Course Labs Log

Sina Behnam Sharabafan
student ID : 322796
email : s322796@studenti.polito.it

Winter 2024

Abstract

This document consists of two concatenated parts. The first part presents a labs offline log that provides an overview of laboratory activities completed throughout the course, including implementation details and peer review feedback. The second part, appended to this PDF, contains the final project of the course.

Contents

1 Lab 0: Git and GitHub Basics

1.1 Overview

Introduction to version control using Git and GitHub. The lab focused on learning basic Git commands and collaborative workflows through peer reviews.

1.2 Tasks Completed

- Created a new GitHub repository
- Created and added `joke.md` file with content
- Committed and pushed changes to remote repository
- Learned fundamental Git commands (add, commit, push)

1.3 Peer Reviews

- **Review 1**: Opened issue on peer's repository (sina-behnam, October 9, 2025). Feedback provided: "The joke was simple and funny, nice."
- **Review 2**: Completed review for second peer. Note: Issue is not trackable/findable in history.

2 Lab 1: Local Search for Knapsack Problem

2.1 Overview

Implementation of local search algorithms to solve the multidimensional knapsack problem. The lab focused on applying hill climbing and evolutionary strategies to find optimal or near-optimal solutions.

2.2 Tasks Completed

- Implemented hill climbing algorithm with local search
- Applied evolution strategy for knapsack optimization
- Developed neighbor generation and fitness evaluation functions
- Tested algorithm on multiple knapsack problem instances

2.3 Peer Reviews

- **Review 1** (October 24, 2025): Provided feedback on random improvement strategy and local optima handling. Suggested that introducing random improvement after no progress maintains the same tweaking/climbing strategy and may fail to escape local optima. Recommended broader exploration mechanisms similar to Simulated Annealing. Also suggested printing the last obtained value after fixed iterations for problems 2 and 3 instead of searching for exact optimal solution.
- **Review 2** (October 24, 2025): Acknowledged correct implementation following evolution strategy and hill climbing with reasonable results. Suggested exploring other optimization methodologies (Tabu search, Simulated Annealing) for potentially better results. Recommended adding stopping conditions based on item assignment completion. Suggested checking if `neighbor_fitness == current_fitness` to enable exploration. Questioned the necessity of two-layer solution comparison logic when only checking `best_solution`.

3 Lab 2: Evolutionary Algorithm for TSP Problems

3.1 Overview

Implementation of Evolutionary Algorithm (EA) to solve diverse types of Traveling Salesman Problems (TSP), including symmetric, asymmetric, and problems with negative edge weights. The lab explored various genetic operators and selection strategies.

3.2 Problem Types

- **G-type**: Symmetric positive TSP instances with triangle inequality property
- **R1-type**: Asymmetric TSP instances with different directional distances
- **R2-type**: TSP instances with negative edge weights

3.3 Tasks Completed

- Implemented complete EA framework with population initialization
- Developed Tournament and Roulette Wheel selection methods
- Created custom mutation and crossover operators for TSP
- Implemented elitist evaluation strategy combining current and offspring populations
- Applied algorithm to multiple problem instances (G-types, R1-types, R2-types)
- Achieved improvements on all tested graph conditions
- Implemented Hill Climbing for comparison purposes

3.4 Implementation Structure

- `tsp.py`: Problem structure and testing functions
- `ea.py`: Main EA components (operators, selection, EA loop)
- `G_types.ipynb`: Symmetric TSP experiments
- `R_types.ipynb`: Asymmetric and negative-weight TSP experiments
- `hill_climbing.py`: Baseline comparison implementation

3.5 Results

Successfully achieved improvements on all problem instances across different graph types (symmetric, asymmetric, negative weights). Due to the extensive amount of experiments and results, they are not included in this report. All implementations and notebooks with saved experiments and their results are available in the repository.

3.6 Peer Reviews

No peer reviews were conducted for this lab.

4 Lab 3: Shortest Path Algorithms

4.1 Overview

Implementation and comparison of various shortest path algorithms from scratch. The lab focused on graph problem generation and implementing classic pathfinding algorithms with validation against NetworkX library implementations.

