Jagiellonian University
Department of Theoretical Computer Science

Adam Szwaja

Implementation of Single Source Shortest Path Algorithms for Planar Graphs

Bachelor's Thesis

Supervisor: dr hab. inż. Krzysztof Turowski

June 2025

Contents

Chapter 1

Introduction

1.1 Important notions

Before proceeding with the main analysis, it is important to define key terms and concepts that will be used throughout this work. This ensures clarity and consistency. This chapter introduces the primary definitions and theoretical background relevant to the topic of single source shortest path algorithms and planar graphs. Additional terms will be defined as they become significant in later chapters. The definition provided by this work are based on [2, 6, 12].

Definition 1.1 (weighted graph [12])**.** *A* ***weighted graph*** *is a graph in which each edge is assigned a numerical value, called a* ***weight****. Formally, a weighted graph is a triple* $G = (V, E, w)$*, where:*

- V *is nonempty set of vertices,*
- $E \subseteq \{\{u, v\} : u, v \in V, u \neq v\}$ *is the set of edges, and*
- $w : E \to \mathbb{R}^+ \cup \{0\}$ *is a function that assigns a real number (the weight) to each edge.*

For convenience, we will write $w(u, v)$ *to denote* $w(\{u, v\})$*.*

This type of graph is also called *undirected weighted graph.*

Definition 1.2 (directed weighted graph [12])**.** *A* ***directed weighted graph****, is a weighted graph where each edge has also an associated direction. It is defined as a triple* $G = (V, E, w)$*, where:*

- V *is nonempty set of vertices,*
- $E \subseteq V \times V$ *is a set of ordered pairs representing directed edges, and*
- $w : E \to \mathbb{R}^+ \cup \{0\}$ *assigns a real-valued weight to each directed edge.*

For convenience, we will write $w(u, v)$ *to denote* $w((u, v))$*.*

It is worth noting that, in the context of shortest path algorithms, directed graphs are a more general case than undirected graphs. Any undirected graph can be transformed into a directed graph by replacing each undirected edge $\{u, v\}$ with two directed edges (u, v), (v, u) and updating the weight function accordingly.

Definition 1.3 (adjacency, neighbour [2])**.** *Two vertices u, v of a graph G are **adjacent** or **neighbours**, if $\{u, v\}$ is an edge of G. The set of all neighbours of a vertex v is denoted by $N(v)$. More generally for $U \subseteq V$ the neighbours in $V \setminus U$ of vertices in U are called **neighbours of** U and their set is denoted by $N(U)$.*

Definition 1.4 (degree of a vertex [12])**.** *The **degree of a vertex** in an undirected graph is the number of edges incident with it. The degree of the vertex v is denoted by* $\deg(v)$.

Definition 1.5 (in-degree and out-degree of a vertex [12])**.** *In a directed graph, the **in-degree** of a vertex is the number of edges directed towards it, and the **out-degree** is the number of edges directed away from it.*

Definition 1.6 (induced graph [2])**.** *If $G' \subseteq G$ and G' contains all the edges $\{u, v\} \in E$ with $u, v \in V'$, then G' is an **induced subgraph** of G; we say that V' induces G' in G. We will denote as $G[V']$ a subgraph induced in G by V'*

Definition 1.7 (cluster [7])**.** *A **cluster** in a graph G is a set of vertices S such that the induced subgraph $G[S]$ is connected.*

The definitions introduce fundamental concepts that are essential for understanding the structure of graphs and their components. We now proceed to notions that are specific to planar graphs and their geometric representations.

Definition 1.8 (plane graph [2])**.** *A **plane graph** is a pair (V, E) of finite sets with the following properties plane graph (the elements of V are again called vertices, those of E edges):*

- $V \subseteq R^2$,
- *every edge is an arc between two vertices,*
- *different edges have different sets of endpoints,*
- *the interior of an edge contains no vertex and no point of any other edge.*

Definition 1.9 (planar embedding [2])**.** *An **embedding** in the plane, or **planar embedding**, of a graph G is an isomorphism between G and a plane graph G'. The latter will be called a **drawing** of G.*

*This embedding induces a **cyclic order** of the edges incident to each vertex, corresponding to the order in which these edges are arranged around the vertex in the plane. The collection of these cyclic orders for all vertices defines the **rotation system** of the embedding, which uniquely determines the embedding up to homeomorphism.*

Definition 1.10 (planar graph [2]). *A **planar graph**, is a graph for which exists planar embedding. A graph is called planar if it can be embedded in the plane. A planar graph is **maximal**, or **maximally planar**, if it is planar but cannot be extended to a larger planar graph by adding an edge.*

We are now ready to define the main problem addressed in this work.

Definition 1.11 (single source shortest path (SSSP) for planar graphs [6]). *Given a weighted (directed or undirected) planar graph $G = (V, E, w)$ and a source vertex $s \in V$, the **single source shortest path (SSSP)** problem requires to find the shortest distances from s to every other vertex $v \in V$. That is, for each $v \in V$, algorithm needs to compute the minimum total weight of any path from s to v in G.*

An important aspect of SSSP algorithms for planar graphs is leveraging planarity of the graph, primarily by dividing it into appropriate regions. The following definitions will help in understanding the concepts and techniques used in this approach.

Definition 1.12 (region, boundary, interior [6]). *A **graph region** is a subset of vertices of a graph such that if a vertex belongs to the region, then all its neighbors also belong to the same region. A vertex that belongs to multiple regions is called a **boundary vertex**, while a vertex that belongs to exactly one region is called an **interior vertex**. Additionally set of all boundary vertices of a region R will be denoted by $\mathcal{B}(R)$ and set of all interior vertices will be denoted by $\mathcal{I}(R)$.*

Definition 1.13 (division [6]). *A **graph division** is a collection of regions such that every vertex belongs to at least one region, and every edge is entirely contained in at least one region (i.e., both its endpoints belong to the same region).*

In order to simplify the analysis and improve algorithmic efficiency, we apply a standard transformation see Figure 1.1 that reduces the maximum degree of the graph to 3 (and maximum in/out degree to 2). The transformation for directed graph proceeds as follows. For each vertex v of degree $d = \deg(v) > 3$ where $u_0, ..., u_{d-1}$ is a cyclic ordering of the vertices adjacent to v in the planar embedding, replace v with new vertices $v_0, ..., v_{d-1}$. Add edges $(v_i, v_{i+1 \mod d})$ for $i = 0, ..., d-1$, with $w(v_i, v_{i+1 \mod d}) = 0$, and replace the edges of the form (v, u_i) with (v_i, u_i) or (u_i, v) with (u_i, v_i), preserving the original edge weights. [6]

The transformation for undirected graphs is essentially the same, with the only difference being that the edges are not directed.

It is important that this transformation introduces at most $2m - n \leq 5n - 12$ new vertices, where n is the number of vertices and m is the number of edges in the original graph. This bound follows from Euler's formula for planar graphs and ensures that the size of the transformed graph remains linear in n, preserving the asymptotic complexity of the algorithm.

In this paper, we assume, without loss of generality, that all graphs have bounded degree.

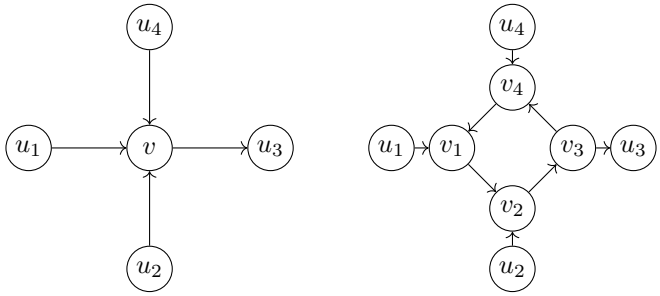

Figure 1.1: An example of degree transformation for vertex v

1.2 Background and motivation

The main goal of this thesis is to provide a clear and accessible presentation of the most important techniques and ideas behind the algorithms for the single source shortest path problem in planar graphs. By summarizing and comparing different approaches, we aim to give the reader a deeper understanding of how exploiting the structure of planar graphs can lead to significant algorithmic improvements.

Planar graphs possess several theoretical properties that are essential for designing efficient algorithms. For instance, planar graphs are sparse: the number of edges is $O(n)$, as follows from Euler's formula [14]. Moreover, by the Planar Separator Theorem [11], for any planar graph on n vertices, it is possible to find a set of $O(\sqrt{n})$ vertices whose removal separates the graph into smaller, roughly balanced parts. These structural properties will be frequently used throughout this paper.

The single-source shortest path problem is one of the most fundamental and important problems in the family of planar graphs. The solutions presented in this paper may be used to improve the time complexity of various algorithms, such as finding feasible flows and matching in bipartite planar graphs.

The standard algorithm for finding shortest paths in weighted graphs is Dijkstra's algorithm. Its implementation using heap achieves time complexity of $O(n \log n)$ [8].

In this paper, we will present two algorithms for the single-source shortest path (SSSP) problem in planar graphs: the first is proposed by Greg Frederickson in 1986 [6], achieving time complexity of $O(n\sqrt{\log n})$, and the second is a simplified version of algorithm proposed by Monika Henzinger [9], which achieves time complexity of $O(n \log \log n)$. Both of these algorithms are based on Dijkstra algorithm, but they additionally exploit properties of planar graphs to significantly reduce asymptotic running time. Currently, the best-known algorithm and also the one that achieves theoretical lower bound for finding the shortest path to every node in planar graphs is the SSSP algorithm from [9], which runs in $O(n)$ time.

We will now briefly describe the Dijkstra algorithm to provide necessary in-

tuition for understanding the proofs presented later in this paper. The main idea behind Dijkstra's algorithm serves as a foundation for more complex algorithms for planar graphs.

Dijkstra's algorithm computes the shortest path from a single source vertex s to all other vertices in a weighted graph with non-negative edge weights. The algorithm maintains a set of vertices whose shortest distance from the source has already been determined, and for every other vertex v, it keeps track of the current best known distance $p(v)$ from s to v. Initially all vertices are open, $p(s) = 0$ and $p(v) = \infty$ for all $v \neq s$. Algorithm in each iteration closes the open vertex v with the smallest $p(v)$ and relaxes all edges outgoing from v. That is, for every neighbor w of v, the algorithm updates $p(w)$ if a shorter path from s to w via v is found.

The distances $p(v)$ can be maintained in a heap, which allows for $O(\log n)$ time per update. Since there are $O(n)$ heap updates, and each edge is relaxed at most once, the total running time is $O(n \log n)$ [8].

One approach to reducing the asymptotic running time of shortest path algorithms is to minimize the maximum number of nodes present in the heap at any given time. This means that a preprocessing phase is necessary to identify this subset of vertices a heap will be used on. How can we exploit properties of planar graphs to achieve this goal? The key idea shared by both of the algorithms presented in this paper is to partition the graph into appropriate sized regions during the prepossessing phase. In Chapter 2, we will describe the specific properties that such a division must have in order to take advantage of the structure of planar graphs, as well as how to construct this partition efficiently. Once the partitioning is completed, the algorithm must carefully coordinate heap updates between the regions to avoid increasing the total number of heap operations. In Chapters 3 and 4, we will explain the approaches taken by each of the two algorithms in detail.

Chapter 2

Graph Division

In this chapter, we will present and formalize the key properties of planar graph regions that will be exploited in the following chapter. We will also describe efficient methods for obtaining such a division.

Definition 2.1 (r-division [6])**.** *We call* ***r-division*** *a graph division into* $\Theta(n/r)$ *regions of* $O(r)$ *vertices each and* $O(\sqrt{r})$ *boundary vertices each.*

This definition only restricts the graph division in terms of region size and number of boundary vertices, but these criteria do not guarantee a desirable structure for the regions. There are two main issues with defining division in this way. Firstly, a single boundary vertex may be shared by more regions than its degree. Secondly, a single region might consist of multiple disconnected parts scattered throughout the entire graph. To not lose the benefits of graph planarity we introduce stricter definition of graph division.

