Non-deterministic Search | Simulated Annealing

1. Overview of Simulated Annealing:

Simulated Annealing (SA) is a non-deterministic optimization algorithm inspired by the process of annealing in metallurgy.
It is widely used for solving combinatorial optimization problems, including the Travelling Salesman Problem (TSP).
SA is based on the concept of accepting "worse" solutions with a certain probability to explore the solution space effectively.
2. How Simulated Annealing Works:

At each iteration, SA considers a neighboring solution by making a small random change to the current solution.
If the neighboring solution improves the objective function (e.g., reduces the total tour distance in TSP), it is always accepted.
If the neighboring solution worsens the objective function, it may still be accepted with a probability determined by the "temperature" parameter.
The temperature parameter controls the likelihood of accepting worse solutions early in the optimization process, allowing for exploration of the solution space.
As the optimization progresses, the temperature is gradually decreased, leading to a more deterministic search towards the end.
3. Key Components of Simulated Annealing:

Initial Solution: SA starts with an initial solution, which can be randomly generated or obtained through a heuristic method.
Neighborhood Structure: SA defines a neighborhood structure that determines how neighboring solutions are generated.
Cooling Schedule: The cooling schedule determines how the temperature parameter decreases over iterations, balancing exploration and exploitation.
Acceptance Criterion: The acceptance criterion specifies the probability of accepting worse solutions based on the current temperature and the magnitude of the objective function change.
4. Advantages of Simulated Annealing for TSP:

Ability to Escape Local Optima: SA can escape local optima by accepting worse solutions early in the optimization process, leading to better exploration of the solution space.
Flexibility: SA is flexible and can handle various objective functions and constraints, making it suitable for solving complex optimization problems like TSP.
Convergence to Near-Optimal Solutions: With an appropriate cooling schedule, SA can converge to near-optimal solutions for TSP instances with large search spaces.
5. Limitations of Simulated Annealing for TSP:

Sensitivity to Parameters: The performance of SA can be sensitive to the choice of parameters such as initial temperature, cooling schedule, and neighborhood structure.
Slow Convergence: SA may require a large number of iterations to converge to near-optimal solutions, especially for TSP instances with highly irregular cost functions.
Lack of Determinism: Due to its probabilistic nature, SA does not guarantee the same solution for multiple runs, which can pose challenges for result reproducibility.

Closing Remark: Simulated Annealing offers a powerful approach for tackling the Travelling Salesman Problem, providing a balance between exploration and exploitation to find high-quality solutions in large solution spaces.

Visuals: Consider including diagrams illustrating the concept of Simulated Annealing, such as a depiction of the temperature schedule or the exploration of the solution space over iterations.

CODE:

