Optimization Strategies for the Reviewer Assignment Problem

Pham Cong Hoang*, Le Tien Hop, Tran Phong Quan, Nguyen Gia Minh
{*hoang.pc2416698, hop.lt2400105, quan.tp2416739, minh.ng2400111@sis.hust.edu.vn*}*@sis.hust.edu.vn*
Major: Data Science & Artificial Intelligence
School of Information and Communications Technology
Hanoi University of Science and Technology

***Abstract*—This report considers a variant of the scheduling problem, specifically the reviewer assignment problem. Given two sets of papers and reviewers, the objective is to design a feasible assignment of reviewers to papers such that exactly a constant number of reviewers are assigned to each paper and minimize the load of all reviewers. An integer linear programming (ILP) model is presented and solved to optimality with small and normal-sized instances due to the computational complexity of this problem. Thus, some heuristic and meta-heuristic algorithms such as greedy algorithm, hill-climbing local search, adaptive large neighborhood search, and genetic programming are proposed. Experiments are then carried out to evaluate the performance of the proposed algorithms.**

***Keywords*—paper-reviewer assignment, integer linear programming, heuristic, meta-heuristic.**

I. Introduction

The Reviewer Assignment Problem (RAP) is an important task in organizing academic conferences [1]. It involves matching submitted papers with the right reviewers based on their knowledge, interests, and availability. This problem is important in academic conferences, where papers need fair and expert reviews, while also making sure no reviewer is given too much work. Finding an efficient way to make these assignments helps improve the quality and fairness of the review process. As the number of papers and reviewers increases, this task becomes more difficult.

The goal of this report is to present, compare, and evaluate various methods for solving the RAP on a set of instances ranging from small to large. The study aims to analyze how each method performs under different conditions, focusing on solution quality, computational efficiency, and the ability to balance reviewer workloads. By testing across a diverse set of problem sizes, the report provides insights into the strengths and limitations of each approach, offering guidance on selecting suitable techniques for practical use in academic conference management. Specifically, our proposed methods are experimented on a generated dataset with the number of papers to assign ranging from 50 to 20,000.

The main outcomes of this report are the following:

- Define a mathematical formulation model for the RAP.
- Propose exact solver model, heuristic and meta-heuristic algorithms to address the RAP.
- Evaluate and compare the performance of the proposed methods based on two main criteria: solution quality and computational time.

The source code and dataset are publicly available at github.com/kongwoang.

II. Problem Description

The RAP involves assigning a set of submitted papers to a set of available reviewers while satisfying specific constraints and optimizing certain objectives. Each paper has a list of reviewers who are willing to review that paper and it must be reviewed by a fixed number of reviewers, typically to ensure fairness and diversity of feedback.

The goal is to find a feasible assignment that meets the following key requirements:

Coverage constraint: Each paper must be assigned to exactly b reviewers.

Optimization objective: Minimize the maximum number of assignments (load) across all reviewers, or alternatively, balance the workload while maximizing the overall suitability or preference scores between papers and reviewers.

The RAP is known to be an NP-hard problem [2], as a result, practical approaches often rely on heuristic, approximation, or optimization-based algorithms to obtain good-quality solutions within a reasonable amount of time.

III. Model Formulation

Let $\mathcal{P} = \{1, 2, \ldots, n\}$ and $\mathcal{R} = \{1, 2, \ldots, m\}$ respectively represent sets of papers and reviewers. For each paper i, we also denote $L(i)$ as the list of reviewers who are willing to review that paper. Parameter b represents the exact number of reviewers assigned to each paper.

We consider binary decision variables x_{ij} that determine reviewers j is assigned to review paper i.

We use the notation summarized in Table I to formulate the RAP as a mathematical model by defining the following constraints and objective function.

$$\text{Objective: } \min z = \max_{j \in \mathcal{R}} \sum_{i \in \mathcal{P}} x_{ij} \tag{1}$$

This report is written to present the mini project on the third topic of the Fundamentals of Optimization Course, taught by Dr. Pham Quang Dung in the 20242 semester.

* This is a self-report, written by the first author, is based on the results of research conducted by our team.

TABLE I
NOTATION USED IN THE MATHEMATICAL MODEL

Notation	Description
$\mathcal{P}$	Set of papers
$\mathcal{R}$	Set of reviewers
b	Number reviewers assigned to each paper
$L(i)$	Set of reviewers who are willing to review paper i
x_{ij}	Binary variable; 1 if paper i is assigned to reviewer j and 0 otherwise
z	Maximum load (number of assigned papers) of any reviewers

Define decision variable:

$$x_{ij} \in \{0; 1\} \quad i \in \mathcal{P}, j \in \mathcal{R} \tag{2}$$

Ensure reviewer j is not assigned to review paper i if eligible to review this paper:

$$\sum_{j \in \mathcal{R} \setminus L(i)} x_{ij} = 0 \quad \forall i \in \mathcal{P} \tag{3}$$

Each paper must be reviewed by exactly b reviewers:

$$\sum_{j \in L(i)} x_{ij} = b \quad \forall i \in \mathcal{P} \tag{4}$$

IV. INTEGER LINEAR PROGRAMMING APPROACH

The Integer Linear Programming (ILP) approach offers an exact method to solve the RAP by leveraging the mathematical model presented in Section III. This method formulates RAP as a set of linear constraints and an objective function to minimize the maximum reviewer load, ensuring optimal assignments that satisfy all problem constraints.

Due to the NP-hard nature of RAP, ILP is particularly effective for small instances but faces scalability challenges for larger ones. This section details the ILP formulation, its implementation using OR-Tools CP-SAT [3], OR-Tools Linear Programming (pywraplp) [4], and Gurobi solvers [5]. Then, we evaluate their advantages and limitations, setting the stage for performance comparisons with heuristic and meta-heuristic methods in later sections.

TABLE II
COMPARISON OF SOLVERS

Solver	Search strategy	Speed
pywraplp	Simplex, Branch-and-Bound	Moderate
CP-SAT	SAT-based Search	Fast (discrete)
Gurobi	Branch-and-Cut, Heuristics	Very fast

The overview of 3 solvers have described in Table II. The pywraplp is part of Google's OR-Tools and is good for solving basic linear and mixed-integer problems, but it can be slower on large problems. CP-SAT is also from OR-Tools and is designed for solving logic and constraint-based problems. Gurobi is a powerful commercial solver that can handle large and complex problems very efficiently, including linear, integer programming. However, Gurobi requires a license, although students and researchers can use it for free.

V. GREEDY ALGORITHM APPROACH

Greedy Algorithm (Algorithm of Kruskal) is a simple heuristic method to make locally optimal choices at each step with the hope of finding a global optimum solution. The strategy typically used is rather short-sighted: we always choose the element that seems best at the moment (that is, of all the admissible elements, we choose the element whose weight is optimal) and add it to the solution we are constructing.

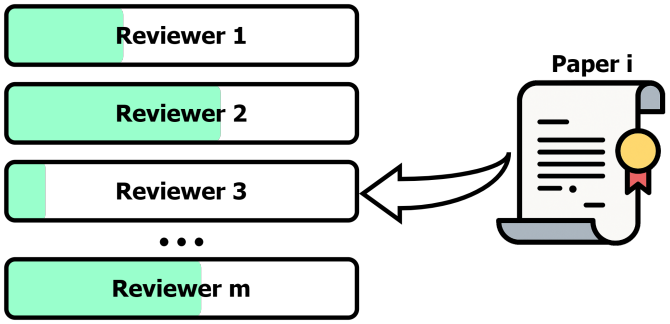

Fig. 1. Greedy Strategy. We consider Reviewer 3 has the minimum load.

