Analysis, Comparison and Optimization of different algorithms used in Travelling Salesman Problem (TSP)

Author Krupal J. Kathrotia (17BCE0589), Under guidance of: Sir Boominathan

***Abstract*— The aim of this project is to discuss various aspects and techniques proposed to solve Travelling salesman problem. PROBLEM: The problem involves of traveling N cities (nodes) given the distance between all cities (every two pair of cities) with two major constraints:1)Travelling all the cities only once.2)Coming back at the starting city (vertex).The travelling-salesman problem is one of the classical NP-Complete hard problems for which no algorithms are available which can solve it in polynomial time (steps increases as a polynomial depends on size of input).The problem aims at finding minimum distance to be traversed or finding shortest possible route. Total possible routes are N! and so the steps increase factorially as the number of cities increases. Analysis on three ways of approaches can be made: Iterative algorithm, a recursive algorithm, and a branch and bound algorithm. The iterative algorithm generates tours as the permutations of the first n-1 integer. The recursive algorithm begins with a subtour consisting of the starting city and uses a recursive subroutine to build all tours. The branch and bound algorithm builds a tree which represents tours. Each node of the tree has an associated bound, and when the bound of a node becomes larger than the cost of the best tour found so far, that node is no longer eligible for exploration. We shall discuss various aspects such as efficiency in varying cases, prerequisite knowledge of programming and complexity of different algorithms and to review how genetic algorithm applied to these problems and find an efficient solution.**

***Index Terms*—Travelling Salesamn Problem, Classical NP-Complete hard problem, Iterative alogorithm, Recursive Algorithm, branch and bound, genetic algorithm, subtour**

I. 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.

Asymmetric and symmetric: In the symmetric TSP, the distance between two cities is the same in each opposite direction, forming an undirected graph. This symmetry halves the number of possible solutions. In the asymmetric TSP, paths may not exist in both directions or the distances might be different, forming a directed graph. Traffic collisions, one-way streets, and airfares for cities with different departure and arrival fees are examples of how this symmetry could break down.

∗ Corresponding author.
E-mail addresses: krupalkathrotia1729@gmail.com (Krupal J. Kathrotia), Reg. No. : 17BCE0589

The travelling purchaser problem and the vehicle routing problem are both generalizations of TSP.

The related problems are Hamiltonian path problem, Bottleneck traveling salesman problem (bottleneck TSP) , generalized travelling salesman problem (travelling politician problem) and travelling purchaser problem.

II. LITERATURE SURVEY

Genetic algorithm (GA) as a computational intelligence method is a search technique used in computer science to find approximate solutions to combinatorial optimization problems.

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

III. PROBLEM FORMULATION

A. FLIGHT TOUR:

TYPE: ASYMMETRIC TSP

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.

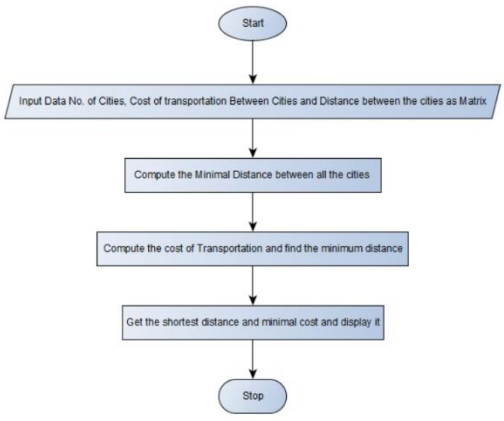
B. SCHOOL BUS TRANSPORTATION:

TYPE: SYMMETRIC TSP

APPROACH: HELD KARP, ANT COLONY OPTIMISATION, 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.
Newer approaches can be made to optimally solve this type of symmetric TSP.

IV. FLOW DIAGRAM



V. APPROACH

A. DETERMINISTIC APPROACH:

One of the earliest deterministic solutions to TSP was provided by Dantzig et al., in which linear programming (LP) relaxation is used to solve the integer formulation by adding suitably chosen linear inequality to the list of constraints continuously. Held and Karp presented a dynamic programming formulation for an exact solution of the travelling salesman problem, however it has a very high space complexity, which makes it very inefficient for higher values.

B. NON-DETERMINISTIC APPROACH:

The exact solutions provide an optimal tour for TSP for every instance of the problem; however their inefficiency makes it unfeasible to use those solutions in practical applications. Therefore Non−Deterministic solution approach is more useful for the applications which prefer time of run of the algorithm over the accuracy of the result. There has been a vast research in past to solve the TSP for an approximate result. (for example, “IEEE” in the title of this article).

VI. COMPUTING SOLUTION

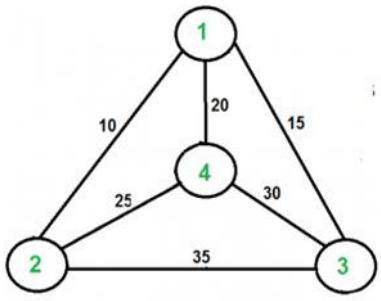
WAYS OF APPROACHES:

A. Exact algorithms
B. Suboptimal or heuristic algorithms
C. Special cases for the problem (subproblems)

A. Exact algorithms:

(i) Naive solution:

Generate all (n-1)! Permutations of cities.
Calculate cost of every permutation and keep track of minimum cost permutation.
Return the permutation with minimum cost.
Consider city 0 as the starting and ending point.



TIME COMPLEXITY: O(n!)

B. *Suboptimal or* heuristic algorithms:

Triangle-Inequality: The least distant path to reach a vertex j from i is always to reach j directly from i, rather than through some other vertex k (or vertices), i.e., dis(i, j) is always less than or equal to dis(i, k) + dist(k, j). The Triangle-Inequality holds in many practical situations.

