# Skyline Query Framework for the Analysis of Electric Vehicle Trajectories

Ion-Anastasiu Sanporean
Aalborg University
Aalborg, Denmark
isanpo14@student.aau.dk

Paulius Galinauskas
Aalborg University
Aalborg, Denmark
pgalin14@student.aau.dk

## ABSTRACT

Electrical vehicle's travel is affected by battery capacity, distance and charging stations along the road network. Thus charging routs are necessary to be optimized or to identify possible locations for charging stations. This paper proposes first a framework to identify trajectories in a dataset that are both inconvenient and important using Skyline Queries Algorithm (SQA) and second, a method to generate data under different conditions. Skyline queries are performed using a bicriteria approach: detour time of fastest route/path from source to target (deviation) and its frequency of occurrence in the historical dataset (support). Experiments were performed considering two different sizes maps: central Aalborg as a small map and North East Canada as a big map. The tests were performed to prove the efficiency of SQA by comparing it with 2 other algorithms: brute force skyline and sort filter skyline (SFS). The results showed that SQA running time outperforms the other two on the big map, while on the small map is comparable with Brute Force running time. The SQA algorithm identifies the skyline set in less than 3/4 of the time Brute Force algorithm does, using efficient pruning techniques. The Skyline obtained for different number of routes shows that the larger the support values, the smaller deviation values on both types of maps. The results have revealed that the framework is efficient in cases of analyzing large datasets, however a negative impact on performance was observed when analyzing small datasets.

## 1. INTRODUCTION

Electric vehicles (**EVs**) are becoming more and more popular due to the advantages they offer, such as low level of environment pollution, high level efficiency, low level of noise and that they can be charged from any energy source that produces electricity. Automotive manufacturers like TESLA, Nissan, BMW have started to promote intensively the sale of EVs and hybrid vehicles. Since EVs have started to be used, slowly but surely, in both public (buses) and private sectors (cars), it is expected that they will be common in the near future.

The restriction enforced by EV's battery capacity/charging level affects travel distance. Battery charging may be needed to reach desired destinations. This is also influenced by the placement of charging stations along the road network (see *Figure 1*). Travel distances that are beyond EV's battery capacity limitation require the EV to follow a charging route. There are cases when, because of the placements of charging stations, an EV travels longer in time/distance than the fastest/shortest route between desired starting and ending points. By optimizing or identifying possible locations for charging stations, an EV travel might be reduced in terms of time/distance and the "inconvenience" of battery capacity limitation might diminish.

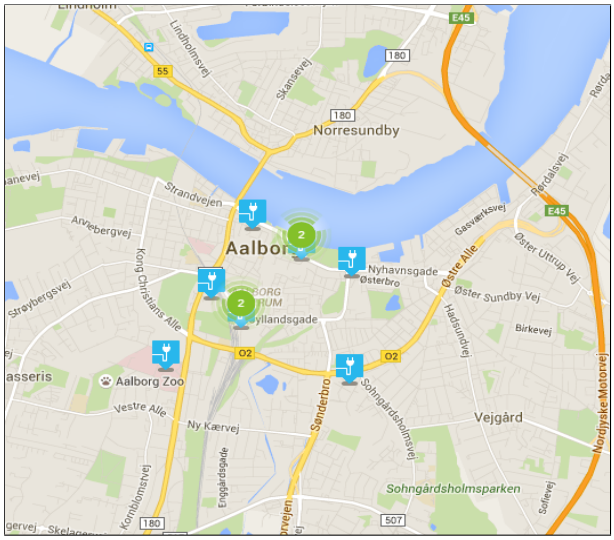



Figure 1: Illustration of charging stations placement

This paper proposes first, a framework that analyzes a dataset containing information about EVs' routes in order to help with future placement of charging stations along the road network. The purpose of this framework is to find EVs routes that are both: inconvenient (large detour from the fastest route) and important (high frequency of occurrence in a dataset as a route or a subroute) using Skyline queries. The second proposition is a method for generating the previously mentioned dataset under different conditions (charging stations along the road network, road type, EV's battery capacity/state of charge), in particular, EVs' routes.

The paper is organized as follows: first of all, the problem chosen to be solved is briefly presented. The next section is dedicated to related work. In chapter 4, the main algorithms used are presented. The next section offers a debriefing on how a synthetic dataset (EVs routes) is generated and ends with results obtained using different parameters values. The paper ends with conclusions.

## 2. PROBLEM DEFINITION

The following problem setting is considered: the road network is represented as a directed graph $G = (V, E)$, where $V$ denotes the set of vertices and $E$ denotes the set of edges in $G$. The charging stations that are placed along the road network are represented as a set $C_s$ and $\forall c \in C_s \subset V$ has a charging rate as a definitory attribute.

*Definition 1.* **Charging rate**, denoted as $C - rate(c)$, represents the rate of which a battery is charged in charging station $c \in C_s$.

Furthermore, $\forall e \in E$ has a set of attributes: road type, maximum speed, length between end points, power consumed by an EV to traverse it.

*Definition 2.* **Road type** is an attribute of an $e \in E$ and depicts the characteristic of a road based on physically layout and use of it (i.e. highway, street, private road etc.).

*Definition 3.* **Maximum speed**, denoted as $maxSpeed$, represents the highest speed at which an EV traverses a road segment of a certain type (i.e. for highways $maxSpeed = 130$ km/h).

*Definition 4.* **Length between end points** represents the distance between the starting point and ending point of an edge $e \in E$.

*Definition 5.* **Power consumed** to traverse an edge $e \in E$, denoted as $pow$, represents the energy consumed over time by an EV to travel between the endpoints of a road segment.

*Definition 6.* **Travel time on an edge**, denoted as $t_e$, is a positive weight representing the time to travel between edge's endpoints at maximum speed.

The dataset analyzed by the framework proposed in this paper is called historical dataset, denoted as $H$. This dataset is associated with a road network and contains information about different EVs and their routes.

*Definition 7.* **Charging route**, denoted as $R_c(s, dest)$, is a route in the historical dataset and is represented as a sequence of vertices describing the path from source ***s*** to target ***dest***, containing at least one charging station, where the EV stops for recharging battery.

*Definition 8.* **Charging time**, denoted as $t_c$, represents the time spent by an EV in a charging station $c \in C_s$ for recharging its battery and is depending on $C-rate(c)$ (*Definition 1*) and EV's battery capacity/state of charge.

As mentioned in *Definition 7*, $R_c(s, dest)$ is a sequence of vertices describing the path between endpoints:

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

As an observation, a charging route may contain a set of charging stations, but only a subset of them may actually be used for charging EV's battery.

The travel time for a charging route has two components. First, the total time to traverse every edge on the path from source ***s*** to target ***dest*** and second, the time spent to recharge the battery:

$$t_r(R_c) = \sum_{v_i \in R_c} t_e(v_i, v_{i+1}) + \sum_{v_i \in C_s} t_c(v_i)$$

*Definition 9.* **Fastest route**, denoted as $R_f(s, dest)$, represents the path describing the shortest distance from source ***s*** to destination ***dest***, traversed at maximum speed.

*Definition 10.* **Optimal charging time**, denoted as $t_{opt}$, represents the minimum time spent for charging in a charging station c with $C-rate(c) = max\{C-rate(c_i) \mid c_i \in C_s\}$.

The travel time for fastest route has two components as well. First, the total time to traverse every edge on the path from source ***s*** to target ***dest*** and second, an optimal charging time.

$$t_r(R_f) = \sum_{v_i \in R_f} t_e(v_i, v_{i+1}) + \sum_{v_i \in C_s} t_{opt}(v_i)$$

Having the time to travel along the fastest route, a new attribute for charging route, ***deviation***, is computed.

*Definition 11.* **Deviation**, denoted as $dev(R_c)$ represents a positive weight associated with extra time needed by EV to travel along the charging route $R_c$ compared with the time needed for the fastest route $R_f$, from same source ***s*** to same target ***dest***.

$$dev(R_c) = t_r(R_c) - t_r(R_f)$$

A second attribute of a charging route, ***support***, is also computed.

