Preface

The following parts of this thesis have been taken from our previous semester project "Skyline Queries Framework for Electric Vehicles" with authors myself - Dimitar Shkodrov, Ion-Anastasiu Sanporean and Paulius Galinauskas:

Chapter 3.3 - Skylines - the whole paragraph

Chapter 3.4 - Previous work - here I summarize the main topics, discussed in the previous paper. Improvements in text, examples given are the same.

Chapter 4.2 - some minor improvements were done in text and the pseudocode (see algorithms 2 and 3)

Chapter 4.3 - some minor improvements were done in text and the pseudocode (see algorithms 4, 5 and 6)

Creating a framework for analyzing historical travel data of electric vehicles and identifying heavily used parts of a road network using skyline queries

Dimitar Shkodrov
Aalborg University
Aalborg, Denmark

ABSTRACT

Electrical vehicles have a limited autonomy of movement based on their battery capacity. This means that for travels, larger than the said autonomy a charging route must be followed. This paper proposes a framework to help us analyze historical traveling data of electric vehicles. The charging routes that are inconvenient (have large deviation in time or distance) and important (are frequently present in the data set, referred to as support in the paper) are found using Skyline queries. Since a real dataset of EV travel data is unavailable, a synthetic one must be generated. This is done by combining a number of classical routing algorithms with a variety of approaches to recreate real-world phenomena, which might affect the construction of an EV's charging route. Furthermore, since identifying the routes with high support does not give the full picture of what's going on in the road network, one final thing the paper proposes is a way to identify parts of the road network which are the most heavily used by the vehicles. Such parts of the road network have higher support than all the routes that are in the skyline queries but are not independent routes by themselves, so it's hard to identify them straight away. The algorithms showcased in the paper use a variety of well established methods to achieve their goals - dynamic programming, usage of heuristics, derivatives of classical routing algorithms and so on. The end result is a framework that not only creates synthetic data sets of electrical vehicles' charging routes but also gives us the right tools to help us analyze them and this could be used as s cornerstone for future work in the field.

1. INTRODUCTION

Humanity will have a lot of problems to deal with in the coming years - climate change, the depletion of non-renewable resources of oil and natural gas just to name a few. Such issues put in serious jeopardy our current way of living, so in recent years we've seen a huge advancement in technologies that aim to make us more independent from the usage of fossil fuels. One such technology is the electric vehicle - a rapidly developing novelty which promises to replace conventional cars in the near future. With the increasing demand of various types of electric vehicles, covering the needs of both regular citizens and businesses, there is also an ongoing process to optimize their usage - new types of batteries are being developed, more and more charging stations are installed on road networks, various ways of conserving the car's energy are also being tested.

An electric vehicle's range (refered to as autonomy in the paper) is restricted by the battery capacity. Reaching destinations way out of the car's original autonomy may require numerous recharging operations. The route a car follows while recharging one of more time on the way is called a charging route. Of course, an important factor here is the coverage of charging stations across the road network (see *fig. 7*). The denser the coverage, the more variations for creation of different types of charging routes there are - choosing the shortest, fastest or the most economical route may be of key importance to providing the best traveling experience. Since the era of electric vehicles is still at its beginning, proper collections of real data about the journeys of EVs is still not available. Developing a framework to help analyze the data can be created in advance though, since EV travel data can be simulated with a signifficantly high degree of accuracy. Different real world conditions may be taken into consideration and simulated and although there is not perfect substitute for genuine EV travel data, synthetic data can give us enough valuable insight to at least identify and try to tackle some problems while still in the lab and not in the field.

What this paper proposes is a framework for generating synthetic travel data for electric vehicles, collecting the data and analyzing it - trying to identify the most frequently used charging routes between a starting and an ending point (the one that have high support value). Since cars often need to make detours from the shortest possible route from source to destination ,in order to get to a charging station, this deviation in distance and time is also measured. The framework also provides some tools to identify the most heavily used parts of the road network that have higher support than any independent charging route but are not part of the synthetic data set.

The paper is organized as follows: first we introduce the problem definition in the next chapter, then in chapter 3 we will have a glance at some related work done in the field. Chapter 4 is dedicated to introducing some novelties - a new way for generating synthetic data using different route discovery algorithms, finding the shortest routes from source to destination regarding distance or time, some real world factors that may influence the construction of a charging route are also taken into consideration. In chapter 5, we shall explore a method for identifying the most heavily used parts of the road network - road segments that have higher support than any route in the synthetic data set but are not independent routes themselves. An implementation has been done to show how the proposed framework handles

and the results of the experiments and tests done is also showcased in chapter 6. The paper ends with conclusions and possible future improvements.

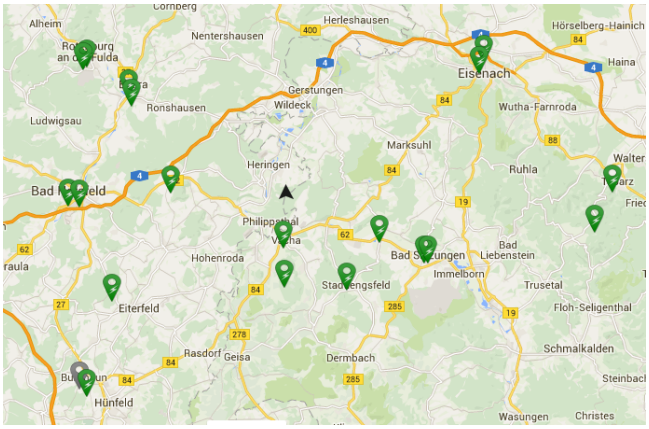

Figure 1: Illustration of charging stations placement

2. PROBLEM DEFINITION

As previously implied, the model, proposed in this paper can be conditionally divided into two parts. Since we don't have authentic historical travel data for electrical vehicles a module for generating synthetic travel data has been developed. Another module provides tools to analyze this data - finding the most frequently used independent charging routes and also identifying the most frequently used parts of the road network that are not part of the independent charging routes data set.

In order to present a detailed problem definition we need to consider the following problem settings: the road network is represented as a directed connected graph $G = (V, E)$, where V denotes the set of vertices and E denotes the set of edges in G. There exists a charging stations set C_s and every charging station c in C_s is included in V ($c \in C_s \subset V$). Furthermore, every edge $e \in E$ is associated with a positive weight e_L. Edges have different properties, so various aspects of weight are associated with the same edge. In other words, the same edge has several different weights, depending on what we need to know about it - the distance between the end nodes, defining the edge; the time it takes to travel through that edge (we calculate that knowing the free-flow speed limit for the edge and its length); the power the car needs to use to travel through that edge (also calculated, this value is dependent on edge length, allowed speed and battery discharge rate, see fig 2.)

EV's **autonomy** *(aut)* is a parameter which defines the possible range of an electrical vehicle, i.e. the number of kilometers a car can travel without charging its battery. In the previous model (see Previous work), the autonomy was a specifically set number of kilometers an EV could travel. Since one of the goals of the model, proposed in this paper is to generate highly realistic synthetic data, the autonomy parameter is no longer measured in kilometers but in Kilowatts hour (KWh) instead. This, at a glance minor improvement, actually makes a huge difference when it comes to generating synthetic data on different types of maps comprised of road networks of different types and density. Since an EV's battery discharge rate is hugely influenced by the free flow speed of the vehicle, this reflects on the car's autonomy. Speeds that are too high or too low can greatly decrease the battery's charge and thus force the car to recharge more often which may lead to different structures of charging routes. Given that we know the length of the edges a car travels through, the free flow speed of the car on this edge and the discharge rate at this speed we can easily calculate how much energy a car needs to travel through the edge. Furthermore, the model provides formulas for calculating the losses in energy a car might experience due to its aerodynamics and friction with air at different speeds (chapter 4). In the following chapters we will further investigate the autonomy.

Every route, referred to as a charging route $R_c(s, d)$ in the historical data set is an ordered sequence of vertices:

$$R_c(s, d) = (s, v_1, v_2, \ldots, d)$$

where s and d represent the source and destination of the car's journey. In order for a route to be classified as a charging route it has to contain at least one charging station $c \in C_s$. The length of a charging route is denoted as:

$$d_r(R_c) = \sum_{e \in E} e_L(e)$$

Each charging route R_c is then juxtaposed with the shortest possible path R_s from source V_s to destination V_d. Since the cars often have to make a detour in order to charge, we end up with a positive weight difference between the charging route and the shortest possible route, which we are going to note as **deviation**. More formally, we can represent is as:

$$dev(R_c) = d_r(R_c) - d_r(R_s)$$

Another attribute of a charging route - support is also computed.

Support, denoted $sup(R_c)$, is the frequency at which the route R_c occurs in the historical data set as a route or a sub route of any longer routes (with respect to number of vertices traversed). More detailed information and an example of estimating support can be found in the next chapter.

