MAST90098 Group Project
Group 5

Genetic Algorithm for Makespan scheduling with knowledge based operator

Wenrui Yuan *(1159794)*
Yaning Ma *(1096450)*
Chayasin Sae-tia *(1059862)*

Contents

1 Introduction

1.1 Motivation

The Makespan Scheduling Problem(MS) is a famous classic NP-Hard problem. Research on this problem began in the 1950s. From the 1950s to the 1980s, the main research direction was precise optimization methods, such as efficient rule-based methods, mathematical programming methods, and branch definition methods, which can achieve precise optimal solutions to specified MS problems in polynomial time. But only small-scale problems can be solved with precise optimization methods. With the continuous development of computer software, hardware and algorithms, the research methods for MS have gradually changed from the method of precise optimization program to the approximate method since the 1980s. Among them, there are not only construction-based methods such as priority scheduling rules, insertion algorithms, and bottleneck-based heuristic algorithms, but also artificial intelligence methods such as neural networks, genetic algorithms, simulated annealing and particle swarm optimization, etc.

Genetic algorithm is a global and heuristic parameter search method based on natural genetic mechanism. It has the characteristics of fast running speed and strong versatility. It is also suitable for dealing with complex nonlinear problems that are difficult to solve by traditional search methods. Hence, in this article we choose the genetic algorithm to try to solve the MS problem.

1.2 Problem specification

In this problem, we want to schedule n jobs with designated processing times on m identical machines in such a way that the whole processing time is minimised. Firstly, we assume that p_i represents the time for completion of each job i. At the same time, we use the product of the indicator function $s(i,j)$ and p_i to represent the time to process job j on machine i. In other words, if $s(i,j)=1$, it means job j is processed on machine i and if $s(i,j)=0$, it means job j is not processed on machine i. Then the sum of the time spent on all jobs processed on all machines is the cost of the problem, which is the goal we want to minimize.The input of this problem can be seen as the $m*n$-dimensional indicator matrix S and the n-dimensional vector P. The element $s(i,j)$ of matrix S represents that job j is processed on machine i. Matrix S has the following properties:
1. All elements are either 1 or 0;
2. The sum of all elements is n;
3. Only one element in each row is 1.
Vector $P=(p_1,p_2,...,p_n)$, where p_i represents the operation time of job i. The cost of each instance in output is

$$C(S,P)=max\{\sum_{j=1}^{n}s(1,j)p_j,...,\sum_{j=1}^{n}s(k,j)p_j,...,\sum_{j=1}^{n}s(m,j)p_j\}$$

And our object is $minC$.

1.3 Expectation

In the second chapter of this article we discussed some developments on MS problem. In the third chapter, the principle of genetic algorithm is introduced, and in the fourth chapter, several numerical experiments are done to explore the effect of genetic algorithm on MS problem under different parameters. At the same time, in Chapter 5, we summarized the results and looked forward to the development and application of genetic algorithms in MS problems.

2 Literature Review

At present, there are many researches on algorithms for MS problems, which can be divided into optimization algorithms and approximate algorithms. The optimization algorithms for scheduling are mainly divided into exact optimization methods and approximate methods. The exact optimization methods include efficient rule approaches, mathematical programming approaches, branch definition methods, and etc. The approximate methods include constructive methods, artificial intelligence, local search and meta-heuristic algorithms.

2.1 Exact optimization procedure methods

2.1.1 Early exact optimization procedure methods

Efficient rule-based methods are the earliest solution to this problem. People can formulate a series of rules according to the input data and task objectives, process tasks in a certain order accurately, and then the exact optimal solution could be obtained. In 1954, the well-known Johnson rule[11] was proposed to solve the flow shop scheduling problem of two machines with the goal of maximum process time. In addition, Wagner[29] discovered that mathematical programming techniques could solve MS problems in 1959. However, the above methods cannot be applied on a large scale due to long calculation time or difficulty in finding the optimal solution. Therefore, enumeration technology was introduced. The most famous of these methods are the branch and bound method[16] and the backtracking algorithm[9] in 1965. There are many branch and bound methods based on enumeration thought. The main difference lies in the three aspects of branching rules, delimitation mechanism and upper bound. Although the running time of these algorithms is within an acceptable range, the scale of instances they can calculate is still very limited. Experimental reports show that even with today's large machines, these methods usually cannot use more than 100 processes.

2.1.2 Combination algorithms based on exact optimization procedure methods

In 1960, Manne[17] mixed discrete integer and linear programming methods and then applied it to typical makespan scheduling problems. Compared with Wagner's formula, this mathod contains much fewer variables and is more computationally efficient. In addition, Sarin[25] and Potts[23] improved the B&B method in terms of analysis rules, boundary mechanism and upper bound generation respectively.

People tended to P=NP during this period in the 1960s, so they tried to design exact optimization algorithms with polynomial time complexity in order to find the optimal solution to the problem. An exact algorithm can theoretically get an optimal solution to the problem. However, for large-scale problems, exact optimization methods cannot complete the calculation within a good running time.

2.2 Approximate methods

After the 1970s, people no longer pursued the use of precise algorithms to find the optimal solution of the problem, but sought a satisfactory solution to the problem within an acceptable time through an approximate algorithm.

2.2.1 Constructive methods

The priority dispatch rules method is the simplest constructive method. The earliest dispatch rules were proposed by Jackson[10] and Smith[27]. The algorithm of Giffer and Thompson[7] is a typical representative of priority rule scheduling algorithms. In 1977, Panwalker and Iskander[20] summarized more than 100 methods. The most important are the following 8 methods: SPT (Shortest Processing Time) law, that is, the process with the shortest processing time is preferred; SLT (Longest Processing Time) law, that is, priority is given to the process with the longest processing time; FCFS (First Come First Served) law, that is, first come first served; MWKR (Most Work Remaining) rule, that is, the job with the longest remaining processing time is given priority; LWKR (Least Work Remaining) rule, that is, the job with the shortest remaining processing time is given priority; FA (First Available) rule, that is, the process that can be processed immediately is preferred; MOPNR (Most Operation Remaining) law, that is, the operation with the largest number of remaining processes is preferred; and the RANDOM rule is to randomly select a process.

A constructive approach can sometimes get a JSP solution very quickly, but it may produce an infeasible solution, especially when the problem is complex. In order to optimize the final results of the search, it is usually necessary to set up complex heuristic rules. When the system and rules are too complex, there may be many rules that restrict each other or contradict each other or fall into an endless loop. Therefore, it is difficult to find a feasible solution to satisfy all the rules.

