HADI: Multi-Objective Test Case Prioritization Framework Using NSGA-II Algorithm

Abdullahi Hussein
Department of Computing
JKUAT, Nairobi-Kenya

Dennis Kaburu Mugambi
Department of Information Technology
JKUAT, Nairobi-Kenya

Isaac Nyabisa Oteyo
Department of Information Technology
JKUAT, Nairobi-Kenya

***Abstract*—Test case prioritization (TCP) addresses the challenge of efficiently ordering test cases to improve fault detection rates while managing execution time constraints in regression testing. This study presents *HADI* framework for multi-objective test case prioritization using Non-dominated Sorting Genetic Algorithm II (NSGA-II). The study evaluates the framework using the `Defects4J` benchmark. Our framework optimizes two competing objectives: (i) minimizing the execution time to fault detection (ETAF), and (ii) maximizing the Average Percentage of Faults Detected (APFD). The findings show that HADI achieves a mean APFD of 0.98 compared to 0.51 for original orderings, representing a 92% improvement. Statistical validation using Wilcoxon signed-rank tests confirms significance ($p < 0.001$) with large effect sizes (Vargha-Delaney $A_{12} > 0.89$). Convergence analysis shows that HADI typically reaches optimal solutions within 10-20 generations, with consistent performance across 30 independent runs per dataset.**

***Index Terms*—Test case prioritization, NSGA-II, multi-objective optimization, Defects4J, Pareto optimization.**

I. Introduction

As software development evolves toward agile methodologies and continuous integration/continuous deployment (CI/CD) practices, test suites grow substantially in size and complexity. Running complete test suites after every code change can consume significant time and computational resources, creating bottlenecks in software development workflows [1]. Test case prioritization (TCP) offers a solution by reordering test cases to improve fault detection effectiveness within resource constraints [2]. The fundamental challenge in TCP is to balance multiple competing objectives, i.e., to detect faults as early as possible while minimizing the total execution time. This inherently multi-objective problem requires sophisticated optimization approaches that can explore trade-offs between conflicting goals. Multi-objective evolutionary algorithms (MOEAs), such as Non-dominated Sorting Genetic Algorithm II (NSGA-II), have emerged as promising techniques for TCP optimization. The NSGA-II algorithm is particularly well suited for this domain due to its efficient Pareto front generation and diversity preservation mechanisms [3]. However, comprehensive empirical evaluation of NSGA-II for TCP on standardized benchmarks remains limited in the literature.

This study addresses this gap by proposing and presenting the *HADI* framework based on NSGA-II for multi-objective TCP. The framework provides a complete automated pipeline from data extraction to optimized test orderings with minimal manual intervention. Secondly, this study presents the empirical evaluation of the proposed framework for multi-objective TCP using the complete `Defects4J` benchmark[1] and the insights drawn from the evaluation. The benchmark covers 830 faulty versions across 17 projects. The evaluation includes statistical validation, comparison with multiple baselines, and detailed convergence analysis. The remainder of the paper is organized as follows. Section II presents a review of the literature. Section III describes our experimental approach. Section IV presents the findings and their discussion. Lastly, Section V presents the conclusion and provides directions for future work.

II. Related Work

In this section, we review traditional TCP approaches, search-based TCP, multi-objective TCP, and lastly `Defects4j` in TCP research.

Traditional TCP Approaches: Early TCP research focused on greedy algorithms that prioritize test cases based on coverage criteria. Rothermel *et al.* [2] established foundational TCP strategies using statement and branch coverage metrics. Although computationally efficient, these approaches often fail to consider multiple objectives simultaneously and may converge to suboptimal local solutions.

Search-Based TCP: Search-based software engineering has been successfully applied to TCP challenges. Li *et al.* [4] demonstrated that genetic algorithms could outperform greedy methods by exploring broader solution spaces. Single-objective evolutionary approaches typically optimize individual metrics like APFD but do not explicitly handle trade-offs between competing objectives such as execution time and fault detection rate.

