Applying Skeletons to Speed Up the Arc-Flags Routing Algorithm

Ivan Khomutovskiy * Rebekka Dunker * Jessica Dierking * Julian Egbert *
Christian Helms * Finn Schöllkopf * Katrin Casel * Philipp Fischbeck *
Tobias Friedrich * Davis Isaac * Simon Krogmann * Pascal Lenzner *

Abstract

The Single-Source Shortest Path problem is classically solved by applying Dijkstra's algorithm. However, the plain version of this algorithm is far too slow for real-world applications such as routing in large road networks. To amend this, many speed-up techniques have been developed that build on the idea of computing auxiliary data in a preprocessing phase, that is used to speed up the queries. One well-known example is the Arc-Flags algorithm that is based on the idea of precomputing edge flags to make the search more goal-directed. To explain the strong practical performance of such speed-up techniques, several graph parameters have been introduced. The *skeleton dimension* is one such parameter that has already been used to derive runtime bounds for some speed-up techniques. Moreover, it was experimentally shown to be low in real-world road networks.

We introduce a method to incorporate *skeletons*, the underlying structure behind the skeleton dimension, to improve routing speed-up techniques even further. As a proof of concept, we develop new algorithms called SKARF and SKARF$^+$ that combine skeletons with Arc-Flags, and demonstrate via extensive experiments on large real-world road networks that SKARF$^+$ yields a significant reduction of the search space and the query time of about 30% to 40% over Arc-Flags. We also prove theoretical bounds on the query time of SKARF, which is the first time an Arc-Flags variant has been analyzed in terms of skeleton dimension.

1 Introduction

The problem of finding the shortest path between two nodes in a graph is generally solved via Dijkstra's algorithm [13]. However, despite its low asymptotic time complexity, this algorithm is too slow for many real-world applications on large graphs. But, it is possible to get shorter query times by storing auxiliary data that is computed once in a preprocessing phase and then use it for every routing query. Several such speed-up techniques have been developed, including Arc-Flags [20], Contraction-Hierarchies [15], Transit-Nodes [4], Hub-Labels [1] and Reach [16]. Experimental studies by Bast et al. [3] and Sommer [25] demonstrate different trade-offs between these algorithms in terms of query time, preprocessing time, and space complexity, and generally show their good performance on real-world road networks. All of the above techniques are able to achieve query times below a millisecond on continent-wide networks, being significantly faster than Dijkstra, which is in the order of seconds. However, on graphs other than road networks, the runtime of these algorithms often deteriorates to the one of Dijkstra's.

Hence, street graphs must have special properties that make these algorithms fast in practice. A popular way to explain this behavior is to bound the runtime by graph parameters that are small for real-world instances. This has been done with the help of graph parameters such as the *highway dimension*, introduced by Abraham et al. [2], and the *skeleton dimension* (κ), introduced by Kosowski and Viennot [18]. Both parameters capture the intuition that all sufficiently long shortest paths can be hit by a small set of roads.

The skeleton dimension is a parameter originally introduced to prove runtime bounds for the Hub-Labels algorithm [18]. Intuitively, the skeleton of a node u is the tree obtained by taking the union of the first p-fraction of all shortest paths starting at u. Originally, Kosowski and Viennot fixed $p = \frac{2}{3}$, while we use $p = \frac{1}{2}$. The skeleton dimension captures the maximum, taken over all nodes u and all radii $r > 0$, of the number of branches the skeleton of u has at distance r from u. A study by Blum and Storandt [10] shows that for $p = \frac{2}{3}$ the skeleton dimension of real-world road networks is small with a value of around 100 for graphs of different sizes. It also seems to depend on the densest region in the graph rather than on its size. Note that by setting $p = \frac{1}{2}$, the skeleton dimension can only decrease.

In this work, we set out to explore how to use skeletons to reduce the search space and the query time of modern routing algorithms. In particular, we focus on the popular Arc-Flags algorithm proposed first by Lauther [20, 19]. Arc-Flags makes use of the structure of shortest path trees to guide Dijkstra's algorithm to

*Hasso Plattner Institute, University of Potsdam, Germany.
This paper is based on the research results of a student group research project. The first and the second authors are the main contributors, then the other students and the advisors each are listed alphabetically.

the target without exploring many unnecessary nodes. To do so, Arc-Flags stores only few flags per edge and hence has the advantage of using significantly less space compared to Hub-Labels and Transit-Nodes. However, the Arc-Flags algorithm has worse query times than Contraction-Hierarchies, Hub-Labels, and Transit-Nodes. On a Western Europe street graph, Bast et al. [3] measure query times of around $0.56\mu s$, $2.09\mu s$ and $110\mu s$ for Hub-Labels, Transit-Nodes and Contraction-Hierarchies respectively, while the Arc-Flags queries took $408\mu s$ on average. Nevertheless, the Arc-Flags technique plays an important role in modern routing for two reasons. The first is that it is one of the few speed-up techniques that allows for unidirectional search. This advantage is of high practical relevance since most applications have dynamic graphs with changing edge costs that complicate bidirectional search. The second reason is that Arc-Flags can be combined with other techniques to achieve extremely fast query times. Four such combinations are CHASE (Contraction-Hierarchies + Arc-Flags) [6], SHARC (Shortcuts + Arc-Flags) [5], TNR+AF (Transit-Nodes + Arc-Flags) [6], and Reach-Flags (Arc-Flags + Reach) [6]. These algorithms are on the Pareto front of the space vs. query-time [25, Fig.4] and preprocessing vs. query-time [3, Fig.7] trade-offs. Also, Arc-Flags is used in practical routing, e.g., a variation of Arc-Flags with shortcuts is the routing algorithm employed by the TomTom navigation systems [24], one of the market leaders for routing services and navigation devices.

Related Work. An overview of low complexity bounds depending on the highway dimension or the skeleton dimension is given by Blum, Funke, and Storandt [9] for most of the speed-up techniques mentioned above. For graphs with bounded highway dimension h, Abraham et al. [2] prove a query time of $\mathcal{O}(h \log D)$ for Contraction-Hierarchies and Hub-Labels, and $\mathcal{O}(h^2)$ for Transit-Nodes. Here, D is the diameter of the graph. The respective space complexities are in $\mathcal{O}(nh \log D)$ and $\mathcal{O}(nh)$.

For Hub-Labels, Kosowski and Viennot [18] establish a query time of $\mathcal{O}(\kappa \log D)$ and a space complexity of $\mathcal{O}(n\kappa \log D)$ in terms of the skeleton dimension. The latter was also used by Blum, Funke, and Storandt [9] to prove a query time bound of $\mathcal{O}(\kappa^2 \log^2 n)$ for Transit-Nodes with a space complexity of $\mathcal{O}(n \log n(\kappa + \log n))$.

Blum, Funke, and Storandt [9] introduced another concept called *bounded growth* to capture properties of road networks. A network is said to have bounded growth if the number of nodes at a distance at most r from a vertex is at most quadratic in r. For bounded-growth networks, they show query times of $\mathcal{O}(\sqrt{n} \log n)$, $\mathcal{O}(n^{2/3} \log^{8/3} n)$, and $\mathcal{O}(\sqrt{n})$ for Contraction-Hierarchies, Transit-Nodes, and Hub-Labels respectively. The space complexities were shown to be $\mathcal{O}(n \log D)$, $\mathcal{O}(n^{4/3} \log^{4/3} n)$, and $\mathcal{O}(\sqrt{n})$ respectively. Bauer et al. [8] analyze Contraction-Hierarchies in terms of treewidth t, and prove a query time of $\mathcal{O}(t \log n)$ and space complexity of $\mathcal{O}(nt \log n)$.

To the best of our knowledge, no theoretical bounds in the skeleton dimension were proved for Arc-Flags or the algorithms building upon it. However, Arc-Flags has been extensively evaluated experimentally [3], where it performs best among the purely goal-oriented algorithms. The performance of Arc-Flags heavily depends on the flags and thus on the graph partition on which the flags are based. Bauer et al. [7] show that optimal partitioning for Arc-Flags is NP-hard on binary trees. However, many partitioning algorithms have been developed that produce balanced partitions with few boundary nodes. Möhring et al. [21] give a comparison of different partitioning algorithms and conclude that the METIS partition algorithm presented by Karypis and Kumar [17] is a suitable choice as a partition for Arc-Flags. It does not require any geometric information about the network and the low number of boundary nodes per cell speeds up the remaining preprocessing steps. These steps of Arc-Flags are classically done with Dijkstra searches starting from all boundary nodes and setting corresponding flags along the way. This takes multiple hours for larger graphs, but can be replaced with the PHAST algorithm by Delling et al. [12], reducing the preprocessing after the partition to minutes.

