# Time-optimised Route Planning for Electric Vehicles

A. B. Eriksen
Institute of Computer Science
Aalborg University
aeriks11@student.aau.dk

M. A. Madsen
Institute of Computer Science
Aalborg University
mman10@student.aau.dk

M. M. Andersen
Institute of Computer Science
Aalborg University
mama11@student.aau.dk

S. B. Jensen
Institute of Computer Science
Aalborg University
sbje11@student.aau.dk

## ABSTRACT

In the future electric vehicles will play an increasingly important role in the traffic and due to the limited recharging possibilities as well as long recharging time of electric vehicles, the requirement for a different route planning system emerges.

We propose a greedy heuristic route planning algorithm that compute time-optimised paths, in a road network, for electric vehicles. Unlike many of today's navigation systems, the greedy heuristic algorithm presented in this paper takes both the drive time and the recharge time into account. We further introduce a linear programming solution, for solving simple paths optimally. For an entire road network, the linear programming solution is not plausible in practice, instead the linear programming solution is used for evaluating the quality of the route plan outputted by the greedy heuristic algorithm.

We extract map data from OpenStreetMap to create a road network for experimentation. The objective of the experiments is to analyse the route plan and the time complexity of the greedy heuristic algorithm through various experiments. This is done to asses the importance of parameters such as the charge rate of charging stations and driving distance.

From the quality assessment we can conclude that the greedy heuristic algorithm's route plan for a path is 0.1% worse than an approximately optimal route plan for the same path.

## Keywords

Fastest Path, Constrained Shortest Path, Electric Vehicles, Road Network, Route Planning, Linear Programming, Greedy Algorithms, Heuristics

## 1. INTRODUCTION

As the adoption rate of Electric Vehicles (EV) increases [1] and because the charging station infrastructure is poor, the need for a route planning system that incorporates charging stations and recharge time emerges. EVs have a short range, generally between 100km and 250km. Though some EV models like Tesla's Model S can reach up to 426km on a single charge [2]. EVs require a significantly longer time to recharge, compared to refuelling a gasoline car. Charging stations can be split into two groups: the less widespread "fast charging" stations takes between 30 minutes and 2 hours to recharge an EV and the more widespread "slow charging" stations takes between 4 and 12 hours. Globally, there exist about 50.000 slow charging stations and 2.000 fast charging stations [3]. As the time spent recharging is a significant factor in terms of the total travel time, a route planning system for EVs should take charging time and driving time into account.

A central aspect is the EVs energy consumption rate. Slow recharging combined with a high consumption rate while driving fast, can in turn result in a combined slower route, due to increased charge time. The energy consumption rate of vehicles increases polynomially with speed due to aerodynamics[1].

It should be clear that no traditional shortest path algorithm is able to accommodate for the relationships between the speed, consumption rate, charge rate and current battery of the electric vehicle.

In Section 2 we describe the mathematical notation and model of this paper and present the definition of the fastest path problem. Furthermore, we give a brief example of why a shorter path does not necessarily imply a faster path for EVs. In Section 3 we present the related work. In Section 4 we define the problem of optimising a simple path (a path without cycles) and present two possible solutions for solving this problem. One solution approximates an optimal solution using Linear Programming (LP) and the other solution uses

[1]Aerodynamics play a big role in the energy consumption for vehicles. The energy losses increase greatly at high speeds. The force of air friction is given by the following equation: $F = \frac{1}{2}\rho V^2 A C_d$, where $\rho$ is the air density, $A$ is the frontal area of the vehicle, $C_d$ is the drag coefficient and $V$ is the speed

greedy choices and heuristics. In Section 5 we present a greedy heuristic algorithm for solving an entire road network. In Section 6 we present the experiments used for evaluation of the greedy heuristic solution. A naive algorithm was implemented for comparison in the experiments. Finally, in Section 7 we present the conclusion of this paper.

# 2. PRELIMINARIES

In this section the mathematical notation and model used in this paper, is introduced. We further present a brief example of why a shorter path does not necessarily equal a faster path for EVs.

## 2.1 Notation

We assume a road network can be modelled as a graph. Edges represent road segments, and vertices represent charging stations or road intersections. Along with the vertices and edges, the model should also include: edge distance, speed limits and charge rates. Thus a road network, $RN$, is defined as a tuple $RN = (V, E, D, v_{max}, v_{min}, R_{ch})$ comprising of a set of vertices, $V$, and a set of edges, $E$. Where $V$ is a finite set and $E$ is a binary relation on $V$. The tuple also includes the following functions:

$$D(e) \to d$$

$$v_{max}(e) \to v$$

$$v_{min}(e) \to v$$

These functions respectively returns the length of edge $e$, the maximum speed of edge $e$ and the minimum speed of edge $e$. The charge rate of vertex $u$, is described by the following function:

$$R_{ch}(u) \to c$$

$c$ is the charging rate of vertex $u$ given in Watt. Vertices with a charge rate of $c = 0$W are considered road intersections, while vertices where $c > 0$W are considered charging stations.

An electric vehicle $EV = (R_{co}, B_{cur}, B_{cap})$ is a tuple defined by three variables: Battery capacity, $B_{cap}$, given in Wh, consumption rate, $R_{co}$, given in Wh/m. The consumption rate is a function on the following form: $R_{co}(v) = av^2 + bv + c$ where $v$ is the speed of the EV and the constants $a$, $b$ and $c$ are dependent on the specific EV. The current battery level of the EV, in a given state, $B_{cur}$, is dependent on three variables: The starting battery, energy consumed and energy charged.

We define a path $P$ of length n in a graph as a sequence of vertices $\langle u_1, u_2, \ldots, u_n \rangle$ and a set of edges $(u_i, u_{i+1}) \in E$ for $i = 1, 2, \ldots, n-1$.

