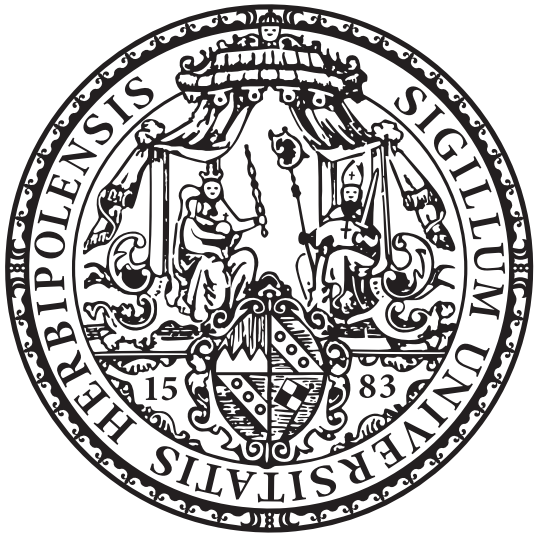
Practical Course Report

Accelerated Update Procedures for the Angle Penalised Shortest Walk Problem in Infrastructure Planning

Samuel Wolf

Date of Submission: September 10, 2024
Advisor: Prof. Dr. Alexander Wolff



Julius-Maximilians-Universität Würzburg
Lehrstuhl für Informatik I
Algorithmen und Komplexität

Contents

1 Introduction

Infrastructure planning, including the design of railroads, streets, and transmission lines, is a multifaceted endeavour that strives to optimise many, sometimes conflicting, objectives on extensive geographical regions. Particularly in the case of power-line planning the data sets frequently are major parts of countries making computers aiding the design process indispensable. Objectives such as monetary, environmental, and social costs are modelled as *cost surfaces*, where each geographical coordinate corresponds to a cost computed according to these factors.

Aside from geographically dependent costs, there are also technical costs that arise when building the infrastructure. For instance in transmission line planning, in which two geographical points need to be connected via a transmission line consisting of pylons and cables, consecutive pylons that deviate from a hypothetical straight line invoke usually higher technical costs varying on the angle of deviation such that high angles of deviation should be avoided if possible. The raster-based algorithm by Wiedemann and Adjiashvili [WA22] and the vector-based algorithm by Wolf [Wol21] both aim to find a minimum angle cost path for transmission lines. The algorithms differ in the nature of the expected input where Wiedemann and Adjiashvili expect a discrete cost surface that is rastered similar to pixels in an image while Wolf works on a cost surface that is a polygonal subdivision of the plane into polygonal regions each having a cost assigned to it.

Both algorithms build an auxiliary graph based on their respective inputs, where a vertex corresponds to a potential pylon and an edge constitutes a potential transmission cable between two consecutive pylons. The costs of building a pylon and placing the cable is mapped to the weight of the directed edges of the graph, whereas the costs based on the deviation angle is described by an *angle cost function*. In the raster-based approach vertices are usually placed in the center of each raster cell. In contrast, the vector-based approach by Wolf places vertices on the boundary of weighted regions inspired by the approximation algorithm for the *Weighted Region Problem* by Lanthier et al. [LMS01]. This auxiliary graph is then used to explore an implicit line graph to find suitable edge pairs of incident edges at a vertex with respect to the inflicted costs by the angle cost function. The approach of exploring the line graph also differs in both approaches. Wiedemann and Adjiashvili tackle the exploration of the implicit line graph with an adapted version of the Bellman-Ford algorithm, while Wolf uses a modification of the $A^\star$ algorithm.

In both procedures, the update step in which predecessors are determined is the dominating factor with respect to the asymptotic runtime, assuming the update step is implemented in a straightforward and naive way. In fact, Wolf [Wol21] shows in his experimental analysis that the update process contributes more to the runtime than the graph construction step and the maintenance of the $A^\star$ approach together even when the number of potential pylons on the border of the regions is small. A similar behaviour is also expected for the raster-based approach as the creation of the auxiliary graph and the maintenance of the Bellman-Ford-like adaptation is simpler and is therefore expected to produce less overhead. Symmetrically to the number of potential pylons on the border

of a region, the size of the neighbourhood, determined by the granularity of the rasters, is a important parameter that influences the runtime.

Restricting the angle cost function to convex functions, Wiedemann and Adjiashvili provide an accelerated update procedure with log-linear runtime complexity. This procedure requires book keeping with several datastructures and is consequently implementation-heavy with overhead that is hidden in the asymptotic log-linear time complexity. As it was devised within the Bellman-Ford framework, their algorithm proves unsuccessful when adapting it for Dijkstra-based frameworks due to the inherently different perspectives both frameworks take to find shortest-paths.

Contributions. Due to the heavy book keeping of the update procedure proposed by Wiedemann and Adjiashvili we aim to find a simpler method in terms of implementation effort and overall practical performance. Moreover, since the Dijkstra-based framework by Wolf is lacking a better update algorithm than the naive approach, we want to fill this gap.

- We provide a concrete example of a walk that is cheaper than any path for the angle penalised shortest walk problem with a convex angle cost function, showing that walks are indeed a possibility, when using a convex angle cost function.
- We propose a new update method that is suitable for both the Bellman-Ford derivative by Wiedemann and Adjiashvili and for the modified Dijkstra approach by Wolf which does not require extensive book keeping at the cost of restricting the angle cost function to linear functions.
- We show that log-linear time update algorithms are worst case optimal under the algebraic computation tree model.
- We compare the performance of the update procedure for convex angle cost functions by Wiedemann and Adjiashvili with the performance of our new method in an experimental analysis.

Organisation. We first formalise the aforementioned problem setting and give an overview of how the Bellman-Ford derivative and the modified Dijkstra approach work in Section 2. Then, in Section 3, we formalise the Update Problem arising in the Bellman-Ford approach and further describe how the update method for convex angle cost functions solves the Update Problem in log-linear time. Moreover, we propose our simplified version in Section 3.2 that solves the Update Problem and how it can be adapted for Dijkstra-based algorithms. Finally, we show in Section 3.4 that both procedures are worst case optimal within the algebraic computation tree model. We conclude our work with an experimental analysis of both methods in Section 4 and provide next steps for future work in Section 5.

2 Preliminaries

We first give a more detailed description of the problem statement, how it was tackled by Wiedemann and Adjiashvili [WA22] in a raster-based setting, and later adapted by Wolf in a vector-based scenario. Since both approaches generate a directed auxiliary graph whose vertices constitute possible pylons and edges possible transmission line cables, we formulate the problem as a graph problem and omit the process of generating the actual graph from the input data.

The problem of finding an angle penalised shortest path can naturally be translated into the *Adjacent Quadratic Shortest Path Problem* (AQSPP), in which the objective is to locate a path P within a directed graph $G = (V, E)$ that minimises the cost

$$c(P) = \sum_{j=1}^{k} w(e_j) + \sum_{j=1}^{k-1} \hat{c}(e_j, e_{j+1}),$$

where k is the length of the path P, $w : E \to \mathbb{R}_0^+$ is a function assigning edge weights to each $e \in E$ and the function $\hat{c} : E \times E \to \mathbb{R}_0^+$ charges costs between incident edges. However, the AQSPP has been shown to be NP-hard by Rostami et al. [RCH$^+$18] in directed graphs. Thus, Wiedemann and Adjiashvili relaxed the problem by asking for a *walk* that minimises the cost. The cost function $\hat{c}$ is naturally modelled as a function that takes the angle $\sphericalangle(e_1^-, e_2^+)$ between two incident edges e_1^-, e_2^+ as an input. The angle $\sphericalangle(e_1^-, e_2^+)$ measures the deviation from the straight-line extended from the incoming edge e_1^- of the shared edge as shown in Figure 1. Since a small deviation angle is desirable in this context, it is assumed that an angle cost function $\hat{c}$ takes its global minimum at a deviation angle $\sphericalangle(e_1^-, e_2^+)$ of zero.

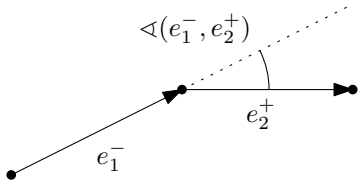


Fig. 1: Angle $\sphericalangle(e_1^-, e_2^+)$ of the incident edges e_1^-, e_2^+ measures the deviation from the straight line extended from e_1^- which serves as the argument for the angle cost function that models the inflicting costs of connecting pylons depending on the angle.

