Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Master's Thesis

A Hybrid Algorithm for Finding Shortest Paths Between Arbitrary Coordinates using a Combination of Network and Geometric Routing

Submitted by:
Hauke Stieler

Course of study:
M.Sc. Informatics

Matr.-Nr.:
6664494

1st Reviewer:
Dr. Fabian Panse

2st Reviewer:
Martin Poppinga

Supervisor:
Daniel Glake, Martin Poppinga

Universität Hamburg
Faculty of Mathematics, Informatics and Natural Sciences
Department of Informatics
Databases and Information Systems

Hamburg, July 31, 2023

Abstract

Determining the shortest path between two locations is a common problem in real-world and scientific applications, such as agent-based simulations. It is usually done by graph-based algorithms, which are widely used and highly optimized but limited to the edges of the underlying graph, which usually represents a network of roads and ways. This also means that graph-based algorithms cannot reach every possible location but only locations on edges of the underlying graph. Finding shortest paths between arbitrary locations in open spaces, meaning without using roads and ways, is the task performed by geometric shortest path algorithms and is especially useful for agent-based pedestrians models simulating human behavior. However, not considering roads with their potentially helpful information, e.g. surface conditions or access restrictions, likely yields unrealistic routes.

In this thesis, a new routing algorithm is presented to enhance the pathfinding component of agent-based simulations. This is done by combining graph-based and geometric shortest path approaches to create a flexible routing algorithm that can reach arbitrary locations, traverse open spaces and is still able to use detailed road data. This is done by generating a visibility graph and merging road edges into it, allowing routing algorithms to switch between these edges at their intersection points.

An evaluation of the performance and route quality is conducted using artificial and real-world datasets. The performance analysis shows the expected quadratic runtime complexity as well as the effectiveness of the implemented optimizations. An analysis of the route quality shows, that it heavily depends on the quality and completeness of the input data and chosen weight function for the graph-based routing algorithm. Assuming complete and high-quality data, the routing approach presented in this thesis yields realistic routes and is therefore beneficial for the pathplanning routine of agent-based models.

Contents

1 Introduction

1.1 Motivation

Traveling through a network of roads and paths often requires a computer system determining the optimal route from a source to a destination location. The software that solves this problem is often called a *routing engine* and uses shortest path algorithms. A graph is used as the underlying data structure, which abstracts the real world into vertices and edges, representing roads and ways. Detailed information, such as surface conditions or the number of lanes, is usually added to the geometries of the graph via attributes, usually in form of key-value pairs.

Algorithms for graph-based routing, such as Dijkstra or A*, are optimized and used in numerous scientific and real-world applications, including this work. To enhance performance in large networks, several speedup methods exist that further decrease query times allowing global routing requests to be answered within milliseconds.[1]

Aside from line-based shapes, which are very suitable for routing, the real world also consists of polygonal areas, which can not be abstracted to a single line, such as marketplaces, parking lots or parks. Classical routing algorithms can therefore only find routes along the outer edges of these polygons. To cross these open areas, auxiliary paths are added to connect opposite sides of polygons, which is a common practice in OpenStreetMap, a crowdsourced geospatial database. A few lines can be added by hand, but manually filling large areas or even a whole city with these auxiliary paths is not feasible. However, geometric routing algorithms exist that find so-called euclidean shortest paths across open spaces in the presence of impassible obstacles, such as buildings or walls. There are two main strategies for geometric routing, which will both be covered in detail in Section 2.3.

A similar problem arises with the source and destination locations of real-world routing. These locations are often not part of roads or ways and therefore the underlying routing graph, which is the case for most addresses, points of interest (*POIs*) or manually chosen locations. However, only objects in the routing graph are reachable by graph-based routing algorithms. Simple approaches to connect points to a graph (e.g. connecting to the nearest vertex) are often inaccurate. Such inaccuracies arise because the space between an arbitrary location and the road network may contain obstacles, such as walls or buildings.

Even if only vertices of the routing graph are chosen, the quality of graph-based routing results is limited due to the abstractions of the road network. Missing sidewalks, inaccurate geometries and missing connections between parallel ways reduce the usefulness of graph-based routes, especially for routes determined for pedestrians.

[1] Tested for a routing request from Hamburg, Germany to Beijing, China on openstreetmap.org using the OSRM car profile which is based on Contraction Hierarchies (s. Section 2.2.3). The HTTP request returned the result in under 215 ms – including network latency of several milliseconds.

1.2 Problem and contributions

The previous section mentions one of the disadvantages of graph-based routing: No assumptions can be made on whether source and destination locations are represented as vertices in the graph. Furthermore, trajectories of pedestrians in the real world often do not follow edges on routing graphs, which is especially the case for insufficiently connected open spaces. Routes strictly following the edges of a graph are therefore inaccurate for pedestrian routing [1], which means the calculated route contains detours and, thus, is longer than the actual trajectory and additionally may not reach the exact destination location at all. Such inaccuracies significantly affect the quality of agent-based simulations, as well as the helpfulness and user experience of real-world applications, such as mobile navigation apps.

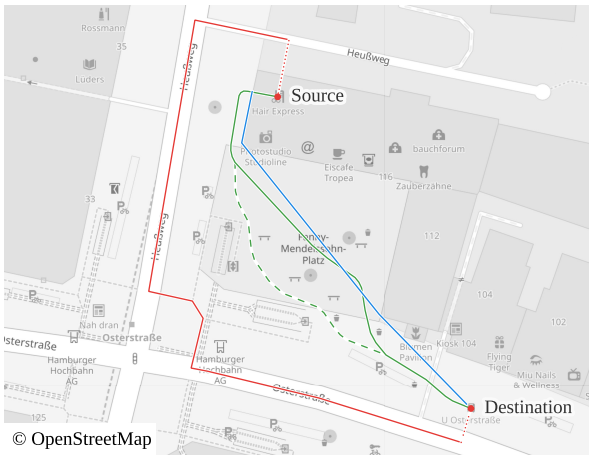


Figure 1: Comparison of an expected and realistic route (green) with a graph-based route (red) and an actual result of the algorithm presented in this work (blue).

In this thesis, these issues are solved by proposing and designing an algorithm, called the *hybrid routing algorithm*, that combines the accuracy of geometric routing using visibility graphs with the optimizations and wide use of network-based routing. Additionally, an implementation of the proposed algorithm was created and evaluated regarding performance, correctness and usefulness. Determining shortest paths using the resulting approach has a simple structure, enables the use of common speedup techniques and reaches all accessible locations independent of the presence of corresponding vertices in the graph. Any other arbitrary location, for example, a location chosen by the user, can be added to the graph. Routing from and to this vertex is possible after connecting the newly added vertex to the graph. Real-world applications and agent-based models, which are used to simulate the behavior of pedestrians, benefit from this algorithm.

1.3 Structure of this thesis

First, the basics of spatial data, data formats, graph routing, geometric routing and agent-based simulations are covered in Chapter 2. The scientific work related to this thesis is presented in Chapter 3 for graphs and networks, routing techniques and pedestrian pathfinding in particular. In Chapter 4, the design of the system developed for this thesis is described and an overview of the different components and elements is given. A more detailed view of the implementation is presented in Chapter 5, including technical aspects of the combination of network and geometric routing as well as used frameworks. The hybrid routing algorithm is then evaluated in Chapter 6 regarding its performance, correctness and quality of the resulting routes.

2 Fundamentals

In this chapter, basic concepts are described, which are relevant for this thesis. Primarily, this includes geodata, routing algorithms as well as agent-based simulations. The section about geodata gives a broad overview of coordinate systems, file formats, standards and OpenStreetMap, which is the geodata source in this thesis. Geodata can generally be used to find the shortest paths between two locations using routing algorithms. Two main strategies exist to find such paths, graph-based and geometric routing, and both will be covered in this chapter. Agent-based simulations play an essential role in pedestrian simulations and often use these routing algorithms for the agent's movements. Hence, the topic of agents and simulations will be covered as well.

2.1 Geospatial data

According to the ISO standard 19109:2015 [2], the term *geospatial data* refers to "data with implicit or explicit reference to a location relative to the Earth". In other words, data with an address, coordinate or any other kind of location reference belongs to the class of *geospatial data*, or *geodata* for short.

Examples are customer information (address information of each customer), meteorological data (measured temperature and humidity with a coordinate and altitude) and, primarily used in this thesis, road and path data suitable for routing. Addresses are less important for this thesis, as they are not globally standardized, only describe a vague location and usually belong to a whole area. Geodata may have an additional time dimension and is therefore categorized as *spatiotemporal data* [3]. Even though the time dimension is relevant in many real-world use cases, this type of data is of less importance for this thesis as well. This thesis uses the term *geodata* to refer to data structures described below in Section 2.1.1.

Besides the mentioned explicit references to locations, such as coordinates, implicit location references can also be encoded within the metadata of datasets. One example is the image-based GeoTIFF format described in Section 2.1.3.

2.1.1 Data structures

Spatial data can be stored in a variety of data structures. Some are used in many frameworks, libraries and formats, but some are vendor specific. An overview of common data structures in spatial data is given in this section.

The *Simple Feature Access* (SFA) standard by the Open Geospatial Consortium (OGC) defines numerous geometry types [4]. Not all of them are implemented in all data formats, and most are irrelevant for this thesis. The GeoJSON file format (described in more detail in Section 2.1.3) implements the most important geometries and data structures of the SFA standard and is frequently used in this work. This section gives an overview of these geometry types and an introduction to the concept of features.

Geometries

A *geometry* only represents geometric coordinates and their relation to each other. Common types of geometries are:

Point Simple geometry type with only one coordinate.

LineString An ordered list of multiple coordinates building a line with a certain direction.

Polygon An ordered list of coordinates of which the first and last coordinates are equal and thus form an area. A polygon might consist of additional polygons inside of it, forming holes. Polygons inside such holes form islands, which can have holes again.

Multi-Geometries Grouping geometries of the same type together forms multi-geometries, such as `MultiPoint`, `MultiLineString` and `MultiPolygon`. They can be used to share properties of the contained geometries.

Features

A *feature* is an abstraction of a world phenomenon, for example, a tree, road or lake. From a technical perspective, a feature consists of a geometry, describing *where* the feature is, and attributes (also called "properties" or "tags"), describing *what* the feature represents. Different file formats have different properties regarding geometries and attributes, which will be described in the following sections.

Standardization

Standards exist to keep a data source consistent and allow system interoperability. Common standards were defined by organizations like the OGC as well as companies or government agencies.

A widely used proprietary standard established by Esri Inc. is the Shapefile file format, which will be covered in Section 2.1.3. One example of a governmental standard is the *INSPIRE* standard based on an initiative of the European Commission[1]. Next to proprietary or governmental standardizations, the OpenStreetMap project uses a system based on proposals and democratic votes, which is further described in Section 2.1.6.

2.1.2 Spatial indices

Storing data for efficient access, e.g. to query all features within a particular area, a so-called *spatial index* is used. Some commonly used indices are the following.

The *k-d tree* [5, Ch. 19, p. 4] divides the space at each vertex along one axis at a time alternating through all axis (dimensions) of the dataset, which leads to a binary tree for 2-dimensional data. A *quadtree* [5, Ch. 19, p. 1] works on 2-d data and divides the space equally into four equally sized quadrants. Depending on the data in each quadrant, it gets divided again, which leads to a tree with four children per node. The *R-tree* [5, Ch. 21, p. 2] determined bounding rectangles for areas with dense data and efficiently stores a hierarchy of these areas, i.e. all features of child nodes are within the bounding rectangle of the parent node.

[1] `https://inspire.ec.europa.eu`

In the implementation of the hybrid routing algorithm, some of these indices are used to enhance performance, as presented and analyzed in the later chapters of this work.

2.1.3 File formats

Spatial data can be stored in various file formats with different properties and use cases. This section gives an overview of some popular formats.

GeoJSON

Another open but not OGC-standardized format is *GeoJSON*, which is the most important format for this thesis as all used datasets were in this file format. As the name suggests, a GeoJSON file follows the JSON specification defined in RFC7946[2]. The file can either contain a collection of features (of type `FeatureCollection`) or geometries (of type `GeometryCollection`, which itself is a geometry type). Each feature in a `FeatureCollection` consists of a geometry and, in contrast to elements in a `GeometryCollection`, attributes adding additional information to the feature. The number of geometries per feature as well as their length, number and character choice of attributes is not restricted.

GeoTIFF

All formats mentioned so far store vector data, but raster data can also be georeferenced. One image-based format to store such raster data is the OGC-standardized *GeoTIFF* format [6], which is a TIFF formatted image with spatial metadata such as the coordinate system and a mapping of pixel to coordinates. Typical use cases for such images are aerial or satellite imagery and digital elevation models (DEM). However, other use cases are thinkable, such as field-based routing algorithms as mentioned in Section 3.3.1 using raster data to store the vector fields.

Other formats

Other formats are for example *Well-known Text* (WKT), consisting of the geometry type followed by a list of coordinates [4, p. 51], *GeoPackage*[3], consisting of a SQLite database with support for spatial functions and indices, or the widely used and proprietary *Esri Shapefile* format [7]. However, numerous additional formats exist for general and special use cases.

2.1.4 OGC web services

Besides file-based formats, web-based standards exist as well. The OGC standardized some APIs for web-based services. Popular standards are the *Web Map Service* (WMS), *Web Map Tile Service* (WMTS) and *Web Feature Service* (WFS). These services are only used to interact with the raw data but do not offer higher-level functionalities such as routing.

[2] `https://datatracker.ietf.org/doc/html/rfc7946`
[3] `http://www.geopackage.org`

2.1.5 GIS

A file format alone, as described in Section 2.1.3, is only useful when storing or transferring data and to ensure interoperability. Processing the data is part of applications referred to as *geographic information systems* (GIS).

The term GIS is very broad and describes nearly all systems working with spatial data. This includes databases like Postgres with the PostGIS extension, software libraries like the NetTopologySuite, servers like GeoServer and desktop applications like ArcMap or QGIS. Such desktop applications often combine multiple functionalities to visualize, analyze or otherwise process the data.

2.1.6 OpenStreetMap

OpenStreetMap (OSM) is a publicly available and freely accessible geospatial database that is maintained by a global community and licensed under the *Open Data Commons Open Database License* (ODbL) [8]. It can therefore be used, changed and redistributed as long as proper attribution is given and results stay under the ODbL[4]. All spatial data used in this thesis is data from the OSM database. In 2006, two years after the OSM project started, the OpenStreetMap Foundation[5] was established to perform fund-raising, maintain the servers and also act as a legal entity for the project. People contributing to OSM are called *mappers* or *contributors*, often working voluntarily and mapping their local vicinity or concentrating on specific topics.

Data model

The model of OSM is much simpler compared to many of the aforementioned data formats. There are three main data types in OSM: *node*, *way* and *relations* [9].

Nodes and ways are analogous to `Point` and `LineString` from the OGC Simple Feature Access (SFA) specification described in Section 2.1.1. Areas also exist but are modeled as closed ways and therefore do not form a new type of geometry. A way is closed when the first and last coordinates are identical and specific area-compatible attributes are set.

Relations form a collection of features, so they correspond to `MultiPoint`, `MultiLineString` and `MultiPolygon` of the SFA specification. Typical use cases are multipolygons, road segments involved in turn restrictions and ways included in a bus route.

Attributes

Attributes are called *tags* in OSM, are simple unrestricted key-value pairs and mostly describe properties of features. For example, a restaurant (`amenity=restaurant`) may have additional tags such as the name, address and opening hours.

Some tags, like `area=[yes|no]` or the relation-specific `type`-key, influence the type of geometry. A closed way with `area=no` does, in fact, *not* represent a polygon, but just a closed linestring. This not only affects visualizations but may also need to be considered in other processing and analysis tasks. Other tags add metadata to objects, for example, the last time the feature was surveyed, the source of the data or internal notes to other mappers.

[4] `https://opendatacommons.org/licenses/odbl`
[5] `https://osmfoundation.org`

The OSM community organizes the standardization of tags in the project's wiki via a special proposal process [10]. To ensure flexibility, it is explicitly allowed to use new, reasonable and unstandardized tags, which may become de facto accepted if they are commonly used. Proposed tagging scheme changes to not only get accepted or rejected by a democratic vote, a mandatory public discussion of at least two weeks needs to take place prior to voting. Due to the vast amount of contributors, different opinions, use cases, editors and knowledge about the tagging schemes, multiple competing schemes evolved for some attributes (for example `phone=*` and `contact:phone=*`). Tagging schemes change over time and some may become deprecated or are officially abandoned via a proposal, which might lead to outdated tags on objects.

Contributions to OSM

Uploads to OSM always happen in so-called *changesets* combining multiple changes on the map [11]. Like features, each changeset itself can contains tags adding metadata to the change. Tags like `created_by` (containing the username), `comment` (describing the change) and `source` (data sources like aerial imagery or manual survey) are the most common ones but editors may add additional information. The `source` tag is particularly important since the ODbL is not necessarily compatible with licenses of other data sources and this tag helps to verify this compatibility. Ideally, a changeset should be coherent, which means that it should focus on one thematic aspect in one local area, but this is not always the case and some editors automatically split larger changesets into smaller ones.

Data contained in OSM

OSM has no focus on specific topics and nearly anything can be added to OSM due to the flexible tagging scheme. However, some types of features are very common according to *taginfo*, an online service providing daily updated statistics on keys and values in OSM[6]. According to these statistics, the most common objects are buildings with over 542 million occurrences. Furthermore, 6% of all objects and nearly 60% of all ways are buildings. Highways (mainly roads and streets but also paths, bridleways, railways and more) are the second most common features with over 219 million ways (23% of all ways). Addresses are also very common: about 32% of all nodes and 7% of all ways (mainly building) have a house number. Other often added area features are forests and lakes as well as line features like barriers and waterways.

Data *not* contained in OSM

Even though the tagging scheme can be extended arbitrarily, some data will probably not be added to OSM, which can have multiple reasons:

- Some data is too detailed and therefore mappers are unlikely to invest the time necessary to create or maintain that data. For example, features representing historic buildings are made of long straight lines even though the real building facade is detailed and richly designed.
- Data that cannot be verified on the ground will likely not be added to OSM, because anyone visiting the data in the real world should be able to verify its existence and correctness.

[6] `https://taginfo.openstreetmap.org/keys`

- Temporary data should not be added since OSM is not a real-time database. However, there is no strict definition of when a feature is temporary and when it is (potentially) permanent.
- Data under an incompatible license will not be added. This also includes data from many public authorities and nearly all companies.
- Private areas are often not part of OSM or contain only a few details. This is mainly because access to private areas is usually restricted, rendering them unverifiable for unauthorized mappers.

Unfortunately, data relevant to routing is often added to ways representing roads and paths but not to areas. This includes primarily accessibility and surface information. However, the latter can sometimes be inferred or approximated from other tags on a polygon, for example, `surface=grass` from `leisure=park`, but there is always the possibility for errors.

Data used in this thesis

This thesis uses OSM data for routing purposes. Other sources of spatial data exist but are non-free (in terms of price and the license of the data), fragmented over multiple sources (files, servers, APIs) or not as detailed as OSM.

One example of such a fragmented source is provided by the state office for geoinformation and surveying (Landesbetrieb für Geoinformationen und Vermesseung – LGV) in Hamburg, Germany. Many datasets can be downloaded from the official website geoportal-hamburg.de but roads and building data are contained in separate datasets, primarily "HH-SIB" and "ALKIS". In addition to the fragmentation, some of data is not suitable for routing. The "HH-SIB" dataset contains numerous line segments, which represent ways and roads, that are not connected to the remaining road network. The "ALKIS" dataset also contains road data, but only as areas, which is not suitable for routing as well. However, OpenStreetMap offers free access to numerous usable features in one dataset.

Since geometric routing is the major topic, not only the road network is used, but also certain features of OSM representing impassable obstacles, which are avoided during routing:

- **Barriers**: The `barrier`-tags describe non-passable barriers but also passable structures like gates. Thus, only specific barriers (mainly fences and walls) are considered.
- **Buildings**: The `building` tag describes any type of building. However, values such as `building=roof`, `=no` and `=demolished` are not considered.
- **Natural areas**: Areas with the `natural`-key describe areas filled with a specific form of vegetation as well as water areas. All natural areas, except grass-covered land, are considered.
- **Railway**: Any type of railway infrastructure (`railway=*`) is considered.
- **Waterways**: All waterways, such as rivers, canals or ditches, are considered.

The exact choice of the passable and impassable features is debatable or depends on the simulation's domain. For example, train tracks might be passable in an evacuation scenario.

The road network is also used, which is done by importing all features with a `highway` key. Features such as stop signs or streetlights, which are not part of the road network, are not removed because they may contain information useful for routing.

2.2 Graph-based routing

2.2.1 Theoretic considerations

Routing refers to the process of finding the optimal (often shortest) path between two given locations. Graph-based routing takes place on a graph $G = (V, E)$ with V being the set of vertices and $E \subseteq V \times V$ being the set of edges connecting the vertices [12, pp. 643–644]. To goal is to find a path from a source $s \in V$ to a destination (or target) $t \in V$ of minimal weight using a weight function $w : E \to \mathbb{R}$. A path $p = \langle v_0, v_1, \dots v_n \rangle$ is an ordered list of connected vertices $v_0, v_1, \dots, v_n \in V$, which means for two consecutive vertices v_i and v_{i+1}, $(v_i, v_{i+1}) \in E$ must hold.

Often the shortest path is to be determined, which is why the weight function is used to calculate the length of an edge. Even though the following sections refer to shortest paths, determining "optimal" (or minimal) paths is the general case. Thus, routing is a minimization problem, trying to find a path p of minimum weight $w(p) = \sum_i w(v_i, v_{i+1})$.

In theoretic computer science, the fundamental problem behind routing algorithms is called the *shortest paths problem*, which exists in the following variants.

Single-source shortest paths

One source vertex is given and the shortest paths to all other vertices should be determined. Algorithms solving this problem are, for example, the Dijkstra algorithm, described in detail in Section 2.2.2, or the Bellman-Ford algorithm, which can additionally handle negative edge weights [12, p. 651].

Single-destination shortest paths

This is the opposite of the above problem. All shortest paths to a specific vertex from any other vertex should be found. However, no new algorithms are needed to solve this problem. Instead, the direction of each edge can be reversed, turning this problem into the single-source problem.

Single-pair shortest paths

The shortest path from one specific source to one specific destination vertex should be determined. Even though there are specialized algorithms for this problem like TRANSIT (described in Section 2.2.3), more general approaches to solve the single-source problem can be used as well, such as the Dijkstra algorithm as presented in Section 2.2.2.

All-pair shortest paths

This problem is similar to the single-source problem, but every vertex is a source vertex. A naive approach would likely use an algorithm to solve the single-source problem of each vertex. However, there are faster ways to solve this, such as the Floyd-Warshall and Johnson's algorithm [12, pp. 693, 700] as well as PHAST [13] with a complexity of $\Theta(|V|^2)$.

2.2.2 Routing engines and shortest path algorithms

The term *routing engine* refers to software built to find an optimal route between a source and destination location. The optimality of this route is determined by a routing profile, which is a

weighting function assigning a weight to each edge in the input graph. Routing engines might perform other operations as well, such as the calculation of *isochrones*, which are a way to visualize the reachability of a region [14]. Each isochrone is an area reachable from a given source location within the same amount of time, which is determined using routing.

To find desirable paths, the weight function w mentioned in Section 2.2.1 needs to be carefully chosen based on the domain of the application, vehicle type, personal preferences and other influencing factors. Navigation applications in everyday life bundle these aspects into a *routing profile*, which maps attributes from the input data to a certain weight. Input data might consist of multiple sources like a road network, a digital elevation model or additional traffic information. The open-source software GraphHopper is one example of a routing engine and comes with predefined profiles for different modalities and situations. For example, the `car_delivery` profile is made for car-based delivery services enabling the use of private roads [15], which are avoided in the normal `car` profile.

To enhance performance, speedup methods were developed, allowing routing queries of continental or even global sizes to be answered quickly. Some popular methods are, among others, *contraction hierarchies* and *landmarks*, which will both be described with more details in Section 2.2.3.

Dijkstra

Dijkstra's algorithm, often just called *Dijkstra*, is originated in the 1950s and can be used to solve the single-source shortest paths problem. When no destination vertex is given, it creates a spanning tree rooted in the source vertex s containing solely shortest paths to all other vertices. Despite its age, Dijkstra's algorithm and optimized versions of it are frequently used in science and real-world applications. Algorithm 1 presents an enhanced version using a priority queue for the vertices [12, p. 658].

Algorithm 1 Pseudocode of Dijkstra's algorithm using a priority queue and starting at vertex $s \in V$.

```
1:  for all vertices v ∈ V do
2:      Set shortest path distance v.d = ∞ and predecessor vertex v.π = undefined
3:  end for
4:  s.d = 0
5:  Insert all vertices into min-priority queue Q with v.d being the priority
6:  S = ∅
7:  while Q ≠ ∅ do
8:      Get closest vertex u = Q.min
9:      for all adjacent vertices v of u do
10:         Add u to S
11:         if the path to v via u is shorter than v.d, so if u.d + d(u, v) < v.d then
12:             Set v.d = u.d + d(u, v) and v.π = u
13:         end if
14:     end for
15: end while
```

Once a vertex is taken from the queue, it is considered as *visited* by adding it to S. It can be proven that for each visited vertex v, the distance $v.d$ after the last iteration of the inner for-loop is optimal [12, pp. 659–661]. This also means following back the predecessor relation $v.\pi$ yields the shortest path. When u is equal to a destination vertex t, the algorithm can safely be stopped after line 14 since the optimal path from s to t has been found.