Multi-Objective TCP: The inherently multi-objective nature of TCP has led to increased interest in MOEAs. For instance, Epitropakis *et al.* [5] conducted early evaluations of Pareto-efficient algorithms for TCP, showing promising results for multi-objective approaches. Recent systematic reviews by Bajaj and Sangwan [6] confirm the growing adoption of MOEAs in TCP research. Additionally, more recent work has expanded the range of objectives considered in TCP optimization. Chen *et al.* [7] introduced fault sensitivity indices, while other researchers have incorporated factors such as test

[1] https://github.com/rjust/defects4j

execution cost, historical failure patterns, and code change information [8]. However, comprehensive empirical studies covering all `Defects4J` projects with rigorous statistical validation remain limited.

Defects4J in TCP Research: The `Defects4J` benchmark has become increasingly important for TCP evaluation due to its collection of real-world bugs and reproducible experimental conditions [9]. Recent studies have used `Defects4J` for TCP evaluation [10], although most focus on single-objective optimization or limited project subsets.

III. Methodology

Figure 1 shows the architecture of our proposed framework *HADI*. The framework optimizes two competing objectives that simultaneously minimize execution time to detect all faults (ETAF) and maximize fault detection rate (APFD). It operates in three phases as follows.

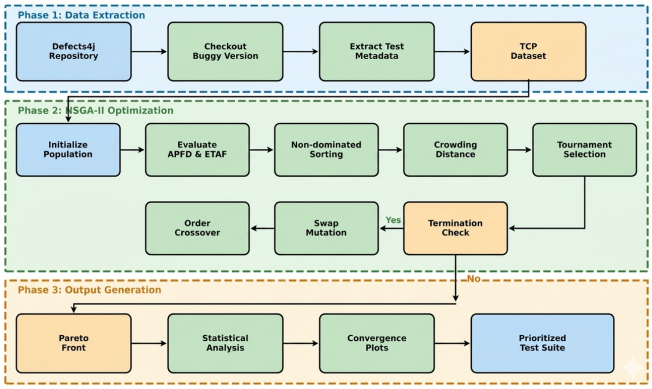

Fig. 1: HADI Framework architecture for multi-objective TCP.

1) Phase 1: Data Extraction: HADI extracts test case metadata from the `Defects4J` benchmark repository [9]. For each buggy program version, the framework collects: (i) test case identifiers, (ii) execution times, (iii) pass/fail status, and (iv) fault detection mappings. These data form the TCP dataset used for optimization.

2) Phase 2: NSGA-II Optimization: The core of HADI employs NSGA-II [3] to evolve test case orderings. Each individual in the population represents a complete permutation of test cases. The evolutionary process involves the following.

- *Fitness Evaluation*: Computing APFD and ETAF for each ordering
- *Non-dominated Sorting*: Ranking solutions into Pareto fronts.
- *Crowding Distance*: Maintaining diversity within fronts.
- *Selection*: Tournament selection based on rank and crowding.
- *Reproduction*: Order crossover and swap mutation operators.

3) Phase 3: Output Generation: Upon termination, HADI extracts the Pareto front containing non-dominated solutions representing optimal trade-offs between APFD and ETAF. Statistical analysis validates the results on 30 independent runs per dataset.

A. Implementation

We implemented the HADI framework using the Distributed Evolutionary Algorithms in Python (DEAP) library [11]. The library provides efficient implementations of evolutionary algorithm components. Each individual in the population represents a complete test case ordering, encoded as a permutation of test indices.

