Algorithmic approaches for optimizing linear infrastructure planning

Master Thesis

Nina Wiedemann

Supervisor:

Prof. Rico Zenklusen

Dr. David Adjiashvili

Dr. Stefano Grassi

Institute for Operations Research
Department of Mathematics

ETH Zürich

November 4, 2020

Contents

Abstract

The ubiquitous expansion and transformation of the energy supply system involves large-scale power infrastructure construction projects. With more than a million dollars of investment per kilometre, planning authorities aim to minimise the resistances posed by multiple stakeholders, including for example environment protection and distance control. Mathematical optimisation research offers efficient algorithms to compute globally optimal solutions for least-cost transmission line routing based on geographic input data.
We propose a framework, *PowerPlanner*, that utilizes a graph model where vertices represent possible locations of power transmission towers and edges are placed according to the feasible distance between neighbouring towers. Our experiments show that compared to previous work our approach reduces the resistances by more than 10% in feasible time, while at the same time offering much more flexibility and functionality.
As part of the computational framework, we developed and analyzed new algorithms to cope with the challenges arising in linear infrastructure layout: First, we introduce a variant of the Bellman-Ford algorithm that efficiently computes the *minimum-angle* shortest path. Furthermore, we discuss and analyse methods to output several diverse path alternatives, and last provide a method to optimise the pylon height in complex terrain. Apart from some problems that are shown to be NP-hard, we prove efficient runtimes for the implemented approaches.
Finally, our methods are demonstrated in a simple and intuitive graphical user interface. All code is available at https://github.com/NinaWie/PowerPlanner.

Acknowledgements

I would like to express my very great appreciation to David and Stefano for their supervision throughout the project. They both spent considerable time on our sometimes lengthy meetings, where I received highly useful feedback from the theoretical and practical point of view. Special thanks to David for his valuable help in my career choices, and also to Prof. Zenklusen for his advice in this regard and his support of the project.
In addition I would like to thank the company Gilytics in general for their involvement in this project, which enabled me to work on a topic that is directly relevant to industry. I am looking forward to our further collaboration.
I wish to thank my parents for their support and encouragement throughout my study, and my friends for their sympathetic ear. Finally, I am deeply grateful for my relationship with Jannis whose enthusiasm and support for my work is a great source of energy, and who makes every break worthwhile.

1 Introduction

The layout of power transmission lines has always been a contentious topic involving many stakeholders. In the context of climate change it has become even more pressing, since huge infrastructure is required to serve the energy demands reliably. With planning authorities aiming at minimal costs, environmentalists criticizing the impact on conservation areas and bird mortality and local residents opposing huge visible infrastructure, the planning of power lines has fittingly been called a "field of tension"[13].

Power grid planning has come to the fore in recent years due to necessary changes in energy supply in times of climate change. Since renewable energies are more location-bound than conventional energies, their expansion requires extensive adaptation and enlargement of power grids, as analyzed in [67], [7] (focused on India) and [70] (in Greece). On the other hand, climate change also directly affects power infrastructure with the increase in frequency and severity of extreme weather events [36]. Panteli and Mancarella [76] argue that extreme weather strongly impacts power infrastructure, as it can be accounted for 80% of large-scale power outages from 2003 to 2012 [59]. It is therefore widely recommended to modernize power infrastructure with respect to this aspect [22, 101]. The European Union has reacted to the new demands with the introduction of a Ten-Year Network Development Plan (TYNDP), which is updated every two years and identifies necessary investments in all EU-transmission networks. The TYNPDF 2014 for example lists required expansion projects of around 50,000 km overall length, where 80% directly or indirectly target the integration of renewable energies [35, 66]. With an estimated cost of 1.5 million euros per km [97], this leads to investments of approximately 75 billion euros just for EU-infrastructure within the next ten years.

Considering this huge scale of ongoign and future power infrastructure projects, it is more important than ever to optimize the planning process. While in practice the planning process is still often highly manual, possible approaches towards the optimal layout of power lines have for a long time been discussed in the literature. Early work include research by Popp et al. [78], Ranyard and Wren [79] and Mitra and Wolfenden [71] that regard pylon height optimization and construction constraints, whereas later the focus shifted to route selection with respect to environmental aspects [104] and other geographic features. Now, there exists extensive analysis covering all aspects of transmission line planning [60, 57], ranging from design to material selection and construction. Most of these works regard the planning process as a communicative process, a discussion of stakeholders and planners, or in other words as a typical multi-criteria decision analysis (MCDA) [68] with geographic information system (GIS) data. Research on transmission line planning thus focused on simplifying and accelerating the discussion between experts. For example, Bevanger et al. [13] and [12] suggest the *Delphi process* [99] as a MCDA method for criteria selection, where experts rank criteria in several workshops. Grossardt et al. [44] take a more mathematical approach with the proposal of Analytic Minimum Impedance Surface, where criteria are weighted to form a map of summarized resistances, but as in [72] the exact route is finally determined in a compromise-seeking discussion process.

However, in the end the optimal placement of power lines is a combinatorial problem, when provided with appropriate GIS data. While MCDA always constitutes at least the first step in the planning process - a step necessary to define the input data in the form of weighted criteria - the use of appropriate software and optimization algorithms is at least as important as the MCDA process, in particular during the current efforts to expand the power grid in the most cost- and time-efficient way. Besides software such as the one provided by ArcGIS [34], there has been extensive literature on finding the least cost path in the context of (power) infrastructure planning. In the large majority of cases, so-called raster-based approaches are employed, where

the area is rasterized and each cell is assigned a cost or resistance according to the considered criteria. Bagli et al. [9] evaluate the output of GIS-software for least-cost paths in an iterative multi-criteria analysis. Bevanger et al. [13] worked on environment-friendly power-line routing with the Norwegian Institute for Nature Research (NINA), developing a least-cost path (LCP) toolbox published later by Hanssen et al. [46, 47]. Similarly, Monteiro et al. [73] and de Lima et al. [23] describe algorithms that partly even consider the slope of the terrain and costs on direction change [73].

Despite these advances on optimal power infrastructure layout, all approaches listed above suffer from a common drawback, namely that they only compute a shortest path through a regular grid where each cell is connected to its 8-neighborhood. Pylons and cables are not differentiated in the optimization process, but instead the transmission line is viewed as a homogeneous path through a resistance raster. This approach might be appropriate for infrastructure such as pipes where all crossed locations are equally affected, but for power lines the cost of placing a pylon can differ significantly from the cost of traversing a cell with a cable. It might even be impossible to place a pylon in one location, while it is not problematic if the same spot is between two pylons and traversed by the cable. Thus, a globally optimal solution with minimal costs can only be achieved with an approach that optimizes the placement of pylons directly, instead of finding a good path and spotting pylons only as a second step. Shandiz et al. [89] compare route-generation methods including raster and vector based systems, and conclude that conventional LCP methods are not necessarily optimal, while non-corridor methods tend to find the least-resistance alternative in most cases.

On top of that, an additional challenge in power infrastructure planning is the avoidance of angles on the route, since they induce additional costs and electrical resistance. Even if angles are actually taken into account in a LCP computation through a grid, they do not directly resemble the final angles between pylons. More often, angles are only reduced by applying a straightening of the route *after* the optimization, clearly leading to suboptimal solutions.
One of the rare proposals of graph-based and angle-constrained shortest path computation or at least refinement is given by Santos et al. [87], who first compute a wide corridor, but then define possible pylons as vertices and cables as edges. Building up on work of Piveteau et al. [77], they construct the *line graph* in this corridor to account for costs on the angle between edges. This graph representation and the integration of angle costs are important steps towards optimized pylon spotting, but leave room for improvement with respect to computational efficiency.

Contribution. Here, we aim to fill this gap with an efficient framework for the automatic and optimal placement of pylons for power transmission lines. Given geographic data as input, the setting is modeled as a graph, where each raster cell is represented by a vertex, and two vertices are connected if and only if their potential pylons could be connected by a cable. Based on this model, various shortest path algorithms are implemented, compared and adapted, and novel methods are developed to cope with the challenges specific to linear infrastructure layout. In detail, the framework offers the following functions:

- Computation of the least cost path from source to destination with given (possibly different) cable and pylon resistance values[1] and cost category weights.
- Support for penalizing angles (or possible other costs on two adjacent edges) efficiently, omitting the need for larger graph representations.

[1]The term "resistance" is used interchangeably with "cost" here and thus does not refer to electrical resistance. Instead, it denotes the summarized resistance of several geographic layers.

- Implementation of several algorithms to compute a set of k paths that are short but also *diverse* among each other according to a suitable metric. This is beneficial for planners to compare and analyse alternatives in different regions.
- Possibility to consider constraints caused by the sag of the cable in complex terrain.
- Visualization of pareto-optimal paths with varying cost weights, allowing insights into the sensitivity to the input parameters.

The *PowerPlanner* framework was developed in collaboration with Gilytics, a software company providing optimization and visualization software for linear infrastructure planning. With the data provided by Gilytics we were able evaluate our framework on realistic power infrastructure projects. In addition to the computational framework, the runtime and optimality guarantees are discussed for the proposed algorithms. Most importantly, it is proven that a shortest path with angle costs can be computed efficiently: In contrast to the computation in a line graph as in [87, 77] that utilizes memory of md edges (for a d-regular graph with m edges) and thus performs $\mathcal{O}(mdn)$ operations in the Bellman-Ford shortest path algorithm, our algorithm only requires as much space as the normal graph ($\mathcal{O}(m)$). With a linear angle cost function the runtime also decreases to $\mathcal{O}(mn \cdot \log d^2)$.

Here, we present the framework in detail in the following steps: First, we discuss related work and introduce the problem statement, setting and notation. Second, the system with its component and engineering challenges is presented in section 3, and an overview of methods is given. The implementation is evaluated in experiments on three diverse project instances in subsection 3.3. Section 4 provides an in-depth description of the novel algorithms for minimal-angle shortest paths as well as a theoretical analysis of the efficiency of these algorithms, while section 5 covers implementations and analysis of approaches towards computing k diverse shortest paths. Further theoretical results in the context of power infrastructure planning are given in section 6, before presenting a proof-of-concept user interface in section 7. Last, the contributions and limitations of the proposed framework are discussed.

2 Problem statement

2.1 Background

Power infrastructure layout Although the planning process for power infrastructure is often highly manual even nowadays, attempts on the automation of transmission line planning date back to the start of using computers in research. In 1963, Popp et al. [78] presented very early work on optimal tower placement and even the selection of pylon heights based on the line sag and terrain. Similarly, [71] already proposed in 1968 to use the Bellman Ford shortest path algorithm for transmission line layout. They formulate the task as an optimal selection of tower locations out of a given set of test towers.

Later, the work on power infrastructure planning diverged, partly viewing the problem as a communicative process of experts from geography and engineering, others developing optimization methods. Regarding the former, an example was termed "Analytic Minimum Impedance Surface" [44]: Environmental, engineering and social factors are input by stakeholders, weighted and traded off, yielding a surface of resistances that is displayed and analyzed in GIS software. An application of the method is shown in Gill et al. [41]. With respect to optimization problems on the other hand, research questions range from optimal network layout [39] to so-called structural optimization [96] that optimizes the construction design of transmission towers to minimize its weight and cost. We do not go into detail here, since our focus is on the most prevalent use case in power line planning, namely finding an optimal route from a given start to target location in a restricted region.

Graph-based modeling of infrastructure layout A natural approach to optimal transmission line layout is to apply shortest path algorithms from graph theory. Note that we use the terms "least cost path" and "shortest path" as well as "cost" and "resistance" interchangeably, since a shortest path minimizes the encountered resistances and thereby reduces the involved costs for planning and construction.
Graph models have been used in linear infrastructure planning for a long time, including applications on the layout of pipes [28], canals [21] or highways [108]. A graph modelling is particularly suitable in the case of power infrastructure due to the difference between pylon-costs (resistance to place a pylon) and cable-cost (costs to traverse an area with a cable). In contrast to previous work [73, 102, 105, 63, 112], we aim to explicitly optimize the placement of transmission towers, instead of computing only a line or "corridor". In such approaches, usually the least cost path is computed in a rasterized graph with edges connecting neighboring cells, i.e. every vertex is connected to its 8-neighborhood in the plane (see Figure 1b). Our model on the other hand places edges between distant cells which could possibly be connected by a cable.

Shortest path algorithms In above literature on linear infrastructure layout, standard shortest path algorithms are frequently used, most prominently *Dijkstra's algorithm* [27] or the dynamic program proposed by Shimbel [90], Bellman [11], Ford Jr [37] and later Moore [74], known as the *Bellman-Ford (BF) algorithm*. Although *Dijkstra's algorithm* is more time-efficient in most cases ($\mathbf{O}(m \cdot \log n)$ versus $\mathbf{O}(m \cdot n)$ for a graph with n vertices and m edges), BF can be preferable for large scale infrastructure layout for the following reasons:

- The runtime of BF can be reduced with prior knowledge about the maximum length of the path. The parameter n in the runtime $\mathbf{O}(m \cdot n)$ refers to the maximal length of the shortest path, which is much smaller than n in most applications

- BF requires less memory resources, since Dijkstra's algorithm uses a priority queue while BF simply updates all edges in each iteration
- BF can deal with negative edge weights which might appear in geographic data[2] if not normalized appropriately
- BF can be parallelized well

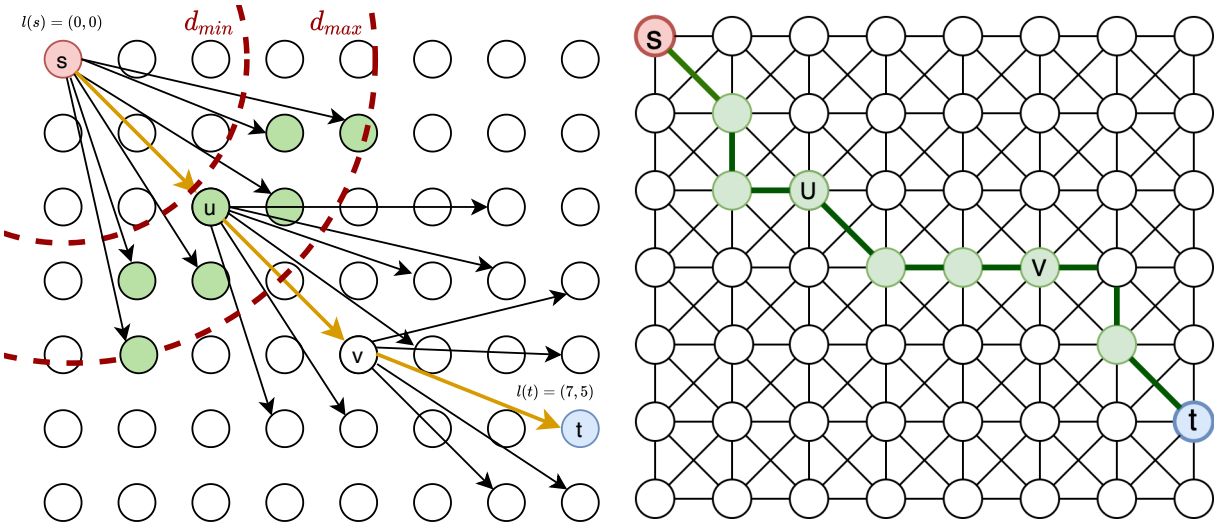
More generally, the shortest path problem can be expressed as a linear program. Let $c(e)$ be the weight on an edge $e = (u, v)$ in a graph $G = (V, E)$. Then the variables $D[u]$ represent the respective distances from the source vertex s to vertex u, where t is the target vertex. The linear program is given by

$$\begin{aligned} &\text{maximize } D[t] \\ &s.t. \\ D[v] \leq D[u] + c(e) \ \forall e = (u, v) \in E \\ D[s] = 0 \end{aligned}$$

Notably, a feasible dual of the linear program corresponds to an upper bound on the length of the path, and thus also to a *consistent* heuristic for the A^* algorithm [50], a path search method that is oftentimes preferred over Dijkstra or BF, particularly in robotics. Linear or integer programming formulations are used to solve or approximate several important variants of the shortest path problem, and is for example mentioned below in the context of the Quadratic Shortest Path Problem (subsection 4.1).

The main challenge in using graph models for power infrastructure layout is that one must allow for possibly enormous graphs, since regions of several hundred kilometers are traversed. Since similar problems appear in applications on road networks, there exists extensive work on the optimization of shortest path queries in large graphs. An overview of early speed-ups of route planning is provided in Delling et al. [24]. Later, Geisberger et al. [40] introduced *Contraction Hierarchies*, a method that iteratively replaces the least important node by shortcuts and retains sparsity of the graph. Others have proposed distance-lookup-based methods that require more space but less query-time, most importantly hub labeling [2, 3], the fastest existing algorithm for road networks. For a summary and discussion of available speed-up techniques, please refer to [10]. However, these methods are not as well-suited for transmission line planning where the source and target vertex are fixed. In such a setting it is not conducive to use a contraction process that only improves the *query time* in an all-pairs-shortest-path setting, instead of a single shortest path computation.

An entirely different formulation is given by [91] who suggest to optimize transmission line planning by finding a least cost *corridor* of certain width. Shirabe [92] proposes to transform the raster grid into a graph where each node represents a neighborhood, and they achieve an efficient representation whose size does not exceed the size of the initial grid. Although planning agencies in the power infrastructure industry might be interested in such techniques since traditionally they often employ two-stage processes where an optimal corridor is found as the first step, we do not go into detail here because our work aims at a globally optimal transmission *tower* placement.



(a) Our graph model

(b) Grid-model often used in power infrastructure planning

Figure 1: Graph models for power infrastructure layout. b) In related works, often simply the path through a grid is computed, and the pylons are spotted as a second step. Vertices are placed in each cell of the raster, and edges are placed between each pair of vertices with distance between d_{min} and d_{max}. Each vertex corresponds to a coordinate location $l(v)$ corresponding to a geographic location. Here, edges are restricted to the ones with sufficiently low angle to the straight line between s and t, thereby yielding a directed acyclic graph (DAG).

2.2 Graph model setup and notation

Graph model. In the PowerPlanner framework, all possible transmission line routes are modelled in a graph $G = (V, E)$ with $n := |V|$ vertices and $m := |E|$ directed or undirected edges. Figure 1a visualizes the graph layout. Each vertex is one cell in the raster, i.e. there exists a *bijective* function $l : V \to \mathbb{N} \times \mathbb{N}$, such that $l(v)$ outputs the set of raster coordinates modelled by vertex v. Thus, by $l^{-1}(x, y)$ we denote the vertex representing the cell at coordinates x, y. In the following, if not noted otherwise, $s \in V$ always denotes the source vertex and t the target. Furthermore, edges are placed between each pair of possible neighboring pylons in a transmission line. Specifically, the neighborhood of a pylon (or vertex) is restricted to a range of distances, d_{min} to d_{max}. For example, in a raster with 10m resolution, realistic values are $d_{min} = 15$ and $d_{max} = 25$, corresponding to a minimum of 150m and a maximum of 250m of cable separating two pylons. Translated to the graph setting, we compute a ring of raster cells as the neighbors of vertex v, such that

$$E = \{(u, v) \mid d_{min} \leq \|l(u) - l(v)\| \leq d_{max}\}.$$

Note that the graph is therefore regular, when ignoring instance boundaries.

While the most general setting is an undirected graph, with simplifying assumptions one can model the problem as a directed acyclic graph (DAG) $G = (V, A)$. The runtime for shortest path algorithms in DAGs is linear and thus significantly shorter, and the necessary assumptions do not seem too restrictive: A maximal angle deviation θ_α can be defined as the maximum angle between the vector $\vec{a}$ representing an arc $a \in A$ and the straight line vector connecting s and t, $\vec{\zeta} = l(t) - l(s)$. Then

$$A = \left\{(u, v) \mid d_{min} \leq \|l(u) - l(v)\| \leq d_{max} \wedge cos\left(\frac{\vec{a} \cdot \vec{\zeta}}{\|\vec{a}\|\|\vec{\zeta}\|}\right) < \theta_\alpha\right\}$$

[2] a negative weighting could indicate that it is beneficial to place a transmission tower in this location, e.g. close to a highway

This definition is based on the assumption that in power infrastructure layout it is very unlikely that a tower is placed "behind" another one in a transmission line from s to t. If $\theta_{max} < \frac{\pi}{2}$, then the graph is a DAG, since the outgoing edges of each vertex are the ones on the semicircle ring directed towards the target (see Figure 1a).

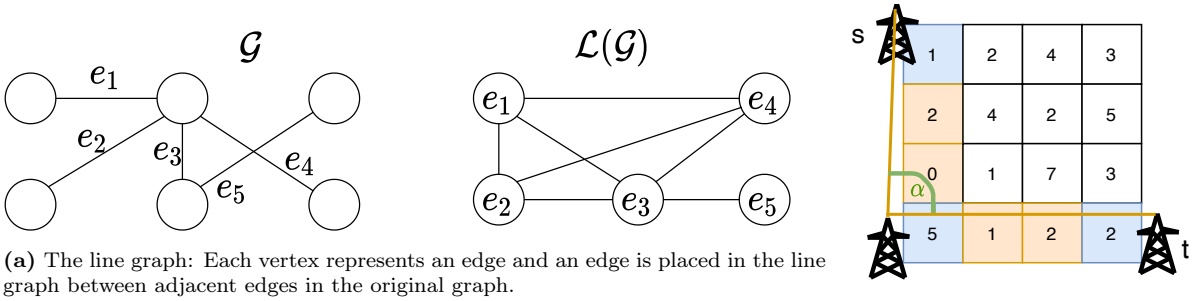
Edge costs. The input data consisting of diverse geographic raster layers is summarized in a "resistance" or "cost surface" C (subsection 3.1). In general, there can be two separate resistance maps: C_p is the pylon resistance, such that $C_p[x, y]$ is the resistance to place a transmission tower at the raster coordinates (x, y). On the other hand, $C_e[x, y]$ is the cost to traverse the raster cell at (x, y) with a transmission cable. In a typical application scenario, the values in C_e are much lower than the ones in C_p. In practice, one might simply approximate the costs as $C_e = C_p$ and only weight the edge costs by $w_e \in [0, 1]$ in the shortest path computation (see below) since higher pylon resistance (e.g. for environment protection) usually imply higher cable-traversal costs.

In the graph model, C_e and C_p are combined in the edge cost. However, in order to compute the cable cost, the location of the cable (a straight line) above a discrete raster of cells must be approximated. For this purpose we utilize the Bresenham line [15], a well-known line drawing algorithm developed for computer graphics, since a line formed of pixels resembles the cells crossed by a power line. The algorithm is comparably efficient, but in the graph model presented here it is only computed once for each feasible neighbor shift (lines from s to the green vertices in Figure 1a), and can then be shifted to all other vertices. The computation of the Bresenham line M_e for an edge $e = (u, v)$ yields a set of coordinates corresponding to the raster cells on the line, formally $M_e = \{(x_1, y_1), (x_2, y_2), \ldots, (x_d, y_d)\}$. M_e does not include the pylon locations, i.e. $l(u), l(v) \notin M_e$. There are many possibilities to combine pylon and cable costs, and we define a parameter w_e in order to offer most flexibility, such that w_e describes the importance of cable costs in contrast to pylon costs. Furthermore, we compute the cable cost of an edge by averaging the cell-wise values of C_e, in order to yield a value in a comparable range as the pylon cost. The pylon costs are summed with the weighted average cable-cell cost to yield a single edge cost $c(e)$, formally

$$c(e) = \frac{C_p[l(u)] + C_p[l(v)]}{2} + w_e \cdot \frac{\sum_{(x,y)\in M_e} C_e[x, y]}{d} \tag{2.1}$$

As visualized in Figure 2b, in general we distinguish three different costs: normal pylon resistances, cable resistances (weighted with w_e) and angle resistances (weighted with w_a).

Line graph. In the context of angle minimization in power infrastructure layout, we will also construct the "line graph" or "dual graph" of the above graph. Each edge in the graph G is



(a) The line graph: Each vertex represents an edge and an edge is placed in the line graph between adjacent edges in the original graph.

(b) Pylon, cable and angle costs

Figure 2

represented as a vertex in the line graph $L(G)$, and an edge in $L(G)$ is placed between to vertices if their corresponding edges are adjacent in G. An example is given in Figure 2a. Formally, the line graph $L(G) = (V_L, E_L)$ transforms the graph $G = (V, E)$ in the following way:

$$V_L = \{u_e \mid e \in E\}$$
$$E_L = \{(u_e, u_f) \mid e, f \in E, \quad e = (-, v), f = (v, -)\},$$

where $e = (v, -)$ denotes an edge from v to any other vertex. For simplicity, we do not distinguish between directed or undirected graphs here, since the line graph is constructed in the same way.

In the following, the notation introduced in this section is used, as long as not explicitly defined otherwise.

3 A computational framework for power infrastructure planning

The proposed *PowerPlanner* framework proceeds in several steps: First, the input data is pre-processed to construct a single resistance raster from a set of geographic layers and corresponding weights. Secondly, a graph is build as described above. Methods are proposed that considerably reduce the size of the graph by employing an iterative region-restriction scheme. Last, a shortest path algorithm is applied to find the globally optimal least-resistance-placement of pylons from a given source to target location. The algorithms, data structures and implementation details will be outlined in this section, and the methods are evaluated in experiments on real infrastructure projects.

3.1 Input data and pre-processing

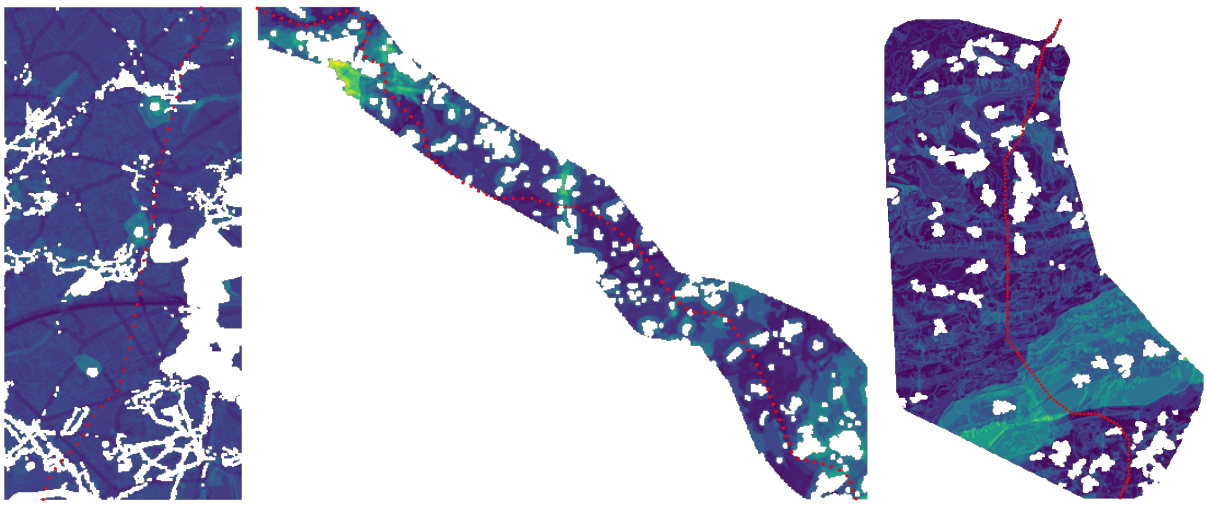
The work presented in this thesis was done in collaboration with Gilytics (https://www.gilytics.com), a software company providing solutions for transportation and energy infrastructure planning. Gilytics provided geographic layers $L_1, \cdots, L_k$ extracted from GIS software which are two-dimensional binary `tif` files. Each layer indicates the occurrence of a geographic feature on a cell, e.g. one layer L_i could depict for each cell whether it is part of a forest ($L_i[x, y] = 1 \iff$ forest exists on cell x, y). In addition, the layers were grouped into cost categories and layer- and class weights w_l, w_c were specified in discussions with the involved stakeholders. The number and type of layers differ between the available instance. For data protection reasons, the project location and exact details can not be unveiled, but Table 1 provides an overview of specifications and Figure 3 shows the summarized resistance raster C and an optimal route to place pylons on the raster. In all further experiments, we refer to these instances by the numbering introduced here. From Table 1 it is apparent that **Instance 1** is a rather small project region leading to

Instance	Raster width	Raster height	Project region (not forbidden)	d_min	d_max	#neighbours	Estimated #edges	Traversing forbidden cells	Complex terrain
1	1300	739	454 392 cells (81%) 45.4 km^2	15	25	632	287 175 744	Allowed	No
2	3078	3724	1 498 585 (17%) 149.8 km^2	35	50	2002	3 000 167 170	Forbidden	No
3	3120	2092	3 574 446 (57%) 357.4 km^2	15	25	632	2 259 049 872	Forbidden	Yes

Table 1: Instance properties, sizes and parameters: All numbers refer to a raster with a size (resolution) of 10 times 10 meters for one cell. Three real-life projects were used for our experiments. It is apparent that both the size of the project regions as well as the number of feasible pylon neighbors determines the graph size.

a graph of only 454,392 vertices, whereas the graphs of **Instance 2 and 3** at high resolution have billions of edges. **Instance 2** traverses a huge region but the project region is restricted to a narrow corridor (Figure 3b). The number of edges is nevertheless enormous because of the higher distance bounds ($d_{min} = 350,\ d_{max} = 500$)[3]. **Instance 3** is the only instance traversing complex terrain of significantly varying altitude. **Instance 1 and 3** have four and three cost categories respectively, whereas no clear categories were specified for **Instance 2**. In short, the high diversity of instances verifies the ability of our framework to cope with different requirements and features.

[3]Each vertex in the graph is connected to its feasible neighboring pylons, which are all the ones in the ring between d_{min} and d_{max}. Since the area of a circle increases quadratically with the radius, also the number of edges is approximately quadrupled when doubling the radius



(a) Instance 1 **(b)** Instance 2 **(c)** Instance3

Figure 3: Resistance rasters C and optimal paths for each of the three instances. The rasters correspond to the weighted sum of geographic input layers; overlayed with the forbidden regions $\mathcal{F}$ (white). Dark areas are of low resistance, green-yellow regions of high resistance. The angle-optimal least cost path is shown in red. The instances are diverse in terms of size, shape and distribution of resistance

.

Data pre-processing. A **DataReader** class takes care of the pre-processing of all geographic data. The input to the graph processing framework is a collection of resistances layers $L_1, \ldots, L_k$, grouped into cost classes and with given layer- and category weights (w_l, w_c). For example, multiple layers could be part of the category "environment protection costs". For simplicity, we only describe the construction of the pylon resistance raster C_p here; if C_e (the cable resistance) is different from C_p, the processing steps are analogous.

Two pre-processing methods were implemented: In the first option, the cost surface is constructed simply by a weighted sum of the layers, multiplied by their category- and resistance weights:

$$C = \sum_{i=1}^{k} w_c(L_i) \cdot w_l(L_i) \cdot L_i$$

The second option is visualized in Figure 4. Layers are first grouped into classes or categories $\mathcal{D} = \{D_1, D_2, \ldots, D_{|\mathcal{D}|}\}$, in order to allow for an analysis and comparison of the respective category costs of the output path. For example $L_i : i \in I_{\text{env}}$ are the layers of environmental cost. The layers of one class are weighted and added up to yield the class costs C_D:

$$\forall D \in \mathcal{D}: \quad C_D = f_{norm}\Big(\sum_{i \in I_D} w_l(L_i) \cdot L_i\Big)$$

A normalization function f_{norm}, here chosen as a min-max scaling, is applied in order to equalize the value ranges of different categories.
Furthermore, the cost classes are again combined in a weighted sum to yield a two dimensional cost instance:

$$C = \sum_{D \in \mathcal{D}} w_c(C_D) \cdot C_D$$

Only **Instance 2** was pre-processed with the first option (simple weighted sum, no category-wise processing); **Instance 1 and 3** are processed as shown in Figure 4 (second option).

