An Approximate Distance Oracle for Social Networks

Bachelor Thesis of

Katharina Flügel

At the Department of Informatics
Institute of Theoretical Computer Science

Reviewers:	Prof. Dr. Peter Sanders
	Prof. Dr. Dorothea Wagner
Advisors:	Prof. Guy E. Blelloch
	Laxman Dhulipala
	Yan Gu

9th November 2018

Statement of Authorship

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

Karlsruhe, 9th November 2018

Abstract

Computing shortest paths is a fundamental graph problem with many real-life applications. Naive methods such as Dijkstra's algorithm are not fast enough for today's graphs with millions of vertices and edges. We present *approximate transit node routing (aTNR)*, an approximate distance oracle for social networks with a limited additive error. It is an extension to the transit node routing framework which was originally developed for road networks. Distances can be computed significantly faster with aTNR compared to the naive approaches. We can adjust the maximum error for each graph and even compute exact distances at the cost of a longer preprocessing time.

The distance between two vertices is computed using a three-hop via the transit nodes, a small set of central vertices. For each vertex, a set of access nodes is chosen from the transit nodes via which the three-hop is computed for that vertex.

We evaluate aTNR on multiple different graphs regarding the preprocessing time, preprocessing size, query time, and approximation error. Smaller graphs are preprocessed in mere seconds, and even graphs with more than one hundred million edges can be preprocessed in less than two minutes. Most queries take about $20\,\mu s$ and cause only a very small approximation error. Our distance oracle is particularly proficient in terms of space consumption as it requires less than 50 bytes per vertex. In addition to aTNR, we define the *Voronoi search distance oracle*, a variation of aTNR which increases the approximate error in exchange for potentially faster preprocessing and queries.

Zusammenfassung

Die Berechnung kürzester Wege ist eines der grundlegendsten Graphprobleme. Es ist Grundbaustein zahlreicher Algorithmen und wird in vielen Anwendungen genutzt. Viele der heutigen Graphen haben Millionen von Knoten und Kanten. Naive Algorithmen, wie beispielsweise Dijkstras-Algorithmus, sind deshalb oftmals nicht schnell genug, um die immer größer werdenden Datenmengen zu verarbeiten.

Diese Bachelorarbeit definiert *Approximate Transit Node Routing (aTNR)*, ein approximatives Distanzorakel für soziale Netzwerk-Graphen, das Distanzen zwischen zwei Knoten mit einem begrenzten, additiven Fehler berechnet. Es ist eine Erweiterung des Transit-Node-Routing-Frameworks, welches ursprünglich für Straßennetzwerke entwickelt wurde, und berechnet Distanzen deutlich schneller als naive Algorithmen. Der maximale Approximationsfehler ist variabel, sodass selbst exakte Distanzen berechnet werden können.

Kürzeste-Wege-Anfragen werden mit Hilfe eines 3-Hops über eine kleine Menge zentraler Knoten – sogenannter Transit Nodes – berechnet. Jedem Knoten wird eine Menge von Access Nodes zugewiesen, die aus den Transit Nodes gewählt werden und über die alle 3-Hop-Distanzen des entsprechenden Knotens berechnet werden.

Wir analysieren aTNR auf verschiedenen Graphen bezüglich der Vorberechnungszeit, der Vorberechnungsgröße, der Laufzeit der Anfragen und dem Approximationsfehler. Kleinere Graphen werden in wenigen Sekunden vorberechnet. Selbst Graphen mit mehr als 100 Millionen Kanten benötigen weniger als zwei Minuten Vorberechnungszeit. Anfragen können in etwa $20\,\mu s$ und mit geringem Approximationsfehler beantwortet werden. Unser Distanzorakel sticht vor allem durch seinen geringen Platzverbrauch von weniger als 50 Bytes pro Knoten auf sozialen Netzwerk-Graphen hervor.

Zusätzlich zu aTNR präsentieren wir *Voronoi Search Distanzorakel*, eine Variation von aTNR mit potenziell schnellerer Laufzeit aber größerem Approximationsfehler.

Acknowledgements

Contents

1. Introduction

Computing shortest paths is a fundamental graph problem [DSGNP10, AIY13] and a building block for many graph applications and algorithms [DSGNP10, AIY13, QXSW13]. The shortest path distance between two vertices can, for example, indicate the closeness between two users of a social network [AIY13, VFD+07]. It can be used in socially-sensitive search, recommending related users or content to the user of a social network, or context-aware search, finding related web-pages [AIY13, YBLS08, UCDG08]. Moreover, measurements like betweenness centrality [Fre77, Bra01] and network diameter are based on shortest path distances [QXSW13].
Large graphs have become very common within the last decades [QXSW13, DSGNP10]. Using a simple shortest path computation without any preprocessing is now often too slow [DSGNP10, BFSS07]. Dijkstra's algorithm [Dij59], for example, is a well-known algorithm solving the single-source shortest path problem on graphs with non-negative edge weights. However, computing Dijkstra's algorithm takes more than a second on large graphs which is "too slow for many applications" [BFSS07]. Even on unweighted graphs, where a simple breadth-first search suffices to determine shortest paths, a preprocessing is often necessary to reach fast enough query times for real-time interactions [AIY13].

In many cases, multiple shortest path queries are computed on the same instance. It is often possible to spend some time on computing auxiliary data if that speeds up the queries. This approach is the idea behind *distance oracles* [TZ05], an algorithmic approach solving multiple shortest path queries on the same graph in near constant time by reusing a preprocessing. As shown by Thorup and Zwick [TZ05], constant query time with sub-quadratic space requirement is assumed to be unreachable on general graphs without approximation. However, it is often sufficient to compute the distance within a certain approximation. This can speed up the computation significantly.

Route planning on road networks has been researched very thoroughly. There is a wide range of fast algorithms solving various problems related to shortest paths on road networks. Road network graphs share a special set of properties. They have a very low, constant vertex degree and are usually almost planar [BFM+07]. Furthermore, there is a hierarchy of the edges in road networks when using the travel time as edge weight. Far away from the start and end point, shortest paths tend to use only edges high up the hierarchy [BFM+07]. These properties can be used as an advantage when computing shortest paths on road networks.
In more "complex networks such as social networks, web graphs, biological networks and computer networks" [AIY13], there has not been as much research on route planning. Since

they generally do not share these properties with road networks, solving distance queries on these types of graphs "is still a highly challenging problem" [AIY13]. In this thesis, we generalize *transit node routing* (TNR) [BFSS07, BFM+07, ALS13], a very fast algorithm solving the shortest paths problem on road networks, to work on more general graphs such as the aforementioned social networks and web graphs.

Our Contribution

This thesis presents two different variants of an approximate distance oracle based on transit node routing. Both modify the transit node routing approach, that was initially developed for road networks, to work on more complex graphs such as social networks and web graphs.

Approximate transit node routing (aTNR) approximates the distance between any two vertices in the graphs with an additive error of at most $4r$ where $r \geq 0$. We extend the TNR framework to allow approximate distances by introducing *marked nodes*, which are all vertices within a distance of r of a transit node. The access nodes are then selected with a single-source shortest path search from every (unmarked) vertex that is pruned at marked vertices. By this pruning, we avoid computing a complete single-source shortest path search for all vertices. The maximum error can be adjusted with the tuning parameter r to the specific needs of the application. Allowing a greater error leads to faster preprocessing and query times. We can guarantee that regardless of the value chosen for r, the computed distance is never shorter than the actual shortest path distance. Furthermore, we evaluate different heuristics of selecting the transit nodes – the landmarks used to compute a 3-hop distance – specifically for our distance oracle.

We evaluate our distance oracle on multiple different network graphs with 150 thousand to 100 million edges. Smaller networks are processed in mere seconds and even larger social networks, with up to one hundred million edges, can be preprocessed in only a few minutes. Our preprocessing generally requires less space with a maximum of 50 bytes per vertex on social networks and web graphs. The global query computes most distances correctly within about a microsecond but cannot guarantee a maximum error. Combined queries, which guarantee the maximum error of $4r$, require about $20\,\mu s$ and have an average error of less than r. The maximum error is only reached by few outliers. We also tested aTNR on computer and road network graphs. The computed distances do not exceed the maximum error, but both the preprocessing and the queries are not as efficient as on social networks, for which aTNR is designed.

Additionally, we present a *Voronoi search distance oracle* that does not guarantee an upper bound for the computed distance. It is a modification of aTNR and is motivated by the fact that on many inputs aTNR yields on average a much smaller error than the allowed $4r$. The single-source shortest path searches computing the access nodes are pruned at the Voronoi diagram borders which reduces their running time significantly.
Both of our algorithms can be applied to both directed and weighted graphs and are described as such. Our evaluation, however, considers only undirected and unweighted graphs due to time constraints.

The remainder of this chapter gives an overview of previous work related to our contribution. After that, Chapter 2 defines the notation and terminology used in this thesis and gives a brief overview of the algorithms on which our distance oracle is based. The two contributed distance oracles are described in Chapter 3 and 4. At first, Chapter 3 defines our main contribution – approximate transit node routing for social networks. After that, Chapter 4 illustrates the necessary modifications to obtain the Voronoi search distance oracle. An overview of the implementation details, such as the used tools and considerations about data structures, is given in Chapter 5. Chapter 6 contains the experimental results of

our approach and compares them to existing algorithms. Finally, our contribution is summarized in Chapter 7 which also gives an outlook on possible future work.

1.1. Related Work

Much work has been done on computing shortest paths and their length in graphs. Starting from Dijkstra's algorithm [Dij59], which solves the complete single-source shortest path problem on all graphs with non-negative edge weights, we today reached many more specialized algorithms that yield much faster query times by speeding the queries up with preprocessed data. These algorithms are often limited to a specific class of graphs and take advantage of their specific properties.

This thesis modifies a speedup technique designed initially for road networks to an approximate distance oracle for social networks. We therefore give a brief overview of both speedup techniques in road networks, and social and communication networks. Furthermore, a short summary of distance oracles is given.

Shortest Distances in Road Networks

Speedup techniques on road networks have been researched very thoroughly. Delling et al. [DSSW09] and Bast et al. [BDG$^+$16] present a good overview of the different techniques. An experimental evaluation of multiple shortest path and distance queries on road networks is presented by [WXD$^+$12].

Contraction hierarchies (CH) [GSSD08, GSSV12] are a fast and simple technique to find shortest paths in road networks. It is based on successively contracting the vertices "and adding shortcut edges [...] to preserve [the] shortest path distances between the remaining nodes" [GSSV12]. The query is based on a bidirectional version of Dijkstra's algorithm and has a very small search space of "only a few hundred nodes" [GSSV12].

Transit node routing (TNR) [BFSS07, BFM$^+$07] is a framework for solving shortest path queries. A small set of important *transit nodes* is selected via which all long-distance connections are routed. TNR is covered in more detail in Section 2.4. An implementation of TNR with CH is described by [ALS13] and yields even faster query times with only about twice the preprocessing time of CH.

Another speedup technique is *hub labeling* [CHKZ03, GPPR04]. Every vertex is assigned a label consisting of a set of hubs and their distance such that the shortest path is covered by the intersection of the labels of the start and target vertex. In general, the labels can be very large. However, Abraham et al. [ADGW11] reached good results on road networks using the upward CH search spaces. Their queries are significantly faster than the previously mentioned CH-TNR [ALS13] and consume less space than some TNR implementations [ADGW11]. Still, it has a "much higher space consumption" [ALS13] than CH-TNR. Hub labeling has been extended to *hierarchical hub labeling* [ADGW12] which yields good results on even more graph classes than just road networks.

Furthermore, variations of the classical shortest path problem have been studied. This includes, for example, route planning for electric vehicles [EFS11, BDPW13, BDG$^+$15], in public transit networks [DKP12], or on graphs with dynamic edges weights that are changed to adjust the travel time to the current traffic conditions [BGSV13, BDPW16, Str17].

Shortest Distances in Social Networks and Web Graphs

Most of the previously mentioned approaches make use of the special properties of road networks, such as a low vertex degree and a natural edge hierarchy. In general, social networks and web graphs do not possess these characteristics. We now give a short overview of different approaches to compute shortest paths on these types of graphs.

Potamias et al. state that "exact algorithms do not scale to huge graphs encountered

on the web, social networks, and other applications"[PBCG09]. They therefore introduce an approximate approach based on landmark-based distance indexing. It uses different heuristics to select landmarks because they prove that selecting optimal landmarks is NP-hard.

A sketch-based distance oracle is given by [DSGNP10]. It computes a "sketch of the neighborhood structure" [DSGNP10] for every vertex in the preprocessing phase and estimates the distance between two vertices using their sketches. This yields a simple algorithm that is scalable to web graphs and works on both undirected and directed graphs.

A modification of this sketch-based approach is introduced by [GBSW10]. It focuses on graphs with short path lengths and stores complete path sketches. They observe that this applies to "most real-world graph data (apart from road networks)" [GBSW10]. In total, they reach "several orders of magnitude speedup over traditional path computations" [GBSW10] with average errors of less than one percent.

Tretyakov et al. [TACGB+11] improve the previous landmark based approaches on undirected graphs by storing a shortest path tree for every landmark. Furthermore, they also improve the landmark selection itself, increasing the number of shortest paths covered by the landmarks. They achieve higher accuracy with query times "within a few milliseconds [. . .] and a space consumption comparable to previous state of the art methods" [TACGB+11]. Additionally, their approach supports dynamic updates of the graph such as edge insertions and deletions.

Highway-Centric Labeling [JRXL12] is another approach to compute both exact and approximate distances on large sparse graphs. It is a variation of 2-hop labeling schemes, such as hub labeling [CHKZ03, GPPR04], but uses a *tree structured highway* to connect the source and target vertex and compute their distance. According to [JRXL12], Highway-Centric Labeling can outperform state-of-the-art algorithms in both space consumption and query time.

Akiba et al. [AIY13] also introduce an exact algorithm based on 2-hop labeling. Their algorithm conducts a breadth-first search (BFS) from every vertex but prunes these searches with the already selected labels. This is very similar to our distance oracle, which is also based on computing pruned BFS from most vertices. However, the decision of when to prune such a search is different from our approach. Additionally, they compute multiple BFS simultaneously using bitwise operations. This yields an efficient and scalable algorithm with query times of about ten microseconds. It can handle both directed and weighted graphs, but results are only presented for undirected and unweighted graphs.

Distance Oracles and Spanners

Approximate distance oracles were initially introduced by Thorup and Zwick [TZ05]. There has since been a lot of research on them and the related spanners. We now present a short overview of relevant works on distance oracles and spanners. A definition of both distance oracles and spanners is given in Section 2.3.

On general graphs, Thorup and Zwick [TZ05] define a distance oracle with preprocessing time $O(kmn^{1/k})$, space $O(kn^{1+1/k})$ and query time $O(k)$ with a stretch of $2k-1$ for any integer $k \geq 1$. For fixed k, this results in a constant query time. An $O(n^2 \log n)$ algorithm with a data structure of size $O(kn^{1+1/k})$ and query times of $O(k)$ for $k > 2$ and $O(\log n)$ for $k = 2$ is presented by [BK06]. Baswana and Sen show in [BS06] that approximate distance oracles for unweighted graphs can be constructed in expected $O(n^2)$ time. Additionally, they present the "first expected linear-time algorithm for computing an optimal size $(2, 1)$-spanner of an unweighted graph" [BS06]. Pătraşcu and Roditty [PR10] further improve the space-approximation trade-off and give a distance oracle for unweighted graphs with $O(n^{5/3})$ space complexity, a stretch of two, and an expected preprocessing time of $O(m \cdot n^{2/3})$.

2. Preliminaries

This chapter defines the notation used in the following chapters and provides the foundations on which our results are based. Section 2.1 gives a short overview of the basic concepts of graph theory. After that, Section 2.2 defines Voronoi diagrams, their extension to Graph Voronoi diagrams, and how to compute them. A more formal definition of distance oracles is given in Section 2.3 extending the short overview given in Section 1.1. Section 2.4 summarizes the original transit node routing algorithm for road-networks, the foundation of our approximate distance oracles. Finally, Section 2.5 describes three covering search algorithms used as building blocks in our algorithm.

2.1. Graph Theory

This section introduces the basic notations and definitions of graph theory used in this thesis. Furthermore, Section 2.1.2 and 2.1.3 describe two well-known graph algorithms used heavily in our distance oracle – breadth-first search and Dijkstra's algorithm. More details can be found, for example, in the books by Cormen et al. [CLRS09], Mehlhorn and Sanders [MS08], or Korte and Vygen [KV18].

2.1.1. Graphs

A *graph* $G = (V, E)$ consists of n vertices and m edges. In general, we describe our algorithms for directed graphs with non-negative edge weight $w(e) \in \mathbb{R}_0^+, e \in E$. Our implementation, however, only supports undirected and unweighted graphs. We therefore sometimes differentiate between these different types. The reverse graph $G_r = (V, E_r)$ is created by reversing the direction of all edges in the directed graph $G = (V, E)$. We define a unique *vertex ID* from $\{i \in \mathbb{N}_0 \mid 0 \le i \le n - 1\}$ for every vertex and use it synonymous with the vertices $v \in V$. This allows us, for example, to use a vertex as an array index simplifying the notation for many algorithms.

"A *path* $p = \langle v_1, v_2, \ldots, v_k \rangle, v_i \in V$ is a sequence of [vertices] connected by the edges" [MS08, page 50]. We say p *contains* a vertex v or it goes *via vertex* v if $v \in \{v_1, \ldots, v_k\}$. The *length* or *weight* of a path p is defined as the sum $w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$ of its edge weights. For unweighted graphs, this is simply the number of edges.

The path p^* from v_1 to v_k is called the *shortest-path* from v_1 to v_k if there is no other path connecting v_1 and v_k with a shorter distance than $w(p^*)$, that is

$$\forall\ v_1\text{-}v_k\text{-paths } p : w(p^*) \le w(p).$$

There can be multiple shortest paths between two vertices v and w. The *distance* $\delta(v, w)$ of two vertices $v, w \in V$ is defined as the weight of the shortest path p^* from v to w, i.e., $\delta(v, w) = w(p^*)$. For every vertex $v \in V$ $\delta(v, v) = w(\langle v \rangle) = 0$ applies.

We use $v \rightsquigarrow w$ to denote the shortest path p^* from v to w. However, $v \rightsquigarrow w$ is not well-defined since there can be multiple shortest paths between v and w. We therefore use this notation only if it is irrelevant, which exact shortest path is chosen.

A *subpath* $p_{[v_i, v_j]}$ of the path p is defined as $p_{[v_i, v_j]} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ with $0 \leq i \leq j \leq k$. Two paths $p_1 = \langle v_1, \ldots, v_k \rangle, v_i \in V$ and $p_2 = \langle w_1, \ldots, w_l \rangle, w_i \in V$ in the graph $G = (V, E)$ can be *concatenated* to a path $p_3 = p_1; p_2 = \langle v_1, \ldots, v_k, w_1, \ldots, w_l \rangle$. Every subpath of a shortest path is itself a shortest path [CLRS09, page 645].

A vertex w is *reachable* from $v \in V$ if there is a path connecting v to w. Otherwise, w is *unreachable* and the distance is defined as $\delta(v, w) = \infty$ by convention. For many problems, such as computing shortest paths, it is sufficient to consider only reachable vertices [KV18, page 26]. We therefore consider only the largest strongly connected component of a graph.

2.1.2. Breadth-First Search (BFS)

Breadth-first search (BFS) is a simple graph-traversal algorithm used as a building block or base in many more complex algorithms [CLRS09, page 594]. It can be used to compute the shortest path distance in unweighted graphs and has a running time of $O(n + m)$. A proof of the complexity and correctness can be found, for example, in [CLRS09, page 594-602].

The BFS traverses the vertices in a graph $G = (V, E)$ starting at the source $s \in V$. Vertices are processed in so-called *levels*, sets of vertices all with the same unweighted distance from s. The initial level contains only s itself. A level is processed by visiting every vertex and adding its unvisited out-neighbors to the next level. This is continued until all vertices reachable from the source are visited.

Additionally, it defines a so-called *search tree* $T = (V_T, E_T)$ consisting of all visited vertices and the edges $(\mathtt{p}[v], v)$ traversed by the search when visiting a vertex v from its parent $\mathtt{p}[v]$.

2.1.3. Dijkstra's Algorithm

Dijkstra's algorithm [Dij59] is one of the best-known algorithms to solve the single-source shortest path problem. It computes the correct shortest distance $\mathtt{d}(v) = \delta(s, v)$ from a given source $s \in V$ to all other vertices $v \in V$. Using a Fibonacci-Heap [FT87], Dijkstra's algorithm has a running time of $O(n \log n + m)$ which is optimal for non-negative edge weights [KV18, page 162]. For different sets of edge weights even faster running times can be achieved.

We give a short overview of the algorithm in the following. A more detailed description including proofs for its correctness and complexity can, for example, be found in [CLRS09, page 658-662], [KV18, page 161], or [MS08, page 196-199].

Dijkstra's algorithm manages two sets of vertices: *visited* and *queued*. A queued vertex u is *visited* by *relaxing* all its outgoing edges (u, v). An edge (u, v) is relaxed, by testing whether the shortest s-v-path can be improved using the shortest path to u and the edge (u, v). Initially, all vertices are unvisited, and the source s is the only queued vertex with a distance of 0. In every step, the algorithm selects the queued vertex with minimal distance and visits it. This is repeated until all vertices reachable from s are visited.

As in Section 2.1.2, a search tree T can be defined containing the shortest paths computed by the algorithm.

2.2. Voronoi Diagrams

Voronoi diagrams were originally defined for points in $\mathbb{R}^2$ and the Euclidean distance

$$d(x,y) = \sqrt{\left(\sum_{i=1}^{n}(x_i - y_i)^2\right)}, x, y \in \mathbb{R}^n.$$

We give a short definition of Voronoi diagrams in accordance with [BCKO08, page 148-149]. A Voronoi diagram is defined by a set $C = \{c_1, c_2, \ldots, c_k\} \subseteq P$ of k distinct *Voronoi centers* in a plane $P \subseteq \mathbb{R}^2$. The *Voronoi Assignment Model* assigns every point p in the plane to its closest Voronoi center $\mathcal{V}(p) = c_i$ with $d(p, c_i) < d(p, c_j), \forall c_j \in C, j \neq i$.

This assignment divides the plane into k *Voronoi cells* $\mathcal{V}(c_i) = \{p \in P \mid \mathcal{V}(p) = c_i\}$, one for every center point $c_i \in C$. The cell $\mathcal{V}(c_i)$ contains all points p for which c_i is the closest center point. The *Voronoi diagram $Vor(C)$* is the partition of the plane P into the Voronoi cells $\mathcal{V}(c_i), c_i \in C$.

2.2.1. Graph Voronoi Diagrams

Graph Voronoi diagrams [Meh88, Erw00] are an extension of the Voronoi diagram to a graph $G = (V, E)$. The Voronoi centers of a Graph Voronoi diagram are a set $C = \{c_1, c_2, \ldots, c_k\} \subseteq V$ of k vertices [Erw00]. As for the original Voronoi diagram, the Voronoi Assignment Model assigns for every vertex $v \in V$ the closest Voronoi center $\mathcal{V}(v) = c_i$. Instead of the Euclidean distance, Graph Voronoi diagrams are based on the shortest path distance. The Voronoi cell of a Voronoi center c_i is the set of vertices $\mathcal{V}(c_i) = \{v \in V \mid \mathcal{V}(v) = c_i\}$ whose assigned center is c_i. Additionally, isolated vertices that are not reachable from any of the Voronoi centers must be taken into account. An additional set $U = \{v \in V \mid \mathcal{V}(v) = \bot\}$ is defined containing all those unreachable vertices [Erw00].

