ElectroWay: Smart Routing for Electric Car Charging

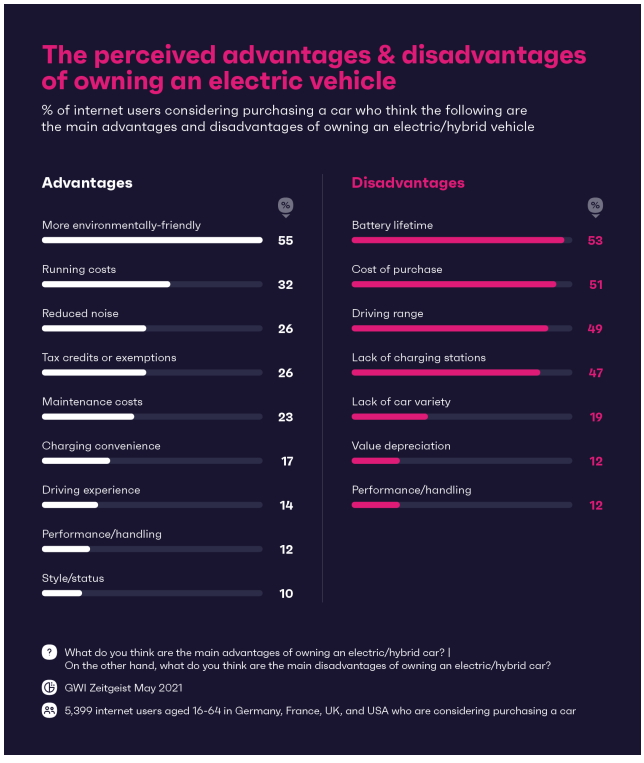
Dragos Tudorache
"Alexandru Ioan Cuza" University
Faculty of Computer Science
Iasi, Romania
Email: dragoscosmintudorache@gmail.com

Cristian Simionescu
"Alexandru Ioan Cuza" University
Faculty of Computer Science
Iasi, Romania
Email: cristian@nexusmedia.ro

***Abstract*—Electric vehicles present us with a unique set of constraints and considerations when creating routing algorithms. In this paper, we present one such real-world routing algorithm and the specific issues that need to be taken into account when designing such a system. We integrated this algorithm into an Android mobile application named ElectroWay. The code is available on GitHub https://github.com/QuothR/ElectroWay.**

I. Introduction

The world is in the process of transitioning to more sustainable energy production and consumption methods. As part of this process, we are seeing the beginning of an exponential S curve in the adoption of electric vehicles (EV), the issue of creating long trip routes with the special characteristics of electric cars arises.



A recent GWI[1] report which included a survey of more than 5000 people who were considering purchasing a car, shows that 2 of the main concerns people have when switching to an EV are the driving range and lack of charging infrastructure. Since EV adoption has only recently started to gain traction, the general ecosystem is still seeing high variance in specifications and standards. The existence of multiple types of charging connectors, car charge capacity, charging station capacities, and wildly different car ranges, introduces complexity for EV car owners or prospective car owners when having to plan trips or daily commutes. This task can be completed by humans fairly easily but it requires them to invest a sizable amount of time to search and compare various charging locations along their route to estimate and compare which option would lead to the least total route time. It might often be the case that simple heuristics used by drivers such as generating the shortest route using Google Maps and then looking for the closest stations along their path might lead to drastically worse results. For example, if the stations found closest to the driver's route might be 'slower' 50kW where they need to wait an hour while a slightly further station which adds only 30 minutes extra driving time, with 250kW charging which will require them to wait only 15 minutes, resulting in an overall shorter trip.

Another consideration especially for people interesting in buying an EV or who have just recently bought one is the worry of running out of power before reaching a charger. While charging infrastructure is seeing a lot of growth and seems to only be accelerating, the fact that electric cars have smaller ranges and their efficiency can be severely impacted by many factors such as temperature, speed, wind, rain, and snow or the use of internal systems such as the AC, it is often the case that the manufacturer's estimated range is inaccurate. This can further complicate the process of planning a route for drivers since if they don't know of these implications they might overestimate the distance they can reach before running out of power.