Skyline points. Calculating the support s and deviation d for each route is an important prerequisite to identify a set of skyline points $S(s, d)$. A skyline point in a database context is a point, which is not dominated by any other in any dimension. A skyline point S_1 dominates a point S_2 if it is as good or better in at least one dimension. Skyline queries in the framework are constructed using a two criterial approach - the deviation d of a charging route R_c and its support s are used to create skyline points that expose the charging routes that are not dominated in the dimensions of maximum deviation and support.

Identifying the most frequently used parts of the road network. Finding the routes with the highest values of both distance and support may not tell us the whole picture of what's going on in the road network. The framework simulates charging routes and compares them using their entirety in order to calculate support and create the skyline points. That means that we can compare routes R_c1 and R_c2 paths and check of one of them is part of the other, but we are not able to compare partial extracts of a route's path with another partial extract of another route's path. For example, imagine a map with a river running through the middle and only one bridge connecting the two parts.

It is evident that if a car wants to cross the river, it would have to pass through the said bridge. That means that if we have multitude of charging routes crossing the river we would be able to compare them, calculate support and deviation, but we will never be able to identify straight away that the bridge is the most heavily used part of the road network since its length is not an independent route itself. The goal of the framework proposed below is to find a way to identify such parts of the road network. The information that we might get can be used for road traffic and routing analysis for electric vehicles and can be an important prerequisite for future work in the field.

3. RELATED WORK

3.1 Routing algorithms

The problem of finding suitable routing algorithms for cars has been around for a number of years. Throughout the years many people, especially those involved in the field of graph theory have developed a number of algorithms and research is still being done today. The earliest algorithms from the mid-late 50's like Bellman-Ford's [1], Dijkstra's and its variations [2] laid the foundations of what was to come in recent years [3]. However, the most classic algorithm of all, that gave rise to a multitude of variations is Dijkstra's algorithm. Variations, using heuristics such as A*, D*, Beam search algorithms have been used extensively in many fields - from video games to modern routing devices for cars and tracking systems. The framework, proposed in this paper also uses A* extensively, as it is one of the corner stones of the Charging Route Discovery Algorithm, explained in the next section. The details about A* are also explained there.

3.2 Support

Support is one of the parameters the framework needs to construct the skyline queries. The whole idea of searching through a data set, finding items, in our case charging routes, and discovering patterns in the set is a problem, related to sequential pattern mining. Since we work with recorded paths of charging routes, which are in a sense, strings of characters one approach to solve our problem is to turn to the field of bioinformatics and string mining. In its essence, string mining deals with a limited alphabet for items that appear in a sequence. Sequences may be very long and the applications are many - in biology this is used to discover sequences of proteins or examine genes. One tool that uses string mining is BLAST (Basic Local Alignment Search Tool) [4]. A BLAST search gives us the opportunity to compare a query sequence with a library or database of sequences, and identify library sequences that resemble the query sequence above a certain threshold. BLAST compares only one sequence with many others, however there are some other tools like ClustalW[5], which can compare multiple sequences between each other. However, for this framework a rather different approach has been chosen, inspired by the SPADE (Sequential PAttern Discovery using Equivalence classes) algorithm[6]. The basic concepts of SPADE are adopted - execute length-wise extensions while counting frequency/support starting from the shortest routes possible.

3.3 Skylines

A problem of choosing a subset of a given set by several criteria has been introduced by H. T. Kung in middle 80s' [8]. The mathematical approach titled as "Maxima vector problem" laid a foundation for a recently rather popular Skyline calculation problem in database context. Pioneering work was done by German scientist S. Borzsonyi in early 2000 [7]. Not only he introduced a *Skyline operator* concept as such, but also provided a study on possible SQL extensions for efficient Skyline calculation.
In recent years scientists' efforts to optimize this problem have boosted significantly, it can be observed in these articles [9] [10]. However, the work by H. P. Kriegel [11] stands out as an inspiration for this paper.

3.4 Previous work

In order to make a stark contrast between what has been done in the past and what is being proposed in this paper we will first have a quick glance at the contents of our previous work. In the previous paper, called "Skyline queries framework for electrical vehicles" a model was proposed which tried to tackle similar problems, although in a different fashion. The previous model could also be conditionally divided into two parts - generating synthetic data of electrical vehicles' journeys which includes calculating charging routes and corresponding deviation; the second part is calculating support for the charging routes and computing skyline queries based on the support of the charging routes and the deviation. The end result was a skyline query, which identified the routes with the highest deviation at the different levels of support.

One of the key novelty ideas of that model was the way charging routes were generated. First of all, in the preprocessing stage, as the map was being built, all the charging stations on the map were identified. The map was represented as an undirected connected graph. Running a modified version of the A* algorithm the shortest paths from every charging station to all others were calculated and stored in memory. The sub-graph, containing only the charging stations scattered all over the map and the paths connecting them was labeled as a **metagraph** $G'(C_s, E')$, where $C_s \subset V$ and $E' \subset E$. The metagraph data structure was of key importance when it came to calculating the vehicles' charging routes. The construction of each route began firstly by plugging in a car's source and destination points into the metagraph, after which all of the charging stations within reach were found. One important prerequisite of that idea was that from source and destination at least one charging station ($c \in C_s \subset V$) had to be within the car's autonomy, thus making it reachable. One other prerequisite was that once at a charging station the car needs to have at least one other **unvisited** charging station to fall within the its autonomy.

Thus, by making sure there was always a path from source to destination and by jumping from station to station, the shortest possible of all paths was constructed using the metagraph data. Although convenient, since most of the computation was done in the preprocessing stage, this model has one serious limitation - as we said all of the paths in the metagraph were stored in memory. On a small map, containing only a few charging stations that's not a problem, but on large maps, containing possibly dozens of stations certain issues may arise. First of all, having to calculate, store and retrieve hundreds of paths from may significantly

increase the time for preprocessment and charging route construction. Moreover, once the source V_s and destination V_d points were plugged into the metagraph, the way the shortest possible path was found, given all of the above mentioned prerequisites were met, was by brute force calculation. Once V_s and V_d are connected to all reachable charging stations, the meta-graph is traversed in order to find all feasible path combinations from source to destination. Of course, deviation is also a factor, so generally a path with the lowest possible deviation from the shortest possible route from source to destination is preferred. The following equation reflects the nature of this operation:

$$d_r(R_c(s,d)) = \min\big\{(d_r(R_s(s,c_i)) + d_r(R_s(c_i,c_j)) + d_r(R_s(c_j,d)) \mid c_i, c_j \in C_s \text{ and } c_i \neq c_j\big\}$$

If the case is such that V_s and V_d are within reach from the same charging station, obliging the car to charge only once, the following equation is applied:

$$d_r(R_c(s,d)) = \min\big\{(d_r(R_s(s,c_i)) + d_r(R_s(c_i,d)) \mid c_i \in C_s\big\}$$

Clearly, the previously proposed model, although having certain strengths, was too dependent on memory and time allowed for calculation. One of the improvements this paper proposes in the following chapters is a substitute algorithm for generating charging routes of electrical vehicles which does not heavily rely on memory, nor does it have to make brute force calculations in order to construct the routes, but rather does that "on the fly".

Next, we shall investigate the conditions under which a an already generated and saved route gained support.

Support, denoted $sup(R_c)$, is the frequency of R_c occurrence in the historical data set as a route or a subroute of longer routes (*w.r.t.* number of vertices traversed).

An example of support calculation is given below.

Example

Let's assume that *Figure 14* is the road network and the data set of just 4 routes is given:

1. $c_1 \rightarrow c_4$
2. $c_2 \rightarrow c_4$
3. $v_6 \rightarrow c_4$
4. $v_6 \rightarrow v_9$

The source and destination pairs can be expanded to a full path sequences - routes.

1. $c_1, c_2, v_6, v_8, c_3, v_9, c_4$
2. $c_2, v_6, v_8, c_3, v_9, c_4$
3. v_6, v_8, c_3, v_9, c_4
4. v_6, v_8, c_3, v_9

Finally, the result returned by SCA would look like this:

Route	Support
$c_1 \rightarrow c_4$	1
$c_2 \rightarrow c_4$	2
$v_6 \rightarrow c_4$	3
$v_6 \rightarrow v_9$	4

The table illustrates route's influence to subroutes. It can be observed that route $v_6 \rightarrow v_9$ is a subroute of all 3 longer routes and $v_6 \rightarrow t$ takes part in 2 other routes, therefore support is 4 and 3 respectively.

Using the results obtained for deviation and support of all charging routes, the framework proposes to identify the set of skyline points $S(s,d)$. A Skyline point in database context is a point, which is not dominated by any other. In other words, a point dominates another if it is as good or better in all dimension and better at least at one dimension. Skyline queries are performed using a bicriterial approach: the detour distance of shortest route from source s to destination d (deviation) and its frequency of occurrence in the historical data set (support). The goal of the skyline queries is to find the charging routes that have maximum support and deviation.

4. GENERATING THE SYNTHETIC DATA

As said before, since we do not have a genuine historical data set of EV travel data, one of the main focuses of the model, proposed in this paper is to generate more realistic synthetic data for constructing EV's charging routes. Recreating aspects of the real world environment distances the model proposed from the sphere of the purely theoretical and also provides a valuable insight into possible issues and limitations that may not be obvious at first.