*Definition 12.* **Support**, denoted as $sup(R_c)$, represents the frequency of $R_c$ occurrence in the historical dataset 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 3* is a road network. In order to show how ***support*** is calculated we consider the following dataset of routes:

1. $c_1 \rightarrow c_4 = c_1, c_2, v_4, c_3, v_5, c_4$
2. $c_2 \rightarrow c_4 = c_2, v_4, c_3, v_5, c_4$
3. $v_4 \rightarrow c_4 = v_4, c_3, v_5, c_4$
4. $c_3 \rightarrow c_4 = c_3, v_5, c_4$

The support calculated for the above mentioned routes is presented in the next table:

| Route | Support |
|---|---|
| $c_1 \rightarrow c_4$ | 1 |
| $c_2 \rightarrow c_4$ | 2 |
| $v_4 \rightarrow c_4$ | 3 |
| $c_3 \rightarrow c_4$ | 4 |

This table gives an illustration of how routes' support influences their subroutes' support values. It can be observed that route $c_3 \rightarrow c_4$ is a subroute of all the other routes in the dataset. Therefore it has the highest support, 4. The longest route $c_1 \rightarrow c_4$ has support 1, since it is not a subroute of another route and it appears only once in the dataset.

For a deeper analysis of historical data, a set of "hidden" charging paths is extracted from existing charging routes. The purpose for data mining after these particular paths is to identify the subroutes that are intensively trafficked (high support). They might reflect bottlenecks cases, when, in absence of multiple route choices for traveling from source to destination, **EV**s follow only these paths.

*Definition 13.* **Critical charging path**, denoted as $P_c(s, dest)$ represents an individual subroute of existing charging routes, with the particularity of exceeding certain threshold values for support and for its length w.r.t. number of traversed vertices:

$$sup(P_c) > sup\_threshold\_value$$

$$length(P_c) > length\_threshold\_value$$

Using deviation and support of all charging routes, the framework proposes to identify two sets of skyline points $S(s, dest)$: one for charging routes and one for critical charging paths. Skyline queries are performed using a bicriteria approach: detour time of fastest route/path from source ***s*** to target ***dest*** (*deviation*) and its frequency of occurrence in the historical dataset (*support*). The results obtained represent charging routes/critical charging paths that have maximum support and deviation.

# 3. RELATED WORK

An early version of the framework proposed by this paper was introduced in [1]. Aspects from future work presented in [1] are implemented in the new framework (like charging level of EV's battery at the initiation of a journey, time needed to charge different levels of battery, travel speed, road types). The main analysis' focus on EVs trajectories in the new implementation is switched from distance to time.

In the field of *trajectories data mining* has been done extensive and intensive research. *Trajectory pattern mining* is one in a variety of mining tasks performed on different datasets. In [2] and [3] sequential pattern mining algorithms, such as *longest common subsequence*(LCSS) are presented. Suffix Trees, which are designed for strings, can be adapted and used for finding sequential trajectory patterns. In our work we focused on finding the longest common subroutes which are not members of a given dataset. Therefore, we propose the use of an algorithm for finding the *longest common subtring* (LCS) instead of using LCSS algorithms.

In middle 80s' H. T. Kung introduced in [7] the problem of choosing a subset of a given set by several criteria. "Maxima vector problem" as mathematical approach, laid a foundation for Skyline calculation problem in database context. In early 2000 the german scientist S. Borzsonyi has done pioneering work in this field [8]. Beside introducing the concept of *Skyline operator*, he also provided a study on possible SQL extensions for efficient Skyline calculation.

Scientists' efforts to optimize this problem have increased significantly, as observed in articles [9], [10]. The work done by H. P. Kriegel [11] surpass the other as inspiration for this paper.

# 4. ALGORITHMS

In this chapter, the main algorithms used by the framework that analyzes EVs' historical data are presented in a detailed manner. The purpose of this framework is to find EVs' routes that are both important (high frequency of occurrence in dataset as routes or subroutes) and inconvenient (large detour from the fastest route). Finding these skyline points has to be done using efficient methods and algorithms.

***Terms and Variables***

As mentioned previously, the analysis of historical dataset $H$ focuses on two attributes: support and deviation for each route, which are used to obtain the skyline points. Since these attributes are not provided, they need to be calculated.

There are two possibilities to obtain the values for support and deviation: before and during computing the set of skyline points. The first case is not attractive, since it demands high computation, proportional with the number of routes in the historical dataset and their length (w.r.t. number of vertices). Instead, it is preferable to calculate support and deviation for a minimum set of routes (***candidate set***), values that can be used as reference for computing the skyline points set. Having calculated support values for some routes, estimations on support values for longer routes that contain them may be done. The same goes for deviation values.

Let there be a set, association list of a route, denoted as $AssocList$ :

$$AssocList(R_c(s, dest)) = \{R_c(s_i, dest_i) | R_c(s, dest) \subset R_c(s_i, dest_i) \in H\}$$

An association list has the following property:

**Property 1.** A route has support value lower than its subroute:

$$sup(R_c(s_i, dest_i)) < sup(R_c(s, dest))$$

**Proof**. Support represents the frequency of $R_c$ occurrence in the historical dataset as a route or a subroute of longer routes (w.r.t. number of vertices traversed). If a route $R_c(s, dest)$ is identified as subroute of a larger route $R_c(s_i, dest_i)$, then $sup(R_c(s, dest)$ will be bigger with at least $+1$, since its occurence is more frequent. (See **Example** in *Chapter 2*).

It can be observed that the values for support and deviation that each member of an association list has, can be estimated. Later on this chapter, it will be explained how these estimations are done and how they help during the process of computing the skyline points set.

We will continue now with presenting main algorithms. In *Table 1* the main variables used are introduced.

## 4.1 Support

Support of a route has been introduced in *Chapter 2*. This parameter is related with the number of cars following a particular charging route, once or several times - the more frequent the route, the larger the support value for it.

Support calculation algorithm has as input the charging route from historical dataset. The output is the support value (a positive integer representing frequency of occurrence) for every **unique** route in this dataset.

A particularity considered is that a long route (w.r.t. number of vertices traversed) may have a lot of different subroutes within. In other words, this route will be a member of an association list for every of its subroutes.

The idea behind support calculation is to work in ascending manner, starting from the shortest routes (w.r.t. number of vertices traversed) and check whether these are subroutes of longer routes. If there is a longer route that checks the subroute requirement, the longer route becomes a member of its subroute association list. Furthermore, an estimate support value is attributed to the longer route. The maximum value for this estimate cannot exceed its subroute support value, and more, is at the most $support(subroute) - 1$ *(Example in Chapter 2)*. In case, a longer route is member in association list for multiple subroutes, its estimate support value is adjusted to the minimum value of these.