To address this problem of optimizing the planning of routes specifically for electric cars, which tries to take into account various real-world factors we propose simple, efficient, and real-world usable routing algorithms which leverage existing maps and routing services to offer a routing service at low to no cost. We packaged this algorithm in an

[1]https://blog.gwi.com/chart-of-the-week/electric-vehicles/

Android mobile application named ElectroWay which offers the additional feature of letting users add their own charging stations in the system. The code is available on GitHub https://github.com/QuothR/ElectroWay.

II. Proposed Algorithm

The application is given a car object containing relevant data related to:

- the current electric energy supply in kilowatt-hours (kWh).
- maximum electric energy supply, in kilowatt-hours (kWh), which can be stored in the vehicle's battery.
- charging speed in kilowatt-hours (kWh).
- maximum speed of the vehicle in km/hour.
- the amount of power, in kilowatts (kW), consumed by auxiliary systems.
- the speed-dependent component of consumption called **Constant Speed Consumption**. Provided as an unordered list of speed/consumption-rate pairs.
- a list of plug types to ensure compatibility with charging stations.

It will also receive the geographical coordinates of two points representing the starting point(**A**) and the ending point(**B**).

In case of the need to recharge the car battery, it will have access to a collection containing charging station objects. An instance of this object contains:

- the geographical coordinates of a point, representing the location of the station.
- a list of charging point objects. A charging point object contains:
  - the connector type.
  - the price, per kilowatt-hours (kWh).
  - charging speed in kilowatt-hours (kWh).

To determine the route between two points we use **TomTom**, a mapping API that generates routes without considering the collection of stations or the fact that the car may not be able to follow the generated route due to insufficient energy level. In addition to the generated route (which is represented by a list of geographical coordinates), the API also provides useful information about route length, travel time, and consumption.

A. Data processing

Initially, the speed-dependent component of consumption has general values, but these values may vary depending on external factors such as temperature, wind, and road condition. In order to determine more realistic values that correspond to the area through which the car will pass, the following steps will be performed:

1) request an initial route from TomTom API, a route from point **A** to point **B**, without taking into account the car. This initial route will not be very different(when it comes to the areas through which the car will pass) from the one we will calculate, as such we poll weather data along the initial TomTom route since the weather conditions in those regions will be nearly identical to our route.
2) use **OpenWeather** API to find out details about the weather conditions for a certain amount of points(5-10%) from the initial route. Not all the points, because a request to an API is quite expensive in terms of time complexity. Usually, a route has many points(e.g. a 300 km route has ≈2600 points), and making a request for each would be too slow.
3) for the uniformly selected points, the following statistics will be extracted: temperature in degrees Celsius, wind power(in mps) and direction, and road conditions such as rain and snow.
4) calculates the average temperature, the average wind power, the average wind direction, and how many areas with bad weather conditions are.
5) calculates the general direction of the route (the bearing) which we approximate by only using the start and destination points. Necessary to see how the wind will affect consumption.

Algorithm 1 Get Bearing

function BEARING_ANGLE(lat1, lon1, lat2, lon2)

$lat1 \leftarrow \frac{lat1 * 3.14}{180}$

$lon1 \leftarrow \frac{lon1 * 3.14}{180}$

$lat2 \leftarrow \frac{lat2 * 3.14}{180}$

$lon2 \leftarrow \frac{lon2 * 3.14}{180}$

$y \leftarrow sin(lon2 - lon1) * cos(lat2)$

$x \leftarrow cos(lat1) * sin(lat2) - sin(lat1) * cos(lat2) * cos(lon2 - lon1)$

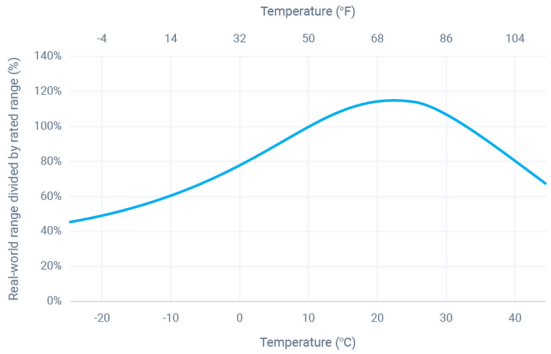
$teta \leftarrow atan(\frac{y}{x})$

// Convert to degrees.

