A GENETIC ALGORITHM APPROACH FOR SOLVING DYNAMIC JOB SHOP SCHEDULING PROBLEM

Pitigalage Buddhika Chandradeepa Kurera

(138116D)

Degree of Master of Engineering

Department of Mechanical Engineering

University of Moratuwa
Sri Lanka

February 2019

A GENETIC ALGORITHM APPROACH FOR SOLVING DYNAMIC JOB SHOP SCHEDULING PROBLEM

Pitigalage Buddhika Chandradeepa Kurera

(138116D)

Thesis/Dissertation submitted in partial fulfillment of the requirements for the degree of Master of Engineering

Department of Mechanical Engineering

University of Moratuwa
Sri Lanka

February 2019

Declaration

I declare that this is my own work and this thesis does not incorporate without acknowledgement any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief, it does not contain any material previously published or written by another person except where due reference is made in the text.

Also, I hereby grant to University of Moratuwa the non-exclusive right to reproduce and distribute my thesis, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).

Signature : ....
Buddhika Kurera (138116D)

Date : ....

The above candidate has carried out research for the Master's thesis under my supervision.

Signature of the Supervisor : ....
Dr.V.P.C. Dassanayake

Date : ....

Acknowledgements

I would like to thank my project supervisor Dr. V.P.C. Dissanayake, Senior Lecturer Department of Mechanical Engineering, University of Moratuwa, for being the sole guidance throughout the project.

I would like to extend my sincere gratitude to the Python and LaTex communities for their efforts on maintaining these Free and Open Source Software projects which are used by the whole scientific community.

I would also use this opportunity to express my gratitude to everyone who supported me from the beginning to the end of this research project, in numerous ways, especially my family and friends.

Abstract

Job Shop Scheduling Problem (JSSP) is a non-deterministic, polynomial-time (NP) hard combinatorial optimization problem. It is one of the most common problems in manufacturing due to its widespread application and the usability across the manufacturing industry. Due to the vast solution space the JSSP problem deals with, it is impossible to apply brute force search techniques to obtain an optimal solution. Indeed, it is not possible to obtain an optimal solution when the number of jobs and the machines increase. Numerous researches have been carried out studying many approaches to solve this problem. In this research, Genetic Algorithm (GA) which is another widely used nonlinear optimization technique has been used to propose an algorithm. A novel chromosome representation (indirect) with an encoding based on time is introduced in this research. The proposed solution is capable of handling multiple disruptions which are new job arrivals, sudden machine breakdown and unplanned machine maintenance. The proposed algorithm is tested against benchmark problems in Static JSSP and some developed scenarios to simulate Dynamic JSSP conditions. The results show that the proposed algorithm generates near optimal schedules for Static JSSP. This algorithm can be used as a planning tool by the planners. It is possible to simulate almost all the real-life scenarios using this algorithms and schedules can be generated satisfying the required conditions. The algorithm can be developed further by employing a local search algorithm which produced more precious, optimal schedules.

Keywords: Dynamic Job Shop Scheduling Problem, Genetic Algorithm, Disruptions in Job Shops

List of Tables

4.1 Type codes 44

5.1 Parent Selection Method – TS and Next Generation Selection Method – TS for problems of 10 x 10 (Jobs x Machines) 50
5.2 Parent Selection Method – Random and Next Generation Selection Method – TS for problems of 10 x 10 (Jobs x Machines) 50
5.3 Parent Selection Method – RWS and Next Generation Selection Method – RWS for problems of 10 x 10 (Jobs x Machines) 50
5.4 Fully Completed Operations per Job, $LA19^A$ 52
5.5 Partially Completed Operations per Job, $LA19^A$ 52

List of Figures

3.1 Genearted Feasible Solutions 30
3.2 Assigning Operations to Machines 31
3.3 Continuity of STU on the timeline 32
3.4 Crossover and Mutation process 35
3.5 Crossover and Mutation 36

4.1 Format of the Events 43
4.2 Examples for defined Events 43

5.1 Movement of Mutation Rate 48
5.2 Convergence against the number of generations 49
5.3 Generated Schedule for modified LA19 problem ($LA19^A$) 51
5.4 After including the new job in $LA19^A$ 53

List of Abbreviations

AI - Artificial Intelligence
ANN - Artificial Neural Network
BKS - Best-Known Solutions
BLSA - Bottleneck Local Search Agent
CR - Crossover Rate
CSV - Comma-Separated Values
DSPN - Distributed Problem Solving Networks
EDD - Earliest Due Date
FCFS - First Come, First Served
GA - Genetic Algorithm
GBML - Genetic Algorithm-based Machine Learning
JSSP - Job Shop Scheduling Problem
LSOA - Local Search Optimization Algorithm
MR - Mutation Rate
MUSA - Matchup Schedule Algorithm
PBX - Position Based Crossover
PSO - Particle Swarm Optimization
RP - Rescheduling Point
RS - Random Selection
RWS - Roulette Wheel Selection
SE - Scheduling Engine
SI - Swarm Intelligence
SPT - Shortest Processing Time
STU - Starting Time Unit
TS - Tournament Selection
VNS - Variable Neighbor Search

Table of Contents

Declaration ii

Acknowledgements iii

Abstract iv

List of Tables v

List of Figures vi

List of Abbreviation vii

1 Introduction 1

2 Literature Review 3
2.1 Job Shop Scheduling Problem (JSSP) 3
2.1.1 Varieties of JSSP 4
2.2 Methods for Solving Static JSSP 5
2.2.1 Heuristic Approach 5
2.2.2 Neighborhood Search Method 6
2.2.3 Shifting Bottleneck 6
2.2.4 Simulated Annealing 7
2.2.5 Tabu Search 7
2.2.6 Artificial Intelligence 7
2.2.7 Expert Systems 8
2.2.8 Distributed AI – Agents 8
2.2.9 Artificial Neural Network (ANN) 9
2.2.10 Evolutionary Algorithms 10
2.3 Method for Solving Dynamic JSSP 12
2.3.1 Heuristics 15
2.3.2 Metaheuristics 16
2.3.3 Simulated annealing and Tabu search 17
2.3.4 Genetic Algorithms (GA) 17
2.3.5 Artificial Intelligence (AI) 18
2.3.6 Artificial Neural Network (ANN) 18
2.4 Chapter Summary 19

3 Methodology **20**
3.1 Problem Definition 20
3.1.1 Disruption Demanding A Schedule Change 22
3.1.2 Conditions to Satisfy When Scheduling 24
3.1.3 Requirements to Satisfy 24
3.2 Methodology 25
3.2.1 GA related to JSSP 26
3.2.2 How GA works 26
3.3 Design of the Algorithm 28
3.3.1 Chromosome Representation - A novel approach 28
3.3.2 Generating Initial Population 29
3.3.3 Fitness Calculation 33
3.3.4 Crossover 34
3.3.5 Mutation 36
3.3.6 Natural Selection 36
3.4 Optimization 37
3.5 Dynamic JSSP 37
3.6 Disruptions 38
3.6.1 Arrival of new jobs 38
3.6.2 Unplanned machine breakdown 39
3.6.3 Unplanned machine maintenance 40

4 Implementation **41**
4.1 Implementation Approach 41
4.2 Standards in the Implementation 42
4.3 Inputs to the Algorithm 42
4.3.1 Operation Sequence per Job 42
4.3.2 Duration per operations 43
4.3.3 Information on machine unavailability 43
4.3.4 Historical Information on the Current Schedule 44
4.3.5 Information on Constraints 44
4.3.6 Graphically Visualization the Schedule 45

5 Results and Analysis **46**
5.1 Tests for Solving Static JSSP 46
5.1.1 Parent and Next Generation Selection 48
5.2 Dynamic JSSP 51
5.2.1 Scenario A - Arrival of new jobs 51
5.2.2 Scenario B - Machine break-down 53
5.2.3 Scenario C - Unplanned machine maintenance 54
5.2.4 Scenario D - Start time fixed operations 54
5.3 Analysis 54

6 Conclusion **57**
6.1 Conclusion 57
6.2 Future Work 58

A Program Code **65**
A.1 Implementation of the Proposed Algorithm 65
A.2 Tournament Selection Algorithm 75
A.3 Roulette Wheel Selection Algorithm 78

Chapter 1

Introduction

Job Shop Scheduling Problem (JSSP) is a nson-deterministic, polynomial-time (NP) hard combinatorial optimization problem which is one of the widely studied problems in manufacturing engineering, due to the strong relevance to the manufacturing industry. As a result of the various complex requirements in the industry, solving the scheduling problems incorporating all the real-life scenarios become extremely important.

The classical version of the problem, also known as the Static Job Shop Scheduling Problem (JSSP) does not consider requirements faced by the manufacturing industry, as it does not consider disruptions such as new job arrivals, machine breakdown etc. Therefore, Dynamic JSSP which takes such disruptions into consideration become one of the widely used approaches nowadays. In this research the objective was to study the Dynamic JSSP.

Chapter 2 is the Literature review where prior studies on JSSP has been discussed. Both Static and Dynamic problem has been included in detail with techniques, methods and findings in this chapter.

Chapter 3 elaborates on the Problem definition; First part of the chapter is where the problem has been identified with the requirements or the expectations regarding the solution is discussed. The second part of the chapter discusses the Methodology where the, designing of the Genetic Algorithm and Optimization Algorithm are brought into light.

Implementation details of the proposed algorithm as outlined in the Chapter 3 and other related details are elaborated in Chapter 4.

In Chapter 5, obtained test results are presented and then an analysis of the results has been carried out. It includes a simulation of a problems based on the problem definition as well. Performance of the Algorithm also has been extensively analyzed

in this chapter.

Chapter 6 is the conclusion of the findings and suggestions for areas for future studies.

Chapter 2

Literature Review

Prior readings related to solving the Job Shop Scheduling Problem (JSSP) include in this chapter. The JSSP is two fold. One is Static JSSP and the other is Dynamic JSSP. In the first section of the literature review, literature on Static JSSP is discussed. The rest is devoted to discuss the literature on Dynamic JSSP.

2.1 Job Shop Scheduling Problem (JSSP)

With increasing demand for manufacturing facilities, it is quite important to plan the resource allocations for operations so that the demanded constraints such as deadlines are met and the finish products are released to the clients with no delays. In this process, effective utilization of resources such as machines and labor while lowering the total cost of production is a tedious work. The modern needs of manufacturing, attributes the increasing complexity of the problem.

The classic JSSP (also known as Static JSSP) deals with assigning set of jobs with pre-determined order of operations per job to set of machines to minimize the time taken to finish the jobs. Therefore, JSSP can be considered as one of the most complex and combinatorial optimization problems in manufacturing systems, mathematically well known as non-deterministic polynomial hard (NP-hard) combinatorial optimization problem [1].

In classical JSSP, n jobs J_i (i = 1, 2, ... n) have to be processed on m machines M_i (i = 1, 2, ... m). Some assumptions are associated with the problem. Assumptions are as follow:

1. Only one operation can be performed by a machine at a given time (no parallel operations on a machine)

2. no preemption also known as splitting is allowed (terminating an ongoing operation on a machine with the intention of resuming it later)
3. only one operation of a given job can be performed at a given time (no parallel processing operations per job).

A well structured notation to denote JSSP is discussed in paper [2]. It uses three symbols separated by $|$ to denote the problem. $\alpha \mid \beta \mid \gamma$. α is the machine environment, job characteristics are denoted by β and γ represents the optimal criterion.

Few methods have been developed to derive an optimal solution for JSSP, where the number of machines are limited. However, there is no optimal method exists to solve this problem when the number of machines increase. Lot of work has been carried out in this area in the literature, addressing many aspects. But due to the complexity of the problem no single method proves capable of solving vast majority of JSSP with a guaranteed optimally. Therefore, the methods are problem dependent and should be carefully chosen based on the nature of the JSSP that is to be solved. The problem consists with multi variables, therefore the outcome depends on combination of values. As the problem expands, there is no way to get an optimal answer. The interest lies on finding a schedule which satisfies the given constraints.

2.1.1 Varieties of JSSP

Apart from the classical Job Shop Scheduling Problem (JSSP), there are few variants of the problem. This section briefs those problems.

Flow Shop Scheduling Problem Flow Shop Scheduling is a special case of JSSP. The restriction is in the machine order. Each job is processed in similar machine order[3]. For an example if there are n jobs, all n jobs are processed in m machines in the same order, m_i where $i = 1,2,3... m$.

Open Shop Scheduling Problem Open Shop Scheduling is one of the special cases of JSSP, the main difference is that there is no pre-determined order in which the operations of a job should be performed. Each machine performs a different operation. Only one operation per job can be processed at a time and no multiple operations are allowed on a machine at a given time [4].

Flexible Job Shop Scheduling Problem Also know as Job Shop Scheduling with multi-purpose machines. The Flexible Job Shop Problem is a generalization of the classical job shop scheduling problem which allows an operation to be processed by

any machine from a given set. These kinds of problems arise in Flexible Manufacturing Systems. The problem is to assign each operation to a machine and to order the operations on the machines, such that the maximal completion time (makespan) of all operations is minimized. [5]

2.2 Methods for Solving Static JSSP

Static Job Shop Scheduling Problem (Static JSSP) is a problem where the environment is fixed throughout the time span of the problem considered. Once the problem has been formulated it is assumed that the parameters of the environment will not be changed. In this approach disturbances to the current schedule will not be considered or assume negligible though such situation is highly uncommon in real-life situations. But this is the kind of JSSP studies first by simplifying the problem based on series of assumptions.

Static Job Shop Scheduling Problem is widely and commonly addressed. As lot of work has already been carried out in this area, there are plenty of methods covering varieties of JSSP using many different approaches. The methods can be classified based on the approach uses to solve the problem. In this sub section, some widely used approaches are discussed briefly.

2.2.1 Heuristic Approach

Static JSSP has been initially dealt by Johnson in 1954 and he has developed a heuristic method [6] to get the optimal process schedule. The method which is termed as two stage production schedules or three stage production schedules with many number of jobs could handle up to three machines. This approach can be used in the simplest form of the Static JSSP to solve 2 machines or 3 machines, for n jobs problem. Following the above-mentioned method, some methods are designed for m machines and n jobs, for example this method [7] suggests a heuristic method based on collection of heuristic methods to get a near optimal solution.

In this paper [8] four methods have been tested for the performance for 3 machines 6 jobs problem. Integer linear programming, linear programming with answers rounded to integers, a heuristic algorithm based on Johnson's method and Monte Carlo approach are tested. Johnson approximation is used to apply the two-machine method of Johnson to three-machine problem. The Monte Carlo approach is the most promising approach however there is no indication that the best solution obtained is the optimal.

Priority rule based scheduling is also another approach in heuristics. [4]. That is arranging the schedule following some pre-defined rules. Indeed, this is the widely used approach in early days. The sequencing of jobs is carried out as per the set rules. For example, widely used sequencing rule, Shortest Processing Time (SPT) rule is a priority sequencing rule which specifies that the job requiring the shortest processing time is the next job to be processed. Likewise there are rules such as First Come, First Served (FCFS), Earliest Due Date (EDD), Critical Ratio (CR) etc.[9] One rule or set of collection of rules can be used to solve the problem.

Hyper-heuristic method termed NELLI-GP [10] has been developed for solving JSSP based on rule based dispatching rules. The initial problem is divided into sub-sets of problems and then use heuristics to solve each sub-set problems. Using this method, it was able to achieve superior results on known benchmark problems in the literature.

Using methods based on Branch and Bound was also used to solve JSSP in early days. Heuristics developed based on Branch and Bound method seems effective [11]. However, when the number of machines and/or jobs increases this heuristic method has failed to obtain solutions in the given time span.

2.2.2 Neighborhood Search Method

Heuristic methods mentioned in the earlier section, are promising and popular early days. However, as the complexity increases heuristic method alone cannot produce satisfying solutions. Therefore, new ways of solving JSSP emerges. One of such methods are neighborhood search methods. Neighborhood search methods are popular among the solutions for JSSP . Shifting bottleneck method, Tabu search, Simulated annealing and Genetic algorithms are some widely used techniques related to neighborhood search method.

2.2.3 Shifting Bottleneck

Earlier priority rule based dispatching methods are used to obtain schedules for JSSP. It is a local optimization method. In order to improve the scheduling efficiency Shifting Bottleneck method has been proposed [12]. In this method, one machine at a time is sequenced. Then one-machine scheduling problem has been solved for each machine not yet sequenced. The result is used to rank the machines and to sequence the machine with highest rank. Once a machine is ranked, the already ranked machine sequences are re-considered and re-optimized. That solved the local optimization issue

increasing the efficiency of the schedules.

2.2.4 Simulated Annealing

The Simulated Annealing algorithm is commonly used for solving combinatorial optimization problems. In this research [13], decomposition based optimization algorithm is proposed for solving large JSSP. Constraint Propagation theory has been used initially to derive the orientation of a portion of disjunctive arcs. Then use the simulated annealing algorithm to find a decomposition policy. Simultaneously, each sub problem is successively solved using the simulated annealing algorithm which in return leads to a solution of the initial JSSP. Method has been tested against benchmark JSSP and results are promising.

2.2.5 Tabu Search

Tabu Search is a global iterative optimization method. It begins in the same way as ordinary local search or neighborhood search. It progresses iteratively from one solution to another until the chosen termination criterion is met. Tabu Search uses adaptive memory, as a result the search behavior is comparatively flexible. Compared to other neighborhood search methods, the Tabu Search approach keeps tracks the visited solutions in a short-term memory. As a result, this method effectively gets rid of the local optimal solutions.

When Tabu Search is applied to solve the job shop scheduling problem, it was observed that this method is much more efficient than shifting bottle neck procedure and simulated annealing implementations [14]. Dynamic Tabu Search method has used in this study [15].

2.2.6 Artificial Intelligence

Different approaches were taken to solve JSSP based on Artificial Intelligence (AI) related methods. The advantages of AI methods are using the collected prior knowledge on decision making process. Those methods are capable of developing more complex problems specific heuristics using the prior knowledge which is a step forward when compared to other heuristic methods. However, these AI methods are difficult to organize as they are highly problem specific. The AI system should be trained first to solve a specific problem, and vast number of data is required for that purpose. When there is a new problem then again, the AI system should be trained to solve the

new problem. Due to this limitation, in the literature AI based methods are combined with other methods to solve JSSP.

2.2.7 Expert Systems

Expert systems are also known as Knowledge-base Systems as well. It contains a knowledge-base and an inference engine. As the name suggests the inference engine uses the gathered knowledge to get into conclusions. The process is similar to how humans come to consultations using their prior experience and knowledge on that specific problem and the domain. The specialty is, Expert systems can handle more knowledge in a higher processing speed than a human can do due to the advancement of the processing power of computers.

ISIS [16] is the first major expert system specifically designed for solving JSSP using three level, hierarchical constrained-directed search approach. The method uses five constraint categories out of three are main constraint categories as mentioned below.

