UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

COMPUTATION OF ISOCHRONES

DIPLOMOVÁ PRÁCA

2014 Bc. Martin Kolínek

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

COMPUTATION OF ISOCHRONES

DIPLOMOVÁ PRÁCA

Študijný program:	Informatika
Študijný odbor:	2508 Informatika
Školiace pracovisko:	Katedra Informatiky FMFI
Školitel:	RNDr. Richard Ostertág, PhD.

Bratislava 2014 Bc. Martin Kolínek

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Martin Kolínek
Študijný program: informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Computation of isochrones
Vypočítanie hranice oblasti časovej dosiahnuteľnosti

Cieľ: Ciele tejto práce sú:
• Napísať prehľad súčasného stavu riešení pre generovanie hranice oblasti časovej dosiahnuteľnosti
• Navrhnúť algoritmus pre čo najrýchlejší výpočet hranice pre vybraný východiskový bod a časový limit
• Implementácia a testovanie výkonu navrhnutého algoritmu a porovnanie so súčasnými riešeniami

Vedúci: RNDr. Richard Ostertág, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: doc. RNDr. Daniel Olejár, PhD.

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Dátum zadania: 21.11.2012

Dátum schválenia: 28.11.2012 prof. RNDr. Branislav Rovan, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Martin Kolínek
Study programme:	Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	9.2.1. Computer Science, Informatics
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Computation of isochrones

Aim: Goals of this thesis are:
• Write an overview of current state of solutions for generation of isochrones
• Design an algorithm for fast computation of isochrones for selected starting point and time limit
• Implementation and benchmarking of designed algorithm and comparison with current solutions

Supervisor:	RNDr. Richard Ostertág, PhD.
Department:	FMFI.KI - Department of Computer Science
Head of department:	doc. RNDr. Daniel Olejár, PhD.
Assigned:	21.11.2012

Approved: 28.11.2012 prof. RNDr. Branislav Rovan, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Chcel by som sa poďakovat svojmu vedúcemu, rodine, spolužiakom, priateľom a všetkým ostatným, bez ktorých by táto práca nemohla vzniknúť.

Abstrakt

V tejto práci sa venujeme oblastiam v mape, ktoré sú dosiahnuteľné z daného bodu v danom časovom limite. Tejto téme sa venovalo niekoľko autorov. Majú niekoľko využití v navigácií, plánovaní miest a podobne. Pracujeme s dvoma formami presunov. Jeden z nich je obmedzený na pohyb po cestách. Druhý spôsob umožňuje pohyb kadekoľvek, ale je pomalší.

Venujeme sa spôsobom, ktorými sa dá získať takáto dosiahnuteľná oblasť. V poslednej dobe sa urobil veľký pokrok v oblasti plánovania trás v mapách. Keďže problém hľadania oblasti dosiahnuteľnej v časovom limite je problému plánovania trás príbuzný, snažili sme sa využiť techniky z algoritmov na plánovanie trás na hľadanie dosiahnuteľnej oblasti.

Kľúčové slová: časová dosiahnuteľnosť, cestná sieť

Abstract

Isochrones are parts of a map which can be reached from a given starting point within given time limit. Several authors have studied them. They have som uses in urban planning, or navigation. We study isochrones for two modes of transportation. One of them is constrained to a road network. The other one is slower, but is not constrained.

In this thesis, we deal with the ways isochrones can be computed. There have been recent advances in route planning algorithms. Because finding isochrones is a related problem to route planning, we tried to find ways to speed up isochrone search using the techniques from route planning algorithms.

Keywords: isochrone, road network, spatial network, reachability

Contents

Introduction

Isochrones in maps are regions which can be reached within given time from given starting point. They are useful in urban planning, navigation software, tourism information etc.

There are algorithms for solving the problem of finding specific isochrones in maps. However these algorithms are far from perfect. Problems include:

- High time complexity. This problem manifests itself when searching for larger isochrones.
- Precision of the resulting isochrone.

Because in recent years many advancements were made in solving the problem of finding shortest path between two points in a map, it may be possible to improve existing solutions for the problem of finding isochrones.

In this thesis we will first formally describe an isochrone model which uses two modes of transportation. One constrained to a road network, but faster, and the other one unconstrained, but slower.

Then we will describe an approach to computing isochrones in our model. This approach takes into account both modes of transportation, which makes it more precise than existing solutions.

After that we examine two of the approaches for finding shortest paths, and try to adapt them to finding isochrones. Then we will describe the implementation of the techniques and their adaptation to a real world map. Finally, we test the implementation on real world data, and describe the results.

Chapter 1

Related work

In this chapter we will describe the existing research concerning isochrones. We will look at existing definitions of isochrones, and the approaches to finding them. Then we will describe accomplishments in a related problem of route planning.

1.1 Spatial network

Spatial network can be defined as a weighted graph with each vertex having a position in euclidean space. They have many applications in geo-spatial systems, navigation, urban planning etc. The weight of the edges can represent distance, traversal time etc.

Spatial networks can be classified as continuous and discrete. In continuous spatial networks, the points along edges of the graph can be accessed whereas in discrete spatial networks only the nodes can be accessed [11].

In [11], multimodal spatial networks are defined. Multimodal spatial networks allow for multiple modes of transportation through the network. Each mode of transportation may have different properties and constraints.

For purposes of this thesis we do not need multimodal spatial network. We will deal with single continuous spatial network - representing the road network and possible movement by car. The weights of edges represent driving times. Additionally, we will consider possible transportation, which is not constrained to the road network.

1.2 Isochrone

Isochrones in multimodal spatial networks are again defined in [11]. Informally, they define isochrones as a sub-graph of the spatial network which contains all the nodes and edges which can be reached from a given starting point in specified amount of time. What this means is that the sum of weights of all edges on a shortest path between the source node and any node or edge in the isochrone is smaller than the time limit.

1.2.1 Visualization of isochrones

When showing the isochrone, we also want consider the possibility of moving away from the road network. For this reason we don't want to display only the isochrone sub-graph.

For this reason, [17] describes two approaches to visualizing them. These approaches transform isochrone in form of a subgraph into an area which encloses the isochrone.

The usual methods for converting a set of points into an area are concave hull and alpha shapes. Isochrone however, does not contain only points, it also contains edges. The requirement for them to be within the resulting area is hard to incorporate into alpha shapes or concave hulls [17].

Both of the approaches proposed in [17] work with buffers around objects in isochrone. This buffer represents the area in vicinity of an object within isochrone. This tries to simulate the possibility to be reachable from the source point not only for the object, but also for immediate vicinity of the object. The size of this buffer is a parameter to the method. What this means is that every object in the isochrone gets the same buffer. This does not correspond to the reality because even the objects on the very boundary of the isochrone get the same buffer. This may make the isochrone appear larger than it really is. To compensate for this the proposed methods make the input isochrone smaller by the size of the buffer.

The difference between the two methods is that the first one creates buffers around all the edges in the isochrone. The other method first creates a minimal bounding polygon and then creates a buffer around the polygon. It is claimed that for smaller

buffer sizes the second approach produces better results and, for larger buffer sizes there is no significant difference.

The flaw in these approaches is that the constant sized buffer is not really precise. If we want to simulate the possible movement by foot then the buffer has to be smaller close to the boundary of the isochrone than at the source point.

Other possible problem is that to calculate just the boundary of the area within which the isochrone is, the whole isochrone is calculated. If we could restrict the computation to only look for the boundary we could potentially lower the computation time.

This thesis will try to combine computation of isochrone and transformation of isochrone to an area. That might eliminate the need to compute the whole isochrone first, thus computing only the parts required for creating the area. But first we will introduce some methods to computing isochrones.

1.3 Algorithms for computing isochrones

Some algorithms for computing isochrones were introduced in recent years. In this section we will give an overview of them.

1.3.1 MINE

This algorithm was introduced in [11]. It was designed for computing isochrones in multimodal spatial networks where some edges are continuous and some are discrete. It is based on Dijkstra's algorithm for computing single source shortest paths. It works by incrementally expanding the resulting isochrone by edges within the spatial network. When traversing an edge would mean that the time would be higher than the threshold, the expansion is stopped at that edge.

Similar algorithm is also mentioned in [3]. However, that paper focuses on public transport and works with a modified spatial network where bus stops are on edges.

The memory complexity of the MINE algorithm is $O(|V^{iso}|)$, where V^{iso} is the number of nodes within the isochrone.

1.3.2 MINEX

MINEX is introduced in [10]. This algorithm is based on the aforementioned MINE algorithm. The improvement over the MINE algorithm is based on discarding elements from the isochrone which are no longer needed to expand the isochrone further. This leads to lowering of memory requirements of the algorithm if we are not required to keep the whole isochrone in memory.

MINE algorithm, just as the expanding Dijkstra's algorithm uses a set of closed nodes, which is used to avoid cyclic expansion of already expanded nodes. As the expansion progresses most of the nodes in the closed set are no longer needed [10].

To limit the size of the closed set, the MINEX algorithm introduces expired nodes. These are nodes which have all their neighbors either expired or in the closed set. The algorithm uses counters for each node to eagerly determine whether the node is expired. It is also proven that the expired nodes no longer need to be present in the closed set for the algorithm to be correct.

This reduces the memory requirements of the MINEX algorithm to $O(\sqrt{|V^{iso}|})$ in grid graphs, where V^{iso} is the number of nodes in the resulting isochrone.

1.4 Shortest path

We have seen that the algorithms specifically designed for finding isochrones are based on Dijkstra's algorithm for single source shortest paths problem. For real world maps, various approaches were developed to speed up finding of shortest routes. Most of these approaches use some sort of preprocessing to speed up the searches. A rather long list can be found in [6]. We will now go through some of these approaches, and we will try to tell whether they might be useful for computing isochrones.

1.4.1 A*

This approach is based on Dijkstra's algorithm and a heuristic which directs it towards the destination. Since in isochrones there is no destination this approach is not useful.

1.4.2 Bidirectional search

When looking for the shortest path between two points, using bidirectional search can speed up the process by decreasing the number of inspected nodes in the road graph. Thus bidirectional search is a standard technique used in many algorithms for shortest path. In fact most of the algorithms mentioned in [6] use bidirectional search.

On the other hand, because bidirectional search requires both source and destination points, algorithms for searching for isochrone will probably not benefit from it.

1.4.3 Highway hierarchies

Highway hierarchies is an approach for computing shortest paths which is described in [24]. The approach is based on an idea used by many commercial navigation systems – that when computing shortest path between two points, the less important roads are used only near the start and the end of the route.

Highway hierarchies formalize the notion of important roads. The only parameter to the method is the distance within which queries are considered local. Then individual roads are checked whether they are used in non local queries. If they are, they are put into a new layer. This layer is a new road network which consists only of the important roads.

This process can be repeated to create multiple levels of detail. The algorithm for finding the shortest paths then uses roads on different levels based on the distance between the source and the destination. If the source and destination are close to each other the algorithm is equivalent to Dijkstra's algorithm. Otherwise, the search is advanced to a higher level where the road graph is more sparse. This means that the search in the higher level can traverse longer distances while examining less nodes.

The problem with this approach is the preprocessing phase where we need to efficiently compute which roads are important. An solution to this problem is shown in [24] which finds the important roads in linear time.

What this approach could bring to computing isochrones is that if only the important roads are used for travelling longer distances, it may not be needed to examine all roads

when searching for larger isochrones. It may be sufficient to use only the more sparse levels of the hierarchy. However, because highway hierarchies use bidirectional search to know when to descend to lower levels of the hierarchy, it will be hard to anticipate that when searching for isochrones.

1.4.4 Subdivision

Subdividing the road graph into multiple regions was studied by multiple authors with reasonable success.

The approach studied in [8] works by subdividing the road graph into connected subgraphs. The author examines how to create a subdivision which gives the best results. It is noted that finding an optimal subdivision is a hard problem and some heuristic algorithms are presented.

After the road graph is subdivided, each region is replaced by a simpler graph. This graph keeps the border nodes of the region. These are connected in a way that preserves costs of paths within the region.

That allows for a modified Dijkstra's algorithm to search on a graph created by the regions. This reduces the search space roughly by a factor equal to the size of the regions. This can lead to considerable speedups [8].

Other way examined in [15] subdivides the graph into smaller graphs by removing nodes. Using a heuristic algorithm from [13], the authors achieve subdivision into regions of relatively same size with small amount of nodes removed.

The subdivided road graph is then used in a similair way than in the approach in [8]. The difference is that the border nodes are the removed nodes so there is no need to keep any of the original edges as part of the reduced graph used by the final Dijkstra's algorithm.

The advantage of subdivision approaches is that they can be adapted for isochrone searching. The regions created by the preprocessing step can be used in isochrone searching just as they are used in shortest path searching. This should result in some speedup when searching for larger isochrones.

1.4.5 Edge flags

Flagging edges examined in [15] also provides speedup for shortest path searching between two points. This approach is also based on subdivision. The road network is divided into multiple regions and each edge is assigned a boolean flag for each region. These flags are set to true for those regions which contain nodes towards which a shortest path uses the edge. It is also set to true also for edges which lie in the region.

This then allows for faster searching for shortest route because only the edges with flag for the region with the destination node set to true need to be looked at. According to [15] this speedup is significant.

Since with isochrone it is impossible to determine the destination region, this approach will not help much. However the idea of flags for nodes in the road graph may be useful.

1.4.6 Transit nodes

Transit nodes [1] are based on the observation that for road graphs there exists a relatively small set of nodes (call them transit nodes) with the property that every shortest path between nodes which are far apart goes through at least one of these transit nodes. Other property of these transit nodes is that the first transit node encountered on a shortest path from specific node is one of a very small set (call these entry nodes) [1].

Because the set of transit nodes is small, it is possible to precompute distance between all pairs of transit nodes. It is also possible to precompute distances from each node to each of its entry nodes. This then allows for answering distance queries in three table lookups [1].

The only problem is identifying the transit nodes and the entry nodes for each node. In [1] several ways are presented for identifying them. First approach is based on grid decomposition of the road graph. Other two approaches pick transit nodes from highway hierarchies.

From approaches examined in [6] the transit nodes achieve highest speedups. However

the preprocessing time is rather long and the resulting data rather large.

The advantage of this approach is the sheer speed with which it can answer distance queries. This can be useful for determining if a some points lie in the isochrone. This can help to get the idea of how large the isochrone is.

1.4.7 Contraction hierarchies

Contraction hierarchies introduced in [12] is another approach for hastening shortest path queries. Same as with highway hierarchies, the idea is to create a hierarchy of more and more simplified road graphs and then searching for the shortest path on multiple levels based on distance between the points.

The difference is that while highway hierarchies remove edges from the road graph which will not be used in queries for shortest path between distant points, contraction hierarchies remove nodes from the road graph. Nodes get an assigned 'importance'. Then each new level is created by removing non important nodes and by adding shortcuts which ensure that the cost of shortest paths stays the same.

Assigning importance to nodes is not an easy task, but [12] proposes some approaches to deal with it. The authors also present a way of determining which shortcuts are really necessary, as they point out that not all are.

Usefulness of contraction hierarchies for isochrones is probably the same as for highway hierarchies. The preprocessed data can probably be used for isochrones in a similar way than for shortest paths. However, same as with highway hierarchies it will probably be difficult to determine when to descend to levels with higher detail on the edge of isochrone.

1.4.8 SHARC

The SHARC algorithm is introduced in [2]. It is a combination of several approaches. It uses ideas from edge labels, contraction hierarchies and highway hierarchies.

The preprocessing is done by creating shortcuts in the same way as in contraction hierarchies. When creating shortcuts, the bypassed edges get assigned labels in the

same way as in edge labels method. These labels use a decomposition computed at the beginning of the preprocessing step. The problem of determining the values of labels is solved by using an approach inspired by highway hierarchies.

1.5 Nearest neighbors

The problem of k nearest neighbors is not directly related to computing isochrones but it is a similar problem which got a little more attention. The approaches used in solving this problem could help improve approaches for computing isochrones.

The problem of k nearest neighbors is to find k nearest road graph nodes from a given set (the so called points of interest), which are closest to a starting point.

Same as with shortest paths there are approaches which do not use preprocessing and approaches which use preprocessing. As expected, the approaches with preprocessing are faster, but the preprocessing takes time and the result of preprocessing takes up space.

In the end, we did not use any of these approaches to compute isochrones.

1.5.1 Incremental Euclidean Restriction

This approach introduced in [20] is one of those that does not use preprocessing. The high level idea is described in listing 1.1

Listing 1.1: Incremental Euclidean Restriction

```
Input: s - starting point, I - points of interest,
       R - road network
current = find_nearest_point(I, s)
I = I - current
while true:
    net_dist = road_distance(R, s, current)
    next = find_nearest_point(I)
    next_dist = euclidean_distance(s, next)
    if next_dist >= net_dist:
        return current
```

```
11      end
12      current = next
13      I = I - current
14  end
```

This is the algorithm for $k = 1$. Its extension to other values is rather straitforward and is explained in [20].

According to [20] this approach is useful when there is a relatively small difference between euclidean distance and network distance between any two road graph nodes.

1.5.2 Incremental Network Expansion

In case that the difference is larger, another algorithm is presented in [20]. It is based on breadth first search of the network. The search space is limited by the euclidean distance to the found points of interest.

This algorithm really resembles Dijkstra's algorithm. The only difference is the restriction of search space by the euclidean distance.

1.5.3 Voronoi based KNN

This approach described in [16] uses preprocessing to speed up k nearest neighbor queries. This approach generates a Voronoi diagram with each point of interest in the center of each cell. The distance used in Voronoi diagrams is the network distance.

After constructing the Voronoi diagram, finding the nearest neighbor is as simple as determining in which cell the starting point is. However finding other nearest points is a little harder. How this approach deals with it is that it precomputes network distances between points on cell boundary. This helps in the same way as precomputation of distances between boundary nodes in subdivision methods when searching for shortest paths.

