are similar in setup (with reservation for minor differences such as unconsidered sub-processes).

Another aspect to consider is that the libraries themselves may or may not be representative of either approach. The use of open-source libraries, in this case OR-Tools (Google, 2023) and GeneticSharp (Giacomelli, 2023b), may limit the study. There is a possibility that commercially viable tools that are considered closer to an industry standard for each approach. Furthermore, the implementation of the solutions may contain flaws or not be optimal, leading to non-optimal results.

Insufficient testing on the GA implementation was conducted with various penalty calculations due to time constraints. This limitation potentially impacts the conclusiveness of the results regarding the accuracy of a GA implementation in solving the given problem. Additionally, there is potential for further optimization of the fitness evaluation process, which could result in improved execution time. Therefore, the question arises as to whether this implementation accurately reflects what GA is capable of achieving.

The presented CP solution exceeds reasonable execution time, since preliminary tests had to stop after 72h+ without any final result. The long execution times indicate that there is a need for optimizations in the modeling of the CP solutions. Considering a general lack of experience with CP, the solution may be subject to a handful of different faults. The solution may also have flaws at its core, one of which is its incomplete state (missing soft constraint). Although some minor tests have been made to confirm the different constraints functionality, it is hard to consider all options and combinations. As such, confirming whether it is the best possible implementation is difficult and would require more time spent on verification.

The size of the dataset contributes to a separate array of concerns. Considering the issue of execution time, it is very likely due to a combinatorial explosion when the CP solver works through a large number of possible solutions. That would mean that an increase in any aspect of the problem space would increase the execution time even further. Be it either more jobs, batch groups, rules, or any other addition. The more possible combinations, the longer the solver has to work. This increase in the workload depends in large part due to the nature of the soft constraints. The soft constraints require the solver to consider more possibilities, as they cannot easily be pruned out (compared to the hard constraints). The solver may have to consider variables with unfavorable values, in an effort to hopefully find a combination with a lower objective value.

**Internal Validity**

Due to the time constraints placed on this study, a stable test environment was not successfully acquired. Consequently, the tests had to be conducted on different computers at varying loads. As a result, the results gathered from these tests may not fully represent the problem area, as the computing power could not be isolated to a specific process. While the results may lack precision, they still provide insight into the general performance of the resulting implementations and offer ideas for potential improvements.

The different libraries used for the GA implementation and CP implementation may differ in their respective efficiency. It may be hard to determine if the results gathered by these libraries represent the efficiency between the libraries rather than the different paradigms. As such, it may be difficult to provide a definitive answer regarding how any GA solution compares against any CP solution in the general problem area. Instead, this study leans towards comparing how these two specific presented solutions, together with their respective libraries, fare against each other. Another consideration when using libraries is that they may present invisible or not-yet-

considered side effects. Mitigation of these side effects can be achieved by a deep understanding of how the different libraries work and how to use them optimally.

The use of only open-source libraries for the different implementations may run the risk of not being representative of the wider set of tools available, including commercial tools. It may prove hard to present the findings of either solution as representative of its paradigm, especially considering the vast array of tools that are not tested in this study. Further investigation into this may find an optimal tool for the job.

The use of the stagnation level introduces instability in the execution times of the GA. This instability can lead to execution times that are more than three times longer for certain tests. Due to the randomness inherent in the algorithm, predicting these fluctuations is impossible. Nevertheless, stagnation plays a crucial role in the solution by increasing the likelihood of reaching the global optimum.

Due to inexperience and time constraints, the presented CP solution may include flaws. These flaws may affect the results and need to be considered when analyzed. A clear problem is the execution time that exceeds reasonable limits. Steps should be taken to simplify the model to alleviate the computing needed to produce a schedule. However, reaching this conclusion may have required a more naive solution at first (such as the one in this study), before seeing where the optimizations could be done.

**External Validity**

The dataset utilized in this study may not adequately represent the general problem area due to its small size. With only 50 jobs included, compared to the 300 to 500 jobs typically found in real-world scenarios, producing an effective solution at this scale may prove notably more challenging, given the combinatorial nature of the problem. Despite its limited size, the dataset offers insights into potential approaches. Furthermore, while there is a need for future expansion to encompass a larger dataset, the current sample size is sufficient to provide an initial understanding of potential strategies. Additionally, the analysis of the current results identifies areas for improvement in each solution and highlights the weaknesses present in the given solutions.

The rules applied to the problem are limited in number, and may not necessarily be diverse enough to be generalizable for all scheduling problems. However, they do illustrate how conflicting constraints can arise during a scheduling activity. To ensure that a solution can handle conflicting constraints, a certain degree of flexibility needs to be incorporated into the solution.

# 7 | Conclusions

Two approaches have been evaluated to address a multi-constraint job order balancing scheduling problem: an exact approach using the CP model, and an approximate approach using the GA. These approaches were initially tested individually, where their accuracy, measured in terms of accuracy score and execution time, was assessed. The solutions were compared against a baseline schedule and estimated completion time to evaluate their practical utility, followed by a comparison between the solutions.

The results show that, in the solutions' current state, the GA solution outperforms the CP solution in terms of both accuracy score and execution time. The GA solution consistently achieves more accurate results and finds them faster compared to the CP solution. While the GA solution occasionally surpasses the baseline in terms of accuracy score, these improvements are not consistently replicated. The tests do, however, show that the GA on average produces a schedule comparable to the baseline completion time. Despite its inconsistencies, the GA solution shows promise and is closer to being a viable solution compared to the CP solution. With that said, the CP solution has its merits, it may become a viable solution with additional time and effort dedicated to it. The groundwork for the model is set but requires optimizations and simplifications of the applied constraints to reduce the search space for the solver.

The code used in this study has been published in a public Github repository which can be found in Appendix C.

### Future Work

The current GA implementation has flaws, particularly linked to its penalty system. Future work would, therefore, focus on improving this system to make the algorithm more reliable. Additionally, the algorithm currently employs only a single selection operation. However, it may be possible to enhance the selection process by incorporating Elitist selection on top of the Tournament selection. Exploring this approach could yield better results. The GeneticSharp library also offers additional operations for crossover and mutation that support permutation. Unfortunately, there was not enough time to test these functionalities, but they could be explored in future work.

During the tests, it was observed that the solution appeared to suffer from premature convergence. Hence, exploring the Cellular Genetic Algorithm used by Zhang et al. (2020) would be interesting, as they claim it is designed to maintain diversity among the population for as long as possible. As mentioned during the discussion, the use of the stagnation level was presumed to lead to early termination, because of premature convergence. Hence, it would be interesting to explore alternative stopping criteria or even combinations of several criteria, as this is also possible.

Aside from finishing the implementation with the missing soft constraint, the CP solution would benefit from testing which constraints affect the execution time and accuracy the most.

Some of the results in this study imply that the *cooldown* constraint increases the execution time beyond reasonable limits, but its inclusion in the model also seems to lead to better fitness values from the resulting schedules. Principal Component Analysis (PCA) may offer a way to *"identify latent variables in large datasets that are represented by highly correlated input variables"* (Boslaugh, 2012). PCA may be used to find the most important variables in the CP model, to then see how their modifications affect the end result. Testing different representations for the decision variables may also improve the performance in execution time, such as the Manne-based representation (Naderi et al., 2023).

Comparing different tools, both commercial and open-source, would be an intriguing avenue to explore as it may offer insights into the most effective tools for each approach in addressing the given problem.

**Special Thanks**

Special thanks to *Volvo Group Digital & IT* for their support and help in this study.

# Bibliography


2023. Sustainable Development Goal 7 (SDG7) | Sustainable Energy for All. https://www.seforall.org/our-work/sustainable-development-goal-7-sdg7 accessed on 2024-03-28.

Beatriz Andres, Raquel Sanchis, and Raúl Poler. 2016. A Cloud Platform to support Collaboration in Supply Networks. *International Journal of Production Management and Engineering* 4, 1 (Jan. 2016), 5–13. https://doi.org/10.4995/ijpme.2016.4418

Christian Bessiere. 2006. Constraint Propagation. In *Handbook of Constraint Programming*, Francesca Rossi, Peter Van Beek, and Toby Walsh (Eds.). Elsevier, The Netherlands, Amsterdam, Chapter 3, 29–84.