4.2 Tasks Completed

- Generated diverse graph problems for testing
- Implemented Dijkstra's algorithm from scratch
- Implemented A* search algorithm from scratch
- Implemented Bellman-Ford algorithm from scratch
- Validated implementations against NetworkX library for correctness
- Compared performance and efficiency of different algorithms

4.3 Results

All implementations were successfully validated against NetworkX library, ensuring correctness and efficiency. Detailed experiments and results are available in the repository notebook.

4.4 Peer Reviews

No peer reviews were conducted for this lab.

Solving the Weighted Traveling Collector Problem with an Evolutionary Algorithm

Computation Intelligence Course Project Work Report

Sina Behnam Sharbafan
student ID : 322796
email : s322796@studenti.polito.it

February 9, 2026

Abstract

We address the Weighted Traveling Collector Problem (WTCP), a variant of the classical Traveling Salesman Problem in which a collector must gather gold from cities and deliver it to a central depot, subject to a nonlinear cost function that penalizes carrying weight over distance. The cost function $C(P, w) = d_P + (\alpha \cdot d_P \cdot w)^\beta$ introduces a parameter β that fundamentally changes the structure of optimal solutions: when $\beta > 1$, the super-linear growth of the weight penalty makes it dramatically cheaper to split deliveries into many light trips, whereas for $\beta = 1$ the penalty is linear and the baseline single-city strategy is already near-optimal. We design an Evolutionary Algorithm (EA) with β-regime-specific initialization, mutation operators, and crossover strategies. Our approach achieves approximately 97.9% improvement over baseline when $\beta = 2$, while also producing modest gains for $\beta = 1$.

1 Problem Overview

1.1 Problem Definition

We are given a connected graph $G = (V, E)$ where $V = \{0, 1, \ldots, n-1\}$ represents cities. City 0 serves as the *depot*—the starting and ending point for all trips. Every non-depot city $i \in V \setminus \{0\}$ contains a gold amount $g_i \in [1, 1000]$ kg that must eventually be collected and delivered to the depot. Each edge $(i, j) \in E$ carries a Euclidean distance weight d_{ij} computed from the 2D coordinates of the cities. The graph is randomly generated with a given density parameter controlling edge probability, while ensuring connectivity.

1.2 Cost Function

The cost of traversing a path P while carrying weight w is defined as:

$$C(P, w) \;=\; d_P \;+\; \left(\alpha \cdot d_P \cdot w\right)^\beta, \tag{1}$$

where $d_P = \sum_{(i,j)\in P} d_{ij}$ is the total Euclidean length of path P, and $\alpha, \beta > 0$ are parameters controlling the severity of the weight penalty.

The first term d_P represents a base travel cost that is incurred regardless of the load. The second term $(\alpha \cdot d_P \cdot w)^\beta$ is the *weight penalty*, which depends on both the distance traveled and the weight carried. The interplay between these two terms, and in particular the role of β, is central to understanding the problem.

1.3 The Role of β

The exponent β fundamentally determines the structure of optimal solutions:

- **When** $\beta = 1$: The cost becomes $C(P, w) = d_P + \alpha \cdot d_P \cdot w = d_P(1 + \alpha w)$. The weight penalty grows *linearly* with both distance and weight. Consolidating gold from multiple cities into a single trip increases weight but saves on distance by avoiding repeated traversals. The marginal cost of adding more weight to an already-loaded trip is modest, **making longer multi-city tours with heavier loads a reasonable strategy.**
- **When** $\beta > 1$: The penalty term grows *super-linearly* with the product $\alpha \cdot d_P \cdot w$. For instance, with $\beta = 2$, doubling the carried weight quadruples the penalty. This creates an enormous incentive to minimize the weight carried on any individual trip. The optimal strategy shifts radically toward **making many small, lightly-loaded trips**—even if this means visiting the same city multiple times and traveling greater total distance.

This observation is the cornerstone of our approach: rather than designing a single strategy, we adapt both the solution representation and the evolutionary operators to the β regime.

