Computational Intelligence Report

Niccolo Antonelli-Dziri

s350296

Politecnico di Torino

January 2026

Abstract

This report summarizes the work carried out during the course and provides a self-contained record suitable for offline evaluation. It documents both the final project and the development process, including laboratory exercises, design decisions, feedback received, and subsequent revisions.

Github repository link: https://github.com/NiccoloAntonelliDziri/project-work

Contents

1   Introduction

This document describes the work performed in the course laboratories and the final project. It is intended as a self-contained record that allows evaluators to understand, reproduce, and assess the laboratory exercises, design decisions, code contributions, feedback iterations, and final outcomes without immediate access to the GitHub repository.

The report is organized as follows. Each lab is presented as a self-contained chapter describing goals, environment and tools, the work performed, test results, and a discussion of the pros and cons of the chosen approach. Important design decisions and external sources are explicitly declared in the relevant sections to ensure full transparency.

2   Laboratory 1: Knapsack problem

The goal of the first laboratory is to solve a multi-knapsack, multi-dimensional knapsack problem using techniques from the lectures.
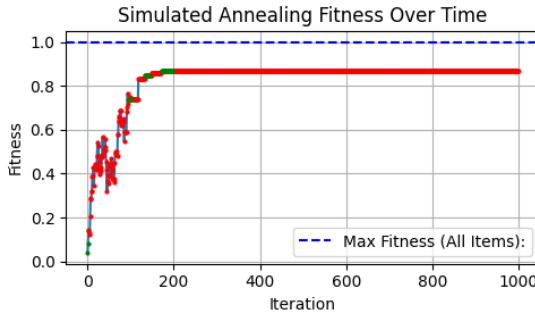
2.1   Proposed Approach

The main difficulty of this lab was finding valid solution that does not violate capacity constraints. A purely random approach often leads to invalid states, especially with multiple knapsacks and dimensions. To address this, I refined the neighborhood generation mechanism (the `tweak` function). Initially, the standard `tweak` simply modified a random item's assignment. However, this did not actively repair invalid solutions. I therefore introduced a `tweak_remove` variant: if a random change causes a constraint violation, the algorithm attempts to repair the solution by removing items from the overfilled knapsack, guifding the search back towards the feasible region.
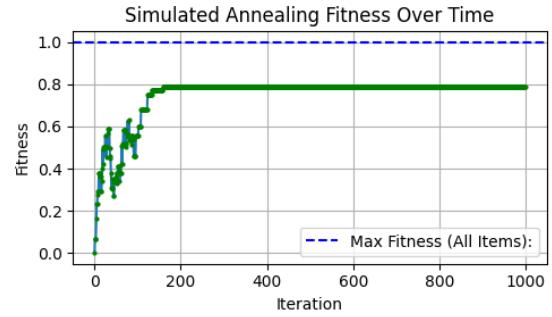
I also experimented with the initialization strategy. While Simulated Annealing typically starts from a random state, I found that starting from an "empty" solution (all items unassigned) yielded better results. This allows the algorithm to incrementally construct a valid solution instead of spending early iterations fixing a heavily violated random initialization.

To further improve performance, I tested two greedy variations. First, a greedy `tweak` that preferentially attempts to add high-value items to knapsacks with remaining capacity. Then, I used a greedy initialization strategy, where the simulated annealing process starts from a solution generated using a value-density heuristic rather than starting empty state. While the greedy tweak
often produces good local solutions, especially when combined with greedy initialization, it tends to getstuck in local optima because it reduces exploration of the search space.
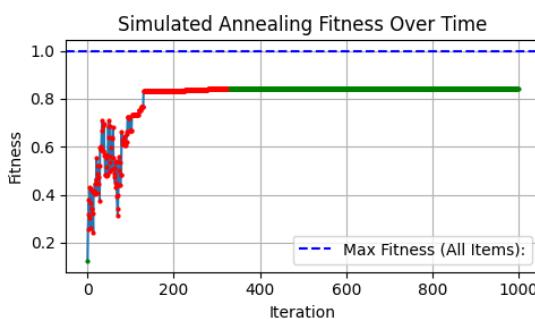
Figure 1 illustrates the typical progress of the algorithm. The plot distinguishes between valid (green) and invalid (red) states during the search. The penalty-based fitness function allows the algorithm to traverse invalid states to reach better local optima, while the cooling schedule eventually stabilizes the solution into a valid configuration.
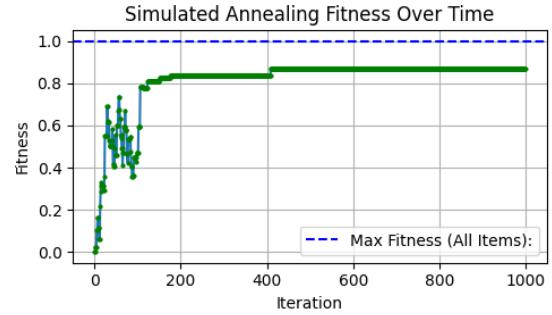
(a) `tweak`: Simple random reassignment
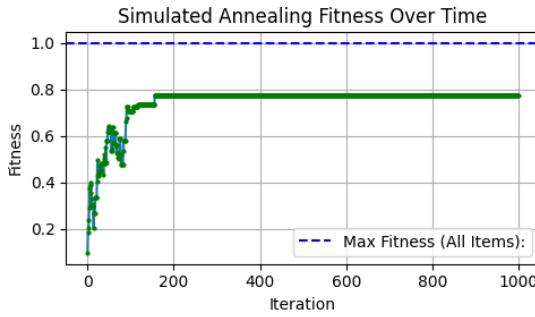
(b) `tweak_remove`: Reassignment with repair

(c) `tweak_multiple`: Multiple random changes
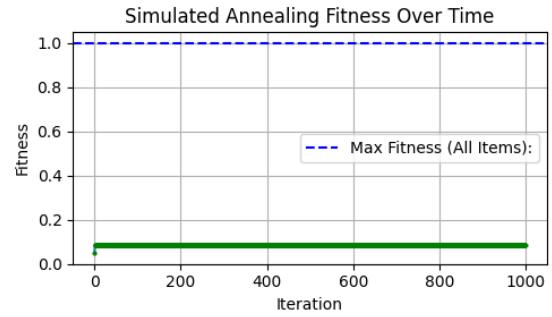
(d) `tweak_multiple_remove`: Multiple changes with repair

(e) `tweak_gready`: Greedy addition

(f) `tweak_swap`: Swapping items between knapsacks

Figure 1: Comparison of fitness evolution using different tweak functions in Simulated Annealing. Green dots indicate valid solutions, red dots indicate constraint violations.

