THE TRAVELLING SALESMAN PROBLEM

This is a real-world problem that involves a fixed set of points, all must be traversed and return to the starting point, without reaching a point twice and taking the shortest route possible.

Input: Data structure

Points are stored as a dictionary where the;

Keys: All the individual points.

Values: Dictionaries containing the keys as cities adjacent to the key(point) and their respective distances from the current 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.

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

Solution;

Two types of solutions (algorithms) are available;

Exact (Non-Heuristic)

These guarantee a valid best path.

Heuristic (Approximate)

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

Heuristic

Nearest-Neighbour

Genetic Algorithm

Simulated Anealing

Christofides Algorithm (Best approximation technique)

Self-Organising-Maps

Exact

Brute-force (Exhaustive Search)

Dynamic Programming (Held-Karp)

Branch and Bound

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.

Algorithm	**Description**	**Complexity**	Suitability	**Pros**	**Cons**
Dynamic Programming	Solves optimally by breaking the problem into subproblems	$O(n2 \cdot 2n)$	20-30 cities	Optimal solution, avoids recomputation	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	$O(n2)$	Large problems needing	Fast, simple to implement	No optimality guarantee, can get stuck in local minima

			quick solutions		
Self-Organizing Map (SOM)	Neural network clustering approach, useful for approximation	Varies (depends on grid size and iterations)	Large problems,	Scalable, good for large datasets higer 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	Always 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	$O(n3)$	where the triangle inequality holds	Guarantees a 1.5 times approximation to the optimal. Easy to implement. Most accurate approximation.	No guarantee of optimality, 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**n* (Number of Cities)	**Brute Force O(n!) *O*(*n*!)**	**Dynamic Programming O(n2·2n)*O*(*n*2·2*n*)**	**Nearest Neighbor O(n2) *O*(*n*2)**	**Christofides O(n3) *O*(*n*3)**	**Branch and Bound (Exponential)**
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.

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.

script1.py

Source code:

```
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)

Unsupervised neural network that clusters high dimensional data through creating a discretized version of its input space(map) in lower dimensions.

They use competitive learning instead of error-correcting learning by using a function related to neighbourhoods 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.

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 of weights with random values

Sampling

Go through dataset and select samples randomly in each iteration from map

Matching

Finding winner (with weight vector closest (shortest eucledian distance) to the input sample) and its topological neighbours

Updating

Changing neuron weights of the winner and its topological neighbours based on a specific equation

$$\Delta w_{ij} = \eta(t).T_{j,I(x)}(t).(x_i - w_{ji})$$
$$\eta(t) = \eta_0.exp(\text{-}t/\tau_\eta)$$
$$T_{j,I(x)} = exp(-S_{j,I(x)}^2/2\sigma^2)$$
$$\sigma(t) = \sigma_0\ exp(-t/T_\sigma)$$

Where;

Dw – weight change of the neuron.

- η: Learning rate
- T: Topological neighborhood
- σ: Size of the neighborhood
- S: Lateral distance between points in the feature map
- $I(x)$: Index of the winner
- t: The epoch
- Others: Hyperparameters

Topological neighbourhood 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 neighbourhood

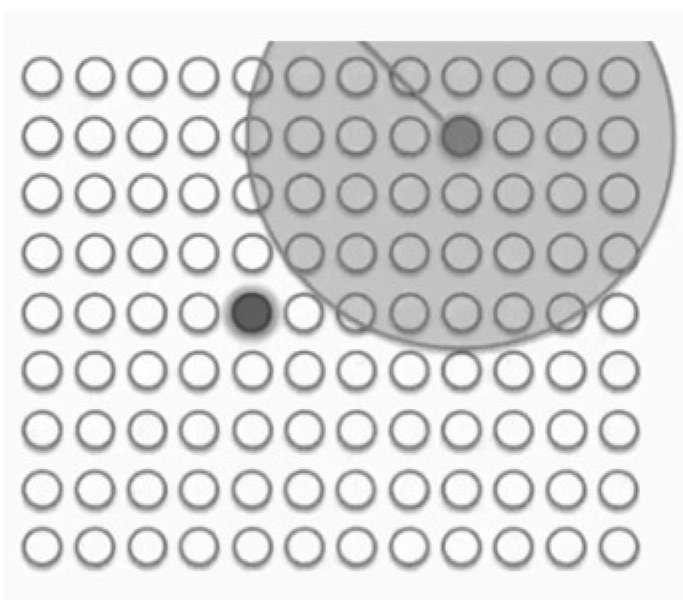
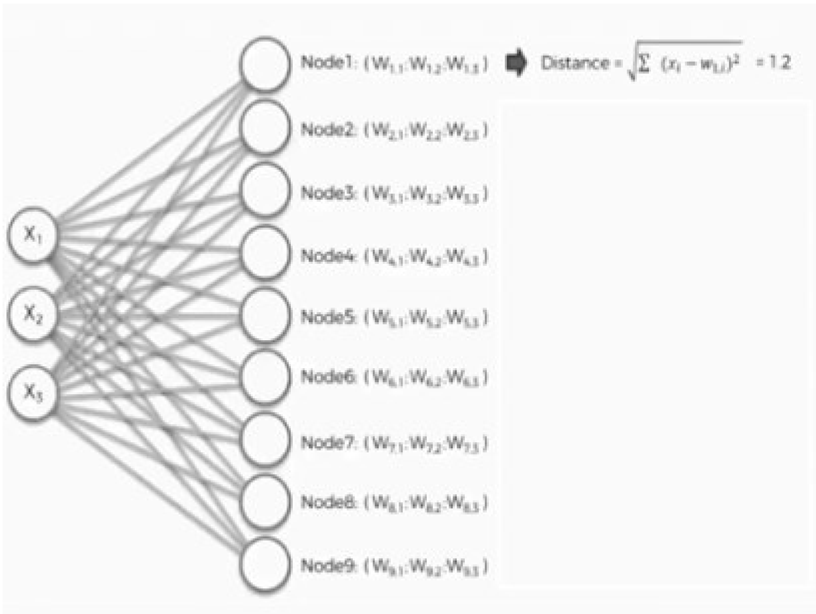


Illustration of how mapping occurs



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.

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

[add screenshots of speed measurements in console]

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 in Kampala.

Continue

Extensions