1) Objective Functions: The multi-objective optimization targets two competing goals as follows.

a) Objective 1: Minimize Execution Time to All Faults (ETAF): This objective minimizes the cumulative execution time required to detect all known faults for a test ordering (π) and a set of faults (F) as shown in Equation (1) [12].

$$ETAF(\pi) = \sum_{i=1}^{k} t_{\pi(i)} \tag{1}$$

where k is the position of the last test that detects a previously undetected fault, and $t_{\pi(i)}$ is the execution time of the test at position i in ordering π.

b) Objective 2: Maximize Average Percentage of Faults Detected (APFD): APFD measures how quickly faults are detected during test execution as shown in Equation (2) [2].

$$APFD = 1 - \frac{\sum_{i=1}^{m} TF_i}{n \times m} + \frac{1}{2n} \tag{2}$$

where m is the number of faults, n is the total number of test cases, and TF_i is the position of the first test that detects fault i. Higher APFD values indicate better early fault detection.

2) Genetic Operators: We used permutation-specific genetic operators as follows.

- *Order Crossover (OX)*: Preserves relative ordering of elements while combining genetic material from two parents.
- *Swap Mutation*: Randomly exchanges the positions of two test cases in the ordering.

3) Framework Configuration: Based on preliminary experiments and literature recommendations, we configured HADI with the parameters shown in Table I [5], [13].

TABLE I: Configuration parameters.

Parameter	Value
Population size	100
Generations	50
Crossover probability	0.9
Mutation probability	0.1
Selection method	Tournament with crowding distance
Independent runs	30 per dataset

B. Evaluation

For evaluation, we built an automated pipeline to extract TCP-relevant data from all 17 `Defects4J` projects. For each faulty version, the pipeline collects test execution times, pass/fail outcomes, and fault detection information. Figure 1 shows the three phases of the framework. The second and third phases run NSGA-II optimization over 50 generations and extract the final Pareto front respectively, repeated 30 times per dataset with median values reported to account for algorithmic stochasticity. Table II shows the characteristics of the data set in all 17 projects. The pre-processing pipeline executes the following steps for each faulty version.

- *Repository checkout*: Automated checkout of the specific buggy version from the `Defects4J` repository.
- *Test suite execution*: Execution of all `JUnit` tests with timing instrumentation and JVM warm-up to ensure accurate measurements.
- *Data collection*: Capture of test results (pass/fail) and precise execution times.
- *Fault identification*: Recording of tests that detect each fault based on `Defects4J` trigger test information.
- *Dataset generation*: Creation of standardized CSV files containing test identifiers, execution times, and failure status.

TABLE II: `Defects4J` project coverage.

Project	Versions	Avg Tests	Domain
Closure	174	7,927	JS compiler
JacksonDatabind	110	308	JSON processing
Math	106	243	Math library
Jsoup	93	142	HTML parser
Lang	61	113	Java utilities
Compress	47	156	Compression
Cli	39	94	CLI parsing
Mockito	38	278	Testing framework
Chart	26	426	Charting
Time	26	167	Date/time
JacksonCore	26	215	JSON core
JxPath	22	24	XPath
Gson	18	88	JSON serialization
Codec	18	198	Encoders
Csv	16	54	CSV processing
JacksonXml	6	56	XML processing
Collections	4	89	Data structures
Total	**830**	–	–

1) Baseline Methods: We compared the HADI framework with four baseline approaches that represent different prioritization strategies.

- *Original ordering*: Default test sequence as defined in the `Defects4J` repositories. This represents the ordering that developers would use without any prioritization.
- *Random ordering*: Test cases executed in random sequence, averaged over 30 runs to account for variance. This provides a lower-bound baseline.
- *Greedy-Time*: Tests ordered by execution time (shortest first). This strategy minimizes execution time, but ignores fault detection.
- *Greedy-Fault*: Failing tests placed first, then remaining tests ordered by execution time. This oracle-based strategy provides an upper bound on single-ordering performance. While both HADI and Greedy-Fault require full test execution results, they differ fundamentally: Greedy-Fault produces one deterministic ordering, whereas HADI generates a diverse Pareto front enabling practitioners to select trade-offs between APFD and ETAF based on project-specific constraints.