Pseudocode of *Algorithm 1* presents how the support values are calculated.

| Candidate's property | Description |
|---|---|
| *vector of pointers* AssocList | Association list. A list which contains pointers to routes in which the candidate was found |
| *bool* isInAssocList | Is in association list. Reveals if candidate is in someone's association list. |
| *bool* calculated | A bool variable, which indicates if candidate's support and deviation has been calculated. |
| *vector of vertices* chargingRoute | Sequence of vertices from **s** to **dest**. *Definition 7* |
| *vector of vertices* fastestRoute | Fastest route from **s** to **dest**. *Definition 9* |
| *double* estDeviation | Estimates deviation. Deviation, which has been assigned through association list. Serves as an upper bound. |
| *double* deviation | *Definition 11* |
| *double* maxDev | Maximal deviation. Is a maximal value (based on power consumption) from candidate's association list |
| *int* estSupport | Estimated support. Support value, which has been assigned through candidate's association list. Serves as an upper bound. |
| *int* support | *Definition 12* |
| *double* powerConsumption | Power consumed to travel along a charging route. |

Table 1: Variables explanation used in pseudocodes

**Algorithm 1** Support calculation algorithm (supportCalc)

**supportCalc(candidate)**
**Input:** *candidate* - a single route
**Output:** Support for *candidate* and estimated support for *candidate*'s association list

```
1: i =candidate's position in CanSet vector
2: for (i=0; i < canSet.size - i - 1; i++) do
3:     j=i+1
4:     if (candidate is part of CanSet[j]) then
5:         candidate.support++
6:         candidate.AssocList.push_back(CanSet[j])
7:     end if
8: end for
9: for (k=0; k < candidate.AssocList.size; k++) do
10:    if (candidate.AssocList[k].estSupport > candidate.support − 1) then
11:        candidate.AssocList[k].estSupport = candidate.support − 1
       ▷ every route in association list of candidate is assigned with support estimate from candidate
12:        candidate.AssocList[k].isInAssocList = true
13:    end if
14: end for
```

The first phase, *Lines 2-8*is to calculate support for a route and construct its association list.

The second phase, *Lines 9-14* is attributing the estimate support values to every existing member in the association list. In *Lines 10-11* the estimated support value is adjusted to a more appropriate value. In *Line 12*, the fact that a longer route appears in an association list is marked, using a boolean flag. This helps later on during skyline computation, presented in *Section 4.3*.

## 4.2 Deviation

Deviation is the second dimension of the skyline set and is found by subtracting the travel time of fastest route $R_f(s, dest)$ found by an A* algorithm, from the travel time of charging route $R_c(s, dest)$, which is an object in the historical dataset.

The cost function of A* is the following :

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

where *g(n)* represents the time to travel from initial point to $n$ and *h(n)* represents the heuristic function from node $n$ to destination point *dest*.

As heuristic, the time to travel on Euclidian distance between two points is chosen. Finding this distance is trivial and effective, and traveling on shortest path ensures shortest traveling time as well.

Deviation algorithm has as input the charging route from historical dataset. The output is the deviation value for this route.

During the phase of calculating deviation for one route/ candidate, the corresponding association list is also updated with estimate values.

***Deviation estimation***

*Definition 13.* **Maximum possible deviation** of a charging route, denoted as *maxDev*, is the difference between the time to travel along the charging route and the time to travel on the Haversine distance between the same source and destination.

The time to travel on Haversine distance between source ***s*** and destination ***dest*** is similar to the time to travel along the fastest route. The only difference is that the travel speed on Haversine route, denoted as $R_H$, is considered to be the maximum possible speed available for the road network w.r.t existing road types (i.e. highway).

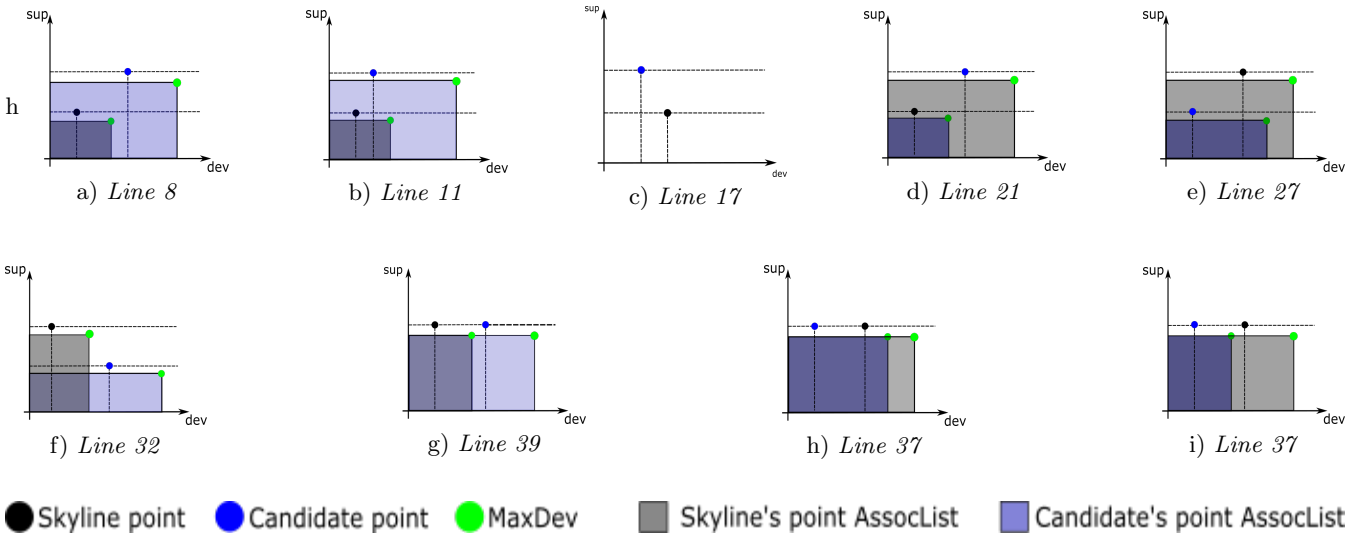



Figure 2: Pseudocode illustrations

$$t_r(R_H) = \sum_{s,dest \in R_H} t_e(s, dest) + \sum_{v_i \in C_s} t_{opt}(v_i)$$

Identifying the route(s) in the *AssocList* that have a large difference between $t_r(R_c)$ and $t_r(R_H)$ gives us the chance to obtain an upper bound value for deviation estimate.

Having an estimate for deviation as a maximum possible deviation for one route, helps with the fact that $t_r(R_f)$ is not computed unless truly needed. This might reduce the complexity of operations, by not calculating a fastest route for every charging route existing in the analyzed dataset.

Maximum possible deviation of a route is expressed as follows:

$$maxDev(R_c) = t_r(R_c) - t_r(R_H)$$

The upper bound value for estimated deviation is computed by founding the member/route with highest *maxDev* in candidate's association list.

Pseudocode of *Algorithm 2* presents how deviation values are calculated.

First, in *Lines 1-2*, the fastest route for candidate and associated deviation are calculated. After, in candidate's association list, the member/route with largest/higher power consumption is found and an estimate value for associated deviation is computed (*Lines 3-4*).

The last phase, *Lines 5-7*, is updating the deviation values of every member in candidate's association list with the value computed in *Line 4*.

## 4.3 Skyline

As introduced in *Problem Definition* (*Chapter 2*), a skyline point is a point that is not dominated by any other point in a dataset.

The Skyline algorithm focuses on finding skyline points that have large support and large deviation (max-max problem). The efficiency of this algorithm relies on execution time (number of operations executed). Therefore, having a good set of data for analysis and usage of proper pruning techniques are essential.

**Algorithm 2** Deviation calculation algorithm (deviationCalc)

**deviationCalc(candidate)**
**Input:** *candidate* - a single route
**Output:** Deviation for *candidate* and estimated deviations for *candidate*'s association list

1: $candidate.fastestPath = A\_star(candidate)$
2: $candidate.deviation = t_r(candidate.chargingPath) - t_r(candidate.fastestPath)$
3: **for** (k=0; k < candidate.AssocList.size; k++) **do**
4: $candidate.AssocList[k].estDeviation = t_r(candidate.chargingPath) - haversineDist(s, dest)/maxSpeed$
5: **end for**
6: $maxPath = argmax\{a.estDeviation \quad | a \in candidate.AssocList\}$
▷ find route with highest estimated deviation in candidate's association list
7: $candidate.maxDev = maxPath$

Previously in this chapter, the notion of association list along with the methods of computing estimated support/ deviation values for its members were introduced.

The main idea behind Skyline algorithm is pruning the Candidate Set using estimate values combined with real calculated values for support and deviation. The pruning method proposed identifies possible cases when a candidate can be discarded along with its association list.

In *Algorithm 3* the checking procedure of a candidate from Candidate Set with Skyline set is presented. The algorithm has as input a candidate for Skyline set. The output result is that the candidate is discarded (in some cases along with his association list) or added to Skyline set.

