```python
import itertools
import random
import sys
import math

"""
credit to https://github.com/CarlEkerot

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:
"""

def held_karp(dists):
    """
    Implementation of Held-Karp, an algorithm that solves the Traveling
    Salesman Problem using dynamic programming with memoization.

    Parameters:
        dists: distance matrix

    Returns:
        A tuple, (cost, path).
    """
    n = len(dists)

    # Maps each subset of the nodes to the cost to reach that subset, as well
    # as what node it passed before reaching this subset.
    # Node subsets are represented as set bits.
    C = {}

    # Set transition cost from initial state
    for k in range(1, n):
        C[(1 << k, k)] = (dists[0][k], 0)

    # Iterate subsets of increasing length and store intermediate results
    # in classic dynamic programming manner
    for subset_size in range(2, n):
        for subset in itertools.combinations(range(1, n), subset_size):
            # Set bits for all nodes in this subset
            bits = 0
            for bit in subset:
                bits |= 1 << bit

            # Find the lowest cost to get to this subset
            for k in subset:
                prev = bits & ~(1 << k)

                res = []
                for m in subset:
                    if m == 0 or m == k:
                        continue
                    res.append((C[(prev, m)][0] + dists[m][k], m))
                C[(bits, k)] = min(res)

    # We're interested in all bits but the least significant (the start state)
    bits = (2**n - 1) - 1

    # Calculate optimal cost
    res = []
    for k in range(1, n):
        res.append((C[(bits, k)][0] + dists[k][0], k))
    opt, parent = min(res)

```

```
    # Backtrack to find full path
    path = []
    for i in range(n - 1):
        path.append(parent)
        new_bits = bits & ~(1 << parent)
        _, parent = C[(bits, parent)]
        bits = new_bits

    # Add implicit start state
    path.append(0)

    return opt, list(reversed(path))


def generate_distances(n):
    dists = [[0] * n for i in range(n)]
    for i in range(n):
        for j in range(i+1, n):
            dists[i][j] = dists[j][i] = random.randint(1, 99)

    return dists


def read_distances(filename):
    dists = []
    with open(filename, 'rb') as f:
        for line in f:
            # Skip comments
            if line[0] == '#':
                continue

            dists.append(map(int, map(str.strip, line.split(','))))

    return dists


if __name__ == '__main__':
    """
    ### uncomment to generate data given arg n
    arg = sys.argv[1]

    if arg.endswith('.csv'):
        dists = read_distances(arg)
    else:
        dists = generate_distances(int(arg))
    """

    inf = math.inf
    dists = [[0, 14,  9, 7, inf, inf, inf, inf, inf, inf],
             [14, 0, 14, inf, inf, inf, inf, inf, inf, inf],
             [9, 14, 0, 10, inf, inf, inf, inf, inf, inf ],
             [7, inf, 10, 0, 9, 15, inf, inf, inf, inf ],
             [inf, 7, inf, 9, 0, 6, 9, inf, 11, inf],
             [inf, inf, inf, 15, 6, 0, inf, 9, inf, inf],
             [inf, 9, inf, inf, 9, inf, 0, inf, 9, inf],
             [inf, inf, inf, inf, inf, 9, inf, 0, 6, 11],
             [inf, inf, inf, inf, 11, inf, 9, 6, 0, 9],
             [inf, inf, inf, inf, inf, inf, inf, 11, 9, 0]];

    # Pretty-print the distance matrix
    for row in dists:
        print(''.join([str(n).rjust(4, ' ') for n in row]))

    print('')

    print(held_karp(dists))

```