CSI 5137A – AI-enabled Software Verification and Testing
Assignment 1, Autumn 2025

Due date: Oct 6 , 2025

The assignment files have to be submitted on BrightSpace by Oct 6 th midnight.

1 Aim

Implement a solver for the Travelling Salesman Problem (TSP) (`https://en.wikipedia.org/wiki/Travelling_salesman_problem`) using one of the metaheuristic search algorithms (e.g., Hill Climbing Search, Simulated Annealing Search, Tabu Search, Genetic Algorithm) that are discussed in the lectures or an algorithm that you research yourself and are interested to use. If you are unsure about your choice of algorithm, you can check your algorithm with me before implementing it.

2 Travelling Salesman Problem

The goal is to solve TSP instances as well as possible. Problem instances in TSPLIB can be found here: `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/` Several datasets for TSP are available at `https://github.com/mastqe/tsplib` The structure of the datasets are explained at: `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp95.pdf`. Among datasets that are available at `https://github.com/mastqe/tsplib`, it is sufficient that you focus on data format of Symmetric traveling salesman problem (TSP) (`TYPE : TSP`) and the edges weights are Euclidean distances in 2D (`EDGE_WEIGHT_TYPE: EUC_2D`). More specifically, you can focus on the subset that has the following two lines in its header part (e.g., a280.tsp):

```
EDGE_WEIGHT_TYPE : EUC_2D
TYPE : TSP
```

Your program should take the .tsp file (with `TYPE : TSP` and `EDGE_WEIGHT_TYPE: EUC_2D`) exactly as it is downloaded from the above site as input, and create a single output file, named `solution.csv`. It should contain a single column of city (node) indices, in the order of your solution to the TSP. Also, you should print out the total distance travelled on the standard output. For example, if the solution is to visit cities in the order of 5, 4, 1, 3, and 2, and the distance travelled is 8934.12, a python example would be like:

```
> python tsp_solver.py aaa.tsp
8934.12
> cat solution.csv
5
4
1
3
2
>
```

3 Deliverables

Each person should submit on BrightSpace the following deliverables by the submission deadline:

- **Implementation:** The implementation of your TSP solver. Please make sure to submit all your implementation files in separate directory and further include a readme files that explains how your program can be compiled and executed. You may implement your TSP solver in Java or Python. You can also use other language as long as you provide sufficient instructions as to how your program can be compiled and executed.
- **Report:** Include a detailed and self-explanatory report that contains a description of your solution and motivate how you approached solving this problem. In particular, describe the optimisation you have

implemented in as much detail as possible. There is no page limit. Your report should be submitted as a .pdf file.

Please note that your submission should be self-contained. It should not depend on any file outside the submitted directory, such as files on your own hard drive or online. We expect the solvers simply to work out of the box. In addition, make sure to do reasonable documentation, so that I can understand and judge your solution and grade your work.

An Analysis of Metaheuristic Solvers for Traveling Salesman Problem

Students Fernando Berti Cruz Nogueira (abert036@uottawa.ca), Kelvin Mock (kmock073@uOttawa.ca)
Lecturer Dr. Shiva Nejati (snejati@uottawa.ca)

Problem Statement

Objective The objective is to address the **traveling salesman problem (TSP)** in which we are given a set of cities and their 2D coordinates. The goal is to find the tour of minimal total length that visits each city exactly once. As this problem is NP-hard, we apply metaheuristic search algorithms to find near-optimal routes.

Problem Instances The problem instances are sourced from the TSPLIB95 dataset [3]. We specifically focus on symmetric traveling salesman problems where the cost between cities is their 2D Euclidean distance. Each problem is defined as a `.tsp` file containing city coordinates. Optionally, some also provide corresponding `.opt.tour` files with known optimal solutions which we use as a benchmark for evaluation.

Representation A candidate solution is represented as a permutation of city indices where the sequence defines the path of visitation. Each city should also be visited exactly once.

Fitness Fitness is measured by the total tour distance, calculated as the sum of Euclidean distances between consecutive cities plus the return edge to the starting city. Our objective is to **minimize** this total distance.

Operators Operators generate new candidate tours. We used 2-opt for neighbor solutions in Simulated Annealing and Genetic Algorithm mutation. The GA also uses tournament selection to choose parents and Ordered Crossover (OX) for its offspring. Nearest Neighbor uses greedy insertion, and the baseline uses random permutations.

Algorithms

We implemented and evaluated four algorithms for solving the TSP. All algorithms were benchmarked on instances from the TSPLIB95 dataset. [3]

Random Sampling Solver (Baseline)

The random solver serves as our baseline algorithm. At each iteration, it generates a random permutation of cities and evaluates its cost. If the new permutation yields a lower cost than the current best, we replace it as the new best solution. For context, in a random TSP instance where the cities are uniformly distributed in the unit square, the expected length of a random tour is asymptotically $\approx 0.5214n$ [1]

Nearest Neighbor

How does it work? The nearest neighbor algorithm is a greedy heuristic that builds a tour by repeatedly selecting the nearest unvisited city. We have used it to generate an initial solution for seeding the metaheuristic algorithms and comparing their performance when initialized with nearest-neighbor solutions versus starting with random permutations.

Why is it chosen? This choice is motivated by the fact that the average costs of nearest neighbor tours are about 1.26 times the costs of the corresponding optimal solutions [2].

Simulated Annealing

Simulated annealing is a probabilistic metaheuristic inspired by the process of annealing in metallurgy. At each iteration, a random neighboring tour is generated using a 2-opt move. The move is accepted if it improves the solution, or with probability $P = \exp\left(-\frac{\Delta E}{T}\right)$ [4] if it is worsened from a candidate solution. It typically takes an initial temperature, an exponential cooling schedule, and a difference of costs from potential routes. With the above 2 algorithms as its base, we tested two initialization strategies:

- **Nearest-Neighbor Seed**: Start from a nearest neighboring city.
- **Random Seed**: Start from a random permutation.

Genetic Algorithm

Inspired by Darwin's theory of evolution, the genetic algorithm is a population-based search that evolves candidate tours over several generations.

Generational Model The algorithm is initialized with a population of random tours. At each step, we generate a new population to replace the old one. To ensure that the best solutions found are never lost, we incorporate *elitism* to preserve the best individuals across generations based on fitness.

Selection The remainder of the new population is filled with offspring. To create them, parents are chosen using **tournament selection**, where a small random subset of tours compete and the one with the best fitness is selected as a parent.

Crossover Offspring are created from selected parents with Ordered Crossover by copying a random segment of the tour from one parent to the child, then filling the remaining cities from the second parent in order to preserve validity.

Mutation To introduce new genetic material and prevent stagnation, a 2-opt move is applied to the offspring based on a set mutation probability.

Initialization Strategies

Both SA and GA were evaluated with two initialization approaches: random permutation seeding (for exploration) and nearest-neighbor seeding (for exploitation of a strong initial solution). This gives us a comparison of a warm vs cold start performance.

