VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

FACULTY OF ELECTRICAL ENGINEERING AND
COMMUNICATION
DEPARTMENT OF CONTROL AND INSTRUMENTATION

OPTIMAL ROUTE PLANNING FOR ELECTRIC VEHICLES

PLÁNOVÁNÍ OPTIMÁLNÍ TRASY PRO ELEKTRICKÁ VOZIDLA

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. TOMÁŠ JUŘÍK
AUTHOR

VEDOUCÍ PRÁCE doc. Ing. PETR FIEDLER, Ph.D.
SUPERVISOR

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta elektrotechniky a komunikačních technologií
Ústav automatizace a měřicí techniky

Diplomová práce

magisterský navazující studijní obor
Kybernetika, automatizace a měření

Student: Bc. Tomáš Juřík
Ročník: 2

ID: 109667
Akademický rok: 2012/13

NÁZEV TÉMATU:

Optimal Route Planning for Electric Vehicles

POKYNY PRO VYPRACOVÁNÍ:

The task is to design and test algorithms capable of calculating a route to destination based on real-time and static information available for electric vehicles.
Firstly, focus on analysis and treatment of information sources such as the real-time traffic information flow obtained from SYTADIN as well as those obtained from Open-Street Map and NASA's Shuttle Radar Topography Mission (SRTM).
Secondly, create a modelling of the road segment energy consumption and propose algorithmic solutions to the route planning problem.
Thirdly, implement the chosen algorithms from the preceding phase and implement them on an Android Smart Device (Galaxy Tab) and validate the results by real tests. This internship project is supported by AKKA Technologies (http://www.akka.eu/), our industrial partner.

DOPORUČENÁ LITERATURA:

[1] A. Hrazdira, B. Rezende, et al.: Optimal Energy Management of an Hybrid Electric Vehicle, Internal report, Embedded System department, ESIEE Paris, 2012
[2] Ehsani, M.; Gao, Y.; Emadi, A.: Modern Electric, Hybrid Electric and Fuel-Cell Vehicles. Fundamentals, Theory and Design. Second Edition. 2010.
[3] SYTADIN: A real Time Traffic Information of Paris Area, http://www.SYTADIN.fr/

Termín zadání: 11.2.2013

Termín odevzdání: 9.8.2013

Vedoucí práce: doc. Ing. Petr Fiedler, Ph.D.
Konzultanti diplomové práce:

[illegible]

doc. Ing. Václav Jirsík, CSc.
předseda oborové rady

UPOZORNĚNÍ:

Autor diplomové práce nesmí při vytváření diplomové práce porušit autorská práva třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta elektrotechniky a komunikačních technologií
Ústav automatizace a měřicí techniky

ABSTRAKT

V této práci popisujeme algoritmy, které umí vypočítat trasy pro elektické vozidla. Tyto trasy mohou být vypočítány v závislosti na jednoduchých metrikách, jako jsou například vzdálenost a doba dojezdu, nebo v závislosti na pokročilejší metrice, jako je například energeticky optimální metrika. Tato metrika je parametrizovatelná konstrukcí elektrického vozidla. Dále popisujeme nový algoritmus, který vypočítá energeticky optimální trasy, které jsou více přijatelné pro řidiče, protože zároveň zohledňují metriku času při výpočtu trasy.

KLÍČOVÁ SLOVA

Plánování optimální trasy, Elektrická vozidla, Plánování trasy s vícero omezeními, Plánování optimální trasy s vícero omezeními, Optimální trasa, SYTADIN, Open Street Map, Shuttle Radar Topography Mission, Android.

ABSTRACT

In this work we present algorithms that are capable of calculating paths to destination for electric vehicles. These paths can be based on the simple metrics such as the distance, time or the paths can be based on more advanced metric such as the minimum energy demanding metric. This metric is parameterizable by the physical construction of the electrical vehicle. We also propose a new algorithm that computes energy optimal paths that are more acceptable by the driver, because it also takes into consideration the time metric while computing the path.

KEYWORDS

Optimal Route Planning, Electric Vehicles, Multiconstrained path, Multiconstrained optimal path, Shortest path, SYTADIN, Open Steet Map, Shuttle Radar Topography Mission, Android.

JUŘÍK, T. Optimal Route Planning for Electric Vehicles. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2013. 56 s. Vedoucí diplomové práce doc. Ing. Petr Fiedler, Ph.D.

PROHLÁŠENÍ

PODĚKOVÁNÍ

Rád bych poděkoval mému vedoucímu panu Arbenovi Celovi a jeho kolegům z ESIEE Paris za odborné vedení, konzultace, trpělivost a cenné rady k diplomové práci. Chtěl bych také poděkovat Františkovi Zezulkovi za aktivní podporu a umožnění pracovat na mé diplomové práci na zahraniční univerzitě.

V Brně dne 31. července 2013 ....

(podpis autora)

CONTENTS

LIST OF FIGURES

INTRODUCTION

The task of this work is to design and test algorithms capable of calculating a route to destination for electric vehicles. The routing algorithm is an algorithm that finds a route between two nodes in the routing map. Well know and widely used algorithm used for this type of task is the Dijkstra's algorithm [1] and its variation A* algorithm [2]. These algorithms are preferred because of their lower computation complexity compared to other algorithms, which most of are actually unsuitable for routing in large graphs such as the road network. The road network, represented by graph, consists of millions of nodes and edges and we have to compute the path in acceptable time for the driver. From early research on such algorithms, we may assume, that any modern GPS navigation uses some kind of modification of the Dijkstra's algorithm.

To use the Dijkstra's algorithm we must be able to compare edges based on the required metric. This common metrics can be the shortest, the fastest path and in our case the lowest energy demanding path. Apparently, we can see, that we need some evaluation function which can compare paths and can clearly decide which of those paths is the optimal one. For the distance and time the computation of such function is clear, but for the energy we must have some more advanced model, which takes in consideration the physical parameters of the vehicle.

In the first part, this work describes how to model the electric vehicle and create a suitable formula for calculating energy cost so we can apply the Dijkstra's algorithm and some of its modifications to our problem.

In the second part, we propose a new algorithm that is able to find an energy optimal path, which is more likely acceptable by the driver. The need of such algorithm was discovered during the development of the energy metric, because we consider the energy optimal path as simply unacceptable for the driver, as it always prefers the slowest possible path.

In the third part, we briefly describe what data sources we use to create the routing graph and implement the prototype navigation. In the last part, tests show and confirm some of the characteristics that we assumed during the development of the prototype application.

1 ENERGY MODELING

To use the Dijkstra's algorithm we must be able to compare edges based on the energy demand needed to traverse the edge. In the following sections we create a mathematical model that will lead to obtaining an energy cost function. The modeling is described in several steps, where we define intermediary energy models, but only the last one is used in the implementation.

1.1 Bicycle model

In the first time the discussion of vehicle fundamentals will be restricted to one-dimensional movement (bicycle model). The movement behaviour of a vehicle along its moving direction is completely determined by all the forces acting on it in this direction. This basic model in its description is from [5].

According to the Newton's second law, vehicle acceleration can be written as

$$\frac{dV}{dt} = \frac{\sum F_t - \sum F_r}{\delta M} \tag{1.1}$$

where V is the speed of the vehicle, $\sum F_t$ is the total traction effort of the vehicle, $\sum F_r$ is the total resistance, M is the total mass of the vehicle, and δM is the mass factor that equivalently converts the rotational inertias of rotating components into translational mass.

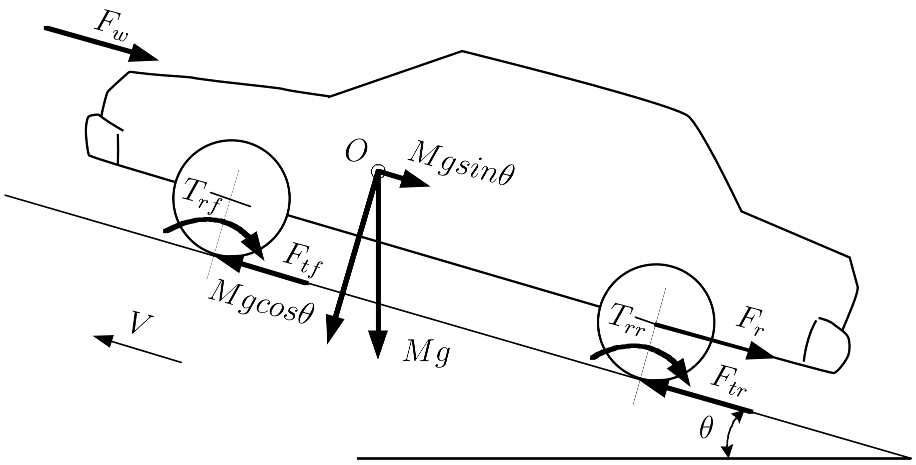


Fig. 1.1: Schematic representation of the forces acting on a vehicle [3]

As shown in figure 1.1, the total resistance is composed mainly of rolling resistance F_r (which combines the torques T_{rf} and T_{rr}), wind resistance F_w, and climbing resistance F_c (the term $Mg\sin\theta$). In the opposite direction acts total traction effort F_t which is composed of the front and rear tires efforts F_{tf} and F_{tr} generated by the prime movers and minus all friction losses. By developing the equation above, we obtain the dynamic equation in the form:

$$M\frac{dV}{dt} = F_t - (F_r + F_w + F_c) \tag{1.2}$$

Using the dynamic equation, we can express the maximal traction effort for both front and rear wheels. Then we can also examine the tire-ground adhesion which is connected with the maximal traction effort. The detailed description of the characteristics summarized in this chapter is presented in [4].

1.2 Longitudinal model

The vehicular model of longitudinal dynamics is constructed based on the theory of vehicle multi-body dynamics as seen before.

1.2.1 Wind resistance

The wind resistance is caused generally by two facts: the viscous friction of the surrounding air on the vehicle surface and the losses by the pressure difference between the front and the rear of the vehicle, generated by a separation of the air flow.

The resistance expression can be usually simplified by considering a vehicle as a prismatic body with a frontal area A_f. The pressure difference force can by multiplied by an aerodynamic drag coefficient C_d also known as Reynold's coefficient (estimated by simulation or experiment in wind tunnel) so the wind resistance can be written as

$$F_w(v) = \frac{\rho}{2} A_f C_d v^2 \tag{1.3}$$

where v is the vehicle speed and ρ is the density of the ambient air.

1.2.2 Tire rolling resistance

The tire rolling resistance can be expressed as

$$F_r(\theta) = Mg\mu_r \cos\theta,\ v > 0 \tag{1.4}$$

where M is the vehicle mass, g the acceleration due to gravity, term $\cos\theta$ is road slope influence and μ_r is the rolling friction coefficient.

The rolling friction coefficient depends on many variables. Especially vehicle speed v, tire pressure p, and road surface conditions.

1.2.3 Climbing resistance

The climbing resistance is conservative force induced by gravity and considerably influences the vehicle behaviour.

$$F_c(\theta) = Mg\sin\theta \tag{1.5}$$

1.2.4 Acceleration resistance

The acceleration resistance due to the inertia of the vehicle and of all rotating parts inside the vehicle causes frictious (d'Alembert) forces. The rotating parts are here represented by

$$F_a(\dot{v}) = M(1+\delta_{eqm})\frac{dv}{dt} \tag{1.6}$$

where δ_{eqm} is the vehicle equivalent moment inertia representing the rotating parts such as wheels and powertrain inertia.

According to an analysis of the summation of performing forces acting on the vehicle body in the longitudinal direction, the power balance for the controlled vehicle is governed by:

$$\begin{aligned} P_D\left(t\right) &= f_{P_D}\left(v,\dot{v},M,\theta\right) = \left(F_w+F_r+F_c+F_a\right)v \\ &= \frac{\rho}{2}A_f C_d v^3 + Mg\left(\mu_r\cos\theta+\sin\theta\right)v + M\left(1+\delta_{eqm}\right)\dot{v}v \end{aligned} \tag{1.7}$$

1.2.5 Position-based power calculation

As an important parameter in equation (1.7), the road grade may change frequently according to the actual road environment, especially in the mountain terrain, and has become a major impact on the energy consumption. In general, the proper measurement to record the road grade by GPS and 3D-map is dependent on position rather than time. Therefore, the time-based equation (1.7) is not suitable for the purpose of control algorithm design, and a power calculation depending on position is preferable. Using the transformation

$$v = \frac{ds}{dt} \tag{1.8}$$

$$\frac{dv}{ds} = \frac{dt}{ds}\frac{dv}{dt} = \frac{1}{v}\frac{dv}{dt}, v \neq 0 \tag{1.9}$$

equation (1.7) can be transferred into a position-based form

$$\begin{aligned} P_D(s) = {} & \tfrac{\rho}{2} A_f C_d v^3 + Mg\left(\mu_r \cos\theta + \sin\theta\right) v \\ & + M\left(1+\delta_{eqm}\right) v^2 \tfrac{dv}{ds}. \end{aligned} \tag{1.10}$$

The prototype vehicle to carry out the simulations is the *DaimlerChrysler* car [3]. The numerical values of the vehicle parameters used in further tests are listed in table 1.1.

Parameter		**Value**
M	total mass	2095 kg
C_d	Reynolds coefficient	0.32 kg/m^3
A_f	windward area	2.31 m^2
μ_r	rolling resistance coefficient	$8.80 \cdot 10^{-3}$
ρ	air density	1.25 kg/m^3
δ_{eqm}	equivalent moment inertia	0.195 kg·m^2

Tab. 1.1: Numerical values of the vehicle parameters [3]

1.3 Energy cost function model

Before continuing developing the model, we have to realize that the routing algorithm computes with only estimated and limited information. We do not know the speed profile along the road and the the exact road slope. This means that we consider that the velocity and the road slope is constant on each road segment. Taking into account these assumptions, we may following from the equation (1.10) give the energy needed to move the vehicle from the beginning to the end of the road segment (v_i, v_{i+1}) as:

$$E_D(i) = \frac{\rho}{2} A_f C_d v_i^2 l_i + Mg\mu_r l_i \cos\theta_i + Mgl_i \sin\theta_i \tag{1.11}$$

where the newly introduced l_i is the length of the road segment.