- **Organizational Goals** – Determines measures of how the organization should perform. For example, due dates, Work-In-Process (WIP), Resource Level etc.
- **Physical** – Specify characteristics which limit functionality, for an example there is a maximum limit for the work-piece that can be formed in a CNC machine. Work-piece larger than the maximum specification cannot be formed in that specific CNC machine due to the workbench limitation on its dimensions.
- **Casual** – Define what conditions must be satisfied before initiating an operation. For an example, precedence of operations of the job, resource requirement, machine availability etc.
- **Preference** – This can be identified as a resource selection criterion, for an example when selecting a machine for an operation if there are more than one machine available, then operator might prefer to select one machine over the other considering a parameter (if the machine are not identical) such as quality of the output etc.
- **Availability** – the availability of a resource for the time-period of the operation.

2.2.8 Distributed AI – Agents

When solving large-complex problems, single AI agents are less effective due to their limited knowledge and limited capacity on problem solving. Instead of having a

single AI agent (or an expert system or a knowledge-base) to solve the problem, in this approach several AI agents are employed.

Single agent in the networks acts as an expert in a specific domain. DPSN interconnects single intelligent agents which are considered as problem solving nodes which can solve a part of the problem related to the domain of their knowledge and the solution is communicated to the other sub-problems.

Compared to heuristic methods discussed earlier, Expert systems are less popular for solving JSSP as organizing problem specific knowledge-bases and maintaining them is quite challenging.

2.2.9 Artificial Neural Network (ANN)

Artificial Neural Network (ANN) is a concept of processing data inspired by the way biological nervous system works. Neurons are the building blocks of the human nervous system. In ANN this system is simulated using processing power of computers. Neural Networks are capable of identifying patterns based on set of data, which are complex for humans to notice or even for another computer system.

Another remarkable feature of ANN is the ability to learn. Apart from those, self-organization, real time operation and fault tolerance can be identified as key advantages of using ANN. However, using ANN is not comparatively yielding better results against conventional methods. The parallel processing is an advantage in this specific method. Further studies are required in this filed [17].

Swarm Intelligence Swarm Intelligence (SI) is the collective behavior of decentralized, self-organized systems inspired by the behaviors of the insects in the nature.

Particle Swarm Algorithm Particle Swarm Optimization (PSO) algorithm is a method for optimizing continuous nonlinear functions using random optimization approach based on SI. The inspirations come from a motion of flock of birds searching for food [18]. In this method, each particle remembers the best position of the particles and shares the information between particles. The optimization is achieved through cooperation and competition between the particles of the population.

This algorithm is tested against the benchmark JSSP and it was able to achieve optimal results [19]. Due to the fact that there is no guarantee on global convergence in traditional PSO algorithms, much improved version of the concept is developed in the literature as well. A hybrid PSO algorithm has also been developed. PSO algorithm has been combined with Giffler and Thompson algorithm (known as G&T Algorithm)

[20] to avoid local optimal solutions stagnation. To avoid the local optimal solutions, an improved version of PSO has been developed [21]. The performance of this algorithm is compared with the results of the benchmark problems solved using Genetic Algorithm and tabu search. The results are inline with the results obtained using the other methods.

Hybrid algorithm [1] is developed combining PSO algorithm and the Artificial Immune System algorithm. It was possible to obtain comparable results for benchmark problem and obtained optimal solutions for some benchmark problems. Therefore, PSO (highly effective when used with other algorithms) can be identified as a reliable approach to solve the static JSSP.

Ant Colony Optimization Ant Colony optimization is a stochastic optimization algorithm, a model derived from the study of artificial ant colonies and inspired by the studies of the behavior of the real ants[22].

Ant Colony Optimization has been used to solve the JSSP in few studies. If an isolated ant is considered, it moves according to a local, greedy heuristic. It generates local optimal moves, the tour followed by the ant consists of good and bad parts. Therefore, instead of an isolated ant, now the presence of many ants is considered simultaneously. Each ant contributes to a part of the tour distributions. Good parts are followed by many while the bad parts are followed by few. Due to that effect the process converges to a good solution without getting stagnated in local optimal solutions [23]. However the Ant system easily deals with combinatorial optimization problems defined on non-symmetric graphs.

The time-complexity of Ant systems increase exponentially when the population size increases [24]. That is the main disadvantage for Ant System approach. In the same study, it is suggested that Genetic Algorithm approach will be more suitable for optimization problems that can be defined by symmetric graphs.

Bee Colony Optimization Bee Colony algorithm is inspired by the behavior of the bees, same as the Ant Colony algorithms discussed earlier. In the literature, attempts were made to use this approach to solve JSSP. Results are in line with the Ant Colony approach but less effective than the Tabu Search methods [25].

2.2.10 Evolutionary Algorithms

Evolutionary Algorithms are inspired by the Davinchi's biological evolution.

Genetic Algorithms (GA) Relatively more work has been carried on applying genetic algorithm to solve static JSSP in the literature. Genetic algorithm is first used in this study to solve JSSP [26].

In the following study [27], it has been studied on how to apply the conventional genetic algorithms through representation, evaluation and survival to solve the JSSP. Results obtained for benchmark problems in this study shows that the results are as good as the results obtained using branch and bound method.

GA method is a general optimization method. Therefore in order to be effective the method should be altered to align with the problem. It has been studied that the effectiveness increases when problem related operators are introduced. It has been proposed a multi-step crossover (MSX) method which uses neighborhood structure and a distance in the problem space [27]. It attributes promising performance development over the new operator.

However even though the scope of the problem can be addressed with Genetic Algorithm method, GA are not good at finding the global optimal solutions as it tends to trap in local optimal, [28] as a result it is advantageous to couple GA with a local search algorithm such as Tabu search, Simulated Annealing etc. Finally in the local search the final solution depends on the initial solution.

Hybrid algorithms are developed [29] to avoid premature convergence, in other words getting trapped in the local optima. In this research Single Genetic Algorithm and Parallel Genetic Algorithm are used. Instead of applying GA operations to a single initial population, in this method various sub-populations are used simultaneously, these sub populations evolve independently, hence it is called Parallel Genetic Algorithm. Having independently evolving populations on genetic Algorithm also known as the island model is studied [30] and proven effective to avoid premature convergence. This approach was tested on five benchmark JSSP problems and achieved good performances. It was observed that processing time is limited, this method achieved effective superior solutions.

Often ANN and GA combined used to solve problems, same approach is taken in this research, developing a hybrid model of ANN and GA [31].

Compared to other approaches, in the literature it seems like GA is one of the methods extensively used to solve JSSP.

2.3 Method for Solving Dynamic JSSP

Unlike the Static Job Shop Scheduling Problem (JSSP), Dynamic JSSP handles uncertainties in scheduling which is very common in real life scenarios. Although JSSP has received great attention in the literature, the usefulness of the classical scheduling is limited in the real world. Many studies in the literature were trying to solve the classical JSSP which is discussed in the earlier sub section under solving Static JSSP. In order to fill the gaps between classical scheduling and the practical use of the scheduling, the dynamic behavior of the factors which influence the manufacturing processes should be considered. Otherwise the schedule formulated for the Static JSSP becomes useless once a single change occurs in the ongoing manufacturing process.

Dynamic behavior of the factors related to manufacturing change as information related to the JSSP changes time to time. This can be identified as a disruption to the ongoing schedule. In static JSSP, these disruptions are ignored, based on the assumptions. But in the real world, possibility of occurring such events are high and cannot be simply ignored or compensated based on assumptions.

The parameters involve can be deterministic or stochastic. If the change of parameters can be assumed those parameters can be considered deterministic. For example, arrival of new jobs might roughly follows poison distribution[32]. On the other hand, if a parameter is stochastic, it is not possible to assume the degree of the change of the specific parameter. Sudden machine breakdown can be taken as an example.

These disruptions can be identified into two categories.[33]

Data Uncertainties – Processed data plays a major role in the scheduling process, for an example the time required to complete an operation in a machine. Operation time is merely an estimation based on prior experience and/or data. However the time required can be vary from the estimated time. The time depends on the condition of the machine and may be unseen complexity could be occurred which are not encountered at the estimation process.

Real-time Events – This category can be further divided into two sub categories.

- **Disruptions during Execution** – Disruptions can be occurred during an execution of an operation such as haphazard machine breakdown. The operations could be partially completed at that time.
- **Unpredictable Events** – Unpredictable events might occur as the schedule processes. These are not haphazard but influence the schedule, for an example arrival of new jobs in a continues and stochastic pattern can be considered. In such situations, most of the time the operations that are currently being processed in a

machine might not be disturbed, but the active schedule needs to be re-evaluated to check how the new requirements can be accommodated.

Again, in the real world scenario, most of the time not a single disruption occurs. Most of the time many disruptions can be encountered simultaneously. Problems associated with more than one change in the scheduling are known as Multi-Constraint Dynamic JSSP. Therefore, to be effective, multiple disruptions could be handled at once, understanding the effect of those disruptions to the ongoing schedule. Sometimes disruptions are linked, one disruption can trigger one or more other disruptions. For an example, shortage of machine operators due to high work load leads to the reduction of the efficiency hence increase operation times than estimated values.

As noted when discussing the Dynamic JSSP, the classical static JSSP Solutions become useless as there is no mechanism to respond to the developments/ changes/ requests. There is a gap between the schedule derived from the theories of the classical problem and the practical usage [34]. The paper discussed the use of real-time information in scheduling.

Uncertainties can be categorized into three categories [35].

Completely Unknown – Highly unpredictable events which are completely unknown till such an event takes place.

Suspicions about the future – These kinds of events are quite predictable with the experience.

Known Uncertainties – These are the events where some information is available at the time of scheduling. Such events can be predicted. For example, machine breakdown can be considered. With available information machine breakdown might be predicated.

Events which are completely unknown and suspicious about the future are hard to accommodate in the scheduling process as they make the Dynamic JSSP much more complex and demand complex computation.

However, it is not practical to consider all the disturbances at the scheduling phase as it is not possible to consider all the uncertainties that could occur. It is required to understand the impact of the disruptions and then priorities their effect on meeting the objectives of Dynamic JSSP.

Typical objectives of Dynamic JSSP can be categorized into three main categories [33].

- **The Shop Efficiency** – Among the several objectives relate to job shop efficiency, completion time based and tardiness based objectives are commonly

used. Completion time is the time taken to complete a job from the time it is arrived the shop. Tardiness on the other hand, is the penalty on delayed jobs. When the completion time is greater than the due date a penalty can be imposed. Maximizing flow time, maximizing the utilization of machines, minimizing machine idle time, minimizing the total waiting period can be identified as other objectives related to the shop efficiency.

- **Robustness** – Due to data uncertainties in dynamic JSSP, robustness measures the sensitivity of the schedule quality to disturbances. A schedule is robust if it's performance deviates in a small amount under disruptions. [36]
- **Stability** – Once a disruption takes place the current schedule needs to be rescheduled to accommodate the change. Stability is a measurement that measure the deviation from the original schedule to the revised schedule.

It is required to understand the trade-off between the objectives and select one or few objectives to carry out the scheduling effectively.

In the literature, Dynamic JSSP has been defined using following categories. Completely Reactive Scheduling – This method uses local scheduling most of the time using priority rule based methods. This method is not suitable as the global optimally is not considered.

Robust Pro-active Scheduling – In this method disruptions are predicted, and those disruptions are accommodated in the schedule. As a result, even though a disruption occurs it does not affect the schedule as the effect is minimal. The greatest challenge in this method is to determine the disruptions. Widely used in accommodating machine breakdowns. Policies, Frameworks are developed [37] to encounter this issue.

Predictive-reactive schedule – This is the most common strategy used in the literature. It accommodates revising the schedule in a presence of a real-time disruption. However as rescheduling takes place each time when a disruption occurs leads to bad performance. Therefore, the focus is to generate robust predictive–reactive schedules that minimize the effect of reschedule on performances. A typical solution is to reschedule considering the schedule efficiency and the schedule stability simultaneously. [16]

Finally, once a disruption occurs the schedule should be revised. As discussed earlier if the schedule is robust enough no modification will be required. However due to the nature of the real-life events modifications might be required. In such situations it is important to understand when-to-reschedule and how-to-reschedule.

When-To-Reschedule – In the literature three main methods can be identified to solve this problem.

Periodic Scheduling – In this method schedules are periodically generated taking information available at that specific instance. The advantage of this method is decomposition of the Dynamic JSSP into series of Static JSSP and solve it using classical Static JSSP solving methods. Once the schedule is generated for the period there will be no revision till the next periodical schedule generation. The time intervals can be fixed or variable.

Event Driven Scheduling – In this method real-life disruption triggers the rescheduling. For example, once a new job has been arrived the existing schedule is revised with a newly scheduled schedule using the information available at that instance. A slight variance of this method is known as adaptive scheduling in which the rescheduling takes place after predetermined amount of deviations from the original schedule is observed. For example, if series of events cause the schedule deviation of 2 days compared to the original schedule, that triggers rescheduling. [x11]

Hybrid Scheduling – A combination of Periodic and Event Driven method can be identified as the hybrid method. As an example, the period scheduling is carried out in specific time intervals. However, if a real-life event occurs which is considered 'critical' then a rescheduling takes place immediately. [need10]

Hybrid and the Period methods come under rolling horizon policy. However most of the dynamic scheduling use Event Driven approach.

How-To-Reschedule – Determining how to revise or update the schedules at a reschedule, also knowns as rescheduling strategies. Two methods are schedule repair and complete reschedule. In schedule repair, the schedule is locally adjusted to counter the change. On the other hand, the schedule will be completely revised in complete rescheduling method. Complete rescheduling is an optimal way to handle the change. However, the new schedule might me completely different to the original schedule. Such situations might contribute shop floor nervousness and/or additional costs due to drastically changes. [need12]

It is possible to find some of the literature for methods of solving Dynamic JSSP. However due to the complexity of real world events the nature of the events, one problem is different from one another.

2.3.1 Heuristics

Heuristic methods are widely discussed in the literature to handle rescheduling due to the simplicity. Heuristic approach, priority rule based scheduling can be used to find a solution for dynamic JSSP in a short amount of time as it does less calculations relatively. [need11]

In a research where the performance of schedule repair using right-shift heuristic and complete rescheduling using branch and bound method. The result shows that the right-shift heuristics outperformed the other approaches. However, it is known fact that the right shifting method increases the makespan as it moves the current schedule forward by the amount of the disruption. Therefore, even though the method is less complex, it is not the right way to address the dynamic scheduling.

In order to overcome the issue in the above method, it is required to focus on a complete reschedule while matching the new schedule to the preplanned schedule as well. Such a method to match up the schedule[29] was developed. The term Matchup Schedule Algorithm (MUSA) uses integer programming approach and alternatively priority rule approach as well. MUSA assumes that the processing and setup times can vary continuously, and an optimal matching schedule can be found. However, in real-life scenarios it is not true always. However, the match up schedule strategy is effective in terms of schedule quality, computation time and schedule stability.

Studies has carried out to study various factors, as an example the queueing time of the remaining operations has been considered in the following study even though the main objective is to meet due dates per jobs. Consideration of queue time achieved significant improvement over the methods that consider meting due date as the only criterion.

In complete reactive scheduling, variety of dispatching rules have been used to deal with real-time events. Simulations [32] were carried out to assess the performance of various dispatching rules under different dynamic and stochastic conditions. Variety of dispatching rules were examined against common performance criteria discussed in the literature, such as minimum and maximum flow time, mean tardiness, maximum tardiness and variance of flow time, etc. The study shows that no rule performs well in all the situations. Therefore, it can be concluded that according to the nature of the problem, rules that should be selected vary.

Therefore heuristic approach to solve dynamic JSSP cannot be considered a promising approach.

2.3.2 Metaheuristics

In the literature, it is possible to find out meta-heuristic methods used to solve dynamic JSSP successfully. Meta-heuristics are high-level heuristics which helps heuristics to escape from local optimal. Meta-heuristic methods such as Tabu search, simulated annealing, genetic algorithms and varieties of these methods are widely used in static JSSP but little work has been carried out in dynamic JSSP.

2.3.3 Simulated annealing and Tabu search

Simulated annealing method has been used in iterative repair schedule method, the inspiration was to avoid cycles and local optima. Another study [30] on uncertainty of operation times, Tabu search is used to escape from the local optima. Both studies show the importance of using meta-heuristics to avoid getting trapped in local optimal solutions.

2.3.4 Genetic Algorithms (GA)

Genetic Algorithms (GA) is a meta-heuristic approach to solve the Dynamic JSSP. This method is the most widely used approach to solve Dynamic JSSP in the literature next to heuristic methods such as methods based on dispatching rules.

Solving Dynamic JSSP considering arrival of jobs has been studied [38] in this study. Both deterministic and stochastic approaches are considered. The stochastic approach, tested under various manufacturing environments with respect to machine workload imbalances and tight due dates. In this study GA used to solve Static JSSP has been slightly modified to accommodate the dynamic nature. Two approaches of rescheduling which are repair schedule and complete reschedule has been studies. In complete reschedule a new JSSP has been generated and the rescheduled from the scratch. However, in repair schedule approach, the last population of the earlier schedule was taken with few modifications as the initial population of the new problem. This is a special feature in GA observed in the following study by Bierwirth. [39]. However in the same study[34] they have incorporated G&T Algorithm to further modify the adapted population. Repair schedule approach is performing better than the complete reschedule according to the results of this study.

Same result has obtained in this study[41]. Order-Based operators combined with the G&T repair method (hence the method called OBGT) performed well in large problems. The study suggested future studies to implement parallel version of OBGT using Island Model.

Another gift [39] has focused on alternate job routings, unreliable machine (machine breakdown) and multiple scheduling criterion. The method is tested on various job shop conditions such as alternate number of suitable resources to process a job operation, tightness of the due dates and inclusion of machine breakdowns/ repairs. The method was tested against some selected dispatching rules such as SPT, FIFO, LIFO and random dispatching rule. Results indicated that the performance of the processed method is superior than the dispatching rules heuristics [41].

In fact, Genetic Algorithms are widely used in the literature due to promising performance factor. Genetic algorithms based on Bottleneck Resource is introduced using Intelligent Agents for the dynamic JSSP. In the research [5] to perform bottleneck local search on the selective Chromosome s of the population an AI has been used, termed as Bottleneck Local Search Agent (BLSA).

Methods have been developed to solve dynamic JSSP combining meta-heuristic approaches. Such as using heuristic rules to formulate the initial population[44-SPECIAL]. In this research Genetic Algorithm has been combined with tabu-search [40] to solve a multi-objective Dynamic JSSP. The results are evaluated against the schedule efficiency and the schedule stability. When job arrival rate goes high the method shows some performance drop in terms of schedule efficiency, but it has performed well in other circumstances.

2.3.5 Artificial Intelligence (AI)

Study has carried out introducing Artificial Intelligence (AI) based method to solve dynamic JSSP. Multiple AI agent based [41]method has been introduced. The method considers different manufacturing environment conditions such as loading factor, stochastic machine breakdown and job arrivals. The task assignment mechanism is distributed and executed through the communication network for inter-cell scheduling and a knowledge-based system for cell-level scheduling.