Definition 2.2 (suitable r-division [6])**.** *A* ***suitable r-division*** *of a planar graph is an r-division that also satisfies:*

- *every boundary node is at most in three different regions,*
- *every region is either connected component or union of connected components that share boundary nodes with the same set of either one or two regions that are connected components (see Figure 2.1).*

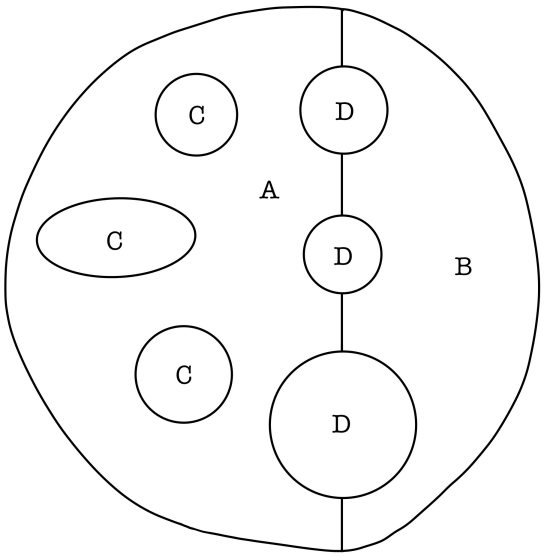

Figure 2.1: Examples of regions that are unions of connected components [6].

2.1 Recursive division

To find r-division implementation of Planar Separator Theorem [11] will be used. The separator algorithm given graph G with weighted vertices, partitions the vertices of G into three sets A, B and C (the separator). The partition is such that no edge joins a vertex in set A with a vertex in set B. Moreover the sum of weights in sets A and B does not exceed $2/3$ of the sum of weights in the entire graph. Algorithm also ensures that set C is small, i.e. it does not contain more than $O(\sqrt{n})$ vertices. For the purpose of this analysis if not stated differently the nodes will have equal weight. This means that the sum of weights can be treated as number of vertices in a given set.

There are several known implementations of the Planar Separator algorithm, each providing different constants for the separator size. For the purposes of this work, it suffices to use any separator of size $O(\sqrt{n})$. However, for completeness, we present some of these variants below.

Algorithm	Separator size	Time
Lipton, Tarjan [11]	$\leq 2\sqrt{2n} = 2.83\sqrt{n}$	$O(n)$
Alon, Seymour and Thomas [1]	$\leq 2.13\sqrt{n}$	$O(n)$
Djidjev and Venkatesan [5]	$\leq 1.97\sqrt{n}$	$O(n)$
Fundamental Cycle Separator [10]	$O(d)$ d is diameter of a graph	$O(n)$

Table 2.1: Example of planar separator implementations

The best known lower bound for the size of a planar separator is $1.55\sqrt{n}$ [4].

Although the separator size produced by the Fundamental Cycle Separator algorithm can be as large as $O(n)$ in the worst case, this approach performs very well in practice and is easy to implement, which is why it is often chosen in practical applications.

Our algorithm computing r-division consists of two parts. The first part starts by obtaining a planar graph division with $O(n/r)$ regions of size $O(r)$ and in total $O(n/\sqrt{r})$ boundary vertices using Algorithm 1. The second part builds on top of that further dividing regions ensuring appropriate number of boundary vertices per region.

Algorithm 1 RecursivePartition

input: Planar graph $G = (V, E)$, region size parameter r
output: Partition of G into regions of size at most r

```
1: procedure RecursivePartition(G, r)
2:     R ← {V}
3:     while exists S ∈ R that |S| ≥ r do
4:         R ← R \ {S}
5:         A, B, C ← Separator(G[S])
6:         R ← R ∪ {A ∪ C, B ∪ C}
7:     end while
8:     return R
9: end procedure
```

Lemma 2.1.1 (Lemma 1 in [6])**.** *The planar graph on n vertices can be divided into $O(n/r)$ regions of size $O(r)$ with $O(n/\sqrt{r})$ boundary vertices in $O(n \log n/r)$.*

Proof. The time complexity claim can be proven by analyzing the recursion tree of Algorithm 1. Since the recursion stops when regions reach size $O(r)$, the process terminates $\log r$ levels before reaching the leaves. As a result, the recursion tree has $\log n - \log r = \log(n/r)$ levels, each requiring $O(n)$ work. It is obvious that procedure described above will create $O(n/r)$ regions of size $O(r)$. We need to now prove that the number of boundary nodes is appropriate. By $b(v)$ we will denote one less than the number of regions that v is contained. Let $B(n, r) = \sum_{v \in V} b(v)$. Intuitively, we can treat $B(n, r)$ as number of copies of all boundary vertices in a division. By applying bounds for resulting sets size and number of boundary vertices we get the following recurrence:

$$B(n,r) \leq 2\sqrt{2n} + B(\alpha n + O(\sqrt{n}), r) + B((1-\alpha)n + O(\sqrt{n}), r) \text{ for } n > r,$$
$$B(n,r) = 0 \text{ for } n \leq r,$$

where $1/3 \leq \alpha \leq 2/3$. Intuitively, the number of new copies in a division is bounded both by the separator size and subsequent copies added by recursive divisions.

We will prove by induction that for some constant $d > 0$ it holds that:

$$B(n,r) \leq 2\sqrt{2}\frac{n}{\sqrt{r}} - d\sqrt{n}.$$

By applying the inductive step and summing up, we get:

$$\begin{aligned} B(n,r) &\leq 2\sqrt{2n} + 2\sqrt{2}\frac{\alpha n}{\sqrt{r}} + 2\sqrt{2}\frac{(1-\alpha)n}{\sqrt{r}} + O\left(\frac{\sqrt{n}}{\sqrt{r}}\right) - d(\sqrt{\alpha n} + \sqrt{(1-\alpha)n}) \\ &= 2\sqrt{2n} + 2\sqrt{2}\frac{n}{\sqrt{r}} + O\left(\frac{\sqrt{n}}{\sqrt{r}}\right) - d(\sqrt{\alpha n} + \sqrt{(1-\alpha)n}) \\ &\leq 2\sqrt{2}\frac{n}{\sqrt{r}} - d\sqrt{n} \end{aligned}$$

By choosing d large enough, the gap between $-d(\sqrt{n})$ and $-d(\sqrt{\alpha n}+\sqrt{(1-\alpha)n})$ will dominate the additive term $2\sqrt{2n}+O\left(\frac{\sqrt{n}}{\sqrt{r}}\right)$ for large n, and the error terms can be absorbed into the main inequality. □

Let us assume that we have a division with $O(n/r)$ regions of size $O(r)$ and a total of $O(n/\sqrt{r})$ boundary vertices. We now state that from some constant c the following algorithm returns an r-division.

Algorithm 2.1.1
For every region with more than $c\sqrt{r}$ boundary vertices, we recursively apply the following procedure: Assign a weight of 1 to each boundary vertex and a weight of 0 to each interior vertex. Then, apply the separator algorithm and partition the region into two subregions, $A_1 = A \cup C$ and $A_2 = B \cup C$.

By using the separator algorithm with weights chosen in this way, we ensure that each resulting region contains at most $2b/3 + 2\sqrt{2r}$ boundary vertices, where b is the number of boundary vertices in the input region.

By combining both parts, we obtain the following lemma.

Lemma 2.1.2 (Lemma 2 in [6])**.** *A planar graph of n vertices can be divided into an r-division in $O(n \log n)$ time.*

Proof. The above algorithm's recurrence relation takes the form

$$T(n) = T(\alpha n + O(\sqrt{n})) + T((1-\alpha)n + O(\sqrt{n})) + O(n)$$

. Time complexity claim can be shown by simple analysis of recursion tree. We get $O(\log n)$ levels of size $O(n)$. Due to the nature of algorithm resulting regions size is obviously $O(r)$ with at most $O(\sqrt{r})$ boundary vertices per region. What's left to show is that the overall number of regions and boundary vertices did not increase over bounds stated in Definition 2.1.

Let us consider a division obtained from the first part of the algorithm. Let t_i be number of regions with i boundary vertices. Using the notation from the previous proof $b(v) + 1$ is a number of regions to which v belongs and it

follows $\sum_i it_i = \sum_{v \in V_b}(b(v)+1)$ where V_b is a set of all boundary vertices. $\sum_{v \in V_b}(b(v)+1) < \sum_{v \in V_b} 2b(v) = 2B(n,r)$ which is $O(n/\sqrt{r})$. Any region with $i > c\sqrt{r}$ boundary vertices will require at most $di/c\sqrt{r}$ splits, which will result in at most $1 + di/c\sqrt{r}$ smaller regions for some constants c and d. This follows from the fact that every split of a region with z boundary vertices results in two regions, each containing at most $2z/3 + 2\sqrt{2}\sqrt{r}$ boundary vertices. By choosing the constant c large enough, any region will require at most $O(\log n)$ splits to achieve the desired boundary size. Analyzing the recursion tree of this division allows us to bound the total number of splits.

Every split adds at most $2\sqrt{2}\sqrt{r}$ new boundary vertices. It gives us upper bound for number of new vertices:

$$\sum_i (c\sqrt{r})(di/c\sqrt{r})t_i \leq d\sum_i it_i = O(n/\sqrt{r})$$

and upper bound for number of new regions:

$$\sum_i di/(c\sqrt{r})t_i = O(n/r).$$

□

2.2 Suitable graph division

Now that we have established that a planar graph can be divided into an r-division, we proceed to analyze an algorithm for constructing a suitable r-division. A modification of the standard implementation of the Planar Separator Theorem in Algorithm 2 will ensure that the resulting regions align with the criteria outlined in Definition 2.2.

Algorithm 2 FIND CONNECTED SUBSETS

input: set S of vertices, graph G,
output: sets of vertices inducing connected subgraphs

```
1:  procedure FindConnectedSubsets(G, S)
2:      Get A, B, C by running planar separator algorithm on G[S]
3:      C' ← {v ∈ C : N(v) ∩ ( A ∪ B) = ∅}
4:      C'' ← C \ C'
5:      (V_1, E_1), ..., (V_q, E_q) ← ConnectedComponents(G[A ∪ B ∪ C'])
6:      for each v ∈ C'' do
7:          S ← {j : N(v) ∩ V'_j ≠ ∅}
8:          if |S| = 1 then
9:              V'_i ← A'_i ∪ {v}                          ▷ where {i} = S
10:         end if
11:     end for
12:     return {V_1, V_2, ..., V_q}
13: end procedure
```

There are two main benefits to this approach. First, the induced subgraphs of the returned sets are connected. Second, a boundary vertex is included in a region if and only if it is adjacent to an interior node of that region.

The algorithm for finding suitable r-division starts with recursively applying Algorithm 2 to regions with more than r vertices and more than $c\sqrt{r}$ boundary vertices, just like in Algorithm 1.

Algorithm 3 FindSuitableRDivision

input: Planar graph G, parameter r, c
output: Partition of G into regions of size at most r

procedure FindSuitableRDivision(G, r)
  $R \leftarrow \{V\}$
  while exists $S \in R$ that $|S| \geq r$ **or** $|\mathcal{B}(S)| \geq c\sqrt{r}$ **do**
    $R \leftarrow R \setminus \{S\}$
    $\{V_1, V_2, \ldots, V_q\} \leftarrow$ FindConnectedSubsets(G, S)
    $R \leftarrow R \cup \{V_1, V_2, \ldots, V_q\}$
  end while
  return Merge(R, r, c)
end procedure

It is important that Algorithm 2 returns division with regions smaller in size and with less or equal number of boundary vertices to division returned by Separator Algorithm. This means that time complexity does not increase. Unfortunately, more that $O(n/r)$ regions can be created using this procedure, hence post processing phase that merges resulting regions is necessary to provide suitable r-division from Definition 2.2.