Sarah Boslaugh. 2012. *Statistics in a nutshell: A desktop quick reference.* " O'Reilly Media, Inc.".

Nils Boysen, Philipp Schulze, and Armin Scholl. 2022. Assembly line balancing: What happened in the last fifteen years? *European Journal of Operational Research* 301, 3 (2022 SEP 16 2022), 797–814. https://doi.org/10.1016/j.ejor.2021.11.043

Edmund K. Burke, Timothy Curtois, Gerhard Post, Rong Qu, and Bart Veltman. 2008. A hybrid heuristic ordering and variable neighbourhood search for the nurse rostering problem. *European Journal of Operational Research* 188, 2 (2008), 330–341. https://doi.org/10.1016/j.ejor.2007.04.030

Thomas Bäck. 1996. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms.* Oxford University Press. https://doi.org/10.1093/oso/9780195099713.001.0001

Thomas Bäck, David B. Fogel, and Z. Michalewicz. 2000. *Evolutionary computation.* Institute of Physics Publishing.

Márton Drótos, Gábor Erdős, and Tamás Kis. 2009. Computing lower and upper bounds for a large-scale industrial job shop scheduling problem. *European Journal of Operational Research* 197, 1 (2009), 296–306. https://doi.org/10.1016/j.ejor.2008.06.004

Parviz Fattahi, Fariborz Jolai, and Jamal Arkat. 2009. Flexible job shop scheduling with overlapping in operations. *Applied Mathematical Modelling* 33, 7 (2009), 3076–3087. https://doi.org/10.1016/j.apm.2008.10.029

Eugene C. Freuder and Alan K. Mackworth. 2006. Constraint Satisfaction: An Emerging Paradigm. In *Handbook of Constraint Programming*, Francesca Rossi, Peter Van Beek, and Toby Walsh (Eds.). Elsevier, The Netherlands, Amsterdam, Chapter 2, 13–28.

Kaizhou Gao, Zhiguang Cao, Le Zhang, Zhenghua Chen, Yuyan Han, and Quanke Pan. 2019. A Review on Swarm Intelligence and Evolutionary Algorithms for Solving Flexible Job Shop Scheduling Problems. *IEEE/CAA Journal of Automatica Sinica* 6, 4 (JUL 2019), 904–916. https://doi.org/10.1109/JAS.2019.1911540

Antonio Garrido, Marlene Arangu, and Eva Onaindia. 2009. A constraint programming formulation for planning: from plan scheduling to plan generation. *Journal of Scheduling* 12, 3 (JUN 2009), 227–256. https://doi.org/10.1007/s10951-008-0083-7

Ian P. Gent, Karen E. Petrie, and Jean-François Puget. 2006. Symmetry in Constraint Programming. In *Handbook of Constraint Programming*, Francesca Rossi, Peter Van Beek, and Toby Walsh (Eds.). Elsevier, The Netherlands, Amsterdam, Chapter 10, 329–376.

Farhad Soleimanian Gharehchopogh. 2023. Quantum-inspired metaheuristic algorithms: comprehensive survey and classification. *Artificial Intelligence Review* 56, 6 (JUN 2023), 5479–5543. https://doi.org/10.1007/s10462-022-10280-8

Diego Giacomelli. 2023a. GeneticSharp/src/GeneticSharp.Domain/Crossovers/OrderBasedCrossover.cs at master · giacomelli/GeneticSharp · GitHub. https://github.com/giacomelli/GeneticSharp/blob/master/src/GeneticSharp.Domain/Crossovers/OrderBasedCrossover.cs accessed on 2024-03-12.

Diego Giacomelli. 2023b. GitHub - giacomelli/GeneticSharp: GeneticSharp is a fast, extensible, multi-platform and multithreading C Genetic Algorithm library that simplifies the development of applications using Genetic Algorithms (GAs). https://github.com/giacomelli/GeneticSharp/tree/master?tab=readme-ov-file accessed on 2024-03-12.

Google. 2023. OR-Tools - Google Optimization Tools. https://github.com/google/or-tools accessed on 2024-03-29.

Tias Guns, Siegfried Nijssen, and Luc De Raedt. 2011. Itemset mining: A constraint programming perspective. *Artificial Intelligence* 175, 12 (2011), 1951–1983. https://doi.org/10.1016/j.artint.2011.05.002

Eduardo Guzman, Beatriz Andres, and Raul Poler. 2022. Models and algorithms for production planning, scheduling and sequencing problems: A holistic framework and a systematic review. *Journal of Industrial Information Integration* 27 (2022). https://doi.org/10.1016/j.jii.2021.100287 Cited by: 20.

Julia Heil, Kirsten Hoffmann, and Udo Buscher. 2020. Railway crew scheduling: Models, methods and applications. *European Journal of Operational Research* 283, 2 (JUN 1 2020), 405–425. https://doi.org/10.1016/j.ejor.2019.06.016

E. Horowitz and S. Sahni. 1978. *Fundamentals of Computer Algorithms*. Pitman. https://books.google.se/books?id=n8c8PgAACAAJ

Vincenzo Iannino, Valentina Colla, Joachim Denker, and Marc Göttsche. 2019. A CPS-Based Simulation Platform for Long Production Factories. *Metals* 9, 10 (2019). https://doi.org/10.3390/met9101025

Amir Jalilvand-Nejad and Parviz Fattahi. 2015. A mathematical model and genetic algorithm to cyclic flexible job shop scheduling problem. *Journal of Intelligent Manufacturing* 26, 6 (2015), 1085 – 1098. https://doi.org/10.1007/s10845-013-0841-z

Daniel Kahneman. 2017. *Thinking, fast and slow.*

S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. 1983. Optimization by Simulated Annealing. *Science* 220, 4598 (1983), 671–680. https://doi.org/10.1126/science.220.4598.671 arXiv:https://www.science.org/doi/pdf/10.1126/science.220.4598.671

Anton J Kleywegt and Jason D Papastavrou. 1998. The dynamic and stochastic knapsack problem. *Operations research* 46, 1 (1998), 17–35.

Daria Leiber, David Eickholt, Anh-Tu Vuong, and Gunther Reinhart. 2022. Simulation-based layout optimization for multi-station assembly lines. *Journal of Intelligent Manufacturing* 33, 2 (2022), 537 – 554. https://doi.org/10.1007/s10845-021-01853-5

Arnaud Malapert, Hadrien Cambazard, Christelle Guéret, Narendra Jussien, André Langevin, and Louis-Martin Rousseau. 2012. An optimal constraint programming approach to the open-shop problem. *INFORMS Journal on Computing* 24, 2 (2012), 228–244.

Pedro Meseguer, Francesca Rossi, and Thomas Schiex. 2006. Soft Constraints. In *Handbook of Constraint Programming*, Francesca Rossi, Peter Van Beek, and Toby Walsh (Eds.). Elsevier, The Netherlands, Amsterdam, Chapter 9, 281–328.

Jean-Philippe Metivier, Patrice Boizumault, and Samir Loudni. 2009. Solving Nurse Rostering Problems Using Soft Global Constraints. In *Principles and Practice of Constraint Programming (Lecture Notes in Computer Science, Vol. 5732)*, IP Gent (Ed.). Assoc Constraint Programming; Natl Informat & Commun Technol Australia; Fdn Sci & Technol; Ctr Artificial Intelligence; Portuguese Assoc Artificial Intelligence, 73–87. 15th International Conference on Principles and Practice of Constraint Programming (CP 2009), Lisbon, PORTUGAL, SEP 20-24, 2009.

Bahman Naderi, Rubén Ruiz, and Vahid Roshanaei. 2023. Mixed-integer programming vs. constraint programming for shop scheduling problems: new results and outlook. *INFORMS Journal on Computing* 35, 4 (2023), 817–843.

RA Russell and TL Urban. 2006. A constraint programming approach to the multiple-venue, sport-scheduling problem. *Computers Operations Research* 33, 7 (JUL 2006), 1895–1906. https://doi.org/10.1016/j.cor.2004.09.029

Barbara M. Smith. 2006. Modelling. In *Handbook of Constraint Programming*, Francesca Rossi, Peter Van Beek, and Toby Walsh (Eds.). Elsevier, The Netherlands, Amsterdam, Chapter 11, 377–406.

