Exploring Multifactorial Techniques in Rat Swarm Optimization: Preliminary Results

Haw Yuan Kang
Fakulti Teknologi Maklumat dan Komunikasi, Universiti Teknikal Malaysia Melaka, Malacca, Malaysia
hawyuankang11@gmail.com

Raja Rina Raja Ikram
Fakulti Teknologi Maklumat dan Komunikasi, Universiti Teknikal Malaysia Melaka, Malacca, Malaysia
raja.rina@utem.edu.my (corresponding author)

Kamal Zuhairi Zamli
Faculty of Computing, Universiti Malaysia Pahang Al-Sultan Abdullah, Pahang, Malaysia
kamalz@umpsa.edu.my

Nurul Akmar Binti Emran
Fakulti Teknologi Maklumat dan Komunikasi, Universiti Teknikal Malaysia Melaka, Malacca, Malaysia
nurulakmar@utem.edu.my

Received: 24 February 2025 | Revised: 7 April 2025 | Accepted: 12 April 2025

 DOI: https://doi.org/10.48084/etasr.10690

ABSTRACT

Test Suite Reduction (TSR) is a critical optimization challenge in software testing that aims to reduce the number of test cases while maintaining maximum requirement coverage. Traditional algorithms, such as the Rat Swarm Optimizer (RSO), struggle with scalability, especially when dealing with large datasets. Additionally, RSO is unable to solve multiple tasks simultaneously, which leads to an increased time to complete the optimization process across multiple datasets. To resolve this constraint, this paper introduces the Multi-Factorial Rat Swarm Optimizer (MFRSO), which combines Multi-Factorial Optimization (MFO) principles to allow knowledge transfer between tasks, thus increasing optimization efficiency. The performance of MFRSO was compared to that of RSO on five datasets of varied sizes, with results averaging over ten runs. Experimental results show that MFRSO consistently delivered a higher Percentage of Test Suite Reduction (PTSR) while maintaining full requirement coverage, as opposed to RSO, which loses efficiency significantly with larger datasets. Furthermore, MFRSO reduced the optimization time compared to RSO, indicating its scalability and reliability. Future work will investigate adaptive knowledge transfer methods and apply MFRSO to dynamic test suite settings.

Keywords-rat swarm optimization; multifactorial optimization; test suite reduction; transfer learning; multitasking

I. INTRODUCTION

Software testing is a phase of software development that ensures the correctness, reliability, and performance of software systems [1]. However, as software becomes more complicated, the number of test cases necessary for complete testing increases dramatically, resulting in significant computing expenses. Test Suite Reduction (TSR) strategies intend to reduce the amount of test cases while maintaining fault detection capability, thus increasing efficiency without sacrificing software quality [2].

Metaheuristic optimization algorithms have been widely used for TSR because of their ability to explore large solution spaces effectively. A systematic review in [3] analyzed 57 relevant articles, revealing that algorithms based on swarm intelligence constitute a significant portion (40%) of search-based TSR techniques. Among these, 91% of swarm intelligence-based approaches addressed single-objective optimization problems, while only 9% tackled multi-objective optimization. The review also found that 67% of TSR research focused on single-objective optimization, while 33% focused on multi-objective optimization. However, very few studies have investigated multitasking strategies, revealing a research gap in Multi-Factorial Optimization (MFO).

A. Rat Swarm Optimizer (RSO)

RSO is a swarm intelligence-based metaheuristic inspired by the foraging behavior of rats, mimicking their ability to

explore and exploit the search space efficiently by balancing exploration and exploitation [4]. The algorithm models the movement of rats within a solution space, where each rat represents a candidate solution. Movement strategies incorporate adaptive position updates based on leader-following mechanisms and local search behaviors to improve convergence toward optimal solutions. Among optimization algorithms, RSO has demonstrated strong performance in solving complex optimization problems in recent years, as shown in Table I. This makes it a suitable foundation for integration with MFO. However, standard single-objective optimization methods may be insufficient when dealing with several tasks simultaneously, such as improving TSR across different software versions or components. RSO has demonstrated effectiveness in solving various optimization problems due to its adaptability and efficient search capabilities. However, in its standard form, RSO is designed primarily for single-objective optimization [4], limiting its applicability in multitasking scenarios such as TSR across multiple software components.

