Heuristic NSGA-II for Flexible Job Shop Scheduling Problems

Yuhang Jiang
University of Trento
Trento, Italy
yuhang.jiang@studenti.unitn.it

Saad Raza Hussain Shafi
University of Trento
Trento, Italy
saad.hussainshafi@studenti.unitn.it

***Abstract*—Flexible Job Shop Scheduling Problem (FJSP) is an NP-hard optimization challenge in manufacturing. This paper enhances NSGA-II by integrating heuristic initialization and local search strategies. Heuristic initialization assigns machines based on shortest processing times, ensuring high-quality initial populations tailored to practical manufacturing needs, while local search adjusts solutions during initialization to improve diversity. Experiments on benchmark datasets demonstrate improvements in solutions diversity and runtime efficiency compared to the standard NSGA-II. The enhanced algorithm achieves fewer makespans, making it highly suitable for real-world applications. Code and results can be found at this repository.**

***Index Terms*—NSGA-II, FJSP, Multi-Objective Optimization, Heuristic Models, Genetic Algorithms**

I. Introduction

THIS problem is inspired by potential applications in the manufacturing sector, one of the largest industrial domains. Scheduling involves the effective utilization of production resources while considering various constraints. The flexible job shop scheduling problem (FJSP) is described as the completion of n jobs using m machines, with each machine performing at least one operation. Operations may be processed on one or more machines. The objective of this study is to determine the production sequence that minimizes the objective function while allocating the most suitable processing machines. Thus, machine selection and process sequencing are the two primary sub-problems in FJSP. The FJSP is classified as an NP-hard problem, but significant efforts have been made to solve it using exact, heuristic, and meta-heuristic algorithms. Among these, meta-heuristics often provide higher efficiency and require lower computational power for solving multiple objective functions. Some important objective functions include makespan, maximum workload, total energy consumption, quality, carbon emissions, and tardiness.

In practice, under the constraint that machines meet the load-bearing requirements, decision-makers tend to prioritize faster task completion. For instance, introducing weighted adjustments to the objective functions can shift the optimization preference towards shorter makespan or balanced load distribution, depending on the weights assigned. This reflects the practical need to balance efficiency and reliability in real-world manufacturing scenarios. Recent studies [1] highlight the effectiveness of weighted objective adjustments and hybrid meta-heuristic algorithms in addressing such challenges, particularly in cases involving multiple conflicting objectives. However, applying weights to the objectives is an explicit way to alter the importance of each objective, thereby influencing the distribution of solutions. [2] This approach does not directly enhance the efficiency of population initialization or the evolutionary process. This work focuses on guiding the algorithm to find solutions with smaller makespan while emphasizing the optimization of the algorithm's internal mechanisms, using different heuristic methods to improve the quality of initial populations.

This project is a two-objective optimization task, which aims to minimize the maskespan and the load balance of the machine. The definitions of the two indicators are as follows:

1) Makespan: Makespan measures the total time required to complete all tasks. The goal is to minimize makespan, ensuring all jobs are finished as quickly as possible. Mathematically:

- **Definition:** The total time needed to complete all jobs, defined by the maximum completion time of all jobs.
- **Formula:**
$$C_{\max} = \max_j \{c_{j,h_j}\},$$
where:
  - $C_{\max}$: the maximum completion time.
  - c_{j,h_j}: the completion time of the last operation h_j of job j.
- **Objective:**
$$\min C_{\max}$$

2) Load Balance: Load balance measures the even distribution of tasks across machines. A lower standard deviation of machine loads (σ_{load}) indicates better balance, improving resource utilization and avoiding machine overload. The objective is to minimize load imbalance:

- **Machine Load Definition:** The load of a machine i is the sum of the processing times for all tasks assigned to it.
- **Machine Load Formula:**
$$T_i = \sum_{j,h} p_{ijh},$$
where:

  - T_i: the total load (time) of machine i.
  - p_{ijh}: the processing time of the h-th operation of job j on machine i.

- **Load Balance Formula:**

$$\sigma_{\text{load}} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(T_i - \overline{T}\right)^2},$$

where:

  - σ_{load}: the standard deviation of machine loads.
  - T_i: the load of machine i.
  - $\overline{T}$: the average load across all m machines.
  - m: the total number of machines.

