INDRAPRASTHA INSTITUTE OF INFORMATION TECHNOLOGY, DELHI

MASTER'S THESIS

Upper and Lower bounds of various Centrality Measures on Planar and Sparse Graphs

Author:
Sudatta BHATTACHARYA

Supervisor:
Dr. Debajyoti BERA

A thesis submitted in fulfillment of the requirements for the degree of Master of Technology

in the

Theoretical Computer Science
Department of Computer Science and Engineering

August 3, 2020

INDRAPRASTHA INSTITUTE *of* INFORMATION TECHNOLOGY **DELHI**

Declaration of Authorship

"~~Happiness~~ Ideas can be found, even in the darkest of times, if one only remembers to turn on the light."

Albus Dumbledore and Me

INDRAPRASTHA INSTITUTE OF INFORMATION TECHNOLOGY, DELHI

Abstract

Department of Computer Science and Engineering

Master of Technology

Upper and Lower bounds of various Centrality Measures on Planar and Sparse Graphs

by Sudatta BHATTACHARYA

In this thesis we address the problem of computing closeness centrality, Harmonic centrality and a few related centrality measures that operate on the shortest paths in a graph. We consider sparse graphs, especially planar graphs and this makes our results widely applicable to real-world networks such as social, geographical, citation, biological, communication, etc. on which centrality values are often evaluated in practice. We introduce a generalisation of Harmonic centrality and two simplifications of betweenness centrality, a more well-known but more complicated notion of centrality. We show that closeness, Harmonic and number-farness centrality values of all nodes of a planar graph can be computed in $o(n^2)$. On the other hand for sparse graphs we show that the optimal algorithms for computing these values of all nodes cannot be truly subquadratic. These problems are, therefore, computationally no different from betweenness centrality.

We also show that one centrality measures that involves shortest paths passing *through* a particular node can be computed in $O(n^2)$ in planar graphs and no faster, making it a harder problem compared to the others but probably slightly easier compared to betweenness centrality which, as of now, requires $O(n^2 \log n)$ for planar graphs. One of the centralities that we introduce, *between number-farness centrality*, has a tight bound of $O(n^2)$ for one node and all nodes in the case of sparse graphs, putting it into a league of its own. Based on these results, we conjecture that for planar graphs, computing betweenness centrality of only a single node can possibly be done in subquadratic time but not of all nodes.

Acknowledgements

There is a long list of people I would like to thank. I would like to start chronologically. First of all, I would like to thank almighty (Generator Operator and Destroyer) for everything that I have. Words cannot describe how thankful I am to my parents. They have always prioritized me and my education over everything. They have always supported and encouraged me at each and every point of my life till date. I owe my success to them.

Before coming to IIITD, I had minimal knowledge about algorithm and complexity and knew some basic competitive coding. I never thought of doing research to this extent in theoretical computer science. The journey at IIITD began two years back with the orientation program and the refresher module. We had two subjects, one of which was taken by Dr. Syamantak Das. From that period only, my interest towards theoretical computer science escalated. Because of that, in the first semester only I took two courses in TCS. One of them was "Introduction to Graduate Algorithms" which was taught by my advisor Dr. Debajyoti Bera. Before attending this course, I did not even know that coming up with an algorithm is not enough, one also has a prove the correctness of that algorithm. After attending the lectures for about two weeks only, I was pretty sure of doing my thesis in TCS under him. The topic that he gave me was very new and challenging to me but at the same time it seemed very fascinating. I am grateful to him for introducing me to such an interesting concept. I am also thankful to him for all those discussions and meetings which helped me the most for completing this thesis. In addition to guiding me in the thesis, he also taught me numerous other things throughout this period of 1.5 years, which I know will be helpful in my future research career, starting from how to properly write emails to writing technically correct papers. Apart from these, he also taught me how to write formal proofs and when to stop thinking and start writing. Out of many, the one thing that I admire about him the most is his dedication towards students. He is always open to questions and encourages students to approach him openly. I know these words are not enough to thank him, but I would like to say that if I ever get a chance to become a professor in future, then I would really like to be like him.

At the same time I was also attending "Modern Algorithm Design" taught by Dr. Syamantak Das. He is the one who encouraged me to do a thesis in TCS and further pursue my career in research. He also taught me how to approach a problem more formally. He not only helped me understand many concepts in algorithm design, but also helped me choose my research career. I have also taken other courses taught by him and did TAship in some of his courses from which I have learnt a lot. I will forever be grateful to Syamantak sir for his guidance and would like to work with him in the future too.

This was the first time that I stayed away from my home and my stay at IIITD (second home) would not have been so enjoyable and enlightening without the group of friends whom I was lucky to have. They not only encouraged me but also taught me how to deal with the real world. I would especially like to thank Sonali, Saumya, Rachita, Sehaj, Maleeha, Rohit, etc. (the list is long) for being outspoken but supportive and for being a part of my life.

I would also like to thank my lab mates, especially Rahul da for introducing me to so many new concepts, for showing me some parts of Delhi and also guiding me through numerous tough situations.

I would also like to thank IIITD for providing such a nice campus environment and all the infrastructure, including the canteen,mess and housekeeping facilities which made the stay very comfortable. Staying away from home is very tough, especially when it comes to food. So, many many thanks to all the food delivery systems for delivering Bengali food at the doorstep. I would also like to thank everyone in the BMS and FMS. I would also like to thank the academic office of IIITD. Special thanks to Ashutosh Brahma sir because every time someone faced a problem related to exams or some other academic tasks, we used to say, "go to Brahma sir, he will fix it".

Although I have not explored much of the national capital, Delhi, but the places in Delhi that I have visited were full of life. Thank you to Delhi and the people of Delhi for being so welcoming and charming. I loved Delhi and would like to visit again.

It is very hard to find humble people who don't even know you and still answers your queries (no matter how stupid they are). I would like to convey my sincere gratitude to Prof. Sergio Cabello, Dr. Virginia Vassilevska Williams, Dr. Amir Abboud and last but not the least Dr. Kral Bringmann for answering my numerous queries. Overall, I would like to thank the entire theory community for being so accepting, approachable and always open to discussions.

Finally, I would like to convey my gratitude to my examiners, Prof. Saket Saurabh and Prof. Sandeep Sen for their valuable suggestions.

Contents

List of Figures

List of Tables

List of Abbreviations

BC	**B**etweenness **C**entrality
CC	**C**loseness **C**entrality
HC	**H**armonic **C**entrality
BHC	**B**etween **H**armonic **C**entrality
NFC	**N**umber **F**arness **C**entrality
BNFC	**B**etween **N**umber **F**arness **C**entrality
AWV	**A**dditively **W**eighted **V**oronoi (Diagram)

Dedicated to my dear friend Sehaj
and
to all those who are still waiting for their acceptance letter from Hogwarts.

Chapter 1

Introduction

–

Betweenness centrality (BC) is undoubtedly the most widely studied notion of graph centrality [28, 16]. BC of a node roughly measures the fraction of shortest paths that pass through a node; we think this is the reason that makes it attractive in various applications like congestion control [29], community detection [19], etc. Another reason behind its popularity among scientists, at least according to us, is a nifty algorithm proposed by Brandes that remains the starting point of all BC algorithms even today[5]. However, Brandes' algorithm (and all subsequent adaptations) are computationally expensive requiring $O(n^3)$; in fact, BC has been shown to be no easier than the all-pairs-shortest-paths problem [3]. In practice, therefore, many applications tend to use alternative notions of centrality, like closeness centrality, reach centrality, harmonic centrality, etc. all of which are also based on shortest-paths but *appear* simpler than BC. In fact, we found several recent applications that were computing closeness centrality and Harmonic centrality of large networks[35, 34].

But are these alternative centrality notions, particularly closeness centrality (CC), really simple? Can they be computed significantly faster than BC? It is known that BC cannot be solved in truly subquadratic time, but only for sparse graphs [3], and to the best of our knowledge, there are no truly-subquadratic upper bound or quadratic lower bounds known for closeness and Harmonic centralities for sparse and planar graphs.

Given the widespread applications of sparse networks, in this thesis, we address the above question for sparse graphs in general and planar graphs in particular. Our first result is that CC has a truly subquadratic solution for planar graphs [1]. Our preliminary attempts show that BC may require $\Omega(n^2 \log n)$ even for planar graphs; however, a formal lower bound, even under standard hardness assumptions, appears difficult. To understand their relative difficulties, let's compare their formulæ for BC and (unnormalized) CC given below in which we use "$s \rightsquigarrow t$" to denote the shortest path from s to t.

$$\begin{matrix}\text{Betweenness}\\\text{centrality}\end{matrix}\left(v\right) = \sum_{s,t \in V} \frac{\text{num. of } s \rightsquigarrow v \rightsquigarrow t}{\text{num. of } s \rightsquigarrow t} \qquad \begin{matrix}\text{Closeness}\\\text{centrality}\end{matrix}\left(v\right) = \frac{1}{\sum_{t \in V} \text{distance of } v \rightsquigarrow t}$$

The motivation of this thesis comes from the different nature of these expressions: use of number of shortest path [2] *vs.* the length of shortest paths, placement

[1] We use "truly subquadratic" to indicate a running time of $n^{2-\Omega(1)}$. We also use $\widetilde{O}$ to indicate additional poly-log(n) factors.

[2] For the centrality algorithms that uses the number of shortest paths, we have assumed that, addition and subtraction of very large numbers can be computed in $O(1)$ time (Real RAM model). Otherwise, even if the input graph has small weights, the number of shortest paths between a pair of nodes can be exponential, and that would add to the time overall time complexity.

Table 1.1: Centrality measures considered in this thesis. "num" denotes the number of shortest-paths and "dist" denotes the length of shortest-paths. Notations are described in Table 1.2.

Centrality	Definition	similar to ...	based on ...
Closeness	$CC\ (v) = \frac{1}{\sum_t dist(v,t)}$	SSSP	*dist*
Harmonic	$HC\ (v) = \sum_t \frac{1}{dist(v,t)}$	SSSP	*dist*
Between Harmonic	$BHC\ (v) = \sum_s \sum_t \frac{1}{dist(s,t,v)}$	APSP	*dist*
Number Farness	$NFC\ (v) = \sum_s \sigma(v,t)$	SSSP	*num*
Between Number Farness	$BNFC\ (v) = \sum_s \sum_t \sigma(s,t,v)$	APSP	*num*

of aggregation (summation) and the role of v in the relevant shortest paths as an end-point *vs.* an intermediate vertex. One notable difference is the use of shortest paths starting/ending at v (these are similar to "single-source shortest paths") *vs.* among all possible starting and ending nodes (these are similar to "all-pairs shortest paths"). We formulated several centrality measures, each focusing on one of those aspects — these measures are tabulated in Table 1.1. Out of these measures, Closeness, Harmonic and Between number farness (known as stress centrality in the literature) centrality are well established [32], but we could not spot the others in relevant literature. We did not consider $\frac{1}{\sum_s \sum_t dist(s,t,v)}$ as a separate centrality since it can be shown to be same as $CC(v)/n$. Although structurally similar, BHC on the other hand, is not equivalent to any of the other notions and is, in fact, harder than CC (see Table 1.3).

Table 1.2: Table of notations

Notations	Meaning
$dist(s,t)$	shortest path distance from s to t
$dist(s,t,v)$	shortest path distance from s to t among paths passing through v
$\sigma(s,t)$	number of shortest paths from s to t
$\sigma(s,t,v)$	number of shortest paths from s to t passing through v
n	number of nodes in the graph
P	piece obtained using r-division
∂P	boundary of piece P
$P(s)$	piece that contains a node s
$V(s)$	Voronoi region that contains a node s

We did not find any existing work on the upper or lower bound on the worst-case complexities of CC, HC and $BNFC$ (the others are defined by us). This led us to study the complexity of these centralities for planar and sparse graphs with a goal to understand the reasons behind the apparent hardness of computing BC for those graphs — is it the use of number of shortest paths, is it the placement of aggregation, is it the APSP style of considering all possible pairs of source and target nodes? We considered both the problems of computing a centrality of *a specific node* and computing the centrality of *all nodes*. The upper bounds and (conditional) lower bounds that we arrived at are summarized in Table 1.3 for planar graphs and Table 1.4 for sparse graphs.

The lower bounds that we obtain are conditional, i.e., based on the hardness of the 3SUM problem and the satisfiability problem. *3SUM hardness* is a commonly

Table 1.3: Upper and lower bounds of different centralities for *planar graphs*

Centrality	Upper Bound		Lower Bound
	Single Node	All nodes	All Nodes
CC	$O(n)$ [Note 1, Sec. 1.1]	$\widetilde{O}(n^{\frac{5}{3}})$ [Sec. 4.2.1]	Open
HC	$O(n)$ [Note 1, Sec. 1.1]	$\widetilde{O}(n^{\frac{9}{5}})$ [Sec. 4.2.2]	Open
BHC	$\widetilde{O}(n)$ [Sec. 4.1]	$O(n^2)$ [Note 2, Sec. 4.1]	3SUM-hard [Sec. 5.2]
NFC	$O(n)$ [Note 1, Sec. 1.1]	$\widetilde{O}(n^{\frac{9}{5}})$ [Sec. 4.2.3]	Open
$BNFC$	$\widetilde{O}(n^{\frac{9}{5}})$ [Sec. 4.2.4]	$O(n^2)$ [Note 2, Sec. 1.1]	Open

Table 1.4: Upper and lower bounds of different centralities for *sparse graphs*

Centrality	Upper Bound		Lower Bound	
	Single Node	All nodes	Single Node	All Nodes
CC	$\widetilde{O}(n)$ [Note 1, Sec. 1.1]	$\widetilde{O}(n^2)$ [Note 1, Sec. 1.1]	Open	not truly sub-quadratic [Sec. 5.1]
HC	$\widetilde{O}(n)$ [Note 1, Sec. 1.1]	$\widetilde{O}(n^2)$ [Note 1, Sec. 1.1]	Open	not truly sub-quadratic [Sec. 5.1]
BHC	$\widetilde{O}(n)$ [Sec. 4.1]	$\widetilde{O}(n^2)$ [Note 2, Sec. 4.1]	Open	3SUM-hard [Sec. 5.2]
NFC	$\widetilde{O}(n)$ [Note 1, Sec. 1.1]	$\widetilde{O}(n^2)$ [Note 1, Sec. 1.1]	Open	not truly sub-quadratic [Sec. 5.1]
$BNFC$	$\widetilde{O}(n^2)$ [Note 2, Sec. 1.1]	$\widetilde{O}(n^2)$ [Note 2, Sec. 1.1]	not truly sub-quadratic [Sec. 5.1]	not truly sub-quadratic [Note 3]

accepted belief that finding a, b, c in an array of numbers that sum to a specific target cannot be solved in truly subquadratic time. It is a common choice for proving quadratic lower bounds and has been used extensively for geometric problems and for some graph problems as well [3]. However, to the best of our knowledge 3SUM hardness has never been used for shortest-path based graph problems, let alone problems *wrt.* centrality.

The other common approach for obtaining *fine-grained complexity* is by reducing CNF-SAT to a problem and then using the *strongly exponential time hypothesis (SETH)* that the CNF-SAT problem cannot be solved in truly sub-exponential time. SETH was recently employed to show that truly subquadratic algorithms for exactly and approximately computing BC, and exactly computing reach centrality, probably do not exist.

For the upper bounds, we design algorithms based on graph-partitioning. Partitioning a graph into suitable regions, solving a problem on each region, and then combining the results over all regions has been shown to be useful in practical scenarios for many centrality problems [12, 33]. However, it was recently shown that even this technique does not give an asymptotically faster solution compared to running Brandes' algorithm for computing BC on sparse graphs [3]. So we resort to partitioning for planar graphs, and follow the footsteps of Cabello et al. [7] who showed how to use r-division and Voronoi decomposition to compute the largest shortest-path and sum of all shortest-paths in randomized subquadratic time. This was later improved to deterministic $\widetilde{O}(n^{5/3})$ [18]. However the complexity of a problem depends a lot on the exact expression to be calculated (e.g., $\max_{s,t} dist(s,t)$ *vs.* $\sum_{s,t} dist(s,t)$ *vs.* $\sum_{s,t} \frac{1}{dist(s,t)}$; hence, our proofs for CC, HC and NFC are different from those mentioned above. Although, we use the AWV decomposition from the paper and also use similar data structure ($ABBST$)

We can observe from the above tables that using number of shortest paths appears to be slightly more complicated compared to using only the shortest path distances, but the real difficulty creeps in when all (source, target) pairs are used to compute centrality of a node *ala.* APSP. Having both of these features, we conjecture that BC of even a single node in a planar graph probably cannot be computed in truly subquadratic time.

1.1 Overview of results

We consider undirected weighted n-node graphs denoted by $G = (V, E)$. First, we discuss a few simple upper and lower bounds that can be readily obtained using known techniques.

▶ Note 1. The CC, HC and NFC of single node can be calculated in $O(n)$ time by running a linear-time "single-source shortest path" algorithm (SSSP)[23] for planar graphs and by running Dijkstra's algorithm in $\widetilde{O}(n)$ time for sparse graphs. The CC, HC and NFC of all nodes in sparse graphs can be computed in $\widetilde{O}(n^2)$ time by simply running Dijkstra's algorithm n times. The centrality values can then be computed by storing the $dist$ or σ as necessary and adding up these values.

▶ Note 2. The BHC and $BNFC$ of all nodes can be computed in $\widetilde{O}(n^2)$ time for sparse graphs by using modified Brandes' algorithm[5]. The BHC and $BNFC$ of all nodes in planar graphs can also be calculated in $O(n^2)$ time using a modified Brandes' algorithm in which the linear time SSSP algorithm[23] is used instead of Dijkstra. In Brandes' algorithm, partial centrality values are "accumulated" at all nodes during backtracking. These accumulation rules have to be also modified for those centralities in the manner given below.

BHC(v)**:** For a source vertex s and for all $v \in pred(w)$, update $\text{BHC}(v) \leftarrow \text{BHC}(v) + \frac{1}{dist(s,w)} + \text{BHC}(w)$.

BNFC(v)**:** For a source vertex s and for all $v \in pred(w)$, update $\text{BNFC}(v) \leftarrow \text{BNFC}(v) + \text{BNFC}(w) \times (1 + \sigma(s, v))$.

Note that $BNFC$ of a single node in sparse graphs can also be computed using this technique in $\widetilde{O}(n^2)$ time.

▶ Note 3. The lower bound for $BNFC$ of all nodes follows from the same lower bound for $BNFC$ for a single node.

We present an algorithm to compute BHC of single node in planar and sparse graphs in $\widetilde{O}(n)$ time in Section 4.1 that uses a fast multipoint polynomial-evaluation technique [37, 4].

In Section 4.2 we present algorithms for computing CC, HC and NFC of all nodes of a planar graph in $o(n^2)$ time. We also present a truly-subquadratic algorithm for calculating $BNFC$ of a single node in planar graphs. These use Voronoi diagrams over r-divisions along with clever pre-processing and polynomial evaluation.

Then we derive a (conditional) lower bound for computing BHC of all nodes in a planar graph (and hence in a sparse graph) in Section 5.2 by constructing a reduction from the 3SUM problem.

Finally, in Section 5.1, we present reductions from the CNF-SAT problem to computing CC of all nodes, HC of all nodes, NFC of all nodes and $BNFC$ of a single node in sparse graphs. Thus, conditioned on the fact that SAT cannot be solved using $O(poly(n) \cdot 2^{\epsilon n})$ algorithms, it is unlikely that the above problems have truly subquadratic algorithms as well.

1.2 Road map

In chapter 2, we discuss the various tools and techniques needed for our upper and lower bounds. Next, in Chapter 3, we design efficient data structures needed for the subquadratic upper bound of planar graphs. Ultimately in Chapter 4, we give the upper bounds of various centralities in planar and sparse graphs. Next, in Chapter 5, we have proved the lower bound of some centralities in planar and sparse graphs. In Chapter **??**, we have introduced Betweenness Centrality and finally, in Chapter 6, we have concluded the overall thesis.

Chapter 2

Background

In this thesis, we consider both undirected planar graphs and their generalization, undirected sparse graphs [1]. The graphs are weighted with positive real edge weights. Planar graphs are graphs that can be drawn on a plane in such a way so that it's edges intersect only at their endpoints. A planar graph with n nodes can have at most $3n - 6$ edges. Sparse graphs need not always be planar but have $O(n)$ edges.Also, we use the terms vertices and nodes interchangeably throughout the thesis.

Now we discuss a few additional concepts used in our algorithms and lower bounds.

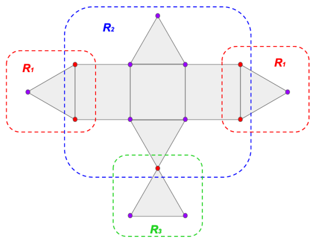
2.1 r-Division

The planar separator theorem proves that in any planar graph on n vertices, deletion of $O(\sqrt{n})$ number of vertices will make the graph disconnected. In other words, the remaining graph will have two "pieces" with no edges between them. Applying this theorem recursively, Frederickson[15] showed that an r-division of any planar graph with $O(n/r)$ *regions* or pieces can be constructed. A piece is an edge induced subgraph of G and all the pieces are pairwise edge disjoint. *Boundary* of a piece constitutes of vertices that have incident edges belonging to more than one piece. All other vertices, whose edges belong to only one piece, are referred to as *internal vertices*.

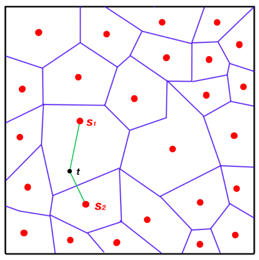
- Number of vertices in each piece is at most r.
- Number of edge-disjoint pieces is $O(\frac{n}{r})$.
- Boundary of each piece has $O(\sqrt{r})$ vertices.

There is no edge between any two internal nodes of different pieces; thus, the paths between such internal nodes have to pass through at least one boundary node. It should be noted that the subgraph corresponding to a piece may be disconnected (see Figure 2.1a). r-division takes $\widetilde{O}(n)$ time using the initial recursive approach by Frederickson[15]. However, in this thesis, we consider the r-division with constant number of *holes* per piece. *Holes* are faces of a piece, that are not the faces of graph G (Figure 2.2). This can be done using a linear time algorithm discussed in [30]. For this, they have assumed the graph is biconnected and triangulated ([38]). It can be proved that all the *holes* contain atleast one boundary vertex and every boundary vertex must lie on some hole. In fact, it can be proved that all vertices on a hole are boundary vertices if the graph is triangulated. The details are given in Appendix A.

[1] Although we have shown the algorithms for undirected graphs, but the same algorithms will also for directed graphs with very slight and obvious modifications. For the lower bounds, it gets naturally extended to directed version



(a) The *r-division* of a planar graph. Red dots represent boundary vertices. Note that the red piece R_1 is disconnected.



(b) The Voronoi Diagram of a planar graph showing sites in red. t is a vertex in the Voronoi cell of s_1 because $min_{s_i \in sites}\{dist(s_i, t)\} = dist(s_1, t)$

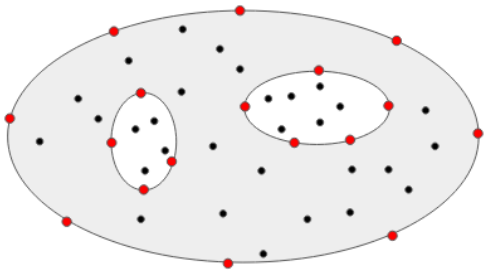


Figure 2.2: A piece with 3 *holes*. Red dots are the boundary nodes.

2.2 Assumptions for the Voronoi Diagram Construction

- The graph is triangulated (and hence biconnected–the graph cannot be disconnected by removing only 1 vertex). This can be achieved in linear time [38] , [26].

- We assume that the shortest path distance between every pair of nodes inside a piece P is distinct. We also assume that there exists only one shortest path between every pair of nodes inside P (except for the discrete set of critical values, where there exist exactly two shortest paths to exactly one vertex from the source, see section 2.3.1 and section 3.2). Both of these assumptions can be achieved by using deterministic lexicographic perturbation ([13, 8, 22]) with an additional factor of $\widetilde{O}(1)$ in the running time.

  Perturbations are nothing but adding small values to the edge weights of a piece P (values ϵ that are added to the edge weights are extremely small, this will ensure that shortest paths are preserved, i.e., the shortest path between any pair of nodes after perturbation was also one of the shortest paths before perturbation). Since, the shortest path distances are preserved, therefore it will not affect the centrality value (We need some additional techniques to handle NFC, because it uses the number of shortest paths instead of the distances). This is to ensure that a vertex is in one Voronoi cell only. This is also a preprocessing step of AWV decomposition.

- Holes are simple cycles and are disjoint–a boundary node lies in exactly one hole. This is done by creating multiple copies of some boundary nodes ([27], section 5.1). But only a constant number of vertices are copied, and the total number of vertices per piece remains $O(r)$ (and the number of boundary vertices remains $O(\sqrt{r})$).

2.3 Voronoi Diagram

Voronoi Diagram(VD) of a planar graph also represents a partitioning of the graph based on the distances of each vertex from a given set of *sites*, which are additionally specified special vertices. The partitions are known as *Voronoi regions* (or Voronoi cells), and there is exactly one Voronoi region for every site. The Voronoi region corresponding to a site comprises of those vertices for which that site, say s, is the closest among all sites. That is, if v belongs to a region corresponding to s then $dist(v,s)$ is the smallest among all $dist(v,s_i)$ for all sites s_i.

In the *additively weighted Voronoi (AWV)* decomposition problem, each site has a weight, say $w(s)$, and the task is to perform a Voronoi decomposition using a modified distance, say $dist'$, defined as $dist'(s,v) = w(s) + dist(s,v)$. There exist efficient algorithms for AWV, both randomized [7] and deterministic [18]. We will employ the latter in our algorithms, that constructs AWVD of pieces obtained from r-division.

▶ **Theorem 4.** *[[18], Theorem 1.1] Let P be a piece with r vertices and $O(\sqrt{r})$ sites $\{b_1, b_2, \ldots\}$ with weights on each site. One can preprocess P in $\widetilde{O}(r \times \sqrt{r}^2)$ time, so that, one can construct a representation of AWV diagram in $\widetilde{O}(\sqrt{r})$ time wrt to the weights on the sites.*

We denote the Voronoi cell or Voronoi region of a site u by $Vor(u)$–it contains the set of vertices that are inside the Voronoi cell of u. Let T_u be the shortest path tree rooted at u, consisting of all the nodes in piece P.