In the equation (1.11) the only term which may have negative value is the third one which corresponds to potential energy. If its absolute value is also superior to the sum of the two other terms, then the edge energy cost is negative. Lest us note by E_R the energy corresponding to wind and rolling resistance and by E_P the potential energy. We can define them as:

$$E_D(i) = E_R(i) + E_P(i) \tag{1.12}$$

$$E_R(i) = \frac{\rho}{2} A_f C_d v_i^2 l_i + Mg\mu_r l_i \cos\theta_i \tag{1.13}$$

$$E_P(i) = Mgl_i \sin\theta_i \tag{1.14}$$

Finally, the optimization criterion for the energy optimal route, which the algorithm is minimizing, may be given by the following expression:

$$J(P^j) = \sum_{i=0}^{i=N} E_R(i) + E_P(i) \tag{1.15}$$

where $P^j = (v_0^j, v_1^j, v_2^j, \ldots, v_N^j)$ is the j^{th} possible path.

1.3.1 Positive cost function transformation

We have defined the energy cost function (1.12) and deduced that under certain scenarios the resulting sum may be negative. This poses a significant complication because the Dijkstra's algorithm and its modifications can only work with non-negative edge costs. Therefore, we have to transform this cost to obtain only positive values.

The two components of (1.12) are rolling resistance energy E_R and the potential energy E_P. We are certain that the rolling resistance energy is always positive and that this energy is irreversibly lost. On the other hand, the potential energy is recoverable when the vehicle is descending. In the case of a perfect recuperation of the potential energy we could completely ignore this component in the equation, because if we take two different paths of different elevation profile, but between the same origin and destination, then the difference of the potential energy will be always the same for both of these paths.

In the case of imperfect recuperation, we only recuperate a fraction of the potential energy during the downhill movement. We introduce this recuperation efficiency with a coefficient α. In literature [7, 8, 9] we may find that in general cases a preprocessing with the Johnson's algorithm [6] is used to obtain a potential function, which solves the negative cost function, but there is also thoroughly deduced that the potential energy is already such one potential function. Therefore, using the transformation described in [7, 8, 9], we obtain a new energy cost function

$$E_D^m(i) = \begin{cases} E_R(i) + (\alpha - 1) \cdot E_P(i) & \text{if } E_P(i) \leq 0 \\ E_R(i) & \text{if } E_P(i) > 0 \end{cases} \tag{1.16}$$

where α is the downhill energy recuperation coefficient ($0 < \alpha < 1$). The transformed function is always positive and can be used with the Dijkstra's algorithm in this form. This transformation introduces a penalization of the descending road segments in a way that when the vehicle moves downhill $E_P \leq 0$ we know that a fraction $\alpha - 1$ of the potential energy is always lost, because even if the vehicle moves later upwards, to the same elevation, it can only recuperate α of the potential energy.

1.3.2 State of charge

Additionally, we also want to consider the state of the energy source of the vehicle. This source may be a battery or super-capacitor. We consider the state of charge in simple and intuitive way such that if the remaining energy is not sufficient to move the vehicle to the next vertex of the road segment its cost is set to infinity.

The ideas used in this implementation are from [8] and it is actually a further transformation of the energy cost function. Firstly, we define the state of charge function as

$$C(r_i) = \begin{cases} 0 & \text{if} \quad i = 0 \\ \Delta^i & \text{if} \quad i > 0,\, \Delta^i \leq C_{init} \\ \infty & \text{if} \quad i > 0,\, \Delta^i > C_{init} \end{cases} \tag{1.17}$$

where the $r_i = (v_0, v_1, v_2, \ldots, v_{i-1})$ is a sequence of nodes taken to the current node v_i, C_{init} is the initial state of charge at the origin and Δ^i is the difference to the state of charge after we move to the current node and is defined as

$$\Delta^{i+1} = C(r_i) + E_D^m(i+1). \tag{1.18}$$

The C is a function that tell us the state of charge base on what nodes we have taken to the current node. The value is actually the free capacity and not the amount of stored energy and it is lowering with recuperation and rising with using energy. We can see that there are three cases, the fist one means that in the beginning the we have zero free capacity, the energy source is fully charged, the second case is that we have enough of energy to move to the next node and the third last case is that we do not have enough energy and so the returned value is infinity.

As we said before we need to modify the energy cost function, this is introduced by another additive component. We denote this component as $\hat{E}_D$ and it is defined as

$$\hat{E}_D(i) = \begin{cases} 0 & \text{if} \quad \Delta^i \leq C_{init} \\ \infty & \text{if} \quad \Delta^i > C_{init} \end{cases} \tag{1.19}$$

where we can see that there are two cases. In the first one zero is returned when the energy needed to reach the next node is less than the initial state of charge. In the second one infinity is returned because we have depleted the stored energy.

We follow with insertion of this additive component into the equation (1.12) as

$$E_D(i) = E_R(i) + E_P(i) + \hat{E}_D(i). \tag{1.20}$$

Furthermore, we apply the transformation to the positive edges as demonstrated in (1.16) and finally obtain the function as

$$E_D^m(i) = \begin{cases} E_R(i) + (\alpha - 1) \cdot E_P(i) + \hat{E}_D(i) & \text{if } E_P(i) \leq 0 \\ E_R(i) + \hat{E}_D(i) & \text{if } E_P(i) > 0 \end{cases} \tag{1.21}$$

The effect of this modification is that if the energy needed to reach to a current node is grater than the initial state of charge plus the recuperated energy along the way, then we do not consider this path feasible.

1.3.3 Battery maximum capacity

In the previous section we described how penalize the insufficient energy to move forward. Here we slightly modify the definition of the given function in such way that the mode also penalize overcharging of the battery. What it means is, that if the energy storage is full, either because of initial full charge or excessive recuperation, and we are moving downhill, then we are not able to recuperate anymore thus the potential energy is lost.

The modification to the previous functions is following

$$C(r_i) = \begin{cases} C_{max} - C_{init} & \text{if} \quad i = 0 \\ 0 & \text{if} \quad i > 0,\, \Delta^i < 0 \\ \Delta^i & \text{if} \quad i > 0,\, 0 \leq \Delta^i \leq C_{max} \\ \infty & \text{if} \quad i > 0,\, \Delta^i > C_{max} \end{cases} \tag{1.22}$$

where the C_{max} is the maxim capacity of the energy storage. The first term sets the free capacity to the difference of the initial capacity and maximum capacity. In the former definition the initial capacity was the same as the maximum capacity, here we can define different values. The second term handles the situation when the energy source is fully charged and so there is zero free capacity left. The overcharging penalty is finally introduced in

$$\hat{E}_D(i) = \begin{cases} -\Delta^i & \text{if} \quad \Delta^i < 0 \\ 0 & \text{if} \quad 0 < \Delta^i \leq C_{max} \\ \infty & \text{if} \quad \Delta^i > C_{max} \end{cases} \tag{1.23}$$

where the first term is added and it means that if we are trying to recuperate and the free capacity is smaller than we return amount of energy that could not be stored.

The effect of this modification is that if a downhill segment potential energy can not be recuperated because of lack of additional capacity, then the cost of the energy function is increased with the value of this lost energy.

2 ALGORITHM FOR FINDING AN ENERGY OPTIMAL PATH SUBJECT TO A TIME CONSTRAINT

2.1 Multiconstrained path (MCP) problem

A path P in G is a sequence of nodes $\langle v_1, \ldots, v_k \rangle$ with $(v_i, v_{i+1}) \in E$ for all $i = 1, \ldots, k-1$. For $1 \leq i \leq j \leq k$ the subpath of P between v_i abd v_j is denoted by $P_{v_i \to v_j} = \langle v_i, \ldots, v_j \rangle$.

The cost function can be extended to this path as the accumulated sum of all edge costs, see Figure 2.1:

$$c(P) = \sum_{i=1}^{k-1} c(v_i, v_{i+1}) \tag{2.1}$$

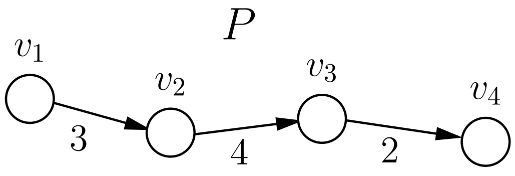


Fig. 2.1: A path $P = \langle v_1, v_2, v_3, v_4 \rangle$. The numbers on the edges denote the edge cost, so $c(P) = 9$

In order to be consistent with the definitions used in the literature and to help the reader to easily understand the notations used here, we redefine in the following the edge (u, v) energy cost function c_e (2.2) and introduce the related edge time cost function c_t (2.3).

$$c_e(u,v) = \begin{cases} E_R(u,v) + (\alpha - 1) \cdot E_P(u,v) + \hat{E}_D(u,v) & \text{if } E_P(u,v) \leq 0 \\ E_R(u,v) + \hat{E}_D(u,v) & \text{if } E_P(u,v) > 0 \end{cases} \tag{2.2}$$

$$c_t(u,v) = l(u,v)/v(u,v) \tag{2.3}$$

Definition [10]: Consider a graph $G = (V, E)$. Each edge $(u, v) \in E$ is specified by a link weight vector with m additive edge weights $w_i(u, v) \geq 0$ for all $1 \leq i \leq m$. Given m constraints L_i, where $1 \leq i \leq m$, the problem is to find a path P from a source node s to a destination node t such that

$$w_i(P) = \sum_{(u,v)\in P} w_i(u,v) \leq L_i \tag{2.4}$$

for all $1 \leq i \leq m$.

A path that satisfies all m constraints is referred to as a feasible path. There may be many different paths in the graph $G = (V, E)$ that satisfy the constraints. Any of these paths is a solution to the MCP problem. However, it might be desirable to retrieve the path with smallest cost $c(P)$ from the set of feasible paths. The problem that additionally optimizes some cost function $c(P)$ is called the *multiconstrained optimal path problem* and is formally defined as follows.

2.1.1 Multiconstrained optimal path (MCOP) problem

Definition [10]: Consider a graph $G = (V, E)$. Each edge $(u, v) \in E$ is specified by a link weight vector with m additive edge weights $w_i(u, v) \geq 0$ for all $1 \leq i \leq m$. Given m constraints L_i, where $1 \leq i \leq m$, the problem is to find a path P from a source node s to a destination node t satisfying (2.4) and, in addition, minimizing cost function such that $c(P)$ is less than or equal to the cost of any other (s, t) path in the graph. In other words, $c(P) \leq c(P^j)$ for all paths P^j between s and t.

2.1.2 K-shortest path problem

A related problem is the *k-shortest path problem*, where the goal is to find just not the cheapest path, but find k cheapest paths in the graph. A solution to this problem is a set of (s, t) paths, that can be arranged like $P_1, P_2, \ldots, P_K$ with costs $c(P_1) \leq c(P_2) \leq \ldots \leq c(P_K)$ so that all other paths than $P_1, P_2, \ldots, P_K$ have a greater or equal costs compared to $c(P_K)$.

2.1.3 Concordance of the definitions

To apply the former definitions to our problem we give the following explanation. In our case each edge $(u, v) \in E$ is characterized by two weight functions that assign positive weights, which are the energy cost function $c_e(u, v)$ (2.2) and the time cost function $c_t(u, v)$ (2.3). The energy and time can be seen as the constrained resources according to the MCOP definition. However, the energy cost function has already the constrain logic included, because we track the free capacity of the energy source and if it reaches the empty state the energy cost is set to infinity (1.23), thus this path is effectively pruned. Therefore, we do not have to consider the energy cost in the same sense as it is defined by the MCOP definition. Consequently, we are left with only the time constraint and as a result we can deduce for the MCOP definition the following:

$$\begin{aligned} c(u,v) &= c_e(u,v) \\ w(u,v) &= c_t(u,v) \\ L &= \beta \cdot c_t\left(P^{sp}_{s\to t}\right) \end{aligned} \tag{2.5}$$

Where $c(u,v)$ is the function whom cost is to be minimized, $w_1(u,v)$ is the additive constraining weight. The $c_t\left(P^{sp}_{s\to t}\right)$ is the shortest path length for the time cost function, it can be equally understood as the shortest possible time a driver can get from s to t, it is also called time *lower bound*. This maximum time limit is denoted as L and it is also called time *upper bound* or time constraint. The β is a parameter which allow us to obtain L by multiplying it with the time lower bound. Simply said the β tell us how much more time the driver is willing to spend relatively to the time optimal path.

The values of $\beta \in (1,\infty)$ define the objective of the routing problem. If the value of $\beta = 1$ we have a time optimal routing problem. On the other hand, if $\beta = \infty$ we have an energy optimal routing problem. A condition that $\beta \geq 1$ has to be met, otherwise we would be querying for a path that is faster than the fastest possible path.

2.2 Solutions

2.2.1 Generate and test

Since the shortest path problem (SP) is easy, it might be a good idea to use it in a strategy to solve the MCOP. A straightforward way to do this is to solve a SP instead of a MCOP, then check if the solution violates some of the resource constraints. If it does not, then the optimal solution to the MCOP is found. If some constraint is violated the second shortest path is produced and so on.

The k-shortest path problem is used for this scheme in practice, due to efficiency. Produce the k shortest paths and then pick the shortest of these that does not violate the constraints. The main key point of this approach is related to the choice of the size of k in order to guarantee a feasible solution. In the worst case all paths has to be enumerated, since the most expensive might be the only feasible path. If the constraints are not that limiting in the sense that not many paths are illegal, this might be a good solution technique.

2.2.2 Dynamic programming

Dynamic programming is a method for solving complex problems by breaking them down into simpler subproblems. Every sub-problem is solved in order to get a solution to the entire problem. In the case of a shortest path problem the dynamic programming approach is to use the fact that an optimal (s, t) path is built up from optimal sub-paths. If a path including node v is used, then the path from s to v is optimal. If you have the optimal path from s to all nodes that are neighbors to the source node, then it is easy to compute the shortest path.