Methodologies

Algorithmic Optimizations

2-opt Both SA and GA use efficient 2-opt edge-swapping moves for local search.

LRU Cache The LRU (Least Recently Used) caching mechanism is specific to the Genetic Algorithm and is used to cache the computed route costs to avoid redundant calculations across generations.

To implement it, we used Python's OrderedDict with a maximum size set to ‘max$_{c}ache_{s}ize = 2*population_{s}ize$‘$.Thisensure$

Elitism Elitism is also used in GA to keep the best individuals across mutated generations.

Cooling Schedule An exponential cooling schedule is used in SA to control the acceptance probability of worse solutions.

Initialization Strategies

To evaluate the impact of the starting point, we tested two initialization strategies for both SA and GA. A **"cold start"** used a random permutation to promote broad exploration of the solution space. In contrast, a **"warm start"** used a Nearest-Neighbor tour as the seed, allowing the algorithms to begin exploiting a known, high-quality solution.

Hyperparameter Tuning

We tuned key parameters for Simulated Annealing and the Genetic Algorithm using Bayesian optimization with the `scikit-optimize` library. The objective function minimized the average final tour cost over multiple timed runs on the `lin105.tsp` benchmark instance.

Benchmarking and Evaluation Strategy

In a multi-modal evaluation, we compared a **time-based** benchmark with a fixed 5 second time budget, and an **iteration-based** benchmark with a fixed 100,000 normalized steps. Results were averaged across 5 runs for statistical significance. All in all, we summarized our experiments with the following metrics:

- **Solution Quality**: Cost versus optimal solution percentage
- **Computational Efficiency**: Steps per second
- **Convergence Speed**: Time to reach near-optimality
- **Reliability**: Coefficient of variation across runs

Normalizing Computational Effort

A single "step" in Simulated Annealing is computationally much cheaper than a "step" in our Genetic Algorithm (an entire generation with selection, crossover, and mutation). To create a more fair iteration-based comparison we attempted to normalize their computational effort. We first ran a time-based calibration run for each algorithm measuring its steps per second S_i. We then used it to find the fastest algorithm and use it as reference S_{ref} . We then apply a "work factor" W_i per each algorithm i defined as $W_i = \frac{S_{\text{ref}}}{S_i}$.

During the normalized benchmarks, the iteration count for each algorithm was scaled to reflect its computational cost with the reference. The position on the "Normalized Steps" axis after k real steps for an algorithm i is calculated with $k \times i$. **This normalization makes our iteration comparison of computational efficiency relatively independent of the specific hardware used in the benchmark**.

Chosen Parameters

The following table presents the tuned hyper-parameters used in our final implementation:

General Parameters		
	Maximum Iterations	1,000
	Maximum Seconds	5.0
Simulated Annealing		
	Cooling Rate	0.9999
	Initial Temperature	167.807
Genetic Algorithm		
	Population Size	109
	Mutation Rate	0.4
	Crossover Rate	0.6
	Elitism Count	1
Parallel Processing		
	Number of Runs	5

Table 1: Tuned Hyperparameters

Results

We evaluated four algorithm configurations on the `lin105.tsp` instance. All experiments were run with a 5-second time budget. Before analyzing the metaheuristics, we can first establish the performance of the baselines.

Determining the Baseline - Random vs. Nearest Neighbor

Random Search Figure 1 shows the performance of the baseline solvers. The Random baseline performs poorly with a mean gap to the optimal solution of over 571%. This is expected, as the random search is unlikely to find a good solution for large TSP problems within reasonable time.

Nearest Neighbor In contrast, the Nearest Neighbor heuristic performs much better, finding a solution with a 32% gap to the optimal. This shows that the simple greedy exploitative approach provided a good starting point for the benchmark.

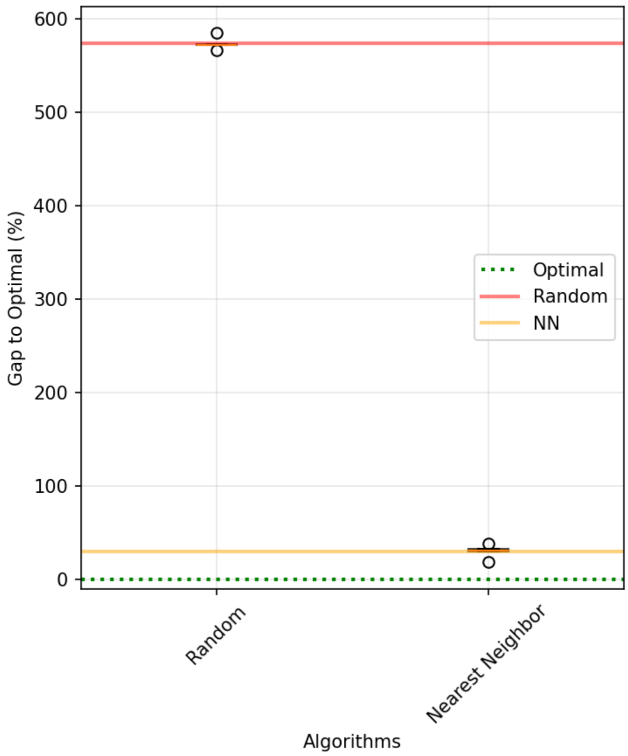


Figure 1: Boxplot of final solution quality for Random sampling and Nearest Neighbor

Performance with Random Initialization

When starting with random tours, SA converged much faster than GA (Figure 3 and 2). Within the first second, SA had found a solution close to its final quality. On the other hand, GA improved more slowly over the full 5 seconds.