Algorithm 4 Merge

input: Set of regions R, parameters r, c
output: Updated set of regions after merging

while exist $R_1, R_2 \in R$ with $\mathcal{B}(R_1) \cap \mathcal{B}(R_2) \neq \emptyset$ **and** $|R_1|, |R_2| < r/2$ **and** $|\mathcal{B}(R_1)|, |\mathcal{B}(R_2)| < c/2\sqrt{r}$ **do**
  $R \leftarrow R \setminus \{R_1, R_2\}$
  $R_{new} \leftarrow R_1 \cup R_2$
  $R \leftarrow R \cup \{R_{new}\}$
end while
while exist $R_1, R_2 \in R$ with $N(R_1) = N(R_2)$ **and** $|N(R_1)| \in \{1, 2\}$**and** $|R_1|, |R_2| < r/2$ **and** $|\mathcal{B}(R_1)|, |\mathcal{B}(R_2)| < c/2\sqrt{r}$ **do**
  $R \leftarrow R \setminus R_1, R_2$
  $R_{new} \leftarrow R_1 \cup R_2$
  $R \leftarrow R \cup \{R_{new}\}$
end while
return R

One way to merge components in $O(n)$ time is to iterate over boundary ver-

tices and merge regions that share a current vertex and satisfy criteria. Further implementation details no how to merge regions in $O(n)$ time will be discussed in Chapter 5.

This gives us:

Theorem 2.2.1. *[Theorem 1. in [6]] A planar graph of n vertices can be divided into a suitable r-division in $O(n \log n)$ time.*

Proof. We claim that Algorithm 3 described above generates suitable r-division in correct time complexity. The time complexity claim, the size of regions and number of boundary vertices in each region, can be easily proven using exactly the same approach as in proof of Lemma 2.1.2. What's left to show is that the number of regions in resulting division after merging does not exceed $O(n/r)$.

Consider a *region graph*. Region graph is graph that meets following properties. There is one node per region and two nodes are adjacent to one another in region graph if and only if two regions share common boundary vertex. The region graph is obviously planar. We will consider two types of nodes:

- *small node* - represents region with either less than r vertices or less than $c\sqrt{r}$ boundary vertices,
- *normal node* - represents other regions.

From the foregoing procedure the following properties will hold. Small nodes are only adjacent to normal nodes. For every normal node there can be only one small node of degree 1 adjacent to it, thus the number of small nodes of degree 1 is proportional to the number of normal nodes.

For every pair of normal nodes there can be only one small node of degree 2 adjacent to both normal nodes. Any small node of degree 2 can be replaced by an edge without breaking region graph's planarity, hence number of small nodes of degree 2 is proportional to the number of normal nodes.

To bound number of all small consider region graph with small nodes of degree less than 3 removed. Consider its planar embedding and add all possible edges between two normal nodes to not break planarity. Than remove edges adjacent to small nodes. Now there is at most one small node for every face of a planar embedding making it proportional to the number of normal nodes. This concludes that in resulting division there is $O(n/r)$ regions. The time complexity claim follows as time for applying Algorithm 2 recursively is $O(n \log n)$ and time to perform region merging is $O(n)$.

□

2.3 Quick suitable graph division

In the previous section, we proved that it is possible to compute a suitable r-division in $O(n \log n)$ time. For the algorithms presented in this paper to achieve their desired time complexity, the preprocessing phase that divides the graph into regions must take no more time than the main part of the algorithm itself. This section focuses on reducing the complexity of the division algorithm.

2.3.1 Clusters

The main idea presented in [6] is to reduce input size to the suitable r-division algorithm from Theorem 2.2.1. We will shrink input graph on clusters resulting from Algorithm 6 described below.

As a reminder we operate under the assumption that graph has been modified to have degree of every vertex to be at most 3.

Algorithm 5 CSearch [7]

input: Graph G, current node v, cluster size z
output: Last cluster with size less than z

```
1: global R                                          ▷ see Algorithm 6
2: procedure CSearch(G,v, z)
3:     C ← {v}
4:     for each child w of v do
5:         C ← C∪ CSearch(G, w, z)
6:     end for
7:     if |C| < z then
8:         return C
9:     else
10:        R ← R ∪ {C}                               ▷ saves cluster C to R
11:        return ∅
12:    end if
13: end procedure
```

Algorithm 6 FindClusters [7]

input: Graph G, cluster size z
output: Clusters of G with size $O(z)$

```
1: procedure FindClusters(G, z)
2:     R ← ∅
3:     lastCluster ← CSearch(G,any vertex of G, z, R)
4:     if lastCluster ≠ ∅ then
5:         L ← last saved cluster in R
6:         R ← R \ {L}
7:         R ← R ∪ {L ∪ lastCluster}
8:     end if
9:     return R
10: end procedure
```

Lemma 2.3.1. *A* FindClusters *procedure given integer z and a graph G on n vertices will divide vertex set of G into clusters of $O(z)$ size. The whole procedure will take $O(n)$ time.*

Proof. The procedure is based on depth-first search, so for a planar graph it runs in $O(n)$ time. The CSearch procedure can return a set containing at

most $z-1$ vertices. Since graph G has bounded degree, any cluster saved by the procedure can have size at most $3z-2$. That is, the union of three neighboring clusters plus the root node. Thus, for any given vertex, the largest saved cluster is of size $O(z)$. The last cluster saved may be a union of two clusters. This guarantees that the subgraph induced by the last cluster is connected, since both *lastCluster* and L are connected and does not affect the asymptotic bound. □

2.3.2 Suitable r-division algorithm improvement

Now we can present the improved algorithm for suitable r-division.

Algorithm 7 FindSuitableRDivisionQuickly

input: Planar graph $G = (V, E)$, region size parameter r
output: Suitable r-division of G

1: $A_1, ..., A_k \leftarrow$ FindClusters(G, $\sqrt{r}$)
2: $G_s \leftarrow$ ShrinkClustersToVertices(G, $A_1, ..., A_k$)
3: $R_1, ..., R_L \leftarrow$ FindSuitableRDivision(G_s, r)
▷ $|R_i| = O(r)$ and $L = O(n/r^{3/2})$
4: $B_1, ..., B_l, I_1, ..., I_m \leftarrow$ ExpandVertices($R_1, ..., R_L$,G, $A_1, ..., A_k$)
▷ $|B_i| = O(\sqrt{r})$, $l = O(n/r^{3/2})$ from boundary vertices of G_s and $|I_i| = O(r^{3/2})$, $m = O(n/r^{3/2})$ from interiors of R_i
5: Infer boundary vertices and expand regions $B_1, ..., B_l, I_1, ..., I_m$ that share these boundary vertices
6: *division* $\leftarrow$ FindSuitableRDivision($B_1, ..., B_l, I_1, ..., I_m$, r)
▷ Starting from $\{B_1, ..., B_l, I_1, ..., I_m\}$ instead of set of vertices $\{V\}$
7: **return** *division*

Theorem 2.3.2 (Lemma 4. in [6])**.** *A suitable r-division of graph on n vertices can be found in $O(n \log r + (n/\sqrt{r}) \log n)$ time.*

Proof. We claim that the Algorithm 7 generates required division in a given time. The time complexity claim:

It takes $O(n)$ to shrink graph G into G_s and

$$O((n/\sqrt{r})(\log(n/\sqrt{r})) = O((n/\sqrt{r}) \log n)$$

time to get suitable r-division of graph G_s. The boundary vertices can be inferred by iterating over the set of edges of graph G, thus it takes $O(n)$ time. To expand graph G_s back to G also takes $O(n)$ time. And finally to divide remaining sets of size $O(r^{3/2})$ it takes $O(n \log r)$ time which in total gives us $O(n \log r + (n/\sqrt{r}) \log n)$.

The original proof in [6] lacked one step: bounding the number of boundary vertices after ExpandVertices step. The number of boundary vertices can be bounded by number of vertices in $O(n/r)$ regions of size $O(\sqrt{r})$. This gives as a total of $O(n/\sqrt{r})$ new induced boundary vertices. It is also worth noting that further dividing the regions $O(n/r^{3/2})$ regions of size $O(r^{3/2})$ will only result

in $O(r^{3/2}/\sqrt{r}) = O(r)$ vertices per region which gives in total $O(n/\sqrt{r})$ new vertices. $\square$

Chapter 3

Frederickson's Single Source Shortest Path Algorithm

In this chapter, we will describe Frederickson's SSSP algorithm[6]. The algorithm runs in $O(n\sqrt{\log n})$ time and utilizes a clever data structure-the Topology-Based Heap-to efficiently handle updates on a substantially reduced set of vertices. In contrast to Henzinger's SSSP algorithm, all vertices are stored in a single heap-like structure. To achieve the required update times, the structure is constructed to satisfy additional criteria enforced during the initialization stage.

3.1 Topology based heap

Definition 3.1 (boundary set)**.** *A* ***boundary set*** *is a maximal subset of boundary nodes such that every member of a set is shared by exactly the same set of regions.*

The structure will provide two main operations:

- look up of the boundary vertex with smallest current value in $O(1)$ time,
- update of all vertices in boundary set S called *batch-update*. The required time will be described later.

But first, we need to identify all boundary sets. The Algorithm 8 identifies all boundary sets by creating for every boundary node a sorted list of all regions that contains it. After sorting all the lists using linear sorting algorithm e.g. *radix-sort*, boundary sets can be identified as groups of identical lists.

Note that every boundary vertex is shared by at most three regions, so the entire procedure takes $O(n)$ time.

Now, given all boundary sets, we can define the structure. The topology-based heap is represented as a balanced binary tree, in which the values associated with boundary nodes are stored at the leaves. The values corresponding

Algorithm 8 FindBoundarySets

input: Partition of graph into regions R
output: Boundary sets for all boundary vertices

```
1: R_1, ..., R_k ← R                                   ▷ number all regions
2: for each boundary vertex v do
3:     S[v] ← sorted list of numbers associated with regions that contain v
4: end for
5: Sort a set of all lists S using radix sort
6: lastList ← ∅
7: id ← 0
8: for each S[v] ∈ S do
9:     if S[v] ≠ lastList then
10:        id ← id + 1
11:        lastList ← S[v]
12:    end if
13:    B_id ← B_id ∪ {v}
14: end for
15: return B_1, ..., B_l
```

to a particular boundary set are stored in consecutive leaves, in the order determined by the sorted list in Algorithm 8. Thus, the entire initialization procedure can be performed in $O(n)$ time. For the purpose of Algorithm 9 parent of an node v in binary tree is denoted as $parent(v)$.

To perform a *batch update* on some boundary set S, we update all leaves associated with S. Then, moving layer by layer up the tree, we update the corresponding interior nodes. It is easy to see that this procedure will modify at most $\log n + 2|S|$ nodes. This follows from the fact that there are at most $|S|$ interior nodes in the tree whose descendants correspond to nodes in the boundary set. The only other affected nodes are the interior nodes adjacent to ancestors of the boundary set nodes, and there are at most two such nodes per tree level.

We can now describe how the heap is used during the main phase of the algorithm. The algorithm requires the distances between every pair of boundary vertices within each region. These distances can be precomputed before the main phase begins.

Before main thrust we initialize $d[v] = \infty$ for all boundary vertices and $d[s] = 0$ for the source node s. We also calculate distances from s to all boundary nodes that share region with s using Dijkstra algorithm on subgraph induced by region of s.