▶ **Lemma 5.** *[[18], Lemma 2.1] For each site u, the vertices in $Vor(u)$ form a connected subtree (rooted at u) of T_u.*

This is because we have assumed that the shortest paths are unique and the shortest distances between every pair of nodes are distinct. So, the shortest paths from u to all the nodes in T_u are the only shortest paths from u, therefore if a node t is not in $Vor(u)$, then none of it's descendants in T_u will be in $Vor(u)$. See figure 2.3, suppose t is not in $Vor(u)$ and t' is in $Vor(u)$ and t' is a descendant of t in T_u. So, $dist(u,t') = dist(u,t) + dist(t,t')$. Therefore t must be closer to some other site, say v . So, $dist(v,t) < dist(u,t)$, which will imply $dist(v,t') < dist(u,t')$, which is a contradiction. Also, note that the subtrees corresponding to each voronoi cell are disjoint, since the edge weights are perturbed and the distances are distinct. In other words, each vertex in P lies in exactly one voronoi cell.
Let the set of sites be S. The set of edges β are those edges (i,j), such that i and j lie in different Voronoi regions. Therefore β is the set of boundary edges. Let the set of duals of edges in β be denoted by β^*. Let P^* be the dual graph of P. The edge induced subgraph of P^* with the edge set as β^* is $VD^*(S)$.

▶ **Lemma 6.** *[[18], Lemma 2.2] The graph $VD^*(S)$ consists of atmost $|S|$ faces, so that each of its faces corresponds to a site $u \in S$ and is the union of all faces of P^* that are dual to the vertices of $Vor(u)$.*

This is an immediate consequence of Lemma 5. Since the vertices in primal graphs are the faces in dual graphs and vice versa, and the vertices in $Vor(u)$ for a connected tree in the primal graph, therefore the union of the faces dual to these vertices are connected.

Consider a piece P and two sites u and v. The *boundary* of the cell consist of all edges whose one endpoint is in $Vor(u)$ and the other endpoint is in $Vor(v)$. The

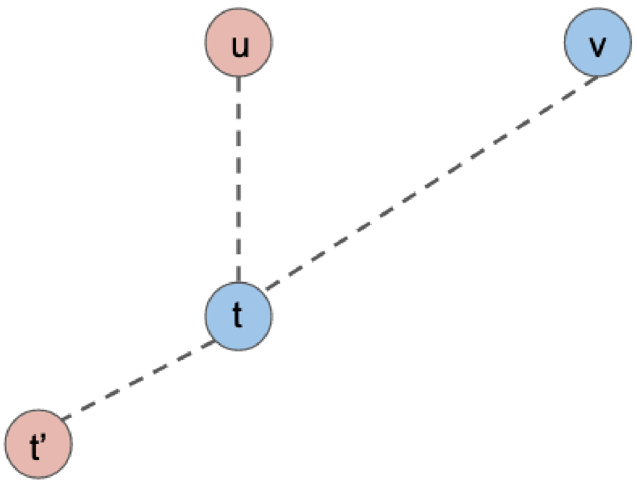
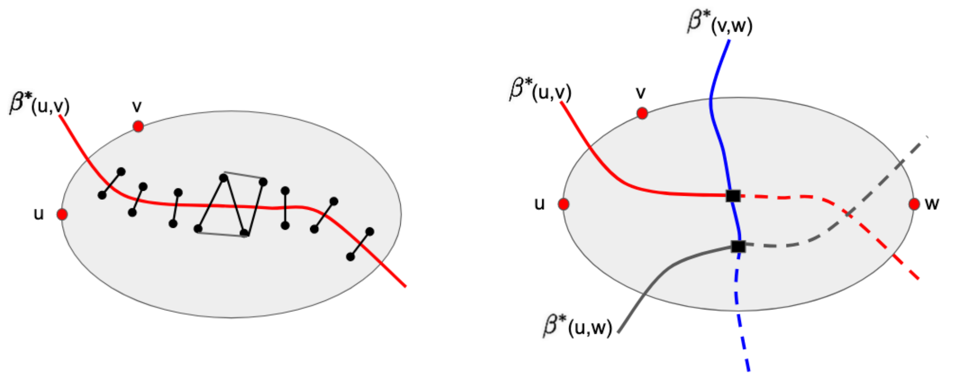


Figure 2.3: Two sites u and v. $t, t' \in T_u$ and T_v



(a) $\beta^*(u,v)$ with the primal edges that lie on the boundary of the voronoi regions.

(b) Part of the bisectors that appears in the voronoi diagram of 3 sites are shown using solid lines. Black dots are the voronoi vertices.

dual of these edges forms the $uv - bisector$ and is denoted by $\beta^*(u,v)$ (Refer to figure 2.4a). Only a part of $\beta^*(u,v)$ appears in the voronoi diagram whenever more sites are there (Figure 2.4b).

▶ **Lemma 7.** *[[18], Lemma 2.4] $\beta^*(u,v)$ forms a simple cycle in P^*. In other words, every u, v bisector is a simple cycle consisting of dual nodes and vertices from P.*

Since, the voronoi cells of u and v form a simple $uv-$cut, therefore the dual of the cut edges forms a simple cycle.

The dual vertex where more than one bisectors intersect is called a *voronoi vertex*. These are the faces whose vertices lie in more than two Voronoi cells. For simplicity we triangulate each face other than the *holes*, therefore the Voronoi vertices(except those that are dual to the *holes*) have degree 3. All the other vertices with degree 2 on the bisectors are contracted to form $voronoi - edges$.

The important step is the preprocessing step, where the bisectors for each pair of sites is computed. It can be shown that for a particular pair of sites there can only be $O(r)$ bisectors. These $O(r)$ bisectors for every pair can be computed in $\widetilde{O}(r)$ time.

Therefore the total preprocessing time is $O(\sqrt{r} \times \sqrt{r}) \times \widetilde{O}(r) = \widetilde{O}(r^2)$. The details are explained in section 2.3.1.
For the construction of the actual additively weighted Voronoi diagram, the appropriate bisector of every pair of sites is fetched from the preprocessing step. They are merged to get the actual diagram. It uses divide and conquer techniques to do that. This is rather involved and beyond the scope of this thesis. Ultimately, this step outputs segments of the pairwise bisectors that appear in the original Voronoi diagram.

2.3.1 Computing bisectors for every pair of sites

For every pair of sites (u, v) and every possible weight assignments on these sites, the bisectors are computed. Since the weights on the sites are not known during the preprocessing stage, there can be infinitely many weight assignments. The important point to note here is that, a bisector $\beta^*(u, v)$ depends on the weight difference, $\delta = w(v) - w(u)$. When $\delta = +\infty$, $Vor(u)$ contains no vertices other than u. Therefore as the value of δ changes from $+\infty$ to $-\infty$, the bisector shifts away from u. The shift in bisectors is also continuous. When the dual edge of the edge (p, q) (assuming that p is parent of q in T — the shortest path tree in P rooted at u) is part of some bisector $\beta^*(u, v, .)$, it means that p is in $Vor(u)$ and q is in $Vor(v)$.

Let $\delta^{uv}(p) = dist(u, p) - dist(v, p)$ for some internal node p in P. We call an edge (p, q) 'tense edge' if p is in $Vor(u)$ and q is in $Vor(v)$, i.e. the dual edge of (p, q) is part of the bisector. Note that (p, q) remains tense for $dist(u, p) - dist(v, p) < \delta < dist(u, q) - dist(v, q)$ (i.e. $\delta^{uv}(p) < \delta < \delta^{uv}(q)$). Technically, when $\delta = dist(u, p) - dist(v, p)$, then p will be in both $Vor(u)$ and $Vor(v)$, because additive distance from u to p is equal to the additive distance from v to p. We need the voronoi cells to be disjoint, therefore we break the tie and assume that p is in $Vor(v)$, when $\delta = dist(u, p) - dist(v, p)$. Therefore we have the following lemma.

▶ **Lemma 8.** *(p, q) remains tense for $dist(u, p) - dist(v, p) < \delta \leq dist(u, q) - dist(v, q)$ (i.e. $\delta^{uv}(p) < \delta \leq \delta^{uv}(q)$).*

We first compute all the critical values of δ for which the bisector changes. As described above, these are nothing but $\delta^{uv}(p)$ for all internal nodes p in P. Algorithm is given in section 3.

▶ **Lemma 9** ([18], *Lemma 3.1*). *Consider some critical value δ. The dual edges that newly join $\beta^*(u, v)$ at δ form a contiguous portion of the new bisector (let's call this linked-list/contiguous portion $\beta_{in}(u, v, \delta)$), and the dual edges that leave $\beta^*(u, v)$ when the critical value shifts from δ' to δ ($\delta' > \delta$) form a contiguous portion of the old bisector (let's call this linked-list/contiguous portion $\beta_{out}(u, v, \delta')$).*

Note the contiguous portion of the old bisector may or may not form a cycle. In case it forms a cycle, the entire old bisector is replaced by the new bisector(See Figure 2.5). Details of how to construct β_{in} and β_{out} are given in algorithm 1.

▶ **Lemma 10.** *An edge if deleted from a bisector $\beta^*(u, v)$ for some δ will not be added again in the bisector for any critical value (weight difference) $< \delta$.*

This is because the dual edge of (p, q) remains a part of the bisector $\beta^*(u, v)$ when the δ is in between $\delta^{uv}(p)$ and $\delta^{uv}(q)$ (lemma 8), where (p, q) is the primal edge. Since the bisector for every possible δ are constructed by iterating δ from $+\infty$ to $-\infty$, therefore the edge if deleted, won't be added again.

▶ **Lemma 11** ([18], *Section 3*). *Total number of edges and vertices that are added and deleted to and from the bisector $\beta^*(u,v)$ for every possible δ is $O(r)$.*

From lemma 10, we can say that total number of edges added and deleted can be $O(r)$ because there are $O(r)$ edges per piece and an edge can be added to the bisector only once.

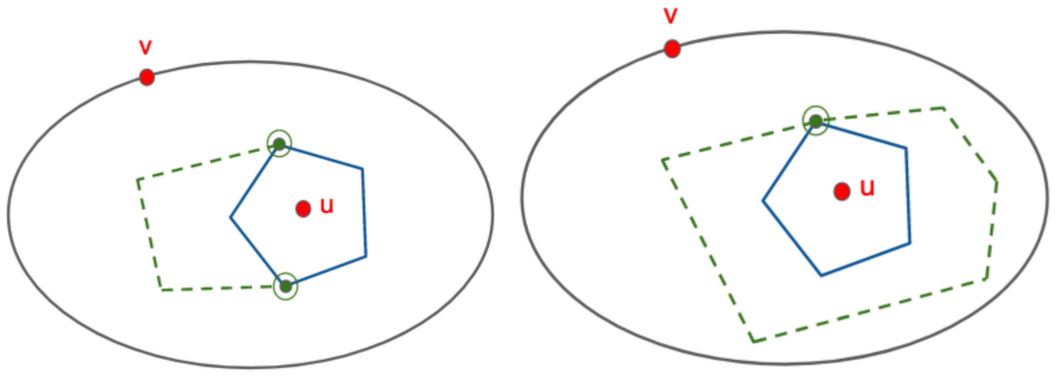


Figure 2.5: The contiguous portion of the new bisector that gets added is shown in dotted line.

2.3.2 Additional Structure and Properties of Additively Weighted Voronoi Diagram

For every boundary node b, the boundary of the voronoi cell of b (i.e. $Vor(b)$) is represented as segments of bisectors $\beta^*(b,.,.)$. The boundary of a voronoi cell is a collection of non-self crossing cycles $\mathcal{C}^*$, where $|\mathcal{C}^*| \leq t(t$ is the number of holes in piece P).

▶ **Lemma 12** ([18], *Section 7*). *Each cycle either encloses a unique hole or passes through the dual node representing a unique hole.*

Let h be the hole of b and h^* be the dual vertex of hole h. Let C_0^* be the cycle in $\mathcal{C}^*$ that passes through h^*. If $\nexists$ any such cycle then we can always introduce a dummy cycle C_0^* which will enclose all the other cycles in $\mathcal{C}^*$ and will be a self-loop through h^*.

Let T^* be the cotree of T(shortest path tree in piece P rooted at b). A face f is said to be in $Vor(b)$ if all it's vertices are in $Vor(b)$.

▶ **Lemma 13** ([18], *Lemma 7.1*). *For every root to leaf path in T^*, $\exists$ atmost one edge f^*g^* on the path, where $f^* \leftarrow parent(g^*)$ in T^*, such that f^* is on C_0^* and either g^* is on some other cycle in $\mathcal{C}^*$ (this is because there may exists an edge (a^*, b^*) in T^*, such that a^* and b^* lie on different cycles) or face g is in $Vor(b)$. We call such an edge a 'penetrating edge'. Also $\nexists$ any f^*g^* edge on the path, such that f^* is on the cycle C_0^* and no vertex of face g is in $Vor(b)$, i.e. g is entirely outside $Vor(b)$.*

In other words, any root to leaf path in T^* enters $Vor(b)$ through the cycle C_0^* atmost once, and once it enters the cycle, it never leaves $Vor(b)$ through C_0^*. As it can be seen from the figure 2.6[2], the black dots are the g^*($parent(g^*)$ in T^* in on C_0^*)

[2] This figure is from the paper [18] and it does not entirely match our setting. In our setting all the vertices on the holes are boundary vertices.

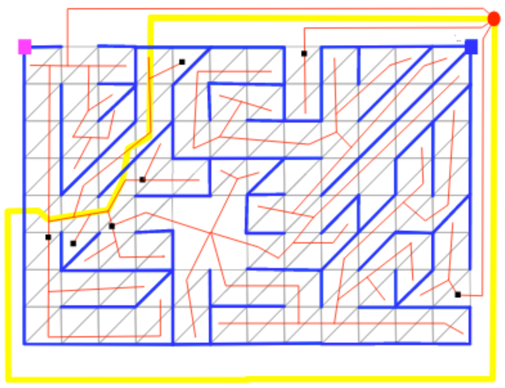


Figure 2.6: Two boundary nodes on the outer hole are shown in blue(boundary node b) and pink. The yellow line is the C_0^* cycle of the bisector $\beta^*(b,.)$. Red lines are edges of T^* and blue lines are the edges of T.

where the root to leaf path enters the cycle C_0^* into $Vor(b)$(blue vertex on the top right). And there is no root to leaf path of T^* that exists $Vor(b)$ through C_0^*.
It is clear from the above lemma that any vertex in $Vor(b)$ is either adjacent to the cycle C_0^*[i.e. lies on a face f whose dual vertex f^* is on C_0^*] or lies on a face y such that y^* belongs to the subtree of T^* rooted at some g^* and $(.,g^*)$ is the edge that enters C_0^*(i.e. g^* is a black dot in the figure 2.6).

▶ **Corollary 14.** *For any node t^* such that face t is in $Vor(b)$, there exist a unique ancestor of t^* in T^* (let's say q^*) that is adjacent to the cycle C_0^*, i.e there exists an edge (p^*,q^*) in T^*, such that p^* is on C_0^* and q^* is not on C_0^* (face q is inside $Vor(b)$) (See figure 2.6).*

We shall design efficient data structures to store the necessary information on these bisector segments(or cycles) by exploiting the properties of the cotree and the edges of the cotree that enter or leave the cycles.

▶ Note 15. Cotree is used just to ensure that a vertex in $Vor(b)$ is considered exactly once while calculating the centralities. We could have used the shortest path tree rooted at b(i.e., T), but the difficulty arises because the bisector can cut a branch twice, and it would have been difficult to store the information of the entire branch. To do this, we would have to traverse T from root to every node that were pruned by the bisector.

Properties of Voronoi diagram in the presence of multiple holes

The first part of the lemma 13 and corollary 14 still hold for multiple holes case. However, the second part is not true as a root to leaf path can enter and exit a cycle, which is not C_0^*.

▶ **Lemma 16** ([18], *Section 7*). *Consider a cycle $C^* \in \mathcal{C}^*$. For every root to leaf path in T^*, $\exists$ atmost one edge f^*g^* (penetrating edge) on the path such that f^* is on a cycle C^* and either g^* is on some other cycle in $\mathcal{C}^*$ (this is because there may exists an edge a^*, b^* in T^*, such that a^* and b^* lie on different cycles) or face g is in $Vor(b)$ ($f^* \leftarrow Parent(g^*)$ in T^*).*

Every root to leaf path in T^* will have atmost one penetrating edge from each cycle in $\mathcal{C}^*$. But note that, the path may contain multiple penetrating edges from different cycles Figure 2.8.

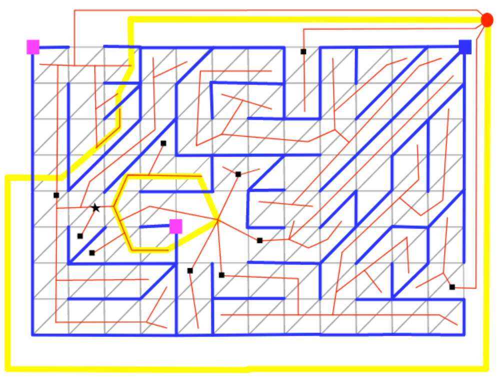


Figure 2.7: There are two holes in this piece. Yellow lines represent the boundary of the Voronoi region of b(blue node). Red lines represent the cotree. Blue lines are the edges of the shortest path tree rooted at b. Black dots are the endpoints of the edges of cotree that are entering into the Voronoi region of b through the cycles. Black star represents the unique edge that is exiting the Voronoi cell of b through the cycle inside.

▶ **Corollary 17.** *Consider a cycle $C^* \in \mathcal{C}^*$. For any node t^* such that face t is in $Vor(b)$, there exist a unique ancestor of t^* in T^* (let's say q^*) that is adjacent to the cycle C^*, i.e there exists an edge (p^*, q^*) in T^*, such that p^* is on C^* and q^* is not on C^* (face q is inside $Vor(b)$) (See figure 2.7).*

▶ **Lemma 18** ([18], ***Lemma 7.2, Corollary 7.3***). *For every cycle $C^* \in \mathcal{C}^* \setminus \{C_0^*\}, \exists$ exactly one edge $f^*g^* \in T^*$ ($f^* \leftarrow Parent(g^*)$ in T^*) such that g^* lies on C^* and either f^* lies on some other cycle in $\mathcal{C}^*$ or face f is in $Vor(b)$ C^*. We call such an edge the 'exiting edge'. Suppose $h' \neq h$ be a hole and C^* encloses h'. Then the unique edge f^*g^* lies on the path from h^* to h'^* in T^*.*

As it can be seen in figure 2.7[3] that the endpoint of the unique edge is represented by a black star.

▶ **Lemma 19** ([18], ***Section 7.1***). *The unique edge (exiting edge) in lemma 18 can be found in $O(1)$ time per cycle using a data structure which can be constructed in $\widetilde{O}(r)$ time per piece per boundary node.*

The vertices in $Vor(b)$ are those that are enclosed by C_0^* and not enclosed by any other cycle in $\mathcal{C}^*$.

▶ **Lemma 20.** *Consider a cycle $C^* \in \mathcal{C}^* \setminus \{C_0^*\}$. Let (f^*, g^*) be the exiting edge of the cycle C^*. All the nodes p^* on the cycle C^* are in the subtree of f^* in T^*.*

Proof. This is by the definition of an exiting edge. If there exists another node q^* on cycle C^* whose ancestor in T^* that is not on C^* is not f^* , there will exist another exiting edge, which is a contradiction. ◀

▶ **Corollary 21.** *Consider a cycle $C^* \in \mathcal{C}^* \setminus \{C_0^*\}$. All the penetrating edges from the cycle C^* are in the subtree (of T^*) of the exiting edge of C^*.*

▶ **Lemma 22.** *For any node $f^* \in T^*$, let Path be the root to f^* path in T^*:*

- *If Path has 0 nodes that lie on some cycle in $\mathcal{C}^*$, then f^* lies outside $Vor(b)$.*

[3] Figure is taken from the paper [18].

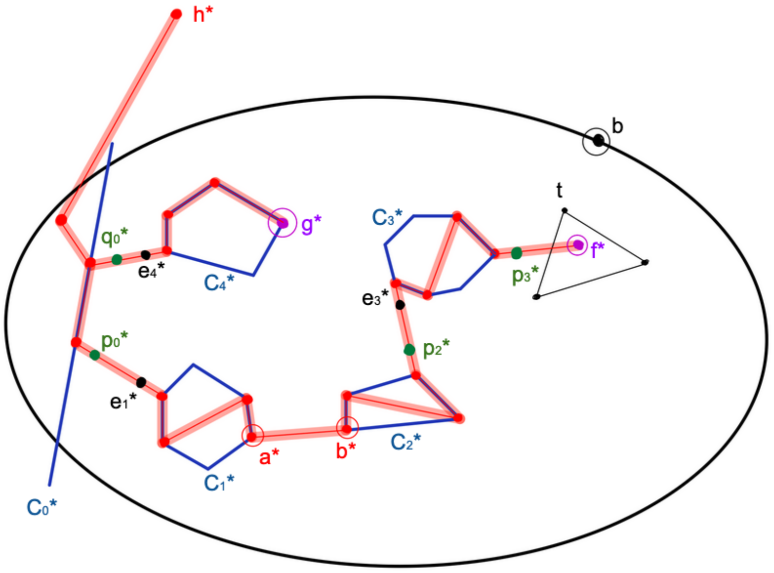


Figure 2.8: A piece with a voronoi region is shown. Boundary of $Vor(b)$ is shown in blue. Two paths ($h^* \rightsquigarrow g^*$ and $h^* \rightsquigarrow f^*$) from T^* are shown in red. g^* lie on a cycle, and face f is in $Vor(b)$. Endpoints of penetrating edges are shown in green and endpoints of exiting edges are shown in black. Edge (a^*, b^*) is both a penetrating edge and an exiting edge.

- *If f^* lies on C_0^*, then Path has only 1 node which lies on some cycle in $\mathcal{C}^*$.*
- *If there are x (atmost $|H|$) penetrating edges on Path then there can be either $x-1$ exiting edges on Path or x exiting edges on Path.*
  - *If there are x (atmost $|H|$) penetrating edges on Path and face f is in $Vor(b)$ then there will be $x-1$ exiting edges on Path.*
  - *If there are x (atmost $|H|$) penetrating edges on Path and f^* lies on a cycle $C^* \in \mathcal{C}^* \setminus \{C_0^*\}$, then there will be x exiting edges on Path. Moreover, the last exiting edge on Path will be the exiting edge of cycle C^*.*

Proof. • If *Path* has 0 nodes which lie on some cycle, then *Path* contains no penetrating edges or exiting edges, and since for every node p^* (such that face p has some nodes which are in $Vor(b)$), root to p^* path must have atleast 1 node which lie on some cycle, therefore, face f does not belong to $Vor(b)$.

- Since, there does not exist any exiting edge for C_0^*, therefore, if f^* lies on C_0^*, then *Path* will have 1 node which lie on some cycle.
- For any cycle other than C_0^*, there will be a unique exiting edge per cycle. Also, from corollary 21, we know that the exiting edge is a common ancestor of the penetrating edges of a cycle, therefore, there can be atmost one penetrating edge per cycle on *Path* (if there are more than 1 penetrating edge of a cycle on *Path*, then the path must also have more than one exiting edges of that cycle—corollary 17). Every penetrating edge on *Path* must also have a corresponding exiting edge. Therefore, if there are $x-1$ penetrating edges for $x-1$ cycles (excluding C_0^*), then there will be $x-1$ exiting edges.

If face f is in $Vor(b)$, then there will be a penetrating edge of cycle C_0^*. Therefore the total number of penetrating edges will be x.

If f^* is on a cycle C^*, then the cycle C^* does not have any penetrating edge on $Path$ (otherwise, there will be more than one exiting edge of that cycle). Excluding the exiting edge of C^*, there will be $x-2$ exiting edge and $x-1$ penetrating edges (including the penetrating edge of C_0^*). Therefore, total number of exiting edge=total number of penetrating edge = $x-1$.

◀

2.3.3 Representation of the Voronoi Diagram — same representation used in [18]

The Voronoi diagram is represented as segments of bisectors (Figure 2.4b). The Voronoi cell of a site b may consist of a sequence of segments of bisectors $\beta^*(b,.)$. All the degree 2 nodes on the bisector segments are contracted. Therefore, the Voronoi diagram is represented as a reduced graph using the DCEL data structure (refer to [10]), which is used to represent planar graph embeddings.

By Lemma 23, the path $(f^*,\dots,g^*)$, which is a contiguous portion of some bisector $\beta^*(u,v,.)$ is represented as a single edge (f^*,g^*). Along with the endpoints, some extra pointers are also stored, so that given the voronoi vertices (endpoints) (f^*,g^*), one can identify in constant time, the adjacent sites u,v, and the bisector $\beta^*(u,v,.)$ from which the segment was taken.

▶ **Lemma 23** ([18], *Lemma 5.1*). *If f^* and g^* are two consecutive voronoi vertices on the common boundary between $Vor(u)$ and $Vor(v)$, then the path between f^* and g^* along this boundary forms a connected segment of the bisector $\beta^*(u,v,.)$.*

2.3.4 Some more structural properties of AWV Diagram

Since our setting is a bit different, therefore we are going to introduce some more properties of the AWV diagram in this subsection.

▶ **Lemma 24.** *The cycles in $\mathcal{C}^*$ other than C_0^* do not pass through any hole, rather they enclose a unique hole.*

Proof. Since in our setting, each hole only contains boundary vertices, therefore if any cycle C^* in $\mathcal{C}^* \setminus \{C_0^*\}$ does not enclose a unique hole and passes through it (suppose it passes through h'), then there will be some boundary vertex b' on h' that is not enclosed by C^*. This boundary vertex cannot lie inside $Vor(b)$ and the cycles do not cross, therefore there must exist some other cycle that can enclose b', but then it must either pass through h' or enclose it, which is a contradiction because two cycles cannot enclose or pass through the same hole. ◀

As in figure 2.9, it can be seen that if the inner cycle does not enclose the hole h_1, then atleast one boundary vertex will be outside that cycle and that boundary vertex will then either be in $Vor(b)$ or is enclosed by some other cycle which is not possible (as a cycle corresponds to a unique hole).

▶ **Lemma 25.** *All the dual nodes on the cycles in $\mathcal{C}^*$ have degree ≤ 3 (except h^* which is on the cycle C_0^*).*

Proof. Let $\{h_1,h_2,\dots,h_t\}$ be the holes and h be the hole of the boundary vertex b. From lemma 24 we can say that none of the h_i^* lie on any of the cycles in $\mathcal{C}^* \setminus \{C_0^*\}$.

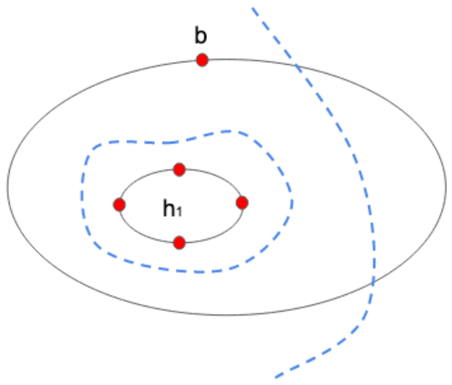


Figure 2.9: A piece with two holes, the infinite hole h_0 and h_1. h_1 has 4 boundary nodes. The boundary of $Vor(b)$ is shown in dotted blue.