(i) MST (Minimum Spanning Tree):

Let 1 be the starting and ending point for salesman.
Construct MST from 1 as root using Prim’s Algorithm.
List vertices visited in preorder walk of constructed MST and add 1 at the end.
TIME COMPLEXITY: Brute force (trial and error or differ for different values of n)

(ii) Christofides' Algorithm

(A) ITERATIVE IMPROVEMENT:

(1) Ant Colony Optimization
(2) Pairwise Exchange
(3) k-opt Heuristic or Lin–Kernighan Heuristics
(4) V-opt Heuristic
(5) Randomized Improvement

(B) DYNAMIC APPROACH:

(i) Held Karp Method:

For every other vertex i (other than 1), we find the minimum cost path with 1 as the starting point, i as the ending point and all vertices appearing exactly once. Let the cost of this path be cost(i), the cost of corresponding Cycle would be cost(i) + dist(i, 1) where dist(i, 1) is the distance from i to 1. Finally, we return the minimum of all [cost(i) + dist(i, 1)] values.

Let us define a term *C(S, i) be the cost of the minimum cost path visiting each vertex in set S exactly once, starting at 1 and ending at i.*

- 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.

(ii) Nearest Neighbor Algorithm (NN) (greedy algorithm) (Constructive heuristics)

It follows a very a very simple greedy procedure. The algorithm starts with a tour containing a randomly chosen city and then always adds to the last city in the tour the nearest nor yet visited city. The algorithm stops when all cities are on the tour.

(iii) BRANCH AND BOUND ALGORITHM:

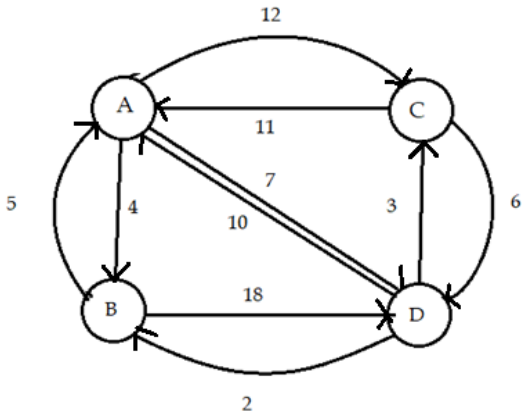
In Branch and Bound method, for current node in tree, we compute a bound on best possible solution that we can get if we down this node. If the bound on best possible solution itself is worse than current best (best computed so far), then we ignore the subtree rooted with the node

C. Special cases for the problem (subproblems)

(1) Metric TSP (minimum spanning tree)
(2) Euclidean TSP (Euclidean TSP is a particular case of metric TSP)
(3) Asymmetric TSP (Solving by conversion to symmetric TSP)

VII. PROPOSED METHODOLOGY

FRAMEWORK FOR BRANCH AND BOUND ALGORITHM TO SOLVE TSP:



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	∞

REDUCTION:

	A	B	C	D	reduce by
A	8	4	12	7	**4**
B	5	8	8	18	**5**
C	11	8	8	6	**6**
D	10	2	3	8	**2**

AFTER ROW REDUCTION →

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

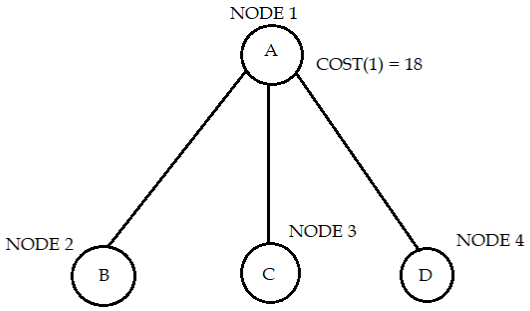
	A	B	C	D
A	8	0	7	3
B	0	8	8	13
C	5	8	8	0
D	8	0	0	8

AFTER COLUMN REDUCTION ←

Therefore, COST OF NODE 1 is :
COST(1) = REDUCTION
= 4+5+6+2+1
= 18
SO, after step1,

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] = ∞.

Resultant cost matrix

	A	B	C	D
A	8	8	8	8
B	8	8	8	13
C	5	8	8	0
D	8	8	0	8

AFTER ROW AND COLUMN REDUCTION →

Reduced matrix after row and column reduction

	A	B	C	D	
A	**8**	**8**	**8**	**8**	
B	**8**	**8**	**8**	**0**	(13)
C	0	8	8	0	
D	3	8	0	8	

(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	8	8	8	8
B	0	8	8	13
C	8	8	8	0
D	8	0	8	8

AFTER ROW AND COLUMN REDUCTION →

Reduced matrix after row and column reduction

	A	B	C	D
A	**8**	**8**	**8**	**8**
B	0	**8**	**8**	13
C	8	8	8	0
D	8	0	8	8

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	8	8	8	8
B	0	8	8	8
C	5	8	8	8
D	8	0	0	8

AFTER ROW AND COLUMN REDUCTION →

Reduced matrix after row and column reduction

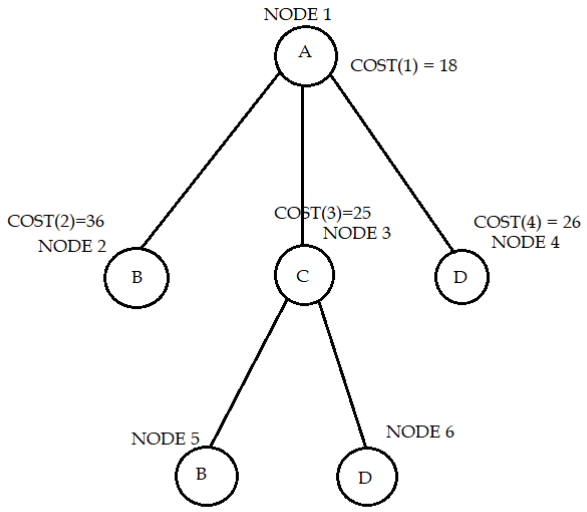
	A	B	C	D	
A	**8**	**8**	**8**	**8**	
B	**0**	**8**	**8**	**8**	
C	0	8	8	8	(5)
D	8	0	0	8	

