Karlsruhe Institute of Technology

Realistic Pedestrian Routing

Bachelor Thesis of

Simeon Danailov Andreev

At the Department of Informatics
Institute of Theoretical Computer Science

Reviewers:	Prof. Dr. Dorothea Wagner
	Prof. Dr. Peter Sanders
Advisors:	Dr. Martin Nöllenburg
	Dipl.-Inform. Julian Dibbelt
	Dipl.-Inform. Thomas Pajor

Time Period: 17th August 2012 – 16th November 2012

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Karlsruhe, 16th November 2012

Abstract

Pedestrian routing receives more and more attention, as devices capable of route planning are nowadays increasingly available. A problem arises – road network databases generally lack explicit information about street walkways. This bachelor thesis shows how the walkways can be computed based on such a database. A solution is also designed to avoid large traffic junctions, as pedestrians generally do. An additional problem are areas where pedestrians can walk freely, for instance plazas, squares and parks. This problem is solved by computing visibility graphs and adding special handling for parks – areas where footpaths are preferred to walking across. Experimental evaluations are then shown for all solutions.

Deutsche Zusammenfassung

Mit der steigenden Verfügbarkeit von Geräten, die routen können, wird die Routenplanung für Fußgänger noch aktueller. Dabei entsteht das Problem, dass die meisten in Datenbanken verfügbaren Straßennetze keine (bzw. nicht alle) expliziten Fußwege am Straßenrand modellieren. Dieses Problem wird in dieser Bachelor-Thesis gelöst, indem solche Fußwege automatisch berechnet werden. Außerdem wird ein Mechanismus entwickelt, um große Straßenkreuzungen beim Routen zu vermeiden. Ein weiteres Problem sind die Bereiche, wo die Fußgänger frei laufen können – auf Plätzen oder in Parks. Für die Plätze wie eine Piazza werden Sichtbarkeitsgraphen berechnet. Die Parkanlagen werden spezifisch behandelt, da dort die Fußgänger normalerweise die Gehwege vorziehen. Die vorgestellten Lösungen werden in der Arbeit experimentell evaluiert.

Contents

1. Introduction

With the raising availability of devices capable of route planning, such as smart phones, pedestrian and bicycle routing is more and more feasible. Route planners for vehicles nowadays are capable of avoiding traffic jams, warning of speed limits and much more. Route planning when travelling on foot or with a bicycle however has received less attention, as few would buy a standalone system for this task alone. Routing software on a mobile device on the other hand is much more adequate and the problems of pedestrian and bicycle routing can be explored without fear of having no field of application.

The goal of this bachelor thesis is to address and find solutions to two problems that arise with route planning for pedestrians.

First, pedestrian walkways are generally not modeled in non-commercial road network databases. So far, existing route planners for pedestrians simply use the street as where the pedestrian should walk. No indication is given which side of the street a pedestrian should walk on and where streets should be crossed when this is needed. Additionally, when computing a fastest way between two points, traffic junctions are not taken into consideration. A pedestrian may be forced to wait long periods of time in order to pass crossings and other quicker ways (such as bridges) may be found when large junctions are avoided. In this thesis a mechanisms is devised to automatically create walkways around streets and hinder the usage of traffic intersections.

The second problem is how areas where pedestrians can walk freely (parks, plazas, etc.) are handled. This is not a problem in vehicle routing – such areas for vehicles are mapped with lanes that indicate how the area can be traversed. For pedestrians however a mechanism is needed that will guide the routing through the area, so that the route remains optimal. This problem is similar to movement planning for robots[AG92] and its solution in this thesis is the usage of visibility graphs[GM91]. The definition of visibility graphs and their application to solve this second problem can be found in the following chapters.

Related work

Due to its many practical applications, route planning is a widely researched area in the field of informatics. As the topic of this thesis is also route planning in one of its many forms, an array of publications with related content exist. The thesis can be decomposed in four general problems – walkway generation, crossing areas, actual routing and consideration of traffic junctions. Sources and publications are listed for each of those problems.

Automatic creation of walkways based on a road network database is a rarely reviewed topic. In [BPS11] the authors show how walkways can be generated based mainly

on building layout. While the results are good, building layout is not necessarily part of road network databases. Considering the building layout however can be used to improve the approach in this thesis – which is based solely on street layout. On the other hand, multiple publications show how road networks can be extracted via satellite photos [TW98, Pet03, GM04, MZ07]. While adaptation may be possible in order to extract walkways, this is not the aim of this thesis.

Crossing open spaces is a common problem in many fields, especially in robotics. Thus a multitude of publications exist, that describe how this problem can be solved [OIRK87, AG92, MAN04]. The common approach is the usage of visibility graphs. The problem of computing a visibility graph for a set of n obstacles can be solved in $O(E + n \log n)$, where E is the number of edges in the visibility graph [GM91].

Exhaustive works exist covering shortest paths algorithms and speed-up techniques [Dij59, CGR96, DSSW09, ADGW11, DGW11].

Time spent waiting for traffic lights to turn green is why traffic junctions should be considered. A modeling of traffic lights and computation of minimum time paths can be found in [AOPS02]. This approach requires the knowledge of traffic lights as well as their switching periods. It also changes the costs in the routing network from static to dynamic, as arrival time at traffic lights dictates how much time is spent waiting. In order to not complicate matters further, a simple heuristic is used in place of such a model. It is also worth mentioning that traffic junction are modeled in context of vehicle routing [HB07, Faw00]. Thus the models are mostly impractical for pedestrians – for example their waiting time is normally independent of the number of waiting pedestrians.

In addition, the modeling done in this thesis relies heavily on geometrical computation. Information about algorithms such as intersections of line segments, yet more about visibility graphs and some used data structures can be found in [dBCvKO08].

Thesis organization

The thesis is organized in following manner:
Chapter 2 gives definitions to any used terms.
Chapter 3 deals with any computations that are done as preparation for the actual routing, i.e. the modeling.
Chapter 4 explains how the route is computed.
Chapter 5 gives evaluations as to the amount of extra information and extra computational cost that are needed to support the given solutions when routing.
Chapter 6 offers a summary of what the thesis achieves, what improvements are possible and what further problems could be reviewed when routing for pedestrians.

2. Preliminaries

Throughout this bachelor thesis several common terms and several terms specific to the thesis are used. This chapter gives definitions to those terms as well as explanations to their meaning and their significance for the thesis.

2.1 Common terms

The necessary common terms such as *graph* and *quadtree* are now defined. Those terms are common in the field of informatics and no alteration from their standard definitions are needed.

Graphs

Graphs are a commonly used concept in the field of informatics. A *graph* G is defined as the tuple (V, E). The set V contains the *vertices* $\{v_1, v_2, ..., v_n\}$, which are also commonly named *nodes*. The set $E \subseteq V \times V$ represents the *edges* that connect those vertices.

Some important types of graphs for this thesis are *simple graphs*, *multigraphs*, *directed graphs* and *undirected graphs*.

In a *simple graph* any edge $(u, v) \in E$ may appear only once in E, i.e. only one edge may connect two vertices. In addition, no edge $(u, u), u \in V$, is allowed in E. Such an edge is also called a *loop*. In a *multigraph*, E is a multiset, i.e. E allows any number of edges between any two vertices of the graph, including loop edges.

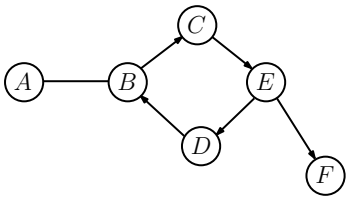
In an *undirected graph* an edge $(u, v) \in E$ implies that v is also connected to u, i.e. $(v, u) \in E$. In contrast to undirected graphs, a *directed graph* can have an edge (u, v) in E, without having the reverse edge (v, u) in E.

In this thesis a *directed simple graph* is used to determine where a pedestrian can walk. Edges represent streets, walkways, etc. Each vertex has assigned geographic coordinates (*longitude* and *latitude*), so each edge is equivalent to a walkable physical stretch.

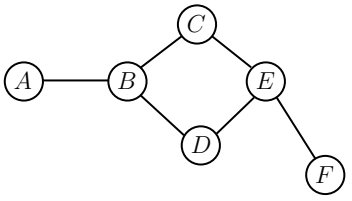
Figure 2.1 shows the sets E for different types of graphs with a common set of nodes $V := \{A, B, C, D, E, F\}$.

Node degree

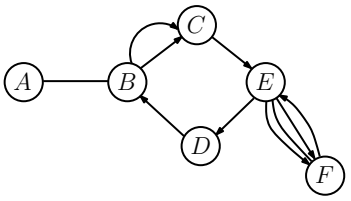
Each node $v \in V$ in a graph $G = (V, E)$ has a *node degree*, denoted by $\deg v$. The degree of a node v is the number of nodes connected with an edge to v, where v may be either the source of the edge or its target. Formally, $\deg v := |\{u \in V \mid (u, v) \in E \vee (v, u) \in$



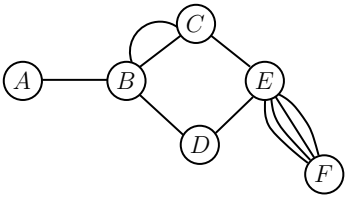
(1)
$E := \{(A,B),(B,A),(B,C),$
$(C,E),(D,B),(E,D),(E,F)\}$



(2)
$E := \{(A,B),(B,A),(B,C),(C,B),(C,E),$
$(E,C),(D,B),(B,D),(D,E),(E,D),$
$(E,F),(F,E)\}$



(3)
$E := \{(A,B),(B,A),(B,C),(E,F),$
$(C,E),(D,B),(E,D),(E,F),$
$(B,C),(E,F),(E,F),(F,E)\}$



(4)
$E := \{(A,B),(B,A),(B,C),(C,B),(C,E),$
$(E,C),(D,B),(B,D),(D,E),(E,D),$
$(E,F),(F,E),(B,C),(C,B),(E,F),$
$(E,F),(E,F),(F,E),(F,E)\}$

Figure 2.1: A directed simple graph **(1)**, an undirected simple graph **(2)**, a directed multigraph **(3)** and an undirected multigraph **(4)**.

$E\}|$.

Additionally, the incoming degree of a node v is defined as $\operatorname{indeg} v := |\{u \in V \mid (u,v) \in E\}|$, i.e. the number of ingoing edges. The outgoing degree of a node v is defined as $\operatorname{outdeg} v := |\{u \in V \mid (v,u) \in E\}|$, i.e. the number of outgoing edges. Note that $\deg v = \operatorname{indeg} v + \operatorname{outdeg} v$ is not necesarrily true.

The node degree is needed to determine whether a node is an *intersection node* or it lies on a *intersection-free path*. Intersection nodes and intersection-free paths are defined in Chapter 2.2.

Planar graphs

A graph G is said to be *planar* if it can be drawn in a plane without intersecting edges, which are drawn as plane curves. Planar graphs are important when computing the faces of a graph.

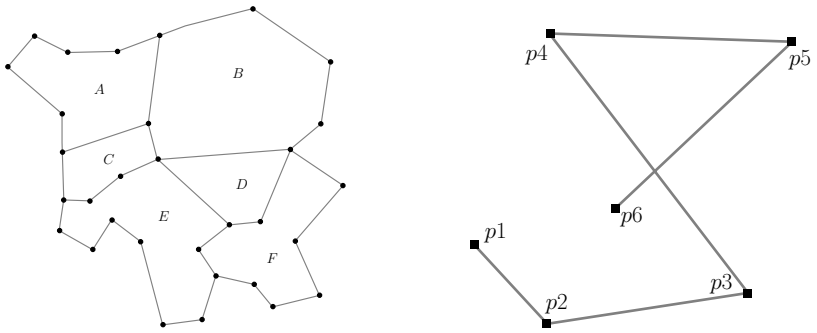
Face

The *faces* of a *planar embedded graph* G (G's nodes have assigned coordinates in the plane) are defined as the smallest regions bounded by edges. Figure 2.2a shows a planar graph drawn on a plane. The graph's faces are denoted by letters from A to F. A face of a planar graph G is also often named a *facet of* G.

Faces are important when computing the areas between street lanes (*street polygons*, Chapter 2.2), where no walkways should be placed.

Polyline

A *polyline* is a set of several connected line segments. For this thesis, each point on a polyline may belong to a maximum of two line segments of the polyline. A *closed polyline* is a polyline with the same start and end point. Polylines that are not closed are paths.



(a) A planar graph and its faces A, B, C, D, E, F.

(b) A polyline with its points drawn as squares.

Figure 2.2: A planar graph and a polyline.

Figure 2.2b shows a polyline composed of five lines, where the endpoints of those lines are depicted as squares.

Polylines are important for the creation of the walkways.

Polygon

A *polygon* in a plane is a figure bounded by a closed polyline. The line segments of its bounding polyline are generally called edges and the points of the polyline are called vertices. Two important types are *simple polygons* and *not necesarrily simple polygons*. A *simple polygon* has no inner holes, where *not necesarrily simple polygons* may have inner holes.

In this thesis free areas and parks are described by not necesarrily simple polygons, as they may contain inaccessible areas such as lakes or fountains. In addition, the inner area of streets is described by simple polygons.

Quadtree

A *quadtree*[dBCvKO08] is a data structure that stores geometric figures in the plane, i.e. 2D shapes such as circles, polygons, line segments and points. The main function of the quadtree is to locate all stored figures in a specified area. Ideally, such queries should be run in $O(\log n + m)$, where n is the number of stored figures and m is the number of returned figures. The quadtree also defines procedures that add and remove figures to and from the tree, ideally in $O(\log n)$.

The structure of the quadtree is a tree. Each node of the tree defines a bounding box (i.e. rectangle) and is either a node with exactly 4 children or a leaf containing figures that are all within the bounding box. In other words, the leaf's figures have common area with the leaf's bounding box. A leaf containing more than a specified number of figures generates 4 other leaves and becomes their parent. The new children leaves partition the bounding box of their parent into 4 new bounding boxes and the figures of the parent are redistributed in the leaves. Queries to the quadtree then locate all leaves with a bounding box within a given area. For this, all nodes with a bounding box that shares common area with the area are visited.

