THE TRAVELLING SALESMAN PROBLEM

By Group M

Clone URL: Clone URL: https://github.com/mawande-johnpaul/TSP.git

Repo URL: [mawande-johnpaul/TSP: Comparison of SOM and Exhaustive Algorithms]

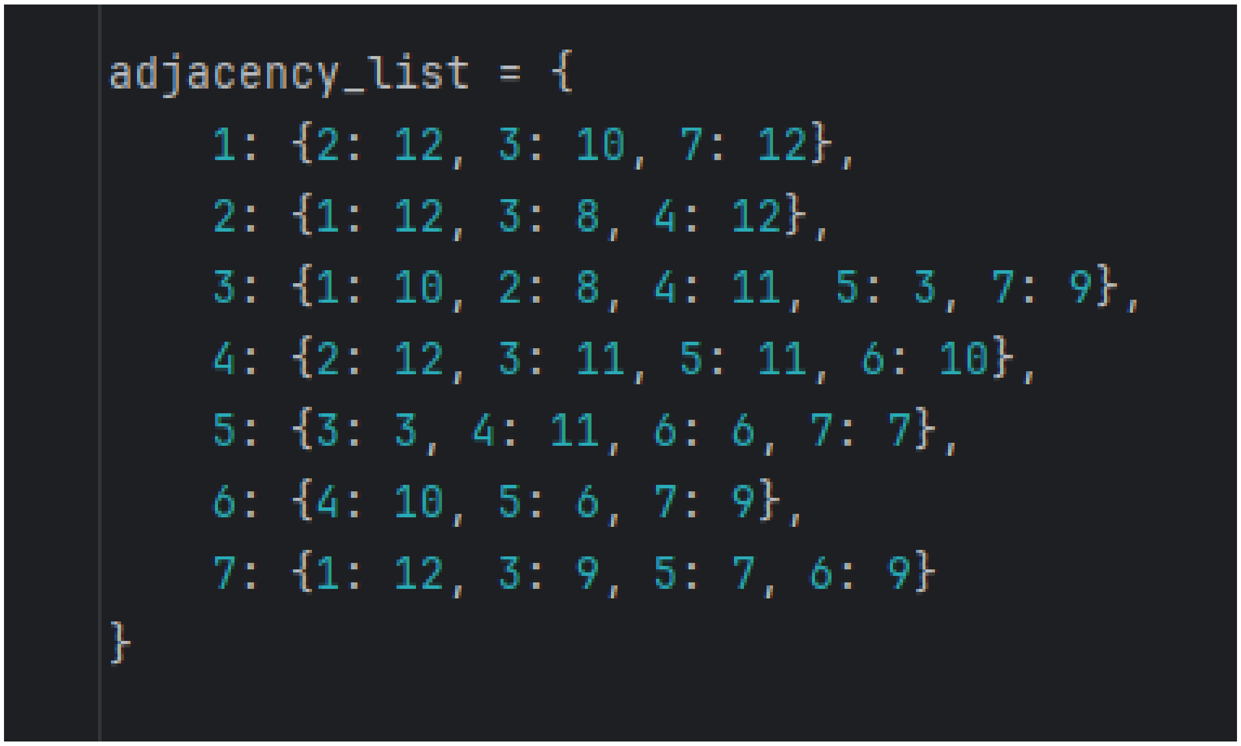
NAMES	REGISTRATION NUMBER	CONTACT
Mawande John Paul	24/U/0678	787 130 130
Econgu Paul	24/U/22570	789 513 778
Lyazi Patrick	24/U/06571/PS	782 726 299
Namuli Angel Rebecca	24/U/09332/PS	756 055 966
Namayanja Mary Prechard	24/U/09046/PS	773 366 103

DEFINITION AND INTRODUCTION

The Travelling Salesman Problem (TSP) is a real-world combinatorial optimization problem that involves finding the shortest possible route that visits each city exactly once and returns to the starting point. The objective is to minimize the total travel distance or cost while ensuring computational efficiency.

Input: Data structure

- Points are stored as a dictionary where the;
  - Keys: All the individual points.
  - Values: Dictionaries containing the keys as points adjacent to the current point and their respective distances from the current point (node).



```
adjacency_list = {
    1: {2: 12, 3: 10, 7: 12},
    2: {1: 12, 3: 8, 4: 12},
    3: {1: 10, 2: 8, 4: 11, 5: 3, 7: 9},
    4: {2: 12, 3: 11, 5: 11, 6: 10},
    5: {3: 3, 4: 11, 6: 6, 7: 7},
    6: {4: 10, 5: 6, 7: 9},
    7: {1: 12, 3: 9, 5: 7, 6: 9}
}
```

Effectiveness

- It is easy to check validity of a path by checking if the next node is in the adjacent points of that point.
- Every adjacent point has a distance from the current node tied to it so a direct reference can be made for rapid computation.

Output: A sequence of numbers representing the most optimal route.

Solution;

- Two types of solutions (algorithms) are available;
  - Exact (non-heuristic) algorithms

    These guarantee an optimal solution.

    - Brute-force (Exhaustive Search)
    - Dynamic Programming (Held-Karp)
    - Branch and Bound

  - Heuristic (Approximate)

    These do not guarantee the best path or a valid one.