A*

The *A** shortest path algorithm was introduced in 1968 and uses a heuristic $h : V \to \mathbb{R}$ in combination with a distance function $g : V \to \mathbb{R}$ to decide which vertex to visit next [16]. The heuristic h estimates the distance from a vertex v to the given destination (or target) vertex t. The distance function g yields the already known shortest path distance from the source s to v. Summing up the two functions to $f(v) = g(v) + h(v)$, gives the estimated length of a shortest path from s via v to t and is used to decide which successor of v to process next. A crucial criterion to h is that it never overestimates the distance to t, which is especially important for the triangle inequality used by the ALT speedup method presented in Section 2.2.3. The structure of A* is relatively simple and is expressed in pseudocode in Algorithm 2.

Algorithm 2 Pseudocode of the original A* algorithm [16] with length estimate function $f : V \to \mathbb{R}$.

1: Initialize empty output graph G
2: Mark s as *open* and set $u = s$ since s is the only open vertex
3: **while** $u \neq t$ **do**
4: Mark u as *closed*
5: Store u with an edge to its predecessor to the output graph G
6: Mark all non-closed successors of u as *open*
7: Mark each closed successor u' of u as *open* when $f(u')$ became lower since the last visit of u' where it has been marked as *closed*
8: Select open vertex u with minimal $f(u)$
9: **end while**
10: Take G, follow t back to the source s and output the path

The performance of A* relies heavily on the quality of the heuristic, therefore no general complexity formula can be given [17]. However, for a specific problem, the so-called *effective branching factor* b^* can be determined, which approximates the number of new open vertices per processed vertex (see line 6 and 7 in Algorithm 2). The optimal branching factor is 1 and a good heuristic has a factor of close to 1. Having b^* and the number of vertices n in the shortest path, a general runtime complexity of $\mathcal{O}((b^*)^n)$ can be assumed. Since all closed vertices are stored in the output graph G, the space complexity is equal to the time complexity. Therefore, a bad heuristic can turn A* into an algorithm of potentially unmanageable exponential complexity. However, A* is fast in practice and used frequently in science, real-world applications as well as in this thesis.

2.2.3 Speedup methods

Speedup methods reduce the number of processed vertices and edges during routing, primarily by transferring computationally complex tasks into a preprocessing step.

Contraction Hierarchies

Contraction hierarchies are based on the idea of reducing the number of vertices a routing algorithm has to visit by adding so-called *shortcut edges* [18]. Consider a vertex v with incoming edge (u, v) and outgoing edge (v, w), meaning there is a path from u via v to w. Vertex v is *contracted* by adding a shortcut edge $e = (u, w)$ with weight $w(e) = w(u, v) + w(v, w)$. If e already exists with a higher weight, its weight is decreased accordingly.

A crucial part of this technique is the selection of vertices to contract [18, p. 14], which are selected by a total order defining the level of importance for each vertex. Creating a good order is difficult and heuristics are used to approximate an optimal ordering.

Routing then uses an adjusted bidirectional Dijkstra algorithm [18, pp. 29–30] on two graphs, an upward and downward graph. The upward graph contains only edges from lower to higher order vertices and the downward graph only from higher to lower order vertices. Two sub-queries are performed, one in the upward and one in the downward graph, which terminate if they meet in the same vertex and no lower weighted edge towards an unprocessed vertex exists. All shortcut edges must now be recursively relaxed, i.e. replaced by the original edges used to create the shortcut, to obtain the actual shortest path.

Landmarks

A widespread usage of so-called *landmarks* is the ALT algorithm, which stands for A*, landmarks and triangle inequality [19]. The heuristic in A* is usually the Euclidean distance to the destination, which is simple but inaccurate. A better, but also computationally more complex approach is the use of landmarks $L \subseteq V$ and for each $l \in L$ to precompute the distances to and from all other vertices as illustrated in Figure 2. The heuristic takes all landmarks into account and uses the triangle inequality between the source, destination and landmarks to determine the tightest bound on the estimated shortest path distance. The standard A* algorithm can then use this heuristic to determine good choices for the next vertex.

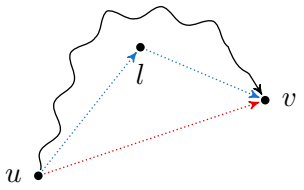


Figure 2: The heuristic using landmark l (blue) is a longer but better estimate than the Euclidean heuristic (red).

Other speedup methods

There are numerous other speedup methods, such as the following methods.

Filtering out unnecessary edges during routing can be done with the use of so-called *arc flags* [13]. In a preprocessing step, the graph is subdivided into K cells of roughly equal size. An edge e, also called *arc*, has a flag consisting of K bits where the i-th bit is 1 if e is on any shortest path to any vertex in the cell i. When answering shortest path queries with the destination being in cell j, all edges without the j-th bit set to 1 can be ignored. Arc flags significantly reduce query times and can be used with other speedup methods to further enhance performance.

Precomputing shortest path distances is done by the *hub label* strategy [13]. For every vertex v, a set $H_v \subseteq V$ of *hubs* is selected such that any shortest path between two vertices contains at least one hub. Then labels $L(v)$ are attached to v containing the lengths from v to all its hubs. A linear search through the common labels of two vertices yields their shortest path distance. Using hub labels as the heuristic in A* yields a fast routing algorithm, but precomputation requires much time and space.

Compressed path databases

A *compressed path database* (CPD) itself is not directly a routing algorithm but rather a technique to efficiently store shortest paths between any two locations [20]. Grids are used in the following, but CPDs can also be created on graphs. For a grid of n cells, n many such grids exist in the CPD, each

belongs to a cell c and stores the information on how to get from c to each other cells in the grid as illustrated in Figure 3. This information is a movement operation, e.g. a target cell t containing the operation "move left" means the left cell of c is the next cell on the shortest path from c to t. After performing the movement operation and getting from c to c', the next movement towards t is stored in the movement grid of c' in cell t. These steps are performed until t has been reached, which means answering shortest paths queries is a linear time operation.

The complex part of constructing a CPD is the compression [20] of the all-pairs shortest path solution. Uncompressed data is usually too large as the space requirement is in $\mathcal{O}(n^2)$ or even $\mathcal{O}(n^2 \log n)$ for graphs. A combination of compression techniques can be used to get sufficient results. This includes the encoding of whole areas with the same movement operations, run-length encoding (RLE) and the use of default values to reduce the number of explicitly stored values [20]. Compared to uncompressed data, a compression factor of 950 was achieved for a road network.

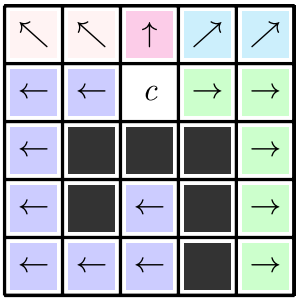


Figure 3: Movement-grid for cell c in a grid-based CPD [20].

2.3 Geometric routing

Finding paths in a geometric domain, also called the *euclidean shortest path* problem, is the process of determining shortest paths without a graph but through open spaces avoiding obstacles. There are two main strategies for finding Euclidean shortest paths: Generating edges for graph-based routing or creating a shortest paths map, a structure similar to CPDs.

2.3.1 Visibility graphs

The first mechanism uses a standard graph-based shortest path algorithm, but the edges in the graph are generated using one of multiple possible approaches. One approach creates a so-called *visibility graph*, which is a normal graph with edges between vertices that are visible to each other. Or in other words, for any two $u, v \in V$ an edge $(u, v) \in E$ exists if and only if no obstacle intersects with this edge. A visibility graph might become very large, the worst-case regarding the size is a complete graph with $\mathcal{O}(|V|^2)$ many edges.

Alternatives to the visibility graph generation are based on Voronoi diagrams or skeletonization methods. However, routing results on such graphs are not necessarily optimal [1], since a straight edge between visible vertices is the shortest possible connection.

Source and destination locations must be part of the graph and can be connected with the same algorithm used to create the graph itself.

2.3.2 Continuous Dijkstra paradigm

The second strategy, the *continuous Dijkstra paradigm*, creates a map of regions, similar to CPDs, starting from a source vertex s. Regions are characterized by the fact that all vertices within a region have the same predecessor on the shortest path from s. Following back the predecessors yields the shortest path from s to a given location. This map can also be seen as a tree with root s and each region as a node, which might remind one of Dijkstra's algorithm giving this approach its name [21].

To precisely determine these regions, so-called *wavefronts* propagating through open spaces are used. A well-fitting analogy for this approach is the propagation of water waves traveling through a 2D space, folding around obstacles, colliding with each other and finally reaching the destination location. Collisions between wavefronts and obstacles as well as collisions between two wavefronts determine the regions' boundaries with the source of the wavefront as its predecessor. After detecting such a collision, the angular region of the wavefront is adjusted depending on the type of collision. The fundamental difficulty of this approach is the efficient detection and handling of collisions. Efficient algorithms are presented in Section 3.2.1.

2.4 Agent-based systems and simulations

Simulating complex systems with numerous autonomous individuals is difficult, but using so-called *agents* with independent behavior and decision-making breaks down this complexity [22]. Interactions between two agents and between agents and the environment are essential to this.

Modeling agents can be arbitrarily complex because agents make decisions independently without any central controlling unit. More sophisticated agents may have a specific goal, can adapt to the environment and thus must be able to memorize and plan things.

Agents usually model humans but can represent non-living things like companies or cars. Many simulations use pedestrians (or generally humans) as agents to better understand or utilize human behavior but other scenarios and types of agents are possible as well [22].

Studying pedestrians requires a spatial environment and algorithms finding optimal paths [23, 24, 25]. This includes graph-based routing but may utilize geometric routing as well [23]. In this thesis, these two concepts are combined to enhance the pathfinding component of agent-based simulations.

2.4.1 The MARS framework

This work aims to enhance agent-based simulations created using the framework MARS (Multi-Agent Research and Simulation). MARS is a C#/.NET framework to create and execute agent-based simulations and is developed by the eponymous research group of the Hamburg University of Applied Sciences (HAW) in Hamburg, Germany[7]. The framework provides components, algorithms and tools to create tick-based simulations with multiple agents, different data layers and numerous static entities populating the environment.

MARS contains several data structures and algorithms specifically for pathfinding, such as a spatial graph and the A* algorithm. It also contains numerous spatial indices, mathematical operations and classes to serialize the trajectories of agents.

This work is based on the MARS framework to be directly usable by simulations and uses the mentioned spatial graph and A* algorithm. More details on the design and implementation are given in Chapter 4 and 5.

[7] `https://www.mars-group.org`

3 Related work

In this chapter, an overview of scientific literature from the areas of graph construction, routing and (pedestrian) path planning is given.

3.1 Graph generation

Generating edges to create a graph is an essential topic in this thesis. However, enhancing and filling existing graphs is closely related and use similar or the same mechanisms. Numerous approaches exist and have been used and discussed in the literature. This section gives an overview of some approaches.

3.1.1 Visibility graphs

Constructing a visibility graph is a central part of this thesis. Even though numerous approaches to creating such graphs have been presented over the last decades, the worst-case time complexity of this problem will always be quadratic. This is due to the worst-case graph where all n vertices are visible to each other (complete graph), creating $\Omega(n^2)$ edges in the output graph (precisely $\frac{n^2-n}{n}$, which is the highest possible number of edges in a graph).

The construction of visibility graphs in $\mathcal{O}(n^2)$ time and space (with n line segments), under the condition of no collinear vertices and no intersecting line segments, was presented by Welzl in 1985 [26].

A few years later, in 1988, Overmars and Welzl presented two new methods reducing the space complexity to $\mathcal{O}(n)$ [27] and one of these methods has a time complexity of $\mathcal{O}(E \log n)$ with E being the number of output edges. Their new algorithms are based on Welzl's earlier paper and implement the idea of a rotational sweep through neighboring vertices.

The time complexity of Overmars' and Welzl's approach is output-sensitive with respect to the number of edges in the output graph [28]. Ghosh and Mount presented an algorithm with an output-sensitive complexity of $\mathcal{O}(E + n \log n)$, which is a significant improvement since E is in $\mathcal{O}(n^2)$. However, their approach needs $\mathcal{O}(E + n)$ space compared to $\mathcal{O}(n)$ from the algorithms by Overmars and Welzl. Ghosh and Mount use plane-sweeps to create a triangulation of the free space and introduced so-called *funnel sequences*, which are sets of vertices visible to parts of an edge. To avoid numerous edge cases, their approach assumes no collinear vertices and unique x-coordinates.

Kapoor and Maheshwari introduced two algorithms, one for determining shortest paths and one for creating a visibility graph, which are both based on triangulations as well [29]. With their approach, finding shortest paths in the presence of m polygonal obstacles works in $\mathcal{O}(m^2 \log n + n \log n)$, but only uses a certain subset of the visibility graph to actually determine the shortest path. Their visibility graph creation is done in $\mathcal{O}(E + T)$, with T being the time needed to triangulate the space

between the obstacles, and therefore has the same complexity as Ghosh and Mounts algorithm since a triangulation of the space between n obstacles can be done in $\mathcal{O}(N \log N)$ [30], where N is the number of vertices. Due to its simpler structure, the approach by Kapoor and Maheshwari might lead to a better performance in practice.

Krisp et al. also implemented a visibility graph-based routing algorithm and, more interesting in this case, measured its performance [31]: It took over 9.5 hours to determine a path through a city quarter. One must mention, that the paper was published in 2010 and the speed of computers has increased since then. Also, it is unknown how well optimized their implementation was. Nonetheless, this time gives an indication of the computational complexity of creating a visibility graph.

3.1.2 Enhancing existing graphs

Graph generation techniques can be used to add missing edges to routing graphs or to enhance these in case of a low degree of detail.

Funke et al. achieved this by filling holes in an existing road graph, which was the OpenStreetMap network in their case [32]. To do so, they developed a metric to detect holes in an existing graph based on the following idea: If the shortest path between two vertices is much longer than the Euclidean distance, the network probably has a hole. However, this metric produces false positive results when large obstacles, such as rivers, are between the two vertices. In this case, the Euclidean and shortest path distances significantly differ, which is indeed correct in such cases. To handle these problems caused by large obstacles, they discussed the use of Mitchell's implementation of the continuous Dijkstra paradigm as presented in Section 3.2.1. Due to the complexity and lower performance, they decided to use a dense grid instead, cutting out edges intersecting obstacles and use this remaining grid for navigation. Unfortunately, the use of other graph generation algorithms (for example visibility graph construction) was not discussed.

Andreev et al. also worked on the existing OpenStreetMap graph and added sidewalks to roads for more realistic routing and also tried to handle arbitrary source and destination locations of routing queries [33]. They not only added sidewalks but also dealt with open spaces such as plazas and parks. For each plaza, which usually does not contain designated footways, they create a visibility graph to generate edges within the plaza. Edges not relevant for paths through the plaza are removed afterward. Parks are handled differently since they often contain designated paths. If the source or destination starts within a park, special edges are added based on the walking speed and existing edges for paths within the park. After this preprocessing, Dijkstra is used to determine the complete shortest path.

3.1.3 Filling open spaces

A task, which is more relevant for this thesis, is filling open spaces with edges for navigation purposes. This can be done by using one of several approaches.

Kneidl, Borrmann and Hartmann used visibility graphs to produce a sparse graph suitable for navigation [23]. They chose to construct a visibility graph rather than generalized Voronoi diagrams for more realistic edges.

Liu and Zlatanova used the same strategy for indoor routing through open spaces [34], however, they took the spatial size of pedestrians into account and added a buffer area around obstacles to

prevent pedestrians from intersecting with obstacles. Xu et al. also created a precise indoor navigation algorithm, but they used a certain Delaunay triangulation, which is faster than the visibility graph creation and still creates paths of sufficient quality [35].

Another interesting paper on enhancing existing navigation graphs was presented in 2016 by Anita Graser, where she integrated visibility graph edges into the existing OSM data model [1]. Again, the visibility graph was the preferred method to generate edges. Considered and discussed alternatives were two skeletonization methods and the usage of a simple grid filling the open spaces.

3.1.4 Generate graphs from existing maps

A graph is not always available, which motivated Walter et al. to create a procedure using raster maps [36]. They proposed an algorithm with four steps: First, the raster data is classified in obstacle and non-obstacle pixels. Second, the data is used to create a skeletonization graph from it. Third, this graph is used for routing with the A* algorithm. Fourth, the result is smoothed since the skeletonization might yield edges that pedestrians will not naturally use. Even though this technique works well on large datasets, the reason for choosing skeletonization instead of visibility edges or other generation algorithms was not further discussed.

A similar work was presented by Birgit Elias, merging multiple data sources, like topographical maps or indoor building plans, into one vector dataset for routing [37]. To create a graph from existing maps, she used a skeletonization method based on a Delaunay triangulation. She also considered that important point-like structures, like entrances or other points of interest, must also be connected to the graph. This was solved by creating a perpendicular line to the nearest edge of the graph. The problem of unconnected points of interest is one topic of this thesis and is solved during graph creation as mentioned later in Section 5.3.1.

3.1.5 Crowd sourced graph generation

All graph generation methods presented so far are algorithmic approaches. However, mobile devices can store the user's location and trajectory, which can be used as a data source to construct a navigation graph.

Kasemsuppakorn et al. used GPS traces to create a pedestrian navigation graph [38]. Due to the inherent inaccuracies, outliers and general noise of GPS data, the traces are preprocessed to ensure a certain level of quality (for example, by only allowing points created using at least four satellites). The graph generation process takes one trace at a time and first determines the most significant points on that trace, which build a simplified shape of the trace. New segments are created and possibly merged with existing closely located segments with an almost parallel trajectory.

Zhou et al. introduced another approach using crowd-sourced location data to create a navigation graph [39]. They also filtered out anomalies due to inaccuracies in the GPS data and created a density map from this cleaned location data. The ridge lines of this density map were turned into edges since many people walked along these ridge lines. An evaluation of their approach showed that even small amounts of data are enough to create a sufficient routing network. Unsurprisingly, the accuracy and density of the network increases with the number of input trajectories.

3.2 Routing

In this section, related work on general algorithms for geometric and network-based routing is presented. Section 3.3 contains additional specific work related to pedestrian navigation.

3.2.1 Geometric routing

Shortly after Welzl and others published their work on visibility graphs, which can be used in combination with Dijkstra to find shortest paths, Mitchell presented a purely geometric method to find Euclidean shortest paths around obstacles in the plane in 1993 [40]. This approach is the continuous Dijkstra paradigm and its functioning is described in Section 2.3.2. His approach creates a shortest path map of size $\mathcal{O}(n)$ for one specific source vertex, which allows routing requests to be answered in $\mathcal{O}(\log n)$. Generating this map has a subquadratic time and space complexity of $\mathcal{O}(n^{5/3+\epsilon})$.

Another approach to solve the geometric single-source shortest path problem was presented by Hershberger and Suri one year later, in 1997 [41]. They also use the continuous Dijkstra paradigm but increase performance by introducing a special quadtree-like structure as a sophisticated subdivision of the plane and also introduce approximate wavefronts. The difficult part is the collision handling of non-neighboring wavelets. Maintaining some important properties is crucial to the efficiency when creating the subdivision, most important is the $\mathcal{O}(1)$ amount of neighboring cells around an edge. Their algorithm calculates a shortest path map allowing queries to be answered in only $\mathcal{O}(n \log n)$ time and space, where n is the number of vertices in all obstacles.

In a more recent paper, Shen et al. combined an obstacle-avoiding pathfinding algorithm with compressed path databases [42]. One aspect of their work, particularly important for this thesis, is the filtering of vertices used to create a visibility graph. They only considered the set of vertices, which are part of the convex hull of an obstacle. This does not yield a subquadratic time complexity but reduces the number of generated edges.

A visibility graph can also be generated on a grid, which was done by Oh et al. [43]. They introduced two new algorithms to create sparse navigation graphs on a grid. Next to the convex vertex filtering, which was also used by Shen et al., they introduced the concept of so-called taut regions. For each vertex, these regions are determined and used to prune edges, which will never be part of any shortest path. This additional filtering step further increases performance by reducing the number of output edges without affecting the accuracy of shortest paths. Another aspect relevant to this thesis is their handling of collinear vertices. Instead of naively considering them as k sized clique, which would result in $\mathcal{O}(k^2)$ edges for k collinear vertices, they instead form one line with $k - 1$ edges.

Both filtering methods, convex vertices and taut regions, were implemented independent and without any prior knowledge of the two papers mentioned above. Details on the implementation used in this thesis are given in Chapter 5.

3.2.2 Network-based routing

The topic of network-based routing, meaning the problem of determining shortest paths in a graph, is a well-studied area with widely used algorithms.

One of the most popular algorithms is the A* algorithm introduced by Hart, Nilsson and Raphael in 1968 [16]. A detailed description of A*, including its time and space complexities, can be found in Section 2.2.2.

Because the result of this thesis can be used with any network-based algorithm, all previously described routing algorithms and speedup methods are related to this thesis. The combination of speedup methods is also possible and has been studied by Bauer et al. [44] with promising results on the combination of arc-flags and highway hierarchies, which they called CHASE. A comparison by Bast et al. showed the performance increase against normal speedup methods [13], for example, contraction hierarchies had a 19 times slower query but a six times faster preprocessing time.

3.3 Pedestrian pathfinding

A special case of the general single-source shortest path problem is pedestrian routing, also called pedestrian shortest pathfinding. This problem is especially interesting since pedestrians have the maximum degree of flexibility/freedom compared to other traffic participants, such as bikes or cars. Therefore, geometric routing is a good way to find shortest paths for pedestrians.

Pedestrian pathfinding is not only relevant for real-world users who want to know how to get from one location to another. Such algorithms are also used in (multi) agent simulations to model pedestrian behavior accurately. Scenarios in such simulations are, for example, evacuations or crowd behaviors. The area of pedestrian path planning and simulations is divided into two strategies: network- and field-based pathfinding [45]. Works from both strategies will be covered in the following.

3.3.1 Field-based and dynamic navigation

Besides geometric and network-based routing, potential fields can also be used to find shortest paths within this field.

Teknomo and Millonig created a dynamic algorithm that does not pre-compute the path with one of the approaches mentioned above [25]. This is interesting since the probably most common and naive approach is to pre-compute the route used by an agent within a simulation. The term “dynamic” refers to the fact that real-world pedestrians often only consider the near neighborhood within their path planning. Even if a person knows the whole area, dynamic changes (closed doors, construction work, crowds blocking the way) can still occur at any point in time. Therefore, considering such dynamic changes make agent-based simulations more realistic.

The aforementioned dynamic routing by Teknomo and Millonig has strong similarities to approaches based on potential fields, like the dynamic pathfinding from Hartmann [45]. He proposed a mechanism using vector fields in a cellular automaton model with hexagonal cells to find paths through open spaces. Interestingly, this method can be interpreted as wavefronts, like in the continuous Dijkstra paradigm, propagating through the vector field.

3.3.2 Graph-based path planning

An alternative to field-based approaches are graph-based ones, where a graph is constructed and then used in navigation. The construction can be based on potential fields but also on the aforementioned visibility graphs.

In fact, Gloor, Stucki and Nagel presented a pedestrian navigation model and a comparison of these two strategies regarding their performance [24]. They developed this model for a pedestrian simulation in the Alps, where a hiking network should be enhanced with additional navigation information. Two approaches were presented: a potential field, which is used in navigation when needed, and a visibility graph, which is merged into the existing hiking network. For the performance evaluation, they used an evacuation scenario with the results that both navigation mechanisms are comparably fast.

The work of Kneidl et al. from Section 3.1.3 was used by Kielar et al. as a basis for a generalized routing model for pedestrian simulations [46] taking cognitive models of agents into account. Cognitive models take into account that humans have their own mixed and fuzzy path-finding behavior, which differs from optimal mathematical algorithms like Dijkstra. The basic behaviors of human crowds are herding and learning from others, which makes communication and observation between agents necessary.

4 Design

As mentioned in the introductory motivation, the task of the developed *hybrid routing algorithm* is to find optimal paths through open spaces between arbitrary source and destination locations. Current routing approaches are either incapable of correctly reaching arbitrary locations (in the case of graph-based algorithms like A*) or do not allow the use of existing road edges and speedup techniques (for example in the continuous Dijkstra approach). Additionally, none of the existing approaches uses both, the open spaces between obstacles and an existing road network. Latter may still contain relevant information for pedestrian navigation, such as sidewalk information, surface conditions or access restrictions, and are therefore relevant for routing.

The resulting hybrid routing algorithm does not have these disadvantages and therefore offers a flexible and accurate routing. This chapter covers the implemented algorithm's design to accomplish these tasks. First, the overall context is presented, especially the requirements and constraints of the algorithm. Second, details on the routing strategy and general design decisions are presented. Finally, an overview of the separate components and a description of the deployment of this algorithm is given.

4.1 Requirements and constraints

Enhancing the routing capabilities of the MARS framework is an integral aspect of this work. Even though the implementation is based on MARS, the overall design and the resulting architecture is independent of MARS. Integrating the algorithm into a different context or using it as a standalone application is therefore possible with no or minimal changes in the code.

Apart from the integration into MARS, there are several requirements on the algorithm itself, which are presented in the following. Technical constraints also exist and are discussed at the end of this section.

4.1.1 Requirements

Functional requirements of the resulting software can be formulated without problems since this is not a complex software system but a single algorithm with a distinct task. The hybrid routing algorithm should determine an optimal path between two arbitrary locations by following existing ways but also by traversing open spaces while avoiding obstacles. An optional weight function should assign custom edge weights to find minimum-weight paths. The weight function should enable the algorithm to alternate between road segments and open spaces.

More complex and time-consuming are the quality requirements. Because this is not part of a commercial software development, these requirements are not part of any specification. Nonetheless, they exist and consist of the following aspects.

