

## **ABSTRACT**

Many applied and experimental situations exist in our day-to-day working in factories and industrial production concerns, and so forth, in which different jobs are processed on various machines in a fixed order. Job scheduling is a process executed by a job scheduler where jobs can be scheduled not only on single systems, but also on multiple systems, like scheduling the different manufacturing components in an automobile manufacturing plant. It can be necessarily complex. In a flow shop, the work in a job is broken down into separate tasks called operations, and each of these operations are performed in a different machine of the designated flow shop. In this context, a job is a collection of operations with a special precedence that must be carried out by it to be declared as complete. With reference to our work, the flow of work is assumed to be unidirectional, and has been represented as a flow shop having static job allocation such that each job passes through each machine for a constant time-slice. Our project aims to optimize of the scheduling of the above mentioned jobs on machines. We shall be using natural algorithms to obtain an optimal schedule. Our use of natural algorithms permits us to define the optimal solution and, unlike heuristic methods, natural algorithms operate on a population of solutions rather than a single solution. We have implemented flow shop scheduling with make span computation based on efficiency of different jobs and machines using natural algorithms. Comparison of the outputs with previously generated outputs by different researchers has been done and the time complexity based on the Average Relative Percentage Difference (ARPD) has been improvised.

## **ACKNOWLEDGEMENTS**

We are grateful to our mentor and project guide Prof. Siladitya Mukherjee for his unwavering help and assistance. We are grateful to the Department of Computer Science for giving us the opportunity to do research work concerning natural algorithms.

<b>1. INTRODUCTION.....</b>	<b>1</b>
1.1. Background.....	1
1.2. Objectives.....	3
1.3. Purpose, Scope and Applicability .....	4
1.4. Achievements.....	6
1.5. Organization of Report .....	8
<b>2. SURVEY OF TECHNOLOGIES .....</b>	<b>12</b>
2.1. Python Programming Language.....	12
2.2. Python Libraries .....	13
<b>3. REQUIREMENTS AND ANALYSIS .....</b>	<b>17</b>
3.1. Problem Definition.....	17
3.2. Requirements Specification.....	18
3.3. Planning and Scheduling .....	20
3.4. Software and Hardware Requirements .....	22
3.5. Preliminary Product Description .....	23
3.6. Conceptual Models .....	24
<b>4. SYSTEM DESIGN .....</b>	<b>25</b>
4.1. Basic Modules.....	25
4.2. Procedural Design.....	30
4.3. User Interface Design .....	36
4.4. Test Cases Design.....	37
<b>5. IMPLEMENTATION AND TESTING .....</b>	<b>41</b>
5.1. Plan of Implementation.....	41
5.2. Code Details and Efficiency.....	42
5.3. CODE TESTING.....	47
<b>6. RESULTS AND DISCUSSIONS .....</b>	<b>63</b>
6.1. Results.....	63
6.2. User Documentation .....	71
<b>7. CONCLUSIONS .....</b>	<b>75</b>
<b>REFERENCES.....</b>	<b>76</b>
<b>APPENDIX .....</b>	<b>77</b>
Program Code.....	77

## TABLE OF FIGURES

	Index	Content	Page No.
<i>Chapter 4</i>			
	Figure 4.1	Front End Design	42
<i>Chapter 5</i>			
	Figure 5.1	Crossover Function	53
	Figure 5.2	Mutation Function	54
	Figure 5.3	Genetic Module	55
	Figure 5.4	Random Number Generator Matrix	56
	Figure 5.5-5.11	Integrated testing	57-62
	Figure 5.12-5.19	Modified Code Snippet	63-67

## TABLE OF TABLES

	Index	Content	Page No.
<i>Chapter 3</i>	Table 3.1	PERT chart	24
	Table 3.2	Gantt Chart	25
	Table 3.3	Gantt Chart (contd.)	26
	Table 3.4	Basic flow chart	29
<i>Chapter 4</i>			
	Table 4.1	Explaining makespan computation	31
	Table 4.2	Explaining makespan computation (contd.)	32
	Table 4.3	Test Cases	43-45
<i>Chapter 6</i>			
	Table 6.1	Results	68-75

# 1. INTRODUCTION

## 1.1. Background

A schedule or a timetable, as a basic time-management tool, consists of a list of times at which possible tasks, events, or actions are intended to take place, or of a sequence of events in the chronological order in which such things are intended to take place. The process of creating a schedule - deciding how to order these tasks and how to commit resources between the varieties of possible tasks - is called scheduling, and the entity responsible for making a particular schedule may be called a scheduler. Making and following schedules is an ancient human activity. Job scheduling [1] is a process executed by a job scheduler where jobs can be scheduled not only on single systems, like scheduling jobs or processes in the CPU memory for execution, but also on multiple systems, like scheduling the different manufacturing components in an automobile manufacturing plant. This is commonly called batch scheduling, as execution of non-interactive jobs is often called batch processing. Basic features expected of job scheduler software include interfaces which help to define workflows and/or job dependencies, automatic submission of executions, interfaces to monitor the executions and priorities and/or queues to control the execution order of unrelated jobs. Job scheduling is performed using job schedulers. Job schedulers are programs that enable scheduling and, at times, track computer "batch" jobs, or units of work like the operation of a payroll program. Job schedulers have the ability to start and control jobs automatically by running prepared job-control-language statements or by means of similar communication with a human operator. Generally, the present-day job schedulers include a graphical user interface (GUI) along with a single point of control.

In a scheduling process, we need to know the type and amount of each resource so that we can determine when the tasks can feasibly be accomplished. While specifying the resources, we actually define the boundary of the scheduling problem. In addition, we describe each task in terms of such information as its resource requirement, its duration, the earliest time at which it may start, and the time at which it is due to complete. In general, the task duration is uncertain, but we may want to suppress that uncertainty when stating the problem. We should also describe any technological constraints (precedence restrictions) that exist among the tasks. Information about resources and tasks defines a scheduling problem. Many of the early developments in the field of scheduling were motivated by problems arising in manufacturing. Therefore, it was natural to employ the vocabulary of manufacturing when describing scheduling problems. Now, although scheduling work is of considerable significance in many non-manufacturing areas, the terminology of manufacturing is still frequently used. Thus, resources are usually called machines and tasks are called jobs. Sometimes,

jobs may consist of several elementary tasks called operations. Jobs flow from an initial machine, through several intermediate machines, and ultimately to a final machine before completing. Traditionally, we refer to this design as a flow shop, even though an actual shop may contain much more than a single serial configuration. Ideally in the job scheduling process, the objective function should consist of all costs that depend on scheduling decisions. In practice, however, such costs are often difficult to measure, or even to completely identify. The major operating costs—and the most readily identifiable—are determined by the planning function, while scheduling-related costs are difficult to isolate and often tend to appear fixed. We categorize the major scheduling models by specifying the resource configuration and the nature of the tasks. For instance, a model may contain one machine or several machines. If it contains one machine, jobs are likely to be single stage, whereas multiple-machine models usually involve jobs with multiple stages. In either case, machines may be available in unit amounts or in parallel. In addition, if the set of jobs available for scheduling does not change over time, the system is called static, in contrast to cases in which new jobs appear over time, where the system is called dynamic. Traditionally, static models have proved more tractable than dynamic models and have been studied more extensively. Although dynamic models would appear to be more important for practical application, static models often capture the essence of dynamic systems, and the analysis of static problems frequently uncovers valuable insights and sound heuristic principles that are useful in dynamic situations. Finally, when conditions are assumed to be known with certainty, the model is called deterministic. On the other hand, when we recognize uncertainty with explicit probability distributions, the model is called stochastic. In this context it is required to define the term makespan. Let us imagine that we are given  $m$  machines for scheduling, indexed by the set  $M = \{1, \dots, m\}$ . There are furthermore given  $n$  jobs, indexed by the set  $J = \{1, \dots, n\}$ , where job  $j$  takes  $p_{i,j}$  units of time if scheduled on machine  $i$ . Let  $J_i$  be the set of jobs scheduled on machine  $i$ . Then  $L_i = \sum_{j \in J_i} p_{i,j}$  is the load of machine  $i$ . The maximum load  $L_{\max} = c_{\max} = \max_{i \in M} L_i$  is called the makespan of the schedule. The primary objective of the project is to design an intelligent job scheduling strategy with makespan computation using natural algorithms.

## 1.2. Objectives

This project deals with a model based on a design in which machines are arranged in series. In this design, jobs flow from an initial machine, through several intermediate machines, and ultimately to a final machine before completing. Traditionally, we refer to this design as a flow shop, even though an actual shop may contain much more than a single serial configuration. In a flow shop, the work in a job is broken down into separate tasks called operations, and each operation is performed at a different machine. In this context, a job is a collection of operations with a special precedence structure. In particular, each operation after the first has exactly one direct predecessor and each operation before the last has exactly one direct successor. Thus, each job requires a specific sequence of operations to be carried out for the job to be complete. The shop contains  $m$  different machines, and in the “pure” flow [1] shop model, each job consists of  $m$  operations, each of which requires a different machine. The machines in a flow shop can thus be numbered  $1, 2, \dots, m$ ; and the operations of job  $j$  numbered  $(1, j), (2, j), \dots, (m, j)$ , so that they correspond to the machine required. In the general case, jobs may require fewer than  $m$  operations, their operations may not always require adjacent machines, and the initial and final operations may not always occur at machines 1 and  $m$ . Nevertheless, the flow of work is still unidirectional, and we can represent the general case as a pure flow shop in which some of the operation times are zero.

Our project deals with the optimization of the scheduling of the above mentioned jobs on machines. We shall be using natural algorithms to obtain an optimal schedule. Our use of natural algorithms enables us to determine the best solution and, unlike heuristic methods, natural algorithms operate on a population of solutions rather than a single solution. We will be using a heuristic method which uses an incremental approach similar to that proposed by Laha and Chakravorty [2], which, like many other algorithms used to solve the PFSSP, is based on the Nawaz-Enscore-Ham heuristic [3]. We will implement flow shop scheduling with make span computation based on efficiency of different jobs and machines using natural algorithms. Comparison of the outputs with previously generated outputs by different researchers will be done and improvise the time complexity based on the Average Relative Percentage Difference (ARPD).

### 1.3. Purpose, Scope and Applicability

#### 1.3.1. Purpose

The main purpose of the project is to create an optimized scheduler for a set of jobs or machines using natural algorithms. The project aims to optimize job scheduling using natural algorithms. So it falls under the category of optimization and scheduling.

The model on which this project is based deploys machines which are arranged in series. In this design, jobs flow from an initial machine, passes consecutively through several intermediate machines, and ultimately to a final machine before completing in a static arrangement. Traditionally, this is mentioned as a flow shop, even though in a practical scenario a shop may encompass much more than a single serial configuration. In a flow shop, the work in a job is broken down into separate tasks called operations, and each of these operations are performed in a different machine of the designated flow shop. In this context, a job is a collection of operations with a special precedence structure. In particular, each operation after the first has exactly one direct predecessor and each operation before the last has exactly one direct successor. Thus, each job requires a specific sequence of operations to be carried out for the job to be declared as complete. In the general case, jobs may require fewer operations than the number of allocated machines, their operations may not always require adjacent machines, and the initial and final operations may not always occur at the first and the last machine in the flow shop arrangement. Nevertheless, the flow of work is still unidirectional, and we can represent the general case as a pure flow shop in which some of the operation times are zero.

Our project deals with the optimization of the scheduling of the above mentioned jobs on machines. We shall be using natural algorithms to obtain an optimal schedule. Our use of natural algorithms permits us to define the optimal solution and, unlike heuristic methods, natural algorithms operate on a population of solutions rather than a single solution. We will implement flow shop scheduling with make span computation based on efficiency of different jobs and machines using natural algorithms. Comparison of the outputs with previously generated outputs by different researchers will be done and improvise the time complexity based on the Average Relative Percentage



Difference (ARPD). Many applied and experimental situations exist in our day-to-day working in factories and industrial production concerns, and so forth, in which different jobs are processed on various machines in a fixed order. The flow shop scheduling optimization obtained through this project can be used in several applications of system engineering. This is meant to help in efficient scheduling of jobs in a series of machines in such a way that the work flow efficiency can be maximized. In this regard some sophisticated tweaking in the program module can help us to extend this principle in dynamic scheduling as well and help in optimum utilization of resources. The benefits of efficient flow shop scheduling can be used in medical science too.

### **1.3.2. Scope and Applicability**

The need for scheduling in system engineering cannot be ignored. In a systems environment the efficiency of a system can be determined by parameters such as how much work the organization, program or project resources have the current capacity to perform within a specified time frame , what resources are overcommitted or underutilized ,what work is actively proceeding ,how much work is actively proceeding ,what work is blocked and thus inactive ,what is the continuing impact of unpredicted changes in priority, urgent maintenance or critical development responses, changes in requirements or scope, or other emergent issues ,what is the actual progress toward various project outcomes and how often and when is value finally delivered to the user. Thus to maximize the efficiency of a workflow an efficient scheduling strategy is required. In this case an attempt has been made to reduce the make-span computation while scheduling a set of jobs to a set of machines. The same idea can be expanded to a larger system and with certain priority parameters to suit changing needs. The applicability of this project can thus be extended to several instances of system engineering requirements. One major scope of this solution is to optimize the radiation therapy process for cancer patients. This method can be used to optimize area which will be affected by radiation. This area will have maximum number of cancer cells and minimum number of unaffected cells. Various image processing algorithms exists which can be used to detect cancer cells. Once detected, the radiation therapy can be concentrated on those locations which have maximum number of cancer cells.

## 1.4. Achievements

During the course of this project, we developed a firm understanding of genetic algorithms and how genetic algorithm operators can be used for optimization. As a result, we understood the different types of genetic operators [4] (crossovers [5], mutations [6], selections [7]) and how each of these operators contributed to the overall optimization process. In genetic algorithms, crossover is a genetic operator used to vary the programming of a chromosome or chromosomes from one generation to the next. It is analogous to reproduction and biological crossover, upon which genetic algorithms are based. Crossover [5] is a process of taking more than one parent solutions and producing a child solution from them. We have implemented ordered crossover technique to increase the efficiency of the overall scheduling process. In ordered crossover a portion of one parent is mapped to a portion of the other parent. From the replaced portion on, the rest is filled up by the remaining genes, where already present genes are omitted and the order is preserved.

Mutation [6] is a genetic operator used to maintain genetic diversity from one generation of a population of genetic algorithm chromosomes to the next. It is analogous to biological mutation. Mutation alters one or more gene values in a chromosome from its initial state. In mutation, the solution may change entirely from the previous solution. Hence GA can come to better solution by using mutation. Mutation occurs during evolution according to a user-definable mutation probability. We have implemented inverse mutation, which swaps jobs from the start and end positions of the jobs array, pairwise swap mutation, which swaps the pair of jobs at the start and end positions and insert mutation, which inserts the job at the end position just after the job at the start position.

Selection [7] is the stage of a genetic algorithm in which individual genomes are chosen from a population for later breeding. We have implemented Roulette Wheel Selection technique. The basic part of the selection process is to stochastically select from one generation to create the basis of the next generation. The requirement is that the fittest individuals have a greater chance of survival than weaker ones. This replicates nature in that fitter individuals will tend to have a better probability of survival and will go forward to form the mating pool for the next generation. Weaker individuals are not without a chance. In nature such individuals may have genetic coding that may prove useful to future generations.

Since we have used python as our coding language, we have also learnt a lot about some of its important libraries, read numpy, openpyxl and itertools. The code syntax in python is relatively simpler as compared to other high level languages.

Now our project was designed to optimize the makespan for a set of jobs given a set of machines. After comparison, it turns out that the proposed heuristic technique is suitable for use in large PFSSPs, such as those with 500 jobs or maybe even more.

## **1.5. Organization of Report**

The user documentation has been arranged into seven chapters.

### **1.5.1. Chapter 1: Introduction**

A rudimentary description of the context of the project and its relation to work already done in the area has been summarized. A brief statement of the aims and objectives of the project has been proposed. The purpose of this chapter deals with the description of the topic of our project that answers questions on why we are doing this project and how theoretically significant improvements have been made by our design. Scope includes a brief overview of the methodology, assumptions and limitations and the main issues being dealt with by us. The direct and indirect applications of our work has been described. The subsection named achievements explain what knowledge we have achieved after the completion of our work and what contributions our project has made. The subsection named “Goals achieved” describes the degree to which the findings support the original objectives laid out by the project. The goals may be partially or fully achieved, or exceeded.

### **1.5.2. Chapter 2: Survey of Technologies**

In the chapter Survey of Technologies, we have demonstrated our awareness and understanding of Available Technologies related to the topic of our project. We have given the detail of all the related technologies that are necessary to complete our project. We have described the technologies available in our chosen area and present a comparative study of all those Available Technologies.

### **1.5.3. Chapter 3: Requirements and Analysis**

**Problem Definition:** The problem has been defined providing the details of the overall problem and then the problem has been divided into sub-problems.