Consequently, the problem statement of finding transmission lines with respect to some angle penalisation expressed in an angle cost function is transformed to the following:

Definition 1 (Angle Penalised Shortest Walk Problem (APSWP))**.** *Given a directed graph $G = (V, E)$ with positive edge weights $w : E \to \mathbb{R}_0^+$, angle cost function $\hat{c} : [0, \pi] \to \mathbb{R}_0^+$ as well as two vertices $s, t \in V$, find a walk $P = \langle e_1 = (s, v_1), \ldots, e_k = (v_{k-1}, t)\rangle$ which minimises the cost $c(P) = \sum_{j=1}^{k} w(e_j) + \sum_{j=1}^{k-1} \hat{c}(\sphericalangle(e_j, e_{j+1}))$.*

Optimal walks, as opposed to optimal paths, indeed might happen as Figure 2 illustrates. With the convex angle cost function $c(\varphi) = \varphi^4$ and unit weights for all edges,

the walk $P = \langle v_1, v_2, v_3, v_4, v_2, v_5 \rangle$ has a cost of $c(P) = 5 + \alpha_1^4 + \alpha_2^4 + \alpha_3^4 + \alpha_4^4 = 39.26$ while the path $P' = \langle v_1, v_2, v_5 \rangle$ with cost $c(P') = 2 + \beta_1^4 = 40.94$ is more expensive.

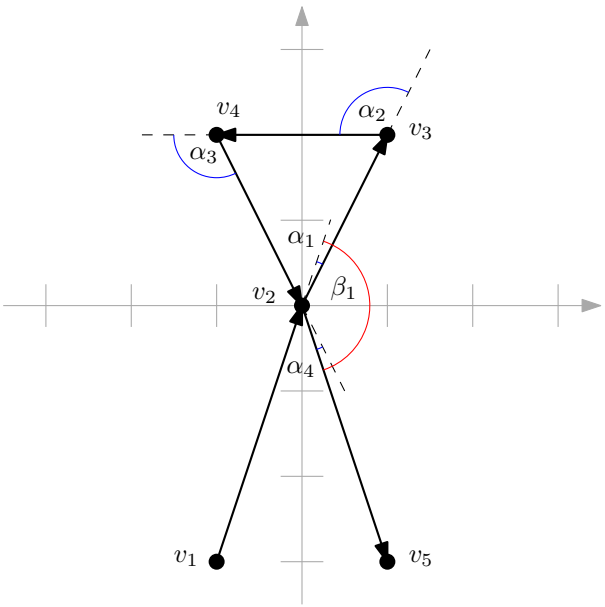


Fig. 2: In this graph, in which each edge has weight 1 and the given cost function is $c(\varphi) = \varphi^4$, the walk $P = \langle v_1, v_2, v_3, v_4, v_2, v_5 \rangle$ is cheaper than the path $P' = \langle v_1, v_2, v_5 \rangle$. With angles $\alpha_1 = \alpha_4 = 0.003$, $\alpha_2 = \alpha_4 = 2.03$, and $\beta_1 = 2.50$.

Though, in the special case of concave angle cost functions Wiedemann and Adjiashvili [WA22] prove that there always exists a *path* that minimises the total costs. Linear angle cost functions are therefore particularly suitable since they are concave functions.

Conceptually, the APSWP can by solved by building a line graph in which the original graph $G = (V, E)$ is translated into an auxiliary graph $L(G) = (V', E')$ such that $V' = E$, i.e., each edge of the original graph becomes a vertex and two vertices $e, f \in V'$ are adjacent if and only if the corresponding edges in G are incident to the same vertex. Edges in $L(G)$ therefore describe possible adjacent edge pairs in G that might be traversed in an optimal walk. Thus, we can give an edge $ef \in E'$ the corresponding angle cost $\hat{c}(\sphericalangle(e, f))$ and each vertex $e \in V'$ the original weight $w(e)$. Running a suitable shortest path algorithm on $L(G)$ will then yield a solution.

However, the increased size of $L(G)$ to $|E|$ vertices and $\mathcal{O}(\sum_{v \in V} \text{indeg}(v) \cdot \text{outdeg}(v))$ edges renders an explicit construction impractical on large data sets. Wiedemann and Adjiashvili addressed this issue by modifying the Bellman-Ford algorithm to leverage on the line graph in an implicit manner, thereby avoiding the significantly increased space requirement.

In the original Bellman-Ford algorithm [Bel58, FF56] we solve a recursive formulation of the shortest s-t-path problem via dynamic programming. For a given graph with edge weights, we define $BF(v, i)$ as the cost of a shortest s-v-path of length at most i which

we can calculate by the following equation:

$$BF(v,i) = \begin{cases} 0 & \text{if } v = s \\ \infty & \text{if } v \neq s \text{ and } i = 0 \\ \min\limits_{(u,v)\in E}\{BF(v,i-1), BF(u,i-1)+w(u,v)\} & \text{otherwise.} \end{cases}$$

Wiedemann and Adjiashvili [WA22] take the perspective of the edges in the graph instead of the vertices as would be the case when employing a line graph. Thus, the recurrence $BF_2(e,i)$ for the APSWP is defined as the shortest walk from source s to edge e of length at most i such that

$$BF_2(e=(v,w),i) = \begin{cases} w(e) & \text{if } v = s \\ \infty & \text{if } v \neq s, i = 0 \\ \min\limits_{f=(u,v)\in E}\{BF_2(e,i-1), & \text{otherwise.} \\ \qquad BF_2(f,i-1) + w(e) + \hat{\mathrm{c}}(\sphericalangle(f,e))\} & \end{cases}$$

In this way Wiedemann and Adjiashvili utilise the structure of the underlying line graph but only maintain information for each edge in G, hence evading the memory issues of explicitly constructing a line graph. Wolf [Wol21] applies the idea of using the underlying line graph implicitly to the $A^\star$ algorithm, a variant of Dijkstra's algorithm, by maintaining a priority queue of edges e and their costs of a current shortest s-e walk. The priority queue is initialised with edges $(s,v) \in E$ and their corresponding cost $w((s,v))$ as priority. During each update iteration, the cheapest edge e is retrieved and all incident edges of e are explored or updated, if needed, according to accumulated costs, weights, and angle costs, thus also avoiding an explicit construction of $L(G)$.

Note that, while both approaches remedy the memory issues, the increased size of the line graph $L(G)$ is effecting the runtime of both algorithms. Consider, for example, a vertex $v \in V$. In the line graph $L(G)$ vertex v contributes $\mathrm{indeg}(v) \cdot \mathrm{outdeg}(v)$ many edges to $L(G)$, each representing a different way to traverse v. Thus, the naive way of calculating the minimisation step is to mimic Bellman-Ford and consider each pair of incident edges in E amounting to $\mathcal{O}(\sum_v \mathrm{indeg}(v)\cdot\mathrm{outdeg}(v))$ time per recursive step. This can also be done in the Dijkstra-based approach of Wolf, resulting in the overall runtime of $\mathcal{O}(\log(E)\sum_v \mathrm{indeg}(v)\cdot\mathrm{outdeg}(v))$ for all relaxation steps, where the logarithmic factor stems from the maintenance of the priority queue. Thus, the dominating factor in the time complexity of both algorithms is this minimisation or relaxation, motivating a deeper investigation into the improvement of this process.

A promising way to obtain algorithms with better runtime is to assume certain properties of the given angle cost function, which can be used to reduce the number of pairs of edges that need to be considered during the relaxation or minimisation. As previously noted, considering a linear angle cost function is sensible, when seeking a path instead of a walk. This consideration is particularly relevant in the context of transmission lines, which inherently constitute paths.