The most important quality requirement in terms of effort is performance. It affects large parts of the software architecture since the resulting algorithm, despite its complexity, should ideally have a negligible impact on the overall performance of a simulation compared to current graph-based routing approaches. Routing algorithms and engines often consist of a preprocessing and query-answering stage. In order to create fluent and fast simulations, the performance of answering numerous routing requests must be as good as possible. The time needed for preprocessing, however, is of less importance.

The most essential requirement is the correctness of the resulting routes, meaning the algorithm must determine the truly minimum-weight paths according to a given weight function. Another quality requirement is the closeness of the resulting routes to actual pedestrian behavior. Unfortunately, this can hardly be measured without having extensive data of pedestrians and their actual tracks in the real world. Ideally, the trajectory of a calculated route should be identical to a real-world pedestrian track and should at least make sense to an outside observer. The trajectories of the shortest path and a real-world route should be as similar as possible. However, real pedestrian behavior, for example, keeping a minimum distance to obstacles, is not part of the shortest path calculations presented in this thesis, but rather a task of modeling a pedestrian agent.

4.1.2 Constraints

One major constraint is the ability to integrate the implementation of the final algorithm into the MARS framework. This means the programming language C# and the use of the *NetTopologySuite* (NTS) library are predetermined since MARS is based on these as well. However, these technical constraints do not affect the higher-level architecture and its components.

The planned integration into another code base, in this case the one of the MARS framework, affects the management of dependencies. On the one hand, the amount of newly introduced dependencies should be kept to a minimum, which means dependencies on MARS and NTS are to be preferred over other third-party libraries. On the other hand, using only libraries on which MARS depends as well is not always possible due to version mismatches. More specifically, a triangulation method was needed, which is part of a more recent NTS version incompatible with MARS at the point of development. However, this case only appeared once, can easily be circumvented and should disappear with future dependency updates.

An additional constraint is the limitation on vector data as the tape of geodata used for obstacles and roads. Using raster data is theoretically possible, as mentioned in Section 3.1.4 for the work of Walter et al., but requires additional and different processing steps. Therefore, only vector data is supported.

4.2 Combination of routing algorithms

This section describes the core aspect of this thesis: The decision on a strategy to combine graph and geometric routing algorithms. A decision on this “merge” strategy is crucial to the design and architecture of the application.

The following four approach candidates were considered and are discussed in the following. The last approach was considered the most promising and was therefore implemented.

1. Ad hoc generation of edges
2. Concurrent routing
3. Concurrent routing on smaller segments
4. Merge of an existing network with a visibility graph

4.2.1 Ad hoc generation of edges

The idea of an ad hoc generation of edges is the following: Whenever the graph routing algorithm reaches a road junction, it is paused and the continuous Dijkstra algorithm is started. No destination vertex is defined, which means the geometric routing will be stopped after the furthest wavefront reaches a certain distance. The continuous Dijkstra approach creates a shortest path map, so the shortest paths to all reached vertices in the road graph are calculated. All shortest path edges are then added to the road graph and the paused graph-based routing resumes. A real-world example of this approach would be a pedestrian walking down a road, stopping at a junction and consulting a map to decide on which path to continue.

The advantage of this approach is the realistic behavior of pedestrians not planning the entire route in beforehand. In fact, Teknomo and Millonig introduced a routing mechanism for agent-based simulations with the assumption of little to no apriori knowledge of agents about their environment [25]. This approach would, therefore, partially implement their assumptions on an agent's behavior.

One disadvantage is a relatively high complexity since no standard algorithm from software frameworks used in this work supports a pause functionality, so any existing routing algorithm would have to be manually adjusted.

Also, this approach will likely cause performance issues. When using Dijkstra as a graph-based routing algorithm, $\mathcal{O}(|V|)$ many vertices are visited, causing $\mathcal{O}(|V|)$ many geometric routing queries for the continuous Dijkstra algorithm. Even when using the algorithm from Hershberger and Suri [41] with a time complexity of $\mathcal{O}(n \log n)$, it can be assumed that $n > |V|$ resulting in a complexity of $\omega(|V|^2)$. Caching the shortest path map using a CPD would increase query performance but it would also increase implementation complexity and processing time during preprocessing. Instead, precomputing all shortest path edges and directly merging them into the road graph would eliminate complexities, enhance performance and is used in the fourth approach.

The ad hoc idea of this approach yields no significant advantage compared to a preprocessing-based algorithm. Therefore, this approach was not further pursued.

4.2.2 Concurrent routing

Combining the results of two concurrent routing procedures, one graph-based and one geometric, is another approach. Splitting the resulting paths at intersection points and only choosing minimum-weight segments yields the overall shortest path, which potentially alternates between graph-based segments and segments from geometric routing.

The most prominent advantage is the simplicity of answering routing requests since known routing algorithms could be used. Therefore, it would be relatively easy to implement and speedup techniques could be used for the graph-based routing.

However, there are two significant disadvantages. First, it is uncertain whether the two paths are actually intersecting at any point. Second, even if they do intersecting, too few intersections on a long

route result in suboptimal paths because routing parts of a long segment with the respective other routing technique might further reduce the overall weight. Only if two routes intersect frequently enough, the selection of segments based on the weight function could result in good routes.

4.2.3 Concurrent routing on smaller segments

This approach is very similar to the one above, but it tries to fix the uncertainty of intersections between the two resulting paths. Multiple ways are conceivable to ensure enough intersections exist or to otherwise guarantee that segments are small enough to be merged.

One way is to stop the two routing algorithms after a certain distance at the next available vertex. After stopping for the first time, there are two such end vertices, one where the graph-based and one where the geometric routing stopped. From each vertex, two new routing queries start and stop again after a certain distance resulting in four new end vertices. This continues until one query reaches the destination. On the one hand, this yields arbitrarily small segments for later merging, but on the other hand, this results in $\mathcal{O}(2^n)$ many routing queries with n segments in the overall shortest path. Even though each query is short, this approach would not scale very well even with the help of heuristics or other helping mechanisms.

A different approach to obtain smaller segments would be the following: First, the shortest path on the road graph is determined and split into segments of specific length connecting the vertices $v_0, v_1, ..., v_n$ with v_n being the destination. Geometric shortest paths are determined between all points, resulting in $O(n^2)$ many routing queries. The graph-based segments and the geometric routing results are then merged, forming a new intermediate graph. Finally, one last routing query on this intermediate graph is performed to get the final optimal route. Other combinatorial approaches could be used instead of an intermediate graph since the shortest path problem is a combinatorial problem and, thus, can be solved using linear programming [47].

There are likely to be more possible ways to ensure a sufficient amount of segments or intersections. However, this overall approach was not further pursued because both approaches have worse time complexities than any popular routing algorithm, including pure geometric routing. The first one would cause $\mathcal{O}(2^n)$ and the second one $\mathcal{O}(n^2)$ many routing requests, of which each request has a certain complexity itself. Even though the second idea might work well for appropriate segment lengths, the complexity and definite time overhead make it an unfavorable choice.

4.2.4 Merge of an existing network with a visibility graph

Previous approaches tried first to calculate shortest paths and then merge their results. This last approach, which was ultimately implemented, first merges a generated visibility graph with an existing road graph. The actual merge operation is rather simple: Whenever a road edge and a visibility edge intersect, split the edges, create a new vertex at the intersection point and connect the split edges accordingly.

Even though this approach is simple, the main disadvantage is the graph size. A visibility graph has $\mathcal{O}(|V|^2)$ many edges, which negatively affects the routing performance.

Another disadvantage is the time complexity of the merge operation. All edges must be considered and, depending on the number of intersections, edges might be processed multiple times. This leads

to at most $|E_R| \cdot |E_V|$ many merge operations for the routing graph edges E_R and visibility edges E_V, which is in $\mathcal{O}(|E|^2)$ given all edges E.

Fortunately, road networks are very sparse and often have a vertex degree between three and six [48] [49], resulting in a probably much better runtime behavior in practice. An analysis of OpenStreetMap data for the road network of Germany's densely populated states North Rhine-Westphalia (NRW) and Hamburg (HH) showed node degrees of 3.1 (NRW) and 3.11 (HH). A vertex with a degree of 10 or higher did not exist in either of these states.

Despite the disadvantages of this approach, I considered the advantages to be more important. As already mentioned, this strategy is not only simple, it also allows the use of speedup methods for routing queries (as presented in Section 2.2.3) and has potential for a wide range of performance optimizations as outlined in Section 7.1. Moreover, the resulting path is optimal based on the given weight function due to the usage of visibility edges, which are the shortest possible connections between vertices. Also, no complex postprocessing and only one routing algorithm is needed.

Taken all aspects into account, this strategy seemed to be the simplest and most promising with the least significant disadvantages. It was therefore chosen to be implemented.

4.3 Design decisions

Several design decisions were made after the decision of the strategy to combine graph-based and geometric routing fell. Some decisions did not influence the overall architecture and were just made to enhance performance or simplify the implementation, which are therefore described in Chapter 5. This section solely covers fundamental decisions relevant to the overall architecture.

In this work, I implemented the visibility graph without using algorithms presented in the corresponding related work section. None of the existing approaches would have been easy to implement without problems. Details on these problems and the exact reasons that led to the decision of a custom implementation are also presented in this section.

4.3.1 Requirements and considerations for the visibility graph creation

One essential requirement for the visibility graph creation is the ability to work with arbitrary obstacles, which primarily includes the fundamental point, linestring and polygon geometries. However, real-world datasets also contain multi-geometries such as multipolygons. Furthermore, assumptions about the collinearity, position of obstacles and intersections between them cannot be made for real-world datasets.

An important special case arises for linestring obstacles with more than two vertices. Simply determining visibility edges to and from both sides of the linestring would result in shortest paths leading right through it because vertices of linestrings are visible from both sides. The same situation would arise with two arbitrary obstacles, including polygonal ones, touching each other in a single vertex. Both cases must be considered to ensure that shortest paths lead *around* the obstacle.

A core aspect of the routing algorithm is the reachability of arbitrary locations not represented by vertices within the graph. Therefore, adding visibility edges after the graph creation must be possible and should ideally be independent of the algorithm chosen to generate the graph and edges.

As the generation of the routing graph happens in a preprocessing step, the time and space needed

for this task was not a critical aspect of the whole algorithm. Further optimizations are possible and discussed in Section 7.1. The routing behavior is significant to agent-based simulations as it directly affects the time required for a simulation. Creating the graph in a preprocessing step decouples its processing time from the simulation, especially when persisting the graph for later uses.

All these requirements and considerations, including problems and efforts outlined below, led to the decision of implementing a simple graph generation approach with some easy but effective optimizations.

4.3.2 Suitability of existing approaches

The largest and most influential design decision was the decision against an approach from the literature as presented in Section 3.1.1. Several approaches from the literature have either restrictions on the geometric structures or assume that a certain preprocessing of the data has already been performed.

The approach by Welzl [26], for example, explicitly assumes that there are no collinear vertices, i.e. three or more vertices in a straight line. It may seem unlikely to occur in real-world datasets, but it is a common practice in OpenStreetMap to align touching buildings and other straight obstacles like walls.

The optimized approach by Overmars and Welzl [27] only works on non-intersecting line segments. However, real-world data is too diverse to fulfill this criterion as polygons, longer linestrings and other more complex data structures exist. Splitting all geometries into such non-intersecting line segments would increase complexity and would also introduce new problems to solve. For example, splitting a polygon into line segments would require metadata on these segments to reconstruct the former polygon membership in order to avoid visibility edges within the former polygon, which would be created for an actual chain of linestrings. Handling all those special cases, preventing unwanted edges and still using the approach by Overmars and Welzl would be possible but significantly increases the complexity of their approach.

Another approach with the limitation of no collinear vertices is the plane-sweep algorithm by Ghosh and Mount [28]. In addition to the limitation on collinear vertices, they assume x-coordinates to be unique, which would need to be handled as well. Even though coordinates are usually floating point numbers and therefore have a great range of possible values, rounding and size limitations reduce this range and might result in a high number of non-unique x-coordinate values. Counting the number of non-unique x-coordinates in OpenStreetMap illustrates this problem: Of all coordinates in the OSM dataset of the German state of Hamburg, 22.8% are non-unique.

Kapoor and Maheshwari presented a visibility graph creation as well [29] but assumed a triangulation of the open space between the obstacles. Such a triangulation can be done relatively fast, algorithms with time complexities of $\mathcal{O}(n \log n)$ and better are known [50, pp. 58–60]. However, libraries such as MARS or the NetTopologySuite do not support triangulation for open spaces, only for closed geometries. Implementing an algorithm for this task would be needed in addition to the visibility graph creation algorithm.

All approaches from the literature would probably work but preprocessing, restrictions on the input data or adjustments to the approaches would be necessary. Since this work focuses more on the overall concept of a hybrid routing algorithm, the choice of the graph generation algorithm was

less important to me. Therefore, the implementation in this work, as shown in Chapter 5 in detail, is simpler and independent of the aforementioned approaches.

4.3.3 Early implementation based on the continuous Dijkstra paradigm

Before the decision for a routing strategy fell, a prototype implementation was created based on the continuous Dijkstra paradigm with propagating wavefronts. However, next to the overall strategy decision on the visibility graph approach, two additional reasons existed for a visibility graph-based algorithm.

One reason was the implementation of performance enhancement for the continuous Dijkstra algorithm. Early stages used a very naive and simple implementation without optimizations mentioned in recent literature on this topic. Wavefronts did not move continuously, but instead, they "snapped" to the next event, i.e. to the next collision with a visible vertex where new wavelets can spawn. In this early implementation, no extensive efforts went into optimizations, e.g. by implementing the approach presented by Hershberger and Suri [41]. Instead, a simple preprocessing was introduced, which led to a significant performance improvement. Because a wavefront only reaches vertices visible to the origin of the wavefront (which is vertes itself), the visibility between all vertices was determined and used for the events mentioned above. Such predetermined visibilities are the core idea of a visibility graph, thus, moving to an approach using an actual visibility graph was only a small step.

A second reason against this early continuous Dijkstra implementation was the difficulties and the disadvantages of combining the network-based routing with the continuous Dijkstra algorithm, as described in Section 4.2.

Therefore, this first continuous Dijkstra approach was not further pursued but converted to a generator for a routable visibility graph with components described in the following section.

4.4 Components

The hybrid routing algorithm consists of two main components presenting public interfaces. Generating the underlying routing graph, the so-called *hybrid visibility graph*, is the first step and resides in its own component, the `HybridVisibilityGraphGenerator` class. The hybrid visibility graph itself is contained in the `HybridVisibilityGraph` class, which primarily contains the graph structure and is used to answer routing queries. A third but internal component, not intended to be used from outside the algorithm, is the `VisibilityGraphGenerator`, which creates the visibility edges for the routable hybrid visibility graph. Since this work resides in a rather technical and algorithmically oriented context, no other components are involved, such as a user interface, application server or database.

In an agent-based simulation, determining routes is part of the agent's path-planning routine. The hybrid visibility graph has the necessary interfaces to answer routing queries, meaning an agent needs access to an instance of this graph. Figure 4 illustrates this with an exemplary agent holding an instance of the hybrid visibility graph, which can be created once and shared among multiple agents.

The `HybridVisibilityGraphGenerator` (abbreviated as "H.V.GraphGenerator" in Figure 5) presents a public `Generate()` method for the task of the graph generation and, in this exemplary

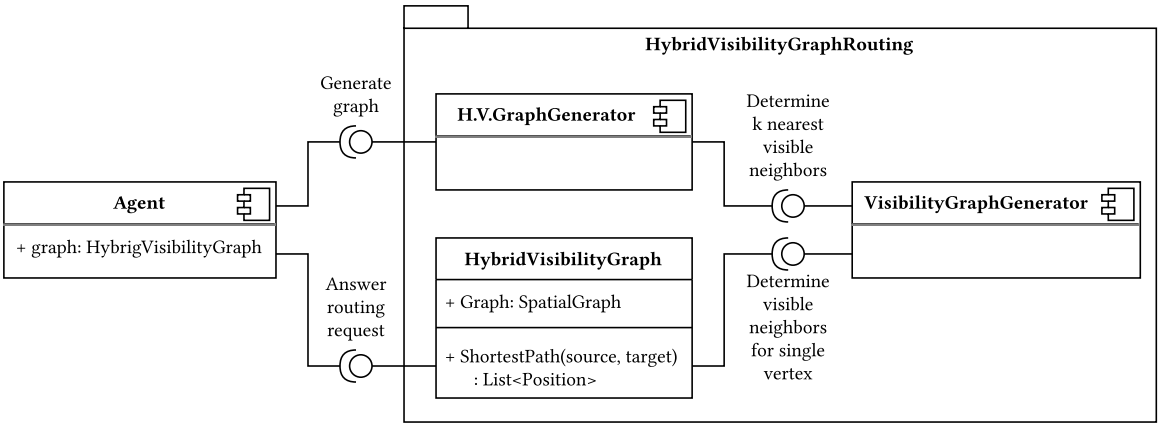


Figure 4: The components and usages of interfaces of the resulting implementation.

setup, is used by the agent. It returns an instance of the `HybridVisibilityGraph` class offering public methods to answer shortest path queries. Figure 6 illustrates the internal tasks of a routing request using the `OptimalPath(source, destination, weightFunction)` query method. The result is a list of coordinates representing the shortest path between two input coordinates.

To fulfill the overall goal of this work of finding optimal paths between arbitrary locations, it must be ensured that these locations are reachable via the hybrid visibility graph. Without further consideration of the source and destination locations, there is no guarantee that these locations are represented by any vertex in the graph. Therefore, the hybrid visibility graph has to temporarily connect these locations to the graph if they are not present already. This is done by determining all visible neighbors of the source and destination locations and merging the resulting edges into the underlying graph. Because this is the same step performed during graph creation, the `VisibilityGraphGenerator` is used during routing as well to determine visibility neighbors of a single vertex. Section 5.4 covers the process of answering routing queries in more detail, including a clean-up step that is performed after the shortest path has been found.

The shortest path algorithm used is A*, which uses the underlying spatial graph of the hybrid visibility graph, which temporarily contains the additional edges to and from the source and destination vertices. A custom weight function allows exact control on how strong to prefer road edges. The effects of different weightings are evaluated in Section 6.3.3.

4.5 Deployment

The goal of this thesis is to enhance route quality for agent-based simulations. Therefore, the algorithm was implemented using the MARS framework. In order to use this work to improve MARS, the code is published on GitHub[1]. The C#-Project “HybridVisibilityGraphRouting” contains all necessary classes and interfaces, only the `Triangulation` project will be needed as well due to the aforementioned version mismatch regarding the triangulation of polygons.

[1] `https://github.com/hauke96/master-thesis/`

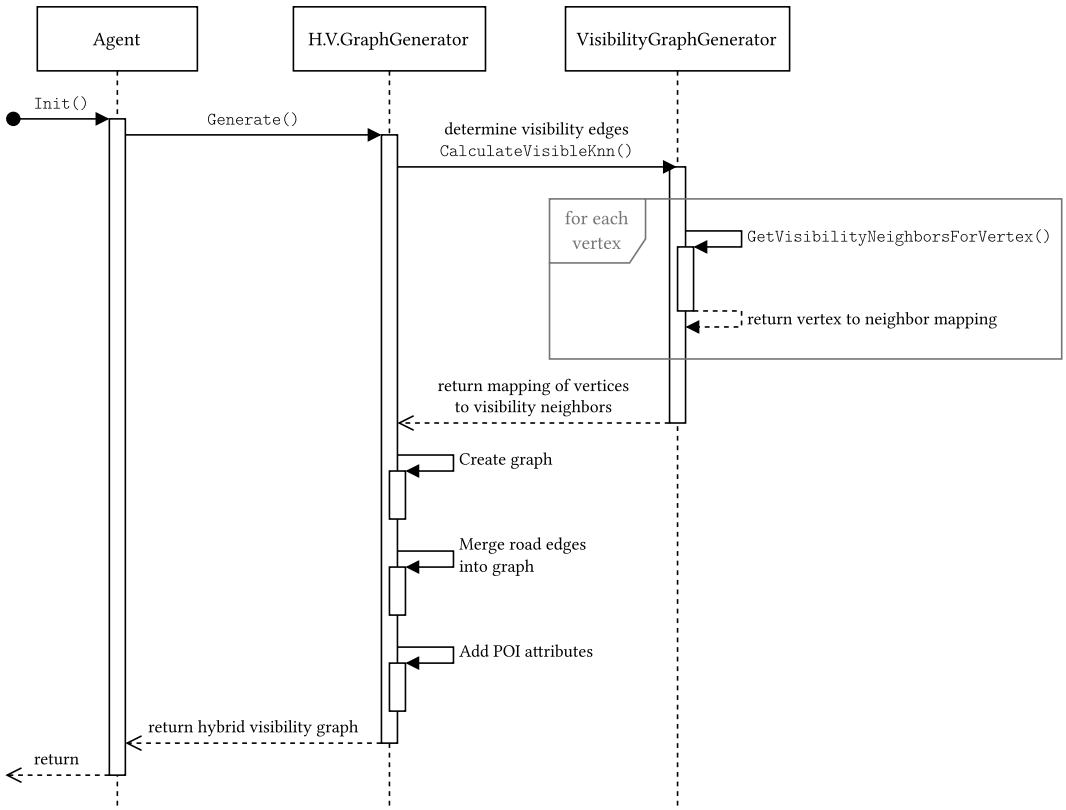


Figure 5: The generation process to obtain the hybrid visibility graph.

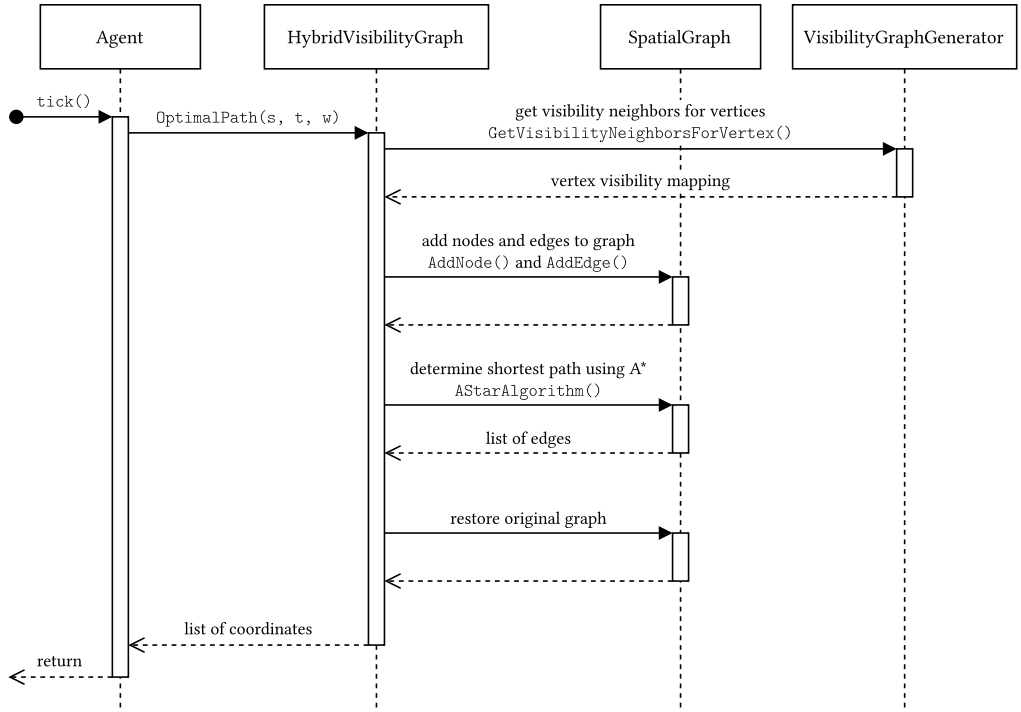


Figure 6: The most important steps performed during a routing request using the hybrid visibility graph. The variables `s`, `t` and `w` are the source vertex, the target vertex and the weight function.

5 Implementation

This section covers details on the implementation of the previously described design. First, an overview of the algorithm is given, including a short description of used frameworks and technologies. Second, two new and important data structures are described in detail. And third, each algorithmic step of the graph generation and query answering is described in the main sections of this chapter.

5.1 Algorithm overview

This section presents further details on the code dependencies and gives a broad overview of the algorithm.

5.1.1 Frameworks and technology

As mentioned in Section 4.1.2 regarding the constraints, the used programming language is C# due to the dependency to the MARS framework. The MARS framework is based on the widely used C# library *NetTopologySuite* (NTS), which contains numerous data structures and algorithms to work with geospatial data. No additional third-party libraries were used.

Primarily, the NTS was used for basic data structures like coordinates, geometries and features but also for calculations, e.g. the calculation of distances. Serialization of data is also done with the help of the NTS, however, this was primarily used for testing and development purposes. The MARS framework is also used, primarily for the `Position` class and high-level structures such as the `SpatialGraph` and `QuadTree` classes.

Not all data structures and algorithms already existed that were required to create the hybrid routing algorithm. Most notably, a fast line intersection check, a bin-based interval index and a fast vertex filtering method were implemented, which are all presented with more details in this chapter.

5.1.2 Algorithm steps

This section gives details on the main parts of the algorithm, namely the generation of the hybrid visibility graph and the answering of routing requests.

Graph generation

The `HybridVisibilityGraphGenerator` class provides the central method to create an instance of the `HybridVisibilityGraph` class from a given collection of features. This method consists of the following top-level steps, which are all described in Section 5.3 in more detail:

1. Filter the given features for obstacles
2. Determine the visibility neighbors

3. Use this visibility relation to create a routable visibility graph
4. Merge the existing road network into this graph

A fifth but simple and not necessarily relevant step adds attributes of all POIs to the corresponding nodes of the resulting hybrid visibility graph, which can then be used during routing.

