



# VIT<sup>®</sup>

## Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

### CSE2003

## DATA STRUCTURES AND ALGORITHMS

### Embedded Project J Component

### Fall Semester 2020-21

Slot: C1

Professor Kauser Ahmed. P

## The Travelling Salesman Problem

### Review II

19BCE2249

19BCE2250

Siddharth Chatterjee

Ishan Sagar Jogalekar

# Abstract

In the previous Review, (Review I), we described how the principal goal of our project is to understand and apply different Travelling Salesman Problem algorithms (to be referred to as TSP from now onwards) to solve real world scenarios. For practical implementation what we will do is, optimize current TSP algorithms to reduce computation time and propose a novel and more effective answer to the famous question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?"

## Aim

The aim of our project is to consider a ground level scenario where a company 'Surakshit' has supplied a large number of RO devices to retailers and distributors over a certain geographical area and according to the service agreements, the company has to attend to service requests and to carry out mandatory service visits to each and every unit that is bought and installed. Due to stiff competition for supply and installation of RO Devices with other companies in the same market, an optimum route is to be constructed to minimize the cost of the services with a well-planned schedule to carry out services efficiently.

## Objective

The objectives of our project are:

- a) To come up with working algorithm for the above said problem scenario
- b) To make the algorithm precise and functional
- c) To fine-tune the algorithm and hence reduce the time complexity and space complexity.
- d) To try and implement the program using different Data Structures so as to come up with the best one.

For example, we will start with devising 'exact algorithms', which work reasonably fast for such problem statements accounting a small geographical area where RO water filters need to be serviced.

## Applicability

What applicability basically means, in this context, is the quality of being appropriate and relevant. As already discussed in 'Aim' of our project; we will start with devising 'exact algorithms', which work reasonably fast for such problem statements accounting a small geographical area where RO water filters need to be serviced for optimal TSP solution. After Exact Algorithms (which consist of implementations from Branch and Bound method to Dynamic Programming for Held-Karp method) we will move on to some Heuristic Algorithms to compare them, and provide our final conclusion.

Hence, this project basically elaborates on existing solutions to TSP and finally a formal answer structure to the scenario of 'Surakshit'.

# Introduction

Briefly, there are 4 different types of algorithms with various methods in them to solve the TSP problem.

## Different Approaches for Solving TSP

### ➤ Exact

- Enumeration
- Linear Programming
  - Miller-Tucker-Zemlin (1960)
- Dynamic Programming

### ➤ Heuristics

- Simulated Annealing
- Genetic Algorithm
- Ant Colony Optimization
- Electromagnetism like Algorithm

### ➤ Approximation Algorithms

- Nearest Neighbour
- Greedy
- K-Opt, Kernighan Lin (70's)
- Christofides (1965)

### ➤ Lower Bound

- Held-Karp

Any TSP consists of determining a minimum distance circuit passing through each vertex once and only once. Such a circuit is known as a tour or Hamiltonian circuit (or cycle).

Let  $G = (V, A)$  be a graph where  $V$  is a set of  $n$  vertices.  $A$  is a set of arcs or edges, and let  $C: (C_{ij})$  be a cost or time travel matrix associated with  $A$ . Now, a TSP can be modelled as an undirected weighted graph, such that cities are the graph's vertices, paths are the graph's edges, and a path's distance is the edge's length.

Often, the model is a complete graph (i.e. each pair of vertices is connected by an edge). If no path exists between two cities, adding an arbitrarily long edge will complete the graph without affecting the optimal tour.

We can try all permutations (ordered combinations) and see which one is cheapest (using brute-force search algorithm). The running time for this approach lies within a polynomial factor of  $O(n!)$ , the factorial of the number of cities, so this solution becomes impractical even for only 20 cities. Therefore, this is something we will not go forward with.

Now, instead of brute-force algorithm; using dynamic programming (DP) approach the solution can be obtained in lesser time, though there is no polynomial time algorithm. One of the earliest applications of dynamic programming is the Held-Karp algorithm that solves the problem in time  $O(n^2 2^n)$ . This is the Lower Bound algorithm.

The project plan will also include some other approaches to compare the time complexity and determine the efficiency of each one of them:

- Various branch-and-bound algorithms, which can be used to process TSPs containing 40-60 cities.

- Progressive improvement algorithms which use techniques reminiscent of linear programming. Works well for up to 200 cities.
- Implementations of branch-and-bound and problem-specific cut generation (branch-and-cut);

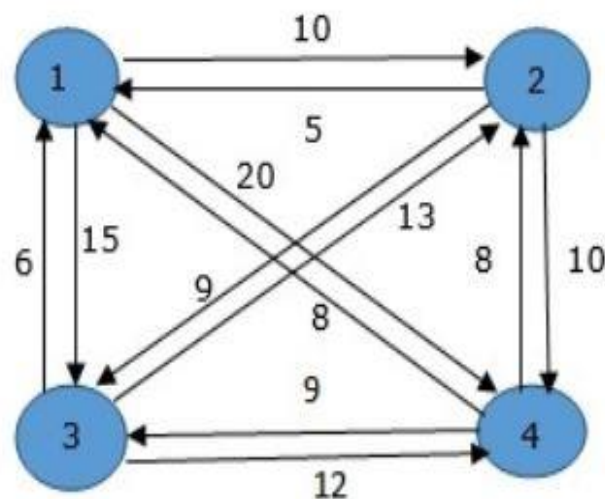
The prime mathematical deduction for our project plan will consider a subset of cities  $S \subseteq \{1, 2, 3, \dots, n\}$  that includes 1, and  $j \in S$ . Now let  $C(S, j)$  be the length of the shortest path visiting each city in  $S$  exactly once, starting at 1 and ending at  $j$ . When  $|S| > 1$ , we define  $C(S, 1) = \infty$  since the path cannot start and end at 1.