The fastest path problem can now be defined using the above notation:

**Definition 1:** Given a road network, $RN = (V, E, D, v_{max}, v_{min}, R_{ch})$ and an EV, $EV = (R_{co}, B_{cur}, B_{cap})$. The fastest path from $s$ to $t$, is defined as the path $P = \langle u_1, u_2, \ldots, u_n \rangle$ where $u_1 = s$, $u_n = t$ and the time spent driving and charging is minimal. The output should include a path and a complete schedule of:

- Where and how long to charge
- The speed to drive on each road segment

## 2.2 Shorter does not imply faster

A shorter path does not necessarily imply a faster path for electric vehicles. This is partly due to the fact that electric vehicles use polynomially more energy as their speed increases, but also due to the fact that charge times on charging stations varies a lot. Driving a longer path with a fast charging station, can therefore turn out to be a faster choice than driving a shorter path with a slow charging station. This is illustrated in Figure 1. In the example, the EV is assumed to have a battery capacity and a starting battery of 100kWh and a consumption rate of 0.4kW h/km at the speed of 100km/h.

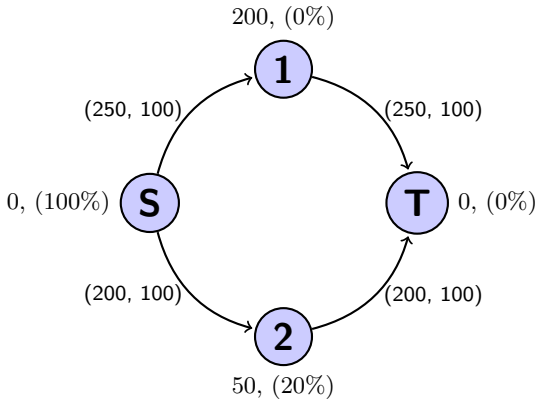


Figure 1: A simple road network with starting point s and end point t. The charge rates on the road network dictates that the longer path is in fact the fastest

Path 1 consists of two edges with distance: 250km and speed limit: 100km/h and a charging station with a charge speed of 200kW. Path 2 consists of two edges with distance: 200km and speed limit: 100km/h and a charging station with a charge speed of 30kW. The current battery capacity of the vehicle is noted at each vertex in parenthesis. The total time of each path:

**Path 1:** $\frac{250\text{km}+250\text{km}}{100\text{km/h}} + \frac{100\text{kWh}}{200\text{kW}} = 5.5\text{h}$

**Path 2:** $\frac{200\text{km}+200\text{km}}{100\text{km/h}} + \frac{60\text{kWh}}{30\text{kW}} = 6\text{h}$

# 3. RELATED WORK

To the best of our knowledge, there is no directly related research to the fastest path problem for EVs, which is the focus of this paper. There is however a lot of work on the shortest path problem. Probably best known amongst all of the shortest path algorithms is Dijkstra's algorithm [4], which we will use later in Section 5.

The fastest path problem is an instance of the constrained shortest path problem (CSPP) [5], because one does not only have to find the shortest path, but also adhere to certain constraints, in this case the battery. Our problem of time-optimised route planning can be framed as a CSPP,

though CSPP is known to be NP-complete [6]. Another interesting topic, in the domain of electric vehicles, is working with energy-optimal route planning instead of time-optimised route planning. [7] uses weights in a graph to represent energy gain and loss. The objective then becomes to find the path which has the lowest energy loss. This is possible because of the energy recuperation property of EVs.

Lastly, if one is interested in the practical usage of shortest path algorithms, for real-world road networks, many navigation systems introduce hierarchical substructures to perform the shortest path computation faster. This is known as contraction hierarchies. It is a concept which could also be applied to the approach introduced in this paper, in order to reduce the running time. It is however not the main concern of the research in this paper.

# 4. FASTEST PATH

As shown in Section 2.2, finding the fastest path for an EV does not only depend on the lengths and speed limits of the road network's edges, but also on the charge rates of its vertices. In this section we will focus on solving an arbitrary simple path. We first describe the path optimisation problem in Section 4.1. Then we use this to arrive at an approximately optimal solution for a path in Section 4.2 and finally we propose a different way to solve a simple path using a greedy heuristic choice in Section 4.3. Later, we will concern ourselves with solving an entire road network in Section 5

## 4.1 Optimisation Problem

This section introduces a formal description of how to solve a simple path of length $n$ optimally.

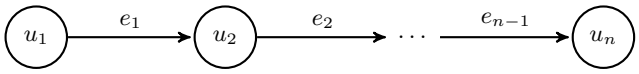


Figure 2: Example of a path consisting of $n$ edges

The objective of the optimisation problem is to minimise the time spent driving and the time spent charging on a path. The total time spent on a path can be expressed as the following objective function:

$$\text{min:} \quad \sum_{i=1}^{n-1} \left( \frac{D(e_i)}{v_{e_i}} + CT_{u_i} \right)$$

Where $D(e_i)$ is the length of road segment $i$, $v_{e_i}$ is the speed driven on road segment $i$ and $CT_{u_i}$ is the charge time at the $i^{\text{th}}$ vertex of the path illustrated in Figure 2. It should be clear that the objective function of this problem is concerned with time because $D(e_i)/v_{e_i}$ is the time spent driving and $CT_{u_i}$ is the charge time. The objective function has two unknown variables: $v_{e_i}$ and $CT_{u_i}$. The unknown variables of the objective function are constrained by the physical properties of the *EV* driving the path. We will now ensure that the physical constraints defines the feasible region of the optimisation problem.

On every edge the speed of the EV must be within the speed limits. $v_{e_i}$ is the speed on edge $e_i$. $v_{min}(e_i)$ and $v_{max}(e_i)$ is the minimum and maximum speed of edge $e_i$. Thus, the first constraint is formulated as:

$$\forall_{i \in 1 \ldots n-1} : \; v_{min}(e_i) \leq v_{e_i} \leq v_{max}(e_i)$$

When charging at the $i^{\text{th}}$ vertex, the energy acquired is given by $EA(u_i)$. The energy acquired is calculated as charge rate, $R_{CH}(u_i)$, multiplied by charge time, $CT_{u_i}$:

$$\forall_{i \in 1 \ldots n} : \; EA(u_i) = R_{CH}(u_i) \times CT_{u_i}$$

The energy spent passing edge $e_i$ is given by $ES(e_i)$ and is calculated by multiplying the distance $D(e_i)$ by the consumption rate of the vehicle $R_{CO}(v_{e_i})$) at speed $v_{e_i}$.

$$\forall_{i \in 1 \ldots n-1} : \; ES(e_i) = D(e_i) \times R_{CO}(v_{e_i})$$

Furthermore, the battery level of the EV must be between 0 and $B_{cap}$. This constraint is split into two constraints. The first constraint ensures that the EV can not pass an edge without having the required energy. The second constraint ensures that the EV can not overcharge at any charging station.

