Personalized Route Planning with Recharging for Electric Vehicles

01 February 2016 - 13 June 2016

Date	Mathias Højsleth
Date	Andreas Thor Lau
Date	Andreas Strandfelt

This article studies the problem of route planning for electric vehicles (EVs) in a road network. Route planning is often performed with interest in the fastest path. However, some constraints for computing the fastest path for an EV are different from the constraints involved in route planning for conventional vehicles. For example, recharging of an EV is more time consuming than refueling a conventional vehicle. Furthermore, the pricing for recharging EVs follows different standards than the pricing for conventional vehicles.

This article suggests an approach to balance travel time with price for EV route planning. This is done by taking a user-specified preference function for travel time and price as input, while taking charging and recuperation into account. The path, which, according to the preference function, is the most optimal, is returned. The work presented in this article is an extension of our previous work, which considered almost the same problem. Even though the problems are slightly different, the solution is reused in this article.

Two different query algorithms are proposed, along with the previous solution, to solve the problem of returning the optimal path according to a preference function. The first query algorithm presented makes no use of precomputation and serves as a baseline. The second query algorithm uses a precomputation technique to speed up the queries. All three query algorithms uses the principles of Dijkstra's algorithm, which is modified to compute the optimal path given a preference function. Additionally, we propose two optimization techniques to speed up the queries, namely optimal sub-structure and A* search.

At last, we perform experiments on real world data to evaluate the proposed algorithms in terms of the precomputed graph sizes and running time of the query algorithms. The experiments show that the proposed query algorithms are practical in terms of running time for smaller graphs. Additionally, the query algorithms, which make use of preprocessing, is magnitudes faster than the baseline query algorithm, which does not make use of preprocessing.

Personalized Route Planning with Recharging for Electric Vehicles

Mathias Højsleth
mhajsl11@student.aau.dk

Andreas Thor Lau
alau11@student.aau.dk

Andreas Strandfelt
astran10@student.aau.dk

PREFACE

This paper extends previous work described in [1] by the same authors. In some sections of this article, material from [1] has been reused or paraphrased. The relevant sections in this article are listed here:

The abstract has been paraphrased from [1].

In Section 3.1, the sections Section 3.1.1, Section 3.1.2, Section 3.1.3, and Section 3.1.4 have been reused from [1].

In Section 3.2, the following definitions have been reused or paraphrased from [1]: Definition 1, Definition 2, Definition 3, Definition 4, Definition 5, Definition 6, Definition 7, Definition 8.

In Section 5, Section 5.1 paraphrases material from [1].

The acknowledgements also paraphrases material from [1].

ABSTRACT

This article studies the problem of route planning for electric vehicles (EVs) in a road network. Route planning is often performed with interest in the fastest path. However, some constraints for computing the fastest path for an EV are different from the constraints involved in route planning for conventional vehicles. For example, recharging of an EV is more time consuming than refueling a conventional vehicle. Furthermore, the pricing for recharging EVs follows different standards than the pricing for conventional vehicles.

This article suggests an approach to balance travel time with price for EV route planning. This is done by taking a user-specified preference function for travel time and price as input, while taking charging and recuperation into account. The path, which, according to the preference function, is the most optimal, is returned. The work presented in this article is an extension of our previous work, which considered almost the same problem. Even though the problems are slightly different, the solution is reused in this article.

Two different query algorithms are proposed, along with the previous solution, to solve the problem of returning the optimal path according to a preference function. The first query algorithm presented makes no use of precomputation and serves as a baseline. The second query algorithm uses a precomputation technique to speed up the queries. All three query algorithms use the principles of Dijkstra's algorithm, which is modified to compute the optimal path given a preference function. Additionally, we propose two optimization techniques to speed up the queries, namely optimal substructure and A* search.

At last, we perform experiments on real world data to evaluate the proposed algorithms in terms of the precomputed graph sizes and running time of the query algorithms. The experiments show that the proposed query algorithms are practical in terms of running time for smaller graphs. Additionally, the query algorithms, which make use of preprocessing, is magnitudes faster than the baseline query algorithm, which does not make use of preprocessing.

1. INTRODUCTION

Electric vehicles (EVs) are becoming increasingly popular as a means of transportation due to its advantages over the alternative; fossil fueled vehicles. An example of the advantages of EVs is its independence of fossil fuels, and compared to fossil fueled vehicles, the fuel efficiency is magnitudes better. The cruising range of early EV models was limited, which required careful planning with focus on energy efficient routes, especially because charging stations were sparsely located. Therefore, algorithms making use of speed-up techniques were introduced to compute energy efficient routes. Some of these did not consider charging stations, as there was no guarantee of being able to reach a charging station with a fully charged battery. The use of an auxiliary graph consisting of charging stations exclusively was one of the speed-up techniques that were introduced.

In more recent models, the cruising range of EVs is comparable to fossil-fueled vehicles. As an effect of this, eco-friendly or energy-efficient routes are of less importance as EVs are already eco-friendly compared to fossil-fueled vehicles. Instead, the focus might be to reach the destination as fast as possible or as cheap as possible. Also, the network of charging stations has become more dense, enhancing the chances of being able to reach a charging station. Therefore, newer research within the domain of route planning for EVs consider visiting charging stations. However, charging stations have different charging rates and prices. The route planning problem becomes harder, as the charging rate affects the travel time, and the price of charging affects the total price of the route.

As an effect of longer cruising range and more charging stations, previous speed-up techniques, such as auxiliary graphs, becomes less efficient as more vertices (charging stations) are introduced to the auxiliary graph. Also, the number of edges between charging stations increases because of increased cruising range, i.e. being able to reach more charging stations.

The experiments from our previous article [1] shows that the auxiliary graph of Germany's motorways was of significantly larger compared to the actual graph of Germany's motorways. The main reason for the auxiliary graph be-

ing that size, was because of the problem investigated. The problem was to find Pareto-optimal paths that minimized total travel time and price, i.e. several edges could exist between two charging stations, as long as the edge weights did not dominate each other. Also, the cruising range of the newer EVs allowed the possibility of reaching more charging stations compared to older EVs. This made the auxiliary graph more interconnected and dense.

The running time experiments for query answering showed that EVs with a large cruising range made the algorithm impractical for two reasons. The first reason being that all Pareto-optimal paths had to be found and returned, which takes more time compared to only finding one path. Also, the auxiliary graph was significantly larger in size, compared to the original graph, meaning that it was not that much of a speed-up in the end, being the second reason for slow running times.

Therefore, this article considers efficient bi-criterion route planning for EVs, where the goal is to find one route, which is the best according to a user-defined preference function. The preference function contains d preference parameters α_i, where d is the number of costs and α_i is the preference parameter for cost i. The conditions for the preference parameters are as follows: $0 \leq \alpha_i \leq 1$ and $\sum_{i=1}^{d} \alpha_i = 1$. To compute the *preference cost* of a path using a preference function, the sum of each cost in the path is multiplied with the according preference parameter. The path with the smallest preference cost is returned as it is the most optimal path according to that preference function. E.g. two paths with bi-criteria costs; $p_1 = (5, 5)$ and $p_2 = (6, 2)$. Given preference function $\alpha = (0.5, 0.5)$, the preference cost for p_1 is $(5 * 0.5) + (5 * 0.5) = 5$ and the preference cost for p_2 is $(6 * 0.5) + (2 * 0.5) = 4$, hence p_2 is preferred given that preference function. Given another preference function $\alpha = (1, 0)$, the preference cost for p_1 is $(5 * 1) + (5 * 0) = 5$ and the preference cost for p_2 is $(6 * 1) + (2 * 0) = 6$, hence p_1 is preferred given that preference function.

As the edge count of the auxiliary graph from the previous solution is very high, we investigate two other route planning techniques, which aims to reduce the number of edges in the query graph. The first makes use of no preprocessing techniques and is inspired by Dijkstra's algorithm. The second technique is a preprocessing technique, which makes use of elements from the customizable route planning approach.

We propose two algorithms and a modification to the algorithm from our previous solution described in [1]. The two new algorithms are a baseline algorithm and a more time-efficient algorithm, which makes use of a precomputed partitioned graph. The previous solution is also more time-efficient than the baseline approach, and makes use of a precomputed auxiliary graph. Both algorithms utilizing precomputation lies within the personalization framework. Experiments performed on the algorithms show that the algorithms are practical for smaller graphs. Furthermore, the experiments show that the two algorithms making use of precomputation are magnitudes faster than the baseline algorithm.

This article is structured as follows. Section 2 describes related work applicable for this area of research. Section 3 describes the prelimininaries, including the assumptions we will make throughout the article along with definitions related to the problem. The problem statement is also presented in Section 3. In Section 4, the algorithms used to solve the problem, are described. This includes a basic algorithm, which does not use preprocessing, along with a more time-efficient algorithm, which uses preprocessing. Also, a description of the modifications needed for our previous solution from [1] is given in Section 4. Experiments on the algorithms are presented in Section 5, which also describes the speed-up techniques used. Finally, a conclusion of the article is given in Section 6, while Section 7 discusses the article and future work.

2. RELATED WORK

The section describes new related work in the area of route planning for EVs. Also, preprocessing techniques that may improve the running time of route planning algorithms for EVs are described.

2.1 Shortest Feasible Paths with Charging Stops for Battery Electric Vehicles

In [2], Baum et al. solves the bi-criteria pathfinding problem of minimizing the total travel time and time spent on charging at charging stations. The weights of the edges are driving time and energy consumption, where energy consumption can be negative due to recuperation. To respect battery constraints, i.e. that the state of charge is always non-negative and is always less than or equal to the battery capacity, a consumption profile is defined, which determines the minimum and maximum state of charge that is required to traverse a path. If the path is not feasible, i.e. the path violates the battery constraints, the consumption profile returns infinity.

Charging stations with different charging rates are also modeled in the solution under the assumption that some charging functions exist. It is possible to leave a charging station at any point, if the combined travel time to the target becomes slower because of further charging at the charging station. This allows for an infinite amount of different charging configurations, where all configurations result in different paths. To avoid this, a label is kept for each vertex. The label is a 4-tuple, which represents all trade-offs between the state of charge gained and the time spent on charging, for the last visited charging station.

The label for a vertex v stores the total travel time from source vertex to v. Also, the state of charge of the latest visited charging station u is stored. The third element of the tuple is the last visited charging station u. By using this labeling approach, there is no fixed state of charge associated with a vertex, as it depends on how much is charged at u. Therefore, the consumption profile is used for the sub-path from u to v and stored as the fourth element of the label.

When a new charging station is visited, the consumption profile is evaluated by fixing the charging time at u. Thereby, the total time and state of charge can be determined and updated at the new charging station's label. However, the same problem about infinite charging configurations exists, resulting in an infinite amount of labels. It is proven by the authors that only a small finite number n of charging configurations is necessary to represent all non-dominated paths; resulting in n labels at the newly visited charging station.

To speed up the queries, A* search and contraction hierarchies are considered. The average running time for the query algorithm, which combines A* search and contraction hierarchies is approximately 86 seconds for the road network

of Europe using a Peugeot iOn with a fictive battery of 85 kWh to simulate newer models of Tesla. Also, heuristics are considered to minimize the running time. The general idea is to keep only one label at each visited vertex. Which label to keep is determined by the heuristic, i.e. the path which has the highest state of charge or lowest travel time, which decreased the running time with a factor of 1.8 and 3.2 respectively.

Baum et al. provide a non-trivial model and solution to the problem of computing bi-criteria paths. Some of the concepts, which are used in [2], are also used in this article. Some of these include the Dijkstra-like expansion, the usage of two different sets of settled and unsettled labels at each vertex and dominance checks with breakpoints.

Because the problem is NP-hard, assumptions to make the problem easier has to be made. In [2], the amount of energy to charge at the last visited charging station is decided when a new charging station or the target vertex is reached by the use of a consumption profile. In this article, the amount of energy to charge, at the last visited charging station, is the amount of energy that is needed in order to reach the next charging station or target vertex.

The two criteria in [2], i.e. total travel time and time spent on charging, is not working against each other as the time spent on charging is a part of the total travel time. Therefore, the problem assessed in this article of finding a path that minimizes the total travel time and price is a different challenge since the two criteria work against each other as argued in Section 3.3.

2.2 Customizable Route Planning

In [3], Delling et al. consider the problem of computing a multi-criteria shortest path from a starting point to a destination point. This problem has been studied extensively and Dijkstra's algorithm is a well-known solution for this problem. However, on larger road networks, Dijkstra's algorithm has a running time of several seconds. Therefore, preprocessing techniques have been introduced to minimize the query answering time with a trade-off of a longer preprocessing phase and larger space-consumption. Several preprocessing techniques like hub labeling and contraction hierarchies are very efficient wrt. query answering time.

Whenever metrics in the road network changes, the preprocessing phase needs to be recomputed before queries can be run. The disadvantage of the previous preprocessing techniques are long preprocessing times, e.g. if metric changes happen frequently, the preprocessing techniques are not viable in real-time systems and are not a speed-up compared to Dijksta's algorithm. Therefore, the focus is to speed-up the preprocessing time under the assumption that metric changes happen to make preprocessing viable in real time systems.

The solution to the problem is called Customizable Route Planning (CRP) and has three activites; construction of a metric independent overlay, customize metrics and query answering.

2.2.1 Metric-independent overlay

The goal for the metric-independent overlay activity is to partition the original graph and construct an overlay graph which consists of *boundary vertices* and *overlay edges*. Two types of overlay edges exist, namely *boundary edges* and *shortcut edges*. However, in order not to confuse the terminology used in this article, boundary vertices and boundary edges will be referred to as *border vertices* and *border edges*, respectively, since these are two different names covering the exact same vertices and edges.