Now, let us express  $C(S, j)$  in terms of smaller sub-problems. We need to start at 1 and end at  $j$ . We should select the next city in such a way that:

$$C(S, j) = \min_{i \in S, i \neq j} C(S - \{j\}, i) + d(i, j) \quad \text{where } i \in S \text{ and } i \neq j$$

$$C(S, j) = \min_{i \in S, i \neq j} C(S - \{j\}, i) + d(i, j) \quad \text{where } i \in S \text{ and } i \neq j$$

We will use a simple example in the beginning to illustrate the steps to solve the travelling salesman problem.



From the above graph, the following table is prepared:

	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

For  $S = \Phi$ :

$\text{Cost}(3, \Phi, 1) = d(3, 1) = 6$

Similarly for 2 and 4 it will be 5 and 8 respectively.

For S = 1:

$$Cost(2, \{3\}, 1) = d[2, 3] + Cost(3, \Phi, 1) = 9 + 6 = 15 \quad cost(2, \{3\}, 1) = d[2, 3] + cost(3, \Phi, 1) = 9 + 6 = 15$$

$$Cost(2, \{4\}, 1) = d[2, 4] + Cost(4, \Phi, 1) = 10 + 8 = 18 \quad cost(2, \{4\}, 1) = d[2, 4] + cost(4, \Phi, 1) = 10 + 8 = 18$$

Similarly, there are others like  $(3, \{2\}, 1) = 18$ ;  $(3, \{4\}, 1) = 20$ ;  $(4, \{3\}, 1) = 15$ ;  $(4, \{2\}, 1) = 13$

S = 2

$$Cost(2, \{3, 4\}, 1) = \begin{cases} d[2, 3] + Cost(3, \{4\}, 1) = 9 + 20 = 29 \\ d[2, 4] + Cost(4, \{3\}, 1) = 10 + 15 = 25 \\ \{d[2, 3] + cost(3, \{4\}, 1) = 9 + 20 = 29 \\ d[2, 4] + cost(4, \{3\}, 1) = 10 + 15 = 25 \end{cases} = 25$$

$$Cost(3, \{2, 4\}, 1) = \begin{cases} d[3, 2] + Cost(2, \{4\}, 1) = 13 + 18 = 31 \\ d[3, 4] + Cost(4, \{2\}, 1) = 12 + 13 = 25 \\ \{d[3, 2] + cost(2, \{4\}, 1) = 13 + 18 = 31 \\ d[3, 4] + cost(4, \{2\}, 1) = 12 + 13 = 25 \end{cases} = 25$$

$$Cost(4, \{2, 3\}, 1) = \begin{cases} d[4, 2] + Cost(2, \{3\}, 1) = 8 + 15 = 23 \\ d[4, 3] + Cost(3, \{2\}, 1) = 9 + 18 = 27 \\ \{d[4, 2] + cost(2, \{3\}, 1) = 8 + 15 = 23 \\ d[4, 3] + cost(3, \{2\}, 1) = 9 + 18 = 27 \end{cases} = 23$$

S = 3

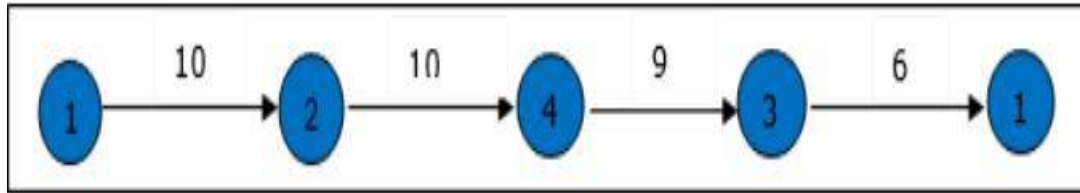
$$Cost(1, \{2, 3, 4\}, 1) = \begin{cases} d[1, 2] + Cost(2, \{3, 4\}, 1) = 10 + 25 = 35 \\ d[1, 3] + Cost(3, \{2, 4\}, 1) = 15 + 25 = 40 \\ d[1, 4] + Cost(4, \{2, 3\}, 1) = 20 + 23 = 43 \\ \{d[1, 2] + cost(2, \{3, 4\}, 1) = 10 + 25 = 35 \\ d[1, 3] + cost(3, \{2, 4\}, 1) = 15 + 25 = 40 \\ d[1, 4] + cost(4, \{2, 3\}, 1) = 20 + 23 = 43 \end{cases} = 35$$

Now, after implementing the above mathematical statements, we can deduce that the minimum cost path is 35.

