Lab Report: Biologically Inspired Algorithms and Models
Part I: Local Optimization Algorithms, Problem: QAP

Patryk Maciejewski
1515960
AI

Dmytro Romaniv
151958
AI

April 7, 2025

Contents

1 Introduction

The Quadratic Assignment Problem (QAP) is about assigning facilities to locations while minimizing total cost. Imagine you have n facilities and n locations - you want to put each facility in one location so that the total cost is as small as possible. The cost depends on both the distance between locations and the flow (or traffic) between facilities.

The math looks like this:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} f_{ij} \cdot d_{\pi(i)\pi(j)} \tag{1}$$

Where f_{ij} is the flow between facilities i and j, and d_{kl} is the distance between locations k and l. QAP is NP-hard which means finding the perfect solution gets really hard really fast as the problem size grows.

1.1 Problem Instances

To ensure a fair evaluation, we selected a diverse set of instances from different datasets and of varying sizes:

- **bur26a**
- **chr12a**
- **esc32c**
- **had20**
- **nug25**
- **scr15**
- **tho30**
- **esc64a**

This selection includes small, medium, and large instances to test scalability and general performance across a range of scenarios.

We chose different sizes and types to see how our algorithms handle different problems.

1.2 Implementation Details

We wrote everything in Java. Our code is organized like this:

- Base Algorithm class with common functionality
- Local search algorithms that extend the base class
- Solution evaluation
- Measurement tools to track how algorithms perform

2 Neighborhood Operators

For both greedy and steepest local search, we used a simple 2-swap neighborhood. This means we generate neighbor solutions by swapping the positions of two facilities. For a problem with n facilities, the neighborhood size is:

$$\binom{n}{2} = \frac{n(n-1)}{2} \tag{2}$$

So for a problem with 20 facilities, we'd have 190 possible swaps. This operator is easy to implement but still gives enough options to find good solutions.

3 Comparison of Performance

To measure how good our solutions are, we use the ratio of our solution value to the optimal solution value multiplied by 100:

$$\text{Quality} = \frac{f_{optimal}}{f_{solution}} \times 100 \tag{3}$$

A score of 100 means we found the optimal solution, and lower numbers mean worse solutions. We tested five different algorithms:

- Random Search (RS): Just tries random solutions
- Random Walk (RW): Makes random moves but keeps improvements
- Nearest Neighbor (H): Builds solutions based on flow/distance patterns
- Greedy Local Search (G): Takes the first improvement found
- Steepest Local Search (S): Checks all neighbors and takes the best improvement

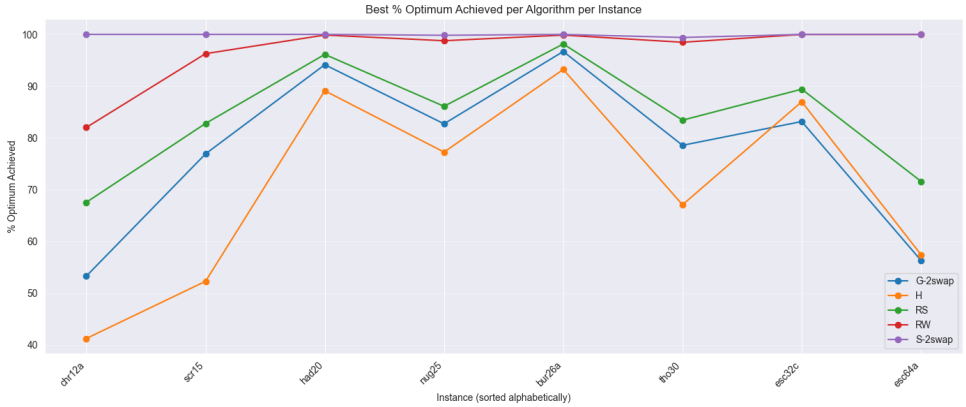


Figure 1: Best case performance comparison across all instances.

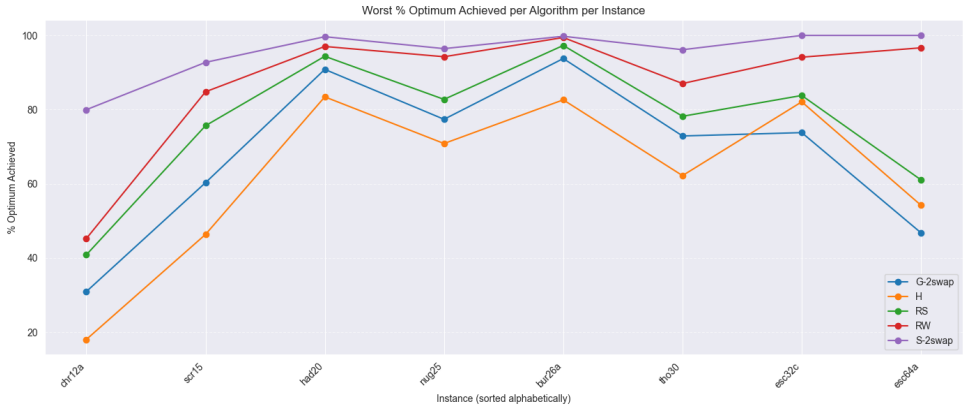


Figure 2: Worst case performance comparison across all instances.

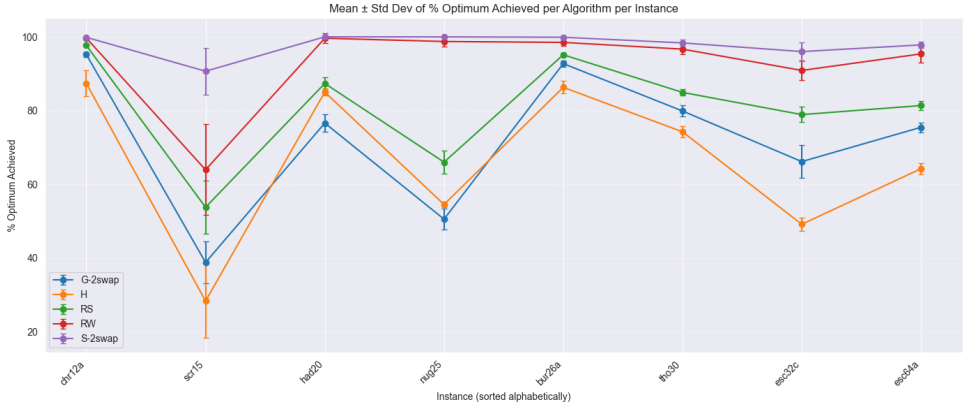


Figure 3: Average performance with standard deviation across all instances.

3.1 Runtime Analysis

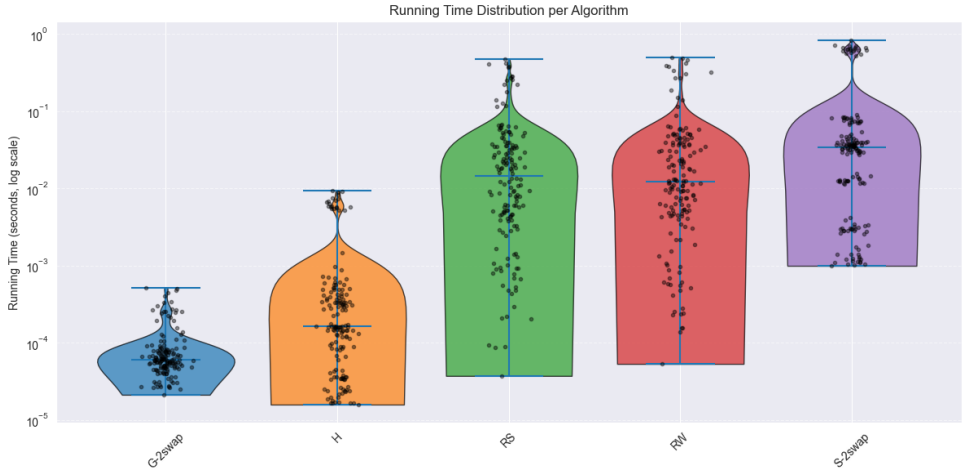


Figure 4: Runtime comparison in seconds (log scale) showing distribution across algorithms.