For the first constraint: The sum of energy acquired at the charging stations prior to the current edge, must be larger or equal to the sum of energy spent on the edges prior to the current edge and the energy needed to pass the current edge. As shown in Figure 2, vertex $i$ is prior to edge $i$. $B_{cur} + \sum_{j=1}^{i} EA(u_j)$ is the amount of energy acquired on the path at the $i^{\text{th}}$ vertex, where $B_{cur}$ in this context is the starting battery level of the EV. $\sum_{j=1}^{i} EA(u_j)$ is the combined energy spent on the path at the $i^{\text{th}}$ edge. Combining these we get the following constraint:

$$\forall_{i \in 1 \ldots n-1} : \; 0 \leq B_{cur} + \sum_{j=1}^{i} EA(u_j) - \sum_{j=1}^{i} ES(e_j) \leq B_{max}$$

The second constraint is almost exactly like the previous constraint, however it looks one charging station further to ensure the energy does not exceed $B_{cap}$ at any charging station:

$$\forall_{i \in 1 \ldots n-1} : \; 0 \leq B_{cur} + \sum_{j=1}^{i+1} EA(u_j) - \sum_{j=1}^{i} ES(e_j) \leq B_{max}$$

Finally, it should not be possible for the EV to spend a negative amount of time at a charging station:

$$\forall_{i \in 1 \ldots n} : \; 0 \leq CT_{u_i}$$

The optimisation problem is a NP-complete problem, as it is a non-linear optimisation problem [8]. The reason it is non-linear is because the consumption rate of an EV is a quadratic function and the length divided by speed also is a non-linear function.

## 4.2 Approximating an Optimal Path

In this section an approach to finding an approximately optimal solution to the optimisation problem proposed in the previous section is formulated. As mentioned, the optimisation problem is a non-linear optimisation problem, due to the distance divided by speed, $D(e_i)/v_{e_i}$, and the consumption rate of EVs, $R_{CO}(v_{e_i})$, are non-linear functions. However, these functions can be used in a linear programming problem by formulating $D(e_i)/v_{e_i}$ and $R_{CO}(v_{e_i})$ as piece-wise linear functions [9]. Specifically, the problem is modelled as a mixed

integer problem (MIP) problem. This section introduces the extra constraints necessary to solve the problem as a MIP problem and thereby end up at an approximately optimal solution [10]. The quality of the solution depends on how many pieces the non-linear function is split into.

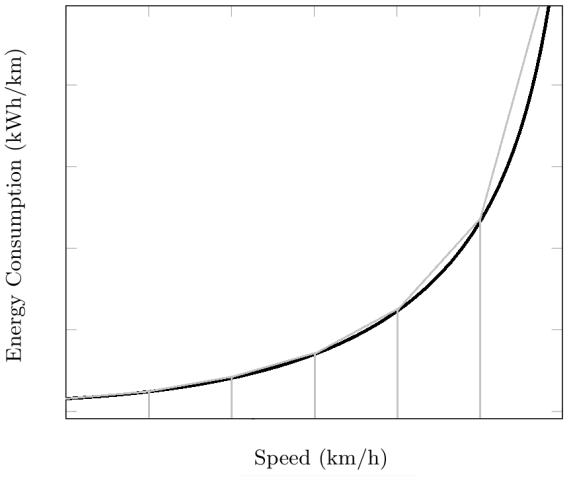


Figure 3: Example of a non-linear function which is split into six line segments

The linearization of a function is handled as shown in Figure 3. The non-linear function is split into a number of linear line segments. To handle the linearization three new sets are introduced: a set of known variables which is the pre-computed function of each linear line segment, and another set of known variables which is the start and end points of each line segment and a set of unknown variables which will be the line segment chosen. The unknown variable is called selected line, SL, which has a two dimensional size $n \times m$ where $n$ is the amount of edges with a unique function. $m$ is the number of linear lines every edge's function is split into. This leads to the following constraints:

For n edges, exactly n line segments needs to be selected. In other words, for every edge only one linear line is chosen.

$$\forall_{i \in 1...n} : \sum_{j=1}^{m} SL_{i,j} = 1$$

$\sum_{j=1}^{m} SL_{i,j} = 1$ ensures that only one line segments can be selected, because exactly one line segment needs to be chosen. In other words, $SL$ needs to be binary:

$$\forall_{i \in 1...n, j \in 1...m} : \quad SL_{i,j} \in 0, 1$$

Because we now have $m$ points describing each line, we need $m$ values for each $v_{e_i}$. Thus $v_{e_i}$ is extended with an additional index parameter to handle this additional dimension. The speed of the EV needs to be constrained by the start and ending point of the selected line segment:

$$\forall_{i \in 1...n, j \in 1...m-1} : \; SL_{i,j} \times P_{i,j} \leq v_{e_i,j} \leq SL_{i,j} \times P_{i,j+1}$$

$P$ is the set of starting points for all line segments. $P_{i,j}$ is the starting point of the $j^{\text{th}}$ line segment for the $i^{\text{th}}$ edge, $P_{i,j+1}$ is the ending point of the $j^{\text{th}}$ line segment and it is also the starting point of the $(j+1)^{\text{th}}$ line segment. $SL_{i,j} \times P_{i,j}$ is the minimal speed the EV is allowed to drive on edge $i$ and $SL_{i,j} \times P_{i,j+1}$ is the maximal speed. Note that the values will either be a valid value or 0, if the segment is not selected.

To linearize $R_{CO}(v_{e_i})$ we precompute a set of line functions on the form $ax + b$. We store the slope $a$ as $Slopes$, and the constants $b$ as $Constants$. In total $n \times m$ function are defined. The linearized model of $R_{CO}(v_{e_i})$ for every edge $i$ will look like this:

$$\sum_{j=1}^{m} Slopes_{i,j} \times v_{e_i,j} + \sum_{j=1}^{m} SL_{i,j} \times Constants_{i,j}$$

$\sum_{j=1}^{m} Slopes_{i,j} \times v_{e_i,j}$ is equivalent to $ax$ and $\sum_{j=1}^{m} SL_{i,j} \times Constants_{i,j}$ is equivalent to $b$. Finally note that this is a way to determine the solution of $R_{CO}(v_{e_i})$ and before a solution to a path can be found, the line segments and points for *both* of the non-linear functions needs to be precomputed.