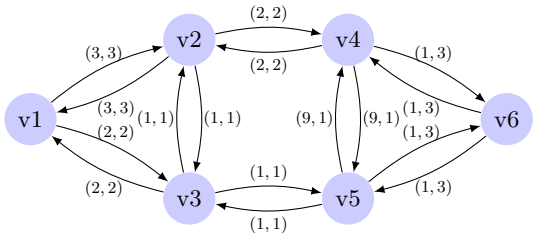

Figure 1: Example of a road network with bi-criteria costs

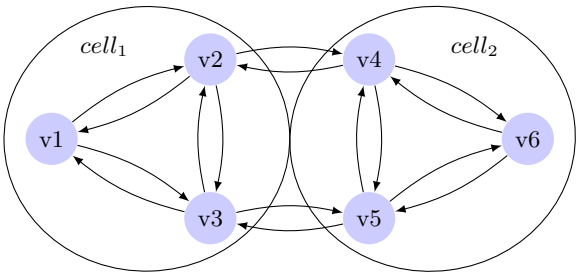

Figure 2: Example of a partitioned graph using Figure 1 as input. $v1$, $v2$ and $v3$ are vertices of $cell_1$. $v3$, $v4$ and $v5$ are vertices of $cell_2$

To construct the partitioned graph, the original graph is divided into *cells*, such that each vertex in the graph is contained in exactly one cell.

A border vertex is added to the overlay graph if a vertex has an edge in the partitioned graph, which links to a vertex from a different cell. The edge, which connects the two border vertices, is added as a border edge to the overlay graph. Shortcut edges are added in the overlay graph between all pair of border vertices within the same cell. This activity is called metric-independent because it considers only the topology of the original graph, where changes happen infrequently. Therefore, Delling et al. argues that this activity is not of big concern wrt. optimization of space and time because it is precomputable and executed frequently.

An example of an input graph for partitioning the graph and constructing an overlay graph can be seen in Figure 1, where an edge represents two distinct costs. Keep in mind, the costs does not matter at this stage, only the topology. The output of this stage can be seen in Figure 2 and Figure 3. Vertices $v1, v2$, and $v3$ are part of $cell_1$ while vertices $v4, v5$, and $v6$ are part of $cell_2$. Where $v2$ and $v3$ are border vertices of $cell_1$, $v4$ and $v5$ are border vertices of $cell_2$. The edges between $v2$ and $v4$, and $v3$ and $v5$ are border edges as they connect two vertices from different cells. The edges between $v2$ and $v3$, and $v4$ and $v5$ are shortcut edges as they connect two vertices from the same cell.

2.2.2 Metric customization

The goal of the metric customization activity is to update the edge costs of the partitioned graph and the overlay

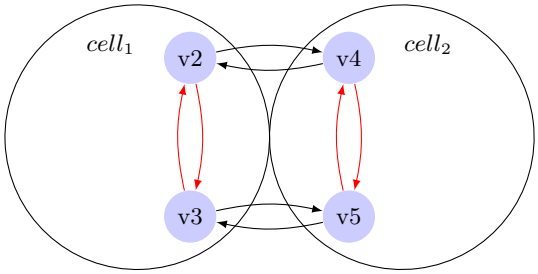

Figure 3: Example of an overlay graph using Figure 1 as input. $v2$ and $v3$ are border vertices of $cell_1$. $v4$ and $v5$ are border vertices of $cell_2$. The red edges between $v2$ and $v3$, $v4$ and $v5$ are shortcut edges, while the black edges between $v2$ and $v4$, $v3$ and $v5$ are border edges

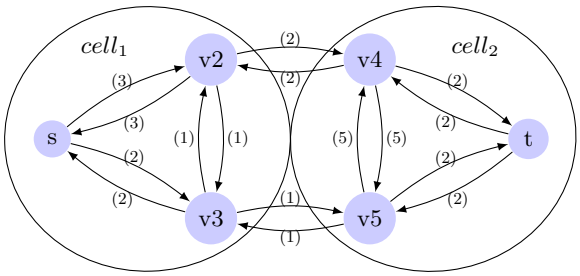

Figure 4: Example of a customized partitioned graph with bi-criteria costs using preference parameters $\alpha = (0.5, 0.5)$

graph, if metric changes has happened, for instance a different set of metric preferences. This activity affects the running time, as it depends on the preference parameters, which are given at query time by the user.

First, the edges of the partitioned graph are updated using the cost function, which makes use of the preference parameters. Afterwards, the overlay edges are updated. The border edges are customized with the same edge cost as the edge from the partitioned graph. The shortcut edges are customized with the sum of the shortest path cost between the border vertices in the partitioned graph.

Because metrics or preferences to metrics can potentially change for each query, this activity might be executed for each query. However, all the heavy computation is done in the metric-independent overlay activity. Therefore, this activity is fast (only a few seconds on continental sized road networks) compared to the metric-independent overlay activity.

An example of metric customization for the graph shown in Figure 1 is seen in Figure 4. Each edge in the partitioned graph is customized by multiplying the cost of the edge with the preference parameters, which in this case is $\alpha = (0.5, 0.5)$. E.g. the edge from $v4$ to $v5$ has a cost of $(9, 1)$. Therefore, the customized cost is $(9 * 0.5) + (1 * 0.5) = 5$.

Afterwards, the overlay graph needs to be customized with costs. The customized costs from Figure 4 are propagated directly to the border edges as shown in Figure 5. The costs for the shortcut edges are propagated with the preference

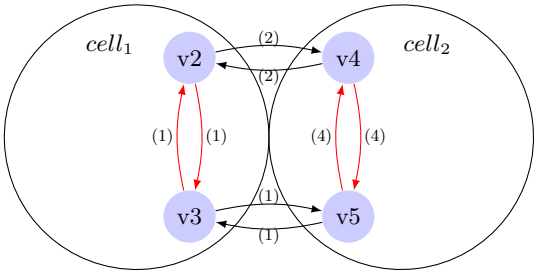

Figure 5: Example of a customized overlay graph propagating the customized edge values for the border edges and shortest path values for the shortcut edges from Figure 4

cost of the shortest path between the border vertices. E.g. the shortcut edge from $v4$ to $v5$ in Figure 4 is a shortcut from $v4$ to t and from t to $v5$ resulting in a preference cost of 4.

2.2.3 *Query answering*

The input to the query answering activity is a partitioned graph, an overlay graph, preference parameters, along with source and target vertices. The goal is to find the optimal path, according to the preference parameters, from the source to the target. If the preference parameters changed since the last query, the metric customization phase has to be run again.

Otherwise, the query algorithm is a simple Dijksta's algorithm with a small modification in order to handle the overlay graph. If the source and target vertices are not border vertices and assigned to different cells, the query algorithm starts in the partitioned graph, but limited to the cell where the source vertex is located.

When a border vertex is dequeued from the cell where the source vertex is located, only overlay edges of the overlay graph are considered. When a border vertex, which is not located in the same cell as source and target vertices, is dequeued, only edges from the overlay graph are considered. When a border vertex is dequeued from the cell where the target vertex is located, overlay edges of the overlay graph and edges in the partitioned graph leading to a vertex of the same cell as where the target is located, are considered. When a non-border vertex from the cell where the target vertex is located, is dequeued, only edges from the partitioned graph leading to a vertex of the same cell as where the target is located, are considered, until target is found. It is possible to have multiple overlay layers, but the query algorithm behaves similarly.

The example makes use of Figure 4 and Figure 5 where the query is from vertex s to vertex t using preference parameters $\alpha = (0.5, 0.5)$. Vertex s is enqueued and dequeued immediately. Then, $v2$ and $v3$ are enqueued and s is closed. The algorithm dequeues $v3$ (as it has lower preference cost than $v2$) and notices that $v3$ is a border vertex. The outgoing overlay edges lead to $v2$ and $v5$. The cost of the path leading to $v2$ is not better than what is already stored at $v2$ and is not updated. $v5$ is enqueued as it has not been visited before.

Now, $v2$ is dequeued; the explanation is skipped as this process is similar to what was just explained when $v3$ was de-

queued. Therefore, $v5$ is dequeued by the algorithm. Since $v5$ is a border vertex of the cell where t is located, the algorithm considers both overlay edges of the overlay graph and edges from the partitioned graph leading to vertices located in the same cell as t. The overlay edges lead to $v3$ and $v4$, but the cost of the paths are greater than what is already stored and is not enqueued. The outgoing edges of the partitioned graph lead to $v4$ and t. The cost of the path to $v4$ is greater than what is already stored and is not updated. t is enqueued as it has not been visited before. Now, t is dequeued (since it has the lowest cost in the queue) and the algorithm terminates.

2.2.4 *Discussion*

CRP is an interesting approach as it is fast in answering queries for dynamic metrics. Also, CRP shares similar features compared to the solution from [1] in the sense that an overlay graph is created, which consists of only some vertices, from the original graph, where shortcut edges connect the overlay vertices. Obviously, this approach is not directly applicable to the problem of route planning for EVs, because of the constraints of EVs, i.e. modeling charging stations, such that the EV can charge when running out of energy.

2.3 Customization vs. Personalization

In [4], Funke et al. argues that if the preference parameters change for each query, CRP has to run the customization phase for each query, which results in almost the same running time as a plain Dijkstra. Therefore, Funke et al. propose a framework called personalization, to speed-up query answering time compared to a plain Dijkstra.

The customized framework, which CRP belongs to, consists of two phases; a preprocessing phase and a query phase. The preprocessing phase has one activity; construction of the metric-independent overlay. The query phase has two activities; metric customization and query answering. Likewise, the personalized framework consist of similar phases. However, the preprocessing phase has two activities; construction of the metric-independent overlay and assigning cost vectors. The query phase has one activity; query answering. Funke et al. shows how to convert existing customized approaches, i.e. customized contraction hierarchies and CRP to personalized approaches, i.e. personalized contraction hierarchies and personalized route planning.

The construction of the metric-independent overlay of the personalized framework is the exact same process, as in the customized framework.

The second activity is the main difference of the two frameworks. The customized framework assigns the single-valued best cost according to the metric preferences for each overlay edge, where the metric preferences is given at query time. The personalized framework assigns several cost vectors for each overlay edge, one for each simple path in the partitioned graph that lead from the source to the target vertex of the overlay edge. As this activity does not depend on any metric preferences, this activity is precomputable.

The output of a personalized overlay graph is shown in Figure 6 using the road network from Figure 1. The shortcut edge in Figure 6 from $v4$ to $v5$ has two cost vectors because two paths (limited to the cell) from $v4$ to $v5$ exist in Figure 1. The direct path from $v4$ to $v5$ has a cost of $(9, 1)$, resulting in a cost vector of $(9, 1)$. The second path passing trough $v6$ has a total cost of $(1, 3) + (1, 3) = (2, 6)$ resulting in a cost vector of $(2, 6)$.

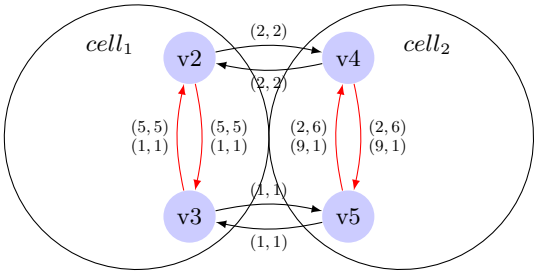

Figure 6: Example of a personalized overlay graph propagating the weights directly for border edges and shortest path weights for the shortcut edges from Figure 1

The query answering activity for the customized framework is simple, as it only has to consider one cost for each overlay edge. When an overlay edge is relaxed in the personalized framework, the query algorithm has to apply the preference parameters to all cost vectors, to determine which cost vector is best. E.g. the shortcut edge from $v4$ to $v5$ in Figure 6 has two cost vectors i.e. $c_1 = (9, 1)$ and $c_2 = (2, 6)$. Given preference parameters $\alpha = (0.5, 0.5)$, the optimal path can be determined by multiplying the cost vector with the preference parameters. In this example, the preference cost of the paths evaluate to: $c_1 * \alpha = 5$ and $c_2 * \alpha = 4$. Therefore, c_2 would be chosen based on the given preference parameters.

The number of cost vectors on an overlay edge affects the running time of the query algorithm. Therefore, Funke et al. expands the second activity with pruning techniques to remove cost vectors that will never be part of the optimal solution. This lowers the amount of cost vectors that needs to be considered at query time. By applying the pruning techniques on Figure 6, the cost vectors $(5, 5)$ attached to the edge from $v2$ to $v3$ could be pruned as that path never would be preferred over the other path which has a cost vector $(1, 1)$.

2.3.1 *Discussion*

It is obvious that advantages and disadvantages are existing wrt. the customization and personalization frameworks. Therefore, choosing one of the frameworks is dependent on which domain the application is applied. E.g. choosing the customization framework in a domain where the application is used by lots of users would not be a good choice, as there is a potential of a change in the preference parameters for each query. When the preference parameters change, the metric customization activity has to be executed again, which takes additional time in order to answer the query. Instead, the personalization would be a good choice as it handles different preference parameters without adding additional time to the query answering time.

On the other hand, if the domain consists only of a single user, i.e. the preference parameters rarely change, the metric customization activity in the customization framework is skipped. Also, at most, only one overlay edge exists for each pair of overlay vertices in the overlay graph, which ensures fast query answering times compared to the personalization framework. In the personalization framework, several over-

lay edges can exist and no advantage wrt. query time can be taken given consecutive queries with same preference parameters. Therefore, the customization framework would be a good choice for fast query answering times in such a domain.

3. PRELIMINARIES

This section describes the assumptions made in this article. Furthermore, definitions of concepts needed to solve the problem are presented along with the problem statement.

3.1 Assumptions

This section describes the assumptions, which will be used to either simplify the problem or reflect a realistic use of the solution.

3.1.1 Speed Assumption

One assumption we will make, is that one always drives at the speed limit of the current road. This will, in our opinion, please more to the users of electric vehicles as they would not be forced to drive a lot slower than the rest of the road users.

This results in a less complex problem, as the speed at which to drive on an edge is predefined. This makes it possible to calculate e.g. energy consumption. This assumption is also seen in [5] and [6].

3.1.2 Charge Assumption

We will also make the assumption, that when one charges at a charging station, one only charges the amount of energy required to reach the next charging station or target location. This can, however, result in a more expensive path, as one could possibly have charged more at a cheaper charging station in exchange for charging less at a more expensive charging station. Furthermore, it can result in a slower path, for similar reasons.