Many studies found in the literature have engaged AI based methods to improve the performances of the main method. AI based method is used in conjunction with GA [42] hence termed as Genetic Algorithm-based Machine Learning (GBML). GBML is an automated learning process to discover condition action rules which are IF-THEN clauses to perform desired actions. In this method GA generates and develops condition action rules. Using reward mechanism, the strength of successful condition action rules are increased and the strength of unsuccessful condition action rules are decreased. GA try to improve the overall performance by replacing unsuccessful rules with successful rules. Learning is achieved generating condition action rules that maximize the amount of rewards.

2.3.6 Artificial Neural Network (ANN)

Artificial Neural Network (ANN) has been used to solve Dynamic JSSP. Approaches such as Hopfield type ANN has been used in the literature [43]. ANN adopts learning based on the prior data. However as the ANN should be trained per each set

up and it takes time and should available data to perform the training. It is a drawback of ANN.

Most of the time ANN are used with combination of another algorithm. Genetic Algorithm-based Machine Learning (VNS) [44] and ANN is used to develop an effective method to solved multi objective Dynamic JSSP [45].

2.4 Chapter Summary

In this chapter, the study on the literature related to Dynamic JSSP has been discussed. The different methods of solving static and dynamic JSSP has been studied extensively.

Chapter 3

Methodology

Chapter defines the problem and discusses the methodology. A brief overview of Genetic Algorithms (GA) related to Job Shop Scheduling Problem (JSSP) is also mentioned in the beginning of this chapter. Afterwards the design of the algorithm is discussed in detail.

3.1 Problem Definition

As described in the previous chapter, deviations can be expected in a job shop for the ongoing schedule due to various reasons, termed as disruptions. Due to the complex nature of modern businesses, dynamic scheduling plays an important role. Unlike in the classic JSSP, in Dynamic JSSP, the focus is to address the dynamic nature of the Job Shop and the orders (jobs). Deviations could occur due to number of unseen reasons. The status of the Job Shop changes with the time. Once such deviation occurs, a new schedule has to be generated to meet the set expectations, specially parameters like order fulfillment dates (due dates). The objective is to manage several disruptions affecting a schedule simultaneously.

The initial problem is based on n number of jobs and m number of machines in the manufacturing environment which operates as a Job Shop.

Assumptions are as follows;

1. Each job contains pre-defined, ordered set of operations which needs to be carried out in order.
2. A machine can only perform one specific task at any given time. Machines are not multi-tasking (run more than an operation at once) nor diversified to handle variety of jobs.

3. No preemption is allowed when scheduling. (it is not possible to pause an on-going operation on a machine and then resume and/or assign another operation till the current operation on the machine is completed).
4. Time required for a given operations is known before hand.

Scheduling is carried out based on the above assumptions while following Disruptions are considered.

1. **Stochastic Job Arrival** – Job Shop should be able to accept new jobs as those jobs arrive. A level of priority has been assigned to each and every job. When scheduling, critical jobs are given a high priority level.
2. **Stochastic Machine Break-down** – Machine breakdowns are not deterministic but stochastic. A break-down could occur at any given time. Once such incident occurs, a requirement arises to re-evaluate the schedule with the available new information.
3. **Unplanned Machine Maintenance** – Machines might experience unpredictable maintenance requirements. Those are not planned maintenance activities. Such incidents, if possible, should be dealt in a manner in which the machine maintenance would not disrupt the currently processing operation. However, the machine should be maintained once the current operation finishes. In such a situation, a new schedule is required to be generated if the machine is kept unavailable for a longer time. In other words the algorithm should be able to handle future maintenance of machines which are in operation (such information is not available at the time of generating the initial schedule).
4. **Other Disruptions** – Apart from the above-mentioned forms of disruptions, there can be number of other factors that demand a schedule change. Change in operational time due to a newly discovered fault of a machine, skilled operator shortage due to a sudden unavailability of operators etc. are some common situations to note. However, these disruptions can be modeled as one of the disruptions (1,2 and 3) mentioned above.

When an aforementioned disruption occurs, the schedule needs to be evaluated again incorporating the information of the new change. Thus, a new schedule subsequently generated and followed-up until another rescheduling incident takes place. A time instance of demanding a rescheduling when a disruption occur is termed as a Rescheduling Point (RP) on the schedule. The threshold of the RP needs to be decided based on the control required.

3.1.1 Disruption Demanding A Schedule Change

Stochastic Job Arrival While the current schedule progresses, new jobs can be arrived at the facility, making the current schedule no longer valid as it doesn't facilitate the new jobs. Once there is a new order in the Job Shop, a new schedule has to be generated considering the parameters of the new job. For example if the criticalness of the newly arrived job is higher, it should be given higher priority among jobs that are already being executed. As long as the new jobs satisfy the basic requirements and assumptions of the JSSP, such jobs can be included in the new schedule.

In most of the practical scenarios, new jobs cannot be started directly as and when it is received to the facility, due to other resource constraints such as material handling, resource allocation etc. Allocating required raw material is one of good examples. Normally manufacturing facilities do not store required material for all of the future needs and when required such materials will be ordered from the supplier. Therefore, it should be possible to schedule operations considering a future start date/time. The usual practice is to terminate the schedule when the new operation of the new job is initiating. However, it might be beneficial to check if there is an added advantage of terminating the current schedule before the new operation physically initiated. Of course, the new schedule should be generated including the newly arrived job and its information. Often such simulations could help the planners to decide which schedule to follow. Should the current schedule be terminated with immediate effect or in a future date/time.

For an example if the current time of the schedule is T_1 and due to resource constraints on raw materials, the earliest projected starting date of the newly arrived jobs could be T_2, then it should be possible to simulate and check if it worths terminating the current schedule at time unit T_3 which is in between time T_1 and T_2. If there is an overall effect of terminating the current schedule on T_3, the planner then can make the decision of terminating the current schedule early.

Unplanned Machine Break-Down Machine breakdown is an unpredictable disruption on ongoing schedules demanding a generation of a new schedule. At the time of the machine breakdown, an ongoing operation could be getting partially completed on that machine. Therefore, the current operation should be continued once the machine is repaired (as per one of the assumptions made above). New operations on this machine can only be scheduled after the ongoing operation is finished, once the repair work is finished.

However, in many practical scenarios, it is beneficial to complete the operations of

the current schedule up to some extend without those operations getting rescheduled. For an example, consider an instance where the required material for the next operation of the machine has already been received to the facility. Once the machine breaks down at the current operation, the option is to generate a new schedule taking the machine repair time into consideration. Given that the related job (the next job for which raw material has already received) is re-allocated in the new schedule in an different date, there might be a cost involved in handling material which is an unnecessary additional cost if it is decided to adopt the new scheduled time for the operation. In such a situation, without an argument, it is beneficial to generate the new schedule keeping the next operation fixed. When scheduling, the next operation is ignored and the other operations are scheduled accordingly. The time required to complete the fixed operation should be allocated on the respective machine as only one operation can be executed at a given time.

This requirement leads to the concept of fixed operations in the schedule. This is one of the features that can be used in the planning and simulation process. Because some operations cannot be scheduled as required. Such operations should be fixed and ignored by the scheduler. However no other operation should be allocated to that machine in that time period. Therefore such operations cannot be simply ignored but need provisions to handle such requirements in the algorithm.

Unplanned Machine Maintenance In some situations, even though a machine in the current schedule requires a fix, it can afford to wait for few time units or at least till the current operation is completed. In such situations immediate termination of the schedule might not be required. If an immediate termination is demanded, then it falls to the scenario explained above, unplanned machine break-down.

Therefore, it should be possible to simulate the future outage of a machine and generate a new schedule to facilitate the future outage of the machine.

This is similar to consider planned machine break-down at the initial schedule generation stage. However, the current ongoing schedule is to be terminated at a time point where at the same time most probably the other operations are being partially completed on the other machines. For those machines, the problem can be modeled as an unplanned machine break-down of zero duration. The new schedule should be generated considering current operation completion amount in such operations as well.

3.1.2 Conditions to Satisfy When Scheduling

The generated schedule should be a feasible schedule for all machines. Apart from being feasible, the schedule should meet set expectations. Such expectations are discussed in detail in this section.

Meeting Due Dates : Completing the jobs before the set deadlines is one of the expectations and responsibilities of a planner. Therefore, generating schedule should be optimized considering minimization of makespan which is minimizing the time required to complete jobs. However, when tight due dates are imposed, it might not be possible to meet all of the due dates for all the jobs. In such scenarios, it should be possible to minimize the total tardiness (decrease the lateness of jobs), also known as minimization of total tardiness. However, minimization of makespan for critical jobs is required, in the presence of tight due dates minimization of total tardiness of critical jobs can be considered.

Schedule Efficiency and Productivity : Generated schedules should be efficient. The first objective is to meet due dates and then the focus should be increasing the schedule efficiency. In order to increase the schedule efficiency, it is required to eliminate machine idle time in between the operations of the machines. In return that helps reducing the makespan of the jobs as well. However, this becomes quite challenging when disruptions occur often demanding schedule changes eventually. The objective is to maximize the utilization of a machine.

3.1.3 Requirements to Satisfy

According to the discussion in the above section, an algorithm is to be developed to address the following requirements in summary:

1. Dynamic JSSP to solve for n jobs, m machines problem. Operations are sequences and the number of the operations per job can be vary.
2. The solution should be capable of handling stochastically arriving new jobs.
3. The solution should be capable of handling unplanned machine break-down and unplanned machine maintenance in an ongoing schedule. Once a machine break-down occurs no preemption is allowed on operations.
4. The solution needs to handle future machine unavailability at the scheduling stage and starting time fixed operations so that such operations kept unchanged when a new schedule is being generated. Such feature would be highly useful in simulations.

5. Finally, considering all the requirements and constraints, schedules should be feasible schedules meeting the set expectations.

3.2 Methodology

The problem define in the previous section should be dealt with the proposed algorithm. In this section the methodology for developing the algorithm is discussed.

The proposed method is based on Genetic Algorithms (GA). The purpose of using GA was noted in the literature survey such that GA is the heavily used meta-heuristic method to solve these kind of optimization problems. It is considered robust over other approaches to solve Job Shop Scheduling Problem (JSSP) and thus used in this research as well. However the main reason for the selection of GA for the implementation is the reliability of this method over the number of complex scenarios. Not depending on the problem formation, this approach is stable when generating solutions. This algorithm is capable of searching the global solution space effectively. However, it was found in the literature survey that there has not been a significant amount of work carried out to solve Dynamic JSSP using GA covering the scenarios mentioned in the problem definition.

GA by design is capable of exploring global solutions for a given optimization problem, specially the solution space contains nearly infinitely large, feasible combinations. However, it has two main drawbacks, as GA in elementary form is not competitive among other meta-heuristics. GA is good at exploring solutions in the global solution space, but it is not capable of identifying local optimal solutions within the identified subset of global solutions space. GA needs several iterations to identify such local optimal solutions and there is a high possibility of missing the optimal solution as well [46]. The other issue is, once such local optimum has been found there is a tendency of getting trapped in that local optimum without exploring the other solutions in the solution space.

Therefore, to make it more effective, it is required to associate the GA with a local search algorithm to explore local optimal solutions within the feasible local solutions space. A way of getting rid of local optima is also required. In this research, GA is used to explore the solutions space and a local search algorithm to optimize the results making the resultant output is much effective than using the GA in isolation.

3.2.1 GA related to JSSP

GAis an algorithm inspired by the process of "Natural Selection" necessary for evolution or the "Survival of the fittest" [47]. The intelligent use of random search is used in GA to generate high quality solutions for optimization problems. Randomized, exploitation of historical information is used in GA to direct the search into a better performance space.

GA simulates the survival of the fittest through consecutive generation of individuals who are the points in a search space and are also feasible solutions to the problem. Then the population of individuals get subjected to a process of evolution. Each individual (Chromosome in genetic analogy) is coded as a finite length vector of components or variables (Gene in genetic analogy). The individuals or solutions are then given a fitness score based on the ability to compete where the GA aims to produce offspring better than parents through selective breeding of highly fit parents. As offspring is produced, individuals in the population die and replaced by new solutions to keep the population at a static size. This ensures the survival of the fittest. The better generations thrive while the generations with least fit die out.

Eventually, when the population reaches to a point where offspring is not noticeably different from the previous generations, the algorithm itself is said to have converged to a set of solutions to the problem at hand.

The solutions are mapped to a structure called Chromosome. The genetic operations are applied to the Chromosomes and evolve them. In common approaches to solve JSSP using GA, operation order based Chromosome structure has been used in the literature. This is a widely-used and also efficient method of representing the Chromosome for Static JSSP (also known as the classic JSSP).

3.2.2 How GA works

This section is a brief introduction of Genetic Algorithms (GA).

The first step of the algorithm is to generate a population. This is randomly generated. That ensures a wide variety set of solutions to start with. Those solutions are ranked against a fitness criteria, so that the fitness of each member of the generation is known.

After the random generation of the Population, the Algorithm evolves through 3 operations;

1. Selection
2. Crossover

3. Mutation

Selection - During each successive generation, a portion of the existing population is selected to generate a new generation where individuals are selected based on their fitness values. Certain selection methods rate the fitness of each solution to find the best solutions (individuals) while some rate only a random sample of the population, since the former is time-consuming. The fitness function is defined over the genetic representation and measures the quality of the represented solution. The fitness function is always problem dependent where in some problems, it is hard or even impossible to define the fitness expression. In JSSP context, most of the studies, a simulation has been carried out to identify overlapping of machine allocations and to calculate the fitness value which is most of the time the makespan, tardiness etc.

Crossover - Crossover operator distinguishes GA from other optimization techniques. Crossover is a genetic operator used to vary the programming of a Chromosome or Chromosomes from one generation to the next. Since it recombines portions of good individuals, the process is likely to create better offspring. There are several methods for selection of the Chromosome s.

a) **Single Point** All data beyond a selected crossover point is swapped
b) **Two Point** Data between the selected crossover points is swapped
c) **Uniform and Half Uniform** Fixed mixing ratio between two parents is used. Unlike single-point and two-point crossover, the uniform crossover enables the parent Chromosomes to contribute the gene level rather than the segment level enabling more complete search of a design space.

Ex: If the mixing ratio is 0.5, the offspring has approximately half of the genes from first parent and the other half from second parent while cross over points can be randomly chosen.

In the half uniform crossover scheme (HUX), exactly half of the non-matching bits are swapped [48].

Mutation - To maintain diversity within the population and inhibit premature convergence, concept of mutation is used in GA. A randomly decided portion of the new individuals will have some of their bits flipped. However, the probability of flipping should be set low. Unless otherwise the search will turn into a primitive random search.

Among the types of mutation are;

a) **Bit string mutation** - Bit strings are flipped at random positions.
b) **Flip bit** - Inverting the bits of a chosen Chromosome .
c) **Boundary** - Chromosome is randomly replaced with the lower or upper bound.
d) **Uniform** - This operator replaces the value of the chosen gene with a uniform random value selected between the user-specified upper and lower bounds for that gene.
e) **Non-uniform** - The probability that amount of mutation will go to 0 with the next generation is increased by using non-uniform mutation operator. It keeps the population from stagnating in the early stages of the evolution. It tunes solution in later stages of evolution.

After Mutation operations, the selection process makes sure a new generation is selected and it is called the off-spring. The process repeats until there is no significant improvements of the generations. This is a brief explanation of how GA works.

3.3 Design of the Algorithm

The development of the proposed algorithm is elaborated in this section. It's broken down to basic steps of a GA which are;

1. Chromosome representation
2. Generating initial population
3. Crossover and mutation operations
4. Selection of the new generation
5. Termination criteria

3.3.1 Chromosome Representation - A novel approach

The first step in GA is the representation of the Chromosome. A Chromosome is a feasible solution to the problem. The Gene is the building-block of the chromosome. Genotype is the term used to describe the representation of the solution in the algorithm.

Basically, there are two main methods of representing the genotype in GA. They are Direct and Indirect representation[49]. As per the literature, however the direct representation is the widely used approach to represent genotype based on machine order. This method is faster as there is no encoding.

The research started with a direct representation and the genotype is coded according to the operations for jobs. This is one of the common ways found in the literature.

However, it is realized that use of this representation contributes lot of infeasible solutions when it comes to disruptions handling in Dynamic JSSP. Lot of computation power is required to identify infeasible solutions, therefore is it is not computationally profitable. Due to the above fact, in this research, the indirect representation is used with an encoding.

Each operation is associated with a starting time unit of the specific operation which is referred to as Starting Time Unit (STU). Genetic operations such as crossover, mutation is also being carried-out on this value. The objective is to identify the optimal operation start time for each operation which directly leads to optimizing the entire schedule. Whereas use of operation order is the most common approach which did not work as expected in this problem. Use of such encoding was not found in the literature therefore it is decided to check how this new representation performs. In fact this is a novel concept introduces in this research.

The main reason of eliminating other representations and adopting a new encoding based on time, is a point of importance in this study. As time being a major factor in a schedule, the variable now can be used to optimize the schedule. Time plays a vital role in schedules without an argument. However in Static JSSP, information on time is not a necessity at the first place. Based on the duration of the operations starting time units can be calculated. In contrast, in Dynamic JSSP, it is quite crucial as it deals with time as an vital information.

On the other hand, this method helps elimination of generating infeasible schedules. It saves a considerable computation time in contrast to other approaches which generates infeasible solutions and then later correct them or discard them. In the Dynamic JSSP, there are lot of constraints such as machine outages, fixed operations etc. On top of that it is required to maintain the operation order of the job. As a result, the chances of getting infeasible schedules are higher. Therefore, generating solutions and later correcting them is not an efficient method when Dynamic JSSP is considered. This is one of the main reasons to develop an alternative method to manage the problem, introducing STU to the chromosome structure. Introduction of the STU increases the efficiency of the algorithm by reducing generation of infeasible solutions to zero.

3.3.2 Generating Initial Population

Generating the initial population is the starting point of a GA. It is considered as the first generation which is used to reproduce the next generation of the population. Generating the initial population is a vital step as it evolves through the entire scheduling process. In other words, the quality of the initial population affects the quality of

the output of the entire process. Chromosome is a collection of Genes. A gene can be mapped to an operation on a machine.

Since the population is a collection of Chromosomes, the objective is to generate Chromosomes. An individual Chromosome in a population is termed as a ”Member” of the population. There are plenty of techniques found in the literature in generating a Chromosome.

Rule based techniques have been used throughout the literature for the same purpose (discussed in Chapter 2). As already mentioned, Chromosomes are solutions, feasible schedules and therefore generating a Chromosome can be considered as generating a schedule.

It is also important to figure out the kind of schedule required to be developed. In order to maximize the randomness of the solutions, it is possible to generate schedules randomly. The drawback of this method is many infeasible schedules is getting generated. Therefore, this method is not a good approach on JSSP, especially in Dynamic JSSP unless a mechanism is in place to eliminate the generation of infeasible solutions.