Start from cost  $\{1, \{2, 3, 4\}, 1\}$ , we get the minimum value for  $d[1, 2]$ . When  $s = 3$ , select the path from 1 to 2 (cost is 10) then go backwards. When  $s = 2$ , we get the minimum value for  $d[4, 2]$ . Select the path from 2 to 4 (cost is 10) then go backwards.

When  $s = 1$ , we get the minimum value for  $d[4, 3]$ . Selecting path 4 to 3 (cost is 9), then we shall go to then go to  $s = \Phi$  step. We get the minimum value for  $d[3, 1]$  (cost is 6).





## Alternate Methods and their Drawbacks

### Factorial Method:

The most obvious way to solve the travelling salesman problem would be to write down all of the possible sequences in which the cities could be visited, compute the distance of each path, and then choose the smallest. But the number of possible itineraries for visiting  $n$  cities grows as the factorial of  $n$ , which is written, appropriately, as  $n!$

And, these numbers grow very rapidly, so as we increase the number of cities, the number of paths we need to compare blows up in a combinatorial explosion which makes finding the optimal path for 'Surakshit' RO company, by brute force computation a hopeless solution. This is the main drawback of basic Permutation-Combination

The number of possible paths along which we can visit the thirty cities is equal to the number of permutations of a set of thirty distinct members, which is equal to the factorial of the number of members, or  $30!$ . This becomes a very large number.

$$30! = 265,252,859,812,191,058,636,308,480,000,000 \approx 2.6525 \times 10^{32}$$

### Branch and Bound Method:

- This method solves discrete/combinatorial/mathematical optimization
- Top-down recursive search [Consists of 'Branching' and 'Bounding']

**Branching:** It recursively splits the search space into smaller spaces and minimizes  $f(x)$  on these smaller spaces

**Bounding:** It keeps track of bounds on the minimum and uses them to "prune" the search space

Therefore, 'Branch and Bound' is an algorithmic technique which finds the optimal solution by keeping the best solution found so far. If partial solution can't improve on the best it is abandoned, by this method the number of nodes which are explored can also be reduced. It also deals with the optimization problems over a search that can be presented as the leaves of the search tree.

Concept:

Step 1: Traverse the root node.

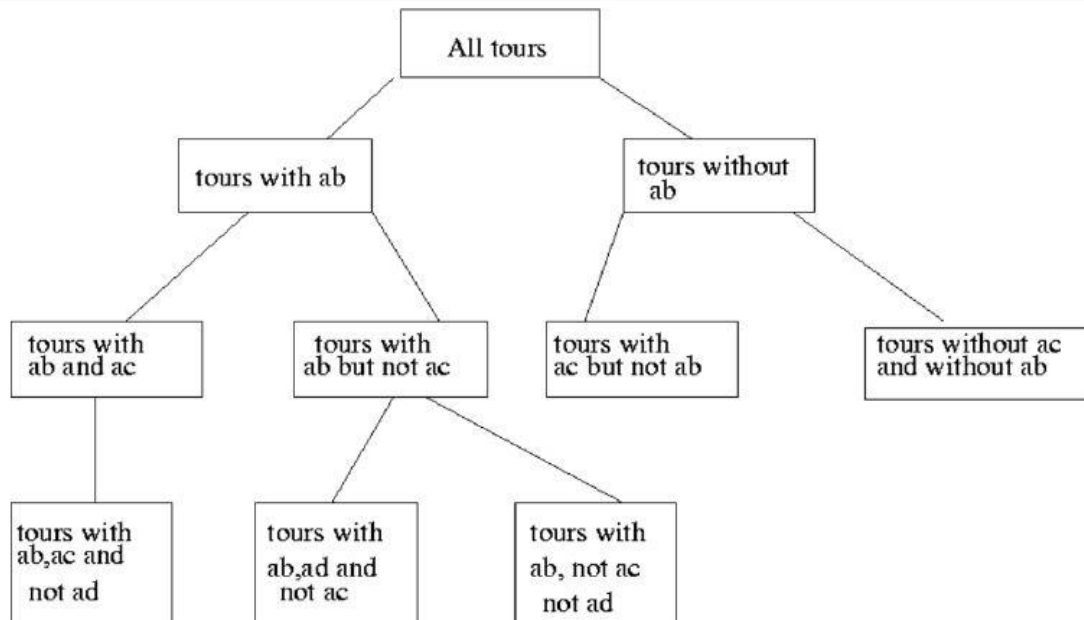
Step 2: Traverse any neighbour of the root node that is maintaining least distance from the root node.

Step 3: Traverse any neighbour of the neighbour of the root node that is maintaining least distance from the root node.

Step 4: This process will continue until we are getting the goal node.

## Applying to TSP

- Cost of a tour =  $\frac{1}{2} \sum_{v \in N} (d_{uv} + d_{wv})$  where  $uv, wv \in T$
- Lower bound: cost of a tour  $\geq \frac{1}{2} \sum_{v \in N} (d_{uv} + d_{wv})$  where  $uv, wv$  are the two least cost edges adjacent to  $v$



Algorithm:

Step 1: PUSH the root node into the stack.

