A Separator-Based Framework for Graph Matching Problems

Nathaniel Lahn

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Sharath Raghvendra, Chair
Lenwood S. Heath
T. M. Murali
Godmar Back
Sanjeev Khanna

May 12, 2020
Blacksburg, Virginia

Keywords: Matching, graphs, graph separators, combinatorial optimization.
Copyright 2020, Nathaniel Lahn
A Separator-Based Framework for Graph Matching Problems

Nathaniel Lahn

(ABSTRACT)

Given a graph $G(V, E)$, a matching $M \subseteq E$ is a set of vertex-disjoint edges. Graph matchings have been well studied, since they play a fundamental role in algorithmic theory as well as motivate many practical applications. Of particular interest is the problem of finding a maximum cardinality matching of a graph. Also of interest is the weighted variant: the problem of computing a minimum-cost maximum cardinality matching. For an arbitrary graph with $m$ edges and $n$ vertices, there are known, long-standing combinatorial algorithms that compute a maximum cardinality matching in $O(m\sqrt{n})$ time. For graphs with non-negative integer edge costs at most $C$, it is known how to compute a minimum-cost maximum cardinality matching in roughly $O(m\sqrt{n}\log nC)$ time using combinatorial methods. While non-combinatorial methods exist, they are generally impractical and not well understood due to their complexity. As a result, there is great interest in obtaining faster matching algorithms that are purely combinatorial in nature. Improving existing combinatorial algorithms for arbitrary graphs is considered to be a very difficult problem. To make the problem more approachable, it is desirable to make some additional assumptions about the graph. For our work, we make two such assumptions. First, we assume the graph is bipartite. Second, we assume that the graph has a small balanced separator, meaning it is possible to split the graph into two roughly equal-size components by removing a relatively small portion of the graph. Several well-studied classes of graphs have separator-like properties, including planar graphs, minor-free graphs, and geometric graphs. For such graphs, we describe a framework, a general set of techniques for designing efficient algorithms. We demonstrate this framework by applying it to yield polynomial-factor improvements for several open-problems in bipartite matching.
A Separator-Based Framework for Graph Matching Problems

Nathaniel Lahn

(GENERAL AUDIENCE ABSTRACT)

Assume we are given a list of objects, and a list of compatible pairs of these objects. A matching consists of a chosen subset of these compatible pairs, where each object participates in at most one chosen pair. For any chosen pair of objects, we say the these two objects are matched. Generally, we seek to maximize the number of compatible matches. A maximum cardinality matching is a matching with the largest possible size. In many cases, there are multiple options for maximizing the number of compatible pairings. While maximizing the size of the matching is often the primary concern, one may also seek to minimize the cost of the matching. This is known as the minimum-cost maximum-cardinality matching problem. These two matching problems have been well studied, since they play a fundamental role in algorithmic theory as well as motivate many practical applications. Our interest is in the design of algorithms for both of these problems that are efficiently scalable, even as the number of objects involved grows very large. To aid in the design of scalable algorithms, we observe that some inputs have good separators, meaning that by removing some subset $S$ of objects, one can divide the remaining objects into two sets $V_1$ and $V_2$, where all pairs of objects between $V_1$ and $V_2$ are incompatible. We design several new algorithms that exploit good separators, and prove that these algorithms scale better than previously existing approaches.
Acknowledgments

This work is supported in part by National Science Foundation grants CCF-1464276 and CCF-1617851. Many thanks to my advisor, Sharath Raghvendra, for his funding, his many hours of helpful discussions, and, especially, for first exposing me to Theory of Algorithms topics in his Spring 2016 CS5114 course. Had I not taken that course senior year, I almost certainly would not have entered graduate school. I would also like to thank Dr. Godmar Back for his wonderful work with the Virginia Tech ACM/ICPC programming team. All those competitive coding years prepared me more for this work than perhaps all my other undergraduate coursework combined. Finally, thank you to my committee members for their feedback towards improving the presentation of this work, both with writing and with presentation to a wider audience.

I would not have made it through this process without the support family and friends, too numerous to name here. I would like to give a special shout-out to Jeff Robertson, Sarah and Daniel Patiño, my parents, my siblings, Tobi, and especially my wife Renee for all the years of mental, social, emotional, and spiritual support. I love you all.
Contents

List of Figures xii

1 Introduction 1

1.1 Prior Work ........................................ 11

1.1.1 Arbitrary Graphs .............................. 11

1.1.2 Planar Graphs ................................. 14

1.1.3 Minor-Free Graphs ............................. 15

1.1.4 Geometric Graphs .............................. 16

1.2 Applications .................................... 19

2 Our Framework and Results 23

2.1 The Framework ................................... 23

2.2 Separator-Based Maximum Matching Algorithm .......... 25

2.2.1 Our Results .................................. 25

2.2.2 Our Approach ................................ 26

2.3 Minimum-Cost Planar Graph Matching Algorithm ........ 27

2.3.1 Our Result and Approach .................... 28

2.4 Min-Cost Minor-Free Graph Matching Algorithm ........ 30
3.4.3 Algorithm for a Scale ........................................... 65
3.4.4 Efficiency of Algorithm ........................................ 66
3.4.5 Concerning the Length of Augmenting Paths ............... 70
3.5 Graph Separators ....................................................... 72
3.6 The Lipton-Tarjan Algorithm ...................................... 74

4 A Weighted Approach to Maximum Cardinality Bipartite Matching 75
4.1 Preliminaries ............................................................. 76
4.2 Our Algorithm .......................................................... 77
4.3 Proof of Invariants ...................................................... 83
4.4 Weights for Graphs with Good Separators ....................... 91
4.5 Minimum Bottleneck Matching .................................... 93

5 Minimum-Cost Bipartite Matching on Planar Graphs 101
5.1 Background ............................................................. 102
5.2 Our Scaling Algorithm ............................................... 104
5.3 Algorithm for Each Scale .......................................... 107
  5.3.1 Computing a Planar Feasible Matching from an \( \mathcal{R} \)-feasible matching 114
  5.3.2 Computing an \( \mathcal{R} \)-Feasible Matching from a Planar Feasible Matching 122
  5.3.3 Second Step of the Algorithm ................................ 127
5.4 Analysis of the Algorithm ......................................... 129
5.4.1 Correctness ......................................................... 129
5.4.2 Efficiency ......................................................... 141

6 Minimum-Cost Matching on Minor-Free Graphs ............................. 146

6.1 Alternative Interpretation of the GT Algorithm ......................... 147
6.2 Preliminaries .......................................................... 152
6.3 Scaling Algorithm ..................................................... 159
6.4 Algorithm for Each Scale .............................................. 163
  6.4.1 Computing a Compressed Feasible Matching from an \( \mathcal{R} \)-Feasible Matching ........................................... 173
  6.4.2 Computing an \( \mathcal{R} \)-Feasible Matching from a Compressed Feasible Matching ........................................... 176
  6.4.3 Second Step of the Algorithm ................................... 179
  6.4.4 The Third Step of the Algorithm ................................ 183
6.5 Analysis of Algorithm .................................................. 183
  6.5.1 Correctness ......................................................... 184
  6.5.2 Efficiency of the Second Step ................................... 190
  6.5.3 Regarding \( r \)-Clusterings in \( K_h \)-Minor Free Graphs .......... 197
6.6 Planar Graph Matching Algorithm ...................................... 198
  6.6.1 Algorithm .......................................................... 200
  6.6.2 Nearest Neighbor Data Structure ................................ 203
6.6.3 Data Structure on a Monge Matrix ............................................. 205

6.7 Appendix .................................................................................. 208

6.7.1 Discussion of Correctness for REDUCE and REDUCESLACK ........ 208

6.7.2 Details of the CONSTRUCT Procedure ...................................... 209

6.7.3 Proofs for Properties of SYNC ................................................. 210

7 An Approximation Algorithm for RMS Matching in a Plane 218

7.1 Introduction .............................................................................. 219

7.2 Our Distance Function ............................................................. 220

7.2.1 Initial Input Transformation .................................................. 220

7.2.2 Randomly Shifted Quadtree Decomposition ............................. 221

7.2.3 Division of a Cell Into Subcells ............................................. 222

7.2.4 Well-Separated Pair Decompositions .................................... 224

7.2.5 Distance Function .................................................................. 227

7.3 Dual Feasibility Conditions ....................................................... 228

7.4 Algorithm ................................................................................. 231

7.4.1 Analysis of the Algorithm ................................................... 240

7.5 Data Structure .......................................................................... 245

7.5.1 Preliminaries ........................................................................ 245

7.5.2 Vertices of the Associated Graph ........................................ 247
7.5.3 Relating parent-child clusters ........................................ 250
7.5.4 Edges of the Associated Graph ..................................... 252
7.5.5 Compressed Feasibility ................................................. 253
7.5.6 The CONSTRUCT Procedure .......................................... 254
7.5.7 SYNC Procedure ......................................................... 256
7.5.8 Data Structure Operations ............................................. 262
7.5.9 AUGMENT Operation .................................................. 270
7.5.10 Simple Augmenting Paths ............................................ 272
7.6 Transforming Input ......................................................... 275
7.7 Omitted Proofs ............................................................. 278
  7.7.1 Proof of Lemma 7.2 .................................................... 278
  7.7.2 Proof of Lemma 7.3 .................................................... 279
  7.7.3 Computing an \((1 + \varepsilon)\)-Approximate Matching with High Probability 282

8 Conclusion ................................................................. 284
  8.1 Open Questions for Planar Graphs ................................. 287
  8.2 Open Questions for Minor-Free Graphs .......................... 288
  8.3 Open Questions for Geometric Matching ........................ 289
  8.4 Extensions to Flow Problems ....................................... 290
  8.5 Extensions to Non-Bipartite Graphs ................................. 290
# List of Figures

1.1 (a) A bipartite graph $G$. (b) The two possible maximum-cardinality matchings in $G$. .......................................................... 3

1.2 (a) A matching $M$ on a bipartite graph $G$. Edges of $M$ are solid and edges not in $M$ are dashed. Vertices of $B$ are on the left and vertices of $A$ are on the right. (b) The residual graph $\overrightarrow{G}_M$. (c) An augmenting path $P$ with respect to $M$, depicted both in the original graph (left) as well as the residual graph (right). (d) The resulting perfect matching $M' = M \oplus P$ after augmentation. $M'$ .......................................................... 5

1.3 (a) A complete bipartite graph $G$ formed from two point sets $A$ (crosses) and $B$ (squares), each of size 4. (b) An optimal Euclidean matching of $G$. (c) An optimal RMS matching of $G$. (d) An optimal bottleneck matching of $G$ . . 17

1.4 (a) A bipartite graph $G$. (b) The two possible maximum-cardinality matchings in $G$. .......................................................... 20

2.1 Comparison of execution times of various matching algorithms. Here $C$ is the largest cost edge; $\omega < 2.373$ is the exponent of matrix multiplication complexity. Our algorithm is the fastest for both $K_h$-minor free graphs (resp. planar graphs) if $C = n^{O(1)}$ and $C = \tilde{\Omega}(n^{0.23})$ (resp. $C = \tilde{\Omega}(n^{0.03})$). . . . . . . 32
2.2 An illustration of the process for converting an instances of the bottleneck matching problem into an input for the algorithm of Section 2.2. Left: The original set of input points. Middle: The sparse graph obtained from overlaying a fine grid and clustering vertices. Right: The pieces formed by using a coarse grid. Boundary vertex clusters are circled.

5.1 Transforming any bipartite graph to a constant degree graph

5.2 (a) A partition $\mathcal{R}_j$. The squares represent vertices of $B$ and the circles represent vertices of $A$. Free vertices are filled in. (b) The boundary-to-boundary edges of $H$ for $\mathcal{R}_j$. (c) The edges from $b_j$ to $\mathcal{K}_j$. (d) The edges from $\mathcal{K}_j$ to $a_j$. (e) There is a single edge from $b_j$ to $a_j$.

5.3 (a) A division of the boundary vertices of a partition into $O(\log n)$ bipartite groups. (b) Let $d(u, v)$ be the shortest path distance within the partition from $u$ to $v$. The shortest path from $v_i$ to $v'_j$ and the shortest path from $v'_i$ to $v_j$ must cross at a vertex $x$. (c) The Monge property within a bipartite group. $M_{i,j} + M_{i',j'} \leq M_{i',j} + M_{i,j'}$.

6.1 (a) A piece $\mathcal{R}_j$. The squares represent vertices of $B$ and the circles represent vertices of $A$. Filled vertices are free internal vertices. Specifically, the filled diamond represents an inactive free vertex, and the filled squares represent active free vertices. (b) The boundary-to-boundary edges and self-loop edges of $H$ for $\mathcal{R}_j$. (c) The edges from $b^A_j$ and $b^T_j$ to $\mathcal{K}_j$. (d) The edges from $\mathcal{K}_j$ to $a_j$. (e) There is a single edge from both $b^A_j$ and $b^T_j$ to $a_j$. 
6.2 The status of a piece in the \( r \)-clustering and its compressed representation before and after augmentation. By augmenting along a path containing \( \langle u, x, x', v \rangle \), a new path is created from \( \langle y, x', x, z \rangle \).

7.1 (a) A division of \( \Box \) into subcells. (b) Examples of a few possible WSPD pairs of \( W_\Box \). Every pair of subcells in different children of \( \Box \) would be represented by some pair.

7.2 (a) A set of local edges between a WSPD pair of cells. Solid edges are in the matching, and dashed edges are not. (b) A local non-matching edge from \( u \in B \) to \( v \in A \) implies the existence of a length 3 alternating path \( P = \langle u, u', v', v \rangle \) with net-cost \( \phi(P) = d_Q(u, v) \).

7.3 (a) The active cells of the current layer. Each full active cell (bold) has descendants in its active tree. (b) The pieces of a full active cell. (c) Each full piece has 4 descendants in the active tree. (d) Each branch of the active tree terminates with a sparse leaf cell.

7.4 (a) A local path \( S \) of length 3 between the cells \( \Psi \) and \( \Psi' \). Although \( S \) does not contain any non-local edges, augmentation will reduce the number of matching edges between the two cells by 1. (b) The number of matching edges between the cells \( \Psi \) and \( \Psi' \) can only increase when a local path containing at least one non-local edge between \( \Psi \) and \( \Psi' \) participates in an augmenting path.

7.5 Internal and boundary clusters of a subcell \( \xi \) within the child \( \Box' \) of \( \Box \). (a) All vertices matched within \( \Box' \) are part of an internal cluster. (b) All vertices matched outside of \( \Box' \) are divided into boundary clusters based on WSPD pairs at higher levels.
7.6 Examples of edges of $E_{\Box}$. Internal edges (solid) represent shortest paths between clusters within a child of $\Box$. Bridge edges (dashed) are between clusters in two different children of $\Box$. 

251
Chapter 1

Introduction

Graphs are mathematical constructs used to model relationships between objects. A graph \( G(V, E) \) consists of two sets: a set of vertices \( V \) and a set of edges \( E \subseteq V \times V \). Generally speaking, the set of vertices corresponds to a set of real-world objects, and the edges represent some sort of relationship between pairs of objects. For the scope of this document, we are interested in using graphs to model matching problems. Specifically, we are interested in developing faster algorithms for the maximum-cardinality matching problem as well as the minimum-cost maximum-cardinality matching problem.

As a simple motivating example, consider a scenario where an instructor teaching a class of students wishes to create pairs of students for a group project. However, it is clear that some students, for one reason or another, would not work well-together in a group. The instructor is interested in pairing off the students in a fashion so as to maximize the number of pairs of compatible students.

We can model this scenario as a graph matching problem as follows: Each student corresponds to a vertex. There is an edge present between any pair of students if and only if those students are compatible as a pair. The problem then is to select as many edges (i.e., compatible pairs of students) as possible such that no two edges share a vertex (i.e., no student is part of more than one pair).

More formally, a matching \( M \subseteq E \) is a set of vertex-disjoint edges of \( G \). Matchings corre-
spond to sets of pairs such that no object belongs to more than one pair. Perhaps the most fundamental matching problem is that of finding a maximum cardinality matching, a matching in \( G \) with the largest possible size. For examples of maximum cardinality matchings, see Figure 1.1. Often, it is useful to assign a cost \( c(u, v) \) to every edge \((u, v) \in E\) with a goal of simultaneously optimizing on both the size and cumulative cost of the computed matching. The problem of computing a maximum-cardinality matching \( M \) with minimum total cost \( c(M) = \sum_{(a, b) \in M} c(a, b) \) is known as the minimum-cost maximum-cardinality matching problem [18].

The goal of this work is the design of faster algorithms for solving these fundamental optimization problems. By ‘faster’ we mean in the sense of worst-case asymptotic running times (i.e., Big-O). Asymptotic running times are used to compare the execution times of algorithms for very large inputs. Asymptotic notations inherently ignore multiplicative and additive constants in the algorithms’ running times. While minimizing constants for specific applications is certainly important when designing practical implementations of algorithms, the use of asymptotic notations allows the algorithm designing to focus on more general questions. What makes this problem fundamentally hard to efficiently solve? How can we design techniques that address this fundamental hardness of the problem? The answers to these questions can help guide practitioners in the design of robust and scalable algorithms for a wide-variety of real-life inputs.

This work focuses on designing matching algorithms for bipartite graphs. A graph is bipartite if its vertex set can be divided into two disjoint subsets \( A \) and \( B \) such that all edges are in \( A \times B \). Equivalently, a graph is bipartite if and only if it contains no odd-length cycles. The absence of odd-length cycles makes designing matching algorithms for bipartite graphs significantly easier than for general (non-bipartite) graphs.

As an example application of matchings on bipartite graphs, consider the following organ
Figure 1.1: (a) A bipartite graph $G$. (b) The two possible maximum-cardinality matchings in $G$.

donation problem. We are given a set of donors $A$ who are willing to donate an organ (a kidney, for example) to someone else who is in-need. We are also given a set of potential recipients $B$, who are in-need of potentially life-saving organs. For any donor-recipient pair $(a, b) \in A \times B$, there is a possibility that donor’s organ is incompatible with the recipient, whether for biological reasons or logistical ones. Our first and foremost goal should be to save as many lives as possible by maximizing the number of compatible donor-recipient pairings. We can model this problem by forming a bipartite graph with vertex set $A \cup B$. There is an edge between any donor $a$ and any recipient $b$ if and only if they form a compatible pair. A maximum-cardinality matching on this bipartite graph would then maximize the number of compatible pairs.

In many cases, there are multiple maximum-cardinality matchings available. However, not all maximum-cardinality matchings are equal. For example, each compatible donor-recipient pair may also come with a monetary or other logistical cost $c(a, b)$. In addition to maximizing the number of compatible pairs, we should also seek to minimize the total cost of our matching; although, this minimization of cost would be only a secondary concern. This cost-minimizing variant of the organ donation problem can be modeled using the same bipartite graph described earlier, but with each edge $(a, b)$ having an assigned cost $c(a, b)$. 
Then, solving the minimum-cost maximum cardinality matching problem on the resulting weighted graph would give a maximum number of compatible pairings with minimum total cost. For more information on the relevance of bipartite matching to the organ donation problem, see [42].

The algorithms described in the current work are for solving matching problems on bipartite graphs. However, the techniques introduced by this work are not inherently limited to bipartite graphs; it is the author’s belief that the new algorithmic results described in this work can also be extended to the non-bipartite setting using the same framework, but the technical details of doing so are highly non-trivial. Standard matching algorithms for bipartite graphs and non-bipartite graphs have the same overall approaches, but the actual implementation of that approach is significantly more difficult for non-bipartite graphs. We discuss prospects for extending this work to non-bipartite graphs in section 8.5.

Given a matching $M$, we say a vertex is free if it is not incident on an edge of $M$. Otherwise, a vertex is matched. A path (resp. cycle) is alternating if it alternates between edges of $M$ and edges not in $M$. An augmenting path with respect to $M$ is any alternating path that begins and ends at a free vertex. Augmenting paths are significant in that they allow us to obtain a matching $M'$ with one more edge than $M$. Specifically, given a matching $M$ and augmenting path $P$ with respect to $M$, a matching $M'$ can be obtained by removing from $M$ all matching edges of $P$ and adding to $M$ all non-matching edges of $P$. We recall that $\oplus$ is the symmetric difference operator. If we let $M' \leftarrow M \oplus P$, then $M'$ is a matching with $|M'| = |M| + 1$. This process is called augmentation. See [18]. This ability to find matchings of larger size makes augmenting paths a foundational component of standard combinatorial matching algorithms, and many approaches focus on how augmenting paths can be found efficiently.

When dealing with bipartite graphs, the set of alternating paths can be easily represented
by assigning directions to the edges based on whether they are in the matching $M$ or not. Given a bipartite graph $G(A \cup B, E)$, where $E \subseteq A \times B$, and a matching $M$ on $G$, we obtain a directed residual graph $\overrightarrow{G}_M$ as follows: The vertex set of $\overrightarrow{G}_M$ is $A \cup B$. Every edge of $E$ is assigned a direction in $\overrightarrow{G}_M$. Any edge in $M$ is directed from $A$ to $B$, and any edge not in $M$ is directed from $B$ to $A$. Any directed path in $\overrightarrow{G}_M$ is an alternating path in $G$ with respect to $M$. For examples of a residual graph, an augmenting path, and augmentation, see Figure 1.2. When the matching $M$ is obvious from the context, we may omit the matching $M$ from the residual graph’s notation, using $\overrightarrow{G}$ instead. Note that a directed residual graph has no straight-forward extension to non-bipartite graphs.

![Figure 1.2](image)

(a) A matching $M$ on a bipartite graph $G$. Edges of $M$ are solid and edges not in $M$ are dashed. Vertices of $B$ are on the left and vertices of $A$ are on the right. (b) The residual graph $\overrightarrow{G}_M$. (c) An augmenting path $P$ with respect to $M$, depicted both in the original graph (left) as well as the residual graph (right). (d) The resulting perfect matching $M' = M \oplus P$ after augmentation. $M'$.

All algorithms discussed take a graph (either implicitly or explicitly) as input. Unless otherwise stated, we use $n$ to denote the number of vertices in the graph and $m$ to denote the number of edges.

The current ‘state of the art’ matching algorithms for arbitrary (bipartite or non-bipartite) graphs with $m$ edges and $n$ vertices have running times of $\Omega(m\sqrt{n})$ (see [17], [18], [22]). While there are other algorithms that offer better tradeoffs for sparse or dense graphs, they are
complex and impractical to implement; these approaches are discussed briefly in Section 1.1.

Breaking the $\Omega(m\sqrt{n})$ barrier remains an important (and difficult) open question. Instead, of considering arbitrary graphs, we ask whether the state of the art matching algorithms can be improved for specific classes of graphs. Such improvements may potentially offer insights into matching problems on arbitrary graphs as well.

For the scope of this work, we consider classes of graphs with good separators. Roughly speaking, a graph $G(V, E)$ has a good separator if it is possible to select some relatively small subset of vertices whose removal disconnects the graph into two roughly-equal-sized pieces. More formally, given a graph $G(V, E)$, a separator is any subset of vertices $S \subseteq V$ whose removal disconnects the graph into two disjoint subgraphs. These subgraphs correspond to the vertex-induced subgraphs of the vertex sets $V_1$ and $V_2$, where $V = V_1 \cup S \cup V_2$ and there are no edges in $V_1 \times V_2$. A separator $S$ is balanced if both $V_1$ and $V_2$ have size at least $O(|V|/c)$ for some constant $c \geq 2$. A separator is small if the size of the set $S$ is $O(n^\delta)$ for some constant $0 < \delta < 1$. A separator is good if it is both small and balanced. A graph $G$ has good separators if, for every subgraph $G'$ of $G$, it is possible to efficiently compute a good separator on $G'$. This work seeks to answer the following question:

(Q1) Can we design a framework for developing asymptotically faster combinatorial matching algorithms on graphs with good separators?

Many common classes of graphs admit good separators. The most notable example is the class of planar graphs, which contains the set of all graphs that can be drawn in a plane (or on a sphere) without two edges crossing. A graph $H$ is a minor of $G$ if $H$ can be obtained via vertex/edge deletions and/or edge contractions in $G$. A graph is planar if and only if it does not include the complete graph $K_5$ or the complete bipartite graph $K_{3,3}$ as a minor. It was shown by Lipton and Tarjan [37] that planar graphs admit an $O(\sqrt{n})$-sized separator.
that can be computed in $O(n)$ time.

Another type of graph that admits good separators is the class of $K_h$-minor-free graphs, which contains all graphs without the complete graph $K_h$ as a minor. For a fixed constant $h$, it is possible to compute a separator of size $\tilde{O}(\sqrt{n})$ on any $K_h$-minor-free graph [25, 59]. Note that we use the $\tilde{O}(f(n))$ notation to suppress factors of $\log^c(n)$ for any constant $c$. There are also several matching-related problems in geometric settings that admit separator-like properties.

The existence of good recursively-computable separators lends itself to divide-and-conquer strategies for a variety of problems. For graphs with good separators of size at most $n^\delta$, Lipton and Tarjan [38] gave a simple divide and conquer matching algorithm for the minimum-cost maximum cardinality bipartite matching problem with running time $O(mn^\delta \log n)$. For classes of graphs where $\delta < 1/2$, this algorithm improves upon the $\Omega(m\sqrt{n})$ bound. There are also faster setting-specific matching algorithms for planar graphs and minor-free graphs. We focus on the design of a new framework for matching problems on graphs with good separators that can be used to improve upon these existing methods.

Many applications of minimum-cost matching have costs assigned using geometric distances. For example, a taxi-service, in seeking to minimize customer wait times, may want to find a minimum-cost matching between a set of taxis $A$ and a set of customers $B$. In this case, it may be possible to assign any taxi $a \in A$ to any customer $b \in B$, but some assignments would require a longer customer wait time based on the distance between the taxi and the customer. This wait time can be approximated by using geometric costs, such as by using the $L_1$ distance, also known as the Manhattan distance. We then seek to allocate taxis to customers with minimum total wait-time (i.e., cost). This problem can be solved by finding a minimum-cost matching on a complete bipartite graph. We call such problems, where the goal is to compute a minimum-cost matching on a complete bipartite graph with geometric
costs, geometric matching problems.

At first glance, it may not be clear how the goal of this work, as described in (Q1), is at all applicable to geometric matching problems. After all, geometric matching problems involve the use of a complete bipartite graph, which clearly has no good separators. However, geometric graphs often have separator-like properties. For example, if we were to partition the city into carefully chosen regions (using, for example, a grid), then we might observe that the optimal matching is more likely to contain taxi-customer pairs within a region than it is to contain taxi-customer pairs between two different regions. There has been some prior work (for example [51, 56]) that exploits this connection between geometric matching and graph separators to some extent. However, it is reasonable to ask whether the separator-based framework, mentioned in (Q1), could be applied to yield yet further improvements in geometric settings. This prompts the second question of the current work:

(Q2) Can we use the framework from (Q1) to develop faster algorithms for geometric matching problems?

In an effort to answer (Q1) and (Q2), we develop a framework and demonstrate how to apply it to yield the following results:

- We give an $O(mn^{\delta/(1+\delta)})$ time algorithm for computing maximum-cardinality matchings on classes of bipartite graphs with efficiently and recursively computable balanced separators of size $O(n^\delta)$, improving upon the running time of the Hopcroft-Karp algorithm for any bipartite graph with recursively-computable balanced separators of sub-linear size. The techniques for producing this result are summarized in Section 2.2, and a full, detailed explanation of the result can be found in Chapter 4. A preliminary version of this work appeared in the Symposium on Computational Geometry (SOCG) [34] conference. This work has also been invited to appear in the Journal of
Computational Geometry (JO CG).

- We give an algorithm that computes a minimum-cost perfect matching on a bipartite planar graph in $\tilde{O}(n^{4/3} \log(nC))$ time. Prior to our work, all combinatorial algorithms had execution times of $\Omega(n^{3/2})$. We summarize this result in Section 2.3 and give a full explanation of the paper in Chapter 5. A preliminary version of this work appeared in the Symposium on Discrete Algorithms (SODA) conference [5] as well as the ACM Transactions on Algorithms (TALG) journal [6].

- We give an algorithm that computes a minimum-cost maximum-cardinality matching on a bipartite $K_h$-minor free graph in $\tilde{O}(n^{7/5} \log(nC))$ time. Prior to our work, all combinatorial algorithms had execution times of $\Omega(n^{3/2})$. We also extend the new techniques introduced by this algorithm to the context of planar graphs, improving the previously mentioned $\tilde{O}(n^{4/3} \log(nC))$ time result for minimum-cost perfect matching in bipartite planar graphs to yield an $\tilde{O}(n^{6/5} \log(nC))$ running time. We summarize the techniques introduced by this result in Section 2.4 and give a full explanation in Chapter 6. A preliminary version of this work appeared in the SODA conference [33].

- We apply the previously mentioned separator-based algorithm of [34] to an application in geometric matching, the bottleneck matching problem. In the bottleneck matching problem, the goal is to compute a perfect matching between the input point sets that minimizes the length of the longest edge. We give an $\tilde{O}(n^{4/3} \text{poly}(1/\varepsilon))$ time algorithm for computing a $(1 + \varepsilon)$-approximate bipartite bottleneck matching for 2-dimensional point sets. This result also extends to arbitrary dimension, yielding a
$\frac{1}{\epsilon}\log n^{1+\frac{d-1}{2d-1}} \text{ time algorithm for computing an } (1+\epsilon)\text{-approximate bipartite bottleneck matching between two point sets } A, B \subset \mathbb{R}^d. \text{ Prior to our work, all algorithms for } d > 1 \text{ had running times of } \Omega(n^{3/2}). \text{ We summarize the techniques used by this result in Section 2.5. A full explanation can be found in Section 4.5. This result appeared, alongside the previously mentioned maximum-cardinality matching result, in the SOCG conference paper [34], which was invited to the JOCG journal.}

- We apply our framework to a second result in geometric matching, the root-mean-squared (RMS) matching problem. In this geometric matching problem, the cost of each edge is the square of its Euclidean length. Given two point sets $A, B \subset \mathbb{R}^2$ each containing $n$ points, we give an algorithm that computes a $(1+\epsilon)$-approximate RMS matching in time $\tilde{O}(n^{5/4}\text{poly}(1/\epsilon))$. Prior to this work, all other $(1+\epsilon)$ approximations algorithms took $\Omega(n^{3/2})$ time. We summarize the techniques of this result in Section 2.6 and give a detailed description of the result in Chapter 7. This result is currently in submission to the Symposium on Foundations of Computer Science (FOCS) conference.

The rest of the document is organized as follows: In Section 1.1, we give a literature review of matching algorithms in a variety of graph settings. In Chapter 2, we describe our separator-based framework and summarize the specific techniques used in each of our results. In Chapter 3, we give a detailed description of algorithmic background information that is necessary to understand the technical details of our results. Finally, in Chapters 4, 5, 6, and 7 we give full descriptions of the these results. We conclude in Chapter 8, with some closing remarks, including directions of future work.
1.1 Prior Work

In this section, we give a summary of prior results for the maximum-cardinality matching problem and the minimum-cost maximum cardinality matching problem. We begin with a summary of results for arbitrary graphs (bipartite and non-bipartite). We follow with a summary of results for classes of graphs with good separators. Finally, we discuss prior algorithmic results for geometric matching problems.

1.1.1 Arbitrary Graphs

Perhaps the most fundamental approach for finding a maximum cardinality matching on a bipartite graph is the Ford-Fulkerson algorithm [16] [1956]. The Ford-Fulkerson algorithm is actually designed to compute a maximum flow, but there is a simple reduction from the maximum-cardinality bipartite matching problem to the maximum-flow problem. Therefore, the Ford-Fulkerson maximum-flow algorithm has a natural interpretation in the context of matching; we focus on this matching interpretation. The Ford-Fulkerson algorithm finds a maximum-cardinality matching on a bipartite graph by iteratively computing an augmenting path. Through the process of augmentation, an augmenting path can be used to increase the size of the current matching by 1. The algorithm executes $O(n)$ iterations, where each iteration executes an $O(m)$ time search to find a single augmenting path, leading to a total execution time of $O(mn)$.

The Ford-Fulkerson algorithm can be improved by observing that it is possible to find multiple disjoint augmenting paths per iteration. Hopcroft and Karp [1973] [22] showed that, if a maximal set of vertex-disjoint shortest augmenting paths is computed in each iteration, then the number of iterations reduces from $O(n)$ to $O(\sqrt{n})$. They went on to provide an algorithm that executes each iteration in $O(m)$ time on bipartite graphs. Even after the
passage of nearly five decades, no further combinatorial improvements to the Hopcroft-Karp (HK) algorithm have been found.

The Ford-Fulkerson and Hopcroft-Karp maximum-cardinality matching algorithms have natural analogues in the weighted setting. The ‘Hungarian’ algorithm for minimum-cost maximum cardinality bipartite matching was presented by Kuhn and Munkres in 1956 [30], although other earlier solutions were known. The Hungarian algorithm exhibits a very similar structure to the Ford-Fulkerson algorithm in that it executes $O(n)$ iterations where each iteration executes a search to find a single augmenting path. Specifically each iteration finds a single augmenting path of minimum ‘cost’ (for an appropriately defined cost). This can be done using Dijkstra’s algorithm in $O(m + n \log n)$ time per iteration, for a total time of $O(mn + n^2 \log n)$. The Hungarian algorithm remains the best strongly-polynomial time algorithm for minimum-cost maximum-cardinality matching.

When the input bipartite graph has non-negative integer edge costs at most $C$, it is possible to improve the Hungarian algorithm’s dependency on $n$ while incurring only an additional logarithmic dependency on $C$. For such graphs, Gabow and Tarjan [1989] [18] gave a bit-scaling algorithm with a running time of $O(m \sqrt{n} \log(nC))$. The Gabow-Tarjan (GT) algorithm executes $O(\log nC)$ scales, where the algorithm for each scale can be seen as a generalization of the Hopcroft-Karp algorithm to the weighted setting; each iteration of a scale computes a maximal set of vertex-disjoint minimum ‘cost’ augmenting paths. The Hopcroft-Karp and Gabow-Tarjan algorithms together form the foundation of much of our work. Detailed explanations of the Ford-Fulkerson, Hopcroft-Karp, Hungarian, and Gabow-Tarjan algorithms can be found in Chapter 3.

Each of the four discussed algorithms for bipartite matching have analogues for non-bipartite graphs. For non-bipartite graphs, Edmonds gave an $O(mn)$ time algorithm for maximum-cardinality matching [13]. After several years, Micali and Vazaran gave an improved algo-
1.1. Prior Work

Algorithm with a running time of $O(m\sqrt{n})$ [41], although their original proof contained non-trivial bugs, which were later found and fixed. Other $O(m\sqrt{n})$ time algorithms followed; recently, a much simpler approach was presented by Gabow that also achieves the $O(m\sqrt{n})$ bound [17]. Gabow and Tarjan gave an $O(m\sqrt{n}a(m, n)\log n \log(nC))$ time algorithm for minimum-cost maximum-cardinality matching on non-bipartite graphs using similar approaches to their earlier algorithm for bipartite graphs [19]. Here, $a(\cdot, \cdot)$ is the incredibly slow-growing inverse Ackermann function, which is the amortized complexity per operation of the most efficient union-find data structure [53]. Each of these algorithms for non-bipartite graphs have a similar overall structure to their bipartite counterparts, but the implementation details and proofs for the algorithms within each iteration are significantly more complex for non-bipartite graphs. As a result, advances in non-bipartite graphs tend to lag years behind the respective advancements in bipartite graphs.

We have summarized the fundamental augmenting-path-based combinatorial matching algorithms, although there are recent parallel lines of work that give other approaches. These other approaches tend to fall into two categories. First, there are the electric flow algorithms that have spawned from recent work by Madry [2013] [40]. This work gave an algorithm that computes a maximum flow on a network with unit capacities in $\tilde{O}(m^{10/7})$ time. Through well-known reductions, this gives an $\tilde{O}(m^{10/7})$ time algorithm for maximum-cardinality matching, improving upon the running time of the Hopcroft-Karp algorithm when the graph is sufficiently sparse. More recently, Cohen et al. [10] generalized the ideas of Madry’s algorithm to work for the minimum-cost flow problem on unit-capacity graphs, yielding an $\tilde{O}(m^{10/7} \log C)$ running time, where $C$ is the largest edge cost. Once again, using known reductions, this yields an $\tilde{O}(m^{10/7} \log C)$ time algorithm for minimum-cost maximum-cardinality matching. Note that these electric flow algorithms are inherently limited to computing matchings on bipartite graphs, since non-bipartite matching problems cannot be reduced to flow problems.
There is another line of work involving the use of fast matrix multiplication for solving matching problems. In 2004, Mucha and Sankowski gave an approach that computes a maximum cardinality matching on an arbitrary graph (not necessarily bipartite) in $\tilde{O}(n^\omega)$ time [43]. Here, $\omega$ is the running time exponent of the fastest algorithm for multiplying together two $n \times n$ matrices. There has been a long line of improvements on the value of $\omega$; currently, it is known that $\omega < 2.372873$ [58]. This implies that the algorithm of Mucha and Sankowski runs asymptotically faster than the Hopcroft-Karp algorithm for sufficiently dense graphs. Later, Sankowski extended this approach to the weighted setting, giving an $\tilde{O}(n^{\omega}C)$ time algorithm for computing a minimum-cost perfect matching on graphs with maximum edge weight $C$ [47]. This improves upon the Gabow-Tarjan algorithm when $C$ is a sufficiently small polynomial in $n$ and the graph is sufficiently dense. Note that these algorithms are randomized and are not guaranteed to obtain a correct solution. Due to high constants, they are completely impractical from an implementation standpoint.

1.1.2 Planar Graphs

It is well known that for planar graphs, $m = O(n)$. Therefore, using the algorithms for arbitrary graphs, a maximum cardinality matching can be found in $O(n^{3/2})$ time, and a minimum-cost maximum-cardinality matching can be found in $\tilde{O}(n^{3/2} \log nC)$ time. For bipartite graphs, unweighted (resp. weighted) matching can be solved in $\tilde{O}(n^{10/7})$ (resp. $\tilde{O}(n^{10/7} \log C)$) time using electric flow algorithms. Mucha and Sankowski extended their matrix-multiplication approach to yield an $\tilde{O}(n^{\omega/2})$ (resp. $\tilde{O}(n^{\omega/2}C)$) time algorithm for maximum-cardinality matching in unweighted (resp. weighted) planar graphs.

A key technique in the design of many planar graph algorithms is the use of planar separators. Given any planar graph, Lipton and Tarjan gave an algorithm that computes in $O(n)$ time.

---

4 We recall that a perfect matching is one that matches all vertices.
1.1. Prior Work

A set of separator vertices of size $O(\sqrt{n})$ whose removal disconnects the graph into two parts each having size at most $2n/3$ [37]. Using straight-forward generalizations of the Ford-Fulkerson and Hungarian algorithms, this in turn yields an $O(n^{3/2})$ time algorithm for maximum-cardinality matching and an $O(n^{3/2} \log n)$ time algorithm for minimum-cost maximum-cardinality matching on planar graphs. Note that both these results are strongly polynomial. There has been a line of work towards using separators to more efficiently compute maximum matchings and flows on planar graphs, which resulted in an $O(n \log^3 n)$ time algorithm for computing planar maximum flow with multiple sources and multiple sinks [9]. This, in turn, provides a near-linear time algorithm for the maximum-cardinality matching problem on bipartite planar graphs. However, these approaches for maximum flow and maximum matching have no clear extension to the weighted setting. In fact, until our recent work (described in Chapters 5 and 6), the fastest algorithms\footnote{Assuming logarithmic dependency on $C$.} for minimum-cost matching on planar graphs were those for arbitrary graphs. As of now, the current best result for weighted matching on planar graphs computes a minimum-cost perfect matching in $\tilde{O}(n^{6/5} \log n C)$ time [33].

1.1.3 Minor-Free Graphs

On $K_h$-minor free graphs, there are comparable separator theorems to those for planar graphs. One of the most notable results, by Wulff-Nilsen, showed that it is possible to compute a separator of size $O(h\sqrt{n \log n})$ in $O(\text{poly}(h)n^{5/4+\varepsilon})$ time, for an arbitrarily small positive value of $\varepsilon$ [59]. For values of $h$ that are constant or poly-logarithmic in $n$, the separator size nearly matches that of Lipton and Tarjan’s separator theorem for planar graphs. Furthermore, it can be shown that for such values of $h$, the number of edges $m = \tilde{O}(n)$ is near-linear in the number of vertices. As a result, the presence of such a separator im-
mediate implies that Lipton and Tarjan’s previously mentioned separator-based matching approach [38] can be applied, yielding an $\tilde{O}(n^{3/2})$ time algorithm for both unweighted and weight matching problems.

For $h = \tilde{O}(1)$, the fact that $m = \tilde{O}(n)$ implies that the electric flow algorithms also are applicable to $K_h$-minor free graphs, giving an $\tilde{O}(m^{10/7})$ time algorithm for unweighted matching and an $\tilde{O}(m^{10/7}\log C)$ time algorithm weighted matching. There are also matrix-multiplication approaches for minor-free matching. Using his previously mentioned separator theorem, Wulff-Nilsen shows that a maximum-cardinality matching on $K_h$ minor-free graphs can be found in $\tilde{O}(\text{poly}(h)n^{1.239})$ time [59]. The dependency on $n$ can be improved slightly to $\tilde{O}(n^{1.188})$ if a different separator theorem by Kawarabayashi and Reed is used, but it comes at the cost of a hyper-exponential dependency on $h$ [25, 59]. For weighted matching, the best known matrix-multiplication result is given by the $\tilde{O}(n^{\omega/2}C)$ algorithm of [47], as was the case for planar graphs. Once again, all of these matrix-multiplication approaches are randomized and impractical to implement if the asymptotically ‘fastest’ matrix multiplication algorithms are used. Recently, we produced a combinatorial algorithm, based on the Gabow-Tarjan algorithm, that breaks the $\Omega(n^{1.5})$ barrier while ensuring only a logarithmic dependency on $C$; this $\tilde{O}(n^{7/5}\log(nC))$ time algorithm is discussed in detail in Chapter 6.

1.1.4 Geometric Graphs

There are a variety of matching-related problems in geometric settings, way too many to be fully discussed here. We focus on a select few problems which are applicable to the current work. The quintessential Euclidean matching problem can be defined as follows: The set of vertices $V$ corresponds to points in $\mathbb{R}^d$, and there is an edge between every pair of vertices, i.e., the graph is complete. The cost of each edge is given by the Euclidean distance between
its endpoints. The problem is then to compute the minimum-cost perfect matching in the resulting graph.\footnote{Assume that \( n \) is even; thus, a perfect matching exists.} A related problem is the \textit{Euclidean bipartite matching} problem, which differs from its non-bipartite counterpart in that the vertex set is given by the union of two disjoint sets \( A \) and \( B \), with \(|A| = |B| = n\), and the edge set is given by \( A \times B \), i.e., the graph is a complete bipartite graph. Once again, the edge costs are assigned based on Euclidean distance and the goal is to compute a minimum-cost perfect matching.

![Figure 1.3: (a) A complete bipartite graph \( G \) formed from two point sets \( A \) (crosses) and \( B \) (squares), each of size 4. (b) An optimal Euclidean matching of \( G \). (c) An optimal RMS matching of \( G \). (d) An optimal bottleneck matching of \( G \).](image)

A variety of related geometric matching problems can be produced by modifying the definition of cost. One example is geometric matching under root-mean squared (RMS) distance, where the cost assigned to each edge is assigned as the square of the Euclidean distance between its endpoints, and the goal is to compute a perfect matching of minimum cost. Once again, this problem can be defined using both bipartite and non-bipartite variants. Matching under RMS distance is significantly harder than matching under Euclidean distance because the costs associated with the edges are not metric in that they do not obey the triangle inequality. As a result, many techniques for the Euclidean matching problems do not extend to RMS matching.
Finally, there is the bottleneck matching problem. This problem is similar to the Euclidean matching problem in that all edge costs are given by the Euclidean distance between their endpoints. However, instead of minimizing the sum of the edge costs in the computed matching, the goal is to compute a perfect matching that minimizes the maximum edge cost. The optimal value for this maximum edge cost is known as the bottleneck distance. If the bottleneck distance is known, the problem of computing a bottleneck matching reduces to a maximum-cardinality matching problem on an unweighted unit disk graph, a graph where two points are connected only if the distance between them is within some unit radius. Furthermore, by using a binary-search-like technique, the bottleneck distance can be guessed using roughly $O(\log n)$ attempts. As a result, bottleneck matching effectively reduces to an unweighted matching problem with geometric properties. Using the Hopcroft-Karp algorithm [22], it is possible to compute an exact bottleneck matching in $\tilde{O}(n^{2.5})$ time and a $(1+\varepsilon)$-approximate bipartite bottleneck matching in $\tilde{O}(n^{3/2}\text{poly}(1/\varepsilon))$ time. Efrat et al. [14] gave an $\tilde{O}(n^{3/2})$ algorithm for computing an exact bipartite bottleneck matching.

We note that in all aforementioned geometric problems, the number of edges is $\Theta(n^2)$, implying that direct applications of standard augmenting-path-based combinatorial algorithms have running times of $\Omega(n^{2.5})$. However, it is possible to use dynamic weighted nearest neighbor data structures to find a single augmenting path in time near-linear in $n$. As a result, there are solutions that exactly solve geometric matching problems in roughly $\tilde{O}(n^{3/2})$ time [14, 50].

Obtaining any exact $o(n^{3/2})$ time algorithm for any geometric matching problem remains an open question. However, it is known how to overcome the $\Omega(n^{3/2})$ barrier if we allow for approximate solutions. We say a solution is an $(1+\varepsilon)$-approximation if it has cost at most $(1+\varepsilon)$ times the optimal cost for the respective problem. After a long line of work, there are now multiple results for Euclidean costs that produce an $(1+\varepsilon)$-approximate matching in time
1.2. Applications

\(\tilde{O}(n \text{poly}(1/\varepsilon))\); see, for example, \([26, 51]\). However, for the \((1 + \varepsilon)\)-approximate bottleneck and RMS matching problems, breaking the \(n^{3/2}\) barrier was still an open question until the current work.

There is good reason to believe that geometric matching problems admit separator-like properties. A good example can be seen in the \(\tilde{O}(n^{3/2})\) time divide-and-conquer algorithm of Varadarajan for the Euclidean matching problem \([56]\). One goal of this work is to apply our separator-based framework to geometric matching problems in order to break the \(\Omega(n^{3/2})\) barrier. We show that our framework yields an \(o(n^{3/2})\) time algorithm for solving the \((1 + \varepsilon)\)-approximate bipartite bottleneck matching problem for any constant dimension \([34]\). This bipartite bottleneck matching algorithm is described in Chapter 4. We have also extended this framework to the \((1 + \varepsilon)\)-approximate RMS matching problem, yielding an \(\tilde{O}(n^{5/4} \text{poly}(1/\varepsilon))\) time algorithm. This RMS matching algorithm is described in Chapter 7.

1.2 Applications

A simple example of a maximum-cardinality matching problem on a planar bipartite graph is the \textit{domino tiling} problem, where we are given a grid, with some cells filled, and a set of dominos each of which occupy \(1 \times 2\) grid cells. The objective is to figure out if there is some way to place dominos on the grid so that all empty grid cells are filled. Here, each empty grid cell would correspond to a vertex, any pair of orthogonally adjacent grid cells would form an edge, and an edge is in the current matching if there is a domino covering its two adjacent grid cells. There is a way to tile the grid with dominos if and only if the graph has a perfect matching. See Figure 1.4.

While real-life graphs may not exactly fall into a theoretical class of graphs with good separators, it is often possible to compute good separators using heuristics. For example,
the existence of good separators on road-networks has been investigated. The work of [49] gave an algorithm for recursively computing separators on road networks, and demonstrated their approach on the United States road network. On this network, consisting of roughly 24 million vertices and 29 million edges, they were able to, by removing just 1,413 edges, split the graph into 27 disjoint pieces with the largest piece having less than 6 percent of the original graph’s size. Note that this also implies the presence of a vertex separator with similar properties. Another study [57] analyzed the ability to partition an Internet network, modeled using 36,715 vertices and 99,852 edges, and found that the removal of roughly 1,000 vertices was sufficient to divide the graph into pieces, each containing at most 1/2 of the vertices.

In many applications of minimum-cost bipartite matching, the input graph $G$ is complete and, therefore, does not explicitly have good separators. However, as the current work will demonstrate (see Sections 2.5 and 2.6), when the underlying edge costs are geometric costs, it is sometimes possible to find a suitable modification $G'$ of the input graph such that (i) an optimal matching in $G'$ approximates the optimal matching in the original graph $G$, and (ii) $G'$ has good separators. As a result, the input graph $G$ has implicit separators. Therefore, there are several applications of this work in geometric matching problems as well.

In many applications, the input graph is a complete bipartite graph, but the costs are given by a shortest path metric on a graph $G$ with good separators. For example, given a set $B$
of supply locations and a set $A$ of demand locations, the problem of efficiently transporting goods from the supply locations to the demand locations can be modeled as a complete bipartite graph $G(A \cup B, A \times B)$ where the cost between any supply center $a \in A$ and any demand center $b \in B$ is given by the shortest path from $a$ to $b$ along a road network. While the bipartite graph itself is complete, and, therefore, has no good separators, the underlying road network does have good separators. Our current work does not explicitly address such situations, but there is some possibility that our approaches for geometric matching could to these settings.

Many applications work with a suitable generalization of minimum-cost bipartite matching. A good example of such a generalization is the problem of computing an optimal transport between two discrete probability distributions $A$ and $B$. Both sets $A$ and $B$ consist of $n$ points. Each point $a \in A$ has a non-negative demand $d_a$ and each point $b \in B$ has a non-negative supply $s_b$, where $\sum_{a \in A} d_a = \sum_{b \in B} s_b = 1$. The problem also takes as input an $n \times n$ matrix of costs $c(\cdot, \cdot)$ defined on all edges of $A \times B$. A transport plan $\sigma : A \times B \to \mathbb{R}_{\geq 0}$ is a function that describes how to move the probability mass at the supply points of $B$ to the demand points at $A$, without violating the supply or demand constraints: for every $a \in A$, $\sum_{b \in B} \sigma(a, b) = d_a$ and for every $b \in B$, $\sum_{a \in A} \sigma(a, b) = s_b$. The cost $c(\sigma)$ of the transport plan $\sigma$ is given by $\sum_{(a,b) \in A \times B} \sigma(a,b)c(a,b)$, and the goal of the problem is to compute a minimum-cost transport plan $\sigma^*$. When, for some metric distance $d(\cdot, \cdot)$ and some integer $p > 1$, the cost $c(a, b)$ between any pair of points $(a, b) \in A \times B$ is given by $d(a, b)^p$, the value $c(\sigma^*)^{1/p}$ is known as the $p$-Wasserstein distance. Wasserstein distances have found numerous applications as a measure of similarity between objects, such as images [29], 3D models [52], news articles [31], and musical scores [54]. They have also been used as an objective function for Generative Adversarial Neural Networks (GANs) [3, 20, 39], which are used to generate fake images. There are also applications of optimal transport in biology for
changes in cellular states over time [48].

When the input distributions are uniform, i.e., each vertex has a supply or demand of $1/n$, the problem of computing an optimal transport can be solved by finding a minimum-cost perfect matching on the complete bipartite graph $G(A \cup B, A \times B)$. As a result, the optimal transport (or $p$-Wasserstein distance) can be computed using algorithms for minimum-cost bipartite matching. When the ground distance metric $d(\cdot, \cdot)$ between points is given by Euclidean distance, Wasserstein distance is given by solving a corresponding geometric matching problem. The solution for $p = 1$ is given by the optimal Euclidean bipartite matching, the solution for $p = 2$ is given by the optimal RMS matching, and the solution for $p = \infty$ is given by the optimal bottleneck matching.

Fast computation of the optimal transport is a topic of high interest in the machine-learning community. However, exact algorithms are often too slow, and approximation algorithms are used in practice [2, 11, 12, 36, 45]. These approximation algorithms focus on the design of additive approximations, which are weaker than $(1 + \varepsilon)$ relative approximations but much more efficiently computable. Recently, Lahn et al. showed that the Gabow-Tarjan algorithm for the minimum-cost transportation problem [55] can be used to compute a transport plan with additive error at most $\delta$ away from the optimal in time $O(n^2(C/\delta) + n(C/\delta)^2)$, where $C$ is the largest edge cost. Given that the techniques of the current work are also based on the approaches of Gabow and Tarjan, the current work has the potential to lead to improved practical approximation algorithms for computing Wasserstein distances, especially when the ground metric is Euclidean.
Chapter 2

Our Framework and Results

In this section, we outline the set of techniques that form our framework, as well as how these techniques can be used to improve matching algorithms for planar graphs, minor-free graphs, and geometric graphs. We start with a high-level description of the framework (see Section 2.1). Then, we provide more detailed overview of each of these five results (see Sections 2.2–2.6).

2.1 The Framework

As part of this work, we have compiled a general set of techniques that are applicable towards addressing (Q1) and (Q2). We have demonstrated the effectiveness of this framework by applying it to matching problems on planar graphs, minor-free graphs, and geometric graphs. These preliminary results use some or all of the following ideas:

- The methods used are combinatorial and based on computing augmenting paths. The basic principles of our algorithms strongly coincide with those of some of the most fundamental matching algorithms. A detailed description of these matching algorithms is provided in Chapter 3.

- By recursively using separators, the input graph can be decomposed into ‘pieces’ of relatively small size, with each piece having a relatively small number of ‘boundary’
vertices. In order for a path to cross between two pieces, it must use one of the boundary vertices.

- As a preprocessing step, we execute a matching algorithm in order to match the majority of the vertices in the graph. For example, in the context of unweighted matching, after fully executing the Hopcroft-Karp algorithm within each piece, it can be shown that the number of unmatched vertices is proportional to the total number of boundary vertices. By localizing the preprocessing step within each piece, it can be shown that the total time taken by the preprocessing step is small.

- To match the remaining vertices, the algorithms use the set of boundary vertices between pieces to compute augmenting paths efficiently. The augmenting paths computed may span multiple pieces; such pieces are affected by the augmenting path. Augmentation causes changes to the matching in all affected pieces. Accounting for these changes incurs an additional running-time cost proportional to the number of affected pieces.

- To bound the efficiency of our algorithms, it is critical to argue that the number of affected pieces is relatively small. We introduce problem-specific weighting schemes that penalize boundary crossings by assigning them a higher cost. The additional costs added to the boundaries between pieces ensure that the augmenting paths computed intersect relatively few pieces.

- By finding multiple augmenting paths per iteration, it is often possible to greatly reduce the number of iterations required. Traditional techniques that find multiple augmenting paths per iteration converge to a maximum matching in $O(\sqrt{n})$ iterations (see 3.2 and 3.4). We introduce techniques that reduce the number of iterations to be roughly proportional to the square-root of the number of boundary vertices.
• In some situations, we can exploit setting-specific data structures to execute each iteration of the algorithm faster, yielding further improvements.

2.2 Separator-Based Maximum Matching Algorithm

2.2.1 Our Results

We present a weighted approach to compute a maximum cardinality matching in an arbitrary bipartite graph. Our main result is a new matching algorithm that takes as input a weighted bipartite graph $G(A \cup B, E)$ with every edge having a weight of 0 or 1. Let $w \leq n$ be an upper bound on the weight of any matching in $G$. Consider the subgraph induced by all the edges of $G$ with a weight 0. Let $\{K_1, K_2, \ldots, K_l\}$ be the connected components in this subgraph and let, for any $1 \leq i \leq l$, $V_i$ and $E_i$ be the vertices and edges of $K_i$. We refer to each connected component $K_i$ as a piece. Suppose $|V_i| = O(r)$ and $|E_i| = O(mr/n)$. Given $G$, we present an algorithm to compute a maximum matching in $G$ in $\tilde{O}(m(\sqrt{w} + \sqrt{r} + \frac{wr}{n}))$ time. Consider any graph in which removal of sub-linear number of separator vertices partitions the graph into connected components with $O(r)$ vertices and $O(mr/n)$ edges. We can apply our algorithm to any such graph by simply setting the weight of every edge incident on any separator vertex to 1 and weights of all other edges to 0.

When all the edge weights are 1 or all edge weights are 0, our algorithm will be identical to the HK-Algorithm algorithm and runs in $O(m\sqrt{n})$ time. However, if we can carefully assign weights of 0 and 1 on the edges such that both $w$ and $r$ are sub-linear in $n$ and for some constant $\gamma < 3/2$, $wr = O(n^{\gamma})$, then we can compute a maximum matching in $G$ in $o(m\sqrt{n})$ time. Using our algorithm, we obtain the following result for bottleneck matching: The minimum-cost matching algorithm of Lahn and Raghvendra [33] for bipartite $K_k$-minor
free graphs requires the clusters to have a small number of boundary vertices, which is used to create a compact representation of the residual network. This compact representation becomes prohibitively large as the number of boundary vertices increase. For instance, the algorithm of \[33\] has an execution time of \(\Omega(m\sqrt{n})\) for the case where \(G\) has a balanced vertex separator of \(\Theta(n^{2/3})\). This result \[34\], on the other hand, extends to any graph with a sub-linear vertex separator. Given any graph \(G(A \cup B, E)\) that has an efficiently computable balanced vertex separator for every subgraph \(G'(V', E')\) of size \(|V'| \delta\), for \(\delta \in [1/2, 1]\), there is a 0/1 weight assignment on edges of the graph so that the weight of any matching is \(O\left(n^{2\delta \over 1+\delta}\right)\) and \(r = O\left(n^{1\over 1+\delta}\right)\). This assignment can be obtained by simply recursively sub-dividing the graph using balanced separators until each piece has \(O(r)\) vertices and \(O(mr/n)\) edges. All edges incident on the separator vertices are then assigned a weight of 1 and all other edges are assigned a weight of 0. As a result, we obtain an algorithm that computes the maximum cardinality matching in \(\tilde{O}(mn^{\delta \over 1+\delta})\) time.

### 2.2.2 Our Approach

Initially, we compute, in \(O(m\sqrt{r})\) time, a maximum matching within all pieces. Similar to the GT-Algorithm, the rest of our algorithm is based on a primal-dual method and executes in phases. Each phase consists of two stages. The first stage conducts a Hungarian search and finds at least one augmenting path containing only zero slack (with respect to the dual constraints) edges. Let the admissible graph be the subgraph induced by the set of all zero slack edges. Unlike in the GT-Algorithm, the second stage of our algorithm computes augmenting paths in the admissible graph that are not necessarily vertex-disjoint. In the second stage, the algorithm iteratively initiates a DFS from every free vertex. When a DFS finds an augmenting path \(P\), the algorithm will augment the matching immediately and terminate this DFS. Let all pieces of the graph that contain the edges of \(P\) be affected.
2.3 Minimum-Cost Planar Graph Matching Algorithm

Unlike the GT-Algorithm, which deletes all edges visited by the DFS, our algorithm deletes only those edges that were visited by the DFS and did not belong to an affected piece. Consequently, we allow for visited edges from an affected piece to be reused in another augmenting path. As a result, our algorithm computes several more augmenting paths per phase than the GT-Algorithm, leading to a reduction of number of phases from $O(\sqrt{n})$ to $O(\sqrt{w})$. Note, however, that the edges of an affected piece may now be visited multiple times by different DFS searches within the same phase. This increases the cumulative time taken by all the DFS searches in the second stage. However, we are able to bound the total number of affected pieces across all phases of the algorithm by $O(w \log w)$. Since each piece has $O(mr/n)$ edges, the total time spent revisiting these edges is bounded by $O(mrw \log(w)/n)$. The total execution time can therefore be bounded by $\tilde{O}(m(\sqrt{w} + \sqrt{r} + \frac{mr}{n}))$.

2.3 Minimum-Cost Planar Graph Matching Algorithm

Consider a bipartite graph $G(A \cup B, E)$ where $|A| = |B| = n$, the edge set $E \subseteq A \times B$ and every edge $(a, b) \in E$ has a cost specified by $c(a, b)$. A matching $M \subseteq E$ is any set of vertex-disjoint edges, and we represent its cost by $c(M) = \sum_{(a, b) \in M} c(a, b)$. $M$ is a perfect matching if $|M| = n$. A minimum-cost perfect matching is a perfect matching with the smallest cost. In this paper, we consider the problem of computing a minimum-cost perfect matching in a planar bipartite graph. There are two previous algorithms for this problem that run in $\tilde{O}(n^{3/2})$ time. The first algorithm by Lipton and Tarjan [38] is a divide and conquer algorithm based on the well-known planar separator theorem whereas the second approach by Gabow and Tarjan is based on the bit-scaling paradigm [18]. In this paper, we combine these two approaches and achieve an improved $\tilde{O}(n^{4/3})$ time algorithm to compute a minimum-cost perfect matching in bipartite planar graphs.
2.3.1 Our Result and Approach

In this paper, we use the scaling paradigm to design a new planarity exploiting algorithm that computes minimum-cost perfect matching in planar bipartite graphs. Our algorithm runs in $O(n^{4/3} \log^2 n \log nC)$ time where $C$ is the largest cost edge in the graph.\(^1\) In comparison, the previous algorithm by Lipton and Tarjan runs in $O(n^{3/2} \log n)$ time.

Both the Hungarian algorithm and the Gabow-Tarjan algorithm iteratively compute a minimum net-cost augmenting path and augment the matching along this path. To obtain a speed-up, Gabow and Tarjan observed that for graphs where the edges have positive integers as edge costs, and the optimal matching has a cost of $O(n)$, one can integrate several properties of the Hopcroft-Karp algorithm with the Hungarian algorithm. To do this, they introduce an additive error of 1 on every edge that is not in the matching and iteratively compute the minimum-cost augmenting path. The error of +1 on every non-matching edge results in longer paths having a larger cost. As a consequence, their algorithm would pick many short (both in cost and length) augmenting paths in each iteration. This is similar to how the Hopcroft-Karp algorithm behaves in the unweighted setting. As a result, they obtain a running time that is similar to the running time of Hopcroft and Karp’s algorithm, i.e., $O(m\sqrt{n} \log nC)$. Using the scaling paradigm, they guarantee that the cost of the optimal matching on each scale is $O(n)$ while also eliminating any error in cost that is introduced due to the +1 that was added edge costs.

In our algorithm, we build an $r$-division of the planar graph which breaks the graph into $O(n/r)$ edge-disjoint pieces each of $O(r)$ size. In addition, each piece has only $O(\sqrt{r})$ vertices that also participate in other pieces (such vertices are boundary vertices). Therefore, in total, there are $O(n/\sqrt{r})$ boundary vertices.

\(^1\)The algorithm will be presented for non-negative edge costs. If there are negative edge costs, a constant can be added to the weight of every edge to make them non-negative.
Similar to the Gabow-Tarjan algorithm, we use a scaling approach where each scale corresponds to a bit in the edge costs. The computation proceeds over $O(\log nC)$ scales. In each scale, our algorithm iteratively computes minimum-cost augmenting paths. We speed-up the Gabow-Tarjan $O(n^{3/2})$ computation time per scale to $\tilde{O}(n^{4/3})$ time as follows:

- Within each scale, we first execute the Gabow-Tarjan algorithm for $O(\sqrt{r})$ iterations to match all but $O(n/\sqrt{r})$ vertices. This takes $O(n\sqrt{r})$ time.

- We introduce an additive error of $+[\sqrt{r}]$ on every edge that is not in the matching and is incident on a boundary vertex. We show that doing so does not asymptotically increase the total error in the cost of the matching. Using the scaling paradigm, we can eliminate this error to obtain the optimal matching (see Section 5.2).

- Next, we compress the residual graph into a weighted and directed multigraph $H$ with the $O(n/\sqrt{r})$ boundary and free (unmatched) vertices as the vertex set. There is an edge between two boundary vertices if and only if they are connected by a directed path in one of the pieces of the $r$-division. This compressed residual graph $H$ has $O(n/\sqrt{r})$ vertices and $O(n)$ edges (see Section 5.3).

- The shortest path between two free vertices in $H$ can be used to compute the minimum-cost augmenting path. Using efficient data structures and an efficient implementation of Dijkstra’s algorithm by Fakcharoenphol and Rao (described in [15], [23], and [27]), we compute the minimum-cost augmenting path in $H$ in time $O((n/\sqrt{r}) \log^2 r)$. Therefore, the total time taken to compute the remaining $O(n/\sqrt{r})$ augmenting paths is $O((n^2/r) \log^2 r)$ (see Section 5.3.3).

- After we augment the matching, the residual graph changes and so does its compressed representation $H$. For every piece of the $r$-division that the augmenting path passes through, we recompute its edges in $H$. Each such affected piece can be
updated in $O(r \log r)$ time. Although every augmenting path can potentially pass through all the $O(n/r)$ pieces, we prove that throughout the course of the algorithm, the total number of pieces that the augmenting paths touch is $O((n/\sqrt{r}) \log n)$ (see Lemma 5.22), and therefore, the total time taken to recompute edges of these pieces is only $O(n\sqrt{r} \log r \log n)$.

- Together, the total time taken by the algorithm for computation over a single scale is $O((n^2/r) \log^2 r + n\sqrt{r} \log r \log n)$ which is $O(n^{4/3} \log^2 n)$ when $r = n^{2/3}$. By summing up over all scales, we obtain the a running time of $O(n^{4/3} \log^2 n \log nC)$.

Relation to the Thesis Work of Mudabir Kabir Asathulla  

Mudabir Kabir Asathulla, a co-author of [6], worked in conjunction with the current author on the design of matching algorithms for planar graphs. He included some results of this work in his Master’s Thesis [4]. However, the result presented in this thesis is for the unweighted planar graph matching problem, for which he provided an $\tilde{O}(n^{6/5})$ algorithm.

2.4 Min-Cost Minor-Free Graph Matching Algorithm

In this chapter, we present an efficient algorithm for minimum-cost maximum-cardinality matching for any graph that admits an $r$-clustering. A clustering of a graph $G$ is a partitioning of $G$ into edge disjoint pieces. A vertex is a boundary vertex if it participates in more than one piece of this clustering. For a parameter $r > 0$, an $r$-clustering of a graph is a clustering of $G$ into edge-disjoint pieces $\{R_1, \ldots, R_k\}$ such that $k = \tilde{O}(n/\sqrt{r})$, every piece $R_j$ has at most $O(r)$ vertices, and each piece has $\tilde{O}(\sqrt{r})$ boundary vertices. Furthermore, the total number of boundary vertices, counted with multiplicity, is $\tilde{O}(n/\sqrt{r})$.

