Code:

```
import copy
from typing import Union
inf = float('inf')
class TSP_AI:
    """Traveling Salesman Problem
        ----------------------------------------------------------------
        Traveling Salesman Problem using Nearest Neighbour AI algorithm
    """


    def __init__(
        self,
        matrix: list[list[Union[int, float, str]]] = None,
        source: int = 0
    ) -> None:
        """TSP
            ===
            Parameters
            ----------
            1. matrix (Default 4x4 Matrix of Zeros) : User can provide the matrix beforehand
            2. source (Default 0) : The index of the city which should be considered as source
        """
        self.matrix = [[0]*4]*4 if matrix is None else matrix
        self.n: int = len(self.matrix)
        self.source: int = source
```

```
def Input(self):
    self.n = int(input('Enter city count : '))

    # Get the distances between cities
    for i in range(self.n):
        self.matrix.append([
            inf if i == j else int(
                input(f'Cost to travel from city {i+1} to {j+1} : '))
            for j in range(self.n)
        ])

    # Get the source city
    self.source = int(input('Source: ')) % self.n

def solve(self):
    # Initially minCost is infinity
    minCost: float = inf
    for i in range(self.n):
        print("Path", end='')
        # Calling solver for each as source city
        cost = self._solve(copy.deepcopy(matrix), i, i)
        print(f" -> {i+1}    :    Cost = {cost}")
        if cost and cost < minCost:
            minCost = cost                # If this cost is optimal, save it

    return minCost
```

```
    def _solve(
        self,
        matrix: list[list[Union[int, float, str]]],
        currCity: int = 0,
        source: int = 0
    ) -> int:


        if self.n < 2:
            return 0
        print(f" -> {currCity+1}", end='')


        for i in range(self.n):
            # Set all values in the currCity column as infinity (once visited, shouldn't be visited anymore)
            matrix[i][currCity] = inf


        currMin, currMinPos = inf, 0
        for j in range(self.n):
            # Get the nearest city to the current city
            if currMin > matrix[currCity][j]:
                currMin, currMinPos = matrix[currCity][j], j


        if currMin == inf:
            # If currMin is infinity(i.e. all cities have been visited, return cost of moving from this last city to start city to complete the path-loop)
            return self.matrix[currCity][source]
        # Set distance from currCity to next city and vice versa to infinity
```

```
        matrix[currCity][currMinPos] = matrix[currMinPos][currCity] = inf
        # Calling the next recursion for selected city
        return currMin + self._solve(matrix, currMinPos, source)
# Driver code
if __name__ == '__main__':
    matrix = [
        [inf,  10,   15,    20,   299,  67,   24],
        [10,   inf,  35,    25,   5,    88,   44],
        [15,   35,   inf,   30,   52,   454,  13],
        [20,   25,   30,    inf,  139,  23,   89],
        [299,   5,   52,    139,   inf,  93,   10],
        [67,   88,   454,  23,    93,   inf,  89],
        [24,   44,   13,    89,    10,   89,   inf],
    ]

    source_city = 0
    tsp = TSP_AI(matrix, source_city)
    print(f"Optimal Cost : {tsp.solve()}")
```

Output:

```
...   Path -> 1 -> 2 -> 5 -> 7 -> 3 -> 4 -> 6 -> 1    :    Cost = 158
      Path -> 2 -> 5 -> 7 -> 3 -> 1 -> 4 -> 6 -> 2    :    Cost = 174
      Path -> 3 -> 7 -> 5 -> 2 -> 1 -> 4 -> 6 -> 3    :    Cost = 535
      Path -> 4 -> 1 -> 2 -> 5 -> 7 -> 3 -> 6 -> 4    :    Cost = 535
      Path -> 5 -> 2 -> 1 -> 3 -> 7 -> 4 -> 6 -> 5    :    Cost = 248
      Path -> 6 -> 4 -> 1 -> 2 -> 5 -> 7 -> 3 -> 6    :    Cost = 535
      Path -> 7 -> 5 -> 2 -> 1 -> 3 -> 4 -> 6 -> 7    :    Cost = 182
      Optimal Cost : 158
```