An illustration of how *Algorithm 3* works is presented in *Figure 2*. The variables/parameters that are checked represent candidate points for skyline set and their association lists, points that are already in skyline set along with their

**Algorithm 3** Candidate's check with Skyline vector (checkSkyline)

**checkSkyline(candidate)**
**Input:** *candidate* - a single route
**Output:** Candidate is discarded or pushed in to the Skyline vector

```
1: if (Skyline.size == ∅) then
2:     Skyline.push_back(candidate)
3: else
4:     for (i=0; i < Skyline.size; i++) do
5:         if (candidate.support > Skyline[i]) then
6:             if (candidate.deviation ≥ Skyline[i].deviation) then
7:                 Skyline.push_back(candidate)
8:                 if (candidate.deviation > Skyline[i].maxDev) then
9:                     delete(Skyline[i].AssocList) from canSet
10:                    delete(Skyline[i]) from Skyline
11:                else
12:                    for (j=0; j < Skyline[i].AssocList.size; j++) do
13:                        Skyline[i].AssocList[j].isInAssocList = false
14:                    end for
15:                    delete(Skyline[i]) from Skyline
16:                end if
17:            else
18:                Skyline.push_back(candidate)
19:            end if
20:        end if
21:        if (candidate.support < Skyline[i]) then
22:            if (candidate.deviation ≤ Skyline[i].deviation) then
23:                if (candidate.maxDev < Skyline[i].deviation) then
24:                    delete(candidate.AssocList) from canSet
25:                    delete(candidate) from canSet
26:                else
27:                    for (j=0; j < candidate.AssocList.size; j++) do
28:                        candidate.AssocList[j].isInAssocList = false
29:                    end for
30:                    delete(candidate) from canSet
31:                end if
32:            else
33:                Skyline.push_back(candidate)
34:            end if
35:        end if
36:        if (candidate.support == Skyline[i]) then
37:            if (candidate.deviation > Skyline[i].deviation) then
38:                Repeat Line 7-15
39:            else if (candidate.deviation < Skyline[i].deviation) then
40:                Repeat Line 23-34
41:            end if
42:        end if
43:    end for
44: end if
```

association lists and maximum deviation *maxDev*. Candidate and skyline points are represented "above" their association lists as a consequence of *Property 1* (*Chapter 4*). Several cases are identified during checking procedure and each of them is detailed bellow.

As seen in *Figure 2(a)* , if candidate's values for support and deviation exceed the values for support, respectively the upper bound value for deviation (maxDev) of the point in Skyline set/its association list, the latter is pruned along with its association list (from Candidate Set).

In *Figure 2(b)* only the point in Skyline Set is pruned, since its association list's members may have a larger deviation than the candidate. As a consequence, the association list of the point that was pruned is returned to the Candidate Set.

*Figure 2(c)* illustrates the case when the point in Skyline set and the candidate are not comparable and so are their association lists. As a result, the candidate becomes a point in Skyline set.

*Figure 2(d)* presents the opposite case of *Figure 2(a)*, when the candidate is pruned along with its association list from Candidate Set.

Another mirror case is also illustrated in *Figure 2(e)* in relation to *Figure 2(b)*. In this situation, the candidate point is pruned and its association list is expanded in Candidate Set.

*Figure 2(g)* and *Figure 2(i)* present the situation in which the support values are equal, but candidate's/point's in Skyline deviation value exceeds the upper bound value (*maxDev*) for point's in Skyline/candidate's association list. As a result the candidate/skyline point is pruned along with its association list.

In the case illustrated in *Figure 2(g)*, when the candidate and skyline points have the same support, but the latter has a deviation value that exceeds the upper bound value for candidate's association list, the result is the same as the one presented for *Figure 2(e)*.

In the beginning of this chapter it was mentioned that our intention is to spend as less time as possible in calculating real values for support and deviation, which are needed for Skyline set computing. This can be achieved by using efficient pruning method.

As presented in *Algorithm 3*, pruning can be done not only on real/calculated values, but also on estimates.

The following algorithm, *Algorithm 4*, presents how the Skyline set is obtained and more, when support and deviation are actually calculated. This algorithm has as input the historical dataset and as output the Skyline set.

Three specific situations are identified during Skyline set computation.

In case of candidate not being a member of an association list but has estimated values for support and deviation, the check is done on estimates (*Lines 5-8*). This situation occurs after the cases presented in *Figures 2(b),2(e),2(h)*.

When a candidate that is not a member of an association list and has no values for support and deviation is met, they are calculated *(Lines 9-13)* and afterwards it is checked with the Skyline set *(Line 14)*.

The last of three situations is when a candidate is a member of an association list and doesn't have calculated values for support and deviation. This occurs when during the checking procedure presented in *Algorithm 3*, this candidate can not be pruned from Candidate set (Example *Figures 2(b),2(e),2(h)*). In other words, it is needed to calculate the real values for both dimensions: support and deviation *(Lines 16-17)*. A high occurrence of this situation has a negative impact on algorithm's performance.

**Algorithm 4** Skyline Query Algorithm (SQA)

**Input:** $CanSet$ - all routes
**Output:** $Skyline$ - routes in Skyline

1: $Skyline = \emptyset$
2: Sort charging routes in CanSet ascendingly *w.r.t* sequence length
3: **while** $CanSet \neq \emptyset$ **do**
4: **bool** *1stPhaseFound == false*
5: **for** (i=0; i < *CanSet.size*; i++) **do**
6: **if** ($CanSet[i].isInAssocList == false$ **and** $CanSet[i].estimated == true$) **then**
7: $checkSkyline(CanSet[i])$
▷ *checkSkyline algorithm is performed on estimated values*
8: **end if**
9: **if** ($CanSet[i].isInAssocList == false$ **and** $CanSet[i].calculated == false$) **then**
10: *1stPhaseFound = true*
11: $CanSet[i].calculated = true$
12: $supportCalc(CanSet[i])$
▷ *Algorithm No x*
13: $deviationCalc(CanSet[i])$
▷ *Algorithm No x*
14: $checkSkyline(CanSet[i])$)
15: **end if**
16: **if** ($1stPhaseFound == false$) **then**
17: **if** ($CanSet[i].calculated == false$) **then**
**Repeat** *Line 11-14*
18: **end if**
19: **end if**
20: **end for**
21: **end while**

## 4.4 LCS

The algorithm presented in this section proposes to find the critical paths. The problem can be solved using a longest common substring algorithm (LCS algorithm).

Adapted to our situation, the problem of finding the LCS can be described as follows:

Having the historical dataset $H = \{R_1, R_2, ..., R_n\}$, where $R_i$, $0 < i < n + 1$, represent the routes, we need to find the longest common critical paths. These paths have to be long enough (w.r.t. number of vertices) and need to have a minimum frequency of occurrence in the dataset analyzed ($sup(P_c) > sup\_threshold\_value$).

The scientific literature presents different approaches to solve the LCS problem: brute force, dynamic programming, suffix trees. The latter of these seems to be the most efficient since it has the best time complexity $O(n)$.

The longest common substrings of trajectories $R_i \in H$ can be found by building a generalized suffix tree (GST) for the routes, and then finding the deepest internal nodes, which have leaf nodes from all the routes in the subtree below it. In order to build the generalized suffix tree, each route has a unique terminator string (*i.e.$1*).

GST is built using Ukkonen's algorithm [4].