Schedules can be divided into two main categories which are feasible and infeasible schedules. Feasible schedule has the correct operation job order in JSSP and importantly there is no overlapping operations or machine resource allocations where as in infeasible schedules, both can be contained[45]. Extra computation is required to filter-out the infeasible solutions thereafter, which is not an efficient method of handling the problem.

Active schedule means, there are no idle time and jobs which cannot be completed without delaying some operations. The interest lies on generating feasible-active schedules. In order to maintain the feasibility a special routine has been introduced in this algorithm.

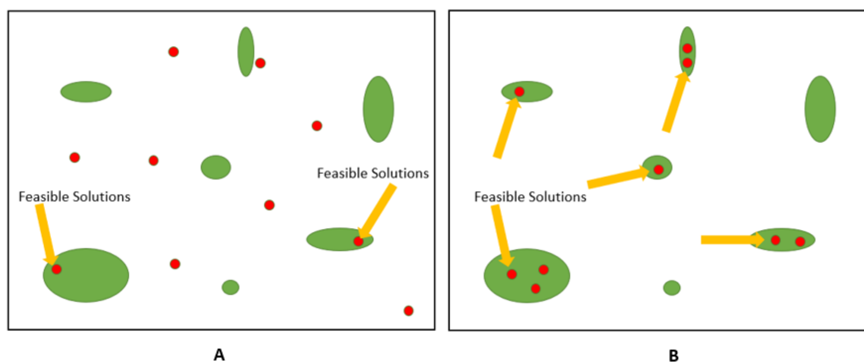

Figure 3.1: Genearted Feasible Solutions

Figure: 3.1 shows generation of solutions in solutions space. In the left figure

it shows random generation of solutions, in that case only few solutions are feasible. When solutions are generated using the controlled mechanism described afterwards, all the solutions are feasible solutions in the solution space.

When generating the schedule, the algorithm iteratively creates Genes. Assigning STU for a Gene is controlled in order to avoid getting infeasible schedules. When generating the STU of an operation, a routine first checks the next available STU related to the job and it's order of operations. The next available STU for the operation is set as follow:

$nextSTU = endtimeunitofthelastoperationofthejobscheduled + 1$

Then the machine availability is checked against the above tentative STU. If the machine is available for the entire duration of the operation, then it returns the tentative STU as the STU of the operation. If the machine is not available throughout the duration of the operation then the end time unit of the event (as there can be already assigned operations or planned machine maintenance or fixed STU operations) will be assigned to the tentative STU and runs the machine availability check until an allocation is found. Due to this process, the assigned STU is not violating the operation order of the job and there will be no machine allocation overlaps.

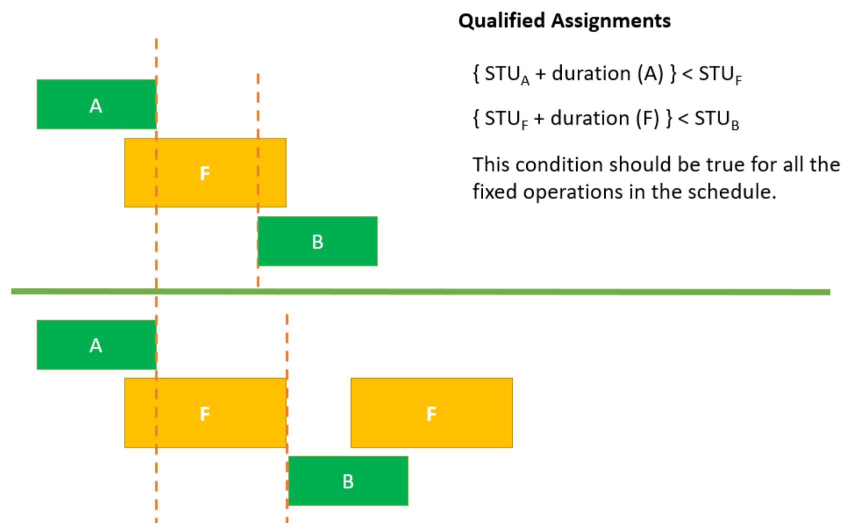

Figure 3.2: Assigning Operations to Machines

In Figure: 3.2 , it shows qualified operation assignments on machines. If operations of job A and Bare already assigned to a machine as in the figure, it is not possible to assign another job, F to the machine.

Following relationship should always be true. Otherwise it is not possible to add an operation of a job to a machine. Where STU_x is the Starting Time Unit (STU) of operation x.

$$STU_A + duration(A) < STU_F$$

AND

$$STU_F + duration(F) < STU_B$$

1. **Random**: A job is randomly picked and get the next operation to be scheduled as the next operation to assigned to the Gene of the Chromosome. This the method used more frequently when generating initial solutions, as it introduces plenty of variations compared to other methods.
2. **Weighted Shortest Processing Time**: Duration of the operation is being considered with the priority.
3. **Weighted Eeriest Due Date**: Operations of the job which needs to be completed earlier will be considered with the job's priority.

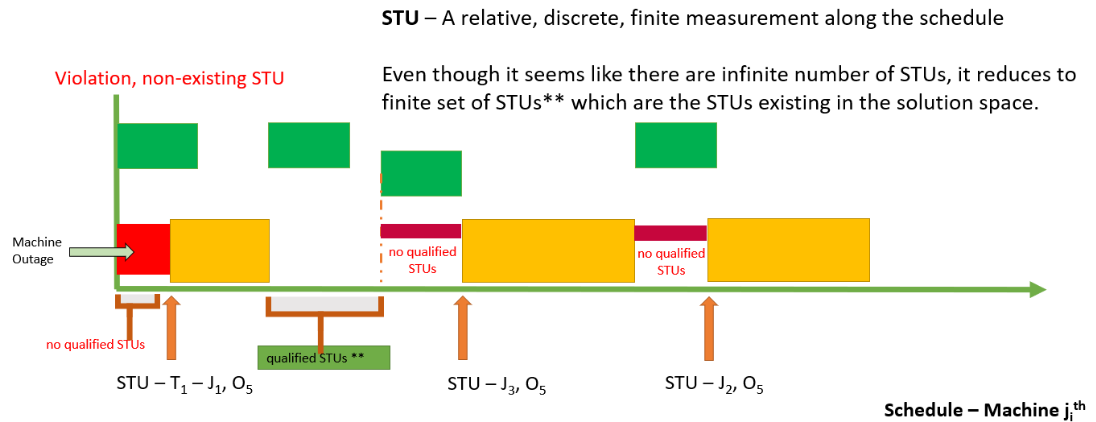

Figure 3.3: Continuity of STU on the timeline

Dis-Continuity of Starting Time Unit (STU) Even though it seems like STU a random, continuous value, it is not. In a given operation (machine), STU cannot take all the continues values along the time line. Some values are not part of the specific solutions. Hence, the ultimate purpose is to find out possible values for STU per given schedule. If an infeasible value is generated for a STU of an operations, it automatically adjusted to reflect the qualified STU for the solution. The concept is illustrated in Figure: 3.3

3.3.3 Fitness Calculation

Once the initial population is ready, the next step is to rank them according to their fitness values. Members of the population are ranked based on the fitness value. The fitness function is directly related to the objectives which are attempted to be optimized. For an example, if the objective is to minimize the total makespan for all the jobs, then the members of the Chromosomes are ranked where the schedules having lowest makespan is ranked as higher as it has the largest fitness among other member Chromosomes. However, not every schedule performs in the same manner and depends on the factors which are trying to be optimized.

Therefore, objectives are followed by different schedules to achieve their relevant objective. A member of the population can be ranked higher for a fitness function where the objective is to minimize the total makespan while the same Chromosome can be ranked lower when using a fitness function with different objective.

Following are the optimization approaches considered in this study.

Minimum Makespan Makespan is the time required to complete a job, in this approach it is planning to minimize the time taken to complete all the jobs. Therefore, the time required to complete each job is sum up and then taken the reciprocal of the sum. That ensures the higher fitness for members having smaller total makespan. In the equation, j_n denotes the total makespan for n^{th} job.

$$fitness = \frac{1}{\sum_{n=1}^{n} j_n}$$

Weighted Minimum Makespan Meeting the set completions dates per job is important, meeting the due dates for higher prioritize jobs are crucial. In such situation, if all jobs aren't possible to schedule to meet due dates of the all the jobs, then it should focus on scheduling jobs which are crucial. In such instance this method can be used as it tries to schedule the operations considering the completion of highest prioritized jobs first.p is the priority of the job.

$$fitness = \frac{1}{\sum_{n=1}^{n} p \times j_n}$$

Weighted Minimum Tardiness In some situations, it is not possible to schedule all the jobs within the given deadline. In such situations, it is required to complete the critical jobs within the deadline and then try to reduce the other jobs overshoots

(possible overrun after the deadline). D_n is the deadline for the nth job which is a fixed amount. c_n is the time required to complete the nth job.

$$fitness = \frac{1}{\sum_{n=1}^{n} D_n - c_n}$$

3.3.4 Crossover

Once the ranking has been carried out for the population the next step in the GA is the reproduction of new members. The members of the population is ranked according to the fitness and the mating is being carried out according to their position in the ranking. The idea is to give a priority for the members having a higher fitness value (higher rank). The objective is to propagate and improve the generations.

There are two steps in mating, one is selecting members out of the population. The next step is performing the crossover and generate a new member. The real meaning of the crossover operation is to introduce values in the other solutions to the new solution, so that the suitability can be checked with the fitness function.

Selecting Parents In this approach, it is required to select high ranked members more often and it helps passing of nearly optimized STU in those members to the new generations. Tournament Selection (TS) method and Roulette Wheel Selection (RWS) will be carried out to select parents as those methods select parents in a biased-random manner.

Roulette Wheel Selection (RWS)

1. Select Regions of ranked population
2. Assign priorities to regions
3. Select a region with bias probability
4. Select members from the selected region randomly

Tournament Selection (TS)

1. Select n members randomly and form k teams.
2. Select the weakest and the strongest members from a team.
3. Based on the fitness values of those two members execute a bias random selection.
4. Eliminate the member based on the biased probability value from the selected two members.

5. Repeat this till m members are selected from each team.
6. Repeat the same for other teams
7. Form the new population from those selected members.

Crossover Process Once parent members have been selected, crossover is being carried out. Crossover rate governs the rate of applying the crossover operation. Crossover rate decides if it is required to perform the crossover. One member is treated as the weak member and the other is considered as the strong member. The weak member's Chromosome is the base for the new member. If it decides to perform the crossover, then the STU of the corresponding operation of the weak member is replaced by STU of the strong member. This is illustrated in Figure: 3.4 . This process is being carried out for each and every Gene in the Chromosome to produce the new member in the new generation. This method of crossover is a slight variation of which is known in the literature as Position Based Crossover (PBX) [50].

Again, the generated new member should be a feasible schedule as in the initial population. There are two approaches, one is to check the value of the STU before assigning the value to the gene, as it was carried out when generating the initial population. The other approach is to adjust the schedule once the member is generated.

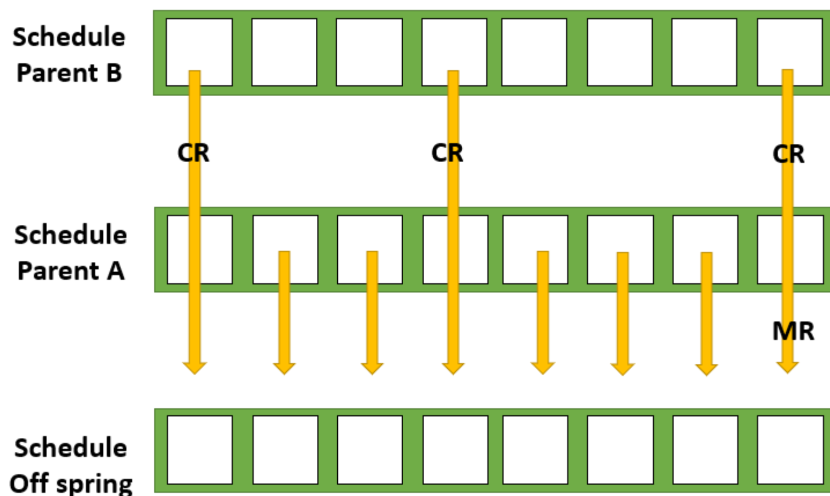

Figure 3.4: Crossover and Mutation process

The first approach is used in this instance (reason for this selection is described under the mutation operation), check the value before assigning it to the Gene. The same algorithm noted in the generation of the initial population is used here as well.

3.3.5 Mutation

In the crossover operation values exist in other solutions are introduced. But in the mutation operation new values can be introduced to the solutions. That is the main purpose of the mutation operation. Introducing new values help the GA to explore the other areas of the solution space and also get rid from the local optimal.

In this algorithm mutation rate governs the rate of mutation per Gene in the Chromosome after the crossover. The value, STU is generated randomly and then the value is checked against the feasibility of the schedule. If the value contributes to not feasible schedule, then the values is altered, as in the crossover operation using the same algorithm.

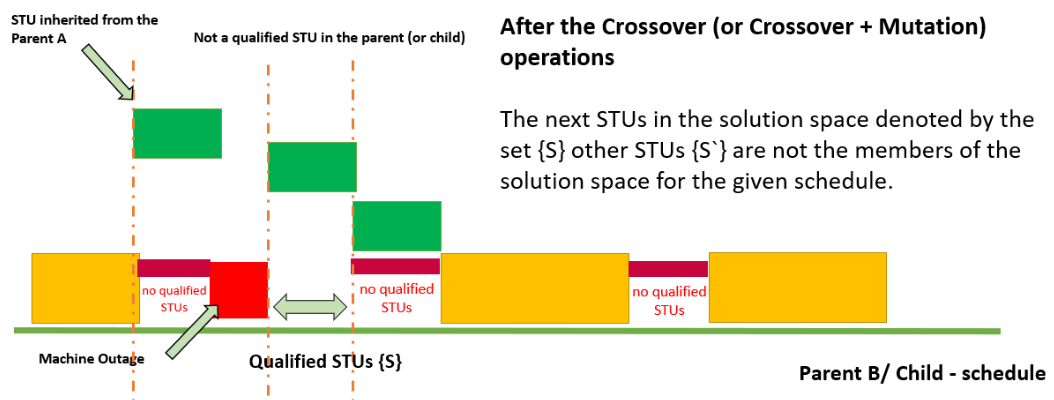

Figure 3.5: Crossover and Mutation

As described earlier, the values for STU represent the solutions in the solution space. As shown in the Figure: 3.5 , even though the value of STU is inherited by a parent, that value cannot be directly used in the schedule. Remember that the STU is encoded and the value is related to the schedule. In some instances, the inherited value cannot be directly used because that value is not in the related solution space. In that case the related value for the STU should be calculated for the right schedule.

3.3.6 Natural Selection

The created new members are known as the off-spring. Before absorbing the new members to the population, their fitness should be evaluated. This is carried out by the same function discussed in the earlier section, calculating the fitness values. Once the members of the offspring is evaluated against the fitness function, these members need to be positioned according to the fitness values.

The new generation should include not only new members but also the members of the previous generation as well. The strategy used in this algorithm is to rank the members of the previous generation and the members of the offspring and then select out of those and create the new population.

To select the members for the new generation following methods has been employed. As same in the methods used in parent selection, TS and RWS methods have been used for selecting new members for the new generation. The purpose is to introduce the biased randomness over the selection.

3.4 Optimization

As noted Genetic Algorithms (GA) alone does not suffice and does not yield good results. GA is good to explore the solution space though it is not good at finding the local optimum, at least with an effective speed. Therefore, a separate algorithm is required to get this work done.

The main objective is to optimize the generated solutions further. That algorithm optimizes each Chromosome, once the crossover and the mutation process are completed before calculating the fitness value.

The local search algorithm employed in this model, eliminated machine idle time to optimize the generated solution. The local search algorithm is not allowed to make drastic changes other than adjusting the STU. Otherwise it might destroy the values derived by the GA. More sophisticated searching algorithms, such as Tabu Search could have been applied if new local solutions are expected.

In the local search used in this study, optimizes the schedule by removing idle time in between operations (it considers machine outages and fixed time operations as well).

3.5 Dynamic JSSP

The previous section describes the development of the algorithm and the optimization techniques used. In this section it is described how to use the developed algorithm and techniques to manage the dynamic behavior of the JSSP. The Dynamic JSSP is possible to convert to a Static JSSP and solve once a disruption occurs. However, the converted problem is not efficient if a GA is not specially design to handle the problem.

This algorithm has added-features to deal with the historical operations which was in the initial schedule. When a disruption occurs, the new problem is to be generated along with the previous information on operations which are partially or completely

executed. Such information is used by the algorithm to produce feasible effective schedules.

Time based schedule Generation As described in the Chromosome representation section, the small building block of the Chromosome is the Gene. Gene contains information on the starting time of the operation compared to conventional algorithms based on GA. That helps to define starting time points of the operations along the schedule. Hence the start of any operation is not relative, it is abstract. It is easy to define the exact time the operation has executed in the history (for operation which has already started). Other than the operations, it helps defining future and ongoing machine break down on the schedule as well. Therefore, the algorithms refrains assigning operation on machines when there are outages on specific machines. Further disregarding whether an operation has taken place or not, it is possible to define the starting point of the future operation along the schedule.

Completion Factor Introduction of the operation starting point, Starting Time Unit alone does not provide any advantage discussed in the last paragraph. The completion factor to be associated as well. If an operation is nearly completed, as mentioned earlier the starting time of the operation can be fixed. However, the Algorithm doesn't have information to identify if the operation is already taken place or not. That is why the completion factor is required. It holds information on the degree of completion of the operation. If an operation is already completed it is 100% and if it is partially completed, the amount of completed amount can be noted. So that when scheduling the Algorithm understand and calculated the remaining duration.

For an example if an operation says it is 20% completed and the total duration of the operation is 5 time units, the algorithm allocates the resource to complete the specific operation in 4 days. As 1 day of work has already being carried out on that operation.

How these features and concepts individually useful in given disruptions modes are discussed extensively in coming sections.

3.6 Disruptions

3.6.1 Arrival of new jobs

When new jobs arrive, the ongoing schedule could no longer be valid. A new schedule has to be generated including the new operations. The new jobs have levels

of priority and due dates to meet. Furthermore, in some situations the start time of the first operation of the new job needs to be fixed. Because it is not possible to start a job once it is received to the manufacturing facility, as discussed earlier. May be the material required to carry out an operation takes time to receive from a supplier even though the information of the job is received to the facility beforehand.

In such a situation it should be possible to generate a feasible schedule for the future jobs. That helps the facility to get prepared to accept the new job and alter the schedule effectively without waiting till the new job's actual starting date. It is possible to simulate and check the best solution and decide whether it is beneficial to wait till the new jobs start date (where materials are expected to be received) or if it is beneficial to terminate the ongoing schedule beforehand.