1.4 Baseline Solution

The baseline strategy visits each city individually: for every non-depot city i, the collector travels from city 0 to city i via the shortest path (empty), **picks up all gold** g_i, and returns to 0 (loaded). Let P_i^* denote the shortest path between 0 and i. The baseline cost is:

$$C_{\text{baseline}} = \sum_{i=1}^{n-1} \Big[C(P_i^*, 0) + C(P_i^*, g_i) \Big]. \tag{2}$$

This strategy makes exactly one round trip per city, collecting all gold in a single visit. It serves as the benchmark against which our solutions are evaluated. Our goal is to find solution plans with cost $C_{\text{solution}} < C_{\text{baseline}}$.

1.5 Partial Collection and Multiple Visits

A crucial feature of the WTCP is that *partial gold collection is allowed*: the collector may take any amount of gold from a city during a visit, and return later to collect the remainder. This is especially important when $\beta > 1$, because splitting a city's gold across multiple trips—each carrying a fraction g_i/k—can drastically reduce the total weight penalty compared to collecting all g_i in one trip.

2 Methodology

2.1 Solution Representation

We represent a solution as a list of *trips*, where each trip is a sequence of (city, gold) pairs. **A trip starts and ends at the depot (city** 0**).** Within a trip, the collector visits cities in order, picking up the specified amount of gold at each stop, accumulating weight as it progresses, and finally returns to the depot carrying the total weight gathered during that trip.

Formally, a solution $S = \{T_1, T_2, \ldots, T_m\}$ consists of m trips, where each trip

$$T_j = [(c_1, g'_1), (c_2, g'_2), \ldots, (c_{|T_j|}, g'_{|T_j|})]$$

specifies the cities visited and the gold collected at each. The solution is *valid* if and only if for every city $i \in V \setminus \{0\}$, the total gold collected across all trips equals g_i:

$$\sum_{\substack{j,\,(c,g')\in T_j \\ c=i}} g' = g_i \qquad \forall\, i \in V \setminus \{0\}. \tag{3}$$

This representation naturally supports both multi-city trips (beneficial when $\beta = 1$) and multi-trip cities (beneficial when $\beta > 1$), giving the EA flexibility to explore the full solution space.

2.2 Efficient Distance Computation

Since path evaluations dominate the computational cost, we invest in efficient distance infrastructure. At initialization, we precompute the full pairwise Euclidean distance matrix from city coordinates:

$$d_{ij}^{\text{euc}} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}. \tag{4}$$

For sparse graphs where edges do not exist between all city pairs, shortest paths must be computed via the graph structure. We use the A* algorithm with the Euclidean distance as an admissible heuristic, and cache all computed shortest paths and their distances for reuse.

For dense graphs (density > 0.99), we detect this condition at initialization and use direct Euclidean distances as a proxy for shortest paths, since in a complete graph the direct edge between any two cities exists and is the shortest path. This optimization substantially reduces overhead for large, dense instances.

2.3 Trip Evaluation

The cost of a single trip $T = [(c_1, g'_1), (c_2, g'_2), \ldots, (c_k, g'_k)]$ is evaluated by simulating the collector's journey:

1. Start at the depot (city 0) with weight $w = 0$.
2. For each stop (c_i, g'_i) in order: travel from the current city to c_i along the shortest path, accumulating cost $\sum_{(a,b)\in P} C([a,b], w)$ for each edge in the path; then add g'_i to the carried weight w.
3. After the last stop, return to the depot, paying $\sum_{(a,b)\in P} C([a,b], w)$ for the return path.

The total solution cost is the sum of all trip costs: $C(S) = \sum_{j=1}^{m} C(T_j)$.

Note that within a trip, **the order of city visits matters** because weight accumulates: visiting a far city first (while light) and a near city last (while heavy) typically reduces cost, especially when $\beta > 1$.

2.4 Population Initialization

The initialization strategy is designed to produce **high-quality starting solutions** by exploiting analytical insight into the cost function, rather than generating random solutions.

