EVOLUTIONARY MULTI-OBJECTIVE OPTIMIZATION

Supporting Report for the INF421 Programming Project

February 16, 2025

SILVA CLAUDINO Ariel, ZUIN RUIZ Luis Henrique

SUMMARY

1 INTRODUCTION

This project focuses on implementing and analyzing the **Non-Dominated Sorting Genetic Algorithm II (NSGA-II)**, one of the most widely used multi-objective evolutionary algorithms. The NSGA-II maintains a diverse set of solutions and employs specialized selection mechanisms, such as non-dominated sorting and crowding distance, to ensure a well-distributed Pareto front. Our implementation will be evaluated on the **LeadingOnesTrailingZeros (LOTZ)** benchmark function, a bi-objective pseudo-Boolean function designed for theoretical analysis. Additionally, we will explore a modified version of NSGA-II that dynamically updates the crowding distance, aiming to enhance performance and solution diversity.

Through this study, we aim to assess the efficiency of NSGA-II in covering the **Pareto front**, compare its performance under different problem settings, and evaluate the impact of the proposed modification. The results will provide valuable insights into the strengths and limitations of evolutionary multi-objective optimization in handling complex optimization tasks.

2 QUESTIONS

2.1 Task 1

This task's goal was to implement a structure that allows efficient representation and comparison of individuals in a multi-objective optimization setting. Two key components, the **Individual** class and the **ObjectiveValue** class, were defined to achieve this.

The Individual class represents a candidate solution, storing its bit-string representation and the corresponding objective values. Each individual is associated with an **ObjectiveValue** instance, which encapsulates the evaluation of the multi-objective function. To facilitate the necessary comparisons, the ObjectiveValue class implements methods for weak and strict domination, enabling the determination of whether one individual's objective values outperform another's in all or some objectives.

Beyond those two classes, we also introduced the ObjectiveValueConstructor, which generalizes how objective values are built from an Individual. By abstracting the creation of ObjectiveValue objects, we can seamlessly swap different benchmark functions (like LOTZ or more complex ones) without changing the rest of the code. Besides, it allows us to create instances of ObjectiveValues by calling the constructor just like we would call a mathematical function.

The implementation ensures that individuals can be compared based on their **dominance relationships**, which is crucial for the functioning of evolutionary multi-objective algorithms like NSGA-II. This structure provides a well-defined and efficient approach to handling multi-objective optimization problems.

2.2 Tasks 2 & 3

2.2.1 • Code implementation

For tasks 2 and 3, we decided to implement the classes **LOTZ**, **mLOTZ**, and **mLOTZConstructor**. These classes represent the *LeadingOnesTrailingZeros* benchmark and its multi-objective extension. The constructor class encapsulates the `mLOTZ` class and inherits from `ObjectiveValueConstructor`, allowing for a better integration with the earlier components. These classes are crucial for tasks 2 and 3, as well as for testing the

`Individual` and **ObjectiveValue** classes. For instance, **LOTZ** evaluates a single bit string for its number of leading ones and trailing zeros, while **mLOTZ** extends this logic to multiple chunks and multiple objectives, illustrating how the newly introduced classes handle multi-objective evaluations.

2.2.2 • Pareto Set and Pareto Front of *mLOTZ*

The *mLOTZ* function partitions an individual $\mathbf{x} \in \{0,1\}^n$ into $\frac{m}{2}$ independent chunks, each of length $n' = \frac{2n}{m}$. Each chunk is evaluated using the bi-objective *LeadingOnesTrailingZeros (LOTZ)* function, where the two objectives count consecutive 1s from the start and consecutive 0s from the end.

A solution is **Pareto optimal** if and only if each chunk is of the form $1^i 0^{n'-i}$ for some $i \in \{0, \ldots, n'\}$. The **Pareto set** consists of all individuals formed by concatenating such optimal chunks, while the **Pareto front** contains the corresponding objective vectors.

2.2.3 • Cardinality and Justification of Pareto Optimality

Since each chunk has $n' + 1$ possible Pareto optimal configurations, and there are $\frac{m}{2}$ independent chunks, the cardinality of both the **Pareto set** and **Pareto front** is:

$$\left(\frac{2n}{m} + 1\right)^{\frac{m}{2}}.$$

A solution is **Pareto optimal** because any deviation from the form $1^i 0^{n'-i}$ introduces an inefficient bit (a misplaced 0 or 1), which would allow another solution to dominate it by improving one objective without sacrificing the other. Since each chunk is independent, the overall Pareto set is the Cartesian product of all chunk-level Pareto sets.