    - Nearest-Neighbor
    - Genetic Algorithm
    - Simulated Annealing
    - Christofides Algorithm (Best approximation technique)
    - Self-Organizing-Maps

Comparison of solutions

- Algorithms are classified mainly by complexity (number of steps), performance and how much of a guarantee they offer for an optimal solution.

Table showing comparison in complexity, pros and cons of each algorithm.

Algorithm	Description	Complexity	Suitability	Pros	Cons
Dynamic Programming	Solves optimally by breaking the problem into subproblems.	*O*(*n*2·2*n*)	20-30 cities	Optimal solution, avoids recompilation.	Memory-heavy, computationally expensive for large n
Branch-and-Bound	Systematically explores the solution space with pruning.	O(e^n)	Problems requiring optimal solution.	Guarantees optimal solution if fully explored Prunes parts of the search space to avoid unnecessary checks.	Still computationally expensive, impractical for large datasets.
Nearest Neighbor	Greedy heuristic. Picks nearest city next.	*O*(*n*2)	Large problems needing quick solutions.	Fast, simple to implement.	No optimality guarantee, can get stuck in local minima.
Self-Organizing Map (SOM)	Neural network clustering approach, useful for approximation.	Varies (depends on grid size and iterations)	Very large problems.	Scalable, good for large datasets higher dimension, captures patterns.	No guarantee of optimality, solution quality depends on map configuration.
Brute-force	Tests all possible solutions and picks the best.	*O*(*n*!)	< 10 points	Guarantees optimal path, Easy to implement.	factorial time complexity increases extremely rapidly with n.
Christofides Algorithm	The direct distance between two cities is always less than or equal to the sum of the distances through a third city.	*O*(*n*3)	Where the triangle inequality holds.	Guarantees a 1.5 times approximation to the optimal. Easy to implement.	No guarantee of optimal solution.
Genetic Algorithm	Depends on population size and generations.	Depends on population size and generations	When an exact solution is not needed.	Provides a near optimal approximation.	No guarantee of optimality, solution Implementation is hectic.
Simulated Annealing	Probabilistic and depends on temperature decay and iterations.	Depends on temperature decay and iterations.	When an exact solution is not needed.	Less computational since they path with highest probability is considered.	No guarantee of optimality, solution Implementation is hectic.

Example showing complexity versus number of points.

N (Number of Cities)	Brute Force O(n!)	Dynamic Programming O(n^2·2^n)	Nearest Neighbor O(n^2)	Christofides O(n^3)	Branch and Bound (e^n)
1	1	1	1	1	1
2	2	4	4	8	2
3	6	36	9	27	6
4	24	384	16	64	24
5	120	4,800	25	125	120
6	720	61,440	36	216	720
7	5,040	823,680	49	343	5,040
8	40,320	10,485,760	64	512	40,320
9	362,880	134,217,728	81	729	362,880
10	3,628,800	1,717,986,918,400	100	1,000	3,628,800

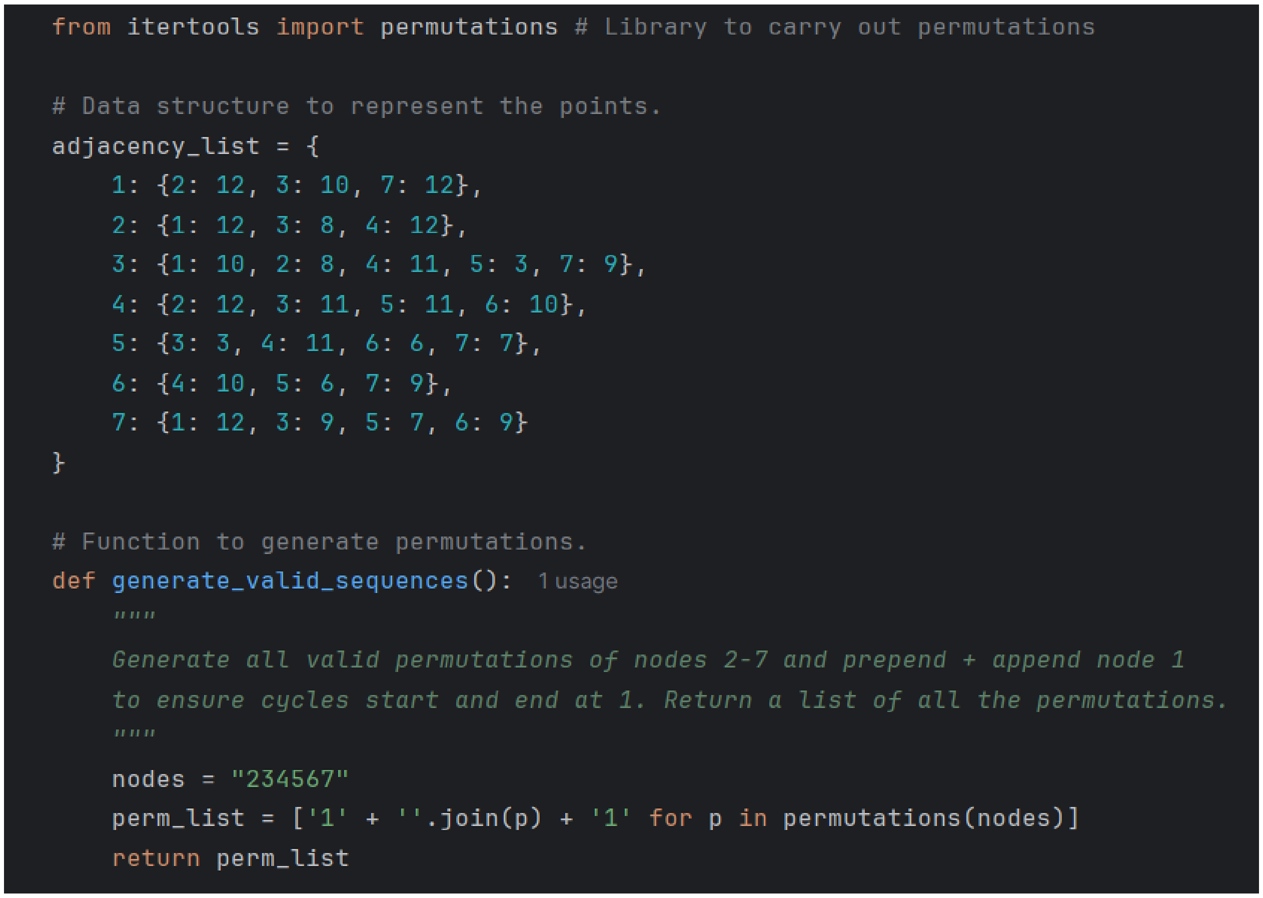
According to the statistical facts Exhaustive Search (Brute-Force) will be used because;

