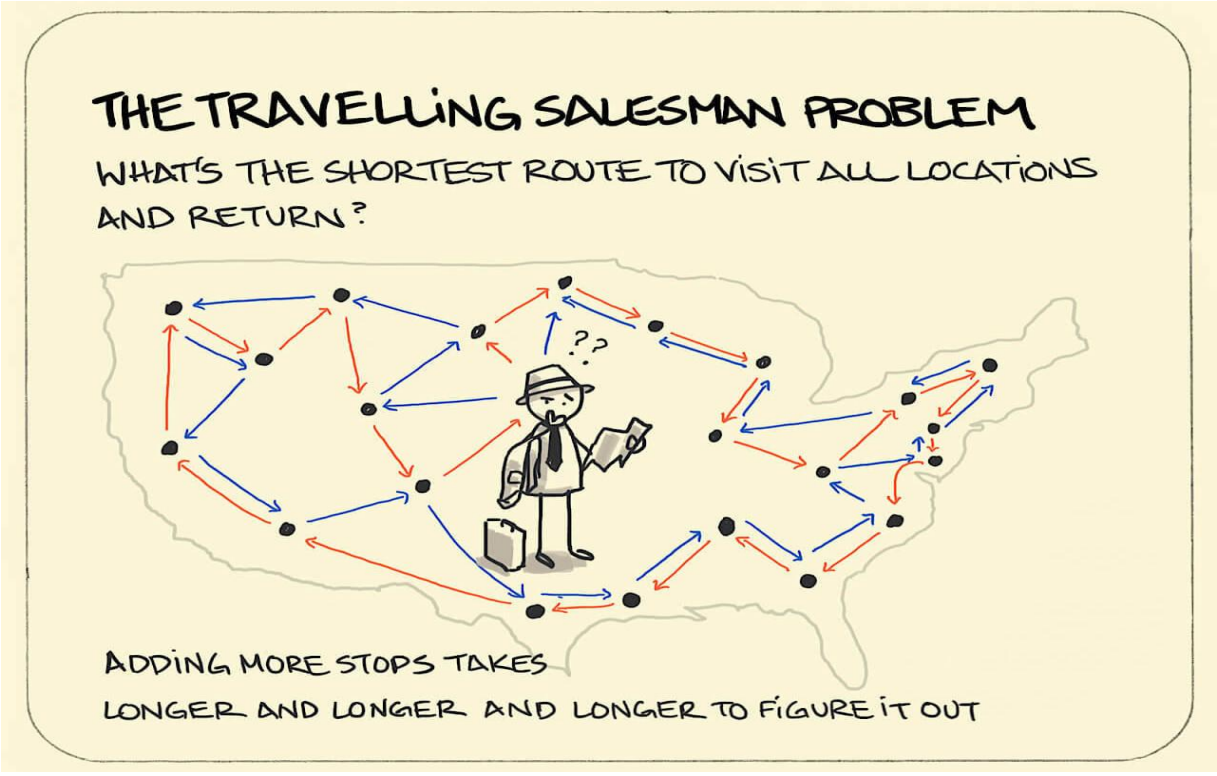
LNMIIT
The LNM Institute of
Information Technology

ARTIFICIAL INTELLIGENCE

TRAVELLING SALESMAN PROBLEM

Team Members:

Shubham Sharma(19ucs038)
Yash Jain(19ucs197)
Garvit Vijayvargia(19ucs201)
Tanush Garg(19ucs203)
Ashutosh Kumar(19ucc082)



INTRODUCTION :

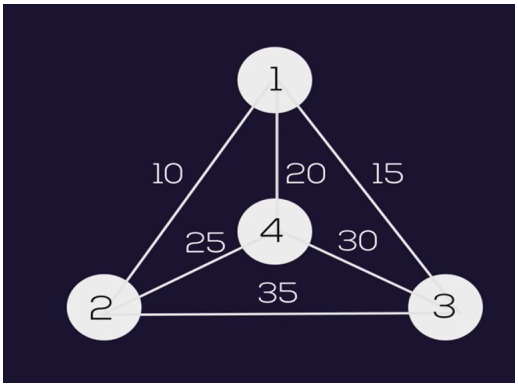
The traveling salesman problem (TSP) is an algorithm problem tasked with finding the shortest route between a set of points and locations that must be visited. In the problem statement, the points are the cities a salesperson might visit. The salesman‘s goal is to keep both the travel costs and the distance traveled as low as possible.
The traveling salesman problem (TSP) is a NP Hard problem in combinatorial optimization and has several applications, such as vehicle routing problems, logistics, planning and scheduling.
For example, delivery service companies that every day face problems, it occurs because the route followed to get to the destination is not optimal. Examples of cases in **delivery service**, a courier needs to deliver goods to some customers with different destinations. Therefore, time is of considerable concern in the delivery of goods; it relates to the reputation of the company. To reach these goals required a system capable of providing an optimal travel route so that the travel time to be minimum. So it needs an application that can optimize the TSP solution.

PROBLEM STATEMENT:

Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

A traveling salesman is getting ready for a big sales tour. Starting at his hometown, suitcase in hand, he will conduct a journey in which each of his target cities is visited exactly once before he returns home. Given the pairwise distances between cities.

Denote the cities by 1,2, . . . ,n-1, n, the salesman’s hometown being 1, and let D = (d_{ij}) be the matrix of intercity distances. The goal is to design a tour that starts and ends at 1, includes all other cities exactly once, and has a minimum total length. Even in this tiny example where total no of cities are 4, it is tricky for a human to find the solution.



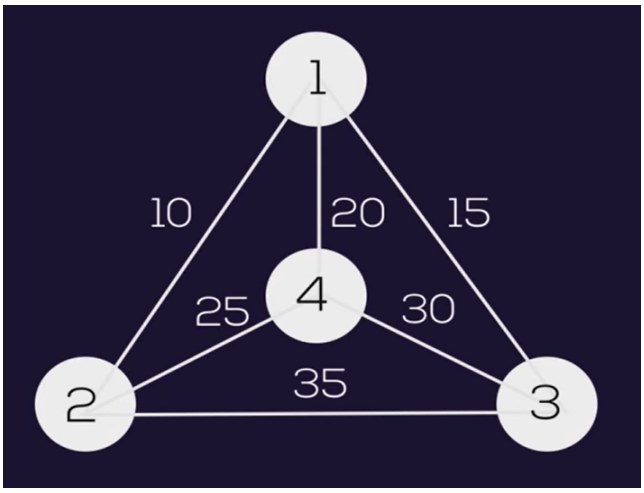
So it is hard to imagine what happens when hundreds of cities are involved.

Naive Approach:

We have to generate all permutations taking every city one by one as the starting point and flag the minimum of them and that will be our solution.

STEPS:

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



Permutations	Corresponding cost
1-2-3-4-1	95
1-2-4-3-1	80
1-3-2-4-1	95

Permutations	Corresponding cost
1-3-4-2-1	80
1-4-3-2-1	95
1-4-2-3-1	95

Here we can take 2 paths 1-2-4-3-1 or 1-3-4-2-1 with a minimum cost of 80.

Time Complexity: O(n!)

It is impractical as the number of permutations is (n−1)! .
Which means that in terms of time complexity, we will be looking at O(n!) which is highly undesirable.

DP Approach:

TSP can also be solved by using Dynamic programming by having a recursive relation in terms of sub-problems and even though this method is slightly better in terms of time complexity than brute force, its time complexity is still exponential in nature.

STEPS:
Suppose we have started at city 1 as required, have visited a few cities, and are now in city j.

S: Subset of graph which is not yet traversed.
dist(i,1): distance from i to 1.