On the other hand, some layers do not define a cost, but forbidden areas where no pylons can be placed, such as conservation areas. The intersection of the set of forbidden layers F yields the

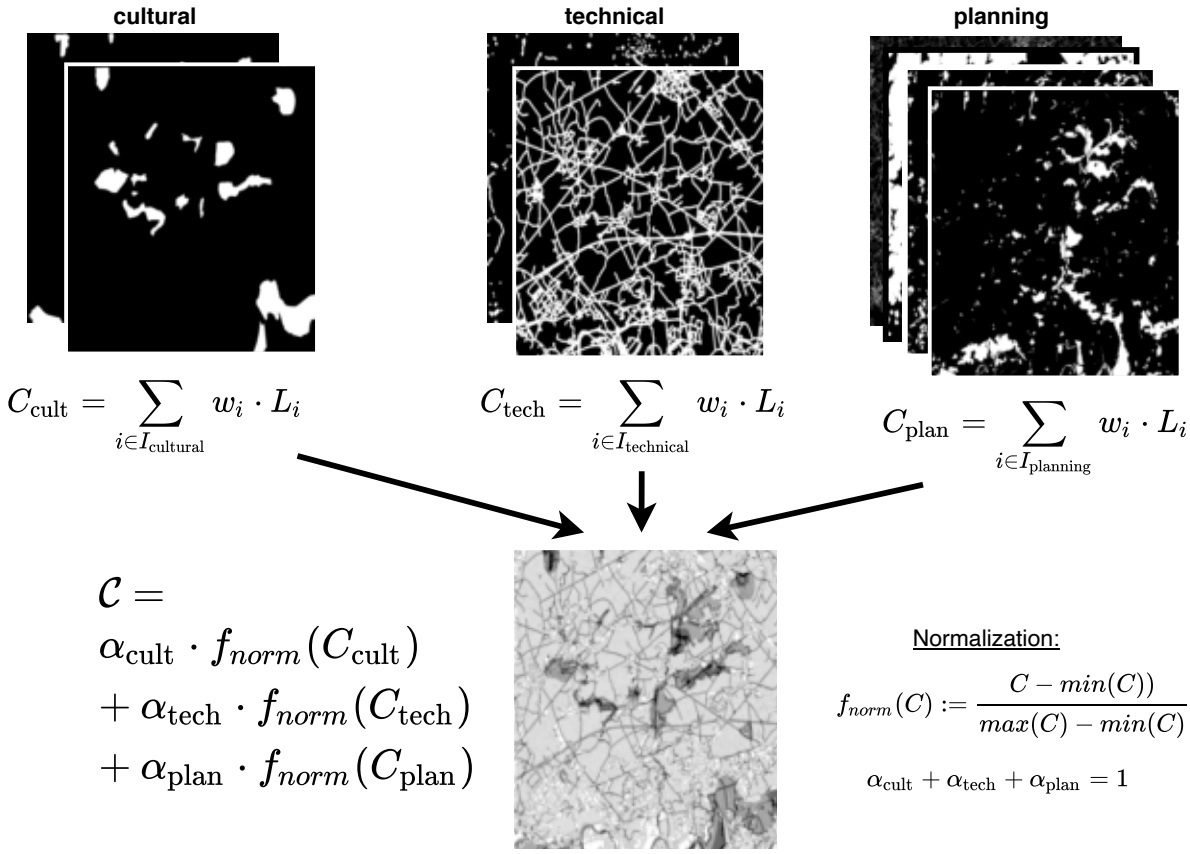


Figure 4: Data pre-processing: raster data extracted from GIS-software is grouped in given categories, and within each category a weighted sum of layers is computed and normalized to values between zero and one. Last, the category resistance maps are again combined in a weighted sum to yield the complete cost raster C.

hard constraint raster

$$\mathcal{F} = \bigcap_{L \in F} L \, .$$

If the forbidden areas can not be traversed with a cable, C and $\mathcal{F}$ can be merged by placing infinity costs in locations in C that are forbidden by $\mathcal{F}$.

3.2 Computational methods and implementations

The computational framework was implemented in `Python 3` and is available on GitHub (https://github.com/NinaWie/PowerPlanner). The code structure is presented as a UML diagram in Figure 5. The left side present the baseline algorithms, including all classes inheriting from `GeneralGraph`, and implementing a normal graph and a line graph representation of the planning problem. The right side on the other hand shows the `ImplicitLG`, our own implementation of the proposed angle-cost shortest path algorithm.

3.2.1 Basic graph models

First, the class **GeneralGraph** functions as an interface to the python package `graph-tool`. The `graph-tool` library was chosen for its performance that is comparable to pure C++, as all core functions are implemented in C++ and it relies on the `Boost Graph Library`. Here,

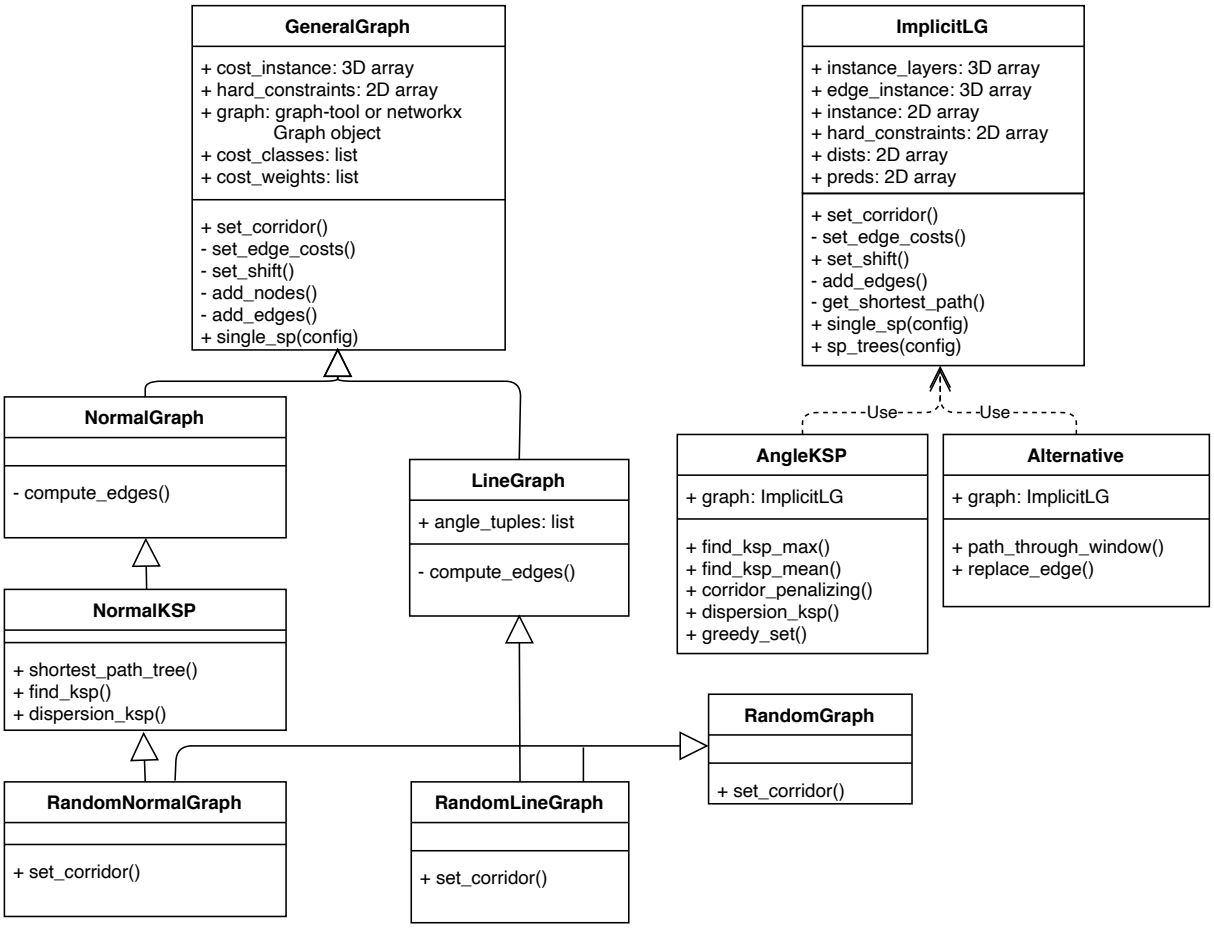


Figure 5: UML class diagram

the `graph-tool` library is only used in **GeneralGraph** to build the graph and to run the Bellman-Ford algorithm described in section 2.1. The computation of required vertices and edges is however specific to the subclasses **NormalGraph** and **LineGraph** which overwrite the method `compute_edges`. The normal graph is built as explained in section 2.2 and edge costs are computed according to Equation 2.1.

In the **LineGraph** class on the other hand, vertices of the line graph represent an edge of the normal graph, and an edge is placed between all adjacent edges (see subsection 2.2). In both cases, the addition of edges is efficiently implemented with `numpy arrays`: The algorithm only loops over the neighbor-shifts, i.e. all vectors (s_x, s_y) such that $d_{min} \leq \|(s_x, s_y)^T\| \leq d_{max}$. In one iteration of the loop, the edges are computed for all vertices and its shifted neighbors simultaneously, i.e. all edges (u, v) with $l(v) = l(u) + (s_x, s_y)$. The function `numpy.roll` is used to shift the two-dimensional cost instance by s_x in x-direction and s_y in y-direction. The shifted costs is then simply summed with the original cost and divided by two in order to yield the edge costs. The advantage of the **LineGraph** is that its structure allows to represent angle costs, or in general any di-edge cost in the original graph. A weight w_a describes the importance of the angle costs with respect to the other resistance values, and an angle constraint δ describes the maximum feasible angle. Let $\hat{e}$ be an edge in the line graph $L(G)$, mapping to the tuple (e, f) of two adjacent edges in G. Then the new edge cost $c_{L(G)}$ is computed as the previous edge cost c (Equation 2.1) together with the weighted angle cost c_a:

$$c_{L(G)}(\hat{e}) = \frac{c(e) + c(f)}{2} + w_a \cdot c_a(\sphericalangle(e, f)) \ .$$

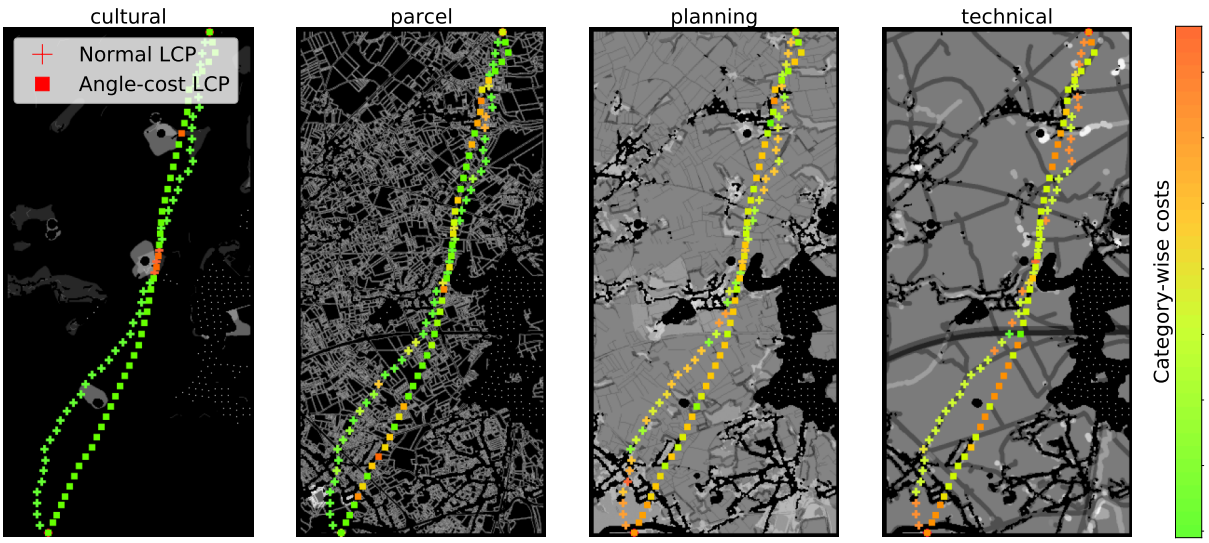


Figure 6: Category-wise costs for two shortest path outputs: Crosses show a normal least cost path, computed with the **NormalGraph** class. Squares show the optimal angle-cost shortest path, computed with the **LineGraph** class. The colouring shows for example that the technical costs for the **NormalGraph** path are lower (in the bottom part) since the route leads along streets, while the angle-penalized path rather follows a straight line.

Shortest path algorithms. Both **NormalGraph** and **LineGraph** call the shortest path method of **GeneralGraph**, and respectively transform the output. In the **LineGraph** it is necessary to add auxiliary source and target vertices and connect those to all vertices in $L(G)$ corresponding to outgoing (incoming) edges of the source (target) vertex in G respectively. Further, the `get_shortest_path` method yields a path of vertices in $L(G)$, which are thus converted to edges in G and transformed to cell coordinates of transmission towers.

Figure 6 shows exemplary outputs of a shortest path computation. Note that the four parts of the plot show the same paths, but visualized on different cost categories ("cultural", "parcel"[4], "planning" and "technical" costs). The colour of a pylon describes the relative resistance in this location. The figure compares the costs of a least cost path (LCP) computed with the **NormalGraph** class with a **LineGraph** path that simultanously minimizes the angles. Clearly, the angle-optimal path accepts higher resistances for the sake of straightening the route (see for example the parcel or technical cost in the bottom part). The **NormalGraph** output instead takes many small turns to place pylons at parcel borders and along roads.

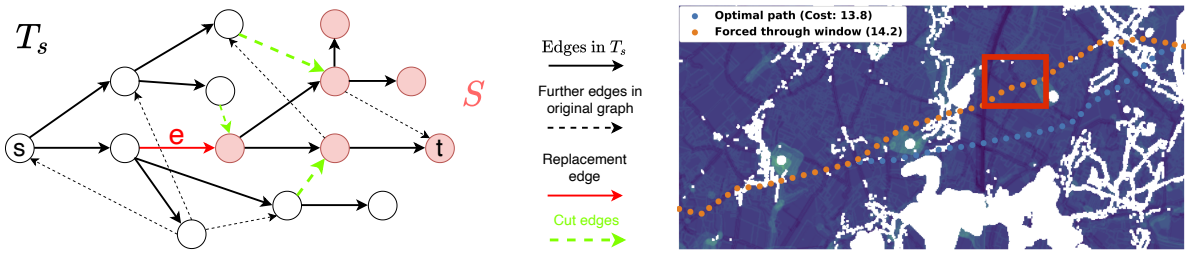
K shortest paths. Furthermore, investors might be interested in comparing a set of k diverse, comparatively short paths instead of a single shortest path. Theoretical background as well as details on the algorithms implemented in **NormalKSP** and **AngleKSP** are discussed in section 5. Most implemented algorithms are based on Eppstein's algorithm [31] that involves the computation of two shortest path trees, i.e. the shortest paths from source to all vertices as well as from the target to all vertices (in the complementary graph of reversed edge directions).
The method `shortest_path_tree` implements this functionality and yields distance and predecessor maps for both directions. The methods `find_ksp` and `dispersion_ksp` subsequently return a set of sufficiently diverse paths.

[4]It is beneficial to place transmission towers at the border of parcels in order to reduce the disturbance of cropland or other land usage

3.2.2 Implicit line graph

In this work we developed a novel algorithm and graph implementation that we name "Implicit Line Graph", to describe that the size of the graph remains the same and the line graph is not build explicitly, but instead the angle costs are minimized efficiently with a variant of the Bellman-Ford algorithm. Our algorithm and theoretical analysis will be explained in section 4. Regarding the implementation it is important to note that no graph library was used, thus missing the efficiency of the `graph-tool` library. Instead, the code was optimized with the `numba` package in Python which translates and compiles Python code to C++ at runtime. The graph itself is not build and stored explicitly; instead, only a `dists` array stores the distance of each edge from the source vertex for each edge. During execution of the shortest path algorithm, the distances are updated and the predecessor of each edge is stored in a `preds` array. The utilized data structures exploit the regularity of the graph in power infrastructure layout, where each vertex has the same number of neighbors if edges in forbidden areas are represented with infinity cost. To be specific, `dists` and `preds` are arrays of size `number of feasible pylons` × `number of possible neighbors`. **ImplicitLG** implements similar methods as **NormalGraph** and **NormalKSP**: Neighbors and edge weights can be initialized, and a single shortest path can be computed, or the two shortest path trees rooted in source and and target vertex.

Diverse and alternative paths An implicit line graph object where both shortest path trees have been computed can serve as input to the classes **AngleKSP** or **Alternative**. The former implements the same functionality as described above and discussed in section 5, namely the computation of k diverse shortest paths, whereas the latter is an extension that allows to compute *replacement paths* or to force the path through a specified region. Although the *replacement path problem* in the literature usually denotes the problem to find shortest paths that avoid each of the edges on the path, here we only consider the replacement of *one* edge. The reason is that in the present application on power infrastructure layout it is not relevant to find a "robust" path where any edge can be avoided, but it might indeed be a desired functionality to specify one component on the path that should be circumvented without the need to recompute the shortest path. This way, a planning agency might see a critical point on the route and can efficiently avoid it.



(a) Avoid a region **(b)** Traverse a region

Figure 7: Methods implemented in the class **Alternative**: A user can specify an edge on the path that should be avoided (a) or a region that the path must traverse (b)

.

In the PowerPlanner framework we tackle the single-edge replacement path problem with a method following for example the work by Kare [58]. Our algorithm utilizes the shortest path trees T_s rooted in s and T_t rooted in t, and their corresponding predecessor maps P_s and P_t. The setting is visualized in Figure 7a. Let $e = (u, v)$ be an edge on the optimal path Q that is to be replaced. Then we consider the set of vertices S in T_s that use e on their shortest path from s to t (red vertices in Figure 7a). Note that S is itself a subtree of T_s rooted in v. It then only remains to compute the optimal *cut edge* of S: As visible in Figure 7a, the cut edges

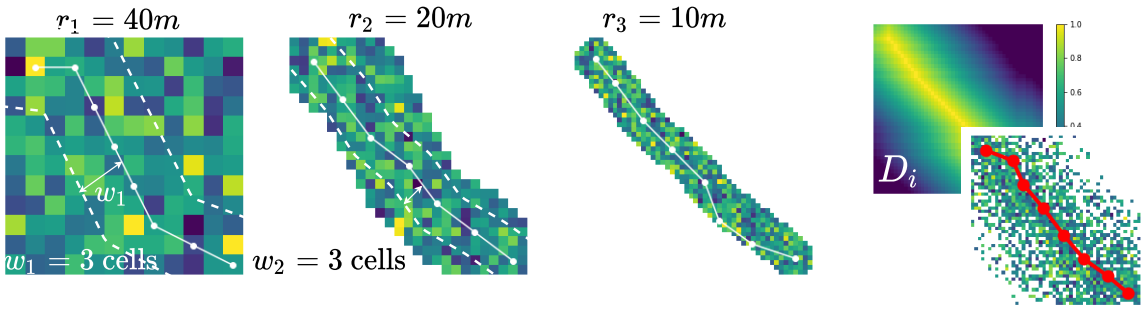
coloured in green cover all alternative connections from s to t that circumvent e. The possible cut edges can easily be determined from T_s. Furthermore, with Eppstein's algorithm [31] one can efficiently compute for all edges the distance of the shortest path that uses this edge (details in algorithm 5). Thus, Eppstein's algorithm [31] together with one iteration over the edges to determine the cut edges are sufficient to compute the optimal replacement path.

On the other hand a user can also decide to force a path through a specific region. This functionality is useful if prior knowledge is available that is not expressed in the input data, or simply for comparison. The `path_through_window` method in the **Alternative** class implements a further extension of Eppstein's algorithm that provides this functionality. Given a window in the raster or simply a set of vertices, the Eppstein distances of each of these vertices can be compared and the optimal one selected. The path is subsequently reconstructed from P_s and P_t. An example is shown in Figure 7b, where the orange path is forced to cross the region marked by the red window.

3.2.3 Pipelines

Despite efficient representations and processing, the graphs can still become extremely large. Problematically, an increase in resolution leads to a quartic increase in m, the number of edges, i.e. a resolution of 10m instead of 20m results in 16 times as many edges. This is due to the upsampling-caused quadratic increase of the number of cells, which affects both the number of vertices as well as the number of neighbors of each vertex, because the number of cells in the rasterized ring between d_{min} and d_{max} is increased quadratically.

To decrease the memory requirements and runtime, we suggest to start with a low resolution and refine the graph only in relevant regions. The procedure can be repeatedly applied to increase the resolution and decrease the size of the considered region in an iterative process, as visualized in Figure 8a. The approach is implemented in all graph classes, where the method `set_corridor` takes care of downsampling the instance and defining the corridor. A *pipeline* is a list of tuples of the resolution r and the corridor width w around the previously path: $\mathcal{R} = [(r_1, w_1), \dots (r_t, w_t)]$. For example, in the pipeline $\mathcal{R} = [(2, 50), (1, 0)]$ the instance is first scaled down by a factor of 2 (e.g. 20m resolution instead of 10m) and the optimal path P_1 is computed. Subsequently, the corridor is restricted to all cells with less than 50 cells distance from P_1, and this time the instance is not downsampled (factor 1). The shortest path in the corridor is computed and returned. Note that for efficiency reasons the corridor around the current path P_i is formed by a



(a) Basic pipeline processing **(b)** Random edge sampling

Figure 8: Pipelines: An instances is downsampled to a lower resolution r_i, and the optimal path is found in this simplified setting. Then a corridor of width w_i around this optimal path is selected and the shortest path is computed on a raster with higher resolution only in this corridor. The steps are executed iteratively until reaching the highest available resolution.

morphological operation called *binary dilation*. Instead of computing the distance to P_i for each cell, a binary array (where only the cells of P_i are set to one) is convolved multiple times with an all-ones kernel that marks all surrounding cells of the current path. For details on binary dilation, see [49].

Watershed Tranform. In the downsampling step of the pipeline described above, neighboring cells are simply summarized by their mean value. However, it is possible to preserve more information of the cost surface with image segmentation methods. The *Watershed Transform* [16] is a well-known segmentation algorithm in computer vision, achieving that neighboring pixels of similar values are assigned to the same segment. First, an edge image is computed with a gradient-kernel filtering. Seeds for the later segments are distributed in a regular grid. Then, the image is "flooded from the seeds", whereby strong edges (strong differences in value) separate the "water pools" (corresponding to segments). After applying the Watershed Transform to the cost surface, vertices can be placed in the center of each segment, and their cost is the mean value of its segment. It can be observed that the path costs converge faster to the correct costs when using a *Watershed Transform* instead of simple downsampling. The improvement of course comes at the cost of a longer runtimes for the downsampling step itself.

Random graphs and pipelines. An alternative approach to reduce the number of edges is (informed) random sampling, such that an edge is only added with probability p. A prior on the location of edges can be defined, for example with highest probability close to the straight line between source and target, and with decreasing probability towards the instance boundaries (Figure 8b). The prior could also be a previously computed shortest path in a *random pipeline*, as explained further below.
The class **RandomGraph** contains methods to set the prior (`set_corridor`) and to sample edges, which are used during the respective `compute_edge_costs` methods of **NormalGraph** and **LineGraph**.

The **RandomGraph** class provides functionality to randomly sample edges from a given prior distribution. A pipeline of random sampling is defined by a list $\mathcal{R}_{\text{random}} = [D^1, \dots, D^t]$ where $D^i \in [0,1]^{m \times n}$ is an array with values between zero and one defining the keep-probability of an outgoing edge for each cell. For example, a value in D^i could be computed as a normal distribution (centered on zero) of the distance of the cell from the shortest path. The mean value, $\mu(D^i)$, determines the overall expected ratio of kept edges. Edges are sampled according to the cell-probability D_i in the following manner: For each neighbor-shift (s_x, s_y), a 2-dimensional array of uniformly random values A is generated ($\mu(A) = 0.5$). If $\hat{D}^i_{k,l} > A_{k,l}$, the edge $\left(l^{-1}(k,l), l^{-1}(k+s_x, l+s_y)\right)$ is added to the graph.

The keep-edge-probability can even be set automatically based on a given maximum space capacity. If a limit on the number of edges is placed, e.g. $m \leq \theta_m$, then the mean of the prior distribution D^i is scaled to place an expected maximum of θ_m edges. This approach is user-friendly since the user does not have to understand how much memory is required by a graph of a certain resolution and specific $d_{\min}$ and $d_{\max}$, but instead only needs to set an upper bound dependent on his memory resources.

3.2.4 Multi-objective optimization

Transmission line layout clearly presents a multi-objective optimization problem, aiming at the simultaneous minimization of monetary costs, angles, and GIS-based resistances of various cate-

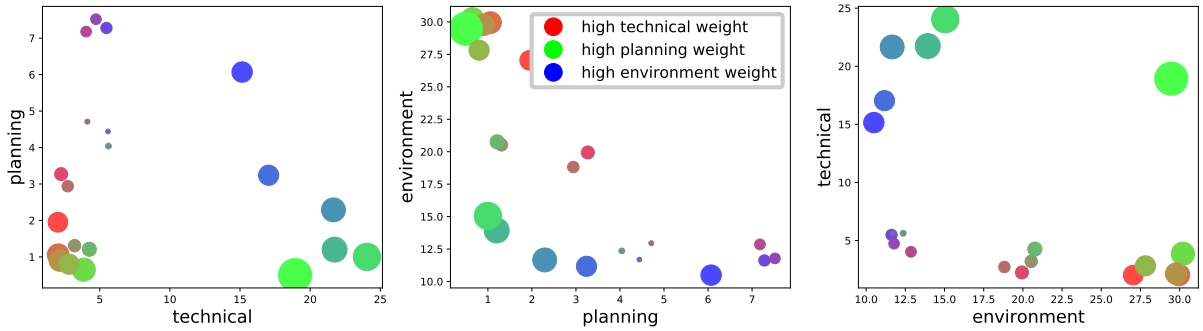


Figure 9: Convex envelope of the pareto frontier for multi-objective optimization of three categories. The size of each bubble corresponds to the unweighted sum of costs. From the pareto frontier one can for example conclude that planning costs and technical costs can be optimized simultaneously, whereas the trade-offs are significant between environment and planning costs.

gories. In the PowerPlanner framework, these trade-offs are parametrized with weights, namely angle (w_a), edge (w_e) and category weights (w_c).

In an application, planners might struggle with the exact weight setting, and may be interested in an analysis of the effects of changes in the parameter setting. Bagli et al. [9] show that "the use of different weight scenarios may help making the model adaptable to varying environmental and social contexts." Since the solution of a multiple-objective optimization problem can not be described in a single result, research aims at the computation of the pareto-optimal solutions, i.e. the solutions where decreasing the cost of one criterion leads to the increase of another one, such that no strictly better solution in both criteria exists [100, 38]. The set of all such solutions is the *pareto frontier*.

Due to numerous applications, many methods were proposed for bi- and multiobjective shortest path problems [26]. Often, label-setting-type shortest path algorithms are adapted to compute all pareto-optimal path for the multi-objective [69, 98, 55] and bi-criterion case [94]. However, it was shown that even for shortest path problems the corresponding decision problem is NP-hard to solve, namely to determine whether a solution exists that does not exceed any threshold on the criteria [29]. An efficient fully polynomial time approximation scheme (FPTAS) was given by Hassin [51] and improved [32, 65], but these methods rather target the *resource constrained* shortest path problem than the computation of pareto-optimal solutions. A practically efficient approximation scheme for multi-objective shortest path problems was developed by [14] who even present an application on power infrastructure layout. Other applied work in resource-constrained or multi-objective optimization, for example [48], iterate through the k shortest paths until finding feasible or pareto-optimal solutions, and Handler and Zang [45] developed methods to reduce k in this approach.

Although the computation or approximation of the pareto frontier is hard since even the number of solutions can be exponential, each minimization of a weighted sum of criteria yields a point on the convex envelope of the pareto frontier (see for example [88]). For our purposes the convex points were found to suffice, as they already visualize trade-offs between criteria in power infrastructure layout. We therefore simply vary the weights of two or three weights systematically, and visualize the resulting solutions and their corresponding costs.

An example is shown in Figure 9, depicting the effects of the three category weights on planning-, environmental- and technical costs in **instance 3**. A red point for example represents a path computed only on technical costs, i.e. with zero weight on planning and environmental costs. The size of the point is proportional to the unweighted sum of costs. For example, it is apparent

that mixtures of higher technical- and environment weighting yield comparably low sums of cost (not weighted). In the example in Figure 9 it is visible that planning and technical costs can be minimized simultaneously, while very low environmental resistances lead to high planning or high technical costs.

3.3 Experiments

We evaluate the implemented methods on the three instances introduced in subsection 3.1. Unless noted otherwise, the default parameter settings for each instance are used as listed in Table 5 in appendix A. In consultation with the collaborator Gilytics it was decided for a hyperparameter setting that allows to process the instances as a *directed acyclic graph (DAG)*. The significant reduction in runtime in a DAG justifies the constraints on the edges, excluding only arcs in opposite direction to the straight line between source and target. Such turns are highly unlikely to appear in linear infrastructure layout. Note also that the comparisons below are still fair because any of the algorithms would (at least in theory) be slower by the same factor if it was not processed as a DAG. Furthermore, we do not assume a linear angle cost function here which would lead to further acceleration of the processing in the **ImplicitLG** class.
All experiments were executed on a compute node equipped with two 64-core AMD EPYC 7742 processors (2.25 GHz nominal, 3.4 GHz peak) and 512 GB of DDR4 memory, clocked at 3200 MHz. Note that no parallelization was used and the runtime in the possibly cyclic case could be improved with a parallelized implementation of the Bellman-Ford algorithm.

3.3.1 Graph comparison

First, we compute the optimal route from a given source to a given target vertex for all combinations of instances and graph implementations, and at 10m, 20m and 50m resolutions. A comprehensive overview of all results is shown in Table 2. Note that some configurations were left out (partly empty lines); those are the cases where the graph became extremely large - in the order of trillions of edges - and could therefore not be processed on consumer hardware anymore[5]. However, due to efficient implementations, all cases that can be executed actually run in less that 7 minutes.

As mentioned in the previous chapter, the size of the graphs increase cubically with the resolution, since both the number of vertices and the number of feasible neighbors increases quadratically. Table 2 shows that in particular the **LineGraph** graphs increase significantly in size with increasing resolution, causing two trillion edges already for the smallest instance at 20m resolution. Figure 10a shows the graph size in log scale for 10m, 20m and 50m resolution. The difference between the size of **NormalGraph** or **ImplicitLG** objects compared to **LineGraph** objects is substantial. While with a **NormalGraph** or **ImplicitLG** it is feasible to process all example instances at 10m resolution if a big RAM is available (2-3 trillion edges), the **LineGraph** is clearly restricted to smaller instances and lower resolution. Note that the number of edges of **NormalGraph** objects and **ImplicitLG** objects is equal and the values in Table 2 only differ due to their internal representation. Also, the runtime of **NormalGraph** objects and **ImplicitLG** objects is suprisingly similar in some cases, which is however due to implementation

[5]Instance 3 at 10m resolution for example is modeled by a graph of 2 trillion edges, which corresponds to 2GB if each edge is stored in one byte. However, the edge cost, distance and predecessor of each edge must be kept in RAM, and floating points require 8 bytes in Python, leading to a factor of $8 \cdot 3 = 24$ more required bytes. The largest machine available for our experiments thus failed for a graph with more than a trillion edges.