In this problem, we use a greedy searching strategy to assign reviewers to papers by choosing locally optimal choices, which is illustrated in Figure 1. At each step, the algorithm selects, for each paper, a reviewer who is available, has the fewest current assignments, and can review the fewest total papers—thus balancing the workload and preserving flexibility. While it does not guarantee a global optimum, this method is fast and effective in practice.

VI. HILL-CLIMBING LOCAL SEARCH APPROACH

Local Search algorithm is a heuristic method that is employed to find high-quality solutions in large and complex problem spaces. To solve the RAP by Local Search, we implement the Hill-Climbing algorithm, which is a straightforward local search that iteratively moves to better solutions.

A baseline of Hill-Climbing Local Search (HCLS) is to construct a fitness function, an initial solution, a move strategy to transition to the neighbor, and stop conditions.

A. Fitness Function

The fitness function in HCLS is designed to measure the quality of a solution. In our approach, we define the fitness function as the **maximum reviewer load** thatis the highest number of papers assigned to any single reviewer and the goal is to minimize this value. Formally, for a solution S, let $\text{load}_r(S)$ denote the number of papers assigned to reviewer r. Then, the fitness function is defined as:

$$f_{\text{HCLS}}(S) = \max_{r \in \mathcal{R}} \text{load}_r(S) \tag{5}$$

It can be seen that, this fitness function and the objective function described in equation (1) are equivalent.

B. Initial Solution

The initial solution is generated randomly by assigning each paper b reviewers from its eligible reviewer list. This random construction respects all feasibility constraints: no paper is assigned more or fewer than b reviewers, and all selected reviewers are drawn from the allowed list for each paper. Although this method does not guarantee a balanced load across reviewers, it provides a valid starting point for the hill-climbing process. If any paper has fewer than b eligible reviewers, the algorithm terminates with an error to preserve feasibility.

C. Move Strategy

The move strategy defines how the algorithm explores the neighborhood of the current solution. In our implementation, we identify the reviewer with the highest current load—denoted as the *overloaded reviewer*—and attempt to reduce their load. This is done by iterating through the papers they are assigned to, and attempting to *replace* the overloaded reviewer with an alternative eligible reviewer who is not yet assigned to that paper and who currently has a lower load. The move is accepted immediately if it leads to a decrease in the maximum load, thereby improving the fitness value. This greedy replacement constitutes a hill-climbing step toward a better local solution.

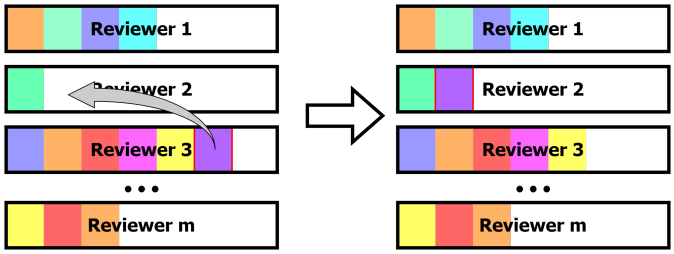

Fig. 2. Example of moving paper from Reviewer 3 to Reviewer 2.

D. Stop Conditions

The HCLS process terminates when no improving move can be found, i.e., the algorithm reaches a **local optimum** where any single reviewer replacement would not lead to a reduction in the maximum load. Additionally, to avoid infinite loops and ensure computational efficiency, we impose a maximum number of iterations *max_iter*. If this limit is reached before convergence, the algorithm halts and returns the best solution found so far. These two criteria together ensure that the algorithm either reaches a locally optimal solution or stops after a reasonable amount of search effort.

VII. Adaptive Large Neighborhood Search Approach

Large Neighborhood Search (LNS) is a metaheuristic optimization method designed to explore the solution space more effectively than traditional local search algorithms. Instead of making small incremental changes to a solution, LNS removes a significant part of the current solution (destroy phase) and reconstructs it using a heuristic method (repair phase). This strategy allows the algorithm to escape local optima and explore new regions of the solution space. However, the performance of LNS can be highly dependent on the choice of destroy and repair heuristics.

To overcome this limitation, **Adaptive Large Neighborhood Search (ALNS)** extends the LNS framework by integrating a learning mechanism that dynamically selects among multiple destroy and repair operators based on their historical performance. This adaptive mechanism improves the robustness and effectiveness of the search process across a variety of problem instances.

A. Fitness Function

Extending from the fitness in our proposed HCLS, we define a multi-factor fitness function that balances several important attributes of the problem. The fitness function for any solution S is formulated as follows:

$$f_{ALNS}(S) = \alpha \cdot f_{HCLS}(S) + \beta \cdot V(S) + \gamma \cdot \#T \quad (6)$$

Where:

- f_{HCLS} is fitness function of HCLS we have described in equation (5), it penalizes unbalanced solutions with highly loaded reviewers.
- $V(S)$ is the variance of the reviewer loads distribution, encouraging uniform distribution.
- $\#T$ is the number of elements of set T, which counts the number of reviewers whose load exceeds the average load $\frac{b \cdot N}{M}$.
- α, β, γ are weighting coefficients controlling the importance of each term and we consider $\alpha + \beta + \gamma = 1$

This multi-objective fitness function helps the algorithm not only reduce the maximum number of papers assigned to any single reviewer, but also improve fairness by keeping the workload more evenly distributed. In addition, it makes the ALNS algorithm more responsive to changes in the solution, allowing it to better compare different solutions and move towards better ones more effectively.

B. Initial Solution

The initial solution is constructed by the greedy strategy, which is referred to in the section V. For each paper i, the algorithm selects b reviewers from its candidate list $L(i)$ who currently have the lowest loads. This provides a feasible solution that respects the constraint that each paper must be assigned to exactly b reviewers, and it serves as a good starting point for the ALNS.

C. Destroy and Repair Operators.

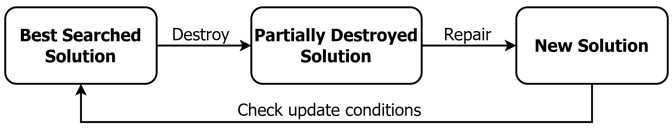

Fig. 3. ALNS Search Flow Chart

According to Figure 3, the proposed ALNS relies on two types of operators:

1) ***Destroy operators****:* These heuristics partially remove assignments to create room for improvements. Two strategies are used:

- *Random Destroy*: Randomly removes assignments from a subset of papers.
- *Worst-load Destroy*: Targets papers assigned to heavily loaded reviewers.

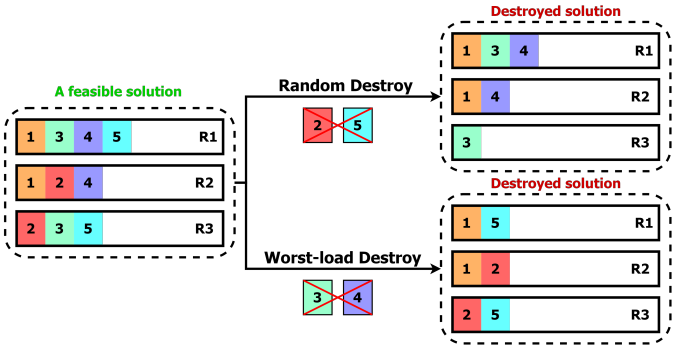

Fig. 4. Example of 2 destroy operators. By the random destroy operator, ALNS chooses random papers (we consider ratio = 0.4) and removes all assignments to these papers. About the worst-load destroy operator, like random destroy, but ALNS chooses random from the reviewer who has the max load.

2) ***Repair operators****:* These heuristics reconstruct assignments for the removed papers. Two methods are applied:

- *Greedy Repair*: Assigns reviewers with the lowest current load.
- *Random Repair*: Assigns reviewers randomly from the eligible list.

