**Traveling Salesman Problem TSP with DP**

**Iheb Gafsi**\*

INSAT Student

**Iheb.engineer@gmail.com**

**Definition:**

The Traveling Salesman Problem (TSP) is a well-known optimization problem in graph theory and combinatorial optimization. The goal of the TSP is to find the shortest possible route that visits a given set of cities exactly once and returns to the starting city. The problem is NP-hard, which means that there is no known polynomial-time algorithm to solve it for large instances. However, there are several heuristic and approximation algorithms that can find reasonably good solutions in a reasonable amount of time for practical instances.

One of the most popular algorithms for solving the TSP is the Held-Karp algorithm, also known as the dynamic programming algorithm which is what we’re going to implement in this document. The Held-Karp algorithm works by breaking the problem into smaller subproblems and solving them recursively. It maintains a table to store the optimal solutions to these subproblems and uses them to build the final solution for the original problem. The time complexity of the Held-Karp algorithm is , where n is the number of cities in the TSP instance. While this is an improvement over the naive brute-force approach (which has a time complexity of O(n!), the Held-Karp algorithm is still not efficient enough for large instances.

**Use cases:**

The Traveling Salesman Problem has numerous real-world applications in various fields, including logistics, transportation, and network design. For example, in logistics, it can be used to optimize delivery routes for a fleet of vehicles to minimize travel time and costs. In transportation, it can help plan efficient routes for public transport systems. In network design, it can be used to find the shortest path for data transmission in communication networks. The TSP also has applications in manufacturing, where it can be used to optimize the order in which different tasks are performed to minimize production time. Despite its computational complexity, the TSP's wide range of applications makes it an essential problem to study and develop efficient approximation algorithms to find near-optimal solutions in practical scenarios.

**Algorithm:**

1. #Variables and constants

2. adj\_matrix = adjacency matrix

3. INF = float('inf')

4. # This recursive method is used in the next function to create bit sets

5. def combinations(s, i, r, n, subs):

6.     if n - i < r: return

7.     if r == 0:

8.         subs.append(s)

9.     else:

10.         for j in range(i, n):

11.             s = s | 1<<j

12.             combinations(s, i+1, r-1, n, subs)

13.             s = s & ~(1<<j)

14. # returns all combinations of size N where there are r bits set to 1

15. # subsets(3, 4) = [0111, 1011, 1101, 1110]

16. def subsets(r, n):

17.     subs = []

18.     combinations(0, 0, r, n, subs)

19.     return subs

20. # ith bit in subset == 0 ?

21. def izero(i, subset):

22.     return ((1<<i) & subset) == 0

23. def tsp(graph, S):

24.     n = len(graph)

25.     # initialize table with null or -1 or +inf to prevent errors

26.     dp = [ [None] \* (1<<n) for \_ in range(n) ]

27.     # Catch the optimal solution from start node to others

28.     for i in range(n):

29.         if i == S: continue

30.         # Store the optimal value from node S to each node i

31.         dp[i][ 1<<S | 1<<i ] = graph[S][i]

32.     # solve

33.     for r in range(3, n+1):

34.         for subset in subsets(r, n):

35.             if izero(S, subset): continue

36.             for next in range(n):

37.                 if next==S or izero(next, subset): continue

38.                 #subset state without next node

39.                 state = subset ^ (1<<next)

40.                 minDist = INF

41.                 for e in range(n):

42.                     if e == S or e == next or izero(e, subset): continue

43.                     minDist = min(dp[e][state] + graph[e][next], minDist)

44.                 dp[next][subset] = minDist

45.     # Find minimum cost

46.     # the end state is the bit mask with n bits set to 1

47.     END\_STATE = (1<<n)-1

48.     mc = INF

49.     for e in range(n):

50.         if e==S: continue

51.         mc = min(dp[e][END\_STATE] + graph[e][S], mc)

52.     # Now Just find the optimal tour

53.     lastindex = S

54.     state = END\_STATE

55.     tour = [S]

56.     for i in range(1, n):

57.         index = -1

58.         for j in range(n):