Answering routing queries

To determine optimal paths between arbitrary locations, the resulting graph needs to be extended with edges from and to the given source and destination locations, because vertices on the graph might not represent these locations. The generated edges themselves are visibility edges and can therefore be created and merged just like the visibility edges during the graph generation before. Further details are given in Section 5.4 describing the following steps:

1. Determining visibility edges for the source and destination coordinates
2. Merging these edges into the graph such that they can be removed afterward
3. Routing along the resulting graph with a graph-based routing algorithm such as A*
4. Restoring the original graph by removing all vertices and edges added before

The last step ensures that subsequent routing queries are answered based on the original graph and not on an altered version of it. It also prevents an uncontrolled growth of the graph.

5.2 Data structures

Before the separate steps of the graph generation are described in detail, the newly created data structures are presented.

5.2.1 Shadow areas

Shadow areas are a method I designed and implemented to quickly determine and filter out vertices that are not visible to each other. A well-suited analogy for shadow areas is a light bulb and obstacles casting shadows. Anything within a shadow is not visible from the light bulb. Interpreting a vertex v as a light bulb illuminating its surroundings, an obstacle o casts a shadow outwards. The fundamental property of this idea is that every vertex within this shadow is definitely not visible from v. Each shadow is determined by three values: Two values for the angular range (also referred to as *angle area*) and a third value for the minimum distance from which other vertices within the angular range are definitely not visible.

The angular range is determined by two vertices, the so-called *bounding vertices*, which are marked in red in Figure 7, one with the smallest and one with the largest angle from the processed vertex v. Thus, the shadow area covers the largest possible angular range for the obstacle it belongs to.

The furthest of the two bounding vertices determines the minimum distance of the shadow area. Using the distance of the nearest bounding vertex would result in a shadow area leading to false positive results, meaning a vertex might be in the shadow even though it is visible from v. Taking the larger distance into account results in false negative results (e.g. v' in Figure 7), meaning the vertex is not in the shadow area of an obstacle even though it is hidden by it. These false negative

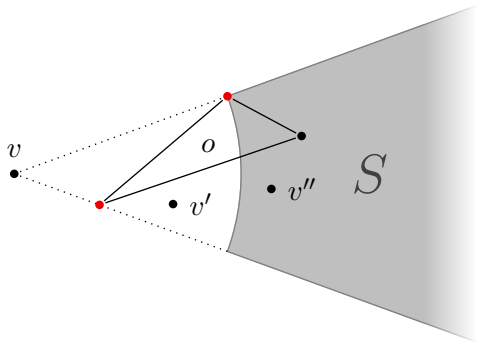


Figure 7: Shadow area S cast by obstacle o seen from vertex v. The two red vertices of obstacle o are the bounding vertices determining the angular range and distance of S. The vertex v'' is not visible from v since it lies inside the shadow area. Note that v' is not visible either, even though it is not inside the shadow area.

results can be handled by a full visibility check. More importantly, when using the distance to the furthest bounding vertex, *every* positive result, meaning the checked vertex is within a shadow area, is a true positive result and needs no further checking.

When processing a vertex, creating the corresponding shadow area of an obstacle only required one iteration over its vertices. Each shadow is an interval (between 0° and 360°) of certain distance, which turns the visibility problem into an interval intersection problem. Thus, checking whether a specific coordinate is within any given shadow area is a simple operation using just a few numeric comparisons.

Keeping track of these shadow areas for each vertex significantly improves performance, especially with the use of the `BinIndex` data structure described below. Enabling the shadow area optimization yielded a speedup factor of 10.62 for a 0.5 km^2 real-world dataset as described in Section 6.1.2).

5.2.2 BinIndex for intervals

The `BinIndex` class implements a linear bin-based index structure to store and access intervals. It consists of an array of n many bins, each bin covering a certain interval. Items are added to all bins intersecting with the range of the item and linked lists are used to store multiple items per bin.

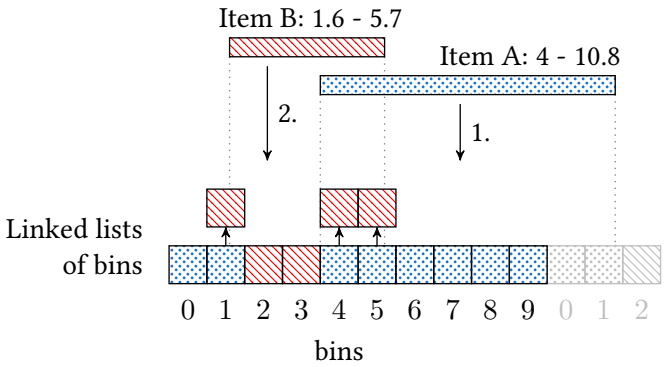


Figure 8: Bin index filled with two items A (added first) and B (added last). Since item A was added first, B's entries, which fall into a bin where an entry for item A already exists, are moved to the next item in the linked list of the bin. Each bin covers a range, meaning non-integer values are rounded down (for from-values) or up (for to-values).

Point queries can be answered in $\mathcal{O}(1)$ (array access and returning the list) and range queries in $\mathcal{O}(n)$ for n items in the index. However, the range query functionality was not implemented as it is not needed for this work. Figure 8 illustrates this data structure with two items added to the index.

The NetTopologySuite offers several indices specifically made for intervals, namely the `Bintree`, `SIRtree` and `SortedPackedIntervalRTree`, of which only the first one is dynamic and thus allows insertions and deletions after the first query was made. This dynamic behavior is crucial for the use of shadow areas because the creation and handling happens in an online fashion by processing each vertex and its adjacent obstacles.

Tree-based structures offer already good logarithmic query times, but the simple and list-based `BinIndex` with constant-time complexity is significantly faster, especially for larger datasets with hundreds of thousands of vertices. Table 1 compares the `BinIndex` with the `Bintree`, showing the significant performance differences between these two data structures. The list-based `BinIndex` is multiple orders of magnitude faster than the `Bintree`.

	`BinIndex`		`Bintree`	
Items	**Insert**	**Query**	**Insert**	**Query**
10,000	0.05 s	0.32 ms	3.83 ms	1.75 s
100,000	0.91 s	2.65 ms	22.71 ms	164 s
200,000	1.95 s	4.27 ms	58.86 ms	635 s
1,000,000	9.86 s	21.27 ms	89.11 ms	12,549 s

Table 1: Comparison of `BinIndex` and `Bintree` on random intervals with one query per interval.

5.3 Routing graph creation

This section presents implementation details on the steps given in Section 5.1.2 for graph generation, which includes generating a visibility graph and merging it with a road network.

5.3.1 Step 1: Obstacle filtering and feature preprocessing

The first step is to filter all input features by their attributes to get all relevant obstacle features, regardless of their geometry. For performance reasons (during triangulation as described below), multi-geometries and polygons are split into their separate geometries. The NTS implementation of a polygon can have outer and inner rings, which are unwrapped into separate geometries. Unwrapping `MultiLineString` geometries is also unproblematic since the result is just a collection of ordinary linestrings. Features made of `Point` geometries are also considered so that visibility edges can reach points of interest (*POIs*).

The second task within this first step is the triangulation of all polygonal shapes, which happens during the unwrapping of multi-geometries. The main reason is a better performance for intersection checks, which is an important operation when checking if an obstacle is between two vertices. Details on the intersection check implementation are given below.

Also worth mentioning is the determination of each obstacles' convex hull, which is fortunately not a complex task for polygons. As described below, only vertices on the convex hull of obstacles will be considered in the further processing, which is a simple performance enhancement.

5.3.2 Step 2: Determining k nearest visible neighbors

This step performs the main task of the visibility graph creation, namely for each vertex the determination of all visible k many nearest neighbors (kNN). Before the actual algorithm is presented, some concepts and optimizations are described to understand the implemented algorithm. Then an overview of the performed steps is given, followed by details on each step.

Terminology

Some terms need to be defined before describing the details of determining the k nearest neighbors:

obstacle An *obstacle* is a geometry that should be avoided when determining shortest paths. It can be any type of geometry, not only a polygon.

visibility neighbor A *visibility neighbor* (sometimes also *visible neighbor*) of a vertex v is another vertex u which is visible to v and vice versa. In other words, an imaginary linestring from v to u does not intersect with any obstacle.

obstacle neighbor Two vertices v and u are *obstacle neighbors* if they are part of the same obstacle and directly next to each other in its coordinate list.

valid angle area A *valid angle area* describes a specially determined angular range around a vertex in which potential visibility neighbors are checked.

Parameter k and consequences on the shortest paths

As a performance enhancement providing an upper bound on the number of visibility neighbors, only k many neighbors per vertex are determined. The parameter k is not a single numeric parameter but rather a tuple of two separate values: The number of bins is k_b and the number of maximum neighbors per bin is k_n, which means a maximum of $k_b \cdot k_n$ many neighbors are determined and considered.

Each bin covers a certain angle area of the currently processed vertex, for example, for a bin count of 36, each bin would cover a 10° area. In fact, the chosen default values of 36 bins with ten neighbors per bin worked well in all and usages of the implementation. When a bin is full but a new vertex should be inserted, an existing vertex might be removed from the bin. The criterion on which vertex to remove is its distance such that only the k_n nearest vertices of a bin remain. Since shortest paths are wanted, storing the k_n furthest visibility neighbors might yield too long paths, unnecessary detours and maybe even very close but unconnected vertices.

Without this subdivision into bins and by only considering a static maximum number of neighbors, i.e. with k being a single numeric value, it is not endured that visibility neighbors are evenly distributed. Considering some complex buildings next to a large, open park illustrates this effect. The k nearest neighbors of a vertex v might all be part of these buildings and the other distant side of the park is not within the set of visibility neighbors, even though the other side of the park is unquestionably visible. Subdividing the neighbors into bins ensures that there will definitely be some connections to the other side of the park.

Generally restricting the number of neighbors reduces the number of created edges and leads to less accurate routing results, i.e. to non-shortest paths. However, the negative impact due to

this restriction is not considered significant for sufficient values of k_b and k_n. From a theoretical perspective, using the above-mentioned default values of ten edges per 10° range, an omitted neighbor u of a vertex v was located in a similar direction as all other neighbors in this bin did. The additional distance of a detour from v over some other vertices to u, due to the missing direct connection, is most likely relatively small. From a practical perspective, described in more detail in Section 6.1.2, no large detours were observed in any used dataset. Using a 0.5 km^2 large real-world dataset, enabling this optimization with the described default values reduced the number of edges by 5.4% and increased performance by 9.3%.

Convex hull filtering

In order to be connected to other vertices, a vertex must be part of a non-polygonal obstacle or the convex hull of a polygonal obstacle. A polygonal vertex (a vertex on a polygonal obstacle), which is not part of any convex hull, is not connected to other vertices. This filtering strategy is a simple performance enhancement with some implications on the route quality, as discussed below.

The reason for this filtering is quite simple: Polygonal vertices not being on any convex hull will never be part of any shortest path. This is the case since the convex hull itself is by definition the smallest set of points that creates an area, which in turn contains all other points of the geometry [50, p. 2]. Interpreting the convex hull as path, the triangle inequality implies that it is indeed the shortest path around the geometry. Therefore, when determining shortest paths through open spaces, the part of a path bending around an obstacle is a subset of its convex hull.

Of course, a routing query might start within a concave part of an obstacle or somewhere completely else. This case, however, is independent of the graph generation and belongs to answering routing queries, which is covered later in Section 5.4.

Also, the weighting function for the routing queries might not determine shortest, but for otherwise weighted routes. In such case, the above filtering might result in inaccurate and suboptimal paths due to missing edges. Since this is solely a performance enhancement and therefore optional, deactivating this filtering increases preprocessing time but solves this inaccuracy problem.

Valid angle areas

As mentioned above, a subsection of a shortest path bending around an obstacle always follows a part of the obstacle's convex hull. Eventually, the path switches from following the convex hull to following a visibility edge, which leads to another obstacle, meaning another convex hull part.

This means, a shortest path consists of many convex hull parts that are either connected by straight lines or touch each other where the underlying obstacles touch. A connecting line, i.e. a line segment connecting two convex hull parts, is the shortest possible connection between the two convex hull parts. In other words, connecting lines cannot be relaxed to an alternative edge, which is shorter.

Assuming a shortest path contains a convex hull edge $e_h = (u, v)$ followed by a connecting line edge $e_c = (v, w)$, then there is no relaxation possible to an edge $e_r = (u, w)$. According to the triangle inequality, the edge e_r would be shorter than the two others, which means $d(e_r) < d(e_h) + d(e_c)$ would be true. But since e_h and e_c are on the shortest path, no such e_r can exist because then the shortest path would not have been shortest. Figure 9 illustrates this scenario with the convex hull segment (v_0, v_1), the connecting line (v_1, v_2) and the impossible relaxation (v_0, v_2).

In fact, edges, which can be relaxed as described, will *never* be part of *any* shortest path. If such an edge would be part of a shortest path, then the path is not shortest and would rather contain the relaxed edge. Determining these relaxable and, therefore, irrelevant edges is the key motivation of angular ranges called *valid angle areas*. Only visible neighbors of a vertex v within its valid angle areas are possible subsequent vertices on shortest paths via v. In Figure 9, the segment (v_1, v_3) is not part of any shortest path from s because it can be relaxed to v_0, v_3.

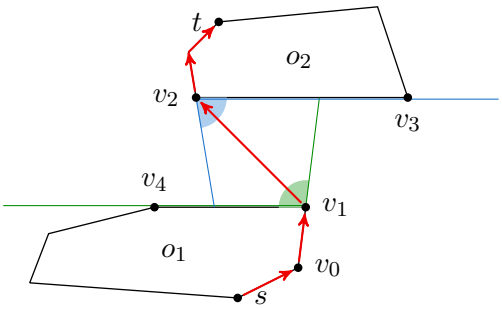


Figure 9: The path from s via v_1 and v_2 to t bends along the convex hulls of o_1 and o_2. The segment from v_1 to v_2 is a connecting line between the hull parts. The angle area at v_1 (green) indicates one of its valid angle areas. A visibility neighbor within this angle area can appear as a successor to v_1 in a shortest path. Vertex v_3, however, is not part of any of $v_1's$ valid angle areas and will never be a successor of v_1 on any shortest path.

These valid angle areas of a vertex v are determined by its obstacle neighbors. For each pair of adjacent obstacle neighbors being at least $180°$ apart from each other, two valid angle areas a_v^1 and a_v^2 exist for vertex v and its neighbors u and w. The function $b : V \times V \to \mathbb{R}$ is called *bearing*, meaning the clockwise increasing angle from a vertex v to v' starting at $0°$ if v' is directly above v.

$$a_v^1 = [b(v,u), b(v,w) - 180°]$$
$$a_v^2 = [b(v,u) - 180°, b(v,w)]$$

For example, for v_1 in Figure 9 with the obstacle neighbors v_0 and v_4 the angular range $[b(v_1, v_4), b(v_1, v_0) - 180°] = [270°, 187.125° - 180°] = [270°, 7.125°]$ is one valid angle area. Subtracting $180°$ of both bearings yields the second one of $[90°, 187.125°]$.

If one or no obstacle neighbor exists, only one valid angle area covering $360°$ will be created. No valid angle areas exist for obstacle neighbors being less than $180°$ apart. A maximum of four valid angle areas exist, each of $0°$, for the central vertex of three collinear vertices, which has two obstacle neighbors being exactly $180°$ apart.

Analogous to the convex hull filtering, this is solely a performance enhancement and might lead to suboptimal results depending on weighting. Removing this optimization solves this problem.

Intersection checks

There are two types of intersection checks needed: One for line segment intersections and one to check whether or not a point lies within a triangle. Both intersection checks are performed using a custom implementation instead of methods from the NetTopologySuite.

Using custom implementations has solely performance reasons: Performing ten million collision

checks between random line segments took 195 ms on average with the custom implementation and 746 ms with the fastest NTS method `RobustLineIntersector.ComputeIntersection()`. Analogously, performing one million random triangle intersection checks required 138 ms on average using the custom implementation and 174 ms using the fastest NTS method `Triangle.Intersects()`.

The line segment intersection check uses cross products as presented in *Introduction to algorithms* by Cormen et al. [12, p. 1018]. Due to the performance of this approach, it is used to check whether a line segment intersects with a whole obstacle by performing this intersection check for each segment of the obstacle.

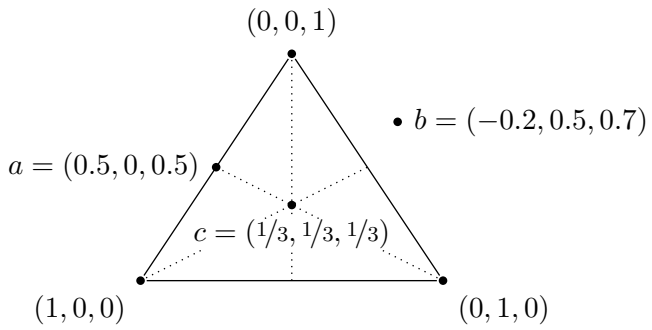


Figure 10: A triangle with barycentric coordinates and dotted coordinate axes. Noteworthy is coordinate b being outside the triangle, which can be directly seen from the negative value -0.2.

Triangle intersection checks are used for closed obstacles, being triangulated during preprocessing. Coordinates in a barycentric coordinate system have the form $(\lambda_1, \lambda_2, \lambda_3)$ of which for each corner of the triangle exactly one value of the coordinate is set to one, and all other ones are zero. Due to the properties of barycentric coordinates, a point p is inside the triangle when the condition $0 < \lambda_i < 1$ holds for all three values as illustrated in Figure 10 with some exemplary points.

It may sound complex to create a whole coordinate system for a single collision check, but this method only uses a few multiplications, additions and boolean operations and is therefore very fast.

Visibility checks using shadow area

Shadow areas are stored in the BinIndex and are always determined for a single vertex v. Checking the visibility to a different vertex u is quite simple. Query all shadow areas for u's bearing from v and check for each shadow area s whether u is further away than the minimum distance of s and if u is exactly within the angular range of s. If both conditions are satisfied for any shadow area s, then $u \in s$ and is therefore not visible from v. One visibility check has a runtime complexity of $\mathcal{O}(n)$ for n shadow areas due to the $\mathcal{O}(1)$ query time of the BinIndex.

Visibility graph generation overview

The following steps, which are described in detail in the following sections, are performed to determine all visibility neighbors of a vertex:

2.1. Determine the obstacle neighbors for each vertex.
2.2. Determine the visibility neighbors of every convex hull vertex.
2.3. Sort the visibility neighbors into bins based on the obstacle neighbors.

Step 2.1: Determining obstacle neighbors

Obstacle neighbors are needed to correctly handle the closed surface of obstacles, especially when two obstacles touch each other. This handling of neighbors is then used to form the valid angle areas used for vertex filtering as described in Section 5.3.2.

Determining the obstacle neighbors is relatively simple and straightforward. Each vertex v of an obstacle o is processed by looking at the previous and next vertices v_p and v_n on the same obstacle, which are both potential obstacle neighbors. If the line segment $(v, v_p) \in o$ (the same applies to v_n) does not intersect with any other obstacle and no other obstacle contains this edge as well (i.e. if no other obstacle touches o at this edge), it is considered an obstacle neighbor. Latter case can be seen in Figure 12 at the line segment (v_1, v_3), which is part of the two touching obstacles.

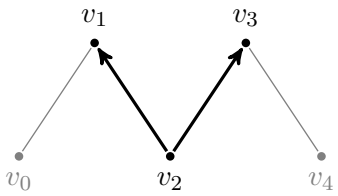


Figure 11: The black arrows point to the obstacle neighbors v_1 and v_3 of v_2.

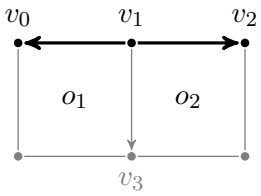


Figure 12: The black arrows point to the obstacle neighbors v_0 and v_2 of v_1. Vertex v_3 is also a neighbor, but hidden because o_1 and o_2 touch each other at the edge (v_1, v_3).

As shown in the next step, obstacle neighbors are crucial to a correct edge generation. The `AddObstacleNeighborsForObstacles` method in the `VisibilityGraphGenerator` class implements this important procedure.

Step 2.2: Determining visibility neighbors

The `VisibilityGraphGenerator` class also contains the necessary methods to determine the visibility neighbors for every vertex. Essential to this process is the `GetVisibilityNeighborsForVertex` method determining the visibility neighbors for a single vertex. This method uses the concept of shadow areas stored in a BinIndex structure with a bin count of 360, each bin covering a 1° area. For any given vertex v, the following steps are performed for every potential visibility neighbor u:

2.2.1. Continue with the next potential visibility neighbor if any of this is true: u is in a shadow area, u is outside a valid angle area or u is not part of any convex hull.

2.2.2. If none of the above condition applies to u, query all obstacles in the extent spanned by v and u. For each obstacle o, perform the following steps:

   a) Create the shadow area S for o.
   b) If $u \in S$, mark u as hidden and skip all subsequent obstacles and continue with the next potential visibility neighbor.
   c) If $u \notin S$ but still within the angular range of S, perform a full visibility check as described in Section 5.3.2 and mark u as hidden if the check was positive.

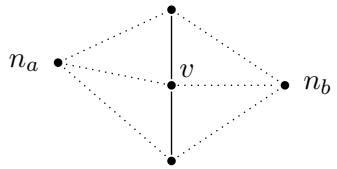
2.2.3. If u has been marked as hidden, continue with the next potential visibility neighbor.

2.2.4. If u is visible, store it in the according bin for its angle and continue with the next vertex.

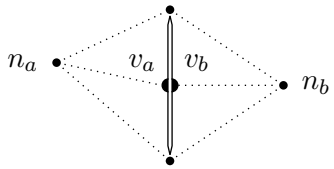
All visibility neighbors for v are grouped based on the angular ranges of v's obstacle neighbors. This procedure is described in the following section and its result is stored for the graph creation, which is described below in Section 5.3.3.

Step 2.3: Sort resulting visibility neighbors into bins

A naive approach to create a visibility graph would be to create edges between vertices that are visible to each other. This would make routing through line-based obstacles possible, as Figure 13 illustrates. Such routing behavior is, of course, not desired.



(a) Visibility graph with naive connections. The two dotted visibility edges at v allow a shortest path to go through the obstacle from n_a via v to n_b.



(b) The visibility graph with connection respecting the obstacle. Vertex v is split into two, v_a and v_b, but with exact same location (all gaps are for illustration purposes only). However, v_a and v_b are not connected, which ensures that a shortest path from n_a to n_b does not lead through the middle vertex of the obstacle.

Figure 13: Both illustrations show two of v's visibility neighbors n_a and n_b next to a vertical obstacle. The dotted lines are visibility edges and the solid lines are normal line segments of the obstacle.

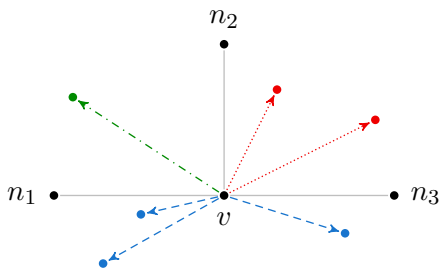


Figure 14: All visible neighbors of v, colored by their resulting bin. The neighbors n_1, n_2 and n_3 are added to both adjacent bins, for example, n_2 is added to the green (left) and red (right) bins.

To correctly split and connect vertices, it must be known which visibility neighbors are within the angular range of which two obstacle neighbors. Figure 14 illustrates this by differently colored visibility neighbors of v, resulting in three bins for each of the three angular regions. Therefore, each bin of a vertex covers the area between two adjacent obstacle neighbors. Later, in step 3 of the overall algorithm, each bin leads to one new vertex, which is connected to the corresponding visibility neighbors as illustrated by Figure 13b.

5.3.3 Step 3: Visibility graph creation

The third step is the graph generation using the grouped visibility neighbors from the previous step. One vertex is created for each visibility neighbor bin, which corresponds to the angular range between two adjacent obstacle neighbors. To correctly connect vertices in the output graph, as

described in Section 5.3.2, mappings between output vertices in the generated graph, input vertices of the dataset and visibility neighbor bins have to be created.

Two iterations over all vertices are performed. The first iteration is used to create one output vertex per bin (so-called *bin-vertex*) and to create a mapping of each newly bin-vertex to the corresponding original input vertex. In Figure 13, v_a and v_b are the bin-vertices corresponding to the obstacle vertex v. In a second iteration, this mapping is used to correctly connect each bin-vertex u to every other bin-vertex which bin contains u. Figure 15 illustrates this with two bin-vertices and color-coded angular ranges of the bins: The bin-vertex v is connected to bin-vertex u because the bin of u contains v. The same applies to u being connected to v since u is within v's bin.

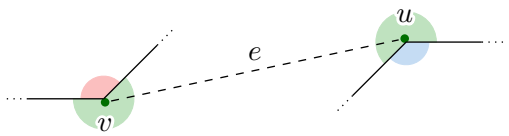


Figure 15: Edge e connecting the two bin-vertices v and u corresponding to the green angular ranges representing the bins of v and u.

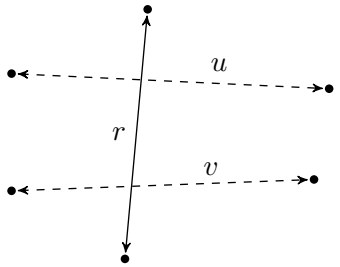
Without a mapping between bin and bin-vertex, it would not be clear to which vertex to connect since all bin-vertices of the same input vertex have the exact same location, which is the problem illustrated earlier in Figure 13: Without such a mapping, it would be unclear whether to connect n_a to v_a or to v_b. Both are on the exact same location but correspond to different angular ranges.

The resulting spatial graph, together with some of the mappings, which will also be used to answer routing queries, are stored in an instance of the `HybridVisibilityGraph` class. It contains an instance of the `SpatialGraph` class, provided by the MARS framework, along with all data needed to correctly add and connect new vertices during routing.