Chapter 2

Our approach

In this chapter, we will introduce our approach to solving the problem of finding isochrones. First we will describe the model we are using. Then we will describe the computation of isochrones.

2.1 Model

Isochrones are defined in papers in different ways. These definitions are usually very general. In this section we will formally specify the isochrones we are investigating.

We are trying to find isochrones in a map. We assume that the map is a directed graph. Edges of the graph represent roads. Each node of the graph corresponds to a point on the surface of the Earth. This is similar to embedding of the graph into the surface of the earth. However, edges in a map can intersect (e.g. bridges or tunnels).

We consider two modes of transportation. First one (driving mode) is constrained to edges of the map, the other (walking mode) allows movement over the whole surface.

We have an actor starting at a specific point. Each edge has a cost corresponding to the time it takes to traverse this edge using the first mode (driving) of transportation. The time of traveling between any two points using the second mode (walking) of transportation is the distance between these two points. We also assume that the cost of an edge is not larger than the distance between its endpoints.

This represents the possibility to travel using a car and going on foot. The car is constrained to roads, but we can travel anywhere on foot. This assumption is a simplification, because there can be obstacles which prevent traversing on foot, e.g. fences, buildings, or lakes.

The isochrone for given starting point and a time limit is the set of points which can be reached by such actor from the starting point within the time limit.

In the investigated model, the mode of transportation can be switched as many times as needed. This represents the option, that the actor has planned the route ahead and has cars at specific places.

Another option is not to allow switching back to the first mode of transportation after switching to the second. This corresponds to traveling by car, then getting out and traveling on foot.

2.1.1 Formal definitions

We begin by a definition of a road network inspired by [25]. We define an efficient road network for which the roads are at least as fast as going on foot.

First we need to define what we consider to be a map surface. For simplification we consider only a few metric spaces to be a map surface.

Definition 1 (Map surface)**.** Map surface is a tuple (M, d_M), where M is a set, and d_M is a function, $d_M : M \times M \to \mathbb{R}_0^+$, d_M is a metric over M. M and d_M are either:

1. $M = (\mathbb{R} \times \mathbb{R}, d_M = d_{(\alpha,\beta)})$, for some $\alpha, \beta \in \mathbb{R}$. The function $d_{(\alpha,\beta)}$ is defined as follows:

$$d_{(\alpha,\beta)}((x_1, y_1), (x_2, y_2)) = \sqrt{((y_2 - y_1) \cdot \beta)^2 + ((x_2 - x_1) \cdot \alpha)^2}.$$

   Note that $d_{(\alpha,\beta)}$ is a metric over $\mathbb{R} \times \mathbb{R}$ which measures the distance as if the plane was stretched by α horizontally, and by β vertically.

2. $M = S, d_M = d_S$, where S is a sphere and d_S is the great-circle distance over this sphere. The great-circle distance is the shortest distance between two points over the surface of the sphere.

3. $M = \mathbb{R}$, and

$$d_M(a, b) = |b - a|.$$

In this definition, d_M is the cost function for the walking mode of transportation.

Definition 2 (Efficient road network)**.** Let (M, d_M) be a map surface. Let G be a weighted directed graph with $N(G)$ being the set of its nodes, $E(G)$ the set of its edges, and $d_G : E(G) \to \mathbb{R}^+$ the cost function for edges of the graph. Let $c : N(G) \to M$ be a function which for each node in G specifies its position in M. The tuple (G, c, M, d_M) is an efficient road network iff $\forall e = (e_1, e_2) \in E(G) : d_G(e) \leq d_M(c(e_1), c(e_2))$. G is sometimes called the road graph.

The d_G function specifies the cost of traversing an edge using the driving mode of transportation.

Next we define some utility objects. Edge space is the part of M which is covered by an edge. Union of the edge spaces of all edges in $E(G)$ is the road space. This represents the part of M where there are roads. Given a point $p \in M$ we want to know the edges which go through this point. This is given by the graph edges function e_R. We also need to define what intersection of edges mean.

Definition 3 (Edge space)**.** Let $R = (G, c, M, d_M)$ be an efficient road network.

Edge space of an edge $e = (e_1, e_2) \in E(G)$ is the set

$$M_e = \{p \in M; d_M(c(e_1), p) + d_M(p, c(e_2)) = d_M(c(e_1), c(e_2))\}.$$

Definition 4 (Road space)**.** Let $R = (G, c, M, d_M)$ be an efficient road network.

Road space is the set

$$M_R = \bigcup_{e \in E(G)} M_e$$

Definition 5 (Graph edges)**.** Let $R = (G, c, M, d_M)$ be an efficient road network.

For each point p in the road space M_R we define its graph edges by means of function e_R

$$e_R : M_R \to \mathcal{P}(E(G)),$$
$$e_R(p) = \{e \in E(G); p \in M_e\},$$

Where $\mathcal{P}$ is the power set.

Definition 6 (Intersecting edges)**.** Let $R = (G, c, M, d_M)$ be an efficient road network.

We say that edges $e = (e_1, e_2) \in E(G)$, and $f = (f_1, f_2) \in E(G)$ intersect iff $\{e_1, e_2\} \cap \{f_1, f_2\} = \emptyset$ and $M_e \cap M_f \neq \emptyset$.

Then we define the road distance between any two points of the road space. Between points corresponding to nodes of the graph it is the cost of the shortest path between the nodes. We also extend it to points which lie on edges.

Definition 7 (Road distance)**.** Let (G, c, M, d_M) be an efficient road network. Let $d_G : N(G) \times N(G) \to \mathbb{R}_0^+$ be the shortest distance function over nodes of G.

We begin by defining road distance in edge space of an edge $e = (e_1, e_2)$ as

$$d_e : M_e \times M_e \to \mathbb{R}_0^+,$$
$$d_e(p, q) = \frac{d_M(p, q)}{d_M(e_1, e_2)} \cdot d_G(e_1, e_2).$$

Then we define it for arbitrary points in the road space. We start with a helper function:

$$d'_0 : M_R \times M_R \to \mathbb{R}_0^+,$$
$$d'_0(p, q) = \min_{\substack{e=(e_1,e_2)\in e_R(p) \\ f=(f_1,f_2)\in e_R(q)}} d_e(p, c(e_2)) + d_G(e_2, f_1) + d_f(c(e_1), q).$$

And finally we define the road distance as

$$d_0 : M_R \times M_R \to \mathbb{R}_0^+,$$
$$d_0(p, q) = \begin{cases} d'_0(p, q), \text{if } e_R(p) \cap e_R(q) = \emptyset \\ \min(d'_0(p, q), \min\limits_{e \in e_R(p) \cap e_R(q)} d_e(p, q)) \end{cases}$$

Note that we need a special case for when p and q lie on the same edge.

Then we define the distance which allows for at most k switches between the modes of transportation.

Definition 8 (k-switch distance)**.** Let $R = (G, c, M, d_M)$ be an efficient road network. The k-switch distance d_k between two points $p \in M_R$ and $q \in M$ is defined inductively as

$$d_k(p, q) = \min_{r \in M_R} d_{k-1}(p, r) + d_M(r, q), k \geq 1.$$

d_0 is the road distance function from above. Note that d_{k-1} is used with both parameters from the road space. This allows us to use d_0 as the basis.

Definition 9 (Minimum distance)**.** Let $R = (G, c, M, d_M)$ be an efficient road network. The minimum distance d_∞ between two points $p \in M_R$ and $q \in M$ is defined as

$$d_\infty(p, q) = \min_{k \in \mathbb{N}} d_k(p, q).$$

Definition 10 (Isochrone)**.** Let $R = (G, c, M, d_M)$ be an efficient road network. Let $s \in M_R$ be the starting point. Let $l \in \mathbb{R}$ be the time limit. Isochrone is the set

$$I(s, l) = \{p \in M; d_\infty(s, p) \leq l\}.$$

The other possible definition uses 1-switch distance instead of minimum distance.

Definition 11 (Single switch isochrone)**.** Let $R = (G, c, M, d_M)$ be an efficient road network. Let $s \in M_R$ be the starting point. Let $l \in \mathbb{R}$ be the time limit. Single switch isochrone is the set

$$I_1(s, l) = \{p \in M; d_1(s, p) \leq l\}.$$

2.2 Computing isochrones

Our approach to computing isochrones is based on dividing the isochrone computation into two stages:

- The first stage tries to find nodes of the road graph which are reachable from the specified starting point in specified time.
- The second phase then uses these nodes to find all points of the isochrone over the whole surface of the Earth.

To achieve this, we need

$$\forall p \in M_R, q \in M : d_1(p, q) = d_\infty(p, q).$$

We start by showing what problems in road networks can prevent this two step approach from being used. Then we will try to show ways to amend the graph so that this approach can be used.

2.2.1 Disconnected components

First such problem is that the road graph may not be connected. This leads to problems when finding distances between points corresponding to nodes in different components.

For example consider road network $R = (G, c, \mathbb{R}, d_M)$ in figure 2.1, where

$$N(G) = \{0, 1, 2\},$$
$$E(G) = \{(1, 2)\},$$
$$d_G(1, 2) = 0.1,$$
$$\forall n \in N(G) : c(n) = n.$$

In this network

$$d_\infty(0, 2) = 1.1,$$
$$d_1(0, 2) = 2.0$$

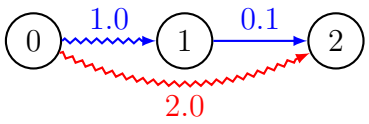

Figure 2.1: Example of a disconnected graph, curly lines show distances computed as d_M and the straight line represents the road graph. Red arrow shows d_1, blue arrows show d_∞

When this graph is connected by adding the edge $(0, 1)$ with cost 1.0 (see figure 2.2), we achieve that

$$d_1(0, 2) = d_\infty(0, 2) = 1.1$$

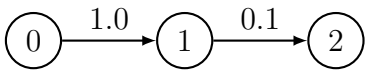

Figure 2.2: By connecting the graph from figure 2.1, we achieve the same value for d_1 and d_∞.

To show how this situation can occur in real map, refer to figure 2.3. Another situation when this can occur are incomplete data, where not all roads are identified in the map.

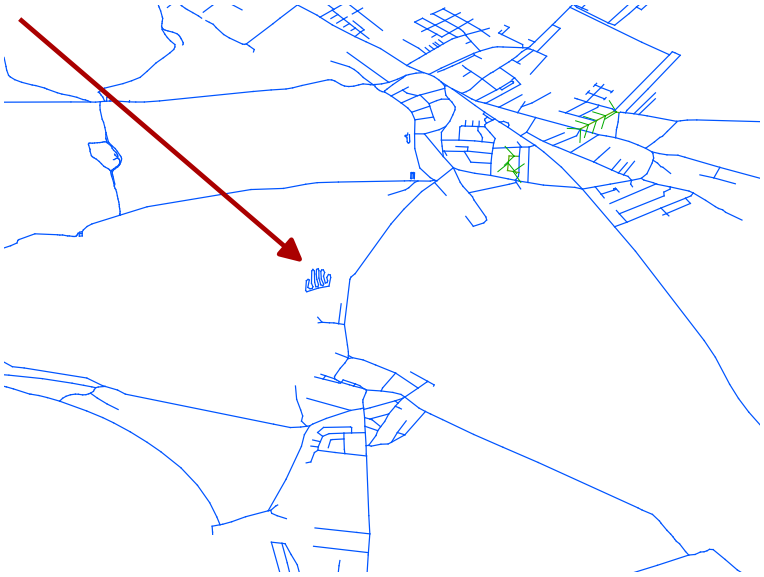

Figure 2.3: Example of disconnected map at 48.068°N, 17.249°E, the arrow points to a race track which is not connected to the road network.

2.2.2 Intersections

Another problem are intersecting roads without a node in the intersection. For example consider the road network $R = (G, c, \mathbb{R} \times \mathbb{R}, d_{(1,1)})$ (see figure 2.4), where

$$N(G) = \{(0,0), (2,2), (0,2), (2,0)\},$$
$$E(G) = \{((0,0), (2,2)), ((0,2), (2,0))\},$$
$$\forall n \in N(G) : c(n) = n,$$
$$\forall e = (e_1, e_2) \in E(G) : d_G(e_1, e_2) = 0.1$$

In this network again

$$d_\infty((0,0), (2,0)) = 0.1,$$
$$d_1((0,0), (2,0)) = 0.05 + \sqrt{2}.$$

Changing the network in this situation is a little more contrived. We need to insert a node i at the intersection of the intersecting edges. Then we need to remove the intersecting edges and for each such edge $e = (e_1, e_2)$ we need to add new edges (e_1, i) with cost $d_e(e_1, i)$ and (i, e_2) with cost $d_e(i, e_2)$ (see figure 2.5).

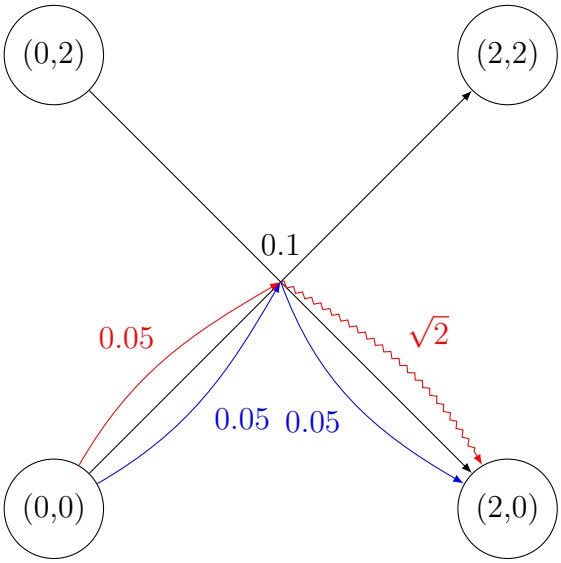

Figure 2.4: An example of intersecting edges, curly lines show distances computed as d_M and straight lines represent distances computed as d_G. Red shows the d_1 distance, blue shows the d_∞ distance.

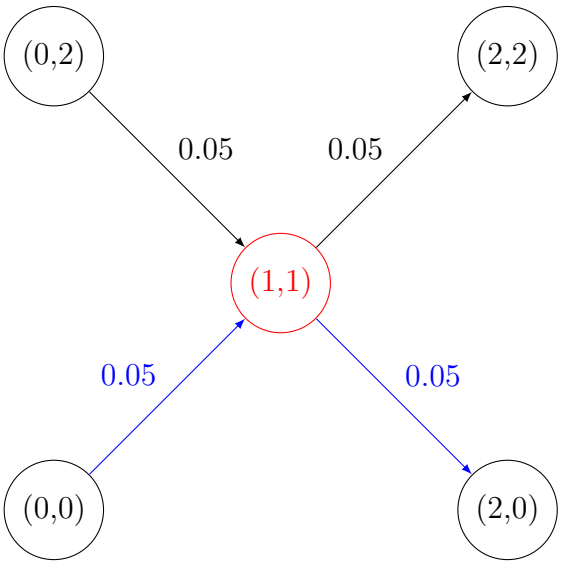

Figure 2.5: The road graph from figure 2.4 with intersection removed by adding a node at the intersection (red colour). Now $d_1((0,0),(2,0))$ is equal to $d_\infty((0,0),(2,0))$ (blue colour).

There are basically two reasons for occurrence of intersections in the road network. One is wrong data. The map data being used may not have completely correct edges and some may have wrong coordinates and intersect.

The other reason are bridges and tunnels. See figure 2.6 for an example. These pose a question of whether these intersections should be handled by adding the node at intersection point, because that would imply the ability to jump off a bridge or dig out of a tunnel. There are situations where that may be possible and we may want to incorporate that into our isochrone computations, but there are also situations where this is not desired.

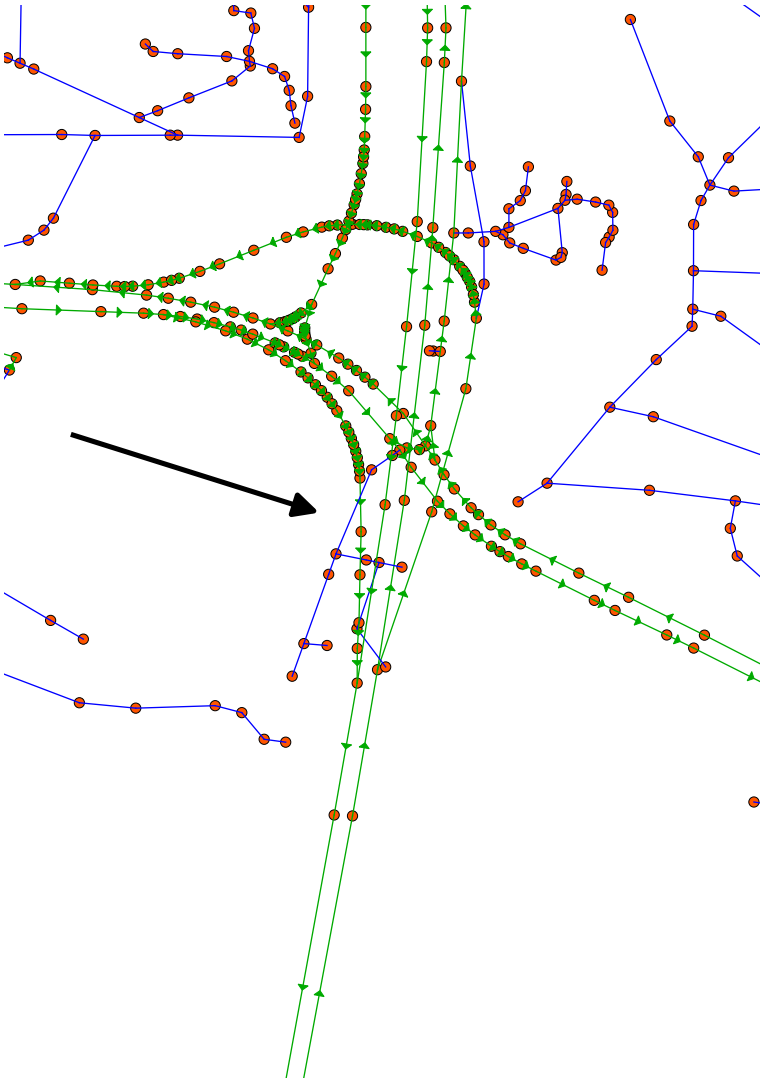

Figure 2.6: Example of intersections without node. There are intersecting edges where one is a bridge and the other is a road beneath the bridge. At 48.145N, 17.075E.