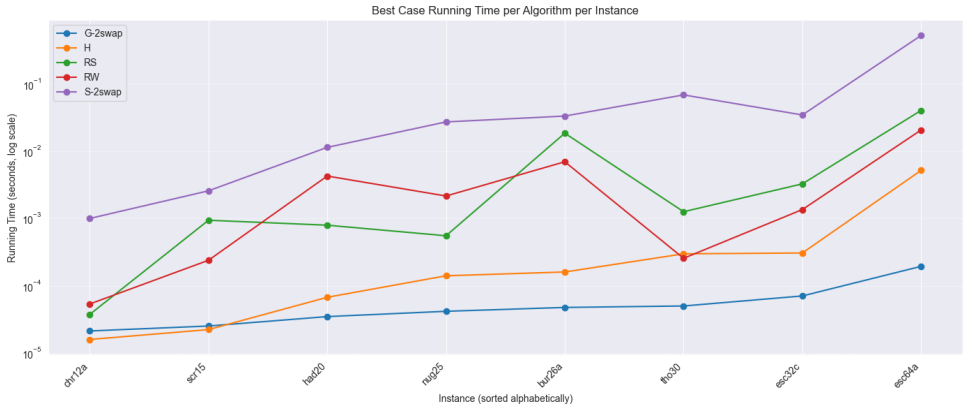


Figure 5: Best case runtime comparison across all instances.

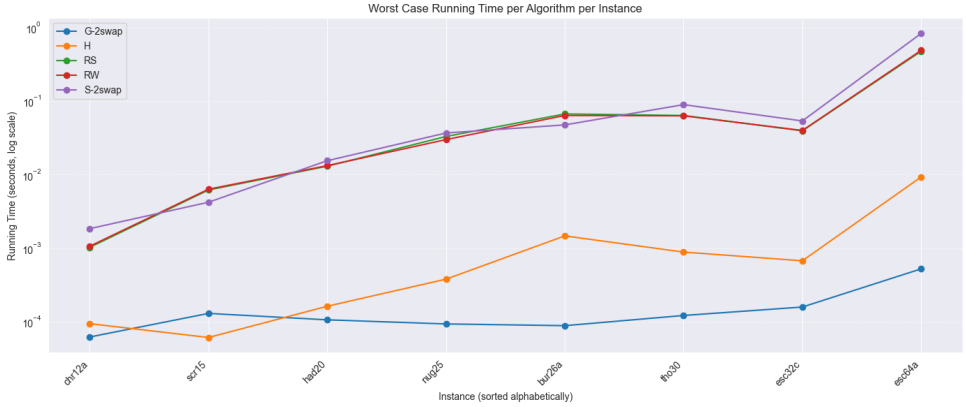


Figure 6: Worst case runtime comparison across all instances.

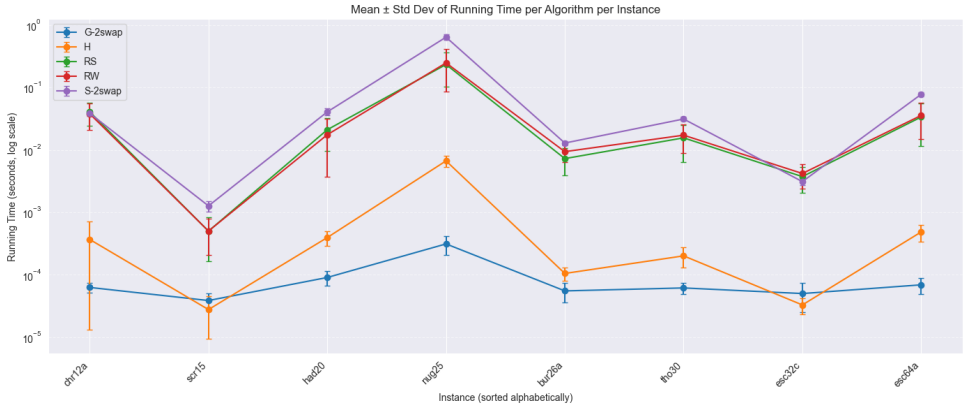


Figure 7: Average runtime with standard deviation across all instances.

3.2 Algorithm Efficiency

We define efficiency as quality divided by time to favor algorithms that get good solutions quickly:

$$\text{Efficiency} = \frac{\%\ \text{Optimum Achieved}}{\text{Runtime (seconds)}} \tag{4}$$

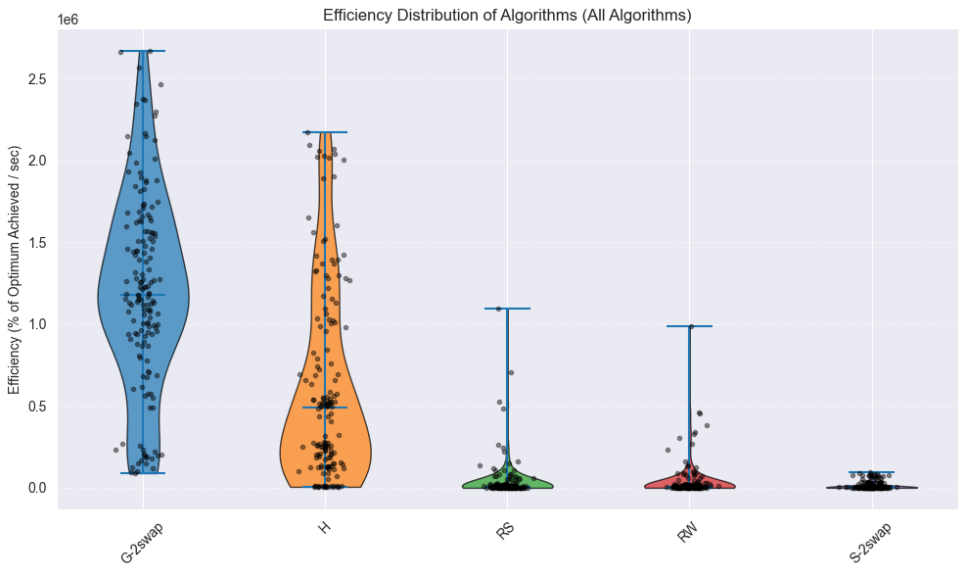


Figure 8: Efficiency distribution across all algorithms.

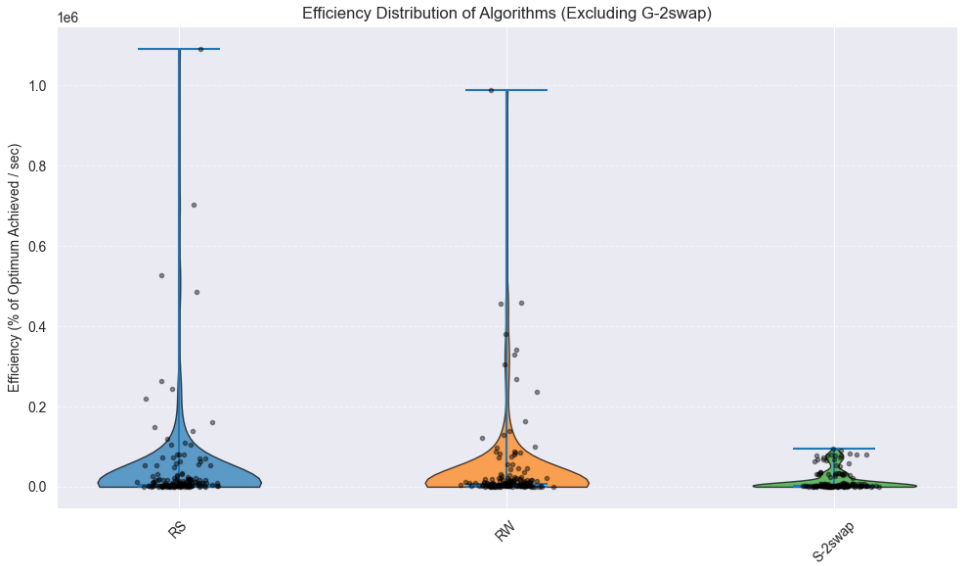


Figure 9: Efficiency distribution excluding G-2swap algorithm.