Quadtrees are extensively used throughout this thesis in order to efficiently locate figures that share common area with a given figure. Additional information on quadtrees can be found in [dBCvKO08].

Planarization

In different parts of the modeling nodes must be placed where specific edges intersect, if no such nodes exist. Additionally, when computing the inner areas of wide streets we are

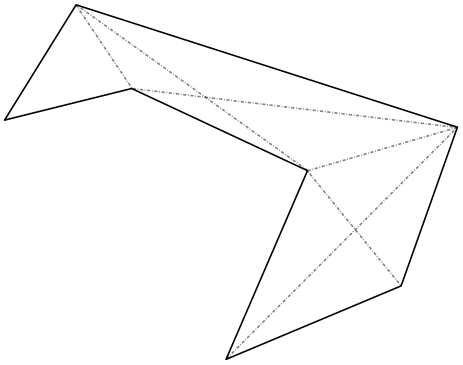


Figure 2.3: The visibility graph of a polygon.

interested in areas closed by street edges. If nodes exist at every intersection of edges, such areas are easily described by facets. So a simple procedure is used to add nodes where edges cross, effectivly making the graph planar (thus *planarization*). This procedure is now described.

All intersections between edges of the graph are computed. For each such intersection we check, whether a node exists on the coordinate of the intersection. If no such node exists, we add a new node to the graph with the same coordinate as the intersection. The existing or added node is denoted by s. Each edge (u, v) involved in this intersection, with $u \neq s$ and $v \neq s$, is deleted and the edges (u, s) and (s, v) are added to the graph. Those two edges have the same properties as the original edge (u, v) (same layer, category and so on).

In order to find the intersections fast, a quadtree is used. For each edge in the graph $e \in E$ nearby edges are located by using the quadtree. Those nearby edges are then tested for intersections with e. Edges that are added/deleted to/from the graph are also added/deleted to/from the quadtree. An improvement for the planarization method would be to use the line-sweep algorithm to compute all intersections in linear time. This improvement is not used as the described method is fast enough.

In order to utilize the planarization procedure further, an additional set of edges $B \subseteq E$ is passed as an argument. Intersections between edges in B will be ignored during the procedure. Another argument indicates whether different layers of the edges should be respected. If not, intersections between edges in different layers are ignored.

The time complexity of the planarization is bounded by $\Theta(|E'| \operatorname{QTQuery} |E'|)$, where $\operatorname{QTQuery} n$ stands for a query to a quadtree with n elements and E' is the set of edges in the graph after the planarization. The size of E' is bounded by $O(\sum_{i \in I} \deg i + |E|)$. Here E is the set of edges before the planarization and I is the set of intersections that will be processed. The number of edges that cross at an intersection $i \in I$ is denoted by $\deg i$.

Visibility graph

The *visibility graph*[GM91] of a polygon P is defined as $G_P := (V_P, E_P)$, where V_P is the set of vertices of the polygon P. The set of edges is defined as $E_P := \{(u, v) \in V_P \times V_P \mid u \neq v$ and the line segment between u and v is completely within $P\}$. Two nodes connected with an edge $e \in E_P$ are said to be visible to each other. Figure 2.3 shows the visibility graph of a polygon. The vertices that are visible to each other are connected with dotted lines.

Visibility graphs are important for the processing of the free areas. A visibility graph is computed for each free area, so that a route can pass through the free area, instead of going around it.

2.2 Thesis specific terms

The terms defined in this chapter are specific to this bachelor thesis. They are needed to understand the problems of the thesis and their solutions, as well as to name specific places and objects of interest, such as areas that can be traversed freely and streets that have walkways.

Intersection node

In this thesis an *intersection node* is any node with node degree greater than 2. The set of all intersection nodes is denoted by $V_\text{i} := V_\text{intersections} := \{v \in V \mid \deg v > 2\}$. Intersection nodes are important when creating the walkways around wider streets.

Intersection-free path

An *intersection-free path* represents a path of nodes P connecting two intersection nodes. The path P starts with an intersection node $s \in V_\text{i}$ and ends with an intersection node $t \in V_\text{i}$. In between s and t any number of nodes with degree 2 may exist, however s and t are the only intersection nodes on P.

Formally, $P := \{s, v_1, v_2, ...v_m, t\}$, where $s, t \in V_\text{i}, m \in \mathbb{N}_0$ and $\forall i \in \{1, ..., m\} : deg(v_i) = 2$. In addition, the set E must contain the edges (v_i, v_{i+1}), where $i \in \{0, ..., m+1\}$, $v_0 := s$ and $v_{m+1} := t$.

The set of intersection-free paths is denoted by $E_\text{i} := E_\text{intersections}$ and is also important for the creation of the walkways.

Edge layer

In this thesis each edge has a *layer* value. Edges with higher layer value are physically above edges with lower layer value. E.g. a bridge passing over a street has a higher layer value than the street.

Layers are important when connecting the created walkways to the already existing routing network. If layers are not respected, the created walkways will also be connected to bridges passing over the street and tunnels passing under them.

Splittable edge

A *splittable edge* is an edge representing a street of a category which has walkways for pedestrians. Such edges will be "split" in two walkways to both sides of the represented street. As already mentioned, walkways for most streets are not explicitly modeled routing databases. Thus the need for this "splitting". Edges of the following street categories are considered splittable:

- A roads. This category marks major roads. Such roads usually connect cities, ports and airports.
- B roads. Such roads connect smaller cities and towns.
- C roads. Major streets in cities are often marked as such.

Several additional categories are used for the routing, however edges of those categories are not considered splittable. Such *non-splittable* edges denote streets traversable without the usage of walkways. Living streets, footpaths, walkways, cycleways, etc. fall in those categories.

$E_\text{s} := E_\text{splittable} := \{e \in E \mid e \text{ is splittable}\}$ denotes the set of all splittable edges. $V_\text{s} := V_\text{splittable} := \{v \in V \mid (v, u) \in E_\text{s} \vee (u, v) \in E_\text{s}\}$ denotes the vertex set induced by E_s. Both sets are needed when creating the walkways (Chapter 3.1).

Street polygon

A *street polygon* represents the inner area of a wider street. Those polygons are used to prevent walkways being placed on streets with many lanes. The street polygons can be seen as purple polygons in some figures throughout the thesis. A more detailed explanation for their calculation and purpose can be found in Chapter 3.1.1.

Free area

Free areas are places where one can walk freely. Only pedestrian areas such as squares and piazzas are considered free areas. For every free area additional edges will be added during the modeling, so that it can be traversed as one would traverse a square – walking directly through the square instead of going around it.

In this thesis areas such as parks, forests, gardens, etc. are considered to be traversable only by using the existing footpaths, so no additional edges are added for them.

Park

While it may be obvious what a park is, a *park* in this thesis denotes the free areas which will not be traversed freely. Such areas are expected to have footpaths that should be used, instead of simply walking through the area. When starting at a location in a park however, reaching the nearby footpaths is done in a more realistic fashion (Chapter 4.4).

3. Modeling

In this chapter the modeling of the routing graph is described. This modeling has the following goals: Creating walkways around wide streets, as road network databases generally have no information about pedestrian walkways. Adding extra edges to allow routing through, instead of around, free areas. Finding the facets in parks, so that a route starting or ending in a park can find its way (in a realistic fashion) out of the facet where the start/destination is.

The modeling achieves these goals in a number of steps:

1. The routing graph is extracted from the input data. As this step is specific to the input data, it is discussed in Chapter 5.1 (along with other implementation details).
2. The positions of the walkways are computed. Nodes and edges corresponding to the walkways are added to the graph. This includes the connection of the walkways to the routing graph.
3. Overlapping and touching free areas are merged. Visibility graphs are then computed for each free area and the visibility edges are added to the routing graph. Two types of such edges are distinguished here. Edges that are needed to completely pass a free area and edges that are only needed when starting or ending in a specific free area. The first type will always be considered while routing. The second type will only be used when starting or ending in the free area the edges are in.
4. Overlapping and touching parks are merged. All facets composed of edges in parks are then computed.

The modeling is done in a preprocessing phase and so is also referred to as preprocessing.

3.1 Creating the walkways

A problem when creating the walkways is that different lanes of the same street are generally modeled as different nodes and edges in the road network databases. Figure 3.1 shows an example of a street with many lanes. Simply adding the walkways for each edge will put walkways in the inner area of streets. As already mentioned in Chapter 2.2, the street polygons are used to avoid this. The first two parts of this section describe how these street polygons are computed.

The next three sections describe the actual computation of where walkways should be, if every edge was an actual street. Walkways that are inside street polygons are then removed, resulting in the desired output.

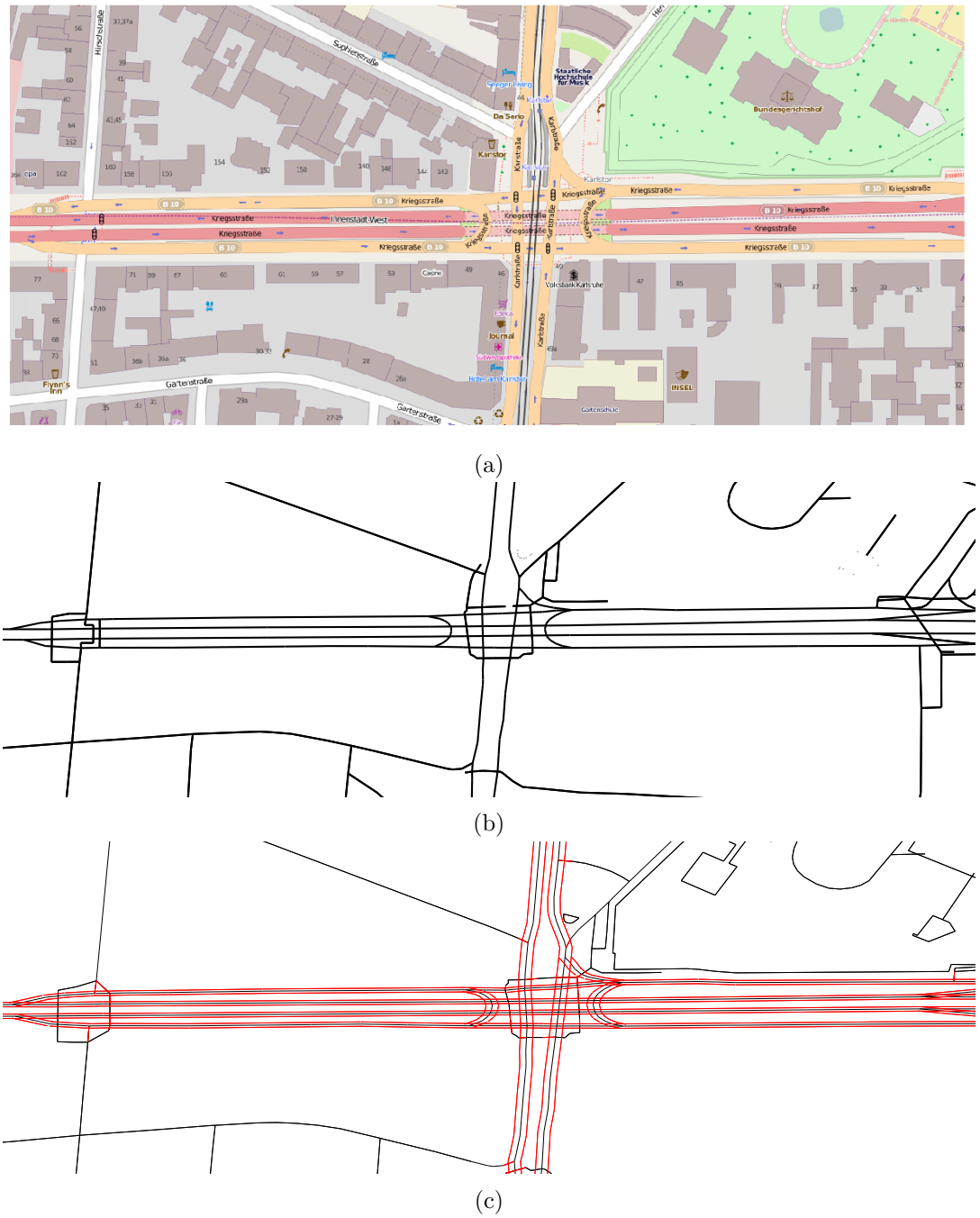


Figure 3.1: A street with many lanes **(a)**, its representation in the input data **(b)** and the computed walkways (when assuming that each edge is a part of a single street) colored in red **(c)**.

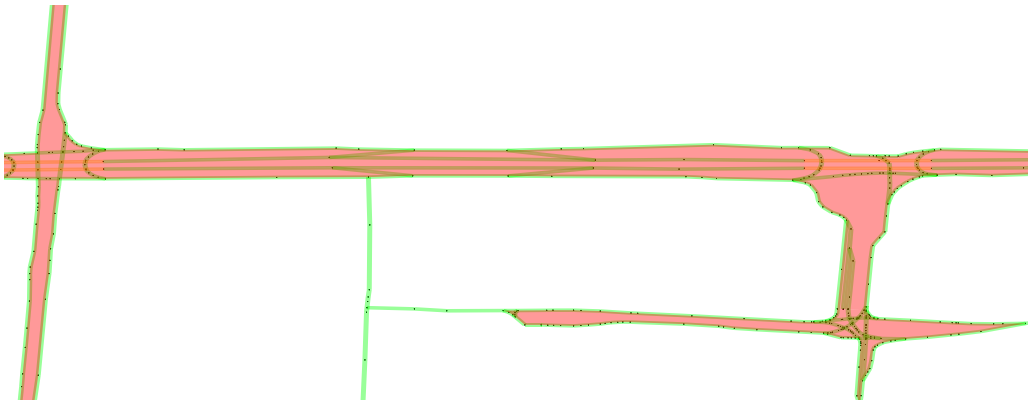


Figure 3.2: Street polygons (colored red) between different lanes of streets and the actual streets (colored green).