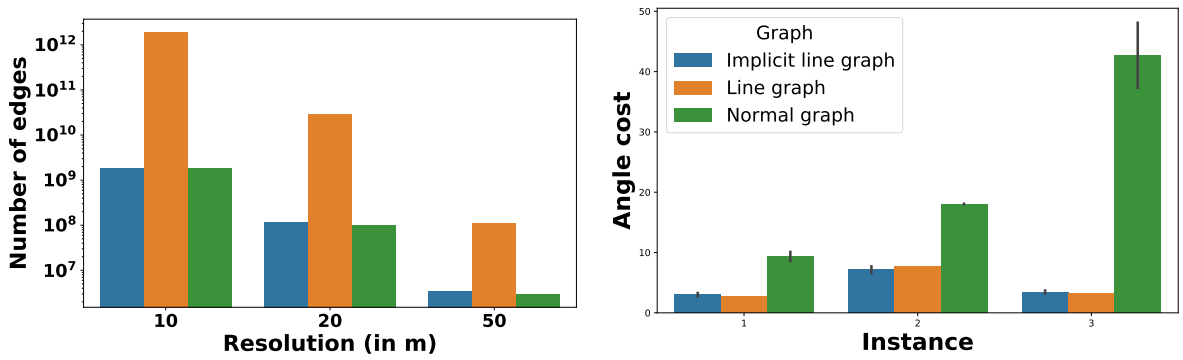
details. The **NormalGraph** is implemented with the `graph-tool` package, which is generally more efficient, but the shortest path computation is not even the bottleneck: Most time is spent for building the graph, since the interface between Python (where the edges are defined) and C++ (where the edges are added to the graph) is slower than the shortest path computation that is done completely in C++ once the graph is complete. It would be a more fair comparison to use the structure of the **ImplicitLG** and compute the shortest path without angle cost, but here we aimed to compare with an implementation using a widely-used graph library. The actual efficiency follows from the theoretical analysis in section 4.

	Resolution (meters)	Graph model	Time (seconds)	Number of edges (in millions)	Angle cost	Resistance	Combined cost
Instance 1 (angle weight: 0.1)	**10**	**Implicit line graph**	409.3	286.3	3.3	8.9	9.2
		Line graph		13 621.1			
		Normal graph	248.8	248.0	8.6	8.7	9.5
	20	**Implicit line graph**	19.3	18.2	3.3	9.9	10.2
		Line graph		2 129.1			
		Normal graph	16.6	16.0	9.5	9.7	10.5
	50	**Implicit line graph**	3.6	0.6	2.7	11.0	11.2
		Line graph	15.8	9.6	2.7	11.0	11.2
		Normal graph	0.7	0.5	10.1	10.8	11.7
Instance 2 (angle weight: 0.2)	**10**	**Implicit line graph**		3 001.6			
		Line graph		4 504 370.9			
		Normal graph		3 001.6			
	20	**Implicit line graph**	230.3	187.1	6.6	9.8	11.0
		Line graph		69 380.7			
		Normal graph	108.8	152.1	18.0	9.3	12.3
	50	**Implicit line graph**	7.4	5.4	7.7	10.6	11.9
		Line graph	334.3	241.0	7.7	10.6	11.9
		Normal graph	3.3	4.5	18.1	10.0	13.0
Instance 3 (angle weight: 0.4)	**10**	**Implicit line graph**		2 259.3			
		Line graph		1 068 835.5			
		Normal graph		2 259.3			
	20	**Implicit line graph**	144.4	139.8	3.7	5.2	6.3
		Line graph		16 295.6			
		Normal graph	191.2	132.7	37.3	4.2	14.9
	50	**Implicit line graph**	13.0	4.1	3.2	6.7	7.6
		Line graph	191.8	75.2	3.2	6.7	7.6
		Normal graph	7.7	3.9	48.1	5.0	18.8

Table 2: Comparison of implemented graph algorithms. All values are averages over various parameter configurations. A row that is partly blank indicates that the shortest path could not be computed because of insufficient memory resources. Both the line graph and the implicit line graph achieve the same improvement in angle cost while maintaining low resistances. Thereby the combined cost (weighted angle cost plus resistance) is optimized. However, the size of the line graph is by orders of magnitudes larger than the normal graph, while the implicit line graph yields an optimal solution with the same space requirement. The higher the angle weight which is different for each instance, the larger the difference of achieved combined costs.

Meanwhile, the motivation to build a **LineGraph** was the possibility to minimize angle costs (see section 4). Figure 10b shows our approach(**ImplicitLG**) achieves the desired reduction in angles equally well as the **LineGraph** that was implemented as a baseline[6]. Accepting only slightly higher resistance values, both algorithms substantially reduce the angle costs. Thus, the combined cost, i.e. the angle weight times the angle cost plus the resistances, is minimized. In all cases, the sum of resistances is increased marginally (less than 10% for **instances 1 and 2**, at most 34% for **instance 3**), while the angle cost decreases by 50-90%. The decrease of angle costs is controlled by calibrating the angle weight, which was set to 0.1 for **instance 1**, 0.2 for **instance 2** and 0.4 for **instance 3** after qualitative tests. Therefore the gain in the combined cost is largest for **instance 3**: The overall cost achieved by the normal graph is more than twice as high as for the angle optimization algorithms (see **instance 3** in Table 2).

[6]Note that the difference between angle costs achieved by the line graph and the implicit line graph only stem from averaging over resolutions, where the missing values for the line graph impact the average (compare Table 2)



(a) Graph size (log scale)

(b) Angle cost comparison

Figure 10: Comparison of graph implementations: A normal graph is efficient but not able to minimize angle costs (b), while a line graph structure optimizes angles at the cost of significantly higher costs (a). The implicit line graph combines both advantages.

3.3.2 Comparison to baseline algorithms

In addition to the comparison of different global optimization methods, we were also interested to contrast our work, the implicit line graph, to previous approaches in the literature. As explained above, most related works compute a corridor through the resistances and optimize pylon positions subsequently, instead of computing a globally optimal *pylon route* as in our graph model. Thus, we assume as a baseline that only a path through a grid-like graph is computed, and then the transmission towers are placed on the output path as a second step, similar to the methods in [73, 102, 105, 63, 112]. In the implementation of this baseline approach we place edges in the 8-neighborhood of each vertex, resembling a grid structure. We then compute the shortest path through the grid, which yields a dense path of cells from source to target vertex. In this optimal "corridor", we further use our usual processing to place transmission towers. Note that here we even optimize the tower placement, although in practice it would often be done manually after the corridor computation.

		Time (seconds)	Number of edges in millions	Angle cost	Resistance	Avg. distance (in m) to baseline	Max. distance (in m) to baseline
Inst. 1	**Baseline**	39.8	**2.2**	9.9	110.6	0.0	0.0
	Ours	436.8	276.7	**4.7**	**97.9**	422.2	1187.0
Inst. 2	**Baseline**	56.3	**3.5**	29.5	23.5	0.0	0.0
	Ours	155.9	138.1	**15.8**	**18.9**	743.7	3078.8
Inst. 3	**Baseline**	20.2	**1.4**	15.5	13.7	0.0	0.0
	Ours	337.3	178.3	**11.2**	**11.4**	354.1	1514.9

Table 3: Comparison of our graph modelling to the baseline, a grid-like approach. The better result is marked bold in each case. Although our approach leads to much larger graphs, the optimal path can still be computed in minutes and in all cases leads to significantly lower angle and resistance costs, and substantially diverse routes.

Table 3 shows the results for all three instances. **Instance 1** was processed at 10m resolution, **instances 2 and 3** at 20m. All results are averages over three runs with varying angle weight (0, 0.05, 0.1 and 0.3). The angles of the corridor in the baseline approach were also reduced with our minimal-angle shortest path algorithm to provide a fair comparison with respect to angle costs.

Obviously, our graphs are much larger than the baseline, by a factor of 40 to 125 dependent on the resolution and the minimum and maximum distance between pylons. However, all three

instances can still be processed in less than 10 minutes with the given resolutions[7]. Most importantly though, the analysis proves that the costs can be reduced significantly with our approach. The angles are reduced by around 50% for **instance 1 and 2**, and 28% for **instance 3**, and resistances by 11-22%. Considering that millions are spent on transmission line construction, these differences are of huge impact. Thus, even an algorithm that takes days but reduces the resistances significantly is desirably, in particular since the algorithm only needs to be executed *once* - after a suitable choice of parameters - during a sometimes years-long planning process.

Furthermore, we were interested to observe how much the output paths differ. It turns out that the optimal path as computed with our method is on average 422m away from the baseline path for **instance 1**, and the largest distance between both amounts to 1.2km (see Table 3). Similar and even larger differences are shown for **instance 3 and 2** respectively. Clearly, a local optimization around the baseline corridor is not sufficient to yield the optimal path, since it follows an entirely different route.

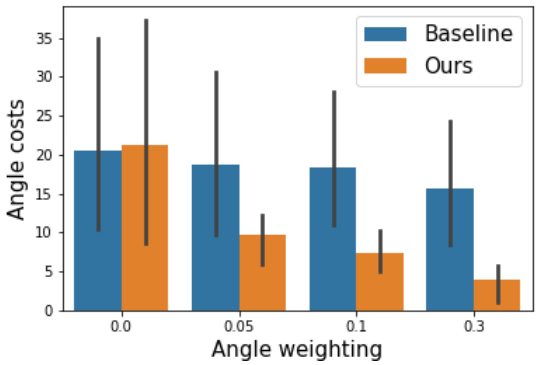


Figure 11: Average angle costs and the standard deviation over experiments on diverse instances are shown. In a global optimization, the angle weight can directly optimize the angles between transmission towers.

Last, besides the averaged results in Table 3 the costs were compared for different angle weights. In all cases, our computation yielded strictly better resistance and angle costs than the baseline method. A comparison of the angle costs is shown in Figure 11. While our approach directly optimizes the angle between pylons, in the baseline approach the angle weight does not affect the path equivalently, because only the angles of the corridor are optimized and the pylon spotting leads to different angles than the angles of the corridor itself. In some cases, a higher angle weight even led to a *larger* angle cost in the baseline approach, demonstrating the inability of the corridor-optimization approach to model the angles between the discrete pylon locations in the end.

3.3.3 Pipeline experiments

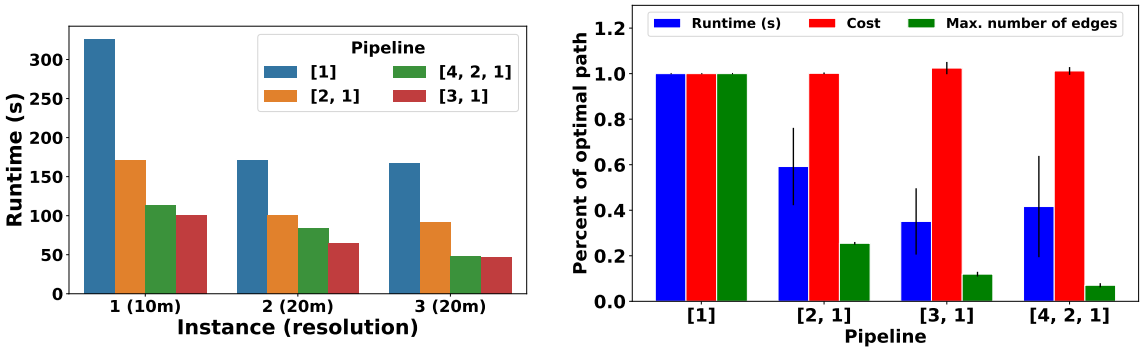
Furthermore, we evaluate the ability of iterative approaches to reduce runtime while at the same time retaining low resistances. First, four different pipeline configurations were tested and compared to direct path optimization. Here, **instance 1** was processed at 10m and **instance 2 and 3** with 20m resolution. Four different pipelines are used: $[1], [2, 1], [3, 1]$ and $[4, 2, 1]$. The values correspond to the downsampling factors, e.g. in $[2, 1]$ the resolution of the instance is first reduced by a factor of 2, retaining 25% of the vertices. We automatically compute the corridor width for later stages of the pipeline to prohibit the graph from exceeding the number of edges in the first stage[8]. Pipeline [1] refers to a shortest path computation in a single step and thus corresponds to the baseline, i.e. a direct execution without pipeline.

Figure 12a demonstrates that the pipeline approach does indeed improve the runtime signifi-

[7] The runtimes differ slightly from the ones in Table 2 since they were processed at another time and thus at a different rate of utilization of the processor. Similarly, the resistances are different simply due to another scaling method that was used (the absolute values are not meaningful)

[8] In the first stage, the whole instance is considered, so only the sampling factor determines the number of edges. In later steps the width of the corridor around the currently optimal path can be set to the maximal size possible that leads to less edges than in the first stage

cantly. The higher the resolution, the larger the difference in runtime between direct and iterative computations. Although the shortest path needs to be computed multiple times (in each pipeline step), the lower number of edges leads to reduced runtimes for each step. It is however apparent in Figure 12a that due to its three steps the runtime of pipeline $[4, 2, 1]$ is longer than the one of $[3, 1]$, even though the latter one uses more edges.



(a) Runtime by instance and pipeline

(b) Normalized costs, time and space efficiency of pipelines

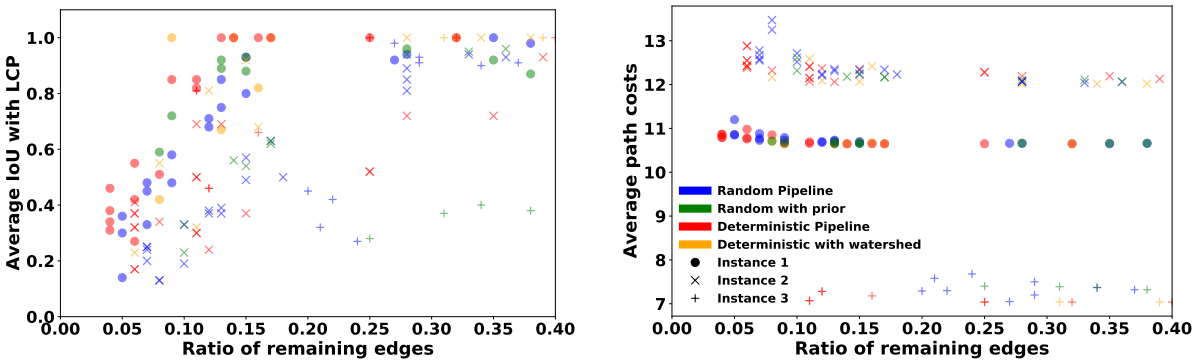
Figure 12: Evaluation of iterative (pipeline) methods to reduce time and space requirements. The outputs were computed for three instances with four different pipelines ([1] refers to a direct optimization in one iteration). The runtime is decreased significantly for all instances when using iterative procedures (a). The space requirements with respect to the number of edges is even lower. Meanwhile, the resistances increase only marginally as desired.

Meanwhile, Figure 12b shows that the reduction in runtime does not come with substantially worse resistances. The runtime, the number of edges and costs are shown here as averages over all instances, but normalized with respect to the baseline pipeline ([1]). For example, pipeline $[3, 1]$ requires 10% of the original space since in none of its pipeline steps the graph exceeds 10% of the number of edges in the baseline. Clearly, the path resistances are only slightly larger on average with low standard deviation (< 0.03), whereas the runtime and space efficiency is improved by far. For example, for **instance 3** the optimal path is found with a third of the baseline runtime and 8% of the baseline space requirements.

It is thus shown that the application of pipeline approaches is extremely useful for large instances since it offers a way to process any realistic instance and most probably yields close to optimal solutions.

Comparison of random and deterministic pipelines. Furthermore, it was explored how deterministic and randomized pipeline strategies perform in different parameter settings. Because of the large number of tests, **instance 1** was only processed with maximally 20m resolution and **instance 2 and 3** with 50m. We aimed to analyze how much the number of edges and thereby the memory requirements and runtime can be reduced, while maintaining an output path of low cost and high similarity with the optimal route. For this purpose, we varied the pipeline parameters (sampling factors r_i and corridor widths w_i, or distributions D^i) and observed the achieved path costs after the last pipeline step. In the last step of the pipeline, the output path is always computed at maximum resolution (20m or 50m), but in a narrow corridor.

Figure 13 and Figure 14 present the results on **Instance 1, 2 and 3**. In the randomized case, the values are averages over 30 iterations with the same configuration. The "ratio of remaining edges" is the maximum of edges used over all pipeline steps. In contrast to the analysis above, we did not adapt the width of the corridor to fit the number of edges in the first stage, but set a parameter γ (in a range from 0.05 to 0.4) which sets an upper bound on the ratio of remaining



(a) Average intersection with the optimal path

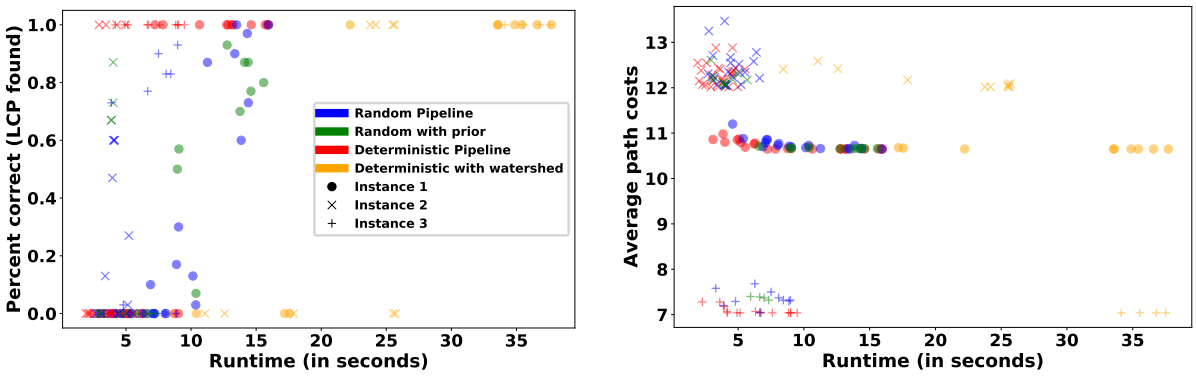
(b) Average path costs by space requirements

Figure 13: Comparison of pipeline strategies to iteratively refine the path: We evaluate the output path (after the last stage of the pipeline) by its costs and its intersection with the least cost path. It is apparent that regular deterministic downsampling methods (yellow and red) outperform randomized strategies (green and blue) on all instances, even if a good prior is used (green). A prior however does help in the randomized procedure, since lower costs and higher intersections can be achieved. The watershed transform produces similar results as simple downsampling.

edges with respect to the baseline. The following configurations were tested, where each value again corresponds to the downsampling factor: $[2, 1], [3, 1], [3, 2, 1], [4, 1], [4, 2, 1], [5, 1], [5, 2, 1]$. For example, in a pipeline $[3, 2, 1]$ and $\gamma = 0.2$, the instance is downsampled by a factor of 3 in the first step of a deterministic pipeline, or edges are sampled with a probability of $1/9$ in a randomized pipeline. Then, we compute the maximal width that the corridor can have in the next step, given the constraints that first the instance is only downsampled by a factor of 2 in the next step and second the number of edges must be below $0.2 \cdot m$. Dependent on the first factor in the pipeline, different γ values are explored because $\gamma \cdot m$ must be larger or equal than the number of edges remaining after the first downsampling step of the pipeline. For random pipelines, the distribution D was chosen as a Gaussian distribution, centered on the previously computed path. The scale of the Gaussian is also computed from the sampling factor in a similar manner as the corridor width in the deterministic case.

Figure 13 in general shows that the deterministic downsampling approach outperforms randomized methods slightly. For each instance, the IoU with the optimal path is higher for deterministic outputs (yellow and red) than for randomized procedures (green and blue). Similarly, the costs of the former are a bit lower than the average costs of the latter, and sometimes randomized pipelines lead to a signicant increase of costs (see for example Figure 13b, Instance 2, 10% remaining edges). However, in general the costs are less than 10% higher than the optimal costs, showing that the pipeline approach is able to find good outputs at significantly reduced costs. In some cases even with 10-15% of the edges the optimal path is found successfully.

Furthermore, we tested different options as introduced above. Firstly, we set a prior on the distribution of edges in the first stage of the random pipeline. So far, D_1 is a uniform distribution where each edge is equally likely to be sampled. Instead, one can define a prior, for example a Gaussian centered on the straight line between source and target. The green dots in Figure 13 show randomized pipelines with this prior in contrast to the uniform case (blue). It is visible in Figure 13a and Figure 14a that a prior does indeed improve the results (compare for example the green and blue points in Instance 1), but is not sufficient to achieve the same performance as the deterministic pipeline. Secondly, in the deterministic case we compare normal downsampling (the replacement of several neighboring pixels by their mean value) to a clustering approach, the Watershed Transform. As apparent in Figure 14, the Watershed Transform requires more time and has to be executed in each step of the pipeline, thus increasing the runtime significantly.



(a) Ratio of correctly found paths

(b) Average path costs by time

Figure 14: Pipeline comparison by runtime: Clearly, the watershed transform significantly increases the runtime. A prior improves the ratio of correct computations for random pipelines (a). The costs do not directly decrease with runtime (see for example b, Instance 2); instead the performance seems to depend largely on the specific pipeline parameters.

This is however also due to the generally low runtime; if much larger graphs were used, the downsampling would not be the bottleneck. In Figure 13a it is not clear whether the Watershed Transform improves the convergence to the ground truth path. Only in pipelines where more edges are remaining, the Watershed Transform sometimes leads to the optimal path whereas normal deterministic pipelines do not (compare red and yellow crosses for Instance 2 in the range of $0.25 \leq \gamma \leq 0.4$).

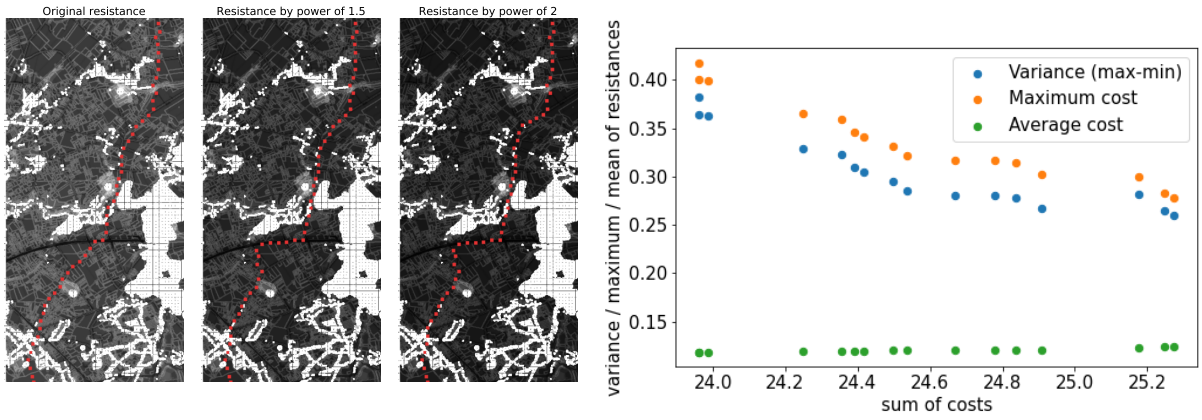
Last, observe in Figure 14 that a larger runtime is only loosely correlated with a better performance. The Pearson correlation of the runtime and the ratio of correct outputs (Figure 14a) for random pipelines is only 0.5. In addition Figure 14b shows that optimal outputs can be achieved with very low runtime. Note for example that for **Instance 2** no reduction in cost with pipelines of larger runtime is visible. It follows that the specific parameter settings might be as important as the maximal graph size determined by γ. No significant difference was found between short and longer pipelines (2 versus 3 stages).

In short, both randomized and deterministic pipeline approaches can compute the optimal shortest path with low space requirements and runtime, however only with a good parameter configuration. The deterministic approach is preferable except for the case that a good initial prior distribution is known.

3.3.4 Minimum variance shortest path

In power infrastructure planning it can be desirable to avoid regions of high resistances by any means. In subsection 6.2 we discuss theoretical results on the complexity of the last *average* cost path, while simple practical approaches are presented here. Two strategies were explored. First, regions of high cost were penalized unproportionally by applying a non-linear function to the resistance values. For example, squaring the resistances yields routes with larger resistance with respect to the original cost surface, but possibly lower maximum resistance. However, the function applied to the resistances must be carefully calibrated. Figure 15a examplifies this approach, demonstrating how low resistances (dark) become unproportionally lower, and how the path changes accordingly, for example following a street with low resistance.

On the other hand, we propose to lower the maximum resistance appearing on the output path



(a) Penalizing high resistances

(b) Variance lowering by iterative cost thresholding

Figure 15: Two methods are employed to avoid high cost regions: a depicts how high resistance is penalized by applying a power of 1.5 or 2 to the resistances. The output routes in the modified resistance map tend to follow roads which stay as the only low resistance areas. b shows how iterative lowering of an upper bound on the resistances affects the sum of costs and average costs.

in several iterations: In the first iteration, the unmodified resistance instance is input. Let m_i be the maximum cost of the path P_i computed in iteration i, i.e. $m_i = \max_{e \in P_i} c(e)$. Then in iteration $i+1$ all cells of the instance with resistance larger than m_i are set to infinity cost. The path P_{i+1} thus has a lower maximum cost than P_i. The last iteration is reached as soon as the source and target location are not connected anymore due to infinity resistances. Figure 15b shows the change in maximum and average cost for **instance 2** over all iterations. Clearly, the "variance" defined as the maximum minus the minimum cost is decreasing, but naturally the sum of costs is increased. The average cost is almost constant.

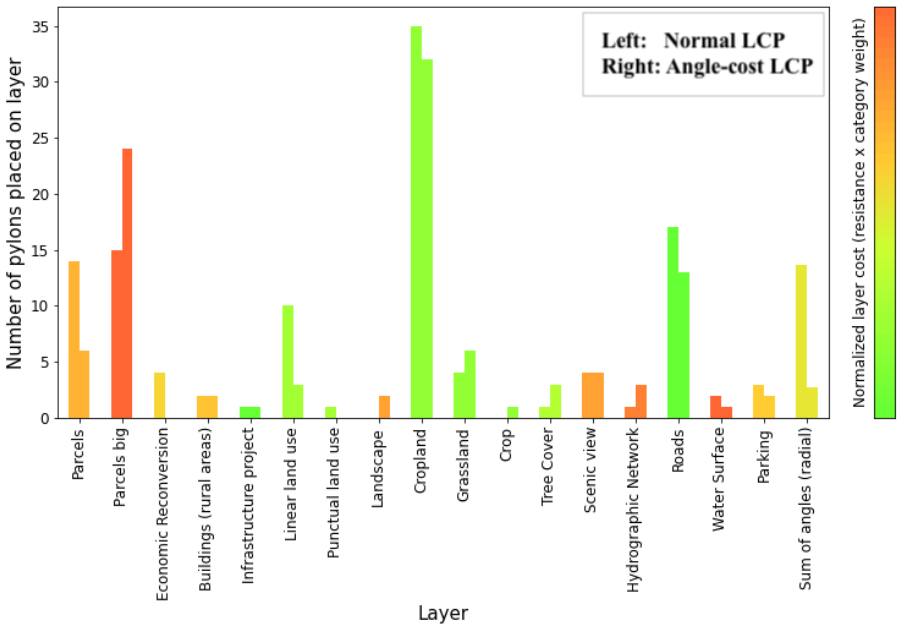


Figure 16: Layer wise cost comparison: The normal graph uses more roads and less inner parts of parcels for example, while the angle-penalized path takes less turns but therefore crosses more grassland and parcels

3.3.5 Geographic analysis and parameter sensitivity

Besides algorithmic work, we provide visualizations to analyze and compare output routes. Such visualizations can help planners to analyse the results and adjust the layer if needed. Figure 16 for examples compares the layer-wise costs of a path through **instance 1** computed with **NormalGraph** to the ones of an angle-optimal path computed with the **LineGraph** class (see paths in Figure 6). It is apparent that the angle-optimal path is less flexible to use parcel borders, roads and cropland, but the lower resistances of the normal least cost path (LCP) come at the cost of substantially higher angle cost. Furthermore, the PowerPlanner framework implements methods to conduct a sensitivity analysis with respect to the layer and category weights. An example and further details are given in appendix D. The analysis of one instance **instance 2** was presented to the client, a planning company, and was received well. Further work could evaluate the usability of the framework by conducting a study with experienced power infrastructure planners.

4 Efficient algorithms for the angle-cost shortest path problem

An interesting variant of shortest path problems is posed by considering costs or constraints on the *angle* between edges. Such di-edge costs can not be represented in the regular graph structure, and common shortest path algorithms are not directly applicable. Here, we first discuss previous work on turn constraints and adjacent-edge-costs, in particular the *Quadratic Shortest Path Problem (QSPP)*[83]. We then present a novel efficient algorithm to compute the shortest path with penalized angles.

4.1 Related work: The Quadratic Shortest Path Problem

Only recently the problem of costs on tuples of arcs was formally defined, named the *Quadratic Shortest Path Problem (QSPP)* [83]. Together with the regular arc cost $c : A \longrightarrow \mathbb{R}^+$, a cost on the interactions of arcs is defined as $q : A \times A \longrightarrow \mathbb{R}^+$, and the QSPP is the problem to minimize the cost of the shortest path P^*:

$$P^* = \underset{P}{\operatorname{argmin}} \sum_{a \in P} \Big(c(a) + \sum_{b \in P} q(a, b) \Big)$$

Applications of the QSPP problem include the variance-constrained shortest path [93] relevant for the transport of hazardous material, or lot-sizing [111] and the vehicle routing problem with correlated time reliability [84]. If the quadratic costs are constraints rather than costs, the problem is in practice often solved greedily by enumerating the shortest paths, e.g. with Yen's algorithm [109], until a path satisfies the constraint (see for example [93]).

Rostami et al. [83] prove that the general case, where interaction costs appear for any tuple of edges, is strongly NP-hard by reduction from the Quadratic Assignment Problem. Several algorithms have been proposed to approximate the general QSPP or solve it exactly. Rostami et al. [85] give an exact Branch-and-Bound algorithm and several ways to compute lower bounds, partly based on schemes proposed by Gilmore [42] and Lawler [62]. Hu and Sotirov [54] improve the approximate algorithms in various relaxations.
The authors of [83] further consider a variant of QSPP where only the interaction costs of *adjacent* arcs are nonzero. Crucially, this definition includes the case of angle costs appearing in the application on power infrastructure layout. The *Adjacent QSPP (AQSPP)* problem has been analyzed earlier, for example titled *reload cost paths* [107] by Amaldi et al. [6], who state that the reload cost problem is solvable in polynomial time based on a vertex-splitting construction. Rostami et al. [83] originally provide a second proof that AQSPP is solvable in polynomial time, using an auxiliary graph that closely resembles a line graph. In a later analysis Rostami et al. [85] however correct their claim, proving (by reduction from `3SAT`) that AQSPP is actually NP-hard as well. Hu and Sotirov [53] clarify that AQSPP is indeed a polynomial problem in the case of a DAG and they independently give a another proof on the NP-hardness for AQSPP on cyclic graphs. Hu and Sotirov [53] further describe special cases that are *linearizable*, i.e. the interaction costs can be expressed as regular costs, which for instance applies for regular grid or hypercube graphs with symmetric weak sum cost-matrix.

It is however important to note that both Rostami et al. [83] and Hu and Sotirov [53] only regard directed graphs. In the undirected case, at least the proof of Hu and Sotirov [53] for cyclic AQSPP fails: They reduce from the 2-disjoint shortest path problem (k-DPP) to AQSPP, because k-DPP is NP-hard in a directed graph even for a fixed k. In an undirected graph k-DPP is polynomial if k is fixed [43]. Thus, the proof can not be transferred directly to the undirected case, but obviously this does not conclude anything about NP-hardness.