This situation can be resolved by specifying which edges of the road graph are the so called *bridges*. Each edge which cannot be left (e.g. a tunnel or a bridge from which it is impossible to jump off) should be marked as *bridge*. Points within road space which are only on these *bridge* edges cannot be used as a starting point for traveling on foot. This leads to a different distance definition.

Definition 12 (k-switch distance with bridges)**.** Let $R = (G, c, M, d_M)$ be an efficient road network. Let $B \subseteq E(G)$ be the set of *bridge* edges.

First, we define road space without bridges as

$$M_R^B = \bigcup_{e \in E(G) \setminus B} M_e$$

The k-switch distance with bridges between two points $p \in M_R$ and $q \in M$ is defined recursively as

$$d_k(p, q) = \min_{r \in M_R^B} d_{k-1}(p, r) + d_M(r, q), k \geq 1.$$

Based on this definition of k-switch distance, we can create an altered definition of minimum distance and isochrone.

Since this is only a minor modification, which can be easily added to our approach, we have not dealt with it further.

2.2.3 Amending graph

Even after removing the problems from the sections above, there are still situations when there are points $p, q \in M_R$ for which $d_\infty(p, q) \neq d_0(p, q)$. We try to reduce the number of such pairs by adding new edges to the graph. Each such edge $e = (e_1, e_2)$ has its cost equal to $d_M(e_1, e_2)$.

We tried two heuristic approaches to deal with this issue.

The first one was based on adding edges $e = (e_1, e_2)$ for which

$$d_M(c(e_1), c(e_2)) < d_G(e_1, e_2) \leq C$$

for some $C \in \mathbb{R}$.

This can be easily implemented by starting a Dijkstra search from each node and stopping when reaching C. The constant C limits the search scope making this step feasible. For each found node the d_M distance is computed and if it is lower than the Dijkstra distance obtained in the search, the edge is added. This leads to adding some non required edges.

Another disadvantage of this approach is that the added edges may create intersections, which lead to the problem with intersections. Removing this problem creates even more edges and some nodes.

The other approach starts by removing the intersections, making the graph connected, and considering the road graph embedded into the map surface. This allows for identifying faces in this embedding. Then we investigate each face and look at it as a polygon.

Definition 13 (Road graph face)**.** Let $R = (G, c, M, d_M)$ be an efficient road network, where no edges intersect. Let $C = (n_0, n_1, \ldots, n_k = n_0)$ be a cycle in G. Let

$$E_C = \{e \in E(G) : \exists 0 \leq i < k : e = (n_i, n_{i+1}\} \tag{2.1}$$

$$M_C = \bigcup_{e \in E_C} M_e \tag{2.2}$$

An area F in M surrounded by M_C is a road graph face iff

$$\forall n \in N(G) \setminus C : c(n) \notin F$$

Area nodes of face F are

$$N_F = \{n_0, n_1, \ldots, n_{k-1}\}$$

Area edges of face F are

$$E_F = E_C$$

We will denote by $F(R)$ the set of all faces in the road network R. For each node we denote

$$F_n = F \in F(R), n \in N_F,$$

i.e. the nodes which lie on the boundary of F.

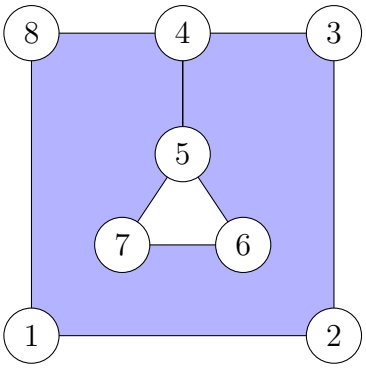

Figure 2.7: Example of a face in a road graph. Blue area is the road graph face.

See figure 2.7 for an example.

The intersections need to be removed so that the faces of the graph form simple polygons. Also, the graph needs to be connected to avoid having holes in the polygon. These actions allow for simpler finding of the faces of the road graph.

Then we look at each face and try to add edges which make going between two boundary nodes of the face faster. Finding these edges is a similar problem to finding a convex decomposition of the face. There are known algorithms for convex decomposition, e.g. Hertel-Mehlhorn algorithm [14]. This algorithm is based on finding a triangulation of the polygon and then removing edges which do not break the convexity of the resulting pieces. Even though it is not optimal, it gives a reasonable approximation of the minimal convex decomposition.

For our requirements we used a similar approach. We start with a triangulation of the polygon. Then we consider each diagonal edge added by triangulation and compute the distance between its endpoints as if the diagonal was not present. If this distance is smaller than the cost of the added edge, the edge is removed. See figure 2.8

The drawback of this approach is that depending on the structure of the input road network, it may not add necessary edges. This may also happen for arbitrarily short edges. This happens because the edges added must be local to the faces of the graph. If they are not, they are not found by the triangulation. See figure 2.9 for an example. The previous approach does not have this problem.

In chapter 6 we compare these two approaches.

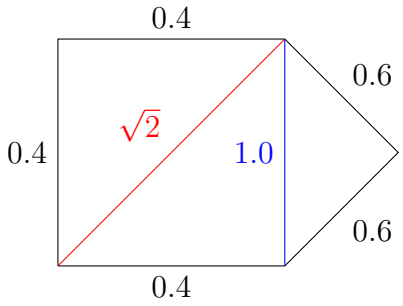

Figure 2.8: An example of the algorithm based on Hertel-Mehlhorn algorithm for minimal convex decomposition. The black pentagon represents the original graph face. The numbers denote the costs of traversing each edge. Triangulation added the red and blue edges. The blue edge is kept, because the shortest path between its endpoints has a cost of 1.2. The red edge is removed, because the cost of shortest path between its endpoints is 0.8

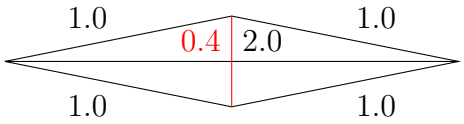

Figure 2.9: Example of an edge missed by face based adding of edges. Even though the red edge should be added, it is not because it spans two faces. Note that the missed edge could be arbitrarily short.

Chapter 3

Finding the isochrone

In this chapter we will describe our approach to finding the nodes of the road graph within the isochrone. Then we describe a way to find the rest of the points within the isochrones.

3.1 Simple search

Dijkstra algorithm [7] solves the single source shortest path problem. It works by keeping information about the cost of the shortest path to each node in the graph. The nodes are kept in a priority queue sorted by the cost of the shortest path to them from the starting node. When a node n is taken from the priority queue with cost $c(n)$, all its neighbors n' have their costs $c(n')$ in the priority queue updated as

$$c(n') = \min(c(n'), c(n) + d_G(n, n')),$$

where $d_G(n, n')$ is the cost of the edge (n, n').

This means that the nodes are taken from the priority queue in the order defined by their distance from the starting point. This leads to a simple algorithm for finding all nodes which lie within the isochrone $I_1(s, l)$. We do a Dijkstra search from s, and stop it when a node with a cost larger than l is taken from the priority queue.

This algorithm returns all the nodes of the road graph that lie within the isochrone. Additionally, for each such node n we have the remaining cost, that is $r_n = l - d_1(s, n)$

3.2 Expanding from the road graph nodes

In this section, we will describe how we get from the road graph nodes within the isochrone to the isochrone itself.

To do that we look at each node n and each edge $e = (n, o)$ for which n is the starting point. Edge e has cost c_e where $d_1(s, n) \leq l$, and $d_1(s, o) > l$. We also have the remaining cost at n, $r_n = l - d_1(s, o)$.

When walking from n and having r_n time left, we can reach a circle of radius r_n centered at $c(n)$. The other reachable points can be reached by traversing part of the edge (up to time t) and then walking the rest of in time $(r_n - t)$. For each point p on e we can compute the remaining cost for that point as

$$r_p = r_n - \frac{d_M(c(n), p)}{d_M(c(n), c(o))} \cdot c_e.$$

Then we can apply the same approach as with the $c(n)$ point and use a circle of radius r_p.

The union of all these circles makes up a part of the isochrone reachable from n through e. See figure 3.1 for an example.

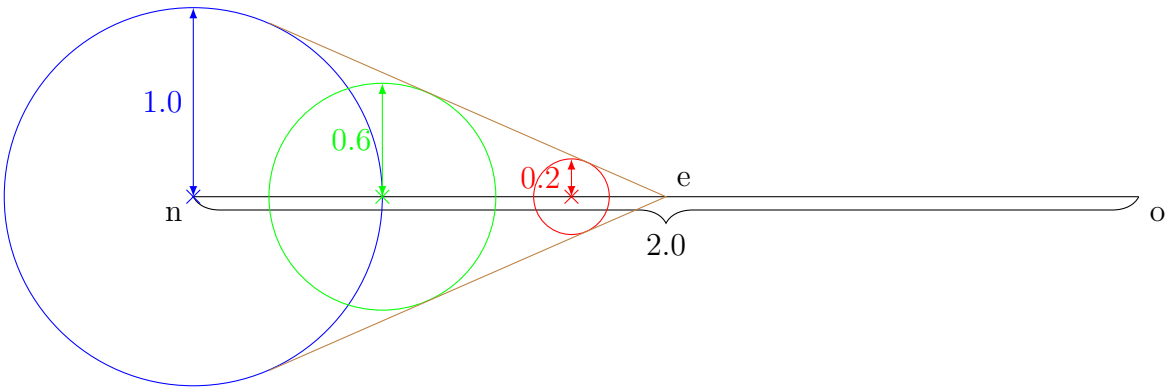

Figure 3.1: Points reachable from n in r_n via edge e when $r_n < c_e$. The colored circles show areas reachable from points marked by crosses of corresponding colors. The brown lines enclose the union of all such circles.

We will show another example which shows how this looks when we have an edge $e = (e_1, e_2)$ and $r_{e_1} > r_{e_2}$ and $r_{e_1} < r_{e_2} + c_e$, where c_e is the cost of edge e, in figure 3.2.

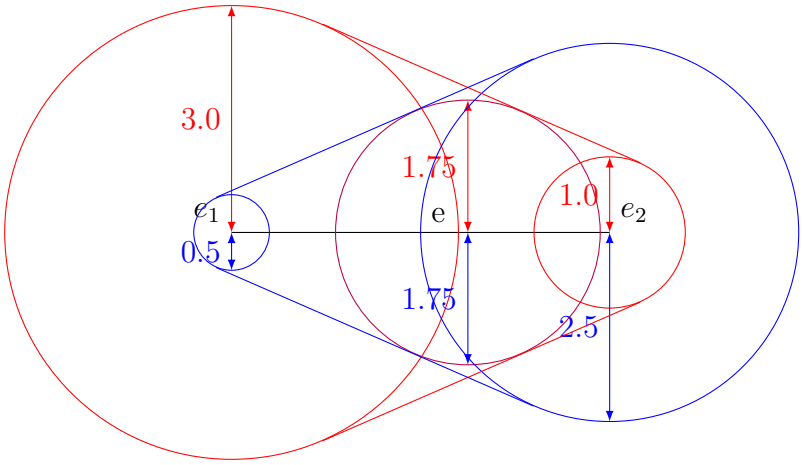

Figure 3.2: Points reachable from e_1 and e_2 via edge e when $r_{e_1} > r_{e_2}$ and $r_{e_1} < r_{e_2} + c_e$. Red lines show points reachable from e_1, blue ones show points reachable from e_2.

3.3 Reducing the number of nodes

It turns out that not all the road graph nodes within the isochrone are needed to find the isochrone itself. In this section we will describe the way this is possible.

We start by looking at the faces of the road graph, similarly to how we did when we were trying to add edges to the graph in previous chapter. We have an isochrone I. We examine the sets

$$PartiallyCovered = \bigcup_{\substack{F \in F(R) \\ F \not\subseteq I \\ F \cap I \neq \emptyset}} F,$$

and

$$Covered = \bigcup_{\substack{F \in F(R) \\ F \subseteq I \\ F' \in F(R) \\ F' \subseteq PartiallyCovered \\ N_F \cap N_{F'} \neq \emptyset}} F.$$

The *Covered* set forms a set of polygons. These polygons can have holes. There can be two reasons for these holes. The first reason is that not all the points within the hole lie within the isochrone. The other reason is, that the the faces within this hole are not touching any node in *PartiallyCovered*, but still lie within the isochrone.

We also look at *PartiallyCovered*. If we make a union of all the faces in it, we also get a set of polygons. We remove holes from all of these polygons. After that, of them will

contain the starting node. If we ignore the polygon with the starting node in it, we get the union of faces which lie within the union of $Covered$, and are not completely covered by the isochrone. We will denote this set of points $PartiallyCovered'$.

Now when we look at $Covered$, remove all the holes, and subtract $PartiallyCovered'$, we get the set of points we will call $IsoInterior$.

This way we partitioned the isochrone I into $IsoInterior$, and

$$IsoDetails = PartiallyCovered \cap I = I \setminus IsoInterior.$$

Now we need to find a set of nodes, which allows us to find $PartiallyCovered$, $Covered$, and $PartiallyCovered \cap I$.

When given a face F and a node n from N_F (nodes which form the face), we need to find a limit function $l(n, F)$ such that

$$\forall p \in F : d_1(c(n), p) \leq l(n, F).$$

The problem is that $l(n, F) > r_n$ does not imply, that $F \subseteq PartiallyCovered$. So instead of finding $Covered$ and $PartiallyCovered$, we will search for

$$PartiallyCovered_2 = \bigcup_{\substack{F \in F(R) \\ n \in N(G) \\ c(n) \in I \\ l(n,F) > r_n}} F,$$

$$Covered_2 = \bigcup_{\substack{F \in F(R) \\ F \subseteq I \\ F' \in F(R) \\ F' \subseteq PartiallyCovered_2 \\ N_F \cap N_{F'} \neq \emptyset}} F.$$

Note that $Covered_2$ is defined in terms of $PartiallyCovered_2$ in the same way $Covered$ was defined in terms of $PartiallyCovered$. Also,

$$PartiallyCovered_2 = PartiallyCovered$$

if $l(n, F)$ was minimal for each $n \in N(G), F \in F(R)$. This means, that the construction of isochrone from $PartiallyCovered$ and $Covered$ also works with $PartiallyCovered_2$ and $Covered_2$. This also allows us to define $IsoDetails_2$, and $PartiallyCovered'_2$ using

$PartiallyCovered_2$ similarly to how we defined $IsoDetails$, and $PartiallyCovered'$ from $PartiallyCovered$.

Using the limit function, we can find

$$N_{Border} = \{n \in N(G) : \exists F \in F(R), l(n, F) > r_n\},$$

where r_n is the remaining time at n. It can be seen, that

$$\bigcup_{\substack{F \in F(R) \\ N_F \cap N_{Border} \neq \emptyset}} F = PartiallyCovered_2.$$

It can be also seen, that when expanding from N_{Border} using the techniques from section 3.2, we get a superset of $IsoDetails_2$.

What remains, is to find nodes, which allow us to find $Covered_2$. To do that, we define function $l : N(G) \to \mathbb{R}$ as

$$\begin{aligned} l_0 &: N(G) \to \mathbb{R}, \\ l_0(n) &= \max_{F \in F_n} l(n, F), \\ l(n) &= \max_{\substack{F \in F_n \\ n' \in N_F}} l_0(n') + l(n, F). \end{aligned}$$

Lemma 1.

$$n \in N_{Border} \Rightarrow l(n) > r_n$$

Proof. Let $n \in N_{Border}$. Then $\exists F \in F(R)$, such that $l(n, F) > r_n$. But then

$$l(n) \geq l_0(n) \geq l(n, F) > r_n.$$

□

Now we look at all the nodes n, for which $l(n) > r_n$. We denote these N_l.

Lemma 2.

$$Covered_2 \subseteq \bigcup_{\substack{F \in F(R) \\ n \in N_l \\ l(n,F) \leq r_n}} F$$

Proof. Let $p \in Covered_2$. Then $\exists F \in F(R)$, such that $p \in F$, and there is $F_B \in F(R)$ which borders with F. Also $\forall n_B \in N_{F_B} : l(n_B, F_B) > r_{n_B}$.

Let $n_B \in N_{F_B} \cap N_F$. Then $r_{n_B} < l(n_B, F_B)$. Also let $n \in N_F$. It holds that

$$r_n - d(n, n_B) \leq r_{n_B}.$$

If it was not the case, then there would be a shorter path from the starting point to n_B via n. Also, from definition of l_0

$$l_0(n_B) \geq l(n_B, F_B) > r_{n_B}.$$

But then

$$l_0(n_B) \geq l(n_B, F_B) > r_{n_B} \geq r_n - d(n, n_B).$$

Also, because $d(n, n_B) = d_1(c(n), c(n_B)) \leq l(n, F)$,

$$l(n) \geq l_0(n_B) + l(n, F) \geq l_0(n_B) + d(n, n_B) > r_{n_B} + d(n, n_B) \geq r_n.$$

And because $l(n) > r_n$, $n \in N_l$. This way we proved that $\forall n \in N_F : n \in N_l$.