3.1.1 Computing the street polygons

To compute the street polygons, a graph is composed of all splittable edges, denoted by $G_\mathrm{s} := G_\mathrm{splittable} := (V_\mathrm{s}, E_\mathrm{s})$. This graph is then converted into a planar graph G_s^p using the general planarization procedure (Section 2.1). The set of faces F of G_s^p are then computed. This can be done by any procedure that computes faces of a planar graph. The method used in this thesis traverses the planar graph in a *Depth-First-Search* manner and attempts to close one face at a time. For this, the next edge that is visited by the *Depth-First-Search* is always a non-visited edge with the smallest angle to the previously visited edge.

The set F is then filtered to keep only *thin* polygons and polygons covering *small areas*. Polygons with an area less than $1000m^2$ are considered to be small places surrounded by streets (pedestrian islands, intersections, etc.) where no walkways should exist. *Thin* polygons are long and narrow polygons, places where streets run parallel for a long distance. Figure 3.2 shows some thin polygons and some small polygons that are colored red.

A polygon P is considered thin if ratio P is between a lower bound and an upper bound:

$$\text{ratio}\, P := \frac{area P_{[cm^2]}}{\text{perimeter}\, P_{[cm]}} \in [0, 3170]$$

So a face P of the graph G_s^p is considered a street polygon, iff:

$$\text{ratio}\, P \in [0, 3170] \vee \text{area}\, P_{[m^2]} \in [0, 1000]$$

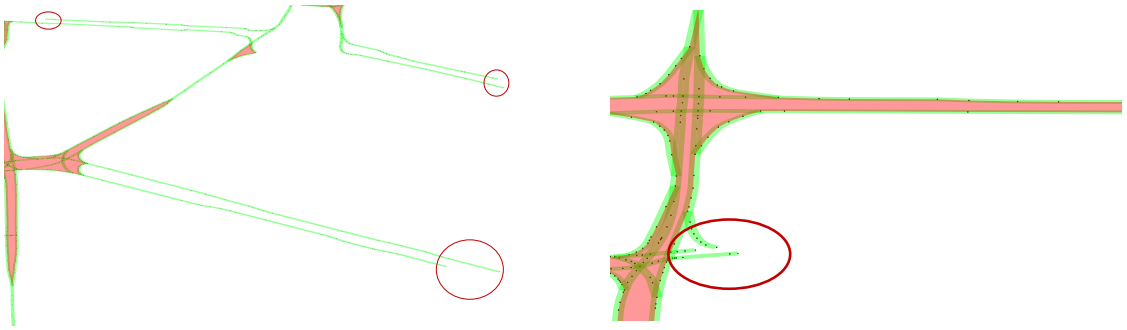
The bounds for ratio P and the area of P were determined empirically. Further criteria could be used to detect faces that are street polygons, however this proved to be unnecessary.

All street polygons define an area where no walkways should be created. Edges created by the splitting process (Section 3.1.4) are removed if they are in this forbidden area.

The time complexity needed to compute the street polygons is bounded by $\Theta(|E_\mathrm{s}| + \text{planarization}\, G_\mathrm{s})$.

3.1.2 Tying stubs

Due to the used bounding box and places where the type of the street changes from splittable to non-splittable, stubs occur in the graph (Figure 3.3). Many of those stubs run parallel, or are near parts of the graph. The relation of those stubs to the rest of the



(a) (b)

Figure 3.3: Stubs that should be connected to the graph in order to close a street polygon (surrounded by red circles).

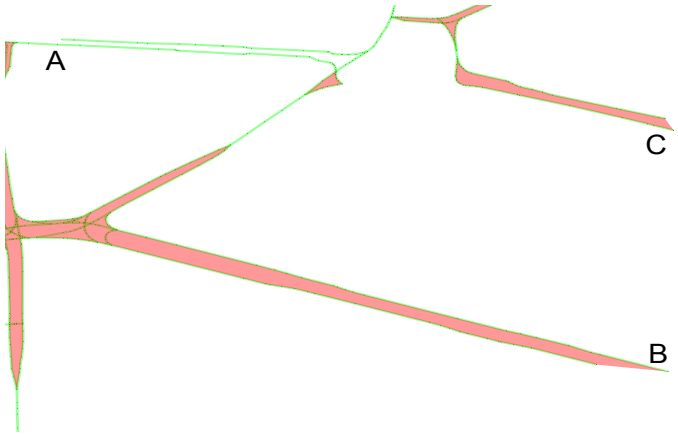


Figure 3.4: The first approach when tying stubs, situations that are handled well (B and C) and a situation that is not handled well (A).

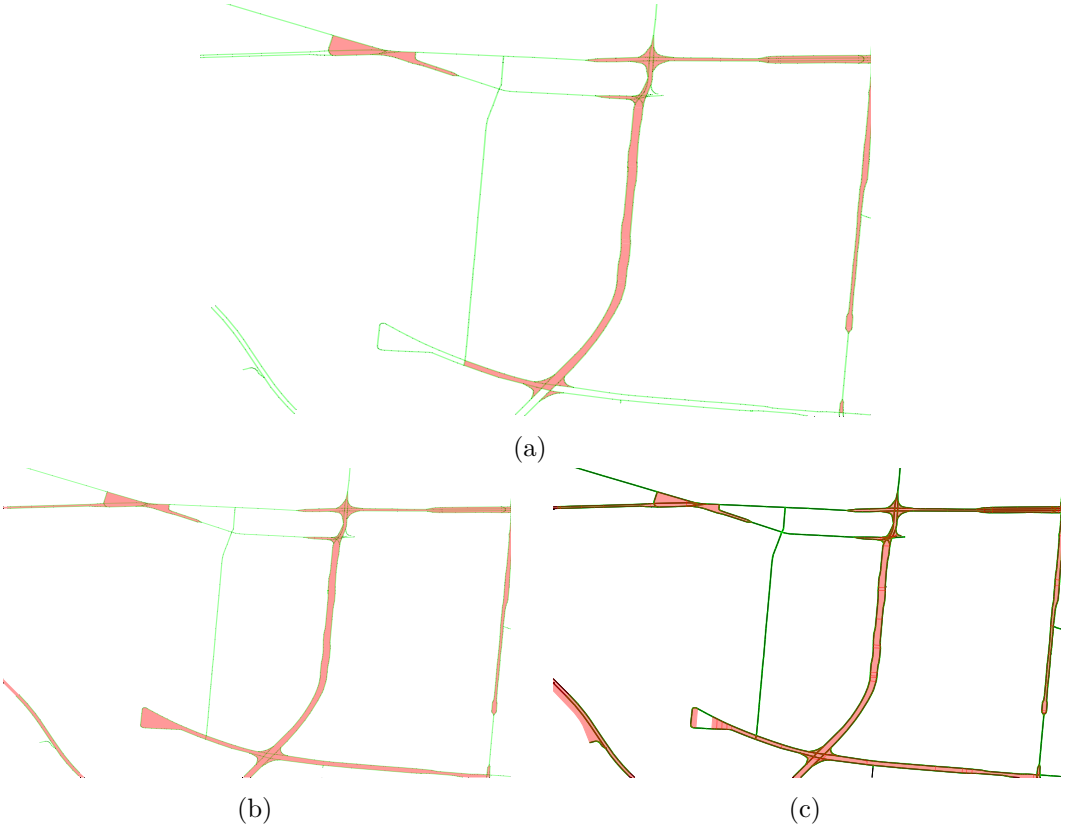
graph are (to a human) obvious. A single drawn line would close a street polygon, where a street polygon obviously should be. As the computing of the street polygons is done automatically, an approach is needed to create this closing line. Several such approaches were tried and then one is used. Those approaches and their problems are described below.

The first approach connects each degree-1 node to the nearest degree-1 node, in terms of the Euclidean distance between the nodes. To avoid connecting single stubs that are unrelated, an upper bound for the distance between the connected nodes is used. This approach works well for cases where two stubs end close to each other (Figure 3.4: B and C). However, a single stub causes problems for this approach (Figure 3.4: A).

The second approach adds the bounding box of the graph as lines. For stubs caused by the bounding box limitations this solves the problem. However, stubs in the inner parts of the graph remain a problem. Figure 3.5: (b) shows some results by using this method.

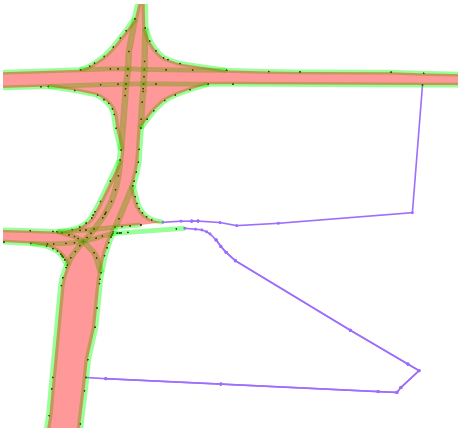
The third approach is to use non-splittable edges to connect the stub to the rest of the graph. A shortest path P (consisting of only non-splittable edges) to the graph is computed with its start being the end of the stub. The main problem of this approach is that in some cases, the path P creates a face which is not considered a street polygon. Figure 3.6a shows an example of this problem.

An additional fourth approach is to use all edges (split and non-split) to compute the facets. Here some facets, that are easily recognized in the previous graph, are cut into

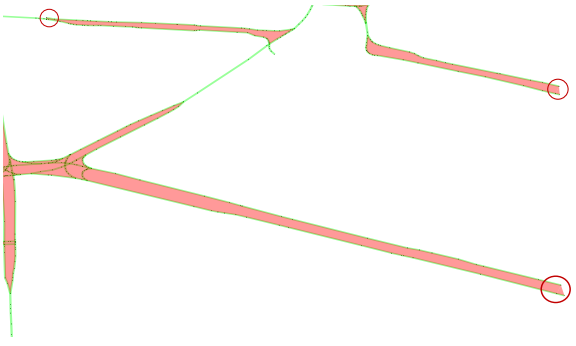


(a) (b) (c)

Figure 3.5: A part of the original problem (a), the result of the second approach (b) and the result of the fourth approach (c).

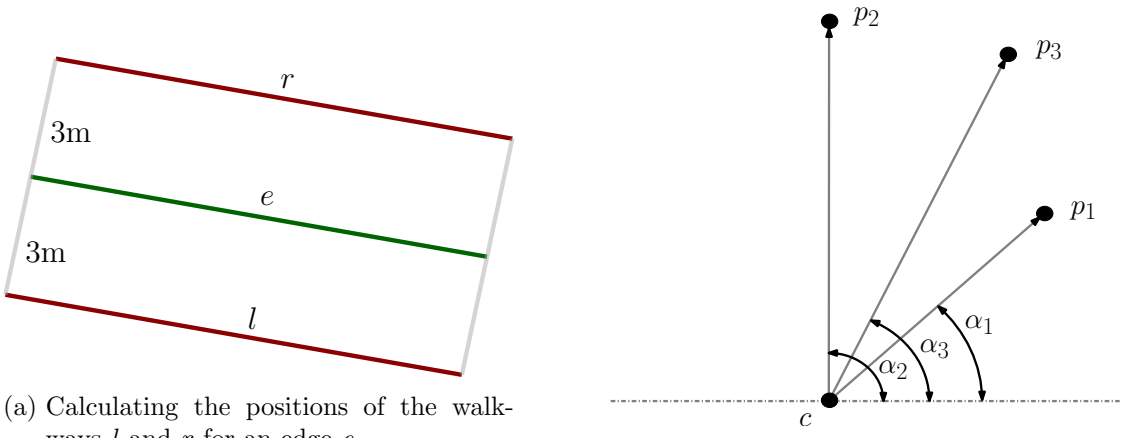


(a) Two detours of non-splittable edges (colored purple) that creates two faces which are not thin polygons.



(b) A partial result of the used method for tying stubs, showing the same situations from Figure 3.4.

Figure 3.6: Some results of the third approach and the used approach.



(a) Calculating the positions of the walkways l and r for an edge e.

(b) Sorting points p_1, p_2 and p_3 using a center point c, resulting in $p_1 < p_3 < p_2$.

Figure 3.7: The positioning of walkways and sorting of points.

many smaller facets. Those smaller facets no longer satisfy the thinness-criteria (Section 3.1.1). Figure 3.5: (c) shows some results by using this method.

The fifth method, which is the one used for the tying, adds a perpendicular edge at the end of each stub, where the length of this edge is a parameter. Nodes are then added at the intersections of this line with the graph. Edges between the stub and these nodes are added, so that the graph remains planar. The planarization method (Section 2.1) is used to do this, where intersections between original graph edges are ignored.

An improvement to the used method would be using only the nearest intersection, instead of adding nodes for all intersections. This improvement was omitted in order to avoid further complications of the planarization method. In addition, if the intersections of this line with the graph are more than one, then there are graph edges that run nearly parallel and are close to each other. The newly added edges will either be in an already existing red (forbidden) polygon, or will create one. Due to the meaning of the street polygons, both cases are plausible.

The used length for the perpendicular edge is 5.5m. Figure 3.6b shows some results of this method.

The time complexity needed when tying stubs is bounded by $\Theta(\#stubs + \text{planarization}\, G)$.

3.1.3 Splitting edges

A procedure is needed to create walkways around a single edge. The method used in this thesis is to create a rectangle R. The edge $e \in E$ being split is a side of R. The two other sides of R that are perpendicular to e have length equal to 3m. The side of R that is parallel to e is then one of the walkways around e. Two such polygons exist and they yield the two walkways l and r around e.

Figure 3.7a shows an example of the method for an edge $e \in E$.