3 Accelerated Update Procedures

We first describe the accelerated update procedure by Wiedemann and Adjiashvili [WA22] for convex and monotonically increasing angle cost functions and then provide an alternative procedure for linear angle cost functions. Later we explain how we can use our version for linear angle cost functions as an update mechanism for the $A^\star$ method. Finally, we show that these methods are indeed asymptotically optimal under the algebraic computation tree model. We refer to the UPDATE PROBLEM as the following problem statement:

Definition 2 (UPDATE PROBLEM)**.** *Let a directed weighted graph $G = (V, E)$ and an angle cost function $\hat{c}$ as defined in Definition 1 be given. Further, let $v \in V$ with incoming edges $e_1^-, \ldots, e_k^-$ and outgoing edges $e_1^+, \ldots, e_\ell^+$ be given, where the optimal distance $D[e_i^-] = BF_2(e_i^-, h-1)$ to every incoming edge e_i is already known. Find the optimal distances $BF_2(e_j^+, h) = \min\{BF_2(e_j^+, h-1), D[e_j^+]\}$ to all outgoing edges with*

$$D[e_j^+] = \min_i \{w(e_j^+) + D[e_i^-] + \hat{c}(\sphericalangle(e_i^-, e_j^+))\}.$$

Note that the UPDATE PROBLEM is precisely the formulation of the minimisation step in the recursive formula of BF_2 described in Section 2. For the sake of simplicity we will focus on finding the optimal predecessor e_i^- for an incoming edge e_j^+ that minimises the term $D[e_i^-] + \hat{c}(\sphericalangle(e_i^-, e_j^+))$. Subsequently, the weight $w(e_j^+)$ does not matter when finding the optimal predecessor e_i^- and is omitted in further considerations.

3.1 The Update Method by Wiedemann and Adjiashvili

The general idea behind the update method by Wiedemann and Adjiashvili that solves the UPDATE METHOD is that the incoming edges partition the cyclic domain of angles $[0, 2\pi)$ with respect to a center vertex v and a reference direction $\vec{d}$ into intervals in which an incoming edge e_i^- is the optimal predecessor for any outgoing edge e_j^+ whose angle with respect to $\vec{d}$ is in the corresponding interval. For arbitrary angle cost functions, an incoming edge e_i^- might have multiple disconnected intervals in which e_i^- is the optimal predecessor for outgoing edges that leave v in an angle contained in these intervals. In contrast, when the angle cost function $\hat{c}$ is monotonically increasing and convex, the interval of angles in which an incoming edge e_i^- is an optimal predecessor is a single connected sub-interval of the cyclic interval $[0, 2\pi)$. See Figure 3 for an example of a partitioning where the angle cost function is monotonically increasing and convex. Here, the interval in which e_2^- is an optimal predecessor is neighboured by the interval of e_5^- in counterclockwise direction and by the interval of e_4^- in clockwise direction. The thick lines between the intervals correspond to the *intersections* in which both incoming edges that correspond the neighbouring non-empty intervals would be optimal.

More precisely, Wiedemann and Adjiashvili [WA22] define a clockwise cyclic angle domain $[0, 2\pi)$ with respect to an arbitrary direction $\vec{d}$, in which an incoming/outgoing edge enters/leaves a vertex v with an angle $\alpha_i = \measuredangle(\vec{d}, e_i^-)/\beta_j = \measuredangle(\vec{d}, e_j^+)$. In this reference

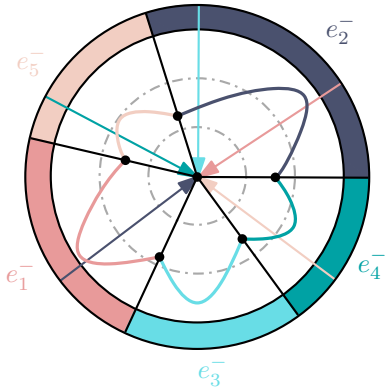


Fig. 3: Each incoming edge e_i^- corresponds to a non-empty interval of optimality coloured in the same colour as their incoming edges. The curves going towards the center are sketches of the respective angle cost functions of each incoming edge, where zero cost would be at the border of the circle defining the intervals. These functions intersect at the lines that define the intervals.

frame it holds that $\alpha_i = \beta_j$ if and only if e_i^- and e_j^+ are diametrically opposite from each other which allows for a consistent assignment of intervals of angles to incoming and outgoing edges. This setting permits Wiedemann and Adjiashvili to search for a predecessor e_i^- for an outgoing edge e_j^+ that minimises $D[e_i^+] + \hat{c}(|\alpha_i - \beta_j|_m)$, where $|x|_m = \min\{|x|, 2\pi - |x|\}$ to ensure that the input of $\hat{c}$ is within the domain $[0, \pi]$ of $\hat{c}$.

The algorithm. Wiedemann and Adjiashvili iteratively construct a datastructure that stores the intervals of angles in which incoming edges are optimal predecessors. These intervals are represented by the intersection angles of functions f_i which are defined for every incoming edge e_i^- where $f_i(\beta) = D[e_i^-] + \hat{c}(|\alpha_i - \beta|_m)$. The function f_i describes the cost for an outgoing edge e_j^+ with angle β if the incoming edge e_i^- was its predecessor. The datastructure is a balanced binary search tree $\mathcal{T}$ whose elements are triplets (x, p, q), where x is the intersection angle of f_p, f_q and key of $\mathcal{T}$, and e_p^-, e_q^- are the optimal incoming edges for outgoing edges with angles next to x in counterclockwise and clockwise direction, respectively.

At first, the incoming edges are sorted by their total cost $D[e_i^-]$. For simplicity, we assume that they are already sorted, so $D[e_1^-] \leq \cdots \leq D[e_k^-]$. The update algorithm processes the incoming edges in this order and initialises $\mathcal{T}$ by finding the first intersections between f_1 and a function f_s for the smallest $s > 1$. Note that since we are dealing with a cyclic domain, if two functions f_p and f_q intersect at one point, they also intersect at another point. If there is no function f_s for $1 < s \leq k$ that intersects f_1, then the incoming edge e_1^- is the best predecessor for all outgoing edges and the algorithm terminates. Otherwise, $\mathcal{T}$ now contains two entries for the two intersections between f_1 and f_s and the algorithm proceeds to process the remaining incoming edges e_i^- for $i = s+1, \ldots, k$ the following way: The algorithm determines the counterclockwise and clockwise neighbouring edges e_p^-, e_q^- of e_i^- that have already been processed and

currently have non-empty intervals. Wiedemann and Adjiashvili achieve this by using and maintaining an auxiliary balanced binary search tree $\mathcal{T}'$ that stores processed incoming edges e_r^- with currently non-empty intervals with key α_r. Thus, e_p^- and e_q^- can be retrieved from $\mathcal{T}'$ by finding the hypothetical predecessor and successor of e_i^- in $\mathcal{T}'$, respectively.

Since e_p^- and e_q^- are currently neighbours in $\mathcal{T}'$, their current intervals are also neighbours at an intersection angle $x_{pq} \in [0, 2\pi)$. If $f_i(x_{pq})$ is smaller than $f_p(x_{pq}) = f_q(x_{pq})$, then e_i^- currently has a non-empty interval among all processed incoming edges and needs to be inserted in $\mathcal{T}$. The edge e_i^- might completely outperform already processed incoming edges e_r^- with the consequence that the intervals associated with these e_r^- become empty. Therefore, intersections, i.e., interval borders, in $\mathcal{T}$ in (counter-)clockwise order starting from x_{pq} must be examined by testing how f_i is performing at these intersections compared to its (counter-)clockwise partners. If f_i is outperforming f_r at the counterclockwise and clockwise border of the interval of an incoming edge e_r^-, the respective intersection entries are deleted from $\mathcal{T}$ and the entry of e_r^- is deleted from $\mathcal{T}'$.

Once f_i is not outperforming the next function f_{p*} in counterclockwise and f_{q*} in clockwise direction, the intersections of f_i with f_{p*} and f_{q*} are computed and $\mathcal{T}$ is updated accordingly. Since the interval of e_i^- is now present in $\mathcal{T}$, e_i^- needs to be inserted into $\mathcal{T}'$. Wiedemann and Adjiashvili do the (counter-)clockwise walk in $\mathcal{T}$ by maintaining a sorted cyclic doubly-linked list with the triplets in $\mathcal{T}$, so there is a constant access to the predecessor and successor of an element in $\mathcal{T}$.

Finally, once all incoming edges have been processed, $\mathcal{T}$ represents the partition of the cyclic angle domain $[0, 2\pi)$ into intervals associated with incoming edges e_i^- which are optimal predecessors for outgoing edges e_j^+ that leave v in an angles contained in the respective interval. The binary tree $\mathcal{T}$ can now be used to find the optimal predecessor e_q^- for each outgoing edge e_j^+ by searching for the closest intersections (x_1, p, q), (x_2, q, r) in counterclockwise and clockwise direction, respectively.

