Zakaria Majidi and Arul Dhar 12/12/2025

ISEN 320 Project

1. Formulate the TSP as a Binary Integer Program

First each city can be designated as a node and we can say that the set of all the cities is V = [1,2,3,....n] with the first city on the route designated as starting node $s \in V$. For each distinct ordered pair of cities $i,j \in V$, let d_{ij} be the distance from city i to city j.
We can set X_{ij} to be a binary variable that equals 1 if the route goes from node i to node j and 0 if otherwise. This is shown below:

$$x_{ij} = \left\{\frac{1, \textit{if the route travels from city i to city j}}{0, \textit{otherwise}}\right\}$$

The main goal of the TSP is to minimize the total distance of travel. This can be depicted by the following function:

$$\text{Minimize } Z = \sum_{i \in V} \sum_{j \in V, j \neq i} d_{ij}x_{ij}$$

But this function is not complete because additional constraints are required in order to meet the other requirements of the TSP problem. First step is to write a constraint to accommodate for the salesman leaving each city exactly once. This can be depicted as:

$$\sum_{j \in V, j \neq i} x_{ij} = 1 \quad \forall i \in V$$

Along the same idea, each city must also be entered exactly once. This is depicted as:

$$\sum_{i \in V, i \neq j} x_{ij} = 1 \quad \forall j \in V$$

At this stage we now have the main objective function minimizing the distance of the entire route and constraints added to make sure that each city is entered and exited exactly once but they do not currently prevent disconnected subroutes. So the next step is to make sure that the route does not skip any cities and that we only locate routes that include all cities. This is depicted as:
$u_i - u_j + nx_{ij} \leq n - 1 \quad \forall i \neq j, i, j \in \{2,\dots,n\}$ where u_i are continuous variables

To enforce binary condition, we need to apply final constraints depicted as:
$x_{ij} \in \{0, 1\} \quad \forall i, j \in V, i \neq j$

We now have a fully formulated binary integer problem that focuses on our main goal along with all the necessary constraints.

2. Write out the branch-and-bound approach to solving the TSP exactly

- This method begins by solving a variation of the TSP that allows the x_{ij} variables to be relaxed to take values that lie between 0 and 1. Once we find the solution to this relaxed variation of the TSP, we will have found the lower bound of the optimal route length.

- The next stage of the method is performed by selecting a fractional value and forming branches of the decision variables: one in which $x_{ij} = 1$ (include this edge in the ongoing branch solutions) and another in which $x_{ij} = 0$ (exclude this edge from the branch).

- Solve each branch independently to get a lower bound value.

- Continue this process and for each new branch solve to find the lower bound within that branch.

- If the lower bound of a branch exceeds the current best upper bound, that branch is then pruned off since it cannot have an optimal solution. If any branch gives a possible integer solution, the upper bound is updated.

- This method of branching, bounding and pruning is repeated until every branch has either been solved or removed. By the end, the final best possible solution is the final optimal answer.

3. **Heuristic Design for the TSP (Sources: https://en.wikipedia.org/wiki/2-opt, https://slowandsteadybrain.medium.com/traveling-salesman-problem-ce78187cf1f3)**

For the heuristic, we focused on a two-stage approach:

1. Constructive heuristic to quickly build a feasible tour.

2. Local improvement heuristic to refine that tour.

We chose a Nearest Neighbor (NN) construction heuristic combined with a 2-opt local search. Both are standard TSP heuristics, which we implemented in this scenario.

Nearest Neighbor Construction Heuristic

Starting from some initial city, repeatedly go to the closest unvisited city until all cities are visited, then return to the start. This guarantees a feasible tour but not necessarily a good one.

We store the cities in a list, which we read from the file ‘usa13509.tsp’:

- cities[i] = (x_i, y_i) where x_i and y_i are the x and y coordinates of city i.

Our implementation of the nearest neighbor function (in Python) is:

- Input: cities (list of (x, y)); start (index of starting city).
- Data structures:
    1. visited: Boolean array of length n (number of cities).
    2. tour: list of city indices in visiting order.
- For each step:
    1. Let current be the last city in tour.
    2. Scan all cities j that are not yet visited.
    3. Compute Euclidean distance with the formula provided to find the least distance between current city and city j.
    4. Append that city to tour, mark it visited, and add d to total_dist.
- After all cities are visited, close the tour by returning to the start city and add that final distance to total_dist.

This implementation runs in O(n^2) time because for each of the n steps we scan up to O(n) candidates. On the usa13509.tsp instance (13,509 cities), a single NN run from one starting city took about 15 seconds in Python, which is consistent with the theoretical complexity.

Multi-Start Strategy and Starting City Selection

A NN tour depends heavily on the starting city. Running NN once from an arbitrary start can produce a poor tour. Ideally, we could run NN from every possible starting city (13,509 starts), but this would take forever:

- Rough estimate:
  ~15 seconds per NN run × 13,509 starts ≈ several days of CPU time.