Together they define the Voronoi diagram $Vor(C)$, a partition $\{\mathcal{V}(c_1), \mathcal{V}(c_2), \ldots, \mathcal{V}(c_k), U\}$ of the graph G. In general, Graph Voronoi diagrams are not well-defined. That is, a vertex v might have multiple closest center points of whom only one is arbitrarily assigned as Voronoi center $\mathcal{V}(v)$ of v. For directed graphs, we need to differentiate between the *inward Voronoi diagram*, based on the shortest paths leading *towards* the Voronoi centers, and the *outward Voronoi diagram*, based on the paths leading *away from* the Voronoi centers.

For two Voronoi cells $\mathcal{V}(c_i)$ and $\mathcal{V}(c_j)$, a set of *Voronoi bridges* E_{ij} is defined containing all edges connecting a vertex in $\mathcal{V}(c_i)$ with a vertex in $\mathcal{V}(c_j)$. Two Voronoi cells $\mathcal{V}(c_i)$, $\mathcal{V}(c_j)$ are called *direct neighbors* if $E_{ij} \neq \emptyset$. The Voronoi diagram including the Voronoi bridges is called *extended Voronoi diagram.*
The Graph Voronoi diagram can be stored in $O(n)$ space. The extended Voronoi diagram may consume up to $O(n + m)$ space [Erw00].

A Graph Voronoi diagram can be computed with a multi-source shortest path search using the Voronoi centers C as sources [Meh88, Erw00]. The initial source $c \in C$ is passed along the shortest paths found by the algorithm and is assigned as Voronoi center $\mathcal{V}(v)$ of the vertices $v \in V$ on the path, just like setting the parent in the standard version of Dijkstra's algorithm. Using a multi-source version of Dijkstra's algorithm for weighted graphs and a multi-source BFS for unweighted graphs, the Graph Voronoi diagram can be computed in $O((n + m + (n - k)log(n - k))$ or $O(n + m)$ respectively.

2.3. Distance Oracles

The all-pairs shortest path problem is a well-known problem to compute the distance $\delta(v, w)$ between all pairs of vertices $v, w \in V$ in a graph $G = (V, E)$. However, in many

applications, not all of those distances are indeed needed. We only want to be able to retrieve the distance between some vertices quickly [TZ05].

A *distance oracle* solves this problem by running a preprocessing on the graph and storing results from this preprocessing in a data structure to speed up subsequent queries. A query efficiently computes the distance between any two vertices [TZ05], [PR10]. These queries are very fast, often in constant time, which is why they are called oracle [TZ05]. In general, there is a trade-off between the preprocessing time, the necessary space, the query time, and the approximation of a distance oracle.

A trivial example of a distance oracle would be to compute all-pairs shortest path in $O(n^2 \log n + nm)$ time, store the complete distance table in $O(n^2)$ space and answer queries in $O(1)$ by looking up the stored distance [TZ05]. However, computing all-pairs shortest path might take too much computation time or storing $O(n^2)$ data might be too much. Usually, we aim for sub-quadratic space complexity. This is, however, assumed to be unreachable for general graphs while guaranteeing exact results and constant query times [TZ05, PR10]. Therefore, many distance oracles guaranty only some approximation of the actual shortest path distance.

We differentiate between the *multiplicative* and *additive error* of a distance oracle. Let $\hat{\delta}(s,t)$ be the distance from s to t computed by the distance oracle. A distance oracle has a multiplicative error of $\alpha \geq 1$ and an additive error of $\beta \geq 0$ if

$$\forall s, t \in V : \delta(s,t) \leq \hat{\delta}(s,t) \leq \alpha \cdot \delta(s,t) + \beta.$$

If $\alpha = 1$ then there is no multiplicative error, likewise there is no additive error for $\beta = 0$. The additive error α is also called the *stretch* of the distance oracle. We define an (α, β) approximate distance oracle as a distance oracle with a multiplicative error of α and an additive error of β for every pair $s, t \in V$ of vertices.

An area closely related to distance oracles is *spanners* [TZ05, PR10]. An (α, β)-spanner of a graph G is a subgraph H of G such that for every distance $\delta_H(s,t)$ in H

$$\delta_G(s,t) \leq \delta_H(s,t) \leq \alpha \cdot \delta_G(s,t) + \beta$$

applies. The distance $\delta_G(s,t)$ is the corresponding distance in the original graph G to the distance $\delta_H(s,t)$ in H. A distance oracle with the multiplicative error α and the additive error β, capable of also returning the shortest *s-t*-path for the computed distance $\hat{\delta}(s,t)$, must contain an (α, β)-spanner [TZ05].

2.4. Transit Node Routing

Transit node routing [BFSS07, BFM$^+$07] is an algorithmic approach to solve shortest path queries. It was originally developed for road networks and utilizes their special characteristics, namely a low vertex degree and some sort of edge hierarchy. Most queries can be answered by transit node routing with only a few table lookups. This yields query times of $2.5 \, \mu s$, an improvement of over two orders of magnitude over the best previous results [BFSS07, ALS13].

Transit node routing is based on the intuition that when driving to a faraway location, there is a small set of traffic junctions of which at least one is passed if the distance is "large enough". These traffic junctions are called *transit nodes*, a subset $\mathcal{T} \subseteq V$ of the vertices. For every vertex $v \in V$, the transit nodes passed first on long-distance shortest-paths are called *access nodes*. The set $\mathcal{A}(v) \subseteq \mathcal{T}$ of access nodes is even smaller. On the US road network, a graph with 24 million vertices, [BFSS07] reports about 10 000 transit nodes and ten access nodes per vertex.

In general, transit node routing can be interpreted as a framework defining the basic structure of the algorithm while still leaving many design decisions up to the implementation. There are multiple different implementations [BFSS07, ALS13] of this general framework, most of which are tailored particularly to road networks. In the following, we define the general transit node framework. Chapter 3 defines our specific implementation of the transit node framework, resulting in an approximate algorithm for more general graphs, specifically social network graphs.

2.4.1. General Transit Node Framework

The transit node routing framework consists of six steps that need to be implemented and can be roughly grouped into a *preprocessing*, a *locality filter*, and the *queries*.

The preprocessing is computed once per instance before running the queries. During the preprocessing, a set $\mathcal{T}$ of *transit nodes* is selected with the intuition that all long shortest paths should go via at least one transit node. For every vertex $v \in V$, the preprocessing selects a set $\mathcal{A}(v)$ of *access nodes* and computes the distance $\mathtt{d}_{\mathcal{A}}(v, a) = \delta(v, a), a \in \mathcal{A}(v)$. For directed graphs, forward $\mathcal{A}_f(v)$ and backward $\mathcal{A}_b(v)$ access nodes are chosen. Forward access nodes are selected as the first transit node occurring on long-distance shortest paths starting at v. Correspondingly, backward access nodes occur last on such paths ending at v. In undirected graphs both sets are identical. Depending on the implementation, one could also select the access nodes first and set the transit nodes to the union of all access nodes $\mathcal{T} = \bigcup_{v \in V} \mathcal{A}(v)$. Additionally, a *distance table* $\mathtt{d}_{\mathcal{T}} : \mathcal{T}^2 \to \mathbb{R}_0^+$ storing the all-to-all distances between all transit nodes is computed.

The *locality filter* $\mathcal{L} : V^2 \to \{\mathtt{true}, \mathtt{false}\}$ is used during the queries to determine whether an s-t-query is global in the sense that the shortest s-t-path contains a transit node. Usually, the locality filter needs some precomputed data to make this decision which is computed during the preprocessing phase. Global queries are much faster than local queries. Therefore, we aim to maximize the percentage of global queries. However, computing a query with a global query algorithm might result in an incorrect distance if the shortest path does not contain a transit node. A good locality filter is conservative – meaning it always answers $\mathcal{L}(s, t) = \mathtt{true}$ if the global query is incorrect – but still answers $\mathcal{L}(s, t) = \mathtt{false}$ for most global queries.

A transit node query consists of two different approaches: a global and a local query. It uses the locality filter to decide which query to use. The *global query* computes the correct distance from s to t, $s, t \in V$ if a shortest s-t-path passes a transit node. The distance is computed by combining the precomputed data to the shortest path distance. For each pair $a_s \in \mathcal{A}_f(s)$ and $a_t \in \mathcal{A}_b(t)$, it computes the distance $\mathtt{d}_{\mathcal{A}}(s, a_s) + \mathtt{d}_{\mathcal{T}}(a_s, a_t) + \mathtt{d}_{\mathcal{A}}(a_t, t)$ and returns the minimum. All necessary distances have been precomputed by the preprocessing; thus the global query consists merely of a small number of table lookups. Actual shortest paths – instead of only the distances – can be reconstructed using multiple distance queries along the shortest path (see [BFM$^+$07, Section 3.6]).

The *local query* is a fall-back method used for those queries where the global query would give incorrect results. It should compute the correct distance for every pair for vertices but is usually much slower than the global query.

2.5. Covering Search

The preprocessing to our distance oracle is strongly based on the *covering searches* defined in [SS07, Section 2]. This section mostly gives a summary of Section 2 in [SS07] adjusted to our notation. At first, Section 2.5.1 states the problem which is then solved by different covering searches. After that, Section 2.5.2 to Section 2.5.4 describe three of the four covering searches defined in [SS07].

2.5.1. Problem Definition

Consider a single-source shortest path search from a source $s \in V$ using Dijkstra's algorithm (see Section 2.1.3). Let T be the shortest path tree with root s containing the shortest paths from s to all other vertices $v \in V$ visited so far. A set $V' \subseteq V$ of vertices is defined as capable of covering the shortest path search. The vertex $v \in V$ is *covered* by the vertex set V' if the shortest path from s to v contains at least one vertex in V'. A *covering node* is a vertex $c \in V'$ visited by the search whose parent $\mathtt{p}(c)$ is not covered. Thus, c is the first vertex in V' on its shortest path and also all shortest paths in the sub-tree of T with root c. It therefore covers all vertices it that sub-tree. The set of covering nodes is defined as $\mathcal{C}_G(V', s)$.

The objective is to determine such a set of covering nodes $\mathcal{C}_G(V', s)$, preferably while keeping the search space small and selecting no more covering nodes than necessary. Three different approaches to solve this problem are given in the following.

We additionally define the distance $\mathtt{d}_{\mathcal{C}}(v), v \in V$ computed during the covering search. Since all described covering searches are based on single-source shortest path algorithms, the distance $\mathtt{d}_{\mathcal{C}}(v)$ is computed correctly if the search is not truncated on the shortest s-v-path. [SS07]

2.5.2. Conservative Covering Search

The conservative approach is the most basic approach described in [SS07, Section 2]. It stops the search only when all vertices in the queue are covered. While this is easy to implement and yields correct distances, the search might get very large. If there is just a single long uncovered path, the search is continued for the complete length of that path even though all other branches of the shortest path tree have long been covered. This can cause the search to visit much more vertices than necessary. [SS07]

2.5.3. Aggressive Covering Search

The aggressive approach [SS07] truncates the search at the vertices in V'. This stops the search from visiting already covered vertices. The aggressive covering search is stopped when all queued vertices are covered, just like the conservative search.

However, the aggressive approach causes two other problems. First of all, computed paths are not guaranteed to be correct shortest paths, that is the computed distances might be larger than the actual shortest path distance. This is caused by the search detouring "around" covering nodes and visiting vertices that are technically covered, but unvisited since the search was stopped at the covering node. It can lead to an even bigger search space and incorrect distances. Furthermore, the aggressive approach might add more covering nodes than necessary. These superfluous covering nodes only cover vertices already covered by other transit nodes. [SS07]

2.5.4. Stall-on-Demand Covering Search

To solve the problems caused by the conservative and aggressive approaches, [SS07] introduces the Stall-on-Demand approach. "It is an extension of the aggressive variant" [SS07] preventing the search from traveling "around" covering nodes. To guarantee correctness the search sometimes needs to continue after a covering node was found. However, the search is unable to correctly decide in advance which edges, leaving covered vertices, need to be expanded. Therefore, the decision is postponed until it is both necessary and can be made correctly.

To manage this postponed decision-making, the search algorithm is extended as described in the following. The search is separated into a *main search* – the standard search from the source s – and a second BFS called multiple times at different points during the main search. We call this second search the *stalling-BFS*.

The main search is a variant of the aggressive approach. It starts from the source and is initially pruned when reaching a vertex $c \in V'$. However, if the main search continues around c and reaches it again via another path, c is "woken up", and a stalling-BFS from c is computed. It is stopped when all queued vertices are covered. The stalling-BFS from c considers only vertices already visited by the main search. Such a vertex w is inserted into the stalling-BFS queue if the path found by the main search is suboptimal. This is definitely true if the path via c to w yields a shorter distance. The stalling-BFS marks all vertices it visited as stalled. Stalled vertices prune the main search and are never selected as covering nodes. [SS07]

3. Approximate Transit Node Routing

The approximate transit node routing (aTNR) is an implementation of the transit node routing framework which is described in the previous Section 2.4. Unlike the previously mentioned TNR implementations [BFM$^+$07, ALS13], it is not specifically designed for road networks. It works particularly well for graphs with a small set of central vertices which cover many of the long-distance shortest paths, for example, social networks and web graphs. Furthermore, we add the option to approximate the computed distances to achieve a faster computation time. Point-to-point queries can be approximated with an additive error of $\beta = 4r$ where $r \in \mathbb{N}_0$ is a tuning parameter. This means that for every computed s-t-distance $\hat{\delta}(s,t)$ it applies that

$$\delta(s,t) \leq \hat{\delta}(s,t) \leq \delta(s,t) + 4r.$$

Thus, our distance oracle is a $(1, 4r)$-distance oracle as defined in Section 2.3. Section 6 describes this parameter r in more detail and explains what causes the approximation.

Just like the distance oracles described in Section 2.3, our algorithm consists of two phases. The preprocessing phase is computed once per instance and generates data that is then used to speed up the query phase. Computing the preprocessing takes more work than a naive query and becomes only worthwhile when multiple queries are computed on the same instance. Our preprocessing consists of three main components: selecting a set of transit nodes, computing the distance between all the transit nodes and selecting a small set of access nodes for every vertex. These access nodes are chosen from the previously selected transit nodes. Additionally, the distance between a vertex and its access nodes is stored.

An s-t-query can then be computed with the TNR global query defined in Section 2.4. The distance is computed as the shortest three-hop-distance $s \rightsquigarrow a_s \rightsquigarrow a_t \rightsquigarrow t$ where a_s is an access node of s and a_t an access node of t. Additionally, a local query is defined as a fall-back method for shortest paths not passing the access nodes of s and t. To identify those s-t-queries for which the global query would be incorrect, we use a heuristic locality filter. This locality filter is then combined with the global and local query to the actual query algorithm.

The remainder of this chapter describes the complete aTNR algorithm in detail. At first, the preprocessing is described in Section 3.1. After that, Section 3.2 gives an overview of the different query algorithms answering point-to-point queries. Section 3.3 defines the locality filter that determines which query algorithm to use. Finally, Section 3.4 proves the correctness of our algorithm and evaluates its time and space complexity.

3.1. Preprocessing

Our distance oracle algorithm consists of three main parts: the preprocessing, the locality filter and the queries. The preprocessing, which is the focus of this chapter, is only computed once for multiple queries on the same graph. During the query phase, the data precomputed by the preprocessing is used to speed up the queries.

Our preprocessing selects a set $\mathcal{T} \subseteq V$ of transit nodes and precomputes the distance $\mathtt{d}_{\mathcal{T}}(t_1, t_2)$ between every pair $t_1, t_2 \in \mathcal{T}$ of transit nodes. Furthermore, it selects a small number of access nodes $\mathcal{A}(v) \subseteq \mathcal{T}$ for every vertex $v \in V$ from the transit nodes $\mathcal{T}$. Together with the access nodes, it also calculates the distance $\mathtt{d}_{\mathcal{A}}(v, a)$ between v and its access nodes $a \in \mathcal{A}(v)$. We aim to cover all long-distance shortest paths from or to v with those access nodes.

The preprocessing offers essentially the following functionality: First, listing all access nodes $\mathcal{A}(v)$ of a vertex v and their precomputed distance $\mathtt{d}_{\mathcal{A}}(v, a)$ from v. Second, retrieving the distance $\mathtt{d}_{\mathcal{T}}(t_1, t_2)$ between any two transit nodes $t_1, t_2 \in \mathcal{T}$.
Apart from this, any precomputations necessary for the locality filter are also completed during the preprocessing phase. They are, however, described in Section 3.3 together with the other parts of the locality filter.

This chapter describes the necessary steps to compute a preprocessing for our distance oracle. Firstly, Section 3.1.1 describes different algorithms to select a set of transit nodes. Secondly, we give two algorithms to compute the transit node distance table in Section 3.1.2. Finally, Section 3.1.3 focuses on choosing the access nodes and computing their distance.

3.1.1. Transit Node Selection

The global query algorithm of the TNR framework computes the distance between two vertices as the length of the shortest path via their access nodes. Intuitively, the global query from s to t is correct if the shortest path contains at least one of the access nodes of both s and t. We try to select the transit nodes, i.e., the possible access nodes, so that they cover many shortest paths, and especially the long-distance shortest paths. A shortest path is covered by the transit node set if it contains at least one transit node. This is equivalent to the definition of covered paths in Section 2.5.

The number of transit nodes $k \in \mathbb{N}$ is a tuning parameter that has an impact on both computation time and required memory. In general, choosing $k \in O(\sqrt{n})$ keeps, for example, the size of the distance table linear. We therefore select $k = \lceil \sqrt{n} \rceil$ throughout this work.

The correctness of the queries does not depend on the chosen transit nodes. However, a good set of transit nodes speeds up both the preprocessing and the average query times. It reduces the required time to select the access nodes and the number of local queries that are necessary. Furthermore, the global query is faster if the average number of access nodes is small. If the transit nodes cover more shortest paths per transit node, fewer access nodes are necessary to cover the same number of shortest paths.

We implemented several algorithms heuristically selecting such a set of transit nodes. All of them are based on *betweenness centrality (BC)*, a measurement of vertex centrality. We first give a quick definition of betweenness centrality and then describe our different selection algorithms. An evaluation and comparison of the selection algorithms can be found in Section 6.2.1.

Betweenness Centrality

Betweenness centrality [Fre77] is a measurement for the centrality of a vertex in a graph based on the number of shortest paths containing it. The number of shortest paths from s

to t is denoted by σ_{st} with $\sigma_{ss} = 1$. For a vertex $v \in V$, the number of shortest s-t-paths containing v is defined as $\sigma_{st}(v)$. Then the betweenness centrality of a vertex v is defined as

$$BC(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

[Bra01]. This means the betweenness centrality of a vertex is the percentage of shortest paths containing v, compared to the total number of shortest paths in G, excluding the paths starting or ending at v. To compute the betweenness centrality, we use the implementation of Brandes' algorithm [Bra01] in the Ligra framework [SB13].

Simple-BC Transit Node Selection

The Simple-BC selection algorithm is based solely on betweenness centrality. It selects the k vertices with the highest betweenness centrality as transit nodes. While this algorithm is very simple (hence the name), it can result in transit nodes in very close proximity to each other. These transit nodes might all cover very similar sets of shortest paths. In that case, many of the transit nodes are superfluous because they do not cover any path that is not already covered by another transit node. Choosing better-distributed vertices with smaller betweenness centrality could result in more covered shortest paths.

Partition-BC Transit Node Selection

To increase the spread of the transit nodes in the graph, the Partition-BC algorithm uses a partition of the graph. It receives a balanced partition of the graph into k similar sized components. In each component, the vertex with the highest betweenness centrality is chosen as a transit node.

Farthest-BC Transit Node Selection

The Farthest-BC algorithm puts an even greater emphasis on the distance between transit nodes. The transit nodes are chosen consecutively starting with the highest betweenness centrality node. Each subsequent step determines a set of vertices with the maximum distance from the transit nodes chosen so far, using a multi-source BFS. From this set of vertices, the one with the highest betweenness centrality is chosen as the next transit node. These steps are repeated until k transit nodes are selected.

k-means++-BC Transit Node Selection

The k-means algorithm is "one of the most popular and simple clustering algorithms" [Jai10]. It computes a cluster of a set $\mathcal{X} \subset \mathbb{R}^d$ by minimizing the "squared error between the empirical mean of a cluster and the points in the cluster" [Jai10]. While the k-means algorithm is simple and fast, there are no guarantees for its accuracy [AV07].

K-means++ was initially developed by Arthur and Vassilvitskii [AV07] to improve the k-means algorithm by selecting better initial cluster centers. Instead of selecting the centers uniformly at random, they weight them such that data points with a higher distance to the current centers are favored. Defining $D(x)$ as the shortest distance between the data point $x \in \mathcal{X}$ and the closest center already chosen, they use the probability

$$\frac{D(x')^2}{\sum_{x \in \mathcal{X}} D(x)^2}$$

to select a new center point. The first center is selected uniformly at random from $\mathcal{X}$. All further center points are selected with the aforementioned probability until k centers are chosen.

We use a modified version of the k-means++ algorithm to select k transit nodes from the vertex set V. In addition to the distance $D(x)$, our version also considers the betweenness centrality of a vertex. To reflect the potentially high differences of the betweenness centrality and the squared distance, we normalize both to values between 0 and 1. We select x as a new transit node with the probability

$$\frac{1}{2} \cdot \left(\frac{BC(x')}{\sum_{x \in V} BC(x)} + \frac{D(x')^2}{\sum_{x \in V} D(x)^2} \right).$$

3.1.2. Distance Table Computation

To compute the global queries, TNR requires a distance table containing the shortest path distances $d_{\mathcal{T}}(t_1, t_2) = \delta(t_1, t_1)$ between all transit nodes $t_1, t_2 \in \mathcal{T}$. There are several approaches to computing this distance table; we describe two of them in the following.

The Naive Approach

The naive approach computes a single-source shortest-path search from every transit node until it reached all other transit nodes. An example of such a search is Dijkstra's algorithm which is described in Section 2.1.3. For unweighted graphs, a BFS is also sufficient (see Section 2.1.2). Since all the searches are independent of one another, they can be computed in parallel which makes this approach well parallelizable.

Lemma 3.1. *The distance table computed by the naive approach is correct.*

The correctness of Lemma 3.1 follows directly from the correctness of the single-source shortest-path algorithm used.

However, if the k searches have to visit a large proportion of the graph until all other transit nodes have been found, the naive approach can be rather slow.

The Overlay-Graph Approach

Another approach is to compute a so-called *overlay graph* $G_{\mathcal{T}}$ consisting only of the transit nodes and weighted edges connecting the transit nodes. The distance table can then be computed with an all-to-all shortest path algorithm on the overlay graph. This graph can be smaller than the original graph G which can result in a significant speed-up of the shortest path searches.