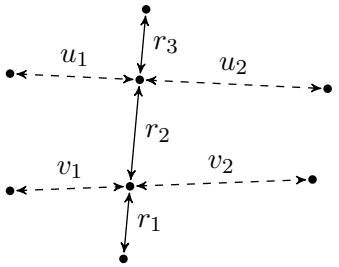
5.3.4 Step 4: Merging a road network into the visibility graph

Allowing a route to alternate between road and visibility edges, required the road network to be merged into the visibility graph. This is done by cutting and connecting all road and visibility edges at their intersection points, as illustrated in Figure 16. This final graph allows any routing algorithm to switch between road and visibility edges without further instructions or adjustments to the existing routing algorithm.

In the following, all edges are bidirectional and counted once, even though the MARS implementation of the `SpatialGraph` models bidirectional edges as two separate directed edges. The runtime complexity of the road merge operation is in $\mathcal{O}(|E_R| \cdot |E_V|)$ with E_R containing all road and E_V all visibility edges. This complexity arises from the fact that a road segment intersecting with n visibility edges leads to n additional road segments, n new intersection vertices and n new visibility edges connecting them. In the worst case, a road segment intersects with all $|E_V|$ many visibility edges increasing the number of edges by $2 \cdot |E_V|$ and the number of vertices by $|E_V|$. Therefore, up to $|E_R| \cdot 2 \cdot |E_V|$ new edges and $|E_R| \cdot |E_V|$ new vertices are created. This means that, even though runtime is linear in the input size $|E_R|$, the size of the output graph is significantly higher than of the input graph. Fortunately, as shown in Section 4.2.4, road networks are typically very sparse networks.



(a) Before merging r into the graph with six vertices and six edges.



(b) After correctly splitting and merging r. The graph size increases by one vertex and four directed edges per intersection.

Figure 16: Merging the road edge r with two bidirectional visibility edges v and u.

5.4 Answering shortest path queries

The process of answering routing queries is directly implemented in the `HybridVisibilityGraph` class and a custom weighting function can be passed to alter the routing behavior. The implementation performs the following steps:

1. Add vertices at the source and destination locations of the routing query to the graph.
2. Determine visibility edges for the two newly created vertices and add them to the graph.
3. Use a normal shortest path algorithm, such as A*, together with a weight function to find the optimal path from the source to the destination.
4. Remove all previously added vertices and edges to restore the augmented graph to the original one to have a clean state for future routing requests.

Steps 1, 2 and 4 are only performed for newly added vertices, i.e. for locations that did not already exists in the graph.

Because step 2 is already implemented for the general graph generation, the core method `VisibilityGraphGenerator.GetVisibilityNeighborsForVertex()` can be used without further adjustments. The most complex part is to determine the correct existing vertices to connect to. Thanks to the mappings of neighbor bins and their covered angle areas, created during graph generation, the correct bin for each newly added vertex can be determined and a bidirectional edge be created. The implementation required very little code and only simple filtering operations on the bins.

As mentioned above, the graph-based shortest path algorithm, A* in this implementation, can now be executed on the augmented graph.

Fortunately, for the clean-up step performed after the shortest path has been determined, the `SpatialGraph.RemoveNode(int)` method removes the vertex given by its ID and also removes all adjacent edges. This makes step 4 very easy since step 1 and 2 yield the IDs of all created vertices.

The performance of routing and quality of the determined routes depend on several factors, with are discussed in the next chapter.

6 Evaluation

In this chapter, the implementation of the hybrid routing algorithm is evaluated regarding performance, correctness and usefulness. Next to the actual results, methods and design details on the evaluation are described as well.

6.1 Methods & Measurements

The performance evaluation uses different datasets to measure graph generation times, routing times and memory consumption.

6.1.1 Collected data

The collected data consists of time and memory usage measurements on the detail level of single algorithm steps. Also, the amount of data is measured, namely the number of edges and vertices at various steps in the process, as well as the length of routes, which includes the beeline and actual route distances.

6.1.2 Datasets

Two overall types of datasets are used: Pattern-based datasets are created using a recurring pattern of various sizes. OSM-based datasets use differently sized extracts from OpenStreetMap. While the OSM-based datasets contain obstacles and roads (except in the "without roads/obstacles" datasets), the pattern-based datasets do not contain any roads.

For both types, several categories of datasets exist with different properties:

Maze pattern Datasets with touching and collinear linestrings forming a maze-like structure.

Rectangle pattern Datasets with numerous differently sized, positioned and rotated rectangles.

Circle pattern dataset with differently sized circles consisting of many vertices.

OSM city Real-world extracts from the OpenStreetMap database with data from the city of Hamburg, Germany. The data has been filtered to remove all over- and underground features.

OSM rural Equivalent to the "OSM city" dataset, but located outside the city of Hamburg and therefore containing more natural obstacles (lakes, ditches, forest), more open spaces and less regular distributed buildings.

OSM export without roads OSM extracts but without roads.

OSM export without obstacles Analogous to the "without roads" datasets, but without any obstacles, i.e. buildings, walls and natural areas such as lakes and forests.

The “OSM city” and “OSM rural” categories each contain six datasets of the sizes 0.5, 1, 1.5, 2, 3 and 4 km^2. The two OSM categories “without roads/obstacles” use the 4 km^2 datasets from the city and rural categories. All OSM-based datasets are filtered to not contains any over- or underground features, because the hybrid routing algorithm was not made to handle this third spatial dimension.

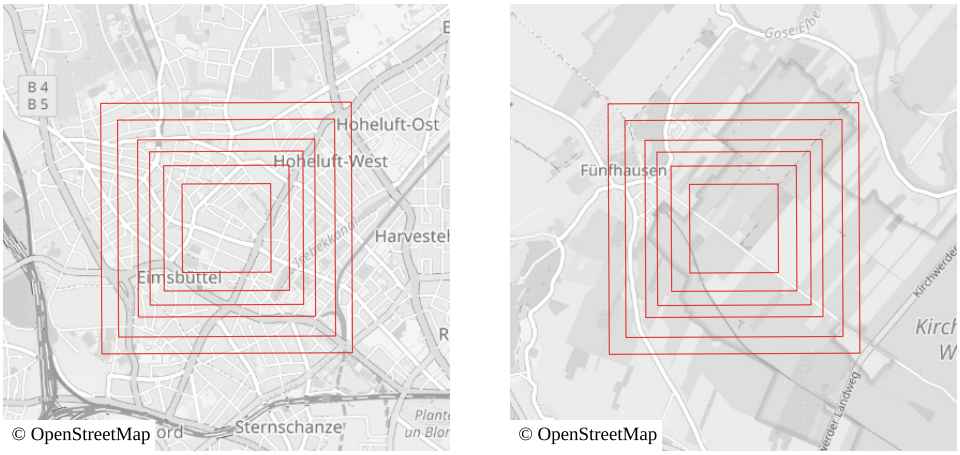


Figure 17: All six regions from 0.5 km^2 to 4 km^2 of the “OSM city” datasets (left) and “OSM rural” datasets (right).

6.1.3 Optimizations

As described in Chapter 5, several optimizations were made to the implementation. The effectiveness of these optimizations was evaluated using the 4 km^2 “OSM city” dataset. Each of the following optimizations was deactivated or replaced for the evaluation:

Shadow areas Instead, every visibility check was performed using the custom intersection check described in Section 5.3.2.

Custom intersection check The custom intersection check was replaced by the `RobustLineIntersector` class from the NTS to determine intersections between line segments.

BinIndex Instead, the `Bintree` from the NTS was used.

Convex hull The restriction to only consider vertices on the convex hull of obstacles was removed.

Valid angle areas Considering only potential visibility neighbors within certain angular ranges was deactivated.

kNN search Instead, all visibility neighbors in all directions were determined.

6.1.4 Measurement method

Measuring the performance was done by a small agent-based simulation project called `HikerModel`, consisting of one agent consecutively visiting a given list of coordinates (routing waypoints) and using a given dataset as input to the hybrid routing algorithm. The coordinates are given via a linestring within a GeoJSON file, of which each coordinate of the linestring is visited by the agent in order. Each waypoint was within the range of the dataset, meaning each coordinate was surrounded

by obstacles. The Euclidean distances (beeline distances) of the line segments within this linestring were distributed evenly to measure the required routing time relative to the distance and dataset size.

Because the OSM datasets within one category cover differently sized areas, each waypoint linestring of an OSM-based dataset contains all waypoints of the next smaller one plus some additional ones. This means that the waypoints of the smallest dataset are also used by every other dataset.

6.1.5 Technical considerations

The measurement was done by an auxiliary class `PerformanceMeasurement`, which provides a method that measures the execution time of a passed function delegate.

As part of its memory management, C#/.NET uses automatic garbage collection, adding unavoidable noise to the measurements. Unfortunately, the garbage collector cannot be turned off and controlling it is only partially possible.

Before each measurement, the garbage collector was triggered to provide equal circumstances to all iterations. This was done by the `GC.Collect()` and `GC.WaitForPendingFinalizers()` methods of the .NET framework. Using these two methods forces a garbage collection and waits for it to finish[1].

To prevent the garbage collection from interfering with the execution, a 256 MiB large no-GC-region is placed around the measured function call via `GC.TryStartNoGCRegion(256 * 1024 * 1024)`. Introducing this no-GC-region noticeably reduced noise in the measured times.

C#/.NET also uses just-in-time (JIT) compilation changing the code during runtime, which potentially yields different measurements for subsequent method calls. JIT compilation can not be turned off for regular .NET executions via the command `dotnet program.dll`. An alternative would be the usage of ahead-of-time (AOT) compilation, negatively affecting the performance of LINQ operations[2], which are frequently used in the implementation. Because both compilation strategies have disadvantages, the default JIT compilation was chosen.

The dynamic behavior of JIT compilation was mitigated by calling the measured function three times without storing the measurement results (warm-ups) before executing it five times and storing these last five results. This ensures that any JIT compilation and garbage collection of prior code was performed during the warm-up iterations and interfered with the actual iteration as little as possible.

Another step to get stable and reproducible results was increasing the process priority to exclusively use one CPU core on which the single-threaded application ran. Increasing the process priority was done by settings the `Thread.CurrentThread.Priority` to `ProcessPriorityClass.High`, which required root permissions on Linux systems.

6.1.6 System and hardware

The measurements were performed on an up-to-date Arch Linux operating system (Kernel 6.4.3) with .NET Core 7.0.107 and MARS framework 4.5.2. Apart from necessary operating system processes and a minimal desktop environment, no other applications ran during the performance measurements.

The hardware consisted of an octa-core Intel® Xeon® E3-1231 v3 CPU at 3.40 GHz, a total of 16GB DDR3 1333 MHz RAM and a Samsung EVO 850 SSD. However, the whole algorithm and the

[1] `https://learn.microsoft.com/en-us/dotnet/api/system.gc.waitforpendingfinalizers?view=net-7.0`
[2] `https://learn.microsoft.com/en-us/dotnet/core/deploying/native-aot/?tabs=net7`

`HikerModel` simulation is single-threaded. File system operations are only performed to initially load the input data and to write the measurement results after completing all executions.

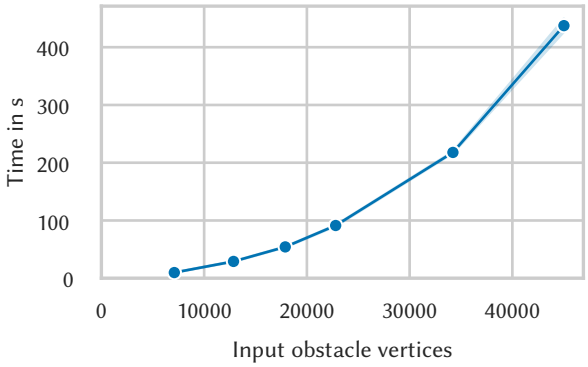
6.2 Performance evaluation

In this section, the results of the performance evaluation of the hybrid routing algorithm are presented. First, results using OSM-based datasets are discussed, followed by the artificial pattern-based datasets.

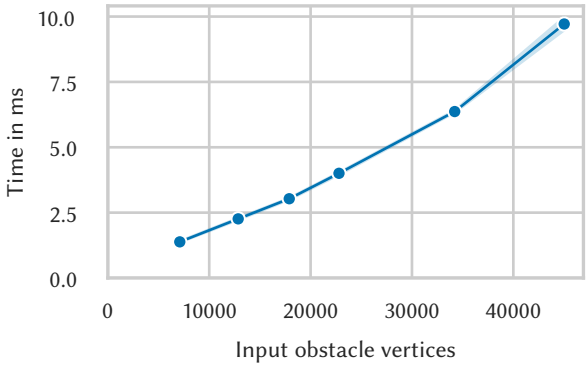
6.2.1 OSM-based datasets

Import and graph generation

A few aspects regarding the runtime behavior can be inferred from the general graph generation times shown in Figure 18 and Figure 19. First, as mentioned at the beginning of Section 3.1.1, the process of generating a visibility graph has an inherent quadratic runtime. This fact is directly visible in the import measurements shown in Figure 18a and Figure 19a, even though it is less prominent in the "OSM rural" datasets. Second, Figure 18b and 19b show an increase in the per-vertex processing time, which not necessarily grows quadratic as the "OSM rural" dataset shows.

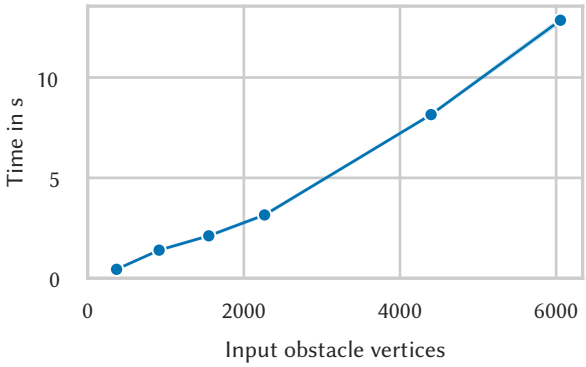


(a) Total graph generation times.

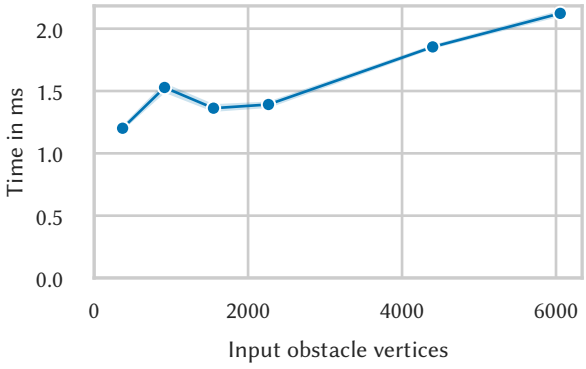


(b) Graph generation times per input vertex.

Figure 18: Graph generation times using the "OSM city" datasets.



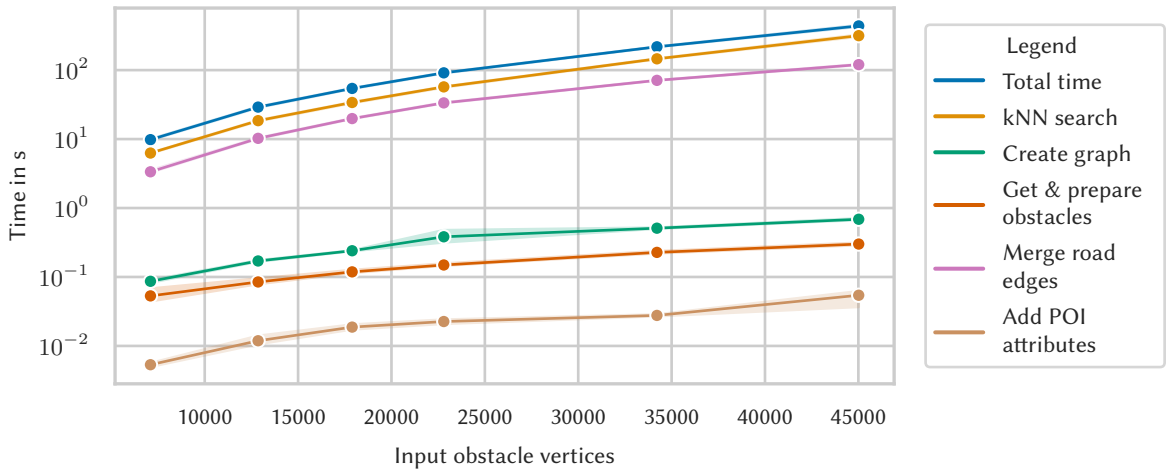
(a) Total graph generation times.



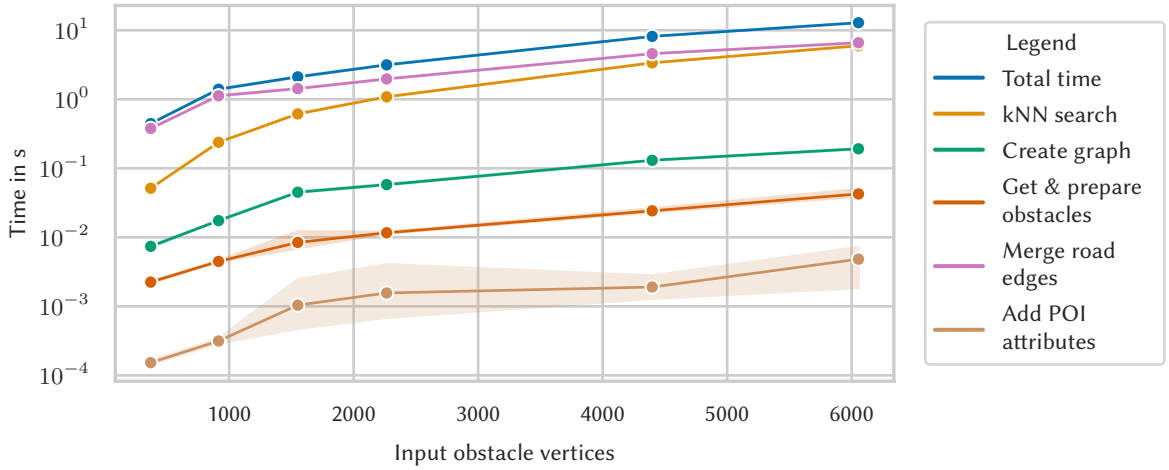
(b) Graph generation times per input vertex.

Figure 19: Graph generation times using the "OSM rural" datasets.

In Figure 20, both OSM category import times are split into the performed tasks. The task with the greatest effort regarding the required time differs from the dataset category.



(a) Import time of the "OSM city" dataset by tasks.



(b) Import time of the "OSM rural" dataset by tasks.

Figure 20: Graph generation times by task for the OSM-based datasets.

In the "OSM city" datasets, the k nearest neighbor (kNN) search is the most time-consuming task with a share on the total time of constantly over 60%. The second most time-consuming task is the merge operation, where visibility and road edges are merged. Together, these two steps account for over 98.1% of the total graph generation time throughout all "OSM city" datasets.

Different proportions of the tasks can be seen in the "OSM rural" category. Here, the merge operation is the more time-consuming task with a share of at least 51.4% on the overall graph generation time. This effect of a more time-consuming merge operation is further discussed in Section 6.2.1. The times of the kNN search and the merge operation sum up to at least 96.5% of the graph generation time.

Even though the order of tasks is different, both times of the kNN search and the merge operation are the most significant ones. The graph creation task in the "OSM rural" datasets is the only additional task throughout all tasks in all OSM-based datasets with a share of over 1%. All other tasks have a negligible share of the total graph generation time of below 1%, which even decreases with the dataset size.

Routing

Routing consists of several tasks with different impact on the total routing time, which is mainly influences by two factors. First, the dataset size is a strong influencing factor but with differently large impact depending on the dataset. Second, the structure of the data, i.e. size, distribution and number of obstacles, also influences the routing time. However, the dataset size has the strongest influence as it affects all tasks during routing. Figure 22 and 23 show the routing times from the "OSM city" and "OSM rural" datasets as well as details on the separate tasks during routing.

Starting with the total routing times, both categories show different runtime behavior relative to the route length. While the "OSM city" datasets shows no clear correlation between the beeline distance and the routing time, the "OSM rural" dataset shows a negative correlation (longer distance means shorter routing time).

This negative correlation is most prominent in the largest "OSM rural" dataset between the first and fifth routing request. Since the majority of the time is needed to connect the source and destination locations to the graph, the presence of close obstacles has a significant impact on the routing time. As illustrated in Figure 21, the destination vertex of the first routing request has no nearby obstacles. This means shadow areas can not be used efficiently, resulting in more visibility checks and a slower runtime. The close obstacles of the destination vertex of the fifth request cast large shadow areas towards many buildings and therefore cause fewer visibility checks. As a result, 86 visibility edges are connected to the source vertex of the first and 18 to the source vertex of the fifth request.

This phenomenon of longer routing times for closer waypoints does not appear in the "OSM city" dataset due to the high density of obstacles and efficient use of shadow areas. However, the effect is still visible in Figure 22a at the requests with a beeline distance between 0.5 km and 1 km. Requests of a longer processing time within this range contain vertices at junctions or wide roads with only a few surrounding obstacles.

However, the A* algorithm in both OSM-based categories takes longer for longer routes. It can therefore be generally assumed that the routing time increases with longer routes.

Request no.	1	5
Beeline distance	100.42 m	300.78 m
Routing time	519.5 ms	127.19 ms
Time per m	5.17 ms	0.42 ms

Table 2: Measured values for the first and fifth routing requests illustrated in Figure 21.

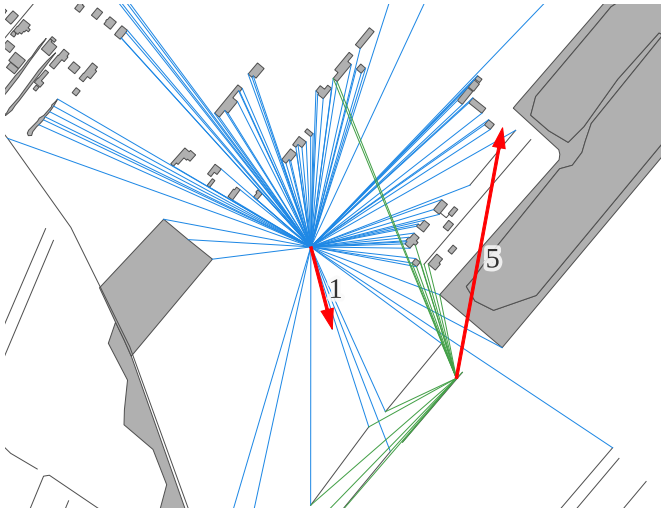
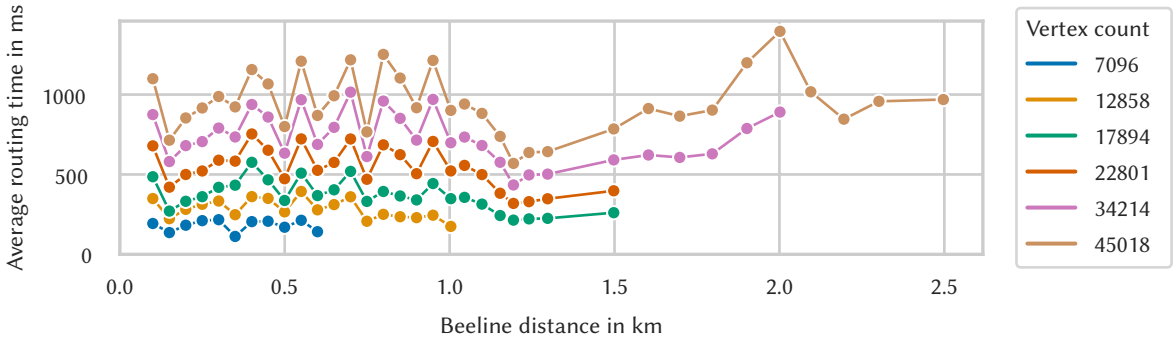
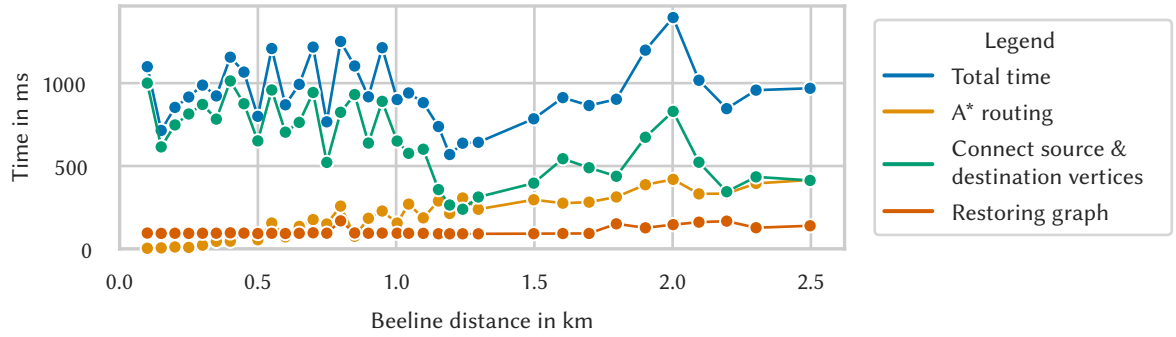


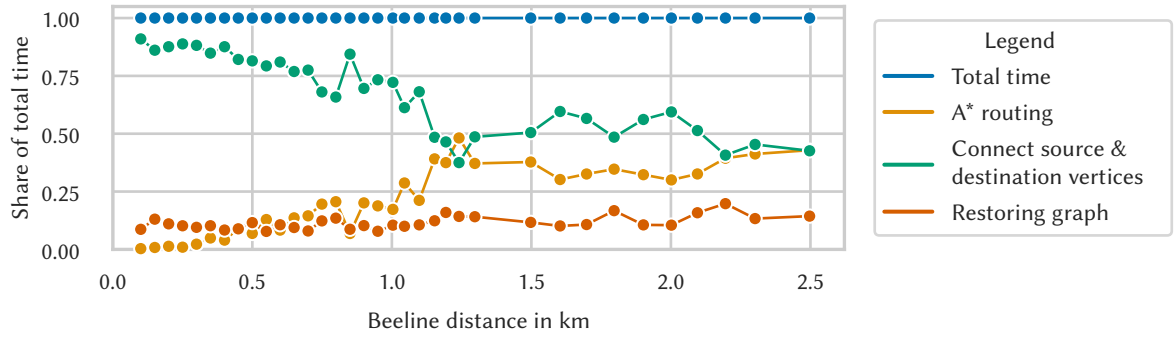
Figure 21: The first and fifth routing requests (red arrow between source and destination) with the nearby obstacles (gray). The bidirectional visibility edges of the two destinations are shown in blue and green.