Instead, we developed a diverse multi-start strategy designed to get a good spread of starting points:

1. Compute arrays:
    - xs = [x_i] and ys = [y_i] for all cities.
2. Identify four extreme cities:
    - idx_min_x = index with smallest x-coordinate.
    - idx_max_x = index with largest x-coordinate.
    - idx_min_y = index with smallest y-coordinate.
    - idx_max_y = index with largest y-coordinate.
3. Use these four extremes as initial start cities.
4. For each extreme city e:
    - Compute the squared distance to every other city (saves time since sqrt doesn't matter when ordering distances)
    - Sort cities in descending order of distance.
    - Select the 25 farthest cities from that extreme, skipping any that were already selected as starts.

Since squared distance preserves ordering, we avoided using time-costly square roots in this step and used just dx^2 + dy^2 instead. This gave us up to:

- 4 extreme starts + 4 × 25 farthest = 104 distinct starting cities.

We then ran the NN heuristic from each of these 104 starting cities, recorded the resulting tour and tour length in an output file, and selected the best few NN tours (smallest distance) for further improvement.

This approach balances:

- Diversity (extreme/farthest points cover different regions)
- Runtime (104 starts is manageable compared to 13,509 starts)

Local Improvement via 2-Opt

To improve the NN tours, we implemented the 2-opt local search.

Given a current tour:

- A 2-opt move:
    - removes edges (A,B) and (C,D),
    - adds edges (A,C) and (B,D),
    - reverses tour in between the two edges

The change in tour length from this move is defined by adding the Euclidean distances between the old edges, and subtracting that from the sum of the distances of the new edges.

If the delta (change) of this process is below 0 or below the previous best delta, then we know the move improved the tour.

Best-Improvement 2-Opt Algorithm

Our 2-opt implementation takes as input:

- The list of city coordinates cities,
- A feasible tour (from NN),
- The current tour length (from NN).

1. Set total_length = tour_length.
2. Repeat:
    - Initialize best_delta = 0 and best_i = best_j = -1.
    - For all pairs (i, j) with 1 ≤ i < j < n and j != i + 1:
        - Let A = tour[i - 1], B = tour[i], C = tour[j], D = tour[(j + 1) % n].
        - Compute old_len = dist(A,B) + dist(C,D) and new_len = dist(A,C) + dist(B,D).
        - Let delta = new_len - old_len.
        - If delta < best_delta, record this as the best improving move.

    - If best_delta < 0:
        - Reverse tour from best_i to best_j in place to apply the 2-opt move.
        - Update total_length += best_delta.
    - Otherwise, stop; no improving 2-opt move remains.

Our initial version had no stopping cap other than no improving moves left, and on the full 13,509-city instance, the code ran for hours with no clear end in sight. To address this, we experimented with a first-improvement variant, where we accept the first improving move we encounter and immediately restart the search. This version was extremely fast (multiple passes in a few seconds), but when we pushed it harder (up to 100 passes) it converged to tours that were noticeably worse than those obtained by running only 3 passes of best-improvement from the same starting tour. Based on these observations, we reverted to best-improvement, but introduced a max_passes parameter to cap the number of full best-improvement sweeps. This preserves the higher solution quality of best-improvement while giving us explicit control over runtime. For the final solution, we applied best-improvement 2-opt with a fixed max_passes of 5 passes for the top 10 NN tours with the shortest distance. This configuration gave us a strong balance of shorter routes (more optimal solution) and runtime.

We also considered avoiding repeated sqrt calls by using squared distances for comparisons. However, since we are actually updating the true tour length, we kept explicit Euclidean distances in the local search and used squared distances only where ordering mattered (e.g., when choosing farthest cities for starting points).

Since 2-opt is O(n^2) per pass, and typically requires several passes to converge, we were careful to apply it only to a small number of NN tours (just the best ones), not all 104, to keep runtime reasonable.

4. Comparison: Branch-and-Bound vs. Heuristic

- Branch-and-Bound (B&B) is an exact algorithm:
    - It systematically explores subsets of tours, using lower bounds to cut off branches.
    - It guarantees finding the optimal tour, but at the cost of exponential time in the worst case.
- Our heuristic (NN + multi-start + 2-opt) is approximate:

- It does not guarantee the optimal solution.
- It aims to find a high-quality tour in a much shorter, more predictable time, especially for large instances.

Branch-and-Bound:

- For a TSP with n cities, the worst-case search tree can be on the order of O(n!) nodes.
- Each iteration is:
    - exploring a node
    - computing a bound
    - either branching or ending
- Even with strong bounds, for large n (13,509), B&B is infeasible in practice. Exact methods are typically used only for small or moderate-sized instances.

Nearest Neighbor Heuristic:

- Each NN run performs about n^2/2 distance comparisons
- For usa13509.tsp, this is about 10^8 comparisons per run
- For 104 starting cities, we theoretically perform about 104 × n^2/2 comparisons. In practice, this was still manageable on our computer (took about 25 minutes).

2-Opt Local Search:

- One full best-improvement pass considers all valid (i,j) pairs:
    - Roughly n^2/2 moves per pass.
- If it takes P passes to converge, the complexity is on the order of O(P*n^2)
- In practice, P is relatively small; we expect a small number of passes (often between 3 and 10) before reaching a 2-opt local minimum.

- Because each candidate move involves computing only four distances (and we update only when improvement is found), 2-opt is significantly more time efficient than B&B for large n, but still much heavier than a single NN run.