The overlay graph is defined as $G_{\mathcal{T}} = (\mathcal{T}, E_{\mathcal{T}})$ with $E_{\mathcal{T}} \subseteq \mathcal{T} \times \mathcal{T}$. It consists of all transit nodes connected by weighted edges. The weight $w(e)$ of an edge $e = (t_1, t_2)$ corresponds to the distance $w(e) = \delta(t_1, t_2)$ in the original graph G.
Intuitively, the neighbors of a transit node $t \in \mathcal{T}$ in $G_{\mathcal{T}}$ should be all transit nodes that are passed first on a shortest path from t to any other transit node. Based on this intuition, a covering search as defined in Section 2.5 is used to determine which transit nodes should be connected by an edge. A covering search with $V' = \mathcal{T} \setminus \{t\}$ is conducted from every transit node $t \in \mathcal{T}$. The covering nodes $\mathcal{C}_G(\mathcal{T} \setminus \{t\}, t)$ are the neighbors of t in the overlay graph $G_{\mathcal{T}}$. The edge weight is defined through the distance computed by the covering search, that is $w(t, t') = d_{\mathcal{C}}(t')$ for every $t' \in \mathcal{C}_G(\mathcal{T} \setminus \{t\}, t)$.

Lemma 3.2. *The overlay graph $G_{\mathcal{T}}$ preserves the distance between all transit nodes. That is, the distance $\delta(t_1, t_2)$ in G is equivalent to the distance $\delta_{G_{\mathcal{T}}}(t_1, t_2)$ in $G_{\mathcal{T}}$ for all $t_1, t_2 \in \mathcal{T}$.*

Proof. The edges in $G_{\mathcal{T}}$ correspond to paths in G thus all for distances applies $\delta(t_1, t_2) \leq \delta_{G_{\mathcal{T}}}(t_1, t_2)$. We therefore only need to prove $\delta_{G_{\mathcal{T}}}(t_1, t_2) \leq \delta(t_1, t_2)$.

For two arbitrary transit nodes $t_1, t_2 \in \mathcal{T}$ consider the shortest t_1-t_2-path $p = t_1 \rightsquigarrow t_2$ in G that contains the maximum number of transit nodes on any shortest t_1-t_2-path. Let $t_i' \in \mathcal{T}, i \in [1, k]$ be all transit nodes on the path p that is $p = t_1' \rightsquigarrow t_2' \rightsquigarrow \cdots \rightsquigarrow t_k'$ with $t_1 = t_1'$ and $t_2 = t_k'$.
According to Bellman's principle of optimality, the paths $p_i = t_i' \rightsquigarrow t_{i+1}', i \in [1, k-1]$ are shortest paths. Furthermore, there exists no shortest t_i'-t_{i+1}'-path containing a transit node other than t_i' and t_{i+1}' as that would increase the number of transit nodes on p.
Thus, the covering search from t_i cannot be truncated before visiting t_{i+1}, meaning t_{i+1} is set as covering node and the distance $\mathtt{d}_{\mathcal{C}}(t_{i+1})$ is correct. Therefore, the edge $e_i = (t_i, t_{i+1})$ is added to the overlay graph with the correct distance $w(e_i) = \mathtt{d}_{\mathcal{C}}(t_i, t_{i+1}) = \delta(t_i, t_{i+1})$. This means that there is a path $p' = \langle t_1', t_2', \ldots, t_k' \rangle$ in $G_{\mathcal{T}}$ with weight

$$ w_{G_{\mathcal{T}}}(p') = \sum_{i=1}^{k-1} w_{G_{\mathcal{T}}}(t_i', t_{i+1}') = \sum_{i=1}^{k-1} \delta(t_i', t_{i+1}') = \sum_{i=1}^{k-1} w(p_i) = w(p). $$

It follows that $\delta_{G_{\mathcal{T}}}(t_1, t_2) \leq w_{G_{\mathcal{T}}}(p') = \delta(t_1, t_2)$. $\qquad\square$

The distance table can then be computed with an all-to-all shortest path algorithm on the overlay graph $G_{\mathcal{T}}$. Some examples of such algorithms are the Floyd–Warshall algorithm [Flo62, War62] which takes $O(n^3)$ asymptotic time and Johnson's algorithm [Joh77] which runs in $O(n^2 \log n + nm)$. For graphs with non-negative weights, it is sufficient to use Dijkstra's algorithms once from every source which has the same asymptotic running time as Johnson's algorithm. However, Johnson's algorithm conducts a preprocessing for negative weights, which is not necessary in our case. We therefore use Dijkstra's algorithm once from every vertex in $G_{\mathcal{T}}$ to compute the actual distances between all transit nodes. Since $G_{\mathcal{T}}$ can be smaller than the original graph G, this can be faster than computing k single-source shortest path-searches.

Lemma 3.3. *The distance table computed by the overlay-graph approach is correct.*

This follows directly from Lemma 3.2 and the correctness of the all-to-all shortest path algorithm.

The overlay-graph approach is also well parallelizable. Both phases consist of k independent searches from the k transit nodes that can be computed in parallel. Adding edges in parallel to the overlay graph might cause synchronization problems depending on the graph data structure used. This can be solved by first computing all searches and storing the edges separately and then sequentially combining them to the graph. In the second phase, each line in the distance table is accessed by exactly one Dijkstra search. Thus, no synchronization is necessary.

3.1.3. Access Node Determination

This section deals with selecting the access nodes of every vertex and computing the distance to them. The access nodes are chosen from the set of transit nodes selected in Section 3.1.1. The global query computes distances as shortest paths via an access node of the source and the target. Because of this, the access nodes should cover most shortest paths leaving a vertex. A shortest path is covered by a set of access nodes if it contains at least one of them. The locality filter should be able to recognize shortest paths that are not covered by an access node so an alternative query algorithm can be chosen.

As defined in Section 2.4, we use $\mathcal{A}(v) \subseteq \mathcal{T}$ to denote the set of access nodes of a vertex $v \in V$. Furthermore, $\mathsf{d}_{\mathcal{A}}(v, a)$ is the computed distance from a vertex v to its access node $a \in \mathcal{A}(v)$. For directed graphs, we differentiate between the forward access nodes $\mathcal{A}_f(v)$ used for paths starting at v and the backward access nodes $\mathcal{A}_b(v)$ used for paths ending at v. For simplicity, this section generally describes the algorithms only for undirected graphs. Computing the forward access nodes in a directed graph can be directly transferred from the undirected version. The backward access nodes are selected by conducting the algorithms on the reversed graph. If the undirected algorithm cannot be easily transferred to directed graphs, an additional variant for directed graphs is given.

For a transit node $t \in \mathcal{T}$, all shortest paths leaving t contain t. Therefore, t itself is chosen as its only access node, that is $\forall t \in \mathcal{T} : \mathcal{A}(t) = \{t\}$ and $\mathsf{d}_{\mathcal{A}}(t, t) = 0$. For all other vertices $v \in V$, a covering search – as defined in Section 2.5 – with source v is computed to determine the set of access nodes $\mathcal{A}(v)$.

Section 3.1.3 describes in more detail how the covering searches are used to select the access nodes. After that, Section 6 introduces marked nodes, an extension of the transit nodes made to speed up the computation at the price of approximation. Finally, Section 9 gives an algorithm to remove superfluous access nodes.

Covering Search

To determine the access nodes of all vertices $v \in V$ a covering search from every $v \in V \setminus \mathcal{T}$ with $V' = \mathcal{T}$ is conducted. The resulting covering nodes $\mathcal{C}_G(\mathcal{T}, v)$ are selected as the access nodes $\mathcal{A}(v)$ with the distance $\mathsf{d}_{\mathcal{A}}(a_v) = \mathsf{d}_{\mathcal{C}}(a_v)$, $a_v \in \mathcal{A}(v)$ computed by the covering search. All three covering search approaches introduced in Section 2.5 can be used for our preprocessing and result in a correct distance oracle as proven in Section 3.4.2. Algorithm 3.1 describes the general access node determination.
The shortest paths that are not covered by the access nodes are traversed completely. This allows a straightforward implementation of the locality filter based on the search spaces. Section 3.3 describes this locality filter in detail.

Algorithm 3.1: Access Node Determination with Generic Covering Search

---

1 $V' \leftarrow \mathcal{T}$
2 foreach $v \in V \setminus V'$ **do**
3 $\quad$ $(\mathcal{C}_G(V', v), \mathsf{d}_{\mathcal{C}}) \leftarrow \texttt{CoveringSearch}(v, V')$
4 $\quad$ $\mathcal{A}(v) \leftarrow \mathcal{C}_G(V', v)$
5 $\quad$ **foreach** $a_v \in \mathcal{A}(v)$ **do**
6 $\quad\quad$ $\mathsf{d}_{\mathcal{A}}(a_v) \leftarrow \mathsf{d}_{\mathcal{C}}(a_v)$

Marked Nodes

We extend the transit nodes to their surrounding nodes within a certain *marking radius r*. The *marked nodes* $\mathcal{M}$ are all vertices within this radius and are used as subsidiary transit nodes. By definition, the transit nodes are a subset of the marked nodes $\mathcal{T} \subseteq \mathcal{M}$. In the covering search, the marked nodes are treated as transit nodes meaning they cover the search, that is $V' = \mathcal{M}$. This speeds up the access node determination and is what causes the approximation in our algorithm.

We define the marked nodes as

$$\mathcal{M} = \{m \in V \mid \exists t_m \in \mathcal{T} : \ \delta(m, t_m) \leq r \text{ and } \delta(t_m, m) \leq r\}.$$

That is, for every marked node there is at least one transit node t_m whose forward and backward distance from m is at most the marking radius r. For each marked node $m \in \mathcal{M}$,

this closest transit node t_m is selected as the access node of m, that is $\mathcal{A}(m) = \mathcal{A}_f(m) = \mathcal{A}_b(m) = \{t_m\}$. The access node distances are set to the distances $\mathtt{d}_\mathcal{A}(m, t_m) = \delta(m, t_m)$ and $\mathtt{d}_\mathcal{A}(t_m, m) = \delta(t_m, m)$. By definition, $\mathtt{d}_\mathcal{A}(m, t_m) \le r$ and $\mathtt{d}_\mathcal{A}(t_m, m) \le r$ apply. Ties between multiple closest transit nodes with the same distance are broken arbitrarily. Note how transit nodes are still selected as their only access node with distance 0.

Computing the Marked Nodes

For undirected graphs, the marked nodes are selected using a multi-source shortest-path search. This search starts from the transit nodes and is truncated after distance r. All vertices visited before the search is stopped are marked and assigned to the transit node from which they were visited first. The distance between a marked node and its transit node can also be obtained directly from the search.

For directed graphs, the computation is slightly more complicated as we need to make sure both the forward and the backward distance does not exceed the radius r. We describe an algorithm determining the marked nodes in directed graphs in the following. However, since our implementation of the distance oracle only handles undirected graphs, we did not use this algorithm in practice.

The forward distances $\delta(t_m, m)$ from a transit node $t_m \in \mathcal{T}$ to a vertex $m \in V$ are computed with a single-source shortest path search from t_m stopped after distance r. The backward distances $\delta(m, t_m)$ are calculated using the same algorithm on the reverse graph.
It is not sufficient to intersect the vertices visited by the forward and the backward search because this does not ensure that every marked vertex is marked by the same transit node in both directions. After the forward search we therefore store every potential marked node m with $\delta(t_m, m) \le r$ at the corresponding transit node t_m. Then, the backward search checks whether these potential marked nodes are still within the marking radius, that is $\delta(m, t_m) \le r$. If this applies, they are marked, and t_m is assigned as their access node with the distances $\mathtt{d}_\mathcal{A}(t_m, m) = \delta(t_m, m)$ and $\mathtt{d}_\mathcal{A}(m, t_m) = \delta(m, t_m)$. If there are multiple such transit nodes, it is sufficient to choose only one of them as access node, for example, the closest transit node.

Using the Marked Nodes in the Covering Search

In the covering search which determines the access nodes of a vertex $v \in V$, the marked nodes are treated as transit nodes. Thus, we set $V' = \mathcal{M}$ instead of $V' = \mathcal{T}$. The only access node of a marked node is the transit node it is assigned to. It is therefore not necessary to compute a covering search from the marked nodes. For all other nodes $v \in V$, a covering search is conducted and produces a set $\mathcal{C}_G(\mathcal{M}, v) \subseteq \mathcal{M}$ of covering nodes. We only want to set transit nodes as access nodes and therefore set the access nodes of the covering nodes as access nodes of v, that is

$$\mathcal{A}(v) = \bigcup_{c \in \mathcal{C}_G(\mathcal{M}, v)} \mathcal{A}(c).$$

The distance $\mathtt{d}_\mathcal{A}(v, a)$ of such an access node $a \in \mathcal{A}(v)$ is computed as the sum

$$\mathtt{d}_\mathcal{A}(v, a) = \mathtt{d}_\mathcal{C}(c) + \mathtt{d}_\mathcal{A}(c, a),$$

where $c \in \mathcal{C}_G(\mathcal{M}, v)$ is the covering node through which a was added. If an access node is added multiple times from different covering nodes, the minimum distance is kept. Algorithm 3.2 describes the access node determination with marked nodes in pseudocode.

Algorithm 3.2: Access Node Determination with Marked Nodes $\mathcal{M}$

---

1 $V' \leftarrow \mathcal{M}$
2 **foreach** $v \in V \setminus V'$ **do**
3 $\mathcal{A}(v) \leftarrow \emptyset$
4 $\mathtt{d}_\mathcal{A} \leftarrow \infty$
5 $(\mathcal{C}_G(V', v), \mathtt{d}_\mathcal{C}) \leftarrow \mathtt{CoveringSearch}(v, V')$
6   **foreach** $c \in \mathcal{C}_G(V', v)$ **do**
7 $\mathcal{A}(v) \leftarrow \mathcal{A}(v) \cup \mathcal{A}(c)$
8   **foreach** $c \in \mathcal{A}(c)$ **do**
9 $\mathtt{d}_\mathcal{A}(a_v) \leftarrow \mathbf{min}(\mathtt{d}_\mathcal{A}(a_v), \mathtt{d}_\mathcal{C}(c) + \mathtt{d}_\mathcal{A}(c, a))$

Lemma 3.4. *Let $v \in V$ be a vertex, $m \in \mathcal{M}$ a marked node and $a \in \mathcal{A}(m)$ an access node of m. If m is the first marked node on all shortest v-m-paths, then a is an access node of v with distance*

$$\mathtt{d}_\mathcal{A}(v, a) \leq \delta(v, m) + r.$$

Proof. By definition, there is no v-m-path containing any marked node other than m. Thus, the covering search is not truncated before visiting m. As a result, m is a covering node, that is $m \in \mathcal{C}_G(\mathcal{M}, v)$, and its distance $\mathtt{d}_\mathcal{C}(m) = \delta(v, m)$ is computed correctly.

Therefore,

$$\mathcal{A}(v) = \bigcup_{c \in \mathcal{C}_G(\mathcal{M}, v)} \mathcal{A}(c) \supseteq \mathcal{A}(m) \supseteq \{a\}$$

applies, meaning a is set as access node of m. Furthermore,

$$\mathtt{d}_\mathcal{A}(v, a) \leq \mathtt{d}_\mathcal{C}(m) + \mathtt{d}_\mathcal{A}(m, a) \leq \delta(v, m) + r$$

applies for the computed distance. $\qquad\square$

The marked nodes speed up the access node determination for two reasons. First of all, they reduce the number of necessary covering searches. Selecting the marked nodes is in general much faster than computing a covering search for all of them. Furthermore, the marked nodes are assigned only a single access node. This speeds up the global query whose running time depends mostly on the number of access nodes.
Secondly, the covering searches for unmarked nodes are accelerated because the number of vertices covering the search is increased. This allows the covering search to stop earlier.

However, the marked nodes also introduce an *approximation* to our distance oracle. Since the covering search from s is already pruned at marked nodes, it is possible that some shortest paths are only covered by a marked node m but not by an actual transit node.
The global query, however, considers only paths passing directly through access nodes of s. The closest access node a of the start s has a distance of at most r from m. When forced to pass a, the shortest path is in the worst case prolonged by $2r$. The same situation can arise at the target t causing another error of $2r$. In total, an additive error of up to $\beta = 4r$ can occur.

A schematic illustration of this worst case can be found in Figure 3.1. The transit nodes are $\mathcal{T} = \{T_1, T_2\}$ which mark the vertices $\mathcal{M} = \{a_s, a_t\}$. If the distance between the transit

and the marked nodes were any larger than r, a_s and a_t would not be marked. The access nodes of s and t are $\mathcal{A}(s) = \{T_1\}$ and $\mathcal{A}(t) = \{T_2\}$.

The actual shortest s-t-path is $p^* = \langle s, a_s, a_t, t \rangle$ with length $\delta(s, t) = w(p) = \mathrm{w}_1 + \mathrm{w}_2 + \mathrm{w}_3$. It is represented by a dotted line in Figure 3.1. The shortest path does contain a marked node, and the query is therefore classified as global. However, the shortest path p' via the access nodes of s and t is $p' = \langle s, a_s, T_1, a_s, a_t, T_2, a_t, t \rangle$. This path has a length of $w(p') = \mathrm{w}_1 + 2r + \mathrm{w}_2 + 2r + \mathrm{w}_3 = \delta(s, t) + 4r$ and is represented by a dashed line.

The distance computed by the TNR global query is the length of p'. Thus, the global distance has a maximum additive error of $4r$. Section 3.4 gives a formal proof that this additive error is never exceeded.
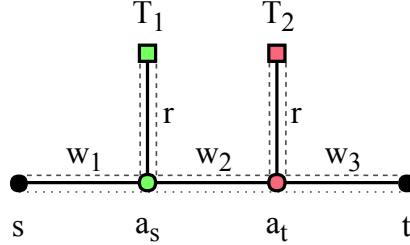
Figure 3.1.: The actual shortest s-t-path p^* (dotted line) and the shortest s-t-path p' via the access nodes T_1 and T_2 of s and t (dashed line).

In other words, while the marked nodes accelerate the preprocessing, they also introduce an additive error. This causes a trade-off between preprocessing time and accuracy: The larger the marking radius r, the faster the preprocessing, but the more inaccurate the queries. Therefore, one should carefully choose the correct marking radius.

Postprocessing to Remove Superfluous Access Nodes

As already mentioned, the preprocessing might add access nodes that are not actually necessary. Since we aim for a small number of access nodes, we try to find and remove these superfluous access nodes. This reduces both the space required to store the preprocessing and the running time of the global query.

An access node $a \in \mathcal{A}(v)$ of a vertex $v \in V$ is superfluous if it can be "replaced" by another access node $b \in \mathcal{A}(v)$ in the sense that the distance $\mathsf{d}_{\mathcal{A}}(v, a)$ can be computed as $\mathsf{d}_{\mathcal{A}}(v, b) + \mathsf{d}_{\mathcal{T}}(b, a)$. We therefore check if

$$\mathsf{d}_{\mathcal{A}}(v, a) \geq \min_{b \in \mathcal{A}(v), b \neq a} \mathsf{d}_{\mathcal{A}}(v, b) + \mathsf{d}_{\mathcal{T}}(b, a)$$

applies. If it does, a is removed from the access nodes $\mathcal{A}(v)$. This is repeated for all access nodes $a \in \mathcal{A}(v)$ of a vertex v and for all vertices $v \in V$ with more than one access node. It is important to make sure that access nodes cannot prune each other such that both are removed. We ensure this by processing the access nodes in $\mathcal{A}(v)$ sequentially and updating the access node set after every step. The access nodes $\mathcal{A}(v)$ of different vertices $v \in V$ are independent of each other and thus can be post-processed in parallel.

3.2. Queries

This section defines the point-to-point queries used to retrieve actual distances from the preprocessed data. Every query algorithm discussed in this section takes two vertices s and t and computes a distance $\hat{\delta}(s, t)$. Ideally, this distance is within the allowed additive error $\beta = 4r$, that is

$$\forall s, t \in V : \delta(s, t) \leq \hat{\delta}(s, t) \leq \delta(s, t) + 4r.$$

At first, the global and the local query are described in Section 3.2.1 and 3.2.2. After that, Section 3.2.3 combines them with the locality filter to the actual query algorithms called the *combined queries*. As already mentioned, the intention of this combination is to use the global query whenever possible while answering all queries correctly.

3.2.1. Global Query

The global query in our distance oracle is the same as in the original transit node routing (see Section 2.4.1). To answer an *s-t*-query, the global query computes the global distance

$$\hat{\delta}_g(s,t) = \min_{a_s \in \mathcal{A}_f(s), a_t \in \mathcal{A}_b(t)} \mathtt{d}_{\mathcal{A}}(s, a_s) + \mathtt{d}_{\mathcal{T}}(a_s, a_t) + \mathtt{d}_{\mathcal{A}}(a_t, t).$$

However, even if s and t are correctly classified as global, the global distance is only a $(1, 4r)$-approximation of the actual distance. This is due to the approximation introduced in Section 3.1.3 and is proven in Lemma 3.6.

3.2.2. Local Query

The local query is used as a fallback method if the global query gives a wrong result. Because of this, it has to compute a correct (approximate) distance. We define $\hat{\delta}_l(s,t)$ as the local distance from s to t computed by the local query. In general, any query algorithm resulting in a correct distance can be used as local query. For example, the original transit node routing uses a multi-level approach for the local query with multiple levels of transit nodes [BFM$^+$07]. Usually, the local query is much slower than the global query. This applies especially for long distances. Transit node routing is therefore the most useful if the vast majority of queries is global and can be recognized as such. To decide when to use the local query, Section 3.3 defines a locality filter.

Our Approximate TNR distance oracle uses a bidirectional shortest path search as local query algorithm. For weighted graphs, we use a bidirectional version of Dijkstra's algorithm, for unweighted a bidirectional BFS. A bidirectional shortest path search starts two unidirectional searches simultaneously from s and t. They are alternated until the search spaces meet. Bidirectional approaches generally reduce the search space compared to a unidirectional approach and therefore speed up the query [MS08, page 209-210].

Lemma 3.5. *The bidirectional version of Dijkstra's algorithm and BFS are correct, that is they compute the correct distance $\delta(s,t)$ for every $s, t \in V$.*

An explicit description of these bidirectional algorithms and a proof of Lemma 3.5 can, for example, be found in [MS08, page 209-210].

Local Query with Limited Distance

Furthermore, variations of the local search can be created by using additional stop-criteria. We define a *local query with limited distance* that is stopped when visiting a certain distance. The local query might not find an *s-t*-path even though such a path exists. It computes the correct distance to all vertices visited before it is truncated, but there is no guarantee on the distance to unvisited vertices. The distance at which the search is truncated can be either constant for all queries or be chosen dynamically for each query.

3.2.3. Combined Query

Transit node routing combines the global and local query with the locality filter to one combined query algorithm. It takes the advantages of both the global and the local query and uses them to compensate for the other's disadvantages. This yields a query algorithm that is faster than the local query while still computing (approximately) correct distances. We use $\hat{\delta}_c(s,t)$ to denote the combined distance returned by the combined query. In the following, we describe three different variants of the combined query.

TNR Combined Query

The combined query in the original transit node routing [BFSS07, BFM$^+$07] uses the locality filter to decide which query to use and then runs the global or the local query accordingly. If the locality filter classifies the query as global, the global query always returns a correct result as proven in Section 3.4. Otherwise, the global query might be incorrect and the local query, which is always correct, is used as fallback. The global query is given in pseudocode by Algorithm 3.3.