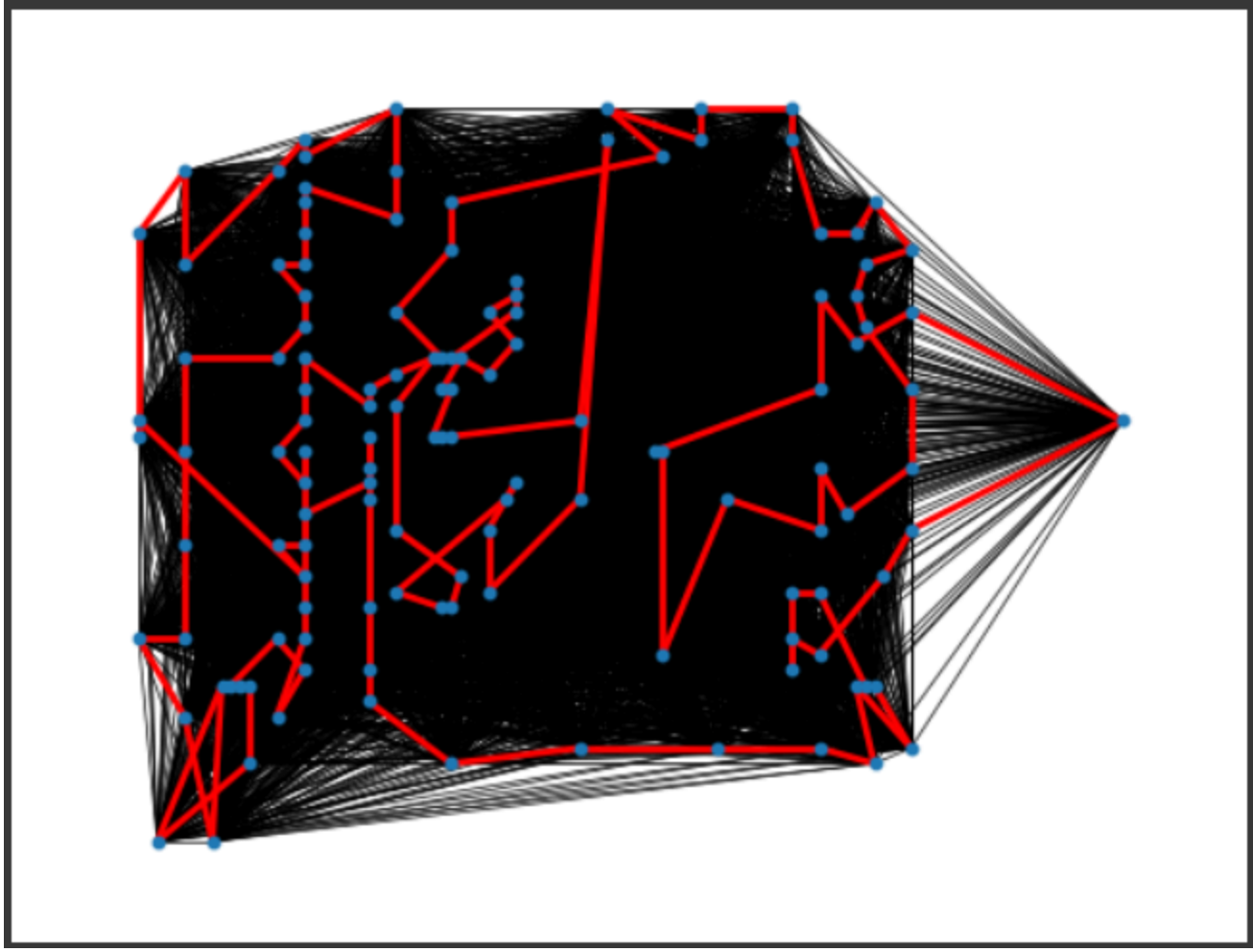
```
import networkx as nx
import math
import random
import matplotlib.pyplot as plt

# Function to calculate Euclidean distance between two points
def euclidean_distance(coord1, coord2):
    return math.sqrt((coord1[0] - coord2[0])**2 + (coord1[1] - coord2[1])**2)

# Read the TSP file and extract node coordinates
tsp_file = "/content/drive/MyDrive/xqf131.tsp"

nodes = {}
with open(tsp_file, "r") as file:
    lines = file.readlines()
    # Find the line index where the node coordinates start
    coord_start_idx = lines.index("NODE_COORD_SECTION\n") + 1

    # Iterate over lines starting from the line containing node coordinates
    for line in lines[coord_start_idx:]:
        if line.startswith("EOF"):
            break
        parts = line.split()
        node_id = int(parts[0])  # Assuming the first column contains the node IDs
        x = float(parts[1])
        y = float(parts[2])
        nodes[node_id] = (x, y)

# Create a graph
G = nx.Graph()

# Add nodes with coordinates
for node_id, coords in nodes.items():
    G.add_node(node_id, pos=coords)

# Add edges with distances
```

```python
for i in range(1, len(nodes) + 1):
    for j in range(i + 1, len(nodes) + 1):
        distance = euclidean_distance(nodes[i], nodes[j])
        G.add_edge(i, j, distance=distance)

# Function to calculate the total distance of a tour
def tour_distance(tour, graph):
    distance = 0
    for i in range(len(tour) - 1):
        distance += graph[tour[i]][tour[i + 1]]['distance']
    distance += graph[tour[-1]][tour[0]]['distance']  # Return to the starting point
    return distance

# Function to generate a random initial tour
def initial_tour(graph):
    tour = list(graph.nodes)
    random.shuffle(tour)
    return tour

# Function to generate a random neighbor by reversing a segment of the tour
def random_neighbor(tour):
    neighbor = tour.copy()
    i, j = sorted(random.sample(range(len(tour)), 2))
    neighbor[i:j+1] = reversed(neighbor[i:j+1])
    return neighbor
# Simulated Annealing algorithm for TSP with Metropolis rule
def simulated_annealing(graph, max_iterations, initial_temperature=100.0,
cooling_rate=0.95):
    current_solution = initial_tour(graph)
    current_cost = tour_distance(current_solution, graph)

    best_solution = current_solution.copy()
    best_cost = current_cost

    temperature = initial_temperature

    for _ in range(max_iterations):
        neighbor_solution = random_neighbor(current_solution)

        delta = tour_distance(neighbor_solution, graph) - current_cost

        if delta < 0 or random.random() < math.exp(-delta / temperature):
            current_solution = neighbor_solution
            current_cost = tour_distance(current_solution, graph)
```

```
            if current_cost < best_cost:
                best_solution = current_solution.copy()
                best_cost = current_cost

        # Update temperature
        temperature *= cooling_rate

    return best_solution, best_cost


# Number of iterations for Simulated Annealing
max_iterations = 10000

# Run Simulated Annealing
best_tour, best_cost = simulated_annealing(G, max_iterations)

print("Best tour:", best_tour)
print("Best cost:", best_cost)

# Visualize the best tour on the graph
pos = nx.get_node_attributes(G, 'pos')
nx.draw_networkx_nodes(G, pos, node_size=10)
nx.draw_networkx_edges(G, pos, width=0.5)
nx.draw_networkx_edges(G, pos, edgelist=[(best_tour[i], best_tour[i+1]) for i in
range(len(best_tour)-1)], edge_color='r', width=2)
nx.draw_networkx_edges(G, pos, edgelist=[(best_tour[-1], best_tour[0])], edge_color='r',
width=2)
plt.axis('off')
plt.show()
```