The Algorithm can handle new job arrivals. New information of the new jobs has to be provided and it is required to alter the schedule to reflect operation that has already been partially or completely executed. Partially completed operation should be indicated by the amount of competition percentage and the time the operation has actually started.

The Algorithm ignores the operation which are completed. Operations which are completed partially are taken as fixed operations and the duration is calculated based on the completed percentage of the operation.

Furthermore, it is possible to fix the start time of the first operation of the new job or any other operation as required. The Algorithm ignore the fixed operations and start scheduling other operations to meet the given objective.

3.6.2 Unplanned machine breakdown

The next disruption is identified as machine breakdown in a middle of an ongoing schedule. When a machine fails without a warning there is no option other than delaying the currently processing operation till the machine get fixed. In such a situation, a new schedule is required to generate.

Information on duration of the machine unavailability (time required to fix the machine) and the current progress of the operations need to be provided. The algorithm ignores already completed operations and fixed operations if there is any. Then currently processing operation is scheduled first before start planning other operations. As it is possible to fix future operations, it is possible to use this algorithm and instruct the algorithm to fix adjacent operation after the currently processing operation. Because there can be instances like the material required to process the next operation is already arrived at the facility before the machine breaks down.

As this suddenly get out when it is fixed it is required to run the at least the operation where there are materials as planned. In such situations this Algorithm can be effectively used as it is possible to fix operations.

3.6.3 Unplanned machine maintenance

If there is any machine maintenance required and if those information is available at the time of Generating the schedule it is possible to provide those information to the Algorithm. But there can be situations such requirements occur suddenly as the schedule progresses. These incidents are not critical as unplanned machine breakdown. The machine needs a maintenance before the next operation can be executed.

In such a situation if the time required to carry out the maintenance is not enough as the next operation is schedule with a lesser gap, the schedule is required to be re-evaluated with the newly discovered information. The only difference at this situation compared to unplanned machine breakdown is, that there is no ongoing operations in the particular machine. However, information on the duration of the maintenance of the machine is required to provide. Apart from that if any operation needs to be fixed such information can be provided as well.

Chapter 4

Implementation

The basic design of the algorithm is discussed in the previous chapter. In this chapter, the implementation of the designed algorithm is discussed.

4.1 Implementation Approach

The main algorithm consists of two sub algorithms. To perform the global search Genetic Algorithms (GA) is employed. To optimize the local search, another algorithm is used. It is identified as Local Search Optimization Algorithm (LSOA). Together these algorithms perform the scheduling and the systems is termed as Scheduling Engine (SE).

Inputs to the system should be feed in text file format. The output is then written to a text file. Both are in Comma-Separated Values (CSV) file type.

Python programming language (version 3.0) has been used to implement the algorithm as it is a widely used and popular scripting language among the scientific community. It is not as fast as other well known languages such as C or C++, but in contrast it is easy to use. As a result Python has been chosen to implement the algorithm. A visualizer has also been developed to visualize the output which is saved as a Comma-Separated Values (CSV) file. It is developed using PHP programming language. The visualizer helps understanding the generated schedule graphically.

The SE acts as the server which receives inputs and then carry out the scheduling operation to solve the given problem. Then the output file can be fed to the visualizer which generates an easy to understand graphic image of the output. Due to selected two languages it is fairly easy to implement this program as a client – server architecture program which can be used over the World Wide Web. SE needs high performance computer as the algorithm uses more resources as the complexity increases. Due to

the client-server architecture, it is possible to implement SE in a high performance machine and access it using workstations/ personal computers.

4.2 Standards in the Implementation

The chromosome representation is one of the vital features in GA as it relates to the phenotype and genotype representations as well. Genotype contains encoded data of the schedule while Phenotype presents the actual solution to the problem which is the schedule. The building block of the chromosome is the Gene and it is decided to use indirect representation with an encoding using the start time of the operation, termed as Starting Time Unit (STU).

Time Unit – The time unit used in the algorithm is a non-abstract continuous value. It is a relative measurement of the time, it can be hours, days or even weeks as it suits the problem. The unit needs to be defined as required. The STU concept is introduced as the starting time unit of an operation. Together with the duration information, the end time of the operation can be calculated.

4.3 Inputs to the Algorithm

Input to the Scheduling Engine (SE) should consists following information. For the Static Job Shop Scheduling Problem (JSSP) item no. 1 and 2 would suffice. For the Dynamic JSSP other items are required.

1. Operation sequence per job
2. Duration per operations
3. Information on machine unavailability
4. Historical information on the current schedule
5. Information on constraints

4.3.1 Operation Sequence per Job

Each job should have to have sequence of noninterchangeable set of operations. The operations should be followed in order, sequentially. Number of operations can be different from job to job. In the input file, operations are denoted by numbers. As per the assumptions laid out in the previous sections. There is a one to one relationship in operations to machines. Because only one operation can be carried out in a given machine in a given time.

4.3.2 Duration per operations

The operations described in the previous sections has a pre-determined duration. That is a mandatory value per operation which should be included in the input file as well.

4.3.3 Information on machine unavailability

This is not a mandatory information to solve the problem and can be skipped. However this is a mandatory information if the algorithm should consider planned machine maintenance or machine un-availability or fixed STU operations constraints. This is handled as an array with a pre-defined format as follow.

Index 1	Index 2	Index 3	Index 4	Index 5

Figure 4.1: Format of the Events

- **Index 1** – Type of the event (noted in Table 4.1)
- **Index 2** – Job number, 0 for machine outages
- **Index 3** – Percentage of completion of the operation or the duration of an event
- **Index 4** – Starting Time Unit of the operation or the event.

The input of a machine outage has been illustrated in the first figure of Figure: 4.2 . It translates as, machine breakdown at time unit 300 for machine 3. The duration is 100 and as this is a machine outage job should be 0.

A

50	0	3	100	300

B

70	5	4	75	250

Figure 4.2: Examples for defined Events

The second figure in Figure: 4.2 , illustrates an input of a partially completed operation. It is the operation no. 4 of the job number 5. Type has been set as 70 as

Table 4.1: Type codes

Code Type	Related Events
50	Future machine outage (Job no and Completion perc. not required)
60	Historical information, 100% completed operations
70	Partially Completed Operations
80	Operations with fixed starting time

it is the corresponding type code for partially completed operations. 75% has been completed so far and it needs to be started at 250th time unit in the new schedule.

4.3.4 Historical Information on the Current Schedule

It is required to provide historical information as part of the problem when a disruption occurs. Historical information includes, completion amounts of the operations which are being executed at the moment of the disruptions. The amount of completion is given in percentage (%) and the STU of the operation should be provided. It is not required to provide the STU of the operation, if the operation is fully processed, then completion amount is 100%.

4.3.5 Information on Constraints

To denote operations with Fixed Starting Units, then this feature can be used. There are instance where the starting time an operation can be pre-defined. For example if the raw-materials required for an operation might be arriving late to the facility.

In the algorithm, if it encounters a fixed starting time unit, then it checks the completion amount. If it is 0% then the algorithm identifies it as a new job and total duration of the operation is used when planning.

In case the completion amount is greater than 0% but lesser than 100%, the algorithm considers the operation as a partially executed operation with a fixed time unit. The remaining duration for the operation is derived by calculating the original duration against the remaining completion time.

- Original duration = $\mathrm{t_o}$
- Amount of Completion = C
- Remaining duration = tr

$$t_r = t_o \times \frac{(1-C)}{100}$$

4.3.6 Graphically Visualization the Schedule

The output of the Scheduling Engine (SE) is a Comma-Separated Values (CSV) file which is hard to understand by humans. Therefore the visualizer generates an graphical image to visualize the final schedule which is easy to understand.

Chapter 5

Results and Analysis

In this chapter, obtained test results are presented in the beginning. Then the results are analysed.

Unlike the Dynamic JSSP, Static JSSP is well investigated optimization problem in the literature. As a result, the proposed algorithm is tested against static JSSP benchmark problems[51] in the literature. The performance of the new algorithm can be assessed on it's capability of solving basic problem. Another reason to test the proposed algorithm on the static JSSP is, it is not possible to find benchmark problems in the same settings as we defined the problem in the previous chapter. If the algorithm solves the static JSSP benchmark problems then it can be fairly assumed that the conceptual design of the novel approach is a feasible approach to tackle JSSP problems. Therefore the first section is devoted to test the algorithm in general in Static JSSP context and evaluate the performance of the algorithm.

5.1 Tests for Solving Static JSSP

The idea here is to analyse the performance of the algorithm against the selected parameters in the proposed algorithm. GA generates stochastic solutions. Therefore the final output could depend on the selections of the intermediate steps. Following are important steps in GA.

1. Generation of the initial population
2. Ranking of the fitness of the individual member
3. Crossover and Mutation genetic operations
4. Generation of the Off-Spring

It is important to identify the behavior of the each step against the techniques use in the algorithm. Therefore initially few common techniques are selected and tested.

Out of above mentioned four steps, step 1 has been ignored. The used technique is random generation of the initial population. The reason for the decision is to include random sets of solution members initially, covering the whole solution space. Otherwise there is a chance of getting biased solutions which then caused stagnation of local minima.

Calculation of the fitness of the generated members depends on the problem itself. It is a function of the variables that it is trying to be optimized and termed as the object function. We have considered three approaches in deriving the object functions which are mentioned in the chapter 4 under the section **Fitness Calculation**.

1. **Minimization of Total Make Span** - Trying to optimize the time required to complete all jobs.
2. **Minimization of Total Tardiness** - Trying to optimize the schedule by reducing the delays of the jobs.
3. **Minimization of the Make Span of weighted jobs** - Trying to minimize the total time required by prioritizing the important jobs.

Then the other vital steps, Crossover and Mutation genetic operations. These are the most important steps in Genetic Algorithms. Selection of the Crossover rate and the Mutation rate is essential and then the method to achieve the operations need to be designed. Early testings of the algorithm experienced trapping the solutions in local minima, therefore it is decided to use variable crossover and mutation rates. As a result, the algorithm starts with low Crossover and Mutation rates. But when a local minima is hit, the algorithm automatically increases the rates so that it can escape the local minima. Probability of having a Mutation based on the probability of having a Crossover. The rate of Mutation is controlled unless otherwise it can lead to an uncontrolled random search.

As the rates are automatically adjusting, when a local minima is hit, the algorithm tries increasing mainly the mutation rate to introduce new members with higher fitness. When the algorithm fails to locate an improved fitness members after some trials, the algorithm terminates.

The pattern of the Crossover and Mutation rates are different from execution to execution as the rate is automatically adjusted based on the situation in the run-time. However the common trend is increase of the rate as number of population grows.

In Figure: 5.1 , the moving Crossover and Mutation rates against the problem LA19 is illustrated. Note that these values are specific to an instance but the profile remains same.

As the number of generation increases the solution tent to converge to near optimal solutions. Hence the rates should be higher to escape current local optimal situations and find new solutions in the solution space. That is why the rate is continue to increase as the populations grow. The initial generation is generated randomly and out of those members new members are evolved with higher fitness. Initial optimization happens in the initial generations (till around first 50 - 100 generations) and then generations trap in a local minima. That is why there is a peak in the rates.

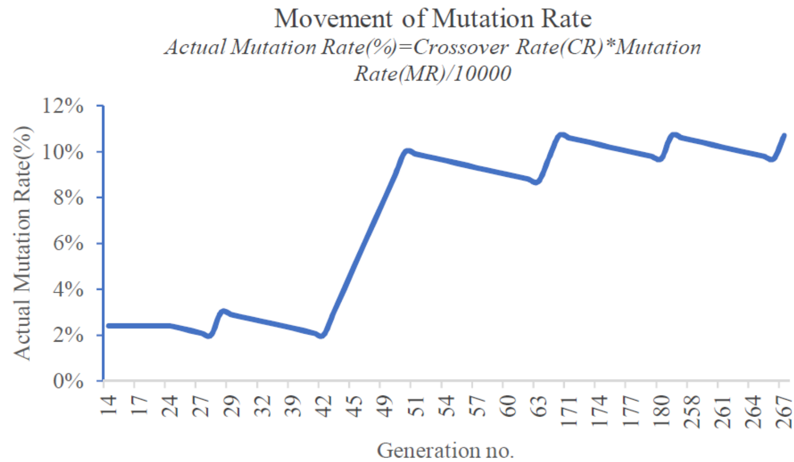

Figure 5.1: Movement of Mutation Rate

Actual rate for Crossover and Mutation is calculated using Critical Ratio (CR) and Mutation Rate (MR) as follow:

$$ActualMutationRate = CR \times \frac{MR}{100}$$

5.1.1 Parent and Next Generation Selection

Out of many available techniques to select parents and next generation (in GA terms, off-spring), three common approaches have been used in this study to test the algorithm. The algorithm is tested with combination of these techniques to get an idea how the algorithm works based on the techniques selected.

1. Tournament Selection (TS)
2. Roulette Wheel Selection (RWS)
3. Random Selection (RS)

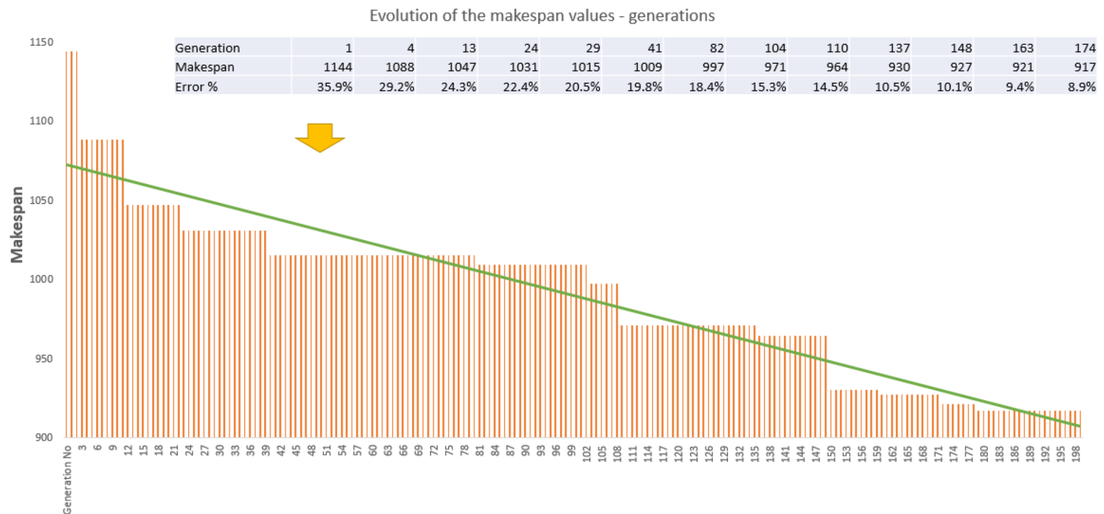

Figure 5.2: Convergence against the number of generations

Five Static JSSP benchmark problems have been selected from the literature as mentioned in the beginning of this chapter. They are in same size which is 10 jobs and 10 machines, knows as 10×10 problems. For each problem 20 schedules have been generated and the best schedule is selected.

Table 5.1, table 5.2 and table 5.3 present the obtained results. The BKS column denotes the Best-Known Solution for the given problem. The makespan column denotes the best values achieved by the proposed algorithm , out of 20 generated solutions. The first % column indicate the percentage optimally of the achieved solution against the BKS. Then the mean of the generated 20 schedules and the variance of those schedules are mentioned.

According to the test results, the proposed algorithm has been able to generate optimal solutions with around 85 - 90%. This is a promising result as the algorithm does not include a strong local search algorithm. The novel concept of STU, inclusion of data of time component seems to be a feasibility too.

Based on the results, the parent selection method is set to Random Selection (RS) and next generation method to TS.

Table 5.1: Parent Selection Method – TS and Next Generation Selection Method – TS for problems of 10 x 10 (Jobs x Machines)

Problem	**BKS**	**Makespan**	%	μ	%	σ
LA16	945	1079	87.58	1116.70	84.62	34.21
LA17	784	903	86.82	957.80	81.85	28.26
LA19	842	1016	82.87	1056.20	79.72	32.00
LA20	902	1048	86.07	1089.30	82.81	32.90
FT10	930	1180	78.81	1241.10	74.93	28.97

Table 5.2: Parent Selection Method – Random and Next Generation Selection Method – TS for problems of 10 x 10 (Jobs x Machines)

Problem	**BKS**	**Makespan**	%	μ	%	σ
LA16	945	1066	88.65	1087.80	86.87	16.92
LA17	784	837	93.67	884.30	88.66	21.53
LA19	842	917	91.82	959.80	87.73	18.81
LA20	902	991	91.02	1030.80	87.50	24.05
FT10	930	1056	88.07	1123.10	82.81	33.19

Table 5.3: Parent Selection Method – RWS and Next Generation Selection Method – RWS for problems of 10 x 10 (Jobs x Machines)

Problem	**BKS**	**Makespan**	%	μ	%	σ
LA16	945	1052	89.83	1076	87.83	19.71
LA17	784	890	88.09	903	86.82	16.99
LA19	842	928	90.73	953.1	88.34	19.47
LA20	902	996	90.56	1035.75	87.09	36.83
FT10	930	1077	86.35	1122.10	82.88	22.72

5.2 Dynamic JSSP

In the last section, the performance of the GA has been tested over the static JSSP. In this section, the algorithm should be tested against the problem defined. Because it is the ultimate goal of the algorithm and solutions should be generated satisfying the conditions outlined in the problem definition section.

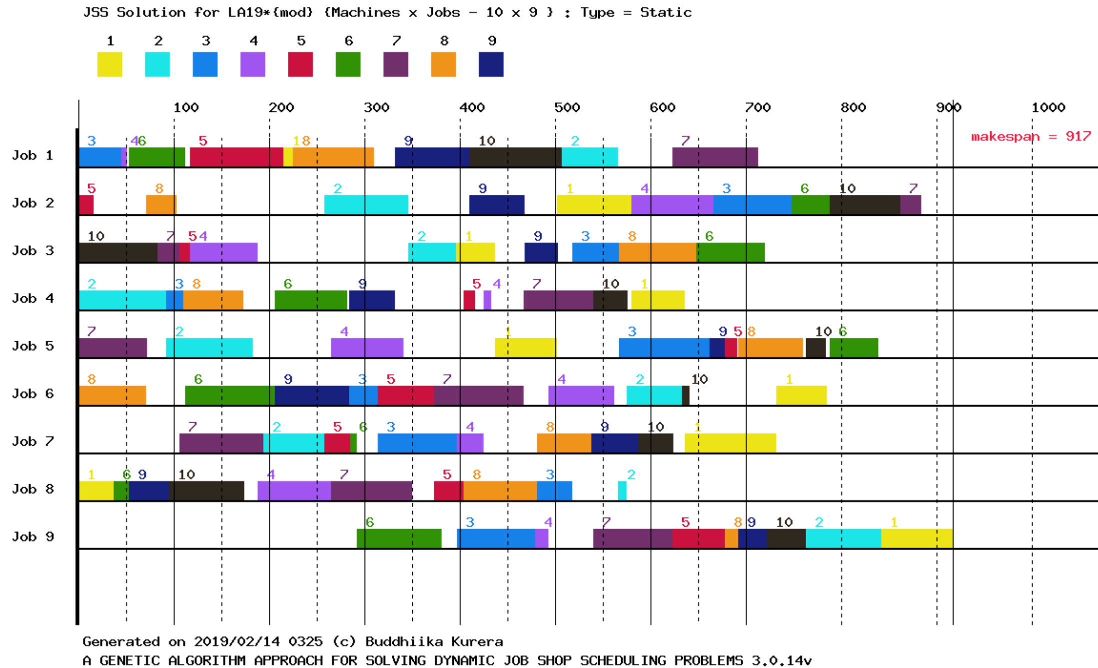

Figure 5.3: Generated Schedule for modified LA19 problem ($LA19^A$)

Since there is no benchmark problems, a new problem is developed based on the LA19 Static JSSP to simulate the requirements. Disruptions are simulated on the above mentioned problem and the algorithm is tested.