Now for F to be within $Covered_2$, there must be a node $n' \in N_F$ for which $l(n', F) \leq r_{n'}$. Otherwise there would be a point in F which does not lie within the isochrone. But from above

$$n' \in N_l,$$

and then

$$F \subseteq \bigcup_{\substack{F' \in F(R) \\ n' \in N_l \\ l(n', F') \leq r_{n'}}} F'$$

□

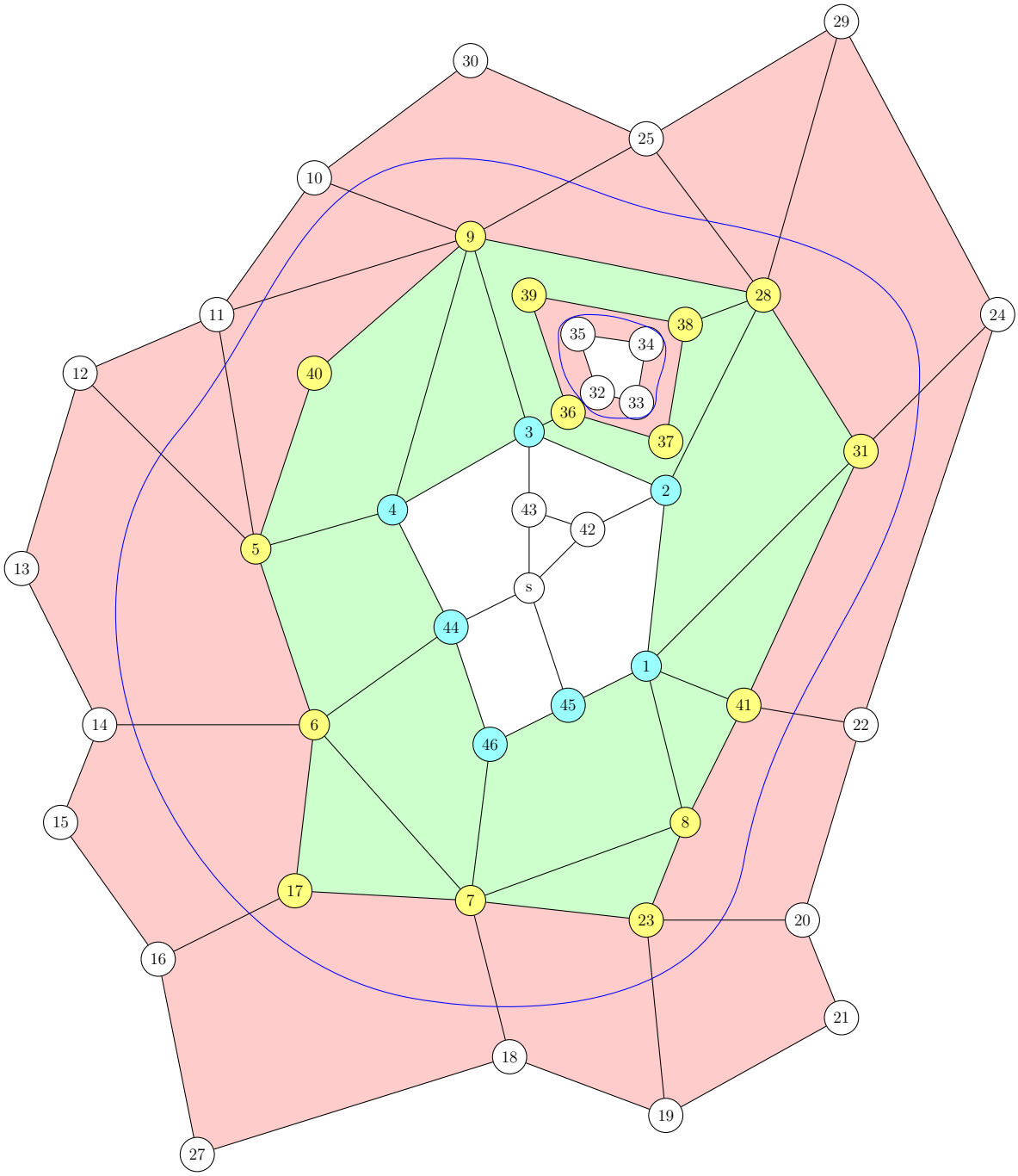

Figure 3.3: Isochrone example. s is the starting point, the nodes in N_{Border} have yellow color, nodes n, for which $l(n) > r_n$ and $n \notin N_{Border}$ are teal. Border of the isochrone is colored blue. $Covered_2$ has green color and $PartiallyCovered_2$ has red color. There are two holes in the $Covered_2$ polygon. One of them should be left in (the 36, 37, 38, 39 hole) and the other one should be removed. $PartiallyCovered'_2$ is the polygon between nodes 36, 37, 38 and 39. When we remove holes from the $Covered_2$ polygon and subtract $PartiallyCovered'_2$, we get the interior of the isochrone. All that remains then is to add the parts of the $PartiallyCovered_2$ which lie within the isochrone.

Chapter 4

Optimization of graph based search

In this chapter, we will show two approaches we took to try to reduce the amount of graph nodes searched during the process of finding an isochrone.

4.1 Partitioning

In the first chapter, we introduced a partitioning approach to route planning described by [8]. We selected one of the approaches introduced in that thesis and tried to adapt it to our problem. The approach was based on partitioning of the road graph.

4.1.1 Finding the partition

The authors of [8] examined two approaches to partitioning of the road graph – the splitting algorithm (section 3.4.1 of [8]), and the merging algorithm (section 3.4.2. of [8]). The merging algorithm gave better results for them, so we decided to use that approach.

The main idea of partitioning is dividing the nodes of the road graph into subsets. These subsets are called cells in [8]. Additionally, we want the graph induced by each of these subsets to be connected.

The merging algorithm starts by assigning each node its own cell. The algorithm then proceeds in steps. During each step, two candidate cells are selected and merged into

one cell. The candidate cells need to have an edge between them. After the merging occurs, a score is computed for the current partition. When it is no longer possible to merge any two cells (that is, when the cells contain all the connected components of the road graph), the partition with the best score is selected as the resulting partition.

The selection of candidate nodes is done by computing a merge priority for each two cells which are connected by an edge. Then the pair of cells with the highest merge priority is selected.

The merge priority function we used is the same as the function in [8].

$$p(C_1, C_2) = \frac{m_B(C_1, C_2) \cdot (1 + b(C_1) + b(C_2) + b(C_1 \cup C_2))}{n(C_1) \cdot n(C_2)} \cdot random(1, 1.01)$$

This equation shows how the priority between cells C_1 and C_2 is computed. $m_B(C_1, C_2)$ is the number of edges between cells C_1 and C_2, $b(C)$ is the number of boundary nodes of the cell C, and $n(C)$ is the number of nodes in the cell C.

The partition score function we used is different from the function used in [8]. They use the estimated number of evaluated edges when searching for the shortest path between two random points (see section 3.2 and Appendix A of [8] for more details). One of the features of this function is that it has no parameters to tweak.

However we wanted better control over the generated partition. Especially the size of the cells. So we used a different function, which assigns the score based on the sizes of the cells in the partition. For partition P and target size t we use

$$c(P) = \sum_{C \in P} \left(\frac{n(C)^2 + 1}{n(C) - 2} \right)^2$$

We came to this function by experimenting with different constants and forms of functions. This function tries to achieve relatively small variance of sizes of the cells.

4.1.2 Higher level graph

We then construct the higher level graph using the simplest approach described in [8]. The first step in constructing this graph is selecting edges which connect different cells in the partition.

Then we process each cell individually. Within a cell we find the border nodes, that is the nodes from which there is an edge leading outside of the cell. Then we create a clique of the border nodes with the cost of each edge being the cost of the shortest path between the two endpoints. See figure 4.1 for an example.

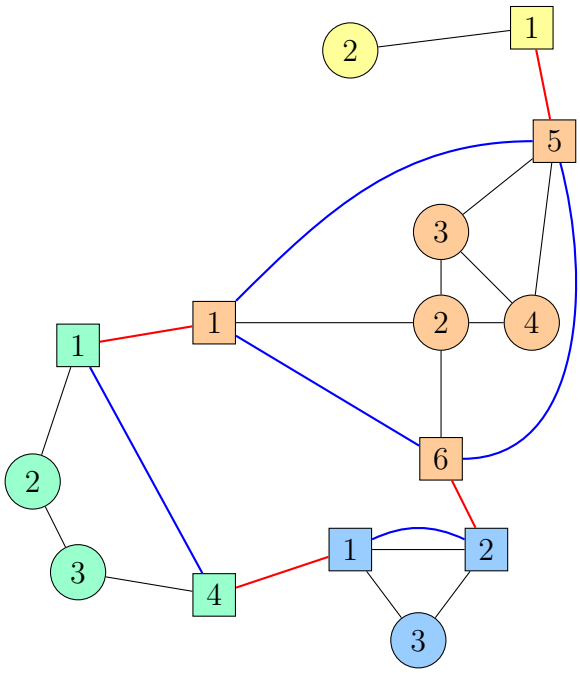

Figure 4.1: Example of creating the higher level graph. Original graph's edges are black. Cells are differentiated by background color of nodes. Red edges are the edges between cells. Border nodes are the squares. Blue edges are the shortcuts created by creating a clique between border edges of each cell. The higher level graph is made out of colored edges.

We will call the graph we got using this method the higher level graph G'.

4.1.3 Querying

In this section we will describe the algorithm for finding N_l using the higher level graph constructed in previous section. Again we are inspired by the techniques shown in [8].

In their thesis, they are using such higher level graph for finding the shortest path between two nodes of the road graph. For our need we need to alter this approach.

The approach to find N_l is very similar to the simple search approach from section 3.1.

We start at the starting point and perform a Dijkstra search. This search is limited only to a single cell. When the search in this cell is completed, the search is moved to the higher level graph.

The search in higher level works the same way as the search on the original graph. However we also need to descend back down to the original graph at some point. Otherwise we would not find nodes $n \in N_l \setminus N(G')$.

To determine when to descend to the original graph, we add another piece of information to each cell. We compute cell diameter as

$$diam(C) = \max_{\substack{n \in B(C) \\ o \in B(C) \\ n \neq o}} d'(n, o) + l(o),$$

where $B(C)$ are the border nodes of cell C, d' is the distance function in the higher level graph, and l is the limit function from section 3.3.

We descend to the original graph when the remaining time r_n at node $n \in N(G')$ is lower than the diameter of $C, n \in C$.

Theorem 1. *Query algorithm described in this section finds all nodes in N_l.*

Proof. The only problem are nodes $n \in N_l \setminus N(G')$. There is a cell C, such that $n \in C$. Let us look at the shortest path between the starting point, and n. This shortest path has to pass through some node $n' \in B(C)$. That means that $r_{n'} = r_n + d(n', n)$. But then

$$diam(C) \geq d(n', n) + l(n),$$

and because $l(n) > r_n$,

$$diam(C) > d(n', n) + r_n = r_{n'}.$$

That means that the query algorithm descended into original graph, and so n is found. □

4.2 Highway hierarchies

Another approach to finding shortest paths we tried to adapt, is highway hierarchies. This approach was studied in [24] and in [26].

Similarly to the partitioning approach it creates a higher level graph. The detailed procedures of constructing this higher level graph and using it for finding shortest path are described in [26]. In the next sections, we will briefly describe these procedures. We will also describe how we adapted this approach to find N_l.

4.2.1 Construction

The input parameter for the construction method is $\mathcal{N}_H$. This parameter determines how far two nodes need to be to be considered far enough.

We start by finding the neighborhood size of each node n. This is done by finding the set $Neigh'(n)$ of $\mathcal{N}_H$ closest (in terms of graph distance) nodes. The neighborhood size of n is

$$\mathcal{N}_n = \max_{o \in Neigh'(n)} d(n, o),$$

where $d(n, o)$ is the cost of the shortest path between n and o (the graph distance).

We then use this neighborhood size to find the neighborhood $Neigh(n)$ of n as

$$Neigh(n) = \{o \in N(G), d(n, o) \leq \mathcal{N}_n\}$$

After this is done, there are two steps required to create the higher level graph. In the first step, the important roads are identified. Important roads are those that lie on the canonical (see [26], for simplicity, imagine there is only one) shortest path between two nodes n_1, n_2, such that

$$n_1 \notin Neigh(n_2) \vee n_2 \notin Neigh(n_1)$$

After we have the important roads, we look at the graph G' induced by these road edges. As the second step, we take the 2-core of G'. This removes nodes of degree 2 and less. However, we add shortcuts from these nodes to the nodes of the 2-core. We also add shortcuts between nodes, which had a path between them made out of only nodes with degree 2. The second type of shortcuts is added back to the 2-core to create the higher level graph. See figure 4.2 for an example.

The cost of these shortcuts is always the cost of the shortest path between the two endpoints.

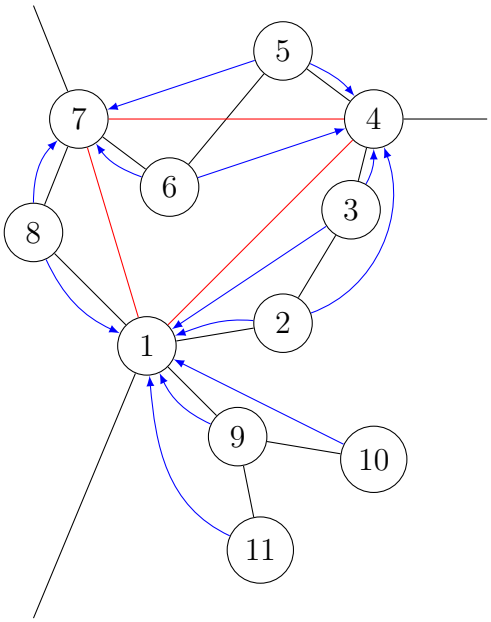

Figure 4.2: Example of creating the higher level graph from important roads. The black edges are the important road edges. Blue are the shortcuts from removed nodes. Red are the shortcuts that get added to the 2-core.

This allows for construction of a higher level road net graph, which has fewer edges than the original graph. This process is then iterated and a hierarchy of graphs is created. Different iterations may have different values for the $\mathcal{N}_H$ parameter.

However, as is noted in [26], the construction step takes a long time if there are edges with high cost relatively to the the rest of the edges in the road graph. They encounter such edges when dealing with ferries. In our case, the edges added in section 2.2.3 are such edges.

The way this is solved in [26] is by changing the technique of identifying the important edges. With this modification, it is no longer true that the highway hierarchies algorithm finds the shortest paths. We also used this approach, which led to problems as described in chapter 6.

4.2.2 Query

The query algorithm is based on the algorithm for shortest path queries in [26]. They are using a bi-directional search from the starting node, and the target node. There are

also cost-less edges added between two instances of the same node on different levels.

This way all the levels of the graph create one large graph. This graph is searched by a Dijkstra search with some limitations. The Dijkstra search is limited to the neighborhood of the last entrance node. Entrance nodes are either the starting nodes, the nodes through which the search advanced to a higher level, or nodes which were passed using a shortcut.

Because they are using bi-directional search, the authors of [26] don't need to descend back to lower levels of the hierarchy. However, when searching for an isochrone, we cannot use bi-directional search. Because of this, we need some piece of information about when to descend to the lower level.

For this purpose we introduce descend limits for each node. We use these in a similar way we used cell diameters in the partitioning approach. When we reach a node during the search where the remaining time is lower than the descend limit, we let the search descend to a lower level. Also, since shortcuts can only be traversed in one direction in the original search, we also define shortcut reverse limit. When we reach an endpoint of a shortcut with less time left than its reverse limit, we can traverse this shortcut in the opposite direction.

To find the descend limits for each node, we use the l function from section 3.3.

We need to simulate the backwards search through the road graph. We look at each node n of the original graph G and do a Dijkstra search limited to its neighborhood. For each encountered node n', which is also a node of the higher level graph G', we track the maximum of the value $l(n) + d(n, n')$, where $d(n, n')$ is the Dijkstra distance between n and n'. So the descend limit is defined as

$$Descend(n') = \max_{\substack{n \in N(G) \\ n' \in Neigh(n)}} l(n) + d(n', n).$$

We also need the limits for reversing the shortcuts. These are computed using the descend limits as

$$ShortcutRevLimit(n) = \max_{\substack{n' \in N(G') \\ (n', n) \in S}} Descend(n') + s(n', n),$$

where S is the set of shortcuts, and $s(n, n')$ is the cost of the (n', n) shortcut.

Lemma 3. *The query algorithm using two levels finds all reachable nodes n, for which $r_n < l(n)$, when starting from s with a time limit l.*

Proof. We use the assumption that the highway hierarchies query algorithm from [26] finds the shortest paths.

Let $n \in N(G)$ such that $r_n < l(n)$. Let us examine, what would the highway hierarchies algorithm do when searching for the shortest path between s and n.

If the search never left the lower level of the graph, it is basically only a Dijkstra search.

If the forward search advanced to the higher level, our search algorithm would advance to the higher level at the same node.

So the problem is the backward search. The only things that the backward search does differently are using shortcuts, and going to the higher level. When simulating the backward search with forward search, the shortcuts are taken in reverse. Also, the level switching, which happened only in the direction to the higher level, needs to be reversed (that is from higher level to lower level).

First we will show, that the backward search in respect to level switching is simulated correctly. Assume that n' is the node in which the backward search switched level.

Backward search is limited to the neighborhood of the n. That means, that

$$Descend(n') \geq l(n) + d(n', n).$$

Because up to now, our algorithm behaves in the same way as the shortest path algorithm, it is true that $r_{n'} = r_n + d(n', n)$. And because $r_n < l(n)$,

$$Descend(n') \geq l(n) + d(n', n) > r_n + d(n', n) = r_{n'}$$

And because $Descend(n') > r_{n'}$, we descend to the lower level in our algorithm.

Shortcuts work in a similar way. Assume the backward search took a shortcut $s = (n_1, n_2)$ with cost d_s. Because shortcuts are edges that start in nodes which are not the endpoint of any edge in the higher level graph, the only way the backward search could have reached n_1 is by switching level at n_1. That means that $Descend(n_1) > r_{n_1}$.