2.2.4 • Time Complexity Analysis of Non-Dominated Sorting

To implement non-dominated sorting, we use an iterative approach that assigns individuals into non-dominated fronts based on dominance relationships. Given a population of size $N \in \mathbb{N}_{\geq 1}$, the objective is to maintain a total runtime complexity of $O(N^2)$, assuming constant time per comparison.

1. **Outer While Loop:** Runs until all individuals are assigned to a front. If half of the individuals are removed per iteration, the number of iterations is approximately $O(\log N)$.
2. **Inner Loops per Iteration:**
   - **Finding Non-Dominated Elements:** Iterates over the population, costing $O(m)$ per iteration, where m is the remaining number of individuals. The total across all iterations is:
   $$N + \frac{N}{2} + \frac{N}{4} + \cdots = O(N).$$
   - **Updating Remaining Dominations:** For each removed individual, updates are performed on remaining elements. If about half of the population is removed per iteration, the total work follows a geometric series:
   $$\frac{N^2}{4} + \frac{N^2}{16} + \frac{N^2}{64} + \ldots.$$
   The sum of this series is bounded by:
   $$O(N^2).$$
3. **Dictionary Operations:** Dictionary deletions and key iterations in Python operate in $O(1)$ on average, contributing no additional logarithmic factor.

Although the while loop may run $O(\log N)$ iterations if half the elements are removed per step, the work per iteration decreases geometrically. The dominant cost arises from the nested loops inside the while loop, forming a geometric series summing to $O(N^2)$.

2.3 Task 4

2.3.1 • Implementation of Crowding Distance

In the main codebase, the *crowding distance* calculation is implemented in the method `crowding_distance` (or `compute_crowding_distance` for the modified ngsa), located in the `nsga_ii.py` module. This function plays a crucial role in the NSGA-II algorithm by evaluating the density of solutions and promoting diversity in the population.

The main computational effort arises from sorting the population for each objective, leading to a complexity of $O(mN \log N)$, where m is the number of objectives and N is the population size. The subsequent distance calculations add an extra $O(mN)$ complexity. Thus, the total complexity of `crowding_distance` remains:

$$O(mN \log N),$$

which aligns with the expected performance requirements.

2.4 Tasks 5, 6 & 7

The implementation and evaluation of the *NSGA-II* algorithm, as required in Tasks 5, 6, and 7, involve analyzing its performance on the *mLOTZ* benchmark function. These tasks focus on assessing how well the algorithm approximates the Pareto front, its convergence behavior, and the impact of a modified crowding distance strategy. To maintain clarity and avoid redundancy, the results (including experimental observations and performance metrics) will be thoroughly discussed in section 3.

For which concerns the implementation, we have created the NSGA-II classes (**NSGA_II_Optimized** and **NSGA_II_Modified**), which serve as the orchestrators of the evolutionary multi-objective optimization process. They integrate all the components introduced above: individuals, objective-value constructors, and the various benchmark functions. Moreover, they implement the main loop of the algorithm: generating an initial population, performing mutation, applying non-dominated sorting, and selecting the new population based on crowding distance. By doing so, the NSGA-II classes embody the core evolutionary strategy of repeated variation and selection, leveraging the infrastructure set up by **Individual**, **ObjectiveValue**, and the specialized data structures like the **BinaryHeap**. Their design makes it straightforward to swap in different objective-value constructors or adjust the mutation strategy, while still adhering to the canonical NSGA-II workflow.

Additionally, for Task 7, we had to implement the **BinaryHeap** class, which is a key component of the modified crowding-distance selection. Its role is to store individuals (or their identifiers) keyed by a priority (for example, a crowding-distance value). It also relies on each **Individual** being consistently identifiable (through a unique ID) and on our ability to update priorities or remove elements as needed.

3
RESULTS

3.1 Pareto Front Coverage

In figure 1, we plot the average fraction of the Pareto front covered for different problem sizes n in 4LOTZ, using a maximum of $9n^2$ generations. Each of the 30 runs uses a distinct seed, ensuring independent trials. These results can be reproduced by running our `plot_pareto_coverage` function with an initial seed of 42.