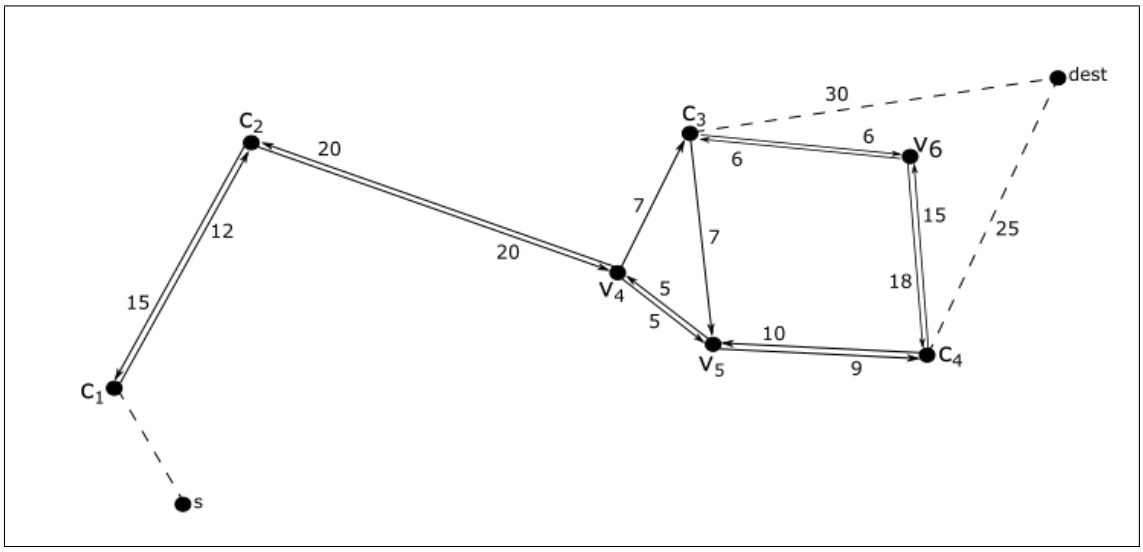
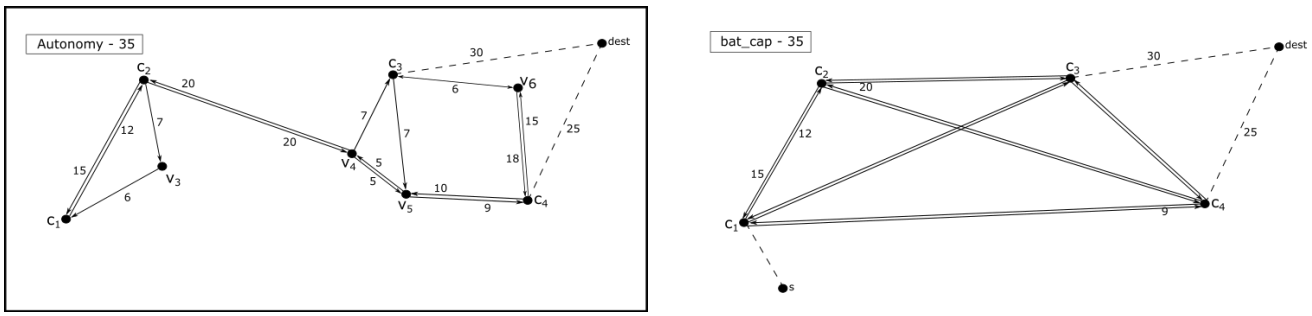
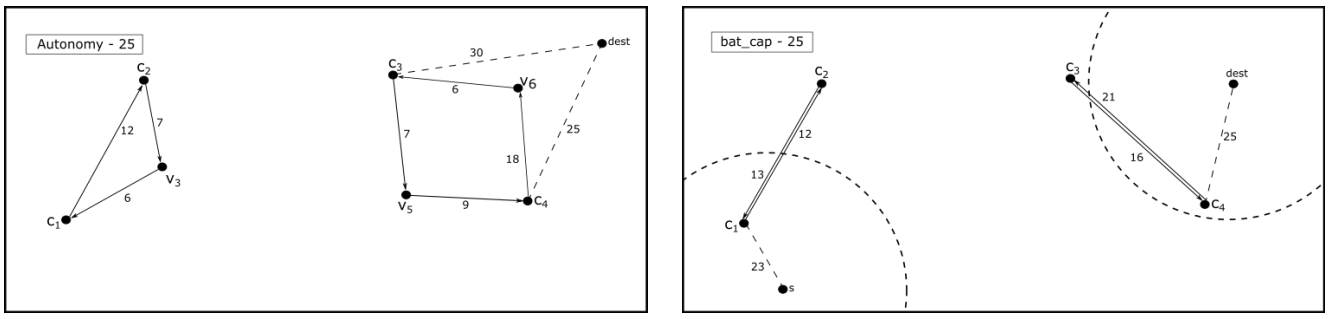



Figure 3: Road network




Extracted meta-graph (a) Adding **s** and **d** (b)

Figure 4: Meta-graph construction




Extracted meta-graph (a) Adding **s** and **d** (b)

Figure 5: Meta-graph construction

After finding the longest common critical paths, they will represent the *Candidate Set* for *Skyline* computation (*Section 4.3*).

# 5. EXPERIMENTS

The experimental part of this paper has two subsections. The first one introduces the method used for generating a synthetic dataset. The tests performed on synthetic dataset using the algorithms presented in *Chapter 4* and results are presented in the second subsection.

## 5.1 Charging route

Even if the number of EVs is increasing more and more, it is still difficult to get good, reliable/precise data from them. In order to perform experiments, a synthetic dataset needs to be generated.

The equivalent of an actual EV historical route, is a syn-

thetically created path in the above mentioned dataset. This path represents a charging route, meaning that at least one of the vertices traversed from source ***s*** to target ***dest***, is a charging station $c \in C_s$ and more, the EV stops here for charging its battery. It is to be observed that EV's battery capacity influences directly the size of the subset of vertices representing charging stations.

For the simplicity of generating synthetic data, traffic conditions are not taken into consideration and as a consequence, EV's speed to traverse an edge is maximum speed, based on road type. Another aspect that needs to be mentioned is that EV's battery is charging only when it stops at a charging station.

Another aspect that was considered when generating synthetic data, was the initial state of charge (SOC) of EV's battery. SOC is defined as the remaining capacity of a battery.

$$SOC = \frac{bat\_lvl}{bat\_cap} \tag{1}$$

where *bat_lvl* represents the charging level of battery and *bat_cap* represents the battery capacity.

Generating charging routes is a complex operation that requires high amount of resources (CPU, memory). The resources' consumption is proportional with the size of the road network (w.r.t. number of existing vertices) and the various paths along it. Reducing the number of computations will directly affect the execution time and the consumed resources. How can the number of calculated paths be optimized/reduced?

Since we are generating charging routes, this process relies on the position of charging stations on the roadmap. If we can reduce the set of vertices $V$ to a set comparable in size with the set of charging stations $C_s$, the number of possible calculated paths decreases as well.

The idea of constructing feasible paths may give us a solution to the matter presented above.

A feasible path, denoted as $p$ is defined as the fastest path from source $s$ to target *dest*, and the power consumed $pow(p)$ traversing it, does not exceed EV's battery remaining capacity:

$$pow(p) \leq bat\_lvl, \text{ where } bat\_lvl = SOC * bat\_cap$$

All feasible paths have the following property:

***Property 2.***: A path constructed out of feasible paths is also feasible.

The goal of reducing the number of possible paths to be calculated in order to generate the charging routes, can be achieved with the use of feasible paths.

A new, directed graph $G'(C_s, E')$ is precomputed, where $C_s \subset V$ denotes the set of charging stations and $E'$ denotes the set of edges in $G'$. This will be referred as meta-graph and its edges as meta-edges.

A meta-edge between two charging stations $c_i$ and $c_j$ is defined: $e_m(c_i, c_j) \in E'$ and has three attributes $e_m(c_i, c_j) = (t, pow, path)$, where $t$ - time to traverse the meta-edge, *pow* - power consumed traversing it, and *path* is a feasible path from $c_i$ to $c_j$, described by the sequence of vertices between them. The time calculated for traversing a meta-edge includes also the time spent by an EV charging its battery.