The reason for this assumption is that the problem becomes less complex; we do not have to consider at which charging stations it would be most optimal to charge.

3.1.3 Charging Price Assumption

In addition to charging stations, we assume that there exists a charging price function *cp*, which outputs the price of charging a certain amount of energy at a charging station. The function is assumed to be increasingly monotonic, such that charging more energy will never result in a lower price.

3.1.4 Charging Time Assumption

We also assume there exists a charging time function *ct*, which outputs the time it takes to charge a certain amount of energy at a charging station. The function is assumed to be increasingly monotonic, such that spending more time charging never results in a lesser charged battery. Also, the charging rate drops, when the battery is closer to being fully charged, to avoid battery damage. Therefore, *ct* is concave down increasing, such that a battery that is closer to a full charge can never be charged faster than a battery with lower battery charge, for the same charging time function.

3.2 Definitions

This section presents definitions of the concepts needed to understand and solve the problem of bi-criterion route planning for EVs.

Definition 1. *Let a directed graph $G = (V, E)$ consist of a finite set of vertices V, connected by a finite set of directed edges E.*

Definition 2. *Let a vertex v = (long, lat, ele, CS_{rate}, CS_{own}) consist of a pair of coordinates representing the location of the vertex in the real world, where long denotes longitude and lat denotes latitude, along with ele denoting the elevation of the vertex above sea level. In the case where the vertex represents a charging station, let CS_{rate} denote the highest available charging rate, and CS_{own} denote the owner of the charging station.*

A vertex in the graph can represent two different cases. The first case is that the vertex is simply a point at the end of a road, or connecting two or more roads, and the fields CS_{rate} and CS_{own} are not available. In the second case, the vertex represents a charging station, where an EV can regain charge at the cost of time. The ratio between charge and time is denoted CS_{rate}.

Definition 3. *Let an electric vehicle $EV = (b, SoC, CS_{sub})$ consist of a battery with a battery capacity denoted b and the current state of charge be denoted SoC, such that $0 \leq SoC \leq b$. Let CS_{sub} be a set containing subscriptions to charging station owners.*

Definition 4. *Let a charging price function $cp(CS_{sub}, c_{curr}, c_{req}, v)$ given the EV's CS_{sub}, output the price for charging c_{req} kWh of energy at the charging station v, when c_{curr} kWh of energy has already been charged.*

Let a charging time function $ct(b, SoC_c, c_{req}, v)$ output the charging time for charging c_{req} kWh of energy at the charging station v, given a battery capacity b, the SoC when v was reached SoC_v and c_{curr}, such that $SoC_c = SoC_v + c_{curr}$.

The total price of charging at v is seen in Equation (1). The function to find the price of charging one kWh is denoted $CS_p(CS_{own}, CS_{sub})$ and takes as parameter $v.CS_{own}$ and information on subscriptions from the EV.

Some charging station owners charge a constant price per charging station visit, no matter how much is charged. This is denoted as a function called p_{charge}, which is dependent of the charging station owner and the subscription to the given charging station.

The functions *ct* and *cp* are used in the algorithms, which use backward planning wrt. charging. Therefore, *cp* needs to handle incremental charging, such that the charging fee is not added each time the algorithm decides to charge more at the charging station.

As seen in Equation (2), the charging rate is treated in three different ways, which is dependent of SoC_c. Charging in the interval from $0 - 80$ % of the battery capacity is a linear process and no penalty is added. A 20 % penalty of the charging rate is added, when charging in the interval from $80 - 90$ % of the battery capacity. Charging in the interval from $90 - 100$ % of the battery capacity adds a charging rate penalty of approximately 43 %. Equation (2) is inspired by Zündorf's charging function from [5], but is formatted such that it fits the needs of this problem. Zündorf's charging function outputs the amount of energy charged given a charging time, which is the opposite of the function *ct*.

Definition 5. *Let a directed edge $e = (v_f, v_t, dt, ec)$ connect vertex v_f to v_t. Let the weight dt be a floating point*

describing the time it takes to drive the distance between v_f and v_t and the weight ec be a floating point describing the energy consumed by driving the distance between v_f and v_t in a specific EV. The weights of an edge are denoted as $w(e) = (dt, ec)$.

An edge in the graph represents a road in the real world road network. The weight ec is specific to an EV, chosen when creating the graph G. The *cost* of an edge is a pair of floating points representing total travel time and price, defined as $c(e) = (tt, price)$, where $tt = dt + ct(b, SoC_c, ec, v)$ and $price = cp(CS_{sub}, c_{curr}, ec, v)$. Variable v denotes the last visited charging station. Note that the cost $c(e)$ is $(dt, 0)$ when no charging station has been visited.

Definition 6. *Let a path $p = (V, E)$ consist of a finite sequence of vertices V connected by a finite sequence of edges E. The weight of p is a pair of floating points representing driving time and total consumed energy, defined as $w(p) = (dt, ec) = \sum_{e \in E} w(e)$. The cost of a path is defined as $c(p) = (tt, price) = \sum_{e \in E} c(e)$*

Definition 7. *For a path $p = \{\{v_1, v_2, ..., v_k\}, \{e_{v_1,v_2}, ..., e_{v_{k-1},v_k}\}\}$, the set of charging stations present in the path $CS = \{cs_1, cs_2, ..., cs_n\}$ where $CS \subseteq p.V$ and an EV ev with an initial state of charge $ev.SoC$, the state of charge at a vertex $v_j \in p$, where $1 \leq j \leq k$, is defined as:*

$$ev.SoCA_{v_j} = \begin{cases} ev.SoC, & \text{if } j = 1 \\ ev.SoCD_{v_{j-1}} - w(e_{v_{j-1},v_j}).ec, & \text{otherwise} \end{cases} \quad (3)$$

$$ev.SoCD_{v_j} = \begin{cases} ev.SoCA_{v_j} + ec_{v_j,v_k}, & \text{if } v_j = cs_n \\ ev.SoCA_{v_j} + ec_{v_j,cs_{i+1}}, & \text{if } v_j = cs_i \\ ev.SoCA_{v_j}, & \text{otherwise} \end{cases} \quad (4)$$

where $ev.SoCA_{v_j}$ and $ev.SoCD_{v_j}$ is the state of charge when arriving at and departing from a vertex v_j and ec_{v_x,v_y} is the sum of the energy consumption of the edges between a charging station v_x and a vertex v_y.

Definition 8. *Let a path $p = \{\{v_s, ..., v_t\}, \{e_{v_s,v_{s+1}}, ..., e_{v_{t-1},v_t}\}\}$ be feasible for an EV ev with maximum battery capacity $ev.b$ iff:*

$$\{\forall v \in p \mid 0 \leq ev.SoCA_v \leq ev.b \wedge 0 \leq ev.SoCD_v \leq ev.b\} \quad (5)$$

3.3 Problem Statement

Transportation is most often preferred to happen as fast as possible, while still being as cheap as possible. The arguments for both would be that one would have time and money to spend on other activities. However, these two criteria often work against each other, as the fastest route is often the most expensive, and vice versa. In relation to electric vehicles, a vehicle consumes more energy, the faster one drives, and thereby becomes more expensive. We aim to to find a fast and cheap path between two points for electric vehicles. We will do this by finding the optimal path from a source to a target, where the dimensions are the total travel time, including recharging at charging stations, and the cost of the path in terms of payment for charging en route, by taking user preferences wrt. travel time and price into account. The user preferences are given at query time and may change for each query. This makes it possible for people to personalize the path found, such that the path suits them the most, be it fast or cheap. To accommodate longer routes, we will consider recuperation as well as charging at charging stations. The problem we will solve can be formulated as follows:

For a predefined directed graph $G = (V, E)$, where V is a set of vertices and E is a set of edges, an electric vehicle ev, and a set of charging stations $CS \subseteq V$, given a source and target vertex $s, t \in V$, and a preference function $\alpha(\alpha_t, \alpha_p)$ where $0 \leq \alpha_t, \alpha_p \leq 1$ and $\alpha_t + \alpha_p = 1$, the problem is to:

- Compute a feasible path p for ev where $c(p) * \alpha(\alpha_t, \alpha_p)$ is lowest of all feasible paths from s to t.

Furthermore, we will investigate different methods of solving the problem in order to measure if one method outperforms other methods in terms of query answering time.

First, a query algorithm, which makes use of no precomputation, is investigated. The reason of using no precomputation is because most precomputation techniques require some preprocessing at query time. For instance, source and target vertices have to be connected to the auxiliary graph in [7], which affects the running time of the query algorithm. By avoiding this preprocessing state the query algorithm is executed directly on G, in the expectation of fast query answering times or at least a benchmark.

Afterwards, the feasibility of using an auxiliary graph, similar to what Storandt proposed in [7], is investigated. The purpose of using an auxiliary graph is to speed up the queries by reducing the size of the graph.

At last, we investigate a second precomputation method, which purpose is to construct a partitioned graph by adopt-

$$cp(CS_{sub}, c_{curr}, c_{req}, v) = \begin{cases} CS_p(v.CS_{own}, CS_{sub}) * c_{req} + p_{charge}(v.CS_{own}, CS_{sub}) & \text{if } c_{curr} = 0 \\ CS_p(v.CS_{own}, CS_{sub}) * c_{req} & \text{otherwise} \end{cases} \quad (1)$$

$$ct(b, SoC_c, c_{req}, v) = \begin{cases} \frac{c_{req}}{v.CS_{rate}}, & \text{if } SoC_c + c_{req} < b * 0.8. \\ \frac{b*0.8 - SoC_c}{v.CS_{rate}} + ct(b, b*0.8, -(b*0.8 - SoC_c - c_{req}), v), & \text{if } SoC_c < b * 0.8. \\ \frac{c_{req}}{v.CS_{rate} * \frac{4}{5}}, & \text{if } SoC_c + c_{req} < b * 0.9. \\ \frac{b*0.9 - SoC_c}{v.CS_{rate} * \frac{4}{5}} + ct(b, b*0.9, -(b*0.9 - SoC_c - c_{req}), v), & \text{if } SoC_c < b * 0.9. \\ \frac{c_{req}}{v.CS_{rate} * \frac{4}{7}}, & \text{if } b * 0.9 \leq SoC_c. \end{cases} \quad (2)$$

ing elements from [3]. Again, the aim is to reduce the search space and also limiting the interconnection of edges between vertices in the partitioned graph.

4. ALGORITHMS

This section describes the algorithms suggested for solving the problem of bi-criterion route planning for EVs. First off, a basic approach is presented in Section 4.1. In this section, many of the concepts used for solving the problem will be described. Afterwards, the choice of framework for the solutions, which makes use of preprocessing, is made. The basic approach is used as a baseline in Section 4.3, where a more efficient algorithm will be described. At last, the modifications needed for the solution from [1], to be able to solve the problem of this article, is presented in Section 4.4.

4.1 GQueryAlgorithm

This section explains the *GQueryAlgorithm* seen in Algorithm 1. The algorithm is designed to solve the problem described in Section 3.3 without any form of preprocessing by using the same expansion technique as Dijkstra's algorithm. The algorithm is label-correcting because it keeps a two-dimensional skyline of paths, where the dimensions are the preference cost and the reach wrt. battery constraints. When a path visits a charging station, the algorithm decides how much should be charged at that charging station, when either the target vertex is found or when another charging station is visited, i.e. the charge assumption described in Section 3.1.2. It performs queries directly on the graph G defined in Definition 1. The algorithm takes as input the graph G, an electric vehicle ev, the source and target vertices s and t, and a preference function α. The output of the algorithm is the cost of the best path from s to t according to the preference function. First, the data structures are explained, followed by the algorithm itself.

4.1.1 Data structures

As seen on line 2 of Algorithm 1, the algorithm uses a queue Q and six different temporary variables. An overview of the variables can be seen in Table 1. The queue Q is a min-heap used to track which vertex is next in the iterations of the algorithm. Variable u is the current vertex that is being processed. The temporary variables is explained later in this section.

A vertex in the graph contains two sets, *settled* and *unsettled*, holding paths from the source vertex to that vertex, similar to [2]. The set *settled* is an unordered list of already expanded paths. The set *unsettled* is a min-heap, sorted by the path with lowest preference cost, containing paths, which has yet to be expanded. When a vertex is dequeued, each path in *unsettled* is checked for dominance against the paths in *settled*. If the path is not dominated, it is removed from *unsettled*, added to *settled* and expanded. If the path is dominated by any path in *settled*, it is skipped. Since the algorithm is label-correcting, a vertex may be visited several times. If only one set was used to store paths at a vertex, the same path could potentially be expanded several times, which is unnecessary.

Paths are added to *unsettled* by the vertex member function *addLabel*. A label corresponds to a path reaching the vertex, i.e. several labels can exist at a given vertex. The function *addLabel* returns a boolean depending on whether the new path added to *unsettled* has a lower cost than the first label in *unsettled*, i.e. the label with the lowest cost in *unsettled*. If *addLabel* returns true, the vertex of the label is enqueued in Q. Obviously, the label with lowest preference cost is always preferred at the target vertex, i.e. the first label in *unsettled*. However, on an intermediate vertex of the path from source to target, the label with the lowest preference cost, may run out of energy before the target is reached. Another label with a higher preference cost, but less energy consumed, may be able to reach the target and therefore be preferred over the label with lowest preference cost at an intermediate vertex of the path. Therefore, all labels in *unsettled* have to be expanded. The order of which labels are picked at first is important, as expansion of dominated labels should be avoided.