## 4.3 Greedy Heuristic Path

As mentioned in Section 4.1 finding an optimal solution to a path is an NP-complete problem. In this section a greedy heuristic approach is proposed which solves the problem one edge at a time. The problem then becomes finding the optimal speed for the current edge, in the interval of $v_{min}$ and $v_{max}$.

Given a path there are two possible cases when choosing the optimal speed for passing an edge:

1. Case 1: Driving is fastest
2. Case 2: Charging and driving is fastest

The intuition is then to find which speed we want to drive and whether or not we should charge before driving. However, it might not be possible to drive at all, given the limited battery of the EV.

For case 1: The optimal speed when passing the edge $e_i = (u_i, u_{i+1})$ without charging, can be found by solving this equation for $v$:

$$B_{cur} - D(e_i) \times R_{CO}(v) = 0$$

We call this speed $v_{opt1}$. If $v_{opt1}$ is lower than $v_{min}(e_i)$, there is not enough energy in the battery to drive from $u_i$ to $u_{i+1}$ and thus the time is set to $\infty$, indicating that just driving is not an option. If it is possible to drive the edge using only energy from the battery, the time it will take to pass the edge can be calculated as $D(e_i)/v_{opt1}$

For case 2: The equation for solving an edge, $e_i$, when driving and charging is given by the following equation:

$$T(v) = \frac{D(e_i)}{v} + \frac{R_{CO}(v) \times D(e_i) - B_{cur}}{charge_{rate}(u_i)}$$

This equation is on the form: $av^2 + bv + c$, due to the fact that $R_{CO}(v)$ is a quadratic function. $a$, $b$ and $c$ are some constants which are given by the instance of the vehicle. Represented in a Cartesian coordinate system, $T(v)$ is a parabola, as can be seen in Figure 4, note that the graph is only defined for positive speeds larger than 0 and can only be calculated if $charge_{rate}(u_i)$ is larger than 0. On the x-axis is the speed of

the vehicle and on the y-axis is the time spent. The turning point of the graph represents the optimal speed, $v_{opt2}$, to pass $e_i$. The turning point is easily calculated by finding a tangent line with a slope of 0.

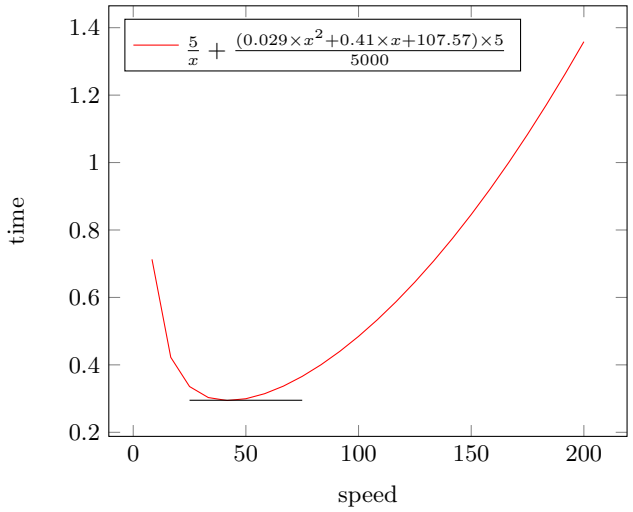


Figure 4: In this instance of $T(v)$, going from $u_i$ to $u_{i+1}$, we have a distance of 5km and a charge speed of 5kW on $u_i$. The optimal speed in this case is 42.12km/h, and the total time is roughly 17 minutes to pass through the edge when driving and charging

If $v_{opt2}$ is smaller than $v_{min}(e_i)$, then $v_{min}(e_i)$ defines the optimal speed for the edge. Similarly if $v_{opt2}$ is larger than $v_{max}(e_i)$, $v_{max}(e_i)$ defines the optimal speed for the edge. If $v_{opt2} = v_{min}(e_i)$ there might be a possibility that driving and charging is not an option. The time when passing edge $e_i$ in case 2, can then be found with $T(v_{opt2})$.

Having the two ways of deciding the time it takes to pass an edge, we can formulate an algorithm which decides how to drive each edge:

The *travel_time* function takes as input a road network $RN$, the two vertices which makes up the edge to be driven and an EV. The procedure utilises a few helping functions. The two solve functions *solve*1 and *solve*2 are function solvers specified for case 1 and case 2. The solvers returns the optimal speed for the EV according to the edges driven.

The function $getCS(u_1)$ returns a list of charging stations. If $u_1$ is a charging stations with a higher charge rate than any charging station previous to $u_1$ in the path, then $u_1$ is returned. Otherwise, the list of charging stations prior to $u_1$ is returned, with the best charging station in the beginning of the list. The function also maintain the potential energy remaining on each charging station. This means that every time the EV spend energy driving, this energy is removed from the potential energy on all charging station. When a charging station is reached, the potential energy is set, based on the difference between the battery capacity and the current battery: $EV.B_{cap} - EV.B_{cur}$. For instance, if the EV reaches a charging station with 50% battery, the remaining 50% battery capacity is the potential energy set for that charging station.

**Algorithm 1** The travel time algorithm

```
1: function TRAVEL_TIME(RN, u1, u2, EV)
2:     f_drive = EV.B_cur − RN.D(e) × EV.R_CO(v)
3:     f_charge−drive = RN.D(e)/v + (RN.D(e) ×
           EV.R_CO(v) − energy)/charge_rate)
4:     e = (u1, u2)
5:     v_opt1 = solve1(v, f_drive)
6:     if EV.B_cur − RN.D(e) × EV.R_CO(v_opt1) < 0 then
7:         time_1 = ∞
8:     else
9:         time_1 = RN.D(e)/v_opt1
10:        energy_needed_1 = RN.D(e) × EV.R_CO(v_opt1)
11:    CS = getCS(u1)
12:    energy_needed_2 = ∞
13:    energy = EV.B_cur
14:    time_added = 0
15:    time_2 = ∞
16:    while energy_needed_2 > energy & len(CS) ≠ 0 do
17:        best_CS = extractmax(CS)
18:        B_possible = best_CS.B_possible
19:        charge_rate = RN.R_CH(best_CS)
20:        v_opt2 = solve2(v, f_charge−drive)
21:        energy_needed_2 = RN.D(e) × EV.R_CO(v_opt2)
22:        energy = energy + B_possible
23:        if energy − energy_needed_2) < 0 then
24:            time_added = time_added +
                   (B_possible/charge_rate)
25:            CS.remove(0)
26:            updateCS(CS)
27:    if (energy − energy_needed_2) < 0 then
28:        time_2 = (RN.D(e)/v_opt2) +
               (energy_needed_2/charge_rate) + time_added
29:    if time_1 = ∞ & time_2 = ∞ then
30:        return ∞, NIL, ∞, ∞
31:    if time_1 ≤ time_2 then
32:        return (time_1, CS,
               EV.B_cur − energy_needed_1, energy_needed_1)
33:    else
34:        return (time_2, CS,
               energy − energy_needed_2, energy_needed_2)
```