Dijkstra's algorithm uses this, like all shortest path algorithms. Compute the shortest path to all nodes one edge away from the source, save the distance from the source, then continue with the nodes two edges away from the source, using information from the step before, save the distance from the source and from which previous node the shortest path originated.

The scheme described above works fine for shortest path but for the MCOP there is a problem. In the SP it is easy to decide if one path is better than another, if it is cheaper, is better. For the MCOP problem a cheap path that consumes a lot of resources early will probably be worthless in a later step since it will become illegal, but a seemingly expensive path in an early stage might prove to be optimal. Because of this more than one path has to be stored. The question is then how many of the paths should be saved at each node. For the SP all paths but the optimal one can be disregarded at each node, this is impossible for the MCOP. The paths that become infeasible along the way can of course be disregarded, but not counting these there are still a large number of candidate paths.

2.3 Proposed solution

We propose a new algorithm that finds a path whose energy cost is optimal and also the time cost is at most β times the cost of the time optimal path. The algorithm is based on a combination of ideas that have been used and proved in [10], [11] and [8]. The core algorithm is based on the *SAMCRA* [10] and *A*Prune* [11] algorithms, which are a modification of Dijkstra's algorithm that finds k shortest paths, where not only the optimal path but also many candidate paths are tentatively stored. Additionally, we combine the *A** search speedup technique for which we calculate the lower bounds with a classic Dijkstra's algorithm going backward. During the relaxation step we employ pruning techniques called *path dominance* and *look-ahead*, both effectively reduce the number of paths that would otherwise needed to be explored. These techniques are explained in the following sections. Moreover, we also use the modified energy cost function mentioned earlier because we are still required to have a non-negative edge costs. The resulting algorithm with these concepts is executed in the following sequence of steps:

I. Time lower bounds computation

To speedup the algorithm we employ the look-ahead technique, which can be described as following. We are searching forward in the graph from origin s and we have arrived at node v. We know that we have already traveled 10 minutes from s to v and we also know that we may spend 15 minutes at maximum. This tells us that we have five minutes left to spend to reach the destination t. The fastest way to decide if we can or cannot reach the destination in this time is *to actually know the shortest possible time* from the actual node v to destination t. This shortest possible time is called as a lower bound. For example, if the lower bound is two minutes then we are able to reach the destination from this node in 12 minutes or longer, thus we can continue to search from this node. However, if the lower bound would be for example seven minutes, then we are certain we cannot reach the destination within the 15 minutes. Consequently, we may disregard the path we took to reach this node v as it is unfeasible, this is also called as *prunning*.

In this example the 15 minutes is the constraint L defined in (2.5). The 10 minutes path we call as a *tail path*, the two or seven minutes path are called *head path* and the joint of them is called *projected path*.

Hence, to be able to use this technique, we are precomputing these head paths for a subset of nodes from the graph in this first step.

II. Energy heuristic computation

Another, well known, speedup technique reducing the search space is the A* algorithm. It is a modification of the Dijkstra's algorithm where instead of using only tail path cost to determine the priority in the queue we use the whole projected path. To know the projected path we also need to know the estimate of energy to reach the destination. This estimate is given by a heuristic function, which must be a so called admissible heuristic. That is, it must not overestimate the projected path cost. Usually a function based on a straight-line to the goal is used, since that is physically the smallest possible distance between any two nodes. However, this heuristic is usually weak and does not limit the search space much, thus we use a Dijkstra distance as the heuristic function which provides stronger estimates.

III. Forward constrained search

This is the main part of the algorithm. We use the precomputed information from the two former steps, which are indispensable to effective pruning. The search is terminated once the path is found or there is no more possible paths to explore.

2.3.1 Time lower bounds computation for the look-ahead concept

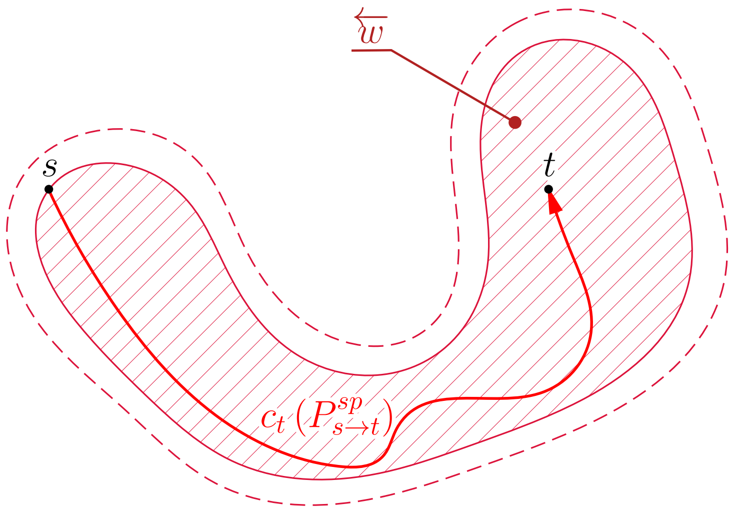


Fig. 2.2: Example search space of the computation when the shortest path is found and this search is stopped. Nodes within the hatched area have the head path computed in the $\overleftarrow{w}$ set. The dashed line marks the area of nodes that can be reached within the constraint L. The search space is not displayed as a disc, because the axes of this figure are in units of distance whereas the search space is in time, so this case can be seen as that the fastest route is achieved by driving a motorway around the city.

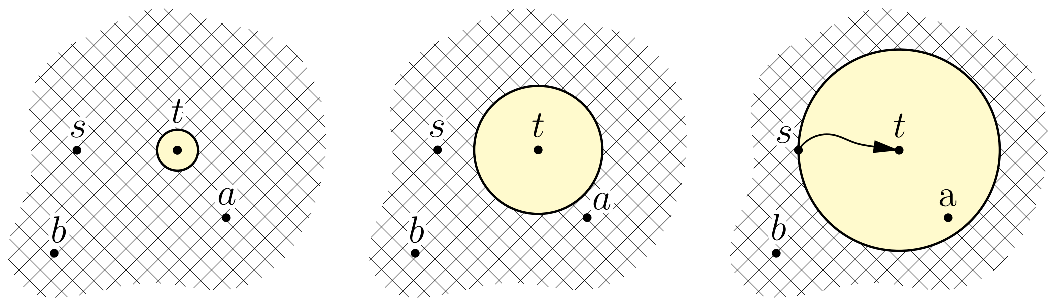
The look-ahead concept can be viewed as an additional mechanism to reduce the search space of possible paths. The idea, is to limit the set of possible paths by using information of the remaining subpath towards the destination. The look-ahead concept proposes to compute the shortest path tree rooted at the destination to each node in the graph. The basic importance of look-ahead is to provide each node with an exact, lower bound cost.

Considering this description of the look-ahead concept from [10], we have to realize that we cannot precompute the lower bound for all nodes in the graph because simply the graph is too large and it would mean that we have to compute Dijkstra distance for the whole graph. Such precomputation would take more time that the main part of the algorithm.

To overcome this problem we have, firstly, used a stopping condition in the Dijkstra's algorithm which ensured that we do the computation only for those nodes that are reachable within the given time constraint. We start from the destination node t towards the origin s, when the origin is settled we store the calculated cost, which in this case represents the minimum time that is needed to travel from s to t, or in other words, it is the cost $c_t\left(P_{s\to t}^{sp}\right)$ for the shortest path of the time cost function c_t. However, at this moment we do not stop the search as we normally would, but we keep the computation running until the summed cost $c_t\left(P_{v\to t}\right)$ for some node $v \in V$ is greater than $\beta \cdot c_t\left(P_{s\to t}^{sp}\right)$ or there are no other unsettled nodes. When the search stops we have a lower bound set that contain an estimated minimal time needed to reach a destination from that node. We denote this set as $\overleftarrow{w}(v)$, which contains the distance for every node v that meets the condition (2.6).

$$c_t\left(P_{v\to t}\right) \leq \beta \cdot c_t\left(P_{s\to t}^{sp}\right) \tag{2.6}$$

The idea behind this is that we will compute the distance only for those nodes that can eventually be part of the solution. In addition, we are certain that if a node is not included in this set then we cannot reach it within the given time constraint. Nevertheless, this still meant that with increasing β the search space was also becoming larger and eventually useless for large β (e.g. ≥ 2).



(a) Backward search starts from destination node t

(b) The search area grows as a disc around the destination node

(c) Once the origin node s is settled the search stops

Fig. 2.3: Schematic representation of backwards dijkstra search space growth

Realizing that we are computing lower bound, for which we may choose a value that is less than or equal to the exact distance from t, we can avoid all of this overhead. The following explanation is illustrated by Figure 2.3. The backward search starts from the destination node t and grows towards the origin node s. Any node directly on the circle has the same Dijkstra distance, time in this case, to the destination node. Once the origin node is settled we stop the search and we know what is the shortest distance between origin s and destination t. There are also marked nodes a and b. We can see that the node a is within the search space (disc) whereas the node b is not. This means that node a has also been settled and we know the shortest distance, the lower bounds, between this node and the destination t. Apparently, the distance is not know for the node b, yet we are sure that the distance is greater than the one between s and t. This is noted by (2.7).

$$c(P_{a \to t}) < c(P_{s \to t}) < c(P_{b \to t}) \tag{2.7}$$

In conclusion, we may use the distance between the s and t as the lower bound for the node b, because we are certain that we do not overestimate the distance. Consequently, we use the distance $c(P_{s \to t})$ for all the nodes that are not within the search space as their lower bound.

2.3.2 The backward Dijkstra's algorithm

We use a priority queue Q that is initialized with the destination node t. Each node u has associated a tentative distance $h(u)$ from t, which is the only component that decides the priority within the queue. The algorithm works in the same fashion as the classic Dijkstra's algorithm, but the predecessor list is not stored, because we are only interested in obtaining the distances from all vertices $v \in V$ that have cost $c(P_{v \to t}) \leq c(P_{s \to t})$ to destination t. This procedure is given in Algorithm 1.

Algorithm 1 Backward Heuristic exploration by Dijkstra's algorithm

Input: Directed weighted graph $G = (V, E)$, source vertex s, destination vertex t, cost function c

Output: Lower bounds set h representing estimated accumulated cost to destination t from all vertices $v \in V$ that have cost $c(P_{v \to t}) \leq c(P_{s \to t})$

```
 1: procedure DIJKSTRA($G$, $s$, $t$, $c$)
 2:     $h(t) \leftarrow 0$;
 3:     $Q \leftarrow \{t\}$;
 4:     while $Q \neq \emptyset$ do
 5:         $u \leftarrow$ choose the node with minimal $h(u)$ from $Q$;
 6:         $Q \leftarrow Q \setminus u$;
 7:         if $u = s$ then
 8:             return $h$;
 9:         end if
10:         for all ingoing edges $(v, u)$ of node $u$ do
11:             $h' \leftarrow h(u) + c(v, u)$;
12:             if $v \notin Q$ then
13:                 $h(v) \leftarrow h'$;
14:                 $Q \leftarrow Q \cup v$;
15:             else if $h(v) > h'$ then
16:                 $h(v) \leftarrow h'$;
17:             end if
18:         end for
19:     end while
20: end procedure
```

2.3.3 Energy heuristic computation

Since we are using the Dijkstra's distances as the heuristic, we can use identical computation as the one used for the time lower bounds in the previous section. We are using the same procedure given in Algorithm 1. The difference is, that instead of the time cost function c_t, we are using the energy cost function c_e. Although, we cannot use the energy cost function as it is defined in (2.2), because this search goes backward from the destination t and we do not know what state of charge of the energy source will be at the destination. We only know what is the state of charge at the origin node and we will know (estimate) the state of charge only after the energy optimal path is found.

Therefore, we need to remove the dependence on the state of charge. We know that the state of charge is used to penalize the states of overcharging or empty battery. Additionally, we know that this two extreme states can only increase the cost, so if we decide to ignore these two extreme states and the possible increase of

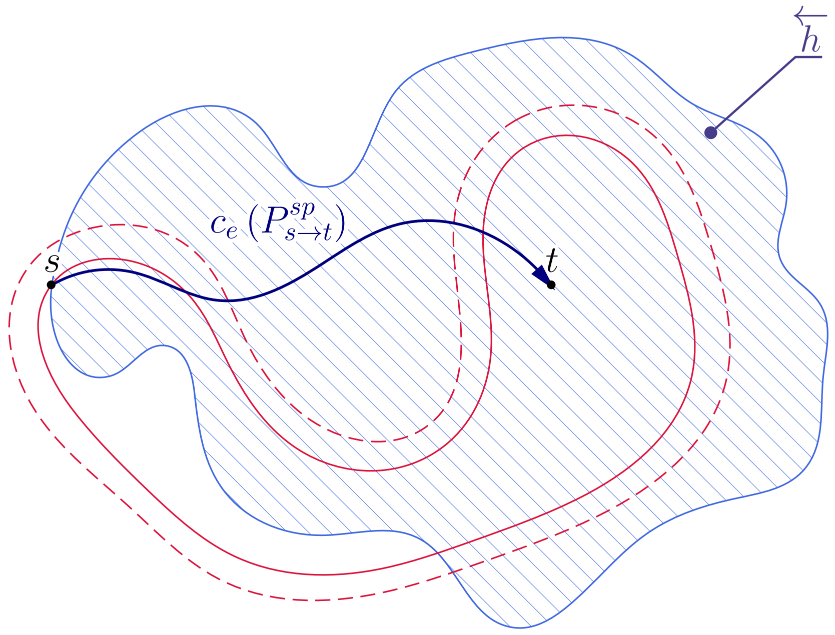


Fig. 2.4: Example search space of the computation when the shortest path is found and this search is stopped. Nodes within the hatched area have their head path's cost computed in the $\overleftarrow{h}$ set. We can see that the search space and the optimal path is different from the time search (see Fig. 2.2). This can be explained by the fact that the energy function prefers slower and more direct (shorter) paths.

cost. Moreover, we can prove that we may afford this modification by realizing that to have an admissible heuristic, we must not overestimate the projected path cost. Our modification is in concordance with this definition.