Average vs. Worst Case:

- B&B worst case: exponential time
- B&B average case: highly problem-dependent; with good branching rules and bounds it can be much better, but for very large instances like our problem, even the average case is not practically solvable.
- Heuristic worst case:
  - NN: O(n^2) comparisons.
  - 2-opt: O(P*n^2) for relatively small P.
- Heuristic average case:
  - Similar order, but often with fewer passes and early stopping (e.g., when improvements become very small).

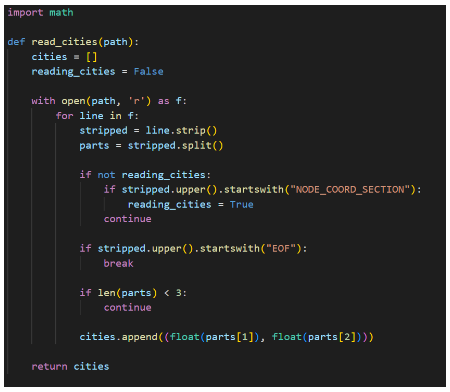
Thus, for this large real-world instance, our heuristic is implementable while B&B is not. For small instances, B&B can find exact solutions and can be used to benchmark how far our heuristic is from optimal.

In conclusion, you should use Branch and Bound when sample size n is small and when the most optimal solution is needed. Our heuristic should be used when n is very large, an approximate solution is acceptable, and runtime is more precious than exact optimality.

5. Programming Implementation of Algorithm (in Python)

nearest_neighbhor.py

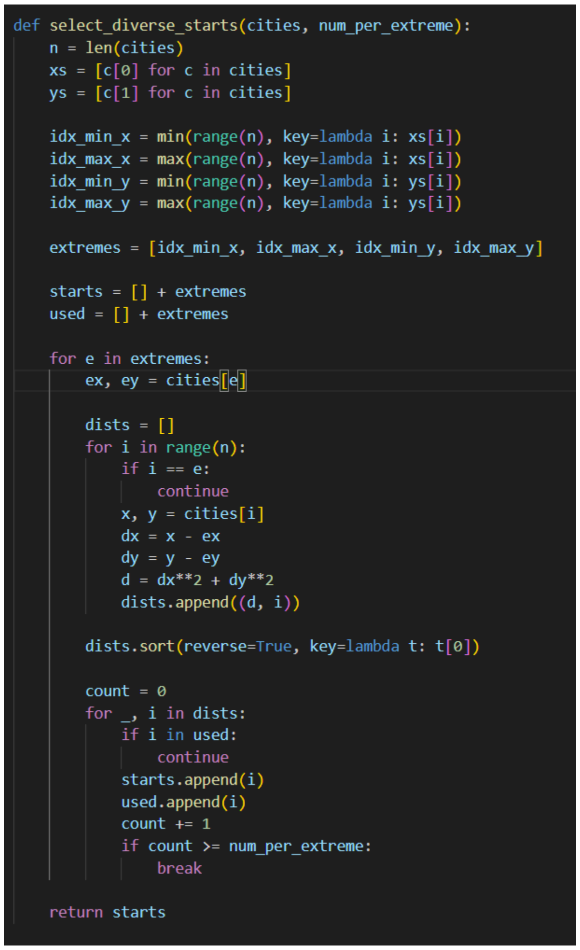
Function “read_cities”: Takes as input the .tsp file, reads the contents, and returns a cities list that stores the x and y coordinates of each city (city denoted by position in the list)



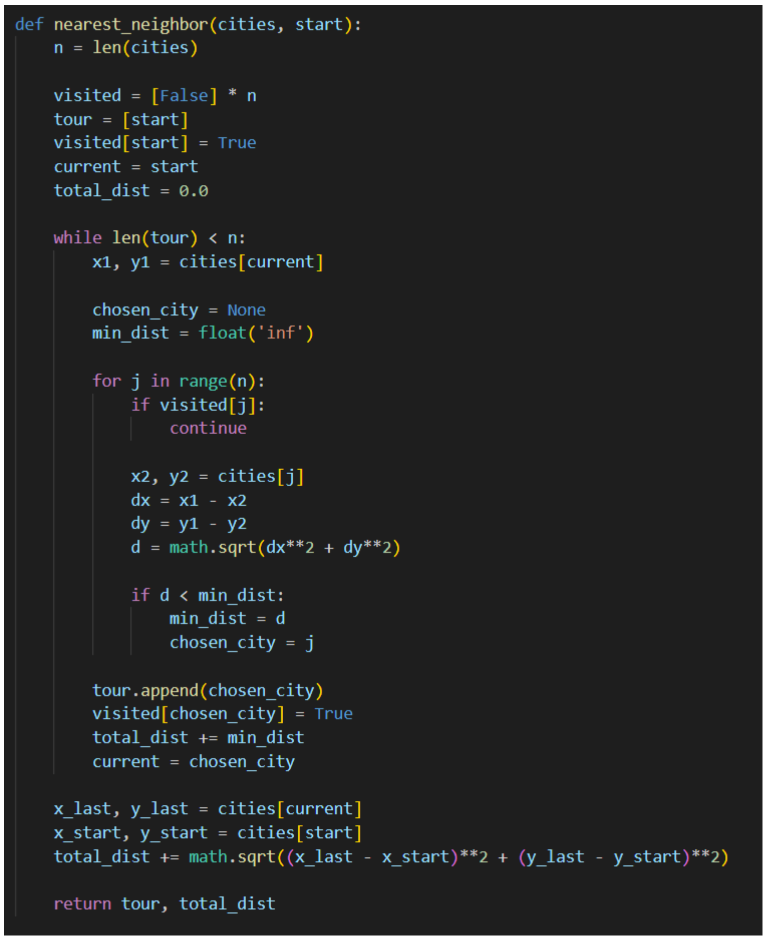
```python
import math

def read_cities(path):
    cities = []
    reading_cities = False

    with open(path, 'r') as f:
        for line in f:
            stripped = line.strip()
            parts = stripped.split()

            if not reading_cities:
                if stripped.upper().startswith("NODE_COORD_SECTION"):
                    reading_cities = True
                continue

            if stripped.upper().startswith("EOF"):
                break

            if len(parts) < 3:
                continue

            cities.append((float(parts[1]), float(parts[2])))

    return cities
```