TABLE I. LIST OF RSO APPLIED IN OTHER SCOPES

Reference	Scope
[5]	Covid-19 diagnosis
[6]	Computer vision
[7]	Photovoltaic systems
[8]	Cybersecurity
[9]	Fish species classification

B. Multi-Factorial Optimization (MFO)

MFO is a multitasking optimization framework that enables simultaneous optimization of multiple tasks by exploiting knowledge transfer mechanisms [10]. It leverages the concept of skill factors to associate solutions with specific tasks while allowing for the sharing of knowledge between them. This approach improves search efficiency and overall optimization performance by using intertask dependencies.

Genetic operators, such as crossover and mutation, facilitate knowledge transfer across tasks, ensuring better convergence and solution diversity [11]. This approach is particularly beneficial in scenarios where tasks share common structural properties, making it ideal for TSR in diverse software testing environments. The concept of MFO has also been widely used in other studies, as shown in [12-15]. Therefore, by integrating MFO principles with RSO, MFRSO extends the capabilities of traditional RSO, enabling multitask learning for TSR.

II. PROPOSED ALGORITHM

This study adopted an experimental design to develop and evaluate the proposed MFRSO algorithm, following the research structure outlined in [16-18]. The algorithm integrates RSO principles into MFO to enable simultaneous optimization across multiple tasks. Figure 1 provides an overview of the proposed MFRSO workflow, illustrating the key stages from initialization to solution evaluation. The workflow consists of population initialization, fitness evaluation, ranking mechanism, skill factor assignment, scalar fitness, population update, and evolution.

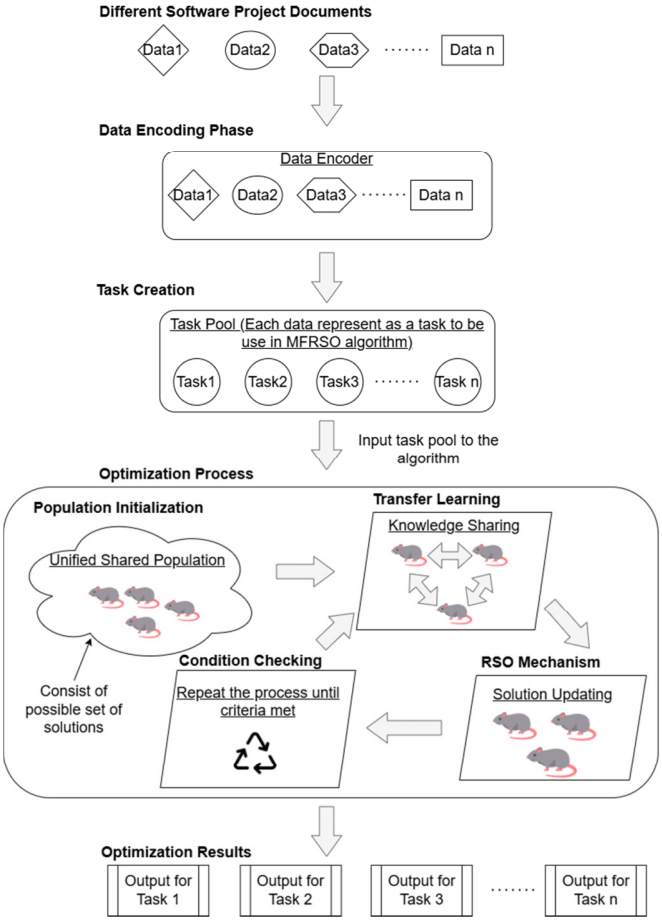

Fig. 1. Multifactorial Rat Swarm Optimizer (MFRSO) workflow.

The key components of the optimization process are explained in the following section.