2.3.4 Forward constrained search

The third step of the algorithm is the forward search where the actual path is found. On Figure 2.5 we can see how the resulting path might look like regarding the precomputation steps. The search space is hatched in green and we can notice that it never crosses the red dashed line, which represents the time constraint boundary that was settled by the time lower bounds precomutation step. We can claim that if any node located outside of this boundary can not be part of the feasible solution, because it would break the time constraint, and so if any path attempts to expand over this boundary it is pruned. In reality the pruning happens even in larger distance from this border due to the look-ahead concept. The algorithm is given in Algorithm 3.

Using the terminologies described in the previous sections, the algorithm can be described by words as: starting from the path $P_{s\to s}$, potentially, all the paths $P_{s\to v}$ where $v \in V$ can be reached. However, with a proper pruning against the

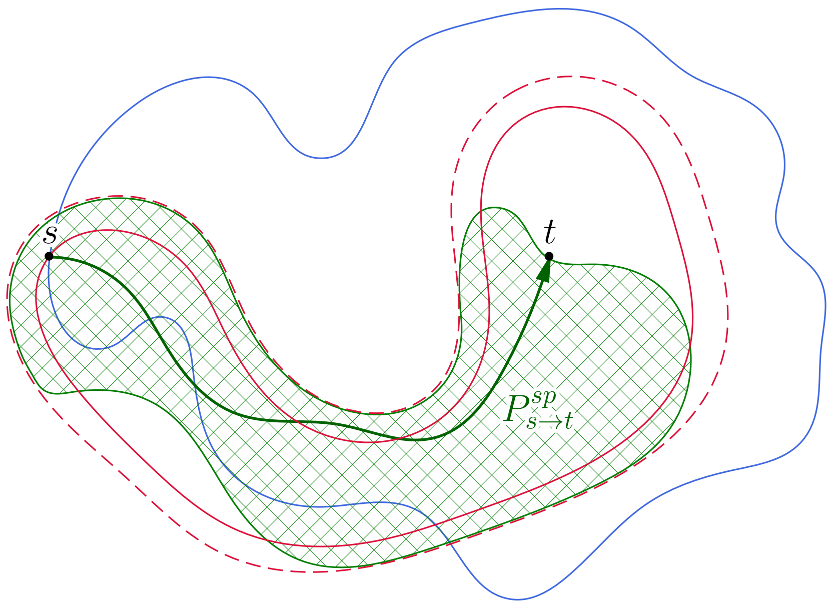


Fig. 2.5: Forward constrained search space

given constraint L, only the admissible paths remain as candidate paths for further expanding. We call a path admissible if sum of time, or other resource, spent on the tail path $\overrightarrow{w}(P_{s\to v})$ and the estimated cost of the head path $\overleftarrow{w}(v)$ is lower than the constrain, this is expressed by (2.3.5). Furthermore, the candidate paths are ordered properly such that the path with lowest projected cost $\overrightarrow{h}(P_{s\to v}) + \overleftarrow{h}(v)$ is selected and expanded first, then we can terminate our expansion procedure once we have found the constrained shortest path or there are no candidate paths left.

$$\overrightarrow{w}(P_{s\to v}) + \overleftarrow{w}(v) < L \tag{2.8}$$

2.3.5 Components of the projected cost of the look-ahead technique

Further explanation of the look-ahead technique and of the used notation can be seen on Figure 2.6. We can see a red disc grown from the destination node t. It represents the search space for the time lower bounds precomputation, as explained previously in the description of the algorithm's first step. The green disc grows from the origin node s, but in reality during the forward search the search space is not disc, but some uncertain region protruding to the destination. The algorithm state, as shown on the Figure 2.6, is at the step of deciding if a path $P_{s\to v}$ is considered admissible. If the condition given by equation is true then the path $P_{s\to v}$ will be added to the priority queue, otherwise it will be pruned and forgotten. Simple example is: we have reached an intermediate node v in 10 minutes and the fastest path from here to destination takes at least 5 minutes; however, our limit, the constraint, is 12 minutes; consequently, so we can clearly see that there is no need to expand this path, as it will never lead to feasible path.

The look-ahead, or lower bound, is denoted as $\overleftarrow{w}(v)$. We can simply write just $\overleftarrow{w}(v)$ instead of $\overleftarrow{w}(P_{v\to t})$ because there is always only one shortest path from intermediate node v to t. On the other hand, there can be many expanded paths, tail paths, from s to v. The time spent on the tail path to this intermediate node v is denoted as $\overrightarrow{w}(P_{s\to v})$.

In the Algorithm 3 we can see that the constrain L is substituted as $L = \beta \cdot c_t\left(P^{sp}_{s\to t}\right) = \beta \cdot \overleftarrow{w}(s)$. This is based on (2.5) and on the fact the we also know that $\overleftarrow{w}(s)$ is the fastest path cost from s to t.

On Figure 2.6 there is also displayed an alternative node b, which is outside of the look-ahead search space, the red disc. Therefore, the exact distance to the target $\overleftarrow{w}(b)$ is not known, but as explained earlier, we can use for evaluation of the node b the shorted distance to the origin $\overleftarrow{w}(s)$ as the estimated distance.

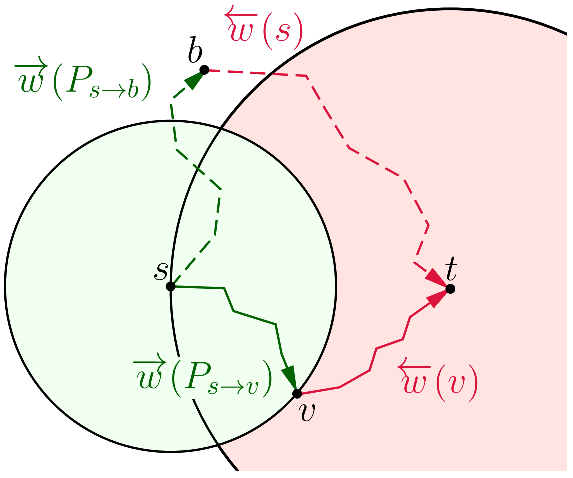
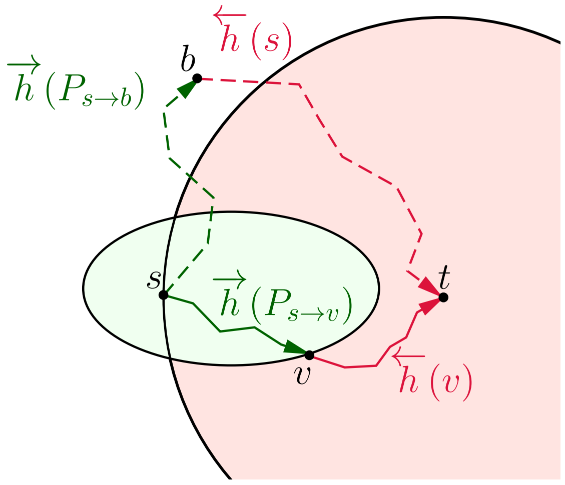

$\overleftarrow{w}(s)$
b
$\overrightarrow{w}(P_{s\to b})$
s
t
$\overrightarrow{w}(P_{s\to v})$
$\overleftarrow{w}(v)$
v

Fig. 2.6: Components of the projected cost of the look-ahead technique

2.3.6 Components of the projected cost of the A* heuristic

On Figure 2.7 we can see how the projected path cost for the energy is calculated. It is identical to the look-ahead technique, but this time the projected cost $\overrightarrow{h}(P_{s\to v}) + \overleftarrow{h}(v)$ is used to ordered the candidate paths. This in the sense of the A* algorithm, where the $\overleftarrow{h}(v)$ is used as the heuristic function. In the case of the look-ahead technique, the goal was to prune the unfeasible paths as soon as possible. Although, the A* heuristic does not speedup the search by pruning any paths, but it speeds it up by better, more informed ordering of the candidate paths in the priority queue.

On Figure 2.7 there is also displayed an alternative node b, which is outside of the precomputed search space, the red disc. Therefore, the exact distance to the target $\overleftarrow{h}(b)$ is not known, so for evaluation of the node b we use the shorted distance to the origin $\overleftarrow{h}(s)$ as the estimated distance.



$\overleftarrow{h}(s)$
b
$\overrightarrow{h}(P_{s\to b})$
s
$\overrightarrow{h}(P_{s\to v})$
t
v
$\overleftarrow{h}(v)$

Fig. 2.7: Components of the projected cost of the A* heuristic

2.3.7 Path dominance

Another speedup technique is called *path dominance*. To explain this technique we describe an example scenario. We have reached some intermediate node v by path P_A. The time we have used is 15 minutes. Without this approach, the path would be normally added to the priority queue for further expansion. However, if there is some another path P_B, which also leads to the node v and has been found earlier, has used only 12 minutes and the same amount of energy as path P_A, then we do not need to store the path P_A, because we already know that it will lead to same or worse quality paths as the P_B. By quality of path we understand the amount of used resources needed to follow this path. Such path is called *dominated path* and it is prunned. The test of path dominance is given by Algorithm 2.

Algorithm 2 Path dominance test

Input: Directed weighted graph $G = (V, E)$, set $\overrightarrow{w}$ representing time spent on path P from source vertex s to intermediate vertex v, set of non-dominated paths S

Output: Returns 1 if the path $P^*_{s\to v}$ cannot be a subpath of a shortest path, otherwise 0

```
1: procedure IsDominated($S$, $\vec{h}$, $\vec{w}$, $P^*_{s\to v}$)
2:     for all paths $P^i_{s\to v}$ from set $S$ do
3:         if $\overrightarrow{h}(P^i_{s\to v}) \leq \overrightarrow{h}(P^*_{s\to v})$ and $\overrightarrow{w}(P^i_{s\to v}) \leq \overrightarrow{w}(P^*_{s\to v})$ then
4:             return 1;
5:         end if
6:     end for
7:     $S \leftarrow S \cup P^*_{s\to v}$;
8:     return 0;
9: end procedure
```

2.3.8 Simple path search only

We can further limit the search space by considering only the simple paths. A simple path is such path that does not contain one node twice or in other words, the path does not return to the node that it is already going through. This is accomplished without any further effort because the path dominance test will always prune all paths that return to the same nodes. Simply said, if we go from first node to second and then back, the time and energy used to return from the second will always increase the resources used to reach the first node.

Algorithm 3 Energy optimal path search with time constraint

Input: Directed weighted graph $G = (V, E)$, source vertex s, destination vertex t, constraint multiplier β, time cost function c_t, energy cost function c_e

Output: A shortest path between s to t obeying the time cost constraint

```
1: procedure FindPath($G$, $s$, $t$, $\beta$, $c_t$, $c_e$)
2:     $\overleftarrow{w} \leftarrow$ Dijkstra($G$, $s$, $t$, $c_t$); noting that $\overleftarrow{w}(a) = \overleftarrow{w}(P_{a \to t})$
3:     $\overleftarrow{h} \leftarrow$ Dijkstra($G$, $s$, $t$, $c_e$); noting that $\overleftarrow{h}(a) = \overleftarrow{h}(P_{a \to t})$
4:     $\overrightarrow{w}(P_{s \to s}) \leftarrow 0$;
5:     $\overrightarrow{h}(P_{s \to s}) \leftarrow 0$;
6:     $S \leftarrow \emptyset$;
7:     $Q \leftarrow \{P_{s \to s}\}$;
8:     while $Q \neq \emptyset$ do
9:         $P_{s \to u} \leftarrow$ choose the path with minimal $\overrightarrow{h}(P_{s \to u}) + \overleftarrow{h}(u)$ from $Q$;
10:        $Q \leftarrow Q \setminus P_{s \to u}$;
11:        if $u = t$ then
12:            return $P_{s \to u}$;
13:        end if
14:        for all outgoing edges $(u, v)$ of node $u$ do
15:            if $v \notin \overleftarrow{w}$ then
16:                $\overleftarrow{w}(v) \leftarrow \overleftarrow{w}(s)$;
17:            end if
18:            if $\overrightarrow{w}(P_{s \to u}) + c_t(u, v) + \overleftarrow{w}(v) > \beta \cdot \overleftarrow{w}(s)$ then
19:                continue; prune this path as it is not feasible
20:            end if
21:            $P_{s \to v} \leftarrow$ extend $P_{s \to u}$ with edge $(u, v)$;
22:            $\overrightarrow{w}(P_{s \to v}) \leftarrow \overrightarrow{w}(P_{s \to u}) + c_t(u, v)$
23:            $\overrightarrow{h}(P_{s \to v}) \leftarrow \overrightarrow{h}(P_{s \to u}) + c_e(u, v)$;
24:            if IsDominated($S$, $\vec{h}$, $\vec{w}$, $P_{s \to v}$) then
25:                continue; prune this path as it is dominated
26:            end if
27:            Path $P_{s \to v}$ is considered feasible so it is added to the queue.
28:            if $v \notin \overleftarrow{h}$ then
29:                $\overleftarrow{h}(v) \leftarrow \overleftarrow{h}(s)$;
30:            end if
31:            $Q \leftarrow Q \cup P_{s \to v}$;
32:        end for
33:    end while
34: end procedure
```

3 IMPLEMENTATION

In this section we describe the data-sources that were used the implementation of the prototype application. The application is written in Java programming language and runs on Android operating system. All the development and testing was conducted on the Samsung Galaxy Tab tablet.

3.1 Sytadin

Sytadin is an on-line traffic information system for Ile-de-France. It is possible to have access to real-time traffic information in form of XML feed at the address `http://www.sytadin.fr/diffusion`. The information is updated every two minutes.

3.1.1 Distributed Information

Sytadin provides different files to describe the traffic information:

- Traffic net geometry files – these files are in MapInfo format [13] and describe the geometry of the traffic net. It is particularly the geographical information about net points, arcs, axes, segments and poles. This information is static and does not change.
- Dynamic traffic information – information about speed and traffic states, traffic tendencies, incidents and dysfunctions. This information is updated every few minutes. The files are in XML format with XML Schema XSD file available.
- Route information – route mark-ups and closings. Updated once per day. The file is in XML format with XSD Schema file available.