**Requirements Specification:** In this phase we have defined the requirements of the system, independent of how these requirements will be accomplished. The Requirements Specification describes the things in the system and the actions that can be done on these things by identifying the operation and problems of the existing system.

**Planning and Scheduling:** Planning and scheduling is a complicated part of software development. Planning, for our purposes, can be thought of as determining all the small tasks that must be carried out in order to accomplish the goal. Planning also takes into account, rules, known as constraints, which, control when certain tasks can or cannot happen. Scheduling can be thought of as determining whether adequate resources are available to carry out the plan.

**Software and Hardware Requirements:** The details of all the software and hardware needed for the development and implementation of our project has been defined.

**Preliminary Product Description:** The requirements and objectives of the new system has been defined. The functions and operation of the application/system we have developed as our project has been explored.

**Conceptual Models:** The problem domain which describes operations that can be performed on the system, and the allowable sequences of those operations has been described.

### **1.5.4. Chapter 4: System Design**

Describes desired features and operations in detail, including screen layouts, business rules, process diagrams, pseudocode and other documentation.

**Basic Modules:** We have followed the divide and conquer theory, and divided the overall problem into more manageable parts and developed each part or module separately. When all modules are ready, we have integrated all the modules into one system.

Data Design: Data design consists of how we organise, manage and manipulate the data.

- Schema Design: The structure and explanation of schemas used in our project has been defined.
- Data Integrity and Constraints: The validity checks and constraints have been provided to keep an integrity check.

Procedural Design: Procedural design is a systematic way for developing algorithms or procedurals and consists of the following modules.

- Logic Diagrams: Define the systematical flow of procedure that improves its comprehension and helps the programmer during implementation. e.g., Control Flow Chart, Process Diagrams etc.
- Data Structures: Create and define the data structure used in your procedures.
- Algorithms Design: With proper explanations of input data, output data, logic of processes, design and explain the working of algorithms.

User Interface Design: User, task, environment analysis have been identified and how we intend to map those requirements in order to develop a “User Interface. Some rough pictorial views of the user interface and its components has been shown.

Test Cases Design: Test cases which provide easy detection of errors and mistakes within a minimum period of time and with the least effort has been shown.

### **1.5.5. Chapter 5: Implementation and Testing**

Implementation Approaches: This chapter defines the plan of implementation, and the standards being used in the implementation.

Testing Approach consists of the submodules unit testing and integrated testing.

- Unit Testing: Unit testing deals with testing a unit or module as a whole. This tests the interaction of many functions.
- Integrated Testing: Brings all the modules together into a special testing environment, then checks for errors, bugs and interoperability. It deals with tests for the entire application. Application limits and features are tested here.

Modifications and Improvements deals with the modifications introduced to increase the code efficiency.

#### **1.5.6. Chapter 6: Results and Discussion**

Test Reports: Explain the test results and reports based on our test cases, which shows that our software is capable of facing any problematic situation and that it works fine in different conditions. Different sample inputs and outputs have been shown.

#### **1.5.7. Chapter 7: Conclusions**

Conclusion: The conclusion has been summarized in a fairly short chapter explaining the limitations we have encountered during the testing of our software that we were not able to modify.

Future Scope of the Project describes two things: firstly, new areas of investigation prompted by developments in this project, and secondly, parts of the current work that were not completed due to time constraints and/or problems encountered.

## 2. SURVEY OF TECHNOLOGIES

### 2.1. Python Programming Language

Python [8] is a very powerful high level, object oriented programming language. It is free (open source) and platform independent. Python's design emphasizes code readability and it has an extremely simple syntax. Expressions and statements can be written in fewer lines as compared to other programming languages such as Java, C, C++. All these features, along with its huge library support, make Python a prominent programming language among programmers.

Python is a multi-paradigm programming language: object-oriented programming and structured programming are fully supported, and there are a number of language features which support functional programming. An important goal of the Python developers is making Python fun to use. This is reflected in the origin of the name, which comes from Monty Python. Python is intended to be a highly readable language. It is designed to have a simple visual layout, frequently using English keywords where other languages use punctuation. Python uses whitespace indentation, rather than curly braces or keywords, to delimit blocks; this feature is also termed the "off-side rule". An increase in indentation comes after certain statements; a decrease in indentation signifies the end of the current block.

Python programs are generally expected to run slower than Java programs, but they also take much less time to develop. Python programs are typically 3-5 times shorter than equivalent Java programs. This difference can be attributed to Python's built-in high-level data types and its dynamic typing. For example, a Python programmer wastes no time declaring the types of arguments or variables, and Python's powerful polymorphic list and dictionary types, for which rich syntactic support is built straight into the language, find a use in almost every Python program. Because of the run-time typing, Python's run time must work harder than Java's. For example, when evaluating the expression  $a+b$ , it must first inspect the objects  $a$  and  $b$  to find out their type, which is not known at compile time. It then invokes the appropriate addition operation, which may be an overloaded user-defined method. Java, on the other hand, can perform an efficient integer or floating point addition, but requires variable declarations for  $a$  and  $b$ , and does not allow overloading of the  $+$  operator for instances of user-defined classes. It has been found that the programs written in Python are 5-10 times shorter than the programs written in C++.



## 2.2. Python Libraries

In this section, we shall discuss the libraries we have used to develop our code. The extensive nature of Python's standard library is one of its greatest strengths. It provides tools suited for many tasks.

### 2.2.1. The “NumPy” library

The NumPy [9] library was mainly used because of its support for the N-dimensional array object among other things. Multi-dimensional arrays can be created using the list data structure. But manipulating high order lists becomes a problem. The NumPy library enables the programmer to easily define and manipulate multi-dimensional arrays. NumPy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases. NumPy increases program execution by providing multi-dimensional arrays and functions and operators that operate efficiently on arrays. The core functionality of the NumPy library is its “ndarray”, for n-dimensional array, data structure. It is because of this fact that we have used this library. The operations provided by the library have enabled us to easily append arrays after we have applied the various genetic algorithm operators.

### 2.2.2. The “Openpyxl” library

The “openpyxl” [10] library is a Python library for reading and writing Excel 2010 xlsx/xlsm/xltx/xltx files. It was created as there was no existing library to read or write natively from Python the Office Open XML format. We have used this library to read data from the Taillard benchmark's file. We have also used it to write data in the same file. There is no need to create a file on the file system to get started with openpyxl. We just need to import the “workbook” class to start using it. A “workbook” is always created with atleast one worksheet. Cells can be directly accessed as keys of the worksheet.

### 2.2.3. The “random” library

The “random” [11] library implements pseudo-random number generators for various distributions. For integers, uniform selection from a range of integers. For sequences, uniform selection of a random element, a function to generate a random permutation of a list-in-place and a function for random sampling without replacement. Python uses the Mersenne Twister as the core generator. It produces 53 bit precision floats and has a period of  $2^{19937} - 1$ . The Mersenne Twister is one of the most extensively tested random number generators in existence. However, being completely deterministic, it is not suitable for all purposes, and is completely unsuitable for cryptographic purposes.

The “random” library was mainly used by us in our various genetic operations. We used the library to randomly select start and end ‘alleles’. This generated two random numbers based on which we performed various genetic operations.

### 2.2.4. The “sys” library

The “sys” [12] module provides information about constants, functions and methods of the Python interpreter. One of the main utilities of the “sys” module is the `dir(system)`, which gives a summary of the available constants, functions and methods. We have used the “sys” module to ensure that the complete nd-array is being displayed. Usually for large arrays, the values in the middle are not displayed. The values in the beginning and end are displayed. Importing this library and using `sys.maxsize`, we can ensure that the complete nd-array is being displayed.

### 2.2.5. The “itertools” library

The “itertools” [13] module implements a number of iterator building blocks inspired by constructs from APL, Haskell and SML. Each has been recast in a form suitable for Python. The module standardizes a core set of fast, memory efficient tools that are useful by themselves or in combination. Together, they form an “iterator algebra” making it possible to construct specialized tools succinctly and efficiently in pure Python.

Iterator based code may be preferred over code which uses lists for several reasons. Since data is not produced from the iterator until it is needed, all of the data is not stored in the memory at the same time. Reducing memory usage can reduce swapping and other side-effects of large data sets, increasing performance.

Our primary use of this module was to generate a sequence list, which contains permutations of the number of jobs being considered. For example, if we start with 4 jobs, we end up with a sequence list containing 24 sequences, (0,1,2,3), (0,1,3,2), (0,3,1,2) and so on.

### 2.2.6. The “tkinter” module

The “tkinter” [14] module (“Tk interface”) is the standard Python GUI toolkit from Scriptics (formerly developed by Sun Labs). “tkinter” consists of a number of modules. The “tk” interface is provided by a binary extension module named `_tkinter`. This module contains low-level interface to “tk”, and should never be used directly by application programmers.

The public interface is provided through a number of Python modules. The most important interface module is the “tkinter” module itself. The “tkinter” module only exports widget classes and associated constants. To initialize “tkinter”, we have to create a “tk” root widget. This is an ordinary window, with a title bar and other decoration provided by the window manager. Only one root widget should be created for each program and it must be created before any widgets. A label widget can display either text or an icon or other image. There is a method called `pack`, which is used to “pack” the given text, image etc. so that it is visible. Even after all this, we must call the “tkinter” event loop, to make the window appear. The program will stay in the event loop until we close the window. The event loop doesn’t only handle events from the user ( such as mouse clicks and key presses) or window system (such

as redraw events and window configuration messages), it also handles operations queued by “tkinter” itself. Among these operations are geometry management (queued by the pack method) and display updates. This also means that the application window will not appear before we enter the main loop.

### 3. REQUIREMENTS AND ANALYSIS

#### 3.1. Problem Definition

This project will implement and optimize flow shop scheduling with makespan computation based on efficiency of different jobs and machines using natural algorithms. Comparison of the outputs with previously generated outputs by different researchers will be done and improve the time complexity based on the Average Relative Percentage Difference (ARPD). We have used the Genetic Algorithms to optimize the whole process. Jobs flow from an initial machine, through several intermediate machines, to a final machine before completing [1]. Traditionally, we refer to this design as a flow shop, even though an actual shop may contain much more than a single serial configuration. We categorize the major scheduling models by specifying the resource configuration and the nature of the tasks. For instance, a model may contain one machine or several machines. If it contains one machine, jobs are likely to be single stage, whereas multiple-machine models usually involve jobs with multiple stages. In either case, machines may be available in unit amounts or in parallel. Let us imagine that we are given  $m$  machines for scheduling, indexed by the set  $M = \{1, \dots, m\}$ . There are ' $n$ ' jobs, indexed by the set  $J = \{1, \dots, n\}$ , where job  $j$  takes  $p_{ij}$  units of time if scheduled on machine  $i$ . Let  $J_i$  be the set of jobs scheduled on machine  $i$ . Then  $L_i = \sum_{j \in J_i} p_{ij}$  is the load of machine  $i$ . The maximum load  $L_{\max} = c_{\max} = \max_{i \in M} L_i$  is called the makespan of the schedule. The primary objective of the project is to design an intelligent job scheduling strategy with makespan computation using natural algorithms.

## 3.2. Requirements Specification

The program takes as input the benchmarks specified by Eric Taillard [15]. In his benchmarks he has specified the number of jobs, number of machines, timeseed values for each job-machine pair and lower and upper bound makespan values. Our program takes as input the jobs, machines and the respective timeseed value. Our program uses a heuristic method which uses an incremental approach similar to that proposed by Laha and Chakravorty [2], which, like many other algorithms used to solve the PFSSP, is based on the Nawaz-Enscore-Ham heuristic [3]. However, what sets our method apart is that we combine this with natural algorithms to further improve the sequences obtained at each stage. We will be using various crossover, mutation and selection techniques to optimize the makespan computation value. We employ natural algorithms to improve the efficiency of the system by using makespan computation. The computed makespan value is then compared with the given range. Efficiency of the obtained algorithm needs to be compared with existing techniques to evaluate its worth. Based on the ARPD, the efficiency obtained should be an improvement upon previous works. In doing so, the following processes will be used.

### 3.2.1. Crossovers

In genetic algorithms, crossover [5] is a genetic operator used to vary the programming of a chromosome or chromosomes from one generation to the next. It is analogous to reproduction and biological crossover, upon which genetic algorithms are based. Cross over is a process of taking more than one parent solutions and producing a child solution from them. We aim to implement ordered crossover technique to increase the efficiency of the overall scheduling process. In ordered crossover a portion of one parent is mapped to a portion of the other parent. From the replaced portion on, the rest is filled up by the remaining genes, where already present genes are omitted and the order is preserved.

### 3.2.2. Mutations

Mutation [6] is a genetic operator used to maintain genetic diversity from one generation of a population of genetic algorithm chromosomes to the next. It is analogous to biological mutation. Mutation alters one or more gene values in a chromosome from its initial state. In mutation, the solution may change entirely from the previous solution. Hence GA can come to better solution by using mutation. Mutation occurs during evolution according to a user-

definable mutation probability. We aim to implement inverse mutation, which will swap jobs from the start and end positions of the jobs array, pairwise swap mutation, which will swap the pair of jobs at the start and end positions and insert mutation, which will insert the job at the end position just after the job at the start position.

### **3.2.3. Selection**

Selection [7] is the stage of a genetic algorithm in which individual genomes are chosen from a population for later breeding. We will implement Roulette Wheel Selection technique. The basic part of the selection process is to stochastically select from one generation to create the basis of the next generation. The requirement is that the fittest individuals have a greater chance of survival than weaker ones. This replicates nature in that fitter individuals will tend to have a better probability of survival and will go forward to form the mating pool for the next generation. Weaker individuals are not without a chance. In nature such individuals may have genetic coding that may prove useful to future generations.

### **3.2.4. Average Relative Percentage Difference (ARPD)**

We will use Average Relative Percentage Difference (ARPD) to improve the time complexity. This is used to compare two quantities while taking into consideration their actual “sizes”. The comparison is expressed as a ratio. Multiplying this by 100 gives us the relative percentage difference.

### 3.3. Planning and Scheduling

#### 3.3.1. PERT Chart

The following table shows the PERT chart relevant to our project.

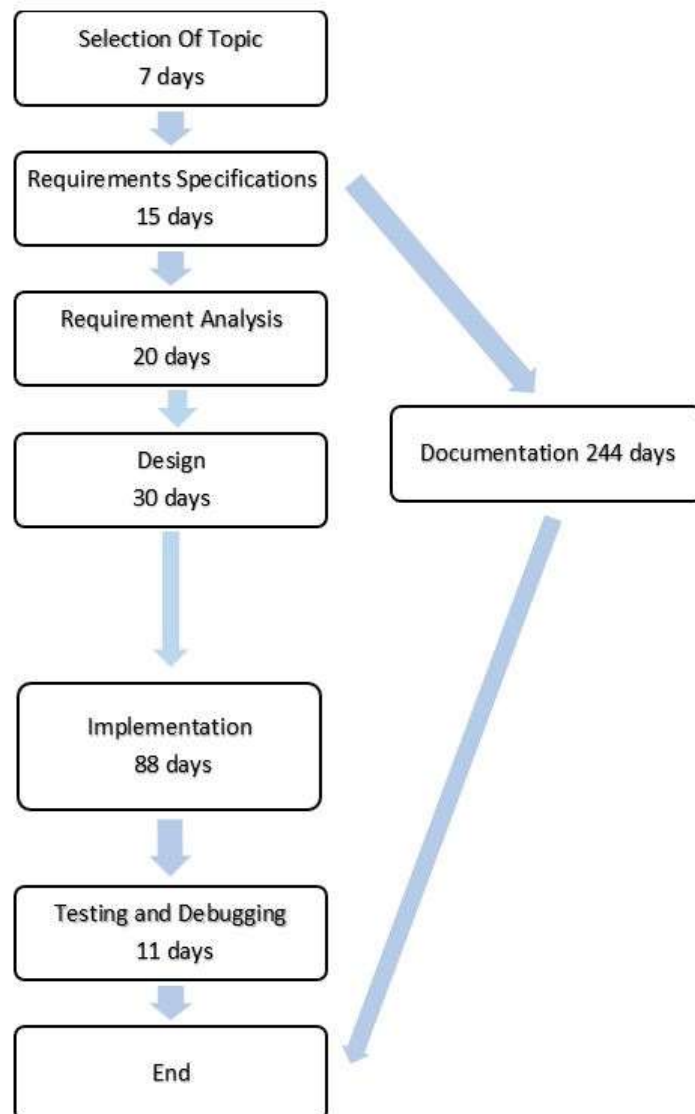


Table 3.1: PERT chart



### 3.3.2. Gantt Chart

The following table shows the Gantt chart relevant to our project.