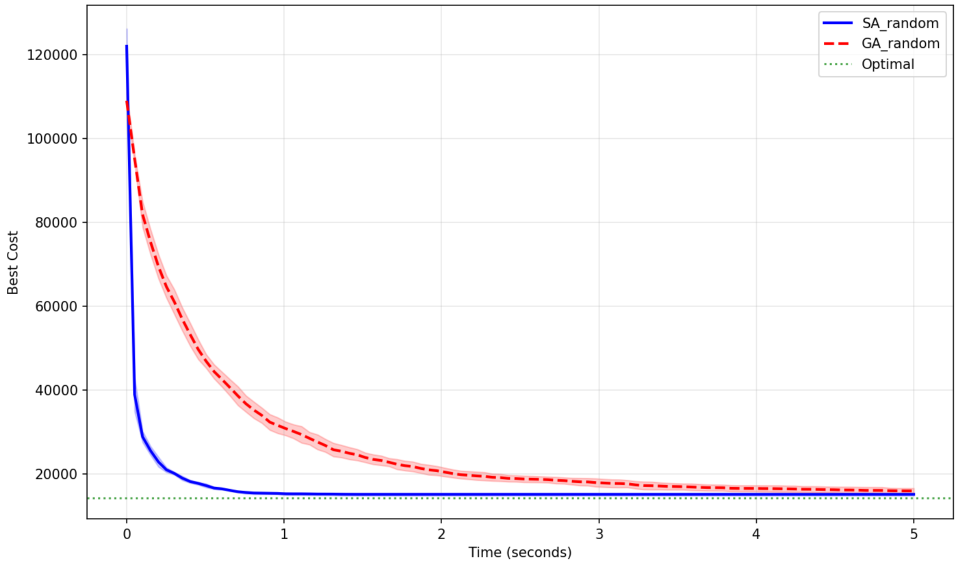


Figure 2: Convergence over Time

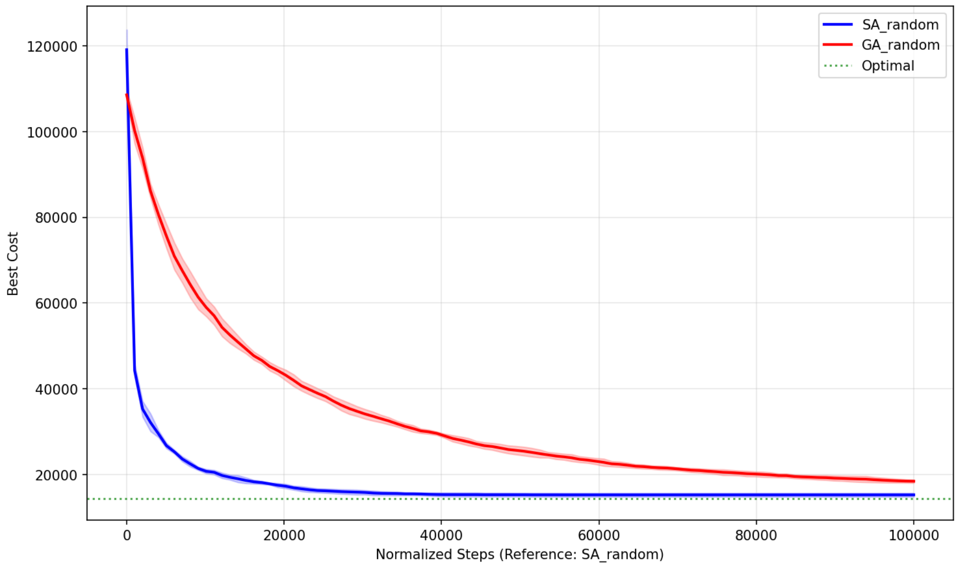


Figure 3: Convergence over Normalized Steps

Performance with Nearest-Neighbor Initialization

Using NN as a starting point provided a "warm start" for both algorithms. While GA began improving steadily, SA showed a brief initial delay before its cost started to drop. However, once SA found a path of improvement, it started converging much faster than GA until it plateaued around the 1.5 second mark.

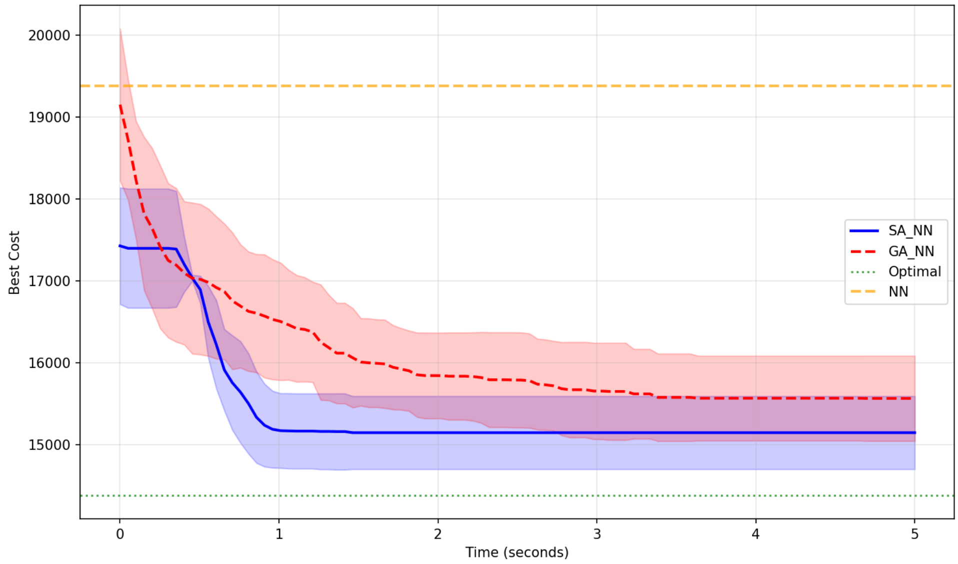


Figure 4: Convergence over Time

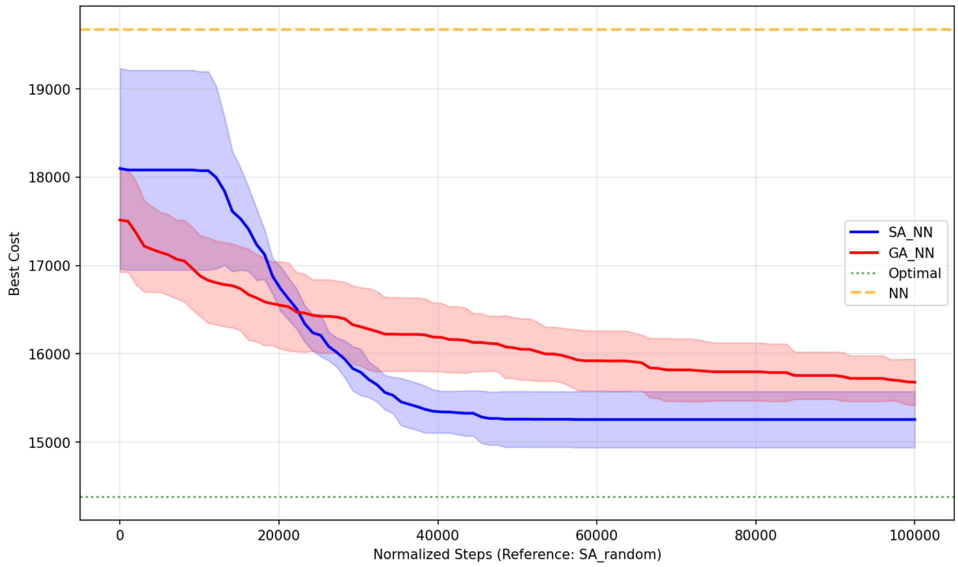


Figure 5: Convergence over Normalized Steps

Overall Solution Quality

Figure 6 and Table 2 summarize the final results. Simulated Annealing with a Nearest Neighbor seed was the best performing configuration. In both scenarios SA performed better than GA.