Even though it has been shown that AQSPP is NP-hard and a line graph is only sufficient in a DAG, the line graph has still been used in applications, even in the context of power lines. Caldwell [17] already proposed such "pseudo-dual graphs" for turn constraints in 1961. In the context of infrastructure and transportation networks, the line graph is used to express turn constraints in [8] and picked up by Winter [106] who summarize approaches and applications in this context. For example, in road networks only a subset of turns is allowed at each intersection. Lately, the line graph has even been used in the context of power transmission. Santos et al. [87] build up on [77] and construct the line graph in a pre-selected corridor.

However, a major drawback of line-graph constructions is that the size of the graph is strongly increased: An undirected graph with n vertices $v_1, \dots, v_n$ and degrees $d_i = d(v_i)\ \forall i = 1..n$ leads to $\sum_{i=1}^{n} \frac{d_i(d_i-1)}{2}$ edges in the line graph. Less memory-consuming representations have been found in work on turn constraints in road networks [25], where Dijkstra is modified to consider turn-tables associated with each vertex, or in robotics [86]. Here, we propose a similar approach as the latter [86], modifying the Bellman-Ford algorithm to operate on an implicit line graph that requires no more space than the original graph. We also significantly decrease the number of operations with an efficient update algorithm.

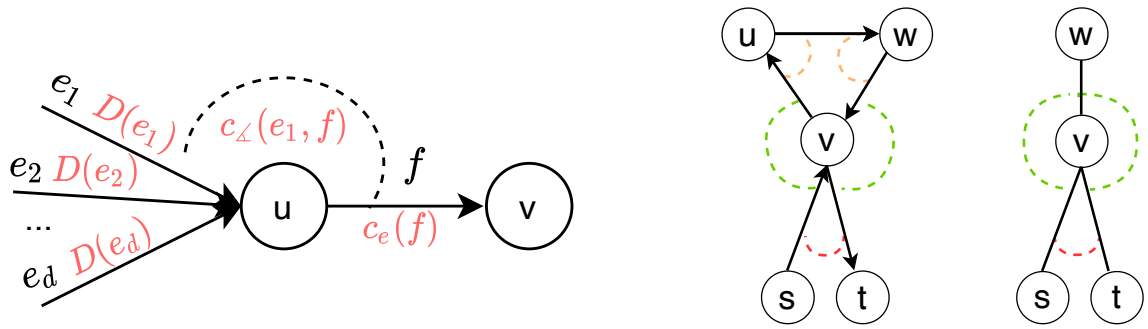
4.2 Implicit line graph and minimal-angle Bellman-Ford

We propose a variant of the Bellman Ford algorithm, where the distances from the source vertex are updated in a dynamic program, with the crucial difference that in contrast to the BF algorithm not the distance to each vertex is updated, but to each *edge*. A cost function $c(e) : E \longrightarrow \mathbb{R}$ describes the weight (cost) on each edge, as introduced in 2.2. Furthermore, a cost function for angle costs $c_a : [0, 180] \longrightarrow \mathbb{R}$ is defined. Note that c_a can in principle be any function on a relation of two adjacent edges e, f; i.e. the algorithm described in the following can be employed for any di-edge-constraint and is not restricted to angles. Furthermore, let D denote a distance map containing the costs from a single source to each edge, i.e. $D[e]$ is the accumulated cost for the (current) shortest path from the source s to the edge e. D is initialized to infinity for all edges except for the outgoing edges $e = (s, -)$ of the source s which are set to their respective edge cost, i.e. $D[e] = c(e) \quad \forall e = (s, -)$. Note that the angle at the source vertex is zero. In addition to the distance map D, a predecessor map P is defined that stores the optimal predecessor edge for each edge. If $P[e] = f$ then the shortest path from s to e contains f and f is adjacent to e.

Algorithm 1 outlines the proposed procedure to find an angle-penalized least cost *walk* on graph G with source s. Importantly, with a general angle cost function it is possible that detours or cycles appear on the output path, since problematic angles are avoided with a cycle (see Figure 17). It will be proven in subsubsection 4.2.1 below that the statement can be extended to shortest *paths* with certain requirements on c_a.

We define p to denote the maximum length of the shortest walk. It holds that $p \leq |V|$, but in many cases prior knowledge on the instance can be used to require p to be much smaller, leading to less iterations. The problem setting at each vertex is visualized in Figure 17a: in an update step, an edge must be assigned an optimal predecessor based on the distance of the predecessor edge from the source and the angle between both edges. In p iteration, all m edges are updated by taking the minimum over the distances D of their possibly preceding edges in addition to the respective angle cost. Formally, the updated cost $D[f]$ of each outgoing edge $f = (v_k, -)$ is

$$D[f] = \min_{i:e_i \in E,\ e_i=(-,v_k)} D[e_i] + c[f] + c_a(\langle e_i, f \rangle) \ .$$



(a) Angle-penalized Bellman Ford **(b)** Cycle **(c)** Detour

Figure 17: a) depicts the problem setting, where the distance D from the source vertex is given for each incoming edge, and $D[f]$ is updated as the minimum sum of distance and angle cost $c_\measuredangle$ over all incoming edges. Dependent on the cost function, the least cost solution might contain cycles (b) or even detours (c), since high angle costs can be avoided this way. For example, assume the cost of red angles are infinity, the one of a green angle is 0 and for orange angles it is 1. Then in b) the cycle with cost 2 will be used to avoid the infinity cost angle, or in c) the edge (v, w) as an outgoing edge first and then the edge (v, t) is updated with (w, v) as its predecessor.

Algorithm 1: Bellman-Ford algorithm with angle constraints

Input: G, e_s, s, c, c_a
// Initialize
$D := \infty^m$;
$D[e_s] = 0$;
$P[e] = e \quad \forall e \in E$;
// Run for sufficiently many iterations
for p *iterations* **do**
  for $i \leftarrow 0$ **to** m **do**
    // get next edge
    $(v_k, -) \leftarrow e_i$;
    // Update distance and predecessor map for e_i
    $D[e_i] = \min_{f:f\in E,\ f=(-,v_k)} D[f] + c[e_i] + c_a(\langle f, e_i\rangle)$;
    $P[e_i] = \operatorname{argmin}_{f:f\in E,\ f=(-,v_k)} D[f] + c[e_i] + c_a(\langle f, e_i\rangle)$
  end
end

Algorithm 1 formalizes the complete dynamic program. In the following, it is shown that the guarantees of correctness for the Bellman-Ford algorithm still hold for the proposed variant.

Theorem 4.1. *Let $e_t = (-, v_t)$ be any edge in G. Furthermore, let P^* be a least cost walk of length $|P^*| \leq p$ from s to v_t via e_t (if it exists), and denote the edges on P^* as $P^* = \{e_1, ...e_t\}$ $(t \leq p)$.*

After p update iterations in algorithm 1,

$$D[e_t] = c(e_1) + \sum_{i=2}^{t} c(e_i) + c_a(\langle e_{i-1}, e_i\rangle) := c(P^*)$$

and $c(P^*)$ is the cost of the shortest minimal-angle walk from s to v_t with $|P^*| \leq p$.

Proof. Proof by induction on p. As the base case, any path of length 1 from the source is given by the outgoing edges of s, which are already updated before the algorithm. Specifically, their

distance is set to $D[e] = c(e)$ which correctly corresponds to $c(P^*)$ for paths of length 1.

The induction step follows the proof of the normal Bellman-Ford algorithm. Consider an optimal shortest walk P^* from s to a target edge $e_t = (u, v_t)$, such that $c(P^*)$ is minimal. Let f^* be the edge preceding e_t on P^*, so $f^* = (-, u)$. Then $P^* \setminus e_t$ is a shortest walk from s to f^* of length $p-1$. Thus, by induction, after $p-1$ iterations we have $D[f^*] = c(P^* \setminus e_t)$. More general, for all incoming edges of u, defined as $f_j : f_j \in E, f_j = (-, u)$, $D[f_j]$ contains the overall cost of the shortest walk P_j from s to f_j with $|P_j| \leq p-1$.

In iteration p, $D[e_t]$ is updated as $D[e_t] = \min_j D[f_j] + c[e_t] + c_a(\langle f_j, e_t \rangle)$. Assume for sake of contradiction that the minimum is not attained at f^*, but at a different $\hat{f}$. Then there would be a walk $\hat{P}$ that is different from P^* because it does not pass through f^*, and with cost $c(\hat{P}) = D[\hat{f}] + c[e_t] + c_a(\langle \hat{f}, e_t \rangle)$, because as argued above $D[\hat{f}]$ is the the cost of the shortest walk to $\hat{f}$ by induction. However, this would mean that $\hat{P} = (\hat{e_1}, \dots \hat{e}_{t-2}, \hat{f}, e_t)$ has lower costs than P^*:

$$
\begin{aligned}
& c(\hat{P}) \\
& = c(e_1) + \sum_{i=2}^{t-2} \Big(c(\hat{e}_i) + c_a(\langle \hat{e_{i-1}}, \hat{e}_i \rangle) \Big) + c(\hat{f}) + c_a(\langle \hat{f}, e_t \rangle) + c(e_t) \\
& \underset{Ind.}{=} D[\hat{f}] + c_a(\langle \hat{f}, e_t \rangle) + c(e_t) \\
& < D[f^*] + c_a(\langle f^*, e_t \rangle) + c(e_t) \\
& \underset{Ind.}{=} c(e_1) + \sum_{i=2}^{t-2} \Big(c(e_i) + c_a(\langle e_{i-1}, e_i \rangle) \Big) + c(f^*) + c_a(\langle f^*, e_t \rangle) + c(e_t) \\
& = c(P^*)
\end{aligned}
$$

In conclusion, f^* is correctly assigned as the predecessor of e_t and the distances is updated as

$$D[e_t] = D[f^*] + c[e_t] + c_a(\langle f^*, e_t \rangle) = c(P^*)$$

□

The main advantage of using Algorithm 1 instead of constructing the line graph lies in the space-efficiency. Note that a shortest path cannot be computed efficiently due to the NP-hardness of the AQSPP as shown in [53]. However, a shortest path computation in the line graph would indeed yield an optimal walk. Since the minimum operation in algorithm 1 leads to the same runtime as shortest path finding in the line graph, the algorithm can be seen as an implicit line graph that provides the same functionality at significantly reduced memory.
In the following, let $\delta^+(v)$ denote the number of outgoing edges of v and $\delta^-(v)$ the incoming edges in a directed graph.

Theorem 4.2. *Algorithm 1 runs in time $\mathcal{O}\big(p \cdot \sum_v \delta^-(v) \cdot \delta^+(v)\big)$ with space requirements of $\mathcal{O}(m)$. For a d-regular graph, the runtime is equivalently $\mathcal{O}\big(pmd\big)$.*

Proof. The proof is trivial from the definition of Algorithm 1. Only the graph and the two mappings D and P need to be stored ($\mathcal{O}\big(m\big)$), and the runtime amounts to the two for-loops (iterating p times over all edges in $\mathcal{O}(pm) = \mathcal{O}(p \cdot \sum_v \delta^+(v))$) and taking the minimum. Since the minimum over the incoming edges is taken for each outgoing edge of a vertex, the update runtime for one pass over all edges amounts to $\mathcal{O}\big(\sum_v \delta^-(v) \cdot \delta^+(v)\big)$. □

4.2.1 Minimal-angle shortest paths

As mentioned above, even with non-negative edge costs algorithm 1 can output *walks* with detours or cycles. Examples are shown in Figure 17: In an undirected graph, algorithm 1 can even lead to detours as in 17c, where the edge towards v is updated and then considered as an incoming edge to update the edge (v, t). This can be prevented by simply transforming the graph into a directed graph, where each edge is doubled, but nevertheless case Figure 17b is still possible. Consider for example a graph where all normal edge costs are zero, and the angle cost is 1 only for angles above 60 degrees and zero otherwise. An optimal path would describe a large circle, circumventing the high angle cost of the direct path (Figure 17b).

However, with certain requirements on the angle cost function it can be argued that the algorithm always yields a path and does not lead to cycles or detours.

Lemma 4.3. *Let $c_a : [0, 180] \longrightarrow \mathbb{R}^+$ be a concave cost function on the angle between a pair of adjacent arcs in a digraph G. Then algorithm 1 computes the shortest s-t-path as long as c does not lead to negative cost cycles.*

Proof. Since theorem 4.1 proves that the output of algorithm 1 is a shortest walk, it only remains to prove that it is a path and not a walk. In the following, directed edges are viewed as vectors in a 2-dimensional Euclidean space.

As a first step, we argue that the angles encountered along a cycle are always larger or equal to 180 degrees. It is sufficient to consider simple non-intersection polygons, because a detour (a "cycle" of only two edges) forms exactly a 180-degrees angle in a digraph, and an intersecting cycle just adds additional angle costs along the second cycle.

According to the Closed-Path theorem formulated by Abelson and DiSessa [1], "A total turning along any closed path is an integer multiple of 360°", which directly applies to a cycle in a digraph. However, the angle at the entry vertex of the cycle is not used, as can be seen in Figure 17, where the angle at vertex v of the triangle does not count to the total angle costs. Nevertheless, this single angle must for sure be less than 180°, leaving more than 180° to the other angles on the cycle. It thus follows from the Closed-Path theorem that with $x_1, \ldots, x_n$ denoting all angles encountered on the cycle,

$$\forall i : \; x_i \in [0, 180] \text{ and } \sum_i x_i \geq 180 \tag{4.1}$$

Let α be the angle at the entrance vertex v which would be avoided with the cycle or detour via the angles $x_1, \ldots, x_n$ (e.g. α is the red angle in Figure 17b). Note that $\alpha \leq 180$. To conclude the proof, it remains to show that the angle costs on the cycle are larger than $c_a(\alpha)$, i.e. $\sum_i c_a(x_i) \geq c_a(\alpha)$ with a concave angle cost function mapping to $\mathbb{R}^+$ as defined above.
For a concave function with $c_a(0) \geq 0$ it holds that

$$\forall \lambda \in [0, 1] : \; c_a(\lambda x) = c_a(\lambda x + (1 - \lambda) \cdot 0) \underset{concave}{\geq} \lambda c_a(x) + (1 - \lambda) c_a(0) \geq \lambda c_a(x) \tag{4.2}$$

This result finalizes the proof that the sum of angle costs encountered on the cycle is larger than the cost for the simple angle of the path:

$$\sum_i c_a(x_i) = \sum_i c_a(180 \cdot \frac{x_i}{180}) \underset{4.2}{\geq} \sum_i \frac{x_i}{180} \cdot c_a(180) \underset{\sum_i x_i \geq 180}{\geq} c_a(180) \geq c_a(\alpha) \tag{4.3}$$

□

Consequently, the angle costs encountered on the cycle are larger than the single angle at the entry vertex v which would be avoided by the cycle. Note that the proof is not restricted to monotonically increasing functions. If the function is decreasing, larger angles are preferable and thus detours and cycles would not appear since they only add additional higher angle costs.
To complete the argument, it is clear that the non-angle edge costs $c(e)$ are equal or larger on a walk compared to the corresponding path, since no edge of the path is avoided by taking a detour. Thus, as long as there are no negative cost cycles, the dynamic program would always yield a path, due to additional edge costs and larger angle costs on the cycle compared to the direct route.

4.2.2 Directed acyclic graphs

Lemma 4.4. *Applied on a directed acyclic graph, the runtime of minimal-angle Bellman-Ford reduces to $\mathcal{O}(md)$ for a d-regular graph.*

Proof. The proof is given by Algorithm 2: Topological sorting of the vertices is possible in linear time $\mathcal{O}(m)$, and in the iterations of Algorithm 2 every edge is updated only once. In each update operation, again the minimum over the incoming edges is taken, leading to a runtime of $\mathcal{O}(md)$. □

Algorithm 2: Angle constrained shortest path algorithm in a directed acyclic graph

```
// Sort
V* = topological_sort(V);
// Initialize
D := ∞^m;
D[e] = c(e) ∀e = (s, −);
P[e] = e  ∀e ∈ E;
for v in V* do
    for i : e_i ∈ E, e_i = (v, −) do
        D[e_i] = min_{f:f∈E, f=(−,v)} D[f] + c[e_i] + c_a(f, e_i);
        P[e_i] = argmin_{f:f∈E, f=(−,v)} D[f] + c[e_i] + c_a(f, e_i)
    end
end
```

4.3 An accelerated update algorithm for convex angle cost functions

So far, in minimal-angle Bellman-Ford the edge distances $D[e]$ were updated by taking the minimum distance plus the angle cost over the incoming edges. At a vertex with k incoming edges and l outgoing edges $k \cdot l$ operations must be executed. Note that these operations appear

regardless of the type of graph (DAG, directed or undirected). Here, we propose a procedure that decreases the number of update operations significantly. First, we developed an algorithm that is applicable for *linear* angle cost functions and reduces the number of operations to $\mathcal{O}\Big(pm \log d^2\Big)$ steps in a d-regular graph. The algorithm and proof can be found in appendix B. However, since the requirement of linearity seems quite restrictive and also does not resemble the situation in our application where the angles are constrained, we worked on a generalized version and were able to define an algorithm that is efficient for any **convex** increasing angle cost function. The algorithm will be presented in the following.

Theorem 4.5. *Let $c_a : [0, 180] \longrightarrow \mathbb{R}^+$ be a convex and monotonically increasing angle cost function. Then the runtime of algorithm 1 reduces to $\mathcal{O}\Big(p \cdot \sum_v \big(\delta^-(v)+\delta^+(v)\big) \log\big(\delta^+(v)\delta^-(v)\big)\Big)$, since at each vertex with k incoming and l outgoing edges only $\mathcal{O}\big((k+l)\log kl\big)$ operations are executed.*
This leads to a runtime of $\mathcal{O}\Big(pm \log d^2\Big)$ in a d-regular graph.

We denote the incoming edges by the set $E^- = \{e_1^-, \dots, e_k^-\}$ and the outgoing edges by $E^+ = \{e_1^+, \dots, e_l^+\}$. Note that if the graph is undirected, then it is simply transformed into a directed graph, such that each original edge appears once in E^- and once in E^+, and $k = l$. Thus, from now on a directed graph is assumed. Furthermore, each incoming/outgoing edge reaches/leaves the vertex v in a certain angle. This angle is computed for all edges with respect to an arbitrary direction $\vec{w}$ yielding values $\alpha_i = \sphericalangle(\vec{w}, e_i^-)$ for $e \in E^-$ and $\beta_j = \sphericalangle(\vec{w}, e_j^+)$ for $e \in E^+$. Crucially, we define the angle $\sphericalangle(\alpha)$ as the angle in clockwise direction from 0 to 360 degrees. Only in this setting it holds that $\alpha_i = \beta_j$ if and only if e_i^- and e_j^+ form a straight line (e_j^+ leaves v in the same angle as e_i^- reaches it. With this notation, the aim to find the optimal predecessor in terms of angle and distances for each outgoing edge can be formulated as

$$\forall j: \quad \text{Find } \operatorname*{argmin}_i D[e_i^-] + c_a(|\alpha_i - \beta_j|_m) \ , \quad |x|_m = \min\{|x|, 360 - |x|\} \ .$$

The operation $|x|_m$ computes the smaller angle between both and thereby ensures that the input of c_a lies within its domain $[0, 180]$.

The value $D[e_i^-] + c_a(|\alpha_i - \beta_j|_m)$ can be seen as a function associated to the incoming edge i and taking an angle β_j as input. We denote these functions by f_i such that $f_i(x) = D[e_i^-] + c_a(|\alpha_i - x|_m)$. Observe that each function is a shifted version of $c_a(|x|_m)$, displaced by α_i in x- and $D[e_i^-]$ in y-direction. Figure 18 shows an example, where each incoming edge induces a cost function for the angles of possible outgoing functions. Note that f_i is symmetric if the x-axis is rotated such that $\alpha_i = 180$ and it has a global minimum at α_i because c_a is monotonically increasing. Also, if $\alpha_i = 180$ then f_i is convex because it is a (shifted) composition of two convex functions, c_a and $|x|$. The key observation used in the following algorithm is that due to convexity an incoming edge is the optimal predecessor for a *closed* range of angles (without gaps) and thus neighboring outgoing edges. To determine the "area of responsibility", i.e. the range of angles in which an outgoing edge would get e_i^- assigned as its predecessor (Figure 18 bottom left), the intersections of the functions f_i are computed. The idea is to construct a tree T_x containing the intersection values (Figure 18 right) that can be traversed in $\mathcal{O}(\log k)$ in the end for each outgoing edge to find the closest intersection and thereby its optimal predecessor edge.

First, the incoming edges are sorted by their distance from the source $D[e_i^-]$. For simplicity of notation, we from now on assume that they are already sorted, such that $D[e_1^-] \leq D[e_2^-] \leq \cdots \leq D[e_k^-]$. Then, the incoming edges are successively processed and the intersection of their respective function f_i with previously responsible functions in their range is computed and added to a balanced AVL tree T_x. algorithm 4 describes the procedure in detail. The

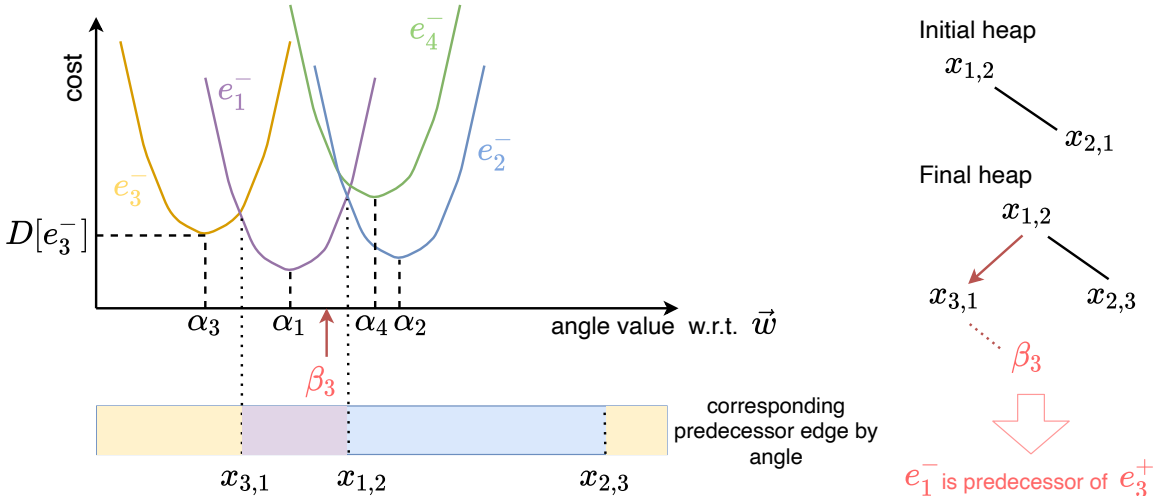


Figure 18: Angle update algorithm for convex, monotonically increasing angle cost functions. The intuition is that the intersections of neighboring functions define the borders of the "area of responsibility"for each incoming edge. Intersections are iteratively added to a tree, starting with the incoming edge with lowest base cost D. The tree is sorted and balanced. After the tree is fully constructed, each outgoing edge e_j^+ is assigned its optimal predecessor by finding the closest value to β_j in the tree.

tree is built iteratively by insertion and replacement of intersection points. We define an operation `compute-intersection` that outputs the intersection of two functions. Given f_p, f_q, `compute-intersection`(f_p, f_q) yields a x value between α_p and α_q at which the functions intersect. Crucially, the angle values are circular, i.e. an angle of 0 degrees corresponds to an angle of 360 degrees. Therefore, `compute-intersection`$(f_p, f_q) \neq$`compute-intersection`(f_q, f_p) because if the two functions intersect at any point x, then they intersect at least at two points, x and $360 - x$. Thus, for each intersection we store which function was lower in clockwise (positive x-) direction and which one in counter-clockwise (negative x-) direction. For example, a triple (x, p, q) means that e_p^- is the optimal predecessor for β-angles that are close to x in counter-clockwise direction and e_q^- is optimal for outgoing edges close to x in clockwise direction. Due to the circularity it is for example possible that $\alpha_p = 300$, $\alpha_q = 10$ and $x = 330$. It will be argued further below that `compute-intersection` can be approximated efficiently.

Secondly, algorithm 4 uses an operation called `find-closest` that returns the two closest value in a balanced AVL-tree. The algorithm is outlined in detail in algorithm 3. The balanced and sorted tree is traversed and the closest left and right values σ_l, σ_r with respect to the input element are memorized and returned.
Last, in addition to the sorted tree of intersections T_x we also maintain a double-linked list L of neighboring intersections. If (x, p, q) is the intersection triple of f_p and f_q, then (x, p, q).next is defined as the next intersection in clockwise direction which must therefore be of the form (y, q, r) with some r. Similarly, (x, p, q).previous is of the form (x, o, p) and denotes the next intersection counter-clockwise. In the main algorithm detailed in algorithm 4, for each new incoming edge e_i^- the closest angle values of incoming edges that were already processed are found with `find-closest`. To do this efficiently in logarithmic time, all α_i are iteratively inserted into a second tree T_α and `find-closest`(T_α, α_i) is called. Let the closest angles be denoted by α_p and α_q to α_i.

Algorithm 3: `find-closest`

```
Input: AVL tree T element x
// define σ_l and σ_r as the currently closest elements
σ_l, σ_r = ∞
n = T.root()
// Traverse tree
while not isLeaf(n) do
    if x > n.key() then
        σ_l = n
        n = getRightChild(n)
    else
        σ_r = n
        n = getLeftChild(n)
end
// check border cases if one is still infinity
if σ_l is ∞ then
    // set left bound to the largest element in the tree
    σ_l = T[−1]
if σ_r is ∞ then
    // set right bound to the smallest element in the tree
    σ_r = T[0]
return σ_l, σ_r
```

If the new function f_i obtains a better cost at the intersections between f_p and f_q, the triple is deleted from T_x and the next triples in both directions are checked. This is repeated until encountering an intersection point where f_i can not improve on the previous functions. Due to convexity, once it is outperformed by another function it will not be lower again in this direction. Once these functions that mark the end of the area of responsibility of edge i have been determined, the intersections with them are computed and added to T_x. Also, the double-linked list is updated accordingly.
After all incoming edges were considered and possibly added to T_x, the outgoing edges are processed, i.e. for β_j its left and right neighboring intersections triples $(x_1, -, i)$ and $(x_2, i, -)$ are found with a call to `find-closest` and the corresponding incoming edge e_i^- is assigned as the predecessor of e_j^+.

Consider the example in Figure 18. First of all, the first function must be found that intersects with f_1 (PART 1 in algorithm 4). Clearly, $f_1(\alpha_1) \leq f_i(\alpha_i)\ \forall i$, so it must be checked whether f_2 is better at any other point, formally whether $\exists x : f_1(x) > f_2(x)$. Lemma 4.6 will show that it is sufficient to check whether f_2 is lower at the opposite point of α_1, namely $(\alpha_1 + 180) \bmod 360$, where f_1 is maximal. Since f_2 satisfies this condition, the two intersection (clockwise and counter-clockwise from α_1) between f_1 and f_2 are computed and added to the tree as shown in Figure 18 (upper right). T_x is sorted and balanced. In the first iteration of PART 2 in algorithm 4, α_3 is considered and its neighboring $alpha_i$ are determined with `find-closest`. In the example, α_3 is between α_2 counter-clockwise and α_1 clockwise It is therefore checked in algorithm 4 whether f_3 improves on the value at the intersection $x_{2,1}$. Here, indeed $f_3(x_{2,1}) < f_1(x_{2,1})$ and therefore the triple $(x_{2,1}, 2, 1)$ is deleted from T_x. In the next step, the new intersections with f_2 as the counter-clockwise better function and f_1 as the clockwise better function is computed and $(x_{3,1}, 3, 1)$ and $(x_{2,3}, 2, 3)$ are inserted into T_x. On the other hand, in the last iteration f_4 does not outperform the value at the intersection between f_1 and f_2 and consequently α_4 is not an optimal predecessor for any outgoing angle. Finally, the β values of the outgoing edges are processed. In Figure 18 for example β_3 is between the intersection triples $(x_{3,1}, 3, 1)$ and $(x_{1,2}, 1, 2)$ and thus e_1^- is the

optimal predecessor for e_3^+.

Algorithm 4: Efficient angle cost update algorithm for convex angle cost functions

```
// PART 1: Find first intersecting functions
i_start = 2
α̂_1 ⟵ (α_1 + 180) mod 360 // opposite point to α_1
while i_start ≤ k and f_{i_start}(α̂_1) > f_1(α̂_1) do
    i_start ⟵ i_start + 1
end
if i_start > k then
    // Break: no function better than f_1 at any point
    T_x.insert(α_1, 1, 1)
    return T_x
x_1 = compute-intersection(f_1, f_{i_start})
x_2 = compute-intersection(f_{i_start}, f_1)
(x_1, 1, i_start).next = (x_1, 1, i_start).previous = (x_2, i_start, 1)
(x_2, i_start, 1).next =(x_2, i_start, 1).previous = (x_1, 1, i_start)
T_x.insert((x_1, 1, i_start), (x_2, i_start, 1))
T_α.insert(α_1, α_{i_start}) // initialize T_α
// PART 2: Process further incoming edges
for i = i_start..k do
    // Get closest α values
    α_p, α_q ⟵ find-closest(T_α, α_i)
    x_{p,q} ⟵ compute-intersection(f_p, f_q)
    // Does f_i outperform the previous functions at any intersection?
    if f_i(x_{p,q}) < f_p(x_{p,q}) or f_i(x_{p,q}) < f_q(x_{p,q}) then
        T_α.insert(α_i) // Check counter-clockwise intersection
        (x_1, o, p) ⟵ (x_{p,q}, p, q).previous
        while f_i(x_1) < f_p(x_1) do
            // delete and get next counter-clockwise
            T_x.delete((x_1, o, p))
            T_α.delete(α_p)
            (x_1, o, p) ⟵ (x_1, o, p).previous
        end
        p* ⟵ p
        // Check clockwise intersection
        (x_2, q, r) ⟵ (x_{p,q}, p, q).next
        while f_i(x_2) < f_q(x_2) do
            // delete and get next clockwise
            T_x.delete((x_2, q, r))
            T_α.delete(α_q)
            (x_2, q, r) ⟵ (x_2, q, r).next
        end
        q* ⟵ q
    x_1* ⟵ compute-intersection(f_{p*}, f_i)
    x_2* ⟵ compute-intersection(f_i, f_{q*})
    T_x.insert((x_1*, p*, i),(x_2*, i, q*))
    (x_1, o, p*).next = (x_2*, i, q*).previous = (x_1*, p*, i)
    (x_1*, p*, i).next = (x_2, q*, r).previous = (x_2*, i, q*)
end
// PART 3: process outgoing edges
for j = 1..l do
    (x_1, p, q), (x_2, q, r) ⟵ find-closest(T_x, β_j)
    // update predecessor (in area of responsibility of v)
    P[e_j^+] = e_q^-
end
```

In the following, we will describe several properties of the proposed method that will be necessary to proof the correctness and runtime efficiency of algorithm 4.