Our Contribution. We incorporate the concept of skeletons into the Arc-Flags algorithm to achieve both provable theoretical runtime guarantees and significant speed-ups in practice. To this end, we introduce a new parameter called *cell skeleton dimension* ($\tilde{\kappa}$) for a graph with a given partition. Our experiments reveal that, although being large for partitions with large cells, $\tilde{\kappa}$ gets closer to the classical skeleton dimension κ, the smaller the cells are chosen. It turns out to be a useful parameter to analyze algorithms that use graph partitioning.

As a proof of concept for the versatility of using skeletons, we present the SKARF algorithm (SKeleton ARc-Flags), which is based on the idea of searching only along skeletons using the Arc-Flags framework.

On the theoretical side, we prove that a SKARF query has a search space of $\mathcal{O}(\tilde{\kappa}D)$ and runs in $\mathcal{O}(\Delta\tilde{\kappa}D + \tilde{\kappa}D \log(\tilde{\kappa}D))$ time, where Δ is the maximum node degree, which is typically very small for street-graphs, and D is the diameter of the underlying graph. To

the best of our knowledge, this is the first theoretical analysis of an Arc-Flags variant using a skeleton-based graph parameter.

On the practical side, we show via extensive experiments that the idea of incorporating skeletons into the Arc-Flags algorithm gives significantly smaller search spaces. More precisely, we show that SKARF$^+$, a combination of SKARF and Arc-Flags, gives a 30 % to 40 % reduction of the search space and query times over Arc-Flags in most cases. For small partition sizes and bidirectional algorithm versions we even achieve improvements between 40 % to 50 %. We emphasize that this speed-up is achieved while maintaining a similar preprocessing time and using about three times the storage space of Arc-Flags in the unidirectional case and two times the storage space of bidirectional Arc-Flags in the bidirectional case.

Organization. In Section 2 we give the basic definitions and notations regarding road networks, Arc-Flags, bidirectional search and skeleton dimension. In Section 3, we introduce the cell skeleton dimension and conduct an experimental study to estimate its size on real-world road networks. Then, in Section 4 we introduce our new algorithm SKARF and perform a theoretical analysis of its runtime, using the cell skeleton dimension parameter. In Section 5 we introduce SKARF$^+$ as a combination of SKARF and Arc-Flags that is more efficient in practice and compare it experimentally with classical Arc-Flags in Section 6. We conclude in Section 7 and give an outlook for future research. The source code [1] and data for all our experiments is publicly available.

2 Preliminaries

We start with basic notions regarding road networks.

2.1 Road Network Modeling and Notation We model a *road network* as a weighted, strongly connected directed graph $G = (V, E, l)$ with $l : E \to \mathbb{R}^+$. For two nodes $u, v \in V$ we refer to the shortest path from u to v in G as $P_G(u, v)$ or simply $P(u, v)$ if the graph is clear from context. We refer to the *distance* from u to v, which is given by the sum of all edge weights in $P(u, v)$, as $d_G(u, v)$ or simply $d(u, v)$ if the graph is clear from context. We denote the *transposed graph of* G by $G' = (V, E', l')$, where $E' = \{(v, u) \mid (u, v) \in E\}$ and $\forall (v, u) \in E' : l'((v, u)) = l((u, v))$.

We assume that G has minimum edge weight 1 and a bounded maximum node degree of Δ. Since real-world networks can be modeled as finite graphs, they have minimum edge weights that can be scaled linearly so that the minimum weight is 1. Furthermore, we assume every edge to be a shortest path in G and all shortest paths in G to be unique. If this is not the case, this property can be achieved by slightly perturbing the edge weights and removing edges that are no shortest paths. Note that under this assumption, there exists exactly one shortest path tree originating from a given source. We refer to the shortest path tree of a node u as T_u.

On road networks, for two distinct nodes $u, v \in V$, the distance from u to v is usually very similar to the distance from v to u. To capture this useful property in our model, we assume that there is a small constant $\delta \geq 1$ such that $d(u, v)/d(v, u) \leq \delta$ holds. The experimental study by Mori and Samaranayake [22] gives justification for assuming that δ is small. There it is shown that in a collection of large cities for paths that are at least 4 500 meters long, δ does not exceed a value of 4.

We denote a partition of V by $\mathcal{P}$ and refer to elements $C \in \mathcal{P}$ as *cells* of the partition $\mathcal{P}$. The *diameter of a cell* C is denoted by $D_C := \max_{u,v \in C} d(u, v)$, the *maximum cell diameter* by $D_{\max} := \max_{C \in \mathcal{P}} D_C$ and the *maximum cell size* by $\mu := \max_{C \in \mathcal{P}} |C|$. For easier notation we refer to the cell of some node $v \in V$ as C_v. We also define the *boundary node set of a cell* as $\mathfrak{B}_C \subseteq C$, $\mathfrak{B}_C := \{u \in C \mid \exists v \notin C : (u, v) \in E \vee (v, u) \in E\}$. These are the nodes that connect the different cells.

2.2 Arc-Flags The Arc-Flags algorithm [20, 19] is a goal-directed routing speed-up technique that relies on precomputed data to direct the search to the target. In the preprocessing phase, the graph is partitioned and labels are attached to each edge e. A label contains flags for every cell of the partition and a flag is set to *true* if and only if there is a shortest path to a node in the respective cell that starts with edge e. These labels are classically precomputed by building shortest path trees from all boundary nodes of a cell in the transposed graph and setting the flag of the given cell for each edge on the shortest path tree to *true*. This data is then used in the shortest path query that runs a modified Dijkstra query relaxing only those edges for which the target cell flag is set. We consider the Dijkstra implementation by Fredman and Tarjan [14], where Fibonacci heaps are used, yielding a runtime in $\mathcal{O}(|E| + |V| \log |V|)$. In practice, Arc-Flags has extremely short query times on real-world road networks when compared to Dijkstra, which is not yet theoretically explained.

The performance of Arc-Flags depends heavily on the graph partition. A good partition should have few boundary nodes and balanced cell sizes. Möhring et

[1] https://github.com/SKARF-Routing-Algorithm/SKARF-Routing-Algorithm.git

al. [21] compare different partitioning algorithms and conclude that METIS, presented by Karypis and Kumar [17], meets these criteria and is thus a suitable choice as a partitioner for Arc-Flags. It also has the advantage that it does not need any geometric information about the network. These advantages apply unaltered to SKARF, which is why we use this partitioning algorithm in our experiments in Section 3 and Section 6.

2.3 Bidirectional Search Plain Dijkstra can already be accelerated by searching from source and target node simultaneously. A correctness proof for different stopping criteria has been provided by Pohl [23]. Given that Arc-Flags runs a Dijkstra search ignoring unnecessary edges, a bidirectional version of Arc-Flags is correct as well. In this work, we discuss bidirectional algorithm versions that switch between forward and backward search based on the smallest tentative distance in the two Dijkstra queues.

2.4 Skeleton Dimension The notion of *skeletons* and *skeleton dimension* was first introduced by Kosowski and Viennot [18]. It captures the intuition that for every source node, the shortest paths that reach far away from the source all lead over the same small set of subpaths for most of the way. In order to define skeletons, we need to introduce the *reach* first.

DEFINITION 2.1. (REACH) *Let $s \in V$ and $v \in T_s$ with $T_s(v) \subseteq T_s$ being the set of descendants of v in T_s. We define the* reach *of v in T_s as* $\mathrm{Reach}_s(v) := 0$ *if $T_s(v) = \emptyset$ and otherwise as*

$$\mathrm{Reach}_s(v) := \max_{x \in T_s(v)} d_{T_s}(v, x).$$

With this, we are ready to define the skeleton of a node.

DEFINITION 2.2. (SKELETON) *Let $s \in V$. The* skeleton T_s^* *of s is the subgraph of T_s induced by the edges $e = (u, v) \in T_s$ satisfying*

$$\mathrm{Reach}_s(v) + l(e) > d(s, u).$$

The parameter *skeleton dimension* is the maximum width of all skeletons in a graph. Formally it is defined over the *cut* as follows.