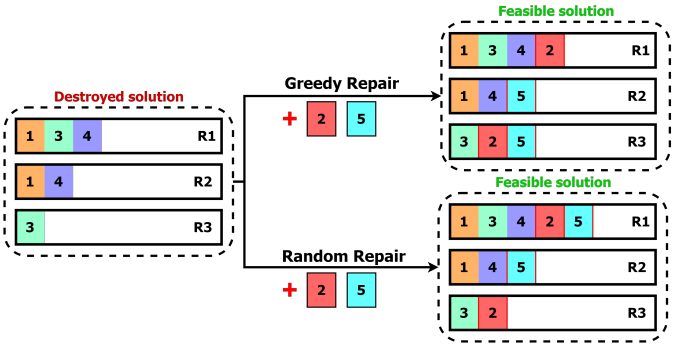

Fig. 5. Example of 2 repair operators after the destroy phase removes the reviewer assignments for papers 2 and 5. By greedy repair operator, ALNS selects reviewers for each removed paper from its candidate list based on the lowest current load. About random repair, reviewers are selected randomly from the eligible set for each paper, without considering load

D. Adaptive Mechanism

ALNS introduces a learning mechanism to select destroy and repair operators adaptively based on their past performance. Each operator is associated with a weight that is updated throughout the search:

- Operators that result in improved solutions receive a higher reward, increasing their probability of being selected in future iterations.
- The weights are updated using a decay-and-reward scheme that balances exploration and exploitation.

The selection of operators follows a roulette-wheel mechanism using the normalized weights, allowing the algorithm to focus on more effective heuristics over time.

E. Stop Conditions

The algorithm runs for a fixed number of iterations, which serves as the primary stopping condition. This iteration-based approach ensures predictable runtime and is suitable for benchmark comparison. Alternative stop conditions such as convergence thresholds or runtime limits can also be integrated if desired.

VIII. Genetic Programming Approach

Genetic Programming (GP) is an evolutionary algorithm-based meta-heuristic that automatically evolves computer programs, which are represented in tree form, to solve (approximately) problems [6].

Similar to Genetic Algorithm, GP mimics natural selection by applying genetic operators like mutation, crossover, and selection to a population of candidate programs, iteratively improving them over generations.

A. Individuals

Each individual in GP is a program represented as a tree, comprising internal nodes and terminal nodes. These nodes define the program's structure and functionality, tailored to the problem's requirements.

1) Terminal Nodes: Terminal nodes are the leaves of the GP tree, representing inputs or constants with no children. For the RAP, terminal nodes include attributes in table III:

TABLE III
Problem Attributes present on Terminal Nodes

Attribute	Description
CL (Current load)	The number of papers currently assigned to a reviewer.
S (Slack)	The difference between a reviewer's quota and their current load.
D (Degree)	The number of papers for which a reviewer is eligible.
CC (Candidate count)	The number of reviewers eligible for a specific paper.
Rand (Random)	A random value between 0 and 1, introducing stochasticity.
Const (Constant)	A fixed numerical value, typically in a predefined range (e.g., $[-2, 2]$).

2) Internal Nodes: Internal nodes represent operations or functions that process inputs from their child nodes. In the context of this report, internal nodes include arithmetic operators such as addition (+), subtraction (-), multiplication (*), division (/), minimum (min), and maximum (max). These operators

combine values from child nodes to compute intermediate results, enabling complex expressions within the program.

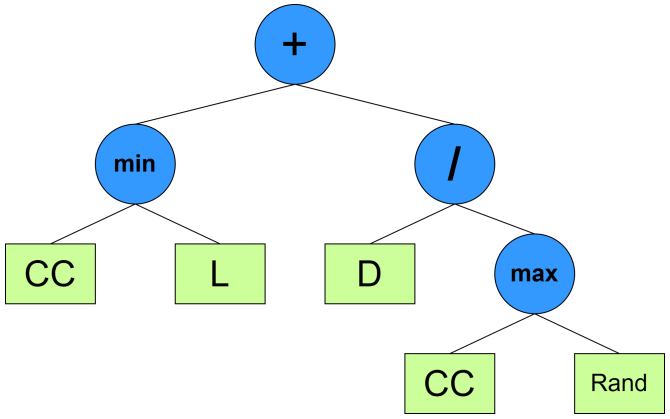

Fig. 6. Example of GP tree

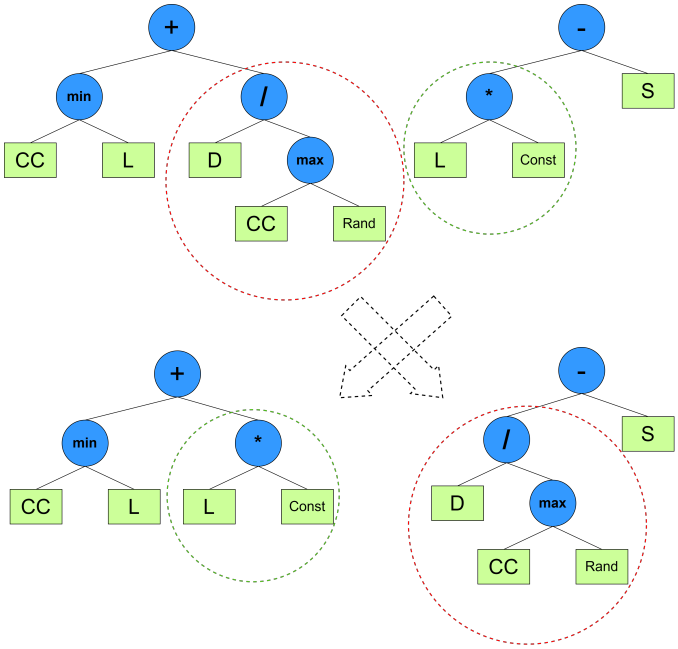

Fig. 7. Example of GP tree crossover

B. Initialization

GP initialization employs the *ramp-half-and-half* method to create a diverse initial population of program trees. This method combines two strategies, Grow and Full, across a range of tree depths (e.g., 2 to a maximum depth), ensuring varied tree sizes and structures:

- **Grow**: Trees are constructed starting from the root, with each node randomly chosen as either a function (with children) or a terminal. Nodes can become terminals before reaching the maximum depth, resulting in trees of varied shapes and sizes.
- **Full**: Trees are constructed such that all branches reach a predetermined depth, with internal nodes as functions until the maximum depth, where all leaves are terminals. This produces balanced, fully expanded trees.

The *ramp-half-and-half* method generates half the trees using Grow and half using Full for each depth in the range.

C. Genetic Operators

1) Crossover: Crossover combines two parent trees to produce child trees, promoting the exchange of useful program components. A random subtree is selected from each parent, then they are swapped to create two new trees. This operator preserves tree structures while introducing novel combinations, typically applied with a high probability (crossover rate).

2) Mutation: Mutation introduces random changes to a tree to maintain diversity and explore new solutions. A randomly selected subtree is replaced with a newly generated subtree, often created using the Grow method to allow varied depth and structure. Alternatively, a terminal node's value (e.g., a constant) may be altered, or the entire tree may be replaced with a new one. Mutation is applied with a lower probability to balance exploration with exploitation of existing solutions.