- It guarantees an exact, optimal path.
- Easy to implement.
- Practical for this dataset n = 7 points.
- Can be optimized to use up less memory easily.

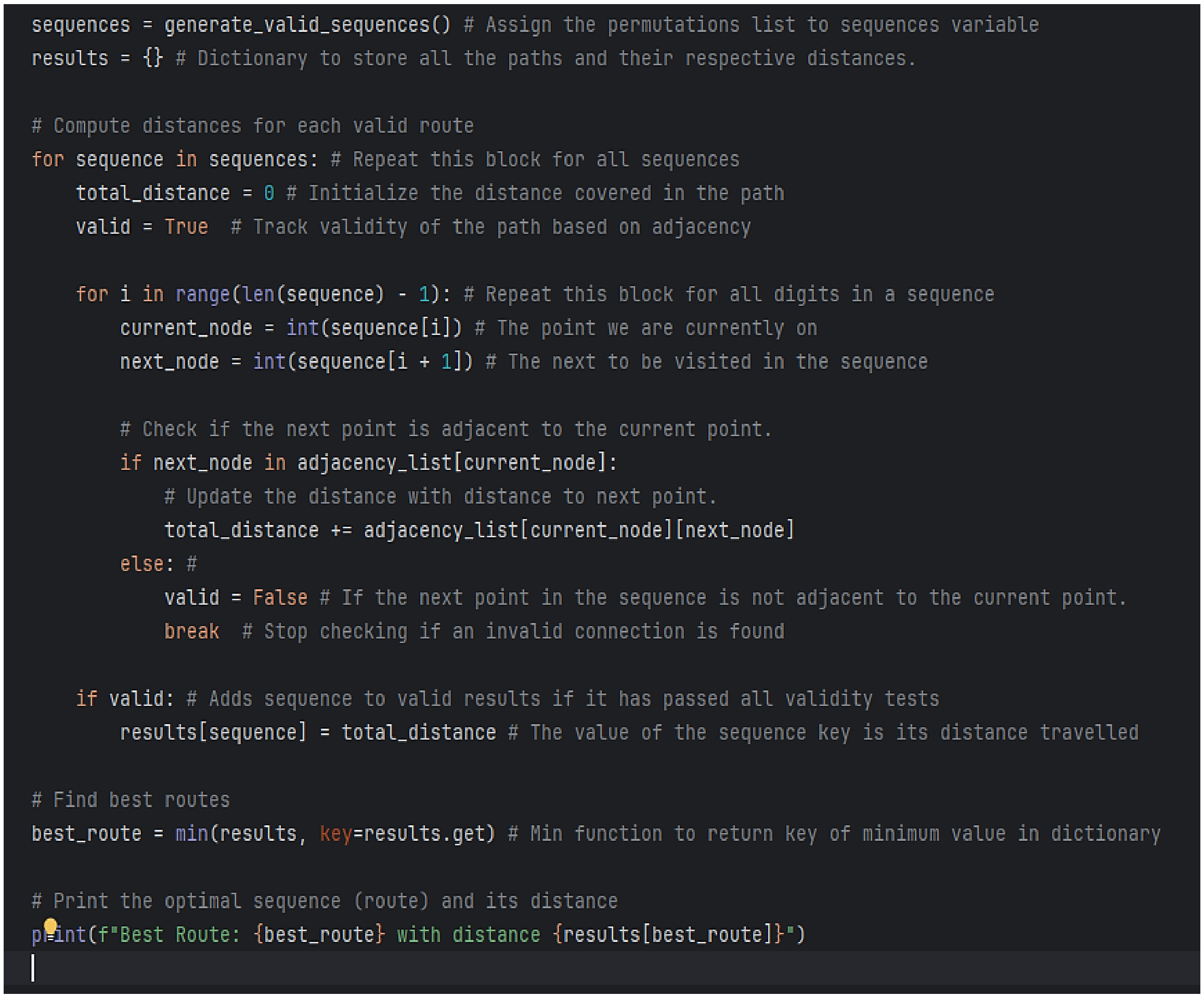
FINDINGS

How it works

- Rule 1: Visit all points.
    - Find all numbers with n-1 digits without 1 in them.
- Rule 2: Visit each point exactly once.
    - Eliminate all numbers with any repeating digits e.g. 223457
- Rule 3: Start and end at the same point.
    - Add 1 at the beginning and end of each number obtained to obtained all the possible paths.
- Rule 4: Shortest distance possible.
    - Use the adjacency list to calculate the distance of the current digit (current node) from the next digit in the sequence (next node) while checking that the next node is adjacent to the current node. If not, that path is discarded.



```python
from itertools import permutations # Library to carry out permutations

# Data structure to represent the points.
adjacency_list = {
    1: {2: 12, 3: 10, 7: 12},
    2: {1: 12, 3: 8, 4: 12},
    3: {1: 10, 2: 8, 4: 11, 5: 3, 7: 9},
    4: {2: 12, 3: 11, 5: 11, 6: 10},
    5: {3: 3, 4: 11, 6: 6, 7: 7},
    6: {4: 10, 5: 6, 7: 9},
    7: {1: 12, 3: 9, 5: 7, 6: 9}
}

# Function to generate permutations.
def generate_valid_sequences():  1 usage
    """
    Generate all valid permutations of nodes 2-7 and prepend + append node 1
    to ensure cycles start and end at 1. Return a list of all the permutations.
    """
    nodes = "234567"
    perm_list = ['1' + ''.join(p) + '1' for p in permutations(nodes)]
    return perm_list
```



```python
sequences = generate_valid_sequences() # Assign the permutations list to sequences variable
results = {} # Dictionary to store all the paths and their respective distances.

# Compute distances for each valid route
for sequence in sequences: # Repeat this block for all sequences
    total_distance = 0 # Initialize the distance covered in the path
    valid = True  # Track validity of the path based on adjacency

    for i in range(len(sequence) - 1): # Repeat this block for all digits in a sequence
        current_node = int(sequence[i]) # The point we are currently on
        next_node = int(sequence[i + 1]) # The next to be visited in the sequence

        # Check if the next point is adjacent to the current point.
        if next_node in adjacency_list[current_node]:
            # Update the distance with distance to next point.
            total_distance += adjacency_list[current_node][next_node]
        else: #
            valid = False # If the next point in the sequence is not adjacent to the current point.
            break  # Stop checking if an invalid connection is found

    if valid: # Adds sequence to valid results if it has passed all validity tests
        results[sequence] = total_distance # The value of the sequence key is its distance travelled

# Find best routes
best_route = min(results, key=results.get) # Min function to return key of minimum value in dictionary