2.2.2 Artificial intelligence methods

In the summer of 1956, a group of visionary young scientists, led by McCarthy, Minsky, Rochester, and Shennon, gathered together to study and discuss a series of related problems of using machines to simulate intelligence, and put forward for the first time The term "artificial intelligence". Artificial intelligence is a branch of computer science. It attempts to understand the essence of intelligence and produce a new intelligent machine that can react in a similar way to human intelligence. Research in this field includes robotics, language recognition, image recognition, Natural language processing and expert systems, etc. Since the birth of artificial intelligence, the theory and technology have become increasingly mature, and the field of application has also continued to expand.

In 1988, Foo and Takefuji[26] proposed a two-dimensional Hopfield TSP type neuron matrix and coding strategy to solve the MS problem. In 1999, Rovithakis.G.A[24] proposed neural network for FMS system. In 2000, Yang.S.X and others[30] adopted neural networks and heuristic algorithms that satisfy constraints for general-purpose workshop scheduling. Akyol and Bayhan[1] conducted an extensive literature review on the application of neural networks (NN) in scheduling problems, and divided them into four categories: Hopfield type networks (HNN), multilayer perceptrons, competition based networks and hybrid approaches in accordance with the different structures.

Besides, driven by actual needs, knowledge-based methods and expert systems have been greatly developed. The knowledge-based scheduling method is that the expert system automatically generates scheduling or assists people to schedule it. It is the product of the combination of traditional scheduling methods and knowledge-based scheduling rules. The more famous expert systems are: ISIS, MPECD, OPIS, SONIA[34], etc.

Neural networks have many shortcomings, such as unsatisfactory training approximation, over-fitting, generalization, over-learning and non-convergence, etc. In addition, some expert systems and knowledge-based methods are very difficult to establish the rules of the entire system, because there are too many factors to consider in a large and complex system.

2.2.3 Meta-heuristic Methods

Methods based on local search and heuristic algorithms are widely used to find the optimal solution of complex models. Methods such as genetic algorithm, ant colony algorithm, particle swarm algorithm and so on that imitate the natural world or the biological world have been applied in the Makespan scheduling problem and achieved good results.

Simulated Annealing Algorithms[14](SA) is a new search technique proposed by Kirkpatrick et al. applying the annealing process in metal thermal processing to the field of combinatorial optimization problems. The algorithm simulates the statistical physical process of heating solids gradually annealing from the maximum energy state to the minimum energy state through the control parameters. Metropolis acceptance criteria are used to control the algorithm process with a set of cooling schedule parameters, and the approximate optimal solution can be found in polynomial time. The improvement of simulated degradation algorithm includes heating annealing method, memory annealing method and so on. The simulated annealing algorithm can better avoid local optima, but the convergence speed of the algorithm is very slow, and the search space is too large to reduce the temperature and it is difficult to grasp. These factors become resistance to further application.

Tabu Search is a heuristic algorithm that seeks approximate optimal solutions for optimization combinatorial problems. It was proposed and formalized by Glover[8], and Barnes and Nowicki improved on TS[2][19]. TS is a deterministic algorithm designed to jump out of the local optimum. When it falls into the local optimum, it does an upward movement. According to the search information stored in the memory, the same or similar actions as the previously obtained solution are forbidden to avoid getting local optimal solution. However, tabu search has to face many problems, such as initial solution problem, neighborhood structure problem, search strategy and length of taboo table and so on. Only if these problems are solved well, the tabu search algorithm can perform better. Aihua proposed a new neighborhood structure for tabu search[33], using an improved conversion bottleneck algorithm to obtain the initial solution and improving the search strategy[32]

In 1985, Davis[4] first used genetic algorithm (GA) to solve scheduling problems. It expresses the solution of the problem as the survival process of the fittest of the chromosome. Through the continuous evolution of the chromosome group from generation to generation, including operations such as selection, crossover, and mutation, it finally converges to the "most adapted to the environment" individual, so as to find the optima solution or satisfactory solution. Falkenauer et al.[6] enhanced this method by encoding all operations of each machine as a preferred string of symbols. Kuzapski et al.[15] implemented a method for generating a good initial population by evolving the priority scheduling rule into the arrival of the optimal final solution for the genetic algorithm. Zhang et al.[35] proposed a multi-group genetic algorithm based on multi-objective scheduling of flexible job shop.

There are some other algorithms that can be used to deal with MS Problem and other broader problems. For example, ant colony optimization (ACO), which mimics the foraging process of ant colonies, is also a meta-heuristic algorithm proposed by Colorni et al[3] in 1991. He is also the first researcher to use this method to solve the Traveling Salesman Problem, which is also a classical NP-Hard problem. Particle swarm optimization (PSO) is a evolutionary meta-heuristic technology derived from bird prey behavior, which was proposed by Eberhart and Kennedy[13]. On the basis of observing the law of predation activities of flying birds, the core idea of the algorithm is to use the sharing of information by individuals in the group so that the movement of the entire group produces an evolutionary process from disorder to order in the problem solving space, so as to obtain the pptimal solution of problem. Besieds, there are methods such as differential evolution[28], fuzzy logic[21], and firefly algorithm[31] that are not described here.

The meta-heuristic method has achieved fruitful results, but there are still many unsolved problems.Some meta-heuristic algorithms can often achieve better efficiency in global search, but they are easy to fall into local optimal for local search, and some are just the opposite. Mixing the existing multiple heuristic methods may be more conducive to obtaining the optimal solution, that is, integrating various scheduling algorithms to solve the makespan scheduling problem to adapt to the complex scheduling environment, and give full play to the advantages of various scheduling algorithms. At the same time, further research on the constraints of the local search algorithm and the strengthening of convergence and calculation speed are needed. The neighborhood selection of the local search algorithm also needs further research. When it falls into the local optimal solution, it is necessary to design a reasonable jumping strategy to make it escape smoothly.

3 Algorithms

3.1 The concept of Genetic Algorithm

The fundamental concept of the genetic algorithm is to perform the natural selection virtually in an algorithm. The population we want to consider is the population of *solutions*, we consider each individual solution instance as an individual of the population, which can be reproduce and mutate to build new generation with some improve in their gene, therefore they have better change to survive (better solution).

More formally, the genetic algorithm consist of four main different steps. First, how we encode a solution instance into a chromosome. Next is the reproduction function, we need to consider the probability of how to choose individuals parents, one might think that if a individual which has better solution might have higher chance to reproduce and that is, in fact, what we implement in the algorithm. Then, we need to look deeper to see how each pair of individuals reproduce, we call this the *crossover* step. Similar to the crossover in human chromosome, we need to define how each gene swap, i.e. position and number of gene. Next, like in nature, not every crossovers are completely successful, some children might have different gene from their parent because of the mutation, so we need to let the new generation have some probability to mutate, which might lead to a better fitness individual. The algorithm repeat those steps of reproduction and mutation until a certain criteria is satisfied then the algorithm stop.

