A Hybrid Genetic Algorithm using Wisdom of the Crowds

|  |  |  |  |
| --- | --- | --- | --- |
| Nikhil Thimmadasaiah  CSE  Speed School of Engineering University of Louisville, USA nwthim01@louisville.edu | Christopher Lyles  CSE  Speed School of Engineering University of Louisville, USA  cllyle02@louisville.edu | Renz Tolentino  CSE  Speed School of Engineering University of Louisville, USA rstole01@louisville.edu | Mason Mechlin  CSE  Speed School of Engineering University of Louisville, USA msmech01@louisville.edu |

*Abstract*—The goal of this paper is to showcase a hybrid genetic algorithm that uses a wisdom of the crowds approach to solved the Job shop scheduling problem. The performance of the hybrid algorithm will be compared to the performance of other approaches to the Job shop scheduling problem. This performance is compared to see if the Job shop scheduling problem can be solved with an even more optimal solution than a regular genetic algorithm can find. The main difference with the wisdom of crowds approach is that an aggregation method using the Borda count rank technique is used to find a more optimized solution. The genetic algorithm with wisdom of crowds approach outperformed the regular genetic algorithm in early generations. However, with even more generations the regular genetic algorithm started to outperform. The amount of generations that the regular genetic algorithm required to finally outperform the combined wisdom of crowds approach also depended on the complexity of the problem set. Overall, the wisdom of crowds approach improved much faster in finding a better solution but stagnated as well.

# INTRODUCTION

The Job Shop Scheduling problem (JSSP) is a variation of the job scheduling problem in which each job a set of operations that must be performed on a given machine for a fixed amount of time in a predefined order [1]. The first set of constraints, the precedence constraints, specify that no operation for a job may be executed before the finishing time of its successor in the predefined sequence.

Where is the start time of operation ‘l’ in job ‘i’ and is the same operation’s duration. Then there are capacity constraints which specify that no machine can be occupied by more than one job at a time.

These constraints, with no additions or relaxations, are

considered the constraints for the Classical JSSP [2]. The objective of the problem is generally to finish all jobs in as little time as possible, which often means minimizing the completion time of the last operation or the make span of the jobs. [20]

There are many variations on the constraints and objectives of JSSP [2]. There are variations of JSSP that allow for durations of tasks to change in runtime (Dynamic JSSP) or for changes in the order of tasks or a flexibility of machines on which a task can be performed (Flexible JSSP). There are also formulations for the objective of JSSP that assign deadlines to jobs and punish and/or reward solutions for jobs completed late or early respectively as well as objectives that seek to maximize the rate of machine utilization. Nevertheless, the classical JSSP with the objective of minimizing make span will be used to compare the approach outlined in this paper to other approaches as the classical JSSP is considered the standard for comparing approaches [2]. For the sake of brevity, classical JSSP will be referred to as simply JSSP from here after.

An NP-complete problem is a problem where every example problem can only be solved in non-deterministic polynomial time. It’s difficult to explain what NP is, but in the classification of P (Polynomial) problems vs NP (Non-deterministic polynomial) problems, P problems can be solved in polynomial time, while NP can be verified if a proposed solution is valid in polynomial time [15].

Because the JSSP is an NP complete [3] an approach that seeks an exact solution, i.e., solutions that are optimal for the objective are infeasible for problems of large size. Incomplete search techniques like the genetic search algorithm must be used, thus a genetic algorithm was implemented for this paper.

The genetic algorithm approach was formulated by Holland in his book Adaptation in Natural and Artificial Systems [4]. It has since been a popular method for tackling NP complete problems for its simplicity and aesthetic appeal. Its logic parallels that of the theory of natural selection, specifically for organisms that reproduce sexually. Organisms that are more fit for the environment are more likely to survive and pass on their characteristics to offspring. Eventually, the population becomes better adapted and the species evolves. Analogously, the solutions produced by the genetic algorithm are selected for reproduction with a probability that depends on the fitness of an individuals determined by a fitness function.

