Journal of the Korean Institute of Industrial Engineers
Vol. 51, No. 1, pp. 1-10, February 2025.
ISSN 1225-0988 | EISSN 2234-6457
https://doi.org/10.7232/JKIIE.2025.51.1.001

잡샵 일정계획 문제의 최적화를 위한 유전 알고리즘의 탐색 성능 향상 기법

백지원[1] · 우종훈[1,2†]

[1]서울대학교 조선해양공학과 / [2]서울대학교 해양시스템공학연구소

Enhancing Search Efficiency of Evolutionary Algorithm for Job Shop Scheduling Problem via Predetermined Sub-optimal Solutions

Jiwon Baek[1] · Jong Hun Woo[1,2]

[1]Department of Naval Architecture and Ocean Engineering, Seoul National University
[2]Research Institute of Marine Systems Engineering, Seoul National University

This paper introduces a novel evolutionary algorithm for the Job Shop Scheduling Problem (JSSP) using Genetic Algorithm (GA). The methodology incorporates a new fitness function that aligns with the intrinsic nature of JSSP, improving search efficiency for makespan optimization. The study proposes three enhancement methods for GA, based on analyzing Machine Input Order (MIO) Score derived from Machine-Job Sequence (MJS) and Machine-Operation Sequence (MOS). These methods intuitively leverage JSSP's structure, initiating searches from near-optimal solutions regardless of problem size. This approach is expected to significantly improve GA's efficiency, especially for larger problems.

Keywords: Scheduling, Genetic Algorithm, Job Shop, Optimization

1. Introduction

Scheduling plays a critical role in various domains, ranging from manufacturing to transportation and service industries. Efficient scheduling ensures the optimal utilization of resources, minimizes costs, and maximizes productivity, making it a fundamental aspect of modern-day operations management. Many scheduling environments where there exists distinguished type of work, or jobs, requiring subsequent operations in a predetermined order, with each operations needs to be allocated to the limited number of resources, can be modeled as Job Shop Scheduling Problem (JSSP).

JSSP is a combinatorial optimization problem that involves scheduling a set of jobs on a set of machines, where each job consists of a sequence of operations that must be processed on specific machines, subject to various constraints such as precedence and resource availability. Due to its complexity, JSSP is recognized as NP-hard, indicating that finding an optimal solution becomes increasingly difficult as the problem size grows. During the last decades, JSSP has been extensively studied, and the works varies from the one in the field of conventional optimization technique to those in the field of newly arising, metaheuristic approaches and AI-based techniques.

Although conventional optimization techniques can suggest mathematical insights to the problem, such as optimal value, or

이 논문은 서울대학교 해양시스템공학연구소의 지원을 받아 수행되었음.
† 연락저자 : 우종훈 교수, 08826 서울시 관악구 관악로1 서울대학교 34동 410호, Tel: 02)880-7330, E-mail: j.woo@snu.ac.kr
2024년 7월 8일 접수; 2024년 8월 9일 수정본 접수; 2024년 9월 4일 게재 확정.

the lower (or upper) bound of the specific object function such as makespan, these approaches are limited in that they cannot efficiently solve the problem with the increased complexity. To overcome this, a number of studies have been conducted to apply metaheuristic approaches on JSSP. Genetic Algorithm (GA) is a well known metaheuristic strategy for solving JSSP. Since suggested by Holland (1975), GA showed outperforming result compared to the traditional heuristic approaches when applied to the various problems in the field of combinatorial optimization.

While GA and other metaheuristic approaches are not optimization method and therefore does not guarantee the optimality of the solution, these techniques are still useful in that they provide efficient, stable strategies for near-optimal solutions (Banu Çalış and Serol Bulkan, 2012). The first attempt to adopt GA for JSSP was by Davis (1985). Since then, various research efforts have applied GA to JSSP. Nakano and Yamada (1991) used a conventional GA for binary encoding of JSSP, employing methods like binary representations to design chromosomes and reduce solution space. Zhou and Feng (2001) addressed JSSP with n jobs and m machines by encoding only the initial job assignment on m machines and assigning the remaining jobs based on priority rules. They applied GA and later adopted neighborhood search techniques to improve solution quality.

Mattfeld and Bierwirth (2004) encoded operation priorities in permutation form, where machines select jobs based on priority at each decision timestep. The resulting schedule is a non-delay schedule. To create an optimal active schedule, they introduced a look-ahead parameter, allowing the machine to remain idle until a specific job arrives instead of selecting an immediate job. Goncalves *et al.* (2005) also used priority rule-based encoding, which required a separate decoding procedure. To achieve optimal solutions, they introduced the concept of a parameterized active schedule, allowing delay times between two consecutive jobs on a machine.

A more general encoding technique, commonly referred to as the repetitive permutation encoding, was initially researched by Gen *et al.* (1994) and Bierwirth (1995). This method encodes JSSP solutions by repeating numbers from 1 to n, m times. Cheng *et al.* (1996) highlighted this technique as a representative encoding method in their survey paper on applying GA to JSSP.

