Complexity Analysis for a Labeled Routing Scheme

Nils Goldmann

March 22, 2021

First Reviewer: M. Ed. Max Willert

Second Reviewer: Prof. Dr. Wolfgang Mulzer

Selbstständigkeitserklärung

Ich erkläre gegenüber der Freien Universität Berlin, dass ich die vorliegende Bachelorarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.

Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Schriften entnommen sind, habe ich als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch bei keiner anderen Universität als Prüfungsleistung eingereicht.

Nils Goldmann
Berlin, March 22, 2021

Abstract

Let us consider $\varepsilon \in (0,1)$ and an undirected, weighted graph G with n nodes and low doubling dimension α. Konjevod, Richa and Xia introduced a $(1+\varepsilon)$-stretch labeled routing scheme with $\lceil \log n \rceil$-bit routing labels, $O\left(\frac{\log^2 n}{\log \log n}\right)$-bit packet headers and $\left(\frac{1}{\varepsilon}\right)^{\mathcal{O}(\alpha)} \log^3 n$-bit routing tables at every node for such graphs G [KRX16]. Now, we want to analyze the preprocessing step of this method. We show that the running time for this is $\mathcal{O}(n^3 + \varepsilon^{-1} \cdot n^2 \log^2 n)$. Also, we adapt the method for Unit Disk Graphs and improve the running time to be $\mathcal{O}(\varepsilon^{-1} \cdot n^2 \log^2 n)$. We show that the doubling dimension for Unit Disk Graphs is $\mathcal{O}(\max(1, D^2))$, with D being the diameter of the graph. Additionally, we simplify Konjevod et al.'s routing algorithm and transform it into a recursive form.

Acknowlegements

First and foremost, I would like to thank my supervisor Max Willert for everything: From introducing me to routing in the very beginning up until my final draft, he provided me with all the support I could ask for. That includes countless video calls (often even on short notice), answering all of my stupid and not-so-stupid questions, offering helpful advice for pretty much every aspect of this work as well as giving me a little push to keep up my work at the right times. This thesis would not have been possible without him.

Also, I would like to thank Wolfgang Mulzer for presenting me with very worthwhile current subjects of research, including this one, and awakening my interest in routing.

I am very grateful for the help and support of my partner Carola. She did not only read my overcomplicated, abstract verbiage and helped me making it somewhat understandable, but also encouraged me when my motivation was lacking and provided emotional support for the whole time. Thank you so much!

This work was improved a good deal by the contributions of my roommate Leonard, especially by his vast knowledge about LaTeX and typography in general. He also helped me to improve my general workflow by quite a bit.

I would like to thank my father Kristian for providing great feedback, while also helping me not to suffer from tunnel vision on details too much.

Last but not least, I want to thank my friends and my family, and especially my roommates for their interest in my work, but also for distracting me at the right moments. It was great to always be able to cook, play and chat with you or to complain about everything.

Contents

1 Introduction

In this thesis, we are dealing with the concept of *routing*. We consider routing to be a network or graph problem: We want to transmit data packets between arbitrary source and destination nodes in the graph efficiently. This is a fundamental problem of network communication (see also [Gav01], [GT08]), e.g. the data traffic via the Internet consists of data packets which have to be transmitted between different devices.

To realize efficient routing in a given graph, we use a *routing scheme*, that is, a strategy for routing which involves two major steps: The *preprocessing* and the *routing algorithm*. During the preprocessing, we inspect the graph and construct *routing tables* and *routing labels* at all nodes. Routing tables are data structures containing information for routing. Routing labels are replacements for the nodes' original names, which could include useful information. The routing algorithm performs the actual routing. The input is a data packet with a packet header and the name of its destination node. We start at a given source node and apply the algorithm: In every step, we decide which adjacent node to visit next, using the information from the packet header as well as the current node's label and routing table. We repeat this procedure until we arrive at the destination node. If necessary, we can use the header for storing additional information on the way.

The naive way to approach routing would be to calculate all shortest paths in the graph during preprocessing. Knowing all shortest paths, we could store next-hop-information for every possible destination node in each routing table: This way, the routing algorithm would just need to look up the next-hop-information at the current node and could always take the shortest path. However, this approach is completely impractical for big graphs, since it requires every node to store $\Theta(n \log n)$ information for the next hop, with n being the number of nodes in the graph. The total required memory would be $\Theta(n^2 \log n)$, which is not feasible for large n.

Since our storage capacities are limited in reality, we cannot always route along the shortest path. Instead, we have to find a tradeoff between the length of the routing path and the size of routing tables and packet headers: Storing less information might increase the length of the routing path. In many cases, we want the storage requirements to be polylogarithmic in the size of nodes in the graph. We call such a routing scheme *compact*. To judge the quality of the resulting routing paths, we use a criterium called the *stretch* of the routing scheme. The stretch is the maximum ratio between the lengths of the computed routing path from u to v and the shortest possible path from u to v, considering all possible source/destination-pairs $u, v \in V$.

This tradeoff between quality of the routing path and the space requirements has been described by Peleg and Upfal [PU89]. They showed that obtaining a k-stretch, $k \geq 1$, for routing in a general graph requires $\Omega\left(n^{1+\frac{1}{2k+4}}\right)$ bits overall. However, better results have been achieved for routing in more specific graph classes. For example, planar graphs have been researched on by different authors, e.g. [BM86], [Tho04], [GT08]. The results of Thorup [Tho04] show that planar graphs allow routing with $(1+\varepsilon)$-stretch for any $\varepsilon > 0$ with a polylogarithmic number of bits per node. Also, routing in trees has been investigated, most notably by Fraigniaud and Gavoille [FG01] as well as Thorup and Zwick [TZ01]. It has been shown that we can route along the shortest paths of trees while only requiring $\mathcal{O}(\log n)$ bits for headers as well as labels and routing tables at every node.

A graph class we consider is graphs with low *doubling dimension*. The doubling dimension is defined as the least value α such that every ball of radius r in the graph can be covered by at most 2^α balls with radius $\frac{r}{2}$. A ball with radius r is a node set consisting of all nodes within distance r of a given center node. Graphs with low doubling dimension are of special interest because they are a generalization of growth-bounded graphs. Growth-bounded graphs provide a good approximation for real-world communication networks like the Internet [KRX16].

Unit Disk Graphs (UDGs) are another class of graphs we consider. In a UDG, two nodes are connected by an edge if and only if their Euclidean distance is at most one. Among other applications, UDGs are of special interest for routing in wireless networks: Suppose every node in the UDG is a communication device capable of sending and receiving signals within a fixed range. This property is reflected by the edges, i.e. two devices can only communicate with each other if their distance is at most one. For further reading on UDGs in general, we recommend the work of Clark, Colbourn and Johnson [CCJ90].

Ideally, the efficiency of a routing scheme should not depend on the normalized diameter Δ of the graph, which is the diameter D divided by the length of the shortest path. D is the maximum length of any shortest path in the graph. For a given set of nodes and edges, the (normalized) diameter could be arbitrarily big even for a small number of nodes. If the space requirements are not dependent on Δ, we call our routing scheme *scale-free*.

In this work, we focus on a scale-free, compact $(1+\varepsilon)$-stretch labeled routing scheme described by Konjvod, Richa and Xia [KRX16] for graphs with low doubling dimension. They proved that this scheme requires $\mathcal{O}(\log n)$ bits per routing label, $\mathcal{O}\left(\frac{\log^2 n}{\log\log n}\right)$ bits for packet headers and $\left(\frac{1}{\varepsilon}\right)^{\mathcal{O}(\alpha)} \log^3 n$ bits of routing information at each node. Their work included a detailed analysis for those space requirements. Also, they described the necessary data structures to retrieve those information during preprocessing. However, it remains unclear how to actually perform the preprocessing step and what its time complexity is. Necessarily, our work is largely based on Konjevod et al. and most of our definitions are adopted directly or changed slightly.

We first introduce some basic definitions and make a few assumptions for our input graphs. Additionally, we show an upper bound of $\mathcal{O}(D^2)$ for the doubling dimension of UDGs, with D being the diameter of the graph. We continue by explaining our method for calculating the preprocessing steps and analyzing their time complexity. Overall, this is $\mathcal{O}(n^3 + \varepsilon^{-1} n^2 \log^2 n)$ time for graphs with low doubling dimension. As a special class of input graphs for the routing problem, we consider UDGs: We adapt our preprocessing method and conclude an improved running time of $\mathcal{O}(\varepsilon^{-1} \cdot n^2 \log^2 n)$. Also, we transform the routing algorithm into a recursive form which does not require any global data structures. Finally, we conclude our results and take a look at further possible improvements.

2 Preliminaries

2.1 General definitions

As already mentioned, we consider our networks for routing to be graphs. In the following, let $G = (V, E)$ be a simple, connected, edge-weighted, undirected graph with n nodes and shortest-path metric d. We assume that G has the following properties:

(i) The edge with the minimal weight in E has weight 1. There might be multiple such edges.

(ii) $n = 2^k$ for some $k \in \mathbb{N}$, i.e., the number of nodes is a power of two.

(iii) The length of the longest shortest path (i.e. the *diameter* D) is a power of two.

(iv) All lengths of shortest paths are different, meaning that for all $s, t, u, v \in V$, we have
$$\{s, t\} \neq \{u, v\} \Rightarrow d(s, t) \neq d(u, v)$$

The *normalized diameter* Δ is defined as the diameter D divided by the length of the shortest path. The first property of G implies that the normalized diameter equals the diameter of G.

Let us consider this example graph G_1:

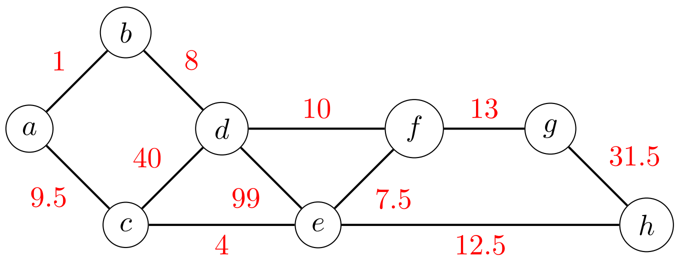


Figure 1: Example graph G_1

G_1 is an undirected weighted graph with $n = 8$ and $\Delta = 32$. We use G_1 to demonstrate how different data structures behave.

An important input value is $\varepsilon \in (0, 1)$. We already mentioned that the routing scheme should achieve a $(1+\varepsilon)$-stretch, so many of our computations will depend on ε. We assume that $\varepsilon^{-1} \leq n$.

We already described the *routing scheme* to be a strategy for routing: It involves constructing routing tables and packet headers as well as assigning labels during the preprocessing phase, and recursively routing from a source node to a destination node during the routing algorithm. More formally, a routing scheme can be defined as follows:

Definition 1 (Routing Scheme)**.** *Let $\mathcal{G}$ be a graph class. A* routing scheme *for $\mathcal{G}$ consists of the following components:*

(i) A family of label functions $l_G : V_G \to \{0, 1\}^*$. *A single label function assigns routing labels to all nodes of a given graph G. The routing labels are represented by bitstrings.*

(ii) A family of table functions $\text{tab}_G : V_G \to \{0, 1\}^*$. *A single table function constructs routing tables for every node of a given graph G. The routing tables are represented by bitstrings as well.*

(iii) A family of routing function $a_G : V_G \times \{0, 1\}^* \times \{0, 1\}^* \to V_G \times \{0, 1\}^*$. *A single routing function receives a node $w \in V_G$, a destination node's routing label and a packet header h, decides which node $w' \in V_G$ to visit next and outputs w' along with the (potentially modified) header. Again, the packet headers are represented by bitstrings.*

This definition allows us to use different routing schemes for different graph classes. For example, we may want to use the same way of constructing routing tables (i.e., a certain family of table functions), routing labels and executing the routing for all trees. On the other hand, we might want to use a completely different routing scheme for graphs with low doubling dimension.

One of the key structure elements we use in our method is the so-called *r-net*:

Definition 2 (r-net)**.** *Let $r \in \mathbb{R}$. An* r-net *in a graph $G = (V, E)$ is a subset $Y \subseteq V$ with the following properties:*

(i) Any node $u \in V$ has distance $d(u, Y) \leq r$, with $d(u, Y) = \min\{d(u, v) \mid v \in Y\}$.

(ii) For any two nodes u, v in Y, we have $d(u, v) \geq r$.

