Artificial Intelligence: Foundations and Applications Project Report

Under Prof. PPC and Prof. Arijit Mondal

By:
1)Anil Yogi - 19MA20004,
2) Sakshi Dwivedi - 19MA20047
3) Yogesh Chawla - 19AG3FP02
4)Cherry Taly - 19IE10013

STATE or CONFIGURATION:

- Variables (Given)
    - Sr- source node
    - Dr- destination node
    - Br- battery charge status initially
    - cr- charging rate for battery at a charging station (energy per unit time)

- dr- discharging rate of battery while traveling (distance travel per unit charge)
- Mr- maximum battery capacity
- sr- average traveling speed (distance per unit time).

- State Variables (Assumed)
  - Current_node - representing the current-node for the respective EV.
  - Next_node - representing the next-node for the respective EV.
  - Bool node_free - whether the next node in the optimized path of the EV is free or it has another EV already charging at that node.
- Domain
  Node_free domain = {True , False}
  True - when the next node is free for that EV
  False - if it has another EV already charging at the next node

  Current_Node = {Starting Node, End Node, In-between Node}
  Next_Node = {End Node, In-between Node}

- Valid configuration for a single EV
  - When a vehicle is following its optimized path and it doesn't have any EV already charging on the next node. Then node_free is True.

- Valid configuration for whole network
  When node_free is True for each EV in the network i.e. for each EV there is no conflict.

STATE TRANSFORMATION RULES or MOVES:

- For each EV if curent_node is not equal to Dr, if Br is sufficient enough to cover the distance between the current node and the next node, it moves to the next node toward its optimized path (obtained using A* algorithm for heuristic solution and using Dijkstra for optimal solution). If it is not sufficient it charges at the current node just sufficient enough to move to the next node.

  ```
  for(each EV)
      if(current_node!=Dr)
          if(Br<(distance/Dr))
              Charge and move to the next node
          else
              Move to the next node
  ```

- If for an EV (say EVi) the current node has another EV (say EVj) already charging at it (that means out of EVi and EVj, EVj must have reached at that node first in terms of time and it started charging itself) and EVi needs to be charged at the current node, we call it a conflict then the EVi has two options -
  - Wait for EVj to finish charging itself and then charge itself at that node then move forward
  - Replan its path and see if 2nd best path to the goal gives less time then the total time to reach goal with addition of the waiting time at the current node.

  EVi explores both these ways and considers the way which gives minimum time out of them.

STATE SPACE or IMPLICIT GRAPH:

We are going to make a 2-D array in which Row represents EV and the column represents the Nodes. In each cell, if that particular node is busy (an EV is getting charged on that node) we are going to fill it with a tuple of values (time of arrival, time of departure). If that node is free (no EV is getting charged on that node) then we'll leave that cell empty.

The initial obvious choice was a 3-D graph but considering time as an axis we couldn't solve our problem as our problem revolves around float values of time as the charging time of an EV is constant.

This idea is based on our approach of algorithm that is if an EV (say EVi who is at node N1) finds another EV(say EVj who is at node N2) already charging on its future path node then it can either wait or replan and this 2-D array will calculate the time difference.

```
for EVi
    If ( (time (N1 to N2) + (time of departure of EVj - arrival time of EVi) + time from N2 to final node)  > time from 2nd best path if replanned form N1)
        Move to the next node and Charge
    else
        Replan the path from N1 to 2nd best path
```

INITIAL or START STATE(s), GOAL STATE(s):

Initial State: for(each EV) current_node = Sr (start node)
At the start, for each vehicle their current node is initialised to their starting node. Initially, the OPEN list contains the start Node s. CLOSED list is empty.

Goal State: for(each EV) current_node = Dr (end node)
At the end, for each vehicle the current node will be the destination node.

SOLUTION(s), COSTS:

Assume that all vehicles start their journey at t = 0 and EVi reaches its destination at time = Ti . We need to route all the vehicles from their respective sources to destinations such that max{Ti } is minimum.
Ti comprises of two components -
i) total travelling time required to cover the distance from start node (Sr) to goal node (Dr) (say T_{travel}).
ii) total charging time required to charge the battery of EVi at certain nodes inclusive of the waiting time as at times EVi has to wait for another EV to finish its charging before charging itself (if any). Let's call it $T_{charging.}$

$$\text{Total time cost} = T_{travel} + T_{charging}$$

We intend to minimise this total time cost for each vehicle.

SEARCH ALGORITHMS:

For an optimal solution, we used dijkstra.

Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph.

Working Principle/Approach:
Dijkstra's Algorithm basically starts at the node that you choose (the source node) and it analyzes the graph to find the shortest path between that node and all the other nodes in the graph.
The algorithm keeps track of the currently known shortest distance from each node to the source node and it updates these values if it finds a shorter path.
Once the algorithm has found the shortest path between the source node and another node, that node is marked as "visited" and added to the path.
The process continues until all the nodes in the graph have been added to the path. This way, we have a path that connects the source node to all other nodes following the shortest path possible to reach each node.

Complexity:
Time Complexity: O(E Log V)
where E denotes the number of edges and V denotes the number of vertices.
Space Complexity: O(V)

Applications:
This algorithm is used in GPS devices to find the shortest path between the current location and the destination. It has broad applications in industry, especially in domains that require modeling networks.

For heuristic solution, we used A* algorithm
A * algorithm is a searching algorithm that searches for the shortest path between the initial and the final state. It is one of the best and popular techniques used in path-finding and graph traversals. It is an advanced BFS algorithm that searches for shorter paths first rather than the longer paths.

Working Principle/Approach:

```
1.  We start off by initialising the opening list.
2.  Now, initialize the closed list by putting the starting node on the open
list.
3.  While the open list is not empty,
    a) find the node with the least f on
       the open list, call it "q"
    b) pop q off the open list
    c) generate q's 8 successors and set their
       parents to q
    d) for each successor
        i) if successor is the goal, stop search
          successor.g = q.g + distance between
                              successor and q
          successor.h = distance from goal to
          successor (This can be done using many
          ways, we will discuss three heuristics-
          Manhattan, Diagonal and Euclidean
          Heuristics)
          successor.f = successor.g + successor.h
        ii) if a node with the same position as
            successor is in the OPEN list which has a
           lower f than successor, skip this successor
        iii) if a node with the same position as
            successor  is in the CLOSED list which has
            a lower f than successor, skip this successor
            otherwise, add  the node to the open list
     end (for loop)
    e) Now, push q on the closed list ;
    end (while loop)
```

Applications:

A* algorithm is used in various applications, for example:
Used as maps => used to calculate the shortest distance between the source (initial state) and the destination (final state).
A* algorithm is very useful in graph traversals as well.