1. INTRODUCTION

The Traveling Salesman Problem (TSP) is a classical problem in combinatorial optimization, relevant in logistics, routing, and supply chain management. Since finding optimal solutions quickly becomes infeasible for large instances, heuristics and metaheuristics are widely used to obtain good solutions within reasonable time.

In this work, constructive heuristics (Nearest Neighbour, Outlier Insertion) were implemented as baselines, followed by a Greedy Randomized Adaptive Search Procedure (GRASP) combined with 2-opt local search. Additional metaheuristics, including simulated annealing and iterated local search, were tested to improve scalability.

The results show that Outlier Insertion already produces strong initial tours, while GRASP+2opt improves them further. For very large instances, Iterated Local Search proved most effective, consistently delivering better solutions with shorter runtimes.

2. METHODOLOGY

To tackle the Traveling Salesman Problem, I combined constructive heuristics with local search and metaheuristics. The approach started with simple constructive methods to generate feasible tours, and then progressively applied more advanced algorithms to improve solution quality.

Two constructive baselines were implemented: the Nearest Neighbour heuristic (tested from all starting cities) and the Outlier Insertion heuristic, which iteratively selects the most distant city and inserts it at the position that causes the smallest additional cost.

Outlier Insertion was then extended into a Greedy Randomized Adaptive Search Procedure (GRASP), which introduces randomness into both city and position selection via Restricted Candidate Lists (RCL). Each GRASP solution was further refined with a 2-opt local search.

For instances with fewer than 500 cities, GRASP + 2-opt consistently produced strong solutions. For larger instances, however, the runtime became prohibitive. To handle these cases, I experimented with alternative metaheuristics (Simulated Annealing, k-nearest neighbour 2-opt, and sampled 2-opt). These did not significantly improve the Outlier Insertion solutions, so they are included in the code but not emphasized in the results.

Iterated Local Search (ILS) proved most effective for large instances, as it perturbs the current solution and applies fast local search to escape local optima. The final implementation therefore applies:

- GRASP + 2-opt for <500 cities,
- ILS for ≥500 cities.

2.1 Nearest Neighbour Heuristic

The Nearest Neighbour (NN) heuristic constructs a tour by always selecting the closest unvisited city. Since results vary depending on the starting city, the algorithm was run from all possible starting points to compare solution quality.

2.2 Outlier Insertion

2.2.1 Cost Calculation for Insertion

The function “delta_insertion_cost” computes the additional distance caused by inserting a city between two consecutive cities in the tour. If a city *k* is placed between *a* and *b*, the original edge *(a,b)* is replaced with *(a,k)* and *(k,b)*

$$\Delta = d(a,k) + d(k,b) - d(a,b)$$

This calculation is repeated for all possible insertion positions, and the position with the smallest increase is chosen.

2.2.2 Outlier Insertion Algorithm

The Outlier Insertion heuristic grows a tour by repeatedly adding the city that is farthest from the current tour and placing it in the position that minimizes the increase in distance. The process starts with a random city paired with its farthest neighbour to form an initial two-city tour.

At each step, the next city k is chosen as:

$$k^* = \arg\max_{\{k \notin tour\}} \min_{\{t \in tour\}} d(k,t)$$

This city is inserted at the best position determined by the insertion cost function. The procedure continues until all cities are included.

While the insertion logic is deterministic, the choice of the starting city introduces variability. To keep results reproducible, a fixed random seed was used when selecting the initial city.

2.3 GRASP with Outlier Insertion

2.3.1 Restricted Candidate List (RCL) Construction

In the GRASP framework, the Restricted Candidate List (RCL) is used to balance greediness with randomness. Instead of always selecting the single best candidate, a set of good candidates is built, and one of them is chosen at random. This diversification mechanism allows the algorithm to explore different solutions and avoid being trapped in poor deterministic choices.

Two helper functions were implemented:

_build_rcl_by_quality: constructs the RCL by including all candidates within a certain quality threshold relative to the best and worst scores, controlled by the parameter α ∈ [0,1]. A smaller α results in a greedier selection, while a larger α increases randomness.

_build_rcl_by_k: constructs the RCL by selecting the top-k candidates, offering an alternative to the α-based approach.

Both functions return a set of promising candidates, from which the final choice is made randomly. This ensures that the algorithm generates diverse tours across different runs while still focusing on high-quality moves.

2.3.2 GRASPed Insertion Heuristic

The GRASPed insertion heuristic extends the deterministic outlier insertion by introducing controlled randomization in two steps of the construction process. Specifically, there are two points where randomization can be applied:

1. City selection – In the deterministic version, the next city is always the one with the maximum "nearest-to-tour" distance. Instead of deterministically picking this city, a Restricted Candidate List (RCL) can be formed consisting of the most promising candidates (e.g., based on an α-threshold or the top-k options). One city is then randomly chosen from this list.
2. Position selection – Once a city is chosen, the deterministic version always inserts it in the position that minimizes the increase in tour length. This step can also be randomized by forming an RCL of the best insertion positions and sampling one at random.