Also, because the shortest path algorithm finds the shortest path,

$$r_{n_2} = r_{n_1} + d_s.$$

That means that

$$ShortcutRevLim(n_2) \geq Descend(n_1) + d_s > r_{n_1} + d_s = r_{n_2}.$$

And from that follows, that our query algorithm will take the shortcut in the opposite direction.

□

This lemma can be inductively applied to show that this query algorithm works for multiple levels of the highway hierarchy. So it can be seen that the highway hierarchies algorithm finds all nodes of N_l.

Chapter 5

Implementation

In this chapter we will introduce some aspects of implementation of techniques described in previous chapters.

5.1 Map representation

To store the map data, we used the PostgreSQL[23] database engine with PostGIS[22] extension for storing geographical data. The data we used comes from OpenStreetMap [18]. We used Osmosis [19] to get it into the PostgreSQL database.

We represent the map using a table which stores all the nodes of the road graph. Each node has its position also saved in this table. Additionally there is a table which stores the adjacency relation of the road graph nodes. Each edge of the road graph has its cost, and the geometry that represents it, stored in this table.

Because the road graph may be too large to fit into memory, we use lazy loading of the road graph. For this reason, the nodes are partitioned into non overlapping regions. We used a least recently used cache of regions.

When asking for neighbors of a node, all nodes that are in the same region are loaded from the database into memory. Also when loading new region, and the amount of loaded regions exceeds a selected number, the least recently used region is removed from this cache.

Listing 5.1: The `road_areas` table

```
1 CREATE TABLE road_areas (
2     id bigint NOT NULL,
3     node_id bigint NOT NULL,
4     sequence_no integer NOT NULL,
5     cost_to_cover double precision NOT NULL
6 );
```

Because partition of the nodes from section 4.1 gives connected cells, these cells are good candidates for use as these regions. The size of the cells determines how often the database is queries.

Because we used the cells from the partitioning approach as regions in the lazy loading of the graph, we also save the diameters of the cells.

Additionally we saved information about faces of the road network. Faces are represented by a table `road_areas`. Listing 5.1 shows its schema. The `id` column stores the identification number of the face. The `node_id` column, and the `sequence_no` column determine the nodes which form the face. The `cost_to_cover` column stores the $l(n, F)$ value where n is the node with identification number `node_id`, and F is the face with identification number `id`.

5.2 Removing intersections

The reason for removing intersections was presented in section 2.2.2. Because the geometries representing road edges are stored in the database using PostGIS, we can use its geometry operations to find intersecting edges.

The PostGIS spatial indexes[4] allow for an efficient query plan, which uses these indexes to find candidate edges in logarithmic time. This allows for fast finding of intersecting edges.

The intersecting pairs of edges are processed one after another, and using the procedure from section 2.2.2, new edges are created. These then replace the intersecting edges in

the database.

5.3 Partitioning

The partitioning of the graph is important for us because we use it to determine which parts of the graph to load. The merging algorithm for partitioning from [8] is global. That is, it finds the candidate cells to merge globally for the whole road graph. In each step, it also has to evaluate the cost of the partition. Each step of the partitioning reduces the number of cells by one. Since computing the cost of the partition has at least linear complexity in the number of cells, the overall complexity of partitioning is at least quadratic in the number of nodes.

However, with the size of the road graph, this complexity is prohibitively expensive. For this reason, we needed to improve the running time of partitioning.

Even though the partitioning looks at the graph globally, the results are actually local. The algorithm does not gain much information by looking at the whole graph. Changes in the road network that are distant won't affect the local partitioning much.

This led to a different partitioning approach. First, we partition the road graph geographically into square pieces. Then we take each of these pieces, together with a buffer of surrounding pieces, and use the merging partitioning algorithm from [8] on the resulting part of the map. Then we look at the cells produced, and remove those that are not within the center piece. We then save the remaining cells. Then we repeat the same process with the next piece. However, this time we ignore the nodes which already are in some cell. Figures 5.1, and 5.2 demonstrate this approach.

It can be seen, that the quality of the resulting partition depends on the sizes of the resulting cells, and the size of the buffer. There is a trade-off between the performance of this approach and the quality of the resulting partition.

If the buffer is too large, the partitioning work gets repeated, because each step only keeps a small part of the partition. If the buffer is too small, the partition is not determined by the road network, but rather by the partition into squares.

If the road network is fairly regular, then the size of the graph which has to be par-

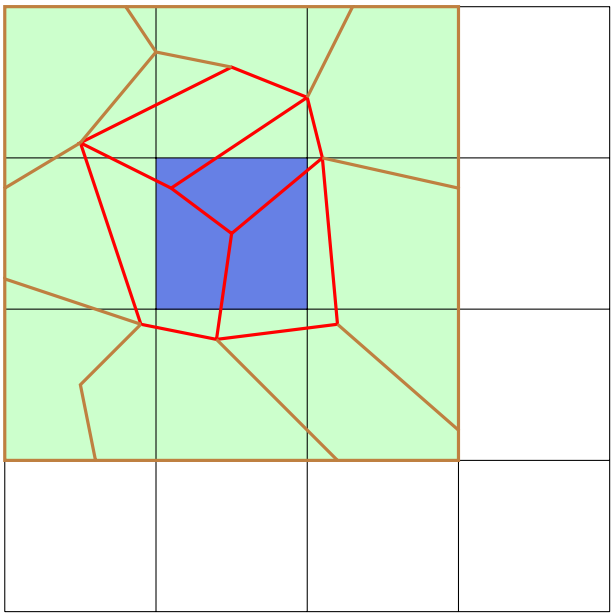

Figure 5.1: Our partitioning approach. The map is partitioned into squares. Then one square is selected (blue), with a buffer around it (green). This part of the map is partitioned. Cells that intersect the blue square (the red ones) are kept. Others (brown ones) are removed.

titioned in each step is bound. That means the work in each step can be bound by a constant, which leads to linear time complexity of partitioning when using this approach.

5.4 Connecting the graph

In section 2.2.1, we showed that having multiple connected components in the road graph leads to problems when searching for isochrone. Additionally, it prevents easy finding of faces of the road network (see section 5.5).

We use a simple approach to finding the connected components. Because the whole graph may be too large to fit into memory, and the partitioning algorithm creates cells which are connected, we use these cells for finding the connected components.

First we query the database for the cells obtained by partitioning the graph. Then we find edges which connect two different cells. After that, we use the common approach

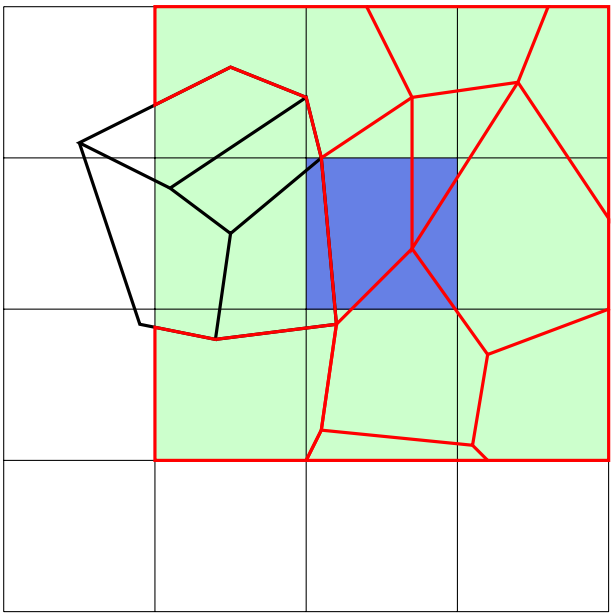

Figure 5.2: Second step of our partitioning approach (first step in figure 5.1). Next square with buffer is selected. This is then partitioned, while ignoring the parts of the map which were partitioned before (red is the new partition, black the partition from previous step).

to finding disjoint sets described in [9]. This gives us the sets of cells which constitute the connected components of the road graph.

When we have the connected components, we try to connect the one by one to the rest of the graph. We start with the smallest connected component and find the closest node which is not part of the component. To do this, we also use the spatial indexes of PostGIS. We then add an edge connecting the component to the rest of the graph. That may introduce some intersections (see figure 5.3). These can be removed using the approach in section 5.2.

5.5 Finding faces of the road network

Finding faces would be easy if we had a connected graph, where all edges are bi-directional. In section 5.4 we described a way to get a connected graph. Luckily making all edges bi-directional is also not a problem, because we can add reverse edges with cost equal to the time it takes to walk the edge in the opposite direction.

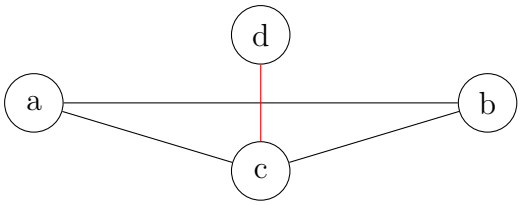

Figure 5.3: Connecting the graph creates an intersection. The closest node to d is node c. However, connecting d to c creates an intersection. This could be avoided if we did not connect to the closest node, but to closest edge.

This allows us to use a simple algorithm for finding the faces. We start at an arbitrary node, and traverse the graph in counter-clockwise direction. This way we traverse the border of a face. Because the graph is connected, there cannot be a hole in any face. Figure 5.4 demonstrates this approach.

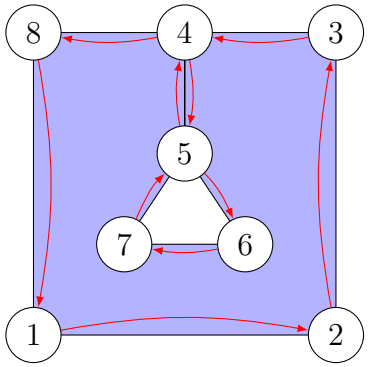

Figure 5.4: Face finding algorithm. Face is traversed counter-clockwise (red arrows).

The only problem with this approach is again the fact that we cannot store the list of processed nodes in memory because it may be too large. We again help ourselves with the partition. We process each cell of the partition individually. When processing a cell, each of its nodes is processed. While processing a cell, we remember a list of traversed edges. This allows us to find all faces which intersect the cell.

We also remember a list of processed cells. When traversing a face and we encounter a node from a processed cell, the search is aborted because the face was already found.

This approach also finds the outer unbounded area. Because this area causes problems in some of the subsequent steps, we identify and remove it. Because this outer area was

always much larger than any other area, it was easy to identify. A different approach to identifying it, is that it is the only area which has its nodes in clockwise direction.

5.6 Amending the graph

In section 2.2.3 we presented the need to create additional edges in the road graph.

5.6.1 Face triangulation method

We used an poly2tri [21] for triangulation of the faces. This is a library for constrained Delaunay triangulation. Constrained triangulation as a problem is studied in [5]. Basically it is a problem of finding a triangulation of a set of points, when some edges have to be in the resulting triangulation.

Finding a triangulation of a face can be solved by an algorithm solving the constrained Delaunaly triangulation. We used the poly2tri library to get a triangulation of the face.

The implementation of the rest of this approach followed the description in section 2.2.3.

5.6.2 Adding edges which are shorter than some constant

This approach is trivial to implement. We use the lazy loaded graph for the Dijkstra search. To compute d_M we use the great-circle distance[1].

The only problem is that the added edges may intersect. However, we already have an approach to remove these intersections.

[1] We use the great-circle distance for all distances.

5.7 Isochrone finding

In chapter 3, we divided the isochrone search into graph based search, and expanding from the found nodes.

The graph based search is just a Dijkstra search from the starting node cut off at some point. We implemented the Dijkstra's search algorithm with the possibility to stop the search prematurely.

The result of the isochrone algorithm is a polygon which represents the isochrone. Ideally this would not be a polygon, but rather a union of circles. However, the usual tools for spatial systems cannot really work well with unions of circles. For this reason we approximated circles with polygons.

To implement the techniques from section 3.2, we need to be able to be able to find a circle around a point with given radius. Because we are working with the surface of the Earth as the map surface, we need to use a projection to obtain this circle. The correct projection for this case is the equidistant azimuthal projection. This projection has the property that it preserves distances from a single point. This means that when we create a circle within the projection, it is still a circle in the map surface. This allows us to easily create a polygon which approximates this circle.

Another thing we need is to find the union the circles from figure 3.1. To approximate that, we used a convex hull. There are two cases. If the whole edge is covered, we used the convex hull of the circles centered at the endpoints. Otherwise, we used the convex hull of the circle from previous paragraph, and the farthest reachable point on the edge. See figures 5.5, and 5.6 for an example.

5.8 Graph based search optimization

The implementation of techniques from chapter 4 is rather straightforward. The highway hierarchies were implemented based on [26]. The implementation of the partitioning approach is also straightforward.

Because the construction of the highway hierarchies took too long, we also implemented

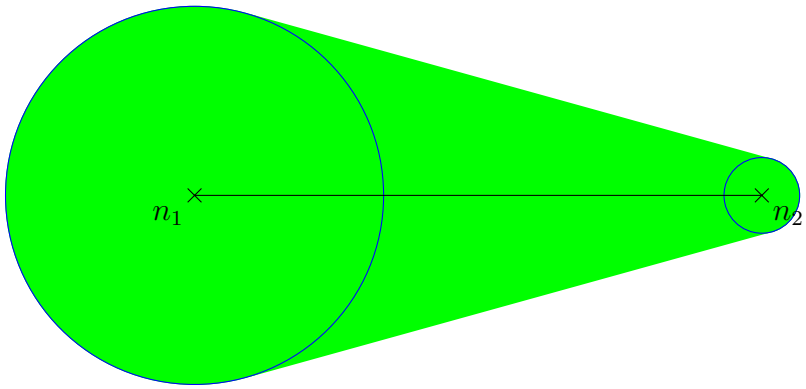

Figure 5.5: Convex hull used to approximate the union of circles when the whole edge is covered. The output is the convex hull of the two circles centered at n_1 and n_2.

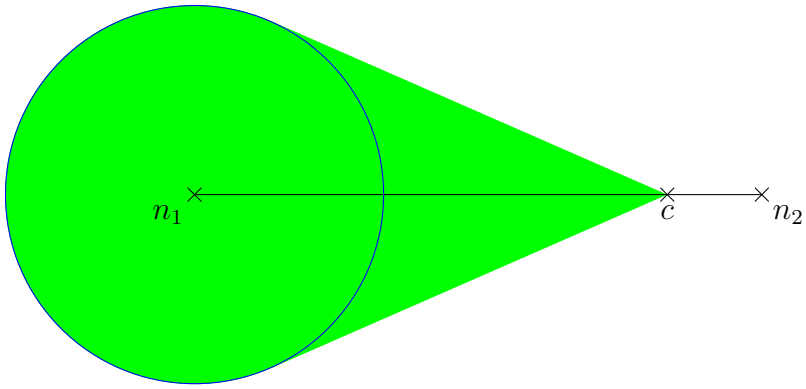

Figure 5.6: Convex hull used to approximate the union of circles when only a part of the edge is covered. The output is the convex hull of the circle centered at n_1 and the point c.

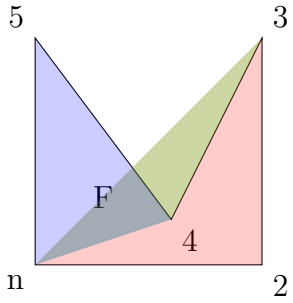

Figure 5.7: Covering of the face F by triangles from the node n. Triangles have different colors, the whole face is covered by three triangles: $(n, 2, 3), (n, 3, 4), (n, 4, 5)$.

the speedup technique from [26]. This led to highway hierarchies not being completely accurate.

The only thing remaining then is the l function from this chapter. In this section we will describe the construction of this function. To recapitulate, we want a function which when given a node n and a face $F \in F_n$ gives us the time required to reach any point within F from the node.

Let us look at nodes $n_1, n_2 \in N_F$, which have an edge between them. If we can reach every point in the triangle formed by n, n_1, and n_2 for each such pair of nodes, then we can reach the points of the whole face (see figure 5.7).

So now we need to find the time required to reach the whole triangle. We do this by looking at the edge between n_1, and n_2. We find the point on this edge with the longest road distance from n. Let that point be p, and the distance d. Then if t is the time required to reach the line between n and p, and $d_M(c(n), p) > d$ then

$$(t - d) + t = d_M(c(n), p).$$

From that

$$t = \frac{d_M(c(n), p) + d}{2}.$$

If $d_M(c(n), p) \leq d$, then

$$t = d_M(c(n), p).$$

Chapter 6

Results

In this chapter we will show the results of applying techniques described in previous chapters on real world data. As mentioned before, the data we worked with comes from OpenStreetMap. We tested on a map of Slovakia, or parts of it. We used the speeds of 80km/h for traveling on roads, and 6km/h for travelling on foot.

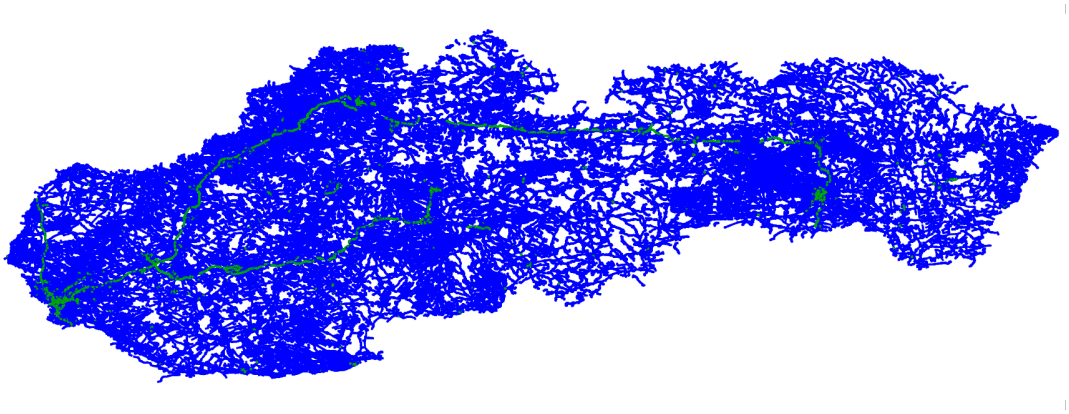

Figure 6.1: Map of Slovakia we used for testing.