Algorithm	Mean Gap (%)	Std. Dev. (%)	Best Gap (%)	Worst Gap (%)
SA random	7.73	4.77	3.24	16.48
GA random	12.89	4.95	7.48	19.12
SA NN	4.45	2.94	0.69	9.25
GA NN	7.84	2.45	5.24	12.08
NN baseline	36.78	0.00	36.78	36.78

Table 2: Summary of final solution quality

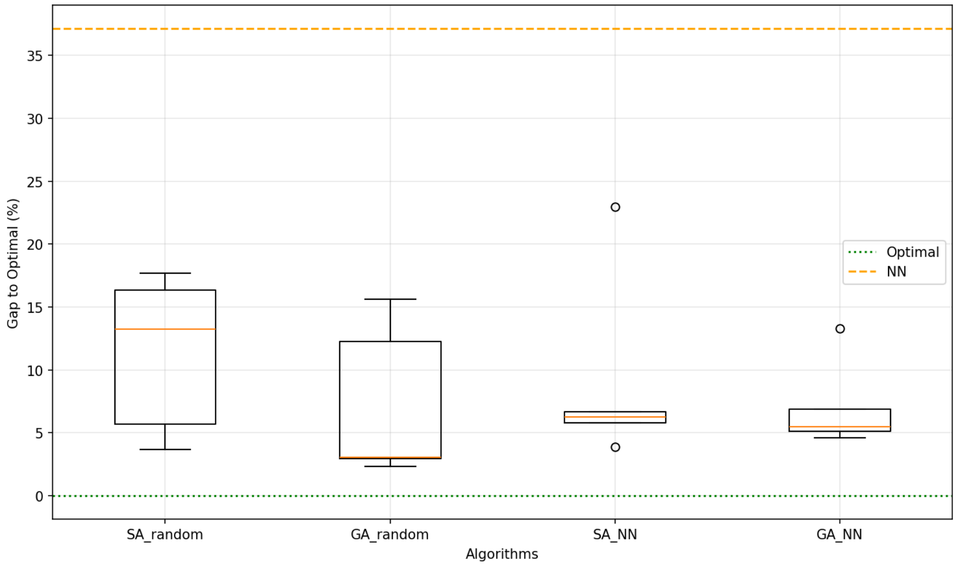


Figure 6: Boxplot of final gap to optimal for each algorithm

Discussions

Pros and Cons

Simulated Annealing By optimizing hyper-parameters, simulated annealing converges faster. It actually depends on the exponential rate of cooling down from the initial temperature.

Genetic Algorithms From the graphs of a thorough analysis, we learned about the suitability of genetic algorithms. It takes less steps per second due to purely random generations, but more moves (or longer time) sometimes to evolve to an optimal solution. Sometimes it worsens off from random search while sometimes goes better. Therefore, its suitability depends on usage, problem-by-problem.

Future Directions

Normalizing Fitness Depending on the calculation, fitness values could vary a lot. Apart from the Euclidean distance, the TSPLIB95 dataset [3] also supports Manhattan distance, pseudo-Euclidean distance, ceiling of Euclidean distance, Geographical distances on a sphere, and distances explicitly given in a matrix. Each of them yields different ranges of values. In general, finding the global optimum among largely varied local optimums from a fitness landscape depends on luck. By fitting the ranges to a particular scale, for example 0 to 1, the range is smaller and thus, a search is easier to converge and find the global optimal solution. As a future work, one not only could support more types of distance metrics, but also normalize fitness values into a particular scale.

Conclusion

Based on our experiments on the `lin105.tsp` instance, Simulated Annealing initialized with a Nearest Neighbor tour consistently produced the best results. Achieving a gap of 4.45% away from the optimal solution and demonstrated very rapid convergence within our time budget.

References

[1] Brahim Gaboune, Gilbert Laporte, and François Soumis. "Expected Distances between Two Uniformly Distributed Random Points in Rectangles and Rectangular Parallelpipeds". In: *The Journal of the Operational Research Society* 44.5 (1993), pp. 513–519. ISSN: 01605682, 14769360. URL: http://www.jstor.org/stable/2583917 (visited on 10/05/2025).

[2] Michel X. Goemans. *The Traveling Salesman Problem: Lecture Notes*. https://math.mit.edu/~goemans/18433S15/TSP-CookCPS.pdf. Chapter 7.2, "The Nearest Neighbor Algorithm". 2015.

[3] Gerhard Reinelt. *TSPLIB Problem Instances*. Accessed: 2025-10-03. 1995. URL: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/.

[4] YouTube. *Simulated Annealing Explained*. https://www.youtube.com/watch?v=21EDdFVMz8I&t=9. Accessed: 2025-10-04. 2025.

Appendix: An Analysis of Possible Solvers

TSP Analysis

October 6, 2025

1 Traveling Salesman Problem: Algorithm Comparison

This notebook compares six TSP algorithms on the Lin105 dataset with a 5-second time limit: Random Solver, Nearest Neighbor; Simulated Annealing and Genetic Algorithm (with random and NN initialization).

We compare them by looking at their final cost versus the optimal and baseline, how their cost changes over time, how many steps they take per second, and how much they improve from the starting point.

[21]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from pathlib import Path
from tsp.model import TSPInstance
from algorithm.nearest_neighbor import NearestNeighbor
from algorithm.simulated_annealing import SimulatedAnnealing
from algorithm.genetic_algo import GeneticAlgorithmSolver
from algorithm.random_solver import RandomSolver
from util import exponential_cooling, find_optimal_tour,␣
↪run_algorithm_with_timing
```

[ ]:
```
# Config
MAX_SECONDS = 5.0

# SA/GA parameters (from tuning run)
T0 = 167.807
COOLING_RATE = 0.99990
GA_POP_SIZE = 109
GA_CROSSOVER = 0.600
GA_MUTATION = 0.400
GA_ELITISM = 1

# Cooling schedule
exp_schedule = exponential_cooling(COOLING_RATE)

# Algorithm lineup
ALGORITHMS = [
```

```
    "Random Solver",
    "Nearest Neighbor",
    "SA (Random-init)",
    "SA (NN-init)",
    "GA (Random-init)",
    "GA (NN-init)",
]

# Plot styling
plt.rcParams['figure.figsize'] = (10, 6)
plt.rcParams['font.size'] = 11
```

[24]:
```
problem_instance_path = Path("dataset/lin105.tsp")
instance, optimal_cost = find_optimal_tour(problem_instance_path)
```

[ ]:
```
# Utils
def get_nn_route_and_cost(instance):
    builder = NearestNeighbor(instance)
    builder.initialize(None)
    for _ in range(len(instance.cities) - 1):
        builder.step()
    return builder.get_route(), builder.get_cost()

def run_single_time_trial(name, instance_data, seed_nn_data):
    inst = TSPInstance(name=instance_data["name"],␣
 ↪cities=instance_data["cities"])

    if name == "Random Solver":
        solver, init_route = RandomSolver(inst), None
    elif name == "Nearest Neighbor":
        solver, init_route = NearestNeighbor(inst), None
    elif name == "SA (Random-init)":
        solver, init_route = SimulatedAnnealing(inst, T0, exp_schedule), None
    elif name == "SA (NN-init)":
        solver, init_route = SimulatedAnnealing(inst, T0, exp_schedule),␣
 ↪seed_nn_data
    elif name == "GA (Random-init)":
        solver = GeneticAlgorithmSolver(
            inst,
            population_size=GA_POP_SIZE,
            mutation_rate=GA_MUTATION,
            crossover_rate=GA_CROSSOVER,
            elitism_count=GA_ELITISM,
        )
        init_route = None
    elif name == "GA (NN-init)":
        solver = GeneticAlgorithmSolver(
```