To keep objective values comparable across instances, the fitness is normalized to lie in $[0, 1]$. Concretely, the fitness computes the total value of selected items, subtracts a penalty equal to the sum (over knapsacks and dimensions) of the capacity overflows, and divides the result by the maximum attainable total value.

```python
def fitness(s):
    """
    Compute the fitness of a solution.
    The fitness is the total value of the items in the knapsacks,
    minus a penalty for each constraint violation.
    """
    total_value, total_weights = totals(s)

    penalty = 0
    for k in range(NUM_KNAPSACKS):
        for d in range(NUM_DIMENSIONS):
            if total_weights[k, d] > CONSTRAINTS[k, d]:
                penalty += total_weights[k, d] - CONSTRAINTS[k, d]

    max_possible_value = np.sum(VALUES)

    return (total_value - penalty) / max_possible_value
```

2.2 Received reviews

2.2.1 Adr44mo

Peer Review: Multi-Dimensional Knapsack Solver

Well-structured code with clear functions for constraint checking, fitness evaluation, and solution repair, making it easy to follow and maintain.

Effective use of simulated annealing for optimization, including neighbor generation, acceptance criteria, and cooling schedule, which is appropriate for the multi-dimensional knapsack problem.

Robust repair mechanism that removes low-value items to ensure feasibility, enhancing solution quality.

Includes logging, early stopping, and visualization of fitness progress, aiding in monitoring and debugging.

In repair_solution, you can try to maintain an incremental capacity updates during item removal to skip recomputing totals from scratch each time.

For neighbor generation in simulated annealing, try swapping items between knapsacks instead of random reassignment to maintain better solution quality with fewer invalid states.

This implementation demonstrates strong algorithmic design and attention to detail, providing a reliable heuristic for solving complex knapsack instances.

2.2.2 Michele-Curci

Great choice using a 1D vector as a solution solving inherently the problem of multi-knapsack assignment of a single item. When calculating the fitness it could be helpful to use a tunable penalty coefficient, moreover the normalization may reduce the effect of the penalty when total_value is large. The function is_solution_valid() is correct but could be included in the fitness function. The function tweak_remove() simply removes the last item inserted but it could be more useful to remove the item that most violates the constraint (for example using the value-to-weight ratio). The same goes for the multiple variation. The hill climbing method besides suffering from getting stuck in local maxima (as expected), it does not have a constraint repair so it may waste iterations exploring unfeasible states, it might also bounce around in a flat region (plateu) since it accepts equal fitness. Good implementation of simulated annealing. In the function simulated_annealing_gready() it is invoked the function totals(s) when it should have been totals(current_solution). Could have changed the titles in the plots for better understandabilty.

Overall it is an extensive exploration of the simulated annealing algorithm with different tweak strategies, maybe interpreting the fitness as the total value instead of a percentage could have made the results easier to read but within the code it provides good comparison and understanding of the solutions.

2.2.3 Tom-KB

Review Lab 1 - K/BIDI Thomas - s350263

This notebook implements two solutions to solve the knapsack problem : Hill Climbing and Simulated Annealing. However, only the SA algorithm is tested.

The usage of graphs is a good idea to show the evolution of the algorithm, but the lack of independent legend for the different tweak functions makes them difficult to read. The fitness function is complex, the concept of penalties and the use of a ratio over the maximal value possible in a solution is difficult to interpret, since the optimal solution doesn't always mean having all the items every time. A simple sum of the values would have been easier to read, and the total unconstrained sum of values would have provided good information, but as an indicator. Multiple tweak functions were proposed, a mix of them could have give better solutions to the problems, since right now you're almost only using one at a time. For example, the swap alone can be useless, since you are only using tweak_remove if the swap creates a clash with the constraints, but if all your items can be swapped without creating any clash, you are stuck on a plateau.

Inside the simulated_annealing_gready(...), the line ", total_weights = totals(s)" should be ", total_weights = totals(current_solution)"

Some typos make the comments hard to read, the docstring could also have more information because your three variations of the SA algorithm have the exact same docstring.

The idea of using a 1D vector to represent the knapsack by an index is good, it avoids having one item in multiple knapsacks easily.

These solvers seem to provide solutions to the problem, the implementations of the algorithms worked.

Good job !

2.3 Given reviews

2.3.1 ChristianPanchetti

Your algorithm seems really good and fast.

To better visualize the evolution of it you can plot the current fitness over the iterations. It's also useful for adjusting the number of iterations if the curve is still increasing and does not seem to plateau.

To better compare the behavior of the three problems, you can normalize the fitness between 0 and 1 where 1 is the "ideal" solution (where every item is taken and fitness is in it's theoretical maximum). It's not perfect because that solution may not be valid but it's gives a hint of the different difficulty of each problem.

2.3.2 Michele-Curci

To optimize your simulated annealing algorithm you can force the mutation instead of letting the random generator choosing the same knapsack as the item is currently in. For example like this:

```
knapsacks = rng.integers(-1, NUM\_KNAPSACKS, size=num\_changes)
for idx, knapsack in zip(mutation\_idx, knapsacks):
    if knapsack == neighbor[idx]:
        knapsack = (knapsack + 2) % NUM_KNAPSACKS - 1
    neighbor[idx] = knapsack
```

Or with a while loop and randomly generating an other until they are different.

You curve SA seems like it still increasing at the last iteration, so maybe it's worth increasing the number of steps to improve the solution. You can also increase the cooling rate to make more worsening changes for longer if there is more steps.

Another way of visualizing the greatness of the best_solution is to compare it to the "ideal" solution, where each item is taken. That way you can calculate a percentage of completeness, which is more meaningful than the raw fitness value, although not perfect because it's almost certainly not a valid solution.

For the genetic algorithm and the ant colony, I don't understand why the fitness is negative and the curves almost vertical. Maybe lowering the penalty in the fitness can help and lowering the amount of iteration to better see the evolution of those algorithms.

It would be nice to also check the behavior of each algorithm with each problem, that way they can be compared to each other.

2.4 Reflections on Peer Feedback

One key observation from the reviews was the importance of computational efficiency, especially regarding the repair mechanism. Initially, my solution recalculated knapsack totals at every repair step. To improve performance, I decided to incorporate the repairing directly inside of the tweak functions.

I implemented a `tweak_swap` function that cycles items between knapsacks instead of removing them entirely, increasing diversity in the search space and helping preserve feasibility. However, `tweak_swap` performed worse than other tweak functions in the experiments (see Figure 1f): swapping tends to produce smaller, local changes, which reduces invalid states but is less effective at escaping plateaus or yielding large objective improvements. Due to time constraints I did not combine multiple tweak functions for this lab but I applied that idea later in the genetic algorithm of the final project.

3 Laboratory 2: Traveling Salesman Problem (TSP)

The goal of the second laboratory is to solve the Traveling Salesman Problem (TSP), a classical optimization problem, using evolutionary algorithms.

The TSP requires finding the Hamiltonian path of a given graph, i.e. the shortest path that visits all cities exactly once and returns to the starting point. Two variants of the problem are explored: the symmetric TSP, in which the distance between any pair of cities is the same in both directions, and the asymmetric TSP, where the cost of traveling from one city to another may differ depending on the direction of travel.