Function “select_diverse_starts”: Takes as input cities list and number of starting points to find for each extreme, and returns a list of chosen starting points for the NN tours

Function “nearest_neighbor”: Takes as input cities list and starting index for the tour, and returns a NN tour along with its total distance

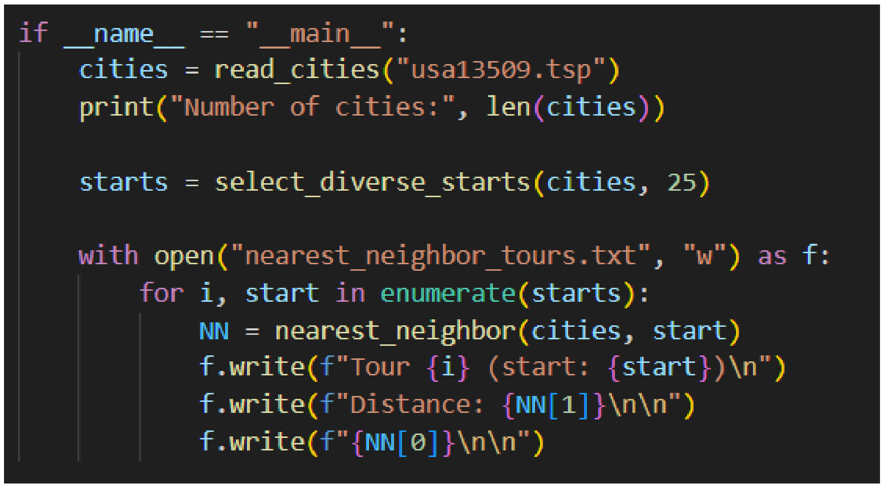


```python
def select_diverse_starts(cities, num_per_extreme):
    n = len(cities)
    xs = [c[0] for c in cities]
    ys = [c[1] for c in cities]

    idx_min_x = min(range(n), key=lambda i: xs[i])
    idx_max_x = max(range(n), key=lambda i: xs[i])
    idx_min_y = min(range(n), key=lambda i: ys[i])
    idx_max_y = max(range(n), key=lambda i: ys[i])

    extremes = [idx_min_x, idx_max_x, idx_min_y, idx_max_y]

    starts = [] + extremes
    used = [] + extremes

    for e in extremes:
        ex, ey = cities[e]

        dists = []
        for i in range(n):
            if i == e:
                continue
            x, y = cities[i]
            dx = x - ex
            dy = y - ey
            d = dx**2 + dy**2
            dists.append((d, i))

        dists.sort(reverse=True, key=lambda t: t[0])

        count = 0
        for _, i in dists:
            if i in used:
                continue
            starts.append(i)
            used.append(i)
            count += 1
            if count >= num_per_extreme:
                break

    return starts
```