Algorithm 3.3: TNR Combined Query

---
1 if $\mathcal{L}(s,t)$ **then**
2 | **return** localQuery(s,t)
3 else
4 | **return** globalQuery(s,t)

Exact Combined Query with Limited Local Query

The locality filter is not perfect and might decide to use a local query even though the global query would also yield an approximately correct distance. For the correctness, it suffices to ensure that the global distance is not larger than $\delta(s,t)+4r$. The *exact combined query with limited local query* is based on this idea and truncates the local query at the distance $\hat{\delta}_g(s,t) - 4r$. At first, the global distance $\hat{\delta}_g(s,t)$ is computed. After that, the locality filter decides whether a local query is necessary. If it is necessary, such a limited local query is conducted. The local distance is returned if the limited local query does find an s-t-path, that is $\hat{\delta}_l(s,t) < \hat{\delta}_g(s,t) - 4r$. Otherwise, the previously computed global distance $\hat{\delta}_g(s,t)$ is returned. In total,

$$\hat{\delta}_c(s,t) = \begin{cases} \hat{\delta}_l(s,t) & \text{if } \mathcal{L}(s,t) = \texttt{true} \text{ and } \hat{\delta}_l(s,t) < \hat{\delta}_g(s,t) - 4r \\ \hat{\delta}_g(s,t) & \text{otherwise} \end{cases}$$

applies. Algorithm 3.4 describes the exact combined query with limited local query in pseudocode.

Algorithm 3.4: Exact Combined Query

---
1 $\hat{\delta}_g(s,t) \leftarrow$ globalQuery(s,t)
2 if $\mathcal{L}(s,t)$ **then**
3 | limit $\leftarrow \hat{\delta}_g(s,t) - 4r$
4 | $\hat{\delta}_l(s,t) \leftarrow$ localQuery$(s,t,\texttt{limit})$
5 | **if** $\hat{\delta}_l(s,t) <$ limit **then**
6 | | **return** $\hat{\delta}_l(s,t)$
7 | **else**
8 | | **return** $\hat{\delta}_g(s,t)$
9 else
10 | **return** $\hat{\delta}_g(s,t)$

Approximate Combined Query

Additionally, we define an *approximate combined query* where the local query might be truncated even though the global query is not necessarily correct. As in the previously defined query, a global query is conducted for all queries followed by a local query if the locality filter classifies the query as `local`. If the local query does not find an *s-t*-path, the global distance is returned. Contrary to the exact combined query, the approximate combined query might truncate the local query even earlier than at $\hat{\delta}_g(s,t) - 4r$. To do so, we introduce an additional parameter called the local query `limit` $\in \mathbb{R}_0^+$. The local query is then truncated at the distance $\mathbf{min}(\hat{\delta}_g(s,t) - 4r, \texttt{limit})$. The pseudocode of the approximate combined query is given in Algorithm 3.5.

Algorithm 3.5: Approximate Combined Query

---

1 $\hat{\delta}_g(s,t) \leftarrow \texttt{globalQuery}(s,t)$
2 **if** $\mathcal{L}(s,t)$ **then**
3 $\texttt{limit} \leftarrow \mathbf{min}(\hat{\delta}_g(s,t) - 4r, \texttt{limit})$
4 $\hat{\delta}_l(s,t) \leftarrow \texttt{localQuery}(s,t,\texttt{limit})$
5   **if** $\hat{\delta}_l(s,t) < \texttt{limit}$ **then**
6   **return** $\hat{\delta}_l(s,t)$
7   **else**
8   **return** $\hat{\delta}_g(s,t)$
9 **else**
10   **return** $\hat{\delta}_g(s,t)$

3.3. Locality Filter

A locality filter $\mathcal{L} : V^2 \rightarrow \{\texttt{true}, \texttt{false}\}$, as defined in Section 2.4.1, is a heuristic to decide whether the distance between two vertices $s, t \in V$ is best computed with a local or a global query. If $\mathcal{L}(s,t) = \texttt{false}$ the query from s to t is regarded as `global`. It is correspondingly regarded as `local` if $\mathcal{L}(s,t) = \texttt{true}$. The locality filter should be conservative, that is it should only answer `global` if the global distance $\hat{\delta}_g(s,t)$ is correct. However, it should still classify queries as `global` as often as possible since the global query is much faster than the local query. A global distance $\hat{\delta}_g(s,t)$ is regarded as correct if it is within the allowed $(1, 4r)$-approximation. Intuitively, this should be the case if the shortest *s-t*-path contains any marked nodes. Section 3.4 proves that this intuition applies.

3.3.1. Graph Voronoi Filter

Our distance oracle uses a locality filter based on the *Search Space Based Locality Filter* given by [ALS13]. The transit node routing in [ALS13] is based on *contraction hierarchies* [GSSD08, GSSV12], a speedup-technique for shortest path queries in road networks. The Search Space Based Locality Filter considers the search spaces of a contraction hierarchy query from *s* to *t* before reaching the transit nodes. If they are disjoint, the shortest path must contain a transit node. It is therefore save to set $\mathcal{L}(s,t) = \texttt{false}$. In return, if the search spaces intersect, there might be a shortest path not containing a transit node, so the locality filter is set to $\mathcal{L}(s,t) = \texttt{true}$.
The search space is compressed to reduce the required space and speed up the locality filter. Adding more vertices to the search space does not change the correctness of the locality filter. As a result, the search spaces can be approximated by the visited Graph Voronoi cells with the transit nodes as Voronoi centers. This Search Space Based Locality Filter with compressed search spaces is called *Graph Voronoi Filter* [ALS13].

Our algorithm basically uses the same approach. Since it is not based on contraction hierarchies, the search spaces of the covering searches are used instead of contraction hierarchy queries.

To precompute the data necessary for our Graph Voronoi Filter, we first compute the Voronoi diagram using the transit nodes as Voronoi centers. The search spaces of the covering searches are a by-product of the preprocessing. They can be easily transformed into their approximation by looking up the Voronoi cell of each visited vertex. Hence, there is not much overhead caused by computing the approximated search spaces. As in the original Graph Voronoi Filter, we exclude transit nodes from our search spaces. We additionally exclude marked nodes which are an extension of our transit nodes.

Queries from or to transit and marked nodes are always `global`. We therefore store an empty search space for those vertices.

To answer whether a query is `local`, that is, $\mathcal{L}(s, t) = \texttt{true}$, the locality filter only needs to intersect the approximated search spaces. If the search spaces are disjoint, the global query is correct as shown in Section 3.4. Otherwise, the global query might be incorrect, and a local query should be used. Formally, the locality filter can be defined as

$$\mathcal{L}(s, t) = \begin{cases} \texttt{false} & \text{if } \mathcal{S}(s) \cap \mathcal{S}(t) = \emptyset \\ \texttt{true} & \text{else} \end{cases}$$

where $\mathcal{S}(v), v \in V$ is the (approximated) search space of v.

On directed graphs, there is a difference between the inward and the outward Voronoi diagram. Since the Graph Voronoi Filter uses the Voronoi diagram only to compress the search space, it is not too important which one of them is chosen. It is, however, important that the same diagram is used for both the forward and the backward covering search space to allow intersecting them.

3.4. Complexity and Correctness

3.4.1. Time and Space Complexity

This section gives a brief overview of the time and space complexity of aTNR. In general, the experimental time and space are much better than the worst-case considered here.

Time Complexity of the Preprocessing

The time complexity T_{Prep} of the preprocessing consists of the complexity of the individual preprocessing steps:

$$T_{\text{Prep}} = T_{\text{TransitNodes}} + T_{\text{DistanceTable}} + T_{\text{AccessNodes}}.$$

The following paragraphs consider all three steps separately.

We use T_{SSSP} to denote the complexity of a single-source shortest path search. On weighted graphs, this is $T_{\text{SSSP}} = O(n \log n + m)$ using Dijkstra's algorithm. On unweighted graphs, this can be improved to $T_{\text{SSSP}} = O(n + m)$ using a BFS. Note that the multi-source versions of both searches are asymptotically not slower than single source version, thus $T_{\text{MSSP}} \subseteq T_{\text{SSSP}}$.

In general, we cannot give a theoretical bound on the number of vertices visited by a covering search. They thus have the same worst-case complexity as a simple single-source shortest path search. In practice, however, the search space is often much smaller which is utilized by our distance oracle.

We assume the betweenness centrality and partition to be already precomputed and, thus, neglect their computation time in $T_{\text{TransitNodes}}$. Finding the top k vertices with

maximum betweenness centrality takes $O(n \log k)$ time with a minimum priority queue storing the best vertices found so far. Thus, for the Simple-BC transit node selection $T_{\text{TransitNodes}} = O(n \log k)$ applies. The vertex with maximum betweenness centrality in each component of the partition can be determined by iterating all vertices in the component once. This yields a complexity of $T_{\text{TransitNodes}} = O(n)$ for the Partition-BC transit node selection. Both the Farthest-BC and the k-Means++-BC transit node selection have to compute the distance between the already selected transit nodes and all other vertices in each of the k steps. This takes T_{MSSP} time each step and dominates the running time of the transit node selection. In total, this yields a running time of $T_{\text{TransitNodes}} = k \cdot T_{\text{MSSP}}$.

Using the naive approach, the distance table can be computed in $T_{\text{DistanceTable}} = k \cdot T_{\text{SSSP}}$. The overlay-graph approach yields the same complexity as the all-pairs shortest path search on the overlay graph is dominated by the k covering searches necessary to compute the overlay graph.

Computing the access nodes is broken down into three steps: selecting the marked nodes, computing the access nodes of all other (unmarked) nodes, and removing superfluous access nodes in the postprocessing. Thus, complexity is defined as the sum

$$T_{\text{AccessNodes}} = T_{\text{Marked}} + T_{\text{Unmarked}} + T_{\text{Post}}.$$

The marked nodes are computed with one or two multi-source searches thus $T_{\text{Marked}} = T_{\text{MSSP}}$ applies. To determine the access nodes of the unmarked nodes, a covering search is computed from every unmarked node. This has a worst-case complexity of $T_{\text{Unmarked}} = n \cdot T_{\text{SSSP}}$. Every vertex v has at most $|\mathcal{A}(v)| \leq k$ access nodes. The postprocessing tests for each of the access nodes if they can be reached via any of the other access nodes. This takes $O(|\mathcal{A}(v)|^2) \subseteq O(n)$ time per vertex and $T_{\text{Post}} = O(n^2)$ time in total. Collectively, computing the access nodes has a complexity of

$$T_{\text{AccessNodes}} = T_{\text{MSSP}} + n \cdot T_{\text{SSSP}} + O(n^2) = n \cdot T_{\text{SSSP}}.$$

The access node determination also dominates the total preprocessing time of

$$T_{\text{Prep}} \leq k \cdot T_{\text{MSSP}} + k \cdot T_{\text{SSSP}} + n \cdot T_{\text{SSSP}} = n \cdot T_{\text{SSSP}}.$$

The precomputations for the locality filter only need to additionally compute a Voronoi diagram which takes T_{MSSP} time. This obviously does not affect the complexity of the preprocessing.

Time Complexity of the Queries

The global query needs to check all distance combinations using the access nodes of s and t. This takes $O(|\mathcal{A}(s)| \cdot |\mathcal{A}(t)|)$ time. The maximum number of potential access nodes is k resulting in a theoretical worst-case time of $O(k^2) = O(n)$. However, the average number of access nodes in practice is much smaller than k, resulting in much faster global queries. In the worst case, the local query has to search the whole graph, resulting in a complexity of T_{SSSP}. The locality filter intersects the visited Voronoi cells of s and t. Since there are only k Voronoi cells, this has a complexity of $O(k)$ but is a lot faster if s and t did not visit all cells. Among these three complexities, the local query dominates the other two. Thus, the combined query, which is a combination of the global query, the local query, and the locality filter, has a worst-case complexity of $O(T_{SSSP})$.

Space Complexity

To compute the global query, the distance oracle needs to store the distance table together with the access nodes of each vertex and the corresponding distances. The distance table has a space complexity of $O(k^2) = O(n)$. Every vertex has at most k access nodes thus all access nodes can be stored in $O(n \cdot k)$ space. Furthermore, the locality filter needs to store the visited Voronoi cells for all vertices resulting in an additional $O(n \cdot k)$ space. Therefore, storing the preprocessed data has a worst-case complexity of $O(n \cdot k)$. However, in practice, both the average access nodes and the average visited Voronoi cells are generally much less than k.

3.4.2. Correctness

In this section, we prove that the distances computed by our distance oracle are correct. As already mentioned, we regard a computed distance $\hat{\delta}(s,t)$ as correct if it is a $(1, 4r)$-approximation, that is

$$\delta(s,t) \leq \hat{\delta}(s,t) \leq \delta(s,t) + 4r.$$

All distances computed by our distance oracle are based on an actual path in the graph and are thus never shorter than the shortest path distance.

First, we consider the correctness of the global query and the locality filter in Lemma 3.6 and 3.7. Secondly, we observe the correctness of the three combined queries defined in Section 3.2.3. Theorem 3.8 and 3.9 prove the correctness of the first two combined queries. The third, approximate combined query cannot guarantee an upper bounded error.

Lemma 3.6. *If a shortest s-t-path for $s, t \in V$ contains a marked node, the global distance is a $(1, 4r)$-approximation.*

Proof. Let p be the shortest s-t-path containing the maximum number of marked nodes. Let $m_s \in \mathcal{M}$ be the first marked node on p, and $m_t \in \mathcal{M}$ the last, that is $p = s \rightsquigarrow m_s \rightsquigarrow m_t \rightsquigarrow t$. Thus, for the distance from s to t applies

$$\delta(s,t) = w(p) = \delta(s, m_s) + \delta(m_s, m_t) + \delta(m_t, t).$$

Since p contains the maximum number of marked nodes, m_s must be the first marked node on all shortest s-m_s-paths and m_t the last on all shortest m_t-t-paths. Let $a_s \in \mathcal{A}(m_s)$ be the transit node associated with m_s and $a_t \in \mathcal{A}(m_t)$ the one associated with m_t. According to Lemma 3.4, the distance oracle sets a_s as forward access node of s and a_t as backward access node of t and with the distance

$$\mathtt{d}_{\mathcal{A}}(s, a_s) \leq \delta(s, m_s) + r \text{ and } \mathtt{d}_{\mathcal{A}}(a_t, t) \leq \delta(m_t, t) + r.$$

If a_s was removed by the postprocessing, another access node $a_s' \in \mathcal{A}(s)$ of s exists with $\mathtt{d}_{\mathcal{A}}(s, a_s') + \mathtt{d}_{\mathcal{T}}(a_s', a_s) \leq \mathtt{d}_{\mathcal{A}}(s, a_s)$. The same applies for a_t. Thus, the following considerations are not invalidated by the postprocessing.

According to Lemma 3.1 and 3.3, the distance $\mathtt{d}_{\mathcal{T}}(a_s, a_t) = \delta(a_s, a_t)$ is correctly computed. By definition of the marked nodes, $\delta(a_s, m_s) \leq r$ and $\delta(m_t, a_t) \leq r$ applies. Using the triangle inequality, the distance $\mathtt{d}_{\mathcal{T}}(a_s, a_t)$ can be bounded as follows:

$$
\begin{aligned}
\mathtt{d}_{\mathcal{T}}(a_s, a_t) &= \delta(a_s, a_t) \\
&\leq \delta(a_s, m_s) + \delta(m_s, a_t) && \text{(triangle inequality)} \\
&\leq \delta(a_s, m_s) + \delta(m_s, m_t) + \delta(m_t, a_t) && \text{(triangle inequality)} \\
&\leq \delta(m_s, m_t) + 2r
\end{aligned}
$$

In total,

$$
\begin{aligned}
\hat{\delta}_g(s,t) &= \mathtt{d}_{\mathcal{A}}(s,a_s) + \mathtt{d}_{\mathcal{T}}(a_s,a_t) + \mathtt{d}_{\mathcal{A}}(a_t,t) \\
&\leq \delta(s,m_s) + r + \delta(m_s,m_t) + 2r + \delta(m_t,t) + r \\
&\leq \delta(s,m_s) + \delta(m_s,m_t) + \delta(m_t,t) + 4r \\
&= w(p) + 4r \\
&= \delta(s,t) + 4r
\end{aligned}
$$

applies for the global distance which proves $\hat{\delta}_g(s,t) \leq \delta(s,t) + 4r$. Since $\delta(s,t) \leq \hat{\delta}_g(s,t)$ applies for all computed distances, this proves Lemma 3.6. $\qquad\square$

Lemma 3.7. *The locality filter is conservative, that is if* no *shortest s-t-path for $s,t \in V$ contains a marked node, the locality filter returns $\mathcal{L}(s,t) = \mathtt{true}$.*

Proof. Let $s,t \in V$ be two vertices such that no shortest s-t-path contains a marked node $m \in \mathcal{M}$. Consider the covering search from s with $V' = \mathcal{M}$. Since there is no marked node on any shortest s-t-path, the covering search cannot be covered before visiting t. A covering search from t obviously visits t itself meaning both searches visit at least the Voronoi cell containing t. The Graph Voronoi filter $\mathcal{L}$ defined in Section 3.3.1 returns $\mathcal{L}(s,t) = \mathtt{true}$ if the approximated search spaces of the forward covering search from s and the backward covering search from t visit the same Voronoi cell. Therefore, the locality filter answers $\mathcal{L}(s,t) = \mathtt{true}$ and is thus conservative. $\qquad\square$

From Lemma 3.6 and 3.7 follows directly that the global query is a correct $(1,4r)$-approximation if the locality filter returns $\mathcal{L}(s,t) = \mathtt{false}$.

We now consider the correctness of all three combined queries defined in Section 3.2.3.

Theorem 3.8. *The TNR combined query always computes a correct $(1,4r)$-approximation.*

Proof. The combined distance $\hat{\delta}_c(s,t)$ is defined as

$$
\hat{\delta}_c(s,t) = \begin{cases} \hat{\delta}_l(s,t) & \text{if } \mathcal{L}(s,t) = \mathtt{true} \\ \hat{\delta}_g(s,t) & \text{if } \mathcal{L}(s,t) = \mathtt{false} \end{cases}
$$

in Section 3.2.3.

We just showed with Lemma 3.6 and 3.7 that the global distance $\hat{\delta}_g(s,t)$ is always a correct approximation if $\mathcal{L}(s,t) = \mathtt{false}$. Furthermore, the local query without stop criterion always computes the correct distance according to Lemma 3.5. Thus, the distance $\hat{\delta}_c(s,t)$ computed by the TNR combined query is always a correct $(1,4r)$-approximation. $\qquad\square$

Theorem 3.9. *The exact combined query with limited local query always computes a correct $(1,4r)$-approximation.*

Proof. The combined distance $\hat{\delta}_c(s,t)$ is defined as

$$
\hat{\delta}_c(s,t) = \begin{cases} \hat{\delta}_l(s,t) & \text{if } \mathcal{L}(s,t) = \mathtt{true} \text{ and } \hat{\delta}_l(s,t) < \hat{\delta}_g(s,t) - 4r \\ \hat{\delta}_g(s,t) & \text{otherwise} \end{cases}
$$

in Section 3.2.3.

Obviously, the local query computes the correct distances before it is truncated. Since it is truncated at distance $\hat{\delta}_g(s,t) - 4r$, the local distance $\hat{\delta}_l(s,t)$ is correct if $\delta(s,t) = \hat{\delta}_l(s,t) < \hat{\delta}_g(s,t) - 4r$. This proves the correctness of the first case.

The second case occurs only if $\mathcal{L}(s,t) = \texttt{false}$ or $\delta(s,t) \geq \hat{\delta}_g(s,t) - 4r$. As shown by Lemma 3.6 and 3.7, the global distance $\hat{\delta}_g(s,t)$ is a correct approximation if $\mathcal{L}(s,t) = \texttt{false}$. Furthermore, $\delta(s,t) \geq \hat{\delta}_g(s,t) - 4r$ directly implies $\hat{\delta}_g(s,t) \leq \delta(s,t) + 4r$. Since $\delta(s,t) \leq \hat{\delta}_g(s,t)$ applies for all $s,t \in V$, the global distance is a correct approximation. The combined distance is therefore in both cases a correct $(1, 4r)$-approximation. $\square$

The approximate combined query is not necessarily a correct $(1, 4r)$-approximation. If $\texttt{limit} < \hat{\delta}_g(s,t) - 4r$ applies, the previous estimation $\hat{\delta}_g(s,t) \leq \delta(s,t) + 4r$ does not necessarily apply.

In conclusion, all preprocessing variants of our approximate transit node routing distance oracle yield correct results with an additive error of $4r$ if they are combined with either the TNR combined query or the exact combined query with limited local query. The approximate combined query cannot give such a guaranteed maximum error.

We wish to note that the correctness is independent of the selected transit nodes. Selecting good transit nodes can affect computation time, memory usage and the experimental error but not the guaranteed approximation.

4. Voronoi Search Distance Oracle

On most graphs, the global query computes the correct distance for a large proportion of queries. We can, however, not guarantee its correctness and use combined queries to guarantee a maximum error of $4r$. These combined queries are significantly slower than the global query. We furthermore achieve preprocessing times only about as fast as existing exact approaches on social networks and web graphs and significantly slower on other types of networks. It therefore seems worthwhile to investigate increasing the error for faster preprocessing and query times.

In the following, we introduce our *Voronoi search distance oracle.* It is strongly based on Graph Voronoi diagrams (see Section 2.2.1) and is a variant of the previously described aTNR approach (see Chapter 3). Selecting the transit nodes and computing the distance table is done as described in Section 3.1. The access node determination is based on the Graph Voronoi diagram using the transit nodes as Voronoi centers. Intuitively, the closest transit node should cover many shortest paths. When adding more nearby transit nodes, for example the centers of some neighboring cells, most long-distance shortest paths should be covered.

Instead of computing costly covering searches, the Voronoi search distance oracle selects access nodes with a small, bounded local search. The Voronoi centers of cells visited during this search are selected as access nodes. Additionally, the local query is truncated after a certain, relatively small number of visited vertices. This speeds up both the preprocessing and the queries, but it cannot guarantee a maximum error anymore.

In the following, the Voronoi search preprocessing is described in Section 4.1. After that, Section 4.2 defines the query algorithms used to approximate the shortest path distances. Finally, Section 4.3 considers the complexity guarantees of the Voronoi search distance oracle and its correctness. In general, we only describe the differences to aTNR which is defined in Chapter 3.

4.1. Preprocessing

This section describes in detail how the Voronoi search distance oracle precomputes the access nodes. As already mentioned, selecting the transit nodes and computing their distance is done exactly as in the original aTNR algorithm.

The access node determination is based on the Graph Voronoi diagram of the transit nodes. For every vertex, the distance to its assigned Voronoi center is stored. As in Chapter 3, the transit nodes are assigned as their own access nodes.