- **Objective:**

$$\min \sigma_{\text{load}}$$

II. Problem Description

The mathematical model of the flexible job shop scheduling problem is defined as follows:

- n: Total number of jobs.
- m: Total number of machines.
- W: Total set of machines.
- i, e: Machine indices, where $i, e = 1, 2, 3, \ldots, m$.
- j, k: Job indices, where $j, k = 1, 2, 3, \ldots, n$.
- h_j: Total number of operations for job j.
- l: Machine indices, where $l = 1, 2, 3, \ldots, h_j$.
- W_{jh}: Set of available machines for the h-th operation of job j.
- m_{jh}: Number of available machines for the h-th operation of job j.
- O_{jh}: The h-th operation of job j.
- M_{ijh}: Processing of the h-th operation of job j on machine i.
- p_{ijh}: Processing time of the h-th operation of job j on machine i.
- s_{jh}: Start time of the h-th operation of job j.
- c_{jh}: Completion time of the h-th operation of job j.

The decision variables are defined as follows:

- If operation O_{jh} is assigned to machine i, then $M_{ijh} = 1$; otherwise, $M_{ijh} = 0$.
- If operation O_{ijh} is processed before operation O_{ikjl}, then $x_{ijh,ikjl} = 1$; otherwise, $x_{ijh,ikjl} = 0$.

The model constraints are as follows:

$$s_{jh} + x_{ijh}p_{ijh} < c_{jh}, \tag{1}$$

$$c_{jh} < s_{j(h+1)}, \tag{2}$$

$$c_{jh_j} < C_{\max}, j, h \tag{3}$$

$$s_{jh} + p_{ijh} \leq s_{kl} + L(1 - y_{ijhkl}), \tag{4}$$

$$c_{jh} \leq s_{j(h+1)} + L(1 - y_{ikjlj(h+1)}), \tag{5}$$

$$\sum_{i=1}^{m_{jh}} x_{ijh} = 1, \tag{6}$$

$$\sum_{j=1}^{n}\sum_{h=1}^{h_j} y_{ijhkl} = x_{ikl}, \tag{7}$$

$$\sum_{k=1}^{n}\sum_{l=1}^{h_k} y_{ijhkl} = x_{ijh}, \tag{8}$$

Equations (1) and (2) define the constraints before and after each process, ensuring that operations follow the correct sequence. Equation (3) imposes job completion time constraints, stating that the completion time of each job cannot exceed the total job completion time.

Equations (4) and (5) specify that the same machine can only process one operation at a time. Equation (6) ensures that a given operation is processed by only one machine. Finally, Equations (7) and (8) ensure that each machine operates in a cycle, adhering to the job scheduling sequence.

III. Data Processing

This project utilizes three datasets: Barnes [3], Brandimarte [4], and Dauzere [5]. The FJSP datasets used in this study follow the structure shown in fig1 The first line contains three values: the first value indicates the number of jobs, which also corresponds to the number of lines that follow; the second value represents the total number of available machines; the third value specifies the average number of machines available for each operation. Starting from the second line, each line corresponds to a job. The first value in each line specifies the number of steps required for that job. Following this, the first step lists the number of available machines. For each operation, subsequent pairs of values indicate the machine number capable of performing the operation and the corresponding processing time, followed by the number of available machines for the next step, and so on.

The preprocessing step involves reading and organizing data from the FJSP datasets. The function first parses the input file to extract key metadata such as the number of jobs (`num_jobs`), the number of machines (`num_machines`), and the number of operations for each job. Each job's operations are further analyzed to extract machine-time pairs, indicating the machines capable of processing the operation and their respective processing times. This data is then used to construct a 3D matrix (`processing_times`) with dimensions corresponding to jobs, operations, and machines. Each matrix entry contains the processing time of an operation on a machine, while unavailable operations are represented by `inf` to indicate infeasibility. This structured representation facilitates efficient scheduling and optimization.