Thus, cost of node-4 is:
COST (4) = COST (1) + REDUCTION + M [A, D]
= 18+5+3+0
=26

Thus, we have:

COST (2) = 35
COST (3) = 25 (minimal)
COST (4) = 26

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	8	8	8	8
B	8	8	8	13
C	8	8	8	8
D	8	8	8	8

AFTER ROW AND COLUMN REDUCTION →

Reduced matrix after row and column reduction

	A	B	C	D	
A	∞	∞	∞	∞	
B	∞	∞	∞	0	(13)
C	8	8	8	8	
D	0	8	0	8	(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] = ∞.

Thus, cost of node-6 is:
COST (6) = COST (3) + REDUCTION + M [C, D]
= 25+0+0
= 25

Resultant cost matrix

	A	B	C	D
A	8	8	8	8
B	0	8	8	8
C	8	8	8	8
D	8	8	8	8

AFTER ROW AND COLUMN REDUCTION →

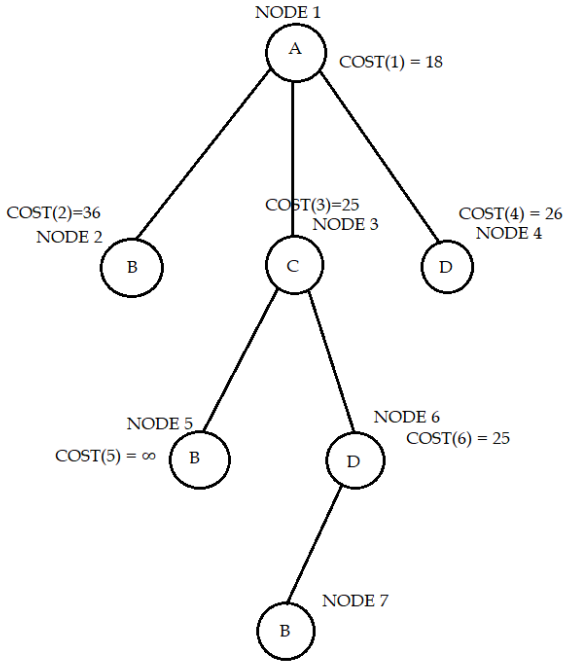
Reduced matrix after row and column reduction

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

."

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.

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	8	8	8	8
B	8	8	8	8
C	8	8	8	8
D	8	8	8	8

AFTER ROW AND COLUMN REDUCTION →

Reduced matrix after row and column reduction

	A	B	C	D	
A	8	8	8	8	
B	8	8	8	8	(13)
C	8	8	8	8	
D	8	8	8	8	(8)