Task Name	Start	End	Duration (days)
<b>Selection of Topic</b>	15 - Jul	22 - Jul	7
<b>Requirement Specification</b>	02 - Aug	17 - Aug	15
<b>Requirement Analysis</b>	20 - Aug	09 - Sep	20
<b>Design</b>	01 - Oct	31 - Oct	30
<b>Implementation</b>	17 - Dec	14 - Mar	88
<b>Testing and Debugging</b>	15 - Mar	26 - Mar	11
<b>Documentation</b>	20 - Aug	20 - Apr	244

Table 3.1 Gantt Chart

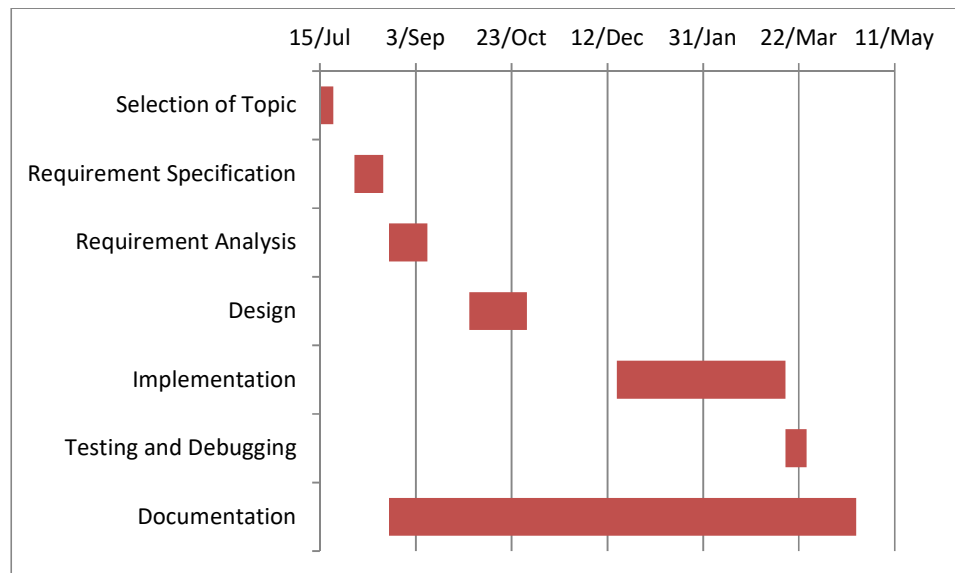


Table 3.3 Gantt Chart (Contd.)

### **3.4. Software and Hardware Requirements**

#### **3.4.1. Software Requirements**

We developed this project using the Python programming language [8]. Python is an open source programming language and is platform independent. As a result, our project can be executed on any operating system. We had to download some additional libraries to aid us in handling two dimensional arrays and read from and write to excel files. The “numpy” [9] library helped us in handling two dimensional arrays and executing advanced array operations and vectorized operations, while the “openpyxl” [10] library helped us to read from and write to excel files as the benchmark file was in excel format. The “random” [11] library was used to generate random numbers, which were used for list slicing operations as well as calculation of other essential values. The “sys” [12] library was used to enable us to display complete n-dimensional arrays. The “itertools” [13] library was used to generate permutations of sequences. We have also imported various classes from the tkinter [14] (python’s GUI) package to develop the user interface. We used Sublime Text 3 to write the code. We used the Anaconda Python distribution, which includes hundreds of popular Python libraries, as this enabled us to use all the required libraries without installing each one individually.

Therefore, to execute the program, the following software and packages are required on a user’s computer:

Python 3.0 or higher with the following libraries – numpy, random, openpyxl, itertools, tkinter. Anaconda distribution of Python 3 will automatically satisfy all these requirements.

Since Python is platform-independent, it can be executed on any operating system.

#### **3.4.2. Hardware Requirements**

The 3 members of this group have executed this project on different personal computers with different specifications. The specifications of the personal computers are as follows:

Member 1: 5<sup>th</sup> Generation Intel Core i7 Processor, 16GB RAM, 2.40 GHz

Member 2: AMD A8 Quad Core Processor, 4GB RAM, 2 GHz,

Member 3: 4<sup>th</sup> Generation Intel core i3 Processor, 4GB RAM, 2.60 GHz

To execute the program, a computer will require (approximately) the following specifications:

A dual core 2 Ghz processor,

2GB RAM.

### **3.5. Preliminary Product Description**

The project will take as input the various job-machine values, create the job matrix using the corresponding timeseed value and then calculate the makespan of the respective job matrix. We then proceed to use natural algorithms to optimize the makespan of such a matrix. Now, the project requires that the main program file, the Taillard's benchmark [15] excel file and the genetic program file, which we have developed, to be in the same folder, since the main program will read from and write to the excel file and import the various genetic algorithm techniques from the genetic file. The executable main program will present the user with an interface. The interface will display relevant information such as the current job, current number of machines, corresponding timeseed value and the calculated makespan value for such a combination. The user can also enter the specific row number to jump to the respective job-machine pair and get the optimal makespan value for such a pair. Now, the user can also use the previous and next buttons to navigate to the other job-machine pairs. The optimal sequence and the job matrix will be displayed on the command prompt as they will fluctuate and be too big to be properly fitted into the interface.

### 3.6. Conceptual Models

The following table shows the basic flow chart relevant to our project.

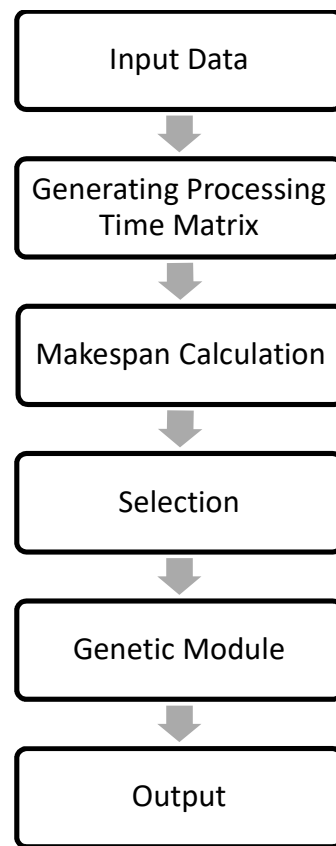


Table 3.4 Basic Flow Chart

This program deals with pure flow shop scheduling that only supports static job allocation. The processing time of each job on each machine is constant. Here we input a set of jobs that must be executed in a particular static sequence on a set of machines. The output to the problem is an optimum sequence.

## 4. SYSTEM DESIGN

### 4.1. Basic Modules

Our project deals with the optimization of the scheduling of the above mentioned jobs on machines. We shall be using natural algorithms to obtain an optimal schedule. Our use of natural algorithms enables us to determine the best solution and, unlike heuristic methods, natural algorithms operate on a population of solutions rather than a single solution. We will implement flow shop scheduling with makespan computation based on efficiency of different jobs and machines using natural algorithms. Comparison of the outputs with previously generated outputs by different researchers will be done and improvise the time complexity based on the Average Relative Percentage Difference (ARPD).

The program will begin with a set of operations and starting times. The input to the problem will be a set of jobs taken in an array which needs to be split to a number of operations and assigned to different machines. We will be using various crossover, mutation and selection techniques to optimize the makespan computation value. We employ natural algorithms to improve the efficiency of the system by using makespan computation. Efficiency of the obtained algorithm needs to be compared with existing techniques to evaluate its worth. Based on the ARPD, the efficiency obtained should be an improvement upon previous works. In doing so, the following processes will be used.

The program consists of the following basic modules :-

#### 4.1.1. Random Number Generator

The implementation of the random number generator [16] we have used which is based on the recursive formula  $X_{i+1} = (16807 X_i) \bmod (2^{31} - 1)$ . This implementation uses only 32-bit integers and provides a uniformly distributed sequence of numbers between 0 and 1 (not contained) :

0) Initial seed and  $X_0$  ( $0 < X_0 < 2^{31} - 1$ ) constants :  $a = 16807$ ,  $b = 127773$ ,  $c = 2836$ ,  $m = 2^{31} - 1$

1) Modification of  $k := X_i/b$  the seed :  $X_{i+1} := a(X_i \bmod b) - kc$  If  $X_{i+1} < 0$  then let  $X_{i+1} := X_{i+1} + m$

2) New value of the seed :  $X_{i+1}$  Current value of the generator :  $X_{i+1}/m$

Below, we shall denote by  $U(0,1)$  the pseudorandom number that this generator provides. We have  $0 < U(0,1) < 1$  for every generated number. With the help of the random number generator function the seed value is generated.

Random Matrix Generator- A random matrix generator function is written to generate a time matrix for machine\*jobs matrix with the help of the seed value produced by the random number generator function .

#### 4.1.2. Makespan Calculator Module

One of the primary modules is the makespan calculator module. Makespan is defined as the time required to complete all the tasks. The process of makespan computation is explained below with an example.

Let us consider a jobs\*machine matrix having 5 rows and 4 columns each cell representing the total time required by each job to complete using each machine . Makespan calculation is shown in the following tables.

	M1	M2	M3	M4
J1	8	3	8	2
J2	5	7	4	1
J3	7	7	3	2
J4	6	4	9	3
J5	4	7	8	7

Table 4.1: Explaining makespan computation

Let us imagine the sequence to be J2,J5,J1,J4,J3.

The makespan of the above sequence would be calculated as follows .

J2 5	5 7	12 4	16 1	17
J5 4	9 7	19 8	27 7	34
J1 8	17 3	22 8	35 2	37
J4 6	23 4	27 9	44 3	47
J3 7	30 7	37 3	47 2	49

Table 4.2: Explaining makespan computation (contd.)

Here the second job ie J2 is first in line and takes 5 units to complete in machine M1. The second job in line is J5 and it takes 5(waits for the previous task to be completed)+4=9 units in machine M1. The job J2 would have taken 7 units of time independently in machine M2 and hence takes 12 units (5(waiting time for the job to stop using machine M1 ) +7)=12 units. The second job J5 takes 19 units in machine 2 M2 because  $12 > 9$  we add  $12 + 7 = 19$  units. The time required by the last job in the sequence ie J3 to finish processing in machine M4 is 49 . The value is affected by the cumulative time required by each of the preceding jobs in each machine and hence also on the sequence in which the jobs are arranged. This value 49 is the makespan value for the particular sequence J2,J5,J1,J4 and J3. So to sum up the procedure of makespan calculation in the program module we have calculated the first row and the first column followed by completing the rest with the formula  $b[i,j] = a[i,j] + (b[i - 1,j] \text{ if } b[i - 1,j] > b[i,j - 1] \text{ else } b[i, j - 1])$  where i refers to the number of machines and j for jobs.

### 4.1.3. Genetic Module

We have used Roulette Wheel Selection [16], or fitness-proportionate selection, which ensures that the fitter an individual solution is, the likelier it is for it to be selected. In this case, each individual in the population is a permutation sequence, and the lower its makespan is, the fitter it is. In fact the measure of fitness used for each individual is the normalized value of the multiplicative inverse of the makespan. Thus, when Roulette Wheel Selection is done, the better sequences get chosen a large number of times, while the less fit sequences get chosen more rarely. This results in a fitter population, making it more probable for the optimal values to be obtained.

All of these functions have been defined in the `genetic.py` file.

In roulette wheel selection usually a proportion of the wheel is assigned to each of the possible selections based on their fitness value. This could be achieved by dividing the fitness of a selection by the total fitness of all the selections, thereby normalizing them. Then a random selection is made similar to how the roulette wheel is rotated.

While candidate solutions with a higher fitness will be less likely to be eliminated, there is still a chance that they may be. Contrast this with a less sophisticated selection algorithm, such as truncation selection, which will eliminate a fixed percentage of the weakest candidates. With fitness proportionate selection there is a chance some weaker solutions may survive the selection process; this is an advantage, as though a solution may be weak, it may include some component which could prove useful following the recombination process. The sequences having lower makespan values are considered to be fitter.

A Mutation [6] operation introduces minor changes in an individual, usually by modifying the sequence in very few random positions. We have used two types of mutation - Inverse mutation, in which a segment between two randomly decided positions is reversed, and Pairwise Swap mutation, in which two randomly selected elements in a sequence are exchanged. Mutation makes small but often significant changes in individuals in a population, thus ensuring diversity in a population, while also preventing the solution from getting stuck in local optima. Inverse and pairwise swap mutation are used to generate permutable sequences for which makespan is calculated and compared. Inverse mutation is defined as an operation in which a chromosome segment is clipped out, turned upside down, and reinserted back into the chromosome. A chromosome inversion can be inherited from one or both parents, or it may be a mutation that appears in a child whose family has no history of chromosome inversion. Here the start and end indices are generated randomly. The module contains another sequence generator function implementing pairwise swap mutation. After applying sort and reduce techniques to sequences generated by the mutation and selection module genetic method ordered crossover is applied to select M best sequences from the list ( here M=20) and add to the list.



A Crossover [5] operation combines parts of two individuals, known as parents, to obtain a new individual, known as a child. Crossovers result in a mixing of characteristics of different individuals; if the good parts of two parents are combined, it may result in a fitter child. We have used Ordered Crossover, which maps a portion (between two randomly generated crossover points) of one parent sequence directly to the child, and then fills up the rest of the child by copying the elements according to the order of the second parent, omitting the ones already present. We use ordered crossover on every pair of sequences in a population to generate a large number of new solutions, which may have very different characteristics, and thus may rapidly improve the solutions.

The program starts by reading jobs, machines and timeseed values from an input file and stores them in three lists. Initial Sequence of  $n$  jobs are sorted in ascending order of timeseed values. The first 4 jobs are selected and permuted to  $\text{fact}(4)$  sequences i.e. 24 sequences. Set  $i = 4$  and the list of Jobs are then arranged in ascending order of time seed value. Out of these sequences the best  $M$  sequences are selected where  $M \leq 24$ . The selected sequences are cloned to  $2M$  sequences using roulette wheel principle. The basic part of the selection process is to stochastically select from one generation to create the basis of the next generation. The requirement is that the fittest individuals have a greater chance of survival than weaker ones. This replicates nature in that fitter individuals will tend to have a better probability of survival and will go forward to form the mating pool for the next generation. Weaker individuals are not without a chance. In nature such individuals may have genetic coding that may prove useful to future generations. Again  $M$  sequences are selected from the resulting sequences. For each of the  $M$  sequences Partially Mapped Crossover is applied with remaining  $(M-1)$  sequences creating  $M \times (M-1)$  sequences. In this method, two crossover points are selected at random and PMX proceeds by position wise exchanges. The two crossover points give matching selection. It affects cross by position-by-position exchange operations. In this method parents are mapped to each other, hence we can also call it partially mapped crossover. Again  $M$  sequences are selected out of the resulting sequences. After this Ordered Crossover is applied for each of the  $M$  sequences with remaining  $M-1$  sequences creating  $M \times (M-1)$  sequences. The best  $M$  sequences are selected by the selection module and with the next iteration the subsequent job in the queue is added. Improvement if any is recorded Comparison of the outputs with previously generated outputs by different researchers will be done and improvise the time complexity based on the Average Relative Percentage Difference (ARPD).