Step 2: If stack is empty, then stop and return failure.

Step 3: If the top node of the stack is a goal node, then stop and return success.

Step 4: Else POP the node from the stack. Process it and find all its successors. Find out the path containing all its successors as well as predecessors and then PUSH the successors which are belonging to the minimum or shortest path.

Step 5: Go to step 5.

Step 6: Exit.

Drawbacks of this method:

1. The Branch and Bound algorithm is limited to small size network. In the problem of large networks, where the solution search space grows exponentially with the scale of the network, the approach becomes relatively prohibitive.
2. The load balancing aspects for Branch and Bound algorithm make it parallelization difficult.
3. Extremely time-consuming: Sometimes, the number of nodes in a branching tree can be too large. The algorithm finds the first complete schedule and then tries to improve it. Often developing the “promising” branches may lead to a huge number of offspring that finally may not give an improvement.

4. Thus the size of the tree may grow exponentially without improving the best solution obtained.

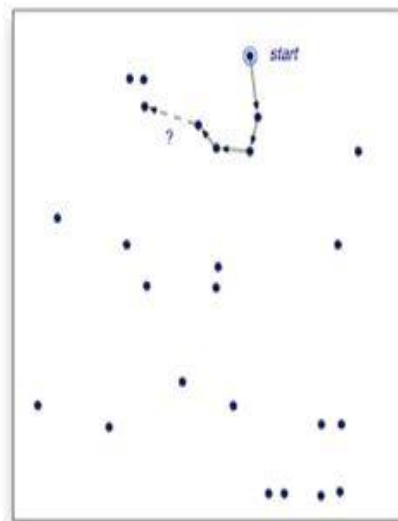
## Exhaustive Search:

One way to find the optimal tour is to consider all possible paths

The method (suppose we name it `each_tour`) is just like the `each` method that iterates over a list:

- ❖ it iterates over all possible tours
- ❖ even though we don't have an actual list of tours, this iteration is a type of search
- ❖ computer scientists call it an *exhaustive search* since all combinations are tried

#cities	#tours
5	12
6	60
7	360
8	2,520
9	20,160
10	181,440



*The number of tours for 25 cities:*

**310,224,200,866,619,719,680,000**

There is a problem (drawbacks of the existing method) with the exhaustive search strategy:-

- The number of possible tours of a map with  $n$  cities is  $(n - 1)! / 2$
- Also known as 'Brute-Force' this is a very general problem-solving technique and algorithmic paradigm that consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement.
- Hence, there is a chance of combinatorial explosion



# Proposed Method

Our project proposes an enhancement on the 'Dynamic Programming' method (often referred to as DP) and subsequently the Held-Karp Algorithm.

DP is particularly useful to understand in the context of algorithm design for this problem scenario, and even for future use. We've dealt with lots of alternative algorithms, from sorting algorithms, to traversal algorithms, to path-finding algorithms, discussed their drawbacks. However, we haven't really talked about the ways that these algorithms are *constructed* or to be precise, how the Held Karp Algorithm is constructed.

Dynamic Programming is synonymous to 'Dynamic Optimization', an approach to solving complex problems by breaking them down into smaller parts, and storing the results to these subproblems so that they only need to be computed once.

The Held-Karp algorithm, created by Michael Held and Richard Karp in 1962, has the lowest asymptotic complexity of any algorithm that solves a TSP to date. The critical idea behind the dynamic program is the calculation of suboptimal shortest paths that must:

- a) traverse an explicit set of nodes
- b) end on a specific node.

## Held-Karp Optimization Property

Every subpath of a path of minimum distance is itself of minimum distance.

## Local Optimum

Data structure: Dictionary with key = (set, integer), mapping to integer.

Local optimum (start, subset, end, G):

- Calculate path from start to end via one-vertex set
- Calculate path from start to end via k-vertex set, using the best k-1 vertex path

## TSP

For  $v$  in  $G \setminus \text{root}$ :

cost = Local optimum(root,  $G \setminus v$ , v, G)

If cost < min\_cost: min\_cost  
return min\_cost

## Complexity

- $n$  options for last vertex;  $2^n$  sets in between; linear scan for each
- $O(n^2 2^n)$

The time complexity of the Held-Karp algorithm is  $O(n^2 2^n)$ . From the  $n - 1$  nodes that must be traversed (we exclude the starting first node), choose  $k$  nodes to be the set  $S$ . There are  $k$  possible nodes that can be selected as the end node  $l$ , and  $(k - 1)$  comparisons of  $C$  that must be made for each one.

# Advantages of Proposed Method

The dynamic programming approach for Held-Karp algorithm offers **faster execution** than exhaustive enumeration.

Cutting plane algorithm from linear programming does the work by adding inequality constraint to gradually converge at an optimal solution but when people apply this method to find a cutting plane, **they often depend on experience. Therefore, Held-Karp is preferred.**

**Branch-bound** algorithm is a search algorithm widely used, but it's **not good for solving the large-scale problem.**

Therefore, because of the disadvantages of other methods, our proposed method subsequently has its own advantages.