The last function $updateCS(CS)$ finds a new best charging station and deletes all charging stations prior to it, this is needed because previously best charging station has been removed by $CS.remove(0)$. Lines 5-10 handles case 1: Driving using the battery already in the battery. Lines 11-26 handles case 2: Charging and driving. The while loop ranging from line 16 till 26 makes sure the EV always charge as much as possible, at the best previous charging station, but without allowing overcharging. The if statement on line 6 and 27 checks if a valid solution to the case is found, if so, the time for the case is calculated. Lines 29-34 checks if a solution is found and if so, which solution is better. Having a way to solve a single edge, the time of the path can be computed by looping over all edges.

## 5. GREEDY HEURISTIC SOLUTION

In Section 4.2 and 4.3 two different approached for solving a path are introduced. We are now ready to describe an algorithm for solving the fastest path problem for an entire road network.

One approach for solving the fastest path problem, for a whole road network, would be to iterate over every single simple path in the graph. The problem would then be to figure out the best way to travel every path according to either the *travel_time* procedure or the Linear Programming solution. However doing this would be too complex, as the number of simple paths in a graph can be as much as $O(n!)$, which in itself is too expensive for practical usage.

Instead one can utilise existing graph algorithms for reducing the number of paths to look at. For instance in Dijkstra's algorithm, the number of paths is substantially reduced by relaxing pairs of nodes. This is possible because the shortest path problem has an optimal substructure. Unfortunately the fastest path problem does not have such a substructure, however applying a heuristic to Dijkstra's algorithm, will allow us to build useful paths, see Algorithm 2.

Instead of simply relaxing a static weight of edges, we relax accordingly the time being calculated by the *travel_time* procedure. Note that the Linear Programming solution could theoretically be applied here instead of *travel_time*. To guarantee that charging stations are used, we only relax an edge from vertex $u_1$ to $u_2$ if both a lower travel time is found and the charging possibilities are not lowered. The charging possibilities are not considered lowered, if vertex $u_2$ is a charging station, if $u_2$ does not have any charging stations on it predecessor path or if vertex $u_1$s predecessor path has a charging station which still has potential energy.

Alternative greedy choices can be utilised, for instance the following:

- Time
- Time and charging stations with higher potential energy
- Time and a ratio for charging rate and potential energy of charging stations

Relaxing according to time might give faster paths, but will also be less robust as it will not prioritize paths with charging station. This could result in paths which are impossible for an EV to drive. Relaxing according to time and charging stations with higher potential energy, might lead to both faster and slower paths. However, the algorithm will likely be more robust, as a higher potential energy is propagated, allowing the EV to travel farther. Relaxing according to time and a ratio for charge rate and potential energy, will result in a trade-off between route time and robustness of the algorithm.

The input of the algorithm is a road network $RN$, the stating vertex is $s$ and target vertex is $t$. $EV$ is an instance of the EV driving. Lines $2-6$ initializes the values of each vertex. Lines $7-8$ sets proper values on the starting vertex ($s$), naturally the time spend driving from $s$ to $s$ is 0. The current battery level on $s$ is set to the current battery level of the EV. Lines $9-10$ sets up a priority queue and insert $s$. Lines $11-21$ while the priority queue is not empty, the vertex $u_1$ with the lowest time is removed from the queue, and the edges from the vertex is relaxed. Lines $13-21$ goes through every adjacent vertices to $u_1$, computes the time

**Algorithm 2** The greedy heuristic algorithm

```
1: function GreedyHeuristic(RN, s, t, EV)
2:     for all u ∈ RN.V do
3:         u.time = ∞
4:         u.predecessor = NIL
5:         u.preCS = NIL
6:         u.B_cur = 0
7:     s.time = 0
8:     s.B_cur = EV.B_cur
9:     Q = PriorityQueue
10:    insert(Q, (s.time, s))
11:    while Q ≠ ∅ do
12:        u_1 = extractmin(Q)
13:        for all u_2 ∈ RN.adj(u_1) do
14:            time, preCS, B_cur, energy =
15:            travel_time(RN, u_1, u_2, EV)
16:            If u_2.time > u_1.time + time &
                   greedy_choice(u_1, u_2) then
17:                u_2.time = u_1.time + time
18:                u_2.predecessor = u_1
19:                u_2.B_cur = B_cur
20:                u_2.preCS = update_CS(preCS, energy)
21:                insert(Q, (u_2.time, u_2))
22:    return t.time, path(t)
```

with the *travel_time* function. Note that *travel_time* uses the current battery of the vertex $u_1.B_{cur}$ instead of $EV.B_{cur}$. If the time is lower and *greedy_choice*$(u_1, u_2)$ returns true, the adjacent vertex $u_2$ is relaxed and all the data from *travel_time*; the time used, the charge stations prior to $v$ ($preCS$), the current battery level $B_{cur}$ and the energy used $energy$ is assigned to $u_2$. The if statement on line 16 decided whether or not the path to the vertex adjacent to $u$ should be relaxed. The function *greedy_choice*$(u_1, u_2)$ is the heuristic that decides when the modified Dijkstra's algorithm should relax, besides simply using time. This implementation of *greedy_choice*$(u_1, u_2)$ return True if $u_1$ has a charging station on it predecessor path with some potential energy, if $u_2$ is a charging station or if both $u_1$ and $u_2$ have no charging stations in their predecessor paths with remaining potential energy. False is returned if only $u_2$ has charging station on its predecessor path with a charging station with potential energy. The function *update_CS*$(preCS, energy)$ on line 20 is a procedure that maintain the list of charging stations with potential energy for the relaxed vertex. The procedure reduces the potential energy on all charging stations on the predecessor path of $u_2$, with the amount of energy spent driving to $u_2$. Line 21 inserts $u_2$ into the priority queue with its relaxed time. Line 22 returns the time on $t$ and the path leading to $t$. This is done by following the line of predecessors until we reach $s$.