And since the faces of P other than the holes are triangulated, therefore the nodes on these cycles will have degree ≤ 3. ◀

▶ **Lemma 26.** *The cycles in $\mathcal{C}^*$ do not cross one another.*

Proof. There can be two ways in which the cycles can cross(Figures 2.10):

- Two cycles meet at a point:
  In this case, we can see that the point where the two cycles meet has degree 4, which is not possible according to lemma 25.

- Two cycles meet at more than one points:
  In this case, the common points where the cycles meet are not part of the bisector of $Vor(b)$, therefore removing that part will create a new cycle that encloses two holes, which is not possible according to lemma 12.

◀

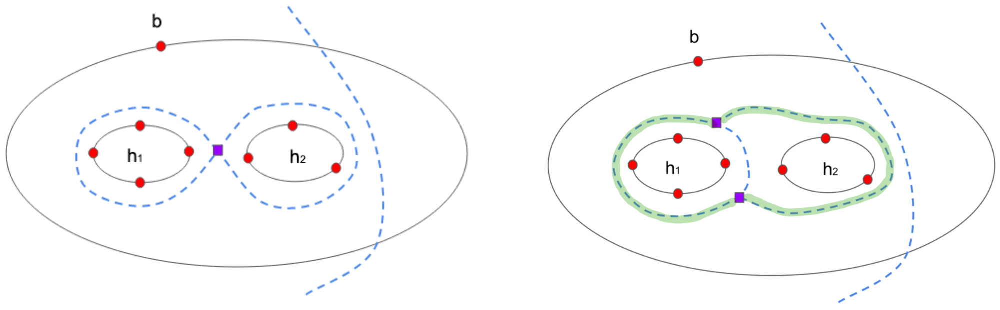


Figure 2.10: The cycles in blue cannot cross like this.

As given in the paper [18] too, the cycles in $\mathcal{C}^* \setminus \{C_0^*\}$ are not nested because $Vor(b)$ is connected(Figure 2.11).

Since the AWV construction was mainly designed to handle problems that require shortest path distances, it is not immediately clear how to use this to our advantage for designing a subquadratic algorithm for NFC. The main challenge is that, AWV construction assumes that the edge weights are perturbed, therefore there exist

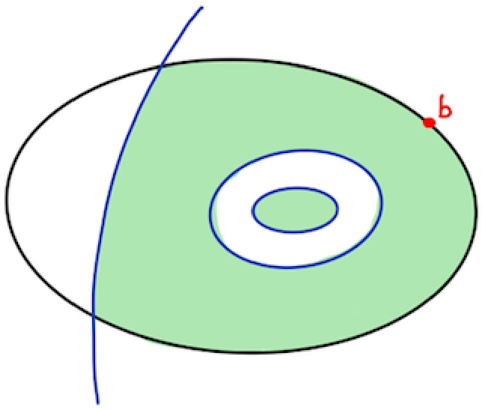


Figure 2.11: Green region is $Vor(b)$, which is disconnected if the cycles in $\mathcal{C}^*$ are nested.

a unique shortest path between every pair of vertices in G. Since NFC requires the number of shortest paths between pairs of vertices, AWV diagram cannot be used as it is.
We have mainly defined the properties of a Voronoi cell on the basis of cotree. But it can be observed that similar properties also hold if we consider the shortest path tree T.

▶ **Lemma 27.** *Any root to leaf path of the shortest path tree T (shortest path tree in P rooted at b) crosses any cycle $C^* \in \mathcal{C}^*$ atmost once.*

Proof. We prove this by contradiction. Let us assume that there exist a root to leaf path of T that crosses C^* twice. Since b is inside $Vor(b)$, therefore the path will first leave the voronoi cell and again re-enter (See figure 2.12a), which is not possible since $Vor(b)$ is a connected subtree of T (lemma 5). ◀

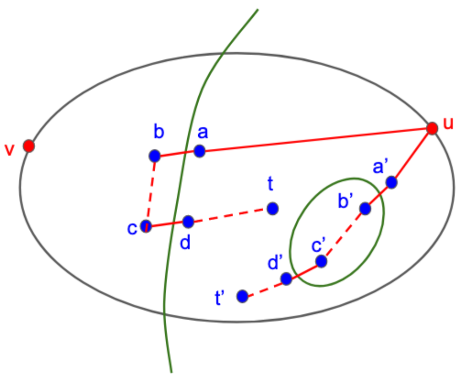
▶ **Corollary 28.** *For any node t which is not in $Vor(b)$, there exist a unique ancestor of t in T (let's say q) that is adjacent to the boundary, i.e there exists an edge pq in T, such that p is inside $Vor(b)$ and q is outside $Vor(b)$ (See figure 2.12b). In other words, there does not exist any other node q' outside $Vor(b)$ which is adjacent to the boundary and is an ancestor of t.*

2.4 Fast Multipoint Evaluation

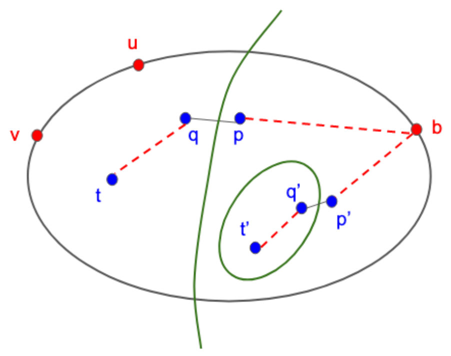
Given a polynomial $P(x)$ of degree d in coefficient form and n arbitrary points, $P(x)$ can be easily evaluated in $O(nd)$ time by evaluating the value of each x for each term of $P(x)$. However we make use of a faster algorithm that can evaluate a degree-d polynomial in coefficient form on n arbitrary points in $O(\max(n,d)\log^2 d)$ time [37, 4].

2.4.1 Converting polynomials in root form to coefficient form

Consider a list of n numbers $A = \{a_1, \ldots, a_n\}$. Consider a polynomial $Poly(x) = \sum_i \frac{1}{a_i+x}$. This polynomial can also be written as $\frac{P(x)}{Q(x)}$, where $P(x) = \sum_{a_i \in A} \prod_{a_j \in A \setminus \{a_i\}} (a_j +$



(a) Two paths of T are shown that are crossing both the cycles twice, which is not possible as proved in Lemma 27



(b) Two outside nodes t and t' are shown. q is the ancestor of t and q' is the ancestor of t' which are adjacent to the boundary (shown in green).

$x)$ and $Q(x) = \prod_{a_i \in A}(a_i + x)$. We call the first form as Form α_1 (i.e. $Poly(x)$) and the second form as Form α_2.

Algorithm 0: $P1(x)$ and $P2(x)$ in their coefficient forms

```
input : $\{a_1, a_2, \ldots a_n\}$ representing
        degree-$(n-1)$ polynomial $P1(x) = \sum_{i \in \{1,n\}} \prod_{j \in \{1,n\}, j \neq i} (a_j + x)$ and
        degree-$n$ polynomial $P2(x) = \prod_{i \in \{1,n\}} (a_i + x)$
output: Coefficients of $P1(x)$ and $P2(x)$
1 If $n = 1$, return $P1(x) = 1$ and $P2(x) = (a_1 + x)$
2 Define four polynomials:
    $P1_{left}(x) = \sum_{i \in \{1,\frac{n}{2}\}} \prod_{j \in \{1,\frac{n}{2}\}, j \neq i} (a_j + x)$,    $P1_{right}(x) = \sum_{i \in \{\frac{n}{2}+1,n\}} \prod_{j \in \{\frac{n}{2}+1,n\}, j \neq i} (a_j + x)$
    $P2_{left}(x) = \prod_{i \in \{1,\frac{n}{2}\}} (a_i + x)$,    $P2_{right}(x) = \prod_{i \in \{\frac{n}{2}+1,n\}} (a_i + x)$
3 Recursively obtain coefficient forms of $P1_{left}(x)$ and $P2_{left}(x)$ by calling this
    algorithm on $\{a_1, \ldots a_{n/2}\}$.
4 Recursively obtain coefficient forms of $P1_{right}(x)$ and $P2_{right}(x)$ by calling
    this algorithm on $\{a_{n/2+1}, \ldots a_n\}$.
5 Compute $P1(x) = P1_{left}(x) \times P2_{right}(x) + P1_{right}(x) \times P2_{left}(x)$ in which
    FFT is used for polynomial multiplication
6 Compute $P2(x) = P2_{left}(x) \times P2_{right}(x)$ using FFT
7 Return $P1(x)$ and $P2(x)$
```

The recurrence relation for the above recursive algorithm is $T(n) = 2T(\frac{n}{2}) + O(n \log n)$ yielding a solution of $O(n \log^2(n))$.

2.5 Complexity Assumptions

Let F be a CNF-SAT formula on n variables; we can use the Sparsification lemma to assume *wlog* that F contains $m = O(n)$ clauses [25]. The trivial algorithm for

CNF-SAT involves trying out all assignments and requires $O(poly(n) \cdot 2^n)$ time. The *Strong Exponential Time Hypothesis* (SETH) states that CNF-SAT cannot be solved in time $O((2-\delta)^n)$ [25].

The *3SUM problem* takes as input a set of n integers $\{a_1, a_2, \ldots, a_n\}$ and asks if there exists $a_i, a_j, a_k, i \neq j \neq k$, such that $a_i + a_j = a_k$. According to the 3SUM conjecture, it was believed that the lower bound of any deterministic algorithm for 3SUM is $\theta(n^2)$. But in 2014 it was refuted by Allan Grønlund and Seth Pettie[21] who gave a deterministic algorithm that runs in slightly subquadratic time $O(\frac{n^2}{(\log n/\log\log n)^{\frac{2}{3}}})$. They also gave the complexity on a decision tree model that requires $O(n^{1.5}\sqrt{\log n})$ time. The logarithmic factors were improved over time [9, 17, 20]. But there is no deterministic truly subquadratic algorithm that can run on a real RAM model for 3SUM till date. In this thesis we consider the version, where $a_i \neq 0$. The conjecture still holds for this version.

2.6 Related Work

As mentioned in Chapter 1, papers [7, 18] compute the diameter of a planar graph in truly-subquadratic time using Voronoi decomposition. Diameter of a graph G with n nodes and $O(n)$ edges is given by the following equation: $Diam(G) = \max_{s,t} dist(s,t)$ for all nodes s, t in G. The overall high-level steps of the algorithm in both the papers are the same. They start with a planar graph G and computes the $r - division$ of G with $O(1)$ holes. After that, the algorithm consists of 3 main steps: (i) Finding the maximum distance when s is a boundary node. This can be done by simply running SSSP algorithm from each boundary node. (ii) Finding the maximum distance when s and t belong to the same piece. The details of this step are explained in our upper bounds. It basically creates a graph G_P with the vertices of P only and with $O(r)$ edges, then finally, it runs APSP algorithm in G_P, which gives the maximum distance within that piece. (iii) Finding the maximum distance when s and t are internal nodes of different pieces. This step requires the AWV decomposition, so that the maximum distance in each Voronoi cell can be found in $O(\sqrt{r})$ time. Since every shortest path will pass through some boundary node of P, therefore, the maximum distance of a node in each Voronoi cell $Vor(b)$ added with the distance from s to each boundary node b will give us the maximum distance when s and t lie in different pieces. Note the three steps (except the AWV decomposition) along with the $r - division$ are common steps for any planar APSP based algorithms like [15, 24, 31]. Our upper bounds in section 4.2 also follow these steps along with the use of AWV decomposition.

We have used the "same" AVW decomposition as given in the paper [18] (with additional properties —the proofs of which are given in section 2.3). All the results (Lemmas and Theorems) from the paper (like Theorem 4,Lemma 5, Lemma 6, etc.) have the relevant section or theorem numbers from the paper (along with the citation). In this thesis, we have used a slightly different variant of AWVD—all the vertices on the holes are boundary vertices—due to this, we were able to prove some new properties of AWVD which were used in the upper bounds (like Lemma 8,22,24,etc.). Although the AWVD of the paper also uses a triangulated graph (they needed the triangulation for a different reason—which has been used in our paper too), but they haven't mentioned the other properties explicitly (mainly the properties that involve holes to include only boundary vertices). In fact they have assumed that their holes can have internal nodes too (figure from the paper that we used–Figure 2.6,Figure 2.7).

The data structure (Persistent binary search tree - PBST) that they use in the paper for finding the diameter, can also be used for CC (they have mentioned it in their introduction section, but have not proved it). This data structure is not enough for the other centralities (especially NFC and $BNFC$ because they uses number of shortest paths instead of shortest path distances). For the centralities, we have designed AL and $ABBST$ (along with the augmented cotree—which was also introduced in the paper). $ABBST$ is similar to PBST in a way that both are persistent binary search trees on the nodes of the bisectors, but they have different structures. For HC, $ABBST/AL$ alone is not sufficient, therefore, we have also designed HT for each piece. For NFC, we have a new type of AL (although similar to the AL mentioned before, but also has information from outside the piece)— the whole augmentation is different. For NFC especially, we also have proved some new properties of AWVD (the properties in other paper were not sufficient enough) like Lemma 27,58 and Corollary 28 (some new Lemmas are also proved in Section 4.2.3).

And finally, for computing the diameter, they have accumulated the information on the bisectors (stored as PBST) in a very efficient way using the data structure. We have also used the same notion of fetching the information stored on the bisectors (in our case $AL/ABBBST$), the techniques used are different from that of the paper, because we have used different data structures (especially for HC and NFC).

Chapter 3

Data Structure

Let T be the shortest path tree of P(only consist of vertices that are in piece P) rooted at b (b is a boundary node in P). Let T^* denote the set of dual edges of P that are not in T. It can be shown that T^* forms a tree — such a tree is known as a *cotree* (also known as *interdigitating tree* in the literature).

▶ Note 29. How can we build the cotree? The data structure that is used for representing a planar graph embedding (DCEL), also shows the dual graph (if not, then also the dual graph can be found in linear time). So, we can find out the edges that are not in T in linear time, which is essentially the cotree.

▶ Note 30. We also keep track of the pointers to every node $f^* \in T^*$, such that any node can be found in $O(1)$ time in T^*. We also assume that a node f^* can be referred and accessed from any data structure in $O(1)$ time.

3.1 Augmented Cotree

First, we build the augmented cotree T^*, i.e., add additional information to the nodes of T^*.

- For every dual vertex $f^* \in T^*$ (f not a hole), store the list of vertices on the face f, call this value $count\_list(f^*)$. If two faces are adjacent, then the common vertices are associated with only one face. This will ensure that we do not double count any vertex.
- We traverse T^* and store another value $nodes\_list(f^*)$ for each node f^*. $nodes\_list(f^*)$ stores the union of $count\_list(g^*)$ for all g^* in the subtree of f^* in T^*, including $count\_list(f^*)$ itself.

3.1.1 Augmented Cotree for CC

Given the cotree T^* and the unperturbed distances($Dist()$) from every vertex from b, we augment the cotree in the following manner for computing CC.

- For every dual vertex $f^* \in T^*$ (f not a hole), store the number of vertices associated with the face f, call this value $count(f^*) = |count\_list(f^*)|$. We can see that $count(f^*) \leq 3$.
- We also calculate and store $dist\_dual(f^*) = \sum_{v \in count\_list(f^*)} Dist(b, v)$ for each node $f^* \in T^*$.
- We traverse T^* and store another value $count\_sum(f^*)$ for each node f^*. $count\_sum(f^*)$ stores the sum of $count(g^*)$ for all g^* in the subtree of f^* in T^*, added with $count(f^*)$ itself.

- While traversing the cotree we also store another value $dist\_dual\_sum(f^*)$ for each node f^*. $dist\_dual\_sum(f^*)$ stores the sum of $dist\_dual(g^*)$ for all g^* in the subtree of f^* in T^*, added with $dist\_dual(f^*)$ itself.

▶ **Theorem 31.** *Augmented cotree for CC can be constructed in $\widetilde{O}(n\sqrt{r})$ time for a all the boundary nodes of all pieces in G.*

Proof. This only involves traversing the cotree and shortest path trees in a pieces for a single boundary node. Building the shortest path tree require $\widetilde{O}(r)$ time per boundary node, and the corresponding cotree can be constructed in $O(r)$ time. Therefore all the augmented cotrees can be constructed in $\widetilde{O}(\frac{n}{r} \times \sqrt{r} \times r) = \widetilde{O}(n\sqrt{r})$ time. ◀

3.1.2 Augmented Cotree for HC

Since, calculating HC, require the fractional terms, $\frac{1}{dist(.,.)}$, the augmentation is not as simple as in CC. We need to store the following for every dual node in T^*:

- We also calculate and store a list $dist\_list1(f^*) = \bigcup_{v \in count\_list(f^*)} Dist(b, v)$ for each node $f^* \in T^*_b$.
- While traversing the cotree we also store another list $dist\_list2(f^*)$ for each node f^*. $dist\_list2(f^*)$ stores the union of $dist\_list1(g^*)$ for all g^* in the subtree of f^* in T^*, including $dist\_list1(f^*)$ itself. Note that this can be done in $O(r)$ time per piece by traversing the cotree from leaves to roots and updating the lists as follows:
  - If current node f^* is a leaf node, then $dist\_list2(f^*) = dist\_list1(f^*)$.
  - If current node f^* is not a leaf node with *children* being the list of children of f^* in T^*, then do $dist\_list2(f^*).append(dist\_list2(g^*))$ for all $g^* \in children$.

  We also store 2 polynomials: $P1_2(f^*)(x)$ and $P2_2(f^*)(x)$ along with an id. Polynomials are constructed with the values in $dist\_list2(p^*)$ using algorithm 0.

▶ **Theorem 32.** *Augmented cotree for HC can be constructed in $\widetilde{O}(nr^{\frac{3}{2}})$ time for a all the boundary nodes of all pieces in G.*

Proof. Along with traversing the cotree, we also construct polynomials at each node of the cotree. Since the size of the list $dist\_list2()$ can be $O(r)$, therefore it will take $\widetilde{O}(r)$ time to construct the polynomials at each node. Therefore the augmented cotree for HC can be constructed in $\widetilde{O}(\frac{n}{r} \times \sqrt{r} \times r \times r) = \widetilde{O}(nr^{\frac{3}{2}})$ time. ◀

3.1.3 Augmented Cotree for NFC

This is almost the same as in CC, only instead of storing the $dist()$ we store the number of shortest paths σ.

- We calculate and store $num\_paths1(f^*) = \sum_{v \in count\_list(f^*)} \sigma(b, v)$ for each node $f^* \in T^*$.
- We traverse T^* and store another value $num\_paths2(f^*)$ for each node f^*. $num\_paths2(f^*)$ stores the sum of $num\_paths1(g^*)$ for all g^* in the subtree of f^* in T^*, added with $num\_paths1(f^*)$ itself.

▶ **Theorem 33.** *Augmented cotree for NFC can be constructed in $\widetilde{O}(n\sqrt{r})$ time for a all the boundary nodes of all pieces in G.*

The proof is exactly same as theorem 31.

3.2 Computing β_{in} and β_{out}

Let us take two boundary nodes u, v in P. At first the critical values of δ are computed and stored in $c\_val(u, v)$. They are nothing but the $\delta^{uv}(t) = dist(u, t) - dist(v, t)$ for each node $t \in P$.
We also store the unperturbed δ's in $\delta.original$. This is needed for NFC and $BNFC$.
As described in section 2.3.1, lemma 9, we will construct the lists: $\beta_{in}(u, v, \delta)$ *and* $\beta_{out}(u, v, \delta)$ for all possible critical values δ. Let δ' be the immediate predecessor of δ in c_val, then $\beta_{in}(u, v, \delta)$ will represent the contiguous portion(dual edges) of the bisector $\beta^*(u, v, \delta)$ that was newly added at δ and $\beta_{out}(u, v, \delta)$ will represent the contiguous portion of the old bisector $\beta^*(u, v, \delta')$ that got deleted at δ.

We will now figure out the δ for which each dual edge (p^*, q^*) enters the bisector (insert p^*, q^* into $\beta_{in}(u, v, \delta)$) and the δ' for which (p^*, q^*) leaves the bisector (insert p^*, q^* into $\beta_{out}(u, v, \delta')$). This is given in lemma 8.

```
Algorithm 1: Computing $\beta_{in}$ and $\beta_{out}$ and $c\_val$
  input  : Piece $P$, boundary nodes $u, v$ in $P$
  output: $\beta_{in}(u,v,.)$ and $\beta_{out}(u,v,.)$
  /* We first compute all the critical values of $\delta = w(v) - w(u)$ for
     which the bisectors shift.  We store the critical values in
     the list $c\_val(u,v)$.  */
1 $c\_val(u,v).append(+\infty)$;
2 for each vertex $t \in P$ do
3     $\delta \leftarrow dist(u,t) - dist(v,t)$;
4     $\delta.original \leftarrow Dist(u,t) - Dist(v,t)$;
5     $c\_val(u,v).append(\delta)$;
6 end
7 $c\_val(u,v)$ ←sort $c\_val$ in decreasing order;
  /* We now create two doubly linked lists:  $\beta_{in}(u,v,\delta)$ and
     $\beta_{out}(u,v,\delta)$ for each element $\delta \in c\_val$.  */
  /* Let $\delta'$ be the immediate predecessor of $\delta$ in $c\_val$, then
     $\beta_{in}(u,v,\delta)$ will represent the contiguous portion(dual edges) of
     the bisector $\beta^*(u,v,\delta)$ that was newly added at $\delta$ and $\beta_{out}(u,v,\delta)$
     will represent the contiguous portion of the old bisector
     $\beta^*(u,v,\delta')$ that got deleted at $\delta$.  */
8 for every edge $(p,q) \in P$ do
9     if $\delta^{uv}(p) < \delta^{uv}(q)$ then
          /* $(p^*,q^*)$ will be part of the bisector $\beta^*(u,v,\delta)$ for
             $\delta^{uv}(p) < \delta \leq \delta^{uv}(q)$.  Therefore, $(p^*,q^*)$ will join the
             bisector for the smallest $\delta > \delta^{uv}(p)$ and will leave the
             bisector for the smallest $\delta > \delta^{uv}(q)$ */
10        $\delta_{in}$ ←Find the smallest element in $c\_val$ which is $> \delta^{uv}(p)$.
11        $\delta_{out}$ ←Find the smallest element in $c\_val$ which is $> \delta^{uv}(q)$.
12        $\beta_{in}(u,v,\delta_{in}).add(p^* \leftrightarrow q^*)$;
13        $\beta_{out}(u,v,\delta_{out}).add(p^* \leftrightarrow q^*)$;
14    if $\delta^{uv}(p) > \delta^{uv}(q)$ then
          /* $(p^*,q^*)$ will be part of the bisector $\beta^*(u,v,\delta)$ for
             $\delta^{uv}(q) < \delta \leq \delta^{uv}(p)$.  Therefore, $(p^*,q^*)$ will join the
             bisector for the smallest $\delta > \delta^{uv}(q)$ and will leave the
             bisector for the smallest $\delta > \delta^{uv}(p)$.  */
15        $\delta_{in}$ ←Find the smallest element in $c\_val$ which is $> \delta^{uv}(q)$.
16        $\delta_{out}$ ←Find the smallest element in $c\_val$ which is $> \delta^{uv}(p)$.
17        $\beta_{in}(u,v,\delta_{in}).add(p^* \leftrightarrow q^*)$;
18        $\beta_{out}(u,v,\delta_{out}).add(p^* \leftrightarrow q^*)$;
      /* If $\delta^{uv}(p) = \delta^{uv}(q)$, then $(p^*,q^*)$ cannot be part of any
         bisector, because for any $\delta$, either both $p,q$ will be in
         $Vor(u)$ or both will be in $Vor(v)$.  */
19 end
  /* We fix a direction and specify the ‘start’(or ‘head’) and
     ‘end’(or ‘tail’) of the doubly linked lists by traversing each
     list, i.e., if for a pair of nodes $p^*,q^*$, $p^*$ appears before $q^*$
     in some list, then the order will be same in all the lists
     that have both $p^*$ and $q^*$.  */
```

▶ **Theorem 34.** *$\beta_{in}(u,v,\delta)$ and $\beta_{out}(u,v,\delta)$, for all pairs of boundary nodes u,v in piece P and for all pieces can be constructed in $\tilde{n}r$ time.*

Proof. For a pair of boundary nodes u,v in piece P, $c\_val(u,v)$ can be constructed in $O(r)$ time, as it only calculates $dist(u,t) - dist(v,t)$ for each internal node t in P. Sorting $c\_val(u,v)$ wil require $\widetilde{O}(r)$ time. For each edge, we check when will it's dual enter and leave the uv-bisector . This check can be done in $O(1)$ time per edge. Also, we need to traverse every $\beta_{in}(u,v,\delta)$ and $\beta_{out}(u,v,\delta)$ for all possible δ for fixing the start and end of each list. This will incurr an additional $O(r)$ cost for each pair of bisector, as only $O(r)$ vertices can be added and deleted (lemma 11). Therefore, the total running time of algorithm 1 is $\widetilde{O}(r)$ for each pair of bisector.
Computing this for all the pieces will take $\widetilde{O}(\frac{n}{r} \times \sqrt{r}^2 \times r) = \widetilde{O}(nr)$ time. ◀

For efficiently computing the centralities, we build the required data structure using the already augmented cotree,β_{in} and β_{out} . We propose two simple data structures: *Augmented List*(AL) and *Augmented Balanced Binary Search Tree*$(ABBST)$.

3.3 Augmented List Data Structure

We maintain AL for every bisector $\beta^*(u,v,\delta)$. The number of elements in AL is equal to the number of dual vertices on $\beta^*(u,v,\delta)$. The list will contain a key and list of augmented values. Keys will be the dual vertices on the bisectors.

We first construct the empty $AL(u,v,\delta)$ with the keys inserted (unaugmented AL). This can be done by inserting β_{in} and deleting β_{out} at the appropriate values of δ (lemma 9).