OUTPUT:

```
Best tour: [22, 9, 8, 7, 6, 1, 12, 13, 14, 5, 18, 17, 16, 15, 19, 26, 25, 27, 2
Best cost: 802.223218586425
```



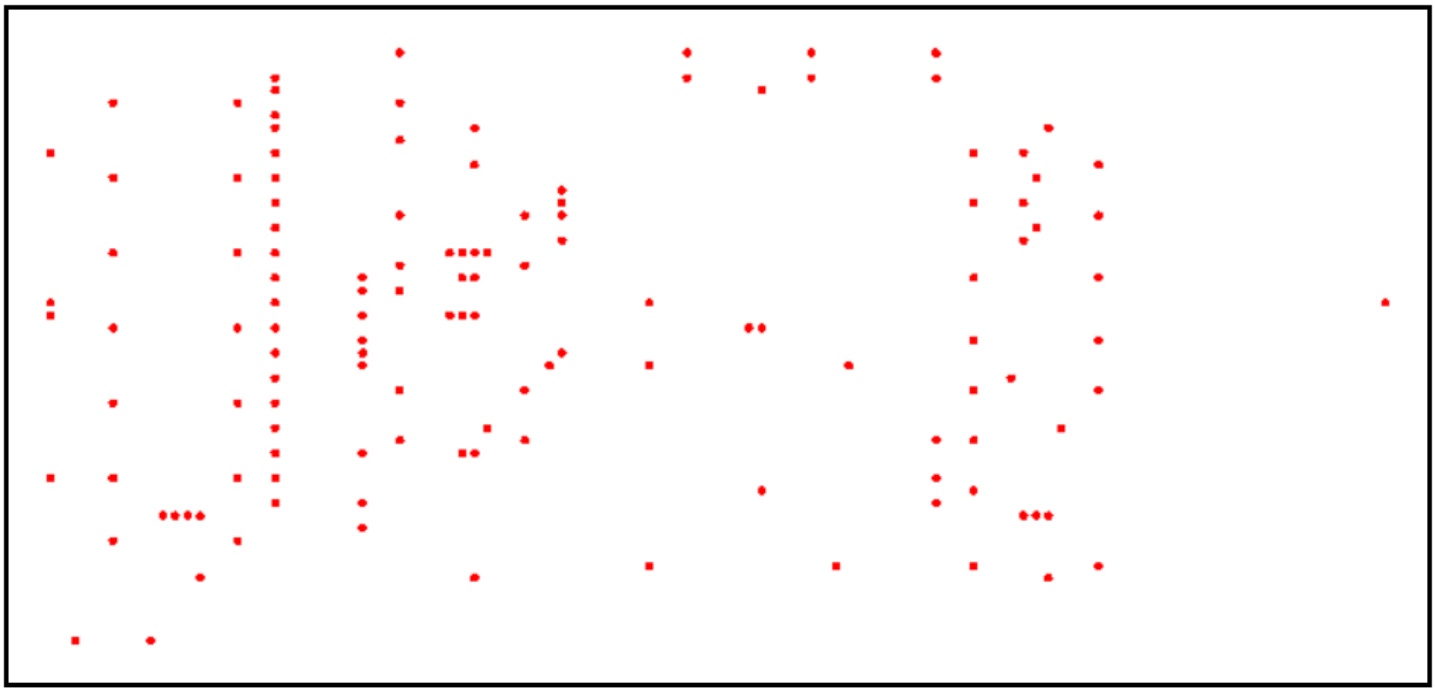
Different Approch:

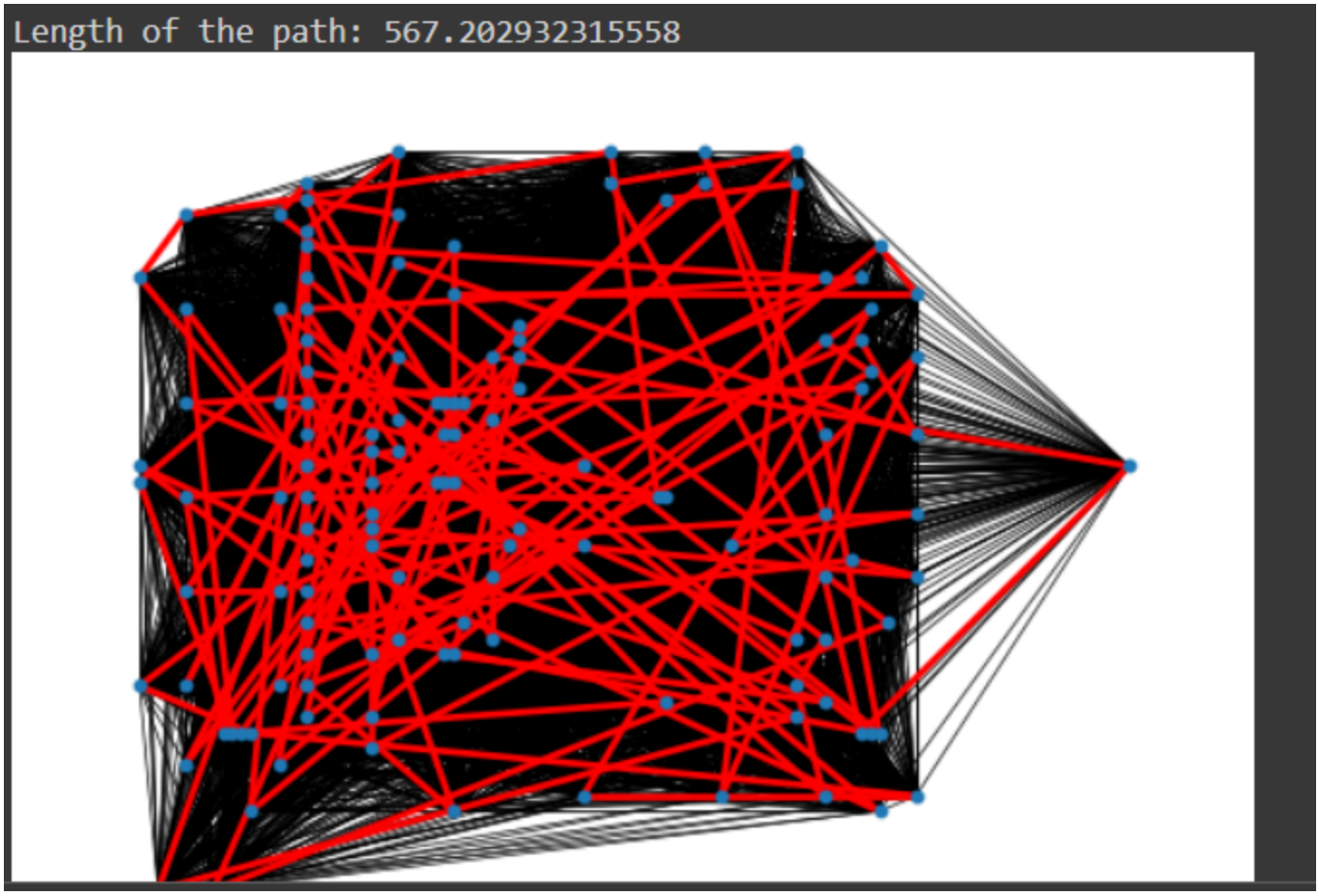
1. Brute Force: Enumerates all possible permutations of the cities and selects the one with the minimum total distance. Computationally expensive for large instances due to factorial time complexity.
Branch and Bound: Systematically explore the solution space by branching on decisions and bounding the search based on the current best solution. Guarantees an optimal solution but can be computationally expensive.
The Brute Force approach, also known as the Naive Approach, calculates and compares all possible permutations of routes or paths to determine the shortest unique solution. To solve the TSP using the Brute-Force approach, you must calculate the total number of routes and then draw and list all the possible routes. Calculate the distance of each route and then choose the shortest one—this is the optimal solution.

In theoretical computer science, the TSP has commanded so much attention because it's so easy to describe yet so difficult to solve. The TSP is known to be a combinatorial optimization problem that's an NP-hard problem, which means that the number of possible solution sequences grows exponential with the number of cities. Computer scientists have not found any algorithm that can solve this problem in polynomial time, and therefore rely on approximation algorithms to try numerous permutations and select the shortest route with the minimum cost.

The main problem can be solved by calculating every permutation using a brute force approach and selecting the optimal solution. However, as the number of destinations increases, the corresponding number of roundtrips grows exponentially and surpasses the capabilities of even the fastest computers. With 10 destinations, there can be more than 300,000 roundtrip permutations. With 15 destinations, the number of possible routes could exceed 87 billion.

This is only feasible for small problems, and rarely useful beyond theoretical computer science tutorials.
Brute Force: O(n!), where n is the number of cities. Brute force exhaustively searches through all possible permutations of cities to find the optimal tour, resulting in factorial time complexity.





2. Dynamic Programming (Held-Karp): Breaks down the problem into subproblems and solves them recursively. It is efficient for small instances but has exponential time complexity.
Dynamic Programming (Held-Karp): O(n^2 * 2^n), where n is the number of cities. Dynamic programming efficiently computes the shortest tour length by memoizing subproblem solutions, resulting in a significantly lower time complexity compared to brute force.

3. Hill Climbing algorithm: Hill climbing is a local search algorithm used to find a satisfactory solution to optimization problems. In the context of the Traveling Salesman Problem (TSP), the goal is to find the shortest possible tour that visits each city exactly once and returns to the starting city. It is a greedy algorithm in which we move to the next state when the next state has better heuristic value. This algorithm does not have an idea about the whole global problem; instead it uses a greedy approach to find the optimal solution. That's why it does not always give the optimal solution. Benefit of this algorithm is that it works in constant space complexity and has a better time complexity than brute force and DP solution.