Subsequent efforts to optimize JSSP makespan using GA with repetitive permutation encoding focused on two main directions: The first direction focused on identifying characteristics of superior candidate solutions to search in areas of the solution space that are most likely to contain optimal solutions. Someya and Yamamura (1999) introduced a search area adaptation procedure to facilitate this process, which was further developed by Watanabe *et al.* (2005). The second direction aimed to resolve the early convergence issues of traditional GA by integrating other local search heuristics or metaheuristic approaches. Zhang (2005) suggested a hybrid GA combined with Simulated Annealing (SA), generating neighborhood solutions based on critical paths of good solutions and performing local searches. Tamilarasi (2010) also proposed a GA combined with SA, adjusting parameters using the temperature concept of simulated annealing. Park *et al.* (2003) introduced Parallel GA (PGA), dividing the population into subpopulations for independent evolution and diversity maintenance. Over the past decades, extensive research has continued, considering various objectives beyond makespan minimization, such as minimizing setup or tardiness and optimizing resource utilization.

This paper attempts to extend the domain of GA-based methodologies by suggesting a new fitness calculation method for the evolutionary phase of the algorithm. The proposed approach enhances search efficiency and serves as an effective initialization method, compatible with any problem size. Firstly, this paper will take a brief look at the structure of JSSP and the application process of GA to this problem. Then the new fitness function, named as 'MIO score', will be introduced, along with the mathematical characteristics of the problem that enables the use of this fitness function efficient. Various methods of implementing 'MIO score' in the both initialization phase and evolutionary phase of the GA algorithm will be suggested. Derived from the observation on the graph structure of JSSP, the suggested function is expected to be an efficient metric that enables efficient initialization and reproduction in evolutionary algorithms for JSSP.

2. Job Shop Scheduling Problem

The basic form of JSSP is defined as follows. There exist N jobs $J=\{J_1, J_2, \cdots, J_N\}$ and M machines, where $M=\{m_1, m_2, \cdots, m_m\}$, and each job J_i comprises M operations, denoted as $O_i=\{O_{1i}, O_{2i}, \cdots, O_{Mi}\}$. Each operation occupies one of the M machines for a duration of p_{ij} time units and must be executed in a predetermined sequence. Additionally, each job can only have one operation in progress on a machine at any given time, and each machine can process at most one operation concurrently, subject to these constraints.

The solution to JSSP entails determining a schedule that satisfies these constraints, which may be reduced to the problem of determining the start times for all operations, or determining the sequence of jobs (or operations) to be processed on each machine. Typically, the objective of JSSP optimization is to minimize the time required for all jobs to be completed, or makespan.

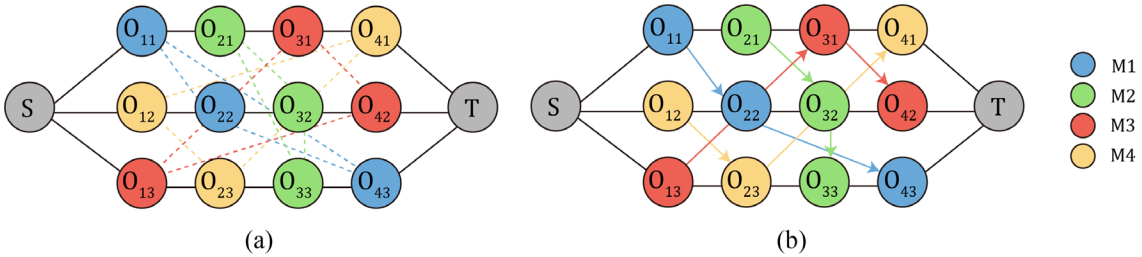

Figure 1. Disjunctive Graph Model of JSSP

Table 1. Notations

i	Index of jobs, $i=1, \cdots, N$
j	Index of operations, $j=1, \cdots, M$
k	Index of machines, $k=1, 2, \cdots, M$
l	Index, or the work order of the operation within the set of operations requiring same machine
J	Set of Jobs, $J=\{J_1, \cdots, J_N\}$
O_j	Set of operations comprising Job J_i
o_{ji}	Operation processed as jth order on Job J_i
p_{ji}	Processing time of o_{ji}
M	Machine set, $\{m_1, \cdots, m_M\}$
v_k	Set of operations requiring Machine m_k
v_{lk}	Operation processed as lth order on machine m_k
D_k	Clique, or undirected graph consisting v_k
D'_k	Directed path of disjunctive graph model consisting v_k

JSSP can be modeled as a graph structured problem, namely the classical disjunctive graph $G=\{V, C\cup D\}$. An example of the disjunctive graph representation of a simple job shop problem is suggested in <Figure 1>. Here, V represents the set of vertices, or operations of JSSP. C refers to the path, or the conjunctive arcs that connect the operations of the same job according to the precedence order, and D indicates a set of disjunctive arcs between operations required to be processed on the same machine. Determining a JSSP schedule is equal to establishing the direction of edges in the disjunctive graph (<Figure 1(a)>). Here, a directed path within operations processed on the same machine is created, thus indicating their working order, as in <Figure 1(b)>. In this paper, the notations and indices for the elements of JSSP follow the conventional notation suggested by Pinedo (2012) and are shown in <Table 1>.

3. GA for JSSP

GA is an evolutionary algorithm inspired by the process of natural selection and genetics. The optimization processed is handled by maintaining a population of potential solutions, known as chromosomes, and iteratively applying genetic operators such as selection, crossover, and mutation to evolve better solutions over successive generations.