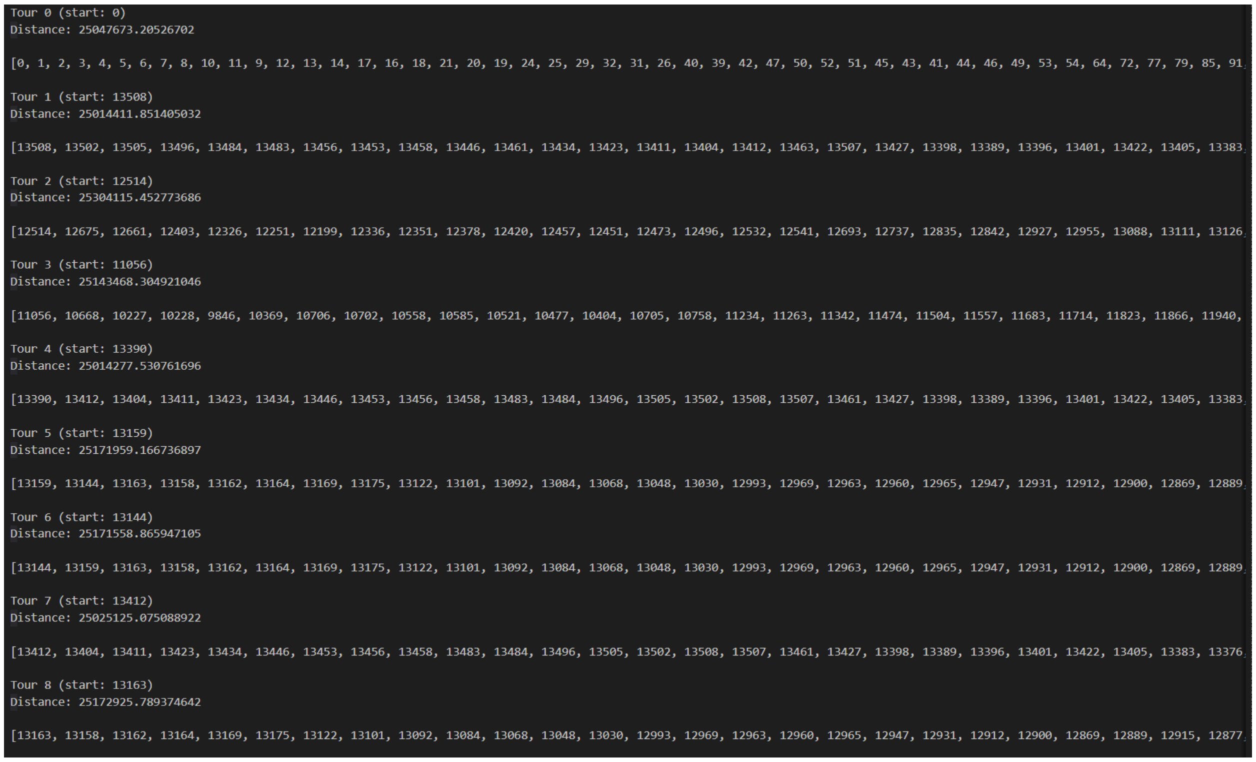
```python
def nearest_neighbor(cities, start):
    n = len(cities)

    visited = [False] * n
    tour = [start]
    visited[start] = True
    current = start
    total_dist = 0.0

    while len(tour) < n:
        x1, y1 = cities[current]

        chosen_city = None
        min_dist = float('inf')

        for j in range(n):
            if visited[j]:
                continue

            x2, y2 = cities[j]
            dx = x1 - x2
            dy = y1 - y2
            d = math.sqrt(dx**2 + dy**2)

            if d < min_dist:
                min_dist = d
                chosen_city = j

        tour.append(chosen_city)
        visited[chosen_city] = True
        total_dist += min_dist
        current = chosen_city

    x_last, y_last = cities[current]
    x_start, y_start = cities[start]
    total_dist += math.sqrt((x_last - x_start)**2 + (y_last - y_start)**2)

    return tour, total_dist
```

Main code block: Calls the functions to read cities from file, creates a list of 104 distinct starting points, and uses that list to make 104 distinct NN tours, which it writes to an output file



```python
if __name__ == "__main__":
    cities = read_cities("usa13509.tsp")
    print("Number of cities:", len(cities))

    starts = select_diverse_starts(cities, 25)

    with open("nearest_neighbor_tours.txt", "w") as f:
        for i, start in enumerate(starts):
            NN = nearest_neighbor(cities, start)
            f.write(f"Tour {i} (start: {start})\n")
            f.write(f"Distance: {NN[1]}\n\n")
            f.write(f"{NN[0]}\n\n")
```

Output File “nearest_neighbors.txt” (first few tours to show what it looks like)