Common to most modern genetic algorithms are the mutation, crossover, and selection operators which are used to manipulate the population to allow for the evolution of a solution [5]. Mutation operators are analogous to mutations in the theory of evolution which allow for traits that are not present in a population to be spontaneously generated and facilitates adaptation by allowing new approaches to increasing fitness to be explored. If one is to think of a genetic algorithm in the terms of any other search algorithm, they might liken a mutation operation to searching novel paths. The crossover operator, on the other hand, combines fragments of solutions and develops better solutions based on what already exists in the population. Without the mutation operation, the genetic search would be stuck exploring extremely similar solutions, in other words, the search would reach a plateau or local optimum and would not be able to escape. For this reason, it is necessary to maintain diversity in the population of a genetic search to take advantage of the unique features of a genetic search approach. If there is too much mutation, however, then the genetic search will not be able to develop its population by disseminating successful schema as these schemata could be overwritten by mutations. The selection operator implements the natural selection part of the theory of evolution. The selection operator essentially assigns probabilities to each member of the population to be selected for crossover based on their fitness. It essentially determines the speed at which the population evolves.

The wisdom of the crowds phenomenon is the observation that the aggregate of solutions to problems provided by individuals in a group often perform better than the average performance of solutions of individuals of the group [6]. With this in mind, the genetic algorithm will additionally be augmented by a wisdom of the crowds approach analogous to that taken by [7] where the problem is instead the JSSP, and the crowd is the population of the genetic algorithm. Additionally, the wisdom of crowds component will experiment with the size of the expert population or the top percentile of the population performance wise. As an addendum not included in [7], each generation of this paper’s approach will produce an aggregate individual from the population that will be in the next generation. The results of the unaugmented as well as the augmented genetic algorithm will be compared to those of previous approaches to solving the problem.

# Prior Work

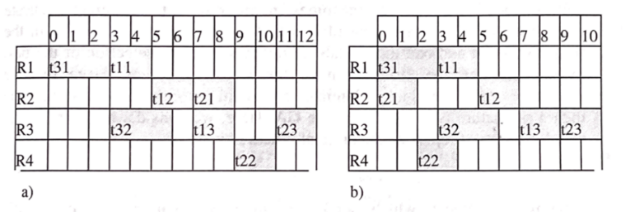
As mentioned before, the classical JSSP with the objective minimizing make span will be used to compare the results of this paper with those of previous papers. Conveniently, one of the most widely used approaches for JSSP is a genetic algorithm as noted by [8]. The same work also claims that the genetic algorithm is efficient for solving the JSSP as well. Stochastic methods are predominant; other such approaches outlined in [8] are ant colony and simulated annealing.

# Proposed Approach

1. *Representation*

Compared to most other combinatorial problems, the representation of a JSSP solution is particularly challenging to work with when it comes to implementing a genetic algorithm and requires that compromises are made. The precedence constraints, in particular, complicate matters in that scheduling operations on the machines randomly through generation of initial solutions, mutation, or crossover can easily lead to infeasible solutions. The simplest approach for encoding the schedules is the operation-based representation [9]. In this representation, each job has a symbol of its own and each task in the string is represented by the symbol of the job to which it corresponds. [19]

Take a problem with 3 jobs each having 3 tasks. The string “33111222” orders each task within each job implicitly. Since it is obviously not feasible to perform the second operation from job 2 before the first operation from job 2, one can simply assume that the second ‘2’ is the second task from job 2. With the string representation, or “genotype”, of a solution in mind, attention must turn to the solution that is to be constructed from the representation, also known as the “phenotype”. In this case there are two different representations of genotype.



Gantt charts provided by [10]

In solution a, the tasks are run in the order they appear in the string. In other words, the start time of any task is the latest finish time of all previous tasks (in the chromosome) that share a constraint. The chromosome is traversed from left to right. If the current task cannot be run the algorithm essentially “waits” until it can be run, paying no consideration to any tasks after it in the string representation.

What makes the other representation different is that if all resources can be occupied by the tasks in the schedule, then tasks in the chromosome that cannot be run at a given time will be skipped and the leftmost tasks that can run will be run before those that are skipped. When tasks are finished, and resources are opened, the string is essentially searched from left to right once again for tasks that can be run. Under the lens of using function b to construct solutions out of a chromosome, the order of the tasks in the chromosome is simply an implicit assignment of priority and not an ordering as one would observe them in the representative Gantt chart. Although this always results in a fitness that is better than that provided by the other fitness function, this would also mean that an algorithm of time complexity is run to assign the fitness of each individual. The former fitness algorithm, on the other hand, is of time complexity , making it more feasible for larger problems.