In order to apply GA to JSSP, it is necessary to transform the problem into a format suitable for genetic operators. This involves encoding the solutions into chromosomes that can undergo genetic operations. One widely used encoding method is the repeated permutation encoding proposed by Gen *et al.* (1994). This encoding represents solutions to JSSP as repeated permutations of integers ranging from 1 to N, with total length as $N\times$ M, where each integer corresponds to a job. Upon decoding, each job number indicates the assignment of an operation to a corresponding machine at the respective time step. Each repeated permutation uniquely represents a solution to JSSP, although the reverse is not necessarily true. Therefore, the repeated permutation encoding method proposed by Gen *et al.* (1994) can be classified as a many-to-1 encoding technique. This encoding method offers the advantage of addressing some infeasibility issues that may arise when considering the solution space of JSSP as permutations of operations.

Executing random initialization as a part of evolutionary algorithm can lead to infeasibility problems when the chromosomes are represented after the permutation of the operation number. This is due to the existence of precedence constraints between operations. Representing arbitrary solutions with permutations of operations can be interpreted as adding new directed edges that connect operations to the existing graph structure. However, if

these added directed edges combine with the existing structure to form cyclic paths, infeasiblity problem may arise. To deal with these constraints, masking operations is one of the most widely used practice. When the operations that belong to the same job are masked to a same number, the schedule can be always decoded in a manner that always satisfies the precedence constraint, thereby avoiding the creation of such cyclic paths and ensuring the feasibility of solutions.

In practical, permutations ranging from 1 to $N\times M$ (the number of operations) are utilized as chromosomes in order to apply genetic operators with ease, when combined with appropriate masking strategy i.e. masking M numbers as a single integer.

3.1 Selection

In the selection phase of GA, individuals with higher fitness values are more likely to be chosen for reproduction, mimicking the process of natural selection. In the context of JSSP selecting individuals with shorter makespans increases the likelihood of generating offspring with improved scheduling solutions.

3.2 Evaluation

In GA, evaluation involves assessing the fitness of each individual chromosome within the population. This fitness evaluation typically measures how well a chromosome's solution performs with respect to the problem's objective function. In the context of JSSP, evaluation refers to calculating the makespan associated with each chromosome's schedule, with the aim of identifying chromosomes that represent more efficient scheduling solutions.

3.3 Modification

Two types of modification strategy, the crossover and the mutation, are the most frequently used. Crossover involves combining genetic information from two parent chromosomes to produce offspring chromosomes with potentially superior characteristics. However, if a certain crossover method result in violating the making heuristic of individual chromosomes, the feasibility of the solution cannot be guaranteed. That is, a repeated permutation that does not repeat the job numbers for exact m times respectively, should be avoided. Therefore, crossover operators such as Order Crossover (OX) or Partially Mapped Crossover (PMX) are commonly employed to preserve the overall composition of the chromosome. Mutation introduces random changes to individual chromosomes, promoting diversity within the population and preventing premature convergence to suboptimal solutions. When applied to JSSP, mutation operations enable exploration of new scheduling configurations, potentially leading to further reductions in makespan and enhancement of solution quality.

3.4 Reproduction

During the reproduction phase, genetic operators such as crossover and mutation are applied to selected parent chromosomes, resulting in the generation of two new offspring chromosomes. These offspring chromosomes are incorporated into a new temporary population. This process is iterated until a new temporary population of predetermined size is formed. Selected chromosomes constitute the new population, initiating the next iteration.

4. GA with MIO Score

One drawback of GA is their susceptibility to being influenced by the initial solutions and getting trapped in local optima, making it challenging to find global optimal solutions. To address this issue, this study introduces a novel methodology called 'MIO score'. This methodology originates from the simple principle: "If a machine has to perform multiple operations, wouldn't it be more efficient to prioritize them in the order of their urgency?"

Imagine someone should handle tasks for multiple individuals; one will naturally prioritize tasks based on their customer's urgency. In JSSP, this "urgency" corresponds to the number of remaining operations ahead, and the preceding operations are considered more urgent compared to those at the behind. Thus, we derive the following research hypothesis: If we prioritize the operations within the operations that shares same machine, based on their job sequence numbers, solutions obtained in this manner (called 'MIO solutions') may serve as relatively close approximations to optimal solutions, thereby enhancing search performance. Although MIO solutions might not be optimal, using MIO solutions as guidance towards the optimal solution in the initial phase of the search is expected to be significantly enhance the search speed.

4.1 Definition of MIO Score

Let D_k be the set of vertices $\{v_{1k}, v_{2k}, \cdots, v_{Nk}\}$ where each v_{ik} refers to the arbitrary i-th operation processed on machine m_k. The schedule of m_k, denoted as D'_k is the directed path within the elements of D_k. To implement the proposed method, this paper suggests three concepts: First, a Machine-Job Sequence (MJS) of machine m_k indicates the sequence of job index of the operations in

D'_k. Second, a Machine-Operation Sequence (MOS) of machine m_k indicates the sequence of relative positions within the job for each operations in D'_k. The two type of indices can be clearly distinguished by incorporating the two functions $f_o(v_{lk})$ and $f_j(v_{lk})$, where $f_o(v_{lk})$ returns the operational index of an arbitrary operation v_{lk} and $f_j(v_{lk})$ for the job index of v_{lk}.