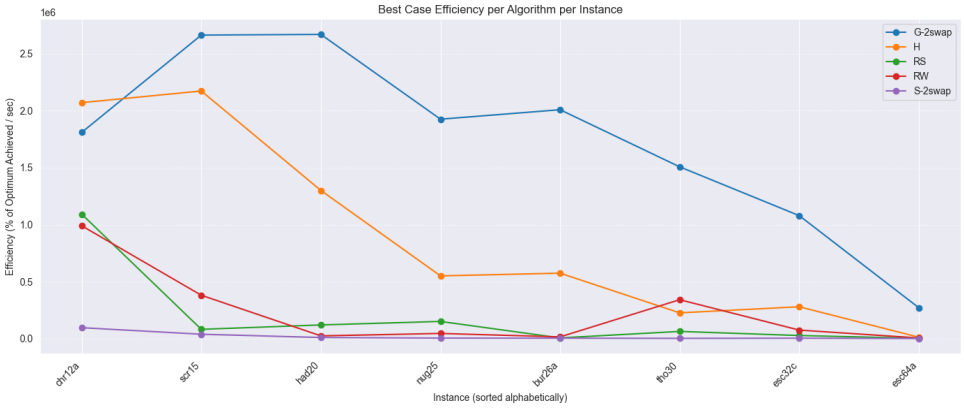


Figure 10: Best case efficiency comparison across all instances.

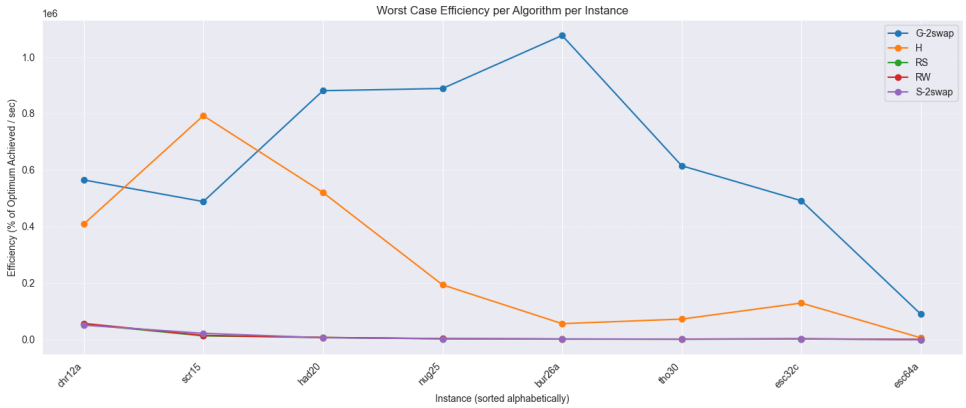


Figure 11: Worst case efficiency comparison across all instances.

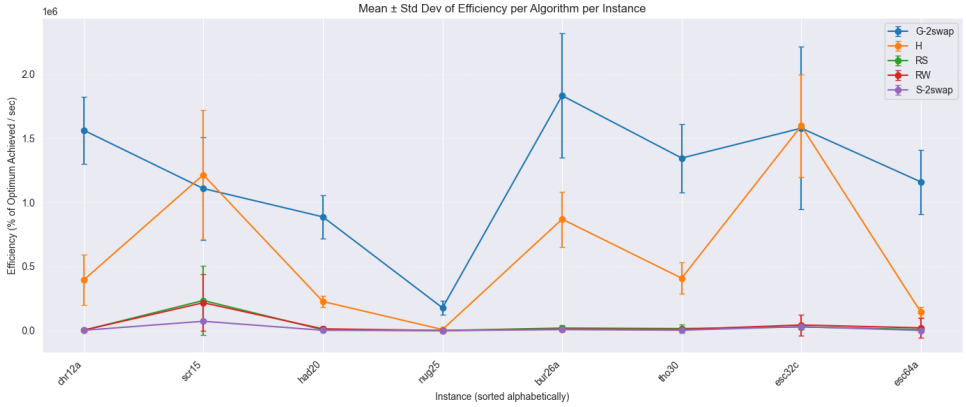


Figure 12: Average efficiency with standard deviation across all instances.

3.3 Algorithm Steps

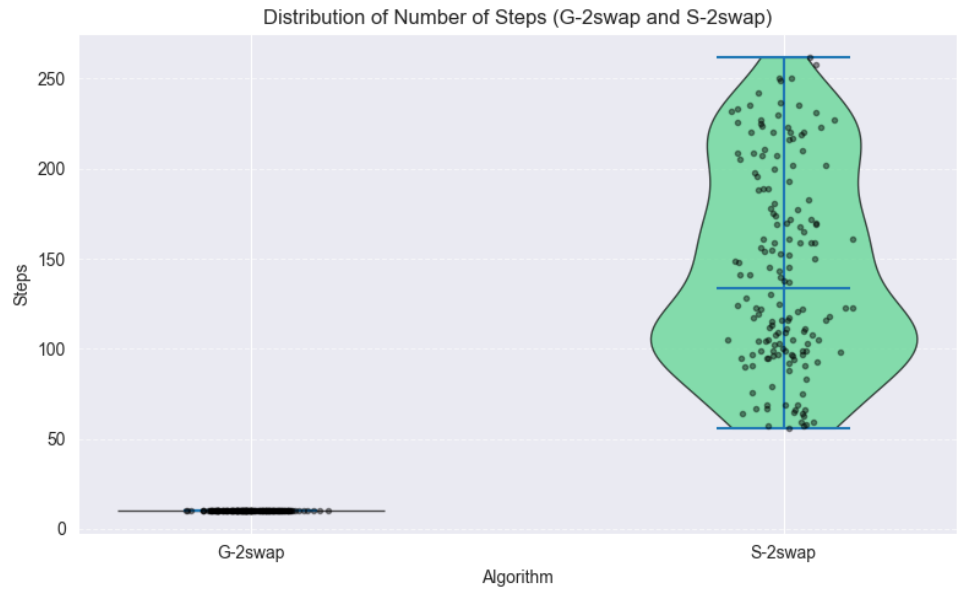


Figure 13: Number of steps comparison between greedy and steepest search.

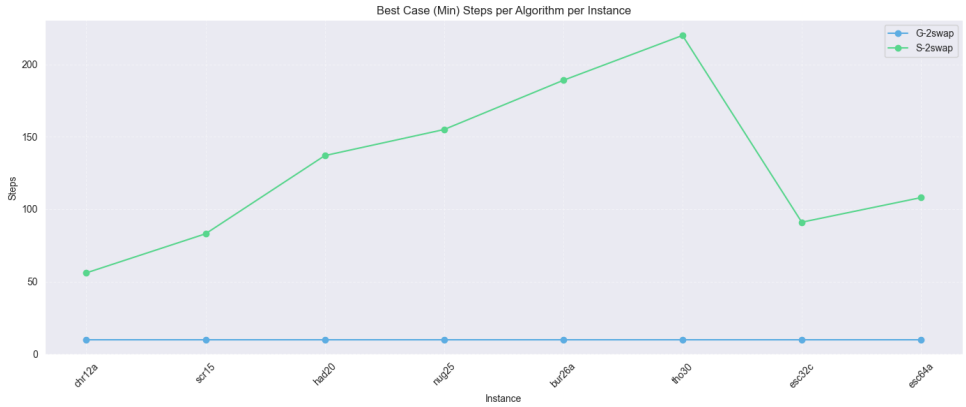


Figure 14: Best case (min) steps comparison across all instances.