3.1 Proposed Approach

To address the TSP, the following evolutionary algorithm was implemented:

```
1  def evolutionAlgorithm(distance_matrix, lmbda, mu, num_generations, swap_function,
       max_no_improvement, force_greedy_start, use_fast_distance):
2
3      n_cities = distance_matrix.shape[0]
4      population = generateInitialPopulation(n_cities, lmbda, force_greedy=
       force_greedy_start, d_matrix=distance_matrix)
5
6      best_fitness = float('inf')
7      best_solution = None
8      evolution_distance = []
9      no_improvement_counter = 0
10
11     # Check if we can use fast distance (only works with swap2opt on symmetric matrices)
12     can_use_fast = use_fast_distance and (swap_function == swap2opt)
13
14     # Initial evaluation
15     fitness_values = evaluatePopulation(population, distance_matrix, lmbda, n_cities)
16
17     for generation in range(num_generations):
18
19         selected_indices = selectedParentsIndices(fitness_values, mu)
20
```

```
21          current_fitness = np.min(fitness_values)
22          evolution_distance.append(current_fitness)
23
24          if current_fitness < best_fitness:
25              best_fitness = current_fitness
26              # Keep a copy of the best solution found so far
27              best_solution = population[selected_indices[0]].copy()
28              no_improvement_counter = 0
29          else:
30              no_improvement_counter += 1
31              if no_improvement_counter >= max_no_improvement:
32                  print(f"No improvement for {max_no_improvement} generations. Stopping
      early at generation {generation}.")
33                  return best_solution, best_fitness, evolution_distance
34
35          parents = population[selected_indices]
36          parent_fitness = fitness_values[selected_indices]
37
38          # Create children with or without fast distance
39          if can_use_fast:
40              population, fitness_values = createChildrenFast(parents, parent_fitness,
      n_cities, lmbda, mu, distance_matrix, swap_function)
41          else:
42              population = createChildren(parents, n_cities, lmbda, mu, swap_function)
43              fitness_values = evaluatePopulation(population, distance_matrix, lmbda,
      n_cities)
44
45      # Final check after all generations
46      if np.min(fitness_values) < best_fitness:
47          best_solution = population[np.argmin(fitness_values)].copy()
48          best_fitness = np.min(fitness_values)
49
50      return best_solution, best_fitness, evolution_distance
```

The algorithm maintains a population of size λ and selects the top μ parents for each generation. Each parent produces $\frac{\lambda}{\mu}$ children by applying a swap operator (`swap2opt` or `swapRandom`), and the best parent is copied to avoid fatal regressions. When `swap2opt` is applied to symmetric distance matrices, the `fastDistance` routine computes the cost of a 2-opt move in $O(1)$ using a delta formula, substantially reducing per-child evaluation cost. The 2-opt is effective for TSP because it directly targets edge crossings, which commonly constitute large, improvable structures in tours. Random pair swaps serve as smaller exploration steps and are useful for diversification but are weaker as a replacement for 2-opt. For this reason I used them mainly on the simpler G instances.

The greedy initial solution is a nearest-neighbor greedy heuristic from every starting city and keeping the best tour found. Greedy seeding provides much better initial solutions and accelerates convergence. This is observed especially in the R2 problems where the greedy solution seems already optimal. However, populating the initial population entirely with greedy duplicates reduces diversity and can cause premature convergence, as we can see from problem instance R2_1000 where no matter the choice of parameters, the best distance stayed constant all along the iterations. As a result, having just 1 generation does not change the result.

Asymmetric TSP: Solving the asymmetric TSP was challenging because the standard 2-opt operator and its $O(1)$ delta update assume a symmetric distance matrix, while computing full tour costs is too slow on larger asymmetric instances. I wanted a way to apply the same optimization of the symmetric problems to the asymmetric ones. To handle asymmetry, I transformed the graph using a graph-doubling technique. Each original city is split into an entering and a leaving node, producing a symmetric $2n \times 2n$ matrix, effectively doubling the size of the problem. A tour in the doubled graph (e.g., $0 \rightarrow 3 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 5 \rightarrow 0$) maps to a valid tour on the original cities via the conversion routines. I then applied the same evolutionary framework using a specialized operator that performs 2-opt on the simple path and reconverts to the doubled representation.

Observing the evolution paths, the algorithm benefits significantly from using greedy initialization in terms of convergence rate. The penalty for invalid solutions during the asymmetric TSP caused occasional disruptions but generally performed well with the simulated annealing approach.

Table 1: Summary of experiments: parameters and best distances

Instance	Type	λ	μ	Generations	max_no_improve	Force greedy	Swap	Best distance
Test	Test	200	50	200	50	False	swap2opt	2823.79
G10	G	200	50	200	50	False	swap2opt	1497.66
G20	G	200	50	200	50	True	swap2opt	1755.51
G50	G	200	20	1000	100	True	swap2opt	2629.99
G100	G	200	20	1000	300	True	swap2opt	4287.97
G200	G	200	20	5000	1000	True	swap2opt	5688.67
G500	G	200	20	20000	3000	True	swap2opt	9191.54
G1000	G	200	20	50000	4000	True	swap2opt	13118.49
R1_10	R1	200	50	1000	50	False	swap2optNonS	184.27
R1_20	R1	500	50	200	100	False	swap2optNonS	340.21
R1_50	R1	200	20	1000	200	False	swap2optNonS	596.59
R1_100	R1	500	20	1000	200	True	swap2optNonS	736.37
R1_200	R1	500	20	1000	200	True	swap2optNonS	1103.90
R1_500	R1	500	20	1000	200	True	swap2optNonS	1744.17
R1_1000	R1	500	20	1000	200	True	swap2optNonS	2533.81
R2_10	R2	500	20	1000	200	True	swap2optNonS	-411.70
R2_20	R2	500	20	1000	200	True	swap2optNonS	-796.86
R2_50	R2	500	20	1000	200	True	swap2optNonS	-2232.38
R2_100	R2	500	20	1000	200	True	swap2optNonS	-4667.68
R2_200	R2	500	20	1000	200	True	swap2optNonS	-9603.77
R2_500	R2	500	20	1000	200	True	swap2optNonS	-24584.05
R2_1000	R2	1	1	1	1	True	swap2optNonS	-49477.87

3.2 Received reviews

I did not receive any review for this laboratory.

3.3 Given reviews

3.3.1 sina-behnam

Hi,

Overall your code is well organized and clear. If I'm correct you handle the R1 and R2 problems by changing the opt function and thus you are loosing the optimization of fast_2opt provided for the symmetric case. Another way you could handle the problem is transforming the asymmetric matrix into a symmetric one by doubling it size. If the optimizations are good it may be worth trying and comparing the results.

3.3.2 Ale-DalMas

Hi,

Your implementation seems quite good but it would be nice to see how it works for the three types of problems.

Plotting the evolution of the fitness can help you finetune the parameters. And then you could add a early stop if the fitness doesn't improve for a fixed number of generations.

4 Laboratory 3: Shortest Path Problem

The objective of the third laboratory is to solve the Shortest Path Problem in directed graphs. This involves finding the path with the minimum total weight between a source node and all other reachable nodes.