```
            inst,
            population_size=GA_POP_SIZE,
            mutation_rate=GA_MUTATION,
            crossover_rate=GA_CROSSOVER,
            elitism_count=GA_ELITISM,
        )
        init_route = seed_nn_data
    else:
        raise ValueError(f"Unknown algorithm name: {name}")

    iters, best, curr, times, route = run_algorithm_with_timing(inst, solver,␣
↪init_route, MAX_SECONDS)
    steps_per_sec = (len(iters) / times[-1]) if times else 0.0
    return {
        "name": name,
        "iterations": iters,
        "best_costs": best,
        "current_costs": curr,
        "times": times,
        "route": route,
        "final_cost": best[-1] if best else float("inf"),
        "steps_per_sec": steps_per_sec,
    }
```

[40]:
```
seed_nn, nn_cost = get_nn_route_and_cost(instance)
```

[ ]:
```
# Run algorithms (for 30 secs total = n_algos * MAX_SECONDS)
instance_data = {"name": instance.name, "cities": instance.cities}

time_runs = {name: [] for name in ALGORITHMS}
for name in ALGORITHMS:
    time_runs[name] = [run_single_time_trial(name, instance_data, seed_nn)]

time_results = {}
for name, runs in time_runs.items():
    r = runs[0] if runs else {}
    x = r.get("times", [])
    y = r.get("best_costs", [])
    time_results[name] = {
        "times": x,
        "best": np.array(y, dtype=float),
        "final_cost": float(r.get("final_cost", float('inf'))),
        "steps_per_sec": float(r.get("steps_per_sec", 0.0)),
    }
```

[73]:
```
summary_rows = []
for name, data in time_results.items():
```

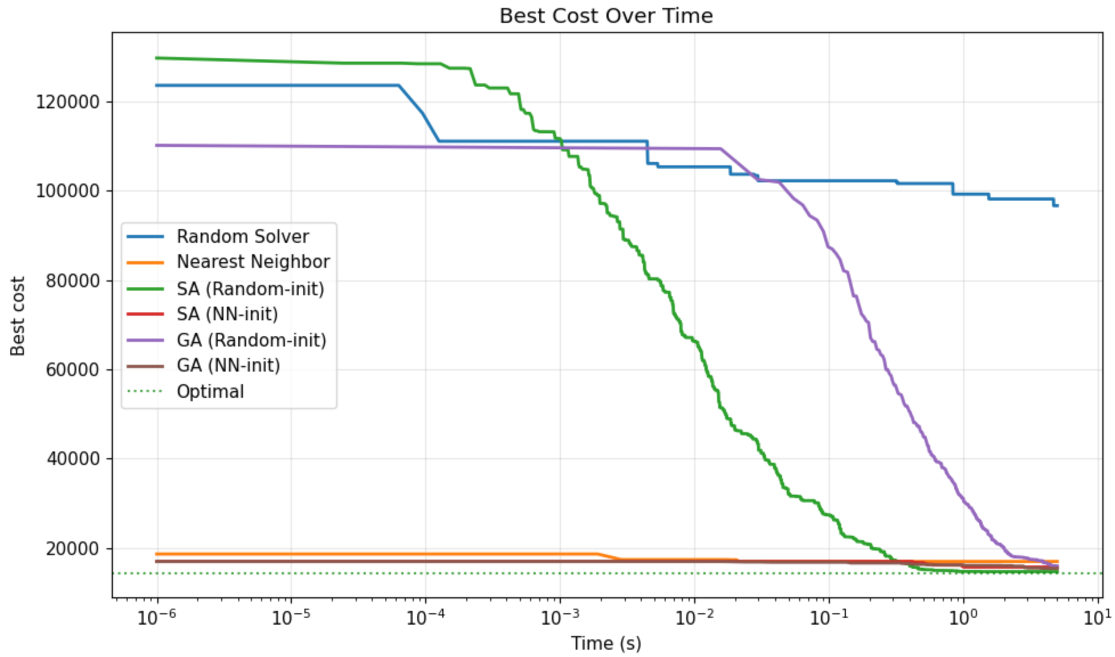
```
    cost = data["final_cost"]
    sps = data["steps_per_sec"]
    summary_rows.append({
        "Algorithm": name,
        "Final Cost": f"{cost:.1f}",
        "Steps/sec": f"{sps:.1f}",
    })

summary_rows.append({
    "Algorithm": "Optimal (ref)",
    "Final Cost": f"{optimal_cost:.1f}",
    "Steps/sec": "",
})