By randomizing these two steps, the heuristic generates diverse solutions across different runs, which helps avoid deterministic bias and allows exploration of different regions of the solution space. In this work, both steps were randomized, controlled by user-specified α or k values. This setup enables balancing greediness and randomness, while ensuring reproducibility when a random seed is fixed.

2.4 2-Opt Local Search

The 2-opt heuristic improves a tour by checking pairs of non-adjacent edges and reversing segments when this reduces the distance. In the implementation, a helper function computes the cost difference (delta) directly, avoiding reconstruction of the entire tour and improving efficiency. The algorithm supports two modes:

- First-improvement: the search stops as soon as a better move is found within a scan of the tour, leading to faster runtimes.
- Best-improvement: the algorithm evaluates all possible swaps in a scan and applies the best one, which is slower but may yield slightly better results.

For efficiency reasons, the default configuration was set to first-improvement, as preliminary tests showed negligible gains from best-improvement at the cost of significantly longer runtimes.

Correctness of the implementation was validated in two ways:

- Feasibility checks: after every swap, the resulting tour was verified to contain each city exactly once and to return to the starting city.
- Consistency checks: the computed delta values were cross-validated by comparing them with the actual cost difference before and after applying a swap on test instances.

This process ensured that the 2-opt method was implemented correctly and produced valid improved tours.

2.5 GRASP + 2-Opt Combination

To systematically combine the constructive power of GRASP with the refinement of 2-opt, a dedicated routine was implemented. The method repeatedly generates GRASP-based tours and applies 2-opt local search to each of them. Across multiple iterations, the algorithm keeps track of both the initial GRASP costs and the improved costs after 2-opt. This structure serves two purposes:

- Performance monitoring – It allows visualizing the improvement effect of 2-opt by plotting the costs before and after local search for all iterations.
- Best solution tracking – Since GRASP is randomized, running the procedure multiple times increases the likelihood of finding a high-quality solution. The method records the best tour and cost encountered across all iterations.

This combined approach ensured that GRASP did not only provide diverse initial solutions but also benefited from systematic local search improvements, as required in the assignment. For small and medium-sized instances (fewer than 500 cities), this method was effective and consistently improved results. However, for larger instances, runtime became prohibitive. Therefore, an if–else rule was introduced:

- If the number of cities is below 500, the algorithm applies GRASP + 2-opt.
- If the number of cities is 500 or more, the program switches to Iterated Local Search (ILS), which proved to be more efficient.

2.6 Faster Alternatives for Large Instances

For large instances (500+ cities), GRASP combined with 2-opt became too slow to be practical. To address this, several faster alternatives were implemented and tested:

- Simulated Annealing (SA): A metaheuristic that accepts worse moves with decreasing probability to escape local optima.
- makeTwoOpt_fast: A variant of 2-opt restricted to k-nearest neighbors instead of all city pairs, reducing computational cost. Although makeTwoOpt_fast did not improve solutions on its own, it proved useful inside ILS as a fast local search component.
- makeTwoOpt_sampledBest: A stochastic version of 2-opt that samples a fixed number of candidate swaps per iteration.
- Iterated Local Search (ILS): A method that perturbs the current solution and applies local search repeatedly to escape local optima.

Among these, only ILS consistently produced improvements over the Outlier Insertion tour while maintaining fast runtimes. The other methods did not provide significant gains and are therefore not actively used in the experiments, though they remain in the codebase for documentation.

3. Results

To evaluate the algorithms, ten problem instances provided in the assignment folder were selected:

- Small instances: berlin52, pr76, lin105, ulysses22, kroA100
- Medium instances: rat195, st70, a280
- Large instances: d1291, pr1002

A fixed random seed was used where applicable to ensure reproducibility. The results are reported separately for small, medium, and large instances, in order to compare solution quality across different problem sizes.

3.1 Nearest Neighbour Heuristic

The nearest neighbour heuristic was tested from all starting cities. Table 1 shows the min, max, and average costs. Results confirm that solution quality strongly depends on the starting city: in some cases (e.g., berlin52) the gap is large, while in others (e.g., ulysses22) it is small.

Table 1 Summary of NN heuristic results

	min NN	max NN	avg NN
berlin52	8182.2	10299.4	9373.4
pr76	130921	157075.9	147180
lin105	16939.4	20590.3	18719.9
ulysses22	86.9	99.2	91.4
kroA100	24698.5	28694.7	27045.2
rat195	2628.6	3022.7	2779.6
st70	761.7	901.3	825.8
a280	3094.3	3607.5	3315
d1291	59040	66277.6	61868
pr1002	311748.8	348952.9	327473.6

3.2 Outlier Insertion

Outlier Insertion (OI) was compared with the Nearest Neighbour (NN) heuristic using the same starting cities. While NN extends tours by always picking the nearest city, OI selects the farthest city and inserts it in the least costly position.