2) Statistical Validation: We used the Wilcoxon signed-rank test [14] to compare paired samples from HADI and the baseline methods. A `p-value` below 0.05 indicates a statistically significant difference between the two methods being compared. To measure the magnitude of the differences, we calculate the size of the Vargha-Delaney effect (A_{12}) [15]. This metric represents the probability that a randomly selected result from the HADI framework will be better than a randomly selected result from a baseline method. We interpret A_{12} values as follows: (i) $A_{12} < 0.44$ indicates that the baseline is better, (ii) $0.44 \leq A_{12} \leq 0.56$ indicates a negligible difference, (iii) $0.56 < A_{12} \leq 0.71$ indicates a medium effect, and (iv) $A_{12} > 0.71$ indicates a large effect in favor of HADI. Following the guidelines for the evaluation of the randomized algorithm [16], we performed 30 independent executions per data set to account for the stochastic nature of HADI and to allow meaningful statistical analysis.

3) Number of Runs: As mentioned above, we executed each data set 30 times with different random seeds as recommended in the literature to obtain reliable statistics [16].

4) Evaluation Environment: We executed all experiments on a machine with an `Intel i7-8750H` processor and `16GB RAM` running Ubuntu on WSL. Before measuring test execution times, we applied JVM warm-up to get consistent readings.

IV. Results and Discussion

This section presents the findings of our evaluation of the 830 faulty versions in all 17 `Defects4J` projects.

A. Overall Performance of HADI

Table III summarizes the performance of HADI in all 17 projects compared to the original orderings. Figure 2 visualizes the Pareto fronts and the convergence behavior for nine representative projects. The findings show that HADI achieved a mean APFD of **0.98** (Std Dev: 0.02) compared to **0.51** (Std Dev: 0.25) for the original orderings, representing a **92% improvement**. This improvement was consistent in all 17 projects, ranging from 80% (Closure) to 111% (Gson).

B. Comparison with Baseline Methods

Table IV presents the comparison of HADI against all four baseline methods. The comparison reveals the following findings.

- *HADI vs Original*: 92% APFD improvement with 164× faster fault detection (0.15s vs 24.6s average ETAF).