As mentioned before, the number of operations performed, in terms of path calculated for generating charging routes, can be reduced using the meta-graph. Constructing it in precomputation phase ensures apriori knowledge by holding information on the fastest route network between all charging stations. We have therefor only these unknown variables: the starting and ending points of a charging route and how are they connected to the "charging stations mesh".

### 5.1.1 Meta-graph

In this subsection we will describe the operations performed during meta-graph construction. It is reminded that the goal of these operations is finding how the vertices $c \in C_s$ are inter-connected through meta-edges.

During first step, repeated A* algorithm discovers all feasible paths between $\forall c \in C_s$ in $G(V, E)$ w.r.t. $\nexists c \in C_s$ as intermediary points. The sequence of vertices describing these paths are memorized. The time to traverse these paths and the power consumed along are also computed. The result of the first step is a set of meta-edges.

$$pow(p(c_i, c_j)) \leq bat\_lvl, \forall c_i, c_j \in C_s$$

The second and last step is finishing to build the meta-graph $G'(C_s, E')$. Continuing from step one, where we obtained the first meta-edges, Floyd-Warshall algorithm [6] is performed to discover all feasible paths between all charging stations that are not already connected. During this operation the attributes required for constructing new discovered meta-edges (time, power consumed, path) are also computed. As an observation at this step, a feasible path may contain $c \in C_s$ as intermediary vertex.

For a better understanding of how the meta-graph is constructed, we use the example shown in *Figures 3, 4, 5.*

Let's assume that *Figure 3* is the road network having 4 charging stations and the battery capacity of EV *bat_cap*=30. The first step is to find all fastest paths connecting all 4 charging stations, with regards to the fact that the power consumed to traverse them does not exceed the battery capacity restriction. The results obtained are the following:

$(c_1, c_2)$ *with pow = 12*
$(c_2, v_3, c_1)$ *with pow = 13*
$(c_2, v_4, c_3)$ *with pow = 27*
$(c_3, v_5, c_4)$ *with pow = 16*
$(c_4, v_6, c_3)$ *with pow = 21*

At this step is not possible to have other paths, since the power consumed to traverse them exceed the battery capacity restriction (for example $pow(R_f(c_1, c_3)) = 39 > 30$)

The graph presented in *Figure 4(a)* illustrates how the first discovered meta-edges are constructed. The result for first step of constructing the meta-graph is than:

- vertices : $c_1, c_2, c_3, c_4$
- meta-edges:

$e_1(c_1, c_2) = (x_1, 12, (c_1, c_2))$
$e_2(c_2, c_1) = (x_2, 13, (c_2, v_3, c_1))$
$e_3(c_2, c_3) = (x_3, 27, (c_2, v_4, c_3))$
$e_4(c_2, c_4) = (x_4, 34, (c_2, v_4, v_5, c_4))$
$e_5(c_3, c_2) = (x_5, 32, (c_3, v_5, v_4, c_2))$
$e_6(c_3, c_4) = (x_6, 16, (c_3, v_5, c_4))$
$e_7(c_4, c_2) = (x_7, 35, (c_4, v_5, v_4, c_2))$
$e_8(c_4, c_3) = (x_8, 21, (c_4, v_6, c_4))$

The rest of meta-edges are added to meta-graph in the last step. By running Floyd-Warshall algorithm, they are computed and the meta-graph is completed (*Figure 4(b)*):

$e_9(c_1, c_3) = (x_9, 39, (c_1, c_2, v_4, c_3))$
$e_{10}(c_1, c_4) = (x_{10}, 46, (c_1, c_2, v_4, v_5, c_4))$
$e_{11}(c_3, c_1) = (x_{11}, 45, (c_3, v_5, v_4, c_2, c_1))$
$e_{12}(c_4, c_1) = (x_{12}, 48, (c_4, v_5, v_4, c_2, c_1))$

As seen in *Figure 4* and *Figure 5*, a different battery capacity value leads to a different meta-graph.

### 5.1.2 Charging route construction

Having the precomputed metagraph, finding desired charging routes becomes trivial. Both, source **s** and target **dest** are added as artificial vertices to the meta-graph with a battery capacity restriction/boundary - instead of calculating paths to all charging stations, we search only within points' range regions.

A more precise requirement for source and destination points of a charging route is that they are within a reachable range (w.r.t. to power consumed) to at least a charging station. In other words, there exists at least one feasible path from both of them to a/multiple charging station(s).

Another factor, deviation, also influences a charging route. In general, it is preferable to obtain a route with a minimum deviation. The "perfect" choice has to be made in the case of more than one charging station existing within a reachable distance to source or/and destination. When this situation occurs, all feasible path combinations from source to destination are computed and the route that has the minimum deviation is chosen.

During construction of meta-graph/meta-edges, the fastest routes between all charging stations are also calculated. This precomputation reduces the execution time of obtaining the preferred route, as the number of variables is reduced significantly:

$$t_r(R_c(s, dest)) = \min\{(t_r(R_f(s, c_i)) + t_r(R_f(c_i, c_j)) + t_r(R_f(c_j, dest)) \mid c_i, c_j \in C_s \text{ and } c_i \neq c_j\}$$

A particular situation occurs when both, starting and ending point of the route are within a reachable range to the same charging station(s). Then, the previous relation becomes:

$$t_r(R_c(s, dest)) = \min\{(t_r(R_f(s, c_i)) + t_r(R_f(c_i, dest)) + + t_c(c_i) \mid c_i \in C_s\}$$

As presented in previous chapter, a component in calculating the time to travel from source to target, is the time spent in charging stations when charging the battery. This component is in direct relationship with EV's battery SOC: the lower the level, the longer it takes to recharge. Also, knowing the power consumed to travel along edges to reach certain vertices (for example target or a charging station closer to target), the time spent for battery charging can be optimized/minimized. For a better understanding let's take a look at the situation where both source and target are within a reachable range to the same charging station. Knowing the power consumption to arrive at destination, the remaining battery capacity after charging should be comparable with it:

$$pow(p(c, dest)) = bat\_lvl, c \in C_s$$

An example on how a charging route is constructed is illustrated in *Figure 4(b)* and *Figure 5(b)*. We are given the starting point ***s*** and destination point ***dest***. In *Figure 4(b)* ***s*** has one charging station $c_1$ within reachable range (23) and ***dest*** has $c_3$ and $c_4$ as nearest accessible charging stations (within 30, respectively 25).


Figure 6: Big map with starting and ending regions

## 5.2 Test and Results

### 5.2.1 Experiments settings

Experiments were performed on a computer with 6GB of RAM and Intel(R) Core(TM) i7-2670QM CPU @2.20GHz. Code was written in C++ using C++11 standard and core functionality was acquired with help of open source library BOOST [15].

We have chosen two different sizes maps (road networks) to generate synthetic historical data and perform tests on it.

First, a small map (central Aalborg, see *Figure 1*), which has GPS coordinates $57.04^o$ **S**, $57.055^o$ **N**, $9.938^o$ **E** and $9.9^o$ **W**. During dataset generation process no boundaries were set - starting and ending points were chosen completely random.

The second map, bigger than the first one (North East Canada, see *Figure 6*), corresponds to rectangle of GPS coordinates $55.507^o$ **S**, $56.68^o$ **N**, -118.98$^o$ **E** and -121.94$^o$ **W**. In this case, two bounding rectangles were defined (see *Figure 6*) and 70% of routes were forced to start and end within them, while the rest 30% routes were completely random. The boundaries were chosen to imitate real world travel patterns between 2 major areas on a map.