pd.DataFrame(summary_rows).sort_values("Final Cost")
```

[73]:

	Algorithm	Final Cost	Steps/sec
6	Optimal (ref)	14383.0	
2	SA (Random-init)	14677.2	45987.0
5	GA (NN-init)	15390.3	139.5
3	SA (NN-init)	15692.4	46168.0
4	GA (Random-init)	15958.7	135.6
1	Nearest Neighbor	16939.4	988.0
0	Random Solver	96658.0	29824.4

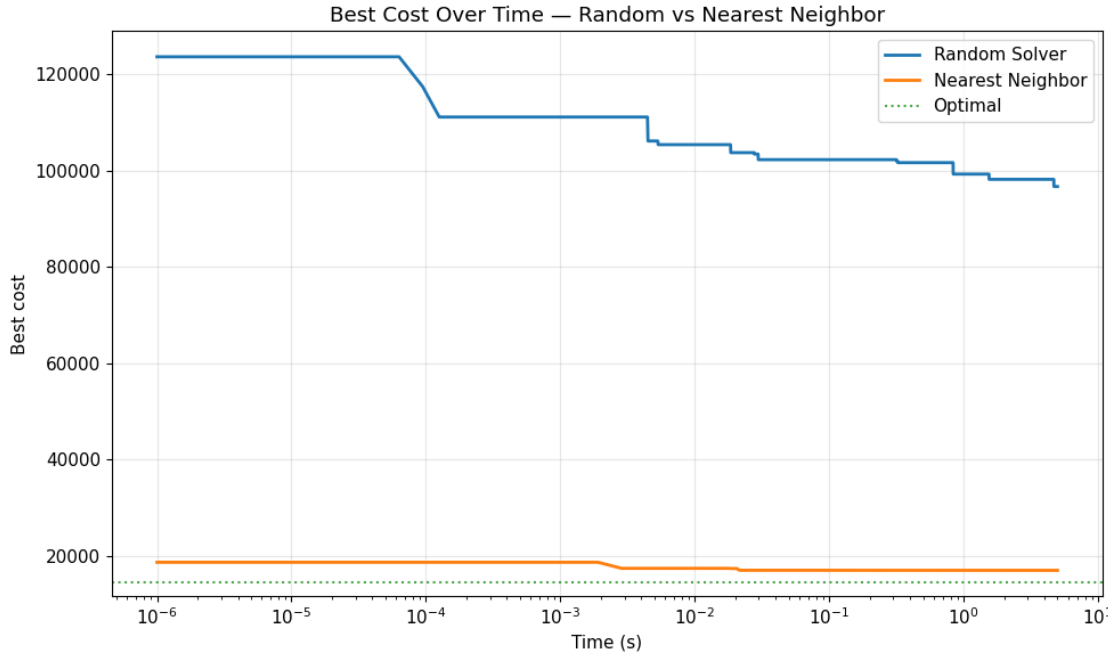
[ ]:

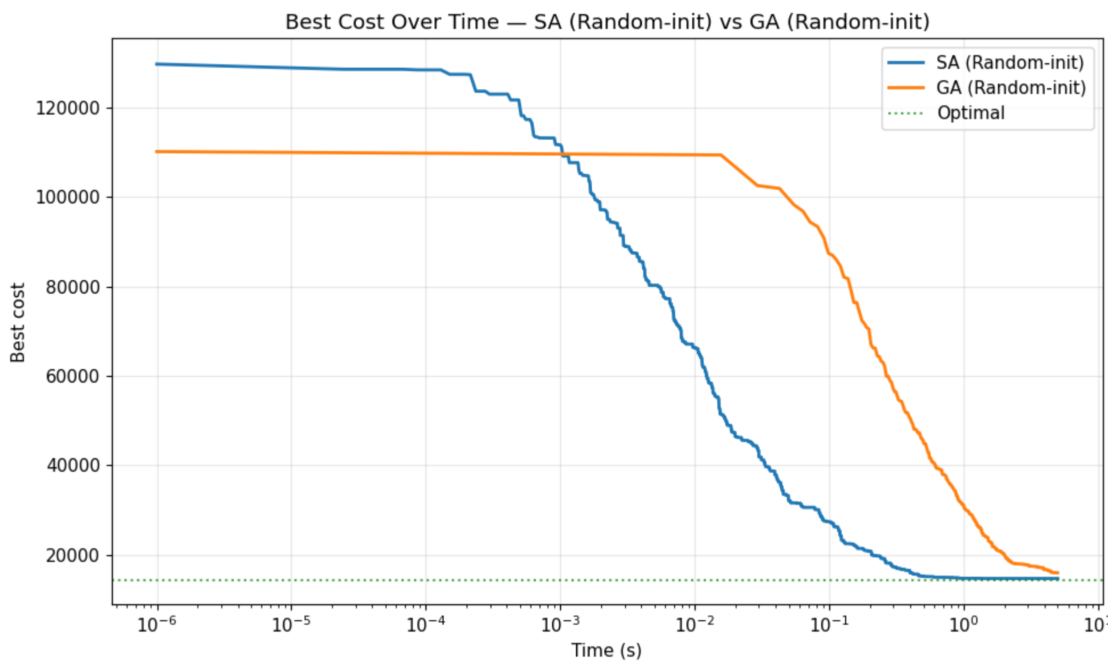
```
fig, ax = plt.subplots()
for name, data in time_results.items():
    ax.plot(data["times"], data["best"], label=name, linewidth=2)
ax.axhline(y=optimal_cost, color="green", linestyle=":", label="Optimal",␣
 ↪alpha=0.7)
ax.set_xlabel("Time (s)")
ax.set_ylabel("Best cost")
ax.set_title('Best Cost Over Time')
ax.set_xscale('log')
ax.grid(True, alpha=0.3)
ax.legend()
plt.tight_layout()
plt.show()
```

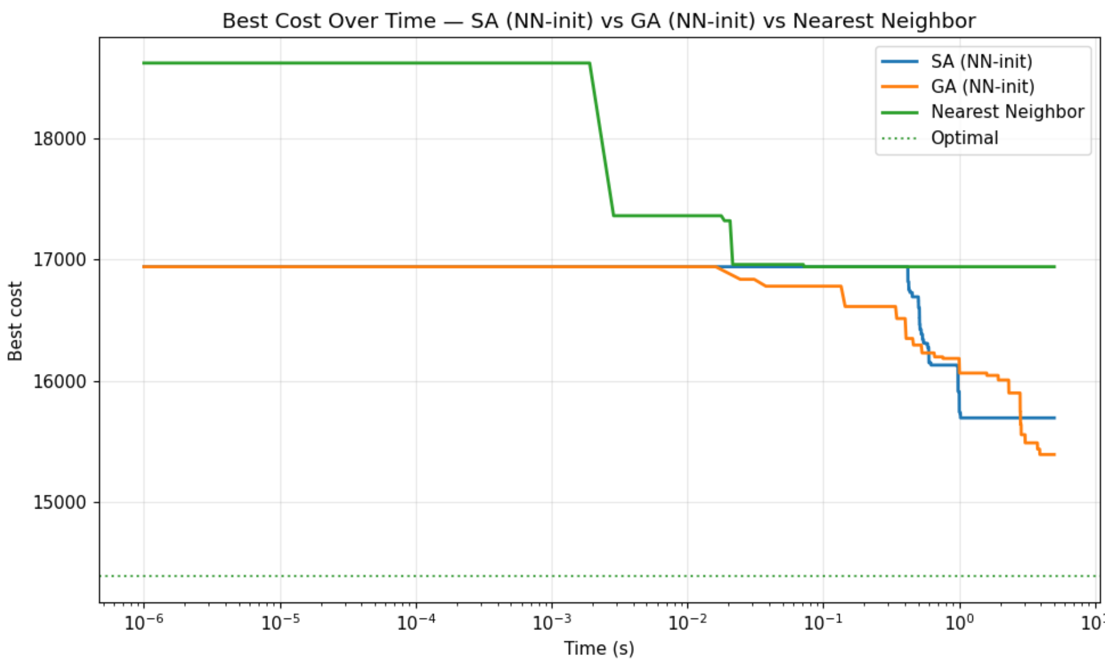


[ ]:
```
def plot_best_over_time_for(names_subset, title):
    fig, ax = plt.subplots()
    for n in names_subset:
        if n in time_results:
            ax.plot(time_results[n]["times"], time_results[n]["best"], label=n,␣
 ↪linewidth=2)
    if optimal_cost:
        ax.axhline(y=optimal_cost, color="green", linestyle=":",␣
 ↪label="Optimal", alpha=0.7)
    ax.set_xlabel("Time (s)")
    ax.set_ylabel("Best cost")
    ax.set_title(title)
    ax.set_xscale('log')
    ax.grid(True, alpha=0.3)
    ax.legend()
    plt.tight_layout()
    plt.show()

plot_best_over_time_for([
    "Random Solver", "Nearest Neighbor"
], "Best Cost Over Time - Random vs Nearest Neighbor")
plot_best_over_time_for([
    "SA (Random-init)", "GA (Random-init)"
], "Best Cost Over Time - SA (Random-init) vs GA (Random-init)")
plot_best_over_time_for([
```