4. Minimum Spanning Tree (MST) Heuristic: Constructs a minimum spanning tree of the graph and then generates a tour by traversing the tree using a depth-first search.
Christofides Algorithm: Combines the MST with a matching algorithm to guarantee a tour with at most 1.5 times the optimal tour length for metric TSP instances.

Minimum Spanning Tree (MST) Heuristic: O(n^2 * log n), where n is the number of cities. This algorithm constructs a minimum spanning tree and then generates a tour by traversing the tree, resulting in quadratic time complexity.
Time complexity of prism is E*log(v)

```
##Minimum Spanning Tree (MST) Heuristic: O(n^2 * log n)
#
# Add edges with distances
for i in range(1, len(nodes) + 1):
    for j in range(i + 1, len(nodes) + 1):
        distance = euclidean_distance(nodes[i], nodes[j])
        G.add_edge(i, j, distance=distance)

# Function to construct Minimum Spanning Tree (MST) using Prim's algorithm
def prim_mst(graph):
    mst_edges = nx.minimum_spanning_edges(graph, algorithm='prim', data=False)
    mst = nx.Graph(list(mst_edges))
    return mst

# Function to generate a tour from a Minimum Spanning Tree (MST)
def mst_tour(mst):
    tour = list(nx.dfs_preorder_nodes(mst, source=1))  # DFS traversal of the MST
    tour.append(tour[0])  # Add the starting node to complete the tour
    return tour

# Construct Minimum Spanning Tree (MST)
mst = prim_mst(G)

# Generate tour from MST
mst_tour = mst_tour(mst)

# Compute tour length
tour_length = sum(G[mst_tour[i]][mst_tour[i + 1]]['distance'] for i in range(len(mst_tour)
- 1))

print("Tour:", mst_tour)
print("Tour length:", tour_length)
```

5. The nearest neighbor method

To implement the Nearest Neighbor algorithm, we begin at a randomly selected starting point. From there, we find the closest unvisited node and add it to the sequencing. Then, we move to the next node and repeat the process of finding the nearest unvisited node until all nodes are included in the tour. Finally, we return to the starting city to complete the cycle.

While the Nearest Neighbor approach is relatively easy to understand and quick to execute, it rarely finds the optimal solution for the traveling salesperson problem. It can be significantly longer than the optimal route, especially for large and complex instances. Nonetheless, the Nearest Neighbor algorithm serves as a good starting point for tackling the traveling salesman problem and can be useful when a quick and reasonably good solution is needed.
Starts from a random city and iteratively selects the nearest unvisited city until all cities are visited. Simple and fast but may not always produce optimal solutions.
Insertion Heuristics: Builds a tour incrementally by iteratively inserting cities into a partially constructed tour based on some criteria (e.g., nearest, farthest, cheapest insertion).
Genetic Algorithms: Simulates the process of natural selection and evolution to iteratively evolve a population of candidate solutions using genetic operators like selection, crossover, and mutation.

This greedy algorithm can be used effectively as a way to generate an initial feasible solution quickly, to then feed into a more sophisticated local search algorithm, which then tweaks the solution until a given stopping condition.
Nearest Neighbor: O(n^2), where n is the number of cities. Nearest Neighbor iteratively selects the nearest unvisited city for each city, resulting in quadratic time complexity.

```
import networkx as nx
import math
import random

# Function to calculate Euclidean distance between two points
def euclidean_distance(coord1, coord2):
    return math.sqrt((coord1[0] - coord2[0])**2 + (coord1[1] - coord2[1])**2)

# Read the TSP file and extract node coordinates
tsp_file = "/content/drive/MyDrive/xqf131.tsp"

nodes = {}
with open(tsp_file, "r") as file:
    lines = file.readlines()
    # Find the line index where the node coordinates start
    coord_start_idx = lines.index("NODE_COORD_SECTION\n") + 1

    # Iterate over lines starting from the line containing node coordinates
    for line in lines[coord_start_idx:]:
        if line.startswith("EOF"):
```

```
            break
        parts = line.split()
        node_id = int(parts[0])  # Assuming the first column contains the node IDs
        x = float(parts[1])
        y = float(parts[2])
        nodes[node_id] = (x, y)

# Create a graph
G = nx.Graph()

# Add nodes with coordinates
for node_id, coords in nodes.items():
    G.add_node(node_id, pos=coords)

# Add edges with distances
for i in range(1, len(nodes) + 1):
    for j in range(i + 1, len(nodes) + 1):
        distance = euclidean_distance(nodes[i], nodes[j])
        G.add_edge(i, j, distance=distance)

# Simulated Annealing algorithm for TSP
def simulated_annealing(graph, max_iterations):
    def tour_length(tour):
        return sum(graph[tour[i]][tour[i + 1]]['distance'] for i in range(len(tour) - 1)) +
graph[tour[-1]][tour[0]]['distance']

    current_tour = list(graph.nodes)
    random.shuffle(current_tour)
    current_length = tour_length(current_tour)

    best_tour = current_tour.copy()
    best_length = current_length

    for _ in range(max_iterations):
        neighbor_tour = current_tour.copy()
        i, j = sorted(random.sample(range(len(current_tour)), 2))
        neighbor_tour[i:j+1] = reversed(neighbor_tour[i:j+1])

        neighbor_length = tour_length(neighbor_tour)
        delta = neighbor_length - current_length

        if delta < 0 or random.random() < math.exp(-delta / current_length):
            current_tour = neighbor_tour
            current_length = neighbor_length
```

```
            if current_length < best_length:
                best_tour = current_tour.copy()
                best_length = current_length

    return best_tour, best_length

# Number of iterations for Simulated Annealing
max_iterations = 10000

# Run Simulated Annealing
best_tour, best_length = simulated_annealing(G, max_iterations)

print("Best tour:", best_tour)
print("Best length:", best_length)

# Visualize the best tour on the graph
pos = nx.get_node_attributes(G, 'pos')
nx.draw_networkx_nodes(G, pos, node_size=10)
nx.draw_networkx_edges(G, pos, width=0.5)
nx.draw_networkx_edges(G, pos, edgelist=[(best_tour[i], best_tour[i+1]) for i in
range(len(best_tour)-1)], edge_color='r', width=2)
nx.draw_networkx_edges(G, pos, edgelist=[(best_tour[-1], best_tour[0])], edge_color='r',
width=2)
plt.axis('off')
plt.show()
```