Given the preprocessing step where all necessary distances between boundary vertices within each region are precomputed, the main phase of the algorithm can then proceed as follows. The topology-based heap is initialized with the distances from the source to the boundary vertices in the region containing the source. Then while there exists an open vertex v we process it similarly

Algorithm 9 BatchUpdate

input: Topology-based heap H, boundary set S, node v, distances $dist(v, w)$ for all $w \in S$ to v
output: Updates all values in H for S and propagates changes up the tree

$P \leftarrow \emptyset$ ▷ queue of parents
for each leaf node w in H corresponding to $w \in S$ **do**
    $value \leftarrow H[w]$
    if $d[v] + dist(v, w) < value$ **then**
        $H[w] \leftarrow \min(d[v] + dist(v, w), value)$
        $P.push(parent(w))$
    end if
end for
while P is not empty **do**
    $v \leftarrow P.front()$
    $H.fixHeap(v)$
    $P.remove(v)$
    $P.push(parent(v))$
end while

Algorithm 10 MainThrust

input: Precomputed distances $dist(\cdot, \cdot)$, source vertex s, graph G
output: Array of shortest distances from s to all boundary vertices

for each boundary vertex v **do**
    $d[v] \leftarrow \infty$
end for
$d[s] \leftarrow 0$
$H \leftarrow initialize(d)$ ▷ topology-based heap
for each region R containing s **do**
    for each boundary set $S \subseteq R$ **do**
        BatchUpdate(H, v, S)
    end for
end for
while $H.minValue() \neq \infty$ **do**
    $v \leftarrow H.minVertex()$ ▷ vertex with minimal value in heap
    $d[v] \leftarrow H.minValue()$
    for each region R containing v **do**
        for each boundary set $S \subset R$ **do**
            BatchUpdate(H, v, S)
        end for
    end for
end while
return d

to Dijkstra's algorithm, we perform batch-updates for every boundary set of each region that contains v. In each batch-update, the values associated with boundary vertices are updated using the precomputed distances and the current value of $d[v]$, in a manner analogous to relaxing edges in Dijkstra's algorithm. Than vertex v is closed.

Importantly, for each pair of a vertex v and a boundary set S, a batch-update will be performed at most once, since v is marked as closed after all its updates have been completed. This observation is crucial for proving time complexity of Algorithm 10

After the main phase is complete, the shortest distances from the source to all boundary vertices are known. To obtain the shortest distances to all other vertices, a final "mop-up" step is performed in each region, using the known distances to the boundary vertices as starting values for the Dijkstra algorithm.

It is a well-know fact that for a shortest path algorithm on w graph G with a source vertex s to be correct 3 conditions must hold.

- $d[s] = 0$ for the source vertex s.
- For every vertex $v \in V$, $d[v]$ must be an upper bound of and actual distance from s to v
- For every edge $\{u, v\}$ two inequalities $d[v] \leq d[u] + w(u, v)$ and $d[u] \leq d[v] + w(u, v)$ hold.

Lemma 3.1.1. *The main thrust of the algorithm, given correctly precomputed distances in each region, will compute the correct distances from the source s to all boundary nodes.*

Proof. This claim can be easily proven by constructing an appropriate graph G', and observing that all three conditions hold.

The construction of G' is as follows, let G' be the subgraph induced by all boundary nodes and the source s. For every pair of vertices for which a distance was precomputed, we add an edge with the corresponding weight. Specifically, we add edges from the source s to all boundary nodes in the region containing s, and, for each region, we add edges between every pair of its boundary nodes with weights equal to the precomputed distances.

The first condition is satisfied, since $d[s] = 0$ is set at the beginning of the algorithm. The second condition can be easily proven by induction on the steps of the algorithm. By the induction hypothesis, $d[v]$ is an upper bound on the distance from s to v. After any batch update of a boundary set S, for every $w \in S$ the value in the heap is set to $d[v] + dist(v, w)$, so the upper bound property is maintained.

To prove that the third condition holds, assume for the sake of contradiction that after the algorithm terminates, there exists an edge (u, v) such that

$$d[u] > d[v] + w(u, v).$$

Consider the moment when vertex v was processed (i.e., removed from the priority queue). At that point, the value $d[v]$ was finalized and was the smallest

among all values in the queue. Since u is adjacent to v, the value of $d[u]$ would have been set to at most $d[v] + w(u, v)$ during the update step, contradicting our assumption. Thus, the third condition is also satisfied.

□

Theorem 3.1.2 (Theorem 2. in [6])**.** *Let an planar graph with n vertices be divided into a suitable r-division. Using the associated topology-based heap, the main thrust of Frederickson's SSSP algorithm will perform set of batch-update operations which cost $O(n + (n/\sqrt{r}) \log n)$.*

Proof. Since there are at most $O(\sqrt{r})$ boundary vertices in each region, and each boundary set can belong to at most three regions, a batched update can be performed on any given boundary set at most $O(\sqrt{r})$ times. Therefore, the total work for all batched updates associated with a single boundary set B is $O(\sqrt{r} \log n + \sqrt{r}|B|)$.

The number of boundary sets can be easily estimated using the fact that it is less than the sum of the number of edges and the number of faces in the planar region graph. This yields a total of $O(n/r)$ boundary sets. Thus, the overall cost of the first term is $O((n/\sqrt{r}) \log n)$. Since the total number of boundary vertices is $O(n/\sqrt{r})$, the combined cost of the second term over all boundary sets is $O(n)$. □

3.2 Main search

The Frederickson's SSSP algorithm for planar graphs consists of three separate parts:

1. division,
2. distance preprocessing,
3. main search.

It will also make use of two parameters $r_1 = \log n$ and $r_2 = (\log \log n)^2$ describing sizes of regions on different levels of division.

3.2.1 Nested graph division

The first parts is to divide the graph into level 1 regions of size r_1 using the strategy (see Algorithm 7) described in Chapter 2. Afterwards we divide every level 1 region into regions of size r_2. Call each region in this division a level 2 region. When generating level 2 this division, we start with each level 1 boundary vertex automatically being a level 2 boundary vertex.

Lemma 3.2.1. *The above strategy can be implemented in $O(n\sqrt{\log n})$ time and does not create more than $O(n/\sqrt{r_2})$ level 2 boundary vertices.*

Proof. It is easy to see that starting Algorithm 7 with this modification is identical to running it once with single parameter r_2. All we do is just continue recursively division from one of the regions. This observation proves that this strategy will not cause more than $O(n/\sqrt{r_2})$ boundary vertices of level 2 regions to be created.

Based on Chapter 2 calculation running graph division algorithm on level 1 regions will take $O(r_1 \log r_1)$ time per region. There are a total of $O(n/r_1)$ level 1 regions in the division, thus it gives us overall time of $O(n \log \log n)$ which after adding time it takes to divide graph into level 1 regions and substituting parameters we get $O(n\sqrt{\log n})$ time. □

3.2.2 Distance preprocessing

Given the nested division of planar graph G the goal of this phase is to calculate the distances between every pair of level 1 boundary nodes within each level 1 region. Since level 2 regions share their boundary nodes with the level 1 division, it is sufficient to perform only Algorithm 10 and mop-up phase is required. The main thrust already calculates distances from source to all boundary vertices.

We will treat every level 1 region separately, and describe procedure that have to be repeated for every level 1 region.

Algorithm 11 DISTANCE PREPROCESSING

input: Induced subgraph G by a level 1 region; level 2 division of G
output: Distances between all boundary nodes in G

```
1:  procedure DistancePreprocessing(G)
2:      D_1 ← ∅                          ▷ maps pair of level 1 boundary nodes on distance
3:      D_2 ← ∅                          ▷ maps pair of level 2 boundary nodes on distance
4:      for each level 2 region R_2 in G do
5:          for each boundary vertex b in R_2 do
6:              dist ← Dijkstra(G[R_2], b)
7:              for each boundary node w in R_2 do
8:                  D_2[b, w] ← dist(b, w)
9:              end for
10:         end for
11:     end for
12:     for each level 1 boundary vertex b in G do
13:         MainThrust(G,b,D_2)
14:         for each level 1 boundary vertex w in G do
15:             D_1[b, w] ← dist(b, w)
16:         end for
17:     end for
18:     return D_1
19: end procedure
```

To compute the distances between level 2 boundary vertices in each level 2 region, Dijkstra's algorithm is used. For each boundary node in a region, we run Dijkstra's algorithm on the subgraph induced by the corresponding level 2 region.

Lemma 3.2.2. *Given nested graph division of n-vertex planner graph. It is possible to calculate distances between every pair of boundary vertices shared by one region in* $O(n\sqrt{\log n})$ *time.*

Proof. Using Dijkstra's algorithm, the time required to find a shortest path tree in a region of size at most r_2 is $O(r_2 \log r_2)$. Computing these shortest path trees for each of the $O(n/\sqrt{r_1})$ level 2 boundary vertices requires a total time of $O(n \log\log n \log\log\log n)$.

Using the main thrust algorithm, the time to find a shortest path tree in a region of size at most r_1 is $O(r_1 + (r_1/\sqrt{r_2}) \log r_1)$, or simply $O(r_1)$. Therefore, the total time to find these shortest path trees for all $O(n/\sqrt{r_1})$ level 1 boundary vertices is $O(n\sqrt{r_1})$, which simplifies to $O(n\sqrt{\log n})$. □

3.2.3 Main search

Now given that for every level 1 region distances between all boundary nodes in each region are calculated, we can run main thrust on an entire graph. This will result in list of all shortest distances from source vertex to all level 1 boundary vertices. What's left is a quick mop-up phase to calculate distances to interior vertices. Mop-up phase for a single level 1 region can be calculated by labeling all boundary nodes with their distance to the source vertex and running Dijkstra algorithm to calculate all the remaining distance. This is identical to running Dijkstra algorithm on a graph induced by the region with an extra source node and extra edges connecting source to boundary vertices with appropriate weight.

Algorithm 12 FredericksonSSSP

input: Planar graph $G = (V, E)$, source vertex s,
output: Shortest path distances from s to all $v \in V$

```
1: D ← GetNestedDivision(G)
2: dist ← ∅
3: for each level 1 region R ∈ D do
4:     dist ← dist∪DistancePreprocessing(G[R])
5: end for
6: for each v ∈ V do
7:     d[v] ← ∞
8: end for
9: d[s] ← 0
10: MainThrust(G, D, dist)
11: MopUp(G,D,d)
12: return d[v] for all v ∈ V
```

Lemma 3.2.3. *The mop-up phase will take at most $O(n \log\log n)$ time.*

Proof. The mop-up for each of level 1 region will take $O(r_1 \log r_1)$ time. There are $O(n/r_1)$ regions which gives $O(n \log\log n)$ time in total. □

Parts 1 and 2 both run in $O(n\sqrt{\log n})$, dominating the overall running time of the algorithm. As a corollary of the lemmas proven above, we obtain the following theorem:

Theorem 3.2.4 (Theorem 3. in [6])**.** *A SSSP problem on an n-vertex planar graph with non-negative edge weights can be solved in $O(n\sqrt{\log n})$ time.*

Chapter 4

Henzinger's Single Source Shortest Path Algorithm

4.1 Algorithm details

Although both algorithms share a common idea and both use the algorithm from Chapter 3 to divide the graph into regions, they take different approaches. There are two main differences between Frederickson's and Henzinger's algorithms. First, Henzinger's algorithm operates on *directed* planar graphs, which makes it more general. Second, both algorithms manage heap updates in entirely different ways. For contrast, Frederickson's algorithm stores all boundary vertices in a single, cleverly organized heap throughout the main search phase. Henzinger's algorithm maintains multiple heaps and updates them in a more controlled and distributed manner.

Henzinger's algorithm has a shorter "attention span" than Frederickson's algorithm. The latter processes regions in complete phases, similar to Dijkstra's algorithm. In contrast, Henzinger's algorithm may stop the search phase in a region before it is fully completed, often switching focus between regions as the computation proceeds.

One further difference between the two algorithms is the number of layers in the division scheme. In contrast to Frederickson's approach, Henzinger's algorithm employs three levels of division: a trivial level 0, where each region consists of a single edge; a nontrivial level defined by the parameter r, analogous to Frederickson's main division; and level 2, where the entire graph forms a single region.