## 4.2. Procedural Design

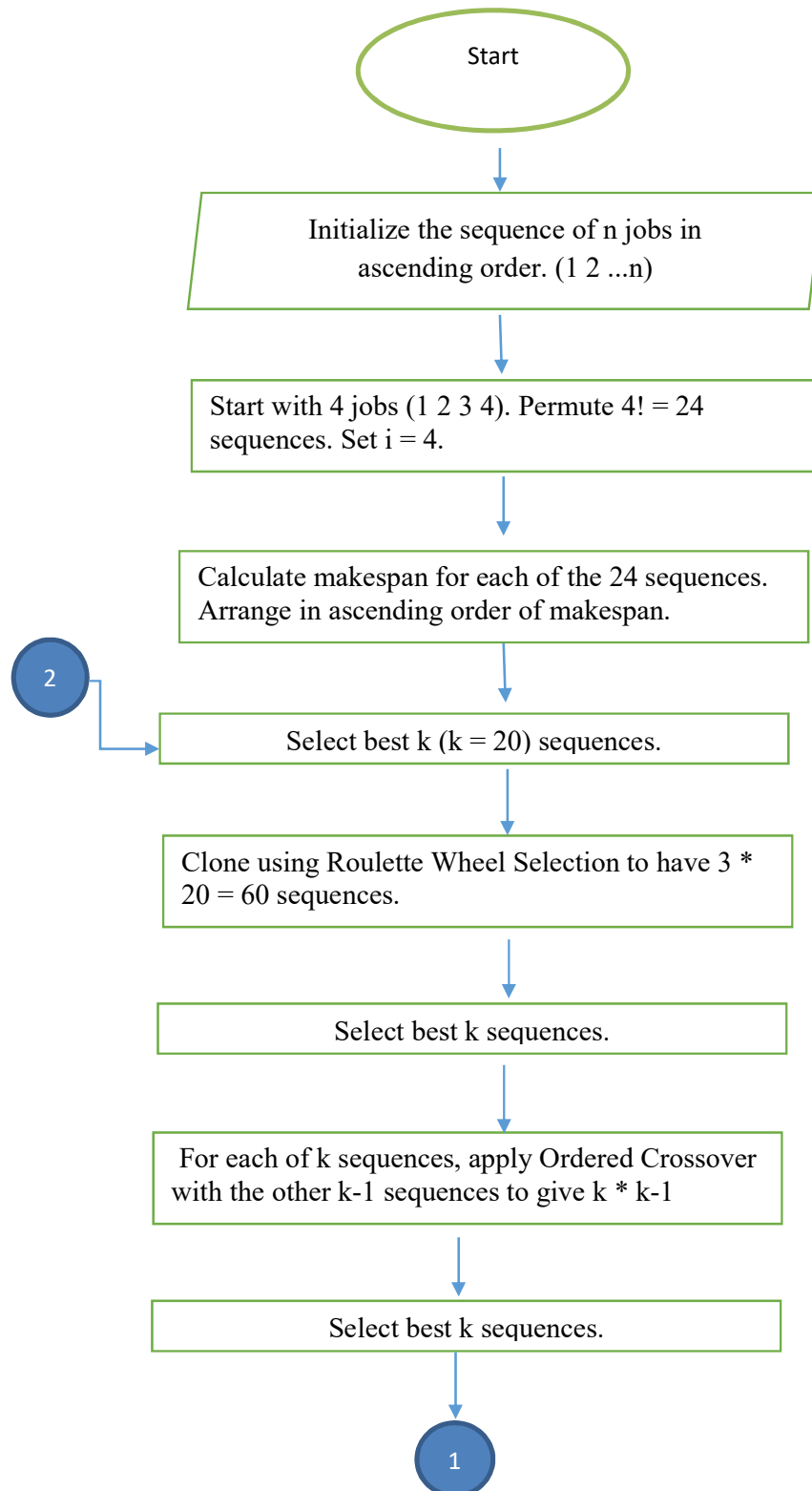
### 4.2.1. Data Structures

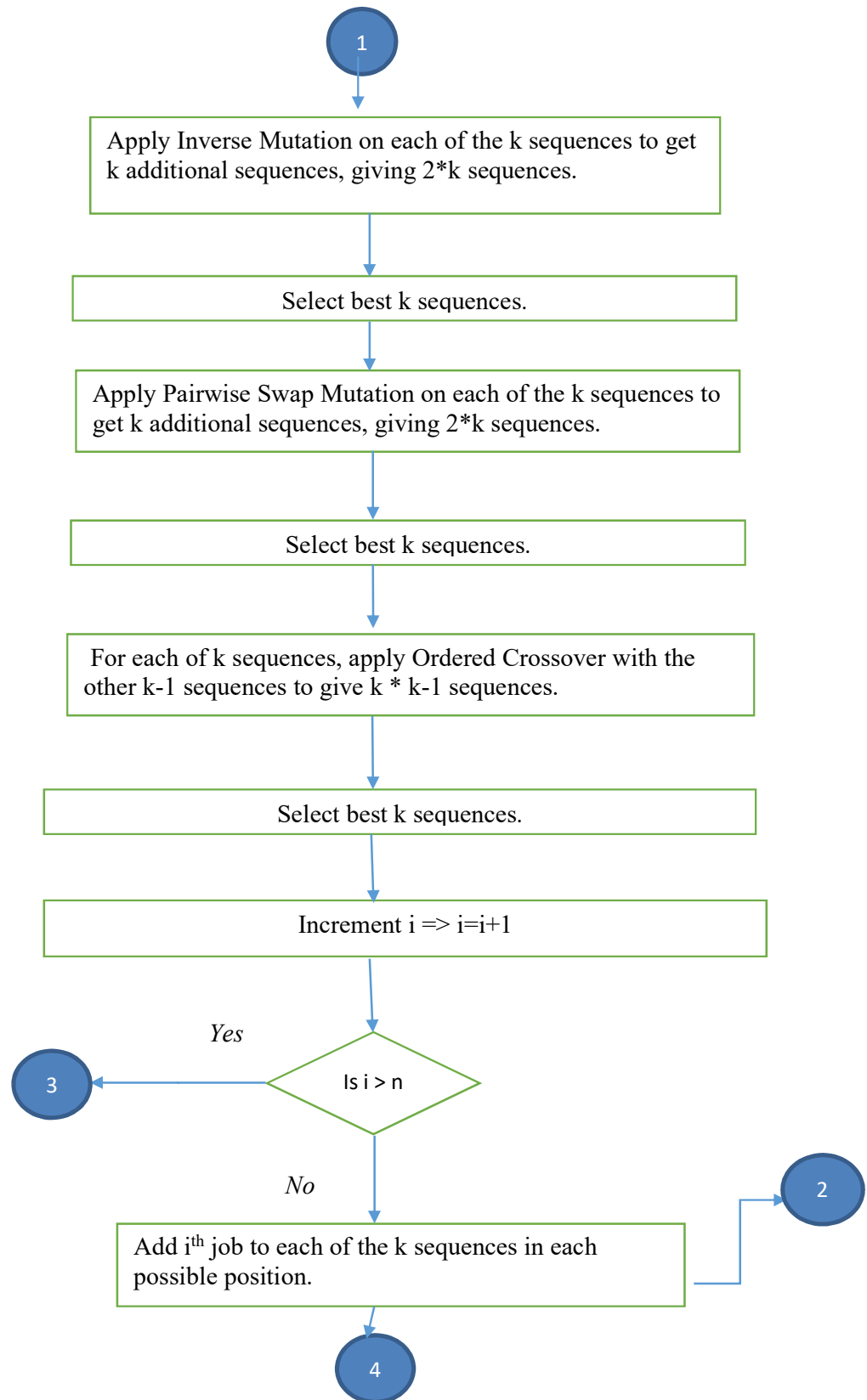
The following data structures are used on the implementation of our program:

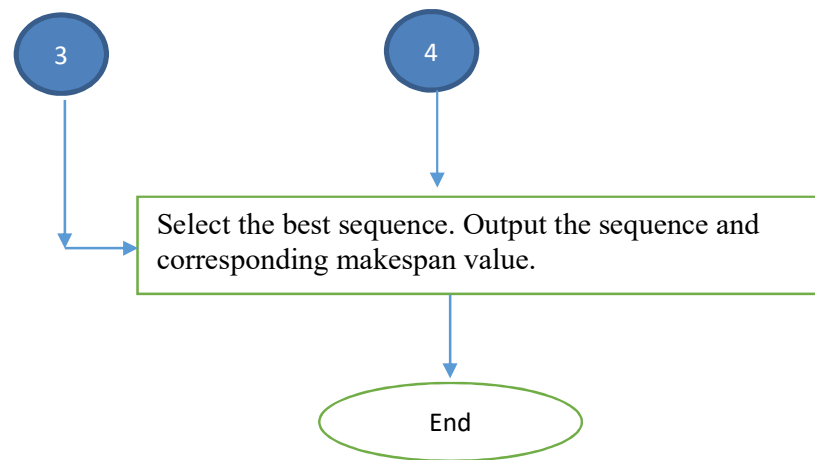
1. Three arrays have been used to store the problem instances from the benchmarks. One array stores all the numbers of jobs of each problem, the second array stores the numbers of machines, and the third array stores the seed values used for the random number generator [16].
2. For each subproblem, for the given number of jobs and machines, a processing time matrix is generated, which is stored in the form of a 2-dimensional matrix. This 2D matrix is implemented as a numpy [9] ndarray data structure.
3. For each subproblem, the population of sequences at any point is stored as a list of arrays, where each array is a sequence of jobs. Each sequence is implemented as a numpy array.
4. Another list stores all the makespan values corresponding to each sequence in the list of sequences.

#### 4.2.2. Control Flow Chart

The diagram below depicts a graphical representation of the program in relation to its sequence of functions.







#### 4.2.3. Algorithmic Design

The following is a formal description of the procedure followed by the proposed heuristic technique -

1. Initialize the sequence of  $n$  jobs in ascending order. (1 2 ... $n$ )
2. Start with 4 jobs (1 2 3 4). Permute  $4! = 24$  sequences. Set  $i = 4$ .
3. Calculate makespan for each of the 24 sequences. Arrange in ascending order of makespan.
4. Select best  $k$  ( $k = 20$ ) sequences.
5. Clone using Roulette Wheel Selection to have  $3 * 20 = 60$  sequences.
6. Select best  $k$  sequences.
7. For each of  $k$  sequences, apply Ordered Crossover with the other  $k-1$  sequences to give  $k * k-1$  sequences.
8. Select best  $k$  sequences.

9. Apply Inverse Mutation on each of the  $k$  sequences to get  $k$  additional sequences, giving  $2*k$  sequences.
10. Select best  $k$  sequences.
11. Apply Pairwise Swap Mutation on each of the  $k$  sequences to get  $k$  additional sequences, giving  $2*k$  sequences.
12. Select best  $k$  sequences.
13. For each of  $k$  sequences, apply Ordered Crossover with the other  $k-1$  sequences to give  $k * k-1$  sequences.
14. Select best  $k$  sequences.
15. Increment  $i$ . If  $i > n$ , go to Step 18.
16. Add  $i^{\text{th}}$  job to each of the  $k$  sequences in each possible position.
17. Goto step 4.
18. Select the best sequence. Output the sequence and corresponding makespan value.

#### 4.2.4. Input Data

The input to the program consists of a set of jobs taken in an array which needs to be split to a number of operations and assigned to different machines. As of now we have not accommodated user inputs. Since we intend to compare our results with Taillard's scheduling benchmarks [15], each problem that we solve corresponds to one instance in the set of benchmarks. To reproduce these instances, we need to use the same random number generator [16] according to Taillard's benchmarks. To ensure reproducibility, all we need to do is to use the seed values used and that will ensure we generate the same processing time matrices for each problem.

Thus, for each problem, the inputs are -

1. The number of jobs  $n$
2. The number of machines  $m$
3. The seed value, which is used by the random number generator to generate the processing time matrix.

The outputs are -

1. The minimized makespan value using the heuristic
2. The sequence of jobs for which the makespan is minimal.

The procedure we use begins with the first four jobs (we can begin with any four jobs, without any loss of generality), and generates all possible permutations ( $4! = 24$  permutations) of those jobs. This forms our initial population of sequences. From this point onwards, after each operation, we reduce the population to the  $k$  (we considered  $k = 20$ ) best (lowest makespan value) sequences, so that the solution space remains manageable. We use a combination of genetic algorithms as shown in the procedure below to generate new sequences. Once this is done, we add the next job in each possible position of each sequence in the population, and repeat this entire process iteratively until all the jobs have been taken into account. Finally, the sequence with the minimum makespan is isolated, and its makespan value is output and compared to the benchmark.

#### **4.2.5. Output Data**

Our principal means of measuring the efficacy of the proposed heuristic is by comparing our obtained results against the benchmarks defined by Taillard [15]. Since the heuristic is non-deterministic, we ran our program multiple times to improve the odds of getting better results, until a few successive executions showed no further improvement. Finally, as a quantitative measure of the amount of improvement brought about by this method, we have calculated the Average Relative Percentage Difference (ARPD). Having done so, we observe that the new method performs significantly better for larger problems than for small ones. For the set of problems with the largest number of jobs, that is, Taillard's instances 111 to 120, the overall ARPD turns out to be negative, which indicates that the technique results in better makespans on average than the benchmark values for the largest problem size.

### 4.3. User Interface Design

We will be using natural algorithms to obtain an optimal schedule on the basis of makespan computation based on efficiency of different jobs and machines to find a proficient job scheduling strategy. The program will take jobs, number of machines and time seed as input and calculate the make-span value. It will then use natural algorithm operators such as crossovers, mutation and selection to try and improve the calculated make-span. The integrated program takes inputs of jobs and machines and outputs optimal permutation schedules. The outputs of previously developed algorithms are available to us, and we expect to obtain improved results on the basis of the ARPD. The program will begin with a set of operations and starting times. The three set of inputs to the problem being machines, jobs and time seed. The output would be a sequence of jobs assigned to the set of machines resulting in minimum make-span. The objective of the project is to improve the make-span value obtained from previous research and to infer if the value obtained from this experiment is better than the previous make-span value. We will simply use the previously considered values and check if we have succeeded in minimizing the make-span value. Comparison of the outputs with previously generated outputs by different researchers will be done and improvise the time complexity based on the Average Relative Percentage Difference (ARPD).

We will be using TKinter [14], Python's standard GUI package to design the user interface. Figure 1 demonstrates a basic user interface that we have designed. The user interface consists of a field where the Taillard instance that we want to execute can be entered. There is also a button to initiate the program and another two buttons to roll through the process, i.e. move to the previous or next problem. With machines, jobs and time seed as input a machine \* job matrix would be created in the process and the final computation would result in a sequence having a relatively more optimum makespan value than previously generated values. A button may be designed which would display the measure of improvement when clicked. By clicking on the exit button the user will be able to withdraw from the environment.

Since the size of the processing time matrix and the job sequence can be very large, we have not displayed them in the graphical interface. Instead, they are printed in the Python command line window. The figure below depicts the user interface.



The screenshot shows a software window titled "Scheduling Optimization". Inside the window, there is a "Problem No." input field containing the number "1". To the right of this field is a "Calculate" button. Below the input field, the following parameters are displayed: "Jobs: 20", "Machines: 5", "Timeseed: 379008056", and "Optimal makespan: 1383.0". At the bottom of the window, there are two buttons: "Previous" and "Next". The "Next" button is styled with a dashed border, while the "Previous" button has a solid border.

Figure 4.1: Front End Design

#### 4.4. Test Cases Design

Since we are running our algorithm with respect to the Taillard benchmarks [15] and comparing all our results against them, we do not need to design any new test cases. The test cases are the instances of the benchmarks.

The benchmarks consist of 120 problems of different sizes. Associated with each subproblem is a seed used for the random number generator [16]. The problems range from 20 to 500 jobs and from 5 to 20 machines. Each problem size with 20, 50 or 100 jobs has 10 instances, each with different timeseed values. The problems with 200 jobs have 10 or 20 machines, and the largest problem size has 500 jobs and 20 machines. Thus, in total, there is a total of 120 instances, each of which acts as one test case on which the algorithm can be run.

The details of the test cases are shown in the following table:

Job	M/C	Time seed
20	5	873654221
20	5	379008056
20	5	1866992158
20	5	216771124
20	5	495070989
20	5	402959317
20	5	1369363414
20	5	573109518
20	5	88325120
20	10	587595453
20	10	1401007982
20	10	873136276
20	10	268827376
20	10	1634173168
20	10	691823909
20	10	73807235
20	10	1273398721
20	10	2065119309
20	10	1672900551
20	20	479340445
20	20	268827376
20	20	1958948863
20	20	918272953
20	20	555010963
20	20	2010851491
20	20	1519833303
20	20	1748670931
20	20	1923497586
20	20	1829909967
50	5	1328042058
50	5	200382020
50	5	496319842
50	5	1203030903
50	5	1730708564
50	5	450926852
50	5	1303135678
50	5	1273398721
50	5	587288402
50	5	248421594
50	10	1958948863
50	10	575633267
50	10	655816003
50	10	1977864101
50	10	93805469
50	10	1803345551
50	10	49612559

50	10	1899802599
50	10	2013025619
50	10	578962478
50	20	1539989115
50	20	691823909
50	20	655816003
50	20	1315102446
50	20	1949668355
50	20	1923497586
50	20	1805594913
50	20	1861070898
50	20	715643788
50	20	464843328
100	5	896678084
100	5	1179439976
100	5	1122278347
100	5	416756875
100	5	267829958
100	5	1835213917
100	5	1328833962
100	5	1418570761
100	5	161033112
100	5	304212574
100	10	1539989115
100	10	655816003
100	10	960914243
100	10	1915696806
100	10	2013025619
100	10	1168140026
100	10	1923497586
100	10	167698528
100	10	1528387973
100	10	993794175
100	20	450926852
100	20	1462772409
100	20	1021685265
100	20	83696007
100	20	508154254
100	20	1861070898
100	20	26482542
100	20	444956424
100	20	2115448041
100	20	118254244
200	10	471503978
200	10	1215892992
200	10	135346136
200	10	1602504050
200	10	160037322
200	10	551454346
200	10	519485142

200	10	383947510
200	10	1968171878
200	10	540872513
200	20	2013025619
200	20	475051709
200	20	914834335
200	20	810642687
200	20	1019331795
200	20	2056065863
200	20	1342855162
200	20	1325809384
200	20	1988803007
200	20	765656702
500	20	1368624604
500	20	450181436
500	20	1927888393
500	20	1759567256
500	20	606425239
500	20	19268348
500	20	1298201670
500	20	2041736264
500	20	379756761
500	20	28837162

Table 4.3 : Test Cases

## 5. IMPLEMENTATION AND TESTING

### 5.1. Plan of Implementation

We started developing and designing the genetic module first. The function flow of the genetic module is of the following order – selection [7], crossover [5] and mutation [6]. We have used Roulette Wheel Selection [16] as a selection method. It is basically a fitness-proportionate selection, which ensures that the fitter an individual solution is, the likelier it is for it to be selected. In this case, each individual in the population is a permutation sequence, and the lower its makespan is, the fitter it is. In fact the measure of fitness used for each individual is the normalized value of the multiplicative inverse of the makespan. Thus, when Roulette Wheel Selection is done, the better sequences get chosen a large number of times, while the less fit sequences get chosen more rarely. This results in a fitter population, making it more probable for the optimal values to be obtained.