A. Population Initialization

MFRSO, like most MFO techniques [10, 11], begins by initializing a unified population of candidate solutions (rats). Each rat is randomly assigned an initial test suite subset to form a possible solution. To ensure a fair comparison, the population size is fixed at 50 and the iteration of the optimization process at 100 for all algorithms, a value determined through preliminary experiments. Additionally, during population initialization, at least one solution is guaranteed to achieve 100% requirement coverage for every task. This ensures that the optimization process starts with a feasible solution and avoids premature convergence to suboptimal results.

B. Fitness Evaluation

After population initialization, the fitness of each rat is evaluated across all tasks. The fitness evaluation for each task considers the size of the test suite, penalties for uncovered requirements, and oversized test suites. The evaluation process utilizes the following parameters.

- *taskTestCaseSize*: The original number of test cases in a task.

- *taskTotalReqs*: The total number of requirements in a task.
- *uncoveredReqs*: The number of requirements not covered by the test suite.
- *coveredReqs*: The number of requirements covered by the test suite.
- *testSuiteSize*: The number of test cases in the rat.
- *penalty*: A penalty applied for each uncovered requirement.

$$penalty = (uncoveredReqs) \times 10000$$

- *overSizePenalty*: A penalty applied for each exceed test cases based on *taskTestCaseSize*.

$$overSizePenalty = \left(testSuiteSize - taskTestCaseSize\right) \times 10000$$

A penalty weight of 10000 was chosen to ensure that solutions with uncovered requirements or oversized test suites are heavily penalized, guiding the optimization process toward compact solutions. The overall fitness is calculated as:

$$fitness = testSuiteSize + penalty + overSizePenalty$$

Additionally, the requirements coverage percentage of a rat for the task is computed as:

$$coverPercent = \left(\frac{coveredReqs}{taskTotalReqs}\right) \times 100$$

C. Ranking Mechanism

After evaluating the fitness of each rat for all tasks, the rats are ranked individually for each task based on their fitness values. This process is repeated for all tasks, ensuring that each rat has a unique rank for each one. The ranking process works as follows:

- Sorting: For each task, the rats are sorted in ascending order of fitness (lower fitness values indicate better solutions).
- Rank Assignment: Each rat receives a rank based on its place in the sorted list. The rat with the lowest fitness value earns rank one, followed by rank two, and so on.

D. Skill Factor Assignment

The skill factor assignment mechanism ensures that each rat is assigned a specific task depending on its overall performance. This procedure assigns at least one rat to each task using the criteria listed below. Once each task has at least one assigned rat, the remaining rats are assigned to the task they perform the best.

- Rank Priority: The rat with the lowest rank (rank 1) for the task is prioritized.
- Fitness Consideration: In the case of ties (rats with the same rank), the rat with the lower fitness value is selected.
- Uniqueness Constraint: The selected rat must not already be assigned to another task.

E. Scalar Fitness

To facilitate the optimization process, a scalar fitness value is calculated for each rat. This scalar fitness represents the rat's overall performance across all tasks and is used to guide the selection and evolution of the population. The scalar fitness is calculated as the inverse of the best rank achieved by the rat across all tasks [19].

$$scalarFitness = \left(\frac{1}{bestRank}\right)$$

where the $bestRank$ is the lowest rank achieved by the rat across all tasks. This inverse relationship ensures that rats with lower ranks (better performance) receive higher scalar fitness values, prioritizing them in the optimization process.

F. Population Update and Evolution

The population update process ensures that each rat's solution is refined based on its assigned skill factor. The key steps are as follows:

- Each rat in the population is processed individually, creating a temporary copy of it.
- The copied rat's solution is refined using the RSO update equations (see [4]).
- A temporary population is created, where the copied rat replaces the original rat position.
- The copied rat's solution is evaluated for all tasks, followed by ranking and skill factor assignment.
- The scalar fitness of the copied rat is compared with that of its original counterpart. If it performs better, it replaces the original, and the improvement is recorded. Otherwise, the update is discarded.