Algorithm 2: Initializing $AL(u,v,.)$

```
   input  : $c\_val(u,v), \beta_{in}(u,v,.), \beta_{out}(u,v,.)$
   output: unaugmented $AL(u,v,.)$
   /* Let $h$ be the hole of boundary node $u$ and $h^*$ be the dual node
      of $h$ */
 1 $AL(u,v,+\infty) \leftarrow h^*$;
 2 $l \leftarrow length(c\_val)$;
 3 $i \leftarrow 0$;
 4 while $i < l - 1)$ do
 5     $\delta \leftarrow c\_val[i+1]$;
 6     $\delta' \leftarrow c\_val[i]$;
 7     $AL(u,v,\delta) \leftarrow NULL$;
 8     $AL(u,v,\delta).delete\_segment\_AL(\beta_{out}(u,v,\delta'))$;
 9     $AL(u,v,\delta).insert\_segment\_AL(\beta_{in}(u,v,\delta))$;
10     $i+=1$;
11 end
```

Initializing $AL(u,v,.)$ will take $O(r)$ time for each pair of boundary nodes u,v in piece P, because $delete\_segment\_AL()$ and $insert\_segment\_AL()$ require $O(1)$ time (endpoints can be identified and accessed in $O(1)$ time—note 30, therefore, it is just insertion and deletion in a linked list). Therefore, for all pieces and all pairs of boundary nodes, it will require $O(nr)$ time.

3.3.1 AL for CC

We augment four values to each dual node p^* in $AL(u, v, .)$:

- $num\_nodes(p^*)$: This is the sum of $count(f^*)$ for all f^* in the subtree of p^* in T^*, added with the number of nodes on the face p, that are closer to u than to v (additively).
- $num\_nodes1(p^*)$: This is the sum of $num\_nodes(q^*)$ for all q^* which are successors of p^* in $AL(u, v, .)$, excluding $num\_nodes(p^*)$.
- $sum\_dist(p^*)$: This is the sum of $dist\_dual(f^*)$ for all f^* in the subtree of p^* in T^*, added with the $Dist(u, t)$ of nodes t on the face p, that are closer to u than to v (additively).
- $sum\_dist1(p^*)$: This is the sum of $sum\_dist(q^*)$ for all q^* which are successors of p^* in $AL(u, v, .)$, excluding $sum\_dist(p^*)$.

Algorithm 3: Augmenting in $AL(u, v, .)$

```
input  : Augmented cotree $T^*$, initialized $AL(u,v,.)$ and $c\_val(u,v)$
output: Augmented $AL(u,v,.)$
1  for every $\delta \in c\_val(u,v)$ do
2      Let $AL(u,v,\delta)$ be the initialized $AL$;
3      for every dual node $f^*$ in $AL(u,v,\delta))$ do
4          $num\_nodes(f^*) \leftarrow 0$.
5          $sum\_dist(f^*) \leftarrow 0$.
6          for every vertex $t \in count\_list(f^*)$ do
7              if $dist(u,t) - dist(v,t) < \delta$ then
8                  $num\_nodes(f^*) += 1$;
9                  $sum\_dist(f^*) += Dist(u,t)$;
10         end
11         for every child $g^*$ of $f^*$ in $T^*$ do
12             if $g^*$ is not adjacent to $f^*$ in $AL(u,v,\delta)$ and all the vertices on the face
                $g$ is closer to $u$ than to $v$ then
13                 $num\_nodes(f^*) += count\_sum(g^*)$
14                 $sum\_dist(f^*) += dist\_dual\_sum(g^*)$
15         end
16     end
17     $l^* \leftarrow$ last node in $AL(u,v,\delta)$;     /* end of $AL(u,v,\delta)$.  */
18     $num\_nodes1(l^*) \leftarrow 0$;
19     $sum\_dist1(l^*) \leftarrow 0$;
20     $f^* \leftarrow pred(l^*)$ in $AL(u,v,\delta)$;
       /* Traverse $AL(u,v,\delta)$ from last node(tail) to first(head),
          $pred(l^*)$ calls the immediate predecessor $l^*$ in $AL(u,v,\delta)$.  */
21     while $f^*$ is not $NULL$ do
22         $num\_nodes1(f^*) = num\_nodes1(l^*) + num\_nodes(l^*)$;
23         $sum\_dist1(f^*) = sum\_dist1(l^*) + sum\_dist(l^*)$;
24         $l^* \leftarrow f^*$;
25         $f^* \leftarrow pred(l^*)$;
26     end
27 end
```

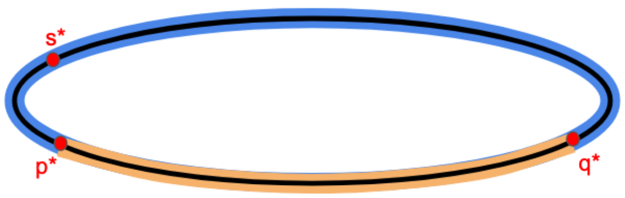
The above algorithm takes $O(r)$ time for each bisector. There can be $O(r \times \sqrt{r})$ bisectors for every boundary node of a piece. Therefore, for all possible bisectors of all boundary nodes in all pieces, it will take $O(\frac{n}{r} \times \sqrt{r} \times (r \times \sqrt{r}) \times r) = O(nr^2)$.

▶ **Lemma 35.** *Given a segment $(p_i^*, \dots, p_j^*)$ of bisector $\beta^*(b,.,.)$ (maintaining the same cyclic order as in AL), the sum of $num\_nodes(p_k^*)$ for all $p_k^* \in \{p_i^*, \dots, p_j^*\}, p_k^* \notin \{p_i^*, p_j^*\}$ and sum of $sum\_dist(p_k^*)$ for all $p_k^* \in \{p_i^*, \dots, p_j^*\}, p_k^* \notin \{p_i^*, p_j^*\}$ can be computed in $O(1)$ time.*

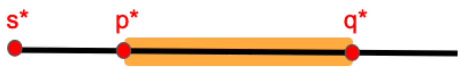
Proof. Since each bisector is a simple cycle (from lemma 7), therefore given two points on the bisector we can get two possible segments (See figure 3.1). We will use one of the following steps to get the desired value:

- If p^* appears before q^* in $AL(b,.,.)$ (Figure 3.1b):
  We need to look for the nodes that are after p^* and before q^* in $AL(b,.,.)$. We will return $num\_nodes1(p^*) - [num\_nodes1(q^*) + num\_nodes(q^*)]$ and $sum\_dist1(p^*) - [sum\_dist1(q^*) + sum\_dist(q^*)]$.
- If p^* appears after q^* in $AL(b,.,.)$ (Figure 3.1c):
  Here we need to look for the nodes that are before p^* and after q^* in $AL(b,.,.)$. Let s be the first node in $AL(b,.,.)$(i.e. the head if $AL(b,.,.)$). We will return $[num\_nodes1(s) + num\_nodes(s)] - \{[num\_nodes1(q^*) + num\_nodes(q^*)] - num\_nodes1(p^*)\}$ and $[sum\_dist1(s) + sum\_dist(s)] - \{[sum\_dist1(q^*) + sum\_dist(q^*)] - sum\_dist1(p^*)\}$.

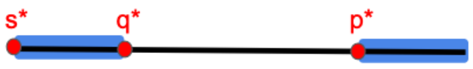
◀



(a) A pairwise bisector is shown as a simple cycle. s^*, p^*, q^* are the dual vertices on the bisector and s^* is the starting vertex of the bisector in AL.



(b) Here p^* appears before q^* in AL



(c) Here p^* appears after q^* in AL

Figure 3.1: A bisector as a simple cycle and the two possible segments (when the end points of the segments are given) is shown. The two possible segments are highlighted in different colors.

Therefore we have the following theorem.

▶ **Theorem 36.** *We can build a data structure in $O(nr^2)$ for all pieces in G, such that it has the property stated in the lemma 35.*

3.3.2 AL for HC

We augment the following to each dual node p^* in $AL(u,v,.)$:

- $dist\_list3(p^*)$ (along with polynomials $P1_3(p^*)(x)$ and $P2_3(p^*)(x)$): This is the union of $dist\_list1(f^*)$ for all f^* in the subtree of p^* in T^*, including the $Dist(u,t)$ of nodes t on the face p, that are closer to u than to v (additively). Corresponding polynomials are constructed with the values in $dist\_list3(p^*)$ using algorithm 0.
- $dist\_list4(p^*)$ (along with polynomials $P1_4(p^*)(x)$ and $P2_4(p^*)(x)$): This is the union of $dist\_list3(q^*)$ for all q^* which are successors of p^* in $AL(u,v,.)$, excluding $dist\_list3(p^*)$. Again corresponding polynomials are constructed with the values in $dist\_list4(p^*)$ using algorithm 0.

Algorithm 4: Augmenting in $AL(u,v,.)$

```
   input  : Augmented cotree $T^*$, initialized $AL(u,v,.)$ and $c\_val(u,v)$
   output: Augmented $AL(u,v,.)$ for $HC$
1  for every $\delta \in c\_val(u,v)$ do
2      Let $AL(u,v,\delta)$ be the initialized $AL$;
3      for every dual node $f^*$ in $AL(u,v,\delta))$ do
4          $dist\_list3(f^*) \leftarrow NULL$.
5          for every vertex $t \in count\_list(f^*)$ do
6              if $dist(u,t) - dist(v,t) < \delta$ then
7                  $dist\_list3(f^*).append(Dist(u,t))$;
8          end
9          for every child $g^*$ of $f^*$ in $T^*$ do
10             if $g^*$ is not adjacent to $f^*$ in $AL(u,v,\delta)$ and all the vertices on the face
                $g$ is closer to $u$ than to $v$ then
11                 $dist\_list3(f^*).append(dist\_list2(g^*))$;
12         end
13         $P1_3(f^*), P2_3(f^*) \leftarrow$ Construct the Polynomials using 2.4.1 and the
            values as $dist\_list3(f^*)$;
14     end
15     $l^* \leftarrow$ last node in $AL(u,v,\delta)$;     /* end of $AL(u,v,\delta)$ */
16     $dist\_list4(l^*) \leftarrow NULL$;
17     $f^* \leftarrow pred(l^*)$ in $AL(u,v,\delta)$
       /* Traverse $AL(u,v,\delta)$ from last node(end) to first(start),
          $pred(l^*)$ calls the immediate predecessor $l^*$ in $AL(u,v,\delta)$.  */
18     while $f^*$ is not NULL do
19         $dist\_list4(f^*) = dist\_list4(l^*).append(dist\_list3(l^*))$;
20         $l^* \leftarrow f^*$;
21         $f^* \leftarrow pred(l^*)$;
22     end
23     for every dual node $f^*$ in $AL(u,v,\delta)$ do
24         $P1_4(f^*), P2_4(f^*) \leftarrow$ Construct the Polynomials using 2.4.1 and the
            values as $dist\_list4(f^*)$;
25     end
26 end
```

The above algorithm takes $O(r \times r\log r)$ time for each bisector. There can be $O(r \times \sqrt{r})$ bisectors for every boundary node of a piece. Therefore, for all possible bisectors of all boundary nodes in all pieces, it will take $O(\frac{n}{r} \times \sqrt{r} \times (r \times \sqrt{r}) \times r^2 \log r) = \widetilde{O}(nr^3)$.

▶ **Lemma 37.** *Given a segment $\gamma = \{p^*, \ldots, q^*\}$ of bisector $\beta^*(b,.,.)$ (maintaining the same cyclic order as in AL), value of $\sum_{f^* \in \gamma \setminus \{p^*, q^*\}} \frac{P1_3(f^*)(x)}{P_3(f^*)(x)}$ for any x, can be calculated in $O(1)$ time (given that we already know the values of polynomials $P1_4(f^*)(x), P2_4(f^*)(x), , P1_3(f^*)(x), P2_3(f^*)(x)$ for all nodes $f^* \in \gamma$).*

Proof. Since each bisector is a simple cycle (from lemma 7), therefore given two points on the bisector we can get two possible segments (Figure 3.1a). We will use one of the following steps to get the desired value:

- If p^* appears before q^* in $AL(b,.,.)$ (Figure 3.1b):
  We need to look for the nodes that are after p^* and before q^* in $AL(b,.,.)$. We will return $\frac{P1_4(p^*)(x)}{P2_4(p^*)(x)} - [\frac{P1_4(q^*)(x)}{P2_4(q^*)(x)} + \frac{P1_3(q^*)(x)}{P2_3(q^*)(x)}]$.
- If p^* appears after q^* in $AL(b,.,.)$ (Figure 3.1c):
  Here we need to look for the nodes that are before p^* and after q^* in $AL(b,.,.)$. Let s be the first node in $AL(b,.,.)$(i.e. the head if $AL(b,.,.)$). We will return $[\frac{P1_4(s)(x)}{P2_4(s)(x)} + \frac{P1_3(s)(x)]}{P2_3(s)(x)} - \left([\frac{P1_4(p^*)(x)}{P2_4(p^*)(x)} + \frac{P1_3(p^*)(x)}{P2_3(p^*)(x)}] - \frac{P1_4(q^*)(x)}{P2_4(q^*)(x)}\right)$.

◀

Using the above lemma, we know the exact polynomials to fetch (in $O(1)$ time) when a segment of bisector is given. We will use the following algorithm(5) for this:

Algorithm 5: Fetching list of polynomials needed for Lemma 37

```
input : $AL(u,v,\delta)$, segment of bisector $\gamma = \{p^*, \ldots, q^*\}$
output: List of polynomials and their corresponding signs
1  $Polynomials[] \leftarrow Null$; /* Each entry of $Polynomial[]$ contain two
       polynomials and a sign (+ or -).  */
2  if $p^*$ appears before $q^*$ in $AL(b,.,.)$ then
       /* All the polynomials are from $AL(u,v,\delta)$.  */
3      $Polynomials[].insert((P1_4(p^*)(x)), (P2_4(p^*)(x)), +)$;
4      $Polynomials[].insert((P1_4(q^*)(x)), (P2_4(q^*)(x)), -)$;
5      $Polynomials[].insert((P1_3(q^*)(x)), (P2_3(q^*)(x)), -)$;
6  if $p^*$ appears after $q^*$ in $AL(u,v,\delta)$ then
       /* All the polynomials are from $AL(u,v,\delta)$.  */
7      $s \leftarrow$ first node in $AL(u,v,\delta)$;
8      $Polynomials[].insert((P1_4(s^*)(x)), (P2_4(s^*)(x)), +)$;
9      $Polynomials[].insert((P1_3(s^*)(x)), (P2_3(s^*)(x)), +)$;
10     $Polynomials[].insert((P1_4(q^*)(x)), (P2_4(q^*)(x)), -)$;
11     $Polynomials[].insert((P1_3(q^*)(x)), (P2_3(q^*)(x)), -)$;
12     $Polynomials[].insert((P1_4(p^*)(x)), (P2_4(p^*)(x)), +)$;
13 return $Polynomials[]$;
```

Therefore we have the following theorem.

▶ **Theorem 38.** *We can build a data structure in $O(nr^3)$ for all pieces in G, such that it has the property stated in the lemma 37.*

3.3.3 AL for NFC

We augment the following to each dual node p^* in $AL(u,v,.)$:

- $num\_paths3(p^*)$: This is the sum of $num\_paths1(f^*)$ for all f^* in the subtree of p^* in T^*, added with the $\sigma(u,t)$ of nodes t on the face p, that are closer to u than to v (additively).
- $num\_paths\_edge(p^*)$: Suppose (p^*, q^*) is in $AL(u, v, .)$ and q^* is successor of p^*. Let (p, q) be the dual edge of (p^*, q^*). $num\_paths\_edge(p^*)$ contains the sum of $\sigma(u,t)$ of nodes t in the subtree of q (including q), which are equi-distant from both u and v, additively (i.e. after adding weights to u, v).
- $num\_paths4(p^*)$: This is the sum of $num\_paths3(q^*) + num\_paths\_edge(q^*)$ for all q^* which are successors of p^* in $AL(u, v, .)$, excluding $sum\_dist(p^*)$. $num\_paths\_edge(p^*)$ is also added.

Algorithm 6: Augmenting in $AL(u,v,.)$

```
input  : Augmented cotree $T^*$, initialized $AL(u,v,.)$,$c\_val(u,v)$ and the
         shortest path tree in $P$ rooted at $u(T)$
output: Augmented $AL(u,v,.)$ for $NFC$
1  for every $\delta \in c\_val(u,v)$ do
2      Let $AL(u,v,\delta)$ be the initialized $AL$;
3      for every dual node $f^*$ in $AL(u,v,\delta))$ do
4          $num\_paths3(f^*) \leftarrow 0$;
5          for every vertex $t \in count\_list(f^*)$ do
6              if $dist(u,t) - dist(v,t) < \delta$ then
7                  $num\_paths3(f^*) + = \sigma(u,t)$;
8          end
9          for every child $g^*$ of $f^*$ in $T^*$ do
10             if $g^*$ is not adjacent to $f^*$ in $AL(u,v,\delta)$ and all the vertices on the face
                $g$ is closer to $u$ than to $v$ then
11                 $num\_paths3(f^*) + = num\_paths2(g^*)$;
12         end
13     end
       /* Traversing the shortest path tree and looking for the nodes
          that are not in $vor(u)$. */
14     for every edge $e = (f^*, g^*)$ on $AL(u,v,\delta)$ do
15         Let $pq(p \leftarrow parent(q)$ in $T)$ be the primal edge of $e$;
16         $num\_paths\_edge(e) \leftarrow 0$;
17         for every vertex $t$ in the subtree of $q$ including $q$ in $T$ do
18             if $Dist(u,t) - Dist(v,t) = \delta.original$ then
19                 $num\_paths\_edge(f^*) + = \sigma(u,t)$;
20         end
21     end
22     $l^* \leftarrow$ last node in $AL(u,v,\delta)$;      /* end of $AL(u,v,\delta)$ */
23     $num\_paths4(l^*) \leftarrow 0$;
24     $f^* \leftarrow pred(l^*)$ in $AL(u,v,\delta)$;
       /* Traverse $AL(u,v,\delta)$ from last node(end) to first(start),
          $pred(l^*)$ calls the immediate predecessor $l^*$ in $AL(u,v,\delta)$. */
25     while $f^*$ is not $NULL$ do
26         $num\_paths4(f^*) =$
            $num\_paths4(l^*) + num\_paths3(l^*) + num\_paths\_edge(f^*)$;
27         $l^* \leftarrow f^*$;
28         $f^* \leftarrow pred(l^*)$;
29     end
30 end
```

The above algorithm takes $O(r)$ time for each bisector. It also takes total $O(r)$ time for traversing the shortest path tree T for every bisector. There can be $O(r \times \sqrt{r})$ bisectors for every boundary node of a piece. Therefore, for all possible bisectors of all boundary nodes in all pieces, it will take $O(\frac{n}{r} \times \sqrt{r} \times (r \times \sqrt{r}) \times r) = O(nr^2)$

▶ **Lemma 39.** *Given a segment $(p^*, \ldots, q^*)$ of bisector $\beta^*(b,.,.)$ (maintaining the same cyclic order as in AL), the sum of $num\_paths3(p^*_k)$ for all $p^*_k \in \{p^*, \ldots, q^*\}$ and $p^*_k \notin \{p^*, q^*\}$ added with the sum of $num\_paths\_edge(p^*_k)$ for all $p^*_k \in \{p^*_i, \ldots, p^*_j\}$ and $p^*_k \neq q^*$ can be computed in $O(1)$ time.*

Proof. The proof is similar to the proof of lemma 35.
Since each bisector is a simple cycle (from lemma 7), therefore given two points on the bisector we can get two possible segments (Figure 3.1a). We will use one of the following steps to get the desired value:

- If p^* appears before q^* in $AL(b,.,.)$ (Figure 3.1b):
  We need to look for the nodes(and the edges) that are after p^* and before q^* in $AL(b,.,.)$. We will return $num\_paths4(p^*) - [num\_paths4(q^*) + num\_paths3(q^*)]$.
- If p^* appears after q^* in $AL(b,.,.)$ (Figure 3.1c):
  Here we need to look for the nodes(and edges) that are before p^* and after q^* in $AL(b,.,.)$. Let s be the first node in $AL(b,.,.)$(i.e. the head of $AL(b,.,.)$). We will return $[num\_paths4(s) + num\_paths3(s)] - \{[num\_paths4(q^*) + num\_paths3(q^*)] - num\_paths4(p^*)\}$.

◀

Therefore we have the following theorem.

▶ **Theorem 40.** *We can build a data structure in $O(nr^2)$ for all pieces in G, such that it has the property stated in the lemma 39.*

3.4 Augmented Balanced Binary search tree (ABBST)

Here also, we maintain $ABBST(u,v,\delta)$ for each bisector $\beta^*(u,v,\delta)$ (for some pair of boundary nodes u,v and for each weight difference δ). For this we assume that along with the augmented cotree T^* (cotree of the shortest path tree T in P rooted at u), we are also given two linked lists: $\beta_{in}(u,v,\delta)$ and $\beta out(u,v,\delta)$, as mentioned in section 3.2 (Algorithm 1). $ABBST(u,v,\delta)$ is a binary search tree (AVL tree) where the nodes are same as in $\beta^*(u,v,\delta)$. The order of the nodes in $AL(u,v,\delta)$ are the relative keys of the nodes in $ABBST(u,v,\delta)$. In other words, the inorder traversal of $ABBST(u,v,\delta)$ starting from root, will give us the list $AL(u,v,\delta)$ starting from 'start' to 'end'.
We build the trees($ABBST(u,v,.)$) gradually, while iterating over δ, and augmenting the necessary values to the trees using the algorithm 7.
Another important point to note is that, $ABBST$ is a 'partially persistent' AVL tree. Partially persistent data structures are those, where the previous versions can be accessed, but only the current version can be modified. Fully persistent data structures are those where all the versions can be accessed and modified. For our purpose, this means that all versions of $ABBST(u,v,\delta)$ for all possible δ can be accessed, but only the one version (current) can be modified. Note that a data structure can be easily made partially persistent by simply keeping a copy of each version of the data structure. This will require us to copy the entire current version first, which will incur a cost of $O(n)$ (n is the size of the data structure) extra for each version. AVL tree can be converted into a partially persistent AVL tree using the copying method, but this would result in an increased total cost of building the data structure. There are other methods to achieve the same, like Fat node method, where the slowdown per operation is $O(logm)$ (m is the number of versions–in our case $m = O(r)$)i.e. total time per insertion, deletion and searching is $O(\log n \log m)$ (in our case this will be $O(\log^2 r)$). There is another method called node copying method where there is $O(1)$

slowdown per query and $O(1)$ amortized slowdown per update operation. Therefore, total amortized time per insertion, deletion and query is $O(\log n)$ ($O(\log r)$ in our case). Refer to [11, 36].

Algorithm 7: Building $ABBST(u,v,.)$

```
input  : boundary nodes $u,v \in \delta P$,$c\_val(u,v)$, $\beta_{in}(u,v,.)$, $\beta_{out}(u,v,.)$
output: $ABBST(u,v,.)$
1 $ABBST(u,v,+\infty) \leftarrow h*$;
2 $\delta' \leftarrow +\infty$;
3 for $\delta \in c\_val(u,v)$-/* Traverse $c\_val(u,v)$ in descending order.  */
4  do
5     $ABBST(u,v,\delta) \leftarrow ABBST(u,v,\delta')$; /* Persistent data structure
         implementation.  */
      /* $T^*$ is the augmented cotree */
6     $ABBST(u,v,\delta).del\_segment\_ABBST(T^*,\beta^*(u,v,\delta'))$;/* Algorithm 8 */
7     $ABBST(u,v,\delta).insert\_segment\_ABBST(T^*,\beta^*(u,v,\delta'))$;/* (This is
         different for $CC$,$HC$ and $NFC$) */
8     $\delta' \leftarrow \delta$;
9 end
```

Algorithm 8: $del\_segment\_ABBST()$ - Deleting a segment(many nodes) from $ABBST(u,v,\delta)$

```
input  : Augmented cotree $T^*$, $\beta_{out}(u,v,\delta')$
output: $ABBST(u,v,\delta)$ with some deleted nodes
1 for every dual node $f^*$ in $\beta_{out}(u,v,\delta')$ except the start and the end node) do
2     Delete $f^*$ from $ABBST(u,v,\delta)$;
3     Re-balance the tree using Rotations;
4 end
```

After deleting $\beta_{out}(u,v,\delta')$, either the tree will be empty or not: In case it is empty, that means $\beta_{u,v,\delta'}$ was a cycle and we build the entire $ABBST(u,v,\delta)$ from $\beta_{in}(u,v,\delta)$.

3.4.1 ABBST for CC

The $ABBST$ for CC will be augmented as follows:

- $num\_nodes(p^*)$—(same as in AL): This is the sum of $count(f^*)$ for all f^* in the subtree of p^* in T^*, added with the number of nodes on the face p, that are closer to u than to v (additively).
- $num\_nodes1(p^*)$: This is the sum of $num\_nodes(q^*)$ for all q^* which are in the right subtree of p^* in $ABBST(u,v,.)$, excluding $num\_nodes(p^*)$.
- $sum\_dist(p^*)$—(same as in AL): This is the sum of $dist\_dual(f^*)$ for all f^* in the subtree of p^* in T^*, added with the $Dist(u,t)$ of nodes t on the face p, that are closer to u than to v (additively).
- $sum\_dist1(p^*)$: This is the sum of $sum\_dist(q^*)$ for all q^* which are in the right subtree of p^* in $ABBST(u,v,.)$, excluding $sum\_dist(p^*)$.

Algorithm 9: *insert_segment_ABBST()* - Inserting a segment(many nodes) to $ABBST(u,v,\delta)$

```
input  : Augmented cotree $T^*$, $\beta_{in}(u,v,\delta)$
output: $ABBST(u,v,\delta)$ with some inserted nodes
1  $s \leftarrow \beta_{in}(u,v,\delta).start$;
2  $e \leftarrow \beta_{in}(u,v,\delta).end$;
3  for every dual node $f^*$ in $\beta_{in}(u,v,\delta)$ do
4      if $f^*$ is not the start or end node, i.e. not s or e then
5          Add $f^*$ to $ABBST(u,v,\delta)$ with $key(s) < key(f^*) < key(e)$;
6      $num\_nodes(f^*) \leftarrow 0$.
7      $sum\_dist(f^*) \leftarrow 0$.
8      for every vertex $t \in count\_list(f^*)$ do
9          if $dist(u,t) - dist(v,t) < \delta$ then
10             $num\_nodes(f^*) += 1$;
11             $sum\_dist(f^*) += Dist(u,t)$;
12     end
13     for every child $g^*$ of $f^*$ in $T^*$ do
14         if $g^*$ is not adjacent to $f^*$ in $AL(u,v,\delta)$ and all the vertices on the face $g$ is
             closer to $u$ than to $v$ then
15             $num\_nodes(f^*) += count\_sum(g^*)$
16             $sum\_dist(f^*) += dist\_dual\_sum(g^*)$
17     end
18     Re-balance the tree using Rotations and maintain the following two
         augmented values:
19     $num\_nodes1(f^*)$ is the sum of all the $num\_nodes()$ in the right subtree
         rooted at $f^*$ excluding $num\_nodes(f^*)$ itself. $sum\_dist1(f^*)$ is the sum
         of all the $sum\_dist()$ in the right subtree rooted at $f^*$ excluding
         $sum\_dist(f^*)$ itself.
       /* Lines 19 and 19 be achieved using normal augmented AVL
          insertion,deletion and rotation operations.  There will be
          $O(\log r)$ values that will be modified per insertion.  */
20     $s \leftarrow f^*$;
21 end
```

Since the above algorithm is basically augmented AVL tree(persistent), the total time required to construct $ABBST(u,v,.)$ for every pair of boundary nodes u,v in a piece is $O(r\log r)$. Note that all the $ABBST(u,v,\delta)$ for every possible δ can be constructed in $O(r\log r)$ time because from lemma 11, there can only be $O(r)$ edges and vertices that are added and deleted, and each insertion and deletion takes $O(\log r)$ time(as we need to traverse from a node to the root for updating the augmented values when a new node is added.). There are total $O(r)$ pairs of boundary nodes and $O(\frac{n}{r})$ pieces. Therefore constructing all the $ABBST$ will require $O(\frac{n}{r} \times r \times r\log r) = \widetilde{O}(nr)$ time.
Using the above algorithm, we can prove the following lemma.