The rest of the section will explains more details about each steps stated above, such as coding of chromosome, the distribution for reproduction, crossover function and mutation function. The last part will illustrate the pseudocode of the algorithm and description of the parameters, as well as the analysis. It will be many symbols and notations later on this section, however if one can imagine the picture of real chromosomes that will help a lot in understanding the algorithm.

3.2 Chromosomes Coding

There are different ways of coding, most problems use binary-coding (0 and 1) to code the chromosome, however for the makespan scheduling problem is it difficult to use only 0 and 1, that will make the crossover and mutation function complicated. Therefore, in this project we consider a chromosome is an array of n integers, which n in the number of jobs, and each integer has value between 1 and m, where m is the number of machine [18]. For example, a chromosome might be

$$\text{chrom}_1 = \{k_1, k_2, \dots, k_n\} \quad \text{for} \quad k_i \in 1, \dots, m,$$

k_1 represents the k_1^{th} machine where the job 1 goes in, similarly, k_i represent the k_i^{th} machine where the job i goes in.

For the initial population, we can generate k sample of n-digit of random number from $\{1, 2, \dots, m\}$.

3.3 Reproduction

First, we consider the cost of the chromosome, which is the maximum time of in the machines if we put all the jobs to its machine corresponding to that chromosome.

$$G(ch_i) = max \left\{ \sum_{k \in S_1} ch_i[k], \sum_{k \in S_2} ch_i[k], \dots, \sum_{k \in S_m} ch_i[k] \right\}$$

Then, we define the fitness function as follows,

$$f_i = \alpha \exp\{-\beta G(ch_i)\},$$

where α and β is positive real number which are turning parameters, we will set it to 1 by default. Now, we assume that p_i is a selection probability of a chromosome i, that is

$$p_i = \frac{f_i}{\sum_{\text{all } i} f_i}.$$

We have $\boldsymbol{p} = \{p_1, \ldots, p_k\}$ for selection distribution, which intuitively enough to say that the chromosome whose has a lower cost (better fitness) will have higher chance of reproducing and will lead to better new generation.

3.4 Crossover

The crossover process is similar to the crossover in nature which is a child is reproduced by combining gene from its parents. In this project we consider uniform crossover and k-points crossover from 2 parents which chosen from the selection distribution $\boldsymbol{p}$ in previous part. For example we consider 2-points crossover, suppose we have 2 parents chromosomes, we select the length, say ℓ, and the starting point, say $indxS$. Then, we swap the gene in that range to produce children of the next generation as show in 1, we can either select both of them or just one of them.

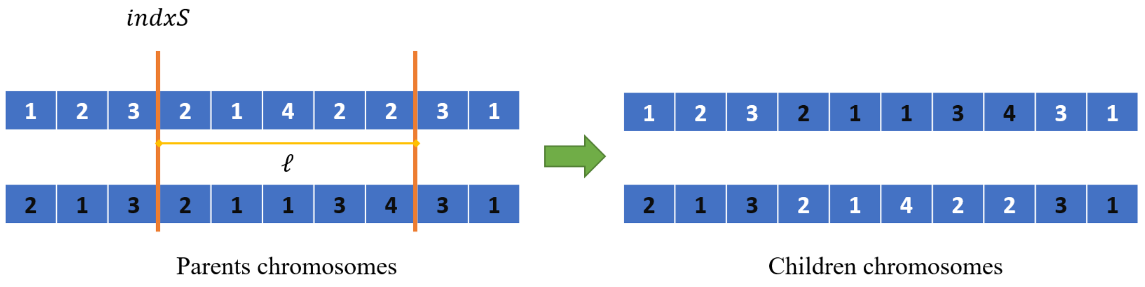

Figure 1: 2-point crossover.

For the uniform crossover, we set a fixed probability,p_u,of selecting a gene from a parent and for each gene j we choose a gene from the parent with probability p_u and the other parent with $1 - p_u$, this might similar to all point mutation but we consider it in probabilistic way.

3.5 Mutation

In this project, we have two mutation functions. The first is a simple mutation function, we select a number from $\mu_{rate} \in [0, 1]$ to be our mutation rate and than for each gene in the chromosomes, with μ_{rate} probability, we replace a random number to that gene, i.e. we put that job to a new random machine; and with $1 - mu_rate$ probability, we do not change the gene.

The second mutation is we fix amount of gene (we fixed the percentage) to be mutated and than random the the position of the gene, without replacement, and replace a new random number to that gene.

One might think that if we already have that optimal solution, we should not perform the mutation, by this reason, some fix number of the best chromosomes will not be mutated. However, this mutation step improves the results of the algorithm significantly especially with appropriate parameters.

3.6 Knowledge-based operator

The genetic algorithm (GA) does provide insights for various combinatorial optimization problems and surprisingly good performance given a small instance size [18]. Suffering from its problem-agnostic nature, however, the genetic algorithm in itself does not provide any promising lower bounds nor optimal solutions. Moreover, some observation from early stage of our GA development do affirm that the traditional GA approach becomes less practical as the instances size increases, where results are often no better than randomized search [5]. To address this shortcoming of GA and provides better solution bounds, we borrow novel designs of hybrid genetic algorithm from recent literature [22, 12] and propose a problem-specific operator that serves a similar function as the mutation operator. The basic structure can be interpreted with the following pseudocode:

Algorithm 1 Knowledge-based operator

```
  Input: single chromosome, $ch \leftarrow \{k_1, \cdots, k_n\}$
  Compute partial fitness: $f \leftarrow \alpha \exp(-\beta \cdot g(ch)) = \{f_1, \cdots, f_m\}$
  $old \leftarrow \{i \mid f_i < \texttt{mean}(f)\}$
  $new \leftarrow \{j \mid f_j > \texttt{mean}(f)\}$
  $P \leftarrow new / \sum_j^{|new|} new_i$
  for $i \leftarrow 1$ to $|ch|$ do
      select $j \in new$ based on $P$
      if $k_i \in old$ then
          $k_i \leftarrow j$
      end if
  end for
  Output: modified $ch \leftarrow \{k'_1, \cdots, k'_n\}$
```

The basic idea of our proposed method is similar to the swapping operation often seen in traditional mutation operator design. With problem-specific knowledge, we know that a given solution is "bad" if there are too many jobs on a single machine, which translates to a chromosome with too many same entries. Ideally we want n/m jobs assigned to each machine if there are n jobs and m machines in total. This inspires us to investigate the local piece of chromosome to find a machines with bad assignment (i.e. too many jobs) and swap the genes with a good one. In which case, we are "evening out" the number of jobs assigned to each machine. We believe that this is also a randomized search, but in a controlled fashion, where we apply the knowledge that may help to generate better solutions.