# Print the optimal sequence (route) and its distance
print(f"Best Route: {best_route} with distance {results[best_route]}")
```

SELF-ORGANISING-MAPS (SOM'S) / KOHONEN MAPS

Self-Organizing Maps (SOMs), also known as Kohonen Maps, are unsupervised neural networks that use competitive learning to map high-dimensional data into a lower-dimensional grid. They preserve the topological relationships of the data, making them well-suited for clustering tasks such as the Traveling Salesman Problem (TSP). In this implementation, the SOM is applied to approximate the optimal route between cities using a force-directed approach to generate coordinates for each city.

SOMs are characterized by the following principles:

- Weights: Represent the coordinates of each output neuron. In the context of TSP, these represent the cities.
- Competitive Learning: Neurons compete for activation based on their proximity to input data points. The neuron closest to the data (the winner) adjusts its weight vector to better represent the data point.

The learning process consists of several phases:

1. Initialization: The data structures and training data are loaded, and neurons are initialized.
2. Competition: The "winner" neuron is determined by finding the neuron closest to the input data point.
3. Cooperation: The winner's topological neighbors are identified.
4. Adaptation: The weights of the winner and its neighbors are updated based on their proximity to the input data.
5. Continuation: The process repeats for a set number of iterations, adjusting the weights progressively until convergence.

Phases of the Learning Process in SOM

1. Initialization:

In this phase, the weight vectors for the neurons are initialized. The input cities' positions are first converted into 2D coordinates using a simplified force-directed approach. Then, the neurons (representing the cities) are randomly initialized within a small radius around the center of the cities.

The initial weight vectors for each neuron W_{ij} are calculated based on randomly sampled city positions from the adjacency matrix.

The initialization process also involves setting the number of neurons in the map, which is typically a multiple of the number of cities. The learning rate and number of iterations are also set during this phase.

2. Sampling:

In each iteration, a random city is selected from the input data (the cities' coordinates). This data point will be used to determine which neuron in the SOM is closest to the city. The weight vectors of neurons will then be adjusted based on this input.

3. Matching:

The matching phase determines the "winner" neuron, i.e., the neuron whose weight vector is closest to the input city. This is done by calculating the Euclidean distance between the city's coordinates and each neuron's weight vector. The neuron with the smallest distance is selected as the winner.