In listing 3.1 is the example of the dynamic traffic information XML file. It gives the information about two arcs which are specified by their IDs. In the ArcDynamique tags are the tags describing the actual arc: EtatTrafic (the state of traffic flow - fluid, jammed, unknown), TPBride (estimated travel time), TPReference (reference travel time), EcartTypeTPReference (standard deviation of the travel time), VitesseInstantanneeBridee (estimated actual speed), NiveauService (service state) and IndiceConfiance (confidence index).

Listing 3.1: Example of Sytadin's dynamic traffic information file with two arcs.

```
<DonneesDynamiquesArcs VersionConfiguration="14"
          DateDiffusion="2012-12-02T16:48:49">
  <ArcDynamique ID_ARC="13010493">
    <EtatTrafic>Fluide</EtatTrafic>
    <TPBride>−1</TPBride>
    <TPReference>59</TPReference>
    <EcartTypeTPReference>0.00</EcartTypeTPReference>
    <VitesseInstantanneeBridee>60</VitesseInstantanneeBridee>
    <NiveauService>1.00</NiveauService>
    <IndiceConfiance>1.00</IndiceConfiance>
  </ArcDynamique>
  <ArcDynamique ID_ARC="13009571">
    <EtatTrafic>Fluide</EtatTrafic>
    <TPBride>−1</TPBride>
    <TPReference>52</TPReference>
    <EcartTypeTPReference>0.00</EcartTypeTPReference>
    <VitesseInstantanneeBridee>0</VitesseInstantanneeBridee>
    <NiveauService>1.00</NiveauService>
    <IndiceConfiance>0.00</IndiceConfiance>
  </ArcDynamique>
</DonneesDynamiquesArcs>
```

3.1.2 Traffic Net Geometry Coordinates Transformation and Export

As already mentioned, the traffic net geometry files are in MapInfo format. It is possible to visualize (see Figure 3.1) these files for example with free program ShapeView [14].

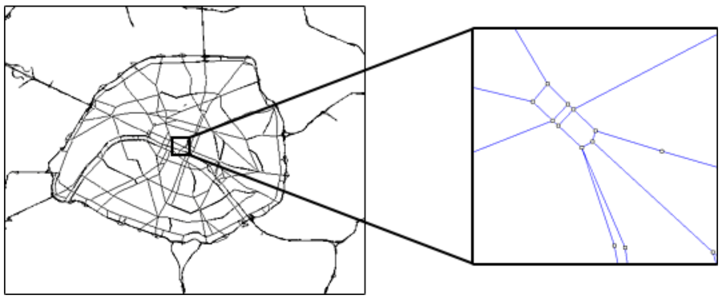


Fig. 3.1: Visualization of the Sytadin's traffic net geometry of Paris using ShapeView program and a zoom-in of connected arcs

Sytadin does not provide geographical data in World Geodetic System (WSG84) used by GPS but in Lambert Conformal Conic projection. Therefore, it is necessary to transform these coordinates. For that purpose, a simple custom MIF2Geo application (shown in Figure 3.2) is used, which reads the MapInfo file, transforms the coordinates from Lambert projection to WSG84 and saves the data in CSV file for further processing.

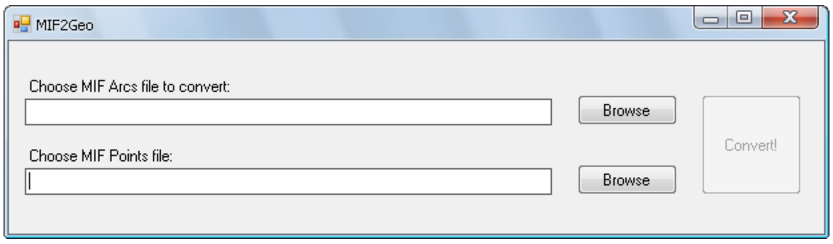


Fig. 3.2: MIF2Geo application user interface

The CSV file is a text file containing lines of values separated by commas and therefore creating columns. In the first line is a text description of each column. The columns are the following: id (Sytadin's ID of the current arc), XMin (minimal longitude of the rectangle enveloping the current arc in decimal degrees), XMax (maximal longitude of the rectangle), YMin (minimal latitude of the rectangle), YMax (maximal latitude of the rectangle), OriginType (flag describing the start of the arc - see below), alfaRads (the angle between a rectangle's down-side and the diagonal in radians) and RefSpeed (reference arc's speed in km/h). A graphic explanation of these parameters is in Figure 3.3.

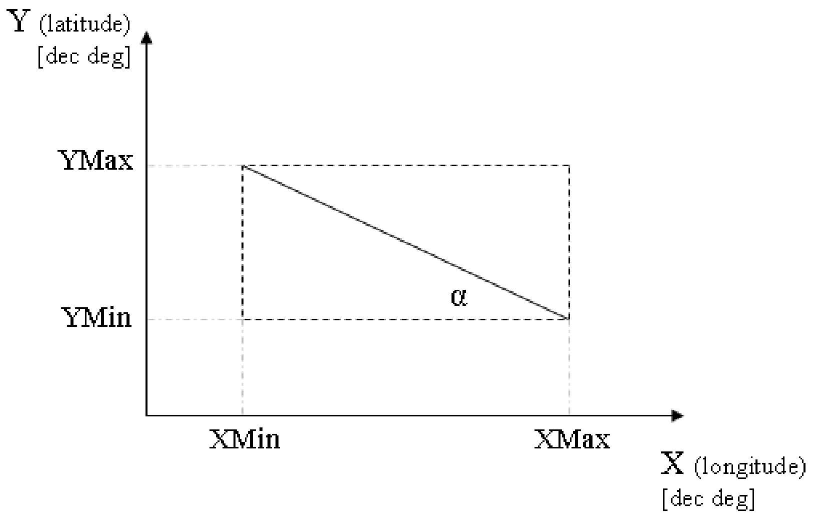


Fig. 3.3: An arc (solid line) with enveloping rectangle (dashed line) and its parameters

In some cases, there is only one arc defining the geographical position of two roads. More precisely, it concerns mainly the roads with two directions without some kind of boundary in the middle. Therefore, there are two arcs with different IDs and different traffic information in the database, but they have the same geographical coordinates so it is necessary to know also the origin point of the road to determine the traffic direction. This information is needed during the fusion of Sytadin's traffic information and OSM map data. In the arcs' MapInfo file (.mid file) there is a field with origin point ID. Then, we have to find this point in another MapInfo file with points (the next file input in MIF2Geo application) and get it's coordinates. Once having the origin point's position, we can bind it with arc's enveloping rectangle coordinates and find, in which corner of the rectangle is the origin. Then, there are four possibilities shown in Figure 3.4.

Additionally, the angle of the diagonal is calculated to speed up and help some other calculations. The reference speed is one of the MapInfo fields and is also exported to use it afterwards in routing database.

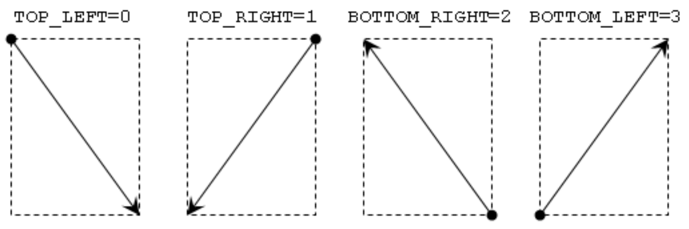


Fig. 3.4: Different types of starting points of an arc and their respecting flags used in CSV file

3.2 Open Street Map Export

Open Street Map propose the geographical data in OSM XML file format. The main advantages are mostly human readable and clear structure, machine independence and good compression ratio. However, the files can be huge and parsing may take a long time. The OSM XML file is composed of data primitives describing the map. These can be nodes (a single geospatial point using a latitude and longitude), ways (an ordered list of 2 to 2000 nodes, ways can be used to represent linear features (vectors) or polygons) relations (used to model logical and usually local or geographic relationships between objects) or tags (two free format textual fields, a "key" and a "value", describing map features like max speed or road type). Complete description of the XML file format is available in the wiki documentation [15]. In order to maximize the performance on Android devices, it is necessary to think about the data format for routing, because the full map in native XML format is inefficient and too bulky. For example the file for Ile-de-France has approximately 2.7 GB. Therefore, only necessary information for routing is exported.

The map export can be divided into three main steps: preparing the geographical for the routing database, preparing the traffic net geometry information and finally exporting the map to the routing database. This process is displayed by flowchart in Figure 3.5.

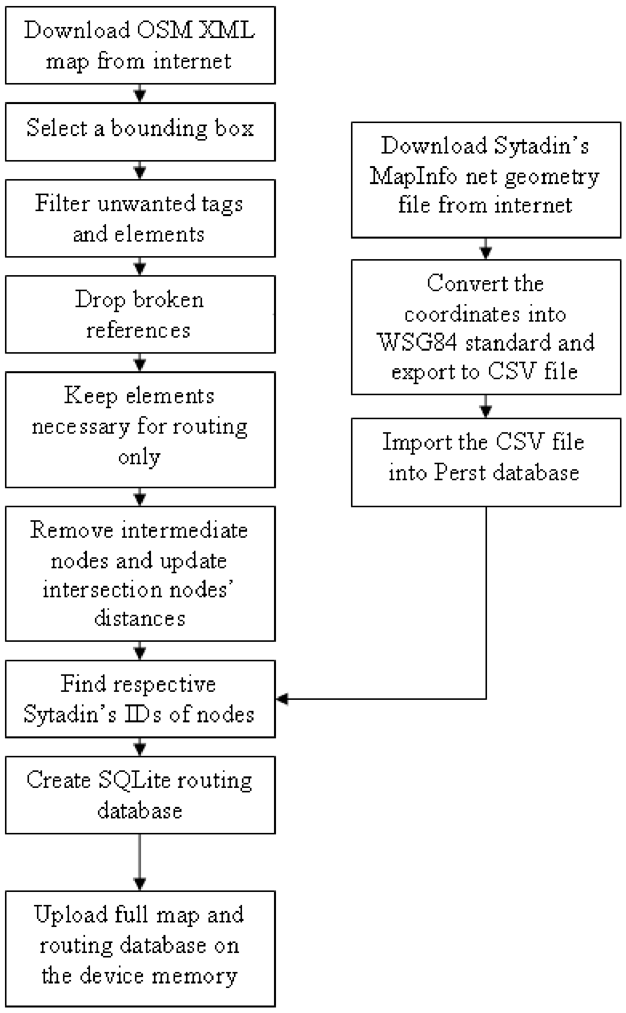

Download OSM XML map from internet
Select a bounding box
Filter unwanted tags and elements
Drop broken references
Keep elements necessary for routing only
Remove intermediate nodes and update intersection nodes' distances
Find respective Sytadin's IDs of nodes
Create SQLite routing database
Upload full map and routing database on the device memory
Download Sytadin's MapInfo net geometry file from internet
Convert the coordinates into WSG84 standard and export to CSV file
Import the CSV file into Perst database

Fig. 3.5: Flowchart of tasks during the export of maps

3.2.1 Preparing the geographical data

Firstly, it is necessary to download the map data. There are more possibilities. Either you can download the data of the whole planet but this is not recommended, because the file Planet.osm in the XML format has over 250 GB in uncompressed variant, 16 GB compressed. Then, there is a possibility to export data directly from the web based map on the www.openstreetmap.org web page by selecting a region on the map. However, this method is useful only for small areas and testing purposes because the server can export only a limited number of nodes. The best way seems to be to download already prepared maps of continents, countries or regions. Good source of this map is for example on the CloudMade project web site `dowloads.cloudmade.com`. Actually, we have traffic information only for the area of Paris and suburbs so only the map for Ile-de-France is used.

The main tool for processing and converting OSM files is Osmosis tool. Download links and full documentation can be found at `wiki.openstreetmap.org/wiki/Osmosis`. We use this tool to filter the map in such way that only the relevant information for routing remains. By relevant are considered all roads that are accessible by a motor vehicle. A useful link about how the construct the filter command can be found at "tags for routing": `wiki.openstreetmap.org/wiki/OSM_tags_for_routing`. Osmosis command that is used is in Listing 3.2.

Listing 3.2: Filter tags and keep only those desired for routing

```
osmosis.bat --read-pbf file="s2-ile-de-france.pbf"
        --way-key-value keyValueList = "highway.residential, highway.unclassified,
            highway.motorway, highway.motorway_link, highway.trunk, highway.trunk_link,
             highway.primary, highway.primary_link, highway.secondary, highway.
            secondary_link, highway.tertiary, highway.tertiary_link, junction.roundabout,
            highway.mini_roundabout"
        --tf reject-ways access=private
        --tf reject-relations
        --used-node
        --write-pbf file="s3-ile-de-france.pbf"
```

3.2.2 Preparing the traffic net geometry information

This part consists in downloading the MapInfo files from Sytadin's server and in converting it into WSG84 standard coordinates. This is done by application MIF2Geo as mentioned before. The application generates a CSV file which is going to be used during the creation of the routing database.

3.2.3 Routing database export

The export of the routing database consists in more tasks. These are:

- more filtering of unnecessary elements,
- removing intermediate nodes,
- recalculating distances between intersection nodes,
- importing CSV traffic geometry file into a database for fast search,
- associating traffic information with intersection nodes,
- export to SQLite database (map.db).

In Figure 3.6, you can see the full path of a short driving cycle from node A to node B. The orange dots represent intersection nodes. These nodes are necessary for routing to calculate the shortest path (the path with the minimal cost corresponding to a given criteria). There are also blue dots which represent intermediate nodes, which are not necessary for routing, because there is no crossing or turn. However, they are necessary to know the exact shape of the road.