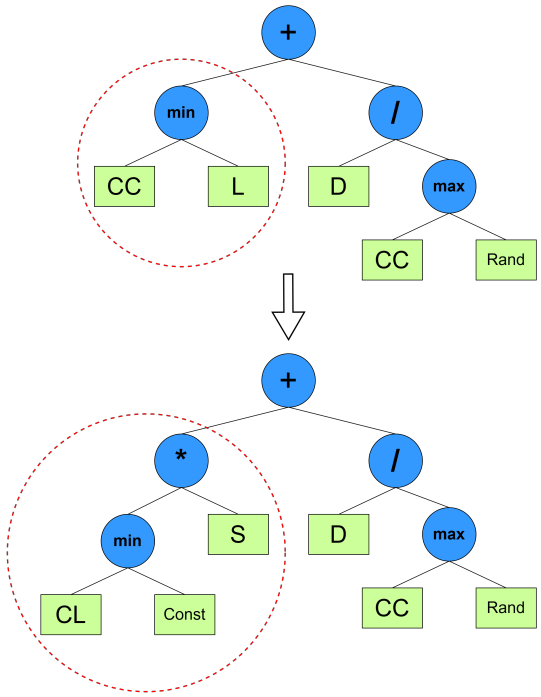

Fig. 8. Example of GP tree mutation

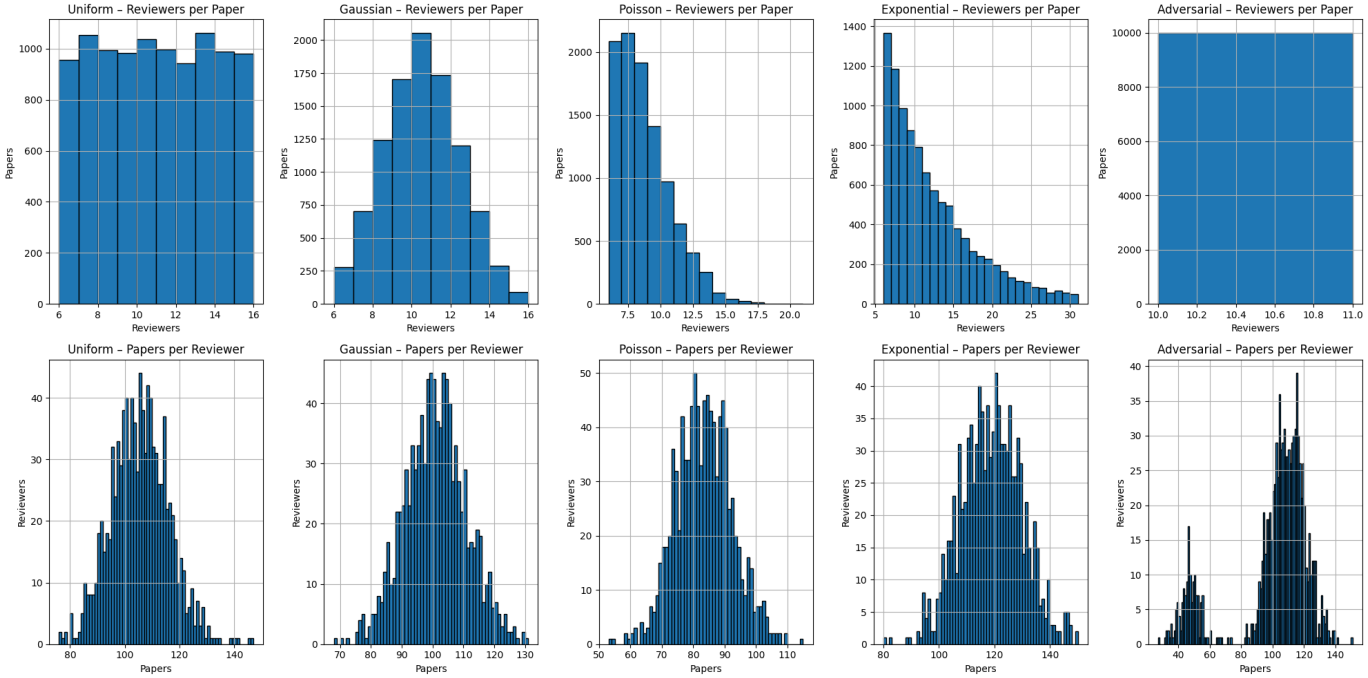

Fig. 9. Synthetic data's distributions example (N=10000, M=2000, b=5)

IX. Experiments

A. Dataset

We generate a dataset including 55 testcases with a different range of variables (N, M, b) following distinct distributions to diversify the eligibility of each paper and reviewer and to test the algorithms scalability:

TABLE IV
Experiment parameter settings

Parameter	Values
#papers	50, 100, 200, 500, 800, 1000, 2000, 5000, 10000, 20000
#reviewers	10 to 9000
b	2,3,4,5,6
Distribution	Uniform, Gaussian, Poisson, Exponential, Adversarial

To diversify the data, we first simulate popular distributions including Uniform, Gaussian, Poisson, and Exponential using random library. Unfortunately, figure 9 shows that despite different distributions of papers' capable reviewers, the correspond distribution of eligible number of papers per reviewer mostly follows normal distribution. We suspect that this phenomenon is a result of **Central Limit Theorem (CLT)** [7].

According to our research, the CLT states that when a variable is the sum of a large number of independent random variables, its distribution tends towards a normal shape. In our case, each reviewer's total eligibility count is the sum of many independent paper-reviewer selection events. This cumulative random sampling in numerous papers causes the overall eligibility counts to naturally converge into a bell-shaped curve and apparently reduces the diversity of the test data. Therefore, we manually create an additional distribution called "Adversarial" which mimics the rarity of certain reviewers and follows a different distribution.

This process involves dividing reviewers into a small "rare" group and a larger "common" group. The papers are then processed in two distinct phases. Initially, papers in the first phase are assigned eligible reviewers from both the "rare" and "common" pools. Subsequently, papers in the second phase are exclusively assigned reviewers from the "common" pool. This structure is designed to create an uneven distribution of reviewer eligibility, aiming to challenge algorithms like greedy by concentrating an imbalance workload onto the "common" reviewers.

B. Testing Environment

All of our proposed approaches have been experimentally tested on the server Github Codespaces[1]. Figure V shows the specifications detail.

TABLE V
Specifications of the GitHub Codespaces Environment

Specification	**Details**
CPU	2-core of AMD EPYC 7763 64-Core Processor
RAM	8 GB
Operating System	Ubuntu 20.04 LTS (64-bit)
Platform	GitHub Codespaces
Runtime Environment	Python 3.12.1

[1]https://github.com/codespaces

C. ILP Experiments

We construct the same model, including both objective and constraints, for all three solvers, with a time limit of 600 seconds. Table VI shows the summary of execution results, including the correct objective (optimal) and time-consuming. The detailed result is in the Appendix section.

TABLE VI
SUMMARY OF ILP SOLVERS RESULTS

Solver	Correct Objectives (/55)	Avg. Runtime (ms)
OR-Tools pywraplp	39	260951
OR-Tools Cp-SAT	55	271465
Gurobi Solver	55	130110

As we can see, Gurobi demonstrates the best overall performance, achieving both full correctness and the shortest average runtime. While CP-SAT also guarantees correct solutions, its longer runtime may limit its practicality for larger instances. OR-Tools pywraplp, although faster than CP-SAT in some cases, fails to deliver optimal results consistently, making it less reliable under strict time constraints.

The difference in results is mainly due to the 600-second time limit and the threading execution. OR-Tools pywraplp runs in a single-threaded mode, which restricts its ability to explore the solution space quickly. Meanwhile, Gurobi and CP-SAT support multi-threaded execution, allowing them to utilize multiple CPU cores and consistently achieve optimal solutions within the same time frame.

