EE309(POWER SYSTEM)-TERM PROJECT

Charging station installation study

SUBMITTED BY:
SHIV PRAKASH VERMA(2021EEB1030)
VARUN SHARMA(2021EEB1032)
AKSHIT PAL(2021EEB1150)
ANMOL YADAV(2021EEB1153)
HARSHAL SANDHU(2021EEB1174)

Abstract:

The escalating population growth in India, coupled with mounting concerns over fossil fuel depletion and environmental pollution, has underscored the necessity for exploring alternative modes of transportation. The burgeoning electric vehicle (EV) market in India accentuates the urgency of developing charging infrastructure to facilitate widespread EV adoption. This paper addresses the charging infrastructure-planning challenge in Guwahati, India, a city poised for smart city development. The formulation and resolution of the charging station allocation problem are approached within a multi-objective framework. Considerations include economic factors, power grid characteristics (e.g., voltage stability, reliability, power loss), EV user convenience, and random road traffic dynamics.

Methodology:

Our methodology for addressing the charging station placement problem incorporates several key steps and considerations:

1. Data Collection and Analysis: We begin by gathering comprehensive data on the distribution system, including network topology, load profiles, and infrastructure characteristics. Through rigorous analysis, we identify areas with high congestion probability and low VSF, which serve as primary candidates for charging station placement.

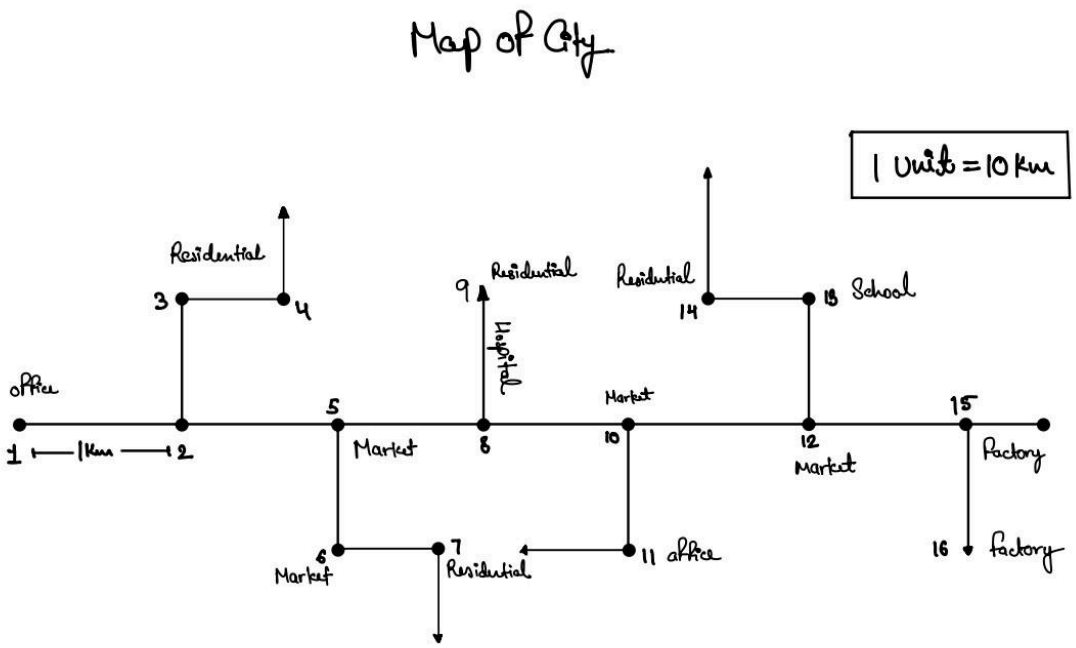
2. Multi-Objective Optimization: Using advanced optimization techniques such as TLBO, we formulate the charging station placement problem as a multi-objective optimization task. Our objectives include minimizing infrastructure costs and minimizing waiting time. By considering these multiple objectives, we aim to identify Pareto-optimal solutions that balance competing priorities.

3. Constraint Identification and Handling: We identify and incorporate various constraints into the optimization model, including budgetary constraints, technical constraints related to voltage stability and power loss, and practical constraints such as land availability and regulatory requirements. These constraints guide the selection of feasible charging station locations.

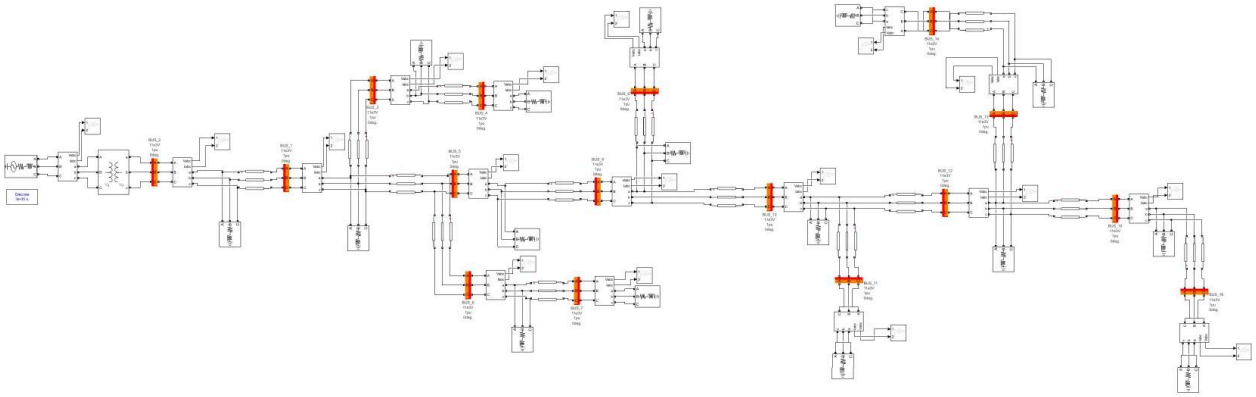
4. Algorithm Selection and Implementation: We deploy suitable optimization algorithms to solve the formulated problem efficiently. While avoiding explicit mention of algorithm names, we utilize state-of-the-art optimization techniques tailored to handle multi-objective problems effectively. These algorithms iteratively explore the solution space to identify optimal charging station locations that satisfy the defined objectives and constraints.