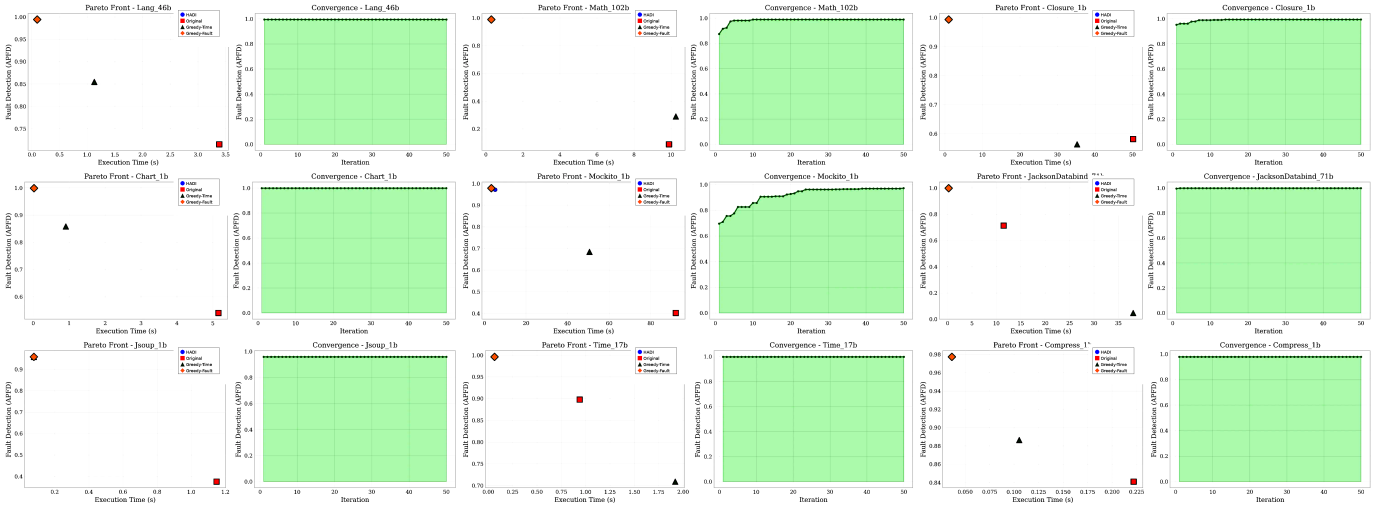

Fig. 2: Pareto fronts and convergence behavior across 9 representative `Defects4J` projects. Left panels show the trade-offs between execution time (ETAF) and fault detection (APFD). Right panels show the convergence over 50 generations.

TABLE III: HADI APFD performance by project.

Project	**N**	**HADI**	**Original**	**Improv.**
Closure	174	0.99	0.55	+80%
JacksonDatabind	110	0.99	0.52	+90%
Math	106	0.98	0.51	+92%
Jsoup	93	0.98	0.53	+85%
Lang	61	0.99	0.50	+98%
Compress	47	0.98	0.53	+85%
Cli	39	0.97	0.48	+102%
Mockito	38	0.98	0.49	+100%
Chart	26	0.98	0.52	+88%
Time	26	0.98	0.52	+88%
JacksonCore	26	0.99	0.54	+83%
JxPath	22	0.97	0.48	+102%
Gson	18	0.99	0.47	+111%
Codec	18	0.98	0.51	+92%
Csv	16	0.98	0.50	+96%
JacksonXml	6	0.98	0.51	+92%
Collections	4	0.97	0.49	+98%
Overall	**830**	**0.98**	**0.51**	**+92%**

- *HADI vs Random*: 100% APFD improvement (0.98 vs 0.49) with substantially lower variance.
- *HADI vs Greedy-Time*: 88% APFD improvement. The Greedy-Time achieves only 0.52 APFD despite optimizing execution time. This demonstrates that time minimization alone is insufficient for effective TCP.
- *HADI vs Greedy-Fault*: Both achieve comparable APFD of 0.98. Greedy-Fault serves as an oracle upper bound, and HADI successfully matches this performance while providing diverse Pareto-optimal solutions.

C. Statistical Validation

Table V presents results for the Wilcoxon signed-rank tests and effect size analysis for all 830 datasets. The statistical analysis confirms the following.

- HADI significantly outperforms Original, Random, and Greedy-Time ($p < 0.001$).

TABLE IV: Comparison with baseline methods.

Method	**APFD**	**ETAF (s)**	**Std Dev**
HADI	**0.98**	**0.15**	0.02
Greedy-Fault	**0.98**	**0.15**	0.02
Original	0.51	24.6	0.25
Random	0.49	21.3	0.18
Greedy-Time	0.52	18.4	0.28

- Large effect sizes ($A_{12} > 0.89$) indicate substantial practical significance.
- There are no significant differences with Greedy-Fault ($p = 0.847$, $A_{12} = 0.50$), confirming that HADI achieves oracle-level performance.

TABLE V: Statistical validation results (Wilcoxon signed-rank test on APFD distributions, 30 runs $\times$ 830 datasets; A_{12} = Vargha-Delaney effect size).

Comparison	**p-value**	**A_{12}**	**Effect**
HADI vs Original	<0.001	0.91	Large
HADI vs Random	<0.001	0.92	Large
HADI vs Greedy-Time	<0.001	0.89	Large
HADI vs Greedy-Fault	0.847	0.50	Negligible

D. Convergence Analysis

Figure 3 illustrates the HADI framework convergence behavior for nine representative projects. The analysis shows the evolution of the best APFD values over 50 generations with 30 independent runs.