Yuri N. N. Sotskov. 2023. Assembly and Production Line Designing, Balancing and Scheduling with Inaccurate Data: A Survey and Perspectives. *Algorithms* 16, 2 (FEB 2023). https://doi.org/10.3390/a16020100

Éric D Taillard. 2023. *Design of Heuristic Algorithms for Hard Optimization With Python Codes for the Travelling Salesman Problem.* Springer.

El-Ghazali Talbi. 2009. *Metaheuristics: from design to implementation.* John Wiley & Sons.

Jorne Van den Bergh, Jeroen Belien, Philippe De Bruecker, Erik Demeulemeester, and Liesje De Boeck. 2013. Personnel scheduling: A literature review. *European Journal of Operational Research* 226, 3 (MAY 1 2013), 367–385. https://doi.org/10.1016/j.ejor.2012.11.029

Willem-Jan van Hoeve and Irit Katriel. 2006. Global Constraints. In *Handbook of Constraint Programming*, Francesca Rossi, Peter Van Beek, and Toby Walsh (Eds.). Elsevier, The Netherlands, Amsterdam, Chapter 6, 169–208.

Rico Walter, Philipp Schulze, and Armin Scholl. 2021. SALSA: Combining branch-and-bound with dynamic programming to smoothen workloads in simple assembly line balancing. *European Journal of Operational Research* 295, 3 (2021), 857–873. https://doi.org/10.1016/j.ejor.2021.03.021

Ezra Wari and Weihang Zhu. 2016. A survey on metaheuristics for optimization in food manufacturing industry. *Applied Soft Computing* 46 (2016), 328–343. https://doi.org/10.1016/j.asoc.2016.04.034

Georges Weil, Kamel Heus, Patrice Francois, and Marc Poujade. 1995. Constraint programming for nurse scheduling. *IEEE Engineering in medicine and biology magazine* 14, 4 (1995), 417–422.

Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering.* Springer Science & Business Media.

Young-Bin Woo and Byung Soo Kim. 2018. Matheuristic approaches for parallel machine scheduling problem with time-dependent deterioration and multiple rate-modifying activities. *Computers Operations Research* 95 (2018), 97–112. https://doi.org/10.1016/j.cor.2018.02.017

Francisco Yuraszeck, Elizabeth Montero, Darío Canut-De-Bon, Nicolás Cuneo, and Maximiliano Rojel. 2023. A Constraint Programming Formulation of the Multi-Mode Resource-Constrained Project Scheduling Problem for the Flexible Job Shop Scheduling Problem. *IEEE Access* 11 (2023), 144928–144938. https://doi.org/10.1109/ACCESS.2023.3345793

Beikun Zhang, Liyun Xu, and Jian Zhang. 2020. A multi-objective cellular genetic algorithm for energy-oriented balancing and sequencing problem of mixed-model assembly line. *Journal of Cleaner Production* 244 (Jan 20 2020). https://doi.org/10.1016/j.jclepro.2019.118845

# A | Appendix A

## A.1 Configuration - tests and conclusions

The following tests were conducted to determine the configuration of the selected operations. The tests were performed on Computer A, with the primary goal of these tests being to minimize the accuracy score, aiming for it to be as close to zero as possible, while also striving to achieve the lowest possible execution time.

The initial settings tested were population size and mutation rates, while all other settings were kept at their default values: crossover rate = 0.75 and stagnation = 100. Population sizes of 500, 1000, 2500, 5000, 7500, and 10000 were chosen for testing. Mutation rates selected for testing were 0.05, 0.1, 0.15, 0.25, 0.35, 0.45, 0.55, 0.65, 0.75, 0.85, and 0.95. Each combination of these settings was tested 30 times. However, tests with the largest population sizes, 7500 and 10000, were too time-consuming to perform. Therefore, only a subset of mutation rates covering these sizes was tested: 0.1, 0.25, 0.35, and 0.45.

### A.1.1 Population size

Even though the tests were limited, a clear trend can be observed, as seen in Figures A.2 and A.1. With population sizes over 5000, only marginal improvements to the mean score can be seen. However, no matter the chosen mutation rate, the cost in execution time almost doubles after an increase of approximately 2000 individuals. With a population size of 10000, the execution time is almost three times as long. Even though tests were conducted on the other population sizes, following these observations, a population size of 5000 individuals was selected as the foundation for further analysis and testing.

### A.1.2 Mutation rate

The inherent randomness of GA complicates a detailed analysis of the mutation rate as the collected data with the population size 5000, is riddled with numerous outliers and exhibits a wide dispersion among data points, as can be seen in Figures A.3 and A.4, which plot accuracy and execution time respectively. The accuracy diagram suggests a trend of higher scores with increasing mutation rates, yet this trend is inconsistent. For instance, the plots for 0.25 and 0.55 display similar results, while 0.35, 0.45, and 0.65 yields worse outcomes in general, this does not follow the trend pattern. Similarly, the execution time diagram hints at longer execution times with higher mutation rates, although the results are highly inconsistent. In this scenario, it can be observed that the mutation rate of 0.85 stands out with an execution time almost double that of either 0.75 or 0.95, with the latter having a much lower mean. Table A.1 corroborates the results and provides a more descriptive view of the data. What can be seen, is that both the

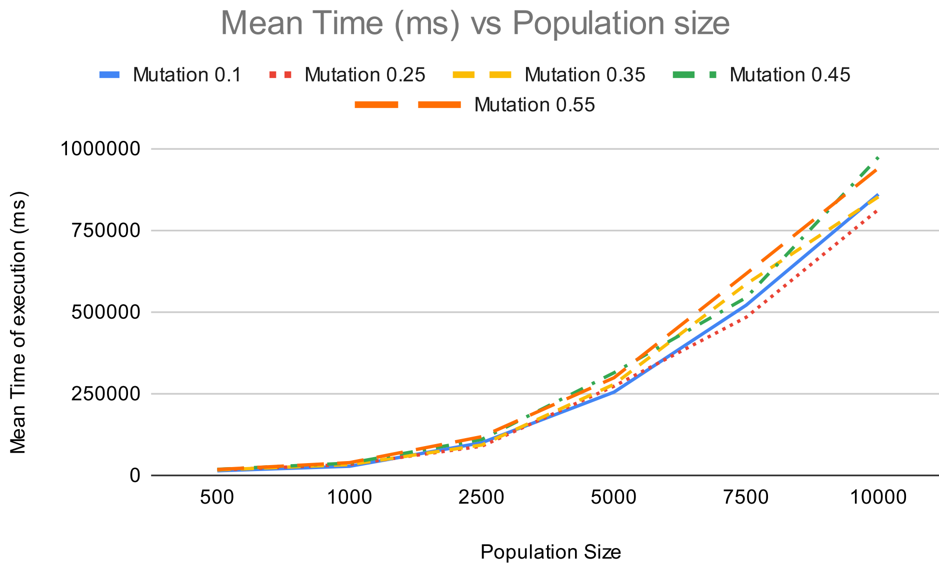


Figure A.1: PilotLargeMeanTimeVSPopulation

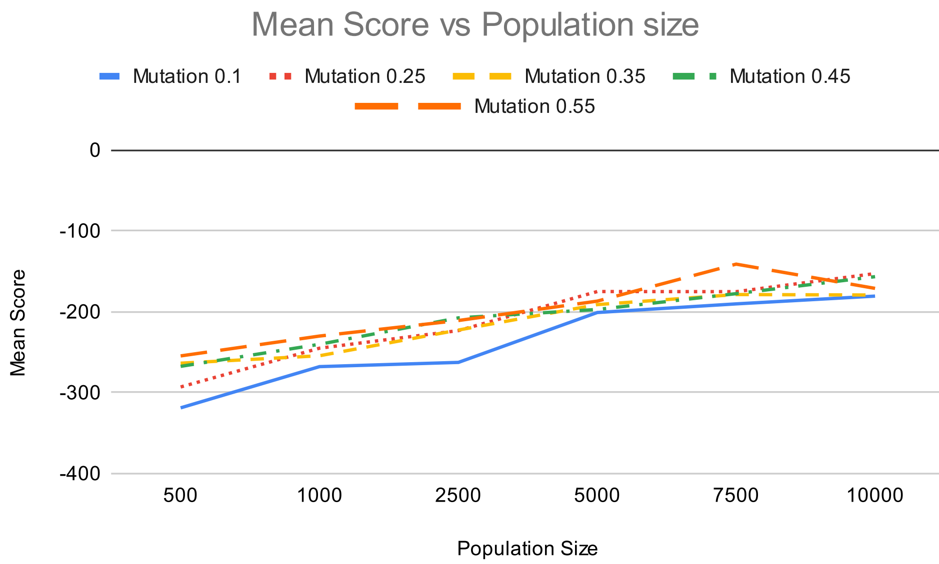


Figure A.2: PilotLargeMeanScoreVSPopulation

Table A.1: Descriptive data of test run with a population of 5000 related to the score and time.