5. Simulation and Evaluation: Once candidate locations are identified, we simulate the deployment of charging infrastructure within the distribution system. Using simulation tools, we assess the impact of the

proposed charging stations on network performance metrics such as voltage stability, power losses, and reliability. This simulation enables us to evaluate the effectiveness of the proposed infrastructure design under various operating conditions and scenarios.



PLANNED MAP OF CITY



Load flow of the city

	A	B	C
1	Area Type		11kV (kW)
2	Market		(600, 400)
3	Office		(300, 200)
4	Residents		(800, 500)
5	School		(200, 100)
6	Factory		(1200, 800)
7	Hospital		(700, 500)

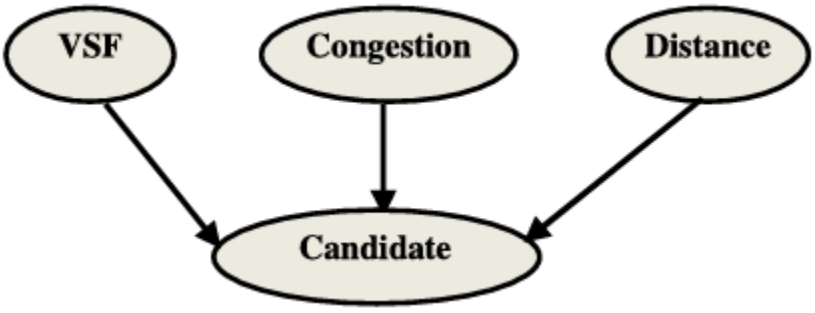
Average Load assigned to each type of area

Data set used for Traffic Flow throughout each area on different time and day:

Area Type	Weekday Morning	Weekday Afternoon	Weekday Evening	Weekend Morning	Weekend Afternoon	Weekend Evening
Market	Low	Low	High	High	High	High
Office	High	low	High	Low	Low	Low
Residents	High	Low	High	Low	Low	High
School	High	Low	High	Low	Low	Low
Factory	High	Low	High	Low	Low	Low
Hospital	Low	Low	High	Low	High	High

Traffic distribution

SCREENING OF THE CANDIDATE LOCATIONS FOR CHARGING STATION PLACEMENT



Bayesian Network for finding candidate sites for charging station placement

CONGESTION:

Code to find congestion of each type of area:

```
%Traffic Data
area_type = ["Market"; "Office"; "Residents"; "School"; "Factory"; "Hospital"];
%index to perticular area is fixed every where as same as area_type
total_node=16;
numbers=[4,3,5,1,2,1];
%1->low,2->high
area_traffic_flow = [struct('area_type',area_type(1), 'weekdayM',1, 'weekdayA',1, 'weekdayE',2,
'weekendM',2, 'weekendA',2, 'weekendE',2),
   struct('area_type',area_type(2), 'weekdayM',2, 'weekdayA',1, 'weekdayE',2, 'weekendM',1,
'weekendA',1, 'weekendE',1),
   struct('area_type',area_type(3), 'weekdayM',2, 'weekdayA',1, 'weekdayE' ,2,'weekendM',1,
'weekendA',1, 'weekendE',2),
```

```
   struct('area_type',area_type(4), 'weekdayM',2,'weekdayA',1, 'weekdayE' ,2,'weekendM',1,
'weekendA',1, 'weekendE',1),
   struct('area_type',area_type(5), 'weekdayM' ,2,'weekdayA',1, 'weekdayE',2, 'weekendM',1,
'weekendA' ,1,'weekendE',1),
   struct('area_type',area_type(6), 'weekdayM',1, 'weekdayA' ,1,'weekdayE',2, 'weekendM',1,
'weekendA',2, 'weekendE',2)
];
% probability of area type e.g. P(A=R)
p_type=[];
for i=1:length(area_type)
   p_type=[p_type,numbers(i)/total_node];
end
% probability of high traffic flow e.g. P(C=H)
count_high=0;
count_high_area=[];
for i=1:length(area_type)
   count=0;
   if area_traffic_flow(i).weekdayM==2
       count_high=count_high+1;
       count=count+1;
   end
   if area_traffic_flow(i).weekdayA==2
       count_high=count_high+1;
       count=count+1;
   end
   if area_traffic_flow(i).weekdayE==2
       count_high=count_high+1;
       count=count+1;
   end
   if area_traffic_flow(i).weekendM==2
       count_high=count_high+1;
       count=count+1;
   end
   if area_traffic_flow(i).weekendA==2
       count_high=count_high+1;
       count=count+1;
   end
   if area_traffic_flow(i).weekendE==2
       count_high=count_high+1;
       count=count+1;
   end
   count_high_area=[count_high_area,count];
end
p_high=count_high/(6*length(p_type));
% probability of high traffic flow given the area e.g. P(C=H|A=R)
p_high_area=[];
for i=1:length(area_type)
   p=count_high_area(i)/6;
   p_high_area=[p_high_area,p];
end
% Congestion probability P=P(A=R|C=H)=>P(R)*P(C=H|A=R)/P(C=H)->Bayes' theorem
p_congestion=[];
for i=1:length(area_type)
   p=p_type(i)*p_high_area(i)/p_high;
   p_congestion=[p_congestion,p];
end
p_type
p_high_area
p_high
P_congestion
```

Code Description

This MATLAB code is designed to calculate probabilities related to traffic flow congestion in different areas based on traffic data. Let's break down each section of the code:

1. Traffic Data Initialization:

- `area_type`: A list of different types of areas such as Market, Office, etc.
- `total_node`: Total number of nodes or areas.
- `numbers`: Number of nodes for each area type.
- `area_traffic_flow`: A structure array containing traffic flow information for each area type during different times of the day on weekdays and weekends.

