THE TRAVELLING SALESMAN PROBLEM

By Group M

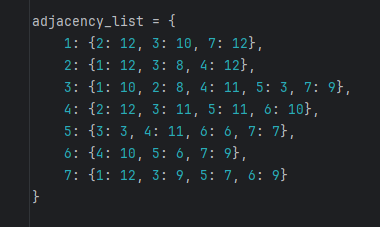
|  |  |  |
| --- | --- | --- |
| NAMES | REGISTRATION NUMBER | CONTACT |
| Mawande John Paul | 24/U/0678 | 787 130 130 |
| Econgu Paul | 24/U/22570 | 789 513 778 |
| Lyazi Patrick | 24/U/06571/PS | 782 726 299 |
| Namuli Angel Rebecca | 24/U/09332/PS | 756 055 966 |
| Namayanja Mary Prechard | 24/U/09046/PS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



**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;**

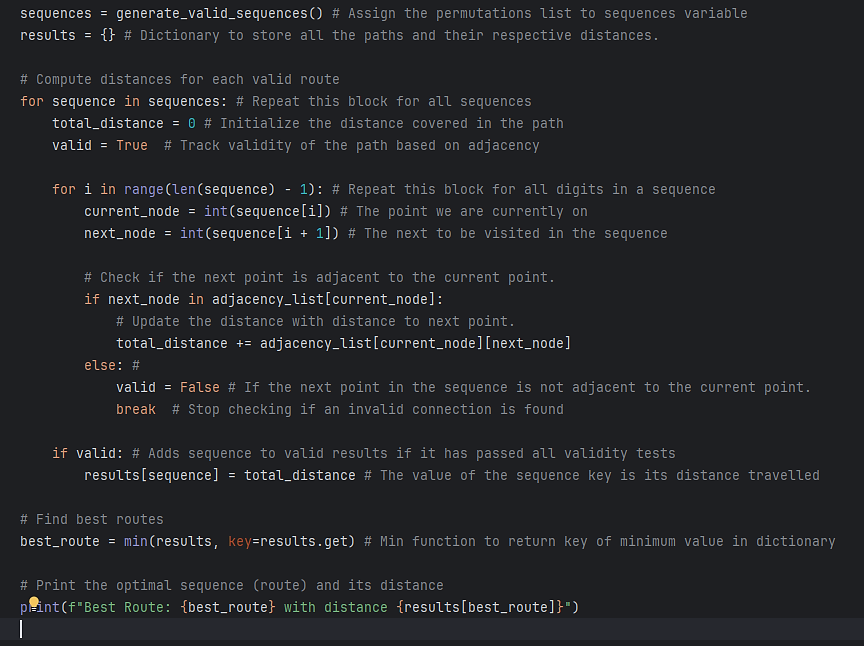
* It guarantees an exact, optimal path.
* Easy to implement.
* Practical for this dataset n = 7 points.
* Can be optimized to use up less memory easily.

**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.

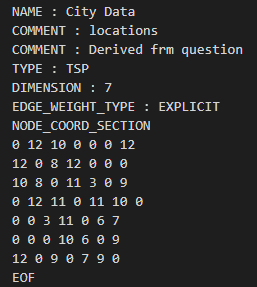




**SELF-ORGANISING-MAPS (SOM’S) / KOHONEN MAPS**

* Unsupervised neural network that clusters high dimensional data through creating a discretized version of its input space(map) in lower dimensions.
* Created by Teuvo Kohonen.
* They use competitive learning instead of error-correcting learning by using a function related to neighborhoods to preserve the topological properties of the map.
* They consist of the input layer, the weights and the output (Kohonen/feature/competitive layer)
* They are great for dimensional reduction, clustering, natural language processing, geology, astronomy and finance.
* Weights – Coordinates of each output neuron.
* Output neurons must compete to be activated, hence become the winner.

Before that algorithm must be used, the adjacency list is converted into an adjacence matrix, using np.zeros() then filling the distances. It is then saved in a .tsp file (standard file format for SOM input.)



**The learning scheme is comprised of;**

* Initialization
* Data structures, samples and training data is loaded
* Competition
* Winner is determined
* Cooperation
* Topological neighbor of winner is determined
* Adaptation
* The weights are updated to increase importance of neurons is updated
* Continuation
* Algorithm is repeated for a given number of iterations

**Phases of the learning process**

* Initialization
* Generate weights from the input map from equation through sampling.

Wij = Xij

Where W is a weight and X is a point selected randomly.

* Sampling
* Go through dataset and select samples randomly in each iteration from map.
* Matching
* Finding winner (with weight vector closest (shortest Euclidian distance, d) to the input sample) and its topological neighbors.

d = ||X - W|| = sqrt (sum ((Xi - Wi) ^2))

* The winner’s index is calculated from;

I(x) = argmin(d), a ‘NumPy’ function.