For any directed graph $G$ with an $r$-clustering, one can compress $G$ to a graph $H$ as follows.
The vertex set of $H$ is the set of boundary vertices. There is an edge from $u$ to $v$ in $H$ if there is a directed path $\vec{P}_{u,v}$ from $u$ to $v$ that is completely contained within one of the pieces. The cost of the edge $(u, v)$ in $H$ is assigned to be the cost of the shortest path from $u$ to $v$ within the piece. It is easy to see that $H$ has $\tilde{O}(n/\sqrt{r})$ vertices and $\tilde{O}(n)$ edges. Planar graphs and $K_h$-minor graphs admit efficiently computable $r$-clusterings.

### 2.4.1 Our Results

We design an algorithm to compute minimum-cost perfect matching in bipartite graphs with an $r$-clustering. Our algorithm runs in $\tilde{O}(mr + m\sqrt{n}/r^{1/4})$ time. For $r = n^{2/5}$, we obtain an $\tilde{O}(mn^{2/5})$ time algorithm to compute the optimal matching. As consequences, we obtain the following results:

- For $K_h$-minor free graphs, we obtain an $\tilde{O}(n^{7/5} \log nC)$ time algorithm to compute the minimum-cost matching. In comparison, a min-cost matching can be computed in $\tilde{O}(n^{10/7} \log C)$ time [10].

- For planar graphs, our approach leads to an execution time of $\tilde{O}(n^{6/5} \log nC)$ improving the previous $\tilde{O}(n^{4/3} \log nC)$ time algorithm by Asathulla et al. [6] (AKLR algorithm; described in 2.3).

- For any graph with $m$ edges, $n$ vertices, and an efficiently computable $O(\sqrt{n})$-sized separator on all subgraphs, our algorithm computes a minimum-cost matching in $\tilde{O}(mn^{2/5} \log nC)$ time.\(^2\)

A summary of how our results for planar graphs and minor-free graphs compare to existing work can be found in Figure 2.1. Our algorithm relies only on the availability of an

\(^2\)To accommodate such graphs, our presentation will take into account the number of edges.
Chapter 2. Our Framework and Results

Arbitrary Graphs | $K_h$-Minor Free Graphs | Planar Graphs

| Strongly Polynomial | $O(mn)$ [30] | $\tilde{O}(n^{3/2})$ [38] | $\tilde{O}(n^{3/2})$ [38] |
| Cost Scaling Algorithms | $O(m\sqrt{n}\log nC)$ [18] | $O(n^{3/2}\log nC)$ [18] | $\tilde{O}(n^{5/3}\log nC)$ [6] |
| Matrix Multiplication | $\tilde{O}(n^\omega C)$ [47] | $\tilde{O}(n^{\omega/2}C)$ [44] | $\tilde{O}(n^{\omega/2}C)$ [44] |
| Electric Flow | $\tilde{O}(m^{10/7}\log C)$ [10] | $\tilde{O}(n^{10/7}\log C)$ [10] | $\tilde{O}(n^{10/7}\log C)$ [10] |
| Our Results | – | $\tilde{O}(n^{7/5}\log nC)$ | $\tilde{O}(n^{6/5}\log nC)$ |

Figure 2.1: Comparison of execution times of various matching algorithms. Here $C$ is the largest cost edge; $\omega < 2.373$ is the exponent of matrix multiplication complexity. Our algorithm is the fastest for both $K_h$-minor free graphs (resp. planar graphs) if $C = n^{O(1)}$ and $C = \tilde{\Omega}(n^{0.23})$ (resp. $C = \tilde{\Omega}(n^{0.03})$).

$r$-clustering. The reduction of Gabow and Tarjan from maximum cardinality minimum-cost matching to minimum-cost perfect matching preserves the $r$-clustering in the input graph. Therefore, we can use the same reduction to also compute a minimum-cost maximum cardinality matching in $\tilde{O}(mn^{2/5})$ time. Our results are based on a new strategy to speed-up augmenting path based matching algorithms. We begin by giving an high-level overview of the approach.

2.4.2 Our Approach

The HK, Hungarian, GT, and AKLR algorithms rely upon computing, in each phase, one or more vertex-disjoint minimum-cost augmenting path for an appropriate cost. To assist in computing these paths, each algorithm defines a weight on every vertex.

For instance, the HK algorithm assigns a layer number to every vertex by conducting a BFS from the set of free vertices in the residual graph. Any augmenting path that is computed in a layered graph – a graph consisting of edges that go from a vertex of some layer $i$ to layer $i + 1$ – are of minimum length. Similarly, the Hungarian, GT and AKLR algorithms assign a dual weight to every vertex that satisfy a set of dual constraints defined for each edge. Any augmenting path in an admissible graph – containing edges for which the dual constraints...
are “tight” and have zero slack – is a minimum-cost augmenting path for an appropriate cost. The Hungarian and AKLR algorithms iteratively compute such augmenting paths and augment the matching along these paths.

The GT-algorithm (resp. the HK-algorithm) computes, in each phase, a maximal set of vertex-disjoint augmenting paths in the admissible graph (resp. layered graph) by iteratively conducting a partial DFS from every free vertex. Each such DFS terminates early if an augmenting path is found. Moreover, every vertex visited by this search is immediately discarded from all future executions of DFS for this phase. This leads to an $O(m)$ time procedure to obtain a maximal set of vertex-disjoint augmenting paths.

To obtain a speed-up, we deviate from these traditional matching algorithms as follows. In each phase, we compute substantially more augmenting paths that are not necessarily minimum-cost or vertex-disjoint. We accomplish this, by allowing the admissible graph to have certain edges with positive slack. We then conduct a partial-DFS on this admissible graph. Unlike traditional methods, we do not discard vertices that were visited by the DFS and instead allow them to be reused. As a result, we discover more augmenting paths. Revisits increase the execution time per phase. Nonetheless, using the existence of an $r$-clustering, we bound the amortized execution time by $O(m)$ per phase.

So, how do we guarantee that our algorithm computes significantly more augmenting paths in each phase? The HK-algorithm measures progress made by showing that the length of the shortest augmenting path increases by at least one at the end of each phase. After $\sqrt{n}$ phases, using the fact that the length of the shortest augmenting path is at least $\sqrt{n}$, one can bound the total number of free vertices by $O(\sqrt{n})$.

In the GT-algorithm, the dual weights assist in measuring this progress. Gabow and Tarjan show that the free vertices of one set, say $B$, increases by at least 1 in each phase whereas
the dual weights of free vertices of $A$ always remains 0. After $\sqrt{n}$ phases, using the fact that the dual weights of all free vertices is at least $\sqrt{n}$, one can bound the total number of free vertices by $\sqrt{n}$. Note that this observation uses the fact that, at the beginning of each scale, the cost of the optimal matching is $O(n)$.

In our algorithm, which is also based on the scaling paradigm, we achieve a faster convergence by aggressively increasing the dual weight of free vertices of $B$ by $O(n^{1/5})$ while maintaining the dual weights of free vertices of $A$ at 0. So, the progress made in one phase of our algorithm is comparable to the progress made by $O(n^{1/5})$ phases of the GT-algorithm. As a result, at the end of $O(n^{2/5})$ phases, the dual weight of every free vertex is at least $n^{3/5}$ and the number of free vertices remaining is no more than $O(n^{2/5})$. Each of the remaining can be matched in $O(m)$ time by conducting a simple Hungarian Search leading to an execution time of $O(mn^{2/5})$.

Next, we present an overview of our algorithm.

### 2.4.3 Overview of the Algorithm

Our algorithm uses the bit-scaling framework as used in the AKLR-algorithm. Algorithm 1 provides an overview of each scale of our algorithm. The speed-up of our algorithm is achieved by the FastMatch Procedure of line 6 and 7. Similar to the AKLR-algorithm, after the preprocessing step, we have $O(n/\sqrt{r})$ free vertices remaining. The FastMatch procedure takes this matching, along with a compressed residual graph $H$ and returns a matching with all but $O(\sqrt{n}/\sqrt{r})$ many free vertices remaining. In steps 7–9, these unmatched vertices are matched by simply finding one augmenting path at a time and augmenting the matching along this path to find a perfect matching.

We give an overview of the FastMatch procedure next.
Algorithm 1 A scale of our algorithm with complexities.

1: $M \leftarrow \emptyset$ \hfill \triangleright \textbf{Time}
2: **Preprocessing Step**: Run $\sqrt{r}$ iterations of GT-Algorithm \hfill $\tilde{O}(m\sqrt{r})$
3: Compute compressed residual graph $H$ \hfill $\tilde{O}(m\sqrt{r})$
4: for $i$ from 1 to $O(\sqrt{n\sqrt{r}})$ do
5: \hspace{1em} Execute `FastMatch` to find many augmenting paths, \hfill $\tilde{O}(m)$ per iteration
6: \hspace{1em} augment and update $H$ \hfill $\tilde{O}(mr^{5/4}/\sqrt{n})$ per iteration
7: for $i$ from 1 to $O(\sqrt{n\sqrt{r}})$ do
8: \hspace{1em} $P \leftarrow \text{HungarianSearch}(G)$ \hfill $\tilde{O}(m)$ per iteration
9: \hspace{1em} Augment $M$ along $P$ \hfill $\tilde{O}(n)$ per iteration
10: return $M$

- We associate a slack with every edge of the residual graph $G$ and its compressed representation $H$. The *projection* of an edge $(u, v)$ of the compressed residual graph $H$ is a path with the smallest total slack between $u$ and $v$ inside any piece ${\mathcal{R}}_i$ of the $r$-clustering. The slack of edge $(u, v)$ is simply the total slack on the projection. Similar to the AKLR-algorithm, we say that any edge between two non-boundary vertices is admissible only if it has a zero slack. However, unlike the AKLR-algorithm, we allow for admissible edges of the residual graph that are incident on a boundary vertex to have a slack of $\sqrt{r}$. As a result of this, augmenting path of admissible edges need not be a shortest augmenting path. For an edge $(u, v)$ of the compressed residual graph, we define it to be admissible if it has a slack of at most $\sqrt{r}$.

- The `FastMatch` procedure conducts a DFS-style search from every free vertex $v$ in the admissible graph of $H$. For every vertex of $H$ that does not lead to an augmenting path, this search procedure raises its dual weight (magnitude) by $\sqrt{r}$. As a result, the DFS either finds an augmenting path and matches $v$ or raises the dual weight of $v$ by $\sqrt{r}$ as desired. When the search procedure finds an augmenting path $P$ of admissible edges in $H$, it adjusts the dual weights (through the `Sync` procedure), projects $P$ in $H$ onto the residual graph $G$ to compute an augmenting path $\bar{P}$ of admissible edge in
the residual graph $G$.

- When mapping a path $P$ in the compressed graph $H$ onto the original graph $G$, the algorithm takes each edge $(u, v) \in P$ between two boundary vertices and replaces it with a shortest path from $u$ to $v$ within one of the pieces. We call the resulting path $\vec{P}$ the projection of $P$. Due to the fact that an augmenting path $P$ in $H$ computed by the search procedure need not be a path with minimum net-cost, its projection $\vec{P}$ may be a non-simple path in the underlying graph $G$. Non-simple paths not only create difficulties in bounding execution time but also do not necessarily have admissible edges. To avoid creating such non-simple projections, when our DFS style search encounters a cycle $C$, the algorithm computes its projection $\vec{C}$ and flips the edges on the cycle immediately by setting $M \leftarrow M \oplus \vec{C}$. This modification requires us to update all pieces that contain edges of $\vec{C}$. When the search finds an augmenting path (resp. cycle) in the admissible graph of $H$, due to the active elimination of cycles, we can guarantee that its projection indeed a simple path (resp. cycle) of admissible edges.

- When the partial DFS style search finds a path between two free vertices in $H$, the algorithm has to immediately compute the projection and augments the matching. Unlike, GT or HK-Algorithm, the algorithm does not throw away vertices visited by the search. This may cause several vertices of the graph to be visited multiple times within the same phase. See below for a description of a scenario described in Figure 6.2. For this reason, unlike the Gabow-Tarjan Algorithm (where each phase takes $O(m)$ time), we cannot bound the time taken by each phase of our algorithm. We note that every vertex visited by the DFS style search either lies on an augmenting path or has its dual weight (magnitude) increased by $\sqrt{r}$. Therefore, any vertex $v$ of $H$ that is unsuccessfully visited $O(\sqrt{n}/r^{1/4})$ times by the search will have a dual
2.4. Min-Cost Minor-Free Graph Matching Algorithm

weight of a magnitude that is at least $\sqrt{nr^{1/4}}$. To overcome the difficulty posed by multiple visits, any visit by the DFS-style search after the dual weight of $v$ exceeds $\sqrt{nr^{1/4}}$ forces the search procedure to compute the projection of the path from the free vertex which initiated the DFS-style search to $v$ in $G$. The algorithm by flips the edges along this admissible alternating path to create a new free vertex of $B$ with a dual weight greater than $\sqrt{nr^{1/4}}$. This vertex is then marked as inactive and will not participate in any future execution of the FastMatch procedure. The FastMatch procedure terminates when all free vertices become inactive. Using the fact that the optimal matching has cost of $O(n)$, we can show that the number of inactive vertices cannot exceed $\tilde{O}(\sqrt{n}/r^{1/4})$.

- The total time taken by the FastMatch procedure can be attributed to the time taken by the DFS style search to find augmenting paths, alternating paths, and alternating cycles and the time taken to project, flip the edges and update the compressed residual graph for all paths and cycles computed by the search. The dual weight of any vertex cannot exceed $\sqrt{nr^{1/4}}$ and so, every vertex is visited by the search at most $\sqrt{n}/r^{1/4}$ many times. Since the compressed residual graph has $O(n)$ edges, the total time taken by the search-related operations is bounded by $\tilde{O}(n^{3/2}/r^{1/4})$. We also show that the total length of all the cycles, alternating paths, and augmenting paths computed in the compressed graph $H$ does not exceed $O((n/\sqrt{r}) \log n)$. The total time for the update-related operations is $\tilde{O}(mr)$.

Summing up over all scales, and setting $r = n^{2/5}$ gives the claimed running time.
2.5 Approximate Bottleneck Matching Algorithm

Given point sets $A$ and $B$ each consisting of $n$ $d$-dimensional points, let $G(A \cup B, A \times B)$ be the complete bipartite graph formed on the points. Each edge $(a, b) \in A \times B$ has a cost equal to the Euclidean distance $\|a - b\|$ between its endpoints. The cost of a matching $M$ on this graph $G$ is given by the cost of its longest edge; we call this edge the bottleneck edge of the matching. A bipartite bottleneck matching of the point sets $A$ and $B$ is a perfect matching in $G$ that minimizes the cost of its bottleneck edge. Let $\beta^*$ be the bottleneck edge cost of this optimal matching. A perfect matching $M$ is a $(1 + \varepsilon)$-approximate bottleneck matching if its bottleneck edge cost is at most $(1 + \varepsilon)\beta^*$. We give an algorithm that, for any fixed dimension $d$, computes a $(1 + \varepsilon)$-approximate bottleneck matching in $\tilde{O}(n^{\frac{1}{2} + \frac{d-1}{2d}} \log n)$ time. We describe the algorithm for the two-dimensional case, where the running time is $\tilde{O}(n^{1/3}/\varepsilon^4)$. The same algorithm extends to the higher-dimensional case in a straightforward fashion. This bottleneck matching algorithm uses (a slight modification of) the algorithm of Section 2.2 as a subroutine.

Although the optimal bottleneck distance $\beta^*$ is not known, by executing our algorithm $O(\log n/\varepsilon)$ times on different guesses of the value of $\beta^*$, we can ensure that at least one of these executions has a guess $\delta$ such that $\delta \leq (1 + \varepsilon/3)\beta^*$. As a result, it is sufficient to describe our algorithm under the assumption that a good approximation $\delta$ of the optimal bottleneck distance is known.

We observe that the graph $G$ has $\Theta(n^2)$ edges. In order to reduce the time taken by the algorithm, we cluster vertices in order to obtain a sparse graph with only $O(n/\varepsilon^2)$ edges. First, we overlay a grid $G$ with cell-width roughly $\varepsilon \delta$ on top of the point set. Next, we snap all points within a grid cell to its center. As a result, all edges between vertices of the same pair of cells can be treated as having identical cost, allowing the vertices within each cell to
be represented using $O(1)$ vertex clusters. The vertex clusters of a cell $\xi$ only need to be connected to clusters in nearby cells that are within a distance of $\delta$. From a simple packing argument, it follows that each cluster only needs to be connected to $O(1/\varepsilon^2)$ neighbors. As a result, the residual graph on the original point set can be approximated using a graph $G$, built on the clusters, with $O(n)$ vertices and $O(n/\varepsilon^2)$ edges. It then suffices to describe an algorithm for computing a perfect matching using this sparse residual graph. Using a slight modification of the Hopcroft-Karp algorithm, it is possible to use the sparse residual graph to compute a perfect matching in $\tilde{O}(n^{3/2}\text{poly}(1/\varepsilon))$ time. To improve upon this, we can replace the Hopcroft-Karp algorithm with the algorithm described in Section 2.2. However, this algorithm requires a 0-1 edge weight assignment such that the total weight of any matching is upper bounded by a value $w$. We create this 0-1 cost assignment as follows: First, overlay a larger grid $G'$ on top of the smaller grid $G$. For a carefully chosen parameter $r$, each cell of the larger grid contains $\sqrt{r} \times \sqrt{r}$ cells of the smaller grid. Therefore, each cell of the larger grid $G'$ has $O(r)$ vertices and $O(r/\varepsilon^2)$ edges in the sparse graph $G$. A point $p$ that is within a small cell $\xi \in G$ and also within a larger cell $\square \in G'$ is a boundary point if the
distance between the boundary of $\xi$ and the boundary of $\Box$ is at most $\delta$. By choosing the best possible shift of the large grid $G'$, it is possible to ensure that the number of boundary points is only $O(n/(\varepsilon \sqrt{r}))$. Finally, we can assign any edge that crosses between two different cells of $G'$ a weight of 1 and any other edge, whose endpoints are contained within the same large cell, a weight of 0. From the bound on the number of boundary points, it follows that the total weight of any matching is at most $w = O(n/(\varepsilon \sqrt{r}))$. By executing a slight variant of the algorithm described in Section 2.2 on the sparse residual graph $\mathcal{G}$, a perfect matching can be computed in $\tilde{O}((n^2/\varepsilon^2)((\sqrt{r} + \sqrt{w} + \frac{w}{n}))$ time. By setting $r = n^{2/3}$, we get $w = O(n/(\varepsilon \sqrt{r})) = O(n^{2/3}/\varepsilon)$, and the running time of one execution of the algorithm from Section 2.2 is $\tilde{O}(n^{4/3}/\varepsilon^3)$. Over all $O(\log(n/\varepsilon))$ guesses of the optimal bottleneck value, the overall time needed to compute a $(1 + \varepsilon)$-approximate bottleneck matching is $\tilde{O}(n^{4/3}/\varepsilon^4)$. For higher dimensions the algorithm is predominantly the same as for the two-dimensional case, except higher-dimensional grids $\mathcal{G}$ and $\mathcal{G}'$ are used, resulting in an execution time of $\frac{1}{\varepsilon \log n} n^{1 + \frac{d-1}{2d-1}} \text{poly} \log n$.

### 2.6 RMS Matching Algorithm

Given two sets $A, B$ each containing $n$ points in a two-dimensional plane, let $G(A \cup B, A \times B)$ be the complete bipartite graph formed on these point sets. Let the cost $c(a, b)$ of any edge $(a, b) \in A \times B$ be the square of the Euclidean distance between its endpoints, i.e., $c(a, b) = \|a - b\|^2$. The cost of any matching $M$ is given by $\sum_{(a,b) \in M} c(a, b)$. Let $M^*$ be an optimal RMS matching. A minimum-cost perfect matching in $G$ is an optimal RMS matching. A perfect matching $M$ is a $(1 + \varepsilon)$-approximate RMS matching if $c(M) \leq (1 + \varepsilon) c(M^*)$. In this section, we summarize our Monte Carlo randomized algorithm, which computes in $\tilde{O}(n^{5/4} \text{poly}(1/\varepsilon))$ time a $(1 + \varepsilon)$-approximate RMS matching with high probability. Throughout this section,
as well as the corresponding Chapter 7, we use the $\tilde{O}(f(n))$ notation to suppress not only poly-logarithmic factors in $n$, but also polynomial factors in $1/\varepsilon$, i.e., for any running time function $f(n)$, the notation $\tilde{O}(f(n))$ is equivalent to $O(f(n)\text{poly}(\log n, 1/\varepsilon))$.

### 2.6.1 Overview of Our Approach

Our algorithm draws insight from the $\tilde{O}(n^{4/3})$ time algorithm for computing a minimum-cost perfect matching in bipartite planar graphs [6] (described in Section 2.3). The algorithm of [6] relies on the existence of a planar vertex separator of size $O(\sqrt{n})$. A complete bipartite graph is far from planar and does not have any vertex separators. Despite this, we are able to adapt the approach of [6] to our setting. For the purpose of giving context to the RMS matching algorithm, we begin with a summary of their algorithm.

**Planar Bipartite Matching Algorithm** The algorithm of [6] is a primal-dual algorithm that iteratively adjusts the dual weights of the vertices to find an augmenting path containing zero ‘slack’ edges and then augments the matching along this path. For a parameter $r > 0$, this algorithm conducts an $O(n\sqrt{r})$ time preprocessing step and computes a matching of size $n - O(n/\sqrt{r})$. After this, their algorithm finds the remaining augmenting paths in sub-linear time by the use of an $r$-division: An $r$-division divides any planar graph into $O(n/r)$ edge-disjoint pieces, each of size $O(r)$, with only $O(n/\sqrt{r})$ many *boundary vertices* that are shared between pieces. They conduct a search for each augmenting path as follows:

- Using an $r$-division of a planar bipartite graph $G(A \cup B, E)$, they construct a compact residual graph $\tilde{G}$ with a set $\tilde{V}$ of $O(n/\sqrt{r})$ vertices – each boundary vertex of the $r$-division is explicitly added to this vertex set. In addition, the compact graph has $O(r)$ edges per piece and $O(n)$ edges in total. They define a dual weight for every vertex
of $\tilde{V}$ that satisfies a set of dual feasibility constraints on the edges of $\tilde{G}$. Interestingly, given dual weights on $\tilde{V}$ that satisfy the compressed feasibility conditions, one can derive dual weights for $A \cup B$ satisfying the classical dual feasibility conditions, and vice versa. Therefore, instead of conducting a search on $G$, their algorithm searches for an augmenting path in $\tilde{G}$.

- Their algorithm builds, for each piece of $G$, a data structure (see [15]). This data structure stores the $O(r)$ edges of $\tilde{G}$ belonging to the piece and can be constructed in $\tilde{O}(r)$ time. Using this data structure, they conduct the primal-dual search for an augmenting path in $\tilde{O}(|\tilde{V}|) = \tilde{O}(n/\sqrt{r})$ time. Over $O(n/\sqrt{r})$ augmenting path searches, the total time taken is bounded by $\tilde{O}(n^2/r)$.

Augmenting along a path reverses the direction of its edges in the residual graph. Therefore, their algorithm has to re-build the shortest path data structure for every affected piece, a piece containing at least one edge of the augmenting path. This is done in $\tilde{O}(r)$ time per piece. To reduce the number of affected pieces, an additive cost of $\sqrt{r}$ is introduced to every edge incident on the boundary vertices. It is then shown that the total additive error across all augmenting paths found by the algorithm cannot exceed $O(n \log n)$, implying that the number of affected pieces is at most $O((n/\sqrt{r}) \log n)$. The time taken to rebuild the data structure for the affected pieces is $\tilde{O}((n/\sqrt{r} \log n) \times \tilde{O}(r)) = \tilde{O}(n\sqrt{r})$. By choosing $r = n^{2/3}$, they balance the search time with the re-build time, leading to an $\tilde{O}(n^{4/3})$ time algorithm.

The ideas of a compact residual network as well as the additive error of $\sqrt{r}$ on the edges apply due to the existence of an $r$-division in planar graphs. In order to extend these ideas to our setting, we build upon the ideas from another matching algorithm, which produces a $(1 + \varepsilon)$-approximation for the Euclidean bipartite matching problem [51]. We give a brief overview of this algorithm next.
Approximate Euclidean Matching  The algorithm of [51] introduces a \((1 + \varepsilon)\) approximation of the Euclidean distance based on a quad-tree \(Q\). Using a standard technique, the input is transformed so that the optimal matching cost is \(O(n/\varepsilon)\) and the height of the quad-tree \(Q\) is \(O(\log n)\). Any edge of the complete bipartite graph appears at the least common ancestor of its endpoints in \(Q\). The set of edges appearing within each quadtree square is then partitioned into \(\text{poly}(\log n, 1/\varepsilon)\) many bundles and all edges within the same bundle are assigned the same cost. This assigned cost is an upper bound on the actual Euclidean cost.

Furthermore, the authors show that, if the quad-tree is randomly shifted, the expected cost assigned to any edge is at most \((1 + \varepsilon)\) times the Euclidean distance. Using this, the authors switch to computing a matching with respect to this new quad-tree distance.

Using the edge bundles and certain carefully precomputed shortest paths in the residual graph, the algorithm of [51] stores a \(\text{poly}(\log n, 1/\varepsilon)\) size associated graph at each square of the quad-tree. Their algorithm iteratively finds a minimum-cost augmenting path \(P\). Note that this is not done by using a primal-dual method, but by executing a Bellman-Ford search on the associated graph of each square that contains at least one point on the path \(P\). Since each point of \(P\) has at most \(O(\log n)\) ancestors and the size of the associated graph is \(\text{poly}(\log n, 1/\varepsilon)\) within each square, the total time taken to find an augmenting path can be bounded by \(\tilde{O}(|P|)\). Augmenting the matching along \(P\) requires the associated graph to be reconstructed for the \(O(\log n)\) ancestors of each of the points of \(P\). This again can be done using the Bellman-Ford algorithm, resulting in a total update time of \(\tilde{O}(|P|)\). The total length of all the augmenting paths computed by the algorithm can be shown to be \(\tilde{O}(n \log n)\), and so the total time taken by the algorithm is near-linear in \(n\).

Our Algorithm  Similar to the Euclidean case, we can transform our input so that our optimal matching cost is \(O(n/\varepsilon^2)\) (see Section 7.2.1) and store the input in a quadtree \(Q\)
of height $O(\log n)$. For the squared-Euclidean distance, we combine the ideas from the two algorithms of [6] and [51] in a non-trivial fashion. First, we note that using poly($\log n, 1/\varepsilon$) edge bundles leads to an explosion in the expected distortion. In order to keep the expected distortion small, we create approximately $\tilde{O}(2^{i/2})$ edge bundles for a square of side-length $2^i$. This causes larger squares have many more bundles of edges (See Section 7.2). For instance, a square of side-length $n$ can have roughly $\sqrt{n}$ edge bundles. A useful property of this distance approximation is that any edge appearing in a square of side length $3^2 2^i$ has a quad-tree distance value roughly between $\Omega(2^i)$ and $O(2^{2i})$. This implies that all edges with a small quad-tree distance appear within edge bundles of the smaller squares. Like in the Euclidean case, we can show that our distance is an upper bound on the squared-Euclidean distance. Furthermore, if $Q$ is a randomly shifted quad-tree, we can show that the expected cost of our distance is at most $(1 + \varepsilon)$ times the squared-Euclidean distance.

In the squared-Euclidean quad-tree distance, the number of edge bundles at each square of the quad tree is a polynomial in $n$. Using these bundles, we define a sub-linear sized associated graph. However, unlike the algorithm of [51], using the Bellman-Ford search procedure to find an augmenting path in the associated graph will lead to an $\Omega(n^{3/2})$ time algorithm. Therefore, instead of the Bellman-Ford algorithm, we employ a primal-dual approach.

Prior to describing our algorithm and the data structure, we note that primal-dual search procedures, such as Hungarian search and our algorithm, find augmenting paths in increasing order of their “costs”. As a result, such a search on quad-tree distances will initially involve only the edges with small quadtree distance and, as the algorithm progresses, larger quad-tree distances get involved. Therefore, the searches can initially be localized to smaller squares of the quad-tree and our algorithm only needs to build the associated graphs in the smaller

---

$^3$Throughout this paper, we set the side-length of the square to be the difference in the x-coordinate values of the the vertical boundaries, i.e., the Euclidean length of each of its four edges.
squares. As the algorithm progresses, however, longer edges participate in the augmenting paths, which forces our algorithm to build associated graph data structures in larger squares, increasing the time taken to conduct a Hungarian search. We refer to these squares where the data structure is maintained as active squares.

Now we present an overview of our algorithm and the data structure within an active square $\Box^*$ of width $2^i$. We partition $\Box^*$ into $O(2^{2i/3})$ pieces using a grid of side-length $2^i = 2^{2i/3}$. Similar to the planar graph algorithm, we build an associated graph $\tilde{G}$ as follows: The vertex set $\tilde{V}$ contains $\tilde{O}(2^{i/3})$ vertices per piece and $\tilde{O}(2^i)$ vertices in total. For pairs of vertices $u, v$ that belong to the same piece, we explicitly store an edge; we refer to these edges as internal edges. There are $\tilde{O}(2^{2i/3})$ internal edges per piece and the internal edges in each piece can be constructed in $\tilde{O}(2^i)$ time (see Sections 7.5.2–7.5.4). Similar to the approximate Euclidean matching algorithm, these internal edges of the associated graph represent certain shortest paths in the residual graph. Additionally, for any pair of vertices $u, v \in \tilde{V}$, we add a bridge edge between them with a cost that is approximately the squared Euclidean distance between the end-points. We do not store the bridge edges explicitly. Instead, we build a Well Separated Pair Decomposition (WSPD) of size $\tilde{O}(2^i)$ to store them [21]. Therefore, the total size of the graph is restricted to $\tilde{O}(2^i)$ vertices and $\tilde{O}(2^{4i/3})$ edges.

Next, we define dual weights on every vertex of the associated graph and define compressed feasibility conditions that are satisfied by its edges (see Section 7.5.5). Interestingly, one can use the feasible dual weights on the associated graph vertices to derive a set of dual weights satisfying the classical matching feasibility conditions (see Section 7.5.7). Using compressed feasibility, we provide a quick way to conduct primal-dual searches on the associated graph resulting in a running time of $\tilde{O}(2^{4i/3})$ per search (see Section 7.5.8). We show that the number of primal-dual searches on the associated graph of any active square with side-length $2^i$ is only $\tilde{O}(n/2^i)$ (see Section 7.4.1). Therefore, the total time spent for all searches
within active squares of side-length $2^i$ is $\tilde{O}(n2^{i/3})$.

Suppose the primal-dual search at $\square^*$ returns an admissible augmenting path. The algorithm then augments the matching along this path. Augmentation forces the algorithm to rebuild the set of internal edges within every affected piece of the associated graph at $\square^*$, i.e., pieces that contain at least one edge of $P$. In order to reduce the number of such updates, similar to [6], we assign an additive cost of roughly $\frac{2^{2i/3}}{\log n}$ to every bridge edge of the associated graph. We argue that this additional error does not increase the optimal matching cost by more than a multiplicative factor of $\varepsilon$.

To bound the time taken to rebuild the internal edges, similar to [6], we argue that the total additive cost of the edges on the augmenting paths, computed over the entire algorithm, cannot exceed $\tilde{O}(n)$ (see Section 7.4.1). Every bridge edge of the associated graph $\tilde{G}$ has an error of at least $\frac{2^{2i/3}}{\log n}$. Therefore, the number of times such edges participate across all augmenting paths is only $\tilde{O}(\frac{n}{2^{2i/3}})$. As a result, the total number of rebuilds of internal edges, for pieces of all active squares of side-length $2^i$, across the entire algorithm, is $\tilde{O}(n/2^{2i/3})$.

Rebuilding the internal edges of one piece takes $\tilde{O}(2^i)$ time (see Section 7.5.6). Therefore, the total time spent rebuilding pieces is $\tilde{O}(n2^{i/3})$, which matches the total time taken for all searches on the associated graph for layer $i$ active squares.

As the algorithm progresses, larger squares become active. When the side-length of the active square is approximately $n^{3/4}$, the time taken to execute a single search on the associated graph becomes $\Omega(n)$. At this point, we show that there are only $\tilde{O}(n^{1/4})$ free vertices remaining. Each remaining free vertex can be matched by conducting an efficient Hungarian search on the original points in $\tilde{O}(n)$, taking $\tilde{O}(n^{5/4})$ time in total. The total time spent on searches and rebuilds on active squares with side-length at most $2^{(3/4)\log_2 n} = n^{3/4}$ using our data structure is $\tilde{O}(n2^{(1/4)\log_2 n}) = \tilde{O}(n^{5/4})$, giving a total running time of $\tilde{O}(n^{5/4})$. 
Chapter 3

Background

In this section, we describe in detail four fundamental matching algorithms that form the foundation of our approaches. In addition, we present relevant background information concerning graph separators. We begin with a description of the Ford-Fulkerson algorithm for maximum cardinality matching, which runs in $O(mn)$ time (Section 3.1). This algorithm is based on the idea of finding so-called ‘augmenting paths’, which is also a fundamental approach in the realm of network flows. Next, we describe the Hopcroft-Karp algorithm for maximum cardinality matching, which improves the running time to $O(m \sqrt{n})$ by computing multiple augmenting paths per iteration (Section 3.2). We follow with a description of the Hungarian algorithm (Section 3.3) and the Gabow-Tarjan algorithm (Section 3.4), which can be seen as extensions of the Ford-Fulkerson and Hopcroft-Karp algorithms to the weighted setting. We also present relevant background related to graph separators (Section 3.5).

Throughout this chapter, we denote by $A_F$ (resp. $B_F$) the subset of vertices of $A$ (resp. $B$) that are free. Given a bipartite graph $G$ and a matching $M$ in this graph, we recall that it is possible to construct a directed residual graph $\overrightarrow{G}_M$ that captures the set of alternating paths. This is accomplished by assigning each edge of $E$ a direction. Any edge in $M$ is directed from $A$ to $B$, and any edge not in $M$ is directed from $B$ to $A$. Any directed path in $\overrightarrow{G}_M$ is an alternating path in $G$ with respect to $M$. Furthermore, any path $P$ in $\overrightarrow{G}_M$ that starts at a free vertex $b \in B_F$ and ends at a free vertex $a \in A_F$ forms an augmenting path. Augmenting along this path results in a new matching $M' \leftarrow M \oplus P$ of one higher
cardinality, i.e., $|M'| = |M| + 1$. This notion of a residual graph is used throughout the algorithms described in this chapter.

### 3.1 The Ford-Fulkerson Algorithm

#### 3.1.1 The Algorithm

The Ford-Fulkerson algorithm [16] works by repeatedly finding an augmenting path and augmenting along it. The algorithm maintains a matching $M$; initially, $M = \emptyset$. It executes $O(n)$ iterations, where each iteration finds a single augmenting path. To find an augmenting path, the algorithm constructs the residual graph, with two additional vertices $s$ and $t$ added. The vertex $s$ is connected to every vertex of $B_F$ by a directed edge, and every vertex of $A_F$ is connected to $t$ by a directed edge. Then, the algorithm can find a path from $s$ to $t$ in $O(m)$ time, using any standard graph search algorithm, such as DFS or BFS. If there is no path from $s$ to $t$, then there is no augmenting path in the graph, the matching is of maximum cardinality, and the algorithm terminates. Otherwise, the path from $s$ to $t$ with the vertices $s$ and $t$ removed forms an augmenting path $P$. The algorithm augments along $P$ by setting $M \leftarrow M \oplus P$. This completes the description of the algorithm for an iteration. Each iteration increases the size of the matching by 1, and the algorithm iterates until the matching becomes maximum. Since the maximum size of any matching is $n/2$, the algorithm executes $O(n)$ iterations. Each iteration takes $O(m)$ time, so the total time is $O(mn)$.

#### 3.1.2 Correctness of the Algorithm

We begin by giving a detailed explanation as to why the process of augmentation results in a valid matching, as shown by the following lemma.
Lemma 3.1. Let $M$ be a matching, and let $P$ be an augmenting path with respect to $M$. Let $M' = M \oplus P$. Then $M'$ is a matching with $|M'| = |M| + 1$.

Proof. First, observe that, since an augmenting path is an alternating path that begins and ends at a free vertex, any augmenting path has odd length and begins and ends with a non-matching edge. Therefore, there are exactly $\lceil |P|/2 \rceil$ edges that enter and exactly $\lfloor |P|/2 \rfloor$ edges that exit the matching due to augmentation. This implies that $|M'| = |M| + 1$.

Next, we argue $M'$ is a matching. Consider any vertex $v$. If $v$ is not on $P$, then the matching edges incident on $v$ are the same for $M$ and $M'$. If $v$ is on $P$, and is anything but the first or last vertex, then there was one edge of $P \cap M$ incident on $v$ and one edge of $P \setminus M$ incident on $v$. Therefore, the number of matching edges incident on $v$ remains unchanged from $M$ to $M'$. Finally, if $v$ is an endpoint of $P$, then $v$ is not matched by $M$, and has exactly one edge of $P \setminus M$, which becomes a matching edge in $M'$. Therefore, exactly one edge of $M'$ is incident on $P$. We conclude that $M'$ is a matching.

To show that the algorithm terminates with a maximum-cardinality matching, we must show the following lemma, which is commonly referred to as Berge’s Lemma.

Lemma 3.2. A matching $M$ is maximum if and only if there is no augmenting path with respect to $M$.

Proof. It is easy to see that a matching is maximum only if there is no augmenting path because, if there was an augmenting path, then it could be used to obtain a larger matching from Lemma 3.1.2. The more difficult direction requires showing that $M$ is maximum if there is no augmenting path. Let $M^*$ be any maximum matching. Consider the graph formed from the set of edges $S = M \oplus M^*$. Every edge of $S$ is in either $M$ or $M^*$. Since $M$ and $M^*$ are matchings, every vertex has degree at most 2 in the graph of $S$. Therefore, $S$ must
contain solely the following types of components:

(i) Single isolated vertices.

(ii) Cycles whose edges alternate between edges of $M$ and edges of $M^*$.

(iii) Even-length alternating paths with respect to $M$.

(iv) Augmenting paths with respect to $M$.

Note that, since $M^*$ is maximum, there are no augmenting paths with respect to $M^*$. Therefore, all paths in $S$ must either begin or end with an edge of $M$.

Observe that every edge not in $S$ is either in both $M$ and $M^*$ or neither $M$ nor $M^*$. Therefore, $S$ contains exactly $|M^*| - |M|$ more edges of $M^*$ than $M$. Each component that falls under cases (i)–(iii) have an equal number of edges of $M$ and $M^*$. Therefore, there must be exactly $|M^*| - |M|$ components that fall under case (iv). We conclude that $M$ is maximum if there is no augmenting path with respect to $M$. \hfill \Box

**Corollary 3.3.** $M^* \oplus M$ contains exactly $|M^*| - |M|$ augmenting paths with respect to $M$.

### 3.2 The Hopcroft-Karp Algorithm

The Hopcroft-Karp algorithm [22] improves the running time of the Ford-Fulkerson algorithm by computing multiple augmenting paths per phase, resulting in an $O(m\sqrt{n})$ time algorithm for computing a maximum matching in a bipartite graph. First, we describe the algorithm, and then we discuss its analysis.
3.2. The Hopcroft-Karp Algorithm

3.2.1 Algorithm

The Hopcroft-Karp executes in iterations, which we will call phases. Each phase will use standard graph search techniques to find a maximal set of vertex-disjoint shortest augmenting paths and then augment along all paths in this set. This set of augmenting paths is found by executing two steps: the BFS step and the DFS step.

The BFS Step The algorithm begins by constructing the residual graph $\overrightarrow{G}$ with respect to the current matching $M$. The algorithm executes a breadth-first search starting from all the free vertices of $B_F$. This BFS procedure computes a layer number $\ell(v)$ for every $v \in A \cup B$; every $v \in B_F$ has a layer number $\ell(v) = 0$, all neighbors of vertices of $B_F$ have a layer number of 1, etc. Let $\ell = \min_{a \in A_F} \ell(a)$ be the minimum layer number of all free vertices of $A$. If $\ell = \infty$ (i.e., no free vertices of $A_F$ were reached), then the matching is maximum and the algorithm terminates. Otherwise, using these layer numbers, the algorithm constructs an admissible graph $\mathcal{A}$, the subgraph of $\overrightarrow{G}$ containing every edge directed from $u$ to $v$ such that:

(i) $\ell(v) = \ell(u) + 1$, and,

(ii) $\ell(v) \leq \ell$.

This admissible graph is useful in that it represents the set of shortest augmenting paths with respect to $M$. Specifically, a path is a shortest augmenting path with respect to $M$ if and only if it is present in $\mathcal{A}$.

The DFS Step The purpose of the DFS step is to identify a maximum set of vertex disjoint augmenting paths $\mathcal{P}$ in the admissible graph $\mathcal{A}$. This is done by executing a series of partial DFS searches, one from each vertex of $B_F$. For each vertex $v \in B_F$, the algorithm executes
a DFS search from \( v \). If a free vertex of \( A \) is reached at any point during a partial DFS, a shortest augmenting path \( P \) has been found. The algorithm adds \( P \) to \( \mathcal{P} \) and terminates the partial DFS. If no augmenting path is found, the partial DFS terminates when the search backtracks from \( v \). After completing a partial DFS search from \( v \), regardless of whether an augmenting path was found or not, the algorithm deletes all vertices that were explored during the search.

After a partial DFS is executed from every free vertex of \( B_F \), the set \( \mathcal{P} \) is a maximum set of vertex-disjoint shortest augmenting paths. The algorithm proceeds to augmenting \( M \) along every path \( P \in \mathcal{P} \). Since the augmenting paths are disjoint, the result is a valid matching \( M' \) with \( |M'| = |M| + |\mathcal{P}| \).

### 3.2.2 Analysis

Next, we give an analysis of the Hopcroft-Karp algorithm. The correctness of the algorithm follows from the fact that, like the Ford-Fulkerson algorithm, at least one augmentation occurs per phase. This continues until the algorithm obtains a maximum matching. The main concern of the analysis is bounding the efficiency of the algorithm. For simplicity, we assume \( m \geq n - 1 \); otherwise, the graph can be split into disjoint components, and each component can be treated as a disjoint problem. First, we bound the time taken by each phase. The BFS portion of each phase takes \( O(m) \) time. The DFS portion executes several partial DFS searches, but each edge is only processed once across all partial DFS searches. Therefore, the total time taken for each phase is \( O(m) \).

Next, we bound the total number of phases executed by the algorithm as \( O(\sqrt{n}) \). We first state a key invariant of the Hopcroft-Karp algorithm.
(II) After phase $i$ of the algorithm, the length of the shortest augmenting path is at least 
$i + 1$.

We first bound the number of phases assuming this invariant holds. Then, we prove the 
invariant.

**Lemma 3.4.** After $i$ phases, there are $O(n/i)$ free vertices remaining.

*Proof.* Let $M^*$ be a maximum matching. From Corollary 3.3, $S = M^* \oplus M$ has exactly 
$|M^*| - |M|$ augmenting paths with respect to $M$, all of which are vertex-disjoint. From 
invariant (II), each of these augmenting paths have a length at least $i + 1$. We observe that 
$S$ contains at most $2n$ edges, so the number of augmenting paths in $S$ can be bounded by 
$O(n/i)$. \qed

From Lemma 3.4, we have that, after $\sqrt{n}$ phases, there are only $O(\sqrt{n})$ free vertices remaining. Each of these free vertices will be matched using only $O(\sqrt{n})$ additional phases, since 
at least one augmenting path is found during each phase. Therefore, the total number of 
phases is $O(\sqrt{n})$. Since each phase takes $O(m)$ time, the total time taken by the algorithm 
is $O(m\sqrt{n})$.

Finally, we prove the invariant (II).

**Lemma 3.5.** Let $M$ be a matching, and let $\mathcal{P}$ be a maximum set of vertex-disjoint shortest 
augmenting paths with respect to $M$, where each path in $\mathcal{P}$ has exactly $k$ edges. Let $M'$ be 
the matching after augmenting $M$ along every path in $\mathcal{P}$. Then the shortest augmenting with 
respect to $M'$ is at least $k + 1$.

*Proof.* Assume for the sake of contradiction that there is some augmenting path $P'$ with 
respect to $M'$ such that $|P'| \leq k$. First consider the case where $P'$ is disjoint from all paths
of $\mathcal{P}$. Since $P'$ is disjoint from all paths of $\mathcal{P}$, we have that $P'$ is also an augmenting path with respect to $M$. If $|P'| = k$, then $P'$ can be added to the set $\mathcal{P}$, contradicting the assumption that $\mathcal{P}$ was a maximal set. Otherwise, $|P'| < k$, which contradicts the assumption that the shortest augmenting path with respect to $M$ had length $k$.

Therefore, we can assume that $P'$ intersects at least one path $P \in \mathcal{P}$. Since $P$ and $P'$ share a vertex, they must also share an edge that is not in $M$. The set $P \oplus P'$ contains exactly 2 augmenting paths $P_1$ and $P_2$ with respect to $M$ such that $|P_1| + |P_2| \leq |P| + |P'| - 2 \leq 2k - 2$. Therefore, either $|P_1| < k$ or $|P_2| < k$. This contradicts the assumption that the shortest augmenting path with respect to $M$ had length $k$. We conclude that there can be no augmenting path with respect to $M'$ with length $k$ or less.

\[\square\]

### 3.3 The Hungarian Algorithm

In this section, we present the Hungarian algorithm \[30\] for computing a minimum-cost maximum cardinality matching on a bipartite graph $G(A \cup B, E)$ in $O(mn + n^2 \log n)$ time. Each edge $(u, v)$ has a cost $c(u, v)$ and the cost of any matching is given by $c(M) = \sum_{(u, v) \in M} c(u, v)$. Our goal is to find a matching $M$ of maximum cardinality that minimizes $c(M)$. Without loss of generality, we can assume that all edge costs are non-negative. Negative edge costs can be removed by adding a sufficiently large value uniformly to the costs of all the edges; doing so does not change the relative cost ordering between any pair of matchings.

The Hungarian algorithm can be seen as analogous to the Ford-Fulkerson algorithm, except that each iteration computes an augmenting path of minimum cost (for an appropriately defined definition of cost). First, we introduce the notion of dual feasibility, which is important for both the sake of correctness and efficiency of the algorithm.
3.3. The Hungarian Algorithm

3.3.1 Feasibility

Throughout its execution, the Hungarian algorithm maintains a set of dual weights $y(\cdot)$ on the vertices of $A \cup B$, and a matching $M$. To show that the matching maintained by the algorithm has a sufficiently low cost, we introduce a set of dual feasibility constraints. We say that a matching $M$ and set of dual weights $y(\cdot)$ are feasible if, for every edge $(u, v)$,

\begin{align*}
  y(u) + y(v) &\leq c(u, v) \quad \text{if} \ (u, v) \notin M. \quad (3.1) \\
  y(u) + y(v) &= c(u, v) \quad \text{if} \ (u, v) \in M. \quad (3.2)
\end{align*}

As an additional requirement for feasibility, we require that the dual weights on the vertices satisfy the following constraints:

(i) For any $v \in A$, $y(v) \leq 0$.

(ii) For any $v \in B$, $y(v) \geq 0$.

(iii) For any $v \in B_F$, $y(v) = y_{\max}$, where $y_{\max} = \max_{v' \in A \cup B} y(v')$ is the largest dual weight of any vertex.

(iv) For any $v \in A_F$, $y(v) = 0$.

The Hungarian algorithm can be seen as optimizing two objectives. The first objective is to maximize the number of edges in the matching, and the second objective is to minimize the cost of the matching. The problem can be formulated as a linear program that captures these constraints and solved using any standard linear programming method, such as the simplex algorithm. Within the scope of this work, however, we approach the problem from a purely graph-theoretic standpoint. Although, it is worth noting that the feasibility constraints presented here can be seen as constraints for the linear programming dual of this problem.
Within this context, the Hungarian algorithm can be seen as a primal-dual algorithm in that it maintains ‘dual-feasibility’ while working to obtain primal feasibility (i.e., a maximum matching). However, this interpretation of dual weights is not necessary for understanding the work presented here.

We introduce a notion of net-cost with respect to any matching $M$. Any edge $(u, v) \notin M$ has a net-cost of $\phi(u, v) = c(u, v)$. Any edge $(u, v) \in M$ has a net-cost of $\phi(u, v) = -c(u, v)$. For any set of edges $S$, the net-cost is simply $\phi(S) = \sum_{(u,v) \in S} \phi(u, v)$, the sum of the net-costs of its edges. The net-cost corresponds to the resulting change in the cost of the matching from setting $M \leftarrow M \oplus S$. For example, if $S$ is an augmenting path, then $\phi(P)$ is the change in the cost of the matching that results from augmentation along $P$. Next, we relate net-costs to the feasibility conditions.

**Lemma 3.6.** Let $M, y(\cdot)$ be a feasible matching and set of dual weights. Let $P$ be any even-length alternating path with respect to $M$ that begins with a free vertex $u$ and ends with a matched vertex $v$. Then,

$$\phi(P) \geq 0.$$  

**Proof.** From the feasibility conditions, we have,

$$\phi(P) = \sum_{(u', v') \in P} \phi(u', v')$$

$$= \sum_{(u', v') \in P \setminus M} c(u', v') - \sum_{(u', v') \in P \cap M} c(u', v')$$

$$\geq \sum_{(u', v') \in P \setminus M} y(u') + y(v') - \sum_{(u', v') \in P \cap M} y(u') + y(v')$$  

$$= y(u) - y(v).$$  

The last inequality follows from the fact that, since $P$ is alternating, each vertex of $P$, aside
from the first and the last, participates in exactly one matching edge and exactly one non-matching edge. The vertex \( u \), since it is free, only participates in a non-matching edge of \( P \), which contributes a non-negative net-cost. In contrast, the last vertex of \( P \) only participates in a single matching edge of \( P \) with a non-positive net-cost. There are two possible cases: either \( u, v \in B \) or \( u, v \in A \). For the case where \( u, v \in B \), the quantity \( y(u) - y(v) \) is non-negative from the assumptions that free vertices of \( B \) have the largest dual weight and all vertices of \( B \) have non-positive dual weights. Similarly, if \( u, v \in A \), then the net-cost is non-negative from the assumptions that \( y(u) = 0 \) and \( y(v) \leq 0 \). Therefore, \( \phi(P) \leq 0 \).

**Corollary 3.7.** Let \( M, y(\cdot) \) be a feasible matching and set of dual weights. Let \( C \) be any alternating cycle with respect to \( M \). Then, \( \phi(C) \geq 0 \).

**Proof.** The claim follows from equation (3.3) and the fact that every vertex of \( C \) participates in exactly one edge of \( C \cap M \) and exactly one edge of \( C \setminus M \).

Next, we argue that if \( M \) is a feasible maximum-cardinality matching, then \( M \) is also a minimum-cost maximum cardinality (optimal) matching.

**Lemma 3.8.** Let \( M \) be a maximum matching, and let \( y(\cdot) \) be a feasible set of dual weights with respect to \( M \). Then, for any other maximum cardinality matching \( M' \), we have \( c(M) \leq c(M') \).

**Proof.** The edges of the set \( S = M \oplus M' \) form solely even-length alternating cycles and even-length alternating paths with respect to \( M \). Any alternating cycle has non-negative net-cost from Corollary 3.7. Any even-length alternating path \( P \) in \( S \) must begin at free vertex \( u \) w.r.t \( M \) and end at a matched vertex \( v \). Therefore, \( P \) has a non-negative net-cost from Lemma 3.6. We conclude that the net-cost of \( S \) with respect to \( M \) is non-negative, i.e., \( \phi(S) \geq 0 \). Noting that \( \phi(S) = c(M') - c(M) \) gives \( c(M) \leq c(M') \).
3.3.2 Algorithm

From Lemma 3.8, any feasible maximum-cardinality matching is also optimal. Therefore, our algorithm simply needs to iteratively increase the size of its matching while maintaining a feasible set of dual weights. Initially, the algorithm assigns every vertex a dual weight of 0 for every vertex \( v \in A \cup B \). Since all edge costs are non-negative, it is clear that feasibility condition (3.1) holds. It is also easy to see that conditions (i) through (iv) hold as well. The algorithm will execute \( O(n) \) iterations, where each iteration increases the size of the matching \( M \) by one while maintaining feasibility of \( M, y(\cdot) \). Each iteration takes \( O(m + n \log n) \) time, giving a total time of \( O(mn + n^2 \log n) \) for the algorithm. It remains to describe the algorithm for each iteration, but before doing so, we must introduce a few additional definitions.

We define the slack of each edge with respect to to any matching \( M \) and set of dual weights \( y(\cdot) \) as follows:

\[
\begin{align*}
    s(u, v) &= c(u, v) - y(u) - y(v) \quad \text{if } (u, v) \notin M. \\
    s(u, v) &= y(u) + y(v) - c(u, v) \quad \text{if } (u, v) \in M.
\end{align*}
\]

If \( M, y(\cdot) \) are feasible, then the slack of any edge is non-negative, and the slack of any matching edge is exactly 0. We say an edge \((u, v)\) is admissible if it has zero slack, i.e., \( s(u, v) = 0 \). Note that feasible matching edges are always admissible. Admissible edges are useful because any admissible edge simultaneously satisfies both feasibility conditions (3.1) and (3.2), and therefore can enter or exit the matching freely without violating feasibility. Specifically, given any admissible augmenting path \( P \) with respect to \( M, y(\cdot) \), the matching \( M' = M \oplus P \) after augmenting along \( P \) remains feasible with respect to \( y(\cdot) \). Therefore, it is sufficient to find an admissible augmenting path per iteration and augment along it.
Hungarian Search  At the beginning of each iteration, an admissible augmenting path may not exist. To create one, the algorithm performs dual adjustments using the following procedure, which is commonly known as ‘Hungarian search’. Recall that the residual graph $\overrightarrow{G}_M$ is formed by directing non-matching edges from $B$ to $A$ and matching edges from $A$ to $B$. The procedure assigns each edge $(u, v)$ in the residual graph a weight equal to its slack $s(u, v)$. It adds a source vertex $s$ and connects it to every free vertex of $B_F$ using a 0 cost edge. The procedure executes a Dijkstra search started at $s$, which computes the distance $\ell_v$ to every vertex $v \in A \cup B$ from the nearest free vertex of $B$. Let $\ell = \min_{v \in A_F} \ell_v$ be the minimum distance to any free vertex of $A$. If $\ell = \infty$, then there is no augmenting path, the matching $M$ is maximum, and the algorithm terminates with an optimal matching. Otherwise, the algorithm uses the shortest path distances to perform dual adjustments as follows: Any vertex $v$ whose distance $\ell_v \geq \ell$ is left unchanged. Otherwise, $\ell_v < 0$, and the algorithm sets $y(v) \leftarrow y(v) + (\ell - \ell_v)$ if $v \in B$ and $y(v) \leftarrow y(v) - (\ell - \ell_v)$ if $v \in A$.

We will show that the Hungarian search procedure does not violate the feasibility conditions. Furthermore, we will argue that there is at least one admissible augmenting path after executing the Hungarian search. Specifically, the shortest path tree computed by Dijkstra’s algorithm must contain at least one augmenting path, assuming that matching is not maximum. We will show that all edges of the Dijkstra shortest path tree are admissible after performing dual adjustments. Given the admissible augmenting path $P$, the algorithm proceeds to augment the matching by setting $M \leftarrow M \oplus P$. This completes the description of a single iteration of the algorithm. The algorithm executes $O(n)$ iterations, where each iteration performs a Dijkstra search that takes $O(m + n \log n)$ time. Therefore, the total time taken is $O(mn + n^2 \log n)$.

Next, we argue that the dual weights after Hungarian search are feasible. Note that the dual adjustments only increase the dual weights of vertices of $B$ and only decreases the
dual weights of vertices of \( A \). Furthermore, free vertices of \( B_F \) have the largest dual weight increase, and free vertices of \( A_F \) have no change in dual weight. Therefore, the dual weights continue to satisfy conditions (i) through (iv) after performing the dual adjustment. The following lemmas argue that the dual adjustment does not violate feasibility conditions (3.1) and (3.2).

**Lemma 3.9.** Let \( M, y(\cdot) \) be a feasible matching and set of dual weights maintained by the algorithm prior to executing a Hungarian search. Let \( y'(\cdot) \) be the dual weights after the Hungarian search performs its dual adjustments. Then, for any non-matching edge \((u, v) \notin M\), condition (3.1) holds with respect to \( y'(\cdot) \).

**Proof.** Consider four cases for any non-matching edge \((u, v)\) directed from \( u \in B \) to \( v \in A \).

(a) \( \ell_u < \ell \) and \( \ell_v < \ell \): From the properties of shortest path distances, we have that \( \ell_v - \ell_u \leq s(u, v) \). Therefore,

\[
y'(u) + y'(v) = y(u) + (\ell - \ell_u) + y(v) - (\ell - \ell_v) \\
= y(u) + y(v) + \ell_v - \ell_u \\
\leq y(u) + y(v) + s(u, v) \\
= c(u, v)
\]  

and \((u, v)\) is feasible.

(b) \( \ell_u < \ell \) and \( \ell_v \geq \ell \): We have,

\[
y'(u) + y'(v) = y(u) + (\ell - \ell_u) + y(v) \\
\leq y(u) + (\ell - \ell_u) + y(v) - (\ell - \ell_v) \\
\leq c(u, v) \quad \text{from (3.4)}
\]
3.3. The Hungarian Algorithm

and \((u, v)\) is feasible.

(c) \(\ell_u \geq \ell\) and \(\ell_v < \ell\): We have,

\[
y'(u) + y'(v) = y(u) + y(v) + (\ell - \ell_v) \\
\leq y(u) + (\ell - \ell_u) + y(v) - (\ell - \ell_v) \\
\leq c(u, v) \quad \text{from (3.4),}
\]

and \((u, v)\) remains feasible.

(d) \(\ell_u \geq \ell\) and \(\ell_v \geq \ell\): Neither \(y(u)\) nor \(y(v)\) change, so the edge \((u, v)\) remains feasible.

\[\square\]

**Corollary 3.10.** For any non-matching edge \((u, v)\) that is part of the shortest path tree computed by Dijkstra’s algorithm, where \(\ell_u \leq \ell\) and \(\ell_v \leq \ell\), then \(s(u, v) = 0\).

*Proof.* Since \((u, v)\) is an edge in the shortest path tree, we have \(\ell_u - \ell_v = s(u, v)\). The claim easily follows from the fact that (3.4) becomes a strict equality. \[\square\]

Next, we argue that the matching edges are feasible after performing a Hungarian search.

**Lemma 3.11.** Let \(M, y(\cdot)\) be a feasible matching and set of dual weights maintained by the algorithm prior to executing a Hungarian search. Let \(y'(\cdot)\) be the dual weights after the Hungarian search performs its dual adjustments. Then, for any matching edge \((u, v) \in M\), condition (3.2) holds with respect to \(y'(\cdot)\).

*Proof.* Since \((u, v) \in M\), \((u, v)\) is directed from \(u\) to \(v\), where \(u \in A\) and \(v \in B\). Since \(v \in B\), and is a matched vertex, the only incoming edge to \(v\) is via the edge \((u, v)\), which has a slack
of 0. Therefore, \( \ell_u = \ell_v \). If \( \ell_u, \ell_v \geq \ell \), the dual weights of \( u \) and \( v \) remain unchanged, and feasibility continues to hold. Otherwise, \( \ell_u, \ell_v \leq \ell \), and we have,

\[
y'(u) + y'(v) = y(u) - (\ell - \ell_u) + y(v) + (\ell - \ell_v) \\
= y(u) + y(v) - \ell_v + \ell_u \\
= y(u) + y(v) \\
= c(u, v),
\]

and the edge \((u, v)\) is feasible. \(\square\)

Finally, we argue that there is at least one augmenting path of admissible edges after the Hungarian search. From Corollary 3.10, all non-matching edges in the Dijkstra shortest path tree are admissible after performing dual adjustments. All matching edges are feasible and therefore, by definition, also admissible. Provided that the matching \( M \) is not maximum, the Dijkstra search will find at least one path from some \( b \in B_F \) to some \( a \in A_F \) in the shortest path tree, which is admissible.

### 3.4 The Gabow-Tarjan Algorithm

In this section, we present the Gabow-Tarjan scaling algorithm for minimum-cost maximum cardinality matching on bipartite graphs \([18]\). The algorithm executes in \( O(\log nC) \) scales, where the algorithm for each scale takes as input a bipartite graph \( G(A \cup B, E) \), \( |A| = |B| = n \) with optimal matching \( M^* \) whose cost is \( O(n) \). The algorithm for each scale returns a maximum-cardinality matching whose cost is within an additive error of \( n \) of the optimal, i.e., \( c(M) \leq c(M^*) + n \). This additive error is accounted for by scaling up the edge costs. Using the fact that the optimal matching has cost \( O(n) \), it can be shown that each scale
executes only $O(\sqrt{n})$ phases, with each phase taking $O(m)$ time, like the Hopcroft-Karp algorithm. This, in-turn, leads to an $O(m\sqrt{n}\log nC)$ running time.

The Gabow-Tarjan algorithm assumes that the input graph has a perfect matching, a matching in which all vertices are matched, and computes a minimum-cost perfect matching. However, they provide a simple reduction that allows the algorithm to work for the minimum-cost maximum cardinality case as well. We give our presentation under the assumption that the input graph has a perfect matching.

### 3.4.1 1-feasibility

A critical component of the Gabow-Tarjan (GT) algorithm’s analysis is its set of modified feasibility conditions. Like the Hungarian algorithm, the GT algorithm maintains a set of dual weights $y(\cdot)$ for the vertices of $A \cup B$. A matching $M$ and set of dual weights $y(\cdot)$ are 1-feasible if, for every edge $(u, v),$ 

$$y(u) + y(v) \leq c(u, v) + 1 \quad \text{if } (u, v) \notin M.$$  

$$y(u) + y(v) = c(u, v) \quad \text{if } (u, v) \in M.$$

These feasibility conditions are identical to those used in the Hungarian algorithm except the sum of dual weights for non-matching edges are allowed to exceed the edge cost by 1. Intuitively, the relaxation of the classical feasibility constraints by 1 introduces an error of at most 1 per edge of the optimal matching. This in turn means that any 1-feasible perfect matching is within an additive error of $n$ from the optimal. The addition of the ‘+1’ is important to bound the time taken by the algorithm, as we will see later.
3.4.2 Scaling Paradigm

The end goal of the Gabow-Tarjan algorithm is to obtain an optimal matching, not an approximate matching. This is achieved via scaling. At its very beginning, the algorithm can simply multiply all edge costs by $n + 1$. This ensures that the difference in cost between any optimal perfect matching and any sub-optimal matching is at least $n + 1$. As a result, any 1-feasible perfect matching must be optimal with respect to the scaled costs, so it suffices to compute a 1-feasible perfect matching. Note that the maximum edge cost with respect to the scaled costs becomes $O(nc^2)$.

To bound the number of phases by $O(\sqrt{n})$, the GT algorithm requires the assumption that the optimal matching has cost $O(n)$. To enforce this assumption, the algorithm uses a bit-scaling approach. The algorithm executes $O(\log nc)$ scales, where the $i$th scale uses only the most significant $i$ bits of all edge costs. For any edge $(u, v)$, let $c_i(u, v)$ be the cost of the edge during the $i$th scale. For the first scale, it is easy to see that the optimal matching cost is $O(n)$. During the $i$th scale, the GT algorithm computes a 1-feasible perfect matching $M_i$ and set of dual weights $y_i(\cdot)$. It uses the dual weights from scale $i+1$ to generate an initial set of 1-feasible dual weights for scale $i+1$ by setting, for every $v \in A \cup B$, $y_{i+1}(v) \leftarrow 2y_i(v) - 1$.

To ensure that the optimal matching for scale $i+1$ has a cost of $O(n)$, the algorithm uses the initial set of dual weights $y_{i+1}(\cdot)$ to compute reduced costs. The reduced cost of any edge $(u, v)$ is given by:

$$c'_{i+1}(u, v) = c_{i+1}(u, v) - y_{i+1}(u) - y_{i+1}(v).$$

Using the fact that the matching $M_i$ produced during the previous scale was 1-feasible with respect to $y_i(\cdot)$, it can be shown that the optimal matching with respect to the reduced costs $c_{i+1}(\cdot, \cdot)$ has cost $O(n)$. The algorithm proceeds to compute an 1-feasible matching $M_{i+1}$ and set of dual weights $y'_{i+1}(\cdot)$ with respect to the reduced costs $c'_{i+1}(\cdot, \cdot)$, using an
3.4. The Gabow-Tarjan Algorithm

$O(m\sqrt{n})$ time algorithm. Note that each reduced cost is at least $-1$. For every vertex $v \in A \cup B$, the algorithm sets $y_{i+1}(v) \leftarrow y_{i+1}(v) + y'_{i+1}(v)$. It is relatively straightforward to see that the matching $M_i$ and resulting set of dual weights $y_{i+1}(\cdot)$ are 1-feasible with respect to the original costs $c_{i+1}(\cdot)$ for scale $i$.

3.4.3 Algorithm for a Scale

It remains to describe the algorithm for computing a 1-feasible matching within each scale. Using the scaling paradigm, we can assume that the optimal matching $M^*$ has cost $c(M^*) = O(n)$, and each edge has a cost of at least $-1$. Initially, the algorithm assigns every vertex a dual weight of 0; since all edges are initially not in the matching, this initial dual assignment is 1-feasible. The structure of the GT algorithm for a scale is very similar to the Hopcroft-Karp algorithm. It executes $O(\sqrt{n})$ phases, where each phase has two stages.

**First Stage** The first stage consists of a Hungarian search. This Hungarian search is nearly identical to the procedure described in Section 3.3 except that the definitions of slack must be slightly modified to account for the additional ‘+1’ in the feasibility conditions, i.e.,

\[
s(u, v) = c(u, v) + 1 - y(u) - y(v) \quad \text{if } (u, v) \notin M.
\]
\[
s(u, v) = y(u) + y(v) - c(u, v) = 0 \quad \text{if } (u, v) \in M.
\]

Given this modification to the slack definition, the GT algorithm performs an identical dual weight adjustment to the Hungarian algorithm. We define the *admissible graph* $\mathcal{A}$ as the subgraph of the residual graph consisting of all the 0 slack edges. Using a very similar argument to the Hungarian algorithm, it can be shown that the dual adjustments performing by the Hungarian search stage do not violate 1-feasibility. Furthermore, there is at least 1
augmenting path in the admissible graph $A$.