For each city c with gold g_c, we determine the optimal number of trips k^* by considering independent round-trip deliveries. Let d_c denote the Euclidean distance from the depot to city c. If the collector makes k equal trips, each carrying g_c/k kg, the approximate cost is:

$$\hat{C}(k) = k \cdot \left[2d_c + \left(\alpha \cdot d_c \cdot \frac{g_c}{k} \right)^{\beta} \right]. \tag{5}$$

The first term $2d_c$ is the base round-trip distance cost (going empty and returning loaded, where the empty leg contributes only d_c). The second term is the weight penalty for carrying g_c/k over distance d_c. We evaluate this expression for $k = 1, 2, \ldots$ and select the k^* that minimizes $\hat{C}(k)$, with early termination when the cost exceeds $1.05\times$ the current best (to avoid unnecessary computation).

Behavior across β regimes. When $\beta = 1$, the penalty term becomes $\alpha \cdot d_c \cdot g_c/k$, and the total is $k \cdot 2d_c + \alpha \cdot d_c \cdot g_c$, which is minimized at $k = 1$. Therefore, **the initialization naturally recovers the baseline strategy** of one trip per city. When $\beta > 1$, the penalty term decreases faster than $1/k$ (specifically as $1/k^{\beta}$), making trip splitting highly beneficial. For example, with $\beta = 2$ and a heavy city, k^* can easily reach values of 10–40, resulting in many lightweight trips that individually incur very low weight penalties.

This initialization ensures that even before evolution begins, the population already contains solutions that are near-optimal in terms of trip splitting for $\beta > 1$, and that match the baseline structure for $\beta = 1$. All individuals in the initial population share this deterministic construction; diversity arises through the evolutionary operators.

2.5 Trip Order Optimization

For trips containing multiple city visits, the order in which cities are visited can significantly affect cost due to weight accumulation. We employ a trip-order optimization routine that, for small trips (up to 6 stops), evaluates all permutations and selects the cheapest ordering. For larger trips, a greedy nearest-insertion heuristic is used: starting from the depot with zero weight, the next city is chosen to minimize the incremental travel cost at the current weight, and the process repeats until all cities in the trip are ordered. This routine is applied during initialization and after certain mutation and crossover operations that modify trip composition.

3 Evolutionary Operators

The EA uses a $(\mu + \lambda)$ selection strategy: in each generation, λ offspring are produced through mutation or crossover, then the best μ individuals from the combined parent-plus-offspring pool survive to the next generation. The choice of mutation and crossover operators is conditioned on the β regime, reflecting the fundamentally different optimization landscapes.

3.1 Selection

We employ two selection methods:

- **Roulette wheel selection**: Individuals are selected with probability proportional to their relative fitness advantage. Specifically, given maximum fitness $f_{\max}$ in the population, individual i receives weight $f_{\max} - f_i + 1$, ensuring all individuals have positive selection probability while favoring lower-cost (fitter) solutions.
- **Tournament selection**: A random subset of individuals (tournament of size 3) is drawn, and the fittest (lowest cost) is selected. This provides stronger selection pressure than roulette wheel while being computationally efficient.

3.2 Mutation Operators

We define five mutation operators, each targeting a different structural modification of the solution. The operator mix is β-dependent.

3.2.1 Swap Mutation

The swap operator selects two random (city, gold) entries from the solution and exchanges them. If both entries belong to the same trip, a simple within-trip swap is performed, altering only the visit order. If they belong to different trips, the operator performs an inter-trip exchange: city–gold pairs are swapped between trips, with special handling for the case where the destination city already exists in the target trip (in which case the gold amounts are merged rather than creating a duplicate entry). **This operator provides local perturbation that can improve both trip ordering and trip composition.**

3.2.2 Change-k Mutation (for $\beta > 1$)

This operator directly adjusts the number of trips allocated to a randomly chosen city. It collects all current entries for that city across all trips, computes the total gold, removes them, and then redistributes the gold evenly across a new number of trips $k' = \max(1, k \pm \delta)$, where $\delta \in \{-2, -1, 1, 2\}$ is chosen **uniformly** at random. Each new trip contains exactly g_c/k' kg of gold from city c. This operator is the primary mechanism for fine-tuning the trip-splitting granularity around the analytically estimated k^*, allowing the EA to discover per-city adjustments that the uniform initialization may have missed.