When |S| > 1, we define C(S, 1) = ∞ since the path cannot both start and end at 1.

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.

However, its time complexity would exponentially increase with the number of cities. The time complexity with the DP method asymptotically equals N^2 × 2^N where N is the number of cities.

TRIANGLE INEQUALITY:
The triangle inequality states that the sum of the lengths of any two sides of a triangle is greater than the length of the remaining side.
When the cost function satisfies the triangle inequality, we can make sure that the approximate algorithm for TSP will return a tour whose cost is never more than twice the cost of an optimal tour.

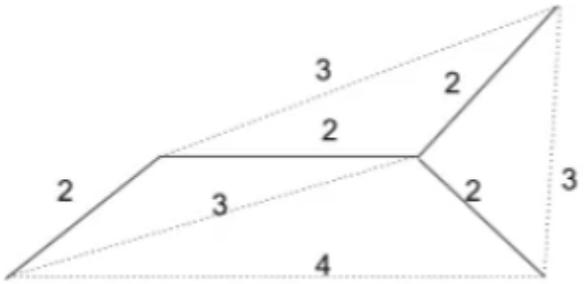
MINIMUM SPANNING TREE:
Given an undirected graph G=(V, E) with arc lengths d(ij),a spanning tree is a subgraph of G such that it is a tree and it touches all the nodes of the graph G. A minimum spanning tree does not contain any cycles which means that for n nodes it contains n-1 arcs.The minimum spanning tree problem is to find a spanning tree with the minimum possible total edge weight.

SOLUTION TO PROBLEM:

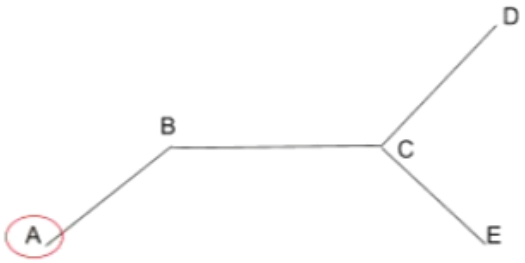
Topic a): Study and solve it using minimum-spanning-tree (MST) heuristic.
A minimum spanning tree(MST)is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. That is, it is a spanning tree whose sum of edge weights is as small as possible.
There is no polynomial time solution to this problem as it is a NP Hard problem. So we will be using an approximate algorithm to solve this problem**.**

STEPS:

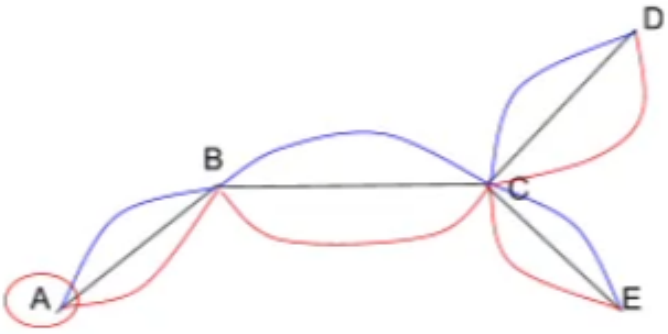
STEP 1: Construct a minimum spanning tree.



STEP 2: Let the root node be an arbitrary vertex.

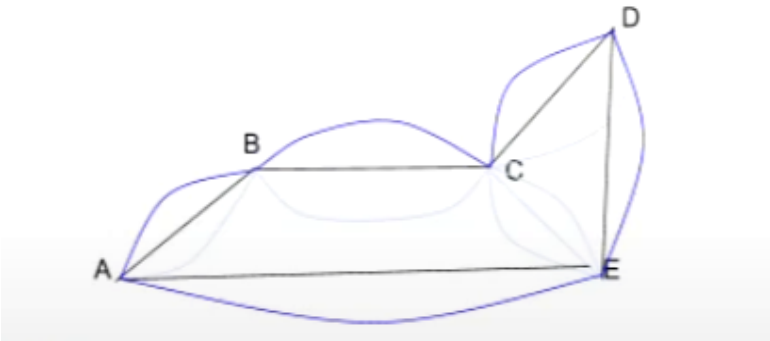


STEP 3: Traverse all the vertices by DFS(Depth First Search),record sequence of vertices (both visited and unvisited).



The Sequence is A-B-C-D-C-E-C-B-A

STEP 4: Use a shortcut strategy to generate a feasible tour A-B-C-D-(C)-E-(C-B)-A



The MST Tour is A-B-C-D-E-A

We can use Kruskal's Algorithm or Prim's Algorithm to construct the MST.

Topic b): Find and implement an efficient algorithm in the literature for constructing the MST, and use it with A* graph search to solve TSP.

There are two algorithms that are used to construct an Minimum spanning Tree(MST)

- Kruskal's algorithm
- Prim's algorithm

Both algorithms are greedy algorithms. They elaborate the generic algorithm because they uses a specific rule to determine a safe edge in line-3 of Generic-MST Algorithm

```
Generic-MST(G,w)
1    A = ∅
2    while A does not form a spanning tree
3        Find an Edge (u,v) that is safe for A
4        A = A U {(u,v)}
5    return A
```

In Kruskal's Algorithm :

- The set A is a **Forest**
- The safe Edge added to A is always a least-weight edge in the graph that connects two distinct components.

In Prim's Algorithm

- The set A forms a Single **Tree**.
- The safe edge added to A is always a least-weight edge connecting the tree to a vertex not in the tree.

A **Tree** is a connected , acyclic , undirected graph.
A **Forest** is a set of trees(non necessarily connected)

Kruskal's Algorithm –

Kruskal's algorithm is a minimum-spanning-tree algorithm which finds an edge of the least possible weight that connects any two trees in the forest. It is a greedy algorithm in graph theory as it finds a minimum spanning tree for a connected weighted graph adding non decreasing cost edges at each step. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

Steps:
1 Initialize an Empty set A
2 Create V Trees, One containing each Vertex
3 Sort the edges of the graph G(V,E) in non-decreasing order by weight
4 For each edge taken in non-decreasing order by weight,
check whether the end points of the edge (u,v) belong to the same tree.
If yes, the edge is discarded , else merge the Two Vertices into a single set
And add the edge to A

Disjoint-sets play a key role in Kruskal's algorithm for finding the minimum spanning tree of a graph

- A disjoint-set data structure is a data structure that tracks a set of elements partitioned into a number of disjoint (non-overlapping) subsets.
- It provides near-constant-time operations to add new sets, to merge existing sets,and to determine whether elements are in the same set
- A Union-Find data Structure performs three useful operations :
  - **DSU(v):** Makes a new set by creating a new element with a unique id, and initializing the parent to itself. The DSU operation has O(1) time complexity, so initializing n sets has O(n) time complexity
  - **Find-Set(x):** Follows the chain of parent pointers from x till the root element, whose parent is itself. Returns the root element. Time Complexity: O(logn)
  - **Union(x,y):** Merges x and y into the same partition by attaching the root of one to the root of the other. If this is done naively, such as by always making x a child of y, the height of the trees can grow as O(n). To prevent this *union by rank or union by size* is used in Union operation.

Algorithm :-

```
MST-Kruskal(G,w)
1  A = ø
2  for each vertex v ∈ G.V
3     DSU(v)
4  sort the edges of G.E into non-decreasing order by weight (w)
5  for each edge (u,v) ∈ G.E , taken in non-decreasing order by weight
6     if FIND-SET(u) ≠ FIND-SET(v)
7        A = A U {(u,v)}
8        UNION(u,v)
9  return A
```

Time Complexity Analysis :-

The set A in **line-1** takes O(1) time,
DSU(v) : makes set takes O(V) time **(line 2-3**)
Time to sort the edges in **line 4** is O(E * logE)
For loop in **line 5-8** performs O(E) FIND-SET and UNION operations in the disjoint - set forest.
Observing that $|E| < |V|^2$, we have $\log|E| = O(\log V)$, and so we can restate the running time of Kruskal's algorithm as O(E * logV).