## Proposed System Architecture

Any common algorithmic design paradigm might involve:

Divide and Conquer design – An example of this is Merge Sort algorithm

Greedy Algorithm design – An example of this is Dijkstra's algorithm

For Held-Karp, we have assumed that the input is a complete graph given as a distance matrix and that its vertices are numbered 0 through  $(n-1)$ . We will start and end the TSP route in vertex  $(n-1)$ . Each state of the dynamic programming will be of the form (the set of other vertices already visited, the last vertex visited). For each state we are asking the question what is the cheapest way to reach this state. Then, the cheapest TSP route = the cheapest way to visit all vertices + return back from the last of them to vertex  $(n-1)$ .

The architecture we are proposing is simple; **A more efficient yet simple approach to Held-Karp algorithm for the technical people at the company to understand and provide an optimal path for salesmen of 'Surakshit' RO to provide warranty as well as repair services.**

The gist of dynamic programming is two parts:

- 1) A base case that we already know the answer of (*stopping condition*)
- 2) Decreasing the problem domain and calling our algorithm again. (*Recursion*)

We will put this very simply, and without going into much mathematical lingo. The algorithm takes two inputs, the vertex we are starting from `vertex`, and a list of vertices to visit vertices.

### Client (Salesman):

The client, in our scenario, the salesman of 'Surakshit RO', need to be updated continuously with the optimum route for carrying out necessary company services within stipulated time. If he/she carries a laptop, there can be a system software, installed on the laptop as a application, provided by the company which contains this algorithm. Or there can be a mobile app, where the information is provided, and with help of directions and map, the salesman knows the optimal route to travel.

### Server(Controller/Technical Support):

Here the person who gets the requests and complaints from customers, manages the data, and arranges it in a format. Let's say for example, there are 6 servicings to be done in Odisha and 4 in West Bengal. Now, there are 2 salesmen who are given the job. The algorithm at server end will help find the optimum route for each of the salesmen, and allot the cities accordingly

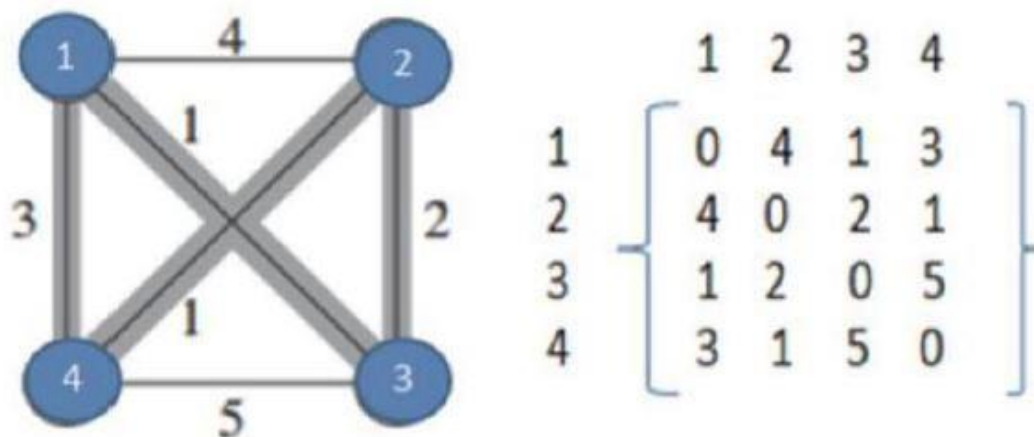
### Payment Gateway/Transaction:

There will be a separate server and system that process a particular order, after confirming with both the salesman and the customer that the issue is resolved and the payment has been done. Basically, it is responsible for a particular complaint or request's payment, record keeping and other business aspects of the transaction. But most importantly, it communicates with Server, to update the optimal route and tick off the cities that are already covered by Salesman.

## Implementation Details

For implementation of our project, we first look at an **example problem**:

### Example Problem



Above we can see a complete directed graph and cost matrix which includes distance between each city. There are 4 cities named 1, 2, 3 and 4. We can observe that cost matrix is symmetric that means distance between city 2 to 3 is same as distance between city 3 to 2.

**Here problem is travelling salesman wants to find out his tour with minimum cost.**

Say it is  $T(1, \{2, 3, 4\})$ , means, initially he is at city 1 and then he can go to any of  $\{2, 3, 4\}$ . From there to reach non-visited vertices (cities) becomes a new problem. Here we can observe that main problem spitted into sub-problem, this is property of Held-Karp algorithm.

Note: While calculating below right side values calculated in bottom-up manner. Red color values taken from below calculations.

$T(1, \{2, 3, 4\}) = \text{minimum of}$

$$= \{(1, 2) + T(2, \{3, 4\})\} 4+6=10$$

$$= \{(1, 3) + T(3, \{2, 4\})\} 1+3=4$$

$$= \{(1, 4) + T(4, \{2, 3\})\} 3+3=6$$

Here minimum of above 3 paths is answer but we know only values of (1,2) , (1,3) , (1,4) remaining thing which is  $T(2, \{3,4\})$  ...are new problems now. First we have to solve those and substitute here.