6.1 Amending the road graph

In this section we will compare the two approaches to amending the road graph. First, figure 6.2 shows a part of the map before amending. Figure 6.3 shows the result of amending the graph using the triangulation technique, and figure 6.4 shows the result of adding edges shorter than 50 meters.

The part of the graph we used for this demonstration had 12712 edges. After using the triangulation technique, there were 14164 edges, and after using the other technique, there were 19018 edges. However, this technique also creates intersecting edges. After removing those using the technique from 2.2.2, there were 98202 edges.

Method	Number of edges
Original graph	12712
Triangulation	14164
Short edges	19018
Short edges without intersections	98202

Table 6.1: Numbers of edges for different methods of amending the graph.

We can see that the adding of short edges leads to a large number of added edges even for really short edges. For this reason we used the triangulation approach, even though the other approach can be better tuned to allow for more precise isochrones.

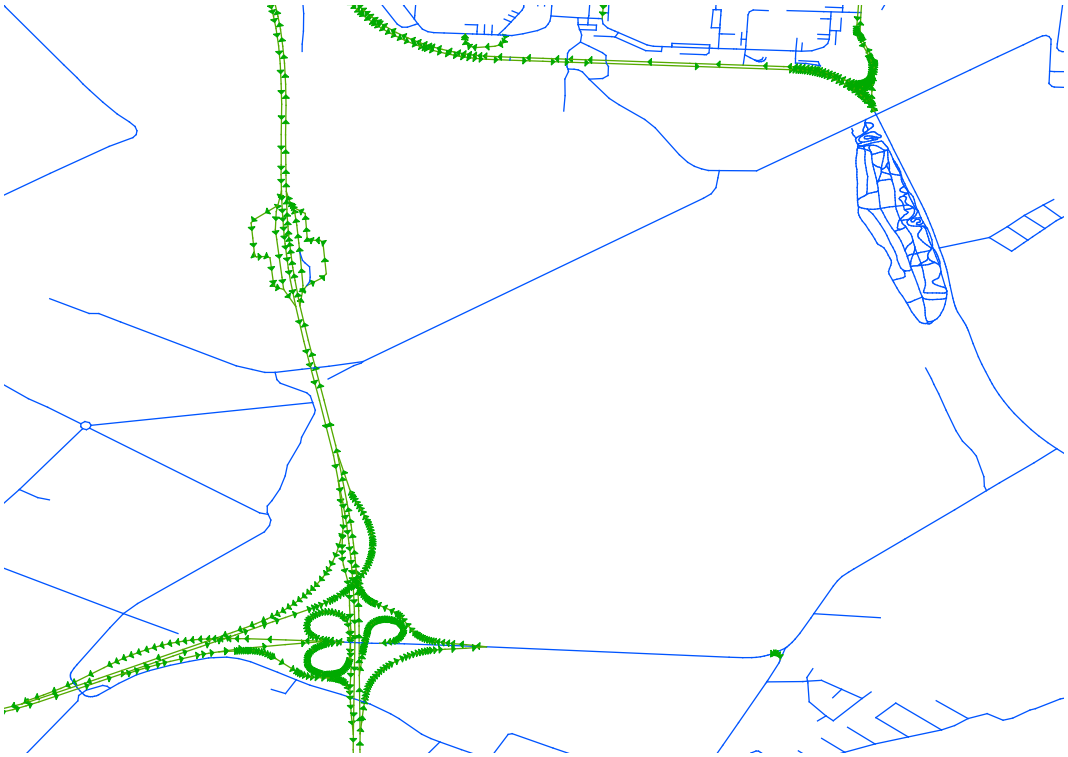

Figure 6.2: Part of the map before amending the road graph.

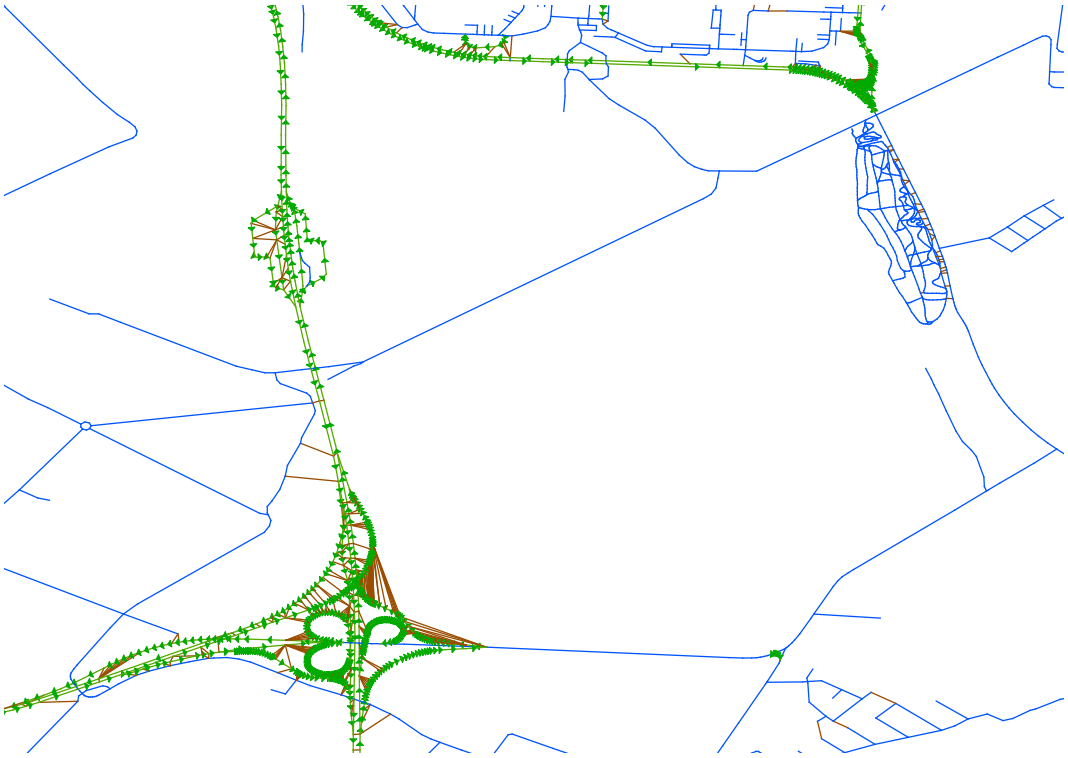

Figure 6.3: Amended graph using the triangulation technique. Brown edges were added.

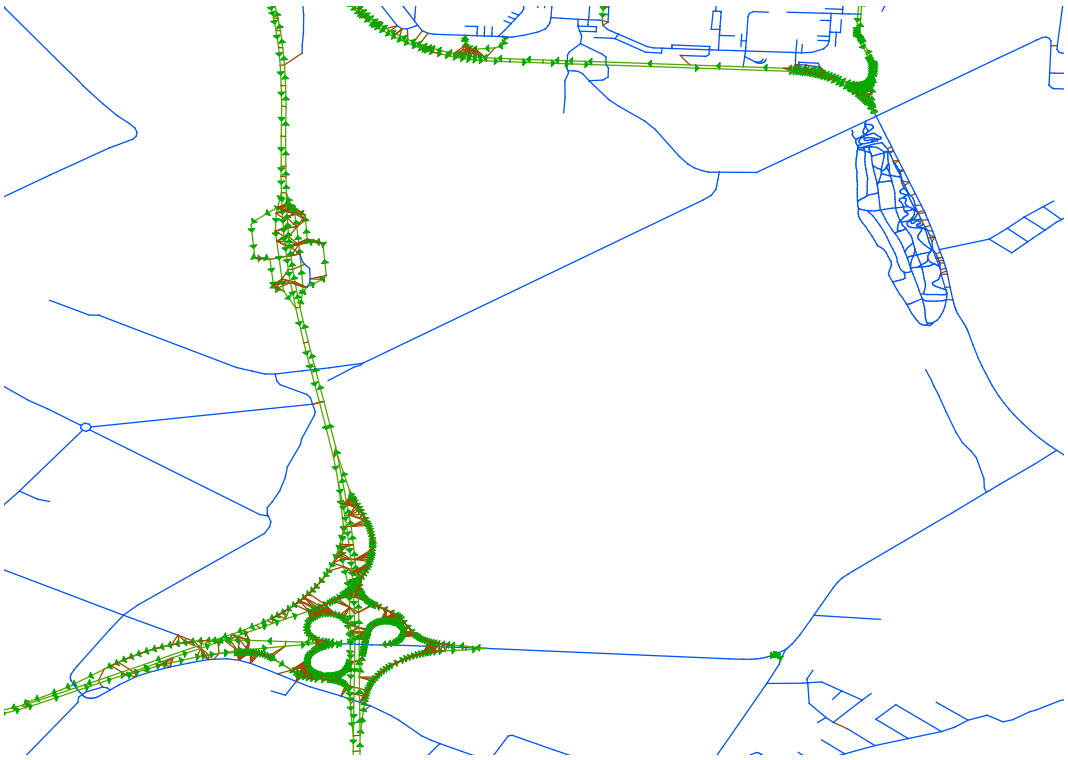

Figure 6.4: Amended graph using the short edges technique. Brown edges were added.

6.2 Dijkstra isochrone searching

In this section we will present the results of the basic Dijkstra algorithm for isochrone searching. Figure 6.5 shows the set of nodes within an isochrone, and figure 6.6 shows the complete isochrone.

We also present a larger example in figure 6.7.

Figure 6.8 compares the running times for different sizes of the isochrones. Note that this includes lazy loading of the graph from the database. We can see that the graph based search is shorter than computing the parts of the isochrone not lying on roads. This is because of time consuming geometric operations like convex hull computation, or geometry union.

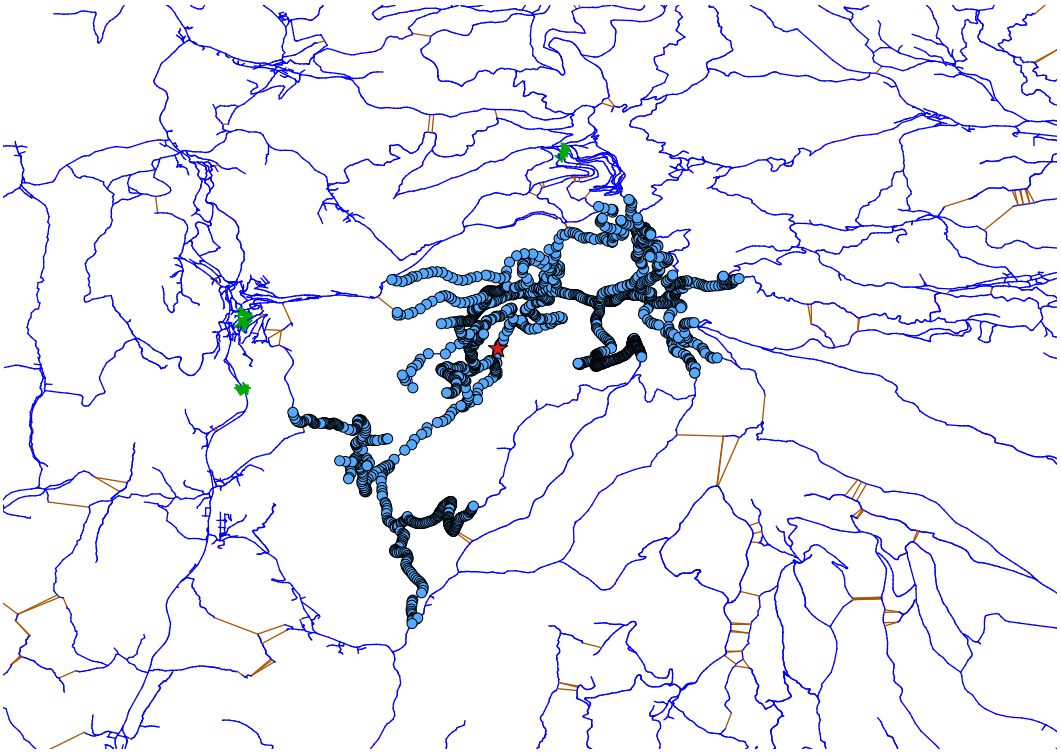

Figure 6.5: Nodes within the isochrone (blue circles) from starting point marked by a star, and a time limit of 6 minutes.

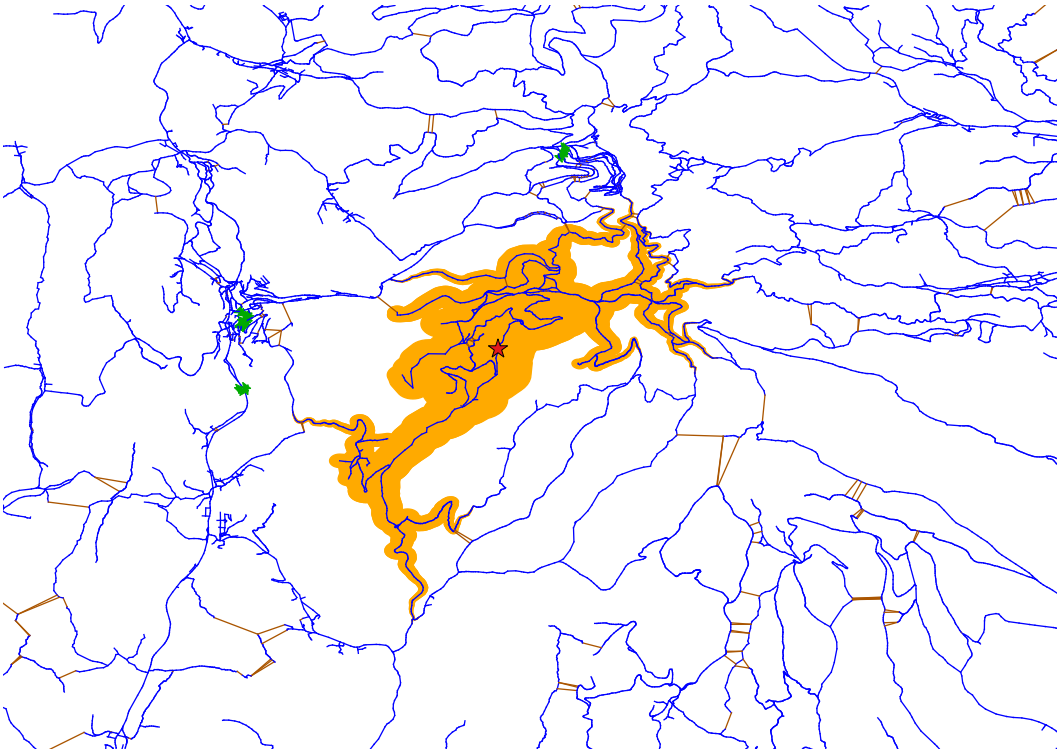

Figure 6.6: The isochrone (orange area) from starting point marked by a star, and a time limit of 6 minutes.

6.3 Graph search optimization

In this section we will present the results of applying the techniques from chapter 4 to our testing map. The results of this optimization can be only seen on rather large isochrones because average cost required to reach any point in a face from a node is 0.213 hours. This leads to the average value of the l function from chapter 4 being 0.576 hours.

For this reason we will present the results of the optimization on a 1 hour isochrone from figure 6.7. Figure 6.9 shows the set of road graph nodes which lie within this isochrone.

The set of nodes which are required to be in the output of an isochrone algorithm can be seen in figure 6.10. We can see that even for an isochrone of one hour, most of the nodes need to be processed.

Figure 6.11 shows the nodes within the isochrone computed by the partitioning approach from section 4.1. We can see that the nodes near the starting point are more

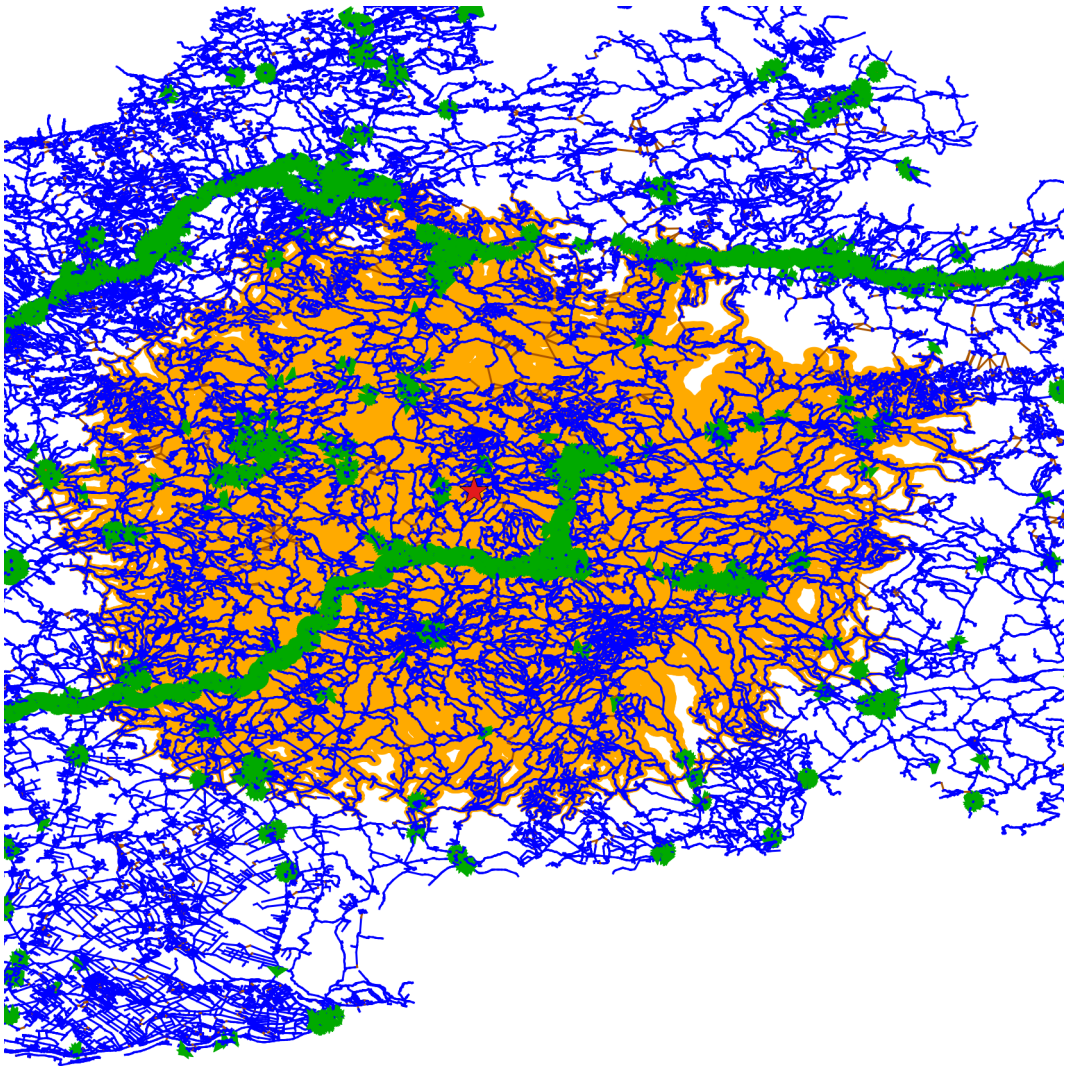

Figure 6.7: A larger isochrone (orange area), starting point marked by a star, one hour time limit.

sparse. This is the result of the partitioning approach skipping some nodes not required to find the isochrone.