Input :-

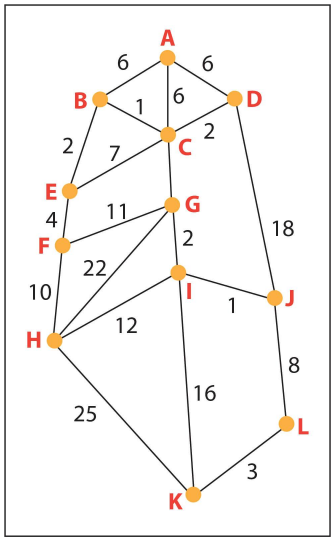
Adjacency-Matrix Representation of the Graph G(V,E)

Vertice	A	B	C	D	E	F	G	H	I	J	K	L
A	0	6	6	6	∞	∞	∞	∞	∞	∞	∞	∞
B	6	0	1	∞	2	∞	∞	∞	∞	∞	∞	∞
C	6	1	0	2	7	∞	2	∞	∞	∞	∞	∞
D	6	∞	2	0	∞	∞	∞	∞	∞	18	∞	∞
E	∞	2	7	∞	0	4	∞	∞	∞	∞	∞	∞
F	∞	∞	∞	∞	4	0	11	10	∞	∞	∞	∞
G	∞	∞	2	∞	∞	11	0	22	2	∞	∞	∞
H	∞	∞	∞	∞	∞	10	22	0	12	∞	25	∞
I	∞	∞	∞	∞	∞	∞	2	12	0	1	16	∞
J	∞	∞	∞	18	∞	∞	∞	∞	1	0	∞	8
K	∞	∞	∞	∞	∞	∞	∞	25	16	∞	0	3
L	∞	∞	∞	∞	∞	∞	∞	∞	∞	8	3	0

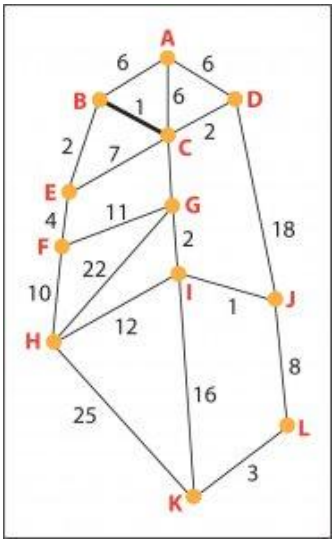
Software Input

```
Vertices : 12
edges : 20
0 1 6
0 2 6
0 3 6
1 2 1
1 4 2
2 3 2
2 4 7
2 6 2
3 9 18
4 5 4
5 6 11
5 7 10
6 7 22
6 8 2
7 8 12
7 10 25
8 9 1
8 10 16
9 11 8
10 11 3
```

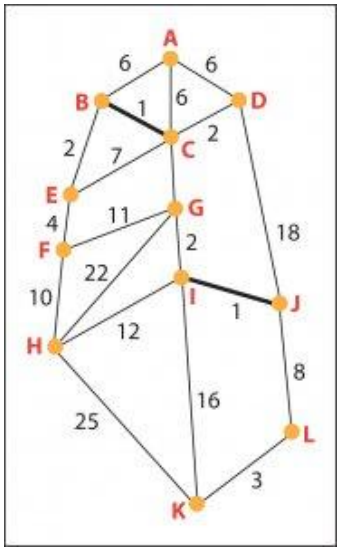
1.

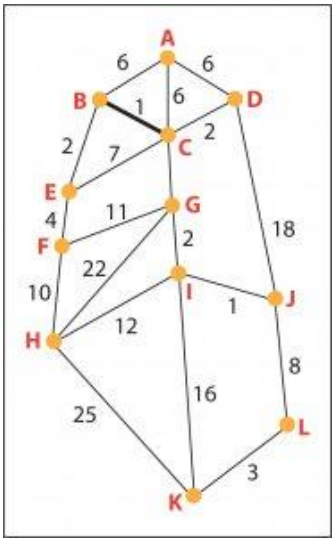


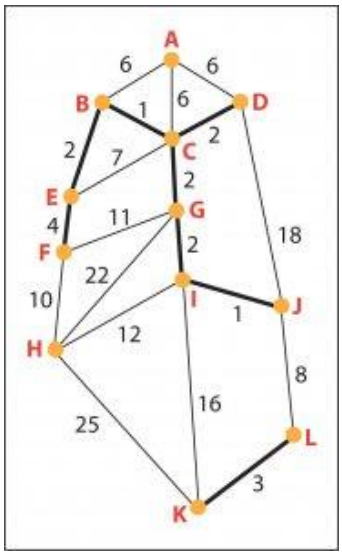
2.

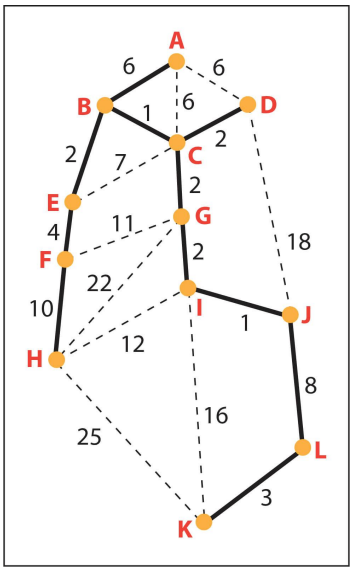


3.



4 

5. 

6. 

Output:

```
Weight of Minimum Spanning Tree : 41
```

 # NOTE: Code for Kruskal's Algorithm is in the Zip File itself!!

PRIM'S ALGORITHM-

This algorithm was originally discovered by the Czech mathematician Vojtěch Jarník in 1930. However this algorithm is mostly known as Prim's algorithm after the American mathematician Robert Clay Prim, who rediscovered and republished it in 1957. Additionally Edsger Dijkstra published this algorithm in 1959

The minimum spanning tree is built gradually by adding edges one at a time. At first the spanning tree consists only of a single vertex (chosen arbitrarily). Then the minimum weight edge outgoing from this vertex is selected and added to the spanning tree. After that the spanning tree already consists of two vertices. Now select and add the edge with the minimum weight that has one end in an already selected vertex (i.e. a vertex that is already part of the spanning tree), and the other end in an unselected vertex. And so on, i.e. every time we select and add the edge with minimal weight that connects one selected vertex with one unselected vertex. The process is repeated until the spanning tree contains all vertices (or equivalently until we have V-1 edges)

This algorithm is Greedy algorithm because the tree is augmented at each step with an edge that contributes the minimum amount possible to the tree's weight.

Algorithm :-

```
MST-PRIME(G,w,r)

1     for each u ∈ G.V
2           u.key = ∞
3           u.π = NIL
4     r.key = 0
5     Q = G.V
6     while Q ≠ ∅
7           u = EXTRACT-MIN(Q)
8           for each v ∈ G.Adj[u]
9                 if v ∈ Q and w(u,v) < v.key
10                      v.π = u
11                      v.key = w(u,v)
```

Explanation :-

Lines 1-5 set the key of each vertex to ∞ (except for the **root r** whose **key is set to 0** so that it will be the first vertex processed).

Set the parent of each vertex to **NIL** , and initialize the min-priority queue to **Q** to contain all the vertices.