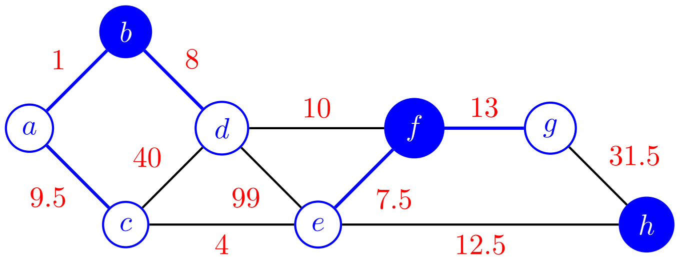


Figure 2: $\{b, f, h\}$ is a 15-net for G_1

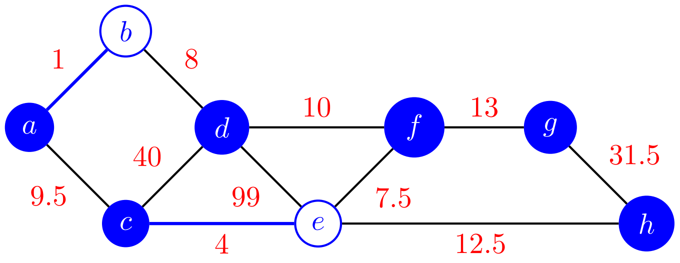


Figure 3: $\{a, c, d, f, g, h\}$ is a 5-net for G_1

As we can see from the examples, r-nets are smaller subsets of V, while still all nodes remain in a guaranteed distance of at most r. We can use multiple r-nets with different values for r to create some sort of hierarchy in G: The net with the highest value for r can be considered the first in the hierarchy, since we have a relatively small set of nodes and can guarantee any node to be within distance r. For decreasing values for r, we can reach nodes more easily (i.e. the guaranteed distance gets smaller) while requiring a bigger set to cover everything.

One such hierarchy we consider in particular involves increasing powers of two for r, up to the point where the net consists of just a single node v. We can guarantee this to happen for $r = \Delta$. Thus, for a given graph G, we construct $r = 2^i$-nets Y_i, with $i \in \{0, 1, \ldots, \log \Delta\}$.

We have:

$$Y_{\log \Delta} \subseteq Y_{\log \Delta - 1} \subseteq \ldots \subseteq Y_1 \subseteq Y_0 = V$$

It is possible to construct a 2^0-net $Y_0 \subsetneq V$. However, we require $Y_0 = V$ for adding labels to all nodes in one of our steps.

For our example graph G_1, we have $\Delta = 32$ and thus $\log \Delta = 5$. One possibility to construct 2^i-nets Y_i for $i \in \{0, 1, \ldots, 5\}$ would be

- $Y_5 = \{g\}$
- $Y_4 = \{a, g, h\}$
- $Y_3 = \{a, c, d, f, g, h\}$
- $Y_2 = \{a, c, d, f, g, h\}$
- $Y_1 = \{a, c, d, e, f, g, h\}$
- $Y_0 = \{a, b, c, d, e, f, g, h\}$

Keep in mind that this sequence is not unique by any means. For example, there are already $n = 8$ possible choices of the very first node for Y_5.

Throughout this paper, we are dealing with many sets of natural numbers $A = \{0, 1, \ldots, k\}$, for which we use the notation $A = [k]$. In the example above we could also write $i \in [5]$ instead of $i \in \{0, 1, \ldots, 5\}$.

Based on the definition for r-nets, we can define *zooming sequences* for the individual nodes:

Definition 3 (Zooming Sequence)**.** *Let $u \in V$ be a node and Y a sequence of 2^i-nets Y_i for $i \in [\log \Delta]$. The* zooming sequence *for u is a sequence of nodes*

$$u(0), u(1), \ldots, u(\log \Delta) \in V$$

which is defined recursively

1. *$u(0) = u$ and*
2. *$u(i) = v$, where v is the nearest neighbour to $u(i-1)$ in Y_i.*

The zooming sequences describe a way to reach the highest-level node in $Y_{\log\Delta}$ from any node with no more than $\log\Delta$ steps. Later on, we use this for defining a data structure called *netting tree* which reflects the hierarchy given by the 2^i-nets.

For G_1 and the 2^i-nets from the previous example, we have the following zooming sequences:

- a: a, a, a, a, a, g
- b: b, a, a, a, a, g
- c: c, c, c, c, a, g
- d: d, d, d, d, a, g
- e: e, e, c, c, a, g
- f: f, f, f, f, g, g
- g: g, g, g, g, g, g
- h: h, h, h, h, h, g

Balls are another structure of great importance for the upcoming method:

Definition 4 (Ball). *Let $B_u(r) := \{v \in V \mid d(u, v) \leq r\}$. We call $B_u(r)$* a ball of radius r centered at *u.*

We also can describe balls by their number of nodes and their center, thus determining the smallest possible radius:

Definition 5. *Let $u \in V$ and $j \in [\log n]$. Then, $r_u(j)$ is the smallest possible radius of a ball centered at u with 2^j nodes.*

One of our central applications for balls is subdividing our graph into smaller subgraphs which all have the same amount of nodes. A very useful property for this is the *packing lemma*:

Lemma 1. *Let $j \in [\log n]$. There exists a* ball packing *$\mathcal{B}_j$ of G, which is a maximum set of non-intersecting balls such that*

(i) For any ball $B \in \mathcal{B}_j$: $|B| = 2^j$.

(ii) For any node $u \in V$, there exists a ball $B \in \mathcal{B}_j$ centered at $c \in V$ such that the radius of B is at most $r_u(j)$ and $d(u, c) \leq 2r_u(j)$.

One important thing to keep in mind when dealing with ball packing is that a ball packing $\mathcal{B}_j, j \in [\log n]$ does not necessarily cover all nodes. We always have

$$\dot{\bigcup_{B \in \mathcal{B}_j}} B \subseteq V$$

with equality if and only if the number of balls is $\frac{n}{2^j}$. Similarly, we have

Lemma 2. *The total number of balls in any ball packing $\mathcal{B}_j, j \in [\log n]$ is no more than $\frac{n}{2^j}$.*

2.2 Unit Disk Graphs

A special class of graphs we deal with are the so-called *Unit Disk Graphs (UDGs)*:

Definition 6 (Unit Disk Graph)**.** *Let V be a set of points in the Euclidean plane and $E = \{\{p, q\} \in \binom{V}{2} \mid |pq| \leq 1\}$ with $|pq|$ being the Euclidean distance of p and p. Then, $G = (V, E)$ is called a* Unit Disk Graph.

In the following, we consider the UDG to have weighted edges with weight $c_{pq} = |pq|$ for the edge between any two points p and q.

An interesting property of UDGs is that their doubling dimension α can be bounded by their squared diameter, i.e. $\alpha \in \mathcal{O}(D^2)$.

Lemma 3. *Let G be a UDG on the node set V, $v \in V$ a vertex and $r > 0$. Then, we can cover any ball $B = B_v(r)$ with $\mathcal{O}(\max(1, r^2))$ balls with radius at most $\frac{r}{2}$.*

This was already shown by Max Willert [Wil20]. However, we revisit his proof and add a bit more detail:

Proof. We consider a Euclidean disk E centered at v with radius r. Obviously, we have $B \subset E$. Now, we show that E can be covered by a disk set $\mathcal{E}$ of $K \in \mathcal{O}(\max(1, r^2))$ disks E_i with radius $r' = \min(\frac{r}{4}, \frac{1}{2})$ each. We distinguish two cases for this.

1. $r < 2$: E can be covered by $K \in \mathcal{O}(1)$ disks with radius $r' = \frac{r}{4}$ due to this simple geometric argument: E can be covered by a square with side length $2r$. We subdivide this square by a grid of 64 square cells with side length $\frac{r}{4}$, as illustrated in Figure 4. Every one of these cells can be covered by a disk with radius $\frac{r}{4}$ centered at the cell's center. This means that the disks cover all cells of the big square and thus E as well.

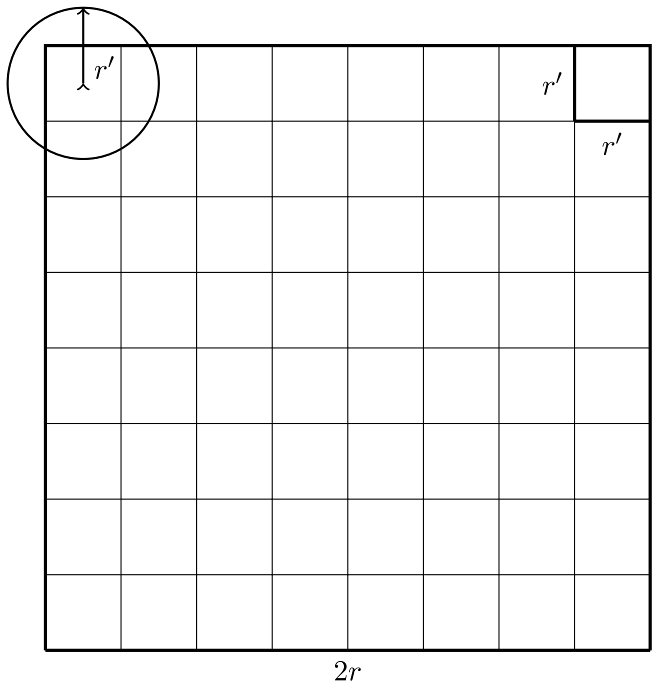


Figure 4: E can be covered by 64 disks with radius $r' = \frac{r}{4}$

2. $r \geq 2$: We use a similar geometric argument to show that E can be covered by $K \in \mathcal{O}(r^2)$ disks with radius $r' = \frac{1}{2}$: Again, E can be covered by a square with side length $2r$. This time, we subdivide the square by a grid of $64r^2$ square cells with side length $\frac{1}{4}$. Every cell can be covered by a disk with radius $\frac{1}{2}$. Just like before, these disks cover the big square, which includes E.

We know that E can be covered by $\mathcal{O}(\max(1, r^2))$ of disks E_i with radius r' for both cases, which implies that

$$B \subset E \subset \bigcup_{i=1}^{K} E_i$$

and especially,

$$B = \bigcup_{i=1}^{K} E_i \cap B.$$

Next, we want to show that we can also cover $B_v(r)$ with balls instead of disks. For that purpose we fix a node $v_i \in B_v(r)$ for every disk E_i. The choice of v_i can go one of two ways:

1. $E_i \cap B \neq \emptyset$: v_i is an arbitrary node in $E_i \cap B$.
2. $E_i \cap B = \emptyset$: v_i is an arbitrary node in B.

For both cases we have $E_i \cap B \subseteq B_{v_i}(2r')$.

Next, $r \leq \frac{1}{2}$ implies that the nodes in E_i are a clique in G and $B_{v_i}(2r')$ necessarily contains all nodes in $E_i \cap B$. Also, we have $B_{v_i}(2r') \subseteq B_{v_i}(\frac{r}{2})$ due to $r' \leq \frac{r}{4}$.

We conclude:

$$B = \bigcup_{i=1}^{K} E_i \cap B \subseteq \bigcup_{i=1}^{K} B_{v_i}(2r') \subseteq \bigcup_{i=1}^{K} B_{v_i}\left(\frac{r}{2}\right)$$

□

If any ball with radius r in G can be covered by either $\mathcal{O}(1)$ or $\mathcal{O}(r^2)$ balls with radius $\leq \frac{r}{2}$, this has to be true especially for $r = D$. Thus, we have:

Corollary 4. *Let G be a UDG with diameter D. The doubling dimension α of G satisfies $\alpha \in \mathcal{O}(\max(1, D^2))$. For $D \geq 1$, we have $\alpha \in \mathcal{O}(D^2)$.*

3 Preprocessing analysis

The most important part of our work is provided in this section: We describe how to perform the preprocessing for the routing scheme by Konjevod et al. [KRX16] and analyse its running time. This can be divided in four big parts:

1. Assign global routing labels
2. Add range and shortest path information for all nodes
3. Add local routing labels for substructures
4. Construct search trees for substructures

Each of these parts is explained separately. Also, every part is divided into sub-parts which we refer to as *steps*. Each step consists of a description and a running time analysis. At the end, we add up all times to achieve the overall running time.

During the preprocessing, we want to avoid any calculations that depend on Δ. The routing scheme itself is scale-free since the space requirements do not depend on Δ and we want to keep this property for the preprocessing time as well.