**Second Stage** During the second stage, the algorithm executes a partial DFS search from every free vertex of $B$ in the admissible graph to find a maximal set of vertex-disjoint admissible augmenting paths $\mathcal{P}$, following a procedure identical to that used by the Hopcroft-Karp algorithm.

After computing the set of augmenting paths $\mathcal{P}$, the algorithm proceeds to augment along every path $P \in \mathcal{P}$. By doing so, any newly-entering matching edge $(a, b)$ become infeasible since $y(a) + y(b) = c(u, v) + 1$. To ensure that any such edge on $P$ remains feasible, the algorithm reduces the dual weight of every vertex $b \in B$ on $P$ by 1.

This completes the description of the GT algorithm for a phase. The algorithm for a scale repeatedly executes the algorithm for a phase until a perfect matching is obtained. Next, we give the analysis of this algorithm.

### 3.4.4 Efficiency of Algorithm

In this section, we bound the time taken by the GT algorithm for a scale as $O(m \sqrt{n})$. Thus, over all $O(\log nC)$ scales, the total time taken is $O(m \sqrt{n} \log nC)$. First, we consider the time taken per phase. A phase consists of two stages. The first stage executes a Hungarian search, which executes Dijkstra’s algorithm. Dijkstra’s algorithm takes $O(m + n \log n)$ time on an arbitrary graph by using a Fibonacci heap as a priority queue; however, using the fact that the maximum dual weight adjustment is $O(n)$, it is possible to replace the Fibonacci heap with an array of size $O(n)$ and perform bucket-sorting instead. Using this technique allows Dijkstra’s algorithm to be performed in $O(m)$ time for this situation. The second stage executes a DFS search identical to that done by the Hopcroft-Karp algorithm, taking
3.4. The Gabow-Tarjan Algorithm

$O(m)$ time. We conclude that the GT algorithm takes $O(m)$ time per phase.

It remains to bound the total number of phases. Using an argument similar to that of the Hopcroft-Karp algorithm, along with the assumption that the optimal matching cost is $O(n)$, it can be shown that the number of phases is $O(\sqrt{n})$. To show this, we first argue the following:

**Lemma 3.12.** At the end of phase $i$, the dual weight of every free vertex of $B$ is at least $i - 1$.

*Proof.* It is sufficient to argue that the dual weight of every free vertex of $B$ increases by at least 1 each phase, except for perhaps the first phase of the scale. The dual weights of free vertices of $B$ only change during the first stage's Hungarian search. This Hungarian search will increase the dual weight of every free vertex of $B$ if and only if no admissible augmenting path existed prior to the Hungarian search. Therefore, it suffices to argue that no admissible augmenting path exists at the end of a phase.

Assume for the sake of contradiction that there remains an admissible augmenting path $P$ at the end of some phase. The second stage of this phase found a maximal set of vertex-disjoint admissible augmenting paths $\mathcal{P}$ and augmented along them. If $P$ is disjoint from all paths in $\mathcal{P}$, then $P$ was also an admissible augmenting path prior to augmenting along the paths in $\mathcal{P}$, and $\mathcal{P}$ was not maximal, a contradiction. Therefore, $P$ must use at least one vertex $v$ of some $P' \in \mathcal{P}$. First, consider if $v \in B$. Then, since $v \in P'$, the dual weight of $v$ decreased by 1 when augmenting along $P'$, causing all incident non-matching edges to become inadmissible. Since all outgoing edges from $v$ are non-matching, $P$ must contain at least one inadmissible edge, a contradiction. Next, consider if $v \in A$. Then $v$ must be matched to a vertex $u \in B$ that is also on both $P$ and $P'$, and the only way to exit $v$ is via the newly-entered matching edge $(v, u)$. Therefore, $P$ must contain $u$, but we have already
shown this leads to a contradiction. We conclude that there is no admissible augmenting path $P$ at the end of a phase.

To bound the number of phases, we define the net-cost of an edge as,

$$
\phi(u, v) = \begin{cases} 
  c(u, v) + 1 & \text{if } (u, v) \not\in M; \\
  -c(u, v) & \text{if } (u, v) \in M.
\end{cases}
$$

The net-cost of any set of edges $S$ is given by the total net-cost of its edges, $\sum_{(u,v) \in S} \phi(u, v)$.

The GT algorithm maintains that all free vertices of $A$ have a dual weight of 0 and all free vertices of $B$ have a dual weight of $y_{\text{max}}$. From Lemma 3.12, we have that $y_{\text{max}} \geq i - 1$ after $i$ phases. As a result, the value of $y_{\text{max}}$ acts as a lower bound on the net-cost of the current minimum net-cost augmenting path, as shown in the following lemma.

**Lemma 3.13.** For any augmenting path $P$ with respect to $M$ from a free vertex $b \in B_F$ to a free vertex $a \in A_F$, we have $\phi(P) \geq y_{\text{max}}$.

**Proof.** We have,

$$
\phi(P) = \sum_{(u,v) \in P} \phi(u, v) \\
= \sum_{(u,v) \in P \setminus M} (c(u, v) + 1) - \sum_{(u,v) \in P \cap M} c(u, v) \\
\geq \sum_{(u,v) \in P \setminus M} y(u) + y(v) - \sum_{(u,v) \in P \cap M} y(u) + y(v) \quad (3.5) \\
= y(b) + y(a).
$$

The last inequality follows from the fact that, for every vertex $v$ on $P$ except for the first and last vertices, $v$ participates in exactly one non-matching edge of $P$ and exactly one matching
edge of \( P \). The endpoints \( b \) and \( a \) participate in exactly one non-matching edge. The lemma follows from the fact that \( y(b) = y_{\text{max}} \) and \( y(a) = 0 \).

\[ \square \]

**Corollary 3.14.** For any alternating cycle \( C \) with respect to a matching \( M \), \( \phi(C) \geq 0 \).

**Proof.** The claim follows from equation (3.5) and the fact that all vertices of an alternating cycle have exactly one incident edge of both \( C \cap M \) and \( C \setminus M \). \[ \square \]

Next, we relate the value of \( y_{\text{max}} \) to the number of free vertices of \( B \) remaining.

**Lemma 3.15.** At any point in the algorithm, for a phase, let \( |B_F| \) be the number of free vertices of \( B \) remaining. Then \( y_{\text{max}}|B_F| = O(n) \).

**Proof.** Let \( M^* \) be an optimal perfect matching, and let \( M \) be the current matching maintained by the algorithm. The symmetric difference \( S = M \oplus M^* \) consists of alternating cycles and augmenting paths with respect to \( M \). The net-cost of these edges with respect to \( M \) is given by \( \phi(S) = c(M^*) - c(M) + |S \setminus M| \). From our assumption on the input for a scale, \( c(M^*) = O(n) \) and \( c(M) \geq -n \). Also, \( |S \setminus M| = O(n) \). Therefore, \( \phi(S) = O(n) \). From Corollary 3.14, the total net-costs of the cycles in \( S \) is at least 0. Therefore, the augmenting paths in \( S \) have a total net-cost of \( O(n) \). From Lemma 3.13, each of these augmenting paths has a net-cost of at least \( y_{\text{max}} \). Therefore, since the number of augmenting paths in \( S \) is equal to \( |B_F| \), we have \( y_{\text{max}}|B_F| = O(n) \). \[ \square \]

Using Lemma 3.15, it is easy to see that the number of phases executed by the algorithm is \( O(\sqrt{n}) \). First, observe that \( y_{\text{max}} \) increases by at least one every phase (except possibly the first). Second, observe that \( |B_F| \) decreases by at least one every phase until the matching is perfect. After \( \Theta(\sqrt{n}) \) phases, then we have \( y_{\text{max}} = \Omega(\sqrt{n}) \), \( |B_F| = O(\sqrt{n}) \), and the remaining vertices are matched using only \( O(\sqrt{n}) \) phases. This implies that the algorithm
terminates in only $O(\sqrt{n})$ phases. Recall that each phase takes $O(m)$ time, so the total time taken per scale over all phases is $O(m\sqrt{n})$. Over all $O(\log nC)$ scales, the total time is $O(m\sqrt{n}\log nC)$.

### 3.4.5 Concerning the Length of Augmenting Paths

For some applications, it is useful to bound the total length of all augmenting paths computed by a scale of the GT algorithm. This quantity is not necessary to bound the time taken by the above GT matching algorithm, although it is a necessary component of the time complexity of Gabow and Tarjan’s generalization to the minimum-cost transportation problem \[18\].

Next, we bound the total length of all the augmenting paths computed by the GT algorithm.

**Lemma 3.16.** Let $P_j$ be the $j$th augmenting path computed by the algorithm, then $\sum_{j=1}^{n} |P_j| = O(n \log n)$.

**Proof.** First, observe that the total net-cost of all augmenting paths can be written as,

$$\sum_{j=1}^{n} \phi(P_j) = \sum_{j=1}^{n} (c(M_j) - c(M_{j-1}) + \lceil|P_j|/2\rceil) = c(M_n) - c(M_0) + \sum_{j=1}^{n} \lceil|P_j|/2\rceil.$$

Since $c(M_n) \geq c(M_0)$, we can conclude that the total length of the augmenting paths is upper bounded by the total net-cost of the augmenting paths, i.e., $\sum_{j=1}^{n} |P_j| = O(\sum_{j=1}^{n} \phi(P_j))$.

Therefore, it suffices to bound the total net-cost.

Using a calculation similar to 3.13, we have that any admissible augmenting path from a free vertex $b \in B_F$ to a free vertex $a \in A_F$ has a net-cost exactly equal to $y(b)$. Let $y_{max}^i$ be the value of $y_{max}$ when $P_j$ was found; then, $\phi(P_j) = y_{max}^i$. From Lemma 3.15, we have
The maximum error $y^j_{max}(n - j + 1) = O(n)$. Summing over all paths gives,

$$\sum_{j=1}^{n} \phi(P_j) = O(n) \cdot \sum_{j=1}^{n} \frac{1}{(n - j + 1)} = O(n) \cdot \sum_{j=1}^{n} \frac{1}{j} = O(n \log n).$$

The last inequality follows from the fact that the $n$th harmonic number is $O(\log n)$.

This analysis of the augmenting path lengths is particularly useful for our purposes. We observe that the ‘+1’ error is absolutely critical for bounding the augmenting path lengths. In a sense, it forces the algorithm to simultaneously prioritize augmenting paths that are small in both length and cost. Using the fact that the optimal matching has cost $O(n)$, the total error of all augmenting paths is $O(n \log n)$. The total length of the augmenting paths then follows from the fact that each vertex of every augmenting path contributes $\Theta(1)$ error to the $O(n \log n)$ budget.

For our purposes regarding matching problems on small-separator graphs, consider what would happen if we assign a much larger error of say $+k$ to some $O(n/k)$ vertices instead of the normal $+1$ error. Then the total error would still be $O(n)$, but as an additional side-effect, the vertices with $+k$ error would be used in significantly fewer augmenting paths.

Using arguments similar to those of Lemma 3.16, one can show that these vertices with $+k$ error would be used only $O(n \log(n)/k)$ times among all augmenting paths computed. This observation forms a critical component of our separator-based matching algorithms; by assigning a larger additive error to boundary vertices between pieces, we can ensure that the augmenting paths intersect a relatively small number of pieces in total.
3.5 Graph Separators

In this section, we define material relevant to graph separators. First, we formally define the notion of an $n^\delta$ separator property. Given a graph $G(V, E)$, a separator is any subset of vertices $S \subseteq V$ whose removal disconnects the graph into two disjoint subgraphs. These subgraphs correspond to the vertex-induced subgraphs of vertex sets $V_1$ and $V_2$, where $V = V_1 \cup S \cup V_2$ and there are no edges in $V_1 \times V_2$. A separator $S$ is balanced if both $V_1$ and $V_2$ have size at least $O(|V|/c)$ for some constant $c \geq 2$. A graph has an $n^\delta$ separator property if, for every subgraph $G'(V', E')$ of $G$, the graph $G'$ has a balanced separator of size $O(|V'|^\delta)$.

We assume that such a separator is efficiently computable on all subgraphs, meaning the time taken to compute separators does not pose a performance bottleneck.

Several types of graphs have $n^\delta$ separator properties. The most canonical example is the class of planar graphs, which have an $n^{1/2}$ separator property. A foundational result by Lipton and Tarjan gave an $O(n)$ time algorithm to compute a separator of size $O(\sqrt{n})$ whose removal disconnects the graph into two subgraphs each containing at most $2n/3$ vertices [37].

Minor-free graphs also have separator properties. Kawarabayashi and Reed showed that, for $h = O(1)$, $K_h$-minor free graphs have a separator of size $O(\sqrt{n})$ [25]. This implies that such graphs have an $n^{1/2}$ separator property, like planar graphs. They showed how to compute such a separator in $O(n^{1+\varepsilon})$ time for an arbitrarily small $\varepsilon > 0$. However, the dependency on $h$ in their algorithm’s running time is enormous; it is a power-tower of $h$ whose height itself is a function of $h$ [59]. From a more practical standpoint, one may be interested in computing separators with a polynomial dependency on $h$. Wulff-Nilsen [59] gave an algorithm with a polynomial dependency on $h$ that runs in $O(n^{5/4+\varepsilon})$ time and computes a slightly worse separator of size $O(h\sqrt{n\log n})$, which, within logarithmic terms, is good enough for most applications.
For many algorithms, it is useful to apply separator theorems recursively to decompose the graph into smaller pieces, with each containing few boundary vertices. In planar graphs, for any reasonable value of $r$, it is possible to compute in $\tilde{O}(n)$ time a division of the edges of the graphs into edge-disjoint pieces so that the following conditions are satisfied:

- The total number of pieces is $O(n/r)$.
- Each piece has $O(r)$ vertices (and therefore also $O(r)$ edges).
- Each piece has $O(\sqrt{r})$ boundary vertices, which are vertices shared between multiple pieces.
- The total number of boundary vertices among all pieces is $O(n/\sqrt{r})$.

Such a division of a planar graph into pieces is called an $r$-division. These $r$-divisions lend themselves well to divide-and-conquer style schemes and have been used in a variety of algorithms for planar graphs. There is a generalization of an $r$-division to $K_h$-minor free graphs described by Wulff-Nilsen called an $r$-clustering, which differs from an $r$-division in that it allows for as many as $\tilde{O}(n/\sqrt{r})$ pieces and has additional logarithmic factors. For our purposes, an $r$-clustering is nearly identical to an $r$-division.

Finally, we can generalize the concept of recursive decomposition to any class of graphs with an $n^\delta$ separator property. By recursively computing separators to split the graph until each piece has $O(r)$ vertices and $O(mr/n)$ edges, it is possible to obtain a set of pieces such that the total number of boundary vertices is $O(n/r^{1-\delta})$. We use this technique in Chapter 4.
3.6 The Lipton-Tarjan Algorithm

In this section, we describe the separator-based matching algorithm of Lipton and Tarjan [38]. This algorithm is a good example of how separators can be used to facilitate a divide and conquer approach. We summarize the approach for computing a maximum cardinality matching; an analogous approach works for the minimum-cost maximum-cardinality matching problem as well. This algorithm works on any graph $G$ with a recursively-computable balanced separator of size $O(n^\delta)$. For simplicity in exposition, we assume that, for every subgraph $G'$ of $G$, the number of edges in $G'$ is linear in the number of vertices. Given a graph $G(V, E)$, the algorithm finds a separator set $S$ of size $O(n^\delta)$ whose removal disconnects $G$ into disjoint subgraphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$. The algorithm computes a maximum-cardinality matching $M_1 \subseteq E_1$ and $M_2 \subseteq E_2$ within each of these subgraphs recursively. Afterwards, let $M = M_1 \cup M_2$ be the resulting matching; then the difference in cardinality between $M$ and the maximum-cardinality matching $M^*$ is at most $|S| = O(n^\delta)$ because at most $|S|$ edges of $M^*$ are incident on some vertex of $S$. The remaining $|S|$ vertices are then matched by finding a single augmenting path at a time (i.e., using an iteration of the Ford-Fulkerson algorithm), resulting in a total time of $O(mn^\delta) = O(n^{1+\delta})$. Since the separator $S$ is balanced, the recurrence for the algorithm can be expressed as:

$$T(n) = T(c_1 n) + T(c_2 n) + O(n^{1+n^{\delta}}),$$

where $c_1 + c_2 = 1$ and $c_1, c_2 \geq 1/c$ for some constant $c \geq 2$. In evaluating this recurrence, the largest subproblem’s time dominates, resulting in a total running time of $O(n^{1+\delta})$ for the algorithm. This algorithm can be extended to the minimum-cost maximum-cardinality matching problem in a straight-forward fashion by using the Hungarian algorithm instead of the Ford-Fulkerson each iteration. This results in a total execution time of $O(n^{1+\delta} \log n)$. 
A Weighted Approach to Maximum Cardinality Bipartite Matching

We present a weighted approach to compute a maximum cardinality matching in an arbitrary bipartite graph. Our main result is a new algorithm that takes as input a weighted bipartite graph $G(A \cup B, E)$ with edge weights of 0 or 1. Let $w \leq n$ be an upper bound on the weight of any matching in $G$. Consider the subgraph induced by all the edges of $G$ with a weight 0. Suppose every connected component in this subgraph has $O(r)$ vertices and $O(mr/n)$ edges. We present an algorithm to compute a maximum cardinality matching in $G$ in $\tilde{O}(m(\sqrt{w} + \sqrt{r} + wr/n))$ time.\(^1\)

When all the edge weights are 1 (symmetrically when all weights are 0), our algorithm will be identical to the well-known Hopcroft-Karp (HK) algorithm, which runs in $O(m\sqrt{n})$ time. However, if we can carefully assign weights of 0 and 1 on its edges such that both $w$ and $r$ are sub-linear in $n$ and $wr = O(n^\gamma)$ for $\gamma < 3/2$, then we can compute a maximum cardinality matching in $o(m\sqrt{n})$ time. Using our algorithm, we obtain a new $\tilde{O}(n^{4/3}/\varepsilon^4)$ time algorithm to compute an $(1+\varepsilon)$-approximate bottleneck matching of $A, B \subset \mathbb{R}^2$ and an $\frac{1}{\varepsilon^{o(\varepsilon)}} n^{1+\frac{d-1}{2\varepsilon}} \text{poly log } n$ time algorithm for computing $(1+\varepsilon)$-approximate bottleneck matching in $d$-dimensions. All previous algorithms take $\Omega(n^{3/2})$ time.

Our algorithm also applies to any graph $G(A \cup B, E)$ that has an easily computable balanced

\(^1\)We use $\tilde{O}$ to suppress poly-logarithmic terms.
vertex separator of size $|V'| \delta$, for every subgraph $G'(V', E')$ where $\delta \in [1/2, 1)$. By applying our algorithm, we can compute a maximum matching in $\tilde{O}(mn^{4\delta})$ time improving upon the $O(m\sqrt{n})$ time taken by the HK-Algorithm.

4.1 Preliminaries

We are given a bipartite graph $G(A \cup B, E)$, where any edge $(a, b) \in E$ has a weight $c(a, b)$ of 0 or 1. Given a matching $M$, a vertex is free if it is not matched in $M$. An alternating path (resp. cycle) is a simple path (resp. cycle) that alternates between edges in $M$ and not in $M$. An augmenting path is an alternating path that begins and ends at a free vertex.

A matching $M$ and an assignment of dual weights $y(\cdot)$ on the vertices of $G$ is feasible if for any edge $(a, b) \in (A \times B) \cap E$:

\begin{align}
y(b) - y(a) &\leq c(a, b) \quad \text{if } (a, b) \not\in M, \quad (4.1) \\
y(a) - y(b) &= c(a, b) \quad \text{if } (a, b) \in M. \quad (4.2)
\end{align}

To assist in describing our algorithm, we first define a residual network and an augmented residual network with respect to a feasible matching $M, y(\cdot)$. A residual network $G_M$ with respect to a feasible matching $M$ is a directed graph where every edge $(a, b)$ is directed from $b$ to $a$ if $(a, b) \not\in M$ and from $a$ to $b$ if $(a, b) \in M$. The weight $s(a, b)$ of any edge is given by the slack of this edge with respect to feasibility conditions (4.1) and (4.2), i.e., if $(a, b) \not\in M$, then $s(a, b) = c(a, b) + y(a) - y(b)$ and $s(a, b) = 0$ otherwise. An augmented residual network is obtained by adding to the residual network an additional vertex $s$ and additional directed edges from $s$ to every vertex in $B_F$, each of having a weight of 0. We denote the augmented residual network as $G'_M$. 
4.2 Our Algorithm

Throughout this section we will use $M$ to denote the current matching maintained by the algorithm and $A_F$ and $B_F$ to denote the vertices of $A$ and $B$ that are free with respect to $M$. Initially $M = \emptyset$, $A_F = A$, and $B_F = B$. Our algorithm consists of two steps. The first step, which we refer to as the preprocessing step, will execute the Hopcroft-Karp algorithm and compute a maximum matching within every piece. Any maximum matching $M_{\text{Opt}}$ has at most $w$ edges with a weight of 1 and the remaining edges have a weight of 0. Therefore, $|M_{\text{Opt}}| - |M| \leq w$. The time taken by the preprocessing step for $K_i$ is $O(|E_i|\sqrt{|V_i|}) = O(|E_i|\sqrt{r})$. Since the pieces are vertex disjoint, the total time taken across all pieces is $O(m\sqrt{r})$. After this step, no augmenting path with respect to $M$ is completely contained within a single piece. We set the dual weight $y(v)$ of every vertex $v \in A \cup B$ to 0. The matching $M$ along with the dual weights $y(\cdot)$ satisfies (4.1) and (4.2) and is feasible.

The second step of the algorithm is executed in phases. We describe phase $k$ of the algorithm. This phase consists of two stages.

**First Stage** In the first stage, we construct the augmented residual network $G'_M$ and execute Dijkstra’s algorithm with $s$ as the source. Let $\ell_v$ for any vertex $v$ denote the shortest path distance from $s$ to $v$ in $G'_M$. If a vertex $v$ is not reachable from $s$, we set $\ell_v$ to $\infty$. Let

$$\ell = \min_{v \in A_F} \ell_v. \quad (4.3)$$

Suppose $M$ is a perfect matching or $\ell = \infty$, then this algorithm returns with $M$ as a maximum matching. Otherwise, we update the dual weight of any vertex $v \in A \cup B$ as follows. If $\ell_v \geq \ell$, we leave its dual weight unchanged. Otherwise, if $\ell_v < \ell$, we set $y(v) \leftarrow y(v) + \ell - \ell_v$. After updating the dual weights, we construct the admissible graph which consists of a subset
of edges in the residual network $G_M$ that have zero slack. After the first stage, the matching $M$ and the updated dual weights are feasible. Furthermore, there is at least one augmenting path in the admissible graph. This completes the first stage of the phase.

**Second Stage** In the second stage, we initialize $G'$ to be the admissible graph and execute DFS to identify augmenting paths. For any augmenting path $P$ found during the DFS, we refer to the pieces that contain its edges as affected pieces of $P$.

Similar to the HK-Algorithm, the second stage of this phase will initiate a DFS from every free vertex $b \in B_F$ in $G'$. If the DFS does not lead to an augmenting path, we delete all edges that were visited by the DFS. On the other hand, if the DFS finds an augmenting path $P$, then the matching is augmented along $P$, all edges that are visited by the DFS and do not lie in an affected piece of $P$ are deleted, and the DFS initiated at $b$ will terminate.

Now, we describe in detail the DFS initiated for a free vertex $b \in B_F$. Initially $P = \langle b = v_1 \rangle$. Every edge of $G'$ is marked unvisited. At any point during the execution of DFS, the algorithm maintains a simple path $P = \langle b = v_1, v_2, \ldots, v_k \rangle$. The DFS search continues from the last vertex of this path as follows:

- If there are no unvisited edges that are going out of $v_k$ in $G'$,
  - If $P = \langle v_1 \rangle$, remove all edges that were marked as visited from $G'$ and terminate the execution of DFS initiated at $b$.
  - Otherwise, delete $v_k$ from $P$ and continue the DFS search from $v_{k-1}$.

- If there is an unvisited edge going out of $v_k$, let $(v_k, v)$ be this edge. Mark $(v_k, v)$ as visited. If $v$ is on the path $P$, continue the DFS from $v_k$. If $v$ is not on the path $P$, add $(v_k, v)$ to $P$, set $v_{k+1}$ to $v$, and,
4.2. Our Algorithm

- Suppose \( v \in A_F \), then \( P \) is an augmenting path from \( b \) to \( v \). Execute the Augment procedure which augments \( M \) along \( P \). Delete from \( G' \) every visited edge that does not belong to any affected piece of \( P \) and terminate the execution of DFS initiated at \( b \).

- Otherwise, \( v \in (A \cup B) \setminus A_F \). Continue the DFS from \( v_{k+1} \).

The Augment procedure receives a feasible matching \( M \), a set of dual weights \( y(\cdot) \), and an augmenting path \( P \) as input. For any \( (b, a) \in P \setminus M \), where \( a \in A \) and \( b \in B \), set \( y(b) \leftarrow y(b) - 2c(a, b) \). Then augment \( M \) along \( P \) by setting \( M \leftarrow M \oplus P \). By doing so, every edge of \( M \) after augmentation satisfies the feasibility condition (4.2). This completes the description of our algorithm. The algorithm maintains the following invariants during its execution:

(I1) The matching \( M \) and the set of dual weights \( y(\cdot) \) are feasible. Let \( y_{\text{max}} = \max_{v \in B} y(v) \).

The dual weight of every vertex \( v \in B_F \) is \( y_{\text{max}} \) and the dual weight for every vertex \( v \in A_F \) is 0.

(I2) For every phase that is fully executed prior to obtaining a maximum matching, at least one augmenting path is found and the dual weight of every free vertex of \( B_F \) increases by at least 1.

Comparison with the GT-Algorithm In the GT-Algorithm, the admissible graph does not have any alternating cycles. Also, every augmenting path edge can be shown to not participate in any future augmenting paths that are computed in the current phase. By using these facts, one can show that the edges visited unsuccessfully by a DFS will not lead to an augmenting path in the current phase. In our case, however, admissible cycles can exist. Also, some edges on the augmenting path that have zero weight remain admissible.
after augmentation and may participate in another augmenting path in the current phase. We show, however, that any admissible cycle must be completely inside a piece and cannot span multiple pieces (Lemma 4.2). Using this fact, we show that edges visited unsuccessfully by the DFS that do not lie in an affected piece will not participate in any more augmenting paths (Lemma 4.7 and Lemma 4.9) in the current phase. Therefore, we can safely delete them.

**Correctness** From Invariant (I2), each phase of our algorithm will increase the cardinality of $M$ by at least 1 and so, our algorithm terminates with a maximum matching.

**Efficiency** We use the following notations to bound the efficiency of our algorithm. Let \{\(P_1, \ldots, P_t\}\} be the \(t\) augmenting paths computed in the second step of the algorithm. Let \(K_i\) be the set of affected pieces with respect to the augmenting path \(P_i\). Let \(M_0\) be the matching at the end of the first step of the algorithm. Let, for \(1 \leq i \leq t\), \(M_i = M_{i-1} \oplus P_i\), i.e., \(M_i\) is the matching after the \(i\)th augmentation in the second step of the algorithm.

The first stage is an execution of Dijkstra’s algorithm which takes \(O(m + n \log n)\) time. Suppose there are \(\lambda\) phases; then the cumulative time taken across all phases for the first stage is \(O(\lambda m + \lambda n \log n)\). In the second stage, each edge visited by a DFS is discarded for the remainder of the phase, provided it is not in an affected piece. Since each affected piece has \(O(mr/n)\) edges, the total time taken by all the DFS searches across all the \(\lambda\) phases is bounded by \(O((m + n \log n)\lambda + (mr/n) \sum_{i=1}^{t} |K_i|)\). In Lemma 4.3, we bound \(\lambda\) by \(\sqrt{w}\) and \(\sum_{i=1}^{t} |K_i|\) by \(O(w \log w)\). Therefore, the total time taken by the algorithm including the time taken by preprocessing step is \(O(m\sqrt{r} + m\sqrt{w} + n\sqrt{w} \log n + \frac{mrw \log w}{n}) = \tilde{O}(m(\sqrt{w} + \sqrt{r} + \frac{w}{n})).\)

**Lemma 4.1.** For any feasible matching \(M, y(\cdot)\) maintained by the algorithm, let \(y_{\text{max}}\) be the dual weight of every vertex of \(B_F\). For any augmenting path \(P\) with respect to \(M\) from a free
4.2. Our Algorithm

vertex \( u \in B_F \) to a free vertex \( v \in A_F \),

\[
c(P) = y_{\text{max}} + \sum_{(a,b) \in P} s(a,b).
\]

Proof. The weight of \( P \) is

\[
c(P) = \sum_{(a,b) \in P} c(a,b) = \sum_{(a,b) \in P \setminus M} (y(b) - y(a) + s(a,b)) + \sum_{(a,b) \in P \cap M} (y(a) - y(b)).
\]

Since every vertex on \( P \) except for \( u \) and \( v \) participates in one edge of \( P \cap M \) and one edge of \( P \setminus M \), we can write the above equation as

\[
c(P) = y(u) - y(v) + \sum_{(a,b) \in P \setminus M} s(a,b) = y(u) - y(v) + \sum_{(a,b) \in P} s(a,b).
\]

The last equality follows from the fact that edges of \( P \cap M \) satisfy (4.2) and have a slack of zero. From (I1), we get that \( y(u) = y_{\text{max}} \) and \( y(v) = 0 \), which gives,

\[
c(P) = y_{\text{max}} + \sum_{(a,b) \in P} s(a,b).
\]

\[\square\]

Lemma 4.2. For any feasible matching \( M \), \( y(\cdot) \) maintained by the algorithm, and for any alternating cycle \( C \) with respect to \( M \), if \( c(C) > 0 \), then

\[
\sum_{(a,b) \in P} s(a,b) > 0,
\]

i.e., \( C \) is not a cycle in the admissible graph.
Proof. The claim follows from (4.4) and the fact that the first vertex $u$ and the last vertex $v$ in a cycle are the same.

Lemma 4.3. The total number of phases is $O(\sqrt{w})$ and the total number of affected pieces is $O(w \log w)$, i.e., $\sum_{i=1}^{t} |K_i| = O(w \log w)$.

Proof. Let $M_{\text{Opt}}$ be a maximum matching, which has weight at most $w$. Consider any phase $k$ of the algorithm. By (I2), the dual weight $y_{\text{max}}$ of every free vertex in $B_F$ is at least $k$. The symmetric difference of $M$ and $M_{\text{Opt}}$ will contain $j = |M_{\text{Opt}}| - |M|$ vertex-disjoint augmenting paths. Let $\{P_1, \ldots, P_j\}$ be these augmenting paths. These paths contain edges of $M_{\text{Opt}}$ and $M$, both of which are of weight at most $w$. Therefore, the sum of weights of these paths is

$$\sum_{i=1}^{j} c(P_i) \leq 2w.$$ 

Let $y_{\text{max}}$ be the dual weight of every vertex $b$ of $B$ that is free with respect to $M$, i.e., $b \in B_F$. From (I2), $y_{\text{max}} \geq k$. From Lemma 4.1 and the fact that the slack on every edge is non-negative, we immediately get,

$$2w \geq \sum_{i=1}^{j} c(P_i) \geq jy_{\text{max}} \geq jk. \quad (4.5)$$

When $\sqrt{w} \leq k < \sqrt{w} + 1$, it follows from the above equation that $j = |M_{\text{Opt}}| - |M| \leq 2\sqrt{w}$. From (I2), we will compute at least one augmenting path in each phase and so the remaining $j$ unmatched vertices are matched in at most $2\sqrt{w}$ phases. This bounds the total number of phases by $3\sqrt{w}$.

Recollect that $\{P_1, \ldots, P_t\}$ are the augmenting paths computed by the algorithm. The matching $M_0$ has $|M_{\text{Opt}}| - t$ edges. Let $y_{\text{max}}^t$ correspond to the dual weight of the free vertices of $B_F$ when the augmenting path $P_i$ is found by the algorithm. From Lemma 4.1, and the fact
that $P_l$ is an augmenting path consisting of zero slack edges, we have $y_{\text{max}}^l = c(P_l)$. Before augmenting along $P_l$, there are $|M_{\text{Opt}}| - t + l - 1$ edges in $M_{l-1}$ and $j = |M_{\text{Opt}}| - |M_{l-1}| = t - l + 1$. Plugging this in to (4.5), we get $c(P_l) = y_{\text{max}}^l \leq \frac{2w}{t-l+1}$. Summing over all $1 \leq l \leq t$, we get,

$$
\sum_{l=1}^{t} c(P_l) \leq w \sum_{l=1}^{t} \frac{2}{t-l+1} = O(w \log t) = O(w \log w). \quad (4.6)
$$

For any augmenting path $P_l$, the number of affected pieces is upper bounded by one plus the number of non-zero weight edges on $P_l$, i.e., $|K_l| \leq c(P_l) + 1$. Therefore,

$$
\sum_{l=1}^{t} |K_l| \leq \sum_{l=1}^{t} (c(P_l) + 1) = O(w \log w).
$$

\[\square\]

### 4.3 Proof of Invariants

We now prove (I1) and (I2). Consider any phase $k$ in the algorithm. Assume inductively that at the end of phase $k - 1$, (I1) and (I2) hold. We will show that (I1) and (I2) also hold at the end of the phase $k$. We establish a lemma that will help us prove (I1) and (I2).

**Lemma 4.4.** For any edge $(a, b) \in M$, let $\ell_a$ and $\ell_b$ be the distances returned by Dijkstra’s algorithm during the first stage of phase $k$, then $\ell_a = \ell_b$.

**Proof.** The only edge directed towards $b$ is an edge from its match $a$. Therefore, any path from $s$ to $b$ in the augmented residual network, including the shortest path, should pass through $a$. Since the slack on any edge of $M$ is 0, $\ell_b = \ell_a + s(a, b) = \ell_a$. \[\square\]

**Lemma 4.5.** Any matching $M$ and dual weights $y(\cdot)$ maintained during the execution of the algorithm are feasible.
Chapter 4. A Weighted Approach to Maximum Cardinality Bipartite Matching

Proof. We begin by showing that the dual weight modifications in the first stage of phase $k$ will not violate dual feasibility conditions (4.1) and (4.2). Let $\tilde{y}(\cdot)$ denote the dual weights after the execution of the first stage of the algorithm. Consider any edge $(u, v)$ directed from $u$ to $v$. There are the following possibilities:

If both $\ell_u$ and $\ell_v$ are greater than or equal to $\ell$, then $y(u)$ and $y(v)$ remain unchanged and the edge remains feasible.

If both $\ell_u$ and $\ell_v$ are less than $\ell$, suppose $(u, v) \in M$. Then, from Lemma 4.4, $\ell_u = \ell_v$. We have, $\tilde{y}(u) = y(u) + \ell - \ell_u$, $\tilde{y}(v) = y(v) + \ell - \ell_v$, and $\tilde{y}(u) - \tilde{y}(v) = y(u) - y(v) + \ell_v - \ell_u = c(u, v)$ implying $(u, v)$ satisfies (4.2).

If $\ell_u$ and $\ell_v$ are less than $\ell$ and $(u, v) \notin M$, then $u \in B$ and $v \in A$. By definition, $y(u) - y(v) + s(u, v) = c(u, v)$. By the properties of shortest paths, for any edge $(u, v)$, $\ell_v - \ell_u \leq s(u, v)$. The dual weight of $u$ is updated to $y(u) + \ell - \ell_u$ and dual weight of $v$ is updated to $y(v) + \ell - \ell_v$. The difference in the updated dual weights $\tilde{y}(u) - \tilde{y}(v) = (y(u) + \ell - \ell_u) - (y(v) + \ell - \ell_v) = y(u) - y(v) + \ell_v - \ell_u \leq y(u) - y(v) + s(u, v) = c(u, v)$. Therefore, $(u, v)$ satisfies (4.1).

If $\ell_u < \ell$ and $\ell_v \geq \ell$, then, from Lemma 4.4, $(u, v) \notin M$, and so $u \in B$ and $v \in A$. From the shortest path property, for any edge $(u, v)$, $\ell_v - \ell_u \leq s(u, v)$. Therefore,

$$\tilde{y}(u) - \tilde{y}(v) = y(u) - y(v) + \ell - \ell_u \leq y(u) - y(v) + \ell_v - \ell_u \leq y(u) - y(v) + s(u, v) = c(u, v),$$

implying $(u, v)$ satisfies (4.1).

If $\ell_u \geq \ell$ and $\ell_v < \ell$, then, from Lemma 4.4, $(u, v) \notin M$, and so $u \in B$ and $v \in A$. Since $\ell_v < \ell$, we have,

$$\tilde{y}(u) - \tilde{y}(v) = y(u) - y(v) - \ell + \ell_v < y(u) - y(v) \leq c(u, v),$$
implying \((u, v)\) satisfies \((4.1)\).

In the second stage of the algorithm, when an augmenting path \(P\) is found, the dual weights of some vertices of \(B\) on \(P\) decrease and the directions of edges of \(P\) change. We argue these operations do not violate feasibility. Let \(\tilde{y}(\cdot)\) be the dual weights after these operations. Consider any edge \((a, b)\) where \(a \in A\) and \(b \in B\). If \(b\) is not on \(P\), then the feasibility of \((a, b)\) is unchanged. If \(b\) is on \(P\) and \(a\) is not on \(P\), then \(\tilde{y}(b) \leq y(b)\), and \(\tilde{y}(b) - \tilde{y}(a) \leq y(b) - y(a) \leq c(a, b)\), implying \((4.1)\) holds. The remaining case is when both \(a\) and \(b\) are on \(P\). Consider if \((a, b) \in M\) after augmentation. Prior to augmentation, \((a, b)\) was an admissible edge not in \(M\), and we have \(y(b) - y(a) = c(a, b)\) and \(\tilde{y}(b) = y(b) - 2c(a, b)\). So, \(\tilde{y}(a) - \tilde{y}(b) = y(a) - (y(b) - 2c(a, b)) = y(a) - y(b) + 2c(a, b) = c(a, b)\), implying \((4.2)\) holds. Finally, consider if \((a, b) \notin M\) after augmentation. Then, prior to augmentation, \((a, b)\) was in \(M\), and \(y(a) - y(b) = c(a, b)\). So, \(\tilde{y}(b) - \tilde{y}(a) \leq y(b) - y(a) = -c(a, b) \leq c(a, b)\), implying \((4.1)\) holds. We conclude the second stage maintains feasibility.

Next we show that the dual weights \(A_F\) are zero and the dual weights of all vertices of \(B_F\) are equal to \(y_{\text{max}}\). At the start of the second step, all dual weights are 0. During the first stage, the dual weight of any vertex \(v\) will increase by \(\ell_v - \ell_v\) only if \(\ell_v < \ell\). By \((4.3)\), for every free vertex \(a \in A_F\), \(\ell_a \geq \ell\), and so the dual weight of every free vertex of \(A\) remains unchanged at 0. Similarly, for any free vertex \(b \in B_F\), \(\ell_b = 0\), and the dual weight increases by \(\ell\), which is the largest possible increase. This implies that every free vertex in \(B_F\) will have the same dual weight of \(y_{\text{max}}\). In the second stage, matched vertices of \(B\) undergo a decrease in their dual weights, which does not affect vertices in \(B_F\). Therefore, the dual weights of vertices of \(B_F\) will still have a dual weight of \(y_{\text{max}}\) after stage two. This completes the proof of (I1).

Before we prove (I2), we will first establish a property of the admissible graph after the dual weight modifications in the first stage of the algorithm.
**Lemma 4.6.** After the first stage of each phase, there is an augmenting path consisting of admissible edges.

**Proof.** Let \( a \in A_F \) be a free vertex whose shortest path distance from \( s \) in the augmented residual network is \( \ell \), i.e., \( \ell_a = \ell \). Let \( P \) be the shortest path from \( s \) to \( a \) and let \( P_a \) be the path \( P \) with \( s \) removed from it. Note that \( P_a \) is an augmenting path. We will show that after the dual updates in the first stage, every edge of \( P_a \) is admissible. Consider any edge \((u, v) \in P_a \cap M\), where \( u \in A \) and \( v \in B \). From Lemma 4.4, \( \ell_u = \ell_v \). Then the updated dual weights are \( \tilde{y}(u) = y(u) + \ell - \ell_u \) and \( \tilde{y}(v) = y(v) + \ell - \ell_v \). Therefore, \( \tilde{y}(u) - \tilde{y}(v) = y(u) - y(v) - \ell_u + \ell_v = c(u, v) \), and \((u, v)\) is admissible. Otherwise, consider any edge \((u, v) \in P_a \setminus M\), where \( u \in B \) and \( v \in A \). From the optimal substructure property of shortest paths, for any edge \((u, v) \in P_a \) directed from \( u \) to \( v \), \( \ell_v - \ell_u = s(u, v) \). Therefore, the difference of the new dual weights is

\[
\tilde{y}(u) - \tilde{y}(v) = y(u) + \ell - \ell_u - y(v) - \ell + \ell_v = y(u) - y(v) - \ell_u + \ell_v = y(u) - y(v) + s(u, v) = c(u, v),
\]

implying that \((u, v)\) is admissible. \(\square\)

**Proof of (I2)** From Lemma 4.6, there is an augmenting path of admissible edges at the end of the first stage of any phase. Since we execute a DFS from every free vertex \( b \in B_F \) in the second stage, we are guaranteed to find an augmenting path. Next, we show in Corollary 4.10 that there is no augmenting path of admissible edges at the end of stage two of phase \( k \), i.e., all augmenting paths in the residual network have a slack of at least 1. This will immediately imply that the first stage of phase \( k + 1 \) will have to increase the dual weight of every free vertex by at least 1 completing the proof for (I2).

Edges that are deleted during a phase do not participate in any admissible augmenting path.
for the rest of the phase. We show this in two steps. First, we show that at the time of
deletion of an edge \((u, v)\), there is no path in the admissible graph that starts from the edge
\((u, v)\) and ends at a free vertex \(a \in A_F\) (Lemma 4.9). In Lemma 4.7, we show that any such
dge \((u, v)\) will not participate in any admissible alternating path to a free vertex of \(A_F\) for
the rest of the phase.

We use DFS\((b, k)\) to denote the DFS initiated from \(b\) in phase \(k\). Let \(P^b_u\) denote the path
maintained by DFS\((b, k)\) when the vertex \(u\) was added to the path.

**Lemma 4.7.** Consider some point during the second stage of phase \(k\) where there is an edge
\((u, v)\) that does not participate in any admissible alternating path to a vertex of \(A_F\). Then,
for the remainder of phase \(k\), \((u, v)\) does not participate in any admissible alternating path
to a vertex of \(A_F\).

**Proof.** Assume for the sake of contradiction that at some later time during phase \(k\), \((u, v)\)
becomes part of an admissible path \(P_{y,z}\) from a vertex \(y\) to a vertex \(z \in A_F\). Consider the
first time this occurs for \((u, v)\). During the second stage, the dual weights of some vertices of
\(B\) may decrease just prior to augmentation; however, this does not create any new admissible
edges. Therefore, \(P_{y,z}\) must have become an admissible path due to augmentation along a
path \(P_{a,b}\) from some \(b \in B_F\) to some \(a \in A_F\). Specifically, \(P_{y,z}\) must intersect \(P_{a,b}\) at some
vertex \(x\). Therefore, prior to augmenting along \(P_{a,b}\), there was an admissible path from \(y\) to
\(a\) via \(x\). This contradicts the assumption that \((u, v)\) did not participate in any admissible
path to a vertex of \(A_F\) prior to this time.

**Lemma 4.8.** Consider the execution of DFS\((b, k)\) and the path \(P^b_u\). Suppose the DFS\((b, k)\)
marks an edge \((u, v)\) as visited. Let \(P_v\) be an admissible alternating path from \(v\) to any free
vertex \(a \in A_F\) in \(G'\). Suppose \(P_v\) and \(P^b_u\) are vertex-disjoint. Then, DFS\((b, k)\) will find an
augmenting path that includes the edge \((u, v)\).
Chapter 4. A Weighted Approach to Maximum Cardinality Bipartite Matching

Proof. $P_v$ and $P^b_u$ are vertex-disjoint and so, $v$ is not on the path $P^b_u$. Therefore, DFS($b,k$) will add $(u,v)$ to the path and we get the path $P = P^b_v$. We will show that all edges of $P_v$ are unvisited by DFS($b,k$), and so the DFS procedure, when continued from $v$, will discover an augmenting path.

We show, through a contradiction, that all edges of $P_v$ are not yet visited by DFS($b,k$). Consider, for the sake of contradiction, among all the edges of $P_v$, the edge $(u', v')$ that was marked visited first. We claim the following:

(i) $(u', v')$ is visited before $(u, v)$: This follows from the assumption that when $(u, v)$ was marked as visited, $(u', v')$ was already marked as visited by the DFS.

(ii) $(u, v)$ is not a descendant of $(u', v')$ in the DFS: If $(u', v')$ was an ancestor of $(u, v)$ in the DFS, then $P^b_u$ contains $(u', v')$. By definition, $P_v$ also contains $(u', v')$, which contradicts the assumption that $P^b_u$ and $P_v$ are disjoint paths.

(iii) When $(u', v')$ is marked visited, it will be added to the path by the DFS: The only reason why $(u', v')$ is visited but not added is if $v'$ is already on the path $P^b_u$. In that case, $P_v$ and $P^b_u$ will share an edge that was visited before $(u', v')$ contradicting the assumption that $(u', v')$ was the earliest edge of $P_v$ to be marked visited.

From (iii), when $(u', v')$ was visited, it was added to the path $P^b_v$. Since $(u', v')$ was the edge on $P_v$ that was marked visited first by DFS($b,k$), all edges on the subpath from $v'$ to $a$ are unvisited. Therefore, the DFS($b,k$), when continued from $v'$, will not visit $(u, v)$ (from (ii)), will find an augmenting path, and terminate. From (i), $(u, v)$ will not be marked visited by DFS($b,k$) leading to a contradiction.

Lemma 4.9. Consider a DFS initiated from some free vertex $b \in B_F$ in phase $k$. Let $M$ be the matching at the start of this DFS and $M'$ be the matching when the DFS terminates.
4.3. Proof of Invariants

Suppose the edge \((u, v)\) was deleted during DFS\((b, k)\). Then there is no admissible path starting with \((u, v)\) and ending at a free vertex \(a \in A_F\) in \(G_{M'}\).

Proof. At the start of phase \(k\), \(G'\) is initialized to the admissible graph. Inductively, we assume that all the edges discarded in phase \(k\) prior to the execution of DFS\((b, k)\) do not participate in any augmenting path of admissible edges with respect to \(M\). Therefore, any augmenting path of admissible edges in \(G_M\) remains an augmenting path in \(G'\). There are two possible outcomes for DFS\((b, k)\). Either, (i) the DFS terminates without finding an augmenting path, or (ii) the DFS terminates with an augmenting path \(\tilde{P}\) and \(M' = M \oplus \tilde{P}\).

In case (i), \(M = M'\) and any edge \((u, v)\) visited by the DFS\((b, k)\) is marked for deletion. For the sake of contradiction, let \((u, v)\) participate in an admissible path \(P\) to a free vertex \(a' \in A_F\). Since \(u\) is reachable from \(b\) and \(a'\) is reachable from \(u\) in \(G_M\), \(a'\) is reachable from \(b\). This contradicts the fact that DFS\((b, k)\) did not find an augmenting path. Therefore, no edge \((u, v)\) marked for deletion participates in an augmenting path with respect to \(M\).

In case (ii), \(M' = M \oplus \tilde{P}\). DFS\((b, k)\) marks two kinds of edges for deletion.

(a) Any edge \((u, v)\) on the augmenting path \(\tilde{P}\) such that \(c(u, v) = 1\) is deleted, and,

(b) Any edge \((u, v)\) that is marked visited by DFS\((b, k)\), does not lie on \(\tilde{P}\), and does not belong to any affected piece is deleted.

In (a), there are two possibilities (1) \((u, v) \in \tilde{P} \cap M\) or (2) \((u, v) \in \tilde{P} \setminus M\). If \((u, v) \in M\) (case (a)(1)), then, after augmentation along \(\tilde{P}\), \(s(u, v)\) increases from 0 to at least 2, and \((u, v)\) is no longer admissible. Therefore, \((u, v)\) does not participate in any admissible alternating paths to a free vertex in \(A_F\) with respect to \(G_{M'}\). If \((u, v) \notin M\) (case (a)(2)), then the AUGMENT procedure reduces the dual weight of \(u \in B\) by 2. So, every edge going out of \(u\)
Chapter 4. A Weighted Approach to Maximum Cardinality Bipartite Matching

will have a slack of at least 2. Therefore, \((u, v)\) cannot participate in any admissible path \(P\) to a free vertex in \(A_F\). This completes case (a).

For (b), we will show that \((u, v)\), even prior to augmentation along \(\tilde{P}\), did not participate in any path of admissible edges from \(v\) to any free vertex of \(A_F\). For the sake of contradiction, let there be a path \(P_v\) from \(v\) to \(a' \in A_F\). We claim that \(P_v\) and \(P_{u'}^b\) are not vertex-disjoint. Otherwise, from Lemma 4.8, the path \(\tilde{P}\) found by DFS\((b, k)\) includes \((u, v)\). However, by our assumption for case (b), \((u, v)\) does not lie on \(\tilde{P}\). Therefore, we safely assume that \(P_v\) intersects \(P_{u'}^b\). There are two cases:

- **\(c(u, v) = 1\):** We will construct a cycle of admissible edges containing the edge \((u, v)\).
  
  Since \(c(u, v) = 1\), our construction will contradict Lemma 4.2. Let \(x\) be the first vertex common to both \(P_v\) and \(P_{u'}^b\) as we walk from \(v\) to \(a' \in A_F\). To create the cycle, we traverse from \(x\) to \(u\) along the path \(P_{u'}^b\), followed by the edge \((u, v)\), followed by the path from \(v\) to \(x\) along \(P_v\). All edges of this cycle are admissible including the edge \((u, v)\).

- **\(c(u, v) = 0\):** In this case, \((u, v)\) belongs to some piece \(K_i\) that is not an affected piece. Among all edges visited by DFS\((b, k)\), consider the edge \((u', v')\) of \(K_i\), the same piece as \((u, v)\), such that \(v'\) has a path to the vertex \(a' \in A_F\) with the fewest number of edges. Let \(P_{v'}\) be this path. We claim that \(P_{v'}\) and \(P_{u'}^b\) are not vertex-disjoint. Otherwise, from Lemma 4.8, the path \(\tilde{P}\) found by DFS\((b, k)\) includes \((u', v')\) and \(K_i\) would have been an affected piece. Therefore, we can safely assume that \(P_{v'}\) intersects with \(P_{u'}^b\).
  
  Let \(z\) be the first intersection point with \(P_{u'}^b\) as we walk from \(v'\) to \(a'\) and let \(z'\) be the vertex that follows after \(z\) in \(P_{u'}^b\). There are two possibilities:
  
  - The edge \((z, z') \in K_i\): In this case, \((z, z')\) is also marked visited by DFS\((b, k)\), and \(z'\) has path to \(a'\) with fewer number of edges than \(v'\). This contradicts our
4.4. Weights for Graphs with Good Separators

assumption about \((u', v')\).

- The edge \((z, z') \notin \mathcal{K}_i\): In this case, consider the cycle obtained by walking from 
z to \(u'\) along the path \(P_{u'}\) followed by the edge \((u', v')\) and the path from \(v'\) to 
z along \(P_{v'}\). Since \((u', v') \in \mathcal{K}_i\) and \((z, z') \notin \mathcal{K}_i\), the admissible cycle contains at least one edge of weight 1. This contradicts Lemma 4.2.

This concludes case (b) which shows that \((u, v)\) did not participate in any augmenting paths with respect to \(M\). From Lemma 4.7, it follows that \((u, v)\) does not participate in any augmenting path with respect to \(G_{M'}\) as well. 

\[ \square \]

**Corollary 4.10.** At the end of any phase, there is no augmenting path of admissible edges.

### 4.4 Weights for Graphs with Good Separators

A **balanced vertex separator** for any graph \(G(A \cup B, E)\) is a set \(S \subseteq V\) of vertices whose removal from \(G\) disconnects the graph into two pieces, each piece having at least a third of the vertices in \(V\). For any \(\delta \in [1/2, 1)\), we say that a graph has a balanced vertex separator of size \(n^\delta\) if every subgraph \(G'(V', E')\) of \(G\) has a balanced vertex separator \(S'\) such that \(|S'| = O(|V'|^{\delta})\).

Consider any graph \(G(V, E)\) with a balanced vertex separator of size \(n^\delta\). We will assign a weight of 0 and 1 to its edges so that the cost of any matching is no more than \(w = O(n^{\frac{2\delta}{1+\delta}})\) and the value of \(r = O(n^{\frac{1}{1+\delta}})\). Assuming such a weight assignment, our algorithm will compute a maximum cardinality matching in \(\tilde{O}(mn^{\frac{1}{1+\delta}})\) leading to Theorem 4.11.

We partition \(G\) into pieces to obtain the weight assignment for its edges. First, we compute a vertex separator \(S\) of \(G_0 = G\). We recursively partition each of the two pieces obtained by the removal of vertices in \(S\). This recursion terminates when the piece has fewer than \(r = O(n^{\frac{1}{1+\delta}})\) vertices. Note that no piece can have fewer than \(r/3\) vertices. It is easy to
see that the number of pieces is between $n/r$ and $3n/r$ and the total number of vertices designated as separator is $cn/r^{1-\delta}$ for some constant $c > 0$. The average number of edges per piece is at most $mr/n$. We set the weights of every edge incident on a separator vertex to 1 and the edges of every piece with fewer than $4mr/n$ edges to 0. The total number of pieces with at least $4mr/n$ edges is at most $n/4r$. Let $G_1$ be the graph formed by taking the union of all pieces with at least $4mr/n$ edges, and let the total number of vertices in $G_1$ be $n/c_1$ where $c_1 \geq 4$. Similar to $G_0$, the weight assignment for edges in $G_1$ is done in a recursive fashion where the recursion terminates when each piece has at most $r/c_1$ vertices. Again, the total number of pieces is between $n/r$ and $3n/r$ and the average number of edges per piece is at most $mr/n$. The total number of vertices that are designated as a separator vertex is at most $cn/r^{1-\delta}$. Since $c_1 \geq 4$, the total number of separator vertices is at most $c n/r^{1-\delta}$. We take all pieces of $G_1$ with fewer than $4mr/n$ edges and set the weight of its edges to 0. An edge incident on any separator vertex is assigned a weight of 1. The rest of the pieces combined together form $G_2$. We continue this until every piece has fewer than $4mr/n$ edges. The total number of vertices that are designated as a separator vertex will be $c n/r^{1-\delta}$ for any $G_i$ and $O(n/r^{1-\delta})$ across all graphs. All the edges with weight 1 are incident only on the separator vertices and therefore $w = O(n/r^{1-\delta})$. We choose $r = O(n^{1/3})$ and so $w = O(n^{15/13})$ as desired.

Theorem 4.11. For any $\delta \in [1/2, 1)$ and any bipartite graph $G(A \cup B, E)$ with a balanced vertex separator of size $O(n^\delta)$, a maximum cardinality matching in $G$ can be computed in $O(mn^{25/173})$ time.
4.5 Minimum Bottleneck Matching

We are given two sets $A$ and $B$ of $n$ $d$-dimensional points. Consider a weighted and complete bipartite graph on points of $A$ and $B$. The weight of any edge $(a, b) \in A \times B$ is given by its Euclidean distance and denoted by $\|a - b\|$. For any matching $M$ of $A$ and $B$ let its largest weight edge be its bottleneck edge. In the minimum bottleneck matching problem, we wish to compute a matching $M_{\text{Opt}}$ of $A$ and $B$ with the smallest weight bottleneck edge. We refer to this weight as the bottleneck distance of $A$ and $B$ and denote it by $\beta^*$. An $(1 + \varepsilon)$-approximate bottleneck matching of $A$ and $B$ is any matching $M$ with a bottleneck edge weight of at most $(1 + \varepsilon)\beta^*$. We present an algorithm that takes as input $A, B$, and a value $\delta$ such that $\beta^* \leq \delta \leq (1 + \varepsilon/3)\beta^*$, and produces an $(1 + \varepsilon)$-approximate bottleneck matching. For simplicity in presentation, we describe our algorithm for the 2-dimensional case when all points of $A$ and $B$ are in a bounding square $S$. The algorithm easily extends to any arbitrary fixed dimension $d$. For 2-dimensional case, given a value $\delta$, our algorithm executes in $\tilde{O}(n^{4/3}/\varepsilon^3)$ time.

Although, the value of $\delta$ is not known to the algorithm, we can first find a value $\alpha$ that is guaranteed to be an $n$-approximation of the bottleneck distance [1, Lemma 2.2] and then select $O(\log n/\varepsilon)$ values from the interval $[\alpha/n, \alpha]$ of the form $(1 + \varepsilon/3)^i \alpha/n$, for $0 \leq i \leq O(\log n/\varepsilon)$. We will then execute our algorithm for each of these $O(\log n/\varepsilon)$ selected values of $\delta$. Our algorithm returns a maximum matching whose edges are of length at most $(1 + \varepsilon/3)\delta$ in $O(n^{4/3}/\varepsilon^3)$ time. At least one of the $\delta$ values chosen will be a $\beta^* \leq \delta \leq (1 + \varepsilon/3)\beta^*$. The matching returned by the algorithm for this value of $\delta$ will be perfect ($|M| = n$) and have a bottleneck edge of weight at most $(1 + \varepsilon/3)^2 \beta^* \leq (1 + \varepsilon)\beta^*$ as desired. Among all executions of our algorithm that return a perfect matching, we return a perfect matching with the smallest bottleneck edge weight. Therefore, the total time taken to compute the $(1 + \varepsilon)$-approximate bottleneck matching is $\tilde{O}(n^{4/3}/\varepsilon^4)$. 
Chapter 4. A Weighted Approach to Maximum Cardinality Bipartite Matching

Given the value of $\delta$, the algorithm will construct a graph as follows: Let $G$ be a grid on the bounding square $S$. The side-length of every square in this grid is $\varepsilon \delta/(6\sqrt{2})$. For any cell $\xi$ in the grid $G$, let $N(\xi)$ denote the subset of all cells $\xi'$ of $G$ such that the minimum distance between $\xi$ and $\xi'$ is at most $\delta$. By the use of a simple packing argument, it can be shown that $|N(\xi)| = O(1/\varepsilon^2)$.

For any point $v \in A \cup B$, let $\xi_v$ be the cell of grid $G$ that contains $v$. We say that a cell $\xi$ is active if $(A \cup B) \cap \xi \neq \emptyset$. Let $A_\xi$ and $B_\xi$ denote the points of $A$ and $B$ in the cell $\xi$. We construct a bipartite graph $G(A \cup B, E)$ on the points in $A \cup B$ as follows: For any pair of points $(a, b) \in A \times B$, we add an edge in the graph if $\xi_b \in N(\xi_a)$. Note that every edge $(a, b)$ with $\|a - b\| \leq \delta$ will be included in $G$. Since $\delta$ is at least the bottleneck distance, $G$ will have a perfect matching. The maximum distance between any cell $\xi$ and a cell in $N(\xi)$ is $(1 + \varepsilon/3)\delta$. Therefore, no edge in $G$ will have a length greater than $(1 + \varepsilon/3)\delta$. This implies that any perfect matching in $G$ will also be an $(1 + \varepsilon)$-approximate bottleneck matching.

We use our algorithm for maximum matching to compute this perfect matching in $G$. Note, that $G$ can have $\Omega(n^2)$ edges. For the sake of efficiency, our algorithm executes on a compact representation of $G$ that is described later. Next, we assign weights of 0 and 1 to the edges of $G$ so that the any maximum matching in $G$ has a small weight $w$.

For a parameter $r > 1$, we will carefully select another grid $G'$ on the bounding square $S$, each cell of which has a side-length of $\sqrt{r}(\varepsilon \delta/(6\sqrt{2}))$ and encloses $\sqrt{r} \times \sqrt{r}$ cells of $G$. For any cell $\xi$ of the grid $G$, let $\Box_\xi$ be the cell in $G'$ that contains $\xi$. Any cell $\xi$ of $G$ is a boundary cell with respect to $G'$ if there is a cell $\xi' \in N(\xi)$ such that $\Box_{\xi'} \neq \Box_\xi$. Equivalently, if the minimum distance from $\xi$ to $\Box_\xi$ is at most $\delta$, then $\xi$ is a boundary cell. For any boundary cell $\xi$ of $G$ with respect to grid $G'$, we refer to all points of $A_\xi$ and $B_\xi$ that lie in $\xi$ as boundary points. All other points of $A$ and $B$ are referred to as internal points. We

---

2Assume $r$ to be a perfect square.
carefully construct this grid $G'$ such that the total number of boundary points is $O(n/\varepsilon \sqrt{r})$ as follows: First, we will generate the vertical lines for $G'$, and then we will generate the horizontal lines using a similar construction. Consider the vertical line $y_{ij}$ to be the line $x = i(\varepsilon \delta)/(6\sqrt{2}) + j\sqrt{r}(\varepsilon \delta/(6\sqrt{2}))$. For any fixed integer $i$ in $[1, \sqrt{r}]$, consider the set of vertical lines $Y_i = \{y_{ij} \mid y_{ij} \text{ intersects the bounding square } S\}$. We label all cells $\xi$ of $G$ as boundary cells with respect to $Y_i$ if the distance from $\xi$ to some vertical line in $Y_i$ is at most $\delta$. We designate the points inside the boundary cells as boundary vertices with respect to $Y_i$. For any given $i$, let $A_i$ and $B_i$ be the boundary vertices of $A$ and $B$ with respect to the lines in $Y_i$. We select an integer $\kappa = \arg \min_{1 \leq i \leq \sqrt{r}} |A_i \cup B_i|$ and use $Y_\kappa$ as the vertical lines for our grid $G'$. We use a symmetric construction for the horizontal lines.

**Lemma 4.12.** Let $A_i$ and $B_i$ be the boundary points with respect to the vertical lines $Y_i$. Let $\kappa = \arg \min_{1 \leq i \leq \sqrt{r}} |A_i \cup B_i|$. Then, $|A_\kappa \cup B_\kappa| = O(n/(\varepsilon \sqrt{r}))$.

**Proof.** For any fixed cell $\xi$ in $G$, of the $\sqrt{r}$ values of $i$, there are $O(1/\varepsilon)$ values for which $Y_i$ has a vertical line at a distance at most $\delta$ from $\xi$. Therefore, each cell $\xi$ will be a boundary cell in only $O(1/\varepsilon)$ shifts out of $\sqrt{r}$ shifts. So, $A_\xi$ and $B_\xi$ will be counted in $A_i \cup B_i$ for $O(1/\varepsilon)$ different values of $i$. Therefore, if we take the average over choices of $i$, we get

$$\min_{1 \leq i \leq \sqrt{r}} |A_i \cup B_i| \leq \frac{1}{\sqrt{r}} \sum_{i=1}^{\sqrt{r}} |A_i \cup B_i| \leq O(n/(\varepsilon \sqrt{r})).$$

Using a similar construction, we guarantee that the boundary points with respect to the horizontal lines of $G'$ is also at most $O(n/(\varepsilon \sqrt{r}))$.

**Corollary 4.13.** The grid $G'$ that we construct has $O(n/(\varepsilon \sqrt{r}))$ many boundary points.
For any two cells \( \xi \) and \( \xi' \in N(\xi) \) of the grid \( \mathcal{G} \), suppose \( \Box_\xi \neq \Box_{\xi'} \). Then the weights of all edges of \( A_\xi \times B_{\xi'} \) and of \( B_\xi \times A_{\xi'} \) are set to 1. All other edges have a weight of 0. We do not make an explicit weight assignment as it is expensive to do so. Instead, we can always derive the weight of an edge when we access it. Only boundary points will have edges of weight 1 incident on them. From Corollary 4.13, it follows that any maximum matching will have a weight of \( w = O(n/\varepsilon \sqrt{r}) \).

The edges of every piece in \( \mathcal{G} \) have endpoints that are completely inside a cell of \( \mathcal{G}' \). Note, however, that there is no straight-forward bound on the number of points and edges of \( \mathcal{G} \) inside each piece. Moreover, the number of edges in \( \mathcal{G} \) can be \( \Theta(n^2) \). Consider any feasible matching \( M, y(\cdot) \) in \( \mathcal{G} \). Let \( \mathcal{G}_M \) be the residual network. In order to obtain a running time of \( \tilde{O}(n^{4/3}/\varepsilon^3) \), we use the grid \( \mathcal{G} \) to construct a compact residual network \( \mathcal{C}\mathcal{G}_M \) for any feasible matching \( M, y(\cdot) \) and use this compact graph to implement our algorithm. The following lemma assists us in constructing the compressed residual network.

**Lemma 4.14.** Consider any feasible matching \( M, y(\cdot) \) maintained by our algorithm on \( \mathcal{G} \) and any active cell \( \xi \) in the grid \( \mathcal{G} \). The dual weight of any two points \( a, a' \in A_\xi \) can differ by at most 2. Similarly, the dual weights of any two points \( b, b' \in B_\xi \) can differ by at most 2.