The worst case running time of the greedy heuristic algorithm is bound by three procedures. First there is the main procedure that is derived directly from Dijkstra's, which has a worst case complexity of $O(|E| + |V| \log |V|)$, with the Fibonacci heap. To that we need to multiply $O(|V|)$ from the *travel_time* procedure, as this procedure might go through all vertices, if the vehicle is set to charge at a new charging station for every found edge and every vertex in the graph is also a charging station. Finally we have the

procedure $update\_CS(preCS, energy)$ that maintains a set of charging stations relevant on each vertex. In the special case where the vehicle have an infinite battery capacity, but start without any battery, and every vertex found happens to be a charging station with a charge rate in a decreasing order, the procedure would have to multiply $O(|V|)$ in the worst case, as no charging station would be discarded. However only one of the last two procedure's $O(|V|)$ complexity is added, as the case where the algorithm chose to charge at a new charging station for every edge, never happens with an infinite battery capacity. Likewise the case where the procedure maintains a list of every charging station, never happens if the car has a limited battery capacity. Resulting in a worst case running time of $O(|V||E| + |V^2| \log |V|)$.

In practice Dijkstra's algorithm's running time is lower on a sparse graph, which road networks are, and both of the remaining procedures are simultaneously limited by the fact that only a relatively small amount of charging stations exist and that the EV battery capacity.

# 6. EXPERIMENTS

This section presents the experiments for the greedy heuristic algorithm. We chose not to use the LP solution along with Dijkstras algorithm, because it was too slow. Our linear programming solution was instead implemented, such that we can test the quality of the greedy choices. We compare the route times and running time of the greedy heuristic algorithm to a naive algorithm which is presented in section 6.2. The experiments have been conducted in order to illustrate the effects of the different variables in the fastest path problem. The variables include: route distance to drive, charge rate on charging stations, density of charging stations in the road network and the input size.

The default settings for all the experiments are as follows:

- Route distance: 300km
- Charge rates: randomly generated between $10 - 100$kW
- EV consumption rate: $0.019v^2 - 0.77v + 184.4$ Wh/km, where $v$ is the speed (corresponds approximately to Tesla's Model S [2])
- EV battery capacity: 50 kWh
- EV initial battery: 50 kWh
- Density of charging stations: minimum distance of 20km between each charging station

The default setting of the route distance, consumption rate and battery capacity are assigned the above mentioned values to force the EV into charging in most experimentation cases. The charging station density is set to 20km to gain a reliable infrastructure, with an even spread of charging stations. In the default settings we have 326 stations, when in fact Denmark have more than 4000 [3], where most of these charging stations are located in cities.

We have used the drivable part of Denmark as a baseline for the experiments. The dataset features:

- 483407 vertices
- 543482 edges

## 6.1 Road network dataset

To facilitate the experiments, we've had to find a real world road network dataset which contains road distances, speed limits and charging stations of varying charge rates. Such a dataset does not openly exist to our knowledge. Instead we have used OpenStreetMaps (OSM) which is an open-source collection of map data [2]. OSM has a concept of ways and nodes. Ways represent geographical planar objects e.g. roads, cycleways, foot ways etc. A node is a geographical point consisting of a latitude and longitude coordinate. A way is constituted of a set of nodes and some tags which describe meta-information about the way, such as the name of the way and what type of way it is e.g. a road, cycleway, foot way etc.

The ways and nodes can easily be converted into a, for us, useful road network structure for the experiments. This is done, by simply filtering away all types of ways accept roads and use these as edges in our road network. To get a notion of speed limits on edges, we derive general speed limits from the type of the roads. OSM carries such information as whether the road is a motorway, residential way, tertiary way etc. The speed limits are set according to Danish speed limits. Further, we introduce a minimum speed allowed on an edge as 70 % of the speed limit. As for the nodes, we are only interested in creating vertices for the nodes which corresponds to intersections between two roads, all other nodes are ignored. To get a notion of charge rates on the vertices, we have implemented a method which distributes random charge rates on randomly selected vertices in our road network.

## 6.2 Naive algorithm

We have formulated a naive algorithm for comparison with our greedy heuristic algorithm. The naive algorithm should simulate the way a naive electric vehicle driver would choose to travel through a road network. The naive algorithm works the following way: It greedily follows the fastest path in terms of distance divided by speed. Whenever the battery reaches 40 % battery capacity, it starts searching for nearby charging stations in the radius allowed by the 40 % battery capacity given the vehicle's consumption rate. If no charging stations are available with 40 % battery capacity, the path from $s$ to $t$ is not solvable for the naive algorithm. If, on the other hand, one or more charging stations are reachable, the algorithm will choose the closest one and charge to 100 % battery capacity or to the battery capacity needed in order to reach $t$. After charging, the algorithm will resume greedily choosing the fastest roads from the charging station to $t$, in terms of distance divided by speed.

## 6.3 Experiment: Driving Distance

In this experiment we wish to investigate how well the greedy heuristic algorithm solves the fastest path problem, with distance as the independent variable. We have set the charge rates to a constant value of 60kW, such that the charging

[2]One can read more about OSM at `http://www.openstreetmap.org/about`

stations selected becomes irrelevant. The initial route distance is set to 50km and thereafter increases by 50km until it reaches 550km. Each distance is iterated 3 times to acquire an average.

We expect that: as the distance the EV has to travel increases, the difference between the route time the greedy algorithm achieves and the route time the naive achieves, will increase in favour of the greedy algorithm. We expect this because selecting road segments which are optimised according to the charging stations in the path, as the greedy heuristic algorithm does, should result in better route times.