3.1 Global routing labels

The first part deals with constructing essential data structures and assigning global routing labels $l(u)$ to all nodes $u \in V$. This defines the label function l for our routing scheme.

3.1.1 Floyd-Warshall

In many of the later steps, we require shortest path information for the input graph G. For this reason, we apply the algorithm of Floyd-Warshall to G and retrieve the shortest paths between all pairs of nodes as well as their lengths. We also sort the resulting structure in the following way: Every node u has its own list of distances to the other nodes, which we sort by increasing distance to u. Also, we store the corresponding paths, which we use later on.

The algorithm of Floyd-Warshall takes $\Theta(n^3)$ time. The sorting of those lists takes $\mathcal{O}(n^2 \log n)$ time, which still leaves us with $\mathcal{O}(n^3)$ for this step.

3.1.2 Compressed nets

As a next step, we want to construct 2^i-nets Y_i as described above. However, constructing all Y_i for $i \in [\log \Delta]$ would add a term depending on $\log \Delta$ to our running time, which we want to avoid. Instead, we only consider those Y_i featuring new elements, e.g. if $Y_{\log \Delta} = Y_{\log \Delta - 1} = \ldots = Y_k = \{x\}$ and $Y_{k-1} = \{x, y\} \supset \{x\}$, we only consider $Y_{\log \Delta}$ and Y_{k-1}, since all sets in between contain the same elements as $Y_{\log \Delta}$. To further reduce redundant calculations, we also only store the new elements for each set.

Definition 7 (Compressed Net)**.** *Let $Y_{\log \Delta} \subseteq \ldots \subseteq Y_0$ be a sequence of 2^i-nets Y_i for a given graph. For $i \in [\log \Delta]$, let $s(i)$ be the biggest number such that $Y_i = Y_{s(i)}$ and $t(i)$ be the smallest number such that $Y_i = Y_{t(i)}$. Then, a* compressed net *$Y_{s(i),t(i)}$ is defined as follows:*

$$Y_{s(i),t(i)} = Y_i \setminus Y_{s(i)+1}$$

In the following, we also denote the compressed nets by $Y_{s,t}$ if they do not refer to any existing 2^i-nets Y_i.

As an example, let us consider the previous 2^i-nets Y_i for G_1:

- $Y_5 = \{g\}$
- $Y_4 = \{a, g, h\}$
- $Y_3 = \{a, c, d, f, g, h\}$
- $Y_2 = \{a, c, d, f, g, h\}$
- $Y_1 = \{a, c, d, e, f, g, h\}$
- $Y_0 = \{a, b, c, d, e, f, g, h\}$

Now, our corresponding sequence of compressed sets $Y_{s(i),t(i)}$ is:

- $Y_{5,5} = \{g\}$
- $Y_{4,4} = \{a, h\}$
- $Y_{3,2} = \{c, d, f\}$
- $Y_{1,1} = \{e\}$
- $Y_{0,0} = \{b\}$

The central reason for using compressed sets is that we only have $\mathcal{O}(n)$ different sets, since every node appears in only one of those sets. With the original Y_i, we could have a lot more sets if $\log \Delta \gg n$.

In addition, we add an attribute called $i_{\max}(v)$ to every node $v \in V$. $i_{\max}(v)$ is the maximum index i of a Y_i such that $v \in Y_i$.

Lemma 5. *Let $v \in V, i \in [\log \Delta]$ and Y_i a 2^i-net. Then, we have:*

$$v \in Y_i \Leftrightarrow i_{\max}(v) \geq i$$

This follows directly from the definition of $i_{\max}(v)$ and Y_i. The construction of $Y_{s,t}$ and $i_{\max}(v)$ is described in Algorithm 1.

Algorithm 1: Constructing $Y_{s,t}$

```
Input: Graph $G$, distances $d(u,v)$ for all $u, v \in V$
Output: Compressed nets $Y_{s,t}$ according to Definition 7
1  $Y_{\log\Delta,\log\Delta} \leftarrow \{v\}, v \in V$ arbitrary node;
2  $i_{\max}(v) \leftarrow \log\Delta$;
3  $Q$ empty PriorityQueue;
4  for $u \in V \setminus \{v\}$ do
5      $Q.\mathrm{add}(u, d(u,v))$;
6  $s \leftarrow \log\Delta$;
7  while $Q$ not empty do
8      $(u,d) \leftarrow Q.\mathrm{extractMax}()$;
9      if $\lceil\log d\rceil - 1 < s$ then
10         update index of $Y_{s,s}$ to $Y_{s,\lceil\log d\rceil}$;
11         $s \leftarrow \lceil\log d\rceil - 1$;
12         create new set $Y_{s,s}$;
13     add $u$ to $Y_{s,s}$;
14     $i_{\max}(u) \leftarrow s$;
15     if $s = 0$ then
16         for $(u,d) \in Q$ do
17             add $u$ to $Y_{s,s}$;
18             $i_{\max}(u) \leftarrow s$;
19         break;
20     for $(u',d') \in Q$ do
21         if $d(u,u') < d'$ then
22             replace entry $(u',d')$ in $Q$ with $(u', d(u,u'))$;
```

The algorithm starts by choosing an arbitrary node $v \in V$ to be in the top-level compressed net $Y_{\log\Delta,\_}$ and assigns $i_{\max}(v)$ accordingly. Now, all remaining nodes with their distances to v are put into a priority queue. These distances are updated for every newly added node to resemble the minimum distances to all nodes already put into nets. We also maintain a counter s to denote upper index of the set we added the last node to, which is $\log \Delta$ in the beginning.

In every step, we want to extract the node u with the maximum distance d and check if u is already covered by the nodes already picked for the current value s and higher, meaning that the distance to those nodes is $\leq 2^s$. This is represented by the comparison $\lceil \log d \rceil - 1 < s$, since we have:

Lemma 6. *Let $s \in \mathbb{N}$ and $d \in \mathbb{R}$. Now, we have:*

$$d \leq 2^s \Leftrightarrow \lceil \log d \rceil - 1 < s$$

Proof. We distiguish two cases:

1. $d \leq 2^s$:
$$d \leq 2^s \Rightarrow \lceil \log d \rceil \leq s \Rightarrow \lceil \log d \rceil - 1 < s$$
2. $d > 2^s$:
$$d > 2^s \Rightarrow \lceil \log d \rceil \geq s + 1 \Leftrightarrow \lceil \log d \rceil - 1 \geq s$$

□

If $\lceil \log d \rceil - 1 < s$, we do not need additional nodes for the current set $Y_{s,\_}$. We also know that u is required for the set $Y_{\lceil \log d \rceil - 1}$ due to Lemma 6: $\lceil \log d \rceil - 1$ is the highest possible value for s such that $d > 2^s$. We finish the current set by adding a lower index $\lceil \log d \rceil$ and we set $s \leftarrow \lceil \log d \rceil - 1$ to start a new set $Y_{s,s}$.

Now, we can guarantee that the node u has distance $d > 2^s$ from the nodes already picked, so u is added to the the current set $Y_{s,s}$. Also, we can set $i_{\max}(u)$ to be s. If we have $s = 0$, we add all remaining nodes to $Y_{s,s}$, we set $i_{\max} \leftarrow s$ for all these nodes and terminate the algorithm. We know that the (uncompressed) net $Y_0 = V$, so all remaining nodes have to be part of $Y_{0,0}$.

After adding u, we have to update the entrys in the priority queue, since this might reduce some distances. We continue by picking the new maximum entry from the queue and go on until all nodes have been put into exactly one compressed net.

Executing the construction algorithm requires $\mathcal{O}(n^2)$ time. For any iteration of the while-loop, there are potentially three different operations which require more than constant time:

1. Extract maximum from priority queue (Line 8). This takes $\mathcal{O}(\log n)$ time if we use e.g. a max-heap for implementing the queue.

2. Adding all remaining nodes to $Y_{0,0}$ (Line 15 to Line 19). This only happens once for $i = 0$ and takes $\mathcal{O}(n)$ time.

3. Updating distances in priority queue (from Line 20). In the worst case, we have to update $\mathcal{O}(n)$ entries. If we used a max-heap and restored the max-heap-property after every change, we would have $\mathcal{O}(n \log n)$ time per iteration. Instead, we can do all the updates first and perform a bottom-up construction for the entire heap afterwards. This only takes $\mathcal{O}(n)$ time.

Since we have $\mathcal{O}(n)$ time per iteration of the while-loop and $\mathcal{O}(n)$ iterations overall, this step takes $\mathcal{O}(n^2)$ time in total.

3.1.3 Compressed netting tree

To reflect the structure of the 2^i-nets Y_i in a compact way, Konjevod et al. used a *netting tree*:

Definition 8 (Netting Tree)**.** *Let* $Y_{\log\Delta}, Y_{\log\Delta-1}, \ldots, Y_0$ *be* 2^i*-nets for* $i \in [\log\Delta]$*. Consider zooming sequences* $u(0), u(1), \ldots, u(\log\Delta)$ *for all* $u \in V$*. Then, we obtain the* netting tree $T(\{Y_i\})$ for $\{Y_i\}$ *by interpreting every zooming sequence as a path with* $\log\Delta + 1$ *nodes and* $\log\Delta$ *undirected, unweighted edges.*

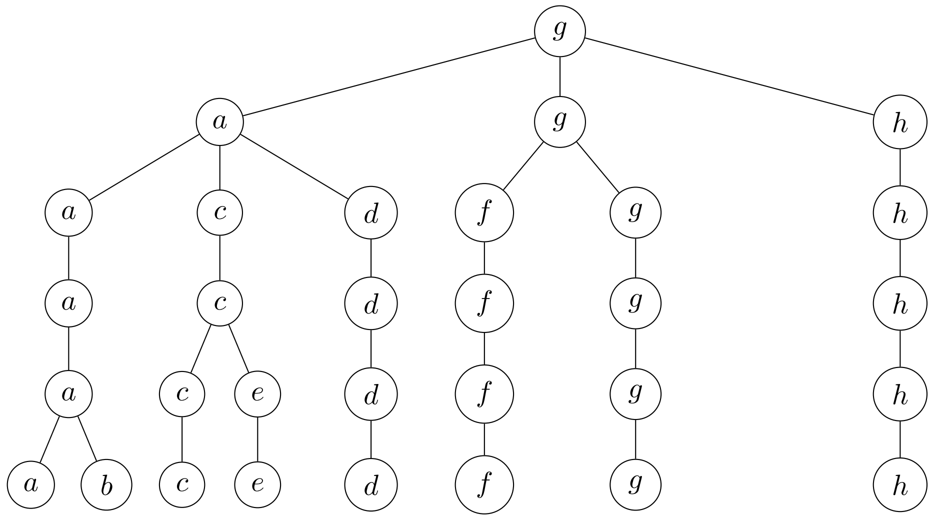
Whenever we have $u(i) = v(i)$ for any $u \neq v$, the paths meet and only one node $u(i)$ appears in the netting tree. This especially holds for the last node $u(\log\Delta)$, which is common for all $u \in V$. Thus, $u(\log\Delta)$ is the root of the netting tree.

Let us consider our example graph G_1 again. We achieved zooming sequences

- a: a, a, a, a, a, g
- b: b, a, a, a, a, g
- c: c, c, c, c, a, g
- d: d, d, d, d, a, g
- e: e, e, c, c, a, g

- f: f, f, f, f, g, g
- g: g, g, g, g, g, g
- h: h, h, h, h, h, g

Now, we can interpret these sequences as paths and construct a netting tree with root g:



However, since the netting tree consists of n zooming sequences, each of which with length $\log \Delta$, we also have $\Omega(n \log \Delta)$ nodes in the tree. One simple way to deal with this is replacing the netting tree with a similar but reduced structure. We can see from the example above that we have a lot of redundancies in the tree: The node h for instance appears five times in a simple path. It comes to mind that we can reduce redundancies by compressing such paths. We obtain a *compressed netting tree* $T_C(\{Y_i\})$ as follows:

1. Let v be a node in $T(\{Y_i\})$ with distance j to the leaves. We change the marking of v to $v_{j,j}$, e.g. the root is marked by $\log \Delta, \log \Delta$, the leaves are marked by $0, 0$.
2. Next, for every inner node $v_{j,k}$ with one child, the child node is $v_{k-1,l}$ for some $l \leq k-1$. Then, we merge the two nodes into one node $v_{j,l}$.

