INDEX

Review 1

PROJECT ABSTRACT

Problem Statement :

LivePure Company has supplied and installed a large number of RO devices all over a geographical area . 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 already installed . Due to stiff competition for supply and installation of RO Devices, the optimal route has to take into account to minimize the cost of the services and to prepare a well-planned schedule to carry out services efficiently .

Abstract:

The aim of this project is to discuss various aspects and techniques proposed to solve Travelling salesman problem.

Introduction:

The idea of the traveling salesman problem (TSP) is to find a tour of a given number of cities, visiting each city exactly once and returning to the starting city where the length of this tour is minimized. The first instance of the traveling salesman problem was from Euler in 1759 whose problem was to move a knight to every position on a chess board exactly once. The traveling salesman first gained fame in a book written by German salesman BF Voigt in 1832 on how to be a successful traveling salesman.

DEFINITION: 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: (Cij) be a distance (or cost) matrix associated with A. The 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). In several applications, C can also be interpreted as a cost or travel time matrix. 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.

REAL TIME PROBLEM (CASE STUDY)

❖ FLIGHT TOUR:

✓ ***APPROACH: HELD KARP, BRANCH and BOUND ALGORITHM***

▪ Consider a flight scheduled to N cities in a country. The flight needs to cover all the city only once and come back to its hub covering all the cities. The main parameter in this problem is that the cost of travelling from one city to another need not to be same as that of from the second to first. So, a better schedule (path) needs to be made to have minimum
cost of travelling for airlines.

▪ Sometimes, other parameters like time consumption also can be made as a part of study.

❖ SCHOOL BUS TRANSPORTATION:

✓ ***APPROACH: HELD KARP, BRANCH AND BOUND ALGORITHM***

▪Consider a school bus that has a daily duty to pick up every student from their home and drop them back at the departure. Here, the cost of transportation from the school to respective house of student is same at the time of arrival and departure. Thus, the bus needs to go along a proper path through which it goes to home of every student only once and covers house of all the students.

LITERATURE SURVEY.

▪ The traveling salesman first gained fame in a book written by German salesman BF Voigt in 1832 on how to be a successful traveling salesman.

▪ Currently the only known method guaranteed to optimally solve the traveling

salesman problem of any size, is by enumerating each possible tour and searching for the tour with smallest cost. Each possible tour is a permutation of 123 . . . n, where n is the number of cities, so therefore the number of tours is n!. When n gets large, it becomes impossible to
find the cost of every tour in polynomial time.

- There does not exists a unique way of approach for all the value of N. As the value of N changes, different ways of approaches and algorithms can be applies to solve it optimally.

The method of finding cost of all permutations if of factorial time but to solve it dynamically reduces it to polynomial time problem. Thus, different approaches can be made to solve optimally.

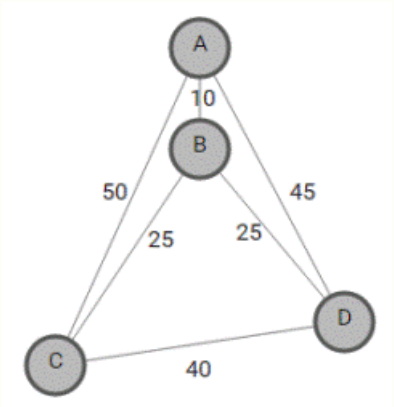
- The approach to the problem was amend by different iterative approaches and every attempt was made to reduce the polynomial time complexity to solve the problem optimally. Dynamical approach was made further to the problem which was a better way to approach the problem.
- Genetic algorithm (GA) as a computational intelligence method is a search technique used in computer science to find approximate solutions to combinatorial optimization problems

Review 2

Branch and Bound:

Branch and bound is an algorithm design paradigm which is generally used for solving combinatorial optimization problems. These problems are typically exponential in terms of time complexity and may require exploring all possible permutations in worst case. The Branch and Bound Algorithm technique solves these problems relatively quickly. During the search bounds for the objective function on the partial solution are determined. At each level the best bound is explored first, the technique is called best bound first. If a complete solution is found

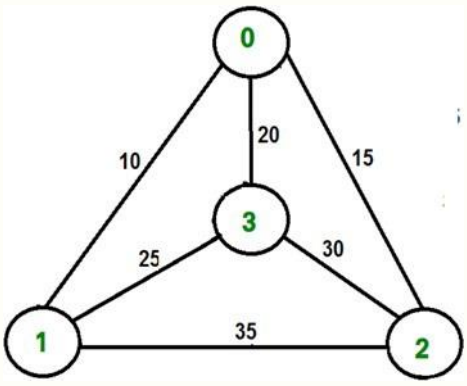
then that value of the objective function can be used to prune partial solutions that exceed the bounds.



AN APPROACH TO SOLVE SYMMETRIC TSP OPTIMALLY:

In Naïve solution, we find all (n-1)! possible permutations and compute the optimal solution for the problem. This approach solves the problem in factorial time. For symmetric TSP, an approach can be made to solve it in polynomial time.

Example:



ALL POSSIBLE PERMUTATIONS:

0—1—2—3—0 ….(a) (95) (a)==(f) (in reverse path order)

0—1—3—2—0 …..(b) (80) (b)==(d) (in reverse path order)

0—2—1—3—0 ….(c) (95) (c)==(e) (in reverse path order)

0—2—3—1—0 …..(d) (80)

0—3—1—2—0 …..(e) (95)

0—3—2—1—0 …..(f) (95)

Thus, we need to find only first (n-1)!/2 permutations to find the optimal solution for symmetric TSP. Consider any one node except starting (ending) node as our pivot node and the find all possible path from starting to pivot city via different number of cities.

0—1 (10) (1) node 0 as pivot node

0—2—1 (50) (2)

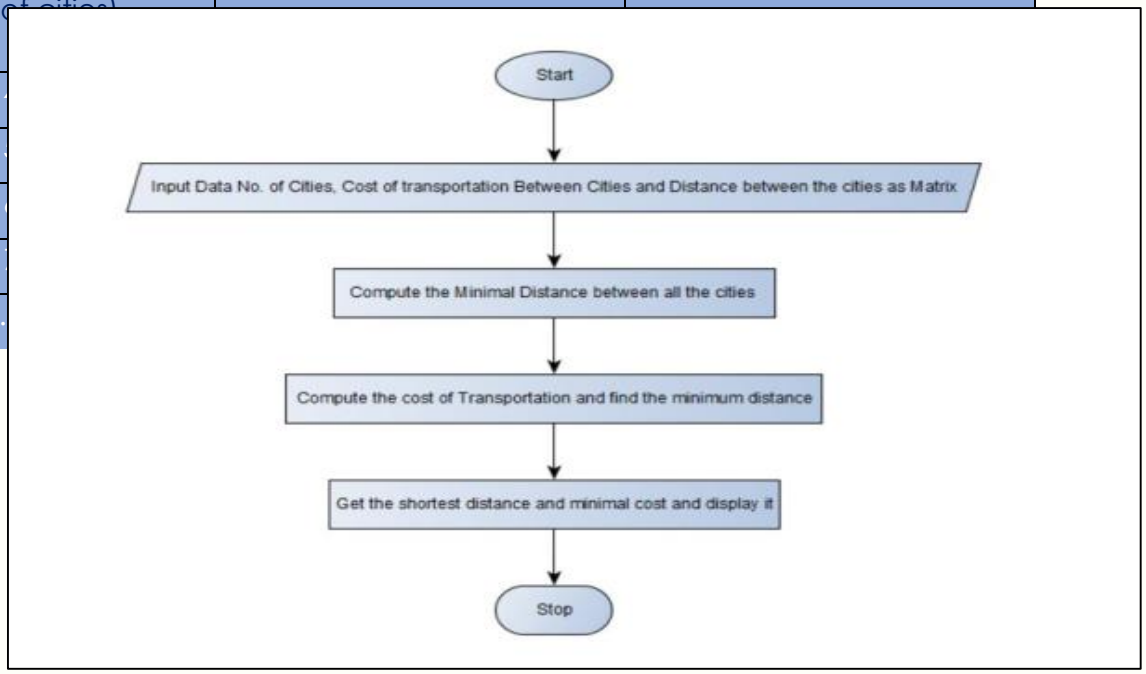
0—3—1 (35) (3)

0—2—3—1 (70) (4)

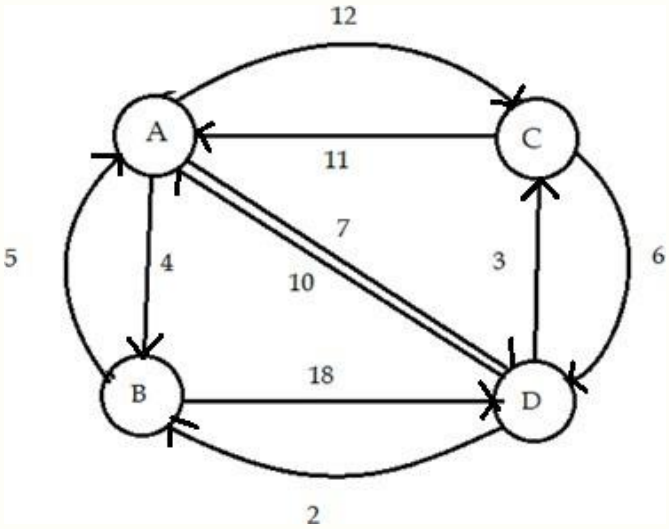
0—3—2—1 (85) (5)

Now, by joining path (1) to (4) and (5), we get path (a), (b), (d), and (f). Similarly, by joining path (2) and (3), we get path (c) and (e). Thus, we get 80 as our optimal solution in less number of traverse.

N (No. of cities)	No. of permutations in	Permutation in this



BRANCH AND BOUND ALGORITHM TO SOLVE TSP:



SOLUTION:

STEP-1: Write the initial cost matrix and reduce it.