In order to do that, several changes have been made. First of all, the undirected graph, representing the map that had been used before is now a directed graph. This, at a glance insignificant change can actually have a great impact on the results we might get, when we do test the new model. For example, many towns have one way streets and the nature of such entities would be preserved and represented in the implementation of the proposed model. A car may choose to opt out on a route, containing segments with one-way roads, because the deviation might be too large, or vice-versa. Where the electrical vehicles choose to pass through is of another key importance, mostly because the main focus of the project is to identify the parts of the road network that are most heavily used but do not fall into the data set of the recorded charging routes and their corresponding deviation metrics. One other change to the theoretical model, proposed in this paper, is the introduction of categorization for the different road types that make up the road network. Since vehicles travel with different free flow speeds according to the road type, the following classification is proposed:

Type of road	Free flow speed (in km/h)	Free flow speed (in m/s)
Motorway	120	33
Trunk	90	25
Primary	80	22
Secondary	80	22
Tertiary	60	17
Residential	50	14
Unclassified	40	11

Note: All road types are taken from Openstreetmap.org via the overpass-turbo API [12]. Since speed limits vary from country to country, all values chosen are averaged. Although it is unusual to come across a road segment of "unclassified" type, a speed limit of 40 km/h has been chosen

since the vast majority of them are within heavily populated residential districts and are usually short, narrow streets.

Having a categorization of road segments and free flow speed limits is an important prerequisite for fulfilling one of the other novelty ideas which the model in this paper proposes. As said before, the autonomy of the electrical vehicles is now measured in kilowatts hour, and since traveling at different speeds yields to different levels of energy consumption, hence differences in range are present.

The rate at which an electrical vehicle's battery discharges is not that simple to determine. Many are the factors which influence it - the type of the battery (li-ion, lead-acid,nickel-metal hydride, etc.) being the main one. Usually the only sure way to determine the discharge rate of a battery is empirical - just take the car to the field, do some tests, record the data and analyze it. Since this framework proposes only a simulation of journeys of electrical vehicles, but at the same time strives for maximal replication of real-world conditions, in order to re-create the process of battery discharge, which in turn reflects on the range of the car, depending on the speed, some already established discharge rates have to be used. The solution was to use the information, provided by Tesla Motors on one of their blogs [13] about the energy consumption of the Tesla Roadster. A graph, displaying the total energy consumption in kWh at different speeds per mile is displayed below. Since the information provided uses miles to measure distance and miles per hour to measure speed, for the implementation the energy levels are adjusted to their corresponding values when measured in Km/h per kilometer.

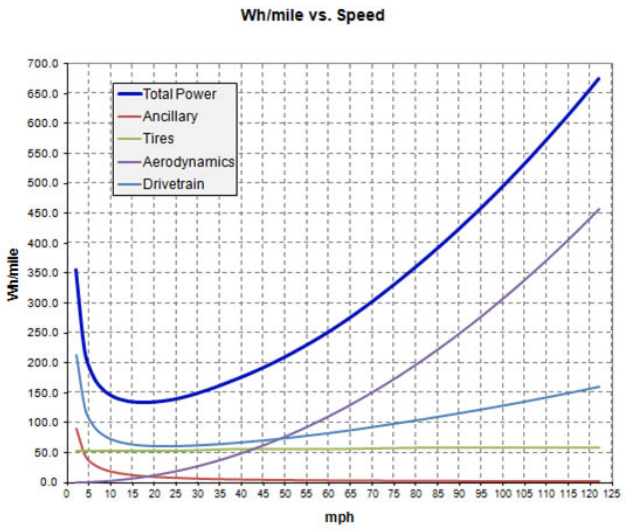

Figure 2: Energy consumption of a Tesla Roadster [13]

Another interesting detail, which is chosen to be presented in the model and the implementation and is again provided by Tesla Motors via their blog [13], is the losses in energy a car might experience due to its aerodynamics in combination with friction with the air. Although those losses can be negligible most of the time, traveling at a high velocity for a prolonged period of time might actually make a difference. The force of air friction on an object is a vector pointing in the opposite direction of movement and it has a magnitude of F_D:

$$F_D = \frac{1}{2}\rho V^2 A C_d$$

Let's break down the equation above - ρ represents the density of air and V is the velocity at which the car travels, A is the frontal area of the and C_D is the drag coefficient, depending on the shape of the vehicle. The more streamlined the car shape, the more easily the car can slice through air without disrupting it. According to Tesla Motors the Tesla Roadster is reported to have a drag coefficient $C_D = 0.35$ [13]. If we multiply the equation above with V we will find the power losses P_L a car experiences because of its aerodynamics:

$$P_L = \frac{1}{2}\rho V^3 A C_d$$

These formulas are used when generating a vehicle's charging route. Wind resistance may greatly influence the way a car moves around the road network - prolonged periods of time, traveling at high speeds may significantly decrease the car's range due to fact that more battery power is needed to maintain the velocity and overcome the negative effects of air friction. This may lead to a variety of possible charging routes for the electrical vehicles, depending on what type of a charging route a car wants to have - seeming fastest routes, passing through highways may turn out ot be time inefficient because of more frequent charging for example. To illustrate how exactly the formulas are used, we shall introduce a simple example. Imagine you have a car with a drag coefficient $C_D = 0.35$, frontal surface area of $2m^2$, traveling at 90 km/h for 1 hour. Knowing that the density of air is roughly $1.2kg/m^3$ if we plug in those values in the second equation P_L we would get a value of 6562.5 W or roughly 6,5 kW. This is the extra power a car needs to overcome its own air resistance at a speed of 90 km/h for the duration of 1 hour, thus making the value 6,5 kWh. If we divide 6562.5 Wh by 90, we get the extra power the car uses to overcome air drag at this speed is roughly 72.91 Wh/km. This value is added on top of the normal consumption of the car per km. If a car uses up, let's say 200 Wh/km at 90 km/h, the new value is 272.91 Wh/km.

4.1 Algorithms

4.1.1 A Algorithm*

The problem of finding the shortest possible path in a graph has been around for many years. Over time, many different algorithms have been developed in order to tackle this issue, Dijkstra's algorithm being the most popular of all. Its simplistic and straightforward approach of traversing vertices in ascending order and using a priority queue to store partial paths laid the foundation for many other algorithms and modifications to be developed in later years. One such modification is the A* algorithm, which to a great extend resembles Dijkstra's algorithm but outperforms it, mainly because of the heuristic function that A* uses to guide its search [14]. A* is an informed search algorithm. It can be described as a best-first search, meaning that it solves problems by searching among all feasible paths to the desired destination for the one that yields to the smallest cost. Among these paths it first considers the ones that appear to lead most quickly to the solution. The algorithm

can be used to find many different types of shortest paths in a graph - distance-based, time-based, most economical, etc. The model, described in the Previous work section uses A* to find distance-based shortest paths, whereas the framework proposed in this paper combines the knowledge of road segment lengths and free flow speed to deliver a swift solution for finding a time-based shortest path in a graph.

A* is a significant modification of Dijkstra's algorithm in the sense that it develops the original idea even more. The main difference, as mentioned above is the presence of a positive value heuristics function $h(n)$. In the case of Dijkstra, the value of this function is zero, meaning that Dijkstra works only with the actual distance parameter $g(n)$ and calculates a possible path by traversing all nodes of the graph. A* makes good use of the heuristic function $h(n)$ to guide the search. In the case of the model, proposed in this paper the Eucledian distance between two points on the earth's surface is used, this is knows as the Haversine formula [15].

$$hav(\frac{d}{r}) = hav(\phi_2 - \phi_1) + cos(\phi_1)cos(\phi_2)hav(\lambda_2 - \lambda_1)$$

where hav is the Haversine function:

$$hav(\Theta) = sin^2(\frac{\Theta}{2}) = \frac{1-cos(\Theta)}{2}$$

d is the distance between the two points.

r is the radius of the Earth (assuming the earth's shape is a perfect sphere).

ϕ_1, ϕ_2: latitudes of points 1 and 2, measured in radians.

λ_1, λ_2: longitudes of points 1 and 2, measured in radians.

By moving along the graph, visiting nodes as we go, we can check the current node's and the destination node's latitudes and longitudes and find the exact distance between them in a straight line, as if we are measuring on the earth's surface.

The whole cost function of the A* algorithm, given a node n is:

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

where $g(n)$ is the distance of the traversed path from source to node n and $h(n)$ is the heuristic function.

Starting with the initial node, the algorithm maintains a priority queue of nodes which are not yet traversed, called an Open set. Intuitively, the lower *f(n)* for a given node n the higher its priority. At each step of the algorithm, the node with the lowest *f(n)* value is removed from the queue, the f and g values of its neighbors are updated accordingly, and these neighbors are added to the queue. The algorithm continues until a goal node has a lower f value than any node in the queue (or until the queue is empty). Goal nodes may be passed over multiple times if other nodes with lower f values remain there, as they may lead to a shorter path to a goal. And finally, the f value of the goal is then the length of the shortest path.