*Table 2* presents the characteristics of these 2 maps.

| **Map** | **Location** | **No. of vertices** | **No. of edges** | **Autonomy**, *(kW)* |
|---|---|---|---|---|
| Small | Central Aalborg | 394 | 75 | 250 |
| Big | North East Canada | 35769 | 5259 | 16000 |

Table 2: Map Information

Another parameter, which has significant impact on routes generation is Power Consumption Rate. The rates were taken for average size vehicle from work of [13]. It is worth to observe, that after complex study, the authors propose

rates which contradict with common intuition - the faster vehicle moves, the less power it consumes. The values used in our data generation are presented in *Table 3*.

| Speed, (km) | Power Consumption, (kW) |
|---|---|
| 30 | 172 |
| 40 | 168.5 |
| 50 | 165 |
| 60 | 161.5 |
| 90 | 151 |
| 130 | 137 |

Table 3: Consumption rates

In order to prove the efficiency of SQA proposed in this paper, the tests were performed in comparison with 2 other algorithms: brute force skyline and sort filter skyline (SFS). An overview of these algorithms is presented bellow.

***Algorithms***

**Brute force**
Brute force solution works in simple manner: first computes support and deviation for all routes, then performs primitive comparison algorithm, where calculated values are compared and Skyline routes are found.

**Sort Filter Skyline (SFS)**
This algorithm [14] is based on a similar principle as Brute force - it computes support and deviation values for all routes. Then for every route sums the values for support and deviation and the Candidate Set is sort descending on previously summed values. An updated Candidate Set is obtained, where the first route/member has biggest support and deviation and is definitely a part of the Skyline Set. An advantage of using SFS is that the sorting process helps to discard most of non-eligible candidates without going through entire Skyline Set later.

A pseudocode version of the SFS used in our experiments is presented bellow in *Algorithm 5*.

**Algorithm 5** Sort Filter Skyline Algorithm (SFS)

**Input:** *CanSet* - all routes
**Output:** *Skyline* - routes in Skyline

```
Skyline = ∅
for (i=0; i < CanSet.size; i++) do
    supportCalc(CanSet[i])
                                        ▷ Algorithm 1
    deviationCalc(CanSet[i])
                                        ▷ Algorithm 2
    canSet[i].SumVariable = canSet[i].deviation +
canSet[i].support
end for
sort CanSet on sumVariable descendingly
SimpleSkyline
                ▷ Comparison algorithm on 2 dimensions
```

### 5.2.2 Data mining - Critical Paths

As mentioned in *Chapter 2* a deeper analysis of trajectories in dataset is performed in order to find critical paths. A solution for mining for these critical paths is described in *Section 4.4*. However, this is a very complex and non trivial task to implement. Since the paper's purpose is not proposing an efficient solution mining for LCS using Suffix tree, existing solution was chosen [16].

The existing implementation works for finding LCS of 2 strings. However, the dataset analyzed in our tests, consists out of thousands routes. Therefore some adjustments were made. First, LCS search is performed for each route, concatenating it with all the rest of the data set. After this, the found LCS is checked with defined threshold values (length of a route $LCS\_length = 0.3$ and number of vertices $LCS\_vertices = 40$). If the conditions are met, the found LCS is added to a new dataset. Finally, SQA is performed.

As an observation, the mining for critical paths was performed only on the small size map.

The pseudocode version of the algorithm used for finding LCS is presented in *Algorithm 6*.

**Algorithm 6** Critical Paths(LCS) discovery

**Input:** *CanSet* - all routes
**Output:** *LCS_dataSet* - new data set

```
Skyline = ∅
for (i=0; i < CanSet.size; i++) do
    string s1 = CanSet[i].pathString
    for (j=i+1; j < CanSet.size; j++) do
        string s2 = CanSet[j].pathString
        path = LCS(s1, s2)
                ▷ Suffix tree construction and LCS search
        if (path.distance ≥ LCS_length) and
(path.pathSize ≥ LCS_vertices) then
            LCS_dataSet.push_back(path)
        end if
    end for
end for
```

### 5.2.3 Results

The following section is dedicated to presenting the results obtained performing tests with the settings mentioned previously. The results are grouped by the type of tests performed: SQA running time, skyline set size and running time for generating synthetic dataset.

***SQA Running Time***
*Figure 7* and *Figure 8* present the results obtained for SQA running time, in comparison with 2 other algorithms Brute force and SFS. SQA running time outperforms the other two on the big map, while on the small map is comparable with Brute force running time. It is observed in *Figure 8* that the larger the set of routes used to identify the skyline points, the better the performance of SQA . The algorithm proposed in this paper, SQA, identifies the skyline set in less than 3/4 of the time Brute force algorithm does.

***Data generation***
A similar trend, as seen in *Figure 7* is observed in the results obtained for performing SQA on the critical paths set *Figure 9*. It seems that for a small dataset, Brute force algorithm might be a good option.

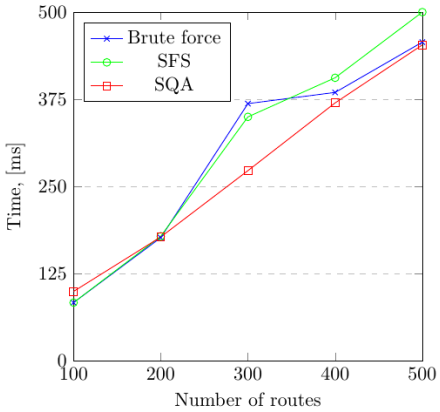



*Figure 7* Running time of SQA on a Small map

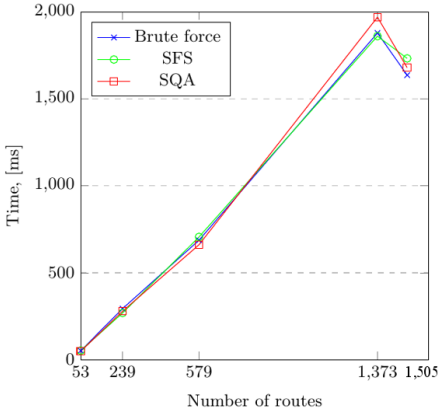



*Figure 8* Running time of SQA on critical path set

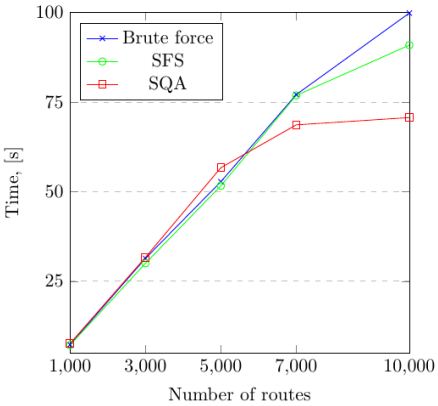
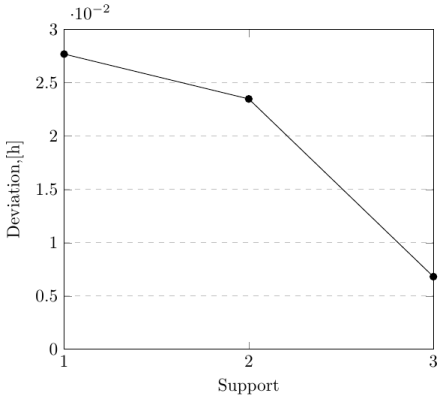
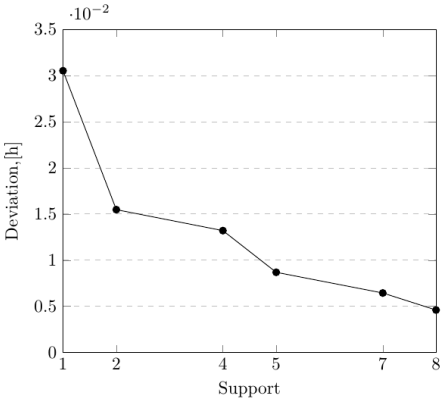