INITIALIZE THE INITIAL COST MATRIX:

	A	B	C	D
A	∞	4	12	7
B	5	∞	∞	18
C	11	∞	∞	6
D	10	2	3	∞

	A	B	C	D	Reduce by
A	∞	4	12	7	4
B	5	∞	∞	18	5
C	11	∞	∞	6	6
D	10	2	3	∞	2

AFTER ROW REDUCTION

	A	B	C	D
A	∞	0	8	3
B	0	∞	∞	13
C	5	∞	∞	0
D	8	0	1	∞
Reduce by	0	0	1	0

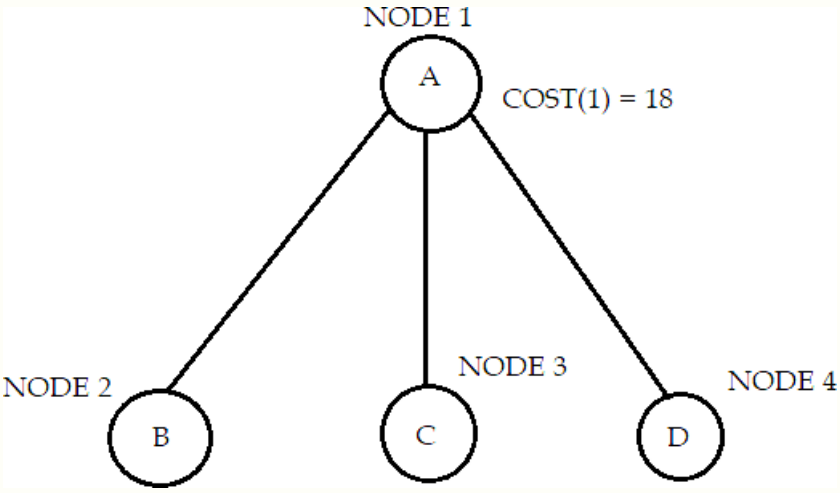
	A	B	C	D
A	∞	0	7	3
B	0	∞	∞	13
C	5	∞	∞	0
D	8	0	0	∞

AFTER ROW REDUCTION

Therefore, COST OF NODE 1 is : COST(1) = REDUCTION

= 4+5+6+2+1

= 18



• Now, there are 3 possible ways for salesman to go from Node 1 (A) i.e. Node 2, 3, 4

• By finding cost of Node 2, 3, 4 in step-2, the salesman will choose the path whose cost is minimum (optimal).

STEP 2: Finding the cost of all the remaining nodes to choose next path.

(i) CHOOSING TO GO TO VERTEX-B: NODE-2 (Path A->B)

• From the Reduced matrix of step-1, M [A, B] = 0.

• Set row A and column B to ∞.

• Set M [B, A] = ∞.

	A	B	C	D
A	∞	∞	∞	∞

	A	B	C	D
A	∞	∞	∞	∞

Resultant cost matrix

B	∞	∞	∞	13
C	5	∞	∞	0
D	8	∞	0	∞

AFTER ROW AND COLOUMN REDUCTION

Reduced matrix after row and column reduction

B	∞	∞	∞	0	(13)
C	0	∞	∞	0	
D	3	∞	0	∞	
	(5)				

Thus, cost of node-2 is: **COST (2) = COST (1) + REDUCTION + M [A, B]**

= 18+13+5+0

= 36

(ii) CHOOSING TO GO TO VERTEX-C: NODE-2 (Path A→C)

- From the Reduced matrix of step-1, M [A, C] = 7.
- Set row A and column C to ∞.
- Set M [C, A] = ∞.

Resultant cost Matrix

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	13
C	∞	∞	∞	0
D	8	0	∞	∞

AFTER ROW AND COLOUMN REDUCTION

Reduced matrix after row and column reduction

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	13
C	∞	∞	∞	0
D	8	0	∞	∞

Thus, cost of node-3 is:

COST (3) = COST (1) + REDUCTION + M [A, C]

= 18+0+7

=25

(iii) CHOOSING TO GO TO VERTEX-D: NODE-4 (Path A→D)

- From the Reduced matrix of step-1, M [A, D] = 3.

- Set row A and column D to ∞.
- Set M [D, A] = ∞

Resultant cost Matrix

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	∞
C	5	∞	∞	0
D	∞	0	0	∞

AFTER ROW AND COLOUM REDUCTION →

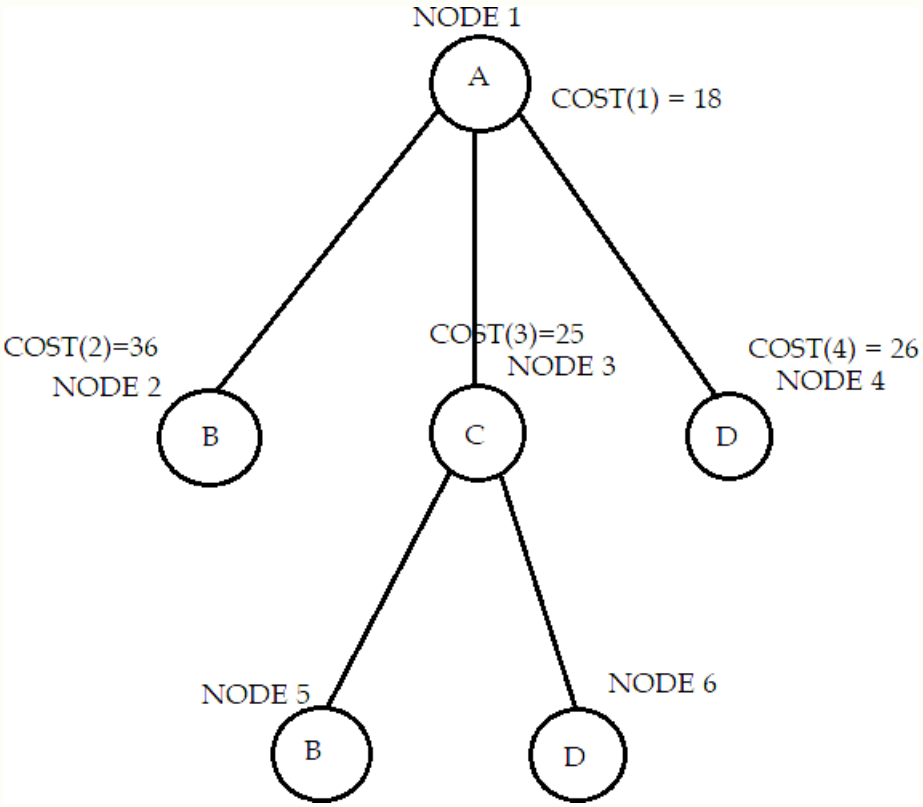
Reduced matrix after row and column reduction

	A	B	C	D	
A	∞	∞	∞	∞	
B	0	∞	∞	∞	
C	0	∞	∞	∞	(5)
D	∞	0	0	∞	

Thus, cost of node-4 is: **COST (4) = COST (1) + REDUCTION + M [A, D]**
= 18+5+3+0

- Thus, we have:
  COST (2) = 35
  COST (3) = 25 (minimal)
  COST (4) = 26

- So the salesman will head towards the path A→C and further path can also be obtained in
the same way.
- Now, again the salesman has two possible ways to move ahead i.e. on path B or D



• Now, there are 2 possible ways for salesman to go from Node 3 (C) i.e. Node 5, 6.
• By finding cost of Node 5,6in step-3, the salesman will choose the path whose cost is
minimum (optimal).

STEP-3: Finding the cost of all the remaining nodes to choose next path.

(i) CHOOSING TO GO TO VERTEX-B: NODE-5 (Path A→C→B)

• From the Reduced matrix of step-1, M [C, B] = ∞.
• Set row C and column B to ∞.
• Set M [B, A] = ∞.

Resultant cost Matrix

	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	13
C	∞	∞	∞	∞
D	8	∞	∞	∞

AFTER ROW AND COLOUMN

REDUCTION

Reduced matrix after row and column reduction

	A	B	C	D	
A	∞	∞	∞	∞	
B	∞	∞	∞	0	(13)
C	∞	∞	∞	∞	(5)
D	0	∞	0	∞	(8)

Thus, cost of node-5 is: COST (5) = COST (3) + REDUCTION + M [C, B]
= 25+13+8+∞
= ∞

(ii) CHOOSING TO GO TO VERTEX-D: NODE-6 (Path A→C→D)

- From the Reduced matrix of step-1, M [C, D] = 0.
- Set row C and column D to ∞.
- Set M [D, A] = ∞.

Resultant cost Matrix

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	∞
C	∞	∞	∞	∞

Thus, cost of node-6 is:
COST (6) = COST (3) + REDUCTION + M [C, D]