4.1.1 Priority queue

During the main part the Henzinger's algorithm makes use of priority queues. The structure of size n must support following operations in a given time bound [9]:

- UPDATE(Q, x, k) which updates the key of x to k in $O(\log n)$,
- MINITEM(Q), which returns the item in Q with the minimum key in $O(1)$,
- MINKEY(Q), which returns the key associated with MINITEM(Q) in $O(1)$.

We will denote by $Q(R)$ the priority queue associated with region R. During the algorithm, all keys are initialized with ∞ which means all items are inactive. The data structure can also allow for the removal of vertices from the priority queue once they are inactive.

4.1.2 Algorithm and proof of correctness

As mentioned before in the first part, the algorithm uses strategy from Chapter 2 to divide graph into level 1 regions of size $r = O(\log^4 n)$ with $\sqrt{r} = O(\log^2 n)$ boundary nodes per region. It was proven that such division can be achieved in

$$O(n \log r + (n/\sqrt{r}) \log n) = O(n \log \log n + n/\log n) = O(n \log \log n)$$

which is exactly time complexity proposed by Monica Henzinger [9].

Now we can briefly describe the main search phase of the algorithm, assuming the division has already been created. Before the algorithm starts, we initialize $d[v] = \infty$ for all vertices and $d[s] = 0$ for the source node s. For each region, a priority queue is created to store the distance for each node and a main priority queue is used to store the minimum distance per region.

The algorithm proceeds in two steps. In step 1, a region containing a node with the smallest $d[v]$ and active edges is selected. Then, step 2 is executed either $\log n$ times or until there are no active edges remaining in the region. In step 2, which operates within a single region, we select the node with the smallest $d[v]$ and active edges (similarly to Dijkstra's algorithm), relax all its active edges within the region, and update the relevant priority queues whenever any value changes.

To prove the complexity of the algorithm, we will introduce some new definitions.

Definition 4.1. *(atomic region) For and edge uv an **atomic region** is a graph region that consists only of two nodes u and v and can be associated with single directed edge (u, v). We will denote **atomic region** as $R(uv)$. This type of regions will be called level 0 region.*

We can transform our division in linear time into division that consists of 3 separate levels. Level 0 regions are atomic regions and there exists one atomic region for every directed edge of graph G. Level 1 regions will be regions generated by strategy from Chapter 2. Due to how regions were defined both vertices of any level 0 region always share at least one level 1 region. By level 2 region we will denote region R_G that consists of the whole graph. There is a clear relations ship between region levels. Level i region is always fully contained

inside some level $(i+1)$ region. For a level i region R all level $(i-1)$ regions that are fully contained inside it will be called *children of region R* (with an analogous definition for *parent* of a region R). The algorithm makes use of two parameters $\alpha_2 = 1$ and $\alpha_1 = \log n$ which intuitively limit the attention span of the algorithm on each level of division. By $l(R)$ we will denote level of a region R. Before running the algorithm for all vertices v we initialize $d[v]$ and all keys of priority queues with ∞.

Recursive version of the algorithm significantly simplifies the proofs and better shows the nature of the algorithm. In Chapter 5 we will propose version of Algorithm 13 that significantly reduces number of recursive calls.

Now we can move on to prove the correctness of the algorithm. It is a known fact that for SSSP algorithm to be correct three conditions must be satisfied:

- $d[s] = 0$ for the source vertex s.
- For every vertex v, $d[v]$ must be an upper bound of and actual distance from s to v
- For every edge (u, v) inequality $d[v] \leq d[u] + w(u, v)$ holds. Edge that satisfies inequality if called *relaxed edge*.

The following lemmas are necessary to prove that all conditions holds.

Lemma 4.1.1 (Lemma 3.1. in [9])**.** *For every vertex v during the algorithm $d[v]$ is always greater or equal to the actual distance from s to v.*

Proof. By induction on the number of steps of the algorithm. We state that the invariant holds during the entire run of the algorithm. Initially all vertices are labeled with ∞ and source vertex s is labeled by 0, the lemma is obviously true. The only steps of the algorithm that requires verification is moment when some edge (u, v) is relaxed. By induction $d[u]$ is an upper bound of distance from s to u so $d[v] + w(u, v)$ is an upper bound of distance from s to v

□

We say that edge (u, v) is active when the key of (u, v) in priority queue $Q(R(uv))$ is not an infinity.

Lemma 4.1.2 (Lemma 3.2. in [9])**.** *If an edge (u, v) during an algorithm is inactive it is also relaxed.*

Proof. The lemma holds before first step 1 of the algorithm begins as all vertices are initialized with ∞ and only edges from source node s are active. During the algorithm values of $d[v]$ can only decrease, thus an edge uv can stop being relaxed only when $d[u]$ changes. If $d[u]$ changes than key in all priority queues that contain the u vertex is also updated. This means that edge is active when is not relaxed. This by contraposition proves the lemma. □

Lemma 4.1.3. *Once the algorithm terminates all edges are inactive.*

Algorithm 13 SimplifiedHenzingerSSSP

input: Planar graph $G = (V, E, w)$, source node s, leveled division D
output: Shortest path distances $d[v]$ from s to all $v \in V$

```
1: procedure SimplifiedHenzingerSSSP(G, s, D)
2:     d[s] ← 0
3:     for all edges (s, x) do
4:         UpdateGlobal(R(sx), (s, x), 0)
5:     end for
6:     while minKey(Q(R_G)) < ∞ do
7:         Process(R_G)
8:     end while
9: end procedure
10:
11: procedure Process(region R)
12:     if R contains a single edge (u, v) then
13:         if d[v] > d[u] + w(u, v) then
14:             d[v] ← d[u] + w(u, v)
15:             for all outgoing edges (v, x) of v do
16:                 UpdateGlobal(R(vx), (v, x), d[v])
17:             end for
18:         end if
19:         updateKey(Q(R), (u, v), ∞)
20:     else
21:         repeat α_{l(R)} times                ▷ l(R) denotes level of region R
22:             if minKey(Q(R)) = ∞ then
23:                 break
24:             end if
25:             R* ← minItem(Q(R))
26:             Process(R*)
27:             updateKey(Q(R), R*, minKey(Q(R*)))
28:         end
29:     end if
30: end procedure
31:
32: procedure UpdateGlobal(region R, item x, key k)
33:     before ← minKey(Q(R))
34:     update(Q(R), x, k)
35:     after ← minKey(Q(R))
36:     if before ≠ after then
37:         UpdateGlobal(parent(R), R, minKey(Q(R)))
38:     end if
39: end procedure
```

Proof. The key in the priority queue of a parent region is updated whenever one of its child regions obtains a new minimum key, and also whenever the child region with the current minimum key is processed. This ensures that the key representing a child region in the parent's priority queue is never equal to ∞ as long as there exists at least one child region with a key not equal to ∞.

Clearly when algorithm runs there must exist at least one region which priority queue has a key smaller than infinity. From how we define active edges it means that during the algorithm there must be an active edge and algorithm terminates once there are no more active edges. □

Theorem 4.1.4. *Algorithm 13 correctly calculates all distances from source vertex s to all vertices in the planar graph.*

Proof. The first condition follows directly from the fact that $d[s]$ is initialized with 0 and all edges have non-negative edge weights so during the algorithm $d[s]$ remains unchanged. The second condition follows directly from Lemma 4.1.1. The third condition is a direct conclusion from Lemma 4.1.2 and Lemma 4.1.3 □

4.2 Charging scheme invariant

4.2.1 Definitions

To conduct a full time analysis of Algorithm 13, we introduce several new definitions and concepts that will be used to prove the *Charging Scheme Invariant* and establish the time bound.

A call to the procedure PROCESS with a region R will be called an *invocation* A, and R is referred to as the region of the invocation. Every invocation that results from line 23 of Algorithm 13 within invocation A is called a *child* of A; similarly, invocation A is called the *parent* of any such child. The notions of *descendant* and *ancestor* are defined in the natural way.

The *end key* of an invocation A, denoted $end(A)$, is defined as the value of MINKEY($Q(R)$) for the invocation's region R after immediately the invocation finishes. The *start key* of an invocation A with region R, denoted $start(A)$, is the value of MINKEY($Q(R)$) before the invocation begins. The *start node* of an invocation A is the first node that gets $start(A)$ value.

An invocation A is called *truncated* if $end(A) = \infty$. Intuitively, this occurs when the process terminates before all $\alpha_{l(R)}$ steps are completed. By this definition, any invocation on an atomic region $R(uv)$ is truncated, and the algorithm terminates when the invocation on the top-level region R_G is truncated.

Definition 4.2. *An **entry node** of a region R is defined differently depending on the level of the region:*

- *For an atomic region $R(uv)$, u is the entry node of that region.*
- *For a level 1 region, the entry nodes are all boundary nodes shared by that region.*

- *For the top-level region R_G, the source node s is the only entry node.*

An important part of the time analysis is the charging scheme, which allows us to bound the number of truncated invocations. We say that each truncated invocation C is *charged* to a pair (R, v), where R is an ancestor of the region of C and v is some entry node of R. Invocation C is called the *charger*. It is worth noting that an invocation can be charged to its own region, since by standard convention a region is considered to be its own ancestor.

4.2.2 Charging scheme

The goal of this section is to provided detailed explanation of how charging scheme works and in conclusion prove the invariant that will allow us to conduct full running time complexity of Algorithm 13.

Lemma 4.2.1 (Lemma 3.6 in [9])**.** *For an invocation A and its children $A_1, A_2, \ldots, A_q$ the following claim holds:*

$$start(A) \leq start(A_1) \leq ... \leq start(A_q) \leq end(A).$$

Proof. The lemma is obvious for level 2 and level 0 regions, as level 2 region have only one child and level 0 regions don't have children at all. For level 1 region R of invocation A. The inequality $start(A) \leq start(A_1)$ is always true because between to calls to the procedure PROCESS value of the $Q(R)$ does not change. Let's assume for the sake of contradiction that from some child level 0 invocation A_i, $start(A_i) \geq start(A_{i+1})$. Let $R(uv)$ be the region of A_i. The inequality implicates that newly calculated value $d[v]$ for edge vw is smaller than smallest key in $Q(R)$, but $start(A_i) \leq d[u] \leq d[u] + w(u, v) = d[v]$ which leads to contradiction. □

We will now define partial order on invocations. We say for invocations A and B that $A \leq B$ when region of both invocations is the same and invocation A happens no later tan invocation B. If $A \neq B$ and $A \leq B$ than we write $A < B$. We call A and immediate predecessor of B when $A < B$ and if there exists invocation C such that $A < C \leq B$ then $C = B$. Due to how partial order is defined it does not mean that there isn't any other invocation between A and B. It just means there is no other invocation on region of A.

Definition 4.3 (stable invocation)**.** *An invocation A is **stable** when for every invocation $B > A$, $start(A) \leq start(B)$*

Rather obvious corollary from Lemma 4.2.1 is that every invocation on region R_G is stable. What's also important that every invocation on atomic region $R(uv)$ is stable if and only if no further invocation on region $R(uv)$ will occur.

Definition 4.4. *For every invocation A we will define its **lowest stable ancestor** as an invocation AS with smallest possible level i such that it is stable and is an ancestor of invocation A. It will be denoted by $stableAncestor(A)$.*

First invocations on region R_G is stable ancestor of any invocation, thus $stableAncestor(A)$ is well defined.

For unstable invocation A with region R there must exist other invocation B on region R' that changes values in $Q(R)$. Invocation on Region R' can only affect nodes inside this region. Hence the invocation B must have an impact on an vertex that is share by both R and R' regions - an entry node R. We provide now more definition to describe such event

Definition 4.5 (Entry predecessor)**.** *An **entry predecessor** of invocation A is an invocation $E \leq A$ such that start node of invocation E is an entry node of region of invocation A. We denote an entry predecessor of invocation A as $entryPredecessor(A)$.*

Now we can describe charging scheme in a very elegant way:

Charging scheme

Truncated invocation C will be charged to pair (E, v) where

$$E = entryPredecessor(stableAncestor(C))$$

and v is an entry node of E.