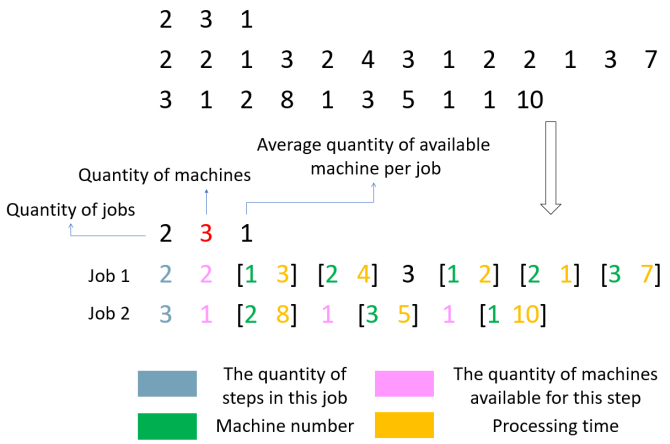

Fig. 1: Data Description

IV. Algorithm Description

The Non-dominated Sorting Genetic Algorithm II (NSGA-II) [6] is a multi-objective optimization algorithm designed to handle multiple conflicting objectives simultaneously. Unlike traditional genetic algorithms, which optimize a single objective, NSGA-II uses non-dominated sorting to classify solutions into Pareto fronts, where solutions in the first front are not dominated by any other solutions. It employs a crowding distance mechanism to maintain diversity within each front and uses elitism to ensure the best solutions are carried over to subsequent generations.

The algorithm is implemented using the `inspyred` Python library [7] and defines key components tailored to the FJSP. Specifically, it introduces the `standard_generator` to randomly assign machines to operations and the `standard_evaluator` to calculate makespan and load balance. Additionally, the algorithm incorporates two advanced strategies: `diversified_generator`, which employs a greedy heuristic to assign machines based on the shortest processing times, and `local_search`, which randomly selects and reassigns machines to improve solutions within the search space. In essence, `local_search` plays a role of random perturbation in the initial stage. Finally, `generator_with_local_search` is used as the generator of the heuristic NSGA-II.

A. Generator Functions

The generator functions generate the initial solutions for the algorithm. Two methods have been used in the experimentation.

1) Standard NSGA-II: In the standard NSGA-II, the algorithm randomly assigns a machine to each operation of each job. The standard generator loops over jobs and operations and finds the available machines for each operation and randomly select from the available one. Then the evaluator calculates the objective functions, make span and load balance. Finally, the algorithm evolves across the generation and calculates the final fitness of the final population.

2) Heuristic NSGA-II: Alternatively, the improved version of NSGA-II works by diversified initialization controlled by two main parameters:

DIVERSIFIED_INIT_PROB This controls whether the heuristic or random generator is used during initialization. If $prob < DIVERSIFIED\_INIT\_PROB$, the heuristic generator is used for the processing times which assigns machines with shortest processing time for each operation.

```
for job_idx in range(num_jobs):
    job_schedule = []
    for op_idx in range(num_operations):
        available_machines = np.where(np.isfinite(
            processing_times[job_idx, op_idx]))[0]
        if len(available_machines) > 0:
            best_machine = available_machines[np.
                argmin(processing_times[job_idx,
                op_idx, available_machines])]
            job_schedule.append(best_machine)
    solution.append(job_schedule)
return solution
```

Listing 1: Heuristic Generator Code

LOCAL_SEARCH_PROB If the local search probability is employed, this call search function randomly elect a job and an operation and reassign a machine.

```
job_idx = np.random.randint(0, num_jobs)
op_idx = np.random.randint(0, len(solution[job_idx])
    )
current_machine = solution[job_idx][op_idx]
available_machines = np.where(np.isfinite(
    processing_times[job_idx, op_idx]))[0]
# If there are other available machines, randomly
    replace
if len(available_machines) > 1:
    available_machines = available_machines[
        available_machines != current_machine]
    new_machine = np.random.choice(
        available_machines)
    solution[job_idx][op_idx] = new_machine
```

Listing 2: Heuristic Generator Code

V. Results

After multiple experiments, setting `LOCAL_SEARCH_PROB` to 0.05 and `DIVERSIFIED_INIT_PROB` to 0.1 yielded satisfactory results. Due to space limitations, this report only presents the results of the Brandimarte dataset, while detailed results for other datasets are available in the repository.