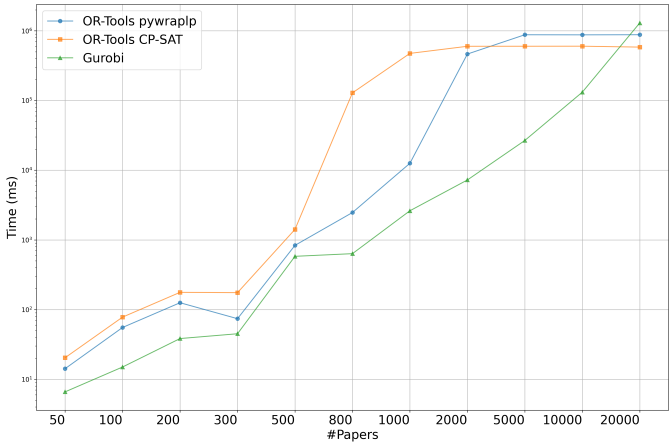

Fig. 10. ILP Solvers Average Execution Time

According to Figure 10, as the number of papers n increases, all solvers exhibit longer runtimes in exponential scaling. CP-SAT quickly reaches the 600-second time limit and becomes saturated, while ILP OR-Tools shows a steady increase until it also nears the limit. Gurobi remains significantly faster for small to medium instances but experiences a sharp increase in runtime on the largest instance, indicating potential scalability limitations beyond a certain problem size.

D. Greedy Algorithm Experiments

In the experiments, Greedy algorithm demonstrated the ability to find good solutions in a short time. Specifically, by our proof, the time complexity of this algorithm is $\mathcal{O}(NlogN)$ for sorting and $\mathcal{O}(b\cdot\sum_{i=1}^{N} L(i))$ for scanning the minimum workload in the worst case. Figure 11 also verifies our results.

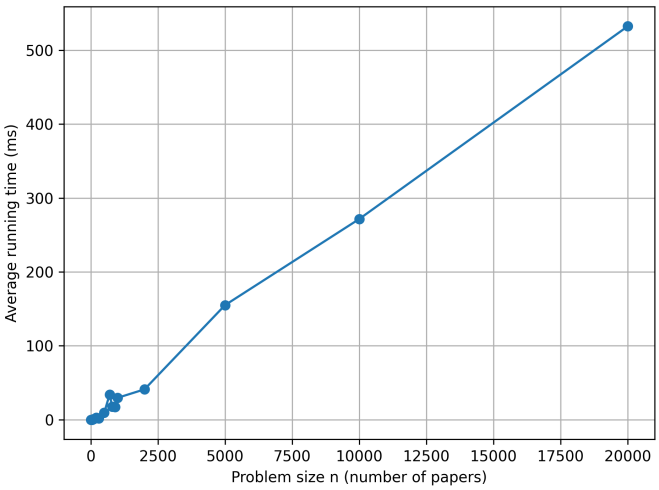

Fig. 11. Greedy algorithm – average running time vs. problem size

Our proposed Greedy Algorithm yielded promising results, achieving optimal solutions in 25 out of 55 test cases (shown in the appendix), with most of the remaining cases deviating by only one test instance. A notable advantage of this approach lies in its ease of implementation, as it requires no parameter tuning. Moreover, it consistently delivers the fastest results among all the proposed methods.

E. Hill-Climbing Local Search Experiments

Through experiments, we tested the algorithm using the following parameter settings and configuration. We could alternatively initialize the solution using a greedy strategy, where each paper is assigned to the reviewers with the current lowest load. While this may yield a better starting point in terms of fitness, it significantly reduces the diversity of the initial solution and may limit the exploration of the search space. As a result, the algorithm may converge prematurely to a local optimum and miss potentially better solutions that could be found through more diverse initial configurations.

TABLE VII
SETTINGS FOR THE LOCAL SEARCH ALGORITHM

Parameter	Value / Description
Initial solution	Randomly select b eligible reviewers for each paper
Fitness function	Maximum reviewer load
Stopping condition	No reviewer swap can further reduce max-load
Random seed	0 (to ensure reproducibility)

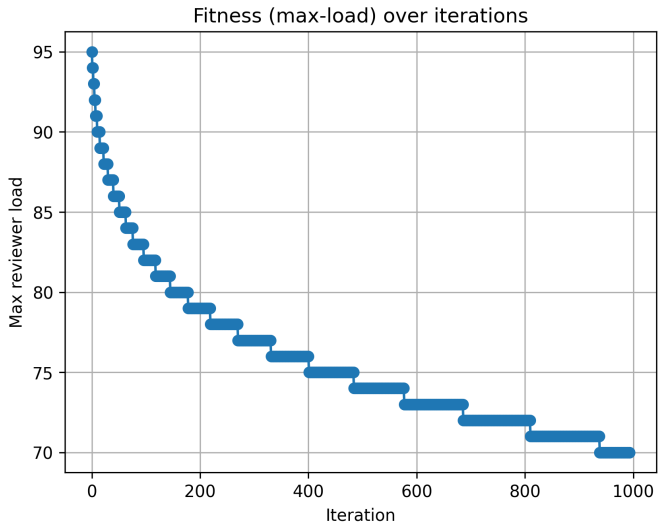

Fig. 12. Fitness function changes over iterations in test N = 2000

Figure 12 shows how the fitness value changes over time during the search for a dataset of size $N = 2000$. As observed, the algorithm quickly finds a better solution in the first few iterations, and then gradually converges to a stable value. This behavior is typical of hill-climbing methods, which often achieve rapid initial improvements followed by slower refinements as they approach a local optimum.

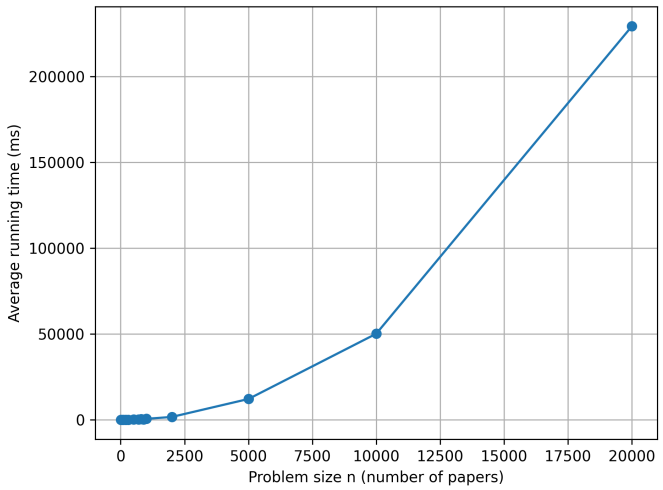

Fig. 13. HCLS – average running time vs. problem size

Although the HCLS algorithm takes longer to run compared to the Greedy approach, shown by Figure 13, its execution time remains reasonable for average-sized instances. Notably, HCLS provides better solution quality: it achieves optimal results in 32 out of 55 (shown in the appendix) test cases, outperforming the Greedy algorithm, which only reaches optimality in 25 cases. Furthermore, for the majority of remaining instances, HCLS produces solutions that are closer to optimal than those obtained by the Greedy approach.

F. Adaptive Large Neighborhood Search Experiments

Through our experiments, we found that the following parameter settings yielded the best performance, as shown in Table VIII.