After each optimization loop, the updated solution is evaluated and only those leading to improved scalar fitness replace the original. If no improvements occur across all rats, a counter tracks stagnation for potential termination. The optimization process terminates when the loop reaches the maximum iteration of 100 or when the stagnation counter reaches 10. Below is the pseudocode of the optimization process.

```
Algorithm 1: Population Optimization
Process
Input: Population P, Tasks T, Parameters
       A, C
Output: Updated population P with
        optimized solutions
Notations:
  P: Population (Contain a set of possible
     solutions)
  P_i: A rat (solution) in the population
  P_r: Best rat in the population within
       the same skill factor
  c_i: Candidate solution proposed by a rat
  τ_i: Skill factor assigned to a rat,
       representing a specific task it
       specializes in
  ϕ:  Scalar fitness of a rat across all
       tasks
```

```
1. Start Procedure optimizePopulation
2. Initialize CheckImprovement = false
3. for each Pi in P do
3.1  Create a copy of Pi which is Pi'
3.2  Determine the skill factor τi of Pi
3.3  Identify the best rat Pr in P within
     same τi
3.4  Optimize the solution for the Pi'
     using the RSO equation from [4]
3.5  Create temporary population P' with
     Pi' replacing Pi
3.6  Evaluate only the copied rat's
     fitness Pi' for each task in P'
3.7  Rank all rats in P'  for each task
     based on fitness
3.8  Assign τi to all rats based on new
     ranks in P'
3.9  Calculate ϕ for all rats based on new
     ranks in P'
3.10 Compare ϕ Pi' with ϕ Pi
3.11 if ϕ Pi'> ϕ Pi then
3.11.1  Accept Pi'
3.11.2  Set CheckImprovement = true
3.12 else
3.12.1  Reject Pi'
4. End for
5.  if CheckImprovement == false then
5.1   Set noImprovementCount += 1
6.  else
6.1   Set noImprovementCount = 0
7. End Procedure
```

III. RESULTS AND DISCUSSION

This section evaluates the proposed algorithm using standard industry metrics such as Percentage Test Suite Reduction (PTSR) and Percentage of Coverage Achieved (PCOA) [3]. The evaluation focuses on RSO [4] and the proposed MFRSO. Since this is a preliminary evaluation, 10 runs and five datasets were sufficient to capture performance trends, assess scalability, and provide a reasonable evaluation. The largest dataset was retrieved from [20], while the other datasets were generated by mimicking its pattern and structure, with variations in the number of test cases, requirements, and their mappings. By ensuring that these self-generated datasets strictly follow the characteristics of real-world data, researchers can utilize them effectively to improve software quality and perform controlled and scalable experiments [21].

A. PTSR Analysis

Figures 2 and 3 illustrate the PTSR trends of MFRSO and RSO algorithms across 10 independent runs for different dataset sizes. Figure 2 shows the performance of MFRSO, where it maintains a consistently high PTSR, even for larger datasets, with values above 65%. This indicates that MFRSO remains stable and reliable in optimizing complex test suites. In contrast, Figure 3 reveals a significant drop in RSO's PTSR when handling larger datasets. Although RSO performs adequately for smaller datasets, its optimization efficiency decreases sharply for large-scale datasets, with PTSR dropping to only 6-32%. This drop occurs because RSO lacks transfer learning between different optimization tasks, leading to inefficient exploration and premature convergence. Furthermore, since the population size and number of iterations are fixed across all algorithms for a fair comparison, RSO struggles to achieve the best-optimized result within the given number of iterations due to its limited search capability and independent task handling. In contrast, MFRSO, leveraging multifactorial optimization techniques, efficiently transfers information between tasks, allowing knowledge gained from optimizing one dataset to enhance the optimization of others. This enables MFRSO to reach better-optimized results within the same number of iterations, maintaining higher performance even for large-scale datasets.