When both the algorithms are benchmarked on Mk01 dataset from Brandimarte Dataset, the resultant job scheduling is visualized below in fig[2, 3].

A. Makespan and Load Balance

In fig4, we can see that since the initialization population tends to choose the one with less processing time, which eventually leads to a smaller makespan of the Pareto front. At the same time, the load balance is also within an acceptable range.

Fig. 2: Job Scheduling using NSGA-II on Mk01

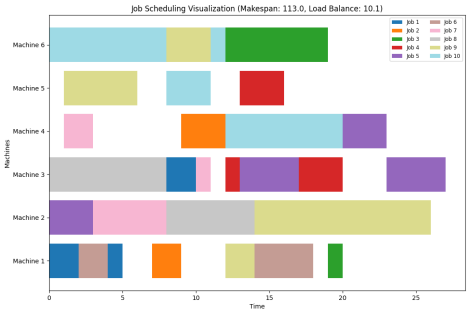

Fig. 3: Job Scheduling using Heuristic NSGA-II on Mk01

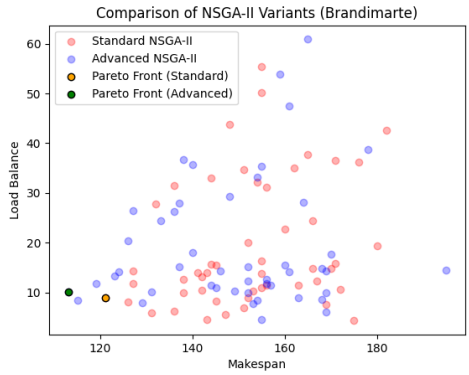

Fig. 4: Solutions and Pareto Front

B. Diversity

When diversity is compared across the two algorithm as shown in fig 5, advanced NSGA-II generally achieves higher diversity, particularly on datasets like MK03, MK06, MK09, and MK10, indicating its ability to explore the solution space more effectively. The two algorithms perform similarly on simpler datasets like MK01 and MK05, suggesting the improvements in Advanced NSGA-II are most beneficial for maintaining diversity in more complex scenarios.

C. Runtime

As shown in fig.6, despite the combination of heuristic initialization and local search, the improved version does not significantly increase the running time, but instead reduces the running time due to the high-quality initialization population,

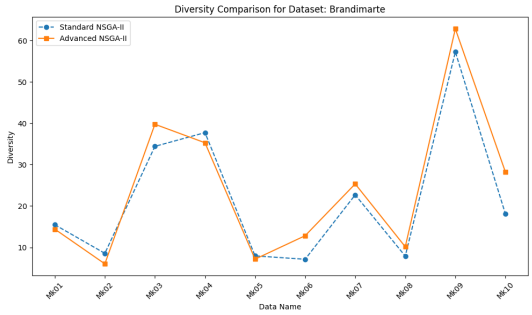

Fig. 5: Diversity Comparison of two Algortihms on Mk01

showing good efficiency and stability. This makes it more suitable for scenarios that require fast convergence while retaining the diversity of solutions.

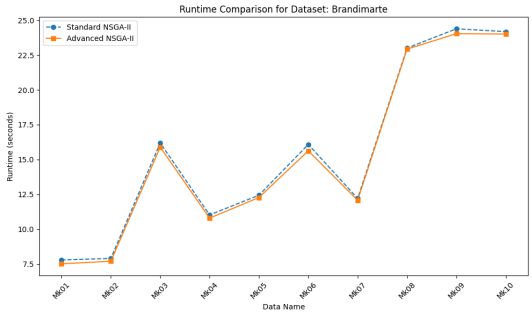

Fig. 6: Runtime Comparison for Datasets with Increasing Comlexity

D. Hypervolume

As shown in fig.7 advanced NSGA-II shows slight improvements, especially on MK08 and MK10. But in general, the difference is not obvious