| Mutation rate | Mean score | Score Std | Score S.E. | Mean Time (ms) | Time Std (ms) | Time S.E. (ms) |
|---|---|---|---|---|---|---|
| 0.05 | -227.04 | 85.40 | 12.59 | 254699.4 | 98964.98 | 18068.45 |
| 0.10 | -200.83 | 68.83 | 12.57 | 254602.6 | 78802.24 | 14387.25 |
| 0.15 | -210.46 | 68.29 | 12.47 | 278067.1 | 83733.32 | 15287.54 |
| 0.25 | -174.89 | 68.34 | 12.48 | 272744.6 | 89038.25 | 16256.09 |
| 0.35 | -191.05 | 59.95 | 10.94 | 278502.9 | 72025.79 | 13150.05 |
| 0.45 | -197.05 | 61.91 | 11.30 | 313945.8 | 90034.02 | 16437.89 |
| 0.55 | -186.73 | 68.49 | 12.50 | 298544.2 | 89321.93 | 16307.88 |
| 0.65 | -209.09 | 77.84 | 14.21 | 267864.2 | 62322.11 | 11378.41 |
| 0.75 | -173.60 | 63.87 | 11.66 | 316796.4 | 114204.20 | 20850.74 |
| 0.85 | -170.65 | 58.44 | 10.67 | 411809.2 | 150641.51 | 27503.25 |
| 0.95 | -154.38 | 59.25 | 10.82 | 353844.3 | 81116.10 | 14809.71 |

Table A.2: The average of all gathered values for population sizes of 500, 1000, 2500, and 5000, categorized by the mutation rate.

| Mutation | Mean Score | Score Std | Score S.E. | Mean Time | Time Std (ms) | Time S.E. (ms) |
|---|---|---|---|---|---|---|
| 0.05 | -276.99 | 92.77 | 8.47 | 98663.41 | 108319.44 | 9888.17 |
| 0.10 | -262.46 | 93.66 | 8.55 | 99051.88 | 104619.10 | 9550.37 |
| 0.15 | -247.33 | 72.49 | 6.62 | 104206.51 | 113640.43 | 10373.90 |
| 0.25 | -233.99 | 82.26 | 7.51 | 102924.61 | 112044.04 | 10228.17 |
| 0.35 | -232.84 | 74.72 | 6.82 | 105232.59 | 111261.64 | 10156.75 |
| 0.45 | -228.01 | 74.17 | 6.77 | 119931.02 | 127064.55 | 11599.35 |
| 0.55 | -220.48 | 79.35 | 7.24 | 118423.81 | 120768.16 | 11024.57 |
| 0.65 | -223.73 | 78.19 | 7.14 | 111190.03 | 105676.24 | 9646.88 |
| 0.75 | -209.09 | 76.64 | 7.00 | 120848.09 | 133446.41 | 12181.93 |
| 0.85 | -204.32 | 82.57 | 7.54 | 147844.13 | 174839.79 | 15960.61 |
| 0.95 | -194.31 | 72.44 | 6.61 | 138822.08 | 140223.08 | 12800.56 |

accuracy and execution time have a high distribution. Solely relying on the samples obtained from tests conducted with a population size of 5000 to determine the best mutation rate proved insufficient as the sample size was deemed too small to draw a meaningful conclusion.

Therefore, an alternative analysis was conducted by combining all recorded values for population sizes 500, 1000, 2500, and 5000, to calculate the overall mean. This larger sample size offered a more stable view of each mutation rate, making patterns easier to distinguish.

The trend remains the same, with higher mutation rates generally producing better scores as illustrated in Figures A.6 and A.5. The mutation rates of 0.75, 0.85, and 0.95 in particular yielded superior results, albeit with longer execution times. Consequently, 0.75 was chosen as a compromise, as it delivers among the highest scores while maintaining the lowest overall execution time among the top three performing mutation rates.

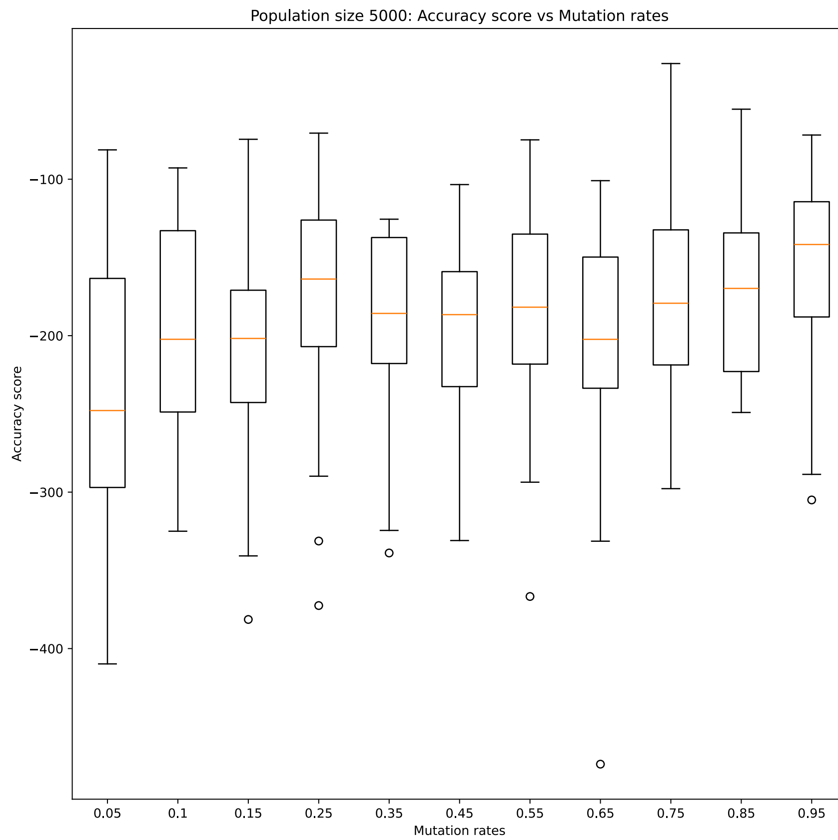


Figure A.3: Boxplot illustrating the score distribution across various mutation rates, using a population size of 5000.