The convergence analysis reveals the following.

- *Rapid convergence*: Most datasets achieve near-optimal APFD within 10–20 generations. The convergence curves may appear nearly flat when plotted over all 50 generations precisely because HADI converges quickly; the substantive optimization occurs in the early generations.

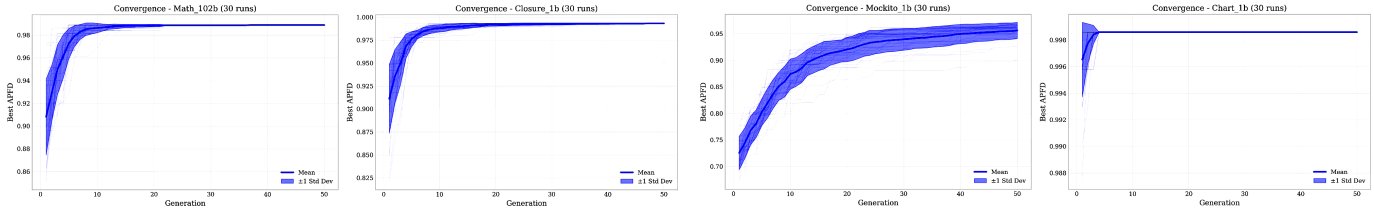

Fig. 3: Convergence analysis across 4 representative `Defects4J` projects (30 independent runs each). Blue lines show mean best APFD per generation; shaded regions indicate ± 1 standard deviation.

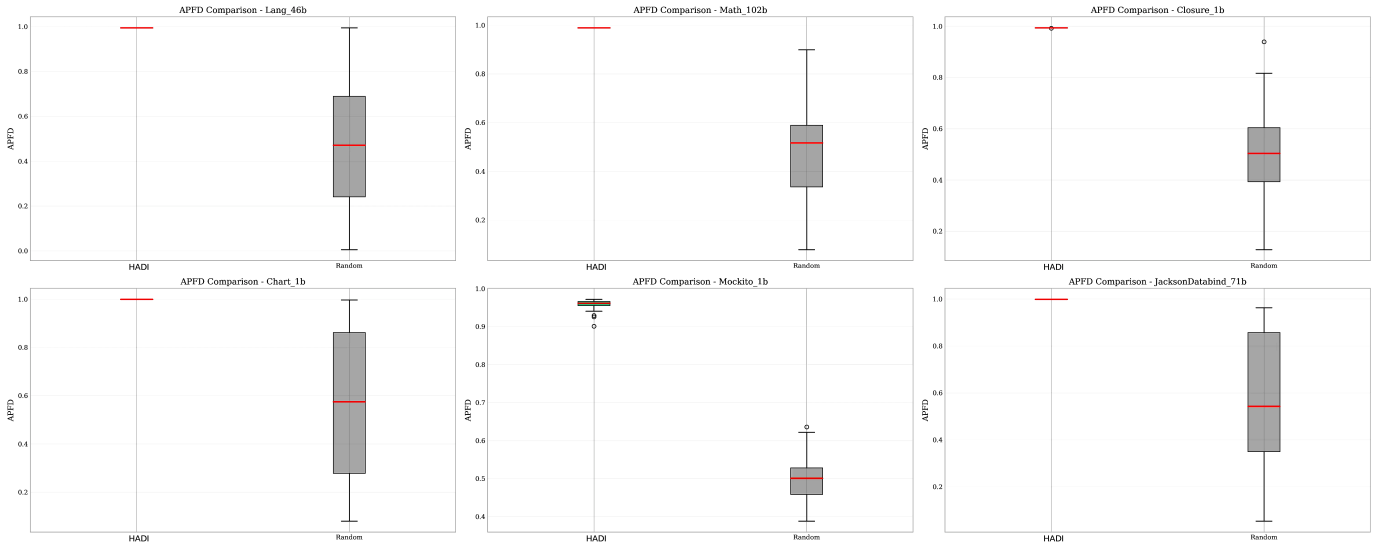

Fig. 4: APFD distribution comparison: HADI vs Random ordering across 6 representative `Defects4J` projects (30 runs each).