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)
```

VIII. CONCLUSION AND FUTURE WORK

The worst case complexity of Branch and Bound remains same as that of the Brute Force clearly because in worst case, we may never get a chance to prune a node. Whereas, in practice it performs very well depending on the different instance of the TSP. The complexity also depends on the choice of the bounding function as they are the ones deciding how many nodes to be pruned. a B&B algorithm performs a top-down recursive search through the tree of instances formed by the branch operation. Upon visiting an instance *I*, it checks whether bound(*I*) is greater than the upper bound for some other instance that it already visited; if so, *I* may be safely discarded from the search and the recursion stops. This pruning step is usually implemented by maintaining a global variable that records the minimum upper bound seen among all instances examined so far.

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.

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.E. Balas (1989). "The Prize Collecting Traveling Salesman Problem," Networks 19,621 -636

10. N. Christofides (1985). "Vehicle Routing," in The Traveling Salesman Problem,Lawler, Lenstra, Rinooy Kan and Shmoys, eds., John Wiley, 431 -448

11. Dantzig GB, Fulkerson DR, Johnson SM (1954). Solution of a Large -scale TravelingSalesman Problem." Operations Research, 2, 393410

12.Held M, Karp RM (1962). A Dynamic Programming Approach to Sequencing Problems." Journal of SIAM, 10, 196 – 210

13. https://www.slideshare.net/kaalnath/comparison-oftsp-algorithms?from_action=save

14.https://en.wikipedia.org/wiki/Nearest_neighbour_alg or ithm

15.https://scholar.google.co.in/scholar?hl=en&as_sdt= 0% 2C5&q=travleeing+sales+man+preoblem&btnG=

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

17.http://www.win.tue.nl/~kbuchin/teaching/2IL15/backt r acking.pdf

18.https://www.intechopen.com/books/travelingsalesman -problem-theory-and-applications/travelingsalesman-problem-an-overview-of-applicationsformulations-and-solution-approaches

19.https://web.engr.oregonstate.edu/~tgd/classes/534/sli d es/part3.pdf

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

21.http://ieeexplore.ieee.org/abstract/document/585892/

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

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

TRAVELLING SALESMAN PROBLEM

ASSIGNMENT VIEW

NAME: KRUPAL J. KATHROTIA REG. NO.: 17BCE0589

Analysis, Comparison and Optimization of different algorithms used in Travelling Salesman Problem (TSP)

ABSTRACT

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

- PROBLEM: The problem involves of traveling N cities (nodes) given the distance between all cities (every two pair of cities) with two major constraints:
  (1) Travelling all the cities only once.
  (2) Coming back at the starting city (vertex).
- The travelling-salesman problem is one of the classical NP-Complete hard problems for which no algorithms are available which can solve it in polynomial time (steps increases as a polynomial depends on size of input).
- The problem aims at finding minimum distance to be traversed or finding shortest possible route. Total possible routes are N! and so the steps increase factorially as the number of cities increases.
- Analysis on three ways of approaches can be made: Iterative algorithm, a recursive algorithm, and a branch and bound algorithm.
- The iterative algorithm generates tours as the permutations of the first n-1 integer. The recursive algorithm begins with a subtour consisting of the starting city and uses a recursive subroutine to build all tours. The branch and bound algorithm builds a tree which represents tours. Each node of the tree has an associated bound, and when the bound of a node becomes larger than the cost of the best tour found so far, that node is no longer eligible for exploration.
- **We shall discuss various aspects such as efficiency in varying cases, prerequisite knowledge of programming and complexity of different algorithms and to review how genetic algorithm applied to these problems and find an efficient solution.**

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.

- **Asymmetric and symmetric**: In the symmetric TSP, the distance between two cities is the same in each opposite direction, forming an undirected graph. This symmetry halves the number of possible solutions. In the asymmetric TSP, paths may not exist in both directions or the distances might be different, forming a directed graph. Traffic collisions, one-way streets, and airfares for cities with different departure and arrival fees are examples of how this symmetry could break down.

- The travelling purchaser problem and the vehicle routing problem are both generalizations of TSP.
- The related problems are Hamiltonian path problem, Bottleneck traveling salesman problem (bottleneck TSP) , generalized travelling salesman problem (travelling politician problem) and travelling purchaser problem.

REAL TIME PROBLEM (CASE STUDY)

❖ FLIGHT TOUR:

✓ TYPE: ASYMMETRIC TSP

✓ 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.

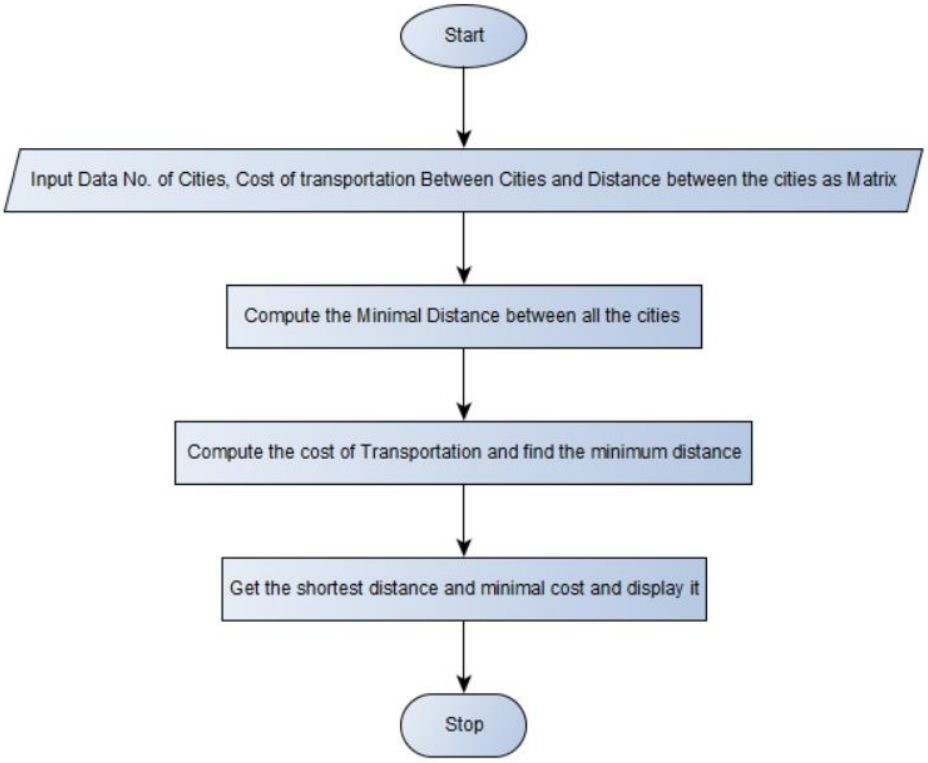
❖ SCHOOL BUS TRANSPORTATION:

✓ TYPE: SYMMETRIC TSP

✓ APPROACH: HELD KARP, ANT COLONY OPTIMISATION, 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.
- Newer approaches can be made to optimally solve this type of symmetric TSP.

FLOWCHART


Start
Input Data No. of Cities, Cost of transportation Between Cities and Distance between the cities as Matrix
Compute the Minimal Distance between all the cities
Compute the cost of Transportation and find the minimum distance
Get the shortest distance and minimal cost and display it
Stop

LITERATURE SURVEY

- Genetic algorithm (GA) as a computational intelligence method is a search technique used in computer science to find approximate solutions to combinatorial optimization problems.
- 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.

- DETERMINISTIC APPROACH:

➢ One of the earliest deterministic solutions to TSP was provided by Dantzig et al., in which linear programming (LP) relaxation is used to solve the integer formulation by adding suitably chosen linear inequality to the list of constraints continuously. Held and Karp presented a dynamic programming formulation for an exact solution of the travelling salesman problem, however it has a very high space complexity, which makes it very inefficient for higher values.

- NON-DETERMINISTIC APPROACH:

➢ The exact solutions provide an optimal tour for TSP for every instance of the problem; however their inefficiency makes it unfeasible to use those solutions in practical applications. Therefore Non−Deterministic solution approach is more useful for the applications which prefer time of run of the algorithm over the accuracy of the result. There has been a vast research in past to solve the TSP for an approximate result. Some of the implemented approximate algorithms as described in are being listed here:

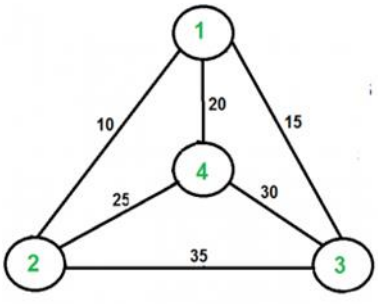
COMPUTING SOLUTION

❖ WAYS OF APPROACHES:

- ✓ Exact algorithms
- ✓ Suboptimal or heuristic algorithms
- ✓ Special cases for the problem (subproblems)

❖ EXACT ALGORITHMS:

(1) Naive solution:



- ➢ Consider city 0 as the starting and ending point.
- ➢ Generate all (n-1)! Permutations of cities.
- ➢ Calculate cost of every permutation and keep track of minimum cost permutation.
- ➢ Return the permutation with minimum cost.

TIME COMPLEXITY: O(n!)

❖ APPROXIMATION ALGORITHMS (HEURISTIC ALGORITM):

- ➢ **Triangle-Inequality:** The least distant path to reach a vertex j from i is always to reach j directly from i, rather than through some other vertex k (or vertices), i.e., dis(i, j) is always less than or equal to dis(i, k) + dist(k, j). The Triangle-Inequality holds in many practical situations.

(1) MST (Minimum Spanning Tree):

- ➢ Let 1 be the starting and ending point for salesman.
- ➢ Construct MST from 1 as root using Prim's Algorithm.

- List vertices visited in preorder walk of constructed MST and add 1 at the end.

TIME COMPLEXITY: Brute force (trial and error or differ for different values of n)

(2) Christofides' Algorithm

❖ ITERATIVE IMPROVEMENT:

(1) Ant Colony Optimization
(2) Pairwise Exchange
(3) k-opt Heuristic or Lin–Kernighan Heuristics
(4) V-opt Heuristic
(5) Randomized Improvement

❖ DYNAMIC APPROACH:

(1) Held Karp Method:

- For every other vertex i (other than 1), we find the minimum cost path with 1 as the starting point, i as the ending point and all vertices appearing exactly once. Let the cost of this path be cost(i), the cost of corresponding Cycle would be cost(i) + dist(i, 1) where dist(i, 1) is the distance from i to 1. Finally, we return the minimum of all [cost(i) + dist(i, 1)] values.
- Let us define a term C(S, i) be the cost of the minimum cost path visiting each vertex in set S exactly once, starting at 1 and ending at i.