Lemma 4.6. *Let f_p and f_q be functions for the incoming edges e_p^- and e_q^- as defined above, and let $D[e_p^-] \leq D[e_q^-]$ and $\alpha_p < 180$ and $\alpha_p < \alpha_q < \alpha_p + 180$, such that f_q is equal to f_p shifted in*

positive x-direction and in positive y-direction. Let $\hat{\alpha_p} = \alpha_p + 180$ *be the point on the opposite side of* α_p *on the cycle.*
If $\exists x: \ f_q(x) < f_p(x)$ *then* $\forall x^{'} \in [x, \hat{\alpha_p}]: \quad f_q(x^{'}) < f_p(x^{'})$

Proof. We will use a property of convex monotonically increasing functions that can be seen as an increasing returns property:

Claim 4.7. *For a convex function* g *and* $a > b$, $a, b, c \geq 0$ *it holds that*

$$g(a+c) - g(a) \geq g(b+c) - g(b) \tag{4.4}$$

Proof. The intuition is that if g is differentiable its gradients are increasing and thus the difference in value becomes larger. To prove the claim, we use another well-known property of convex functions, namely if f is convex and $f(0) \leq 0$ then

$$\forall \ d, c \geq 0: \quad f(c) + f(d) \leq f(c, d) \tag{4.5}$$

We define $f(x) = g(x+b) - g(b)$. Since f is only a shifted version of g, f is convex as well and also note that $f(0) = g(b) - g(b) = 0$. We further set $d = a - b > 0$ and plug in d in property 4.5 to yield

$$\begin{aligned} & f(d) + f(c) \leq f(d+c) \\ \iff & f(a-b) + f(c) \leq f(a-b+c) \\ \iff & g(a-b+b) - g(b) + g(c+b) - g(b) \leq g(a-b+c+b) - g(b) \\ \iff & g(a) - g(b) + g(b+c) \leq g(a+c) \end{aligned}$$

which is equivalent to the claim. □

To continue the proof of lemma 4.6 we consider the point x where $f_q(x) < f_p(x)$. Observe that due to $D[e_p^-] \leq D[e_q^-]$ and with c_a monotonically increasing,

$$\begin{aligned} & f_q(x) < f_p(x) \\ \implies & D[e_q^-] + c_a(|x - \alpha_q|_m) < D[e_p^-] + c_a(|\alpha_p - x|_m) \\ \implies & |x - \alpha_q|_m < |\alpha_p - x|_m \end{aligned}$$

Let $x^{'}$ be any other point between x and the opposite point of α_p as defined above ($x^{'} \in [x, \hat{\alpha_p}]$). Since c_a is monotonically increasing, f_p is maximal for $\hat{\alpha_p}$. We use the claim show that $f_q(x^{'}) < f_p(x^{'})$ by setting $a = |x - \alpha_p|_m$ and $b = |x - \alpha_q|_m$ (leading to $a > b$), and further setting $c = |x - x^{'}|_m$:

$$
\begin{aligned}
& f_q(x') \\
&= D[e_q^-] + c_a(|x' - \alpha_q|_m) \\
&\underset{*}{\leq} D[e_q^-] + c_a(|x' - x|_m + |x - \alpha_q|_m) \\
&= D[e_q^-] + c_a(c + b) \\
&\underset{4.4}{\leq} D[e_q^-] + c_a(c + a) - c_a(a) + c_a(b) \\
&= D[e_q^-] + c_a\big(|x' - x|_m + |x - \alpha_p|_m\big) - c_a\big(|x - \alpha_p|_m\big) + c_a\big(|x - \alpha_q|_m\big) \\
&\underset{**}{=} D[e_q^-] + c_a\big(|x' - \alpha_p|_m\big) - c_a\big(|x - \alpha_p|_m\big) + c_a\big(|x - \alpha_q|_m\big) \\
&= f_q(x) + c_a\big(|x' - \alpha_p|_m\big) - c_a\big(|x - \alpha_p|_m\big) \\
&\underset{f_q(x)<f_p(x)}{<} D[e_p^-] + c_a\big(|x - \alpha_p|_m\big) + c_a\big(|x' - \alpha_p|_m\big) - c_a\big(|x - \alpha_p|_m\big) \\
&= f_p(x')
\end{aligned}
$$

(*) follows from the triangle inequality that also holds in the circular case. (**) is based on the condition that $x' \in [x, \hat{\alpha_p}]$.
In short, due to the property of increasing returns, if f_q is lower than f_p at any point it will remain lower until intersecting with f_p from the other side and thus at least until $\hat{\alpha_p}$. □

In Lemma 4.6 it was assumed that $\alpha_p < \alpha_q < \alpha_p + 180$ which might seem very restrictive. The assumption was made to reduce complexity in the proof, but the statement also holds for the circular case. To see this, observe that given two functions f_p and f_q, the x-axis can be rotated and flipped to fulfill the above condition. Since it is circular, this operation is always possible and does not change the shown property. The same holds for the following observation:

Lemma 4.8. *Let α_i refer to an angle of an incoming edge between α_p and α_q, such that in the simplified non-circular model it can be assumed that $\alpha_p < \alpha_i < \alpha_q$. Assume also that $D[e_i^-] \geq D[e_p^-]$ and $D[e_i^-] \geq D[e_q^-]$. Let $x_{p,q}$ denote the intersection of f_p and f_q.*
If $f_i(x_{p,q}) > f_p(x_{p,q}) = f_q(x_{p,q})$ then $\nexists x : f_i(x) < f_p(x) \wedge f_i(x) < f_q(x)$.

Proof. We again use an observation that was made in Lemma 4.6: If $D[e_i^-] \geq D[e_p^-]$, then $f_i(x) < f_p(x)$ implies $|x - \alpha_i|_m < |\alpha_p - x|_m$ and the same holds for q. Thus, if there exists an x where $f_i(x) < f_p(x)$ and $f_i(x) < f_q(x)$ then it follows that $\alpha_p < x < \alpha_q$. Assume $\alpha_p < x \leq \alpha_i < \alpha_q$ (the other case is analogous). Due to Lemma 4.6 if $f_i(x) < f_p(x)$ at x then it remains lower until the opposite point $\hat{\alpha_p}$. Since the intersection $x_{p,q}$ is at $\hat{\alpha_p}$ at the latest (otherwise f_q would not be better then f_p anywhere), it can be concluded that f_i outperforms f_p also at the intersection. □

Theorem 4.9. *algorithm 4 assigns the correct predecessor to each outgoing edge.*

Proof. Let e_i^- be the optimal predecessor for an outgoing edge j^*. In the first loop of algorithm 4 an incoming edge e_i^- is only skipped if its value at $\hat{\alpha_1}$ is higher than for e_1. By Lemma 4.6, if the value is not better than f_1 at $\hat{\alpha_1}$, then it is larger everywhere. Consequently, any incoming edge that is skipped in the first loop of algorithm 4 is outperformed by e_1^- and would thus not be the unique optimal predecessor of any outgoing edge.
Any higher i that is not skipped in the PART 1 will be considered at some point in PART 2 in algorithm 4. Its closest angles α_p and α_q are retrieved. Due to Lemma 4.8 it suffices to check

whether f_i is better at the intersection of f_p and f_q, otherwise it can be discarded. Lemma 4.8 can be applied because if α_p and α_q were retrieved from T_α, then they have already been processed earlier than α_i and consequently their distance D must be lower (T_α is empty initially).

Furthermore, if f_i is indeed lower at the intersection, then there might be other intersections where f_i outperforms previous incoming edges. Note that any of such points must lie between α_p and α_q. algorithm 4 takes care of such special cases by checking the neighboring intersections in both directions. If at some point it is outperformed by another function, then it can not become better at any further point due to convexity. This procedure thus defines the current area of responsibility for edge e_i^-.

With further iterations, this area can be reduced in size or removed. Due to the conditions in algorithm 4, it is only removed if a new incoming edge indexed with $\hat{i}$ obtains lower values at both intersections of f_i with its neighboring functions. Then algorithm 4 would correctly compute the new intersections of $f_{\hat{i}}$ with f_p and f_q which replace the intersections of f_i. It would thus not be possible that f_i is the optimal predecessor of any outgoing edge, because it was previously outperformed behind the borders of its area of responsibility by the respective intersecting functions, and now it is also outperformed in the area inbetween.

In the last for-loop (PART 3) in algorithm 4 concerning the outgoing edges, the closest intersections (x_1, u, v) and (x_2, v, w) to β_{j^*} are retrieved from T_x. Note that by the procedure of algorithm 4 it is ensured that `find-closest` always yields triples of this form where v appears in both triples and $(x_1, u, v).\text{next} = (x_2, v, w)$ and $(x_2, v, w).\text{previous} = (x_1, u, v)$. To see this, observe that the while-loops in algorithm 4 stops at two triples (x_2, q^*, r) and (x_1, o, p^*). Since all intersections inbetween were deleted, it is correct that (x_2, q^*, r) is assigned the new intersection with of f_i with f_{q^*} as the previous triple and equivalently (x_1, o, p^*) is assigned the new intersection of f_i with f_{p^*}. □

Lemma 4.10. *The `compute-intersection` operation can be approximated efficiently and sufficiently for algorithm 4.*

Proof. While the efficiency to compute an intersection between a function and its shifted version depends on the specific function, it is actually not necessary here to know the exact intersection. Crucially, in the end we are only interested in the values $\beta_1, \dots, \beta_l$, so it suffices to evaluate the functions at these values. Given two functions f_p, f_q and their theoretical intersection $x_{p,q}$, we just aim to find $j^* = \text{argmin}_j |x_{p,q} - \beta_j|$. For this purpose a similar procedure as the `find-closest` function (algorithm 3) is utilized. First, all β_j ($j \in [1..l]$) are inserted in an AVL tree T ($\mathcal{O}(l \log l)$). For each call of `compute-intersection`, the tree is traversed in the following manner: Starting at the root node, f_p and f_q are evaluated at the β value stored in the node. If $f_p(\beta) > f_q(\beta)$, the left child node is selected, otherwise the right child node. The intuition is that if $f_p(\beta) > f_q(\beta)$ then the intersection must be at a smaller β value where f_p takes lower values. During traversal of the tree, the optimal node with respect to the distance $|f_p(\beta) - f_q(\beta)|$ and its distance itself are memorized such that the closest β is determined once a leaf node is reached.

However, one might object that the border case of the circularity is problematic in this case. Indeed, it might happen that the rightmost leaf in the tree is reached but the optimal β value (the one closest to the intersection) is not the leftmost one. To prevent this case, we propose to build two trees with all β_j inserted: One sorted and balanced tree with all β_j, and one with all $\hat{\beta}_j$ where $\hat{\beta}_j = (\beta_j + 180 \bmod 360$. After both trees have been traversed, the value that is closer to the intersection, i.e. where $|f_p(\beta) - f_q(\beta)|$ is lower, is selected. This ensures that the closest β value is found, as it is not a border case in at least one of the trees.

We claim further that the discrete evaluations do not impact the proof of Theorem 4.9. Assume that $\alpha_p \leq \beta_{p,q} \leq x_{p,q}$ (the other side is again analogous). Note that there can not exist any $\hat{\beta}$ that satisfies $\alpha_p \leq \beta_{p,q} \leq \hat{\beta} \leq x_{p,q}$ because then $\hat{\beta}$ would have been selected as the intersection between f_p and f_q.
On the other hand, there can be β' with $\alpha_p \leq \beta_{p,q} \leq x_{p,q} \leq \beta' \leq \alpha_q$. Crucially, it is possible that $f_i(\beta') < f_p(\beta')$ and even $f_i(x_{p,q}) < f_q(x_{p,q})$, but $f_i(\beta_{p,q}) > f_q(\beta_{p,q})$ such that the new function f_i is not better at the approximated intersection but still lower at the actual intersection. However, it has been shown in Lemma 4.6 that in this case $f_i(x) < f_q(x)\ \forall x < \beta'$. This is why in algorithm 4 it is checked whether *both* f_p and f_q outperform f_i at the approximate intersection (which would be redundant if the exact intersection was available because the values for f_p and f_q would be equal). Thus, if β' with the above properties exists, then $f_i(\beta_{p,q}) < f_q(\beta_{p,q})$ and therefore the algorithm would still compute the intersections of f_i with the previous functions and β' would be assigned to the correct range. □

Theorem 4.11. *The runtime of algorithm 4 is* $\mathcal{O}\big((k+l)\log kl\big)$.

Proof. First, the incoming edges are sorted by their distance to the source vertex, taking $\mathcal{O}\big(k \log k\big)$ time. As argued in lemma 4.10, the `compute-intersection` operation can be approximated here by a discrete evaluation on the β values, leading to a runtime of $\mathcal{O}\big(\log l\big)$ for each call to `compute-intersection` once the AVL-tree of β values is build ($\mathcal{O}\big(l \log l\big)$). In addition, each insert and delete operation to T_x takes $\mathcal{O}\big(\log k\big)$ since at most k elements are in the tree (in each iteration, one intersection is deleted and two intersections are added). Similarly, each insert and delete operation, as well as each call to `find-closest` with T_α requires $\mathcal{O}\big(\log k\big)$ time. Last, the `find-closest` operation again only requires a traversal of T_x in $\mathcal{O}\big(\log k\big)$ steps. Thus, each iteration in PART 2 of algorithm 4 requires $\mathcal{O}\big(\log l + \log k\big) = \mathcal{O}\big(\log kl\big)$ operations.

The number of iterations is bounded by the number of incoming edges, since in each iteration of the while-loop one intersection is deleted. The intersections are added iteratively to the tree T_x and for each iteration at most two intersections are added and at least one triple is deleted. The number of possible deletions is thus bounded by the number of elements in the tree (k). Thus, the maximum number of iterations in the while-loop sum up to at most k over all iterations, leading to a maximum of $O(k)$ iterations for the for- and the while-loop together in PART 2.

Finally, in PART 3 of algorithm 4 the closest element in T_x is found for each outgoing edge, causing an additional runtime of $\mathcal{O}\big(l \log k\big)$. Given the optimal predecessor edges, the distances of E^+ can be updated in $\mathcal{O}\big(l\big)$. Altogether, the runtime is $\mathcal{O}\big(k \log k + l \log l + k \log kl + l \log k\big) = \mathcal{O}\big((k+l)\log kl\big)$. □

In short, with a convex angle cost function the shortest *walk* can be computed efficiently: The space efficiency is $\mathcal{O}(m)$ and the runtime increases only by $\log d^2$ for a d-regular graph. With a *concave* function it is instead ensured that path does not encounter cycles, so with a linear angle cost function the shortest *path* is computed efficiently. However, if the normal costs $c(e)$ are nonzero it is unlikely that the output path uses cycles, and even then it might indeed be the optimal solution if angles shall be avoided by all means. We conclude that our algorithm provides a method that is generally applicable since it is performant for any convex increasing cost function and improves substantially over previous approaches such as the line graph.

5 Computing k diverse shortest paths

In many applications of shortest path finding it is desirable to compute not only the best path, but a set of k shortest paths. The problem has been studied for a long time, and since 1970 an approach for the efficient (exact) computation of the k shortest paths was given with Yen's algorithm [110, 109]. The intuition is to take the $k-1$-th shortest path P, and in turn remove each edge of the path and compute the new best path to the sink from the vertex incident to the removed edge. The minimum cost path out of all $|P|$ new paths is added as the k-th path. With Dijkstra's algorithm for shortest path finding the runtime is $\mathcal{O}(kn(m+n\log n))$. A more efficient method was proposed by Eppstein [31]: Instead of computing the shortest path from source to target, two shortest path trees are computed, one from the source and one from the target vertex with reversed edge directions. Summing up the two distances for each vertex yields the length of a shortest path passing through this vertex. The vertex-wise distances are then sorted ($\mathcal{O}(n\log n)$), and k paths can be constructed from the predecessor maps in the shortest path trees, enumerating vertices with increasing cost. In total, constructing the shortest path trees, sorting and computing the k best distances requires only $\mathcal{O}(m+n\log n+k)$. Note that Eppstein's algorithm might lead to loops, and Yen's algorithm is required if the k shortest loopless paths are desired. Notable improvements of Eppstein's algorithm include the work of Hershberger et al. [52] who summarize paths in equivalence classes based on shared segments, thereby achieving an $\mathcal{O}(n)$ improvement, and [5] who index the paths in a new data structure. In the context of power infrastructure the problem was approached by Zinchenko et al. [112] who propose a projection into 3D to find two diverse shortest paths.

5.1 Eppstein's algorithm for angle-cost shortest path problems

Here, we first show that Eppstein's algorithm [31] can be transferred to the implicit line graph model described above to yield an efficient algorithm to compute the k angle-shortest paths. The modified approach is shown in Algorithm 5, describing how distances are computed edge-wise instead of vertex-wise.

Algorithm 5: K-shortest path algorithm for the implicit line graph

```
// Get distance and predecessor maps from source and target vertex
D_s, P_s = Algorithm 1(G, e_s, v_s, c, c_a);
D_t, P_t =Algorithm 1(←G, e_t, v_t, c, c_a);
// Compute SP cost for each edge (c_e appears twice)
C[e] := D_s[e] + D_t[e] − c(e);
C* = sort(C);
// Initialize set of paths
S = {};
K = {};
while |K| < k do
    // Get next edge e with i-th shortest path costs
    e ← C*.next()
    if e ∉ S then
        R = compute path through e from P_s[e] and P_t[e] ;
        S = S ∪ {R};
        K = K ∪ R
end
return K
```

The accumulated path costs are computed two times, first from the source vertex, then from the target vertex with reversed edge directions ($\overleftarrow{G}$). Thereby, the two distance maps contain the distances from source and sink respectively for each vertex. Summing up the distance maps and subtracting the costs of the edge itself that were counted twice, a map C is yielded that contains for each edge the length of the shortest path from source to sink that uses edge e. C is sorted, such that paths can be enumerated from lowest to highest costs. Starting from the edge associated with the least cost path, edges are enumerated in increasing path length order. Once an edge is encountered that does not belong to any of the previous paths, the corresponding path is returned, until k paths have been found.

Lemma 5.1. *Algorithm 5 runs in* $\mathcal{O}(mpd + m\log(m) + kp)$.

Proof. Algorithm 1 takes $\mathcal{O}(mpd)$ as shown above, and is executed only twice. Then, the edge cost maps are summed up ($\mathcal{O}(m)$) and sorted by their shortest path costs which requires $\mathcal{O}(m\log(m))$. Iterating over the edges in the subsequent loop, only k times the path needs to be reconstructed from the predecessor maps ($\mathcal{O}(p)$), therefore taking $\mathcal{O}(p)$ steps in each of k iterations. □

5.2 The K diverse shortest path problem

Enumerating the k shortest paths often leads to similar paths. In the application for power infrastructure in particular it is likely that the second best path is simply the best path except for one pylon that is replaced by another one closeby. For planners of power infrastructure it would be more informative to be presented with k diverse paths that represent the shortest alternatives for different regions. A well-studied problem in graph theory is the computation of arc- or vertex-disjoint shortest paths, but it was shown that the problem is NP-hard if k is part of the input, or even for fixed k in undirected graphs [30]. However, it might not be necessary to compute entirely disjoint paths as soon as they are sufficiently diverse according to some metric. The problem was defined formally by Liu et al. [64] as the "Top-k Shortest Paths with Diversity (KSPD)" problem.

Definition 5.2. *Let* $f_s : P \times Q \longrightarrow \mathbb{R}$ *be a similarity function, and* θ *be a similarity threshold. Let* D *denote the set of all sets of dissimilar paths, i.e.* $D = \{\Psi |\ \forall P_i, P_j \in \Psi : f_s(P_i, P_j) > \theta\}$.

The KSPD problem aims to find a dissimilar set of paths $\Psi \in D$ *with* $|\Psi| \leq k$ *such that there does not exist any* $\hat{\Psi}$ *with* $|\hat{\Psi}| > |\Psi|$ *or* $|\hat{\Psi} = |\Psi|$ *and* $\sum_{P\in\hat{\Psi}} L(P) < \sum_{P\in\Psi} L(P)$, *where* $L(P)$ *denotes the total costs of path* P.

The idea is to find a set of shortest paths such that the diversity of each pair of paths is above the threshold θ. In [64], the authors also provide a proof that KSPD is NP-hard, which was however shown to be a erroneous by Chondrogiannis et al. [19] and corrected. Chondrogiannis et al. [18] had in turn proposed a different formulation for the diverse k shortest path problem called *kSPwLO*, which is basically a relaxation of *KSPD*, corresponding to a greedy approximation: *kSPwLO* requires that every new path is alternative to all paths already in the set and as short as possible. Liu et al. [64] actually provide an algorithm (*FindKSPD*) that solves *kSPwLO*. In their recent work [20], Chondrogiannis et.al. analyze *kSPwLO* further and provide an exact algorithm with optimal complexity. However, there are no approximation algorithms for *KSPD* to the best of our knowledge.

Prior to such formal definitions, the problem has been studied in applied work on the transport of hazardous material. In order to avoid imposing a risk on a single region, hazardous material

should be transported along diverse routes. In this context, Akgün et al. [4] describe possible approaches to find dissimilar paths, long before the formulations described above [64, 18]. One of their methods solves a problem that can be viewed as a reversed formulation of *KSPD*: While in *KSPD* a threshold is set on the similarity of paths and the costs are minimized, one can also optimize the dissimilarity with an upper bound on the costs. We call this formulation the BCDSP problem.

Definition 5.3. ***Bounded Cost Diverse Shortest Paths (BCDSP)***
Let θ denote a threshold on the path costs, and let $d : P_1 \times P_2 \longrightarrow \mathbb{R}$ be a path distance metric.

Then the BCDSP problem aims to find the set Ψ with $|\Psi| = k$ of the most diverse paths with cost lower than θ. Formally, the goal is to find Ψ such that $\forall\ P \in \Psi : L(P) < \theta$ and $\sum_{P_i, P_j \in \Psi} d(P_i, P_j)$ is maximal.

A motivation for solving *BCDSP* is that planners of linear infrastructure aim to minimize the costs, but the exact costs might not be inclusive since they depend on the input parameters set by the planners. Given an upper bound on the cost (that could for example be set to 5% above the cost of the best path), planners could desire to compare maximally diverse routes.

The *BCDSP* formulation turns the diverse k-shortest path problem into a *dispersion* problem which is well-studied in the literature. The dispersion problem tries to choose k points from a set of facilities, such that the minimum or average distance of each point to the other selected points is maximal (MAX-MIN or MAX-AVG formulations). Both formulations where proven to be NP-hard, for example [103] showed that MAX-MIN is NP-hard by reduction from 3-satisfiability. However, [80, 81] prove that a greedy approach starting from the two most diverse points is actually a 2-approximation for the MAX-MIN, if the distances satisfy the triangle inequality. Thus, this approximation algorithm directly provides an approximate solution to *BCDSP*, if given a suitable metric to measure path distances.

5.2.1 Path distance metrics

In the literature on k-diverse-shortest paths, the problem was usually considered in a graph setting where Euclidean distances are not relevant. For example, [64] suggest several similarity functions, including the intersection-over-union (IoU) of edges on the path, or the similarity $Sim(P, Q) = \frac{1}{2}\left(\frac{P \cap Q}{len(P)} + \frac{P \cap Q}{len(Q)}\right)$ proposed in [33]. Here, we implement the Jaccard distance (IoU) as a baseline, but compare analyticaly and experimentally to Euclidean-distance-based metrics. Together we thus focus on the following three metrics:

- **Jaccard distance:** The inversion of the IoU defined as $d_J(P, Q) = 1 - \frac{P \cap Q}{P \cup Q}$ was shown to be a metric for example by Kosub [61]. Here, we consider P and Q as sets of vertices, such that the Jaccard distance measures the intersection over union of vertices on the path.
- **Yau-Hausdorff distance:** The Yau-Hausdorff distance is defined as the maximum of the minimum distances from the vertices of the first path to the ones of the second. However, as this definition is not symmetric, the maximum of both is taken. Formally, the Yau-Hausdorff distance is defined as

$$d_Y(P, Q) = \max\{d_s(P, Q), d_s(Q, P)\}, \text{ with } d_s(P, Q) = \max_{p \in P} \min_{q \in Q} \|p - q\|$$

  Tian et al. [95] provide a proof that the Yau-Hausdorff distance is a metric.

- **Mean Euclidean distance:** Instead of using the maximum over the point-wise distances in the Yau-Hausdorff distance, one can also simply take the average:

$$d_M(P,Q) = \max\{\frac{1}{|P|}\sum_{p\in P}\min_{q\in Q}\|p-q\|, \frac{1}{|Q|}\sum_{q\in Q}\min_{p\in P}\|p-q\|\}$$

  Clearly, given that the average of Eucledian distances is a metric and that the Yau-Hausdorff distance is a metric, the triangle inequality is fulfilled in this case as well.

The choice of metric obviously effects the output significantly. With the maximum Euclidean metric d_M, the paths might mostly intersect and then diverge strongly at one point. Paths that are distant in the Jaccard sense (with d_J) could run in parallel and very close to each other.

5.2.2 Complexity of diverse path selection

Definition 5.4. *Given a graph $G = (V, E)$, a path P and a metric M, we define the **diverse path selection problem** as the problem to find an alternative path P' where the distance exceeds a similarity threshold θ, i.e. $d_M(P, \hat{P}) > \theta$.*

This problem arises in any greedy algorithm to compute a set of diverse paths: Based on $k-1$ previously found paths, an alternative path that is further than θ from the previous path is to be determined. The complexity of this problem with respect to each similarity metrics is analyzed in the following.

Lemma 5.5. *Diverse path selection according to the Yau-Hausdorff distance (the maximum Euclidean distance) is decidable in polynomial time.*

Proof. The proof is straightforward with Eppstein's algorithm. After computing both shortest path trees, one pass over all vertices is sufficient to determine the vertex of shortest assigned path length and sufficient Euclidean distance to the previous path(s) P_{prev}. If such a vertex v^* exists and has non-infinity path-distances to both source and target, the full path P^* can be computed from predecessor trees. The Yau-Hausdorff distance between P^* and P_{prev} is the maximum distance of all vertices of P^*, and thus at least as large as the distance of v^* to P_{prev}:

$$d_Y(P^*, P_{prev}) \geq \max_{p\in P^*}\min_{q\in P_{prev}}\|p-q\| \geq \min_{q\in P_{prev}}\|v^*-q\|$$

On the other hand, if no such vertex exists then clearly there cannot exist a path with sufficient Yau-Hausdorff distance either, because the maximum distance must be attained at some vertex. Thus, a shortest diverse path according to d_Y exists if and only if a vertex v^* exists with sufficient distance, and the diverse path selection problem is decided with Eppstein's algorithm. □

However, this result does not hold for the Jaccard distance d_J and the mean Euclidean distance d_M.

Lemma 5.6. *With the Jaccard distance or the mean Euclidean distance, the diverse path selection problem is NP-hard.*

Proof. Given a graph $G = (V, E)$ and $n = |V|$, we construct an auxiliary graph $H = (V', E')$ as shown in Figure 19:

$$V' = V \cup \{v_i' \mid i \in [1..n]\}$$

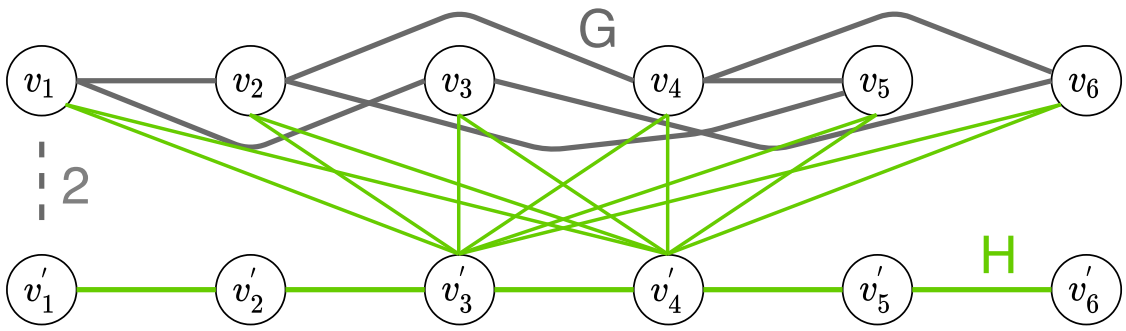


Figure 19: NP-hardness of finding the shortest path with minimum Euclidean distance or minimum Jaccard distance exceeding a threshold. The problem of deciding whether there exists a Hamiltonian path can be reduced to it.

$$E' = E \cup \left\{(v'_i, v'_{i+1}) \mid i \in [1..n-1]\right\} \cup \left\{(v_i, v'_{\lfloor \frac{n}{2} \rfloor}) \mid \forall i \in [1..n]\right\} \cup \left\{(v_i, v'_{\lceil \frac{n+1}{2} \rceil}) \mid \forall i \in [1..n]\right\}$$

G can be any graph, but here we arrange all its vertices in a straight line in an Euclidean space, and place $v'_1 \dots v'_n$ in a parallel line such that $d_M(v_i, v') = 2$.

We reduce the Hamiltonian path problem first to the diverse path selection problem with Jaccard distance d_J. We claim that the Hamiltonian problem can be decided for G if there exists a diverse alternative to the path $P_1 = (v'_1, v'_2, \dots, v'_n)$, which is the straight green path in Figure 19. The threshold for the minimum Jaccard distance is set to $\theta = 0.5$. Only the edges from $v'_{\lceil \frac{n+1}{2} \rceil}$ and $v'_{\lfloor \frac{n}{2} \rfloor}$ allow the path to traverse other vertices than P_1, and note that in any case all vertices of P_1 are part of the alternative path. As the Jaccard distance corresponds to the inverted intersection over union, this requires the path to use at least n new vertices in addition to $v_1, \dots, v_n$ to achieve an IoU of only 0.5. Thus, the only way to find such an alternative path P_2 with $d(P_1, P_2) \geq \theta$ is a path that visits all vertices of G which is a Hamiltonian path in G. If no alternative is found, it is implied that no Hamilton path exists since the connections from $v'_{\lceil \frac{n+1}{2} \rceil}$ and $v'_{\lfloor \frac{n}{2} \rfloor}$ to *all* vertices of G ensure that it would be found if existent.

Similarly, in the Eucledian case with respect to metric d_M we set $\theta = 1$. If a path is found that uses all vertices of G then

$$d(P_1, P_2) = \frac{n \cdot 0 + n \cdot 2}{2n} = 1 \ .$$

Otherwise, if k vertices less are used ($k \in [1..n]$), then

$$d(P_1, P_2) = \frac{n \cdot 0 + (n-k) \cdot 2}{2n - k} = \frac{2n - 2k}{2n - k} < 1$$

In conclusion, if and only if a Hamilton Path exists in G there is an alternative path with sufficient distance d_M. □

5.3 Computational methods

With the metrics and methods described above at hand, we suggest and compare four algorithms to compute the diverse k shortest paths. Most of the methods are based on Eppstein's algorithm, such that for each vertex the distance of the shortest path (or walk) passing through this vertex is given. In the following, each algorithm is explained more in detail, where it is assumed

that the output of Eppstein's algorithm is already pre-computed, i.e. there is a path-cost-map $p : V \longrightarrow \mathbb{R}$, with $p(v)$ as the cost of the shortest (possibly loopy) path through v[9].

5.3.1 FindKSP with Euclidean distance

A straightforward variation of the `FindKSP` algorithm introduced by Liu et al. [64] is to use an Euclidean metric. In the following, `FindKSP` with the Yau-Hausdorff distance will be denoted by **find-ksp-max** and with the mean Euclidean distance **find-ksp-mean**. In the `FindKSP` algorithm the shortest paths are enumerated and greedily selected if the distance to the previous paths is satisfying the constraint. As shown above, this procedure is particularly efficient with the Yau-Hausdorff distance, because a single pass over all vertices yields the next path. Let $\mathcal{K}$ be a set of $k-1$ paths that have already been computed. Then the vertex v^* with sufficient distance and minimal $p(v)$ is selected:

$$v^* = \underset{v \in V}{\operatorname{argmin}}\, p(v) \text{ s.t. } \min_{q \in Q, Q \in \mathcal{K}} \|q - v^*\| > \theta$$

The shortest path through v^* is the k-th path and can be reconstructed from both predecessor maps. Eppstein's method guarantees that this is the shortest path passing through v^*, and if there was a shorter path with sufficient distance, it would have been represented by a different vertex and found beforehand. Note that the distance of v^* to the previous paths is only a lower bound on the Yau-Hausdorff distance, since there can be other vertices in further distance on the same path. If the shortest path trees are given, the runtime is only $\mathcal{O}(kn)$, corresponding to k iterations over all vertices. The path must only be reconstructed from the predecessor map once a diverse vertex is found (k times) and takes at most n steps; it therefore does not increase the runtime.