Compressing the netting tree from the previous example looks like this:

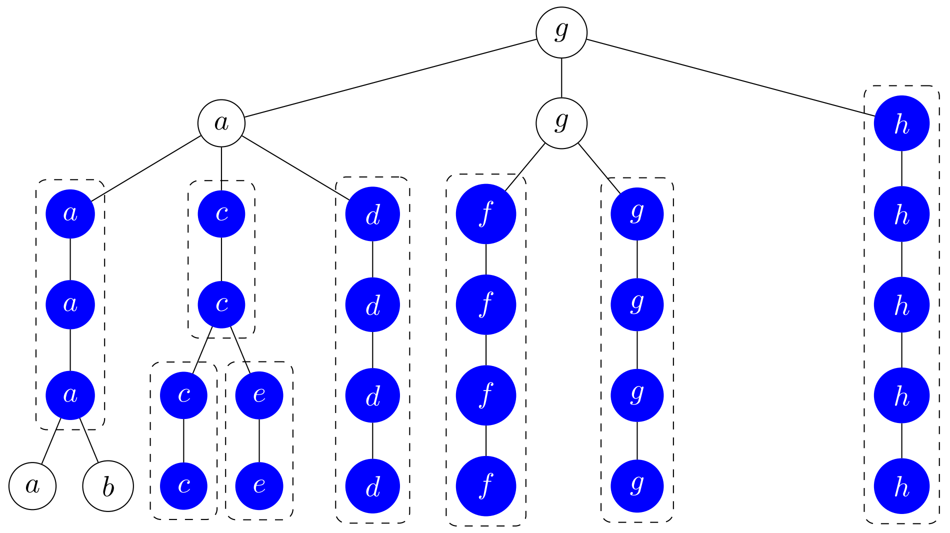


Figure 5: Netting tree before compression

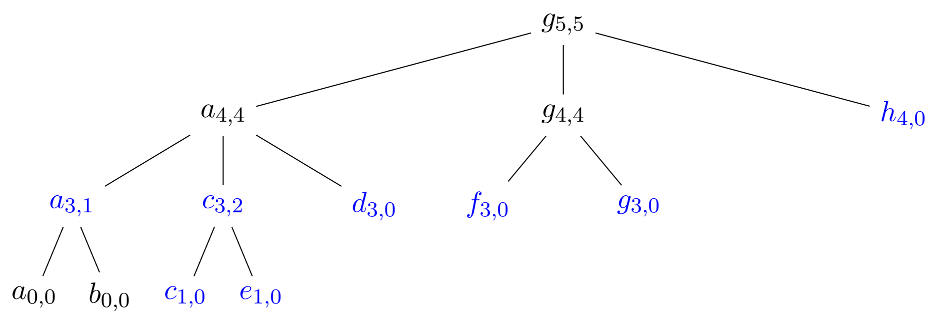


Figure 6: Netting tree after compression

This idea is similar to replacing the nets Y_i with compressed nets: To keep the level information, every node is marked by indices j, l, meaning we compressed appearances on levels j until l of this node in the netting tree. Since we compressed all inner nodes of degree one, all remaining inner nodes have degree at least two. The number of leaves still is n, so we have:

Lemma 7. *The number of nodes in the compressed netting tree is $\mathcal{O}(n)$.*

This means that the compressed netting tree could be useful for avoiding Δ. However, we still need to find a better way to construct the tree, since constructing a regular netting tree and compressing it afterwards would again take too much time. One such way is shown in Algorithm 2.

Algorithm 2: Constructing compressed netting tree

```
Input: Sequence $Y$ of compressed nets $Y_{s,t}$ for $G$ in decreasing order
       by indices
Output: Compressed netting tree $T_C(\{Y_i\})$
$T_C(\{Y_i\})$ empty tree;
$\{r\} \leftarrow$ first set $Y_{\log\Delta,\_}$ in $Y$;
make $r_{\log\Delta,\_}$ the root of $T_C(\{Y_i\})$;
/* Nodes already in the tree                                    */
$F \leftarrow \{r\}$;
for $Y_{s,t}$ in $Y$ with $s < \log\Delta$ do
    /* Add new level to tree                                    */
    for $w \in Y_{s,t}$ do
        determine $v \in F : d(w,v)$ minimum;
        add $w_{s,t}$ as child of $v_{\_,s+1}$;
        add $w$ to $F$;
    /* Finish level                                             */
    for $v \in F$ do
        if $v_{\_,s+1}$ has children then
            add $v_{s,t}$ as child of $v_{\_,s+1}$;
        else
            set $v_{\_,s+1}$ to $v_{\_,t}$;
```

The tree is constructed level by level, with the single node from $Y_{\log\Delta,\_}$ being the root. Now, we add a new level to the tree for every following set $Y_{s,t}$. For every new node w, we choose the parent to be the node in the level above closest to w, according to Definition 3. Also, every added level has to finished, meaning that all nodes from the level above either have to be added to the new level or, in case they do not have any children, the newly added nodes immediately merge with their parents. This effectively only changes the lower index to be t instead of $s+1$. In other words, we want all leaves to have the lower index t.

To illustrate the finishing step, let us consider the compressed netting tree from the previous example (Figure 6). We just added the node b from $Y_{0,0} = \{b\}$:

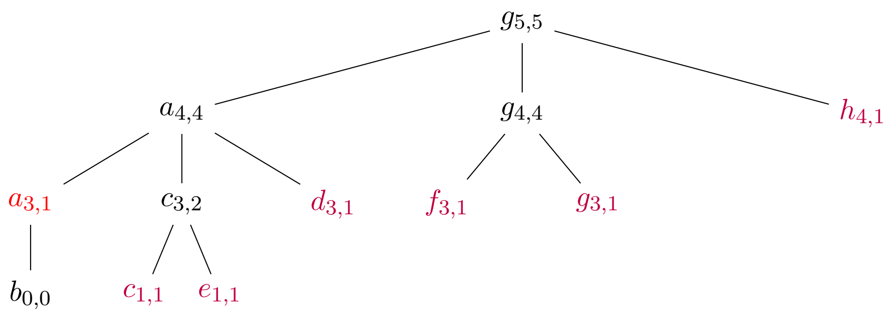


Figure 7: Netting tree before finishing step

It comes to mind that the leaf nodes c, d, e, f, g, h do not share the lower index 0 with the newly added node b. Also, the node a is no leaf anymore. The finishing step fixes both of these problems: First, we add $a_{0,0}$ as a child of $a_{3,1}$:

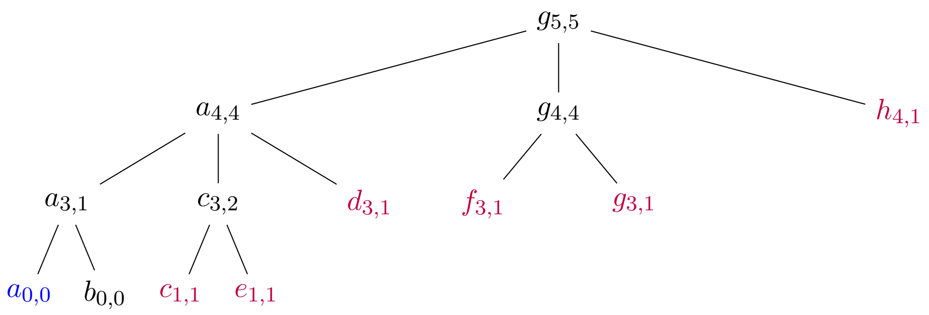


Figure 8: $a_{0,0}$ is added as a child of $a_{3,1}$

Now, the second part is just changing the lower indices of the leafs c, d, e, f, g, h to be 0 instead of 1:

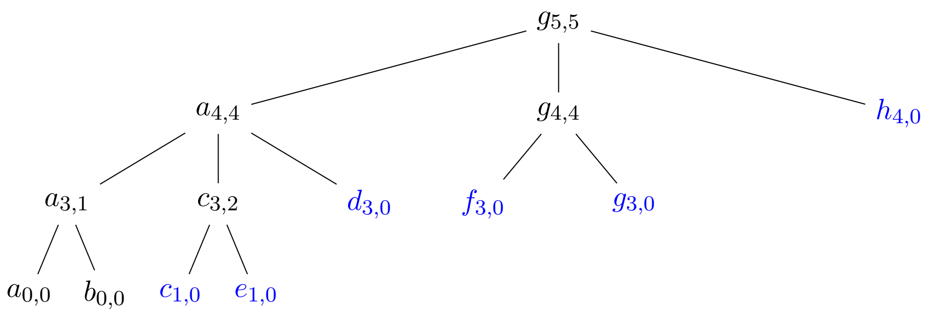


Figure 9: Lower indices of leafs are changed to be 0

Constructing the compressed netting tree requires $\mathcal{O}(n^2)$ time: We have to iterate all nodes $w \in V$, look up the closest node to w in F and add w to the tree. This takes $\mathcal{O}(n)$ time per node due to looking up distances and $\mathcal{O}(n^2)$ time overall. Finishing the levels takes $\mathcal{O}(n^2)$ time as well: There are $\mathcal{O}(n)$ levels in the resulting tree (one for each set $Y_{s,t}$) and for every level, we have $\mathcal{O}(n)$ time. This comes down to iterating F, while adding children and updating indices.

3.1.4 Add routing labels

The next step for our method is to add routing labels to our nodes. We achieve this by enumerating the leaves in the compressed netting tree, which are exactly V. For the enumeration, we use depth-first-search (DFS). We assume that the nodes are visited post-order, so the labels in any subtree are a range of integers:

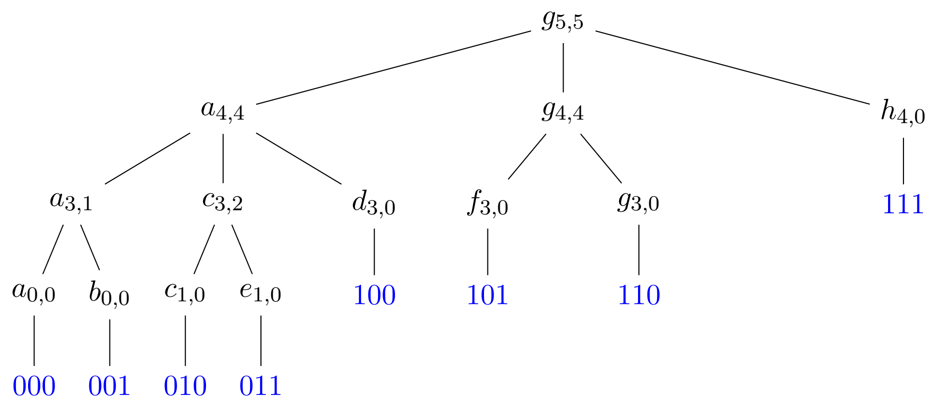


Figure 10: Compressed netting tree with labels

This effectively defines our label function $l : V \to \{0,1\}^*$: We assign a $\lceil \log n \rceil$-bitstring $l(u)$ to every node $u \in V$. In addition, we want every inner node to contain the information which range of labels appears in its subtree.

Definition 9. *Let $T_C(\{Y_i\})$ be a compressed netting tree, $x = x_{k,l}$ and inner node of $T_C(\{Y_i\})$ and $i \in [\log \Delta]$ such that $k \geq i \geq l$. Then,* $\text{Range}(x, i)$ *denotes the range of labels in the subtree spanned by x.*

To demonstrate this, we consider our compressed netting tree with labels (Figure 10). For example, we have $\text{Range}(a, 4) = [000, 100] = \{000, 001, 010, 011, 100\}$ since those are the labels in the subtree spanned by $a_{4,4}$. Similarly, we have $\text{Range}(c, 2) = [010, 011] = \{010, 011\}$ as labels in the subtree spanned by $c_{3,2}$.

In order to do that, we need to modify the DFS such that bitsequences or ranges of bitsequences for any nodes are also transfered to the level above. Algorithm 3 computes the labels and ranges.

Every inner node is visited at most three times: The first time for exploring the subtree, the second time for retrieving the children's labels, the third time for passing its range to its parent node. Similarly, the leaves are visited only twice. So each of the $\mathcal{O}(n)$ nodes is visited a constant number of times. Also, we assume that writing the $\log n$-bitsequences can be done in constant time. This means we have $\mathcal{O}(n)$ time for this step.