**Proof.** We present our proof for two points \( b, b' \in B_\xi \). A similar argument will extend for \( a, a' \in A_\xi \). For the sake of contradiction, let \( y(b) \geq y(b') + 3 \). \( b' \) must be matched since \( y(b') < y(b) \leq y_{\text{max}} \). Let \( m(b') \in A \) be the match of \( b' \) in \( M \). From (4.2), \( y(m(b')) - y(b') = c(b', m(b')) \). Since both \( b \) and \( b' \) are in \( \xi \), the distance \( c(b, m(b')) = c(b', m(b')) \). So, \( y(b) - y(m(b')) \geq (y(b') + 3) - y(m(b')) = 3 - c(b, m(b')) \). This violates (4.1) leading to a contradiction. \( \square \)

For any feasible matching and any cell \( \xi \) of \( \mathcal{G} \), we divide points of \( A_\xi \) and \( B_\xi \) based on their dual weight into at most three clusters. Let \( A^1_\xi, A^2_\xi \) and \( A^3_\xi \) be the three clusters of points
in $A_\xi$ and let $B^1_\xi, B^2_\xi$, and $B^3_\xi$ be the three clusters of points in $B_\xi$. We assume that points with the largest dual weights are in $A^1_\xi$ (resp. $B^1_\xi$), the points with the second largest dual weights are in $A^2_\xi$ (resp. $B^2_\xi$), and the points with the smallest dual weights are in $A^3_\xi$ (resp. $B^3_\xi$).

**Compact Residual Network** Given a feasible matching $M$, we construct a compact residual network $CG_M$ to assist in the fast implementation of our algorithm. This vertex set $A \cup B$ for the compact residual network is constructed as follows. First we describe the vertex set $A$. For every active cell $\xi$ in $G$, we add a vertex $a^1_\xi$ (resp. $a^2_\xi, a^3_\xi$) to represent the set $A^1_\xi$ (resp. $A^2_\xi, A^3_\xi$) provided $A^1_\xi \neq \emptyset$ (resp. $A^2_\xi \neq \emptyset, A^3_\xi \neq \emptyset$). We designate $a^1_\xi$ (resp. $a^2_\xi, a^3_\xi$) as a free vertex if $A^1_\xi \cap M = \emptyset$ (resp. $A^2_\xi \cap M = \emptyset, A^3_\xi \cap M = \emptyset$). Similarly, we construct a vertex set $B$ by adding a vertex $b^1_\xi$ (resp. $b^2_\xi, b^3_\xi$) to represent the set $B^1_\xi$ (resp. $B^2_\xi, B^3_\xi$) provided $B^1_\xi \neq \emptyset$ (resp. $B^2_\xi \neq \emptyset, B^3_\xi \neq \emptyset$). We designate $b^1_\xi$ (resp. $b^2_\xi, b^3_\xi$) as a free vertex if $B^1_\xi \cap M = \emptyset$ (resp. $B^2_\xi \cap M = \emptyset, B^3_\xi \cap M = \emptyset$). Each active cell $\xi$ of the grid $G$ therefore has at most six points. Each point in $A \cup B$ will inherit the dual weights of the points in its cluster; for any vertex $a^i_\xi \in A$ (resp. $a^2_\xi \in A, a^3_\xi \in A$), let $y(a^i_\xi)$ (resp. $y(a^2_\xi), y(a^3_\xi)$) be the dual weight of all points in $A^1_\xi$ (resp. $A^2_\xi, A^3_\xi$). We define $y(b^i_\xi)$, $y(b^2_\xi)$, and $y(b^3_\xi)$ as dual weights of points in $B^1_\xi, B^2_\xi$, and $B^3_\xi$ respectively. Since there are at most $n$ active cells, $|A \cup B| = O(n)$.

Next, we create the edge set for the compact residual network $CG_M$. For any active cell $\xi$ in the grid $G$ and for any cell $\xi' \in N(\xi)$,

- We add a directed edge from $a^i_\xi$ to $b^j_{\xi'}$, for $i, j \in \{1, 2, 3\}$ if there is an edge $(a, b) \in (A^i_\xi \times B^j_{\xi'}) \cap M$. We define the weight of $(a^i_\xi, b^j_{\xi'})$ to be $c(a, b)$. We also define the slack $s(a^i_\xi, b^j_{\xi'})$ to be $c(a^i_\xi, b^j_{\xi'}) - y(a^i_\xi) + y(b^j_{\xi'})$ which is equal to $s(a^i_\xi, b^j_{\xi'}) = c(a, b) - y(a) + y(b) = s(a, b) = 0$.

- We add a directed edge from $b^i_\xi$ to $a^j_{\xi'}$, for $i, j \in \{1, 2, 3\}$ if $(B^i_\xi \times A^j_{\xi'}) \setminus M \neq \emptyset$. Note
that the weight and slack of every directed edge in $B^i_\xi \times A^j_{\xi'}$ are identical. We define the weight of $(b^i_\xi, a^j_{\xi'})$ to be $c(a, b)$ for any $(a, b) \in A^j_{\xi'} \times B^i_\xi$. We also define the slack $s(b^i_\xi, a^j_{\xi'}) = c(b^i_\xi, a^j_{\xi'}) - y(b^i_\xi) + y(a^j_{\xi'})$ which is equal to the slack $s(a, b)$.

For each vertex in $A \cup B$, we added at most two edges to every cell $\xi' \in N(\xi)$. Since $N(\xi) = O(1/\varepsilon^2)$, the total number of edges in $E$ is $O(n/\varepsilon^2)$. For a cell $\square$ in $G'$, let $A_{\square}$ be the points of $A$ generated by cells of $G$ that are contained inside the cell $\square$. A piece $K_{\square}$ has $A_{\square} \cup B_{\square}$ as the vertex set and $E_{\square} = ((A_{\square} \times B_{\square}) \cup (B_{\square} \times A_{\square}) \cap E)$ as the edge set. Note that the number of vertices in any piece $K_{\square}$ is $O(r)$ and the number of edges in $K_{\square}$ is $O(r/\varepsilon^2)$. Every edge $(u, v)$ of any piece $K_{\square}$ has a weight $c(u, v) = 0$ and every edge $(u, v)$ with a weight of zero belongs to some piece of $CG_M$.

The following straight-forward lemma implies that the compact graph $CG_M$ preserves all minimum slack paths in $G_M$.

**Lemma 4.15.** For any directed path $P$ in the compact residual network $CG_M$, there is a directed path $P$ in the residual network such that $\sum_{(u,v) \in P} s(u, v) = \sum_{(u,v) \in P} s(u, v)$. For any directed path $P$ in $G_M$, there is a directed path $P$ in the compact residual network such that $\sum_{(u,v) \in P} s(u, v) \geq \sum_{(u,v) \in P} s(u, v)$.

**Preprocessing Step**  At the start, $M = \emptyset$ and all dual weights are 0. Consider any cell $\square$ of the grid $G'$ and any cell $\xi$ of $G$ that is contained inside $\square$. Suppose we have a point $a^1_{\xi}$. We assign a demand $d_{a^1_{\xi}} = |A^1_{\xi}| = |A_{\xi}|$ to $a^1_{\xi}$. Similarly, suppose we have a point $b^1_{\xi}$, we assign a supply $s_{b^1_{\xi}} = |B^1_{\xi}| = |B_{\xi}|$. The preprocessing step reduces to finding a maximum matching of supplies to demand. This is an instance of the unweighted transportation problem which can be solved using the algorithm of [35] in $\tilde{O}(|E_{\square}| \sqrt{|A_{\square} \cup B_{\square}|}) = \tilde{O}(|E_{\square}| \sqrt{r})$. Every edge of $E$ participates in at most one piece. Therefore, the total time taken for preprocessing across all pieces is $\tilde{O}(|E| \sqrt{r}) = \tilde{O}(n \sqrt{r}/\varepsilon^2)$. We can trivially convert the matching of supplies to
4.5. Minimum Bottleneck Matching

Efficient Implementation of the Second Step  Recollect that the second step of the algorithm consists of phases. Each phase has two stages. In the first stage, we execute Dijkstra’s algorithm in \( O(n \log n / \varepsilon^2) \) time by using the compact residual network \( CG \). After adjusting the dual weight of nodes in the compact graph, in the second stage, we iteratively compute augmenting paths of admissible edges by conducting a DFS from each vertex. Our implementation of DFS has the following differences from the one described in Section 4.2.

- Recollect that each free vertex \( v \in B \) may represent a cluster that has \( t > 0 \) free vertices. We will execute DFS from \( v \) exactly \( t \) times, once for each free vertex of \( B \).

- During the execution of any DFS, unlike the algorithm described in Section 4.2, the DFS will mark an edge as visited only when it backtracks from the edge. Due to this change, all edges on the path maintained by the DFS are marked as unvisited. Therefore, unlike the algorithm from Section 4.2, this algorithm will not discard weight 1 edges of an augmenting path after augmentation. From Lemma 4.3, the total number of these edges is \( O(w \log w) \).

Efficiency  The first stage is an execution of Dijkstra’s algorithm which takes \( O(|E| + |V| \log |V|) = O(n \log n / \varepsilon^2) \) time. Suppose there are \( \lambda \) phases; then the cumulative time taken across all phases for the first stage is \( \tilde{O}(\lambda n / \varepsilon^2) \). In the second stage of the algorithm, in each phase, every edge is discarded once it is visited by a DFS, unless it is in an affected piece or it is an edge of weight 1 on an augmenting path. Since each affected piece has \( O(r / \varepsilon^2) \) edges, and since there are \( O(w \log w) \) edges of weight 1 on the computed augmenting paths, the total time taken by all the DFS searches across all the \( \lambda \) phases is bounded by \( \tilde{O}(n \lambda / \varepsilon^2 + r / \varepsilon^2 \sum_{i=1}^{t} |K_i| + w \log w) \). In Lemma 4.3, we bound \( \lambda \) by \( \sqrt{w} \) and \( \sum_{i=1}^{t} |K_i| \) by \( O(w \log w) \).
Therefore, the total time taken by the algorithm including the time taken by preprocessing step is $\tilde{O}(\frac{n}{\varepsilon^2}(\sqrt{r} + \sqrt{w} + \frac{wr}{n}))$. Setting $r = n^{2/3}$, we get $w = O(n/(\varepsilon \sqrt{r})) = O(n^{2/3}/\varepsilon)$, and the total running time of our algorithm is $\tilde{O}(n^{4/3}/\varepsilon^3)$. To obtain the bottleneck matching, we execute this algorithm on $O(\log(n/\varepsilon))$ guesses; therefore, the total time taken to compute an $(1 + \varepsilon)$-approximate bottleneck matching is $\tilde{O}(n^{4/3}/\varepsilon^4)$. For $d > 2$, we choose $r = n^{\frac{d}{2d-1}}$ and $w = O(n/(d\varepsilon r^{1/d}))$. With these values, the execution time of our algorithm is $\frac{1}{\varepsilon O(d)} n^{1+\frac{d-1}{2d-1}} \text{poly log } n$.

**Theorem 4.16.** Given $A, B \subset \mathbb{R}^d$ and a parameter $\varepsilon > 0$, we can compute a $(1 + \varepsilon)$-approximate bottleneck matching of $A$ and $B$ in $\frac{1}{\varepsilon O(d)} n^{1+\frac{d-1}{2d-1}} \text{poly log } n$ time.
Chapter 5

Minimum-Cost Bipartite Matching on Planar Graphs

Given a weighted planar bipartite graph $G(A \cup B, E)$ where each edge has a non-negative integer edge cost, we give an $\tilde{O}(n^{4/3} \log nC)$ time algorithm to compute minimum-cost perfect matching; here $C$ is the maximum edge cost in the graph. The previous best known planarity exploiting algorithm has a running time of $O(n^{3/2} \log n)$ and is achieved by using planar separators (Lipton and Tarjan ’80).

Our algorithm is based on the bit-scaling paradigm (Gabow and Tarjan ’89). For each scale, our algorithm first executes $O(n^{1/3})$ iterations of Gabow and Tarjan’s algorithm in $O(n^{4/3})$ time leaving only $O(n^{2/3})$ vertices unmatched. Next, it constructs a compressed residual graph $H$ with $O(n^{2/3})$ vertices and $O(n)$ edges. This is achieved by using an $r$-division of the planar graph $G$ with $r = n^{2/3}$. For each partition of the $r$-division, there is an edge between two vertices of $H$ if and only if they are connected by a directed path inside the partition.

Using existing efficient shortest-path data structures, the remaining $O(n^{2/3})$ vertices are matched by iteratively computing a minimum-cost augmenting path each taking $\tilde{O}(n^{2/3})$ time. Augmentation changes the residual graph, so the algorithm updates the compressed representation for each partition affected by the change in $\tilde{O}(n^{2/3})$ time. We bound the total number of affected partitions over all the augmenting paths by $O(n^{2/3} \log n)$. Therefore, the total time taken by the algorithm is $\tilde{O}(n^{4/3})$. 
Chapter 5. Minimum-Cost Bipartite Matching on Planar Graphs

Organization The chapter is organized as follows: In Section 5.1 we present relevant background information. In Section 5.2 we describe our scaling algorithm. In Section 5.3, we present our algorithm for each scale. In Section 5.4 we prove the correctness and efficiency of our algorithm.

5.1 Background

An $r$-Division of a Planar Graph We introduce the notion of an $r$-division of a planar graph which we use in our algorithm.

Definition 5.1. An $r$-division of any planar graph $G(V, E)$, given by $\mathcal{R}(G)$ is a partition of $G$ into $l = O(n/r)$ edge-disjoint subgraphs $\{R_1(V_1, E_1), \ldots, R_l(V_l, E_l)\}$, where $E_j = \{(a, b) | (a, b) \in E, a, b \in V_j\}$, $\bigcup_j E_j = E$ and $\bigcup_j V_j = V$. A vertex that has incident edges from two or more partitions is called a boundary vertex. All other vertices are called as internal vertices. For any partition $R_j$, let $\mathcal{K}_j$ denote its set of boundary vertices. Specifically, the $r$-division $\mathcal{R}(G)$ has the following properties:

- For every partition $R_j$, $|V_j| \leq r$ and $|\mathcal{K}_j| = O(\sqrt{r})$. There are $O(n/r)$ partitions. Let $k$ be a constant such that $\sum_{j=1}^l |\mathcal{K}_j| \leq kn/\sqrt{r}$.
- Every partition $R_j$ contains $O(1)$ faces that are not faces of a planar embedding of $G$ (also called holes).

As mentioned in [23], Klein and Subramanian [28] show that an $r$-division with these properties can be created in $O(n \log n)$ time. Given a value of $r$, we use the same $r$-division during the course of the algorithm. The construction of $r$-division and the subsequent use of data structure for planar graphs requires the assumption that the input graph has a constant
5.1. Background

Figure 5.1: Transforming any bipartite graph to a constant degree graph

maximum degree. Biedl [8] showed how any planar bipartite graph can be transformed into another planar bipartite graph such that the degree of every vertex is at most 3 and the matching is “preserved”. For the sake of completion, we present this transformation in the context of weighted bipartite graphs. First, we provide a transformation of \( G \) to graph \( \tilde{G} \) which reduces the maximum degree in \( \tilde{G} \) by 1 (provided the maximum degree is at least 4). Repeatedly applying this transformation, we obtain a graph where every vertex has a degree of at most 3. We describe this transformation next. Take any vertex \( v \in G \) with a degree at least 4 and consider two vertices \( x \) and \( y \) such that \( (v, x) \) and \( (v, y) \) appear next to each other in the clockwise ordering of all edges incident on \( v \). Suppose \( v \in B \) (a symmetric construction also works when \( v \in A \)). We add two new vertices \( a, b \) and add edges \( (v, a), (a, b), (b, x) \) and \( (b, y) \) with costs \( c(v, a) = 0, c(a, b) = 0, c(b, x) = c(v, x) \) and \( c(b, y) = c(v, y) \). See Figure 5.1. This transformation reduces the degree of \( v \) by 1 and preserves planarity. Using a straight-forward proof from [8] it can be shown that the optimal matching in \( G \) and \( \tilde{G} \) have the same cost and the optimal matching in \( G \) can be retrieved from the optimal matching in \( \tilde{G} \).

**Convention for Notation** Throughout this paper, we assume \( G \) is a planar bipartite graph with \( A \cup B \) as the vertex set and \( E \) as the set of edges. Given a matching \( M \), we refer to its residual graph by \( G_M \). Note that the vertex and edge sets of \( G \) and \( G_M \) are identical (except for the directions) and a matching, alternating path or an alternating cycle in \( G \) is also a matching, directed path or a directed cycle in \( G_M \). So, if there is any subset \( P \) of
edges in $G$, we will also use $P$ to denote the same subset of edges in $G_M$, the directions of these edges are determined by whether or not an edge is in $M$. We will define a net-cost for an alternating path (or cycle) $P$ in our algorithm and denote it by $\phi(P)$. Any directed path or cycle in $G_M$ will inherit its net-cost from $G$. During the course of our algorithm, for any weighted and directed graph $K$, we will use the notation $K'$ to be the graph identical to $K$ where the cost of any directed edge in the graph is replaced by its slack. Recall that an $r$-division $\mathcal{R}(G)$ partitions the edges of $G$. Since $G$, $G_M$ and $G'_M$ have the same underlying set of edges, $\mathcal{R}(G)$ can be seen as an $r$-division of $G_M$ and $G'_M$ as well. Therefore, we use $\mathcal{R}(G), \mathcal{R}(G_M)$ and $\mathcal{R}(G'_M)$ to denote the same $r$-division.

In the following, we present a scaling algorithm to compute minimum-cost matching in planar graphs. Our algorithm is similar in spirit to the Gabow-Tarjan algorithm. However, by using a compressed residual graph, we achieve a faster execution time of $O(n^{4/3} \log^2 n \log n C)$ in computing the optimal matching.

### 5.2 Our Scaling Algorithm

We next introduce a notion of feasibility that is based on an $r$-division of a planar graph. We assume that we are given an $r$-division, $\mathcal{R}(G) = \{\mathcal{R}_1(V_1, E_1), \ldots, \mathcal{R}_l(V_l, E_l)\}$ with ($l = O(n/r)$). Recall that we denote the set of boundary vertices of $\mathcal{R}_j$ by $\mathcal{K}_j$. For every vertex $v \in A \cup B$, we define a 0/1 indicator variable $i_v$ to be 1 if and only if $v$ is a boundary vertex with respect to $\mathcal{R}(G)$. For any edge $(u, v) \in E$, we define a value $\delta_{uv}$ to be $\max\{1, i_u \lceil \sqrt{r} \rceil, i_v \lceil \sqrt{r} \rceil\}$. For any edge induced sub-graph $G^\ast(V^\ast, E^\ast)$ of $G(A \cup B, E)$ and the $r$-division $\mathcal{R}(G)$, we say that a matching $M \subseteq E^\ast$ and a set of dual weights $y(\cdot)$ on the vertices of $V^\ast$ are $\mathcal{R}$-feasible
5.2. Our Scaling Algorithm

if the following conditions are satisfied.

\[ y(u) + y(v) \leq c(u, v) + \delta_{uv} \quad \text{for} \quad (u, v) \in E^* \quad (5.1) \]
\[ y(u) + y(v) = c(u, v) \quad \text{for} \quad (u, v) \in M \quad (5.2) \]

An \( \mathcal{R} \)-optimal matching is a perfect matching that is \( \mathcal{R} \)-feasible. In our algorithm, for the graph \( G(A \cup B, E) \) and the \( r \)-division \( \mathcal{R}(G) \), we would like to compute an \( \mathcal{R} \)-optimal matching \( M \) along with its dual weights \( y(\cdot) \). Note that \( \mathcal{R} \)-feasible matching is defined for any edge induced subgraph of \( G \). In this paper, the only induced subgraphs that we consider are partitions from the \( r \)-division \( \{ \mathcal{R}_1, \ldots, \mathcal{R}_l \} \). Our algorithm will maintain an \( \mathcal{R} \)-feasible matching for each partition. Throughout this paper, we will fix the \( \mathcal{R} \)-division in the definition of \( \mathcal{R} \)-feasibility to be \( \mathcal{R}(G) \). For any \( \mathcal{R} \)-feasible matching, when obvious from the context, we will not explicitly mention the induced sub-graph for which it is defined on.

The following lemma bounds the cost of an \( \mathcal{R} \)-optimal matching on \( G \).

**Lemma 5.2.** For a planar bipartite graph \( G(A \cup B, E) \) with a positive integer edge cost function \( c \), let \( M \) be an \( \mathcal{R} \)-optimal matching and \( M_{\text{OPT}} \) be some optimal matching. Then, \( c(M) \leq c(M_{\text{OPT}}) + (k + 1)n \) where \( k \) is a constant in definition (5.1).

**Proof.** Since \( M, y(\cdot) \) is \( \mathcal{R} \)-optimal, we have that

\[ c(M) = \sum_{u \in A \cup B} y(u). \quad (5.3) \]

From (5.1), every edge \( (u, v) \in M_{\text{OPT}} \) will satisfy \( y(u) + y(v) \leq c(u, v) + \delta_{uv} \). Since \( M_{\text{OPT}} \) is
a perfect matching, we have

\[
\sum_{u \in A \cup B} y(u) = \sum_{(u,v) \in M_{\text{Opt}}} (y(u) + y(v)) 
\leq c(M_{\text{Opt}}) + \sum_{(u,v) \in M_{\text{Opt}}} \delta_{uv}.
\]

There are at most \( kn/\sqrt{r} \) boundary vertices and so there are at most \( kn/\sqrt{r} \) edges \((u,v)\) in \( M_{\text{Opt}} \) such that \( \delta_{uv} = \lceil \sqrt{r} \rceil \). For the other \( n - \frac{kn}{\sqrt{r}} \) edges \((u,v)\) of \( M_{\text{Opt}} \), \( \delta_{uv} = 1 \).

Therefore,

\[
\sum_{(u,v) \in M_{\text{Opt}}} \delta_{uv} \leq \frac{kn}{\sqrt{r}}(\sqrt{r} + 1) + n - \frac{kn}{\sqrt{r}} = (k + 1)n.
\]

Combining (5.3) and (5.4) with (5.5), we get

\[ c(M) = \sum_{u \in A \cup B} y(u) \leq c(M_{\text{Opt}}) + (k + 1)n. \]

As in Gabow-Tarjan Algorithm, for every edge \((a,b) \in E\), we redefine its weight to be \( c^*(a,b) = (k + 1)(n + 1)c(a,b) \). Since this uniform scaling of edge costs preserves the set of optimal matchings, Lemma 5.2 implies that an \( R \)-optimal matching of the vertices of \( A, B \) with edge weights \( c^*(\cdot, \cdot) \) corresponds to an optimal matching with the original edge costs \( c(\cdot, \cdot) \).

Our algorithm also consists of scales. For any edge \((u,v)\), let \( b_1, b_2, \ldots, b_t \) be the binary representation of \( c^*(u,v) \). In the \( i \)th scale, the cost of an edge, \( c_i(u,v) \), corresponds to the most significant \( i \) bits of \( c^*(u,v) \). Each scale of our algorithm takes as input a planar bipartite graph on \( A, B \), with a cost function \( c_i(\cdot, \cdot) \), and a set of dual weights \( y(v) \) for every vertex \( v \in A \cup B \), and returns a \( R \)-optimal matching. We transfer the dual weights from scale \( i \)
5.3. Algorithm for Each Scale

Our algorithm takes a planar bipartite graph $G(A \cup B, E)$ with positive integer cost of $c(u, v)$ for every edge $(u, v)$ such that the optimal matching has a cost no more than $(k + 2)n$. It has two steps. The first step of the algorithm will simply execute $O(\sqrt{r})$ iterations of the Gabow-Tarjan’s algorithm. At the end of this step, our algorithm will have computed a 1-feasible matching $M$ with at most $O(n/\sqrt{r})$ free vertices. Additionally, from the properties of Gabow-Tarjan Algorithm, for every free vertex $a \in A$, $y(a) = 0$, and for every free vertex $b \in B$, $y(b) = \max_{b' \in B} y(b')$. Since every 1-feasible matching also satisfies (5.1) and (5.2), $M$ is also an $\mathcal{R}$-feasible matching. Therefore, at the end of the first step, we have the following.
Lemma 5.3. At the end of the first step of our algorithm, the matching \( M \) and the dual weights \( y(\cdot) \) form an \( \mathcal{R} \)-feasible matching for the graph \( G \), the number of free vertices with respect to \( M \) is at most \( O(n/\sqrt{r}) \), for every free vertex \( a \in A \), \( y(a) = 0 \), and for every free vertex \( b \in B \), \( y(b) = \max_{y(b')} y(b') \).

For our algorithm, we will define the net-cost of an augmenting path \( P \), denoted by \( \phi(P) \), by

\[
\phi(P) = \sum_{(u,v) \in P \setminus M} (c(u,v) + \delta_{uv}) - \sum_{(u,v) \in P \cap M} c(u,v).
\]

This definition also extends to alternating paths and alternating cycles in a straightforward way. We also define an edge \((u,v)\) as an admissible edge if \((u,v) \in M\) or if \((u,v) \notin M\) and,

\[
y(u) + y(v) = c(u,v) + \delta_{uv}.
\]

As in the Hungarian and Gabow-Tarjan algorithms, in the second step of our algorithm we will iteratively compute an augmenting path of admissible edges and augment the matching along this path. The augmenting path computed by our algorithm can also be seen as a minimum net-cost augmenting path.

We define the slack of an edge \((u,v) \in M\), \( s(u,v) = 0 \) and \((u,v) \in E \setminus M\) to be \( s(u,v) = c(u,v) + \delta_{uv} - y(u) - y(v) \). From the \( \mathcal{R} \)-feasibility conditions (5.1) and (5.2), it follows that the slack \( s(u,v) \geq 0 \).

We can easily compute the minimum net-cost augmenting path by conducting a Hungarian search. The search procedure will modify the dual weights of all the vertices in \( G \) and find an augmenting path of admissible edges. Unfortunately, this search may require us to visit and update the dual weights of all the \( n \) vertices of the graph requiring \( O(n) \) time, which is too slow for our purpose. To speed this up, we use a compressed representation \( H \).
5.3. Algorithm for Each Scale

of the residual graph $G_M$. In this representation, we only maintain the boundary vertices of the $r$-division and connect two boundary vertices $u$ and $v$ with a directed edge if and only if there is a directed path from $u$ to $v$ in some partition $R_j$. We assign a weight to $(u, v)$ that is equal to cost of the minimum net-cost path between $u$ and $v$ in $R_j$. Such a compressed representation $H$ of the residual graph has $O(n/\sqrt{r})$ vertices and $O(n)$ edges. These edges can be computed in $O(r \log r)$ time per partition with a total computation time of $O(n \log n)$. Also, any directed path $P$ in $H$ can be projected to an augmenting path in $G$ by simply replacing each edge of $P$ with the minimum net-cost directed path of $G_M$ that it represents.

In order to compute the minimum net-cost augmenting path, we show that it is sufficient to apply Hungarian search on $H$ which can be done in $O((n/\sqrt{r}) \log^2 r)$ time. However, as stated earlier, this search may affect the dual weights of all the $n$ vertices. We avoid updating all the dual weights each search by introducing the notion of a planar feasible matching. In a planar feasible matching, we will maintain a separate set of dual weights for each partition and another set of dual weights for all the vertices of $H$. Using the dual weights of $H$, we find the minimum net-cost augmenting path $P$ and augment the matching along its projection $\overrightarrow{P}$ on $G_M$. This augmentation changes the residual graph and therefore its compressed representation $H$ as well. We will then update the dual weights and the edges of every partition of the compressed residual graph that contains at least one edge of $\overrightarrow{P}$. Note that the dual weights of the vertices in other partitions may be affected as well. However, we will update them only when an augmenting path contains its edges. We show that the edges of a single partition of the compressed residual graph can be updated in $O(r \log r)$ time. The total time taken by the algorithm, therefore, is $O((n/\sqrt{r}) \log^2 r)$ to compute an augmenting path $P$ and $O(|P|r \log r)$ to update the dual weights and the edges of $H$. Therefore, the remaining $O(n/\sqrt{r})$ augmenting paths can be computed in
O((n^2/r) \log^2 r + r \log r \sum_i |P_i|) time where \( \sum_i |P_i| \) is the sum of the lengths of all these \( O(n/\sqrt{r}) \) augmenting paths. Interestingly, we show that the sum of the lengths of the augmenting paths is bounded by \( O((n/\sqrt{r}) \log n) \). This bounds the running time of our algorithm by \( O((n^2/r) \log^2 r + n\sqrt{r} \log r \log n) \) or \( O(n^{4/3} \log n) \) when \( r = n^{2/3} \).

In the following, we will define the compressed residual graph and formally introduce planar feasible matchings. After that, we will show that we can quickly convert any \( \mathcal{R} \)-feasible matching to a planar-feasible matching and vice versa. Finally, we will present the second step of our algorithm that computes an \( \mathcal{R} \)-optimal matching.

**Compressed Residual Graph** \( H \) We now describe the compressed residual graph \( H \), which will guide the execution of each iteration of the second step of our algorithm. For a matching \( M \), let \( G_M \) denote the (directed) residual graph with respect to \( M \). Let \( \mathcal{R}(G_M) = \mathcal{R}(G) \) be the \( r \)-division of \( G_M \) as given by Definition 5.1. Let \( A_F \) and \( B_F \) denote the set of free vertices (not matched by \( M \)) of \( A \) and \( B \) respectively. Our sparse graph \( H \) will be a **weighted multi-graph** whose vertex set is \( V_H \) and the edge set is \( E_H \) is defined next.

We define the vertex set \( V_j^H \) and the edge set \( E_j^H \) for a single partition \( \mathcal{R}_j \) of the \( r \)-division. The vertex set \( V_H \) and the edge set \( E_H \) is simply the union of all the edges and vertices over all partitions. For every partition \( \mathcal{R}_j \), \( V_j^H \) contains the boundary vertices \( K_j \). Also, if there is at least one internal vertex of \( A \) (resp. \( B \)) that is free (unmatched), i.e., \( (V_j \setminus K_j) \cap A_F \neq \emptyset \) (resp. \( (V_j \setminus K_j) \cap B_F \neq \emptyset \)), then we create a special vertex \( a_j \) (resp. \( b_j \)) to represent all vertices in this set in \( V_j^H \). We call these two additional vertices for \( \mathcal{R}_j \) the **free internal vertices** of \( \mathcal{R}_j \). We set \( V_j^H = K_j \cup \{a_j, b_j\} \), \( A_j^H = (K_j \cap A) \cup \{a_j\} \), and \( B_j^H = (K_j \cap B) \cup \{b_j\} \).

The free vertices of \( \mathcal{R}_j \) are represented by \( B_j^F = (B_F \cap K_j) \cup \{b_j\} \) and \( A_j^F = (A_F \cap K_j) \cup \{a_j\} \).

The vertex set \( V_H \) of \( H \) is thus given by \( V_H = \bigcup_{j=1}^l V_j^H \), \( B_H = \bigcup_{j=1}^l B_j^H \) and \( A_H = \bigcup_{j=1}^l A_j^H \).

The free vertices of \( H \) are given by \( A_H^F = \bigcup_{j=1}^l A_j^F \) and \( B_H^F = \bigcup_{j=1}^l B_j^F \).
Next, we define the set of edges $E_j^H$ for each partition $R_j$. For any $u, v \in V_j^H (u \neq v)$ there is an edge from $u$ to $v$ if

1) $u, v \in K_j$, i.e., $u$ and $v$ are boundary vertices and there is a directed path $\overrightarrow{P}$ from $u$ to $v$ in $G_M$ that only passes through the edges of $R_j$. Let $\overrightarrow{P}_{u,v,j}$ be the path consisting only of edges of $R_j$ that has the smallest net-cost. We denote this type of edge as a boundary-to-boundary edge.

2) $u = b_j$, $v \in K_j$, and there is a directed path $\overrightarrow{P}$ in $G_M$ from some free vertex in $B_F \cap (V_j \setminus K_j)$ to $v$ that only passes through the edges of $R_j$. Let $\overrightarrow{P}_{u,v,j}$ be the path consisting only of edges of $R_j$ that has the smallest net-cost.

3) $u \in K_j$, $v = a_j$, and there is a directed path $\overrightarrow{P}$ in $G_M$ from $u$ to some free vertex in $A_F \cap (V_j \setminus K_j)$ that only passes through the edges of $R_j$. Let $\overrightarrow{P}_{u,v,j}$ be the path consisting only of edges of $R_j$ that has the smallest net-cost.

4) $u = b_j$ and $v = a_j$ are free vertices and there is a directed path $\overrightarrow{P}$ in $G_M$ from some vertex in the set $B_F \cap (V_j \setminus K_j)$ to a vertex in the set $A_F \cap (V_j \setminus K_j)$ that only passes through the edges in $R_j$. Let $\overrightarrow{P}_{u,v,j}$ be the path consisting only of edges of $R_j$ that has the smallest net-cost.

We set the weight of $(u, v)$ to be $\phi(\overrightarrow{P}_{u,v,j})$. We also refer to this edge $(u, v)$ as an edge of partition $R_j$ and denote the set of all edges of partition $R_j$ as $E_j^H$. The set of edges of $H$ is simply $E_H = \bigcup_j E_j^H$. Note that $H$ is a multi-graph as there can be directed path from $u$ to $v$ in multiple partitions of the $r$-division. This completes the description of the compressed residual graph $H$. See Figure 5.2 for an example of the construction of $H$ for a partition.

**Planar Feasibility** Next, we will define a planar feasible matching. First, we decompose the edges of the matching based on the partition of the $r$-division they belong to. We denote
Figure 5.2: (a) A partition $\mathcal{R}_j$. The squares represent vertices of $B$ and the circles represent vertices of $A$. Free vertices are filled in. (b) The boundary-to-boundary edges of $H$ for $\mathcal{R}_j$. (c) The edges from $b_j$ to $\mathcal{K}_j$. (d) The edges from $\mathcal{K}_j$ to $a_j$. (e) There is a single edge from $b_j$ to $a_j$.

As $M_j$ the edges of $M$ that belong to partition $\mathcal{R}_j$. $M = \bigcup_{j=1}^J M_j$. For each partition $\mathcal{R}_j$, we maintain a dual weight $y_j(v)$ for every vertex $v \in V_j$. These dual weights $y_j(\cdot)$ along with $M_j$ form an $\mathcal{R}$-feasible matching. Additionally, we also store a dual weight $\tilde{y}(\cdot)$ for every $v \in V_H$. We say that the dual weights $\tilde{y}(\cdot)$ are $H$-feasible if they satisfy the following conditions. For each partition $\mathcal{R}_j$, and for every directed edge $(u, v) \in E^H$,

\[
\begin{align*}
\tilde{y}(u) + \tilde{y}(v) & \leq \phi(\overrightarrow{P}_{u,v,j}) & \text{if } (u, v) \in (B^H_j \times A^H_j) \\
-\tilde{y}(u) - \tilde{y}(v) & \leq \phi(\overrightarrow{P}_{u,v,j}) & \text{if } (u, v) \in (A^H_j \times B^H_j) \\
\tilde{y}(u) - \tilde{y}(v) & \leq \phi(\overrightarrow{P}_{u,v,j}) & \text{if } (u, v) \in (B^H_j \times B^H_j) \\
-\tilde{y}(u) + \tilde{y}(v) & \leq \phi(\overrightarrow{P}_{u,v,j}) & \text{if } (u, v) \in (A^H_j \times A^H_j).
\end{align*}
\]

As in the Gabow-Tarjan algorithm,

(a) For every vertex $v \in A_H$, $\tilde{y}(v) \leq 0$ and for every free vertex $v \in A^F_H$, $\tilde{y}(v) = 0$,

(b) For every vertex $v \in B_H$, $\tilde{y}(v) \geq 0$ and for every free vertex $v \in B^F_H$, $\tilde{y}(v) = y_{\text{max}}$,

where $y_{\text{max}} = \max_{v \in A_H \cup B_H} \tilde{y}(v)$.
Using (a) and (b), we can restate the $H$-feasibility conditions more compactly as

$$|\tilde{y}(u)| - |\tilde{y}(v)| \leq \phi(\overrightarrow{P_{u,v,j}}).$$ (5.6)

For a planar bipartite graph $G$, and an $r$-division $\mathcal{R}(G)$, we say that a matching $M$, a set of dual weights $y_j(\cdot)$ for the vertices of each partition $\mathcal{R}_j$, and a set of dual weights $\tilde{y}(\cdot)$ for the vertices $V_H$, form a planar feasible matching if in addition to (a) and (b), the following three conditions are satisfied.

(c) For every partition $\mathcal{R}_j$, the matching $M_j$ and dual weights $y_j(\cdot)$ form an $\mathcal{R}$-feasible matching,

(d) The dual weights $\tilde{y}(\cdot)$ are $H$-feasible,

(e) For each partition $\mathcal{R}_j$ and any $v \in \mathcal{K}_j$, $|\tilde{y}(v)| \geq |y_j(v)|$. For every vertex $a \in (V_j \setminus \mathcal{K}_j) \cap A_F$ (resp. $b \in (V_j \setminus \mathcal{K}_j) \cap B_F$), $|y_j(a)| = 0$ (resp. $y_j(b) \leq y_{\max}$).

Note that a boundary vertex has many different dual weights assigned to it, one for each of the partitions it belongs to. During the course of our algorithm, the magnitude of the dual weights of vertices in $H$ only increase. As we do not immediately update the dual weights of all vertices in $G$, for some partition $\mathcal{R}_j$ the dual weight $y_j(\cdot)$ may not reflect the updated dual weight. This condition is captured by (e). We refer to any perfect matching which is also a planar feasible matching as a planar optimal matching. For any edge $(u, v) \in E^H_j$, let the slack $s_H(u, v) = \phi(\overrightarrow{P_{u,v,j}}) - |\tilde{y}(u)| + |\tilde{y}(v)|$. Following our convention, we set $H'$ to be a graph identical to $H$ with the weight of every edge replaced by its slack.

At the end of the first step of our algorithm, we have a matching $M$ and a set of dual weights that form an $\mathcal{R}$-feasible matching. In Section 5.3.1, we describe how we can compute a planar feasible matching from this $\mathcal{R}$-feasible matching $M, y(\cdot)$. At the end of each scale, our
algorithm produces a planar optimal matching. In Section 5.3.2, we describe how to compute an \( R \)-optimal matching from this planar optimal matching. Using some of the procedures introduced in Sections 5.3.1 and 5.3.2, we describe the second step of our algorithm in section 5.3.3.

### 5.3.1 Computing a Planar Feasible Matching from an \( R \)-feasible matching

At the end of the first step, we have an \( R \)-feasible matching \( M, y(\cdot) \) that also satisfies \( y(u) = 0 \) for all \( u \in A_F \) and \( y(v) = y_{\text{max}} \) for all \( v \in B_F \). In this section, we will present an algorithm to compute a planar feasible matching from this \( R \)-feasible matching \( M \) and its set of dual weights \( y(\cdot) \). For every vertex \( v \in A \cup B \), and for every partition \( R_j \) such that \( v \in V_j \), we set \( y_j(v) = y(v) \). For every boundary vertex \( v \in K \), we set \( \tilde{y}(v) = y(v) \). We also set, for every partition \( R_j \), \( \tilde{y}(a_j) = 0 \) and \( \tilde{y}(b_j) = y_{\text{max}} \). Note that, from Lemma 5.3, conditions (a)-(c), (e) are trivially satisfied. The next lemma shows that dual weights \( \tilde{y}(\cdot) \) satisfy \( H \)-feasibility and therefore (d) holds.

**Lemma 5.4.** Consider a matching \( M_j \) and a set of dual weights \( y(\cdot) \) for a partition \( R_j \) such that \( M_j, y(\cdot) \) is \( R \)-feasible. Suppose the dual weights of all vertices of \( A \) in \( V_j \) are non-positive and the dual weights of all vertices of \( B \) in \( V_j \) are non-negative. For any two vertices \( u, v \in V_j \), let the directed path \( \overrightarrow{P}_{u,v,j} \) be a minimum net-cost path from \( u \) to \( v \) in \( R_j \). Then,

\[
|y(u)| - |y(v)| \leq \phi(\overrightarrow{P}_{u,v,j}). \tag{5.7}
\]

Furthermore,
5.3. Algorithm for Each Scale

\[ \phi(\overrightarrow{P}_{u,v,j}) - |y(u)| + |y(v)| = \sum_{(a,b) \in \overrightarrow{P}_{u,v,j}} s(a,b). \quad (5.8) \]

**Proof.** Let \( P = \overrightarrow{P}_{u,v,j} \). As \( u \) and \( v \) can belong to either \( A \) or \( B \), we need to consider four possible cases. We will provide a proof for the case where \( u \in B \) and \( v \in A \). An identical argument will also hold for the other three cases.

From the definition of net-cost, we have

\[
\phi(P) = \sum_{(a,b) \in P \setminus M} (c(a, b) + \delta_{ab}) - \sum_{(a,b) \in P \cap M} c(a, b) \\
= \sum_{(a,b) \in P \setminus M} (c(a, b) + \delta_{ab}) - \sum_{(a,b) \in P \cap M} (y(a) + y(b)).
\]

Since \( u \in B \) and \( v \in A \), both the first and the last edge of \( P \) are not in the matching \( M \). Therefore, we can write the above equation as:

\[
\phi(P) = \sum_{(a,b) \in P \setminus M} (c(a, b) + \delta_{ab} - y(a) - y(b)) + y(u) + y(v) \\
= y(u) + y(v) + \sum_{(a,b) \in P \setminus M} s(a, b) \\
= \sum_{(a,b) \in P \setminus M} s(a, b) + |y(u)| - |y(v)|.
\]

The last equality holds from the fact that dual weight of \( u \) is non-negative and the dual weight of \( v \) is non-positive.

\[ \square \]

We can also extend Lemma 5.4 to the entire graph, as shown in the following lemma.
**Lemma 5.5.** Consider a matching $M$ and a set of dual weights $y(\cdot)$ on the vertices of $G(A \cup B, E)$ such that $M, y(\cdot)$ is $R$-feasible. Suppose all vertices of $A$ have a non-positive dual weight and all vertices of $B$ have a non-negative dual weight. For any two vertices $u, v \in A \cup B$, let the directed path $\overrightarrow{P}_{u,v}$ be a minimum net-cost path from $u$ to $v$ in $G_M$. Then,

$$|y(u)| - |y(v)| \leq \phi(\overrightarrow{P}_{u,v}).$$

Furthermore,

$$\phi(\overrightarrow{P}_{u,v}) - |y(u)| + |y(v)| = \sum_{(a,b)\in \overrightarrow{P}_{u,v}} s(a,b).$$

Using the following lemma, we will provide an efficient procedure called CONSTRUCT to compute all the edges of the compressed residual graph $H$. After that, we also describe the data structures that we store these edges in.

**Lemma 5.6.** Let $R'_j$ be a directed graph identical to the directed graph $R_j$ except that the cost of an edge $(a, b)$ is the slack $s(a, b)$. Then, for a partition $R_j$ and any two vertices $u, v$ in $V_j$, the minimum net-cost directed path $\overrightarrow{P}_{u,v,j}$ from $u$ to $v$ in $R_j$ is also the minimum-cost directed path between $u$ and $v$ in $R'_j$. Furthermore, we can derive $\phi(\overrightarrow{P}_{u,v,j})$ from the dual weights $y(u), y(v)$ and the cost of the shortest path in $R'_j$.

**Proof.** We highlight our argument for the case where $u \in B$ and $v \in A$; the argument for every other case is identical.

For any directed path $P$ between $u$ and $v$, since $u \in B$ and $v \in A$ and, from the proof of
Lemma 5.4, we know that
\[
\phi(P) = \sum_{(a,b) \in P \setminus M} s(a,b) + y(u) + y(v) \tag{5.9}
\]
This is true for every path \(P\) from \(u\) to \(v\). Furthermore, the dual weights \(y(u)\) and \(y(v)\) in the above equation are the same for any path \(P\). Therefore, we conclude that the minimum net-cost path will also be the minimum-cost path between \(u\) and \(v\) in \(R'_j\). Let \(P^*\) be this minimum net-cost path. Since \(P^*\) also satisfies (5.9), and so the sum of the slacks of the edges on the path \(P^*\) along with \(y(u)\) and \(y(v)\) will give us the minimum net-cost between \(u\) and \(v\).

For the rest of the algorithm, we define the slack on any directed edge \((u, v) \in E^H_j\) to be \(\phi(P^*_{u,v,j}) - |\tilde{y}(u)| + |\tilde{y}(v)|\). From Lemma 5.6, it follows that slack of the edge \((u, v)\) is non-negative and exactly equal to \(\sum_{(a,b) \in \overline{P}_{u,v,j}} s(a,b)\). Following our convention, we use \(H'\) to denote the compressed residual graph with the same edge set as \(H\) but with the edge weights being replaced with their slacks.

Our initial choice of \(\tilde{y}(\cdot)\) is \(H\)-feasible. To assist in the execution of the second step of our algorithm, we explicitly compute the edges of \(H\) and store them and their slacks in a data structure. This data structure will assist us in the fast execution of Dijkstra’s algorithm on \(H'\).

Using Lemma 5.6 the CONSTRUCT procedure will, for any partition \(R_j\) of the \(r\)-division of \(G_M\), compute the edges of \(E^H_j\) in \(O(r \log r)\) time.

The Construct Procedure This procedure takes a partition \(R_j\) of \(G_M\) as input and constructs the edges of \(E^H_j\). We assume the matching \(M_j\) and the dual weights \(y_j(\cdot)\) are \(R\)-feasible. Let \(R'_j\) be a graph identical to \(R_j\), where each edge has a cost equal to its slack as
defined by the current dual assignment \( y_j(\cdot) \). We note that all edges in \( \mathcal{R}_j' \) are non-negative. Since the dual assignment is feasible with respect to \( M_j \), we know by Lemma 5.6 that the path of minimum net-cost between two vertices is also the path of minimum total slack in \( \mathcal{R}_j' \). Therefore, it is sufficient to compute the shortest path lengths in \( \mathcal{R}_j' \) and use (5.8) to compute the minimum net-cost path in constant time. We will describe how to compute the shortest paths in \( \mathcal{R}_j' \).

Recollect that there are four types of edges in \( E^H_j \). We first explain how to compute the boundary-to-boundary edges \((u, v) \in E^H_j\), that is, \((u, v) \in \mathcal{K}_j \times \mathcal{K}_j\). We construct a multiple-source shortest paths data structure described in [27] on \( \mathcal{R}_j' \) in \( O(|V_j| \log |V_j|) = O(r \log r) \) time. This structure can answer the following query in \( O(\log |V_j|) = O(\log r) \) time: Given any vertex \( s \) in \( V_j \), and a vertex \( t \) on the unbounded face of \( \mathcal{R}_j' \), what is the shortest \( s \) to \( t \) distance in \( \mathcal{R}_j' \)? Using this, we can obtain the boundary-to-boundary edges of \( \mathcal{R}_j \) by simply querying this data structure and using (5.8). We note that the boundary vertices may be located \( O(1) \) different holes. For each hole \( \mathcal{H} \) we can compute in \( O(r) \) time an embedding of \( \mathcal{R}_j \) such that this hole \( \mathcal{H} \) forms the infinite face of \( \mathcal{R}_j \) and then build the multiple source shortest paths data structure for this embedding with the vertices of \( \mathcal{H} \) as the sources. By querying this data structure, we can obtain all the boundary-to-boundary edges from vertices of \( \mathcal{H} \) to the boundary vertices not in \( \mathcal{H} \). Since there are a constant number of holes that contain all the boundary vertices of each partition, this will not increase the asymptotic complexity. To compute the other edges of \( E^H_j \) we execute Dijkstra’s algorithm three times. Let \( \mathcal{R}''_j \) be a graph where every edge of \( \mathcal{R}_j' \) is reversed. The first execution of Dijkstra’s algorithm is on \( \mathcal{R}_j' \) starting from the vertices of \( B_F \cap (V_j \setminus \mathcal{K}_j) \) to all vertices of \( \mathcal{K}_j \). For any vertex \( v \in \mathcal{K}_j \), since the dual weight of every vertex \( B_F \cap (V_j \setminus \mathcal{K}_j) \) is the same, the cost \( \phi(\overrightarrow{P}_{b_j,v,j}) \) can be simply computed using Lemma 5.6. Similarly, the second execution of Dijkstra’s algorithm is on \( \mathcal{R}''_j \) where we find distance to every vertex \( v \in \mathcal{K}_j \) from some
5.3. Algorithm for Each Scale

\( A_F \cap (V_j \setminus \mathcal{K}_j) \) and compute \( \phi(\overrightarrow{P}_{v,a_{j,j}}) \). The final edge from \( b_j \) to \( a_j \) can be computed by simply executing Dijkstra algorithm starting from the vertices of \( B_F \cap (V_j \setminus \mathcal{K}_j) \) and compute the shortest path cost to any point in \( A_F \cap (V_j \setminus \mathcal{K}_j) \). Since the dual weights of every free vertex of \( B_F \cap (V_j \setminus \mathcal{K}_j) \) is identical, and the dual weights of every free vertices of \( A_F \cap (V_j \setminus \mathcal{K}_j) \) is 0, using Lemma 5.6, we can obtain the weight \( \phi(\overrightarrow{P}_{b_{j,a_{j,j}}}) \). Together, these three execution of Dijkstra’s algorithm will compute the remaining edges of \( E_J^H \). The total time taken to compute all the edges of \( E_J^H \) is \( O(r \log r) \).

Monge Property of the Boundary-to-Boundary Edge Costs  

We say that any matrix \( M \) satisfies the Monge property if for any \( i < i' \) and \( j < j' \), \( M_{i,j} + M_{i',j'} \leq M_{i,j'} + M_{i',j} \). Fakcharoenphol and Rao described how we could represent the boundary-to-boundary edges into a number of groups so that the cost matrix for each group satisfies the Monge property. We will briefly provide the intuition for any partition \( \mathcal{R}_j \) where there are no holes. Fakcharoenphol and Rao [15] show that this description extends to any partition with \( O(1) \) holes with the same asymptotic complexity. Suppose all the boundary vertices \( \mathcal{K}_j \) are on the infinite face. We can partition the boundary-to-boundary edges into ordered bipartite groups. The first two groups are simply formed by dividing any clockwise ordering of all the boundary vertices into two equal parts (we refer to the first half as the left side and the second one as the right side). Now we create a group containing all edges going from a vertex in the left side to a vertex in the right side. We also create another group to represent all the edges going from a vertex in the right side to the left side. To capture the edges that go between two vertices of the left (resp. right) side, we simply recursively divide the left (resp. right) partition into two sets of equal sizes and again capture all the edges that go between them. Note that any boundary vertex participates in at most \( O(\log r) \) groups within a partition. Also, if we represent the edge-costs of any bipartite group as a matrix, then this matrix satisfies the Monge property.
Preprocessing the Boundary-to-Boundary Edges  For the sake of our algorithm, it is useful to build a data structure to store the boundary-to-boundary edges and their slacks in the ordered bipartite groups described above. For each ordered bipartite group, we have a cost matrix $\mathcal{M}$. For any edge $(u, v) \in E^H_j$ that participates in the bipartite group, we store its cost, i.e., $\phi(P_{u,v,j})$ in the cost matrix. Note that the slack, $\phi(P_{u,v,j}) - |y(u)| + |y(v)|$ of any edge can be computed in $O(1)$ time given its cost and the dual weights of its end vertices. Thus, we can implicitly represent a matrix $\mathcal{M}'$ based on $\mathcal{M}$ with slacks as costs. As shown in [15], shortest path distances in planar graphs between vertices of the ordered bipartite groups satisfy the Monge-property; see Figure 5.3. From Lemma 5.4, the slack on any edge $(u, v) \in E^H_j$ is the length of the shortest path from $u$ to $v$ in $\mathcal{R}'_j$, meaning $\mathcal{M}'$ also satisfies the Monge property. In the CONSTRUCT procedure, we build the Monge matrix range-minimum data structure, described in [23], on the slack matrix $\mathcal{M}'$ in time $O(\sqrt{x} \log x)$ where $x$ is the number of vertices in the bipartite group. For any vertex $v$ on the left, and any interval of nodes on the right, this data structure can answer the query, what is the minimum edge from $v$ to a vertex in the interval within the group? The Monge matrix range-minimum data structure in conjunction with the bipartite groups can be used to execute Dijkstra’s algorithm on $H'$ quickly, which helps us compute a minimum net-cost
5.3. Algorithm for Each Scale

augmenting path in $H$ efficiently.

**Lemma 5.7.** Given an $\mathcal{R}$ feasible matching $M_j$, $y_j(\cdot)$, the CONSTRUCT procedure builds the edges of $E_j^H$ in $O(r \log r)$ time. In addition, it stores the boundary-to-boundary edges in ordered bipartite groups in $O(r \log r)$ time where the edge cost matrix for each group satisfies the Monge property. It also builds a Monge-matrix range minimum data structure on the slacks of the boundary-to-boundary edges of $E_j^H$ for every bipartite group of $\mathcal{R}_j$ in $O(\sqrt{r} \log^2 r)$ time.

**Corollary 5.8.** Let $M$, $y(\cdot)$ be the $\mathcal{R}$-feasible matching computed at the end of the first step of our algorithm. Given $M$, $y(\cdot)$, we can use the CONSTRUCT procedure to compute the graph $H$ in $O(n \log r)$ time.

Using these data structures, one can compute shortest path between a vertex $b \in B_H^F$ to a vertex $a \in A_H^F$ in $H'$ using an algorithm identical to the one presented in [23] in $O((n/\sqrt{r}) \log^2 r)$ time. The algorithm also computes the distance $c_v$ to a vertex $v \in V_H$ from an arbitrary vertex of $B_H^F$. This algorithm contains three steps.

- In the first step, the algorithm finds the distance from $b_j$ to every boundary vertex of $\mathcal{K}_j$ within $\mathcal{R}_j'$. The minimum net-cost paths have corresponding edges explicitly pre-computed in $H$, and their slacks can be computed in $O(1)$ time. For each boundary vertex $v$, we set $c_v = \min_{R_j \in \mathcal{R}_j, (b_j, v) \in E^F_j} s_H(b_j, v)$. This takes $O(\sqrt{r})$ per partition and a total of $O(n/\sqrt{r})$ time for the entire graph $H'$.

- In the second step, the algorithm executes an implementation of Dijkstra’s given by Fakcharoenphol and Rao [15], which uses the bipartite groups and range-minimum query data structures. Fakcharoenphol and Rao have shown how to compute shortest path from a free vertex of $B$ to any boundary vertex in $O((n/\sqrt{r}) \log^2 r)$ time by
efficiently computing the next edge to add to the shortest path tree. For a description of their algorithm, see section 5.2 of [23].

- In the third step, we set the shortest path distance to every free internal vertex $a_j$ as $c_{a_j} = \min_{(u,a_j) \in E_H} c_u + s_H(u, a_j)$. As in step one, the net-costs of such paths are precomputed in $H$, and the slacks can be computed in $O(1)$ time.

- Let $\alpha$ be the vertex $a \in A_H^F$ with minimum $c_a$. We return the path to $\alpha$ in $H'$. We can compute this path from the shortest path tree in $O(n/\sqrt{r})$ time.

Therefore, we obtain the following lemma.

**Lemma 5.9.** Given the edges of $H$ stored in ordered bipartite groups along with a Monge-matrix range minimum data structure built on the slack matrix for each group, one can compute the shortest path (in terms of slack) between any free vertex of $B_H^F$ to any free vertex of $A_H^F$ in $O((n/\sqrt{r}) \log^2 r)$ time.

### 5.3.2 Computing an $\mathcal{R}$-Feasible Matching from a Planar Feasible Matching

Recollect that an $\mathcal{R}$-feasible matching was obtained at the end of the first step of our algorithm. In the previous section, we described how we can compute a planar feasible matching from this $\mathcal{R}$-feasible matching. The second step of our algorithm will compute a planar optimal matching from this planar feasible matching. At the end of each scale, however, we desire an $\mathcal{R}$-optimal matching. In this section, we describe how to convert any planar optimal matching to an $\mathcal{R}$-optimal matching in $O(n \log r)$ time. In fact, we will show that any planar feasible matching can be converted into an $\mathcal{R}$-feasible matching.
To assist with the presentation, for certain vertices of \( G \), we define their representative in \( H \) as follows. For any vertex \( v \in \mathcal{K}_j \cup (V_j \cap (A_F \cup B_F)) \), if \( v \in \mathcal{K}_j \) then the representative \( v' \) in \( H \) will be the same boundary vertex in \( V^H_j \), and if \( v \) is a free internal vertex of \( A \) (resp. \( B \)), then its representative \( v' \) in \( H \) will be \( a_j \) (resp. \( b_j \)).

In a planar feasible matching, any boundary vertex or a free internal vertex can have multiple dual weights, one corresponding to each of the partitions they belong to. In addition, their representative in \( H \) also have a dual weight. We introduce a synchronization procedure (called Sync) that will take a planar feasible matching along with a partition \( \mathcal{R}_j \) and update the dual weights \( y_j(\cdot) \) so that the new dual weights \( y_j(\cdot) \) and the matching \( M_j = M \cap E_j \) continue to be \( \mathcal{R} \)-feasible and for every boundary vertex and free internal vertex \( v \in \mathcal{K}_j \), \( y_j(v) = \tilde{y}(v') \) where \( v' \) is the representative of \( v \) in \( H \). We can convert a planar feasible matching into an \( \mathcal{R} \)-feasible matching by repeatedly invoking this procedure for every partition.

The Sync procedure is implemented as follows:

- Recollect that the graph \( \mathcal{R}'_j \) is a graph identical to \( \mathcal{R}_j \) with slacks as the edge costs. First, temporarily add a new source vertex \( s \) to \( \mathcal{R}'_j \). Let \( B_{in} \) denote all vertices of \( B \cap V_j \) that are matched inside \( \mathcal{R}_j \) and let \( A_{in} \) be the matches of vertices in \( B_{in} \). For any vertex \( v \in \mathcal{K}_j \cup (V_j \cap (A_F \cup B_F)) \), if \( v \not\in B_{in} \), we add a directed edge from \( s \) to \( v \). Otherwise if \( v \in B_{in} \), then let \( u \in A_{in} \) be its match in \( M_j \). We add a directed edge from \( s \) to \( u \). Consider any \( v \in \mathcal{K}_j \cup (V_j \cap (A_F \cup B_F)) \) and its representative \( v' \in V^H_j \). For any such boundary or free internal vertex \( v \), let \( \kappa_v = |\tilde{y}(v')| - |y_j(v)| \), and let \( \kappa = \max_{v \in \mathcal{K}_j \cup (V_j \cap (A_F \cup B_F))} \kappa_v \). If \( v \not\in B_{in} \), set the weight of the newly added edge from \( s \) to \( v \) to \( \kappa - \kappa_v \). Otherwise, if \( v \in B_{in} \), then set the weight of the edge from \( s \) to \( u \) to \( \kappa - \kappa_v \). This new graph has only non-negative edge costs.
• Execute Dijkstra’s algorithm on this graph beginning from the source vertex $s$. Let $\ell_v$ be the length of the shortest path from $s$ to $v$ as computed by this execution of Dijkstra’s algorithm. For each vertex $v \in V_j$, if $\ell_v \geq \kappa$, then do not change its dual weight. Otherwise, if $v \in B \cap V_j$, set the new dual weight to be $y_j(v) \leftarrow y_j(v) + \kappa - \ell_v$ and if $v \in A \cap V_j$, set its new dual weight to be $y_j(v) \leftarrow y_j(v) - \kappa + \ell_v$.

This completes the description of the Sync procedure. The Sync procedure executes Dijkstra’s algorithm on $R_j^r$ with an additional vertex $s$ and updates the dual weights of all the $O(r)$ vertices. The total time taken for this is $O(r \log r)$. To prove the correctness of this procedure, we have to show the following:

(1) For any vertex $v \in \mathcal{K}_j \cup (V_j \cap (A_F \cup B_F))$ and its representative $v' \in V_j^H$, the dual weight after the execution of Sync procedure, $\tilde{y}(v')$ is equal to $y_j(v)$.

(2) The new dual weights $y_j(\cdot)$ along with the matching $M_j$ form an $\mathcal{R}$-feasible matching.

Lemma 5.10. At the end of the Sync procedure, both (1) and (2) hold.

Proof. Let us denote the dual weights before and after applying the Sync procedure as $y'_j(\cdot)$ and $y_j(\cdot)$ respectively. Consider any $v \in \mathcal{K}_j \cup (V_j \cap (A_F \cup B_F))$. Recall the definition of the representative of $v$ in $H$. Either (i) $v \notin B_{in}$ or (ii) $v \in B_{in}$.

In case (i), let $v'$ be the representative of $v$ in $H$. We will show that the shortest path from $s$ to $v$ in $R_j^r$ is the edge $(s, v)$. Given this, if $v \in B$, the cost of $(s, v)$, i.e., $\ell_v = \kappa - y'_j(v) + \tilde{y}(v')$ and the new dual weight $y_j(v) = y'_j(v) - \kappa + \ell_v = \tilde{y}(v')$ and if $v \in A$, the cost of the edge $(s, v)$, i.e., $\ell_v = \kappa + y'_j(v) - \tilde{y}(v')$ and $y_j(v) = y'_j(v) + \kappa - \ell_v = \tilde{y}(v)$ as desired. Therefore, for case (i), it suffices to show that the shortest path from $s$ to $v$ is the edge $(s, v)$. We show this next. For the sake of contradiction, let the shortest path from $s$ to $v$ be strictly less than the cost of the edge $(s, v)$. Let $P$ be this shortest path from $s$ to $v$ and let $\hat{v}$ be the
first vertex that appears on $P$ after $s$. By construction, $\hat{v} \in \mathcal{K}_j \cup (V_j \cap (A_F \cup B_F))$ and let $\hat{v}'$ be the representative of $\hat{v}$ in $H$. Let $P'$ be the sub-path of $P$ with the vertex $s$ removed, i.e., $P'$ is a path from $\hat{v}$ to $v$. By the optimal substructure property of shortest paths, $P'$ has to be the smallest slack path from $\hat{v}$ to $v$ in $R'_j$. We know that the cost of $P$ in $R'_j$ is \[
abla = (\kappa - |\tilde{y}(\hat{v}')|) + |y'_j(\hat{v})| + \sum_{(a,b) \in P'} s(a,b). \] Since the cost of $P$ is smaller than the cost of the edge from $s$ to $v$, we have

\[
\kappa - |\tilde{y}(v')| + |y'_j(v)| > \kappa - |\tilde{y}(\hat{v}')| + |y'_j(\hat{v})| + \sum_{(a,b) \in P'} s(a,b),
\]

or

\[
|\tilde{y}(v')| - |\tilde{y}(\hat{v})| > |y'_j(v')| - |y'_j(v)| + \sum_{(a,b) \in P'} s(a,b) = \phi(P').
\]

The last statement follows from the fact that $P'$ is the minimum slack path from $v'$ to $v$ and since the matching $M_j$ and dual weights $y'_j(\cdot)$ form an $R$-feasible matching. The above inequality, $|\tilde{y}(v')| - |\tilde{y}(\hat{v})| > \phi(P')$, contradicts the $H$-feasibility condition.

In case (ii), let $u \in A_{in}$ be the match of $v$ and let $v'$ be the representative of $v$ in $H$. Using a proof similar to case (i), one can show that the shortest path from $s$ to $v$ is the path $\langle s, u, v \rangle$. Since $(u, v)$ is an edge in the matching $M_j$, $(u, v)$ has a cost of 0 in $R'_j$. So, the shortest path cost from $s$ to $v$ is equal to the cost of the edge $(s, u)$, i.e., $\ell_u = \kappa - y'_j(v) + \tilde{y}(v')$ and the new dual weight $y_j(v) = y'_j(v) - \kappa + \ell_v = \tilde{y}(v')$. Thus, we have shown that at the end of Sync procedure, (1) holds.

To prove (2), we need to show that the matching $M_j$ along with the new dual weights $y_j(\cdot)$ are $R$-feasible. We show this for the two cases (i) $(a,b) \in M_j$ and (ii) $(a,b) \in E_j \setminus M_j$ separately. For case (i), from $R$-feasibility of $M_j$ and $y'_j(\cdot)$, $y'_j(a) + y'_j(b) = c(a,b)$. We note that by construction of $R'_j$ (that includes $s$) $b$ has exactly one incoming edge from $a$ to $b$. From $R$-feasibility, the slack on the edge from $a$ to $b$ edge is 0. Therefore, $\ell_a = \ell_b$. So, the
dual weights as updated by the Sync procedure satisfy

\[ y_j(a) + y_j(b) = y_j'(a) - \kappa + \ell_a + y_j'(b) + \kappa - \ell_b = c(a, b), \]

implying that \((a, b)\) satisfies (5.2).

For case (ii), the edge \((a, b)\) is not in the matching \(M_j\), and therefore the edge is directed from \(b\) to \(a\). Consider if \(a \in K_j \cup (V_j \setminus K_j) \cap (A_F \cup B_F)\). The shortest path cost from \(s\) to \(a\) satisfies \(\ell_a \leq \ell_b + s(a, b)\); otherwise, there would be a shorter path to \(a\) through \(b\). Since \(M_j\) and \(y_j'(\cdot)\) are \(\mathcal{R}\)-feasible, we have \(y_j'(a) + y_j'(b) + s(a, b) = c(a, b) + \delta_{ab}\). Therefore, the dual weights as updated by the Sync procedure satisfy

\[
\begin{align*}
y_j(a) + y_j(b) &= y_j'(a) - \kappa + \ell_a + y_j'(b) + \kappa - \ell_b \\
&= y_j'(a) + y_j'(b) + \ell_a - \ell_b \\
&\leq y_j'(a) + y_j'(b) + s(a, b) \\
&= c(a, b) + \delta_{ab},
\end{align*}
\]

implying that every non matching edge \((a, b)\) satisfy the \(\mathcal{R}\)-feasibility condition (5.1).

Therefore, the matching \(M_j\) and the new dual weights \(y_j(\cdot)\) are \(\mathcal{R}\)-feasible completing the proof of (2).

\[\square\]

Given the correctness of the Sync procedure, we can convert a planar feasible matching into an \(\mathcal{R}\)-feasible matching by simply applying the Sync procedure to all the partitions. This will guarantee that the dual weight of any vertex \(v \in A \cup B\), is the same across all partitions it participates in and also equal to the dual weight of its representative in \(H\) if one exists.
Let $y(v)$ be this dual weight. Every edge $(u, v)$ of the graph $G$ belongs to some partition in the $r$-division $\mathcal{R}(G)$ and so it must be $\mathcal{R}$-feasible. Therefore the matching $M$ along with the dual weights $y(\cdot)$ form an $\mathcal{R}$-feasible matching. Also note that every free vertex $v \in A_F$ will have a dual weight $y(v) = 0$ and every free vertex $v \in B_F$ will have a dual weight $y(v) = y_{\text{max}}$. This follows from (1), and the fact that the input to $\text{SYNC}$ is a planar feasible matching.