Reduced matrix after row and column reduction

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	∞
C	∞	∞	∞	∞

D ∞ ∞ ∞ ∞ AFTER ROW AND COLOUMN D ∞ ∞ ∞ ∞

REDUCTION

= 25+0+0
= 25

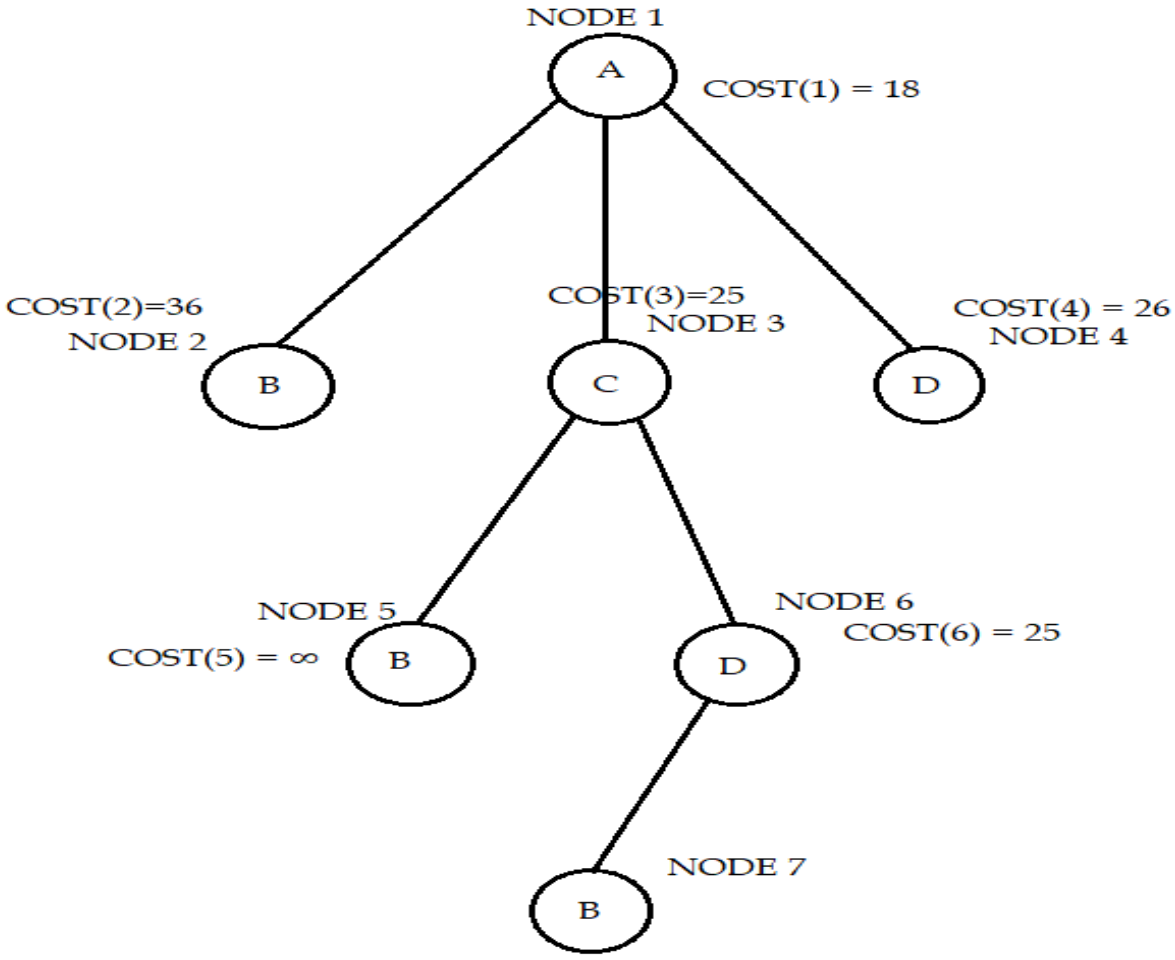
• Thus, we have:

COST (5) = ∞
COST (6) = 25 (minimal)

• So the salesman will head towards the path A→C→D and further path can also be obtained

in the same way

•Now, the salesman has the only possible path available to move ahead i.e. on path B



• Now, there is only one possible way available for salesman to go from Node 6 (D) i.e.
Node 7.

STEP-4: Finding the cost of the remaining nodes.

(i) CHOOSING TO GO TO VERTEX-B: NODE-7 (Path A->C->D->B)

- From the Reduced matrix of step-1, M [D, B] = 0.
- Set row D and column B to ∞.
- Set M [B, A] = ∞.

Resultant cost Matrix

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	∞
C	∞	∞	∞	∞
D	∞	∞	∞	∞

AFTER ROW AND COLOUMN REDUCTION

Reduced matrix after row and column reduction

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	∞
C	∞	∞	∞	∞
D	∞	∞	∞	∞

Thus, cost of node-7 is: **COST (7) = COST (5) + REDUCTION + M [C,D B]**
= 25+0+0
= 25

- **THUS, OPTIMAL PATH IS A->C->D->B->A WITH COST OF 25**

- Steps To Follow:

```
Lower Bound for vertex 1 =
       Old lower bound - ((minimum edge cost of 0 +
                          minimum edge cost of 1) / 2)
                        + (edge cost 0-1)
Lower bound(2) =
       Old lower bound - ((second minimum edge cost of 1 +
                          minimum edge cost of 2)/2)
                 + edge cost 1-2)
```

The time complexity for Branch_and_bound_1 is O(N^2) and that of Branch_and_bound_2 used in program is O(VE) and as dynamic algorithm doesn't have a set time complexity as it is an approach to solve the problem. Hence this shows that for lesser number of places to travel the Branch_and_bound_2 is best as the time complexity is lesser, however if the number of cities are greater than a value then Branch_and_bound_1 comes out to be the optimal out of the two algorithm.

////////////////////// CODE OF BRANCH AND BOUND APPROACH //////////////////////

LANGUAGE- C++

```
#include <iostream>
#include <vector>
#include <queue>
#include <utility>
#include <cstring>
#include <climits>
using namespace std;

// N is number of total nodes on the graph or the cities in the map
#define N 8

// Sentinal value for representing infinity
#define INF INT_MAX
#define inf 999

// State Space Tree nodes
struct Node
{
   // stores edges of state space tree
   // helps in tracing path when answer is found
   vector<pair<int, int> > path;

   // stores the reduced matrix
   int reducedMatrix[N][N];

   // stores the lower bound
```

```
    int cost;

    //stores current city number
    int vertex;

    // stores number of cities visited so far
    int level;
};

// Function to allocate a new node (i, j) corresponds to visiting
// city j from city i
Node* newNode(int parentMatrix[N][N], vector<pair<int, int> > const
&path,
          int level, int i, int j)
{
    Node* node = new Node;

    // stores ancestors edges of state space tree
    node->path = path;
    // skip for root node
    if (level != 0)
        // add current edge to path
        node->path.push_back(make_pair(i, j));

    // copy data from parent node to current node
    memcpy(node->reducedMatrix, parentMatrix,
        sizeof node->reducedMatrix);

    // Change all entries of row i and column j to infinity
    // skip for root node
    for (int k = 0; level != 0 && k < N; k++)
    {
        // set outgoing edges for city i to infinity
        node->reducedMatrix[i][k] = INF;

        // set incoming edges to city j to infinity
        node->reducedMatrix[k][j] = INF;
    }

    // Set (j, 0) to infinity
    // here start node is 0
    node->reducedMatrix[j][0] = INF;

    // set number of cities visited so far
    node->level = level;

    // assign current city number
    node->vertex = j;

    // return node
    return node;
```

```
}

// Function to reduce each row in such a way that
// there must be at least one zero in each row
int rowReduction(int reducedMatrix[N][N], int row[N])
{
   // initialize row array to INF
   fill_n(row, N, INF);

   // row[i] contains minimum in row i
   for (int i = 0; i < N; i++)
      for (int j = 0; j < N; j++)
         if (reducedMatrix[i][j] < row[i])
            row[i] = reducedMatrix[i][j];

   // reduce the minimum value from each element in each row
   for (int i = 0; i < N; i++)
      for (int j = 0; j < N; j++)
         if (reducedMatrix[i][j] != INF && row[i] != INF)
            reducedMatrix[i][j] -= row[i];
}

// Function to reduce each column in such a way that
// there must be at least one zero in each column
int columnReduction(int reducedMatrix[N][N], int col[N])
{
   // initialize col array to INF
   fill_n(col, N, INF);

   // col[j] contains minimum in col j
   for (int i = 0; i < N; i++)
      for (int j = 0; j < N; j++)
         if (reducedMatrix[i][j] < col[j])
            col[j] = reducedMatrix[i][j];

   // reduce the minimum value from each element in each column
   for (int i = 0; i < N; i++)
      for (int j = 0; j < N; j++)
         if (reducedMatrix[i][j] != INF && col[j] != INF)
            reducedMatrix[i][j] -= col[j];
}

// Function to get the lower bound on
// on the path starting at current min node
int calculateCost(int reducedMatrix[N][N])
{
   // initialize cost to 0
   int cost = 0;

   // Row Reduction
   int row[N];
```

```
    rowReduction(reducedMatrix, row);

    // Column Reduction
    int col[N];
    columnReduction(reducedMatrix, col);

    // the total expected cost
    // is the sum of all reductions
    for (int i = 0; i < N; i++)
        cost += (row[i] != INT_MAX) ? row[i] : 0,
            cost += (col[i] != INT_MAX) ? col[i] : 0;

    return cost;
}

// print list of cities visited following least cost
void printPath(vector<pair<int, int> > const &list)
{
    for (int i = 0; i < list.size(); i++)
        cout << list[i].first + 1 << " -> "
             << list[i].second + 1 << endl;
}

// Comparison object to be used to order the heap
struct comp {
    bool operator()(const Node* lhs, const Node* rhs) const
    {
        return lhs->cost > rhs->cost;
    }
};

// Function to solve Traveling Salesman Problem using Branch and
Bound
int solve(int costMatrix[N][N])
{
    // Create a priority queue to store live nodes of search tree;
    priority_queue<Node*, std::vector<Node*>, comp> pq;

    vector<pair<int, int> > v;

    // create a root node and calculate its cost
    // The TSP starts from first city i.e. node 0
    Node* root = newNode(costMatrix, v, 0, -1, 0);

    // get the lower bound of the path starting at node 0
    root->cost = calculateCost(root->reducedMatrix);

    // Add root to list of live nodes;
    pq.push(root);

    // Finds a live node with least cost, add its children to list of
```