DEFINITION 2.3. (CUT) *For a node $s \in V$ and a radius $r \in \mathbb{R}^+$ we define the* cut *on the graph G around the node s at distance r as*

$$\mathrm{Cut}_s^r(G) := \{e = (u, v) \in E \mid d(s, u) < r \leq d(s, u) + l(e)\}.$$

DEFINITION 2.4. (SKELETON DIMENSION) *The* skeleton dimension $\kappa(G)$ *of a graph $G = (V, E, l)$ is*

$$\kappa(G) := \max_{s \in V, r \in \mathbb{R}^+} |\mathrm{Cut}_s^r(T_s^*)|.$$

We omit the reference to the graph G if it is clear from the context. Also, in this case we use κ' for $\kappa(G')$.

Note that our skeleton definition is slightly different from the original one by Kosowski and Viennot [18]. There, the skeleton is defined on the geometric realization of a graph while we introduce a discrete version for better algorithmic applicability. Though in our discrete version the branches of the skeleton become slightly longer, our experiments show that in practice this results in no significant increase of the skeleton dimension. Another difference is that for a node $s \in V$ Kosowski and Viennot [18] include the first two thirds of every shortest path starting at s in its skeleton, while we restrict T_s^* to the first half of every shortest path starting at s. We use this more restrictive skeleton definition because it is sufficient for our proofs and yields smaller skeletons and thus enables smaller search spaces.

3 Cell Skeletons

Our main tool to theoretically study the effect of low skeleton dimension on our modification of Arc-Flags are *cell skeletons* which we define here.

3.1 Cell Skeleton Dimension

DEFINITION 3.1. (CELL SKELETON) *Given a cell $C \in \mathcal{P}$ the* cell skeleton T_C^* *is the union of all skeletons originating in cell C, i.e.,*

$$T_C^* := \bigcup_{v \in C} T_v^*.$$

Analogously to the skeleton dimension, *cell skeleton dimension* describes over all cells and all distances $r \in \mathbb{R}^+$ the maximum cut size at a distance r from the cell. To measure the distance from the cell, we take the distance from its *center node*.

DEFINITION 3.2. (CENTER NODE) *Given a cell $C \in \mathcal{P}$, the* center node $z(C)$ *is defined as the node v of smallest index that minimizes* $\max_{b \in \mathfrak{B}_C} d(v, b)$

DEFINITION 3.3. (CELL SKELETON DIMENSION) *Given a graph G and a node partition $\mathcal{P}$, the* cell skeleton dimension $\tilde{\kappa}(G, \mathcal{P})$ *of graph G with partition $\mathcal{P}$ is the maximum cut size over all cell skeletons. Formally,*

$$\tilde{\kappa}(G, \mathcal{P}) := \max_{C \in \mathcal{P}, r \in \mathbb{R}^+} |Cut_{z(C)}^r(T_C^*)|.$$

We omit the reference to G and $\mathcal{P}$ if they are clear from the context and we refer to the cell skeleton dimension of the transposed graph G' with partition $\mathcal{P}$ as $\tilde{\kappa}'$.

Table 1: The cell skeleton dimension $\tilde{\kappa}$ of the Germany and France road networks and the corresponding cell skeleton dimension of their transposed graphs, $\tilde{\kappa}'$.

	Germany		France	
Partition	$\tilde{\kappa}$	$\tilde{\kappa}'$	$\tilde{\kappa}$	$\tilde{\kappa}'$
50 cells	2230	2178	2278	2307
100 cells	1676	1705	1623	1608
200 cells	1200	1179	1057	1061
500 cells	718	757	751	750
1000 cells	517	503	578	590

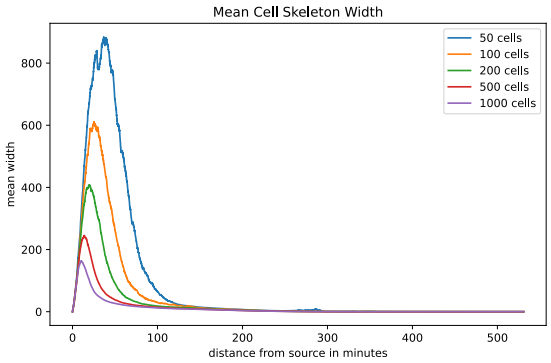

Figure 1: The mean width $\text{mean}_{C\in\mathcal{P}}\text{Cut}^r_{z(C)}(T^*_C)$ on cell skeletons at different radii r with different colors for varying partition sizes.

3.2 Cell skeleton experiments To analyze algorithms using the cell skeleton dimension, we measure its value in real-world road networks for different partition sizes. For this we conduct an experimental study on the Germany and France road networks.

These networks are taken from OpenStreetMap [11]. They both have around 4,000,000 nodes and 10,000,000 edges where the edge weights represent the corresponding travel times. To partition these networks, we use the METIS algorithm [17] and consider five different partitions of sizes 50, 100, 200, 500 and 1000. For each of those we calculate the cell skeleton dimension $\tilde{\kappa}$ and the cell skeleton dimension on the transposed graph $\tilde{\kappa}'$. Table 1 presents our results.

We observe that $\tilde{\kappa}$, with a value of more than 2000 for 50 cells, is quite large when the cells are large. However, it decreases with decreasing cell size, with a value of only 517 when the Germany graph is partitioned into 1000 cells. The corresponding values on the France graph are very similar.

To set this in relation to the well-known skeleton dimension κ, we refer to the study by Blum and Storandt [10]. They compute κ for graphs of different sizes and present results with a skeleton dimension of around 100 with a value of 114 for Germany. This value is smaller than our $\tilde{\kappa}$, but the gap decreases for decreasing cell sizes, i.e., larger partitions. For a 1000-cells partition in Germany $\tilde{\kappa}$ is about 5 times larger than the skeleton dimension by the definition that Blum and Storandt [10] used and that they computed for their Germany street graph. The cell skeleton dimension on the transposed graph $\tilde{\kappa}'$ behaves similarly.

Furthermore, Figure 1 shows the mean width of cell skeletons $\text{mean}_{C\in\mathcal{P}}\text{Cut}^r_{z(C)}(T^*_C)$ at different radii r from the cell's center node. It indicates that cell skeletons have a comparably large width at the start that decreases rapidly at larger distances approaching the width of a single skeleton. This is expected behaviour, since within the cell, every edge belongs to the cell skeleton, but outside of the cell the individual skeletons of the cell nodes seem to overlap strongly. Thus, it is worth to keep in mind that the values for the maximum width of cell skeletons from Table 1 are mostly outliers and for the majority of possible radii, the width of the cell skeletons is considerably lower. This also explains the good performance of our algorithm in practice, which we analyze in Section 6.

To conclude, the cell skeleton dimension $\tilde{\kappa}$ is a graph parameter that can be used for analyzing algorithms that are performed on partitioned graphs. Its size can be controlled by the chosen partition and gets closer to the skeleton dimension for smaller cell sizes, eventually becoming equal to the skeleton dimension when cells are single nodes. This parameter will prove to be especially useful for our theoretical analysis in the following section.

4 SKARF

SKARF (SKeleton ARc-Flags) is based on the observation that each shortest path is covered by the union of the source's skeleton and the target's skeleton (Lemma 4.3). Hence, it is sufficient to route on the union of the two skeletons. In Figure 2, two cells in a Germany street graph that is partitioned into 1000 cells are visualized with their corresponding cell skeletons. Remarkably, the marked edges completely cover all shortest paths between these two cells.

The idea of SKARF is inspired by the Arc-Flags algorithm. Instead of saving shortest path trees as in Arc-Flags, SKARF precomputes cell skeletons for all cells in the graph and in its transpose. The modified Dijkstra query is then restricted to the cell skeletons of the source and target cell.