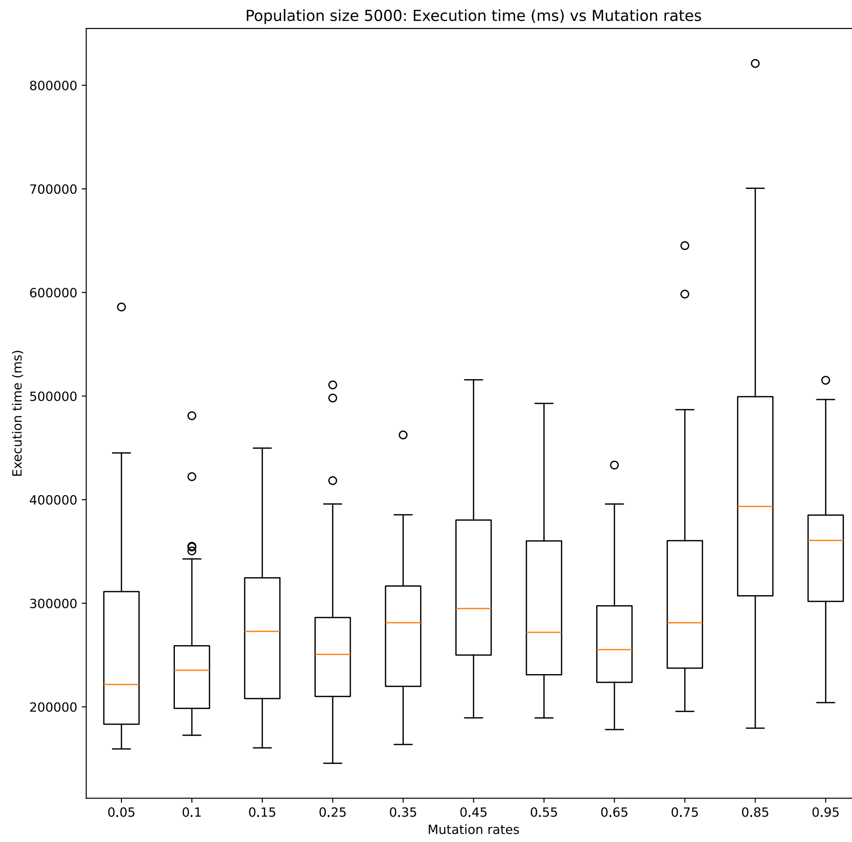


Figure A.4: Boxplot illustrating execution time distribution across various mutation rates, using a population size of 5000.

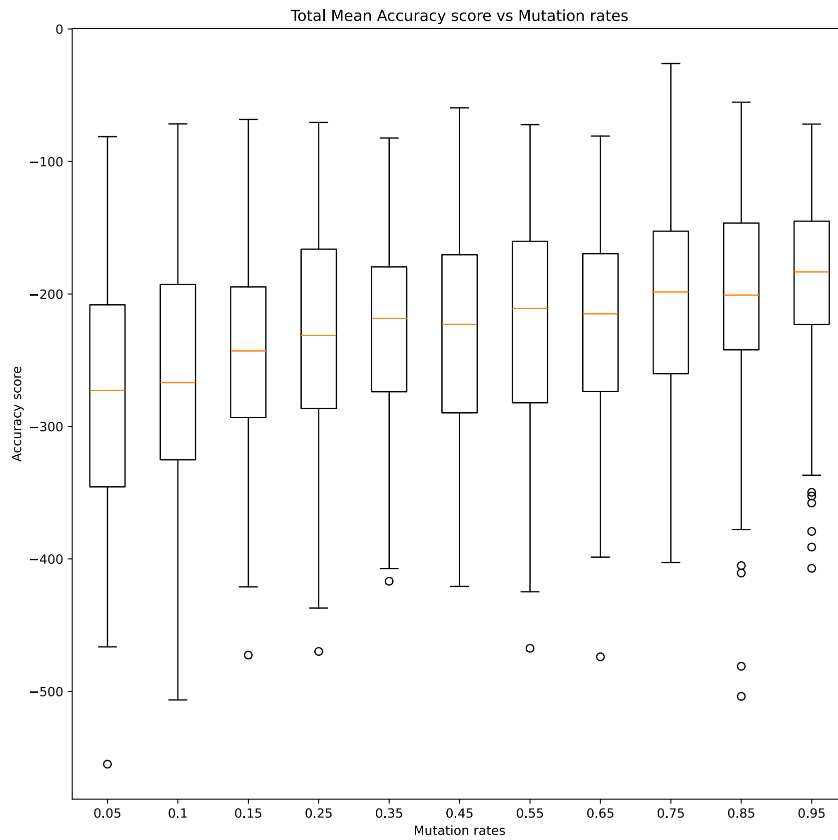


Figure A.5: Boxplot illustrating the score distribution across various mutation rates, using population sizes 500, 1000, 2500, and 5000.

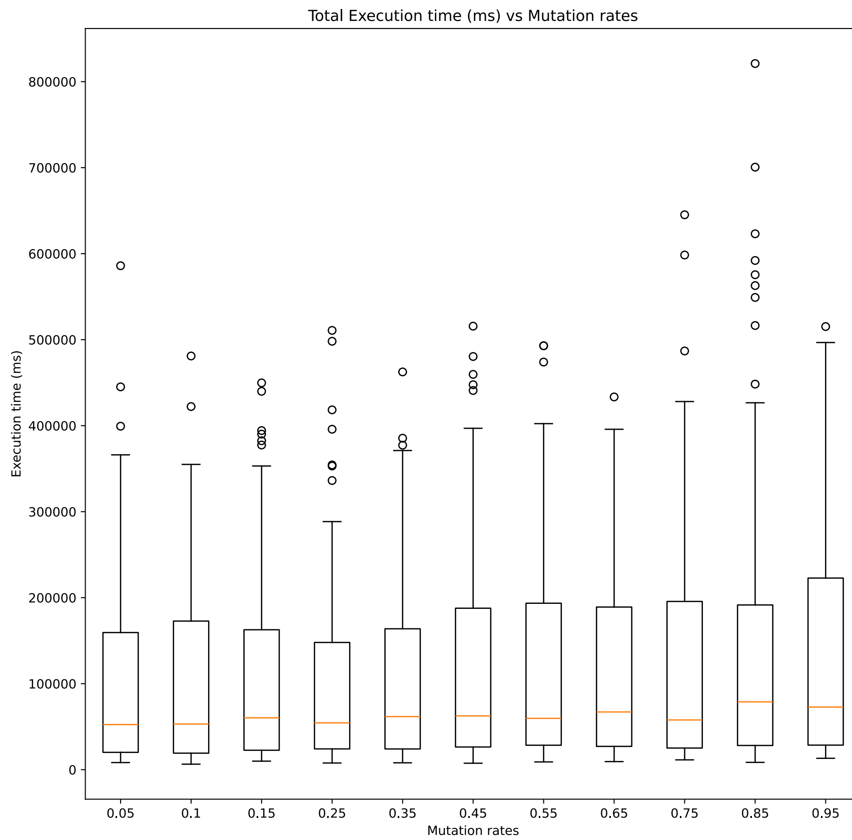


Figure A.6: Boxplot illustrating execution time distribution across various mutation rates, using population sizes 500, 1000, 2500, and 5000.

Table A.3: Summary of data from test runs conducted on crossover rates with a population size of 5000.

| Crossover rate | Mean Score | Score Std | Score S.E. | Mean time | Time Std (ms) | Time S.E. (ms) |
|---|---|---|---|---|---|---|
| 0.45 | -203.81 | 55.23 | 10.08 | 252006.47 | 94334.99 | 17223.13 |
| 0.50 | -173.64 | 48.33 | 8.82 | 287082.70 | 85134.19 | 15543.30 |
| 0.55 | -187.40 | 76.29 | 13.93 | 247031.20 | 57606.43 | 10517.45 |
| 0.60 | -169.41 | 63.93 | 11.67 | 259268.20 | 73256.38 | 13374.72 |
| 0.65 | -165.66 | 55.54 | 10.14 | 297558.60 | 96128.50 | 17550.58 |
| 0.70 | -153.82 | 65.16 | 11.90 | 294867.60 | 82383.96 | 15041.18 |
| 0.75 | -187.52 | 69.62 | 12.71 | 317667.90 | 72231.60 | 13187.62 |
| 0.80 | -180.15 | 62.73 | 11.45 | 314584.70 | 87140.02 | 15909.52 |
| 0.85 | -176.95 | 46.96 | 8.57 | 345729.20 | 94836.73 | 17314.74 |
| 0.90 | -191.17 | 56.70 | 10.35 | 386576.20 | 110982.84 | 20262.60 |
| 0.95 | -233.02 | 81.48 | 14.88 | 355414.30 | 109810.27 | 20048.52 |

### A.1.3 Crossover rate

The crossover tests focused on the range of 0.45 to 0.95, as this range was commonly referenced in the literature (Talbi, 2009; Bäck et al., 2000). The population size and mutation rate were set to the established values of 5000 individuals and 0.75, respectively. The stagnation rate remained at its default value of 100. Given that tests with a population size of 5000 are time-consuming, only 30 tests per rate were conducted.

As depicted in Figure A.7 and detailed in Table A.4, the most effective crossover rate in terms of accuracy is 0.70, with a mean score of approximately -154. Notably, there is a noticeable trend of declining accuracy as the crossover rate deviates further from 0.70 in either direction. However, the same trend does not carry over to execution time, as higher crossover rates generally result in longer execution times.