On the other hand, according to the proof above finding the next shortest path with a minimum *average* Euclidean distance is NP-hard. A method with no guarantees but practical use is again to enumerate the Eppstein paths and reconstructing the paths until finding one that fulfills the constraint. In contrast to the Yau-Hausdorff case, the path must be reconstructued in each iteration, leading to a maximum runtime of $\mathcal{O}(n^2)$. Note however that it is in no way guaranteed that the optimal next path is found, since there might be a path with sufficient mean Euclidean distance that is not represented in the shortest path trees.

5.3.2 Greedy set intersection

A second variant from `FindKSP` that we call **greedy-set** replaces pairwise-operating similarity measures with a measure of vertex intersection with all previously computed path. In other words, a path $P = (v_1, v_2, \ldots v_p)$ is added greedily to the set of paths if less than θ percent of its vertices are contained in any one of the $k-1$ previously computed paths. The method efficiently computes diverse paths in the sense of large non-intersecting parts, but again with no guarantees since the optimal path might not be represented in the shortest path trees.

5.3.3 K-dispersion

On the other hand, in a **k-dispersion** setting the threshold θ describes the maximal cost, and diversity is maximized choosing k paths from a set $K \in C$, $C = \{P : c(P) < \theta\}$ where $c(P)$

[9] If the angle-minimal Bellman-Ford algorithm is used, p would be a map from each edge; $p : E \longrightarrow \mathbb{R}$

denotes the cost of a path P. Here, the 2-approximation in [80, 81] is implemented with the Yen-Hausdorff distance and the Jaccard distance.

5.3.4 Soft similarity penalizing

Finally, a trade-off between diversity and cost was implemented in the sense that vertices close to the previously computed paths K are penalized (denoted in the following by **corridor-penalizing**). Several works have discussed such an approach in relation to the transport of hazardous material [56] and [4]. Here, we set a penalty r to a fraction of the cost of the least cost path, and add it to the Eppstein-formulated vertex cost to yield

$$p'(v) = p(v) + r \cdot d \cdot \min_{u \in P,\ \forall P \in K} \|v - u\|$$

where d is a scale factor defining the penalty radius, and $p'(u) = \infty\ \forall u \in P,\ \forall P \in K$ to avoid selecting the same path twice. Note that only the maximum Euclidean distance is considered, since $p(v)$ is penalized instead of the individual vertices of each path. The method can therefore be seen as a soft version of **find-ksp-max** where paths are not directly excluded if they are too close in the Yau-Hausdorff sense, but only penalized. Of course, a similar strategy could be used on the original resistance values, but then the shortest path would have to be computed k times.

5.4 Experiments

The four algorithms explained above were evaluated on diverse path planning instances. For the experiments presented here we use **Instance 1** of the instances introduced above, at 20m resolution, and always compute 5 diverse alternatives with the class **AngleKSP** that operates on an **ImplicitLG** object (angle costs are also considered). We provide the results for **Instance 2** in appendix C.2 to prove that the results presented here are not instance-specific. Appendix C.1 details the chosen parameters in the experiments.
In Table 4 and Figure 20 we view the problem as a bi-criteria optimization, where summed path costs and path diversity are optimized simultaneously. The diversity values in Table 4 refer to averages over all pairs of paths in the set, and the "sum of costs" corresponds to the sum of all pylon resistances as well as angle costs of the 5 computed paths. The examples in Table 4 were chosen for their similar sum of costs in order to point out differences in the achieved diversity and runtime.

Method	Threshold	Runtime (in s)	Yau-Hausdorff distance	Mean Eucl. distance	Jaccard distance	Sum of costs
find-ksp-max	18.00	6.22	38.80	10.29	0.73	69.44
find-ksp-mean	4.00	72.70	32.10	18.63	0.73	69.39
greedy-set	0.4	4.46	14.56	4.04	0.87	69.44
k-dispersion (Jaccard distance)	1.01	1247.18	29.65	8.94	0.9	69.47
k-dispersion (Yau-Hausdorff)	1.01	8177.36	18.76	4.20	0.67	69.45
corridor-penalizing	80.0	5.05	41.25	10.49	0.66	69.46

Table 4: Overview and comparison of selected algorithms to compute diverse shortest paths. The Yau-Hausdorff and mean Euclidean distances are given in cell units. While all selected configurations lead to similar path costs, the Jaccard distance is optimized by the k-dispersion algorithm with Jaccard distance and by the greedy-set method. Euclidean distance metrics on the other hand are maximal for the corridor-penalizing method or with find-ksp-max and find-ksp-mean. The runtime of k-dispersion methods is significantly larger than for other algorithms.

Path diversity. The results in Table 4 confirm the analytic analysis and motivation for each algorithm. As expected, the **find-ksp-max** and **corridor-penalizing** methods maximize the Yau-Hausdorff distance efficiently, while **find-ksp-mean** is suitable if a high average Euclidean distance is desired. The **greedy-set** algorithm (which adds paths greedily if the intersection of pylons with previous paths is sufficiently small) indeed achieves comparably low IoU (high Jaccard distance), but performs poorly in terms of Euclidean distances between paths.
The long runtime of k-dispersion methods is only partly justified by the results, as the diversity of **k-dispersion (Yau-Hausdorff)** is inferior to other approaches, with an average of only 18.76 cells Yau-Hausdorff distance between paths. The reason is that since the path cost is only bounded by a threshold and not minimized, the algorithm selects paths of higher cost on average than greedy methods and can therefore lead to less diverse paths at the same sum of costs. Here, all paths with a cost below 1.01 of the best path cost were taken into account, and even then the runtime amounts to hours but the achieved Yau-Hausdorff diversity is inferior to other method.

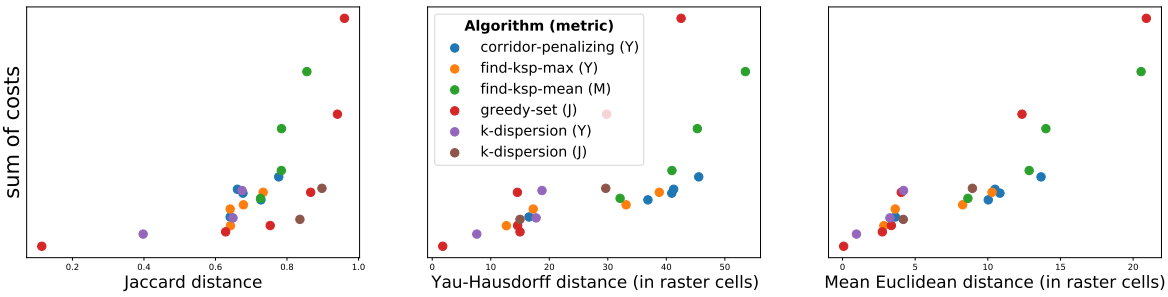


Figure 20: Diversity-cost trade-offs of various diverse k-shortest-path algorithms. Each point refers to a set of five paths that were computed with a specific algorithm and metric. Each algorithm optimizes diversity with respect to a certain metric as shown in the legend (Y - Yau-Hausdorff, J - Jaccard, M - Mean Euclidean distance). The sum of costs (pylon, cable and angle costs) of all five paths is analysed with respect to the diversity of paths in the sense of a Jaccard, Yau-Hausdorff or mean Euclidean distance. The diversity-parameter of each algorithm was varied such that in most cases (but not all) it holds that the higher the diversity (points further right), the larger the diversity parameter was set.
It can be observed that the **greedy-set** method and a **k-dispersion** approximation with Jaccard distances minimize the intersection of paths (left) well while maintaining low costs; i.e. points at the bottom right are optimal in both regards. In an Euclidean setting on the other hand the best performance is achieved by **corridor-penalizing**. Surprisingly there are no major differences in the performance of algorithms when evaluated against Euclidean maximum and average distance metrics.

On the other hand, Figure 20 shows that a **k-dispersion** approximation with the Jaccard distance outperforms other methods in the trade-off between the sum of costs and the Jaccard diversity (averaged over pairs of paths in the set). Arguably, the superiority over faster methods, for example the **greedy-set** procedure, is too small to justify the increase in runtime. Indeed, the **greedy-set** method might be the optimal choice for a user interested in low intersection but not necessarily large Euclidean distance (Figure 20 left). If the Euclidean distance is prioritized, then the `FindKSP` algorithm performs well, with high distances at marginally higher costs as visible in Figure 20 (center and right). Interestingly, **corridor-penalizing** outperforms both slightly, thus efficiently providing a good solution with optimal cost-diversity trade-off. However, in contrast to other methods **corridor-penalizing** is more sensitive to the threshold. Since it is not a hard threshold, the path might be almost the same as a previous one if the penalty is too low. Furthermore, two parameters must be set: The penalty and the distance in which cells are penalized with non-zero cost. One could even choose another function how the penalty reduces with the distance. Here, we fixed the penalty p to 0.01 of the cost of the optimal path, and penalize with linearly decreasing cost, i.e. the the further a cell is from the previous paths, the lower the penalty, and in cells further away than a parameter r, the penalty is zero. Higher diversity than in Figure 20 can be achieved efficiently with a higher penalty p, but then the combination

of p and r must be calibrated. Thus, the choice of parameter is less intuitive than in other methods, where one can directly set a maximum IoU or a minimum Yau-Hausdorff distance. However, since the method is very efficient, the parameter could be calibrated automatically in a grid search to yield a certain average or maximum Euclidean distance.

Time efficiency. Furthermore, observe in Table 4 that the runtime varies widely. Most proposed algorithms iterate efficiently over paths with Eppstein's algorithm and operate greedily, except for the **k-dispersion** algorithm that requires to compute the pairwise distances of all paths whose costs are below a threshold. Thus, a slight increase in the cost threshold leads to a significantly larger set of considered paths, and the run time increases quadratically with the number of paths. In contrast to that, the runtime of **find-ksp-max** and **corridor-penalizing** are *independent of* the diversity threshold, because they only take approximately $k \cdot n$ steps. The **find-ksp-mean** and **greedy-set** algorithms greedily iterate over all Eppstein-paths and computing the distances or intersection, thus taking unpredictably longer the higher the diversity threshold, but still by far less time than k-dispersion methods. Note that the runtime for computing the average Euclidean distance can be improved with space separation methods such as Quad Trees; so far, it is quadratic in the length of the paths.

Qualitative comparison. In addition to the quantitative analysis, the outputs are also contrasted qualitatively. Figure 21 shows a threshold variation for **corridor-penalizing**. It is visible that the paths differ well in both the maximum and average Euclidean distance. The qualitative outputs for all algorithms are shown in appendix C.2 for comparison. To summarize,

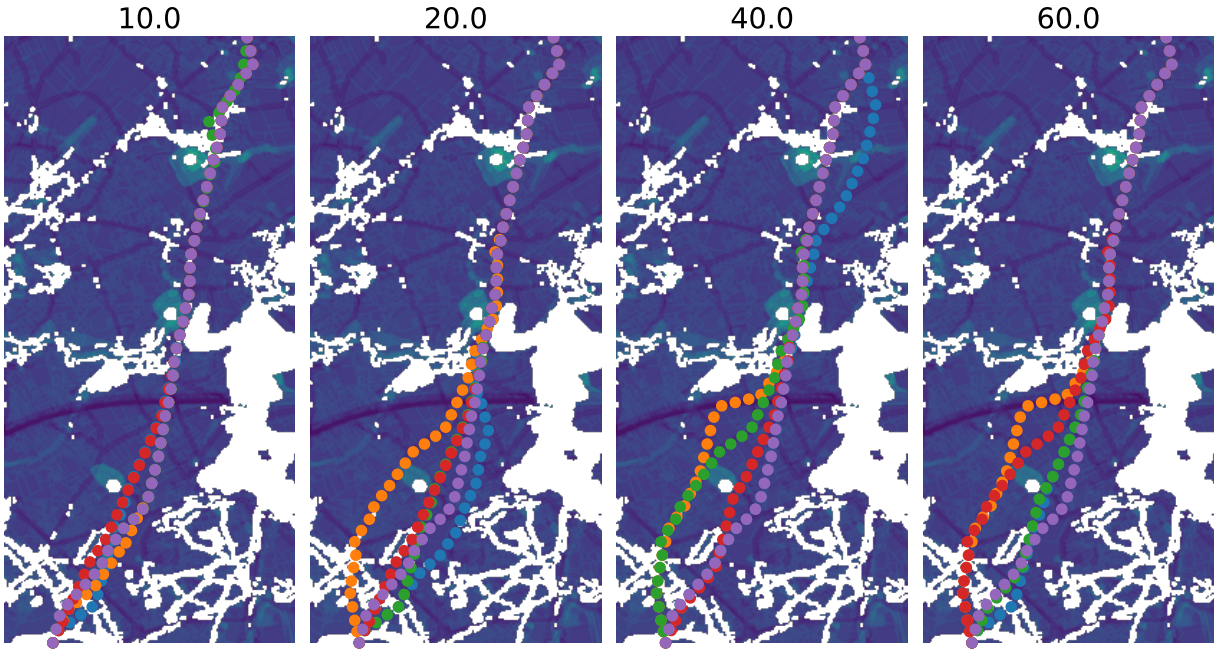


Figure 21: Example output of the **corridor-penalizing** method. The radius of penalty application is noted above each plot, given in raster cell distance. Clearly, the larger the penalty, the more the paths are diverging.

the **greedy-set** algorithm appears to offer the best trade-off between diversity, costs and runtime if spatial distances are not taken into account, whereas **corridor-penalizing** is the first choice if the Euclidean distance is to be optimized. **find-ksp-max** might be preferable since it guarantees an optimal solution and its parameter setting is more intuitive, and **find-ksp-mean** achieves the highest diversity with respect to the mean Euclidean distance.

6 Further shortest path problems in linear infrastructure layout

6.1 Optimizing tower heights in complex terrain

In regions that are not flat, but of significantly varying altitude, additional requirements are placed on transmission line construction. Transmission towers can be of different heights, but higher pylons come at higher costs. In complex terrain, raising the height of the tower is necessary to assure the minimum height of the cable above the ground, here denoted with Δ. Thereby the sag of the cable must be taken into account, taking the shape of a catenary curve. The aim thus becomes to find an optimal height-assignment to each vertex, $h : V \to \mathbb{R}$, in order to compute a path that minimizes the edge costs c and the height costs defined by a function $c_h : \mathbb{R} \to \mathbb{R}$ at the same time. Formally, the goal is to find

$$\min_{P=u_1,\dots,u_{|P|}} \sum_{i=1}^{|P|-1} c((u_i, u_{i+1})) + w_h \cdot c_h(h(u_i)) \ ,$$

where w_h is the weight corresponding to the importance of pylon height costs toward the overall path cost.

However, the assignment of h is constrained to the compliance with a minimum height constraint Δ posed by official construction regulations. At any location along the power line, the cable must always be at least Δ meters above the ground. Let $\mathcal{H} : V \to \mathbb{R}$ define the altitude of the terrain at each point. Then we define an indicator function f that outputs the feasibility of an edge $e = (u, v)$ with a given height assignment function h in the terrain $\mathcal{H}$: If $f(e, h(u), h(v), \mathcal{H}) = 1$, then it is feasible to span a cable between u and v with tower heights $h(u)$ and $h(v)$. Otherwise, it is not feasible, i.e. there is a point between the two pylons where the cable is less than Δ meters above the ground. The general setting is visualized in Figure 22a.

Here, f is restricted to functions where higher towers can not take away feasibility: If $h'(u) \geq h(u)$ and $h'(v) \geq h(v)$ for an edge $e = (u, v)$ and $f(e, h(u), h(v), \mathcal{H}) = 1$, then $f(e, h'(u), h'(v), \mathcal{H}) = 1$ is implied. Intuitively, increasing the height of a tower can never make the cable height lower at any point. We do not want to describe the computation of f in detail, since the calculations are straightforward with standard methods to compute the sag of the line, and subsequent comparison of the cable height and the underlying terrain. For details on the computation of the line sag, please refer to [78, 75]. Suitable parameters were selected from [82].

To the best of our knowledge, there is hardly any literature on the optimization problem with regards to pylon heights. In particular, the general case with continuous values for h and arbitrary complexity of f, the problem seems rather intractable, considering that the height of one pylon modifies the constraints on its neighboring pylons, and it is therefore not straightforward to solve in dynamic programming fashion. We thus constrain ourselves to the discrete case, where the transmission tower height can only be one out of k values. In this application, $h(u) \in \{60, 70, 80\}$ meters is for example a reasonable assumption.

A globally optimal and efficient algorithm for this problem is given by node splitting: Each vertex is replaced by k new vertices, each representing a possible height. Consequently, up to k^2 edges must be placed between a pair of original vertices, connecting the possible combinations of heights. The height costs can be simply added to the normal edge weights. It is trivial to prove that this procedure yields the optimal minimal-height path in polynomial time.

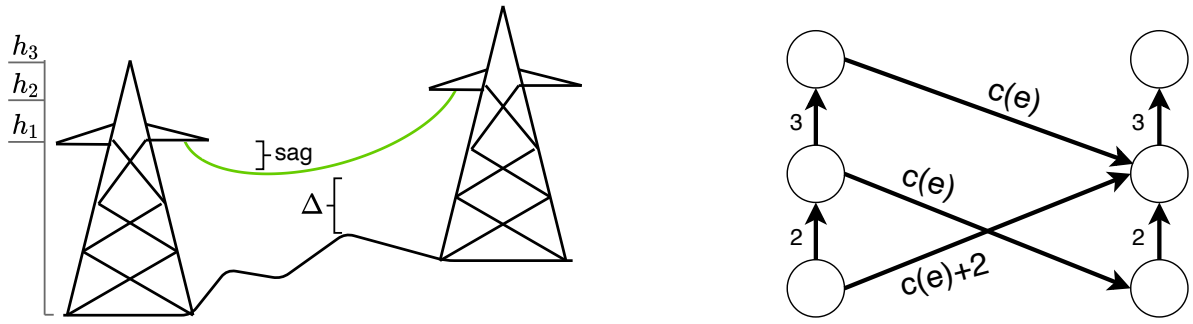
Nevertheless, a factor of k^2 presents a significant increase of graph size. We therefore propose an improved version, exploiting the property that increasing height can not change the feasibility,

such that the lowest possible combination is always desirable. Let $\sigma_1 \dots \sigma_k$ denote the possible discrete height values, in increasing order. As before, the vertices of the graph G are replaced by k new vertices in the auxiliary graph H, but this time only k edges are placed for each edge in G. Consider one edge $e = (u, v)$ and its replacement vertices $u_1' \dots u_k'$ and $v_1' \dots v_k'$. Then the edges of H are the union of two edge sets, namely

$$E_1 = \left\{ (u_i', v_{j^*(i)}') \;\middle|\; \forall e \in E,\ e = (u, v),\ \ \forall i \in [1..k],\ j^*(i) = \min_j j \text{ s.t. } f(\sigma_i, \sigma_j, \mathcal{H}) = 1 \right\}$$

$$E_2 = \left\{ (u_i', u_{i+1}') \;\middle|\; \forall u \in V,\ \forall i \in [1..k-1] \right\}$$

An example is shown in Figure 22, depicting the representation of a single edge in H. With three discrete options for the pylon height, a vertex u is split in three parts u_1, u_2, u_3. In Figure 22, the upmost vertex represents the maximal pylon height θ_k. For each of the three vertices, an outgoing edge is placed, ending in the vertex with minimal feasible height. For example, if the left pylon is of lowest height, then the right pylon must be at least of medium height. An edge would be placed between u_1 and v_2 in H. After the shortest path is computed, the height of a pylon in u corresponds to the maximum height of this vertex passed on the path P, i.e. $\theta_j : \ j = \underset{j:u_j' \in P}{\operatorname{argmax}}\, u_j'$.



(a) Discrete pylon height problem

(b) Auxiliary graph H

Figure 22: Challenges in complex terrain: The sag of the cable must be accounted for when computing the minimal pylon height. An auxiliary graph is proposed that solves the pylon-height-optimal shortest path problem. Vertices are split in k parts, which are connected amongst each other in a single path. Edges are replaced by the minimal feasible connections.

Furthermore, the height costs are placed on each edge $e = (u', v')$, corresponding to the (additional) cost for the pylon in v'. Formally,

$$c\big((u_i', u_{i+1}')\big) = c_h(\sigma_{i+1}) - c_h(\sigma_i) \tag{6.1}$$

$$c\big((u_i', v_{j^*(i)}')\big) = c((u, v)) + c_h(\sigma_{j^*(i)}) \tag{6.2}$$

For example, in Figure 22 $c_h(\sigma_1) = 0,\ c_h(\sigma_2) = 2,\ c_h(\sigma_3) = 5$.

Lemma 6.1. *Any feasible path and height configuration is represented in H, and any path in H is feasible.*

Proof. Since the minimal but yet feasible edges are selected by construction of E_1, all edges between distinct vertices of G are feasible. Furthermore, higher pylon heights are always feasible due to the assumed properties of f, so edges in E_2 of the form (u_i', u_{i+1}') can not change feasibility.

On the other hand, any feasible option is clearly contained in H, since an outgoing edge is added for each $u^{'}$ if there is any feasible edge, and it can be extended to reach a higher pylon of v with an edge of E_2. □

Based on this observation it is easy to see that any shortest path algorithm applied on H will find the optimal height-constrained path. Meanwhile, the size of the graph is larger, namely $|V^{'}| = k \cdot |V|$ due to node splitting, and $|E^{'}| = |E_1| + |E_2| \leq k \cdot |E| + |V| \cdot (k-1)$ because at most k edges replace an edge in G and additional edges are placed between the vertices of $V^{'}$.

6.2 NP-hardness of the Least Average Cost Path Problem

Often planners of power infrastructure aim to avoid high cost areas by all means, since these locations might bring along unpredictable costs, such as legal expenses or delays in construction due to protests of local residents. Since sometimes large detours are accepted to circumvent areas of high cost, the idea was raised to minimize the average costs instead of the sum of costs.

Definition 6.2. *The **Least Average Cost Path Problem** is the problem to find the path P^* with lowest average edge costs, i.e.*

$$P^* = \min_{P=(e_1,\dots,e_n)} \frac{1}{n} \sum_{i=1}^{n} c(e_i)$$

.

Theorem 6.3. *The Least Average Cost Path Problem is NP-hard.*

Proof. The theorem is proven by reduction from the Hamilton Path problem. A similar construction as in subsubsection 5.2.2 is used, depicted in Figure 23b. Given a graph $G = (V, E)$, we show that the existence of an average least cost path in an auxiliary graph H determines whether there is a Hamilton path in G.
$H = (V^{'}, E^{'})$ is constructed by adding two vertices s, t and connecting them to all vertices of G:

$$V^{'} = V \cup \{s, t\}, \quad E^{'} = E \cup \{(s, v) | v \in V\} \cup \{(v, t) | v \in V\}$$

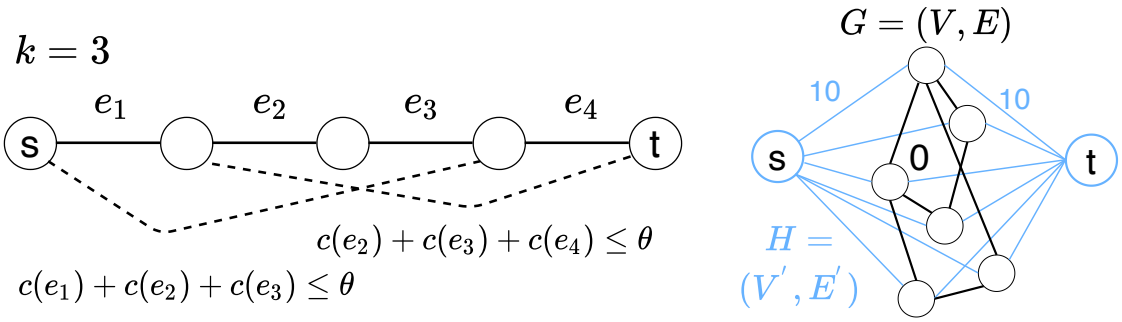
Additionally, $c(e) = 0 \quad \forall e \in E$ (black edges) and $c(e^{'}) = 10 \quad \forall e^{'} \in E^{'} \setminus E$ (blue edges). Let $\theta_c = \frac{20}{|V|+1}$. If there exists a least average cost path P^* in H from s to t, with an average cost of less or equal than θ_c, then *all* vertices in of G occur on P^*, because if $|P^*| < |V|$ then the average cost is

$$\frac{1}{|P^*|} \sum_{e_i \in P^*} c(e_i) = \frac{10 + 0 + \dots + 0 + 10}{|P^*|} > \frac{20}{|V| + 1} = \theta_c$$

Consequently, the computation of the average least cost path in H leads to a Hamilton path in G if possible, and therefore the cost of the average least cost path decides the Hamilton path problem. □

6.3 NP-hardness of the Sliding Window Constrained Shortest Path Problem

The construction in Figure 23b directly implies NP-hardness for a second specific problem appearing in linear infrastructure layout. The problem is visualized in Figure 23a.



(a) Sliding window-constrained shortest path

(b) Auxiliary graph H

Figure 23: Two variations of the shortest path problem are shown to be NP-hard: The least average cost path problem, and shortest path finding with a budget constraint on each window of k consecutive edges, shown in a. b) shows the auxiliary graph H used in the proof: An average least cost path from s to t decides about the Hamiltonian path problem in G (black).

Definition 6.4. *The **Sliding Window Constrained Shortest Path Problem** is the problem to find the shortest path P^* subject to the constraint that the sum of costs in each set of k consecutive edges on P^* is lower than a threshold θ_s. Formally, with $P^* = (e_1, \ldots, e_{|P|})$, it must hold that*

$$\forall j \in [1..|P| - k] : \quad \sum_{i=j}^{j+k} c(e_i) \leq \theta_s$$

Here, the motivation is again rooted in angle constraints. For example, we encountered an application regarding the layout of underground cables, where the angles in a distance of 500 meters cannot exceed 220 degrees because of tension constraints. The problem above regards edge costs instead of angle costs to simplify the problem and generalize it to other applications.

Theorem 6.5. *For non-fixed k, the Sliding Window Constrained Shortest Path Problem is NP-hard.*

Proof. The same auxiliary graph H as above and shown in Figure 23b is constructed from G. We set $k = |V|$ and $\theta_s = 10$. We claim that the only path from s to t where each sliding window fulfills the constraint is a Hamilton path. To see this, note that any path with less than $|V| + 1$ edges has a sliding window of size less or equal than $|V|$ (the window corresponding to the whole path) that includes two edges with cost 10. The existence of a path complying with the sliding window constraint therefore implies a path length of $|V| + 1$ edges which is a Hamilton path in G. □

Note however that for a fixed window size k there might be efficient methods and further work is necessary to explore this possibility.

7 User interface

Power infrastructure planning is an inherently interdisciplinary field, calling for communication between experts from geography, public relations, mathematics and computer science. An automatic planning tool must therefore be easy to use and transparent in the sense that the optimality of routes must be explainable. Thus, there is a clear need for easy-to-use software with intuitive drag-and-drop like features. With the incentive to build a demo version of such software, we build a simple user interface (UI). The UI is built only in Python with the `kivy` package, and implements all major algorithms that were presented here. Figure 24 shows a snapshot of the interface, whose features and functions are explained in detail in the following.

7.1 Data and configuration

First, the data needs to be loaded from the file system. Problematically, the initial input data, geographical layers and weighting, can take various formats and forms of storage, and the pre-processing steps are largely project specific. While there exists a class `DataReader` in the main source code for data pre-processing, this part was not included in the GUI, in order to ensure the applicability of the interface for different data sources. Instead, the input is a generic binary `DAT` file containing three `numpy` arrays and a configuration dictionary. Specifically, the input comprises

- the instance, an array of shape number_categories × raster_width × raster_height, where the category-wise resistance to place a pylon is stored in the corresponding raster cell
- the cable-instance, which can be exactly the same as the instance or vary in value, describing the resistance to span a cable above a cell
- the corridor defining forbidden regions, i.e. a binary array of size raster_width×raster_height with zero in forbidden and ones in allowed cells
- the configuration, a dictionary storing values such as category names and initial weights, minimum and maximum pylon distance and other hyperparameters.

Once the data is loaded, the instance is multiplied with the corridor, such that forbidden regions are displayed in black. In the case of an instance with three categories as in Figure 24, the instance values can be visualized conveniently as RGB values. For example, in Figure 24 the blue region shows a nature reservation area with high environmental costs. In addition to the resistance display, the weight-sliders are initialized to the values given in the configuration file.

7.2 Least cost path computation

After setting the preferred weights of cable (edge) costs, angle costs, and categories, the processing is started either by selecting ”single shortest path” or ”shortest path trees”. While the former takes exactly half of the time, because only the shortest path tree rooted in the source is computed, the latter is required for any further processing, including the computation of diverse alternatives. With the button press, an instance of `ImplicitLG` as introduced in subsubsection 3.2.2 is initialized and the graph is build based on the given parameters and weights. The returned shortest path is visualized on the resistance map in white colour.