Instead of the covering searches, a so-called *Voronoi search* is computed from every vertex $v \in V \setminus \mathcal{T}$ to determine its access nodes. The Voronoi search is a single-source shortest path search from the source v and is truncated at the Voronoi cell borders. To identify these borders, the search checks every visited vertex for its Voronoi cell $\mathscr{V}(w)$. If $\mathscr{V}(v) \neq \mathscr{V}(w)$, the search must have crossed a Voronoi cell border. When a border is crossed, the center $\mathscr{V}(w)$ of the visited neighboring cell is added as access node of v. The distance to such an access node $\mathscr{V}(w)$ can be easily computed as $\mathtt{d}_{\mathcal{A}}(v, \mathscr{V}(w)) = \delta(v, w) + \delta(w, \mathscr{V}(w))$. The Voronoi search computes the distance $\delta(v, w)$ and the distance $\delta(w, \mathscr{V}(w))$ is precomputed during the Voronoi diagram computation. Once $\mathtt{A}$ vertices are visited, the Voronoi search is stopped. The tuning parameter $\mathtt{A}$ influences both the preprocessing speed and the number of access nodes. For directed graphs, a forward and a backward Voronoi search are conducted using the inward and outward Voronoi diagram respectively.

Since there are no covering searches and therefore no covering vertices in the Voronoi search distance oracle, marked nodes are of no use in this algorithm. However, it is still possible that superfluous access nodes are added. We therefore reuse the postprocessing described in Section 9 to remove these superfluous access nodes.

4.2. Queries

This section discusses the query algorithm used with the Voronoi search distance oracle. The same global query as described in Section 2.4 and Section 3.2.1 is used. The local query is a bidirectional search as described in Section 3.2.2. However, we use an additional stop criterion which truncates the bidirectional search after $\mathtt{B}$ visited vertices. The parameter $\mathtt{B}$ needs to be small enough, so the local query is not too slow. Additionally, it has to be large enough to correctly determine the distance of very local queries for which the error of the global query is "too large". We developed two different combined queries for the Voronoi search oracle described in Section 4.2.1 and Section 4.2.2.

4.2.1. Combined Query without Locality Filter

The most basic combined query does not use a locality filter at all. Instead, it always computes a local query which is truncated after $\mathtt{B}$ visited vertices. If the local query did not find a path, the query is regarded as global and answered by the global query. This has the advantage of being very straightforward and easy to implement. However, always computing a local search is something transit node routing originally wanted to avoid and it can significantly affect the query time depending on how $\mathtt{B}$ is chosen.

4.2.2. Combined Query with Neighboring Cells Filter

Another approach is to use a locality filter based on the Voronoi cells of s and t. If the locality filter classifies a query as local, a limited local query is conducted. The local query is again stopped after $\mathtt{B}$ visited vertices. If the local query did not find a path, the global distance is returned. This is equivalent to the combined query without a locality filter previously defined in Section 4.2.1. Otherwise, only the global query is computed. This speeds up the query since the local query is not executed for all queries.

Obviously, the locality filter does not improve the approximation quality and can even increase the error. We now define two different locality filters. Both are based on the Voronoi diagram which is always computed during the preprocessing. As in Section 3.3, it is important to use only one of the Voronoi diagrams in directed graphs.

The Same Cell Locality Filter

The *same cell locality filter* simply checks whether the two vertices s and t are in the same Voronoi cell. This can be tested by looking up the Voronoi cells of each vertex and comparing them. Thus, this locality filter is very fast. However, it can result in many queries wrongfully classified as `global`.

The Neighboring Cells Locality Filter

Furthermore, we define a *neighboring cells locality filter*. It classifies two vertices s and t as `local` if they are in the same or neighboring Voronoi cells. Since it classifies more vertex pairs as local, this locality filter potentially reduces the average error while increasing the average query time. It can be further generalized using the x-hop neighborhood of Voronoi cells. That is, s and t are considered to be `local` if they are separated by x or less Voronoi cells.

Additionally to the Voronoi diagram, we need to store which Voronoi cells are adjacent to check whether two vertices are in neighboring cells which is basically a lighter version of the extended Voronoi diagram (see Section 2.2).

4.3. Complexity and Correctness

4.3.1. Complexity

This section gives a short overview of the time and space complexity of the Voronoi search distance oracle compared to the original algorithm which is examined in Section 3.4.1. The transit node selection and the distance table computation are reused from Chapter 3. Thus, the times $T_{\text{TransitNodes}}$ and $T_{\text{DistanceTable}}$ are equivalent to Section 3.4.1. The time to determine the access nodes is described by $T_{\text{AccessNodeDetermination}} = n \cdot T_{\text{VS}} + T_{\text{Post}}$ where T_{VS} is the running time of a single Voronoi search.

A Voronoi search is stopped after visiting `A` vertices. Let $\Delta_{\max}$ be the maximum vertex degree in the graph, then the Voronoi search observes at most $\texttt{A} \cdot \Delta_{\max}$ edges and thus has a complexity of $T_{\text{VS}} = O(\texttt{A} \log \texttt{A} + \texttt{A} \cdot \Delta_{\max})$ on general graphs and $T_{\text{VS}} = O(\texttt{A} + \texttt{A} \cdot \Delta_{\max})$ on unweighted graphs.

Let $V_{\Delta_{\max}} \leq k$ be the maximum number of direct neighbors in the Voronoi diagram. Since only the centers of the neighboring Voronoi cells are selected as access nodes, their number is bounded by $O(V_{\Delta_{\max}})$. Thus, the postprocessing time can be improved to $T_{\text{Post}} = O(n \cdot V_{\Delta_{\max}}^2)$ and the space complexity of the preprocessing can be improved to $O(n \cdot V_{\Delta_{\max}})$.

Furthermore, the complexity of the global query is improved to $O(V_{\Delta_{\max}}^2)$. Similar to the Voronoi search, the local query is truncated after `B` visited vertices. This yields a running time of $O(\texttt{B} \log \texttt{B} + \texttt{B} \cdot \Delta_{\max})$ or $O(\texttt{B} + \texttt{B} \cdot \Delta_{\max})$ on unweighted graphs. The same cell locality filter is just a table-lookup and thus constant. The neighboring cells locality filter needs to check at most all neighboring cells which takes $O(V_{\Delta_{\max}})$ time. In total, the combined query has a time complexity of $O(V_{\Delta_{\max}}^2 + \texttt{B} \log \texttt{B} + \texttt{B} \cdot \Delta_{\max})$ or $O(V_{\Delta_{\max}}^2 + \texttt{B} + \texttt{B} \cdot \Delta_{\max})$ on unweighted graphs.

4.3.2. Correctness

All distances computed by the Voronoi search distance oracle are based on an actual path in the graph. Thus, the exact shortest path distance $\delta(s, t)$ is a lower bound of all computed distances $\hat{\delta}(s, t)$. As already mentioned in the introduction to this chapter, the Voronoi search distance oracle cannot guarantee a maximum error of the computed distances. We demonstrate this in the following using a specifically defined graph as an example. For

the sake of simplicity, we consider an undirected graph. It can be transferred to directed graphs by replacing all undirected edges with two directed edges with the same weight.

We define a graph $G_{[0,k]}, k \in \mathbb{N}_0$ consisting of k *building block graphs* $G_i, i \in [0, k]$. One of these building block graphs G_i is illustrated in Figure 4.1. It has two vertices v_i and T_i, where T_i is a transit node. They are connected by the edge $e_i = \{v_i, T_i\}$ with weight $w(e_i) = \mathtt{w} \in \mathbb{R}_0^+$. To motivate T_i being a transit node, one could imagine T_i as the center of a star graph. That is, there are many degree-one vertices connected only to T_i. All their shortest paths must contain T_i and, thus, T_i is on many shortest paths.

Multiple of these building block graphs can be combined to create the graph $G_{[0,k]}$ by connecting the vertices v_i and v_{i+1}, $0 \leq i < k$ with the unit-weight edge $e_{i,i+1} = \{v_i, v_{i+1}\}$, $w(e_{i,i+1}) = 1$. The Voronoi cells of the graph $G_{[0,k]}$ correspond exactly to the building blocks G_0 to G_k. Figure 4.1 shows an example of such a graph $G_{[0,4]}$.
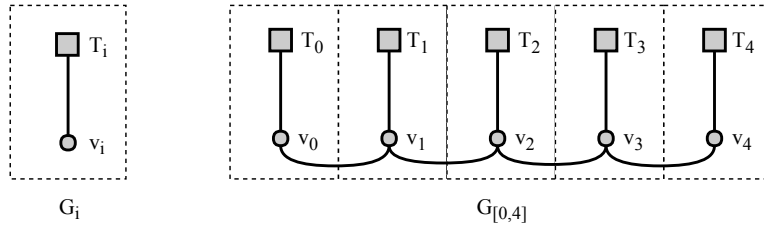
Figure 4.1.: Left: A single building block graph G_i with its two vertices T_i and v_i.
Right: $G_{[0,4]}$ created by connecting 5 building blocks via the vertices v_i.

Since all edge weights are non-negative, the shortest v_0-v_k-path is always the path $p_{[0,k]} = \langle v_0, v_1, \dots, v_{k-1}, v_k \rangle$ consisting of all v_i. It contains $k+1$ vertices in $k+1$ different Voronoi cells and has the distance

$$\delta(v_0, v_k) = w(p_{[0,k]}) = \sum_{i=0}^{k-1} w(e_{i,i+1}) = \sum_{i=0}^{k-1} 1 = k.$$

Let $p'_{[0,k]}$ be the shortest v_0-v_k-path that contains a transit node, that is $p'_{[0,k]} = v_0 \rightsquigarrow T_j \rightsquigarrow v_k$, $j \in [0, k]$ such that $w(p'_{[0,k]})$ is minimal. Obviously, $w(p'_{[0,k]}) \geq w(p_{[0,k]}) + 2\mathtt{w} = k + 2\mathtt{w}$ because the path has to at least make the detour $\langle v_j, T_j, v_j \rangle$ with distance $2\mathtt{w}$.

We can unrestrictedly increase the edge weight $\mathtt{w} \in \mathbb{R}_0^+$. Thus, the minimum error $2\mathtt{w}$ of any v_0-v_k-path via a transit node is unbounded. Furthermore, the number of blocks $k \in \mathbb{N}_0$ composing the graph $G_{[0,k]}$ is not limited. Increasing k directly affects the shortest path distance, the number of vertices on the shortest path, and the number of Voronoi cells visited by the shortest path. Therefore, none of these metrics can be effectively used to detect `local` paths. In total, this means that our Voronoi search distance oracle cannot guarantee a maximum error and no locality filter based purely on the considered metrics could give such a guarantee.

5. Implementation

We implemented the approximate transit node routing in C++ using the Ligra graph framework [SB13, SDB15]. This chapter gives a short overview of the implementation. At first, Section 5.1 describes the features of Ligra. The following Section 5.2 covers the interesting aspects of our distance oracle implementation.

5.1. The Ligra Graph Processing Framework

Ligra [SB13, SDB15] is a "lightweight graph processing framework [specialized] for shared-memory parallel/multicore machines" by Shun and Blelloch [SB13]. It offers "a *vertexSubset* data structure used for representing a subset of the vertices" [SDB15] and two mapping functions – EDGEMAP and VERTEXMAP. EDGEMAP is used to map over all outgoing edges and VERTEXMAP is used to map over all vertices of a given subset of vertices. The Ligra framework automatically decides between a sparse and a dense implementation of the EDGEMAP depending "on the number of vertices and outgoing edges in the vertexSubset" [SDB15]. Processing the edges or vertices in parallel is encapsulated in these functions. A "programmer must [only] ensure the parallel correctness of the functions passed to EDGEMAP and VERTEXMAP" [SDB15]. Ligra makes the implementation of graph traversal algorithms – such as breadth-first search – easy, efficient, and scalable [SB13].

5.2. Implementation of our Approximate TNR Oracle

Our implementation handles only unweighted and undirected graphs. This restriction allows us to take full advantage of Ligra's EDGEMAP as all single- and multi-source shortest path searches can be implemented with a BFS instead of Dijkstra's algorithm. Undirected graphs furthermore avoid the need to differentiate between forward and backward definitions and simplifies selecting the marked nodes.

We assume a pre-existing balanced partition of the graph into k components and a pre-computed betweenness centrality. Consequently, we do not consider their computation in the running time of our preprocessing. In our experiments, we approximate the betweenness centrality from $n^{\frac{5}{8}}$ randomly selected source vertices using the implementation of Brandes' algorithm [Bra01] provided by the Ligra framework [SB13]. All graphs are pre-partitioned into k components using the fastsocial configuration of KaHIP [MSS15]. The partitioning is done separately and is not part of our implementation.

Processing Order in the Conservative Covering Search

The conservative covering search is BFS-based since our implementation only considers unweighted graphs. This means that the vertices are processed in levels. If a vertex is reachable via both a covered and an uncovered vertex, it should always be visited from the covered vertex to maximize the number of covered vertices in each level. We ensure this by splitting the current level into two sets, a set of covered and a set of uncovered vertices. The covered set is always processed first, allowing it to cover the maximum amount of vertices in the next level. Once the uncovered set is empty, the search is completely covered and thus stopped.

5.2.1. Data Structures

We now briefly describe the data structures used in our implementation.

The Distance Table

The distance table can be regarded as a $k \times k$ matrix. For undirected graphs, this matrix is symmetric. It is therefore sufficient to store only the lower triangular matrix. Furthermore, the distance from every vertex $t \in \mathcal{T}$ to itself is always $\delta(t, t) = 0$. Thus, it is not necessary to store the main diagonal of the matrix. This saves about 50 % of the memory required for the distance table. For improved efficiency, we store the distance table in a single, one-dimensional array instead of a two-dimensional array.

Access Nodes and Locality Filter

Both the access nodes and the approximated search spaces are stored in a data structure similar to an adjacency array. This structure is also used by [ALS13], and we reuse their notation in the following. An array A stores all access nodes $a \in \mathcal{A}(v)$ of all vertices $v \in V$ and their distance $\mathtt{d}_\mathcal{A}(v, a)$. It is grouped by the vertices v and sorted by the access nodes a. The access nodes and distances are stored alternatingly in A. A second array I_A stores for each vertex $v \in V$ the index of A at which the access nodes $\mathcal{A}(v)$ start. We use a sentinel value $I_A[n]$ to store the length of A. The locality filter is correspondingly stored in the array S and I_S. However, S contains only the visited Voronoi cells and no distances. To determine whether an *s-t*-query is `local`, we use a simple *merge* operation similar to the one in merge sort. It searches for a Voronoi cell visited by both s and t by iterating simultaneously through their visited cells always increasing the lower index.

The Covering Searches

During the covering searches, we need to store the visited vertices and their distance. In general, the covering searches can visit all n vertices. However, in many cases, the search space is much smaller. We implemented two different versions of such a data structure. The first variant consists of an array of size n. This array is reused for consecutive covering searches to avoid unnecessary overhead caused by reallocating the array for each search. In-between two searches the array needs to be cleared. To avoid clearing many unmodified cells when only a small fraction of the vertices was visited, we keep a stack of modified cells and clear only those cells. The entire array is cleared if more vertices are visited than a specified limit. We currently use $\sqrt{n}$ as limit. However, parameter tuning this limit might improve the performance further.

The second variant of the data structure uses growable hash tables to avoid using much more memory than actually necessary. They are initialized with a rather small size; if more vertices are visited, the hash table size is increased. Furthermore, reallocating such a small growable hash table is usually faster than clearing or reallocating an array of size n. In our

implementation, we use the growable hash table by Maier et al. [MSD16].

However, the space required by the arrays was not prohibitive in our experiments. Furthermore, the array outperformed the hash table by about a factor of three in the covering search running time. We therefore use arrays for the covering searches in all our experiments. The hash table based data structure is still provided and can be useful once the memory requirement of the arrays becomes prohibitive.

6. Evaluation

This chapter contains the experimental analysis and evaluation of our approximate TNR algorithm previously described in Chapter 3. First, Section 6.1 describes our methodology. After that, Section 6.2 analyzes the preprocessing followed by the evaluation of the queries in Section 6.3. Finally, Section 6.4 compares our algorithm to other approaches.

6.1. Experimental Setup

All experiments were conducted on a 64-core machine with four AMD Opteron 6278 processors clocked at 2.4GHz. It has 16 KB L1d, 64 KB L1i, 2 MB L2 cache and 6 MB L3 cache. Our C++ code was compiled with version 5.3.0 of the g++ compiler and `-o2` optimization using CilkPlus [Lei10] for parallelization.
The remainder of this section introduces the graphs we use as input data including a short description and an overview of their key properties.

6.1.1. Data Sets

We use real-world network graphs provided by the Stanford Large Network Dataset (SNAP) Collection [LK14]. The road network of California is used as a comparison to the original transit node routing which was developed for road networks. All graphs are interpreted as undirected and unweighted. We consider only the largest connected component of each graph to avoid edge cases like unreachable vertices. Our set of inputs overlaps significantly with [AIY13] to which we compare our results directly in Section 6.4. We provide a short description of each graph in the following paragraphs. Table 6.1 gives an overview of their properties.

Gnutella. Gnutella is a peer-to-peer file sharing network. This graph depicts the network between different Gnutella hosts. It was obtained by [RFI02] on August 31, 2002 [LK14, LKF07].

Epinions. Epinions is a customer review website whose users can form trust relationships. This graph represents these trust relationships [LK14]. It was obtained by Richardson et al. [RAD03] and is provided by [LK14].

Slashdot. Slashdot is a news website focused on technology news and user moderated content. Users can add each other as friends or foes [LK14]. We use the network graph of these relationships observed in February 2009 [LLDM09].

Name	Type	n	m	Diameter
Gnutella	Computer	63 K	148 K	11
Epinions	Social	79 K	405 K	14
Slashdot	Social	82 K	504 K	11
NotreDame	Web	326 K	1.1 M	46
CA	Road	2.0 M	2.8 M	849
Wiki-Talk	Social	2.4 M	4.7 M	9
Skitter	Computer	1.7 M	11 M	25
Pokec	Social	1.6 M	22 M	11
Orkut	Social	3.0 M	117 M	9

Table 6.1.: Overview of the input graphs used in our experiments. The graphs are classified into different network types according to [LK14, AIY13] and sorted by m.

NotreDame. The NotreDame network contains the web pages of the University of Notre Dame connected by hyper-links between the pages [LK14]. It was obtained in 1999 by Albert et al. [AJB99].

CA. This is a graph of the road network of the state of California obtained on by [LLDM09] and provided by [LK14].

Wiki-Talk. The Wiki-Talk network was obtained in January 2008 and depicts registered Wikipedia users connected if they communicated via their individual talk page [LK14, LHK10b, LHK10a].

Skitter. The Skitter network is an internet topology graph obtained from traceroutes in 2005 [LK14, AIY13] by Leskovec et al. [LKF05].

Pokec. Pokec is a popular Slovak social network. The Pokec graph represents the friendships between Pokec users. We use the version created by Takac and Zabovsky [TZ12] in December 2011.

Orkut. The Orkut graph shows the friendships between users in the free online social network Orkut and is provided by [MMG$^+$07, LK14, YL15].

6.2. Preprocessing

This section analyzes the aTNR preprocessing as described in Section 3.1. At first, Section 6.2.1 compares the transit node selection algorithms and determines the best set of transit nodes for each graph. Secondly, Section 6.2.2 evaluates the running time of the distance table computation. Thirdly, Section 6.2.3 analyzes the access node determination by comparing the three different variants and examining the postprocessing. After that, Section 6.2.4 summarizes the preprocessing time. Finally, the size of the preprocessed data is analyzed in Section 6.2.5.

6.2.1. Transit Node Selection

This section evaluates different sets of transit nodes. Good transit nodes result in fewer access nodes and faster preprocessing times. We compare the four transit node selection algorithms introduced in Section 3.1.1. For the four smaller graphs, we additionally analyze randomly selected transit nodes.

In this section, the preprocessing variant based on the conservative covering search is used exclusively. A comparison of the different covering search variants can be found in

| | | Running Time | | | Access Nodes/Vertex | |
		TNS	AND	Marked	w/o Post.	w/ Post.
Gnutella	simple	0.6 ms	10.9 s	9.8%	191.8	81.2
	partition	0.5 ms	10.8 s	8.9%	202.4	98.6
	farthest	0.7 s	12.7 s	1.0%	244.5	244.3
	k-Means	1.0 s	12.0 s	2.1%	223.6	215.5
	random	0.4 ms	13.1 s	2.3%	226.6	214.9
Epinions	simple	1.0 ms	2.5 s	36.5%	3.1	1.7
	partition	3.9 ms	2.5 s	33.7%	3.6	1.9
	farthest	0.7 s	21.5 s	4.8%	165.6	160.5
	k-Means	1.2 s	20.7 s	2.6%	150.6	138.7
	random	0.6 ms	22.3 s	2.4%	161.1	151.0
Slashdot	simple	0.9 ms	3.9 s	38.5%	4.7	2.5
	partition	0.5 ms	4.8 s	35.6%	6.0	2.9
	farthest	0.8 s	34.0 s	3.8%	245.1	244.2
	k-Means	1.3 s	31.0 s	2.3%	213.5	205.5
	random	1.2 ms	31.7 s	2.9%	209.2	195.2
NotreDame	simple	2.4 ms	49.4 s	39.9%	4.1	1.6
	partition	2.2 ms	44.4 s	38.8%	4.3	1.7
	farthest	4.0 s	272.8 s	3.6%	116.7	80.0
	k-Means	8.1 s	217.2 s	1.2%	81.3	77.6
	random	1.3 ms	238.6 s	1.1%	103.9	96.3
CA	simple	13.3 ms	5963.27 s	0.2%	16.6	9.2
	partition	12.0 ms	1017.97 s	0.3%	12.0	9.4
	farthest	111.8 s	≈ 15000 s	0.2%	≈ 950	
	k-Means	169.0 s	≈ 15000 s	0.3%	≈ 800	
Wiki-Talk	simple	15.2 ms	41.9 s	76.6%	1.2	1.1
	partition	15.5 ms	32.4 s	72.2%	1.1	1.1
	farthest	36.4 s	≈ 16000 s	4.3%	≈ 900	
	k-Means	106.1 s	≈ 11000 s	0.2%	≈ 300	
Skitter	simple	9.2 ms	8399.57 s	38.4%	32.8	4.9
	partition	10.9 ms	9602.85 s	37.4%	31.8	5.5
	farthest	31.8 s	≈ 17000 s	1.6%	≈ 1200	
	k-Means	69.9 s	≈ 15000 s	1.0%	≈ 850	
Pokec *r* = 2	simple	11.3 ms	443.75 s	86.0%	1.5	1.3
	partition	10.2 ms	601.81 s	84.0%	1.6	1.4
	farthest	44.1 s	≈ 30000 s	25.9%	≈ 800	
	k-Means	79.5 s	≈ 22000 s	45.7%	≈ 550	
Orkut	simple	17.9 ms	202.60 s	97.3%	1.1	1.1
	partition	35.7 ms	362.92 s	97.0%	1.2	1.1
	farthest	114.3 s	≈ 32000 s	43.9%	≈ 1300	
	k-Means	209.3 s	≈ 20000 s	80.6%	≈ 300	

Table 6.2.: Transit Node Selection (TNS) algorithms compared regarding their running time, the running time of the access node determination (AND), the percentage of marked nodes and the resulting access nodes per vertex before (w/o Post.) and after the postprocessing (w/ Post.).