To conclude the running time for the first part, we have

$$\mathcal{O}(n^3) + \mathcal{O}(n^2) + \mathcal{O}(n^2) + \mathcal{O}(n) = \mathcal{O}(n^3)$$

Algorithm 3: Adding global labels and enumeration ranges

```
Input: Compressed netting tree $T_C(\{Y_i\})$ with root $r$
Output: $T_C(\{Y_i\})$ with inner nodes saving enumeration ranges
1  $c \leftarrow 0$;
2  $s$ stack containing only $r$;
3  while $s$ not empty do
4      curr $\leftarrow s$.top();
5      if curr . visited then
6          $a \leftarrow \min\{k \mid [k, l]$ ranges from curr's children$\}$;
7          $b \leftarrow \max\{l \mid [k, l]$ ranges from curr's children$\}$;
8          range(curr) $\leftarrow [a, b]$;
9          $s$. pop();
10         continue;
11     curr . visited $\leftarrow$ true;
12     for $v \in$ curr . children do
13         $s.push(v)$;
14     if curr . children empty then
15         $l$(curr) $\leftarrow$ bitsequence of $c$;
16         range(curr) $\leftarrow [c, c]$;
17         $c \leftarrow c + 1$;
```

3.2 Range and shortest path information

The next part deals with adding some range and shortest path information to the nodes: The range information for finding the label of the destination node more easily, and the shortest path information for taking a suitable path. Obviously, we cannot store shortest path information for *all* possible destination nodes, since this would require linear space in every routing table. Instead, we use index sets $R(u)$ and node sets $X_i(u)$ to find a tradeoff between storing useful information and maintaining polylogarithmic space per node.

3.2.1 Index sets

We now define index sets for all nodes $u \in V$ to determine the levels in $T_C(\{Y_i\})$ for which we store range information in u.

Definition 10. *Let $u \in V$. Then, we define the index $R(u)$ as follows:*

$$R(u) = \{i \in [\log \Delta] \mid \exists j \in [\log n] : \frac{\varepsilon}{6} \cdot r_u(j) \leq 2^i \leq r_u(j)\}.$$

According to Konjevod et al., we can bound the size of $R(u)$ nicely:

Lemma 8. *Let $u \in V$. Then, we have*

$$|R(u)| \in \mathcal{O}(\varepsilon^{-1} \cdot \log n).$$

We need to compute $R(u)$ for all $u \in V$. For each $R(u)$, we first consider all $j \in [\log n]$. Now, for a given combination of ε, u and j, we still need to obtain $r_u(j)$ somehow. In other words, we want the radius of a ball with u as a centerpoint containing 2^j nodes. We can achieve this by using the sorted list of shortest paths from u and jump straight to its 2^jth entry. The distance to this node must be $r_u(j)$, since it and all the corresponding previous nodes are in the ball while all the following nodes are not. Next, we can calculate $\frac{\varepsilon}{6}$ and check which numbers 2^i lie between $\frac{\varepsilon}{6} \cdot r_u(j)$ and $r_u(j)$. This can be done by calulating $\lceil \log(\frac{\varepsilon}{6} \cdot r_u(j)) \rceil$ and $\lfloor \log r_u(j) \rfloor$ for each $j \in [\log n]$. To acquire $R(u)$, we consider all i between $\lceil \log(\frac{\varepsilon}{6} \cdot r_u(j)) \rceil$ and $\lfloor \log r_u(j) \rfloor$ for each $j \in [\log n]$ and create a list from the entries.

We require $\mathcal{O}(\log n)$ time for computing the ranges and $\mathcal{O}(\varepsilon^{-1} \cdot \log n)$ for creating the list for a single $u \in V$. The latter is due to Lemma 8. Creating ranges and lists for all $u \in V$ leads to $\mathcal{O}(\varepsilon^{-1} \cdot n \log n)$ time for this step.

3.2.2 Second-level indices

In the previous step, we computed index sets $R(u)$ for storing range information $\text{Range}(x, i)$ in the routing tables. However, we cannot store i itself in any routing table or packet header, because $i \in [\log \Delta]$ and we potentially require $\log \log \Delta$ bits space for every such i. For this reason, we introduce second-level indices a_i.

Definition 11. *Let L be a list containing all indices from $\bigcup_{u \in V} R(u)$ sorted by increasing order. Let $i \in \bigcup_{u \in V} R(u)$. The* order *of i, denoted by a_i, is the index of i in L.*

We construct L by unifying the lists for all $R(u), u \in V$ and sorting by increasing order. Afterwards we iterate L and assign a_i for every element i in L. For a_i, we have the following important property, which follown directly from the definition of a_i:

Lemma 9. *Let $i \in R(u), i' \in R(u')$ be indices for any nodes $u, u' \in V$. Then, we have*

$$i \leq i' \Leftrightarrow a_i \leq a_{i'}$$

We know that $|R(u)| \in \mathcal{O}(\varepsilon^{-1} \cdot \log n)$ for every $u \in V$. That means we have $|L| \in \mathcal{O}(\varepsilon^{-1} \cdot n \log n)$. Sorting takes $\mathcal{O}(\varepsilon^{-1} \cdot n \log^2 n)$ time. Assigning $a_i, i \in L$ takes constant time per i and $\mathcal{O}(\varepsilon^{-1} \cdot n \log n)$ time overall. In total, that is $\mathcal{O}(\varepsilon^{-1} \cdot n \log^2 n)$ time for this step.

Moreover, the required space for a_i is logarithmic:

Lemma 10. *Let $i \in R(u), u \in V$. We require $\mathcal{O}(\log n)$ bits to store a_i.*

Proof. We have $|L| \in \mathcal{O}(\varepsilon^{-1} \cdot n \log n)$. Since a_i is the index for an element in L, a_i requires $\mathcal{O}(\log |L|)$ bits. We have:

$$\log |L| \leq \log(c \cdot \varepsilon^{-1} \cdot n \log n) \leq c \cdot \log(n \cdot n \cdot n) = 3c \cdot \log n \in \mathcal{O}(\log n)$$

□

3.2.3 Node sets and storing ranges

Now, we want to store label ranges in all nodes $u \in V$. The levels in the netting tree which we want to use for that are already given by $R(u)$. We do not use all nodes on those levels however, since this takes too much space. Instead, we only include nodes within distance $\frac{2^i}{\varepsilon}$ for each level i. To specify the included nodes, we introduce the *i-th rings of $X_i(u)$*:

Definition 12. *Let $i \in [\log \Delta]$ and $u \in V$. Then, the* i-th ring of u $X_i(u)$ *is defined as*

$$X_i(u) = B_u\left(\frac{2^i}{\varepsilon}\right) \cap Y_i$$

During the routing algorithm, we use the nodes from $X_i(u)$ as temporary destination nodes to get closer to v. We achieve that by choosing nodes $x \in X_i(u)$ such that Range(x, i) contains $l(v)$.

To store the range information Range(x, i) in every node u, we need to look up those ranges for every $x \in X_i(u), i \in R(u)$. We procede as follows: For every $i \in R(u)$, consider every $x \in V$ and check whether x lies in $X_i(u)$. According to definition, we have the following conditions:

- $x \in B_u(\frac{2^i}{\varepsilon})$: We calculate $\frac{2^i}{\varepsilon}$ and compare it to $d(x, u)$. We have $x \in B_u(\frac{2^i}{\varepsilon})$ if and only if $d(x, u) \leq \frac{2^i}{\varepsilon}$.
- $x \in Y_i$: We have $x \in Y_i$ if and only if $i_{\max}(x) \geq i$ by Lemma 5. We can just look up $i_{\max}(x)$ and make the comparison.

If and only if both conditions are fulfilled for a given combination of u, i, x, we have $x \in X_i(u)$. In that case, we look up the entry Range(x, i) from $T(\{Y_i\})$ and store it in the routing table of u. Along with Range(x, i), we also store a_i and $l(x)$, so we are able to identify the entry. In Section 3.3.4, we extend these entrys to store additional information.

Since we consider every combination of u, i, x and require constant time to check both conditions, we have time complexity

$$\mathcal{O}(|V|^2 \cdot |R(u)|) = \mathcal{O}(\varepsilon^{-1} \cdot n^2 \log n)$$

for this step.

For the second part, we have

$$\mathcal{O}(\varepsilon^{-1} \cdot n \log n) + \mathcal{O}(\varepsilon^{-1} n \log^2 n) + \mathcal{O}(\varepsilon^{-1} \cdot n^2 \log n) = \mathcal{O}(\varepsilon^{-1} \cdot n^2 \log n)$$

running time.

3.3 Local routing labels

The first two parts of preprocessing provide enough information to perform the first phase of the routing algorithm, which is described later. To be able to also route the rest of the way to v, we consider ball packings and subdivide G by Voronoi regions (Definition 13) with the balls' centerpoints. By that, we have the current node lie in the same Voronoi region as v. Also, we use a tree routing scheme as part of our general routing scheme to route to the center of the Voronoi region and from there to v. This tree routing scheme adds local routing labels to the nodes, which are stored in the routing tables, instead of overriding the global routing labels from part one.

3.3.1 Ball packings

First of all, we construct ball packings $\mathcal{B}_j$ for every $j \in [\log n]$. For each $\mathcal{B}_j$, we have to construct the set $\{B_u(r_u(j)) \mid u \in V\}$ first. To get a ball with 2^j nodes, we just take the first 2^j entries from shortest-path-list for every $u \in V$ and receive $B_u(r_u(j))$. We also sort this set by increasing radius of the balls.

Once the construction and sorting of $\{B_u(r_u(j)) \mid u \in V\}$ is done, we greedily pick non-intersecting balls from the set, small radii first. For each ball, we need to check if it is not intersecting with the balls already picked. To efficiently check for intersection, we construct a hashtable storing all the entries already picked. Whenever we consider a new ball, we iterate over all its 2^j nodes and check if any are already stored in the hashtable.

We first examine the running time for a fixed j: We have $\mathcal{O}(n)$ time for constructing $\{B_u(r_u(j)) \mid u \in V\}$, since we have n fixed ranges of nodes in the set. Sorting the set takes $\mathcal{O}(n \log n)$ time. For the actual construction of $\mathcal{B}_j$, we have to check for intersection for all n balls as described above. One ball requires 2^j lookups in the hastable, which have constant cost. For a fixed j, we have the running time:

$$\mathcal{O}(n) + \mathcal{O}(n \log n) + n \cdot 2^j \cdot \mathcal{O}(1) = \mathcal{O}(n \log n + n \cdot 2^j)$$

Summing up the time for all $j \in [\log n]$, we get the total running time for this step:

$$\sum_{j=0}^{\log n} n \log n + n \cdot 2^j = n \log^2 n + n \sum_{j=0}^{\log n} 2^j = n \log^2 n + n \cdot 2n \in \mathcal{O}(n^2)$$

3.3.2 Voronoi regions

As soon as our ball packings are complete, we are ready to define Voronoi regions:

Definition 13 (Voronoi Region)**.** *Let $\mathcal{B}_j, j \in [\log n]$ be a ball packing and C the set of all centers of the balls in $\mathcal{B}_j$. For $c \in C$, we define*

$$V(c, j) = \{u \in V \mid \forall c' \in C : d(u, c) \leq d(u, c')\}$$

to be the Voronoi region of c *for the Voronoi diagram of* C.

For routing efficiently in the Voronoi regions, we use shortest path trees:

Definition 14. *Let $B_c(j)$ be a ball in $\mathcal{B}_j, j \in [\log n]$ and $V(c,j)$ the Voronoi region of c. We define $T_c(j)$ to be the shortest path tree rooted at c spanning $V(c,j)$.*

Considering one ball packing $\mathcal{B}_j$, we need to compute Voronoi regions $V(c,j)$ for every center c of a ball $B_c(j)$ in $\mathcal{B}_j$ and shortest path trees $T_c(j)$ for every region. We can achieve this by adding a temporary dummy node v_0 to G, which also has edges $\{v_0, c\}$ with weight 0 for every center c. After this, we can apply Dijkstra's Algorithm to the graph: After starting at v_0, the centers c are visited first. When finished, we receive a shortest path tree for the whole graph. We can now get the shortest path trees $T_c(j)$ spanning every $V(c,j)$ by removing the root v_0 and interpreting every former subtree as a new shortest path tree. It is easy to see that every node in the graph is part of exactly one shortest path tree, implying

$$\sum_{i=1}^{k} |T_{c_i}(j)| = n$$

Also, we can traverse each tree $T_c(j)$ to get the set $V(c,j)$. Similar to the trees, the Voronoi regions are a partition for the graph G.