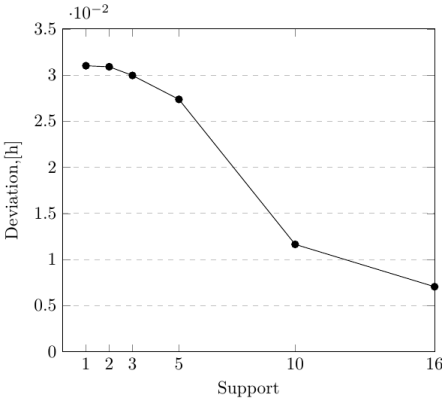

*Figure 9* Running time on big map




a) *100 routes*

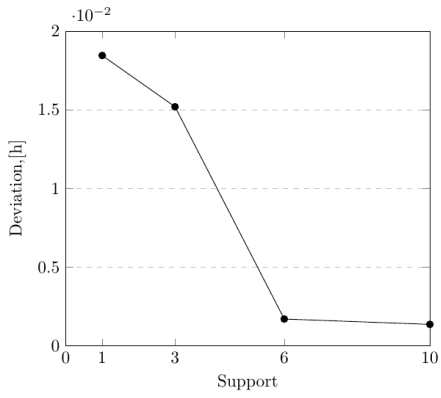



b) *200 routes*

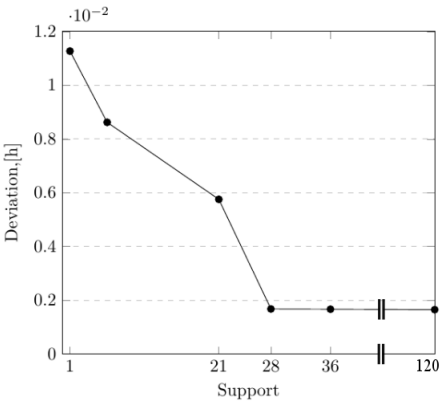



c) *300 routes*

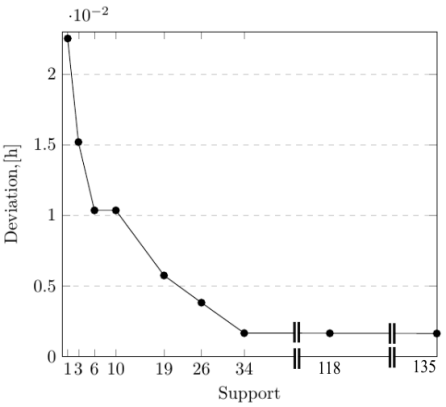
*Figure 10* Skyline points on Small map




a) *100 routes*

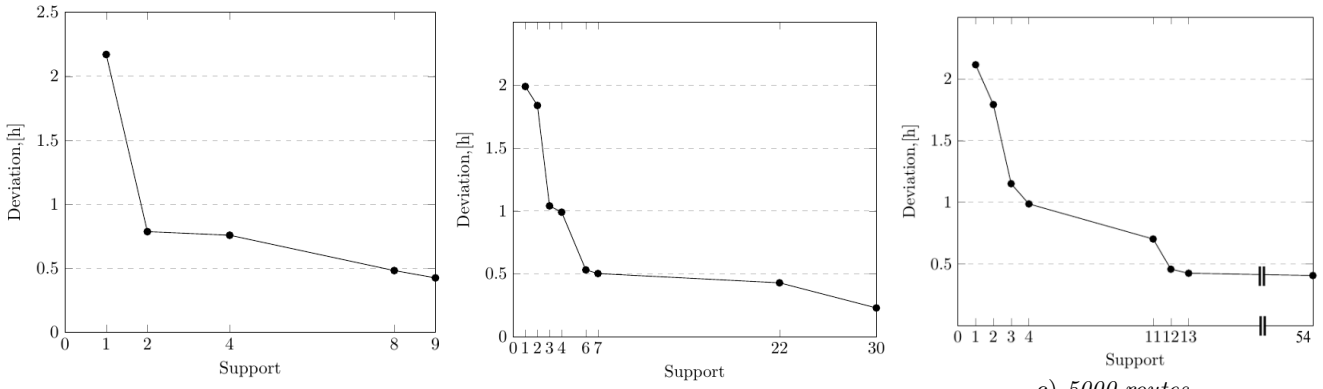



b) *200 routes*




c) *300 routes*

*Figure 11* Critical paths Skyline on Small map




a) *1000 routes* b) *3000 routes* c) *5000 routes*

*Figure 12* Skyline points on Big map

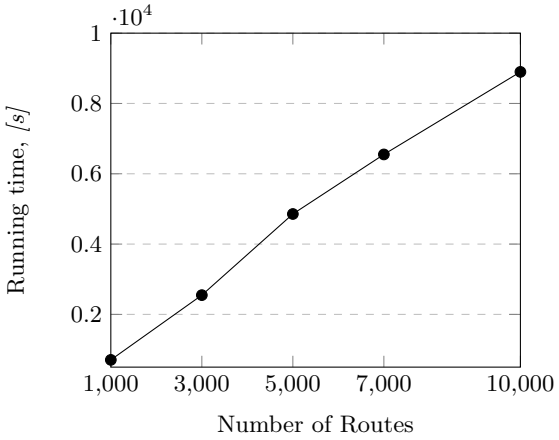



*Figure 13* Running time of Charging route creation on a big map

***Skyline Set***

*Figure 10* shows the skyline obtained for different number of routes, which were generated on the small map. It is observed that the larger the support values, the smaller deviation values. The same trend is seen in *Figure 12*, where the results presented are obtained from the dataset generated on the big map. This tendency might be a consequence of the dataset structure, in other words, the map chosen and the parameters used for generating synthetic data.

In comparison with *Figure 10*, *Figure 11* shows the Skyline set obtained from critical paths discovered from the same routes. For critical paths skyline points the values for support are larger than regular skyline points: more than 3 times larger for 100 routes, 15 times larger for 200 routes and almost 9 times larger for 300 routes. These values prove that within the analyzed dataset, there are patterns (critical paths) that might have a huge influence on deviation, if traffic conditions are to be considered.

***Data generation***

As observed in *Figure 13* data generation method proposed in this paper performs in linear time and increasing number of routes does not affect running time in exponential manner. This proofs that precomputed meta-graph idea works efficiently.

# 6. CONCLUSIONS

In this paper we proposed a framework for the analysis of EV trajectories along with a method to generate synthetic datasets.

The results have revealed that the framework is efficient in cases of analyzing large datasets. This is possible due to efficient pruning techniques used in Skyline Query Algorithm (SQA), which rely on pruning candidates on estimated values, rather than real calculated values for each dimension.

The complex design of SQA and data structure used have a negative impact on performance when analyzing small datasets.

A future development direction for the framework proposed might be analysis of timestamp related trajectories for EV. Improvements might also be made for trajectories pattern mining (finding Critical paths) by applying different strategies for solving LCS problem.

# 7. REFERENCES

[1] I.Sanporean; P.Galinauskas; D.Shkodrov (2016) *Slyline Queries Framework for Electric Vehicles.* AAU, Data Engineering Semester Project

[2] M. Vlachos; G. Kollios ; D. Gunopulos(2002). *Discovering similar multidimensional trajectories.* IEEE, Proceedings 18th International Conference on Data Engineering. p. 673 - 684.

[3] Yu Zheng (20015). *Trajectory data mining: An overview.*. ACM Trans. Intell. Syst. Technol. 6, 3, Article 29 (May 2015).

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

[5] 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.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein *Introduction to Algorithms. Third Edition*

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

[8] Stephan Borzsonyi, Donald Kossmann, Konrad Stocker. (2001)

*The Skyline Operator*. 17th International Conference on Data Engineering p. 421-430.

[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://www.igismap.com/haversine-formula-calculate-geographic-distance-earth/

[13] Kwo Young, Caisheng Wang, Le Yi Wang, Kai Strunz. (2012) *Electric Vehicle Battery Technologies*. Electric Vehicle Integration into Modern Power Networks,2013, p. 15-56.

[14] Jan Chomicki, Parke Godfrey, Jarek Gryz, Dongming Liang (2003) *Skyline with Presorting*. 19th ICDE, Bangalore, India, p. 717-719.

[15] http://www.boost.org/

[16] http://www.geeksforgeeks.org/generalized-suffix-tree-1/