4.1 Proposed Approach

Two fundamental algorithms were implemented and compared: Dijkstra's algorithm and the Bellman-Ford algorithm.
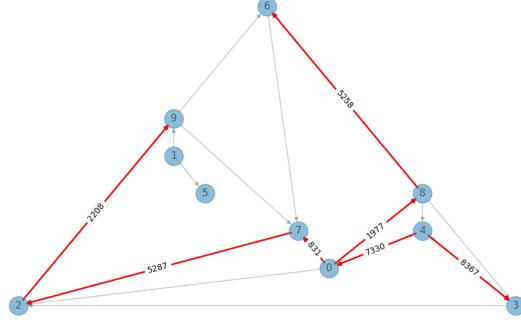
Figure 2: Shortest path from 4 to all with $N = 10$, $\rho = 0.15$ and $\epsilon = 10$.

Dijkstra's Algorithm: For graphs with exclusively non-negative weights, I implemented Dijkstra's algorithm using a priority queue (min-heap). This ensures an efficient exploration of the graph by always expanding the node with the current shortest known distance.
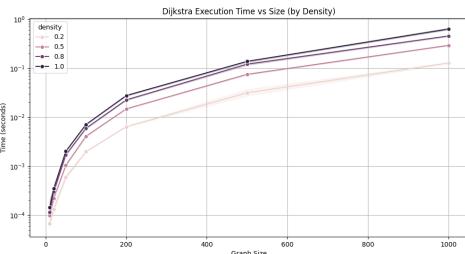
Bellman-Ford Algorithm: To handle graphs with negative edge weights, I implemented the Bellman-Ford algorithm. Beyond computing shortest paths, this implementation is capable of detecting negative-weight cycles.

4.2 Experimental Results
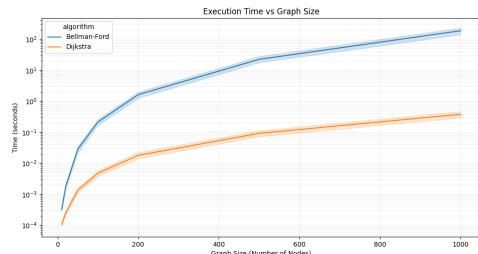
A comprehensive benchmarking script was executed to record the behavior of both algorithms across a wide range of parameters:

- **Size:** $N \in \{10, 20, 50, 100, 200, 500, 1000\}$ nodes.

- **Density:** $\rho \in \{0.2, 0.5, 0.8, 1.0\}$.

- **Noise Levels:** Impacting the variance of the edge weights.

Dijkstra's algorithm proved to be significantly faster, especially as the graph size and density increased, making it the preferred choice for large-scale geographic or transport networks where negative weights are absent. In contrast, Bellman-Ford's execution time scaled more aggressively, but it remained indispensable for instances involving negative noise levels where Dijkstra would fail.

(a) Dijkstra Execution Time vs Size (by Density)

(b) Execution time of both algorithms over the size.

4.3 Received reviews

4.3.1 nicolettatoma

Your solution is really good, i would only recommend some improvements. You repeat several times the creation and definition of graphs as well as the path-search logic, i would avoid those repetitions making some functions to call when necessary. In order to avoid many computations , you can try to use numpy built-in operations next time. By the way, I really appreciated your graph visualizations , they clearly represent all the paths and make the results easy to understand.

4.3.2 parsa-tavakoli

Strengths Comprehensive Algorithm Implementation

Both Dijkstra's and Bellman–Ford are implemented in a clear and structured way.

Dijkstra's version uses a priority queue correctly, handles visited nodes, and cleanly reconstructs paths. Bellman–Ford includes cycle detection and proper edge relaxation. You even provide both single-source and all-pairs variants, which makes the project more versatile.

Robust Problem Generation

The create_problem() function is a highlight:

Variable graph sizes Adjustable density Noise control Optional negative weights Deterministic seeding

And basing distances on Euclidean coordinates adds a nice "real-world" feel. Helpful Visualizations

The graph plots are clear, the shortest-path overlays are readable, and the runtime comparisons make it easy to see performance differences. Areas for Improvement Code Quality Tweaks

A few areas could be polished.

Bellman–Ford's negative cycle detection has a small bug in how cycles are reconstructed. Some commented-out code in the notebook should either be removed or explained. There's an unused import (combinations) that can be cleared out.

Algorithm Correctness

Dijkstra's algorithm shouldn't run on graphs with negative weights, but the code doesn't enforce this. A quick check would help. Path reconstruction currently returns a "path" even for unreachable nodes. A clearer handling would make the results easier to interpret.

Conclusion

This project shows solid effort and understanding of graph algorithms. The foundation is strong, and with a bit more polish especially around verification, clarity, and analysis it can become a high-quality, professional-grade implementation. Overall, you put in good work here. Good job!

4.4 Given reviews

4.4.1 mistru97

It's more a questions than a review because the implementations of the algorithms are good.

How much time did the problems bigger than 100 take ?

Why did you choose A* instead of the Dijkstra algorithm ?

Also I noticed that you use the algorithms to find the best path between a source and a destination and not between a source and every other nodes. You can get this information "free" without making the algorithm that much worse because you are still looking through most of the nodes. It is just a matter of keeping track of them.

For me it took 1h to compute the paths for all of the different problems (every size, density, noise and negative values)

4.4.2 nicolettatoma

I noticed that you used the algorithms provided by networkx. I guess it's much faster and optimized than your own python version of the algorithms. You could have done also source to all or all to all since the execution time is small, to compare efficiencies. You also do not keep the path but only the final cost. I can be helpful to plot the graph and see how the algorithms works or to visually see the optimal path that was found.

4.5 Reflections on Peer Feedback

One piece of feedback that stood out was the recommendation to prefer NumPy over Python built-ins for performance. I followed this advice in the final project, replacing pure-Python loops and data structures with NumPy vectorized operations and array types where appropriate (for example, distance matrix computation and population evaluations), which noticeably reduced runtime. In addition, compute-intensive parts were further sped up by moving the relevant code to Cython.

5 Final project:

The objective of this project is to model and solve a complex path-finding optimization problem that extends the classic Traveling Salesman Problem (TSP) and Vehicle Routing Problem (VRP). The problem uses a dynamic cost model in which the cost to traverse an edge depends not only on the edge distance but also on the payload carried, producing a non-linear, load-dependent component.

In a real-world context, this scenario mimics logistics operations where fuel consumption or energy costs scale non-linearly with the weight of the cargo. The project explores the use of heuristics and meta-heuristics, specifically Genetic Algorithms (GA) enhanced with Cython optimizations, to navigate the vast search space of potential routes and find an efficient solution that minimizes the total operational cost.

5.1 Problem definition

The problem is defined on a connected graph $G = (V, E)$ with $V = \{0, 1, \ldots, n - 1\}$ (nodes embedded in $[0, 1]^2$) and edge set E. Node 0 is the depot (start and end). Each city $i \in V \setminus \{0\}$ contains gold $g_i \in [0, 1000]$ (the depot has $g_0 = 0$). Each edge $(i, j) \in E$ has a travel distance $d_{ij} \geq 0$. The objective is to collect all gold and return it to the depot and the salesman (or thief in this case) may return to the depot multiple times to deposit collected gold before continuing the collection.