3.1.4 Computing the walkways

Similar to the tying of stubs, several approaches were designed to create the walkways, before a satisfying method was discovered. All approaches and their problems are described below.

The first approach uses intersection nodes and their direct incoming edges. The direct edges $(v_o^j, v_i) := (v_{\text{outer}}^j, v_{\text{inner}})$, $j \in \{1, ..., \text{indeg}\, v_i\}$ of an intersection node $v_i \in V_i$

are sorted according to their angle, using the coordinate of v_i as a center point. Figure 3.7b shows how lines $\{(p_1, c), (p_2, c), (p_3, c)\}$ are sorted arount a center point c. In this case the resulting sorted set is $\{(p_1, c), (p_3, c), (p_2, c)\}$, as $\angle p_1 < \angle p_3 < \angle p_2$.

According to this order, each edge $e := (v_\mathrm{o}^j, v_\mathrm{i}) \in E$ has a left unique neighboring edge $l_e := (v_\mathrm{o}^{j+1}, v_\mathrm{i}) \in E$ and a right neighboring edge $r_e := (v_\mathrm{o}^{j-1}, v_\mathrm{i}) \in E$. Here, $i \in \{1, ..., n\}, v_\mathrm{o}^0 := v_\mathrm{o}^n, v_\mathrm{o}^{n+1} := v_\mathrm{o}^1, n := \mathrm{indeg}\, v_\mathrm{i}$.

Each direct incoming edge e of an intersection node v_i is then split into a left walkway, denoted by l_e^s, and a right walkway, denoted by r_e^s. For this, the method described in Section 3.1.3 is used. Additionally, the same sorting procedure is used to determine which created walkway is the left edge (l_e^s) and which is the right edge (r_e^s). For this sorting, as v_i is not an end point of l_e^s and r_e^s, any end point of l_e^s and r_e^s can be connected to v_i to form the lines that represent l_e^s and r_e^s – either way the sorting will determine which created walkway is to the left and which one is to the right of e.

The left walkway edge l_e^s is crossed with the walkway edge $r_{l_e}^\mathrm{s}$ (right walkway of left neighboring edge), resulting in a new node n_l at their intersection point. The right walkway edge r_e^s is crossed with the walkway edge $l_{r_e}^\mathrm{s}$ (left walkway of right neighboring edge), resulting into a new node n_r at their intersection point. The endpoints of l_e^s and r_e^s nearest to the current intersection node v_i are then replaced with respectively n_l and n_r. A bridge edge is then added between n_l and n_r, representing a crosswalk.

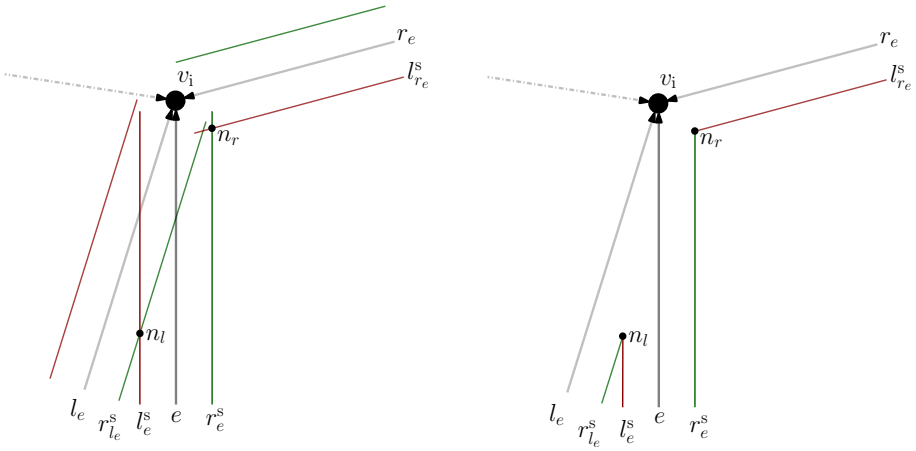
The other endpoints of l_e^s and r_e^s remain unchanged, unless the edge e connects two intersection nodes. In this case the above procedure will be repeated when the other intersection node is being processed. With a simple alteration this procedure can be used to split the edges of all nodes, regardless of their degree. When a node v of degree 1 is processed, its single direct edge e has no neighbors. In this case no crossings are computed and the endpoints of the walkway edges corresponding to v are unchanged.

Figure 3.8a shows an example of the first approach. The main problem for this approach are edges of small length. Such edges can be used to model curves of streets, which is often seen in road network databases. The intersections with a neighboring walkway edge that is too short lies outside this walkway edge. Another problem is the crossing of the walkway edges, when the angle between those edges is of degree greater than 90°. Especially, for an angle with degree greater than 180° the intersection may have a huge distance to the current intersection node. Figure 3.8b shows two such problems.

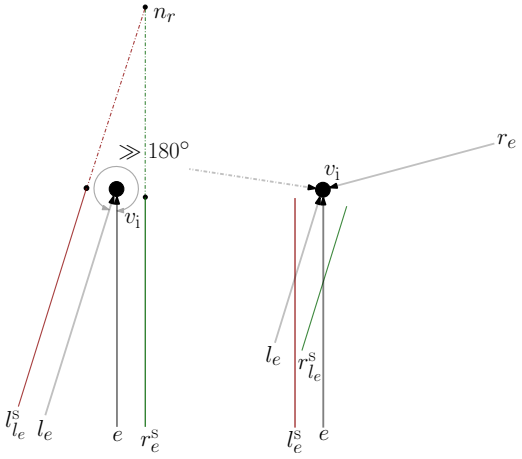
A simple solution is to create an edge between the endpoints (nearest to the current intersection node) of the artificial edges being crossed, if the intersection of those artificial edges is not in their interior. Figure 3.8c shows how this solution works for the two problems in Figure 3.8b. As seen, this solution does not yield a plausible result when a neighboring edge is too short.

The second approach is to use the union of rectangles. The splitting of each edge defines a rectangle, two of its sides being the two artificial edges. For each edge that should be split a rectangle is computed (similar to the method in Section 3.1.3). Then a union of all those rectangles is created and its outline is used as the new pedestrian ways. Figure 3.9 shows an example of this approach.

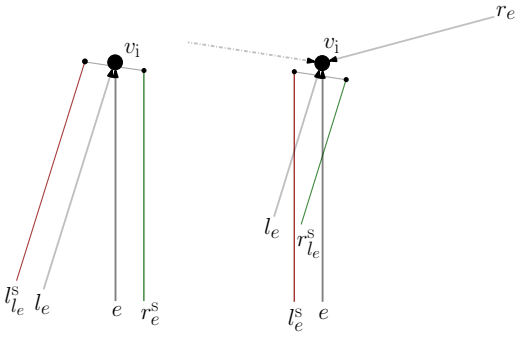
A problem here is the amount of rectangles that need to be unified. Even for small graphs (ca. 4 000 edges that should be split) the computing of this union would last for minutes. As this approach would yield the best results, an attempt was made to reduce the amount of polygons that need to be unified. Nodes with degree of 2 are aggregated into *degree-2 paths*. Each such path P is then split (for instance by using the first approach). The resulting two walkway paths define a polygon. The splitting is a fast procedure and all small rectangles on P no longer need to be unified. Then a union needs to be computed



(a) An example of the first approach for node v_i and edge e.



(b) Problems of the first approach with big angles (left) and short edges (right).



(c) The simple (and flawed) solution for the problems in Figure 3.8b.

Figure 3.8: The first approach for the splitting.

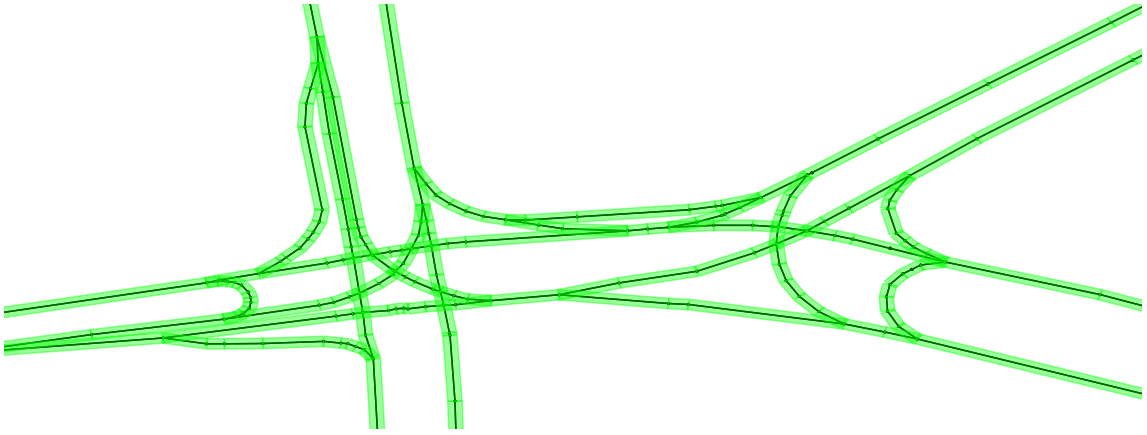
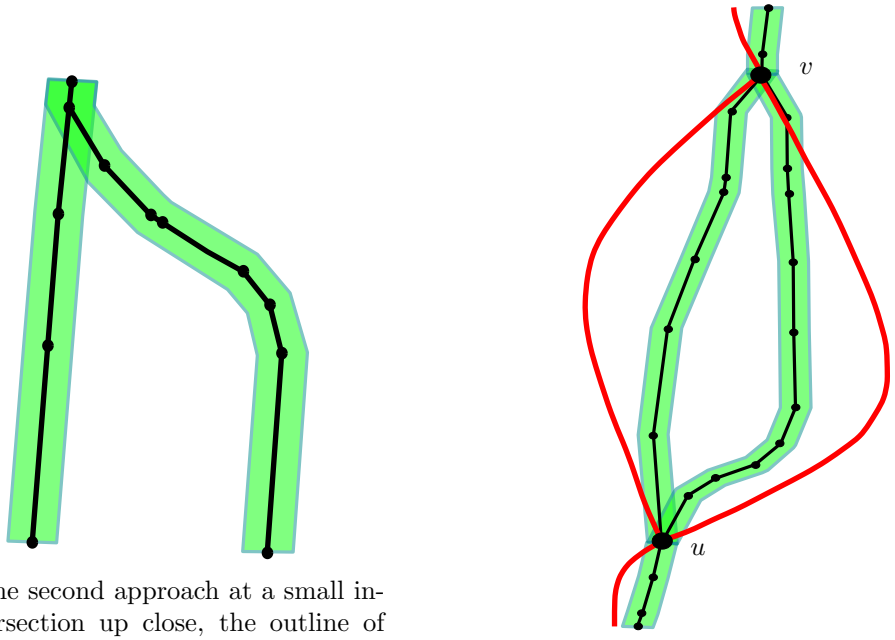


Figure 3.9: The second approach at complex street intersections.



(a) The second approach at a small intersection up close, the outline of the green area is where the walkways should be.

(b) Several intersection-free paths (curved and colored in red) adjacent to two intersection nodes u and v.

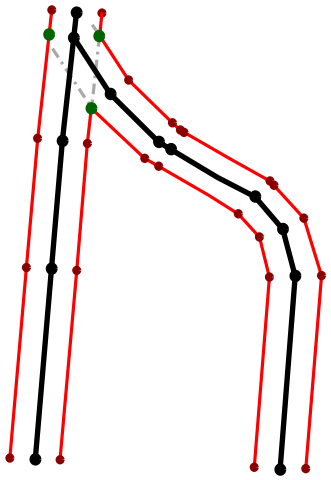
Figure 3.10: The second splitting approach and intersection-free paths.

by using only the polygons around the degree-2 paths. However this, too, proved to be slow. Figure 3.10a shows two polygons around two degree-2 paths and their union. The outline of this union is then added as walkways.

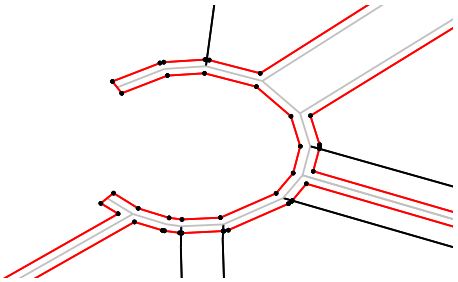
In order to retain the speed of the first approach and deal with its problem, the used method contracts the graph. Each path of degree-2 nodes is replaced by an intersection-free path. This intersection-free path contains the path of nodes and the edges connecting them and its start and end nodes are the start and end nodes of the path All intersection nodes are then connected only by single intersection-free path. This new graph consisting of intersection nodes and intersection-free paths is a multigraph. A small example can be seen in Figure 3.10b.

A procedure splits the intersection-free path into 2 polylines as in the second approach, but without creating polygons by using those polylines. Similar to the first approach, the intersection-free paths of an intersection node are sorted according to their angle, with the intersection node as center. For this sorting only the edge directly connected with the intersection node is used (as in the first approach). The matching between the walkway polylines is done in the same way as in the first approach, however crossings between 2 polylines can be more than 1. In such case, the closest (in euclidean distance) intersection point to the current intersection node is used. When this crossing is computed, parts of the walkway polylines are deleted, as seen in Figure 3.11a. Here the created walkway edges are drawn in red, their endpoints are smaller red dots, the original edges are black, the original nodes are bigger black dots, the intersection points are green and the deleted parts of walkways are gray. The procedure is done for all intersection nodes and so all intersection-free paths are split. Figure 3.11b shows the result of using this method for a small portion of a graph, with the splittable edges drawn in gray, the walkways drawn in red and non-splittable edges drawn in black.

The time complexity needed to compute the walkways is bounded by $\Theta(|E_s|)$.



(a) The walkways, after the intersection points have been computed (deleted segments are colored in gray).



(b) The result of the used walkway-creating method for a couple of intersections.

Figure 3.11: The used splitting approach and some results.