59.             if j==S or izero(j, state): continue

60.             if index==-1: index = j

61.             prev = dp[index][state] + graph[index][lastindex]

62.             new = dp[j][state] + graph[j][lastindex]

63.             if new < prev: index = j

64.         tour.append(index)

65.         state = state ^ (1<<index)

66.         lastindex = index

67.     tour.append(S)

68.     tour.reverse()

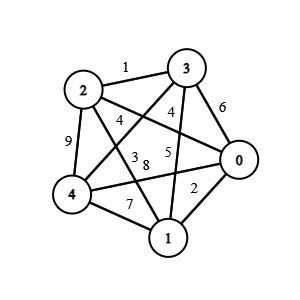
69.     return (mc, tour)

70.

71. print(tsp(adj\_matrix, 0))

**Example:**

Here’s a small example illustrating an example of input outputs for the Held-Karp Algorithm:



We will use the Python code down below to outline the output of the algorithm on this graph:

1. #Variables and constants

2. adj\_matrix = [

3.     [0, 2, 4, 6, 8],

4.     [2, 0, 3, 5, 7],

5.     [4, 3, 0, 1, 9],

6.     [6, 5, 1, 0, 4],

7.     [8, 7, 9, 4, 0]

8. ]

9. INF = float('inf')

10. # This recursive method is used in the next function to create bit sets

11. def combinations(s, i, r, n, subs):

12.     if n - i < r: return

13.     if r == 0:

14.         subs.append(s)

15.     else:

16.         for j in range(i, n):

17.             s = s | 1<<j

18.             combinations(s, i+1, r-1, n, subs)

19.             s = s & ~(1<<j)

20. # returns all combinations of size N where there are r bits set to 1

21. # subsets(3, 4) = [0111, 1011, 1101, 1110]

22. def subsets(r, n):

23.     subs = []

24.     combinations(0, 0, r, n, subs)

25.     return subs

26. # ith bit in subset == 0 ?

27. def izero(i, subset):

28.     return ((1<<i) & subset) == 0

29.

30. def tsp(graph, S):

31.     n = len(graph)

32.     # initialize table with null or -1 or +inf to prevent errors

33.     dp = [ [None] \* (1<<n) for \_ in range(n) ]

34.     # Catch the optimal solution from start node to others

35.     for i in range(n):

36.         if i == S: continue

37.         # Store the optimal value from node S to each node i

38.         dp[i][ 1<<S | 1<<i ] = graph[S][i]

39.

40.

41.

42.

43.

44.

45. # solve

46.     for r in range(3, n+1):

47.         for subset in subsets(r, n):

48.             if izero(S, subset): continue

49.             for next in range(n):

50.                 if next==S or izero(next, subset): continue

51.                 #subset state without next node

52.                 state = subset ^ (1<<next)

53.                 minDist = INF

54.                 for e in range(n):

55.                     if e == S or e == next or izero(e, subset): continue

56.                     minDist = min(dp[e][state] + graph[e][next], minDist)

57.                 dp[next][subset] = minDist

58.     # Find minimum cost

59.     # the end state is the bit mask with n bits set to 1

60.     END\_STATE = (1<<n)-1

61.     mc = INF

62.     for e in range(n):

63.         if e==S: continue

64.         mc = min(dp[e][END\_STATE] + graph[e][S], mc)

65.     # Now Just find the optimal tour

66.     lastindex = S

67.     state = END\_STATE

68.     tour = [S]

69.     for i in range(1, n):

70.         index = -1

71.         for j in range(n):

72.             if j==S or izero(j, state): continue

73.             if index==-1: index = j

74.             prev = dp[index][state] + graph[index][lastindex]

75.             new = dp[j][state] + graph[j][lastindex]

76.             if new < prev: index = j

77.         tour.append(index)

78.         state = state ^ (1<<index)

79.         lastindex = index

80.     tour.append(S)

81.     tour.reverse()

82.     return (mc, tour)

83. print(tsp(adj\_matrix, 0))

The corresponding output is:

Python >> (18, [0, 1, 4, 3, 2, 0])