For example, the disjunctive graph of JSSP suggested in <Figure 2(a)> is comprised of $\{D_1, D_2, D_3\}$. These sets of vertices are then converted to paths $\{D'_1, D'_2, D'_3\}$ in <Figure 2(b)>. In the illustrated example, the elements in $D'_3 = \{v_{13}, v_{23}, v_{33}\}$ is identical to $\{O_{13}, O_{31}, O_{42}\}$. The tuple of $(f_o(v_{13}), f_j(v_{13}))$ returns $(1,3)$, which is effectively described as the tuple of subscripts of O_{13}. Incorporating f_o and f_j, MOS and MJS is defined as a function of m_k.

$$\mathrm{MOS}(m_k) = \{f_o(v_{1k}), \cdots, f_o(v_{Nk}) | v_{lk} \in D'_k\} \quad (1)$$

$$\mathrm{MJS}(m_k) = \{f_j(v_{1k}), \cdots, f_j(v_{Nk}) | v_{lk} \in D'_k\} \quad (2)$$

Finally, there exists a Sorted-MOS of machine m_k, or $\mathrm{SMOS}(m_k)$, which is defined as the sequence of $f_o(v_{lk})$ sorted in the ascending order.

$$\mathrm{SMOS}(m_k) = \mathrm{sorted}(\mathrm{MOS}(m_k)) \quad (3)$$

$$\text{MIO score of } m_k = \sum_{i=1}^{n} |\mathrm{SMOS}(m_k)_i - \mathrm{MOS}(m_k)_i| \quad (4)$$

Once the MOS and its sorted counterpart are obtained, the MIO score is defined as the distance between MOS and SMOS. The MIO score for each machine can be calculated using the Spearman footrule distance between its MOS and SMOS, which involves determining the element-wise absolute differences (L1 norm) between the MOS and SMOS rankings. The total MIO score of a schedule is obtained by summing the MIO scores of all machines. The mathematical formulation is presented in equation (4). It is worth noting that various sorting distances, including Kendall tau distance and bubble sort distance, can also be utilized as metrics for calculating the MIO score.

<Figure 2> illustrates the process of extracting MJS and MOS from the solution sequence, or chromosomes. <Figure 2(a)> describes a 3×4 JSSP, where chromosomes are represented as permutations of 12 operations. In <Figure 2(b)>, these permutations are decoded into ordered combinations where 1, 2, and 3 appear 4 times each to satisfy feasibility constraints. This decoded permutation, with operation numbers assigned according to precedence constraints, can be viewed as a solution to the JSSP. The job sequence number (MJS) for each job on four machines and the relative position of each job's operations on those machines (MOS) is obtained, as shown in <Figure 2(c)> and <Figure 2(d)>.

Looking at m_4, the solution permutation indicates that the machine should process the operations after the job order 3-2-1. That is, m_4 needs to process the operations after the order $\{O_{23} - O_{12} - O_{41}\}$. Therefore, $\mathrm{MOS}(m_4)$ in this case is 2-1-4. When sorted in ascending order, $\mathrm{SMOS}(m_4)$ becomes 1-2-4, indicating that performing O_{12} as the first operation would be more desirable. Thus, a comparison is made between the solution order, 2-1-4, and the desirable order, 1-2-4. The MIO score of m_4 is determined as 2 according to equation (4). Since the other machines are already processing operations in the order of their relative urgency, the MIO score of other machines are all zero. Thus the resulting MIO score of the presented solution is determined as 2.

To further discussion, whether a solution set MIO score as zero will be denoted as 'MIO condition'. A solution satisfying that all of its machines have their MOS in ascending order will be denoted as 'MIO solution'.

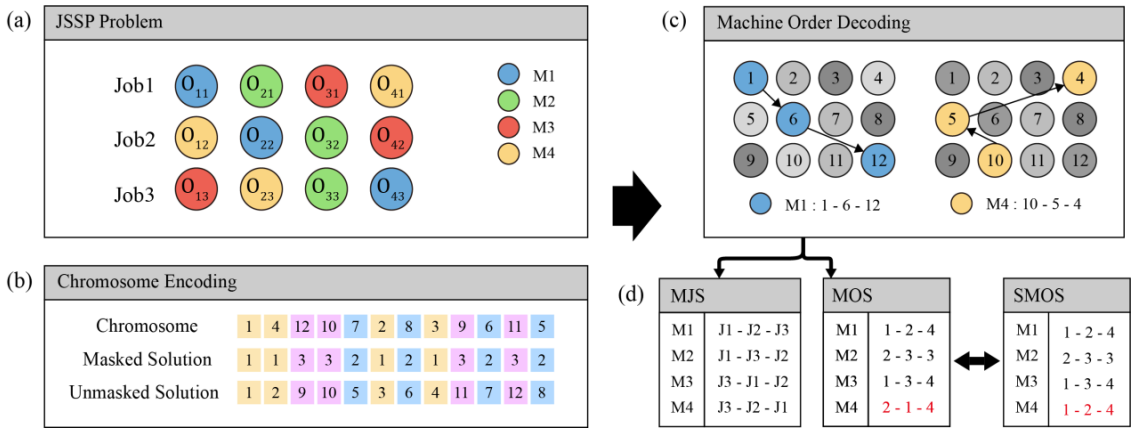

Figure 2. Framework of Calculating MIO Score. (a) A 3×4 JSSP specification (b) Chromosome encoding strategy for GA (c) Decoding machine input order from the chromosome encoding (d) Calculation of MIO score

4.3 GA Strategy based on MIO

Under such research hypotheses, this paper proposes three methods to enhance the performance of GA using the MIO score. Firstly, incorporating the MIO score into the traditional makespan-based fitness function is suggested. This allows individual solutions to reach optimal makespan more rapidly and expected to outperform the traditional GA method, as individuals with higher MIO score might be more desirable in the long run.