Computing the intersection of arbitrary convex monotonically increasing functions is non-trivial. However, Wiedemann and Adjiashvili argue that this computation can be approximated efficiently by finding the angle β_j such that $\beta_j = \arg\min |x_{p,q} - \beta_j|$ by employing yet another balanced binary search tree storing the angles β_j of all outgoing edges. With this method for computing intersections, Wiedemann and Adjiashvili obtain an overall runtime of $\mathcal{O}((k + \ell) \log(k \cdot \ell))$ for their update algorithm.

3.2 A New Update Method for Linear Angle Cost Functions

As pointed out in Section 2, linear angle cost functions grant desirable paths instead of walks. Thus, we propose an update procedure using linear angle cost functions that runs in $\mathcal{O}((k + \ell) \log(k + \ell))$ time. Our algorithm has two major advantages over the algorithm of Wiedemann and Adjiashvili: First, our algorithm relies only on elementary datastructures such as arrays that are locally aligned in memory as opposed to pointer-heavy datastructures such as trees whose local memory alignment is not guaranteed. And more importantly, our approach can be adapted for the $A^\star$ method by Wolf [Wol21], a flexibility that the algorithm by Wiedemann and Adjiashvili does not possess.

Our algorithm. We exploit two properties of linear angle cost functions: We know that the angle cost of e_i^- and e_j^+ is minimal, if their end-points are diametrically opposite from each other. For linear angle cost functions we know that, if there exists a non-empty interval where e_i^- is the optimal predecessor, then it must contain this minimum value. The second property that we exploit is that we know that if an incoming edge outperforms another incoming edge in one direction in terms of angle cost once, it cannot be outperformed by the same edge later in the same direction as each angle cost function f_p shares the same slope. Note that these properties do not hold for arbitrary monotonically increasing convex angle cost functions as the slope of these functions is generally not constant.

The update method by Wiedemann and Adjiashvili described in Section 3.1 uses a two step approach: First a datastructure is created that partitions the cyclic angle domain $[0, 2\pi)$ into intervals that correspond to incoming edges that are optimal for outgoing edges leaving in an angle contained in a specific interval. Then this datastructure is used to find the optimal distance and predecessor for every outgoing edge.

Our approach combines these two steps and does not construct a datastructure but rather updates the distances of outgoing edges by performing walks in (counter-)clockwise order through an array containing representatives of the incoming edges as well as the outgoing edges for different incoming edges.

In the following, we define $\phi(e_i^-)$ as the shifted edge of e_i^- rotated by π around the vertex v which later serve as starting angle for the (counter-)clockwise walk for edge e_i^-. We also assume that the incoming edges are already sorted by their current total costs, i.e., $D[e_1^-] \leq \cdots \leq D[e_k^-]$.

We start by constructing and sorting an array $\mathcal{E} = \langle \phi(e_1^-), \ldots, \phi(e_k^-), e_1^+, \ldots, e_\ell^+ \rangle$ in clockwise order with respect to their end-point $u \neq v$ as illustrated in Figure 4a. Then we iterate over every incoming edge $e_1, \ldots, e_k$ and *expand* it. When expanding an incoming edge e_i^-, we move both in clockwise and counterclockwise direction in $\mathcal{E}$ starting from the index of $\phi(e_i^-)$ as Figure 4b shows.

When expanding, we can encounter an outgoing edge e_j^+ or some other representative of an incoming edge $\phi(e_p^-)$. In the former case, we update the total costs of the outgoing edge e_j^+ if $D[e_j^+] > D[e_i^-] + \hat{c}(\sphericalangle(e_i^-, e_j^+))$. In the latter case, we determine if $D[e_p^-] < D[e_i^-] + \hat{c}(\sphericalangle(e_i^-, \phi(e_p^-)))$, which indicates that e_i^- has been outperformed at this or a smaller angle, resulting in an immediate termination of the expansion in this direction. Otherwise, $\phi(e_p^-)$ is marked as being *absorbed* in the current walking direction and the expansion continues. If in a later iteration e_p^- is expanded, it will only expand in directions that have not been absorbed by expansions of previous iterations. To achieve this, we equip a representative of an incoming edge $\phi(e_i^-)$ with two flags that signify whether another edge has absorbed, i.e., outperformed e_i^- at the position of $\phi(e_i^-)$ previously in a particular direction. These flags can then be checked in each iteration before the expansion of an edge e_i^- occurs. Algorithm 1 gives a description of the algorithm in pseudocode.

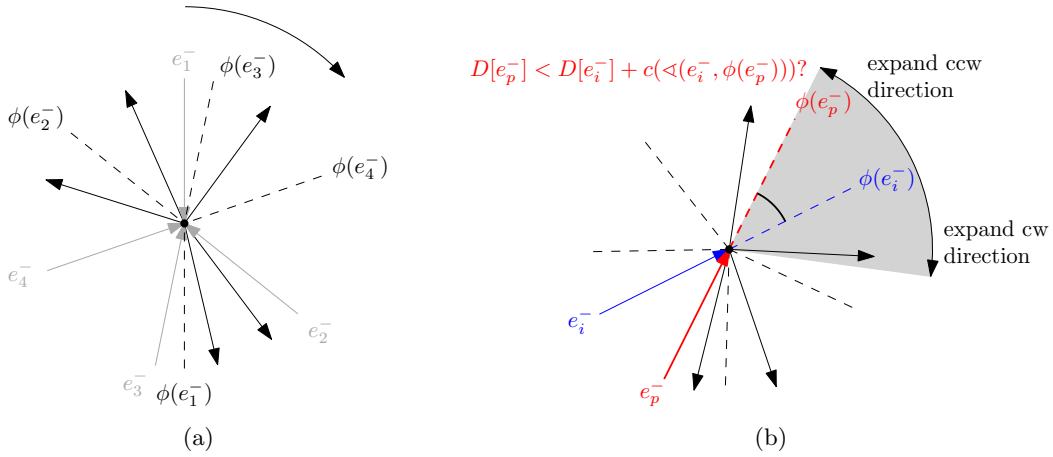


Fig. 4: Figure 4a illustrates the array $\mathcal{E}$ that contains both outgoing edges and the $\phi(e_i^-)$ edges corresponding the incoming edges. Figure 4b shows an expansion of the interval of e_i^-. In an expansion step, we move in (counter-)clockwise direction in $\mathcal{E}$ starting from $\phi(e_i^-)$ and update outgoing edges if needed or absorb other incoming edges.

Algorithm 1: Algorithm for the UPDATE PROBLEM for linear angle cost functions $\hat{c}$.

$\mathcal{E} \leftarrow \langle \phi(e_1^-), \ldots, \phi(e_k^-), e_1^+, \ldots, e_\ell^+ \rangle$; sort $\mathcal{E}$ in clockwise order
foreach $e_i^- \in \langle e_1^-, \ldots, e_k^- \rangle$ **do**
  if $\phi(e_i^-)$ **not** absorbed in clockwise direction **then**
    `expandCW(`$\mathcal{E}$, e_i^-`)`
  if $\phi(e_i^-)$ **not** absorbed in counterclockwise direction **then**
    `expandCCW(`$\mathcal{E}$, e_i^-`)`

`/* Algorithm to expand in clockwise direction. Expansion in counterclockwise direction is analogous. */`
func `expandCW(`*array* $\mathcal{E}$, *incoming edge* e_i^-`)`
  $p \leftarrow$`advanceCW(indexOf(`$\phi(e_i^-)$`))` `// advance index in clockwise direction in` $\mathcal{E}$ `and wrap around, if an end of` $\mathcal{E}$ `is reached.`
  while $\sphericalangle(e_i^-, \mathcal{E}[p]) \leq \pi$ **do**
    if $\mathcal{E}[p]$ is outgoing edge e_p^+ **then**
      $D[e_p^+] \leftarrow \min\{D[e_p^+], D[e_i^-] + \hat{c}(\sphericalangle(e_i^-, e_p^+))\}$
    if $\mathcal{E}[p]$ is shifted edge $\phi(e_p^-)$ **and** $D[e_p^-] \geq D[e_i^-] + \hat{c}(\sphericalangle(e_i^-, \phi(e_p^-)))$ **then**
      mark $\phi(e_p^-)$ absorbed in clockwise direction
    if $\mathcal{E}[p]$ is shifted edge $\phi(e_p^-)$ **and** $D[e_p^-] < D[e_i^-] + \hat{c}(\sphericalangle(e_i^-, \phi(e_p^-)))$ **then**
      break
    $p \leftarrow$ `advanceCW(`p`)`

Correctness. It suffices to show that each outgoing edge is considered at least once by some expansion and that an outgoing edge e_j^+ is covered by the expansion of its optimal predecessor e_i^- since then its distance $D[e_j^+]$ is correctly updated.