▶ **Lemma 41.** *Given a segment $(p_i^*,\ldots,p_j^*)$ of bisector $\beta^*(b,.)$ (maintaining the cyclic order, as in AL or ABBST), the sum of $num\_nodes(p_k^*)$ for all $p_k^* \in \{p_i^*,\ldots,p_j^*\}, p_k^* \notin \{p_i^*,p_j^*\}$ and sum of $sum\_dist(p_k^*)$ for all $p_k^* \in \{p_i^*,\ldots,p_j^*\}, p_k^* \notin \{p_i^*,p_j^*\}$ can be computed in $O(\log r)$ time.*

Proof. It can be seen that the inorder traversal of the $ABBST$ starting from the root, gives us the bisector in the same order as the AL.

Let $num\_segment(p_k^*) = \sum_{p_w^*} num\_nodes(p_w^*)$, where p_w^* are all nodes that are inorder successors of p_k^* in $ABBST(u,v,.)$. Similarly we define $dist\_segment(p^*) = \sum_{p_w^*} sum\_dist(p_w^*)$. We can find $num\_segment(p_k^*)$ and $dist\_segment(p_k^*)$ for any p_k^* in $O(\log r)$ time using the following algorithm(10).
Again as before, given two points on the bisector we can get two possible segments (Figure 3.1a). We will use one of the following steps to get the desired value:

- If p^* appears before q^* in the inorder traversal of $ABBST$(or if $num\_segment(p^*) > num\_segment(q^*)$)(Figure 3.1b):
  We need to look for the nodes that are after p^* and before q^* in the inorder traversal of the $ABBST$. We will return $num\_segment(p^*) - [num\_segment(q^*) + num\_nodes(q^*)]$ and $dist\_segment(p^*) - [dist\_segment(q^*) + sum\_dist(q^*)]$.
- If p^* appears after q^* in the inorder traversal (Figure 3.1c):
  We need to look for the nodes that are before p^* and after q^* in the inorder traversal. Let s be the left most node in $ABBST(u,v,.)$. We will return $[num\_segment(s) + num\_nodes(s)] - \{[num\_segment(q^*) + num\_nodes(q^*)] - num\_segment(p^*)\}$ and $[dist\_segment(s) + sum\_dist(s)] - \{[dist\_segment(q^*) + sum\_dist(q^*)] - dist\_segment(p^*)\}$.

Algorithm 10: Finding $num\_segment(p_k^*)$ and $dist\_segment(p_k^*)$

```
input : $ABBST(u,v,.)$,a node $p_k^*$
output: $num\_segment(p_k^*)$ and $dist\_segment(p_k^*)$
1  $num\_segment(p_k^*) \leftarrow num\_nodes1(p_k^*)$;
   /* This will contain the sum of all $num\_nodes(p_i^*)$, where $p_i^*$ is in
      the right subtree of $p_k^*$ */
2  $dist\_segment(p_k^*) \leftarrow sum\_dist1(p_k^*)$;
   /* This will contain the sum of all $sum\_dist(p_i^*)$, where $p_i^*$ is in
      the right subtree of $p_k^*$ */
3  Let $root$ represent the root node of $ABBST(u,v,.)$;
   /* We need to traverse from $p_k^*$ to the root node of $ABBST$
      following the parent pointers.  We need to find the nodes on
      that path, that are the inorder successors of $p_k^*$ */
4  $p_w^* \leftarrow p_k^*$;
5  while $parent(p_w^*) \neq NULL$ do
6      if $p_w^*$ is the left child of $parent(p_w^*)$ then
7          $num\_segment(p_k^*) += \{num\_nodes1(parent(p_w^*)) + num\_nodes(parent(p_w^*))\}$;
8          $dist\_segment(p_k^*) += \{sum\_dist1(parent(p_w^*)) + sum\_dist(parent(p_w^*))\}$;
9      $p_w^* = parent(p_w^*)$;
10 end
```

Since the algorithm 10 requires to traverse a path in the $ABBST$, it will take $O(\log r)$ time to calculate $num\_segment()$ and $dist\_segment$ for each node.

◀

▶ **Theorem 42.** *We can build a data structure in $\widetilde{O}(nr)$ for all pieces in G, such that it has the property stated in the lemma 41.*

3.4.2 ABBST for HC

$ABBST$ for HC can be augmented as follows:

- $dist\_list3(p^*)$ (along with polynomials $P1_3(p^*)(x)$ and $P2_3(p^*)(x)$)—(same as in AL): This is the union of $dist\_list1(f^*)$ for all f^* in the subtree of p^* in T^*, including the $Dist(u,t)$ of nodes t on the face p, that are closer to u than to v (additively). Corresponding polynomials are constructed with the values in $dist\_list3(p^*)$ using algorithm 0.
- $dist\_list4(p^*)$ (along with polynomials $P1_4(p^*)(x)$ and $P2_4(p^*)(x)$): This is the union of $dist\_list3(q^*)$ for all q^* which are in the right subtree of p^* in $ABBST(u,v,.)$, excluding $dist\_list3(p^*)$. Again corresponding polynomials are constructed with the values in $dist\_list4(p^*)$ using algorithm 0.

Algorithm 11: *insert_segment_ABBST*() - Inserting a segment(many nodes) to $ABBST(u, v, \delta)$

```
input  : Augmented cotree $T^*$, $\beta_{in}(u, v, \delta)$
output: $ABBST(u, v, \delta)$ with some inserted nodes
$s \leftarrow \beta_{in}(u, v, \delta).start$;
$e \leftarrow \beta_{in}(u, v, \delta).end$;
for every dual node $f^*$ in $\beta_{in}(u, v, \delta)$ do
    if $f^*$ is not the start or end node, i.e. not $s$ or $e$ then
        Add $f^*$ to $ABBST(u, v, \delta)$ with $key(s) < key(f^*) < key(e)$;
    $dist\_list3(f^*) \leftarrow NULL$.
    for every vertex $t \in count\_list(f^*)$ do
        if $dist(u, t) - dist(v, t) < \delta$ then
            $dist\_list3(f^*).append(Dist(u, t))$;
    end
    for every child $g^*$ of $f^*$ in $T^*$ do
        if $g^*$ is not adjacent to $f^*$ in $AL(u, v, \delta)$ and all the vertices on the face $g$ is
          closer to $u$ than to $v$ then
            $dist\_list3(f^*).append(dist\_list2(g^*))$;
    end
    $P1_3(f^*), P2_3(f^*) \leftarrow$ Construct the Polynomials using 2.4.1 and the values
      as $dist\_list3(f^*)$;
    Re-balance the tree using Rotations and maintain the following
      augmented values: $dist\_list4(f^*)$ is the union of all the $dist\_list3()$ in
      the right subtree rooted at $f^*$ excluding $dist\_list3(f^*)$ itself.
    $P1_4(f^*), P2_4(f^*) \leftarrow$ Construct the polynomials using 2.4.1 with values as
      $dist\_list4(f^*)$;.
    /* Lines 16 and 17 can be achieved using normal augmented AVL
       insertion,deletion and rotation operations.  There will be
       $O(\log r)$ values($dist\_list4()$) that will be modified per
       insertion.  Also modify the polynomials of these $O(\log r)$
       nodes, or simply re-construct the polynomials of these
       nodes.  */
    $s \leftarrow f^*$;
end
```

$ABBST(u, v, \delta)$ for every possible δ can be constructed in $\widetilde{O}(r^2)$ time. From lemma 11, there can only be $O(r)$ edges and vertices that are added and deleted, and each insertion and deletion takes $\widetilde{O}(r)$ time (as we need to traverse from a node to the root for updating the augmented values when a new node is added and each updation takes $\widetilde{O}(r)$ time because new polynomials are constructed using 0 which will have $O(r)$ values as input). There are total $O(r)$ pairs of boundary nodes per piece and $O(\frac{n}{r})$ pieces. Therefore constructing all the $ABBST$ will require $\widetilde{O}(\frac{n}{r} \times r \times r^2) = \widetilde{O}(nr^2)$ time.
Using the above algorithm we can prove the following lemma.

▶ **Lemma 43.** *Given a segment $\gamma = \{p^*, \ldots, q^*\}$ of bisector $\beta^*(b, ., .)$ (maintaining the same cyclic order as in AL) and the value of polynomials $P1_3(f^*)(x), P2_3(f^*)(x)$, $P1_4(f^*)(x), P2_4(f^*)(x)$ for all $f^* \in \gamma$ and for some x, value of $\sum_{f^* \in \gamma \setminus \{p^*, q^*\}} \frac{P1_3(f^*)(x)}{P2_3(f^*)(x)}$, can be calculated in $\widetilde{O}(1)$ time.*

Proof. It can be seen that the inorder traversal of the $ABBST$ starting from the root, gives us the bisector in the same order as in AL.

Let $\text{HC}_{segment}(p_k^*)(x) = \sum_{p_w^*} \frac{P1_3(p_w^*)(x)}{P2_3(p_w^*)(x)}$, where p_w^* are all nodes that are inorder successors of p_k^* in $ABBST(b,.,.)$. We can find $\text{HC}_{segment}(p_k^*)(x)$ for some x in $O(\log r)$ time using the following algorithm(12).
Again as before, given two points on the bisector we can get two possible segments (Figure 3.1a). We will use one of the following steps to get the desired value (we will also store the list of these polynomials and the corresponding $sign$ in a list using algorithm 13):

- If p^* appears before q^* in the inorder traversal of $ABBST$
(or if $\text{HC}_{segment}(p^*)(x) > \text{HC}_{segment}(q^*)(x)$) (Figure 3.1b):
We need to look for the nodes that are after p^* and before q^* in the inorder traversal of the $ABBST$. We will return
$\text{HC}_{segment}(p^*)(x) - \left[\text{HC}_{segment}(q^*)(x) + \frac{P1_3(q^*)(x)}{P2_3(q^*)(x)}\right].$

- If p^* appears after q^* in the inorder traversal (Figure 3.1c):
We need to look for the nodes that are before p^* and after q^* in the inorder traversal. Let s be the left most node in $ABBST(b,.,.)$. We will return
$\left[\text{HC}_{segment}(s)(x) + \frac{P1_3(s)(x)}{P2_3(s)(x)}\right] - \left(\left[\text{HC}_{segment}(q^*)(x) + \frac{P1_3(q^*)(x)}{P2_3(p^*)(x)}\right] - \text{HC}_{segment}(p^*)(x)\right).$

Algorithm 12: Finding $\text{HC}_{segment}(p_k^*)(x)$ and fetching the polynomials associated with it

```
input : $ABBST(b,.,.)$,a node $p_k^*$ a value $x$ and a sign $sign \in \{+,-\}$
output: Value of $\text{HC}_{segment}(p_k^*)(x)$ and a list of polynomials with their signs
1  $\text{HC}_{segment}(p_k^*)(x) \leftarrow \frac{P1_4(p_k^*)(x)}{P2_4(p_k^*)(x)}$; /* This now contains the sum of all
      $\frac{P1_3(p_i^*)(x)}{P2_3(p_i^*)(x)}$, where $p_i^*$ is in the right subtree of $p_k^*$ excluding $p_k^*$.
      */
2  $Poly_{segment}[] \leftarrow Null$;
3  $Poly_{segment}[].insert((P1_4(p_k^*)(x)),(P2_4(p_k^*)(x)),sign)$; /* Similarly this
      now contains the list of all $P1_3(p_i^*)(x)$ and $P2_3(p_i^*)(x)$, where
      $p_i^*$ is in the right subtree of $p_k^*$ excluding $p_k^*$.  */
   /* We need to traverse from $p_k^*$ to the root node of $ABBST(b,.,.)$
      following the parent pointers.  We need to find the nodes on
      that path, that are the inorder successors of $p_k^*$.  */
4  $p_w^* \leftarrow p_k^*$;
5  while $parent(p_w^*) \neq NULL$ do
6     if $p_w^*$ is the left child of $parent(p_w^*)$ then
7        $\text{HC}_{segment}(p_k^*)(x) += \{\frac{P1_4(parent(p_w^*))(x)}{P2_4(parent(p_w^*))(x)} + \frac{P1_3(parent(p_w^*))(x)}{P2_3(parent(p_w^*))(x)}\}$;
8        $Poly_{segment}[].insert((P1_4(parent(p_w^*))(x)),(P2_4(parent(p_w^*))(x)),sign)$;
9        $Poly_{segment}[].insert((P1_3(parent(p_w^*))(x)),(P2_3(parent(p_w^*))(x)),sign)$;
10    $p_w^* = parent(p_w^*)$;
11 end
12 return $\text{HC}_{segment}(p_k^*)(x)$ and $Poly_{segment}[]$;
```

Since the algorithm 12 requires to traverse a path in the $ABBST$, it will take $O(\log r)$ time to calculate $\mathrm{HC}_{segment}()$ for each node.

◀

Also, same as AL, we use the following algorithm(13) to fetch the list of required polynomials for a given segment of bisector:

Algorithm 13: Fetching list of polynomials needed for Lemma 43

```
input : $ABBST(u,v,\delta)$, segment of bisector $\gamma = \{p^*,\dots,q^*\}$
output: List of polynomials and their corresponding signs
1  $Polynomials[] \leftarrow Null$; /* Each entry of $Polynomial[]$ contain two
       polynomials and a sign (+ or -).  */
2  if If $p^*$ appears before $q^*$ in the inorder traversal of $ABBST(u,v,\delta)$ then
       /* All the polynomials are from $ABBST(u,v,\delta)$.  */
3      $Polynomials[].insert(Poly_{segment(p^*)}$ with sign $= +)$;
       /* This will insert the polynomials ($O(\log r)$ polynomials) in
          $Poly_{segment}(p^*)$(from algorithm 12) into $Polynomials[]$.  */
4      $Polynomials[].insert(Poly_{segment(q^*)}$ with sign $= -)$;
5      $Polynomials[].insert((P1_3(q^*)(x)),(P2_3(q^*)(x)),-)$;
6  if $p^*$ appears after $q^*$ in the inorder traversal of $ABBST(u,v,\delta)$ then
       /* All the polynomials are from $AL(u,v,\delta)$.  */
7      $s \leftarrow$ root node in $ABBST(u,v,\delta)$;
8      $Polynomials[].insert(Poly_{segment(s^*)}$ with sign $= +)$;
9      $Polynomials[].insert((P1_3(s^*)(x)),(P2_3(s^*)(x)),+)$;
10     $Polynomials[].insert(Poly_{segment(q^*)}$ with sign $= -)$;
11     $Polynomials[].insert((P1_3(q^*)(x)),(P2_3(q^*)(x)),-)$;
12     $Polynomials[].insert(Poly_{segment(p^*)}$ with sign $= +)$;
13 return $Polynomials[]$;
```

Since, the polynomials in algorithm 13 are exactly same as described in the proof of lemma 43, it will require $\widetilde{O}(1)$ time per segment of bisector.

▶ **Theorem 44.** *We can build a data structure in $\widetilde{O}(nr^2)$ for all pieces in G, such that it has the property stated in the lemma 43.*

Chapter 4

Upper Bounds

In this chapter, we first present a linear upper bound for calculating BHC of a single node in sparse graphs (in section 4.1) using the fast multipoint evaluation technique for evaluating a $d-$degree polynomial on n points in $O(\max n, d \log^2 d)$ time (section 2.4).

In section 4.2, we present the $o(n^2)$ algorithms for computing CC,HC and NFC of all nodes in planar graphs. We also present the $o(n^2)$ algorithm to compute $BNFC$ of a single node in a planar graph.

4.1 Linear Upper Bound for BHC

In this section, we show that BHC of any one node of a sparse graph can be computed in linear time, up to logarithmic factors. Since planar graphs are sparse in nature, the same $\widetilde{O}(n)$ running time holds for planar graphs as well.

▶ **Theorem 45.** *For a sparse graph with n nodes and $O(n)$ edges, the BHC of single node can be computed in $\widetilde{O}(n)$ time.*

Proof. For computing BHC(s), we first compute $a_i = dist(s, t_i)$ for all other nodes t_i by using Dijkstra's algorithm. Since the graph is sparse, this takes $\widetilde{O}(n)$ time. We will use claim 46 to prove the theorem. ◀

▷ Claim 46. Given n numbers $\{a_1, a_2, \dots a_n\}$, $B = \sum_i \sum_j \frac{1}{a_i + a_j}$ can be computed in $\widetilde{O}(n)$.

Proof. Let $A = \{a_1, \dots a_n\}$. Define two polynomials $P1(x) = \sum_i \prod_{j \neq i}(a_j + x)$ and $P2(x) = \prod_i (a_i + x)$ using those points. Observe that $B = \sum_i \frac{P1(a_i)}{P2(a_i)}$. Therefore, B can be computed by computing $\{P1(x) : x \in A\}$ and $\{P2(x) : x \in A\}$ — which can be accomplished using multi-point evaluation of $P1(x)$ and $P2(x)$ on points in A.

However, multi-point evaluation requires coefficients of a polynomial, whereas in our case, $P1(x)$ and $P2(x)$ are not readily available in the coefficient form. To resolve this, we design an FFT-based recursive algorithm to obtain the coefficients of $P1(x)$ and $P2(x)$ that is explained in Section 2.4.1 and has a time complexity of $O(n \log^2 n)$.

Once we have the coefficients of $P1(x)$ and $P2(x)$ we can evaluate them on all points in A using multipoint evaluation in time $O(n \log^2 n)$. B can be computed by simply summing $P1(a_i)/P2(a_i)$. The total complexity of computing B remains $\widetilde{O}(n)$. ◀

The claim proves the theorem as well since BHC$(s) = \sum_i \sum_j \frac{1}{a_i + a_j}$.

4.2 Subquadratic Upper Bounds for Planar Graphs

In this section, we design strictly subquadratic algorithms for computing *CC, HC and NFC of all nodes* in a planar graph. All these algorithms have a similar structure that is composed of 4 stages. Observe that computing the above centrality values at all nodes requires looking at the shortest paths between every pair of vertices. These stages group these paths based on a decomposition of the graph in a manner that allows us to avoid explicitly looking at all $O(n^2)$ shortest path.

Pre-processing: Perform an r-division of G. r can be chosen depending on the overall complexity.

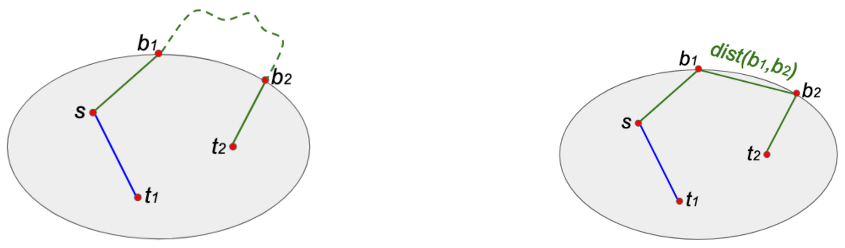
Stage 1: Compute centrality values of all boundary nodes. This step considers the shortest paths with boundary nodes as source vertices.

Stage 2: Compute partial centrality values of all internal nodes by considering each piece separately and singly; essentially, here we consider the shortest paths between two internal nodes of every piece. To separate a piece p in G, we construct a graph G_P that has all the vertices of P. Edges of G_P are those that were inside P only and we also add additional edges as and when needed using the values calculated in stage 1.

Stage 3: Now, we need to consider the shortest paths between two internal nodes but from different pieces. Let s be an internal node and P be a piece that *does not* contain s. To consider the shortest paths from s to those internal to P, we observe that these paths *must* pass through some boundary node of P. This stage can be further sub-divided into sub-stages:

1. Firstly, we triangulate each face of each piece except for the holes. Then we perturb the edge weights such that the distance between each pair of vertices is unique. This implies that there is a unique shortest path between two nodes. We use deterministic lexicographic perturbation for this purpose (See section 2.2).
2. We then consider the shortest path tree T_b from each boundary node b to all the nodes of P (already constructed in stage 2) and construct the augmented cotree T_b^*. We compute and store some values in T_b^*.
3. We then consider the bisectors $\beta^*(u, v, \delta)$ for each pair of boundary nodes u and v in P and each critical value δ. These are stored as either $AL(u, v, \delta)$ or $ABBST(u, v, \delta)$ (details in Chapter 3). We augment some values to the AL and $ABBST$ constructed during the pre-processing stage of AWV construction. Appropriate values are fetched from the bisector segments to calculate the centralities efficiently.

Stage 4: Finally, we update the centrality value of all internal nodes s by considering paths starting from them, passing through some boundary vertex and ending at an internal node of a different piece P. For this, a weighted Voronoi diagram is generated for every piece using its boundary nodes as sites, and the weights on the sites are the distances from s to the respective sites. For each boundary node b, the AWV construction gives us the Voronoi region of b as segments of bisectors. We then use the pre-calculated values in stage 3, and compute the centrality by carefully aggregating them. This allows us to only consider paths ending at boundary nodes leading to the subquadratic running time.



(a) Path from s to t_1 and t_2 in G **(b)** Path from s to t_1 and t_2 in G_P

Figure 4.1: Computing ω_{int} by creating G_P

We describe stages 1,2, 3, and 4 in detail for each of the centralities. We additionally use the fact that the number of edges in a planar graph is $O(n)$ and, therefore, running Dijkstra from a single vertex takes $\tilde{O}(n)$ steps.

4.2.1 CC of all nodes

The main result of this section is the following theorem.

▶ **Theorem 47.** *For a planar graph with n nodes and $O(n)$ edges, the CC of all nodes can be computed in $\tilde{O}(n^{5/3})$ time.*

Instead of *CC*, we actually show how to compute its reciprocal, that we refer to as partial Weiner index (PWI). Denoted $\omega(s)$, PWI of a vertex s is defined as the sum of shortest path distances from s to all other nodes in the graph: $\omega(s) = \sum_{t\in V\setminus\{s\}} Dist(s,t)$. $\omega(s)$ can be computed by simply running Dijkstra from s.

Stages 1 and 2

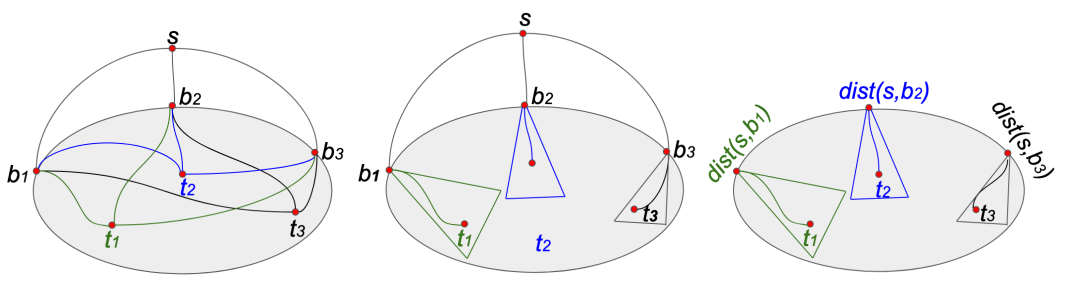
For stage 1, we need to compute $\omega(b)$ for all boundary nodes, and this is done by simply running Dijkstra from each boundary of every piece. Since boundary vertices are part of multiple pieces, a quick check can ensure that each boundary vertex is chosen as b only once.

For any internal node s whose piece we denote by $P(s)$ and we denote the set of all boundary nodes by B , we decompose $\omega(s)$ into three components :

$$\begin{aligned}\omega(s) &= \sum_{t\in B} Dist(s,t) + \sum_{t\in P(s)\setminus B} Dist(s,t) + \sum_{t\notin P(s) \text{ and } t\notin B} Dist(s,t)\\ &= \omega_B(s) + \omega_{int}(s) + \omega_{ext}(s) \qquad (4.1)\end{aligned}$$

Since in stage 1, we have already calculated $Dist(s,b)$ for all boundary nodes b, we can substitute the values in equation 4.1 to get $\omega_B(s)$.

In stage 2 we compute $\omega_{int}(s)$ by taking $G_P = (V_P, E_P)$ and running Dijkstra on G_P from all internal nodes of P. The extra edges in G_P handle those paths that goes outside P. But since the source and target nodes are both inside P, such a path must leave and reenter the piece via some boundary node (illustrated in Figure 4.1). We capture this by adding an edge in G_P between every pair of boundary nodes inside a piece.



(a) Path from s to internal nodes of P

(b) Using Voronoi cells to group such paths

(c) Using AWV regions to compute $\omega_{ext}(s)$

Figure 4.2: Computing $\omega_{ext}(s)$ by grouping internal nodes and their closest boundary nodes

Stage 3

The only value left to compute is $\omega_{ext}(s)$ for some internal node s; however, these involve shortest paths that partly lies inside $P(s)$ and partly lies outside $P(s)$.

The distance from s to any $t \notin P(s)$ can be broken down to $dist(t, b)$ for some boundary $b \in P(t)$ (there can be more than one boundary) and $dist(b, s)$ (see Figure 4.2a). Observe that we already have computed both these distances in Stage 1; however, naively trying all $b \in \partial P(t)$ is not sufficient since it leads to a complexity of $O(n^2\sqrt{r})$ — there are $O(n)$ possible s, $O(r)$ many t in any piece, $O(\sqrt{r})$ many b for any piece and $O(\frac{n}{r})$ pieces.