A Crossover operation combines parts of two individuals, known as parents, to obtain a new individual, known as a child. We have used Ordered Crossover, which maps a portion (between two randomly generated crossover points) of one parent sequence directly to the child, and then fills up the rest of the child by copying the elements according to the order of the second parent, omitting the ones already present. We use ordered crossover on every pair of sequences in a population to generate a large number of new solutions, which may have very different characteristics, and thus may rapidly improve the solutions. We have designed and tested several types of crossover techniques such as ordered and cyclic crossover.

A Mutation operation introduces minor changes in an individual, usually by modifying the sequence in very few random positions. We have designed and tested several mutation techniques such as inverse mutation, pair-wise mutation and insert mutation. Inverse mutation, in which a segment between two randomly decided positions is reversed, and Pairwise Swap mutation, in which two randomly selected elements in a sequence are exchanged. In insert mutation two random mutation points are selected and one allele at the end point of the randomly selected patch is inserted after the start point of the selected patch.

We designed the random number generator [16] after designing and testing the genetic module. The formula described in the code is the same random number generator used by Taillard for the benchmark instances. We have designed the random number generator module in this way and have checked

if the order of the output tallies with Taillard's values. After implementing the random number generator, we have used it to generate the processing time matrices for each problem instance. A random matrix generator function is written to generate a machine\*jobs matrix with the help of the seed value. After this we proceeded to write a function which will calculate the makespan for a specific sequence with respect to the processing time matrix.

Once these basic modules are in place, the entire optimization procedure is implemented using the iterative approach that we intend to use. Thus, the program loops over all the problem instances, generating the processing time matrices for each, and using the genetic operations to optimize the sequences for each.

Since we have developed our project in Python, we have used TKinter [14], Python's standard GUI package to design the user interface. Figure 1 demonstrates a basic user interface that we have designed. The user interface consists of a button which initiates the program and two buttons to roll through the process. With machines, jobs and time seed as input, a machine \* job matrix would be created in the process and the final computation would result in a sequence having a relatively more optimum makespan value than previously generated values. This is displayed in the command prompt. The optimum makespan value will be displayed in the GUI itself. The user can also jump to a specific instance, where he can enter the problem number and the respective optimum makespan is calculated and displayed. By clicking on the exit button the user will be able to withdraw from the environment. We have also written the makespan values to an Excel file, where the ARPD computation has been done.

## 5.2. Code Details and Efficiency

The following is a formal description of the procedure followed by the proposed heuristic technique -

1. Initialize the sequence of n jobs in ascending order. (1 2 ...n)
2. Start with 4 jobs (1 2 3 4). Permute  $4! = 24$  sequences. Set  $i = 4$ .
3. Calculate makespan for each of the 24 sequences. Arrange in ascending order of makespan.
4. Select best k ( $k = 20$ ) sequences.
5. Clone using Roulette Wheel Selection to have  $3 * 20 = 60$  sequences.

6. Select best  $k$  sequences.
7. For each of  $k$  sequences, apply Ordered Crossover with the other  $k-1$  sequences to give  $k * k-1$  sequences.
8. Select best  $k$  sequences.
9. Apply Inverse Mutation on each of the  $k$  sequences to get  $k$  additional sequences, giving  $2*k$  sequences.
10. Select best  $k$  sequences.
11. Apply Pairwise Swap Mutation on each of the  $k$  sequences to get  $k$  additional sequences, giving  $2*k$  sequences.
12. Select best  $k$  sequences.
13. For each of  $k$  sequences, apply Ordered Crossover with the other  $k-1$  sequences to give  $k * k-1$  sequences.
14. Select best  $k$  sequences.
15. Increment  $i$ . If  $i > n$ , go to Step 18.
16. Add  $i$ th job to each of the  $k$  sequences in each possible position.
17. Goto step 4.

The main functional modules are the genetic module, the random number generator module, makespan calculating module and the processing time matrix. The code snippets of these modules are added below.

### 5.2.1. Ordered Crossover

```
def ordered_crossover(parent1, parent2):
    from random import randint
    jobs = len(parent1)
    #Generating start and end indices randomly
    start = randint(0, jobs - 2)
    end = randint(start + 1, jobs - 1)
    #Initialize child list
    child = [-1] * jobs

    #Copy the alleles between start-end from parent1
    child[start:end + 1] = parent1[start:end + 1]
    #Start from 2nd crossover point and copy the rest of the elements in
    #order, cyclically
    parent_index = (end + 1) % jobs
    child_index = (end + 1) % jobs
```

```

while child_index != start:
    if parent2[parent_index] not in child:
        child[child_index] = parent2[parent_index]
        child_index = (child_index + 1) % jobs
    parent_index = (parent_index + 1) % jobs
return child

```

### 5.2.2. Mutations (Inverse, Insert, Pairwise-swap)

```

def inverse_mutation(sequence):
    from random import randint
    #Inverts slice between start and end
    jobs = len(sequence)
    #Generating start and end indices randomly
    start = randint(0, jobs - 2)
    end = randint(start + 1, jobs - 1)
    import numpy as np
    sequence[start:end] = np.fliplr([sequence[start:end]])[0]
    return sequence

def insert_mutation(sequence):
    from random import randint
    #Pops out element at end position and places it next to start
    jobs = len(sequence)
    #Generating start and end indices randomly
    start = randint(0, jobs - 2)
    end = randint(start + 1, jobs - 1)

    import numpy as np
    t = sequence[end]
    np.delete(sequence, end)
    np.insert(sequence, start + 1, t)
    #sequence[start + 1:end + 1] = [sequence[end]] + sequence[start +
1:end]
    return sequence

def pairwise_swap_mutation(sequence):
    from random import randint
    #Swaps start and end
    jobs = len(sequence)
    #Generating start and end indices randomly
    start = randint(0, jobs - 2)

```



```

end = randint(start + 1, jobs - 1)

sequence[start], sequence[end] = sequence[end], sequence[start]
return sequence

```

### 5.2.3. Random Number Generator

```

def rng(seed, low, high):
    # Random number generator, as in Taillard's benchmarks paper
    m = 2147483647 # 2^31 - 1
    a = 16807
    b = 127773
    c = 2836
    k = int(seed / b)
    seed = a * (seed % b) - k * c # Update seed
    if (seed < 0):
        seed = seed + m
    val = float(seed) / m # Random number between 0 and 1
    randnum = low + int(val * (high - low + 1)) # Between low and high
    return randnum, seed

```

### 5.2.4. Makespan Calculation

```

def calculate_makespan(a, seq):
    # Order the jobs (rows) in order of the sequence
    a = a[seq]

    # Refer to Makespan.jpg for the procedure
    b = np.zeros(a.shape)
    jobs = a.shape[0]
    macs = a.shape[1]

```

```

b[0,0] = a[0,0]
# Build first row
for i in range(1, macs):
    b[0,i] = a[0,i] + b[0,i - 1]
# Build first column
for i in range(1, jobs):
    b[i,0] = a[i,0] + b[i - 1,0]
# Build the rest
for i in range(1, jobs):
    for j in range(1, macs):
        b[i,j] = a[i,j] + (b[i - 1,j] if b[i - 1,j] > b[i,j - 1] else b[i, j
- 1])

return int(b[-1, -1])

```

### 5.2.5. Code Efficiency

It is very slow to deal with multi-dimensional lists .Hence we have used the numpy [9] library, which enables us to easily manipulate multi-dimensional lists.

Since library functions are usually way more optimized, so execution is usually faster and more efficient. Since our problem is very computationally intensive, using library function will make a very significant improvement in total execution time.

We have also used the itertools [13] library which helps us to permute a sequence of k jobs.

## **5.3. CODE TESTING**

### **5.3.1. Unit Testing**

#### **5.3.1.1. Genetic Module**

##### **5.3.1.1.1. Crossover Function**

We use ordered crossover [5] on every pair of sequences in a population to generate a large number of new solutions, which may have very different characteristics, and thus may rapidly improve the solutions. The ordered crossover function takes two sequences as input (parents) and generates a child sequence as output. We have tested this function using random sequences. The figure below depicts test cases for the crossover function.



```

[Anaconda3] D:\Dropbox\Project>python genetic.py
Parents:
[7, 9, 4, 8, 0, 2, 3, 6, 5, 1] [0, 8, 6, 4, 1, 3, 7, 2, 5, 9]
Child after ordered crossover:
[8, 4, 1, 7, 0, 2, 3, 6, 5, 9]

[Anaconda3] D:\Dropbox\Project>python genetic.py
Parents:
[3, 2, 4, 9, 0, 8, 7, 5, 6, 1] [0, 4, 9, 8, 2, 7, 3, 6, 1, 5]
Child after ordered crossover:
[0, 4, 9, 8, 2, 7, 3, 5, 6, 1]

[Anaconda3] D:\Dropbox\Project>python genetic.py
Parents:
[4, 6, 3, 8, 2, 1, 5, 7, 0, 9] [5, 9, 8, 2, 3, 6, 0, 1, 7, 4]
Child after ordered crossover:
[9, 6, 3, 8, 2, 0, 1, 7, 4, 5]

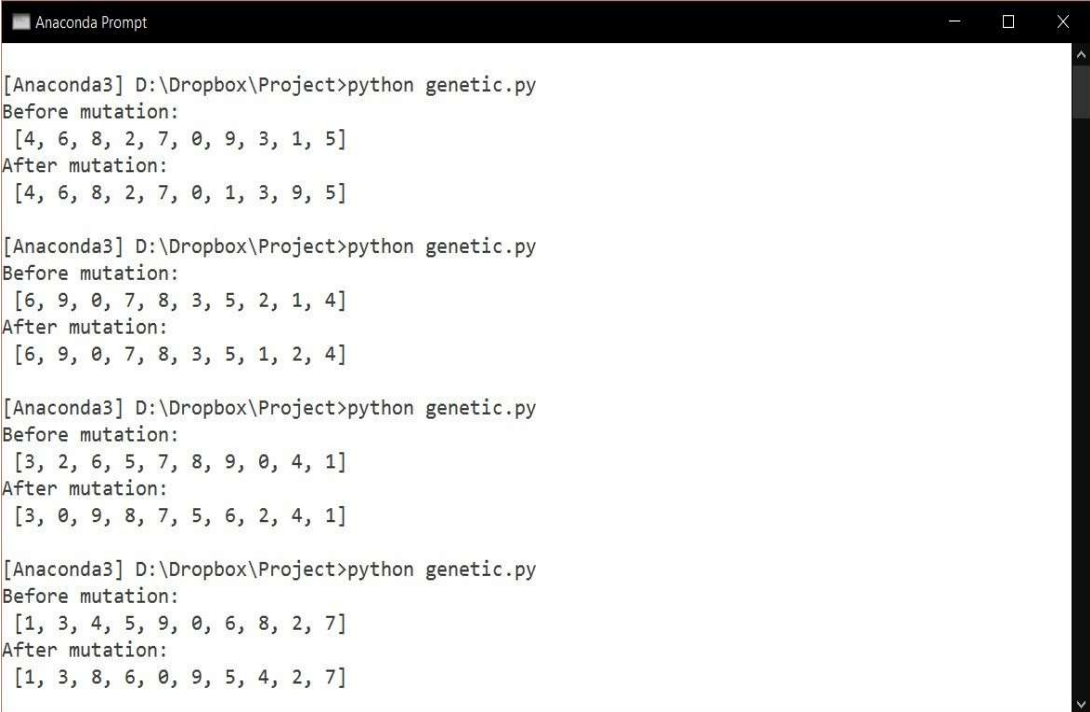
[Anaconda3] D:\Dropbox\Project>python genetic.py
Parents:
[2, 5, 8, 9, 4, 0, 6, 3, 1, 7] [8, 0, 5, 9, 4, 2, 1, 7, 3, 6]
Child after ordered crossover:
[5, 2, 8, 9, 4, 0, 6, 3, 1, 7]

```

Figure 5.1 : Crossover Function

#### 5.3.1.1.2. Mutation Function

Mutation [6] makes small but often significant changes in individuals in a population, thus ensuring diversity in a population, while also preventing the solution from getting stuck in local optima. We have implemented Inverse Mutation and Pairwise Swap Mutation in our project. The mutation function takes a sequence as input and gives a mutated sequence as output. The figure below depicts test cases for the mutation function.




```
[Anaconda3] D:\Dropbox\Project>python genetic.py
Before mutation:
[4, 6, 8, 2, 7, 0, 9, 3, 1, 5]
After mutation:
[4, 6, 8, 2, 7, 0, 1, 3, 9, 5]

[Anaconda3] D:\Dropbox\Project>python genetic.py
Before mutation:
[6, 9, 0, 7, 8, 3, 5, 2, 1, 4]
After mutation:
[6, 9, 0, 7, 8, 3, 5, 1, 2, 4]

[Anaconda3] D:\Dropbox\Project>python genetic.py
Before mutation:
[3, 2, 6, 5, 7, 8, 9, 0, 4, 1]
After mutation:
[3, 0, 9, 8, 7, 5, 6, 2, 4, 1]

[Anaconda3] D:\Dropbox\Project>python genetic.py
Before mutation:
[1, 3, 4, 5, 9, 0, 6, 8, 2, 7]
After mutation:
[1, 3, 8, 6, 0, 9, 5, 4, 2, 7]
```

Figure 5.2 : Mutation Function



```

[Anaconda3] D:\Dropbox\Project>python genetic.py
Before mutation:
[4, 2, 8, 3, 9, 0, 7, 6, 5, 1]
After mutation:
[4, 2, 8, 3, 9, 7, 0, 6, 5, 1]

[Anaconda3] D:\Dropbox\Project>python genetic.py
Before mutation:
[1, 4, 9, 7, 0, 6, 2, 3, 8, 5]
After mutation:
[1, 4, 9, 7, 0, 6, 2, 8, 3, 5]

[Anaconda3] D:\Dropbox\Project>python genetic.py
Before mutation:
[7, 2, 3, 5, 4, 6, 8, 9, 1, 0]
After mutation:
[7, 2, 4, 5, 3, 6, 8, 9, 1, 0]

[Anaconda3] D:\Dropbox\Project>python genetic.py
Before mutation:
[3, 0, 9, 4, 7, 1, 2, 6, 8, 5]
After mutation:
[3, 0, 9, 4, 7, 1, 6, 2, 8, 5]

```

Figure 5.3 : Mutation Function (contd)

### 5.3.1.2. Random Number Generator

A random matrix generator function is written to generate a time matrix for machine\*jobs with the help of the seed value produced by the random number generator [16] function. For testing purposes we have implemented a 10\*10 processing time matrix. We have checked every instance of the processing matrix with its corresponding Taillard's matrix. The figure below depicts test cases for the random number generator matrix.

```

[Anaconda3] D:\Dropbox\Project>python RNGtest.py
[[ 8. 38.  6. 50. 67. 59. 45. 65. 96. 92.]
 [ 3. 17. 65. 72. 62.  1. 49. 98. 16. 42.]
 [86. 72. 76. 37. 67. 29. 65. 19. 99. 90.]
 [21. 98. 53. 64. 82. 34. 39. 11. 38. 39.]
 [39. 50. 60. 56. 19. 86. 35.  8. 10. 12.]
 [94. 32. 74.  7. 65.  8. 18. 56.  9. 64.]
 [53. 23. 92. 33. 42. 76. 45. 33. 64. 49.]
 [ 8. 42. 81. 18. 47. 46. 80. 84. 15.  5.]
 [18. 65. 68. 55. 47. 43. 25. 12. 58. 72.]
 [94. 22. 16. 73. 14. 81. 80. 11. 18. 94.]]

[Anaconda3] D:\Dropbox\Project>python RNGtest.py
[[ 8. 38.  6. 50. 67. 59. 45. 65. 96. 92.]
 [ 3. 17. 65. 72. 62.  1. 49. 98. 16. 42.]
 [86. 72. 76. 37. 67. 29. 65. 19. 99. 90.]
 [21. 98. 53. 64. 82. 34. 39. 11. 38. 39.]
 [39. 50. 60. 56. 19. 86. 35.  8. 10. 12.]
 [94. 32. 74.  7. 65.  8. 18. 56.  9. 64.]
 [53. 23. 92. 33. 42. 76. 45. 33. 64. 49.]
 [ 8. 42. 81. 18. 47. 46. 80. 84. 15.  5.]
 [18. 65. 68. 55. 47. 43. 25. 12. 58. 72.]
 [94. 22. 16. 73. 14. 81. 80. 11. 18. 94.]]