If there exists at least one incoming edge, each outgoing edge is considered at least once, as every outgoing edge will then have a closest incoming edge in the clockwise or counterclockwise direction. Since an expansion either encompasses all edges or halts due to a better alternative that would subsequently expand further in this direction, this outgoing edge must be taken into account by at least one expansion.

Now, consider the expansion of the optimal predecessor e_i^- of some outgoing edge e_j^+. Without loss of generality assume that e_j^+ is located in the right half plane spanned by e_i^-, i.e, the clockwise expansion starting from $\phi(e_i^-)$ is able to reach e_j^+. Suppose e_j^+ was not reached by e_i^-. This means that e_i^- was either absorbed in this direction by an outgoing edge e_p^- that has been previously considered or its expansion has been stopped by an outgoing edge e_q^- that was before e_j^+ in clockwise direction as illustrated in Figure 5. Due to linearity of the angle cost function $\hat{c}$, we know that once an outgoing edge is

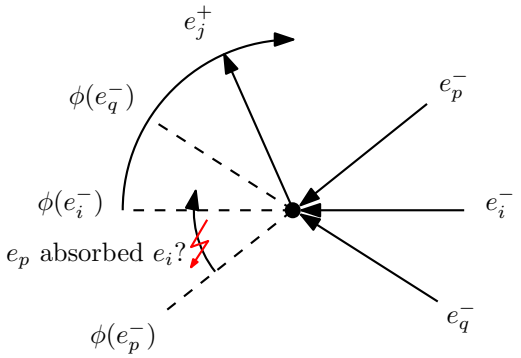


Fig. 5: The outgoing edge e_j^+ lies in the right half plane of its optimal predecessor e_i^-. Neither e_p^- could have absorbed e_i^- nor e_q^- could have stopped the expansion of e_i^- prior to reaching e_j^+ as otherwise e_p^- would have outperformed e_i^- at its minimum or e_i^- would have been worse than e_q^-, thus e_p^- or e_q^- would have been a better alternative than e_i^- which is a contradiction to the assumption that e_i^- is the optimal predecessor to e_j^+.

outperformed by another outgoing edge in a certain direction, it cannot be better later in the same direction. This implies that in both cases either e_p^- or e_q^- would have been a better predecessor than e_i^- which is a contradiction. Thus, the clockwise expansion of e_i^- takes e_j^+ into account and therefore e_j^+ is updated by its optimal predecessor.

Runtime. Sorting $\mathcal{E}$ takes $\mathcal{O}((k+\ell)\log(k+\ell))$ time which dominates the runtime as all expansions take $\mathcal{O}(k+\ell)$ time. To see this, consider an outgoing edge e_j^+ that is encountered during the expansion of e_p^- in a particular direction for the first time. With out loss of generality, assume that e_j^+ has been encountered by the clockwise expansion, i.e., e_j^+ lies in the right side of the half plane spanned by e_i^-, similar to Figure 5. Since e_i^- was not stopped by any $\phi(e_q^-)$ that lies before e_j^+ on the right side of the half plane,

all of their clockwise directions have been absorbed and will not not consider e_j^+ any more. Every other $\phi(e_p^-)$ that lies before $\phi(e_i^-)$ in the clockwise ordering and is able to reach e_j^+ is stopped by e_i^- when traversing in the particular direction since $i < p$ and $D[e_i^-] \leq D[e_p^-]$ due to the sorting of the outgoing edges by total cost, and can therefore not reach e_j^+ from this direction.

Shifted incoming edges $\phi(e_i^-)$ might be encountered more often, as multiple incoming edges that are expanding might be halted by it. However, each incoming edge expansion can be stopped at most twice, so by charging these encounters on each expansion we get that all incoming edges stop $\mathcal{O}(k)$ expansions in total. Also note that an edge $\phi(e_i^-)$ can be traversed at most once in each direction as can be seen with similar arguments to why an outgoing edge is encountered by an expansion at most twice. Thus, the entire expansion process for all incoming edges totals up to $\mathcal{O}(k + \ell)$ time, yielding an overall runtime of $\mathcal{O}((k + \ell) \log(k + \ell))$.

3.3 An Adaptation for Dijkstra-based Algorithms

It is crucial to note the distinction in the problem statement for updating outgoing edges in Dijkstra-based algorithms compared to the Update Problem. In the `relax` method for Dijkstra-based algorithms, not all incoming edges will necessarily have their lowest, i.e., optimal $D[e_i^-]$ values when updating outgoing edges.

First, note that this difference in the problem statement requires to consider update methods that operate over several relaxation steps of each incoming edge of a specific vertex v which means that the previously mentioned update methods must be able to be paused after processing an incoming edge that has been extracted by the priority queue and resumed at a later state. Thus, the previously mentioned update algorithms either fail in correctness or runtime. Wiedemann and Adjiashvili's approach, for example, assume that the angle cost functions f_i do not change in later iterations of incoming edges, thus their algorithm would yield wrong results when one tries to pause it. Note that outgoing edges are only updated once $\mathcal{T}$ has been constructed. Thus, a modification would also be required such that suitable outgoing edges are updated earlier. Our approach fails in terms of runtime as an outgoing edge can be considered $\mathcal{O}(k)$ times, yielding an unsatisfactory runtime. In particular, when expanding an outgoing edge, it might not be stopped appropriately by another incoming edge representative $\phi(e_p^-)$ as the total cost $D[e_p^-]$ might be decreased at a later stage such that its final $D[e_p^-]$ value would have stopped the expansion. However, our approach can be adapted to accommodate these requirements while maintaining the same asymptotic runtime except for the unavoidable time factor for the required maintenance of the priority queue $\mathcal{Q}$ of the Dijkstra-based algorithm.

The main idea is to use the priority queue $\mathcal{Q}$ of the Dijkstra-based algorithm to decide when to resume a relaxation step. This priority queue $\mathcal{Q}$ is an indispensable part of a Dijkstra-based algorithm which dictates the order in which the graph is explored. First, we use the total cost $D[e_{q^\star}]$ of the currently cheapest edge $q^\star$ in the priority queue to decide when to stop the expansion, as $D[e_{q^\star}]$ is a lower bound of all $D[e_p]$

values. When expanding e_i^- we pause the expansion at an edge $\phi(e_p^-)$ if $D[e_{q^\star}] < D[e_i^-] + c(\triangleleft(e_i^-, \phi(e_p^-)))$. We can only decide whether to continue the update or stop the expansion in this direction if we know whether $D[e_p^-] < D[e_i^-] + c(\triangleleft(e_i^-, \phi(e_p^-)))$. Therefore, we add an *update callback* to $\mathcal{Q}$ with priority $D[e_i^-] + c(\triangleleft(e_i^-, \phi(e_p^-)))$. If e_p^- is retrieved from $\mathcal{Q}$ before the update callback, we know that $D[e_p^-] \leq D[e_i^-] + c(\triangleleft(e_i^-, \phi(e_p^-)))$, thus we can abort the expansion of e_i^-. Otherwise, the update callback is extracted before e_p^- which results into an absorption of e_p^- by e_i^- in the direction of the expansion that caused the update callback.