- v.key is the minimum weight of any edge connecting (v) to a vertex in the tree
- v.π is the names of parent of v in the tree

Prior to each iteration of the while loop of line 6-11

1. A = {(v,u,π) : v ∈ V-{r}-{Q}
2. The vertices already placed into the minimum spanning tree are those in V-Q.
3. For all vertices v ∈ Q, if v.π != NIL , then v.key < infinity and v.key is MST-PRIME(G,w,r) the weight of a light edge(v,v.π) connecting v to some vertex already placed into the minimum spanning tree.

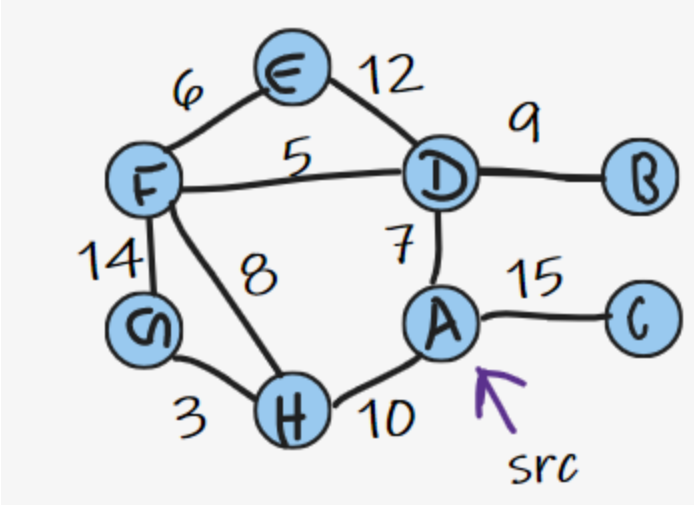
The for loop of lines 8-11 updates the key and π fields of every vertex (v) adjacent to the (u)but not in the tree. The updating maintains the third part of the loop invariant.

Time complexity analysis :

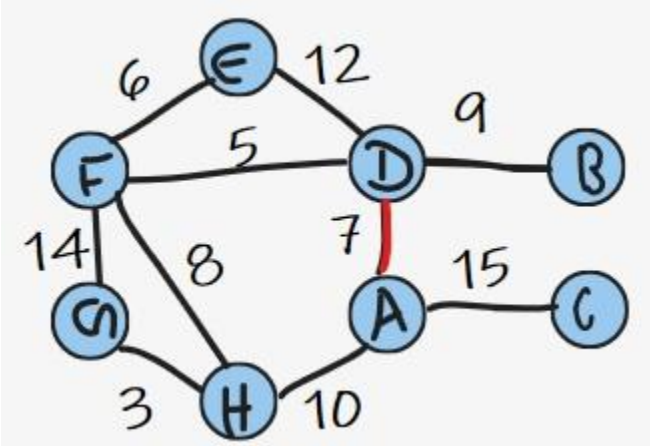
The running time of Prim's algorithm depends on how Min-priority queue Q is implemented as a **binary min-Heap**

- The **BUILD-MIN-HEAP** procedure to performs lines 1-5 in O(V) time
- While loop executes |V| times, and each EXTRACT-MIN operation takes O(logV) time, the total time is O(V*logV)
- The for loop in lines 8-11 executes O(E) times, and line 11 involves in O(logV) time, the total time is O(E*logV)

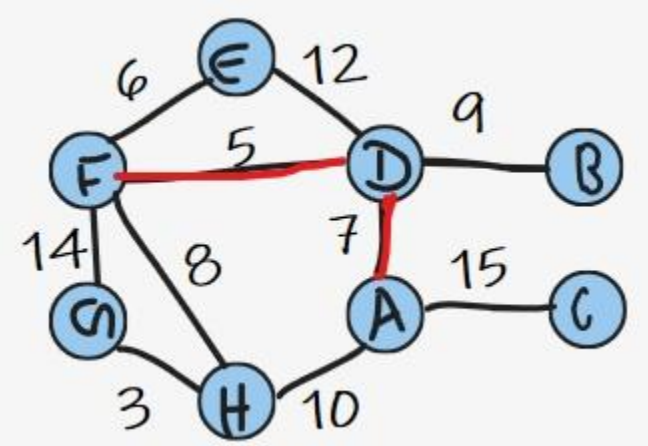
- The total time is O(VlogV + ElogV) = O(ElogV), which is asymptotically the same as for Kruskal's algorithm.



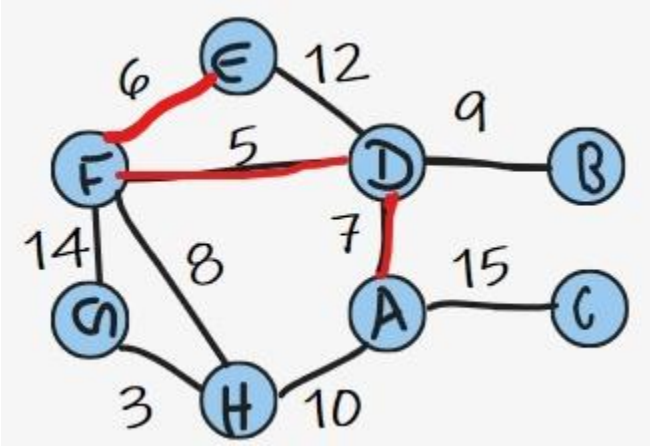
Graph G(V , E)



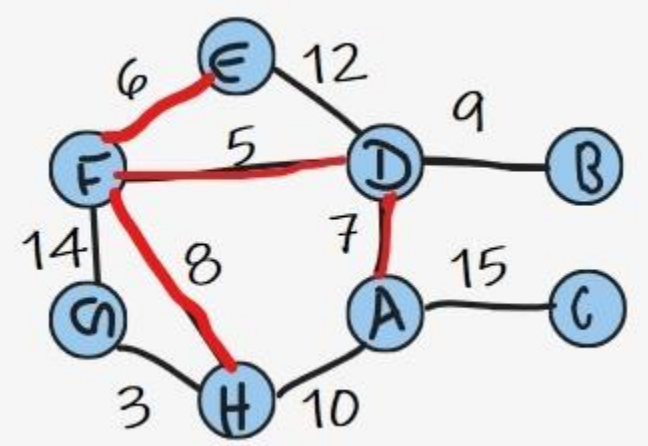
MST edges : (D,A)



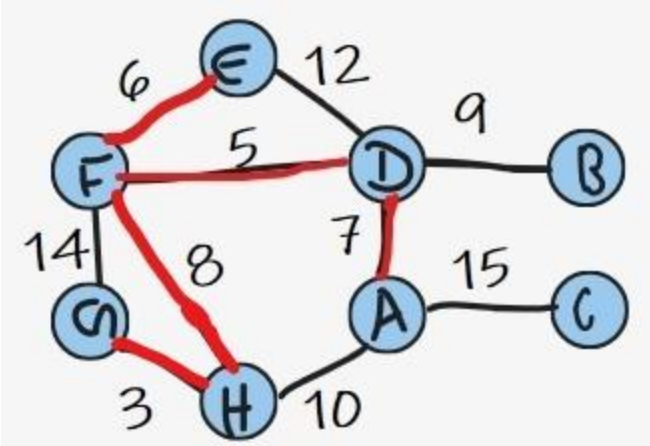
MST edges : (D,A) , (D,F)