In this section, we provide a description and theoretical analysis of the SKARF algorithm, including its

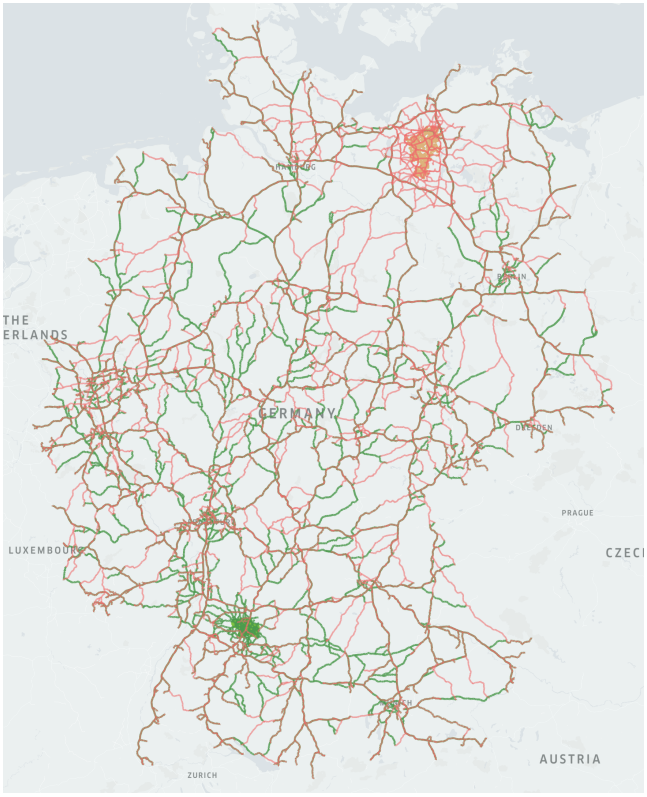

Figure 2: The street graph of Germany is partitioned into 1000 cells. The cell skeletons of the red and green cell are highlighted in the corresponding color.

preprocessing, query handling and search space size.

4.1 Preprocessing Similarly to Arc-Flags, the graph is partitioned into $\gamma \in \mathbb{N}$ cells.

Algorithm. For every edge $e \in E$, two arrays of flags X_e and Y_e are created, each containing one flag for every cell $C \in \mathcal{P}$ which we call $X_e(C)$ and $Y_e(C)$, respectively. All flags are initialized with $false$. We then compute the cell skeleton T_C^* for every cell $C \in \mathcal{P}$, and for every edge $e \in T_C^*$, we set the flag $X_e(C)$ to $true$. This is repeated on the transposed graph such that for every edge $e \in T_C'^*$, the flag $Y_e(C)$ is set to $true$.

Lemma 4.1 states that the cell skeleton of a cell C is equal to the union of the skeletons of its boundary nodes and the inside of the cell.

LEMMA 4.1. *Let $C \in \mathcal{P}$ and $G[C]$ be the subgraph induced by the nodes in C. It holds that*

$$T_C^* = G[C] \cup \bigcup_{b \in \mathfrak{B}_C} T_b^*.$$

Proof. We first show $T_C^* \subseteq G[C] \cup \bigcup_{b \in \mathfrak{B}_C} T_b^*$.
For this let $e = (u, v) \in T_C^*$ be an edge. There must exist a node $s \in C$ for which e lies on its skeleton. According to the skeleton definition, this means that $\text{Reach}_s(v) + l(e) > d(s, u)$. For this to be true there must exist a descendent $x \in T_s(v)$ with $d_{T_s}(v, x) + l(e) > d(s, u)$. If $e \in G[C]$, we are done. Otherwise there must exist a boundary node $b \in \mathfrak{B}_C$ on the shortest path $P(s, v)$, which leads over e. Then

$$\begin{aligned} \text{Reach}_b(v) + l(e) &\geq d_{T_b}(v, x) + l(e) \\ &= d_{T_s}(v, x) + l(e) \\ &> d(s, u) \\ &\geq d(b, u). \end{aligned}$$

Thus e lies on the skeleton of b and therefore also on $\bigcup_{b \in \mathfrak{B}_C} T_b^*$. This proves that every edge in T_C^* is also in $G[C] \cup \bigcup_{b \in \mathfrak{B}_C} T_b^*$.
Let now $v \in T_C^*$ be a node. Then there exists an $s \in C$ with $v \in T_s^*$. If $v \in C$, we are done. Otherwise $v \neq s$ holds, which is why there exists an incident edge $e = (u, v) \in T_s^*$ and thus $e \in T_C^*$. It was shown above that then $e \in G[C] \cup \bigcup_{b \in \mathfrak{B}_C} T_b^*$ follows, which is why v must also be in $G[C] \cup \bigcup_{b \in \mathfrak{B}_C} T_b^*$. This proves that every node in T_C^* is also in $G[C] \cup \bigcup_{b \in \mathfrak{B}_C} T_b^*$.

Now we show $G[C] \cup \bigcup_{b \in \mathfrak{B}_C} T_b^* \subseteq T_C^*$.
For this let $e = (u, v) \in G[C] \cup \bigcup_{b \in \mathfrak{B}_C} T_b^*$ be an edge. If $e \in \bigcup_{b \in \mathfrak{B}_C} T_b^*$, we are done because $\mathfrak{B}_C \subseteq C$ and therefore $\bigcup_{b \in \mathfrak{B}_C} T_b^* \subseteq T_C^*$. Let otherwise $e \in G[C]$, then $u \in C$ holds. Then

$$\text{Reach}_u(v) + l(e) > 0 = d(u, u),$$

which is why e lies on the skeleton of u. Because $u \in C$, it follows that e lies also on the cell skeleton T_C^*. This proves that every edge in $G[C] \cup \bigcup_{b \in \mathfrak{B}_C} T_b^*$ is also in T_C^*.
Let now $v \in G[C] \cup \bigcup_{b \in \mathfrak{B}_C} T_b^*$ be a node. If $v \in \bigcup_{b \in \mathfrak{B}_C} T_b^*$, we are done because $\bigcup_{b \in \mathfrak{B}_C} T_b^* \subseteq T_C^*$. If otherwise $v \in C$, then $v \in T_v^* \subseteq T_C^*$. This proves that every node in $G[C] \cup \bigcup_{b \in \mathfrak{B}_C} T_b^*$ is also in T_C^*. ∎

Thus, in order to compute the cell skeletons, it is sufficient to calculate the skeletons of the boundary nodes and then perform a final iteration over all edges $e \in E$ to set the corresponding own-cell flag if both incident nodes are in the same cell $C \in \mathcal{P}$, i.e., $X_e(C) := true$ and $Y_e(C) := true$.

Runtime. The preprocessing times of both Arc-Flags and SKARF are dominated by computing shortest path trees from all boundary nodes. SKARF preprocessing does this twice, on G and G', but its runtime remains asymptotically equal to Arc-Flags preprocessing.

Lemma 4.2 states the preprocessing runtime of SKARF after the graph is partitioned.

LEMMA 4.2. *Given a graph that is partitioned into γ cells with at most β boundary nodes each, the preprocessing has a runtime in $\mathcal{O}(\gamma\beta(|V|\log|V| + |E|))$.*

Proof. The runtime for computing the skeleton of a node is dominated by computing its shortest path tree, which can be done in $\mathcal{O}(|V|\log|V| + |E|)$ [14]. Thus computing the skeletons of all boundary nodes can be done in $\mathcal{O}(\gamma\beta(|V|\log|V| + |E|))$. The final setting of the own-cell flags does not alter this runtime. ∎

Space consumption. For every edge $e \in E$, the flag arrays X_e and Y_e, each with the size of γ, have to be saved, resulting in a space consumption of $2\gamma|E|$ bits. This is twice as much as the space consumption of Arc-Flags, where only flags for the transposed graph are needed and therefore, for every edge, only one array of size γ has to be saved.

Möhring et al. [21] show that the space consumption of Arc-Flags can be reduced by storing all existing flag combinations per edge in a matrix and assigning pointers to the corresponding combination to the edges. Since not all flag combinations occur in the matrix, the pointers require less space than the flag combinations themselves. This method can also be applied to SKARF.

4.2 Query and Correctness We describe the way SKARF answers queries and prove the correctness.

Algorithm. Given a source $s \in V$ and a target $t \in V$, SKARF runs a modified Dijkstra query where an edge $e \in E$ is only considered if $X_e(C_s) = true$ or $Y_e(C_t) = true$.

Correctness. The skeleton of a node v contains the first half of every shortest path starting at v. This implies that every edge on the first half of $P(s,t)$ is on the skeleton of s and every edge on the second half of $P(s,t)$ is on the skeleton of t on the transposed graph. Lemma 4.3 captures this intuition.

LEMMA 4.3. *For every edge $e = (u,v) \in P(s,t)$ it holds that $(u,v) \in T_s^*$ or $(v,u) \in T_t'^*$.*

Proof. We consider two cases. For the first case let $d(s,u) < d(u,t)$. Then it follows that

$$\text{Reach}_s(v) + l(e) \geq d(v,t) + l(e) = d(u,t) > d(s,u)$$

and therefore by the definition of a skeleton $(u,v) \in T_s^*$.