Upon doing so, we see

$$T(2, \{4\}) = (2,4) + T(4, \{\}) 1+0=1$$

$$T(4, \{2\}) = (4,2) + T(2, \{\}) 1+0 = 1$$

$$T(2, \{3\}) = (2,3) + T(3, \{\}) 2+0 = 2$$

$$T(3, \{2\}) = (3,2) + T(2, \{\}) 2+0=2$$

Here  $T(4, \{\})$  is reaching base condition in recursion, which returns 0 (zero) distance. This is where we can find final answer.

Minimum distance in this case is 11 which includes path 1->3->4->2->1.

After solving example problem we can easily write recursive equation.

### Recursive Equation

$T(i, s) = \min((i, j) + T(j, S - \{j\}))$  ;  $S \neq \emptyset$  ;  $j \in S$  ;  $S$  is set that contains non visited vertices =  $(i, 1)$  ;  $S = \emptyset$ , This is base condition for this recursive equation. Here,

$T(i, S)$  means We are travelling from a vertex "i" and have to visit set of non-visited vertices "S" and have to go back to vertex 1 (let we started from vertex 1).

$(i, j)$  means cost of path from node i to node j.

If we observe the first recursive equation from a node we are finding cost to all other nodes (i,j) and from that node to remaining using recursion ( $T(j, \{S-j\})$ ).

If S is empty that means we visited all nodes, we take distance from that last visited node to node 1 (first node). Because after visiting all he has to go back to initial node.

### Time Complexity

As said earlier, Time complexity  $O(n)$  is  $= O(n^2 2^n)$

Since we are solving this using Held-Karp, we know that Held-Karp approach contains sub-problems. Here after reaching  $i^{th}$  node finding remaining minimum distance to that  $i^{th}$  node is a sub-problem. If we solve recursive equation we will get total  $(n-1) 2^{(n-2)}$  sub-problems, which is  $O(n 2^n)$ .

Each sub-problem will take  $O(n)$  time (finding path to remaining  $(n-1)$  nodes).

Therefore total time complexity is  $O(n 2^n) * O(n) = O(n^2 2^n)$

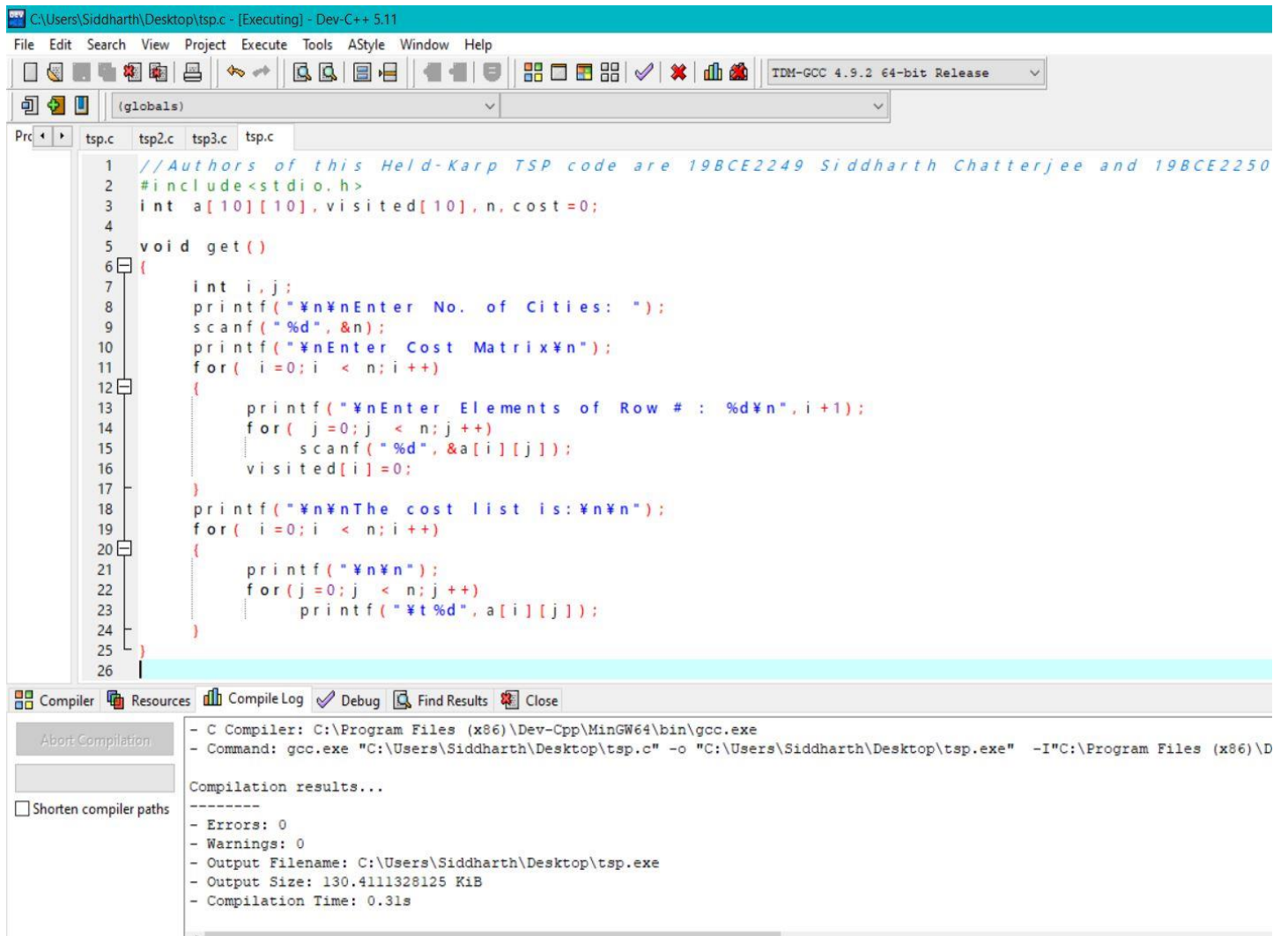
Space complexity is also number of sub-problems which is  $O(n 2^n)$ .