At any rate, the latter fitness function can at least be used as the result to boost the viability of the final solution.

1. *Fitness Function*

As mentioned before, the objective of the JSSP, for the purposes of this paper, is the make span of the solutions. To be more specific the fitness function used between generations will be the make span of solution a from a given string representation.

1. *Crossover Method*

The crossover method, partially mapped crossover (PMX), is defined in [11] but for more clarity, this paper will provide an original review of it. It is also worth keeping in mind that when the representation strings are sent to the crossover method, they are adapted in that each operation is given a unique symbol. At the end the string that results from the crossover method is converted back.

Take two parents P1 and P2. Both are permutations of the same set of characters. Take a random substring from P1 called S1. The substring in the same position in P2 will be referred to as S2. There are two sets of characters to consider:

* All characters in S2 that are not in S1. We will refer to this set as N.
* All characters in S1 that are not in S2. We will refer to this set as M.

For each character in N, we must find the position in the string P2 of a character in M and swap them. If we update S2 and P2 every time we do such a swap, we should eventually get to a point where the set of characters in S2 is the same as that of S1. Once we have achieved this, we take the substrings of P2 that are not S2 as well as S1 and place them in the same positions to make a child. As an addition to this algorithm, two children will be produced for every crossover, swapping the roles of the input parents in the context of the above algorithm.

1. *Mutation Method*

Also defined in [11], the mutation method, known as “Rotation to the Right”. The mutation method used is referred to as rotation to the right. A random substring in a string is chosen and the last character in that substring is sent to the leftmost position of the substring and all characters before it are moved forward by one position. The random substring is chosen by randomly generating two independent integers. If the second generated integer is less than the first, the substring wraps around the string.

1. *Selection Method*

The selection method belongs to a class of selection methods known as rank-based selection methods. These methods assign probabilities of selection to individuals based not on their fitness directly but based on their rank in relation to individuals. That is to say, for any function that determines the probability of selection for an individual, a rank-based selection method uses rank as input rather than fitness or some function of fitness. [12] makes the argument that rank based methods prevent “super individuals” or individuals with outlier fitness values from causing stagnation at local optimal these methods preserve diversity. The probability of selecting any individual in the population for crossover is linearly distributed by the rank of the individual.

As with any probability space, the sum of all these probabilities should be 1, so the sum of these probabilities of k for N individuals equaling 1 provides one constraint. But this is only one constraint while two variables need to be fixed. The selection pressure is therefore defined.

The selection pressure will be the quotient of the probability that the most fit individual is selected over the probability that the median individual is selected.

After these parameters are set, one can solve for [alpha] and [beta] and derive the cumulative probability function in terms of the rank. Reference [13] shows the math for deriving this cumulative probability function. Once the cumulative distribution is found and inverted, the distribution can be sampled using the inverse transform method. Set the cumulative probability function to a randomly generated variable ‘r’ in the range [0,1] and find the rank ‘k’ of the selected individual.

1. Aggregation Method

The aggregation method is based on the Borda count. Though the Borda count method is used for ranked choice voting in many democratic systems [14], it is adapted as the aggregation method for this paper. In ranked choice voting, ballots are orderings all candidates by the voter’s preference. Depending on where each candidate is ranked in a given ballot, they are assigned points. Higher rankings mean the ballot assigns more points to the candidate. A final ranking of the candidates comes from order them by how many points they receive from all ballots. If the operations are candidates, then each individual in the population that orders each operation chronologically is a ballot. The preference each individual gives the operations is the order the operations appear in the representative string.

The aggregation method gives a linear score to every task according to the order in which they appear in chromosomes of the final population. If n is the length of the chromosome for the population and m is the position of the task in the chromosome of a given individual then, for each member of the population, each task has n-m added to it, thus assigning more points to tasks that appear closer to the beginning of the chromosome. After the scores have been added up for each task, the tasks are sorted and the ranking that results from this becomes the chromosome of the aggregate individual. The fitness of this aggregate individual is calculated using scheduling method b.

# Data

Job scheduling data was created by a problem generator script that takes input for the number of resources and range of task counts and duration. From this it randomly generates a dataset of jobs and writes it into a file. The first number represents the number of machines available, as well as the maximum number of tasks possible. The first number in each line afterwards shows the arrival time of the job to the scheduler, that is, when the scheduler is informed of the job essentially. Next, look at the numbers separated by commas. For example, the second and third numbers in each line, separated by the comma. The first of these two numbers represents the duration of the task, while the second represents the specific numbered resource it requires.