The Euclidean distance between the input city XXX and each neuron weight WWW is given by:

$$d = \sqrt{\sum_{i=1}^{n}(X_i - W_i)^2}$$

Where X_i and W_i are the coordinates of the input city and the neuron, respectively.

Once the winner neuron is identified, the topological neighbors of this neuron are determined. The neighbors are neurons that are geographically close to the winner in the map. The neighborhood function is calculated using a Gaussian distribution:

$$H(i*, i) = \exp\left(-\frac{||R_{i*} - R_i||^2}{2\sigma^2}\right)$$

Where:

- H(i∗,i) represents the influence of the neighborhood of neuron i∗ on neuron i.
- σ is the standard deviation of the weight vectors and controls the radius of influence of the neighborhood. It is calculated as:

$$\sigma = \text{np.std(weights)}$$

4. Updating:

In this phase, the weight vectors of the winner neuron and its neighbors are updated. The update rule for each neuron's weight vector is:

$$\Delta W_{ij} = \alpha \cdot H(i*, i) \cdot (X_j - W_{ij})$$

Where:

- ΔW_{ij} is the change in the weight of neuron i.
- α is the learning rate, which decays over time, allowing the SOM to converge smoothly. It is updated as:

$$\alpha(t) = \alpha_0 \cdot \exp\left(-\frac{t}{T}\right)$$

Where t is the current iteration, and T is a constant that controls the rate of decay of the learning rate.

During this phase, the topological neighborhood is updated in such a way that neurons closer to the winner are more strongly influenced by the input city. The neurons adjust their weight vectors based on the learning rate and the neighborhood influence.

5. Continuation / Convergence:

The algorithm repeats the competition, cooperation, and adaptation phases for a fixed number of iterations (or until convergence). As the iterations progress, the learning rate and neighborhood radius decrease, causing the SOM to fine-tune the weight vectors. The map converges when the weight vectors no longer change significantly.

The process continues until a defined number of iterations (epochs) is reached or the system has converged, meaning the weights and the map no longer change noticeably.

Application to the Traveling Salesman Problem (TSP)

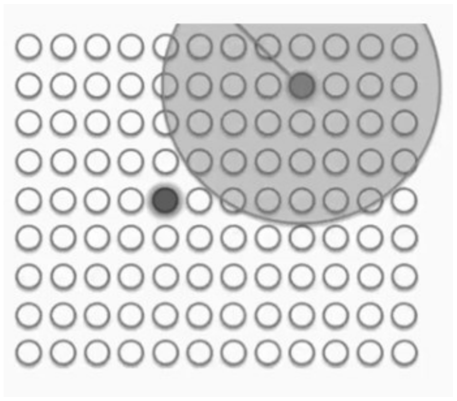
In this implementation of SOM for the Traveling Salesman Problem, the goal is to find the shortest possible route that visits each city exactly once and returns to the origin city. The SOM algorithm is applied to approximate this route by following these steps:

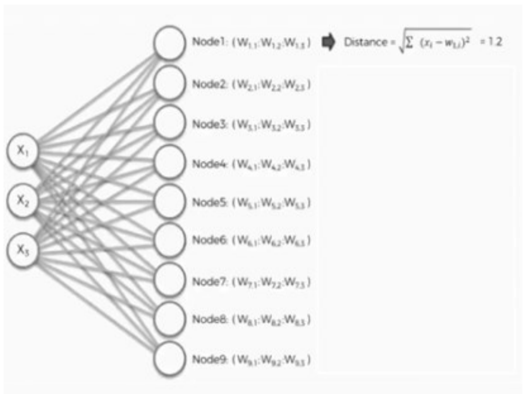
1. Convert the Adjacency Matrix to Coordinates: The adjacency matrix represents the distances between each pair of cities. Using a force-directed method, the cities are assigned initial 2D coordinates based on the adjacency matrix.
2. Train the SOM: The SOM is trained on the city coordinates. Over time, neurons adjust their positions to better represent the cities in a 2D map. As neurons adjust their positions, they become closer to the cities, and their weights become more representative of the cities' positions.
3. Determine the Optimal Route: After training the SOM, the winning neuron for each city is determined. The cities are ordered based on the neuron that wins for each city, which provides a route through the cities.
4. Calculate the Total Distance: Once the route is determined, the total distance of the route is calculated using the distances from the adjacency matrix. This step sums the distances between each pair of consecutive cities in the route, including the return to the starting city.

Conclusion

The Self-Organizing Map (SOM) is an effective method for solving the Traveling Salesman Problem (TSP). By using competitive learning and neighborhood cooperation, the SOM algorithm can approximate an optimal TSP solution. The process involves mapping cities to a 2D space, determining the best route through competitive learning, and refining the map over iterations. This approach offers a robust and flexible method for solving the TSP by adapting the map to the underlying structure of the cities' distances.

Illustration of feature(output) map showing winning neuron and its topological neighborhood and mapping.