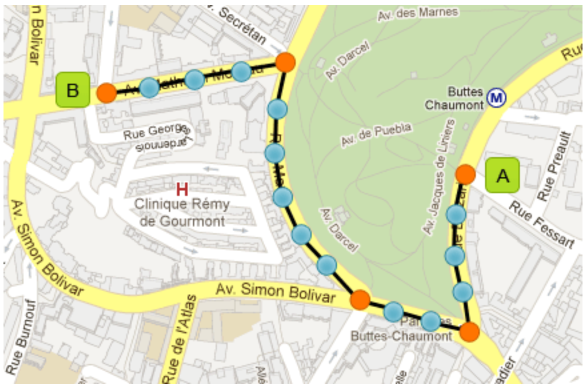


Fig. 3.6: Full path with intersection (orange) and intermediate nodes (blue)

In Figure 3.6 is the same path as in Figure 3.7, but without the intermediate nodes. In the map file there is a huge number of intermediate nodes which are not necessary for routing and therefore increase highly the computational time of the shortest path. This is the reason the intermediate points have to be removed from the routing database. More precisely, they are not removed but marked so they are not considered during the routing. If we compare the two images we can see that the number of nodes decreased from 18 to 5. However, it is important to mention that the distances between two intersection nodes are not necessarily the same if the road is curved. For this reason, we must recalculate the distances between the nodes for the reduced map. The intermediate nodes are used to reconstruct the exact shape of the path which we need for graphic visualization and for sampling altitudes for the power profile calculation.

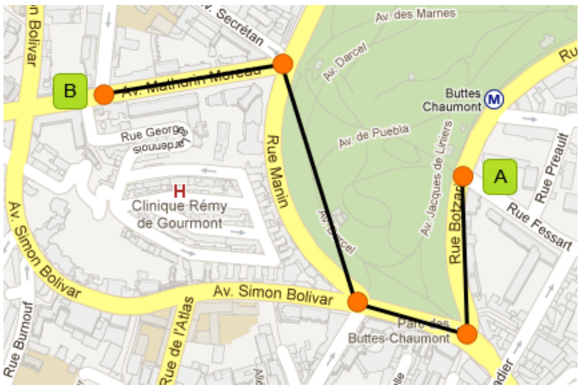


Fig. 3.7: Reduced path with only intersection nodes needed for routing

After exporting the intersection routing nodes, we need to associate them with their respective traffic information IDs. The idea is that we try to overlap the routing map and the Sytadin's traffic net geometry as shown in Figure 3.8. For now, we have exported the Sytadin's arcs in the CSV file which contains its IDs, the enveloping rectangle, the angle of the diagonal and the reference speed of each arc. Searching in a text file is very time consuming. For that reason, the data is inserted into a database which supports so-called R-trees. These trees are optimized for space indexing and for geo-location applications. One very good open-source object oriented database for Java is Perst [18].

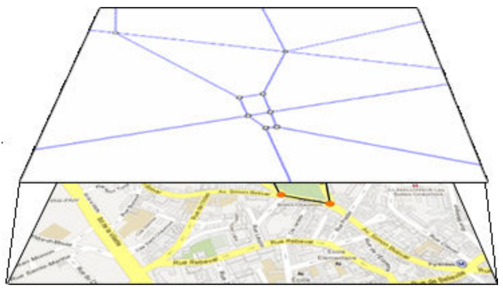


Fig. 3.8: Routing database and Sytadin's geometry fusion illustration

After importing the Sytadin's arcs into Perst database, we can find a record quickly. To associate a node with its Sytadin's ID, we have to find the appropriate arc in the database. We know the geo-coordinates of the node and can use it to search in the database. The result is a list of all rectangles, that envelope the node. In most cases, we will find only one arc that corresponds to the node. However in some cases, more enveloping rectangles can be overlapped. In Figure 3.9 we can see two arcs found for one node (because the enveloping rectangles are overlapped). In that case, it is necessary to evaluate the distances between the node and the actual arc. In Figure 3.10 is shown the distance between a node and an arc. Because the arc is the rectangle's diagonal, we can calculate the distances of each arc and compare them. This way we can decide that the node in Figure 3.9 should be associated with the arc BC and not the arc AB.

The geo-positions of Sytadin's arcs are not strictly the same as positions of the OSM ways. It can be caused simply by the fact, that the real road is very large and the Sytadin's arcs copy the left border of the road, however, OSM way copy the right border. This can create a significant error and we are not sure to find the appropriate arc of the given node. This is why we do not search an exact point in the database but we search the point with some tolerance. It means that we do not search an exact point but a rectangle that overlaps the enveloping rectangle of the arc. In Figure 3.11 you can see an example why it is necessary to search with some tolerance. For now we have used the tolerance of 10^5deg of latitude and longitude (The exact distance is different for latitude and longitude and depends on the position on the globe. It is usually a few meters).

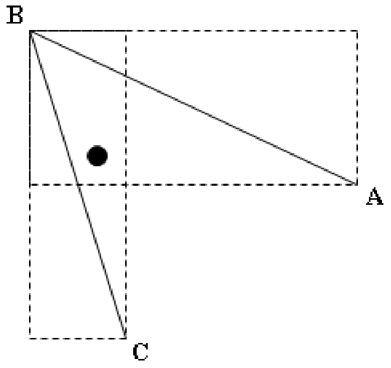


Fig. 3.9: Overlapped enveloping rectangles of two arcs while searching for the node's arc

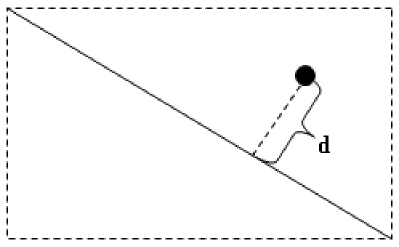


Fig. 3.10: Distance d between a node and an arc

One arc can be associated with two IDs to inform about the traffic state in both directions. While searching in the database, we will find two records with the same distance error between the node and the arc. Than we must decide which ID we will attribute to the node. Because the nodes are grouped in ways we can get the next node of the way and determine the direction of this road or lane.

3.3 Shuttle Radar Topography Mission Elevations

To add the third dimension to the OSM exported map, we used the data provided by the NASA Shuttle Radar Topographic entitled Topography Mission (SRTM). This mission was launched in February 2000 and the objective was to obtain RADAR data of most of the Earth's land surface to produce high resolution topographic maps.

Approximately 80 percent of the land surface was acquired. The data has been released at two horizontal resolutions: 3 arc-seconds (90 m) globally and 1 arc-second (30m) for the United States. Therefore, for Île-de-France's region, we can assume that information is furnished with 3 arcs-seconds or 90m. The initial version of the data was released globally in one degree tiles. These data were not processed to eliminate data voids and there were errors with flat water surfaces and coastlines.

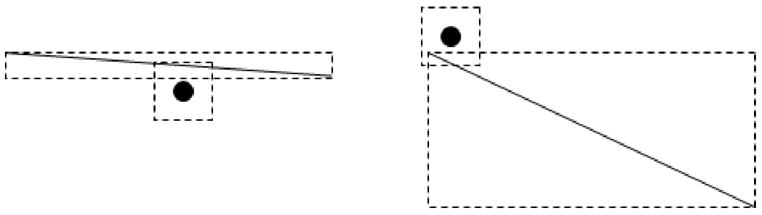


Fig. 3.11: Searching for overlapping rectangles within a tolerance

Programs such as SRTMFill by 3DNature [17] can fill data voids in SRTM height files.

3.3.1 HGT File Encoding

Data are in a height format file that uses a file extension ".HGT" and are compressed using ZIP format. The file name is based on the coordinates of the lower left corner of the tile. For example, tile N51E000.hgt.zip would cover an area including Greenwich, England at 51:29 north and 0:00 east. This is referred to as the HGT file format in the processing section below.

The DEM is provided as 16-bit signed integer data in a simple binary raster. There are no header or trailer bytes embedded in the file. The data are stored in row major order (all the data for row 1, followed by all the data for row 2, etc.). All elevations are in meters referenced to the WGS84/EGM96 geoid as documented at `http://www.NGA.mil/GandG/wgsegm/`. Byte order is ”big-endian” standard with the most significant byte first and because of they are signed integers elevations can range from -32767 to 32767 meters, covering the range of elevation that can be found on the Earth.

These data also contain occasional voids from a number of causes such as shadowing, phase unwrapping anomalies, or other radar-specific causes. Voids are flagged with the value -32768.

3.3.2 Obtaining and preparing the data

To ensure that all the elevations cover the whole Île-de-France's region, we need to have information approximately for the area between the latitude 48° North and 50° North, and longitude 1° East and 4° East. For this purpose, necessary .hgt files were downloaded [16]: N48E001.hgt, N48E002.hgt, N48E003.hgt, N49E001.hgt, N49E002.hgt and N49E003.hgt.

As stated above, the SRTM data has data voids, or missing data, at various locations within the tiles. It is recommended to process these data to fill in the gaps. SRTMFill was used to interpolate values to produce a "complete" dataset.

This program works as interpolation routine and is most effective when the gaps are small and advise is given to take care when the data voids exceed 50 cells in diameter. Each of the files is processed individually. Once all hgt files were filled, we combine all those 6 files together into one. To do the combination, we need to be careful about the indexing of the extracted file hgt file. As we know, SRTM data are sampled at three arc-seconds and contain 1201 rows and 1201 columns. The rows at the north and south edges as well as the columns at the east and west edges of each cell overlap and are identical to the edge rows and columns in the adjacent cell. The final extracted hgt file has maximum indexes [2400,3600].

4 TESTS

In the following sections we will give a demonstrative test of the implemented algorithms described in the previous chapters. These test show some of the observation that we came across while developing the final algorithm.

4.1 Search space of the basic algorithms

The first test is to demonstrate the search space of the basic routing algorithms. The android application was modified for this test in such way, that the algorithm is marking the scanned edges directly on the displayed map. This allows us visually confirm how much of the map has the algorithm to visit before it finds the solution. The basic rule is that, the smaller the search space is the faster the query is. The following tests are conducted with shortest distance criteria, thus we obtain typical disc shaped search spaces.

4.1.1 Dijkstra's algorithm

Classic Dijkstra algorithm searches in circular fashion. We can see its typical search space on Figure 4.1. It starts to grow a disc from the origin A towards the destination B. Once it meets the node B, the algorithm stops. On this figure we cannot see the full extent, because the picture would be too large.

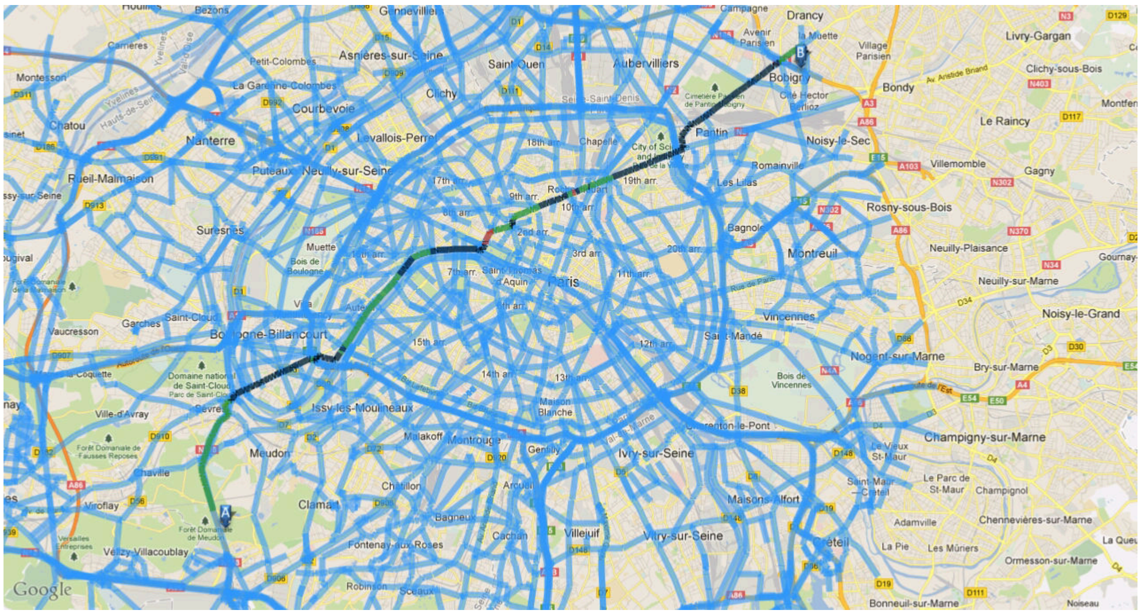


Fig. 4.1: Schematic representation of the search space of the Dijkstra's algorithm

4.1.2 A* algorithm

A* algorithm that is modification of the Dijkstra's in such way that a heuristic function is introduced. This heuristic guides the algorithm towards the destination node so the search extent is reduced and so is the routing time. We can see on Figure 4.2 that the search space is greatly reduced compared to the plain Dijkstra's algorithm shown on Figure 4.1. This works very well, because we can simply use the air line distance to the destination as the admissible heuristic function. If we would have searched for the fastest path and used equivalently simple heuristic function then the reduction would not be so significant.

