Electric Vehicle Path Optimisation Problem

- Dhruv Rai 19QM30002
- Param Chauhan 19IM30007
- Abhishek Mahalunge 19IM30003
- Manas Dedhiya 19MI3EP10

Contents:

Introduction

Time minimization has been at the centre of any optimization problem in almost every sector encompassing real life problems.Transportation sector is one of the major domains that appertains to time minimization with some inevitable constraints at some times. The foremost thing that strikes one head to minimize time travel is to take the shortest path which is clearly visible.But remember, the shortest path is not always the quickest path! Because appearances are often deceptive. Talking specifically about our problem, the complexity of the constraints is the main impediment that hinders the notion of following the shortest path to be the quickest. Hence its not just the path length, but a meticulous & a mathematical analysis of different aspects like energy optimaling , priority preferences at coinciding times based upon battery and the path left to be covered ,etc. This paper shows how these challenges can be met within the framework of A* search. We show how the specific domain gives rise to a consistent heuristic function yielding an O(n2) routing algorithm. Moreover, we show how battery constraints can be treated by dynamically adapting edge costs and hence can be handled in the same way as parameters given at query time, without increasing run-time complexity. Experimental results with real road networks and vehicle data demonstrate the advantages of our solution.

Energy-optimal routing for electric vehicles creates novel algorithmic challenges, as simply understanding edge costs as energy values and applying standard algorithms does not work. First, edge costs can be negative due to recuperation, excluding Dijkstra-like algorithms. Second, edge costs may depend on parameters such as vehicle weight only known at query time, ruling out existing preprocessing techniques. Third, considering battery capacity limitations implies that the cost of a path is no longer just the sum of its edge costs. We show how battery constraints can be treated by dynamically adapting edge costs and hence can be handled in the same way as parameters given at query time, without increasing run-time complexity. Experimental results with real road networks and vehicle data demonstrate the advantages of our solution.

To cite this with an example, if you have two routes to your destination one of the them gives you the shorter path than the other and hence you go for the same

.

Theory

- Depth First Branch & Bound (DFBB) Algorithm

The idea of a branch-and-bound search is to maintain the lowest-cost path to a goal found so far, and its cost. Suppose this cost is *bound*. If the search encounters a path *p* such that *cost(p)+h(p)* ≥ *bound*, path *p* can be pruned. If a non-pruned path to a goal is found, it must be better than the previous best path. This new solution is remembered and *bound* is set to the cost of this new solution. It then keeps searching for a better solution.

Branch-and-bound search generates a sequence of ever-improving solutions. Once it has found a solution, it can keep improving it. Branch-and-bound search is typically used with depth-first search, where the space saving of the depth-first search can be achieved. It can be implemented similarly to depth-bounded search, but where the bound is in terms of path cost and reduces as shorter paths are found. The algorithm remembers the lowest-cost path found and returns this path when the search finishes.

The algorithm for Depth First Branch & Bound is shown below :

```
1: **Procedure** DFBranchAndBound($G,s,goal,h,bound_0$)
2:     **Inputs**
3:         $G$: graph with nodes $N$ and arcs $A$
4:         $s$: start node
5:         $goal$: Boolean function on nodes
6:         $h$: heuristic function on nodes
7:         $bound_0$: initial depth bound (can be $\infty$ if not specified)
8:     **Output**
9:         a least-cost path from $s$ to a goal node if there is a solution
   with cost less than $bound_0$
10:         or $\perp$ if there is no solution with cost less than $bound_0$
11:     **Local**
12:         $best\_path$: path or $\perp$
13:         $bound$: non-negative real
14:         **Procedure** cbsearch($\langle n_0,\ldots,n_k \rangle$)
15:             **if** ($cost(\langle n_0,\ldots,n_k \rangle)+h(n_k) < bound$) **then**
16:             **if** ($goal(n_k)$) **then**
17:                 $best\_path \leftarrow \langle n_0,\ldots,n_k \rangle$
18:                 $bound \leftarrow cost(\langle n_0,\ldots,n_k \rangle)$
19:             **else**
20:                 **for each** arc $\langle n_k,n \rangle \in A$ **do**
21:                     $cbsearch(\langle n_0,\ldots,n_k,n \rangle)$
22:     $best\_path \leftarrow \perp$
23:     $bound \leftarrow bound_0$
24:     $cbsearch(\langle s \rangle)$
25:     **return** $best\_path$
```

The internal procedure *cbsearch*, for cost-bounded search, uses the global variables to provide information to the main procedure.

Initially, *bound* can be set to infinity, but it is often useful to set it to an overestimate, *$bound_0$*, of the path cost of an optimal solution. This algorithm will return an optimal solution - a least-cost path from the start node to a goal node - if there is a solution with cost less than the initial bound *$bound_0$*.

If the initial bound is slightly above the cost of a lowest-cost path, this algorithm can find an optimal path expanding no more arcs than A^* search. This happens when the initial bound is such that the algorithm prunes any path that has a higher cost than a lowest-cost path; once it has found a path to the goal, it only explores paths whose f-value is lower than the path found. These are exactly the paths that A^* explores when it finds one solution.

If it returns ⊥ when $bound_0=\infty$, there are no solutions. If it returns ⊥ when *$bound_0$* is some finite value, it means no solution exists with cost less than *$bound_0$*. This algorithm can be combined with iterative deepening to increase the bound until either a solution is found or it can be shown there is no solution.