TABLE VIII
BEST PARAMETER SETTINGS FOR ALNS

Parameter	**Value / Description**
α	0.9 (weight for max-load)
β	0.05 (weight for load variance)
γ	0.05 (weight for overload count)
Destroy ratio	0.15 (fraction of papers to unassign)
Max iterations	1000
Decay rate	0.9 (for update decay old weights)
Reward (improvement)	2.0
Reward (non-improvement)	0.1

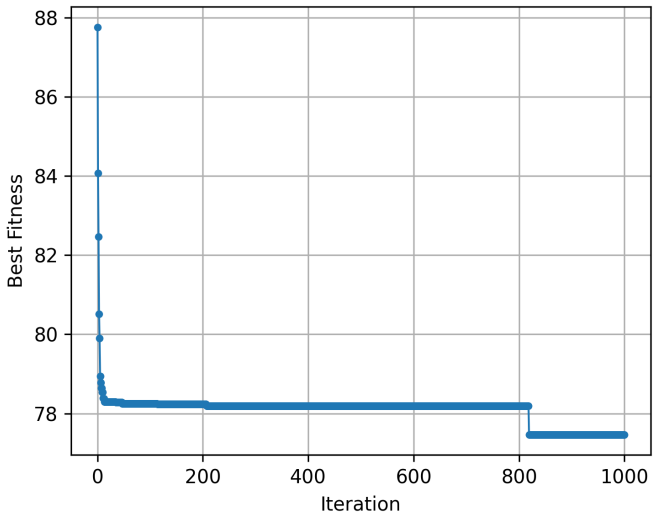

Fig. 14. ALNS Fitness Value over Iterations

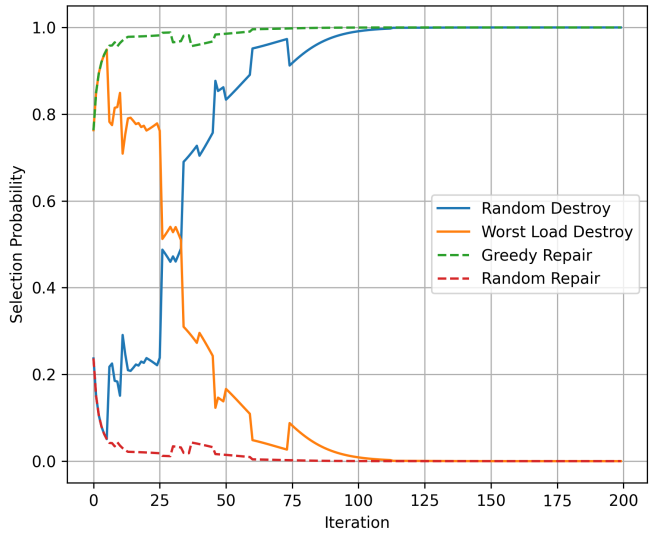

Fig. 15. Operator Selection Probabilities (Iteration 0–200)

According to Figure 14, the fitness curve shows that the algorithm quickly finds better solutions at the beginning. This means it can explore the search space well and improve the solution fast at first. After that, the improvements become slower and smaller, and the curve becomes almost flat.

Figure 15 illustrates the adaptive behavior of the ALNS algorithm. Over time, the algorithm strongly favors the combination of Random Destroy and Greedy Repair, as these operators consistently contribute to better solutions. Initially, other strategies like Worst Load Destroy and Random Repair are explored, but their selection probabilities gradually decrease, indicating lower effectiveness. This learning process highlights ALNS's ability to identify and exploit the most promising operators during the search.

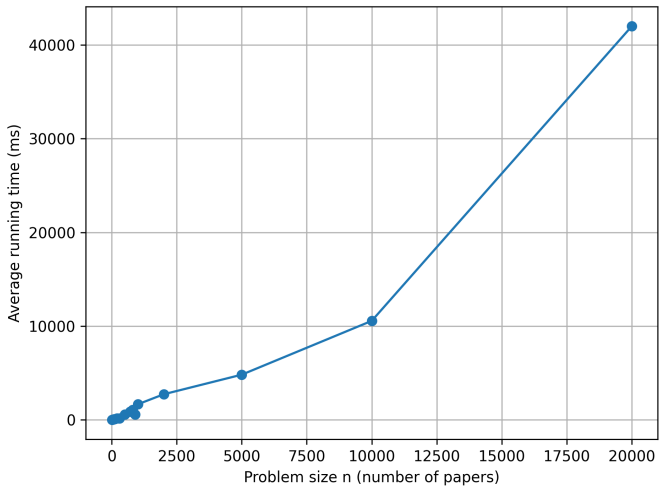

Fig. 16. ALNS – average running time vs. problem size

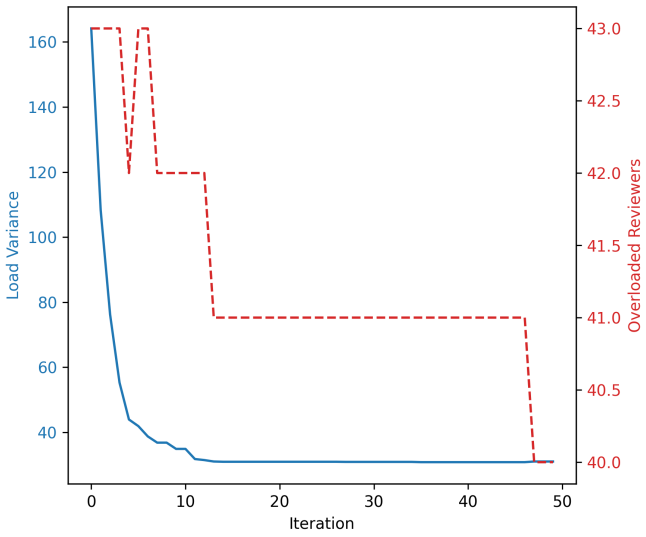

Fig. 17. Variance and Overloaded Reviewers

From Figure 16 and appendix, although the performance of ALNS is lower than that of both Greedy and HCLS in terms of execution time and the number of optimal solutions found, it has the advantage of addressing more complex aspects of the problem. Specifically, shown in Figure 17, ALNS is designed to consider additional factors such as variance or fairness in reviewer workloads, which are often overlooked by simpler methods. This makes ALNS more suitable for practical scenarios where solution quality is not only measured by optimality but also by balance and robustness.

G. Genetic Programming Experiments

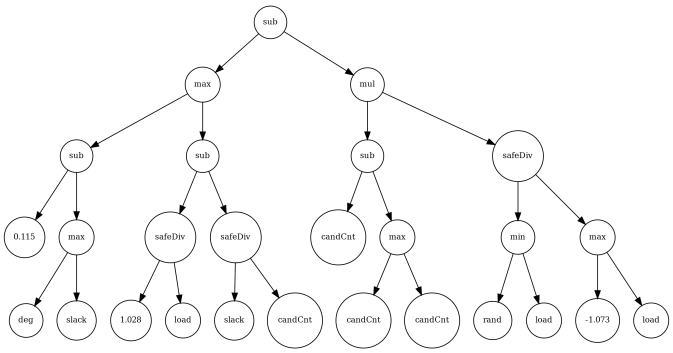

Fig. 18. Example of best GP tree in last iteration

For example of GP tree in Figure 18, from the tree structure, it is evident that the GP favors reviewers with higher `slack`, as indicated by multiple uses of expressions such as `max(slack, ...)` and `safeDiv(..., slack)`. These encourage assigning papers to underloaded reviewers.

The function also considers `candCnt` in multiple branches, reflecting the importance of reviewer diversity and availability in the decision process. Notably, the subtree $\texttt{max}(-1.073, \texttt{load})$ acts as a penalty mechanism for reviewers with high load, encouraging the selection of reviewers whose current load is below a learned threshold. The inclusion of `rand` in `min(rand, load)` suggests that the function retains a degree of randomness to promote exploration and avoid premature convergence to suboptimal solutions.

Overall, the evolved function is neither purely greedy nor fixed-rule based. Instead, it reflects a nonlinear combination of factors balancing exploitation (favoring low-load reviewers) and exploration (randomization and adaptive thresholds). This demonstrates GP's capacity to automatically construct complex, domain-specific heuristics that are difficult to hand-design.