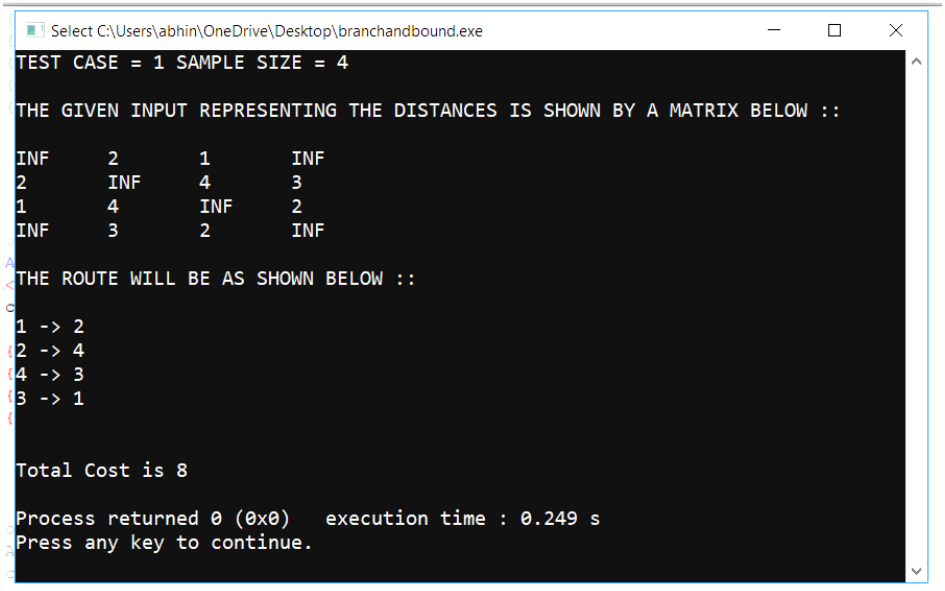
```
// live nodes and finally deletes it from the list
while (!pq.empty())
{
   // Find a live node with least estimated cost
   Node* min = pq.top();

   // The found node is deleted from the list of live nodes
   pq.pop();

   // i stores current city number
   int i = min->vertex;

   // if all cities are visited
   if (min->level == N - 1)
   {
      // return to starting city
      min->path.push_back(make_pair(i, 0));

      // print list of cities visited;
      printPath(min->path);

      // return optimal cost
      return min->cost;
   }

   // do for each child of min
   // (i, j) forms an edge in space tree
   for (int j = 0; j < N; j++)
   {
      if (min->reducedMatrix[i][j] != INF)
      {
         // create a child node and calculate its cost
         Node* child = newNode(min->reducedMatrix, min->path,
            min->level + 1, i, j);

         /* Cost of the child =
            cost of parent node +
            cost of the edge(i, j) +
            lower bound of the path starting at node j
         */
         child->cost = min->cost + min->reducedMatrix[i][j]
                  + calculateCost(child->reducedMatrix);

         // Add child to list of live nodes
         pq.push(child);
      }
   }

   // free node as we have already stored edges (i, j) in vector.
   // So no need for parent node while printing solution.
   delete min;
```

```
    }
}

// main function
int main()
{
    // cost matrix for traveling salesman problem.

   ///SAMPLE TEST CASE 1
/*
    cout<<"TEST CASE = 1 SAMPLE SIZE = "<<N<<endl;
    int costMatrix[N][N] =
    {
        {inf, 2,   1,   inf},
        {2,   inf, 4,   3},
        {1,   4,   inf, 2},
        {inf, 3,   2,   inf}
    };

    ///SAMPLE TEST CASE 2

    cout<<"TEST CASE = 2 SAMPLE SIZE = "<<N<<endl;
    int costMatrix[N][N] =
    {
        {inf, 3,   1,   5,   8},
        {3,   inf, 6,   7,   9},
        {1,   6,   inf, 4,   2},
        {5,   7,   4,   inf, 3},
        {8,   9,   2,   3,   inf}
    };


    ///SAMPLE TEST CASE 3

    cout<<"TEST CASE = 3 SAMPLE SIZE = "<<N<<endl;
    int costMatrix[N][N] =
    {
        {inf, 5,   inf, 6,   5,   4},
        {5,   inf, 2,   4,   3,   inf},
        {inf, 2,   inf, 1,   inf, inf},
        {6,   4,   1,   inf, 7,   inf},
        {5,   3,   inf, 7,   inf, 3},
        {4,   inf, inf, inf, 3,   inf}
    };



    ///SAMPLE TEST CASE 4

    cout<<"TEST CASE = 4 SAMPLE SIZE = "<<N<<endl;
    int costMatrix[N][N]={
```

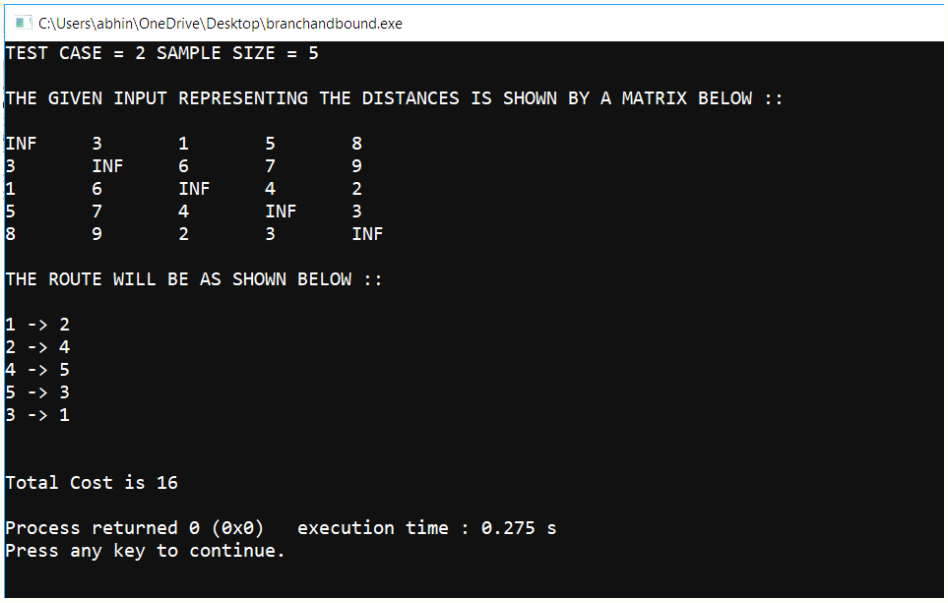
```
    {inf, 5,   inf, 6,   5,   4,    5},
    {5,   inf, 2,   24,   3,   inf,  8},
    {inf, 2,   inf, 1,   inf, inf,  1},
    {6,   4,   11,   inf, 37,   inf,  4},
    {5,   3,   inf, 7,   inf, 13,    5},
    {4,   inf, inf, inf, 3,   inf,  1},
    {8,   55,   22,   1,   inf,   3,  inf}
  };
*/

  ///SAMPLE TEST CASE 5

  cout<<"TEST CASE = 5 SAMPLE SIZE = "<<N<<endl;
  int costMatrix[N][N] =
  {
    {inf, 5,   inf, 6,   5,   4,    5,  2},
    {5,   inf, 2,   24,   3,   inf,  8,  1},
    {inf, 2,   inf, 1,   inf, inf,  1,  6},
    {6,   4,   11,   inf, 37,   inf,  4,  7},
    {5,   3,   inf, 7,   inf, 13,    5,  9},
    {4,   inf, inf, inf, 3,   inf,  1,  4},
    {8,   55,   22,   1,   inf,   3,  inf,  7},
    {2,   7,   2,   7,   inf,   32,  17, inf}
  };




      cout<<endl<<"THE GIVEN INPUT REPRESENTING THE
DISTANCES IS SHOWN BY A MATRIX BELOW :: "<<endl<<endl;
  for(int a=0;a<N;a++)
  {
    for(int b=0;b<N;b++)
    {
      if( costMatrix[a][b] == 999)
      {
        cout<<"INF\t";
      }
      else
      {
        cout<<costMatrix[a][b]<<"\t";
      }
    }
    cout<<endl;
  }
  cout<<endl<<"THE ROUTE WILL BE AS SHOWN BELOW ::
"<<endl<<endl;

  cout << "\n\nTotal Cost is " << solve(costMatrix)<<endl;

  return 0;
```

}

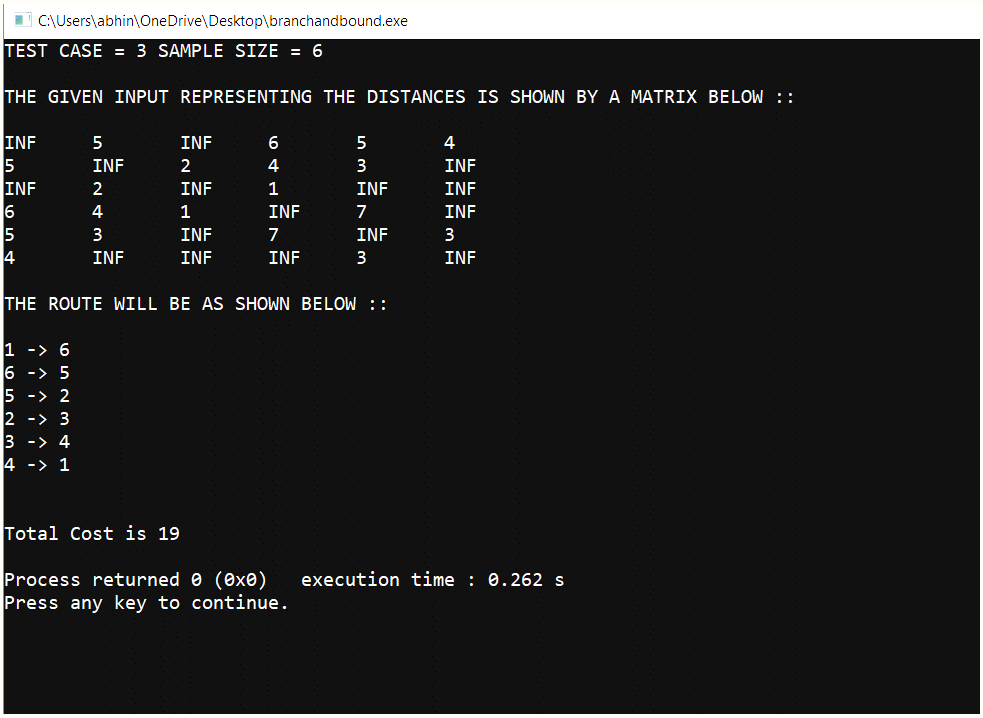
output 1 :



output 2 :



output 3 -



output 4 -

```
C:\Users\abhin\OneDrive\Desktop\branchandbound.exe
TEST CASE = 4 SAMPLE SIZE = 7

THE GIVEN INPUT REPRESENTING THE DISTANCES IS SHOWN BY A MATRIX BELOW ::

INF     5       INF     6       5       4       5
5       INF     2       24      3       INF     8
INF     2       INF     1       INF     INF     1
6       4       11      INF     37      INF     4
5       3       INF     7       INF     13      5
4       INF     INF     INF     3       INF     1
8       55      22      1       INF     3       INF

THE ROUTE WILL BE AS SHOWN BELOW ::

1 -> 6
6 -> 5
5 -> 2
2 -> 3
3 -> 7
7 -> 4
4 -> 1


Total Cost is 20

Process returned 0 (0x0)   execution time : 0.291 s
Press any key to continue.
```

output 5 -

```
C:\Users\abhin\OneDrive\Desktop\branchandbound.exe
TEST CASE = 5 SAMPLE SIZE = 8

THE GIVEN INPUT REPRESENTING THE DISTANCES IS SHOWN BY A MATRIX BELOW ::

INF     5       INF     6       5       4       5       2
5       INF     2       24      3       INF     8       1
INF     2       INF     1       INF     INF     1       6
6       4       11      INF     37      INF     4       7
5       3       INF     7       INF     13      5       9
4       INF     INF     INF     3       INF     1       4
8       55      22      1       INF     3       INF     7
2       7       2       7       INF     32      17      INF

THE ROUTE WILL BE AS SHOWN BELOW ::

1 -> 6
6 -> 5
5 -> 2
2 -> 8
8 -> 3
3 -> 7
7 -> 4
4 -> 1


Total Cost is 21

Process returned 0 (0x0)   execution time : 0.486 s
```

