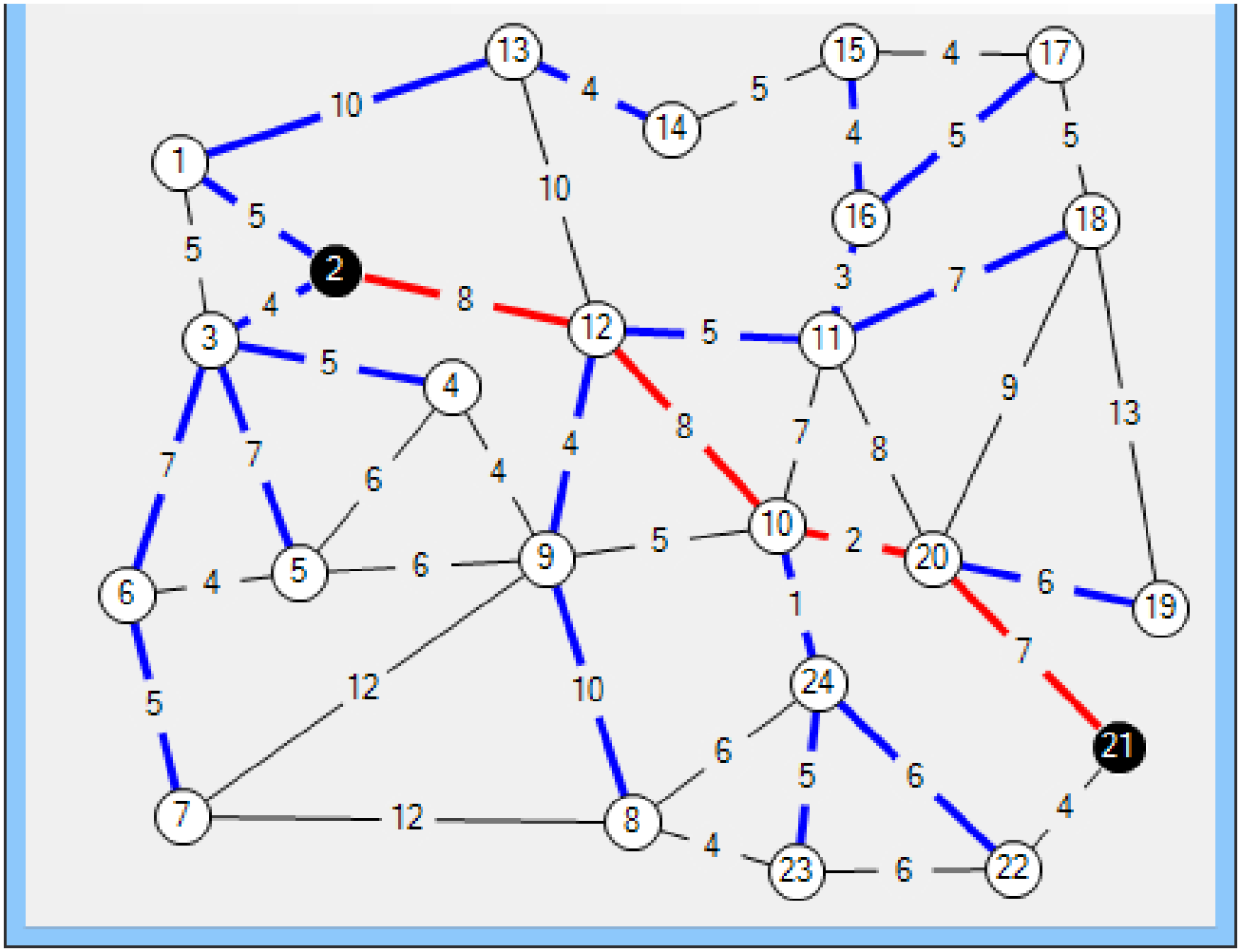
AIFA, Assignment - 1



3.Electric Vehicles

Group members

Pratyush Saha (18EE10039)

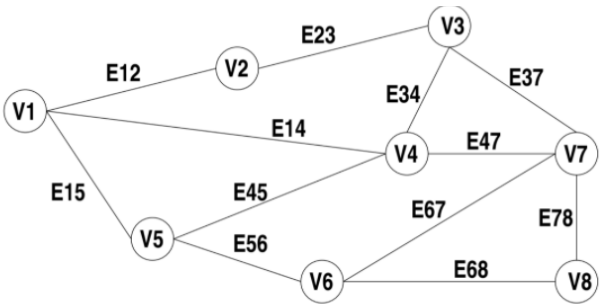
Abhinav Japesh (18EE30001)

Pradnya Anil Shinde (18EE10037)

Sunit Kumar Singha (18EE10064)

Problem Statement

Consider a city network where we need to route a set of electric vehicles which may require to be charged during its journey from some source to some destination. Let us assume that we have n cities (v1, v2, . . . , vn) and the distance between cities vi and vj be eij (if two cities are not connected directly then eij = ∞ and eij = eji). Assume that each city has a single charging station which can charge one EV at a time. Consider a set of k EVs namely P1, P2, . . . , Pk. For each EV the following information is provided -
(a) Sr - source node
(b) Dr - destination node
(c) Br - battery charge status initially
(d) cr - charging rate for battery at a charging station (energy per unit time)
(e) dr - discharging rate of battery while traveling (distance travel per unit charge)
(f) Mr - maximum battery capacity
(g) sr - average traveling speed (distance per unit time).
Assume that all vehicles start their journey at t = 0 and Pr reaches its destination at t = Tr. We
need to route all the vehicles from their respective sources to destinations such that max{Tr}
is minimized. You need to develop both optimal as well as heuristic algorithms.