The second method proposed in this paper is utilizing this MIO solution (or those individuals with their machines' MOS order already in ascending order) during the crossover process. Applying crossover between existing solutions and MIO solutions can help the evolutionary process converge to better solutions more quickly. By following the direction indicated by the suboptimal solution in the entire solution space, an individual might "jump up" closer to the optimal solution. However, it is important to note that the solution minimizing the MIO score may not always be the global optimal. Therefore, it is effective to gradually decrease the probability of such crossovers as generations progress so that those solutions superior to the MIO solution can be considered.

The third method is replacing solutions with MIO solutions during the mutation process. This allows suboptimal MIO solutions to be included in the entire population more rapidly. One advantage of this approach is that the search process starts from the point closer to the optimal solution in the entire solution space. Gradually reducing the replacement probability is required to favor the evolution of solutions superior to the MIO solution as generations progress.

It is worth noting that obtaining MIO solutions in the form of operation permutations which feasibility is guaranteed is crucial. It can be demonstrated that it is always possible to obtain a MIO solution from the given problem information. The simplest, though not the only, method is to sequentially record operation numbers starting from those positioned first within their respective jobs, followed by the second, and so forth.

If the operation data matrix is arranged such that each row represents the set of operations for the same job, properly ordered according to precedence constraints, this method can be understood as listing operations starting from the first column. Once all elements of the first column are recorded, the same procedure is applied to the second column and continued thereafter. The resulting operation permutation satisfies not only the precedence constraints of each job but also the precedence constraints of each machine, automatically setting MOS in ascending order.

For instance, a possible MIO solution of the JSSP problem presented in <Figure 2(a)> is to simply put the operations in the sequence of $\{o_{11}, o_{12}, o_{13}, o_{21}, o_{22}, o_{23}, \cdots, o_{41}, o_{42}, o_{43}\}$. It is obvious that this solution make m_4 process the operations following the order $\{o_{12}-o_{23}-o_{41}\}$, and let MOS of m_4 be $\{1-2-4\}$.

5. Experimental Results

To validate the research hypothesis, the correlation between MIO score and makespan was analyzed. In this study, the correlation was measured using the MIO score calculated by Spearman footrule distance formula. Benchmark dataset suggested by Adams *et al.* (1988) were selected as their optimal solutions are available. 400 optimal solutions, along with 1600 randomly generated solutions were utilized to calculate Pearson correlation coefficient. The results are as in <Figure 3>. As a result, Pearson correlation coefficients were found to be 0.6132 and 0.6169 for the abz5 and abz6 problems, respectively. This indicates a strong correlation between MIO score and makespan.

Assuming the proposed hypothesis is valid, the proposed methodology was tested for its effectiveness using the benchmark datasets abz5, abz6, abz7, abz8, and abz9, introduced by Adams *et al.* (1988),

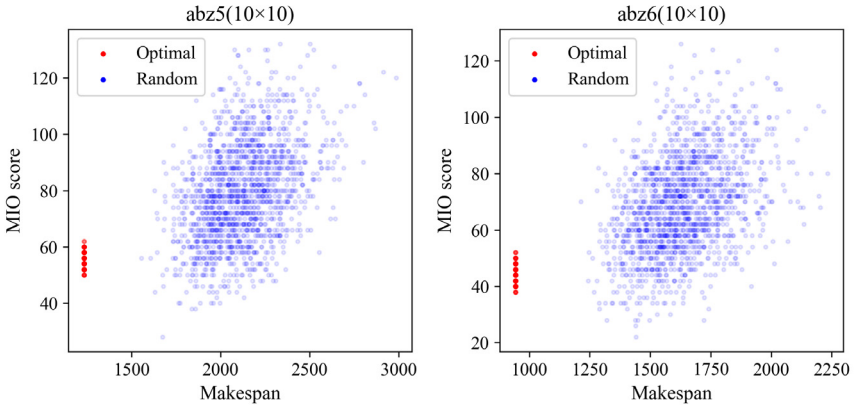

Figure 3. Correlation of MIO-score and Makespan

as well as custom-generated datasets. Custom-generated data differed only in the number of jobs and operations (machines), with the appearance order of specific machines for each job randomized, and processing time uniformly distributed between 11 and 40.

For comparison, the optimization process of minimizing makespan was concurrently conducted using three proposed GA methods from this study and a conventional GA approach. The experiments were held with the same hyperparameters, using 100 individuals evolving through 100 generations. In this study, the PMX (Partially Mapped Crossover) operator was adopted to preserve the characteristics of permutation encoding during crossover. The Roulette Wheel selection method was used for selection, and single swap mutation for the mutation operator. The probabilities for crossover and mutation were set at 0.8 and 0.95, respectively, and were kept consistent across all methods. The combinations of genetic operators used in the experiments are presented in <Table 2>. More specific details of each method are as follows:

Method 1: The adaptive MIO fitness is calculated as the weighted sum of the makespan and the MIO score. The makespan and MIO score of an individual are normalized using the averaged values of the initial population. The weight of each term changes over generations: the weight of the makespan score starts at 0.2 and gradually increases to 1.0, while the weight of the MIO score starts at 0.8 and decreases to zero by the 100th generation. Mathematical representation is in Equation (5).