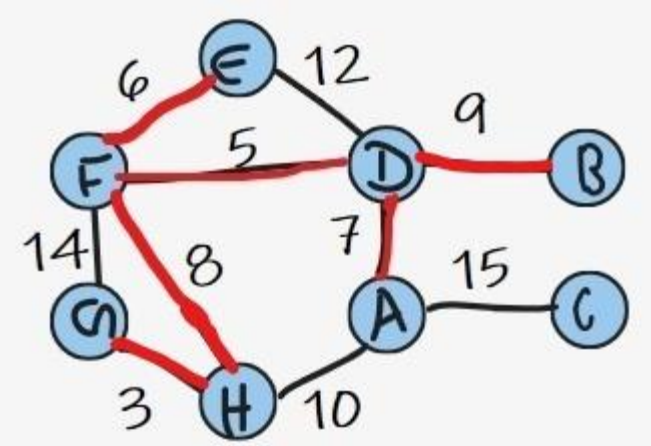
MST edges : (D,A) , (D,F), (F,E)



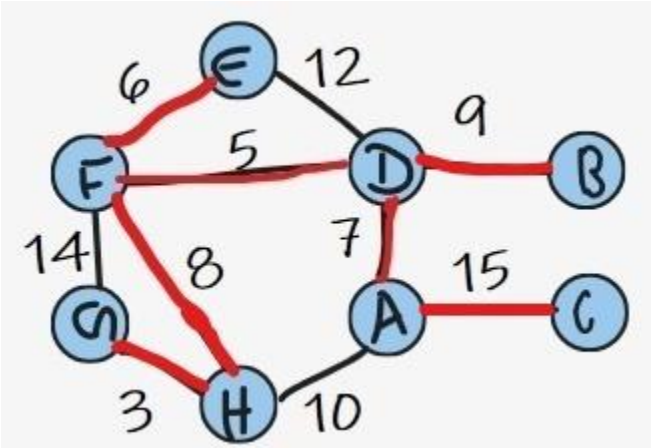
MST edges : (D,A) , (D,F) , (F,E), (F,H)



MST edges : (D,A), (D,F) , (F,E), (F,H),(G,H)



(D,A),(D,F),(F,E), (F,H),(G,H),(D,B)



(D,A),(D,F),(F,E), (F,H),(G,H),(D,B),(A,C)
Weight of MST = 53

Input:-

```
Number of Vertices :8
Number of Edges : 10
Input 10 Edges below :
u :1
v :3
weight of edge :15
```

Similarly put all values of u,v and weight of edge for all edges.
SRC node is 1.

Output:-

```
Weight of Minimum Spanning Tree : 53
```

 # NOTE: Code for Prim's Algorithm is in the Zip File itself!!

Now that we have made the MST, we need to figure out how to get the solution of TSP from it. One method is to apply DFS or Preorder traversal on the obtained MST, note down the nodes that we obtain from DFS and append the source node at the end. This will give us the path that should be taken to solve TSP. However, through this method, we only get the path and

not the minimum weight path. This method is simply an estimation for solving this problem and as such, does not give us the correct output in every case. To get better results we can make use of the A* algorithm as stated in the next section.

USING MST WITH A* GRAPH TO SOLVE TSP PROBLEM:

A* Search algorithm is one of the best and popular techniques used in path-finding and graph traversals.

A* Search algorithms, unlike other traversal techniques, have “brains”. What it means is that it is really a smart algorithm which separates it from the other conventional algorithms

Steps:

1. Choose a vertex to be the starting and ending point for the salesman.
2. Construct a minimum spanning tree for the given graph.
3. Traverse all the vertices by a tree traversal algo(A* in our problem) of your choice,record the sequence of vertices(both visited and unvisited) and in the end, add the starting vertex as well.

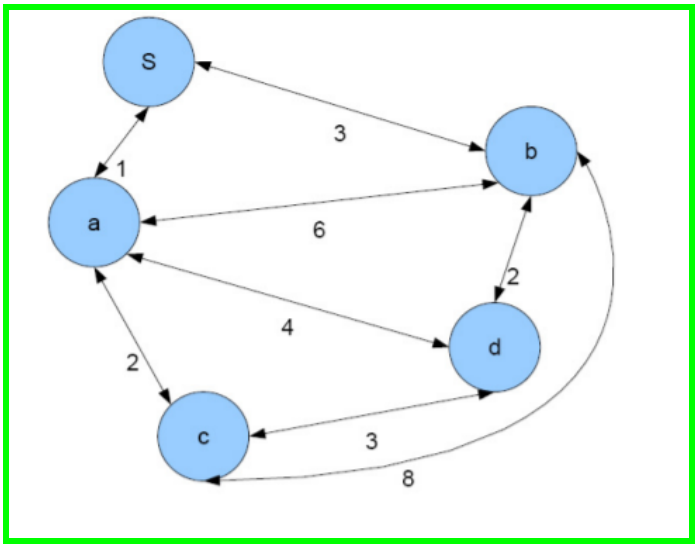
Admissible heuristic: When for each node n in the graph, h(n) never overestimates the cost of reaching the goal.

With respect to A*:

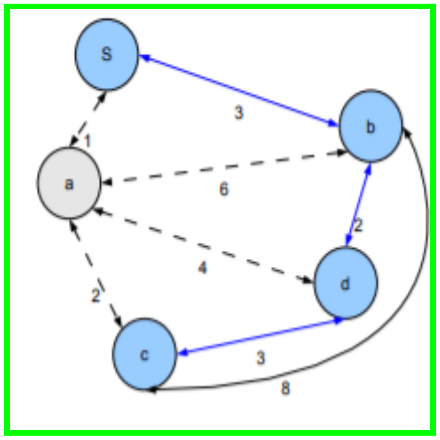
- A* using an admissible heuristic guarantees to find the shortest path from the start to the goal.
- A* Using a consistent heuristic, in addition to finding the shortest path, also guarantees that once a node is explored we have already found the shortest path to this node, and therefore no node needs to be re-explored.

Time Complexity:

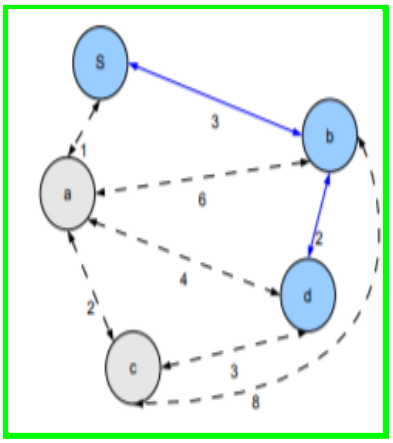
The time complexity of the solution of travelling salesman problem with help of MST heuristic using A* Algorithm is O(E*(V^2))
Complexity of graph matrix method = V^2
Complexity of prim’s algorithm = E*(V^2)
Complexity of A* algorithm = E*(V^2)
E*(V^2) + V^2 + E*log(V) = O(V^2)



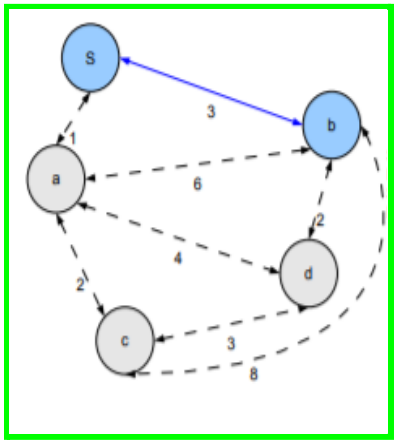
A TSP instance



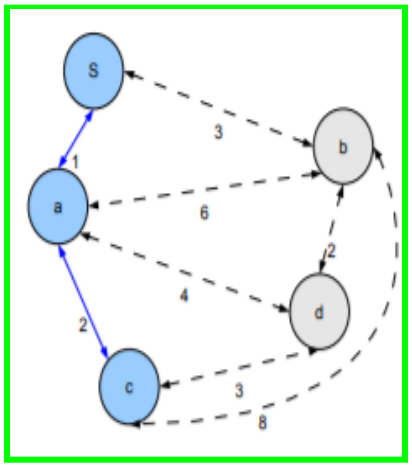
1.MST edges, after (s, a) is added into TSP path