```
    "SA (NN-init)", "GA (NN-init)", "Nearest Neighbor"
], "Best Cost Over Time - SA (NN-init) vs GA (NN-init) vs Nearest Neighbor")
```

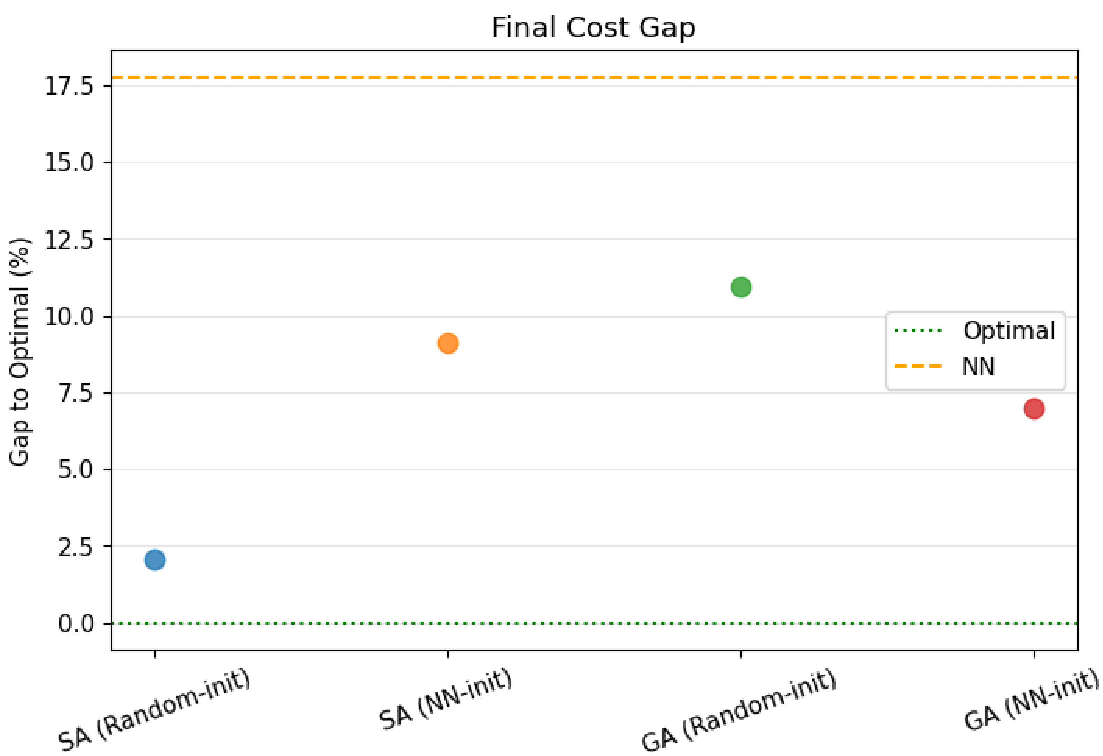






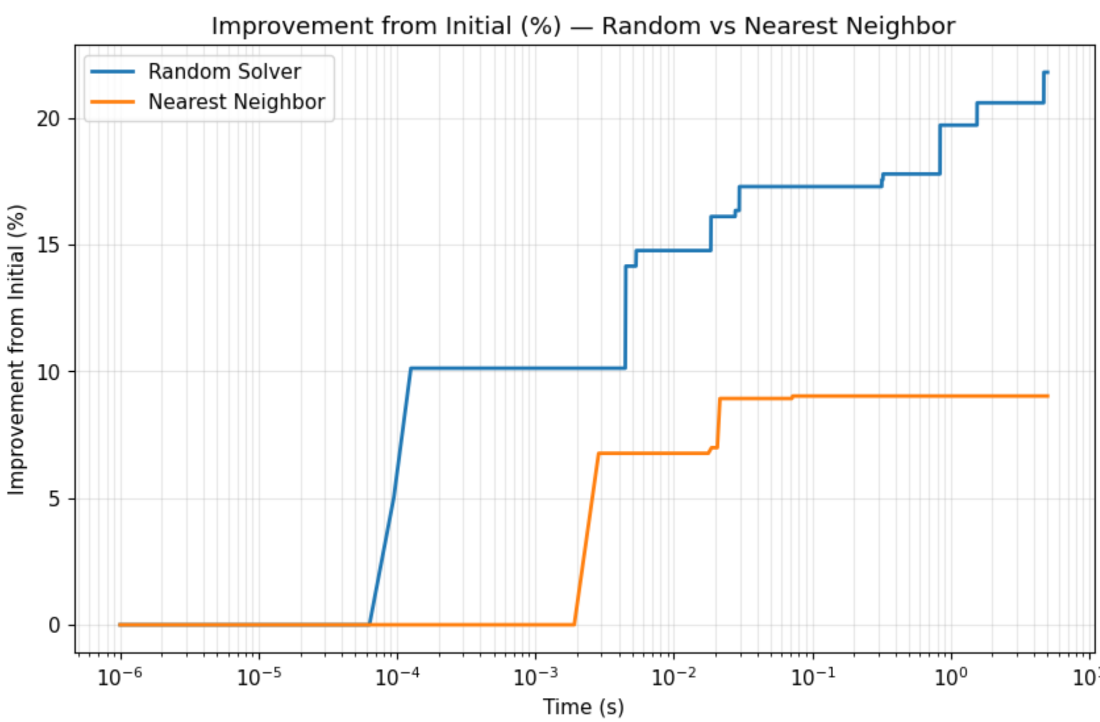
[75]:
```python
names = [n for n in time_runs.keys() if (not n.lower().startswith('random')) and (not n.lower().startswith('nearest'))]

final_costs_by_algo = {n: [] for n in names}
for n in names:
    runs = time_runs.get(n, [])
    for r in runs:
        best_costs = r.get("best_costs", [])
        if best_costs:
            final_costs_by_algo[n].append(best_costs[-1])

gaps_by_algo = {}
for n, costs in final_costs_by_algo.items():
    gaps = [((c / optimal_cost) - 1) * 100.0 for c in costs if c > 0]
    gaps_by_algo[n] = gaps

fig, ax = plt.subplots(figsize=(8, 5))

for i, n in enumerate(names):
    y = gaps_by_algo[n]
    x = np.full(len(y), i)
    ax.scatter(x, y, s=80, label=n, alpha=0.8)