```
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.
```

(2) Nearest Neighbor Algorithm (NN) (greedy algorithm) (Constructive heuristics)

- It follows a very a very simple greedy procedure. The algorithm starts with a tour containing a randomly chosen city and then always adds to the last city in the tour the nearest nor yet visited city. The algorithm stops when all cities are on the tour.

(3) **BRANCH AND BOUND ALGORITHM:**

- In Branch and Bound method, for current node in tree, we compute a bound on best possible solution that we can get if we down this node. If the bound on best possible solution itself is worse than current best (best computed so far), then we ignore the subtree rooted with the node

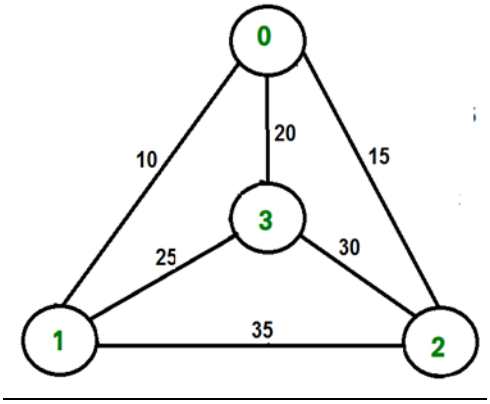
❖ SPECIAL CASES FOR THE PROBLEM (SUBPROBLEMS):

(1) Metric TSP (minimum spanning tree)
(2) Euclidean TSP (Euclidean TSP is a particular case of metric TSP)
(3) Asymmetric TSP (Solving by conversion to symmetric TSP)

❖ 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.

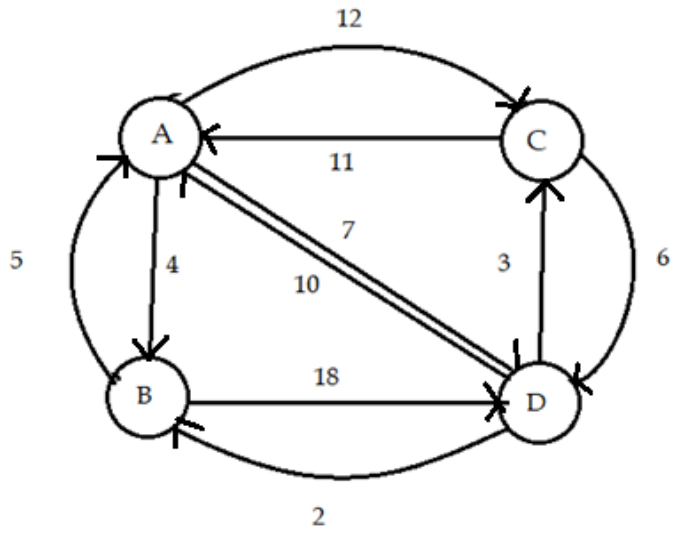
Consider any one node except starting (ending) node as our pivot node and the find all possible path from starting to pivot city via different number of cities.

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 naïve	permutations in this approach
4	6	5
5	24	15
6	120	64
7	720	301
…	….	…

❖ 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	∞

- **REDUCTION:**

	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 COLUMN REDUCTION

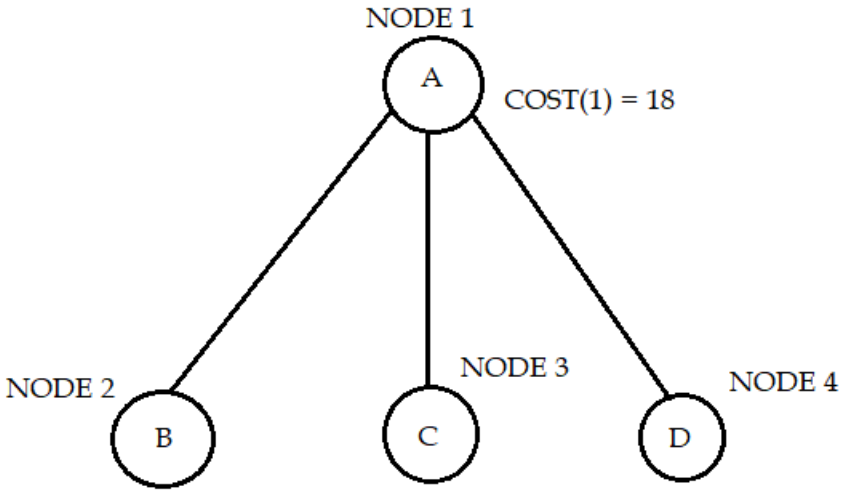
←----------

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

= 4+5+6+2+1

= 18

SO, after step1,



- 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] = ∞.

Resultant cost matrix

Reduced matrix after row and column reduction

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

AFTER ROW AND COLUMN REDUCTION ----------→