Table 2 (Under the heading 3.3) shows tour lengths for ten benchmark instances: “NN” = Nearest Neighbour, “OI” = Outlier Insertion. Across almost all cases, OI achieves shorter tours, confirming its advantage over NN. Figure 1 illustrates the OI tour for berlin52, with the starting city highlighted. Figure 2 shows the GRASP+2opt tour for berlin52.

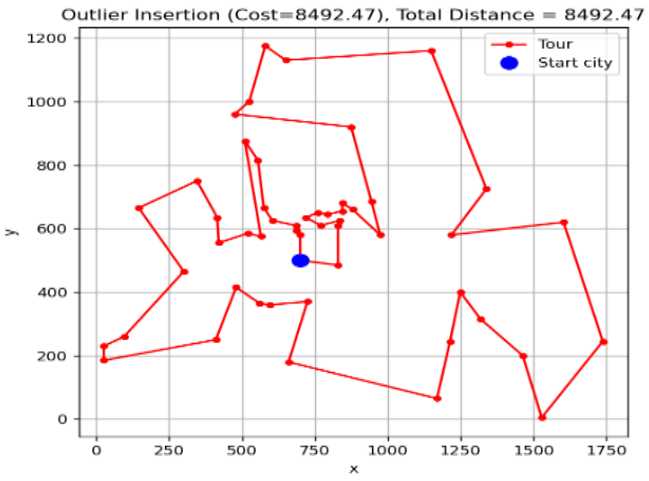

Figure 1

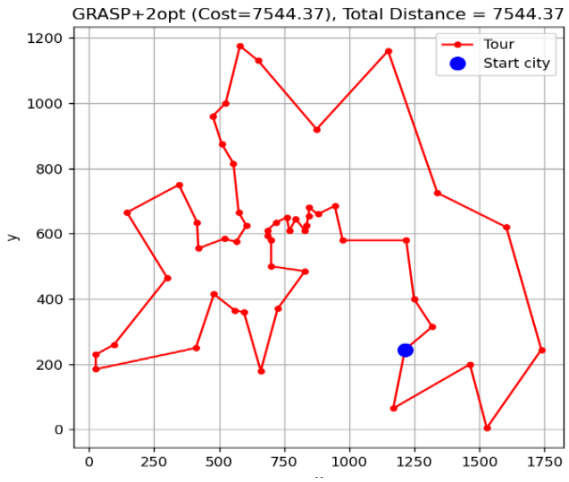

Figure 2

3.3 GRASP + 2-Opt

To further improve the Outlier Insertion tours, the Greedy Randomized Adaptive Search Procedure (GRASP) was combined with 2-opt local search. The GRASP procedure introduces

randomization in city and position selection, producing diverse initial tours, which are then refined with 2-opt.

The scatter plot below illustrates this effect for Berlin52, where each point represents the cost of a GRASP tour before and after applying 2-opt. Almost all points lie below the diagonal line, confirming consistent improvements. In addition, the figure of the resulting tour visually highlights how 2-opt eliminates crossings and shortens the path.

The following table compares the three constructive approaches – Nearest Neighbour (NN), Outlier Insertion (OI), and GRASP + 2-opt – on the selected small and medium instances. All algorithms were run with the same starting city to ensure comparability.

Table 2. Comparison of NN, OI, GRASP + 2-opt, and ILS (applied only to large instances). All methods were run with the same starting city.

	NN	OI	GRASP+2opt	ILS for >500
berlin52	9075.4	8492.2	7544.4	
pr76	151653.5	115147.4	109673.9	
lin105	17912.7	15204.7	14637.2	
ulysses22	87.8	76.3	75.3	
kroA100	27133	24497.3	21370.1	
rat195	2910.3	2676	2444.7	
st70	835.5	730	685.4	
a280	3517.8	2930.2	2729.5	
d1291	62984.4	62246.5		56239.3
pr1002	341433.6	286287.4		276966

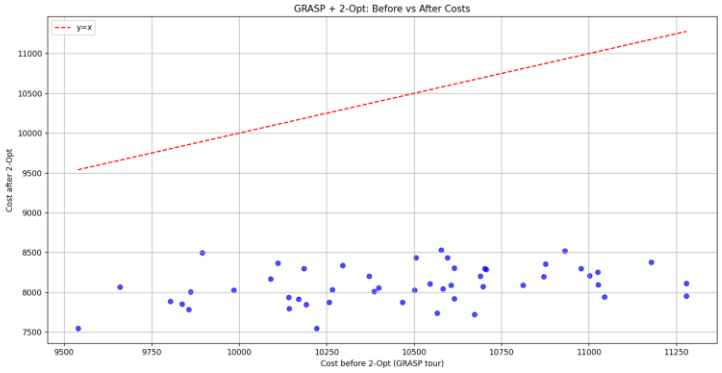

Figure 3

4. Conclusion

This study applied constructive heuristics and local search to the TSP. Nearest Neighbour and Outlier Insertion provided quick initial tours, while GRASP with 2-opt significantly improved solution quality for small and medium instances. For large instances, Iterated Local Search achieved the best balance between quality and runtime. The main limitation is scalability: GRASP+2opt becomes slow as instance size grows. Future improvements could involve more advanced neighbourhood structures or hybrid methods to further enhance efficiency.