Correctness. First, note that when an expansion of an incoming edge e_i^- occurs, e_i^- has its optimal cost $D[e_i^-]$. Also note that these changes still guarantee that an outgoing edge e_j^+ is eventually covered by its optimal predecessor e_i^- as the expansion in one particular direction of e_i^- still traverses a semi-circle unless it has been stopped by another outgoing edge $\phi(e_p^-)$ before. In the adapted version, the expansion is only on hold when $D[e_{q^\star}] < D[e_i^-] + c(\triangleleft(e_i^-, \phi(e_p^-)))$ and only stops if there is an edge $\phi(e_p^-)$ such that $D[e_p^-] < D[e_i^-] + c(\triangleleft(e_i^-, \phi(e_p^-)))$. Thus, by the arguments of the original algorithm, e_j^+ is reached by its optimal predecessor eventually.

Furthermore, for an outgoing edge e_j^+ it also holds that the optimal predecessor e_i^- updates e_j^+ before e_j^+ is extracted from $\mathcal{Q}$. For the moment, assume that e_i^- is the unique optimal predecessor of e_j^+ and suppose that e_j^+ was extracted from $\mathcal{Q}$ before the update callback of the expansion that covers e_j^+ has been extracted. If e_j^+ has been extracted before the update callback, then the priority of e_j^+ must be at most the priority of the update callback. Since the update callback has not covered e_j^+ yet, its priority must be smaller than $D[e_i^-] + \hat{c}(\triangleleft(e_i^-, e_j^+))$. This, however, implies that there is a better predecessor than e_i^- which is a contradiction to e_i^- being the optimal predecessor. If the optimal predecessor is not unique, then it might happen that e_j^+ is extracted before the update callback since its priority is equal to $D[e_i^-] + \hat{c}(\triangleleft(e_i^-, e_j^+))$. In this case another optimal predecessor has already updated e_j^+, so the Dijkstra-based algorithm remains correct.

Runtime. Regarding the time complexity of this adaptation, we argue that the runtime remains the same compared to its original counterpart, with the exception of the necessary update operations on the priority queue $\mathcal{Q}$. We show that the expansion in the alternated algorithm covers at most as many edges as it would have in the original version. In particular, we show that when an incoming edge is expanded, the absorption flags are set correctly.

Observe that the expansion is paused every time the expansion might be stopped by an incoming edge representative $\phi(e_p^-)$ and is only resumed when the distance $D[e_p^-]$ of e_p^- is at least $D[e_i^-] + \hat{c}(\triangleleft(e_i^-, \phi(e_p^-)))$ which means that e_p^- is absorbed by e_i^-. Also, note that if an incoming edge e_i^- is extracted from $\mathcal{Q}$ and is to be expanded, all incident incoming edges that are extracted later are worse at the position of $\phi(e_i^-)$ as all incoming edges

that are extracted later have higher distance values D by the definition of $\mathcal{Q}$ and the angle cost function $\hat{c}$ inflicts only positive costs. Thus, if e_i^- has been outperformed and therefore absorbed in a particular direction, it must have happened during an earlier update. On that account, we know that there are $\mathcal{O}(k)$ stops of expansions in total and $\mathcal{O}(\ell)$ updates of all outgoing edges by the previous considerations of the initial algorithm. During each stop of an expansion and update of an outgoing edge, an update of $\mathcal{Q}$ is necessary. Since each edge can have at most two update callbacks in $\mathcal{Q}$ at the same time, the asymptotic size of $\mathcal{Q}$ remains the same. Hence, the total runtime for all relaxation steps during the execution of an Dijkstra-based algorithm amounts to $\sum_{v \in V} \mathcal{O}((\text{indeg}(v) + \text{outdeg}(v)) \log(E)) = \mathcal{O}(E \log(E))$ time.

3.4 A Lower Bound for the Complexity of the Update Problem

Both algorithms described in Sections 3.1 and 3.2 are indeed worst case optimal under the algebraic computation tree model. We can show this by reducing from the Set Equivalence Problem to the Update Problem, where the Set Equivalence Problem asks whether two given finite sets $A, B \subseteq \mathbb{R}$ are equivalent, i.e., $A = B$ which was shown to require $\Omega(n \log n)$ time by Ben-Or [BO83], where $n = |A| + |B|$. In the following we assume that $|A| = |B|$ as otherwise we can immediately decide that $A \neq B$.

Let $A, B \subseteq \mathbb{R}$ be given. We can transform this Set Equivalence Problem instance to an Update Problem instance in the following way: We start with a single vertex v that is located at $(0, 0)$. For each element $a \in A$ we construct vertices at $(a, 1)$ and for each element $b \in B$ we build vertices at $(-b, -1)$. Now, we add edges (v, a) and (b, v) for every $a \in A, b \in B$. In this way, the set A corresponds to the outgoing edges of v and B corresponds to the incoming edges of v. All edge weights and the distance $D[e_i^-]$ of all incoming edges are set to zero, and the distances $D[e_j^+]$ of all outgoing edges are set to ∞. Let the resulting graph be called $G = (V, E)$ which will be part of the Update Problem instance. We can pick an arbitrary angle cost function $\hat{c}$ whose unique global minimum is at zero and with out loss of generality we assume that the global minimum is also zero. Figure 6 shows an example of the transformation of a Set Equivalence Problem instance to a Update Problem instance.

With this transformation, we can use an algorithm that solves the Update Problem to decide if $A = B$, as $A = B \Leftrightarrow D[e_j^+] = 0$ for every outgoing edge e_j^+. To see this, consider a transformed instance as described above and let $A = B$. For an element $x_i \in A \cap B = A = B$, there is an incoming edge $e_i^- = (b_i, v)$ whose vertex b_i is at $(-x_i, -1)$ and an outgoing edge $e_i^+ = (a_i, v)$ whose vertex a_i is at $(x_i, 1)$. Thus, together they form a straight line segment and since $\hat{c}$ is an angle cost function whose global unique minimum is at zero, e_i^- is an optimal predecessor for e_i^+, as the deviating angle $\sphericalangle(e_i^-, e_i^+) = 0$. Since all edge weights are set to zero and we assume that the unique gobal minimum of $\hat{c}$ is zero, the distance $D[e_i^+]$ will therefore be zero for every outgoing edge e_i^+.

Conversely, consider a transformed instance that produced distances $D[e_j^+] = 0$ for every outgoing edge e_j^+. Since $\hat{c}$ is an angle cost function with a unique global minimum at zero with value $\hat{c}(0) = 0$, each outgoing edge has a predecessor whose edge is collinear

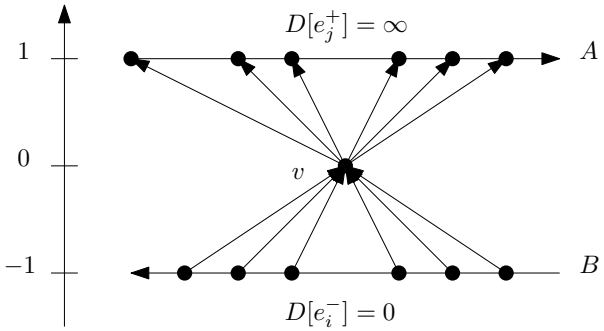


Fig. 6: Graph G that is the transformation of the Set Equivalence Problem with sets A, B to an Update Problem instance. In this example, $A \neq B$ since the optimal predecessor of the left most element of A is the right most element of B, but their corresponding edges do not form a straight line.

to the outgoing edge. Thus, for each outgoing edge with target vertex a_i, located at $(a_i, 1)$, there is some $b_j \in B$ whose vertex is at $(-b_j, -1) = (-a_i, -1)$, which implies that for each $a_i \in A$ there is a $b_j \in B$ such that $a_i = b_i$ and since $|A| = |B|$ by assumption we have that $A = B$.

4 Performance Comparison

As discussed in Section 3.3, our suggested method is capable of updating edges in both Dijkstra-based and Bellman-Ford-like algorithms. In contrast, the update procedure by Wiedemann and Adjiashvili is only applicable to Bellman-Ford-like algorithms. While their approach takes more general angle cost functions, one might want to use linear angle cost functions to make sure that the algorithm produces paths. Since both methods have similar worst-case time complexities, the question arises which of these methods can be implemented with lower constants, i.e., which of these methods can achieve a better runtime performance.