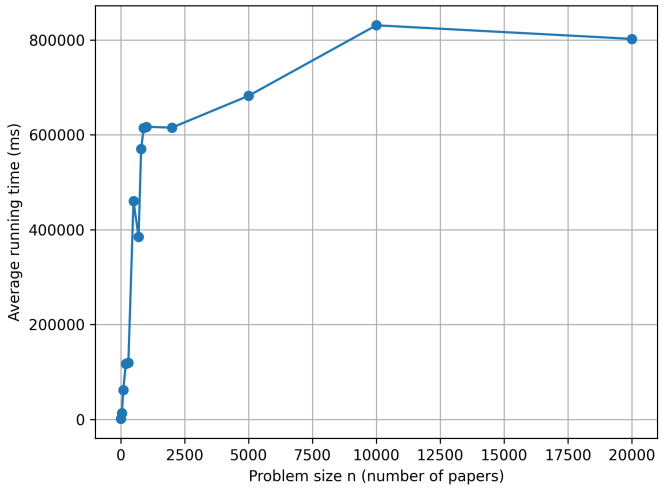

Fig. 19. GP – average running time vs. problem size

Figure 19 illustrates that the running time of the Genetic Programming (GP) algorithm increases noticeably with problem size, indicating poor scalability compared to other approaches. While GP performs adequately on small instances, it becomes significantly slower on larger datasets due to the overhead of evaluating complex tree-based individuals. Moreover, the solution quality obtained by GP is generally inferior to that of the other proposed methods. These limitations suggest the need for future improvements, such as integrating GP with local search strategies, applying adaptive operator selection, or optimizing tree representations to enhance both runtime efficiency and solution effectiveness.

X. Conclusion

This report has explored the Reviewer Assignment Problem (RAP) through both exact and heuristic-based optimization strategies. We first formulated RAP as an Integer Linear Program and evaluated the performance of three solvers—OR-Tools pywraplp, CP-SAT, and Gurobi—showing that Gurobi achieves the best balance of speed and optimality. However, scalability remains a challenge for ILP-based methods.

To address this, we proposed and implemented four approximation strategies: Greedy Algorithm, Hill-Climbing Local Search (HCLS), Adaptive Large Neighborhood Search (ALNS), and Genetic Programming (GP). Our experiments demonstrated that the Greedy algorithm is highly efficient and simple but may lack optimality. HCLS improves solution quality and maintains a good balance between speed and accuracy. ALNS, while slower, incorporates fairness by optimizing variance and overload, making it well-suited for real-world scenarios requiring equitable reviewer workloads. Finally, the Genetic Programming approach introduces structural flexibility and automatic policy learning, although its performance is less consistent.

Overall, each method exhibits unique strengths and trade-offs. The empirical results over 55 synthetic testcases provide valuable insight into when and how each approach should be applied. In future work, we aim to explore hybrid methods that combine local search with evolutionary learning, extend the model to support reviewer preferences or conflicts of interest, and apply these strategies to real-world datasets in academic peer-review systems.

Acknowledgement

Sincere gratitude is extended to Dr. Pham Quang Dung for his guidance and support throughout the course. His expertise and insightful feedback proved invaluable in completing this mini-project. Appreciation is also given for the opportunity to explore and deepen understanding of optimization principles. Thank you for the encouragement and dedication.

I would also like to thank all team members—Le Tien Hop, Tran Phong Quan, and Nguyen Gia Minh—for their active collaboration, dedication, and technical contributions during all phases of the project. Each member played a crucial role in the design, implementation, and evaluation of the proposed algorithms, making this work a true team effort.

References

[1] J. Jovanovic and E. Bagheri, "Reviewer assignment problem: A scoping review," *arXiv preprint arXiv:2305.07887*, 2023.

[2] M. Aksoy, S. Yanik, and M. F. Amasyali, "Reviewer assignment problem: A systematic review of the literature," *Journal of Artificial Intelligence Research*, vol. 76, pp. 1–46, 2023. [Online]. Available: https://www.jair.org/index.php/jair/article/view/14318

[3] L. Perron and F. Didier, "Cp-sat," Google. [Online]. Available: https://developers.google.com/optimization/cp/cp_solver/

[4] Google Developers, *Google OR-Tools: Linear Optimization (LP and MIP) Solver*, Google, 2024, accessed: 2025-06-05. [Online]. Available: https://developers.google.com/optimization/lp

[5] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2024. [Online]. Available: https://www.gurobi.com

[6] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[7] R. E. Walpole, R. H. Myers, S. L. Myers, and K. Ye, *Probability and Statistics for Engineers and Scientists*, 9th ed. Boston, MA: Pearson, 2012.

APPENDIX

APPENDIX A
PSEUDO CODE

Algorithm 1 Greedy Algorithm

Sort papers by increasing number of eligible reviewers ($|R(k)|$)
Initialize $count\_papers[j] \leftarrow 0$ for all reviewers j
for $r = 1$ to b **do**
  for each paper i (in the sorted list) **do**
    Select reviewer $j = \arg\min_{k \in R(i)} \big(count\_papers[k],\ |P(k)|\big)$
    Assign reviewer j to paper i
    $count\_papers[j] \leftarrow count\_papers[j] + 1$
    Remove j from $R(i)$
  end for
end for

Algorithm 2 Hill-Climbing Local Search

Generate a random feasible solution S
while True **do**
  $j^* \leftarrow$ reviewer with the highest current load
  for each paper i assigned to j^* **do**
    for each eligible reviewer $j' \in R(i)$ not in $S[i]$ **do**
      if Swapping j^* and j' reduces $\max$ load **then**
        perform the swap in S
        break out of both loops
      end if
    end for
  end for
  break if no improvement found

Algorithm 3 Adaptive Large Neighborhood Search (ALNS)

Input: instance (N, M, b, L); number of iterations max_iter
Generate initial solution S, compute load map L_S
Compute average load $\bar{L} \leftarrow \frac{b \cdot N}{M}$
Initialize operator weights dw, rw to 1.0
Evaluate fitness of S: $f^* \leftarrow f(L_S)$
for $t = 1$ to max_iter **do**
  Select destroy operator d from $[d_0, d_1]$ using weights dw
  Select repair operator r from $[r_0, r_1]$ using weights rw
  Save current $S_{old} \leftarrow S$, $L_{old} \leftarrow L_S$
  $R \leftarrow d(S, L_S)$ {Destroy step (e.g., random or worst-load)}
  $S \leftarrow r(S, L_S, R)$ {Repair step (e.g., greedy or random)}
  Evaluate new fitness $f(S)$
  if $f(S) < f^*$ **then**
    $f^* \leftarrow f(S)$
    Increase $dw[d]$ and $rw[r]$ with reward $= 2$
  else
    Restore $S \leftarrow S_{old}$, $L_S \leftarrow L_{old}$
    Slightly increase $dw[d]$ and $rw[r]$ with reward $= 0.1$
  end if
end for
return final reviewer load assignment L_S