We will now try to use the scheme to prove flowing invariant:

Lemma 4.2.2 (Lemma 3.15 in [9])**.** *For each pair (R, v) there is a invocation A with region R such that every charger of (R, v) is a descendant of A.*

4.2.3 Proving invariant

Before proving the Invariant Lemma 4.2.2, we need prove some important relationships between $start(A)$ and $end(A)$ for some invocation A.

Lemma 4.2.3 (Lemma 3.9 [9])**.** *Let k_0 be the key associated with R at some time t, and let A be the first invocation with region R occurring after time t. If $start(A) < k_0$ then A starts with an entry node.*

Proof. Due to how algorithm is designed key associated with R in parent priority queue after some invocation ends is equal to the $\text{minKey}(Q(R))$. According to Lemma 4.2.1 invocation on single region R does not lower its key. The only way for $start(A)$ to be lower than k_0 is if the value was changed by some region R'. Invocation on region R' only affect nodes inside region R', and thus it needs to be entry node of region R □

From the proven lemma we get directly two corollaries.

Corollary 4.2.3.1. *Let A and B be invocations with region R such that A is invocation's B immediate predecessor. If $end(A) > start(B)$ then B starts with an entry node of R.*

Corollary 4.2.3.2. *For each region R, the first invocation with region R starts with an entry node of R.*

This follows from the fact that all regions are either initiated with key equal to infinity or include source node initiated with 0.

Lemma 4.2.4. *If $A < B$ are two invocations with the same start node then $start(A) > start(B)$.*

Proof. Let v be the start node of A and B. After invocation A key of outgoing edges of v has been set to infinity. All edges will be process because $\alpha_1 > 1$. Then for the edges to be active again so that invocation B can start with processing them the value of $d[v]$ needs to be updated, but during the algorithm $d[u]$ for any node u only decreases. And thus $start(B) = d[v] < start(A)$. □

Lemma 4.2.5. *For every invocation A, $start(entryPredecessor(A)) \leq start(A)$.*

Proof. Let $E < A_1 < ... < A_p = A$ be invocation. According to definition of entry predecessor of A. None of the A_i starts with entry node. From Lemma 4.2.1 we get that $start(A_i) < end(A_i)$ for every i. If $end(A_i) > start(A_{i+1})$ were true for some i then it would indicated that A_{i+1} started with entry node, but that contradicts with definition of entry predecessor. □

Lemma 4.2.6 (Lemma 3.14 in [9])**.** *Suppose $A < B$ are two invocations such that $entryPredecessor(A) = entryPredecessor(B)$. If B is stable then every child of A is stable.*

Proof. Similarly to prove of the Lemma 4.2.5, let $A = C_0 < ... < C_p = B$. Because $entryPredecessor(A) = entryPredecessor(B)$ we know that for C_i does not start with entry node for $i = 1, ...p$ and by Lemma 4.2.1 and Corollary 4.2.3.1 we get:

$$start(C_0) \leq end(C_0) \leq start(C_1) \leq ... \leq start(C_p) \leq end(C_P). \tag{4.1}$$

Let A' be child of A and C' be invocation such that $A' < C'$. To prove stability of any child of A we need to show that $start(A') \leq start(C')$.

Lets assume by contradiction that $start(C') < start(A')$ and let C be the parent of C'. If $C = C_i$ for some i this contradicts with Equation (4.1), thus $C > B$. B is stable, by definition $start(C) \geq start(B)$ which contradicts Equation (4.1) □

Thanks to the lemmas proven above we can now proceed with proving the Charging Scheme Invariant from lemma 4.2.2.

Proof of invariant Lemma 4.2.2

Proof. Let truncated invocation C be the first charger of pair (R, v). Let $A = stableAncestor(C)$ and $E = entryPredecessor(A)$. According to Charging Scheme R is a region of E and A and v is a start node of E. By previous Lemma 4.2.5

$$start(E) \leq start(A). \tag{4.2}$$

Let $B > A$ be an stable invocation. We want to prove that no descendant of B is a charger of (R, v). By contradiction let C' and $B = stableAncestor(C')$ and a charger of (R, v). Let $E' = entryPredecessor(B)$. Because B is a charger of (R, v) E' and E have the same start node v. Let's assume that E is different invocation than E'. Because $B > A$ and E is entry predecessor of A we get $E < A < E' < B$. According to Lemma 4.2.4 $start(E) > start(E')$ combining that with Equation (4.2) we get $start(E') < start(A)$ which contradicts stability of A and thus $E' = E$.

Now there are two cases to consider.

Either $C \neq A$ - by Lemma 4.2.6 every child of A is stable which means it is contradicting the fact that A is lowest stable ancestor of C

Or $C = A$. In this case A is charger of (R, v) which means A is truncated. Consider A' being immediate successor of A. According to Corollary 4.2.3.1 A' need to start with entry node contradicting the fact that $E = entryPredecessor(B)$ because $E < A' \leq B$. □

4.3 Analysis

Given the Lemma 4.2.2 holds we will conduct the analysis in two parts. We will bound the time spent on the procedures Process and UpdateGlobal separately. First, we will bound the number of invocations s_i at each level i and the time t_i spent by a single call to Process at level i. This allows us to calculate the total time spent by the entire algorithm using the formula $\sum_{i=0}^{2} t_i s_i$. Second, we will bound the time spent in the UpdateGlobal procedure. The bounds for level 0 invocations are particularly important, as UpdateGlobal is called only when processing level 0 regions. Together, these two bounds will complete our analysis.

Procedure Process

Given the invariant from Lemma 4.2.2 we want to bound number of truncated invocations to bound time complexity of Algorithm 13. Firstly we can start by bounding by bounding the number of invocations charging to a pair (R, v). As a reminder the Algorithm uses two parameters $\alpha_2 = 1$ and $\alpha_1 = \log n$ which bound number of children per invocation. The number of charging invocation to a pair changes depending on a region level:

- for level 0 region R and its entry node v, (R, v) is charged by at most one invocation,

- for level 1 region R and its entry node v, (R, v) is charged by at most one level 1 invocation and at most $\alpha_1 = \log n$ level 0 invocations,
- for level 2 region R and its entry node v, (R, v) is charged by at most one level 2 invocation, at most $\alpha_2 = 1$ level 1 invocation and at most $\alpha_1 = \log n$ level 0 invocation. Which gives in $\log n$ in total.

Now we can bound the number of truncated invocations at each level. At level 2, there can be only one truncated invocation, since there is only one pair (R, v) where R is a level 2 region.

At level 1, there is one truncated invocation charging to the level 2 pair. There are $O((n/\log^4 n)\log^2 n) = O(n/\log^2 n)$ pairs involving level 1 regions, which gives us at most $O(n/\log^2 n) + 1 = O(n/\log^2 n)$ level 1 truncated invocations.

At level 0, there are $\log n$ invocations charging to the level 2 pair and $\log n$ invocations per each level 1 pair, which gives $O(n/\log^2(n \log n)) = O(n/\log n)$ invocations charging to level 1 pairs, and $O(n)$ invocations charging to $O(n)$ level 0 pairs. Summing up, the total number of truncated level 0 invocations is $O(n)$.

Now count number of all invocations s_i and time t_i for procedure Process to take with out counting time needed for UpdateGlobal.

- Level 0 regions: Every invocation on level 0 is truncated which means $s_0 = O(n)$ and obviously all operation during invocation on level 0 take constant amount of time - $t_0 = O(1)$.
- Level 1 regions: Every level 1 invocation, which is not truncated can cause at most $\alpha_1 = \log n$ level 0 invocation thus:

  $$s_1 \leq s_0/\log n + O(n/\log^2 n) = O(n/\log n)$$

  Each level 1 invocation uses priority queue of size $\log^4 n$ and calls operation on the queue at most $\log n$ times and thus $t_1 = O(\log n \log\log n)$
- Level 2 region: In a similar fashion to the proof for level 1 invocations:

  $$s_2 \leq s_1 + 1 = O(n/\log n)$$

  Because $\alpha_2 = 1$ every level 2 invocation result in single operation on queue of size $O(n/\log^4 n)$, thus $t_2 = O(\log n)$.

As mentioned before time taken by algorithm excluding operations by procedure UpdateGlobal is equal to $\sum_{i=0}^{2} t_i s_i = O(n \log\log n)$

Procedure UpdateGlobal

Now we will analyze the running time of the algorithm, excluding the time spent in the Process procedure. An important observation is that UpdateGlobal is only called from level 0 invocations, and each call-depending on

changes in the priority queue-may recursively trigger the procedure at higher levels. Since there are at most $O(n)$ level 0 invocations, the procedure can be called at most $O(n)$ times. The running time for a single call to UPDATE-GLOBAL at level 0 is $O(1)$. At level 1, the procedure updates a queue of size $O(\log^4 n)$, which takes $O(\log \log n)$ time. At level 2, it operates on a queue of size $O(n/\log^4 n)$, so each call takes $O(\log n)$ time.Not every level 0 call results in a level 2 UPDATEGLOBAL call. We can bound the total number of calls that result in updates at most at level 1 by $O(n \log \log n)$, since there are at most $O(n)$ calls, each taking at most $O(\log \log n)$ time.It remains to show that the number of calls to UPDATEGLOBAL that result in an update at the level 2 region is at most $O(n/\log n)$.

We can distinguish two case. Level 0 invocations that are charged to level 1 and 2 regions and the rest. In the first case it was calculated that there are at most $O(n/\log n)$ such invocation which gives us at most $O((n/\log n)\log n) = O(n)$ time for all calls to UPDATEGLOBAL.

Now, it remains to bound the time spent in the UPDATEGLOBAL procedure for level 0 invocations that are charged to the corresponding level 0 regions and result in UPDATEGLOBAL being called with a level 2 region. There is at most one such invocation per pair (R, u), where R is a level 0 region.Let us focus on a single vertex v and count how many such invocations can be made on regions of the form $R(uv)$. Since v can belong to at most a constant number of regions (due to the bounded degree and overlap), and its in-degree is also constant, there can only be a constant number of invocations on regions $R(uv)$ for a single vertex v. Furthermore, because the call to UPDATEGLOBAL results in a level 2 update, it indicates that v is an entry node for some region. There are at most $O(n/\log^2 n)$ such entry nodes.

Therefore, the total time spent in UPDATEGLOBAL for these cases is

$$O((n/\log^2 n)\log n) = O(n/\log n)$$

Summing the time spent in the procedures UPDATEGLOBAL and PROCESS gives a total complexity of $O(n \log \log n)$. As a corollary of this analysis, we obtain the following theorem:

Theorem 4.3.1. *The SSSP problem on an n-vertex directed planar graph can be solved using the simplified Henzinger's algorithm in $O(n \log \log n)$ time.*

Chapter 5

Implementation details and benchmarks

5.1 Implementation details

Both algorithms described in this paper were implemented using C++20 for an open-source KOALA NetworKit library [13].

5.1.1 KOALA NetworKit

According to library README file the KOALA NetworKit is a hybrid library that combines the efficiency and scalability of the NetworKit platform with the rich set of classical graph algorithms provided by the KOALA library. NetworKit is a modern, open-source C++ library focused on fast analysis and mining of large-scale networks, offering high-performance implementations of algorithms for centrality, community detection, graph metrics, and random graph generation, with extensive use of parallelization and efficient data structures. KOALA, developed at Gdansk University of Technology, supplements this toolkit with a comprehensive collection of algorithms for classical discrete optimization and graph theory problems, including advanced procedures for coloring, flows, scheduling, and graph recognition. KOALA NetworKit thus provides a unified interface for both large-scale graph analytics and specialized combinatorial algorithms, making it a versatile tool for both theoretical and practical graph research.