3.1.5 Connecting the walkways to the routing graph

After the set of walkways is computed (Section 3.1.4), all original splittable edges are deleted. As the streets that should have walkways now do, these streets are no longer needed. Now the new walkway edges need to be connected to the routing graph, so that they can be used for the routing. To do this, intersections between the created walkways and the non-splittable edges are computed and added as nodes. This is done with the planarization procedure (Chapter 2.1), where the set of non-splittable edges is passed as an argument in order to ignore intersections between non-splittable edges.

Additionally, the layers of edges are respected. This avoids connecting the walkways to bridges above them and tunnels below them, as the walkways are considered to be on the ground layer.

3.2 Handling free areas and parks

As already mentioned, in standard routing free areas are walked around. In order to pass directly through such areas while routing, additional edges are added to the routing graph. Further, when the route starts or ends in a park, the nearby footpaths are of interest. Thus, the facets of the routing graph that lay in parks must be computed.

This chapter explains how this all is done. Extracting information about free areas and parks, adding visibility graph edges for free areas and dividing parks in facets are described here.

3.2.1 Extracting the free areas

Information about free areas is contained in the input data. Generally, any representation of the free areas can be in the input. Thus an extraction procedure is needed, that is dependant on the input data. The extracted information about the free areas is in the form of simple polygons, to enable uniform handling in the rest of the procedures.

3.2.2 Merging free areas

Because of their representation in this thesis, free areas may overlap, touch or be completely within other free areas. A mechanism is needed to merge such areas, so that

Figure 3.12: An example for touching areas that need to be merged, before they can be properly traversed via visibility graph edges.

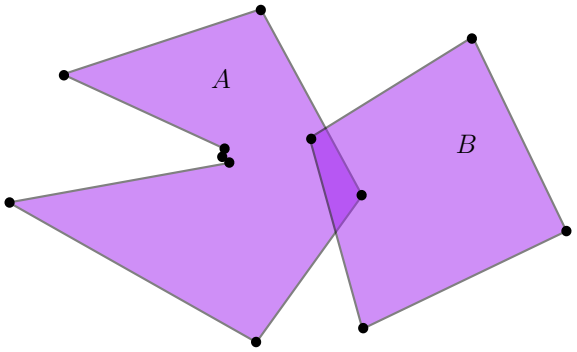


Figure 3.13: Intersection of areas where no node exists in the graph.

a pedestrian can freely traverse an area as a whole, instead of having to traverse many smaller areas one by one. Figure 3.12 shows a simple example with two overlapping areas. For this, the union of geometrical figures is used. As the free areas are generally few (∼200 for a city with a population of ∼300 000), this procedure is fast.

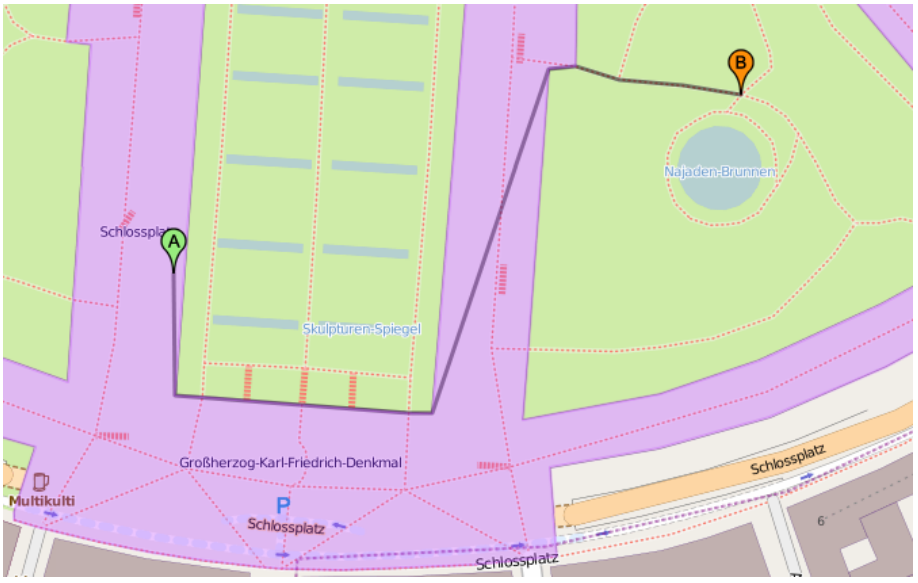
It is possible that areas intersect at points where no node in the graph exists. Figure 3.13 shows an example of this. To simplify the merging, the planarization procedure is used on the graph. The set of edges that do not lay to areas is passed as an argument, so that only intersections between area edges are handled.

For the merging, all unions of free areas that share common area are computed. This is done by using the union operator for 2D figures. Let A denote the set of all areas. An area $a \in A$ is chosen and the set $S_a := \{p \in A \mid p \text{ and } a \text{ have common area}\}$ is computed. To compute S_a faster, a quadtree is used. This quadtree returns all nearby free areas of a, which are then tested for common area with a. If $S_a = \{a\}$, nothing is done. Otherwise $U_a := S_a \cup \{a\}$ is unified into a single figure g (g is a not necessarily simple polygon). All areas in U_a are removed from A, g is added to A and the quadtree is updated in the same way. This procedure is repeated until no areas in A overlap or touch, i.e. until $\forall a \in A : S_a = \{a\}$. A then contains all merged areas, and is hence denoted by $A_\text{m} := A_\text{merged}$.

Finally, barriers are removed from the merged areas. A *barrier* is an area that cannot be traversed on foot, such as a lake. Similar to the free areas barriers should also be present in the input data. When not, the removal of barriers is omitted.

The removal of the barriers from the merged areas is done in the same way as the

(a) A route passing through a free area.



(b) A route starting in a free area.

Figure 3.14: Two scenarios for traversal of free areas.

merging. For $a \in A_{\mathrm{m}}$ nearby barriers are located by using a quadtree containing all barriers. All such barriers that aren't disjoint to a are then removed from a. This is done by using the difference operator for 2D figures.

The time complexity when merging n figures and removing m barriers is bounded by $O(n(\mathrm{QTQuery}\, n + \mathrm{QTQuery}\, m))$. Again, $\mathrm{QTQuery}\, n$ denotes a query to a quadtree with n elements.

3.2.3 Adding visibility graph edges

A route can traverse, start in or end in a free area. Figure 3.14 shows an example of two scenarios. For the first scenario only the shortest paths between entry points of the areas are needed. An *entry point* is any intersection between the routing network and the boundary of the free areas. However, when the route starts or ends in a free area all visibility graph edges, denoted by $E_{\mathrm{v}} := E_{\mathrm{visibility}}$, are needed.

Generally, it is not known beforehand if the route will pass through a free area. So

visibility graph edges that lay on shortest paths, denoted by the set $E_{\text{osp}} := E_{\text{on shortest path}}$, are always needed. On the other hand, whether the route starts or ends is in a specific free area is known. So the rest of the visibility graph edges, $E_{\text{v}} \setminus E_{\text{osp}}$, can either be used or ignored. How this is done is explained in Chapter 4.3. For this chapter, it is only important to know why E_{osp} is explicitly computed, as this means additional computational and implementation cost.

First, the boundary of the free areas is added as edges and nodes. The free areas represent places where pedestrians can walk freely, thus their boundary can also be traversed. After that, all intersections between the routing network and the boundary of the free areas are added as nodes. Again, this is done by the general planarization procedure. The nodes at those intersections are marked as *entry* nodes, i.e. the entry nodes of an area $a \in A_{\text{m}}$ are denoted by $a_{\text{e}} := a_{\text{entries}}$.

For each area $a \in A_{\text{m}}$ the set of visibility edges E^a_{v} is then computed. Each line l between two different vertices $u, v \in a$, that is completely within a, is added to the set E^a_{v} as the edge (u, v).

To find the set $E^a_{\text{osp}} \subseteq E_{\text{v}}$ a shortest path tree T_v is computed for each node $v \in a_{\text{e}}$. The root node of T_v is v and T_v is computed only for the graph induced by the set of edges E_{v}, namely $G_{\text{v}} := (\{v \in V | (v, u) \in E_{\text{v}} \vee (u, v) \in E_{\text{v}}\}, E_{\text{v}})$. Here *Dijkstra's algorithm* (Algorithm 4.1) is used to compute each tree, however any other shortest paths algorithm can be used. The edges present in T_v are the set E^a_{osp}. Those edges are then added to the routing network.

This is done for all areas and each area a now has its set of entry nodes a_{e} and its set of visibility graph edges that do not lay on shortest paths between entry nodes $E^a_{\text{vo}} := E^a_{\text{visibility only}} := E^a_{\text{v}} \setminus E^a_{\text{osp}}$. Both sets will be needed when the route's start or destination is in the free area a.

The time complexity when adding the visibility graph edges is bounded by $O(K + \text{planarization}\, G)$, where $K := \sum_{a \in A_{\text{m}}} |a|^2 \log |a|$. Here $|a|$ denotes the number of vertices in the merged area a.

3.2.4 Processing parks

Similar to the free areas, the parks are extracted and then merged. This is done in the same way as described in Section 3.2.1 and Section 3.2.2. The set of merged parks is then denoted by $P_{\text{m}} := P_{\text{merged}}$, a set of not necessarily simple polygons (figures).

The boundary of the merged parks are added to E and all edges that do not lay on the ground layer are removed from E. All intersections between edges in parks are added as nodes, so that the facets in parks can be computed. This is done by the planarization procedure and the set $E_{\text{np}} := E_{\text{non-park}} := \{e \in E \mid e \text{ has no common area with any } p \in P_{\text{m}}\}$ is passed as an argument, to ensure that only the parks are planarized.

For each park $p \in P_{\text{m}}$ the set $E^p_{\text{p}} := E^p_{\text{park}} := \{e \in E \mid e \text{ has common area with } p\}$ is computed via the quadtree containing the figures of P_{m}. The graphs $G_p := (\{v \in V | v \text{ is an end point of } e \in E\}, E^p_{\text{p}})$ are constructed and their facets are computed. Each such facet will represent a park area in which the route could have its start or destination.

The edges representing boundaries of parks are removed from E and the edges that lay on layers different than the ground one are restored.

Finally, a not necessarily simple polygon is created for each computed facet. Barriers are then removed from these polygons, as done in Section 3.2.2. Those polygons will be used to determine whether the start or destination is in one of the facets and if so, in exactly which facet. At this point barriers are also removed from the figures in P_{m} (by using the same method).

The time complexity when processing the parks is bounded by $O(|E|\, \text{QTQuery}\, |P_{\text{m}}| + \text{planarization}\, G)$.

4. Routing

After the modeling is done, the actual routing can take place. This chapter describes how the route is computed and what additional procedures are needed when handling free areas and parks.

4.1 Routing algorithm

We use the classic *Dijkstra*'s algorithm[Dij59] to compute all routes (Algorithm 4.1). However, some modifications are required.

The start or end of a route can be at a random point of a park or a free area. In order to compute such a route, some connection must be made from the start/destination to the routing graph. This rises the need to either modify the graph before the routing, or to alter Dijkstra's algorithm so that a route with many start and end nodes can be computed. For example, a route starting in a free area will have multiple start nodes – all nodes that are visible from the start location. Another example would be a route ending in a park – the multiple targets are nodes of the facet where the target location is in. To allow simultaneous route queries on the same graph, the second approach is used. The alterations of the algorithm are described below.

The input set of start nodes is denoted by S and the input set of target nodes is denoted by T. Each node $v \in S \cup T$ also has an additional cost c_v that is known before the routing.

The queue is initialized with all nodes in S and their costs. So for $v \in S$ we set $d(v) = c_v$, instead of 0. When a path to a source node v is found, with a cost less than c_v, v is no longer a source node and is removed from S.

When a node $v \in T$ is removed from the priority queue, c_v is added to $d(v)$ and we say that v is *reached*. The routing algorithm finishes when all targets have been reached, or the current route length is longer than the distance to the first reached target.

The target of the computed route t is then the node $v \in T$ with the least $d[v]$. The start of the route s is the first node $v \in S$ discovered on the path of parents of t. Thus the shortest route between all pairs of start and target nodes is computed.

A binary heap is used as a priority queue, so the time complexity of a query is bounded by $O(n \log n + m \log n)$, with $n := |V|$ and $m := |E|$.

Algorithm 4.1: DIJKSTRA

Input: Graph $G = (V, E, \omega)$, source node s
Data: Priority queue Q
Output: Distances $\mathsf{d}(v)$ for all $v \in V$, shortest-path tree of s given by $\mathsf{pred}(\cdot)$

```
// Initialization
forall v ∈ V do
    d(v) ← ∞
    pred(v) ← null
Q.INSERT(s, 0)
d(s) ← 0
// Main loop
while Q is not empty do
    u ← Q.DELETEMIN()
    forall (u, v) ∈ E do
        if d(u) + ω(u, v) < d(v) then
            d(v) ← d(u) + ω(u, v)
            pred(v) ← u
            if Q.CONTAINS(v) then
                Q.DECREASEKEY(v, d(v))
            else
                Q.INSERT(v, d(v))
```

4.2 Considering road junctions

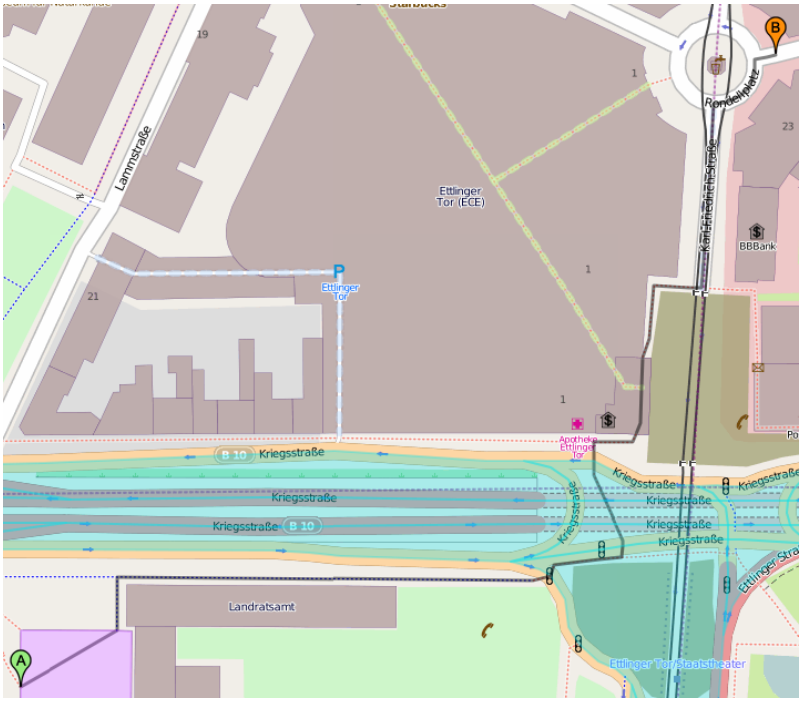
Due to traffic lights, road junctions are places where a pedestrian often waits some time before continuing to walk. Bigger intersections normally take longer to traverse as the traffic lights have longer switching times. Without the knowledge of the waiting times on the different intersections, two general approaches were designed. Both exploit the street polygons and the information needed for them can be computed during the preprocessing.