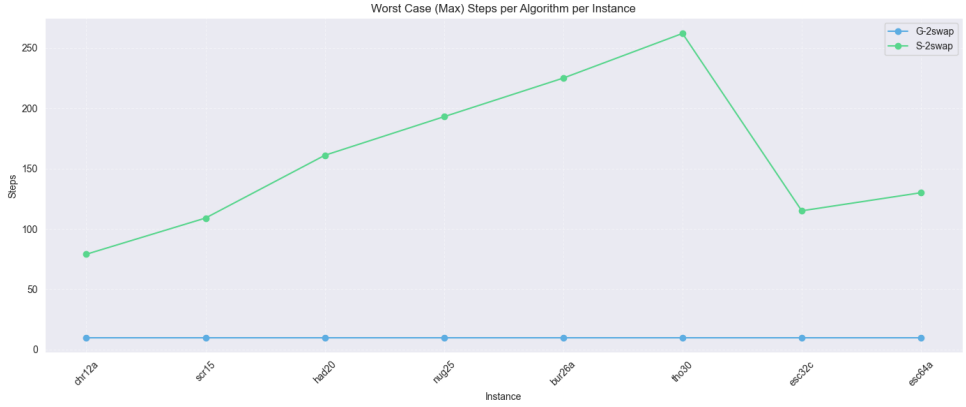


Figure 15: Worst case (max) steps comparison across all instances.

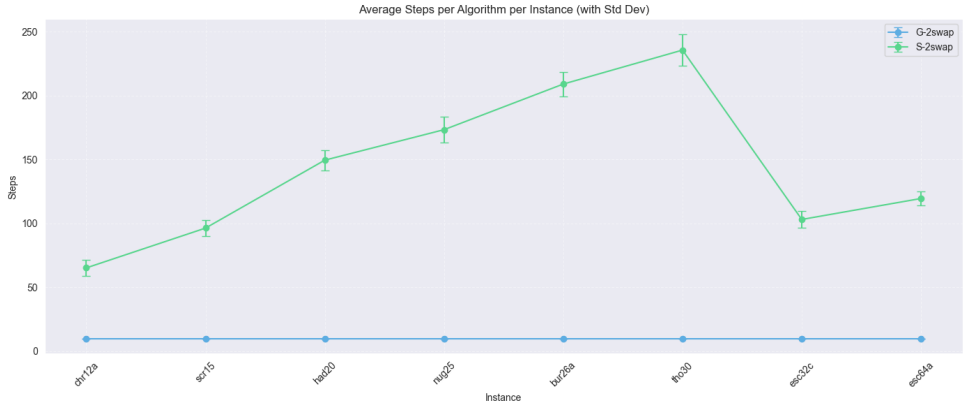


Figure 16: Average steps with standard deviation across all instances.

3.4 Solution Evaluations

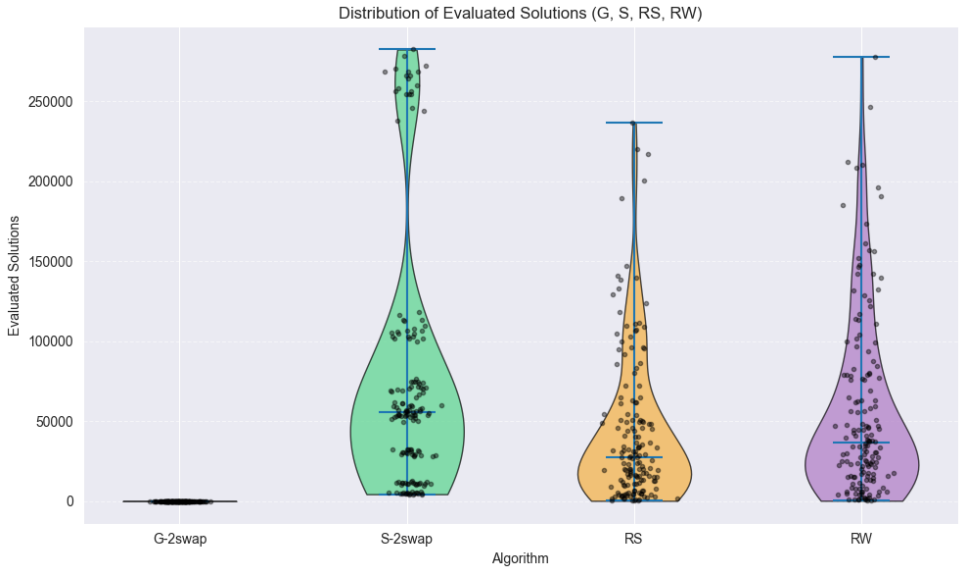


Figure 17: Distribution of solution evaluations across G, S, RS, and RW algorithms.

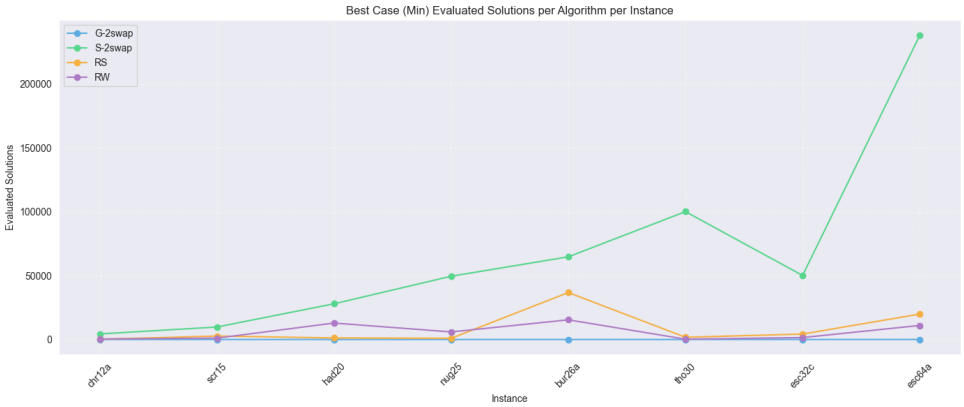


Figure 18: Best case (min) evaluations comparison across all instances.

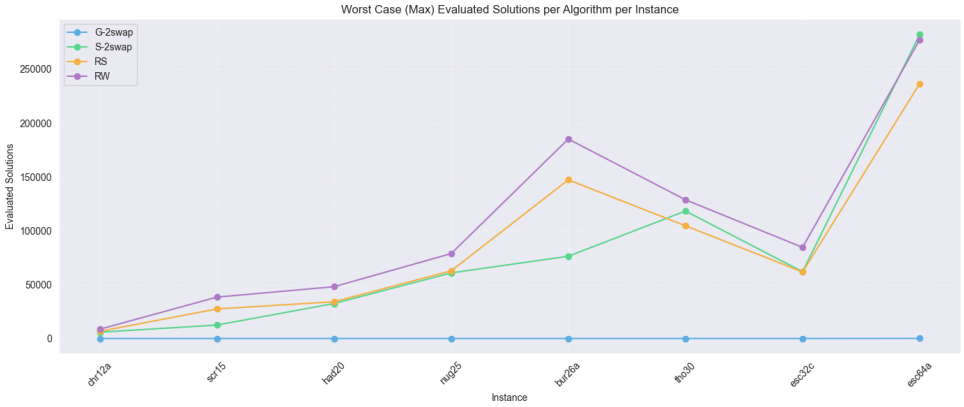


Figure 19: Worst case (max) evaluations comparison across all instances.