3.7 Pseudocode

Algorithm 2 Genetic Algorithm for MSP

```
Initial Step
Setting parameters for the function.
Set $t_0 \leftarrow time(now)$
Randomly generate chromosomes, $\boldsymbol{chs} \leftarrow \{ch_1, ch_2, \ldots, ch_k\}$.
Find the cost, $g_i$, of each chromosome.
while the criteria is not satisfied do
    Sort the chromosomes by its cost $\boldsymbol{chs} \leftarrow \{ch_{(1)}, ch_{(2)}, \ldots, ch_{(k)}\}$, such that $cost(ch_{(i)}) \geq cost(ch_{(j)})$ if $i < j$.
    Compute relative fitnes, $f_i \leftarrow \alpha \exp(-\beta g(ch_i))$.
    Compute probability from the fitness, $\boldsymbol{p} \leftarrow \{p_1, \ldots, p_k \mid p_i = f_i / \sum_{\text{all } i} f_i\}$.
    Fix $n_{keep}$ chromosomes to the new generation.
    for $i$ from $n_{keep}$ to $k$ do
        Select $ch_i, ch_j \in chs$ based on $\boldsymbol{p}$.
        $ch'_i, ch'_j \leftarrow$crossover$(ch_i, ch_j)$
        $chs \leftarrow chs \cup \{ch'_i, ch'_j\}$
    end for
    for $child$ from $n_{keep}$ to $k$ do
        Do the mutation on chromosome $ch_i$ (depend on the type we choose).
    end for
    Update the cost for the new generation chromosomes.
    if $time(now) - t_0 > timeLimit$ then
        break
    end if
end while
return The best chromosome decoded in to machine schedules $\{S_i\}_{i=1}^m$, corresponding makespan $\{C_i\}_{i=1}^m$, $time(now) - t_0$.
```

3.8 Description of algorithm and parameters

There are many steps and parameters in the algorithm. We will describe how every step actually work and parameters that involved. Note that there are many parameters in this algorithm and some might not use in the experiments, we will focus on some important parameters in the experiment.

The input is jobs and number of machine $\{x_1, \ldots, x_n, m\}$, first, the algorithm randomly generate the population. In this step we have to choose a parameter for the number of the population, k, the number of the chromosomes, and this will run in $O(n * k)$ because each chromosome is a list of size n. Now we can find the cost for each chromosome, intuitively, we put all the jobs in its corresponding machine, sum all the jobs in every machine, and find the maximum time, this will take $O(n*k)$ for putting jobs in a machine and $O(m)$ for finding the maximum value.

Now, we can start the while loop, the criteria we considered is number of reproduction and execution time, we simply set the maximum number of round for reproduction and time limited for program's execution, one might ignore one of the parameters here. In the while loop, we start by sorting the chromosomes, $O(k * log(k))$, then calculate the actual fitness function $f_i = \alpha \exp(-\beta g_i)$, where g_i is the cost of the chromosome i, This will take linear time to compute. There are two parameters involved in this step, and we set $\alpha = 1$ and $\beta = max\{x_i\}$ which is the maximum value of jobs. We can adjust these two parameters to different value but for convenient of the experiment and execution it will be fixed. Next, we can create the probability distribution by setting $p_i = f_i / \sum_{\text{all } i} f_i$, where p_i is the probability corresponding to the chromosome i, the time complexity of this step is linear of the number of the chromosomes as well.

In the reproduction set, we keep first n_{keep} chromosomes, which are the best chromosomes in the generation because they are already sorted, here we need to choose the parameter n_{keep}, later on these kept chromosome will not mutate therefore if we keep large amount of chromosomes it will put less chance for mutation but more change to remain good solution for the next generation. Next, we perform the crossover function, as describe in the previous section. First, we draw a pair of chromosomes from the probability $\boldsymbol{p} = p_1, \ldots, p_k$ and perform the crossover. There are two type of crossover, first the uniform crossover, as explain in previous part, we will fixed the probability of choosing parent, and this will take linear time of the size of chromosome to execute. The second is k-point crossover, the parameters is number of points to perform the crossing over, this will take linear time for each new chromosome to execute as well. In the experiment we will discuss more about the different number of crossover points and the results of the algorithm.

We have two type of mutation function, all gene mutation and controlled mutation. For the all gene mutation we have a fixed probability of mutation for each gene, so the algorithm will perform the mutation for each gene in the chromosomes which are not the kept chromosomes, this will take linear time in the size of gene, $O(k * n)$. The second mutation function is controlled mutation, the chromosome will mutated in a fixed percentage, also this can be perform in linear time. For the make-span scheduling problem, the mutation function has high effect on the result, it can improve the algorithm performance significantly if we choose a appropriate type of mutation and mutation rate, therefore our parameters here are the type of the mutation and the mutation rate/percentage.

After the mutation, the algorithm will update the cost again and add new chromosomes to the new generation, then perform the reproduction again until the time limit or the maximum number of reproduction have been reached. The output of the algorithm will be the best chromosome of the last generation which will be represented in a list of lists where each row indicates each machine and the elements in the row are the jobs in that machine, also the execution time will be show after that.

4 Numerical Experiment

In this section, we will present solutions collected from three different algorithms, greedy, genetic algorithm (GA) and a hybrid genetic algorithm scheme with knowledge-based operator (HGA). In the basic setup, all experiments are run on a Intel i7-9700 3.0 GHz CPU and all algorithms are assumed to be fixed with a time limit of 2 minutes for comparable results. Throughout this section, algorithms are evaluated based on their average execution (running) times, solution quality and relative gaps. In relative gaps, we refer to

$$GAP = \frac{OPT_x - OPT_y}{OPT_y} \times 100\%,$$

where OPT_x and OPT_y refers to the solutions of selected algorithm x and y, respectively. Moreover, running time for greedy heuristic is trivial and solution quality will be presented in terms of relative gaps between GA and HGA and will be interpreted later.

4.1 Parameter tunning experiments

In this section, we will explore the effects of parameters on the performance. Since GA and HGA are implemented under the same framework, results are collected from GA solutions and only for describing trends for tuning parameters that are efficient rather than a straight numerical comparisons, which will be covered in the next section In the first experiment, we observe the different population size and reproduction percentage. In fig. 2, we can see that the bigger population size does lead to better results, but at a size of 100, the result already provides a promising bounds, which will be our basic setup for later experiments. Another parameter we consider in this figure is the reproduction rate which is the percentage of population selected to reproduce offsprings (i.e. parents to population ratio). Clearly, a reproduction rate of 20% performs better than that of 10%. However, a further increase in the ratio to 50% does not affect the result by a significant amount so the ideal value for the reproduction rate is 20%.