The first approach penalizes the entry of road junctions. For each edge $e := (u, v) \in E$ information is gathered during the preprocessing, whether u is not within a street polygon and e has common area with any street polygon. If yes, then the edge e is marked for entry penalties. Again, a quadtree is used to find nearby street polygons, so that the conditions can be tested.

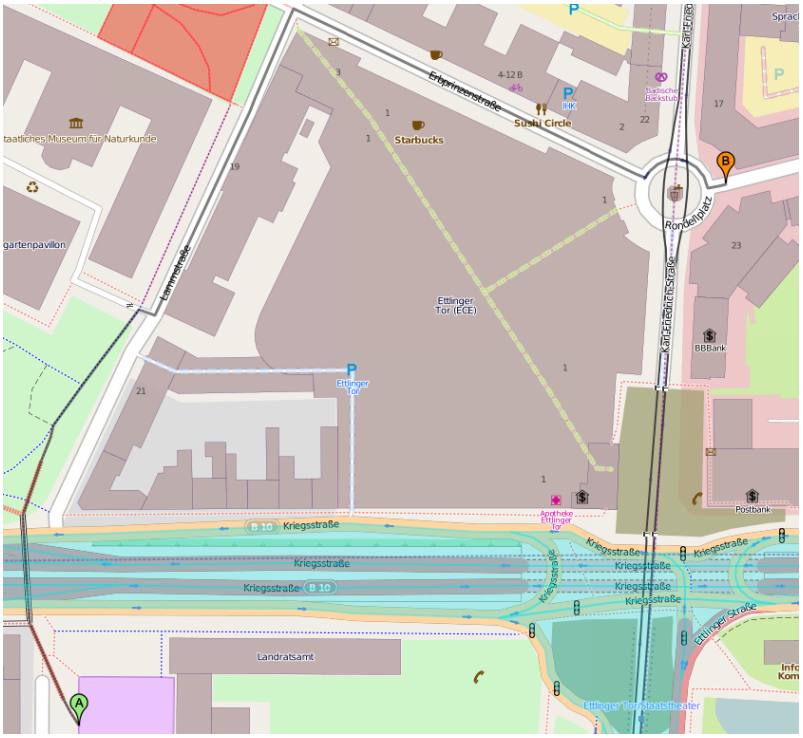
During the routing any marked edge will have additional weight added to it, based on the entry penalty that is chosen currently. Figure 4.1 shows an example of how the route can by the use of entry penalty. Street polygons here are painted in cyan, as bigger streets are painted red in the *OpenStreetMap* tiles (which are used for the visualization of the map).

The second approach penalizes the edges within crossroad areas. The length of each edge $e \in E$ that is within the street polygons area is computed. Here, nearby street polygons of e are located with a quadtree. The common area of e with those polygons is then computed. This area is denoted by $l^e_{\mathrm{w}} := l^e_{\mathrm{within}}$.

During the routing each edge $e \in E$ receives an additional weight $w_e := l^e_{\mathrm{w}} \cdot \mathrm{penalty}_{\mathrm{w}}$, where $\mathrm{penalty}_{\mathrm{w}}$ is the current penalty per centimeter for edges within street polygons. Figure 4.1 shows an example of how the route can differ based on $\mathrm{penalty}_{\mathrm{w}}$ values.

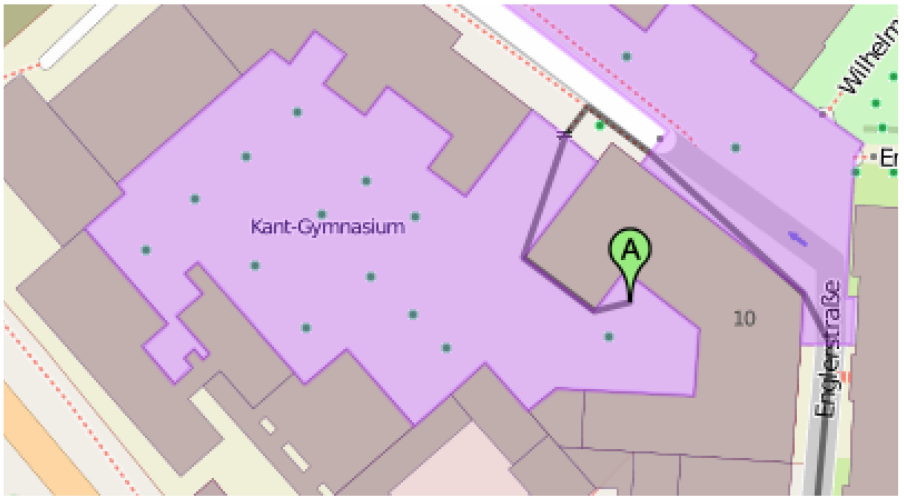


(a) A route computed without penalties passes through the intersection.

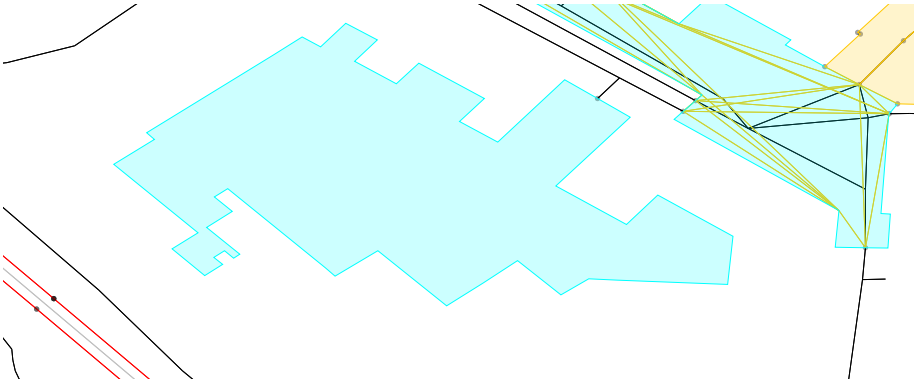


(b) The route for the same query, computed with entry penalty of 60s or within penalty of $1\frac{\mathrm{m}}{\mathrm{s}}$. Here a bridge passing over the street is used.

Figure 4.1: Difference between computed routes based on penalty values.



(a) The start location of the route has no visible entry nodes.



(b) The single entry node for the free area.

Figure 4.2: The need of free area describing edges and nodes in the graph, in order to route properly.

4.3 Start or destination in a free area

While simply passing through a free area can be completely handled during the preprocessing, special handling is needed when the route has its start or destination in a free area. At the very least, all visible graph nodes must be located and connected with the location in a free area. The start location for the route is hence denoted by s_l and the target location by t_l.

When first approaching this problem, only entry nodes were considered to be graph nodes and the set of edges E_vo was ignored (Chapter 3.2.3). In other words, nodes and edges that served only to describe the area were to be discarded. This is desirable as it means fewer edges and nodes in the graph structures. The edges in E_vo and their endpoints are needed only when starting or ending in a free area, which is a special case of the general routing. Unfortunately situations as the one seen in Figure 4.2 show the need for the set E_vo to be in the graph structure. Better routes are achievable with the usage of E_vo, so the edges of this set are also stored.

While the space cost cannot be avoided, the computational cost for using edges in E_vo can be minimized. For this, each edge $e \in E_\text{vo}$ is marked with the free area a it lies in and we say that e *belongs to* a. In addition, a node n that has incident edges in E_vo has those edges stored in the end of its edge list. This way the relaxing of incident edges for n can stop as soon as an edge is discovered, which belongs to an irrelevant area. In other words, if the route does not start or end in an area a, edges that belong to a are not considered. No node can have incident edges belonging to two different free areas, as free areas with common area are merged in Chapter 3.2.2, so the approach yields correct results.

In order to find in which area a location lies, a quadtree is used. This quadtree is built once during the program's initialization and contains information about the free areas. Each free area has its not necessarily simple polygon, the set of nodes on its border and a unique id.

The actual routing regarding the starting or ending in a free area proceeds as follows. The free area in which the route starts, denoted by a_s is found by using the quadtree. Same is done for the free area in which the route ends, denoted by a_t.

If $a_s = a_t$ and s_l is visible to t_l, then the two points are simply connected and the route's distance is the distance between s_l and t_l.

If a_s is empty, i.e. s_l is not in a free area, then the graph node n nearest to s_l is located and the routing algorithm is started from the pair $(n, 0)$. Otherwise all boundary nodes $n_i \in a_s$ visible to s_l are added as pairs $(n_i, \mathit{distance}(n, s_\text{l}))$, to serve as the starting points for the routing.

If a_t is empty the target of the routing is the graph node nearest to t_l. If not, the nodes $n_j \in a_t$ visible to t_l are used as targets for the routing, with distance to them equal to $\mathit{distance}(n_j, t_\text{l})$.

When the route is computed, s_l and t_l are also added to the drawing of the route. Figure 4.2 shows an example of a route starting in a free area.

4.4 Start or destination in a park

Unlike free areas, parks are not traversed freely, as it is expected that footpaths are used to traverse the parks. As already mentioned, those footpaths are to be reached in a more realistic fashion, than simply finding the nearest footpath. Thus, the preprocessing for parks is done to allow special handling when starting or ending in a park. The selection

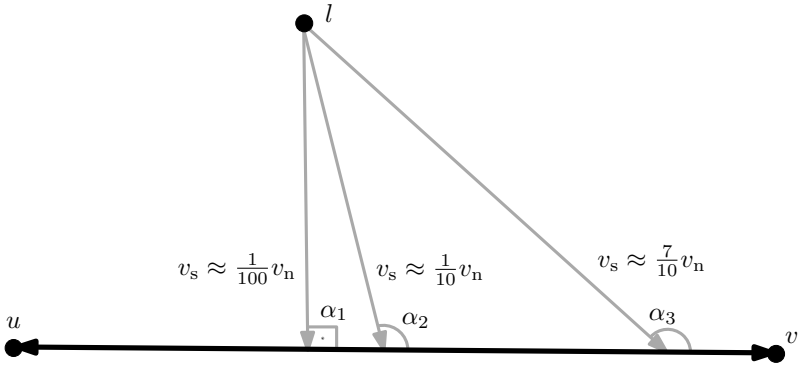


Figure 4.3: Entering an edge (u, v) from a location l to node v by using different speeds for v_s.

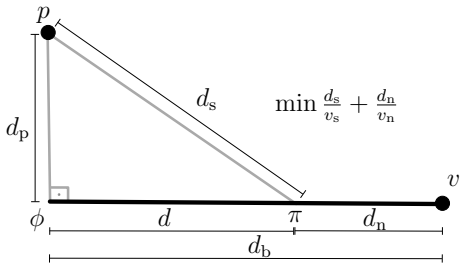


Figure 4.4: Walking inside the park for a distance d_s with speed v_s and on a path of the park for distance d_n with speed v_n.

of the starts, targets and the drawing of the route are now described.

The park facets in which the start and destination lie in, denoted by P_s and P_t are located. As with the free areas, this is done by using a quadtree. For each edge $e := (v, u)$, that shares common area with $P_s \cup P_t$, two additional points $p_v(e)$ and $p_u(e)$ are computed. We say that p_v and p_u are the *projected points* of v and u. Those points define where the route can enter or leave the inner area of a park facet.

In order to connect the route with the start or destination inside a park facet, some distance must be covered inside the facet. As already mentioned, we assume that parks have footpaths which pedestrians prefer. Thus, a different speed, $v_s := v_{\text{slow}}$, is used to make the walking inside a park facet more expensive than walking on the boundary of the facet. This speed v_s is less or equal to the normal walking speed, $v_n := v_{\text{normal}}$. A very small value of v_s compared to v_n should yield as little distance walked inside the facet as possible. Higher values of v_s should allow more distance walked inside the facet, effectivly changing the angle with which the facet boundary is reached. Figure 4.3 shows how an edge e is entered with different angles α, caused by different speeds v_s.

We want to minimize the sum of the expensive walking inside a park facet and walking the rest of the route. Lemma 4.1 yields the desired projection point of a node v on a edge e of a park facet P.