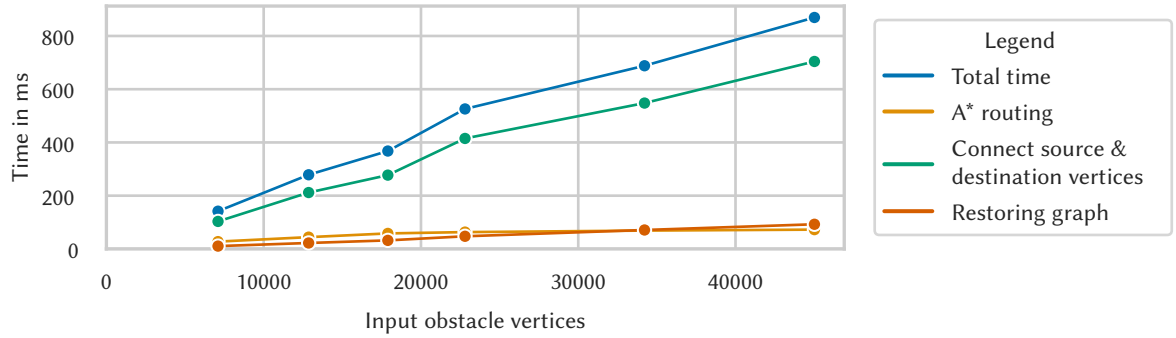
(a) Total routing times of all datasets.



(b) Each task during routing using the largest dataset with 45018 input vertices.

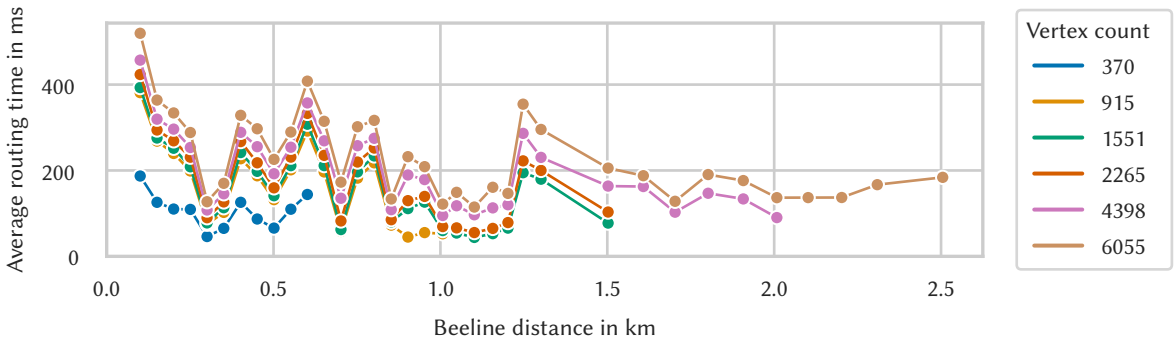


(c) Same as Figure 22b but showing the relative shares on the total time.

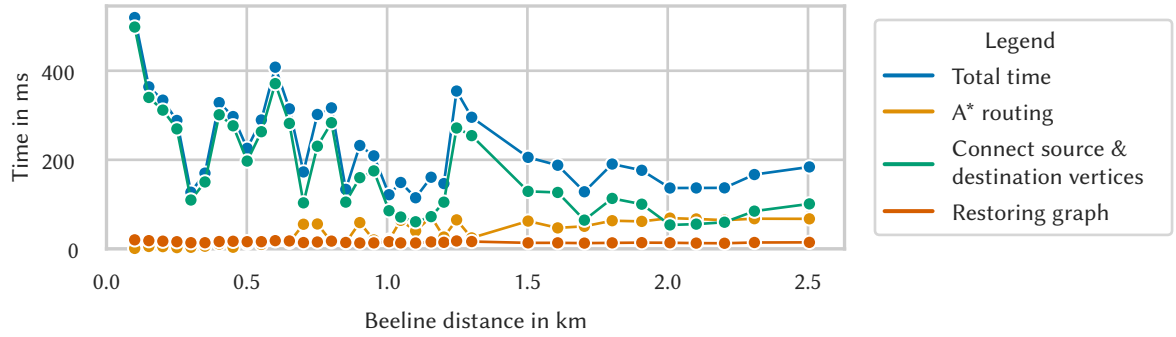


(d) Routing between the same waypoints appearing in all datasets (distance between the waypoints: 600 m).

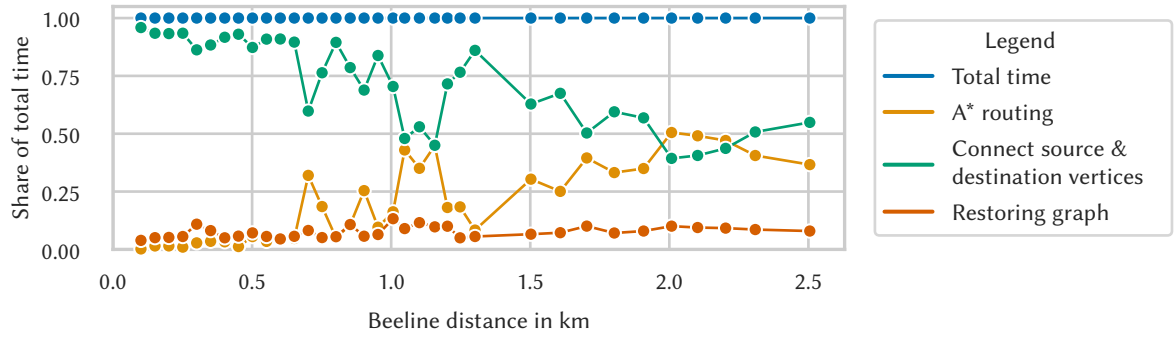
Figure 22: Routing time statistics of the "OSM city" datasets.



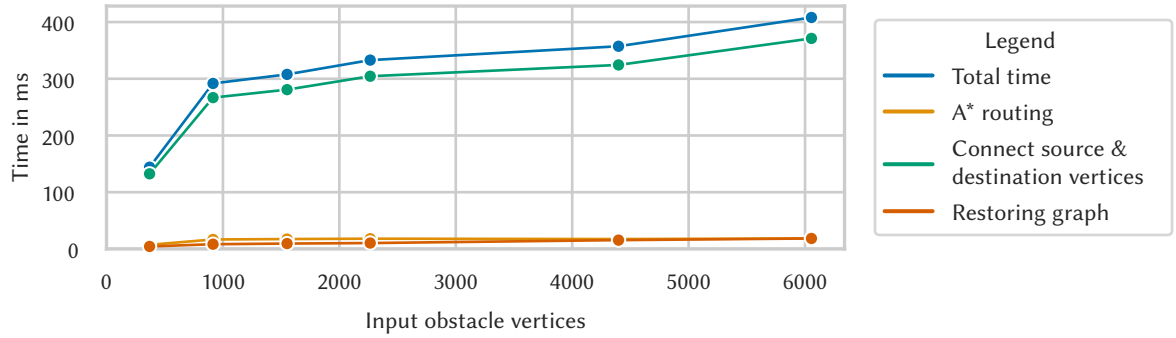
(a) Total routing times of all datasets.



(b) Detailed routing times using the largest dataset with 6055 input vertices.



(c) Same as Figure 23b but showing the relative shares on the total time.



(d) Routing between the same waypoints appearing in all datasets (distance between the waypoints: 600 m).

Figure 23: Routing time statistics of the "OSM rural" datasets.

Dataset without roads or obstacles

The normal OSM-based datasets contain both roads and obstacles. However, datasets containing only roads or only obstacles are thinkable and can also be used with the hybrid routing algorithm. For this evaluation, the 4 km^2 datasets of the "OSM city" and "OSM rural" categories were filtered yielding three datasets for each category: A normal dataset with roads and obstacles, one dataset without roads and one without obstacles. In the dataset without roads, all road edges that do not cross buildings were removed, meaning building passages remained within this dataset to ensure the reachability of backyards. The influences of the two filters are discussed in the following.

Operation	Normal	No roads	Decrease compared to normal	No obstacles	Decrease compared to normal
kNN search	315.86 s	288.69 s	8.60%	0.01 ms	99.99%
Create graph	0.69 s	0.68 s	0.84%	0.01 ms	99.99%
Get obstacles	0.30 s	0.29 s	3.28%	9.54 ms	96.82%
Merge road edges	120.13 s	30.79 s	74.37%	484.55 ms	99.60%
Add POI attributes	0.06 s	0.03 s	53.81%	7.63 ms	85.97%
Total time	437.34 s	320.57 s	26.70%	525.61 ms	99.88%

Table 3

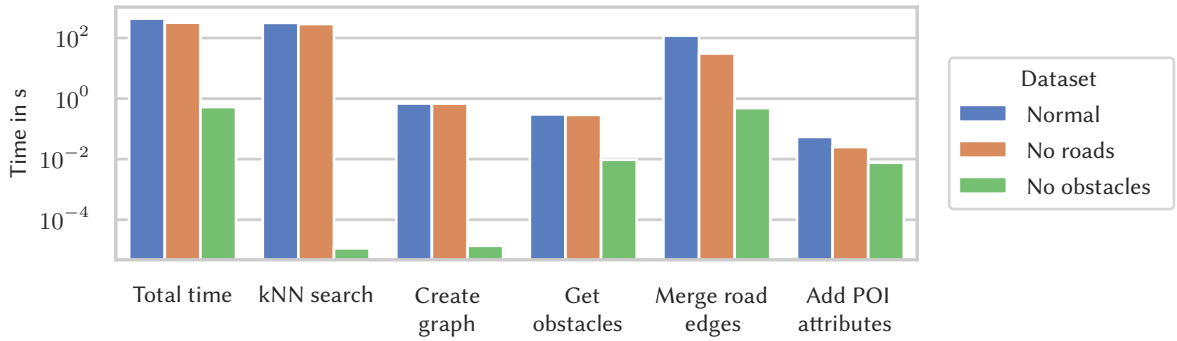


Figure 24: Comparison of the normal 4 km^2 "OSM city" dataset (blue) with the same dataset but without roads (orange) and obstacles (green).

Figure 24 lists and illustrates the results for the "OSM city" datasets. Some noteworthy insights can be inferred from these data:

- The kNN search and visibility graph creation times were significantly reduced for the dataset without obstacles, which is expected since both operations take only obstacles into account.
- The merge operation for the dataset without roads took longer than for the dataset without obstacles. This is due to the 175 remaining building passages within the "no roads" dataset, which means this operation still merged road edges and therefore required some time. However, compared to the normal dataset, the merge operation of the "no road" dataset only required 25.63% of the time, which does not match the number of remaining road edges of 4.82% (175 edges) compared to the normal dataset (3629 edges).

Because building passages consist of short segments, their start and end vertices (dead-end vertices) need to be connected to the rest of the graph. This requires the creation of visibility edges, which is an expensive operation. The normal dataset contains 742 and the dataset without roads (and only building passages) 346 such dead-end vertices, which yields a relatively long merge operation time.

- In the "no obstacles" dataset, the merge operation took 0.4% of the normal time. It is a low but expected value because no visibility edges exist and no merge of edges took place.

Operation	Normal	No roads	Decrease compared to normal	No obstacles	Decrease compared to normal
kNN search	5,957.97 ms	5,924.27 ms	0.57%	0.01 ms	99.99%
Create graph	191.68 ms	210.40 ms	-9.76%	0.01 ms	99.99%
Get obstacles	42.53 ms	40.76 ms	4.15%	0.78 ms	98.16%
Merge road edges	6,599.78 ms	5.67 ms	99.91%	32.79 ms	99.50%
Add POI attributes	4.83 ms	1.53 ms	68.21%	0.42 ms	91.26%
Total time	12,853.51 ms	6,201.96 ms	51.75%	37.19 ms	99.71%

Table 4: Measurements for the 4 km^2 "OSM rural" normal, no roads and no obstacles datasets.

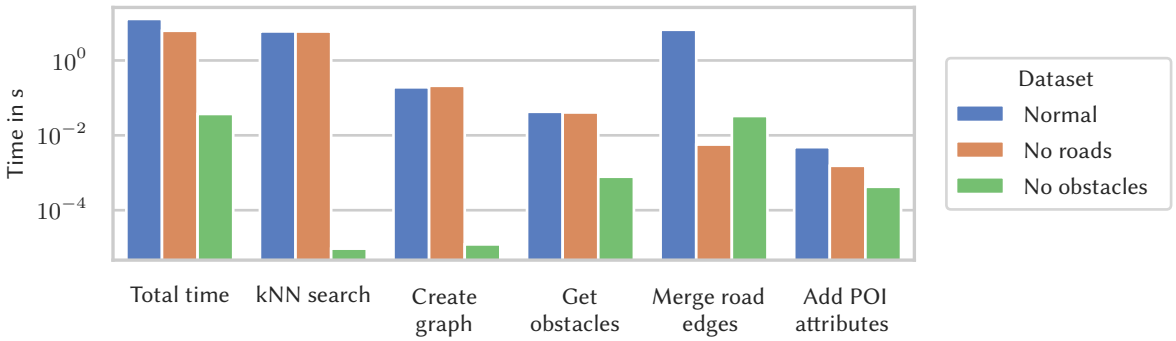


Figure 25: Comparison of the normal 4 km^2 "OSM rural" dataset (blue) with the same dataset but without roads (orange) and obstacles (green).

Figure 25 lists and illustrates the results for the "OSM rural" datasets. There are some differences to the results of the "OSM city" dataset:

- The rural dataset contains no building passages and therefore shows the expected reduction in time for the merge operation.
- A comparison of the kNN search and road merge operations using the normal dataset shows that the merge operation actually took 10.77% longer than the kNN search. Results of the normal "OSM city" dataset shows the opposite relation with 61.97% less time required for the merge operation. A possible explanation is the ratio of road to visibility edges, which is 1:33 for the city dataset (33 visibility edges per road edge) and 1:264 for the rural dataset. This means that more intersections per road edges were processed using the rural dataset, which was actually the case: Using the city dataset, the number of road edges increased due to splitting at intersections

by a factor of 66 and within the rural dataset by a factor of 221. Therefore, the time spent on one road edge was significantly higher within the rural dataset and likely led to the significant increase in time required for the merge operation.

- The graph creation time using the "no roads" dataset is especially noteworthy since it shows an *increase* in processing time compared to the normal dataset. Because no algorithmic reason exists, that would result in an increase of the graph generation time, and because the measurements contain outliers, no further investigation took place finding an explanation for this increase.

Apart from these differences and unique characteristics, both dataset categories show similar results for the kNN search, graph generation and overall processing time. First, removing all obstacles eliminates the main complexity and significantly decreases the processing time. Therefore, decreasing their amount will significantly reduce the processing time due to the overall quadratic runtime complexity of the kNN search. Second, removing roads does not affect the kNN search and graph creation, but the total processing time decreases due to the faster merge operation. Third, even though it is not a significant part of the processing time in the first place, getting obstacles and adding attributes to POIs is also reduced when removing obstacles or roads.

6.2.2 Pattern-based datasets

Import and graph generation

The pattern-based datasets contain repeating patterns of obstacles and thus have a much more regular structure compared to OSM-based datasets. This is also reflected in the results of the graph generation containing fewer deviations as shown in Figure 26. Due to the absence of roads, the overall graph generation time is mainly determined by the kNN search. Only smaller datasets of up to an input vertex count of about 3,500 (depending on the dataset category) show a combined time of all other tasks of up to 5% on the total graph generation time. All datasets show the already mentioned quadratic runtime complexity of the kNN search and no anomalies were observed during the overall graph generation. The maze dataset's detailed absolute and relative times can be seen in Figure 27.

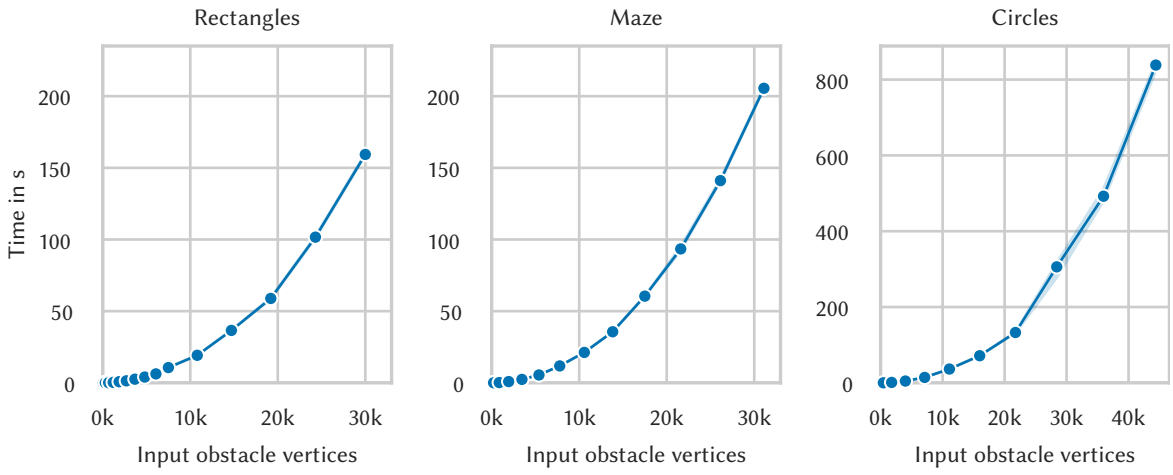


Figure 26: Total graph generation times for all three pattern-based dataset categories.

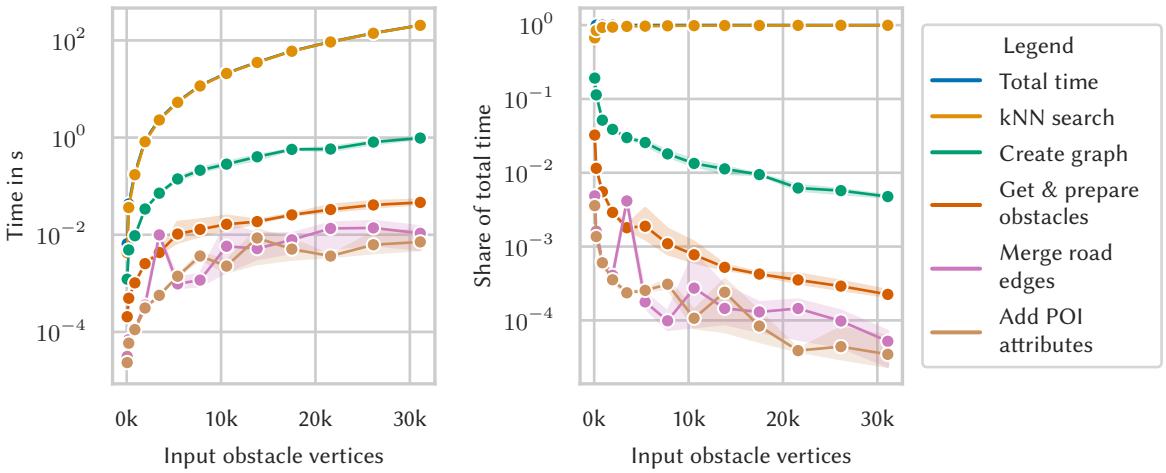


Figure 27: Tasks during graph generation using the maze dataset in absolute time (left) and relative share on the total time (right). The kNN search determines the total processing time covering the data points for the total time.

Routing

Because the pattern-based datasets did not contain any roads, routing times in these datasets solely depended on the size of the visibility graph and therefore on the time required to connect the source and destination vertices, which is the most time-consuming task during routing. Connecting vertices uses the kNN search of the graph generation and its runtime therefore increases quadratically with the number of vertices in the graph.

The A* algorithm uses this quadratically growing graph and therefore shows a quadratic runtime, too. However, this quadratic runtime is not directly visible in all measurements and a linear regression on the maze- and rectangle-based measurements yield a similarly good correlation. The effect of a quadratic complexity might therefore only be significant in larger datasets.

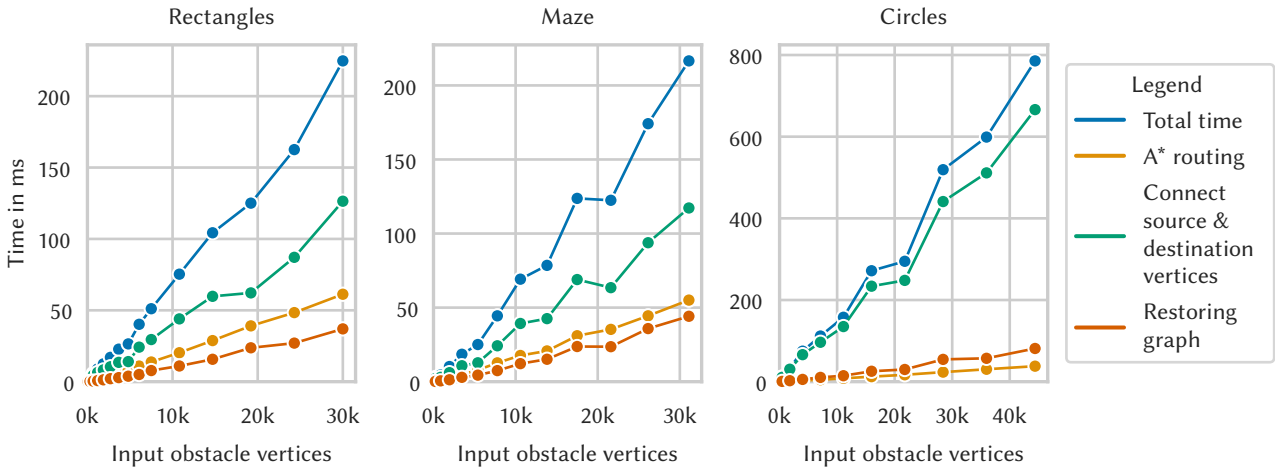


Figure 28: Routing time between the longest distant waypoints in each pattern-based dataset.

6.2.3 Memory consumption

In this section, the memory usage of the hybrid routing algorithm is analyzed using the 4 km^2 “OSM city” dataset as presented in Figure 29.

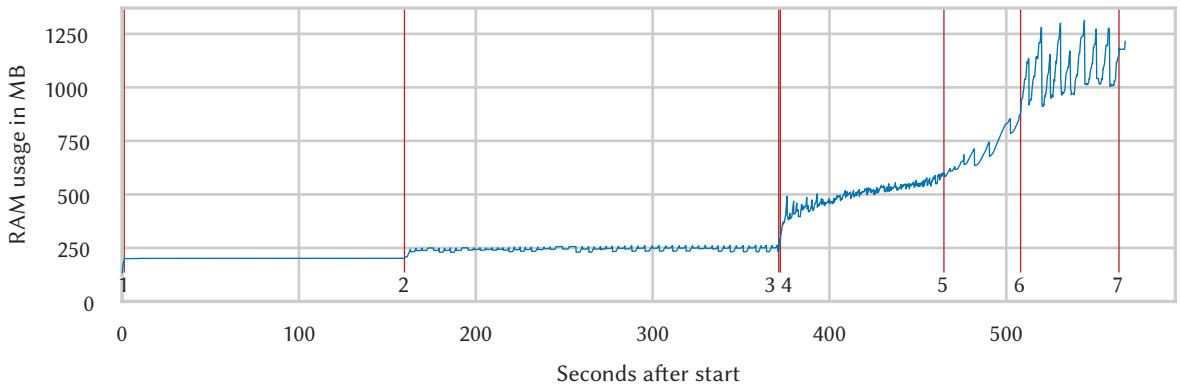


Figure 29: Memory usage of the 4 km^2 "OSM city" dataset. The marks represent the following operations: **(1)** Start of unwrapping and triangulating obstacles and also determining obstacle neighbors, **(2)** start of kNN search, **(3)** create visibility graph, **(4)** prepare road merge operation, **(5)** start of merging road edges into graph, **(6)** end of graph generation and start of agent routing, **(7)** end agent routing and end of simulation.

Until the creation of the visibility graph (3), the memory requirement only shows a very slight linear increase due to simple mapping creations without new and larger data structures. With the start of the merge operation (4), memory usage rises due to additional data structures and newly added vertices and edges to the graph. The period between (4 to 5) determines dead-end road vertices, which are connected with new nodes separately to the graph. Merging road edges (5 to 6) then introduces many new nodes and edges due to the large number of intersections. Routing requests (6 to 7) also require a certain amount of memory and the garbage collection regularly frees up the memory of prior requests resulting in recurring spikes in the memory usage.

Possible memory optimizations and other performance improvements are discussed in Section 7.1.