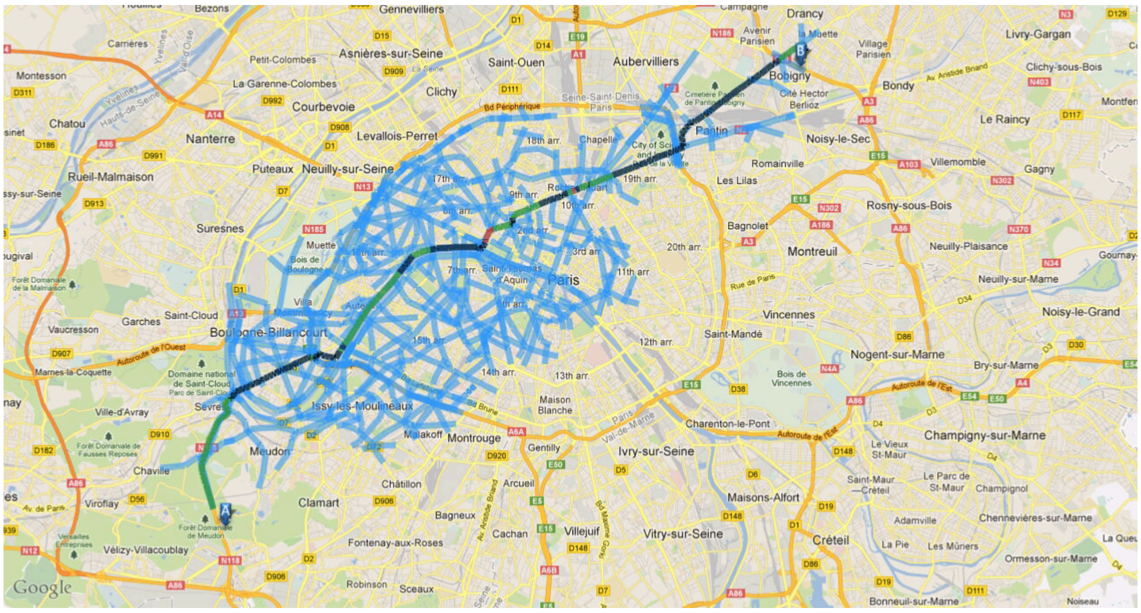


Fig. 4.2: Schematic representation of the search space of the A* algorithm

4.1.3 Bidirectional Dijkstra's algorithm

Bidirectional Dijkstra algorithm is such modification that uses two concurrently running classic Dijkstra's searches in opposite directions. When these two meet at the same node, the search is stopped. The optimal stopping condition is more complicated and so our solution can in some specific situations give a suboptimal result. We can see this search space in Figure 4.3. The two searches create two disks that grow from the origin and destination. The forward search is marked with blue color and the backward with violet color. Additionally, what we should notice is that all the shown paths on Figures 4.1, 4.2 and 4.3 are the same path, which is correct.

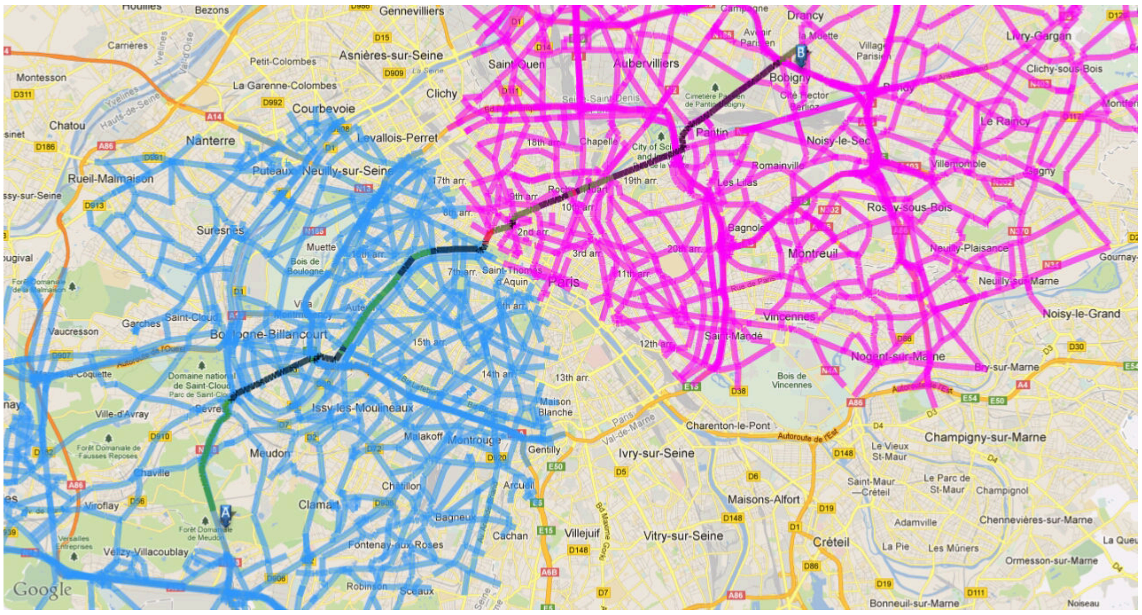


Fig. 4.3: Schematic representation of the search space of the bidirectional algorithm

4.2 Three metrics

In this test we try to find paths between two points for thee different metrics. We can see the resulting paths on Figure 4.4, the shortest path is marked with red color, the fastest path is marked with blue color and the energy optimal route is marked by green color. The origin point corresponds to the red tag and the destination point corresponds to the violet one. The coefficient of energy recuperation α used was 0.85.

We can clearly see that the fastest path goes around the center taking the motorway and that the shortest path goes as directly as possible. From this test and many other, we have deduced that the energy optimal path is usually rather direct, nearly as the shortest path, and at the same time it takes the slowest road segments as possible. If we examine the energy cost function given in the energy modeling section, we can confirm that this behavior is correct, because the energy cost is increasing with greater speed and distance. In table 4.2 we can see the roughly estimated numerical values for these paths.

Metric	Color	Path length	Duration	Arriving state of charge
Shortest	Red	13.8 km	21 min	94.9 %
Fastest	Blue	16.6 km	15 min	92.2 %
Energy optimal	Green	14.3 km	23 min	95.4 %

Tab. 4.1: Numerical values for the three metrics test

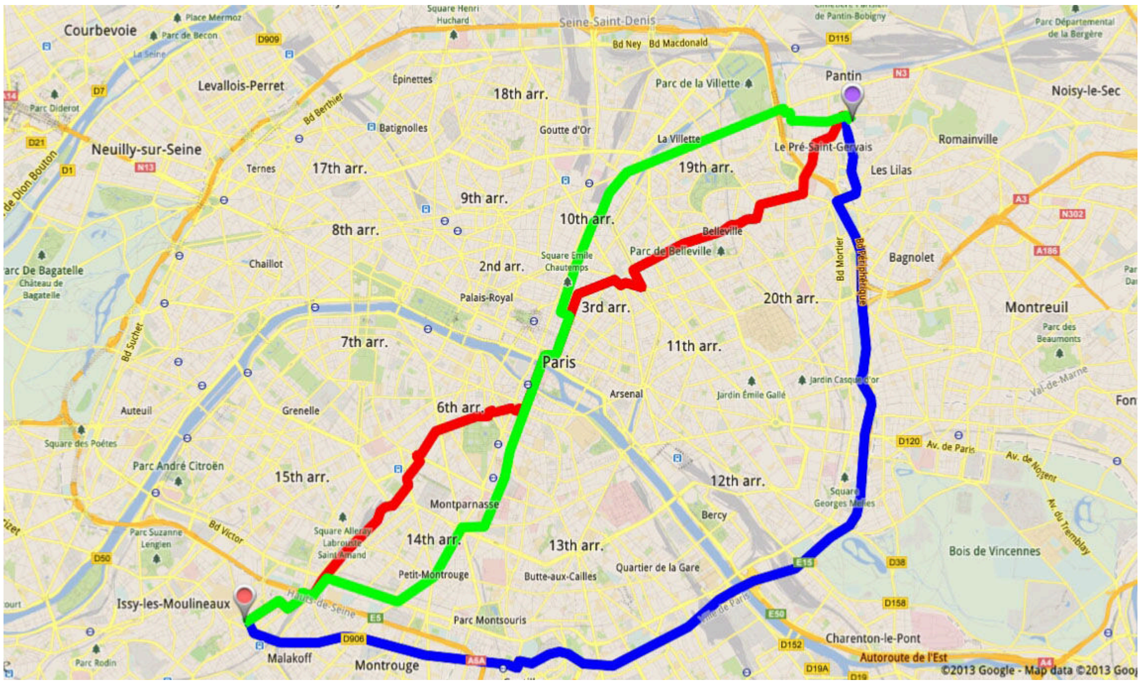


Fig. 4.4: Paths found for the three different metrics

4.3 The dependence of elevation profile on recuperation efficiency

In this test we calculate several routes with different values of parameter α. As explained earlier, this parameter defines the recuperation efficiency of the vehicle. The natural assumption is, that with the lowering α the algorithm will prefer paths with lower elevation oscillations and to find such paths it will sacrifice the length. Therefore, it will find longer paths with smaller elevation gain. This elevation gain is a metric that we calculate so we are able to compare paths based on the elevation oscillations. The higher is the gain the higher are the oscillations. The resulting paths the are displayed in Figure 4.5 and the elevation profiles of these paths are displayed in Figure 4.6. Numerical values are given in Table 4.2. Comparing the numerical values, we can see that they confirm our assumption. We can also see that the black path is identical to the blue one, this is cause by the limited variety of possible paths.

4.4 Routes for different values of parameter β

In Figure 4.7 we can see four calculated paths for different values of parameter β. The route characteristics in term of length, duration and the state of charge at the destination point are given in table 4.3. For this test the $\alpha = 0.8$ for all paths. The

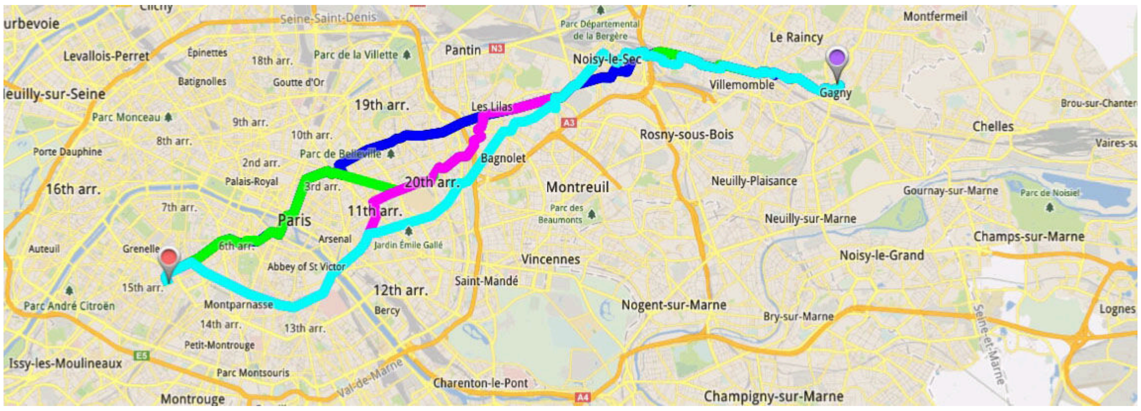


Fig. 4.5: Found paths for different recuperation efficiencies

α	**Color**	**Path length**	**Duration**	**Arriving state of charge**
1.0	Black	20.8 km	52 min	93.8 %
0.9	Blue	20.8 km	52 min	93.5 %
0.6	Green	21.7 km	60 min	92.8 %
0.3	Magenta	22.5 km	46min	91.9 %
0.1	Cyan	22.8 km	41 min	91.3 %

Tab. 4.2: The dependence of elevation profile on recuperation efficiency

β parameter is set different for each of those paths, so we can observe the influence it has on the resulting path. For $\beta = 1$ the problem is the same as we would look for the fastest path, because it is the only that can meet the constraint. On the other hand, if $\beta = \infty$ then the path is the energy optimal path regardless the constraint. Three other paths with the time constraint increased by $10\%, 15\%$ and 30% are given. We can observe that with increasing β the duration of the path and the arriving state of charge increases. That is, the more time we give the algorithm to spend on the path the slower path he can choose and this is in concordance to what we have proved in the previous sections, that the energy metric prefers the slower paths.

β	**Color**	**Path length**	**Duration**	**Arriving state of charge**
1.00	Red	22.2 km	18 min	88.7 %
1.10	Green	20.1 km	19 min	91.1 %
1.15	Cyan	20.2 km	20 min	91.4 %
1.30	Magenta	16.3 km	23 min	93.2 %
∞	Blue	15.0 km	27 min	94.9 %