**Lemma 5.11.** Given a planar feasible matching $M$, we can convert it into an $\mathcal{R}$-feasible matching $M$ with a set of dual weights $y(\cdot)$ in $O(n \log r)$ time. Furthermore, for every $a \in A_F$, $y(a) = 0$ and for every $b \in B_F$, $y(b) = y_{\text{max}}$.

### 5.3.3 Second Step of the Algorithm

In this section, given a planar feasible matching with $O(n/\sqrt{r})$ free vertices, we show how to compute a planar optimal matching in $O((n^2/r) \log^2 r + n\sqrt{r} \log r \log n)$ time. Our algorithm will then convert the planar optimal matching into an $\mathcal{R}$-optimal matching in time $O(n \log r)$, by Lemma 5.11. The second step of our algorithm will consist of multiple phases. In each phase, the size of the matching will increase by one while maintaining planar feasibility. Recall that $H'$ is the graph identical to $H$ with slacks as the edge cost. We will execute the following during each phase.

1. Compute the shortest path distance $c_v$, to every $v \in V_H$ from some free vertex of $B_H^F$ using Lemma 5.9. Let $\alpha = \arg \min_{v \in A_H^F} c_v$ and let $P$ be this shortest path to $\alpha$.

2. Let $\mathcal{T}$ be the set of vertices $v \in V_H$ such that $c_v \leq c_\alpha$. For every vertex $v \in \mathcal{T} \cap B_H$ we increase the dual weight $\tilde{y}(v) \leftarrow \tilde{y}(v) + c_\alpha - c_v$, and for every vertex $v \in \mathcal{T} \cap A_H$ we decrease the dual weight $\tilde{y}(v) \leftarrow \tilde{y}(v) - c_\alpha + c_v$. Note that in either case, this results in an increase in the magnitude of $\tilde{y}(v)$. 
(iii) Execute the Augment procedure (described below). This procedure will augment the matching while maintaining planar feasibility.

(iv) (i)–(iii) will change $\bar{y}(\cdot)$ and therefore also the slack on the edges of $H'$. Thus, the algorithm rebuilds the Monge matrix range minimum data structure for every partition in $\mathcal{R}(G)$. By using Lemma 5.7, this can be done in $O(\sqrt{r} \log^2 r)$ time per partition and $O((n/\sqrt{r}) \log^2 r)$ time in total.

Step 2(i) and Step 2(ii) modify the dual weight of vertices in $H$. Lemma 5.12 shows that this modification does not violate $H$-feasibility. In Lemma 5.13, we show that, after execution of Step 2(ii) the edges of $P$ will have zero slack with respect to $\bar{y}(\cdot)$. Next, we will describe the details of Augment procedure.

**Augment Procedure** After executing Step 2(ii), we have a path $P$ in $H$ and a set of $H$-feasible dual weights $\bar{y}(\cdot)$ such that all edges in $P$ have zero slack. Let $b$ be the first vertex of $P$ and let $a$ be the last vertex of $P$. Now we need to use $P$ to augment the matching. For any edge $(u, v)$ on the path $P$, if $(u, v)$ is an edge of $E^H_j$, we refer to $\mathcal{R}_j$ as an affected partition. Let $\mathcal{A}$ be the set of all such affected partitions. We execute the following operations during the Augment procedure.

(a) Call the Sync procedure on every affected partition $\mathcal{R}_j \in \mathcal{A}$. This procedure will update the dual weights $y_j(\cdot)$.

(b) For any edge $(u, v) \in P$, suppose $(u, v)$ is an edge in $E^H_j$. Project $(u, v)$ to obtain the path $\overrightarrow{P}_{u,v,j}$. This can be done by executing a BFS on the admissible graph of the partition $\mathcal{R}_j$. Combine all the projections and obtain an augmenting path $\overrightarrow{P}$ of admissible edges in the residual graph $G_M$. Next, augment the matching $M$ along $\overrightarrow{P}$. Augmenting the matching will change the residual graph of every affected partition $\mathcal{R}_j \in \mathcal{A}$. Lemma 5.14
shows that $\overrightarrow{P}_{u,v,j}$ consists of admissible edges with respect to $M_j$, $y_j(\cdot)$ after the call to Sync. Lemma 5.15 shows that $\overrightarrow{P}$ is a simple path, i.e., it has no self-intersections.

(c) For any vertex $v \in A \cap \overrightarrow{P}$, let $u$ be the vertex to which $v$ is matched after the augmentation. Set $y_j(v) \leftarrow y_j(v) - \delta_{uv}$ for all partitions $\mathcal{R}_j \in \mathcal{A}$ which contain $v$. If $v \in A_H \cap P$ and $v \neq a_{j'}$, then set $\tilde{y}(v) \leftarrow \tilde{y}(v) - \delta_{uv}$. Lemma 5.16 shows that after this operation, for every affected partition $\mathcal{R}_j$, the matching $M_j$ and $y_j(\cdot)$ is $\mathcal{R}$-feasible. Lemma 5.17 shows that after this operation, $\tilde{y}(\cdot)$ continues being $H$-feasible.

(d) Call the CONSTRUCT procedure on every affected partition $\mathcal{R}_j$ to rebuild the edges of $H$ for each affected partition. Due to the augmentation, there may be several new edges added to $H$ and several other edges whose costs have changed. We refer to the set $\mathcal{E}$ of these edges as the affected edges of $H$.

5.4 Analysis of the Algorithm

5.4.1 Correctness

In this section we show that the matching the algorithm maintains is planar feasible at the end of each phase of the second step. We also show that the augmenting path computed in each phase is a simple path. Since there are only $O(n/\sqrt{r})$ unmatched vertices after the first step of the algorithm, it will conduct $O(n/\sqrt{r})$ phases, where each phase increases the size of the matching by one. Therefore, after $O(n/\sqrt{r})$ phases we will have a planar optimal matching. By Lemma 5.11, a planar optimal matching can generate an $\mathcal{R}$-optimal matching by use of the Sync procedure.
Overview of the Correctness Proof  To prove that the matching generated at the end of each phase is planar feasible, we must show conditions (a)–(e) of planar feasibility are satisfied. Lemma 5.19 shows that conditions (a) and (b) of planar feasibility hold at the end of every phase. Condition (c) requires \( \mathcal{R} \)-feasibility of every partition at the end of each phase. We show, in Lemma 5.16, that condition (c) holds. To show that condition (d) holds, we have to establish that the dual weights \( \tilde{y}(\cdot) \) are \( H \)-feasible at the end of each phase. In each phase, Step 2(i), Step 2(ii) and the AUGMENT procedure modify the dual weights \( \tilde{y}(\cdot) \) of \( H \). Lemma 5.12 will show that the modifications by Step 2(i) and Step 2(ii) do not violate \( H \)-feasibility. In Lemma 5.17, we show that the AUGMENT procedure does not violate the \( H \)-feasibility and condition (d) holds. Finally, we establish that condition (e) holds at the end of each phase in Lemma 5.18. Together, these lemmas prove that we maintain a planar feasible matching at the end of each phase.

Also, Lemma 5.15 and Lemma 5.13 show that the augmenting path computed by the algorithm in each phase does not have any self-intersections.

**Lemma 5.12.** Let us assume the dual weights \( \tilde{y}(\cdot) \) are \( H \)-feasible prior to an execution of Step 2(ii). After executing Step 2(ii), \( \tilde{y}(\cdot) \) remains \( H \)-feasible.

**Proof.** We show that after the execution of Step 2(ii), for every directed edge \((u, v)\) in \( E_j^H \),

\[
|\tilde{y}(u)| - |\tilde{y}(v)| \leq \phi(\vec{P}_{u,v,j}). \tag{5.10}
\]

Let \( \tilde{y}_0(\cdot) \) be the dual weights prior to executing Step 2(ii). Initially, we have that,

\[
|\tilde{y}_0(u)| - |\tilde{y}_0(v)| + s_H(u, v) = \phi(\vec{P}_{a,v,j}).
\]

Additionally, Step 2(i) produces a distance assignment \( c \) such that for the directed edge
5.4. Analysis of the Algorithm

\[(u, v),\]

\[c_u + s_H(u, v) \geq c_v \quad (5.11)\]

Recall that for any vertex \(v \in V_H\), \(c_v\) is the minimum distance in \(H'\) to \(v\) from some vertex of \(B_{H}^{F}\). Let \(\alpha\) be the vertex \(v \in A_{H}^{F}\) with minimum \(c_v\). Let \(T\) be the set of vertices \(v\) for which \(c_v \leq c_{\alpha}\). We consider four cases:

1. \(u \in T, v \in T\). We increase the magnitude of \(\tilde{y}(u)\) by \(c_{\alpha} - c_u\) and increase the magnitude of \(\tilde{y}(v)\) by \(c_{\alpha} - c_v\). Thus, we have,

\[|\tilde{y}(u)| - |\tilde{y}(v)| \leq (|\tilde{y}_0(u)| + c_{\alpha} - c_u) - (|\tilde{y}_0(v)| + c_{\alpha} - c_v)\]

\[\leq (|\tilde{y}_0(u)| - |\tilde{y}_0(v)|) + (c_v - c_u).\]

Combining (5.11) with (5.12) yields,

\[|\tilde{y}(u)| - |\tilde{y}(v)| \leq |\tilde{y}_0(u)| - |\tilde{y}_0(v)| + s_H(u, v)\]

\[= \phi(\overrightarrow{P}_{u,v,j}),\]

and (5.10) holds.
(2) $u \in \mathcal{T}, v \notin \mathcal{T}$. $|\tilde{y}(u)| = |\tilde{y}_0(u)| + c_u - c_v$ and $|\tilde{y}(v)| = |\tilde{y}_0(v)|$. Since $v \notin \mathcal{T}$, $c_u < c_v$, and therefore $c_u - c_v < c_v - c_u$. From (5.11) we have that $c_v - c_u \leq s_H(u, v)$. Therefore,

$$|\tilde{y}(u)| - |\tilde{y}(v)| < |\tilde{y}_0(u)| - |\tilde{y}_0(v)| + s_H(u, v)$$

$$= \phi(\overrightarrow{P}_{u,v,j}),$$

and (5.10) holds.

(3) $u \notin \mathcal{T}, v \in \mathcal{T}$. In this case, $|\tilde{y}(u)| = |\tilde{y}_0(u)|$ and $|\tilde{y}(v)| \geq |\tilde{y}_0(v)|$. Therefore,

$$|\tilde{y}(u)| - |\tilde{y}(v)| \leq |\tilde{y}_0(u)| - |\tilde{y}_0(v)| + s_H(u, v)$$

$$= \phi(\overrightarrow{P}_{u,v,j}),$$

and (5.10) holds.

(4) $u \notin \mathcal{T}, v \notin \mathcal{T}$. In this case, $\tilde{y}(u) = \tilde{y}_0(u)$ and $\tilde{y}(v) = \tilde{y}_0(v)$. Therefore, (5.10) holds.

\[\square\]

**Lemma 5.13.** Let $P$ be the path found in $H$ during Step 2(i). Then after Step 2(ii), all edges in $P$ have zero slack.

**Proof.** We show that for any edge $(u, v) \in P$,

$$|\tilde{y}(u)| - |\tilde{y}(v)| = \phi(\overrightarrow{P}_{u,v,j})$$

Since $(u, v)$ is part of the shortest path, it must be true that $c_u + s_H(u, v) = c_v$. We increase
the magnitude of $\tilde{y}(u)$ by $c_\alpha - c_u$ and we increase the magnitude of $\tilde{y}(v)$ by $c_\alpha - c_v$. Thus, $|\tilde{y}(u)| - |\tilde{y}(v)|$ increased by $s_H(u, v)$. Since the matching $M_j$ is $\mathcal{R}$-feasible, by Lemma 5.4, prior to Step 2(ii) we have $|\tilde{y}(u)| - |\tilde{y}(v)| + s_H(u, v) = \phi(\overrightarrow{P}_{u,v,j})$. Therefore, after Step 2(ii) we have $|\tilde{y}(u)| - |\tilde{y}(v)| = \phi(\overrightarrow{P}_{u,v,j})$.

The following lemma shows that the edges of an augmenting path are admissible prior to augmentation.

**Lemma 5.14.** After the execution of the SYNC procedure on all partitions in $\mathcal{A}$, for every partition $\mathcal{R}_j \in \mathcal{A}$, any edge $(u, v) \in \overrightarrow{P} \cap E_j$ is admissible with respect to $M_j, y_j(\cdot)$. Here $\overrightarrow{P}$ is the augmenting path computed in part (b) of the AUGMENT procedure.

**Proof.** Let $P$ be the path found in $H$ in Step 2(i). By Lemma 5.13, all edges in $P$ have zero slack with respect to the dual weights in $H$. In part (a), we call the SYNC procedure on every $\mathcal{R}_j \in \mathcal{A}$ which, by Lemma 5.10, creates a matching $M_j$ and dual weights $y_j(\cdot)$ which are both feasible, and synchronized with the weights $\tilde{y}(\cdot)$ in $H$. Since the dual weights of $H$ did not change during the SYNC procedure, for any edge $(u, v) \in E_j^H$, $(u, v)$ still has zero slack, i.e., $|\tilde{y}(u)| - |\tilde{y}(v)| = |y_j(u)| - |y_j(v)| = \phi(\overrightarrow{P}_{u,v,j})$. So, by Lemma 5.4, the sum of slacks on all edges of $\overrightarrow{P}_{u,v,j}$, for every $(u, v) \in P$, must be zero. Since slacks are non-negative, every edge in $\overrightarrow{P}_{u,v,j}$ must have zero slack, and therefore be admissible.

By our construction of $H$, any path through $H$ maps to a path in $G_M$. The augmenting path $P$ found in $H$ is a simple path, but it is conceivable that the corresponding projection $\overrightarrow{P}$ is not. However, the following lemma shows that $\overrightarrow{P}$ is in fact a simple path.

**Lemma 5.15.** Let $P$ be a path found in $H$ during Step 2(i) of the algorithm and let $\overrightarrow{P}$ be the projection of $P$ onto $G_M$ (computed in part (b) of the AUGMENT procedure). Then, $\overrightarrow{P}$ is a simple path, i.e., $\overrightarrow{P}$ has no self-intersections.
Proof. We show that our algorithm never introduces a cycle of admissible edges in $G_M$. Since the path $\overrightarrow{P}$ chosen by our algorithm consists only of admissible edges, $\overrightarrow{P}$ is a simple path.

The first step of our algorithm executes $O(\sqrt{r})$ iterations of the Gabow-Tarjan Algorithm. The Gabow-Tarjan algorithm never creates any cycle of admissible edges [18, Lemma 2.3]. Therefore, after the first step of our algorithm, the algorithm has a 1-feasible matching $M$ and dual weights $y(\cdot)$ with no cycle of admissible edges. The admissibility conditions for a 1-feasible matching are different from the admissibility conditions for an $R$-feasible matching. However, given any 1-feasible matching, an edge that is not admissible is also not admissible as defined for $R$-feasibility. Therefore, there is no cycle of admissible edges with respect to the $R$-feasible matching $M$ and $y(\cdot)$. The algorithm converts this $R$-feasible matching into a planar feasible matching, and constructs the compressed residual graph $H$. At the start, $H$ will not have any cycle consisting only of zero slack edges, simply because the projection of any such cycle would form a cycle of admissible edges in the residual graph $G_M$.

Note that during any phase of the algorithm, if the path $\overrightarrow{P}$ computed contains a cycle $\overrightarrow{C}$ of admissible edges (in $G_M$) then either (1) $\overrightarrow{C}$ contains edges of only one partition or (2) $\overrightarrow{C}$ contains edges from more than one partition. We present our proof for case (2). The proof for case (1) uses similar arguments but is significantly simpler. In case (2), the boundary vertices on $\overrightarrow{C}$ partition the cycle into paths where each path is between two successive boundary nodes. We can construct a cycle $C$ in $H$ by replacing this path between two successive boundary nodes in the cycle $\overrightarrow{C}$ with a directed edge of $H$ between them. Before augmentation along $\overrightarrow{P}$, due to execution of the Sync procedure, the dual weight $y(v)$ of any boundary vertex and $\tilde{y}(v')$ of its representative $v'$ in $H$ are the same. Since all the edges of $\overrightarrow{C}$ are admissible, from Lemma 5.4, the slack on the edges of $C$ is zero. Therefore, to show that case (2) does not occur, it suffices to show that during the course of the algorithm, $H$ does not have any cycle consisting only of only zero slack edges.
As already claimed, after the first step of the algorithm, $H$ does not contain any cycle of zero slack edges. The dual weights of $H$ change in either step 2(ii) or in the Augment procedure. We will argue that neither of these steps will create a cycle consisting only of zero slack edges.

Using a proof by contradiction, we show that step 2(ii) cannot create a zero slack cycle in $H$. For the sake of contradiction, consider some phase of the algorithm such that a cycle $C'$ has a non-zero slack at the start of the phase but has only zero slack edges at the end of the phase. In step 2(i), we compute shortest distance from vertices of $B_H^F$ to every vertex in $H$, including all vertices on $C'$. If all the vertices of $C'$ have the same shortest path distance, then the change in the magnitude of the dual weights of all these vertices will be the same, and therefore the slack on every edge remains the same, implying that not all the edges of $C'$ are zero slack at the end of Step 2(ii). Therefore, we can assume that at least one vertex on the cycle $C'$ has a distance that is different from other vertices on $C'$. Note that when the shortest path distances to vertices of $C'$ are different, then there will be at least one edge of the cycle $C'$ from a vertex $u$ to a vertex $v$ such that the shortest distance to $u$ as computed by step 2(i) is greater than the shortest distance to $v$. In this case, the magnitude of the dual weight of $v$ changes by a larger value than the magnitude of the dual weight of $u$ and therefore, the slack on the edge $(u, v)$ only increases. This implies that $C'$ will have at least one edge with a non-zero slack at the end of Step 2(ii). So, step 2(ii) will not introduce a cycle that consists only of zero slack edges in $H$.

The Augment procedure may introduce new edges and change the costs of some edges in $H$. We divide this set of affected edges $\mathcal{E}$ into two groups. Edges of the first group are affected edges that are directed from some vertex $u \in A_H \cap P$ to $v$ ($P$ is the path found by Step 2(i)). The remaining edges in the affected set are of the second type. As we show in Lemma 5.17, all edges of the first type have a slack of at least zero and edges of the second
type will have a slack of at least 1. So, these affected edges of the second group do not create a cycle of zero slack edges in \( H \). For any affected edge \((u, v)\) in the first group, we show that every incoming edge to \( u \) has a slack of at least 1. This is because, after augmenting the matching, we increase the magnitude of the dual weight of \( u \) (equivalently, since \( u \in A_H \), reducing the dual weight of \( u \)), we guarantee that all edges of the residual graph that enter \( u \) have a slack of at least 1. So, although an affected edge of type 2 \((u, v)\) can have zero slack in \( H \), since all incoming edges to \( u \) in \( H \) have a slack of at least 1, it cannot participate in a cycle containing only zero slack edges. Therefore, at the end of this phase, there is no cycle in \( H \) containing only zero slack edges leading to a contradiction. Hence, case (2) will never occur. Since both (1) and (2) cannot occur, the path \( \vec{P} \) will not contain any cycle of admissible edges, i.e., \( \vec{P} \) is a simple path.

We note that the matchings \( M_j \) and the dual weights \( y_j(\cdot) \) only change during Step 2(iii) of the algorithm, that is, during \textsc{Augment}. The following lemma argues that the \textsc{Augment} procedure does not violate \( \mathcal{R} \)-feasibility in any \( \mathcal{R}_j \) for which \( M_j \) or \( y_j(\cdot) \) was modified. Therefore, planar feasibility condition (c) is satisfied.

**Lemma 5.16.** After the \textsc{Augment} procedure, for any affected partition \( \mathcal{R}_j \), the matching \( M_j \) and the dual weights \( y_j(\cdot) \) form an \( \mathcal{R} \)-feasible matching.

**Proof.** Let \( \vec{P} \) be the augmenting path that the algorithm augments along. By Lemma 5.14, the \textsc{Sync} procedure produces an \( \mathcal{R} \)-feasible \( M_j \), \( y_j(\cdot) \) for every affected partition where the edges of \( \vec{P} \) are admissible. We will show that the edges of \( \vec{P} \) are feasible after augmentation.

After augmentation, for any \( v \in A \cap \vec{P} \), let \( u \) be the vertex to which \( v \) is matched. Then we reduce \( y_j(v) \) by \( \delta_{uv} \) for each \( \mathcal{R}_j \in \mathcal{A} \). First consider any edge \((u, v) \notin M_j \). Prior to augmentation, \((u, v) \) was an admissible edge in the matching, so \( y_j(u) + y_j(v) = c(u, v) \). The reduction of the dual weight of \( v \) only decreases the sum \( y_j(u) + y_j(v) \), meaning the
feasibility condition in (5.1) is satisfied. However, we must still show that condition (5.2) is satisfied for all edges \((u, v) \in M_j\). Consider an edge \((u, v) \in M_j\). We know that \((u, v)\) was admissible prior to augmentation from Lemma 5.14. Let \(y'_j(\cdot)\) be the dual weights prior to augmentation. Then \(y'_j(u) + y'_j(v) = c(u, v) + \delta_{uv}\) since \((u, v)\) was not in the matching prior to augmentation. Therefore, \(y_j(u) + y_j(v) = c(u, v)\), and \(M_j, y_j(\cdot)\) form an \(\mathcal{R}\)-feasible matching.

From Lemma 5.12 we have that \(\tilde{y}(\cdot)\) is \(H\)-feasible after Step 2(ii). The \textsc{Augment} procedure, however, modifies the dual weights \(\tilde{y}(\cdot)\), may add new edges to \(H\), and may edit the cost of some of the old edges. Let \(\mathcal{E}\) be the set of affected edges which are either newly added or whose cost has changed. The following lemma establishes that, despite the changes to \(H\), the dual weights \(\tilde{y}(\cdot)\) remains \(H\)-feasible after \textsc{Augment}, meaning requirement (d) of planar-feasibility is satisfied. Furthermore, every affected edge of \(\mathcal{E}\) has a slack of at least 1.

**Lemma 5.17.** After the \textsc{Augment} procedure, the dual assignment \(\tilde{y}(\cdot)\) is \(H\)-feasible. Let \(\mathcal{E}\) be the edges that were either newly added to \(H\) or whose cost changed after \textsc{Augment}. We claim that every edge \((u, v) \in \mathcal{E}\) directed from \(u\) to \(v\) in \(H\) has \(s_H(u, v) \geq 0\) and if \(u \notin A_H\), then \(s_H(u, v) \geq 1\).

**Proof.** We address \(H\)-feasibility separately for \(H\)-edges of each piece \(\mathcal{R}_j\). There are two cases: (i) \(\mathcal{R}_j \notin \mathcal{A}\) and (ii) \(\mathcal{R}_j \in \mathcal{A}\). First consider case (i). Then no edge \((u, v) \in E^H_j\) has experienced a change in net-cost, since the augmenting path did not pass through \(\mathcal{R}_j\). However, it is possible that the dual weight \(\tilde{y}(u)\) or \(\tilde{y}(v)\) changed. Specifically, step (c) of \textsc{Augment} increases the magnitude of \(\tilde{y}(x)\) for any vertex of \(x \in A_H\) along the augmenting path. Consider if \(u \in A_H\). Then the first edge of \(\overrightarrow{P}_{u,v,j}\) must be a matching edge that was along the augmenting path, and \(\mathcal{R}_j\) would be an affected piece, contradicting the assumption
of case (i). Therefore, we can assume that \( \tilde{y}(u) \) remains constant. It is possible that the magnitude of \( \tilde{y}(v) \) increases. However, this will only increase \( s_H(u,v) \). Therefore, we can conclude that \( H \)-feasibility is maintained for case (i).

Next, we address case (ii). Since \( R_j \in A \), the SYNC procedure was invoked on \( R_j \). By Lemma 5.10, this means for any vertex \( x \in V_j \) and its representative vertex \( x' \), \( y_j(x) = \tilde{y}(x') \).

Step (c) of AUGMENT modifies the dual weights \( y_j(x) \) and \( \tilde{y}(x') \) by the same amounts, except if \( x' = a_j \). However, if \( x' = a_j \), then \( x \) was the last vertex of the augmenting path, \( x \) is now matched, and therefore \( x \) is no longer represented by \( a_j \). Therefore, for all vertices \( x \in V_j \) with representative vertex \( x' \), \( y_j(x) = \tilde{y}(x') \) after AUGMENT.

Consider any edge \( (u, v) \in E_j^H \). By Lemma 5.16, the matching after AUGMENT is \( R \)-feasible, and we can invoke (5.8) of Lemma 5.4. Specifically, since \( \tilde{y}(u) = y_j(u) \) and \( \tilde{y}(v) = y_j(v) \), we get that

\[
s_H(u,v) = \phi(\vec{P}_{u,v,j}) - |\tilde{y}(u)| + |\tilde{y}(v)| = \sum_{(a,b) \in \vec{P}_{u,v,j}} s(a,b).
\]

Since \( y_j(\cdot) \), \( M_j \) is \( R \)-feasible, each edge of \( E_j \) has non-negative slack. Therefore, \( s_H(u,v) \) is also non-negative, and \( H \)-feasibility holds for all edges of \( E_j^H \). This completes case (ii).

We next argue the properties on the edges of \( E \) hold. Trivially, since \( H \)-feasibility holds, \( s_H(u,v) \geq 0 \) for any \( (u,v) \in E \). However, it remains to show that \( s_H(u,v) \geq 1 \) for any \( (u,v) \in E \) such that \( u \notin A_H \). Let \( \vec{P}_{u,v,j} \) (resp. \( \vec{P}'_{u,v,j} \)) be the shortest path from \( u \) to \( v \) in \( R_j \) prior to (resp. after) augmentation. There are two cases, (1) \( \vec{P}'_{u,v,j} \) does not share an edge with the augmenting path and (2) \( \vec{P}'_{u,v,j} \) does share an edge with the augmenting path.

Consider case (1). Since \( \vec{P}'_{u,v,j} \) does not share an edge with the augmenting path, it also existed prior to augmentation. Therefore, \( \phi(\vec{P}_{u,v,j}) \leq \phi(\vec{P}'_{u,v,j}) \). Since \( (u,v) \in E \), by definition, \( \phi(\vec{P}_{u,v,j}) \neq \phi(\vec{P}'_{u,v,j}) \). Therefore, \( \phi(\vec{P}_{u,v,j}) < \phi(\vec{P}'_{u,v,j}) \). The dual weights \( \tilde{y}(u) \) and \( \tilde{y}(v) \) did not change because step (c) of AUGMENT only changes dual weights along
the augmenting path, and, by our assumption for case (1), \( \vec{P}'_{u,v,j} \) has no vertices on the augmenting path. Therefore, \( \phi(\vec{P}'_{u,v,j}) - |\vec{y}(u)| + |\vec{y}(v)| < \phi(\vec{P}'_{u,v,j}) - |\vec{y}(u)| + |\vec{y}(v)| \) and \( s_H(u,v) \) only increased as a result of the augmentation. Since \( H \)-feasibility held prior to augmentation, \( s_H(u,v) \) must be at least 1 after augmentation. This completes case (1).

Next, consider case (2). \( \vec{P}'_{u,v,j} \) must intersect the augmenting path at some edge, implying there is some vertex \( x \in A \) on \( \vec{P}'_{u,v,j} \) that was also on the augmenting path. Since we assume, \( u / \notin A \), \( u \neq x \), and there must be at least one edge \((x', x) \in \vec{P}'_{u,v,j} \) going into \( x \). \( s(x', x) \geq 0 \) prior to step (c) of AUGMENT. Step (c) of AUGMENT decreases \( y(x) \) by at least 1 because \( x \) was on the augmenting path. This increases the slack on \((x', x) \) (a non-matching edge), and therefore, \( s(x', x) \geq 1 \). Since all other edges of \( \vec{P}'_{u,v,j} \) have non-negative slack, we conclude that \( s_H(u,v) \geq 1 \), completing case (2). □

The following lemma shows that requirement (e) for planar feasibility is satisfied.

**Lemma 5.18.** Assuming we begin the phase with a planar-feasible matching, at the end of that phase, for each vertex \( v \in V_H \) and for every partition \( R_j \) such that \( v \in K_j \), \( |\vec{y}(v)| \geq |y_j(v)| \) for all \( j \). Furthermore, if \( v = a_j \) (resp. \( v = b_j \)), then for every vertex \( v' \in (V_j \setminus K_j) \cap A_F \) (resp. \( v' \in (V_j \setminus K_j) \cap B_F \)), \( y_j(v') = 0 \) (resp. \( y_j(v') \leq |\vec{y}(b_j)| \)).

**Proof.** Step 2(ii) only increases the magnitude of dual weights or leaves them unchanged. This follows from the fact that for a vertex \( v \in B_H \), \( \vec{y}(v) \) increases by \( c(\alpha) - c(v) \) only if \( c(\alpha) - c(v) \geq 0 \). Similarly, for a vertex \( v \in A_H \), \( \vec{y}(v) \) decreases by \( c(\alpha) - c(v) \) only if \( c(\alpha) - c(v) \geq 0 \). Since we assume that the matching at the beginning of the phase is planar-feasible, any \( \vec{y}(v) \) for \( v \in B_H \) is non-negative and any \( \vec{y}(v) \) for \( v \in A_H \) is non-positive. Therefore, the magnitudes of \( \vec{y}(\cdot) \) only increase during Step 2(ii), and the claims hold after step Step 2(ii).
Step (a) of the \texttt{Augment} procedure calls \texttt{Sync} on the affected partitions. The properties of \texttt{Sync} ensure that the claims are satisfied after step (a). Finally, in step (c) of \texttt{Augment}, for some vertices \( v \in A \), we lower the value of \( y_j(v) \), increasing its magnitude. However, if \( v \in K_j \) then the magnitude of \( \tilde{y}(v) \) increases by the same amount as \( y_j(v) \). There are no free vertices \( v \in A_F \) along the path after augmentation. Therefore, the claims hold true at the end of a phase.

The following lemma guarantees that planar feasibility conditions (a) and (b) are satisfied.

\textbf{Lemma 5.19.} Assuming we are given a planar-feasible matching at the beginning of a phase, then at the end of that phase, every vertex \( v \in A_H \) has a non-positive dual weight \( \tilde{y}(v) \) and for each free (internal or boundary) vertex \( v \) of \( A_H \), \( \tilde{y}(v) = 0 \). Every vertex \( v \in B_H \) has a non-negative dual weight \( \tilde{y}(v) \) and for each free (internal or boundary) vertex \( v \) of \( B_H \), \( \tilde{y}(v) = y_{\max} \), where \( y_{\max} = \max_{v \in A_H \cup B_H} \tilde{y}(v) \).

\textbf{Proof.} As discussed in the proof of Lemma 5.18, the operations we apply do not decrease the magnitude of \( \tilde{y}(\cdot) \). That is, for every \( v \in B_H \), \( \tilde{y}(v) \) only increases, and for every \( v \in A_H \), \( \tilde{y}(v) \) only decreases. It remains to prove the properties of \( \tilde{y}(\cdot) \) for free vertices. For any vertex \( v \in A_H^F \), Step 2(ii) does not change \( \tilde{y}(v) \). This follows from the fact that if \( v \in A_H^F \) and we have \( c_v > c_\alpha \) then \( \tilde{y}(v) \) is unchanged. Otherwise the magnitude of \( \tilde{y}(v) \) increases by \( c_\alpha - c_v \), which is 0 since \( \alpha \) is a vertex in \( A_H^F \) with minimum \( c(\cdot) \). For any vertex \( v \in B_H^F \), \( c_v = 0 \). All distance values \( c \) are non-negative, so \( c_\alpha - c_v \) is maximum over all vertices of \( V_H \). The dual weights of all free vertices of \( B_H \) increase by the same amount, and continue being the largest among all \( \tilde{y}(\cdot) \). Therefore, the claim holds true after Step 2(ii). The \texttt{Augment} procedure reduces the dual weights of vertices in \( P \cap A_H \) after augmenting along \( P \). However, \( P \) has no free vertices after augmentation. Therefore, at the end of a phase, every \( v \in A_F \), \( \tilde{y}(v) \) remains 0 and for every \( v \in B_F \), \( \tilde{y}(v) \) remains maximum over all \( \tilde{y}(\cdot) \). \qed
Combining lemmas (5.16), (5.17), (5.18), and (5.19) yields the following lemma.

**Lemma 5.20.** The matching at the end of each phase of the algorithm is a planar feasible matching.

Lemma 5.20 guarantees that at the end of each scale, we have a planar-feasible matching. We can convert a planar-feasible matching to an \( \mathcal{R} \)-feasible matching in time \( O(n \log r) \) by call the `Sync` procedure on every partition.

### 5.4.2 Efficiency

Step 1 of our algorithm for each scale requires time \( O(n \sqrt{r}) \) since it executes \( O(\sqrt{r}) \) iterations of the Gabow-Tarjan algorithm, with each iteration taking \( O(n) \). This results in \( O(n/\sqrt{r}) \) free vertices remaining.

In Step 2, our algorithm increases the size of the matching by 1 each iteration, and therefore executes \( O(n/\sqrt{r}) \) iterations. Step 2(i) takes \( O((n/\sqrt{r}) \log^2 r) \) time. Step 2(ii) takes \( O(n/\sqrt{r}) \) time since \( \tilde{g}(v) \) could be changed for each \( v \in V_H \). In Step 2(iv) we recompute all of the interval-min search data structures, taking \( O((n/\sqrt{r}) \log^2 r) \) time. Therefore, over \( O(n/\sqrt{r}) \) iterations, the total time for Step 2(i), Step 2(ii), Step 2(iv) is \( O((n^2/r) \log^2 r) \). It remains to bound the time taken by Step 2(iii), i.e., the `Augment` procedure.

Let \( P_i \) be the path in \( H \) computed in Step 2(i) during the \( i^{th} \) phase (i.e., the phase of the second step of the algorithm that produces the \( i^{th} \) augmenting path). During `Augment`, `Sync` is called on all affected partitions. This takes \( O(r \log r) \) time per partition. Next, \( P \) is projected to yield an augmenting path \( \overrightarrow{P} \) in \( G_M \). This requires a BFS search for each projected edge of \( P \), each search taking \( O(r) \) time. Then, `Augment` procedure augments the edges of \( \overrightarrow{P} \) and adjusts dual weights along \( \overrightarrow{P} \). This takes time proportional to \( |\overrightarrow{P}| \), which
is at most \(O(r)\) for each affected partition. Finally, we call \textsc{Construct} on all the affected partitions, taking time \(O(r \log r)\) per partition. Thus, the total time taken by the \textsc{Augment} procedure is \(O(\Delta r \log r)\) where \(\Delta = \sum_{i=1}^{O(n/\sqrt{r})} |P_i|\) is the total length of all augmenting paths found in \(H\) by Step 2(i) during the entire second step. Lemma 5.22 bounds the total length of all paths in \(H\), i.e., \(\Delta = O((n/\sqrt{r}) \log n)\). Therefore, over all the \(O(n/\sqrt{r})\) phases, the total time taken by the \textsc{Augment} procedure is \(O(n \sqrt{r} \log r \log n)\).

Setting \(r = n^{2/3}\) yields a total complexity of \(O(n^{4/3} \log^2 n)\) for each scale. Our algorithm executes \(O(\log nC)\) scales, where \(C\) is the largest edge cost of the original graph. Therefore, the total complexity is \(O(n^{4/3} \log^2 n \log nC)\).

The following lemma shows that we compute the minimum net-cost augmenting path in \(G\), which is useful for bounding \(\Delta\).

**Lemma 5.21.** The augmenting path computed in each phase is a minimum net-cost augmenting path.

**Proof.** Let \(\overrightarrow{P}\) be the augmenting path found in \(G_M\). By Lemma 5.14, any edge \((u, v) \in \overrightarrow{P} \cap E_j\) is admissible with respect to the matching \(M_j\) and the dual weights \(y_j(\cdot)\). Since the \textsc{Sync} procedure has already been executed for partitions in \(A\), we can invoke the \textsc{Sync} procedure on the remaining partitions that are not in \(A\) in order to obtain an \(R\)-feasible matching \(M, y(\cdot)\) on \(G\) by Lemma 5.11. The dual weights \(y_j(\cdot)\) for all \(R_j \in A\) are unchanged, and therefore, the edges of \(\overrightarrow{P}\) would remain admissible with respect to the matching \(M, y(\cdot)\). This means the total slack along \(\overrightarrow{P}\) is 0 with respect to \(M\). Since the matching \(M\) is feasible, all edges must have non-negative slack, and \(\overrightarrow{P}\) is a minimum slack directed path in \(G'_M\). By Lemma 5.5 we have for any path from \(u\) to \(v\) in \(G_M\),

\[
\phi(\overrightarrow{P}_{u,v}) - |y(u)| + |y(v)| = \sum_{(a,b) \in \overrightarrow{P}_{u,v}} s(a,b).
\]
We maintain that for a vertex \( v \in B_F \), \( \bar{y}(v) = y_{\text{max}} \) and for a vertex \( v \in A_F \), \( \bar{y}(v) = 0 \).

Therefore, by the properties of the \textsc{Sync} procedure, for any vertex \( v \in B_F \), \( y(v) = y_{\text{max}} \), and for any vertex \( v \in A_F \), \( y(v) = 0 \). Since all free vertices of \( B \) (resp. \( A \)) have the same dual weight, a zero slack augmenting path with respect to \( M \) and \( y(\cdot) \) is also an augmenting path of minimum net-cost.

The following lemma bounds the total length of all the augmenting paths computed by our algorithm. Gabow and Tarjan assign a error of 1 on every non matching edge and show that the total error on all the augmenting paths generated by their algorithm is \( O(n \log n) \). We use a similar proof strategy and show that the total error of all augmenting paths generated by our algorithm is also \( O(n \log n) \). However, in our case, we assigned a cost of \( \lceil \sqrt{r} \rceil \) for every edge incident on a boundary vertex. So, the number of such edges and therefore, the number of boundary vertices on the augmenting paths can be no more than \( O(n \sqrt{r} \log n) \) leading to a bound on their lengths in \( H \).

**Lemma 5.22.** Let \( P_i \) be the \( i \)th path in \( H \) computed by the second step of our algorithm, and let \( M_i \) be the matching just before computing \( P_i \). Let \( t \) be the number of phases of the second step of our algorithm. Recall \( t = O(n/\sqrt{r}) \). Then, \( \Delta = \sum_{i=1}^{t} |P_i| = O((n/\sqrt{r}) \log n) \)

**Proof.** We first show that
\[
\sum_{i=1}^{t} \phi(P_i) = O(n \log n). \tag{5.13}
\]

Consider \( M_i \oplus M_{\text{Opt}} \) where \( M_{\text{Opt}} \) is some minimum cost perfect matching. \( M_i \oplus M_{\text{Opt}} \) forms alternating paths and cycles. Let the set \( S_i \) consist of all the augmenting paths in \( M_i \oplus M_{\text{Opt}} \).

Then \(|S_i| = n - i + 1\). We will now show that
\[
\sum_{P \in S_i} \phi(P) \leq (2k + 3)n. \tag{5.14}
\]
By definition of net cost, given that all edge costs are positive, we have:

\[ \sum_{P \in S_i} \phi(P) \leq \sum_{P \in S_i} \sum_{(u,v) \in M_{\text{Opt}} \cap P} (c(u,v) + \delta_{uv}). \]

Since all edge costs are positive, the edges in \( M_{\text{Opt}} \) that are also in some \( P \in S_i \) must have a total cost at most \( c(M_{\text{Opt}}) \). Thus

\[ \sum_{P \in S_i} \phi(P) \leq \left( \sum_{P \in S_i} \sum_{(u,v) \in M_{\text{Opt}} \cap P} \delta_{uv} \right) + (k + 2)n. \quad (5.15) \]

By Lemma 5.2, we have

\[ \sum_{(u,v) \in M_{\text{Opt}}} \delta_{uv} \leq (k + 1)n, \]

\[ \sum_{P \in S} \sum_{(u,v) \in M_{\text{Opt}} \cap P} \delta_{uv} \leq (k + 1)n. \quad (5.16) \]

Combining (5.15) and (5.16) yields

\[ \sum_{P \in S_i} \phi(P) \leq (2k + 3)n. \]

Since \( |S_i| = n - i + 1 \), we can say that the average cost length path in \( S_i \) is at most \( (2k + 3)n/(n - i + 1) \). Our algorithm finds the minimum net-cost path available, by Lemma 5.21, which is at most the average net cost path in \( S \). Therefore, we have

\[ \sum_{i=1}^{t} \phi(P_i) \leq \sum_{i=1}^{t} \left( (2k + 3)n/(n - i + 1) \right). \]
5.4. Analysis of the Algorithm

The denominator of the right side forms a harmonic series. Therefore,

\[
\sum_{i=1}^{t} \phi(P_i) = O(n \log n).
\]

Let \( M_{\text{START}} \) be the matching computed after the first step of the algorithm and let \( M \) be the final perfect matching computed by the second step of our algorithm. By definition of the net-cost, from Lemma 5.2 and the fact that \( c(M) \geq 0 \), we have that

\[
\sum_{i=1}^{t} \phi(P_i) = c(M) - c(M_{\text{START}}) + \sum_{P_i} \sum_{(u,v) \in P_i \setminus M_i} \delta_{uv} \\
\geq -c(M_{\text{START}}) + \sum_{i=1}^{t} \sum_{(u,v) \in P_i \setminus M_i} \delta_{uv}.
\]

\( M_{\text{START}} \) is guaranteed to have a cost less than \( c(M_{\text{OPT}}) + n = O(n) \) by the properties of the Gabow-Tarjan algorithm. Therefore,

\[
\sum_{i=1}^{t} \sum_{(u,v) \in P_i \setminus M_i} \delta_{uv} = O(n \log n).
\]

For any \((u, v) \not\in M_i\) where \( u \in \mathcal{X}_j \) or \( v \in \mathcal{X}_j \), we have that \( \delta_{uv} = \lceil \sqrt{r} \rceil \). This means that among all augmenting paths computed in \( G \), there can only be \( O((n/\sqrt{r}) \log n) \) boundary vertices used. The length of any path in \( H \) is at most the number of boundary vertices along the path plus 1, and there are \( t = O(n/\sqrt{r}) \) augmenting paths computed during the second step of the algorithm. Therefore,

\[
\sum_{i=1}^{t} |P_i| = O((n/\sqrt{r}) \log n).
\]
Chapter 6

Minimum-Cost Matching on
Minor-Free Graphs

We give an $\tilde{O}(n^{7/5} \log n C)$-time\footnote{We use $\tilde{O}(\cdot)$ to suppress logarithmic terms throughout the paper.} algorithm to compute a minimum-cost maximum cardinality matching (optimal matching) in bipartite $K_h$-minor free graphs with $h = O(1)$ and integer edge weights having magnitude at most $C$. This improves upon the $\tilde{O}(n^{10/7} \log C)$ algorithm of Cohen et al. [SODA 2017] and the $O(n^{3/2} \log n C)$ algorithm of Gabow and Tarjan [SIAM J. Comput. 1989].

For a graph with $m$ edges and $n$ vertices, the well-known Hungarian Algorithm computes a shortest augmenting path in each phase in $O(m)$ time, yielding an optimal matching in $O(mn)$ time. The Hopcroft-Karp [SIAM J. Comput. 1973], and Gabow-Tarjan [SIAM J. Comput. 1989] algorithms compute, in each phase, a maximal set of vertex-disjoint shortest augmenting paths (for appropriately defined costs) in $O(m)$ time. This reduces the number of phases from $n$ to $O(\sqrt{n})$ and the total execution time to $O(m\sqrt{n})$.

In order to obtain our speed-up, we relax the conditions on the augmenting paths and iteratively compute, in each phase, a set of carefully selected augmenting paths that are not restricted to be shortest or vertex-disjoint. As a result, our algorithm computes substantially more augmenting paths in each phase, reducing the number of phases from $O(\sqrt{n})$ to $O(n^{2/5})$. By using small vertex separators, the execution of each phase takes $\tilde{O}(m)$ time on average.
For planar graphs, we combine our algorithm with efficient shortest path data structures to obtain a minimum-cost perfect matching in $\tilde{O}(n^{6/5} \log(nC))$ time. This improves upon the recent $\tilde{O}(n^{4/3} \log(nC))$ time algorithm by Asathulla et al. [SODA 2018].

**Organization** The remainder of the paper is organized as follows. In Section 6.1 we present a new variant of Gabow and Tarjan’s algorithm that our search procedure is based on. In Section 6.2 we define the notion of an $R$-feasible matching based on an $r$-clustering and related concepts. In Section 6.3, we describe our scaling algorithm. In section 6.4, we present our algorithm for each scale. In Section 6.5 we prove the correctness and efficiency of our algorithm. In Section 6.6, we combine our algorithm with shortest path data structures designed for planar graphs to achieve an $O(n^{6/5})$ time algorithm. Some claims whose proofs are similar to those presented in [6] are presented in the appendix.

### 6.1 Alternative Interpretation of the GT Algorithm

In this section, we present a modified version of the original Gabow-Tarjan algorithm discussed in Section 3.4. We begin by giving an overview of the algorithm and describe the steps it takes between two successive scales. After that, we present the algorithm inside each scale. The steps taken by this algorithm are different from Gabow and Tarjan’s original algorithm and its correctness requires a proof. Instead of providing a proof of correctness here, we adapt this algorithm for our setting and provide a proof for our case directly.

As in the Hungarian algorithm, the Gabow-Tarjan algorithm also maintains a dual weight for every vertex of $G$. We define a 1-feasible matching to consist of a matching $M$ and set of dual weights $y(v)$ such that for every edge between $u \in A$ and $v \in B$ we have
Chapter 6. Minimum-Cost Matching on Minor-Free Graphs

\[ y(u) + y(v) \leq c(u, v) + 1, \]  
\[ y(u) + y(v) \geq c(u, v) - 1 \quad \text{for } (u, v) \in M. \]  

(6.1)  
(6.2)

Gabow and Tarjan presented a similar feasibility constraints for the minimum-cost degree constrained subgraph (DCS) of a bipartite multigraph [18]. Note, however, that our definition of 1-feasibility is different from the original definition of 1-feasibility as given by Gabow and Tarjan:

\[ y(u) + y(v) \leq c(u, v) + 1, \]  
\[ y(u) + y(v) = c(u, v) \quad \text{for } (u, v) \in M. \]  

(6.3)  
(6.4)

A 1-optimal matching is a perfect matching that is 1-feasible. We define the slack of an edge to be \( c(u, v) - y(u) - y(v) + 1 \) if \( (u, v) \not\in M \) and \( y(u) + y(v) - c(u, v) + 1 \) if \( (u, v) \in M \). The following lemma relates 1-optimal matchings to the optimal matchings.

Lemma 6.1. For a bipartite graph \( G(A \cup B, E) \) with an integer edge cost function \( c \), let \( M \) be a 1-optimal matching and \( M_{\text{Opt}} \) be the optimal matching. Then, \( c(M) \leq c(M_{\text{Opt}}) + 2n \).

For every \( (a, b) \in E \), suppose we redefine the edge weight to be \( c^*(a, b) = (2n + 1)c(a, b) \). This uniform scaling of edge costs preserves the set of optimal matchings and guarantees that any sub-optimal matching has a cost that is at least \( 2n + 1 \) greater than the optimal cost. Thus, a 1-optimal matching with the edge weights \( c^*(\cdot, \cdot) \) corresponds to an optimal matching with the original edge weights \( c(\cdot, \cdot) \).

We now describe the bit-scaling paradigm. For any edge \( (u, v) \), let \( b_1, b_2, \ldots, b_\ell \) be the binary representation of \( c^*(u, v) \). Let \( c_i(u, v) \) correspond to the most significant \( i \) bits of \( c^*(u, v) \).
The Gabow-Tarjan Algorithm consists of scales. The algorithm for any scale \( i \) takes a bipartite graph on \( A, B \), with a cost function \( c_i(\cdot, \cdot) \), and a set of dual weights \( y(v) \) for every vertex \( v \in A \cup B \) as input. Let \( c_i(u, v) = c_i(u, v) - y(u) - y(v) \). Then, \( c_i(\cdot, \cdot) \) satisfies the following at the beginning of the \( i \)th scale:

- For every edge \((u, v)\), \( c_i(u, v) \geq 1 \), and,
- The cost of a 1-optimal matching with respect to \( c_i(\cdot, \cdot) \) is \( O(n) \).

Given such an input, the algorithm for each scale returns a perfect matching \( M \) and a set of dual weights \( y(\cdot) \) so that \( M, y(\cdot) \) is a 1-optimal matching. The input to the first scale is the graph \( G(A \cup B, E) \) with the cost \( c_1(\cdot, \cdot) \) and a set of dual weights of \(-1\) on every vertex of \( B \) and dual weights of \( 0 \) on every vertex of \( A \). It is easy to see that \( c_1(\cdot, \cdot) \) satisfies the two conditions. For any scale \( i \), the algorithm computes a matching \( M \) and dual weights \( y'(\cdot) \) so that \( M, y'(\cdot) \) is a 1-optimal matching with respect to the costs \( c^i(\cdot) \). For every vertex \( v \in A \cup B \), let \( y(v) \) be the sum of the dual weight \( y'(v) \) and the initial dual weight assigned at the start of the scale \( i \) to \( v \). Then, it can be shown that \( M, y(\cdot) \) is 1-optimal with respect to \( c_i(\cdot, \cdot) \).

For any \( i \geq 1 \), we use the 1-optimal matching \( M, y(\cdot) \) returned by the algorithm for scale \( i \) to generate an input for scale \( i + 1 \) as follows. First, we set the slack (with respect to \( c_i(\cdot, \cdot) \)) of every edge of the 1-optimal matching \( M \) of scale \( i \) to 0. For any edge \((u, v) \in M \), this can be done by reducing the dual weight of one of its vertices, say \( u \) by \( s(u, v) \), i.e., \( y(u) \leftarrow y(u) - s(u, v) \). Note that any reduction in dual weight of \( u \) does not violate (6.1) or (6.2) and so \( M, y(\cdot) \) remains 1-optimal. After this, we transfer the dual weights from scale \( i \) to scale \( i + 1 \) by simply setting, for any vertex \( v \in A \cup B \), \( y(v) \leftarrow 2y(v) - 2 \). Therefore, at the beginning of scale \( i + 1 \), the reduced cost \( c^{i+1}(u, v) = c_{i+1}(u, v) - y(u) - y(v) \) on every edge is at least 2 and every edge \((u, v)\) in the 1-optimal matching \( M \) of scale \( i \) has
c^{i+1}(u, v) \leq 6. So, the cost of an optimal matching and also any 1-optimal matching with respect to the c^{i+1}(\cdot, \cdot) is upper bounded by O(n) as desired.

**Algorithm for Each Scale**  Now, we present the algorithm for scale $i$. We refer to an edge $(u, v)$ as *admissible* if it has a slack of 0, i.e.,

- $y(u) + y(v) = c^{i+1}(u, v) + 1$ if $(u, v) \not\in M$,
- $y(u) + y(v) = c^{i+1}(u, v) - 1$ if $(u, v) \in M$.

An *admissible graph* is the set of admissible edges. The algorithm runs in two stages. The first stage of the algorithm executes $O(\sqrt{n})$ iterations. In each iteration, the algorithm initiates a DFS search from each free vertex in $B_F$. If it finds an augmenting path in the admissible graph, then it augments the matching right away. Consider a DFS initiated from $b \in B_F$. Let $P = \langle u_1, ..., u_k \rangle$ be the current path of the DFS with $u_1 = b$.

- If there is no admissible edges outgoing from $u_k$, then remove $u_k$ from $P$. If $u_k \in B$, set $y(u_k) \leftarrow y(u_k) + 1$. Otherwise, $u_k \in A$, and set $y(u_k) \leftarrow y(u_k) - 1$.
- Otherwise, suppose there is an admissible edge from $u_k$ to a vertex $v$. If $v$ is a free vertex of $A$, then the algorithm has found an augmenting path of admissible edges and it augments the matching along the path. Otherwise, it adds $v$ to the path as vertex $u_{k+1}$ and continues the search from $v$.

This completes the description of a DFS search for augmenting paths.

In a single iteration, a DFS is initiated from every free vertex of $B_F$. At the end of the iteration, it can be shown that there are no augmenting paths consisting of only admissible edges and the dual weights of every free vertex $b \in B_F$ are increased by exactly 1 when $b$ is
6.1. Alternative Interpretation of the GT Algorithm

removed from \( P \). After \( \sqrt{n} \) iterations, the dual weights of vertices in \( B_F \) will be \( \sqrt{n} \). It can be shown that, the sum of dual weights of \( A_F \) and \( B_F \) cannot exceed the cost of a 1-optimal matching, i.e., \( O(n) \). Furthermore, the dual weights of vertices of \( A_F \) are maintained as 0. So,

$$
\sum_{v \in B_F \cup A_F} y(v) \geq |B_F| \sqrt{n} = O(n).
$$

This bounds \( |B_F| \) by \( O(\sqrt{n}) \). After this, we iteratively (for \( O(\sqrt{n}) \) iterations) execute Hungarian Search in \( O(m) \) time to find an augmenting path and augment the matching. The total computation time for a single scale is \( O(m \sqrt{n}) \), and summed over all \( O(\log nC) \) scales, total time taken is \( O(m \sqrt{n} \log nC) \). Gabow and Tarjan show that the total length of all the augmenting paths found is \( O(n \log n) \), and a similar argument can be applied here.

There are two aspects in which this description of Gabow and Tarjan’s algorithm differs from the original GT-Algorithm. First, in the first \( \sqrt{n} \) phases, we avoid doing a Hungarian Search and only conduct several partial depth-first searches. Second, for any partial DFS, all vertices of \( B \) (resp. \( A \)) that are visited by the DFS but do not lead to an augmenting path undergo an increase (resp. decrease) in their dual weight. The dual weights of \( B \) start at 0 and only increase during the algorithm. Therefore, dual weights of \( B \) are non-negative. Similarly, the dual weights of \( A \) start at 0 and may reduce during the algorithm. So, dual weights of vertices in \( A \) are non-positive. So, if the partial DFS visits any vertex, except for vertices that are along any augmenting path \( P \), there will be an increase in the magnitude of its dual weight by 1. Whenever a vertex is visited, perhaps each of its \( \deg(v) \) neighbors could be explored. By careful analysis, the time taken to execute the first \( \sqrt{n} \) phases can also be bounded by \( O(\sum_{v \in A \cup B} \deg(v)|y(v)| + \sum_{j=1}^{n} |P_j|) = O(m \sqrt{n} + n \log n) \), where \( P_j \) is the \( j^{th} \) augmenting path computed by the algorithm.
In the next section, we introduce definitions for feasibility, admissibility and net-cost as used in our algorithm.

6.2 Preliminaries

In this section, we introduce conditions for feasibility, admissibility and the definitions of slack and net-cost as used in our algorithm. Using these definitions, we establish critical properties (Lemma 6.3 and 6.4) of paths in an admissible graph. These definitions and properties are based on an $r$-clustering which we formally introduce next.

An $r$-Clustering of a Graph $G$ Consider a partitioning of any graph $G(V, E)$ into $l$ edge-disjoint subgraphs called pieces and denoted by $\mathcal{R}(G) = \{\mathcal{R}_1(V_1, E_1), \ldots, \mathcal{R}_l(V_1, E_l)\}$. For each piece $\mathcal{R}_j$, the vertex set $V_j$ and the edge set $E_j$ of any piece $\mathcal{R}_j$ is the set $E_j = \{(a, b) \mid a, b \in V_j\}$. Furthermore, $\bigcup_{j=1}^l V_j = V$ and $E = \bigcup_{j=1}^l E_j = E$. A vertex $v \in V$ which has edges incident from two or more pieces is called a boundary vertex. An edge which is adjacent to one or more boundary vertices is a boundary edge. Any other edge is an interior edge. Let $K_j$ denote the set of boundary vertices of $\mathcal{R}_j$ and let $K = \bigcup_{j=1}^l K_j$.

Definition 6.2. A partition $\mathcal{R}(G) = \{\mathcal{R}_1(V_1, E_1), \ldots, \mathcal{R}_l(V_1, E_l)\}$ of a graph $G$ is an $r$-clustering if $l = O(n/\sqrt{r})$, for each $\mathcal{R}_j$, $|V_j| = O(r)$ and $|K_j| = O(\sqrt{r})$. Furthermore, the total number of boundary vertices is $|K| \leq \sum_j |K_j| = O(n/\sqrt{r})$. Let $k_1, k_2$ be constants such that $\max_j |K_j| \leq k_1 \sqrt{r}$ and $|K| \leq k_2 n/\sqrt{r}$. Also, let $|V_j| = n_j$ and $|E_j| = m_j$.

For any choice of $r$, an $r$-clustering can be computed on $K_r$-minor free graphs in $O(m \log n + n^{1+\epsilon} \sqrt{r})$ time [59]. Note that the $r$-clustering computed in [59] has $\tilde{O}(n/\sqrt{r})$ pieces, $\tilde{O}(n/\sqrt{r})$ boundary nodes (counting multiplicities), and $\tilde{O}(\sqrt{r})$ boundary nodes per piece. For simplicity in exposition, we present for the case without the poly(log $n$) terms. After presenting
our algorithm, we briefly describe in Section 6.5.3, the necessary changes to account for these poly(log n) terms.

Convention for Notation  Throughout this paper, we will deal with a bipartite graph $G$. For any vertex $u \in A \cup B$, throughout this paper we use $\lambda_u = -1$ if $u \in A$ and $\lambda_u = 1$ if $u \in B$. For simplicity of analysis, we assume without loss of generality that $\sqrt{r}$ is an integer. Given a matching $M$ and a set of dual weights $y(\cdot)$, we refer to its residual graph by $G_M$. Note that the vertex and edge sets of $G$ and $G_M$ are identical (except for the directions) and a matching, alternating path or an alternating cycle in $G$ is also a matching, directed path or a directed cycle in $G_M$. So, if there is any subset $P$ of edges in $G$, we will also use $P$ to denote the same subset of edges in $G_M$, the directions of these edges are determined by whether or not an edge is in $M$. We will define a net-cost for an alternating path (or cycle) $P$ in our algorithm and denote it by $\phi(P)$. Any directed path or cycle in $G_M$ will inherit its net-cost from $G$. During the course of our algorithm, for any weighted and directed graph $K$, we will use the notation $K'$ to be the graph identical to $K$ where the cost of any directed edge in the graph is replaced by its slack. Recall that an $r$-clustering $\mathcal{R}(G)$ partitions the edges of $G$. Since $G$, $G_M$ and $G'_M$ have the same underlying set of edges, $\mathcal{R}(G)$ can be seen as an $r$-clustering of $G_M$ and $G'_M$ as well.

We next introduce a notion of feasibility that is based on an $r$-clustering. We assume that we are given an $r$-clustering, $\mathcal{R}(G) = \{\mathcal{R}_1(V_1, E_1), \ldots, \mathcal{R}_l(V_l, E_l)\}$ with ($l = O(n/\sqrt{r})$). Recall that we denote by $\mathcal{K}$ the boundary vertices of $\mathcal{R}$ and by $\mathcal{K}_j$ the set of boundary vertices in $\mathcal{R}_j$. For every edge $uv \in E_j$, we define a 0/1 indicator variable $i_{uv}$ to be 1 iff $uv$ is a boundary edge in $\mathcal{R}$. We define a value $\delta_{uv}$ to be $\max\{1, i_{uv} \frac{m n}{\sqrt{r}}, 2i_{uv} \sqrt{r}\}$. For any edge induced subgraph $G^*(V^*, E^*)$ of $G(A \cup B, E)$, we say that a matching $M \subseteq E^*$ and a set of dual weights $y(\cdot)$ on vertices of $V^*$ are $\mathcal{R}$-feasible if every edge $(u, v) \in E^*$ satisfies the
following two conditions:

\[
\begin{align*}
y(u) + y(v) &\leq c(u, v) + \delta_{uv} \quad \text{for } (u, v) \notin M, \quad (6.5) \\
y(u) + y(v) &\geq c(u, v) - \delta_{uv} \quad \text{for } (u, v) \in M. \quad (6.6)
\end{align*}
\]

The algorithm in [6] used a very similar definition except that it uses a different $\delta_{uv}$. Additionally, here we allow matching edges to violate the traditional feasibility constraints. We can define the slack of an edge $(u, v)$ denoted by $s(u, v)$ with respect to the dual weights as

\[
\begin{align*}
s(u, v) &= c(u, v) + \delta_{uv} - y(u) - y(v) \quad \text{for } (u, v) \notin M, \quad (6.7) \\
s(u, v) &= y(u) + y(v) - c(u, v) + \delta_{uv} \quad \text{for } (u, v) \in M. \quad (6.8)
\end{align*}
\]

We define admissible edges next. Any boundary edge $(u, v)$ is admissible only if $s(u, v) \leq \sqrt{r}$. Any edge $(u, v)$ that does not border a boundary vertex is admissible only if $s(u, v) = 0$. Note that in both cases an admissible edge satisfies

\[\delta_{uv} \geq 2s(u, v).\]

Our algorithm will compute admissible paths and cycles with respect to $R$-feasible matchings. For any such path $P$, we update the matching $M$ by setting $M \leftarrow M \oplus P$. The following lemma shows that the new matching after such an operation remains $R$-feasible.

**Lemma 6.3.** Given an $R$-feasible matching $M, y(\cdot)$ on any bipartite graph $G(V, E)$, let $\vec{P}$ be a path or cycle in $G_M$ consisting of only admissible edges. Then, $M' \leftarrow M \oplus \vec{P}, y(\cdot)$ is also an $R$-feasible matching. Furthermore, the slack on every edge $(u, v)$ of $\vec{P}$ with respect to $M', y(\cdot)$ is at least $\delta_{uv}$. 
Proof. First, consider any edge \((u, v) \in P \cap M\). Suppose \((u, v)\) is not a boundary edge, so \(\delta_{uv} = 1\). Then the slack \(s(u, v)\) is zero and

\[
\begin{align*}
y(u) + y(v) - c(u, v) + \delta_{uv} &= 0, \\
y(u) + y(v) &= c(u, v) - \delta_{uv} < c(u, v) + \delta_{uv}.
\end{align*}
\]

Thus, \((u, v)\) is \(\mathcal{R}\)-feasible with respect to \(M'\) and the slack on the edge with respect to \(M'\) is at least \(\delta_{uv}\). Otherwise, \((u, v)\) is a boundary edge, and \(\delta_{uv} \geq 2\sqrt{r}\). Since \((u, v)\) is an admissible edge in the matching \(M\), \((u, v)\) will satisfy

\[
\begin{align*}
s(u, v) &= y(u) + y(v) - c(u, v) + \delta_{uv} \leq \sqrt{r}, \\
y(u) + y(v) &\leq c(u, v) - \delta_{uv} + \sqrt{r} < c(u, v) + \delta_{uv}.
\end{align*}
\]

So, feasibility condition (6.5) holds for \((u, v)\) with respect to \(M'\) and the slack of \((u, v)\) is at least \(\delta_{uv}\).

Next, consider \((u, v) \in P \setminus M\). Suppose \((u, v)\) is not a boundary edge, so \(\delta_{uv} = 1\). Then we have

\[
\begin{align*}
c(u, v) + \delta_{uv} - y(u) - y(v) &= 0, \\
y(u) + y(v) &= c(u, v) + \delta_{uv} > c(u, v) - \delta_{uv}.
\end{align*}
\]

So, feasibility condition (6.6) holds for \((u, v)\) with respect to the matching \(M'\) and dual weights \(y(\cdot)\) and the slack of \((u, v)\) is at least \(\delta_{uv}\). Otherwise, \((u, v)\) is a boundary edge, and
\[ \delta_{uv} \geq 2\sqrt{r}. \] Then the admissible edge \((u, v)\) will satisfy

\[
\begin{align*}
c(u, v) + \delta_{uv} - y(u) - y(v) & \leq \sqrt{r}, \\
y(u) + y(v) & \geq c(u, v) + \delta_{uv} - \sqrt{r} > c(u, v) - \delta_{uv}.
\end{align*}
\]

So, feasibility condition (6.6) is satisfied with respect to the matching \(M'\) and dual weights \(y(\cdot)\) and the slack of \((u, v)\) is at least \(\delta_{uv}\). Since all edges of \(G\) satisfy the \(R\)-feasibility conditions (6.5) and (6.6), the matching \(M'\) and \(y(\cdot)\) is \(R\)-feasible. \(\square\)

An \(R\)-optimal matching is a perfect matching that is \(R\)-feasible. Our algorithm, for the graph \(G(A \cup B, E)\) and the \(r\)-clustering \(R(G)\), computes an \(R\)-optimal matching \(M\) along with its dual weights \(y(\cdot)\). Note that the notion of \(R\)-feasible matching can be defined for any edge induced subgraph of \(G\). In the context of this paper, the only induced subgraphs that we consider are pieces from the set \(\{R_1, \ldots, R_l\}\) that are given by the \(r\)-clustering of the graph. Our algorithm will maintain an \(R\)-feasible matching for each piece. Throughout this paper, we fix the \(r\)-clustering in the definition of \(R\)-feasibility to be \(R(G)\). For any \(R\)-feasible matching, when obvious from the context, we will not explicitly mention the induced subgraph the matching is defined on.

For our algorithm, we will define the net-cost of an edge \((u, v)\), \(\phi(u, v)\) as

\[
\begin{align*}
\phi(u, v) &= c(u, v) + \delta_{uv} \quad \text{for} \ (u, v) \notin M, \\
\phi(u, v) &= -c(u, v) + \delta_{uv} \quad \text{for} \ (u, v) \in M.
\end{align*}
\] (6.13) (6.14)

For any set of edges \(S\), we can define the net cost as

\[ \phi(S) = \sum_{(u, v) \in S} \phi(u, v). \]
Our interest is in net costs for the case where $S$ is an augmenting path, alternating path, or alternating cycle. Consider if we have any matching $M$ and let $M' \leftarrow M \oplus S$. Then

$$\phi(S) = c(M') - c(M) + \sum_{(u,v) \in S} \delta_{uv}. \quad (6.15)$$

From the definitions of net-cost and slack, we get the following relation:

$$s(a, b) = \phi(a, b) + y(a) + y(b) \quad \text{if} \ (a, b) \in M, \quad (6.16)$$

$$s(a, b) = \phi(a, b) - y(a) - y(b) \quad \text{if} \ (a, b) \notin M. \quad (6.17)$$

For any vertex $u \in A \cup B$, throughout this paper we use $\lambda_u = -1$ if $u \in A$ and $\lambda_u = 1$ if $u \in B$. Given any directed path (resp. cycle) $\overrightarrow{P}$ from a vertex $u$ to a vertex $v$ in $G_M$ if we add the above equations over all the edges of $\overrightarrow{P}$, we get

$$\sum_{(a, b) \in \overrightarrow{P}} s(a, b) = \sum_{(a, b) \in \overrightarrow{P}} \phi(a, b) + \lambda_v y(v) - \lambda_u y(u) = \phi(\overrightarrow{P}) + \lambda_v y(v) - \lambda_u y(u). \quad (6.18)$$

Despite allowing for slacks in the admissible edges, the following lemma shows that for any admissible path the difference in the dual weights of its first and last vertex is related to the change in the matching cost and the number of pieces visited by this path.