EXAMPLE JSH FILE: Problem 1

RESOURCES:

4

JOBS:

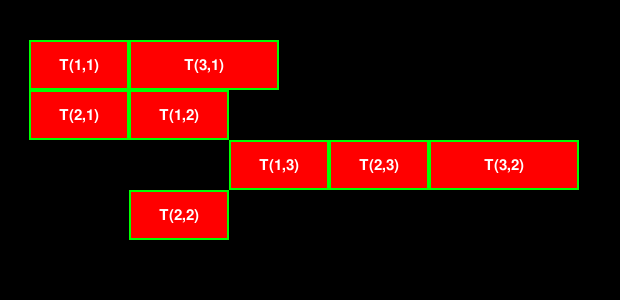
0 2,1 2,2 2,3

0 2,2 2,4 2,3

0 3,1 3,3

# Experimental Results

In this subsection the results of our Genetic Algorithm are presented on three separate Gantt charts showcasing the three different problem sets that were used in the experiment. These Gantt charts are made utilizing Pygame from Python so that the charts are updated without user input [16]. The following are the example solutions. The X-axis represents the time scale, and the y-axis represents the resource that is being used by the given tasks. With the resources changing as you increase or decrease the y values. When looking at the figures, it should be noted that the Job number and Task number are present. These are shown with the labels inside the Gantt Chart. The Job number is the first number in the ordered pair, and the task number is the second number. These charts also correctly follow the constraints that are generated from Job Shop Scheduling Problems [18]. The problem sets for the experiment were randomly generated [17] to ensure that full understanding of the problem was tested.



*Figure 5.1 Problem One Solution*

The following two charts show the solutions to higher complexity problems while keeping the constraints that were verified to exist on the smaller data set.