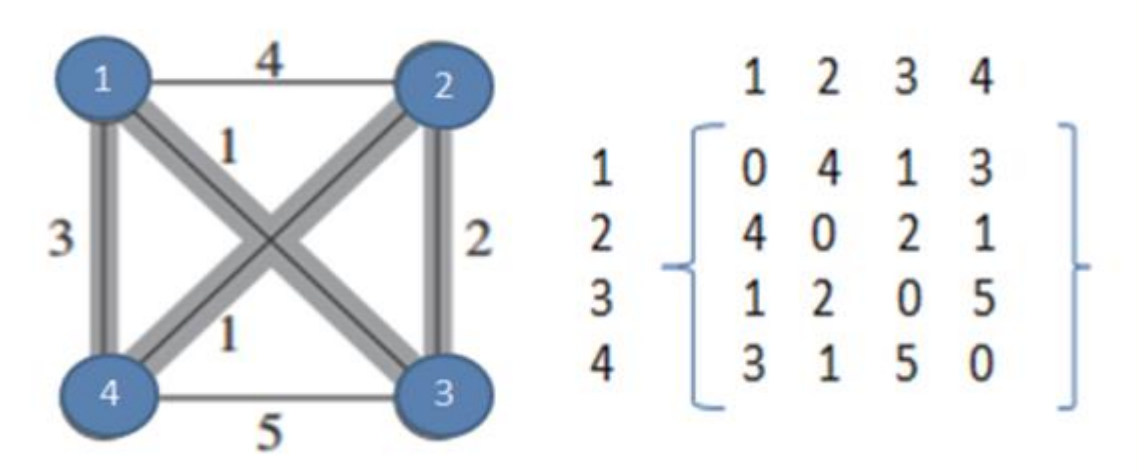
Review 3

METHOD-2

HELD KARP

Travelling Salesman Problem (TSP) Using Held Karp:

Example Problem



Above we can see a complete directed graph and cost matrix which includes distance between each village. We can observe that cost matrix is symmetric that means distance between village 2 to 3 is same as distance between village 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 village 1 and then he can go to any of {2,3,4}. From there to reach non-visited vertices (villages) becomes a new problem. Here we can observe that main problem spitted into sub-problem, this is property of held karp.

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

T (1, {2,3,4}) = 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.

T (2, {3,4}) = minimum of

= { (2,3) + T (3, {4}) 2+5=7
= { (2,4) + T {4, {3}) 1+5=6
T (3, {2,4}) = minimum of
= { (3,2) + T (2, {4}) 2+1=3
= { (3,4) + T {4, {2}) 5+1=6
T (4, {2,3}) = minimum of
= { (4,2) + T (2, {3}) 1+2=3
= { (4,3) + T {3, {2}) 5+2=7
T (3, {4}) = (3,4) + T (4, {}) 5+0=5
T (4, {3}) = (4,3) + T (3, {}) 5+0=5
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,

T (1, {2,3,4}) = minimum of = { (1,2) + T (2, {3,4})
4+6=10 in this path we have to add +1 because this path ends with 3.

From there we have to reach 1 so 3->1 distance 1 will be added total distance is 10+1=11 = {
(1,3) + T (3, {2,4}) 1+3=4 in this path we have to add +3 because this path ends with 3.

From there we have to reach 1 so 4->1 distance 3 will be added total distance is 4+3=7 = {
(1,4) + T (4, {2,3}) 3+3=6 in this path we have to add +1 because this path ends with 3.
From there we have to reach 1 so 3->1 distance 1 will be added total distance is 6+1=7

Minimum distance is 7 which includes path 1->3->2->4->1.

After solving example problem we can easily write recursive equation.
Recursive Equation

T (i , s) = min ((i , j) + T (j , S – { j })) ; S!= Ø ; j € S ;
S is set that contains non visited vertices = (i, 1) ; S=Ø, 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})).

But it is not guarantee that every vertex is connected to other vertex then we take that cost as infinity. After that we are taking minimum among all so the path which is not connected get infinity in calculation and won't be consider.

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:

- 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 $O\ (n2^n)$.
  - Each sub-problem will take $O(n)$ time (finding path to remaining (n-1) nodes).
  - Therefore total time complexity is $O(n2^n)* O\ (n) = O\ (n^2 2^n)$
  - Space complexity is also number of sub-problems which is $O\ (n2^n)$.

```
· If size of S is 2, then S must be {1, i},
· C(S, i) = dist(1, i)
· Else if size of S is greater than 2.
        · C(S, i) = min { C(S-{i}, j) + dis(j, i)} where j belongs to S, j != i and j != 1.
```

////////////////////// CODE OF HELD KARP//////////////////////

LANGUAGE- C

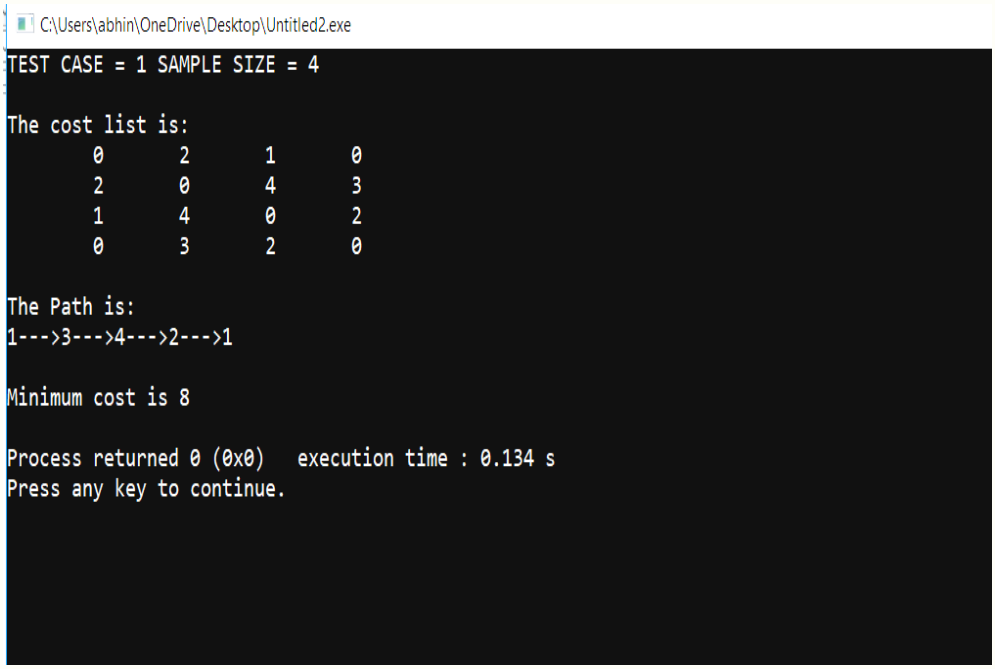
```
#include<stdio.h>

#define n 6
int ary[n][n],completed[n],cost=0;

void takeInput()
{
        int i,j,t;

        /*printf("Enter the number of villages: ");
        scanf("%d",&n);

        printf("\nEnter the 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",&ary[i][j]);

                completed[i]=0;
        }
   ///SAMPLE TEST CASE 1

   t=1;
   printf("TEST CASE = %d SAMPLE SIZE = %d",t,n);
   int temp[n][n] ={

      {0, 2,   1,   0},
      {2,   0, 4,   3},
      {1,   4,   0, 2},
      {0, 3,   2,   0}
   };

   ///SAMPLE TEST CASE 2
    t=2;
   printf("TEST CASE = %d SAMPLE SIZE = %d",t,n);
   int temp[n][n] =
   {
```

```
    {0, 3,  1,  5,  8},
    {3,  0, 6,  7,  9},
    {1,  6,  0, 4,  2},
    {5,  7,  4,  0, 3},
    {8,  9,  2,  3,  0}
  };


   ///SAMPLE TEST CASE 3
*/
  t=3;
  printf("TEST CASE = %d SAMPLE SIZE = %d",t,n);
  int temp[n][n] =
  {

    {0, 5,  0, 6,  5,  4},
    {5,  0, 2,  4,  3,  0},
    {0, 2,  0, 1,  0, 0},
    {6,  4,  1,  0, 7,  0},
    {5,  3,  0, 7,  0, 3},
    {4,  0, 0, 0, 3,  0}
  };

/*
  ///SAMPLE TEST CASE 4
  t=4;
  printf("TEST CASE = %d SAMPLE SIZE = %d",t,n);
  int temp[n][n] =
  {
    {0, 5,  0, 6,  5,  4,  5},
    {5,  0, 2,  24,  3,  0,  8},
    {0, 2,  0, 1,  0, 0,  1},
    {6,  4,  11,  0, 37,  0,  4},
    {5,  3,  0, 7,  0, 13,  5},
    {4,  0, 0, 0, 3,  0,  1},
    {8,  55,  22,  1,  0,  3,  0}
  };

 /*
  ///SAMPLE TEST CASE 5
  t=5;
  printf("TEST CASE = %d SAMPLE SIZE = %d",t,n);
  int temp[n][n] =
  {
    {0, 5,  0, 6,  5,  4,  5,  2},
    {5,  0, 2,  24,  3,  0,  8,  1},
    {0, 2,  0, 1,  0, 0,  1,  6},
    {6,  4,  11,  0, 37,  0,  4,  7},
    {5,  3,  0, 7,  0, 13,  5,  9},
    {4,  0, 0, 0, 3,  0,  1,  4},
    {8,  55,  22,  1,  0,  3,  0,  7},
```