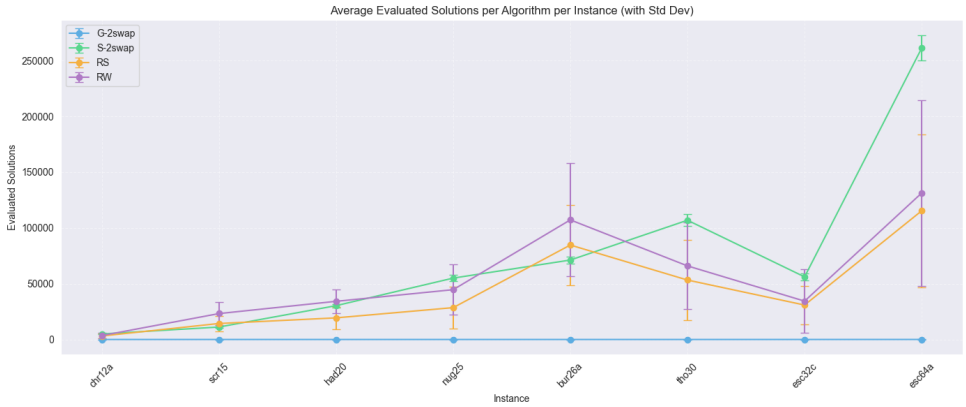


Figure 20: Average evaluations with standard deviation across all instances.

4 Comparing Local Search Strategies

We took a closer look at greedy and steepest local search to see which works better.

4.1 Initial vs Final Solutions

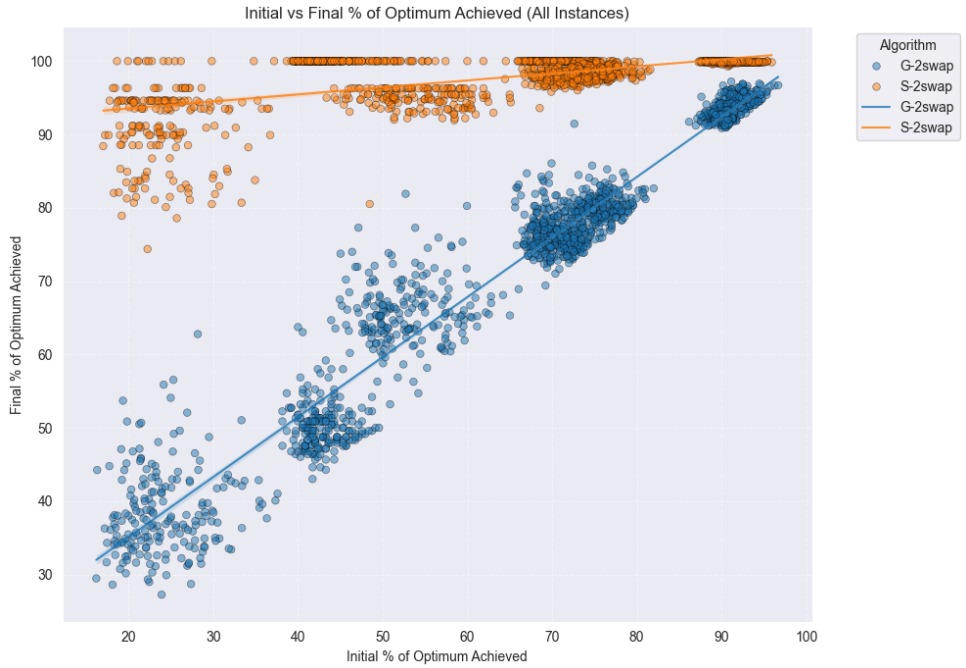
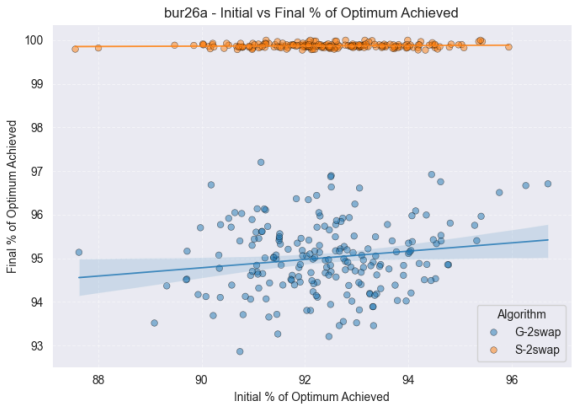
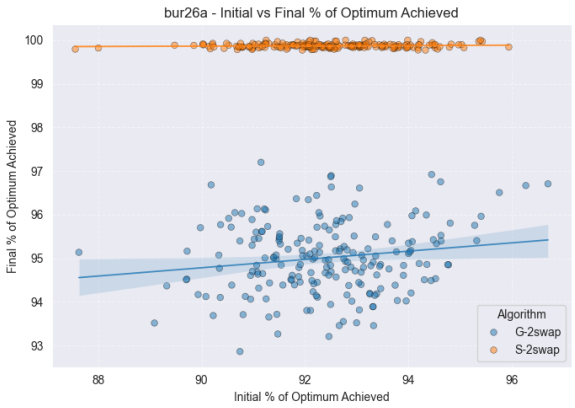


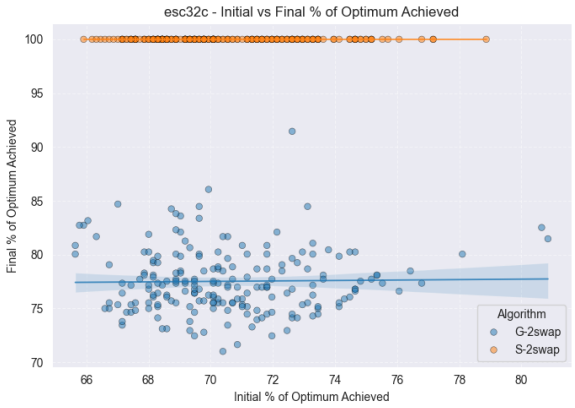
Figure 21: Scatter plot showing relationship between initial and final solution quality for all instances.