Section 6.2.3. If the access node determination takes longer than ten thousand seconds, it is terminated. In those cases, an approximation (denoted by $\approx$) of the running time and the resulting access nodes is given based on a sample from one thousand random vertices. In general, a marking radius of $r = 1$ is used. The preprocessing of both Pokec and Orkut takes significantly longer than the ten thousand seconds limit – independent of the transit nodes. We therefore set $r = 2$ for those graphs.

Table 6.2 gives a complete overview of the results. For each graph, the best transit nodes are highlighted. The Simple-BC and Partition-BC transit nodes yield the best results on all graphs. They mark significantly more vertices, have a shorter preprocessing time and result in substantially fewer access nodes. Both the Farthest and kMeans-BC algorithms perform significantly worse than Simple and Partition-BC and provide no notable improvement over random transit nodes. In all further experiments, only the best transit node set for each graph is used. For CA and Wiki-Talk, the Partition-BC works best. For all other graphs, Simple-BC transit nodes are used.

6.2.2. Distance Table

The time to compute the distance table between all transit nodes (for the transit node set specified in Section 6.2.1) is presented in Table 6.3. We state both the total running time in seconds and the relative time compared to the average BFS running time on that specific graph. There is only little variation between the running times of the different transit node sets. As shown in Table 6.3, the time to compute the distance table correlates roughly with the time to compute k BFS. This correlation is as expected since the naive approach (see Section 3.1.2) consists of computing a BFS from each of the k transit nodes with some additional overhead to construct the distance table.

| | Distance Table | | |
Graph	Time	Time/BFS	k
Gnutella	0.7 s	337	250
Epinions	0.8 s	347	275
Slashdot	0.8 s	353	287
NotreDame	4.7 s	640	571
CA	241.8 s	1483	1399
Wiki-Talk	26.9 s	1809	1546
Skitter	28.1 s	1577	1302
Pokec	36.3 s	1537	1278
Orkut	107.0 s	1955	1753

Table 6.3.: Running time to compute the complete distance table between all transit nodes in seconds and divided by the average time to compute a BFS compared to the number of transit nodes.

6.2.3. Access Node Determination

This section evaluates the two main steps of the access node determination: determining the initial access nodes with the three different covering searches introduced in Section 2.5 and removing superfluous access nodes with the postprocessing. As in Section 3.1.3, we use a marking radius of $r = 1$ for the seven smaller graphs and $r = 2$ for Pokec and Orkut. We abbreviate the covering searches with *cons*, *aggr*, and *stal* in the following tables and figures.

Covering Searches

Table 6.4 gives the running times of all evaluated access node determinations divided by the number of edges. On most graphs, the aggressive and stall-on-demand approaches are significantly faster than the conservative approach. An exception to this are the graphs Gnutella and CA. On Skitter, the running time differences are also not as significant as on the other graphs. These three graphs repeatedly form an exception throughout this chapter.

	Gnutella	Epinions	Slashdot	NotreDame	CA	Wiki-Talk	Skitter	Pokec	Orkut
cons	74.59	5.87	7.79	44.81	373.01	6.04	763.77	13.69	1.70
aggr	52.41	0.47	0.67	6.06	3835.40	0.26	494.37	0.02	0.00
stal	319.05	0.62	0.46	6.54	4993.17	0.29	190.05	0.03	0.00

Table 6.4.: Average running time in microseconds per edge of the covering searches used to determine the access nodes.

The average number of access nodes per vertex both before and after the postprocessing is given in Table 6.5. Section 6.2.3 discusses the postprocessing and the number of removed access nodes in more detail. It is evident that the conservative approach yields the least amount of initial access nodes, followed by the stall-on-demand approach. The aggressive variant always results in the most access nodes.

This behavior is as expected. The aggressive covering search often searches around marked nodes at which it was stopped, reaching other marked nodes via paths that have long been covered. This can add many unnecessary access nodes. The stalling search performed by stall-on-demand detects a proportion of these sub-optimal paths and removes the corresponding access nodes by stalling. The conservative approach adds the fewest access nodes since all vertices are visited via an actual shortest path. Unnecessary access nodes can only be added if there are multiple shortest paths of which only some contain a marked node. The impact of these multiple shortest paths is further reduced by the processing order described in Section 5.2.

		Gnutella	Epinions	Slashdot	NotreDame	CA	Wiki-Talk	Skitter	Pokec	Orkut
before	cons	191.59	3.07	4.74	4.13	11.99	1.11	31.56	1.48	1.13
	aggr	196.39	9.45	15.91	36.87	1393.44	1.12	498.28	1.56	1.17
	stal	195.86	3.97	6.58	13.78	1372.59	1.12	169.59	1.53	1.13
after	cons	80.72	1.71	2.49	1.64	9.44	1.10	4.74	1.34	1.11
	aggr	80.89	1.71	2.49	1.64	9.60	1.10	4.75	1.34	1.11
	stal	81.02	1.71	2.49	1.63	9.38	1.10	4.72	1.34	1.11

Table 6.5.: Average access nodes per vertex before and after the postprocessing.

After the postprocessing removed unnecessary access nodes the difference between the three approaches are negligible, the number of resulting access nodes is almost identical. The selected covering search has therefore no significant impact on the final number of access nodes. However, it still affects the locality filter (see Section 6.2.5 and Section 6.3.1),

which is based on the search space of the covering search. Except for Gnutella, the number of access nodes can be reduced to less than ten on all graphs.

Postprocessing

The postprocessing is conducted at the end of each access node determination and removes superfluous access nodes. As shown by Table A.1, computing the postprocessing takes less than a second on all graphs, except CA and Skitter which are among the largest graphs processed with $r = 1$.
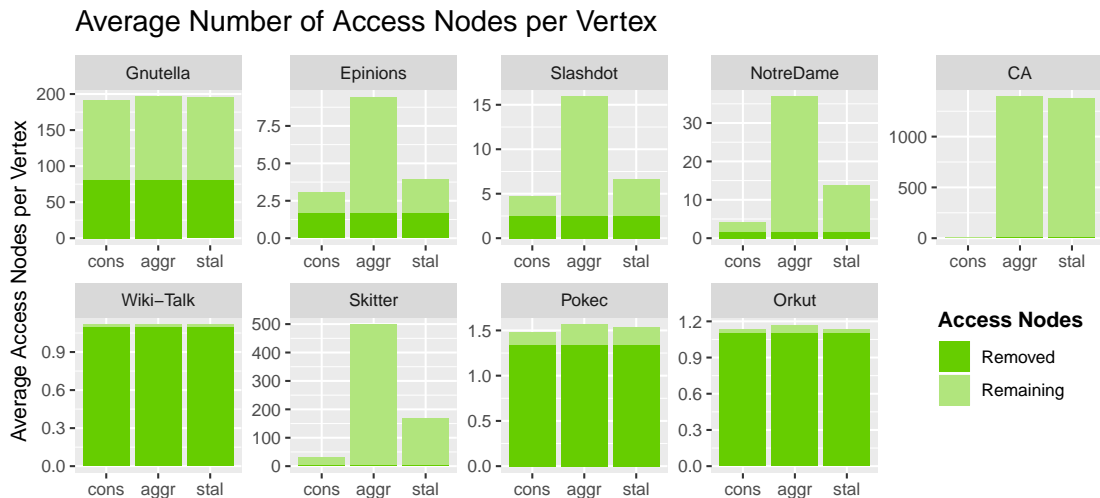

Figure 6.1.: Overview of the number of access nodes removed by the postprocessing. The total bar height corresponds to the access nodes initially added by the covering searches. Of these access nodes, those colored light-green are removed by the postprocessing leaving only the nodes colored bright-green in the final set.

Figure 6.1 is an illustration of Table 6.5 and showcases the number of superfluous access nodes that are removed by the postprocessing. In some instances, up to 99 % of the initial access nodes are removed. The number of remaining access nodes is very similar for all three variants of the covering search regardless of the initial differences. This suggests that the resulting set of access nodes is close to the minimal required set. The postprocessing reduces the average number of access nodes per vertex to less than ten, reaching similarly small label sizes as TNR on road networks [BFM+07, ALS13]. Only the Gnutella network requires about 80 access nodes per vertex even after the preprocessing.

6.2.4. Preprocessing Time Overview

Figure 6.2 presents an overview of the time required for each step of the preprocessing by instance and variant of the access node determination. Furthermore, the exact running times can be found in Table A.1. Selecting the transit nodes and computing the distance table is the same for all three variants of the covering search. The running times are therefore equal across all three approaches.
The preprocessing is dominated by determining the access nodes with the covering search and by computing the distance table. Selecting the transit node set, creating the locality filter from the covering search search-spaces, and removing superfluous access nodes with the postprocessing all consume only a negligible part of the total running time.

Computing the distance table takes between $1\,\mu s$ and $10\,\mu s$ per edge on all instances and is roughly proportional to the number of BFS conducted to compute it (see Section 6.2.2).
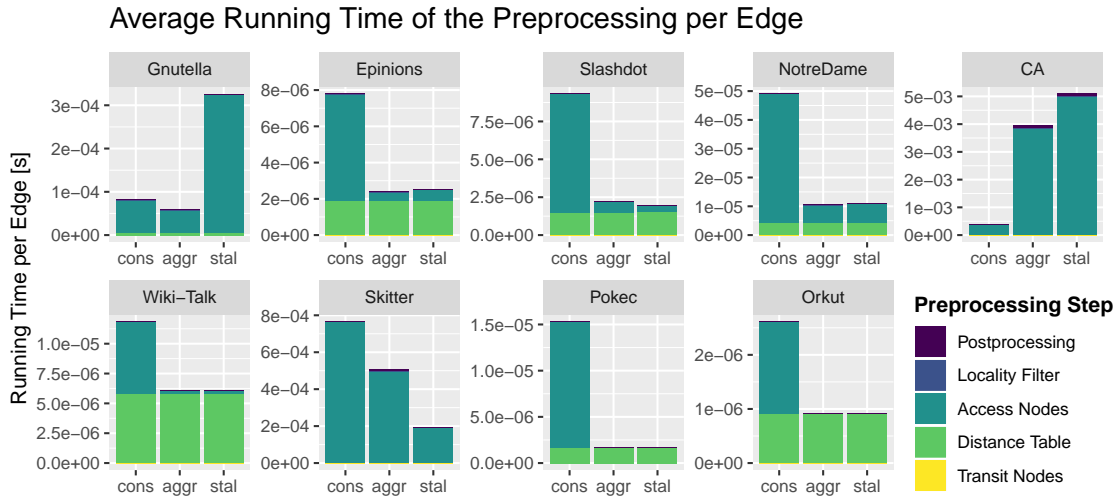
Figure 6.2.: Running time of the complete preprocessing for each graph split into the different preprocessing steps and scaled by the number of edges.

The access node determination ranges between $250\,ns$ per edge reached by the Aggressive approach on Wiki-Talk and $5\,ms$ per edge required by Stall-on-Demand on the CA road network. The total running time for our preprocessing on the social networks and web graphs varies between 1–$50\,\mu s$ per edge depending on the graph and the selected covering search variant. Preprocessing the road network CA and the computer networks Gnutella and Skitter takes significantly longer, mainly due to the slow access node determination. The poor performance of the road network CA is somewhat expected since the marked nodes do not work too well on road networks. Due to the long diameter and low vertex degree, only very few vertices are marked. As shown by Table 6.2, less than one percent of CA's vertices are marked while for most other graphs more than a third are marked. On CA, we therefore need to compute more covering searches that are also not as easily truncated thus causing a significantly longer running time.

6.2.5. Space Consumption

The space consumption of the preprocessed data is determined by the distance table, the access nodes, and the locality filter. An overview of the different parts that make up the total size is given in Figure 6.3. Table 6.6 gives the total space requirement per vertex. The preprocessing is currently stored as a text file. Converting this to a more efficient storage method should save some additional space.

The distance table grows quadratically with the number of transit nodes k. We set $k = \sqrt{n}$ resulting in a linear size distance table. Except for Gnutella, the final number of access nodes is less than ten per vertex on all graphs and all three preprocessing variants. This is reflected in the size of the access nodes data structure.

Furthermore, the approximated search space of each vertex – comprised of the visited Voronoi cells – is stored for the locality filter. As shown by Figure 6.3, these search spaces account for a large proportion of the total space requirement. Some graphs, such as CA and Skitter, spend almost all of their required memory on the locality filter. Reducing the space consumption of the locality filter therefore seems promising for reducing the total required space and is interesting for future work. The locality filter size relates to the number of initial access nodes illustrated in Figure 6.1.
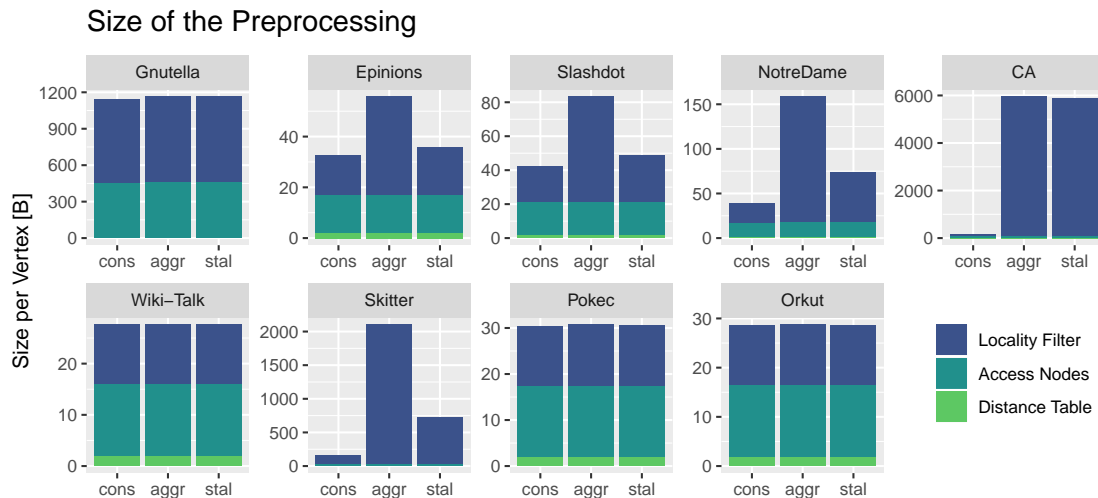
Figure 6.3.: Overview of the preprocessing size in bytes per vertex.

Looking at the total space consumption per vertex shown by Table 6.6, most graphs are processable with a size of less than 50 bytes per vertex. This does not apply for the Gnutella, CA, and Skitter graphs whose preprocessing size ranges between 140 and 6000 bytes.

	Gnutella	Epinions	Slashdot	NotreDame	CA	Wiki-Talk	Skitter	Pokec	Orkut
cons	1147.06	32.59	42.25	39.33	139.91	27.64	166.88	30.41	28.60
aggr	1165.12	55.78	83.38	158.67	5954.41	27.64	2104.99	30.76	28.77
stal	1164.70	35.71	48.51	73.86	5863.28	27.64	726.87	30.64	28.60

Table 6.6.: Total size of the preprocessing in bytes per vertex.

6.3. Queries

This section evaluates the performance of the queries and the quality of their computed distances. Additionally, we analyze the locality filter in Section 6.3.1.

Five different query algorithms are evaluated. First of all, the local and global query are considered on their own as they are described in Section 3.2.1 and 3.2.2. Secondly, the three different variants of the combined query introduced in Section 3.2.3 are analyzed. We denote the original TNR combined query by *combined-TNR*, the exact combined query with limited local query by *combined-exact*, and the approximate combined query by *combined-approx*. A limit of five is used for the approximate combined query.

All queries are evaluated on the same set of point-to-point queries. From each distance in the graph, one thousand random vertex pairs are selected and used as query input. The same pairs are also used to evaluate the locality filter.

6.3.1. Locality Filter

The locality filter is an essential part of our query algorithm. It is called by each combined query and should therefore be reasonably fast to avoid slowing down the query. Figure 6.4

illustrates the running time necessary to classify a vertex pair as local or global. There is only little variation in the running time between the different graphs and preprocessing approaches. Most notably are the CA and Skitter graph with a slightly increased running time compared to the other graphs. As mentioned in Section 6.2.5, these are also the graphs with the largest locality filter data structure.
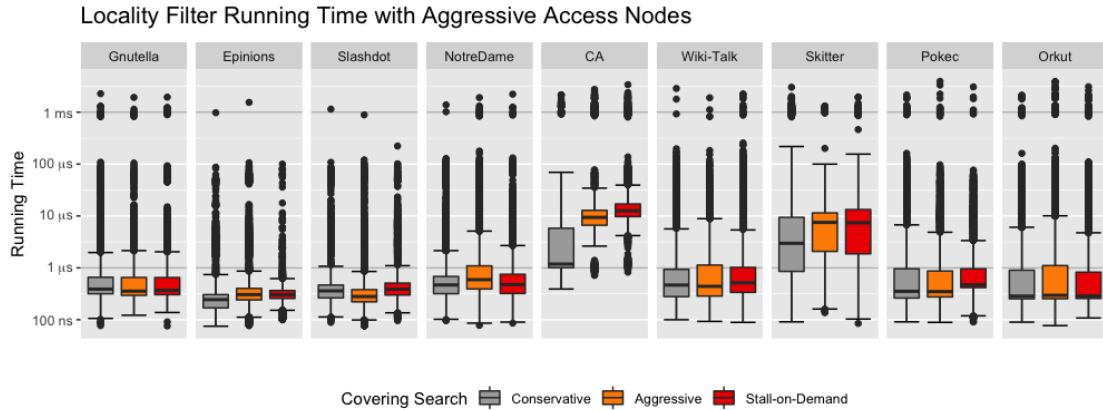

Figure 6.4.: Running time of the locality filter.

Even more important than its running time is the quality of the locality filter as it decides between running a fast global query over a much slower local query. This quality depends on the number of correctly classified vertex pairs compared to those falsely classified as local or global. For the correctness of the combined queries, local vertex pairs should never be classified as global. To determine the actual locality of a vertex pair all shortest paths between them are checked for marked nodes. If there exists a shortest path containing a marked node, the query is global; otherwise, it is local. This locality is then compared to the classification by the locality filter. An overview of the locality filter accuracy and correctness is given in Figure 6.5. There are no queries wrongly classified as global. Thus, our locality filter is correct. This is in accordance with the proof given in Section 3.4.2.
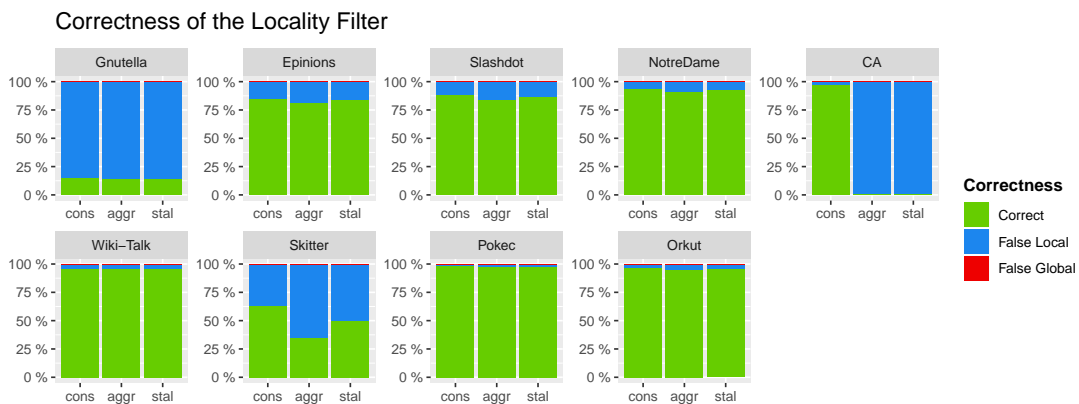
Figure 6.5.: Accuracy of the locality filter. A classification is false local if it is classified as local but should be global. Accordingly, it is false global if it is classified as global but should be local.

Furthermore, Table 6.7 contains the percentage of queries recognized as global for each graph. This directly affects the running time of the combined queries, which is discussed in Section 6.3.2. More global queries directly result in faster combined queries.

	Gnutella	Epinions	Slashdot	NotreDame	CA	Wiki-Talk	Skitter	Pokec	Orkut
cons	4.56	85.08	88.04	92.96	95.87	96.39	63.12	98.28	96.57
aggr	4.10	80.92	83.63	90.45	0.00	96.38	35.48	98.12	95.48
stal	4.10	83.57	86.61	92.17	0.00	96.38	49.87	98.14	96.23

Table 6.7.: Percentage of global queries recognized by the locality filter.

On most graphs, our locality filter is very effective. A large percentage of the queries is correctly classified as global. This leads to many fast global queries.

There are, however, some graphs – specifically Gnutella, CA, and Skitter – on which the locality filter classifies much more queries as local than necessary. This is also reflected in their query time and correctness described in the following sections and their preprocessing performance evaluated in Section 6.2. When comparing these three graphs to the rest, it is noticeable that these groups are represented in the classification of the networks (see Table 6.1). Our distance oracle works well for the social networks and web graphs. Road networks and computer networks are still solved correctly but require much more time and space. These results are not surprising since aTNR is specifically designed for social networks and web graphs.

6.3.2. Query Time

The query running time is in general not strongly affected by the different preprocessing variants. Figure 6.6 shows the query performance based on the conservative preprocessing. Unless mentioned otherwise, the running times for the other two approaches are comparable. A complete overview of all variants is given in Figure B.1 and Table B.2 in the appendix.
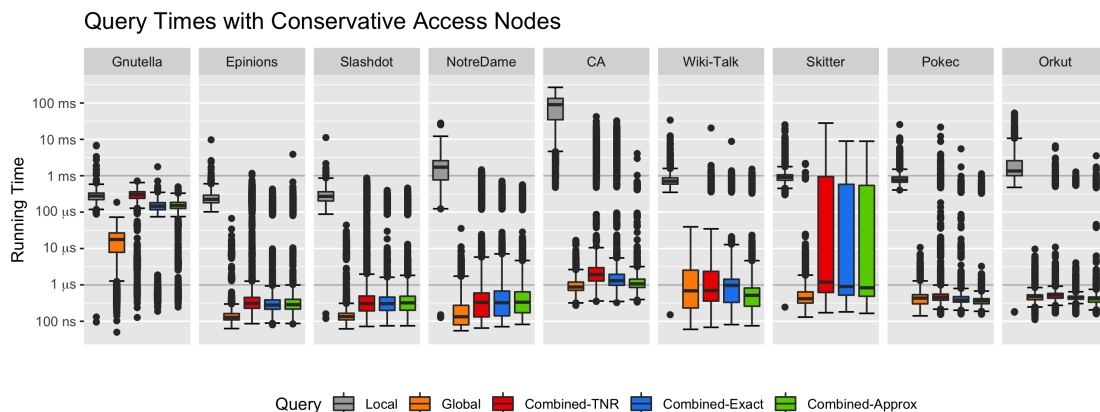
Figure 6.6.: Query running time with preprocessing based on conservative access nodes.

As stated in Table B.2, the average running time of the global query is less than one microsecond for most graphs. Gnutella has the slowest global query with an average running time of about $15\,\mu s$. This disparity can be explained by the much higher number of access nodes (see Table 6.5) since Gnutella is the only graph with more than ten access nodes per vertex.