- *Complex landscapes*: Datasets with multiple faults (e.g., Mockito_1b with 12 faults) show gradual improvement from APFD 0.65 to 0.98 over 50 generations.
- *Low variance*: The narrow confidence bands indicate consistent performance across independent runs.
- *No premature convergence*: The algorithm continues to improve throughout the optimization process for complex cases.

E. APFD Distribution Analysis

Figure 4 presents the box plots that compare the APFD distributions between HADI and Random orderings for six representative projects.

The box plot analysis demonstrates the following.

- *Consistent HADI performance*: Near-zero variance with median APFD of 0.98 for the nine projects.
- *High Random variance*: Inter-quartile range of 0.25-0.70 indicates unpredictable performance.
- *No outliers for HADI*: All runs achieve similar high-quality solutions.
- *Project-independent reliability*: HADI maintains consistent performance regardless of project characteristics.

F. Execution Time

Table VI presents the comparison of the execution time of all faults (ETAF) for representative datasets across different projects. HADI achieves an average **97.5% reduction** in execution time to detect all faults compared to original orderings, with consistent improvements across all projects.

TABLE VI: ETAF comparison for representative datasets.

Dataset	**Original ETAF (s)**	**HADI ETAF (s)**	**Reduction**
Closure_1b	50.2	1.10	97.8%
Mockito_1b	89.4	3.20	96.4%
JacksonDatabind_71b	11.5	0.21	98.2%
Lang_46b	03.4	0.12	96.5%
Math_102b	08.7	0.15	98.3%
Chart_1b	15.3	0.34	97.8%
Time_17b	12.1	0.28	97.7%
Jsoup_1b	06.8	0.18	97.4%
Compress_1b	09.2	0.22	97.6%
Average	**22.9**	**0.64**	**97.5%**

The evaluation of the 830 faulty versions reveals the following insights.

- *Consistent High Performance*: The HADI framework consistently achieves near-optimal APFD (0.98) for all 17 projects, regardless of the project size, domain, or fault characteristics. This consistency is evident in the box plot analysis (Figure 4) showing minimal variance.
- *Practical Significance*: The 92% improvement over original orderings and 97.5% reduction in fault detection

time have substantial practical implications for CI/CD environments where rapid feedback is critical.
- *Greedy-Time Inadequacy*: Poor performance of the Greedy-Time (0.52 APFD) demonstrates that execution time minimization alone is insufficient for effective TCP.
- *Oracle-Level Achievement*: HADI matches the oracle-based Greedy-Fault baseline, indicating that the evolutionary approach successfully discovers optimal or near-optimal solutions without requiring a priori knowledge of fault locations.
- *Statistical Robustness*: Large effect sizes ($A_{12} > 0.89$) and significant p-values ($p < 0.001$) for the 830 datasets confirm the robustness and generalizability of our findings.

G. Comparison with Prior Work

Our results align with and extend prior TCP research in the following ways.

- Epitropakis *et al.* [5] reported APFD improvements of 15-30% in smaller datasets; we observed a 92% improvement in 830 datasets.
- Our coverage of all 17 `Defects4J` projects exceeds previous studies that typically evaluated 2-5 projects.
- The 30 independent runs per data set provide stronger statistical validation than typical 10-run protocols.