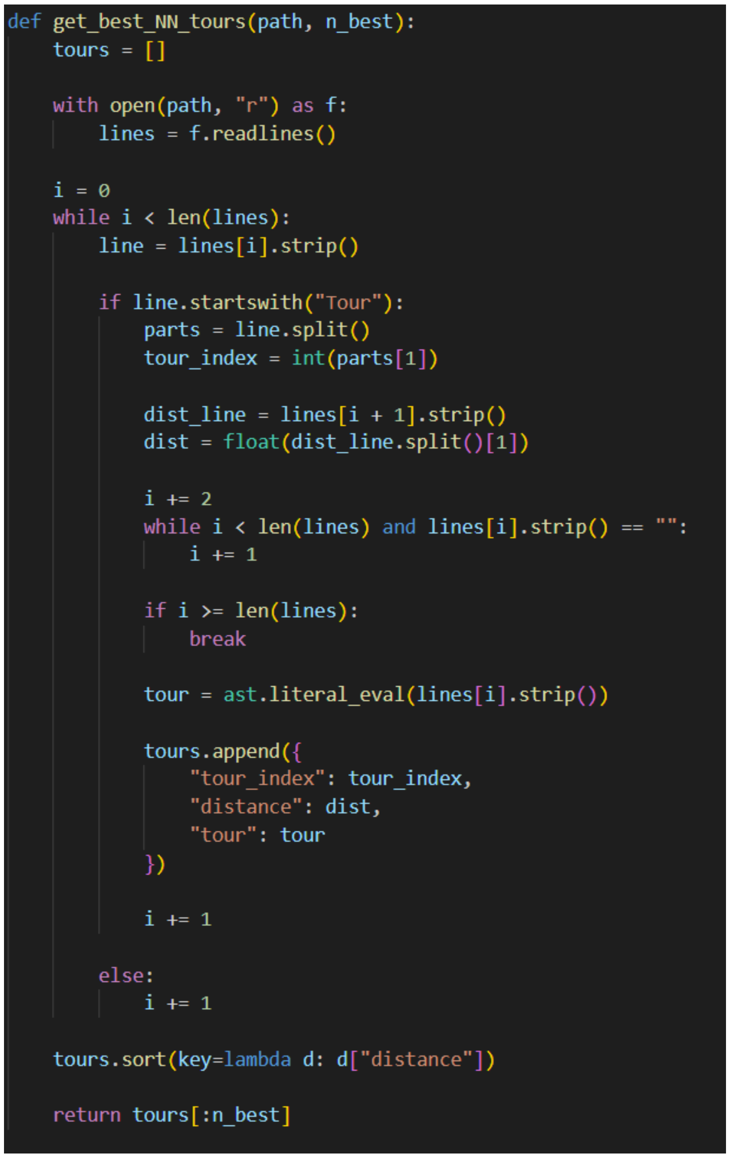


```
Tour 0 (start: 0)
Distance: 25047673.20526702

[0, 1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 9, 12, 13, 14, 17, 16, 18, 21, 20, 19, 24, 25, 29, 32, 31, 26, 40, 39, 42, 47, 50, 52, 51, 45, 43, 41, 44, 46, 49, 53, 54, 64, 72, 77, 79, 85, 91,

Tour 1 (start: 13508)
Distance: 25014411.851405032

[13508, 13502, 13505, 13496, 13484, 13483, 13456, 13453, 13458, 13446, 13461, 13434, 13423, 13411, 13404, 13412, 13463, 13507, 13427, 13398, 13389, 13396, 13401, 13422, 13405, 13383,

Tour 2 (start: 12514)
Distance: 25304115.452773686

[12514, 12675, 12661, 12403, 12326, 12251, 12199, 12336, 12351, 12378, 12420, 12457, 12451, 12473, 12496, 12532, 12541, 12693, 12737, 12835, 12842, 12927, 12955, 13088, 13111, 13126,

Tour 3 (start: 11056)
Distance: 25143468.304921046

[11056, 10668, 10227, 10228, 9846, 10369, 10706, 10702, 10558, 10585, 10521, 10477, 10404, 10705, 10758, 11234, 11263, 11342, 11474, 11504, 11557, 11683, 11714, 11823, 11866, 11940,

Tour 4 (start: 13390)
Distance: 25014277.530761696

[13390, 13412, 13404, 13411, 13423, 13434, 13446, 13453, 13456, 13458, 13483, 13484, 13496, 13505, 13502, 13508, 13507, 13461, 13427, 13398, 13389, 13396, 13401, 13422, 13405, 13383,

Tour 5 (start: 13159)
Distance: 25171959.166736897

[13159, 13144, 13163, 13158, 13162, 13164, 13169, 13175, 13122, 13101, 13092, 13084, 13068, 13048, 13030, 12993, 12969, 12963, 12960, 12965, 12947, 12931, 12912, 12900, 12869, 12889,

Tour 6 (start: 13144)
Distance: 25171558.865947105

[13144, 13159, 13163, 13158, 13162, 13164, 13169, 13175, 13122, 13101, 13092, 13084, 13068, 13048, 13030, 12993, 12969, 12963, 12960, 12965, 12947, 12931, 12912, 12900, 12869, 12889,

Tour 7 (start: 13412)
Distance: 25025125.075088922

[13412, 13404, 13411, 13423, 13434, 13446, 13453, 13456, 13458, 13483, 13484, 13496, 13505, 13502, 13508, 13507, 13461, 13427, 13398, 13389, 13396, 13401, 13422, 13405, 13383, 13376,

Tour 8 (start: 13163)
Distance: 25172925.789374642

[13163, 13158, 13162, 13164, 13169, 13175, 13122, 13101, 13092, 13084, 13068, 13048, 13030, 12993, 12969, 12963, 12960, 12965, 12947, 12931, 12912, 12900, 12869, 12889, 12915, 12877,
```