**Lemma 6.4.** Given an $\mathcal{R}$-feasible matching $M, y(v)$, suppose we have a simple alternating path or simple alternating cycle $\overrightarrow{P}$ from $u$ to $v$ ($u = v$ if $\overrightarrow{P}$ is a cycle) consisting only of admissible edges. Then,

$$\lambda_u y(u) - \lambda_v y(v) \geq c(M \oplus \overrightarrow{P}) - c(M) + \sum_{(p,q) \in \overrightarrow{P}} \frac{\delta_{pq}}{2}. \quad (6.19)$$
Chapter 6. Minimum-Cost Matching on Minor-Free Graphs

Proof. Plugging in (6.15) into equation (6.18), we get

\[
\sum_{(p,q) \in \vec{P}} s(p,q) = c(M \oplus \vec{P}) - c(M) + \sum_{(p,q) \in \vec{P}} \delta_{pq} + \lambda_y(v) - \lambda_y(u),
\]

\[
\lambda_y(u) - \lambda_y(v) = c(M \oplus \vec{P}) - c(M) + \sum_{(p,q) \in \vec{P}} \delta_{pq} - s(p,q).
\]

Consider the case where \(\delta_{pq} = 1\). Since \((p, q)\) is admissible, \(s(p, q) = 0\) and we get \(\delta_{pq} - s(p, q) = 1\). Otherwise, \(\delta_{pq} \geq 2\sqrt{r}\), \((p, q)\) is a boundary edge, and \(s(p, q) \leq \sqrt{r}\). Then we get \(\delta_{pq} - s(p, q) \geq \delta_{pq}/2\). Summing over both cases gives us

\[
\sum_{(p,q) \in \vec{P}} \delta_{pq} - s(p,q) \geq \sum_{(p,q) \in \vec{P}} \frac{\delta_{pq}}{2},
\]

which gives us (6.19).

Corollary 1. Given an \(R\)-feasible matching \(M, y(v)\), suppose we have a simple alternating path or simple alternating cycle \(\vec{P}\) from \(u\) to \(v\) \((u = v\) if \(\vec{P}\) is a cycle) consisting only of admissible edges and let \(B(\vec{P})\) denote the edges participating in \(\vec{P}\) that are incident on boundary vertices. Then,

\[
\lambda_y(u) - \lambda_y(v) \geq c(M \oplus \vec{P}) - c(M) + |B(P)|\sqrt{r}.
\]

Proof. This follows easily from (6.19) and the fact that all values of \(\delta_{pq}\) for boundary edges \((p, q)\) are at least \(2\sqrt{r}\).
6.3 Scaling Algorithm

As in the Gabow-Tarjan Algorithm, for every edge \((a, b) \in E\), we redefine its weight to be \(c^*(a, b) = (kn + 1)c(a, b)\), where \(k\) is a constant defined in the upcoming Lemma 6.5. Since this uniform scaling of edge costs preserves the set of optimal matchings, Lemma 6.5 implies that a \(R\)-optimal matching of the vertices of \(A, B\) with edge weights \(c^*(\cdot, \cdot)\) corresponds to an optimal matching with the original edge costs \(c(\cdot, \cdot)\). For any edge \((u, v)\), let \(b_1, b_2, \ldots b_\ell\) be the binary representation of \(c^*(u, v)\). Let \(c_i(u, v)\), correspond to the most significant \(i\) bits of \(c^*(u, v)\). The following lemma bounds the cost of any \(R\)-optimal matching on \(G\).

Lemma 6.5. For a bipartite graph \(G(A \cup B, E)\) with a positive integer edge cost function \(c\), let \(M\) be an \(R\)-optimal matching and \(M_{\text{Opt}}\) be some optimal matching. Then, \(c(M) \leq c(M_{\text{Opt}}) + kn\) where \(k = (2k_1 + 4k_2 + 1)\) is a constant.

Proof. Edges that are in both \(M\) and \(M_{\text{Opt}}\) need not be considered. Since \(M\), \(y(\cdot)\) is \(R\)-optimal, all edges of \(M \setminus M_{\text{Opt}}\) satisfy (6.6), and we have

\[
c(M \setminus M_{\text{Opt}}) = \sum_{(u,v) \in M \setminus M_{\text{Opt}}} c(u,v) \leq \sum_{(u,v) \in M \setminus M_{\text{Opt}}} y(u) + y(v) + \delta_{uv}. \quad (6.21)
\]

Every edge in \(M_{\text{Opt}} \setminus M\) satisfies (6.5) so we have

\[
c(M_{\text{Opt}} \setminus M) = \sum_{(u,v) \in M_{\text{Opt}} \setminus M} c(u,v) \geq \sum_{(u,v) \in M_{\text{Opt}} \setminus M} y(u) + y(v) - \delta_{uv}. \quad (6.22)
\]

By subtracting (6.22) from (6.21), we have

\[
c(M) - c(M_{\text{Opt}}) \leq \sum_{(u,v) \in M \setminus M_{\text{Opt}}} y(u) + y(v) - \sum_{(u,v) \in M_{\text{Opt}} \setminus M} y(u) + y(v) + \sum_{(u,v) \in M \oplus M_{\text{Opt}}} \delta_{uv}. \quad (6.23)
\]
Since \( M_{\text{Opt}} \) and \( M \) are both perfect matchings,

\[
\sum_{(u,v) \in M \setminus M_{\text{Opt}}} y(u) + y(v) - \sum_{(u,v) \in M_{\text{Opt}} \setminus M} y(u) + y(v) = 0.
\]

Therefore, it is sufficient to bound \( \sum_{(u,v) \in M \oplus M_{\text{Opt}}} \delta_{uv} \).

\( \delta_{uv} \) can take one one of three values for every edge: 1, \( 2\sqrt{r} \), or \( \frac{m_jn}{m\sqrt{r}} \). There are at most \( k_1 \sqrt{r} \) boundary vertices per piece \( R_j \), and at most one edge in both \( M \) and \( M_{\text{Opt}} \) adjacent to each vertex, so there are at most \( 2k_1 \sqrt{r} \) edges \( (u,v) \) in \( M \oplus M_{\text{Opt}} \) such that \( \delta_{uv} = \frac{m_jn}{m\sqrt{r}} \). There are at most \( k_2 n/\sqrt{r} \) boundary vertices in the \( r \)-clustering, so there are at most \( 2k_2 n/\sqrt{r} \) edges in \( M \oplus M_{\text{Opt}} \) for which \( \delta_{uv} = 2\sqrt{r} \). For the other at most \( 2n \) edges \( (u,v) \) of \( M \oplus M_{\text{Opt}} \), \( \delta_{uv} = 1 \). Therefore,

\[
\sum_{(u,v) \in M \oplus M_{\text{Opt}}} \delta_{uv} \leq (\sum_j 2k_1 \sqrt{r} \frac{m_jn}{m\sqrt{r}}) + 2k_2 n(2\sqrt{r}) + 2n \\
\leq 2nk_1(\sum_j \frac{m_j}{m}) + (4k_2 + 2)n \\
\leq (2k_1 + 4k_2 + 2)n. \tag{6.24}
\]

Let \( k = (2k_1 + 4k_2 + 2) \). Then we have.

\[
c(M) \leq c(M_{\text{Opt}}) + kn.
\]

Using an almost identical argument, we can also bound the cost of any \( \mathcal{R} \)-feasible matching \( M \) (not necessarily perfect) and dual weights \( y(\cdot) \) by \( c(M_{\text{Opt}}) + O(n) \) so long as every free
vertex of $A$ has a dual weight of 0 and every free vertex of $B$ has a positive dual weight.

**Lemma 6.6.** For a bipartite graph $G(A \cup B, E)$ with a positive integer edge cost function $c$, let $M$ along with $y(\cdot)$ be an $\mathcal{R}$-feasible matching such that every free vertex $b$, $y(v) \geq 0$ and for every free vertex $a \in A$, $y(a) = 0$, and let $M_{\text{Opt}}$ be any optimal matching. Then, $c(M) \leq c(M_{\text{Opt}}) + kn$ where $k = (2k_1 + 4k_2 + 1)$ is a constant.

**Proof.** Using (6.23), it is sufficient to show that

$$\sum_{(u,v) \in M \setminus M_{\text{Opt}}} y(u) + y(v) - \sum_{(u,v) \in M_{\text{Opt}} \setminus M} y(u) + y(v) \leq 0.$$ 

Since $M_{\text{Opt}}$ is perfect, we can rewrite the left side as $\sum_{u \in F} y(u)$ where $F$ is the set of free vertices with respect to $M$. The fact that all free vertices have non-negative dual weight gives the lemma. \qed

Similar to Gabow and Tarjan’s algorithm, our algorithm consists of scales. The input to any scale $i$ is a bipartite graph on $A, B$, with a cost function $c_i(\cdot, \cdot)$, and a set of dual weights $y(v)$ for every node $v \in A \cup B$. Let the $c^i(u, v) = c_i(u, v) - y(u) - y(v)$ be the reduced cost of $(u, v)$. Then the reduced costs $c^i(\cdot, \cdot)$ satisfy the following properties:

(E1) For every edge $(u, v)$, $c^i(u, v) \geq \delta_{uv}$, and,

(E2) The cost of a 1-optimal matching with respect to $c^i(\cdot, \cdot)$ is $O(n)$.

Given an input with these properties, the algorithm within a scale returns an $\mathcal{R}$-optimal matching $M$ and dual weights $y'(\cdot)$ with respect to the reduced costs $c^i(\cdot, \cdot)$. Reduced costs do not affect the optimal matching. Furthermore, for every vertex $v \in A \cup B$, let $y(v)$ be the sum of the dual weight $y'(v)$ and the dual weight of $v$ that was provided as input to scale $i$. It can be shown that $M, y(\cdot)$ is also $\mathcal{R}$-optimal with respect to $c_i(\cdot, \cdot)$.
The input to the first scale is the graph \( G(A \cup B, E) \) with the cost \( c_1(\cdot, \cdot) \) and a set of dual weights of \(-1\) on every internal vertex \( u \) of \( B \), dual weight of \(-\max_{v \in V} \delta_{uv}\) for every boundary vertex \( u \) of \( B \) and dual weights of \(0\) on every vertex of \( A \). It is easy to see that \( c_1(\cdot, \cdot) \) satisfies (E1) and (E2). For any scale \( i \), using \( c_i(\cdot, \cdot) \) as the cost, the algorithm for a scale (described below) computes a matching \( M \) and dual weights \( y(\cdot) \) so that \( M, y(\cdot) \) is an \( \mathcal{R} \)-optimal matching.

For any scale \( i \geq 1 \), we use the \( \mathcal{R} \)-optimal matching \( M, y(\cdot) \) returned by the algorithm for scale \( i \) to generate an input for scale \( i + 1 \) as follows. It is possible that for any matching edge \((u, v)\) of the \( \mathcal{R} \)-optimal matching \( M \), the sum of the dual weights \( y(u) + y(v) \) greatly exceeds \( c(u, v) \) (see (6.6)). Prior to moving the dual weights to scale \( i + 1 \), the algorithm sets for all edges \((a, b) \in M\), where \( a \in A \) and \( b \in B \),

\[
y(b) \leftarrow y(b) - s(a, b).
\]

It is easy to see that this results in an \( \mathcal{R} \)-optimal matching \( M \) and new dual weights \( y(\cdot) \) such that the slack of every matching edge is 0. Note that reducing \( y(b) \) only increases the slack on adjacent nonmatching edges. The algorithm transfers the dual weights from scale \( i \) to scale \( i + 1 \) as follows. For any vertex \( v \in A \cup B \),

\[
y(v) \leftarrow 2y(v) - 2 \max_{u \in N(v)} \delta_{uv}.
\]

Therefore, at the beginning of scale \( i + 1 \), the reduced cost \( c^{i+1}(u, v) = c_{i+1}(u, v) - y(u) - y(v) \) on every edge is at least \( \delta_{uv} \) implying (E1). The cost of the optimal matching for scale \( i + 1 \), \( M_i \), with respect to the new costs \( c^{i+1}(\cdot, \cdot) \) is at most \( \sum_{(u, v) \in M_i} 2\delta_{uv} + \sum_{v \in A \cup B} 2 \max_{u \in N(v)} \delta_{uv} + n \). Following similar steps as those used in showing (6.24) gives that the cost of the optimal matching or \( \mathcal{R} \)-optimal matching with respect to \( c^{i+1}(\cdot, \cdot) \) is \( O(n) \), implying (E2).
In the next section, we describe an algorithm for each scale. This algorithm takes a graph with positive integer edge costs where the cost of each edge $c(u, v) \geq \delta_{uv}$ (condition (E1)) and the optimal matching has a cost of $O(n)$ (condition (E2)). Given this input, it computes an $R$-optimal matching in $\tilde{O}(nm^{5/2})$ time. After $O(\log((kn + 1)C))$ scales, the $R$-optimal matching returned by our algorithm will also be an optimal matching.

### 6.4 Algorithm for Each Scale

Our algorithm takes a bipartite graph $G(A \cup B, E)$ and its $r$-clustering as input. Each edge $(u, v)$ of this graph has a positive integer cost of $c(u, v)$ with $c(u, v) \geq \delta_{uv}$ and the optimal matching has a cost no more than $O(n)$. Given such an input, it produces an $R$-optimal matching.

The algorithm has three steps. The first step of the algorithm will execute $\sqrt{r}$ iterations of a scale of the Gabow-Tarjan algorithm [18]. This can be executed in $O(m\sqrt{r})$ time. At the end of this step, the algorithm has a 1-feasible matching $M$ and dual weights $y(\cdot)$ that satisfy the original dual feasibility conditions of Gabow and Tarjan ((6.3) and (6.4)) and there are at most $O(n/\sqrt{r})$ free vertices. Additionally, from the properties of Gabow-Tarjan algorithm, for every free vertex $a \in A$, $y(a) = 0$, and for every free vertex $b \in B$, $y(b) \geq \sqrt{r}$. Furthermore, the dual adjustments performed during an iteration of Gabow and Tarjan’s algorithm only decrease dual weights of $A$ and increase dual weights of $B$. A 1-feasible matching also satisfies the requirements for an $R$-feasible matching ((6.5) and (6.6)).

Therefore, at the end of the first step, we have the following.

**Lemma 6.7.** At the end of the first step of our algorithm, the matching $M$ and the dual weights $y(\cdot)$ form an $R$-feasible matching for the graph $G$, and the number of free vertices with respect to $M$ is at most $O(n/\sqrt{r})$. For every vertex $a \in A$, $y(a) \leq 0$ and for every
vertex \( b \in B, y(b) \geq 0 \). For every free vertex \( a \in A, y(a) = 0 \), and for every free vertex \( b \in B, y(b) = \max_{b' \in B} y(b') \geq \sqrt{r} \).

To match the remaining \( O(n/\sqrt{r}) \) unmatched vertices, the algorithm will use the \( r \)-clustering to construct a compressed residual graph similar to that used in [6]. This compressed graph has \( O(n/\sqrt{r}) \) vertices and \( O(n) \) edges.

The algorithm of [6] uses shortest path data structures for planar graphs to iteratively compute the remaining \( O(n/\sqrt{r}) \) augmenting paths. Computing each augmenting path takes \( \tilde{O}(n/\sqrt{r}) \) time, so the total time taken is bounded by \( \tilde{O}(n^2/r) \). However, while working with arbitrary graphs with an \( r \)-clustering, there are no known efficient shortest-path data structures, so computing each path takes \( O(m) \) time for a total time of \( \tilde{O}(nm/\sqrt{r}) \). One should note that for any value of \( r \), this running time is \( \Omega(mn^{1/2}) \). To achieve an improved execution time, our algorithm will compute many augmenting paths in one round. This is similar to the Hopcroft-Karp and Gabow-Tarjan algorithms, which converge to a maximum matching in \( O(\sqrt{n}) \) phases. In our case, we show an even faster convergence in only \( \tilde{O}(\sqrt{n/\sqrt{r}}) \) phases, with the search of each phase (on an average) taking \( \tilde{O}(m) \) time. For \( r = n^{2/5} \), we achieve a running time of \( O(mn^{2/5}) \) time.

The second step of the algorithm takes the matching \( M \) and set of dual weights \( y(\cdot) \) produced by the first step as input. Using a compressed residual graph, it then computes an \( \mathcal{R} \)-feasible matching that matches all but \( \tilde{O}(\sqrt{n/\sqrt{r}}) \) vertices. We describe this step next.

We begin by defining the compressed residual graph and formally introduce compressed feasible matchings. After that, we will show that we can quickly convert any \( \mathcal{R} \)-feasible matching to a compressed feasible matching and vice versa. Finally, we will present the second step of our algorithm.
Active and Inactive Free Vertices During the second step of the algorithm, the dual weight of a free vertex $b \in B$ may exceed a predetermined upper bound $\beta = \lceil \sqrt{n/r} \rceil \sqrt{r}$. Such vertices are called inactive. More specifically, a free vertex $b$ is inactive with respect to an $\mathcal{R}$-feasible matching $M, y(\cdot)$ if $y(b) \geq \beta$. All other free vertices of $B$ are active. In our algorithm, each piece $\mathcal{R}_j$ will have a corresponding $\mathcal{R}$-feasible matching. Therefore, each piece has its own set of inactive vertices, $B^I_j$, and its own set of active vertices, $B^A_j$. During the second step of the algorithm, we only search for an augmenting path from active free vertices. The second step of the algorithm terminates when there are no active free vertices remaining.

Compressed Residual Graph $H$ We now describe the compressed residual graph $H$, which will be useful in the fast execution of the second step. This compressed graph is similar to the one defined in [6]. However, there are two differences from the definition of [6]. First, the vertex set of $H$ is modified to include inactive and active free vertices. In addition, the edges of $H$ will include a self-loop edge from every boundary vertex to itself.

For a matching $M$, let $G_M$ denote the (directed) residual graph with respect to $M$. Let $\mathcal{R}(G_M) = \mathcal{R}(G)$ be the $r$-clustering of $G_M$ as given by Definition 6.2. Let $A_F$ and $B_F$ denote the set of free vertices (vertices not matched by $M$) of $A$ and $B$ respectively. Let $B^A_F$ (resp. $B^I_F$) be the set of free vertices that are also active (resp. inactive). Our sparse graph $H$ will be a weighted multi-graph whose vertex set $V_H$ and the edge set $E_H$ is defined next.

We define the vertex set $V^H_j$ and the edge set $E^H_j$ for each piece $\mathcal{R}_j$. The vertex set $V_H$ and the edge set $E_H$ are simply the union of all the vertices and edges across all pieces. For every piece $\mathcal{R}_j$, $V^H_j$ contains the boundary vertices $\mathcal{K}_j$. Also, if there is at least one internal vertex of $B$ that is also an active free vertex, i.e., $(V_j \setminus \mathcal{K}_j) \cap B^A_F \neq \emptyset$, then we create a special vertex $b^A_j$ to represent all vertices of this set in $V^H_j$. We also create a vertex $b^I_j$
to represent all inactive free vertices of \((V_j \setminus K_j) \cap B^2_j\). Similarly, we create a vertex \(a_j\) to represent all the free vertices in \((V_j \setminus K_j) \cap A_F\) if any exist. We refer to the three additional vertices for \(R_j\) the free internal vertices of \(R_j\) and refer to \(b^A_j\) (resp. \(b^F_j\)) as the active (resp. inactive) free internal vertex. We set \(V^H_j = K_j \cup \{a_j, b^I_j, b^F_j\}\), \(A^H_j = (K_j \cap A) \cup \{a_j\}\), and \(B^H_j = (K_j \cap B) \cup \{b^A_j, b^F_j\}\). The free vertices of \(B\) in piece \(R_j\) of the compressed graph \(H\) are represented by \(B^F_H = (B_F \cap K_j) \cup \{b^A_j, b^F_j\}\) and the free vertices of \(A\) in piece \(R_j\) of \(H\) are represented by \(A^F_H = (A_F \cap K_j) \cup \{a_j\}\). The vertex set \(V^H\) of \(H\) is thus given by \(V^H = \bigcup_{j=1}^l V^H_j\). We also define the sets \(B^H_H = \bigcup_{j=1}^l B^H_j\) and \(A^H_H = \bigcup_{j=1}^l A^H_j\). The free vertices of \(B\) in \(H\) are denoted by \(B^F_H = \bigcup_{j=1}^l B^F_j\) and the free vertices of \(A\) is denoted by \(A^F_H = \bigcup_{j=1}^l A^F_j\). The vertices in \(H\) represent sets of vertices in \(G\). Given a vertex in \(G\), we describe the corresponding vertex in \(H\) as a representative vertex.

Next we define the set of edges \(E^H_j\) for each pieces \(R_j\). Each edge \((u, v)\) in \(E^H_j\) will represent a corresponding shortest net-cost path from \(u\) to \(v\) in \(R_j\). We denote this path as \(\overrightarrow{P}_{u,v,j}\) and describe this mapping from an edge of \(E^H\) to its corresponding path in \(G\) as projection. For any \(u, v \in V^H_j\), where \(u\) and \(v\) are allowed to be the same vertex, there is an edge from \(u\) to \(v\) if

1) \(u, v \in K_j\), i.e., \(u\) and \(v\) are boundary vertices and there is a directed path \(\overrightarrow{P}\) from \(u\) to \(v\) in \(G_M\) that only passes through the edges of the piece \(R_j\). Let \(\overrightarrow{P}_{u,v,j}\) be the path consisting only of edges of \(R_j\) that has the smallest net-cost. We denote this type of edge as a boundary-to-boundary edge. When \(u = v\), \(\overrightarrow{P}_{u,v,j}\) is the smallest net-cost cycle inside \(R_j\) that contains the vertex \(u\). Since \(R_j\) is a bipartite graph, such a cycle must consist of at least 4 edges.

2) \(u = \{b^A_j, b^F_j\}\), \(v \in K_j\), and there is a directed path \(\overrightarrow{P}\) in \(G_M\) from some free vertex in \(B^A_F \cap (V_j \setminus K_j)\) (if \(u = b^A_j\)) or \(B^F_F \cap (V_j \setminus K_j)\) (if \(u = b^F_j\)) to \(v\) that only passes through
6.4. Algorithm for Each Scale

Figure 6.1: (a) A piece $\mathcal{R}_j$. The squares represent vertices of $B$ and the circles represent vertices of $A$. Filled vertices are free internal vertices. Specifically, the filled diamond represents an inactive free vertex, and the filled squares represent active free vertices. (b) The boundary-to-boundary edges and self-loop edges of $H$ for $\mathcal{R}_j$. (c) The edges from $b^A_j$ and $b^F_j$ to $\mathcal{K}_j$. (d) The edges from $\mathcal{K}_j$ to $a_j$. (e) There is a single edge from both $b^A_j$ and $b^F_j$ to $a_j$.

the edges of $\mathcal{R}_j$. Let $\overrightarrow{P}_{u,v,j}$ be the path consisting only of edges of $\mathcal{R}_j$ that has the smallest net-cost.

3) $u \in \mathcal{K}_j$, $v = a_j$, and there is a directed path $\overrightarrow{P}$ in $G_M$ from $u$ to some free vertex in $A_F \cap (V_j \setminus \mathcal{K}_j)$ that only passes through the edges of $\mathcal{R}_j$. Let $\overrightarrow{P}_{u,v,j}$ be the path consisting only of edges of $\mathcal{R}_j$ that has the smallest net-cost.

4) $u = \{b^A_j, b^F_j\}$ and $v = a_j$ are free vertices and there is a directed path $\overrightarrow{P}$ in $G_M$ from some vertex in the set $B^A_F \cap (V_j \setminus \mathcal{K}_j)$ (if $u = b^A_j$) or $B^F_F \cap (V_j \setminus \mathcal{K}_j)$ (if $u = b^F_j$) to a vertex in the set $A_F \cap (V_j \setminus \mathcal{K}_j)$ that only passes through the edges in $\mathcal{R}_j$. Let $\overrightarrow{P}_{u,v,j}$ be the path consisting only of edges of $\mathcal{R}_j$ that has the smallest net-cost.

See Figure 6.1 for an example piece of $H$ from a piece of $G$. We set the weight of each edge to be $\phi(\overrightarrow{P}_{u,v,j})$. We also refer to this edge $(u, v)$ as an edge of piece $\mathcal{R}_j$ in $H$ and denote the set of all edges of piece $\mathcal{R}_j$ as $E^H_j$. The set of edges of $H$ is simply $E_H = \bigcup_j E^H_j$. Note that $H$ is a multi-graph as there can be directed path from $u$ and $v$ in multiple pieces.

Note that the number of vertices in $H$ is $O(n/\sqrt{r})$. For each vertex $v \in \mathcal{X}$, let $\theta_v$ be the
number of pieces in which it belongs. Counting multiplicity, the number of boundary vertices is \( O(n/\sqrt{r}) \) and therefore,

\[
\sum_{v \in \mathcal{X}} \theta_v = O(n/\sqrt{r}).
\]

Any boundary vertex \( v \) in \( H \) can have edges to at most \( \sqrt{r} \) boundary vertices inside any piece it participates in. Therefore, the total number of edges can be bounded by

\[
\sum_{v \in \mathcal{X}} \theta_v \sqrt{r} = O(n),
\]

leading to the following lemma.

**Lemma 6.8.** The compressed residual graph \( H \) has \( O(n/\sqrt{r}) \) vertices and \( O(n) \) edges.

This completes the description of the compressed residual graph \( H \). The compressed residual graph defined here differs from the one in [6] in two ways: We classify free vertices as active and inactive and we allow for self-loops in the compressed residual graph \( H \). We describe the reasons for introducing these changes.

**Need for Active and Inactive Vertices** It is necessary to differentiate between active and inactive vertices because, unlike in our variant of Gabow-Tarjan, we cannot guarantee
that every vertex in $H$ is visited only once in each phase of the algorithm. In fact, we can construct an example where a vertex can be visited an arbitrary number of times in the same phase. In Figure 6.2, a DFS style search on the admissible edges of $H$ (those with slack $\leq \sqrt{r}$) visits $u$ and then $z$. There are no admissible edges from $z$ and so, the search backtracks and goes to $v$ which leads to an augmenting path. After augmenting along the path, the edge $(y,z)$ is created in $H$. Now, in the same iteration, another DFS from a different free vertex may visit $y$ and then $z$ again (note that $(y, z)$ has a slack less than $\sqrt{r}$).

Similar to the Gabow-Tarjan’s variant, we can guarantee that every time a vertex in $H$ is visited, it is either part of an augmenting path (or alternating cycle/path), or its dual weight increases by $\sqrt{r}$. A vertex that is visited many times during the execution of the algorithm, therefore, will accumulate a dual weight with a very large magnitude. When $|\bar{y}(v)|$ for some vertex $v$ in $B_H$ reaches the upper bound $\beta$ in the second step, we flip edges along an alternating path to make $v$ an inactive free vertex, without decreasing the size of the matching. The vertex $v$ will then remain unused for the remainder of the second step.

**Need for Self Loops** Unlike Gabow and Tarjan’s algorithm, we do not compute the shortest augmenting path. So, in order to guarantee that the paths we find in $H$ are simple, we need to actively look for possible cycles. Such a cycle may lie entirely within a piece and involve only a single boundary vertex. Including self-loops in $H$ is helpful in detecting such cycles.

**Compressed Feasibility** Next, we will define the requirements for a compressed feasible matching. We denote as $M_j$ the edges of $M$ that belong to piece $\mathcal{R}_j$. $M = \bigcup_{j=1}^f M_j$. For any vertex $v \in V_H$, let $\lambda_v$ be $-1$ if $v \in A_H$ and $1$ if $v \in B_H$. For each piece $\mathcal{R}_j$, we maintain a dual weight $y_j(v)$ for every vertex in $v \in V_j$. These dual weights $y_j(\cdot)$ along with $M_j$ form
an \( R \)-feasible matching. Additionally, we store a dual weight \( \tilde{y}(v) \) for every \( v \in V_H \). We say that the dual weights \( \tilde{y}(\cdot) \) are \( H \)-feasible if they satisfy the following conditions. For each piece \( R_j \), and for every directed edge \((u, v) \in E_H^j\),

\[
\lambda_u \tilde{y}(u) - \lambda_v \tilde{y}(v) \leq \phi(\overrightarrow{P}_{u,v,j}) \tag{6.25}
\]

For any graph \( G \), and an \( r \)-clustering \( R(G) \), we say that a matching \( M \), a set of dual weights \( y_j(\cdot) \) for the vertices of each piece \( R_j \), and a set of dual weights \( \tilde{y}(\cdot) \) for the vertices \( V_H \), form a compressed feasible matching if the following conditions (a)–(e) are satisfied.

(a) For every vertex \( v \in A_H \), \( \tilde{y}(v) \leq 0 \) and for every free vertex \( v \in A^F_H \), \( \tilde{y}(v) = 0 \).

(b) Let \( y_{\text{max}} \) be \( \max_{v \in B^A_H} \tilde{y}(v) \). For every vertex \( v \in B_H \), \( \tilde{y}(v) \geq 0 \) and for all active vertices \( v \in B^A_H \), \( y_{\text{max}} - \sqrt{r} \leq \tilde{y}(v) \leq y_{\text{max}} \). For all inactive vertices \( v \in B^F_H \), \( \tilde{y}(v) \geq \beta \).

(c) For every piece \( R_j \), the matching \( M_j \) and dual weights \( y_j(\cdot) \) form an \( R \)-feasible matching.

(d) The dual weights \( \tilde{y}(\cdot) \) are \( H \)-feasible.

(e) For each piece \( R_j \) and any \( v \in K_j \), \( |\tilde{y}(v)| \geq |y_j(v)| \). For every vertex \( a \in (V_j \setminus K_j) \cap A_F \), \( |y_j(a)| = 0 \).

Using (a) and (b), we can restate the \( H \)-feasibility conditions compactly as

\[
|\tilde{y}(u)| - |\tilde{y}(v)| \leq \phi(\overrightarrow{P}_{u,v,j}). \tag{6.26}
\]

and we can define the slack of an edge \((u, v) \in E_H\) to be

\[
s_H(u, v) = \phi(\overrightarrow{P}_{u,v,j}) - |\tilde{y}(u)| - |\tilde{y}(v)|.
\]

Let \( u_0 \) be the first vertex and \( v_\ell \) be the last vertex of \( \overrightarrow{P}_{u,v,j} \). Note that, if \( \tilde{y}(u) = y_j(u_0) \) and
\[ \tilde{y}(v) = y_j(v) \] then, using (6.18),

\[
s_H(u, v) = \sum_{(u', v') \in \tilde{P}_{u,v,j}} s(u', v').
\]

(6.27)

We say that an edge \((u, v) \in E_H\) is *admissible* if the slack \(s_H(u, v) \leq \sqrt{r}\).

Note that a boundary vertex has many different dual weights assigned to it, one for each of the pieces it belongs to. During the course of our algorithm, the magnitudes of the dual weights of vertices in \(H\) only increase (with a few exceptions). As we do not immediately update the dual weights of all vertices in \(G\), for some piece \(R_j\) the dual weight \(y_j(\cdot)\) may not reflect the updated dual weight. This condition is captured by (e).

**Conventions for Notations in a Compressed Residual Graph.** For every boundary or free internal vertex, we define the *representative* of \(v\) to be \(v\) itself if \(v\) is a boundary node. Otherwise, if \(v\) is a free internal vertex, then it is one of the three free internal vertices \(a_j\), \(b_j^T\) or \(b_j^A\) depending on whether \(v\) is a free vertex of \(A\), free inactive vertex or a free active vertex respectively. We denote the representative of \(v\) by \(\text{rep}(v)\). For simplicity in exposition, wherever convenient, we will abuse notations and use \(v\) to also denote this representative \(\text{rep}(v)\) in \(H\). Following our convention, we set \(H'\) to be a graph identical to \(H\) with the weight of every edge replaced by its slack.

The compressed residual graph allows our algorithm to search for paths by modifying \(\tilde{y}(\cdot)\) values in \(H\) without explicitly modifying the \(y_j(\cdot)\) for every piece \(R_j\). In doing so, there may be a free vertex \(b \in B_F \cap V_j\) such that \(\tilde{y}(\text{rep}(b))\) may exceed \(\beta\) whereas \(y_j(b)\) remains below \(\beta\). In such a situation, our convention is to assume \(b\) to be inactive with respect to \(R_j\). Thus, when a vertex of \(H\) becomes inactive, all the vertices it represents also become inactive. This convention fits with the notion that the values \(\tilde{y}(\cdot)\) can be seen as up to date,
while the values $y_j(\cdot)$ are lazily updated.

To overcome mild technical challenges encountered in the presentation of the algorithm, we introduce two useful procedures next.

**Procedures for Reducing Dual Weight Magnitudes** We describe two useful procedures, called $\text{REDUCE}(b_j^A / b_j^I, \alpha)$ and $\text{REDUCESLACK}(v)$, that allow us to reduce the dual weights of vertices of $B$ without violating compressed feasibility. During the second step of the algorithm, the dual weights may decrease in magnitude only within these two procedures. Otherwise, the magnitude of the dual weights only increase. We describe these procedures next.

$\text{REDUCE}$ takes as input a free active (resp. inactive) internal vertex $b_j^A$ (resp. $b_j^I$), and a value $\alpha$ such that $0 \leq y_{\text{max}} - \sqrt{r} \leq \alpha \leq \bar{y}(b_j^A)$ (resp. $\beta \leq \alpha \leq \bar{y}(b_j^I)$). For all $v \in (V_j \setminus K_j) \cap B_j^A$ (resp. $B_j^I$), if $y_j(v) \geq \alpha$, it sets the dual weight $y_j(v) \leftarrow \alpha$. Then it sets $\bar{y}(b_j^A)$ (resp. $\bar{y}(b_j^I)$) to $\alpha$.

$\text{REDUCESLACK}$ takes as input a matched vertex $v \in B$. Let $u \in A$ be the vertex that $v$ is matched to, and let $(u, v)$ belong to the piece $R_j$. The procedure sets $y_j(v) \leftarrow y_j(v) - s(u, v)$. If $v$ is also a boundary vertex, i.e., $v \in K_j$, then it sets $\bar{y}(v) \leftarrow y_j(v)$ and for every other piece $R_{j'}$ such that $v \in V_{j'}$, it sets $y_{j'}(v) \leftarrow y_j(v)$.

For a discussion on why the $\text{REDUCE}$ and $\text{REDUCESLACK}$ procedures do not violate the compressed feasibility conditions, see Section 6.7.1 of the appendix. From that discussion, we get the following Lemma.

**Lemma 6.9.** Invoking $\text{REDUCE}$ or $\text{REDUCESLACK}$ procedures on a compressed feasible matching will not violate any of the compressed feasibility conditions (a)–(e).

At the end of Step 1 of our algorithm, we have a matching $M$ and a set of dual weights...
that form an $\mathcal{R}$-feasible matching. In Section 6.4.1, we describe how to compute and store the edges of the compressed feasible matching from this $\mathcal{R}$-feasible matching $M, y(\cdot)$. The algorithm will then maintain a compressed feasible matching until the end of the scale. At the end of each scale, our algorithm produces a compressed optimal matching. In Section 6.4.2, we describe how to compute an $\mathcal{R}$-optimal matching from this compressed optimal matching. Using some of the procedures introduced in Sections 6.4.1 and 6.4.2, we describe the second step of our algorithm in Section 6.4.3.

### 6.4.1 Computing a Compressed Feasible Matching from an $\mathcal{R}$-Feasible Matching

At the end of the first step, we have an $\mathcal{R}$-feasible matching $M, y(\cdot)$ that also satisfies $y(u) = 0$ for all $u \in A_F$ and $y(v) = \max_{v' \in B} y(v')$ for all $v \in B_F$. In this section, we will present an algorithm to compute a compressed residual graph and a compressed feasible matching from this $\mathcal{R}$-feasible matching $M$ and its set of dual weights $y(\cdot)$. For every vertex $v \in A \cup B$, and for every piece $\mathcal{R}_j$ such that $v \in V_j$, we set $y_j(v) = y(v)$. For every boundary vertex $v \in K$, we set $\tilde{y}(v) = y(v)$. We also set, for every piece $\mathcal{R}_j$, $\tilde{y}(a_j) = 0$ and $\tilde{y}(b_j^A) = \gamma$ where $\gamma \geq \sqrt{r}$ is the dual weight of all free vertices of $B_F$ from the first step. Note that $\beta > \gamma$ and therefore, there is no inactive free internal vertex at the end of the first step. So, we do not create a free inactive internal vertex for any piece. Note that, from Lemma 6.7, conditions (a), (b) and (e) are trivially satisfied. (c) is satisfied because the matching $M_j$ and $y_j(\cdot)$ form a 1-feasible matching. Edges satisfying 1-feasibility conditions also satisfy the $\mathcal{R}$-feasibility condition and so $M_j$ and $y_j(\cdot)$ is $\mathcal{R}$-feasible. The next lemma shows that dual weights $\tilde{y}(\cdot)$ satisfy $H$-feasibility and therefore (d) holds.

**Lemma 6.10.** Consider a matching $M_j$ and a set of dual weights $y(\cdot)$ for a piece $\mathcal{R}_j$ such that
Chapter 6. Minimum-Cost Matching on Minor-Free Graphs

$M_j, y(\cdot)$ is $\mathcal{R}$-feasible. Suppose the dual weights of all vertices of $A$ in $V_j$ are non-positive and the dual weights of all vertices of $B$ in $V_j$ are non-negative. For any two vertices $u, v \in V_j$, let the directed path $\bar{P}_{u,v,j}$ be a minimum net-cost path from $u$ to $v$ in $\mathcal{R}_j$. Then,

$$|y(u)| - |y(v)| \leq \phi(\bar{P}_{u,v,j}). \tag{6.28}$$

Furthermore,

$$\sum_{(a,b) \in \bar{P}_{u,v,j}} s(a, b) = \phi(\bar{P}_{u,v,j}) - |y(u)| + |y(v)|. \tag{6.29}$$

**Proof.** From equation (6.18) we have,

$$\sum_{(a,b) \in \bar{P}} s(a, b) = \phi(\bar{P}) - \lambda_u y(u) + \lambda_v y(v).$$

From the fact that all vertices of $B$ have non-negative dual weight and all vertices of $A$ have non-positive dual weight, we get that $\lambda_u = |y(u)|$ and $\lambda_v = |y(v)|$. This gives (6.29). From the fact that all slacks in an $\mathcal{R}$-feasible matching are non-negative, (6.28) follows.

Using the following lemma, we will provide a procedure called $\text{CONSTRUCT}$ to compute all the edges of $H$.

**Lemma 6.11.** Let $\mathcal{R}'_j$ be a directed graph identical to the directed graph $\mathcal{R}_j$ except that the cost of any edge $(a, b)$ is set to be its slack $s(a, b)$. Then, for any two vertices $u, v$ in $\mathcal{R}_j$, the minimum net-cost directed path from $u$ to $v$ in $\mathcal{R}_j$ is the minimum cost directed path between $u$ and $v$ in $\mathcal{R}'_j$. The dual weights $y(u), y(v)$ and the length of the shortest path in $\mathcal{R}'_j$ immediately give us the value of the minimum net-cost between $u$ and $v$ in $\mathcal{R}_j$. 

\qed
Proof. Let \( P \) be any alternating path between vertices \( u \) and \( v \). We can write the net-cost of \( P \) as in equation (6.29).

\[
\min_P \phi(P) = \min_P \sum_{(a,b) \in P \setminus M} s(a, b) + |y(u)| - |y(v)|.
\]

For every path, \( y(u) \) and \( y(v) \) are the same. Therefore, we conclude that computing minimum net-cost path is equivalent to finding the minimum-cost path \( P^* \) between \( u \) and \( v \) in \( R'_j \). Furthermore, the sum of the cost of \( P^* \) with \( y(u) \) and \( y(v) \) will give the value of the minimum net-cost between \( u \) and \( v \).

We define the slack on any directed edge \((u,v) \in E^H_j\) to be

\[
s_H(u,v) = \phi(\overrightarrow{P}_{u,v,j}) - |\tilde{y}(u)| + |\tilde{y}(v)|.
\]

From the Lemma 6.10 above, it follows that slack of the edge \((u,v)\) is non-negative, and exactly equal to \( \sum_{(a,b) \in \overrightarrow{P}_{u,v,j}} s(a, b) \), provided \( \tilde{y}(u) = y_j(u) \) and \( \tilde{y}(v) = y_j(v) \). Following our convention, we use \( H' \) to denote the compressed residual graph with the same edge set as \( H \) but with the edge weights being replaced with their slacks. Our initial choice of \( \tilde{y}(\cdot) \) is \( H \)-feasible. To assist in the execution of the second step of our algorithm, we explicitly compute the edges of \( H \) and sort them in increasing order of their slacks.

Using lemma 6.11, we describe a procedure that computes the edges \( E^H_j \) for a piece \( R_j \). This process will be referenced as the **Construct** procedure. Note that in some graphs, such as planar graphs, **Construct** could use a faster algorithm; see Section 6.6.

**The Construct Procedure** This procedure takes a piece \( R_j \) of \( G_M \) as input and constructs the edges of \( E^H_j \). We next give a summary of how to accomplish this, for further
details, see appendix section 6.7.2. Let $\mathcal{R}_j'$ be the graph of $\mathcal{R}_j$, with all edge weights converted to their slacks according to the current matching $M_j$ and the current dual assignment $y_j(\cdot)$. From equation (6.29) and Lemma 6.11, it is sufficient to compute the shortest path distances in terms of slacks between all pairs of vertices in $V_j^H$. These distances in slacks can then each be converted to net-costs in constant time. Therefore, using $O(\sqrt{r})$ separate Dijkstra searches over $\mathcal{R}_j'$, with each search taking $O(m_j + n_j \log n_j)$ time, the edges of $E_j^H$ can be computed. From the discussion in appendix section 6.7.2, we get the following Lemma and Corollary.

**Lemma 6.12.** Given an $\mathcal{R}$ feasible matching $M_j$, $y_j(\cdot)$, the CONSTRUCT procedure builds the edges of $E_j^H$ in $O(\sqrt{r}(m_j + n_j \log n_j))$ time.

**Corollary 2.** Let $\mathcal{R}$-feasible matching $M$, $y(\cdot)$ be the matching computed at the end of the first step of our algorithm. Given $M$, $y(\cdot)$, we can use the CONSTRUCT procedure to compute the graph $H$ in $O(\sqrt{r}(m + n \log n))$ time.

### 6.4.2 Computing an $\mathcal{R}$-Feasible Matching from a Compressed Feasible Matching

Recollect that an $\mathcal{R}$-feasible matching was obtained at the end of the first step of our algorithm. In the previous section, we described how we can compute a compressed feasible matching from this $\mathcal{R}$-feasible matching. In step 2 and step 3 our algorithm will iteratively increase the size of the matching while maintaining a compressed feasible matching, eventually ending with a compressed optimal matching. However, at the end of each scale, we desire an $\mathcal{R}$-optimal matching. In this section, we introduce a procedure called SYNC that allows the algorithm to convert any compressed feasible matching to an $\mathcal{R}$-feasible matching in $O(m + n \log n)$ time.
In a compressed feasible matching, a boundary vertex has multiple dual weights, one corresponding to each of the pieces it belongs to. It also has a dual weight $\tilde{y}(\cdot)$ with respect to the graph $H$. We introduce a synchronization procedure (called SYNC) that will take a compressed feasible matching along with a piece $R_j$ and update the dual weights $y_j(\cdot)$ so that the new dual weights and the matching $M_j$ continue to be $R$-feasible and for every boundary vertex $v \in K_j$, $y_j(v) = \tilde{y}(v)$. We can convert a compressed feasible matching into an $R$-feasible matching by repeatedly invoking this procedure for every piece.

The SYNC procedure is implemented as follows:

- Recollect that the graph $R'_j$ is a graph identical to $R_j$ with slacks as the edge costs. Temporarily add a new source vertex $s$ to $R'_j$ and add an edge from $s$ to every $v \in K_j$. Also add an edge from $s$ to every unmatched internal vertex $v$ of $A$ and $B$ in $R'_j$, i.e., $v \in ((V_j \setminus K_j) \cap (A_F \cup B_F))$. For any such vertex $v \in K_j \cup (V_j \cap (A_F \cup B_F))$, let $\kappa_v = |\tilde{y}(v)| - |y_j(v)|$ and let $\kappa = \max_{v \in K_j \cup (V_j \cap (A_F \cup B_F))} \kappa_v$. Set the weight of the newly added edge from $s$ to $v$ to $\kappa - \kappa_v$. This new graph has only non-negative edge costs.

- Execute Dijkstra’s algorithm on this graph beginning from the source vertex $s$. Let $\ell_v$ be the length of the shortest path from $s$ to $v$ as computed by this execution of Dijkstra’s algorithm. For each vertex $v \in V_j$, if $\ell_v > \kappa$, then do not change its dual weight. Otherwise, change the dual weight $y_j(v) \leftarrow y_j(v) + \lambda_v(\kappa - \ell_v)$.

This completes the description of the SYNC procedure. The SYNC procedure executes Dijkstra’s algorithm on $R'_j$ with an additional vertex $s$ and updates the dual weights of $O(n_j)$ vertices. The total time taken for this is $O(m_j + n_j \log n_j)$ time. To prove the correctness of this procedure, we have to show the following:

1. The new dual weights $y_j(\cdot)$ along with the matching $M_j$ form an $R$-feasible matching.
(2) After the Sync procedure, for any vertex $v \in K_j \cup (V_j \cap (A_F \cup B_F))$, $\tilde{y}(v) = y_j(v)$.

We give a proof that these two properties hold after executing Sync in Lemma 6.23 of the appendix. The following lemma establishes properties of the Sync procedure that will later be used for showing the correctness of our algorithm. For its proof, see appendix section 6.7.3.

Lemma 6.13. Consider a compressed feasible matching with dual weights $\tilde{y}(\cdot)$ assigned to every vertex of $V_H$. For any piece $R_j$ and any vertex $v \in V_j$, let $y_j^*(v)$ denote the dual weight prior to executing Sync, and for any edge $(u, v) \in E_j$, let $s^*(u, v)$ be the slack prior to executing Sync. Let $y_j(\cdot)$ denote the dual weights of $V_j$ after this execution. For any edge $(u, v) \in E^H_j$ with a projection $P_{u,v,j} = \langle u = u_0, u_1, \ldots, u_t, u_{t+1} = v \rangle$, suppose $|\tilde{y}(u)| - |y_j^*(u)| \geq \sum_{q=0}^{t} s^*(u_q, u_{q+1})$. Let $P_{s,u,t,j}$ be any shortest path from $s$ to $u_t$ in $R'_j$. Then,

(i) If there exists a shortest path $P_{s,u,t,j}$ in $R'_j$ where $u$ is the second vertex on this path, then after the execution of Sync procedure, for every $1 \leq i \leq t - 1$, $s(u_i, u_{i+1}) = 0$ and $s(u_t, v) \leq |\tilde{y}(v)| - |y_j^*(v)|$,

(ii) Otherwise, there is no shortest path $P_{s,u,t,j}$ in $R'_j$ with $u$ as its second vertex. Consider $u^*$ to be the second vertex of some $P_{s,u,t,j}$ and $u^* \neq u$. Then, $u^* \in (K_j \cup (V_j \cap (A_F \cup B_F)))$, and $|\tilde{y}(u^*)| - |y_j^*(u^*)| > \sum_{(u',v') \in P_{u^*,v,j}} s^*(u', v')$.

Given the correctness of the Sync procedure, we can convert a compressed feasible matching into an $\mathcal{R}$-feasible matching by simply applying the Sync procedure to all the pieces. This will guarantee that the dual weight of any vertex $v \in V$, is the same across all pieces and in $H$. Let $y(v)$ be this dual weight. Every edge $(u, v)$ of the graph $G$ belongs to some piece and therefore satisfies the $\mathcal{R}$-feasibility conditions. Therefore the matching $M$ along with
6.4. Algorithm for Each Scale

the dual weights \( y(\cdot) \) form an \( \mathcal{R} \)-feasible matching. Invocation of \textsc{Sync} on a piece \( \mathcal{R}_j \) takes \( O(m_j + n_j \log n_j) \) time.

**Lemma 6.14.** Given a compressed feasible matching \( M \), we can convert it into an \( \mathcal{R} \)-feasible matching \( \bar{M} \) with a set of dual weights \( y(\cdot) \) in \( O(m + n \log n) \) time.

6.4.3 Second Step of the Algorithm

In this section, given a compressed feasible matching with \( O(n/\sqrt{r}) \) free vertices, we show how to compute a compressed feasible matching with \( \sqrt{n/r} \) unmatched vertices in \( \tilde{O}(mn^{2/5}) \) time. The second step of our algorithm will consist of multiple phases. After \( \sqrt{n/r} \) phases, the algorithm will have a compressed feasible matching with \( O(\sqrt{n}/r^{1/4}) \) unmatched vertices remaining. Finally, the third step of the algorithm matches the remaining vertices by performing \( O(\sqrt{n}/r^{1/4}) \) iterations of a slightly modified version of Hungarian search. Next, we give the details of Step 2. This step will iteratively perform a partial DFS style search through the admissible edges of \( H \). Step 2 consists of multiple phases. In each phase, the algorithm will invoke the \textsc{SearchAndSwitch} procedure on every active vertex \( u \in B^A_H \).

This procedure takes as input a free active vertex \( u \) (boundary or internal) and, using a DFS-style search with dual adjustments to vertices in \( H \), will iteratively find paths or cycles of admissible edges in the compressed residual graph.

When a path or cycle \( P \) in the compressed graph is found, the algorithm invokes the \textsc{Switch} procedure. This procedure projects \( P \) to obtain a path \( \vec{P} \) in \( G \) where \( \vec{P} \) is either an:

- Augmenting path,

- Alternating cycle, or

- Alternating path from a free vertex \( u' \in B_F \) to some matched vertex \( v \in B \).
Note that we refer to $P$ (in $H$) as an alternating path, augmenting path, or alternating cycle based on its projection $\hat{P}$.

The Switch procedure then switches along $\hat{P}$ by setting $M \leftarrow M \oplus \hat{P}$. Since switching along a path changes the residual graph, the Switch procedure updates the compressed graph accordingly. This process continues until at least one of the following holds:

(I) $\lambda_u \ddot{y}(u)$ has increased by $\sqrt{r}$, in step (i) of SearchAndSwitch below,

(II) $u$ is a matched boundary vertex,

(III) $u$ was a free internal vertex $b^A_j$ of $H$, that no longer exists in $H$, i.e., there are no free active internal vertices in $R_j$.

Recall that we define the slack of any edge $(u, v) \in E_H$ as $\phi(\hat{P}_{u,v,j}) - |\ddot{y}(u)| + |\ddot{y}(v)|$. $(u, v)$ is admissible in $H$ if it has at most $\sqrt{r}$ slack. For any vertex $v \in V_H$, let $A_v$ be the set of admissible edges of $H$ going out of $v$. Next, we describe the algorithm.

The procedure will conduct a DFS-style search by growing a path $Q = \langle u_0, u_1, \ldots, u_s \rangle$ in $H$. Here, $u_0 = u$. The algorithm grows $Q$ by conducting a search at $u_s$ as follows:

(i) If $A_{u_s} = \emptyset$, then remove $u_s$ from $Q$, set $\ddot{y}(u_s) \leftarrow \ddot{y}(u_s) + \lambda_{u_s} \sqrt{r}$. If $s > 0$, then continue the search from $u_{s-1}$. Otherwise, if $s = 0$, the procedure terminates since stopping condition (I) is satisfied.

(ii) Otherwise, $A_{u_s} \neq \emptyset$, then find the smallest slack edge in $A_{u_s}$. There could be many edges with the same smallest slack. Among these edges, the algorithm will prefer any edge $(u_s, v)$ such that $v$ is already on the path $Q$. Such a vertex would form a cycle in $H$.

If none of the smallest slack edges lie on the path, then choose an arbitrary edge with the smallest slack. Let the chosen edge be $(u_s, v)$ with $s_H(u_s, v) = \min_{(u_s, v') \in A_{u_s}} s_H(u_s, v')$. Add $v$ to the path $Q$ as vertex $u_{s+1}$. 
• If $u_{s+1} \in Q$, then a cycle $C$ is created. Let $u_x = v$, and let $C = \langle u_x, u_{x+1}, \ldots, u_s, u_x = u_{s+1} \rangle$. Then, set $Q \leftarrow \langle u_0, u_1, \ldots, u_{x-1} \rangle$, call the SWITCH procedure (described below) on $C$, and continue the search from $u_{x-1}$.

• Otherwise $u_{s+1} \notin Q$,
  
  – If any of the following three conditions hold true, then set $P \leftarrow \langle u_0, u_1, \ldots, u_s, u_{s+1} \rangle$ and $Q \leftarrow \{u_0\}$, and invoke SWITCH on $P$.
    * $u_{s+1} \in \mathcal{K} \cap B_H$ and $|\tilde{y}(u_{s+1})| \geq \beta$,
    * $u_{s+1} \in \mathcal{K} \cap A_H$ and $|\tilde{y}(u_{s+1})| \geq \beta + \max_{(t, u_{s+1}) \in E} \delta_{tu_{s+1}}$, or
    * $u_{s+1} \in A_{FH}$.
  
  – If $u_0$ is no longer free (i.e., (II) is satisfied), or $u_0$ no longer exists (i.e., (III) is satisfied), then the SEARCH AND SWITCH procedure terminates.

  – Otherwise, continue the search from $u_{s+1}$.

This completes the search portion of SEARCH AND SWITCH. Next, we will describe the details of SWITCH procedure.

**Switch Procedure**  The SWITCH procedure takes an alternating path, augmenting path or an alternating cycle $P = \langle u_0, \ldots, u_{s+1} \rangle$ in $H$ and computes a path or cycle $\overrightarrow{P}$ in $G$ by projecting every edge of $P$. It also updates the dual weights of every vertex on $\overrightarrow{P}$ so that $\overrightarrow{P}$ consists only of admissible edges. Then, the algorithm sets $M \leftarrow M \oplus \overrightarrow{P}$. This changes the residual graph and its compressed representation. Finally, the procedure updates the compressed graph $H$ to reflect the changes to the underlying residual graph.

(a) For every edge $(u, v) \in P$, mark the piece it belongs to as affected. Let $\mathbb{R}$ be the set of all affected pieces. Execute Sync on every piece $\mathcal{R}_j \in \mathbb{R}$. 
(b) Set a value \( \alpha \leftarrow \tilde{y}(u_0) \). For every \( 0 \leq i \leq s \), set \( \tilde{y}(u_i) \leftarrow \tilde{y}(u_i) + \lambda_i s_H(u_i, u_{i+1}) \); here \( s_H(u_i, u_{i+1}) \) is the slack before the dual weights are updated (i.e., prior to this execution of (b)). Execute \text{SYNC} \ again on every piece \( \mathcal{R}_j \in \mathbb{R} \).

(c) Suppose \((u, v)\) is an edge of piece \( \mathcal{R}_j \). Project \((u, v)\) to obtain the path \( \overrightarrow{P}_{u,v,j} \). This can be done by executing Dijkstra’s algorithm over \( \mathcal{R}_j' \). Next, combine all the projections to obtain a path or cycle \( \overrightarrow{P} \) in the residual graph \( G_M \). We show that this path or cycle is a simple path or cycle consisting only of admissible edges.

(d) If \( u_{s+1} \in \mathcal{A}_H \setminus \mathcal{A}_F^H \) and \( P \) is an alternating path, then \( u_{s+1} \) is a matched vertex. Let \((u_{s+1}, v_{s+1})\) be the edge in the matching \( M \) belonging to the piece \( \mathcal{R}_j \). Execute \text{REDUCESLACK}(v_{s+1}) \). This makes the edge \((u_{s+1}, v_{s+1})\) admissible with respect to \( M_j, y_j(\cdot) \) without violating compressed feasibility. Also, add \((u_{s+1}, v_{s+1})\) to \( \overrightarrow{P} \) and add \( \mathcal{R}_j \) to \( \mathbb{R} \). \( \mathcal{R}_j \) is added to the affected set because the edge \((u_{s+1}, v_{s+1})\) will change to a non-matching edge during (e), which will affect the edges of \( E_j^H \).

(e) Update the matching \( M \) along \( \overrightarrow{P} \) by setting \( M \leftarrow M \oplus \overrightarrow{P} \). By Lemma 6.3 the new matching is \( \mathcal{R} \)-feasible within all affected pieces. In the event that \( \overrightarrow{P} \) was an alternating path to some internal vertex \( v \in \mathcal{B}_j \setminus \mathcal{K}_j \), \( v \) is now an inactive free internal vertex in \( G \). It is possible that the dual weights of the inactive free internal vertices in \( \mathcal{R}_j \) differ. Therefore, call \text{REDUCE}(b^L_j, \beta). \) The residual graph changed during this step. Therefore, call the \text{CONSTRUCT} \ procedure on every affected piece \( \mathcal{R}_j \in \mathbb{R} \) and recompute the edges in \( E_j^H \) along with their costs.

(f) If \( P \) is an alternating path or augmenting path, and \( u_0 \) still exists as a vertex \( b^A_j \in B^F_j \), then execute \text{REDUCE}(u_0, \alpha). \) This effectively resets all dual weights associated with \( u_0 \) to their values prior to step (b). Since the dual weights \( \tilde{y}(\cdot) \) of vertices of \( B^A_j \) differ by at most \( \sqrt{r} \), and \( \alpha \geq 0 \), the preconditions to \text{REDUCE} \ are satisfied.
6.4.4 The Third Step of the Algorithm

After Step 2, the algorithm has a compressed feasible matching with $O(\sqrt{n}/r^{1/4})$ unmatched vertices remaining. This can be converted into an $\mathcal{R}$-feasible matching in $O(m+n \log n)$ time by Lemma 6.14. Next, we describe a procedure for computing an $\mathcal{R}$-optimal matching from this $\mathcal{R}$-feasible matching. In section 6.6, we will discuss further optimizations which lead to an even better running time for Step 3 on planar graphs, based on the results from [6].

**Step 3** Given an $\mathcal{R}$-feasible matching with $O(\sqrt{n}/r^{1/4})$ unmatched vertices remaining, the algorithm can use $O(\sqrt{n}/r^{1/4})$ iterations of Hungarian search to match the remaining vertices, matching one vertex each iteration. Let $G'$ be the graph $G$ with all edges having weight equal to their slack with respect to the $\mathcal{R}$-feasible matching. In each iteration of Step 3, the algorithm executes Dijkstra’s algorithm on $G'$ from the vertices of $B_F$ to the vertices of $A_F$ in order to find the minimum total slack augmenting path. For each vertex $v$ in $G$, let $\ell_v$ be the distance assigned by Dijkstra’s algorithm. Let $P$ be a shortest augmenting path found, and let $\ell_{\text{max}}$ be the distance to the free vertex of $A_F$ in $P$. Then for each vertex with $\ell_v \leq \ell_{\text{max}}$, the algorithm sets $y(v) \leftarrow y(v) + \lambda(\ell_{\text{max}} - \ell_v)$. This dual weight change ensures all edges of $P$ are admissible, while also preserving $\mathcal{R}$-feasibility. The algorithm then sets $M \leftarrow M \oplus P$, which increases the matching size by one while preserving $\mathcal{R}$-feasibility.

6.5 Analysis of Algorithm

Step 1 of the algorithm computes a $\mathcal{R}$-feasible matching $M$ with all but $O(n/\sqrt{r})$ unmatched vertices. This matching is then converted into a compressed feasible matching. Step 2 iteratively computes alternating paths, augmenting paths and cycles and switches the edges along them. While doing so, it maintains a compressed feasible matching. In $O(n \sqrt{n/\sqrt{r}} + mr)$
time, we obtain a compressed feasible matching with no more than $O\left(\sqrt{\frac{n}{\sqrt{r}}}\right)$ unmatched vertices. Step 3 computes the remaining augmenting paths iteratively by doing a simple Hungarian search in $O(m\sqrt{\frac{n}{\sqrt{r}}})$ time. For $r = n^{2/5}$, the running time of the algorithm is $O(mn^{2/5})$. To complete the analysis, we need to show the correctness and efficiency of Step 2 of the algorithm. We prove the correctness of Step 2 in Section 6.5.1 and the efficiency of Step 2 in Section 6.5.2.

For any path $\vec{P}'$ in $G$, let $s(\vec{P}') = \sum_{u',v' \in \vec{P}'} s(u', v')$.

### 6.5.1 Correctness

**Overview of the Proof** We show that the paths computed in the second step of the algorithm satisfy certain properties (P1)–(P3). Using these properties, we show that these paths have simple (see Lemma 6.15) and admissible (see Lemma 6.16) projections. Recollect that switching the edges on an admissible path maintains $R$-feasibility, by Lemma 6.3. Using this, we show that our algorithm maintains compressed feasibility.

**Properties of Paths** Our algorithm computes paths and cycles in $H$ that satisfy three properties stated below. Let $P = \langle u_1, u_2, \ldots, u_t \rangle$ be any path or cycle in $H$ such that,

(P1) For any edge $(u_i, u_{i+1})$ on $P$, $s_H(u_i, u_{i+1}) \leq \sqrt{r}$, i.e., $(u_i, u_{i+1})$ is admissible,

(P2) For any edge $(u_i, u_{i+1})$ on $P$, $s_H(u_i, u_{i+1}) = \min_{(u_i, u') \in E_H} s_H(u_i, u')$,

(P3) For any $u_i, u_k \in P$ such that $k < i$, $s_H(u_i, u_k) > s_H(u_i, u_{i+1})$.

The next two lemmas show that when the $\text{SWITCH}$ procedure is called on any path $P$ that satisfies (P1)–(P3), its projection $\vec{P}$ is simple and consists of only admissible edges.
Lemma 6.15. Given a compressed feasible matching, let \( P = (u_1, \ldots, u_t) \) be any (not necessarily simple) path in \( H \) that satisfies properties \((P2)\) and \((P3)\). Then, for any two edges \((u_i, u_{i+1})\) and \((u_k, u_{k+1})\) on \( P \) with \( i < k \) that belong to piece \( E^H_j \), their projections \( \overrightarrow{P}_{u_i,u_{i+1},j} \) and \( \overrightarrow{P}_{u_k,u_{k+1},j} \) are interior-disjoint.

Proof. Without loss of generality, assume that the dual weights \( \tilde{y}(\cdot) \) and \( y_j(\cdot) \) are synchronized. For the sake of contradiction let \( \overrightarrow{P}_{u_i,u_{i+1},j} \) and \( \overrightarrow{P}_{u_k,u_{k+1},j} \) intersect in the interior at a vertex \( x \) in some piece \( R_j \). Since \( x \) is common to both the projections, it immediately follows that there is a path from \( u_i \) to \( u_{k+1} \) and a path from \( u_k \) to \( u_{i+1} \), both passing through \( x \). This implies

\[
s_H(u_i, u_{i+1}) + s_H(u_k, u_{k+1}) \geq s_H(u_i, u_{k+1}) + s_H(u_k, u_{i+1}). \tag{6.30}
\]

From property \((P2)\), we have that \( s_H(u_i, u_{i+1}) \leq s_H(u_i, u_{k+1}) \) and \( s_H(u_k, u_{k+1}) \leq s_H(u_k, u_{i+1}) \). This along with (6.30) implies that \( s_H(u_k, u_{k+1}) = s_H(u_k, u_{i+1}) \) contradicting \((P3)\) since \( i < k \). \( \square \)

The following is a straight-forward corollary of Lemma 6.15.

Corollary 3. Given a compressed feasible matching, let \( P = (u_1, \ldots, u_t) \) be a simple path (resp. simple cycle) in \( H \) that satisfies properties \((P2)\) and \((P3)\). Then, the projection \( \overrightarrow{P} \) of \( P \) is a simple path (resp. cycle).

Let \( P \) be a path (or cycle) that satisfies \((P2)\), Consider an execution of Switch on path \( P \) and let \( \overrightarrow{P} \) be the projection computed in step (c) in Switch. All edges of \( \overrightarrow{P} \) are admissible.

Lemma 6.16. Let \( P \) be the path (or cycle) that is projected during step (c) of Switch. Assume that \( P \) satisfies property \((P1)\) and \((P2)\) at the beginning of Switch. Then for every edge \((u,v)\) on \( P \) with projection \( P_{u,v,j} \), every edge of \( \overrightarrow{P}_{u,v,j} \) is admissible.
Proof. At the end of step (b) of the Switch procedure, Sync is called on the piece containing \((u, v)\). In the Sync procedure, recollect that we add a new vertex \(s\) and connect it with all the boundary vertices to create a graph \(R'_{j}\). For this Sync procedure (of step (b)), we use the notations from Lemma 6.13. Recollect that, due to the execution of Sync procedure in step (a), \(y^*_j(v) = \tilde{y}(v)\), for all \(v \in V^H_j\). Step (b) will only increase the \(\tilde{y}(\cdot)\) for vertices along \(P\). Let \(u_t\) be the vertex that appears before \(v\) in \(\overrightarrow{P}_{u,v,j}\).

Suppose there is no shortest path from \(s\) to \(u_t\) in \(R'_{j}\) with \(u\) as the second vertex, then let \(u^*\) be the second vertex on some shortest path \(\overrightarrow{P}_{s,u_t,j}\). From Lemma 6.13 (ii), it follows that \(|\tilde{y}(u^*)| - |y^*_j(u^*)| > s_H(u^*, v)\), i.e., the change in \(\tilde{y}(u^*)\) in Step (b) of the Switch procedure is greater than \(s_H(u^*, v)\). Since step (b) updates the dual weight \(\tilde{y}(u^*)\), \(u^*\) is on the path \(P\) and let \(y\) be the vertex that appears after \(u^*\) on \(P\). The change in dual weight \(\tilde{y}(u^*)\) is exactly \(s_H(u^*, y)\). Therefore, \(s_H(u^*, y) > s_H(u^*, v)\) contradicting (P2). Therefore, the vertex \(u\) must be the second vertex on some shortest path from \(s\) to \(u_t\) implying Lemma 6.13 (i) holds.