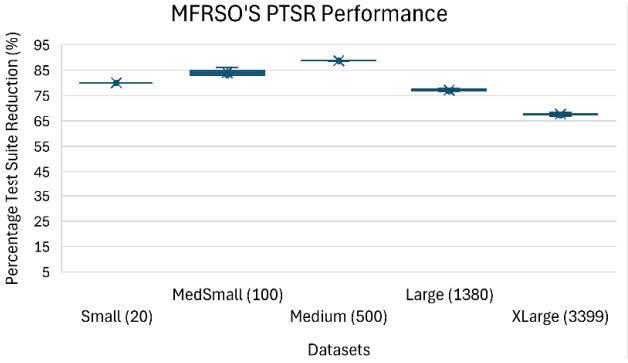

Fig. 2. MFRSO PTSR performance across 10 runs.

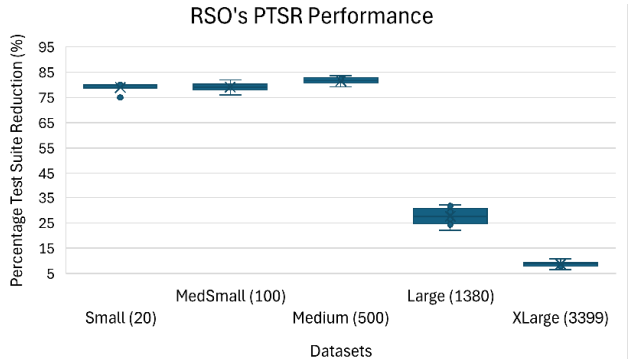

Fig. 3. RSO PTSR performance across 10 runs.

B. PCOA Analysis

Both MFRSO and RSO consistently achieved 100% PCOA across all 10 runs on the five datasets. This is because both algorithms are explicitly designed to enforce full requirement coverage, ensuring the correctness of the test suite selection. In software testing, achieving complete requirement coverage is vital because reducing the test suite size without full coverage would lead to an incomplete and ineffective testing process. Therefore, while minimizing the number of test cases is the primary optimization goal, it must not come at the cost of missing requirements, which would compromise the validity of the testing phase. Figure 4 shows the PCOA performance of both algorithms across all 10 runs for five different dataset sizes.

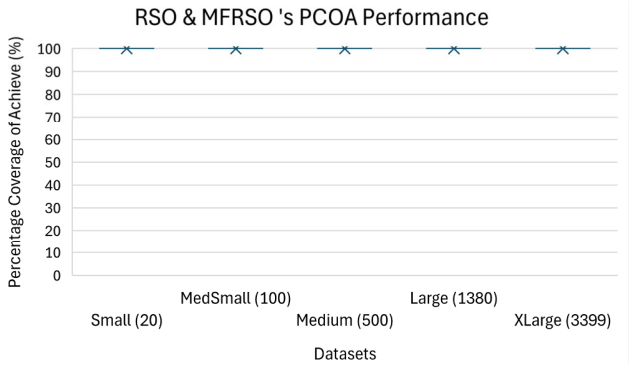

Fig. 4. PCOA performance across 10 runs for RSO and MFRSO

C. Analysis of Time Taken for Execution Process

Figure 5 illustrates the time efficiency of both algorithms, showing the total time taken for the execution process across 10 runs to complete the optimization on five different dataset sizes. All datasets were optimized for each run.

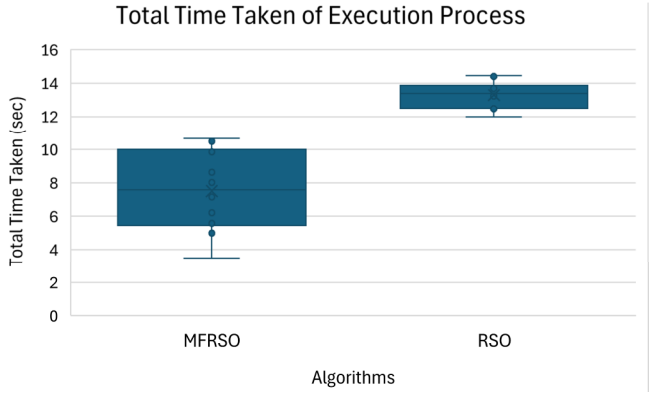

Fig. 5. Execution time taken for MFRSO and RSO across 10 runs.