Figure 6.12 shows the nodes within the isochrone computed by the highway hierarchies approach from section 4.2. Similarly to the partitioning approach, the nodes are more sparse near the starting node. However, with this approach, some required nodes are missed. This is because of the modification when constructing the highway hierarchy. A more detailed view on the missed nodes is in figure 6.13.

Finally, the figure 6.14 and table 6.2 show the comparison of the three approaches in terms of visited nodes for different sizes of the isochrone.

Time limit	Dijkstra	Partitioning	Highway hierarchies
0.0	1	1	1
0.1	1616	1616	1099
0.2	9671	9671	6699
0.3	31290	31288	23981
0.4	61289	61158	45961
0.5	97696	96019	70007
0.6	137343	133370	92089
0.7	178590	165452	113564
0.8	224972	192024	133819
0.9	281653	228947	160689
1.0	346498	274498	194036

Table 6.2: Visited nodes for different time limits and different methods.

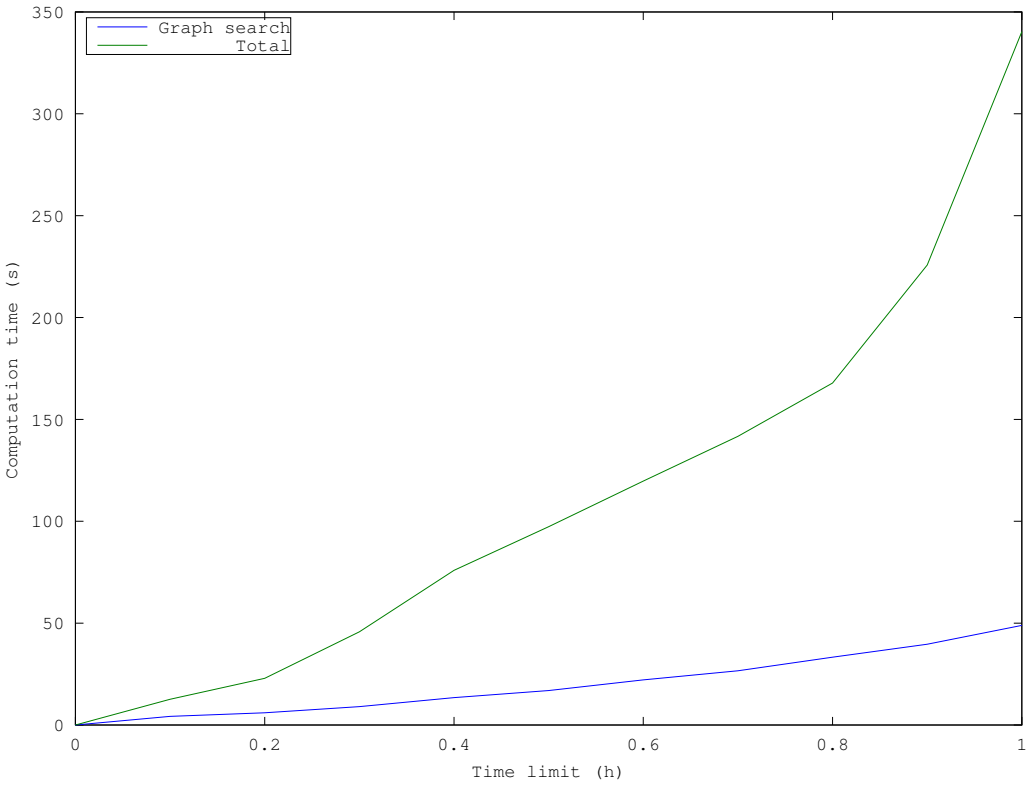

Figure 6.8: Computation times for different sizes of the isochrone.

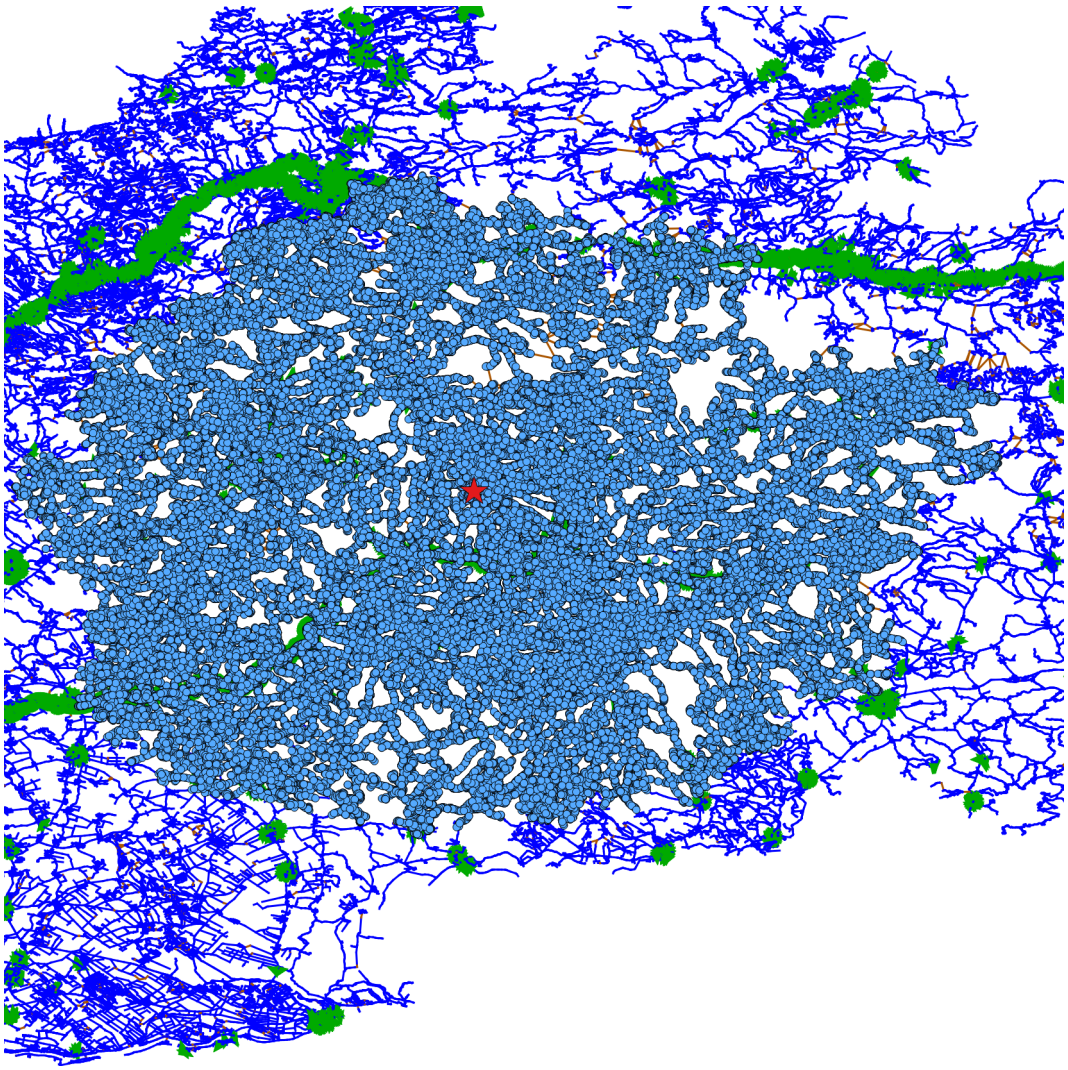

Figure 6.9: The nodes within the isochrone in figure 6.7.

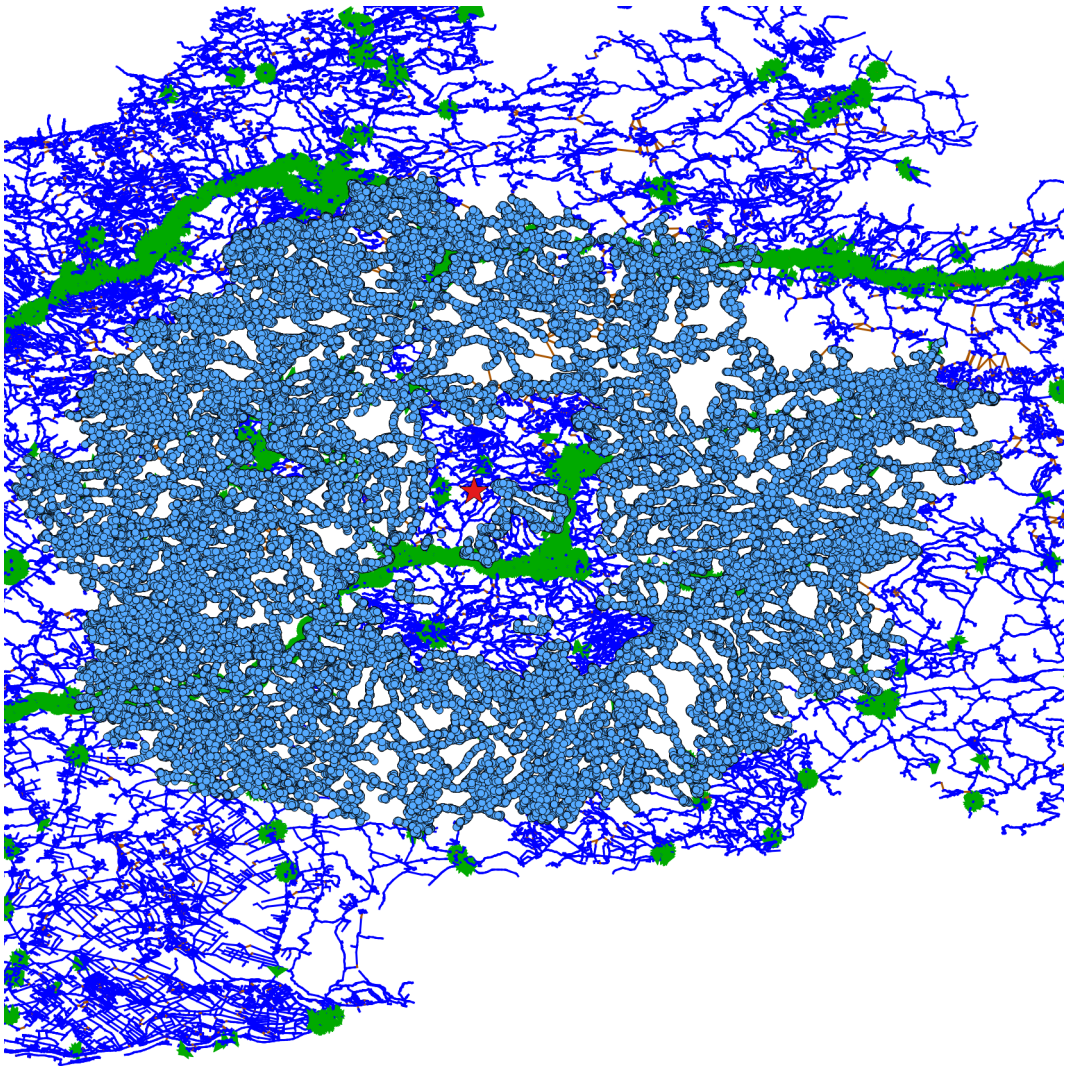

Figure 6.10: The nodes within isochrone from figure 6.7 for which $l(n) > r_n$.

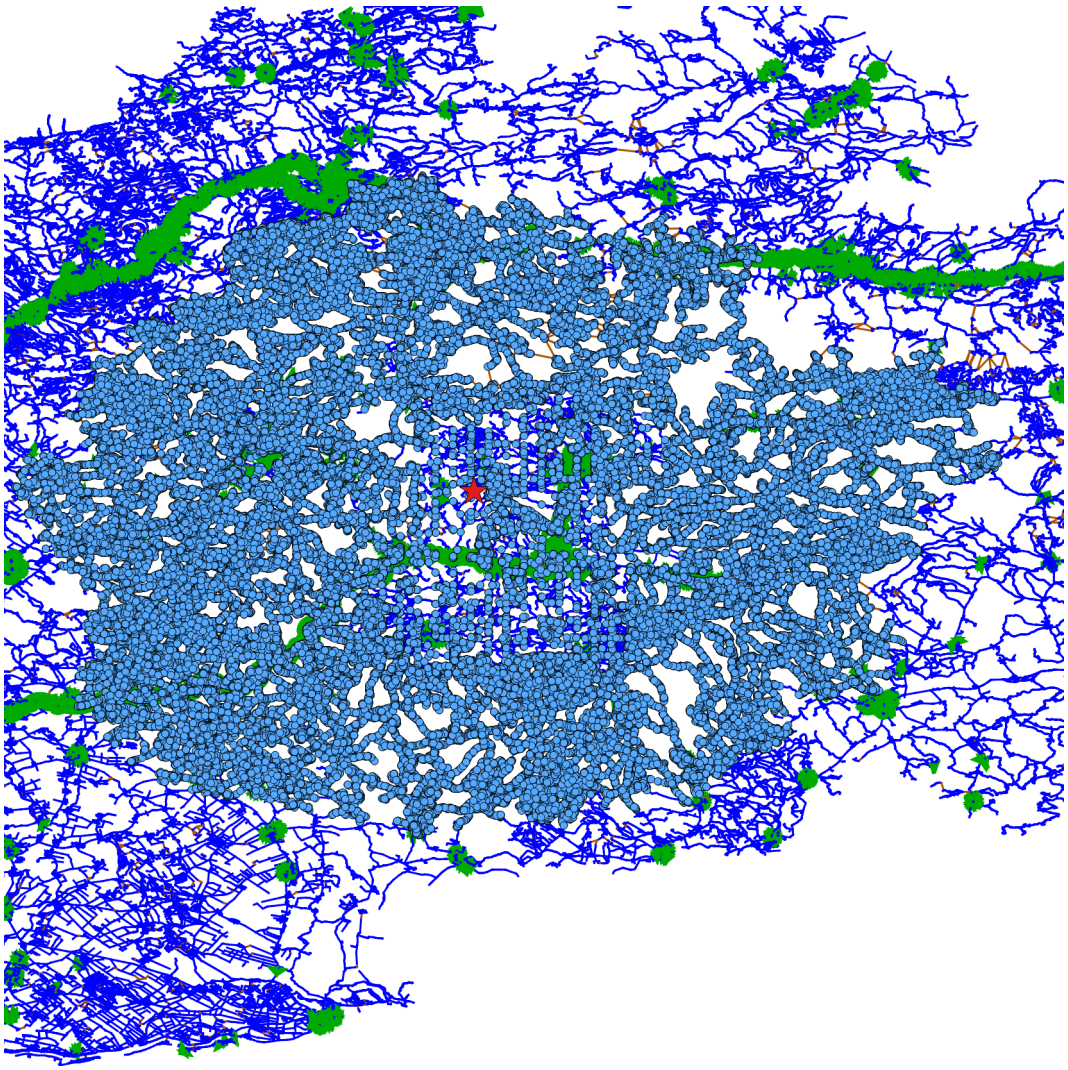

Figure 6.11: The nodes within isochrone found by the partitioning algorithm.