3.2.3 Merge Trips Mutation (for $\beta = 1$)

This operator randomly selects two trips and combines them into a single trip. When merging, if the same city appears in both trips, their gold amounts are summed (maintaining solution validity). After merging, the combined trip is reordered using the trip order optimization routine described in Section 2.5. This operator is particularly important for $\beta = 1$, where consolidating multiple single-city trips into a multi-city tour reduces the total distance traveled and thus the overall cost.

3.2.4 Split Trip Mutation

The inverse of merging: a randomly selected trip is split at a random point into two separate trips. This allows the EA to break apart overly long trips that may have accumulated excessive weight, and is useful in both β regimes as a means of maintaining solution diversity and correcting over-aggressive merging.

3.2.5 Move City Mutation (for $\beta = 1$)

This operator removes a single (city, gold) entry from one trip and inserts it into another trip at a random position. If the destination trip already contains an entry for that city, the gold amounts are merged. This provides a finer-grained alternative to full trip merging, allowing incremental reshuffling of city assignments between trips.

3.3 Operator Assignment by β Regime

For $\beta > 1$, the mutation operator distribution emphasizes trip-count adjustment:

- Change-k: 50%
- Split trip: 20%
- Swap: 30%

For $\beta = 1$, the distribution favors trip consolidation and city redistribution:

- Merge trips: 40%

- Move city: 40%
- Swap: 20%

This regime-specific weighting ensures that the EA's search effort is directed toward the modifications most likely to yield improvement given the problem structure.

3.4 Crossover Operators

3.4.1 Trip Crossover (for $\beta > 1$)

The trip crossover operator inherits a random subset of k trips from parent 1, then fills in the remaining gold requirements using parent 2's trips. Specifically:

1. Select k trips uniformly at random from parent 1 (where k is between 1 and half the number of trips).
2. Track the gold already covered for each city.
3. Iterate over parent 2's trips: for each city in each trip, if the city still has unmet gold demand, include the appropriate amount (up to the remaining need) in a new trip for the child, merging entries within the same trip if necessary.
4. Any city with remaining unmet demand is covered by a new single-city trip.

This crossover preserves trip-splitting patterns from both parents while ensuring solution validity.

3.4.2 Targeted Crossover (for $\beta = 1$)

The targeted crossover was specifically devised to address a critical challenge in the $\beta = 1$ regime: *population convergence*. Because the initialization for $\beta = 1$ produces solutions that are already near-optimal (effectively recovering the baseline's one-trip-per-city structure), the population rapidly converges to highly similar genotypes.

To mitigate this, the targeted crossover explicitly identifies and disrupts the ***common structural patterns*** shared between parents. The key insight is that if two parents assign the same group of cities to the same trip, this shared grouping represents a region of the solution that has not been explored in alternative configurations. By deliberately altering these shared subsequences, the operator creates children that differ precisely where the population has converged, opening room for exploration.

The operator proceeds as follows:

1. For each city, determine which set of cities shares its trip in parent 1 and parent 2 respectively.
2. Identify city groups that appear together in the same trip in *both* parents—these are the converged structural elements that the population has settled on.
3. With probability 0.5, pull a random subset of cities from such a shared group out of their current trip in the child, and either place them in a newly created trip or merge them into a different existing trip (with gold deduplication and trip reordering).

4 Experimental Setup

We evaluate our EA across a systematic grid of problem parameters to understand performance across different scales and β regimes. The experimental configurations are:

- **Number of cities**: $n \in \{100, 1000\}$, representing medium and large instances.
- **Graph density**: $d \in \{0.2, 1.0\}$, representing sparse and complete graphs.
- **Weight penalty parameter**: $\alpha \in \{1, 2\}$.
- **Cost exponent**: $\beta \in \{1, 2\}$, representing the linear and super-linear regimes.

All experiments use a fixed random seed of 42 for reproducibility. The EA parameters are:

- Population size: $\mu = 50$.
- Offspring per generation: $\lambda = 50$.
- Number of generations: 500 for $\beta = 1$ (with $n = 100$), 200 for $\beta = 1$ (with $n = 1000$), and 5 for $\beta = 2$ (since the initialization already provides near-optimal solutions and few generations suffice).
- Mutation rate: 0.3 (probability of applying mutation vs. crossover to produce an offspring).

