Genetic Algorithm Approach to the Travelling Salesperson Problem

Ignat Bojinov
StFX
2025fve@stfx.ca

I. Introduction

A. What is the problem?

The Traveling Salesman Problem (TSP) is a classical NP-hard problem in combinatorial optimization, important in theoretical computer science and operations research. It asks a deceptively simple question:

"Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?"

B. Small review

Over the decades, various methods have been developed for solving TSP. Exact approaches include branch-and-bound and dynamic programming, but these scale poorly. Approximation and heuristic methods such as nearest neighbor and 2-opt are widely used for smaller problems. Metaheuristics such as Genetic Algorithms (GA), Simulated Annealing, and Ant Colony Optimization have shown good performance on medium to large instances. In particular, GAs are often chosen because of their flexibility and ability to balance exploration and exploitation of the search space [1].

II. Problem Description

A. What is TSP?

Formally, the TSP can be expressed as finding a Hamiltonian cycle of minimum length in a weighted complete graph where nodes represent cities and edge weights represent distances. In this assignment, distances are Euclidean, calculated from provided 2D coordinates in JSON-formatted datasets.

III. Algorithm Description

A. How was the GA implemented?

The GA implementation follows the standard framework but is adapted for TSP in Python.

a) Initialization: The population is generated as random permutations of city IDs:

```python
def initial_population(pop_size, cities):
    city_ids = list(cities.keys())
    population = []
    for _ in range(pop_size):
        route = city_ids[:]
        random.shuffle(route)
        population.append(route)
    return population
```

b) Fitness function: Fitness is defined as the total Euclidean distance of the route (shorter routes are better):

```python
def route_dist(route, cities):
    dist = 0
    for i in range(len(route)):
        city_a = cities[route[i]]
        city_b = cities[route[(i + 1) % len(route)]]
        dist += euclid_dist(city_a, city_b)
    return dist
```

c) Selection: Tournament selection (`sorry_loosers`) picks the best out of a random subset:

```python
def sorry_loosers(population, cities, k=5):
    fight = random.sample(population, k)
    fight.sort(key=lambda route: route_dist(route,
        cities))
    return fight[0]
```

d) Crossover: Order crossover preserves a slice from one parent and fills the rest from the other:

```python
def crossover(daddy, mommy):
    size = len(daddy)
    start, end = sorted(random.sample(range(size),
        2))
    child = [None] * size
    child[start:end] = daddy[start:end]
    mommy_el = [c for c in mommy if c not in child]
    pos = 0
    for i in range(size):
        if child[i] is None:
            child[i] = mommy_el[pos]
            pos += 1
    return child
```

e) Mutation: Swap mutation randomly exchanges two cities with probability equal to the mutation rate:

```python
def ninja_turtles(route, mutation_rate=0.01):
    route = route[:]
    for i in range(len(route)):
        if random.random() < mutation_rate:
            j = random.randint(0, len(route)-1)
            route[i], route[j] = route[j], route[i]
    return route
```

f) Next generation with elitism: The best solution is preserved, and the rest are produced by selection, crossover, and mutation:

```python
def bright_future(population, cities, elite_size=1,
    mutation_rate=0.01):
    population.sort(key=lambda r: route_dist(r,
        cities))
    natural_selection = population[:elite_size]
    while len(natural_selection) < len(population):
        daddy = sorry_loosers(population, cities)
        mommy = sorry_loosers(population, cities)
        child = crossover(daddy, mommy)
        child = ninja_turtles(child, mutation_rate)
        natural_selection.append(child)
    return natural_selection
```

B. *Enhancements and modifications*

Several modifications were included in this implementation:

- **Elitism:** Ensures best individuals survive to the next generation.
- **Tournament selection:** Stronger parent competition than random selection.
- **Parameter variety:** Different population sizes, mutation rates, and generations tested.
- **Visualization:** Added convergence plots and final-route diagrams for interpretability.