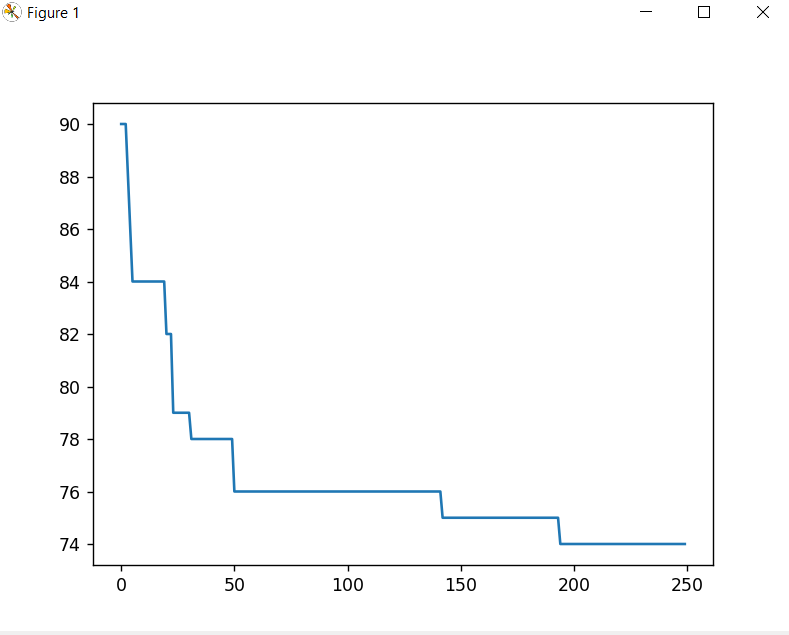


*Figure 5.2 Problem Two Solution*

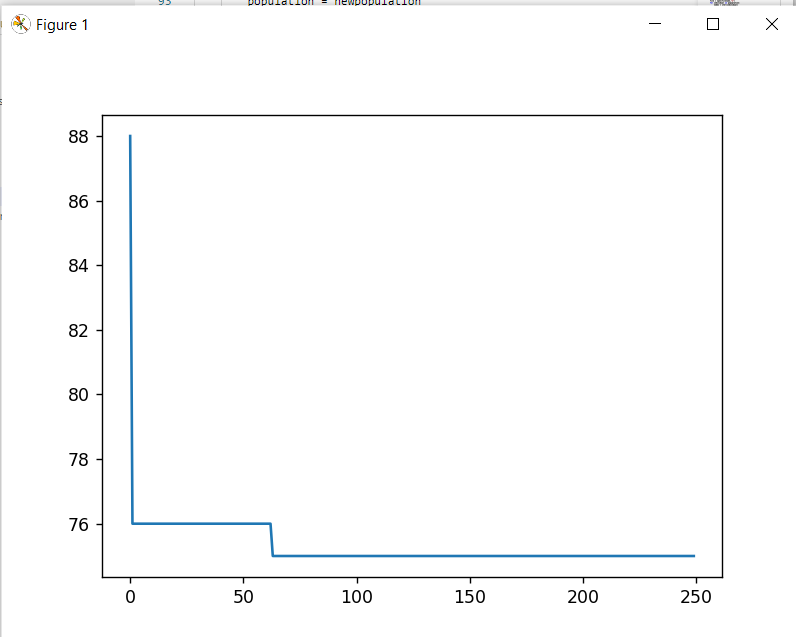


*Figure 5.3 Problem Three Solution*

When running our experiments, the time of execution was collected. When the experiment was looping through the different generations it would output the current generation and the best in the population from that number of generations. These were then input into a step graph to show how the different versions of the Genetic Algorithm performed. These show that the Genetic Algorithm was able to jump out of its local maximum [20] from performing the correct mutations to keep the trend.

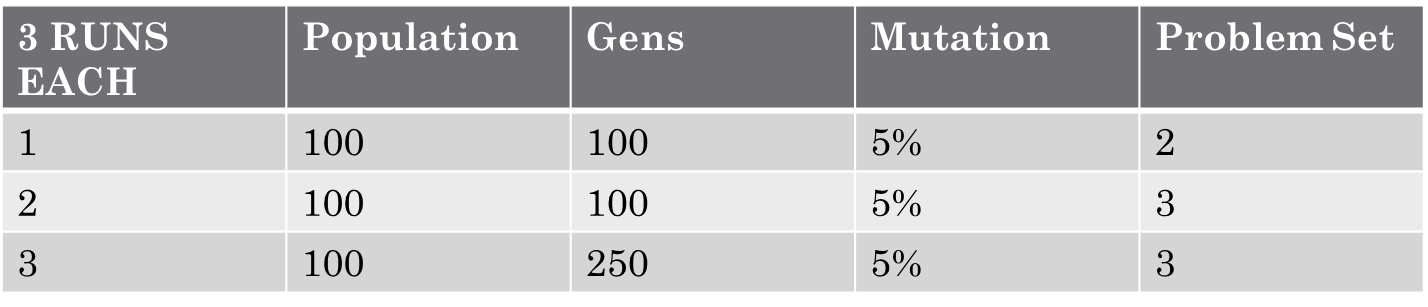


*Figure 5.4 Graph showing the shortest possible time generated by 250 generations using only Genetic Algorithms*



*Figure 5.5 Graph showing the shortest possible time generated by 250 generations from Genetic Algorithms with Wisdom of Crowds*

The experiment was run with three separate trials with different parameters or data sets for each trial.



*Figure 5.6 Table showing how each experiment trial was ran*

Comparing the time results from the three separate trials reveals that the Genetic Algorithm without Wisdom of Crowds ran faster [17].

# Conclusions

The wisdom of the crowds method started out making its improvements in early generations then started to stagnate. The regular generic algorithm had slower improvement but continued to improve. This can be seen in the difference between 100 generations vs 250 generations with Problem 3. Initially, with 100 generations, WOC gave a better solution. However, with more generations, NON-WOC outperformed. Additionally, typically the NON-WOC outperformed on time.

Problem 3 was more complex than Problem 2. According to the results, GA outperformed with a less complex problem even with 100 generations. A more complex problem in Problem 3 needed more generations for NON-WOC to outperform WOC. Thus, a bigger problem takes more generations for the regular NON-WOC to solve better, but no matter what size the problem, the WOC finds a very close approximate solution.

To improve the WOC solution, more settings need to be tested to see why the solution solves quickly but stagnates. An example would be experimenting with more mutation rates.

Future plans are to research into how to get the wisdom of crowds can stop stagnating. The solution improves quickly and its only obstacle is its current state of stagnation which the genetic algorithm alone doesn’t have.