two_opt.py

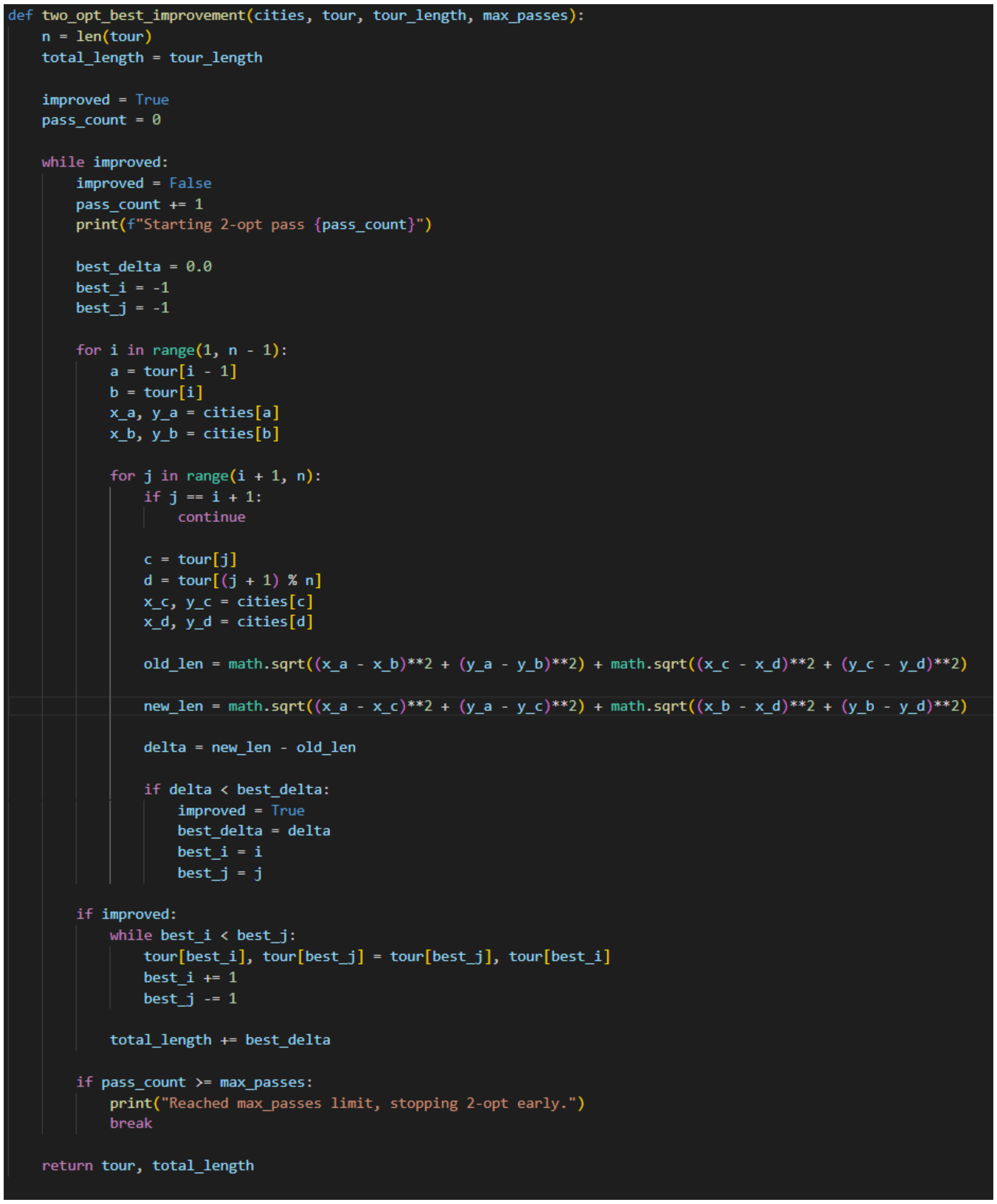
Function “read_cities” (same as before)

Function “get_best_NN_tours”: takes as input the previous output .txt file and number of best tours, and returns the n_best NN tours (sorted by smallest distance)



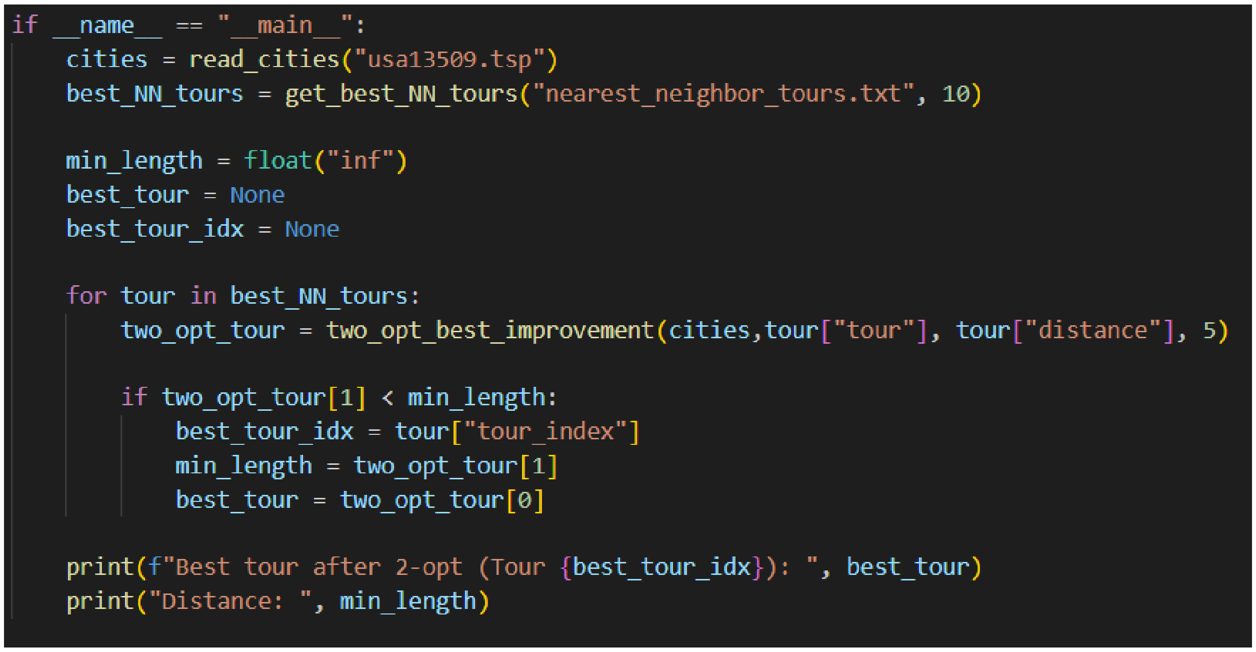
```python
def get_best_NN_tours(path, n_best):
    tours = []

    with open(path, "r") as f:
        lines = f.readlines()

    i = 0
    while i < len(lines):
        line = lines[i].strip()

        if line.startswith("Tour"):
            parts = line.split()
            tour_index = int(parts[1])

            dist_line = lines[i + 1].strip()
            dist = float(dist_line.split()[1])

            i += 2
            while i < len(lines) and lines[i].strip() == "":
                i += 1

            if i >= len(lines):
                break

            tour = ast.literal_eval(lines[i].strip())

            tours.append({
                "tour_index": tour_index,
                "distance": dist,
                "tour": tour
            })

            i += 1

        else:
            i += 1

    tours.sort(key=lambda d: d["distance"])

    return tours[:n_best]
```