From Lemma 6.13 (i), it follows that every edge on \(\overrightarrow{P}_{u,v,j}\) except the last edge has zero slack. Moreover the slack on the last edge is \(|\tilde{y}(v)| - |y^*_j(v)|\) which is less than or equal to \(\sqrt{r}\) (by (P1)). Since \(v\) is a boundary vertex, the last edge \((u_t, v)\) is also an admissible edge.

The second step of the algorithm maintains the following invariants.

(A) Let \(Q\) be the search path in the SearchAndSwitch procedure. Then path properties (P1), (P2), (P3) hold for \(Q\).

(B) After any step of SearchAndSwitch or Switch, the matching \(M\) and the sets of dual weights \(\bigcup_{R_j} y_j(\cdot)\) and \(\tilde{y}(\cdot)\) form a compressed feasible matching.

Given Invariant (A), it is easy to show that any projected path (or cycle) \(\overrightarrow{P}\) in step (e) is
both simple and admissible. From Corollary 3, the projection of \( Q \) is simple. In step (d), a single matching edge \((a, b)\) may be added to \( \overrightarrow{P} \). However, REDUCESLACK is called to ensure that this edge has 0 slack.

**Lemma 6.17.** Let \( P \) be a path or cycle in \( H \) sent as input to SWITCH. Let \( \overrightarrow{P} \) be the path or cycle that is a projection of \( P \) prior to step (e) of SWITCH. Then \( \overrightarrow{P} \) is a simple path or cycle consisting of admissible edges.

Next, we discuss proof of Invariant (A). The SEARCHANDSWITCH procedure adds the smallest slack admissible edge going out of the last vertex on \( Q \). By construction, the edge added satisfies (P1)–(P3). During the execution of the SEARCHANDSWITCH procedure only \( \tilde{y}(\cdot) \) values are modified for vertices from which the search backtracks. For any such vertex \( v \) from which the search backtracks, the algorithm sets \( \tilde{y}(v) \leftarrow \tilde{y}(v) + \lambda_v \sqrt{r} \). This increases the magnitude of the dual weight \( \tilde{y}(v) \). So, for any vertex \( u \in Q \), the slack \( s_H(u, v) \) only increases. Since \( v \) is not on the path, (P1)–(P3) continue to hold.

During the execution of SWITCH procedure for an alternating path or an augmenting path, \( Q \) is set to \( \emptyset \). Therefore, (P1), (P2) and (P3) hold trivially. In the case of an alternating cycle, however, \( Q \) may contain edges. Note that the magnitude of \( \tilde{y}(\cdot) \) values increase for vertices not on \( Q \) in step (b) of SWITCH procedure. Since the magnitude only increases, (P1)–(P3) holds. Since Step (d) is not executed for a cycle, the only other step where slack on the edges of \( H \) are changed are in Step (e). In the following lemma, we show that, for any such \( Q \), (P1)–(P3) hold after the execution of Step (e) of the SWITCH procedure.

**Lemma 6.18.** Assume that the path \( Q \) satisfies properties (P1), (P2) and (P3) prior to executing step (e) of SWITCH with a cycle \( C \) as input. Then (P1), (P2) and (P3) hold for \( Q \) after step (e).

*Proof.* Consider the case during the execution of SEARCHANDSWITCH right after \( v \) is added
to \( Q \) as \( u_{s+1} \). Since before adding \( u_{s+1} \) to \( Q \), \( u_{s+1} \) was already on \( Q \), a cycle is created. By Lemma 6.15, the projection of \( Q \) after the addition of \( u_{s+1} \) has no self intersection except at \( u_{s+1} \). We then remove the cycle \( C \) from \( Q \) and call the Switch procedure on \( C \). It follows that the projection of \( C \) and \( Q \) (after the removal of cycle as described in (ii) of SearchAndSwitch) has no intersections.

Let \((u, v)\) be an edge of \( Q \) before execution of step (e). Let \( \vec{P}_{u,v,j} \) be the projection of \((u, v)\) prior to step (e), and let \( \vec{P}'_{u,v,j} \) be the projection after step (e). For every \((y, z)\) \in \( C \), from the discussion above, we have that \( \vec{P}_{u,v,j} \cap \vec{P}_{y,z,j} = \emptyset \). During step (e), only edges of \( \vec{P}_{y,z,j} \) change direction and therefore, \( \vec{P}_{u,v,j} \) continues to be a directed path after step (e). Therefore, \( s(\vec{P}'_{u,v,j}) \leq s(\vec{P}_{u,v,j}) \) and (P1) holds.

Next, we show that for any vertex \( v' \in V^H_j \) (possibly \( v' = v \)), any new projection \( \vec{P}'_{u,v',j} \) created after step (e) has \( s(\vec{P}'_{u,v',j}) > s(\vec{P}_{u,v,j}) \) implying (P2) and (P3) hold. Since \( \vec{P}'_{u,v',j} \) was created from switching along \( C \), \( \vec{P}'_{u,v',j} \) must intersect with some projection \( \vec{P}_{y,z,j} \) of an edge \((y, z)\) in \( C \). Step (b) of the Switch procedure increases the magnitude of the dual weight of \( z \) say by \( \Delta_z \) and since \((u, v)\) was the smallest slack edge out of \( u \), it follows that

\[
s_H(u, v) \leq s_H(u, z) - \Delta_z.
\]

Let \( \vec{P}_{z,y,j} \) be the path obtained by reversing the edges of \( \vec{P}_{y,z,j} \). Let \((x, x')\) be the first edge in the intersection of \( \vec{P}'_{u,v',j} \) and \( \vec{P}_{z,y,j} \) as we walk along \( \vec{P}_{u,v',j} \) (any two alternating paths that intersect will intersect along at least one edge). Right before switching the edges of the cycle, from Lemma 6.16, it follows that every edge on \( \vec{P}_{y,z,j} \) is zero slack except for the edge incident on \( z \) which has a slack of \( \Delta_z \). Consequently, \( \vec{P}_{u,x,j} \) has a slack of \( s_H(u, v) \), i.e.,

\[
s(\vec{P}_{u,x,j}) \geq s_H(u, v).
\]
Since \((x, x')\) is on \(\overrightarrow{P}_{y,z,j}\), from Lemma 6.3, \((x, x')\) has a positive slack. Therefore, \(s(\overrightarrow{P}_{u,x',j}) \geq s(\overrightarrow{P}_{u,x',j}) > s_H(u, v),\) as desired.

Finally, we show Invariant (B) and establish that the compressed feasibility conditions hold throughout the second step. Without loss of generality, let us assume that the condition holds at the start of an execution of SEARCHANDSWITCH procedure. We will show that this execution of SEARCHANDSWITCH procedure and the subsequent execution of SWITCH does not violate compressed feasibility conditions (a)–(e).

SEARCHANDSWITCH only changes the dual weights in case (i) of SEARCHANDSWITCH, where the procedure sets \(\tilde{y}(u_s) \leftarrow \tilde{y}(u_s) + \lambda u_s \sqrt{r}.\) Since this operation only increases the magnitude of \(\tilde{y}(u_s),\) conditions (a), (b), and (e) of compressed feasibility are satisfied. Note that condition (b) also requires the dual weights \(\tilde{y}(\cdot)\) to be at most \(\sqrt{r}\) apart. However, this is satisfied because a free internal vertex executes (i) exactly once per phase. Also, note that at the beginning of each phase, all vertices \(v \in B_H^A\) have the same dual weight. Condition (c) is unaffected. Finally, observe that, since the dual weight change only occurs when there are no admissible edges outgoing from \(u_s,\) condition (d) continues to hold.

Suppose \(P\) is the path or cycle sent to the SWITCH procedure and suppose \(\overrightarrow{P}\) is its projection. Before projecting \(P,\) the dual weight \(\tilde{y}(v)\) for every vertex \(v \in P\) is increased by the slack of the edge \((v, v')\) in \(P.\) From (P2), \((v, v')\) is the smallest slack edge out of \(v\) and therefore the increase \(\tilde{y}(v) \leftarrow \tilde{y}(v) + \lambda v s(v, v')\) reduces the slack of \((v, v')\) to 0 and all other edges continue to have a non-negative slack. Therefore, the change does not violate \(H\)-feasibility and also preserves (a), (b) and (e). The projection computed by SWITCH is a simple path or cycle consisting only of admissible edges. Switching edges \((M \leftarrow M \oplus \overrightarrow{P})\) on this path does not violate \(R\)-feasibility (Lemma 6.3) of any of the affected pieces. So, after switching the edges, the new matching in each of the affected pieces \(R_j,\) along with the dual weights
Chapter 6. Minimum-Cost Matching on Minor-Free Graphs

$y_j(\cdot)$ form an $\mathcal{R}$-feasible matching. The Construct procedure will recompute edges of $H$ which, from Lemma 6.12 satisfies $H$-feasibility. As discussed in Section 6.4, Sync, Reduce and ReduceSlack preserve compressed feasibility as well.

Therefore, our algorithm iteratively matches vertices while maintaining compressed feasibility. In the following, we discuss the efficiency of our algorithm.

### 6.5.2 Efficiency of the Second Step

Step 2 of our algorithm invokes SearchAndSwitch on free internal vertices of $B_F^A$. This procedure computes cycles and paths in $H$ and passes them to the Switch procedure. Let $\langle P_1, P_2, ..., P_N \rangle$ be the sequence of paths and cycles generated by the second step of the algorithm. These paths and cycles are sorted in the order in which they are computed in Step 2, with $P_i$ being the $i$th such path or cycle. Note that the Switch procedure is executed for each such $P_i$. Let $\overrightarrow{P}_i$ be the projection of $P_i$ as computed by the Switch procedure. Let $M^{(0)}$ be the matching at the start of Step 2. Then $M^{(i)} \leftarrow M^{(i-1)} \oplus \overrightarrow{P}_i$. The operations conducted by the algorithm after the execution of the Switch procedure on $P_{i-1}$ until the end of the execution of the Switch procedure on $P_i$ is referred to as the $i$th iteration of the algorithm. Let $B^i_F$ denote the free vertices of $B$ at the start of iteration $i$. Every compressed feasible matching can be converted into an $\mathcal{R}$-feasible matching by applying Sync to all pieces. For the proof, let $y'(\cdot)$ denote the dual weights of this $\mathcal{R}$-feasible matching at the start of iteration $i$. For any path $\overrightarrow{P}'$ in $G$, let $s(\overrightarrow{P}') = \sum_{u', v' \in \overrightarrow{P}'}, s(u', v')$.

**Efficiency of SearchAndSwitch.** To bound the time taken by SearchAndSwitch procedure, it suffices if we bound the total time taken to find the smallest slack edge from $u_s$ during all executions of SearchAndSwitch in Step 2. We store the edges of $H$ as follows: For each piece $\mathcal{R}_j$, the Construct procedure will store, for every boundary vertex $u \in \mathcal{K}_j$,
all the outgoing edges from $u$ in $E^H_{j}$ sorted by their slacks. Since $u$ could belong to several pieces, we store a different sorted order for each piece and maintain a heap of the smallest admissible edge from each piece. To find the smallest slack edge from $u$, we simply retrieve the smallest element from the heap of $u$, say $(u, v)$. There are two possibilities. Either $(u, v)$ is the smallest slack edge or the slack $s_H(u, v)$ has changed since it was added to the heap. We refer to the second case as false positive. For our analysis, we bound the total number of false positives by $O(n\beta/\sqrt{r} + nr)$.

Note that a change in the dual weight $\tilde{y}(u)$ does not change this relative ordering (in terms of slack) of edges going out of $u$ in $H$. However, for any edge $(u, v)$ in $H$, a change in the dual weight $\tilde{y}(v)$ will increase the slack $s_H(u, v)$ and may push $(u, v)$ down in this sorted order. Since, we do not actively update the position of $(u, v)$ at $u$, a false positive arises. While looking for the smallest slack edge from $u$, suppose we find a false positive edge $(u, v)$, we simply reinsert this edge in the correct location in the sorted order and update the heap accordingly. A false positive can arise due to the change in dual weight $\tilde{y}(v)$ in two places: (i) a search backtracked from $v$ causing the magnitude of the dual weight $\tilde{y}(v)$ to increase by $\sqrt{r}$; and (ii) $v$ lies on some path $P_i$ and the Switch procedure updated $\tilde{y}(v)$ prior to switching the edges.

For false positives of type (i), we charge it to the increase of $\tilde{y}(v)$ by $\sqrt{r}$. Since $u$ can undergo a dual weight change of at most $O((\beta + \max_{v' \in N(u)} \delta_{uv'})/\sqrt{r})$, the total number of false positives is at most $O(d(\beta + \max_{v' \in N(u)} \delta_{uv'})/\sqrt{r})$; here $d$ is the in-degree of $u$ in $H$ during the execution of Step 2. We can upper bound $\delta_{uv}$ by setting $O(mn/\sqrt{r}) = O(r^2n) = O(v^{3/2})$. From this and the fact that the total in-degrees of all vertices is bounded by $O(n)$ we get the total number of false positives of type (i) is at most $O(n\beta/\sqrt{r} + nr) = O(n^{3/2}/\sqrt{r} + nr)$.

For false positives of type (ii), $v$ lies on some path $P_i$. In this case the Switch procedure (step (b)) will update $\tilde{y}(v)$. From Lemma 6.19, if $v \in B_H$, then $(u, v)$ is in an affected
piece and so the \textsc{Construct} procedure will immediately update \( s_H(u, v) \). Therefore, \((u, v)\) cannot lead to a false positive and \( v \) cannot be a vertex of \( B_H \). If \( v \in A_H \), from Lemma 6.19, either \( v \) is never visited again by \textsc{SearchAndSwitch} or during the next visit of \( v \), its dual weight increases by \( \sqrt{r} \). We charge this false positive to the increase in dual weight of \( \tilde{y}(v) \).

Using similar arguments to type (i), we can bound the total number of false positives by \( O(n/\sqrt{r} + n\beta/\sqrt{r}) \). Combining this with the total time for all the search operations during the second step gives \( \tilde{O}(n^{3/2} r^{1/4}) \).

\textbf{Lemma 6.19.} For any vertex \( v \in P_i \), suppose \( v \) is a boundary vertex. consider the compressed residual graphs \( H \) before and after the execution of \textsc{Switch} procedure on \( P_i \).

- Suppose \( v \in B_H \), then every edge of \( H \) (before or after execution of \textsc{Switch}) directed towards \( v \) is in an affected piece.

- Suppose \( v \in A_H \), then either \( v \) is never visited again by \textsc{SearchAndSwitch} procedure in Step 2 of the algorithm or during the next visit of \( v \), the dual weight \( \tilde{y}(v) \) increases by \( \sqrt{r} \).

\textit{Proof.} Since \( v \) is on \( P_i \), the match of \( v \) before and after the \textsc{Switch} are both in affected pieces. If \( v \in B_H \), then the in-degree of \( v \) in the residual graph \( \overrightarrow{G}_M \) is 1 and all edges of \( H \) directed towards \( v \) before (resp. after) the execution of \textsc{Switch} contain the matching edge of \( u \) before (resp. after) the execution of \textsc{Switch}.

Suppose \( v \in A_H \), then let \((v, v')\) be the matching edge after the execution of \textsc{Switch}. Since \((v, v')\) was admissible prior to the execution of \textsc{Switch}, from Lemma 6.3, after the execution of \textsc{Switch}, the slack on \((v, v')\) is at least \( \delta_{vv'} \geq 2\sqrt{r} \). Therefore, every edge going out of \( v \) has a slack of at least \( 2\sqrt{r} \), implying that there are no admissible edges going out of \( v \). Therefore, if the \textsc{SearchAndSwitch} procedure visits \( v \) again, it will backtrack and \( \tilde{y}(v) \) will increase by \( \sqrt{r} \). \hfill \Box
The Switch procedure synchronizes, projects, augments and then re-constructs the affected pieces. Let $\Gamma$ be the set of all affected pieces. Then, the time taken by switch is upper bounded by the time taken to re-construct the edges of $H$ for every affected piece. The following sequences of Lemma bound the time taken by the Switch procedure.

**Lemma 6.20.** Given a compressed-feasible matching before iteration $i$ of Step 2,

$$|B^i_F|\Delta_i \leq O(n).$$  \hfill (6.31)

*Here, $\Delta_i$ is the minimum dual weight among all free vertices of $B$."

**Proof.** Using the Sync procedure, we can create an $R$-feasible matching $M^{(i-1)}$, $y^i(\cdot)$ from the compressed feasible matching. From compressed feasibility, we have that for every free vertex $a \in A^i_F$, $y(a) = 0$. Consider some optimal matching $M^*$. $M^{(i-1)} \oplus M^*$ forms $n - i$ augmenting paths and alternating cycles. Let $C_\oplus$ be the set of cycles in $M^{(i-1)} \oplus M^*$ and let $P_\oplus$ be the set of augmenting paths in $M^{(i-1)} \oplus M^*$.

From (6.15), $\phi(M^{(i-1)} \oplus M^*) = c(M^{(i-1)}) - c(M^*) + \sum_{(u,v) \in M^* \oplus M^{(i-1)}} \delta_{uv}$. Cost of the optimal matching is $O(n)$ and using the arguments of (6.24), $\sum_{(u,v) \in M^* \oplus M^{(i-1)}} \delta_{uv} = O(n)$. Since the dual weights of free vertices of $A$ are 0, from Lemma 6.6, the cost of $M^{(i-1)}$ is also $O(n)$. Therefore,

$$\sum_{P \in C_\oplus \cup P_\oplus} \phi(P) \leq O(n).$$  \hfill (6.32)

Each augmenting path in $P_\oplus$ is a path between a free vertex $b$ of $B$ to a free vertex $a$ of $A$. From properties of compressed feasibility, we know that $y^i(b) \geq \Delta_i$ and $y^i(a) = 0$. Plugging this in (6.18), we get (6.31).

After Step 2, $\Delta$ is at least $\beta$. Therefore, the number of unmatched vertices is at most
\[ O(n/\beta) = \sqrt{n/r^{1/4}}. \] As a corollary, we can show the following:

**Corollary 4.** Recollect that \( \langle P_1, \ldots, P_N \rangle \) are the set of paths and cycles computed by Step 2 of our algorithm and \( \vec{P}_i \) is the projection of \( P_i \). Let \( B^i_F \) be the free vertices and let \( y_i(\cdot) \) denote the dual weights before switching along \( \vec{P}_i \). Define \( \Delta_i = \min_{v \in B^i_F} y_i(v) \). Let \( \kappa_i = 1 \) if \( \vec{P}_i \) is an augmenting path and 0 otherwise. Then \( \sum_{i=1}^N \kappa_i \Delta_i = O(n \log n) \).

**Proof.** Suppose \( \vec{P}_i \) is an augmenting path. From equation (6.31), we have that \( \Delta_i = O(n)/|B^i_F| \). \( |B^i_F| = n - i + 1 \). After augmenting along \( \vec{P}_i \), the number of free vertices reduce by 1 and summing over all \( i \) when \( \kappa_i \) is 1, yields a harmonic series in the denominator. Therefore, \( \sum_{i=1}^N \kappa_i \Delta_i = O(n \log n) \). \( \square \)

**Lemma 6.21.**

\[ \sum_{i=1}^N \sum_{(u,v) \in \vec{P}_i} \delta_{uv} = O(n \log n). \] (6.33)

**Proof.** From Lemma 6.17, all edges on any projection \( \vec{P}_i \) are admissible.

Suppose \( \vec{P}_i \) is an alternating cycle consisting of admissible edges and let \( u \) be any vertex on \( \vec{P}_i \). Then from Lemma 6.4,

\[ 0 = \lambda_u y^i(u) - \lambda_u y^i(u) \geq c(M \oplus \vec{P}_i) - c(M) + \sum_{(p,q) \in \vec{P}_i} \delta_{pq}/2. \]

Suppose \( \vec{P}_i \) is an alternating path and let \( u \) be the first vertex and \( v \) be the last vertex of \( \vec{P}_i \). Then, we know that \( \lambda_i y^i(v) > \beta \) and \( \lambda_i y^i(u) < \beta \). Therefore,

\[ 0 > \lambda_u y^i(u) - \lambda_u y^i(v) \geq c(M \oplus \vec{P}_i) - c(M) + \sum_{(p,q) \in \vec{P}_i} \delta_{pq}/2. \]

Suppose, \( \vec{P}_i \) is an augmenting path with \( u \) as its first vertex and \( v \) as its last vertex. Then,
from Lemma 6.4 and the fact that the dual weight of \( y^i(v) = 0 \) and \( y^i(u) \leq \Delta_i + \sqrt{r} \), we have

\[
\Delta_i + \sqrt{r} > \lambda_u y^i(u) - \lambda_v y^i(v) \geq c(M \oplus \overrightarrow{P_i}) - c(M) + \sum_{(p,q) \in \overrightarrow{P_i}} \delta_{pq}/2.
\]

Let \( \kappa_i \) be 1 if \( \overrightarrow{P_i} \) is an augmenting path and 0 otherwise. Adding over all \( 1 \leq i \leq N \) and since there are at most \( n/\sqrt{r} \) augmenting paths, we immediately get

\[
\sum_{i=1}^{N} \kappa_i \Delta_i + \sqrt{r} \frac{n}{\sqrt{r}} \geq c(M^{(N)}) - c(M^{(0)}) + \sum_{i=1}^{N} \sum_{(p,q) \in \overrightarrow{P_i}} \delta_{pq}/2.
\]

From Corollary 4 and the fact that \( c(M^{(0)}) \) and \( c(M^{(N)}) \) is \( O(n) \), the lemma follows.

**Corollary 5.**

\[
\sum_{i=1}^{N} |P_i| = O((n/\sqrt{r}) \log n). \tag{6.34}
\]

**Proof.** By (6.33), the total \( \delta \) of all projection edges is \( O(n \log n) \). Each boundary edge has a \( \delta \) of at least \( 2\sqrt{r} \) therefore, there can be at most \( O(n/\sqrt{r}) \) boundary vertices in the projections. Every edge of \( H \) has at least one boundary vertex, except the edges from vertices of \( B_H^F \) to vertices of \( A_H^F \). However, there can be at most \( O(n/\sqrt{r}) \) such edges used, since each such edge corresponds to an augmenting path, and there are only \( O(n/\sqrt{r}) \) free vertices at the start of Step 2.

**Lemma 6.22.** The total time taken for all calls to **SYNC**, all projections, and all calls to **CONSTRUCT** during Step 2 is \( O(mr \log m \log^2 n) \).

**Proof.** Other than a single call to **CONSTRUCT** per piece at the beginning of the algorithm, and a single call to **SYNC** per piece at the end of the algorithm, **SYNC**, **CONSTRUCT**, and projections only occur as part of the **SWITCH** procedure, once per affected piece. Out of all these procedures, the time taken for **CONSTRUCT** dominates with total time, taking time
$O(\sqrt{r}(m_j + n_j \log n_j))$ per piece, so bounding the time for \textsc{Construct} is sufficient. To account for different piece sizes, we first divide the pieces into $O(\log m)$ groups, where the $g$th group contains pieces $R_j$ with $2^g \leq m_j < 2^{g+1}$. Since there are at most $m$ edges in total, the $g$th group can contain at most $O(m/2^g)$ pieces. We will show that the total work done for each group over all calls to \textsc{Construct} is $O(mr \log^2 n)$.

First, consider any group where $2^g = O(\sqrt{r})$. By Corollary 5, the maximum number of affected pieces for $g$ is $O(n/\sqrt{r}) \log n$. Since the number of edges in each piece of group $g$ is $O(2^g)$, the \textsc{Construct} time is $O(r \log n)$ per piece, and the total time for $g$ is $O(n \sqrt{r} \log^2 n)$.

Next, consider any group $g$ containing pieces with number of edges much greater than $2^g$. Then for each piece $R_j$ in $g$, each boundary edge $(u, v)$ in $R_j$ has $\delta_{uv} \geq \frac{2^g n}{m \sqrt{r}}$. By Lemma 6.21, $\sum_{P \in \mathcal{P} \cup \mathcal{C} \cup \mathcal{Q}} \sum_{(u, v) \in P} \delta_{uv} = O(n \log n)$. Therefore, the number of times the pieces of $g$ are affected is $O(\frac{m \sqrt{r} \log n}{2^g})$. The time taken for each execution of \textsc{Construct} on a piece of group $g$ is $O(2^g \sqrt{r} \log n)$. Therefore, the total time taken for \textsc{Construct} over all pieces of $g$ is $O(mr \log^2 n)$. Summing over all groups gives a total time for \textsc{Construct} during Step 2 for all pieces as $O(mr \log m \log^2 n) = O(mr \log^3 n)$.

Combining this with the total time for all the search operations during the second step gives $\tilde{O}(mr \log^3 n + \frac{n^{3/2}}{\sqrt{r}})$.

**Efficiency of Steps 1 and 3** The first step of the algorithm executes $O(\sqrt{r})$ iterations of Gabow and Tarjan’s algorithm on the entire graph. This takes $O(m \sqrt{r})$ time. Note that the time for the first step is dominated by the time for the second step.

After the second step, the algorithm has a compressed feasible matching with $O(n/\beta)$ unmatched vertices remaining. This is then converted into an $\mathcal{R}$-feasible matching in $O(m + n \log n)$ time by Lemma 6.14. The remaining $O(\sqrt{n}/r^{1/4})$ vertices are then matched.
one at a time by performing iterations of Hungarian search. Each iteration takes \( O(m \log n) \) time, giving a total complexity of \( O(m \sqrt{n}/r^{1/4} \log n) \) for the third step.

Combining the times taken for the first, second, and third steps of the algorithm gives \( O(mr \log^3 n + m \sqrt{n}/r^{1/4} \log n) \). Setting \( r = n^{2/5} \) gives a total complexity of \( \tilde{O}(mn^{2/5}) \). This is the complexity for a scale of the algorithm. Since there are \( O(\log(nC)) \) scales, the total complexity is \( \tilde{O}(mn^{2/5} \log(nC)) \).

**Extension to Minimum-Cost Maximum Cardinality Matching** The algorithm described thus far computes a perfect matching. However, we can use the following technique, described by Gabow and Tarjan [18], to reduce the perfect weighted matching problem to the maximum weighted matching problem. The technique makes a copy of the graph \( G \) (let this copy be \( G' \)), and, for every vertex \( v \in V(G) \), connects \( v \) to its counterpart \( v' \in V(G') \) by an edge of large cost. This cost could be, for example, the total of all edge costs in the graph plus 1. The new graph has a perfect matching, but a minimum cost perfect matching on \( G' \) corresponds to a minimum cost maximum matching on \( G \). Furthermore, an \( r \)-clustering of \( G \) can also be used as an \( r \)-clustering in the new graph \( G' \). We note that while the technique preserves the \( r \)-clustering property, it does not preserve planarity. Therefore, this reduction technique does not directly extend to the planar graph matching algorithm described in Section 6.6.

**6.5.3 Regarding \( r \)-Clusterings in \( K_h \)-Minor Free Graphs**

Using the result of Wulff-Nilsen [59], one can obtain an \( r \)-clustering for \( K_h \)-minor free graphs. The total number of boundary vertices in their definition is \( \tilde{O}(n/\sqrt{r}) \) instead of \( O(n/\sqrt{r}) \). Similarly, the number of boundary vertices per piece is \( \tilde{O}(\sqrt{r}) \) instead of \( O(\sqrt{r}) \). This increases the sizes of both the vertex and edge sets of \( H \) by a \( \text{poly}(\log n) \) term. To handle
the increase in the size of $H$, our algorithm reduces the error $\delta_{uv}$ on each edge to by a poly($\log n$) factor so that the product of $\delta_{uv}$ and the number of boundary vertices is $O(n)$, which guarantees that the optimal solution at the start of each scale is $O(n)$. For constant $h$, we can set $\delta_{uv} = O(\sqrt{r}/\text{poly}(\log n))$. In SEARCHANDSWITCH, instead of raising the dual weights by $\sqrt{r}$, we raise it by $O(\sqrt{r}/\text{poly}(\log n))$. The convergence rate consequently slows down by a poly($\log n$) factor, with the algorithm taking $\tilde{O}(\sqrt{\frac{n}{\sqrt{r}}})$ phases during the second step. Furthermore, from the efficiency discussion of SEARCHANDSWITCH, the search takes $O(|E_H| + |V_H|) \times \tilde{O}(\sqrt{\frac{n}{\sqrt{r}}}) = \tilde{O}(n \sqrt{\frac{n}{\sqrt{r}}})$ time (Lemma 6.8). From Lemma 6.21, we have that $\sum_{i=1}^{N} \sum_{(u,v) \in P_i} \delta_{uv} = O(n \log n)$. Therefore, we get $(\sqrt{r}/\text{poly}(\log n)) \times \sum_{i=1}^{N} |P_i| = \tilde{O}(n/\sqrt{r})$, and the execution time is $\tilde{O}(n \sqrt{\frac{n}{\sqrt{r}}} + nr)$, which is $\tilde{O}(n^{7/5})$ for $r = n^{2/5}$.

### 6.6 Planar Graph Matching Algorithm

In this section, we give an improved version of the algorithm described in Section 6.4 for planar graphs. The algorithm of [6] uses known results in planar shortest path computation to execute Hungarian search faster, while the algorithm described in Section 6.4.3 uses a Gabow-Tarjan style algorithm to match potentially many vertices each phase, leading to fewer phases being executed. By combining these two approaches, we get an algorithm that both executes fewer phases, and executes each phase efficiently, leading to an $\tilde{O}(n^{6/5} \log nC')$ algorithm.

The speedup for each phase comes from two main prior results in planar shortest paths data structures that were also used in [6]. The first result is a multiple source shortest paths (MSSP) data structure by Klein [27], which can be used in the CONSTRUCT procedure to compute the edges between all pairs of boundary vertices in a piece in $O(r \log r)$ time instead of $O(r^{3/2} \log n)$ time. The second is a Monge property-based range searching data structure
by Kaplan et al. [23], which allows step (i) of \textsc{SearchAndSwitch} to track the minimum slack outgoing edge of from a vertex in $H$ in amortized $O(\text{poly}(n))$ time per operation.

The algorithm in [6], only changes dual weights at the end of a phase, after the single augmenting path of that phase is found, which allows it to completely reconstruct all of the Monge range searching structures each phase. However, the algorithm described in Section 6.4.3 dynamically changes the dual weights $\tilde{y}(\cdot)$, and therefore the slacks in $H$, throughout the \textsc{SearchAndSwitch} procedure, and the affected Monge range searching structures must be updated immediately to support this change. The result of Kaplan et al. [23] does not mention any sort of dynamic cost update operations. However, we give a procedure that allows the data structure to perform these updates efficiently within the setting of our algorithm.

\textbf{$r$-Division} In the planar graph setting, we can efficiently compute a planar graph $r$-division, which satisfies stricter requirements than an $r$-clustering. An $r$-division is a partition of the edge set of the graph into $O(n/r)$ pieces of size at most $r$ each having $O(\sqrt{r})$ boundary vertices. The total number of boundary vertices, counting multiplicities is $O(n/\sqrt{r})$.

We will reuse the same notations described on the $r$-clustering for the $r$-division. For our algorithm, we require an additional property for the $r$-division; each piece $\mathcal{R}_j$ of the $r$-division has $O(1)$ holes, which are faces of $\mathcal{R}_j$ that are not faces of the original graph $G$. Given a constant degree planar graph, an $r$-division with few holes can be constructed in $O(n \log n)$ time [28]. The constant degree assumption can be assumed without loss of generality for planar graphs; an explanation is given in [6].
6.6.1 Algorithm

Our improved planar graph matching algorithm is mostly identical to that presented in Section 6.4.3. In this section, we only describe the modifications. For the second step, this includes using a nearest neighbor data structure to support faster augmenting path searches and speeding up the Construct procedure by using Klein’s MSSP data structure. For the third step, the algorithm of [6] can be used almost directly.

We describe the modifications to the algorithm of Section 6.4.3 under the assumption that we have access to a nearest neighbor data structure on the compressed residual graph $H$ that supports the following operations.

- **FINDMIN**: Given a vertex $u \in V_H$, return the minimum slack outgoing edge $(u, v) \in E_H$.
- **RAISE**: Given a vertex $v \in V_H$ whose dual weight magnitude increased by a value $c$, update the dual weight of $v$ in the data structure.
- **BUILD**: Given a piece $R_j$, build nearest neighbor data structure for edges $E^H_j$ in $R_j$.

We assume that such a structure can be constructed in $\tilde{O}(n/\sqrt{r})$ time. Specifically, the Build operation takes $\tilde{O}(\sqrt{r})$ time per piece, and the data structure can be constructed by calling Build on each of the $O(n/r)$ pieces. The FindMin operation can be implemented in $O(\text{poly}(\log r))$. The time for Raise is bounded in an amortized sense. After $k$ Raise operations and $d$ Build operations, the total time spent for Raise is $\tilde{O}(k + \sqrt{r}d)$. This data structure is described in detail in Subsection 6.6.2.

The first step of the algorithm is unchanged for the planar graph version; the algorithm will still execute $O(\sqrt{r})$ iterations of Gabow and Tarjan’s algorithm, taking $O(n\sqrt{r})$ time. We next describe the changes to the second step of the algorithm.
The planar graph version of the second step has two main sources of improvement. The first source of improvement arises from speeding up the **Construct** procedure. It is easy to see that, using the planar graph MSSP data structure of Klein [27], the edges of a piece of \( H \) can be rebuilt in \( \tilde{O}(r) \) time. The same data structure was used to reconstruct pieces of \( H \) in [6]. As was the case for the algorithm in Section 6.4, the total number of affected pieces is \( \tilde{O}(n/\sqrt{r}) \). Therefore, the total work done by **Construct** is \( \tilde{O}(n\sqrt{r}) \).

The algorithm in the **SearchAndSwitch** procedure described earlier takes \( \tilde{O}(\sqrt{r}) \) average time per vertex visit to identify the minimum slack outgoing edge from the end of the search path and update the sorted orderings. However, we can use the **FindMin** and **Raise** procedures of the nearest neighbor data structure described above to reduce this to \( O(\text{poly}(\log r)) \) amortized time per visit for the case of planar graphs. During step (i) of **SearchAndSwitch**, the minimum slack outgoing edge from the vertex \( u \) at the end of the search path can be identified in \( O(\text{poly}(\log r)) \) time by executing a **FindMin** query on the nearest neighbor data structure. The second step invokes **FindMin** at most \( O(\beta n/r) = O(n^{3/2}/r^{3/4}) \) times for a total time of \( O(n^{3/2}/r^{3/4}) \).

Whenever the dual weight \( \tilde{y}(u) \) of a vertex \( u \in V_H \) increases in magnitude during step (i) of **SearchAndSwitch**, the algorithm will execute **Raise** on \( u \). This can occur at most \( O(\beta n/r) = O(n^{3/2}/r^{3/4}) \) times during the algorithm. The total complexity of a sequence of \( k \) **Raise** operations is \( \tilde{O}(k) \), not counting the cost of \( \tilde{O}(\sqrt{r}) \) incurred from each execution of **Build**. However, these additional \( \tilde{O}(\sqrt{r}) \) terms can easily be taxed on the costs for **Build** itself. We conclude that the total time for all **Raise** and **FindMin** operations, aside from that taxed on **Build**, is \( \tilde{O}(n^{3/2}/r^{3/4}) \).

The dual weights \( \tilde{y}(\cdot) \) also change during **Switch**. However, the algorithm can simply call **Build** on each piece whose slacks changed during **Switch** at the end of **Switch**. The required number of calls to **Build** is proportional to the number of affected pieces. When
the dual weight of a boundary vertex changes during SWITCH, BUILD must be called on each adjacent piece. However, since the graph is constant degree, this does not asymptotically increase the number of BUILD operations. Therefore, the time taken for BUILD is dominated by the time taken for CONSTRUCT.

After the second step, the algorithm has a compressed feasible matching with \( O(\sqrt{n}/r^{1/4}) \) unmatched vertices remaining. Each of these remaining vertices can be matched one at a time using iterations of Hungarian search. The third step of the algorithm in Section 6.4.3 implements each such search in \( \tilde{O}(m) \) time. However, in the planar setting, we can make use of existing planar shortest path data structures to execute Hungarian searches more efficiently. The procedure for this improved Hungarian search is described extensively in [6], and applies to our setting with minimal modification. Using FR-Dijkstra[15, 23], each Hungarian Search is executed in \( \tilde{O}(n/\sqrt{r}) \) time. After finding an augmenting path during the third step, the algorithm must update \( H \) in all pieces containing edges of the augmenting path. However, using the same arguments as those presented for the algorithm of Section 6.4 (or the similar argument in [6]), the total number of such affected pieces can be shown as \( O(n/\sqrt{r} \log n) \)). Reconstructing a piece of \( H \) requires \( \tilde{O}(r) \) time, for a total time of \( \tilde{O}(n\sqrt{r}) \).

Note that this is the same time complexity, ignoring log terms, as that for rebuilding the pieces of \( H \) in the second step.

Combining the times taken for the first, second and third steps gives \( \tilde{O}(n^{3/2}/r^{3/4} + n\sqrt{r}) \). Setting \( r = n^{2/5} \) gives a time of \( \tilde{O}(n^{6/5}) \) per scale as desired. Over the \( O(\log nC) \) scales, the total time taken is \( \tilde{O}(n^{6/5} \log nC) \). It remains to describe the implementation details of the nearest neighbor data structure.
6.6. Planar Graph Matching Algorithm

6.6.2 Nearest Neighbor Data Structure

We next describe the implementation details of the nearest neighbor data structure used in Subsection 6.6.1. We define a set of data structures for each piece $R_j$, using a standard technique. Each data structure of a piece will support a FINDMIN-type operation as well as a RAISE-type operation on a subset of edges in $E^H_j$.

The goal of FINDMIN is to identify the minimum slack outgoing edge from a vertex of $H$. Recall that the slack of any edge $(u, v) \in E_H$ can be computed by using $s_H(u, v) = \phi(u, v) - |\tilde{y}(u)| + |\tilde{y}(v)|$. Since all outgoing edges from $u$ have the same value $\tilde{y}(u)$, it is sufficient to find the edge $(u, v')$ that minimizes $\phi(u, v') + |\tilde{y}(v')|$. For the purposes of identifying the minimum slack edge outgoing from $u$, we define the cost of any edge $(u, v)$ as $c(u, v) = \phi(u, v) + |\tilde{y}(v)|$. Observe that the net-costs only change during SWITCH, after which BUILD is called on each affected piece. Therefore, the data structure only needs to support dynamic cost increase operations, corresponding to an increase in the magnitude of $\tilde{y}(v)$.

To identify the minimum slack outgoing edge from $u$ in the case where $u$ is a boundary vertex, the algorithm will split the boundary-to-boundary edges of each piece into Monge groups using the standard technique given by Fakcharoenphol and Rao in [15] and reiterated by Kaplan et al. in [23]. Each group will have a corresponding Monge matrix. A matrix $M$ is Monge if for any pair of rows $i < j$ and any pair of columns $k < l$, $M_{ik} + M_{jl} \leq M_{il} + M_{jk}$.

For any hole $h$ of a piece, we can define a cost matrix $M^h$ whose row and column orderings correspond to a clockwise ordering of the boundary vertices of $h$. Here $M^h_{ij}$ is the cost of the edge from the $i$th node to the $j$th node in the clockwise ordering. This matrix can be recursively divided into Monge submatrices with each vertex of $h$ belonging to $O(\log r)$

---

2For notational convenience, we do not distinguish between Monge and inverse Monge matrices in this description.
submatrices. For any pair of distinct holes $h \neq h'$, we can define a cost matrix $M^{h,h'}$ whose rows correspond to the clockwise ordering of $h$ and whose columns correspond to the clockwise ordering of $h'$. $M^{h,h'}$ can be replaced by two Monge matrices. Since there are $O(1)$ holes per piece and each vertex is part of $O(1)$ pieces, each vertex belongs to $O(\log r)$ Monge groups. For each of the Monge matrices, we make the common assumption that the Monge matrices are not explicitly represented in memory, rather, the cost of any $M_{ij}$ can be computed in $O(1)$ time by computing $\phi(u, v) + |\tilde{g}(v)|$. For each Monge matrix group, the algorithm will maintain a data structure that supports the following operations.

- **FindMinInColumn**: Given any column of $M$, return the minimum value in the column.
- **RaiseRow**: Given a row of $M$, increase the value of all entries in the row by a constant $c$.

A description of how to construct a data structure that efficiently supports **FindMinInColumn** is given in [23]. However, their result does not explicitly support **RaiseRow**. Since we will be defining the **RaiseRow** function on their data structure, we describe their data structure in some detail in Subsection 6.6.3. For full details, see their paper. The Monge matrix data structure can be built on a $p \times q$ matrix in $\tilde{O}(p)$ time. It supports **FindMinInColumn** queries in $\tilde{O}(\log p)$ time. In the following section, we describe how any sequence of $k$ **RaiseRow** operations can be implemented for this data structure in $\tilde{O}(p + k)$ time.

Given these complexities, the complexities of the global nearest neighbor operations **FindMin**, **Raise**, and **Build** easily follow. Calling **Build** on a piece $R_j$ requires reconstructing all the Monge matrix data structures on the piece. Since each boundary vertex is represented in $O(\log r)$ such Monge matrices, the total time spent is $\tilde{O}(\sqrt{r})$. To support the **FindMin** operation on a vertex $u \in V_H$, it is sufficient to query the $O(\log r)$ Monge matrices that $u$
belongs to. This can be done in \( O(\text{poly}(\log r)) \) time. Finally, the \texttt{RAISE} operation can be supported for a vertex \( u \in V_H \) by calling \texttt{RaiseRow} on all Monge matrix data structures containing \( u \). When \texttt{BUILD} is called on a piece, each of the Monge matrix data structures in the piece are reconstructed. In between two such reconstructions, the \texttt{RaiseRow} cost associated with the number of rows \( p \) could be accumulated once again. Hence, after \( d \) \texttt{BUILD} operations and \( k \) \texttt{RAISE} operations, the total time taken is \( \tilde{O}(k + d\sqrt{r}) \), as desired.

We note that this setup as described is only organized on boundary-to-boundary edges of \( H \). However, it is easy to support a similar set of operations for edges adjacent to free internal vertices within the same time complexity. It remains to present the Monge data structure for a single Monge group.

### 6.6.3 Data Structure on a Monge Matrix

This section gives details of how to implement a data structure on each Monge matrix group that supports the operations \texttt{FindMinInColumn} and \texttt{RAISERow}. The majority of the data structure uses the result of [23]; our only contribution is the \texttt{RAISERow} procedure. We describe the inner workings of the structure in some detail; for full details, see [23].

The structure takes as input a \( p \times q \) Monge matrix \( M \) and supports the following query in \( O(\text{poly}(\log p)) \) time: for any submatrix of \( M \) consisting of any one column of \( M \) and any contiguous interval of rows in \( M \), what is the minimum value within that submatrix?

By plotting the values of any row of \( M \), and linearly interpolating between points, we can obtain a set of pseudolines \( L \). Let \( \ell_y \in L \) be a pseudoline with respect to a row \( y \). \( \ell_y \) is effectively a function \( \ell_y(x) \), where \( x \) is a row and \( \ell_y(x) = M_{yx} \), although by linearly interpolating between points, \( x \) can also be seen as a real number. From the Monge property,

\footnote{The result of [23] mainly describes finding the maximum value, but either can be computed using the same method.}
it can be shown that any pair of pseudolines cross at most once.

The lower envelope of $M$ a function $E(x) \mid x \in \mathbb{R}$, where $E(x) = \min_{y \in L} \ell_y(x)$. The lower envelope is made up of portions of pseudolines; specifically, each pseudoline is part of the lower envelope over at most one contiguous interval. A breakpoint is an intersection of two pseudolines along the lower envelope, and there are at most $O(p)$ breakpoints at any given time. Thus, the lower envelope can be compactly stored using $O(p)$ intervals. Given such a representation, one can find, for any column $x$, the row $y$ that minimizes $M_{yx}$ in $O(\log p)$ time by using binary search over the intervals.

To construct the lower envelope, the approach of [23] builds a balanced binary range tree $T$ on the rows of $M$. The leaves of $T$ represent the rows themselves, and internal nodes represent sets of all their descendants in $T$. Each node of $T$ will store the lower envelope for the set of rows it represents. These lower envelopes are computed in a bottom-up fashion, starting from the leaves. The lower envelope of a node representing a set of size $k$ can be computed from the lower envelopes of its two children in $O(k + \log k \log q)$ time. Summing over the entire tree gives a construction time of $O(p(\log p + \log q))$. Using the range tree $T$ one can, for any range of rows and any column find the minimum element in $O(\text{poly}(\log p))$ time. This is done by taking the minimum over all $O(\log p)$ canonical subsets of the range.

**Updating the Monge Data Structures** To help facilitate dual weight magnitude increases, we describe an additional procedure called RAISEROW for use with the data structure of Kaplan et al. This procedure will allow us to, for any row $y$ of the matrix $M$, increase the cost of every entry in the row by a constant $c$. We give a procedure for repairing the affected portion of the lower envelope as a result of this change. The total time taken will be bounded in an amortized sense; after a sequence of $k$ RAISEROW operations, the total time taken is $\tilde{O}(p + k)$; recall that $p$ is the number of rows.
Increasing the entry of all elements in row $y$ is equivalent to raising the pseudoline $\ell_y$ up by $c$. This may introduce new breakpoints into the lower envelope, and may remove the presence of $\ell_y$ from the lower envelope entirely. Such changes may occur to the lower envelopes of any of the $O(\log p)$ nodes of the range tree $T$ that contain $y$ as a descendant; the other nodes of $T$ are unaffected. We describe how to repair the information starting at the bottom of the tree.

Assume we are given an internal node $t$ of the tree $T$ whose lower envelope information needs to be repaired, and that the lower envelope information of its two children is accurate. Let $M'$ be the submatrix consisting of the rows represented by $t$. Assume $M'$ is a $p'$ by $q$ matrix, where $p'$ is the number of rows represented by $t$, and let $E'(x)$ be the lower envelope of this submatrix. Let $[i, j]$ be the interval of values such that $E'(x) = \ell_y$. The envelope will only change in this interval, and some new breakpoints may need to be created. Given any value of $x$, we can find the pseudoline that contains $x$ on the lower envelope after raising $\ell_y$ in $O(poly(\log p'))$ time by executing two range minimum queries on the subtree of $T$ rooted at $t$. The first query interval will consist of all rows above $y$ and the second query will consist of all rows below $y$. By taking the minimum over the results of these two range queries with the new value $\ell_y(x)$, we obtain the pseudoline on the lower envelope that contains $x$ after the \texttt{RaiseRow} operation. Using this strategy, we can use binary search to find the leftmost breakpoint in the interval $[i, j]$ in $O(poly(\log p'))$ time. This process can be repeated for each successive breakpoint, until no new breakpoints are found. The time complexity is therefore proportional to the number of new breakpoints formed as a result of raising $\ell_y$. Let the number of breakpoints formed be $\alpha$. Then the complexity of \texttt{RaiseRow} is $O(\alpha poly(\log |p'|))$ for the node $t$. Next, observe two facts. First, the maximum number of breakpoints in $E'(x)$ is $O(p')$. Second, each \texttt{RaiseRow} operation reduces the number of breakpoints in $E'(x)$ by at most 1. Therefore, after a sequence of $k$ \texttt{RaiseRow} operations,
the total time taken for node $t$ is $\tilde{O}(p'+k)$. Summing over all nodes of $T$ gives us the desired total time of all \textsc{RaiseRow} operations as $\tilde{O}(p'+k)$, as desired.

6.7 Appendix

6.7.1 Discussion of Correctness for Reduce and ReduceSlack

The following discussion demonstrates that any call to \textsc{Reduce} or \textsc{ReduceSlack} that satisfies the preconditions in those procedures’ definitions will not violate compressed feasibility. For both procedures, the discussion argues conditions (a)–(e) of compressed feasibility hold, establishing Lemma 6.9.

The \textsc{Reduce} procedure sets the dual weight $\tilde{y}(\cdot)$ so that conditions (b) and (e) of compressed feasibility hold based on its preconditions. No vertex of $A_H$ has a change in dual weight during \textsc{Reduce} and so condition (a) holds. For (d), observe that all edges of $b^A_J$ (resp. $b^T_J$) are outgoing. A reduction in the dual weight $\tilde{y}(\cdot)$ will only increase the slack on every edge going out of this vertex, and so the edges in $E^H_J$ remain $H$-feasible. Similarly, every edge incident on any $v \in (V_j \setminus \mathcal{K}_j) \cap B^A_H$ (resp. $B^T_H$) is not in the matching and therefore a reduction of dual weight of $v$ will only increase the slack on the edge, implying (c).

\textsc{ReduceSlack} does not violate (e) for any piece that $v$ participates in. This is because all edges except $(u, v)$ are edges that are not in the matching, and so, a reduction of the dual weight only increases the slack on the other edges, and condition of equation (6.5) holds. From the definition of slack for matched edges, it follows that the new dual weight of $y_j(v) - s(u, v)$, is non-negative and the slack of $s(u, v)$ after the dual update is 0. Therefore the condition of equation (6.6) is satisfied, and (c) holds. For conditions (a), (b), (d) and (e), if $v$ is an internal vertex, then $\tilde{y}(\cdot)$ values are not modified by the procedure and so (a),
(b), (d), and (e) hold trivially. Otherwise, if $v$ is a boundary vertex, since $v \in B$, (a) holds trivially. and since the updated dual weight $\tilde{y}(v)$ is non-negative, (b) holds. $\tilde{y}(v)$ and $y_j(v)$ are updated so that (e) holds. Finally, for condition (d), we need to show $H$-feasibility of edges going out of $v$, we address the case where $v \in V_H$. First, consider any edge $(u', v) \in E_H$ incoming to $v$. The projection of $(u', v)$ must contain the edge $(u, v)$, since $(u, v)$ is the only edge in the residual graph that is directed into $v$. Therefore, the slack $s_H(u', v) \geq s(u, v)$.

The procedure decreases the dual weight $\tilde{y}(v)$. This reduces the slack on $(u', v)$ by at most $s(u, v)$ implying that the slack on the edge $(u', v)$ remains non-negative. Every other edge of $H$ incident on $v$ is directed away from $v$, so a reduction of dual weight only increases the slack on these edges, implying (d).

### 6.7.2 Details of the Construct Procedure

This section describes in further detail how to implement the \textsc{Construct} procedure defined in Section 6.4.1. The input to the \textsc{Construct} procedure is an $\mathcal{R}$-feasible matching $M_j$ and the dual weights $y_j(\cdot)$. Let $\mathcal{R}'_j$ be the graph of $\mathcal{R}_j$, with all edge weights converted to their slacks according to the current matching $M_j$ and the current dual assignment $y_j(\cdot)$. We note that all edges in $\mathcal{R}'_j$ are non-negative. Since the dual assignment is feasible with respect to $M_j$, we know by Lemma 6.11 that the path of minimum net-cost between two vertices is also the path of minimum total slack in $\mathcal{R}'_j$. Therefore, it is sufficient to compute the shortest path lengths in $\mathcal{R}'_j$ and use (6.29) to compute the minimum net-cost path in constant time.

Recall that there are four types of edges in $E^H_j$. To compute the boundary-to-boundary edges $(u, v) \in E^H_j$, for each $u \in K_j$, we execute a Dijkstra search over $\mathcal{R}'_j$ from $u$ to obtain the length of the shortest slack path from $u$ to every other boundary node. To compute an edge from a boundary node $u$ to itself, for each such boundary vertex $u \in K_j$, create
a duplicate vertex \( u' \) and add an edge from (resp. to) \( u' \) to (resp. from) any other vertex \( v \in V_j \) if and only if there is an edge from (resp. to) \( u \) to (resp. from) \( v \) in \( E_j \) with the same cost, i.e., slack \( s(u,v) \). Execute Dijkstra’s algorithm from \( u \) over \( R'_j \) to find the distance to \( u' \).

For piece \( R_j \), we describe how to compute the edges from a vertex \( b_j^A \) to a boundary node or the free internal vertex \( a_j \). A similar argument also applies for computing edges from \( b_j^F \). We add a new vertex \( s \) to \( R'_j \) and connect them to every free internal vertex \( v \in (V_j \setminus K_j) \cap B_f^A \). The cost of the newly added edges is set to zero. Then, we execute Dijkstra’s algorithm from \( s \). For every boundary node \( v \in K_j \), we add an edge from \( b_j \) to \( v \) if there is a path between \( s \) and \( v \) and compute the cost \( \phi(\overrightarrow{P}_{b_j,v,j}) \) by using Lemma 6.11. We add an edge between \( b_j \) and \( a_j \) if there is a path from \( s \) to some \( v \in (V_j \setminus K_j) \cap A_F \) and among all such vertices which have a path from \( b_j \), use the one that has the smallest cost path from \( s \). Using Lemma 6.11, we can obtain the weight \( \phi(\overrightarrow{P}_{b_j,a_j,j}) \). We can use an identical algorithm to compute edges incident on \( a_j \) by applying Dijkstra’s algorithm on the graph \( R''_j \) where \( R''_j \) is a graph identical to \( R'_j \) except that every edge is in the reverse direction. Together, these three searches compute the remaining edges of \( E^H_j \). The total time taken to compute all the edges of \( E^H_j \) is \( O(\sqrt{r}(m_j + n_j \log n_j)) \) time because \( O(\sqrt{r}) \) Dijkstra searches over \( R'_j \) are executed.

From this discussion, Lemma 6.12 follows. By summing over the sizes of all pieces of the \( r \)-clustering, we get Corollary 2.

### 6.7.3 Proofs for Properties of Sync

This section provides proofs for properties (1) and (2) of the Sync procedure given in Section 6.4.2. It also presents Lemma 6.24 which is used in the proof of Lemma 6.13. Finally, we
Lemma 6.23. At the end of the Sync procedure, both (1) and (2) hold.

Proof. Let us denote the dual weights before and after applying the Sync procedure as $y_j^*(\cdot)$ and $y_j(\cdot)$. We also denote the slack on any edge $(u, v)$ with respect to the original dual weights $y_j^*(\cdot)$ as $s^*(u, v)$. To prove (1), we need to show that the matching $M_j$ along with the new dual weights $y_j(\cdot)$ are $\mathcal{R}$-feasible. We first note that the dual weights change only if $\ell_v \leq \kappa$ and the change is by $\lambda_v(\kappa - \ell_v)$. This change is positive for vertices of $B$ and negative for vertices of $A$. Therefore, the magnitude of dual weights does not decrease from the procedure. We show (6.5) and (6.6) next. For any edge $(u, v)$ directed from $u$ to $v$, we know from the properties of shortest paths that $\ell_v \leq \ell_u + s^*(u, v)$ or

$$
\ell_v - \ell_u \leq s^*(u, v),
$$

$$(\kappa - \ell_u) - (\kappa - \ell_v) \leq s^*(u, v). \quad (6.35)
$$

If $(u, v) \in M_j$, then $u \in A$, $v \in B$ and $s^*(u, v) = y_j^*(u) + y_j^*(v) - c(u, v) + \delta_{uv}$. We can rewrite the above equation as

$$
(\kappa - \ell_u) - (\kappa - \ell_v) \leq y_j^*(u) + y_j^*(v) - c(u, v) + \delta_{uv},
$$

$$(y_j^*(u) + \lambda_u(\kappa - \ell_u)) + (y_j^*(v) + \lambda_v(\kappa - \ell_v)) \geq c(u, v) - \delta_{uv},
$$

$$
y_j(u) + y_j(v) \geq c(u, v) - \delta_{uv}.
$$

satisfying (6.6). If the edge $(u, v)$ directed from $u$ to $v$ is not in the matching, then $u \in B$, 


$v \in A$, and $s^*(u,v) = c(u,v) + \delta_{uv} - y^*_j(u) - y^*_j(v)$ Therefore,

$$(\kappa - \ell_u) - (\kappa - \ell_v) \leq c(u,v) + \delta_{uv} - y^*_j(u) - y^*_j(v),$$

$$(y^*_j(u) + \lambda_u(\kappa - \ell_u)) + (y^*_j(v) + \lambda_v(\kappa - \ell_v)) \leq c(u,v) + \delta_{uv},$$

$$y_j(u) + y_j(v) \leq c(u,v) + \delta_{uv}.$$  

implying that the edge $(u,v)$ satisfies (6.5) implying $M_j, y_j(\cdot)$ is $R$-feasible.

To prove (2), we need to show that for any vertex $v \in K_j \cup ((V_j \setminus K_j) \cap (A_F \cup B_F))$, the new shortest path from $s$ to $v$ in $R'_j$ is the direct edge from $s$ to $v$. If we show this, then $\ell_v = \kappa - \kappa_v = \kappa - |\tilde{y}(v)| + |y^*_j(v)|$ or $\lambda_v \tilde{y}(v) - \lambda_v y^*_j(v) = \kappa - \ell_v$. This gives $\tilde{y}(v) = y^*_j(v) + \lambda_v(\kappa - \ell_v) = y_j(v)$ because $\lambda_v \in \{1,-1\}$.

Therefore, we will show that the shortest path from $s$ to $v$ is no less than the cost of the edge from $s$ to $v$. For the sake of contradiction, let the shortest path, $\overrightarrow{P}_{s,v}$ from $s$ to $v$ be strictly less than the cost of the edge $(s,v)$. Let $u$ be the first vertex that appears on $\overrightarrow{P}_{s,v}$ after $s$. By the optimal substructure property of shortest paths, $\overrightarrow{P}_{s,v}$ with the vertex $s$ removed forms a shortest path from $u$ to $v$. Since all edge costs are the original slacks $s^*(\cdot)$, this path was also the shortest slack path, $\overrightarrow{P}_{u,v,j}$, prior to SYNC. We know that the length of $\overrightarrow{P}_{s,v}$ is $\kappa - |\tilde{y}(u)| + |y^*_j(u)| + \sum_{(a,b) \in \overrightarrow{P}_{u,v,j}} s^*(a,b)$. Since the length of $\overrightarrow{P}_{s,v}$ is smaller than the cost of the direct edge from $s$ to $v$, we have

$$\kappa - |\tilde{y}(v)| + |y^*_j(v)| > \kappa - |\tilde{y}(u)| + |y^*_j(u)| + \sum_{(a,b) \in \overrightarrow{P}_{u,v,j}} s^*(a,b),$$

$$|\tilde{y}(u)| - |\tilde{y}(v)| > |y^*_j(u)| - |y^*_j(v)| + \sum_{(a,b) \in \overrightarrow{P}_{u,v,j}} s^*(a,b)$$

$$= \phi(\overrightarrow{P}_{u,v,j}).$$
The last equality holds because, from Lemma 6.10 and the fact that $M_j, y^*_j(\cdot)$ was an $\mathcal{R}$-feasible matching, $|y^*_j(u)| - |y^*_j(v)| + \sum_{(a,b) \in P_{u,v,j}} s^*(a,b)$ will be equal to $\phi(\overrightarrow{P}_{u,v,j})$. The inequality $|\tilde{y}(u)| - |\tilde{y}(v)| > \phi(\overrightarrow{P}_{u,v,j})$ contradicts the $H$-feasibility of the input to SYNC. \hfill $\square$

**Lemma 6.24.** Suppose we are given a piece $\mathcal{R}_j$ and a dual weight $y_j(\cdot)$ for every vertex in $V_j$ and $\tilde{y}(\cdot)$ for each vertex in $V^H_j$. Upon applying the SYNC procedure, let $v$ be any vertex in $V_j$ for which $\ell_v \leq \kappa$. Let $P = \langle s = u_0, u_1, \ldots, u_t = v \rangle$ be the shortest path from $s$ to $v$ in $R'_j$. Then, after the SYNC procedure, the slack on every edge $(u_q, u_{q+1})$ with respect to the updated dual weights $y_j(\cdot)$ for $1 \leq q < t$ is zero.

**Proof.** As in the previous proof, we denote the dual weights prior to the execution of the SYNC procedure by $y^*_j(\cdot)$ and the dual weights after by $y_j(\cdot)$. Also, let $s^*(\cdot, \cdot)$ denote the slack of an edge in $G$ with respect to the dual weights $y^*_j(\cdot)$. Since, $P$ is the shortest path from $s$ to $v$, for any directed edge on this path from $u_q$ to $u_{q+1}$,

$$\ell_{u_{q+1}} = \ell_{u_q} + s^*(u_q, u_{q+1}). \quad (6.36)$$

The shortest path cost from $s$ to $v$, $\ell_v$ is at most $\kappa$ and so, for any vertex $u_q$ on the shortest path to $v$, the shortest path to $\ell_{u_q}$ is at most $\kappa$. SYNC sets dual weights such that $y_j(u_q) = y^*_j(u_q) + \lambda_{u_q}(\kappa - \ell_{u_q})$. Therefore, for any edge $(u_q, u_{q+1})$ on $P$ we have

$$y_j(u_{q+1}) = y^*_j(u_{q+1}) + \lambda_{u_{q+1}}(\kappa - \ell_{u_{q+1}}), \quad (6.37)$$

$$y_j(u_q) = y^*_j(u_q) + \lambda_{u_{q+1}}(\kappa - \ell_{u_q}). \quad (6.38)$$

We consider the cases where $(u_q, u_{q+1}) \in M$, and $(u_q, u_{q+1}) \notin M$. First, we consider the case
where \((u_q, u_{q+1}) \in M\). Matching edges are directed from a vertex of \(A\) to a vertex of \(B\), and so, \(u_q \in A\) and \(u_{q+1} \in B\). By the definition of slack for matching edges, we have

\[
s(u_q, u_{q+1}) = y_j(u_q) + y_j(u_{q+1}) - c(u_q, u_{q+1}) + \delta_{u_q u_{q+1}}
\]

\[
= y_j^*(u_q) + y_j^*(u_{q+1}) - c(u_q, u_{q+1}) + \delta_{u_q u_{q+1}} + \lambda_{u_q} (\kappa - \ell_{u_q}) + \lambda_{u_{q+1}} (\kappa - \ell_{u_{q+1}})
\]

\[
= s^*(u_q, u_{q+1}) - (\kappa - \ell_{u_q}) + (\kappa - \ell_{u_{q+1}})
\]

\[
= s^*(u_q, u_{q+1}) + \ell_{u_q} - \ell_{u_{q+1}} = 0.
\]

The last two equations follow from (6.36) and the fact that \(\lambda_{u_q} = -1\) and \(\lambda_{u_{q+1}} = 1\).

Next, we consider the case where \((u_q, u_{q+1}) \notin M\). Edges that are not in the matching are directed from a vertex of \(B\) to a vertex of \(A\), and so, \(u_q \in B\) and \(u_{q+1} \in A\). By the definition of slack for edges that are not in the matching, we have

\[
s(u_q, u_{q+1}) = c(u_q, u_{q+1}) + \delta_{u_q u_{q+1}} - y_j(u_q) - y_j(u_{q+1})
\]

\[
= c(u_q, u_{q+1}) + \delta_{u_q u_{q+1}} - y_j^*(u_q) - y_j^*(u_{q+1}) - \lambda_{u_q} (\kappa - \ell_{u_q}) - \lambda_{u_{q+1}} (\kappa - \ell_{u_{q+1}})
\]

\[
= s^*(u_q, u_{q+1}) - (\kappa - \ell_{u_q}) + (\kappa - \ell_{u_{q+1}})
\]

\[
= s^*(u_q, u_{q+1}) + \ell_{u_q} - \ell_{u_{q+1}} = 0.
\]

The last two equations follow from (6.36) and the fact that \(\lambda_{u_q} = 1\) and \(\lambda_{u_{q+1}} = -1\).

Next, using Lemma 6.24, we give a proof for Lemma 6.13. We first restate verbatim the claim of Lemma 6.13.
Consider a compressed feasible matching with dual weights \( \tilde{y}(\cdot) \) assigned to every vertex of \( V_H \). For any piece \( \mathcal{R}_j \) and any vertex \( v \in V_j \), let \( y^*_j(v) \) denote the dual weight prior to executing \( \text{SYNC} \), and for any edge \( (u, v) \in E_j \), let \( s^*(u, v) \) be the slack prior to executing \( \text{SYNC} \). Let \( y_j(\cdot) \) denote the dual weights of \( V_j \) after this execution. For any edge \( (u, v) \in E^H_j \) with a projection \( \overrightarrow{P}_{u,v,j} = \langle u = u_0, u_1, \ldots, u_t, u_{t+1} = v \rangle \), suppose \( \left| \tilde{y}(u) \right| - \left| y^*_j(u) \right| \geq \sum_{q=0}^{t} s^*(u_q, u_{q+1}) \). Let \( \overrightarrow{P}_{s,u,t,j} \) be any shortest path from \( s \) to \( u_t \) in \( \mathcal{R}'_j \). Then,

(i) If there exists a shortest path \( \overrightarrow{P}_{s,u,t,j} \) in \( \mathcal{R}'_j \) where \( u \) is the second vertex on this path, then after the execution of \( \text{SYNC} \) procedure, for every \( 1 \leq i \leq t - 1 \), \( s(u_i, u_{i+1}) = 0 \) and \( s(u_t, v) \leq \left| \tilde{y}(v) \right| - \left| y^*_j(v) \right| \).

(ii) Otherwise, there is no shortest path \( \overrightarrow{P}_{s,u,t,j} \) in \( \mathcal{R}'_j \) with \( u \) as its second vertex. Consider \( u^* \) to be the second vertex of some \( \overrightarrow{P}_{s,u,t,j} \) and \( u^* \neq u \). Then, \( u^* \in (\mathcal{K}_j \cup (V_j \cap (A_F \cup B_F))) \), and \( \left| \tilde{y}(u^*) \right| - \left| y^*_j(u^*) \right| > \sum_{(u', v') \in \overrightarrow{P}_{u^*,v,j}} s^*(u', v') \).