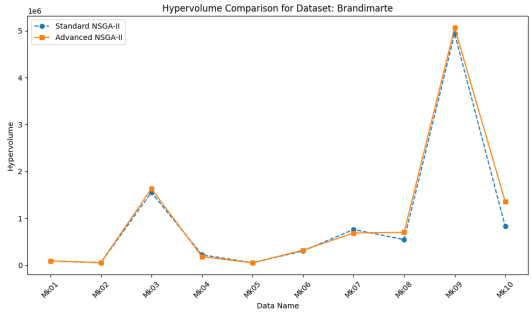

Fig. 7: Hypervolume Performance on Mk01

VI. Conclusion

This paper introduces an improved NSGA-II for FJSP, leveraging heuristic initialization and local search to enhance initial population quality and address practical manufacturing requirements. While these strategies provide a strong starting point for the algorithm, their one-time application limits

adaptability to dynamic or evolving scheduling environments. Additionally, the computational overhead of the initialization phase could pose challenges for large-scale problems. Future work will explore dynamic scheduling, adaptive heuristics, and scalability improvements for broader applications.

References

[1] B. Jiang, Y. Ma, L. Chen, B. Huang, Y. Huang, and L. Guan, "A review on intelligent scheduling and optimization for flexible job shop," *International Journal of Control, Automation and Systems*, vol. 21, no. 10, pp. 3127–3150, 2023.

[2] J. Zhang, Z. Dong, and T. Chen, "Multi-objective optimal allocation of water resources based on the nsga-2 algorithm while considering intergenerational equity: a case study of the middle and upper reaches of huaihe river basin, china," *International Journal of Environmental Research and Public Health*, vol. 17, no. 24, p. 9289, 2020.

[3] J. W. Barnes and J. B. Chambers, "Solving the job shop scheduling problem with tabu search," *IIE transactions*, vol. 27, no. 2, pp. 257–263, 1995.

[4] P. Brandimarte, "Routing and scheduling in a flexible job shop by tabu search," *Annals of Operations research*, vol. 41, no. 3, pp. 157–183, 1993.

[5] S. Dauzere-Peres and J. Paulli, *Solving the General Multiprocessor Job-shop Scheduling Problem*, management report series 182 ed., ser. Management Report Series, 1994.

[6] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[7] A. Garrett, "inspyred (version 1.0.1) [software]," https://github.com/aarongarrett/inspyred, 2012, inspired Intelligence.

Appendix A
Additional Figures

This section includes additional figures that support the main text. More results are available in repository.