	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	**0**
C	0	∞	∞	0
D	3	∞	0	∞

(13)

(5)

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

= 18+13+5+0

= 36

(ii) <u>CHOOSING TO GO TO VERTEX-C: NODE-2 (Path A→C)</u>

- 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

Reduced matrix after row and column reduction

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

AFTER ROW AND COLUMN REDUCTION ----------→

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

D	8	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	∞	∞	∞
D	∞	0	0	∞

AFTER ROW AND COLUMN REDUCTION ----------→

Reduced matrix after row and column reduction

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

(5)

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

= 18+5+3+0

=26

- Thus, we have:

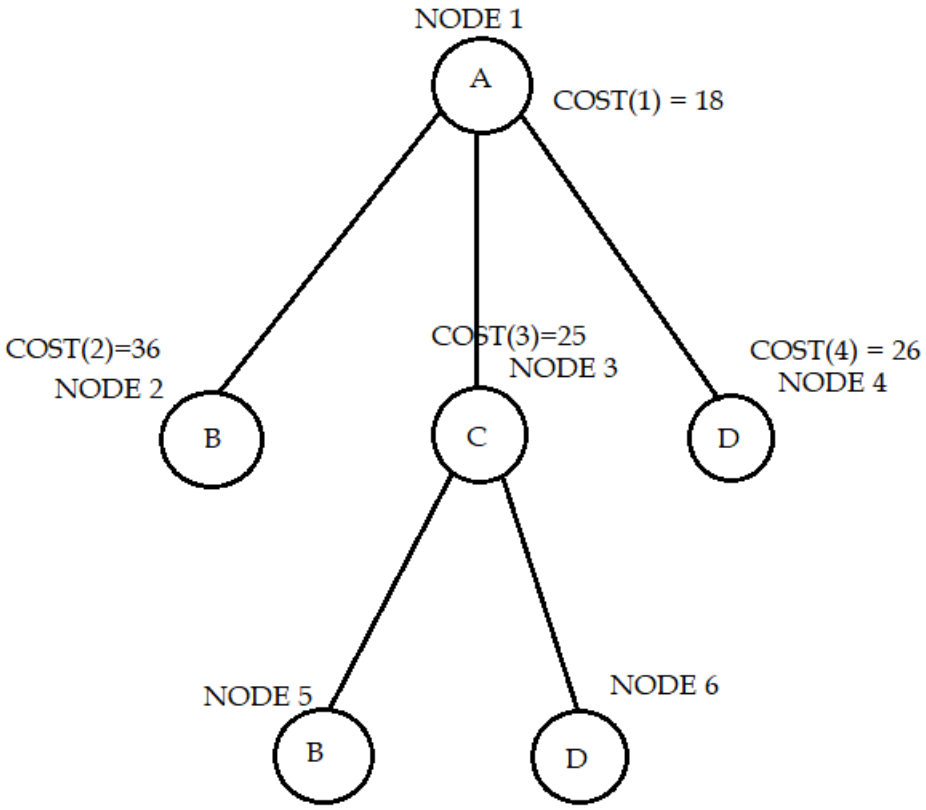
  COST (2) = 35

  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

Reduced matrix after row and column reduction

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

AFTER ROW AND COLUMN REDUCTION ----------→

	A	B	C	D	
A	∞	∞	∞	∞	
B	∞	∞	∞	0	**(**13)
C	∞	∞	∞	∞	
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

Reduced matrix after row and column reduction

	A	B	C	D

	A	B	C	D

A	∞	∞	∞	∞	AFTER ROW AND COLUMN	**A**	∞	∞	∞	∞
B	0	∞	∞	∞	----------→	**B**	0	∞	∞	∞
C	∞	∞	∞	∞	REDUCTION	**C**	∞	∞	∞	∞
D	∞	∞	∞	∞		**D**	∞	∞	∞	∞

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

= 25+0+0

= 25

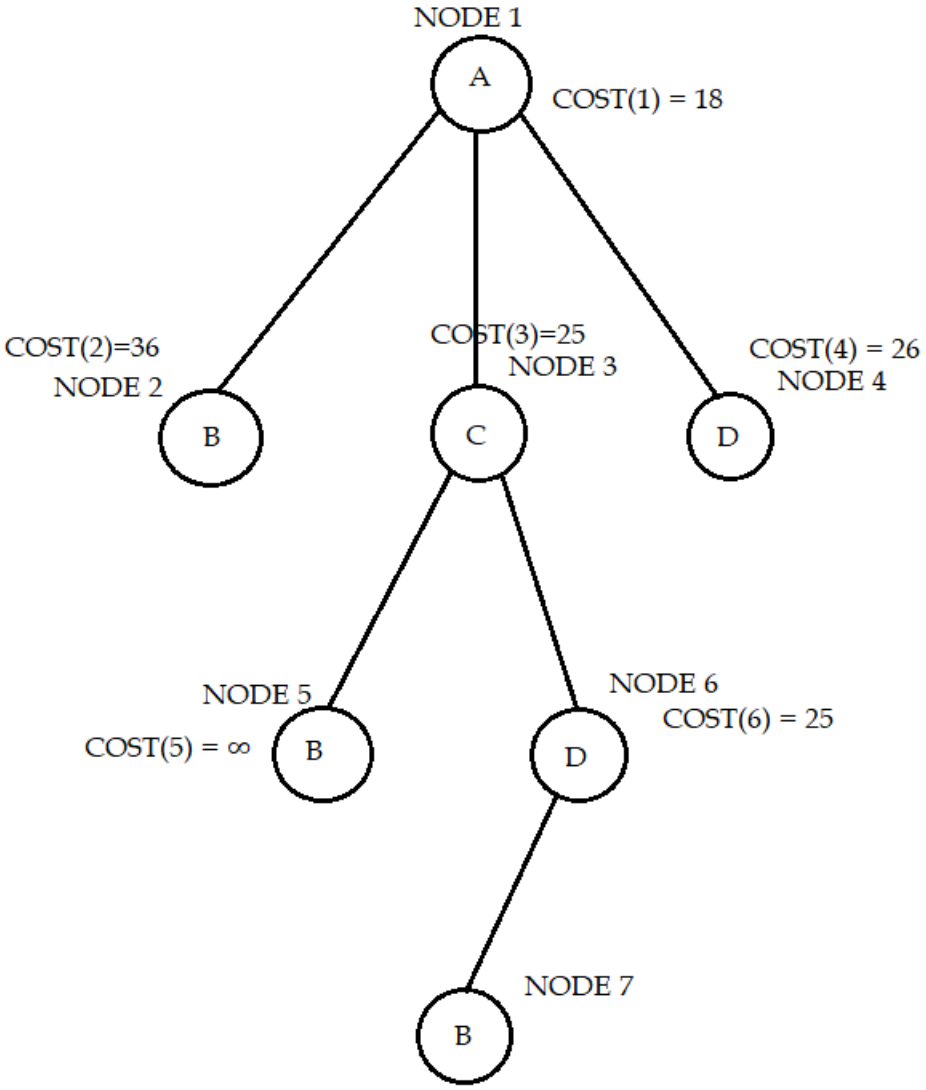
- Thus, we have:

  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	∞	∞	∞	∞
C	∞	∞	∞	∞
D	∞	∞	∞	∞

AFTER ROW AND COLUMN REDUCTION ----------→

Reduced matrix after row and column reduction

	A	B	C	D	
A	∞	∞	∞	∞	
B	∞	∞	∞	∞	**(**13)
C	∞	∞	∞	∞	
D	∞	∞	∞	∞	(8)

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 //////////////////////

############ METHOD - 1 ############