The label or path added by the member function *addLabel* is of type *vLabel*. An overview of the variables needed to create a *vLabel* can be seen in Table 2. A *vLabel* consists of eight different values: *lastCSId*, *cost*, *SoC*, *time*, *price*, *CSSoC*, *charge* and *reservedSoC*. Variable *lastCSId* is the ID of the last visited charging station, i.e. the charging station where the vehicle has the possibility of charging in order to proceed along the path. Variable *cost* is the preference cost of the path, while *time* and *price* is the amount of time and the price for reaching the vertex, including charging. Variable *SoC* is the state of charge of the vehicle at the vertex, where the label is stored. *CSSoC* is the state of charge when arriving at the last visited charging station. Keeping track of this value ensures the correct state of charge is used, when charging at the last visited charging station. The variable *charge* denotes how much has already been charged at the last visited charging station. Variable *reservedSoC* denotes how much battery capacity that is reserved for recuperation along the path since the last visited charging station. Variable *reservedSoC* is equal to $SoC - charge$ of the label, if that value is greater than *reservedSoC* of the label from the previous vertex. Remember, *charge* is the amount of energy that has already been charged in order to reach the latest vertex in the path. Therefore, the battery capacity that was occupied in the battery from *charge* can be used for recuperation. An example of this is if $SoC = 0$, $charge = 7$, $reservedSoC = 0$ and 4 kWh is recuperated when traversing an edge. The new labels *SoC* will then become $0 + 4 = 4$. *reservedSoC* will remain unchanged because the 4 kWh that is recuperated can be stored in the battery that was previously occupied from charging, i.e. $SoC - charge = 4 - 7 \leq 0$. Assume that 5 kWh is recuperated on an edge immediately afterwards, then the new labels *SoC* will then become $4 + 5 = 9$. Because $SoC - charge = 9 - 7 = 2 > 0$, the battery capacity that was occupied in the battery from *charge* is not enough to hold the recuperated energy. Therefore, additional battery capacity has to be reserved, i.e. *reservedSoC* on this label is set to 2.

The sum of *charge* and *reservedSoC* ensures that the battery constraints are not violated. For example, if the battery capacity of the *ev* is 50 kWh, *charge* is 20 kWh, and *reservedSoC* is 25 kWh, it is not preferable to additionally charge more than 5 kWh. The reason for this is that somewhere along the path, energy is recuperated and this recuperated energy would be wasted, if more than 5 kWh is charged, because *SoC* at that point would equal the battery capacity. Therefore, the condition: $charge + reservedSoC \leq ev.b$ has to hold.

The variables *altCost*, *tSoC*, *tTime*, *tPrice*, *tCharge* and

Variable	Description
Q	Heap
u	Current vertex
altCost	Preference cost of the path from vertex s to $e.v_t$, where e is the relaxed edge from u
tSoC	The state of charge after traversing the relaxed edge
tTime	Total time spent charging and traversing the edges from s to $e.v_t$, where e is the relaxed edge from u
tPrice	Total price for charging from s to $e.v_t$, where e is the relaxed edge from u
tCharge	Total amount of energy charged at last visited charging station
tReservedSoC	The amount of energy that is reserved for recuperation in the battery from last visited charging station to $e.v_t$, where e is the relaxed edge from u

Table 1: Variables used by Algorithm 1

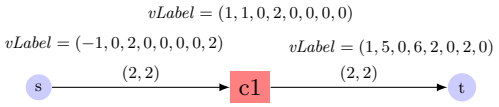

Figure 7: Example of *vLabel* at three different vertices given preference function $\alpha = (0.5, 0.5)$. Charging time and charging price are assumed to be $1:1$ wrt. the energy spend.

tReservedSoC, seen in Table 1, are floating points used by the algorithm for storing temporary values. The temporary values are used for the construction of a *vLabel* at the reached vertex, when an edge is relaxed. Variable *altCost* is the preference cost of the path, which is used to check whether the current path is better than any of the paths stored in the visited vertex according to the preference function. Variable *tSoC* holds the SoC after traversing an edge, and is negative if more energy is required to traverse the edge, than what is already in the battery. Variables *tTime* and *tPrice* holds the total travel time the total price, respectively, from source to the current visited vertex. The variable *tCharge* holds the total amount of energy to be charged at the last visited charging station in order to traverse the current edge. Finally, *tReservedSoC* holds the amount of energy that is reserved in the battery for recuperation.

4.1.2 Example of vLabel

The intuition of how the *vLabel* in the GQueryAlgorithm works, is illustrated in Figure 7. The figure represents a graph with the vertices source s, charging station $c1$ and target t. The weight of the edges are driving time and energy consumption, respectively. For simplicity, the time spent on charging equals the energy spent in this example. Similarly, the price spent on charging equals the energy spent. In this example, the preference function is $\alpha = (0.5, 0.5)$.

Initially, the member function of vertex s is called as follows:
$s.addLabel(-1, 0, 2, 0, 0, 0, 0, 2)$. The first parameter is -1 since no charging station has been visited yet. The third parameter denotes an initial SoC of 2 kWh of energy. The last parameter denotes the reserved capacity in the battery, i.e. $SoC - charge = 2 - 0 = 2$. The *vLabel* is added to *unsettled* of vertex s and *addLabel* returns true since it is the label with lowest preference cost in *unsettled*. Therefore, s is enqueued to the priority queue along with the preference cost of the *vLabel*.

Afterwards, vertex s is dequeued from the priority queue and each label in *unsettled* of s are examined. The *vLabel* is currently located in *unsettled* and has to be checked for dominance against all the labels in *settled*. However, settled is empty, therefore, the *vLabel* is removed from *unsettled*, added to *settled* and expanded. Now, all outgoing edges are relaxed. Vertex s only has one outgoing edge that leads to the charging station $c1$. Notice that $c1$ is reachable with the current state of charge. Therefore, the member function of vertex $c1$ is called as follows:
$c1.addLabel(1, 1, 0, 2, 0, 0, 0, 0)$. The first parameter is the ID of $c1$ because $c1$ is a charging station and the second parameter is the preference cost: $(2*0.5)+(0*0.5) = 1$. The third parameter is 0, since the state of charge was spent to reach $c1$, and the fourth parameter is 2, because two time units are spent on reaching $c1$. The sixth and eighth parameter is set 0 as the state of charge is 0, when the charging station was reached. The *vLabel* is added to *unsettled* of vertex $c1$ and *addLabel* returns true since it is the label with lowest preference cost in *unsettled*. Therefore, $c1$ is enqueued to the priority queue along with the preference cost of the *vLabel*.

Afterwards, vertex $c1$ is dequeued from the priority queue and each label in *unsettled* for $c1$ are examined. Since *settled* for $c1$ is empty, the *vLabel* in *unsettled* for $c1$ is removed from *unsettled*, added to *settled* and expanded. Vertex $c1$ has only one outgoing edge leading to the target vertex t. Vertex t is not reachable with the current state of charge from $c1$ because all of the energy was spent to reach $c1$. Fortunately, a charging station has been visited, i.e. $c1$, and 2 units of energy are to be charged at $c1$. Therefore, the member function of vertex t is called as follows:
$t.addLabel(1, 5, 0, 6, 2, 0, 2, 0)$, where the second parameter is the total preference cost from s to t: $1+(4*0.5)+(2*0.5) = 5$. The fourth parameter is the total time spent on charging and driving: $2+2+2 = 6$. The fifth parameter is the total price spent on charging: 2. The seventh parameter is the amount of energy, which has to be charged at the recently visited charging station $c1$: 2. Since *reservedSoC* of the previous *vLabel*, from $c1$, is 0, the eighth parameter is set to 0, as $0 - 2 = -2$ is not greater than 0. The *vLabel* is added to *unsettled* of vertex t and *addLabel* returns true since it is the label with lowest preference cost in *unsettled*. Therefore, t is enqueued to the priority queue along with the preference cost of the *vLabel*.

Afterwards, vertex t is dequeued from the priority queue

Variable	Description	Property
lastCSId	ID of last visited charging station.	if no CS visited: $lastCSId = -1$ otherwise: ID of last visited CS
cost	Preference cost of the path from vertex s to $e.v_t$, where e is the relaxed edge.	$cost \geq 0$
SoC	State of charge after traversing the relaxed edge.	$SoC \geq 0$
time	Total time spent on charging and driving along the edges from s to $e.v_t$, where e is the relaxed edge	$time \geq 0$
price	Total price of charging from s to $e.v_t$, where e is the relaxed edge	$price \geq 0$
CSSoC	State of charge when arriving at the last visited charging station.	$CSSoC \geq 0$
charge	Total amount of energy charged at last visited charging station.	$charge \geq 0$
reservedSoC	The amount of energy that is reserved for recuperation in the battery from last visited charging station to $e.v_t$, where e is the relaxed edge.	$reservedSoC \geq 0$

Table 2: Variables used to create a *vLabel*

and the algorithm returns the first label in *unsettled* of vertex t, i.e. the label with the lowest preference cost.

4.1.3 Optimal Sub-Structure

To make the algorithm more time efficient, a technique to prune labels, which are never part of an optimal solution, needs to be used. As stated earlier, each vertex keeps a set of unsettled labels. A label should only be settled, if it is not dominated, i.e. may be be part of an optimal solution.

To determine whether a label is dominated, the preference cost and energy consumed is checked against the labels in the set of settled vertices. At first glance, this would seem to be sufficient in order to guarantee correctness and time efficiency.

Given an example of three *vLabels*: $l_1 = (1, 5, 0, 5, 5, 0, 3, 1)$, $l_2 = (2, 4, 0, 4, 4, 0, 4, 0)$ and $l_3 = (2, 6, 0, 6, 6, 0, 6, 0)$ at the same vertex. The preference parameters are: $\alpha = (0.5, 0.5)$ for this example. The preference cost, *cost*, for l_1, l_2 and l_3 are 5, 4 and 6, respectively. The energy consumed, $charge + reservedSoC$, for l_1, l_2 and l_3 are $3+1 = 4$, $4+0 = 4$ and $6+0 = 6$, respectively. Following the concept of optimal sub-structure, labels l_1 and l_3 would be dominated because l_2 has a lower or equal preference cost and energy consumed. However, the last visited charging station that l_1 made use of is different from l_2. For this example, the target vertex is reachable for the labels, but 5 kWh needs to be charged and it takes 5 time units to reach target, charging included. The charging station l_1 made use of has a charging fee of 5$ and 0$ for every kWh charged. The charging station l_2 made use of has a charging fee of 0$ and 1$ each kWh charged. Expanding l_2 to the target vertex would result in following *vLabel*: $l_{2t} = (2, 9, 0, 9, 9, 0, 9, 0)$, since 1$ has to be paid for each of the 5 kWh charged. Assume that label l_1 was not pruned. The expansion of l_1 to the target vertex would result in following *vLabel*: $l_{1t} = (1, 7.5, 0, 10, 5, 0, 8, 1)$, since the 5$ charging fee has already been paid as charging had already occurred. Label l_{1t} would be preferred over l_{2t} at the target vertex. This example focused only on dominance wrt. the charging price, however, similar examples can be made with charging time, and also the combination of price and charging time. Therefore, checking dominance with preference cost and energy consumed at the current state alone does not guarantee correctness.

The solution, which guarantees correctness and time efficiency, is to check dominance in two dimensions, i.e. preference cost and energy consumed, not only at the current state, but also if the label was expanded further, i.e. more energy consumed. Infinitely many charging configurations for future labels exist and to check all of them is inefficient and impossible. However, it is sufficient to check dominance for a limited number of future labels, called breakpoints. Since the charging time function *ct* is concave and piecewise linear with a charging penalty at 80 %, 90 % and 100 % SoC, the first three future labels are when the label has only 20 %, 10 % and 0 % left to charge at the last visited charging station.

In Figure 8, a visualization of the breakpoints given two *vLabels* are shown, where both labels are stored at the same vertex u. The battery capacity is 50 kWh for this example. The first label has a reach of 50 kWh, i.e. it can use an additional 50 kWh of energy, be it from charging or energy residing in the battery, and its last visited charging station has a charging rate of 7 kW. The second label has a reach of 35 kWh, i.e. 15 kWh has already been charged, and its last visited charging station has a charging rate of 12 kW. Currently, the first label has a faster time than the second label because the second label has spent time on charging 15 kWh to reach the current vertex. Therefore, the first label is not dominated wrt. total travel time. Obviously, the second label has a worse time compared to the first label currently. However, at the second label's first breakpoint, the dominance wrt. total travel time changes. Therefore, the second label is not dominated wrt. total travel time.

An example, focusing on how to find the breakpoints, is now given, where the two labels l_1 and l_2 from the first example in this subsection are used. To check whether l_1 is dominated by l_2, the preference cost of the current label l_1 has to be checked against l_2, which is $5 > 4$, i.e. dominated. Now, the breakpoints for the future labels have to be computed. For this example, the battery capacity is 50 kWh. To determine how much should be charged to reach the first breakpoint at 80 % for l_1, the currently occupied battery capacity of l_1, *charge* and *reservedSoC*, is subtracted from the 80 % of the battery capacity, i.e. $toCharge_{l1} = 40-3-1 = 36$. To get the reach, i.e. how much energy can be spent, the *SoC* for l_1 is added to $toCharge_{l1}$ such that $reachBreakpoint_{l1} = 36 + 0$. Now, the future label for l_2 has to be computed. To determine how much to charge in order to reach the same breakpoint as l_1, the *SoC* that l_2 has needs to be subtracted from $reachBreakpoint_{l1}$, i.e. $toCharge_{l2} = 36 - 0 = 36$. Then, the time it takes to charge 36 kWh added with the price of charging 36 kWh, including

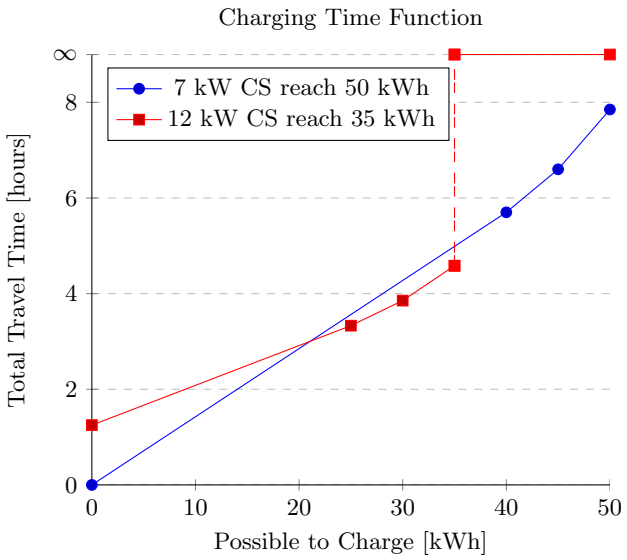

Figure 8: Breakpoints for the charging time function given two labels with different last visited charging stations

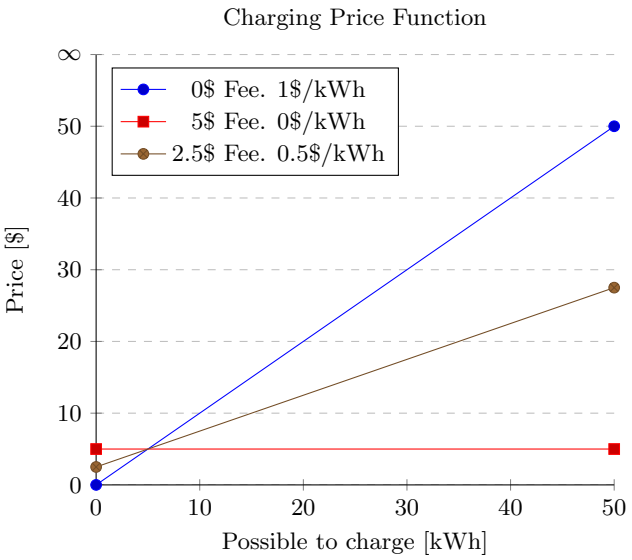

Figure 9: Breakpoints for the charging price function given three labels with different last visited charging stations

the preference function, is added to the current preference cost, i.e. $futureCost_{l1} = 36{*}0.5{+}5{*}0.5{+}5 = 24.5$. Similar is done for label l_2, i.e. $futureCost_{l2} = 36{*}0.5{+}36{*}0.5{+}4 = 40$. Since $24.5 < 40$, label l_1 is not dominated by label l_2 and the dominance checking can stop because l_1 may be part of an optimal solution later.