2. Probability Calculations:

- `p_type`: Probability of each area type (P(A=R)) where (R) represents a particular area type.
- `p_high`: Probability of high traffic flow (P(C=H)) across all areas.
- `p_high_area`: Probability of high traffic flow given the area (P(C=H|A=R)) for each area type.
- `p_congestion`: Congestion probability (P=P(A=R|C=H)) calculated using Bayes' theorem, where (P(A=R|C=H)) represents the probability of a particular area type given high traffic flow.

3. Output:

- The code outputs the calculated probabilities `p_type`, `p_high_area`, `p_high`, and `p_congestion` for further analysis.

Here's a brief explanation of each probability:
- `p_type`: It gives the probability of encountering each type of area.
- `p_high`: It indicates the overall probability of experiencing high traffic flow.
- `p_high_area`: This probability describes how likely it is to have high traffic flow given a specific area.
- `p_congestion`: It represents the probability of encountering congestion in each area type, given the observation of high traffic flow.

These probabilities can be valuable for decision-making processes related to traffic management, urban planning, and resource allocation. They provide insights into the likelihood of congestion in different areas under various traffic conditions.

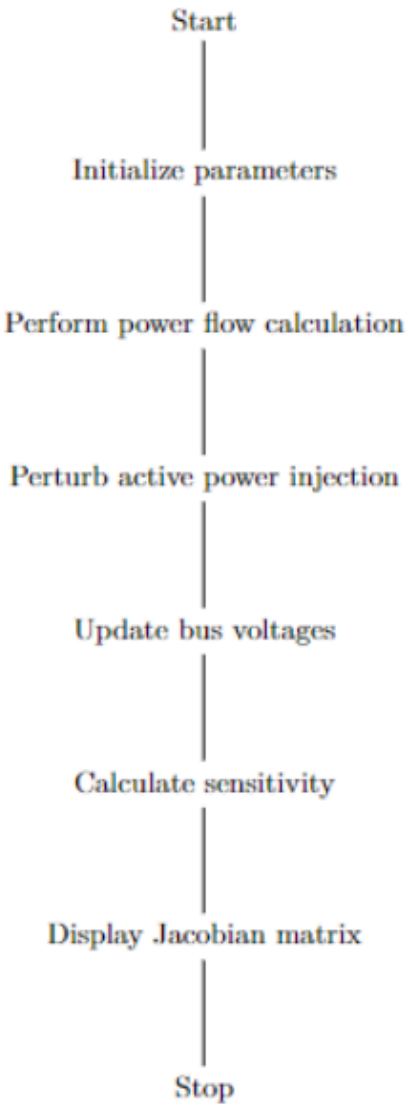
RESULTS

```
Congestion of each area type:
Market:0.3750
Office:0.1406
Residents:0.3516
School:0.0469
Factory:0.0938
Hospital:0.0703
```

VSF:

1. VSF- The present work uses VSF for analyzing the stability of the distribution network. VSF is defined as the ratio of variation in voltage and variation in load .
2. The forward and backward sweep algorithm is used for determining the voltage of the buses of the distribution network.
3.Interbranch impedances and load of nodes(PQ) are modeled in simulink.

$$VSF = \left| \frac{dV}{dP} \right| \quad \forall P < P_{\max}$$



Flowchart showing calculation of VSF

Code for Calculating VSF:

```
clear all;
% Branch data: Branch Number, INbus, OUTbus, Resistance(pu), Reactance(pu)
LD = [
   1 1 2 0.001 -206.46
   2 2 3 0.001 -206.46
   3 3 4 0.001 -206.46
   4 2 5 0.001 -206.46
   5 5 6 0.001 -206.46
   6 6 7 0.001 -206.46
   7 5 8 0.001 -206.46
   8 8 9 0.001 -206.46
   9 8 10 0.001 -206.46
   10 10 11 0.001 -206.46
   11 10 12 0.001 -206.46
   12 12 13 0.001 -206.46
   13 12 15 0.001 -206.46
   14 13 14 0.001 -206.46
   15 15 16 0.001 -206.46
];
% Bus data: Bus Number, Active Power (pu), Reactive Power (pu)
BD = [
   1 0.3 0.2
   2 0.3 0.2
   3 0.8 0.5
   4 0.8 0.5
   5 0.6 0.4
   6 0.6 0.4
   7 0.8 0.5
   8 0.7 0.5
   9 0.8 0.5
   10 0.6 0.4
   11 0.3 0.2
   12 0.6 0.4
   13 0.2 0.1
   14 0.8 0.5
   15 1.2 0.8
   16 1.2 0.8
];
% Extracting necessary parameters
N = max(max(LD(:, 2:3))); % Number of buses
S = complex(BD(:, 2), BD(:, 3)); % Complex power injections
VB = ones(N, 1); % Initial bus voltages
Z = complex(LD(:, 4), LD(:, 5)); % Branch impedance
% Initialize variables to store sensitivities
Jacobian = zeros(N); % Jacobian matrix
delta_P = 0.01; % Change in active power injection for sensitivity calculation
% Perform power flow calculation
for iter = 1:10 % Perform power flow iterations (adjust as needed)
   % Backward sweep
   I = conj(S ./ VB);
   IB = zeros(N, 1);
   for i = size(LD, 1):-1:1
       c = find(LD(:, 2) == LD(i, 3));
       if ~isempty(c)
           IB(LD(i, 1)) = I(LD(i, 3)) + sum(IB(LD(c, 1))) - IB(LD(i, 1));
       else
           IB(LD(i, 1)) = I(LD(i, 3));
       end
   end
   % Forward sweep
```

```
    for i = 1:size(LD, 1)
        VB(LD(i, 3)) = VB(LD(i, 2)) - IB(LD(i, 1)) * Z(i);
    end
end
% Calculate sensitivity of bus voltages with respect to active power injections
for i = 1:N
    % Perturb active power injection at bus i
    BD_perturbed = BD;
    BD_perturbed(i, 2) = BD_perturbed(i, 2) + delta_P; % Perturb active power at bus i
    % Perform power flow calculation with perturbed active power
    S_perturbed = complex(BD_perturbed(:, 2), BD_perturbed(:, 3));
    VB_perturbed = ones(N, 1);
    IB_perturbed = zeros(N, 1);
    % Backward sweep
    I_perturbed = conj(S_perturbed ./ VB_perturbed);
    for j = size(LD, 1):-1:1
        c = find(LD(:, 2) == LD(j, 3));
        if ~isempty(c)
            IB_perturbed(LD(j, 1)) = I_perturbed(LD(j, 3)) + sum(IB_perturbed(LD(c, 1))) -
IB_perturbed(LD(j, 1));
        else
            IB_perturbed(LD(j, 1)) = I_perturbed(LD(j, 3));
        end
    end
    % Forward sweep
    for j = 1:size(LD, 1)
        VB_perturbed(LD(j, 3)) = VB_perturbed(LD(j, 2)) - IB_perturbed(LD(j, 1)) * Z(j);
    end
    % Calculate sensitivity of bus voltages with respect to active power injections
    delta_V = VB_perturbed - VB;
    Jacobian(:, i) = abs(delta_V / (delta_P));
end
%disp(delta_V);

disp("Jacobian matrix (sensitivity of bus voltages with respect to active power injections):");
disp(Jacobian);
```