```

Figure 5.4 : Random Number Generator Matrix

### 5.3.2. Integrated Testing

In this phase, we have integrated our separate modules and tested our project as a whole. For our project, we ensured that the output generated by our project either stays within the predefined upper and lower bounds or is within a respectable amount from the upper limit. The figure below depicts output screens for integrated testing.

```

Anaconda Prompt

[Anaconda3] D:\Dropbox\Project>python RUN3.py
[[ 71.  27.  55.  90.  11.  18.  42.  64.  73.  95.  22.  53.  32.   5.
   94.  12.  41.  85.  75.  38.]
 [ 13.  11.  73.  43.  27.  33.  57.  42.  71.   3.  11.  49.   8.   3.
   47.  58.  23.  79.  99.  23.]
 [ 61.  25.  52.  72.  89.  75.  60.  28.  94.  95.  18.  73.  40.  61.
   68.  75.  37.  13.  65.   7.]
 [ 21.   8.   5.   8.  58.  59.  85.  35.  84.  97.  93.  60.  99.  29.
   94.  41.  51.  87.  97.  11.]
 [ 91.  13.   7.  95.  20.  69.  45.  44.  29.  32.  94.  84.  60.  49.
   49.  65.  85.  52.   8.  58.]]
2016-05-10 17:17:48.950537
Jobs: 20
Machines: 5
Timeseed: 402959317
Optimal sequence: 13, 7, 19, 14, 4, 17, 3, 8, 6, 15, 11, 0, 9, 5, 12, 16, 10, 18, 2, 1
Optimal makespan: 1210.0

2016-05-10 17:17:51.137807

[Anaconda3] D:\Dropbox\Project>

```

Figure 5.5 : Integrated Testing

```

Anaconda Prompt

[Anaconda3] D:\Dropbox\Project>python RUN3.py
[[ 47.  77.  66.  13.  77.  20.  29.  11.  85.  98.  36.  92.  99.  65.
   34.  35.  42.  28.  28.   5.]
 [ 80.  46.  37.  86.  85.  67.  22.  67.  20.   1.  39.  25.   1.  41.
   74.  39.   2.  20.   3.  29.]
 [ 51.  79.  35.  63.  18.  67.  73.   2.  23.  59.  46.  12.  17.  87.
   35.  27.  71.  11.  97.  83.]
 [ 63.  22.  47.  46.  72.  16.  35.   2.  22.  50.  58.  46.  19.  82.
   69.  14.  94.  39.  62.  27.]
 [ 60.  20.  13.  64.  67.  25.  39.  40.  72.  28.  36.  60.   2.   8.
   33.  85.  51.   6.   5.  72.]
 [ 70.  96.  56.  22.  44.  83.  66.  56.  27.  52.  46.  83.  76.  25.
   28.  26.  98.  64.  66.  15.]
 [ 39.  75.  66.  89.  56.  42.  90.  77.  76.  43.  14.   3.  57.  98.
   47.  35.  58.   6.  95.   9.]
 [ 33.   1.  59.  85.   1.  85.  65.  47.  13.  12.  23.  21.  89.  51.
   34.  62.   6.  22.  75.  22.]
 [  5.  37.  72.   6.  90.  71.  47.  60.  93.  17.  65.  12.  97.  29.
   41.   3.  46.  33.  55.  10.]
 [  9.  14.  37.  54.  14.  53.  27.  64.  25.  79.  30.  33.   5.  96.
   1.  51.  42.  78.  70.  91.]]
2016-05-10 17:21:51.504631
Jobs: 20
Machines: 10
Timeseed: 691823909
Optimal sequence: 18, 7, 19, 6, 5, 8, 17, 14, 10, 13, 12, 15, 16, 3, 9, 4, 2, 1, 0, 11
Optimal makespan: 1463.0

2016-05-10 17:21:54.653472

```

Figure 5.6: Integrated Testing (2)



```

Anaconda Prompt
[ 45.  2. 39.  4. 46. 11. 66. 59. 28. 74. 93. 95. 48. 71.
 30. 22.  3. 22. 82. 35.]
[ 56. 14. 88. 41.  4. 52. 17. 95. 15. 85. 12. 11. 24. 71.
 46. 59. 16. 33. 53. 22.]
[ 49. 44. 76. 88. 45. 52. 31. 66. 17. 68. 99. 78. 87. 45.
 78. 51. 87.  4. 45.  9.]
[ 39. 12. 45. 34. 11. 71. 53. 28. 93. 61. 61. 78. 13. 49.
 15. 31. 93. 83. 20.  1.]
[ 13. 36.  6. 22. 44. 65. 87. 50. 96. 51. 48. 89. 69. 83.
 22. 26. 29. 67. 70. 59.]
[ 34. 18.  4. 65. 58. 35. 86. 34. 49. 88. 42. 79. 25. 18.
 75. 45. 54. 19. 80. 15.]
[ 22. 56. 63. 56. 79. 82. 92. 19. 45. 32. 11. 59. 22. 14.
 88. 82. 82. 71.  8. 13.]
[ 53. 71. 98. 99. 54. 20. 62. 18. 81. 60.  1. 93.  8. 92.
 60. 87. 11. 24. 11. 98.]
[ 40. 62.  8. 13. 87. 40. 27. 44. 41. 70. 10. 94. 25. 10.
 36. 87. 37. 86. 76. 70.]
[ 17. 92. 90. 58. 19. 97. 80. 28. 67. 59.  9. 92.  7. 19.
 82. 60. 60. 58. 74. 70.]
[ 72. 33. 71. 68. 94. 82. 46. 78. 37. 69.  7. 34. 69. 18.
 98.  8. 58. 86. 77. 50.]
[ 50. 82. 16. 17. 68. 65. 92. 37. 47. 70. 66. 30.  7. 37.
  5. 52. 86. 36. 29.  4.]
[ 99. 53. 52.  9.  8. 50. 62. 14. 21. 88.  7. 85. 62. 10.
 61. 92. 47. 77. 37. 96.]
[ 50.  4. 20. 19. 58. 22. 56. 92. 59. 34. 17. 39. 59.  7.
 59.  2. 70. 72. 90. 56.]
[ 26. 50.  4.  8. 60. 60. 80. 38. 51. 25. 97. 73. 46. 82.
 92. 42. 65. 12. 34. 23.]
[ 99. 48. 41. 40. 76. 89. 42. 90. 61. 25. 27. 60. 79. 50.
 15. 65. 80. 67. 70. 94.]
[ 61. 55.  2. 55. 78. 27. 13. 20. 38. 75. 23. 61. 37. 43.
 69.  2. 44. 25. 12. 31.]
[  1. 54. 94. 63. 91. 27. 55. 74. 77. 77. 81. 29. 91. 20.
 99. 27. 30. 85.  5.  4.]]
2016-05-10 17:23:43.566733
Jobs: 20
Machines: 20
Timeseed: 2010851491
Optimal sequence: 17, 16, 14, 7, 0, 8, 19, 5, 3, 9, 10, 13, 2, 11, 6, 12, 1, 4, 18, 15

```

Figure 5.7: Integrated Testing (3)

```

[Anaconda3] D:\Dropbox\Project>python RUN3.py
[[ 34. 40. 22.  6. 50. 64. 48. 74. 67. 22. 26. 13. 24. 54.
  22.  9. 28. 48. 83. 85. 79. 78. 40. 15. 95. 75. 73. 41.
  52. 71. 84. 73. 25. 41. 74. 15. 58. 55. 47. 12. 36. 48.
  96. 47. 64.  5. 24. 78. 59.  4.]
 [ 24. 85. 19. 94. 59. 65. 42.  8. 10. 15. 96. 63. 30. 20.
  31. 94. 35. 84. 83. 53. 57. 25. 71. 22. 31. 35. 40. 76.
  14. 48. 42. 11. 92. 68. 55. 88. 52. 36. 62. 42. 51. 68.
  10. 38. 29. 56. 23. 71. 45. 98.]
 [ 20. 61. 69. 28. 69. 35. 17. 59. 16. 54.  6. 18.  2.  4.
  80. 45.  2. 18. 83. 21. 41. 51. 85. 56. 88. 40. 20. 16.
  22. 98. 56. 69. 90.  9. 81. 65. 38. 75. 60. 81. 59. 94.
  3. 19. 15. 29. 94. 48. 39.  3.]
 [ 65. 39. 68. 40. 82. 31. 29. 17. 69. 44. 19. 29. 60. 27.
  93. 35. 17.  1. 99. 96.  1. 62. 84. 42. 60. 42. 66.  1.
  1. 31. 41. 62. 83. 74. 38. 56. 57. 54. 45. 39. 85. 40.
  21. 84. 17. 24. 65. 98. 90. 46.]
 [ 47. 62. 11.  6. 21. 60. 82.  2. 31. 27. 81. 34.  7.  6.
  53. 19. 28. 63. 29. 34. 96. 39. 19. 45. 64. 72. 26. 59.
  51. 74.  3. 11. 96. 50. 23. 43. 36. 52.  9. 44. 32.  2.
  78. 83. 69. 47. 17. 35. 45. 30.]]
2016-05-10 17:26:48.043804
Jobs: 50
Machines: 5
Timeseed: 587288402
Optimal sequence: 39, 11, 12, 9, 23, 43, 47, 32, 7, 49, 48, 44, 3, 26, 31, 0, 28, 25, 33, 42, 15, 38, 46, 10, 40, 8, 20, 41, 22, 17, 34, 1, 24, 5, 18, 37, 3
6, 45, 6, 35, 21, 16, 27, 14, 29, 19, 13, 2, 4, 30
Optimal makespan: 2607.0
2016-05-10 17:27:02.277063

```

Figure 5.8: Integrated Testing (4)

```

Anacrona Prompt
26. 52. 65. 10. 27. 25. 6. 96. 16. 57. 98. 38. 66. 91.
70. 83. 49. 7. 54. 25. 6. 87.]
[ 19. 6. 72. 73. 43. 3. 47. 34. 28. 32. 7. 29. 89. 84.
33. 95. 64. 9. 44. 23. 45. 68. 88. 96. 56. 43. 11. 95.
94. 99. 56. 3. 2. 23. 75. 17. 46. 83. 98. 30. 23. 77.
12. 12. 15. 70. 78. 13. 22. 90.]
[ 98. 38. 80. 84. 28. 15. 46. 67. 78. 39. 85. 93. 22. 22.
36. 67. 7. 83. 78. 68. 75. 7. 65. 23. 32. 13. 38. 51.
79. 50. 25. 83. 6. 16. 36. 16. 43. 66. 79. 40. 8. 71.
77. 31. 83. 31. 76. 9. 45. 76.]
[ 20. 79. 35. 67. 65. 20. 30. 23. 85. 55. 32. 45. 94. 84.
48. 56. 46. 90. 73. 2. 8. 62. 49. 86. 97. 34. 2. 19.
66. 23. 1. 26. 47. 10. 75. 39. 18. 57. 95. 70. 11. 42.
9. 61. 81. 75. 94. 31. 48. 88.]
[ 58. 44. 53. 11. 92. 50. 37. 57. 14. 64. 63. 16. 9. 97.
86. 87. 77. 20. 60. 83. 72. 29. 3. 27. 33. 3. 41. 61.
63. 52. 69. 97. 45. 4. 72. 19. 66. 35. 43. 72. 28. 46.
73. 12. 74. 41. 59. 42. 47. 23.]
[ 59. 66. 35. 61. 70. 17. 98. 40. 46. 88. 93. 11. 33. 65.
15. 67. 41. 1. 90. 8. 16. 12. 86. 71. 53. 13. 32. 64.
53. 51. 58. 95. 6. 93. 47. 22. 83. 4. 17. 17. 86. 16.
59. 43. 1. 18. 87. 27. 94. 57.]
[ 59. 60. 72. 54. 97. 71. 43. 92. 71. 53. 28. 97. 55. 91.
33. 12. 12. 36. 73. 78. 7. 3. 4. 87. 84. 16. 12. 84.
2. 34. 9. 98. 83. 98. 70. 3. 76. 16. 49. 27. 20. 24.
61. 48. 49. 34. 97. 92. 62. 81.]
[ 76. 72. 23. 74. 99. 74. 71. 27. 81. 75. 47. 90. 17. 59.
91. 27. 69. 29. 79. 95. 20. 88. 54. 46. 46. 82. 50. 10.
53. 69. 25. 2. 29. 51. 12. 65. 85. 48. 73. 1. 4. 46.
80. 72. 96. 47. 22. 9. 90. 58.]
[ 77. 34. 66. 3. 52. 3. 51. 89. 78. 62. 40. 8. 32. 37.
14. 80. 10. 43. 87. 27. 85. 36. 11. 27. 9. 16. 57. 87.
58. 25. 52. 92. 67. 1. 69. 37. 93. 20. 21. 53. 37. 98.
68. 74. 7. 64. 24. 85. 31. 51.]]
2016-05-10 17:41:41.609817
Jobs: 50
Machines: 10
Timeseed: 49612559
Optimal sequence: 31, 25, 9, 48, 8, 20, 16, 11, 27, 14, 1, 7, 6, 29, 0, 4, 32, 23, 18, 49, 22, 46, 21, 26, 2, 24, 36, 19, 41, 13, 38, 33, 40, 15, 5, 12, 30,
17, 10, 43, 44, 34, 35, 39, 45, 42, 28, 3, 47, 37
Optimal makespan: 3240.0

```

Figure 5.9: Integrated Testing (5)

```

72. 22. 97. 89. 96. 73. 40. 52. 63. 29. 7. 66. 11. 47.
11. 68. 77. 29. 8. 22. 13. 18.]
[ 87. 61. 91. 63. 65. 95. 70. 60. 34. 4. 39. 44. 87. 50.
71. 50. 82. 80. 75. 30. 74. 43. 48. 10. 81. 53. 56. 97.
76. 18. 21. 33. 69. 19. 50. 99. 63. 37. 85. 5. 59. 18.
61. 68. 82. 70. 24. 35. 90. 94.]
[ 78. 45. 82. 71. 80. 67. 35. 72. 92. 96. 31. 24. 41. 35.
79. 63. 74. 58. 61. 50. 44. 68. 98. 87. 39. 61. 96. 87.
93. 87. 95. 22. 89. 4. 18. 77. 25. 51. 2. 49. 20. 41.
4. 50. 50. 67. 95. 41. 51. 31.]
[ 30. 86. 47. 86. 3. 38. 82. 91. 83. 59. 98. 17. 54. 37.
33. 34. 29. 5. 88. 89. 28. 23. 21. 95. 62. 9. 92. 98.
98. 60. 14. 42. 4. 78. 29. 47. 13. 50. 16. 4. 80. 12.
31. 69. 57. 77. 61. 92. 84. 92.]
[ 63. 18. 23. 55. 25. 47. 91. 8. 14. 7. 37. 74. 56. 88.
9. 14. 64. 40. 22. 10. 17. 16. 1. 52. 11. 10. 17. 49.
60. 81. 79. 71. 22. 44. 27. 3. 34. 20. 30. 5. 42. 63.
20. 43. 1. 81. 83. 99. 58. 69.]
[ 69. 45. 63. 62. 63. 19. 44. 2. 14. 8. 61. 79. 72. 90.
2. 7. 82. 9. 12. 94. 68. 99. 27. 85. 37. 3. 5. 9.
67. 17. 81. 33. 85. 89. 19. 93. 54. 41. 16. 76. 21. 11.
78. 45. 85. 82. 32. 48. 76. 33.]
[ 62. 71. 98. 10. 28. 46. 8. 9. 43. 74. 93. 62. 99. 82.
67. 45. 22. 21. 97. 73. 88. 51. 89. 37. 58. 7. 41. 2.
69. 44. 19. 57. 56. 75. 61. 34. 89. 95. 71. 57. 91. 61.
24. 25. 10. 8. 81. 38. 99. 11.]
[ 30. 63. 24. 57. 8. 4. 17. 97. 38. 66. 35. 86. 7. 85.
10. 3. 59. 74. 52. 78. 40. 84. 39. 4. 44. 62. 1. 67.
12. 7. 61. 95. 15. 96. 67. 23. 2. 74. 70. 13. 90. 17.
11. 22. 64. 27. 13. 76. 79. 81.]
[ 37. 2. 36. 60. 82. 50. 46. 87. 27. 24. 69. 91. 10. 80.
74. 14. 51. 40. 95. 51. 54. 66. 62. 25. 15. 27. 89. 90.
2. 4. 89. 11. 34. 64. 11. 36. 73. 70. 85. 82. 60. 19.
30. 5. 20. 45. 77. 31. 70. 46.]]
2016-05-10 17:43:35.130456
Jobs: 50
Machines: 20
Timeseed: 1805594913
Optimal sequence: 3, 11, 9, 12, 44, 30, 10, 19, 29, 13, 16, 35, 40, 18, 46, 49, 0, 28, 39, 45, 14, 33, 4, 25, 21, 38, 1, 6, 48, 37, 32, 36, 7, 27, 22, 20, 4
7, 31, 2, 23, 26, 8, 34, 43, 5, 17, 24, 41, 42, 15
Optimal makespan: 3888.0