IV. Analysis Plan

The GA was compared against:

- **Baseline:** Greedy nearest neighbor heuristic.
- **Instances:** berlin52 (small), a280 (medium), pcb442 (large).
- **Configurations:** Two mutation rates (0.01, 0.05).
- **Statistics:** 10 runs each → mean, standard deviation, worst/best distances.

V. Results and Discussion

A. *Numerical Results*

Table I summarizes the GA performance against nearest neighbor.

TABLE I
Comparison of GA vs. Nearest Neighbor

Instance	Method	Best Dist	Mean Dist	Std Dev
berlin52	NN	8980.9	-	-
berlin52	GA (0.01)	**8648.0**	9382.7	457.7
berlin52	GA (0.05)	12815.0	13596.5	643.3
a280	NN	3148.1	-	-
a280	GA (0.01)	**19476.3**	20100.6	411.4
a280	GA (0.05)	25671.3	26283.8	364.2
pcb442	NN	61984.0	-	-
pcb442	GA (0.01)	**517358.0**	529792.4	7857.7
pcb442	GA (0.05)	629252.3	642718.7	7137.2

B. *Interpretation*

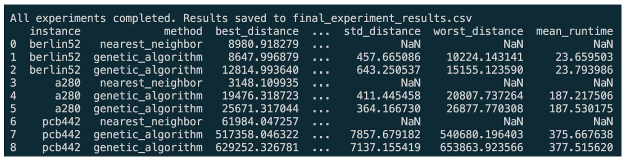
Several trends emerged:

- For **berlin52**, the GA with low mutation rate outperformed nearest neighbor, achieving shorter routes (8648 vs. 8981). However, higher mutation degraded results.
- For **a280**, GA produced routes much longer than nearest neighbor (19k–26k vs. 3.1k). This indicates the GA struggled to scale.
- For **pcb442**, the GA completely failed to compete (517k–629k vs. 62k). The algorithm likely got trapped in poor local minima and could not exploit the search space effectively.
- Runtime scaled significantly: ∼24s for berlin52, ∼187s for a280, and ∼376s for pcb442.

C. *Statistical Observations*

The GA displayed moderate variability (std. dev. 364–7857), meaning results are not deterministic but consistent within ranges. Mutation rate 0.01 consistently outperformed 0.05, highlighting the risk of excessive exploration.

D. *Visual Results*



```
All experiments completed. Results saved to final_experiment_results.csv
   instance             method  best_distance  ...  std_distance  worst_distance  mean_runtime
0  berlin52   nearest_neighbor    8980.918279  ...           NaN             NaN           NaN
1  berlin52  genetic_algorithm    8647.996879  ...    457.665086    10224.143141     23.659503
2  berlin52  genetic_algorithm   12814.993640  ...    643.250537    15155.123590     23.793986
3      a280   nearest_neighbor    3148.109935  ...           NaN             NaN           NaN
4      a280  genetic_algorithm   19476.318723  ...    411.445458    20807.737264    187.217506
5      a280  genetic_algorithm   25671.317044  ...    364.166730    26877.770308    187.530175
6    pcb442   nearest_neighbor   61984.047257  ...           NaN             NaN           NaN
7    pcb442  genetic_algorithm  517358.046322  ...   7857.679182   540680.196403    375.667638
8    pcb442  genetic_algorithm  629252.326781  ...   7137.155419   653863.923566    377.515620
```

Fig. 1. Convergence of GA across datasets.