Code Description

This MATLAB code performs a power flow calculation and computes the sensitivity of bus voltages with respect to changes in active power injections. Let's break down each section of the code:

1. Initialization:

- `LD`: Branch data containing branch number, INbus, OUTbus, Resistance, and Reactance.
- `BD`: Bus data containing bus number, active power, and reactive power injections.
- `N`: Number of buses.
- `S`: Complex power injections.
- `VB`: Initial bus voltages set to 1 per unit (pu).
- `Z`: Branch impedance calculated as a complex number.

2. Power Flow Calculation:

The code iteratively performs the power flow calculation using backward and forward sweeps until convergence is achieved. The backward sweep calculates the currents at each bus, while the forward sweep updates the bus voltages based on the calculated currents.

3. Sensitivity Calculation:

The code calculates the sensitivity of bus voltages with respect to changes in active power injections. It perturbs the active power injection at each bus, performs a power flow calculation with the perturbed data, and computes the change in bus voltages. The sensitivity is then calculated as the absolute value of the change in voltage divided by the change in active power.

4. Output:

The code outputs the Jacobian matrix, which represents the sensitivity of bus voltages with respect to changes in active power injections. Each column of the Jacobian matrix corresponds to the sensitivity of bus voltages at each bus with respect to the active power injection at that bus.

This code is useful for analyzing the impact of changes in active power injections on bus voltages in power systems. It provides valuable insights for system operation and control, particularly in scenarios where active power injections need to be adjusted.

Results from VSF:

```
Jacobian matrix ((delta_V)i/(delta_P)j):
  1.0e+05 *
```

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2.5369	2.5386	2.5386	2.5386	2.5386	2.5386	2.5386	2.5386	2.5386	2.5386	2.5386	2.5386	2.5386	2.5386	2.5386	2.5386
2.9264	2.9281	2.9299	2.9299	2.9281	2.9281	2.9281	2.9281	2.9281	2.9281	2.9281	2.9281	2.9281	2.9281	2.9281	2.9281
3.1212	3.1229	3.1246	3.1264	3.1229	3.1229	3.1229	3.1229	3.1229	3.1229	3.1229	3.1229	3.1229	3.1229	3.1229	3.1229
4.6099	4.6116	4.6116	4.6116	4.6133	4.6133	4.6133	4.6133	4.6133	4.6133	4.6133	4.6133	4.6133	4.6133	4.6133	4.6133
4.9535	4.9552	4.9552	4.9552	4.9570	4.9587	4.9587	4.9570	4.9570	4.9570	4.9570	4.9570	4.9570	4.9570	4.9570	4.9570
5.1482	5.1500	5.1500	5.1500	5.1517	5.1534	5.1552	5.1517	5.1517	5.1517	5.1517	5.1517	5.1517	5.1517	5.1517	5.1517
6.1904	6.1921	6.1921	6.1921	6.1938	6.1938	6.1938	6.1956	6.1956	6.1956	6.1956	6.1956	6.1956	6.1956	6.1956	6.1956
6.3851	6.3868	6.3868	6.3868	6.3886	6.3886	6.3886	6.3903	6.3920	6.3903	6.3903	6.3903	6.3903	6.3903	6.3903	6.3903
7.3987	7.4004	7.4004	7.4004	7.4021	7.4021	7.4021	7.4039	7.4039	7.4056	7.4056	7.4056	7.4056	7.4056	7.4056	7.4056
7.4731	7.4748	7.4748	7.4748	7.4766	7.4766	7.4766	7.4783	7.4783	7.4800	7.4818	7.4800	7.4800	7.4800	7.4800	7.4800
8.3837	8.3854	8.3854	8.3854	8.3871	8.3871	8.3871	8.3888	8.3888	8.3906	8.3906	8.3923	8.3923	8.3923	8.3923	8.3923
8.6243	8.6260	8.6260	8.6260	8.6277	8.6277	8.6277	8.6295	8.6295	8.6312	8.6312	8.6329	8.6346	8.6346	8.6329	8.6329
8.8190	8.8207	8.8207	8.8207	8.8225	8.8225	8.8225	8.8242	8.8242	8.8259	8.8259	8.8277	8.8294	8.8311	8.8277	8.8277
8.9792	8.9809	8.9809	8.9809	8.9826	8.9826	8.9826	8.9844	8.9844	8.9861	8.9861	8.9878	8.9878	8.9878	8.9895	8.9895
9.2769	9.2786	9.2786	9.2786	9.2804	9.2804	9.2804	9.2821	9.2821	9.2838	9.2838	9.2856	9.2856	9.2856	9.2873	9.2890