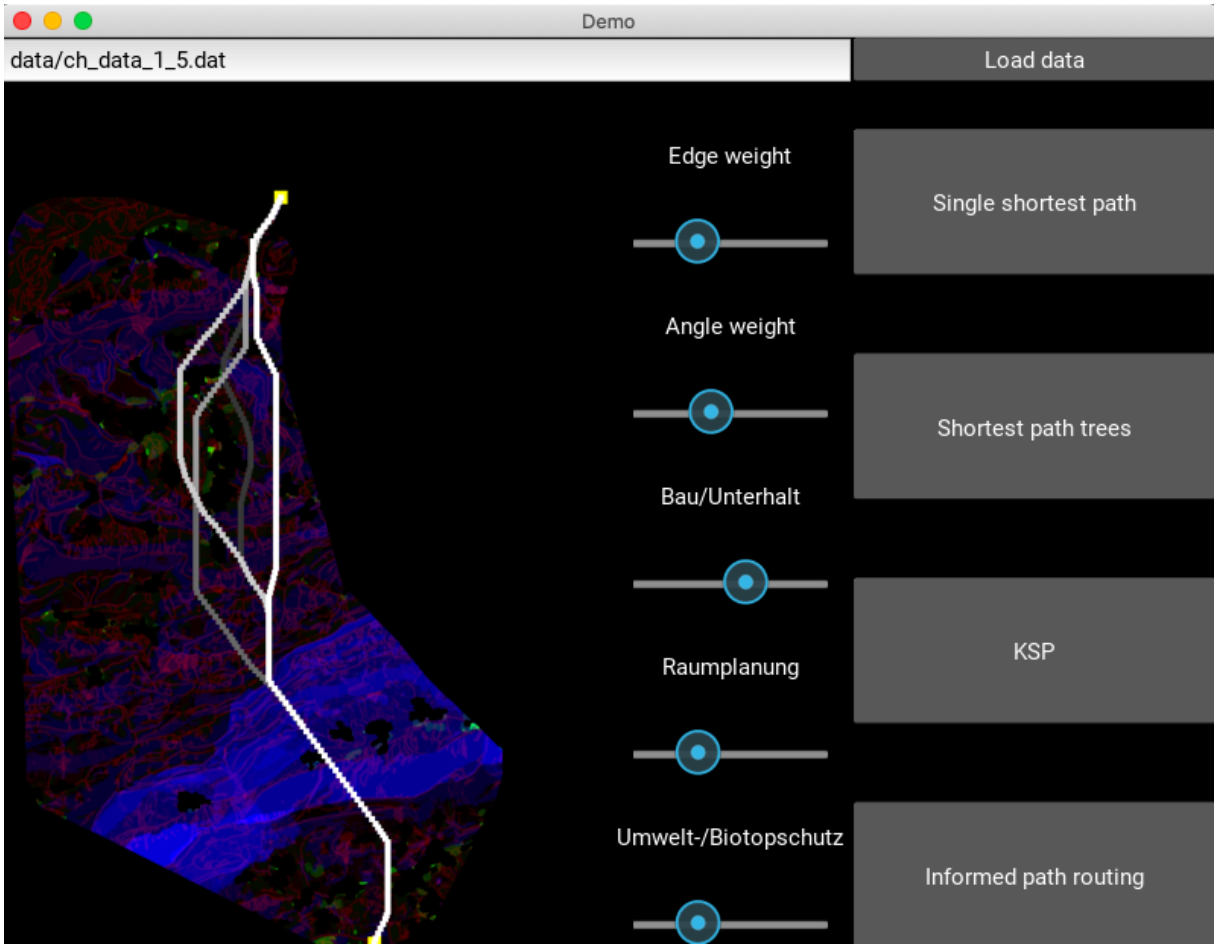


Figure 24: User interface for power infrastructure planning

7.3 K diverse shortest paths and informed routing

Only when both shortest path trees are available, the last two buttons are enabled. Clicking on "KSP" yields a set of k diverse shortest paths computed with the **corridor-penalizing** method as the default (see subsubsection 5.3.4). In this demo version, the parameters cannot be varied, but it is straightforward to ask for further details in a pop-up window. As visible in Figure 24, the output paths are coloured in grey scale: The darker the path, the more resistances. Furthermore, when pressing "Path routing" the user is asked to input (raster-)coordinates of a rectangular area. The optimal path crossing this area is computed with the methods explained in section 3.2.2 (example in Figure 7b).

Together, the interface presents a lean and intuitive demonstration platform to quickly explore and visualize the algorithms for power infrastructure planning introduced here. Meanwhile, it is of course necessary to integrate such algorithms in GIS-software to analyse the suggested paths with respect to the geographic features.

8 Discussion

Power infrastructure layout is a challenge of significant impact on society. Besides ensuring the consistent and robust energy supply, it also plays a major role for the successful transition to renewable energies in view of climate change. At the same time, high investments are at stake, with around 1.5 million euros spent per kilometer of transmission line [97]. Meanwhile, the planning process is still often based on an informed trial-and-error strategy, or at the most a raster-based approach to route a path through a grid-like representation of the geographic resistances. Here, we instead developed a global optimization method, that rigorously and flexibly computes the power line route of minimal cost. Importantly, we explicitly model the placement of pylons in given distances, and thereby jointly optimize the resistances caused by the placement of pylons and the transition of cables.

In summary, our main contribution comprises algorithmic work and a computational framework to compute the angle-optimal least cost path, and possibly diverse alternatives. First, we explained computational methods to efficiently operate on large graphs of even billions of edges. We described an iterative "pipeline" process to increase resolution and at the same time narrow down the region of interest. Secondly, we introduced novel algorithms for the efficient minimization of angles along the path, where best performance can be achieved with an algorithm applicable for linear and monotonically increasing di-edge cost functions. Third, various methods for the computation of a set of alternative routes were discussed, trading off costs and diversity of the routes. We show that with certain metrics the greedy selection of a diverse path is actually NP-hard, and thus develop heuristic methods that are experimentally shown to achieve diverse paths according to these metrics. Last, we presented a simple method to take into account the costs of higher pylons necessary in complex terrain and analyzed further challenges in linear infrastructure layout, namely sliding-window type constraints and the the average least cost path.

While our experiments show the system's efficiency to compute the globally optimal paths, our modelling approach comes at cost of graph size. In contrast to path planning in a grid (with edges to the 8-neighborhood), in our approach the number of edges shows cubic growth with increasing resolution, instead of quadratic growth. An additional limitation is given by the dependency on the minimum and maximum pylon distances. Our methods still offer a feasible time and space efficiency for the discussed real-life application scenarios, in particular with the pipeline approach and efficient angle optimization. It was also shown that the resistances and angle costs are decreased substantially with our method, proving that it it worthwhile to investigate and accelerate globally optimal methods. Further work is required to yield a graph representation or processing pipeline that allows (approximate) shortest path computations for large project regions at high resolution. Similarly, our work on the optimization of pylon heights is improving significantly over the baseline solution, but still leads to a multiple of the graph size, proportional to the number of discrete pylon heights. More theoretical work is required to find an efficient exact algorithm or approximation.

An entirely different challenge is posed by the communication of mathematical methods to planning agencies and public authorities. Although a global optimum is computed, the result is naturally very sensitive to the input, namely the weighting of layers and cost categories, importance of cable resistances, angle costs and pylon heights. Any software for power infrastructure planning must therefore guide the user through parameter setting and provide feedback in the form of visualizations. Our GUI serves as a simple prototype, demonstrating how such a system could use sliders for weight tuning and continuous visualization of the output paths. Although by no means comprehensive, it shows the ease of power infrastructure planning with mathematical

optimization.

Moreover, the proposed methodology is by no means limited to transmission line planning. The presented framework could for example be used for the layout of pipes, underground transmission lines, or highways. In each of those applications it is necessary to optimize costs and angles simultaneously since angled elements are more expensive or lead to higher resistance. In addition, it is equally beneficial in many other applications to compute several diverse alternatives, since it gives planners the possibility to analyze and compare options and to select the optimal one based on their insights and experience in planning and construction. We thus hope to pave the road towards software-assisted linear infrastructure planning with rigorous mathematical methods, with the potential to lower the costs for the planning process considerably, and to decrease the clash between local residents and planning agencies, or environmentalists and construction authorities.

References

[1] Harold Abelson and Andrea A DiSessa. *Turtle geometry: The computer as a medium for exploring mathematics*. MIT press, 1986.

[2] Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *International Symposium on Experimental Algorithms*, pages 230–241. Springer, 2011.

[3] Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. Hierarchical hub labelings for shortest paths. In *European Symposium on Algorithms*, pages 24–35. Springer, 2012.

[4] Vedat Akgün, Erhan Erkut, and Rajan Batta. On finding dissimilar paths. *European Journal of Operational Research*, 121(2):232–246, 2000.

[5] Takuya Akiba, Takanori Hayashi, Nozomi Nori, Yoichi Iwata, and Yuichi Yoshida. Efficient top-k shortest-path distance queries on large networks by pruned landmark labeling. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[6] Edoardo Amaldi, Giulia Galbiati, and Francesco Maffioli. On minimum reload cost paths, tours, and flows. *Networks*, 57(3):254–260, 2011.

[7] Ahmed Sharique Anees. Grid integration of renewable energy sources: Challenges, issues and possible solutions. In *2012 IEEE 5th India International Conference on Power Electronics (IICPE)*, pages 1–6. IEEE, 2012.

[8] Juancarlo Anez, Tomás De La Barra, and Beatnz Pérez. Dual graph representation of transport networks. *Transportation Research Part B: Methodological*, 30(3):209–216, 1996.

[9] Stefano Bagli, Davide Geneletti, and Francesco Orsi. Routeing of power lines through least-cost path analysis and multicriteria evaluation to minimise environmental impacts. *Environmental Impact Assessment Review*, 31(3):234–239, 2011.

[10] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F Werneck. Route planning in transportation networks. In *Algorithm engineering*, pages 19–80. Springer, 2016.

[11] Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.

[12] JK Berry. Optimal path analysis and corridor routing: Infusing stakeholder perspective in calibration and weighting of model criteria. In *Proc. GeoTech Conf. on Geographic Information Systems*, 2004.

[13] Kjetil Modolv Bevanger, Gundula Bartzke, Henrik Brøseth, Espen Lie Dahl, Jan Ove Gjershaug, Frank Ole Hanssen, Karl-Otto Jacobsen, Pål Kvaløy, Roelof Frans May, Roger Meås, et al. Optimal design and routing of power lines; ecological, technical and economic perspectives (optipol). progress report 2010. *NINA rapport*, 2010. URL https://www.nina.no/archive/nina/PppBasePdf/rapport/2010/619.pdf.

[14] Fritz Bökler and Markus Chimani. Approximating multiobjective shortest path in practice. In *2020 Proceedings of the Twenty-Second Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 120–133. SIAM, 2020.

[15] Jack E Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems journal*, 4(1):25–30, 1965.

[16] S Buecher and Christian Lantuejoul. Use of watershed in contour detection. In *Proc. Int. Workshop Image Processing, Real-Time Edge and Motion Detection/Estimation, Rennes, France*, pages 17–21, 1979.

[17] Tom Caldwell. On finding minimum routes in a network with turn penalties. *Communications of the ACM*, 4(2):107–108, 1961.

[18] Theodoros Chondrogiannis, Panagiotis Bouros, Johann Gamper, and Ulf Leser. Alternative routing: k-shortest paths with limited overlap. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 1–4, 2015.

[19] Theodoros Chondrogiannis, Panagiotis Bouros, Johann Gamper, Ulf Leser, and David B Blumenthal. Finding k-dissimilar paths with minimum collective length. In *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 404–407, 2018.

[20] Theodoros Chondrogiannis, Panagiotis Bouros, Johann Gamper, Ulf Leser, and David B Blumenthal. Finding k-shortest paths with limited overlap. *The VLDB Journal*, pages 1–25, 2020.

[21] Walter Collischonn and Jorge Victor Pilar. A direction dependent least-cost-path algorithm for roads and canals. *International Journal of Geographical Information Science*, 14(4): 397–406, 2000.

[22] Jörg Cortekar and Markus Groth. Adapting energy infrastructure to climate change–is there a need for government interventions and legal obligations within the german "energiewende". *Energy Procedia*, 73(June (12)):17, 2015.

[23] Rodolfo Mendes de Lima, Reinis Osis, Anderson Rodrigo de Queiroz, and Afonso Henriques Moreira Santos. Least-cost path analysis and multi-criteria assessment for routing electricity transmission lines. *IET Generation, Transmission & Distribution*, 10(16):4222–4230, 2016.

[24] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering route planning algorithms. In *Algorithmics of large and complex networks*, pages 117–139. Springer, 2009.

[25] Daniel Delling, Andrew V Goldberg, Thomas Pajor, and Renato F Werneck. Customizable route planning. In *International Symposium on Experimental Algorithms*, pages 376–387. Springer, 2011.

[26] Narsingh Deo and Chi-Yin Pang. Shortest-path algorithms: Taxonomy and annotation. *Networks*, 14(2):275–323, 1984.

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

[28] AR Ebrahimipoor, A Alimohamadi, AA Alesheikh, and H Aghighi. Routing of water pipeline using gis and genetic algorithm. *J Appl Sci*, 9(23):4137–4145, 2009.

[29] Matthias Ehrgott. *Multicriteria optimization*, volume 491. Springer Science & Business Media, 2005.

[30] Tali Eilam-Tzoreff. The disjoint shortest paths problem. *Discrete applied mathematics*, 85 (2):113–138, 1998.

[31] David Eppstein. Finding the k shortest paths. *SIAM Journal on computing*, 28(2):652–673, 1998.

[32] Funda Ergun, Rakesh Sinha, and Lisa Zhang. An improved fptas for restricted shortest path. *Information Processing Letters*, 83(5):287–291, 2002.

[33] Erhan Erkut and Vedat Verter. Modeling of transport risk for hazardous materials. *Operations research*, 46(5):625–642, 1998.

[34] ESRI. Creating the least-cost path. https://pro.arcgis.com/en/pro-app/tool-reference/spatial-analyst/creating-the-least-cost-path.htm, 2020. [Online; accessed 20-May-2020].

[35] Olivier Feix, Ruth Obermann, Marius Strecker, and Regina König. Netzentwicklungsplan strom 2014: Erster entwurf der übertragungsnetzbetreiber. *50Hertz Transmission GmbH, Amprion GmbH, TenneT TSO GmbH, TransnetBW GmbH*, 2014.

[36] Christopher B Field, Vicente Barros, Thomas F Stocker, and Qin Dahe. *Managing the risks of extreme events and disasters to advance climate change adaptation: special report of the intergovernmental panel on climate change*. Cambridge University Press, 2012.

[37] Lester R Ford Jr. Network flow theory. Technical report, Rand Corp Santa Monica Ca, 1956.

[38] Xavier Gandibleux and Matthias Ehrgott. 1984-2004–20 years of multiobjective metaheuristics. but what about the solution of combinatorial problems with multiple objectives? In *International Conference on Evolutionary Multi-Criterion Optimization*, pages 33–46. Springer, 2005.

[39] Len L Garver. Transmission network estimation using linear programming. *IEEE Transactions on Power Apparatus and Systems*, (7):1688–1697, 1970.

[40] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *International Workshop on Experimental and Efficient Algorithms*, pages 319–333. Springer, 2008.

[41] R Gill, W Jewell, T Grossardt, and K Bailey. Landscape features in transmission line routing. In *Proc. IEEE Power Eng. Soc. Transmission Distribution Conf. Exhib*, pages 1122–1126, 2006.

[42] Paul C Gilmore. Optimal and suboptimal algorithms for the quadratic assignment problem. *Journal of the society for industrial and applied mathematics*, 10(2):305–313, 1962.

[43] Marinus Gottschau, Marcus Kaiser, and Clara Waldmann. The undirected two disjoint shortest paths problem. *Operations Research Letters*, 47(1):70–75, 2019.

[44] Ted Grossardt, Keiron Bailey, and Joel Brumm. Analytic minimum impedance surface: Geographic information system-based corridor planning methodology. *Transportation Research Record*, 1768(1):224–232, 2001.

[45] Gabriel Y Handler and Israel Zang. A dual algorithm for the constrained shortest path problem. *Networks*, 10(4):293–309, 1980.

[46] FO Hanssen, RF May, J Thomassen, and KM Bevanger. A least cost path (lcp) toolbox for optimal routing of high voltage power lines for a sustainable future. In *10th Int. Symp. Environmental Concerns in Rights-of-Way Management*, pages 181–186. USA Utility Arborist Association Champaign, IL, 2014.

[47] Frank Hanssen, Roel May, Jiska Van Dijk, Bård G Stokke, and Matteo De Stefano. Spatial multi-criteria decision analysis (smcda) toolbox for consensus-based siting of powerlines and wind-power plants (consite). 2018.

[48] Guang Hao, Dian-ye Zhang, and Xun-sheng Feng. Model and algorithm for shortest path of multiple objectives. *Journal of Southwest Jiaotong University*, 42(5):641–646, 2007.

[49] Robert M Haralick, Stanley R Sternberg, and Xinhua Zhuang. Image analysis using mathematical morphology. *IEEE transactions on pattern analysis and machine intelligence*, (4): 532–550, 1987.

[50] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[51] Refael Hassin. Approximation schemes for the restricted shortest path problem. *Mathematics of Operations research*, 17(1):36–42, 1992.

[52] John Hershberger, Matthew Maxel, and Subhash Suri. Finding the k shortest simple paths: A new algorithm and its implementation. *ACM Transactions on Algorithms (TALG)*, 3 (4):45–es, 2007.

[53] Hao Hu and Renata Sotirov. Special cases of the quadratic shortest path problem. *Journal of Combinatorial Optimization*, 35(3):754–777, 2018.

[54] Hao Hu and Renata Sotirov. On solving the quadratic shortest path problem. *INFORMS Journal on Computing*, 32(2):219–233, 2020.

[55] Jeffrey M Jaffe. Algorithms for finding paths with multiple constraints. *Networks*, 14(1): 95–116, 1984.

[56] PE Johnson, DS Joy, DB Clarke, and JM Jacobi. Highway 3.1: An enhanced highway routing model: Program description, methodology, and revised users manual. Technical report, Oak Ridge National Lab., TN (United States), 1993.

[57] Sriram Kalaga and Prasad Yenumula. *Design of electrical transmission lines: structures and foundations*. CRC Press, 2016.

[58] Anjeneya Swami Kare. A simple algorithm for replacement paths problem. *arXiv preprint arXiv:1511.06905*, 2015.

[59] Alyson Kenward and Urooj Raja. Blackout: Extreme weather climate change and power outages. *Climate central*, 10:1–23, 2014.

[60] Friedrich Kiessling, Peter Nefzger, Joao Felix Nolasco, and Ulf Kaintzyk. *Overhead power lines: planning, design, construction*. Springer, 2014.

[61] Sven Kosub. A note on the triangle inequality for the jaccard distance. *Pattern Recognition Letters*, 120:36–38, 2019.

[62] Eugene L Lawler. The quadratic assignment problem. *Management science*, 9(4):586–599, 1963.

[63] Yongfu Li, Qing Yang, Wenxia Sima, Jiaqi Li, and Tao Yuan. Optimization of transmission-line route based on lightning incidence reported by the lightning location system. *IEEE transactions on power delivery*, 28(3):1460–1468, 2013.

[64] Huiping Liu, Cheqing Jin, Bin Yang, and Aoying Zhou. Finding top-k shortest paths with diversity. *IEEE Transactions on Knowledge and Data Engineering*, 30(3):488–502, 2017.

[65] Dean H Lorenz and Danny Raz. A simple efficient approximation scheme for the restricted shortest path problem. *Operations Research Letters*, 28(5):213–219, 2001.

[66] Ingo Lütkebohle. Netzentwicklunsgplan strom. https://www.netzentwicklungsplan.de/de/node/508, 2020. [Online; accessed 20-May-2020].

[67] Marcelino Madrigal and Steven Stoft. *Transmission expansion for renewable energy scale-up: Emerging lessons and recommendations*. The World Bank, 2012.

[68] Jacek Malczewski. Gis-based multicriteria decision analysis: a survey of the literature. *International journal of geographical information science*, 20(7):703–726, 2006.

[69] Ernesto Queiros Vieira Martins. On a multicriteria shortest path problem. *European Journal of Operational Research*, 16(2):236–245, 1984.

[70] S Mirasgedis, Y Sarafidis, E Georgopoulou, V Kotroni, K Lagouvardos, and DP Lalas. Modeling framework for estimating impacts of climate change on electricity demand at regional level: case of greece. *Energy Conversion and Management*, 48(5):1737–1750, 2007.

[71] Gautam Mitra and K Wolfenden. A computer technique for optimizing the sites and heights of transmission line towers—a dynamic programming approach. *The Computer Journal*, 10(4):347–351, 1968.

[72] Cláudio Monteiro, Vladimiro Miranda, Ignacio J Ramirez-Rosado, Pedro J Zorzano-Santamaria, Eduardo Garcia-Garrido, and L Alfredo Fernández-Jiménez. Compromise seeking for power line path selection based on economic and environmental corridors. *IEEE Transactions on Power Systems*, 20(3):1422–1430, 2005.

[73] Claudio Monteiro, Ignacio J Ramírez-Rosado, Vladimiro Miranda, Pedro J Zorzano-Santamaría, Eduardo García-Garrido, and L Alfredo Fernández-Jiménez. Gis spatial analysis applied to electric line routing optimization. *IEEE transactions on Power Delivery*, 20(2):934–942, 2005.

[74] Edward F Moore. The shortest path through a maze. In *Proc. Int. Symp. Switching Theory, 1959*, pages 285–292, 1959.

[75] Michael Muhr, Stephan Pack, Robert Schwarz, and Stefan Jaufer. Calculation of overhead line sags. In *51st Internationales Wissenschaftliches Kolloquium*, volume 10, 2006.

[76] Mathaios Panteli and Pierluigi Mancarella. Influence of extreme weather and climate change on the resilience of power systems: Impacts and possible mitigation strategies. *Electric Power Systems Research*, 127:259–270, 2015.

[77] Nadine Piveteau, Joram Schite, Martin Raubal, and Robert Weibel. A novel approach to the routing problem of overhead transmission lines. Master's thesis, Geographisches Institut der Universität Zürich, 2017.

[78] RE Popp, CJ Dabekis, and FM Fullerton. Electronic computer program permits optimized spotting of electric transmission towers. *IEEE Transactions on Power Apparatus and Systems*, 82(66):360–365, 1963.

[79] JC Ranyard and A Wren. The optimum arrangement of towers in an electric power transmission line. *The Computer Journal*, 10(2):157–161, 1967.

[80] Sekharipuram S Ravi, Daniel J Rosenkrantz, and Giri Kumar Tayi. Heuristic and special case algorithms for dispersion problems. *Operations Research*, 42(2):299–310, 1994.

[81] SS Ravi, Daniel J Rosenkrantz, and Giri Kumar Tayi. Facility dispersion problems: Heuristics and special cases. In *Workshop on Algorithms and Data Structures*, pages 355–366. Springer, 1991.

[82] Manuel Reta-Hernández. Transmission line parameters, 2012.

[83] Borzou Rostami, Federico Malucelli, Davide Frey, and Christoph Buchheim. On the quadratic shortest path problem. In *International Symposium on Experimental Algorithms*, pages 379–390. Springer, 2015.

[84] Borzou Rostami, Guy Desaulniers, Fausto Errico, and Andrea Lodi. *The vehicle routing problem with stochastic and correlated travel times*. GERAD HEC Montréal, 2017.

[85] Borzou Rostami, André Chassein, Michael Hopf, Davide Frey, Christoph Buchheim, Federico Malucelli, and Marc Goerigk. The quadratic shortest path problem: complexity, approximability, and solution methods. *European Journal of Operational Research*, 268(2): 473–485, 2018.

[86] Raza Samar and Waseem Ahmed Kamal. Optimal path computation for autonomous aerial vehicles. *Cognitive Computation*, 4(4):515–525, 2012.

[87] Afonso Henriques Moreira Santos, Rodolfo Mendes de Lima, Camilo Raimundo Silva Pereira, Reinis Osis, Giulia Oliveira Santos Medeiros, Anderson Rodrigo de Queiroz, Bárbara Karoline Flauzino, Arthur Rohr Paschoal Corrêa Cardoso, Luiz Czank Junior, Renato Antonio dos Santos, et al. Optimizing routing and tower spotting of electricity transmission lines: An integration of geographical data and engineering aspects into decision-making. *Electric Power Systems Research*, 176:105953, 2019.

[88] Antonio Sedeno-Noda and Andrea Raith. A dijkstra-like method computing all extreme supported non-dominated solutions of the biobjective shortest path problem. *Computers & Operations Research*, 57:83–94, 2015.

[89] S Ghandehari Shandiz, G Doluweera, WD Rosehart, L Behjat, and JA Bergerson. Investigation of different methods to generate power transmission line routes. *Electric Power Systems Research*, 165:110–119, 2018.

[90] Alfonso Shimbel. Structure in communication nets. In *Proceedings of the symposium on information networks*, pages 119–203. Polytechnic Institute of Brooklyn, 1954.

[91] Takeshi Shirabe. On distortion of raster-based least-cost corridors. In *The Annual International Conference on Geographic Information Science*, pages 101–113. Springer, 2016.

[92] Takeshi Shirabe. A method for finding a least-cost wide path in raster space. *International Journal of Geographical Information Science*, 30(8):1469–1485, 2016.

[93] Raj A Sivakumar and Rajan Batta. The variance-constrained shortest path problem. *Transportation Science*, 28(4):309–316, 1994.

[94] Anders JV Skriver and Kim Allan Andersen. A label correcting approach for solving bicriterion shortest-path problems. *Computers & Operations Research*, 27(6):507–524, 2000.

[95] Kun Tian, Xiaoqian Yang, Qin Kong, Changchuan Yin, Rong L He, and Stephen S-T Yau. Two dimensional yau-hausdorff distance with applications on comparison of dna and protein sequences. *PloS one*, 10(9), 2015.

[96] Cenk Tort, Serkan Şahin, and Oğuzhan Hasançebi. Optimum design of steel lattice transmission line towers using simulated annealing and pls-tower. *Computers & Structures*, 179: 75–94, 2017.

[97] German TSOs. Netzentwicklungsplan strom 2025. zweiter entwurf der übertragungsnetzbetreiber. *50Hertz Transm. GmbH (50 Hertz); Amprion GmbH (Amprion); TenneT TSO GmbH (TenneT); TransnetBW GmbH*, 2015.

[98] Chi Tung Tung and Kim Lin Chew. A multicriteria pareto-optimal path algorithm. *European Journal of Operational Research*, 62(2):203–209, 1992.

[99] Murray Turoff and Harold A Linstone. The delphi method-techniques and applications. 2002.

[100] Ekunda Lukata Ulungu and Jacques Teghem. Multi-objective combinatorial optimization problems: A survey. *Journal of Multi-Criteria Decision Analysis*, 3(2):83–104, 1994.

[101] EC Van Ierland, K De Bruin, RB Dellink, and A Ruijs. A qualitative assessment of climate adaptation options and some estimates of adaptation costs. 2007.

[102] Miguel Vega and Hector G Sarmiento. Image processing application maps optimal transmission routes. *IEEE computer applications in power*, 9(2):47–51, 1996.

[103] DW Wang and Yue-Sun Kuo. A study on two geometric location problems. *Information processing letters*, 28(6):281–286, 1988.

[104] BM Weedy. Environmental aspects of route selection for overhead lines in the usa. *Electric power systems research*, 16(3):217–226, 1989.

[105] NA West, Barry Dwolatzky, and AS Meyer. Terrain based routing of distribution cables. *IEEE computer Applications in power*, 10(1):42–46, 1997.

[106] Stephan Winter. Modeling costs of turns in route planning. *GeoInformatica*, 6(4):345–361, 2002.

[107] Hans-Christoph Wirth and Jan Steffan. Reload cost problems: minimum diameter spanning tree. *Discrete Applied Mathematics*, 113(1):73–85, 2001.

[108] Ferit Yakar and Fazil Celik. A highway alignment determination model incorporating gis and multi-criteria decision making. *KSCE Journal of Civil Engineering*, 18(6):1847–1857, 2014.

[109] Jin Y Yen. An algorithm for finding shortest routes from all source nodes to a given destination in general networks. *Quarterly of Applied Mathematics*, 27(4):526–530, 1970.

[110] Jin Y Yen. Finding the k shortest loopless paths in a network. *management Science*, 17 (11):712–716, 1971.

[111] Yuli Zhang, Zuo-Jun Max Shen, and Shiji Song. Distributionally robust optimization of two-stage lot-sizing problems. *Production and Operations Management*, 25(12):2116–2131, 2016.

[112] Yuriy Zinchenko, Haotian Song, and William Rosehart. Optimal transmission network topology for resilient power supply. In *International Conference on Information Systems, Logistics and Supply Chain*, pages 138–150. Springer, 2016.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten Semester-, Bachelor- und Master-Arbeit oder anderen Abschlussarbeit (auch der jeweils elektronischen Version).

Die Dozentinnen und Dozenten können auch für andere bei ihnen verfasste schriftliche Arbeiten eine Eigenständigkeitserklärung verlangen.

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und Betreuerinnen der Arbeit.

Titel der Arbeit (in Druckschrift):

Algorithmic approaches for optimizing linear infrastructure planning

Verfasst von (in Druckschrift):
Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	**Vorname(n):**
Wiedemann	**Nina**

Ich bestätige mit meiner Unterschrift:
- Ich habe keine im Merkblatt „Zitier-Knigge“ beschriebene Form des Plagiats begangen.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
- Ich habe keine Daten manipuliert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

Ort, Datum	**Unterschrift(en)**
Zürich, 14.09.2020	

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.

Appendices

A Parameter specifications

	Instance 1	Instance 2	Instance 3
Traversing of forbidden regions allowed?	No	Yes	Yes
Angle weight	0.1	0.2	0.4
Cable weight	0.33	0.33	0.33
Cost categories	cultural, parcel, technical, planning]	resistance	Bau/Unterhalt, Raumplanung, Umwelt-/Biotopschutz]
Category weights	[4.0, 1.0, 2.0, 3.0]	[1]	[1, 1, 1]
Minimal pylon distance (in m)	150	350	150
Maximal pylon distance (in m)	250	500	250
Maximal angle between s-t-connection and cable direction	$\frac{\pi}{2}$	$\frac{\pi}{2}$	$\frac{\pi}{2}$
Maximal angle between two cables	$\frac{\pi}{2}$	$\frac{\pi}{2}$	$\frac{\pi}{2}$

Table 5: Parameter specifications for each instance.

B An accelerated update algorithm for linear angle cost functions

Theorem B.1. *Let $c_a : [0, 180] \longrightarrow \mathbb{R}^+$ be a linear monotonically increasing angle cost function. Then the runtime of algorithm 1 reduces to $\mathcal{O}\Big(p \cdot \sum_v \big(\delta^-(v) + \delta^+(v)\big) \log\big(\delta^+(v)\delta^-(v)\big)\Big)$, since at each vertex with k incoming and l outgoing edges only $\mathcal{O}\big((k+l)\log kl\big)$ operations are executed. This leads to a runtime of $\mathcal{O}\Big(pm \log d^2\Big)$ in a d-regular graph.*

In the following, it will be shown that the update of a single vertex v requires $\mathbf{O}\Big(\cdot \sum_v \big(\delta^-(v) + \delta^+(v)\big) \log\big(\delta^+(v)\delta^-(v)\big)\Big)$, which directly implies the overall runtime by the structure of algorithm 1. We denote the incoming edges by the set $E^- = \{e_1^-, \ldots, e_k^-\}$ and the outgoing edges by $E^+ = \{e_1^+, \ldots, e_l^+\}$. Note that if the graph is undirected, then it is simply transformed into a directed graph, such that each original edge appears once in E^- and once in E^+, and $k = l$. Thus, from now on a directed graph is assumed. The algorithm proceeds in several steps: a) for each incoming edge, the two angle-wise optimal outgoing edges are found b) these edge tuples are sorted by their combined path cost and c) the outgoing edges are updated iteratively. Each of these steps is described in detail in the following and correctness is proven.

B.1 Angle assignment

First, the edges in E^+ are ordered according to their radial position. In practice, an arbitrary edge can be picked, e.g. e_1^+, and for all other edges in E^+ and E^- the angle with respect to e_1^+ is computed using the arc tangens, such that the output range is between zero and 360 degrees. Let $\theta(e) : E \longrightarrow [0, 360)$ be the function assigning each edge e the angle to e_1^+, called *angle value* in the following. Only in this setting it holds that $\theta(e_i^-) = \theta(e_j^+)$ if and only if e_i^- and e_j^+ form a straight line (e_j^+ leaves v in the same angle as e_i^- reaches it). For simplicity, it is assumed from now on that the indices of the outgoing edges $1 \ldots l$ correspond to the position of the edge according to the angle order such that $\theta(e_1^+) \leq \theta(e_2^+) \leq \cdots \leq \theta(e_l^+)$[10]. With this notation, the

[10] the edges can be sorted and re-indexed in $\mathcal{O}(l \log l)$

aim to find the optimal predecessor in terms of angle and distances for each outgoing edge can be formulated as

$$\forall j: \quad \text{Find} \ \operatorname*{argmin}_i D[e_i^-] + c_a(|\theta(e_i^+) - \theta(e_j^-)|_m) \ , \quad |x|_m = \min\{x, 360 - x\} \ .$$

The operation $|x|_m$ computes the smaller angle between both edges and thereby ensures that the input of c_a lies within its domain $[0, 180]$. Here, as a first step we find the two closest outgoing edges for each incoming edge *only with respect to the angle* both in clockwise and counter-clockwise direction.

Lemma B.2. *The angle-optimal edge-tuples can be constructed in* $\mathbf{O}\Big((k+l)\log l\Big)$.