Acknowledgment

No conflicts of interest.

References

1. R.L. Graham, E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey, Annals of Discrete Mathematics, Volume 5, Elsevier, 1979, Pages 287-326
2. Hegen Xiong, Shuangyuan Shi, Danni Ren, Jinjin Hu, A survey of job shop scheduling problem: The types and models, Computers and Operations Research, Elsevier, 2022, page 4
3. D. Applegate and W. Cook. "A computational study of the job-shop scheduling problem," ORSA J. on Comput, 1991.
4. J.H. Holland, Adaptation in Natural and Artificial Systems, University of Michigan Press, Ann Arbor, 1975
5. Russel, Stuart, Norvig, Peter, Artificial Intelligence a Modern Approach, 3rd ed, Pearson Education Inc., 2010.
6. Surowiecki, J., The Wisdom of Crowds, W.W. Norton & Company, Inc, 2004
7. Yi, Sheng Kung Michael, et al. "Wisdom of the crowds in traveling salesman problems." *Memory & Cognition* 39 (2011): 914-992.
8. C. Cebi, E. Atac and O. K. Sahingoz, "Job Shop Scheduling Problem and Solution Algorithms: A Review," 2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT), Kharagpur, India, 2020, pages 1-7
9. Cheng, Runwei, Gen, Mitsuo, Tsujimura, Yasuhiro, A Tutorial Survey of Job-Shop Scheduling Problems Using Genetic Algorithms – I. Representation, Computers ind. Engineering, Vol. 30, No. 4, Elsevier, 1996
10. Varela, Ramiro, Vela, Camino R., Puente, Jorge, Gomez, Alberto, Vidal, Ana M., Solving Job Shop Scheduling Problems by Means of Genetic Algorithms, Genetic Algorithms: Principles and Perspectives: A Guide to GA Theory. Kluwer Academic, 2003, Pages 276-279.
11. Eiben, A.E., Smith, J.E., “Representation, Mutation, and Recombination”, Introduction to Evolutionary Computing, Springer, 2015
12. Whitley, Darrell, Schaffer, J.D., "The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best", Proceedings of the Third International Conference on Genetic Algorithms (ICGA), Morgan Kaufmann Publishers Inc., 1989, pp. 116–121.
13. Reeves, Colin R, et al. Genetic Algorithms: Principles and Perspectives: A Guide to Ga Theory. Kluwer Academic, 2003, pp. 33-34.
14. Emerson, P. The original Borda count and partial voting, Social Choice and Welfare 40, Springer 2013, 353–358.
15. Goldreich, Oded. P, NP, and NP-Completeness: The basics of computational complexity. Cambridge University Press, 2010.
16. R. Kumari and C. Fancy, “Analyzing the PyGameGUI modules available in Python,” *2017 International Conference on IoT and Application (ICIOT)*, 2017. doi:10.1109/iciota.2017.8073603
17. B. Chen, B. Chen, H. Liu and X. Zhang, "A Fast Parallel Genetic Algorithm for Graph Coloring Problem Based on CUDA," 2015 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, Xi'an, China, 2015, pp. 145-148, doi: 10.1109/CyberC.2015.33.
18. M. A. Awad and H. M. Abd-Elaziz, "An Efficient Modified Genetic Algorithm For Integrated Process Planning-Job Scheduling," 2021 International Mobile, Intelligent, and Ubiquitous Computing Conference (MIUCC), Cairo, Egypt, 2021, pp. 319-323, doi: 10.1109/MIUCC52538.2021.9447610.
19. L. Nie, L. Gao, P. Li and L. Zhang, "Application of gene expression programming on dynamic job shop scheduling problem," Proceedings of the 2011 15th International Conference on Computer Supported Cooperative Work in Design (CSCWD), Laussane, Switzerland, 2011, pp. 291-295, doi: 10.1109/CSCWD.2011.5960088.
20. S. Xinyi, W. Aimin, G. Yan and Y. Jieran, "Job Shop Scheduling Problem with Job Sizes and Inventories," 2020 IEEE 11th International Conference on Mechanical and Intelligent Manufacturing Technologies (ICMIMT), Cape town, South Africa, 2020, pp. 202-206, doi: 10.1109/ICMIMT49010.2020.9041174.