H. Threats to Validity

- *Internal validity*: We mitigated random variation through 30 independent runs per dataset and rigorous statistical testing. JVM warm-up was applied to reduce timing measurement noise.
- *External validity*: Our evaluation spans 17 diverse Java projects totaling 830 faulty versions, representing the largest study on `Defects4J`. The experiments were conducted using Oracle JDK for compatibility with the original `Defects4J` configuration. However, HADI's prioritization logic is JDK-agnostic and does not depend on Oracle-specific features, so the framework can generalize to OpenJDK or other Java implementations.
- *Construct validity*: We used standard metrics (APFD, ETAF) and established statistical tests (Wilcoxon signed-rank, Vargha-Delaney A_{12}) to ensure a fair comparison with previous studies. We acknowledge that Pareto front quality indicators such as Hypervolume (HV) and Inverted Generational Distance (IGD) were not reported; including these in future work would further characterize HADI's multi-objective effectiveness.

V. Conclusion

This study presents a comprehensive empirical evaluation of HADI for multi-objective test case prioritization across 830 faulty versions from all 17 `Defects4J` projects. Our results demonstrate that the framework proposed and presented in this study achieves 0.98 APFD compared to 0.51 for original orderings (92% improvement), with statistical significance confirmed by Wilcoxon tests ($p < 0.001$) and large effect sizes ($A_{12} > 0.89$). The consistent performance across diverse project types, combined with the 97.5% reduction in fault detection time, demonstrates HADI's practical value for modern CI/CD environments. Future work will explore adaptive parameter tuning based on dataset characteristics, integration with CI/CD systems for incremental re-prioritization.

Acknowledgement

The authors acknowledge the School of Computing & IT at JKUAT. We also acknowledge the anonymous reviewers for their valuable feedback.

References

[1] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test Case Prioritization: A Family of Empirical Studies," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 159–182, 2002.

[2] G. Rothermel, R. Untch, C. Chu, and M. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[3] 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.

[4] Z. Li, M. Harman, and R. M. Hierons, "Search Algorithms for Regression Test Case Prioritization," *IEEE Transactions on Software Engineering*, vol. 33, no. 4, pp. 225–237, 2007.

[5] M. G. Epitropakis, S. Yoo, M. Harman, and E. K. Burke, "Empirical evaluation of pareto efficient multi-objective regression test case prioritisation," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, (New York, NY, USA), p. 234–245, Association for Computing Machinery, 2015.

[6] A. Bajaj and O. P. Sangwan, "A Systematic Literature Review of Test Case Prioritization Using Genetic Algorithms," *IEEE Access*, vol. 7, pp. 126355–126375, 2019.

[7] J. Chen, Y. Gu, S. Cai, H. Chen, and J. Chen, "KS-TCP: An Efficient Test Case Prioritization Approach based on K-medoids and Similarity," in *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 105–110, 2021.

[8] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino, "FAST approaches to scalable similarity-based test case prioritization," in *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, (New York, NY, USA), p. 222–232, Association for Computing Machinery, 2018.

[9] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: a database of existing faults to enable controlled testing studies for Java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, (New York, NY, USA), p. 437–440, Association for Computing Machinery, 2014.

[10] Q. Luo, K. Moran, L. Zhang, and D. Poshyvanyk, "How Do Static and Dynamic Test Case Prioritization Techniques Perform on Modern Software Systems? An Extensive Study on GitHub Projects," *IEEE Transactions on Software Engineering*, vol. 45, no. 11, pp. 1054–1080, 2019.

[11] F. Fortin, F. De Rainville, M. G. Gardner, M. Parizeau, and C. Gagné, "DEAP: evolutionary algorithms made easy," *Journal of Machine Learning Research*, vol. 13, p. 2171–2175, July 2012.

[12] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "TimeAware Test Suite Prioritization," in *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, pp. 1–12, 2006.

[13] K. Garg and S. Shekhar, "Test Case Prioritization Based on Fault Sensitivity Analysis Using Ranked NSGA-2," *International Journal of Information Technology*, vol. 16, pp. 2875–2881, 2024.

[14] F. Wilcoxon, "Individual Comparisons by Ranking Methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[15] A. Vargha and H. D. Delaney, "A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[16] A. Arcuri and L. Briand, "A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering," in *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 1–10, 2011.