4.1.2 The Charging Route Discovery Algorithm

Algorithm 1 Charging route discovery algorithm

```
S - source
D - destination
Cur - current node
Z - range end node
Q - temporary variable
P - temporary variable
aut - autonomy of car
chargingStationsList - a list of the charging stations across the road network
CandidateList - a list of candidate charging stations
Cur = S
fmin(n) = Double.max() - a variable to store the minimum value of f(n)
while Cur != D do
    path = calculate A*(Cur,D)
      ▷ Find the shortest possible path between source and destination using A*
    for i=0; path.size(); i++ do
    dist = path.element(i).distance();
        if dist > aut then Z = path.element(i − 1)
        end if
    end for
    for j=0; chargingStationsList.size(); j++ do
        if A*(Cur, chargingStationsList(j)) < aut then
    put chargingStationsList(j) in CandidateList();
        end if
    end for
    for p=0; CandidateList.size(); p++ do
     Q = A*(Cur, CandidateList(p))
     ▷ A* distance between current node and candidate station
     P = A*(CandidateList(p), Z)
         ▷ A* distance between candidate station and Z
     Dev = (Q + P) - Z.dist()
       ▷ Deviation from Source to charging station then to Z
     h(n) = Haversine(CandidateList(p), D)
      ▷ Calculates the heuristic Euclidian distance from candidate station to Destination
     f(n) = Dev + h(n)
        if f(n) < fmin(n) then
     fmin(n) = f(n)
     Cur = CandidateList(p)
        end if
    end for
end while
```

As mentioned before, the framework, showcased in this paper proposes a new idea for the creation of charging routes for electrical vehicles, meant to substitute the creation and usage of the already presented **metagraph** data structure. The Charging Route Discovery Algorithm comes as a completely different approach to tackle the problem of charging route creation. This algorithm does not rely on a preprocessed metagraph with memorized paths between the charging stations on the road network. It calculates the path of the charging routes on the fly as it traverses the graph in search of the destination point.

First the algorithm uses A* to calculate the shortest possible path from source s and destination d as if the car doesn't

have to recharge on its way. The red dots represent charging stations, dispersed along the map, the blue one are nodes and the lines connecting the nodes is the shortest found path from source to destination. **(fig. 3)**.

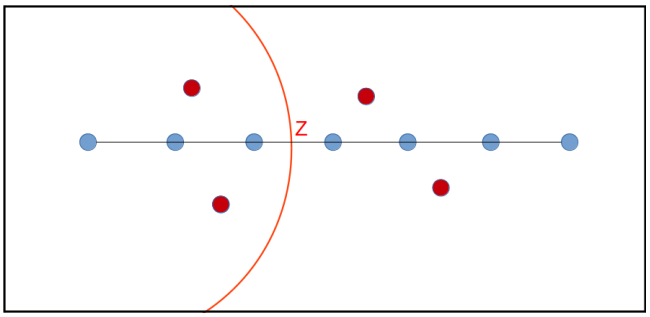

Figure 3: Calculating the shortest possible path from source to destination

Next, knowing that we have an electrical vehicle with a battery and a corresponding *autonomy* value we calculate where exactly on that shortest path, given the lengths and free flow speeds of road segments the car passes through, the autonomy of the car ends. It is represented as Z in **fig. 4**.

Now that we know which stations could be reached by the car on its own, we must choose a candidate station at which the car will charge. To do this we iterate through the list of candidate stations and calculate the distance between the starting point, the candidate station and Z. Subtract from that the length of the shortest possible path from source to Z and we get the deviation in distance a car has to make in order to visit any of the proposed charging stations. Now that we have the deviation we use the same heuristic function from the A* algorithm to calculate the distance between the candidate station and the destination point. The station with the lowest combined score of deviation and heuristic distance, gets chosen to be the place where the car will recharge its battery. Mathematically, the formula can be expressed as follows:

$$F_{cs} = Dev_{cs} + h(cs)$$

where Dev_{cs} is the deviation value for the current candidate station;

$h(cs)$ is the heuristic evaluation of the distance between the candidate and the final destination point;

F_{cs} is the function that combines both deviation and heuristic estimate.

Using this simple formula gives us the opportunity to compare the pros and cons of going to different stations. A natural choice for a driver, without a computing device to tell him where to go, is to head for the station which is closest to Z. Most of the time this would be a right choice, as it is highly likely this strategy would yield to a smaller overall deviation of the journey. However in some edge cases, this strategy may not be as successful as one might think.

Imagine a situation in which the station closest to Z yields to some positive deviation and a few kilometres further back there is a station located on the shortest possible path between source and destination. In this case, since we're trying to minimize our overall deviation, it would make more sense to charge on the station, located on the shortest path and choose to opt out on our seemingly more natural choice. This is where the above showcased formula comes to guide our search - evaluating deviation and heuristic distance may help us find the balance between the amount of kilometers we're ready to go out of the shortest possible path to recharge and the approximate distance left to our goal.

In the example showed in **fig 4** the charging station, which is the closest to Z is chosen as a designated place for recharge of the car. The deviational path from the shortest route to the chosen station is marked in orange.All distance are calculated by using the A* algorithm. The car heads for the station and the path from source is saved.

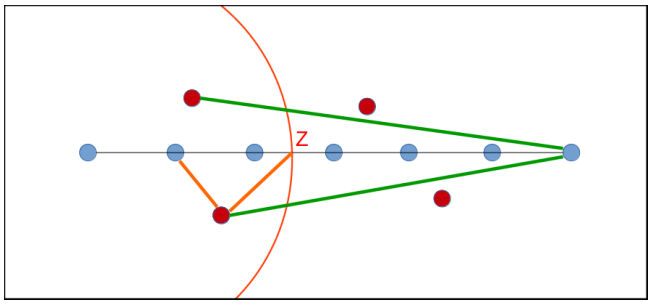

Figure 4: Finding the end of the car's autonomy

Once the car is at the chosen charging station, the whole algorithm is repeated again. A new shortest path from the current source point to destination is found, a new end point of the car's autonomy is established along the shortest path and again, a new suitable charging station is found **fig 5**. Light green on the map marks the path traversed by the car so far. The algorithm ends once the destination point is reachable without further recharging.

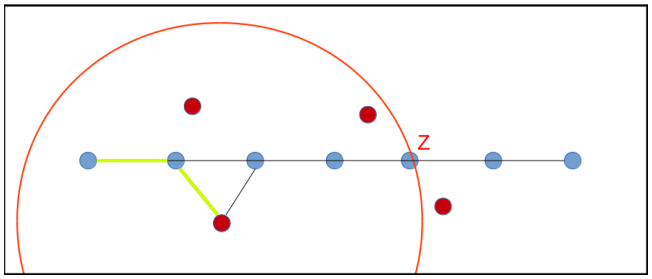

Figure 5: The algorithm is repeated until the destination is reached

Once the algorithm is terminated and the destination has been reached, what's left is the path of the charging route of the car. The path of the charging route is then constructed in a backwards manner - every node has a "parent" feature. Once a node has been chosen to be on a shortest path of some sort , the previously chosen node is set as its Parent. In that way, starting at the final destination point of a car's journey, by back tracing each node to its parent node we will eventually reach the original starting point of the car's journey. What we end up with is the complete route the car covered on its way , charging stations included. Compare

this to the previously calculated shortest possible route and one can also measure the deviation from it.

4.2 Deviation and Support

As described in the Introduction chapter, one of the goals of this paper is to create a framework, which helps in the analysis of historical EVs' travel data. The charging routes that are both: inconvenient (large deviation) and important (support) should be found. In this section the Support and Support calculation algorithm (SCA) are described in detailed manner.

Support calculation algorithm (SCA)

The support of a route has been introduced in Previous work *(chapter 3)*. It can be observed that it is related with the number of cars taking a particular charging route - the more frequent the route, the bigger the support.

SCA takes charging routes (historical data) as an input and returns the support (a positive integer representing frequency) for every unique route in a data set as an output.

Another important aspect of Support is that long routes, with respect to the number of vertices traversed, may have a lot of different subroutes within. This particularity should also be considered by SCA.

As mentioned before, the idea is to work in ascending manner, starting from the shortest routes with respect to the sequence length n and checking whether these are subroutes of longer routes. Then incrementation is done and the algorithm works with routes with the length of *n+1*.

Algorithm 2 Support calculation algorithm (SCA)

Input: Charging routes R_c
Output: Support for every unique route

1: Sort charging routes ascendingly $w.r.t$ sequence length
2: **for** (k=0; $k < db.size$; k++) **do**
▷ $db.size$ - number of routes in a data base
3: **for** (j=k+1; $j < db.size + 1$; j++) **do**
4: **check**(R_k, R_j);
▷ R_k - route which support is calculated
▷ R_j - longer route in which R_k is checked
5: **end for**
6: **end for**

Pseudo code of *Algorithm 2* shows how the Support is calculated. In *Line 1* all routes are sorted in ascended order. *Line 2-3* iterates through the database and finally in *Line 4* the call to procedure *check()* is performed.