If label l_1 was dominated by label l_2 for the breakpoint at 80 %, the breakpoints for 90 % and 100 % *SoC* should also be checked. If l_1 was still dominated after checking the breakpoints for 90 % and 100 % *SoC* for l_1, three additional dominance checks is done, examining the breakpoints of l_2 for 80 %, 90 % and 100 % *SoC*. If l_1 was dominated in these seven dominance checks, the label could be pruned.

For this example, the preference cost is the only thing that has been focused on. As stated earlier, energy consumed should also be used when checking for dominance. Fortunately, it is incorporated directly in the breakpoints as shown in Figure 8. Examining the breakpoint at 80 % *SoC* for the first label requires 40 kWh to be charged. The second label has a reach of only 35 kWh and would have to break the battery constraints to reach the breakpoint. Because the battery constraints have to be respected, the charging time function of the second label returns ∞ at that breakpoint. For that reason the second label is dominated by the first label at that breakpoint.

The six future breakpoints are based on the charging time function *ct*, but that includes only time. The price function can be checked by only two future breakpoints because it is linear; 100 % *SoC* for the first label and 100 % for the second label. Fortunately, those breakpoints are already included in the six future breakpoints and do not have to be checked once again. An example of three labels with different last visited charging station, which have different pricing, can be seen in Figure 9. The figure proves that it is sufficient to check for dominance at the current labels and the 100 % *SoC* breakpoint.

The theory about future dominance checks and breakpoints is inspired by the work of [2].

4.1.4 Algorithm

The query algorithm can be seen in Algorithm 1. The algorithm works in two loops within a while loop, which continues until the heap is empty. One loop iterates all the unsettled labels stored at the dequeued vertex, while the inner loop iterates all outgoing edges of the vertex. A label is expanded only if it is not dominated by any other label in the set of settled labels of the vertex. Intermediate values keeping track of costs and energy consumption are then calculated for each expanded label and the label is moved from the set *unsettled* to the set *settled*. Remaining or recuperated energy is also taken into account here. At last, the new label is calculated and added to the the set of unsettled labels of the target vertex. If the label has the lowest preference cost of all unsettled labels at the target vertex, the vertex is enqueued to the heap.

On line 14-15 in Algorithm 1, the dominance check, described in Section 4.1.3, is performed for the first label in *unsettled* against all the labels that already have been settled. The label is discarded if the label is dominated by one of the labels in *settled*. Otherwise, if the label is not dominated, the label has to be expanded and moved to *settled*.

On the lines 16-33 in Algorithm 1, the cost of driving along the edge is added to the total cost of the expanded path. First, the SoC is updated along with how much charging is required at the previous charging station to reach the next vertex. Then, a check is performed to ensure the battery constraints are satisfied; it should not be possible to charge more energy than the battery can store. After this check, the time and price, including charging if necessary, for driving to the next vertex, is computed by the functions *ct* and *cp*, respectively. *ct* and *cp* are defined in Definition 4. Charging function *ct* takes as parameters the battery capacity, the sum of the already charged energy, the SoC when the charging station was encountered, the amount to charge

and the charging station to charge at. Charging function *cp* takes as parameters the subscriptions of the *ev*, the amount of already charged energy, the amount to charge, and the charging station to charge at.

On the lines 34-50 in Algorithm 1, the new *vLabel* is computed. If the label has a lower preference cost than the labels stored in the *unsettled* set of the vertex, the vertex is enqueued to the heap. First off, the preference cost and the SoC for the *vLabel* is computed. The check for is done to carry over excess energy, while avoiding carrying over energy to be recharged. Excess energy could come from either an initial SoC or from recuperating energy along the edge. Then, a new *vLabel* is added to the set of unsettled labels, but the label is dependent on whether the next vertex is a charging station or not. If the next vertex is a charging station, the last visited charging station is set to be the next vertex itself. This ensures that charging is performed on this charging station, when the path gets further expanded. For the last three parameters, *SoCToUse* and 0 are used. This tells future iterations what the SoC was when the expanded path arrived at the charging station, i.e. the amount of reserved energy, and that it has not charged at it yet, hence the value of 0. If the next vertex is not a charging station, the information about the last visited charging station is carried over to the new *vLabel*.

On the lines 37 and 42, an estimate for reaching the target vertex can be added to the cost of reaching the vertex, enabling *A* search*. This orders the vertex in the heap after an estimate of the cost of reaching the target vertex following this path. As long as the estimate is *admissible*, i.e. an underestimate of the true cost, A* search guarantees to find the path with lowest preference cost.

The algorithm terminates if Q is empty or the target vertex is dequeued from the heap, whereas the label with lowest preference cost in the target vertex's set of unsettled labels is returned, as seen on line 8-9 in Algorithm 1.

4.2 Choice of Framework

In Section 2.3, the two frameworks, personalization and customization, were described. This section describes how the solution proposed in this article relates to these frameworks and how it will affect the solution.

The problem statement in Section 3.3 states that the problem to be solved is to compute a feasible path for an EV with the lowest possible cost given a preference function. Following the arguments of Funke et al. in [4], which were summarized in Section 2.3, the problem assessed in this article is more likely to be solved in an efficient manner using the personalization framework rather than the customization framework. This follows from the possibility that the user preferences changes for each query, and the fact that the metric customization activity of the query phase in the customization framework would have to be performed at query time, every time the parameters of the preference function are changed. For the same case, the preprocessing of the personalization framework assigns cost vectors to the edges, making the query phase independent of the changes to the parameters of the preference function and therefore more time efficient on query time.

For the reasons mentioned above, the solution described in this paper will be contained within the personalization framework.

4.3 PartitionAlgorithm

This section introduces another solution, based on the concepts from PRP described in Section 2.3, to solve the problem described in Section 3.3. The personalization framework is reflected in this solution. Therefore, the algorithm is split in two phases, i.e. preprocessing and query answering. The goal of the preprocessing phase is to partition the graph into cells and construct an overlay graph. The goal of the query answering phase is to run queries on the output data from the preprocessing phase. Since the problem concerns EVs, the activities in the two phases have to be extended to accommodate the problem about charging.

4.3.1 Preprocessing

This section explains the preprocessing phase for the PartitionAlgorithm. The preprocessing phase takes as input the graph G and outputs a partitioned graph G_p separated into cells and an overlay graph G_o. Each cell is a contiguous collection of vertices. Each vertex in the graph is a member of one single cell. The preprocessing phase is further separated into two activities; the partitioning and weight assignment.

Metric-independent Overlay

The input for the metric-independent overlay is the graph G defined in Definition 1. The goal is to construct a partitioned graph G_p and an overlay graph G_o, similarly to PRP.

First, G is partitioned into cells such that each vertex is assigned to exactly one cell. The goal of the partition is to have a minimum number of border edges. An example of G can be seen in Figure 1. The output of this step is the partitioned graph G_p. An example of G_p can be seen in Figure 2.

Afterwards, the overlay graph is constructed, where each border edge, i.e. an edge connecting two border vertices from different cells, in G_p is added to the set of overlay edges in G_o. An example of G_o can be seen in Figure 3. The border vertices, which are connected by the border edges in G_p, are added to the set of overlay vertices in G_o. Compared to CRP and PRP, this is, up until this point, exactly the same.

To accommodate for long distance trips, each charging station vertex from G is added to the set of overlay vertices in G_o, meaning that the union of border and charging station vertices are the overlay vertices. Additionally, shortcut edges between each pair of overlay vertices within the same cell are added to the set of overlay edges in G_o.

Weight assignment

The input for the activity of assigning weights to the edges is G_p and G_o. The goal is to assign a weight to each edge of G_p and each overlay edge of G_o.

First, the edges of G_p gets the weight from the edges of G directly assigned, meaning that the weight of the edges in G_p is measured in driving time and energy consumption.

Afterwards, the border edges in G_o are assigned the same weight as in G_p, meaning that the weight of the overlay edges in G_o is measured in driving time and energy consumption. For each shortcut edge in G_o, an internal edge is constructed for each non-dominated path wrt. driving time and energy consumption, restricted to the cell, from the source vertex to the target vertex of the shortcut edge. The weight of the internal edge is the weight of the path.

Algorithm 1 GQueryAlgorithm

Require: G, ev, s, t, α

function GQueryAlgorithm(G, ev, s, t, α)
    Q, u, $altCost$, $tSoC$, $tTime$, $tPrice$, $tCharge$, $tReservedSoC$
    $Q \leftarrow \emptyset$
    $Q.enqueue(s, 0)$ ▷ Enqueue source vertex to heap
    $s.addLabel(-1, 0, 0, 0, 0, ev.SoC, 0, ev.SoC)$ ▷ Add initial label to source vertex
    while not $Q.empty()$ **do**
        $u \leftarrow Q.dequeue()$
        if $u = t$ **then** ▷ Return best path if target is reached
            return $u.unsettled.dequeue()$
        end if
        while not $u.unsettled.empty()$ **do**
            $lbl \leftarrow u.unsettled.dequeue()$
            for all edges $e \in G.E$ where $e.v_f = u$ **do** ▷ Iterate over all edges for all vLabels
                if $lbl.DominatedByAny(u.settled)$ **then continue end if** ▷ Optimal Sub-Structure
                $u.settled.add(lbl)$
                $tSoC \leftarrow lbl.SoC - e.ec$ ▷ New SoC
                if $tSoC > ev.b$ **then** $tSoC \leftarrow ev.b$ **end if** ▷ Only recuperate up to battery capacity
                $tCharge \leftarrow lbl.charge$
                if $tSoC < 0$ **then** $tCharge \leftarrow tCharge - tSoC$ **end if** ▷ Sum of how much to charge at last CS
                $tReservedSoC \leftarrow tSoC - tCharge$
                if $tReservedSoC < lbl.reservedSoC$ **then** $tReservedSoC \leftarrow lbl.reservedSoC$ **end if**
                if $(tCharge + tReservedSoC) > ev.b$ **then continue end if** ▷ Skip if next vertex is out of range
                if $(tCharge > 0) \wedge lbl.lastCSId = -1$ **then continue end if**
                $tTime \leftarrow lbl.time + e.dt$
                $tPrice \leftarrow lbl.price$
                if $lbl.lastCSId \geq 0$ **then** ▷ CS encountered
                    if $e.ec > 0$ **then** ▷ Energy required to traverse the edge
                        if $tSoC < 0$ **then** ▷ Not enough energy in battery
                            $tTime \leftarrow tTime + ct(ev.b, lbl.charge + lbl.CSSoC, -tSoC, lbl.lastCSId)$ ▷ Charge
                            $tPrice \leftarrow tPrice + cp(ev.sub, lbl.charge, -tSoC, lbl.lastCSId)$
                        end if
                    end if
                end if
                $altCost \leftarrow tTime * \alpha_t + tPrice * \alpha_p$ ▷ New Label
                if $tSoC > 0$ **then** ▷ Carry over excess energy
                    $SoCToUse \leftarrow tSoC$
                else
                    $SoCToUse \leftarrow 0$
                end if
                if $e.v_t.kW > 0$ **then** ▷ CS encountered
                    if $e.v_t.addLabel(e.v_t.id, altCost, SoCToUse, tTime, tPrice, SoCToUse, 0, SoCToUse)$ **then**
                        $altCost \leftarrow altCost + h(e.v_t, t, ev).t * \alpha_t + h(e.v_t, t, ev).p * \alpha_p$
                        $Q.enqueue(e.v_t, altCost)$
                    end if
                else ▷ Use old CS
                    if $e.v_t.addLabel(lbl.lastCSId, altCost, SoCToUse, tTime, tPrice, lbl.CSSoC, tCharge, tReservedSoC)$ **then**
                        $altCost \leftarrow altCost + h(e.v_t, t, ev).t * \alpha_t + h(e.v_t, t, ev).p * \alpha_p$
                        $Q.enqueue(e.v_t, altCost)$
                    end if
                end if
            end for
        end while
    end while
    return null ▷ Queue empty, no path for target found
end function

4.3.2 PartitionQueryAlgorithm

The query answering activity, referred to as *PartitionQueryAlgorithm*, takes as input the partitioned graph G_p and the corresponding overlay graph G_o, an electric vehicle ev, as well as a source vertex $s \in G_p$ and a target vertex $t \in G_p$.