- ## Best First Search (A*) Algorithm

The most widely known form of best-first search is called A∗ A search (pronounced "A-star ∗ SEARCH search"). It evaluates nodes by combining g(n), the cost to reach the node, and h(n), the cost to get from the node to the goal: f(n) = g(n) + h(n).

The steps for A* Algorithm is given below :

Each Node n in the algorithm has a cost g(n) and a heuristic estimate h(n), f(n) = g(n) + h(n). Assume all c(n, m) > 0

1. [Initialize] Initially the OPEN List contains the Start Node s.

g(s) = 0,

f(s) = h(s)

CLOSED List is Empty.

2. [Select] Select the Node n on the OPEN List with minimum f(n). If OPEN is empty, Terminate with Failure

3. [Goal Test, Terminate] If n is Goal, then Terminate with Success and path from s to n.

4. [Expand]

   a) Generate the successors n_1, n_2, … n_k of node n, based on the State Transformation Rules

   b) Put n in LIST CLOSED

c) For each n_i, not already in OPEN or CLOSED List, compute a) $g(n_i) = g(n) + c(n, n_i)$, $f(n_i) = g(n_i) + h(n_i)$, Put n_i in the OPEN List

d) For each n_i already in OPEN, if $g(n_i) > g(n) + c(n, n_i)$, then revise costs as:

$g(n_i) = g(n) + c(n, n_i)$

$f(n_i) = g(n_i) + h(n_i)$

5. [Continue] Go to Step 2

Since g(n) gives the path cost from the start node to node n, and h(n) is the estimated cost of the cheapest path from n to the goal, we have f(n) = estimated cost of the cheapest solution through n.

Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of g(n) + h(n). It turns out that this strategy is more than just reasonable: provided that the heuristic function h(n) satisfies certain conditions, A∗ search is both complete and optimal. The algorithm is identical to UNIFORM-COST-SEARCH except that A∗ uses g + h instead of g.

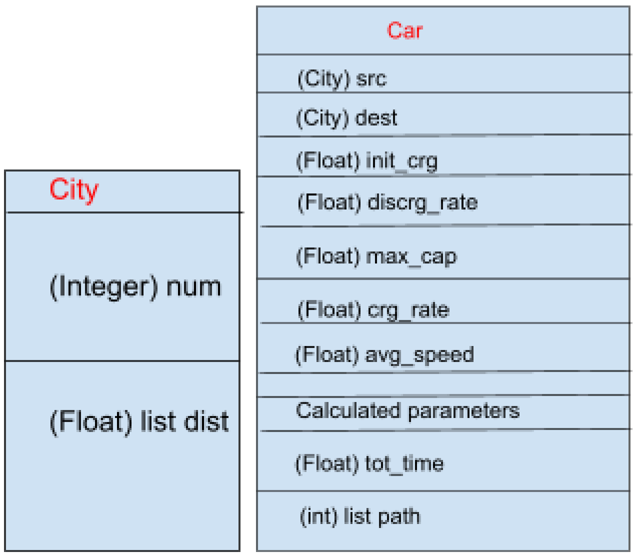
The first condition we require for optimality is that h(n) be an admissible heuristic. An Admissible Heuristic is one that never overestimates the cost to reach the goal. Because g(n) is the actual cost to reach n along the current path, and f(n) = g(n) + h(n), we have as an immediate consequence that f(n) never overestimates the true cost of a solution along the current path through n. Admissible heuristics are by nature optimistic because they think the cost of solving the problem is less than it actually is.

A second, slightly stronger condition called consistency (or sometimes monotonicity) MONOTONICITY is required only for applications of A∗ to graph search.9 A heuristic h(n) is consistent if, for every node n and every successor n of n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n plus the estimated cost of reaching the goal from

n : h(n) ≤ c(n, a, n) + h(n).

Explanation of the code:

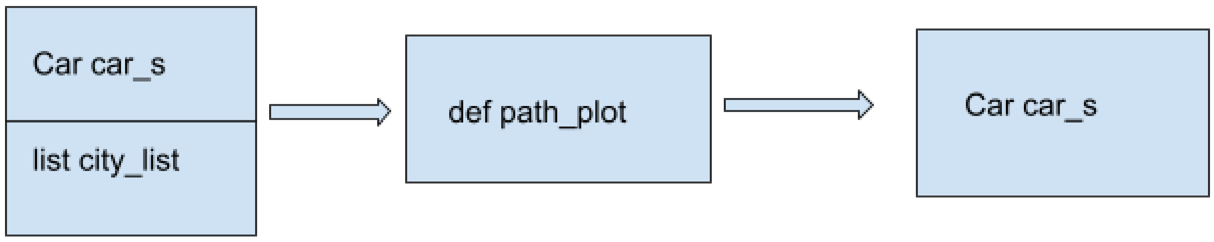
Global Classes



There are two types of data types defined, a city and a car.

The user inputs a list of cars and cities and the program finds the optimal path for all the cars.

The function used is path_plot



The output variable car_s has now calculated parameters tot_time (total time taken in journey) and path which is a list of cities that lie on the path of the electric vehicle.

<u>Input</u>

Input is taken from the user in the terminal using python input function. The input values are fed into custom defined classes city and car which then are stored in lists
After input, path_plot function is called for each car separately.