The *Algorithm 3* describes Procedure *check()*, which gets 2 routes as an input, where R_k is the route for which the support is being calculated and R_j is the routes in which R_k will be searched.

Line 1 defines the size of **for** cycle, it can not be longer than subtraction of given routes. In *Line 3-5* comparison of vertices in a routes is performed together with counter *p++* in *Line 4*, which indicates how many corresponded vertices have been discovered. In *Line 6* the actual length of route $D_k()$ and counter p is done. If the sizes are equal, this means that $D_k()$ is contained in $D_j()$ and Support of $D_k()$ should be increased.

Line 8, 11-13 are not fundamental to support calculation, but are vital in Skyline algorithm, which will be described in the next chapter.

Algorithm 3 Check procedure in SCA

Input: R_k and R_j - routes
▷ $R.length$ - route length

1: **for** (i=0; $i \leq R_j.length - R_k.length$; i++) **do**
2: p=0;
3: **while** p<R_k **and** $R_j(i+p) == R_k(p)$ **do**
4: p++;
5: **end while**
6: **if** p==$R_k.length$ **then**
7: $sup(R_k)$ ++;
8: $R_k.association.list(i) = R_j$
▷ R_j is added to R_k association list
9: **end if**
10: **end for**
11: **for each** i in $R_k.association.list(i)$ **do**
12: $est(sup(R_k.association.list(i)) = sup(R_k) - 1$
▷ every route in association list of R_k is assigned with support estimate from R_k
▷ If a route has been already assigned with a support from previous calculations, the smaller support from 2 is chosen
13: **if** $sup(R_x)$ not null **then** overwrite $sup(R_x)$ with lower value.
14: **end if**
15: **end for**

4.3 Skyline

In this section, the Skyline query algorithm is described in a detailed manner.

As introduced in Problem definition *(Chapter 2)*, a skyline point is a point, that is not dominated by any other point in a data set. In other words - a point dominates another if it is as good or better in all dimension and better at least at one dimension. The algorithm focuses on max max problem - maximum deviation and maximum support. This leads to finding skyline points that have large deviation and large support.

One of the biggest challenges in developing an efficient Skyline query algorithm is reducing the number of operations executed (performance time). The core idea behind this is adopting proper pruning techniques. Yet trade-offs must be made.

Skyline queries algorithm

In this problem setting the chosen trade-off is that deviation of all charging routes is precomputed beforehand.

The main idea behind developing this algorithm is that support is not calculated for every charging route. Points are pruned from candidates set on estimated values rather than precise and costly (computation wise) calculations.

In general, the proposed algorithm exploits the step wise movement. While calculating support for a route R_k, longer routes that have R_k as a subroute are memorized in an association list. After having a precise support value $sup(R_k)$, the algorithm assigns $sup(R_k)$*-1* as an estimate support value, denoted as $est(sup(R_k))$ to all members of R_k association list. This assignment operation holds on the assumption that support for R_k can serve as an upper bound for members of association list. The intuition here is very simple. Imagine we have a data set of only three routes R_k,

R_j, R_p. R_k is the shortest route and is also a subroute in R_j. R_j is a subroute of R_p which is the longest route of them all. So, if we calculate the support for the shortest route R_k we would get a value of 3. Since the other two routes are in R_k's association list, they will be assigned an estimated support value of 2, which is the upper bound of the support they might get. And if we were to calculate the real support values of all the routes we would see that neither R_j, nor R_p would get a support higher than 2. If there existed a route, that contained R_p for example as a subroute, that would automatically increase the support of all shorter routes that are contained within. There is no way for a route to get a higher support than the routes it contains. This estimated support value is later used to perform pruning based on it.

Let's illustrate the usage of estimated support. Suppose there are 2 routes:

- $R_k(n)$ with a length of n and calculated support
- $R_j(n+1)$ with a length of *n+1* and estimated support

And R_k is not necessarily a subroute of R_j.
Based on deviation, 3 cases might occur:

Algorithm 4 Pruning on estimated support value

Input:
R_k - route with a length of n with precise support value
R_j route with a length of *n+1* and estimated support.

1: **if** $dev(R_k) = dev(R_j)$ **then**
2: **if** $sup(R_k) > est(sup(R_j))$ **then**
3: **Discard** R_j
4: **else**
5: **Calculate** $sup(R_j)$
6: **end if**
7: **else if** $dev(R_k) > dev(R_j)$ **then**
8: **if** $sup(R_k) \geq est(sup(R_j))$ **then**
9: **Discard** R_j
10: **else**
11: **Keep** R_j ▷ Points are not comparable
12: **end if**
13: **else if** $dev(R_k) < dev(R_j)$ **then**
14: **if** $sup(R_k) > est(sup(R_j))$ **then**
15: **Keep** R_j ▷ Points are not comparable
16: **else**
17: **Calculate** $sup(R_j)$
▷ The estimation is not enough and precise support must be calculated
18: **end if**
19: **end if**

In the first case, given that the deviation of the two routes is the same we compare the calculated support value of the one with estimated support value of the other. If the route R_k with the calculated support has a value, higher than the estimated support value of the other route R_j, then the latter route is discarded, since there is no way an estimated support value can exceed the calculated support value of the other route. On the other hand, if the estimated support value of a R_j is higher than R_k's calculated support value we should calculate the real support for R_j and based on that we can choose which point to discard.

In the second case, when the deviation of the route with the precisely calculated support value (R_k in the pseudocode) is higher than the support of the other route (R_j), we check the support values and discard R_j if its estimated support is smaller or equal to the support value of R_k. We can safely do so, since estimated support is used as an upper bound for the value of calculated support a route might get. In other words, there is no way R_j would end up with a calculated support higher than R_k, since its upper bound for that support is already worse or equal to R_k's real support value. In the other case, we just keep both points since they are not comparable.

The final case is when the route with estimated support has higher deviation than the route with the calculated support. The points will not be comparable if R_k's precisely calculated support value is higher than R_j's estimated support value. In the other case, we would need to calculate the exact support of R_j to see which point to discard. Another pruning case is observed when support is already calculated for both routes. This situation is almost identical to *Algorithm 4*, the only difference is in *Line 17*, instead of calculating support, the route R_j is discarded.

Algorithm 5 Pruning on calculated support

Input: R_k and R_j - routes with a length of n and precise support value.

Line 1-13 are the same as in *Algorithm 4*, except $sup(R_j)$ is used instead of $est(sup(R_j)$
14: **if** $dev(R_k) < dev(R_j)$ **then**
15: **if** $sup(R_k) > sup(R_j)$ **then**
16: **Keep** R_j ▷ Points are not comparable
17: **else**
18: **Discard** R_j
19: **end if**
20: **end if**

Below, a rough skeleton of the skyline calculation algorithm is presented:

Algorithm 6 Abstract skyline calculation algorithm

1: $S() = \emptyset$ ▷ $S()$ - set of Skyline points
2: **for each** R_c **do**
3: $dev(R_c) = d_r(R_c) - d_r(R_s)$
▷ for every R_c deviation is calculated
4: **add** R_c to C_s
▷ array of candidate set is filled with charging routes from data set
5: **end for**
6: **sort**(C_s)
7: ▷ Candidate set is sorted ascendingly *w.r.t.* number of vertices
8: $n = 2$
9: **while** ($C_s \neq \emptyset$) **do**
10: $sup(R_c(n)) = support.calculate()$
▷ *calls SCA - Algorithm 2.* Support calculation for R_c with the length of n
11: **Prune**(C_s)
▷ *Prune(C_s) calls Algorithm 5*
▷ *Prune(C_s) calls Algorithm 4*
12: **if** $R \in C_s$ not discarded **then**
13: Compare R with *elements* $\in S()$
14: **if** R not dominated **then**
15: **add** R to $S()$
16: **end if**
17: **end if**
18: $n++$;
19: **end while**

5. IDENTIFYING HEAVILY USED PARTS OF THE ROAD NETWORK

Finding the support values for different charging routes and calculating the skyline queries is a prerequisite for the main focus of the model, proposed in this paper - identifying the most heavily used parts of the road network, which have a support higher than any other value of support a charging route might have. In other words, the primary goal here is to identify sub-routes which are "above" the skyline query and try to include them. Of course, for those parts of the road network to be included they also need to have some positive deviation value. To illustrate, let's assume we have routes R_c1 and R_c2 with a corresponding support value of respectively 2 and 1. That means that R_c1 has been traversed by two different vehicles, while R_c2 has been traversed once by a third vehicle. But what if all three routes share a common stretch of road? Then this subroute, shared by all three routes would have a support value of 3 but since it's not an independent route itself, it cannot be identified as such and consequently it won't be represented in the skyline query. The goal of the framework proposed in this paper is to make that possible.

