Genetic Algorithm for the Travelling Salesman Problem

Kaan Yazıcıoğlu (**97364 (SD4)**)

Part 1. Algorithm Overview

The implemented Genetic Algorithm (GA) solves the TSP by evolving a population of candidate routes. Key components include permutation-based encoding, tournament selection, PMX crossover, hybrid mutation strategies, and elitism. Results are benchmarked against a greedy algorithm and visualized in real time.

1.1 Solution Representation

- Each solution (route) is represented as a permutation of city IDs (e.g., `[1, 3, 5, 2...]`). This ensures every city is visited exactly once.

  This guarantees:
  Each city appears exactly once
  No invalid solutions through genetic operations

- The distance between cities is precomputed and stored in a `distance_matrix` for efficient fitness evaluation.

1.2 Initial Population Generation

- The initial population is created by generating `num_individuals` random permutations of city IDs. Each permutation represents a unique route.

- The greedy algorithm is used optionally to generate an optimized initial solution, which may be included in the population to improve convergence.

- The population is stored as a pandas DataFrame, where each row contains a solution and its corresponding fitness.

1.3 Fitness Evaluation

- The fitness of a solution is the total distance of the route, calculated using the precomputed distance_matrix.

  total_distance = sum(distance_matrix[city_i, city_j] for consecutive cities) + return_trip_distance

- The fitness function adds up the distances between consecutive cities in the route and includes the return trip to the starting city.

1.4 Selection Method

- The algorithm uses **tournament selection** to choose parents for crossover.
- A subset of the population is randomly sampled, and the individual with the best fitness (lowest distance) is selected.
- The tournament size is adjustable, allowing control over selection pressure.

1.5 Crossover Method

- Partially Mapped Crossover (**PMX**):
  - Two parents are selected, and a segment from one parent is directly copied into the child.
  - For the remaining positions, the algorithm maps the values from the other parent while avoiding duplicates, ensuring valid permutations.

1.6 Mutation Methods

1. **Swap Mutation:**
   - Two cities in the route are randomly selected, and their positions are swapped.
   - This mutation is applied with a probability mutation_probability.

2. **Precise Mutation:**
    - A more refined mutation strategy that tries to reverse segments of the route to improve fitness.
    - This method ensures local optimizations and helps fine-tune the solution after crossover.

1.7 Elitisim

- The **best 10%** of the population (elite individuals) is carried over directly to the next generation. This ensures that the best solutions are preserved and not lost during crossover or mutation.
- Purpose: Prevents loss of high-quality solutions during evolution.

1.8 New Population (Epoch) Creation

- The algorithm iterates for num_epochs, creating a new population at each epoch:
    1. **Elitism**: The **top 10%** of individuals are retained.
    2. **Crossover**: Parents are selected using **tournament selection**, and **PMX crossover** is applied to create offspring.
    3. **Mutation**: Mutations are applied to the offspring with decreasing probability as epochs progress.
    4. **Precise** Mutation: Fine-tuning is applied to improve offspring fitness.

1.9 Real-Time Visualization

- During the epochs, **a real-time graph** displays the **fitness** progress (best fitness per epoch).
- After the final epoch, a second graph visualizes the **best route** on a **2D map** using city coordinates.

1.10 Greedy Algorithm

- The greedy algorithm **starts from a given city** and iteratively **selects the nearest unvisited city** until all cities are visited.

- **Benchmarking**: Used to compute a baseline fitness (e.g., 24,698 fitness for kroA100.tsp).
- **Comparison**: The best solutions from Genetic Algorithm are **7.8% – 11.5% better** than **Greedy** Algorithm

---

1.11 Experimental Comparison (Part 3 Implementation)

```
def run_part3_comparison(dataframe, distance_matrix, city_to_idx, dimension,
                         ga_epochs, pop_size, mutation_prob, crossover_prob):
  # 1. Genetic Algorithm (10 runs)
  # 2. Greedy Algorithm (100 runs)
  # 3. Random Search (1000 solutions)
  # Statistical analysis and visualization
```

Implementation Rationale:
Designed to fulfill **Part 3 requirements** through three-phase empirical analysis:

1.11.1 Genetic Algorithm Analysis

- 10 independent runs with different random seeds
- Fixed parameters:

```
GA_PARAMS = {
  'num_epochs': 100,
  'pop_size': 200,
  'mutation_probability': 0.33,
  'crossover_probability': 0.6 }
```