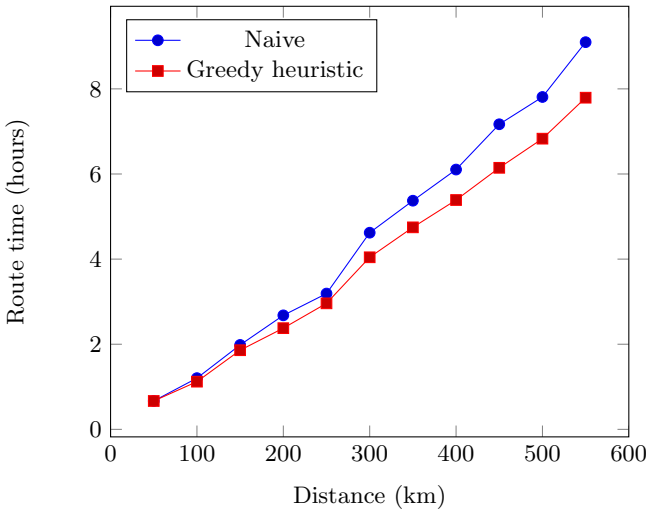


Figure 5: Time spent driving various distances with fixed charging rate

The results of the experiment is illustrated in figure 5. At the very first mark the greedy heuristic algorithm and the naive algorithm achieve the same route time. This might be because no charging is needed and the optimal choice will be to drive the shortest path in terms of distance divided by speed. The results of the experiment shows that: as the distance increases, it becomes more and more relevant to drive at a speed optimised relative to the charging station. This observation matches our expectations.

## 6.4 Experiment: Charge Rates

In this experiment we wish to investigate how well the greedy heuristic algorithm solves the fastest path problem, based on charge rates of the charging stations in the road network. Thus, the charge rate of charging stations is the independent variable. As explained previously, the default charge rates are generated in a range of $10 - 100$kWh. In this experiment we multiply all the charge rates with a scaling factor which starts at $-40\%$, increments by 5% and stops at 75%. This is done 5 times to acquire an average.

In Figure 6, it is quite clear that the behaviour of the naive algorithm scales polynomially with the increasingly better charge rate on the charging stations. However, the greedy heuristic algorithm seems to decrease linearly with better charge rates. It is also clear that the charge time is the bottleneck at low charge rates, while the driving time is the bottleneck at higher charge rates. At mark $-20\%$ there

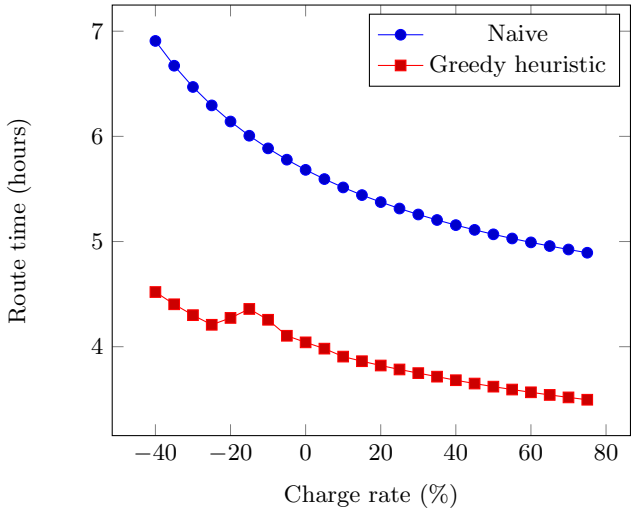


Figure 6: Time spend driving a 300 km path with different charge rates

is a sudden increase in route time for the greedy heuristic algorithm, due to one of the experiments increasing the average substantially. It is hard to explain this situation, but it could be due to the greedy heuristic algorithm making some bad choices. This is a good example of the fact that a chain of local optimal choices, does not yield an optimal solution.

## 6.5 Experiment: Density of Charging Stations

In this experiment, we wish to find the density of charging stations, for which the greedy heuristic algorithm works best. The independent variable for this experiment is the density of charging stations in the road network. Initially, the minimum distance is set to 5 km which generates 3029 charging stations in road network. Table 1 shows the number of charging stations according to the minimum distance between charging stations in the road network.

| Radius (km): | 5 | 10 | 20 | 30 | 40 | 50 |
|---|---|---|---|---|---|---|
| Stations: | 3029 | 827 | 326 | 117 | 76 | 49 |

Table 1: number of charging stations corresponding to the minimum distance between two charging stations

Five experiments were conducted for the greedy heuristic algorithm. Samples were taken in increments of 3km. In Figure 7 the results can be seen, which is an average of all five experiments. Fails were not counted in the average. In total 34 out of 80 experiments failed for the greedy heuristic algorithm and 0 for the naive. Interestingly, as we remove more and more charging stations, the greedy heuristic algorithm seems to stay relatively constant. Whereas the naive algorithm is significantly affected by decreasing the amount of charging stations. From the rest of the points, we can conclude that the greedy heuristic algorithm works best with a density of charging stations below 40km

## 6.6 Experiment: Running Time

We want to compare the running time of the greedy heuristic algorithm and the naive algorithm with Dijkstra's algorithm.

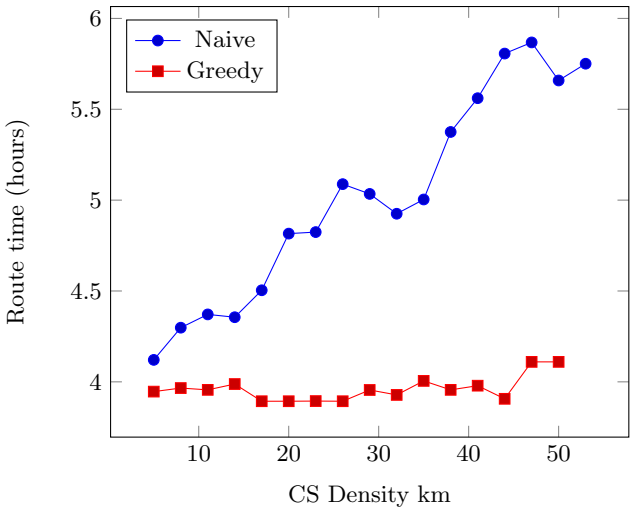


Figure 7: Time spent driving a 300km route with various densities of charging stations

Instead of the default amount of vertices, we start at 400000 vertices and scale downwards to 100000, in steps of 20000. Below 100000 it is no longer possible to find a 300km path in our road network.