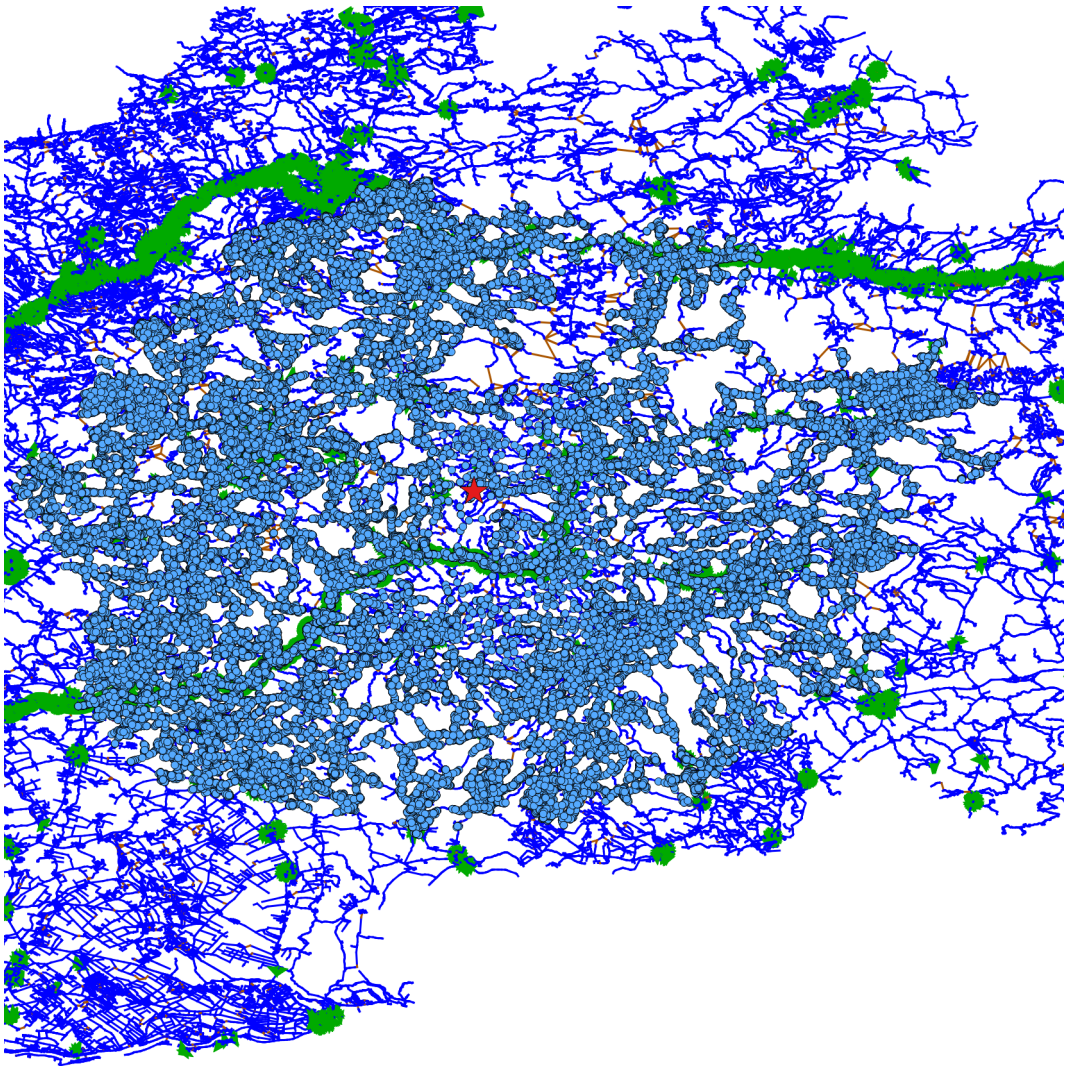

Figure 6.12: The nodes within isochrone found by the highway hierarchies algorithm.

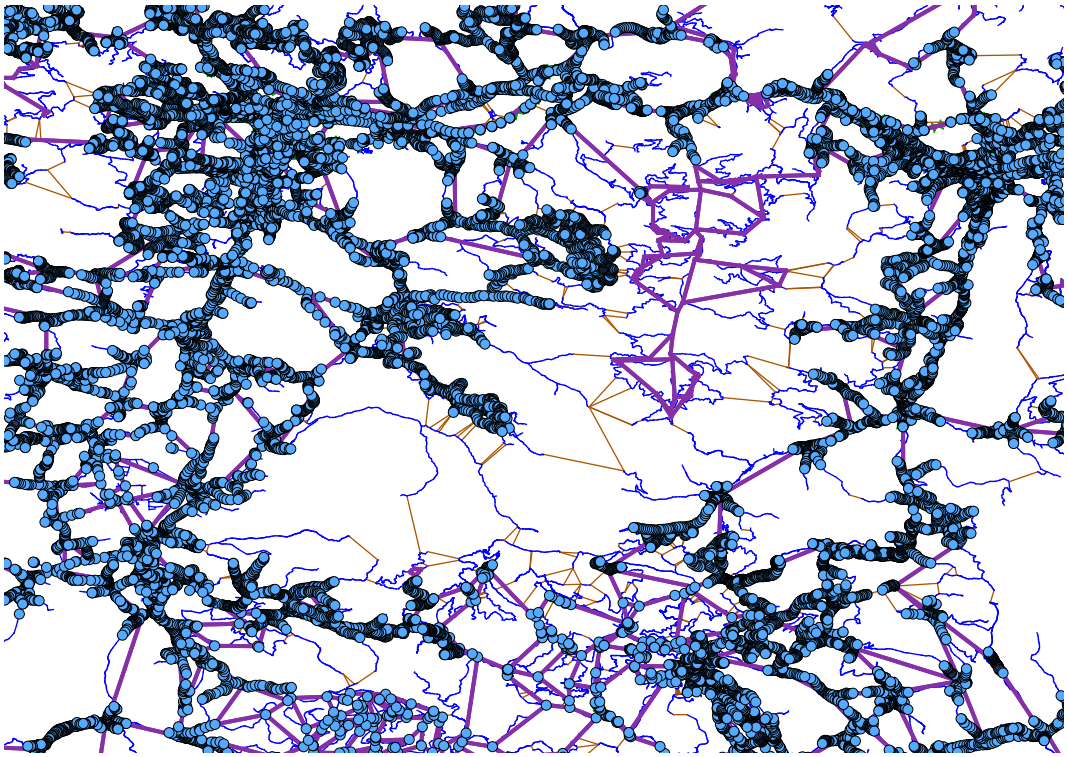

Figure 6.13: The nodes missed by the highway hierarchies algorithm. The higher level graph edges are purple. We can see that there are no edges going to the part of the road graph where the missing nodes are. This is because it is connected to the rest of the road graph using the added edges, which have relatively high cost.

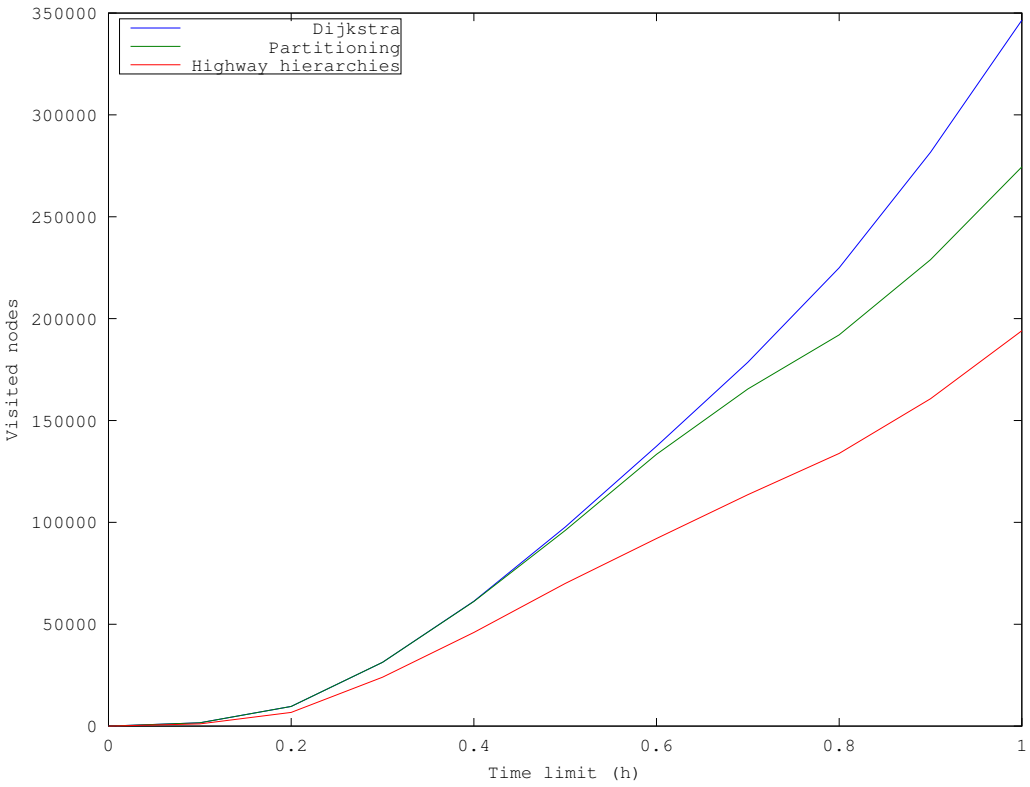

Figure 6.14: Plot of the number of nodes visited by different methods and different time limits.

Conclusion

In this thesis, we presented an approach to computing isochrones by separating the graph based search from the search for the rest of the points. We have also tried to reduce the amount of road graph nodes required to find the isochrone. Then we tried to adapt two approaches for route planning for finding isochrones.

We can see that the large difference between traversal speed along roads, and away from roads leads to only small reduction of the required nodes. This reduction may prove useful for large isochrones.

The partitioning approach to route planning was successfully adapted to finding isochrones. It allowed for lower number of nodes to be traversed to find the whole isochrone, however this was not a large reduction.

The highway hierarchies approach had larger problems. The precise construction of highway hierarchies is unfeasible because of edges added to allow for finding the isochrone in two steps. For this reason a speedup technique had to be implemented. However, this technique allows for the highway hierarchies route planning algorithm to not find the shortest paths. This leads to wrong computation of the isochrone.

Future work

In this thesis we did not take obstacles into consideration. There are many obstacles in real world maps (e.g. rivers, buildings, fences, etc.), and OpenStreetMap contains some of them. It may be possible to incorporate them into the isochrone search, thus making the isochrone a lot more precise.

Another thing we ignored are the different traversal speeds for different parts of the

terrain. For example, a person moves a lot slower through a dense forest than through a field. Some information about the type of terrain is also available from the OpenStreetMap. This information could also be used to create a more precise isochrone.

Also, the speed of the isochrone search is not very good. It may be possible to improve this by employing techniques from other route planning algorithms. However, for this to work, it will probably be required to find a way to identify the isochrone from a smaller amount of information.

Bibliography

[1] Holger Bast et al. "In Transit to Constant Time Shortest-Path Queries in Road Networks". In: *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithmics and Combinatorics*. Philadelphia, PA, USA: SIAM, 2007, pp. 46–59. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.143.7949%5C&%5C#38;rep=rep1%5C&%5C#38;type=pdf.

[2] Reinhard Bauer and Daniel Delling. "SHARC: Fast and robust unidirectional routing". In: *J. Exp. Algorithmics* 14 (Jan. 2010), 4:2.4–4:2.29. ISSN: 1084-6654. DOI: 10.1145/1498698.1537599. URL: http://doi.acm.org/10.1145/1498698.1537599.

[3] Veronika Bauer et al. "Computing isochrones in multi-modal, schedule-based transport networks". In: *Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems*. GIS '08. Irvine, California: ACM, 2008, 78:1–78:2. ISBN: 978-1-60558-323-5. DOI: 10.1145/1463434.1463524. URL: http://doi.acm.org/10.1145/1463434.1463524.

[4] *Boundless: Introduction to PostGIS: Section 14: Spatial Indexing*. URL: http://workshops.boundlessgeo.com/postgis-intro/indexing.html (visited on 04/20/2014).

[5] L. P. Chew. "Constrained Delaunay Triangulations". In: *Proceedings of the Third Annual Symposium on Computational Geometry*. SCG '87. Waterloo, Ontario, Canada: ACM, 1987, pp. 215–222. ISBN: 0-89791-231-4. DOI: 10.1145/41958.41981. URL: http://doi.acm.org/10.1145/41958.41981.

[6] Daniel Delling et al. “Algorithmics of Large and Complex Networks”. In: ed. by Jürgen Lerner, Dorothea Wagner, and Katharina A. Zweig. Berlin, Heidelberg: Springer-Verlag, 2009. Chap. Engineering Route Planning Algorithms, pp. 117–139. ISBN: 978-3-642-02093-3. DOI: 10.1007/978-3-642-02094-0_7. URL: http://dx.doi.org/10.1007/978-3-642-02094-0_7.

[7] E. W. Dijkstra. “A Note on Two Problems in Connexion with Graphs”. In: *NUMERISCHE MATHEMATIK* 1.1 (1959), pp. 269–271.

[8] Ingrid C. M. Flinsenberg. “Route Planning Algorithms for Car Navigation”. PhD thesis. Technische Universiteit Eindhoven, 2004.

[9] Bernard A. Galler and Michael J. Fisher. “An Improved Equivalence Algorithm”. In: *Commun. ACM* 7.5 (May 1964), pp. 301–303. ISSN: 0001-0782. DOI: 10.1145/364099.364331. URL: http://doi.acm.org/10.1145/364099.364331.

[10] Johann Gamper, Michael Böhlen, and Markus Innerebner. “Scalable Computation of Isochrones with Network Expiration”. In: *Proceedings of the 24th International Conference on Scientific and Statistical Database Management*. SSDBM’12. Chania, Crete, Greece: Springer-Verlag, 2012, pp. 526–543. ISBN: 978-3-642-31234-2. DOI: 10.1007/978-3-642-31235-9_35. URL: http://dx.doi.org/10.1007/978-3-642-31235-9_35.

[11] Johann Gamper et al. “Defining isochrones in multimodal spatial networks”. In: *Proceedings of the 20th ACM international conference on Information and knowledge management*. CIKM ’11. Glasgow, Scotland, UK: ACM, 2011, pp. 2381–2384. ISBN: 978-1-4503-0717-8. DOI: 10.1145/2063576.2063972. URL: http://doi.acm.org/10.1145/2063576.2063972.

[12] Robert Geisberger et al. “Contraction hierarchies: faster and simpler hierarchical routing in road networks”. In: *Proceedings of the 7th international conference on Experimental algorithms*. WEA’08. Provincetown, MA, USA: Springer-Verlag, 2008, pp. 319–333. ISBN: 3-540-68548-0, 978-3-540-68548-7. URL: http://dl.acm.org/citation.cfm?id=1788888.1788912.

[13] Michael T. Goodrich. “Planar separators and parallel polygon triangulation (preliminary version)”. In: *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*. STOC ’92. Victoria, British Columbia, Canada: ACM,

1992, pp. 507–516. ISBN: 0-89791-511-9. DOI: 10.1145/129712.129762. URL: http://doi.acm.org/10.1145/129712.129762.

[14] Stefan Hertel and Kurt Mehlhorn. "Fast Triangulation of Simple Polygons". In: *Proceedings of the 1983 International FCT-Conference on Fundamentals of Computation Theory*. London, UK, UK: Springer-Verlag, 1983, pp. 207–218. ISBN: 3-540-12689-9. URL: http://dl.acm.org/citation.cfm?id=647891.739588.

[15] Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. "Acceleration of shortest path and constrained shortest path computation". In: *Proceedings of the 4th international conference on Experimental and Efficient Algorithms*. WEA'05. Santorini Island, Greece: Springer-Verlag, 2005, pp. 126–138. ISBN: 3-540-25920-1, 978-3-540-25920-6. DOI: 10.1007/11427186_13. URL: http://dx.doi.org/10.1007/11427186_13.

[16] Mohammad Kolahdouzan and Cyrus Shahabi. "Voronoi-based K nearest neighbor search for spatial network databases". In: *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*. VLDB '04. Toronto, Canada: VLDB Endowment, 2004, pp. 840–851. ISBN: 0-12-088469-0. URL: http://dl.acm.org/citation.cfm?id=1316689.1316762.

[17] Sarunas Marciuska and Johann Gamper. "Determining objects within isochrones in spatial network databases". In: *Proceedings of the 14th east European conference on Advances in databases and information systems*. ADBIS'10. Novi Sad, Serbia: Springer-Verlag, 2010, pp. 392–405. ISBN: 3-642-15575-8, 978-3-642-15575-8. URL: http://dl.acm.org/citation.cfm?id=1885872.1885904.

[18] *OpenStreetMap*. URL: www.openstreetmap.org/ (visited on 04/19/2014).

[19] *Osmosis*. URL: http://wiki.openstreetmap.org/wiki/Osmosis (visited on 04/19/2014).

[20] Dimitris Papadias et al. "Query processing in spatial network databases". In: *Proceedings of the 29th international conference on Very large data bases - Volume 29*. VLDB '03. Berlin, Germany: VLDB Endowment, 2003, pp. 802–813. ISBN: 0-12-722442-4. URL: http://dl.acm.org/citation.cfm?id=1315451.1315520.

[21] *poly2tri - A 2D constrained Delaunay triangulation library*. URL: http://code.google.com/p/poly2tri/ (visited on 04/20/2014).

[22] *PostGIS*. URL: http://postgis.net/ (visited on 04/19/2014).

[23] *PostgreSQL*. URL: http://www.postgresql.org/ (visited on 04/19/2014).

[24] Peter Sanders and Dominik Schultes. "Highway hierarchies hasten exact shortest path queries". In: *Proceedings of the 13th annual European conference on Algorithms*. ESA'05. Palma de Mallorca, Spain: Springer-Verlag, 2005, pp. 568–579. ISBN: 3-540-29118-0, 978-3-540-29118-3. DOI: 10.1007/11561071_51. URL: http://dx.doi.org/10.1007/11561071_51.

[25] Simon Scheider and Werner Kuhn. "Road Networks and Their Incomplete Representation by Network Data Models". In: *Proceedings of the 5th International Conference on Geographic Information Science*. GIScience '08. Park City, UT, USA: Springer-Verlag, 2008, pp. 290–307. ISBN: 978-3-540-87472-0. DOI: 10.1007/978-3-540-87473-7_19. URL: http://dx.doi.org/10.1007/978-3-540-87473-7_19.

[26] Dominik Schultes. "Fast and Exact Shortest Path Queries Using Highway Hierarchies". MA thesis. Universität des Saarlandes, 2005.