$bearing = (\frac{teta * 180}{3.14} + 360)\%360$

return $bearing$

6) modify Constant Speed Consumption using the calculated parameters:
   a) **Temperature**



According to geotab.com:

- If the average temperature calculated earlier is lower than -10 degrees Celsius, then the consumption will be increased by 30%.
- If it is greater than 40 degrees Celsius or between -10 and 0 degrees Celsius, then the consumption will be increased by 20%.
- If it is between 30 and 40 degrees Celsius or between 0 and 10 degrees Celsius, then the consumption will be increased by 10%.

b) **Wind**
Headwinds and crosswinds significantly increase aerodynamic drag, which results in reduced energy efficiency. For every 10 mph of headwind or crosswind, energy efficiency is reduced by as much as 13%.
Having the bearing of the wind and the bearing of the car, we easily determine how the consumption will change depending on the earlier calculated wind power.

c) **Road condition**
According to freightliner.com:
Rain, snow, or slush on the road increases the vehicle's rolling resistance because, in addition to moving the vehicle, the tires must also push their way through the precipitation on the roadway. The increased rolling resistance and drive-train friction in just a light rain can increase the consumption by 0.2 to 0.3 mpg.
Having the percentage of areas with bad conditions, we calculate the new consumption by adding 0.25 mpg multiplied by the percentage of bad condition locations we queried.

B. The heuristic

Given that the application is facing a real-world problem, large input data and long response times from external APIs, it will use a heuristic approach in order to produce a satisfactory solution in a feasible amount of time.

There are two cases:

1) the car can reach the final destination from its current location without having to charge. In this case, the route offered by TomTom along with the information related to it will be returned unchanged.
2) the car cannot reach the final destination with the current energy level. In this case, an intermediate point will be sought, a point at which the car will stop to recharge.

At each step it will check if the destination can be reached, by making a request to TomTom. If it cannot be reached, a convenient station(a reachable station) will be selected, convenient in terms of time and distance from the current location. This station will become the new current point and the whole process will be repeated until the car is at a point from which it can reach the final destination. As these steps are covered, the portions of the route will be retained, along with significant details such as the time required to complete these parts of the route, the length, the level of energy needed to get to the next point, the price of recharging the car.

Initially, we will request an initial route without considering the car. The stations as close as possible to this initial route will be selected, so that the calculated route is as close as possible to this initial route.

Algorithm 2 Recursive Binary Search

```
function BINARY_SEARCH(start,end, depth)
    if reachable(start, end) then
        return end
    if start == end or depth > 10 then
        return start
    i ← (start + end) / 2
    if reachable(start, i) then
        return binary_search(i, end, depth + 1)
    return binary_search(start, i, depth + 1)
```

Therefore, we are looking for a point that can be reached, a point on the initial route as far away as possible. The generated route is an array of geographical coordinates. A recursive binary search will be performed on that list, see Algorithm 2. Successive requests will be made to TomTom to check if the geographic point in the current index is reachable. If the current index is a geographic point that is reachable then we will search at the top part of the list, otherwise, we continue the search with the bottom part of the list. Because the requests are time-consuming, we will limit the depth of the recursion. When the preset depth has been exceeded and an accessible point has been found, a point different from the current location, then it will be returned.

In the next stage, all the stations that the car can reach will be extracted. A TreeMap will be created in which each station will have a corresponding total time and distance tuple associated with it, see Fig. 1.

$totalTime = a_1 + a_2 + a_3$

a_1 = the time required to get from the current point to the point determined in the previous stage.

a_2 = the time required to get from the point determined in the previous stage to the final destination.

a_3 = the time required to recharge the car. We decided to consider this because, depending on the length of the route, it can represent a big part of $totalTime$. This was inspired by the fact that the current software versions of Volkswagen's ID3 and ID4 electric cars, the included charger routing algorithm doesn't seem to take this into account, leading to the strange situation where the car would take the driver to a slow charger where they have to wait multiple hours instead of a fast charger if the first one was closer.

Distance = the distance from the current point to the point determined in the previous stage.

This TreeMap will be sorted in ascending order by $totalTime$. This will give us a list of the most convenient stations in terms of time. We name this station the $convenientStation$.