Appendix – Python Code

```
   # -*- coding: utf-8 -*-
"""
TSP + NN + Outlier Insertion + GRASP(Insertion) + 2-opt

Author: Yusuf Sami

"""

import math
import numpy as np
import random
import matplotlib.pyplot as plt

class Point2D:
    """Class for representing a point in 2D space"""
    def __init__(self, id_, x, y):
        self.id = id_
        self.x = x
        self.y = y


    def getDistance(c1, c2):
        dx = c1.x - c2.x
        dy = c1.y - c2.y
        return math.sqrt(dx ** 2 + dy ** 2)



class TSP:
    """
    Class for representing a Traveling Salesman Problem

    Attributes
    ----------
    nCities : int
        number of cities
    cities : list[int]
        list of city indices 0..n-1
    distMatrix : np.ndarray
        symmetric n x n matrix with pairwise distances
    """

    # ------------- constructor & reading -------------
    def __init__(self, tspFileName):
        """
        Reads a .tsp file and constructs an instance.
        We assume that it is an Euclidian TSP

        Parameters
```

```
        ----------
        tspFileName : str
            name of the file
        """
        points = list()  # add all points to list
        f = open(tspFileName)
        for line in f.readlines()[6:-1]:  # start reading from line 7, skip
last line
            asList = line.split()
            floatList = list(map(float, asList))

            id = int(floatList[0]) - 1  # convert to int, subtract 1
because Python indices start from 0
            x = floatList[1]
            y = floatList[2]

            c = Point2D(id, x, y)
            points.append(c)
        f.close()

        print("Read in all points, start computing distance matrix")

        self.nCities = len(points)
        self.cities = list(range(self.nCities))
        self.points = points  # try

        # compute distance matrix, assume Euclidian TSP
        self.distMatrix = np.zeros((self.nCities, self.nCities))  # init as
nxn matrix
        for i in range(self.nCities):
            for j in range(i + 1, self.nCities):
                distItoJ = Point2D.getDistance(points[i], points[j])
                self.distMatrix[i, j] = distItoJ
                self.distMatrix[j, i] = distItoJ

        print("Finished computing distance matrix")



    # ------------- basic helpers -------------
    def getCitiesCopy(self):
        return self.cities.copy()

    def isFeasible(self,tour):
        """
        Checks if tour is feasible

        Parameters
        ----------
        tour : list of integers
            order in which cities are visited. For a 4-city TSP, an example
tour is [3, 1, 4, 2]
```

```
    Returns
    -------
    bool
        TRUE if feasible, FALSE if infeasible.

    """
    #first check if the length of the tour is correct
    if len(tour)!=self.nCities:
        print("Length of tour incorrect")
        return False
    else:
        #check if all cities in the tour
        for city in self.cities:
            if city not in tour:
                return False
    return True

def computeCosts(self, tour):
    """
    Computes the costs of a tour

    Parameters
    ----------
    tour : list of integers
        order of cities.

    Returns
    -------
    costs : int
        costs of tour.

    """
    costs = 0
    for i in range(len(tour) - 1):
        costs += self.distMatrix[tour[i], tour[i + 1]]

    # add the costs to complete the tour back to the start
    costs += self.distMatrix[tour[-1], tour[0]]
    return costs

def tour_cost(self, tour):
    return self.computeCosts(tour)

def evaluateSolution(self, tour):
    if self.isFeasible(tour):
        costs = self.computeCosts(tour)
        print("The solution is feasible with costs " + str(costs))
    else:
        print("The solution is infeasible")

# ------------- Nearest Neighbour -------------
def getTour_NN(self, start):
    """
```

```
        Performs the nearest neighbour algorithm

        Parameters
        ----------
        start : int
            starting point of the tour

        Returns
        -------
        tour : list of ints
            order in which the cities are visitied.

        """
        tour = [start]
        notInTour = self.cities.copy()
        notInTour.remove(start)


        for i in range(self.nCities - 1):
            curCity = tour[i]
            closestDist = -1  # initialize with -1
            closestCity = None  # initialize with None

            # find closest city not yet in tour
            for j in notInTour:
                dist = self.distMatrix[curCity][j]
                if dist < closestDist or closestCity is None:
                    # update the closest city and distance
                    closestDist = dist
                    closestCity = j

            tour.append(closestCity)
            notInTour.remove(closestCity)



        return tour
    # 2nd step in the assignment, use it while reporting
    def run_NN_all_starts(self):

        #Run NN from every start; return list of (start, cost) sorted by
cost.

        results = []
        for s in self.cities:
            tour = self.getTour_NN(s)
            results.append((s, self.tour_cost(tour)))