Node1: $(W_{1,1}:W_{1,2}:W_{1,3})$
Distance = $\sqrt{\sum (x_i - w_{1,i})^2}$ = 1.2
Node2: $(W_{2,1}:W_{2,2}:W_{2,3})$
Node3: $(W_{3,1}:W_{3,2}:W_{3,3})$
Node4: $(W_{4,1}:W_{4,2}:W_{4,3})$
Node5: $(W_{5,1}:W_{5,2}:W_{5,3})$
Node6: $(W_{6,1}:W_{6,2}:W_{6,3})$
Node7: $(W_{7,1}:W_{7,2}:W_{7,3})$
Node8: $(W_{8,1}:W_{8,2}:W_{8,3})$
Node9: $(W_{9,1}:W_{9,2}:W_{9,3})$
X_1
X_2
X_3

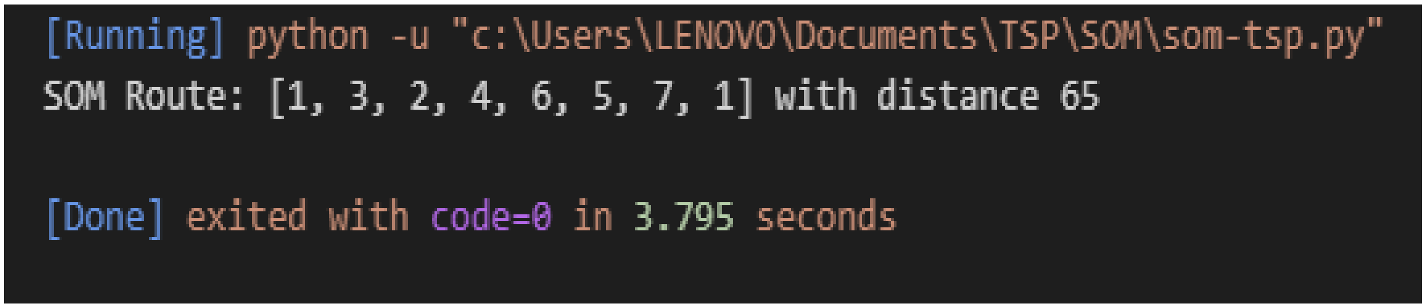
- The weights are similar to the inputs of the input space.
- Step by step, the winning neuron is mapped onto the feature map until the whole map is completely covered or all neurons are mapped.
- More tweaking of variables had to be done for the natural process to be more accurate.
- Still doesn’t converge to the desired optimal path, only an approximate sub-optimal path.

The pseudo code is shown below.

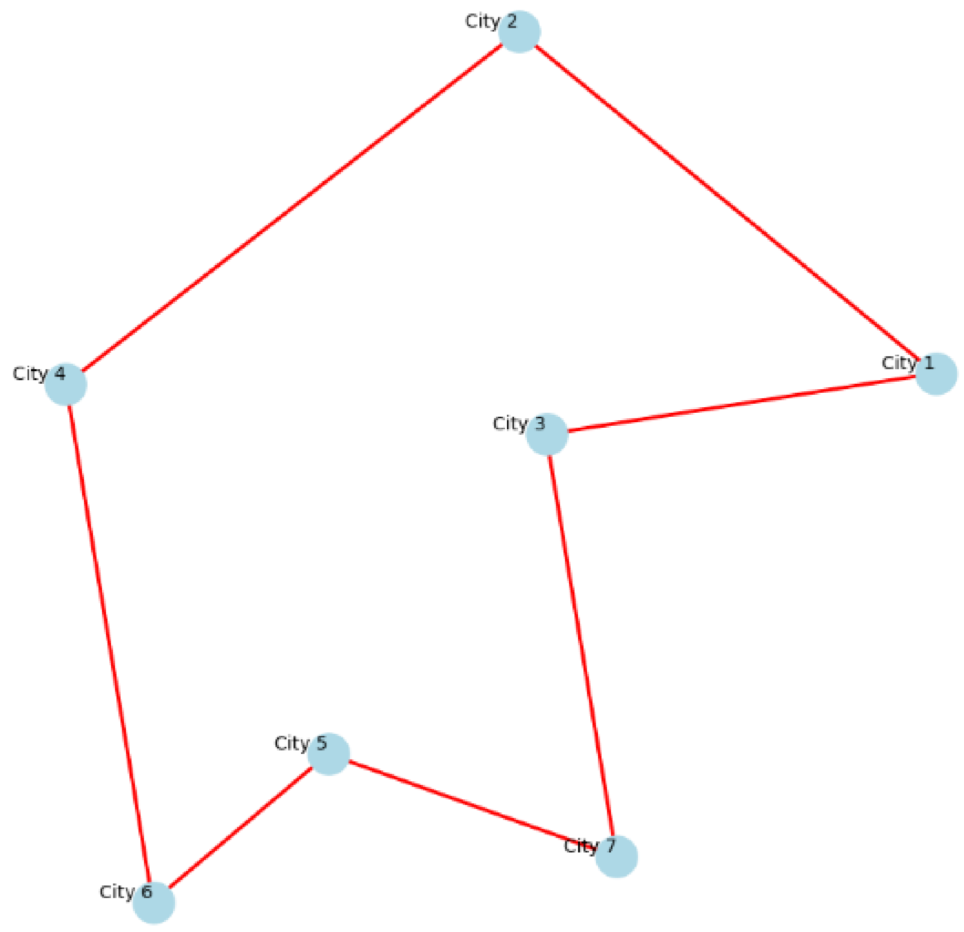
```
Define adjacency_matrix representing distances between cities.

FUNCTION Convert_Adjacency_To_Coordinates(adjacency_matrix):
    - Initialize cities in a circular 2D layout based on their count.
    - Refine the city coordinates by adjusting positions to better match the distance constraints using a force-directed method.
    - Return the optimized city coordinates.

CLASS SOM_TSP:
    - Initialize with city_coordinates (2D coordinates of cities), n_neurons (default based on the number of cities), and learning_rate.
    - Initialize neurons arranged in a circle around the cities.
    - Calculate center of cities and determine the radius for placing neurons in a circle.

FUNCTION Get_Winner(city_idx):
    - For the selected city, calculate the Euclidean distance to each neuron:
      distance = sqrt ((neuron_x - city_x) ^2 + (neuron_y - city_y) ^2)
    - Identify the neuron with the smallest distance, and return the index of the winning neuron.

FUNCTION Get_Neighborhood (winner_idx, iteration):
    - Calculate the radius of the neighborhood, which decreases over time with each iteration:
      radius = n_neurons / 2 * (1 - iteration / total_iterations)
    - Calculate the influence of each neuron using a Gaussian function, which is based on its distance from the winner neuron:
      neighborhood_influence(i) = exp (-(distance^2) / (2 * radius^2))
    - Return the list of neighborhood influences for each neuron.

FUNCTION Train_SOM():
    - For each training iteration:
        - Randomly select a city.
        - Find the winner neuron using Get_Winner(city_idx).
        - Calculate the neighborhood influences using Get_Neighborhood(winner_idx, iteration).
        - Update the neuron positions based on the city's coordinates and the neighborhood influences. Each neuron is adjusted according to:
          neuron_x = neuron_x + learning_rate * neighborhood_influence * (city_x - neuron_x)
          neuron_y = neuron_y + learning_rate * neighborhood_influence * (city_y - neuron_y)
```

```
FUNCTION Get_Best_Route():
    - After training, for each city, find the corresponding winning neuron.
    - Sort the cities based on their corresponding winner neuron indices to determine the optimal route.
    - Ensure the route starts and ends at the first city (city 0) to complete the cycle.

FUNCTION Calculate_Route_Distance(route):
    - For the best route, calculate the total distance by summing the distances between consecutive cities from the adjacency matrix:
        total_distance = sum(adjacency_matrix[route[i]][route[i+1]] for i in range (len (route) - 1))
    - Return the total distance of the route.

MAIN:
    - Convert the adjacency matrix to 2D city coordinates using Convert_Adjacency_To_Coordinates.
    - Initialize the SOM_TSP model with city_coordinates.
    - Train the SOM using Train_SOM to optimize the neuron positions and route.
    - Retrieve the best route using Get_Best_Route.
    - Calculate the total distance of the best route using Calculate_Route_Distance.
    - Output the best route and its total distance.
```