Our strategy is to use Voronoi regions to pre-compute the closest boundary node for every internal node in $P(t)$ — this depends on s as well (see Figure 4.2b). Let P denote the piece of t and let b be a boundary node of P. We define $J(s, b)$ as the set of internal nodes of P whose shortest path to s pass through b. For example, $J(s, b_1)$ contains t_1 in Figure 4.2a. Assuming that $J(s, b_i)$ are disjoint across all b_i's, we can group the internal nodes inside P based on their "closest" boundary.

$$\begin{aligned}\omega_{ext}(s) = \sum_{t \notin P(s)} dist(s,t) &= \sum_{\text{piece } P \neq P(s)} \sum_{b \in \partial P} \sum_{t \in J(s,b)} [dist(s,b) + dist(b,t)] \\ &= \sum_{\text{piece } P} \sum_{b \in \partial P} \left(dist(s,b) \times |J(s,b)| + \sum_{t \in J(s,b)} dist(b,t) \right)\end{aligned} \tag{4.2}$$

The crux of the final stage is a technique to efficiently compute $J(s, b)$ and the bracketed expression in Equation 4.2. Consider an AWV decomposition of P using ∂P as sites and $dist(s, b)$ as the weight of the site b (see Figures 4.2b and 4.2c).

In this stage, we construct the AL or $ABBST$ data structures for all pieces. Details are given in section 3.3.1 and section 3.4.1. We also do the preprocessing needed for constructing the AWV diagram ([18]).

Stage 4

Now in this stage, we actually compute the $\omega_{ext}(s)$ using the values computed in stage 3. The notations are as usual. We will describe the algorithm for a single boundary node b, and repeating this for all the boundary nodes will give us the end result.

1. For each piece P and each internal node s not in P, we compute the AWV diagram with respect to the distance from s to all boundary nodes in P. From this, we get the relevant segments of bisectors. From each segment, we already know the adjacent boundary vertex (or adjacent cell). Suppose, we have a segment of a bisector with the adjacent cell as b'. Now, for finding the right δ, we run binary search on $c\_val(b, b')$ to find the largest value that is $\leq w(b') - w(b)$. Now, we extract the information from the data structure $ABBST(b, b', \delta)$ (or $AL(b, b', \delta)$). Since, δ can be different from the actual weight difference, we need to prove lemma 48

2. $|J(s, b)|$ can be computed by adding num_nodes values of each segment, as described in lemma 35 and lemma 41. Let f^* be the node that is shared between two consecutive segments of bisector on a cycle $C^* \in \mathcal{C}^*$ and $f^* \neq h^*$, where h^* is the dual node of hole h which is the hole of b. Therefore by lemma 25, f^* is a voronoi vertex of degree 3 and the edge f^*g^* which is not on the cycle C^* is not in $Vor(b)$, i.e the vertices of face g are not in $Vor(b)$. So there are no edges from f^* that are entering into $Vor(b)$. We only need to add 1 to $|J(s, b)|$ because only 1 vertex of face f is inside $Vor(b)$.

3. And ultimately, for each of the cycles in $\mathcal{C}^* \setminus \{C_0^*\}$, we figure out the unique edge f^*g^* (lemma 19). We subtract $num\_nodes(f^*)$ from $|J(s, b)|$. If there exists any other child of f^* in T^* (let's say $t^* \neq g^*$), then we add $num\_nodes(t^*)$ to $|J(s, b)|$. Since, there are $O(1)$ holes in piece P, there can be $O(1)$ such cycles in piece P (except the C_0^* of each boundary node), therefore, this step requires $O(1)$ time per piece.

4. For calculating $SUM = \sum_{t \in J(s,b)} dist(b, t)$, we do the same thing as done for $|J(s, b)|$. We add sum_dist values of each segment to SUM, as described in lemma 35 and lemma 41. We also check which vertex of f is inside $Vor(b)$(say vertex t), we add $Dist(b, t)$ to SUM.

5. Finally, for each of the cycles in $\mathcal{C}^* \setminus \{C_0^*\}$, we figure out the unique edge f^*g^*. (lemma 19). We subtract $sum\_dist(f^*)$ from SUM. If there exists any other child of f^* in T^* (let's say $t^* \neq g^*$), then we add $sum\_dist(t^*)$ to SUM. Again this step will require $O(1)$ time per piece.

After we calculate both $|J(s, b)|$ and SUM, we just substitute the values in equation 4.2 to get the value of $\omega_{ext}(s)$.

▶ **Lemma 48.** *For any node t, if t is in $Vor(b)$, then $dist(b, t) + w(b)$ is the least additive distance from any boundary node to t.*

Proof. Suppose t is associated with segment $\beta^*(b, b', \delta)$. Since, t is in $Vor(b)$, therefore, $dist(b, t) - dist(b', t) < \delta \leq w(b) - w(b') \implies dist(b, t) + w(b) < dist(b', t) + w(b')$. ◀

▶ **Lemma 49.** *$|J(s, b)|$ and SUM are calculated correctly for all boundary nodes b. In other words, for each vertex v , v is counted in $|J(s, b)|$ exactly once (and $Dist(b, v)$ is added to SUM exactly once) iff v is in the voronoi cell of b.*

Proof. We will prove that $|J(s, b)|$ is computed correctly. The correctness of SUM will follow. Suppose v is associated with face f and dual node of f is f^*. Let v be associated with the segment of bisector $\beta^*(b, b', \delta)$.

- ($\Rightarrow$) If v is not in $Vor(v)$, then v will not be counted in $|J(s,b)|$:
  f^* can either lie on the cycle C_0^* or face f can be entirely outside $Vor(b)$. In both the cases, v is not counted in $|J(s,b)|$. If f^* lies on C_0^*, then from the lemma 35, each node t on the face f is checked for the following condition: $dist(b,v) - dist(b',v) < \delta$. Since the condition is false in case of v (v not in $Vor(b)$), therefore v is not counted. Now, if face f is entirely outside $Vor(b)$, then the root to f^* path of T^* does not contain any dual node which is on any cycle, therefore, no node of f is counted.

- ($\Leftarrow$) If v is in $Vor(v)$, then v will be counted in $|J(s,b)|$ exactly once:
  To prove this, we use lemma 22.

Case 1: If the root to f^* path of T^* has only one boundary node, then f^* has to lie on C_0^*. As discussed above, if f^* lies on C_0^*, then after checking the condition, v will be counted. Since, the root to f^* path in T^* has only one boundary node and a vertex is associated with a single face (and hence a single dual node in T^*), therefore, v won't be counted again.

Case 2: If the root to f^* path of T^* has x penetrating edges and $x-1$ exiting edges, it means that the count of vertices associated with face f ($count(f^*)$) is excluded (or subtracted) $x-1$ times from $|J(s,b)|$. But since there are x penetrating edges, therefore, $count(f^*)$ is added x times which proves that $count(f^*)$ is ultimately added only once.

Case 3: If the root to f^* path of T^* has x penetrating edges and x exiting edges, it means f^* is on some cycle C^*. Suppose (p^*, q^*) is the exiting edge of cycle C^* and it lies on the path from root to f^*. If we consider the subpath excluding the path $p^* \rightsquigarrow f^*$, then it has x penetrating edges and $x-1$ exiting edges. Therefore, by the previous argument, we can say that for that subpath, $count(f^*)$ is added once. Now, we subtract $count(f^*)$ once for the exiting edge (p^*, q^*), but then we also add $count(f^*)$ once for the node f^* itself. Therefore, in total $count(f^*)$ is added exactly once.

◀

From lemma 48 and lemma 49, we can conclude that the algorithm for computing CC of all nodes is correct.

Running time analysis

We finish the proof of Theorem 47 by discussing it's subquadratic complexity. The running times of the different stages are explained below.

Pre-processing: r-division takes $O(n \log n)$ time using the naive recursive approach by Frederickson[14]. We can also use a recently proposed linear time algorithm [30].

Stage 1: Computing $\omega(s)$ of the boundary nodes can be found by running $Dijkstra's$ algorithm on G taking each boundary as a source. The number of boundary nodes in G is $O(\frac{n}{\sqrt{r}})$ leading to a running time of $O(\frac{n}{\sqrt{r}} \times n \log n)$.

Stage 2: Computing $\omega_{int}(s)$ also involves running $Dijkstra$ on the graph G_P with $O(r)$ nodes and $O(r)$ edges (in G_P we add $O(r)$ edges between every pair of boundary nodes). The running time of this stage is $O(\frac{n}{r} \times r \log r)$.

Stage 3: The cost for AWV pre-processing is $\widetilde{O}(nr)$ (theorem 4). The cost of building the augmented cotree is $\widetilde{O}(n\sqrt{r})$ (theorem 31) and the cost of building $ABBST$ is $\widetilde{O}(nr)$—from theorem 42 ($O(nr^2)$ if we use AL—from theorem 36). Therefore, the total running time of this stage is $\widetilde{O}(\frac{n}{r} \times r^2)$ ($O(nr^2$ if AL is used).

Stage 4: In this stage, we construct AWVs for all internal nodes s and all pieces different from $P(s)$. The construction takes $\widetilde{O}(\sqrt{r})$ time (Theorem 4). That is followed by a processing of all boundary nodes in every piece. For every boundary node, the information from all the segments are aggregated. Fetching the information fom each segment require $\widetilde{O}(1)$ time using $ABBST$—lemma 41 ($O(1)$ time if AL is used—lemma 35). Since, each segment can be part of two voronoi cells and total number of segments per piece is $O(\sqrt{r})$, therefore, fetching of information for all the boundary node will require $\widetilde{O}(\sqrt{r})$ time. Repeating this for each outside node s will result in a running time of $\widetilde{O}(n\sqrt{r})$ per piece. Total number of pieces are $O(\frac{n}{r})$, therefore the total running time of this stage is $\widetilde{O}(1) \times O(n \times \frac{n}{r} \times \sqrt{r}) = \widetilde{O}(\frac{n^2}{\sqrt{r}})$.

Since we use $r = n^{\frac{2}{3}}$, we get the total running time for computing CC of all nodes as $\widetilde{O}(n^{\frac{5}{3}})$ (using AL we will get $\widetilde{O}(n^{\frac{9}{5}})$ using $r = n^{\frac{2}{5}}$).

4.2.2 HC of all nodes

Adaptation of the algorithm for computing CC gives us a subquadratic algorithm for computing HC of all nodes.

▶ **Theorem 50.** *For a planar graph with n nodes and $O(n)$ edges, the HC of all nodes can be computed in $\widetilde{O}(n^{9/5})$ time.*

Stage 1 and 2

Stage 1 is similar to CC. We compute $HC(b)$ for all boundary nodes b, which can be done by running Dijkstra from each boundary of every piece. Since boundary vertices are part of multiple pieces, a quick check can ensure that each boundary vertex is chosen as b only once.
As in CC, we can also write the equation of HC for an internal node s in piece $P(s)$ as follows:

$$\begin{aligned} HC(s) =& \sum_{t \in B} \left(\frac{1}{Dist(s,t)} \right) + \sum_{t \in P(s) \setminus B} \left(\frac{1}{Dist(s,t)} \right) + \sum_{t \notin P(s) \text{ and } t \notin B} \left(\frac{1}{Dist(s,t)} \right) \\ =& HC_B(s) + HC_{int}(s) + HC_{ext}(s) \end{aligned} \tag{4.3}$$

Stage 2 is also similar to that for computing CC (see Section 4.2.1) in which HC_B and HC_{int} are calculated for all internal nodes s in all pieces. We give an overview of an algorithm to compute $HC_{ext}(s)$ in stages 3 and 4.

The challenge of considering all internal nodes and all boundary-nodes of P one-at-a-time, as that in Section 4.2.1, remains but a bigger problem arises due to the aggregation of $1/Dist(\cdot, t)$ instead of $Dist(\cdot, s)$ as in CC. Using b to denote the boundary node through which the shortest path from s to t passes, it was possible in CC to add $Dist(s,b)$ to $Dist(b,t)$ for all t whose shortest path from s passes through b. However, here we need to compute for all such t the expression $\frac{1}{Dist(s,b)+Dist(b,t)}$. The clever trick employed here is to use multipoint evaluation of polynomials (section

2.4), using $Dist(s,b)$ as the variable, to simultaneously compute many such expressions.

Stage 3

As in CC, HC_{ext} can be written as:

$$\begin{aligned} HC_{ext}(s) = \sum_{t \notin P(s) \text{ and } t \notin B} Dist(s,t) &= \sum_{\text{piece } P \neq P(s)} \sum_{b \in \partial P} \sum_{t \in J(s,b)} \left[\frac{1}{dist(s,b) + dist(b,t)} \right] \\ &= \sum_{\text{piece } P} \sum_{b \in \partial P} \left(\frac{P1_{J(s,b)}(Dist(s,b))}{P2_{J(s,b)}(Dist(s,b))} \right) \end{aligned} \tag{4.4}$$

where $P1_{J(s,b)}(x) = \sum_{t_i \in J(s,b)} \prod_{t_j \in J(s,b) \setminus \{t_i\}} (x + Dist(b, t_j))$
and $P2_{J(s,b)}(x) = \prod_{t_i \in J(s,b)} (x + Dist(b, t_i))$.

For efficiently computing $P2_{J(s,b)}(x)$ and $P2_{J(s,b)}(x)$, we use the data structure described in section 3.4.2 (or section 3.3.2). In this stage, we build the necessary data structure and also do the pre-processing needed for the AWV construction.

Stage 4

In this stage, we figure out the polynomials $P1_{J(s,b)}(x)$ and $P2_{J(s,b)}(x)$ for each boundary node b. We cleverly, aggregate many such polynomials to get the exact value of $P1_{J(s,b)}(x)$ and $P2_{J(s,b)}(x)$ for every x. We then use multipoint evaluation on these polynomials. The details of stage 4 is given in Algorithm 14. In this algorithm, we build a hash table $HT(P)$ for each piece P. The keys of the hash table are $< integer\ id, polynomial\ P1, polynomial\ P2 >$, where id is the corresponding id associated with the polynomial $P1$. The value of HT is of the form $< node\ s, number\ n, sign >$, where s is some node in G, n is some number and $sign$ is either + or -. Since id is associated with a polynomial, and a polynomial is associated with a dual vertex on a bisector, therefore total number of $id's$ (and hence total number of polynomials) is $\#bisectors\ per\ piece \times \#dual\ vertices\ on\ a\ bisector = O((\sqrt{r})^2 \times r \times r) = O(r^3)$.

Algorithm 14: Building the Hash table

```
input : all $ABBST/AL$ of all pieces, graph $G$ with the pieces and the
        augmented cotrees of all boundary nodes of all pieces
output: $HT(P)$ for all $P$
1  for each internal node $s$ do
2      for each Piece $P \neq P(s)$ do
3          Construct the AWV diagram of $P$ wrt the $dist(s,b)$ for all boundary
             nodes $b \in \partial P$;
4          for a bisector segment $\gamma = p^*, \ldots, q^*$ of some bisector $\beta^*(u,v)$ do
               /* We first find the appropriate $\delta$, which is a value
                  from $c\_val(u,v)$.  */
5              $\delta \leftarrow$ Binary search on $c\_val(u,v)$ to find the largest value which is
                 $\leq w(v) - w(u)$;
6              $Polynomials[] \leftarrow$ Fetch the polynomials using algorithm 13 (or
                 algorithm 5);
7              for each entry in $Polynomials[]$ do
8                  Let the entry be $P1(x), P2(x), sign$;
9                  Let $id \leftarrow P1(x).id$;
10                 $HT(P)[(id, P1(x), P2(x))].insert((s, Dist(s,b), sign))$;
11             end
12         end
           /* Now we need to handle the case when there are multiple
              holes.  */
13         for each cycle $C^* \in \mathcal{C}^* \setminus \{C_0^*\}$ do
14             Let $f^*, g^*$ be the unique edge where $f^*$ is parent of $g^*$ in $T^*$ and $g^*$
                 is on the cycle $C^*$ (lemma 19);
15             Let $id \leftarrow P1_2(f^*)(x).id$;
16             $HT(P)[(id, P1_2(f^*)(x), P2_2(f^*)(x))].insert((s, Dist(s,b), -))$;
17             if $\exists t^* \neq g^*$ child of $f^*$ in $T^*$ then
18                 Let $id \leftarrow P1_2(t^*)(x).id$;
19                 $HT(P)[(id, P1_2(t^*)(x), P2_2(t^*)(x))].insert((s, Dist(s,b), +))$;
20         end
21     end
22 end
```

▶ **Lemma 51.** *Building $HT(P)$ for all P requires $\widetilde{O}(\frac{n^2}{\sqrt{r}})$ time using algorithm 14.*

Proof. For each segment of bisector, fetching $Polynomials[]$ takes $\widetilde{O}(1)$ time using the $ABBST$ data structure, from lemma 43 (or $O(1)$ time using AL, from lemma 37). There can be $O(\sqrt{r})$ segments of bisector, and each segment is used twice (for two adjacent sites). Therefore, fetching all the $Polynomials[]$ for all boundary node of a piece will take $\widetilde{O}(\sqrt{r})$ time. Also, size of each $Polynomial[]$ is $\widetilde{O}(1)$ time, therefore, storing the polynomials from the list to $HT(P)$ require $\widetilde{O}(1)$ time (it will take $O(1)$ time if AL is used). To handle the case of multiple holes, it will require additional $O(1)$ time per piece, because there can be $O(1)$ holes (hence $O(1)$ cycles) in P. Therefore building $HT(P)$ for all s will take $\widetilde{O}(n \times \sqrt{r})$ time. Building all the hash tables for all pieces will require $\widetilde{O}(\frac{n}{r} \times n\sqrt{r}) = \widetilde{O}(\frac{n^2}{\sqrt{r}})$ time. ◀

Now, we use the algorithm 15 to evaluate all the stored polynomials and calculating HC_{ext} of all nodes.

Algorithm 15: Evaluating all the polynomials in $HT(.)$ and calculating the HC of all nodes

```
input  : $HT(P)$ for all $P$
output: $HC_{ext}(.)$ of all nodes
/* Evaluate all the polynomials stored in $HT$. */
for each piece $P$ do
    /* Traverse $HT(P)$; */
    for every non-empty entry in $HT(P)$ do
        Let the entry be $HT[(id, P1(x), P2(x))]$;   /* This contains list of triplets. */
        $Numbers[] \leftarrow Null$;
        for each triplet in $HT[(id, P1(x), P2(x))]$ do
            Let the triplet be $(s, Dist(s, b), sign)$;
            $Numbers[].insert(Dist(s, b))$;
        end
        Evaluate $P1$ and $P2$ on the values in the list $Numbers[]$ using Multipoint evaluation (section 2.4);
        /* Now we replace the second entry of each triplet in $HT[(id, P1(x), P2(x))]$ with the evaluated value $\frac{P1(x)}{P2(x)}$. */
    end
    /* Again traverse $HT(P)$; */
    for every non-empty entry in $HT(P)$ do
        Let the entry be $HT[(id, P1(x), P2(x))]$;   /* This contains list of triplets. */
        for each triplet in $HT[(id, P1(x), P2(x))]$ do
            Let the triplet be $(s, val, sign)$;
            $HC_{ext}(s) += (sign)val$;
        end
    end
end
```

▶ **Lemma 52.** *Evaluating all the polynomials on all the stored values in $HT(P)$ for all P require $\widetilde{O}(\frac{n^2}{\sqrt{r}})$ time.*

Proof. For a piece P, the polynomials are computed for each dual node on the bisectors. There can be $O(r)$ dual nodes on each bisector and $O(r)$ total bisectors for each pair of sites. Polynomials are also constructed for each node of each cotree. Therefore total number of polynomials per piece can be $O((\sqrt{r}^2 \times r \times r) + (\sqrt{r} \times r)) = O(r^3)$. Also, degree of each such polynomial is $O(r)$. There are $M = O(r^3)$ many polynomials per peice of degree $d = O(r)$ each that need to be evaluated on $N = O(n \times \sqrt{r})$ different points. Therefore these N points can be grouped into atmost M groups, such that the points in each group are evaluated for the same polynomial.

Suppose c_i denote the number of points in each group and $\sum_i c_i = N$. Then the total time complexity for evaluating all the M polynomials for all N points is $\sum_{i \leq M} \max(c_i, d) log^2 d$ (as each polynomial is of degree $d = O(r)$). Suppose $c_i = O(n\sqrt{r})$, then there can only be $\theta(\sqrt{r})$ many such terms and $\max(c_i, d) = O(n\sqrt{r})$. Therefore the total time taken to evaluate will be $O(\sqrt{(r)} \times n \log^2 r)$. If $c_i = O(r^3)$, then there can be $\theta(\frac{n \times \sqrt{r}}{r^3})$ such terms. So total time taken in this case will be $O(\frac{n\sqrt{r}}{r^3} \times r^3 \log^2 r) = O(n\sqrt{r}\log^2 r)$.

Repeating this for all the pieces will take $\widetilde{O}(\frac{n^2}{\sqrt{r}})$ time. ◀

Lemma 48 also holds for HC, since δ may not be same as the weight difference.

▶ **Lemma 53.** *For any node v in piece P and some internal node s not in P, $\mathcal{T}(b,v) = \frac{1}{Dist(b,v)+w(b)}$ is added to $HC(s)$ exactly once iff v is in $Vor(b)$ (AWV decomposition w.r.t the distances from s to the boundary nodes of P).*

Proof. Suppose v is associated with face f and dual node of f is f^*. Let v be associated with the segment of bisector $\beta^*(b,b',\delta)$.

- $(\Rightarrow)$ If v is not in $Vor(v)$, then $\mathcal{T}(b,v)$ will not be added to $HC(s)$:
  f^* can either lie on the cycle C_0^* or face f can be entirely outside $Vor(b)$. If f^* lies on C_0^*, then from the lemma 37, each node t on the face f is checked for the following condition: $dist(b,v) - dist(b',v) < \delta$. Since the condition is false in case of v (v not in $Vor(b)$), therefore $\mathcal{T}(b,v)$ will not be added to $HC(s)$. Now, if face f is entirely outside $Vor(b)$, then the root to f^* path of T^* does not contain any dual node which is on any cycle, therefore, for no node of f, $\mathcal{T}(b,v)$ will be added to $HC(s)$.

- $(\Leftarrow)$ If v is in $Vor(v)$, then $\mathcal{T}(b,v)$ will not be added to $HC(s)$ exactly once:
  To prove this, we use lemma 22.

  Case 1: If the root to f^* path of T^* has only one boundary node, then, f^* has to lie on C_0^*. As discussed above, if f^* lies on C_0^*, then after checking the condition, $\mathcal{T}(b,v)$ will be added to $HC(s)$ (since, v is in $Vor(b)$). Since, the root to f^* path in T^* has only one boundary node and a vertex is associated with a single face (single dual node in T^*), therefore, $\mathcal{T}(b,v)$ won't be added to $HC(s)$ again.

  Case 2: If the root to f^* path of T^* has k penetrating edges and $k-1$ exiting edges, then polynomials $P1_2(e^*)(x)$ and $P2_2(e^*)(x)$ will be added to HT with value $< s, Dist(s,b), - >$ inserted for all exiting edge endpoint e^* on the path. Therefore, value $< s, Dist(s,b), - >$ is inserted $k-1$ times. Since, the α_1 form of the polynomial $\frac{P1_2(e^*)(x)}{P2_2(e^*)(x)}$ (see section 2.4.1) for all e^* has a term $\frac{1}{Dist(b,v)+x}$ (as f^* is in the subtree of all these e^* in T^*), therefore, $\frac{1}{Dist(b,v)+x}$ will be evaluated for $x = Dist(s,b)$ and will be subtracted from $HC(s)$ (because $sign = -$) $k-1$ times. Therefore, we can say that $(k-1) \times \mathcal{T}(b,v)$ is subtracted from HC (as $Dist(s,b) = w(b)$).
  Now, for every penetrating edge endpoint p^* on the path, $P1_2(e^*)(x)$ and $P2_2(e^*)(x)$ will be added to HT with value $< s, Dist(s,b), + >$ inserted. Since f^* is in the subtree of these p^* in T^* and each p^* has a unique parent pp^* which lies on one of the cycles, therefore, each of these p^* is associated with a unique segment of a bisector (on each cycle). For each of these segments, we have a polynomial $P1(x)/P2(x)$ (given in lemma 43 or lemma 37). All these polynomial in the α_1 form will have $\frac{1}{Dist(b,v)+x}$ as a term. Therefore, $\frac{1}{Dist(b,v)+x}$ will be evaluated and added to $HC(s)$ for $x = Dist(s,b)$, k many times. So, $k \times \mathcal{T}(b,v)$ is added to HC (as $Dist(s,b) = w(b)$).
  Therefore, we can say that $\mathcal{T}(b,v)$ is added to $HC(s)$ exactly once.

Case 3: If the root to f^* path of T^* has x penetrating edges and x exiting edges, it means f^* is on some cycle C^*. Suppose (p^*, q^*) is the exiting edge of cycle C^* and it lies on the path from root to f^*. If we consider the subpath excluding the path $p^* \rightsquigarrow f^*$, then it has x penetrating edges and $x-1$ exiting edges. Therefore, by the previous argument, we can say that, for that subpath, $\mathcal{T}(b, v)$ is added once. Now, we subtract $\mathcal{T}(b, v)$ once for the exiting edge (p^*, q^*), but then we also add $\mathcal{T}(b, v)$ once for the node f^* itself. Therefore, in total $\mathcal{T}(b, v)$ is added exactly once.

◀

Therefore, the above lemma and lemma 48 proves the correctness of the algorithm for computing HC of all nodes in planar graphs.

Running time analysis

We finish the proof of Theorem 50 by discussing it's subquadratic complexity. The running times of the different stages are explained below.

Pre-processing: r-division takes $O(n \log n)$ time using the naive recursive approach by Frederickson[14]. We can also use a recently proposed linear time algorithm [30].

Stage 1: Computing $HC(s)$ of the boundary nodes can be found by running $Dijkstra's$ algorithm on G taking each boundary as a source. The number of boundary nodes in G is $O(\frac{n}{\sqrt{r}})$ leading to a running time of $O(\frac{n}{\sqrt{r}} \times n \log n)$.

Stage 2: Computing internal $HC(s)$ also involves running $Dijkstra$ on the graph G_P with $O(r)$ nodes and $O(r)$ edges (in G_P we add $O(r)$ edges between every pair of boundary nodes). The running time of this stage is $O(\frac{n}{r} \times r \log r)$.

Stage 3: The AWV pre-processing requires $\widetilde{O}(\frac{n}{r} \times r^2)$ time. Construction of the cotree require $\widetilde{O}(nr^{\frac{3}{2}})$ time (theorem 32). Construction of the data structure $ABBST$ (or AL) HC require $\widetilde{O}(nr^2)$ (or $O(nr^3)$) time by theorem 44 (or theorem 38).

Stage 4: In this stage, we construct AWVs for all internal nodes s and all pieces different from $P(s)$. The construction takes $\widetilde{O}(\sqrt{r})$ time (Theorem 4). From lemma 51 and lemma 52, we can say that, the total time required for this stage is $\widetilde{O}(\frac{n^2}{\sqrt{r}})$ time.

We use $r = n^{\frac{2}{5}}$, we get the total running time for computing HC of all nodes as $\widetilde{O}(n^{\frac{9}{5}})$. If we use AL instead of $ABBST$, the running time will be $\widetilde{O}(n^{\frac{13}{7}})$ using $r = O(n^{\frac{2}{7}})$.