4.1 Implementation Details

We implemented both algorithms in `C++20`. For the balanced binary search trees we used the red black tree implemented in the `libstdc++` library. In order to perform a fair comparison we altered the implementation for the algorithm by Wiedemann and Adjiashvili slightly as our approach is more restricted. In particular, instead of using two balanced binary search trees to approximate clock-wise intersections of two functions f_p and f_q in $\mathcal{O}(\log \ell)$ time, we calculated the intersection of the linear functions in $\mathcal{O}(1)$ time and space in the following way:

Given a linear angle cost function $\hat{c}(\varphi) = a\varphi + b$, we distinguish between two cases depending on the relative position of the corresponding edges of f_p and f_q, when calculating the intersection angle with respect to the direction $\vec{d}$ chosen for the angles α_p and α_q. If $\alpha_p < \alpha_q$ we can formulate the functions f_p and f_q as

$$f_p(\alpha) = a(\alpha - \alpha_p) + b + D[e_p]$$
$$f_q(\alpha) = -a(\alpha - \alpha_q) + b + D[e_q]$$

in order to calculate the intersection. Equating both functions, we obtain the intersection angle α_i for

$$\alpha_i = \frac{\alpha_p + \alpha_q}{2} + \frac{D[e_q] - D[e_p]}{2a}.$$

Note that this angle has to be in the domain between α_p and α_q, that is, angle α_i only qualifies as intersection angle if

$$\min\{\alpha_p, \alpha_q - \pi\} \leq \alpha_i \leq \min\{\alpha_q, \alpha_p + \pi\}$$

holds. Similarly, if $\alpha_p > \alpha_q$ we shift f_p by 2π such that $f_p(\alpha) = a(\alpha - (\alpha_p - 2\pi)) + b + D[e_p]$, and apply similar ideas to obtain the intersection angle α_i for

$$\alpha_i = \frac{\alpha_p + \alpha_q}{2} + \frac{D[e_q] - D[e_p]}{2a} - \pi.$$

In this case, we verify the validity of α_i by

$$\min\{\alpha_p - 2\pi, \alpha_q - \pi\} \leq \alpha_i \leq \min\{\alpha_p - \pi, \alpha_q\}.$$

Due to the shift it is possible that $\alpha_i < 0$. To remedy this, we return $\alpha_i + 2\pi$ if α_i is negative.

4.2 Experiments

A major contributing factor for the runtime of both algorithms is the number of intersections of the angle cost functions, i.e., the number of non-empty intervals of optimality for different incoming edges. In particular, with a low number of intersections, the balanced binary search tree $\mathcal{T}$ representing the intervals of optimality remains small and subsequent updates on $\mathcal{T}$ are fast. In contrast, if there are only a few incoming edges with non-empty intervals of optimality, our algorithm proposed in Section 3.2 has expansions that are not stopped in the vicinity of their intersection angle which produces an overhead in terms of runtime.

Inputs. We study this behaviour closer with the following inputs. Each instance is a vertex v whose neighbourhood is built with respect to a regular k-gon with center v, where each vertex of the k-gon is part of the incoming edges into v. We further place outgoing edges randomly around the k-gon such that there are an equal amount of incoming and outgoing edges. We can influence the number of non-empty intervals of optimality by assigning incoming edges consisting of neighbouring vertices on the k-gon sufficiently large $D[e_i^-]$ values. For instance, if we want $k/2$ intersections resulting into $k/2$ non-empty intervals, every incoming edge corresponding to every second vertex of the k-gon will get a large $D[e_i^-]$ value such that this edge is not a viable predecessor for any outgoing edge. Since we also want to observe the runtime behaviour with growing input size, we need to make sure that the intersection between two incoming edges is always valid by scaling the slope a of the linear angle cost function with the input size appropriately. Note that it is not always possible to force a precise ratio of non-empty intervals and the input size, thus in the following we will always report the average ratio. The input size that we used for testing starts with $n = 10$ edges and increments in steps of ten until $n = 5000$.

Results. Figures 7 to 9 show the runtime behaviour depending on the average ratio and the input size. Figure 7 also shows the runtime of the naive $\mathcal{O}(k \cdot \ell)$ algorithm as a baseline. With our implementation of both algorithms we can confirm our prediction and observe that the runtime of the algorithm by Wiedemann and Adjiashvili increases if there are many non-zero intervals while our algorithm proposed in Section 3.2 has a longer runtime if the number of non-zero intervals decreases. Table 1 provides a more precise runtime comparison that shows that our approach is approximately 3 times faster than the algorithm by Wiedemann and Adjiashvili if almost all incoming edges have a non-zero interval of optimality, even with the runtime boost that Wiedemann and Adjiashvili's procedure got due to the modifications described in Section 4.1.

The naive implementation is clearly bested by both approaches for sufficiently large input sizes, as Figure 7 illustrates. Examining the performance closely for input sizes up to $n = 200$, as depicted in Figure 10, confirms that both methods exhibit a runtime advantage over the naive implementation even for small values of n. Additionally, the runtime difference between both approaches, along with the modifications outlined in Section 4.1, is found to be negligible.

ratio	99.73%	49.71%	9.8%
runtime factor	3.10	1.74	0.92

Tab. 1: Runtime factors between the algorithm of Wiedemann and Adjiashvili and our algorithm depending on the percentage of incoming edges with non-empty intervals. A runtime factor of x at ratio $y\%$ means that our algorithm is x times faster than the implementation of Wiedemann and Adjiashvili's algorithm, when $y\%$ of the incoming edges have a non-empty interval of optimality.

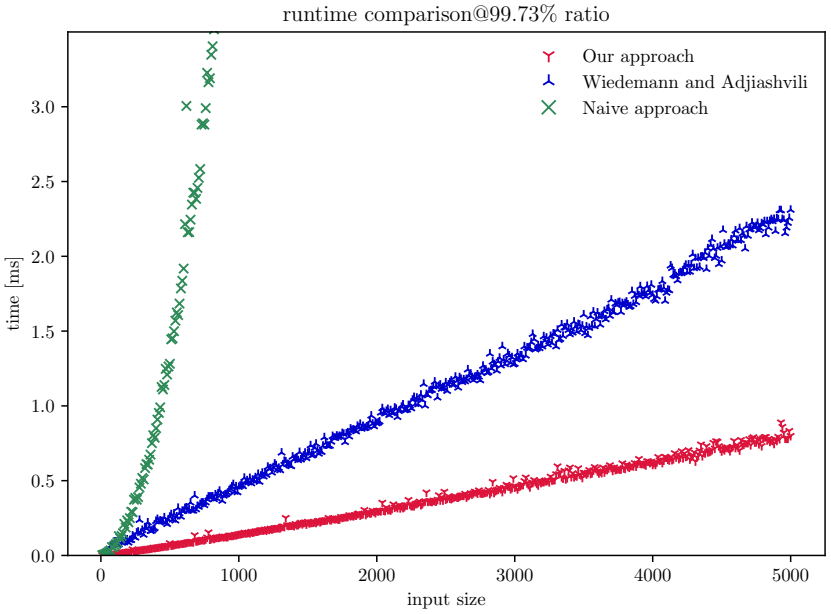


Fig. 7: Runtime comparison of the implementation of our approach and the approach by Wiedemann and Adjiashvili together with the naive implementation as a base line, where almost all incoming edges have a non-empty interval of optimality. In this setting, our approach is significantly faster than the procedure by Wiedemann and Adjiashvili. Nonetheless, both algorithms clearly outperform the base line already for smaller values of n.

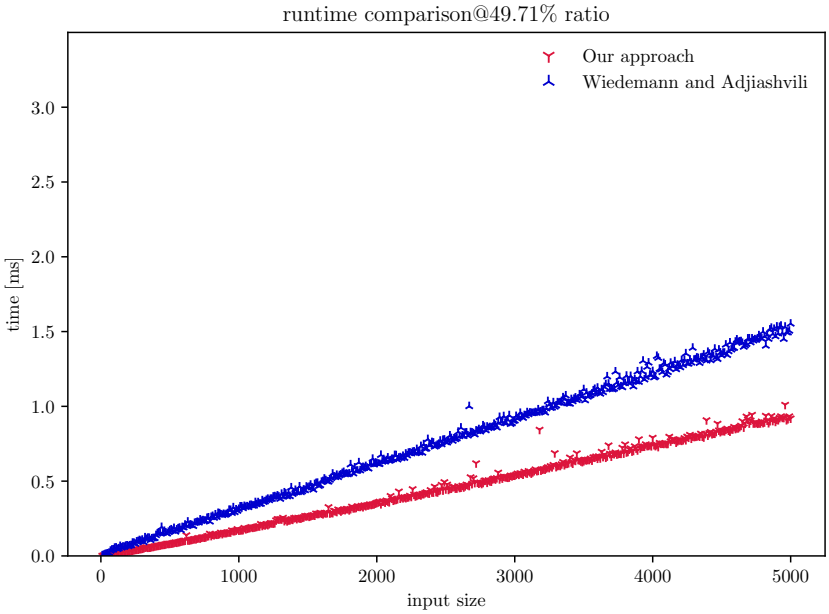


Fig. 8: When reducing the percentage of incoming edges with a non-empty interval of optimality, the performance of our approach declines while the performance of the algorithm by Wiedemann and Adjiashvili increases.