Lemma 4.1. *The fastest route from a point p in a polygon P to the endpoint v of a polygon edge $e := (u, v)$ consists of a straight segment from p to a point π on e and then from π along e to v, where*

$$\pi = \frac{d}{\text{length}\, e} \cdot (v - \phi),$$

$$\phi := \textit{the perpendicular projection of } p \textit{ on } e,\ d := \frac{\rho \cdot \text{distance}\,(p,\phi)}{\sqrt{1-\rho^2}} \textit{ and } \rho := \frac{v_s}{v_n}$$

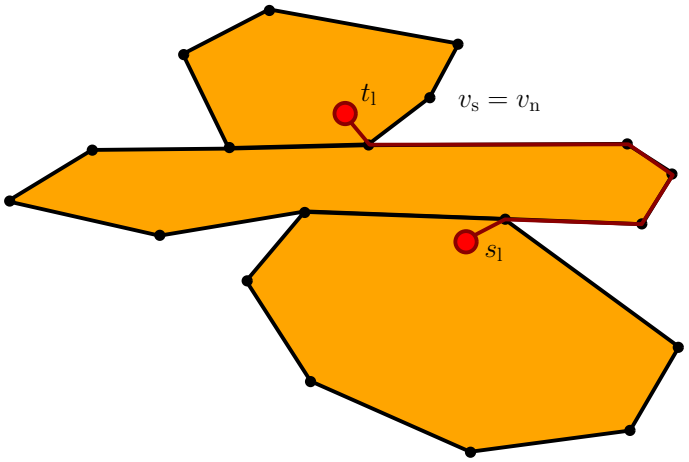


Figure 4.5: A detour caused by a thin park facet.

Proof. Figure 4.4 depicts the situation. The straight segment from p to π is denoted by d_s and the straight segment from π to v is denoted by d_n. The length of e is denoted by d_b.

We want to minimize the function $\frac{d_\mathrm{s}}{v_\mathrm{s}} + \frac{d_\mathrm{n}}{v_\mathrm{n}} = d_\mathrm{s} + d_\mathrm{n} \cdot \rho$, i.e. the time spent "slowly" walking inside the facet plus the time spent walking "normally" on the boundary of the facet. Note that any distance walked "slowly" is acceptable, as any projection of p that is not on the edge e will be set to v (clipping). In other words, we are solving an unrestricted minimization problem. Thus, we simply check where the derivative is equal to 0.

To simplify matters, let $d_\mathrm{n} := d_\mathrm{b} - d$ and $d_\mathrm{s} := \sqrt{d_\mathrm{p}^2 + d^2}$ (we consider only positive distances). In this way our function to minimize is $f(d) := (d_\mathrm{b} - d) \cdot \rho + \sqrt{d_\mathrm{p}^2 + d^2}$. We set its derivative to be equal to 0, i.e. $-\rho + \frac{d}{\sqrt{d_\mathrm{p}^2+d^2}} \doteq 0$, and receive the value we set for distance $(\pi, \phi) = d$. Plotting the function f shows that this single extremum is a minimum. □

Each node n incident to a park facet edge of P_s or P_t is given an additional cost $c_{n,l} := v_\mathrm{s} \cdot \mathrm{distance}\,(l, p_n) + v_\mathrm{n} \cdot \mathrm{distance}\,(p_n, n)$, where p_n is n's projected point on the edge n is incident to and l is the start location, if $n \in P_s$, or the destination, if $n \in P_t$. If n is incident to two edges, then the point that yields smaller distance is chosen. The pair (n, d_{n,s_l}) is then added as a start, if $n \in P_s$, or (n, d_{n,t_l}) is added as a target, if $n \in P_t$.

A final step after the routing is to check whether the pedestrian could walk directly from s_l to t_l. If $P_s = P_t$, s_l is visible to t_l and the cost of walking directly (in the park) is less than the cost of the already computed route, the actual route consists of the line between the start and destination.

A problem here occurs when P_s or P_t have thin neighboring facets. This could cause a considerable detour, as depicted in Figure 4.5. A simple mechanism is used to avoid this. Instead of using only the facets P_s and P_t, the procedures above are applied for all facets in an ϵ-Neighborhood of s_l and t_l. Again, those facets are found with the quadtree and the value of ϵ can be chosen freely before the routing. Edges that belong to facets other than P_s and P_t must also be within the ϵ-Neighborhood in order to be used. This avoids using far away edges of neighboring huge facets, as such edges may be far beyond the ϵ-Neighborhood.

To check whether s_l or t_l are visible to any projected point, the whole merged park geometry (in which s_l and t_l lay) is used.

While this solution yields plausible results, the time needed by the quadtree to find

nearby facets grows the bigger ϵ is and is generally huge compared to the time needed for the routing.

5. Experiments

In this chapter exemplary computed routes are shown and compared to standard routing software routes. Statistics about the amount of additional information computed in the preprocessing are given, as well as statistics about the routing queries. The modeling is done for several cities and statistics are gathered. For the queries the city Karlsruhe (Baden-Württemberg, Germany) is used, as it is a standard city of population 300 000. It has a reasonable amount of plazas and parks, as well as a structured road network. Its *OpenStreetMap* data is also rich and frequently updated.

5.1 Implementation details

The implementation of the preprocessing is written in Java. This choice is made for convenience only and does not affect how the modeling is done. The complete modeling is done in a preprocessing phase, so the slower execution time is not an issue.

The routing is written in C++, as the routing queries are done online and their execution time is important. For compiling *GCC 4.6.2* with optimization level 3 is used.

This chapter shows some information about query and modeling execution time. The specifications of the machine on which they are run can be seen in Table 5.1.

Input data

The used road network database is the *OpenStreetMap* (*OSM*). It is free to use in non-commercial projects and offers highly detailed information about roads, landscapes, points of interest, etc. In addition, it is updated frequently, well documented and easy to

Table 5.1: Reference machine specifications.

CPU Count	2
Cores per CPU	8
CPU Vendor	Intel
CPU Type	Xeon(R) E5-26700
Clock speed	2.6GHz
Platform	64bit
Operating System	SuSE 12.1-64
Total Memory	64532M

use.

The *OSM* information is stored in XML format. As the modeling part is written in Java, the java SAXParser is used to handle the input format. It is a SAX (Simple API for XML) parser implementation that is fast and easy to use.

Generally, the *OSM* information consists of two elements: Nodes mark significant locations, such as road junctions, points of interest and so on. Ways are paths of nodes that represent the actual roads, area and park boundaries. Nodes and ways are marked by tags (tuples *(key, value)*) that clarify what the node or way represents. For the modeling, several important tags are:

- The tag *(highway, *)* defines the type of a way. All ways of a type traversable by a pedestrian are added to the routing graph.
- The tag *(layer, *)* indicates the level of a way or a node. This tag is important when connecting the created walkways to the routing graph, as they are only connected to edges and nodes of the same layer. Otherwise a walkway could be connected to the middle of a bridge or a tunnel. The tag is also important when finding nearby paths in a park.
- Closed ways (ways that start and end with the same node) that belong to the routing graph and are marked by the tag $(area, yes)$ are considered to be free areas.
- Closed ways marked by any of the tags *(leisure, common), (leisure, dog_park), (leisure, park), (leisure, garden)* in combination with the tag *(area, yes)* are considered to be parks.
- Closed ways marked by any of the tags *(natural, water), (leisure, swimming_pool), (landuse, pond), (barrier, *)* in combination with the tag *(area, yes)* are considered to be barriers.

The information in a *OSM* file generally covers huge areas. To model specific areas, during the extraction of the input a bounding box is used for the nodes and edges that should be in the routing graph. This bounding box is a rectangle and all nodes and edges that are not completely within it are not added to the routing graph.

Projection

Similar to other road network databases, in the *OpenStreetMap* database coordinates are kept in the geographic coordinate system, i.e. latitudes and longitudes. To allow simple geometrical computations during the modeling, all node coordinates are projected from the geographic coordinate system in the Euclidean plane.

For this, the Elliptical Mercator[Sny87] projection is used, as it offers good precision. It is generally slower than other projections, however this is not an issue – the coordinates of each node are projected only once. The Elliptical Mercator gives the projection a specific metric, so centimeters were chosen for higher precision.

An exemplary implementation can be found at the *OSM* wiki[1]. This is also the used implementation for this thesis.

Java Topology Suite

The JTS (Java Topology Suite[2]) is an open source library with a convenient API and a port to C++ (Geometry Engine Open Source[3]). It provides various operators for 2D geometrical figures (unification, difference, coverage, area computing, etc.) and the funct

[1] http://wiki.openstreetmap.org/wiki/Mercator#C_implementation
[2] http://www.vividsolutions.com/jts/jtshome.htm
[3] http://trac.osgeo.org/geos/

to save them in the WKB (Well Known Binary) format. This library is used extensively for the various procedures described in Chapter 3.2.

Storing free area and park information

The merged free areas and the merged parks are not necessarily simple polygons. As they are needed for the routing, a mechanism is needed to store them when the modeling is done. The JTS library offers functionality to store and load its geometry objects (that represent the not necessarily simple polygons) using the WKB (Well Known Binary) format. Its C++ port, GEOS, offers the same functionality, so this format is used to store and load the merged free areas and the merged parks.

Precision problems

When determining the visible nodes of a start or destination that lays in a free area, precision problems with the GEOS library occur. To test whether two points p_1 and p_2 are visible to each other in an area a, a geometry object l representing the line segment between p_1 and p_2 is created. The points are visible if a contains all points of l (a.covers(l)). This condition fails for some lines within an area, when the line has one or both of its endpoints on the boundary of the area. To go around this problem, a is removed from l ($l := l$.difference(a)) and the remaining length of l is checked. If this length is smaller than an epsilon value, l is considered to be in a and the two endpoints of l are visible to each other. The epsilon value we use is $\epsilon = 1$cm. The remaining length of l generally ranges from less than 0.01cm to less than a μm, so we consider the failing of the $a.covers(l)$ method to be due to a precision problem.

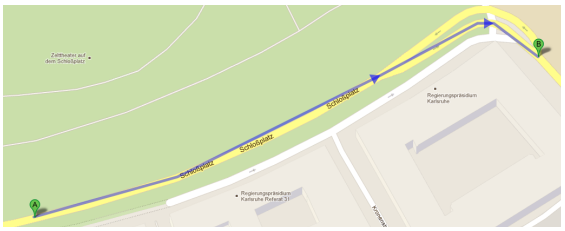
5.2 Comparison to standard routing

As standard routing software for pedestrians *OpenRouteService*[4], *Google Maps*[5] and *Bing*[6] are used. They route in the same way standart routing software for vehicles does – the middle of the street is used, free areas are not taken in consideration and street junctions do not cost extra time. In addition, *OpenRouteService* uses the same input data and has the same map visualisation. Several other route planners for pedestrians that use the same input data were also tried. Their results are not shown, as they are almost identical to the results of *OpenRouteService*:

- *CloudMade*[7]. Its difference to *OpenRouteService* is that the nearest node is chosen for the start/destination, instead of the nearest edge.
- *Routino*[8]. Similar to *CloudMade*.
- *YOURS*[9]. Similar to *CloudMade*.
- *MapQuest*[10]. Some computed routes have odd detours, otherwise similar to *CloudMade*.

The first route query (Figure 5.1) shows the usage of the walkways as opposed to walking on the streets. Aside from knowing which side of the street the pedestrian should use, a very small difference in the route length is also present.

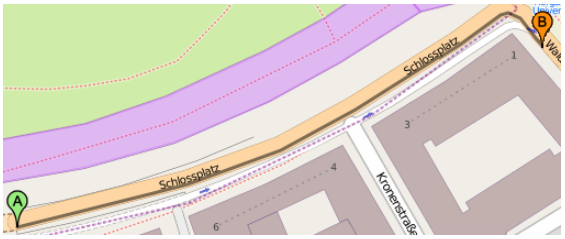
[4] http://www.openrouteservice.org/
[5] https://maps.google.de/
[6] http://www.bing.com/maps/
[7] http://maps.cloudmade.com/#
[8] http://www.routino.org/
[9] http://wiki.openstreetmap.org/wiki/YOURS
[10] http://www.mapquest.com/



(a) A route query computed with *Google Maps*.

(b) A route query computed with *Bing*.

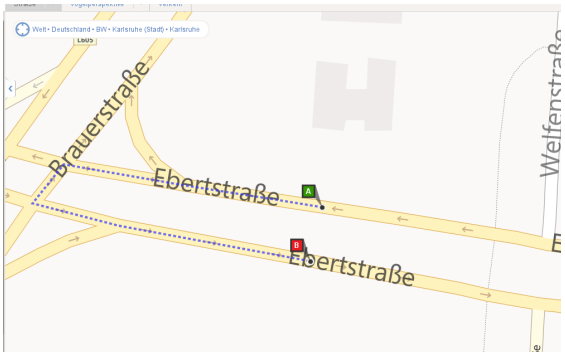
(c) A route query computed with *OpenRouteService*.



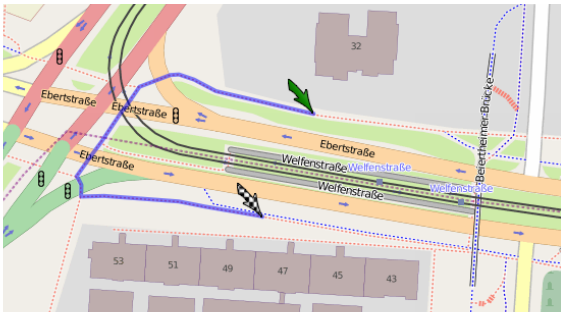
(d) The same route query when using the created walkways.

Figure 5.1: Route difference when using the walkways instead of their streets.

(a) A route query computed with *Google Maps*.