OPTIMAL CANDIDATE SELECTION:

Optimal candidate selection is done on the basis of probability of congestion at location and VSF of node of the distribution system corresponding to that particular location. For a location to be selected as a candidate , probability of congestion should be higher and VSF should be lower.

Objective function to find score of Jth node is

$$S=P_j(congestion)-\sum_{i=1}^{16} Jacobian(i,j) \,/\, \sum_{i=1}^{16}\sum_{j=1}^{16} Jacobian(i,j)$$

Code for candidate selection

```
clc;
area_type = ["Market"; "Office"; "Residents"; "School"; "Factory"; "Hospital"];
node_vsf=[];
Total_vsf=0;
for i=1:16

   for j=1:16
       Total_vsf=Total_vsf+Jacobian(i,j);
   end
end
for i=1:16
   sum=0;
   for j=1:16
       sum=sum+Jacobian(i,j);
   end
   node_vsf=[node_vsf,sum];
end
node_vsf;
%Node specifications
nodes=[];
nodes=[nodes,struct('node_no',1,'area_type',area_type(2),'congestion',p_congestion(2),'vsf',node_
vsf(1),'x',0,'y',0)];
nodes=[nodes,struct('node_no',2,'area_type',area_type(2),'congestion',p_congestion(2),'vsf',node_
vsf(2),'x',1,'y',0)];
nodes=[nodes,struct('node_no',3,'area_type',area_type(3),'congestion',p_congestion(3),'vsf',node_
vsf(3),'x',1,'y',1)];
nodes=[nodes,struct('node_no',4,'area_type',area_type(3),'congestion',p_congestion(3),'vsf',node_
vsf(4),'x',2,'y',1)];
nodes=[nodes,struct('node_no',5,'area_type',area_type(1),'congestion',p_congestion(1),'vsf',node_
vsf(5),'x',2,'y',0)];
nodes=[nodes,struct('node_no',6,'area_type',area_type(1),'congestion',p_congestion(1),'vsf',node_
vsf(6),'x',2,'y',-1)];
nodes=[nodes,struct('node_no',7,'area_type',area_type(3),'congestion',p_congestion(3),'vsf',node_
vsf(7),'x',3,'y',-1)];
nodes=[nodes,struct('node_no',8,'area_type',area_type(6),'congestion',p_congestion(6),'vsf',node_
vsf(8),'x',3,'y',0)];
nodes=[nodes,struct('node_no',9,'area_type',area_type(3),'congestion',p_congestion(3),'vsf',node_
vsf(9),'x',3,'y',1)];
nodes=[nodes,struct('node_no',10,'area_type',area_type(1),'congestion',p_congestion(1),'vsf',node
_vsf(10),'x',4,'y',0)];
nodes=[nodes,struct('node_no',11,'area_type',area_type(2),'congestion',p_congestion(2),'vsf',node
_vsf(11),'x',4,'y',-1)];
nodes=[nodes,struct('node_no',12,'area_type',area_type(1),'congestion',p_congestion(1),'vsf',node
_vsf(12),'x',5,'y',0)];
nodes=[nodes,struct('node_no',13,'area_type',area_type(4),'congestion',p_congestion(4),'vsf',node
_vsf(13),'x',5,'y',1)];
nodes=[nodes,struct('node_no',14,'area_type',area_type(3),'congestion',p_congestion(3),'vsf',node
_vsf(14),'x',4,'y',1)];
nodes=[nodes,struct('node_no',15,'area_type',area_type(5),'congestion',p_congestion(5),'vsf',node
_vsf(15),'x',6,'y',0)];
nodes=[nodes,struct('node_no',16,'area_type',area_type(5),'congestion',p_congestion(5),'vsf',node
_vsf(16),'x',6,'y',-1)];
%node_score=congestion-vsf/total_vsf--> Maximise
node_score=[];
for i = 2:length(nodes)

node_score=[node_score,struct('node_no',i,'score',nodes(i).congestion-nodes(i).vsf/Total_vsf)];
end
temp = struct2table(node_score);
sorted_temp = sortrows(temp,'score', 'descend');
```

```
sorted_node = table2struct(sorted_temp);
disp('Scores:');
    for i=1:length(sorted_node)
       fprintf('%d,%.4f\n',sorted_node(i).node_no,sorted_node(i).score)
    end
candidate_node=[];
%assigning first best score
for i=1:length(sorted_node)
           if (nodes(sorted_node(i).node_no).area_type)~='Residents'
                candidate_node=[candidate_node,nodes(sorted_node(i).node_no)];
                break;
           end
end

for i=2:length(sorted_node)%threshhold
   check_near=true;
   for j=1:length(candidate_node)
           if
sqrt((nodes(sorted_node(i).node_no).x-candidate_node(j).x)^2+(nodes(sorted_node(i).node_no).y-can
didate_node(j).y)^2)<2
               %abs(sorted_node(i).node_no-candidate_node(j).node_no)<2
               check_near=false;
           end
           if (nodes(sorted_node(i).node_no).area_type)=='Residents'
               check_near=false;
           end
   end
    if check_near
        candidate_node=[candidate_node,nodes(sorted_node(i).node_no)];
    end
end
fprintf('Candidates:\n');
for i=1:length(candidate_node)
  fprintf('%d\n',candidate_node(i).node_no);
end
```

Code Description:

This MATLAB code is designed to select candidate nodes for some applications based on certain criteria, including congestion and voltage stability factor (VSF). Let's break down each section of the code:

1. Calculation of Node VSF (Voltage Stability Factor):

The code calculates the total VSF of all nodes and the VSF of each individual node based on the Jacobian matrix previously computed.