Edge Relaxation

The PartitionQueryAlgorithm is similar to the GQueryAlgorithm, described in Section 4.1, as the computations, when an edge is relaxed, are exactly the same.

When an edge e is relaxed, each *vLabel* $uVLbl \in e.v_f.unsettled$ is removed from $e.v_f.unsettled$ and checked for dominance against each $vLabel \in e.v_f.settled$.

If $uVLbl$ is not dominated, the temporary variables are computed such that a new *vLabel*, $vLbl$, is created and $vLbl$ is enqueued to $e.v_t.unsettled$. If $vLbl$ has the lowest preference cost in $e.v_t.unsettled$, vertex $e.v_t$ is enqueued to the priority queue Q along with the preference cost of $vLbl$.

If vertex t is dequeued from Q, the algorithm returns $t.unsettled.dequeue$, i.e. the *vLabel* with lowest preference cost stored at vertex t, and terminates.

Graph Maneuvering

The difference between the PartitionQueryAlgorithm and the GQueryAlgorithm, is the graph maneuvering. The PartitionQueryAlgorithm takes the partitioned graph G_p and the corresponding overlay graph G_o as input. Because two graphs are used, careful maneuvering has to be done. Fortunately, the graph maneuvering is similar to what is described in Section 2.2.4.

Initially a *vLabel* is added to $s.unsettled$ and s is enqueued in Q. A listing of which edges are considered, when a vertex is dequeued, depending on the properties of the vertex, is defined as follows:

1. Non-border vertex from the same cell as t is located; edges of G_p are considered
2. Border vertex from the same cell as t is located; edges of G_p and overlay edges of G_o are considered
3. Border vertex not located in the same cell as s and t; overlay edges of G_o are considered
4. Border-vertex from the same cell as s is located; overlay edges of G_o are considered
5. Non-border vertex from the same cell as s is located; edges of G_p are considered

The listing is prioritized such that if s is a border vertex, then edges from G_o are considered because a border vertex in the cell where s is located has higher priority than a non-border vertex in the cell where s is located. Another example is if vertex s and vertex t are located in the same cell, then edges are chosen based on item 1 or item 2, depending on whether the dequeued vertex is a border vertex or not.

4.4 AGAlgorithm

This section summarizes the work presented in [1]. The objective of the work described in the article is to introduce an approach to answer route planning queries with a skyline considering total travel time and charging price as the dimensions. Finally, a description of the modifications made to the solution in order to solve the problem is described in this article.

The main contributions of the work are two algorithms. The first algorithm is a preprocessing algorithm meant to be executed only once before query time, and the second algorithm is a query algorithm, which will be run for every individual query at query time.

4.4.1 Preprocessing Algorithm

The preprocessing algorithm takes as input an electric vehicle ev and the original graph G, representing a road network, and creates an auxiliary graph AG, which at first contains only the vertices from G, which represent charging stations, e.g. vertices with a charging rate of more than 0 kW.

Next, edges are added to connect the vertices in AG. This is done by running skyline route planning queries in G from every charging station vertex to every other charging station vertex. Only feasible paths following the constraints of ev are considered, and no charging stops are allowed en route.

For every feasible path in the skyline between every pair of charging station vertices, an edge representing the path is inserted into AG with travel time and price as weights. This means that upon computing AG, the reachability between any pair of charging stations can be easily determined by looking up if there is an edge connecting these two vertices. If not, then the only possible way to travel between these two charging stations, is to charge at a third intermediary charging station.

4.4.2 AGQueryAlgorithm

The query algorithm takes as input both G and AG as described in the previous paragraph, an electric vehicle ev, as well as a source vertex $s \in G$ and a target vertex $t \in G$. The first step of the algorithm is to connect s and t directly to the vertices in AG. This is done by using a method similar to the one previously described when computing AG.

s is connected by running a skyline route planning query in G from s to every vertex present in AG, e.g. all charging stations in G. Then, all the feasible paths found in the skyline are added as edges to represent paths from s to each charging station vertex.

For t, the approach is different. Since reachability must be taken into account, and the current state of charge is unknown, a backwards route planning query is made from t to every charging station in G. However, since this is a backwards search and it is unknown which charging station a path will be connected to, charging price and rate is not available. Therefore, the weights for the backwards query are driving time and energy consumption instead. When a charging station is reached, these values can be used to compute the total travel time and price of the path.

Once both s and t have been connected to AG, a modified version of Dijkstra's algorithm is executed from s to t, returning a skyline of paths from s to t with two dimensions, namely total travel time and charging price.

A problem about this approach is when computing the auxiliary edges between charging stations. To compute the travel time of an auxiliary edge, the time spent charging the energy needed to traverse the auxiliary edge is added to the driving time of the edge. However, since this is done in the preprocessing, the SoC at the source of the auxiliary edge is unknown, and is therefore assumed to be 0. Since the

SoC at s is assumed to be the maximum battery capacity of the EV, the time penalties in the charging time function can be reached at an earlier point in time than computed by the preprocessing. This means that the algorithm does not guarantee the optimal solution.

4.4.3 Modifications

Experiments performed on the previous solution show that it is practical for smaller graphs with smaller battery capacities for *ev*, however, for larger, yet realistic battery capacities, the solution becomes impractical, due to the high number of edges generated in the precomputation, leading to an almost fully connected auxiliary graph. Modifications to the previous solution will be made, to be able to compare it to the new solutions presented in Section 4.1 and Section 4.3.

For this algorithm to work with preference functions, a few modifications have to be made. First off, the preference function has to be considered when connecting the source and target vertices, s and t, to the auxiliary graph AG. Instead of having skylines of paths connecting these two vertices to the reachable charging stations in the auxiliary graph, only a single path will connect the two vertices to each of the reachable charging stations. This can be done as a single cost can be calculated using the preference function and only the path with the lowest preference cost is chosen.

Additionally, the preference function has to be considered when performing queries. When the skyline of outgoing edges from a charging station is considered, only one of them will be used, i.e. the edge with the lowest preference cost. This is achieved by computing the preference cost for each edge, using the preference function, and choosing the edge with the lowest preference cost.

By using a preference function, the number of different paths leading to a vertex from the source vertex s, is reduced to one. This is the result of the preference function translating the edge weights into a single preference cost, making a skyline impossible. The reason for not using the preference function in the precomputation, is that the precomputation is independent from the queries. As the preference function is specific for each query, it is not available at the time of precomputation. Because of this, the precomputation stays unmodified.

5. EXPERIMENTS

In this section, experiments performed on the solutions introduced in this article will be described. The main objective of the experiments is to determine how useful the proposed solutions are in a realistic setting, mostly considering running times. First, the experiment setup and input data will be described. Next, the speed-up techniques using A* search for the queries is described. Finally, the experiment results are listed and analyzed.

The experiments have been performed on a Intel Core i7-2600 processor clocked at 3.4 GHz with 8 GB of DDR3 RAM. The proposed solutions are implemented in C++ using Microsoft Visual Studio 2015 compilers for Windows 64-bit, which supports most C++11 features, and some C++14 and C++17 features[1]. Additionally, features of the Boost 1.60.0 C++ library[2] are used for the implementation. For the queue implementation, a 3-ary heap has been used, since preliminary experiments showed very little, yet measurable, improvements in performance when using 3-ary heaps compared to 2, 4, and 5-ary heaps as well as a Fibonacci heap for the algorithms. The partitioning of vertices, as part of the preprocessing for the PartitionAlgorithm, is done with the help of the tool *gpmetis*, which is provided by George Karypis of University of Minnesota[3]. The objective of gpmetis is to create a partitioning of a graph into a specified number of cells, such that the number of border edges is minimized.

The first experiment is a comparison of the size of computed partitioned graphs for varying cell sizes, i.e. the number of vertices in each cell, as well as for two different EVs with different battery capacities, referred to as EV_{53} and EV_{90}. The size of the graph is analyzed using the number of vertices and the number of edges in the graph.

The second experiment performed is a comparison of the running time of the query algorithms proposed in this paper for both EV_{53} and EV_{90}. The speed-up techniques developed for these algorithms will also be evaluated, since all experiments will be performed with and without the speed-up techniques applied.

5.1 Input Data

This section describes the data, which has been used for the experiments in this paper. This includes the road network, elevation data, charging station data and EV data. It is also described how these different sources of data has been combined into single data sets.

5.1.1 Road Network data

The experiments are performed on a graph converted from OpenStreetMap (OSM) data, freely available from the Geofabrik website[4], which offers files containing OSM data dumps of geographical regions, such as entire countries and continents. The OSM data used for these experiments represents the motorways of Germany. A rendering of the data can be seen in Figure 10. Roads in the OSM data have associated max speeds, which are used as the driving speed, as described in Section 3.1.1. Roads with missing or unlimited maximum speeds have been assigned a value of 130 kph. The OSM data is converted to a road network consisting of vertices and edges instead of nodes and ways.

5.1.2 Elevation data

The road network is combined with height data from the Shuttle Radar Topography Mission (SRTM) version 4, which is freely available from the CGIAR Consortium for Spatial Information[5].

5.1.3 Charging station data

4,058 German charging stations are added to the road network using data freely available from OpenChargeMap.org[6]. Charging price data has been manually collected for five

[1] https://msdn.microsoft.com/en-us/library/hh567368.aspx

[2] http://www.boost.org/users/history/version_1_60_0.html

[3] http://glaros.dtc.umn.edu/gkhome/metis/metis/overview

[4] http://download.geofabrik.de/europe.html

[5] http://srtm.csi.cgiar.org/

[6] http://openchargemap.org/site/develop/api

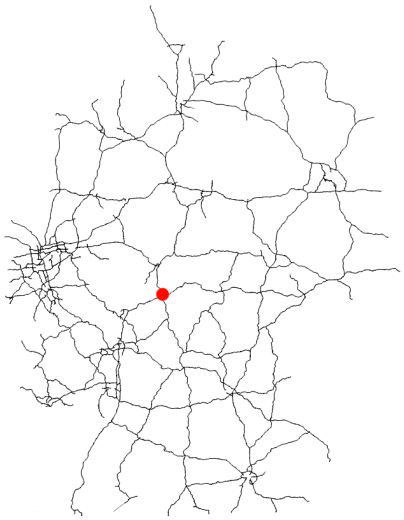

Figure 10: The road network of Germany used for the experiments. The red dot marks the reference vertex for the (x, y) coordinates used by the heuristics.

common providers in the data set, since this is not included in the data from OpenChargeMap.org. All charging stations, which are not owned by one of these providers with known associated price data, are randomly replaced with one of these providers, so that all charging stations have realistic price data. The charging stations are added as vertices and connected to the existing road network by adding edges from and to the nearest vertex in the road network.

5.1.4 Graph generation

The road network is converted to a graph matching the graph definition from Definition 1. This is done by identifying the vertices which make up intersections or end-points of roads in the road network and use these as vertices in the new graph. Roads and their intermediate nodes in the road network are then replaced by edges in the new graph, which significantly reduces the number of vertices in the graph. The weights of the edges is set to the distance of the corresponding road in the road network. Since the OSM data does not provide distances on roads, the distance of a road has to be calculated by computing the distance between the road's start and endpoints. However, latitude and longitude coordinates are available for every vertex in the graph, so the *haversine* formula is used for this purpose. This conversion results in a graph consisting of 94,993 vertices and 107,457 edges.

The auxuliary graphs created using the preprocessing algorithm described in Section 4.4.1 contains 4,058 vertices, i.e. the number of charging stations, and 4,908,531 edges for EV_{53} and 9,074,999 edges for EV_{90}.

5.1.5 EV data

The electric vehicle data used for the experiments is based on the *Tesla Roadster*, which has a battery capacity of 53 kWh. This vehicle is referred to as EV_{53}. Additionally, an imaginary EV with the same properties as the Tesla Roadster, but with a larger battery capacity of 90 kWh, is used as well and is referred to as EV_{90}. This vehicle is used to simulate the capacity of the newer *Tesla Model S*. The physical properties of the Tesla Roadster used in the road network have been retrieved from [8], [9], and [10].

5.2 Speed-up Techniques

This section describes the details of the speed-up techniques used for these experiments. As mentioned in Section 4.1.4 and Section 4.3.2, the query algorithms can receive a speed-up by applying A* Search utilizing a heuristic. For the heuristic to be admissible, the estimated cost can never be an overestimate of the true cost. The heuristic is a function h which takes as input the current vertex u, the target vertex v_t and the EV ev.

The cost of a path in this solution is the output of a cost function of price and total travel time, which depends on a preference function. Since both price and total travel time are dependent on the distance between vertices, an efficient way of estimating the distance between two vertices is needed. The obvious solution would be to use the longitude and latitude coordinates of the vertices to calculate the distance in meters using the *haversine formula*. This would guarantee that the heuristic is admissible. However, the haversine formula consists of the trigonometric functions sine, cosine and inverse tangent. Considering the number of times the heuristic will have to compute these functions during a query, the C++ trigonometric function implementation may not yield a performance gain, compared to running the query without any heuristics at all.

Instead, a more efficient way of calculating the distance is used, where a precomputation assigns a point in a two-dimensional coordinate system with (x, y) coordinates to each vertex. The (x, y) coordinates of a vertex is computed as follows: First, a reference vertex $v_r \in G$ is chosen, preferably in the center of the graph for the highest accuracy. The coordinates of v_r are set to $(0, 0)$. Then, for each vertex $u \in G$, the x coordinate is computed by using the haversine formula to compute the distance in meters from the point $(u.lat, v_r.long)$ to the point $(u.lat, u.long)$, i.e. the distance between v_r and u, if the points had the same latitude coordinate, but not the same longitude coordinate. If $v_r.long > u.long$, the distance stored in the x coordinate of u must be multiplied with -1. Likewise, the y coordinate is computed by maintaining the same longitude coordinate while using the respective latitude coordinates of v_r and u. Finally, the (x, y) coordinates of vertices can be used to estimate the distance between any two vertices using the euclidean distance formula. It should be noted that the accuracy of this approach is worsened, as the graph used covers a larger and larger geographical area. Likewise, this approach does not guarantee that the heuristic is admissible, as it may overestimate the distance. The position of v_r used for these experiments can be seen in Figure 10.