6.2.4 Optimizations

Impacts of the implemented optimizations were measured using the 0.5 km^2 "OSM city" dataset as listed in Table 5. Noteworthy is the discrepancy between the product of all speedup/slowdown factors and the overall speedup of about 71 when all optimizations are active. Activating all optimizations should, according to the separate speedup factors on the right side of Table 5, result in a speedup of over 255, but only 71 can be observed. This is because the filtering methods interfere with each other. When a vertex is filtered out by one filtering method, it reduces the efficiency of the other filtering methods. Therefore, the selective (de)activation of single optimizations yields a differently strong impact than the (de)activation of all optimizations together.

Some filtering optimizations presented in Chapter 5, namely the convex hull and valid angle area filtering, remove vertices that might be useful when determining non-shortest but otherwise optimal paths based on a custom weighting. Deactivating these two optimizations, and solving the problem of non-optimal paths, results in a measured slowdown factor of about 1.9.

	All activated except		All deactivated except	
Optimization	**Time**	**Slowdown**	**Time**	**Speedup**
Shadow areas	121.6 s	10.97	22.2 s	35.56
kNN filtering	11.3 s	1.02	751.7 s	1.05
Vertices on convex hull	18.6 s	1.68	287.9 s	2.74
Valid angle areas	12.8 s	1.16	322.5 s	2.44
Custom collision detection	11.6 s	1.05	769.3 s	1.02
No active optimization	788.5 s	71.13	788.5 s	1.00
All optimizations active	11.1 s	1.00	11.1 s	71.13

Table 5: Optimization impact on the 0.5 km^2 "OSM city" dataset import. The optimization of a row was selectively deactivated (left) or activated (right).

6.3 Route correctness and quality

6.3.1 Weight function

Before the correctness and quality of the routes are discussed, a brief introduction into the functioning of the weight function is given.

When determining actual shortest paths, the length of an edge is its weight, which should be minimized to obtain the shortest path. When certain edges should be preferred or avoided, a factor is determined based on edge attributes and multiplied to the edge length. Due to the minimization algorithm, a factor lower one means an edge is preferred, a factor greater one rather avoids those edges. A factor of positive infinity on an edge e completely avoids this edge since any other path has a lower total weight than a path containing e.

The implementation of the hybrid visibility graph provides three routing methods. One determines the actual shortest path (all weight factors are one), one determines a weighted path (factor of 0.8 for road edges) and one accepts a custom weight function.

All following analyses use the predefined weight function with a factor of 0.8 on road edges. The factor 0.8 is based on experience and yields routes where the agent switches between road and visibility edges. Impacts of different weight factors on the resulting paths are discussed in the following manual route analysis.

6.3.2 Correctness

In this section, theoretic considerations on the correctness of the hybrid routing algorithm are presented, which therefore does not include mistakes in the implementation. The hybrid routing algorithm is considered *correct* if the resulting path between any two given locations is the shortest possible path with respect to the obstacles of the dataset and the given weight function. In the following, the weight function represents the distance of edges to find truly shortest paths.

Edges in the generated visibility graph are the shortest possible segments between vertices of obstacles. In other words, for any two vertices v and v', a visibility edge $e = (v, v')$ is the shortest possible connection between these two vertices. A shortest path through a visibility graph is therefore equal to a shortest geometric path around obstacles in the plane.

For shortest path algorithms, such as A* or Dijkstra, the term *correct* refers to the fact that their result is the shortest possible path between two vertices in a graph. Using a correct graph-based shortest path algorithm on a visibility graph yields the shortest path in the plane. The hybrid visibility routing is therefore considered correct regarding the shortest path between existing vertices.

In addition to routing on a visibility graph, the presented hybrid routing algorithm actively connects locations to the graph as part of answering the routing query yielding a temporarily augmented graph. This means the resulting path p from a newly added vertex s to a newly added vertex t contains two additional new edges $e_s = (s, v_s)$ and $e_t = (v_t, t)$, which are not part of the normal non-augmented hybrid visibility graph, resulting in $p = \langle s, v_s, ..., v_t, t \rangle$. The correctness of the path from v_s to v_t follows from the argumentation above. The edges e_s and e_t are visibility edges and the correctness argument applies to them as well, meaning that the overall path p is shortest on the augmented visibility graph and therefore shortest in the plane under the presence of obstacles. Therefore, the overall hybrid routing algorithm is considered to be correct.

One noteworthy aspect is the filtering of vertices, as described in Section 5.3.2, which might result in missing edges. However, these filters are solely performance optimizations and can be deactivated without negative effects on the algorithms functioning, except the performance.

6.3.3 Quality of routes

Definition and measurement of route quality

An ideal route, meaning a route of maximum quality, would exactly match a path chosen by an actual pedestrian. To approximate a real pedestrian, the expected paths used in the following sections were obtained using aerial imagery, data contained in OSM and local knowledge. Therefore, an expected path is realistically walkable in the real world and would likely be chosen by actual pedestrians. This also means that an expected path contains a certain subjective component, for example, the exact location of a road crossing, the preference of shortcuts via unpaved areas or the avoidance of steps. Therefore, the quality analysis of routes uses mathematical metrics in addition to a manual analysis.

Manual route analysis

In the following examples, routes determined by graph-based routing and the hybrid routing algorithm are compared against an expected result. This expected route was created based on OpenStreetMap data, aerial imagery and local knowledge. It was *not* created to intentionally match the results from the hybrid routing algorithm.

The first situation, shown in Figure 30, illustrates the problem of missing edges for graph-based algorithms. The square "Fanny-Mendelssohn Platz" in Hamburg, Germany, is a mostly unconnected area and therefore not

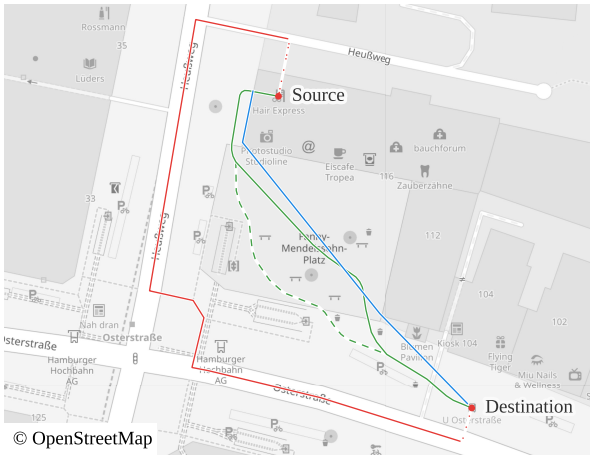


Figure 30: Graph-based route (red), hybrid routing algorithm result (blue) and the expected route (green). The dashed part of the expected result is an equally good alternative route.

traversable by graph-based algorithms. A routing request using GraphHopper[3] yields the red route, which does not cross the square due to missing edges and therefore takes a detour. The hybrid routing algorithm (blue path) does not entirely match the expected route but is a better approximation than the graph-based result. Next to this example, analyzing routes determined within the 1 km^2 "OSM city" dataset yields some noteworthy findings.

- The route quality significantly decreases with missing or wrong data as shown in Figure 30 as well as Figure 31b to 31c. Red passages are not usable in the real world, often traversing private areas, which usually contain obstacles such as fences or vegetation. As described in Section 2.1.6, data of private areas in OSM is often sparse, resulting in these unrealistic routes.
- Within densely built-up areas, larger open spaces are rare and the combination of roads and buildings form corridors where routes tend to lie. When and how often a route determined by the hybrid routing algorithm follows a road depends on the weighting function.
- Larger roads result in wide obstacle-free corridor areas where unrealistic road crossings might appear. Such a situation can be seen in Figure 32 with crossings over a six-lane road, which would be dangerous and unrealistic for real-world pedestrians.
- The weight function has a huge impact on the route quality. Figure 33 illustrates this with different preferences on road edges. Neither a complete avoidance nor a complete preference of road edges yields a realistic route. Adjusting and defining fine-grained weight functions may help to improve the route quality.
- In the "OSM city" datasets with only smaller open areas, graph-based routes and routes determined by the hybrid routing algorithm were often similar and some parts are even identical.

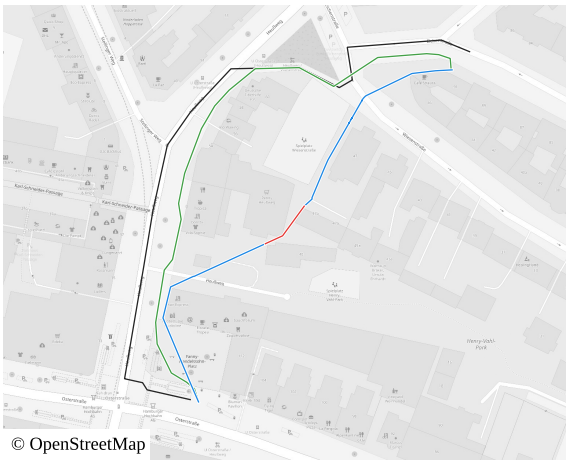
Figures Figure 34 and 35 give examples of routes in a rural area with large open spaces. Two findings can be inferred from the analysis of the rural routing results.

- Datasets with larger open areas and irregularly distributed obstacles, as in the "OSM rural" datasets, can greatly benefit from the hybrid routing algorithm because graph-based routes might take long detours due to a sparse road network.
- The difference between graph-based routes and routes from the hybrid routing algorithm is significantly larger. They might not have any location in common and graph-based routes tend to be significantly longer. This can be seen in Figure 35a with a 1.73 km long graph-based and 0.67 km long expected route. Figure 35b shows a similar behavior even though the difference in distance is smaller because the graph-based, expected and actual route share common parts.
- The accuracy and therefore the usefulness for real-world applications (such as navigation apps) heavily depends on the level of detail in the dataset. Figure 34a illustrates the problem of missing data, in this case missing ditches within the farmland. The determined route has a length of 365 m, while the shortest possible route (determined using aerial imagery as seen in Figure 34b) is 677 m long.

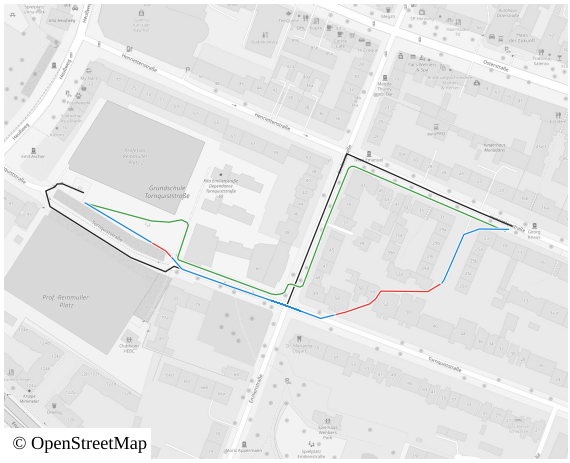
In general the manual route analysis showed that two main factors determine the route quality: the quality of the data (mainly determined by its completeness) and the weighting factor. The routes might still be good for densely built-up city datasets, even though a graph-based routing algorithm might yield similar routes.

[3] `https://www.osm.org/directions?engine=graphhopper_foot&route=53.57657,9.95210;53.57601,9.95268`

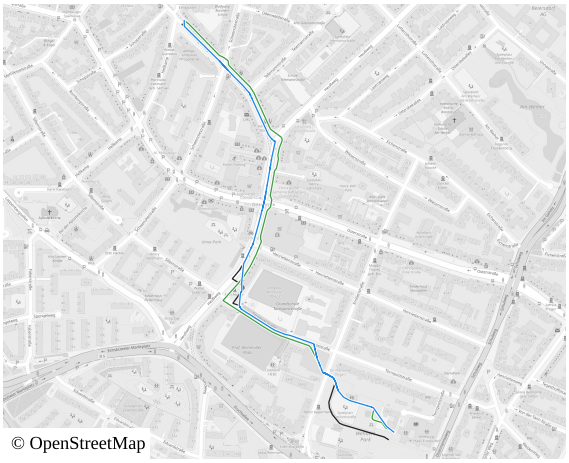
(a) Simple route with only one small passage of missing data.



(b) Missing obstacles (mainly walls and fences between buildings) lead to non-realistic routes.



(c) Missing walls, fences and hedges on private property are a common type of missing obstacles.



(d) Cities with closed rows of buildings do not provide many degrees of freedom, resulting in similar or equal routes, regardless of the algorithm.

Figure 31: Expected (green), graph-based (black) and hybrid routing algorithm results (blue) using the 1 km^2 city dataset. Unexpected parts due to missing or faulty data are marked in red.

Mathematical route quality analysis

Two mathematical approaches were used to obtain quantifiable results on the route quality. This was done by comparing the expected route to the graph-based and hybrid routing algorithm results. The first metric uses the beeline distance between two waypoints and compares it to the distances of the three corresponding routes. The second metric uses the Hausdorff distance (described below) to obtain a numeric similarity between the expected route and the two corresponding routing results. Both comparisons were performed on the first ten routing requests of the city and rural OSM datasets.

Figure 37 shows the relation between the route and the beeline distances. The hybrid routing algorithm creates the shortest routes on average, which tend to be even shorter than the expected routes. Even though determining shortest routes is the overall goal, it indicates that missing obstacles lead to unrealistically short routes. This is in line with the results of the previous manual route analysis, according to which missing data leads to short but unrealistic routes.

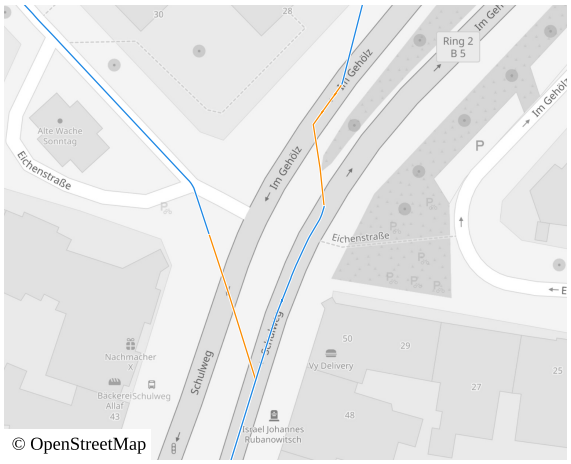


Figure 32: Two unrealistic road crossings (yellow) across a six-lane road.

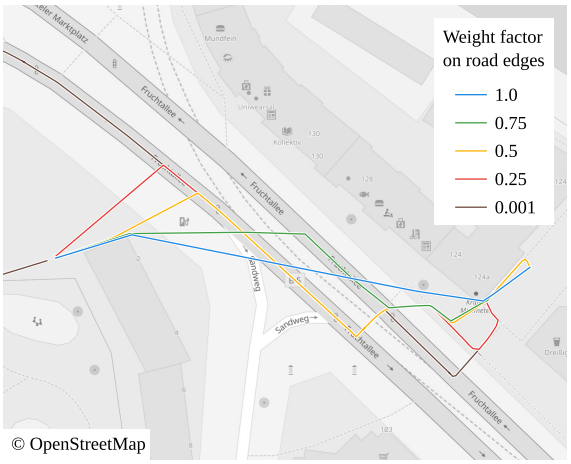
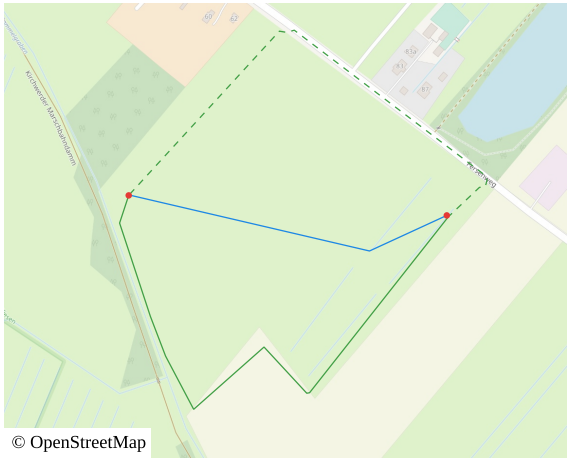


Figure 33: Influence of weight factors. Lower value result in a stronger preference for roads.



(a) The expected route differs significantly from the actual route taken by the agent. The dashed line is an alternative route under the assumption that the farmland is reachable from the upper road.

(b) Aerial imagery shows the amount of missing data, in this case numerous missing ditches within the farmland.

Figure 34: Routing on farmland illustrating the importance of correct data in the routing result showing the expected route (green) and determined route by the hybrid routing algorithm (green).

The route length comparison also supports the previous results that graph-based routing leads to longer routes with sometimes significant detours, which can be seen in the routing requests 6 and 7 in Figure 37b. Comparing routes by length only yields a vague indication of the route's quality. Therefore, the second metric uses the *Hausdorff distance* to quantify the similarity of two routes by determining the maximum distance between them as illustrated in Figure 36. Figure 38 shows the results for both OSM datasets and yields some insights into the route similarities:

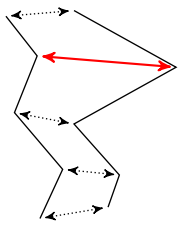
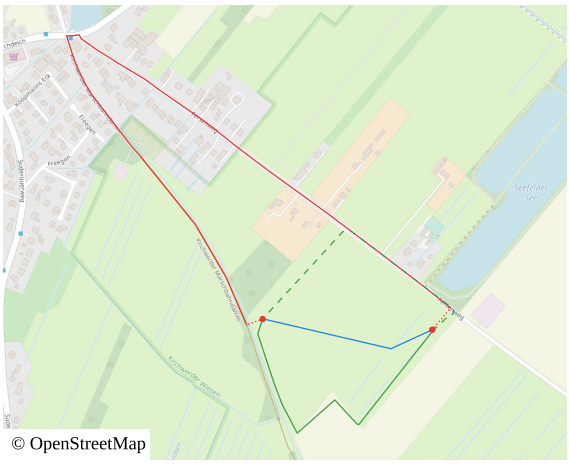
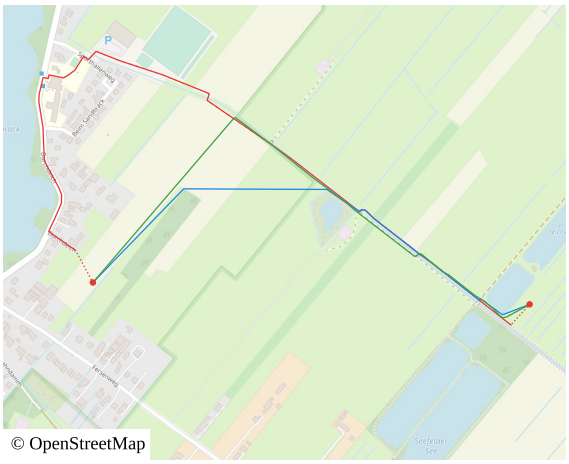


Figure 36: Hausdorff distance (red) between two linestrings.

- In both datasets, the average Hausdorff distances of the routes determined using the hybrid routing algorithm are shorter than those of the graph-based algorithm.



(a) Same waypoints from Figure 34 with the additional result of a graph-based routing request creating a 2.6 times longer path.



(b) Detours created by a graph-based routing algorithm, even though the first part of the result is equal to the one of the hybrid routing algorithm. The shortcut of the actual result (horizontal part of the blue route) is a result of missing obstacles similar to the situation visible in Figure 34b.

Figure 35: Comparison of graph-based routing results (red) with the expected route (green) and actual results (blue) of the hybrid routing algorithm.

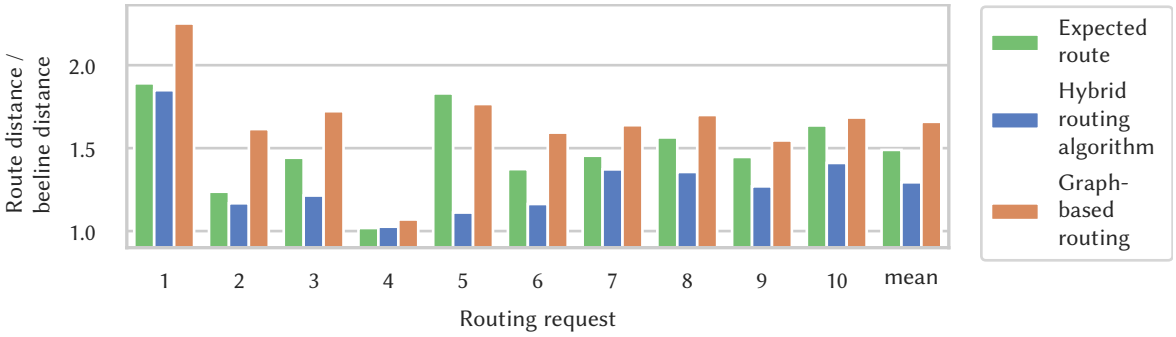
- The difference is smaller in the city dataset, which supports the aforementioned hypothesis, that the buildings in a city create corridors leading to similar routes for graph-based and geometric routing algorithms.
- Except for the routing requests 6 and 7 in the "OSM rural" datasets, the Hausdorff distances of the routes by the hybrid routing algorithm tend to be larger than the distances of the graph-based routes. This is likely due to missing data, as discussed in the previous section and illustrated in Figure 34, leading to shorter but less realistic routes.

Route quality analysis summary

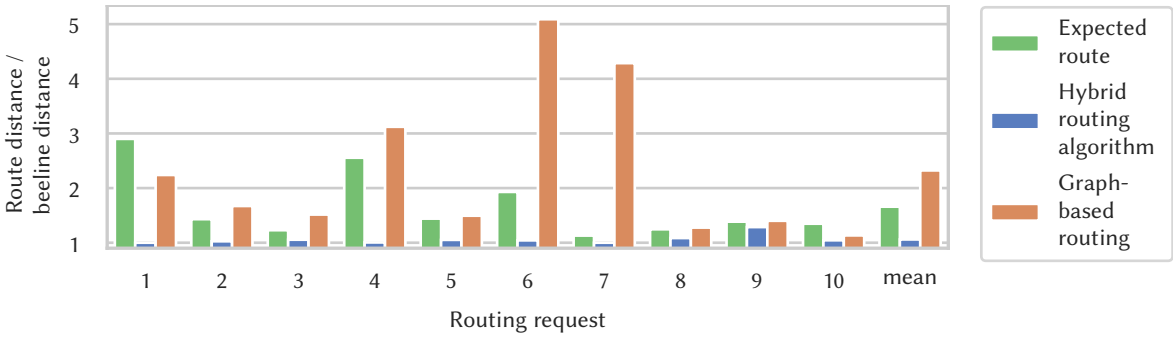
The manual route analysis and both mathematical approaches, the comparison to the beeline distance and the Hausdorff distance, yield similar results regarding the quality of routes:

- The route quality heavily depends on the data quality. Missing data, such as a small number of missing ditches on farmland or one fence between two larger buildings, results in shorter but unrealistic routes determined with the hybrid routing algorithm.
- Cities and densely built-up areas form corridors through which geometric- and graph-based routing results are very similar.
- Graph-based routes between waypoints in rural or remote areas tend to create long detours. Such detours often are unrealistic, especially in agent-based models simulating scenarios in which pedestrians do not necessarily respect access restrictions, such as in emergency or evacuation scenarios.

In scenarios with a high quality in data, for example when official indoor room plans of buildings are used or when a manual on-ground survey ensured the data quality, the hybrid routing algorithm will likely create high quality routes well reflecting the behavior of real pedestrians.

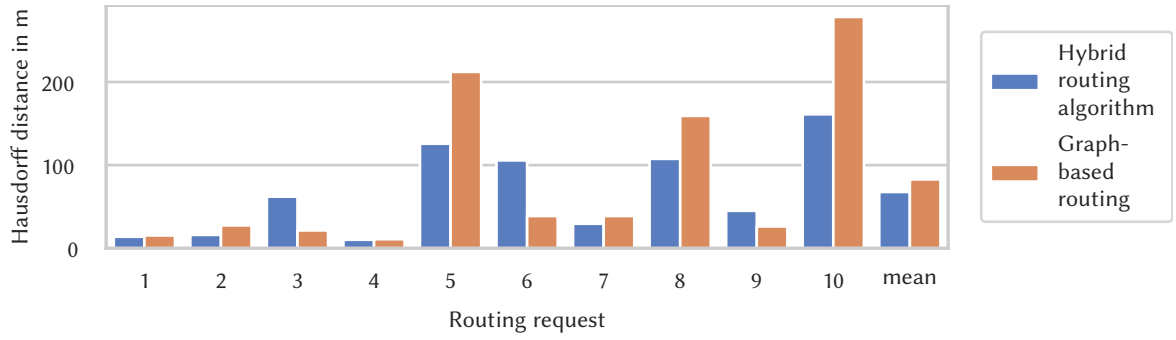


(a) "OSM city" dataset.

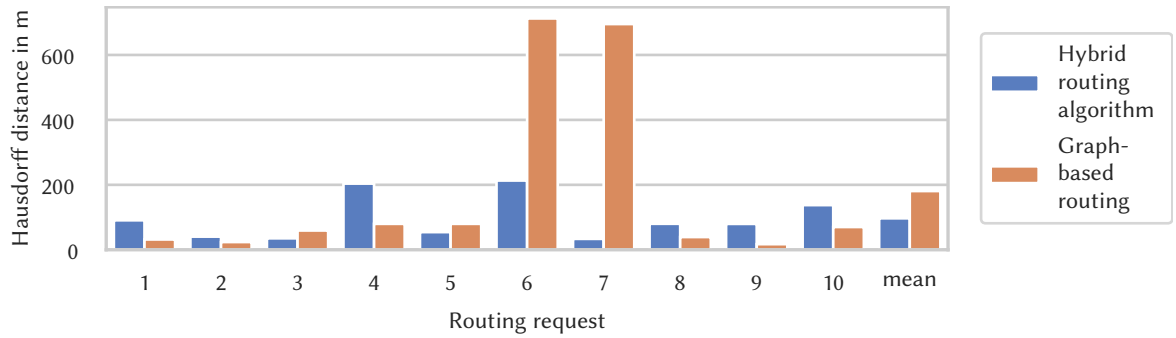


(b) "OSM rural" dataset.