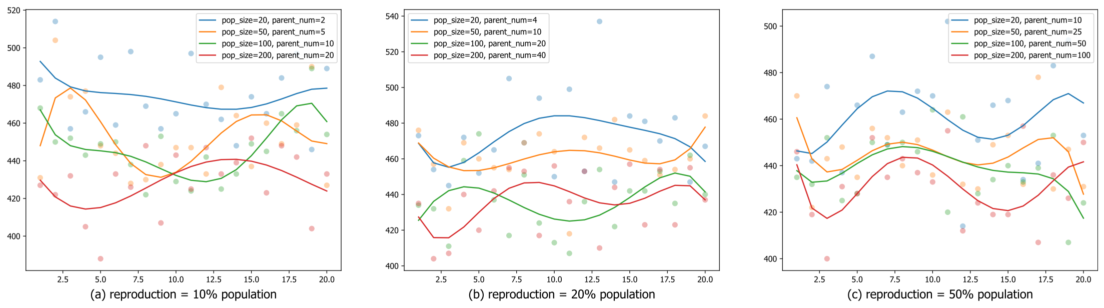

Figure 2: Performance profile of solutions tested on different population sizes and reproduction rate from instances of size $n = 100, m = 20$

The second parameter is type of crossing-over function, as shown in the figure 3, where we consider 1 point, 2 point, 3 point, and uniform crossover. As the results shows, they have slightly difference in the terms of both best and worst solution. In the case of best solution, the uniform crossover perform worse in most of the runs, although it is not significant but the 2 point crossover already achieved promising result in the best solution case. For the worst solution case, there are no difference between the result of different type crossover function. This is, however, results capped on a 2-minute computation, it might have slightly different results if we set the time limit higher, but we believe that a 2 point crossover should suffice for most test cases.

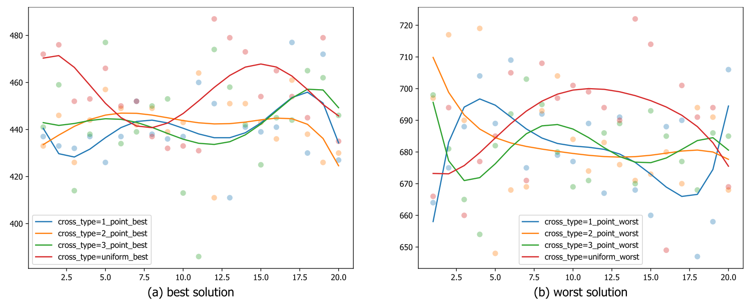

Figure 3: Performance profile of solutions tested on different crossover types from instances of size $n = 100, m = 20$

4.2 Algorithm performance experiments

We first report the running results of time and solutions for both GA and HGA. From table 1 we observe that the objective value computed by GA increases significantly as n increases and drops when m decreases, which is clearly as expected. This is accompanied by the running time results from table 3, where we observe that increases in both m and n results in significantly increase of computational time. A similar conclusion can also be drawn for HGA results as shown in table 2, where we observe that HGA achieves notably better solution when m increases, showing a better handling of large instances. Computational time of HGA, although scales with instance sizes, but are less predictable than GA. We argue that this is due to the non-deterministic nature of our proposed operator, where certain batch of populations are more prone to be modified and thus resulting in a longer execution time.

Intance sizes	$n = 50$	$n = 100$	$n = 200$	$n = 300$	$n = 400$	$n = 500$	$n = 600$	$n = 700$	$n = 800$	$n = 900$	$n = 1000$
$m = n/10$	638.8	715.3	789.3	828.8	845	903.6	920.3	928.5	947.7	946.5	952
$m = 2n/10$	361.5	430.9	492.7	515.1	538	567.4	574.4	587.6	597.4	603.8	619.7
$m = 3n/10$	299.4	340.1	379.9	405.9	424.5	449.2	445.6	467.9	474.4	488.2	498.5
$m = 4n/10$	244.5	284.7	327.7	349.9	373.6	385.5	394.7	399.2	409.5	418.5	424.8
$m = 5n/10$	219.7	259.2	290.7	318.7	328.9	344.7	356.5	367.4	371.3	377.4	371.6
$m = 6n/10$	202	238	267.9	292.3	303.3	317	328.5	335.2	333.8	344.9	348.3
$m = 7n/10$	194.6	222.6	257.2	270.2	285.5	298.3	306.1	307.2	322.4	323.2	328.9
$m = 8n/10$	179.2	213.6	243.6	256.6	274.3	279.9	280.9	286.8	298.6	308.5	309.9
$m = 9n/10$	170.4	203.5	230.9	244.8	260.4	263.5	268.9	284.4	286.3	294.5	295.6

Table 1: Performance of GA tested on varying instance sizes, averaged over 10 runs of same-size instances

We will then be looking closely at the performance profiles of these algorithms of selected instances size. As shown in fig. 4, GA and HGA do provide promising bounds over greedy solution. Specifically, HGA maintains and even closes the gap with greedy as m increases, while GA, on the other hand, fails to close such a gap.

The gap widen as we increases n, as shown in fig. 5, where HGA follows the same pattern as observed in profiles for $n = 50$.

Intance sizes	$n = 50$	$n = 100$	$n = 200$	$n = 300$	$n = 400$	$n = 500$	$n = 600$	$n = 700$	$n = 800$	$n = 900$	$n = 1000$
$m = n/10$	610.8	642.1	652	668	680	685	708	701	710	722	717
$m = 2n/10$	318.7	345.9	365.5	375	391	381	392	393	401.5	397	402.8
$m = 3n/10$	234	251.2	274.5	285.5	300.5	312.5	316	325	318	335	333.1
$m = 4n/10$	201	224.3	248.5	261	262.5	277	276.5	283.5	287	294.5	292.9
$m = 5n/10$	155.4	169.5	180	183	185.5	187	188.5	194	193	201	200.4
$m = 6n/10$	164.3	181.7	196.5	213.5	217	230	237.5	233	239.5	239	241.2
$m = 7n/10$	154.7	168	192.5	196.5	202.5	211.5	219.5	219.5	229	220.5	226.7
$m = 8n/10$	117.2	137.9	149.5	164	172.5	174	181.5	182.5	188.5	185	192.8
$m = 9n/10$	97.7	100.4	114	122.5	129.5	129	130	140	146	146	142.5

Table 2: Performance of HGA tested on varying instance sizes, averaged over 10 runs of same-size instances