5.2.1 Scenario A - Arrival of new jobs

The algorithm should be capable of handling a disruption occurs due to arrival of a new job.

The base problem, L19, has been modified removing the 10th job in it. An initial schedule has been generated based on LA19 problem but with no 10th job. This problem is denoted as $LA19^A$. The generated schedule for this problem is shown in Figure: 5.3 . Assume the new job arrives at the time unit 200. At that time, the planner needs to generate a new schedule including the new job, which is the 10th job in original LA19 problem.

Table 5.4: Fully Completed Operations per Job, $LA19^A$

Job	Completed Operations
1	3, 4, 6
2	5, 8
3	10, 7, 5, 4
4	2, 3, 8
5	7, 2
6	8
7	7
8	1, 6, 9, 10
9	-

Table 5.5: Partially Completed Operations per Job, $LA19^A$

Job	Operation	Duration	Completed %	Remaining Duration
1	5	97	93	14
6	6	93	97	5
7	2	63	77	54
8	4	76	75	20

The first thing to initiate new schedule generation is taking the historical information of the earlier schedule. The completed operations and partially completed operations per job needs to be identified and should be provided as an input. Completed operations per job listed in the table 5.4 where the partially completed operations per job with their degree of completion amounts are mentioned in table 5.5.

Handling the partially completed operations on machines is vital once the new schedule resumes. In the new schedule, these operations should be unchanged.

The proposed algorithm generates the new schedule considering mentioned historical information, in this scenario it is partially and fully completed operations. The new schedule is shown in Figure: 5.4 . Note that the completed operations are filtered.

Schedule starts with partially completed operations. The newly created schedule starts with the time unit 200. The schedule is relative, that is why the starting time unit is marked as zero in the schedule. However the abstract values on the should be calculated by adding another 200 time units to the value in the newest schedule.

Accordingly the total makespan for the problem $LA19^A$ is $1077 = 200 + 877$

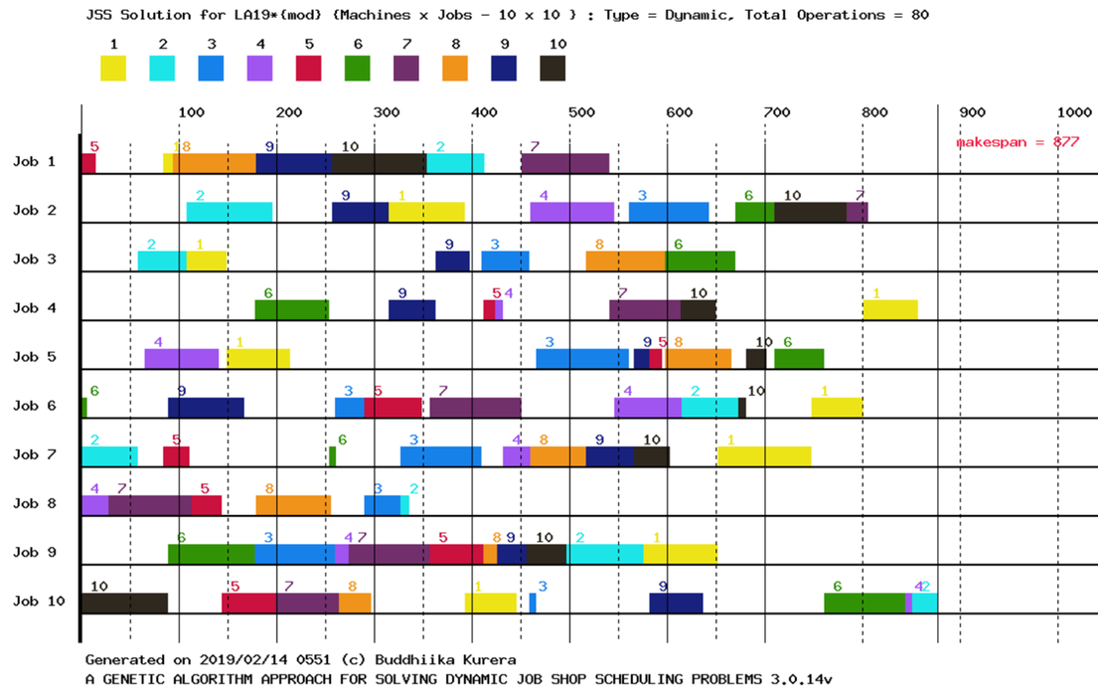

Figure 5.4: After including the new job in $LA19^A$

5.2.2 Scenario B - Machine break-down

When a machine suddenly goes out of order, the ongoing operation in that machine cannot be progressed. The machine needs to be fixed and it will be out of service till the issue is fixed. The current schedule becomes invalid once the machine outage occurs and need a new schedule based on the new information. If the time required for fixing the machine is known, it is possible to generate a new schedule with inclusion of the downtime for the specific machine. However the operation which is being performed at the time of the breakdown in the machine which goes out of order should be allocated to the same machine once it is fixed. The concept of fixed operations (remaining time units) comes into play in this scenario. Further the other operations which are being processed on the other machines can resume the operations at the start of the new schedule, same as what happen when a new job arrives.

5.2.3 Scenario C - Unplanned machine maintenance

Planned machine maintenance is bit different to the scenario discussed in the previous section because it does not demand an immediate schedule change. This scenario occurs when a machine requires a maintenance after generating a schedule. It is assumed that the maintenance can wait till the currently processing operation finishes on the machine. Then after a new schedule can be employed. This scenario can be modeled as the new job arrival scenario which is discussed in detailed.

5.2.4 Scenario D - Start time fixed operations

Disregarding the completed percentage of the operation, it should be able to instruct the starting time of the operation. Operations which are partially completed can be rescheduled with a fixed starting point for the remaining duration. This concept has been used in all the scenarios discussed earlier. In new job arrival scenario, the starting point of partially completed operations at the beginning of the time line of the new schedule. In the machine break-down scenario, it is a future time unit on the time line.

5.3 Analysis

In this section, obtained results are analyzed. Testing has been carried out to evaluate performance of the proposed algorithm and to understand how it behaves under different scenarios and selected methods.

In testing, the effect on crossover rates and mutation rates has been observed by varying those rates on standard benchmark problem, LA19. Then some selected methods used in parents selection and forming new generations (off-spring) have been tested. Five benchmark problems[51] (LA16, LA17, LA19, LA20, FT10) have been used for the purpose. Only problems of 10 jobs and 10 machines, commonly known as 10×10 problems have been used in this study.

Apart from the FT10 problem, other problems are easier to schedule. In FT10 problem, operations order is quite similar in jobs. Therefore, the demand for machine at a given time is higher making scheduling tough leading to higher makespan compared to other problems. That is the reason for selecting FT10 problem to test the algorithm. However the algorithm was able to generate a feasible schedule for FT10 with around 88% optimally.

When executing the algorithm, it keeps generating solution till the exit criteria is met. When Genetic Algorithms (GA) is the only approach in the algorithm, it is quite

challenging to explore the solution space as it converges to local optima in a short duration. Once it is trapped, GA alone is not capable of escaping itself. That is why the GA is very powerful when it is used in combination with another local search algorithm.

In this research, as GA alone used to develop the algorithm, it is set to exit the program after executing pre-determined execution cycles. In other words, the algorithm stops the execution once number of generated generations reach pre-determined threshold. Maximum generation of 400 has been used in testing as it is observed that the algorithm no longer generates new solutions once it passes around 300 generations in earlier testings. It is possible to terminate when there is no change in new off-spring members. However with variable crossover and mutation rates this approach gave unpredictable outcomes. Hence it was decided to terminate after reaching the threshold.

In GA, crossover rate and mutation rate play a major role. Crossover rate decides if a crossover operation should be carried out. In the meantime, if a crossover operation takes place application of the mutation operation is decided by the mutation rate. Crossover is considered as a convergence operation which directs the population towards local optima. On the other hand, mutation is a divergence operation. It helps inclusion of new solutions which eventually helps discovering new local optima in the solution space.

To find out the effects of crossover and mutation operations, several tests have been carried out keeping the mutation rate at a constant value and also varying the mutation rate. This is carried out on LA19 problem and when observing the test results, it seems like high crossover rate and high mutation rate contributes towards achieving better results. in the same time, high mutation rates contribute to random search as well. In order to prevent that effect rate were controlled using variable rates rather than fixed rates. In the beginning the rates are at minimum. When the search hits a local minima the algorithm increases the rate so that it helps escaping the local minima. This is illustrated in Figure: 5.1 .

In the proposed algorithm, for selecting parent set Random Selection (RS) is used. For selecting the next generation Tournament Selection (TS) is used. Tests have been carried out to determine the suitable combination and results suggest the above combination.

Once the algorithm is tested for benchmark static JSSP, the results are used to fine tune the algorithm for solving the dynamic JSSP which is the main objective of this research.

A problem constructed based on LA19 static problem, $LA19^A$ has been used to test the algorithm. As discussed there are three main requirements to be fulfilled by the

algorithm . The algorithm should be able to handle disruptions due to new job arrival, machine outage due to sudden breakages and planned maintenance requirements. Tests have been carried out to check the performance of the algorithm.

Chapter 6

Conclusion

As discussed throughout the report, the main objective is to develop an algorithm to handle Dynamic Job Shop Scheduling Problem (JSSP) as per the requirements outlined in the problem definition section. Genetic Algorithms (GA) is used to implement the algorithm. Tests have been carried out to measure the performances of the proposed algorithm with combinations of different configurations.

6.1 Conclusion

The main objective of this research is to develop an algorithm based on genetic algorithms which can be used to solve Dynamic JSSP. As there is no proper way of benchmarking the performance of the Dynamic JSSP as the constructed requirements in problem definition, the algorithm is tested on Static JSSP. Tests have been carried out on five benchmark Static JSSP in the literature, namely LA16, LA17, LA19, LA20 and FT10 which are 10×10 problems.

Based on the test results, it can be fairly stated that the algorithm is capable of solving the Job Shop Scheduling problem. The generated schedules are nearly optimal, around 85% - 90%. The algorithm did not successfully achieve the Best-Known Solutions (BKS) for the problems.

The algorithm has been developed for solving the Dynamic JSSP . The algorithm focuses on managing the dynamic requirements based on the problem definition. For an example, the introduction of the Starting Time Unit (STU) of an operation to the Chromosome representation can be taken. Therefore, according to the results obtained, it seems that the new representation requires further optimization approaches to reach to the optimal solutions. Genetic Algorithms (GA) alone does not help achieving optimal results.

Further, the time-based representation used in this research is a novel concept to solve the Dynamic JSSP using Genetic Algorithms. The results show that it is a feasible and promising. The proposed algorithm can be used to generate nearly optimal solutions for a given problem. Therefore this algorithm can be used in simulating planning of jobs in the Job Shop. For the time being the proposed solutions cannot replace the planner as it is generating nearly optimal schedules. However this algorithm can be used as a tool to assist the planner's work as it can be used to generate feasible schedules covering all the scenarios which could occur in a Job Shop.

In conclusion, the proposed algorithm performed as expected solving Dynamic JSSP as outlined in the problem definition. It generates nearly optimal schedules as the solutions for the problems which cover most of the real-life scenarios in a Job Shop. The results prove that the inclusion of time encoded value as a novel concept in the Gene is a feasible approach which is a dominant part of this study. For the time being this algorithm can be used as a tool, as an assistive tool for planners in decisions making.

6.2 Future Work

There are few limitations identified when carrying out the testings of the proposed algorithm.

The inclusion of time encoded value in the Gene, termed as Starting Time Unit (STU) can be developed further. Critically observing and analyzing the behavior of the concept could be carried out as a separate study.

Another obstacle identified in testing the algorithm is the convergence of the algorithm and getting trapped in local minima of the solution space. In order to solve this problem, variable Critical Ratio (CR) and Mutation Rate (MR) in run-time have been introduced. The algorithm can be coupled with a local search algorithm such as Tabu Search [52] [53]. According to the literature the combination of GA and Tabu Search work well.

Another approach to diverge the solutions generated by this algorithm is to implement the algorithm as a parallel genetic algorithm [54], where other members of the other parallel solution can be introduced among the solutions using migration operation. Then, optimal schedules can be obtained for dynamic JSSP and real-life requirements that are faced by the industry.

The algorithm in this research is designed to handle JSSP of any number of jobs and machines. However, the tests have been carried out only for 10 jobs and 10 machines

problems. Therefore, the algorithm can be tested for larger problems and measure the performance.

Less focus on performance improvement in time domain was given when implementing the algorithm in the research. The main reason is the use of not high performance programming language. The main objective in this research was to prove that the novel concept, STU is a feasible solution. As a result, easy to use high-level programming language has been used. It compromises the performance level. Therefore, it is possible to search for methods to improve the performances of the algorithm by optimizing the code structure and methods used or re-implement the algorithm using a high performance language such as C++.

Further this algorithm can be extended for solving open floor shop scheduling problems and flexible job shop scheduling problems, the concept of inclusion of time encoding can be tested in those problems.

REFERENCE LIST

[1] T. Gonzalez and S. Sahni, "Flowshop and jobshop schedules: Complexity and approximation," *Operations research*, vol. 26, no. 1, pp. 36–52, 1978.

[2] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. R. Kan, "Optimization and approximation in deterministic sequencing and scheduling: A survey," in *Annals of discrete mathematics*, vol. 5, Elsevier, 1979, pp. 287–326.

[3] I. H. Osman and C. Potts, "Simulated annealing for permutation flow-shop scheduling," *Omega*, vol. 17, no. 6, pp. 551–557, 1989.

[4] T. Gonzalez and S. Sahni, "Open shop scheduling to minimize finish time," *Journal of the ACM (JACM)*, vol. 23, no. 4, pp. 665–679, 1976.

[5] N. Nahavandi, S. Zegordi, and M. Abbasian, "Solving the dynamic job shop scheduling problem using bottleneck and intelligent agents based on genetic algorithm," 2016.

[6] S. M. Johnson, "Optimal two-and three-stage production schedules with setup times included," *Naval research logistics quarterly*, vol. 1, no. 1, pp. 61–68, 1954.

[7] H. G. Campbell, R. A. Dudek, and M. L. Smith, "A heuristic algorithm for the n job, m machine sequencing problem," *Management science*, vol. 16, no. 10, B–630, 1970.

[8] M. R. Garey, D. S. Johnson, and R. Sethi, "The complexity of flowshop and jobshop scheduling," *Mathematics of operations research*, vol. 1, no. 2, pp. 117–129, 1976.

[9] A. Kaban, Z. Othman, and D. Rohmah, "Comparison of dispatching rules in job-shop scheduling problem using simulation: A case study," *International Journal of Simulation Modelling*, vol. 11, no. 3, pp. 129–141, 2012.

[10] E. Hart and K. Sim, "A hyper-heuristic ensemble method for static job-shop scheduling," *Evolutionary computation*, vol. 24, no. 4, pp. 609–635, 2016.

[11] Z. Lomnicki, "A "branch-and-bound" algorithm for the exact solution of the three-machine scheduling problem," *Journal of the Operational Research Society*, vol. 16, no. 1, pp. 89–100, 1965.

[12] J. Adams, E. Balas, and D. Zawack, "The shifting bottleneck procedure for job shop scheduling," *Management Science*, vol. 34, pp. 391–401, Mar. 1988. DOI: 10.1287/mnsc.34.3.391.

[13] R. Zhang, "A simulated annealing-based heuristic algorithm for job shop scheduling to minimize lateness," *International Journal of Advanced Robotic Systems*, vol. 10, no. 4, p. 214, 2013.

[14] E. D. Taillard, "Parallel taboo search techniques for the job shop scheduling problem," *ORSA journal on Computing*, vol. 6, no. 2, pp. 108–117, 1994.

[15] J. B. Chambers and J. W. Barnes, "New tabu search results for the job shop scheduling problem," *The University of Texas, Austin, Technical Report Series ORP96-06, Graduate Program in Operations Research and Industrial Engineering*, 1996.

[16] M. S. Fox, "Constraint-directed search: A case study of job-shop scheduling.," CARNEGIE-MELLON UNIV PITTSBURGH PA ROBOTICS INST, Tech. Rep., 1983.

[17] A. Jones, L. C. Rabelo, and A. T. Sharawi, "Survey of job shop scheduling techniques," *Wiley encyclopedia of electrical and electronics engineering*, 2001.

[18] R. Eberhart and J. Kennedy, "A new optimizer using particle swarm theory," in *MHS'95. Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, Ieee, 1995, pp. 39–43.

[19] T.-L. Lin, S.-J. Horng, T.-W. Kao, Y.-H. Chen, R.-S. Run, R.-J. Chen, J.-L. Lai, and I.-H. Kuo, "An efficient job-shop scheduling algorithm based on particle swarm optimization," *Expert Systems with Applications*, vol. 37, no. 3, pp. 2629–2636, 2010.

[20] B. Giffler and G. L. Thompson, "Algorithms for solving production-scheduling problems," *Operations research*, vol. 8, no. 4, pp. 487–503, 1960.

[21] Z. ZHONG, "Research on job-shop scheduling problem based on improved particle swarm optimization.," *Journal of Theoretical & Applied Information Technology*, vol. 47, no. 2, 2013.

[22] M. Dorigo and G. Di Caro, "Ant colony optimization: A new meta-heuristic," in *Proceedings of the 1999 congress on evolutionary computation-CEC99 (Cat. No. 99TH8406)*, IEEE, vol. 2, 1999, pp. 1470–1477.

[23] M. Dorigo, M. Dorigo, V. Manjezzo, and M. Trubian, "Ant system for job-shop scheduling," *Belgian Journal of Operations Research*, vol. 34, pp. 39–53, 1994.

[24] S. Van der Zwaan and C. Marques, "Ant colony optimisation for job shop scheduling," in *Proceedings of the'99 Workshop on Genetic Algorithms and Artficial Life GAAL'99*, 1999.