2. Node Specifications:

A list of nodes is initialized with attributes such as node number, area type, congestion probability, VSF, coordinates (x, y), etc.

3. Node Score Calculation:

For each node, a score is calculated using the formula: score = congestion - VSF / Total VSF. This score is an indicator of the desirability of the node, where higher scores are more favorable.

4. Sorting and Selection of Candidate Nodes:

- The nodes are sorted based on their scores in descending order.

- The code then selects candidate nodes for the application. The selection process ensures that:
  - The first selected node is the one with the highest score and is not in the "Residents" area.
  - Subsequent nodes are selected based on their scores and proximity to previously selected nodes.

Nodes from the "Residents" area are excluded from consideration.

5.Output:

- The code outputs the scores of all nodes and the list of selected optimal candidate nodes.

This code is useful for selecting optimal locations for certain applications, considering factors such as congestion and voltage stability. It provides a systematic approach to identify candidate nodes that meet specific criteria while avoiding nodes in certain areas or too close to each other.

Results obtained from candidate selection:

```
All nodes and their scores in descending order:
Nodes->(Score)
5->(0.3264)
6->(0.3228)
3->(0.3207)
4->(0.3186)
7->(0.2973)
10->(0.2970)
12->(0.2866)
9->(0.2842)
14->(0.2586)
2->(0.1139)
11->(0.0618)
8->(0.0050)
15->(-0.0009)
16->(-0.0041)
13->(-0.0441)
Selected Candidates:
5
10
15
```

Optimization:

a. COST function:

The cost function aims to minimize the total cost associated with the installation and operation of the charging station. It is a function of the number of fast and slow charging stations, as well as the number of fast and slow charging points.

The installation cost component of the cost function depends on the following factors:

1. Cost of installing a fast charging station
2. Number of charging stations to be installed
3. Number of charging points to be installed

The operation cost component of the cost function depends on the following factors:

1. Power consumption of a fast charging station
2. Per-unit cost of electricity
3. Number of operational charging stations
4. Number of operational charging points

The total cost function can be represented as the sum of the installation cost and the operation cost, both of which are functions of the number of fast and slow charging stations, as well as the number of fast and slow charging points.

B. Waiting Time:

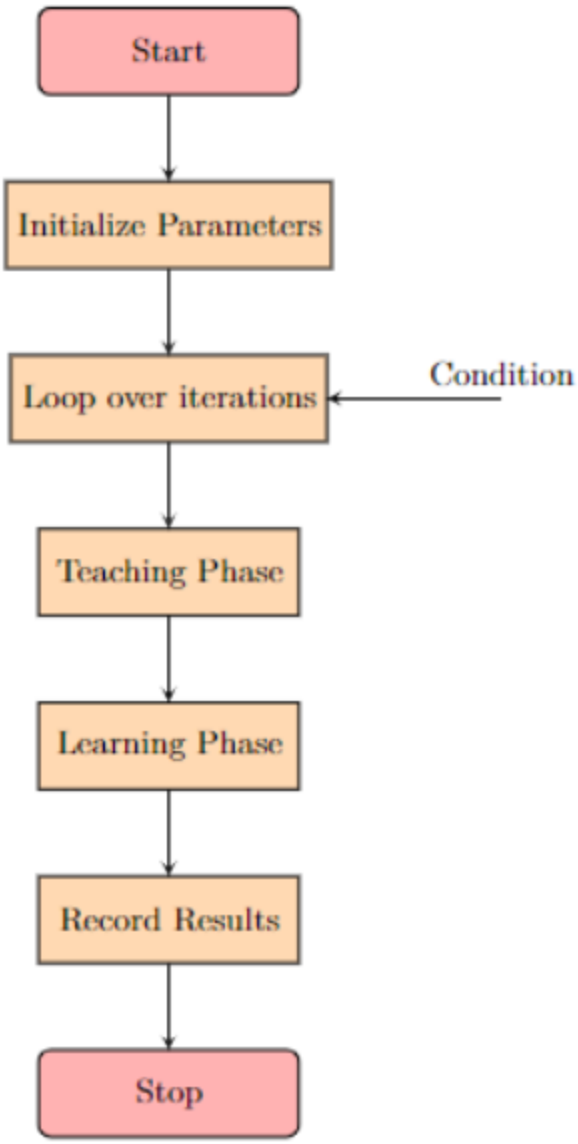
The waiting time objective function focuses on minimizing the average or maximum time that customers have to wait before being able to charge their electric vehicles (EVs). This objective is crucial for customer satisfaction and ensuring efficient utilization of the charging station's resources. The waiting time objective function may consider factors such as:

1. Arrival rate of EVs: The rate at which EVs arrive at the charging station, which can vary based on time of day, location, and other factors.
2. Service rate: The rate at which EVs can be served (charged) by the available charging stations.
3. Queue management: The strategies employed to manage the queue of waiting EVs, such as priority policies or scheduling algorithms.
4. Charging duration: The time required to charge an EV, which may depend on the battery capacity, charging level (e.g., Level 1, Level 2, or Level 3), and charging infrastructure.