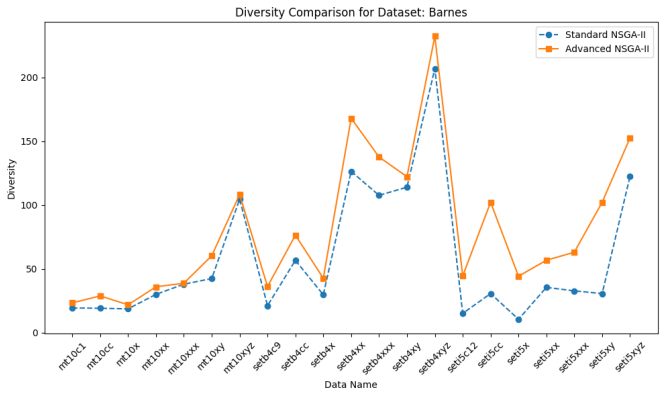

Fig. 8: Barnes Diversity Comparison

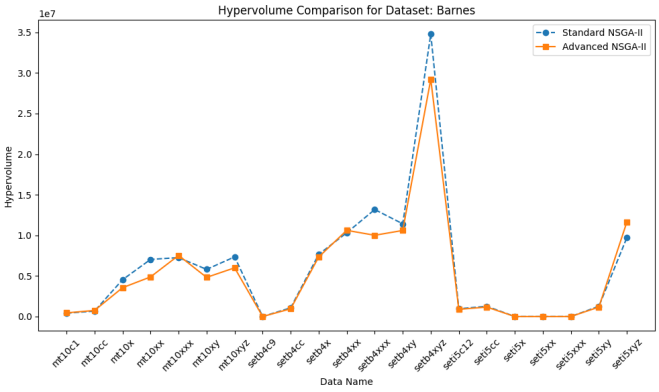

Fig. 9: Barnes Hypervolume Comparison

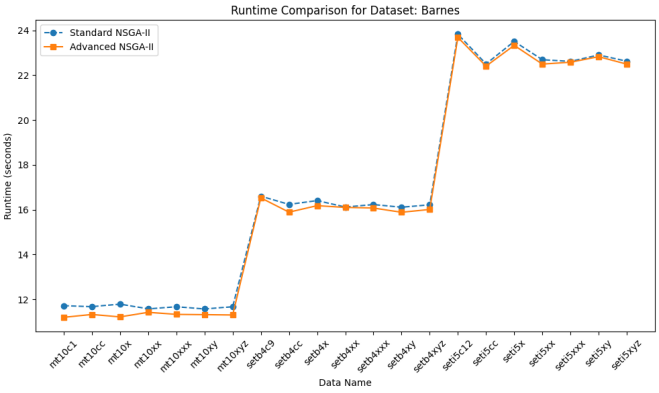

Fig. 10: Barnes Runtime Comparison

Fig. 11: Dauzere Diversity Comparison

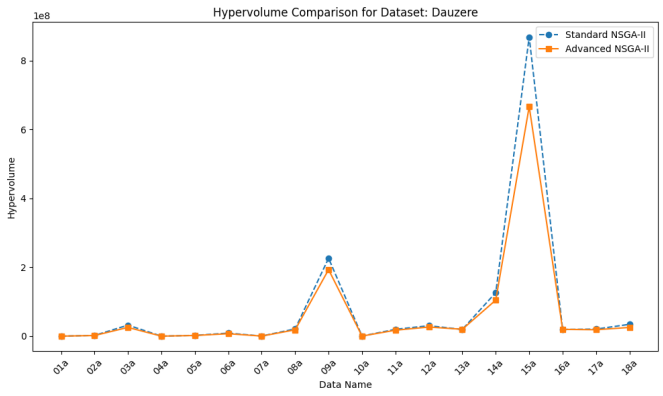

Fig. 12: Dauzere Hypervolume Comparison

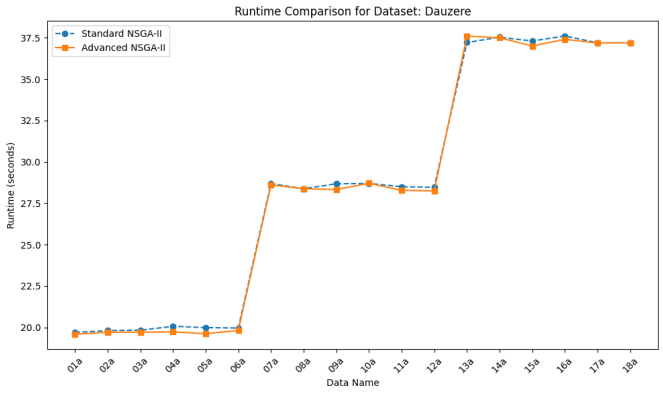

Fig. 13: Dauzere Runtime Comparison

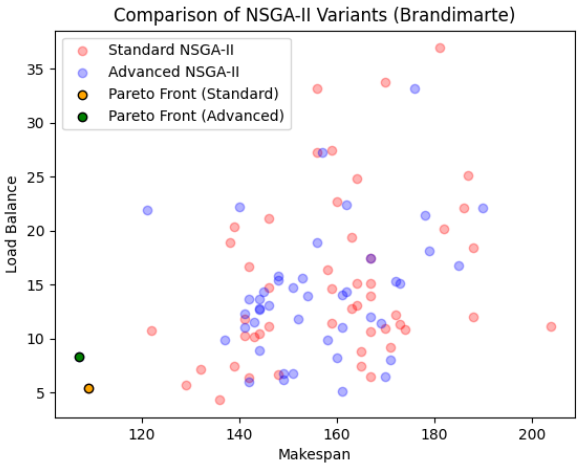

Fig. 14: Mk02 Comparison NSGA-II Variants (Brandimarte)