$$\text{Fitness of individual } I = w_1 \times \frac{\text{makespan of } I}{\text{average makespan}} + w_2 \times \frac{\text{MIO score of } I}{\text{average MIO score}} \quad (5)$$

$$\begin{cases} w_1 = 0.2 + (\text{step size} \times \text{generation}) \\ w_2 = 1.0 - w_1 \end{cases}$$

Method 2 and 3: The probability of replacing one of the parents(method 2), or an individual (method 3) with an MIO sol-

Table 2. Comparison of 3 proposed methodologies

	Method 1 (Adaptive MIO Fitness)	Method 2 (MIO Crossover)	Method 3 (MIO Replacement)
Fitness	Adaptive MIO Fitness (makespan + MIO score)	makespan	makespan
Crossover	PMX Crossover	PMX Crossover One of the parents is replaced to MIO solution	PMX Crossover
Replacement	Single Swap Mutation	Single Swap Mutation	Single Swap Mutation Individuals are randomly replaced to a MIO solution

Table 3. Average Makespan (normalized time) of 4 GA Algorithms

	$N \times M$	Basic GA	Adaptive MIO Fitness	MIO Crossover	MIO Replacement
Adams et al.	abz5 (10×10)	1362.4 (100)	1417.6 (100.8)	1340.6 (99.21)	**1325.1** (97.95)
	abz6 (10×10)	1053.4 (100)	1116.1 (101.99)	1003.4 (98.01)	**991.2** (97.15)
	abz7 (20×15)	891.9 (100)	964.2 (87.79)	806.3 (86.19)	**789.8** (85.91)
	abz8 (20×15)	935.9 (100)	987.7 (101.55)	831.5 (98.55)	**811.1** (100.31)
	abz9 (20×15)	944.8 (100)	1012 (102.79)	898.2 (100.28)	**869.2** (98.87)
Custom Dataset	20×20	1111.1 (100)	1175.9 (101.35)	984.5 (105.95)	**976.1** (106.87)
	30×30	1914.7 (100)	2030.3 (105.84)	1559.4 (91.08)	**1531** (90.81)
	40×40	2695.7 (100)	2843.2 (105.48)	2055.2 (98.37)	**2008.3** (99.21)
	50×20	2122.5 (100)	2226.2 (101.73)	1671.7 (93.25)	**1646.5** (91.71)
	100×15	3448.2 (100)	3552 (102.58)	2994.7 (76.93)	**2960.1** (75.06)

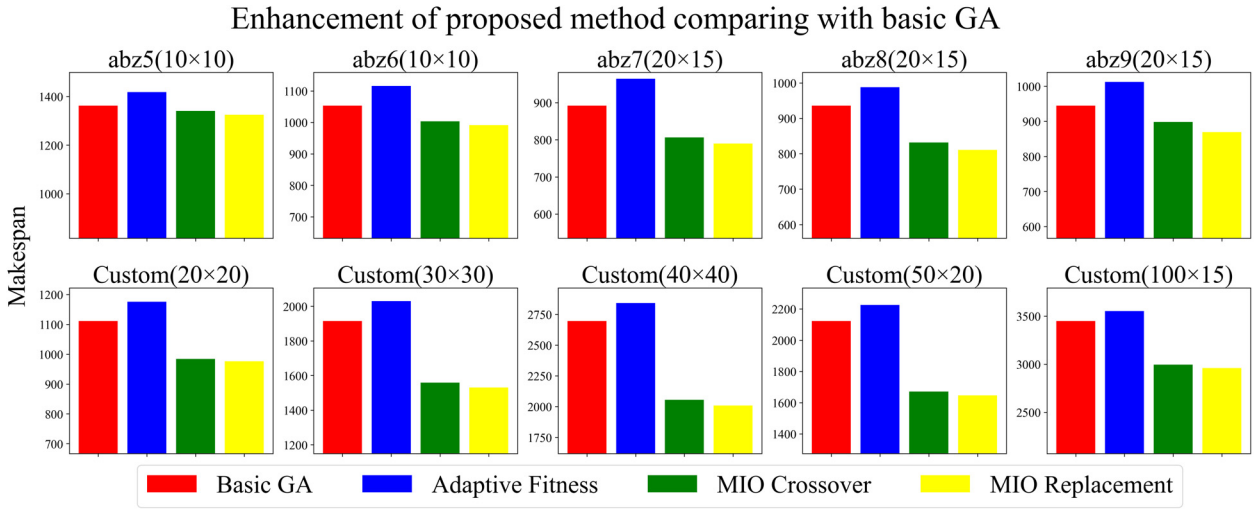

Figure 4. Comparison of the Enhancement of Search Efficiency

ution is both defined as p_{mio}. This probability starts at a value of 0.9 and decreases progressively by multiplying the original probability by 0.99 each time the MIO utilization process is executed. This approach is to prevent the MIO solutions from dominating the entire population, particularly after the evolution process has sufficiently progressed.

Each experiment case was repeated 10 times to ensure robustness of the proposed algorithm. The results are as in <Table 3>. The percentile of performance enhancement with respect to basic GA method are presented in <Figure 4>. The proposed methods generally demonstrated comparable or improved performance with respect to the execution time. In Method 1, the extra computations during the fitness calculation process led to a slight increase in execution time. Conversely, in Methods 2 and 3, the reuse of MIO solutions resulted in a slight reduction in execution time.