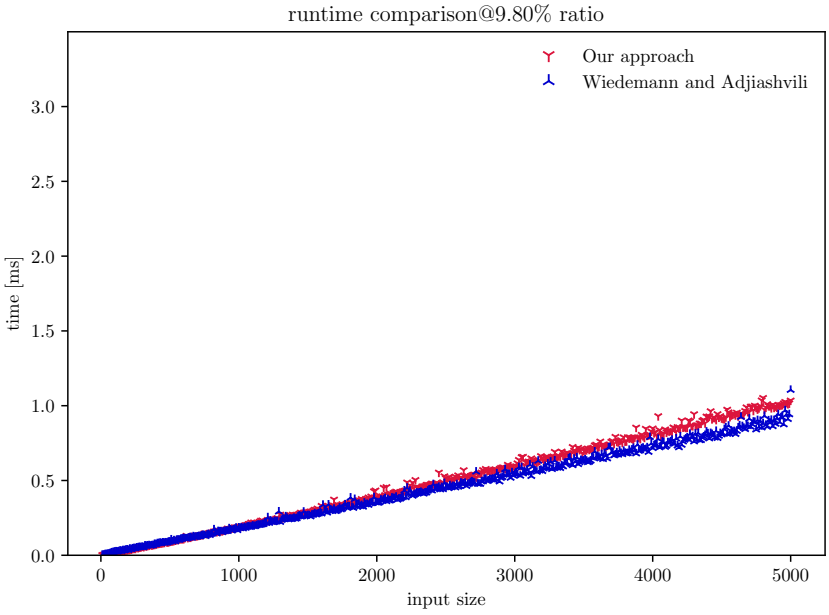


Fig. 9: If only approximately 10% of the incoming edges possess a non-empty interval of optimality, Wiedemann and Adjiashvili's algorithm outperforms our approach.

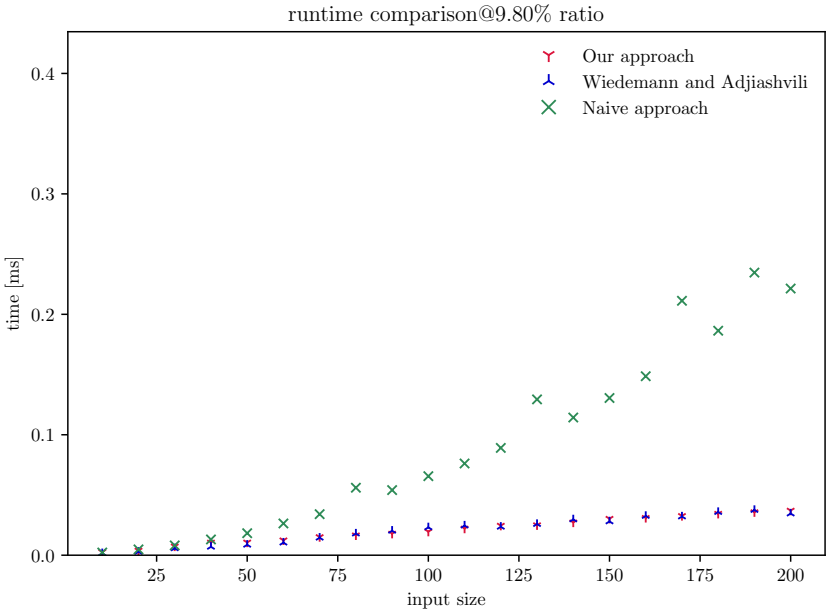


Fig. 10: Considering input sizes of $n = 10$ to $n = 200$, we observe that even for small neighbourhoods, the naive approach is slower than both approaches. We can also see that in this case – when only approximately 10% of the incoming edges have non-empty intervals – the runtime difference of both approaches is negligible.

5 Conclusion and Future Work

We proposed an algorithm for the Update Problem which naturally arises when considering angle-penalised shortest walks by employing implicit line graphs. Our procedure assumes a linear angle cost function which is natural as Wiedemann and Adjiashvili showed that linear angle cost functions guarantee a path instead of a walk. While the asymptotic worst-case runtime is similar to the update algorithm with convex angle cost functions by Wiedemann and Adjiashvili, our approach is simpler and more versatile as it is not only suitable for Bellman-Ford-like algorithms but also for Dijkstra-based algorithms which are utilised in the vector-based approach for finding transmission lines proposed by Wolf [Wol21].

We showed that both algorithms are worst-case optimal under the algebraic computation tree model, providing a lower bound of $\Omega(n \log n)$, where n is the size of the neighbourhood of the vertex v of the Update Problem. Since both algorithms have similar asymptotic complexity and are worst-case optimal, we implemented both procedures in `C++` and tested their performance. In order to have a fair comparison, we modified the more general approach by Wiedemann and Adjiashvili by reducing the computational effort for computing intersections of convex angle cost functions from $\mathcal{O}(\log \ell)$ plus overhead due to the employment of two balanced binary search trees to $\mathcal{O}(1)$ for linear cost functions. Even in this setting, our experiments showed evidence that our algorithm is up to three times faster when at least 10% of the incoming edges are potential predecessors, i.e., have a non-empty interval in which these incoming edges are the best. If there are less than 10% of such edges, we where able to provide evidence that the difference between both algorithms is negligible if the neighbourhood is below 200 vertices.

These findings prompt us to recommend our algorithm within the implicit line graph framework for finding angle-penalised shortest *paths*. Notably, our algorithm offers the added advantage of simplicity in implementation, utilising only arrays and corresponding traversals.

Nevertheless, both the theoretical and the practical side warrant deeper investigation. On the former side, a randomised algorithm that beats the lower bound proposed in Section 3.4 is interesting to pursue. As for the latter side, an extensive experimental study becomes crucial to understand the comparative performance of the update algorithms in practical scenarios. This is particularly important because our experiments were isolated cases and were not evaluated within the context of the Bellman-Ford-like algorithm. However, the intersection ratio is influenced by multiple factors that may undergo changes during the execution of the algorithm. For this purpose, a comprehensive experimental analysis is necessary, comparing both update algorithms within the Bellman-Ford-like algorithm framework outlined in Section 2. The graph instance needs to be modulated with respect to neighbourhood size and neighbour location and various linear angle cost functions must be employed, as these factors significantly impact the performance of these procedures, as discussed in Section 4.2. Special consideration must be given to realistic graph instances and angle cost penalties.

References

[Bel58] Richard Bellman: On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958, 10.1090/qam/102435.

[BO83] Michael Ben-Or: Lower bounds for algebraic computation trees. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, page 80–86. Association for Computing Machinery, 1983, 10.1145/800061.808735.

[FF56] L. R. Ford and D. R. Fulkerson: Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956, 10.4153/CJM-1956-045-5.

[LMS01] Mark Lanthier, Anil Maheshwari, and Jörg-Rüdiger Sack: Approximating shortest paths on weighted polyhedral surfaces. *Algorithmica*, 30(4):527–562, 2001, 10.1007/s00453-001-0027-5.

[RCH+18] 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, 10.1016/j.ejor.2018.01.054.

[WA22] Nina Wiedemann and David Adjiashvili: An optimization framework for power infrastructure planning. *IEEE Transactions on Power Systems*, 37(2):1207–1217, 2022, 10.1109/TPWRS.2021.3099445. `https://arxiv.org/abs/2101.03388`.

[Wol21] Samuel Wolf: Infrastructure planning via algorithms for the weighted region problem. Bachelor's thesis, Julius-Maximilians-Universität, 2021. Accessed: 2024-05-01.