For the second case let $d(s,u) \geq d(u,t)$ and let $\text{Reach}'_t(u)$ denote the reach of u in T'_t. Then $d_{G'}(u,s) \geq d_{G'}(t,u)$ and it follows that

$$\begin{aligned}\text{Reach}'_t(u) + l'((v,u)) &\geq d_{G'}(u,s) + l(e)\\ &\geq d_{G'}(t,u) + l(e)\\ &> d_{G'}(t,v).\end{aligned}$$

Therefore, by the definition of a skeleton we have $(v,u) \in T_t'^*$. ∎

It follows immediately that every edge in $P(s,t)$ is also either in the cell skeleton of C_s or in the cell skeleton of C_t in the transposed graph. Thus, by the algorithm preprocessing specification every edge on $P(s,t)$ must have the flags set that are specified in Lemma 4.4, such that it is not ignored in the query.

LEMMA 4.4. *For any distinct $s,t \in V$ and any edge $e \in P(s,t)$, it holds that $X_e(C_s) = true$ or $Y_e(C_t) = true$.*

Proof. Let $e \in P(s,t)$ be an edge. Lemma 4.3 implies that $e \in T_s^*$ or $e \in T_t'^*$. It follows immediately, that $e \in T^*_{C_s}$ or $e \in T'^*_{C_t}$ holds. By the preprocessing algorithm specification, it follows that $X_e(C_s) = true$ or $Y_e(C_t) = true$. ∎

Thus, the Dijkstra query operates on a graph that contains all edges in $P(s,t)$ and finds it as the shortest path from s to t.

COROLLARY 4.1. *For a source $s \in V$ and a target $t \in V$ the SKARF query outputs the optimal shortest path from s to t.*

Proof. Lemma 4.4 implies that all the edges $e \in P(s,t)$ satisfy $X_e(C_s) = true$ or $Y_e(C_t) = true$. Thus, the Dijkstra query operates on a graph that contains all edges in $P(s,t)$ and finds it as the shortest path from s to t. ∎

4.3 Search Space In order to quantify the search space in terms of the distance from source to target, we use the notion of a *ball*.

DEFINITION 4.1. (BALL) *The* ball *of a node $s \in V$ with radius $r \in \mathbb{R}^+$ on the graph G is defined as*

$$\text{Ball}_s^r(G) := \{v \in V \mid d(s,v) \leq r\}.$$

Recall that we assume a minimum edge length of 1, which is why there are at most $\tilde{\kappa}$ nodes with a distance from the center node in an interval of length 1. Thus the following lemma holds.

LEMMA 4.5. *For a cell $C \in \mathcal{P}$ and all $r \in \mathbb{N}^+$ it holds that*

$$|\text{Ball}^r_{z(C)}(T^*_C) \setminus \text{Ball}^{r-1}_{z(C)}(T^*_C)| \leq \tilde{\kappa}.$$

Proof. Let $U = \text{Ball}^r_{z(C)}(T^*_C) \setminus \text{Ball}^{r-1}_{z(C)}(T^*_C)$ and $r' = \frac{1}{2}\left(r - 1 + \min_{u \in U} d_{T^*_C}(z(C), u)\right)$.
We define the function

$$f : U \to \text{Cut}^{r'}_{z(C)}(T^*_C), \text{ with}$$
$$f : u \mapsto e \iff e \in P_{T^*_C}(z(C), u) \cap \text{Cut}^{r'}_{z(C)}(T^*_C)$$

and show that f is injective by proving the following two claims.

1. For every $u \in U$ there exists an edge $e \in E$ with $e = f(u)$.
   Let $u \in U$ and let $e = (x, u) \in P_{T^*_C}(z(C), u)$ be the last edge on the shortest path from $z(C)$ to u. Recall that we assume minimum edge lengths of 1 and that $d_{T^*_C}(z(C), u) \leq r$. Thus $d_{T^*_C}(z(C), x) \leq r - 1 < r'$ must hold. Because also $r' < d_{T^*_C}(z(C), u)$ it follows that $e \in \text{Cut}^{r'}_{z(C)}(T^*_C)$ and thus $e = f(u)$.

2. For every $e = (x, y) \in \text{Cut}^{r'}_{z(C)}(T^*_C)$ there exists at most one $u \in U$ with $u \in f^{-1}(e)$.
   Let $e = (x, y) \in \text{Cut}^{r'}_{z(C)}(T^*_C)$ and some $u \in U$ with $u \in f^{-1}(e)$. Then $e \in P_{T^*_C}(z(C), u)$ holds. Note that

   $$\begin{aligned} d_{T^*_C}(y, u) &= d_{T^*_C}(z(C), u) - d_{T^*_C}(z(C), y) \\ &\leq r - d_{T^*_C}(z(C), y) \\ &\leq r - r' \\ &< 1. \end{aligned}$$

   Because we assume minimum edge lengths of 1, $y = u$ must hold. Thus $u = y$ is the only node in U for which $u \in f^{-1}(e)$ can hold.

Because f is injective, it holds that $|U| \leq |\text{Cut}^{r'}_{z(C)}(T^*_C)|$ and therefore

$$|\text{Ball}^r_{z(C)}(T^*_C) \setminus \text{Ball}^{r-1}_{z(C)}(T^*_C)| = |U| \leq |\text{Cut}^{r'}_{z(C)}(T^*_C)| \leq \tilde{\kappa}.$$

∎

Then, the number of nodes in a cell skeleton that are inside a certain ball can be bounded as follows.

LEMMA 4.6. *For a cell $C \in \mathcal{P}$ and all $r \in \mathbb{R}^+$ it holds that*

$$|\text{Ball}^r_{z(C)}(T^*_C)| \leq \tilde{\kappa}\lceil r \rceil.$$

Proof. Because for $j, j' \in \mathbb{N}, j \neq j'$ the sets $\text{Ball}^j_{z(C)}(T^*_C) \setminus \text{Ball}^{j-1}_{z(C)}(T^*_C)$ and $\text{Ball}^{j'}_{z(C)}(T^*_C) \setminus \text{Ball}^{j'-1}_{z(C)}(T^*_C)$ are disjoint and $\text{Ball}^r_{z(C)}(T^*_C) \subseteq \text{Ball}^{\lceil r \rceil}_{z(C)}(T^*_C)$, it follows from Lemma 4.5 that

$$\begin{aligned} |\text{Ball}^r_{z(C)}(T^*_C)| &\leq |\text{Ball}^{\lceil r \rceil}_{z(C)}(T^*_C)| \\ &= \sum_{i=1}^{\lceil r \rceil} |\text{Ball}^i_{z(C)}(T^*_C) \setminus \text{Ball}^{i-1}_{z(C)}(T^*_C)| \\ &\leq \sum_{i=1}^{\lceil r \rceil} \tilde{\kappa} = \lceil r \rceil \tilde{\kappa}. \end{aligned}$$

This proves the claim. ∎

The following Lemmas 4.7 and 4.8 bound the search space in the cell skeleton of the source cell and the target cell, respectively.

LEMMA 4.7. *Given a source $s \in V$ and a target $t \in V$, the number of nodes settled by the SKARF query in $T^*_{C_s}$ is upper bounded by $\tilde{\kappa}\lceil d(s, t) + D_{\max} \rceil$.*

Proof. Let $v \in T^*_{C_s}$ be a node that is settled by the SKARF query. Since v got settled before t it follows that $d(s, v) \leq d(s, t)$ and thus

$$d(z(C_s), v) \leq d(z(C_s), s) + d(s, v) \leq D_{\max} + d(s, t).$$

Then, every settled node in $T^*_{C_s}$ is inside $\text{Ball}^{d(s,t)+D_{\max}}_{z(C_s)}(T^*_{C_s})$, and by Lemma 4.6 we obtain $|\text{Ball}^{d(s,t)+D_{\max}}_{z(C_s)}(T^*_{C_s})| \leq \tilde{\kappa}\lceil d(s, t) + D_{\max} \rceil$. ∎

LEMMA 4.8. *Given a source $s \in V$ and a target $t \in V$ and let $T'^*_{C_t}$ denote the transposed cell skeleton of C_t. Then the number of nodes that are settled by the SKARF query in $T'^*_{C_t}$ is upper bounded by*

$$\tilde{\kappa}'\lceil (\delta + 1)d(s, t) + D_{\max} \rceil.$$

Proof. Let $v \in T'^*_{C_t}$ be a node that is settled by the SKARF query. Since v got settled before t it follows that $d(s, v) \leq d(s, t)$ and thus

$$\begin{aligned} d_{G'}(t, v) &\leq d_{G'}(t, s) + d_{G'}(s, v) \\ &= d(s, t) + d(v, s) \\ &\leq d(s, t) + \delta d(s, v) \\ &\leq d(s, t) + \delta d(s, t) \\ &= (\delta + 1)d(s, t). \end{aligned}$$