The primary challenge of this problem lies in its non-linear cost function. When an agent traverses an edge $(u, v) \in E$ with physical distance d_{uv}, carrying a total weight of gold w, the cost C is calculated as:

$$c_{uv}(w) = d_{uv} + (\alpha \cdot d_{uv} \cdot w)^\beta \tag{1}$$

with α a scaling factor that adjusts the impact of the weight and β an exponent that determines the non-linearity of the cost relative to the weight.

The total objective function is:

$$C = \sum_{(i,j) \in E} c_{ij}(w_i) x_{ij} \tag{2}$$

Here, x_{ij} is a binary variable that has the value 1 if the arc going from i to j is considered part of the solution and 0 otherwise.

For simplicity let's use the notation where we sum over the given path p:

$$C = \sum_p c_{ij} = \sum_p d + (\alpha d w)^\beta \tag{3}$$

The goal is thus to find the optimal path p that minimizes the objective C.

I have found similar problems in the literature but no deterministic algorithm I could directly use on this specific problem and optimally solve it. The main difficulty being the non-linearity over the weight carried because gathering gold from many cities in a single long route reduces the total distance traveled but causes the cost to explode due to the accumulated weight w raised to the power of β. Alternatively, performing multiple trips back to the depot keeps the weight low but significantly increases the total distance. The optimization task is to find the sweet spot between these two extremes. However I noticed that this problem can be rephrased as a Vehicle Routing Problem (VRP) with an additional parameter K, the set of homogeneous vehicles with infinite capacity available at the depot and the following two operational constraints:

- **Graph Traversal:** Vehicles may traverse the graph freely. The movement between any two nodes i and j is assumed to follow the shortest path in G. Therefore, we operate on the metric closure of the graph with distance matrix D_{ij} representing the shortest path distance between i and j.

- **Cumulative Load:** Let w_i denote the cumulative load carried by a vehicle immediately after servicing node i. When a vehicle traverses an edge (i, j), it carries the load w_i. Upon arriving at j, the load increases: $w_j = w_i + g_j$.

This reformulation of the problem enables the usage of the OR-Tools software suite developed by Google for solving linear programming, mixed integer programming, constraint programming, vehicle routing, and related optimization problems that is widely used in the industry.

5.2   Challenges and workarounds for standard solvers

Standard VRP solvers (such as Google OR-Tools) generally assume linear cost structures to utilize Integer Linear Programming (ILP) or optimized Constraint Programming (CP) heuristics. Those solvers can only manipulate integers. In the implementation, the distance was scaled up by some big factor and then rounded and the gold value was just rounded to the nearest integer. But in our case, if $\beta \neq 1$, the second term introduces a non-linearity that cannot be directly minimized by the routing engine's standard energy cost evaluators.

OR-Tool processes the Vehicle Routing Problem as if each vehicle could go from anywhere to anywhere, like on a real map, and not like a set of nodes connected on a graph. Therefore, it can't calculate the correct cost of the different paths chosen on the graphs with a density of less than 1. We could circumvent the restriction on the distance matrix, but not for the gold because it would stop calculating all the possible paths. It would be infeasible and would be an equivalent to brute force the solution. This problem can be addressed by using the approximation in section 5.4.

The following sections describe the successive steps of reasoning that structured my thinking while tackling the problem.

5.3   Linearization via Taylor Expansion

To utilize standard solvers, we approximate the non-linear cost function using a first-order Taylor expansion (tangent line approximation) around an estimated operating load.

Let's approximate $f(x) = x^\beta$ the non-linear part of the cost in (1) by it's tangent at a chosen point a, and $x > 0, \beta > 0, \beta \neq 1$.

$$T_a(x) = a^\beta + \beta a^{\beta-1}(x - a) \tag{4}$$

This first-order Taylor linearization assumes $f(x) = x^\beta$ is differentiable on $\mathbb{R}_+$ ($\beta > 0, \beta \neq 1$. Note that f is convex for $\beta > 1$ and concave for $0 < \beta < 1$. As a result, a first-order tangent at a point a is a global under-estimator when $\beta > 1$ and a global over-estimator when $0 < \beta < 1$. The approximation quality depends on the choice of expansion point a and degrades when actual segment loads deviate substantially from a, particularly when β is far from 1. For such cases, higher-order terms become non-negligible, and the linear model can produce large biases in the estimated cost.

We can then replace the expression for T_a inside of (1) and the result in (3) to find the desired approximation. Let's call the approximate cost over an arc $\widetilde{c}$ and the total cost $\widetilde{C}$ where $\widetilde{C} = \sum_p \widetilde{c_{ij}}$. We have:

$$\widetilde{C} = N\alpha^\beta a^\beta (1 - \beta) + \sum_p d + \alpha^\beta \beta a^{\beta-1} dw \tag{5}$$

Where N is the number of arcs used in the path. Note that the first term is constant in respect to d and w and can be omitted in the implementation.

It remains to identify a good choice of the expansion point a in order to obtain an accurate approximation.

Let's have the approximation error $E_p = |C - \widetilde{C}|$. over a fixed path p. Since $f(x) = x^\beta$ is convex for $\beta > 1$ on $\mathbb{R}_+$, the first-order Taylor expansion at point a is a global underestimator of the function:

13
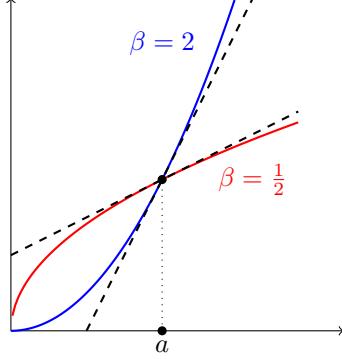
Figure 4: Illustration of the function $f(x) = x^\beta$ and its first-order Taylor approximation (tangent line) at the point $x = a$.

$x^\beta > T_a(x)$. In the same way, if $\beta < 1$, $x^\beta < T_a(x)$. Since both C and $\widetilde{C}$ are defined as a sum of positive terms we can thus eliminate the absolute value and define E as:

$$E_p = \begin{cases} C - \widetilde{C} > 0 & \text{if } \beta > 1 \\ \widetilde{C} - C > 0 & \text{if } \beta < 1 \end{cases} \tag{6}$$

In the case $\beta \geq 1$ we find:

$$E_p(a) = \sum_p (\alpha dw)^\beta - N\alpha^\beta a^\beta(1 - \beta) - \sum_p \alpha^\beta \beta a^{\beta-1} dw \tag{7}$$

Now, we want to find the value for a that minimizes E_p. Let's compute the derivative and set it to 0. Since $\beta \neq 1$ and $\beta \neq 0$ we have:

$$E'_p(a) = 0$$

$$\beta N\alpha^\beta a^{\beta-1}(\beta - 1) = \alpha^\beta \beta(\beta - 1)\sum_p a^{\beta-2} dw$$

$$Na^{\beta-1} = a^{\beta-2}\sum_p dw$$

and thus:

$$a = \frac{1}{N}\sum_p dw \tag{8}$$

Similarly, for the case where $\beta < 1$, we obtain the same value $a = \frac{1}{N}\sum_p dw$.