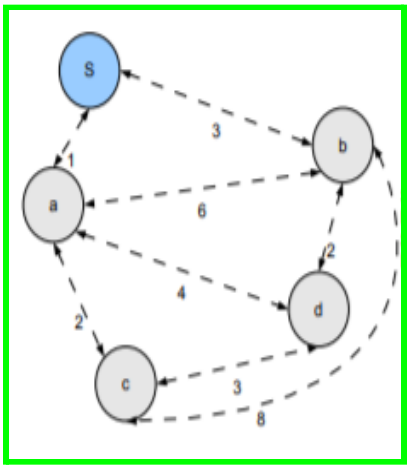
2.MST edges after (s,a), (a,c) is added into TSP path



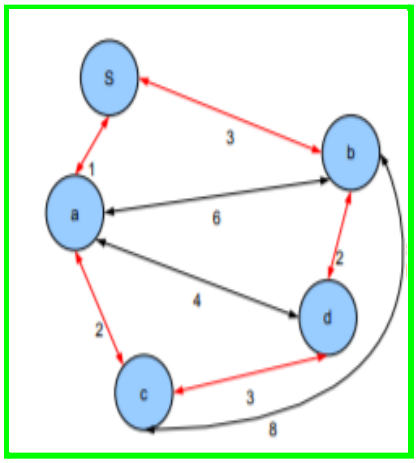
3.MST edges, after (s,a), (a,c),(c,d) is added into TSP path



4.Try another path



5.Return to Previous Choice



6.A TSP solution is found

Input:
Here we have taken 5 cities namely with their coordinates:
A(5,0), B(6,5), C(7,1), D(1,7), E(9,5)

```
5
A 5 0
B 6 5
C 7 1
D 1 7
E 9 5
--
```

Output:

```
The optimum Cost is 22 with path ACEBDA
the number of nodes expanded are 11 total nodes are 25
```

The path with optimal cost is ACEBDA.

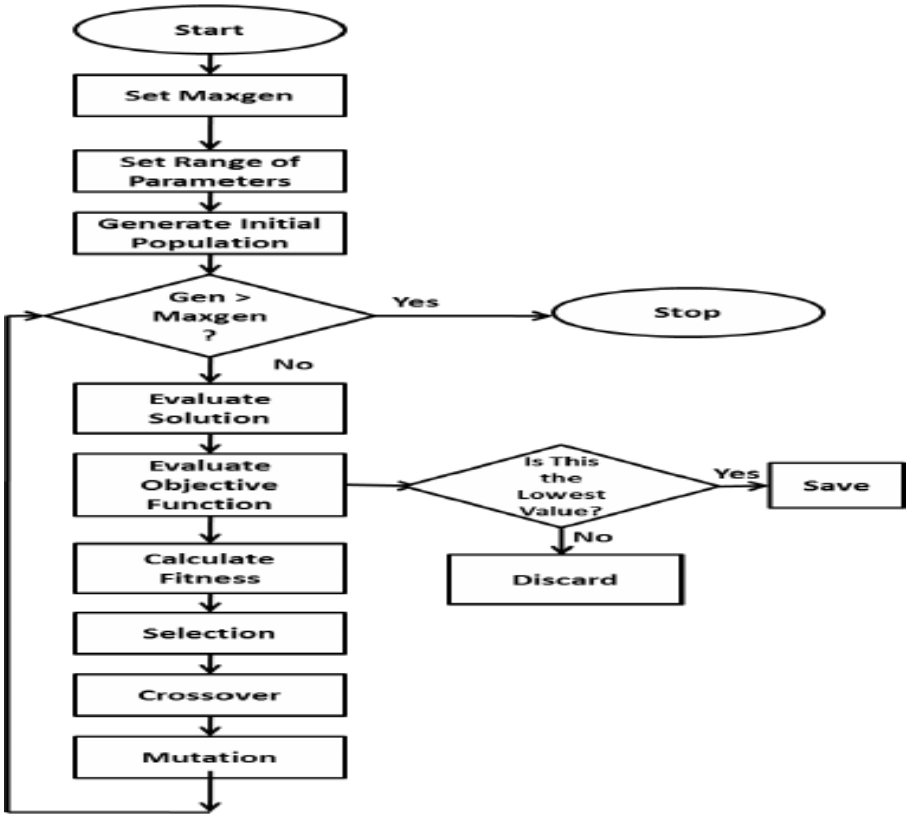
 # NOTE: Code for A* Heuristic Algorithm is in the Zip File itself!!

Topic c): Study and report current state of the art methods which show (significant) Improvement over MST heuristic.
Solution c): MST is itself very popular and efficient method to solve the problem of TSP.There are some other methods:

- Genetic Algorithm
- Ant Colony Optimization
- Particle Swarm Optimization
- Simulated Annealing
- K-opt heuristic

Now , explaining each part one by one-

•Genetic Algorithm (GA):
A genetic algorithm is a search heuristic that is inspired by Charles Darwin's theory of natural evolution. This algorithm reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation.
Genetic programming is an automated method for solving problems. Specifically, genetic programming progressively breeds a population of computer programs over a series of generations. Genetic programming is a probabilistic algorithm that searches the space of compositions of the available functions and terminals under the guidance of a fitness measure. Genetic programming starts with a primordial ooze of thousands of randomly created computer programs and uses the Darwinian principle of natural selection, recombination (crossover), mutation, gene duplication, gene deletion, and certain mechanisms of developmental biology to breed an improved population over a series of many generations.

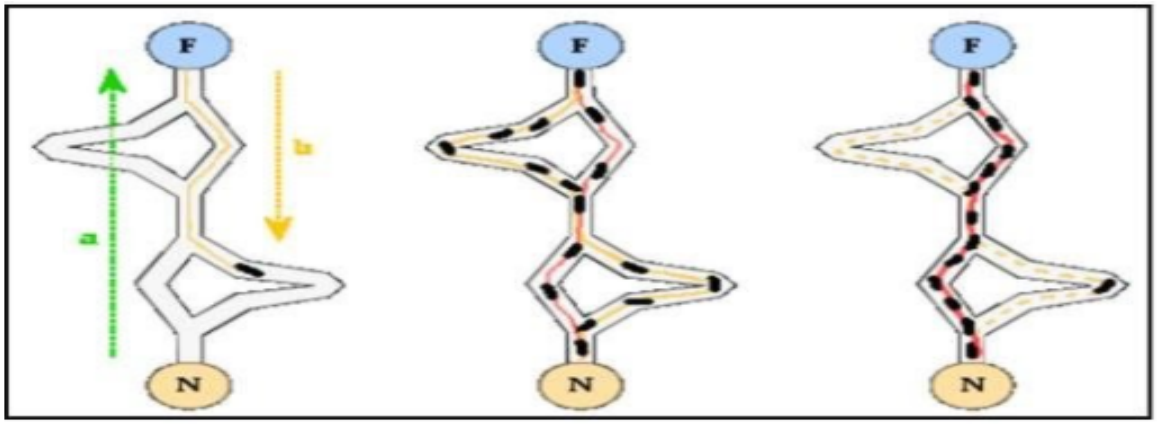


Flowchart of the parameter optimization procedure using GA.