The contribution of this work to the Open Source KOALA NetworKit library is mainly in three file:

- `cpp/planar_sssp/SuitableRDivision.cpp` - implementation of Algorithm 7 and implementation of planar separator algorithm,
- `cpp/planar_sssp/FredericksonPlanarSSSP.cpp` - implementation of Algorithm 12 (class `FredericksonPlanarSSSP`),

- `cpp/planar_sssp/HenzingerPlanarSSSP.cpp` - implementation of Algorithm 13 (class `HenzingerPlanarSSSP`).

5.1.2 Suitable r-division

One important implementation detail has been omitted throughout this work and in [6], namely, how to efficiently merge regions.

The division is implemented as a vector of vectors. While this representation is convenient to use, it presents a challenge for greedy merging of regions. Initially, regions can be of size $O(1)$, which means that a single region may be merged many times with others before reaching the desired size. To ensure a linear time bound, it is sufficient to always merge the smaller region into the larger one. In this approach, whenever two regions are merged, the nodes from the smaller region are moved into the larger one. Although a single node may be moved between regions multiple times, by always merging the smaller region into the larger, the total number of moves over all nodes is amortized and remains $O(n)$.

For the planar separator implementation, the Fundamental Cycle Separator was used. This choice is motivated by its simplicity of implementation and its favorable experimental separator sizes.

5.1.3 Henzinger's SSSP

There is a major optimization that can be applied to improve the efficiency of Algorithm 13, we can eliminate almost all recursive calls. Below, we present an iterative and simplified version of Algorithm 13, based on the brief description in [9]. The iterative approach is particularly well-suited to the three-level version of the algorithm described in the paper. In the recursive variant, certain calls to the Process procedure serve primarily a theoretical purpose, especially for level 2 regions. Additionally, the priority queues corresponding to single atomic regions have been omitted.

Algorithm 14 OPTIMIZEDSIMPLIFIEDHENZINGERSSSP

input: Planar graph $G = (V, E)$, source node s
output: Shortest path distances $d(v)$ from s to all $v \in V$

```
1: while minKey(Q) < ∞ do
2:     R ← minItem(Q)
3:     for i = 1 to log n do
4:         uv ← minItem(Q(R))
5:         d[v] ← d[u] + w(uv)
6:         update(Q(R), (u, v), ∞)
7:         for all outgoing edge (v, w) of v in R do
8:             for all R' that contain edge vw do
9:                 update(Q(R'), (v, w), d[v])
10:                if minKey(Q(R')) changed then
11:                    update(Q, R', minKey(Q(R')))
12:                end if
13:            end for
14:        end for
15:    end for
16:    update(Q, R, minKey(Q(R)))
17: end while
18: return d[v] for all v ∈ V
```

One more optimization that can be done as mentioned in [9] is to use priority queue that allows for deletion and insertion of regions. Because simplified version of the algorithm uses only three levels of division, finding the appropriate queue is not problematic and results in improvement of a constant. Unfortunately this optimization does not result in improved complexity.

For the implementation of priority queue standard `std::set` has been used with extra caching of the smallest key to ensure constant lookup times.

5.2 Benchmarks

Running time of both algorithms have been tested with following experiments:

1. grid, triangular grid, hexagonal grid, and single path graphs with fewer than 10000 nodes were tested with randomly generated edge weights using the suitable r-division algorithm (see Figures 5.2 to 5.5),

2. a hexagonal grid with more than 1,000,000 nodes was tested using a precalculated division (see Figure 5.10),

3. the effect of different division parameters was examined for grid, hexagonal grid, and triangular grid graphs (see Figures 5.7 to 5.9),

4. subgraphs with fewer than 10,000 nodes were sampled from the New York City map dataset [3] (see Figure 5.1),

5. maximal planar graphs(see Figure 5.6).

All edge weights were generated randomly by sampling from a uniform distribution over the range $[0, 1000]$. The subgraphs of the NY city map were created by randomly selecting nodes and then running BFS until the desired size was reached. Maximal planar graphs were generated by first creating a cycle of the desired number of vertices and then triangulating the graph by adding edges as long as possible without breaking graphs planarity.