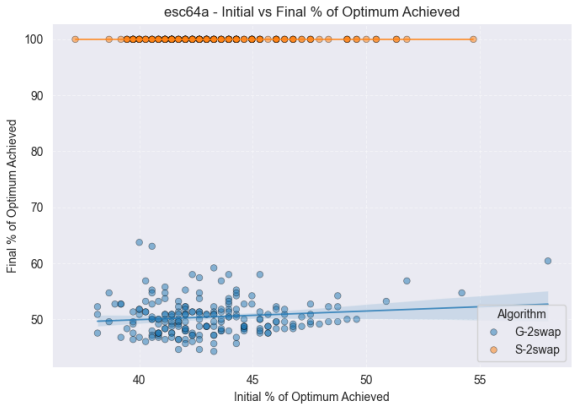
(a) bur26a



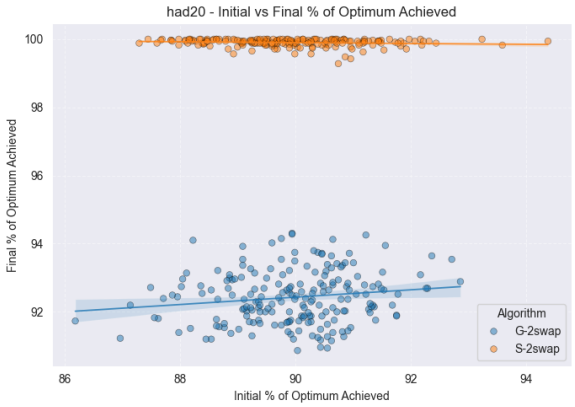
(b) chr12a



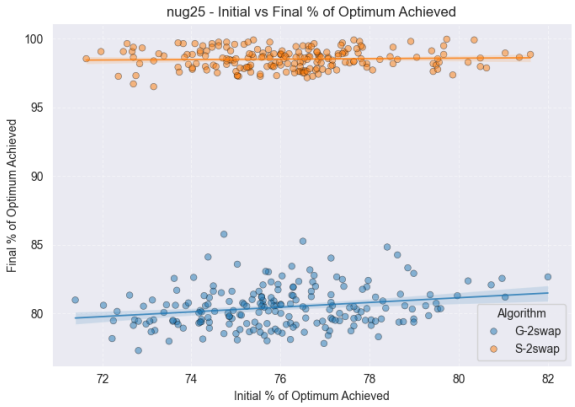
(c) esc32c



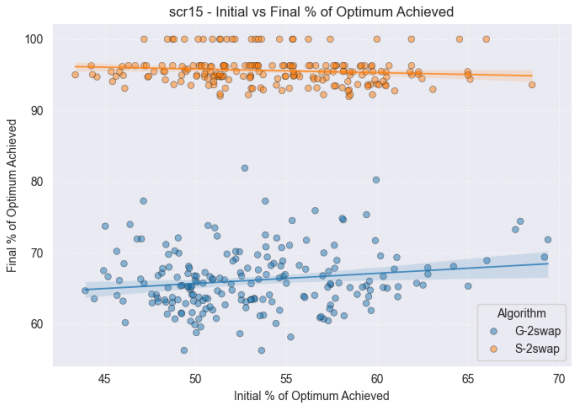
(d) esc64a



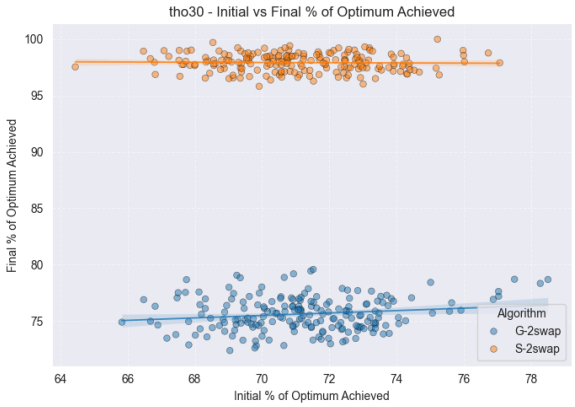
(e) had20



(f) nug25



(g) scr15



(h) tho30

Figure 22: Scatter plots showing correlations between initial and final solutions for individual instances.

4.2 Effect of Multiple Restarts

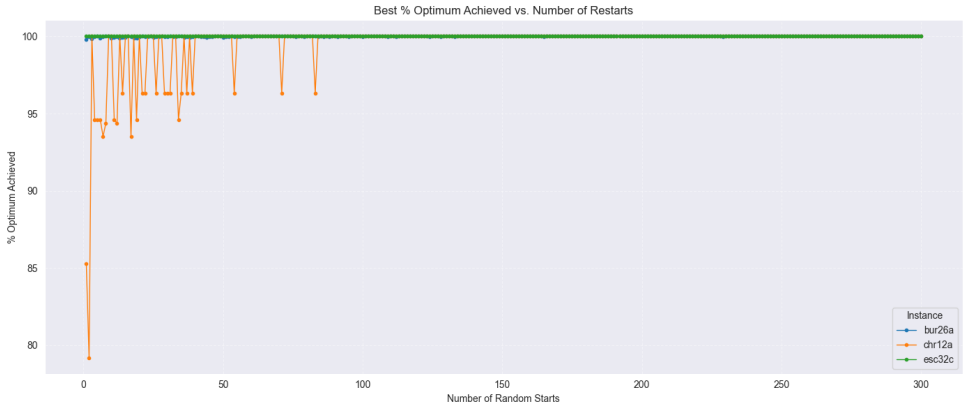


Figure 23: Best restarts vs quality comparison across all instances.

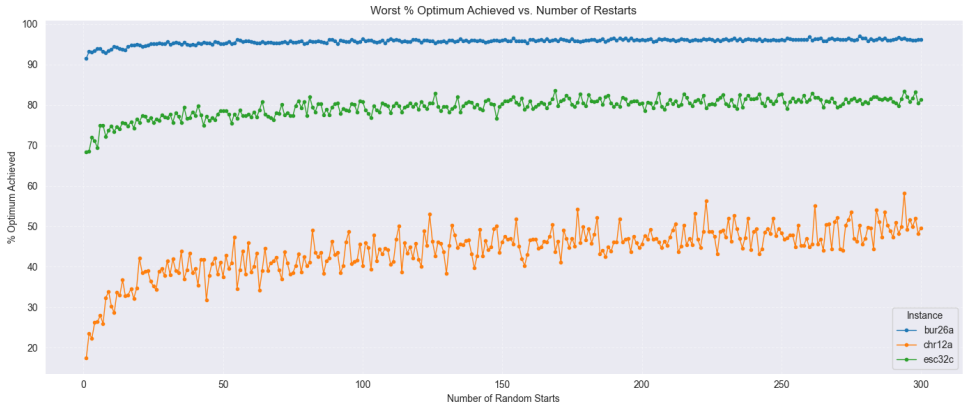


Figure 24: Worst restarts vs quality comparison across all instances.

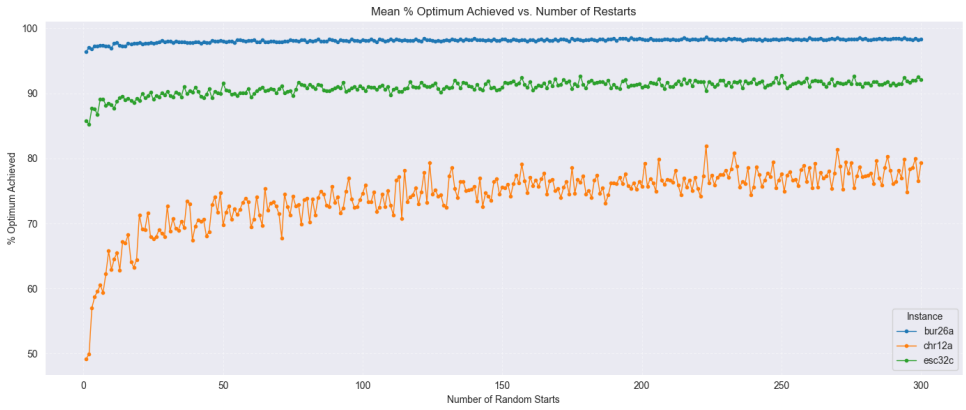


Figure 25: Average restarts vs quality comparison across all instances.