[25] C. S. Chong, A. I. Sivakumar, M. Y. H. Low, and K. L. Gay, "A bee colony optimization algorithm to job shop scheduling," in *Proceedings of the 38th conference on Winter simulation*, Winter Simulation Conference, 2006, pp. 1954–1961.

[26] A. S. Jain and S. Meeran, "Deterministic job-shop scheduling: Past, present and future," *European journal of operational research*, vol. 113, no. 2, pp. 390–434, 1999.

[27] T. Yamada and R. Nakano, "Genetic algorithms for job-shop scheduling problems," 1997.

[28] A. Moraglio, H. Ten Eikelder, and R. Tadei, "Genetic local search for job shop scheduling problem," *sottoposto per la pubblicazione a European Journal of Operational Research*, 2005.

[29] B. J. Park, H. R. Choi, and H. S. Kim, "A hybrid genetic algorithm for the job shop scheduling problems," *Computers & industrial engineering*, vol. 45, no. 4, pp. 597–613, 2003.

[30] D. Whitley, S. Rana, and R. B. Heckendorn, "The island model genetic algorithm: On separability, population size and convergence," *Journal of computing and information technology*, vol. 7, no. 1, pp. 33–47, 1999.

[31] J. Chen and B. J. Adams, "Integration of artificial neural networks with conceptual models in rainfall-runoff modeling," *Journal of Hydrology*, vol. 318, no. 1-4, pp. 232–249, 2006.

[32] R. Ramasesh, "Dynamic job shop scheduling: A survey of simulation research," *Omega*, vol. 18, no. 1, pp. 43–57, 1990.

[33] X. Shen, M. Zhang, and J. Fu, "Multi-objective dynamic job shop scheduling: A survey and prospects," *Int J Innov Comput Inf Control*, vol. 10, no. 6, pp. 2113–2126, 2014.

[34] P. Cowling and M. Johansson, "Using real time information for effective dynamic scheduling," *European journal of operational research*, vol. 139, no. 2, pp. 230–244, 2002.

[35] P. Lou, Q. Liu, Z. Zhou, H. Wang, and S. X. Sun, "Multi-agent-based proactive–reactive scheduling for a job shop," *The International Journal of Advanced Manufacturing Technology*, vol. 59, no. 1-4, pp. 311–324, 2012.

[36] V. Jorge Leon, S. David Wu, and R. H. Storer, "Robustness measures and robust scheduling for job shops," *IIE transactions*, vol. 26, no. 5, pp. 32–43, 1994.

[37] G. E. Vieira, J. W. Herrmann, and E. Lin, "Rescheduling manufacturing systems: A framework of strategies, policies, and methods," *Journal of scheduling*, vol. 6, no. 1, pp. 39–62, 2003.

[38] I. Moon and J. Lee, "Genetic algorithm application to the job shop scheduling problem with alternative routings," *Pusan National University*, 2000.

[39] G. Chryssolouris and V. Subramaniam, "Dynamic scheduling of manufacturing job shops using genetic algorithms," *Journal of Intelligent Manufacturing*, vol. 12, no. 3, pp. 281–293, 2001.

[40] L. Zhang, L. Gao, and X. Li, "A hybrid genetic algorithm and tabu search for a multi-objective dynamic job shop scheduling problem," *International Journal of Production Research*, vol. 51, no. 12, pp. 3516–3531, 2013.

[41] A. Madureira and J. Santos, "Proposal of multi-agent based model for dynamic scheduling in manufacturing.," *WSEAS Transactions on Information Science and Applications*, vol. 2, no. 5, pp. 600–605, 2005.

[42] M. Kapanoglu and M. Alikalfa, "Learning if–then priority rules for dynamic job shops using genetic algorithms," *Robotics and Computer-Integrated Manufacturing*, vol. 27, no. 1, pp. 47–55, 2011.

[43] T. Eguchi, F. Oba, and T. Hirai, "A neural network approach to dynamic job shop scheduling," in *Global Production Management*, Springer, 1999, pp. 152–159.

[44] N. Mladenović and P. Hansen, "Variable neighborhood search," *Computers & operations research*, vol. 24, no. 11, pp. 1097–1100, 1997.

[45] M. Adibi, M. Zandieh, and M. Amiri, "Multi-objective scheduling of dynamic job shop using variable neighborhood search," *Expert Systems with Applications*, vol. 37, no. 1, pp. 282–287, 2010.

[46] D. Ouelhadj and S. Petrovic, "A survey of dynamic scheduling in manufacturing systems," *Journal of scheduling*, vol. 12, no. 4, p. 417, 2009.

[47] J. H. Holland, "Genetic algorithms," *Scientific american*, vol. 267, no. 1, pp. 66–73, 1992.

[48] D. Thierens and D. Goldberg, "Convergence models of genetic algorithm selection schemes," in *International Conference on Parallel Problem Solving from Nature*, Springer, 1994, pp. 119–129.

[49] R. Raghavjee and N. Pillay, *A comparative study of genetic algorithms using a direct and indirect representation in solving the south african school timetabling problem*, 2013.

[50] T. Murata and H. Ishibuchi, "Positive and negative combination effects of crossover and mutation operators in sequencing problems," in *Proceedings of IEEE International Conference on Evolutionary Computation*, IEEE, 1996, pp. 170–175.

[51] S. Lawrence, *Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques*. In Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1984.

[52] A. Baykasoğlu, A. İ. Sönmez, *et al.*, "Using multiple objective tabu search and grammars to model and solve multi-objective flexible job shop scheduling problems," *Journal of Intelligent Manufacturing*, vol. 15, no. 6, pp. 777–785, 2004.

[53] M. A. González, C. R. Vela, and R. Varela, "Genetic algorithm combined with tabu search for the job shop scheduling problem with setup times," in *International Work-Conference on the Interplay Between Natural and Artificial Computation*, Springer, 2009, pp. 265–274.

[54] S.-C. Lin, E. D. Goodman, and W. F. Punch, "Investigating parallel genetic algorithms on job shop scheduling problems," in *International Conference on Evolutionary Programming*, Springer, 1997, pp. 383–393.

Appendix A

Program Code

A.1 Implementation of the Proposed Algorithm

This is the main program file, main.py. It uses standard time and standard rnd (for random number generation) libraries in Python. Python NumPy library which is an open source library also used. Other libraries (Tournament Selection, Roulette Wheel Selection) are developed from the scratch to implement the proposed algorithm in Python programming language.

Instructions for use: Locate all three files in the same directory. Change the parameters according to the problem in the ”Problem Specific Basic Setting” section. Execute the code, use Python version 3.0 interpreter.

```
## File : main.py ##
##Algorithm implemented by Buddhika Kurera, (c) 2017##
## Email − bckurera AT geneai dot edu dot com ##

import numpy as np
import gen_lib as gen_lib
import debug_ as DEBUG
import time
import tournament_selection as TSel

DO_DEBUG = 0

import json as json
import sys

##Problem Specific Basic Setting##

M = 10 # max number of Machines
J = 10 # max number of Jobs
MR = 10 # mutation rate in %
CR = 80 # crossover rate in %
DEADLINE = 842 # for LA19, 842 is the smallest
MAX_GEN = 200
INIT_POP_SIZE = 100
JSSP_TYPE = 'STATIC' #'STATIC' or 'DYNAMIC'
GENERATION_COUNT = 1
FITNESS_FUNC = 'M' #T and M

Oij = np.array([
    (2, 7, 10, 9, 8, 3, 1, 5, 4, 6),
    (5, 3, 6, 10, 1, 8, 2, 9, 7, 4),
    (4, 3, 9, 2, 5, 10, 8, 7, 1, 6),
```

```
    (2, 4, 3, 8, 9, 10, 7, 1, 6, 5),
    (3, 1, 6, 7, 8, 2, 5, 10, 4, 9),
    (3, 4, 6, 10, 5, 7, 1, 9, 2, 8),
    (4, 3, 1, 2, 10, 9, 7, 6, 5, 8),
    (2, 1, 4, 5, 7, 10, 9, 6, 3, 8),
    (5, 3, 9, 6, 4, 8, 2, 7, 10, 1),
    (9, 10, 3, 5, 4, 1, 8, 7, 2, 6)

]) # Tasks per job and the operation related to the task [Op <-> Machine]

Dij = np.array([
    (44, 5, 58, 97, 9, 84, 77, 96, 58, 89),
    (15, 31, 87, 57, 77, 85, 81, 39, 73, 21),
    (82, 22, 10, 70, 49, 40, 34, 48, 80, 71),
    (91, 17, 62, 75, 47, 11, 7, 72, 35, 55),
    (71, 90, 75, 64, 94, 15, 12, 67, 20, 50),
    (70, 93, 77, 29, 58, 93, 68, 57, 7, 52),
    (87, 63, 26, 6, 82, 27, 56, 48, 36, 95),
    (36, 15, 41, 78, 76, 84, 30, 76, 36, 8),
    (88, 81, 13, 82, 54, 13, 29, 40, 78, 75),
    (88, 54, 64, 32, 52, 6, 54, 82, 6, 26)
]) # Duration for individual operations

Events = np.array([
    (100,50,6,100,0)
]) # Events[Seq, Type, Machine, Duration, STU]

OP_info = np.array([
    (50, 0, 6, 100,0),
    (80, 1, 8, 82, 0),
    (80, 2, 1, 75, 0),
    (80, 3, 2, 51, 0),
    (70, 6, 9, 77, 100),
    (70, 4, 7, 72, 0)
])

PRIOTIRY = ([1,1,1,2,2,2,3,3,3,4])

##End of Basic Settings##

def generate_a_chromo(): ##Generate a chromosome using random job-op-machine allocation method
    ##DEFINE##
    rem_ops_count_per_job = get_ops_count_per_job()
    op_count_per_job = get_ops_count_per_job()
    chromo = [0]*(sum(rem_ops_count_per_job)*6)
    clear_vectors()

    ##EXECUTE##
    process = True
    while process:
        random_job_index = gen_lib.get_random_number(0, J)
        op_index = rem_ops_count_per_job[random_job_index]
        if(op_index!=0):
            rem_ops_count_per_job[random_job_index] = op_index - 1
            order_of_op_in_job = op_count_per_job[random_job_index] - op_index
            gene = generate_a_gene(op_count_per_job, random_job_index, order_of_op_in_job)

            gene_seq = gene[0] * 6

            for j in range(0, 6):
                chromo[gene_seq + j] = gene[j]

        ##exit criteria##
        if(sum(rem_ops_count_per_job)==0):
            DEBUG.__print__(chromo, 0, DO_DEBUG)
            break
    DEBUG.__save_to_file__(chromo, 'chromosome.dat', DO_DEBUG)

    overlap_detect(np.array(chromo))
```

```
    x = fvc_chromo(chromo)
    DEBUG.__save_to_file__(chromo, 'chromosome'+str(x)+'.dat', DO_DEBUG)

    return chromo
##-----end of method generate_a_chromo()-----##

JOB_ORDER_VECTOR = []
MACHINE_ALLOC_VECTOR = []

def get_next_stu(job, op, tentative_stu, duration): #Get the next STU considering the JOB ORDER VECTOR and the
    ↪ MACHINE ALLOC VECTOR
    get_next_stu_by_job_order_vector = JOB_ORDER_VECTOR[job-1]

    if(tentative_stu < get_next_stu_by_job_order_vector):
        tentative_stu = get_next_stu_by_job_order_vector

    tentative_stu = get_next_stu_on_machine(op, tentative_stu, duration)

    return tentative_stu

def get_next_stu_on_machine(machine, tentative_stu, duration):
    no_of_allocations = int(len(MACHINE_ALLOC_VECTOR[machine-1])/5)
    if(no_of_allocations==0):
        return tentative_stu

    fixed_allocations = MACHINE_ALLOC_VECTOR[machine-1]

    check_overlaps = True
    while check_overlaps:
        proposed_stu = [0]
        no_of_overlaps = 0
        for i in range(0, no_of_allocations):
            alloc = fixed_allocations[5*i: 5*i+5]
            fixed_start = alloc[3]
            fixed_end = alloc[4]
            legal_alloc = check_legal_machine_alloc(tentative_stu, tentative_stu + duration, fixed_start, fixed_end)
            if(legal_alloc==False):
                proposed_stu.append(fixed_end + 1)
                no_of_overlaps = no_of_overlaps + 1

        if(no_of_overlaps>0):
            tentative_stu = max(proposed_stu)

        if(no_of_overlaps==0):
            check_overlaps = False

    return tentative_stu

def check_legal_machine_alloc(temp_start, temp_end, fixed_start, fixed_end):
    if((temp_start < fixed_start) and (temp_end < fixed_start)):
        return True

    if((temp_start > fixed_end) and (temp_end > fixed_end)):
        return True

    return False

def update_machine_alloc_vector(job, machine, type, duration, stu):
    global MACHINE_ALLOC_VECTOR
    temp = [0]*5
    temp[0] = job
    temp[1] = type
    temp[2] = duration
    temp[3] = stu
    temp[4] = stu+duration
    MACHINE_ALLOC_VECTOR[machine-1].extend(temp)
    a = 1

def update_job_order_vector(job, time_unit):
```

```
    global JOB_ORDER_VECTOR
    JOB_ORDER_VECTOR[job-1] = time_unit

def clear_vectors(): #Clear and re-define vectors
    global JOB_ORDER_VECTOR
    global MACHINE_ALLOC_VECTOR
    del JOB_ORDER_VECTOR
    del MACHINE_ALLOC_VECTOR

    JOB_ORDER_VECTOR = [0] * J
    MACHINE_ALLOC_VECTOR = [[] for i in range(0, M)]

    if(JSSP_TYPE=='DYNAMIC'):

        for i in range(0, len(Events)):
            job = 0
            machine = Events[i][2]
            type = Events[i][1]
            duration = Events[i][3]
            stu = Events[i][4]
            #update_machine_alloc_vector(job, machine, type, duration, stu)

        for i in range(0, len(OP_info)):
            type = OP_info[i][0]
            job = OP_info[i][1]
            machine = OP_info[i][2]
            comp_prec = OP_info[i][3]
            stu = OP_info[i][4]
            if(type != 50):
                duration = get_duration_by_job_op(job, machine)
            else:
                duration = comp_prec

            adj_duration = 0
            #user needs to feed the STU after deducting the work that has already been took place
            if(type == 70):
                adj_duration = int((100 - comp_prec) * (duration / 100))
                update_machine_alloc_vector(job, machine, type, adj_duration, stu)
                update_job_order_vector(job, adj_duration)

            if(type==80):
                adj_duration = int((100 - comp_prec) * (duration / 100))
                update_machine_alloc_vector(job, machine, type, adj_duration, stu)
                update_job_order_vector(job, adj_duration)


def get_duration_by_job_op(job, machine):
    ext_op_per_job = Oij[job-1]
    #ext_op_per_job = x.tolist()
    op_index = ext_op_per_job.index(machine)
    output = Dij[job-1][op_index]
    return output

def get_ops_count_per_job(): #Return the number of operation each job has
    ops_count = [0]*J
    for i in range(0, J):
        ops_count[i] = len(Oij[i])
    return ops_count

def calculate_gene_position(op_count_per_jobs, job_index, op_index): #The absolute position of the gene in the chromosome
    ↪ is calcualted
    previous_job_sum = sum(op_count_per_jobs[0:job_index])
    return previous_job_sum + op_index

def get_comp_prec_by_op(job, op):

    if(JSSP_TYPE=='DYNAMIC'):
        size = len(OP_info)
```

```
        for i in range(0, size):
            ext_info = OP_info[i]
            if((job == ext_info[1]) and (op == ext_info[2])):
                if(ext_info[0]==80):
                    comp_prec = ext_info[3]
                    return comp_prec

                if(ext_info[0]==70):
                    comp_prec = 1000
                    return comp_prec

                if (ext_info[0] == 50):
                    comp_prec = ext_info[3]
                    return comp_prec
    return 0

def get_stu_by_op(job, op):
    size = len(OP_info)

    for i in range(0, size):
        ext_info = OP_info[i]
        if((job == ext_info[1]) and (op == ext_info[2])):
            if((ext_info[3]==80) or (ext_info[3]==70)):
                return ext_info[4]
            else:
                return ext_info[4]
        else:
            return 0

def generate_a_gene(op_count_per_jobs, job_index, op_index): #Generate a gene 1 x 6 [Seq.][Jop][Operation][Comppletion
    ↪ %][Duration][STU]

    rnd_start = gen_lib.get_random_number(0,100)
    job = job_index + 1

    operation = Oij[job_index] [op_index]
    duration = Dij[job_index][op_index]
    stu = get_next_stu(job, operation, rnd_start, duration)
    comp_prec = get_comp_prec_by_op(job, operation)

    if((comp_prec>0) and (comp_prec!=1000)):
        duration = int(((100−comp_prec)∗duration)/100)
        stu = get_stu_by_op(job, operation)

    gene = [0]∗6
    gene[0] = calculate_gene_position(op_count_per_jobs, job_index, op_index)
    gene[1] = job
    gene[2] = operation
    gene[3] = comp_prec
    gene[4] = duration
    gene[5] = stu
    time_unit = stu + duration + 1
    update_job_order_vector(job, time_unit)
    update_machine_alloc_vector(job, operation, 0,duration,stu)
    return gene

def fvc_chromo(chromo):#Calculates the fitness value for a chromosome
    ops_count_per_job = get_ops_count_per_job()
    end_time = [0]∗J
    for i in range(0, J):
        if(FITNESS_FUNC == 'T'):
            last_op = ops_count_per_job[i]−1
            op_position = calculate_gene_position(ops_count_per_job, i, last_op) ∗ 6
            extract_gene = chromo[op_position: op_position+6]
            end_time[i] = extract_gene[4] + extract_gene[5]
            output = sum(end_time)

        if(FITNESS_FUNC == 'M'):
            output = calculate_makespan(chromo)
```

```
            #if(FITNESS_FUNC == 'WT'):
                #calculated_weighted_tardiness(chromo):

        return output

def calculate_fitness_of_population(population): #Populate the Penalty filed[3] in the population element
        pop_size = len(population)
        for i in range(0,pop_size):
            optimized_chromo = optimize(population[i][4])
            fitness_value = fvc_chromo(optimized_chromo)
            population[i][3] = fitness_value
        return population

def generate_init_population():
        init_pop = [[] for j in range(0, INIT_POP_SIZE)]
        for i in range(0,INIT_POP_SIZE):
            chromosome = generate_a_chromo()
            temp = [[] for k in range(0, 8)]
            temp[0] = GENERATION_COUNT
            temp[1] = i + 1
            temp[2] = 0
            temp[3] = 0
            temp[4] = chromosome
            temp[5] = GENERATION_COUNT*1000 + (i+1)
            temp[6] = 0
            temp[7] = 0
            init_pop[i] = temp
        return init_pop

def rank_members_of_pop(population):
        pop_size = len(population)
        for i in range(0, pop_size):
            temp_pointer = i
            temp_min = population[i][3]
            for j in range(i+1, pop_size):
                temp = population[j][3]
                if(temp_min < temp):
                    temp_min = temp
                    temp_pointer = j
            if(temp_pointer!=i):
                temp_placeholder = population[temp_pointer]
                population[temp_pointer] = population[i]
                population[i] = temp_placeholder
                x = population[i][5]
            id = generate_unique_id(population[i][5], i)
            population[i][5] = id
        #for i in range(0,pop_size):
            #print('Ranking ', population[i][3], population[i][5])
        return population

def generate_unique_id(current_id, ref):
        generation = int(current_id/1000)
        member_id = current_id%1000

        return (generation*1000)+(ref+1)

def generate_mate_plan(population):
        parent_size = 25
        #mate_plan = TSel.get_mate_plan(population, parent_size)
        mate_plan = generate_random_mate_plan(population, parent_size)
        return mate_plan

def generate_random_mate_plan(population, size):
        pop_size = len(population)
        order_list = [i for i in range(0, pop_size)]
        mate_plan = []
        for i in range(0, size):
            parent_set = []
```

```
        for j in range(0,2):
            current_pop_size = len(order_list)
            parent_index = gen_lib.get_random_number(0, current_pop_size)
            parent_index_in_pop = order_list[parent_index]
            parent_id = population[parent_index_in_pop][5]
            parent_penalty = population[parent_index_in_pop][3]
            list_ = [parent_id,parent_penalty]
            parent_set.append(list_)

        mate_plan.append(parent_set)
    return mate_plan


def generate_offspring(population, mate_plan):
    mate_count = len(mate_plan)
    offspring_chromo = []
    for i in range(0, mate_count):
        mate_pattern = mate_plan[i]
        weak_parent = mate_pattern[0][0]
        strong_parent = mate_pattern[1][0]
        weak_parent = get_member_from_unique_id(population, weak_parent)
        strong_parent = get_member_from_unique_id(population, strong_parent)
        offspring_chromo.append(crossover(weak_parent, strong_parent))

    offspring = construct_offspring_population(offspring_chromo, mate_plan)
    return offspring

def construct_offspring_population(chromo_set, mate_plan):
    pop_size = len(chromo_set)
    new_population = []
    for i in range(0, pop_size):
        temp = [[] for k in range(0, 8)]
        temp[0] = 99999
        temp[1] = i + 1
        temp[2] = 0
        temp[3] = 0
        temp[4] = chromo_set[i]
        temp[5] = 99999 * 1000 + (i + 1)
        temp[6] = mate_plan[i][0][0]
        temp[7] = mate_plan[i][1][0]
        new_population.append(temp)
        a =1
    return new_population

def get_member_from_unique_id(population, unique_id):
    pop_size = len(population)
    for i in range(0, pop_size):
        if(population[i][5] == unique_id):
            return population[i]

def get_fixed_stu(job, op):
    size = len(OP_info)

    for i in range(0, size):
        ext_info = OP_info[i]
        if ((job == ext_info[1]) and (op == ext_info[2])):
            return ext_info[4]
    return 0

def crossover(weak_parent, strong_parent):
    weak_chromo = weak_parent[4]
    strong_chromo = strong_parent[4]
    gene_count = int(len(weak_chromo) / 6)
    offspring_member = []

    clear_vectors()

    crossover_operation = True
    p = 0
```

```
while crossover_operation:
    gene = [0]*6
    stu = 0
    seq = weak_chromo[6 * p]
    job = weak_chromo[6 * p + 1]
    operation = weak_chromo[6 * p + 2]
    comp_prec = weak_chromo[6 * p + 3]
    duration = weak_chromo[6 * p + 4]
    stu_from_strong_parent = strong_chromo[6 * p + 5]
    if(comp_prec>0):
        a=1

    if(comp_prec==0):
        if(gen_lib.biased_binary_random_decision(CR)):
            if(gen_lib.biased_binary_random_decision(MR)): #Mutate instead of crossover
                gene[0] = seq
                gene[1] = job
                gene[2] = operation
                gene[3] = comp_prec
                gene[4] = duration
                random_stu = gen_lib.get_random_number(0,stu_from_strong_parent+150)
                stu = get_next_stu(job, operation, random_stu, duration)
                gene[5] = stu
            else: #Crossover operation only
                gene[0] = seq
                gene[1] = job
                gene[2] = operation
                gene[3] = comp_prec
                gene[4] = duration
                stu = get_next_stu(job, operation, stu_from_strong_parent, duration)
                gene[5] = stu
        else:
            gene[0] = seq
            gene[1] = job
            gene[2] = operation
            gene[3] = comp_prec
            gene[4] = duration
            stu = get_next_stu(job, operation, weak_chromo[6 * p + 5], duration)
            gene[5] = stu

    if((comp_prec>0) and (comp_prec<=100)):
        stu = get_fixed_stu(job, operation)
        gene[0] = seq
        gene[1] = job
        gene[2] = operation
        gene[3] = comp_prec
        gene[4] = duration
        # stu = get_next_stu(job, operation, weak_chromo[6 * p + 5], duration)
        gene[5] = get_fixed_stu(job, operation)

    if(comp_prec==1000):
        stu = get_fixed_stu(job, operation)
        gene[0] = seq
        gene[1] = job
        gene[2] = operation
        gene[3] = 1000
        gene[4] = duration
        #stu = get_next_stu(job, operation, weak_chromo[6 * p + 5], duration)
        gene[5] = get_fixed_stu(job, operation)
    offspring_member.extend(gene)
    x = len(weak_chromo)
    y = len(offspring_member)

    time_unit = stu + duration + 1
    update_job_order_vector(job, time_unit)
    update_machine_alloc_vector(job, operation, 0, duration, stu)
    p = p + 1
    if(len(weak_chromo) == len(offspring_member)):
        crossover_operation = False
```

```
        DEBUG.__save_to_file__(offspring_member,'crossover_result.dat',DO_DEBUG)

        return offspring_member

def selection(parent_generation, offspring_1, offspring_2, new_pop_size):
        global GENERATION_COUNT
        GENERATION_COUNT = GENERATION_COUNT+1
        total_pop = []
        total_pop.extend(parent_generation)
        total_pop.extend(offspring_1)
        total_pop.extend(offspring_2)

        #new_population = TSel.ts_for_new_gen(total_pop, 50)
        new_population = random_new_gen_selection(total_pop, 50)

        for i in range(0, len(new_population)):
                new_population[i][0] = GENERATION_COUNT
        DEBUG.__save_to_file__(new_population[25][4],'new.dat',DO_DEBUG)
        return new_population

def random_new_gen_selection(total_pop, size):
        pop_size = len(total_pop)
        order_list = [i for i in range(0, pop_size)]
        new_gen = []
        for i in range(0, size):
                current_pop_size = len(order_list)
                member_index = gen_lib.get_random_number(0, current_pop_size)
                member_index_in_pop = order_list[member_index]
                member = total_pop[member_index_in_pop]

                new_gen.append(member)
        return new_gen

SOL_TRACK = []
SOL_CHROMO = 0

def calculate_makespan(chromo):
        extract_stu=chromo[5::6]
        extract_duration = chromo[4::6]
        #end_point = extract_stu + extract_duration
        end_point = [extract_stu[i] + extract_duration[i] for i in range(0, len(extract_stu))]
        makespan = max(end_point)
        #duration_index = extract_stu.index(makespan)
        #makespan = makespan + extract_duration[duration_index]
        return makespan

def best_solution_tracker(member, fv):
        global SOL_CHROMO

        print('Total_Makespan_',calculate_makespan(member))

        if(len(SOL_TRACK)!=0):
                min_value = min(SOL_TRACK)
                if(fv < min_value):
                        SOL_CHROMO = member
        else:
                SOL_CHROMO= member
        SOL_TRACK.append(fv)

##---------------Main Prog Starts-----------------##
def main():
        started_time = time.time()

        initial_population = generate_init_population()
        initial_population = calculate_fitness_of_population(initial_population)
        initial_population = rank_members_of_pop(initial_population)

        mate_plan = generate_mate_plan(initial_population)
        offspring_G1 = generate_offspring(initial_population, mate_plan)
```

```
    offspring_G1 = calculate_fitness_of_population(offspring_G1)
    offspring_G1 = rank_members_of_pop(offspring_G1)

    mate_plan = generate_mate_plan(initial_population)
    offspring_G2 = generate_offspring(initial_population, mate_plan)
    offspring_G2 = calculate_fitness_of_population(offspring_G2)
    offspring_G2 = rank_members_of_pop(offspring_G2)

    new_population = selection(initial_population, offspring_G1, offspring_G2, 50)

    for i in range(0,MAX_GEN−1):
        min_value = 0
        if(len(SOL_TRACK)!=0):
            min_value = min(SOL_TRACK)

        print('started ',GENERATION_COUNT, new_population[49][3], min_value)
        new_population = calculate_fitness_of_population(new_population)
        new_population = rank_members_of_pop(new_population)
        mate_plan = generate_mate_plan(new_population)
        offspring_G1 = generate_offspring(new_population, mate_plan)
        offspring_G1 = calculate_fitness_of_population(offspring_G1)
        offspring_G1 = rank_members_of_pop(offspring_G1)

        mate_plan = generate_mate_plan(new_population)
        offspring_G2 = generate_offspring(new_population, mate_plan)
        offspring_G2 = calculate_fitness_of_population(offspring_G2)
        offspring_G2 = rank_members_of_pop(offspring_G2)

        new_population = selection(new_population, offspring_G1, offspring_G2, 50)

        best_solution_tracker(new_population[49][4], new_population[49][3])

    DEBUG.__save_to_file__(SOL_CHROMO,'solutions'+str(min(SOL_TRACK))+'.dat',1)

    #print('mate_plan[0]',mate_plan)

    ended_time = time.time()
    print('Execution Time : ',(ended_time − started_time),'s')

##−−−−−−−−−−−−−−Main Prog Ends−−−−−−−−−−−−−−−−##

##Overlap Detection − Use for DeBug
def overlap_detect(chromo):
    #print ('Overlap detect init')
    genes_count = int(chromo.size/6)
    violation_detected = False
    for i in range(0,J):
        op_pointer = 0
        ext_job_order = Oij[i]
        no_of_ops_in_job = len(ext_job_order)
        for j in range (0, no_of_ops_in_job):
            selected_op = ext_job_order[j]
            for k in range(0,genes_count):
                if((chromo[k*6+1]==i+1) and (chromo[k*6+2]==selected_op)):
                    duration = chromo[k * 6 + 4]
                    op_start = chromo[k * 6 + 5]
                    op_end = op_start + duration
                    if(op_end < op_pointer):
                        print ('Violation Found',chromo[k*6:k*6+5],op_start,duration, op_end)
                        violation_detected = True
                        #For DeBug purposes
                        #sys.exit()
                    op_pointer = op_end + 1 #points to the next starting date
    return violation_detected

##−−−−−−−−−−−−−Local Optimization−−−−−−−−−−−−−−−−−−−−−−##

def optimize(chromo):
    clear_vectors()
```

```
        sorted_chromo = sort_chromo(chromo)
        genes = int(len(sorted_chromo)/6)

        for i in range(0, genes):
                ext_gene = sorted_chromo[6 * i : 6 * i + 6]

                job = ext_gene[1]
                operation = ext_gene[2]
                comp_prec = ext_gene[3]
                duration = ext_gene[4]
                if(comp_prec>0):
                        stu = get_fixed_stu(job, operation)
                else:
                        stu = get_next_stu(job, operation, 0, duration)
                sorted_chromo[6 * i + 5] = stu

                time_unit = stu + duration + 1
                update_job_order_vector(job, time_unit)
                update_machine_alloc_vector(job, operation, 0, duration, stu)

        DEBUG.__save_to_file__(sorted_chromo, 'lso.dat',DO_DEBUG)

        return sorted_chromo


def sort_chromo(chromo):
        genes = int(len(chromo)/6)

        for i in range(0, genes):
                temp_min = chromo[6 * i + 5]
                pointer = i
                for j in range(i, genes):
                        if(chromo[6 * j + 5] < temp_min):
                                temp_min = chromo[6 * j + 5]
                                pointer = j
                if(pointer!=i):
                        place_holder = chromo[6 *pointer: 6 * pointer + 6]
                        for k in range(0,6):
                                chromo[pointer * 6 + k] = chromo[i * 6 + k]

                        for k in range(0,6):
                                chromo[i * 6 + k] = place_holder[k]

        return chromo

##END OF PROGRAM##
########################################################
```