In general, MFRSO has much higher optimization efficiency and takes less time to complete the algorithm execution process than RSO. This improvement is related to MFRSO's ability to handle multitasking through its multifactorial optimization approach, which allows it to optimize many datasets at the same time. MFRSO improves the optimization process by transferring knowledge across tasks, resulting in faster convergence and more efficient exploration of solution spaces. In contrast, RSO processes each dataset separately, optimizing one task at a time. This sequential technique leads to increased processing costs and longer execution times, particularly when dealing with huge datasets. The MFRSO ran ten times, with a best time of 3.498 s and a worst time of 10.679 s, with an average of 7.5116 s. On the contrary, the RSO's best time was 11.981 s, worst time was 14.458 s, and average time was 13.3047 s. These findings clearly highlight the benefits of MFRSO's multitasking capacity, which makes it a more efficient and scalable option for TSR across various and complicated datasets.

D. Overall Performance

Figure 6 provides a visual comparison between the RSO and the MFRSO across three key performance metrics: PTSR, PCOA, and Time taken. Both algorithms achieved 100% PCOA consistently, demonstrating their ability to fully meet requirement coverage. However, the chart clearly shows that MFRSO outperformed RSO in terms of PTSR, indicating its superior ability to minimize the test suite size more effectively. In the Time taken performance metric, MFRSO again showed stronger performance by completing the optimization process in a shorter time compared to RSO. This advantage is reflected in the higher performance percentage for MFRSO, highlighting its efficiency.

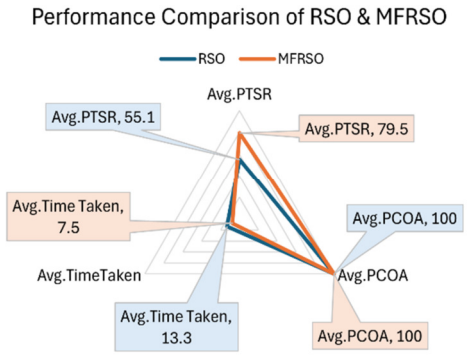

Fig. 6. Algorithm performance summary.

IV. CONCLUSION

The results confirm that MFRSO is more suitable for large-scale TSR, as it maintains higher optimization performance across different dataset sizes while being computationally efficient. The findings strongly support the superiority of MFRSO over RSO in terms of optimization effectiveness, stability, and execution time. These results highlight the potential of MFRSO as a more scalable and efficient TSR solution, contributing to improved software testing processes.

The novelty of this approach lies in the integration of the MFO framework with the RSO algorithm, enabling simultaneous optimization across multiple tasks using a single population. This approach not only improves the convergence behavior but also leverages knowledge transfer between tasks to accelerate the search process. Compared to previous works that focused on single-task or standalone optimization methods [22-25], MFRSO demonstrates clear advantages in handling multitask environments with varying dataset complexities. Experimental comparisons with the original RSO showed that MFRSO consistently achieved higher reduction while maintaining full requirement coverage at less computational overhead. These improvements validate the proposed method's strength in balancing solution quality and runtime efficiency.

Future work can explore adaptive information transfer techniques to further enhance optimization efficiency. Additionally, extending MFRSO to handle dynamic test suite scenarios or applying it to other software engineering challenges could further demonstrate its versatility and effectiveness.

ACKNOWLEDGMENT

This research was supported by the Fundamental Research Grant Scheme (FRGS) No: FRGS/1/2024/ICT01/UTEM/02/2. The authors thank the Ministry of Higher Education (MOHE) of Malaysia, University Technical Malaysia Melaka (UTeM), for their contribution and support.

REFERENCES

[1] L. Neves, O. Campos, R. Santos, C. Magalhaes, I. Santos, and R. D. S. Santos, "Elevating Software Quality in Agile Environments: The Role of Testing Professionals in Unit Testing," in *2024 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Toronto, Canada, May 2024, pp. 293–296, https://doi.org/10.1109/ICSTW60967.2024.00058.