Training results



```
[Running] python -u "c:\Users\LENOVO\Documents\TSP\SOM\som-tsp.py"
SOM Route: [1, 3, 2, 4, 6, 5, 7, 1] with distance 65

[Done] exited with code=0 in 3.795 seconds
```


City 2
City 4
City 1
City 3
City 5
City 7
City 6

Limitations and Difficulties

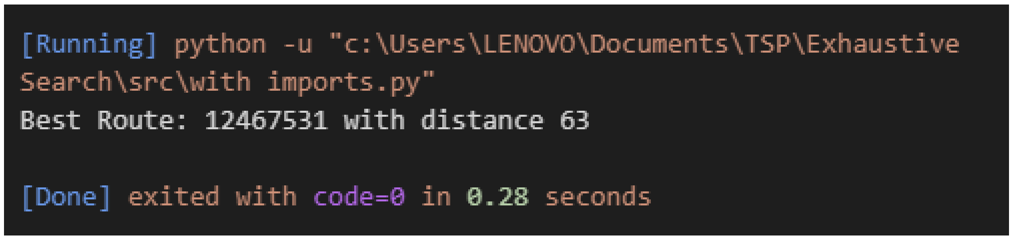
1. Sensitivity to Initial Setup:
   - The performance of SOM can be sensitive to the initial configuration of the neurons and learning rate. The initialization of neuron positions and the learning rate schedule significantly impact the convergence of the algorithm. If the setup is not chosen appropriately, the algorithm might converge to suboptimal solutions.
2. Computational Complexity:
   - While SOM is not inherently slow, the training process can become computationally expensive for very large datasets (in terms of both the number of cities and the number of neurons). The number of iterations required to fine-tune the map increases with the dataset size, which could cause the algorithm to take longer for large-scale problems. For small datasets like the one in the example, this issue is minimal.
3. Choice of Neighborhood Function:
   - The definition of the neighborhood function (which depends on the radius and standard deviation) influences the results. If the neighborhood radius is too large or too small, it might fail to capture meaningful relations between neurons, thus affecting the learning and the final arrangement of the cities.
4. Difficulty in Handling Large Datasets:
   - As the number of cities increases, the SOM may require an increasing number of neurons to represent the solution space effectively. This leads to increased computational cost and memory usage, particularly when the number of neurons exceeds the number of cities by a large factor.
5. Risk of Local Minima:
   - Like most unsupervised learning methods, SOM might be prone to converging to local minima. The algorithm's convergence behavior depends on the initialization of weights and the learning schedule, which could affect the quality of the solution (optimality of the route).
6. Limited Scalability:
   - SOM's performance can degrade as the problem size grows significantly (in terms of cities), and the algorithm may struggle to find optimal solutions quickly when scaling to larger TSP instances. The approach works well for smaller instances (as demonstrated in the code), but for very large TSP problems, it might need adjustments or more advanced heuristics.
7. Interpretability of Results:
   - The final map and non-linear process produced by SOM is often not as interpretable as other methods. While SOM can effectively find approximate solutions to TSP, understanding and explaining.