```
#include <iostream>
using namespace std;
const int N = 4;

int final_path[N+1];
bool visited[N];
int final_res = INT_MAX;


   // FUNCTION 1//

void copyToFinal(int curr_path[])
{
   for (int i=0; i<N; i++)
      final_path[i] = curr_path[i];
   final_path[N] = curr_path[0];
}


  // FUNCTION 2 //

int firstMin(int adj[N][N], int i)
```

```
{
   int min = INT_MAX;
   for (int k=0; k<N; k++)
      if (adj[i][k]<min && i != k)
         min = adj[i][k];
   return min;
}


   // FUNCTION 3 //

int secondMin(int adj[N][N], int i)
{
   int first = INT_MAX, second = INT_MAX;
   for (int j=0; j<N; j++)
   {
      if (i == j)
         continue;

      if (adj[i][j] <= first)
      {
         second = first;
         first = adj[i][j];
      }
      else if (adj[i][j] <= second &&
               adj[i][j] != first)
         second = adj[i][j];
```

```
    }
    return second;
}


     // FUNCTION 4 //

void TSPRec(int adj[N][N], int curr_bound, int curr_weight,
            int level, int curr_path[])
{

    if (level==N)
    {

        if (adj[curr_path[level-1]][curr_path[0]] != 0)
        {

            int curr_res = curr_weight +
                    adj[curr_path[level-1]][curr_path[0]];

            if (curr_res < final_res)
            {
                copyToFinal(curr_path);
                final_res = curr_res;
            }
        }
        return;
```

```
}

for (int i=0; i<N; i++)
{

   if (adj[curr_path[level-1]][i] != 0 &&
      visited[i] == false)
   {
      int temp = curr_bound;
      curr_weight += adj[curr_path[level-1]][i];



      if (level==1)
       curr_bound -= ((firstMin(adj, curr_path[level-1]) +
                  firstMin(adj, i))/2);
      else
       curr_bound -= ((secondMin(adj, curr_path[level-1]) +
                  firstMin(adj, i))/2);

      if (curr_bound + curr_weight < final_res)
      {
         curr_path[level] = i;
         visited[i] = true;

         TSPRec(adj, curr_bound, curr_weight, level+1,
               curr_path);
      }
```