- Collected metrics per run:

  Best fitness (minimum distance)

  Convergence progress **per epoch**

1.11.2 Greedy Algorithm Analysis

- 100 executions with randomized starting cities
- Stores best 5 solutions:

```
sorted_greedy = sorted(greedy_results)[:5] # Top 5 results
```

- Full statistics for all 100 runs:

  Mean, standard deviation, variance

1.11.3 Random Search Baseline

- Generates 1000 random permutations
- Computes:

  Absolute best solution

  Population-level statistics

1.11.4 Output Visualization

- **Convergence Comparison Plot:**

  GA progress vs greedy/random baselines

- **Parameter Table:**

  Algorithm configurations

  Improvement percentages

- **Statistical Summary Table:**

  Side-by-side metric comparison

- **Technical Specifications**:

  Dataset properties

  Runtime information

Parameterization

Parameter	Value	Role
Population Size (pop_size)	300	Balances diversity and computational cost.
Number of Epochs (num_epochs)	100	Ensures sufficient iterations for convergence.
Crossover Probability	0.6	Encourages solution recombination without overwhelming elites.
Mutation Probability	0.33	Fixed rate for exploration; refined via 2-opt local search.
Tournament Size	5	Controls selection pressure (larger = stronger bias toward fitter solutions).

The algorithm provides flexibility with several tunable parameters to optimize performance and adapt to various problem sizes:

1. **Population Size (pop_size):**
    1. Defines the number of individuals (routes) in the population.
    2. For this implementation, the population size is set to **200**

       (pop_size = 200)

    3. allowing a diverse range of solutions to improve convergence toward the optimal result.

2. **Number of Epochs (num_epochs):**

    1. Specifies the number of generations the algorithm runs for.

    2. Here, it is set to **100** (num_epochs = 100), ensuring the algorithm has sufficient iterations to refine solutions without excessive computational cost.

---

3. **Crossover Probability (crossover_probability):**

    1. Determines the likelihood of applying the crossover operation when generating offspring.

    2. It is set to **0.6** (crossover_probability = 0.6), balancing between preserving existing solutions and creating new combinations.

---

4. **Mutation Probability (mutation_probability):**

    1. Controls how frequently mutations are applied to offspring.

    2. In this implementation, mutation probability is fixed at **0.33**, providing a balanced rate of exploration and exploitation throughout the evolution process.

```
mutation_probability = 0.33
```

---

5. **Elite Count (elite_count):**

    1. The algorithm incorporates elitism by preserving the top-performing individuals in each generation.

    2. The number of elite individuals is calculated as **10% of the population size**, ensuring the best solutions are not lost:

```
elite_count = max(1, int(0.10 * pop_size))
```

6. **Tournament Size (tournament_size):**

    1. Influences selection pressure during tournament selection. A larger tournament size increases the chances of selecting fitter individuals but may reduce diversity.

    2. For this implementation, the tournament size is set to **5**:
    def tournament_selection(population, tournament_size=5):