Dijkstra's algorithm takes $\Theta(|E| + n\log n)$ time for a single execution, which we can bound with $\mathcal{O}(n^2)$ since G is simple. Modifying the graph and the tree takes linear time, similar to the tree traversal. This leaves us with $\mathcal{O}(n^2)$ time per ball packing and $\mathcal{O}(n^2 \log n)$ time for this step.

3.3.3 Routing in trees

Fraigniaud and Gavoille introduced a labeled routing scheme on trees [FG01] that requires $\mathcal{O}(n \log n)$ time for a tree with n nodes. Thorup and Zwick came to a similar result [TZ01]. This tree routing scheme enables routing along the shortest paths by enumerating the nodes with DFS-numbers and giving every node a label consisting of the DFS-number and different information about its path from the root. Since both these labels and the routing tables require $\mathcal{O}(\frac{\log^2 n}{\log\log n})$ bits at every node, we can unify the two components into a $\mathcal{O}(\frac{\log^2 n}{\log\log n})$-bit label containing all necessary information for routing. The header only contains the label of the destination node.

We apply this routing scheme to all shortest path trees $T_c(j)$ and obtain local routing labels for our tree routing scheme:

Definition 15. *For any node u in $T_c(j)$, consider the unified label. We call this the* local routing label $l(u; c, j)$ *of u.*

Now, for every shortest path tree $T_c(j)$, every node u stores both its own as well as the center's local routing label $l(c; c, j)$. This ensures that we can route efficiently to c from every node in the Voronoi region $V(c, j)$.

When applying the routing scheme, we consider each of the shortest path trees $T_c(j)$ for each ball packing $\mathcal{B}_j$ from the previous step. For a fixed j, the search trees are disjoint and the scheme takes $\mathcal{O}(n \log n)$ time for preprocessing on a search tree with n nodes. Now, we can conclude:

Lemma 11. *The application of Fraigniaud's and Gavoille's routing scheme on all shortest path trees $T_c(j)$ for all ball packings $\mathcal{B}_j, j \in [\log n]$ takes $\mathcal{O}(n \log^2 n)$ time.*

Proof. Let $\mathcal{B}_j$ be a ball packing consisting of k balls with centers $c_1, c_2, \ldots, c_k$. Following from the previous step, each center c_i is root of a shortest path tree $T_{c_i}(j)$ and we have $\sum_{i=1}^{k} |T_{c_i}(j)| = n$. We want to apply Fraigniaud's and Gavoille's routing scheme on all $T_{c_i}(j)$ for a fixed j. When summing up the running time $T_j(n)$ for that, we get:

$$\begin{aligned} T_j(n) &\leq \sum_{i=1}^{k} c \cdot |T_{c_i}(j)| \log |T_{c_i}(j)| \\ &\leq c \cdot \sum_{i=1}^{k} |T_{c_i}(j)| \log n \\ &= c \cdot \log n \cdot \sum_{i=1}^{k} |T_{c_i}(j)| \\ &= c \cdot \log n \cdot n \\ &\in \mathcal{O}(n \log n) \end{aligned}$$

This is the running time for one ball packing $\mathcal{B}_j$. Since we have to do this for every $\mathcal{B}_j, j \in [\log n]$, our total running time for this step is indeed $\mathcal{O}(n \log^2 n)$. □

3.3.4 Create range entrys

Once we computed the ball packings and Voronoi regions, we can compute range entrys for the routing tables. Range entrys contain a variety of useful information required for the routing algorithm. We already computed entrys consisting of $\text{Range}(x, i)$, $l(x)$ and a_i for $x \in X_i(u), i \in R(u)$ for all nodes $u \in V$ in Section 3.2.3. We now extend these entrys to *range entrys*:

Definition 16 (Range Entry)**.** *Let $u \in V, x \in X_i(u), i \in R(u)$. Then, a* range entry for *u is a tuple*

$$Z = (\text{Range}(x, i), l(x), a_i, l(y), b, j, l(c))$$

such that:

(i) y is the next node on the shortest path from u to x.

(ii) b is a boolean with the value resulting from

$$d(u, x) \geq \varepsilon^{-1} \cdot 2^{i-1} - 2^i.$$

(iii) j is the index in $[\log n]$ such that

$$r_u(j) \leq 2^i < r_u(j + 1).$$

(iv) c is the center of a ball $B \in \mathcal{B}_j$ such that $u \in V(c, j)$.

We already explained the purpose of $\text{Range}(x, i)$, $l(x)$ and a_i. $l(y)$ is necessary for the routing algorithm to route towards x using the shortest path. b is used to execute some comparisons of the routing algorithm in advance, since they potentially involve real numbers and thus require too much space. j and c are necessary to transition between the phases of the routing algorithms smoothly: After we finish the first phase of our algorithm, we want to switch to the tree routing scheme. j and c provide us with the information that we route within the Voronoi region $V(c, j)$.

Now, for every node $u \in V$, we want to extend every entry consisting of $\text{Range}(x, i)$, $l(x)$ and a_i to a range entry. This means that we have to add $l(y)$, b, j and c for a given combination of $u \in V, x \in X_i(u), i \in R(u)$. i can be determined easily since we know a_i.

To compute $l(y)$ and b, we just need to look up shortest path information from Section 3.1.1 and make the comparison for b. For calculating j, we iterate all $j \in [\log n]$ and check the condition until we find a suitable value. $r_u(j)$ can be determined in constant time by looking up $d(u, z)$, with $z \in V$ being the 2^j-th entry in u's list of nodes with distances. We proceed similarly for $r_u(j + 1)$. As soon as we know j, we check which shortest path tree $T_c(j)$ contains u and choose c accordingly.

Extending an entry consisting of $\text{Range}(x, i)$, $l(x)$ and a_i to a range entry requires $\mathcal{O}(\log n)$ time: Looking up shortest path information and the comparison for b take constant time. For j, we have $\mathcal{O}(\log n)$ time for iterating $[\log n]$, a single iteration requires constant time. $l(c)$ can be determined in constant time. We assume that u stores the information which shortest path trees $T_c(j)$ it is part of (during preprocessing).

Also, the additional information in the range entrys still only takes a logarithmic number of bits per entry:

Lemma 12. *The number of bits required for a singular range entry is* $\mathcal{O}(\log n)$.

Proof. Each range entry has a constant number of components, all of which require at most a logarithmic number of bits:

1. Range(x, i) is given by the upper and the lower end of the range, both of which are global routing labels. This implies $\mathcal{O}(\log n)$ space.
2. $l(x), l(y), l(c)$ are global routing labels as well and take $\mathcal{O}(\log n)$ bits each.
3. a_i takes $\mathcal{O}(\log n)$ bits by Lemma 10.
4. b is a boolean and requires only one bit.
5. j is a number in $[\log n]$ and thus requires $\mathcal{O}(\log n)$ bits.

Hence, the total number of bits for a single range entry is $\mathcal{O}(\log n)$. □

This means that we maintain the space constraints for range information given by Konjevod et al.:

Lemma 13. *For every node* $u \in V$*, we require* $\left(\frac{1}{\varepsilon}\right)^{\mathcal{O}(\alpha)} \log^2 n$ *bits for range entrys.*

Proof. According to Lemma 12, every range entry requires $\mathcal{O}(\log n)$ bits. The total number of range entrys for a single node $u \in V$ is

$$\mathcal{O}(|R(u)| \cdot |X_i(u)|) = \mathcal{O}\left(\varepsilon^{-1} \cdot \log n \cdot \left(\frac{1}{\varepsilon}\right)^{\mathcal{O}(\alpha)}\right) = \mathcal{O}\left(\left(\frac{1}{\varepsilon}\right)^{\mathcal{O}(\alpha)} \log n\right)$$

Now it is clear that we require $\mathcal{O}(\left(\frac{1}{\varepsilon}\right)^{\mathcal{O}(\alpha)} \log^2 n)$ bits for the range entrys for u. □

Considering the total number of $\mathcal{O}(\varepsilon^{-1}\cdot n^2\log n)$ entrys we have to extend (see Section 3.2.3), the running time for this step is $\mathcal{O}(\varepsilon^{-1}\cdot n^2\log^2 n)$. This means we have

$$\mathcal{O}(n^2)+\mathcal{O}(n^2\log n)+\mathcal{O}(n\log^2 n)+\mathcal{O}(\varepsilon^{-1}\cdot n^2\log^2 n)=\mathcal{O}(\varepsilon^{-1}\cdot n^2\log^2 n)$$

running time for the third part.

3.4 Search trees

To make the final steps for routing from c to u, we construct search trees $T'(c,r)$. They are used to retrieve the local label of our destination node. To ensure we can search efficiently in those trees, we connect virtual edges and apply another local routing scheme.

3.4.1 Construct trees

Definition 17 (Search Tree II)**.** *Let $\varepsilon\in(0,1)$ and $B_c(r)$ be a ball in G. Also, let $U_0=\{c\}$. We recursively define $2^{\lfloor\log(\varepsilon r)\rfloor-i}$-nets U_i of $B_c(r)\setminus\bigcup_{0\le j<i}U_j$ for $1\le i\le\min(\lceil\log n\rceil,\lfloor\log(\varepsilon r)\rfloor)$. Now, $T'=(V',E')$ is called the* search tree II on *$B_c(r)$, with V' and E' in one of two ways:*

1. *$\lceil\log n\rceil\ge\lfloor\log(\varepsilon r)\rfloor$: We have $0<i\le\lfloor\log(\varepsilon r)\rfloor$. Now, we have $V'=\bigcup U_i$ and obtain E' by connecting every $v\in U_i$ to its closest neighbour w in U_{i-1} and assigning the weight $d(u,w)$.*

2. *$\lceil\log n\rceil<\lfloor\log(\varepsilon r)\rfloor$: We have $0<i\le\lceil\log n\rceil$. Now, we connect every $v\in U_i$ to its closest neighbour w in U_{i-1} and assign the weight $d(u,w)$. Also, for every $u\in U_{\lceil\log n\rceil}$, let*

$$V(u)=\{x\in B_c(r)\mid d(x,u)\le d(x,u')\}$$

for any $u'\in U_{\lceil\Delta\rceil}$. We call $V(u)$ the Voronoi region *of u for the Voronoi diagram of all $u'\in U_{\lceil\Delta\rceil}$. To complete the tree, all nodes of every $V(u)\setminus\bigcup U_i$ are connected into arbitrary paths, which are connected to u. All of these new edges have weight $\frac{2\varepsilon r}{n}$.*

In this step, we want to construct a search tree $T'(c,r)$ for every center c of a ball $B_c(r)\in\mathcal{B}_j$ and $j\in[\log n]$. We first consider a ball packing $\mathcal{B}_j$ for a fixed $j\in[\log n]$. For every ball $B_c(r)$ in $\mathcal{B}_j$, we have to compute nets $U_i, i\in[\min(\lceil\log n\rceil,\lfloor\log(\varepsilon r)\rfloor)]$. We use Algorithm 4.

Algorithm 4: Constructing U_i

```
Input: Ball $B_c(r)$ consisting of $2^j$ nodes
Output: $2^{\lfloor \log(\varepsilon r) \rfloor - i}$-nets $U_i$ according to Definition 2 for
        $1 \leq i \leq \min(\lceil \log n \rceil, \lfloor \log(\varepsilon r) \rfloor)$
1  $U_0 \leftarrow \{c\}$;
   /* $R$ denotes remaining nodes */
2  $R \leftarrow B_c(r) \setminus \{c\}$;
   /* Construct every single $U_i$ */
3  for $i \leftarrow 1$ to $\min(\lceil \log n \rceil, \lfloor \log(\varepsilon r) \rfloor)$ do
4      $U_i \leftarrow \emptyset$;
       /* $S_i$ denotes nodes not yet covered by $U_i$ */
5      $S_i \leftarrow R$;
6      while $S_i$ not empty do
7          $v \leftarrow$ arbitrary node from $S_i$;
8          $U_i \leftarrow U_i \cup \{v\}$;
9          $R \leftarrow R \setminus \{v\}$;
           /* remove nodes covered by $v$ */
10         for $x \in R$ do
11             if $d(v, x) \leq 2^{\lfloor \log(\varepsilon r) \rfloor - i}$ then
12                 $S_i \leftarrow S_i \setminus \{x\}$;
```

The first set $U_0 = \{c\}$ is given by definition. Now, when constructing the subsequent nets U_i, we always need to cover all nodes which are not member of any previous net, since U_i must be a $2^{\lfloor \log(\varepsilon r) \rfloor - i}$-net of $B_c(r) \setminus \bigcup_{0 \leq j < i} U_j$. We reflect $B_c(r) \setminus \bigcup_{0 \leq j < i} U_j$ by the set R in the algorithm. We also use the set S_i to ensure all nodes in R get covered by U_i, i.e. for every node x, there is a node $v \in U_i$ such that $d(x, v) \leq 2^{\lfloor \log(\varepsilon r) \rfloor - i}$. Whenever we add a node v to U_i, we remove all nodes x within distance $2^{\lfloor \log(\varepsilon r) \rfloor - i}$. We proceed adding nodes to U_i until S_i is empty, meaning that all nodes in R are covered. By removing the covered nodes, the nodes in U_i also maintain a distance of more than $2^{\lfloor \log(\varepsilon r) \rfloor - i}$ to each other. Thus, every U_i constructed by Algorithm 4 is a valid $2^{\lfloor \log(\varepsilon r) \rfloor - i}$-net for $B_c(r) \setminus \bigcup_{0 \leq j < i} U_j$.