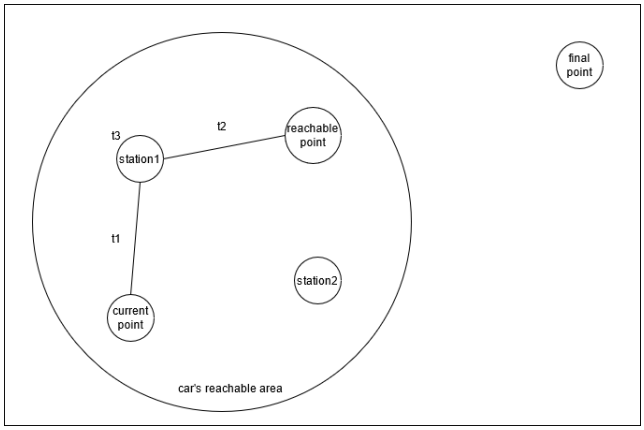


Fig. 1. Car reachable area

Because there may be a station that is most convenient in terms of time but that is too close to the current point, in this way stopping too early without the need to, we need to select stations which are further away but that don't increase the total time too drastically, this way make sure we don't stop too often. This new station should not have a much larger $totalTime$ than the $convenientStation$. In order to do that, we sort the original TreeMap in descending order by $distance$. We will iterate through it and the first station whose $totalTime$ is less than 10% higher than $convenientStation$ in terms of $totalTime$ will be returned. If no other station meets this criterion, the algorithm will still return $convenientStation$ since it has a 0% difference when compared to itself. After determining the station at which the car will stop to recharge, the location of this station will become the new current point and the whole process will be repeated.

III. Other approach

We also developed a similar algorithm initially which had multiple flaws that inspired us to develop the method we first presented. The input data is the same input from the Proposed Algorithm. We will try to build a graph in which there is a path from the starting point to the ending point. The initial graph is $G = (\{startingPoint\}, \emptyset)$. A vertex from G can be the starting point, the ending point, any station from the initial collection. For each vertex, we will keep the shortest current path to it.

While it is not possible to reach the final destination from any leaf, we will execute the following procedure:

For every leaf from G do:

1) determine all the stations that can be reached from the current leaf, along with significant information such as: distance traveled, route duration, the energy required to reach that station.
2) for each station determined in the previous step, we will check if it is already in $V(G)$.
   - if it is, then check if the route from the current leaf to the current station is shorter than the current one and update it if so.
   - if it is not then add the station to $V(G)$ with the route from the current leaf to the current station.

When the loop ends, in the final destination vertex, a route from start to finish with the required stopping points will be stored.

Problems with this approach:

- too many requests to TomTom $\Longrightarrow$ the algorithm is too slow.
- the calculated route may be different from an initially generated route, so we cannot take external factors into account.

An option we looked into to reduce the number of API calls made by this algorithm was to ignore stations that seem to be further away from the destination. We found that it can cause the user to get stuck by ignoring options that need to take longer routes in order to keep charged up. An example of this can be seen in Figure 2. Even though A3 seems further away from B than A1 and A2, it would be a better decision to choose it, because we cannot go through the blue surface(assuming it is an obstacle).

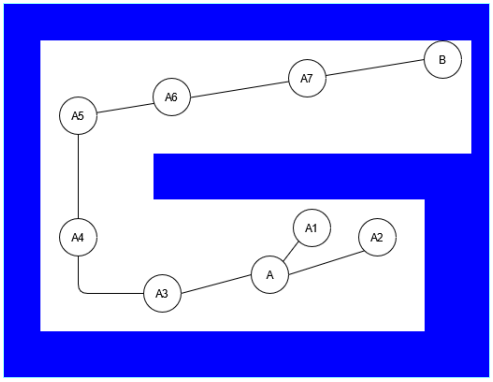


Fig. 2. Particular Case

IV. Conclusion

Further work needs to be done to test this algorithm against real-world data in order to calibrate the various coefficients used to factor in external factors. Another improvement would come from integrating the ElectroWay application with existing charging station networks in order to populate the database with more charging points so that we can create realistic testing scenarios.

The proposed algorithm is capable of generating good routes and its low API call costs also made it sufficiently fast and implementable for free with the current API service supplier offerings.

Acknowledgment

Thanks to the work of group A4 from the Faculty of Computer Science, Iasi who developed Electroway, we were able to expose the routing algorithm in this research paper.