Since all the information about the paths of the generated charging routes is saved as a string sequence of traversed nodes of the graph, representing the map, a naive approach would be to compare every single path of a charging route with every other and look for the longest common substring in the routes that we are comparing. Such longest common substrings elements can then be then extracted and depending on the number of times they occur in the data set, they gain support. We're mostly interested in the ones who would gain support higher than the support of any charging route in the skyline query. However, as simple as this idea is, it's still a very naive approach, we might end up wasting time and memory comparing charging routes that are nowhere near each other. One step towards eliminating unnecessary calculations is to find suitable criteria according to which we can prune parts of the data set.

The model, showcased in this paper proposes this to be done on the basis of using the geographical attributes of the recorded charging routes' paths. Since we're representing real world road networks as a graph, the information about the latitude and longitude of each node of the graph is also available. This is used to create an algorithm which clusters charging routes with similar geographic data together and compares them, the rest of the data set is pruned. The foundation of this algorithm lies in the idea that a map may be divided into quadrants using a latitude and longitude boundary. An example is shown on fig. 6 . On it we can see a map of Aalborg and the surrounding area which has been divided into four quadrants by two boundaries running through 57.0 North latitude parallel and 9.91 East longitude meridian. The boundaries are chosen to divide the map into four equal quadrants but theoretically it doesn't matter how many parallel and meridian boundaries we can have and how big the quadrants are.

Figure 6: Dividing the map into four quadrants

Dividing the map into quadrants and using the coordinates of the nodes, constituting the path of a charging route gives us the opportunity to correctly identify in which parts of the map a charging route is located. For example, a route which is located in Q1 and Q2 would be compared only with routes which share either or both of the same quadrants. Further divisions of such nature could be made, which will increase the accuracy of the algorithm. Theoretically speaking, if the geographical boundaries which divide the map form a grid dense enough, then we would know exactly which routes to compare and no unnecessary comparisons would be done. The density of the grid though would be dependent on the size of the map and it may vary widely.

The Algorithm

Now that we have introduced the basic idea of dividing the map into quadrants in order not to do unnecessary comparisons of routes, it's time to introduce the algorithm for identifying the most heavily used parts of the road network. The first thing the algorithm does is to iterate once through the data set containing the charging routes in order to see the quadrants each route falls into (see algorithm 7). Each quadrant is presented as a list, containing routes.

Algorithm 7 Preprocessment

▷ R_c denotes the recorded paths of charging routes in the data set.
$QLat$ - a latitude value at which the map is divided
$QLon$ - a longitude value at which the map is divided
for each R_c **do**
  for each $R_c.element()$ **do**
    if $R_c.element()latitude \geq QLat$ **then**
      if $R_c.element()$ longitude $< QLon$ **then**
        $R_c \in Q_1$
      else
        $R_c \in Q_2$
      end if
    else
      if $R_c.element()$ longitude $< QLon$ **then**
        $R_c \in Q_3$
      else
        $R_c \in Q_4$
      end if
    end if
  end for
end for
sort$(R_c \in Q_i)$

In the next step, we identify the most commonly used parts of the road network. This is done by comparing the paths of all the routes that fall into a certain quadrant, the longest common substring is found. The routes are compared in the following manner - the first one is compared with all the rest, then the second with all the rest, then the third and so on. This ensures that we won't check the same pair of routes twice. The complexity of this type of iterating is $O((n-1)*\frac{n}{2})$. Although this complexity looks low it is important to note that it resides in $O(n^2)$. During the iteration all routes in a quadrant are compared two at a time, using a dynamic programming algorithm for finding the longest common substring in the recorded paths of the routes (see algorithm 8). The algorithm has a time complexity of $O(p*m)$, where p and m are respectively the lengths of routes being compared. All in all the overall time complexity of the algorithm is $O(n^2*p*m)$. One additional feature which provides further pruning of charging routes to compare is a validating function which keeps track of which routes have been compared and which not. This function stores pairs of routes that were checked in a hash map data structure. Let's illustrate with an example where this comes in handy - assume that routes R_c1 and R_c2 both run through quadrants $Q1$ and $Q2$. In that case they would be present in both lists, containing the routes that run through each quadrant. Once we start, we would see that R_c1 and R_c2 are in $Q1$ so we would compare them in their entirety. The pair (R_c1, R_c2) would be stored in a separate hash map data structure, so we would know that they have been compared. When we come across the same two routes when comparing the contents of $Q2$ we would skip checking. This very simple validation provides additional pruning and significantly reduces the number of operations that need to be done in the cases when we have a low-granularity segmentation of the map and long routes that may run through most if not all the quadrants.

Algorithm 8 Finding the longest common substring

$LCSrep$ - a list, containing all found $LCSvalue$ objects.
▷ In each quadrant group QX
for each R_c, $R_c{+}_1 \in QX$ **do**
▷ We now compare R_c and $R_c{+}_1$
  $LCS.compare(R_c, R_c{+}_1)$
  dynamicArray = array(1..m , 1..n)
  $LCSLength = 0$;
  $LCSvalue = \{\}$
  for i := 1..m **do**
    for j := 1..n **do**
      if R_c [i] == $R_c{+}_1$[j] **then**
        if $i == 1$ or $j == 1$ **then**
  $dynamicArray[i,j] := 1$
        else
  $dynamicArray[i,j] := dynamicArray[i{-}1, j{-}1]{+}1$
        end if
        if $dynamicArray[i,j] > LCSLength$ **then**
  $LCSLength := dynamicArray[i,j]$
  $LCSvalue := R_c[i - LCSLength + 1..i]$
        else
        end if
        if $dynamicArray[i,j] == LCSLength$ **then**
  $LCSvalue := LCSvalue + R_c$[i-LCSLength+1..i]
        end if
      else
  $dynamicArray$[i,j] := 0
      end if
    end for
  end for
  return $LCSvalue$
  $LCSrep.add(LCSvalue)$
▷ We add the LCSvalue to the list of other LCSvalues found. This list would be used later to extract the LCSvalues and find deviation for each of them if such exists
end for

Algorithm 9 Finding the longest common substring-cont.

$LCSskylineCandidatesList$ - a list to keep the candidates from LCSrep that have a positive deviation value as they might appear in the skyline
$LCSHash$ - a hash map data structure to check whether an LCSValue element has been
for b=0; b<LCSrep.size(); b++ **do**
  $dev = A*$ (LCSrep[b].firstNode,LCSrep[b].lastNode)
  if $dev > 0$ **then**
    if $LCSrep[b]$ not in $LCSHash$ **then**
  $LCSHash.add(LCSrep[b])$
  LCSrep[b].sup() = 1
    else
  LCSrep[b].sup() += 1 ▷ Increase support by one
    end if
  $LCSskylineCandidatesList.add(LCSrep[b])$
  end if
end for

Let's do a short explanation of algorithms 8 and 9. Algorithm 8 is a dynamic programming algorithm for finding the

longest common substring between the paths of two routes, i.e. which parts of the road network those two routes share. Once this substring, referred to as *LCSvalue* is found we store it in a list with all other *LCSvalue* elements found. Not every element in the list is of interest to us, we're trying to find the longest common subpaths that have a positive deviation value and support, which is higher than the support of the charging routes in the skyline query, so we can add them to the query. We know turn our attention to algorithm 9 which is a continuation of algorithm 8. To identify which subpaths are of value to us and can be in the skyline, we go through the list of *LCSvalues* that have been found and first of all find the $A*$ distance between the first node of the subpath and the last one. Since we know the exact path and length of each *LCSvalue* element, by finding this A* distance we would know if this *LCSvalue* element has a deviation or not. If it has a positive deviation value we then add it to a hash map data structure which keeps track of every element's support value. If an element is not present in the hash map it is added and a support value of 1 is assigned to it, if it already exists, then the support value is increased by one. The elements with a positive deviation value and a support higher than any of the routes in the skyline are added to a candidate list to be put in the skyline.

6. TESTS AND RESULTS

For the current test a map covering a big part of the region east of Munich has been chosen. The map covers an area roughly the size of 3100 km^2 and has a total network length of 4368 km , excluding the residential roads in the villages and cities. Only the main road network entities has been considered a part of the road network - main roads, passing through towns, major boulevards and streets, motorways and all first class and second class roads. This particular map has been chosen mostly because of its dense road network coverage and the fact that across it around 40 different charging stations are placed **(fig 7)**. The information about the placement of the charging station is taken from OpenChargeMap via their web app API [17].