Heuristic algorithm :

- Take No. of nodes, adjacency matrix of the graph, information of each EV (source node, destination node, battery charge status initially, charging rate for battery at a charging station (energy per unit time), discharging rate of battery while traveling (distance travel per unit charge), maximum battery capacity, average traveling speed (distance per unit time)) as input.
- Calculate shortest distances between every pair of vertices in a given graph by **Floyd Warshall Algorithm**
    - Initialize the solution matrix the same as the input graph matrix as a first step. Then update the solution matrix by considering all vertices as intermediate vertices. The idea is to one by one pick all vertices and update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When picking vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices. For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.
      1) k is not an intermediate vertex in the shortest path from i to j. Keep the value of dist[i][j] as it is.
      2) k is an intermediate vertex in the shortest path from i to j. Update the value of dist[i][j] as dist[i][k] + dist[k][j] if dist[i][j] > dist[i][k] + dist[k][j].

- We are using our heuristic value to expect a minimum time to reach its destination from its current node which is calculated from Floyd warshall and the speed for each car is the same.
- For each EV at each node calculate fn,gn,hn where hn is the minimum time from the current node to the destination gn is the total spent time while reaching the present node(traveling time + waiting time) and fn=gn+hn
- A state is defined as a list of all EVs, and in each state every EV will be at any particular node. So next state is defined as all possible combination of child nodes of present nodes of each EV i.e. if in present state 3 cars are at x1,y1,z1 and x2,x3 are child of x1, y2,y3 are child of y1 and z2 is child of z1
  so(x2,y2,z2),(x3,y2,z2),(x2,y3,z2),(x3,y3,z2) will be 4 new states.
- Apply A* algorithm on state spaces.
    - Create state-space containing all required information of each EV(current node, destination node, battery charge status initially, charging rate for battery at a charging station, discharging rate of battery while traveling, maximum battery capacity, average traveling speed)
    - Calculate f(state)=max f(n) among all cars in its present location
    - Apply A* algorithm:
        - Create an open list containing starting state space (i.e for each node current node=start node)
        - Select a curr_state on the open list with min f(state)
        - If curr_state is the goal state then terminate with f(state) value

    - Otherwise, put all possible states in the open list. Possible states will be all possible combinations of each car going to its next node.
    - Put curr_state in the closed list.
- Check conflicts at each node.
  Condition:
  if (car[i].gn==car[j].gn&&car[i].cur==car[j].cur&&i!=j)
  {
  prioritize the EV according to their fn i.e EV with maximum fn will charge first
  For each EV update gn=gn+charging time+waiting time
  }
- Charging condition at each node: Battery_status=min (Max_battery capacity, total charge required to reach destination from current node).

Optimal approach

- Take No. of nodes, adjacency matrix of the graph, information of each EV (source node, destination node, battery charge status initially, charging rate for battery at a charging station (energy per unit time), discharging rate of battery while traveling (distance travel per unit charge), maximum battery capacity, average traveling speed (distance per unit time)) as input.
- Created a class car, which contains all the parameters related to a car.
- The car object has a data element **“path array”** which contains the shortest path from the source node to the destination node. The **“min_time” and “charging_time required”** are defaultdicts, which store the minimum time required(without collision) and minimum charging time required(without any waiting) respectively for the car from its current node in the path to the destination node.
- A global 2-D matrix **“arrival time”** is maintained with row indices as nodes and column indices as “vehicles”. This matrix stores the expected arrival time of each car to a particular node without any conflict or waiting.
- In case of conflict, all the cars who are to be waited for charging a car with **“priority”** are considered and their **“min_time”** and **“arrival_time”** of those cars in their subsequent nodes of their optimal path are updated.
- For proper synchronization of events with respect to the “continuous” time, we maintain a **global time list of events**. The events are “expected arrival and departure time stamps” of each vehicle at each node. Then the global time list is sorted for inspecting events sequentially.

  Optimizing approach
    - We decided to route the cars according to their **shortest path individually** only.

- Conflict will occur at nodes when another car arrives while a car is already charging.
- In that case proper “scheduling” in charging is done to ensure that the “slowest car” or the car which takes the maximum time to reach the goal node from this current node (ideally) is not delayed much. This is done to ensure that the maximum time of all cars to reach their destination is minimized.
- For the “amount to be charged for a car at a node” we applied a greedy approach. A car is charged to the ***min(Max_battey capacity, minimum_charge_required[node]),*** at each node. Where “minimum_charge_required[node]” is the minimum charge required by the car to reach the destination from its current node. This is checked from the start of the journey only and it is done to avoid collisions in its remaining journey.

Scheduling approach

- When an event **“arrival”** is popped from the global event time list then the *car id and the node* at which it arrives is retrieved from the list.
- We push the arrived car to the “charging queue” of that node and then the scheduling function is called.
- This function iterates over all the cars present in the list and the best suitable car for charging is selected.
- That car is selected which minimizes the **metric - max(**`w.min_time[curr_node] + charge_time_left_list[x.index]`
- Here x is the selected car and “w” is the iterator over all the other cars.
- What it does basically is “if this car is selected for charging then how is the minimum time required for all the other cars to reach the destination from current node increased”. The car for which this maximum increment over all the other cars is minimized, is selected for **charging**
- Parameters of each car and the global event time list are updated by incorporating this additional **“waiting time”** to the departure times and the arrival times to the subsequent nodes in the journey.
- When an event **“departure”** occurs then that “car” is popped from that “node”.
- This scheduling algorithm ensures that in case of conflict the **“waiting time”** for the conflicting cars is minimum.