### A.1.4 Stagnation

Tests executed on the stagnation level maintain the same configuration as those performed on crossover rates. The only modification made is adjusting the crossover rate to 0.70. The tests were conducted with stagnation levels set at intervals of 50, up to stagnation level 400, with each level receiving 30 tests.

Although highly inconsistent, there appears to be a slight enhancement in the obtained score with higher stagnation levels, best illustrated by Figure A.9. However, these improvements are accompanied by significantly longer execution times on average. Unlike the previous tests that show inconsistency in execution time, it is evident with stagnation level that higher levels result in longer execution times, as illustrated in Figure A.10.

Due to the substantial increase in execution time associated with higher levels and the marginal score improvement relative to that, the default stagnation level of 100 was chosen. It offered enough of an improvement over the tested level 50 while keeping the execution time within reasonable bounds.

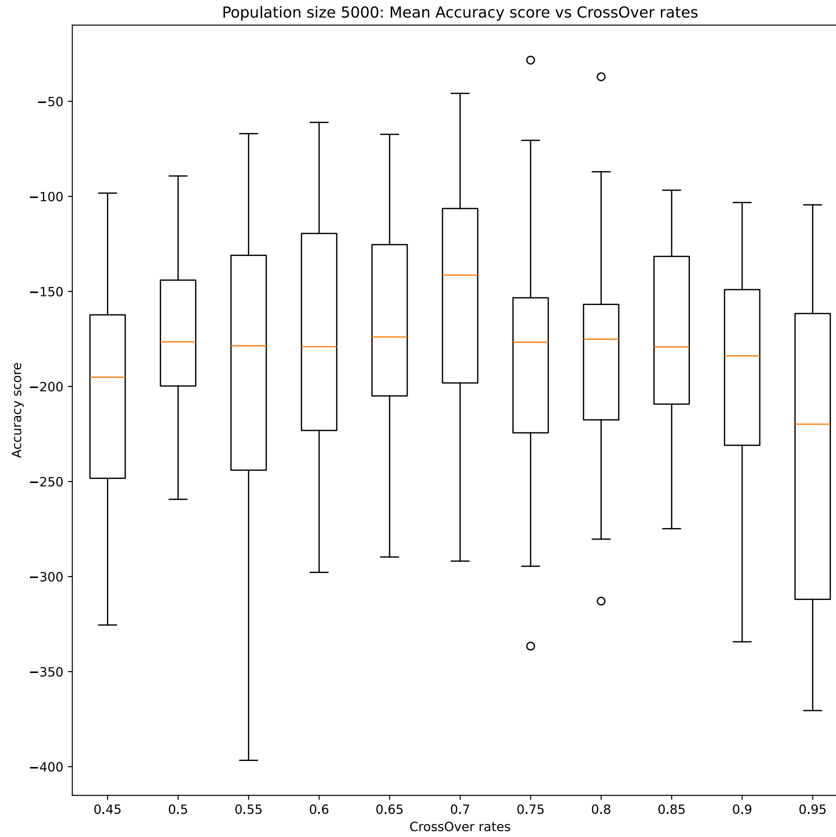


Figure A.7: Boxplot illustrating the distribution of accuracy score across different crossover rates, with a population size of 5000

Table A.4: Summary of data from test runs conducted on different stagnation levels, with a population size of 5000.

| **Stagnation level** | **Mean Score** | **Score Std** | **Score S.E.** | **Mean time** | **Time Std (ms)** | **Time S.E. (ms)** |
|---|---|---|---|---|---|---|
| 50 | -179.61 | 52.60 | 9.60 | 241805.40 | 56737.08 | 10358.73 |
| 100 | -164.34 | 52.31 | 9.55 | 320886.57 | 88085.49 | 16082.14 |
| 150 | -148.25 | 51.58 | 9.42 | 400008.27 | 152526.46 | 27847.39 |
| 200 | -175.10 | 63.12 | 11.52 | 519041.80 | 209128.11 | 38181.39 |
| 250 | -171.93 | 70.99 | 12.96 | 691974.77 | 290857.87 | 53103.14 |
| 300 | -156.13 | 51.79 | 9.46 | 723340.33 | 308208.43 | 56270.90 |
| 350 | -157.92 | 46.44 | 8.48 | 766333.97 | 233938.22 | 42711.08 |
| 400 | -154.17 | 54.09 | 9.87 | 862580.20 | 376178.41 | 68680.47 |

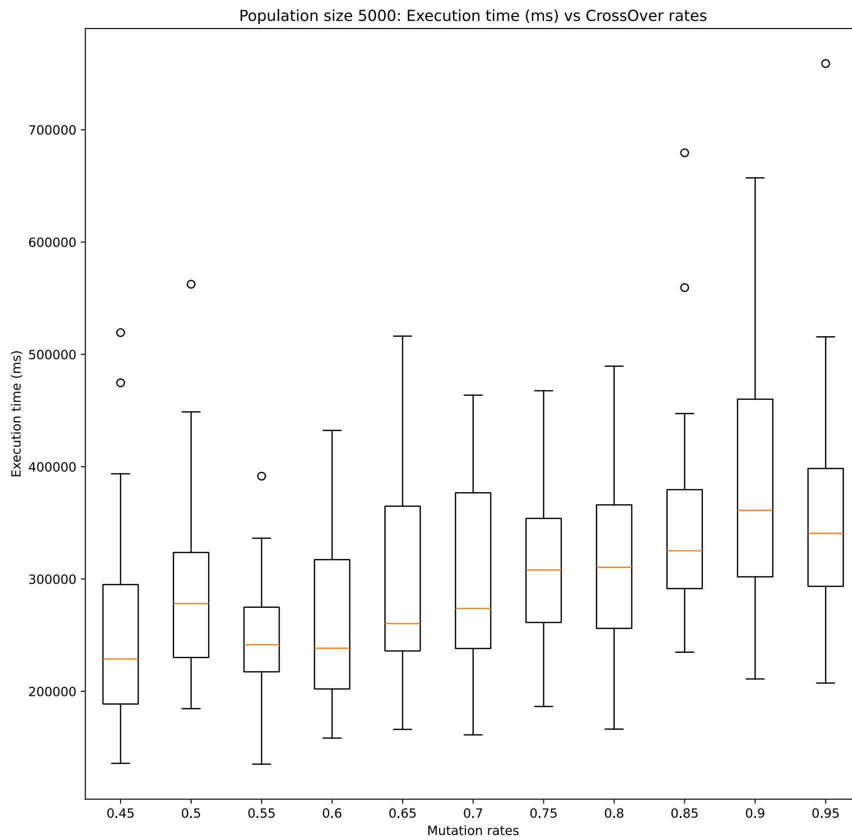


Figure A.8: Boxplot illustrating execution time distribution across different crossover rates, with a population size of 5000.

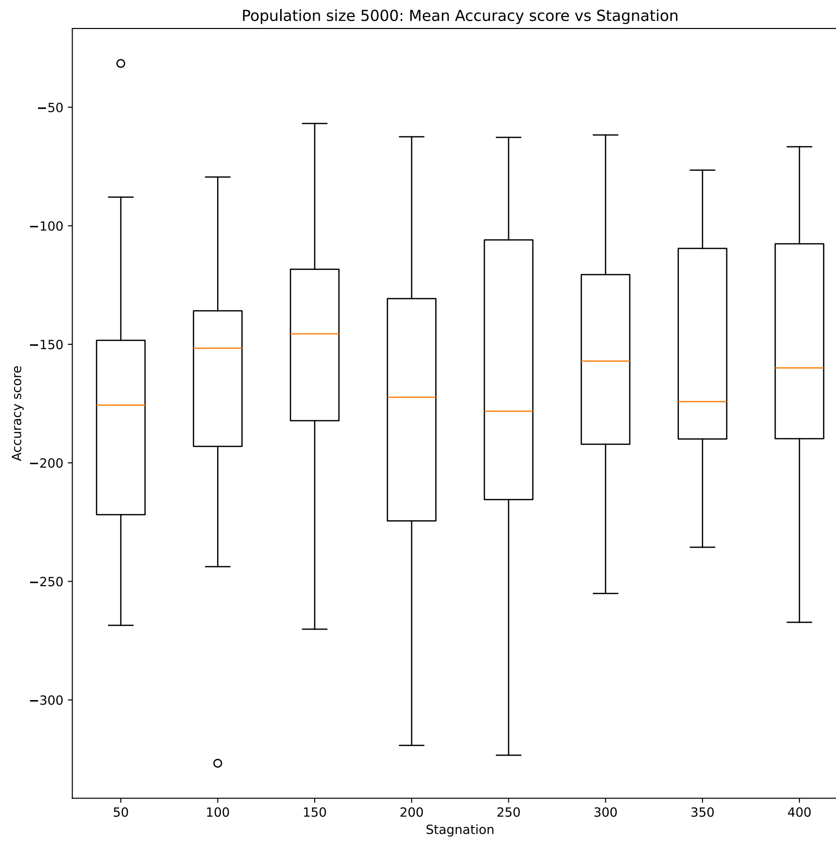


Figure A.9: Boxplot illustrating the distribution of accuracy score across different stagnation levels, with a population size of 5000