[2] A. Alenzi, W. Alhumud, M. K. Khan, R. Michaels, and R. Bryce, "Events-Based Test Suite Reduction for Mobile App Test Suites Generated by Reinforcement Learning," in *2023 Congress in Computer Science, Computer Engineering, & Applied Computing (CSCE)*, Las Vegas, NV, USA, Jul. 2023, pp. 2650–2657, https://doi.org/10.1109/CSCE60160.2023.00423.

[3] A. S. Habib, S. U. R. Khan, and E. A. Felix, "A systematic review on search-based test suite reduction: State-of-the-art, taxonomy, and future directions," *IET Software*, vol. 17, no. 2, pp. 93–136, Apr. 2023, https://doi.org/10.1049/sfw2.12104.

[4] G. Dhiman, M. Garg, A. Nagar, V. Kumar, and M. Dehghani, "A novel algorithm for global optimization: Rat Swarm Optimizer," *Journal of Ambient Intelligence and Humanized Computing*, vol. 12, no. 8, pp. 8457–8482, Aug. 2021, https://doi.org/10.1007/s12652-020-02580-0.

[5] G. I. Sayed, "A Novel Multi-Objective Rat Swarm Optimizer-Based Convolutional Neural Networks for the Diagnosis of COVID-19 Disease," *Automatic Control and Computer Sciences*, vol. 56, no. 3, pp. 198–208, Jun. 2022, https://doi.org/10.3103/S0146411622030075.

[6] M. Ehteram, A. Seifi, and F. B. Banadkooki, "Rat Swarm Optimization Algorithm," in *Application of Machine Learning Models in Agricultural and Meteorological Sciences*, M. Ehteram, A. Seifi, and F. B. Banadkooki, Eds. Springer Nature, 2023, pp. 73–76.

[7] K. K. Mohammed, S. Mekhilef, and S. Buyamin, "Improved Rat Swarm Optimizer Algorithm-Based MPPT Under Partially Shaded Conditions and Load Variation for PV Systems," *IEEE Transactions on Sustainable Energy*, vol. 14, no. 3, pp. 1385–1396, Jul. 2023, https://doi.org/10.1109/TSTE.2022.3233112.

[8] P. Manickam *et al.*, "Empowering Cybersecurity Using Enhanced Rat Swarm Optimization With Deep Stack-Based Ensemble Learning Approach," *IEEE Access*, vol. 12, pp. 62492–62501, 2024, https://doi.org/10.1109/ACCESS.2024.3395328.

[9] M. Bhanumathi and B. Arthi, "Designing a Heuristic Based Hybrid CNN with Attention Mechanism for the Effective Classification of Fish Species," in *2023 5th International Conference on Inventive Research in Computing Applications (ICIRCA)*, Coimbatore, India, Aug. 2023, pp. 981–988, https://doi.org/10.1109/ICIRCA57980.2023.10220648.

[10] A. Gupta, Y. S. Ong, and L. Feng, "Multifactorial Evolution: Toward Evolutionary Multitasking," *IEEE Transactions on Evolutionary Computation*, vol. 20, no. 3, pp. 343–357, Jun. 2016, https://doi.org/10.1109/TEVC.2015.2458037.

[11] A. Gupta, Y. S. Ong, L. Feng, and K. C. Tan, "Multiobjective Multifactorial Optimization in Evolutionary Multitasking," *IEEE Transactions on Cybernetics*, vol. 47, no. 7, pp. 1652–1665, Jul. 2017, https://doi.org/10.1109/TCYB.2016.2554622.

[12] Y. Li, W. Gong, and S. Li, "Multitasking optimization via an adaptive solver multitasking evolutionary framework," *Information Sciences*, vol. 630, pp. 688–712, Jun. 2023, https://doi.org/10.1016/j.ins.2022.10.099.