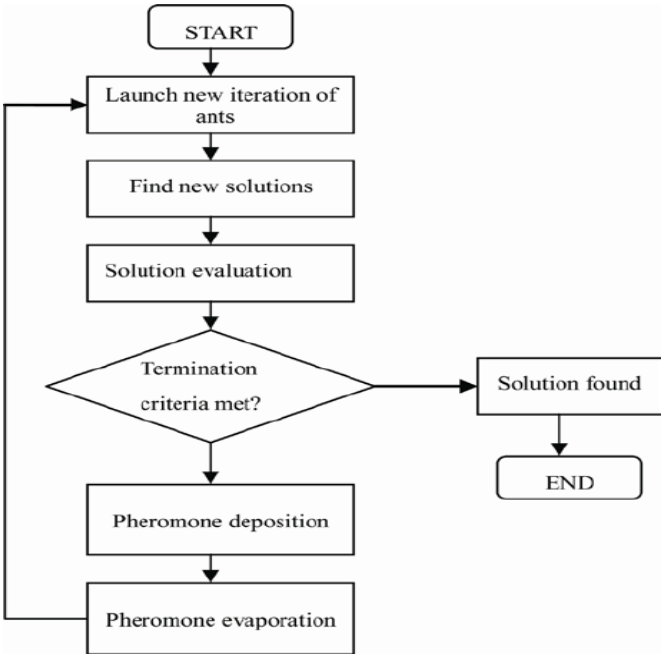
•Ant Colony Optimization(ANO)

In the ant colony optimization algorithms, an artificial ant is a simple computational agent that searches for good solutions to a given optimization problem. To apply an ant colony algorithm, the optimization problem needs to be converted into the problem of finding the shortest path on a weighted graph. In the first step of each iteration, each ant stochastically constructs a solution, i.e. the order in which the edges in the graph should be followed. In the second step, the paths found by the different ants are compared. The last step consists of updating the pheromone levels on each edge.

Ant colony optimization (ACO) is one of the most popular meta-heuristics used for combinatorial optimization (CO) in which an optimal solution is sought over a discrete search space. The well-known CO's example is the traveling salesman problem (TSP) where the search-space of candidate solutions grows more than exponentially as the size of the problem increases, which makes the search for optimal solution infeasible.



Ants use pheromone as indirect communication to build best tour



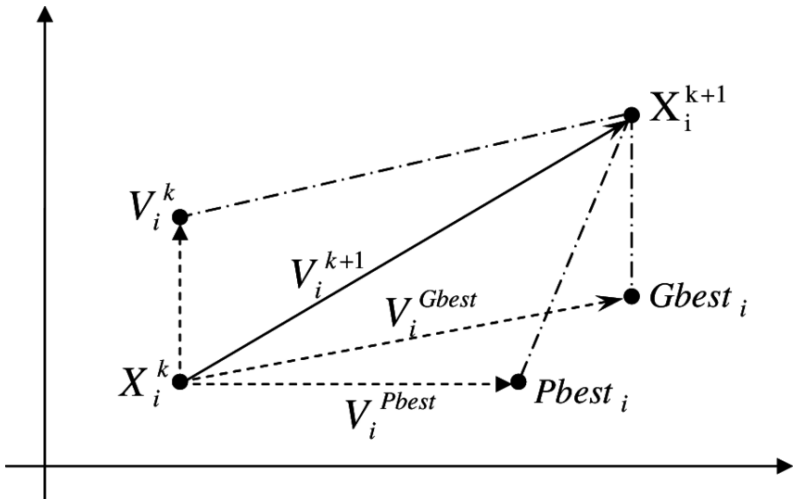
The optimization flowchart of ACO

When choosing their way, they tend to choose, in probability, paths marked by strong pheromone concentrations. As soon as an ant finds a food source, it evaluates the quantity and the quality of the food and carries some of it back to the nest. During the return trip, the quantity of pheromone that an ant leaves on the ground may depend on the quantity and quality of the food. The pheromone trails will guide other ants to the food source. It has been shown that the indirect communication between the ants via pheromone trails enables them to find shortest paths between their nest and food sources.

•Particle Swarm Optimization

Particle swarm optimization (PSO) is one of the bio-inspired algorithms and it is a simple one to search for an optimal solution in the solution space. It is different from other optimization algorithms in such a way that only the objective function is needed and it is not dependent on the gradient or any differential form of the objective. It also has very few hyperparameters.

PSO is a population-based optimization method . Some of the attractive features of PSO include the ease of implementation and the fact that no gradient information is required. PSO technique conducts search using a population of particles, corresponding to individuals. Each particle represents a candidate solution to the problem at hand. In a PSO system, particles change their positions by flying around in a multidimensional search space until computational limitations are exceeded.



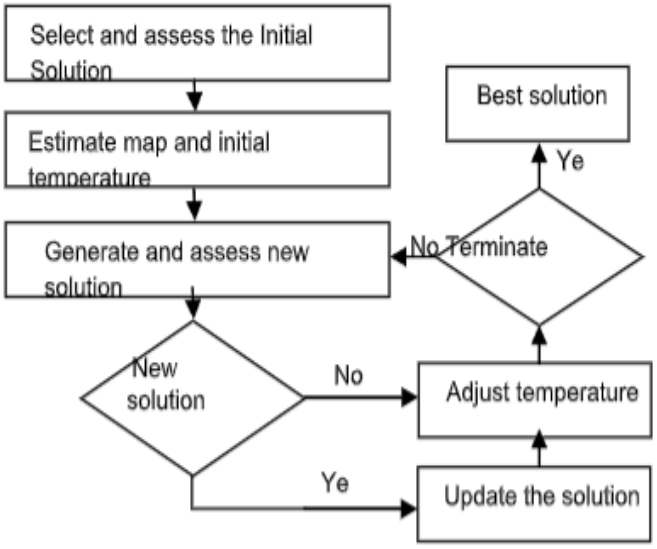
Concept of modification of a searching point by PSO

In PSO, the population dynamics simulates a “bird flock‟ behavior, where social sharing of information takes place and individuals can profit from the discoveries and previous experience of all the other companions during the search for food. Thus, each companion, called particle, in the population, which is called swarm, is assumed to fly over the search space to find promising regions of the landscape. In this context, each particle is treated as a point in a d-dimensional space, which adjusts its own flying according to its flying experience as well as the flying experience of other particles (companions). In PSO, a particle is defined as a moving point in hyperspace. For each particle, at the current time step, a record is kept of the position, velocity, and the best position found in the search space so far.

•Simulated Annealing (SA)

Simulated annealing (SA) is a probabilistic technique for approximating the global optimum of a given function.The name of the algorithm comes from annealing in metallurgy, a technique involving heating and controlled cooling of a material to alter its physical properties.
Simulated Annealing is a generic probabilistic meta-algorithm used to find an approximate solution to global optimization problems. The Simulated Annealing algorithm starts with a random solution. A random nearby solution is formed by every iteration. If this solution is a better solution, it will replace the current solution.If it was a worse solution, it may be chosen to replace the current solution with a probability that depends on the temperature parameter. As the algorithm progresses, the temperature parameter decreases, giving worse solutions a lesser chance of replacing the current solution.
We allow the worst solutions at the beginning to avoid solutions converging to a local minimum rather than the global minimum. We will use the simulated annealing algorithm to solve the Random Travelling Salesman Problem. The salesman must visit each city only once

and return to the same city in which he began. The constraints that affect the outcome of the algorithm are the initial temperature, the rate at which the temperature decreases and the stopping condition of the algorithm.