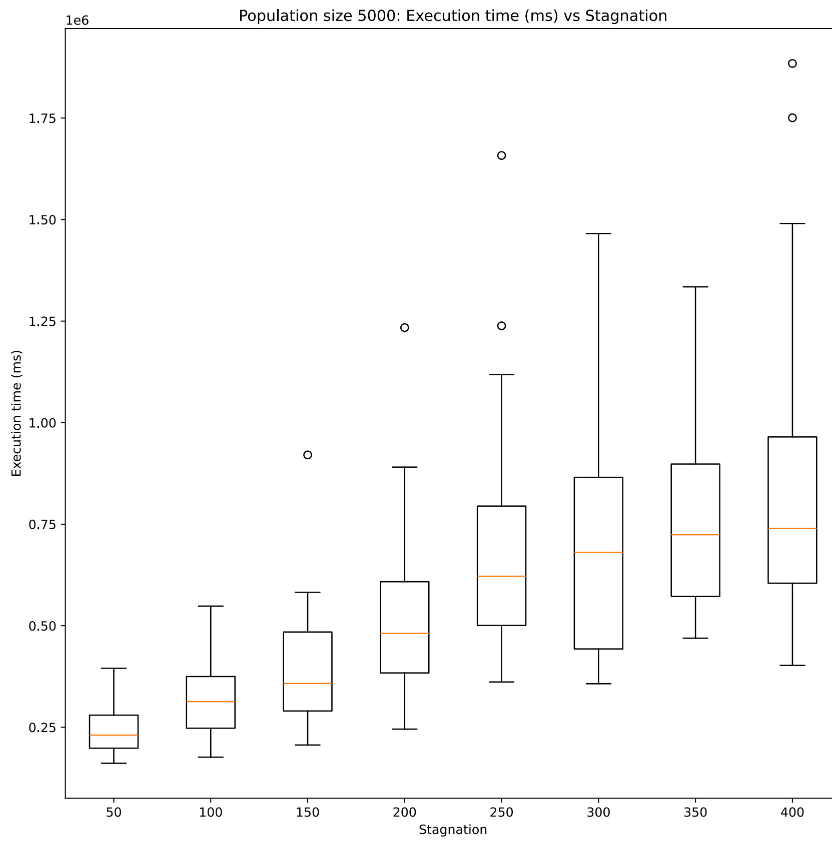


Figure A.10: Boxplot illustrating execution time distribution across different stagnation levels, with a population size of 5000.

# B | Appendix B

population size: 5000
GA running...
Generation 1: = -100018861,42846557
Generation 2: = -45504,7056857506
Generation 3: = -7554,196642437129
Generation 4: = -9087,66657733759
Generation 6: = -7280,4248321639025
Generation 7: = -4769,994903906064
Generation 8: = -4524,618694982855
Generation 9: = -4108,526610667166
Generation 10: = -3203,554216805285
Generation 11: = -2632,6749398590628
Generation 12: = -2698,236616713535
Generation 13: = -2798,992737579508
Generation 14: = -2347,1674408862373
Generation 15: = -2233,2692387099914
Generation 16: = -2090,4236473391315
Generation 17: = -1847,1299579389397
Generation 18: = -1794,0901754960003
Generation 19: = -1619,5111684965077
Generation 21: = -1465,457270762478
Generation 23: = -1450,2874553035576
Generation 24: = -1430,7022636139443
Generation 25: = -1347,5552752997478
Generation 26: = -1105,4466750580625
Generation 31: = -1073,7852818394201
Generation 33: = -1012,1436756373303
Generation 34: = -1010,4074791814762
Generation 35: = -966,9366754261957
Generation 36: = -914,6152820148814
Generation 37: = -894,5686837148028
Generation 39: = -864,7376841006385
Generation 40: = -888,7174841084704
Generation 41: = -871,9551795821221
Generation 42: = -661,0461839437962
Generation 48: = -648,0861832035066
Generation 49: = -623,3561837562652
Generation 50: = -607,8336859046028

Generation 52: = -607,6236858134077
Generation 53: = -580,3636864951359
Generation 54: = -576,601186262566
Generation 55: = -513,428687227379
Generation 56: = -497,1486888105528
Generation 58: = -487,373689507033
Generation 60: = -465,5201910910339
Generation 62: = -439,96619176087086
Generation 63: = -418,6511917597533
Generation 64: = -411,13749127555644
Generation 66: = -382,89519217442285
Generation 67: = -376,46519157785485
Generation 68: = -351,06369272568384
Generation 72: = -340,7586939337955
Generation 73: = -336,2636930564896
Generation 75: = -331,9501929158301
Generation 76: = -323,3874945346639
Generation 78: = -310,88999354593466
Generation 80: = -307,63019205327635
Generation 81: = -279,15269353217934
Generation 83: = -276,60369419933414
Generation 85: = -273,997693296145
Generation 86: = -247,12749554853892
Generation 87: = -236,92519516754453
Generation 88: = -218,25019526678628
Generation 93: = -204,66769526823913
Generation 94: = -211,76019477884773
Generation 95: = -203,39769529662584
Generation 96: = -199,01769539452647
Generation 98: = -194,0001949862719
Generation 100: = -186,3101952904791
Generation 101: = -185,44019536714558
Generation 102: = -182,11019547376338
Generation 103: = -179,02499628625813
Generation 104: = -171,2349964603782
Generation 107: = -162,25779622153644
Generation 108: = -156,90499641947451
Generation 111: = -147,03749662745747
Generation 117: = -145,8677965878815
Generation 118: = -146,69499652251605
Generation 120: = -146,26779666834776
Generation 123: = -146,69499652251605
Generation 125: = -146,29749664399776
Generation 128: = -143,70499658934776
Generation 132: = -142,66749659996484
Generation 133: = -142,21499659404162
Generation 138: = -138,09499671474103
Generation 153: = -136,06499653123322
Generation 162: = -133,8449965808541

Generation 164: = -126,2549967505038
Generation 167: = -119,86499689333144
Generation 232: = -119,85499689355495