▶ Note 54. The same bound ($\widetilde{O}(n^{\frac{9}{5}})$) for HC (different name in the paper) was proved recently in this paper [6]–Corollary 16, using different techniques–also uses fast multipoint evaluation and AWVD.

4.2.3 NFC of all nodes

The main result of this section is the following theorem.

▶ **Theorem 55.** *For a planar graph with n nodes and $O(n)$ edges, the NFC of all nodes can be computed in $\widetilde{O}(n^{9/5})$ time.*

The stages of the algorithm are very similar to that of CC (section 4.2.1).

Stages 1 and 2

For stage 1, we need to compute $NFC(b)$ for all boundary nodes, and this is done by simply running Dijkstra from each boundary of every piece. Since boundary vertices are part of multiple pieces, a quick check can ensure that each boundary vertex is chosen as b only once.

For any internal node s whose piece we denote by $P(s)$ and we denote the set of all boundary nodes by B , we decompose $NFC(s)$ into three components :

$$\begin{aligned} NFC(s) &= \sum_{t\in B}\sigma(s,t) + \sum_{t\in P(s)\setminus B}\sigma(s,t) + \sum_{t\notin P(s)\ and\ t\notin B}\sigma(s,t) \\ &= NFC_B(s) + NFC_{int}(s) + NFC_{ext}(s) \end{aligned} \tag{4.5}$$

Since in stage 1, we have already calculated $\sigma(s,b)$ for all boundary nodes b, we can substitute the values in equation 4.5 to get $NFC_B(s)$.

In stage 2 we compute $NFC_{int}(s)$ by taking $G_P = (V_P, E_P)$ and running Dijkstra on G_P from all internal nodes of P. The extra edges in G_P handle those paths that goes outside P. But since the source and target nodes are both inside P, such a path must leave and reenter the piece via some boundary node (illustrated in Figure 4.1). Note that the graph G_P has to preserve the number of shortest paths between every pair of vertices along with the shortest path distances.

Consider a piece P. There can be three kinds of paths between any pair of vertices in P:

1. *Internal* paths: All the vertices on an internal path belong to P.
2. *External* paths: None of the vertices (except the endpoints) on an external path belong to P. Note that external paths can only exist between boundary vertices, i.e., end points of any external path are boundary vertices of P.
3. *Mixed* paths: Paths that are neither internal nor external. A mixed path can be decomposed into a sequence of internal and external paths. Also note that any contiguous subpath of a mixed path which is external, starts and ends at two boundary vertices in P.

We are interested in finding the number of external shortest paths between every pair of boundary vertices in a piece P. For this, we follow the following steps and construct G_P:

- We first consider the graph G', which is constructed by deleting all the edges that are in P. Now, for every boundary vertex b in P, we run Dijkstra's algorithm in G' and find the number of shortest paths from b to all the other boundary vertices in P (we also find the shortest path distances from b). Graph G' might be disconnected, but we are only interested in the connected component which has b, therefore, we will run Dijkstra's algorithm in that component only. Note that, since there does not exist any edge from P in G', therefore, for any boundary vertices b' in P, the number of shortest paths that is computed in this step is actually the number of external shortest paths between b and b', wrt piece P. Let's denote this quantity by $\sigma_{ext}(b,b')$ (let's denote the shortest path distance between b and b' in G' by $Dist_{ext}(b,b')$). We find the value of

$\sigma_{ext}(b,b')$ (and $Dist_{ext}(b,b')$) for every pair of boundary vertices b,b' in a piece P, and for every piece P.

- Now, we construct G_P. Initially, $G_P = P$ (i.e., G_P will contain all the edges and vertices of P). Then, for every pair of boundary vertices b,b' in P, we check the following condition: if $Dist(b,b') = Dist_{ext}(b,b')$ and $\sigma_{ext}(b) > 0$, then add edge (b,b') in G_P with edge weight $= Dist(b,b')$. If the edge was already there in P, consider this edge to be $\sigma_{ext}(b,b') + 1$ parallel edges of equal weights. If the edge was not there in P before, consider this edge to be $\sigma_{ext}(b,b')$ parallel edges of equal weights.

Note that G_P may not be a simple graph, but Dijkstra's algorithm also works for graphs with multi edges and self loops.

Stage 3

The only value left to compute is $NFC_{ext}(s)$; however, these involve shortest paths that partly lies inside $P(s)$ and partly lies outside $P(s)$.

The number of shortest paths from s to any $t \notin P(s)$,i.e. $\sigma(s,t)$, can be broken down to $\sigma(t,b)$ for some boundary $b \in P(t)$ (there can be more than one boundary) and $\sigma(b,s)$ (see Figure 4.2a), such that $\sigma(s,t) = \sigma(s,b) \times \sigma(t,b)$. Observe that we already have computed both these distances in Stage 1; however, naively trying all $b \in \partial P(t)$ is not sufficient since it leads to a complexity of $O(n^2\sqrt{r})$ — there are $O(n)$ possible s, $O(r)$ many t in any piece, $O(\sqrt{r})$ many b for any piece and $O(\frac{n}{r})$ pieces.

Our strategy is to again use Voronoi regions to pre-compute the closest boundary node for every internal node in $P(t)$ — this depends on s as well (see Figure 4.2b). Let P denote the piece of t and let b be a boundary node of P. We define $J(s,b)$ as the set of internal nodes of P whose shortest path to s pass through b. For example, $J(s,b_1)$ contains t_1 in Figure 4.2a. Note that unlike in CC, in NFC, $J(s,b_i)$ are not disjoint across all b_i's.

$$\begin{aligned}
NFC_{ext}(s) = \sum_{t \notin P(s)} \sigma(s,t) &= \sum_{\text{piece } P \neq P(s)} \sum_{b \in \partial P} \sum_{t \in J(s,b)} [\sigma(s,b) \times \sigma(b,t)] \\
&= \sum_{\text{piece } P} \sum_{b \in \partial P} \left(\sigma(s,b) \times \sum_{t \in J(s,b)} \sigma(b,t) \right) \\
&= \sum_{\text{piece } P} \sum_{b \in \partial P} \left(\sigma(s,b) \times N(s,b) \right) \qquad (4.6)
\end{aligned}$$

The crux of the final stage is a technique to efficiently compute $J(s,b)$ and the bracketed expression in Equation 4.6. We will not compute $J(s,b)$ explicitly, but will compute the expression: $N(s,b)$ efficiently using the voronoi diagram for each boundary node b in P.

In this stage, we construct the AL or $ABBST$ data structures for all pieces. Details are given in section 3.3.3. We also do the preprocessing needed for constructing the AWV diagram ([18]).

Stage 4

Now in this stage, we actually compute the $NFC_{ext}(s)$ using the values computed in stage 3. The notations are as usual. We will describe the algorithm for a single boundary node b, and repeating this for all the boundary nodes will give us the end result.

1. For each piece P and each internal node s not in P, we compute the AWV diagram with respect to the distance from s to all boundary nodes in P. From this, we get the relevant segments of bisectors. We denote $Dist(s, b)$ as $w(b)$ for the proofs in this section.

2. $N(s, b)$ can be computed by adding num_paths3 values of each segment, as described in lemma 39. Let f^* be the node that is shared between two consecutive segments of bisector on a cycle $C^* \in \mathcal{C}^*$ and $f^* \neq h^*$, where h^* is the dual node of hole h which is the hole of b. Therefore f^* is a voronoi vertex of degree 3 and the edge f^*g^* which is not on the cycle C^* is not in $Vor(b)$, i.e the vertices of face g are not in $Vor(b)$. So there are no edges from f^* that are entering into $Vor(b)$. We only need to check which vertex of f is inside $Vor(b)$(say vertex t), we add $\sigma(b, t)$ to $N(s, b)$.

3. And ultimately, for each of the cycles in $\mathcal{C}^* \setminus \{C_0^*\}$, we figure out the unique edge f^*g^* (lemma 19). We subtract $num\_paths3(f^*)$ from $N(s, b)$ for each cycle. If there is some other child of f^* in T^* (let's say $t^* \neq g^*$), then we add $num\_paths2(t^*)$ to $N(s, b)$.

After we calculate $N(s, b)$ for all b, we just substitute the values in equation 4.6 to get the value of $NFC_{ext}(s)$.

Note that lemma 48 does not hold for NFC. There can be vertices outside $Vor(b)$ which are included in $J(s, b)$. Moreover, $J(s, b)$ for all b are no longer disjoint.

▶ **Lemma 56.** *For each vertex t , if t is in $Vor(b)$, then the distances and count are added exactly once.*

The proof is similar to the forward direction proof of lemma 49

▶ **Lemma 57.** *If a node t is in $Vor(b)$, then the the shortest path from s to t will pass through b.*

Proof. Suppose there exist a node $p \in Vor(b)$ and the shortest path from s to p does not pass through b but passes through some other boundary node u, that means $Dist(b, p) + w(b) > Dist(u, p) + w(u)$ and $Dist(b, p) - Dist(u, p) \leq \delta.original \leq w(u) - w(b)$, which is a contradiction. ◀

From lemma 56 and lemma 57, we can say that every node from $Vor(b)$ will be there in $J(s, b)$.

Note that, even though we did not compute $J(s, b)$ explicitly, but it can be said that t is included in $J(s, b)$ iff $\sigma(b, t)$ is added to $N(s, b)$.

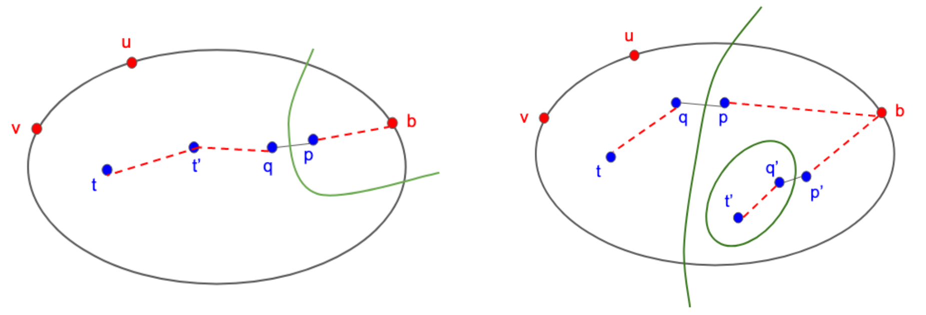
▶ **Lemma 58.** *The nodes in $J(s, b)$ forms a connected subtree of T rooted at b.*

Proof. Suppose t' be the ancestor of t in T and t belongs to $J(s, b)$, but t' does not. Therefore there must exist some u, such that t' belongs to $J(s, u)$ (figure 4.3a). So,

$$\begin{aligned} & Dist(b, t') - Dist(u, t') > \delta.original \\ \implies & Dist(b, t') + Dist(t, t') - Dist(t, t') - Dist(u, t') > \delta.original \\ \implies & Dist(b, t) - Dist(u, t) > \delta.original \end{aligned}$$

But, we know that $Dist(b, t) - Dist(u, t) < \delta.original$, because t belongs to $J(s, b)$. Therefore we arrive at a contradiction.

◀



(a) A path from b to t in T is shown. $t \in Vor(v)$ and $t' \in Vor(u)$. $Vor(u)$ is adjacent to $Vor(b)$.

(b) A path from b to t and a path from b to t' in T is shown. t and t' are not in $Vor(b)$. p and p' are in $Vor(b)$.

▶ **Lemma 59.** *A node t not in $Vor(b)$ is included in $J(s,b)$ (or $\sigma(b,t)$ is added to $N(s,b)$) iff the shortest path from s to t passes through b.*

Proof. Suppose $t \in Vor(v)$ for some boundary node v in P. Let t' be an ancestor of t in T and $t' \in Vor(u)$. Also, $Vor(u)$ is adjacent to $Vor(b)$. Refer to figure 4.3b.

($\Rightarrow$) If t is included in $J(s,b)$ then the shortest path from s to t passes through b: Let us assume that t is included in $J(s,b)$ but the shortest path from s to t does not pass through b. From the lemma 58, we know that t' must also be included in $J(s,b)$. Therefore,

$$\begin{aligned} Dist(b,t') - Dist(u,t') &= Dist(b,t) - Dist(u,t) \\ &= \delta.original \\ &\le w(u) - w(b) \end{aligned} \tag{4.7}$$

The shortest path from s to t does not pass through b and passes through v. Therefore,

$$\begin{aligned} & Dist(b,t) + w(b) > Dist(v,t) + w(v) \\ \implies & Dist(b,t') + Dist(t,t') + w(b) > Dist(v,t) + w(v), \\ & \text{(since } Dist(b,t) = Dist(b,t') + Dist(t,t')) \\ \implies & Dist(u,t') + w(u) > Dist(v,t) - Dist(t,t') + w(v), \\ & \text{(from equation 4.7 we get } Dist(u,t') + w(u) \ge Dist(b,t') + w(b)) \\ \implies & Dist(u,t') + w(u) > Dist(v,t') + w(v), \\ & \text{(this is a contradiction, because } t' \text{ is in } Vor(u)) \end{aligned} \tag{4.8}$$

Note that it will still be a contradiction if $u = v$.

($\Leftarrow$) If the shortest path from s to t passes through b , then t is included in $J(s,b)$: Let us assume that the shortest path from s to t passes through b, but t is not included in $J(s,b)$. Therefore,

$$Dist(b,t) + w(b) = Dist(v,t) + w(v) \tag{4.9}$$

Since t is not included in $J(s,b)$, therefore, $Dist(b,t) - Dist(u,t) > \delta.original$. We can also say that

$$Dist(b,t) - Dist(u,t) > w(u) - w(b) \tag{4.10}$$

, because $w(u) - w(b) > \delta.original$, t will still be outside $Vor(b)$ (as for all weight differences $> \delta$, t remains outside $Vor(b)$).
If $u = v$, then from equation 4.9 and equation 4.10, we get a contradiction.
If $u \neq v$, then we get, $Dist(v, t) + w(v) > Dist(u, t) + w(u)$, which is not possible, since $t \in Vor(v)$. ◀

From corollary 28, we can say that a node not in $Vor(b)$ is in included exactly once iff the shortest path from s to t passes through b. Therefore, from lemma 56, lemma 57 and lemma 59 $N(s, b)$ is computed correctly.

Running time analysis

We finish the proof of Theorem 55 by discussing it's subquadratic complexity. The running times of the different stages are explained below.

Pre-processing: r-division takes $O(n \log n)$ time using the naive recursive approach by Frederickson[14]. We can also use a recently proposed linear time algorithm [30].

Stage 1: Computing $\omega(s)$ of the boundary nodes can be found by running $Dijkstra's$ algorithm on G taking each boundary as a source. The number of boundary nodes in G is $O(\frac{n}{\sqrt{r}})$ leading to a running time of $O(\frac{n}{\sqrt{r}} \times n \log n)$.

Stage 2: Computing $\omega_{int}(s)$ also involves running $Dijkstra$ on the graph G_P with $O(r)$ nodes and $O(r)$ edges (in G_P we add $O(r)$ edges between every pair of boundary nodes). The running time of this stage is $O(\frac{n}{r} \times r \log r)$.

Stage 3: The cost for AWV pre-processing is $\widetilde{O}(\frac{n}{r} \times r^2)$. It will require additional $O(nr^2)$ time to construct the data structure needed for calculating NFC (theorem 40)

Stage 4: In this stage, we construct AWVs for all internal nodes s and all pieces different from $P(s)$. The construction takes $\widetilde{O}(\sqrt{r})$ time (Theorem 4). That is followed by a processing of all boundary nodes in every piece. Therefore the total running time of this stage is $\widetilde{O}(1) \times O(n \times \frac{n}{r} \times \sqrt{r})$.

Since we use $r = n^{\frac{2}{5}}$, we get the total running time for computing NFC of all nodes as $\widetilde{O}(n^{\frac{9}{5}})$.

4.2.4 BNFC of a single node

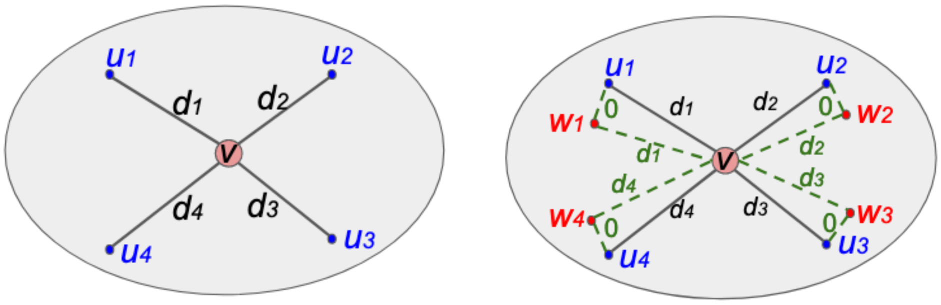
▶ **Theorem 60.** *For a planar graph with n nodes and $O(n)$ edges, the BNFC of single node can be computed in $\widetilde{O}(n^{\frac{9}{5}})$ time.*

Proof. $BNFC$ of single node v can be found using algorithm 15. This is basically a reduction from $BNFC$ (v) to NFC. Since NFC of all nodes in planar graphs can be computed in $\widetilde{O}(n^{\frac{9}{5}})$ time (theorem 55), therefore using the claim 61, we can say that BNFC(v) can be computed in $\widetilde{O}(n^{\frac{9}{5}})$ time. ◀

Algorithm 15: Finding $BNFC$ of v

```
   input  : Graph $G(V,E)$ and node $v$
   output: $BNFC\ (v)$
 1 Create a new graph $G_n$ (see Figure 4.4b).
 2 $G_n \leftarrow G$
 3 for each neighbour $u_i$ of $v$ do
 4     create a node $w_i$
 5     Add weighted edges $(w_i, u_i) = 0$ and $(w_i, v) = dist(u_i, v)$.
 6 end
   /* Suppose $NFC(G) = \sum_{s_i} \sum_{t_i} \sigma(s_i, t_j) - \sum_{s \neq v}(\sigma(s,v) + \sigma(v,s)) =$
      $\sum_{s \neq v}(\mathtt{NFC}(s) - \sigma(s,v))$.   */
 7 $BNFC_1 \leftarrow NFC(G_n)$
 8 $BNFC_2 \leftarrow NFC(G)$
 9 $BNFC(v) \leftarrow \frac{BNFC_1 - BNFC_2}{3}$
```

Suppose $NFC\ (G) = \sum_{s_i} \sum_{t_i} \sigma(s_i, t_j)$



(a) Graph G with a node v and it's neighbours **(b)** Modified graph G_n

Figure 4.4: Showing G and G_n with the node v

▷ Claim 61. $NFC(G_n) - NFC(G) = x$ iff BNFC(v) in G is $\frac{x}{3}$.

Proof. Suppose $\sigma(s,t) = \sigma(s,t,v) + \overline{\sigma}(s,t,v)$, where $\overline{\sigma}(s,t,v)$ is the number of shortest paths between s and t that does not pass through v. The proof follows from the fact that, all the $\overline{\sigma}(.,.,v)$ terms gets cancelled out in $NFC(G_n) - NFC(G) = x$, so x only has $\sigma(.,.,v)$ terms, i.e,. the paths that pass through v. Now note that, if $\sigma(s,t,v) = k$ in G, then $\sigma(s,t,v) = 4k$ in G_n, because of those additional edges. Therefore, subtracting $\sigma(s,t,v)$ in G_n from $\sigma(s,t,v)$ in G will give us $3k$ for each s,t. ◁

Chapter 5

Lower bounds

We present the lower bounds (quadratic) of various centrality measures on both planar and sparse graphs in this chapter.

For the sparse graph lower bounds in section 5.1, we have used a similar technique that was proposed in the paper [3] for proving the quadratic lower bound of BC in sparse graphs. In fact, the underlying unweighted graph (reduced graph) is exactly the same graph as given in the paper (theorem 4.3 of [3]).

Unlike for sparse graphs, there aren't many lower bound results for planar graphs when it comes to problems in P. This may be because of the fact that most of the planar variant of the problems in P have faster upper bounds than their generalized sparse variant, like the ones that are given in this thesis. Most of the algorithms use the planar separator theorem (and some other structural properties) to achieve the faster upper bound. In section 5.2, we have proved that it is unlikely that BHC of all nodes can be computed in truly-subquadratic time even in planar graphs. We have reduced a variant of the 3SUM problem to BHC for planar graphs. Although two recent papers ([2, 1]) have some results on planar lower bounds, but none of the techniques in the papers use 3SUM conjecture. Our reduction in section 5.2 is completely different from the reductions in the other two papers.

5.1 Lower Bound for Sparse graphs

▶ **Theorem 62.** *If we can find an $O(n^{2-\epsilon})$ time algorithm for CC in sparse graph then we can solve CNF-SAT in $O((2-\delta)^n)$(number of variables and clauses is $O(n)$) time, which will imply that SETH is false.*

First, we present Algorithm 16 for reducing CNF-SAT to the problem of computing CC of all nodes.

Algorithm 16: Reduction from CNF-SAT to CC

```
   input : CNF-SAT Formula $F$ on $n$ variables and $m$ clauses
   output: Graph $G(V,E)$
 1 create $Null$ Graph $G(V,E)$
 2 Divide number of variables into two disjoint sets($X$ and $Y$) of size $\frac{n}{2}$ each.
 3 $X_i \leftarrow i^{th}$ Partial Assignment on $X$, where $i \in \{1,2,3,\ldots,2^{\frac{n}{2}}\}$
 4 $Y_i \leftarrow i^{th}$ Partial Assignment on $Y$, where $i \in \{1,2,3,\ldots,2^{\frac{n}{2}}\}$
 5 $c_j \leftarrow j^{th}$ clause, where $j \in \{1,2,3,\ldots,m\}$
 6 for $i \leftarrow 1$ to $2^{\frac{n}{2}}$ do
 7     create node $A_i$ for each $X_i$ and add it to $V$
 8     create node $B_i$ for each $Y_i$ and add it to $V$
 9 end
10 for $i \leftarrow 1$ to $m$ do
11     create node $C_i$ for each clause $c_i$
12     $V \leftarrow V \cup C_i$
13     for $j \leftarrow 1$ to $2^{\frac{n}{2}}$ do
14         if Partial assignment $X_j$ does not satisfy clause $c_i$ or variables in $X$ are
            missing in clause $c_i$ then
15             Add an edge of weight 1 between $C_i$ and $A_j$
16         if Partial assignment $Y_j$ does not satisfy clause $c_i$ or variables in $Y$ are
            missing in clause $c_i$ then
17             Add an edge of weight 1 between $C_i$ and $B_j$
18     end
19 end
20 Create nodes $b, S_a, S_b$ and add them to $V$
21 Add an edge of weight 1 from $S_a$ to every $A_i$, where $i \in \{1,2,3,\ldots,2^{\frac{n}{2}}\}$
22 Add an edge of weight 1 from $S_b$ to every $B_i$, where $i \in \{1,2,3,\ldots,2^{\frac{n}{2}}\}$
23 Add an edge of weight $x$ from $b$ to every $A_i$ and $B_i$, where $i \in \{1,2,3,\ldots,2^{\frac{n}{2}}\}$
```

Next, observe that for showing the reduction for CC it is sufficient to show that $\omega(A_i)$ for all i is $>$ something.

▶ **Lemma 63.** *A CNF-SAT formula is unsatisfiable iff for every pair of (A_i,B_j) the shortest path between them passes through some C_k.*

Proof. For any pair (A_i,B_j), if the shortest path between them does not pass through any C_k, then we can say that atleast one of X_i, Y_j satisfy all the clauses which implies assignment (X_i, Y_j) satisfies every clause. This will imply that F is satisfiable.
Now suppose F is satisfiable, then there is an assignment (X_i, Y_j) that satisfies all the clauses. This implies either X_i or Y_j or both satisfy all the clauses. If X_i satisfy all the clauses, then there won't be any edge between A_i and any C_k. If Y_j satisfy all the clauses, then there won't be any edge between B_j and any C_k. Then, in any case, there won't be any path between A_i and B_j that passes through some C_k. ◀

Proof of Theorem 62. We claim that CNF-SAT formula F is satisfiable *iff* there exists an A_i such that $\omega(A_i) > 5(2^{\frac{n}{2}}) + 4m + 2$ (Similarly we can also take the B_i). Observe that:

I: $dist(A_i, S_a) = 1$ because there is an edge of weight 1 between A_i and S_a.

II: $dist(A_i, A_j) = 2$, for every $j \neq i$, because of the path $A_i \rightsquigarrow S_a \rightsquigarrow A_j$. Therefore $\sum_{j \neq i} dist(A_i, A_j) = 2(2^{\frac{n}{2}} - 1)$.

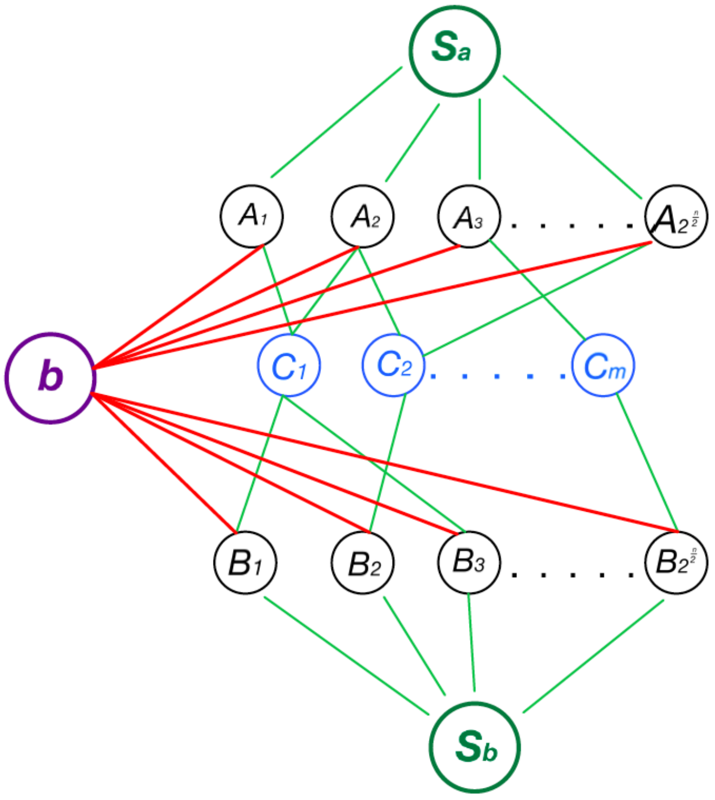


Figure 5.1: red edges have weight $m + 1$ and green edges have weight 1

III: $dist(A_i, b) = x$, because there is an edge of weight 2 between A_i and b.