For all three combined queries, it applies that the majority of point-to-point queries is about as fast as the global query. This is in accordance with the findings in Section 6.3.1

since most queries are correctly classified as global. The average running time as shown in Table B.2 is increased by slow outliers – those queries classified as local. These local pairs make up a rather small proportion of all tested vertex pairs but have a significant impact on the average performance due to their very long running time. This divide of the queries by their locality is also apparent in Figure 6.6 as the outliers of the combined queries are often separated into two groups: one with a running time similar to the local query, and a faster group whose performance is not much worse than the global query.

The combined-exact query is the fastest of the considered queries that still guarantees a maximum additive error of $4r$. It offers a slight improvement over the combined-TNR query and is up to two times faster. The approximate combined query is slightly faster than this but cannot guarantee the same additive error. Concerning the running time, its effect is negligible on most tested graphs. The only graph for which it offers a significant speedup is the road network CA. However, it also introduces a much higher error of up to 35 as discussed in Section 6.3.3. This effect is generally expected since a limit of five has a much higher impact on a road network with a diameter of more than eight hundred than on a social network with diameter ten. An overview of the best average query times reached while guaranteeing an additive error of at most $4r$ is given in Table 6.8.

For CA and Skitter, the running time of the combined queries increases significantly if a preprocessing based on the aggressive or stall-on-demand covering search is used. The combined queries are not much faster than the local query for these variants. On Gnutella even the conservative preprocessing results in very inefficient queries. This is most likely due to the inefficient locality filter on these graphs (see Section 6.3.1).

	Gnutella	Epinions	Slashdot	NotreDame	CA	Wiki-Talk	Skitter	Pokec	Orkut
cons	148.26	23.56	19.09	17.32	269.39	26.49	226.79	9.44	22.84
aggr	176.90	29.53	24.77	24.10	85079.10	29.16	327.63	12.71	32.52
stal	179.51	24.90	21.40	16.02	84711.83	24.74	286.54	9.52	26.52

Table 6.8.: Average running time of the combined-exact query in μs.

6.3.3. Query Error

As for the running time, the different preprocessing variants have no significant effect on the query error. Figure 6.7 gives an overview of the query errors using the conservative preprocessing. Results for the other variants can be found in the appendix in Figure B.2 and B.3 as well as Table B.3. The error of the local query is not evaluated since it always yields the correct distance.

Our distance oracle guarantees an additive error of at most $4r$. As mentioned in Section 6.2, we use $r = 1$ for the seven smaller graphs and $r = 2$ for the Pokec and Orkut graph, therefore allowing a maximum additive error of four and eight respectively. We guarantee this maximum error for the combined-TNR and combined-exact query. The global and combined-approx query are not affected by this.

Overall, the query accuracy is very good. The vast majority of queries has an error of less than or equal to r. The average error is also less than r. This even applies to the global query which is altogether very accurate even without the locality filter. An exception is again the CA road network for which errors of up to 40 can occur (see Figure B.3). This result is reasonable since the diameter is much larger.
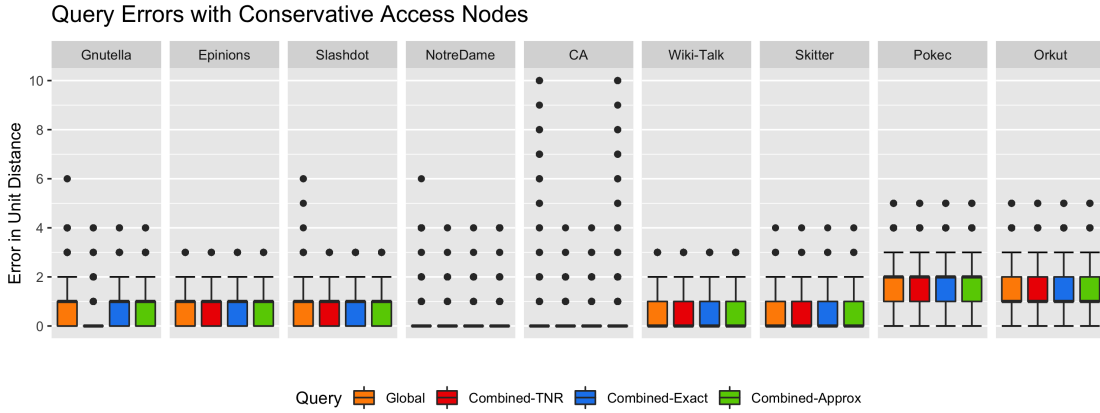
Figure 6.7.: Query error up to ten with preprocessing based on conservative access nodes.

Furthermore, the maximum error is not exceeded by the combined-TNR and -exact queries. This confirms the correctness of our distance oracle which is already proven in Section 3.4.2.

6.4. Comparison to Other Distance Oracles

We compare aTNR to both CH-TNR by Arz et al. [ALS13], and the Pruned Landmark Labeling (PLL) by Akiba et al. [AIY13]. The results for CH-TNR are measured on an "Intel Core i7-920 clocked at 2.67 GHz with four cores and 12 GiB of RAM" [ALS13]. The preprocessing is computed in parallel, but the queries are sequential. They also provide a sequential running time for their preprocessing. Akiba et al. conduct their experiments on an Intel Xeon X5670 machine clocked at 2.93 GHz with 12 cores and 48 GiB main memory [AIY13]. Both the preprocessing and the queries of PLL are sequential.

For better comparability, this section considers only the sequential running times. Furthermore, Table 6.9 provides a scaling factor of the different machines using the `int-speed` benchmark of the SPEC CPU2006 benchmark [Hen06]. This scaling factor is the factor by which the running times need to be multiplied to be comparable according to the SPEC benchmark.

	`int-speed`	Scaling Factor
aTNR	30.4 [1]	1.00
CH-TNR	29.5 [2]	0.97
PLL	42.9 [3]	1.41

Table 6.9.: Comparison of the machines used by the different distance oracles by their SPEC CPU2006 `int-speed` result.

We compare our results for the road network CA to the results obtained for CH-TNR in [ALS13]. As shown by Table 6.9, the machine used by CH-TNR has a very similar sequential performance resulting in a scaling factor of only 0.97. The road networks used in CH-TNR are about ten times larger than CA. It is also important to note that the results for CH-TNR were obtained on weighted graphs while we only evaluate unweighted graphs. For road networks, this can eradicate the underlying edge hierarchy.

---

[1] `https://www.spec.org/cpu2006/results/res2012q2/cpu2006-20120504-21457.html`

[2] `https://www.spec.org/cpu2006/results/res2010q1/cpu2006-20100215-09677.html`

[3] `https://www.spec.org/cpu2006/results/res2011q4/cpu2006-20110928-18635.html`

	PT	PS	QT	Error	SPEC Scaling
aTNR	31557.3 s	140 byte/vertex	147.89 μs	0.09	1.00
CH-TNR	1046 s	147 byte/vertex	1.38 μs	0	1.41

Table 6.10.: Comparison of aTNR and CH-TNR regarding preprocessing time (PT), preprocessing size (PS), query time (QT), and error. All running times are sequential. Additionally, the SPEC scaling factor from Table 6.9 is given.

A comparison of the results for aTNR and CH-TNR is shown in Table 6.10. Both the preprocessing and the queries of aTNR are significantly slower than CH-TNR. Our fastest preprocessing for CA – which is based on the conservative variant – has a running time of about $16\,ms$ per vertex. CH-TNR has a preprocessing time of only $58\,\mu s$ per vertex, that is almost three orders of magnitude faster than aTNR. This result is not that surprising since CH-TNR utilizes the specific properties of road networks whereas aTNR is focused on social networks. It is therefore expected to work better on road networks than aTNR. Concerning the space consumption, aTNR is slightly better with a size of only about 140 bytes per vertex compared the 147 bytes per vertex necessary for CH-TNR.

The combined queries are about three orders of magnitude slower than CH-TNR with $147.89\,\mu s$ compared to $1.38\,\mu s$. However, our global queries require only around $0.89\,\mu s$ which is slightly faster than CH-TNR. Furthermore, our locality filter correctly classifies less than 5 % of the queries as local. A large part of the disparity to CH-TNR is therefore most likely caused by the much slower local query. Our local query uses only bidirectional search as a speed-up technique, unlike CH-TNR, which uses a CH-Query. This causes our local query to be more than three orders of magnitude slower.

CH-TNR is an exact method and therefore yields exact results while aTNR results in an average error of about 0.09.

We furthermore compare our results to the Pruned Landmark Labeling (PLL) introduced by Akiba et al. [AIY13] on the shared input instances. More specifically these are the graphs Gnutella, Epinions, Slashdot, NotreDame, WikiTalk, and Skitter. Table 6.11 gives an overview of the obtained results. Note that they differ from the previous results for aTNR because both the preprocessing and the queries are computed sequentially. The conservative version of the access node determination is notably affected the most by this. Due to prohibitively long running times, not all preprocessings for Skitter are computed sequentially. Instead, the parallel running time is scaled by 30. This is the speedup measured for the preprocessings of Gnutella and CA which behave very similarly to Skitter. As shown by Table 6.9, the machine used by PLL achieves a slightly better result in the SPEC benchmark leading to a scaling factor of 1.41.

The sequential preprocessing time of aTNR depends heavily on the input. On some graphs, such as Epinions and Slashdot, the aggressive and stall-on-demand approaches for determining the access nodes can compete with PLL. Other graphs, however, require up to two orders of magnitude more time than PLL. Having said that, we do offer a parallel version of our preprocessing. But even the parallel running times, given for example by Table A.1, are not always faster than PLL's sequential algorithm. We do, however, outperform PLL regarding the space consumption of the preprocessed data with aTNR requiring less space by an order of magnitude.

The query of PLL is clearly faster and outperforms aTNR by up to two orders of magnitude. Even for those graphs for which the locality filter works efficiently, such as Wiki-Talk and NotreDame, the aTNR combined-Exact query is significantly slower than PLL's query. This result is somewhat expected since it is a 2-hop algorithm, whereas aTNR is based on a 3-hop approach. However, the disparity should not be as significant. The global query of

the graphs Epinions, Slashdot, NotreDame, and Wiki-Talk – those graphs for which the current locality filter is efficient – has an average running time of 0.24 to 0.37 μs. This is about as fast as the PLL query. It therefore seems likely that the different running time is caused to a large extent by the very slow local query. This indicates that accelerating the local query can offer a significant opportunity for improvement.

PLL – just like CH-TNR – is an exact approach whereas aTNR can only guarantee a maximum additive error of $4r$. The actual average error is, however, less than one.

		aTNR				PLL		
	CS	PT	PS	QT	Err	PT	PS	QT
Gnutella	cons	355.5 s	71.8 MB	109.93 μs	0.90			
	aggr	281.6 s	72.9 MB	110.82 μs	0.90	54 s	209 MB	5.2 μs
	stal	717.2 s	72.9 MB	98.36 μs	0.90			
Epinions	cons	69.0 s	2.5 MB	16.04 μs	0.65			
	aggr	2.9 s	4.2 MB	21.46 μs	0.65	1.7 s	32 MB	0.5 μs
	stal	2.6 s	2.7 MB	18.94 μs	0.65			
Slashdot	cons	112.7 s	3.5 MB	15.45 μs	0.74			
	aggr	5.7 s	6.9 MB	20.72 μs	0.74	6.0 s	48 MB	0.8 μs
	stal	2.6 s	4.0 MB	16.28 μs	0.74			
NotreDame	cons	1455.8 s	12.8 MB	21.75 μs	0.07			
	aggr	203.0 s	51.7 MB	35.48 μs	0.07	4.5 s	138 MB	0.5 μs
	stal	125.0 s	24.1 MB	24.68 μs	0.07			
Wiki-Talk	cons	398.2 s	66.0 MB	125.0 μs	0.48			
	aggr	196.0 s	66.0 MB	128.2 μs	0.48	61 s	1.0 GB	0.6 μs
	stal	196.4 s	66.0 MB	126.0 μs	0.48			
Skitter	cons	$\approx$ 70.9 h	282.8 MB	833.51 μs	0.45			
	aggr	$\approx$ 46.7 h	3.6 GB	1.61 ms	0.45	359 s	2.7 GB	2.3 μs
	stal	11.6 h	1.2 GB	1.22 ms	0.45			

Table 6.11.: Comparison of our aTNR algorithm, based on the different covering searches (CS), and the Pruned Landmark Labeling in terms of preprocessing time (PT), preprocessing size (PS), query time (QT) and average error (Err). Since PLL is an exact method, its average error is omitted. The preprocessing times for Skitter marked with $\approx$ are approximated from the parallel running time.

7. Conclusion

This thesis introduces an approximate distance oracle for social networks based on transit node routing. We developed two different variants of this distance oracle. Our main contribution is approximate transit node routing (aTNR) which offers a variable approximation of the distances with a parameter $r \geq 0$. It guarantees a maximum additive error of $4r$.

Approximate transit node routing works well on social networks and web graphs. The global queries – that make up a large proportion of the queries – achieve a running time of about a microsecond. The combined queries take around $20\,\mu s$. On the tested computer and road networks, the combined queries are about an order of magnitude slower. The average additive error is less than r; the maximum allowed error $4r$ is only reached by a small number of outliers. On the smaller social networks, our preprocessing only takes a few seconds. Using $r = 2$, we can even preprocess larger social networks – such as Orkut – in less than two minutes. The preprocessing results can be stored in less than 50 bytes per vertex for the tested social networks and web graphs. Other tested graphs need up to 1.1 KB per vertex which is still less than the Pruned Landmark Labeling by Akiba et al. [AIY13].

Additionally to aTNR, we theoretically describe the Voronoi search distance oracle which has potentially faster preprocessing and query times but no upper bound for the error.

7.1. Future Work

Due to the time constraints of this thesis, there are many interesting aspects we are not able to cover in this work. This section outlines several ideas, ranging from pure implementation to additional algorithmic approaches and variants of the given distance oracles, that would be interesting to pursue in the future.

Evaluation. The current comparison to CH-TNR and Pruned Landmark Labeling is based on the SPEC benchmark. A comparison on the same machine would of course be more informative. It would furthermore be interesting to compare aTNR to CH-TNR on the same graphs since CA is about ten times smaller than the instances tested by Arz et al. [ALS13].
There are also some parameters that could be investigated more thoroughly. The number of transit nodes k is set to $k = \sqrt{n}$ for all experiments in this work. Increasing it increases the size of the distance table quadratically but should reduce the access nodes and search space size. Furthermore, evaluating different values for the marking radius r on the same

instance might be interesting. There is also the limit of the local query in the approximate combined query that can be adjusted for either better approximation or faster queries.

As already mentioned in Section 5.2.1, we tried both arrays and hash tables as data structure for the vertices visited by the covering searches. We might be able to accelerate the covering searches further with a more optimized data structure based on an extensive evaluation.

Implementation. It is certainly interesting to implement and evaluate those variants of our distance oracles that are not yet covered by our implementation. First of all, the implementation can be extended to directed and weighted graphs as the current implementation can only handle unweighted and undirected graphs. Secondly, the Voronoi search distance oracle is currently only described theoretically. Implementing it and comparing it to aTNR is an interesting next step. Furthermore, the naive distance table computation could be replaced by the overlay-graph approach introduced in Section 3.1.2. This could speed up the preprocessing significantly for some graphs. For example on Wiki-Talk and Orkut, computing the distance table makes up almost the complete preprocessing time when using the aggressive or stall-on-demand covering search.

Access Node Determination. Besides computing the distance table, the access node determination has a major impact on the preprocessing time. Especially for the computer and road networks Gnutella, Skitter and CA, the covering searches require the most preprocessing time. It would therefore be useful to accelerate these searches. Using the distance table and the distances stored in the Voronoi diagram, one might be able to truncate the covering searches before even reaching a marked node. To do so, one would combine these two distances and compare them to the distance of a queued vertex to determine whether there are any transit nodes that can be visited on a shortest path via this vertex. However, this might invalidate the correctness of the current locality filter if the searches are stopped too early so that their search spaces do not intersect.

Locality Filter. The locality filter has two main aspects for improvement. First of all, it currently wrongly classifies many global queries as local on both computer and road networks. This slows the combined queries down significantly. Increasing the number of correctly recognized global queries is therefore very important for reducing the query time. The accuracy for social networks is much better but not perfect either, as some combinations have up to 20 % false local classifications. They might therefore also benefit from such improvements. Secondly, the locality filter accounts for a large proportion of the total preprocessing size. This could be improved using more approximation of the search spaces, for example with fewer, but larger Voronoi cells. Storing the search spaces with the access nodes, as for example done by Arz et al. [ALS13], could also reduce the required space. One would, however, need to consider how the combine this with our postprocessing. Another option would be to compress the search spaces. One could, for example, use a compression scheme similar to the one used in Brunel et al. [BDGW10] for compressing arc-flags, by grouping vertices with similar search spaces.

Local Query. Even though most queries are correctly classified as global, the average combined query is up to two orders of magnitude slower than the global query. This indicates that the running time of the local query is too slow. Currently, the local query is only accelerated by using a bidirectional search. The original transit node routing algorithm [BFM+07] used a multilevel approach which speeds up local queries with a recursive application of TNR. An additional larger set of second transit nodes is selected to solve local queries of the first level. The local queries of the second level could be solved using a third level of transit nodes and so on. One could probably develop a similar multilevel approach for aTNR. Furthermore, one could try to apply other speed up techniques unrelated to TNR.

Path Retrieval. Currently, both introduced distance oracles compute only the distance between two vertices. For many applications, it is, however, also required to return an actual shortest path. The original transit node routing algorithm solves this by successively determining the edges (s, u) with $\delta(s, u) + \delta(u, t) = \delta(s, t)$ and repeating the same steps with $s = u$ [BFM$^+$07, Section 3.6]. Since our distance oracles return the distances of actual paths – even though they might not be the shortest paths – it should be possible to implement the path retrieval in the same way. However, the high vertex degree in social networks probably increases the running time significantly compared to road networks. Moreover, the approximation with the marked nodes might cause problems for finding the shortest paths from a transit node t since all neighboring marked nodes are routing via t.

Bibliography

[ADGW11]   Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks. In *10th Symposium on Experimental Algorithms (SEA)*, volume 6630 of *LNCS*, pages 230–241. Springer, 2011.

[ADGW12]   Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Hierarchical Hub Labelings for Shortest Paths. In *20th European Symposium on Algorithms (ESA)*, volume 7501 of *LNCS*, pages 24–35. Springer, 2012.

[AIY13]   Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Fast Exact Shortest-Path Distance Queries on Large Networks by Pruned Landmark Labeling. In *Proceedings of the International Conference on Management of Data*, pages 349–360. ACM SIGMOD, 2013.

[AJB99]   Réka Albert, Hawoong Jeong, and Albert-László Barabási. Internet: Diameter of the World-Wide Web. *nature*, 401(6749):130–131, 1999.

[ALS13]   Julian Arz, Dennis Luxen, and Peter Sanders. Transit Node Routing Reconsidered. In *12th Symposium on Experimental Algorithms (SEA)*, volume 7933 of *LNCS*, pages 55–66. Springer, 2013.

[AV07]   David Arthur and Sergei Vassilvitskii. K-means++: The Advantages of Careful Seeding. In *Proceedings of the 8th Symposium on Discrete Algorithms (SODA)*. ACM-SIAM, 2007.

[BCKO08]   Mark de Berg, Otfried Cheong, Mark van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 3 edition, 2008.

[BDG$^+$15]   Moritz Baum, Julian Dibbelt, Andreas Gemsa, Dorothea Wagner, and Tobias Zündorf. Shortest Feasible Paths with Charging Stops for Battery Electric Vehicles. In *Proceedings of the 23rd International Conference on Advances in Geographic Information Systems*, pages 44:1–44:10. ACM SIGSPATIAL, 2015.

[BDG$^+$16]   Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F Werneck. Route Planning in Transportation Networks. In *Algorithm engineering*, volume 9220 of *LNCS*, pages 19–80. Springer, 2016.

[BDGW10]   Edith Brunel, Daniel Delling, Andreas Gemsa, and Dorothea Wagner. Space-Efficient SHARC-Routing. In *9th Symposium on Experimental Algorithms (SEA)*, volume 6049 of *LNCS*, pages 47–58. Springer, 2010.

[BDPW13]   Moritz Baum, Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. Energy-Optimal Routes for Electric Vehicles. In *Proceedings of the 21st International Conference on Advances in Geographic Information Systems*, pages 54–63. ACM SIGSPATIAL, 2013.

[BDPW16]   Moritz Baum, Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. Dynamic Time-Dependent Route Planning in Road Networks With User Preferences. In *15th Symposium on Experimental Algorithms (SEA)*, volume 9685 of *LNCS*, pages 33–49. Springer, 2016.

[BFM+07]   Holger Bast, Stefan Funke, Domagoj Matijevic, Peter Sanders, and Dominik Schultes. In Transit to Constant Shortest-Path Queries in Road Networks. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 46–59. SIAM, 2007.

[BFSS07]   Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. Fast Routing in Road Networks with Transit Nodes. *Science*, 316(5824):566, 2007.

[BGSV13]   G Veit Batz, Robert Geisberger, Peter Sanders, and Christian Vetter. Minimum Time-Dependent Travel Times with Contraction Hierarchies. *Journal of Experimental Algorithmics (JEA)*, 18:1–4, 2013.

[BK06]   S. Baswana and T. Kavitha. Faster Algorithms for Approximate Distance Oracles and All-Pairs Small Stretch Paths. In *47th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 591–602. IEEE, 2006.

[Bra01]   Ulrik Brandes. A Faster Algorithm for Betweenness Centrality. *The Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[BS06]   Surender Baswana and Sandeep Sen. Approximate Distance Oracles for Unweighted Graphs in Expected $\mathcal{O}(n^2)$ Time. *Transactions on Algorithms (TALG)*, 2(4):557–577, 2006.

[CHKZ03]   Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and Distance Queries via 2-Hop Labels. *Journal on Computing*, 32(5):1338–1355, 2003.

[CLRS09]   Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3 edition, 2009.

[Dij59]   Edsger W Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[DKP12]   Daniel Delling, Bastian Katz, and Thomas Pajor. Parallel Computation of Best Connections in Public Transportation Networks. *Journal of Experimental Algorithmics (JEA)*, 17:4, 2012.

[DSGNP10]   Atish Das Sarma, Sreenivas Gollapudi, Marc Najork, and Rina Panigrahy. A Sketch-Based Distance Oracle for Web-Scale Graphs. In *Proceedings of the 3rd Conference on Web Search and Data Mining (WSDM)*, pages 401–410. ACM, 2010.

[DSSW09]   Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering Route Planning Algorithms. In *Algorithmics of Large and Complex Networks*, volume 5515 of *LNCS*, pages 117–139. Springer, 2009.

[EFS11]   Jochen Eisner, Stefan Funke, and Sabine Storandt. Optimal Route Planning for Electric Vehicles in Large Networks. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence*, pages 1108–1113, 2011.

[Erw00]   Martin Erwig. The Graph Voronoi Diagram with Applications. *Networks: An International Journal*, 36(3):156–163, 2000.

[Flo62]   Robert W Floyd. Algorithm 97: Shortest Path. *Communications of the ACM*, 5(6):345, 1962.

[Fre77]   Linton C Freeman. A Set of Measures of Centrality Based on Betweenness. *Sociometry*, pages 35–41, 1977.