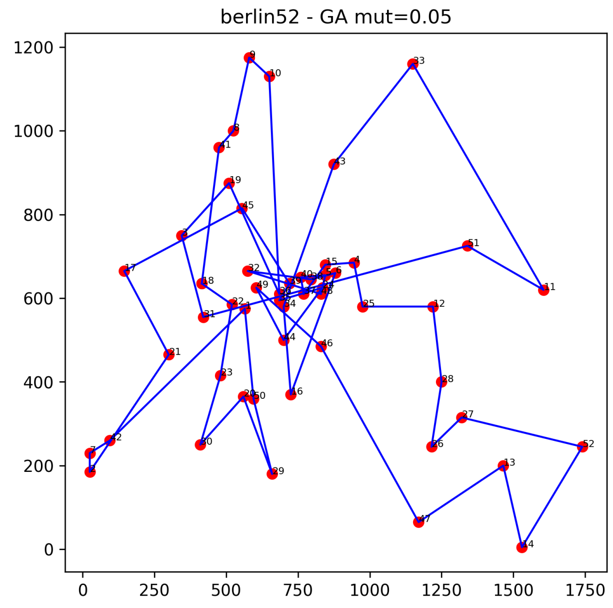


Fig. 2. Best GA route for berlin52 instance.

VI. Conclusions and Future Work

A. *Major takeaways*

- GA achieved modest success on small instances but performed poorly on medium and large ones.
- Mutation rate was critical: lower mutation (0.01) consistently outperformed higher mutation (0.05).
- Nearest neighbor surprisingly outperformed GA for larger instances.

B. *Future directions*

- Integrating local search (e.g., 2-opt or 3-opt) as a hybrid GA could drastically improve results.
- Adaptive mutation rates could balance exploration/exploitation dynamically.
- Parallel implementations would reduce runtimes for larger datasets.
- Testing against additional heuristics (Ant Colony, Lin–Kernighan) would provide broader benchmarks.

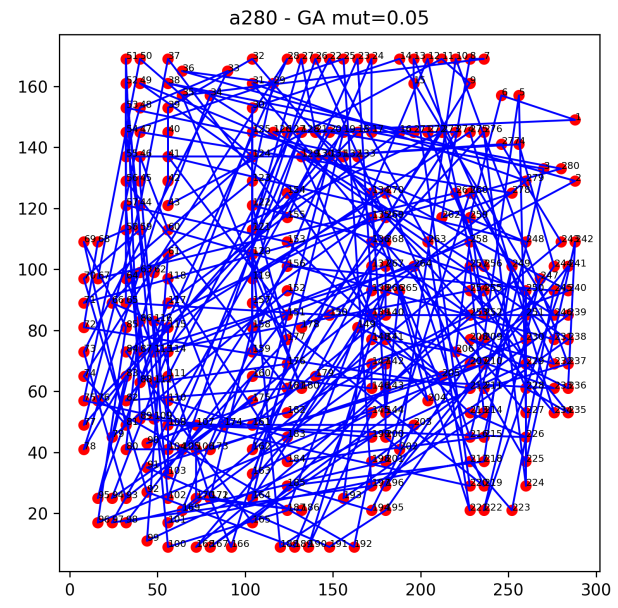


Fig. 3. Best GA route for a280 instance.

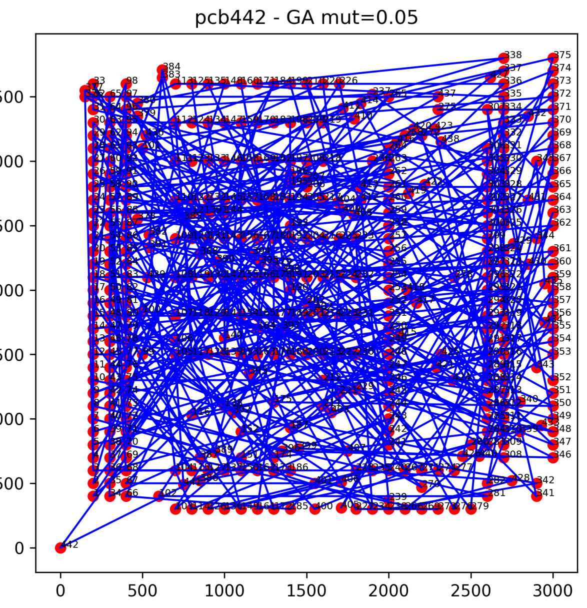


Fig. 4. Best GA route for pcb442 instance.

References

[1] Wikipedia contributors, "Travelling salesman problem," *Wikipedia*, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Travelling$_{salesman_{p}roblem}$