Technical details of the experiment environment:
All the algorithms were implemented in Python 2.7. The road network is kept in memory and the computation were conducted using a single thread on an Intel i7-2600 CPU with 2 x 1333 MHz DDR3 memory.

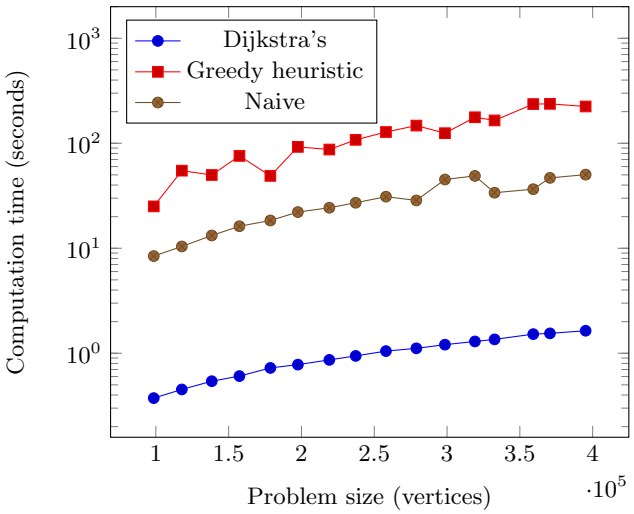


Figure 8: Time spend computing a 300km route on different input sizes. Logarithmic y-scale is used

The results of this experiment is as we expected. Dijkstra's algorithm uses 370 milliseconds solving the smallest input of 100000 vertices, where the greedy heuristic algorithm spends 25 seconds. This shows us that a significant overhead exists from calculating edge weights and managing a larger amount of data for each vertex. From the analysis of the running time, we also see an expected increase in computation time relative to Dijkstra's. While we increase the input size 4 times, Dijkstra's computation time increased about 4.4 times, where the greedy heuristic algorithm increased about 9 times.

## 6.7 Quality Assessment

We will shortly investigate how well the greedy heuristic algorithm solves a path. We are only able to test how well a path is passed in terms of speed and charging. We can not investigate how good the actual path chosen is, since solving the entire road network using the linear programming solution, is simply too time demanding.

To test the quality of the path solution, we implemented the LP solution described in Section 4.1 using an existing LP solver[3]. In our implementation we used 10 lines segments for linearization of the function.

For this quality assessment, 8 experiments were conducted using the same default settings as previously. For each $s$ and $t$ we find a path and the route time using the greedy heuristic algorithm and we find a path and the route time using the naive algorithm, for the same source and destination! We then input both these paths to the LP solver. We then end up with the following average route times:

| | Naive | Naive-LP | Greedy | Greedy-LP |
|---|---|---|---|---|
| Time | 7.461 | 5.684 | 5.238 | 5.228 |

Table 2: The average of the results for the quality test. The results are given in hours

Clearly the LP solving of a path is better, as shown in Table 2. Interestingly there is only a difference of 0.198% in the greedy solution of a path compared with LP. Also, it is interesting to see that solving the path chosen by the naive algorithm, with LP achieves a better time, but *not better* than the greedy route time. The difference between Naive-LP and greedy is 7.845%. This means that the path chosen plays a significant role in the overall time.

# 7. CONCLUSION

As the adoption rate of EVs increases, time-optimised route planning systems for electrical vehicles will become increasingly important. We showed that a route planning system for EVs should take charging stations and charging rates into account, since the time spend charging an EV with today's standards is still relatively long.

The problem of finding a route plan for an EV can be modelled as an optimisation problem. We designed a greedy heuristic algorithm that uses greedy choices and heuristics about the EV and the road network to find time-optimised paths. The worst-case time complexity of the greedy heuristic algorithm is $O(|V||E| + |V^2| \log |V|)$. We ran several experiments on a real world road network and compared the performance with a naive algorithm that simulated the pattern of a naive electric vehicle driver. The experiments showed that the greedy heuristic algorithm both finds better

[3]The LP tool used is the GNU Linear Programming Kit (GLPK). GLPK is intended for solving large-scale LP problems, mixed integer programming problems [11]

paths and better charging plans than the naive algorithm. Along with this, we concluded that the greedy algorithm works best with a charging density below 40km.

We further designed a linear programming solution, which was able to solve a path approximately optimal using linearization. We concluded that using the LP solution for an entire road network is simply too expensive to compute within reasonable time.

Lastly, we assessed the quality of the paths picked by the greedy heuristic algorithm by comparing it with the LP path solver. The results showed that the route time of the greedy heuristic algorithm was 0.1% worse than the route time of the LP path solver when solving the same path.

# 8. REFERENCES


[1] Jim Henry. New data shows accelerating sales of plug-in electric cars. `http://www.forbes.com/sites/jimhenry/`, 2013.

[2] Musk Elon and Straubel JB. Model s efficiency and range. `http://www.teslamotors.com/fr_CH/blog/model-s-efficiency-and-range`, 2012.

[3] International Energy Agency. Global ev outlook understanding the electric vehicle landscape to 2020. `http://www.iea.org/publications/globalevoutlook_2013.pdf`, 2013.

[4] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[5] Hans C Joksch. The shortest route problem with constraints. *Journal of Mathematical analysis and applications*, 14(2):191–197, 1966.

[6] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[7] Andreas Artmeier, Julian Haselmayr, Martin Leucker, and Martin Sachenbacher. The shortest path problem revisited: Optimal routing for electric vehicles. In *KI 2010: Advances in Artificial Intelligence*, pages 309–316. Springer, 2010.

[8] Katty G. Murty and Santosh N. Kabadi. Some np-complete problems in quadratic and nonlinear programming. `http://www-personal.umich.edu/~murty/np.pdf`, 1987.

[9] Ravindra K Ahuja and James B Orlin. A capacity scaling algorithm for the constrained maximum flow problem. *Networks*, 25(2):89–98, 1995.

[10] Robbie Morrison. Sos2 constraints in glpk. `http://winglpk.sourceforge.net/media/glpk-sos2_02.pdf`, December 2008.

[11] GNU. Glpk (gnu linear programming kit). `http://www.gnu.org/software/glpk/`, 2012.