Proof. The proof is given with algorithm 6. Assume one wants to find the closest outgoing edge e^{+*} for a given incoming edge e^-. By "closest", we refer to the edge tuple with the smallest angle, i.e. the incoming and outgoing edge optimally resemble a straight line. By definition of the angle-value function θ, the closest incoming edge is simply the one with the most similar angle value, i.e.

$$e^{+*} = \operatorname*{argmin}_{e^+ \in E^-} |\theta(e^+) - \theta(e^-)|_m$$

A suitable structure to find such matches efficiently is given with an AVL tree structure, where the closest values can be found in logarithmic time once the tree is build. Thus, a balanced AVL tree denoted by H is constructed by inserting the values $\theta(e^+)$ for all $e^+ \in E^+$. Building H only takes $\mathbf{O}(l \log l)$. Then, each incoming edge is input to algorithm 6 which yields the two closest elements.

Algorithm 6: `find-closest`

```
Input: AVL tree H (balanced and sorted by key), element x
// define σ_l and σ_r as the currently closest elements
σ_l, σ_r = ∞
n = H.root()
// Traverse tree
while not isLeaf(n) do
    if x > n.key() then
        σ_l = n
        n = getRightChild(n)
    else
        σ_r = n
        n = getLeftChild(n)
end
// check border cases if one is still infinity
if σ_l is ∞ then
    // set left bound to the largest element in the tree
    σ_l = H[−1]
if σ_r is ∞ then
    // set right bound to the smallest element in the tree
    σ_r = H[0]
return σ_l, σ_r
```

In algorithm 6 for each $e^- \in E^-$ the tree H is traversed to find the closest matches in both directions, which due to the balancing and sorting only requires $\mathbf{O}(\log l)$ for each edge, or $\mathbf{O}(k \log l)$ in total. By traversing H and memorizing the left and right bounds, the closest value is found at the latest when reaching the leaves of the tree. Note that special care has to be taken with

respect to circularity: If an element is larger than the rightmost element in the tree, then its closest edge in clockwise direction is given by the leftmost element (last lines in algorithm 6).

To conclude, traversing the tree for all incoming edges requires $\mathbf{O}(k \log l)$ and together with the tree construction leads to $\mathbf{O}\big((k+l)\log l\big)$. The procedure yields a collection of $2k$ tuples of incoming edges matched with their respective closest outgoing edges. Note that outgoing edges e^+ can be the angle-wise closest edge for multiple e^-, so any e^+ can appear multiple times or not at all. □

B.2 Sorted triple construction

These tuples are now amended with a deviation value δ that is initialized to zero. The triples are of the form (i, j, δ), where $i \in [1..k]$ is the index of the incoming edge and $j \in [1..l]$ the index of the outgoing edge. Since the algorithm described in subsection B.1 yields $2k$ tuples, initially there are $2k$ triples with $\delta = 0$. For simplicity of notation, we assume that the edges correspond to *unit* vectors in a 2-dimensional Euclidean space, such that the angle between e^- and e^+ can simply be denoted by $\langle e^-, e^+ \rangle$.

Furthermore, the triples are sorted by their preliminary distance d defined as

$$d(i, j, \delta) := D[e_i^-] + c_a(\langle e_i^-, e_{j+\delta}^+ \rangle) \ ,$$

initially corresponding to the distance of the incoming edge plus the angle cost between incoming edge i and outgoing edge j ($\delta = 0$), later to a neighboring outgoing edge with index $j + \delta$. Once the optimal preliminary distance $d(i, j, \delta)$ is known for $e_{j+\delta}^+$, only its edge cost $c(e_{j+\delta}^+)$ must be added to $d(i, j, \delta)$ to yield the distance of $e_{j+\delta}^+$ to the source as desired. Formally,

$$D[e_{j+\delta}^+] = D[e_i^-] + c_a(\langle e_i^-, e_{j+\delta}^+ \rangle) + c(e_{j+\delta}^+) = d(i, j, \delta) + c(e_{j+\delta}^+) \ .$$

Sorting all $2k$ triples requires $\mathbf{O}\big(k \log k\big)$. The sorted triples are put on a stack S, such that the first element corresponds to the triple with lowest temporary distance value $d(i, j, \delta)$.

B.3 Cost update algorithm

First, all triples on the stack are used to update the outgoing edges with the according preliminary cost:

$$\forall (i, j, 0) \in S : \quad D[e_j^+] \leftarrow \min\{D[e_j^+], d(i, j, 0) + c(e_j^+)\} \tag{B.1}$$

Since these are valid combinations of in- and outgoing edges, $D[e_j^+]$ can only be overestimated in the update in Equation B.1, but never underestimated. Following this first pass over the stack ($\mathbf{O}(k)$), algorithm 7 further updates the outgoing edges with their optimal costs. Starting from the cheapest combination on the stack, the neighboring edges are updated until encountering one with lower distances, i.e. where the update is unnecessary. Figure 25 shows an example application of algorithm 7: The top triple is retrieved from the stack, and the corresponding edge combination is updated. As long as the update is successful, i.e. the new distance is lower than the previous distance, δ is increased by 1 (clockwise updates) or decreased by 1 (counter-clockwise updates). An indicator array M^+ marks the edges that have been updated by a clockwise update operation, and equivalently M^- marks whether the edge has been updated by a counter-clockwise update. The intuition is that if an edge has already been updated by a

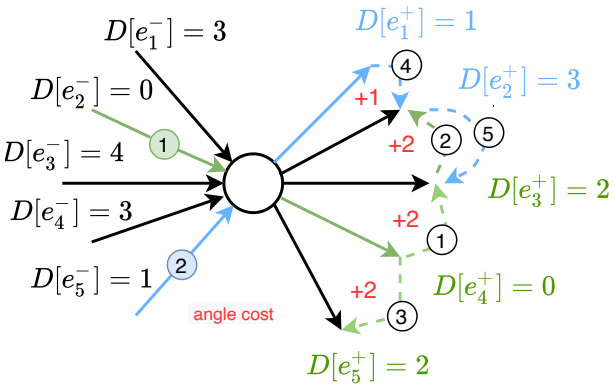


Figure 25: Angle update algorithm: First, the triple $(2, 4, 0)$ coloured in green is updated since it is the triple with lowest preliminary distance d. Furthermore, the neighboring outgoing edges are updated in counter-clockwise direction ①,②, until reaching an edge with lower or equal current distance. Here, $D[e_2^+] = 3 = D[e_4^+] + 1 + 2$ fulfills the stop criterion. Consequently, any further edge in counter-clockwise direction would be of shorter or equal distance if it is updated by the predecessor edge of e_2^+ instead of the predecessor of e_4^+. Thus, it is continued in clockwise direction and e_5^+ is updated ③. Only then the algorithm moves to the next triple on the stack, $(5, 1, 0)$. Since $D[e_1^+] + 1 < D[e_2^+]$, the distance and predecessor of e_2^+ is updated ④. The update with e_5^- stops at e_3^+ because the current preliminary distance is lower than the update ⑤. All other triples on the stack are checked but will not have further effects.

clockwise update, then another triple with positive δ will not improve over it because the triples on the stack are sorted and the cost function is linear. This reasoning will be proved formally in the following.

Algorithm 7: Optimal cost edge assignment

```
Input: S, D, d
// Initialize M^- and M^+ defining which edges where updated already
M^-[j] = M^+[j] = 0  ∀j = 1..l
while not (S.isEmpty()) do
    // Get next best triple from stack
    i, j, δ = S.pop()
    γ = (j + δ)  mod l
    // Update distance and predecessor
    if d(i, j, δ) ≤ D[e_γ^+] − c(e_γ^+) then
        D[e_γ^+] = d(i, j, δ) + c(e_γ^+);
        P[e_γ^+] = e_i;
        if δ ≥ 0 then
            M^+[γ] = 1
        if δ ≤ 0 then
            M^-[γ] = 1
        // Add next closest angle in positive direction if not updated
           yet
        if δ ≥ 0 and M^+[γ + 1] = 0 then
            S.push((i, j, δ + 1))
        // Add next closest edge in negative direction if not updated yet
        if δ ≤ 0 and M^-[γ − 1] = 0 then
            S.push((i, j, δ − 1))
end
```

Theorem B.3. *algorithm 7 takes* $\mathbf{O}(k + l)$ *steps.*

Proof. The runtime is computed by considering the number of triples that can at most be added to the stack. In order to add a triple to S, there must be an update operation, leading to a *marking of the of the updated edge*, i.e. $M^+[j] = 1$ or $M^-[j] = 1$. Note that except for iterations where $\delta = 0$ (the initial triples on the stack), the edge is always unmarked beforehand: By the

structure of a stack, the retrieved element is always the most recently added one, and throughout the algorithm a triple is only added if it was not marked in this direction before.

Therefore, any outgoing edge appears in a triple on the stack at most twice apart from the initial triples: Once with $\delta > 0$ and once with $\delta < 0$. After the first consideration of such a triple, it will be marked in the corresponding M, and no further triples referring to this edge can be added. Consequently, only $2l$ triples can be added besides the $2k$ triples initially on the stack, leading to $\mathbf{O}(k + l)$ iterations. □

Theorem B.4. *algorithm 7 updates all edge distances of $e^+ \in E^+$ correctly and assigns them an optimal predecessor, if the angle cost function is linear and monotonically increasing.*

Proof. Let j^* denote the index of any outgoing edge, and let i^* be the index of its optimal predecessor, for simplicity assumed to be unique. If there is a second predecessor edge leading to the same distance, it follows that one of either will be found. If $(i^*, j^*, 0)$ is a triple on the stack, then clearly this triple will be encountered in the update pass before starting algorithm 7, so the distance will be updated with i^* correctly assigned as the predecessor. As the other distances are larger and the distance is never underestimated, no other update in algorithm 7 will change the assignment.

On the other hand, if $(i^*, j^*, 0)$ is not an initial triple, then it remains to show that a triple $(i^*, \hat{j}, \sigma)$ with $j^* = \hat{j} + \sigma$ will be pushed to the stack at some point. Observe that for sure two triples exist on the stack containing i^* as the incoming edge, since in the triple-creation process the two outgoing edges with closest angle are assigned for each incoming edge. Consider the triple $(i^*, \hat{j}, 0)$ containing i^* with its optimal outgoing edge $\hat{j}$ *in the direction towards* $e^+_{i^*}$. The following observation is sufficient to show that $D[e_{j^*}]$ is updated correctly and $e^-_{i^*}$ is chosen as its predecessor.

Lemma B.5. *All edges with index j, $\hat{j} \leq j \leq j^*$, are assigned $e^-_{i^*}$ as its predecessor and the distance is updated correctly as $D[e_j] = d(i^*, j, 0) + c(e^+_j)$.*

Proof. We prove the lemma by induction on k for $0 \leq k \leq j^* - \hat{j}$ or $j^* - \hat{j} \leq k \leq 0$, dependent on whether $\hat{j} < j^*$. Let $j = \hat{j} + k$. We claim as the base case that for $k = 0$ then the edge $e^+_{\hat{j}}$ is assigned $e^- i^*$ as its predecessor. For the sake of contradiction, assume that when $(i^*, \hat{j}, 0)$ is retrieved from the stack, the distance $d(i^*, \hat{j}, 0) + c(e_{\hat{j}})$ is larger than the current distance $D[e^+_{\hat{j}}]$, and thus no further triples with i^* are added to the stack. Say $e^-_{\hat{i}}$ is the current predecessor for $e^+_{\hat{j}}$, such that

$$D[e^-_{\hat{i}}] + c_a(\langle e^-_{\hat{i}}, e^+_{\hat{j}} \rangle) < D[e^-_{i^*}] + c_a(\langle e^-_{i^*}, e^+_{\hat{j}} \rangle) \tag{B.2}$$

With a linear and monotonically increasing angle cost function, this however leads to the following inequalities

$$\begin{aligned} & D[e^-_{\hat{i}}] + c_a(\langle e^-_{\hat{i}}, e^+_{j^*} \rangle) \\ \leq\ & D[e^-_{\hat{i}}] + c_a(\langle e^-_{\hat{i}}, e^+_{\hat{j}} \rangle) + c_a(\langle e^-_{\hat{j}}, e^+_{j^*} \rangle) \\ \underset{B.2}{<}\ & D[e^-_{i^*}] + c_a(\langle e^-_{i^*}, e^+_{\hat{j}} \rangle) + c_a(\langle e^-_{\hat{j}}, e^+_{j^*} \rangle) \\ =\ & D[e^-_{i^*}] + c_a(\langle e^-_{i^*}, e^+_{j^*} \rangle) \end{aligned}$$

The last equality follows from linearity together with the fact that $\hat{j}$ is the optimal edge in the direction of j^*, such that the angles indeed add up.

This derivation contradicts the fact that i^* is the index of the optimal predecessor of $e^+_{j^*}$, since the distance with $e^-_{\hat{i}}$ as its predecessor is lower. Thus, a contradiction to the assumption is reached and it can be concluded that $e^+_{\hat{j}}$ is indeed updated with i^*, and consequently the triples $(i^*, \hat{j}, -1)$ and $(i^*, \hat{j}, 1)$ are added to the stack if they are not marked. Therefore, no such predecessor $e^-_{\hat{i}}$ exists, and thus $e^+_{\hat{j}}$ was never updated except for the pass over the stack in Equation B.1. It follows that $M^+[\hat{j}] = M^-[\hat{j}] = 0$ before the update with $\hat{i}$.

As the **induction step**, assume now that $\hat{j} < j^*$, so any relevant triple is described by $(i^*, \hat{j}, k)$, with $0 \leq k \leq j^* - \hat{j}$. The other case ($\hat{j} > j^*$) is analogous with $\delta < 0$. First, for any edge $e^+_{\hat{j}+k}$ it holds that $M^+[\hat{j} + k] = 0$. The only case where $M^+[\hat{j} + \gamma] = 1$ is if the edge was updated before by a triple with $\delta \geq 0$. By induction, the previous edge $e^+_{\hat{j}+k-1}$ was unmarked before and never updated, so no triple with $\delta > 0$ could have passed this edge and updated the following one. Thus, the triple $(i^*, \hat{j}, k)$ is clearly pushed to the stack after the update of $e^+_{\hat{j}+k-1}$. When retrieved from the stack, the same argument as above holds: If $e^+_{\hat{j}+k}$ was assigned another predecessor $e^+_{\hat{i}}$ with a better distance, then the equation B.2 and the subsequent inequalities prove that $e^+_{\hat{i}}$ would also be a superior predecessor for $e^+_{i^*}$, contradicting the assumption. Thus, the lemma holds for all k. □

Together, we conclude that every outgoing edge $e^+_{\hat{j}+k}$, $0 \leq k \leq j^* - \hat{j}$ is assigned i^* as the optimal predecessor, including $e^+_{j^*}$ itself. After encountering all triples from the stack, any outgoing edge is updated eventually with its optimal distance, concluding the correctness of algorithm 7. □

In total, the runtimes amount to $\mathbf{O}\big((k+l)\log l\big)$ for finding the angle-closest edges, plus $\mathbf{O}\big(k \log k\big)$ to sort the triples, and finally $\mathbf{O}\big(k+l\big)$ for algorithm 7. Together, all outgoing edges of a vertex can be updated in $\mathbf{O}\big((k+l)\log kl\big)$ steps, which implies the runtime stated in theorem B.1.

C Diverse route alternatives - Experiments

C.1 Parameters

Suitable parameters for the comparison of diversity and cost of KSP outputs were found experimentally. The following parameters refer to **instance 1** with 20m resolution.

- **find-ksp-max:** We use 8, 12, 14 and 18 cells as the minimal maximum distance of the next path to all $k-1$ previous ones.
- **find-ksp-mean:** 4, 6, 8 and 10 cells average distance between the new path and the average path were required.
- **greedy-set:** 0.2, 0.3, 0.4, 0.6, 0.8, 1.0 were used as the maximum intersection over union.
- **k-dispersion:** For the Jaccard distance, all paths with cost less than 100.5% or 101% of the optimal path cost are considered and the most diverse set is used. For the Yau-

Hausdorff distance, only the paths with less than 100.25% or 100.5% of the resistances of the least cost path were taken into account in a feasible processing time.

- **corridor-penalizing:** A penalty of at most 1% is added in descending height the further a cell is from the $k-1$ previous paths. The penalty decreases linearly with the distance, until reaching zero in a distance of 10, 20, 40 or 60 cells from the previous paths.

C.2 Qualitative results

Figure 26, Figure 28, Figure 27 and Figure 29 show further qualitative results for the computation of 5 diverse alternative routes in **Instance 1**. Clearly, the paths in Figure 28 are most diverging, while for other methods the routes are very close to each other at low thresholds. Algorithms that minimize the intersection (e.g. Figure 27) obtain both closely aligned and completely diverse paths. Overall, the outputs are strongly dependent on a good parameter calibration.

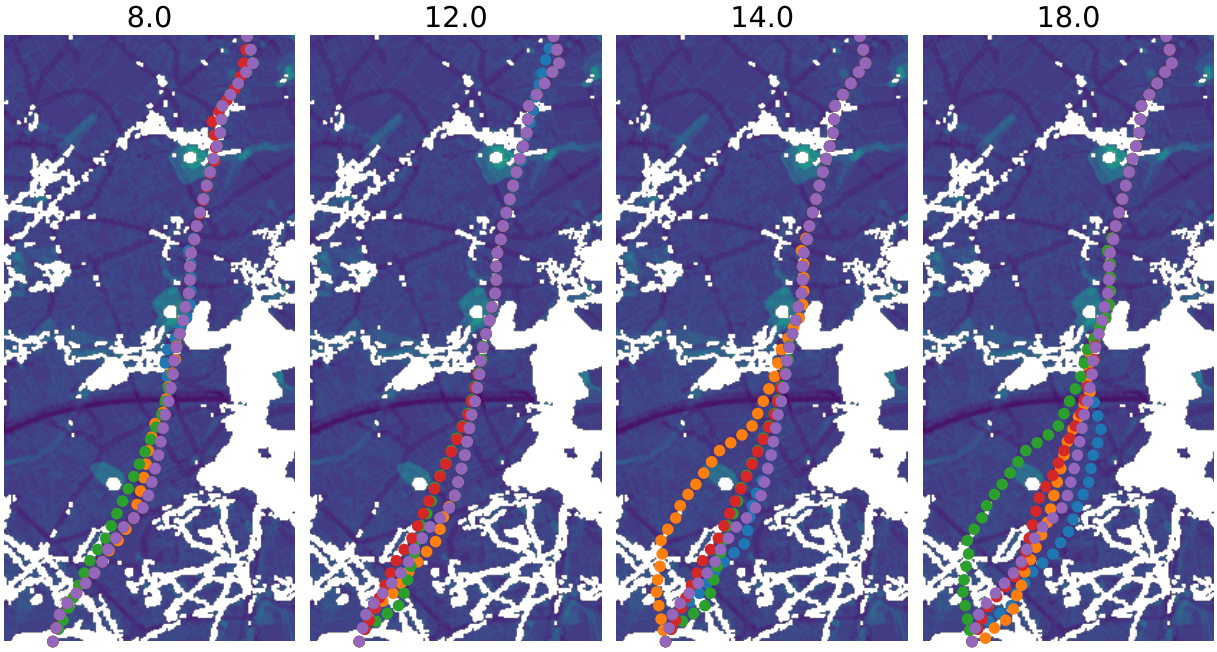


Figure 26: Qualitative evaluation of the **find-ksp-max** method

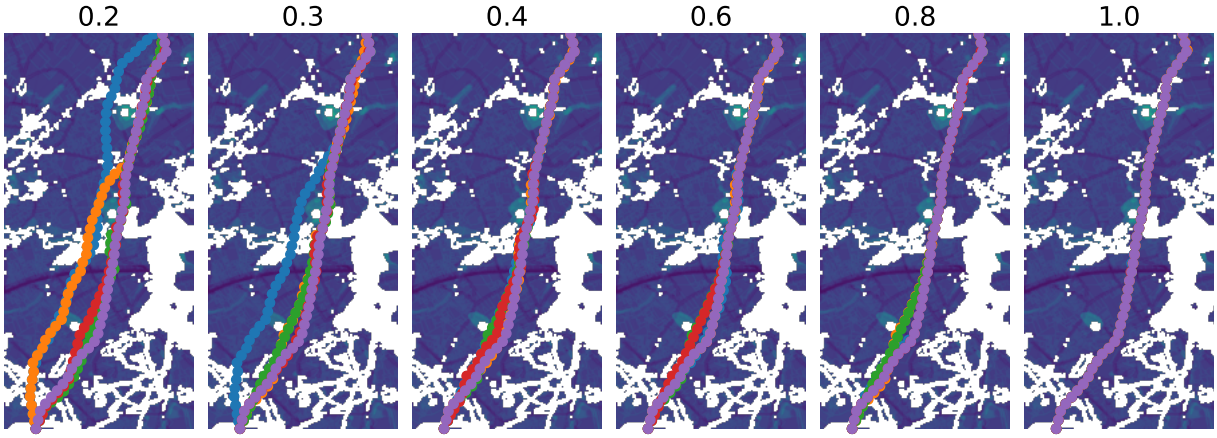


Figure 27: Qualitative evaluation of the **greedy-set** method

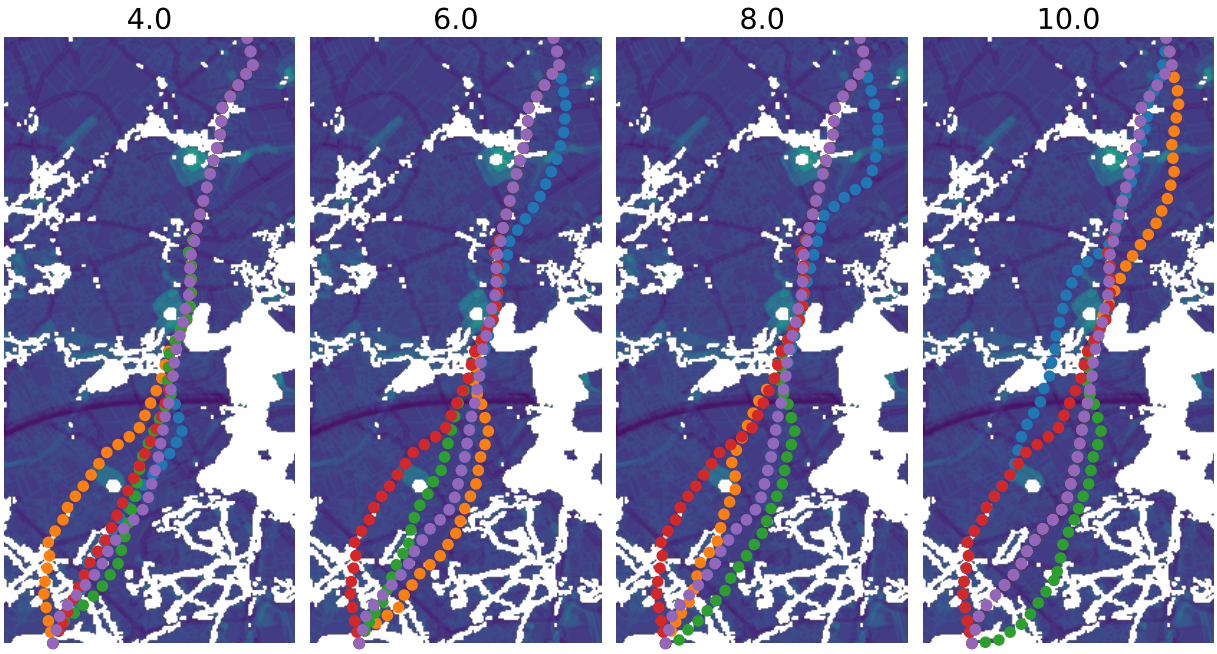
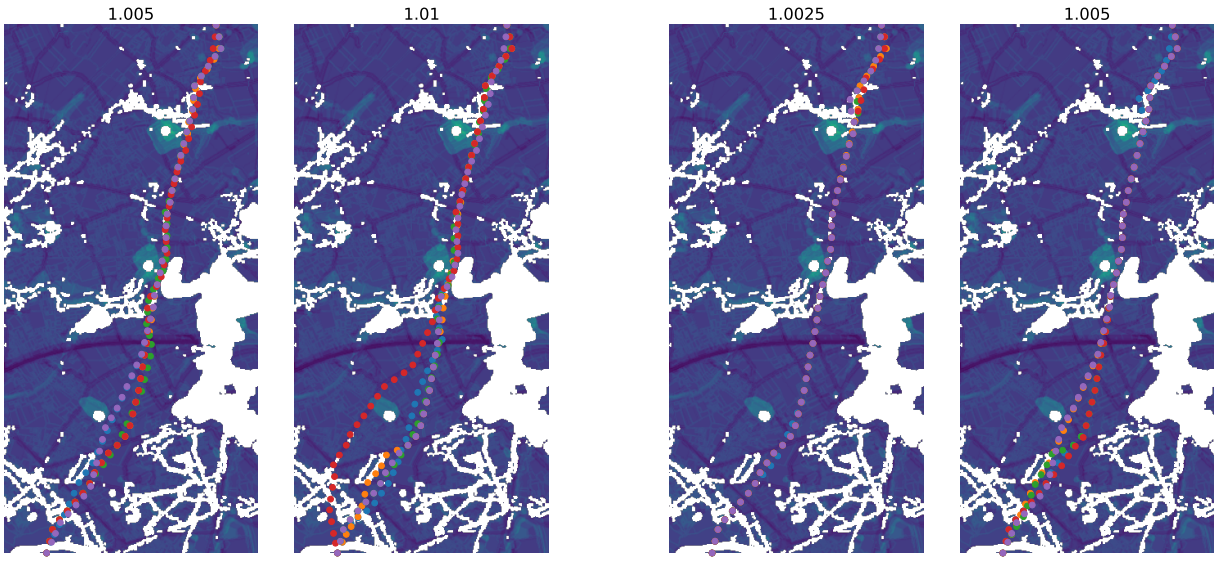


Figure 28: Qualitative evaluation of the **find-ksp-mean** approach



(a) Jaccard distance

(b) Yau-Hausdorff distance

Figure 29: Qualitative evaluation of the **k-dispersion** algorithm

Secondly, in section 5 we focus on one instance, **Instance 1**, because it is the smallest instance and consequently the analysis could be executed with 20m resolution in sensible time, even though more than 20 times 5 alternatives have to be computed. However, we were interested to analyze whether the results are strongly dependent on the instance. We therefore also executed the analysis for **Instance 2** and show the main results in Figure 30. The same conclusions as for **Instance 1** can be drawn: **greedy-set** and **k-dispersion (Jaccard)** clearly outperform other methods with respect to the Intersection over Union or Jaccard metric, whereas **corridor-penalizing** and **find-ksp-mean** maximize the maximum and average Euclidean distance respectively.

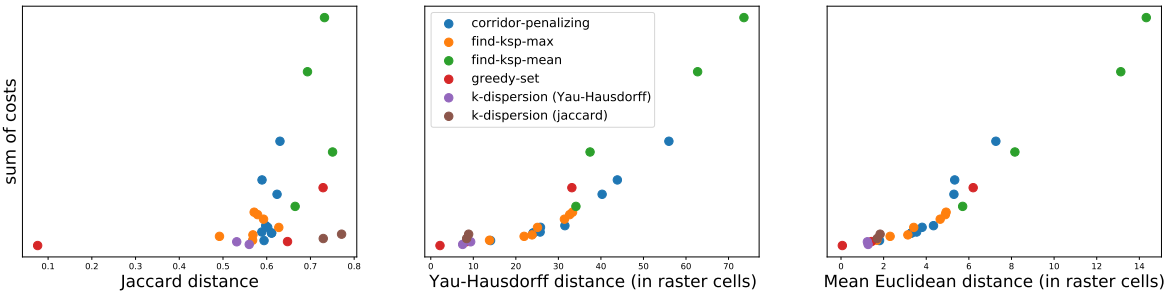


Figure 30: Cost-diversity trade-off for **Instance 2**.

D Sensitivity analysis

Different visualization methods were explored to show the sensitivity to different parameter configurations. For example, the layer weights used in the initial data pre-processing (subsection 3.1) can be varied and the difference in output can be observed. In addition, the pareto-optimal paths with respect to various cable weights and angle weights can be computed. The outputs can be visualized as a distribution of paths across the instance in order to show the most robust routes that optimize the path in most configurations.

Figure 31 shows the distribution of paths across **instance 2** (20m resolution) as computed in 71 configurations: 1 is the optimal path with the given input configuration, 24 paths are pareto-optimal paths computed with the combination of 6 cable weights and 4 angle weights, then the weights of 12 geographic layers were set to 3 different weights, and last 10 alternatives were computed with the **corridor-penalizing** method. This yields $1 + 24 + 36 + 10 = 71$ paths that are spread across the instance. Figure 31 shows the resistances (black: forbidden region, dark to white), and the paths coloured by their occurrence: The green path appears in many configurations, while the orange/red ones are output rarely. In general Figure 31 shows that for this instance the output is rather robust to the input parameters, since there is a clear optimal route and surprisingly few variation considering the large number of 71 paths.

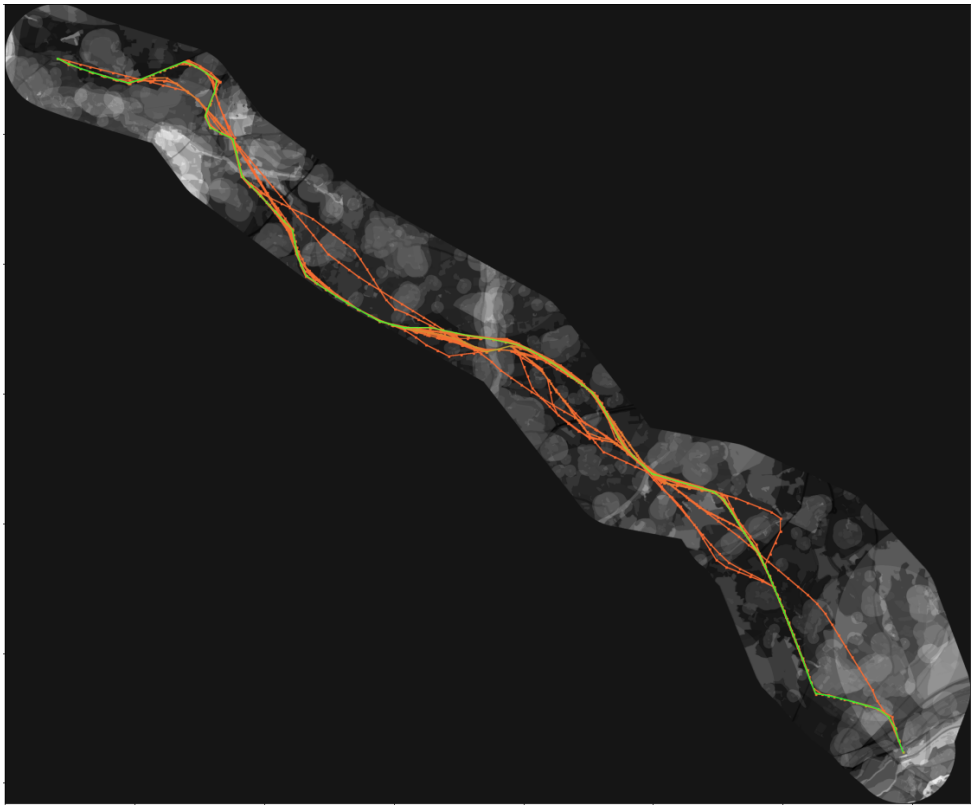


Figure 31: Sensitivity analysis visualized as a distribution of paths across the instance. black: forbidden region, dark to white: lower to higher resistance