Tab. 4.3: Route parameters for different values of parameter β

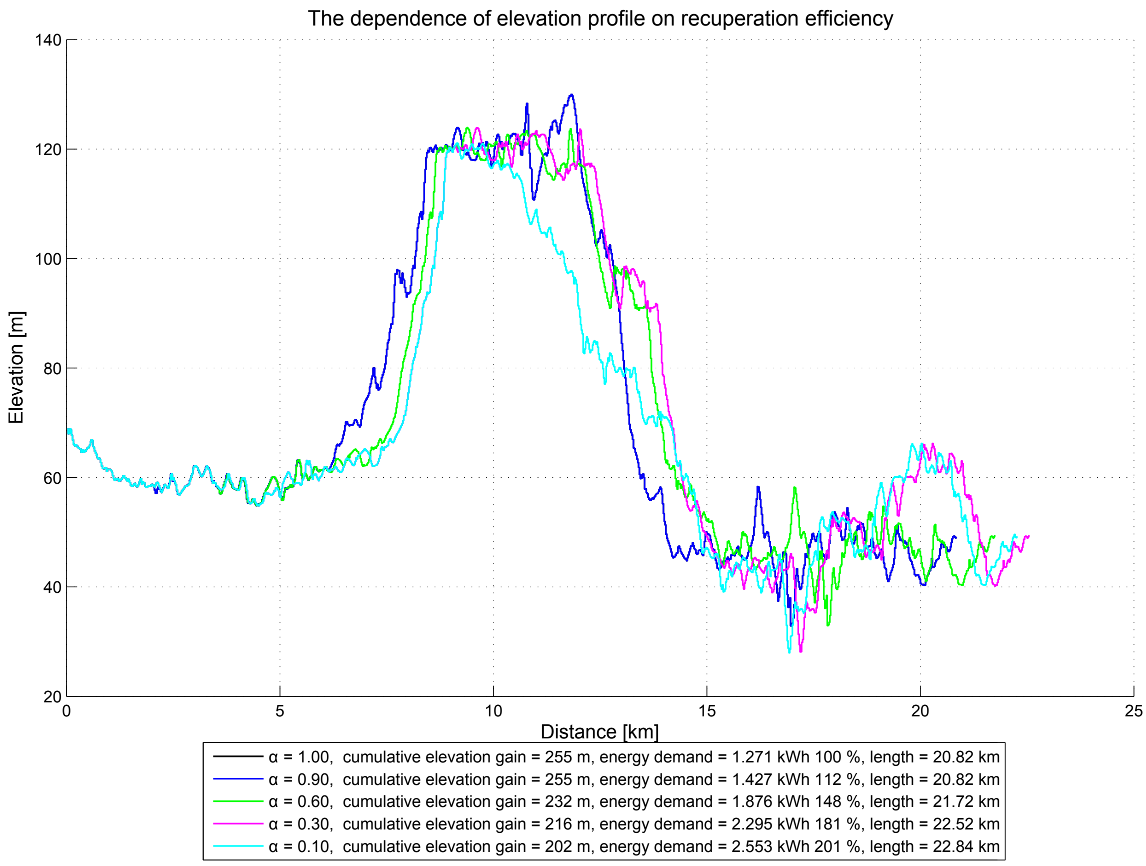


Fig. 4.6: The dependence of elevation profile on recuperation efficiency

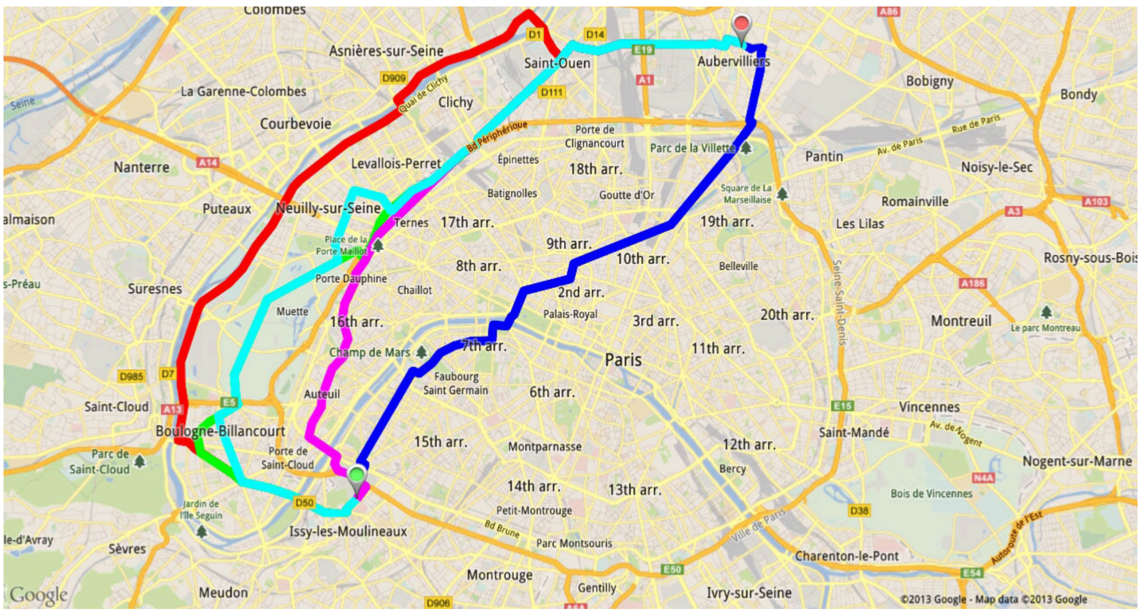


Fig. 4.7: Found paths for different values of parameter β

5 CONCLUSION

In this work we have presented algorithms that are capable of calculating paths for electric vehicles. These algorithm were successfully implemented on the Android tablet device using the Java programming language. Those algorithms are mainly based on the well known Dijkstra's algorithm and its modifications such as the A* and Bidirectional version.

In the first part, this work gives a model of the electric vehicle and a formula for calculating energy cost that is required to traverse a road segment. The process of obtaining this formula is given in several steps, so the reader can understand how its final form was deduced. In this section, we also show that the ability of recuperation of energy may give the edge weight a negative value. This poses a difficulty for the Dijkstra's algorithm, because this algorithm is broken when used with negative values. Therefore, we describe a solution to compensate this problem. We also describe how to introduce a limitation on the amount of battery the vehicle can use and also we show ho the state of overcharging is penalized.

In the second part, we proposed a new algorithm that is able to compute a path based on multiple metrics. We have developed this algorithm after the test showed us that the energy metric prefers the slowest paths possible. Therefore, this algorithm enables us calculates an energy optimal path, which is acceptable by the driver. This is accomplished by computing the path by combining the time metric and energy metric. This part we consider as the most valuable, because it combines many concepts of algorithms used in another problematics and by good understanding of them we were able to create this new algorithm.

This algorithm is parameterizable by the driver in such way, that an additional information is given to the algorithm. This information is how much time the driver is willing to sacrifice to obtain less energy demanding path. This parameter is denoted as β and its values $\beta \in (1, \infty)$ actually define the objective of the routing problem. If the value of $\beta = 1$ we have a time optimal routing problem, because the driver in this case in not willing to travel longer time than the fastest path would take. On the other hand, if $\beta = \infty$ we have an energy optimal routing problem, because the driver is not concerned about the time at all. A condition that $\beta \geq 1$ has to be met, otherwise we would be querying for a path that is faster than the fastest possible path. This algorithm provides an exact solution, but the base of the algorithm works in exponential time. With additional techniques, that are described in this section, it is made to work in sub-exponential time.

This algorithm can be further generalized that it considers several metrics at once. For example, you might want to know the fastest route for visiting your friend but you only want routes where you do not need to refuel your car, or you may want to know the fastest route subject to the condition that the road toll does not exceed a certain limit.

In the third part, we briefly describe how we pre-process the OSM map and its fusion with the SYTADIN's speed information and how we obtain the elevation data from the SRTM data-source. In the last part, we present several tests that demonstrate dependencies of the found paths on the algorithm parameters. The prototype application is fully functional and can demonstrate the described algorithms.

BIBLIOGRAPHY

[1] Dijkstra, E. W.: *A note on two problems in connexion with graphs*, Numerische Mathematik, 1:269–271, 1959, <http://dx.doi.org/10.1007/BF01386390>.

[2] Hart, P.E.; Nilsson, N.J.; Raphael, B., *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*, Systems Science and Cybernetics, IEEE Transactions on , vol.4, no.2, pp.100,107, July 1968.

[3] Guzzella, L.; Sciaretta, A.: *Vehicle Propulsion Systems, Introduction to Modelling and Optimization, Second Edition*, Springer, 2007.

[4] Ehsani, M.; Gao, Y.; Emadi, A.: *Modern Electric, Hybrid Electric and Fuel-Cell Vehicles. Fundamentals, Theory and Design. Second Edition.* 2010.

[5] A. Hrazdira, B. Rezende, E. Campadelli, A. Cela, R. Hamouche, A. Reama, R. Natowicz, *Optimal Energy Management of an Hybrid Electric Vehicle*, Internal report, Embedded System department, ESIEE Paris, 2012.

[6] Johnson D. B.: *Efficient Algorithms for Shortest Paths in Sparse Networks.* Journal of the ACM 24, January 1977, <http://doi.acm.org/10.1145/321992.321993>

[7] Eisner, J.; Funke, S.; Storand, S.: *Optimal Poute Planning for Electric Vehicles in Large Networks* Proc. of the Twenty-Fifth AAAI Conference on Artificial Intelligence, San Francisco, USA, August 2011.

[8] Sachenbacher, M.; Leucker, M.; Artmeier, A.; Haselmayr, J.: *Efficient Energy-Optimal Routing for Electrical Vehicles* Proc. of the Twenty-Fifth AAAI Conference on Artificial Intelligence, San Francisco, USA, August 2011.

[9] Storandt, S.: *Algorithms for Vehicle Navigation*, PhD Thesis, Faculty of Computer Science, Electrical Engineering and Information Technology, University of Stuttgart, December 2012.

[10] Van Mieghem, P.; Kuipers, F.: *Concepts of Exact QoS Routing Algorithms*, IEEE/ACM Transactions on networking, Vol. 12, No. 5, October 2004.

[11] Liu, G.; Ramakrishnan, K. G.: *A*Prune: An algorithm for finding K shortest paths subject to multiple constraints*, In 20th Annual Joint Conference of the IEEE Computer and Communications Societies, 2001.

[12] SYTADIN: *A real Time Traffic Information of Paris Area*, <`http://www.sytadin.fr/`>.

[13] *The MapInfo Interchange File (MIF) Format Specification*. Available on-line: <`http://www.gissky.com/Download/Download/DataFormat/Mapinfo_Mif.pdf`>.

[14] Free ShapeView -View ESRI shape files and MapInfo interchange files. Available on-line: <`http://avangardo.com/software/free-shapeview.html`>.

[15] *OpenStreetMap*. Available on-line: <`http://wiki.openstreetmap.org/wiki/Main_Page`.

[16] *Shuttle Radar Topography Mission Elevation data files*. Available on-line: <`http://dds.cr.usgs.gov/srtm/version2_1/SRTM3/Eurasia/`>.

[17] *SRTMFill software to patch all null data holes*. Available on-line: <`http://3dnature.com/srtmfill.html`>.

[18] *Perst - An open source, object-oriented embedded database*. Available on-line: <`http://www.mcobject.com/android`>.

LIST OF SYMBOLS, PHYSICAL CONSTANTS AND ABBREVIATIONS

M	Total mass [m]
C_d	Reynolds coefficient [kg/m^3]
A_f	Windward area [m^2]
μ_r	Rolling resistance coefficient [-]
ρ	Air density [kg/m^3]
δ_{eqm}	Equivalent moment inertia [kg·m^2]
MCP	Multiconstrained path
MCOP	Multiconstrained optimal path
SP	Shortest path
$P_{u \to v}$	A path from vertex u to vertex v
$P^{sp}_{u \to v}$	The shortest path from vertex u to vertex v
$c\left(P^{sp}_{u \to v}\right)$	Cost of the shortest path from vertex u to vertex v
SAMCRA	Self-adaptive multiple constraints routing algorithm
c_t	Time cost function
c_e	Energy cost function
s, t	Source vertex, Destination vertex
α	Recuperation efficiency
β	Upper bounds constraint multiplier
L	Upper bounds, constraint (energy, time or other)
Q	Priority queue
SYTADIN	**Sy**noptique du **Tra**fic **d**e l'**Î**le de Fra**n**ce
OSM	Open Street Map
SRTM	Shuttle Radar Topography Mission

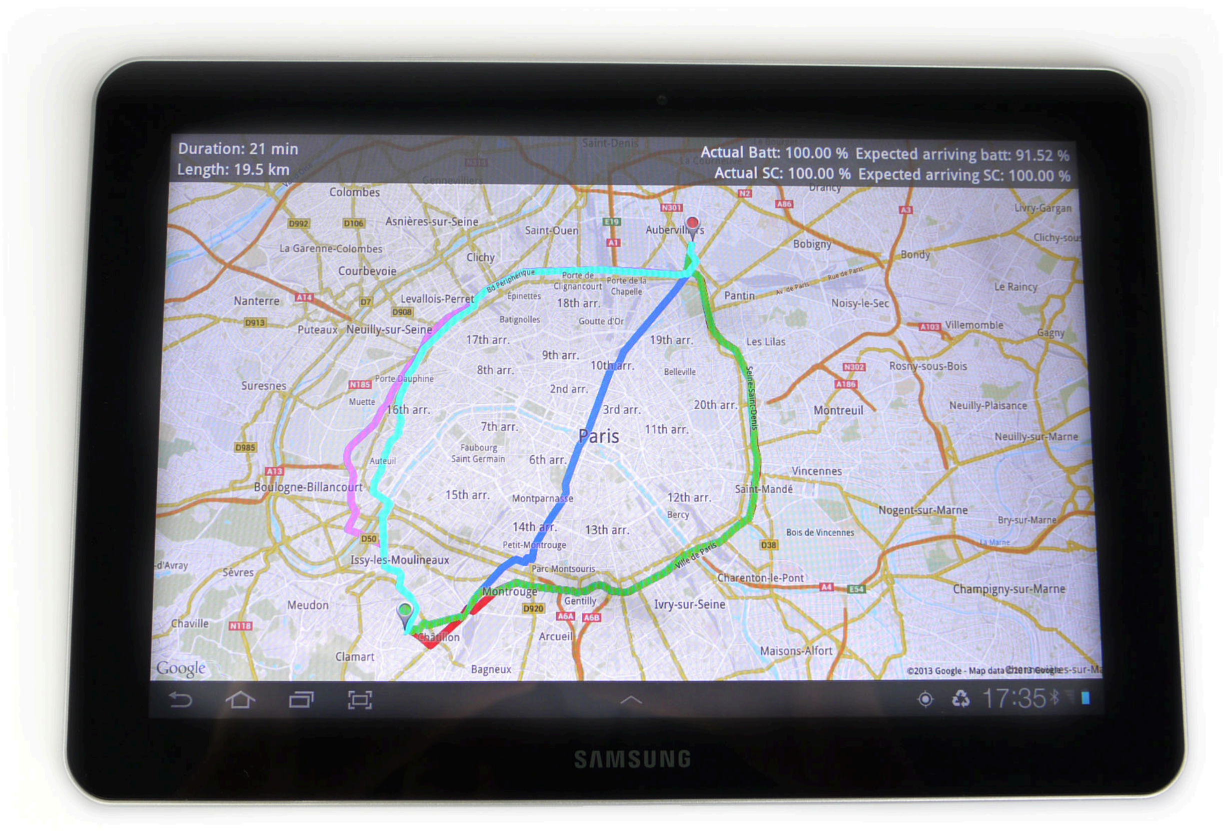
LIST OF APPENDICES

APPENDICES

Appendix 1

A CD is placed into the pocked that is inserted onto the inside of the rear cover. It includes the electronic version of this work and the source code of the android application.

Appendix 2



Duration: 21 min
Length: 19.5 km

Actual Batt: 100.00 % Expected arriving batt: 91.52 %
Actual SC: 100.00 % Expected arriving SC: 100.00 %