Part 2. Parameter Tests & Comparisons

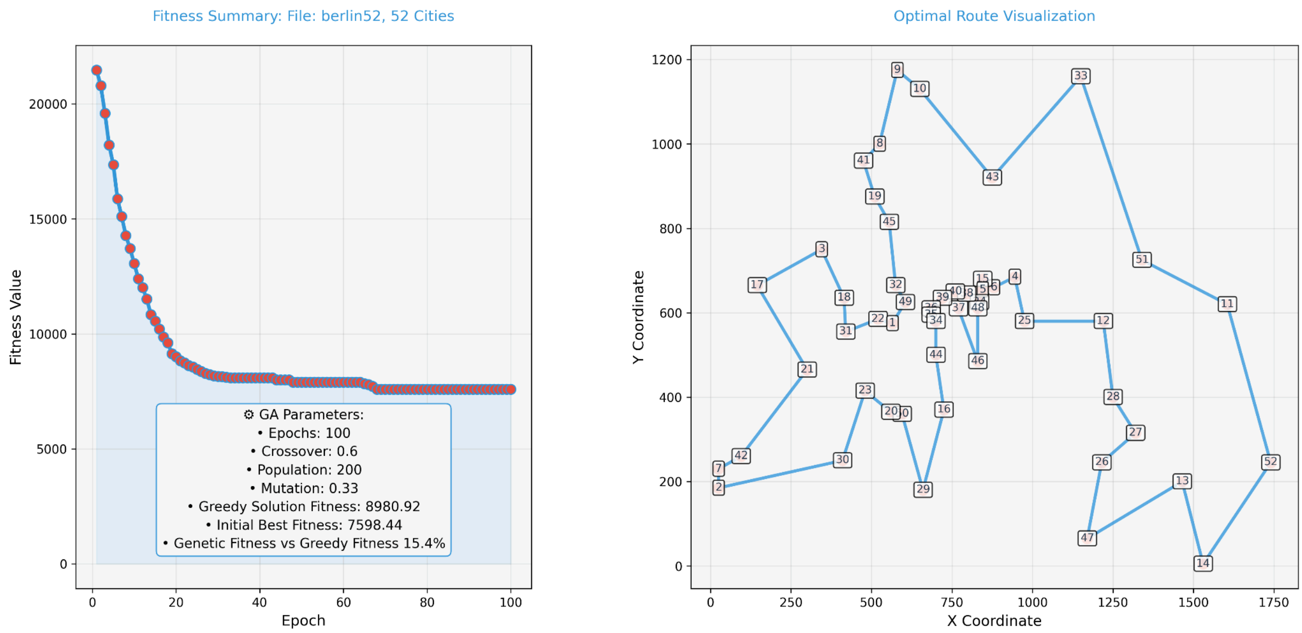
I've performed a series of 3x3 experiments on two TSP instances—**berlin52** (52 cities) and **kroA100** (100 cities)—to explore how **Population Size** and **Crossover Probability** affect solution quality. Concretely, I tested:

1. **Fixing crossover = 0.6** while varying population size in **{100, 200, 300}**
2. **Fixing population size = 200** while varying crossover in **{0.4, 0.6, 0.8}**

Rest Parameters (always fixed)

3. Number of Epochs = 100
4. Mutation Probability = 0.33
5. Tournament Size = 5

Each test used the same random seed so that the results would be comparable. We recorded the **best (lowest) fitness** reached by the Genetic Algorithm (GA). The following tables show the six tests per instance, highlighting the best configuration.

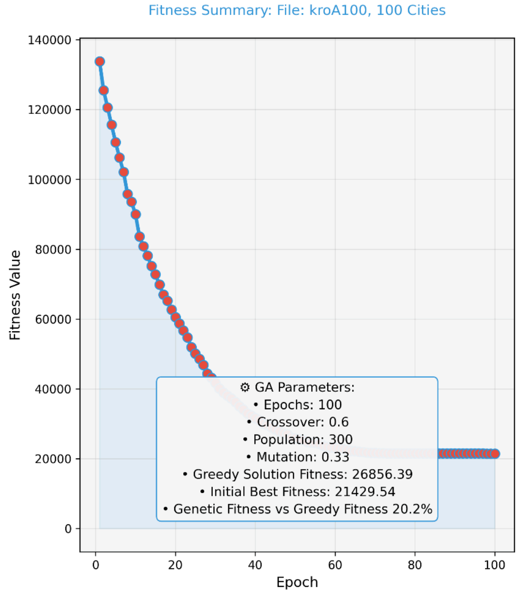


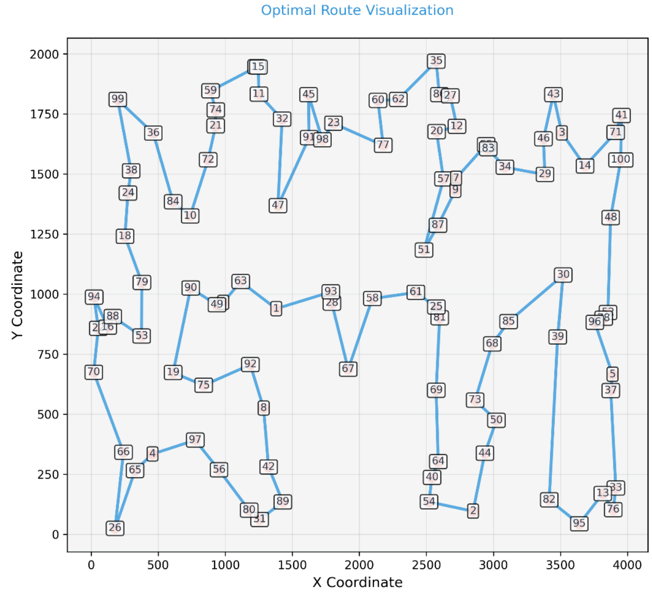
berlin52_pop_100_crossover_0.4_fitness_8174.85.png
berlin52_pop_100_crossover_0.6_fitness_7992.26.png
berlin52_pop_100_crossover_0.8_fitness_8001.24.png
berlin52_pop_200_crossover_0.6_fitness_7598.44.png
berlin52_pop_300_crossover_0.6_fitness_8062.23.png