TABLE IX
APPENDIX B: INTEGER LINEAR PROGRAMMING RESULTS

Instance	#Papers	#Reviewers	b	pywraplp		CP-SAT		Gurobi	
				Objective	Time (ms)	Objective	Time (ms)	Objective	Time (ms)
hustack1	5	3	2	4	3	4	3	4	2
hustack2	200	30	3	20	58	20	121	20	520
hustack3	300	30	3	30	74	30	175	30	45
hustack4	700	70	3	30	537	30	1.621	30	108
hustack5	900	90	3	30	796	30	1.476	30	146
Adversarial 0	50	10	2	10	10	10	16	10	5
Adversarial 1	100	10	3	30	39	30	128	30	8
Adversarial 2	200	15	3	40	68	40	108	40	18
Adversarial 3	500	30	4	67	280	67	326	67	113
Adversarial 4	800	50	5	82	696	82	948	82	204
Adversarial 5	1.000	80	5	65	999	65	3.100	65	664
Adversarial 6	2.000	150	5	69	3.966	69	600.249	69	1.254
Adversarial 7	5.000	300	6	106	41.516	106	600.802	106	14.441
Adversarial 8	10.000	600	6	149	760.855	106	601.574	106	97.468
Adversarial 9	20.000	1.000	6	183	784.204	128	603.446	128	428.173
Exponential 0	50	20	2	5	16	5	22	5	9
Exponential 1	100	50	3	6	57	6	87	6	14
Exponential 2	200	100	3	6	137	6	162	6	47
Exponential 3	500	350	4	6	1.016	6	1.603	6	509
Exponential 4	800	500	5	8	4.826	8	7.664	8	335
Exponential 5	1.000	700	5	8	4.715	8	600.203	8	3.272
Exponential 6	2.000	900	5	26	617.995	12	600.327	12	9.439
Exponential 7	5.000	2.000	6	30	898.355	15	601.621	15	14.820
Exponential 8	10.000	4.000	6	28	885.106	15	602.163	15	61.932
Exponential 9	20.000	9.000	6	30	921.194	14	604.697	14	2.409.530
Gaussian 0	50	20	2	5	15	5	20	5	6
Gaussian 1	100	50	3	6	58	6	98	6	13
Gaussian 2	200	100	3	6	111	6	206	6	32
Gaussian 3	500	350	4	6	691	6	1.004	6	827
Gaussian 4	800	500	5	9	1.716	9	599.903	9	1.029
Gaussian 5	1.000	700	5	8	3.946	8	498.038	8	2.895
Gaussian 6	2.000	900	5	12	342.269	12	600.287	12	5.133
Gaussian 7	5.000	2.000	6	29	848.155	16	600.765	16	21.996
Gaussian 8	10.000	4.000	6	29	855.337	16	601.646	16	177.038
Gaussian 9	20.000	9.000	6	28	800.141	14	568.291	14	774.545
Poisson 0	50	20	2	5	15	5	26	5	6
Poisson 1	100	50	3	6	60	6	61	6	14
Poisson 2	200	100	3	6	109	6	161	6	37
Poisson 3	500	350	4	6	585	6	1.101	6	451
Poisson 4	800	500	5	8	1.404	8	16.425	8	319
Poisson 5	1.000	700	5	8	2.928	8	599.955	8	2.825
Poisson 6	2.000	900	5	12	330.627	12	600.304	12	5.253
Poisson 7	5.000	2.000	6	30	879.356	16	600.721	16	32.804
Poisson 8	10.000	4.000	6	30	884.789	16	601.885	16	177.062
Poisson 9	20.000	9.000	6	31	844.061	14	578.841	14	562.033
Uniform 0	50	20	2	5	14	5	18	5	7
Uniform 1	100	50	3	6	61	6	83	6	16
Uniform 2	200	100	3	6	174	6	245	6	37
Uniform 3	500	350	4	6	1.385	6	2.786	6	766
Uniform 4	800	500	5	8	2.847	8	16.246	8	361
Uniform 5	1.000	700	5	8	48.042	8	600.207	8	2.940
Uniform 6	2.000	900	5	24	760.219	12	600.390	12	10.229
Uniform 7	5.000	2.000	6	28	934.078	15	601.115	15	9.192
Uniform 8	10.000	4.000	6	30	884.006	15	602.582	15	59.760
Uniform 9	20.000	9.000	6	30	997.628	14	604.556	14	2.265.356

TABLE X
APPENDIX C: HEURISTIC AND EXACT SOLVER OBJECTIVES
(ENTRIES IN **BOLD** MATCH THE GUROBI OPTIMUM)

Instance	#Papers	#Reviewers	b	Optimal	Greedy	HCLS	ALNS	GP
Adversarial 0	50	10	2	10	11	**10**	**10**	**10**
Adversarial 1	100	10	3	30	31	**30**	**30**	**30**
Adversarial 2	200	15	3	40	41	**40**	41	41
Adversarial 3	500	30	4	67	69	68	**67**	68
Adversarial 4	800	50	5	82	83	83	**82**	85
Adversarial 5	1000	80	5	65	66	66	66	66
Adversarial 6	2000	150	5	69	70	70	70	71
Adversarial 7	5000	300	6	106	107	107	107	108
Adversarial 8	10000	600	6	106	107	108	107	109
Adversarial 9	20000	1000	6	128	**128**	**128**	**128**	130
Exponential 0	50	20	2	5	**5**	6	**5**	**5**
Exponential 1	100	50	3	6	7	7	7	7
Exponential 2	200	100	3	6	7	7	7	7
Exponential 3	500	350	4	6	**6**	**6**	11	7
Exponential 4	800	500	5	8	9	9	9	9
Exponential 5	1000	700	5	8	**8**	**8**	**8**	**8**
Exponential 6	2000	900	5	12	**12**	**12**	**12**	**12**
Exponential 7	5000	2000	6	15	16	16	16	16
Exponential 8	10000	4000	6	15	16	16	17	16
Exponential 9	20000	9000	6	14	**14**	**14**	15	15
Gaussian 0	50	20	2	5	6	6	6	**5**
Gaussian 1	100	50	3	6	7	7	7	7
Gaussian 2	200	100	3	6	7	7	7	7
Gaussian 3	500	350	4	6	**6**	**6**	12	7
Gaussian 4	800	500	5	9	**9**	**9**	**9**	**9**
Gaussian 5	1000	700	5	8	**8**	**8**	**8**	**8**
Gaussian 6	2000	900	5	12	**12**	**12**	**12**	**12**
Gaussian 7	5000	2000	6	16	**16**	**16**	**16**	**16**
Gaussian 8	10000	4000	6	16	**16**	**16**	**16**	**16**
Gaussian 9	20000	9000	6	14	**14**	**14**	15	15
hustack1	5	3	2	4	**4**	**4**	**4**	**4**
hustack2	200	100	3	6	7	7	7	7
hustack3	300	30	3	30	31	**30**	31	31
hustack4	700	70	3	30	31	**30**	31	31
hustack5	900	90	3	30	31	**30**	31	31
Poisson 0	50	20	2	5	**5**	**5**	**5**	**5**
Poisson 1	100	50	3	6	**6**	7	7	7
Poisson 2	200	100	3	6	7	**6**	7	7
Poisson 3	500	350	4	6	7	**6**	9	7
Poisson 4	800	500	5	8	9	9	9	9
Poisson 5	1000	700	5	8	**8**	**8**	**8**	**8**
Poisson 6	2000	900	5	12	**12**	**12**	**12**	**12**
Poisson 7	5000	2000	6	16	**16**	**16**	**16**	17
Poisson 8	10000	4000	6	16	**16**	**16**	**16**	17
Poisson 9	20000	9000	6	14	**14**	**14**	15	15
Uniform 0	50	20	2	5	6	6	6	6
Uniform 1	100	50	3	6	7	**6**	**6**	7
Uniform 2	200	30	3	20	21	21	21	21
Uniform 3	500	350	4	6	**6**	**6**	9	7
Uniform 4	800	500	5	8	9	9	9	9
Uniform 5	1000	700	5	8	**8**	**8**	**8**	**8**
Uniform 6	2000	900	5	12	**12**	**12**	**12**	**12**
Uniform 7	5000	2000	6	15	16	16	16	16
Uniform 8	10000	4000	6	15	16	16	16	16
Uniform 9	20000	9000	6	14	**14**	**14**	15	15