It follows that

$$\begin{aligned} d_{G'}(z(C_t), v) &\leq d_{G'}(z(C_t), t) + d_{G'}(t, v) \\ &\leq D_{\max} + (\delta + 1)d(s, t). \end{aligned}$$

Then, every settled node in $T'^*_{C_t}$ is inside $\text{Ball}^{(\delta+1)d(s,t)+D_{\max}}_{z(C_t)}(T'^*_{C_t})$ and by Lemma 4.6 we obtain $|\text{Ball}^{(\delta+1)d(s,t)+D_{\max}}_{z(C_t)}(T'^*_{C_t})| \leq \tilde{\kappa}\lceil(\delta+1)d(s,t)+D_{\max}\rceil$. ∎

Now we can make a statement about the maximum number of nodes that can be settled by the SKARF query. For this, note that for a source $s \in V$ and a target $t \in V$, the SKARF query scans only nodes in $T^*_{C_s} \cup T'^*_{C_t}$.

THEOREM 4.1. *Given a source $s \in V$ and a target $t \in V$, the number of nodes that are settled by the SKARF query is upper bounded by* $\max(\tilde{\kappa}, \tilde{\kappa}')((\delta+2)d(s,t) + 2D_{\max} + 2)$.

Proof. By Lemma 4.7 there can be at most $\tilde{\kappa}\lceil d(s,t) + D_{\max}\rceil$ settled nodes on $T^*_{C_s}$ and by Lemma 4.8 at most $\tilde{\kappa}'\lceil(\delta+1)d(s,t)+D_{\max}\rceil$ settled nodes on $T'^*_{C_t}$. Since all settled nodes are in $T^*_{C_s} \cup T'^*_{C_t}$, in total there can be at most

$$\begin{aligned}&\tilde{\kappa}\lceil d(s,t) + D_{\max}\rceil + \tilde{\kappa}'\lceil(\delta+1)d(s,t) + D_{\max}\rceil\\ \leq &\max(\tilde{\kappa}, \tilde{\kappa}')\left((\delta+2)d(s,t) + 2D_{\max} + 2\right)\end{aligned}$$

settled nodes. ∎

Recall that the SKARF query is a modified Dijkstra search. By Fredman and Tarjan [14], a Dijkstra query that scans n' nodes and $\Delta n'$ edges has a runtime complexity of $\mathcal{O}(\Delta n' + n' \log n')$. This leads us to the final query runtime of SKARF.

COROLLARY 4.2. *Given a source $s \in V$ and a target $t \in V$, the SKARF query runs in $\mathcal{O}(\Delta n' + n' \log n')$ with* $n' := \max(\tilde{\kappa}, \tilde{\kappa}')\left((\delta+2)d(s,t) + 2D_{\max} + 2\right)$.

Proof. By Lemma 4.6 $|T^*_{C_s}| = |\text{Ball}^D_{z(C_s)}(T^*_{C_s})| \leq \tilde{\kappa}\lceil D\rceil$ and $|T'^*_{C_t}| = |\text{Ball}^D_{z(C_t)}(T'^*_{C_t})| \leq \tilde{\kappa}'\lceil D\rceil$ hold. The number of all settled nodes is at most $|T^*_{C_s} \cup T'^*_{C_t}| \leq \tilde{\kappa}\lceil D\rceil + \tilde{\kappa}'\lceil D\rceil$, which is in $\mathcal{O}(\tilde{\kappa}D)$. Then the query, which runs a Dijkstra on $\mathcal{O}(\tilde{\kappa}D)$ nodes, has a runtime complexity in $\mathcal{O}(\Delta\tilde{\kappa}D + \tilde{\kappa}D\log(\tilde{\kappa}D))$. ∎

COROLLARY 4.3. *Assuming $\delta \in \mathcal{O}(1)$ and $\tilde{\kappa}' \in \mathcal{O}(\tilde{\kappa})$, the SKARF query has a runtime complexity in $\mathcal{O}(\Delta\tilde{\kappa}D + \tilde{\kappa}D\log(\tilde{\kappa}D))$.*

4.4 Bidirectional SKARF Extending SKARF to a bidirectional version, we improve the result of Lemma 4.8 where we used bounded asymmetry to accomplish a statement about the search space inside the target cell skeleton. Routing on target cell skeletons in the forward search can be avoided by searching the target cell skeleton only within the backward search. This makes the assumption of bounded asymmetry unnecessary.

Algorithm. Bidirectional SKARF uses the same preprocessing as SKARF. During the query a forward Dijkstra search is started from s that relaxes an edge e if and only if $X_e(C_s) = \textit{true}$ and a backward Dijkstra search is started on G' from t that relaxes an edge e if and only if $Y_e(C_t) = \textit{true}$. The processing order is obtained by comparing the top priority queue elements of both searches and choosing the one with the smaller distance to its search origin. The query stops as soon as a node is settled by both searches.

Correctness. It is shown by Pohl [23] that bidirectional Dijkstra is correct. By Lemma 4.4 the shortest path is always contained in the union of the source cell skeleton and the target cell skeleton. To complete the correctness proof of bidirectional SKARF, it only has to be guaranteed that some node is settled by both searches so that the query terminates. Indeed, it can be proven that there is always a node on $P(s,t)$ that is in both the skeleton of s and the skeleton of t in the transposed graph and can therefore be settled by both searches. Lemma 4.9 captures this statement.

LEMMA 4.9. *For a source $s \in V$ and a target $t \in V$ there exists some node $v \in V$ such that $v \in P(s,t)$ and $v \in T^*_s \cap T'^*_t$.*

Proof. Let $e = (u,v) \in E$ such that $e \in P(s,t)$ and $d(s,u) < \frac{1}{2}d(s,t) \leq d(s,v)$. Then

$$\text{Reach}_s(v) + l(e) \geq d(u,t) > \frac{1}{2}d(s,t) > d(s,u),$$

meaning that $(u,v) \in T^*_s$. Let $\text{Reach}'_t(u)$ denote the reach of u in T'_t. It holds that

$$\begin{aligned}\text{Reach}'_t(u) + l'((v,u)) &\geq d_{T'_t}(v,s)\\ &\geq \frac{1}{2}d_{T'_t}(t,s)\\ &\geq d_{T'_t}(t,v).\end{aligned}$$

If $\text{Reach}'_t(u) + l'((v,u)) > d_{T'_t}(t,v)$, then $(v,u) \in T'^*_t$ follows. If otherwise $\text{Reach}'_t(u) + l'((v,u)) = d_{T'_t}(t,v)$, then for the next edge $(v,w) \in P(s,t)$ it follows that

$$\begin{aligned}&\text{Reach}'_t(v) + l'((w,v))\\ \geq\ &\text{Reach}'_t(u) + l'((v,u)) + l'((w,v))\\ =\ &d_{T'_t}(t,v) + l'((w,v))\\ >\ &d_{T'_t}(t,w)\end{aligned}$$

and therefore $(w, v) \in T_t'^*$. In all cases $v \in T_s^* \cap T_t'^*$ holds. ∎

Thus, as soon as bidirectional SKARF settles a node in both searches, it terminates and the correct result can be obtained by finding the node $v \in V$ that minimizes $d(s, v) + d(v, t)$. The shortest path is then given by the combination of $P(s, v)$ and $P(v, t)$.

Search Space. Using the property that in bidirectional SKARF, both searches relax only edges that are in the source cell skeleton or the target cell skeleton, respectively, we can replace the result of Lemma 4.8 with a new one that is rather similar to Lemma 4.7 and can be proven analogously to it.