# Testing of Algorithm and Code Output

## PseudoCode/Algorithm

```
1. // Authors are 19BCE2249 Siddharth Chatterjee and 19BCE2250 Ishan Sagar Jogalekar
2. // This is a sample implementation of how the Held-Karp Algorithm actually works
3. // First, initialize the DP table
4. // best[visited][last] = the cheapest way to reach the state where:
5. // - "visited" encodes the set of visited vertices other than N-1
6. // - "last" is the number of the vertex visited last
7.
8. vector< vector<int> > best( 1<<(N-1), vector<int>( N, INT_MAX ) );
9.
10. // use DP to solve all states
11. // note that whenever we solve a particular state,
12. // the smaller states we need have already been solved
13.
14. for (int visited = 1; visited < (1<<(N-1)); ++visited) {
15.     for (int last = 0; last < (N-1); ++last) {
16.
17.         // last visited vertex must be one of the visited vertices
18.         if (!(visited & 1<<last)) continue;
19.
20.         // try all possibilities for the previous vertex,
21.         // pick the best among them
22.         if (visited == 1 << last) {
23.             // previous vertex must have been N-1
24.             best[visited][last] = distance[N-1][last];
25.         } else {
26.             // previous vertex was one of the other ones in "visited"
27.             int prev_visited = visited ^ 1<<last;
28.             for (int prev = 0; prev < N-1; ++prev) {
29.                 if (!(prev_visited & 1<<prev)) continue;
30.                 best[visited][last] = min(
31.                     best[visited][last],
32.                     distance[last][prev] + best[prev_visited][prev]
33.                 );
34.             }
35.         }
36.     }
37. }
38.
39. // use the precomputed path lengths to choose the cheapest cycle
40. int answer = INT_MAX;
41. for (int last=0; last<N-1; ++last) {
42.     answer = min(
43.         answer,
44.         distance[last][N-1] + best[ (1<<(N-1))-1 ][last]
45.     );
46. }
```

## Code Input Glimpse



```
1 //Authors of this Held-Karp TSP code are 19BCE2249 Siddharth Chatterjee and 19BCE2250
2 #include<stdio.h>
3 int a[10][10], visited[10], n, cost=0;
4
5 void get()
6 {
7     int i, j;
8     printf("\n\nEnter No. of Cities: ");
9     scanf("%d", &n);
10    printf("\nEnter Cost Matrix\n");
11    for( i=0; i < n; i++)
12    {
13        printf("\nEnter Elements of Row # : %d\n", i+1);
14        for( j=0; j < n; j++)
15            scanf("%d", &a[i][j]);
16        visited[i]=0;
17    }
18    printf("\n\nThe cost list is:\n\n");
19    for( i=0; i < n; i++)
20    {
21        printf("\n\n");
22        for( j=0; j < n; j++)
23            printf("\t%d", a[i][j]);
24    }
25 }
26
```

Compiler: C:\Program Files (x86)\Dev-Cpp\MinGW64\bin\gcc.exe  
Command: gcc.exe "C:\Users\Siddharth\Desktop\tsp.c" -o "C:\Users\Siddharth\Desktop\tsp.exe" -I"C:\Program Files (x86)\D

Compilation results...

-----  
- Errors: 0  
- Warnings: 0  
- Output Filename: C:\Users\Siddharth\Desktop\tsp.exe  
- Output Size: 130.4111328125 KiB  
- Compilation Time: 0.31s

## The Code

```
//Authors of this Held-Karp TSP code are 19BCE2249 Siddharth Chatterjee and 19BCE2250 Ishan Jogalekar//
```

```
#include<stdio.h>
```

```
int a[10][10], visited[10], n, cost=0;
```

```
void get()
```

```
{
    int i, j;
    printf("\n\nEnter No. of Cities: ");
    scanf("%d", &n);
    printf("\nEnter Cost Matrix\n");
    for( i=0; i < n; i++)
```



```

{
    printf("\nEnter Elements of Row # : %d\n",i+1);
    for( j=0;j < n;j++)
        scanf("%d",&a[i][j]);
    visited[i]=0;
}
printf("\n\nThe cost list is:\n\n");
for( i=0;i < n;i++)
{
    printf("\n\n");
    for(j=0;j < n;j++)
        printf("\t%d",a[i][j]);
}
}