We observe that the modified version of the NSGA-II (Task 7) consistently outperforms the original algorithm (Task 6). The reason for this improvement lies in how the modified version updates crowding distances each time an individual is removed. In the original version, the crowding distances were fixed at the time of selection, so the algorithm could inadvertently discard individuals who were actually promising. When an individual is removed, its neighbours effectively become more "isolated" (in terms of objective space), but this change was not accounted for in the original crowding distance calculation. Hence, two near-optimal individuals might both be discarded if they happen to be close to each other, whereas it would be better to keep one. In the modified version, re-computing crowding distances after each removal prevents such missteps, thereby yielding better coverage.

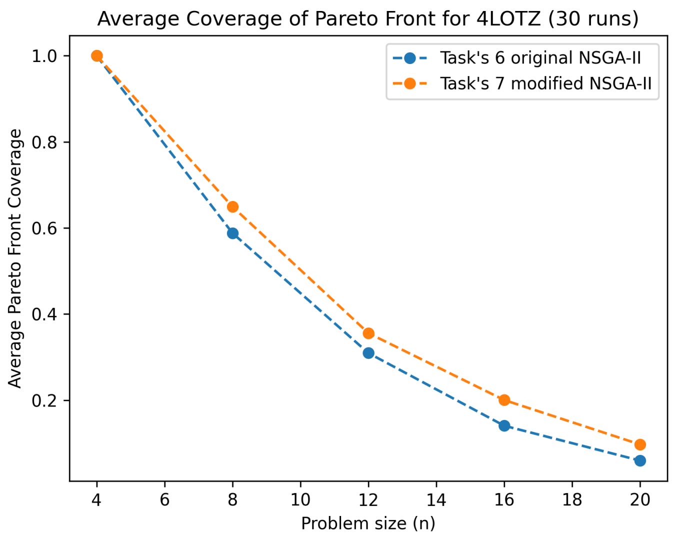

Figure 1. Average pareto front coverage for 4LOTZ

Still, the improvement in performance is not extremely large. The crowding distance is merely a tie-breaking mechanism: non-dominated sorting is the main criterion. For $m = 4$, there are typically more ranks and fewer ties, which narrows the impact of crowding distance. Consequently, the curves for the two approaches remain fairly close, as shown in 1.

We also notice that both algorithms' performance deteriorates for larger n. As n increases, the problem becomes more complex, and $9n^2$ generations may no longer suffice to thoroughly explore the search space. Given the run-time constraints, we did not further increase the maximum number of generations. Our data suggest that the coverage could continue improving if more generations were allowed.

Moreover, in figure 2 we plot the average Pareto coverage for 2LOTZ, again using $9n^2$ as the iteration limit. In this scenario, the modified NSGA-II generally achieves complete coverage of the Pareto front—even at higher n. It often converges well before the iteration limit. By contrast, the original NSGA-II (Task 6) struggles once n becomes larger (notably beyond $n \geq 12$), failing to maintain the entire front. The sequential tie-breaking on updated crowding distances plays a much bigger role when $m = 2$, because there are fewer ranks and thus more ties in each rank. The enhanced crowding-distance update ensures that individuals are more optimally spaced, whereas the original algorithm may repeatedly discard promising solutions that happen to be neighbours, thereby stalling progress toward full coverage.