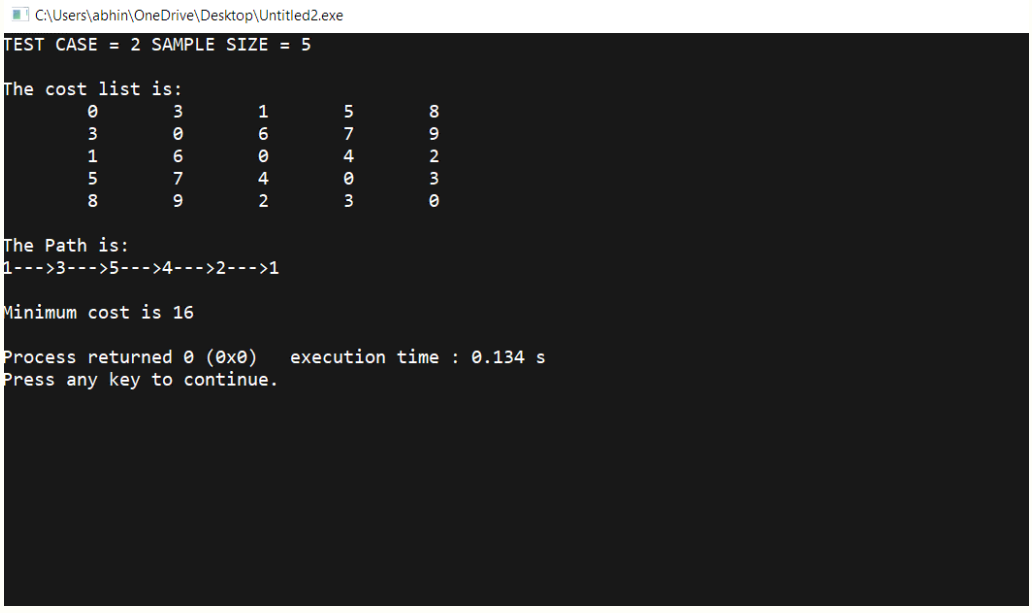
```
    {2,  7,  2,  7,  0,  32,  17, 0}
  };

/*
;*/

  for( i=0;i < n;i++)
      {
    for(j=0;j < n;j++)
                ary[i][j] = temp[i][j];

    completed[i]=0;
      }
      printf("\n\nThe cost list is:");

      for( i=0;i < n;i++)
      {
            printf("\n");

            for(j=0;j < n;j++)
                  printf("\t%d",ary[i][j]);
      }
}



int least(int c)
{
      int i,nc=999;
      int min=999,kmin;

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

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

      return nc;
}

void mincost(int city)
{
      int i,ncity;
```

```
        completed[city]=1;

        printf("%d--->",city+1);
        ncity=least(city);

        if(ncity==999)
        {
                ncity=0;
                printf("%d",ncity+1);
                cost+=ary[city][ncity];

                return;
        }

        mincost(ncity);
}

int main()
{
        takeInput();

        printf("\n\nThe Path is:\n");
        mincost(0); //passing 0 because starting vertex

        printf("\n\nMinimum cost is %d\n ",cost);

        return 0;
}
```

