SMAI Assignment 2

Traveling Salesman Problem (TSP) Report

By Vandita Lodha and Suhani Jain

Approach

Our approach to solving the TSP combines two heuristic methods:

1. Nearest Neighbor Algorithm
2. 2-opt Optimization

Nearest Neighbor Algorithm

The **Nearest Neighbor Heuristic** is a greedy algorithm used to find an initial solution for the TSP. The main idea behind this approach is to iteratively select the nearest unvisited city at each step. The logic behind this algorithm is simple: by always moving to the closest available city, it hopes to minimize the total distance traveled.

1. **Initialization**: Start from an arbitrary city (typically chosen randomly). Mark this city as visited.
2. **Greedy Selection**: From the current city, look at all unvisited cities and select the one that is closest in terms of distance.
3. **Repeat**: Move to the nearest unvisited city, mark it as visited, and repeat the process until all cities have been visited.
4. **Complete the Tour**: Once all cities have been visited, return to the starting city to complete the tour.

PROF

It works on the premise that choosing the locally optimal solution (i.e., moving to the closest city) at each step can produce a reasonably short overall tour. This greedy approach is fast and simple, with a time complexity of (O(n^2)), where (n) is the number of cities.

However, **it doesn't guarantee an optimal solution** because it makes decisions based on local information without considering the global tour. It can get stuck in suboptimal routes by choosing close cities early on, potentially missing better global arrangements. Despite this, the Nearest Neighbor heuristic is useful for generating a quick, initial solution, which can then be refined using optimization techniques like 2-opt.

```
def nearest_neighbor(A: City, cities: Cities) -> City:
    #Find the nearest city to city A using Euclidean distance.
    return min(cities, key=lambda C: distance(A, C))

def nearest_tsp(cities: Cities, start: City = None) -> Tour:
    #Generate a tour using the nearest neighbor heuristic.
```

```
    start = start or next(iter(cities))
    tour = [start]
    unvisited = set(cities) - {start}

    while unvisited:
        nearest = nearest_neighbor(tour[-1], unvisited)
        tour.append(nearest)
        unvisited.remove(nearest)
```

2-opt Optimization

The **2-opt algorithm** is a local search optimization technique designed to improve an existing TSP tour by making small adjustments. The idea is to swap two edges in the tour and check whether this leads to a shorter route. If the swap results in an improvement, it is accepted, and the process is repeated until no more improvements can be made.

1. **Initial Tour**: Start with an initial tour, which could be generated by the Nearest Neighbor heuristic or any other method.
2. **Edge Reversal**: At each iteration, select two edges (i.e., segments of the tour) and reverse the order of the cities between them. This creates a new potential tour.
3. **Improvement Check**: Check if the new tour has a shorter total distance than the previous one. If it does, accept the new tour. Otherwise, discard the change.
4. **Repeat**: Continue swapping edges and checking for improvements until no further improvements are found. This is known as **convergence**.

The 2-opt algorithm works by eliminating "crossovers" or "intersections" in the tour, which tend to increase the total distance. By reversing segments of the tour, 2-opt systematically shortens the route. The algorithm is guaranteed to terminate because each swap either improves the tour or leaves it unchanged, and there is only a finite number of possible tours.

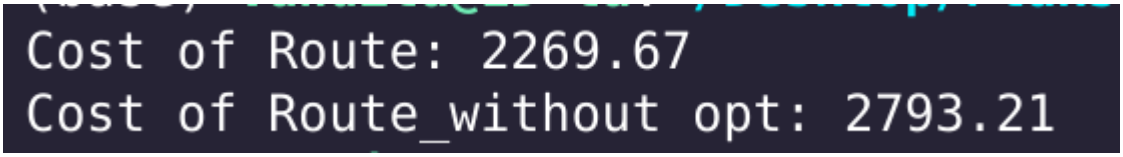
PROF

While 2-opt doesn't guarantee an optimal solution either, it can significantly improve a poor initial solution like the one generated by the Nearest Neighbor heuristic. The main advantage of 2-opt is that it is computationally feasible for large problem instances and provides a good trade-off between solution quality and computation time.

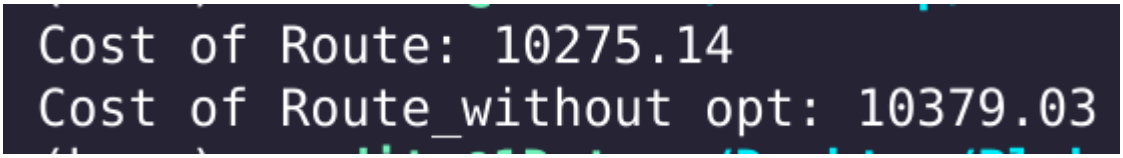
```
def opt2(tour: Tour) -> Tour:
    #Perform 2-opt optimization to improve the tour length.
    for (i, j) in subsegments(len(tour)):
        tour[i:j] = reversed(tour[i:j])
    return tour

def subsegments(N):
    #Generate subsegments for 2-opt optimization.
    return tuple((i, i + length) for length in reversed(range(2, N - 1))
for i in range(N - length))
```

path cost with and without 2-opt for N=200 Euclidean:



path cost with and without 2-opt for N=200 Non-Euclidean:



Implementation

1. **Data Structures**:
    - Cities are represented as complex numbers for easy Euclidean distance calculation
    - Tours are represented as lists of cities
    - A global dictionary `city_to_index` is used to map city coordinates to indices
2. **Key Functions**:
    - `nearest_neighbor`: Implements the Nearest Neighbor algorithm
    - `opt2`: Implements the 2-opt optimization
    - `rep_opt2_nearest_tsp`: Combines Nearest Neighbor and 2-opt, running multiple times with different starting cities

```
def rep_opt2_nearest_tsp(cities: Cities, k=10) -> Tour:
   #Apply nearest neighbor with 2-opt optimization over multiple
initializations.
    return min((opt2(nearest_tsp(cities, start)) for start in
random.sample(list(cities), min(k, len(cities)))),
               key=tour_length)
```

PROF

3. **Input Handling**:
    - The program can handle both Euclidean and non-Euclidean TSP instances
    - For non-Euclidean cases, the distance matrix is used
4. **Output**:
    - The program outputs the best tour found in the required format (zero-based city indices)

Alternative Approaches

While our final implementation uses the Nearest Neighbor algorithm with 2-opt optimization, there are other approaches that we had explored to solve the TSP:

1. Simulated Annealing

Simulated Annealing is a probabilistic technique for approximating the global optimum of a given function. For TSP, it works as follows:

1. Start with a random tour
2. Repeatedly make small random changes to the tour
3. Accept changes that improve the tour length
4. Probabilistically accept some changes that worsen the tour length, with the probability decreasing over time

Pros:

- Can escape local optima
- Generally produces good solutions

Cons:

- Performance depends on cooling schedule
- Was slower than our approach for larger instances

2. Ant Colony Optimization

Ant Colony Optimization is inspired by the behavior of ants finding paths between their colony and food sources. For TSP, it works as follows:

1. Initialize pheromone trails
2. Construct tours for a colony of ants based on pheromone levels and distances
3. Update pheromone levels based on the quality of tours
4. Repeat steps 2-3 for a number of iterations

PROF

Pros:

- Can find high-quality solutions
- Inherently parallel

Cons:

- Requires careful parameter tuning
- Was computationally expensive for large instances

Why Our Approach is Better

The combination of **Nearest Neighbor** and **2-opt** leverages the strengths of both algorithms.

1. **Nearest Neighbor** provides a fast starting point by ensuring that a complete tour is formed quickly, even if it’s not optimal.

2. **2-opt** then optimizes this initial tour by making small, localized improvements, reducing the overall tour length without needing to explore the entire solution space exhaustively.

This approach balances speed (Nearest Neighbor) and quality (2-opt). While neither method alone is guaranteed to find the optimal solution, their combination yields a solution that is close to optimal, especially when the problem size is large.

While Simulated Annealing and Ant Colony Optimization can potentially find better solutions, they often require more computational resources and careful parameter tuning. Our method strikes a balance between solution quality and computational efficiency, making it well-suited for the constraints of this assignment.

In terms of time complexities, this is how the three methods look:

Implemented Approach (Nearest Neighbor with 2-opt)

1. Nearest Neighbor (NN):
    - Time Complexity: O(n^2)
    - For each city, we search through all remaining cities to find the nearest one.
2. 2-opt Optimization:
    - Worst-case Time Complexity: O(n^2) per iteration
    - In the worst case, we might need to check all possible pairs of edges.
    - The number of iterations isn't fixed, but let's assume it's proportional to n for this analysis.
3. Overall Algorithm (rep_opt2_nearest_tsp):
    - Time Complexity: O(k * n^3), where k is the number of repetitions (200 in our code)
    - For each of k starts, we run NN (O(n^2)) and then 2-opt (O(n^3) assuming n iterations).

Simulated Annealing (SA)

PROF

- Time Complexity: O(n^2 * T), where T is the number of iterations
- Each iteration involves:
    - Generating a neighbor solution: O(1)
    - Calculating the change in tour length: O(1)
    - Deciding whether to accept the new solution: O(1)
- The total number of iterations T is typically set as a function of n and the cooling schedule.
- In practice, T is often chosen to be n^2, resulting in an O(n^4) algorithm.

Ant Colony Optimization (ACO)

- Time Complexity: O(N * n^2 * m), where:
    - N is the number of iterations
    - n is the number of cities
    - m is the number of ants

- In each iteration:
    - Each ant constructs a tour: O(n^2)
    - Update pheromone levels: O(n^2)
- Typically, m is chosen to be equal to n, giving a time complexity of O(N * n^3)

Comparison

1. Our Approach: O(k * n^3)
    - Predictable performance
    - Scales cubically with the number of cities
    - The constant factor k (number of restarts) can be adjusted for time/quality trade-off
2. Simulated Annealing: O(n^4) (typical implementation)
    - Can potentially find better solutions given enough time
    - Performance heavily depends on cooling schedule and initial temperature
3. Ant Colony Optimization: O(N * n^3)
    - Can find high-quality solutions for some problem instances
    - Performance depends on the number of iterations and number of ants
    - May require careful tuning of parameters

Potential Improvements for the future

1. **Hybrid Approaches**: Combine our method with elements of Simulated Annealing or Ant Colony Optimization for potentially better results.
2. **Parallelization**: Utilize parallel processing to explore multiple tours simultaneously.
3. **Adaptive Parameters**: Dynamically adjust the number of restarts based on available time and problem size.
4. **Caching**: Implement caching mechanisms to avoid redundant distance calculations.
5. **Early Termination**: Add an early termination condition to stop optimization if improvements become marginal.

Conclusion

The implemented solution provides a balance between solution quality and computational efficiency. By combining the quick initial tour generation of Nearest Neighbor with the improvement capabilities of 2-opt, we achieve reasonably good tours within the given time constraints. Our future work would focus on implementing methods to ensure further path optimisation.