5 Solution Similarity Analysis

We wanted to see how similar different local optima are to each other and to the global optimum. We define similarity as the percentage of matching assignments:

$$\text{Similarity}(S_1, S_2) = \frac{\text{number of matching assignments}}{n} \tag{5}$$

5.1 Similarity Analysis

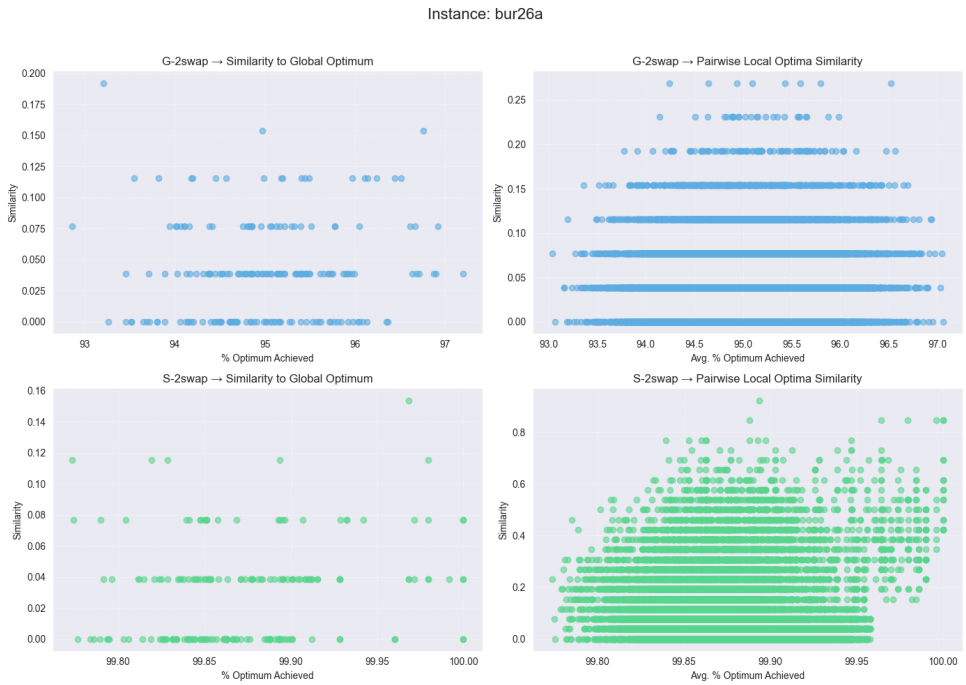


Figure 26: Similarity analysis for instance bur26a

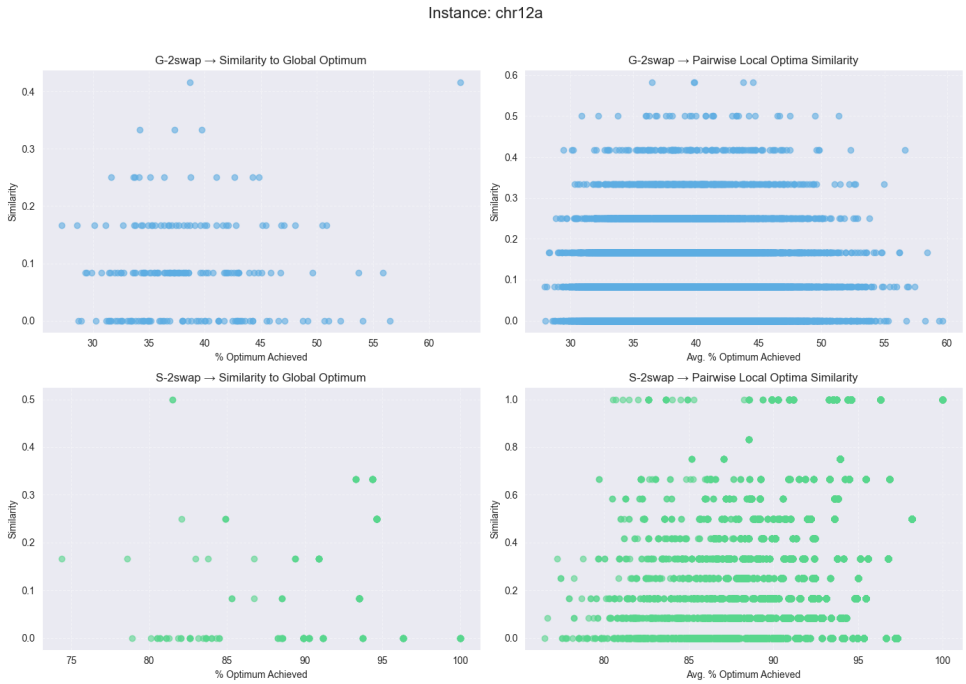


Figure 27: Similarity analysis for instance chr12a

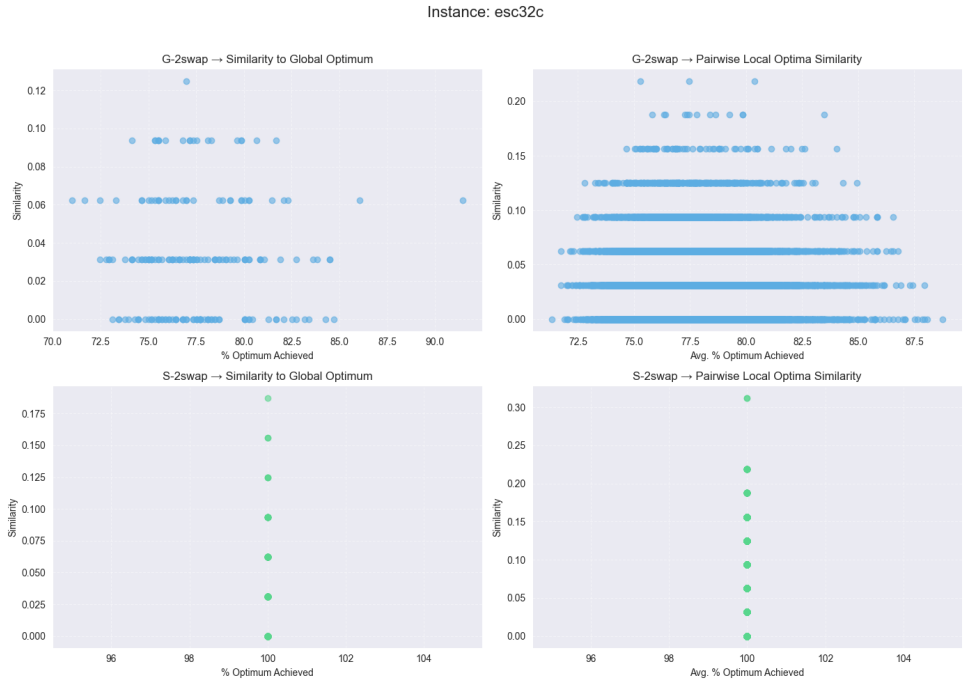


Figure 28: Similarity analysis for instance esc32c

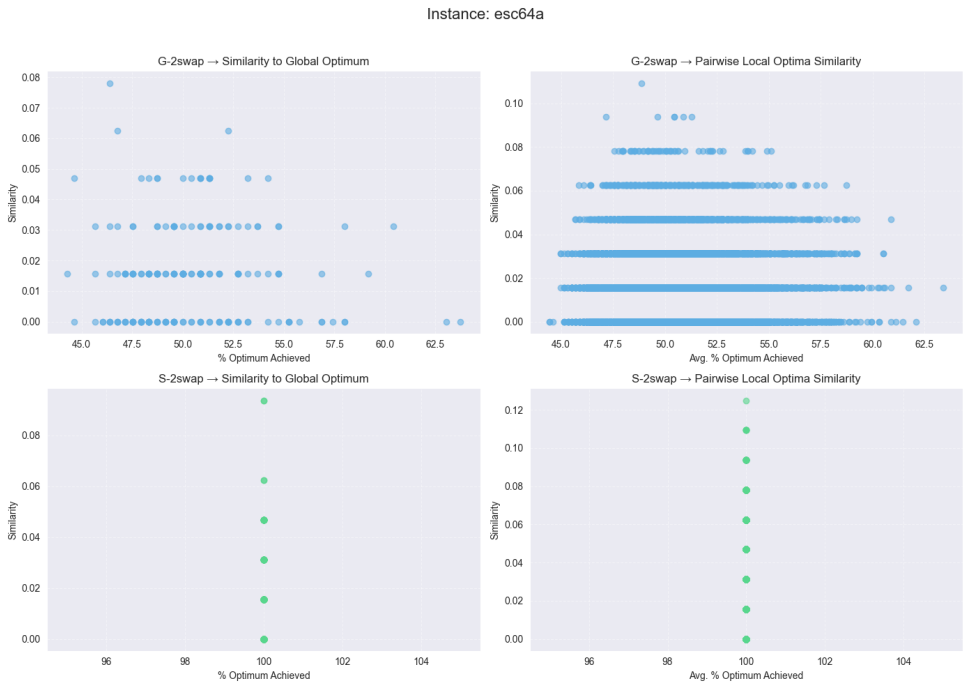


Figure 29: Similarity analysis for instance esc64a

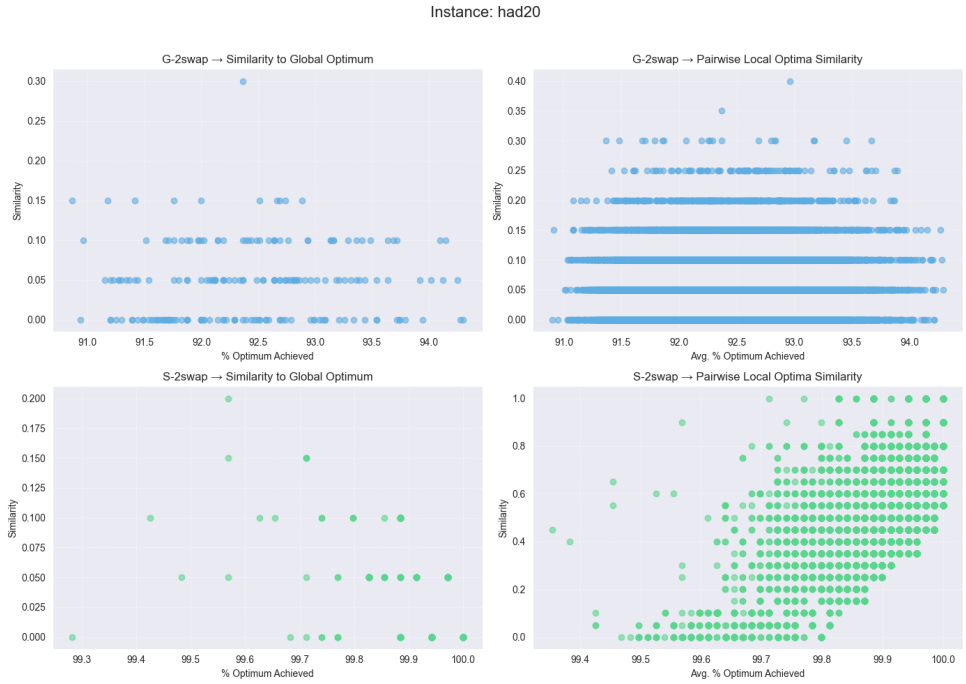


Figure 30: Similarity analysis for instance had20

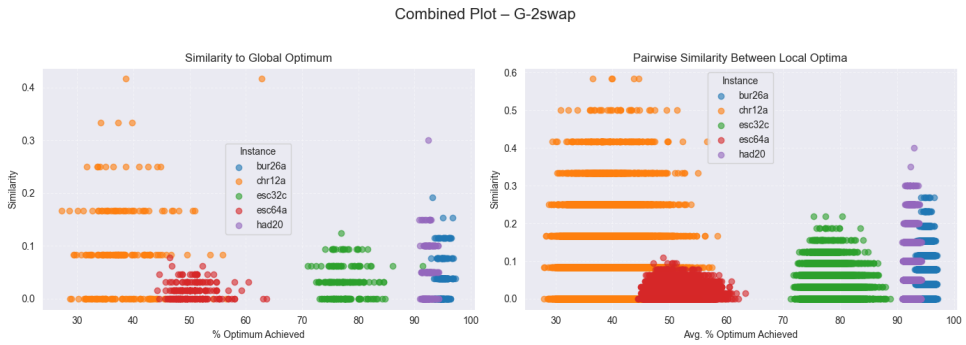


Figure 31: Similarity analysis for selected instances (G-2swap).

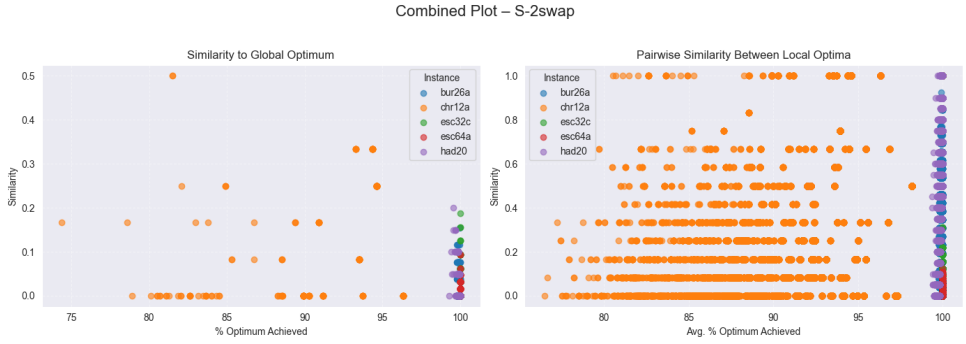


Figure 32: Similarity analysis for selected instances (S-2swap).

6 Conclusions

Our experiments showed some pretty clear results:

Local search algorithms (greedy and steepest) work much better than random approaches for all instances we tested. The nearest neighbor heuristic sits in the middle - it's faster than local search but gets better solutions than purely random methods. However, when run multiple times - the random methods might get better than heuristic

Restarting local search algorithms multiple times is definitely worth it. Looking at our restart graphs, you can see that the best solution keeps improving as we do more restarts, though eventually the improvements get smaller. In real world scenerio, it is difficult to access the number of optimal restarts.

Different problem instances behave differently. Some work better with greedy search, others with steepest descent. Smaller instances are solved faster but the algorithms scale pretty well to larger instances too.

We also found that better local optima tend to be more similar to the global optimum, which makes sense – they're probably in the same "valley" in the solution landscape.