We have just proved that over a fixed path p, the first-order Taylor approximation works best by computing it at the point $a = \frac{1}{N}\sum_p dw$. However we can't choose p (and in particular the path that minimizes C) and we can't compute the optimal a value to approximate the best solution p because p is still unknown. But it still gives a clue of the order of magnitude of the optimal value, and it should be close to mean of the gold values over the entire graph.

5.3.1 Gradient descent

I then thought that a gradient descent could help finding the optimal value for a directly inside of the algorithm, without relying on an arbitrary value.

Assuming we can find a "good enough" value $a = a_0$, we have:

$$a_{k+1} = a_k + \eta \nabla E(a_k)$$

$$a_{k+1} = a_k + \eta \left[\beta N \alpha^\beta a_k^{\beta-1}(\beta-1) - \alpha^\beta \beta(\beta-1)a_k^{\beta-2} \sum_p dw \right]$$

$$a_{k+1} = a_k + \eta \beta(\beta-1)\alpha^\beta a_k^{\beta-2} \left[N a_k - \sum_p dw \right]$$
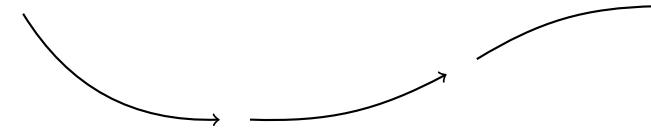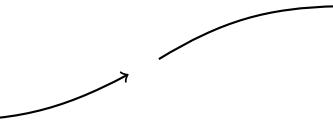
But we must now prove the convergence of this gradient descent.

Let's study the convexity of $E(a)$ (again without the absolute value for simplicity). We can write it like this: $E(a) = -x_1 a^\beta - x_2 a^{\beta-1} + x_3$ with $x_1 = N\alpha^\beta(1-\beta) > 0$ if and only if $\beta < 1$, $x_2 = \alpha^\beta \beta \sum_p dw > 0$ and $x_3 = \sum_p (\alpha dw)^\beta$. We also have $\beta \neq 1$ and $\beta \neq 0$ and we get:
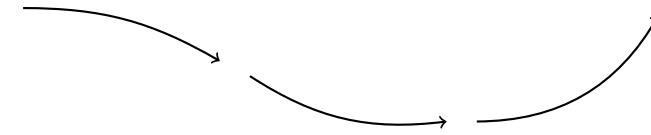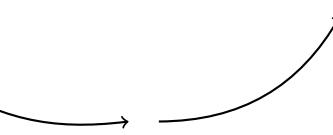
$$E(a) = -x_1 a^\beta - x_2 a^{\beta-1} + x_3$$
$$E'(a) = -\beta x_1 a^{\beta-1} - (\beta-1)x_2 a^{\beta-2}$$
$$E''(a) = -\beta(\beta-1)x_1 a^{\beta-2} - (\beta-1)(\beta-2)x_2 a^{\beta-3}$$

We then have the following variation tables:

- For $\beta < 1$:

a	0		$\dfrac{1-\beta}{\beta}\dfrac{x_2}{x_1}$		$\dfrac{2-\beta}{\beta}\dfrac{x_2}{x_1}$	
f''		$+$		$+$	0	$-$
f'		$-$	0	$+$		$+$
f						

- For $\beta > 1$:

a	0		$\dfrac{2-\beta}{\beta}\dfrac{x_2}{x_1}$		$\dfrac{1-\beta}{\beta}\dfrac{x_2}{x_1}$	
f''		$-$	0	$+$		$+$
f'		$-$		$-$	0	$+$
f						

Theorem 5.1. *The gradient descent for a converges to the global minimum of $E(a)$ for a fixed path p.*

When I implemented it, and dynamically calculating a in regard to the current best solution p; it did not work well and was unable to find better solutions than the baseline (going iteratively to each node and picking up the gold at that node to then go immediately deposit it at the depot). I then tried another approximation approach, that should be more precise than just the first-order Taylor approximation because it relied on a single value a for all of the arc and I thought that was the issue.

5.4 Multinomial approximation

Since w and d are the sums of values over the current part of the path that is traveled, this time we are not trying to linearize the function x^β at a point a but we want to use the multinomial theorem (if β is an integer for simplicity at first) and then linearize the parts that are not at the β power.

Let's note dw as only w for simplicity.

Definition 5.2 (Multinomial theorem). The multinomial theorem describes how to expand a power of a sum in terms of powers of the terms in that sum. It is the generalization of the binomial theorem from binomials to multinomials.

For any positive integer m and any non-negative integer n, the multinomial theorem describes how a sum with m terms expands when raised to the nth power:

$$(x_1 + x_2 + \cdots + x_m)^n = \sum_{\substack{k_1 + k_2 + \cdots + k_m = n \\ k_1, k_2, \cdots, k_m \geq 0}} \binom{n}{k_1, k_2, \ldots, k_m} x_1^{k_1} \cdot x_2^{k_2} \cdots x_m^{k_m} \tag{9}$$

Where:

$$\binom{n}{k_1, k_2, \ldots, k_m} = \frac{n!}{k_1! \, k_2! \cdots k_m!} \tag{10}$$

We can write the non-linear part of the cost as:

$$w^\beta = \sum_{\substack{k_1 + k_2 + \cdots + k_m = \beta \\ k_1, k_2, \cdots, k_m \geq 0}} \binom{\beta}{k_1, k_2, \ldots, k_m} w_1^{k_1} \cdot w_2^{k_2} \cdots w_m^{k_m} \tag{11}$$

using 5.2.

This formula describes the cumulative load w at any given edge along the path p. Here, $w_1, w_2, \ldots w_m$ corresponds to the individual amounts of gold picked-up at the nodes along to way, up to node m. Note that path does not need to be complete and return to the depot for the formula of the cumulative load to still work. This is useful to compute the sum in a cumulative way along the path without knowing the full path in advance.

The expansion (11) assumes β is a positive integer (for the multinomial formula to work). For non-integer β the corresponding expansion is non-finite and requires an infinite series or approximation, which is impractical for exact modeling. Even for integer β the number of mixed terms grows combinatorially with β and the route length m since the number of K-tuples is binomial. Encoding or linearizing all such mixed terms results in many auxiliary variables and dense coupling constraints, which quickly becomes infeasible for bigger instance sizes. For small and moderate instances, careful precomputation of coefficients can make the approach viable, but it does not scale well in practice.

Using (11) we can denote two distinct parts by separating the cases where one of the $k_i = \beta$ and $\forall j \neq i \; k_j = 0$.

$$w^\beta = \sum_{i=1}^{m} w_i^\beta + \sum_{\substack{\sum_{i=0}^{m} k_i = \beta \\ k_i \geq 0, \; \neg(k_i = n \land \forall j \neq i, k_j = 0)}} \binom{\beta}{k_1, k_2, \ldots, k_m} w_1^{k_1} \cdot w_2^{k_2} \cdots w_m^{k_m} \tag{12}$$

For simplicity, let's write the sum as $\sum_K$ ie. as the sum over the permutations of $K = (k_1, \ldots, k_m)$ that are in the sum as previously defined.