Since one ball has 2^j elements, executing this algorithm takes $\mathcal{O}((2^j)^2)$ time: We have $\mathcal{O}(2^j)$ nodes in R initially. During the algorithm, we have $\mathcal{O}(2^j)$ comparisons for every node we remove from R. The total number of these nodes is $\mathcal{O}(2^j)$.

Also, the nodes in a search tree $T'(c, r)$ are a subset of the nodes in $B_c(r)$. Since $|B_c(r)| = 2^j$, we have:

Lemma 14. *The number of vertices in a single search tree $T'(c, r)$ is $\mathcal{O}(2^j)$.*

For connecting the nodes, we distinguish two cases:

1. $\lceil \log n \rceil \geq \lfloor \log(\varepsilon r) \rfloor$: When constructing the tree, we know it contains at most 2^j nodes (since it is constructed from the ball's nodes). We iterate the nodes of each U_i and search for the closest neighbour in U_{i-1} and connect those two nodes. Again, looking up at most 2^j distances per node means $\mathcal{O}((2^j)^2)$ time for constructing the whole tree.

   In case one, we have a running time

   $$\mathcal{O}((2^j)^2 + (2^j)^2) = \mathcal{O}((2^j)^2)$$

   for constructing one tree. This includes computing the nets U_i from above.

2. $\lceil \log n \rceil < \lfloor \log(\varepsilon r) \rfloor$: For case two, we first want to compute Voronoi regions $V(u)$ for every $u \in U_{\log n}$. We can achieve that by simply iterating all nodes x in $B_c(r)$ and determining the node $u \in U_{\log n}$ closest to x, which requires looking up 2^j distances per node x. When we find the closest node u, we assign x to the Voronoi region $V(u)$.

Similar to previous steps, those Voronoi regions partition all the nodes in $B_c(r)$.

Now, we compute

$$V(u) \setminus \bigcup_{0 \leq j \leq \lceil \log n \rceil} U_j$$

for every $u \in U_{\lceil \log n \rceil}$ and connect the nodes with a simple path. The order of the nodes is arbitrary for this matter. Also, we connect the path to u. Every edge we create this way has weight $\frac{2\varepsilon r}{n}$.

Computing the sets takes $\mathcal{O}(2^j)$ time per u and $\mathcal{O}((2^j)^2)$ time overall, since both $U_{\lceil \log n \rceil}$ and $V(u)$ contain more than 2^j nodes. For constructing the paths, we have $\mathcal{O}(2^j)$ running time for introducing the edges, because we have to iterate $\leq 2^j$ nodes and add $\leq 2^j$ edges.

For case two, this leaves us with a running time of

$$\mathcal{O}((2^j)^2 + (2^j)^2 + (2^j)^2 + 2^j) = \mathcal{O}((2^j)^2)$$

for constructing one tree. This includes computing the nets U_i from above.

Both cases yield $\mathcal{O}((2^j)^2)$ time for constructing one tree. Now, we consider the time for one ball packing $\mathcal{B}_j$ for a fixed $j \in [\log n]$. $\mathcal{B}_j$ consists of no more than $\frac{n}{2^j}$ balls, since the balls must be non-intersecting, which leads to

$$\mathcal{O}\left(\frac{n}{2^j} \cdot (2^j)^2\right) = \mathcal{O}(n \cdot 2^j)$$

time per ball packing. Considering all ball packings $\mathcal{B}_j$, for all $j \in [\log n]$:

$$\begin{aligned}
\sum_{j=0}^{\log n} \mathcal{O}(n \cdot 2^j) &\leq \sum_{j=0}^{\log n} c \cdot n \cdot 2^j \\
&= c \cdot n \cdot \sum_{j=0}^{\log n} 2^j \\
&\leq c \cdot n \cdot 2n \in \mathcal{O}(n^2)
\end{aligned}$$

3.4.2 Link virtual edges

The penultimate problem we have to deal with in preprocessing is that the search trees might contain *virtual edges*, i.e. edges that are not edges in G:

Definition 18 (Virtual Edges)**.** *Let T' be a Search Tree II. We call any edge e in T'* virtual*, if and only if e is not an edge in G.*

To search for local routing labels, we potentially need to use virtual edges though. For this reason, we need to connect the endpoints of all virtual edges. By this, we mean looking up the shortest paths between the endpoints and storing next-hop information along the paths.

Next, we need to link the endpoints of virtual edges $\{u, v\}$ in every search tree $T'(c, r)$. In order to achieve this, we first consider every such virtual edge $\{u, v\}$ and look up the shortest path $u \to v$ from step one. We now iterate this path and for every node x, we store its predecessor as well as its successor.

In the worst case, the number of vertices on the path $u \to v$ is n, which means that we need to make $\mathcal{O}(n)$ predecessor and successor entries for connecting u and v in $T'(c, r)$. In other words, we have $\mathcal{O}(n)$ running time per virtual edge.

The overall number of virtual edges in all search trees $T'(c, r)$ is $\mathcal{O}(n \log n)$: We have $\mathcal{O}(\log n)$ ball packings $\mathcal{B}_j$ and for every $\mathcal{B}_j$, we constructed search trees $T'(c, r)$. According to Lemma 14, every such tree has $\mathcal{O}(2^j)$ nodes. This means the entirety of search trees $T'(c, r)$ for a singular ball packing $\mathcal{B}_j$ represents a forest with at most

$$\frac{n}{2^j} \cdot \mathcal{O}(2^j) = \mathcal{O}(n)$$

vertices. A forest with $\mathcal{O}(n)$ vertices also has $\mathcal{O}(n)$ (potentially virtual) edges, implying $\mathcal{O}(n \log n)$ virtual edges for all search trees. In total, we need to make $\mathcal{O}(n \log n) \cdot \mathcal{O}(n) = \mathcal{O}(n^2 \log n)$ predecessor and successor entries for connecting all virtual edges.

3.4.3 Routing in Voronoi regions

Now, we want to construct shortest-path trees for all $V(u), u \in U_{\lceil \log n \rceil}$ for every $j \in [\log n]$ and apply the tree routing scheme to these trees as well. This is because we want to route along the shortest paths of the virtual edges when retrieving the local routing label of our destination node.

We use a modified version of Dijkstra's algorithm, as we did in Section 3.3.2: We add a dummy node v_0 as well as edges $\{v_0, u\}$ for every $u \in U_{\lceil \log n \rceil}$, apply Dijkstra's algorithm and remove v_0 with all its edges. The resulting graph is a forest of shortest-path trees, every one of which with root u and spanning $V(u)$. Just like before, this has do be done for every $j \in [\log n]$ and results in $\mathcal{O}(n^2 \log n)$ running time.

As soon as we have the shortest-path trees, we once more apply Fraigniaud's and Gavoille's routing scheme to every one of those trees. Since we have a forest with n nodes, we require $\mathcal{O}(n \log n)$ time for every ball packing and $\mathcal{O}(n \log^2 n)$ time for all ball packings. This is similar to Lemma 11.

Finally, we need to consider all the virtual edges in the paths spanning

$$V(u) \setminus \bigcup_{0 \leq j \leq \lceil \log n \rceil} U_j, u \in U_{\lceil \log n \rceil}.$$

For each of these edges, we need to add the endpoints' labels to each other. This means that for any virtual edge $\{x, y\}$, x also stores the local label of y and vice versa. Concerning the running time, we again consider $\mathcal{O}(\log n)$ ball packings, for every one of which we have $\mathcal{O}(n)$ virtual edges, leaving us with $\mathcal{O}(n \log n)$ time for storing this information.

Summing up the time for this whole step, we get $\mathcal{O}(n^2 \log n)$ time.

Now, for the fourth part we have

$$\mathcal{O}(n^2) + \mathcal{O}(n^2 \log n) + \mathcal{O}(n^2 \log n) = \mathcal{O}(n^2 \log n)$$

running time.

3.5 Overall complexity and adaptation to UDGs

To conclude the results from all four parts, we achieve our overall running time:

$$\begin{aligned}&\mathcal{O}(n^3) + \mathcal{O}(\varepsilon^{-1} \cdot n^2 \log n) + \mathcal{O}(\varepsilon^{-1} \cdot n^2 \log^2 n) + \mathcal{O}(n^2 \log n)\\ &= \mathcal{O}(n^3 + \varepsilon^{-1} \cdot n^2 \log^2 n)\end{aligned}$$

The space requirements for routing labels and routing tables are the same as described by Konjevod et al. The only difference is that we use range entrys instead of storing just the label ranges. According to Lemma 12 and Lemma 13, this does not affect the asymptotic number of bits though. We also have the same packet header size $\mathcal{O}(\frac{\log^2 n}{\log \log n})$, which we later see in Lemma 17.

Therorem 15. *The preprocessing for Konjevod et al.'s routing scheme takes $\mathcal{O}(n^3 + \varepsilon^{-1} \cdot n^2 \log^2 n)$ time for graphs with doubling dimension α. We have $\lceil \log n \rceil$-bit routing labels, $\left(\frac{1}{\varepsilon}\right)^{\mathcal{O}(\alpha)} \log^3 n$-bit routing tables at every node and $\mathcal{O}\left(\frac{\log^2 n}{\log \log n}\right)$-bit packet headers.*

Now, we can consider the case that our input graph is not just a graph with low doubling dimension, but a Unit Disk Graph. One of our main problems we had to solve during preprocessing was the APSP-Problem, i.e. finding the shortest paths for all node pairs. This required $\mathcal{O}(n^3)$ time for the algorithm of Floyd-Warshall. At this point, we use the fact that our graph is a UDG and resort to more efficient algorithms. Wang and Xue have developed a method with $\mathcal{O}(n \log^2 n)$ time for solving the SSSP-Problem for weighted UDGs [WX20], i.e. finding the shortest paths for a single source node. This means that we can solve the APSP-Problem for weighted UDGs in $\mathcal{O}(n^2 \log^2 n)$ by using Wang and Xues algorithm n times (once for every node in V). Thus, we do not have to use the Floyd-Warshall alogrithm in first step of preprocessing.

As a consequence, the term n^3 in our running time calculation is removed and can be replaced by $\mathcal{O}(n^2 \log^2 n)$. Then, the overall running time for preprocessing in UDGs is $\mathcal{O}(\varepsilon^{-1} \cdot n^2 \log^2 n)$. Also, we have $\alpha \in \mathcal{O}(\max(1, D^2))$ for UDGs by Lemma 3, so we can express the size of the routing tables in terms of $\mathcal{O}(\max(1, D^2))$ instead of α:

Corollary 16. *The preprocessing for Konjevod et al.'s routing scheme takes $\mathcal{O}(\varepsilon^{-1} \cdot n^2 \log^2 n)$ time for Unit Disk Graphs with diameter D. We have $\lceil \log n \rceil$-bit routing labels, $\left(\frac{1}{\varepsilon}\right)^{\mathcal{O}(\max(1,D^2))} \log^3 n$-bit routing tables at every node and $\mathcal{O}\left(\frac{\log^2 n}{\log \log n}\right)$-bit packet headers.*

4 Modified routing algorithm

In addition to our method for calculating the preprocessing step, we want to modify the routing algorithm from Konjevod et al. (Algorithm 5). Our motivation for this is that their algorithm was described in a global, iterative way: For example, it involved a loop for visiting new nodes. Also, the algorithm used global data structures like distances between nodes, $X_i(u)$ and $r_u(j)$-values. Using global data structures this way is not possible with our storage constraints: We would require to store this information in either the packet header or routing table, which are both not large enough due to the polylogarithmic storage constraints. Thus, we want to reformulate their algorithm using only local information in packet header and routing tables while maintaining the given storage constraints. Also, we want our algorithm to be recursive, for this reflects the nature of routing in a better way: During routing, we have to make a decision for every next step based only on the packet header and the routing tables of the current node. This lends itself to a recursive function, which is called for every visited node.