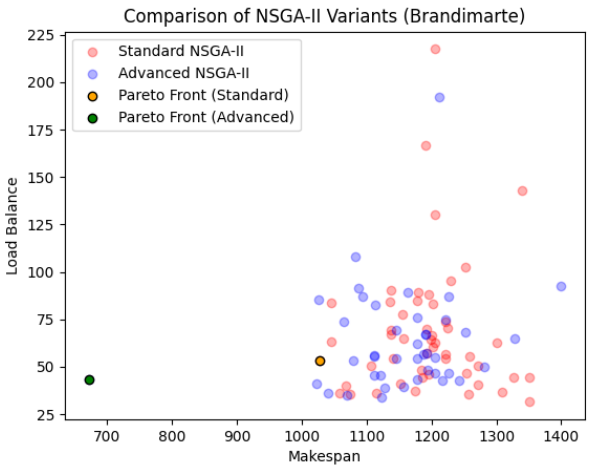

Fig. 15: Mk03 Comparison NSGA-II Variants (Brandimarte)

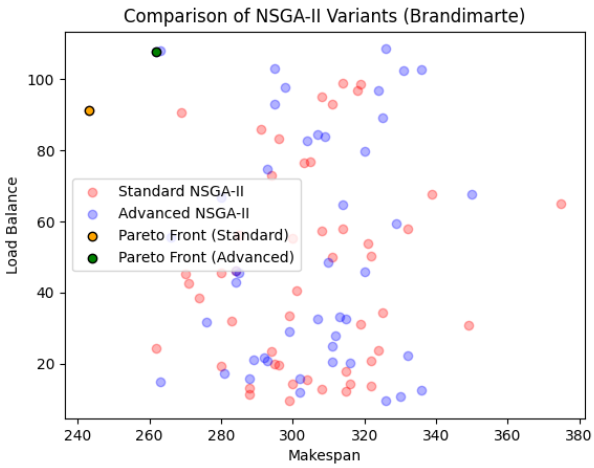

Fig. 16: Mk04 Comparison NSGA-II Variants (Brandimarte)

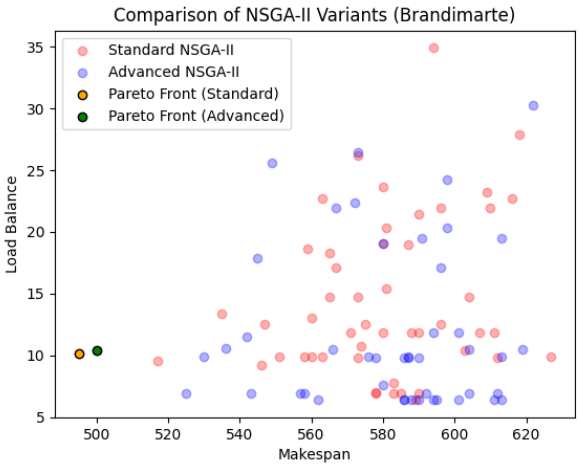

Fig. 17: Mk05 Comparison NSGA-II Variants (Brandimarte)

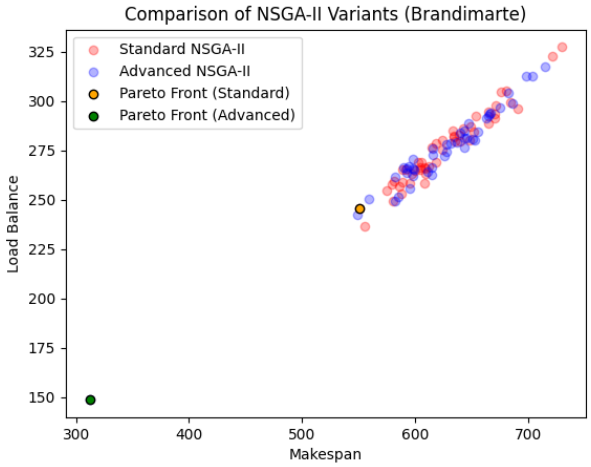

Fig. 18: Mk06 Comparison NSGA-II Variants (Brandimarte)