ax.set_xticks(range(len(names)))
ax.set_xticklabels(names, rotation=20)
```

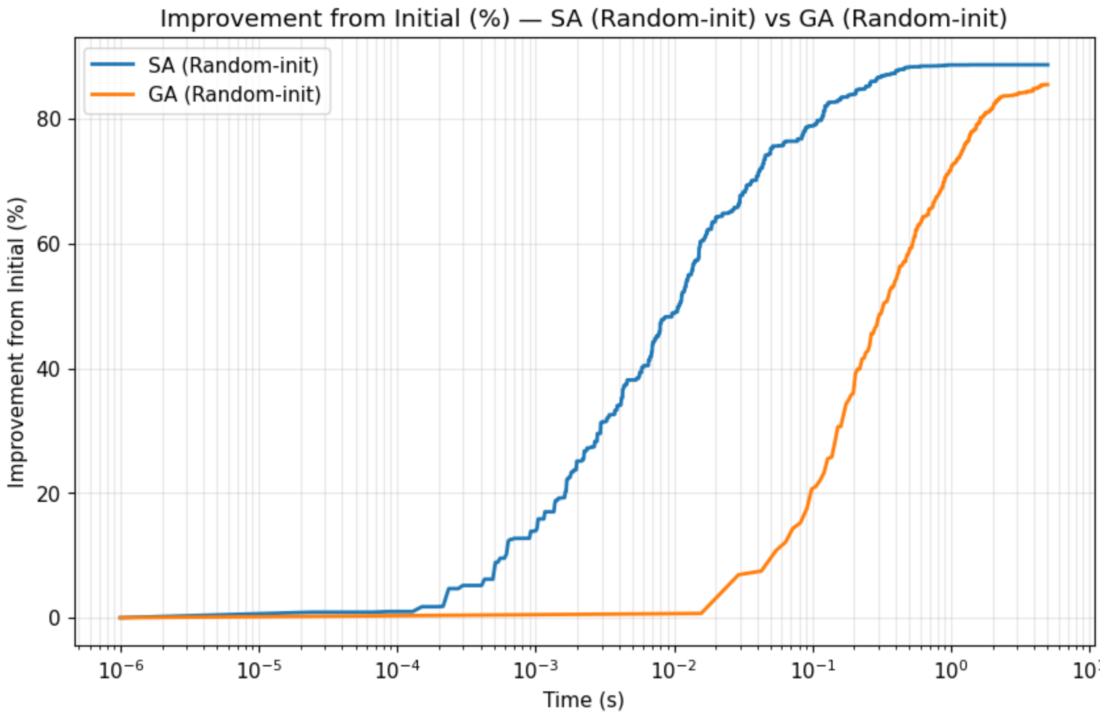
```
ax.set_ylabel('Gap to Optimal (%)')
ax.set_title('Final Cost Gap')
ax.grid(True, axis='y', alpha=0.3)

nn_gap = ((nn_cost / optimal_cost) - 1) * 100.0
opt_line = ax.axhline(y=0, color='green', linestyle=':', label='Optimal')
nn_line = ax.axhline(y=nn_gap, color='orange', linestyle='--', label='NN')
ax.legend(handles=[opt_line, nn_line])
plt.show()
```

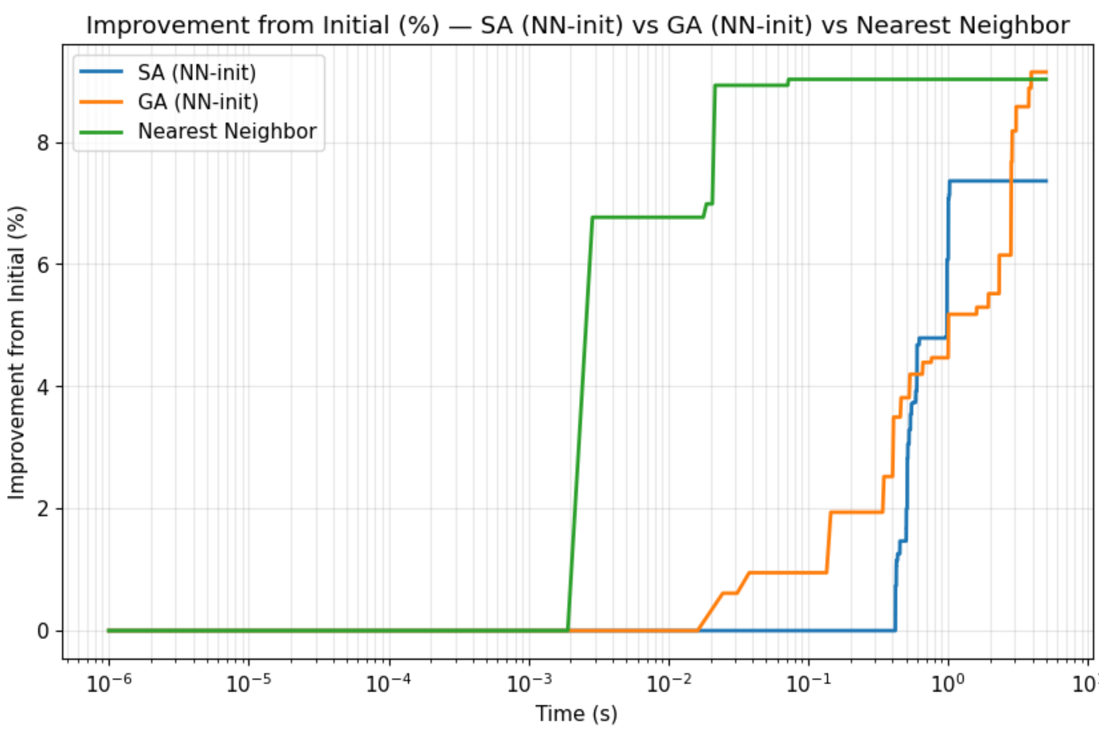


[ ]:
```
def plot_relative_improvement_over_time(names_subset, title=None):
    fig, ax = plt.subplots()
    for n in names_subset:
        if n in time_results:
            times = time_results[n]['times']
            best = time_results[n]['best']
            if len(best) > 0 and best[0] > 0:
                initial = float(best[0])
                improvement_pct = (initial - best) / initial * 100.0
                ax.plot(times, improvement_pct, label=n, linewidth=2)
    ax.set_xlabel('Time (s)')
    ax.set_ylabel('Improvement from Initial (%)')
```

```
    ax.set_title(title)
    ax.set_xscale('log')
    ax.legend()
    ax.grid(True, which='both', axis='x', alpha=0.3)
    ax.grid(True, which='major', axis='y', alpha=0.3)
    plt.show()

plot_relative_improvement_over_time(
    ["Random Solver", "Nearest Neighbor"],
    "Improvement from Initial (%) - Random vs Nearest Neighbor"
)
plot_relative_improvement_over_time(
    ["SA (Random-init)", "GA (Random-init)"],
    "Improvement from Initial (%) - SA (Random-init) vs GA (Random-init)"
)
plot_relative_improvement_over_time(
    ["SA (NN-init)", "GA (NN-init)", "Nearest Neighbor"],
    "Improvement from Initial (%) - SA (NN-init) vs GA (NN-init) vs Nearest␣
 ↪Neighbor"
)
```




Improvement from Initial (%) — SA (Random-init) vs GA (Random-init)
SA (Random-init)
GA (Random-init)
Improvement from Initial (%)
Time (s)
0
20
40
60
80
10^{-6}
10^{-5}
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
10^{1}


Improvement from Initial (%) — SA (NN-init) vs GA (NN-init) vs Nearest Neighbor
SA (NN-init)
GA (NN-init)
Nearest Neighbor
Improvement from Initial (%)
Time (s)
0
2
4
6
8
10^{-6}
10^{-5}
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
10^{1}