* Its topological neighborhood is calculated from;

H (i\*, i) = exp (-||Ri\* - Ri||^2 / (2\* σ^2))

Where σ is the standard deviation of the weights calculated from;

σ = np.std (array of weights) where np.std () is a ‘NumPy’ function

* Topological neighborhood: Radius around winner in which all neurons’ weights are updated.
* Updating
* Changing neuron weights of the winner and its topological neighbors based on a specific equation;

ΔWij = α \* H(i\*, i) \* (Xj - Xij) Where;

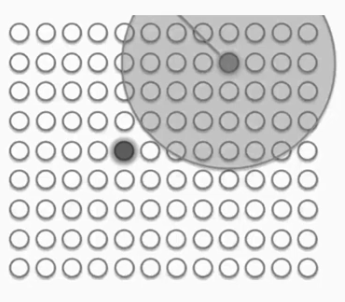
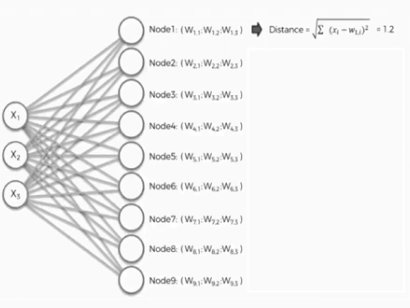
Learning rate, α = α(0) \* exp(-t / T) Where;

t is the current iteration and increases by one each iteration

T is the annealing constant that controls how fast the learning rate decays.

* T can be any value of choice e.g. with T=100, decay is slower than T=1000.
* Learning rate: Refers to how fast the weights are updated.
* Topological neighborhood and learning rate decay as iterations increase.
* Continuation / Convergence
* Repeat the same algorithm until the feature map remains the same or until the number of iterations (epoch t) is reached.

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

* 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.

**PSEUDO CODE**

Initialize a 2D grid of neurons representing possible city positions

Set learning parameters (learning rate, neighborhood radius, number of iterations)

Randomly distribute neurons in a continuous space that covers the cities

FOR each iteration:

    FOR each city in the adjacency list:

        - Find the winning neuron/Best Matching Unit (BMU): The neuron closest to the current city.

        - Update the BMU and its neighbors:

        - Move the BMU towards the city.

        - Adjust neighboring neurons based on the neighborhood radius.

        - Reduce the learning rate and neighborhood radius over time.

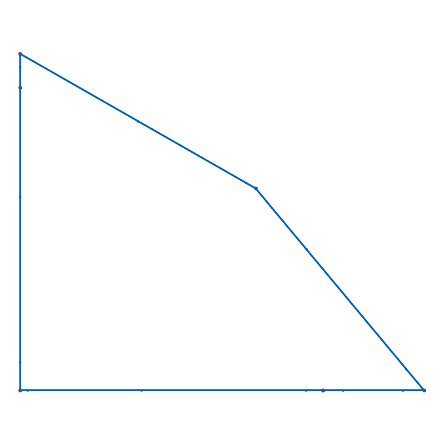
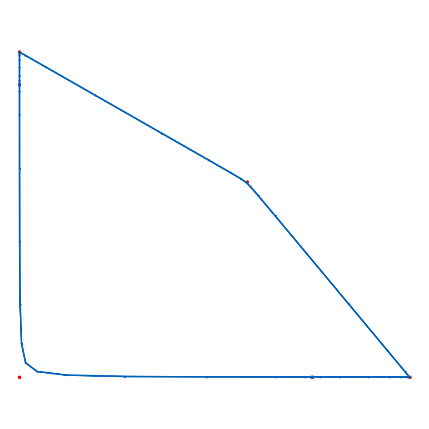
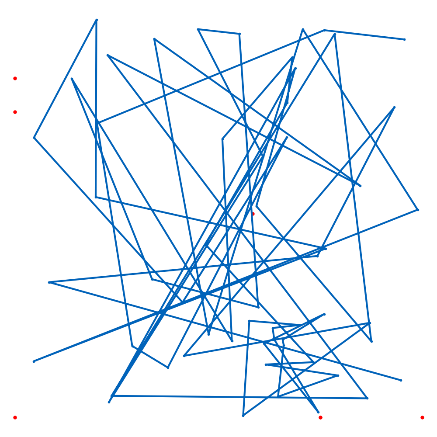
Sort the neurons based on their positions to form a closed-loop path

Map the cities to the nearest neurons to create a tour sequence

Calculate the total distance of the generated tour using the adjacency list

Return the optimized route and its total distance

**Training results**

****

**Limitations and difficulties**

* Data must be converted to standard (.tsp) format for input into the algorithm.
* Implementation is not straight forward.
* Slow and computationally expensive (not suitable for small datasets).

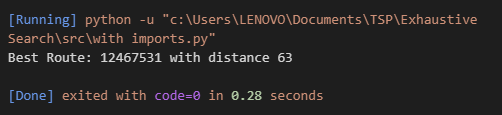
**COMPARISON**

**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

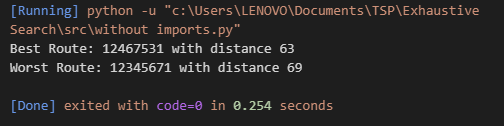
With imports





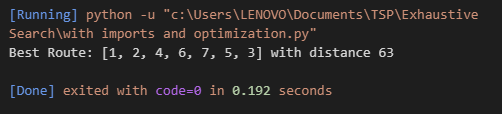
Without imports





With optimizations





**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.