COMPARISON

Rate quality

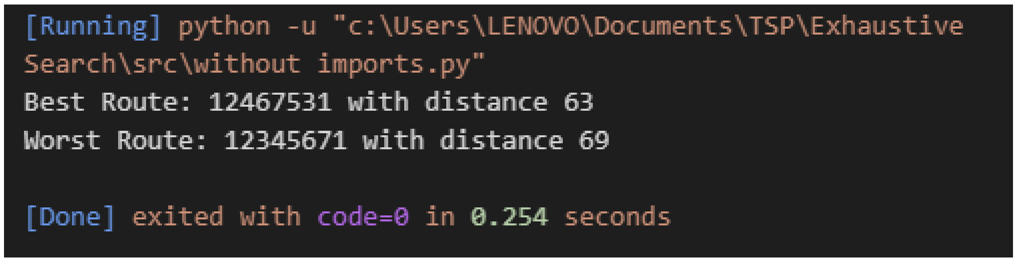
- Exhaustive Search is shorter to implement, run and adopt as opposed to SOM that needs multiple files, equations, functions and processes. Speed can be reduced by reducing number of iterations.

With imports



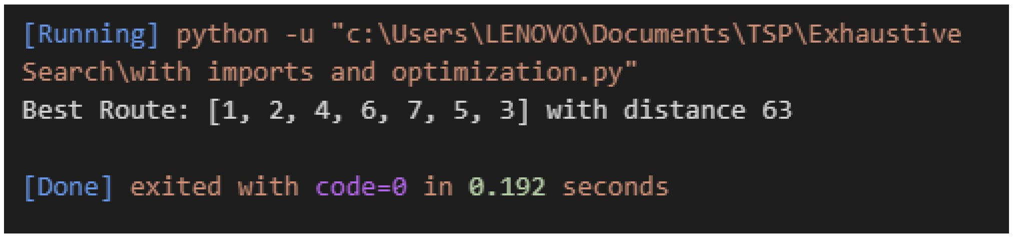
```
[Running] python -u "c:\Users\LENOVO\Documents\TSP\Exhaustive
Search\src\with imports.py"
Best Route: 12467531 with distance 63

[Done] exited with code=0 in 0.28 seconds
```

Without imports



```
[Running] python -u "c:\Users\LENOVO\Documents\TSP\Exhaustive
Search\src\without imports.py"
Best Route: 12467531 with distance 63
Worst Route: 12345671 with distance 69

[Done] exited with code=0 in 0.254 seconds
```

With optimizations



```
[Running] python -u "c:\Users\LENOVO\Documents\TSP\Exhaustive
Search\with imports and optimization.py"
Best Route: [1, 2, 4, 6, 7, 5, 3] with distance 63

[Done] exited with code=0 in 0.192 seconds
```

Complexity

- Exhaustive search is more complex with steps equal to 7! = 5040 while SOM complexity depends on the number of iterations used.

Time complexity

- high-level discussion of the computational cost of the SOM approach (number of iterations, updates per iteration, compare with exhaustive.

Consider a scenario of a mailman charged with delivering mail to 9 locations in Kampala.

- The problem size (**n=9**) is **small enough** for an exact algorithm.
- Since a mailman **must** find the best route, approximate methods may lead to inefficiencies.
- **SOMs require parameter tuning**, whereas brute-force guarantees correctness.
- **Exhaustive Search is easier to verify**, while SOMs rely on probabilistic learning.
- **Brute-Force** can act as a Benchmark when trying another method.

Extensions and optimizations for SOMs.

- While SOMs can approximate TSP solutions, they can be improved using:
  - **Adaptive Learning Rate:** Decay learning rate dynamically to refine weight updates.
  - **Neighborhood Function Tuning:** Modify the function to prevent early convergence to suboptimal paths.
  - **Hybrid Approaches:** Combine SOMs with local search techniques (e.g., 2-opt) to refine solutions.
  - **Parallel Processing:** Use GPU acceleration to speed up large-scale computations.
  - **Edge Weight Adjustments:** Dynamically adjust weights based on distance constraints.

Optimizations for Brute-Force Method.

- Although brute-force is infeasible for large datasets, it can be optimized using:
  - **Branch and Bound:** Prune unnecessary paths early based on cost constraints.
  - **Memoization:** Store intermediate results to prevent redundant calculations.
  - **Parallel Processing:** Utilize multi-threading or distributed computing to explore paths simultaneously.
  - **Bit masking and Dynamic Programming:** Reduce redundant re-computations in Held-Karp algorithm implementations.
  - **Preprocessing:** Sort adjacency lists to prioritize the shortest edges first.