5.3 Results

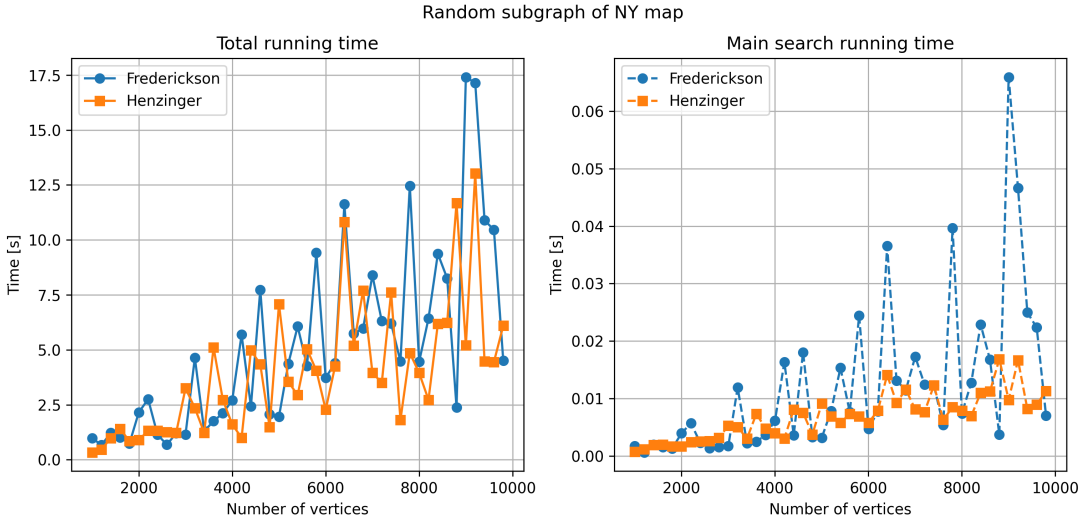

Figure 5.1

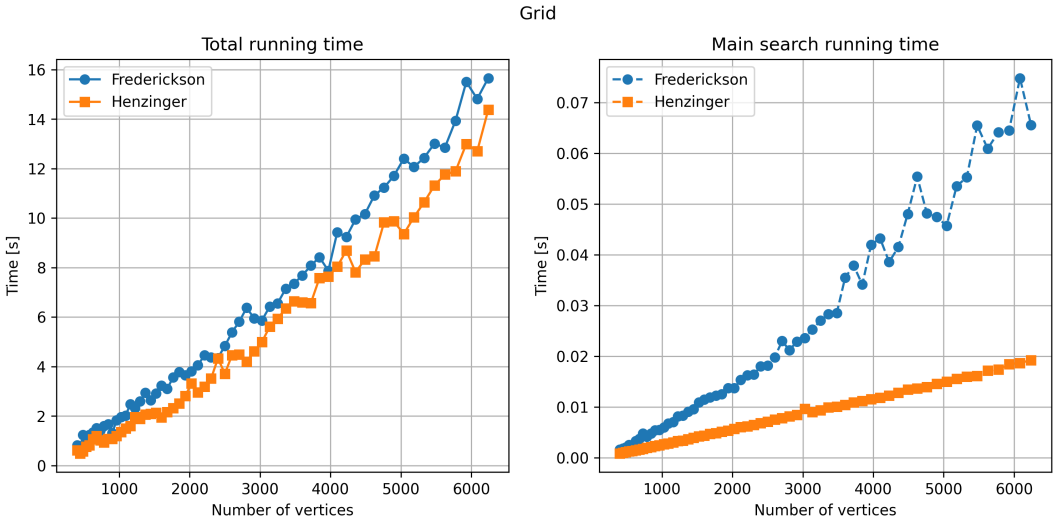

Figure 5.2

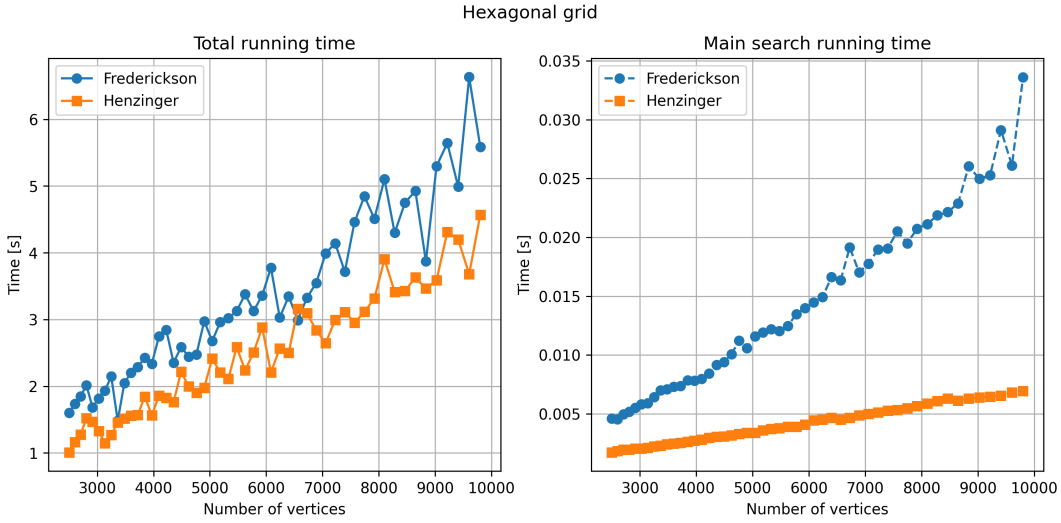

Figure 5.3

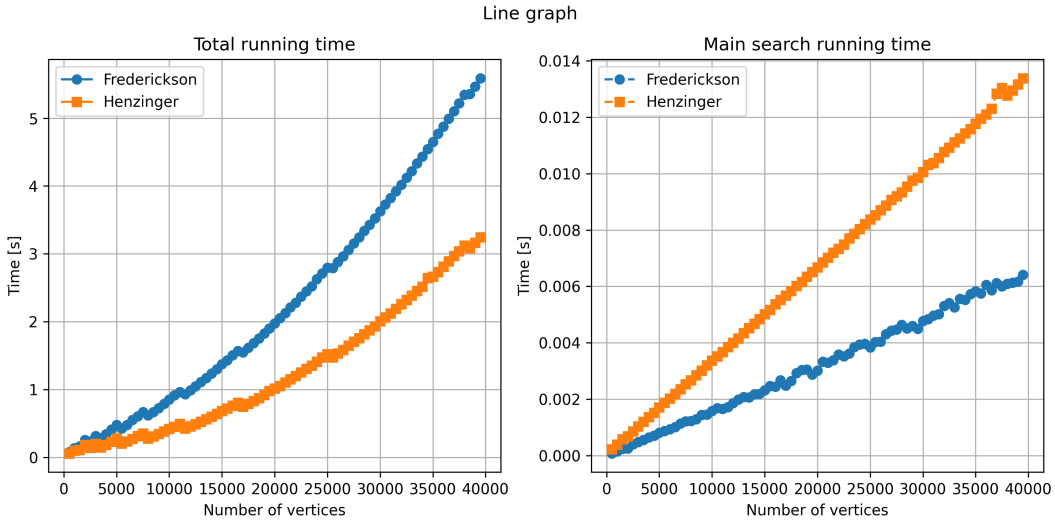

Figure 5.4

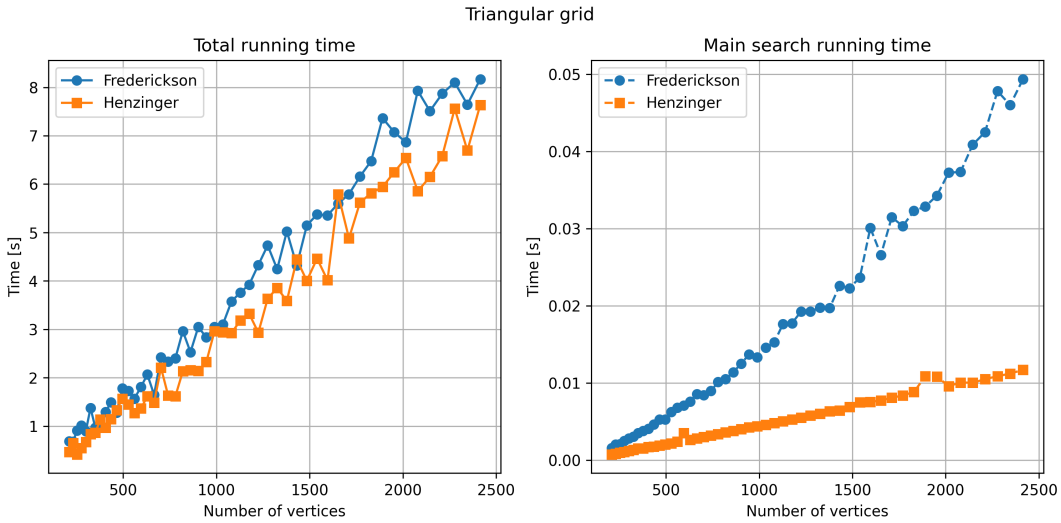

Figure 5.5

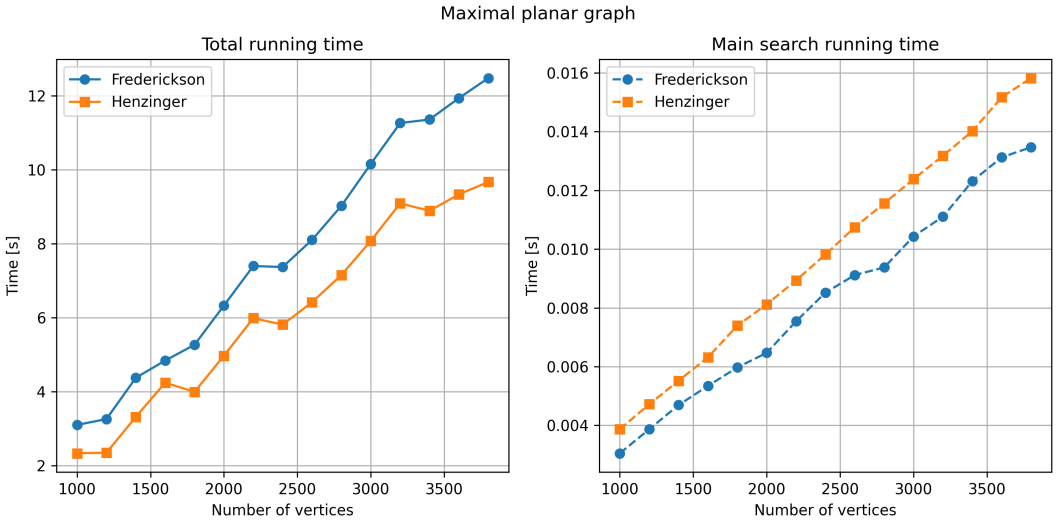

Figure 5.6

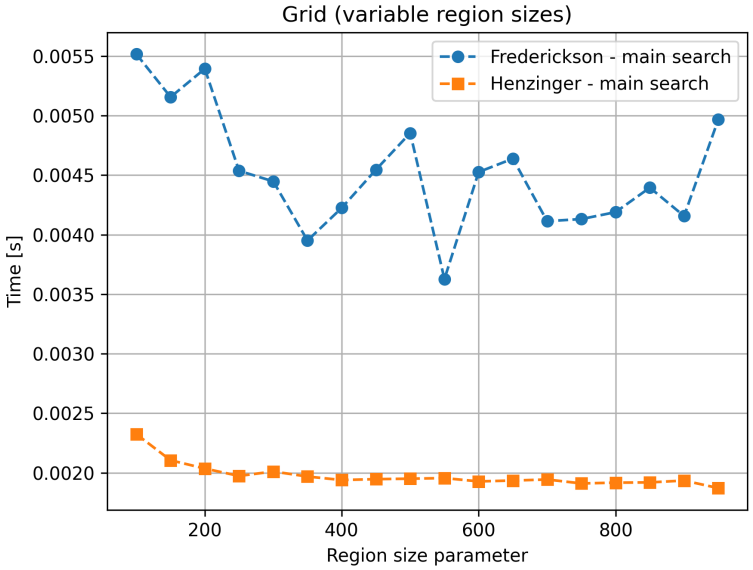

Figure 5.7

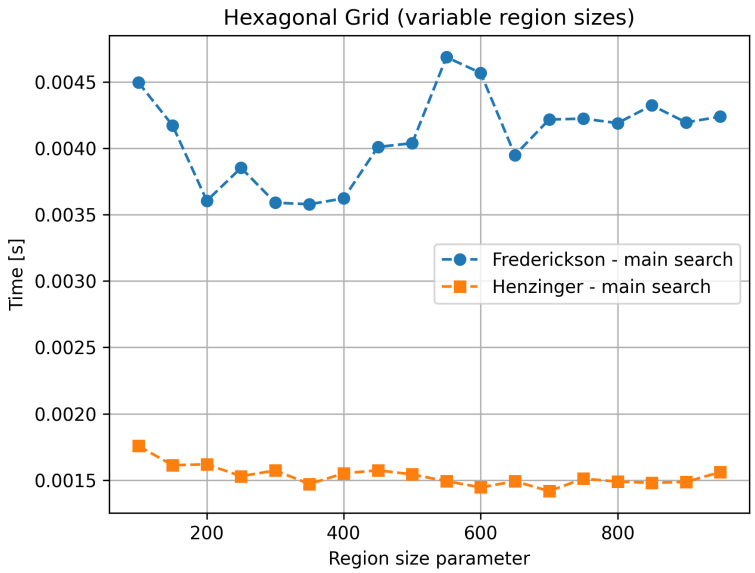

Figure 5.8

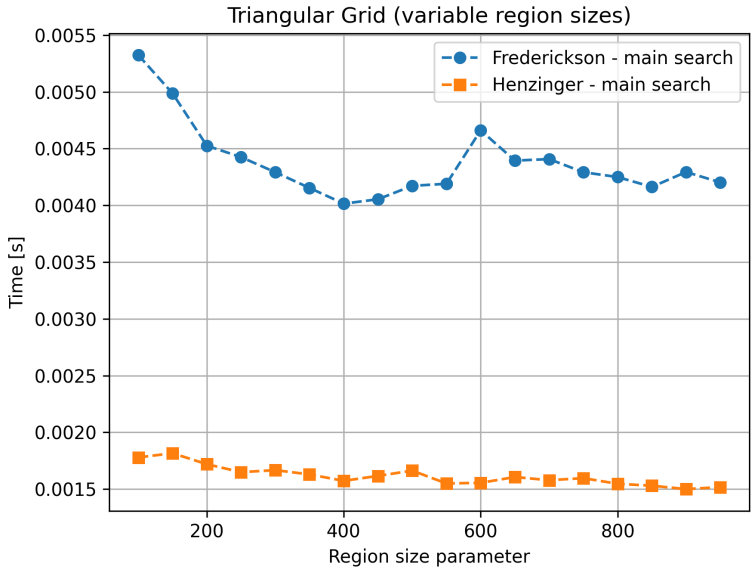

Figure 5.9

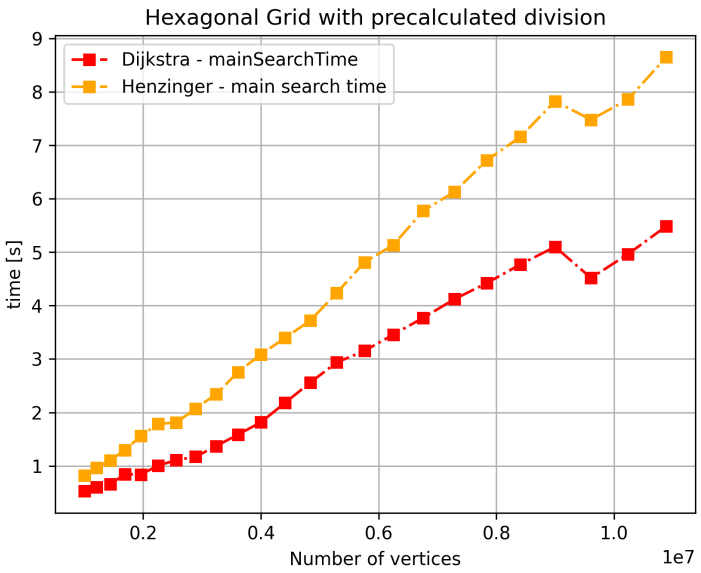

Figure 5.10

The time measurements in the benchmarks were divided into the total running time and the running time of the main search phase only, as the main contributing factor to the total running time was generating the division. The difference between the full running time and the main search phase time was primarily due to the implementation of the separator algorithm.

From the benchmarks, we can conclude that Henzinger's algorithm is faster for denser graphs, although its running time on maximal planar graphs (see Figure 5.6) is comparable to Frederickson's algorithm, with a slight advantage for Frederickson. Even though all graphs were transformed to have bounded degree, the underlying structure of the graph still impacts performance after the transformation. The main search phase of Frederickson's algorithm runs faster on very sparse graphs such as line graphs(see Figure 5.4), but its total running time is significantly worse in all cases compared to Henzinger's algorithm. This is likely due to the extra level of division, smaller parameter sizes, and the dominating effect of the separator algorithm implementation. Benchmarks on subgraphs of the real New York City map (see Figure 5.1) further confirm that the structure of the graph has a significant impact on the running time of the algorithms, resulting in considerable variation in the measured sample times.

Bibliography

[1] Noga Alon, Paul Seymour, and Robin Thomas. "Planar Separators". In: *SIAM Journal on Discrete Mathematics* 7.2 (1994), pp. 184–193.

[2] Reinhard Diestel. *Graph Theory*. Springer-Verlag, 2000.

[3] DIMACS. *9th DIMACS Implementation Challenge - Shortest Paths*. 2025. URL: `https://www.diag.uniroma1.it/~challenge9/download.shtml`.

[4] Hristo Djidjev. "On the Problem of Partitioning Planar Graphs". In: *SIAM Journal on Algebraic Discrete Methods* 3.2 (1982), pp. 229–240.

[5] Hristo Djidjev and Shankar Venkatesan. "Reduced Constants for Simple Cycle Graph Separation". In: *Acta Informatica* 34 (1997), pp. 231–243.

[6] Greg Federickson. "Fast Algorithms for Shortest Paths in Planar Graphs, with Applications". In: *SIAM Journal on Computing* 16.6 (1987), pp. 1004–1022.

[7] Greg Frederickson. "Data Structures for On-Line Updating of Minimum Spanning Trees, with Applications". In: *SIAM Journal on Computing* 14.4 (1985), pp. 781–798.

[8] Michael Fredman and Robert Endre Tarjan. "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms". In: *Journal of the ACM* 34.3 (1987), pp. 596–615.

[9] Monika Henzinger, Philip Klein, Satish Rao, and Sairam Subramanian. "Faster Shortest-Path Algorithms for Planar Graphs". In: *Journal of Computer and System Sciences* 55.1 (1997), pp. 3–23.

[10] Martin Holzer, Frank Schulz, Dorothea Wagner, Grigorios Prasinos, and Christos Zaroliagis. "Engineering Planar Separator Algorithms". In: *Journal of Experimental Algorithmics* 14 (2010), pp. 1–5.

[11] Richard Lipton and Robert Endre Tarjan. "A Separator Theorem for Planar Graphs". In: *SIAM Journal on Applied Mathematics* 36.2 (1979), pp. 177–189.

[12] Kenneth Rosen. *Discrete mathematics and its applications*. McGraw Hill, 2007.

[13] Krzysztof Turowski. *KOALA NetworKit*. 2025. URL: `https://github.com/krzysztof-turowski/koala-networkit` (visited on 06/26/2025).

[14] Wikipedia. *Planar graph*. 2025. URL: https://en.wikipedia.org/wiki/Planar_graph.