        results.sort(key=lambda x: x[1])
        return results

    # ------------- Outlier Insertion (deterministic) -------------
    def delta_insertion_cost(self, tour, city, pos):
        """
```

```
        Cost increase when inserting 'city' between tour[pos] and
tour[pos+1]
        (with wrap-around).
        """
        n = len(tour)
        a = tour[pos]
        b = tour[(pos + 1) % n]
        return (self.distMatrix[a, city] + self.distMatrix[city, b]
                - self.distMatrix[a, b])

    def getTour_OutlierInsertion(self, start):
        """
        Deterministic Outlier Insertion as described in the assignment:
        1) Start with [start] joined with farthest city from start.
        2) While unvisited exists:
           a) choose city k that maximizes nearest-to-tour distance: max_k
min_{t in tour} d(k,t)
           b) insert k in position that minimizes insertion delta
        """
        notIn = set(self.cities)
        notIn.remove(start)

        # farthest from start
        far_city = max(notIn, key=lambda j: self.distMatrix[start, j])
        tour = [start, far_city]
        notIn.remove(far_city)

        while notIn:
            # pick city that is 'furthest to any city in the tour' (i.e.,
maximize nearest-to-tour distance)
            def nearest_to_tour_dist(k):
                return min(self.distMatrix[k, t] for t in tour)
            k_star = max(notIn, key=nearest_to_tour_dist)

            # insert where it increases the length the least
            best_pos = None
            best_inc = None
            for pos in range(len(tour)):
                inc = self.delta_insertion_cost(tour, k_star, pos)
                if (best_inc is None) or (inc < best_inc):
                    best_inc = inc
                    best_pos = pos
            tour.insert(best_pos + 1, k_star)
            notIn.remove(k_star)

        return tour

    # ------------- GRASPed Outlier Insertion -------------

    def _build_rcl_by_quality(self, scored_list, alpha, maximize=True):
        """
        Build RCL by relative quality band.
        scored_list: [(item, score), ...]
```

```
        alpha in [0,1]: 0 => only best; 1 => everyone.
        maximize=True: larger score better; False: smaller better.
        Returns list of items (no scores).
        """
        if not scored_list:  # safeguard
            return []
        # sort by score (best first if maximize=True)
        scored_list = sorted(scored_list, key=lambda x: x[1],
reverse=maximize)
        best = scored_list[0][1]
        worst = scored_list[-1][1]
        if maximize:
            thr = best - alpha * (best - worst)
            rcl = [it for (it, s) in scored_list if s >= thr]
        else:
            thr = best + alpha * (worst - best)
            rcl = [it for (it, s) in scored_list if s <= thr]
        return rcl

    def _build_rcl_by_k(self, scored_list, k, maximize=True):
        """
        Build RCL by top-k (fallback or user preference).
        """
        if not scored_list:
            return []
        scored_list = sorted(scored_list, key=lambda x: x[1],
reverse=maximize)
        k = max(1, int(k))
        k = min(k, len(scored_list))
        return [it for (it, _) in scored_list[:k]]

    def getTour_GRASPedInsertion(self,
                                 start,
                                 alpha_city=0.30,
                                 alpha_pos=0.00,
                                 seed=None,
                                 rcl_city_len=None,
                                 rcl_pos_len=None):
        """
        GRASP Outlier Insertion supporting both alpha-based (quality) RCL
and top-k RCL.
        - If rcl_city_len is provided, use top-k for city selection; else
use alpha_city.
        - If rcl_pos_len is provided, use top-k for position selection;
else use alpha_pos.
        """
        import random
        if seed is not None:
            random.seed(seed)

        notIn = set(self.cities)
        notIn.remove(start)
```

```
        # start pair
        far_city = max(notIn, key=lambda j: self.distMatrix[start, j])
        tour = [start, far_city]
        notIn.remove(far_city)

        while notIn:
            # (i) city scores: score = min distance to any city in current
tour (maximize)
            city_scores = [(k, min(self.distMatrix[k, t] for t in tour))
for k in notIn]
            if rcl_city_len is not None:
                rcl_cities = self._build_rcl_by_k(city_scores,
rcl_city_len, maximize=True)
            else:
                rcl_cities = self._build_rcl_by_quality(city_scores,
alpha_city, maximize=True)
            k_star = random.choice(rcl_cities)

            # (ii) position scores: score = delta increase (minimize)
            pos_scores = [(pos, self.delta_insertion_cost(tour, k_star,
pos)) for pos in range(len(tour))]
            if rcl_pos_len is not None:
                rcl_pos = self._build_rcl_by_k(pos_scores, rcl_pos_len,
maximize=False)
            else:
                rcl_pos = self._build_rcl_by_quality(pos_scores, alpha_pos,
maximize=False)
            pos_star = random.choice(rcl_pos)