The reduced generation count for $\beta > 1$ reflects the observation that the initialization already captures the dominant optimization (trip splitting), and further evolution provides only marginal refinement at substantial computational cost.

5 Results

Table 1 summarizes the experimental results across all parameter configurations.

Table 1: Comparison of baseline and EA solution costs across different problem configurations. Improvement is measured as percentage reduction from baseline cost.

n	Density	α	β	Gens	Baseline Cost	EA Best Cost	Improvement
100	0.2	1	1	500	25 266.41	25 251.93	0.06%
100	0.2	2	1	500	50 425.31	50 412.51	0.03%
100	0.2	1	2	5	5 334 401.93	113 729.51	97.87%
100	1.0	1	1	500	18 266.19	18 260.98	0.03%
100	1.0	2	1	500	36 457.92	36 454.41	0.01%
100	1.0	1	2	5	5 404 978.09	113 654.60	97.90%
1000	0.2	1	1	200	195 402.96	195 399.87	< 0.01%
1000	0.2	2	1	200	390 028.72	390 027.65	< 0.01%
1000	0.2	1	2	5	37 545 927.70	802 435.67	97.86%
1000	1.0	1	1	200	192 936.23	192 931.74	< 0.01%
1000	1.0	2	1	200	385 105.64	385 101.72	< 0.01%
1000	1.0	1	2	5	57 580 018.87	1 211 214.29	97.90%

5.1 Analysis of $\beta = 2$ Results

The most striking result is the consistent approximately **97.9% improvement** over baseline across all $\beta = 2$ configurations, achieved with only **5 generations** of evolution. This confirms that the dominant source of optimization in the super-linear regime is trip splitting—a structural insight captured entirely by the initialization procedure (Section 2.4).

To understand the magnitude, consider a city with $g = 500$ kg at Euclidean distance $d = 0.3$ from the depot with $\alpha = 1, \beta = 2$. The baseline cost for this city is:

$$C_{\text{base}} = 2 \times 0.3 + (1 \times 0.3 \times 500)^2 = 0.6 + 22\,500 = 22\,500.6. \tag{6}$$

If instead we make $k = 50$ trips, each carrying 10 kg:

$$C_{k=50} = 50 \times \left[2 \times 0.3 + (1 \times 0.3 \times 10)^2\right] = 50 \times (0.6 + 9) = 480. \tag{7}$$

This represents a 97.9% reduction for a single city, which explains the global improvement observed. The weight penalty, which dominates the cost when $\beta > 1$, scales as $(w/k)^\beta$ per trip but is multiplied by k trips, yielding a net scaling of $k^{1-\beta}$—which decreases with k when $\beta > 1$.

The consistency of the improvement across different values of n, density, and α confirms that this is a fundamental structural property of the cost function rather than a artifact of specific problem instances. The parameter α merely scales the penalty without changing the optimal trip structure, and graph density affects only path distances, not the weight-splitting logic.

5.2 Analysis of $\beta = 1$ Results

For $\beta = 1$, improvements are marginal. This is expected for several reasons:

When $\beta = 1$, the cost reduces to $C(P, w) = d_P(1 + \alpha w)$, making the total weight penalty independent of how gold is distributed across trips—carrying g/k over k trips costs the same as carrying g in one trip—so trip splitting offers no benefit and the only avenue for improvement is combining multiple cities into a single tour to save on total distance. However, finding such multi-city tours that beat the sum of individual round trips is essentially a TSP variant (NP-hard), and with random Euclidean graphs the opportunities are inherently limited, especially since shortest paths from the depot already share intermediate nodes in sparse graphs. These structural constraints explain the marginal improvements observed, which further degrade as n increases (from ~0.03–0.06% at $n = 100$ to <0.01% at $n = 1000$) because each city's contribution to total cost shrinks and the search space grows, making beneficial combinations harder to discover. Higher α slightly reduces the improvement by amplifying the weight penalty relative to the base distance, and dense graphs ($d = 1.0$) leave even less room for route optimization since direct edges already provide near-shortest paths between most city pairs.