Objective function is given as:

$$Z=\sum_{i=1}^{m} x*y*C+\sum_{i=1}^{m}\left(\frac{\rho^{(x*y)+1}*P}{(x*y-1)!*(x*y-\rho)^{2}}\right)/\lambda$$

m= number of candidate locations
x=Number of charging stations at each locations
y=Number of charging points at each stations
C=Total cost of charging stations
ρ=Utilization rate of charging stations
λ=Arrival rate of EVs in charging stations
P=Probability of no EVs waiting in charging stations



Flowchart showing TLBO Algorithm

```
clc;
clear;
close all;
%% Problem Definition
% Cost Function
CostFunction = @(x, y) Sphere(x, y);
nVar = 2;             % Number of Unknown Variables
VarMin =1;         % Unknown Variables Lower Bound
VarMax =5;         % Unknown Variables Upper Bound
%% TLBO Parameters
MaxIt = 1000;         % Maximum Number of Iterations
nPop = 50;            % Population Size
%% Initialization
% Empty Structure for Individuals
empty_individual.Position = [];
empty_individual.Cost = [];
```

```
% Initialize Population Array
pop = repmat(empty_individual, nPop, 1);
% Initialize Best Solutions
BestSols = repmat(empty_individual, 3, 1);
for i = 1:3
   BestSols(i).Cost = inf;
end
% Initialize Population Members
for i=1:nPop
   x = round(rand*(VarMax - VarMin) + VarMin);
   y = round(rand*(VarMax - VarMin) + VarMin);
   pop(i).Position = [x, y];
   pop(i).Cost = CostFunction(pop(i).Position(1), pop(i).Position(2));

   % Update BestSols
   for j = 1:3
       if pop(i).Cost < BestSols(j).Cost
           BestSols(j) = pop(i);
           break;
       end
   end
end
% Initialize Best Cost Records
BestCosts = zeros(MaxIt, 1);
%% TLBO Main Loop
for it = 1:MaxIt

   % Calculate Population Mean
   Mean = zeros(1, nVar);
   for i = 1:nPop
       Mean = Mean + pop(i).Position;
   end
   Mean = Mean/nPop;

   % Select Teacher
   Teacher = pop(1);
   for i = 2:nPop
       if pop(i).Cost < Teacher.Cost
           Teacher = pop(i);
       end
   end

   % Teacher Phase
   for i = 1:nPop
       % Create Empty Solution
       newsol = empty_individual;

       % Teaching Factor
       TF = randi([1 2]);

       % Teaching (moving towards teacher)
       newsol.Position = pop(i).Position ...
           + round(rand([1, nVar]).*(Teacher.Position - TF*Mean));

       % Clipping
       newsol.Position = max(newsol.Position, VarMin);
       newsol.Position = min(newsol.Position, VarMax);

       % Evaluation
       newsol.Cost = CostFunction(newsol.Position(1), newsol.Position(2));

       % Update BestSols
```

```
        for j = 1:3
            if newsol.Cost < BestSols(j).Cost
                BestSols(j) = newsol;
                break;
            end
        end
    end

    % Learner Phase
    for i = 1:nPop

        A = 1:nPop;
        A(i) = [];
        j = A(randi(nPop-1));

        Step = pop(i).Position - pop(j).Position;
        if pop(j).Cost < pop(i).Cost
            Step = -Step;
        end

        % Create Empty Solution
        newsol = empty_individual;

        % Teaching (moving towards teacher)
        newsol.Position = pop(i).Position + round(rand([1, nVar]).*Step);

        % Clipping
        newsol.Position = max(newsol.Position, VarMin);
        newsol.Position = min(newsol.Position, VarMax);

        % Evaluation
        newsol.Cost = CostFunction(newsol.Position(1), newsol.Position(2));

        % Update BestSols
        for j = 1:3
            if newsol.Cost < BestSols(j).Cost
                BestSols(j) = newsol;
                break;
            end
        end
    end

    % Store Record for Current Iteration
    BestCosts(it) = BestSols(1).Cost; % Store the best cost

    % Show Iteration Information


end
disp(['After Iteration ' num2str(it) ': Top 3 Solutions = [' num2str(BestSols(1).Position(1)) ',
' num2str(BestSols(1).Position(2)) '], [' ...
        num2str(BestSols(2).Position(1)) ', ' num2str(BestSols(2).Position(2)) '], [' ...
        num2str(BestSols(3).Position(1)) ', ' num2str(BestSols(3).Position(2)) ']']);
%% Results
figure;
semilogy(BestCosts, 'LineWidth', 2);
xlabel('Iteration');
ylabel('Best Cost');
grid on;
function z = Sphere(x, y)
    %Objective Function x->no of staions at each location, y->no of points at each stations
    z = sum(x*y*50*(10^3))+sum((0.5*0.5^((x*y)+1))/(fact(((x*y)-1))*((x*y)-1)))/5.6;
```

```
end
function y = fact(n)
% This function calculates the factorial of a non-negative integer
 % Base case: factorial(0) = 1
 if n == 0
   y = 1;
   return;
 end
 j=1; i=1;
while i<=10
   j=j*i;
   i=i+1;
end
y=j;
return
end
```

Code Description:

This MATLAB code implements the Teaching-Learning-Based Optimization (TLBO) algorithm to solve an optimization problem defined by a cost function. Here's a breakdown of the code for the report purpose:

1. Problem Definition:

- The cost function (`Sphere`) represents the objective function to be minimized.
- `nVar` defines the number of unknown variables.
- `VarMin` and `VarMax` set the lower and upper bounds for the unknown variables.

2. TLBO Parameters:

- `MaxIt` specifies the maximum number of iterations.
- `nPop` defines the population size.

3. Initialization:

- An empty structure (`empty_individual`) is defined to store individuals' positions and costs.
- Population array (`pop`) and the array to store the best solutions (`BestSols`) are initialized.
- Random initial positions within the specified bounds are generated for each individual, and their costs are evaluated using the cost function.
- The best solutions (`BestSols`) are updated based on the initial population.