            tour.insert(pos_star + 1, k_star)
            notIn.remove(k_star)

        return tour

    # ------------- 2-opt -------------
    def _two_opt_gain(self, tour, i, j):
        """
        Replace edges (i,i+1) and (j,j+1) by (i,j) and (i+1,j+1); return
delta (after - before).
        Negative delta ⇒ improvement.
        """
        n = len(tour)
        a, b = tour[i], tour[(i + 1) % n]
        c, d = tour[j], tour[(j + 1) % n]
        before = self.distMatrix[a, b] + self.distMatrix[c, d]
        after = self.distMatrix[a, c] + self.distMatrix[b, d]
        return after - before

    def isTwoOpt(self, tour):
        """Return True iff no improving 2-opt move exists."""
        n = len(tour)
        for i in range(n):
            for j in range(i + 2, n):
```

```
                if i == 0 and j == n - 1:
                    continue  # adjacent edges (wrap) not allowed
                if self._two_opt_gain(tour, i, j) < -1e-12:
                    return False
        return True

    def makeTwoOpt(self, tour, first_improvement=False):
        """
        Apply 2-opt until no improving move remains.
        If first_improvement=True, stop at first found improvement per scan
(faster).
        If False, do best-improvement per scan (slower but sometimes
better).
        """
        n = len(tour)
        tour = tour[:]  # work on a copy
        improved = True
        while improved:
            improved = False
            best_gain = 0.0
            best_move = None
            for i in range(n):
                for j in range(i + 2, n):
                    if i == 0 and j == n - 1:
                        continue
                    gain = self._two_opt_gain(tour, i, j)
                    if gain < -1e-12:
                        if first_improvement:
                            # reverse segment (i+1 .. j)
                            tour[i+1:j+1] = reversed(tour[i+1:j+1])
                            improved = True
                            break
                        else:
                            if gain < best_gain:
                                best_gain = gain
                                best_move = (i, j)
                if improved and first_improvement:
                    break
            if (not first_improvement) and best_move is not None:
                i, j = best_move
                tour[i+1:j+1] = reversed(tour[i+1:j+1])
                improved = True
        return tour


    # ------------- GRASP + 2-opt runner -------------
    def run_grasp_with_twoopt(
        self,
        n_iter=50,
        alpha_city=0.30,
        alpha_pos=0.00,
        seed=0,
        start=None,
```

```
        rcl_city_len=None,
        rcl_pos_len=None,
        first_improvement=True,
    ):
        """
        Run GRASP(Insertion) n_iter times; 2-opt each solution; collect
(before, after) costs.
        Returns: best_tour, best_cost, xs_before, ys_after
        """
        rng = random.Random(seed)
        xs_before, ys_after = [], []
        best_tour, best_cost = None, float('inf')

        for _ in range(n_iter):
            s = rng.choice(self.cities) if start is None else start
            tour0 = self.getTour_GRASPedInsertion(
                s,
                alpha_city=alpha_city,
                alpha_pos=alpha_pos,
                seed=rng.randint(0, 10**9),
                rcl_city_len=rcl_city_len,
                rcl_pos_len=rcl_pos_len,
            )
            cost0 = self.tour_cost(tour0)
            tour1 = self.makeTwoOpt(tour0,
first_improvement=first_improvement)
            cost1 = self.tour_cost(tour1)

            xs_before.append(cost0)
            ys_after.append(cost1)

            if cost1 < best_cost:
                best_cost = cost1
                best_tour = tour1

        return best_tour, best_cost, xs_before, ys_after


    # --------- Simulated Annealing -------------
    # Simulated Annealing (SA) was tested as an alternative for large
instances
    # where 2-opt is too slow. However, it failed to improve the Outlier
Insertion
    # solution consistently and is therefore not used in the final
experiments.

    def simulated_annealing(
            self,
            tour,
            initial_temp: float = 5000.0,
            cooling_rate: float = 0.99,
            max_iter: int | None = None,
            min_temp: float = 0.00000001,
```

```
            seed: int | None = None,
            stagnation_limit: int | None = None,
    ):
        """
        Simulated Annealing (SA) using 2-opt neighborhood (safe mode:
recompute costs instead of using delta).
        """

        import math, random

        n = len(tour)
        if max_iter is None:
            max_iter = 1500000 * n

        rng = random.Random(seed)

        # Current solution and its cost
        current = tour[:]
        current_cost = self.tour_cost(current)

        # Best solution found so far
        best = current[:]
        best_cost = current_cost

        # Initial temperature
        T = float(initial_temp)
        it = 0
        no_improve_moves = 0

        while it < max_iter and T > min_temp and (stagnation_limit is None