Instance sizes	$n = 50$	$n = 100$	$n = 200$	$n = 300$	$n = 400$	$n = 500$	$n = 600$	$n = 700$	$n = 800$	$n = 900$	$n = 1000$
$m = n/10$	0.120	0.209	0.404	0.613	0.839	1.045	1.275	1.465	1.667	1.894	2.081
$m = 2n/10$	0.126	0.218	0.422	0.638	0.862	1.072	1.288	1.502	1.710	1.930	2.141
$m = 3n/10$	0.130	0.227	0.438	0.661	0.886	1.106	1.325	1.547	1.763	2.006	2.258
$m = 4n/10$	0.135	0.234	0.455	0.682	0.915	1.139	1.374	1.623	1.889	2.164	2.419
$m = 5n/10$	0.140	0.242	0.471	0.704	0.946	1.183	1.454	1.730	2.005	2.276	2.539
$m = 6n/10$	0.137	0.251	0.487	0.730	0.979	1.257	1.542	1.820	2.099	2.368	2.636
$m = 7n/10$	0.140	0.262	0.504	0.755	1.070	1.324	1.616	1.896	2.179	2.462	2.723
$m = 8n/10$	0.143	0.270	0.523	0.784	1.096	1.388	1.685	1.985	2.253	2.542	2.820
$m = 9n/10$	0.148	0.280	0.541	0.823	1.134	1.446	1.742	2.036	2.333	2.638	2.919

Table 3: Execution time (in seconds) of GA tested on varying instance sizes, averaged over 10 runs of same-size instances

Intance sizes	$n = 50$	$n = 100$	$n = 200$	$n = 300$	$n = 400$	$n = 500$	$n = 600$	$n = 700$	$n = 800$	$n = 900$	$n = 1000$
$m = n/10$	1.551	3.220	6.447	9.638	13.106	16.361	20.370	23.714	26.959	31.178	34.763
$m = 2n/10$	1.593	3.285	6.809	10.155	13.584	17.482	21.727	25.735	28.925	33.879	38.548
$m = 3n/10$	1.825	3.609	7.278	11.043	14.536	18.852	23.852	27.950	32.650	37.096	42.184
$m = 4n/10$	1.918	3.912	7.943	12.094	16.442	21.144	26.863	32.437	37.776	42.690	48.140
$m = 5n/10$	1.728	3.601	7.399	11.495	15.580	20.377	25.942	31.366	35.974	40.556	46.595
$m = 6n/10$	2.030	4.173	8.592	13.256	18.104	23.958	29.972	36.292	41.639	47.594	54.420
$m = 7n/10$	1.784	3.669	7.363	11.102	15.631	20.488	24.958	30.747	35.332	40.680	45.982
$m = 8n/10$	1.632	3.182	6.768	10.398	14.151	18.347	22.638	27.767	31.410	36.139	40.329
$m = 9n/10$	1.574	3.053	6.300	9.723	13.191	16.958	21.131	25.388	29.577	33.517	37.542

Table 4: Execution time (in seconds) of HGA tested on varying instance sizes, averaged over 10 runs of same-size instances

The gap further widens in both GA and HGA for $n = 1000$ as shown in fig. 6. However, we can see a noticeable increase in the gap between GA and HGA, identifying weaknesses of GA for increased instance sizes, where it performs no better than randomized search solutions.

Indeed, the trend described in these profiles can be identified with table 5 and table 6, where we observe that gaps between GA and greedy increases significantly as instance size increases in either m or n. For HGA, it scales well with instance sizes as it closes the gap significantly, compared with GA. However, there are few exceptions for $m = 6n/10$ and $m = 7n/10$. We argue that this is due to the poor determining mechanism in our proposed operator, where it fails to identifies a "good" gene that can swap with current

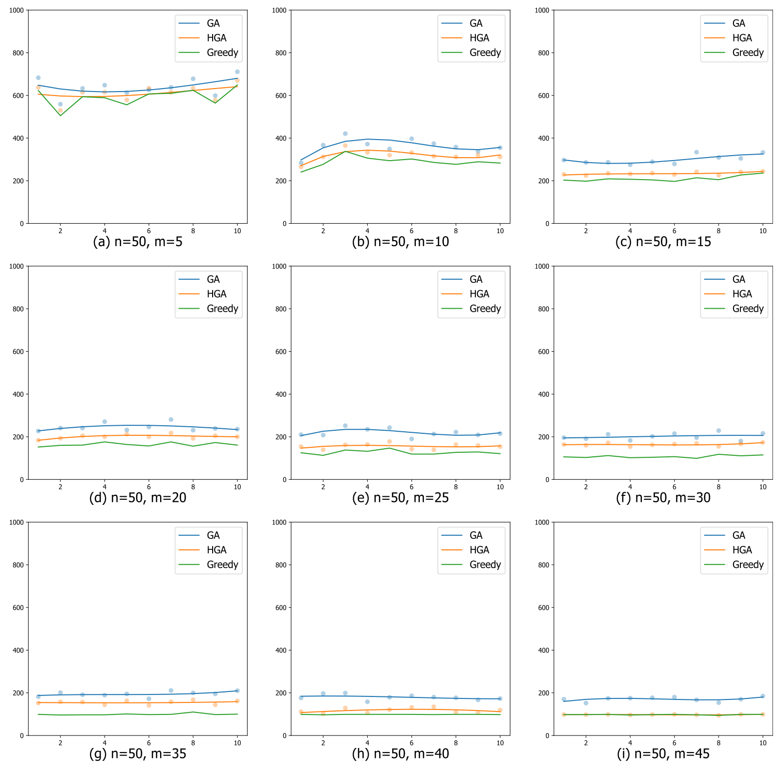

Figure 4: Performance profile of solutions from GA, greedy and HGA on instances of size $n = 50$ with varying $m = n/10, 2n/10, \cdots, 9n/10$.

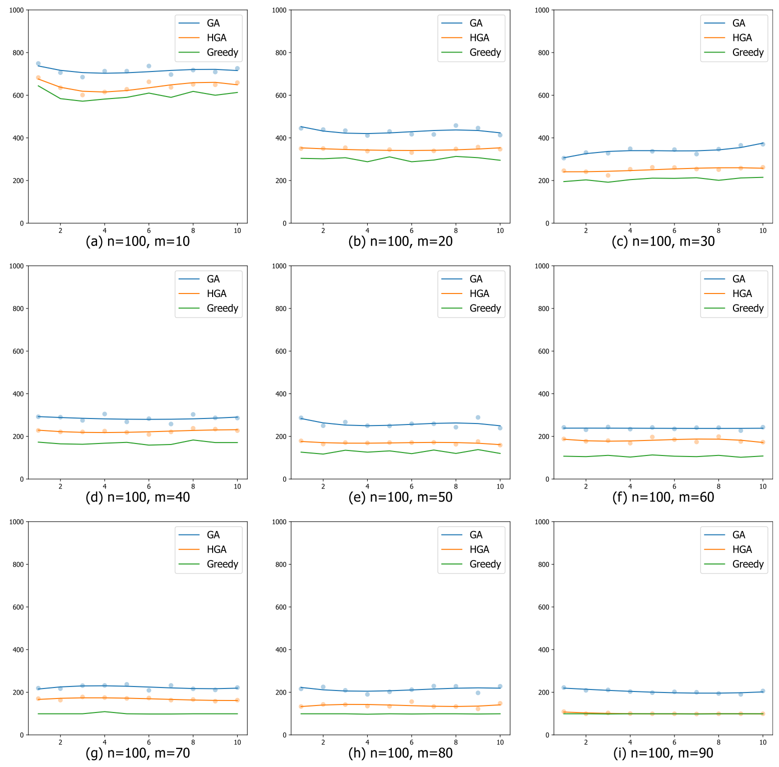

Figure 5: Performance profile of solutions from GA, greedy and HGA on instances of size $n = 100$ with varying $m = n/10, 2n/10, \cdots, 9n/10$.