5.3 Computational Efficiency and Convergence Behavior

An important practical consideration is the dramatically different computational cost *per generation* across β regimes. When $\beta > 1$, the initialization produces solutions with a large number of single-city trips (each city may be visited k times, with k potentially in the tens), leading to solutions that contain hundreds or even thousands of trips for large graphs. Since every fitness evaluation requires computing the cost of each trip individually, the per-generation cost grows substantially with the total number of trips. For $n = 1000$ and $\beta = 2$, a single generation can take over 15 seconds, making extended evolutionary runs prohibitively expensive. Consequently, we limit the $\beta > 1$ runs to only 5 generations—enough to comfortably defeat the baseline, but far from exhausting the optimization potential.

For $\beta = 1$, solutions remain compact (roughly one trip per city), making per-generation evaluation fast and allowing 200–500 generations within reasonable time.

5.4 Convergence Analysis

Figures 1 and 2 illustrate the convergence behavior of the EA for the two β regimes on the same problem instance ($n = 100$, density $= 1.0$, $\alpha = 1$).

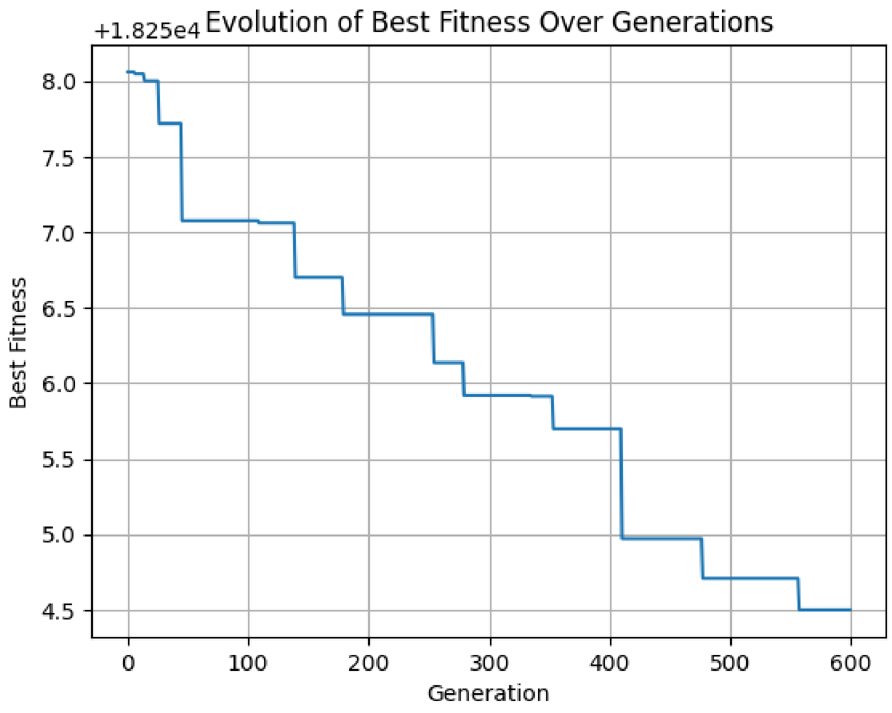

Figure 1: Convergence curve for $\beta = 1$ ($n = 100$, density $= 1.0$). The staircase pattern reflects the difficulty of finding improving moves on a flat fitness landscape, with long plateaus between rare beneficial mutations.