Using the OR-Tools solver we can successfully compute the first sum iteratively along the path by calculating each node value at the β power. The second part is harder and we will try to approximate it by it's plane (first-order Taylor approximation) at a point denoted $a = (a_1, a_2, \ldots, a_m)$, like in the previous approximation.

Let's now define the function f as $f_K(w) = \prod_{i=1}^m w_i^{k_i}$ and $\frac{\partial f_K}{\partial w_i} = k_i w_i^{k_i-1} \prod_{j=1\ i\neq j}^m w_j^{k_j}$ It's tangent plane at point a is:

$$T_a(w) = f_k(a) + \sum_{i=1}^m \left.\frac{\partial f_K}{\partial w_i}\right|_a (w_i - a_i) \tag{13}$$

Given a value for a this is by definition linear in w and should be easily computed by the solver. We then have:

$$w^\beta \approx \sum_{i=1}^m w_i^\beta + \sum_K \left[\frac{\beta!}{K!} \left(f_K(a) + \sum_{i=1}^m \left.\frac{\partial f_K}{\partial w_i}\right|_a (w_i - a_i) \right) \right] \tag{14}$$

We can rewrite equation (14) in a way that the solver can understand:

$$w^\beta \approx \sum_{i=1}^m w_i^\beta + \sum_{i=1}^m \left[(w_i - a_i) \sum_K \frac{\beta!}{K!} \left.\frac{\partial f_K}{\partial w_i}\right|_a \right] + \sum_K \frac{\beta!}{K!} f_K(a) \tag{15}$$

where $w = \sum_{1=1}^m w_i$. Note that we can again remove the last term of the cost because it's constant over the whole graph.

When $\beta \geq 1$ is an integer we can approximate the cost in (1) as follows:

$$\widetilde{c_{ij}} = d_{ij} + \alpha^\beta \cdot d_{ij}^\beta \cdot \left(w_i^\beta + (w_i - a_i) \sum_K \frac{\beta!}{K!} \left.\frac{\partial f_K}{\partial w_i}\right|_a + \sum_K \frac{\beta!}{K!} f_K(a) \right) \tag{16}$$

5.4.1 What can we say about this approximation

Given a permutation K that satisfies the requirements, if $k_i = 0$ then $\frac{\partial f_K}{\partial w_i} = 0$ by definition and

$$c_{ij} \approx d_{ij} + \alpha^\beta \cdot d_{ij}^\beta \cdot \left(w_i^\beta + \sum_K \frac{\beta!}{K!} f_K(a) \right) \tag{17}$$

Since the coefficient multiplying a_i appears with a negative sign, and the solver cannot have negative costs, let's study the following fraction for a fixed K:

$$\frac{\frac{\beta!}{K!} \left.\frac{\partial f_K}{\partial w_i}\right|_a}{\frac{\beta!}{K!} f_K(a)} = \frac{a_i k_i a_i^{k_i-1} \prod_{j=1\ i\neq j}^m a_j^{k_j}}{\prod_{i=1}^m a_i^{k_i}} = \frac{k_i \prod_{i=1}^m a_i^{k_i}}{\prod_{i=1}^m a_i^{k_i}} = k_i \geq 1 \tag{18}$$

Because we already ruled out the case $k_i = 0$. We thus have in the case $k_i = 1$:

$$c_{ij} \approx d_{ij} + \alpha^\beta \cdot d_{ij}^\beta \cdot \left(w_i^\beta + w_i \sum_K \frac{\beta!}{K!} \left.\frac{\partial f_K}{\partial w_i}\right|_a \right) \tag{19}$$

And otherwise the cost has a negative term.

5.5 Why the approximations do not work

After testing the Taylor and multinomial approximations, I could not obtain solutions better than the baseline. The issue was not primarily the algebraic approximations themselves but the solver's modeling limitations. OR-Tools finds the "Hamiltonian" path of the problem, where each city can only be visited once but, in our case, the arcs are often used multiple times. OR-Tools ends up solving a different problem than ours, which made me completely change my approach. For that reason I did not pursue optimizing the expansion point a or extending the approximations to every value of β. However, the approximations remain theoretically valid and I think they could still be useful with alternative solvers, with different modeling choices, or as part of relaxations or heuristics.

5.6 Genetic algorithm

To overcome the limitations of standard VRP solvers and the complexity of the non-linear cost function, I implemented a Genetic Algorithm (GA) combined with an optimal partitioning procedure. This approach uses the Giant Tour representation, where an individual in the population is a permutation of all cities that must be visited, and the routing is handled by a secondary optimization layer.

Because the problem is defined on a graph where vehicles follow shortest paths, the cost of moving from node u to node v with a constant weight w can be decomposed. If the shortest path consists of edges $(e_0, e_1, \ldots, e_k)$ with $e_0 = u$ and $e_k = v$, the cost is:

$$c_{uv}(w) = \sum_{i=1}^{k} d_{e_{i-1}e_i} + (\alpha w)^\beta \sum_{i=1}^{k} d_{e_{i-1}e_i}^\beta \qquad (20)$$

By precomputing the shortest path distance matrix D_{uv} and a secondary matrix $B_{uv} = \sum_{(i,j)\in P_{uv}} d_{ij}^\beta$, the cost of any segment can be computed in constant time $O(1)$ in d during the optimization process:

$$c_{uv}(w) = D_{uv} + (\alpha w)^\beta B_{uv} \qquad (21)$$

The Genetic Algorithm maintains a population of permutations (giant tours), where each individual encodes an ordering of all the nodes. The key insight is to separate the ordering of customers from the routing decisions. the GA optimizes the permutation while a deterministic split procedure (based on "A simple and effective evolutionary algorithm for the vehicle routing problem" by Christian Prins) handles the partitioning of this sequence into feasible trips. The function finds the cheapest way to split an ordered list of customers into multiple depot-based routes, where travel cost depends on both distance and the amount of gold carried. It constructs an auxiliary Directed Acyclic Graph (DAG) where an edge represents a feasible trip serving a subsequence of customers. Finding the shortest path in this DAG from the start to the end of the permutation yields the optimal set of trips (and thus the fitness cost) for that specific order. This reduces the problem complexity for the GA, as it only needs to optimize the permutation, not the route breaks.
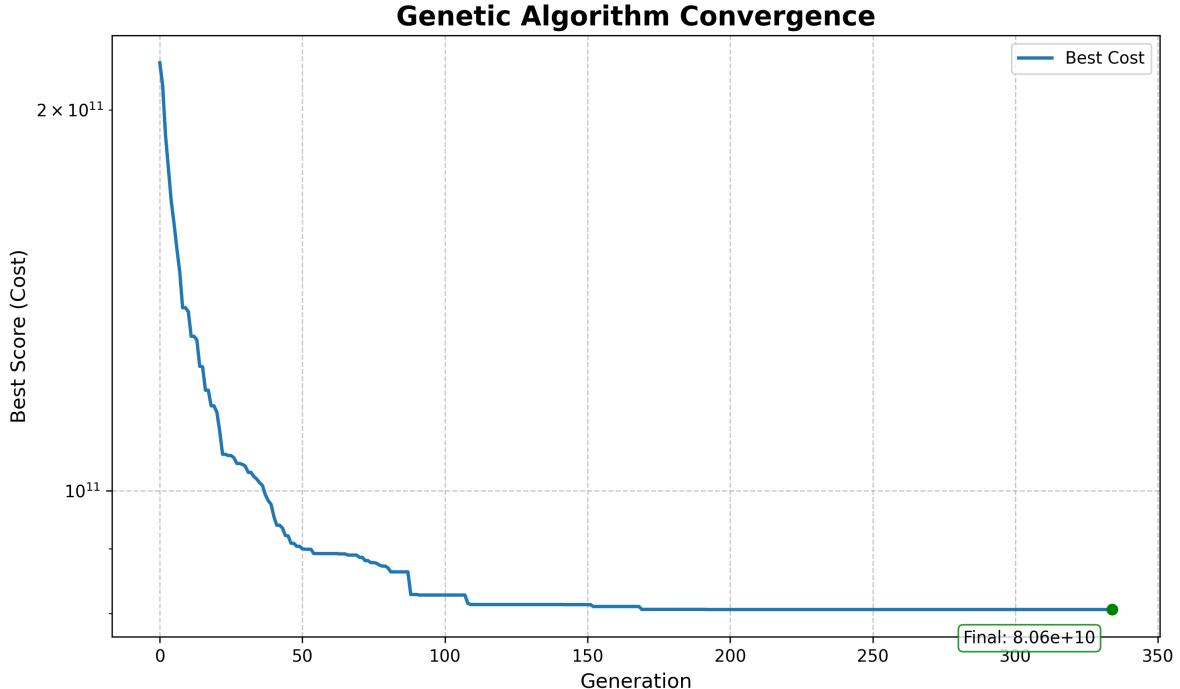

Figure 5: Example of the evolution of the best fitness score across generations in the genetic algorithm.

To maintain diversity and explore the search space effectively, I implemented two kinds of mutations:

- **Swap Mutation**: Two random cities in the permutation are selected and their positions are swapped. This is a disruptive mutation that allows the algorithm to move customers between

different routes or far-apart positions in the sequence, helping to "jump" out of local basins of attraction.

- **Ordered Crossover (OX)**: A contiguous subsequence of nodes is copied from the first parent into the offspring at the same positions. The remaining nodes positions are then filled, in order, with the customers from the second parent that do not already appear in the offspring. This operator preserves the relative order of customers from both parents while ensuring a valid permutation without duplicates.

I also used Elitism, where a small number of the best individuals (the elite) are guaranteed to survive to the next generation unchanged. The thought process behind this is stability but the genetic operators (crossover and mutation) are stochastic and destructive. They can break good solutions as easily as they create them. By forcing the survival of the best known solutions, we ensure that the algorithm's performance curve is monotonically non-decreasing (see figure 5), anchoring the search while the rest of the population explores new territories.

(a) Baseline solution
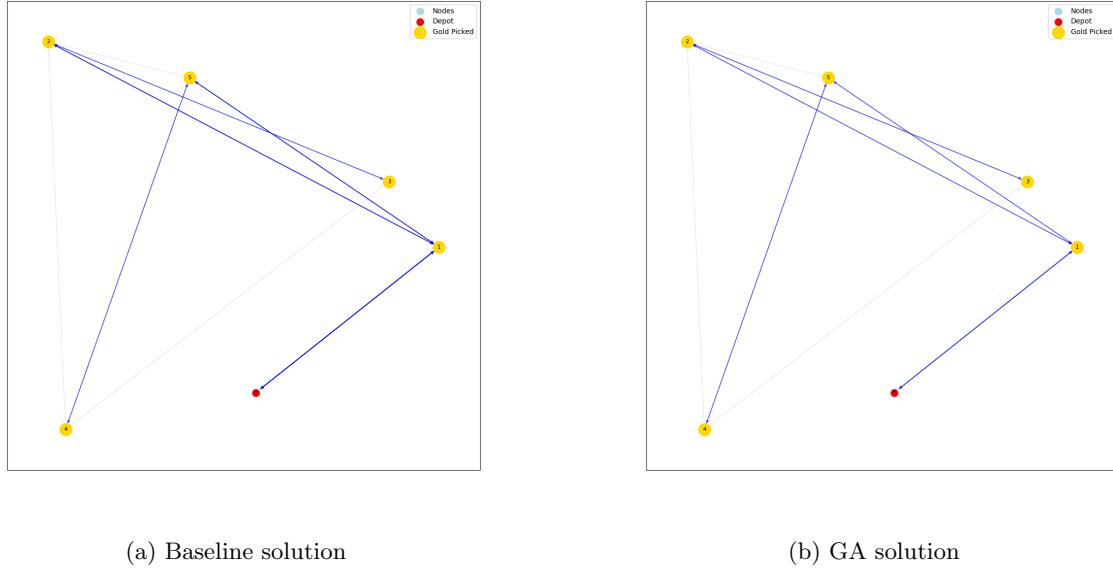
(b) GA solution

Figure 6: `Problem(num_cities=6, density=0.3, alpha=1, beta=1, seed=42)`

GA Solution	Baseline Solution
(4, 228.01)	(1, 644.22)
(5, 555.03)	(0, 0)
(1, 644.22)	(5, 555.03)
(0, 0)	(0, 0)
(3, 443.97)	(2, 822.94)
(2, 822.94)	(0, 0)
(0, 0)	(4, 228.01)
	(0, 0)
	(3, 443.97)
	(0, 0)
Total cost: GA = 2992.04, Baseline = 2997.21, Improvement = 0.17%	

Table 2: Comparison of GA and baseline solutions for the VRP instance. Depot returns are indicated by $(0, 0)$. The GA solution shows a slightly more efficient ordering, resulting in a 0.17% improvement over the baseline.

Figure 6 and Table 2 compare the baseline and Genetic Algorithm (GA) solutions. The baseline provides a reasonable route, but the GA solution tends to collect most of the gold on return trips,

producing more organized and slightly more efficient routes. Visually the routes may appear similar, but their ordering and depot-return structure differ.

5.7 Cython Parallelization

The performance bottleneck of this approach is the evaluation phase. The Split algorithm requires $O(N^2)$ operations for each individual, and the custom cost function involves expensive floating-point power calculations. Running this in pure Python for a population of 100 individuals over hundreds of generations is slow.

To address this, I parallelized this evaluation phase using Cython. I moved the entire `evaluate_population` into a compiled C++ extension. By explicitly typing the variables and using the `nogil` (No Global Interpreter Lock) context, I was able to bypass Python's single-thread limitation. I effectively parallelized the loop over the population using OpenMP (via `cython.parallel.prange`), allowing the fitness of every individual to be computed simultaneously across all available CPU cores. This optimization resulted in a massive speedup, enabling the use of larger populations and deeper searches within a reasonable time frame.