Among the 10 problems, the search performance of Method 3 surpassed all other three. Following closely behind was Method 2. It is also shown that the extent of performance improvement in Methods 2 and 3 is more pronounced as the problem size grows larger. The results of the experiment confirm that utilizing the information of MIO solutions can enhance the search efficiency during the evolutionary process of GA for JSSP. Methods 2 and 3 shares similarity in that both enable utilizing MIO solutions during the crossover process, but differ from the point that in Method 2, one of the parents are replaced only during the crossover process and in Method 3, the MIO solution replaces the child and kept in the population. The results suggest that the higher search efficiency of Method 3, including both the makespan and the shorter execution time, can be attributed to these differences.

<Figure 5> shows the overall tracking result of the makespan and the MIO score of all collected individuals during the evolutionary phase. The contradiction of evolved individual, marked as red, and the initialized population, marked as blue, indicates that MIO solutions offered the opportunity to rapidly approach optimal solutions, thus greatly narrowing the gap between the population's makespan and the optimal solution. One remark is that even after the engagement of MIO solutions in the population, the search for the optimal continued, leading to further advance to optimal solutions in areas where the MIO score was not 0. In the proposed experiments, reducing the probability of crossover and replacement as the evolutionary phase progressed helped avoid the risk of converging to local optima, thereby enabling further exploration. This observation suggests that while the MIO crossover or replacement strategy can provide efficient pre-determined sub-optimal solutions, it does not necessarily guarantee the optimality of the final solution. Therefore, in subsequent search phases, alternative methods that do not rely solely on the MIO score, which carries the risk of trapping in local optima, should be employed.

On the other hand, the Adaptive Fitness Strategy (Method 1) exhibited poorer search performance to basic GA. Introducing the indirect objective of minimizing the MIO score by adding the term in the fitness calculation appears to have led the optimization process to overly prioritize the MIO score, hindering the original objective of makespan minimization. This suggests that while a schedule solution with a good makespan may have a good MIO score, the converse does not necessarily hold true. Therefore, when incorporating the philosophy proposed in this study into optimization, it is necessary to selectively introduce the features of MIO solutions rather than focusing primarily on the MIO score.