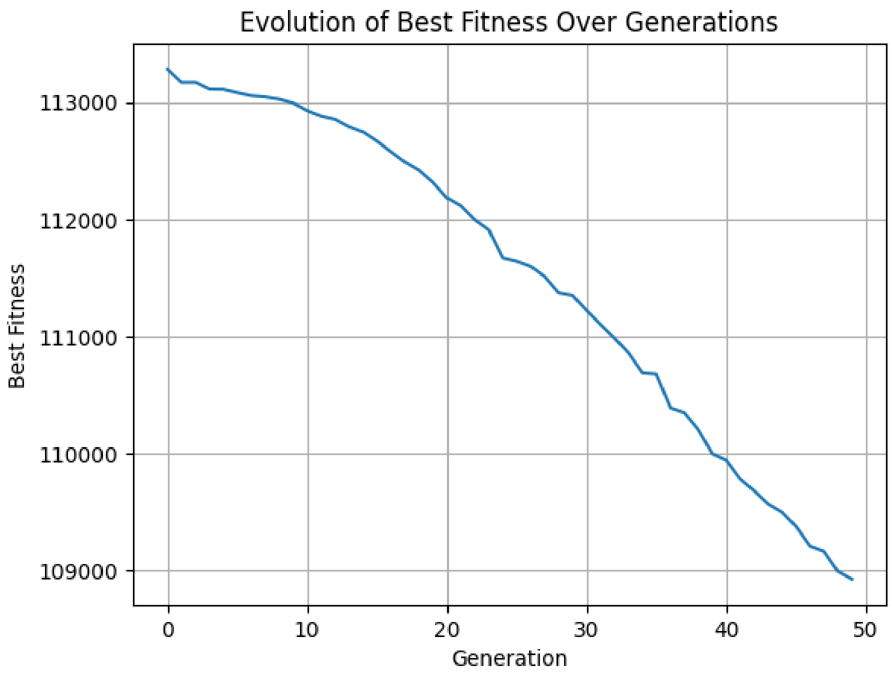

Figure 2: Convergence curve for $\beta = 2$ ($n = 100$, density $= 1.0$). The smooth, steadily decreasing trajectory indicates that significant further improvement remains possible beyond the 50 generations shown.

The two curves reveal strikingly different convergence dynamics. For $\beta = 1$ (Figure 1), the fitness curve exhibits a characteristic *staircase* pattern: long plateaus where no improvement is found, punctuated by occasional discrete jumps when the EA discovers a rare beneficial rearrangement. This confirms that the $\beta = 1$ fitness landscape is largely flat around the baseline solution, and improvements come from infrequent, lucky mutations rather than systematic gradient-following. Even after 600 generations, the curve has not fully plateaued, suggesting that marginal improvements would continue to trickle in with additional generations, though at diminishing rates.

For $\beta = 2$ (Figure 2), the fitness curve shows a *smooth and continuous* descent with no sign of convergence even after 50 generations. The slope remains steep throughout, indicating that substantial further improvement is readily achievable with more generations. This confirms that the 5-generation limit used in our main experiments (Table 1) was a practical concession to computational cost rather than a reflection of convergence—the reported 97.9% improvement over baseline, while already dramatic, leaves considerable room for further optimization through fine-tuning of trip counts and orderings. The richness of the $\beta > 1$ optimization landscape, combined with the high per-generation computational cost, presents an interesting trade-off between solution quality and runtime that could be addressed with more efficient evaluation strategies in future work.

5.5 Solution Validity

All solutions produced by the EA pass the validation check: for every city, the total gold collected across all trips exactly matches the city's gold amount (within floating-point tolerance of 0.01 kg). This confirms that the mutation and crossover operators, despite their complex gold-redistribution logic, correctly maintain solution feasibility throughout the evolutionary process.

6 Conclusion

We presented an Evolutionary Algorithm for the Weighted Traveling Collector Problem that adapts its strategy to the β regime of the cost function. The key insight driving our approach is that the exponent β fundamentally reshapes the optimization landscape: for $\beta > 1$, the super-linear weight penalty makes trip splitting the dominant optimization lever, while for $\beta = 1$ the linear cost structure renders the baseline nearly optimal and leaves only marginal gains through route combination. By embedding this analytical understanding into the initialization (optimal k-splitting) and tailoring mutation and crossover operators to each regime, our EA achieves consistent $\sim$97.9% improvement over baseline for $\beta = 2$ across all tested configurations, while producing valid solutions that correctly account for all gold. The convergence analysis further reveals that the $\beta > 1$ regime holds substantial untapped potential beyond what computational constraints allowed us to exploit, suggesting that more efficient evaluation methods or hybrid approaches could push performance even further. For $\beta = 1$, the flat fitness landscape and rapid population convergence remain fundamental challenges, and overcoming them would likely require problem-specific heuristics beyond the scope of general-purpose evolutionary search.