```

Figure 5.10: Integrated Testing (6)

```

91. 81.]
[ 65. 67. 55. 35. 19. 18. 35. 39. 89. 57. 79. 5. 33. 46.
47. 46. 99. 39. 7. 15. 56. 32. 62. 50. 99. 89. 30. 24.
23. 44. 83. 13. 83. 73. 34. 89. 18. 32. 90. 73. 28. 18.
81. 31. 2. 3. 78. 99. 71. 8. 88. 76. 33. 66. 3. 31.
8. 63. 53. 89. 63. 49. 60. 53. 70. 9. 16. 79. 93. 69.
4. 95. 87. 49. 79. 14. 4. 20. 10. 28. 4. 91. 63. 66.
25. 12. 36. 94. 19. 49. 19. 18. 18. 53. 49. 29. 42. 59.
48. 3.]
[ 51. 20. 55. 55. 15. 46. 15. 60. 40. 67. 46. 29. 14. 35.
81. 8. 55. 55. 1. 62. 97. 93. 58. 46. 92. 62. 58. 1.
18. 14. 74. 29. 85. 95. 11. 30. 73. 24. 83. 15. 77. 63.
43. 40. 98. 17. 89. 18. 54. 33. 66. 97. 61. 51. 2. 51.
72. 33. 46. 83. 17. 16. 93. 63. 39. 55. 28. 78. 68. 76.
48. 64. 87. 26. 14. 48. 78. 82. 33. 22. 45. 49. 90. 82.
73. 70. 95. 99. 64. 97. 73. 44. 68. 88. 21. 58. 25. 1.
27. 15.]
[ 32. 11. 34. 47. 33. 7. 18. 69. 95. 95. 71. 27. 24. 32.
1. 42. 82. 95. 90. 28. 36. 6. 83. 17. 30. 96. 32. 3.
64. 67. 7. 92. 50. 90. 40. 96. 96. 18. 92. 29. 85. 39.
75. 61. 10. 99. 21. 85. 54. 33. 52. 96. 64. 22. 2. 85.
49. 82. 91. 13. 62. 56. 63. 53. 88. 95. 71. 75. 21. 18.
59. 17. 11. 80. 51. 31. 77. 84. 20. 16. 19. 16. 73. 12.
97. 92. 65. 52. 22. 62. 26. 41. 56. 32. 54. 60. 85. 66.
10. 37.]
[ 15. 53. 87. 5. 21. 32. 53. 27. 14. 29. 30. 76. 58. 28.
29. 43. 25. 35. 83. 22. 3. 10. 73. 47. 87. 50. 74. 7.
89. 89. 72. 9. 34. 7. 32. 82. 77. 64. 13. 55. 81. 55.
37. 43. 99. 39. 25. 17. 13. 65. 60. 70. 22. 62. 37. 95.
87. 18. 31. 75. 23. 48. 18. 5. 63. 1. 25. 6. 29. 27.
10. 29. 53. 87. 39. 46. 18. 44. 24. 86. 65. 45. 52. 95.
22. 97. 71. 31. 46. 54. 59. 90. 72. 65. 84. 20. 17. 51.
57. 72.]]
2016-05-10 17:46:47.146279
Jobs: 100
Machines: 5
Timeseed: 1328833962
Optimal sequence: 88, 34, 18, 84, 90, 86, 65, 17, 9, 79, 55, 45, 44, 21, 11, 82, 93, 10, 16, 64, 39, 53, 38, 4, 20, 61, 91, 41, 49, 58, 94, 74, 35, 28, 2, 9
6, 36, 59, 50, 24, 63, 33, 26, 77, 43, 37, 85, 31, 14, 30, 15, 72, 47, 76, 80, 52, 51, 97, 83, 22, 98, 67, 78, 89, 40, 12, 46, 25, 57, 71, 6, 60, 19, 29, 92,
42, 56, 95, 0, 32, 27, 48, 8, 3, 73, 7, 54, 68, 62, 13, 81, 87, 75, 66, 5, 1, 70, 69, 23, 99
Optimal makespan: 5341.0

```

Figure 5.11: Integrated Testing (7)

## 5.4. Modifications and Improvements

The selection of the order and type of crossover [5] and mutations [6] have been made based on the trial and error basis. We have seen that the current order gives the best results. The current program has been designed only to handle static job allocation and in such a way that each job spends a constant amount of time in each machine. However in practical scenarios the situation may be different and the program would need further sophistication.

Our code after modification is given below.

```

1  # Use this for GUI version
2
3  import numpy as np
4  import openpyxl as ox
5
6  # Read jobs, machines, timeseed columns from Taillard.xlsx and store them in three lists
7  timeseed_list = [] # Initialize lists
8  jobs_list = []
9  machines_list = []
10 makespans = []
11 taillard_file = ox.load_workbook('Taillard.xlsx') # Load workbook
12 taillard = taillard_file.get_sheet_by_name('Sheet1') # Load sheet
13
14 # Values are in rows 3 to 122 of Taillard.xlsx
15 for i in range(3, 123):
16     j = taillard.cell(row = i, column = 1).value # Read Jobs value
17     m = taillard.cell(row = i, column = 2).value # Read Machines value
18     t = taillard.cell(row = i, column = 3).value # Read Timeseed value
19     jobs_list.append(j)
20     machines_list.append(m)
21     timeseed_list.append(t)
22
23
24 def rng(seed, low, high):
25     # Random number generator, as in Taillard's benchmarks paper
26     m = 2147483647 # 2^31 - 1
27     a = 16807
28     b = 127773
29     c = 2836
30     k = int(seed / b)
31     # Update seed
32     seed = a * (seed % b) - k * c
33     if (seed < 0):
34         seed = seed + m
35     # Random number between 0 and 1
36     val = float(seed) / m
37     # Between low and high
38     randnum = low + int(val * (high - low + 1))
39     return randnum, seed
40
41
42 def random_matrix(nc, jb, seed_value):

```

Figure 5.12: Modified Code Snippet

```

42 def random_matrix(mc, jb, seed_value):
43     # Generate random mc x jb matrix with rng
44     a = np.ndarray((mc, jb)) # Initialize matrix
45
46     # Generate random matrix of times for jobs x machines matrix
47     for i in range(mc):
48         for j in range(jb):
49             a[i,j], seed_value = rng(seed_value, 1, 99)
50
51     return a
52
53
54 def calculate_makespan(a, seq):
55     # Order the jobs (rows) in order of the sequence
56     a = a[seq]
57
58     b = np.zeros(a.shape)
59     jobs = a.shape[0]
60     mcs = a.shape[1]
61
62     b[0,0] = a[0,0]
63     # Build first row
64     for i in range(1, mcs):
65         b[0,i] = a[0,i] + b[0,i - 1]
66     # Build first column
67     for i in range(1, jobs):
68         b[i,0] = a[i,0] + b[i - 1,0]
69     # Build the rest
70     for i in range(1, jobs):
71         for j in range(1, mcs):
72             b[i,j] = a[i,j] + (b[i - 1,j] if b[i - 1,j] > b[i,j - 1] else b[i, j - 1])
73
74     return int(b[-1, -1])
75
76 def roulette_wheel(sequence, makespan):
77     # Store inverses of makespan values in a list
78     inverse = []
79     for i in makespan:
80         j = 1 / i
81         inverse.append(j)
82
83     total_sum = 0

```

Figure 5.13: Modified Code Snippet(2)

```

84     # Calculating sum of all the inverted values
85     for i in inverse:
86         total_sum = total_sum + i
87
88     # Generate arrays of newsize = 3 * previous size, according to RWS
89     newsize = 3 * len(makespan)
90     seq = np.ndarray((newsize, sequence.shape[1]))
91     mks = np.empty(newsize)
92
93     # Generate 'newsize' number of sequences
94     import random
95     for r in range(newsize):
96         # Generating random value between 0 and total_sum
97         x = random.uniform(0, total_sum)
98         partial_sum = 0
99
100         for i in range(len(inverse)):
101             partial_sum = partial_sum + inverse[i]
102             if partial_sum >= x:
103                 mks[r] = makespan[i]
104                 seq[r] = sequence[i]
105                 break
106
107     return seq, mks
108
109
110 def sort_one_list_by_another(s, m):
111     # Sort elements in s and m according to m
112     indices = m.argsort()
113     s = s[indices]
114     m = m[indices]
115     return s, m
116
117
118 def sort_and_reduce(s, m):
119     s, m = sort_one_list_by_another(s, m)
120     s = s[:20]
121     m = m[:20]
122     return s, m
123
124 def calculate_optimal_makespan(*args):
125     import genetic

```

Figure 5.14: Modified Code Snippet (3)

```

126 import sys
127 # To make command line print complete nd arrays
128 np.set_printoptions(threshold=sys.maxsize)
129
130 # Will run for entire 120 problem range
131 problem = problem_num.get()
132
133 jobs = jobs_list[problem]
134 machines = machines_list[problem]
135 timeseed = timeseed_list[problem]
136
137 # Write these to GUI
138 jobs_val.set(jobs)
139 macs_val.set(machines)
140 time_val.set(timeseed)
141
142 # Generate matrix of times
143 a = random_matrix(machines, jobs, timeseed)
144 print("Problem No.:", problem, "\nMatrix:\n", a)
145 # Transpose matrix to calculate makespan
146 at = a.transpose()
147
148 # Start with all possible permutations of first 4 jobs
149 import itertools
150 init_jobs = 4
151 init_job_list = list(range(init_jobs))
152 # Generate all 24 permutations
153 sequence_list = np.array(list(itertools.permutations(init_job_list)))
154
155
156 # This loop will eventually run till init_jobs <= jobs
157 while init_jobs <= jobs:
158     makespan_list = np.array([])
159     for seq in sequence_list:
160         seq = list(seq)
161         makespan_list = np.append(makespan_list, calculate_makespan(at[init_job_list], seq))
162
163 # Sort both arrays in ascending order of makespan and reduce to 20 best sequences
164 sequence_list, makespan_list = sort_and_reduce(sequence_list, makespan_list)
165
166 # Roulette wheel .. 3 x 20 output
167 sequence_list, makespan_list = roulette_wheel(sequence_list, makespan_list)

```

Figure 5.15: Modified Code Snippet (4)

```

168
169
170 # Again, sort and reduce to 20
171 sequence_list, makespan_list = sort_and_reduce(sequence_list, makespan_list)
172
173 # Now ordered crossover each of the 20 sequences with each other and add to lists
174 for i in range(20):
175     for j in range(20):
176         if i != j:
177             child = genetic.ordered_crossover(sequence_list[i], sequence_list[j])
178             child = np.array(child)
179             sequence_list = np.vstack((sequence_list, child))
180             makespan_list = np.append(makespan_list, calculate_makespan(at[init_job_list], list(child)))
181
182 # Sort both arrays in ascending order of makespan and reduce to 20 best sequences
183 sequence_list, makespan_list = sort_and_reduce(sequence_list, makespan_list)
184
185
186 # Applying mutation. Inverse Mutation.
187 for i in sequence_list:
188     mutated = genetic.inverse_mutation(i)
189     mutated = np.array(mutated)
190     sequence_list = np.vstack((sequence_list, mutated))
191     makespan_list = np.append(makespan_list, calculate_makespan(at[init_job_list], list(mutated)))
192
193 # Sort both arrays in ascending order of makespan and reduce to 20 best sequences
194 sequence_list, makespan_list = sort_and_reduce(sequence_list, makespan_list)
195
196
197 # Applying another mutation. Pairwise Swap Mutation.
198 for i in sequence_list:
199     mutated = genetic.pairwise_swap_mutation(i)
200     mutated = np.array(mutated)
201     sequence_list = np.vstack((sequence_list, mutated))
202     makespan_list = np.append(makespan_list, calculate_makespan(at[init_job_list], list(mutated)))
203
204 # Sort both arrays in ascending order of makespan and reduce to 20 best sequences
205 sequence_list, makespan_list = sort_and_reduce(sequence_list, makespan_list)
206
207
208 # Ordered Crossover again
209 for i in range(20):

```



Figure 5.16: Modified Code Snippet (5)

```

210         for j in range(20):
211             if i != j:
212                 child = genetic.ordered_crossover(sequence_list[i], sequence_list[j])
213                 child = np.array(child)
214                 sequence_list = np.vstack((sequence_list, child))
215                 makespan_list = np.append(makespan_list, calculate_makespan(at[init_job_list], list(child)))
216
217     # Sort both arrays in ascending order of makespan and reduce to 20 best sequences
218     sequence_list, makespan_list = sort_and_reduce(sequence_list, makespan_list)
219
220     # Set first element of sorted list as best makespan.
221     best_sequence = sequence_list[0]
222     best_makespan = makespan_list[0]
223
224     # Bring in next job into every position in sequence_list
225     init_jobs += 1
226     init_job_list = list(range(init_jobs))
227     new_sequence_list = np.array([]).reshape(0, init_jobs)
228     for s in sequence_list:
229         for pos in range(s.size + 1):
230             new_sequence_list = np.vstack((new_sequence_list, np.insert(s, pos, init_jobs - 1)))
231     sequence_list = new_sequence_list
232
233     print("\nBest Sequence:\n", best_sequence, "\n\n")
234     makespan_val.set(best_makespan)
235
236
237 def increment(*args):
238     problem_num.set(problem_num.get() + 1)
239     calculate_optimal_makespan()
240
241 def decrement(*args):
242     problem_num.set(problem_num.get() - 1)
243     calculate_optimal_makespan()
244
245 from tkinter import *
246 # ttk has newer themed versions of the tkinter stuff. Hence importing it separately.
247 from tkinter import ttk
248 root = Tk()
249 # Format window size
250 root.geometry('{}x{}'.format(700, 500))
251 # Give the window a name

```

Figure 5.17: Modified Code Snippet (6)

```

252 root.title("Scheduling Optimization")
253
254 # Creating frame inside window for ttk stuff
255 mainframe = ttk.Frame(root, padding="3 3 12 12")
256 mainframe.grid(column=0, row=0, sticky=(N, W, E, S))
257 mainframe.columnconfigure(0, weight=1)
258 mainframe.rowconfigure(0, weight=1)
259
260 problem_num = IntVar()
261 makespan_val = IntVar()
262 jobs_val = IntVar()
263 mcs_val = IntVar()
264 time_val = IntVar()
265
266 problem_entry = ttk.Entry(mainframe, width=7, textvariable=problem_num)
267 problem_entry.grid(column=2, row=1, sticky=(W, E))
268 ttk.Label(mainframe, textvariable=makespan_val).grid(column=2, row=6, sticky=(W, E))
269 ttk.Button(mainframe, text="Calculate", command=calculate_optimal_makespan).grid(column=6, row=1, sticky=W)
270 ttk.Label(mainframe, text="Problem No.").grid(column=1, row=1, sticky=W)
271 ttk.Label(mainframe, text="Optimal makespan").grid(column=1, row=6, sticky=E)
272
273 ttk.Label(mainframe, text="Jobs:").grid(column=1, row=3, sticky=(W, E))
274 ttk.Label(mainframe, textvariable=jobs_val).grid(column=2, row=3, sticky=(W, E))
275 ttk.Label(mainframe, text="Machines").grid(column=1, row=4, sticky=(W, E))
276 ttk.Label(mainframe, textvariable=mcs_val).grid(column=2, row=4, sticky=(W, E))
277 ttk.Label(mainframe, text="Timeseed").grid(column=1, row=5, sticky=(W, E))
278 ttk.Label(mainframe, textvariable=time_val).grid(column=2, row=5, sticky=(W, E))
279
280 calculating = StringVar()
281 calculating.set("")
282 ttk.Label(mainframe, textvariable=calculating).grid(column=3, row=5, sticky=(W, E))
283
284 ttk.Button(mainframe, text="Next", command=increment).grid(column=3, row=7, sticky=E)
285 ttk.Button(mainframe, text="Previous", command=decrement).grid(column=1, row=7, sticky=W)
286
287 # Padding around every widget in the frame
288 for child in mainframe.winfo_children():
289     child.grid_configure(padx=5, pady=5)
290 # Focus on the entry field at first
291 problem_entry.focus()
292 # Executes function when you press Enter
293 root.bind('<Return>', calculate_optimal_makespan)

```

Figure 5.18: Modified Code Snippet (7)

```

294
295 # Changing theme
296 s = ttk.Style()
297 s.theme_use('clam')
298
299 # Start the infinite loop
300 root.mainloop()

```

Figure 5.19: Modified Code Snippet (8)

## 6. RESULTS AND DISCUSSIONS

### 6.1. Results

The Taillard benchmarks [15] consist of 120 problems, each definable by the number of jobs, number of machines, and the seed value for the random number generator [16]. These problems range from small problems, with 20 jobs and 5 machines, to large problems, with 500 jobs and 20 machines. The benchmarks provide theoretical lower bounds of the problems, along with the best obtained makespan for each.