or no_improve_moves < stagnation_limit):

            # Pick random 2-opt move
            i = rng.randrange(0, n - 1)
            j = rng.randrange(i + 1, n)

            # Skip invalid moves
            if (i == 0 and j == n - 1) or (j == i + 1):
                continue

            # Make a copy of current tour and apply move
            new_tour = current[:]
            new_tour[i + 1:j + 1] = list(reversed(new_tour[i + 1:j + 1]))

            # Compute new cost from scratch
            new_cost = self.tour_cost(new_tour)
            delta = new_cost - current_cost

            # Acceptance rule
            accept = False
            if delta < 0:
                accept = True
            else:
```

```
                if rng.random() < math.exp(-delta / T):
                    accept = True

            if accept:
                current = new_tour
                current_cost = new_cost

                if current_cost < best_cost:
                    best = current[:]
                    best_cost = current_cost
                    no_improve_moves = 0
                else:
                    no_improve_moves += 1
            else:
                no_improve_moves += 1

            # Cool down
            T *= cooling_rate
            it += 1

            # Debug log
            if it < 100:  # first 100 iterations
                print(f"iter={it}, T={T:.2f},
current_cost={current_cost:.2f}, best_cost={best_cost:.2f}")
            elif it % 1000 == 0:  # then every 1000 iterations
                print(f"iter={it}, T={T:.2f},
current_cost={current_cost:.2f}, best_cost={best_cost:.2f}")

        return best

    def makeTwoOpt_fast(self, tour, k=20, max_iter=1000):
        """
        Fast 2-opt using k-nearest neighbors instead of all pairs.

        Parameters
        ----------
        tour : list[int]
            Initial tour to improve.
        k : int
            Number of nearest neighbors to check for each city.
        max_iter : int
            Maximum number of improvement iterations.

        Returns
        -------
        list[int]
            Improved tour.
        """
        n = len(tour)
        tour = tour[:]  # copy
        improved = True
        it = 0
```

```
        # Precompute k nearest neighbors for each city
        neighbors = []
        for i in range(n):
            dists = [(j, self.distMatrix[i, j]) for j in range(n) if j !=
i]
            dists.sort(key=lambda x: x[1])
            neighbors.append([j for j, _ in dists[:k]])

        while improved and it < max_iter:
            improved = False
            it += 1
            for i in range(n):
                for j in neighbors[tour[i]]:
                    j_idx = tour.index(j)
                    if abs(i - j_idx) <= 1 or (i == 0 and j_idx == n - 1):
                        continue
                    delta = self._two_opt_gain(tour, i, j_idx)
                    if delta < -1e-12:
                        tour[i + 1:j_idx + 1] = reversed(tour[i + 1:j_idx +
1])
                        improved = True
                        break
                if improved:
                    break
        return tour

    def makeTwoOpt_sampledBest(self, tour, k=200, max_rounds=1000,
seed=None):
        """
        Random-sampled best-improvement 2-Opt.
        - In each round, generate k random (i,j) pairs.
        - Select the best improving move (lowest delta).
        - Apply it if it improves the tour; else stop.
        - Stops after max_rounds iterations or no improvement.

        Parameters
        ----------
        tour : list[int]
            Initial tour.
        k : int
            Number of random neighbors to sample per round (default=200).
        max_rounds : int
            Maximum number of rounds (default=1000).
        seed : int or None
            Random seed for reproducibility.

        Returns
        -------
        list[int]
            Locally improved tour.
        """
        import random
        rng = random.Random(seed)
```

```
        n = len(tour)
        tour = tour[:]  # copy
        current_cost = self.tour_cost(tour)

        for _ in range(max_rounds):
            best_delta = 0
            best_move = None

            # sample k random neighbors
            for __ in range(k):
                i = rng.randrange(0, n - 1)
                j = rng.randrange(i + 1, n)
                if (i == 0 and j == n - 1) or (j == i + 1):
                    continue
                delta = self._two_opt_gain(tour, i, j)
                if delta < best_delta:
                    best_delta = delta
                    best_move = (i, j)

            # if improvement found, apply best move
            if best_move is not None:
                i, j = best_move
                tour[i + 1:j + 1] = reversed(tour[i + 1:j + 1])
                current_cost += best_delta
            else:
                break  # no improvement in this round

        return tour


    def iterated_local_search(self, tour, n_iter=500, perturb_size=5, k=50,
seed=None):
        """
        Iterated Local Search (ILS).
        - Start from given tour (e.g. Outlier Insertion).