```
            curr_weight -= adj[curr_path[level-1]][i];
            curr_bound = temp;
            memset(visited, false, sizeof(visited));
            for (int j=0; j<=level-1; j++)
               visited[curr_path[j]] = true;
         }
      }
   }


    // FUNCTION 5 //

void TSP(int adj[N][N])
{
   int curr_path[N+1];

   int curr_bound = 0;
   memset(curr_path, -1, sizeof(curr_path));
   memset(visited, 0, sizeof(curr_path));


   for (int i=0; i<N; i++)
      curr_bound += (firstMin(adj, i) +
                  secondMin(adj, i));
```

```
    curr_bound = (curr_bound&1)? curr_bound/2 + 1 :
                          curr_bound/2;

    visited[0] = true;
    curr_path[0] = 0;


    TSPRec(adj, curr_bound, 0, 1, curr_path);
}



    // MAIN FUNCTION //

int main()
{
    int adj[N][N] = { {0, 10, 15, 20},
        {10, 0, 35, 25},
        {15, 35, 0, 30},
        {20, 25, 30, 0}
    };

    TSP(adj);

    printf("Minimum cost : %d\n", final_res);
    printf("Path Taken : ");
    for (int i=0; i<=N; i++)
```

```
        printf("%d ", final_path[i]);

    return 0;
}
```

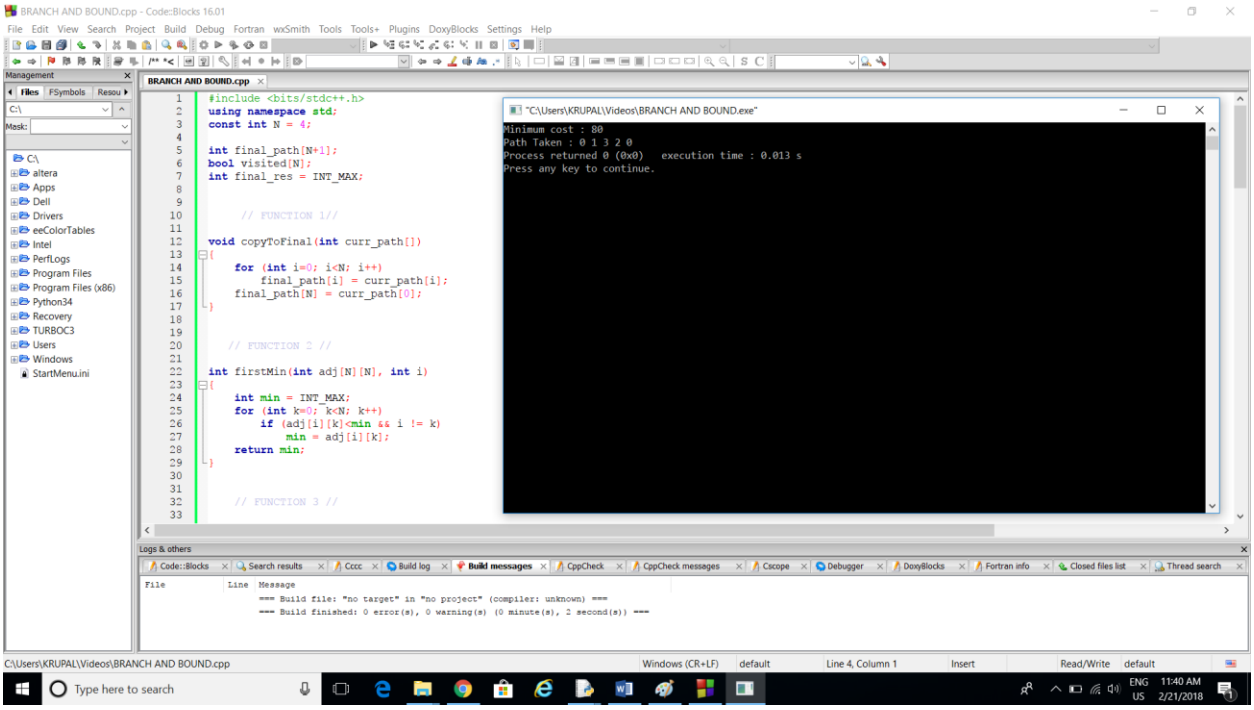
OUTPUT :

Minimum cost : 80

Path Taken : 0 1 3 2 0

Process returned 0 (0x0) execution time : 0.013 s

Press any key to continue.



############ METHOD - 2 ############

```
#include <bits/stdc++.h>
using namespace std;

// N is number of total nodes on the graph or the cities in the map
#define N 5

// Sentinal value for representing infinity
#define INF INT_MAX

// 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> > 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> > 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.
                free(min);
        }
}

// main function
int main()
{
        // cost matrix for traveling salesman problem.
        /*
        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}
};
*/
// cost 34
int costMatrix[N][N] =
{
        { INF, 10,  8,   9,   7 },
        { 10,  INF, 10,  5,   6 },
        { 8,   10,  INF, 8,   9 },
        { 9,   5,   8,   INF, 6 },
        { 7,   6,   9,   6,   INF }
};

/*
// cost 16
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}
```

```
};
*/

/*
// cost 8
int costMatrix[N][N] =
{
        {INF, 2,   1,   INF},
        {2,   INF, 4,   3},
        {1,   4,   INF, 2},
        {INF, 3,   2,   INF}
};
*/
/*
// cost 12
int costMatrix[N][N] =
{
        {INF, 5,   4,   3},
        {3,   INF, 8,   2},
        {5,   3,   INF, 9},
        {6,   4,   3,   INF}
};
*/

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

return 0;
```

}

OUTPUT:

1 -> 3

3 -> 4

4 -> 2

2 -> 5

5 -> 1

Total Cost is 34

Process returned 0 (0x0) execution time : 0.749 s

Press any key to continue.