A.2 Tournament Selection Algorithm

```
## File : tournament_selection.py.py ##
##Algorithm implemented by Buddhika Kurera, (c) 2017##
## Email − bckurera AT geneai dot edu dot com ##
import gen_lib as GL

def get_mate_plan(population, parents_size):
        pop_size = len(population)
        selected_parents = []
        create_mate_plan = True
        while create_mate_plan:
                selection = tournament_selection(population, 4, 6)
                selection_1 = selection[0]
                selection_2 = selection[1]

                if(selection_1[1] < selection_2[1]):
```

```
            selection[0] = selection_2
            selection[1] = selection_1
        else:
            selection[0] = selection_1
            selection[1] = selection_2

        duplicates_found = check_for_duplicates(selection, selection_1[0], selection_2[0])

        if(duplicates_found==False):
            selected_parents.append(selection)

        if(len(selected_parents)==parents_size):
            create_mate_plan = False

    return selected_parents


def check_for_duplicates(string, value_1, value_2):
    occurance_1 = sum(string[i].count(value_1) for i in range(0,len(string)))
    occurance_2 = sum(string[i].count(value_2) for i in range(0, len(string)))

    if((occurance_1>1) or (occurance_2>1)):
        return True
    else:
        return False


def tournament_selection(population, teams , team_size):
    pop_size = len(population)
    team_list = [[] for i in range(0, teams)]
    member_list = []

    for i in range(0, pop_size):
        set = [0]*2
        set[0] = population[i][5]
        set[1] = population[i][3]
        member_list.append(set)

    team_forming = True
    while team_forming:
        remaining_member_count = len(member_list)
        selected_team = GL.get_random_number(0, teams)
        if (len(team_list[selected_team]) != team_size):
            random_no = GL.get_random_number(0, remaining_member_count)
            selected_member = member_list.pop(random_no)
            team_list[selected_team].append(selected_member)

        if(sum(len(team_list[i]) for i in range(0,teams)) == teams*team_size):
            team_forming = False

    for i in range(0, teams):
        team_list[i] = sort_teams(team_list[i])

    for i in range(0, teams):
        for j in range(0,len(team_list[i]) - 1):
            current_team_size = len(team_list[i]) - 1
            competitor_1 = team_list[i][current_team_size]
            competitor_2 = team_list[i][0]

            total_fv = competitor_1[1] + competitor_2[1]
            comp_1_prob = int((competitor_2[1]*100)/(total_fv))

            game_result = GL.biased_binary_random_decision(comp_1_prob)

            if(game_result):#True - competitor 1 wins
                team_list[i].pop(0)
            else:
                team_list[i].pop(current_team_size)
        a = 1
```

```
    final_round_team = []
    for i in range(0, teams):
        final_round_team.append(team_list[i][0])
    final_round_team = sort_teams(final_round_team)


    for i in range(0,len(final_round_team) - 2):
        current_team_size = len(final_round_team) - 1
        competitor_1 = final_round_team[current_team_size]
        competitor_2 = final_round_team[0]

        total_fv = competitor_1[1] + competitor_2[1]
        comp_1_prob = int((competitor_2[1]*100)/(total_fv))

        game_result = GL.biased_binary_random_decision(comp_1_prob)

        if(game_result):#True - competitor 1 wins
            final_round_team.pop(0)
        else:
            final_round_team.pop(current_team_size)

    #print('final_round_team', final_round_team)
    return final_round_team


def sort_teams(team): #[unique id, penalty]
    team_size = len(team)
    for i in range(0, team_size):
        temp_min = team[i][1]
        temp_pointer = i
        for j in range(i, team_size):
            temp = team[j][1]
            if(temp>temp_min):
                temp_min = temp
                temp_pointer = j
        if(temp_pointer!=i):
            place_holder = team[temp_pointer]
            team[temp_pointer] = team[i]
            team[i] = place_holder
    return team

def ts_for_new_gen(old_population, size):
    total_population_size = len(old_population)
    order_list = [i for i in range(0,total_population_size)]
    new_population = []

    team_size = 8
    teams = 5

    run_selection = True
    while run_selection:
        order_list = [i for i in range(0, total_population_size)]
        game_teams = [[] for i in range(0, teams)]
        for i in range(0, teams):
            for j in range(0, team_size):
                current_size = len(order_list)
                select_random_member_index = GL.get_random_number(0, current_size)
                selected_member_ref = order_list.pop(select_random_member_index)
                selected_member = old_population[selected_member_ref]
                game_teams[i].append(selected_member)

        for i in range(0, teams):
            play_game = True
            while play_game:
                remaining_team_size = len(game_teams[i])
                if(remaining_team_size==2):
                    break
                competitor_1 = game_teams[i][remaining_team_size-1]
```

```
            competitor_2 = game_teams[i][0]
            penalty = [competitor_1[3], competitor_2[3]]

            probability = int((max(penalty)*100)/sum(penalty))

            if(GL.biased_binary_random_decision(probability)): #True - competitor 1 wins
                game_teams[i].pop(remaining_team_size-1)
            else:
                game_teams[i].pop(0)

        for i in range(0, teams):
            if (len(new_population) != size):
                occurance = new_population.count(game_teams[i][0])
                occurance = occurance + new_population.count(game_teams[i][0])
                if(occurance == 0):
                    new_population.append(game_teams[i][0])
                    new_population.append(game_teams[i][1])
            else:
                run_selection = False

    return new_population
```

A.3 Roulette Wheel Selection Algorithm

```
## File : rol.py ##
##Algorithm implemented by Buddhika Kurera, (c) 2017##
## Email - bckurera AT geneai dot edu dot com ##

import numpy as np
import random as rand

roulette_wheel = np.array((0))
slot_count = 0

def init_roul_wheel(value_array):

    slot_count = 0
    i=0
    arrsize = value_array.size
    while i < arrsize/2:
        slot_count = slot_count + value_array[2*i+1]
        i = i + 1
    roulette_wheel = np.zeros((slot_count),dtype=np.int)
    #print(roulette_wheel)
    i = 0

    while i < arrsize/2:
        rv = value_array[2*i]
        bv = value_array[2*i+1]
        j = 0
        while j<bv:
            t = rand.randint(0,slot_count-1)
            wheel_alloc = roulette_wheel[t]
            if wheel_alloc == 0:
                roulette_wheel[t] = rv
                j = j + 1
        i = i + 1
    return (roulette_wheel)

def spin(rw):
    slot_count = rw.size
    randno = rand.randint(0,100000)
    rot_degree = randno%360
    #rot_degree = rand.randint(0,359)
    rot_unit = 360/slot_count
```

```
rol_no = (rot_degree − (rot_degree%(rot_unit)))/rot_unit
rol_value = rw[int(rol_no)]
return rol_value
```