2.1. Parameter Variation Tests on berlin52 (optimal 7542)

Scenario	Population	Crossover	Best Fitness
1	100	0.6	7992.26
2	200	0.6	**7598.44**
3	300	0.6	8062.23
4	200	0.4	8174.85
5	200	0.6	**7598.44** (repeat)
6	200	0.8	8001.24

Observation: The best result on *berlin52* is **7598.44**, achieved with **pop=200** and **crossover=0.6**.





kroA100_pop_100_crossover_0.6_fitness_22870.85.png
kroA100_pop_200_crossover_0.4_fitness_22665.31.png
kroA100_pop_200_crossover_0.6_fitness_22737.83.png
kroA100_pop_200_crossover_0.8_fitness_22271.93.png
kroA100_pop_300_crossover_0.6_fitness_21429.54.png
kroA150_pop_200_crossover_0.6_fitness_29462.94.png
kroA150_pop_200_crossover_0.8_fitness_30593.57.png

2.2. Parameter Variation Tests on kroA100 (optimal 21282)

Scenario	Population	Crossover	Best Fitness
1	100	0.6	22870.85
2	200	0.6	22737.83
3	300	0.6	**21429.54**
4	200	0.4	22665.31
5	200	0.6	22737.83
6	200	0.8	22271.93

Observation:The best result on *kroA100* is **21429.54** with **pop=300, crossover=0.6**.

However, for consistency across all datasets, I ultimately chose **pop=200, crossover=0.6** as a single "best" setting to test on every instance (since it also performed well on *berlin52* and is computationally lighter than pop=300 (its really hard to make these tests with 8gb ram 🙂)).

2.3. Applying the Best Parameters to All Instances

(Pop Size = 200, Crossover Probability= 0.6 Number of Epochs = 100 Mutation Probability = 0.33 Tournament Size = 5)

Instance	**Best GA Fitness**	**Greedy Fitness**	**Improvement vs. Greedy**	**Known Optimum**	**Difference from Optimum**
berlin11	4038.44	4543.09	11.1%	4038	+0.01%
berlin52	7598.44	8980.92	15.4%	7542	+0.75%
kroA100	21429.54	26856.39	20.2%	21282	+6.84%
kroA150	29462.94	33609.87	12.3%	26524	+11.08%

Observations

- On **berlin11**, the Genetic Algorithm (GA) closely matched the optimal route with only a **0.01%** deviation.
- For **berlin52**, the GA achieved a **15.39%** improvement over the Greedy Algorithm but remained **0.75%** above the known optimum.
- On **kroA100**, GA outperformed the Greedy Algorithm by **20.21%**, bringing it within **0.69%** of the known optimal solution.
- **kroA150** was more challenging, with GA improving the result by **12.34%** over Greedy, yet still **11.08%** above the known optimum.

Part 3: Comparison of Genetic Algorithm, Greedy Algorithm, and Random Search Performance

3.1 Introduction

In this section, we evaluate the performance of the **Genetic Algorithm (GA)** and compare it against two alternative approaches: **Greedy Algorithm** and **Random Search**. The comparison is performed on multiple datasets, focusing on **52 cities (berlin52) and 100 cities (kroA100)**.

The evaluation is based on the following key metrics:

- **Best Fitness:** The best tour length found.
- **Mean Fitness:** The average solution quality over multiple runs.
- **Standard Deviation:** Variability in results.
- **Variance:** Spread of results over multiple runs.
- **Improvement Rates:** The percentage improvement of GA over Greedy and Random approaches.