Initialization:

Start with a random tour, which is a permutation of all nodes in the graph.
Calculate the total length of the current tour.
Main Loop:

Repeat a specified number of times (controlled by max_iterations):
Generate a neighboring tour by randomly selecting two indices in the tour and reversing the order of nodes between those indices.
Calculate the total length of the neighboring tour.
If the length of the neighboring tour is shorter than the current tour, accept the neighboring tour as the new current tour.
If the length of the neighboring tour is longer than the current tour, accept it with a certain probability, which depends on the length difference and a temperature parameter. This allows the algorithm to escape local optima and explore the solution space.
Update the best tour if the current tour is better than the previous best tour.
Termination:

After a fixed number of iterations, return the best tour found along with its length.
The tour_length function calculates the total length of a given tour by summing the distances between consecutive nodes in the tour, including the distance from the last node back to the first node to complete the cycle.

The algorithm iteratively explores the solution space, gradually improving the tour length by accepting better solutions and occasionally accepting worse solutions to explore the search space more effectively. Finally, it returns the best tour found along with its length.

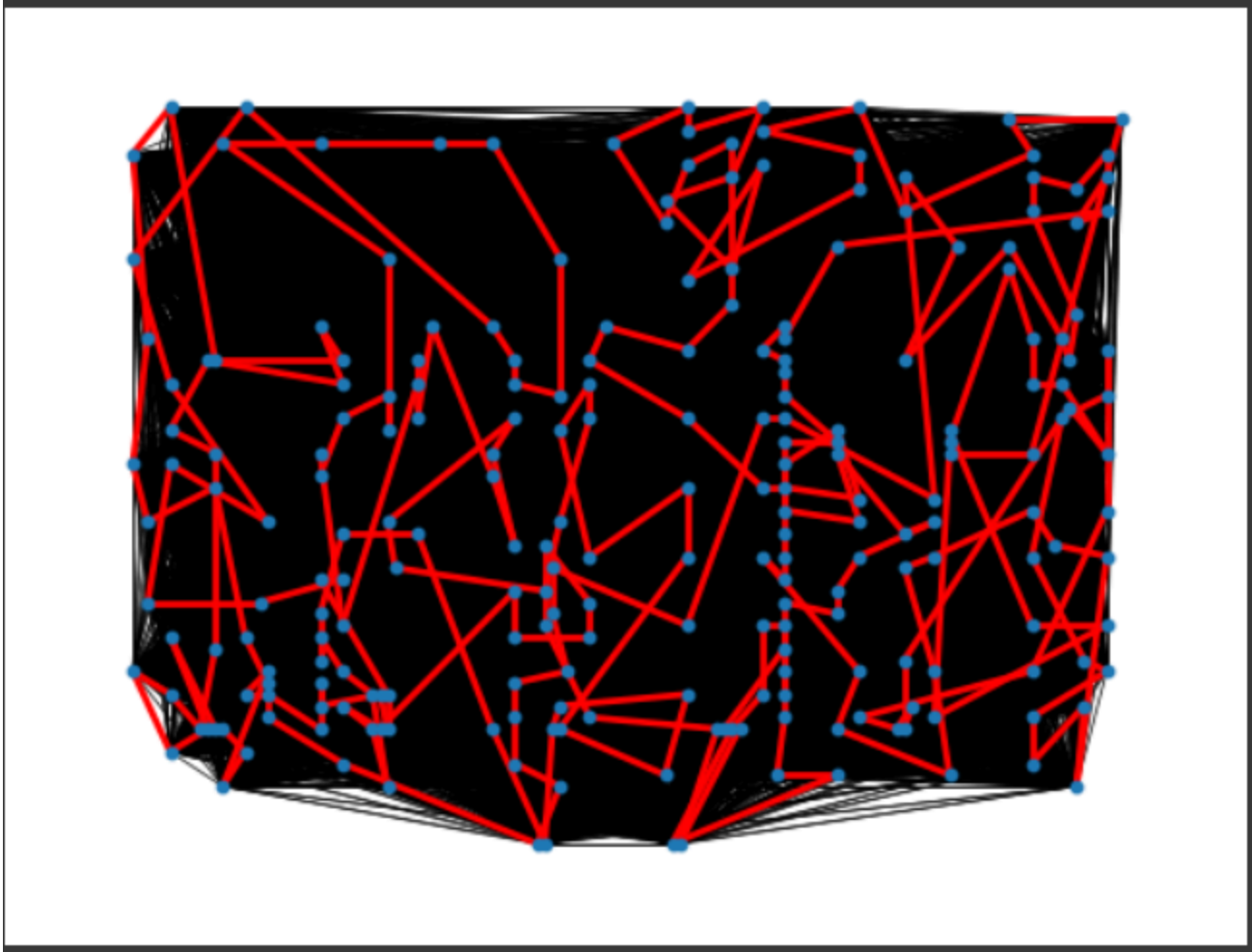
The time complexity of solving the Traveling Salesman Problem (TSP) depends on the specific algorithm used. Here are the time complexities for some common TSP solution algorithms:

Result:

DATA :2

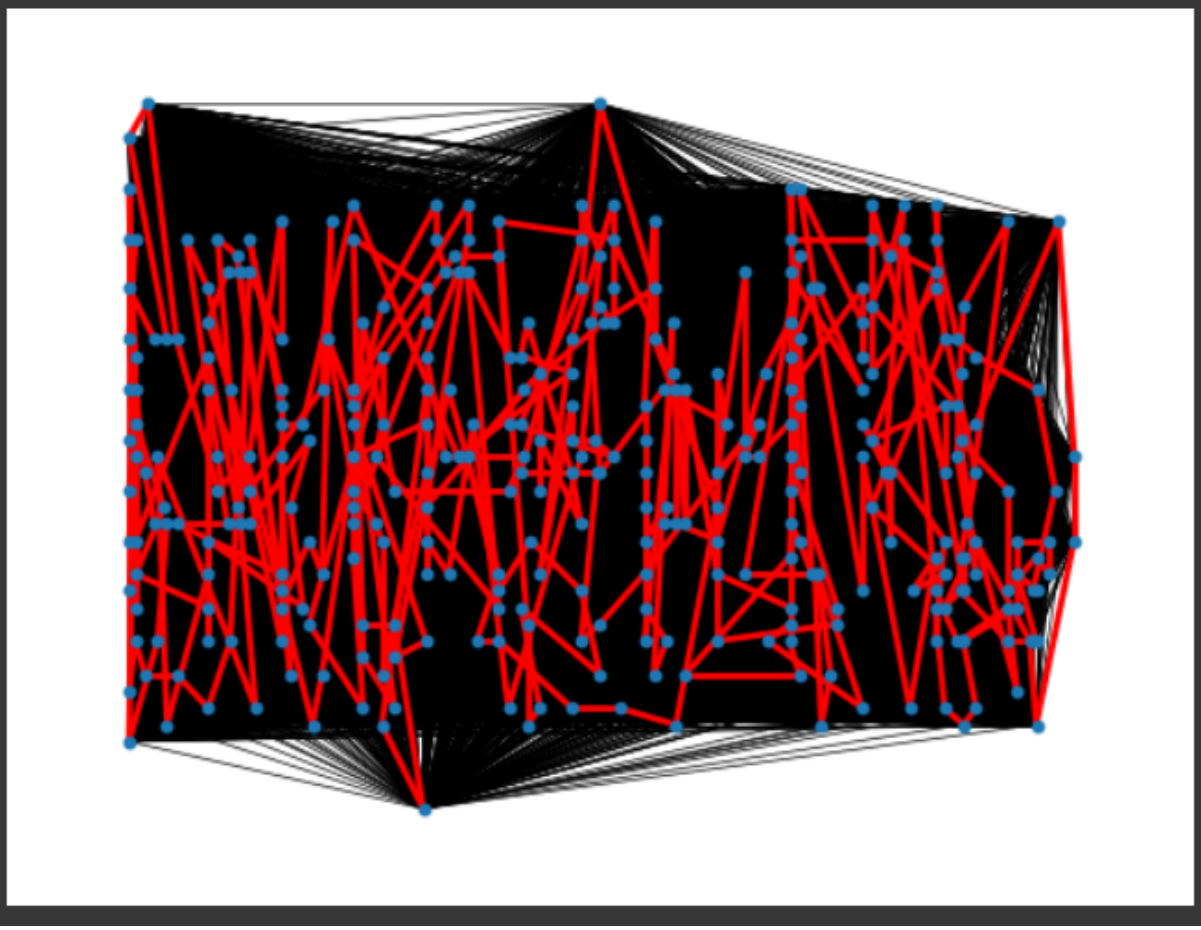
Optimal : 1019

```
Best tour: [126, 144, 133, 114, 132, 142, 124, 178, 179, 143, 180, 187, 213, 202
Best cost: 2248.3113234810257
```