Figure 37: Relative route distances compared to the beeline distance between the waypoints of each routing request using the 0.5 km^2 OSM datasets.



(a) "OSM city" dataset.



(b) "OSM rural" dataset.

Figure 38: Hausdorff distances between the expected routes and the corresponding routes of the hybrid routing algorithm and graph-based routing using the 0.5 km^2 OSM datasets.

7 Conclusion

The presented *hybrid routing algorithm* combines graph-based with geometric routing and determines shortest paths between arbitrary source and destination locations. The geometric routing component is based on visibility graphs containing only edges between vertices that are visible to each other. Edges in visibility graphs are therefore the shortest connection between two vertices, which is a useful property for finding general shortest paths. Generating a routable visibility graph is one core task of the hybrid routing algorithm.

Merging road edges from the input dataset into the visibility graph is the second core task of the hybrid routing algorithm. During this step, intersecting road and visibility edges are split and connected to allow paths to alternate between these two types of edges.

The graph-based routing component consists of a normal graph-based algorithm, A* in this case, and may use speedup techniques. The source and destination locations are added and connected to the graph, as far as they are not represented by existing vertices, which allows the determination of shortest paths between arbitrary locations.

7.1 Future work

Even though the presented algorithm works as intended, some problems are known and a certain potential for further enhancements exists. This section discusses these problems and potential solutions, lists additional technical enhancements and outlines additional functionalities the hybrid routing algorithm might benefit from.

7.1.1 Known problems

The runtime and memory performance of the graph generation and routing routine offer potential for improvements. This can happen by optimizing the current implementation or implementing a different graph generation algorithm.

The first approach might use better data structures for vertices and obstacles. In the current implementation, for every vertex, every other vertex in the dataset is considered a potential neighbor and then filtered out using e.g. valid angle and shadow areas. Using a spatial index to query vertices in certain areas might increase performance since not all vertices need to be checked. However, any additional data structure also introduces additional overhead and might only be beneficial for datasets of a certain size. Therefore, the usefulness of this approach needs to be tested and evaluated.

The second approach is a complete reimplementation of the edge creation. Such a reimplementation might use approaches mentioned in Sections 3.1.1 and 4.3.2, which showed promising sub-quadratic runtime complexities.

In datasets with very few obstacles, the connectivity between road and visibility edges may be reduced due to a sparse visibility graph, meaning edges are missing to switch between road and visibility edges properly. Figure 39 illustrates this: the green expected path cannot be used due to missing edges, which leads to the suboptimal red route. Forcing the creation of additional edges, e.g. by connecting all vertices of roads or splitting roads at certain distances, might solve this problem.

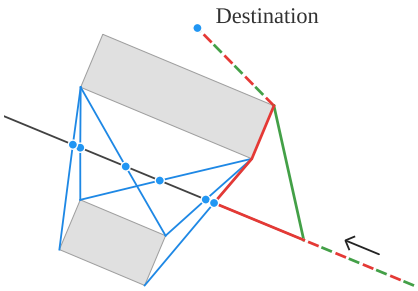


Figure 39: Connectivity problem in case of too few visibility edges. The green path might be the expected result, but the red path is the determined shortest path.

Another problem arises with "islands" inside of obstacles, for example, an actual island inside a lake that is reachable via an edge representing a pedestrian bridge. Even though the hybrid routing algorithm creates edges inside the island, it might also create edges inside the lake. Handling inner rings of (multi-)polygons is therefore needed to correctly navigate within islands of obstacles.

A possible performance optimization of the graph generation process is filtering vertices by convex hulls as introduced in Section 5.3.2. However, this optimization only works for polygons with no hidden vertices, i.e. vertices that are theoretically reachable but inside the obstacle and not visible to the outside. Figure 40 illustrates the problem with the red marked area, which is not reachable when only vertices of the convex hull are connected. The red edges are necessary to create a shortest path from and to the area within the obstacle.

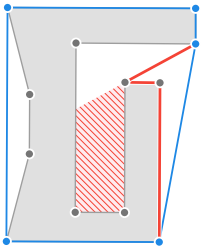


Figure 40: Concave polygons do not always contain all necessary edges (red) but only those edges connecting the convex hull (blue). Some regions are therefore unreachable (red marked area).

A different type of problem arises with road vertices containing attributes that make them a punctual barrier. In OSM, this is for example the case for a node being part of a road and containing the tag `barrier=gate`. In reality, however, such gates and entrances are not only punctual features but often belong to a wall or fence. These linestring barriers are not relevant for purely graph-based routing and therefore do not always exist in OpenStreetMap, but they are very important for the hybrid routing algorithm to ensure correct routing results. Without these barriers, the routing will avoid the road, due to the barrier-attribute, but instead might use a visibility edge next to the road. Adding the missing linestring barriers to the underlying dataset is the ideal solution but algorithmic approaches might also exist to infer these linestrings automatically.

7.1.2 Technical and performance enhancements

Future work might also consider technical enhancements, improving the implementation and eventually improving routing results and performance.

The `HybridVisibilityGraph` class uses a k-d tree as a spatial index for the nodes in the graph, which allows fast queries to find existing nodes at given coordinates. This operation is used multiple times when answering routing queries to add locations to the graph. Even though the data structure is suitable for this task, there are some technical reasons why this can be enhanced. First, the used

k-d tree implementation of MARS internally uses an array-based stack, which is resized multiple times when answering queries to the k-d tree and therefore significantly reduces its performance. Second, the NTS also offers a generic k-d tree implementation that only works on classes and only allows unique values, i.e. one node per coordinate. Unfortunately, both criteria are not fulfilled since each node is a `struct` and multiple nodes might exist per coordinate. Working around these problems, enhancing an existing implementation or choosing a different index structure will likely improve the performance of routing queries.

In the current implementation, visibility neighbors are determined for each input vertex, then sorted into bins based on the obstacle neighbors of the vertex and finally output vertices are created per bin. This approach of determining visibility neighbors first and then splitting the vertices introduces a certain complexity to the code. Splitting vertices based on their obstacle neighbors *before* determining the visibility neighbors makes handling of valid angle areas easier and sorting the visibility neighbors into bins will no longer be necessary. Not only the code complexity but the overall performance might benefit from such a refactoring as well.

Next to performance enhancements in terms of the required time, memory usage can also be improved. As illustrated in Section 6.2.3, merging the road graph led to the most significant increase in memory usage. Further analysis needs to take place to find the steps responsible for this increase in memory. One possible enhancement is the combination of data structures and indices in the `HybridVisibilityGraph`.

The high number of nodes and edges is likely the cause of the significant memory usage. Approaches exist to reduce the number of edges in a routing graph with little effect on the route quality [51], which will speed up graph generation and routing while reduce the memory consumption.

7.1.3 Additional functionalities

Possible future work on the hybrid visibility approach might not only improve the performance and solve existing issue but might also introduce new useful functionalities.

One possible feature is the consideration of road tags. On the one hand, this includes the usage attributes in the weighting function, such as legal restrictions for pedestrian traffic. On the other hand, road attributes can be used to find unrealistic visibility edges. For example, a visibility edge crossing an eight-lane primary road can be removed since real pedestrians, at least under normal circumstances, will probably not cross such roads. Knowing the size of the road can also help to identify visibility edges lying within the area of that road, which means these edges are not realistically usable by pedestrians in the real world. Road attributes can therefore enhance the quality of the routes.

At least in OpenStreetMap, the third dimension, meaning the vertical orientation of features, is specified by specific attributes. This 3D data is currently not supported but would enhance routing in densely built-up urban scenarios with bridges, tunnels or multi-story datasets.

Next to evaluating existing attributes, added attributes on visibility edges can be used by the underlying graph-based algorithm for more accurate routing and also for additional path information for potential end-users. For example, visibility edges traversing grass areas can be augmented with an attribute such as `surface=grass`. Having attributes on visibility edges enables the routing algorithm to prefer or avoid edges, e.g. by their surface conditions.

The current implementation works solely in the main memory, which means the import has to be performed for every execution of a simulation. Storing the hybrid visibility graph to disk detaches the graph generation from its use.

When using the hybrid visibility graph, routing requests might change the underlying graph by adding new nodes and edges. Even though this only adds new possibilities to the routing, parallel routing requests are currently not explicitly supported and might cause problems and unexpected routing results. Allowing parallel routing requests helps autonomous agents to determine optimal paths independently. Possible strategies to introduce parallelism need to be determined.

The presented hybrid visibility graph is static, meaning the data does not change. However, it is already possible to dynamically and temporarily add and connect nodes to the graph. Extending this into the possibility of dynamically changing the existing data of the graph enables the hybrid visibility graph to be used in numerous additional scenarios in which the environment dynamically changes. This is particularly interesting for agent-based simulations, e.g. in a flooding scenario where the flooded area changed over time, but also for real-world applications taking for example real-time traffic data and construction sites into account.

7.2 Conclusion

The presented hybrid routing algorithm enables the determination of shortest paths between arbitrary locations by using a combination of an existing road network with visibility edges to cross open areas. An evaluation of the implementation showed expected results for the following three main aspects regarding performance and route quality.

First, the overall performance shows the expected inherent quadratic time complexity due to the visibility graph creation. Effective optimizations were able to reduce the required processing time significantly. Second, the route quality does benefit from edges covering open spaces and the routes did become shorter and more realistic considering expected routes real pedestrians would likely choose. Agent-based models simulating human behavior can benefit from this increased realism of the routes. In contrast to the hybrid routing algorithm, purely graph-based routes showed a lower quality with longer routes and sometimes large detours, especially in rural areas with larger open spaces and a less dense road network. And third, the data quality significantly impacts the route quality. Missing or wrong data leads to inaccurate or routes that are unusable in the real world.

The presented algorithm has the potential for enhancements, which might be part of future work. However, I am convinced that use cases in agent-based simulations and real-world applications exist in which the hybrid routing algorithm is beneficial for the quality of the determined routes.

Bibliography

[1] Anita Graser. "Integrating open spaces into OpenStreetMap routing graphs for realistic crossing behaviour in pedestrian navigation". In: *GI_Forum* 4.1 (2016), pp. 217–230 (cit. on pp. 2, 13, 17).

[2] *ISO 19109:2015 Geographic information – 4.13 geographic data*. Tech. rep. 2015. URL: https://www.iso.org/obp/ui/#iso:std:iso:19109:ed-2:v1:en (visited on Dec. 31, 2022) (cit. on p. 3).

[3] *ISO 19108:2002 Geographic information — Temporal schema*. Tech. rep. 2015. URL: https://www.iso.org/obp/ui/#iso:std:iso:19108:ed-1:v1:en (visited on May 25, 2023) (cit. on p. 3).

[4] OGC. *OpenGIS® Implementation Standard for Geographic information - Simple feature access - Part 1: Common architecture*. Tech. rep. May 28, 2011. URL: https://portal.ogc.org/files/?artifact_id=25355 (visited on Jan. 29, 2023) (cit. on pp. 3, 5).

[5] Dinesh P Mehta and Sartaj Sahni. *Handbook of data structures and applications*. Chapman and Hall/CRC, 2004 (cit. on p. 4).

[6] *OGC GeoTIFF Standard*. Tech. rep. Sept. 14, 2019. URL: http://docs.opengeospatial.org/is/19-008r4/19-008r4.html (visited on Dec. 31, 2022) (cit. on p. 5).

[7] Esri, ed. *ESRI Shapefile Technical Description*. June 1998. URL: https://www.esri.com/content/dam/esrisites/sitecore-archive/Files/Pdfs/library/whitepapers/pdfs/shapefile.pdf (visited on Jan. 13, 2023) (cit. on p. 5).

[8] OSM Wiki. *About OpenStreetMap*. Apr. 13, 2022. URL: https://wiki.openstreetmap.org/w/index.php?title=About_OpenStreetMap&oldid=2310396 (visited on Jan. 3, 2023) (cit. on p. 6).

[9] OSM Wiki. *Beginners Guide - OSM data explained*. URL: https://wiki.openstreetmap.org/w/index.php?title=Beginners_Guide_1.3&oldid=2528039 (visited on Jan. 3, 2023) (cit. on p. 6).

[10] OSM Wiki. *Proposal process*. URL: https://wiki.openstreetmap.org/w/index.php?title=Proposal_process&oldid=2546273 (visited on Jan. 13, 2023) (cit. on p. 7).

[11] OSM Wiki. *Changeset*. URL: https://wiki.openstreetmap.org/w/index.php?title=Changeset&oldid=2562567 (visited on Jan. 3, 2023) (cit. on p. 7).

[12] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2009 (cit. on pp. 9, 10, 38).

[13] Hannah Bast et al. "Route Planning in Transportation Networks". In: *Algorithm engineering: Selected results and surveys* (2016), pp. 19–80 (cit. on pp. 9, 12, 19).

[14] Jeff Allen. “Using Network Segments in the Visualization of Urban Isochrones”. In: *Cartographica: The International Journal for Geographic Information and Geovisualization* 53.4 (2018), pp. 262–270 (cit. on p. 10).

[15] GraphHopper. *Map Data and Routing Profiles*. URL: https://docs.graphhopper.com/#section/Map-Data-and-Routing-Profiles/OpenStreetMap (visited on Jan. 17, 2023) (cit. on p. 10).

[16] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”. In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. DOI: 10.1109/TSSC.1968.300136 (cit. on pp. 11, 19).

[17] Stuart J Russell and Peter Norvig. *Artificial intelligence a modern approach*. Pearson Education, Inc., 2010 (cit. on p. 11).

[18] Robert Geisberger. “Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks”. Diploma thesis. Institut für Theoretische Informatik Universität Karlsruhe (TH), July 1, 2008 (cit. on pp. 11, 12).

[19] Andrew V Goldberg and Chris Harrelson. “Computing the Shortest Path: A* Search Meets Graph Theory”. In: *SODA*. Vol. 5. 2005, pp. 156–165 (cit. on p. 12).

[20] Adi Botea and Daniel Harabor. “Path Planning with Compressed All-Pairs Shortest Paths Data”. In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 23. 2013, pp. 293–297 (cit. on pp. 12, 13).

[21] Joseph SB Mitchell, David M Mount, and Christos H Papadimitriou. “The discrete geodesic problem”. In: *SIAM Journal on Computing* 16.4 (1987), pp. 647–668 (cit. on p. 13).

[22] Charles Macal and Michael North. “Introductory tutorial: Agent-based modeling and simulation”. In: *Proceedings of the winter simulation conference 2014*. IEEE. 2014, pp. 6–20 (cit. on p. 14).

[23] Angelika Kneidl, André Borrmann, and Dirk Hartmann. “Generation and use of sparse navigation graphs for microscopic pedestrian simulation models”. In: *Advanced Engineering Informatics* 26.4 (2012), pp. 669–680 (cit. on pp. 14, 16).

[24] Christian Gloor, Pascal Stucki, and Kai Nagel. “Hybrid techniques for pedestrian simulations”. In: *International Conference on Cellular Automata*. Springer. 2004, pp. 581–590 (cit. on pp. 14, 20).

[25] Kardi Teknomo and Alexandra Millonig. “A navigation algorithm for pedestrian simulation in dynamic environments”. In: *Proceedings 11th World Conference on Transport Research. Berkeley, California*. 2007 (cit. on pp. 14, 19, 23).

[26] Emo Welzl. “Constructing the visibility graph for n-line segments in O (n2) time”. In: *Information Processing Letters* 20.4 (1985), pp. 167–171 (cit. on pp. 15, 26).

[27] Mark H Overmars and Emo Welzl. “New methods for computing visibility graphs”. In: *Proceedings of the fourth annual symposium on Computational geometry*. 1988, pp. 164–171 (cit. on pp. 15, 26).

[28] Subir Kumar Ghosh and David M Mount. “An output-sensitive algorithm for computing visibility graphs”. In: *SIAM Journal on Computing* 20.5 (1991), pp. 888–910 (cit. on pp. 15, 26).

[29] Sanjiv Kapoor and SN Maheshwari. “Efficient algorithms for Euclidean shortest path and visibility problems with polygonal obstacles”. In: *Proceedings of the fourth annual symposium on computational geometry*. 1988, pp. 172–182 (cit. on pp. 15, 26).

[30] Vasyl Tereshchenko and Yaroslav Tereshchenko. “Triangulating a region between arbitrary polygons”. In: *International Journal of Computing* 16.3 (2017), pp. 160–165 (cit. on p. 16).

[31] J Krisp, Lu Liu, and Thomas Berger. “Goal directed visibility polygon routing for pedestrian navigation”. In: *International Symposium on LBS & TeleCartography*. 2010 (cit. on p. 16).

[32] Stefan Funke, Robin Schirrmeister, and Sabine Storandt. “Automatic Extrapolation of Missing Road Network Data in OpenStreetMap”. In: *Proceedings of the 2nd International Conference on Mining Urban Data-Volume 1392*. 2015, pp. 27–35 (cit. on p. 16).

[33] Simeon Andreev et al. “Towards Realistic Pedestrian Route Planning”. In: *15th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2015 (cit. on p. 16).

[34] Liu Liu and Sisi Zlatanova. “An Approach for Indoor Path Computation among Obstacles that Considers User Dimension”. In: *ISPRS International Journal of Geo-Information* 4.4 (2015), pp. 2821–2841 (cit. on p. 16).

[35] Man Xu, Shuangfeng Wei, and Sisi Zlatanova. “An indoor navigation approach considering obstacles and space subdivision of 2D plan”. In: *Int. Arch. Photogramm. Remote Sens. Spat. Inf. Sci* 41 (2016), p. 339 (cit. on p. 17).

[36] V Walter, M Kada, and H Chen. “Shortest path analyses in raster maps for pedestrian navigation in location based systems”. In: *International Symposium on “Geospatial Databases for Sustainable Development”, Goa, India, ISPRS Technical Commission IV (on CDROM)*. 2006 (cit. on p. 17).

[37] Birgit Elias. “Pedestrian Navigation – Creating a tailored geodatabase for routing”. In: *2007 4th Workshop on Positioning, Navigation and Communication*. IEEE. 2007, pp. 41–47 (cit. on p. 17).

[38] Piyawan Kasemsuppakorn and Hassan A Karimi. “A pedestrian network construction algorithm based on multiple GPS traces”. In: *Transportation research part C: emerging technologies* 26 (2013), pp. 285–300 (cit. on p. 17).

[39] Baoding Zhou et al. “A Pedestrian Network Construction System Based on Crowdsourced Walking Trajectories”. In: *IEEE Internet of Things Journal* 8.9 (2020), pp. 7203–7213 (cit. on p. 17).

[40] Joseph SB Mitchell. “Shortest paths among obstacles in the plane”. In: *Proceedings of the ninth annual symposium on Computational geometry*. 1993, pp. 308–317 (cit. on p. 18).

[41] John Hershberger and Subhash Suri. “An optimal algorithm for Euclidean shortest paths in the plane”. In: *SIAM Journal on Computing* 28.6 (1999), pp. 2215–2256 (cit. on pp. 18, 23, 27).

[42] Bojie Shen et al. “Euclidean pathfinding with compressed path databases”. In: *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*. 2021, pp. 4229–4235 (cit. on p. 18).

[43] Shunhao Oh and Hon Wai Leong. “Edge n-level sparse visibility graphs: Fast optimal any-angle pathfinding using hierarchical taut paths”. In: *Proceedings of the International Symposium on Combinatorial Search*. Vol. 8. 1. 2017, pp. 64–72 (cit. on p. 18).

[44] Reinhard Bauer et al. “Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra’s Algorithm”. In: *Journal of Experimental Algorithmics (JEA)* 15 (2010), pp. 2–1 (cit. on p. 19).

[45] Dirk Hartmann. “Adaptive pedestrian dynamics based on geodesics”. In: *New Journal of Physics* 12.4 (2010), p. 043032 (cit. on p. 19).

[46] Peter M Kielar et al. “A Unified Pedestrian Routing Model for Graph-Based Wayfinding Built on Cognitive Principles”. In: *Transportmetrica A: transport science* 14.5-6 (2018), pp. 406–432 (cit. on p. 20).

[47] Gabriel Y Handler and Israel Zang. “A Dual Algorithm for the Constrained Shortest Path Problem”. In: *Networks* 10.4 (1980), pp. 293–309 (cit. on p. 24).

[48] Pengxiang Zhao et al. “Statistical analysis on the evolution of OpenStreetMap road networks in Beijing”. In: *Physica A: Statistical Mechanics and its Applications* 420 (2015), pp. 59–72 (cit. on p. 25).

[49] Geoff Boeing. “OSMnx: New Methods for Acquiring, Constructing, Analyzing, and Visualizing Complex Street Networks”. In: *Computers, Environment and Urban Systems* 65 (2017), pp. 126–139 (cit. on p. 25).

[50] Mark De Berg et al. *Computational Geometry: Introduction*. Springer, 1997 (cit. on pp. 26, 36).

[51] Quirin Aumann and Peter M Kielar. “A modular routing graph generation method for pedestrian simulation”. In: *28. Forum Bauinformatik*. 2016, pp. 241–253 (cit. on p. 65).

Index

List of Figures

List of Tables

Appendix

In the following code example are given, showing the entry points of the implementation as well as the usage of the hybrid visibility graph.

Implementation of the graph generation and routing

Graph generation

In this section, insights on the implementation of the graph generation from the `HybridVisibilityGraphGenerator` class are given, covering mainly the methods evaluated in Chapter 6. Next to the first and mandatory feature parameter, the $k = (k_b, k_n)$ for the kNN search can be specified with `visibilityNeighborBinCount` (k_b) and `visibilityNeighborsPerBin` (k_n). The remaining arguments are regex-like filtering expressions used to get the respective features from the input data.

```
public static HybridVisibilityGraph Generate(
    IEnumerable<IFeature> features,
    int visibilityNeighborBinCount = 36,
    int visibilityNeighborsPerBin = 10,
    string[]? obstacleExpressions = null,
    string[]? poiExpressions = null,
    string[]? roadExpressions = null)
{
    features = features.ToList(); // Prevent multiple enumerations

    // Getting, unwrapping and triangulatin gobstacles
    var obstacles = GetObstacles(features, obstacleExpressions);

    // Performing the kNN search
    var vertexNeighbors = VisibilityGraphGenerator.CalculateVisibleKnn(
        obstacles, visibilityNeighborBinCount, visibilityNeighborsPerBin
    );

    // Generate the hybrid visibility graph, which currently only contains
    // visibility edges
    var graph = CreateVisibilityGraph(
        vertexNeighbors, obstacles
    );

    // Merge the road edges into the visibility graph
    MergeRoadsIntoGraph(features, graph, roadExpressions);

    // Add attributes to POI nodes in the graph
    AddAttributesToPoiNodes(features, graph, poiExpressions);

    return graph;
}
```

Routing

The following code example is a slightly simplified version of the actual routing method that can be used by agents. An example for the `heuristic` can be seen in the usage examples below.

```
public List<Position> OptimalPath(Position source, Position destination,
    Func<EdgeData, NodeData, double> heuristic)
{
    var sourceNode = AddPositionToGraph(source);
    var destNode = AddPositionToGraph(destination);

    // Connect the nodes to the graph, which creates new nodes and edges
    var (sourceNodes, sourceEdges) = ConnectNodeToGraph(sourceNode, false);
    var (destNodes, destEdges) = ConnectNodeToGraph(destinationNode, false);

    var result = Graph.AStarAlgorithm(sourceNode.Key, destNode.Key, heuristic);

    // Remove temporarily created nodes (which automatically removes the edges too)
    sourceEdges.Each(RemoveEdge);
    destEdges.Each(RemoveEdge);
    sourceNodes.Each(RemoveNode);
    destNodes.Each(RemoveNode);

    // Merge all coordinates of the edges into the final list
    return result
        .Aggregate(new List<Position>(), (list, edge) => { ... })
        .ToList();
}
```

Usage of the hybrid visibility graph

Graph generation

The graph generation itself just requires one method call accepting a list of `VectorStructuredData` objects representing all features (obstacles and roads) of the input data. This code can, for example, be used in a shared class such as a layer in MARS or any other stateful class storing the `HybridVisibilityGraph` instance. In the following example, the implementation of an `InitLayer` method of a `VectorLayer` is shown:

```
public override bool InitLayer(LayerInitData layerInitData, RegisterAgent
    registerHandle = null, UnregisterAgent unregisterAgent = null)
{
    var initLayer = base.InitLayer(layerInitData, registerHandle, unregisterAgent);

    // Get the VectorStructuredData from the input features
    var features = Features.Map(f => f.VectorStructured).ToList();

    // Generate the hybrid visibility graph
    this.HybridVisibilityGraph = HybridVisibilityGraphGenerator.Generate(features);

    return initLayer;
}
```

Routing

The entry point for routing requests is within the hybrid visibility graph class and therefore can be invoked by a single method call. Three methods are available and shown in the following example. The predefined weighted shortest path uses a function with a factor of 0.8 on any road edge. Using a custom weighting function is also possible, in this example a relatively strong preference for bridleways is defined.

```
List<Position> route;

// Using existing methods:
route = this.HybridVisibilityGraph.ShortestPath(source, dest);
route = this.HybridVisibilityGraph.WeightedShortestPath(source, dest);

// Using a custom weighting function:
var weightingFunction =(edge, _) =>
{
    edge.Data.TryGetValue("highway", out object? highwayAttribute);
    var factor = "bridleway".Equals(highwayAttribute)
        ? 0.5
        : 1;
    return edge.Length * factor;
};

route = this.HybridVisibilityGraph.WeightedShortestPath(source, dest);
```

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Masterstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der elektronischen Abgabe entspricht.

Hamburg, den 31.07.2023 Hauke Stieler

Veröffentlichung

Ich stimme der Einstellung der Arbeit in die Bibliothek des Fachbereichs Informatik zu.

Hamburg, den 31.07.2023 Hauke Stieler