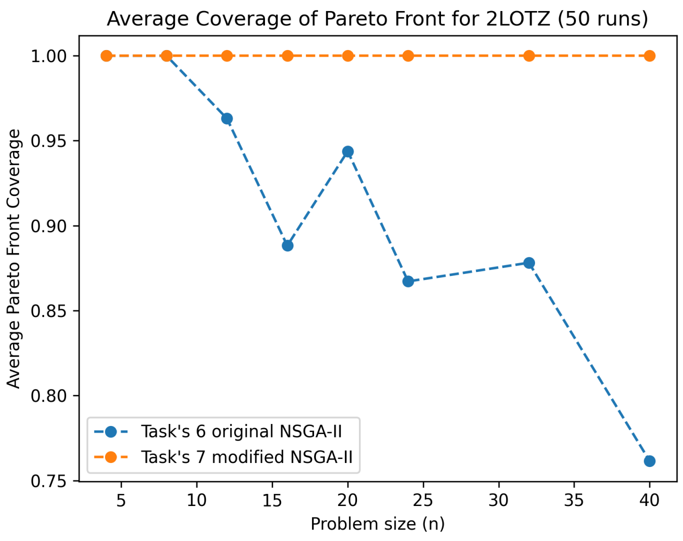

Figure 2. Average pareto front coverage for 2LOTZ

3.2 Run time analysis

Despite the extra overhead of re-computing the crowding distance for $2m$ neighbours each time an individual is removed, the modified NSGA-II still meets the required time complexity. Since only a constant number of individuals (at most $2m$) are updated per removal, the overall cost per iteration remains bounded. Empirically, while the modified version exhibits a somewhat higher per-iteration cost, both approaches share a similar growth rate in runtime as n increases, and thus the same asymptotic complexity. Figure 3 illustrates this observation, showing that although the modified version runs more slowly in absolute terms, it does not diverge from the original method in Big-O terms.

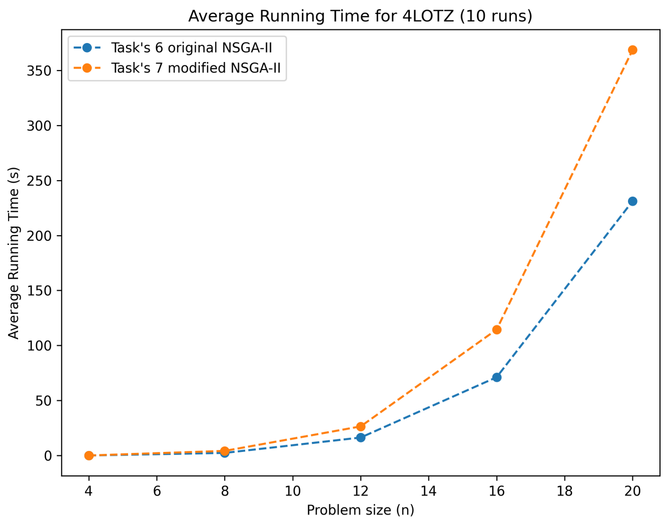

Figure 3. Average runing time for 4LOTZ

4 CONCLUSION

Our results demonstrate that the modified NSGA-II (Task 7) consistently achieves better Pareto front coverage than the original algorithm (Task 6). This improvement is due to the dynamic crowding distance update, which prevents the premature removal of promising individuals and ensures better diversity in the population.

For $m = 4$ (Figure 1), the difference between the two approaches is relatively small since non-dominated sorting remains the dominant selection mechanism. However, for $m = 2$ (Figure 2), where fewer ranks lead to more frequent tie-breaking, the modified algorithm significantly improves performance, achieving full Pareto front coverage even for larger values of n.

In terms of computational efficiency, the additional cost of updating the crowding distance per removal remains bounded, ensuring that the modified NSGA-II maintains the expected $O(mN \log N)$ complexity. As shown in Figure 3, while the modified version exhibits a slightly higher per-iteration runtime, it does not diverge from the original algorithm in terms of asymptotic complexity.

Nonetheless, both versions of NSGA-II struggle as n increases, primarily due to the fixed iteration limit of $9n^2$. Our data suggest that increasing the number of generations could further enhance coverage, particularly for larger problem instances.

Overall, our findings highlight the advantages of adaptive crowding distance updates, particularly in scenarios with a high density of solutions in the Pareto front, while confirming that the modification preserves the algorithm's efficiency.

REFERENCES

[1] 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.

[2] M. Laumanns, L. Thiele, and E. Zitzler, "Running time analysis of multiobjective evolutionary algorithms on pseudo-boolean functions," *IEEE Transactions on Evolutionary Computation*, vol. 8, pp. 170–182, 2004.

[3] W. Zheng and B. Doerr, "Mathematical runtime analysis for the non-dominated sorting genetic algorithm ii (nsga-ii)," *Artificial Intelligence*, vol. 325, p. 104016, 2023.

[4] ——, "Overcome the difficulties of nsga-ii via truthful crowding distance with theoretical guarantees," *CoRR*, vol. abs/2407.17687, 2024.