        - Repeat: perturb, then improve with fast 2-opt.
        - Keep best solution found.

        Parameters
        ----------
        tour : list[int]
            Initial tour (e.g., from Outlier Insertion).
        n_iter : int
            Number of perturbation + local search iterations.
        perturb_size : int
            Number of consecutive cities to shuffle in perturbation.
        k : int
            Number of nearest neighbors checked in fast 2-opt.
        seed : int or None
            Random seed.

        Returns
        -------
```

```
        best_tour : list[int]
            Best tour found.
        best_cost : float
            Cost of best tour.
        """
        import random
        rng = random.Random(seed)

        # Initial tour and cost
        current = tour[:]
        current_cost = self.tour_cost(current)
        best = current[:]
        best_cost = current_cost

        for it in range(n_iter):
            # --- Perturbation: shuffle a small segment ---
            perturbed = current[:]
            start = rng.randrange(0, len(perturbed) - perturb_size)
            segment = perturbed[start:start + perturb_size]
            rng.shuffle(segment)
            perturbed[start:start + perturb_size] = segment

            # --- Local search: fast 2-opt ---
            improved = self.makeTwoOpt_fast(perturbed, k=k, max_iter=200)
            improved_cost = self.tour_cost(improved)

            # --- Acceptance: if better, keep it ---
            if improved_cost < best_cost:
                best = improved
                best_cost = improved_cost
                current = improved
                current_cost = improved_cost
                print(f"Iter {it}: New best = {best_cost:.2f}")
            else:
                # Optionally accept even if not better (to escape local
optima)
                current = improved
                current_cost = improved_cost

        return best, best_cost

    def plot_tour(self, tour, cost=None, title="TSP Tour"):
        """
        Plot a TSP tour using stored point coordinates.
        """
        xs = [self.points[i].x for i in tour] + [self.points[tour[0]].x]
        ys = [self.points[i].y for i in tour] + [self.points[tour[0]].y]

        import matplotlib.pyplot as plt
        plt.figure(figsize=(6, 6))
        # Normal turu çiz (kırmızı çizgi ve küçük noktalar)
        plt.plot(xs, ys, 'r-', marker='o', markersize=4, label="Tour")
```

```
        # Başlangıç şehrini mavi ve büyük göster
        start_x, start_y = self.points[tour[0]].x, self.points[tour[0]].y
        plt.plot(start_x, start_y, 'bo', markersize=10, label="Start city")

        if cost is None:
            cost = self.tour_cost(tour)
        plt.title(f"{title}, Total Distance = {cost:.2f}")
        plt.xlabel("x")
        plt.ylabel("y")
        plt.grid(True)
        plt.legend()
        plt.show()


# ----------------- RUN -----------------
if __name__ == "__main__":

    instFilename = "Instances/Small/berlin52.tsp"
    inst = TSP(instFilename)

    # Nearest Neighbor
    random.seed(3899)
    startPointNN = random.choice(inst.cities)
    tour_nn = inst.getTour_NN(startPointNN)
    print("NN cost:", inst.tour_cost(tour_nn))

    # # Outlier Insertion
    tour_oi = inst.getTour_OutlierInsertion(startPointNN)
    print("Outlier Insertion cost:", inst.tour_cost(tour_oi))
    # Plot Outlier Insertion tour
    inst.plot_tour(tour_oi, title=f"Outlier Insertion
(Cost={inst.tour_cost(tour_oi):.2f})") # plot the tour

    # GRASP + 2-opt demo (quality-based RCLs)
    if inst.nCities <= 400:
        best_tour, best_cost, xs, ys = inst.run_grasp_with_twoopt(
            n_iter=50, alpha_city=0.30, alpha_pos=0.10, seed=125
        )
        print("Best GRASP+2opt cost:", best_cost)
        inst.plot_tour(best_tour, title=f"GRASP+2opt
(Cost={best_cost:.2f})") # plot the tour
        # Scatter plot: before (x) vs after (y)
        plt.figure(figsize=(6, 6))
        plt.scatter(xs, ys, color='blue', alpha=0.7)
        plt.plot([min(xs), max(xs)], [min(xs), max(xs)], color='red',
linestyle='--', label="y=x")
        plt.xlabel("Cost before 2-Opt (GRASP tour)")
        plt.ylabel("Cost after 2-Opt")
        plt.title("GRASP + 2-Opt: Before vs After Costs")
        plt.legend()
        plt.grid(True)
        plt.show()
    else:
```

```
        # Large instance
        start_tour = inst.getTour_OutlierInsertion(0)
        best_tour, best_cost = inst.iterated_local_search(start_tour,
n_iter=200, perturb_size=5, k=50, seed=42)
        print("Best ILS cost:", best_cost)
        inst.plot_tour(best_tour, title=f"ILS (Cost={best_cost:.2f})") #
plot the tour

#---------------- Plot GRASP + 2-Opt results -----------------

#----------------- NN from all starting positions -----------------
nn_results = inst.run_NN_all_starts()  # returns list of (start_city, cost)
# for start, cost in nn_results:
#     print(f"Start city {start}: NN tour cost = {cost}")

# Optional: minimum, maximum, average
costs = [cost for _, cost in nn_results]
print("NN costs summary:")
print(f"Min: {min(costs)}, Max: {max(costs)}, Avg:
{sum(costs)/len(costs):.2f}")
```