LEMMA 4.10. *Given a source $s \in V$ and a target $t \in V$, the number of nodes settled by the backward search of bidirectional SKARF in $T_{C_t}'^*$ is upper bounded by $\tilde{\kappa}'\lceil d(s, t) + D_{\max}\rceil$.*

Proof. Let $v \in T'_{C_t}$ be a node that is settled by the backward search of bidirectional SKARF. Since v got settled before s, it follows that $d_{G'}(t, v) \leq d_{G'}(t, s) = d(s, t)$ and thus

$$d_{G'}(z(C_t), v) \leq d_{G'}(z(C_t), t) + d_{G'}(t, v) \leq D_{\max} + d(s, t).$$

Then, every settled node is inside $\text{Ball}_{z(C_t)}^{d(s,t)+D_{\max}}(T_{C_t}'^*)$ and by Lemma 4.6 we obtain $|\text{Ball}_{z(C_t)}^{d(s,t)+D_{\max}}(T_{C_t}'^*)| \leq \tilde{\kappa}'\lceil d(s, t) + D_{\max}\rceil$. ∎

THEOREM 4.2. *Given a source $s \in V$ and a target $t \in V$, the number of nodes that are settled by the bidirectional SKARF query is upper bounded by $2\max(\tilde{\kappa}, \tilde{\kappa}')\lceil d(s, t) + D_{\max}\rceil$.*

This theorem can be proven analogously to Theorem 4.1. The only difference is that instead of using Lemma 4.8 we use Lemma 4.10, which gives the improvement. We now use this to obtain the following.

COROLLARY 4.4. *Given a source $s \in V$ and a target $t \in V$, the bidirectional SKARF query has a runtime complexity of $\mathcal{O}(n'\Delta + n'log(n'))$, where $n' := 2\max(\tilde{\kappa}, \tilde{\kappa}')\lceil d(s, t) + D_{\max}\rceil$.*

Note that, as claimed above, Corollary 4.4 is entirely independent of our bounded asymmetry assumption.

5 SKARF+

The *SKARF+* algorithm combines the goal-directed speed-up technique Arc-Flags with the additional search space restrictions of SKARF. For a source $s \in V$ and a target $t \in V$, it runs a Dijkstra query that relaxes only edges $e \in E$, which have the according Arc-Flags and SKARF flags set, i.e., $e \in \bigcup_{v \in C_t} T'_v$ and $e \in T^*_{C_s} \cup T'^*_{C_t}$.

Since both Arc-Flags and SKARF allow every edge that lies on a shortest path between the relevant cells of a query to be visited, the combination of the two algorithms also allows every such edge to be visited, which in turn gives us the correctness of SKARF+.

The flag computation runtime is identical to the one of SKARF, since the flag computations for SKARF include building shortest path trees from all boundary nodes. During this process, the Arc-Flags flags can be saved as well.

If the flag vectors of SKARF and Arc-Flags are saved individually, the space consumption of SKARF+ is exactly the sum of the space consumption of SKARF and Arc-Flags. However, further optimization is possible because given an edge e and a cell C, it is enough to differentiate between three different states:

1. either $e \in T_C^*$ (and thus $e \in \bigcup_{v \in C} T_v$),
2. $e \in \left(\bigcup_{v \in C} T_v \setminus T_C^*\right)$ or
3. $e \notin \bigcup_{v \in C} T_v$ (and thus $e \notin T_C^*$).

The search space of SKARF+ is at most the minimum of the ones of SKARF and Arc-Flags, although the following section demonstrates that in practice it is better.

6 Experimental Comparison of SKARF+ and Arc-Flags

To demonstrate the effectiveness of SKARF+ on real-world street graphs, we conduct an experimental study on its performance in comparison with Arc-Flags. Similarly to our experiments in Section 3, these experiments are performed on a Germany and a France street graph from OpenStreetMap [11] with about 4,000,000 nodes and 10,000,000 edges each. Both are again partitioned with METIS [17] into 50, 100, 200, 500 and 1000 cells to have results that depend on different partition sizes.

The preprocessing is performed on 50 cores and 80 GB RAM on an AMD EPYC 7742 with x86-64 architecture that is clocked at 2.25 GHz. In this scenario the preprocessing expense is feasible with between 2 and 10 hours of computing time for each partition. For Arc-Flags, PHAST is a more sophisticated preprocessing algorithm to obtain the shortest path trees and set the flags, achieving total preprocessing times of just 10 minutes on the road network of Western Europe [12]. Since our preprocessing time is dominated by the construction of the shortest path trees, PHAST can reduce it significantly. For SKARF+, only the computation of the reaches would be added, indicating that the SKARF+

Table 2: A comparison of the performance of Arc-Flags and SKARF$^+$ shortest path queries in their unidirectional and bidirectional version after running them for 10,000 random node pairs on the Germany (a) and France (b) street graphs. The performance is measured in terms of average number of settled nodes and query times.

(a) Germany street graph. Average path length: 1157.

	Arc-Flags		SKARF$^+$		Reduction [%]		bid. Arc-Flags		bid. SKARF$^+$		Reduction [%]	
Part. #cells	#sett. nodes	time [μs]	#sett. nodes	time [μs]	#sett. nodes	time [μs]	#sett. nodes	time [μs]	#sett. nodes	time [μs]	#sett. nodes	time [μs]
50	68609	282795	49778	199211	27.45	29.56	8028	32843	4487	18616	44.11	43.32
100	38322	254574	26721	172385	30.27	32.28	4478	26200	2724	15424	39.17	41.13
200	22594	92769	14881	55252	34.14	40.44	3158	12290	2031	8029	35.69	34.67
500	11390	43287	7407	27880	34.97	35.59	2034	7938	1569	6248	22.86	21.29
1000	6957	26030	4634	16848	33.39	35.27	1685	6432	1418	5682	15.85	11.66

(b) France street graph. Average path length: 832.

	Arc-Flags		SKARF$^+$		Reduction [%]		bid. Arc-Flags		bid. SKARF$^+$		Reduction [%]	
Part. #cells	#sett. nodes	time [μs]	#sett. nodes	time [μs]	#sett. nodes	time [μs]	#sett. nodes	time [μs]	#sett. nodes	time [μs]	#sett. nodes	time [μs]
50	77844	363077	56682	231869	27.19	36.14	7423	33073	3910	16723	47.33	49.44
100	43987	220477	30163	138816	31.43	37.04	4377	20624	2309	10876	47.25	47.27
200	25825	107000	16645	66299	35.55	38.04	2682	10891	1583	6504	40.98	40.28
500	12723	48480	7904	29744	37.89	38.65	1693	6401	1169	4682	30.95	26.86
1000	7703	28973	4758	17765	38.23	38.68	1356	5170	1034	4211	23.75	18.55

preprocessing time can be pushed down to less than half an hour as well.

In our query implementation we focus on a fair environment for a comparison of Arc-Flags and SKARF$^+$ and do only basic optimizations that affect both algorithms. For instance, we use a priority queue that allows for decrease key operations. In addition, we optimize the flag access by using C++ vectors and dynamic bitsets instead of slower data structures like hash maps. For tentative distances and auxiliary data for the final path reconstruction, we use hash maps. Note that, although our query times fall short of the ones presented by Bast et al. [3] (which are below one millisecond), we provide similar conditions for both algorithms to achieve a fair comparison.

To measure the query performance, we run an Arc-Flags query and a SKARF$^+$ query in their unidirectional and bidirectional version for 10,000 random node pairs. We measure the query time in micro seconds and, as a metric for the search space, the number of nodes that are settled, i.e., popped out of the Dijkstra queue. The queries were executed on the same machine as the preprocessing. Table 2 presents the resulting numbers.

With unidirectional SKARF$^+$ we achieve a search space reduction over unidirectional Arc-Flags from 27% to 35% on Germany and up to 38% on France. When comparing the corresponding bidirectional versions, SKARF$^+$ even yields a search space decrease of 35% to 44% for 50 to 200 cells on Germany and up to 47% on France. For 500 and 1000 cells however, this improvement is smaller with 16% to 31%. This can be explained by the fact that Arc-Flags performs better, the more cells are used. Therefore the potential for improvement gets smaller with larger partition sizes. The query time reductions are similar to the search space reductions with up to 43% on Germany and up to 49% on France for the bidirectional versions.

The stated improvements come with a trade-off in the space consumption. Bidirectional SKARF$^+$ needs around twice as much space for the flags as bidirectional Arc-Flags, while unidirectional SKARF$^+$ uses three times as much space as unidirectional Arc-Flags. Thus, the practical value of SKARF$^+$ lies in its reduction of search space and query time and is best applied in scenarios where the additional space can be provided and fast queries are essential.

7 Conclusion

In this paper, we consider a routing speed-up technique based on the skeleton dimension. Motivated by the fact that the skeleton dimension is small in real-world road networks and because it has been applied to analyses

of other routing speed-up techniques, we present an algorithm that exploits small skeletons.