With an efficient estimate of the distance d between any two vertices in place, the estimated travel time t between u and v_t can be computed by first computing the driving time by dividing d with the maximum possible driving speed in the graph, which in these experiments is 130 kph. Next, the charging time is calculated by first calculating the energy to be charged in order to drive d at 130 kph. This is done by calculating the energy consumption from driving d and then subtracting the current state of charge. The result is then divided by the fastest charging rate in the graph. This value is 120 kW in the data used for these experiments,

Cell size	# Cells	$\|G_o.V\|$	$\|G_o.E\|$	
			EV_{53}	EV_{90}
2^1	47,497	78,842	95,916	95,916
2^2	23,749	53,578	79,027	79,027
2^3	11,875	32,693	58,129	58,129
2^4	5,938	20,108	45,359	45,359
2^5	2,969	13,467	41,077	41,077
2^6	1,485	9,943	42,481	42,481
2^7	743	7,880	49,305	49,305
2^8	372	6,523	66,822	66,822
2^9	186	5,589	99,247	99,247
2^{10}	93	4,940	188,876	188,876
2^{11}	47	4,700	450,111	450,139
2^{12}	24	4,375	1,182,330	1,184,179
2^{13}	12	4,355	2,717,322	2,875,611
2^{14}	6	4,162	6,565,115	7,308,411
2^{15}	3	4,110	15,468,270	19,311,254

Table 3: The size of the computed overlay graph G_o for an increasing cell size in G_p.

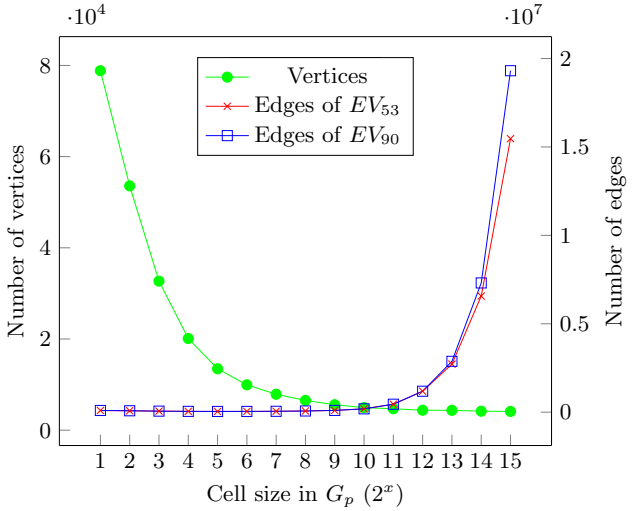

Figure 11: Visualization of the size of the computed overlay graph G_o, for an increasing cell size in G_p.

i.e. one hour of charging provides 120 kWh. The estimated travel time will then be the driving time plus the charging time. The reason for using the maximum driving speed and maximum charging speed is that it is possible for the real path to consist of only edges, where the driving speed is 130 kph, and only charging stations where the charging rate is 120 kW. Battery constraints are not considered by this function, as it is assumed charging stations is available en route where needed. Again, it should be noted that this might lead to an overestimate of the total travel time as the distance to be traversed might be an overestimate.

The estimated price p between u and v_t is found by computing the energy to be charged for driving d at the most energy-efficient speed for ev, which for these experiments is 37.69 kph. This value is vehicle specific and is found by setting the derivative of the energy consumption function equal to zero and solving for speed between 0 kph and 130 kph. The energy to be charged is calculated using the same approach as described for the travel time, but using the most energy-efficient speed instead of the maximum speed. The reason for using the most energy-efficient speed when estimating price, is that the heuristic should return a price as close to the price of the most energy-efficient path as possible, i.e. the most energy-efficient speed will yield the lowest price, following the assumption in Section 3.1.3. Finally, the cheapest possible price of charging the required additional energy is returned as the estimated price of the path. Note that this price depends on the subscriptions held by ev. As with the total travel time, it should be noted that this might lead to an overestimate of the price as the distance to be traversed might be an overestimate.

5.3 Partitioned Graph Sizes

The objective of the first experiment is to explore, how the size of the computed overlay graph G_o changes for an increasing average number of vertices in each cell of the computed partitioned graph G_p. The average number of vertices in each cell is referred to as *cell size* of G_p. Furthermore, the experiments are run for the two different EVs, EV_{53} and EV_{90}, described in Section 5.1.5. The preprocessing phase described in Section 4.3.1 is used to compute two graphs; a partitioned graph G_p and an overlay graph G_o, from the original graph G, described in Section 5.1. This phase takes up to 45 minutes for each cell size processed. The results from this experiment will provide an input in the next experiment, when evaluating the running time of the query algorithm for an increasing cell size.

For this experiment, 15 different cell sizes are used, ranging from the lowest sensible number of vertices in a cell, 2^1, to the highest sensible, 2^{15}. There is no relevance to increase the cell size to more than 2^{15}, i.e. 32,768, since the resulting G_o will be increasingly close to the original graph, as the cell size reaches the number of vertices in the original graph, i.e. 94,993. Also, the memory required to handle large cell sizes with a high number of shortcuts, makes it impossible to implement it on the hardware available for these experiments. For a cell size of n in G_p, the resulting number of cells in G_p and G_o can be described as $\lceil |G.V|/n \rceil$.

The results of the experiment can be seen in Table 3 and are visualized in Figure 11. As seen in the table, the number of vertices in the graphs is the same, no matter what EV is used. This is because vertices are chosen regardless of range, but rather from the topology in the graph.

It is notable that the number of vertices in G_o decreases as the cell size increases. This is due to the fact that a fewer number of border edges is included in the partitioned graph, as the cell size increases, resulting in fewer border vertices as well. The number of vertices in the graph will never be lower than the number of charging stations, i.e. 4,058, since all charging stations are always part of G_o, regardless of the cell size and number of cells.

The resulting number of edges in G_o is more complex to analyze, since both border edges and shortcut edges exist, and neither of these sets of edges are constant in size. Furthermore, the number of edges changes with the EV, as a longer range can result in more shortcuts being feasible. This becomes clear in the table, where the number of edges in G_o remains the same for each cell size from 2^1 to 2^{10}. From a cell size of 2^{11} and up, the number of edges vary depending on the EV. This is due to a larger cell size resulting in fewer partitions covering larger areas in the graph, which

	EV_{53}		EV_{90}	
Speed-up / Algorithm	None	A*	None	A*
AGQuery	97	106	209	216
GQuery	26,789	11,133	5,679	1,129

Table 4: The results from the running times experiments with AGQuery and GQuery. All times are in ms.

	EV_{53}		EV_{90}	
Speed-up / Cell size	None	A*	None	A*
2^1	22,377	8,972	5,114	961
2^2	16,941	6,658	3,904	742
2^3	11,242	5,049	2,538	493
2^4	6,625	2,700	1,678	327
2^5	4,831	1,770	1,426	264
2^6	3,545	1,290	1,122	227
2^7	2,504	999	856	206
2^8	1,921	744	633	162
2^9	1,473	535	460	122
2^{10}	1,355	509	491	148
2^{11}	1,639	653	646	193
2^{12}	2,440	995	1,155	398
2^{13}	5,253	2,244	2,743	713
2^{14}	10,824	4,454	5,188	1,205
2^{15}	29,328	12,501	15,500	3,422

Table 5: The results from the running times experiments with PQuery. All times are in ms.

means that a larger range for the EV yield more shortcut edges in the graph. The number of border edges remain the same regardless of the EV. For the lower range of cell sizes, from 2^1 to 2^5, the total number of edges decreases. This is due to a decrease in the number of border edges. From a cell size of 2^5 to 2^{15}, the total number of edges increases. This is caused by an increasing number of shortcut edges, as fewer cells means more vertices are contained in each cell and thereby the number of shortcuts in each cell increases as well. It is notable that from a cell size of 2^{10} and up, the number of edges in the partitioned graph G_o is actually higher than the number of edges in the original graph G, which contains 107,457 edges.

In the range of cell sizes between 2^4 and 2^9, the size of the overlay graphs in terms of the combined number of vertices and edges is the smallest. However, this can not determine whether the efficiency of these overlay graphs is the best in terms of running time. This will be answered in the second experiment.

5.4 Efficiency of Algorithms

The purpose of the second experiment is to evaluate the time-efficiency of the query algorithms proposed in this article. This includes the GQueryAlgorithm described in Section 4.1.4, referred to as *GQuery*, the PartitionQueryAlgorithm described in Section 4.3.2, referred to as *PQuery*, and the AGQueryAlgorithm described in Section 4.4.2, referred to as *AGQuery*. All three algorithms will be queried the same randomly generated 1,000 feasible source-target pairs two times each for EV_{53} and EV_{90}. First with no speed-up techniques applied to the algorithm, and secondly with the A* speed-up technique described in Section 5.2 included as part of the algorithm. The speed-up technique might result in sub-optimal paths. However, none of the queries performed on the algorithm with the speed-up technique applied, returned a cost different from the cost returned by the algorithm without the use of the speed-up technique, which guarantees optimal results. This means that the probability of computing a sub-optimal path using the speed-up technique is low. However, this may depend on the data used to created the road network. For PQuery, the partitioned and overlay graphs described in the previous experiment will be used. The preference function $\alpha = (0.5, 0.5)$ is used for all experiments. Finally, the average running time of the queries will be evaluated for each algorithm and EV.

The first results described are the results from AGQuery and GQuery. This provides 8 results; four results for each algorithm, namely the average running times without any speed-up techniques applied and with the A* speed-up applied, and for both EV_{53} and EV_{90}. The results seen in Table 4 clearly show, how AGQuery outperforms GQuery in terms of running time for EV_{53}. However, it should be remembered that AGQuery does not guarantee an optimal solution as described in Section 4.4.2. Interestingly, A* does not improve performance on AGQuery, while on GQuery, the query answering time is more than halved when using A*. The reason why the speed-up technique does not improve the running time of AGQuery, could be because of the problem of not being able to consider the current SoC at charging station vertices in the preprocessing of the auxiliary graph, as described in Section 4.4.2. This might mean that the time saved by the guiding of the speed-up technique does not make up for the time it takes to compute the heuristic values. For EV_{90}, the query answering time is improved much in GQuery, since less charging needs to be considered by the algorithm, due to the longer range. However, for AGQuery, the query answering time is slower for EV_{90} compared to EV_{53}. This is due to the high edge count in the auxiliary graph, caused by the larger range. This supports the conclusion in [1] stating that the algorithm is less efficient for larger ranges.

Secondly, the results from PQuery is described. This provides 56 results; 14 pairs of results from running the query on the partitioned graphs with and without the speed-up technique A* applied, for both EV_{53} and EV_{90}. The results can be seen in Table 5 and are visualized in Figure 12 for EV_{53} and Figure 13 for EV_{90}. The graphs G_p and G_o computed with a cell size of 2^{10} provides the best results for EV_{53}, with an average running time of 1,355 ms without the speed-up techniques applied, and 509 ms with the A* speed-up technique applied. Interestingly, G_o with a cell size of 2^{10} contains more edges than the original graph G used for GQuery, as described in the first experiment. For EV_{90}, the cell size performing the best results is 2^9, with an average running time of 460 ms without the speed-up techniques applied and 122 ms with the A* speed-up technique applied.

Furthermore, for the best performing cell sizes used in the experiments, PQuery outperforms GQuery in terms of running time. However, PQuery only outperforms AGQuery in terms of running time with the A* speed-up applied to the algorithm. As in the GQuery experiment, using A* improves the efficiency of the algorithm significantly, since the average

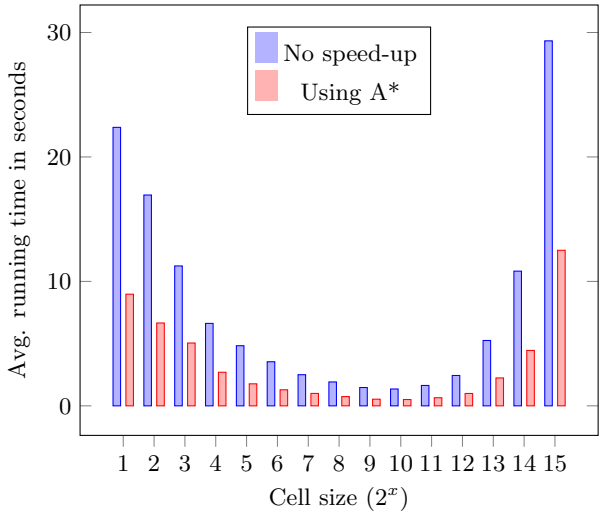

Figure 12: Running times of PQuery for EV_{53}.

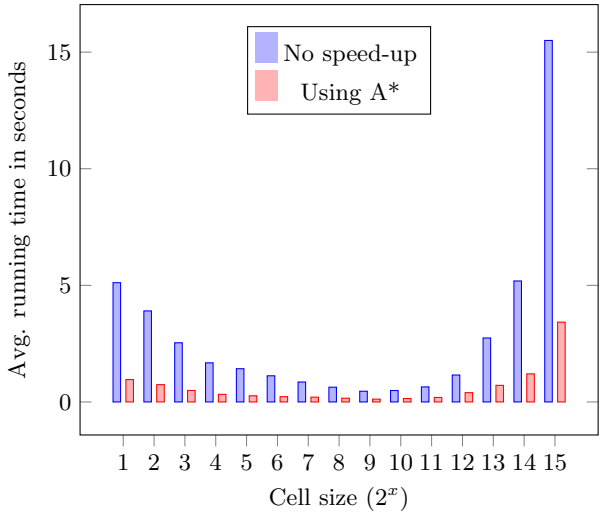

Figure 13: Running times of PQuery for EV_{90}.

running time of the queries utilizing A* is less than half the running time of the queries not utilizing the speed-up techniques, for any of the cell sizes used on EV_{53}. Furthermore, the results from EV_{90} show that the speed-up decreases the running time of queries to around one fourth of the running time without applying the speed-up technique. This could mean that the speed-up technique works even better for longer-ranged EVs.