DATA :3
Optimal: 1368

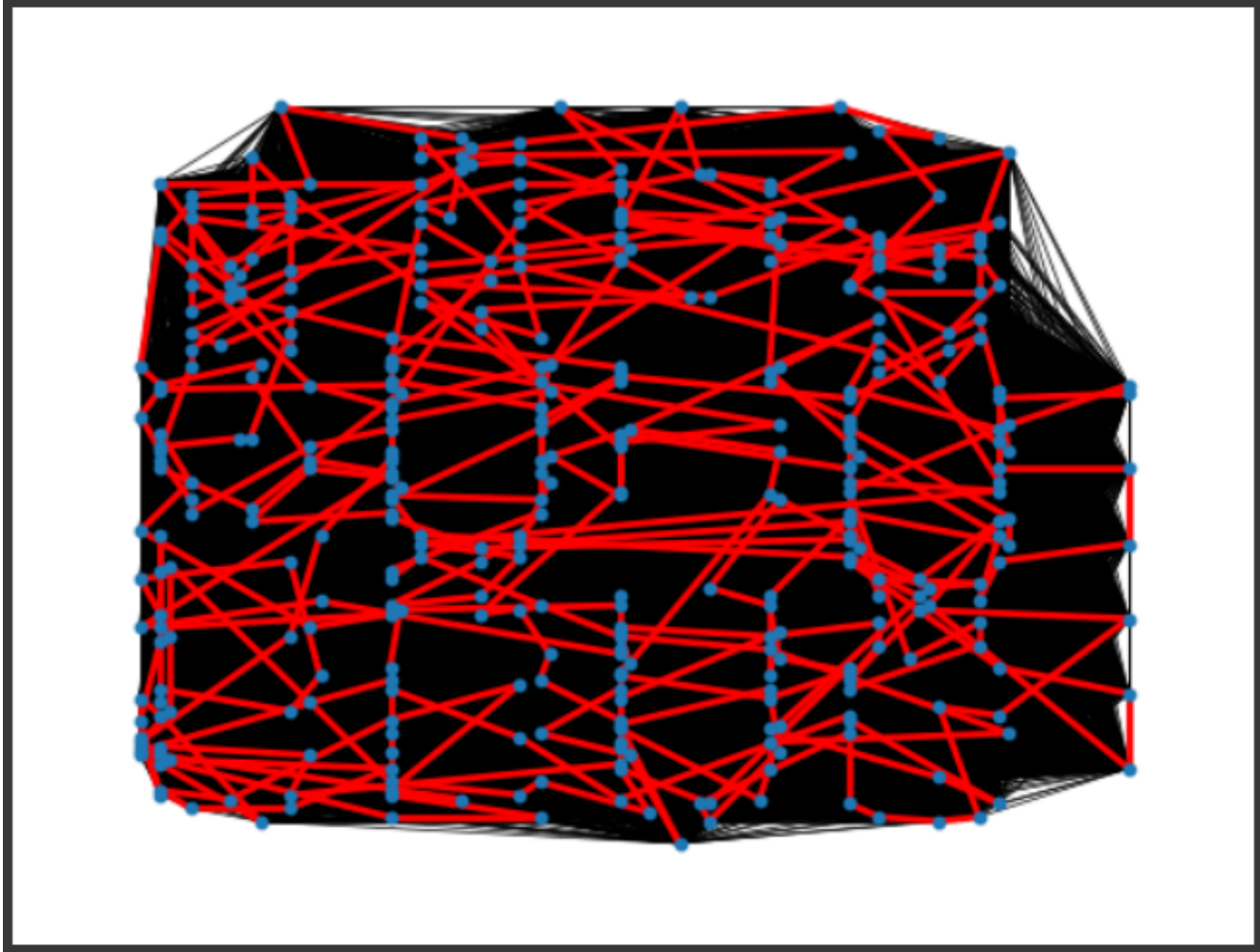
```
Best tour: [241, 229, 218, 210, 211, 213, 214, 217, 233, 231, 251, 215, 234, 2
Best cost: 4379.792465121233
```



Data 4:

Optimal: 1621

```
Best tour: [301, 306, 310, 343, 354, 367, 279, 312, 331, 342, 340, 278, 197
Best cost: 5688.588794073217
```



```
Best tour: [2, 8, 1, 10, 3, 16, 9, 18, 17, 0, 13, 12, 7, 5, 15, 11, 4, 6, 19, 14]
Total distance: 3038
Results saved to tour of rajasthan.tspfile.
```