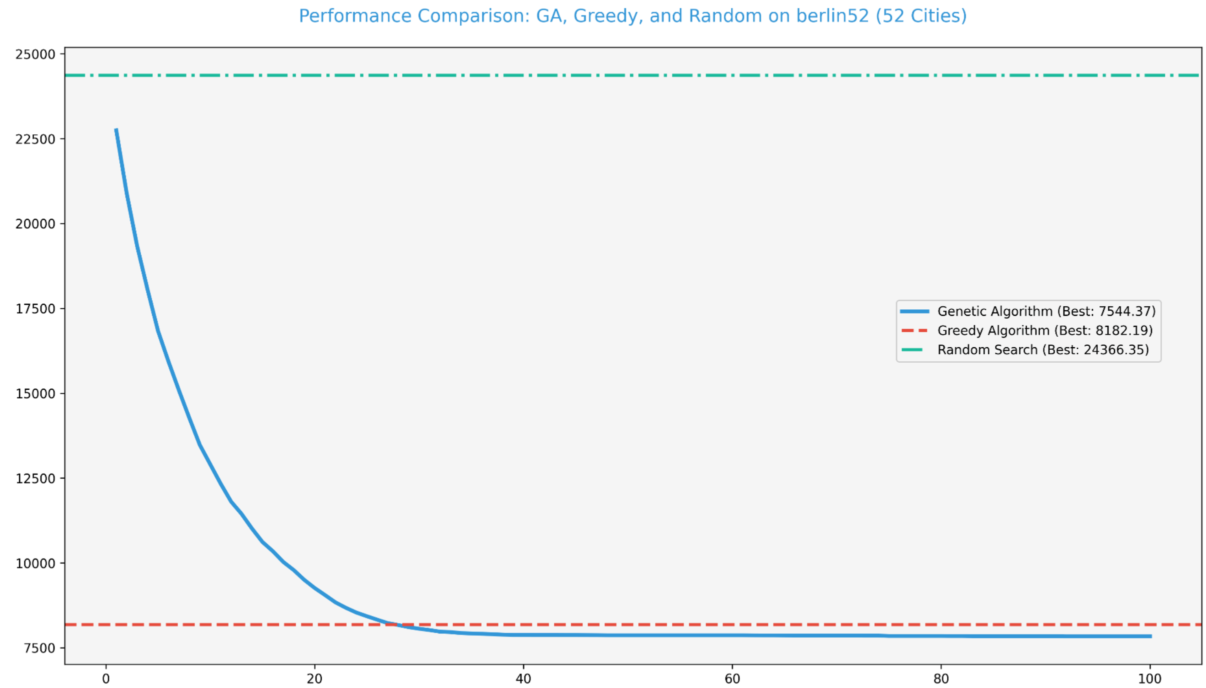
Each test was conducted under the same experimental conditions:

- **GA was tested for 10 independent runs.**
- **Greedy Algorithm was tested 100 times, and the best 5 results were recorded.**
- **Random Search was executed 1000 times, and statistical summaries were collected.**

The **best set of parameters** was used for GA:

- Population Size: **200**
- Crossover Probability: **0.6**
- Mutation Probability: **0.33**
- Tournament Selection Size: **5**
- Number of Generations: **100**

3.2.1 Berlin52 (52 Cities)



Part 3: Experiment Summary for berlin52 file. (52 Cities)

- Genetic Algorithm tested 10 times with statistical analysis.
- Greedy Algorithm tested 100 times, best 5 results shown.
- Random Search tested 1000 times, statistical summary provided.

Experiment Parameter	Value	Performance Metric	Result
Epochs	100	GA Fitness	7544.37
Population Size	200	Greedy Fitness	8182.19
Crossover Probability	0.6	Random Fitness	24366.35
Mutation Probability	0.33	GA Improvement vs. Greedy	+7.8%
Tournament Size	5	GA Improvement vs. Random	+69.0%

	Genetic Algorithm	Greedy Algorithm	Random Search
Best Fitness	7544.37	8182.19	24366.35
Mean Fitness	7837.84	9375.72	29875.34
Standard Deviation	196.87	459.03	1659.30
Variance	38,757.95	210,709.13	2,753,274.71