6. CONCLUSION

In this article, we have studied the problem of finding the optimal bi-criterion path between two points in a graph, according to a user-defined preference function. For the experiments, we have used a graph of the motorways of Germany and two EVs with different battery capacities. Experiments with a basic approach, based on the expansion technique of Dijkstra's algorithm, showed that a path could be found with an average query answering time of less than 12 seconds for the EV with the smallest capacity, and less than 2 seconds for the EV with the larger battery capacity. Using the principles explained in Section 4.3, the graph was partitioned into cells, where each vertex is a member of only one cell. This partitioned graph was used in further precomputation, where the goal was to create an overlay graph of only border and charging station vertices, reducing the size of the graph. Modifying the basic algorithm, to be capable of navigating in the overlay graph, yielded significantly faster query answering times, especially with an A* speed-up technique applied. Query answering times for the algorithm were around half a second for the EV with the smallest capacity and around one tenth of a second for the EV with the larger battery capacity. Finally, experiments with a modified version of the solution from our previous article [1] showed an average response time of approximately one tenth of a second as well for the EV with the lowest range, and around one fifth of a second for the longer ranged EV. This means that the running time of this algorithm increases for longer ranged EVs, which was also described in [1].

All of the mentioned results are made with algorithms, where an A* speed-up technique is applied. At its best, the speed-up technique decreased the query running times to around one fourth of the running time of the same algorithm without the speed-up technique applied. The A* speed-up technique applied to the algorithms does not guarantee the optimal solution. However, the experiments show that the probability of computing a sub-optimal path instead of the optimal path is low in the used data, since it did not occur in the 1,000 randomly generated source-target queries used for the experiments.

It is worth noting that the solution from our previous work is the fastest for the shorter ranged EV. However, this solution requires precomputation that takes hours to complete, whereas the precomputation of the PartitionAlgorithm takes less than one hour in the worst case. Another notable difference between the PartitionQueryAlgorithm and AGQueryAlgorithm, is the number of vertices and edges in the two graphs needed. The experiment in Section 5.3 shows that the number of edges actually decreases for most cell sizes of the partitioned graph, compared to the 5 million edges in the auxiliary graph for the EV with the smallest range and 9 million edges for the EV with the largest range. However, the number of vertices has increased compared to the auxiliary graph. Additionally, the number of vertices and edges in the partitioned graph for the cells, where the source and target vertices are located, should also be added to the total edge and vertex count. Obviously, this increases the vertex count even more, but the edge count for most of the partitions is still less than 5 and 9 million edges. Finally, the AGQueryAlgorithm loses path optimality from the precomputation. This is due to the SoC not being known at the time of precomputation, and the time penalty for charging with a battery above 80 % charge may not be applied.

7. DISCUSSION

This section discusses some of the properties and choices that have been made to solve the problem with the different approaches.

The PartitionAlgorithm described in Section 4.3 is inspired by the CRP and PRP approach introduced in [3] and [4]. However, these approaches are not directly applicable to the electric vehicle routing problem where charging stations are modeled. The reason for this is that there is no guaran-

tee that all of the charging stations are in the overlay graph. A charging station would only be added to the overlay graph if it is a border vertex. If a partition of the original graph does not have a single charging station as a border vertex, it would only be possible to charge in the cells where the source and target vertices are located, given that a charging station vertex is located in the same cell. Therefore, the PartitionAlgorithm ensures that all charging stations are within the overlay graph. This is done by adding shortcut edges to the charging station from each border vertex of the cell where the charging station is located.

Another aspect of CRP and PRP is to have a hierarchy of overlay graphs, i.e. the multi-layered approach. The disadvantage about using a multi-layered approach with the PartitionAlgorithm is that all charging stations have to be in each overlay graph of the different layers. Therefore, we are not sure whether it will provide a speed-up. However, it is possible to implement the multi-layered approach using the same concepts from the PartitionAlgorithm.

The AGAlgorithm described in Section 4.4, is a modified version of the solution from [1]. In [1] the experiments show a high level of interconnectivity of edges between the charging stations in the auxiliary graph. This is because the charging stations are reachable within the battery capacity of the electric vehicle. As a motivation to limit the interconnectivity of edges between charging stations, we introduced the PartitionAlgorithm. The theory was that the number of edges would decrease by limiting the interconnectivity of charging stations to each cell. By limiting the number of edges, the number of vertices increased because of the border vertices. Therefore, we have developed two approaches, i.e. PartitionAlgorithm and AGAlgorithm, which has a trade off between decreasing the number of edges and the number of vertices.

The running time experiments in Section 5.4 show that the AGQueryAlgorithm is at least a factor of 5 faster than the PartitionQueryAlgorithm for the EV with smaller range. However, the PartitionQueryAlgorithm is faster for the EV with a larger range, compared to the AGQueryAlgorithm, but only when the speed-up technique is applied. As stated in Section 4.4.2, the AGQueryAlgorithm does not guarantee the optimal solution. Therefore, the AGQueryAlgorithm has an advantage over the GQueryAlgorithm and PartitionQueryAlgorithm, wrt. running time, because it does not guarantee the optimal solution. Also, the AGQueryAlgorithm has another advantage because it is label setting compared to the GQueryAlgorithm and PartitionQueryAlgorithm, which are label correcting. The reason why the AGQueryAlgorithm is label setting is because the consumed energy is incorporated in the edges of the auxiliary graph, i.e. if a charging station is reachable from another charging station an edge exist between the two of them. Also, the total travel time and price is precomputable because the last visited charging station is known. Therefore, the preference cost is the only dimension that needs to be considered at query time. The GQueryAlgorithm and the PartitionQueryAlgorithm needs to keep track of the energy consumed and the preference cost because the edges do not guarantee feasibility like the edges of the auxiliary graph.

One might wonder, why the PartitionQueryAlgorithm does not incorporate energy consumption like the auxiliary graph. Actually, it is possible to precompute the price, total travel time and incorporate energy consumption for the outgoing shortcut edges from a charging station because the last visited charging station is known. However, this would not guarantee the optimal solution for the same reason that the AGQueryAlgorithm does not guarantee the optimal solution; the SoC at the time of charging is unknown and a possible charging time penalty is not applied. Also, it is not possible to precompute price, total travel time and incorporate energy consumption for the remaining edges, because those depend on the last visited charging station, which is not known before query time.

7.1 Future Work

We have studied the problem of finding the fastest and cheapest path, in terms of total travel time and price, according to a user-specified preference function and proposed different solutions to solve the problem. However, more work can be done to improve the solutions. Ideas of how to improve the solutions is described in this section.

The speed assumption described in Section 3.1.1 is a limitation in the sense that the user has to drive the speed limit of a road. In the real world, most people do not drive with a speed equal to the speed limit. Some people drive faster than the speed limit and some people drive slower. Future work in terms of lifting this assumption could be to take a variable denoting how fast to drive in relation to the speed limit or allow the user to drive at any arbitrary speed.

The charge assumption described in Section 3.1.2 ensures that charging stations and the target vertex is reached with the minimal SoC. However, the cost of the path could be better if optimization of how much to charge at each charging station was performed. An example of this could be a path that makes use of two charging stations, where the first visited charging station provides the cheapest and fastest charging, but the EV can not reach the target from it. In this case, more charging should be done at the first charging station and less charging should be done at the second charging station.

The implementation of a multi-layered approach for the PartitionAlgorithm, as mentioned in Section 7, is a possible improvement for the PartitionAlgorithm. However, each layer needs to include all charging stations for the reason described in Section 7.

The AGQueryAlgorithm does not guarantee an optimal solution as described in Section 4.4.2. However, a solution to make it guarantee an optimal solution would be to precompute the driving time and energy consumption as weights instead of price and total travel time. At query time, the SoC, price and total travel time should be computed as the GQueryAlgorithm and PartitionQueryAlgorithm does when edges are relaxed. It may be a little slower compared to the current solution, in terms of running time, because the functions *cp* and *ct* have to be used to compute the total travel time and price when an edge is relaxed. Also, the SoC needs to be handled, similar to how SoC is managed in the GQueryAlgorithm and PartitionQueryAlgorithm. Nonetheless, the algorithm would guarantee the optimal solution.

The heuristic function used in the A* Search is not admissible because the method to compute the euclidean distance can overestimate the true distance as described in Section 5.2. Therefore, another method of computing the euclidean distance, which always underestimates the distance and is more time efficient than the haversine formula, is another approach to make the heuristic of A* Search admissible.

Acknowledgements

We would like to thank our supervisor Simonas Šaltenis for his help and guidance during this Master's Thesis project in Computer Science at Aalborg University.

We would also like to thank the organizations providing the data used in our experiments. This includes the CGIAR Consortium for Spatial Information for providing height data, OpenStreetMap.org for providing road network data, and OpenChargeMap.org for providing charging station data. Also thanks to Dr. Franz Graf for creating and releasing the osmosis-srtm-plugin tool to import SRTM height data into an OpenStreetMap road network, and George Karypis of University of Minnesota for creating and releasing METIS.

8. REFERENCES

[1] M. Højsleth, A. T. Lau, and A. Strandfelt, "*Skyline Route Planning with Recharging for Electric Vehicles*." Previous work by the same authors of this paper, January 2016.

[2] *Shortest Feasible Paths with Charging Stops for Battery Electric Vehicles*, GIS '15, (New York, NY, USA), ACM, 2015.

[3] T. P. Daniel Delling, Andrew V. Goldberg and R. F. Werneck, "Customizable route planning in road networks," tech. rep., Microsoft Research Silicon Valley & Karlsruhe Institute of Technology, 2014.

[4] S. Funke and S. Storand, "Personalized route planning in road networks," in *Proceedings of the 23rd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '15, (New York, NY, USA), ACM, 2015.

[5] T. Zündorf, "Electric vehicle routing with realistic recharging models," Master's thesis, Karlsruhe Institute of Technology, 2014.

[6] M. Sachenbacher, M. Leucker, A. Artmeier, and J. Haselmayr, "Efficient energy-optimal routing for electric vehicles," 2011.

[7] S. Storandt, "Quick and energy-efficient routes: computing constrained shortest paths for electric vehicles," 2012.

[8] T. Burch, "Poll on battery size." `http://www.teslamotorsclub.com/showthread.php/5690-Poll-on-battery-size/page5`, 2011. Used: 14/12/15.

[9] J. B. Straubel, "Roadster efficiency and range." `https://www.teslamotors.com/blog/roadster-efficiency-and-range`, 2008. Used: 14/12/15.

[10] S. J. USA, "Ev energy consumption." `http://www.solarjourneyusa.com/EVdistanceAnalysis5.php`. Used: 14/12/15.

Summary

This article studies the problem of route planning for electric vehicles (EVs) in a road network. Route planning is often performed with interest in the fastest path. However, some constraints for computing the fastest path for an EV, are different from the constraints involved in route planning for fossil fueled vehicles. For example, recharging of an EV is more time consuming than refueling a fossil fueled vehicle. Furthermore, the pricing for recharging EVs follows different standards than the pricing for fossil fueled vehicles. Charging providers offer various subscriptions, and determining the cheapest option may not be trivial.

Therefore, this article suggests an approach to balance travel time with price for EV route planning. This is done by taking a user-specified preference function for travel time and price as input, while taking charging and recuperation into account. The path, which, according to the preference function, is the most optimal, is returned.

This article is an extension of our previous work, which considered almost the same problem. The difference is that the solution in our previous work returns all Pareto-optimal paths instead of the single optimal path. Even though the problems are slightly different, the solution is reused in this article.

Two different query algorithms are proposed, along with the previous solution, *AGQueryAlgorithm*, to solve the problem of returning the optimal path according to a preference function. The first query algorithm presented, *GQueryAlgorithm*, makes no use of precomputation and serves as a baseline algorithm. The second query algorithm, *PartitionQueryAlgorithm*, uses a precomputation technique to speed up the queries. All three query algorithms uses the principles of Dijkstra's algorithm, which is modified to compute the optimal path given a preference function.

The PartitionQueryAlgorithm is an extension of the baseline algorithm, and uses a partitioned graph, where vertices are partitioned into cells, such that every vertex is contained in exactly one cell. Edges leading from one cell to another cell are referred to as *border edges*, and these are added to an overlay graph along with the source and target vertices of the border edges, referred to as *border vertices*. Furthermore, the overlay graph contains all charging stations, and these are connected to the border vertices of the cell in which the charging station is contained using *shortcut edges*. The AGQueryAlgorithm works in an auxiliary graph, and is a modification of the solution from our previous work, such that it returns the single optimal path according to a preference function. Unfortunately, this solution is not guaranteed to return the true cost of the path because an assumption in the preprocessing is made such that whenever a charging station is visited, the battery in the EV is empty. Otherwise, it is not possible to precompute the price and travel time of the edges because the energy in the battery is different, for each query, when visiting different charging stations. Additionally, we propose two optimization techniques to speed up the queries, namely optimal sub-structure and A* search.

At last, we perform experiments on real world data of Germany to evaluate the proposed algorithms in terms of the precomputed graph sizes and running time of the query algorithms. The experiments are performed on two different EVs, one with a shorter range and one with a longer range.

The experiments show that the GQueryAlgorithm is the slowest algorithm. This was an expected result, since GQueryAlgorithm is a baseline algorithm

for comparison. AGQueryAlgorithm performs the best results for the shorter ranged EV, however, for the longer ranged EV, the time-efficiency of the algorithm decreases. The PartitionQueryAlgorithm shows acceptable results for the shorter ranged EV, and performs the best results for the longer ranged EV. This means that the PartitionQueryAlgorithm has an advantage in a time perspective, since the evolution of EVs is that the ranges are becoming longer and longer. The experiments also show that the A* speed-up technique, at its best, reduces the running time of queries to around one fourth of the running time of the same algorithm without the speed-up technique applied.