Our principal means of measuring the efficacy of the proposed heuristic is by comparing our obtained results against the benchmarks defined by Since the heuristic is non-deterministic, we ran our program multiple times to improve the odds of getting better results, until a few successive executions showed no further improvement.

Finally, as a quantitative measure of the amount of improvement brought about by this method, we have calculated the Average Relative Percentage Difference (ARPD). Having done so, we observe that the new method performs significantly better for larger problems than for small ones. For the set of problems with the largest number of jobs, that is, Taillard's instances 111 to 120, the overall ARPD turns out to be negative, which indicates that the technique results in better makespans on average than the benchmark values for the largest problem size. The values used for comparison are the original benchmarks values from, and not updated values obtained by other research. After comparison, it turns out that the proposed heuristic technique is suitable for use in large PFSSPs, such as those with 500 jobs or maybe even more. The results table is shown below.

Job	M/C	Time seed	UpperBound	LowerBound	Makespan	RPD
20	5	873654221	1278	1232	1290	0.938967136
20	5	379008056	1359	1290	1383	1.766004415
20	5	1866992158	1081	1073	1098	1.572617946
20	5	216771124	1293	1268	1309	1.237432328
20	5	495070989	1236	1198	1250	1.132686084
20	5	402959317	1195	1180	1210	1.255230126
20	5	1369363414	1239	1226	1251	0.968523002
20	5	2021925980	1206	1170	1206	0
20	5	573109518	1230	1206	1230	0
20	5	88325120	1108	1082	1127	1.714801444
20	10	587595453	1582	1448	1620	2.402022756
20	10	1401007982	1659	1479	1700	2.471368294
20	10	873136276	1496	1407	1526	2.005347594
20	10	268827376	1378	1308	1383	0.362844702
20	10	1634173168	1419	1325	1448	2.043692741
20	10	691823909	1397	1290	1438	2.934860415

20	10	73807235	1484	1388	1520	2.425876011
20	10	1273398721	1538	1363	1583	2.925877763
20	10	2065119309	1593	1472	1627	2.134337728
20	10	1672900551	1591	1356	1639	3.016970459
20	20	479340445	2297	1911	2331	1.480191554
20	20	268827376	2100	1711	2152	2.476190476
20	20	1958948863	2326	1844	2376	2.14961307
20	20	918272953	2223	1810	2253	1.349527665
20	20	555010963	2291	1899	2323	1.396769969
20	20	2010851491	2226	1875	2262	1.617250674
20	20	1519833303	2273	1875	2306	1.451825781
20	20	1748670931	2200	1880	2251	2.318181818
20	20	1923497586	2237	1840	2328	4.067948145
20	20	1829909967	2178	1900	2225	2.157943067
50	5	1328042058	2724	2712	2724	0
50	5	200382020	2836	2808	2863	0.952045134
50	5	496319842	2621	2596	2636	0.572300649
50	5	1203030903	2751	2740	2774	0.836059615

50	5	1730708564	2863	2837	2864	0.034928397
50	5	450926852	2829	2793	2831	0.070696359
50	5	1303135678	2725	2689	2736	0.403669725
50	5	1273398721	2683	2667	2688	0.186358554
50	5	587288402	2554	2527	2561	0.274079875
50	5	248421594	2782	2776	2782	0
50	10	1958948863	3037	2907	3095	1.909779388
50	10	575633267	2911	2821	2976	2.232909653
50	10	655816003	2873	2801	2963	3.132613992
50	10	1977864101	3067	2968	3113	1.499836974
50	10	93805469	3025	2908	3060	1.157024793
50	10	1803345551	3021	2941	3088	2.217808673
50	10	49612559	3124	3062	3179	1.76056338
50	10	1899802599	3048	2959	3096	1.57480315
50	10	2013025619	2913	2795	2977	2.197047717
50	10	578962478	3114	3046	3172	1.862556198
50	20	1539989115	3886	3480	4030	3.705609882
50	20	691823909	3733	3424	3860	3.402089472

50	20	655816003	3689	3351	3808	3.225806452
50	20	1315102446	3755	3336	3890	3.595206391
50	20	1949668355	3655	3313	3767	3.064295486
50	20	1923497586	3719	3460	3833	3.065340145
50	20	1805594913	3730	3427	3890	4.289544236
50	20	1861070898	3744	3383	3864	3.205128205
50	20	715643788	3790	3457	3901	2.928759894
50	20	464843328	3791	3438	3898	2.822474281
100	5	896678084	5493	5437	5495	0.036409976
100	5	1179439976	5274	5208	5284	0.189609405
100	5	1122278347	5175	5130	5211	0.695652174
100	5	416756875	5018	4963	5021	0.059784775
100	5	267829958	5250	5195	5255	0.095238095
100	5	1835213917	5135	5063	5139	0.077896787
100	5	1328833962	5247	5198	5304	1.086335049
100	5	1418570761	5106	5038	5130	0.470035253
100	5	161033112	5454	5385	5467	0.238357169
100	5	304212574	5328	5272	5329	0.018768769

100	10	1539989115	5776	5759	5844	1.177285319
100	10	655816003	5362	5345	5428	1.230883999
100	10	960914243	5679	5623	5755	1.338263779
100	10	1915696806	5820	5732	5905	1.4604811
100	10	2013025619	5491	5431	5564	1.329448188
100	10	1168140026	5308	5246	5363	1.036171816
100	10	1923497586	5602	5523	5687	1.517315245
100	10	167698528	5640	5556	5696	0.992907801
100	10	1528387973	5891	5779	5955	1.086402988
100	10	993794175	5860	5830	5903	0.733788396
100	20	450926852	6345	5851	6542	3.104806935
100	20	1462772409	6323	6099	6425	1.613158311
100	20	1021685265	6385	6099	6523	2.161315583
100	20	83696007	6331	6072	6496	2.606223345
100	20	508154254	6405	6009	6585	2.81030445
100	20	1861070898	6487	6144	6650	2.512717743
100	20	26482542	6393	5991	6556	2.549663695
100	20	444956424	6514	6084	6746	3.561559718



100	20	2115448041	6386	5979	6568	2.849984341
100	20	118254244	6544	6298	6621	1.176650367
200	10	471503978	10927	10816	10911	-0.146426284
200	10	1215892992	10570	10422	10674	0.983916746
200	10	135346136	11004	10886	11076	0.654307525
200	10	1602504050	10936	10794	11057	1.106437454
200	10	160037322	10550	10437	10654	0.985781991
200	10	551454346	10378	10255	10467	0.857583349
200	10	519485142	10885	10761	10977	0.845199816
200	10	383947510	10808	10663	10863	0.508882309
200	10	1968171878	10473	10348	10551	0.744772272
200	10	540872513	10727	10616	10792	0.605947609
200	20	2013025619	11441	10979	11569	1.118783323
200	20	475051709	11549	10947	11695	1.264178717
200	20	914834335	11537	11150	11792	2.210279969
200	20	810642687	11580	11127	11676	0.829015544
200	20	1019331795	11484	11132	11654	1.480320446
200	20	2056065863	11416	11085	11609	1.690609671

200	20	1342855162	11659	11194	11760	0.866283558
200	20	1325809384	11587	11126	11730	1.234141711
200	20	1988803007	11498	10965	11570	0.62619586
200	20	765656702	11569	11122	11757	1.625032414
500	20	1368624604	26699	25922	26692	-0.02621821
500	20	450181436	27303	26353	27224	-0.289345493
500	20	1927888393	26928	26320	27007	0.293374926
500	20	1759567256	27009	26424	26918	-0.336924729
500	20	606425239	26771	26181	26827	0.209181577
500	20	19268348	26959	26401	26974	0.055640046
500	20	1298201670	26870	26300	26900	0.111648679
500	20	2041736264	27104	26429	27055	-0.180785124
500	20	379756761	26586	25891	26618	0.120364101
500	20	28837162	26910	26315	26917	0.026012635

Table 6.1: Results

## 6.2. User Documentation

The user documentation has been arranged into seven chapters.

### Chapter 1: Introduction

A rudimentary description of the context of the project and its relation to work already done in the area has been summarized. A brief statement of the aims and objectives of the project has been proposed.

The purpose of this chapter deals with the description of the topic of our project that answers questions on why we are doing this project and how theoretically significant improvements have been made by our design. Scope includes a brief overview of the methodology, assumptions and limitations and the main issues being dealt with by us. The direct and indirect applications of our work has been described. The subsection named achievements explain what knowledge we have achieved after the completion of our work and what contributions our project has made. The subsection named “Goals achieved” describes the degree to which the findings support the original objectives laid out by the project. The goals may be partially or fully achieved, or exceeded.

### Chapter 2: Survey of Technologies

In the chapter Survey of Technologies, we have demonstrated our awareness and understanding of Available Technologies related to the topic of our project. We have given the detail of all the related technologies that are necessary to complete our project. We have described the technologies available in our chosen area and present a comparative study of all those Available Technologies.

### Chapter 3: Requirements and Analysis

**Problem Definition:** The problem has been defined providing the details of the overall problem and then the problem has been divided into sub-problems.

**Requirements Specification:** In this phase we have defined the requirements of the system, independent of how these requirements will be accomplished. The Requirements Specification describes the things in the system and the actions that can be done on these things by identifying the operation and problems of the existing system.

**Planning and Scheduling:** Planning and scheduling is a complicated part of software development. Planning, for our purposes, can be thought of as determining all the small tasks that must be carried out in order to accomplish the goal. Planning also takes into account rules, known as constraints, which,

control when certain tasks can or cannot happen. Scheduling can be thought of as determining whether adequate resources are available to carry out the plan.

**Software and Hardware Requirements:** The details of all the software and hardware needed for the development and implementation of our project has been defined.

**Preliminary Product Description:** The requirements and objectives of the new system has been defined. The functions and operation of the application/system we have developed as our project has been explored.

**Conceptual Models:** The problem domain which describes operations that can be performed on the system, and the allowable sequences of those operations has been described.

#### Chapter 4: System Design

Describes desired features and operations in detail, including screen layouts, business rules, process diagrams, pseudocode and other documentation.

**Basic Modules:** We have followed the divide and conquer theory, and divided the overall problem into more manageable parts and developed each part or module separately. When all modules are ready, we have integrated all the modules into one system.

**Data Design:** Data design consists of how we organise, manage and manipulate the data.

- **Schema Design:** The structure and explanation of schemas used in our project has been defined.
- **Data Integrity and Constraints:** The validity checks and constraints have been provided to keep an integrity check.

**Procedural Design:** Procedural design is a systematic way for developing algorithms or procedurals and consists of the following modules.

- **Logic Diagrams:** Define the systematical flow of procedure that improves its comprehension and helps the programmer during implementation. e.g., Control Flow Chart, Process Diagrams etc.
- **Data Structures:** Create and define the data structure used in your procedures.
- **Algorithms Design:** With proper explanations of input data, output data, logic of processes, design and explain the working of algorithms.

User Interface Design: User, task, environment analysis have been identified and how we intend to map those requirements in order to develop a “User Interface. Some rough pictorial views of the user interface and its components has been shown.

Test Cases Design: Test cases which provide easy detection of errors and mistakes with in a minimum period of time and with the least effort has been shown.

## Chapter 5: Implementation and Testing

Implementation Approaches: This chapter defines the plan of implementation, and the standards being used in the implementation.

Testing Approach consists of the submodules unit testing and integrated testing.

- Unit Testing: Unit testing deals with testing a unit or module as a whole. This tests the interaction of many functions.
- Integrated Testing: Brings all the modules together into a special testing environment, then checks for errors, bugs and interoperability. It deals with tests for the entire application. Application limits and features are tested here.

Modifications and Improvements deals with the modifications introduced to increase the code efficiency.

## Chapter 6: Results and Discussion

Test Reports: Explain the test results and reports based on our test cases, which shows that our software is capable of facing any problematic situation and that it works fine in different conditions. Different sample inputs and outputs have been shown.

## Chapter 7: Conclusions

Conclusion: The conclusion has been summarised in a fairly short chapter explaining the limitations we have encountered during the testing of our software that we were not able to modify.

Future Scope of the Project describes two things: firstly, new areas of investigation prompted by developments in this project, and secondly, parts of the current work that were not completed due to time constraints and/or problems encountered.

## 7. CONCLUSIONS

In this project, we have developed a method to minimize the makespan of a permutation flow shop scheduling problem using natural algorithms. After testing the method on the set of problems defined in the Taillard benchmarks [15], we have seen that the technique performs well enough to match the benchmarks, and in some cases, even better them. The heuristic performs particularly well on larger problem sizes, and has given us a negative average relative percentage difference compared to the benchmarks for the largest problems, which consist of 500 jobs. Thus, the proposed method may be used to optimize large-scale flow shop scheduling problems, where exact algorithms and exhaustive search methods are impossible or infeasible.

The most natural application of this optimization process is to improve the scheduling of jobs in computer systems. Different algorithms such as the Ant Colony Optimization [17], Simulated Annealing [18] etc. can be used to generate the output which can then be compared with the outputs produced by our algorithm. Now we have used a static system i.e. the set of jobs present in the system does not change. We can improve on this by adding some flexibility on job arrival so that the natural algorithms have a larger search space to find the optimum solution. This technique can be used in the industrial Computerized Numerical Control (CNC) machines, so that a single machine can handle various operations.

One major limitation of our project is that it is static in nature. We can improve on this by adding a little dynamism, so that arriving jobs may be considered. This will enable the natural algorithms have a larger search space to find the optimum solution. Another improvement that can be made is that we may consider jobs in batches. Sometimes in certain optimization problems, we may want to find the optimum sequences and makespans for a certain group of jobs.

## REFERENCES

- [1]. K.R. Baker, Introduction to sequencing and scheduling. John Wiley & Sons, 1974. APA
- [2]. D. Laha and A. Chakravorty. "A new heuristic for minimizing total completion time objective in permutation flow shop scheduling." The International Journal of Advanced Manufacturing Technology 53.9-12 (2011): 1189-1197.
- [3]. Muhammad Nawaz, E. Emory Enscore, and Inyong Ham. "A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem." Omega 11.1 (1983): 91-95.
- [4]. [https://en.wikipedia.org/wiki/Genetic\\_operator](https://en.wikipedia.org/wiki/Genetic_operator), 15-05-2016
- [5]. [https://en.wikipedia.org/wiki/Crossover\\_\(genetic\\_algorithm\)](https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm)), 15-05-2016
- [6]. [https://en.wikipedia.org/wiki/Mutation\\_\(genetic\\_algorithm\)](https://en.wikipedia.org/wiki/Mutation_(genetic_algorithm)), 15-05-2016
- [7]. [https://en.wikipedia.org/wiki/Selection\\_\(genetic\\_algorithm\)](https://en.wikipedia.org/wiki/Selection_(genetic_algorithm)), 15-05-2016
- [8]. [https://en.wikipedia.org/wiki/Python\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Python_(programming_language)), 15-05-2016
- [9]. <http://www.numpy.org/>, 15-05-2016
- [10]. <https://openpyxl.readthedocs.io/en/default/>, 15-05-2016
- [11]. <https://docs.python.org/2/library/random.html>, 15-05-2016
- [12]. <https://docs.python.org/2/library/sys.html>, 15-05-2016
- [13]. <https://docs.python.org/2/library/itertools.html>, 15-05-2016
- [14]. <https://wiki.python.org/moin/TkInter>, 15-05-2016
- [15]. E. Taillard, "Benchmarks for basic scheduling problems." European Journal of Operational Research 64.2 (1993): 278-285.
- [16]. P. Bratley, B.L. Fox & L.E. Schrage, "A guide to Simulation", Springer Verlag, New-York, 1983.
- [17]. [https://en.wikipedia.org/wiki/Fitness\\_proportionate\\_selection](https://en.wikipedia.org/wiki/Fitness_proportionate_selection), 15-05-2016
- [18]. [https://en.wikipedia.org/wiki/Ant\\_colony\\_optimization\\_algorithms](https://en.wikipedia.org/wiki/Ant_colony_optimization_algorithms), 15-05-2016
- [19]. [https://en.wikipedia.org/wiki/Simulated\\_annealing](https://en.wikipedia.org/wiki/Simulated_annealing), 15-05-2016



## APPENDIX

### Program Code

#### 1. Genetic.py

Python module containing the functions which perform the genetic operations: Ordered Crossover, Insert Mutation and Pairwise Swap Mutation.

#### 2. Run.py

Python program which executes the algorithm for all the 120 instances of the Taillard benchmarks and writes the outputs to an Excel file, which is used to compute the ARPD.

#### 3. Integrated.py

Python program which launches a GUI which integrates the aforementioned modules and displays number of jobs, number of machines, seed value along with the optimal makespan value for the corresponding individual sub-problems. There are also two buttons to navigate between sub-problems.

The source codes are attached starting from the following page.