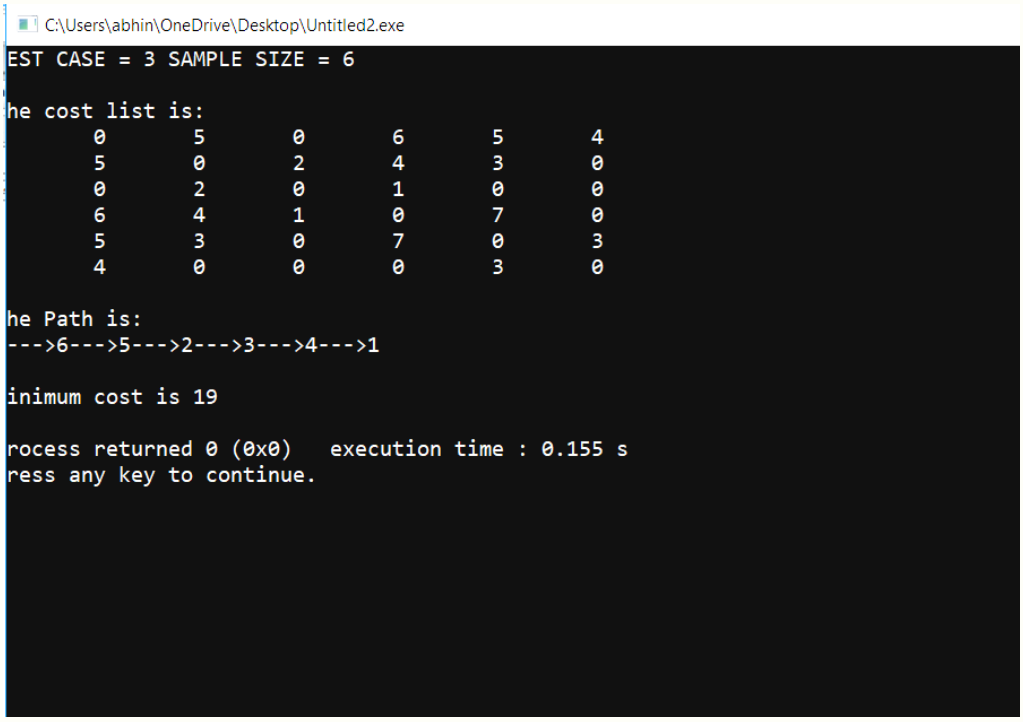
output 1 -



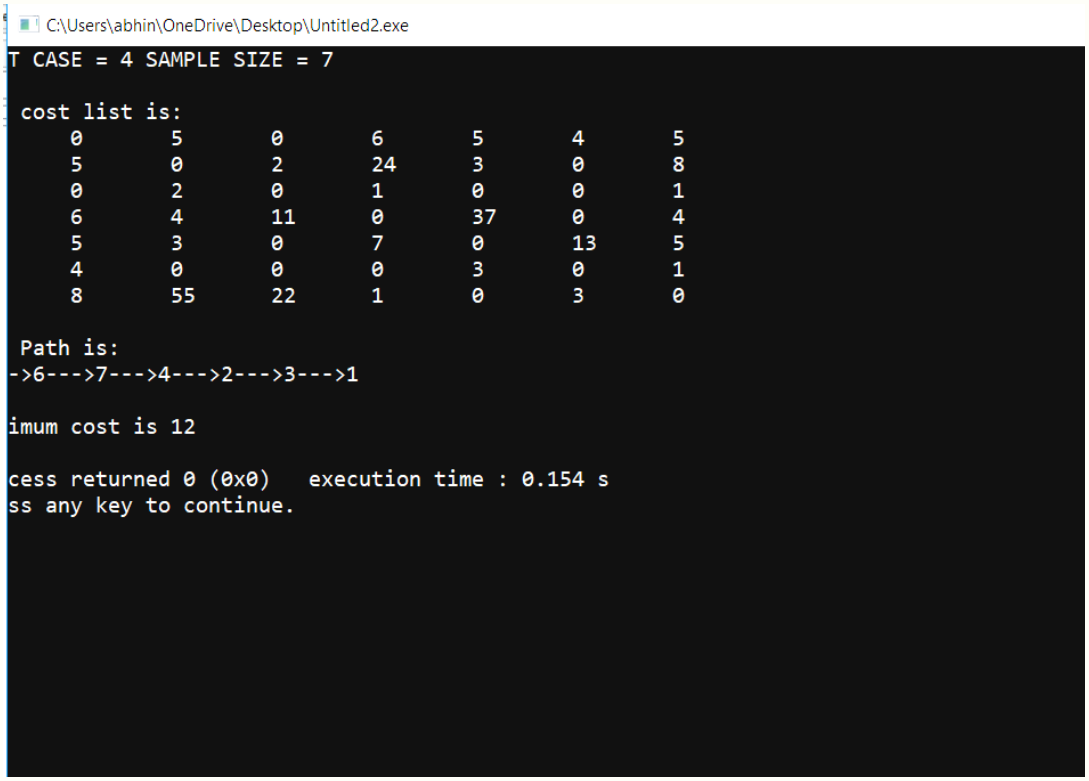
output 2 -



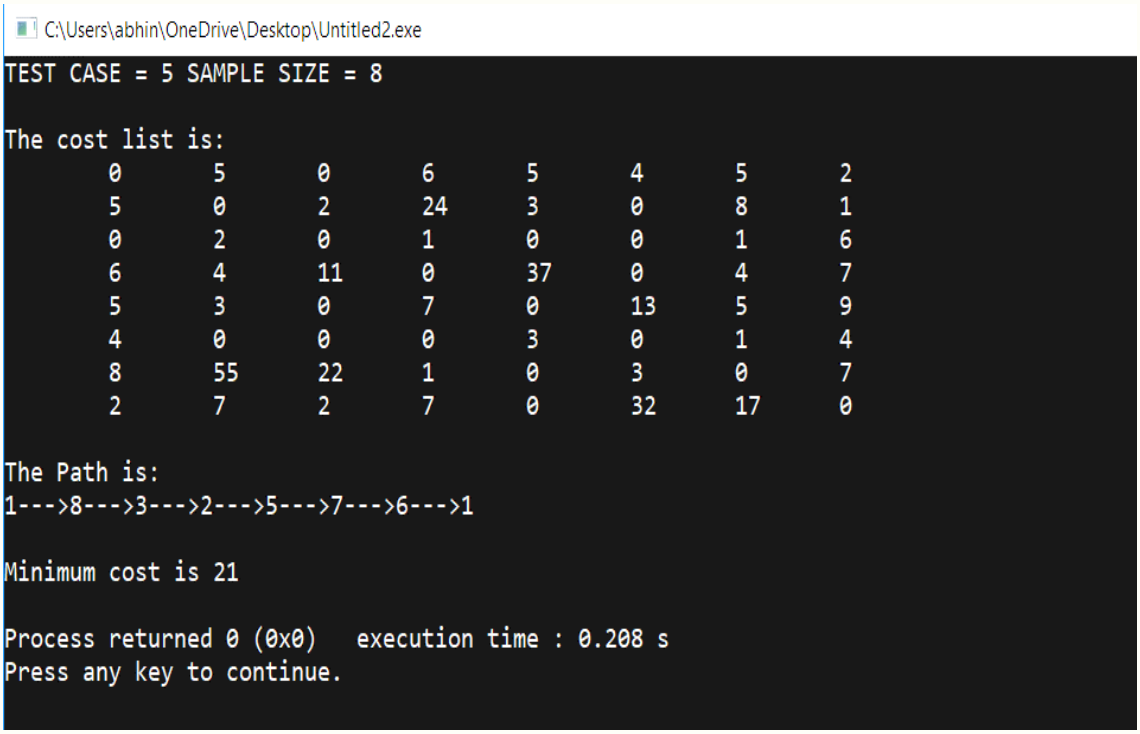
output 3 -



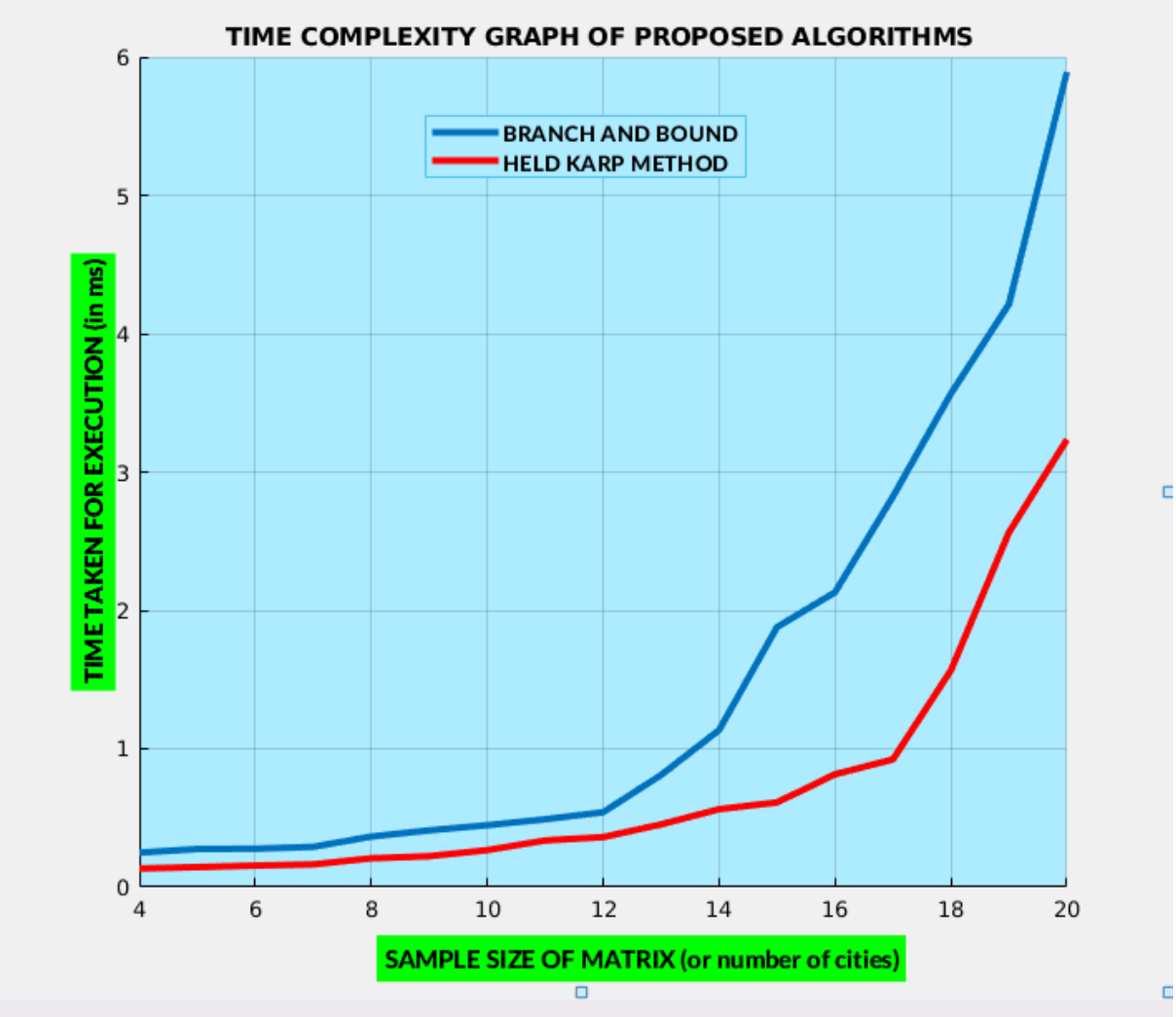
output 4 -



output 5 -


C:\Users\abhin\OneDrive\Desktop\Untitled2.exe
TEST CASE = 5 SAMPLE SIZE = 8
The cost list is:
0 5 0 6 5 4 5 2
5 0 2 24 3 0 8 1
0 2 0 1 0 0 1 6
6 4 11 0 37 0 4 7
5 3 0 7 0 13 5 9
4 0 0 0 3 0 1 4
8 55 22 1 0 3 0 7
2 7 2 7 0 32 17 0
The Path is:
1--->8--->3--->2--->5--->7--->6--->1
Minimum cost is 21
Process returned 0 (0x0) execution time : 0.208 s
Press any key to continue.

Time Complexity


TIME COMPLEXITY GRAPH OF PROPOSED ALGORITHMS
BRANCH AND BOUND
HELD KARP METHOD
TIME TAKEN FOR EXECUTION (in ms)
SAMPLE SIZE OF MATRIX (or number of cities)
0
1
2
3
4
5
6
4
6
8
10
12
14
16
18
20

CONCLUSION:

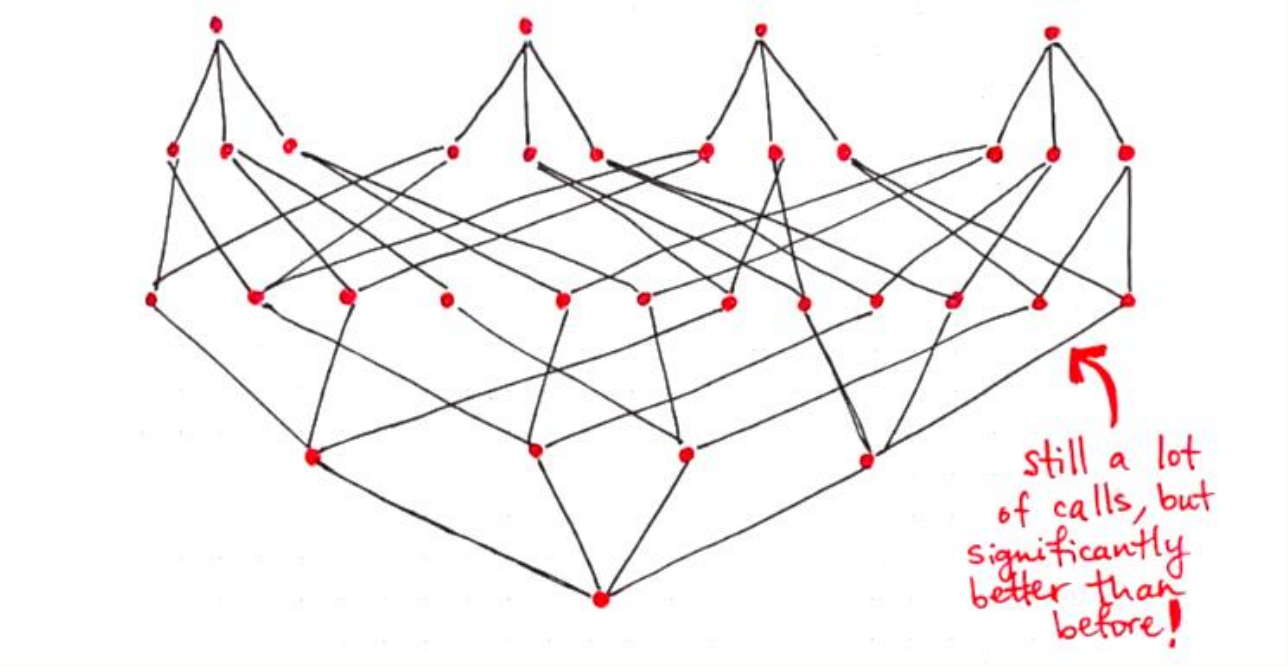
We first tried to solve tsp by using a top down approach in branch and bound, we started with one node and a single function call and use it to generate more calls recursively.

In held karp algorithm, we're starting with the more complex function call initially, and then, from within that, we are invoking three recursive function calls from within it. Each of those three recursive function calls spins off two more recursive calls of its own,

When we first tried to solve TSP, we used an adjacency matrix to help us keep track of the distances between nodes in our graph. We'll lean on our adjacency matrix in this approach yet again.

However, in our bottom up approach, we'll use it to enumerate all the function calls that lead to one another. This is strikingly different than our top down approach, when we were using our adjacency matrix to help us enumerate all the possible paths. In our bottom up approach, we're trying to be a possible bit more elegant about how we do things, so we're aiming to not enumerate more than we need to! This will make more sense as we go on, but it's important to note the difference between enumerating paths versus enumerating function calls.