```

```

void mincost(int city)

```

```

{
    int i,ncity;
    visited[city]=1;
    printf("%d -->",city+1);
    ncity=least(city);
    if(ncity==999)
    {
        ncity=0;
        printf("%d",ncity+1);
        cost+=a[city][ncity];
        return;
    }
    mincost(ncity);
}

```

```

int least(int c)

```

```

{

```

```

int i,nc=999;

int min=999,kmin;

for(i=0;i < n;i++)
{
    if((a[c][i]!=0)&&(visited[i]==0))
        if(a[c][i] < min)
        {
            min=a[i][0]+a[c][i];
            kmin=a[c][i];
            nc=i;
        }
}

if(min!=999)
    cost+=kmin;

return nc;
}

void put()
{
    printf("\n\nMinimum cost:");
    printf("%d",cost);
}

void main()
{
    printf("\n Welcome to Siddharth and Ishan's TSP solver!");
    printf("\n Teammate 1: 19BCE2249 Siddharth Chatterjee");
    printf("\n Teammate 2: 19BCE2250 Ishan Jogalekar");
    printf("\n This is part of our Project: 'Travelling Salesman Problem' for DSA");
    get();
    printf("\n The table above is in Cost List format");
    printf("\n And the Surakshit RO salesman achieves the optimum route: ");
    mincost(0);
}

```

```
put();  
}
```

## Code Output:

```
C:\Users\Siddharth\Desktop\tsp.exe  
  
Welcome to Siddharth and Ishan's TSP solver!  
Teammate 1: 19BCE2249 Siddharth Chatterjee  
Teammate 2: 19BCE2250 Ishan Jogalekar  
This is part of our Project: 'Travelling Salesman Problem' for DSA  
  
Enter No. of Cities: 4  
  
Enter Cost Matrix  
  
Enter Elements of Row # : 1  
0 10 15 20  
  
Enter Elements of Row # : 2  
5 0 9 10  
  
Enter Elements of Row # : 3  
6 13 0 12  
  
Enter Elements of Row # : 4  
8 8 9 0  
  
The cost list is:  
  
    0      10      15      20  
    5       0       9      10  
    6      13       0      12  
    8       8       9       0  
  
The table above is in Cost List format  
And the Surakshit RO salesman achieves the optimum route: 1 -->2 -->4 -->3 -->1  
  
Minimum cost:35  
-----  
Process exited after 52.13 seconds with return value 2  
Press any key to continue . . .
```

## Conclusion and future enhancement

Our project's aim was to **find the optimum path** and consequently provide a better solution to the famous TSP problem. We have made **our workable solution better than we what we started from, so that the salesman of 'Surakshit RO' carrying out servicing requests and checking the complaints across different cities, can travel with minimum cost.**

**Note:** The minimum cost is a function of various factors like speed, distance, etc.

Also, the code for our project is **far less repetitive**, than Branch and Bound method or Exhaustive enumeration and combinatorial permutation.

The implementation shown **exemplifies how the bottom up DP approach would scale for a travelling salesman problem where the salesman has to visit four cities (in our example).**

We have seen that we're making a lot of calls (functions and recursive ones), but our function call "tree" is a bit slimmer and significantly better than before.

By using dynamic programming, we've made our solution for the Travelling Salesman Problem a bit better by choosing to **smartly enumerate function calls rather than brute-force our way through** every single possible path that our salesman could take.

Enumerating function calls in a smart way is better than enumerating through possible paths which also helps us achieve a better time complexity than  $O(n!)$ .

For **future enhancement** there is scope of **elaborating upon Heuristic algorithms** like **Genetic Algorithm, Ant Colony Optimization** and **Stimulated Annealing** or approximation algorithm like **Christofides**.

## References

1. <https://www.math.cmu.edu/>
2. <http://www.math.nagoya-u.ac.jp/>
3. An insight into TSP from University of Tennessee, Knoxville
4. C.H. Papadimitriou and K. Steglitz. "Combinatorial Optimization: Algorithms and Complexity". Prentice Hall of India Private Limited. India. 1997.
5. [https://www.researchgate.net/publication/Accelerating\\_the\\_Held-Karp\\_Algorithm\\_for\\_the\\_Symmetric\\_Traveling\\_Salesman\\_Problem](https://www.researchgate.net/publication/Accelerating_the_Held-Karp_Algorithm_for_the_Symmetric_Traveling_Salesman_Problem)
6. [Travelling Salesman Problem](#), 0612 TV w/ NERDfirst
7. [Traveling Salesman Problem Dynamic Programming Held-Karp](#), Tushar Roy
8. [What is an NP-complete in computer science?](#), StackOverflow
9. [Big O Notation and Complexity](#), Kestrel Blackmore
10. [A Dynamic Programming Algorithm for TSP](#), Coursera
11. [Traveling Salesman Problem: An Overview of Applications, Formulations, and Solution Approaches](#), Rajesh Matai, Surya Singh, and Murari Lal Mittal