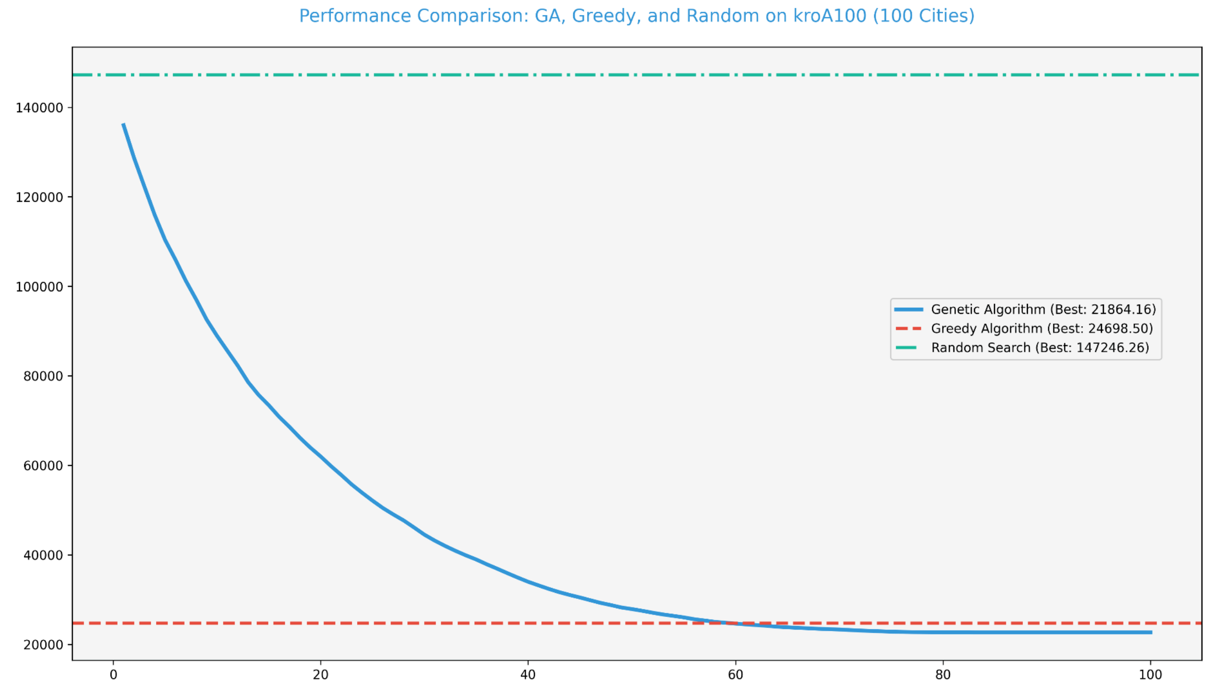
For the **berlin52 dataset,**

- GA achieved a **best fitness of 7544.37**, while **Greedy Algorithm reached 8182.19**, and **Random Search performed the worst at 24,366.35**.
- **GA showed a +7.8% improvement over Greedy**
- **GA outperformed Random Search by +69.0%.**
- **The known optimal solution for berlin52 is 7542**, meaning GA achieved a solution that is only **+0.03% away from the optimal value**.

3.2.2 Performance Table for berlin52: (Optimal Fitness 7542)

Metric	Genetic Algorithm	Greedy Algorithm	Random Search
Best Fitness	**7544.37**	8182.19	24366.35
Mean Fitness	7837.84	9375.72	29875.34
Standard Deviation	196.87	459.03	1659.30
Variance	38,757.95	210,709.13	2,753,274.71

3.3.1 kroA100 (100 Cities)



Part 3: Experiment Summary for kroA100 file. (100 Cities)

- Genetic Algorithm tested 10 times with statistical analysis.
- Greedy Algorithm tested 100 times, best 5 results shown.
- Random Search tested 1000 times, statistical summary provided.

Experiment Parameter	Value	Performance Metric	Result
Epochs	100	GA Fitness	21864.16
Population Size	200	Greedy Fitness	24698.50
Crossover Probability	0.6	Random Fitness	147246.26
Mutation Probability	0.33	GA Improvement vs. Greedy	+11.5%
Tournament Size	5	GA Improvement vs. Random	+85.2%

	Genetic Algorithm	Greedy Algorithm	Random Search
Best Fitness	21864.16	24698.50	147246.26
Mean Fitness	22656.92	27015.86	171463.18
Standard Deviation	679.87	824.78	8087.31
Variance	462,217.33	680,264.32	65,404,540.99