4. TLBO Main Loop:

- The main loop runs for a predefined number of iterations (`MaxIt`).
- In each iteration:
  - Population mean (`Mean`) is calculated.
  - A teacher individual is selected based on having the lowest cost in the population.
  - The teacher phase and learner phase are executed to update the population.
  - The best solutions are updated based on the new population.
  - The best cost of the current iteration is recorded.

5. Results Visualization:

- After completing all iterations, the top 3 solutions and their corresponding costs are displayed.

- A semilog plot is generated to visualize the evolution of the best cost over iterations.

6. Objective Function:

- The `Sphere` function represents the objective function to be minimized.
- It computes the objective value based on the number of stations (`x`) and the number of points at each station (`y`).

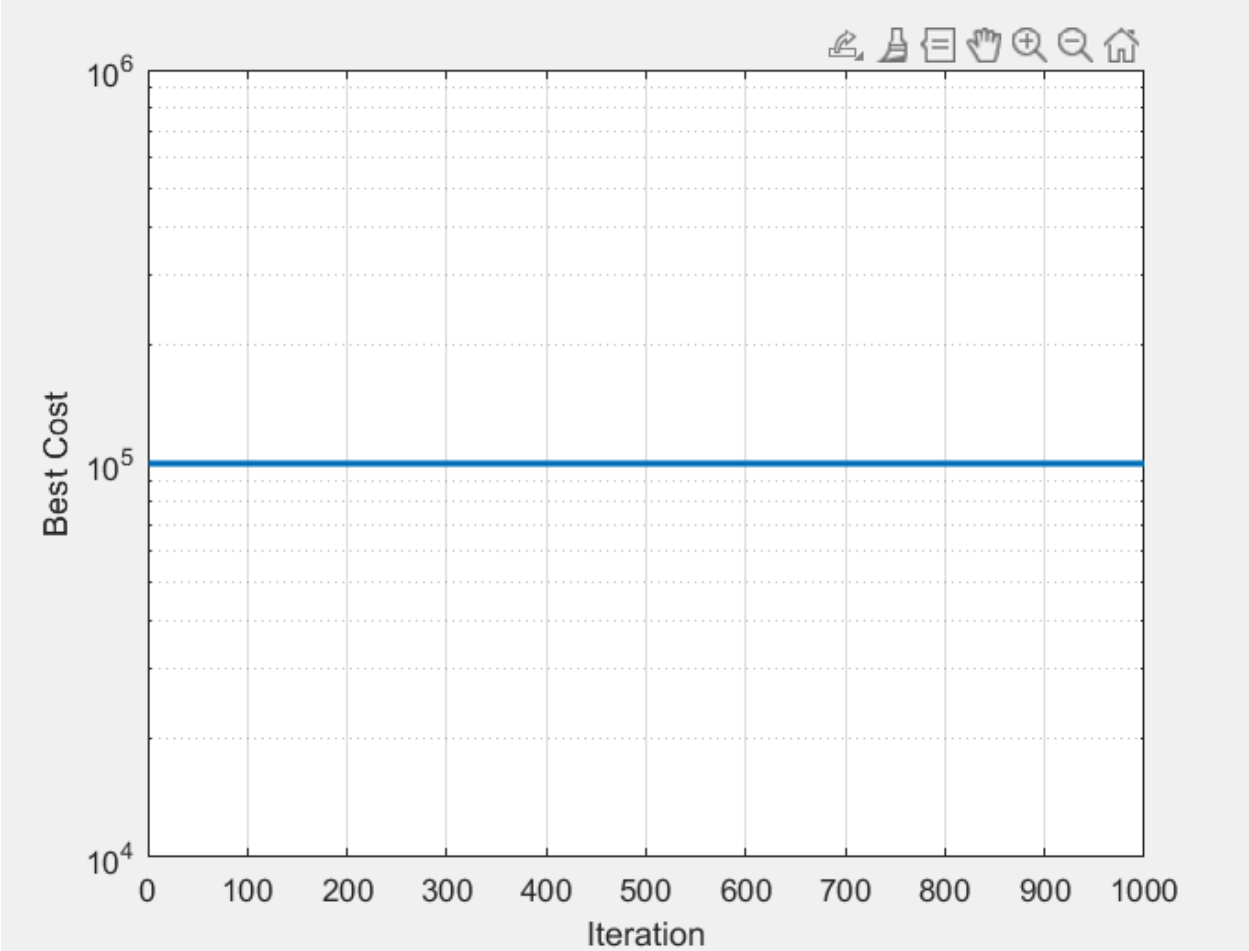
7. Factorial Function:

- The `fact` function calculates the factorial of a non-negative integer (`n`).
- It is used in the objective function to compute a term.

Overall, the code demonstrates the implementation of the TLBO algorithm to solve an optimization problem represented by a cost function. The algorithm iteratively updates the population to find the best solutions within the specified bounds.

Results

```
After Iteration 1000: Top 3 Solutions = [2, 1], [1, 2], [2, 1]
```



Best scores after each iteration

Explanation of Results:

Here in first solution [2,1] ,2 is number of charging stations at each candidate location and 1 is the number of charging points at each station and same goes for the other 2 solutions.Now we have selected location , number of charging stations at each candidate location and number of charging points at each station.

Future Explorations:

Our project successfully used a Teacher-Based Learning Optimization Algorithm to find optimal locations for EV charging stations, considering cost and wait time. Here's how we can build on this:

1. **Advanced Metrics: Include a Voltage and Power Index(compares the values of voltage and power before and after installing the stations) to optimize grid impact alongside cost and wait time.**

2. **Robust Optimization: Explore combining TBLOA with other algorithms like Chicken Swarm Optimization (CSO) for a more robust approach, similar to a referenced research paper.**

These advancements will lead to a more comprehensive and adaptable solution for optimizing EV charging station locations, promoting a more efficient and sustainable EV charging infrastructure.