•K-opt Heuristics

K-opt procedures (k>=2) belong to the class of neighborhood search algorithms (alternatively called local search algorithms).

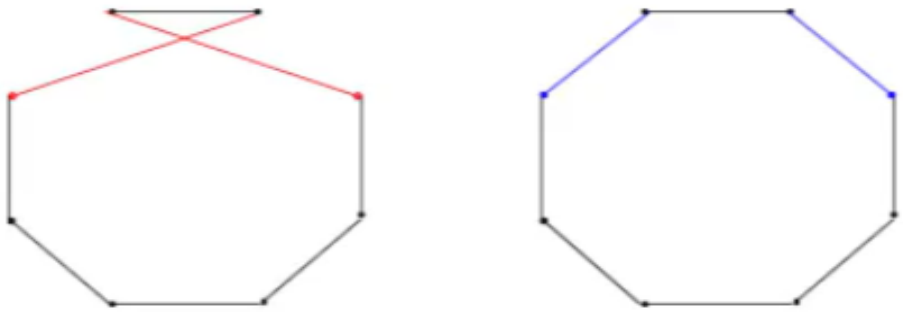
They are some of the oldest algorithms and the most extensively used heuristics developed for the Travel Salesman Problem.

Definition: l-exchange is a procedure that replaces l edges currently in the tour with l edges so that the result is again a travelling salesman tour .

For instance , given tour through set V, say

(l1,l2,.....ln)

A 2-exchange procedure could replace edges {lu,lu+1} and {lv,lv+1} with {lu,lv} and {lu+1,lv+1} and resulted in a new tour.



Example of 2 -Exchange

IMPLEMENTATION:

The experimental results are produced and tested by MATLAB software. To encode the inputs (coordinates of cities, or distance between cities), each dataset is stored in an excel file and read by MATLAB as a matrix.

In this paper we use six datasets of cities with different sizes:

1. Arabic 24.tsp: 24 capital cities in Arabic world
2. berlin52.tsp: 52 locations in Berlin
3. ch150.tsp: 150 cities (churritz)
4. lin318.tsp: 318 cities (Lin/Kernighan)
5. fl1400.tsp: 1400 Drilling (Reinelt)
6. brd 14051.tsp: 14051 BR Deutschland in den Grenzer von 1989 (Bachem/Wottawa)

The Machine Specification that used to implement the experimental results is:

- HP - Z420 Workstation Desktop

Available Processors: 4
Available Cores : i7
Available Memory : 4 G
Available Windows : WIN-7, 64-bit

As shown in the below table, all techniques work well with suitable running time for dataset with small size up to tens of cities. But for datasets with hundreds of cities some techniques converge to local minimum solution, which is not the best one.

On other hand, for thousands of cities all techniques suffer to converge the solution and we cannot guarantee the best solution. Also, the techniques show different results within different long running time.

For dataset with 14050 cities, GA, ABC and BFO cannot complete for different reasons: GA needs more memory,

ABC faced a cycling problem, while BFO did not converge at all running times. On the other side, three techniques ACO,

PSO, and SA complete all datasets although they spent a lot of time (around 24 hours) to converge solutions that are not guaranteed to be best.

Dataset		GA	ACO	PSO	SA
Arabic24	Best Value	229	229	229	229
	# of Iterations	131	155	104	97
	Run Time	1 s.	0 s.	0 s.	0 s.
berlin52	Best Value	7544	7544	7544	7544
	# of Iterations	129	128	116	101
	Run Time	6 s.	4	3	3
ch150	Best Value	6873	6873	6874	6872
	# of Iterations	248	192	207	176
	Run Time	70 s.	27	20	19
lin318	Best Value	47,940	47,941	47,941	47,936
	# of Iterations	615	582	568	571
	Run Time	848 s.	562	517	449
fl1400	Best Value	57,761	57,754	57,749	57,754
	# of Iterations	20,898	21,826	19,517	22,038
	Run Time	32,863 s.	26,352 s.	24,798	20,997
brd14051	Best Value	Out of time	8,183,942	7,970,284	7,963,861
	# of Iterations		152,729	128,097	133,410
	Run Time		87,519	85,725	81,839

Results of All Techniques To Measure Best Value (Path), No. Of Iteration, And Running Time

We can determine if the solution is wrong or may be the best solution from a graph of tour. As in figures of dataset solutions, if there are intersecting lines on the path of solution, we can conclude that solution is not optimal (best), the best solution should be without intersecting lines.

CONCLUSION:

This paper presents some optimization techniques for solving the Travelling Salesman Problem(TSP). The paper starts by explaining why traditional methods like **brute force**(naive method) and **dynamic programming(**DP) are not able to solve TSP optimally as well as why it is better off to just focus on approximating algorithms to solve the Traveling Salesman Problem. The paper then studies and explains Minimum Spanning Tree which helps to solve the Travelling Salesman Problem. The paper then studies how to create a minimum spanning tree using two popular algorithms: **Prim's algorithm** and **Kruskal's method.** After explaining Traveling Salesman Problem ,the paper proceeds to construct an algorithm to solve TSP with MST along with **A* tree traversal**.

Next the paper proceeds towards some modern optimization techniques for solving TSP.All techniques produce good results for small dataset but may be not optimal if the size will increase. The running time and path distance increases with the increasing number of cities.The quality of the results of any dataset increases with increasing population size but when the population size increases, the running time is increased. So, we must do the best to balance between runtime and the solution quality.

RESULT:

Thus, from our experiments we have gathered that in order to solve TSP, we need a fully connected unidirectional graph and we need to start by building its Minimum spanning tree. The MST can be constructed using Prim's algorithm or Kruskal's algorithm. Then we can make use of the MST heuristic which is to traverse the MST in a DFS(Depth First Search) manner to give us a path that helps us visit all of the visits exactly once. However, since this does not take non-MST edges into account, it is merely an estimation and may not give the most optimal output needed.

In TSP, as we know a salesman needs to travel all cities and return to the original city via a path between the cities. We represent cities as nodes and paths as edges in a graph. From the implementation of A* algorithm with help of MST

heuristics for solving TSP,it is NP hard. So, there is no polynomial time complexity algorithm that gives the most optimized answer for each test case. So, we apply A* algorithm where actual cost is a distance (weights) between nodes.

There are different kinds of heuristics to solve A* algorithm like nearest neighbor and minimum spanning tree. In the nearest neighbor heuristics algorithm checks all the paths from a node and selects the shortest path. It has one drawback that it can end up with a non-optimal path if the initial sub-path is long and final sub-path is shorter.

So, the minimum spanning tree heuristics algorithm makes an MST of a graph and then selects the path present in MST. Finally, by repeatedly selecting the MST path it will give a minimum total cost and an optimized path.

ACKNOWLEDGEMENT:

We thank Dr. Poulami Dalapati for giving us this opportunity to have a deeper understanding on the topic "TRAVELLING SALESMAN PROBLEM".We would extend our regards to all our group members from the LNM Institute of Information Technology who provided insight and expertise that greatly assisted this project.

REFERENCES:

- V. Cerny, Thermodynamical Approach to the Traveling Salesman Problem: An Efficient Simulation Algorithm
- Gerard Reinelt. The Traveling Salesman: Computational Solutions for TSP Applications. Springer-Verlag, 1994.
- Genetic Algorithms and the Traveling Salesman Problem by Kylie Bryant December 2000.
- Emile Aarts, Jan Korst, and Wil Michiels. Simulated Annealing, chapter 7.
- Rajesh Matai, Surya Prakash Singh and Murari Lal Mittal, Traveling Salesman Problem: An Overview of Applications, Formulations, and Solution Approaches.