- if F is unsatisfiable then:

  i: $dist(A_i, C_j) \leq 3$, for any j, more accurately either 1 or 3. There can either be a direct edge $A_i \rightsquigarrow C_j$ or a path passing through S_a. But for atleast one C_j, $dist(A_i, C_j) = 1$, because of Lemma 63. Therefore $\sum_j dist(A_i, C_j) \leq 3(m-1) + 1$.

  ii: $dist(A_i, B_j) = 2$, for any j, because of Lemma 63. The path passes through any C_k. Therefore $\sum_j dist(A_i, B_j) = 2(2^{\frac{n}{2}})$.

  iii: $dist(A_i, S_b) = 3$ because of Lemma 63. There is a path $A_i \rightsquigarrow C_j \rightsquigarrow B_k \rightsquigarrow S_b$.

- Then $\omega(A_i) \leq 1[I] + 2(2^{\frac{n}{2}} - 1)[II] + x[III] + [3(m-1) + 1][i] + 2(2^{\frac{n}{2}})[ii] + 3[iii] = 4(2^{\frac{n}{2}}) + 3m + x$, if F is unsatisfiable.

- if F is satisfiable then atleast for one A_i:

  a: $dist(A_i, C_j) \leq 3$, for any j, more accurately either 1 or 3. Same reason as above. Therefore $\sum_j dist(A_i, C_j) \geq m$.

  b: $dist(A_i, B_j) = 2$ or $dist(A_i, B_j) = 2x$ for any j, because of Lemma 63. The path can be either $A_i \rightsquigarrow C_j \rightsquigarrow B_k$ or $A_i \rightsquigarrow b \rightsquigarrow B_j$. Infact there should be atleast one such pair (A_i, B_j) with $dist(A_i, B_j) = 2x$. Therefore $\sum_j dist(A_i, B_j) = 2(2^{\frac{n}{2}} - 1) + 2x$.

c: $dist(A_i, S_b) = 3$ or $dist(A_i, S_b) = 2x + 1$ because of Lemma 63. The path can be either $A_i \rightsquigarrow C_j \rightsquigarrow B_k \rightsquigarrow S_b$ or $A_i \rightsquigarrow b \rightsquigarrow B_j \rightsquigarrow S_b$.

- Then $\omega(A_i) \geq 1[I] + 2(2^{\frac{n}{2}} - 1)[II] + x[III] + m[a] + [2(2^{\frac{n}{2}} - 1) + 2x][b] + 3[c] = 4(2^{\frac{n}{2}}) + m + 3x$, if F is satisfiable.

Therefore for $x = m + 1$, we can say that F is unsatisfiable if $\omega(A_i) \leq 4(2^{\frac{n}{2}}) + 4m + 1$ and is satisfiable if $\omega(A_i) \geq 4(2^{\frac{n}{2}}) + 4m + 3$.
Now suppose for some A_i, B_j, the path between them does not pass through any C_k, then $\omega(A_i) \geq 1[I] + 2(2^{\frac{n}{2}} - 1)[II] + x[III] + m[a] + [2(2^{\frac{n}{2}} - 1) + 2x][b] + 3[c] = 4(2^{\frac{n}{2}}) + m + 3x = 4(2^{\frac{n}{2}}) + 4m + 3$. Then from 63, we can say that F is satisfiable.

Now we can say that if $\omega(A_i)$ can be found in $O(m^{(2-\epsilon)})$ time, for some $\epsilon > 0$, then we can solve CNF-SAT in time $O((n(2^{\frac{n}{2}}))^{(2-\epsilon)}) = O^*((2^{(1-\delta)})^n) = O^*((2 - \delta')^n)$, for $\delta, \delta' > 0$.
The number of nodes in the graph, $|V| = 2^{\frac{n}{2}} + 2^{\frac{n}{2}} + m + 3 = O(2^{\frac{n}{2}})$. The number of edges, $|E| = 2^{\frac{n}{2}} + 2^{\frac{n}{2}} + n(2^{\frac{n}{2}}) + n(2^{\frac{n}{2}}) + 2(2^{\frac{n}{2}}) = O(n(2^{\frac{n}{2}}))$. Therefore the reduction takes $O^*((2 - \delta)^n)$ time. ◀

▶ **Theorem 64.** *For a sparse graph with N nodes and O(N) edges, the HC of all nodes cannot be computed in truly subquadratic time.*

Proof. Here also we reduce CNF-SAT to *HC*.
The reduced graph will be the same as in *CC* (illustrated in Figure 5.1) with weights on red edges to be 2 and weights on green edges to be 1.
Here we only consider F with even number of variables for simplicity.
Note that it is enough to show the hardness of $HC'(A_i) = \sum_j \frac{1}{dist(A_i, B_j)}$ for all A_i. The following claim will finish the proof of the theorem. ◀

▷ Claim 65. We claim that F is unsatisfiable *iff* HC' of all $A_i = \frac{1}{2} \times 2^{\frac{n}{2}}$.

Proof. Note that HC' of each A_i has $2^{\frac{n}{2}}$ terms.
Suppose F is satisfiable. Then there exists X_i, Y_j such that either X_i or Y_j satisfy all the clauses (from lemma 63). This implies that there exists (A_i, B_j), such that the shortest path between A_i and B_j passes through b only. Since $dist(A_i, B_j) = 2$ or 4, then if atleast one term out of $2^{\frac{n}{2}}$ terms is $\frac{1}{4}$, we can say that $HC'(A_i) < \frac{2^{\frac{n}{2}}}{2}$.
Suppose for some A_i, $HC\ (A_i) < \frac{2^{\frac{n}{2}}}{2}$. Since there are $2^{\frac{n}{2}}$ terms in $HC\ (A_i)$ and all the terms are either $\frac{1}{2}$ or $\frac{1}{4}$, there must be atleast one term which is $\frac{1}{4}$. That means there is atleast one such B_j for which $dist(A_i, B_j) = 4$. This implies that the path from A_i to B_j does not pass through any C_k and pass through b only. That means either X_i or Y_j satisfy all the clauses. Therefore F will be satisfied by the assignment (X_i, Y_j).
◁

▶ **Theorem 66.** *For a sparse graph with N nodes and O(N) edges, the NFC of all nodes cannot be computed in truly subquadratic time.*

Proof. We use the same reduction algorithm as used in *CC* (Figure 5.1). Only the weights on red edges will be 1 instead of 2.
Again we only consider F with even number of variables for simplicity.
Note that it is enough to show the hardness of $NFC'(A_i) = \sum_j \sigma(A_i, B_j)$ for all A_i. to finish the proof, we will prove the following claim. ◀

▷ Claim 67. We claim that F is unsatisfiable *iff* NFC' of all $A_i = 2^{\frac{n}{2}} \times 2$.

Proof. Note that NFC' of each A_i has $2^{\frac{n}{2}}$ terms.
Suppose F is satisfiable. Then there exists pair (X_i, Y_j) for which F is satisfiable. That means either X_i or Y_j or both satisfy F which implies there exists (A_i, B_j), such that the shortest path between A_i and B_j passes through only b and not through any C_k. Then $NFC'(A_i) < 2^{\frac{n}{2}} \times 2$ because atleast one term out of $2^{\frac{n}{2}}$ terms is 1 and instead of 2.
Suppose $NFC\prime(A_i) < 2^{\frac{n}{2}} \times 2$ and all the terms are either 1 or 2 and the number of terms is $2^{\frac{n}{2}}$, then atleast one of the terms must be 1 for $NFC\ (A_i)$ to be $< 2^{\frac{n}{2}} \times 2$. That means there exists one (A_i, B_j) for which $dist(A_i, B_j) = 1$ as it only passes through b and not through any C_k. This implies that there exists partial assignments X_i, Y_j such that either X_i or Y_j or both satisfy all the clauses. Therefore F is satisfied by the assignment (X_i, Y_j). ◁

▶ **Theorem 68.** *For a sparse graph with N nodes and $O(N)$ edges, the BNFC of even a single node cannot be computed in truly subquadratic time.*

Proof. This is basically the same reduction as shown in the paper [3]. The claim that was shown in the paper was : "F is unsatisfiable *iff* $BC(b) = 0$ (i.e., there does not exist any path that passes through b)". We can use the exact same claim to prove the hardness of $BNFC$, since $BC(v) = 0$ iff $BNFC(v) = 0$.

The reduced graph will be same as in CC (Figure 5.1) with the weights on the red edges to be 2 and the weights on the green edges to be 1.
Again we only consider F with even number of variables for simplicity.
We will prove the hardness of $BNFC\prime(b) = \sum_i \sum_j \sigma(A_i, B_j, b)$ for all A_i which is just a subproblem of $BNFC$ of all nodes. To finish the proof, we will prove the following claim. ◀

▷ Claim 69. We claim that F is unsatisfiable *iff* $BNFC'(b) = 0$.

Proof. This directly follows from Lemma 63. Suppose F is satisfiable, then there exists atleast one X_i, Y_j such that either one of them satisfies all the clauses. Then there exists (A_i, B_j) such that the shortest path between them passes through b. Therefore $BNFC'(b) > 0$.
Suppose $BNFC'(b) = 0$, that means no path pass through b. That means there does not exist any A_i, B_j for which the path passes through b, which implies there is no X_i, Y_j such that either one of them satisfies all the clauses. Therefore F is unsatisfiable. ◁

5.2 Lower Bound for BHC of all nodes

Here we prove that the trivial $O(n^2)$ algorithm is the best algorithm for computing BHC of $O(n)$ nodes unless there is a strictly subquadratic algorithm for solving 3SUM.

▶ **Theorem 70.** *For a planar graph with n nodes and $O(n)$ edges, if the BHC of $O(n)$ nodes can be computed in $O(n^{2-\delta})$ time then 3SUM can be solved in $O(n^{2-\delta})$ time as well.*

Proof. We show a linear-time reduction from 3SUM to BHC of $O(n)$ nodes in Algorithm 17. ◀

We consider the following version of 3SUM: Given n positive integers $\{a_1, \ldots, a_n\}$, where all integers are distinct, $a_i > 0 \forall 0 \geq i \leq n$, each integer is even and there does not exist any a_i, a_j, such that $2a_i = a_j$, does there exist a_i, a_j, a_k such that $a_i + a_j = a_k$.

Note that this version is as hard as the original version of 3SUM defined in section 2.5.

Algorithm 17: Reduction from $3SUM$ to BHC

```
   input  : set of $n$ numbers $\{a_1, a_2, \ldots, a_n\}$
   output: Graph $G(V, E)$
 1 create two nodes $x$ and $y$
 2 create $n$ nodes $S = \{s_1, s_2, \ldots, s_n\}$, add edges $(s_i, x)$ for $i \in \{1, n\}$ with
     $w(s_i, x) = a_i$
 3 create $n$ nodes $T = \{t_1, t_2, \ldots, t_n\}$, add edges $(t_i, y)$ for $i \in \{1, n\}$ with
     $w(t_i, y) = -a_i$
 4 create $n$ nodes $\{v_1, v_2, \ldots, v_n\}$
 5 add edges $(v_i, x)$ and $(v_i, y)$ for $i \in \{1, n\}$ with $w(v_i, x) = a_i$ and $w(v_i, y) = b$;
```

The reduced graph is illustrated in Figure 5.2. For the proof of the theorem, it suffices to prove the following claim.

▷ Claim 71. $3SUM$ returns true *iff* $\mathrm{BHC}(v_i) > 2n^2 + 4n + 3$ for some v_i.

Proof. Consider any three $s_i \in S$, $t_k \in T$ and $v_j \in \{v_1, \ldots v_n\}$.

First, observe that the sum of any three numbers of this set $\{a_1, \ldots, a_n\}$ is either 0 or always even.

$$
\begin{aligned}
dist(s_i, t_k, v_j) &= dist(s_i, x) + dist(x, v_j) + dist(v_j, y) + dist(y, t_k) \\
&= \begin{cases} a_i + a_j + b - a_k = b & \text{if } a_i + a_j - a_k = 0 \\ m + b & \text{otherwise, where } m \geq 2 \text{ or } m \leq -2 \end{cases}
\end{aligned}
$$

For the first case, $dist(s_i, t_k, v_j) = (1/b) = \frac{1}{4n^2+8n+4}$ for $b = 4n^2 + 8n + 4$.

For the second case, $dist(s_i, t_k, v_j) = m + b$. Note that if $|m| \geq 2$, then either $(m + b) < -1$ or $(m + b) > 1$ which implies that $-1 < \frac{1}{(m+b)} < 1$.

In all the other cases $dist(a, b, v_i) = m + b$ for $m \geq 2$ and $m \leq -2$. We can write the equation for $BHC(v_i)$ as follows:

$$
\begin{aligned}
BHC(v_j) &= 2 \times \left(\sum_{1 \leq i,j,k \leq n} \frac{1}{dist(s_i, t_k, v_j)} \right) + 2 \times \left(\sum_{1 \leq i,j \leq n} \frac{1}{dist(s_i, y, v_j)} \right) + \\
&\quad 2 \times \left(\sum_{1 \leq i,j \leq n} \frac{1}{dist(t_i, x, v_j)} \right) + 2 \times dist(x, y, v_j) \\
&= BHC_1(v_j) + BHC_2(v_j) + BHC_3(v_j) + BHC_4(v_j) \qquad (5.1)
\end{aligned}
$$

From the above equation, we can say that the number of terms in $BHC(v_j)$ is equal to (#terms in $BHC_1(v_j)$ + #terms in $BHC_2(v_j)$ + #terms in $BHC_3(v_j)$+ #terms in $BHC_4(v_j)$) = $2n^2 + 4n + 2$.

Now, suppose that $3SUM$ returns true, then there exists distinct a_i, a_j, a_k such that $a_i + a_j - a_k = 0$. Therefore, $dist(s_i, t_k, v_j) = dist(t_k, s_i, v_j) = \frac{1}{b}$. There are total $2n^2 + 4n + 2$ terms in the summation of $BHC(v_j)$, therefore, $\mathrm{BHC}(v_j) > (-1)(2n^2 + 4n + 2 - 1) + (1/d) > 2n^2 + 4n + 3$ for some v_j when $d = 1/(4n^2 + 8n + 4)$.

For the other direction, suppose that $\mathrm{BHC}(v_i) > 2n^2 + 4n + 3$ for some v_i. Since there are $(2n^2 + 4n + 2)$ terms in its summation and since each term can be either in $(-1, 1)$ or equal to $\frac{1}{b} = 4n^2 + 8n + 4$, therefore at least one term should be $\frac{1}{b}$. That means for at least one (s_j, t_k) pair $dist(s_j, t_k, v_i) = b$ for distinct i, j, k. This implies that $a_j + a_i - a_k = 0$, i.e., $3SUM$ returns true.

◁

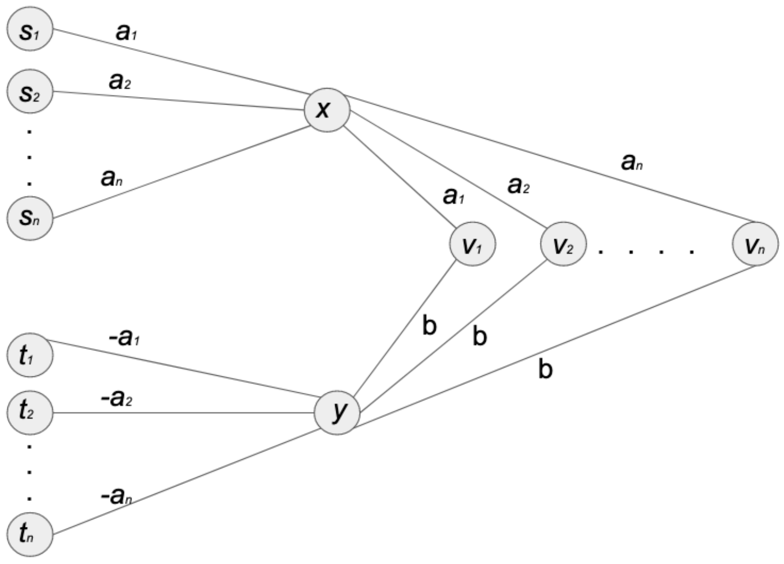


Figure 5.2: The graph obtained after the reduction from 3SUM to *BHC*.

▶ Note 72. The edge weights of the reduced graph in algorithm 17 can be negative. We can also prove theorem 70 if all edge of the planar graph are non-negative. In this case, we have to reduce BHC problem to 3SUM′—Given n non-negative integers $\{a_1, \ldots, a_n\}$, find $SUM' = \left(\sum_{1\leq i,j,k\leq n} \frac{1}{(a_i+a_j+a_k)}\right) + \left(\sum_{1\leq i,j\leq n} \frac{2}{(a_i+a_j)}\right)$. Note that 3SUM is just a decision version of 3SUM′ and 3SUM′ is harder than 3SUM. Now instead of the negative weights on edges (y, t_i), we can put $w(y, t_i) = a_i$ and $b = 0$. So, finding $\sum_i BHC(v_i)$ is basically the 3SUM′ problem because $\sum_i (BHC(v_i) - 2 \times dist(x, y) = 2 \times SUM'$.

Chapter 6

Conclusion

We showed that there a few centrality problems for which the best technique, even for sparse graphs, is probably the trivial quadratic one, and yet a subquadratic algorithm exists for planar graphs. We conclude with the question on the tight complexity of betweenness centrality of planar graphs. Given that it uses number of shortest paths, along with the use of fractions and aggregation, and largely due to the consideration of shortest paths that "pass through" a vertex, instead of merely starting or ending at one, we conjecture that BC of even a single node probably cannot be computed in truly subquadratic time.

Appendix A

Properties of a hole

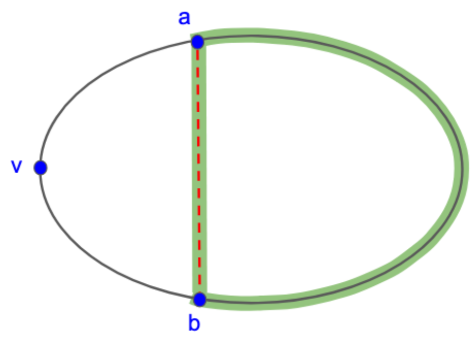
As discussed in section 2.1, a hole is a face of P that is not a face of G. We have the following 2 properties of a hole:

- Every hole must contain atleast one boundary node.
  Since P is a subgraph of G, therefore a face of P which is not a face of G can only be created by deleting some faces from G (deleting vertices and edges). Suppose v is vertex, deleting which created a hole h. Now, $v \notin P$ and v is adjacent to some vertex on h (which by definition will be a boundary vertex of P), therefore, every vertex on h cannot be internal.

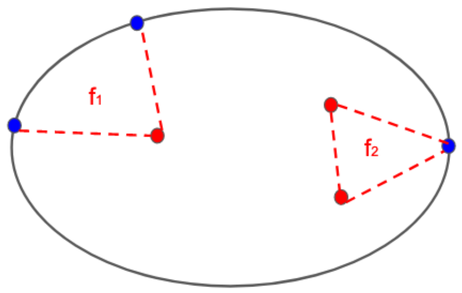
- Every boundary node must lie on some hole.
  Suppose a boundary vertex v lie on a face of P which is also a face of G. In this case all the adjacent vertices of v belong to piece P only, therefore, v by definition is not a boundary node.

▶ **Lemma 73.** *If a planar graph G is triangulated (and biconnected), then all the vertices on a hole are boundary vertices.*

Proof. Suppose v is some vertex on the hole h of piece P and let us assume that v is an internal node of P. That means v was part of some face of G which is not a face of P (since h was created by deleting some faces of G). Therefore, v must be part of some triangulated of G that was deleted. This is only possible if there exists a, b which are adjacent to v and lie on the hole too and there is an edge between a, b (abv forms a triangle). As, it can be seen in figure A.1a, that face abc is not part of the hole because it is a face of G. This leads to a contradiction. ◀



(a) A hole is shown, with a, b, v on the hole.



(b) Two kinds of triangulated face that are allowed.

Bibliography

[1] Amir Abboud, Vincent Cohen-Addad, and Philip N Klein. "New hardness results for planar graph problems in p and an algorithm for sparsest cut". In: *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*. 2020, pp. 996–1009.

[2] Amir Abboud and Søren Dahlgaard. "Popular conjectures as a barrier for dynamic planar graph algorithms". In: *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2016, pp. 477–486.

[3] Amir Abboud, Fabrizio Grandoni, and Virginia Vassilevska Williams. "Subcubic Equivalences Between Graph Centrality Problems, APSP and Diameter". In: *Proceedings of the Twenty-sixth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '15. San Diego, California: Society for Industrial and Applied Mathematics, 2015, pp. 1681–1697. URL: http://dl.acm.org/citation.cfm?id=2722129.2722241.

[4] Jeffrey D Achter et al. *Algebraic Curves and Finite Fields: Cryptography and Other Applications*. Vol. 16. Walter de Gruyter GmbH & Co KG, 2014.

[5] Ulrik Brandes. "A faster algorithm for betweenness centrality". In: *Journal of mathematical sociology* 25.2 (2001), pp. 163–177.

[6] Sergio Cabello. "Computing the inverse geodesic length in planar graphs and graphs of bounded treewidth". In: *arXiv preprint arXiv:1908.01317* (2019).

[7] Sergio Cabello. "Subquadratic algorithms for the diameter and the sum of pairwise distances in planar graphs". In: *ACM Transactions on Algorithms (TALG)* 15.2 (2018), p. 21.

[8] Sergio Cabello, Erin W Chambers, and Jeff Erickson. "Multiple-source shortest paths in embedded graphs". In: *SIAM Journal on Computing* 42.4 (2013), pp. 1542–1571.

[9] Timothy M Chan. "More logarithmic-factor speedups for 3SUM,(median,+)-convolution, and some geometric 3SUM-hard problems". In: *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 2018, pp. 881–897.

[10] Mark De Berg et al. *Computational Geometry: Algorithms and Applications. Santa Clara*. 2008.

[11] James R Driscoll et al. "Making data structures persistent". In: *Proceedings of the eighteenth annual ACM symposium on Theory of computing*. 1986, pp. 109–121. URL: https://www.cadmo.ethz.ch/education/lectures/HS18/SAADS/papers/persistent.pdf.

[12] Dóra Erdős et al. "A divide-and-conquer algorithm for betweenness centrality". In: *Proceedings of the 2015 SIAM International Conference on Data Mining*. SIAM. 2015, pp. 433–441.

[13] Jeff Erickson, Kyle Fox, and Luvsandondov Lkhamsuren. "Holiest minimum-cost paths and flows in surface graphs". In: *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*. 2018, pp. 1319–1332.

[14] Greg N Frederickson. "Planar graph decomposition and all pairs shortest paths". In: *Journal of the ACM (JACM)* 38.1 (1991), pp. 162–204.

[15] Greg N. Frederickson. "Fast Algorithms for Shortest Paths in Planar Graphs, with Applications". In: *SIAM J. Comput.* 16.6 (Dec. 1987), pp. 1004–1022. ISSN: 0097-5397. DOI: 10.1137/0216064. URL: http://dx.doi.org/10.1137/0216064.

[16] Linton C Freeman. "A set of measures of centrality based on betweenness". In: *Sociometry* (1977), pp. 35–41.

[17] Ari Freund. "Improved subquadratic 3SUM". In: *Algorithmica* 77.2 (2017), pp. 440–458.

[18] Pawel Gawrychowski et al. "Voronoi diagrams on planar graphs, and computing the diameter in deterministic Õ (n 5/3) time". In: *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 2018, pp. 495–514.

[19] Michelle Girvan and Mark EJ Newman. "Community structure in social and biological networks". In: *Proceedings of the national academy of sciences* 99.12 (2002), pp. 7821–7826.

[20] Omer Gold and Micha Sharir. "Improved bounds for 3SUM, k-SUM, and linear degeneracy". In: *arXiv preprint arXiv:1512.05279* (2015).

[21] Allan Grønlund and Seth Pettie. "Threesomes, degenerates, and love triangles". In: *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*. IEEE. 2014, pp. 621–630.

[22] David Hartvigsen and Russell Mardon. "The all-pairs min cut problem and the minimum cycle basis problem on planar graphs". In: *SIAM Journal on Discrete Mathematics* 7.3 (1994), pp. 403–418.

[23] Monika R Henzinger et al. "Faster shortest-path algorithms for planar graphs". In: *journal of computer and system sciences* 55.1 (1997), pp. 3–23.

[24] Monika R Henzinger et al. "Faster shortest-path algorithms for planar graphs". In: *journal of computer and system sciences* 55.1 (1997), pp. 3–23.

[25] Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. "Which problems have strongly exponential complexity?" In: *Journal of Computer and System Sciences* 63.4 (2001), pp. 512–530.

[26] Goossen Kant. "Algorithms for drawing planar graphs". PhD thesis. 1993. URL: https://dspace.library.uu.nl/handle/1874/842.

[27] Haim Kaplan et al. "Submatrix maximum queries in monge matrices and partial monge matrices, and their applications". In: *ACM Transactions on Algorithms (TALG)* 13.2 (2017), pp. 1–42.

[28] Shiva Kintali. "Betweenness centrality: Algorithms and lower bounds". In: *arXiv preprint arXiv:0809.1906* (2008).

[29] Alec Kirkley et al. "From the betweenness centrality in street networks to structural invariants in random planar graphs". In: *Nature communications* 9.1 (2018), p. 2501.

[30] Philip N Klein, Shay Mozes, and Christian Sommer. "Structured recursive separator decompositions for planar graphs in linear time". In: *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*. ACM. 2013, pp. 505–514.

[31] Philip Klein et al. "Faster shortest-path algorithms for planar graphs". In: *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*. 1994, pp. 27–37.

[32] Massimo Marchiori and Vito Latora. "Harmony in the small-world". In: *Physica A: Statistical Mechanics and its Applications* 285.3 (2000), pp. 539–546. ISSN: 0378-4371. DOI: https://doi.org/10.1016/S0378-4371(00)00311-3. URL: http://www.sciencedirect.com/science/article/pii/S0378437100003113.

[33] Charudatt Pachorkar et al. "Efficient parallel ear decomposition of graphs with application to betweenness-centrality". In: *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. IEEE. 2016, pp. 301–310.

[34] Alexandros Pappas et al. "Exploring importance measures for summarizing RDF/S KBs". In: *European Semantic Web Conference*. Springer. 2017, pp. 387–403.

[35] Mateusz K Tarkowski et al. "Closeness centrality for networks with overlapping community structure". In: *Thirtieth AAAI Conference on Artificial Intelligence*. 2016.

[36] Dekel Tsur. *Persistent Data Structure*. 2016. URL: https://www.cs.bgu.ac.il/~tids162/wiki.files/lec05.pdf.

[37] Joachim Von Zur Gathen and Jürgen Gerhard. *Modern computer algebra*. Cambridge university press, 2013.

[38] Eric W Weisstein. *Triangulated Graph*.