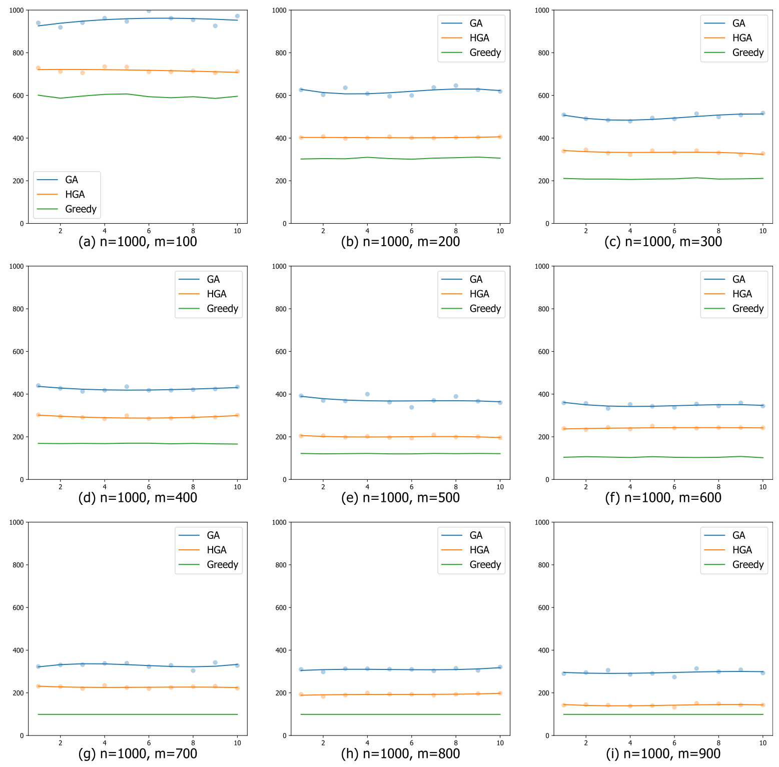

Figure 6: Performance profile of solutions from GA, greedy and HGA on instances of size $n = 1000$ with varying $m = n/10, 2n/10, \cdots, 9n/10$.

ones. This shortcoming is magnified for $m = 6n/10$ and $m = 7n/10$ because in instances of these sizes, the number of jobs at each station is between 1 and 2. Our proposed operator fails to keep these information in evolution of populations, results in a similar performance as GA gives.

Intance sizes	$n = 50$	$n = 100$	$n = 200$	$n = 300$	$n = 400$	$n = 500$	$n = 600$	$n = 700$	$n = 800$	$n = 900$	$n = 1000$
$m = n/10$	7.89	19.16	31.11	39.51	41.83	50.32	52.82	55.45	58.35	57.36	59.84
$m = 2n/10$	24.96	43.11	64.45	70.34	77.15	86.71	89.07	92.40	96.77	98.62	102.85
$m = 3n/10$	42.57	65.42	82.03	96.09	103.89	116.48	114.54	126.15	128.19	134.37	138.29
$m = 4n/10$	49.45	68.76	97.53	111.68	122.91	130.42	134.66	137.34	141.59	148.37	152.41
$m = 5n/10$	72.99	104.26	137.31	163.82	168.49	183.94	192.69	198.94	204.10	208.59	206.85
$m = 6n/10$	87.56	122.01	151.79	174.98	186.40	201.90	210.79	218.33	218.82	226.92	232.66
$m = 7n/10$	95.58	123.05	159.80	172.93	188.38	201.31	209.19	210.30	225.66	226.46	232.22
$m = 8n/10$	81.74	116.63	146.31	159.45	177.07	182.73	183.74	189.70	201.62	211.62	213.03
$m = 9n/10$	74.41	105.97	133.47	147.52	163.03	166.16	171.62	187.27	189.19	197.47	198.59

Table 5: Relative gap (in %) between GA and greedy heuristics tested on varying instance sizes, averaged over 10 runs of same-size instances

Intance sizes	$n = 50$	$n = 100$	$n = 200$	$n = 300$	$n = 400$	$n = 500$	$n = 600$	$n = 700$	$n = 800$	$n = 900$	$n = 1000$
$m = n/10$	3.16	6.96	8.31	12.44	14.13	13.96	17.57	17.36	18.63	20.03	20.38
$m = 2n/10$	10.16	14.88	22.00	24.01	28.75	25.37	29.03	28.68	32.25	30.59	31.85
$m = 3n/10$	11.43	22.18	31.53	37.92	44.33	50.60	52.14	57.08	52.96	60.83	59.23
$m = 4n/10$	22.86	32.96	49.79	57.89	56.62	65.57	64.39	68.55	69.32	74.78	74.03
$m = 5n/10$	22.36	33.57	46.94	51.49	51.43	54.04	54.76	57.85	58.07	64.35	65.48
$m = 6n/10$	52.55	69.50	84.68	100.85	104.91	119.05	124.69	121.27	128.75	126.54	130.37
$m = 7n/10$	55.48	68.34	94.44	98.48	104.55	113.64	121.72	121.72	131.31	122.73	128.99
$m = 8n/10$	18.86	39.86	51.16	65.82	74.24	75.76	83.33	84.34	90.40	86.87	94.75
$m = 9n/10$	0.00	1.62	15.27	23.86	30.81	30.30	31.31	41.41	47.47	47.47	43.94

Table 6: Relative gap (in %) between HGA and greedy tested on varying instance sizes, averaged over 10 runs of same-size instances

5 Conclusion and Discussion

In this report, we explored the genetic algorithm for makespan scheduling problem. Numerical experiments were conducted to compare results from different parameters and implementation. In particular, our HGA outperforms GA in most cases but takes significantly longer computational time compared with GA. Therefore, we believe that GA design must be accompanied with problem specific knowledge if we want a promising bound on the solution, otherwise it is only a adaptive randomized search, with a even higher computational costs that does not scale well with instance sizes. We also identifies that the failure of our GA, lies in its chromosome representation, where the permutation encoding is subject to the nature that it holds no spatial information (i.e. distribution of jobs) on the machine schedule. Specifically, we believe that crossover, or local exchange of information, also fails to capture and even breaks such a spatial information, not to mention that the mutation is inherently a randomized swap. On the other hand, although our adaptation of HGA fails to close the gap with greedy heuristics completely, it does provide thoughtful insights for a general operator design for the makespan scheduling problem.