We define the cell skeleton dimension as the maximum width of cell skeletons over all cells of a partitioned graph and compute its value on real-world instances of road networks. We observe it to be large for large cell sizes, but for more fine-grained partitions it decreases and approaches the classic skeleton dimension, making it a useful parameter for analyzing algorithms that work on partitioned graphs.

We propose the new routing algorithm SKARF, which is based on the observation that for a source and a target, the combination of their skeletons covers the complete shortest path between them. The idea of SKARF is to reduce the search space in the query to the cell skeletons of the start and target nodes. Thus, we provide a theoretical search space size that is bounded in the cell skeleton dimension. In addition, we show that the preprocessing time and the space consumption remain asymptotically the same as in Arc-Flags.

Moreover, we propose the SKARF$^+$ algorithm as a combination of SKARF and Arc-Flags to obtain an algorithm uniting the goal-directed nature of Arc-Flags with the search space restrictions of SKARF. In our experimental analysis we find that SKARF$^+$ performs exceptionally well, with a search space and query time improvement of about 30% to 40% in comparison to Arc-Flags.

Our theoretical and empirical results on SKARF show that the skeleton dimension, a graph parameter that explains the effectiveness of other routing speed-up techniques, can be used for developing new techniques. In particular, the cell skeleton approach can greatly reduce the search space of routing algorithms.

Outlook. Although combining Arc-Flags with SKARF already achieves significant speed-ups, this does not use the full potential of skeleton-based techniques. The idea of combining different flags can be extended by generalizing the concept of skeletons. So far, we considered skeletons that contain a fraction of $p = \frac{1}{2}$ of all shortest paths. But, we can also use $p = \frac{a}{b}$ for start cells and $p = \frac{b-a}{b}$ for target cells to guarantee that shortest paths are completely covered by a combination of two cell skeletons. Such skeleton combinations would yield individual search space restrictions that maintain shortest paths. Thus, for all possible ratios an individual SKARF version can be implemented and combined with the other ones. From this perspective, Arc-Flags is also one such SKARF version, saving cell skeletons with $p = 1$ on the transposed graph, and with $p = 0$ on the initial graph.

Such an idealized combination would finally exhaust the potential of skeletons and achieve even better search space restrictions and query times. To avoid infinite space consumption, one could approximate this idealized SKARF combination by saving only a few different cell skeletons. Instead of a one bit flag that enables the information necessary to distinguish skeleton from non-skeleton, an n-bit flag could enable distinguishing between $2^n - 1$ different skeletons and the state of an edge of not being on any of those. This is possible because smaller skeletons are always subsets of larger skeletons. For a 4-bit approximation of the ideal SKARF version, preliminary experiments indicate similar search space improvements over SKARF$^+$ as SKARF$^+$ has over Arc-Flags.

Having achieved search space guarantees and query time speed-ups for an Arc-Flags variant, the next natural step is to employ SKARF$^+$ in more sophisticated algorithms that use Arc-Flags as a subroutine, i.e., nested Arc-Flags, SHARC, CHASE and Reach-Flags. This may enable theoretical analyses of these algorithms and lead to further speed-ups. It remains an open question to what extent the improvements of SKARF$^+$ over Arc-Flags transfer to the algorithms mentioned above. Improving these algorithms would contribute to advancing state-of-the-art routing.

References

[1] Ittai Abraham et al. "A Hub-Based Labeling Algorithm for Shortest Paths in Road Networks". In: *Experimental Algorithms (SEA)*. 2011, pp. 230–241. DOI: 10.1007/978-3-642-20662-7_20.

[2] Ittai Abraham et al. "Highway Dimension and Provably Efficient Shortest Path Algorithms". In: *Journal of the ACM* 63.5 (2016), pp. 1–26. DOI: 10.1145/2985473.

[3] Hannah Bast et al. "Route Planning in Transportation Networks". In: *Algorithm Engineering: Selected Results and Surveys*. 2016, pp. 19–80. DOI: 10.1007/978-3-319-49487-6_2.

[4] Holger Bast et al. "Fast Routing in Road Networks with Transit Nodes". In: *Science* 316.5824 (2007), pp. 566–566. DOI: 10.1126/science.1137521.

[5] Reinhard Bauer and Daniel Delling. "SHARC: Fast and Robust Unidirectional Routing". In: *ACM Journal of Experimental Algorithmics* 14 (2010). DOI: 10.1145/1498698.1537599.

[6] Reinhard Bauer et al. "Combining Hierarchical and Goal-Directed Speed-up Techniques for Dijkstra's Algorithm". In: *ACM Journal of Experimental Algorithmics* 15 (2010). DOI: 10.1145/1671970.1671976.

[7] Reinhard Bauer et al. "On the Complexity of Partitioning Graphs for Arc-Flags". In: *Journal of Graph Algorithms and Applications [electronic only]* 17 (Jan. 2013). DOI: 10.4230/OASIcs.ATMOS.2012.71.

[8] Reinhard Bauer et al. "Search-space size in contraction hierarchies". In: *Theoretical Computer Science* 645 (2016), pp. 112–127. DOI: 10.1016/j.tcs.2016.07.003.

[9] Johannes Blum, Stefan Funke, and Sabine Storandt. "Sublinear search spaces for shortest path planning in grid and road networks". In: *Journal of Combinatorial Optimization* 42.2 (2021), pp. 231–257. DOI: 10.1007/s10878-021-00777-3.

[10] Johannes Blum and Sabine Storandt. "Computation and Growth of Road Network Dimensions". In: *International Conference on Computing and Combinatorics (COCOON)*. 2018, pp. 230–241. DOI: 10.1007/978-3-319-94776-1_20.

[11] OpenStreetMap contributors. *Germany street graph from https://download.geofabrik.de*. 2018. (Visited on 08/18/2022).

[12] Daniel Delling et al. "PHAST: Hardware-Accelerated Shortest Path Trees". In: *IEEE International Symposium on Parallel and Distributed Processing, (IPDPS)*. 2011, pp. 921–931. DOI: 10.1109/IPDPS.2011.89.

[13] Edsger W. Dijkstra. "A Note on Two Problems in Connexion with Graphs". In: *Numerische Mathematik* 1 (1959), pp. 269–271. DOI: 10.1007/BF01386390.

[14] Michael L. Fredman and Robert Endre Tarjan. "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms". In: *Journal of the ACM* 34.3 (1987). DOI: 10.1145/28869.28874.

[15] Robert Geisberger et al. "Exact Routing in Large Road Networks Using Contraction Hierarchies". In: *Transportation Science* (2012). DOI: 10.1287/trsc.1110.0401.

[16] Ronald J. Gutman. "Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks". In: *Workshop on Algorithm Engineering and Experiments and Workshop on Analytic Algorithmics and Combinatorics (ALENEX/ANALC)*. 2004, pp. 100–111.

[17] George Karypis and Vipin Kumar. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs". In: *SIAM Journal on Scientific Computing* 20 (1998), pp. 359–392. DOI: 10.1137/S1064827595287997.

[18] Adrian Kosowski and Laurent Viennot. "Beyond Highway Dimension: Small Distance Labels Using Tree Skeletons". In: *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2017, pp. 1462–1478. DOI: 10.1137/1.9781611974782.95.

[19] Ulrich Lauther. "An Experimental Evaluation of Point-To-Point Shortest Path Calculation on Road Networks with Precalculated Edge-Flags". In: *The Shortest Path Problem*. 2006.

[20] Ulrich Lauther. "An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background". In: *Geoinformation und Mobilitaet – von der Forschung zur praktischen Anwendung*. Vol. 22. IfGI prints, Institut fuer Geoinformatik, 2004, pp. 219–230.

[21] Rolf H. Möhring et al. "Partitioning Graphs to Speed Up Dijkstra's Algorithm". In: *Experimental and Efficient Algorithms (WEA)*. 2005, pp. 189–202. DOI: 10.1007/11427186_18.

[22] Juan Mori and Samitha Samaranayake. "Bounded Asymmetry in Road Networks". In: *Scientific Reports* 9 (2019). DOI: 10.1038/s41598-019-48463-z.

[23] Ira Pohl. *Bi-directional and heuristic search in path problems*. Tech. rep. Stanford Linear Accelerator Center, 1969.

[24] Heiko Schilling. *TomTom navigation - How mathematics help getting through traffic faster*. Talk given at ISMP. 2012.

[25] Christian Sommer. "Shortest-path queries in static networks". In: *ACM Computing Surveys (CSUR)* 46.4 (2014), pp. 1–31. DOI: 10.1145/2530531.