Table B.1: Genetic Algorithm Test Results

| index | HP Miss | BG10 miss | LP Miss | Accuracy Score | Execution Time |
|---|---|---|---|---|---|
| R1:0 | 0 | 0 | 1 | -160.10 | 484671 |
| R1:1 | 0 | 1 | 0 | -117.09 | 593543 |
| R1:2 | 0 | 0 | 1 | -193.62 | 1043301 |
| R1:3 | 0 | 1 | 1 | -56.50 | 575146 |
| R1:4 | 0 | 0 | 1 | -113.73 | 868873 |
| R1:5 | 0 | 1 | 1 | -48.1025 | 569217 |
| R1:6 | 1 | 1 | 1 | -240.218 | 386464 |
| R1:7 | 0 | 0 | 0 | -244.205 | 590886 |
| R1:8 | 0 | 1 | 1 | -51.8753 | 673831 |
| R1:9 | 1 | 1 | 1 | -200.62 | 743448 |
| R1:10 | 1 | 0 | 1 | -108.09 | 623238 |
| R1:11 | 0 | 0 | 0 | -84.9403 | 460193 |
| R1:12 | 0 | 0 | 1 | -139.8 | 671583 |
| R1:13 | 0 | 1 | 1 | -124.467 | 655296 |
| R1:14 | 1 | 1 | 0 | -114.112 | 697762 |
| R1:15 | 1 | 1 | 1 | -95.2902 | 547004 |
| R1:16 | 1 | 1 | 1 | -161.142 | 646838 |
| R1:17 | 0 | 1 | 1 | -185.862 | 457633 |
| R1:18 | 0 | 1 | 1 | -184.035 | 672327 |
| R1:19 | 0 | 1 | 1 | -105.57 | 790586 |
| R1:20 | 0 | 1 | 1 | -126.377 | 559191 |
| R1:21 | 0 | 1 | 1 | -174.485 | 606371 |
| R1:22 | 0 | 0 | 1 | -124.317 | 1222599 |
| R1:23 | 1 | 0 | 1 | -153.71 | 458477 |
| R1:24 | 0 | 1 | 0 | -196.407 | 577816 |
| R1:25 | 0 | 1 | 0 | -204.405 | 851789 |
| R1:26 | 0 | 1 | 1 | -235.58 | 603594 |
| R1:27 | 1 | 0 | 1 | -135.407 | 564122 |
| R1:28 | 1 | 1 | 1 | -307.56 | 451141 |
| R1:29 | 1 | 0 | 1 | -126.24 | 1371167 |
| R2:0 | 0 | 0 | 1 | -209.787 | 833836 |
| R2:1 | 1 | 1 | 0 | -186.16 | 509726 |
| R2:2 | 1 | 1 | 1 | -131.015 | 695919 |
| R2:3 | 0 | 1 | 1 | -315.722 | 842125 |
| R2:4 | 0 | 1 | 1 | -152.162 | 742208 |
| R2:5 | 1 | 0 | 1 | -124.522 | 713443 |
| R2:6 | 0 | 0 | 1 | -132.072 | 760644 |
| R2:7 | 1 | 1 | 1 | -126.21 | 731581 |
| R2:8 | 1 | 1 | 0 | -246.703 | 396497 |
| R2:9 | 1 | 1 | 1 | -67.255 | 355168 |
| R2:10 | 0 | 0 | 1 | -116.334 | 349476 |
| R2:11 | 0 | 0 | 1 | -146.21 | 926908 |
| R2:12 | 1 | 1 | 1 | -161.378 | 709113 |
| R2:13 | 0 | 0 | 1 | -71.9425 | 1557649 |
| R2:14 | 0 | 1 | 0 | -182.287 | 475094 |
| R2:15 | 0 | 1 | 0 | -115.14 | 680169 |
| R2:16 | 1 | 1 | 0 | -215.555 | 375511 |

| | | | | | |
|---|---|---|---|---|---|
| R2:17 | 1 | 0 | 1 | -165.052 | 744280 |
| R2:18 | 0 | 0 | 1 | -122.513 | 753761 |
| R2:19 | 0 | 0 | 1 | -205.735 | 552506 |
| R2:20 | 0 | 0 | 1 | -59.7225 | 778056 |
| R2:21 | 0 | 0 | 1 | -96.6762 | 824888 |
| R2:22 | 0 | 0 | 0 | -75.3028 | 500240 |
| R2:23 | 1 | 1 | 1 | -158.458 | 672429 |
| R2:24 | 1 | 0 | 1 | -301.899 | 373255 |
| R2:25 | 0 | 1 | 1 | -226.897 | 471755 |
| R2:26 | 0 | 0 | 1 | -193.32 | 443372 |
| R2:27 | 0 | 1 | 1 | -337.172 | 312326 |
| R2:28 | 0 | 0 | 1 | -75.025 | 509764 |
| R2:29 | 1 | 0 | 0 | -182.095 | 1032512 |
| R2:30 | 0 | 0 | 0 | -163.76 | 1016758 |
| R2:31 | 0 | 1 | 0 | -160.238 | 492823 |
| R2:32 | 0 | 0 | 1 | -296.682 | 372587 |
| R2:33 | 1 | 0 | 1 | -88.785 | 770575 |
| R2:34 | 0 | 1 | 0 | -100.718 | 585580 |
| R2:35 | 0 | 0 | 1 | -116.735 | 1477906 |
| R2:36 | 0 | 1 | 1 | -92.0403 | 421024 |
| R2:37 | 0 | 1 | 0 | -232.168 | 497562 |
| R2:38 | 1 | 0 | 0 | -179.69 | 447369 |
| R2:39 | 0 | 0 | 1 | -227.8 | 969192 |
| R2:40 | 0 | 0 | 1 | -89.005 | 1620479 |
| R2:41 | 1 | 1 | 1 | -88.6725 | 744504 |
| R2:42 | 1 | 1 | 1 | -187.885 | 625054 |
| R2:43 | 1 | 1 | 0 | -75.61 | 336245 |
| R2:44 | 1 | 0 | 0 | -159.663 | 410225 |
| R2:45 | 0 | 0 | 1 | -67.53 | 632751 |
| R2:46 | 0 | 0 | 1 | -59.1903 | 521166 |
| R2:47 | 1 | 1 | 1 | -51.3975 | 824546 |
| R2:48 | 1 | 1 | 0 | -176.65 | 860264 |
| R2:49 | 1 | 0 | 1 | -78.1725 | 1181576 |
| R3:0 | 0 | 1 | 0 | -210.505 | 944064 |
| R3:1 | 0 | 0 | 1 | -176.512 | 914442 |
| R3:2 | 0 | 1 | 1 | -121.045 | 719226 |
| R3:3 | 0 | 0 | 1 | -236.952 | 791274 |
| R3:4 | 1 | 1 | 1 | -226.017 | 1019954 |
| R3:5 | 1 | 1 | 1 | -116.365 | 712945 |
| R3:6 | 0 | 1 | 1 | -114.805 | 605772 |
| R3:7 | 1 | 1 | 1 | -259.645 | 362193 |
| R3:8 | 0 | 1 | 0 | -123.717 | 563921 |
| R3:9 | 0 | 1 | 0 | -97.1328 | 659937 |
| KR1:0 | 0 | 0 | 1 | -174.52 | |
| KR1:1 | 0 | 1 | 1 | -188.35 | |
| KR1:2 | 0 | 1 | 0 | -126.763 | |
| KR1:3 | 0 | 1 | 1 | -132.21 | |
| KR1:4 | 0 | 0 | 1 | -216.715 | |
| KR1:5 | 0 | 0 | 1 | -68.2225 | |

| | | | | |
|---|---|---|---|---|
| KR1:6 | 1 | 0 | 0 | -123.045 |
| KR1:7 | 0 | 1 | 1 | -239.657 |
| KR1:8 | 0 | 0 | 1 | -191.065 |
| KR1:9 | 0 | 0 | 0 | -83.6425 |
| KR1:10 | 1 | 1 | 1 | -127.744 |
| KR1:11 | 0 | 1 | 1 | -181.385 |
| KR1:12 | 0 | 0 | 0 | -48.28 |
| KR1:13 | 1 | 0 | 1 | -167.905 |
| KR1:14 | 1 | 0 | 1 | -150.7 |
| KR1:15 | 1 | 0 | 0 | -163.66 |
| KR1:16 | 0 | 1 | 1 | -175.125 |
| KR1:17 | 0 | 0 | 1 | -103.817 |
| KR1:18 | 0 | 1 | 1 | -295.728 |
| KR1:19 | 1 | 1 | 1 | -141.78 |
| KR1:20 | 1 | 1 | 0 | -121.485 |
| KR1:21 | 1 | 0 | 1 | -100.447 |
| KR1:22 | 0 | 1 | 1 | -208.87 |
| KR1:23 | 0 | 1 | 1 | -57.9053 |
| KR1:24 | 1 | 0 | 0 | -118.842 |
| KR1:25 | 0 | 0 | 0 | -148.878 |
| KR1:26 | 0 | 1 | 1 | -177.957 |
| KR1:27 | 0 | 0 | 1 | -115.715 |
| KR1:28 | 1 | 1 | 0 | -182.473 |
| KR1:29 | 0 | 0 | 1 | -173.492 |

# C | Appendix C

Scheduling Job Order Balancing - Repository