Function “two_opt_best_improvement”: takes as input cities list, NN tour, tour length, and number for max passes, and returns the refined two-opt tour along with its total distance



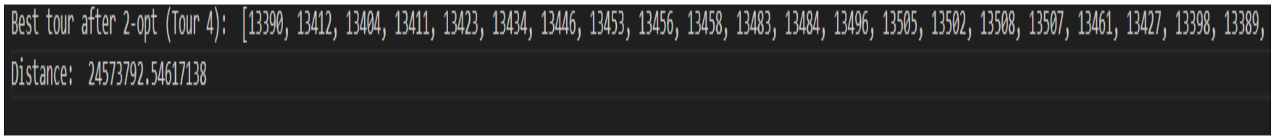
```python
def two_opt_best_improvement(cities, tour, tour_length, max_passes):
    n = len(tour)
    total_length = tour_length

    improved = True
    pass_count = 0

    while improved:
        improved = False
        pass_count += 1
        print(f"Starting 2-opt pass {pass_count}")

        best_delta = 0.0
        best_i = -1
        best_j = -1

        for i in range(1, n - 1):
            a = tour[i - 1]
            b = tour[i]
            x_a, y_a = cities[a]
            x_b, y_b = cities[b]

            for j in range(i + 1, n):
                if j == i + 1:
                    continue

                c = tour[j]
                d = tour[(j + 1) % n]
                x_c, y_c = cities[c]
                x_d, y_d = cities[d]

                old_len = math.sqrt((x_a - x_b)**2 + (y_a - y_b)**2) + math.sqrt((x_c - x_d)**2 + (y_c - y_d)**2)

                new_len = math.sqrt((x_a - x_c)**2 + (y_a - y_c)**2) + math.sqrt((x_b - x_d)**2 + (y_b - y_d)**2)

                delta = new_len - old_len

                if delta < best_delta:
                    improved = True
                    best_delta = delta
                    best_i = i
                    best_j = j

        if improved:
            while best_i < best_j:
                tour[best_i], tour[best_j] = tour[best_j], tour[best_i]
                best_i += 1
                best_j -= 1

            total_length += best_delta

        if pass_count >= max_passes:
            print("Reached max_passes limit, stopping 2-opt early.")
            break

    return tour, total_length
```

Main code block: Calls functions to read cities from file, gets the 10 best NN tours, runs the two opt best improvement with a max_passes value of 5 on each tour, and prints the best tour found (shortest distance). In hindsight, we should've written the tour out to a file, but I will just copy the print output to a file and submit.



```python
if __name__ == "__main__":
    cities = read_cities("usa13509.tsp")
    best_NN_tours = get_best_NN_tours("nearest_neighbor_tours.txt", 10)

    min_length = float("inf")
    best_tour = None
    best_tour_idx = None

    for tour in best_NN_tours:
        two_opt_tour = two_opt_best_improvement(cities,tour["tour"], tour["distance"], 5)

        if two_opt_tour[1] < min_length:
            best_tour_idx = tour["tour_index"]
            min_length = two_opt_tour[1]
            best_tour = two_opt_tour[0]

    print(f"Best tour after 2-opt (Tour {best_tour_idx}): ", best_tour)
    print("Distance: ", min_length)
```

Output:



```
Best tour after 2-opt (Tour 4):  [13390, 13412, 13404, 13411, 13423, 13434, 13446, 13453, 13456, 13458, 13483, 13484, 13496, 13505, 13502, 13508, 13507, 13461, 13427, 13398, 13389,
Distance:  24573792.54617138
```

Final Distance: 24,573,792.54617138