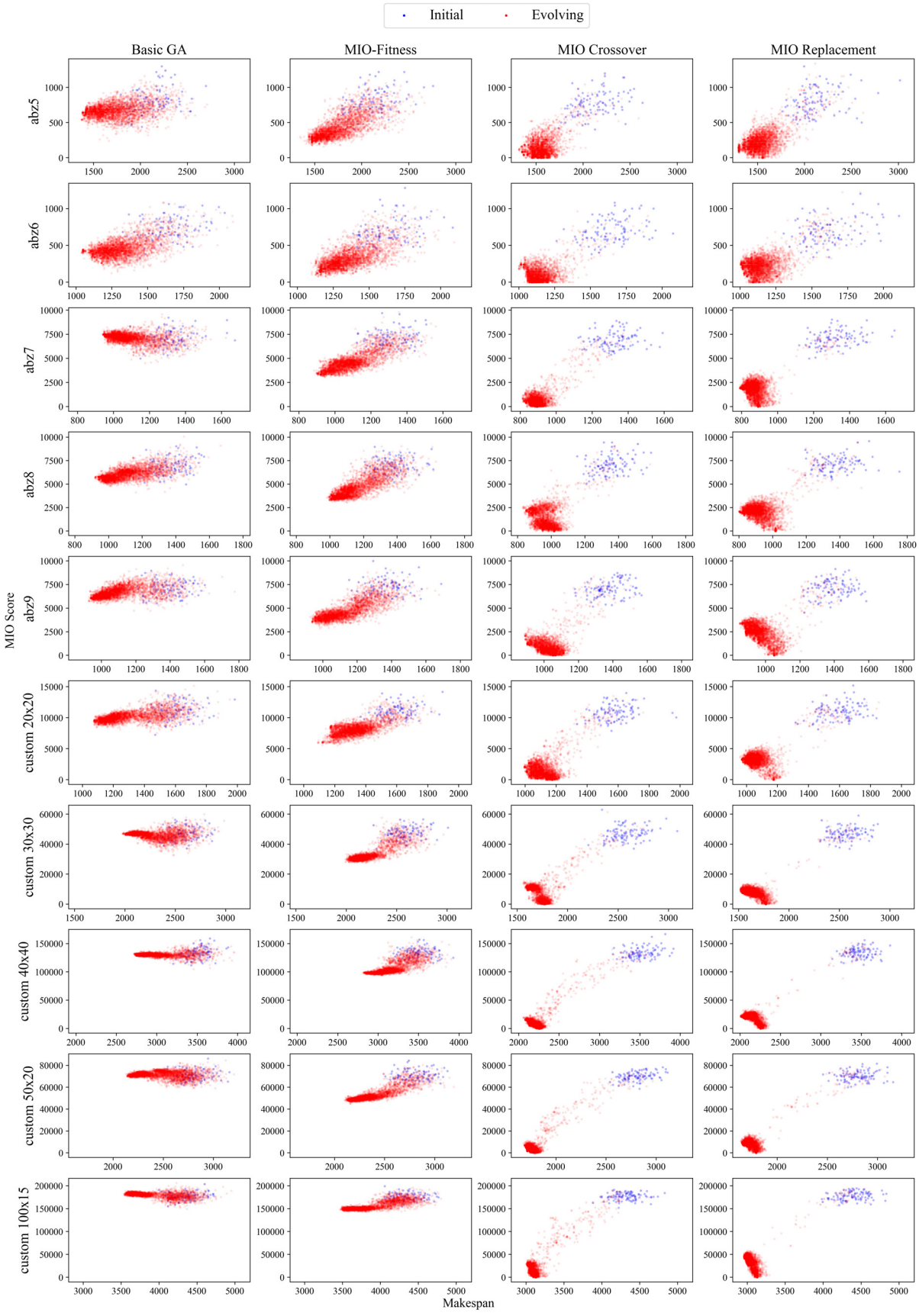

Figure 5. Comparison between the Convergence of basic GA and the MIO-based GA

6. Conclusion

In conclusion, this study proposed three methods to enhance the performance of GA for solving JSSP based on the analysis of Machine Input Order (MIO) Score which derived from Machine-Job Sequence (MJS) and Machine-Operation Sequence (MOS). The novelty of the proposed methods lies in initiating the search from relatively close sub-optimal solutions to optimal ones, regardless of problem size. Moreover, the construction heuristic of MIO solutions is grounded in the observation of the graph structure of JSSP and thus can be applied regardless of problem size. The methodology proposed in this research can be applied to initializing suitable candidate solutions in the makespan optimization process for larger JSSP instances. Additionally, this philosophy may have potential applications beyond JSSP, extending to other scheduling problems as well.

However, it should be noted that the study has limitations as it has not fully examined all hyperparameters necessary to reach an optimal solution. This aspect could be improved through further experiments and leveraging existing research findings.

References

Adams, J., Balas, E., and Zawack, D. (1988), The shifting bottleneck procedure for job shop scheduling, *Management Science*, **34**(3), 391-401.

Bierwirth, C. (1995), A generalized permutation approach to job shop scheduling with genetic algorithms, *Operations-Research-Spektrum*, **17**(2), 87-92.

Çaliş, B. and Bulkan, S. (2015), A research survey: Review of AI solution strategies of job shop scheduling problem, *Journal of Intelligent Manufacturing*, **26**, 961-973.

Cheng, R., Gen, M., and Tsujimura, Y. (1996), A tutorial survey of job-shop scheduling problems using genetic algorithms—I, *Representation, Computers & Industrial Engineering*, **30**(4), 983-997.

Davis, L. (1985), Job shop scheduling with genetic algorithms, *Proceedings of the International Conference on Genetic Algorithms and their Applications*, Hillsdale: Lawrence Erlbaum, 136-149.

Gen, M., Tsujimura, Y., and Kubota, E. (1994, October), Solving job-shop scheduling problems by genetic algorithm, *Proceedings of IEEE International Conference on Systems, Man and Cybernetics*, **2**, 1577-1582. IEEE.

Gonçalves, J-F., de Magalhães Mendes, J-J., and Resende, M-G.(2005), A hybrid genetic algorithm for the job shop scheduling problem, *European Journal of Operational Research*, **167**(1), 77-95.

Holland, J-H. (1975), Adaptation in Natural and Artificial Systems, The University of Michigan Press, Ann Arbor, MI.

Mattfeld, D-C. and Bierwirth, C.(2004), An efficient genetic algorithm for job shop scheduling with tardiness objectives, *European Journal of Operational Research*, **155**(3), 616-630.

Nakano, R. and Yamada, T. (1991, July), *Conventional genetic algorithm for job shop problems*, In ICGA (Vol. 91, 474-479).

Park, B-J., Choi, H-R., and Kim, H-S. (2003), A hybrid genetic algorithm for the job shop scheduling problems, *Computers & Industrial Engineering*, **45**(4), 597-613.

Pinedo, M-L. (2012), *Scheduling* (Vol. 29, 249), New York: Springer.

Someya, H. and Yamamura, M.(1999), A genetic algorithm without parameters tuning and its application on the floorplan design problem, *Proceedings of GECCO '99*, 620-627.

Tamilarasi, A. (2010), An enhanced genetic algorithm with simulated annealing for job-shop scheduling, International Journal of Engineering, *Science and Technology*, **2**(1), 144-151.

Watanabe, M., Ida, K., and Gen, M.(2005), A genetic algorithm with modified crossover operator and search area adaptation for the job-shop scheduling problem, *Computers & Industrial Engineering*, **48**(4), 743-752.

Zhang, C., Li, P., Rao, Y., and Li, S.(2005), A new hybrid GA/SA algorithm for the job shop scheduling problem, In *Evolutionary Computation in Combinatorial Optimization: 5th European Conference*, EvoCOP 2005, Lausanne, Switzerland, March 30-April 1, 2005. Proceedings 5, 246-259. Springer Berlin Heidelberg.

Zhou, H., Feng, Y., and Han, L.(2001), The hybrid heuristic genetic algorithm for job shop scheduling, *Computers & Industrial Engineering*, **40**(3), 191-200.

저자소개

백지원 : 서울대학교 조선해양공학과에서 2024년 학사학위를 취득하고 현재 서울대학교 조선해양공학과 석사과정에 재학 중이다. 연구분야는 최적화, 스케줄링 및 강화학습이다.

우종훈 : 서울대학교 조선해양공학과 교수로 재직 중이다. 연구분야는 생산관리, 시뮬레이션, 최적화, 심층강화학습이다.