For n=5 the

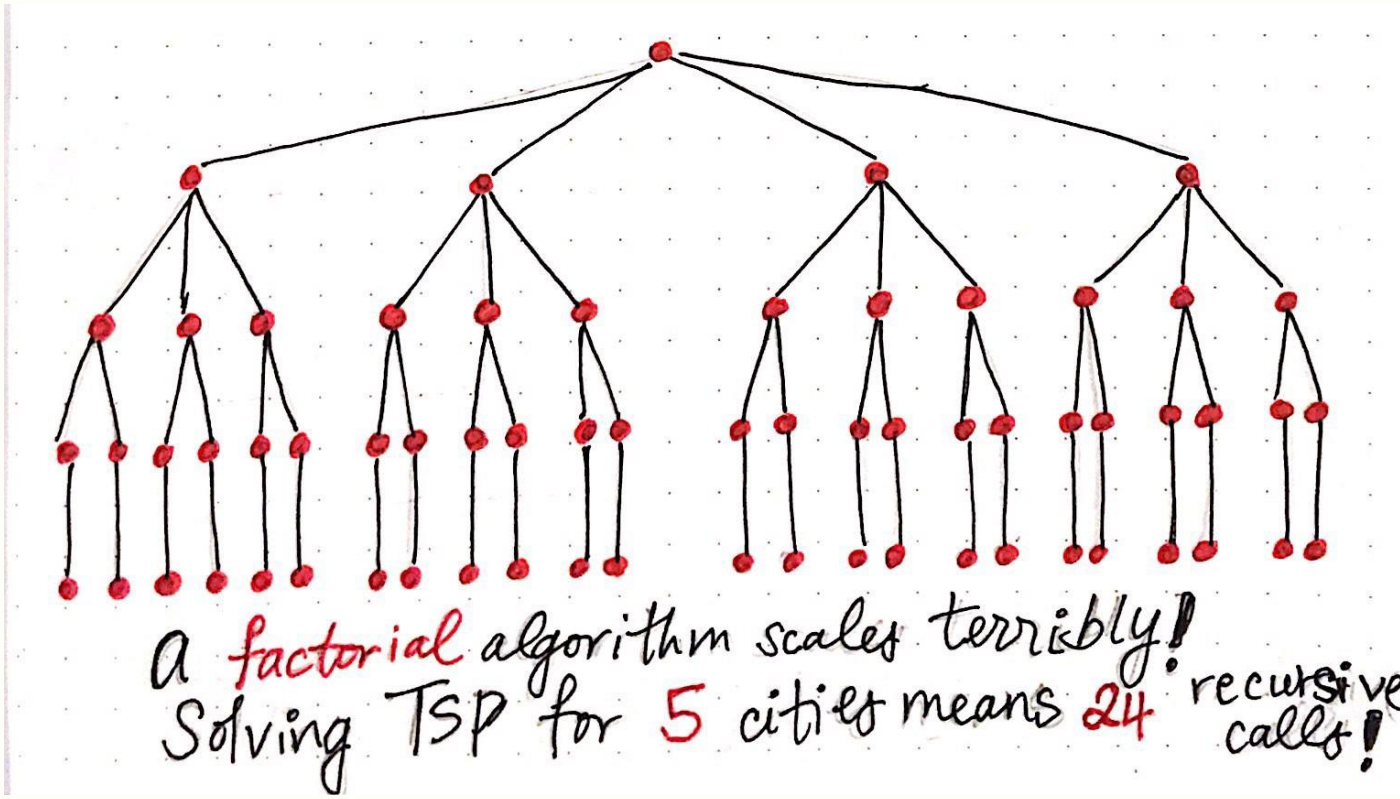


bottom up approach is used.

We no longer need to generate an entire tree to figure out the recursive calls and determine the paths, we saw the sub problem, and cut out the repeated task. Now it starts to become apparent how the bottom up approach is different than our top down method from before.

By using the bottom up approach, we've optimized our TSP algorithm, since we no longer have to generate all the recursive calls as big of a tree structure the held karp algorithm uses dynamic programming to reduce run time of tsp O(n*n!) to exponential time [(2^n)*(n^2)]. Through enumerating through all sets of function calls.

The Held-Karp algorithm actually proposed the bottom up dynamic programming approach as a solution to improving the brute-force method of solving the traveling salesman problem. Bellman, Held, and Karp's algorithm was determined to run in *exponential* time, since it still does a bulk of the work of enumerating through all the potential sets of function calls that are possible. The exponential runtime of the Held-Karp algorithm is still not perfect—it's far from it, in fact!



METHOD 3

GREEDY ALGORITHM

- A greedy algorithm is an algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum. In many problems, a greedy strategy does not in general produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a global optimal solution in a reasonable time.

- For example, a greedy strategy for the traveling salesman problem (which is of a high computational complexity) is of the following heuristic: "At each stage visit an unvisited city nearest to the current city". This heuristic need not find a best solution, but terminates in a reasonable number of steps; finding an optimal solution typically requires unreasonably many steps.

If a Greedy Algorithm can solve a problem, then it generally becomes the best method to solve that problem as the Greedy algorithms are in general more efficient than other techniques like Held Karp. But Greedy algorithms cannot always be applied. For example, Fractional Knapsack problem can be solved using Greedy, but 0-1 Knapsack cannot be solved using Greedy.

The following are the steps of the greedy algorithm for a travelling salesman problem:

Step 1: input the distance matrix, $[D_{ij}]i = 1, 2, 3, ..., n$, where n is the number of nodes in the distance network.

Step 2: Randomly select a base city, let it be X and delete the column X of the distance matrix

Step 3: Include X as the first city in the tour.

Step 4: in the row X, find the least undeleted matrix cell entry and identify the corresponding column (break tie randomly), let this column be Y.

Step 5: include Y as the next city to be visited in the tour.

Step 6: Delete the column Y of the distance matrix.

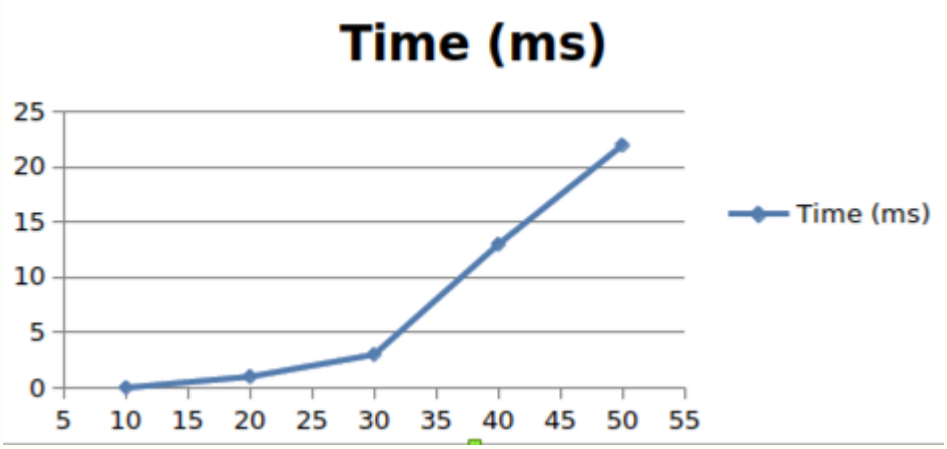
Step 7: Check whether all the columns of the distance matrix are deleted. If yes, to *Step 9:* otherwise go to step 8.

Step 8: Set X = Y and go to step 4.

Step 9: include the first city as the last city in the tour.

Step 10: list the cities in the tour along with the corresponding total distance of

travel.



References:

1.Jin L, Tsai S, Yang J, Tsai J, Kao C. An evolutionary algorithm for large traveling salesman problems. IEEE
Transactions on systems, Man and Cybernetics-Part B: Cybernetics. 2004; 34(4):1718– 29.

2.Basu S, Ghosh D. A Review of the Tabu Search Literature on Traveling Salesman Problems. Indian
Institute of Management. 2008; 10(1):1–16.

3.Basu S. Tabu Search Implementation on Traveling Salesman Problem and Its Variations: A Literature
Survey. American Journal of Operations Research. 2012; 2:163–73.

4. N. Sathyaand A. Muthukumaravelon a Review of the Optimization Algorithms on Traveling Salesman Problem
in Indian Journal of Science and Technology.

5. Scoti'm.Graham on Thetravelingsalesmanproblem:A 5 hierarchicalmodelMemory & Cognition 2000, 28 (7),
J191 -1204.

6.Naixue Xiong, Wenliang Wuand Chunxue Wu on an Improved Routing Optimization Algorithm Basedon
Travelling Salesman Problem for Social Networks sustainability 2017, 9, 985

7.Sanchit Goyal on a Survey on Travelling Salesman Problem

8.Miller and J. Pekny (1991). "Exact Solution of Large Asymmetric Traveling Salesman Problems," Science
- *251, 754 -761.*

9. https://www.slideshare.net/kaalnath/comparisonoftsp- algorithms?from_action=save

10.Ravindra K Ahuja, Özlem Ergun, James B Orlin, Abraham P Punnen:
https://scholar.google.co.in/citations?view_op=view_citation&hl=en&user=6YYJk1QAAAAJ&citation_for_view=6YYJk1QAAAAJ:u-x6o8ySG0sC

11.http://www.win.tue.nl/~kbuchin/teaching/2IL15/backtracking.pdf

12.https://www.intechopen.com/books/travelingsalesman -problem-theory-and-applications/travelingsalesmanproblem- an-overview-of-applicationsformulations-andsolution- approaches

13.https://web.engr.oregonstate.edu/~tgd/classes/534/slides/part3.pdf

14.https://scholar.google.co.in/scholar?hl=en&as_sdt=0% 2C5&q=travvelling+sales+man+problem&btnG=

15.http://ieeexplore.ieee.org/abstract/document/585892/22.http://ieeexplore.ieee.org/abstract/document/676772/

16. http://www.ams.org/journals/proc/1956-007-01/S0002-9939-1956-0078686-7/S0002-9939-1956-0078686-7.pdf

Base Papers:

Branch and bound- *http://cs.indstate.edu/cpothineni/alg.pdf*

Held Karp-
https://stanford.edu/~rezab/classes/cme323/S16/projects_reports/burton.pdf