[13] L. Li, M. Xuan, Q. Lin, M. Jiang, Z. Ming, and K. C. Tan, "An Evolutionary Multitasking Algorithm With Multiple Filtering for High-Dimensional Feature Selection," *IEEE Transactions on Evolutionary Computation*, vol. 27, no. 4, pp. 802–816, Aug. 2023, https://doi.org/10.1109/TEVC.2023.3254155.

[14] C. Lyu, Y. Shi, and L. Sun, "A Novel Multi-Task Optimization Algorithm Based on the Brainstorming Process," *IEEE Access*, vol. 8, pp. 217134–217149, 2020, https://doi.org/10.1109/ACCESS.2020.3042004.

[15] K. Z. Zamli, Md. A. Kader, and A. R. Mekeng, "A Population Division based Multi-Task Sine Cosine Algorithm for Test Redundancy Reduction Optimization," in *Proceedings of the 2024 10th International Conference on Computer Technology Applications*, Vienna Austria, May 2024, pp. 94–102, https://doi.org/10.1145/3674558.3674571.

[16] P. K. Mishra and A. K. Chaturvedi, "An Improved Laxity based Cost Efficient Task Scheduling Approach for Cloud-Fog Environment," *Engineering, Technology & Applied Science Research*, vol. 15, no. 1, pp. 19037–19044, Feb. 2025, https://doi.org/10.48084/etasr.8595.

[17] H. Jafarzadeh, N. Moradinasab, and M. Elyasi, "An Enhanced Genetic Algorithm for the Generalized Traveling Salesman Problem," *Engineering, Technology & Applied Science Research*, vol. 7, no. 6, pp. 2260–2265, Dec. 2017, https://doi.org/10.48084/etasr.1570.

[18] M. Shukla, Y. P. Kosta, and M. Jayswal, "A Modified Approach of OPTICS Algorithm for Data Streams," *Engineering, Technology & Applied Science Research*, vol. 7, no. 2, pp. 1478–1481, Apr. 2017, https://doi.org/10.48084/etasr.963.

[19] Y. Xu, D. Pi, S. Yang, and E. Zio, "Knowledge Transfer-Based Multifactorial Evolutionary Algorithm for Selective Maintenance Optimization of Multistate Complex Systems," *IEEE Transactions on Reliability*, vol. 73, no. 2, pp. 1341–1352, Jun. 2024, https://doi.org/10.1109/TR.2023.3324701.

[20] M. V. Merino and T. van der Storm, "cwi-swat/kogi: Kogi 0.1.0." Zenodo, Jun. 16, 2020, https://doi.org/10.5281/zenodo.4033220.

[21] B. Mayhew, "Guide to synthetic test data | TechTarget," *TechTarget - Search Software Quality*. https://www.techtarget.com/searchsoftwarequality/tip/Guide-to-synthetic-test-data.

[22] X. Y. Ma, Z. F. He, B. K. Sheng, and C. Q. Ye, "A Genetic Algorithm for Test-Suite Reduction," in *2005 IEEE International Conference on Systems, Man and Cybernetics*, Waikoloa, HI, USA, 2005, vol. 1, pp. 133–139, https://doi.org/10.1109/ICSMC.2005.1571134.

[23] A. Deneke, B. Gizachew Assefa, and S. Kumar Mohapatra, "Test suite minimization using particle swarm optimization," *Materials Today: Proceedings*, vol. 60, pp. 229–233, 2022, https://doi.org/10.1016/j.matpr.2021.12.472.

[24] T. Y. Chen and M. F. Lau, "A new heuristic for test suite reduction," *Information and Software Technology*, vol. 40, no. 5, pp. 347–354, Jul. 1998, https://doi.org/10.1016/S0950-5849(98)00050-0.

[25] M. Bharathi and V. Sangeetha, "Weighted Rank Ant Colony Metaheuristics Optimization Based Test Suite Reduction in Combinatorial Testing for Improving Software Quality," in *2018 Second International Conference on Intelligent Computing and Control Systems (ICICCS)*, Madurai, India, Jun. 2018, pp. 525–534, https://doi.org/10.1109/ICCONS.2018.8663102.