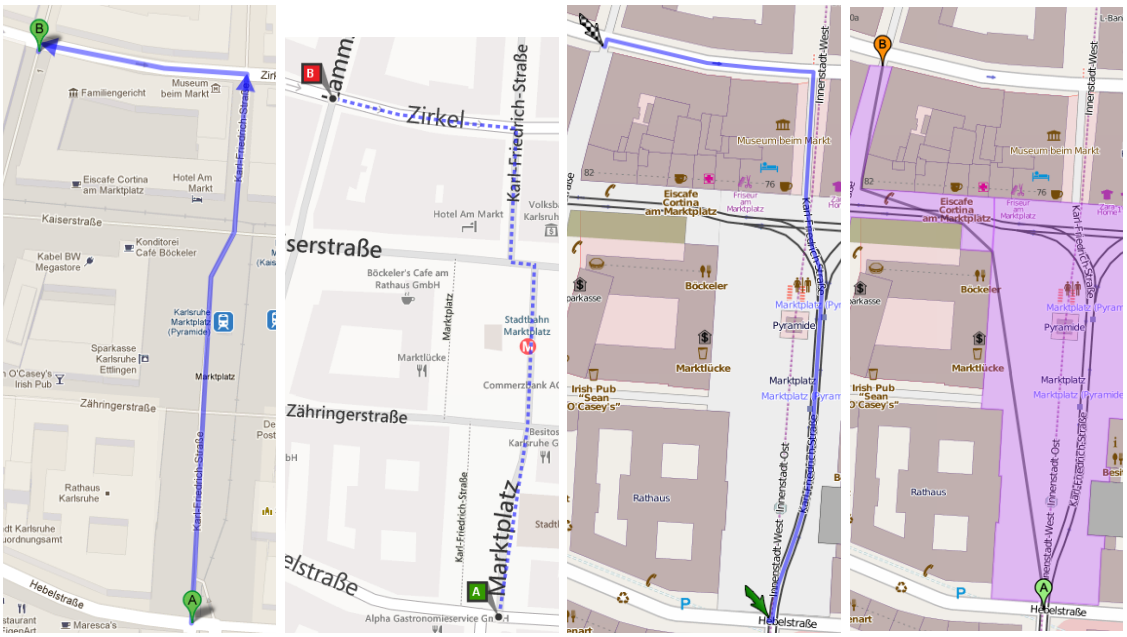
(b) A route query computed with *Bing*.



(c) A route query computed with *OpenRouteService*.

(d) The same route query when adding a small junction penalty.

Figure 5.2: Route difference when using the street junction penalty.



(a) A route query computed with *Google Maps*.

(b) A route query computed with *Bing*.

(c) A route query computed with *OpenRouteService*.

(d) The same route query when using the free area edges.

Figure 5.3: Route difference when using the free area handling.

The second route query (Figure 5.2) shows how the route can differ based even on a small penalty (1 min) added to crossing big street junctions. The route computed with a minute entry penalty uses a bridge over the crossings. Despite the small penalty and the detours of getting on and off the bridge, the traffic junction is avoided. The *OpenStreetService* route crosses four streets with traffic lights, in which case a minute waiting time is usually an underestimation. Ignoring the time needed to traverse such crossings can lead to computed routes that a pedestrian will otherwise avoid, similar to not considering traffic jams in navigational systems for vehicles. The footpaths near the route start and destination are not present in *Google Maps* and *Bing*, leading to a somewhat odd route for a pedestrian.

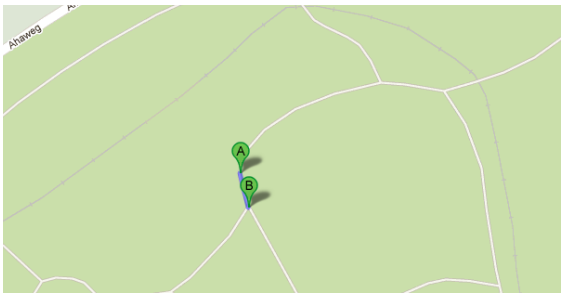
The third route query (Figure 5.3) shows how the route goes through a plaza, instead of going around it. This leads to a faster route and is also something that a pedestrian will generally do. The computed route costs are as follows. *Google Maps*: ∼4 min, *Bing*: ∼4 min, *OpenRouteService*: ∼5 min and ∼3 min when the route is computed with visibility graph edges.

The fourth route query (Figure 5.4) shows how the route can differ when walking in a park. Again, several footpaths are missing in *Google Maps* and *Bing*.

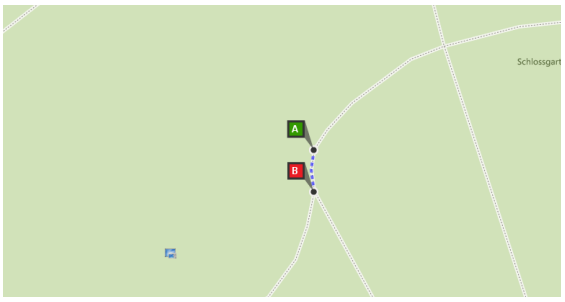
5.3 Modeling statistics

The walkways, penalties, free areas and parks cost additional space. In order to gain some idea about the amount of this space, several figures of the modeling of several cities are given in Table 5.2. In addition, the running time for the different procedures is also given. The row *Other edges* shows the number of edges, other than walkways and visibility graph edges, that are part of the output routing graph.

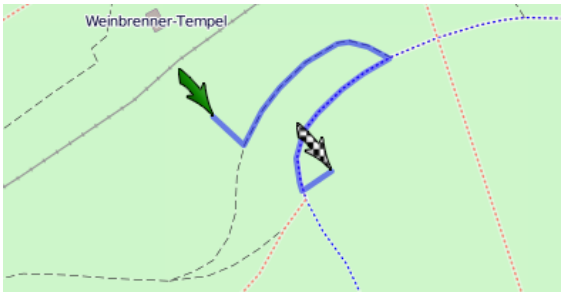
The number of created walkways intuitively grows with the number of edges in the input data. How much it grows can vary, based on the number of streets in the input



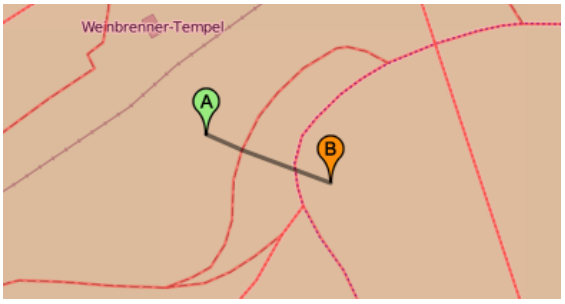
(a) A route query computed with *Google Maps*.



(b) A route query computed with *Bing*.



(c) A route query computed with *OpenRouteService*.



(d) The same route query when considering the park facets.

Figure 5.4: Route difference when using the park handling.

that should have walkways. The average numbers of edges per area and per facet are more dependant on how detailed the descriptions of free areas are in the input data. For instance, the input data for Karlsruhe is near 5 times the size of the input data for Cambridgeshire, even though Cambridgeshire has a significantly higher population and covers a lot more area. Also worth notice is the difference between the edges per area on shortest paths and the total edges per area. In order to route through a free area much less information is needed, compared to starting or ending in a free area. The relatively small number of facets per park shows the presence of many small places marked as parks in the input data. Thus the ϵ value (which defines the ϵ-Neighborhood in which nearby facets are considered) offers a trade-off between how well the few parks with many facets are handled and the time spent for this specific handling.

5.4 Query statistics

Due to the specific handling of start or destination being in a free area or a park, several types of queries are possible. The start and destination can be a node, a point in a free area or a point in a park. The statistics are gathered from 10 000 queries for each different combination. For the queries with a start or end in a free area or park, a random point of a random free area/park is chosen. Table 5.3 shows the running times, as well as the search space covered by the different queries. For the queries the routing graph for Karlsruhe is used, which has 49 023 nodes, 164 648 edges, 702 park facets and 116 free areas.

The column *Vertices* shows the average number of nodes visited by the query and the column *Edges* shows the average number of relaxed edges. The columns *F. area nodes* and *Park edges* show the average number of free area nodes and park edges that are visited before the actual routing algorithm. The free area nodes are tested for visibility and for the park edges special entry points are computed, which are also tested for visibility.

As expected, the search spaces does not vary significantly between the different types of queries. The actual work when the start or destination is in a free area or a park is done

Table 5.2: Modeling statistics (osp. abbreviates *on shortest paths*).

Measure	Karlsruhe	Stuttgart	Cambridgeshire	London
Walkways				
Created walkways	24 918	121 290	285 764	549 920
Procedure time [s]	5.4	35.6	82.0	297.0
Free areas				
Merged f. area	116	293	599	471
Avg. edges per f. area	497.8	180.4	164.6	256.8
Avg. edges per f. area (osp.)	36.9	17.8	7.7	17.6
Procedure time [s]	27.0	128.4	92.4	405.6
Parks				
Merged parks	242	203	825	2625
Avg. facets per park	2.9	6.4	1.7	3.4
Avg. edges per facet	18.2	12.4	12.9	17.0
Procedure time [s]	8.9	35.8	93.5	649.2
Other				
Other edges	81 990	361 102	505 768	1 147 862

Table 5.3: Query statistics (v.v. abbreviates *vice versa*).

Type	Time [ms]	Vertices	Edges	F. area nodes	Park edges
Basic					
Node to node	19.5	23 263	55 908	—	—
Free area					
Node to area (and v.v.)	42.4	22 505	54 256	29.6	—
Area to area	64.6	21 261	51 388	30.1	—
Park					
Park to node (and v.v.)	34.2	22 096	53 079	—	22.6
Park to park	43.0	20 219	48 566	—	22.0
Free area and park					
Area to park (and v.v.)	54.2	21 175	51 011	29.6	22.6

before the Dijkstra algorithm. The cost for this work can be seen in the average running times of the queries. Especially for the free area to free area queries, the processing beforehand takes about 22-23 ms, which is twice as long as Dijkstra's execution time. Additionally, the queries for free areas are more expensive than the park queries – a start or a destination in a park takes from 9 to 15 ms. This can be explained by the fact that the free areas are generally a lot larger than the park facets. Thus, checking whether a location is visible to node in a free area is more expensive than checking if a line is in a park facet.

6. Conclusion

Similar to vehicle route planning, routing for pedestrians has its own set of specific problems. In this bachelor thesis two of those problems were addressed, namely the lack of explicit walkways in free road network databases and the traversal of areas such as parks and plazas.

Feasible solutions to those problems were found and fully described. Most of the computing for the solutions is done during a preprocessing phase and the additional time cost during the actual routing is moderate. It is caused mainly by the special handling of the start or destination being in a free area or park. The additional space cost is moderate and its greater portion is also caused by the information needed to start or end the route in a free area or park.

The designed solutions were then tested in a prototype implementation of a route planner. The result showed that solving the two problems yields a route that is a lot more realistic than a route computed without those solutions.

Several possible improvements to the modeling done in this thesis are: The consideration of building layout when creating walkways, as this would lead to more precise placing of the walkways. A study of how pedestrians walk in parks can be used to better model routes passing through parks. In addition, traffic lights can be modeled in a more precise manner. For the computation of routes, an important improvement to this thesis is the usage of speed-up techniques. Particularly, testing how known speed-up techniques behave on the generated routing graph.

Further problems that can be considered when routing for pedestrians are, for example, the handling of rush hours (similar to traffic jams when routing for vehicles) or the usage of public transportation.

Bibliography

[ADGW11] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks. In Panos M. Pardalos and Steffen Rebennack, editors, *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, volume 6630 of *Lecture Notes in Computer Science*, pages 230–241. Springer, 2011.

[AG92] C. Alexopoulos and P. M. Griffin. Path planning for a mobile robot. *Systems, Man and Cybernetics*, 22:318–322, 1992.

[AOPS02] Ravindra K. Ahuja, James B. Orlin, Stefano Pallottino, and Maria Grazia Scutellà. Transportation Science. *INFORMS*, 36:326–336, 2002.

[BPS11] Miquel Ginard Ballester, Maurici Ruiz Pèrez, and John Stuiver. Automatic Pedestrian Network Generation. *AGILE*, pages 1–13, 2011.

[CGR96] Boris V. Cherkassky, Andrew V. Goldberg, and Tomasz Radzik. Shortest paths algorithms. *Mathematical Programming, Series A*, 73:129–174, 1996.

[dBCvKO08] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark H. Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 2008.

[DGW11] Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Shortest Paths in Road Networks: From Practice to Theory and Back. *it—Information Technology*, 53:294–301, December 2011.

[Dij59] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[DSSW09] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering Route Planning Algorithms. In Jürgen Lerner, Dorothea Wagner, and Katharina A. Zweig, editors, *Algorithmics of Large and Complex Networks*, volume 5515 of *Lecture Notes in Computer Science*, pages 117–139. Springer, 2009.

[Faw00] J. Fawcett. Adaptive routing for road traffic. *Computer Graphics and Applications*, 20:46–53, 2000.

[GM91] Subir Kumar Ghosh and David M. Mount. An output-sensitive algorithm for computing visibility graphs. *SIAM J. Comput.*, 20:888–910, 1991.

[GM04] Thierry Gèraud and Jean-Baptiste Mouret. Fast road network extraction in satellite images using mathematical morphology and Markov random fields. *EURASIP Journal on Applied Signal Processing*, 2004:2503–2514, 2004.

[HB07] Bertrand Haut and Georges Bastin. A SECOND ORDER MODEL OF ROAD JUNCTIONS IN FLUID MODELS OF TRAFFIC NETWORKS. *Networks and Heterogeneous Media*, 2:227–253, 2007.

[MAN04] Ellips Masehian and M. R. Amin-Naseri. A voronoi diagram-visibility graph-potential field compound algorithm for robot path planning. *Robotic Systems*, 21:275–300, 2004.

[MZ07] M. Mokhtarzade and M. J. Valadan Zoej. Road detection from high-resolution satellite images using artificial neural networks. *International Journal of Applied Earth Observation and Geoinformation*, 9:32–40, 2007.

[OIRK87] B. Oommen, S. Iyengar, N. Rao, and R. Kashyap. Robot navigation in unknown terrains using learned visibility graphs. *Robotics and Automation*, 3:672–681, 1987.

[Pet03] R. Peteri. Detection and extraction of road networks from high resolution satellite images. 1:301–304, 2003.

[Sny87] John Parr Snyder. *Map projections–a working manual*, volume 1. 1987.

[TW98] John C. Trinder and Yandong Wang. Automatic Road Extraction from Aerial Images. *Digital Signal Processing*, 8:215–224, 1998.