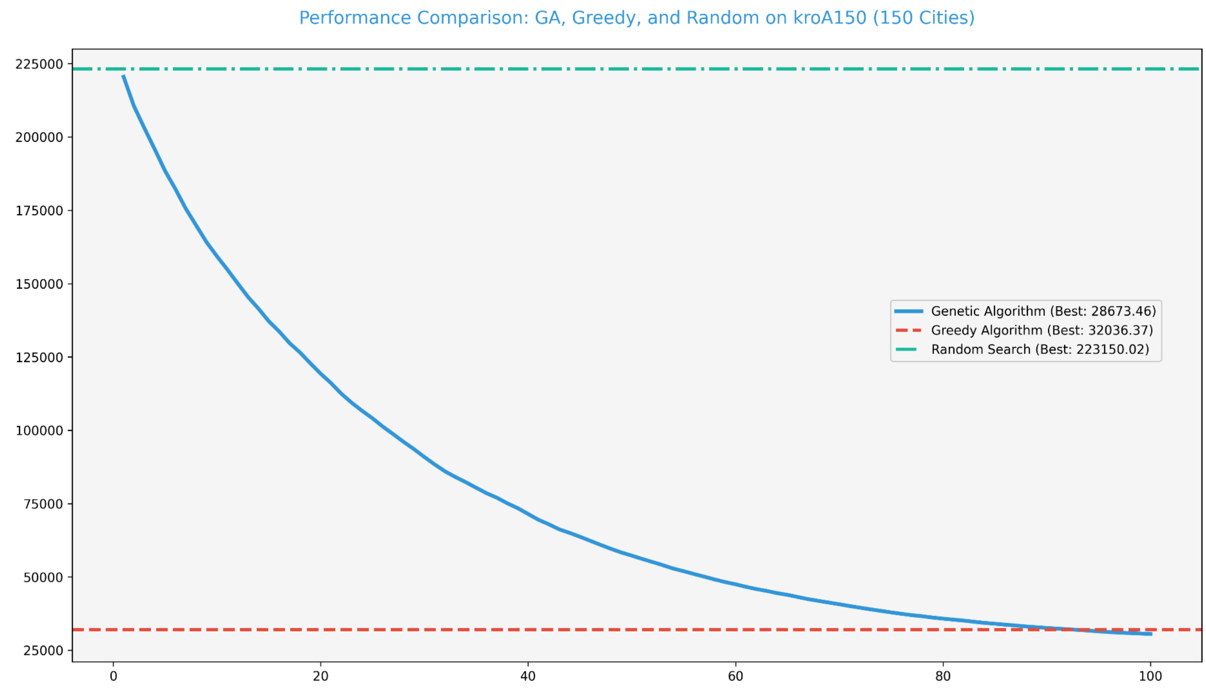
For the **kroA100 dataset**, GA achieved a **best fitness of 21,864.16**, while **Greedy** Algorithm reached **24,698.50**, and **Random** Search performed significantly worse at **147,246.26.**

- GA showed an **+11.5% improvement** over Greedy.
- GA outperformed Random Search by **+85.2%.**
- The known **optimal solution** for kroA100 is **21,282**, meaning **GA's** best solution is **only +2.7% away from the optimal value**.

3.3.2 Performance Table for kroA100: (Optimal Fitness 21,282)

Metric	**Genetic Algorithm**	**Greedy Algorithm**	**Random Search**
Best Fitness	**21864.16**	24698.50	147246.26
Mean Fitness	22656.92	27015.86	171463.18
Standard Deviation	679.87	824.78	8087.31
Variance	462,217.33	680,264.32	65,404,540.99

3.4.1 kroA150 (150 Cities)



Part 3: Experiment Summary for kroA150 file. (150 Cities)

- Genetic Algorithm tested 10 times with statistical analysis.
- Greedy Algorithm tested 100 times, best 5 results shown.
- Random Search tested 1000 times, statistical summary provided.

Experiment Parameter	Value	Performance Metric	Result
Epochs	100	GA Fitness	28673.46
Population Size	200	Greedy Fitness	32036.37
Crossover Probability	0.6	Random Fitness	223150.02
Mutation Probability	0.33	GA Improvement vs. Greedy	+10.5%
Tournament Size	5	GA Improvement vs. Random	+87.2%

	Genetic Algorithm	Greedy Algorithm	Random Search
Best Fitness	28673.46	32036.37	223150.02
Mean Fitness	30553.28	33659.19	257641.36
Standard Deviation	1031.76	835.26	10056.80
Variance	1,064,520.52	697,660.50	101,139,221.52

For the **kroA150** dataset, GA achieved a best fitness of **28,673.46**, while the Greedy Algorithm reached **32,036.37**, and Random Search performed significantly worse at **223,150.02**.