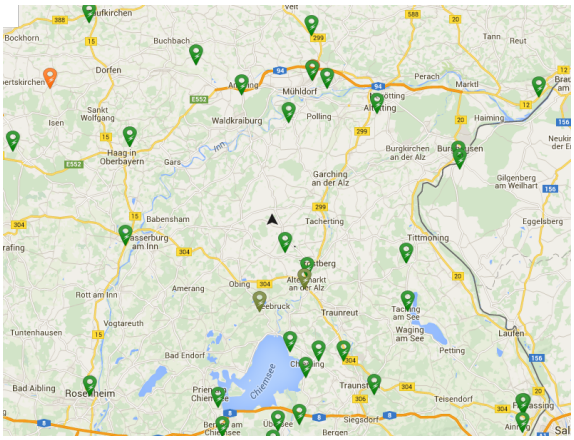

Figure 7: The charging stations coverage of the map tests were run on

Tests have been performed using different configurations of the model. The shortest possible charging routes in respect to distance and time are found. Deviation is also measured in the same way for both types of charging routes. The starting and ending points are chosen randomly among others from two lists of points from both peripheries of the map (west and east). Since we're using the discharge rate of the battery of a Tesla Roadster ,the autonomy value for the electrical vehicles is set to be 9 kWh, which guarantees the car will travel for approximately 45 km with a constant speed of around 90 km/h. This is not the original range of the Tesla Roadster. Since the map is relatively small, limiting the capacity of the car's battery is done so that tests could be performed on this small scale map. The shortest and longest routes measured are respectively 86.6 km and 122.7 km long. The time for calculating the charging routes of the electric vehicles was also measured. The tests are performed on a machine with an Intel® Core™ i5-5200U CPU @ 2.20GHz processor and 8 GB of RAM.

The programming language, chosen for the implementation is Java. Additional libraries such as JUNG [16] are used. JUNG is a software library which provides a common extendable language for the modeling and analysis of data that can be represented as a graph or network. Maps are created by parsing JSON files containing geographical data which is then handled by JUNG to create the basic graph representation of the map. JUNG is rather basic and it cannot entirely suit the needs of this implementation, so its basic functionalities and graph representing capabilities were used as a basis. All the algorithms and additional modules for data analysis were designed and implemented in a way to further enhance the capabilities of JUNG. The modules responsible for support calculation, skyline query calculation and identifying the most frequently used parts of the road network are also developed in Java. Transfer of data between the modules is done by creating, reading and parsing text files. The starting and ending points for the cars are chosen at random from both ends of the map. This is done in the following way - there are two latitudinal separation lines near both ends of the map. They divide the map in three parts - the two small peripheral parts are where the starting and ending points are chosen from. The nodes from those peripheries are kept in two separate lists. When a we want to create a charging route, one point from one list is chosen at random as a starting point and another point from the other list is chosen at random as a destination point. No starting and ending points are chosen from the middle part of the map. The two latitudinal separator lines have nothing in common with the proposed method of splitting the map in quadrants, described in the previous chapter. They are merely used to ensure that the source and destination points of each charging route are located at both ends of the map and are chosen randomly.

The results of the test, run on the implementation are presented in the tables below. The first one shows the time it took to calculate up to 8000 charging routes, which are supposed to be the shortest possible in distance and the second one presents the same calculations done for charging routes supposed to be the shortest in time.

Table 1: Total calculation time for finding distance-based shortest charging routes

Routes	Time (in seconds)
1000	147.257
2000	298.331
5000	751.002
7000	1129.754
8000	1292.907

Table 2: Total calculation time for finding time-based shortest charging routes

Routes	Time (in seconds)
1000	150.021
2000	305.816
5000	768.109
7000	1157.179
8000	1320.345

As it is shown on figure 12, the difference in time needed to calculate the charging routes using the two different configurations is almost linear. However, the model gives a slightly worse performance when calculating the time-based shortest charging routes. One possible explanation for this is that actually looking for the fastest route is closely linked to distance and free-flow speed, so the weight of the edges of the graph may change completely when travel time becomes a primary factor. That may lead to more additional calculations, bigger detours from the shortest possible routes and so on. Performance tests on a single map may give inconclusive results. To entirely understand how the model handles when calculating distance-based and time-based shortest charging routes may require extensive testing on a number of different maps with a variety of road network types. The goal of the tests done and presented in this paper is only to show that the model is functional and achieves the tasks it's supposed to do.

Tests were performed, using two different configurations of the model. In the first case the shortest charging routes in terms of distance are found , deviation in distance and time are also measured (fig. 8 and fig. 9) In the second case the shortest charging routes in term of time are found (fig. 10 and fig. 11). Again, deviation in distance and time are measured.

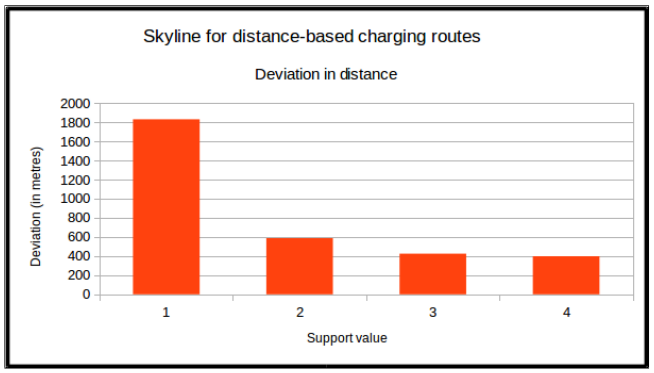

Figure 8: Skyline for distance-based charging routes and their deviation in distance

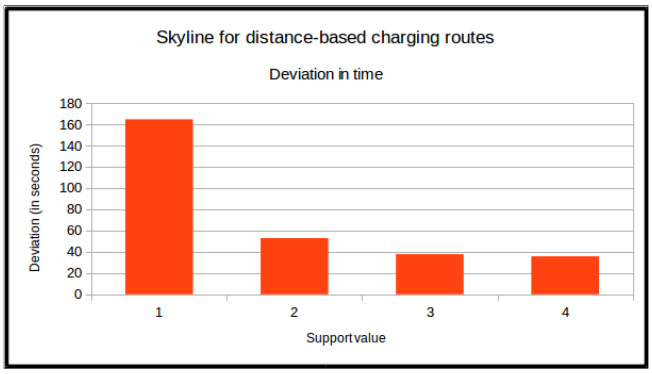

Figure 9: Skyline for distance-based charging routes and their deviation in time

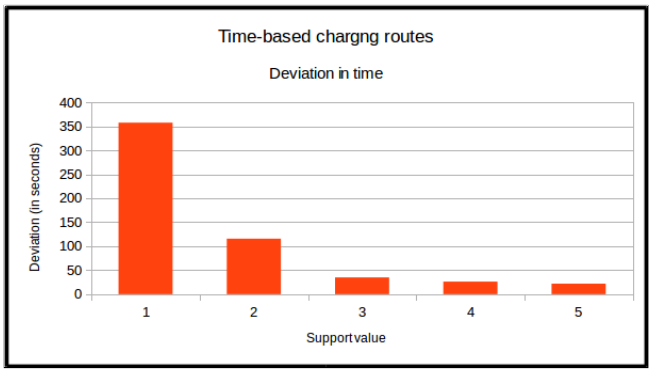

Figure 10: Skyline for time-based charging routes and their deviation in time

What is interesting to note here is that there is a significant difference in the results of both test cases. In the case when we're calculating distance-based shortest charging routes, as expected, the deviation values for distance and time are much lower than the ones of the time-based shortest charging routes. Time-based shortest charging routes also gained higher support values. Of course, there is a reasonable explanation for this. Using different criteria to calculate and construct a charging route leads to different results - distance-based shortest charging routes make better use of the road network available. It doesn't matter if the car uses primary, secondary or tertiary roads as long as it gets to its final destination with the lowest possible deviation. On the other hand, if we're looking for the fastest routes, then the weights of the graph edges, representing the map would change significantly - cars would tend to use motorways and primary roads, if possible as they give them the opportunity to travel at higher speeds. That is also the reason why more routes in this case got a higher support value - cars just prefer the fastest road arteries available. It's visible even in the skyline queries for the time-based fastest charging routes. The charging route, present in the skyline that has a support value of 1 has approx. 8 km in deviation. However, since the car's average speed would be higher if it chooses to use only primary roads and motorways, the deviation in time can still be acceptable.

Once the most frequently used parts of the road network are identified, their deviation and support are calculated and

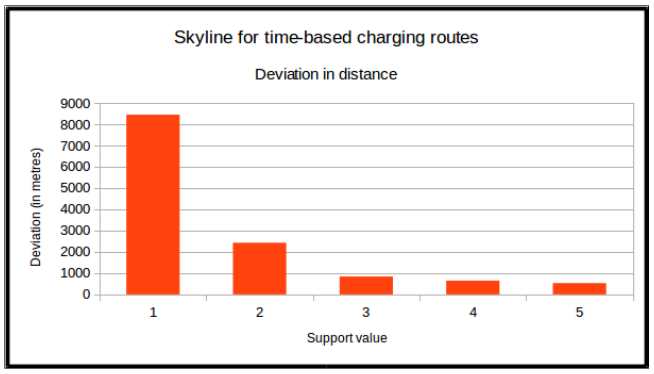

Figure 11: Skyline for time-based charging routes and their deviation in distance

they are added to the skyline (fig. 13 and fig 14). What is interesting to note is that in the updated skyline queries in both cases when we calculate distance-based and time-based charging routes, new points with much higher support have been reported. Deviation is also slowly decreasing as support goes up. For some points we have the same value of deviation but different values of support. One explanation for this phenomenon is that, since deviation is accumulated around charging stations, as cars make a detour to get to the station different subpaths merge their paths to make the same deviation. In other words, around charging stations there is a point of convergence of vehicles' routes and all of them have to make this deviation in order to charge. Longer subpaths that make a deviation get one support value and the shorter subpaths that are closer to the charging station get the same deviation but also have a higher support since they are shared by even more vehicles. There are many other subpaths across the road network that have even higher support, as they are shared by tens, maybe even hundreds of cars, but since they either don't have a deviation or have very little deviation for their support value, they are not present in the skyline query.