The original routing algorithm operates as follows: In the first phase, the algorithm considers the current node u and uses the range information Range(x, i) stored at u to determine a temporary destination node x_k. Ultimately, we want to reach v, but x_k gives us a good idea about the direction. x_k is chosen by the following criteria:

1. There is an entry Range(x_k, i_k) stored at u. Obviously, we can only choose nodes known to us, i.e. the nodes stored in u's routing table.

2. Range(x_k, i_k) contains the global routing label $l(v)$ of the destination node. This guarantees that x_k is part of v's zooming sequence.

3. i_k is minimal. We want x_k to be close to v, so we also want x_k to be on a relatively low level of the netting tree.

The first phase continues until one of the two conditions is satisfied:

1. $i_k > i_{k-1}$: This means that our new choice for x_k and i_k moves further up in the netting tree, and thus potentially further from v. We want to avoid this.

2. $d(u_k, x_k) < \frac{2^{i_k-1}}{\varepsilon} - 2^{i_k}$: The lower ε gets, the better our routing path has to be, since we want to achieve a $(1 + \varepsilon)$-stretch. Further details on this property can be found in Konjevod et al.'s article.

In the second phase, the algorithm picks a parameter j for the ball packing $\mathcal{B}_j$, such that the current node u_k and the destination v are in the same Voronoi region $V(c, j)$. c is the center of a ball $B_c(j) \in \mathcal{B}_j$. Then, the algorithm uses the local tree routing scheme of Fraigniaud and Gavoille to route to the ball's center c using the shortest path.

In the third phase, the algorithm uses the SearchTree-method described by Konjevod et al. to retrieve the local routing label $l(v; c, j)$ from the search tree II rooted at c.

Once $l(v; c, j)$ has been acquired, the fourth and final phase is excuted: The algorithm uses the local tree routing scheme again to route to v along the shortest path.

Algorithm 5 is not exactly the original algorithm, since we already replaced the GoTo-statement with a while-loop. Also, we removed the redundant variable t and used k instead.

Using a recursive approach, the routing algorithm can be defined differently. We want to reflect the four phases by a variable p in the packet header,

Algorithm 5: Algorithm by Konjevod et al.

```
    Input: Source node $u$, label $l(v)$ of destination node, pre-processed
           graph $G$
 1  $k \leftarrow 0$;
 2  $u_0 \leftarrow u$;
 3  $i_{-1} \leftarrow +\infty$;
 4  $i_k \leftarrow \min\{i_k \in R(u_k) \mid \exists x_k \in X_{i_k}(u_k) : l(v) \in \text{Range}(x_k, i_k)\}$;
 5  while $i_k \leq i_{k-1}$ and $d(u_k, x_k) \geq \frac{2^{i_k - 1}}{\varepsilon} - 2^{i_k}$ do
 6      $u_{k+1} \leftarrow$ next hop on shortest path $u_k \rightarrow x_k$;
 7      visit $u_{k+1}$;
 8      $k \leftarrow k + 1$;
 9      $i_k \leftarrow \min\{i_k \in R(u_k) \mid \exists x_k \in X_{i_k}(u_k) : l(v) \in \text{Range}(x_k, i_k)\}$;
10  $j \leftarrow$ index in $[\log n]$ such that $r_{u_k}(j) \leq 2^{i_k} < r_{u_k}(j+1)$;
11  $c \leftarrow$ center of ball $B \in \mathcal{B}_j : u_k \in V(c, j)$;
12  route to $c$ using labeled tree routing on $T_c(j)$;
13  $l(v; c, j) \leftarrow \text{SearchTree}(l(v), T'(c, r_c(j)))$;
14  route to $v$ using $l(v; c, j)$ and labeled tree routing on $T_c(j)$;
```

which indicates the current phase. Also, we check whether we reached v at the beginning of each recursion step, since this is possible to happen even in the first phase.

The initial call for the algorithm is $a(u, l(v), h)$, with h containing the following information:

- $a_{\text{prev}} \leftarrow \infty$
- $p \leftarrow 1$

Later on, h stores the following information:

- Range entry $(r, x, a_{\text{prev}}, y, b, j, c)$ for the previous node. Whenever a new range entry is stored (Line 10 of Algorithm 6), the old one is overwritten.
- p as an indicator for the current phase.
- The local routing label $l(c; c, j)$ of the center node, which is used throughout phase 2.
- The local routing label $l(v; c, j)$ of the destination node v, which is retrieved in phase 3.

Algorithm 6: Recursive algorithm $a(w, l(v), h)$

```
Input: Node $w$, destination label $l(v)$ packet header $h$
if $l(w) = l(v)$ then
    finished;
case $p = 1$ do
    pick range entry $(r, l(x), a_i, l(y), b, j, l(c))$ for $w$ such that $r$
      contains $l(v)$ and $a_i$ minimum;
    if $b = 1$ and $a_i \leq a_{\text{prev}}$ then
        $w' \leftarrow y$;
    else
        $p \leftarrow 2$;
        $w' \leftarrow w$;
    store current range entry in $h$;
    return $(w', h)$;
case $p = 2$ do
    if $w = c$ then
        $p \leftarrow 3$;
        $w' \leftarrow w$;
    else
        $w' \leftarrow$ next node accoding to tree routing scheme with $l(c; c, j)$;
    return $(w', h)$;
case $p = 3$ do
    if $w$ stores entry $l(v; c, j)$ for $l(v)$ then
        store $l(v; c, j)$ in $h$;
    $w' \leftarrow$ next node according to SearchTree-procedure;
    if $w' = c$ then
        $p \leftarrow 4$;
    return $(w', h)$;
case $p = 4$ do
    $w' \leftarrow$ next node accoding to tree routing scheme with $l(v; c, j)$;
    return $(w', h)$;
```

We see that we also maintain the packet header size of Konjevod et al.'s routing scheme. By Lemma 12, we know that one range entry has $\mathcal{O}(\log n)$ bits. The global routing label $l(v)$ takes $\lceil \log n \rceil$ bits. p requires $\mathcal{O}(1)$ bits. The local routing label $l(v; c, j)$ take $\mathcal{O}\left(\frac{\log^2 n}{\log \log n}\right)$ bits according to Fraigniaud and Gavoille [FG01] as well as Thorup and Zwick [TZ01]. We conclude:

Lemma 17. *The packet header size is* $\mathcal{O}\left(\frac{\log^2 n}{\log \log n}\right)$.

The other information required for executing the algorithm is stored in the routing tables:

- Range entrys (r, x, a_i, y, b, j, c) as described in preprocessing analysis
- Local routing labels $l(\_; c, j)$ for the routing scheme of Fraigniaud and Gavoille
- Information on virtual edges for the search trees

The steps of the modified algorithm are basically the same as in the original one. The main difference is the addition of range entrys, which store range information for global routing labels as well as results for some difficult operations of the original algorithm, e.g.

$$d(u_k, x_k) \geq \frac{2^{i_k - 1}}{\varepsilon} - 2^{i_k}$$

One important property for the correct execution of this algorithm is that a_i can substitute i. Due to Lemma 9, the loop-condition $i_k \leq i_{k-1}$ from the original algorithm is satisfied if and only $a_{i_k} \leq a_{i_{k-1}}$. Similarly, choosing the minimum i_k in the original algorithm leads to the same result as choosing the minimum a_i, since the other conditions concerning x_k (respectively x in the range entry) and $l(v) \in \text{Range}(x_k, i_k)$ (respectively $l(v) \in r$) are the same. This means that all nodes visited in phase 1 are the same as in the original algorithm. From there on, the modified algorithm chooses j and c by the same criteria as before and uses local tree routing and the search tree procedure just like in the original algorithm.

5 Conclusion

In our work, we have provided a more detailed description for Konjevod et al.'s labeled routing scheme. We have shown that it requires $\mathcal{O}(n^3 + \varepsilon^{-1} n^2 \log^2 n)$ preprocessing time for graphs in general and $\mathcal{O}(\varepsilon^{-1} n^2 \log^2 n)$ for UDGs. This is due to the fact that the SSSP-Problem (and thus, the APSP-Problem) can be solved more efficiently for UDGs.

Moreover, we reformulated the original routing algorithm to be recursive and to not require any global data structures, but to strictly use only information from the packet headers and the current node's routing table.

Of course, there is still a lot of room for possible improvements: Most importantly, the running time can probably be improved further. During the analysis, we made a few rough estimates to compute data structures or retrieve running times, some of which might not be optimal. For example, for storing range information in Section 3.2.2 and range entrys in Section 3.3.4, we simply considered all possible combinations of $u \in V$, $i \in R(u)$ and $x \in V$, leading to an unpleasant $\mathcal{O}(\varepsilon^{-1} n^2 \log^2 n)$ term for the range entrys. If we could find a better estimate or a better method for calculating this step, we could reduce our running time further.

Also, we used a rather naive method for solving the APSP-Problem for UDGs, that is, solving the SSSP-Problem n times. There is already a more efficient algorithm for the APSP-Problem in *unweighted* UDGs by Chan and Skrepetos [CS16], which has subquadratic running time $\mathcal{O}\left(n^2 \sqrt{\frac{\log \log n}{\log n}}\right)$. Additionally, we did not consider any of the other steps for UDGs in particular, but for general graphs with low doubling dimension. If we could find ways to adapt those too, we might achieve even better results.

In this thesis, we only focused on labeled routing schemes, i.e. routing schemes with the option to replace the orginal names of the nodes with custom names. However, routing schemes also can be name-independent. In the latter case, we do not have this option and have to use the original names. This has the advantage that we do not need to know the destination's label in advance, which could be a problem in some instances. On the other hand, the addition of routing labels can greatly improve the stretch of the routing scheme.

An open problem is the preprocessing time for name-independent routing schemes: Konjevod et al. also introduced a $(9 + \varepsilon)$-stretch name-independent routing scheme with similar space requirements to the labeled routing scheme we considered in our work. However, it is unclear how much time the preprocessing will take for that variant.

References

[BM86] Michael Becker and Kurt Mehlhorn. Algorithms for routing in planar graphs. *Acta Informatica*, 23(2):163–176, 1986.

[CCJ90] Brent N. Clark, Charles J. Colbourn, and David S. Johnson. Unit disk graphs. *Discrete Mathematics*, 86(1-3):165–177, 1990.

[CS16] Timothy M. Chan and Dimitrios Skrepetos. All-pairs shortest paths in unit-disk graphs in slightly subquadratic time. In *27th International Symposium on Algorithms and Computation (ISAAC 2016)*, pages 24:1–24:13, 2016.

[FG01] Pierre Fraigniaud and Cyril Gavoille. Routing in trees. In *Automata, Languages and Programming*, pages 757–772, 2001.

[Gav01] Cyril Gavoille. Routing in distributed networks: Overview and open problems. *SIGACT News*, 32(1):36–52, 2001.

[GT08] Cyril Gavoille and Andrew Twigg. Compact forbidden-set routing on planar graphs. *Unpublished as yet*, 2008.

[KRX16] Goran Konjevod, Andréa W. Richa, and Donglin Xia. Scale-free compact routing schemes in networks of low doubling dimension. *ACM Trans. Algorithms*, 12(3), 2016.

[PU89] David Peleg and Eli Upfal. A trade-off between space and efficiency for routing tables. *J. ACM*, 36(3):510–530, 1989.

[Tho04] Mikkel Thorup. Compact oracles for reachability and approximate distances in planar digraphs. *J. ACM*, 51(6):993–1024, 2004.

[TZ01] Mikkel Thorup and Uri Zwick. Compact routing schemes. In *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 1–10, 2001.

[Wil20] Max Willert. *Routing and Stabbing.* PhD thesis, Freie Universität Berlin, 2020.

[WX20] Haitao Wang and Jie Xue. Near-optimal algorithms for shortest paths in weighted unit-disk graphs. *Discrete & Computational Geometry*, 64:1141–1166, 2020.