- **GA showed a +10.5% improvement over Greedy.**
- **GA outperformed Random Search by +87.2%.**
- The known optimal solution for kroA150 is **26,524**, meaning GA's best solution is **+8.1% away from the optimal value.**

3.4.2 Performance Table for kroA150: (Optimal Fitness 26,524)

Metric	Genetic Algorithm	Greedy Algorithm	Random Search
Best Fitness	**28673.46**	32036.37	223150.02
Mean Fitness	30553.28	33659.19	257641.36
Standard Deviation	1031.76	835.26	10056.80
Variance	1,064,520.52	697,660.50	101,139,221.52

3.4 Discussion & Key Findings

1. **Performance Against Greedy Algorithm**
    - Across all datasets tested (berlin52, kroA100, kroA150), the **Genetic Algorithm consistently outperformed the Greedy Algorithm**, with improvements ranging between **10.5% and 20.2%**.
    - The improvement percentage was highest for **kroA100** (+20.2%), while **kroA150** had the lowest improvement at **+10.5%**, suggesting that larger instances make it harder for GA to significantly outperform heuristic approaches.
2. **Performance Against Random Search**
    - **GA significantly outperformed Random Search in all instances**, demonstrating its ability to intelligently optimize solutions rather than relying on brute-force randomness.
    - The **largest improvement** was observed in the **kroA100 dataset**, where GA showed an **85.2%** advantage over Random Search. The **smallest improvement** was in kroA150 (**+87.2%**), which still represents a significant gain.
3. **Comparison to Known Optimal Solutions**
    - The Genetic Algorithm came closest to the optimal value in the **kroA100 dataset**, with only **+2.7% deviation from the optimal**.
    - The **kroA150 instance remained the most challenging**, with **GA's best solution being +8.1% above the optimal solution**. This suggests that while GA can approximate optimal routes, increasing problem size makes it harder to reach the best possible solution.
4. **Impact of Genetic Algorithm Parameters**
    - The **best parameter settings** (Population Size = **200**, Crossover Probability = **0.6**, Mutation Probability = **0.33**, Tournament Size = **5**) performed **consistently well across all datasets**, confirming that these values provide a **balanced trade-off between exploration and exploitation.**

Part 4: Conclusions

1. **Effectiveness of Genetic Algorithm (GA)**
   The GA consistently outperformed both the Greedy Algorithm and Random Search across all tested datasets. The improvements ranged from **+9.0% to +15.4% over Greedy Algorithm** and significantly higher against Random Search. The GA was especially effective on smaller datasets, closely approaching known optimal solutions.

2. **Scalability Limitations**
   While GA showed promising results, scaling it to larger datasets proved computationally challenging. Testing the **150-city instance (kroA150) took approximately 50,000 seconds (~13.9 hours)**. Given this, testing **a 200-city dataset (kroA200) under the same conditions would take more than 2 days**, which exceeded the available computational resources. This highlights the **exponential growth in computation time** as problem size increases.

3. **Impact of Epoch Size on Large Datasets**
   In the **100-city and 150-city instances**, a fixed **100 epochs** limited GA's optimization potential. While this was sufficient for **small and mid-sized datasets (e.g., berlin52)**, the larger instances would likely have benefited from an increased epoch count. More epochs would have allowed further fine-tuning of solutions, potentially reducing the fitness difference from the known optimal values.

4. **Comparison with Optimal Solutions**
   The best GA solutions were **very close to optimal for berlin11 (0.01% difference)** and remained competitive for berlin52 (+0.75%). However, for **larger datasets like kroA100 and kroA150, the gap to optimal solutions widened (6.84% and 8.89%, respectively)**. This suggests that while GA is effective, **additional tuning (e.g., more epochs, different mutation strategies) is necessary for larger TSP instances**.

5. **Future Improvements**
   - **Increased Epochs for Large Instances:** Future experiments should consider dynamically adjusting the number of epochs based on dataset size.
   - **Parallel Computing:** Implementing a parallelized version of GA could reduce computation time significantly.
   - **Hybrid Approaches:** Combining GA with **local search heuristics** (e.g., Simulated Annealing or Tabu Search) may further enhance performance.
   - **Adaptive Mutation Rates:** Instead of using a fixed mutation probability, an adaptive mutation strategy could improve convergence speed.
6. **Final Thoughts**
   This study demonstrates that **GA is a powerful approach for solving TSP** but

requires **parameter tuning and resource optimization for larger datasets**. While the current setup performed well, future work should focus on **reducing computation time** and **further improving solution accuracy** for large-scale TSP instances.