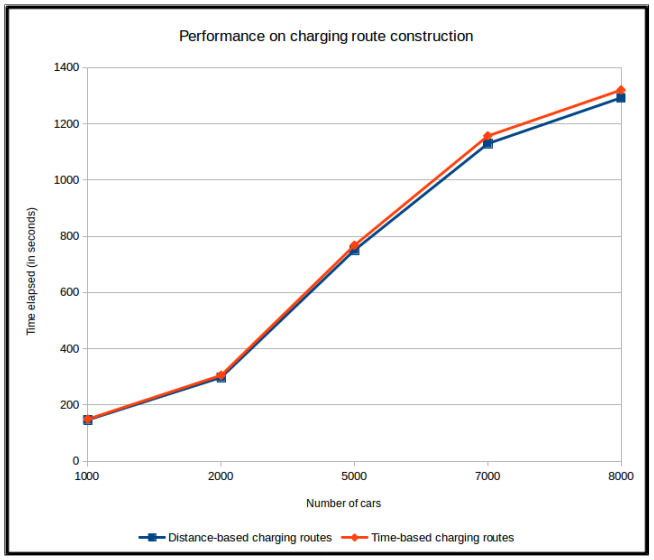

Figure 12: Performance of the model

Of course, as we said before, running tests on a single map may not give conclusive results. The behaviour of the model is largely dependent on the map tests are run on. In our case we have a relatively small map with a very dense road network and a multitude of charging stations, more or less evenly distributed across the network. Most stations are actually located either on or very close by to major road arteries so it's also normal for cars to have relatively low deviation values. However, on a different map with a sparse road network or unevenly distributed charging stations the picture might be quite different. To understand how the model handles numerous tests on various types of maps have to be done in future.

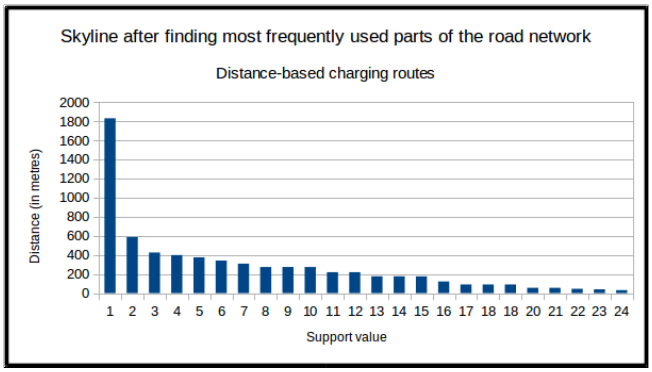

Figure 13: Skyline for time-based charging routes and their deviation in distance

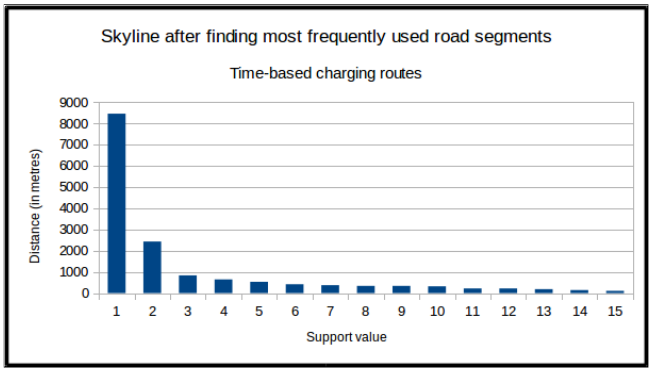

Figure 14: Skyline for time-based charging routes and their deviation in distance

7. CONCLUSIONS

The tests performed show that the theoretical model proposed in the paper can produce some interesting results. However, as we already mentioned, extensive testing on various kinds of maps with diverse types of road network might be needed to fully understand how the model behaves and if there are some potential bottle necks in its performance. It would be interesting for a graphical module to be developed as part of some future work in order to highlight and visualize charging routes, support values of different edges and heavily used parts of the road network.

Of course a number of improvements can be introduced to the theoretical model as well. For example, the simple

charging route discovery algorithm is entirely based and dependent on A* and one future improvement would be to substitute A* with some other algorithm which is even faster or uses less memory. The Simplified Memory Bounded A* (SMA*) is one such algorithm. Its main advantage, as the name suggests is that it uses bounded memory while the A*'s memory usage might be exponential.

Another possible improvement would be for a better algorithm for calculating support on routes to be developed. The current one, uses a pretty straight-forward naive approach which can be very computationally heavy and ineffective in very large data sets.

The dynamic programming algorithm for identifying heavily used parts of the road network may also be substituted by something better performing - the whole algorithm can run using suffix trees, for example. A linear-time algorithm for building suffix trees was developed by Ukkonen in the mid 90's [18], so using it to create the trees and by performing searches for longest common subpaths on the tree will significantly improve the speed at which calculations are done and also give us better memory efficiency.

Also, different types of experiments and tests can be carried out. An improved physics model can be introduced, for example, which takes into consideration not only air drag but also acceleration, deceleration, traffic light cycles and so on. Right now we assume that the cars move under perfect conditions, on empty roads, traveling with their maximum free flow speed. So one huge future step forward would be to develop a framework that not only tries to recreate the physical limitations of the real world but also take in consideration other traffic participants who are also traveling across the road network in real time.

The possibilities are truly endless and there are many directions in which we might go - from simply improving calculation algorithms, to trying to create more realistic synthetic data, to finding even more ways to analyze the charging routes of electrical vehicles in order to understand how they utilize the road network or figuring out how to place charging stations across a road network. Precisely by exploring the possibilities of future development in this area of human knowledge we can prepare for a future world, where electric vehicles would not be something new and strange, but a mere commodity.

8. REFERENCES

[1] Bellman, Richard (1958). *"On a routing problem". Quarterly of Applied Mathematics 16: 87-90. MR 0102435.*

[2] Thomas H. Cormen; Charles E. Leiserson; Ronald L. Rivest; Clifford Stein. (2001) *Introduction to Algorithms* (Second ed.). "Section 24.3: Dijkstra's algorithm. p. 595-601.

[3] Thorup, Mikkel (2004). *"Integer priority queues with decrease key in constant time and the single source shortest paths problem". Journal of Computer and System Sciences 69 (3): 330-353.*

[4] Altschul, Stephen; Gish, Warren; Miller, Webb; Myers, Eugene; Lipman, David (1990). *"Basic local alignment search tool". Journal of Molecular Biology 215 (3): 403-410.*

[5] Chenna R, Sugawara H, Koike T, Lopez R, Gibson TJ, Higgins DG, Thompson JD (2003). *"Multiple sequence alignment with the Clustal series of programs". Nucleic Acids Res 31 (13): 3497-3500.*

[6] Zaki, M. J., (2001). *"Spade: An efficient algorithm for mining frequent sequences".* In: Machine Learning. Vol. 42. p. 31-60.

[7] Stephan Borzsonyi, Donald Kossmann, Konrad Stocker. (2001) *The Skyline Operator*. 17th International Conference on Data Engineering p. 421-430

[8] H. T. Kung et. al, (1975) *Finding the Maxima of a Set of Vectors*

[9] Mullesgaard, Kasper; Pedersen, Jens Laurits; Lu, Hua; Zhou, Yongluan (2014). *"Efficient Skyline Computation in MapReduce".* 17th International Conference on Extending Database Technology (EDBT). p. 37-48.

[10] F. Afrati, P. Koutris, D. Suciu, and J.D. Ullman. (2012) *Parallel Skyline Queries*. International Conference on Database Theory (ICDT), p. 274-284.

[11] Hans-Peter Kriegel, Matthias Renz, Matthias Schubert. (2010) *Route Skyline Queries: A Multi-Preference Path Planning Approach.* ICDE 2010

[12] http://openchargemap.org/app/?view=map-page

[13] http://www.solarjourneyusa.com/EVdistanceAnalysis5.php

[14] Zeng, W.; Church, R. L. (2009). *"Finding shortest paths on real road networks: the case for A*".* International Journal of Geographical Information Science 23. p. 531-543.

[15] van Brummelen, Glen Robert (2013). *Heavenly Mathematics: The Forgotten Art of Spherical Trigonometry. Princeton University Press. ISBN 9780691148922. 0691148929. Retrieved 2015-11-10.*

[16] http://jung.sourceforge.net

[17] http://openchargemap.org/app/?view=map-page

[18] Ukkonen, E. (1995). *"On-line construction of suffix trees". Algorithmica 14 (3): 249-260.*