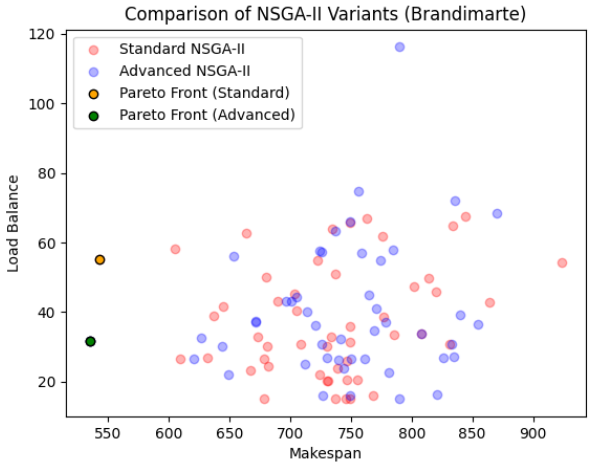

Fig. 19: Mk07 Comparison NSGA-II Variants (Brandimarte)

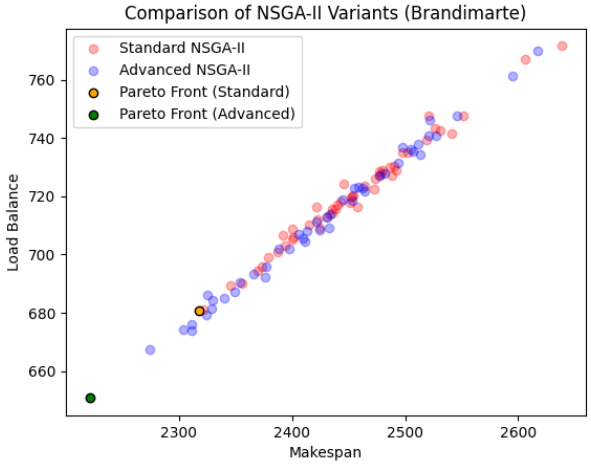

Fig. 20: Mk08 Comparison NSGA-II Variants (Brandimarte)

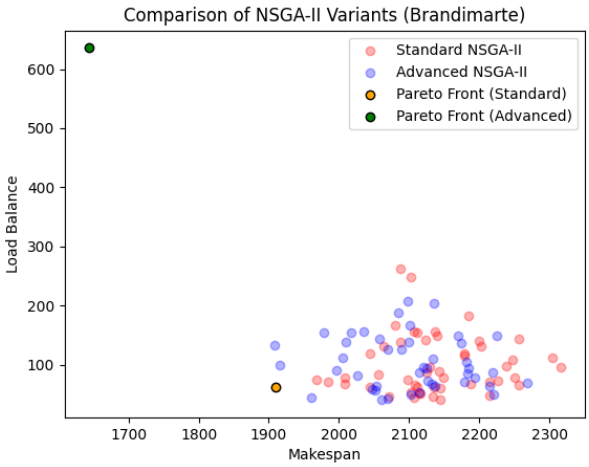

Fig. 21: Mk09 Comparison NSGA-II Variants (Brandimarte)

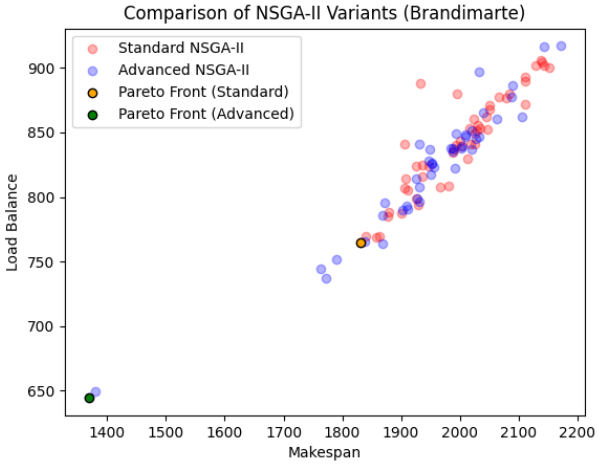

Fig. 22: Mk10 Comparison NSGA-II Variants (Brandimarte)