References

[1] D. E. Akyol and G. M. Bayhan. A review on evolution of production scheduling with neural networks. *Computers & Industrial Engineering*, 53(1):95–122, 2007.

[2] J. W. Barnes and J. B. Chambers. Solving the job shop scheduling problem with tabu search. *IIE transactions*, 27(2):257–263, 1995.

[3] A. Colorni, M. Dorigo, V. Maniezzo, et al. Distributed optimization by ant colonies. In *Proceedings of the first European conference on artificial life*, volume 142, pages 134–142. Paris, France, 1991.

[4] L. Davis et al. Job shop scheduling with genetic algorithms. In *Proceedings of an international conference on genetic algorithms and their applications*, volume 140, 1985.

[5] A. Dias, J. Amaral, A. Teixeira, and C. Neto. Genetic algorithms in optimization: Better than random search? 07 2000.

[6] E. Falkenauer, S. Bouffouix, et al. A genetic algorithm for job shop. In *ICRA*, pages 824–829. Citeseer, 1991.

[7] B. Giffler and G. L. Thompson. Algorithms for solving production-scheduling problems. *Operations research*, 8(4):487–503, 1960.

[8] F. Glover. Tabu search—part i. *ORSA Journal on computing*, 1(3):190–206, 1989.

[9] S. W. Golomb and L. D. Baumert. Backtrack programming. *Journal of the ACM (JACM)*, 12(4):516–524, 1965.

[10] J. R. Jackson. Scheduling a production line to minimize maximum tardiness. *management science research project*, 1955.

[11] S. M. Johnson. Optimal two-and three-stage production schedules with setup times included. *Naval research logistics quarterly*, 1(1):61–68, 1954.

[12] A. Kaur, B. Khehra, and I. S. Virk. Makespan optimization in job shop scheduling problem using differential genetic algorithm. *International Journal of Computer Applications*, 172:30–36, 08 2017. doi:10.5120/ijca2017915218.

[13] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95-international conference on neural networks*, volume 4, pages 1942–1948. IEEE, 1995.

[14] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[15] A. M. Kuczapski, M. V. Micea, L. A. Maniu, and V. I. Cretu. Efficient generation of near optimal initial populations to enhance genetic algorithms for job-shop scheduling. *Information Technology and Control*, 39(1), 2010.

[16] Z. Lomnicki. A "branch-and-bound" algorithm for the exact solution of the three-machine scheduling problem. *Journal of the operational research society*, 16(1):89–100, 1965.

[17] A. S. Manne. On the job-shop scheduling problem. *Operations research*, 8(2):219–223, 1960.

[18] L. Min and W. Cheng. A genetic algorithm for minimizing the makespan in the case of scheduling identical parallel machines. *Artificial Intelligence in Engineering*, 13 (4):399–403, 1999.

[19] E. Nowicki and C. Smutnicki. A fast taboo search algorithm for the job shop problem. *Management science*, 42(6):797–813, 1996.

[20] S. S. Panwalkar and W. Iskander. A survey of scheduling rules. *Operations research*, 25(1):45–61, 1977.

[21] D. Pham, A. Ghanbarzadeh, E. Koc, S. Otri, S. Rahim, and M. Zaidi. The bees algorithm. *Technical Note, Manufacturing Engineering Centre, Cardiff University, UK*, 2005.

[22] H. Piroozfard, K. Y. Wong, and A. Hassan. A hybrid genetic algorithm with a knowledge-based operator for solving the job shop scheduling problems. *Journal of Optimization*, 2016:7319036, Apr 2016. ISSN 2356-752X. doi:10.1155/2016/7319036. URL https://doi.org/10.1155/2016/7319036.

[23] C. N. Potts and L. N. Van Wassenhove. A branch and bound algorithm for the total weighted tardiness problem. *Operations research*, 33(2):363–377, 1985.

[24] G. A. Rovithakis, V. I. Gaganis, S. E. Perrakis, and M. A. Christodoulou. Neuro schedulers for flexible manufacturing systems. *Computers in industry*, 39(3):209–217, 1999.

[25] S. C. Sarin, S. Ahn, and A. B. Bishop. An improved branching scheme for the branch and bound procedure of scheduling n jobs on m parallel machines to minimize total weighted flowtime. *The International Journal of Production Research*, 26(7):1183–1191, 1988.

[26] F. Y.-P. Simon et al. Stochastic neural networks for solving job-shop scheduling. i. problem representation. In *IEEE 1988 International Conference on Neural Networks*, pages 275–282. IEEE, 1988.

[27] W. E. Smith et al. Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, 3(1-2):59–66, 1956.

[28] R. Storn and K. Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4): 341–359, 1997.

[29] H. M. Wagner. An integer linear-programming model for machine scheduling. *Naval research logistics quarterly*, 6(2):131–140, 1959.

[30] S. Yang and D. Wang. Constraint satisfaction adaptive neural network and heuristics combined approaches for generalized job-shop scheduling. *IEEE Transactions on Neural Networks*, 11(2):474–486, 2000.

[31] X.-S. Yang. *Nature-inspired metaheuristic algorithms.* Luniver press, 2010.

[32] A.-H. Yin. A heuristic algorithm for the job shop scheduling problem. In *IFIP International Conference on Network and Parallel Computing*, pages 118–128. Springer, 2004.

[33] A.-H. Yin. A new neighborhood structure for the job shop scheduling problem. In *Proceedings of 2004 International Conference on Machine Learning and Cybernetics (IEEE Cat. No. 04EX826)*, volume 4, pages 2098–2101. IEEE, 2004.

[34] J. Zhang, G. Ding, Y. Zou, S. Qin, and J. Fu. Review of job shop scheduling research and its new perspectives under industry 4.0. *Journal of Intelligent Manufacturing*, 30(4):1809–1830, 2019.

[35] W. Zhang, J. Wen, Y. Zhu, and Y. Hu. Multi-objective scheduling simulation of flexible job-shop based on multi-population genetic algorithm. *International Journal of Simulation Modelling*, 16(2):313–321, 2017.