[FT87]   Michael L Fredman and Robert Endre Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[GBSW10]   Andrey Gubichev, Srikanta Bedathur, Stephan Seufert, and Gerhard Weikum. Fast and Accurate Estimation of Shortest Paths in Large Graphs. In *Proceedings of the 19th Conference on Information and Knowledge Management (CIKM)*, pages 499–508. ACM, 2010.

[GPPR04]   Cyril Gavoille, David Peleg, Stéphane Pérennes, and Ran Raz. Distance Labeling in Graphs. *Journal of Algorithms*, 53(1):85–112, 2004.

[GSSD08]   Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *7th Workshop on Experimental and Efficient Algorithms (WEA)*, volume 5038 of *LNCS*, pages 319–333. Springer, 2008.

[GSSV12]   Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transportation Science*, 46(3):388–404, 2012.

[Hen06]   John L. Henning. SPEC CPU 2006 Benchmark Descriptions. *Computer Architecture News*, 34(4):1–17, 2006.

[Jai10]   Anil K Jain. Data Clustering: 50 Years Beyond K-Means. *Pattern Recognition Letters*, 31(8):651–666, 2010.

[Joh77]   Donald B. Johnson. Efficient Algorithms for Shortest Paths in Sparse Networks. *Journal of the ACM (JACM)*, 24(1):1–13, 1977.

[JRXL12]   Ruoming Jin, Ning Ruan, Yang Xiang, and Victor Lee. A Highway-Centric Labeling Approach for Answering Distance Queries on Large Sparse Graphs. In *Proceedings of the International Conference on Management of Data*, pages 445–456. ACM SIGMOD, 2012.

[KV18]   Bernhard Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer, 6 edition, 2018.

[Lei10]   Charles E Leiserson. The Cilk++ Concurrency Platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.

[LHK10a]   Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. Predicting Positive and Negative Links in Online Social Networks. In *Proceedings of the 19th Conference on World Wide Web (WWW)*, pages 641–650. ACM, 2010.

[LHK10b]   Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. Signed Networks in Social Media. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*, pages 1361–1370. ACM SIGCHI, 2010.

[LK14]   Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data, 2014.

[LKF05]   Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs Over Time: Densification Laws, Shrinking Diameters and Possible Explanations. In *Proceedings of the 11th Conference on Knowledge Discovery in Data Mining (KDD)*, pages 177–187. ACM, 2005.

[LKF07]   Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph Evolution: Densification and Shrinking Diameters. *Transactions on Knowledge Discovery from Data (TKDD)*, 1(1), 2007.

[LLDM09]   Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[Meh88]   Kurt Mehlhorn. A Faster Approximation Algorithm for the Steiner Problem in Graphs. *Information Processing Letters*, 27(3):125–128, 1988.

[MMG+07]   Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and Analysis of Online Social Networks. In *Proceedings of the Internet Measurement Conference (IMC)*. ACM, 2007.

[MS08]   Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox.* Springer, 2008.

[MSD16]   Tobias Maier, Peter Sanders, and Roman Dementiev. Concurrent Hash Tables: Fast and General?(!). *Proceedings of the 21st Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2016.

[MSS15]   Henning Meyerhenke, Peter Sanders, and Christian Schulz. Parallel Graph Partitioning for Complex Networks. *29th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1055–1064, 2015.

[PBCG09]   Michalis Potamias, Francesco Bonchi, Carlos Castillo, and Aristides Gionis. Fast Shortest Path Distance Estimation in Large Networks. In *Proceedings of the 18th Conference on Information and Knowledge Management (CIKM)*, pages 867–876. ACM, 2009.

[PR10]   Mihai Pătraşcu and Liam Roditty. Distance Oracles Beyond the Thorup-Zwick Bound. In *51st Symposium on Foundations of Computer Science (FOCS)*, pages 815–823. IEEE, 2010.

[QXSW13]   Zichao Qi, Yanghua Xiao, Bin Shao, and Haixun Wang. Toward a Distance Oracle for Billion-Node Graphs. *Proceedings of the VLDB Endowment*, 7(1):61–72, 2013.

[RAD03]   Matthew Richardson, Rakesh Agrawal, and Pedro Domingos. Trust Management for the Semantic Web. In *International Semantic Web Conference*, volume 2870 of *LNCS*, pages 351–368. Springer, 2003.

[RFI02]   Matei Ripeanu, Ian Foster, and Adriana Iamnitchi. Mapping the Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems and Implications for System Design. *Computing Research Repository (CoRR)*, 2002.

[SB13]   Julian Shun and Guy E. Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 135–146. ACM, 2013.

[SDB15]   Julian Shun, Laxman Dhulipala, and Guy E Blelloch. Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+. In *Data Compression Conference (DCC)*, pages 403–412. IEEE, 2015.

[SS07] Dominik Schultes and Peter Sanders. Dynamic Highway-Node Routing. In *6th Workshop on Experimental and Efficient Algorithms (WEA)*, volume 4525 of *LNCS*, pages 66–79. Springer, 2007.

[Str17] Ben Strasser. Dynamic Time-Dependent Routing in Road Networks Through Sampling. In *17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS)*, volume 59, pages 3:1–3:17. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017.

[TACGB+11] Konstantin Tretyakov, Abel Armas-Cervantes, Luciano García-Bañuelos, Jaak Vilo, and Marlon Dumas. Fast Fully Dynamic Landmark-Based Estimation of Shortest Path Distances in Very Large Graphs. In *Proceedings of the 20th Conference on Information and Knowledge Management (CIKM)*, pages 1785–1794. ACM, 2011.

[TZ05] Mikkel Thorup and Uri Zwick. Approximate Distance Oracles. *Journal of the ACM (JACM)*, 52(1):1–24, 2005.

[TZ12] Lubos Takac and Michal Zabovsky. Data Analysis in Public Social Networks. In *International Scientific Conference and International Workshop Present Day Trends of Innovations*, volume 1, 2012.

[UCDG08] Antti Ukkonen, Carlos Castillo, Debora Donato, and Aristides Gionis. Searching the Wikipedia with Contextual Information. In *Proceedings of the 17th Conference on Information and Knowledge Management (CIKM)*, pages 1351–1352. ACM, 2008.

[VFD+07] Monique V Vieira, Bruno M Fonseca, Rodrigo Damazio, Paulo B Golgher, Davi de Castro Reis, and Berthier Ribeiro-Neto. Efficient Search Ranking in Social Networks. In *Proceedings of the 16th Conference on Information and Knowledge Management (CIKM)*, pages 563–572. ACM, 2007.

[War62] Stephen Warshall. A Theorem on Boolean Matrices. *Journal of the ACM (JACM)*, 9(1):11–12, 1962.

[WXD+12] Lingkun Wu, Xiaokui Xiao, Dingxiong Deng, Gao Cong, Andy Diwen Zhu, and Shuigeng Zhou. Shortest Path and Distance Queries on Road Networks: An Experimental Evaluation. *Proceedings of the VLDB Endowment*, 5(5):406–417, 2012.

[YBLS08] Sihem Amer Yahia, Michael Benedikt, Laks VS Lakshmanan, and Julia Stoyanovich. Efficient Network Aware Search in Collaborative Tagging Sites. *Proceedings of the VLDB Endowment*, 1(1):710–721, 2008.

[YL15] Jaewon Yang and Jure Leskovec. Defining and Evaluating Network Communities Based on Ground-Truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

Appendix

A. Preprocessing Evaluation

Graph	CS	Time per Preprocessing Step in s				
		Transit Nodes	Distance Table	Access Nodes	Locality Filter	Postpro-cessing
Gnutella	cons	0.001	0.716	11.030	0.140	0.340
	aggr	0.001	0.716	7.750	0.180	0.320
	stal	0.001	0.716	47.180	0.160	0.320
Epinions	cons	0.001	0.768	2.380	0.010	0.010
	aggr	0.001	0.768	0.190	0.010	0.020
	stal	0.001	0.768	0.250	0.000	0.000
Slashdot	cons	0.001	0.759	3.930	0.010	0.010
	aggr	0.001	0.759	0.340	0.010	0.030
	stal	0.001	0.759	0.230	0.000	0.010
NotreDame	cons	0.002	4.681	48.850	0.020	0.030
	aggr	0.002	4.681	6.610	0.070	0.310
	stal	0.002	4.681	7.130	0.030	0.110
CA	cons	0.015	26.945	1029.640	0.410	0.160
	aggr	0.015	26.945	10587.200	72.330	260.730
	stal	0.015	26.945	13783.100	33.140	243.090
Wiki-Talk	cons	0.015	26.945	28.130	0.110	0.030
	aggr	0.015	26.945	1.220	0.150	0.040
	stal	0.015	26.945	1.360	0.140	0.040
Skitter	cons	0.009	28.142	8473.440	0.230	1.170
	aggr	0.009	28.142	5484.610	17.310	74.370
	stal	0.009	28.142	2108.410	1.260	15.300
Pokec	cons	0.011	36.251	305.310	0.120	0.030
	aggr	0.011	36.251	0.440	0.060	0.040
	stal	0.011	36.251	0.620	0.080	0.030
Orkut	cons	0.018	107.035	199.410	0.140	0.040
	aggr	0.018	107.035	0.290	0.120	0.040
	stal	0.018	107.035	0.330	0.120	0.030

Table A.1.: Overview of the running time for the different preprocessing steps and prepro-cessing variants.

B. Query Evaluation

Graph	CS	Average Query Time in μs				
		Local	Global	CombTNR	CombExact	CombApprox
Gnutella $r = 1$	cons	287.95	17.57	301.22	148.26	153.83
	aggr	303.46	14.52	294.32	176.90	137.90
	stal	317.41	14.55	299.11	179.51	143.49
Epinions $r = 1$	cons	250.17	0.27	40.27	23.56	22.59
	aggr	259.68	0.27	53.17	29.53	28.04
	stal	253.70	0.25	43.18	24.90	24.22
Slashdot $r = 1$	cons	290.18	0.41	35.66	19.09	18.63
	aggr	295.00	0.36	49.96	24.77	24.74
	stal	294.66	0.41	40.53	21.40	20.71
NotreDame $r = 1$	cons	1708.63	0.24	24.52	17.32	17.54
	aggr	1669.98	0.29	43.32	24.10	22.95
	stal	1667.52	0.29	27.19	16.02	15.62
CA $r = 1$	cons	85889.26	1.08	300.08	269.39	23.98
	aggr	89256.09	4.44	86565.26	85079.10	562.16
	stal	89287.74	8.64	86036.21	84711.83	435.96
Wiki-Talk $r = 1$	cons	796.82	1.64	29.97	26.49	25.80
	aggr	814.98	3.12	32.63	29.16	28.96
	stal	795.26	1.45	31.86	24.74	24.59
Skitter $r = 1$	cons	988.95	1.67	396.99	226.79	207.20
	aggr	1018.32	0.74	628.17	327.63	307.41
	stal	1098.90	1.79	600.92	286.54	197.99
Pokec $r = 1$	cons	835.28	0.49	20.83	9.44	8.71
	aggr	795.17	0.50	19.30	12.71	11.58
	stal	765.90	0.43	16.50	9.52	10.25
Orkut $r = 1$	cons	2152.54	0.50	31.31	22.84	22.88
	aggr	2103.65	0.60	43.55	32.52	32.97
	stal	2151.74	0.52	37.92	26.52	28.42

Table B.2.: The average running time of the different queries sampled over 1000 random vertex pairs from each distance. A Boxplot of the same experiment is given in Figure B.1. CS denotes the variants of the preprocessing based on the different covering searches. CombTNR, CombExact, and CombApprox are the three variants of the combined query introduced in Section 3.2.3.

Query Times with Conservative Access Nodes

Query Times with Aggressive Access Nodes
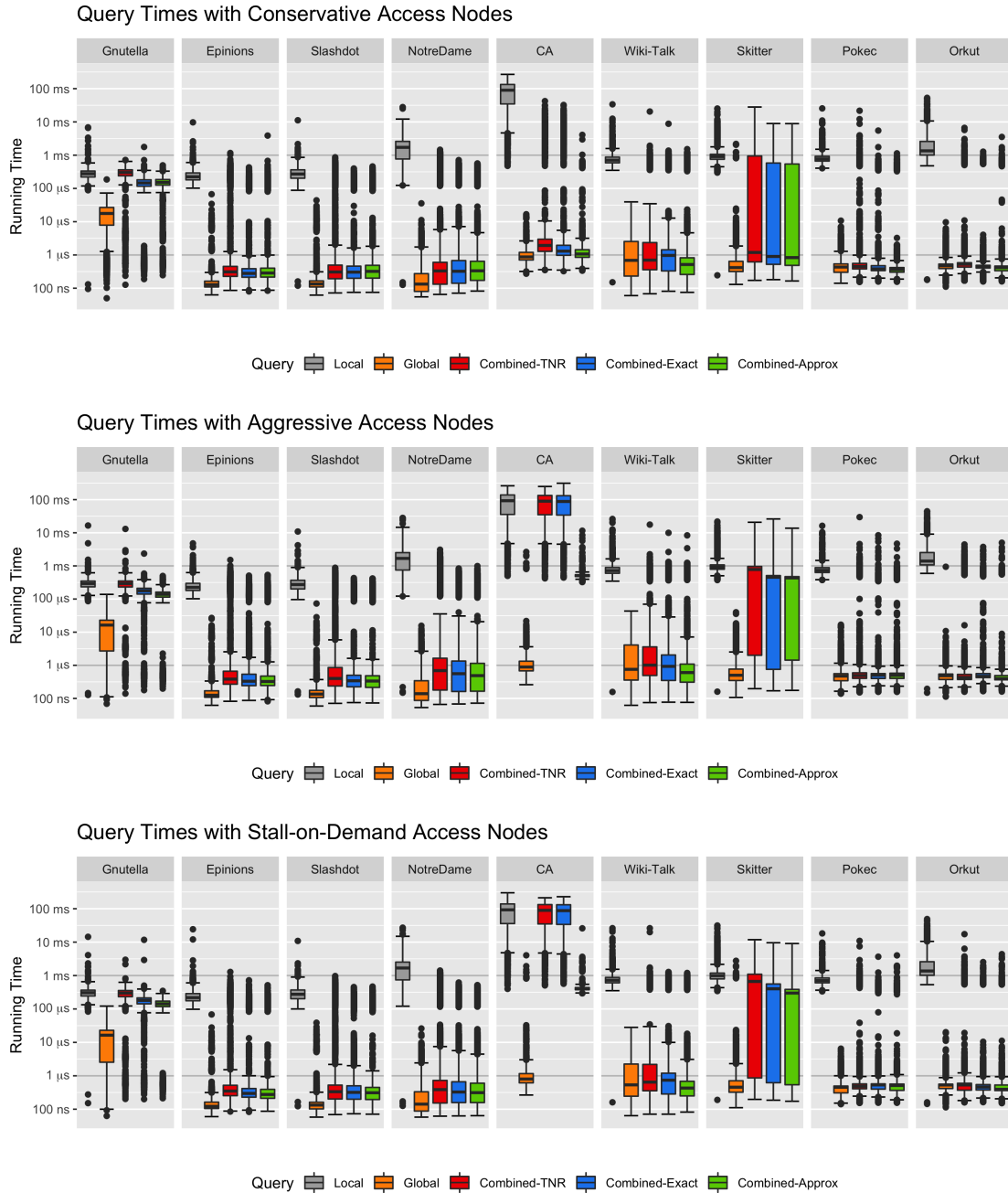

Query Times with Stall-on-Demand Access Nodes

Figure B.1.: The running time of the different queries and graphs sampled over 1000 random vertex pairs from each distance. One plot for each variant of the preprocessing based on the different covering searches.
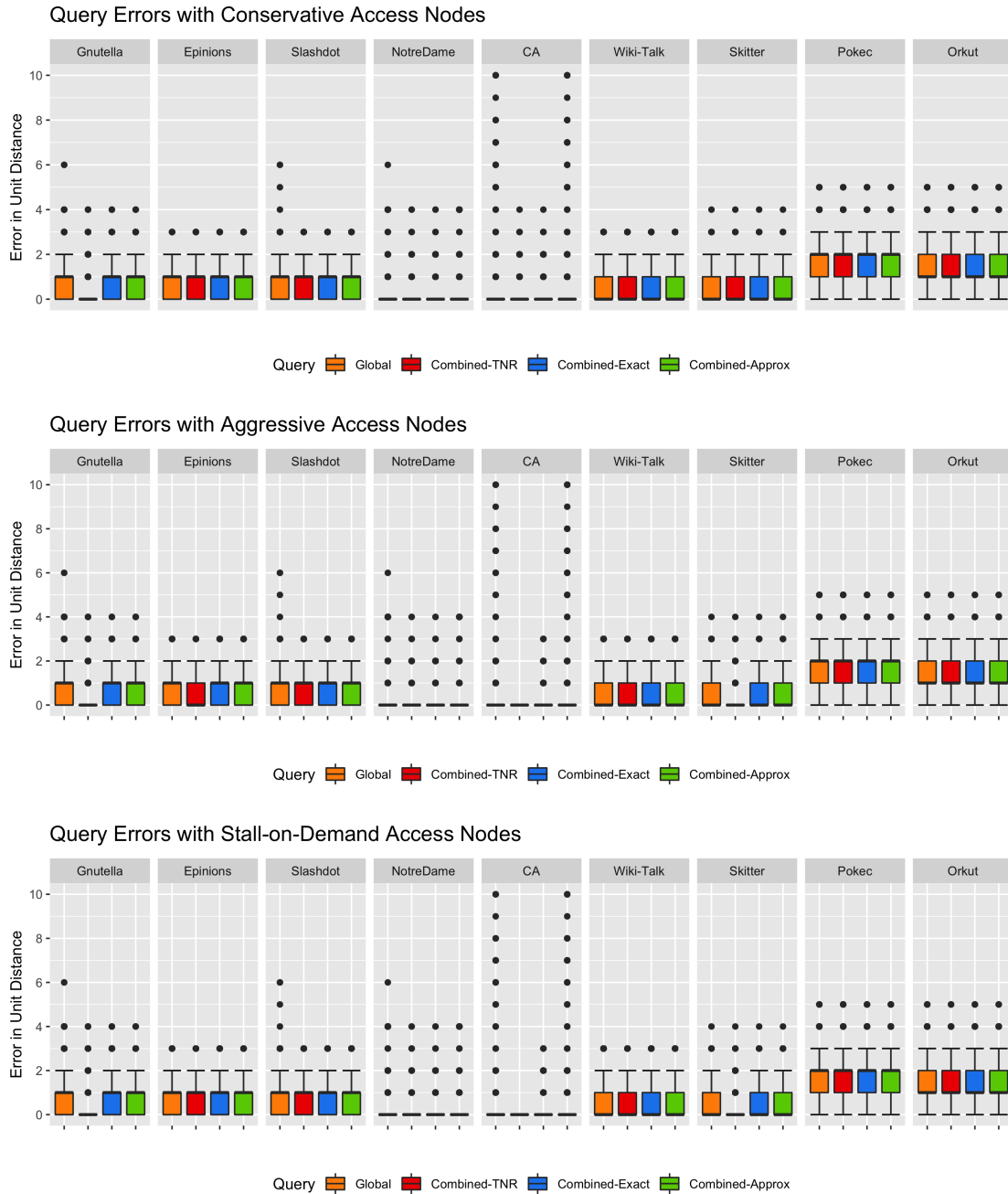
Figure B.2.: The error of the different queries and graphs sampled over 1000 random vertex pairs from each distance. One plot for each variant of the preprocessing based on the different covering searches.

The first seven graphs are processed with $r = 1$. We thus guarantee an error of four or less for the combined-TNR and combined-exact queries. Pokec and Orkut are processed with $r = 2$; the allowed error is therefore eight.

On the CA road network, outliers with an error of up to 40 exist due to the larger diameter. Those outliers are displayed in Figure B.3.
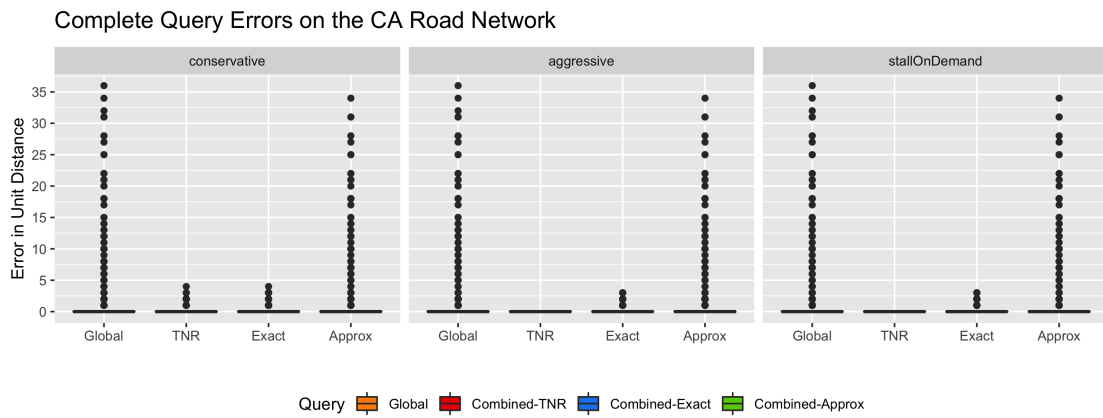
Figure B.3.: The complete error of the different queries on the CA road network containing all outliers, including those omitted in Figure B.2. The allowed error of four is only exceeded for the Global and the combined-approx queries. Our guarantee of an additive error of at most $4 \cdot r$ is therefore not exceeded.

Graph	CS	Average Error			
		Global	CombTNR	CombExact	CombApprox
Gnutella $r = 1$	cons	0.91	0.05	0.90	0.90
	aggr	0.91	0.04	0.90	0.90
	stal	0.91	0.04	0.90	0.90
Epinions $r = 1$	cons	0.65	0.59	0.65	0.65
	aggr	0.65	0.56	0.65	0.65
	stal	0.65	0.58	0.65	0.65
Slashdot $r = 1$	cons	0.75	0.67	0.74	0.74
	aggr	0.75	0.63	0.74	0.74
	stal	0.75	0.66	0.74	0.74
NotreDame $r = 1$	cons	0.11	0.06	0.07	0.08
	aggr	0.11	0.06	0.07	0.08
	stal	0.11	0.06	0.07	0.08
CA $r = 1$	cons	0.21	0.08	0.09	0.19
	aggr	0.20	0.00	0.08	0.18
	stal	0.21	0.00	0.10	0.20
Wiki-Talk $r = 1$	cons	0.50	0.48	0.48	0.48
	aggr	0.50	0.48	0.48	0.48
	stal	0.50	0.48	0.48	0.48
Skitter $r = 1$	cons	0.45	0.29	0.45	0.45
	aggr	0.45	0.16	0.45	0.45
	stal	0.45	0.23	0.45	0.45
Pokec $r = 2$	cons	1.64	1.63	1.64	1.64
	aggr	1.64	1.63	1.64	1.64
	stal	1.64	1.63	1.64	1.64
Orkut $r = 2$	cons	1.38	1.31	1.35	1.35
	aggr	1.38	1.29	1.34	1.34
	stal	1.38	1.30	1.34	1.34

Table B.3.: The average error of the different queries sampled over 1000 random vertex pairs from each distance. A Boxplot of the same experiment is given in Figure B.2. The local query is always correct and thus has an average error of 0. CS denotes the variants of the preprocessing based on the different covering searches.