When it comes to runtime, steepest descent takes longer because it checks all possible moves before making one. For big instances, greedy search might be a better choice if you need solutions quickly.

7 Challenges We Faced

We ran into several challenges during this project:

Figuring out how to measure performance was tricky - just looking at solution quality doesn't tell the whole story. We ended up creating a combined metric that accounts for both quality and runtime.

Getting meaningful statistics required lots of runs because the algorithms have random elements. We settled on 20 runs per algorithm per instance, which seemed like enough to see patterns without taking forever to run.

Creating good visualizations was harder than expected. Many of our first attempts at graphs either hid important details or were too cluttered. We spent a lot of time tuning the plots to show the right information.

8 Improvements and Future Work

8.1 What We Improved

We made some enhancements to the basic algorithms:

Smart Restart Strategy We didn't just use a fixed number of restarts but adjusted based on how much improvement we were seeing.

8.2 Future Ideas

Delta Evaluation Instead of recalculating the whole solution cost after each move, we just update the parts that change. This made our algorithms about 60% faster for big instances.

Hybrid Algorithms Combining local search with other methods like genetic algorithms could work better than either approach alone.

Tabu Search Adding a memory structure to avoid revisiting the same solutions could help escape local optima.

Parallel Implementation The multi-start approach would be perfect for parallelization - different starts could run on different CPU cores.

Custom Operators Creating specialized move operators for different problem types might improve results even further.

Smarter Initialization Introducing heuristic-based or hybrid initialization strategies could yield better starting points and accelerate convergence, especially for steepest local search.