**Proof.** Let \( y^*_j(\cdot) \) and \( y_j(\cdot) \) be the dual weights before and after the execution of the \( \text{SYNC} \) procedure. Also, let \( s^*(\cdot, \cdot) \) denote the slack of an edge in \( G \) with respect to the dual weights \( y^*_j(\cdot) \). Note that the dual weights \( \tilde{y}(\cdot) \) do not change from the execution of the \( \text{SYNC} \) procedure. First, we establish that for every vertex \( u_i \) along \( \overrightarrow{P}_{u,v,j} \), \( \ell_{u_i} \leq \kappa \). By our assumption,

\[
\kappa_u = \left| \tilde{y}(u) \right| - \left| y_j(u) \right| \geq \sum_{q=0}^{t} s^*(u_q, u_{q+1}). \tag{6.39}
\]

For any \( i \), such that \( 0 \leq i \leq t + 1 \), consider the cost of the path \( \langle s, u_0, u_1, \ldots, u_i \rangle \). From (6.39), the cost of the edge \( (s, u_0) = \kappa - \kappa_u \leq \kappa - \sum_{q=0}^{t} s^*(u_q, u_{q+1}) \), and for any \( 0 \leq q < i \), the cost of the edge \( (u_q, u_{q+1}) = s^*(u_q, u_{q+1}) \). Therefore, the cost of the path \( \langle s, u_0, u_1, \ldots, u_i \rangle \)
is at most

$$\ell_{u_i} \leq \kappa - \sum_{q=0}^{t} s^*(u_q, u_{q+1}) + \sum_{q=0}^{i-1} s^*(u_q, u_{q+1}) = \kappa - \sum_{q=i}^{t} s^*(u_q, u_{q+1}).$$  \hspace{1cm} (6.40)$$

This implies $\ell_{u_i} \leq \kappa$, and the dual weight of $u_i$ is updated by the SYNC procedure.

From (6.40), $\ell_{u_i} \leq \kappa - \sum_{q=i}^{t} s^*(u_q, u_{q+1})$. The new dual weight of $u_i$ as updated by the SYNC procedure is $y_j(u_i) \leftarrow y_j^*(u_i) + \lambda_{u_i} (\kappa - \ell_{u_i})$, or,

$$|y_j(u_i)| - |y_j^*(u_i)| \geq \kappa - \ell_{u_i} \geq \sum_{q=i}^{t} s^*(u_q, u_{q+1}).$$ \hspace{1cm} (6.41)

Note that if $\langle s, u_0, u_1, ..., u_i \rangle$ is not the shortest path from $s$ to $u_i$ in $R'_j$, then inequalities (6.40) and (6.41) are strict inequalities.

First, we address the case where there is a shortest path $\overrightarrow{P}_{s,u_1,j}$ in $R'_j$ with $u$ as its second vertex. Since the path $\overrightarrow{P}_{u,v,j}$ is the shortest path from $u$ to $v$ in $R'_j$, from the optimal substructure property, a shortest path from $s$ to $u_t$ is $\langle s, u, u_2, ..., u_{t-1}, u_t \rangle$. Since $\ell_{u_t} \leq \kappa$, from Lemma 6.24, every edge on this path will have a zero slack.

From (6.41), the change in dual weight for $u_t$ is

$$|y_j(u_t)| - |y_j^*(u_t)| \geq s^*(u_t, v).$$ \hspace{1cm} (6.42)

The slack for the edge $(u_t, v)$ is given by

$$s(u_t, v) = s^*(u_t, v) - (|y_j(u_t)| - |y_j^*(u_t)|) + (|y_j(v)| - |y_j^*(v)|)$$

$$\leq |y_j(v)| - |y_j^*(v)|$$

$$= |\tilde{y}(v)| - |y_j^*(v)|.$$
This completes the proof for (i).

Next, we address case (ii), where \( u^* \neq u \). Note that the only edges that are leaving \( s \) are to the vertices of \( K_j \cup \{(A_F \cup B_F) \cap V_j\} \). So, \( u^* \) has to be a vertex of this set. Next, let the path \( \overrightarrow{P}_{s,u^*} = \{s = s_0, s_1, \ldots, s_{\alpha} = u_t\} \) be the shortest path from \( s \) to \( u_t \) with \( s_1 = u^* \). Therefore, from Lemma 6.24, all edges on this path have zero slack with respect to the dual weights \( y_j(\cdot) \). From (6.18),

\[
|y_j(u^*)| - |y_j(u_t)| = \phi(\overrightarrow{P}_{u^*,u_t,j}).
\] (6.43)

Before the execution of the SYNC procedure, from (6.18) we have,

\[
\sum_{q=0}^{\alpha-1} s_s(s_q, s_{q+1}) + |y_j(u^*)| - |y_j(u_t)| = \phi(\overrightarrow{P}_{u^*,u_t,j}).
\] (6.44)

Subtracting (6.44) from (6.43) gives,

\[
(|y_j(u^*)| - |y_j(u_t)|) - (|y_j(u_t)| - |y_j^*(u_t)|) = \sum_{q=0}^{\alpha-1} s_s(s_q, s_{q+1}),
\]

\[
(|y_j(u^*)| - |y_j^*(u^*)|) = \sum_{q=0}^{\alpha-1} s_s(s_q, s_{q+1}) + (|y_j(u_t)| - |y_j^*(u_t)|).
\]

Note that if \( s, u_0, u_1, \ldots, u_t \) is not the shortest path from \( s \) to \( u_t \) in \( R'_j \), then, as stated before inequalities (6.40) and (6.41) are strict inequalities. From applying (6.41) we get that \( |y_j(u_t)| - |y_j^*(u_t)| > s^*(u_t, v) \). Therefore,

\[
(|y_j(u^*)| - |y_j^*(u^*)|) > \sum_{q=0}^{\alpha-1} s_s(s_q, s_{q+1}) + s^*(u_t, v) \geq \sum_{(u', v') \in \overrightarrow{P}_{u^*,v,j}} s_s(u', v').
\] (6.45)

By property (2) of the SYNC procedure, \( y_j(s_1) = \tilde{y}(s_1) \), and therefore, (ii) follows. \( \Box \)
Chapter 7

An Approximation Algorithm for
RMS Matching in a Plane

The 2-Wasserstein distance (or RMS distance) is a useful measure of similarity between probability distributions that has exciting applications in machine learning. For discrete distributions, the problem of computing this distance can be expressed in terms of finding a minimum-cost perfect matching on a complete bipartite graph given by two multisets of points $A, B \subset \mathbb{R}^2$, with $|A| = |B| = n$, where the ground distance between any two points is the square of the Euclidean distance between them. Although there is a previously-existing near-linear time relative $(1 + \varepsilon)$-approximation algorithm for the case where the ground distance is Euclidean [51], all existing relative $(1 + \varepsilon)$-approximation algorithms for the RMS distance took $\Omega(n^{3/2})$ time prior to this work. This is primarily because, unlike Euclidean distance, squared Euclidean distance is not a metric because it does not obey the triangle inequality. In this chapter, for the RMS distance, we present a new $(1 + \varepsilon)$-approximation algorithm that runs in $O(n^{5/4}\text{poly}\{\log n, 1/\varepsilon\})$ time.

This algorithm is inspired by the recent approach for finding a minimum-cost perfect matching in bipartite planar graphs [6] (described in Chapter 5). This algorithm depends heavily on the existence of sub-linear sized vertex separators as well as shortest path data structures that require planarity. Surprisingly, we are able to design a similar algorithm for a complete geometric graph that is far from planar and does not have any vertex separators.
Central components of our algorithm include a quadtree-based distance that approximates the squared-Euclidean distance and a data structure that supports both Hungarian search and augmentation in sub-linear time.

7.1 Introduction

Given two sets $A$ and $B$ of $n$ points in $\mathbb{R}^2$, let $\mathcal{G}(A \cup B, A \times B)$ be the complete bipartite graph on $A, B$. A matching $M$ is a set of vertex-disjoint edges of $\mathcal{G}$. The matching $M$ is perfect if it has cardinality $n$. For any $p \geq 1$, the cost of an edge $(a, b)$ is simply $\|a - b\|^p$; here, $\|a - b\|$ is the Euclidean distance between $a$ and $b$. Consider the problem of computing a matching $M$ that minimizes the sum of all its edges’ costs, i.e., the matching with smallest $w_p(M) = \sum_{(a, b) \in M} \|a - b\|^p$. When $p = 1$, this problem is the well-known Euclidean bipartite matching problem. When $p = 2$, the matching computed minimizes the sum of the squared Euclidean distances of its edges and is referred to as the RMS matching. For $p = \infty$, the matching computed will minimize the largest cost edge and is referred to as the Euclidean bottleneck matching. For a parameter $\varepsilon > 0$ and $p \geq 1$, we say that the matching $M$ is a $(1 + \varepsilon)$-approximate matching if $w_p(M) \leq (1 + \varepsilon)w_p(M_{OPT})$ where $M_{OPT}$ is a matching with the smallest cost. In this chapter, we consider the problem of computing a $(1 + \varepsilon)$-approximate RMS matching in the plane and present a randomized $\tilde{O}(n^{5/4})$ time\(^1\) algorithm. For the remainder of the chapter, we assume that $w(M) = w_2(M)$.

Our Results: Our main result is the following.

**Theorem 7.1.** For any point sets $A, B \subset \mathbb{R}^2$, with $|A| = |B| = n$, and for any parameter $0 < \varepsilon \leq 1$, a $(1 + \varepsilon)$-approximate RMS matching can be computed in $O(n^{5/4}\text{poly}\{\log n, 1/\varepsilon\})$

\(^1\)Recall that, throughout this chapter, we use $\tilde{O}(\cdot)$ to hide $\text{poly}\{\log n, 1/\varepsilon\}$ factors in the complexity.
All previous algorithms that compute a \((1 + \varepsilon)\)-approximate RMS matching take \(\Omega(n^{3/2})\) time. Our approach can also be used to improve the running time of the Euclidean matching algorithm from [51] by polynomial factors in \(1/\varepsilon\) and \(\log n\).

**Organization** The remainder of the chapter is organized as follows: In Section 7.2 we describe the details of our distance function while highlighting differences from the distance function of [51]. In Section 7.3 we introduce a quad-tree based dual-feasibility condition that incorporates an additional additive cost on each edge. In Section 7.4, we give a detailed description of the algorithm, along with its analysis. The algorithm description assumes the existence of a data structure built on active squares. This data structure includes the compressed feasible matching as well as several procedures, such as the sub-linear time Hungarian search and augment that operate on a compressed feasible matching, and is described in detail in Section 7.5.

### 7.2 Our Distance Function

#### 7.2.1 Initial Input Transformation

For the purposes of describing both the distance function and our algorithm, it is useful to make some assumptions about the input to our problem. Given any point sets \(A', B' \subset \mathbb{R}^2\) of \(n\) points each we show how to transform them to point sets \(A\) and \(B\) with \(n\) points each such that:

(A1) Every point in \(A \cup B\) has non-negative integer coordinates bounded by \(\Delta = n^{O(1)}\),
(A2) The optimal RMS matching of $A$ and $B$ has a cost of at most $O(n/\varepsilon^2)$, and,

(A3) Any optimal matching of $A$ and $B$ corresponds to a $(1 + \varepsilon)$-approximate matching of $A'$ and $B'$.

The details of this transformation are described in Section 7.6, but the approach can be summarized as follows: First, we obtain an $n^{O(1)}$-approximation of the optimal matching in linear-time. We further refine this estimate by making $O(\log n)$ guesses of the optimal cost, and at least one guess gives a 2-approximation. By executing our algorithm $O(\log n)$ times, one for each guess, at least one algorithm will have a 2-approximation of the optimal matching cost. Using this refined estimate, we rescale the points such that the optimal cost becomes $O(n/\varepsilon^2)$. Finally, we show that rounding the resulting points to integers does not contribute too much error. As a result, in the rest of the chapter, we assume that properties (A1)–(A3) hold. Given these assumptions, we can proceed with defining our distance function.

In the following, we describe a quad-tree based distance denoted by $d_Q(\cdot, \cdot)$ that approximates the squared-Euclidean distances between points. Our input is a multi-set, i.e., there can be multiple input points with the same $x$ and $y$ coordinate values. For any two points $a$ and $b$ with the same location, we will simply assume $d_Q(a, b) = 0$. It remains to describe our distance approximation for points $a$ and $b$ that are distinct.

### 7.2.2 Randomly Shifted Quadtree Decomposition

Similar to [51], we define our distance function based on a randomly-shifted quadtree. Without loss of generality, assume $\Delta$ is a power of 2. First, we pick a pair of integers $\langle x, y \rangle$ each independently and uniformly at random from the interval $[0, \Delta]$. We define a square $G = [0, 2\Delta]^2 - \langle x, y \rangle$ that contains all points of $P$. This square will form the root node of our quadtree, and each internal node of the tree is subdivided into 4 equal-sized squares to
form its children in the tree.

Specifically, for $\delta = \log_2(2\Delta)$ and a constant $c_1 > 0$, we construct a quadtree $Q$ of height $\delta + 2 \log(\log(\Delta) / \varepsilon) + c_1 = O(\log n)$ (from (A1)). The layers of $Q$ can be seen as a sequence of grids $\langle G_{\delta}, \ldots, G_0, \ldots, G_{-2 \log(\log(\Delta) / \varepsilon) - c_1} \rangle$. The grid $G_i$ is associated with squares with side-length $2^i$ and the grid of leaf nodes $G_{-2 \log(\log(\Delta) / \varepsilon) - c_1}$ is associated with cells of width $1/2^{2 \log(\log(\Delta) / \varepsilon) + c_1}$. Although, cells of grid $G_0$ contain at most one point (or possibly multiple copies of the same point) and can be considered leaf nodes of the quadtree, it is convenient to allow for us to define grids $G_i$ for all $i \geq -2 \log(\log(\Delta) / \varepsilon) - c_1$ and consider their cells to be part of the quadtree. We say that a square $\Box$ has a level $i$ if $\Box$ is a cell in grid $G_i$. For any two cells $\Box$ and $\Box'$, let $\ell_{\min}(\Box, \Box')$ (resp. $\ell_{\max}(\Box, \Box')$) be the minimum (resp. maximum) distance between the boundaries of $\Box$ and $\Box'$, i.e., the minimum distance between any two points $u$ and $v$ where $u$ is on the boundary of $\Box$ and $v$ is on the boundary of $\Box'$. Next, we describe how any cell of this quadtree that has a level greater than or equal to 0 can be divided into subcells, a concept essential to describe our distance function.

### 7.2.3 Division of a Cell Into Subcells

For any grid $G_i$ with $i > 0$, we define the minimum subcell size to be $\mu_i = 2^{\lfloor i/2 \rfloor - 2 \log \frac{\log \Delta}{\varepsilon} - c_1}$, where $c_1 > 0$ is the constant used in the construction of the quad-tree. Each cell $\Box \in G_i$ is subdivided into a set of subcells, with each subcell having width at least $\mu_i$. In [51], the minimum subcell size was much larger, being roughly $2^i - O(\log \frac{\log \Delta}{\varepsilon})$. In their case, dividing $\Box$ into subcells using a uniform grid of side length $\mu_i$ was sufficient, resulting in $O((2^i/\mu_i)^2) = \text{poly}\{\log n, 1/\varepsilon\}$ subcells. However, for squared-Euclidean distances, much smaller subcells are required, and using a uniform grid would result in $\Omega(2^i)$ subcells, which is too large for our purposes. Instead, we replace the uniform grid of subcells with an exponential grid of
7.2. Our Distance Function

subcells, reducing the number of subcells to $\tilde{O}(2^i/\mu_i) = \tilde{O}(2^{i/2})$. We describe this process of forming the exponential grid next. For a visual example of the exponential grid, see Figure 7.1.

For any cell $\Box$ of $Q$ with a level $i > 0$, let $\Box_1, \Box_2, \Box_3$ and $\Box_4$ be its four children. We define subcells of any cell $\Box$ as the leaf nodes of another quadtree $Q_\Box$ with $\Box$ as its root and its four children recursively sub-divided in $Q_\Box$ as follows. Let $u \leftarrow \Box_1$, we recursively divide $u$ into four cells until:

(a) Either the side-length of $u$ is the minimum subcell size $\mu_i$, or

(b) The side length of $u$ is at most $(\varepsilon/144)\ell_{\min}(\Box_1, u)$.

Similarly, we decompose $\Box_2, \Box_3$ and $\Box_4$ into subcells as well. Note that every cell of the quadtree $Q_\Box$ is also a cell in the quadtree $Q$ and the leaves of $Q_\Box$ (the subcells of $\Box$) will satisfy (a) or (b). We denote the subcells of $\Box$ by $G[\Box]$. Note that, for any subcell $u \in G[\Box]$ where $u$ is a descendant of $\Box_1$, the side-length of $u$ is larger than the minimum subcell size if and only if $\ell_{\min}(\Box_1, u)$ is sufficiently large. i.e., as we move away from the boundary of $\Box_1$, the subcell size becomes larger. Using this, in Lemma 7.2, we show that the total number of subcells for any cell $\Box \in G_i$ is $\tilde{O}(\mu_i)$. For brevity, the proof of Lemma 7.2 is included in Section 7.7, but the argument can be seen intuitively from the fact that the outermost ring of subcells along the boundary of $\Box_1$ has size $\tilde{O}(\mu_i)$. Furthermore, subcells increase in size as we move towards the center of $\Box_1$, implying that their count decreases geometrically.

**Lemma 7.2.** For any cell $\Box$ of $Q$ with level $i$, the total number of subcells is $\tilde{O}(\mu_i)$.

For some edge $(u, v)$, let $\Box$ be the least common ancestor of $u$ and $v$ in $Q$. Suppose that $\Box \in G_i$; then we say that the edge $(u, v)$ appears at layer $i$. The quadtree distance between $u$ and $v$ defined in [51] is given by the distance between the subcells $\xi_u$ and $\xi_v$ of $G[\Box]$ that
contain \( u \) and \( v \) respectively. As a result, the set of edges that appear at layer \( i \) can be represented using pairs of subcells from the set \( \bigcup_{\square' \in G_i} \mathbb{G} [\square'] \). However, the use of all pairs of subcells is prohibitively expensive. We further reduce the number of pairs of subcells by grouping them into a Well-Separated Pair Decomposition which we describe next.

7.2.4 Well-Separated Pair Decompositions

In this section, we extend Well-Separated Pair Decomposition (WSPD) that is commonly defined for points to approximate distances between pairs of subcells. A Well-Separated Pair Decomposition (WSPD) is a commonly used tool that, given a set \( P \) of \( n \) points, compactly approximate all \( O(n^2) \) distances between points of \( P \) by using a sparse set \( W \) of only \( \tilde{O}(n) \) well-separated pairs. Each pair \( (S, T) \in W \) consists of two subsets \( S, T \subseteq P \) of points. For any pair of points \( (u, v) \in P \times P \), there is a unique pair \( (S, T) \in W \) such that \( (u, v) \in S \times T \). For each pair \( (S, T) \), an arbitrary pair of representatives \( s \in S \) and \( t \in T \) can be chosen, and the distance between any pair \( (s', t') \in S \times T \) can be approximated using the distance between the representatives \( s \) and \( t \). This approximation will be of good quality so long as the pair \( (S, T) \) is well-separated, meaning the distance between any pair of points within \( S \) or within \( T \) is sufficiently small compared to the distance between any pair of points between \( S \) and \( T \).

For any parameter \( \varepsilon > 0 \), using the construction algorithm of [21], it is possible to build in \( \tilde{O}(n \text{poly}\{\log n, 1/\varepsilon\}) \) time a WSPD of the edges of \( A \times B \) where the costs of the edges belonging to any pair in the decomposition are within a factor of \( (1 + \varepsilon) \) of each other. Furthermore, if the ratio of the largest edge to smallest edge cost is bounded by \( n^{O(1)} \), then it can be shown that every point participates in only \( \text{poly}\{\log n, 1/\varepsilon\} \) pairs. Such a WSPD can be used to execute a single Hungarian search in near-linear time in the number of points.
7.2. Our Distance Function

Figure 7.1: (a) A division of $\Box$ into subcells. (b) Examples of a few possible WSPD pairs of $\mathcal{W}_\Box$. Every pair of subcells in different children of $\Box$ would be represented by some pair.

However, in order to execute a Hungarian search in sub-linear time, we must build a WSPD on the sub-linear number of subcells instead of the original points. Luckily, the algorithm of [21] can be applied in a straightforward fashion to generate a WSPD on subcells. Next, we describe the properties of our WSPD on subcells.

For any level $i$ cell $\Box$ of $Q$, consider two subsets of subcells, $S \subseteq G[\Box]$ and $T \subseteq G[\Box]$. We define $\ell_{\text{max}}(S,T) = \max_{\xi \in S, \xi' \in T} \ell_{\text{max}}(\xi, \xi')$. We say that $S$ and $T$ are $\varepsilon$-well separated if, for every pair of subcells $\xi \in S$ and $\xi' \in T$,

$$\ell_{\text{max}}(S,T) \leq (1 + \varepsilon/12)\ell_{\text{max}}(\xi, \xi').$$

(7.1)

For each cell $\Box$ let $\Box_1, \Box_2, \Box_3$ and $\Box_4$ be its four children. We precompute a WSPD $\mathcal{W}_\Box = \{(S_1, T_1), \ldots, (S_r, T_r)\}$, where $S_i \subseteq G[\Box], T_i \subseteq G[\Box]$ and $S_i, T_i$ are $\varepsilon$-well separated. Furthermore, for every pair of subcells $(\xi_1, \xi_2) \in G[\Box] \times G[\Box]$ (resp.$(\xi_2, \xi_1) \in G[\Box] \times G[\Box]$) where $\xi_1$
and $\xi_2$ are in two different children of $\Box$, there is a unique ordered pair in $(X, Y) \in W_{\Box}$ (resp. $(Y, X) \in W_{\Box}$) such that $\xi_1 \in X$ and $\xi_2 \in Y$. We denote the ordered pair $(X, Y) \in W_{\Box}$ that the pair of sub-cells $(\xi_1, \xi_2)$ maps to as $(S_{\xi_1}, T_{\xi_2})$. For notational convenience, we prefer that the pairs within the WSPD are ordered. Such an $\varepsilon$-WSPD can be constructed by executing a standard quadtree based construction algorithm presented in [21]. This algorithm uses the subtree of $Q$ rooted at $\Box$ to build the WSPD. Since we are interested in $\xi_1$ and $\xi_2$ that are contained inside two different children of $\Box$, we can trivially modify the algorithm of [21] to guarantee that every pair $(S_i, T_i)$ in the WSPD is such that the subcells of $S_i$ and the subcells of $T_i$ are contained in two different children of $\Box$. See Figure 7.1 for examples of WSPD pairs in $W_{\Box}$. Finally, the algorithm of [21] naturally generates unordered pairs. To ensure that every pair of subcells is covered by an ordered pair in the WSPD, for every pair $(X, Y) \in W_{\Box}$ generated by the algorithm, we add $(Y, X)$ to the $W_{\Box}$.

Next, we define terms that will be helpful in describing our data structure in Section 7.5. Any point $p \in A \cup B$ is contained inside one cell of each of the grids $G_i$ in $Q$. Let $\Box = \Box^i_p$ be the cell of $G_i$ that contains $p$. Let $\xi^i_p \in G[\Box]$ be the subcell that contains $p$. As a property of the WSPD construction algorithm, the decomposition $W_{\Box}$ ensures $\xi^i_p$ participates in $\tilde{O}(1)$ WSPD pairs of $W_{\Box}$. Let this set be denoted by $N^i(p)$. All edges of level $i$ incident on $p$ are represented by exactly one pair in $N^i(p)$. Since there are $O(\log n)$ levels, every edge incident on $p$ is represented by $\tilde{O}(1)$ WSPD pairs. We refer to these WSPD pairs as $N^*(p) = \bigcup_i N^i(p)$.

We can have a similar set of definitions for a subcell $\xi$ instead of a point $p$. Consider any cell $\Box \in G_i$ and a subcell $\xi \in G[\Box]$. Using a similar argument, we conclude that all edges of level $i$ incident on any vertex of $(A \cup B) \cap \xi$ are uniquely represented by $\tilde{O}(1)$ WSPD pairs denoted by $N^i(\xi)$. Furthermore, all edges of level $\geq i$ are uniquely represented by $N^*(\xi) = \bigcup_{j \geq i} N^j(\xi)$. Note that $|N^*(\xi)| = \tilde{O}(1)$. 


7.2. Our Distance Function

7.2.5 Distance Function

Given the definitions of subcells and the WSPDs, we can finally define the distance function. For \( p, q \in P \), let \( \Box \) be the least common ancestor of \( p \) and \( q \) in \( Q \) and let \( i \) be the level of \( \Box \). We denote the level of \( (p, q) \) to be the level of the least common ancestor of its end points, i.e., the level of \( \Box \). For some edge \( (p, q) \) with least common ancestor \( \Box \), let \( \xi_p \) and \( \xi_q \) be subcells from \( \mathbb{G}[\Box] \) that contain \( p \) and \( q \) respectively. Note that \( \xi_p \) and \( \xi_q \) are contained inside two different children of \( \Box \). There is a unique ordered representative pair \( (\Psi_p, \Psi_q) \in \mathcal{W}_\Box \) with \( \xi_p \in \Psi_p \) and \( \xi_q \in \Psi_q \). We set the distance between \( p \) and \( q \) to be

\[
  d_Q(p, q) = (\ell_{\text{max}}(\Psi_p, \Psi_q))^2.
\]

From the properties of our WSPD, if the unique representative pair of \( (p, q) \) is \( (X, Y) \), then the representative pair for \( (q, p) \) will be \( (Y, X) \), implying that our distance \( d_Q(\cdot, \cdot) \) is symmetric. For any subset \( E \subseteq A \times B \) of edges, we define its cost by \( d_Q(E) = \sum_{(a,b) \in E} d_Q(a, b) \). Since \( p \in \xi_p, q \in \xi_q \) and \( \xi_p \in \Psi_p, \xi_q \in \Psi_q \), we have

\[
  \|p - q\|^2 \leq (\ell_{\text{max}}(\xi_p, \xi_q))^2 \leq (\ell_{\text{max}}(\Psi_p, \Psi_q))^2 = d_Q(p, q).
\]  

(7.2)

Furthermore, it can be shown that if \( Q \) is a randomly shifted quad-tree, any optimal matching \( M_{\text{OPT}} \) with respect to the original squared-Euclidean costs satisfies

\[
  \mathbb{E}[d_Q(M_{\text{OPT}})] \leq (1 + \varepsilon) \cdot \sum_{(a,b) \in M_{\text{OPT}}} \|p - q\|^2.
\]

(7.3)

As noted before, we introduce an additional additive cost to all the edges. This additive cost on the edges is crucial in minimizing the number of data structure updates. Instead of proving equation (7.3), in Section 7.3, we introduce this additional additive costs as part
Chapter 7. An Approximation Algorithm for RMS Matching in a Plane of the dual feasibility conditions. We show that, to compute a \((1 + \varepsilon)\)-approximate RMS matching, it suffices to compute a feasible perfect matching.

**Note on Multisets**  Note that, for two points \(a, b\) that share the same location, we have defined \(d_Q(a, b) = 0\). In order to simplify the notations in the rest of this chapter, it is useful to extend some of the definitions above to points that have the same location. In particular, for any such \(a, b\), let \(\Box \in G_0\) be the node that contains \(a\) and \(b\).

- We define \(\Box\) as their least common ancestor,
- Let \(\xi \in G[\Box]\) be the subcell that contains \(a\) and \(b\). We set \(\xi_a = \xi_b = \xi\).
- We set the minimum subcell size \(\mu_0 = 0\) and also define \(\ell_{\max}(\xi, \xi) = \ell_{\min}(\xi, \xi)\) to be 0.
- Finally, we create exactly one pair \(\{(\xi, \xi)\}\) in \(W_\Box\).

It is straightforward to verify that the useful properties of the WSPD established earlier in this section will also hold for \(W_\Box\).

### 7.3 Dual Feasibility Conditions

In this section, we introduce a new set of feasibility conditions based on the randomly shifted quadtree. These feasibility conditions will allow our algorithm to find minimum-cost augmenting paths more efficiently. In order to describe this distance function, we partition the edges into a set of local edges and a set of non-local as described next. A similar definition of local and non-local edges was used in [51].
Figure 7.2: (a) A set of local edges between a WSPD pair of cells. Solid edges are in the matching, and dashed edges are not. (b) A local non-matching edge from $u \in B$ to $v \in A$ implies the existence of a length 3 alternating path $P = \langle u, u', v', v \rangle$ with net-cost $\phi(P) = d_Q(u, v)$.

Local and Non-Local Edges  For any two matching edges $(a, b) \in M$ and $(a', b') \in M$, we say that they belong to the same equivalence class if and only if they have the same least common ancestor $\Box$ and their ordered representative pairs in $W$ are the same, i.e., $(\Psi_a, \Psi_b) = (\Psi_{a'}, \Psi_{b'})$. Let $\mathbb{K}_M = \{M_1, ..., M_h\}$ be the resulting partition of matching edges into classes. For each $M_k$ for $1 \leq k \leq h$, let $A_k = \bigcup_{(a_j, b_j) \in M_k} a_j$ and $B_k = \bigcup_{(a_j, b_j) \in M_k} b_j$. The set $\{A_1, ..., A_h\}$ partitions the matched vertices of $A$ and $\{B_1, ..., B_h\}$ partitions the matched vertices of $B$. For any edge $(a, b) \in A \times B$, we say $(a, b)$ is local if $(a, b) \in A_k \times B_k$ for some $1 \leq k \leq h$. All other edges are non-local. We refer to the local edges (both non-matching and matching) of $A_k \times B_k$ as class $k$.

Next, we define a set of feasibility conditions based on the randomly-shifted quadtree. For a matching $M$ in the graph, $G(A \cup B, A \times B)$, we assign a dual weight $y(v)$ for every $v \in A \cup B$. Recall that $\mu_i$ is the minimum subcell size at level $i$ in the quadtree. For any edge $(a, b)$ of level $i$, let $\mu_{ab} = \mu_i$. We say that a matching $M$ and set of dual weights $y(\cdot)$ are $Q$-feasible
Chapter 7. An Approximation Algorithm for RMS Matching in a Plane

if for every edge \((a, b)\),

\[
y(a) + y(b) \leq d_Q(a, b) + \mu_{ab}^2. \tag{7.4}
\]

\[
y(a) + y(b) = d_Q(a, b) \quad \text{if} \ (a, b) \text{ is a local edge.} \tag{7.5}
\]

A \(Q\)-feasible perfect matching is a \(Q\)-optimal matching. Let \(M_{\text{OPT}}\) be the optimal RMS matching in \(G(A \cup B, A \times B)\). Similar to the Gabow-Tarjan [18] and Asathulla \textit{et al.} [6] algorithms, the addition of an additive error of \(\mu_{ab}^2\) for non-local edges distorts the cost of non-local edges of \(M_{\text{OPT}}\) by \(\mu_{ab}^2\). However, it can be shown that this additional error for any non-local edge \((a, b)\) of the optimal matching is, in expectation, much less than \(\varepsilon/2\|a - b\|^2\) due to the random shift. This follows from the fact that short edges of the optimal matching have a small probability of appearing at higher levels of the quadtree. By combining this argument with properties of the distance function, we can show the following lemma, whose proof is delayed until Section 7.7:

\textbf{Lemma 7.3.} For \(A, B \subset \mathbb{R}^2\), let \(M_{\text{OPT}}\) be the optimal RMS matching. For a parameter \(\varepsilon > 0\), given a randomly shifted quadtree \(Q\) and the distance \(d_Q(\cdot, \cdot)\), let \(M\) be any \(Q\)-optimal matching. Then,

\[
\mathbb{E}[w(M)] \leq (1 + \varepsilon/2) \sum_{(a, b) \in M_{\text{OPT}}} \|a - b\|^2.
\]

From Lemma 7.3, it follows that any \(Q\)-optimal matching is, in expectation, a \((1 + \varepsilon)\)-approximate RMS matching. Therefore, it suffices to design an efficient algorithm for computing a \(Q\)-optimal matching. By executing such an algorithm \(O(\log n)\) times, we can obtain a \((1 + \varepsilon)\)-approximate RMS matching with high probability (see Section 7.7).
7.4 Algorithm

Matching Preliminaries  For any matching $M$, an alternating path (resp. alternating cycle) with respect to $M$ is one which alternates between edges of $M$ and edges not in $M$. A vertex is free if it is not the endpoint of any edge of $M$ and matched otherwise. We use $A_F$ (resp. $B_F$) to denote the set of free vertices of $A$ (resp. $B$). An augmenting path $P$ is an alternating path between two free vertices. The matching $M' = M \oplus P$ has one higher cardinality than $M$. An alternating path $P$ is called compact if the largest contiguous set of local edges of $P$ has size at most 3 (see Figure 7.2). Throughout this chapter, we use the notation $a, a'$ and $a_j$ for $1 \leq j \leq n$ to denote points in $A$ and $b, b'$ and $b_j$ for $1 \leq j \leq n$ to denote points in $B$.

For any non-local edge $(a, b)$, we define its slack as $s(a, b) = d_Q(a, b) + \mu_{ab}^2 - y(a) - y(b)$, i.e., how far the feasibility constraint (7.4) for $(u, v)$ is from holding with equality. For all local edges the slack $s(a, b)$ is defined to be 0. Note that, for a $Q$-feasible matching, the slack on any edge is non-negative. We say any edge is admissible with respect to a set of dual weights if it has zero slack. The admissible graph is simply the subgraph induced by the set of zero slack edges. Note that all local edges are also admissible.

As is common, we define the residual graph $G_M$ of a matching $M$ by assigning directions to edges of the graph $G$. For any edge $(a, b) \in A \times B$, we direct $(a, b)$ from $a$ to $b$ if $(a, b) \in M$ and from $b$ to $a$ otherwise. For any $Q$-feasible matching, we construct a weighted residual graph $G'_M$ where the edges of the graph are identical to $G_M$ and each edge $(a, b)$ has a weight equal to $s(a, b)$. Any path in $G_M$ is alternating, and any path in $G_M$ that starts with a free vertex of $B_F$ and ends at a free vertex of $A_F$ is an augmenting path. Our algorithm will maintain a $Q$-feasible matching $M$ and set of dual weights $y(\cdot)$. Initially $M = \emptyset$, and we set $y(v) \leftarrow 0$ for every vertex $v \in A \cup B$; clearly, this initial dual assignment is $Q$-feasible. Similar to the
classical Hungarian algorithm, our algorithm will iteratively conduct a Hungarian search to find an augmenting path consisting only of admissible edges. Then the algorithm augments the matching along this path. This process repeats until a $Q$-optimal matching is found. Conducting a Hungarian search on the entire graph is prohibitively expensive. Therefore, we introduce a data structure that conducts Hungarian search and augment operations by implicitly modifying the dual weights in sub-linear time.

First, our algorithm executes in $\lceil 3 \log n/4 \rceil$ phases, starting with phase 0. At the end of the execution of these phases, it produces a matching that has $\tilde{O}(n^{1/4})$ free vertices. Finally, the algorithm matches the remaining free vertices one at a time by conducting a Hungarian search to find an augmenting path and then augmenting the matching along this path.

At the start of any phase $i \geq 1$, we are given a $Q$-feasible matching $M$ along with a set of dual weights such that every free vertex $b \in B_F$ has a dual weight of $\mu_{i-1}^2$. At the end of phase $i$, we obtain a $Q$-feasible matching with the dual weights of any free vertex $b \in B_F$ risen to $\mu_i^2$.

The data structure is used only during the execution of phases. After the $\lceil 3 \log n/4 \rceil$ phases have been executed, the algorithm will conduct explicit Hungarian searches and augmentations. For any phase $i \leq \lceil 3 \log n/4 \rceil$, we describe the data structure $D_i$. This data structure supports two global operations:

- **BUILD**: This operation takes as input a $Q$-feasible matching $M$ and a set of dual weights $y(\cdot)$ such that for every free vertex $v \in B_F$, the dual weight $y(v) = \mu_{i-1}^2$. Given $M, y(\cdot)$, the procedure builds the data structure.

- **GENERATE DuALS**: At any time in phase $i$, the execution of this procedure will return the matching $M$ stored by the data structure along with a set of dual weights $y(\cdot)$ such that $M, y(\cdot)$ is $Q$-feasible. We denote this matching as the associated $Q$-feasible
The total time taken by both of these operations is bounded by \( \tilde{O}(n\mu_i^{2/3}) \).

The data structure does not explicitly maintain a set of \( Q \)-feasible dual weights at all times because updating all the dual weights after each Hungarian search could take \( \Omega(n) \) time. Instead the data structure maintains a smaller set of ‘up-to-date’ dual weights, and updates other dual weights in a ‘lazy’ fashion. While a similar strategy was used in [6], applying the same strategy in our case requires the design of a new set of compressed feasibility conditions that are significantly more complex than the ones used in [6]. A set of up-to-date \( Q \)-feasible dual weights for all vertices could be recovered after any Hungarian search or augmentation by simply executing \textsc{GenerateDuals}. However, doing so is too expensive. Instead, the algorithm only executes \textsc{GenerateDuals} once at the end of every phase. Nonetheless, the \textsc{GenerateDuals} procedure guarantees the existence of a \( Q \)-feasible dual assignment for the matching \( M \). We use this associated \( Q \)-feasible matching to describe the other operations supported by the data structure.

During phase \( i \), we say that a cell \( \square \in G_i \) is active if \( \square \cap B_F \neq \emptyset \). The edges that go between active cells have a cost of at least \( \mu_i^2 \) and do not become admissible during phase \( i \) because the dual weights of all vertices of \( B \) are at most \( \mu_i^2 \) whereas the points of \( A \) have a non-positive dual weight. Therefore edges between active cells need not be considered during any Hungarian searches or augmentations of phase \( i \). As a result, each Hungarian search and augmentation can be conducted locally with respect to a single active cell.

During any phase \( i \) and for any active cell \( \square^* \in G_i \), our data structure supports the following operations:

- **Search**: This procedure conducts a Hungarian search. At the end of the search, either the dual weight of every free vertex \( b \in B_F \cap \square^* \) with respect to the associated
Figure 7.3: (a) The active cells of the current layer. Each full active cell (bold) has descendants in its active tree. (b) The pieces of a full active cell. (c) Each full piece has 4 descendants in the active tree. (d) Each branch of the active tree terminates with a sparse leaf cell.

- **Augment**: This procedure augments the matching along an augmenting path $P$ returned by \textsc{Search} and updates the data structure to reflect the new matching.

We postpone the implementation details of the four operations supported by this data structure until Section 7.5.

In order to describe the execution time of these procedures, we define \textit{active trees} next.

**Active Tree** We say a cell $□$ at level $i$ in the quadtree $Q$ is \textit{sparse} if $|(A \cup B) \cap □| \leq \mu^2_i$. Otherwise, $□$ is \textit{full}. For each active cell $□^*$ during phase $i$, we maintain an \textit{active tree} denoted by $\mathcal{T}_{□^*}$. The active tree $\mathcal{T}_{□^*}$ is rooted at $□$ and contains a subset of the nodes in the subtree of $□^*$ in $Q$. If $□^*$ is sparse, then $□^*$ is also a leaf node and the active tree contains only one node. Otherwise, if $□^*$ is full, let all cells of $G_{\lfloor 2i/3 \rfloor}$ that partition $□^*$ be the children of $□^*$ in the active tree. We refer to every child of $□^*$ in the active tree as a \textit{piece} of $□^*$. For each piece $□$ of $□^*$, if $□$ is sparse, then $□$ will become a leaf node of the active tree. Otherwise, if $□$ is full, then $□$ is an internal node of the active tree, and the four
children of □ in Q are also contained in the active tree. We recursively apply this process to
construct the active tree for each of the four children; each full child is decomposed into its
two children in Q. Every leaf node of T□∗ is a sparse cell and every internal node is a full
cell.

Consider any augmenting path P computed inside an active cell □∗ during phase i. Let
A(P) be the set of all cells of the active tree, excluding □∗, that contain at least one vertex
of P. We call such cells the affected cells of P. Let A_j(P) be the set of level j affected cells
of P. Then the time taken for a single execution of the Search procedure that returns an
augmenting path P is ˜O(μ^{8/3}_i + \sum_{j=0}^{[2i/3]} |A_j(P)|μ^3_j) and the time taken for an execution of
the Augment procedure on an augmenting path P is ˜O(\sum_{j=0}^{[2i/3]} |A_j(P)|μ^3_j).

Using these operations, we now present our algorithm for any phase 0 ≤ i < [3 log n/4]. At
the start of phase i ≥ 1, the dual weight of every vertex in BF is equal to μ^{2i}_{i-1}. We mark all
active cells as unprocessed. The algorithm for phase i conducts the following steps.

- Build the data structure D using the BUILD procedure.

- While there is an unprocessed active cell □∗,
  - Execute the Search procedure on □∗.
  - If Search returned an augmenting path P, then execute Augment on P.
  - If either BF ∩ □∗ = ∅ (i.e., □∗ is no longer active) or the dual weight of every
    vertex of BF ∩ □∗ is μ^{2i}_{i-1}, then mark □∗ as processed.

- Use GENERATEDUALS to obtain the associated Q-feasible matching M and the dual
  weights y(·).

After the execution of all [3 log n/4] phases, we match the remaining free vertices one at a
time by iteratively executing Hungarian search to find an augmenting path and augmenting
the matching $M$ along the path. Unlike during the phases, each Hungarian search is global and executed on $G_M$ without use of the data structure. We describe the details of this global Hungarian search next.

**Hungarian Search** We add a source vertex $s$ to the graph $G'_M$ and connect $s$ to each vertex $b \in B_F$ with a cost 0 edge. Then, we execute a Dijkstra search in the resulting graph, starting from $s$. For any point $u \in A \cup B$, let $d_u$ be the shortest path distance from $s$ to $u$ as computed by Dijkstra’s algorithm. We define as value $d$ as,

$$d = \min_{f \in A_F} d_f.$$

Next, for every $u \in A \cup B$ with $d_u \leq d$, we perform the following dual adjustment. If $u \in B$, we set $y(u) \leftarrow y(u) + d - d_u$, and if $u \in A$, we set $y(u) \leftarrow y(u) - d + d_u$. Using a straightforward and standard argument, it is easy to show that this dual adjustment maintains $Q$-feasibility. Furthermore, after the Hungarian search, $G_M$ contains an augmenting path $P$ consisting solely of admissible edges.

When implemented naively, the Hungarian search could take $\Omega(n^2)$ time. However, we recall that each vertex of $G_M$ is part of only $\tilde{O}(1)$ WSPD pairs. All edges $(u, v)$ with the same representative WSPD pair have the same value of $d_Q(u, v)$ as well as the same value of $\mu_{uv}^2$. Using this fact, Dijkstra’s algorithm can efficiently find the next edge to add to the shortest path tree in amortized $\tilde{O}(1)$ time per addition. As a result, a single Hungarian search can be executed in $\tilde{O}(n)$ time.

**Augment** Let $P$ be an admissible augmenting path found by the Hungarian search procedure. We describe how to augment $M$ along $P$ while maintaining $Q$-feasibility. First, we set $M \leftarrow M \oplus P$. This causes some non-local edges (potentially both matching and non-
matching) to become local. We must adjust the dual weights along $P$ to ensure that every newly introduced local edge $(u, v)$ satisfies the $Q$-feasibility constraint $y(u) + y(v) = d_Q(u, v)$. Let $(a, b) \in A_k \times B_k$ be an edge of $P$ that is local and in class $k$ edge after augmentation, but was non-local prior to augmentation. If there are no other class $k$ local edges after augmentation that were also local prior to augmentation, we set $y(a) \leftarrow y(a) - \mu_{ab}^2$. Otherwise, there must be at least one local edge $(a', b')$ in class $k$ after augmentation that was local prior to augmentation, and we set $y(a) \leftarrow y(a')$ and $y(b) \leftarrow y(b')$.

**Invariants** During the execution of the phases, the algorithm guarantees the following invariants:

(I1) The associated matching $M, y(\cdot)$ is $Q$-feasible.

(I2) The dual weight $y(b)$ of every vertex $b \in B$ is non-negative. Furthermore, in phase $i \geq 1$, the dual weight of every free vertex $b \in B_F$ is $\mu_{i-1}^2 \leq y(b) \leq \mu_i^2$. The dual weight of every vertex $a \in A$ is non-positive and for every free vertex $a \in A_F$, $y(a) = 0$.

Since each step of the algorithm is a call to the data structure $\mathcal{D}_i$, it suffices to show that $\mathcal{D}_i$ maintains these invariants. We do this in Section 7.5.

After the execution of the phases, the algorithm switches to conducting explicit Hungarian searches. Using a standard argument, it is easy to show that the dual updates of these Hungarian searches maintain $Q$-feasibility. However, the dual adjustments during augmentations are non-standard due to a careful handling of local edges. The following lemmas establish that the augmentation process continues to maintain $Q$-feasibility. Therefore, at the end of the algorithm, we produce a $Q$-optimal matching as desired.

**Lemma 7.4.** Any pair of local edges $(u, v) \in A \times B$ and $(u', v') \in A \times B$ of the same class $k$ are $Q$-feasible if and only if $y(u) = y(u')$ and $y(v) = y(v')$. 
Chapter 7. An Approximation Algorithm for RMS Matching in a Plane

Proof. It is sufficient to argue that claim is true for a pair of matching class $k$ edges, since any local non-matching edge has both its endpoints matched by a class $k$ matching edge. If $(u, v)$ and $(u', v')$ are matching edges of class $k$, then there must also be a pair of non-matching local edges $(v, u')$ and $(v', u)$. Since $(u, v)$ and $(v, u')$ are both feasible, we have

$$y(u) = d_Q(u, v) - y(v) = d_Q(v, u') - y(v) = y(u').$$

Similarly, since $(u', v')$ and $(v, u')$ are both feasible, we have

$$y(v') = d_Q(u', v') - y(u') = d_Q(v, u') - y(u') = y(v).$$

Lemma 7.5. For any vertex $v \in A \cup B$, let $y(v)$ (resp. $y'(v)$) be the dual weight of $v$ prior to (resp. after) augmentation. Then, $y'(v) \leq y(v)$.

Proof. The only dual weights that change are along $P$. Any vertex $u$ on $P$ must be matched to a vertex $v$ that is also on $P$ after augmentation. If $(u, v)$ is a local edge that was non-local prior to augmentation, and there are no edges that were in class $k$ both before and after augmentation, then the procedure sets $y'(u) = y(u) - \mu_{uv}^2$ if $u \in A$, and if $u \in B$, then its dual weight is left unchanged. It is easy to see that the dual weights of $u$ and $v$ do not increase for this case. Otherwise, there must be some other edge $(u', v')$ that is local and in class $k$ both before and after augmentation. Since $(u, v)$ was an admissible non-local edge prior to augmentation,

$$y(u) = d_Q(u, v) + \mu_{uv}^2 - y(v) \geq d_Q(v, u') - y(v),$$

and the dual weight of $u$ only decreases when it is set to match $y(u')$. 

$\square$
Lemma 7.6. Let $P$ be an admissible path with respect to a $Q$-feasible matching $M$ and set of dual weights $y(\cdot)$. Let $M', y'(\cdot)$ be the matching and set of dual weights after augmenting along $P$. Then $M', y'(\cdot)$ are $Q$-feasible.

Proof. Since the augmentation process only changes the dual weights of vertices of $P$, we only to consider edges that have at least one endpoint on $P$; edges disjoint from $P$ are unaffected. First, consider any local edge $(u, v)$ of class $k$ after augmentation that was non-local prior to augmentation. If, after augmentation, there are no other class $k$ local edges that also existed prior to augmentation, then either (i) $(u, v)$ is on $P$, or (ii) $(u, v)$ is not on $P$. For case (i), $(u, v)$ is on $P$, and $(u, v)$ is a matching edge after augmentation, $u \in A$, $v \in B$, and the procedure sets $y'(u) = y(v) - \mu_{uv}^2$. Prior to augmentation, there was an admissible non-local edge directed from $v$ to $u$, and we have $y(u) + y(v) = d_Q(u, v) + \mu_{uv}^2$. Therefore, after augmentation, $y'(u) + y'(v) = d_Q(u, v)$, and $(u, v)$ is feasible. Case (ii) can only occur if two previously non-local edges $(u, v')$ and $(u', v)$ simultaneously enter class $k$ by augmenting along $P$; in this case, it must be true that $y(u') = y(u)$ and $y(v') = y(v)$ and the edge $(u, v)$ is feasible.

Next, consider the case where there is at least one other class $k$ local edge after augmentation. At least one such edge $(u', v')$ must have been in class $k$ prior to augmentation as well. Then, for any newly created class $k$ local edge, the procedure ensures that $y'(u) = y(u')$ and $y'(v) = y(v')$. This implies that all class $k$ local edges are feasible by Lemma 7.4.

Finally, we argue that any non-local edge $(u, v)$ after augmentation is feasible. From Lemma 7.5, we have

$$y'(u) + y'(v) \leq y(u) + y(v) \leq d_Q(u, v) + \mu_{uv}^2.$$
7.4.1 Analysis of the Algorithm

In this section, we bound the time taken by the algorithm under the assumption that the data structure works as described. We begin by defining notations that will be used throughout the analysis. Let \( P = \langle P_1, \ldots, P_t \rangle \) be the \( t \) augmenting paths computed during the \( \lceil 3 \log n/4 \rceil \) phases of the algorithm. Let \( M_0 \) be the initial empty matching and, for any \( k \geq 1 \), let \( M_k \) be the matching obtained after augmenting the matching \( M_{k-1} \) along \( P_k \), i.e., \( M_k = M_{k-1} \oplus P_k \).

For any augmenting path \( P \) with respect to some matching \( M \), let \( N(P) \) be the set of non-local edges of \( P \).

The following Lemma establishes important properties of the algorithm during the \( \lceil 3 \log n/4 \rceil \) phases of the algorithm.

**Lemma 7.7.** The algorithm maintains the following properties during the \( \lceil 3 \log n/4 \rceil \) phases:

(i) The total number of free vertices remaining at the end of phase \( i \) is \( \tilde{O}(n/\mu_i^2) \), and,

(ii) \( \sum_{k=1}^{t} \sum_{(a,b) \in N(P_k)} \mu_{ab}^2 = \tilde{O}(n) \).

**Proof.** First, we consider phase 0. Clearly (i) holds because the number of unmatched vertices at the end of phase 0 is \( O(n) \). It is also easy to show that every augmenting path found during phase 0 contributes only \( O(1) \) to the total given by property (ii), and phase 0 contributes only \( O(n) \) towards (ii) over all its augmenting paths. Therefore, in the remaining arguments, we will assume that \( i \geq 1 \).

Let \( M_{OPT} \) be the minimum cost matching. The symmetric difference of \( M \) and \( M_{OPT} \) will contain \( \ell = n - |M| \) vertex-disjoint augmenting paths. Let \( \{P_1, \ldots, P_\ell\} \) be these augmenting paths. These augmenting paths contain some of the edges of \( M_{OPT} \). Combining this with
the $Q$-feasibility conditions gives,

$$
\sum_{(a,b) \in M_{\text{OPT}}} (d_Q(a, b) + \mu_{ab}^2) \geq \sum_{k=1}^{\ell} \left( \sum_{(a,b) \in P_i \setminus M} (d_Q(a, b) + \mu_{ab}^2) - \sum_{(a,b) \in P_i \cap M} d_Q(a, b) \right) \\
\quad \geq \sum_{k=1}^{\ell} \left( \sum_{(a,b) \in P_i \setminus M} (y(a) + y(b)) - \sum_{(a,b) \in P_i \cap M} (y(a) + y(b)) \right) \\
\quad \geq \ell \mu_{i-1}^2.
$$

The last inequality follows from the facts that, at the end of phase $i - 1$, the dual weight of every vertex in $B_F$ is at least $\mu_{i-1}^2$, the dual weight of every vertex of $A_F$ is 0, and all the vertices of any augmenting path except the first and last are the endpoint of exactly one matching edge and exactly one non-matching edge. At the beginning of phase $i$, since $\sum_{(a,b) \in M_{\text{OPT}}} (d_Q(a, b) + \mu_{ab}^2)$ is $\tilde{O}(n)$, the number of free vertices at the end of phase $i$ is $\tilde{O}(n/\mu_{i-1}^2)$.

Recollect that $\{P_1, \ldots, P_t\}$ are the augmenting paths computed by the algorithm. For any path $P_k$, let $b_k$ and $a_k$ be the two endpoints of this augmenting path and $y_k$ be the dual weight of $b_k$ when the augmenting path $P_k$ was found. Suppose $P_k$ was found in some phase $i$. Then, its dual weight $y_k \leq \mu_i^2 \leq 4\mu_{i-1}^2 \leq \tilde{O}(n)/(n - k)$. Summing over $1 \leq k \leq t$, we get

$$
\sum_{k=1}^{t} y_k = \tilde{O}(n).
$$
Note that because \( d_Q(M_0) = 0 \), we have,

\[
d_Q(M_t) + \sum_{k=1}^{t} \sum_{(a,b) \in P_k \setminus M_{k-1}} \mu_{ab}^2 \leq \sum_{k=1}^{t} d_Q(M_k) - d_Q(M_{k-1}) + \sum_{(a,b) \in P_k \cap M_{k-1}} \mu_{ab}^2 \\
\leq \sum_{k=1}^{t} \left( \sum_{(a,b) \in P_k \setminus M_{k-1}} (d_Q(a,b) + \mu_{ab}^2) - \sum_{(a,b) \in P_k \cap M_{k-1}} d_Q(a,b) \right) \\
= \sum_{k=1}^{t} \left( \sum_{(a,b) \in P_k \setminus M_{k-1}} y(a) + y(b) - \sum_{(a,b) \in P_k \cap M_{k-1}} y(a) + y(b) \right) \\
= \sum_{k=1}^{t} y_k = \tilde{O}(n).
\]

The claim then follows from the facts that \( d_Q(M_t) \geq 0 \) and \( N(P_k) \subseteq (P_k \setminus M_{k-1}) \).

Using Lemma 7.7, we can bound the efficiency of the algorithm. First, we bound the total time taken after the \( \lceil 3 \log n/4 \rceil \) phases have been executed. After the last phase is executed, \( \mu_i^2 = \Omega(n^{3/4}/\text{poly}\{\log n, 1/\varepsilon\}) \). From Lemma 7.7, there are only \( \tilde{O}(n^{1/4}) \) unmatched vertices remaining. Using the WSPD, each of these unmatched vertices are matched in \( \tilde{O}(n) \) time. Therefore, the time taken after the phases have executed is \( \tilde{O}(n^{5/4}) \).

Next, we bound the time taken by the \( \lceil 3 \log n/4 \rceil \) phases of the algorithm. For any such phase \( i \), we execute the Build procedure to create the data structure \( D_i \). At the end of phase \( i \), we execute the GenerateDuals procedure to generate a \( Q \)-feasible matching. Both of these operations take \( \tilde{O}(n \mu_i^{2/3}) \) time during phase \( i \).

Next, we bound the time taken by Augment, which takes \( \tilde{O}(\sum_{j=0}^{2i/3} |A(P)| \mu_j^3) \) time when executed on an augmenting path \( P \). Therefore, to bound the total time taken by Augment, we will bound the total number of level \( j \) edges over all augmenting paths computed during
7.4. Algorithm

phases of the algorithm. From Lemma 7.7, we have

$$\sum_{1 \leq k \leq t} \sum_{(u,v) \in \mathcal{N}(P_k)} \mu_{uv}^2 = \tilde{O}(n). \quad (7.6)$$

Each non-local edge \((u, v)\) of level \(j\) contributes \(\Omega(\mu_{uv}^2) = \Omega(\mu_j^2)\) towards the RHS of equation \((7.6)\). As a result, there can be at most \(\tilde{O}(n/\mu_j^2)\) such edges in all augmenting paths computed during the phases of the algorithm.

Recall that two matching edges \((a_i, b_i)\) and \((a_k, b_k)\) are in the same class if they share the least common ancestor \(\Box\) and their representative pair \((\Psi_{a_i}, \Psi_{b_i}) \in \mathcal{W}\) is the same as \((\Psi_{a_k}, \Psi_{b_k})\). Consider any augmenting path \(P\), and consider any maximal sub-path \(S\) with the property that all its matching edges (resp. non-matching edges) belong to the same class with representative pair \((\Psi, \Psi')\) (resp. \((\Psi', \Psi))\). We will call any such path a local path. Intuitively, all matching edges of \(S\) will belong to the same class and, upon augmentation, the non-matching edges of \(S\) will all enter the matching and belong to the same class. We say that \(S\) is a level \(j\) local path if all edges of \(S\) appear at level \(j\). For any augmenting path \(P\), let \(L_j(P)\) be the set of level \(j\) local paths of \(P\). It is easy to see that either: (1) \(S\) contains at least one non-local edge or (2) the first and the last edge of \(S\) are matching edges. In case (2), the number of matching edges that have \((\Psi, \Psi')\) as their representative pair decreases by 1 after augmenting along \(P\). Furthermore, new matching edges with representative \((\Psi, \Psi')\) can only be created through an occurrence of case (1). Therefore, each occurrence of case (2) can be taxed on an occurrence of case (1). Combining this observation with the bound on the number of non-local edges gives:

$$\sum_{1 \leq k \leq t} |L_j(P_k)| = \tilde{O}(n/\mu_j^2). \quad (7.7)$$

From the fact that each \(P_k\) is compact, there are at most \(6|L_j(P_k)| + 1\) cells in \(A_j(P_k)\). From
Figure 7.4: (a) A local path $S$ of length 3 between the cells $\Psi$ and $\Psi'$. Although $S$ does not contain any non-local edges, augmentation will reduce the number of matching edges between the two cells by 1. (b) The number of matching edges between the cells $\Psi$ and $\Psi'$ can only increase when a local path containing at least one non-local edge between $\Psi$ and $\Psi'$ participates in an augmenting path.

Lemma 7.7, there are $\tilde{O}(n/\mu^2_i)$ augmenting paths found during phase $i$. Combining these observations with (7.7) gives the following for any level $j$,

$$
\sum_{k=1}^{t} |A_j(P_k)|\mu^3_j = \tilde{O}(\mu^3_j(n/\mu^2_i + n/\mu^2_j)) = \tilde{O}(n\mu_j).
$$

When summing over all levels that contain affected pieces, the top level $\left\lfloor \frac{2i}{3} \right\rfloor$ dominates, which bounds the total time for AUGMENT during phase $i$ as,

$$
\sum_{k=1}^{t} \sum_{j=0}^{\left\lfloor \frac{2i}{3} \right\rfloor} |A_j(P_k)|\mu^3_j = \tilde{O}(n\mu^{2/3}_i). \quad (7.8)
$$

**Note Concerning Edges of Layer 0** Any non-local edge $(a, b)$ of layer 0 has a value $\mu_{ab} = 0$. As a result, a slightly different argument is needed to bound the total number of layer 0 edges used on augmenting paths. Observe that the edges preceding or following any local path of level 0 must be part of some level greater than 0. As a result, the total number of layer 0 edges used across all augmenting paths can be taxed on the total number
of non-layer 0 edges across all augmenting paths, which is $\tilde{O}(n)$.

Finally, we bound the time taken by the \textsc{Search} procedure during phase $i$. From Lemma 7.7, the number of unmatched vertices remaining at the beginning of phase $i$ is $O(n/\mu_i^2)$. This value also bounds the number of active cells during phase $i$. Therefore, the number of calls to \textsc{Search} during phase $i$ is $O(n/\mu_i^2)$. Each execution of \textsc{Search} during phase $i$ takes $\tilde{O}(\mu_i^{8/3} + \sum_{j=0}^{\lfloor 2i/3 \rfloor} |A_j(P)|\mu_j^3)$ time. Summing over all paths computed during phase $i$ and applying (7.8) gives a total time of $\tilde{O}(n\mu_i^{2/3})$ for the \textsc{Search} procedure during phase $i$. Combining the times taken by all data structure procedures during phase $i$ gives a total time of $\tilde{O}(n\mu_i^{2/3})$. The time taken for the last phase $\lceil 3 \log n/4 - 1 \rceil$ dominates, taking a total of $\tilde{O}(n^{5/4})$ time.

### 7.5 Data Structure

#### 7.5.1 Preliminaries

To simplify the presentation of the data structure, we introduce additional notations and give an equivalent redefinition of $Q$-feasibility with respect to these additional notations. We also present a few auxiliary properties that will be useful in proving the correctness of the data structure.

We define the \textit{adjusted} cost of an edge $\Phi(a, b)$ as

$$
\Phi(a, b) = \begin{cases} 
  d_Q(a, b) + \mu_{ab}^2 & \text{if } (a, b) \text{ is non-local.} \\
  d_Q(a, b) & \text{otherwise.}
\end{cases}
$$

For any edge $(a, b)$ of $G$, we define its \textit{net-cost} $\phi(a, b)$ as follows. If $(a, b)$ is non-matching edge,
its net-cost is $\phi(a, b) = \Phi(a, b)$. Otherwise, $(a, b)$ if the edge is in the matching, we define $\phi(a, b) = -\Phi(a, b)$. For any set of edges $S$, we define its net-cost as $\phi(S) = \sum_{(a,b) \in M} \phi(a, b)$.

Recollect that $Q$-feasibility was defined with respect to the graph $G(A \cup B, A \times B)$. For the data structure, it is convenient to deal with the residual graph $G_M$ instead. We redefine a $Q$-feasibility constraints that our algorithm maintains. This is done for simplicity in exposition of the algorithm and its proofs. All dual updates done during the course of the algorithm will ensure that each point $b \in B$ is assigned a non-negative dual weight $y(b)$ and each point $a \in A$ is assigned a non-positive dual weight $y(a)$.

A matching $M$ and a set of dual assignments $y(\cdot)$ is $Q$-feasible if for any edge $(u, v)$ of the residual graph $G_M$ directed from $u$ to $v$,

$$|y(u)| - |y(v)| \leq \phi(u, v),$$

$$|y(u)| - |y(v)| = \phi(u, v) \quad \text{if } (u, v) \text{ is local.}$$

For any non-local edge $(u, v)$, we define its slack as $s(u, v) = \phi(u, v) - |y(u)| + |y(v)|$, i.e., how far the feasibility constraint for $(u, v)$ is from being violated. Note that the slack on any edge is non-negative with local edges having a zero slack. We say any edge is admissible with respect to a set of dual weights it has a zero slack. The admissible graph is simply the subgraph induced by the set of zero slack edges. The advantage of redefining $Q$-feasibility conditions in this fashion is that it extends to any directed path in $G_M$ as presented in the following lemma.

**Lemma 7.8.** Let $M, y(\cdot)$ be a $Q$-feasible matching and set of dual weights maintained by the algorithm. Let $P$ be any alternating path with respect to $M$ starting at a vertex $u$ and ending at a vertex $v$. Then,

$$|y(u)| - |y(v)| + \sum_{(u', v') \in P} s(u', v') = \phi(P).$$
7.5. Data Structure

Proof. The proof is straightforward from the definitions of slack and net-cost:

\[\phi(P) = \sum_{(u',v') \in P} \phi(u',v') = \sum_{(u',v') \in P} (|y(u') - y(v')| + s(u',v')) = |y(u) - y(v)| + \sum_{(u',v') \in P} s(u',v').\]

For any phase \(i\), we define our data structure for each active cell \(\Box^* \in G_i\). The data structure is based on the active tree \(T_{\Box^*}\). We define a sub-linear in \(n\) sized associated graph \(A\mathcal{G}_\Box\) for each cell \(\Box\) of \(T_{\Box^*}\). This graph will help us compactly store the dual weights and help conduct the Hungarian Search, find augmenting paths and augment the matching along the path.

For any cell \(\Box\), let \(A_\Box = A \cap \Box\) and \(B_\Box = B \cap \Box\). For any set of cells \(X\), we denote \(A_X = \bigcup_{\Box \in X} A_\Box\) and \(B_X = \bigcup_{\Box \in X} B_\Box\). For any cell \(\Box\) let \(M_\Box\) be the set of edges of \(M\) that have both endpoints contained in \(\Box\), and let \(\mathcal{G}_{M_\Box}\) be the vertex-induced subgraph of \((A_\Box \cup B_\Box)\) on \(\mathcal{G}_M\). For simplicity in notation we use \(\mathcal{G}_\Box\) to denote \(\mathcal{G}_{M_\Box}\).

Recall that a \(\Box\) of level \(j\) is sparse if \(|A_\Box \cup B_\Box| \leq \mu_j^2\) and full otherwise. For any cell \(\Box\) of \(T_{\Box^*}\), our data structure constructs an associated graph \(A\mathcal{G}_\Box\). If \(\Box\) is sparse, the associated graph \(A\mathcal{G}_\Box\) is simply given by \(\mathcal{G}_\Box\). In the following, we define the associated graph for any full cell.

7.5.2 Vertices of the Associated Graph

We define an associated graph for an arbitrary cell in the active tree \(T_{\Box^*}\). For any cell \(\Box\) in the active tree, let \(D(\Box)\) denote the children of \(\Box\). We extend our definition to subcells as well. For any subcell \(\xi \in \mathcal{G}[\Box]\), let \(\Box' \in D(\Box)\) be the cell that contains \(\xi\). Let \(D(\xi) = \{\xi' \mid \xi' \in \mathcal{G}[\Box']\}\) and \(\xi' \subseteq \xi\).

If \(\Box\) is a full cell, for each of its children \(\Box'\), suppose \(\Box'\) is of level \(j\). We cluster \(A_{\Box'} \cup B_{\Box'}\)
Chapter 7. An Approximation Algorithm for RMS Matching in a Plane

Figure 7.5: Internal and boundary clusters of a subcell $\xi$ within the child $\square'$ of $\square$. (a) All vertices matched within $\square'$ are part of an internal cluster. (b) All vertices matched outside of $\square'$ are divided into boundary clusters based on WSPD pairs at higher levels.

into $\tilde{O}(\mu_j)$ clusters. We cluster points in such a way that all edges going between any two clusters $X$ and $Y$, where $X$ and $Y$ are clusters for two different children $\square'$ and $\square''$ of $\square$, have the same net-cost. We create one vertex in $V_{\square}$ for every cluster of $\square'$ and repeat this for every child $\square'$ of $\square$. The clusters created here are similar to that in [51].

Recall that two matching edges $(a_i, b_i)$ and $(a_k, b_k)$ are in the same class if they share the least common ancestor $\square$ and their representative pair $(\Psi_{a_i}, \Psi_{b_i}) \in \mathcal{W}_{\square}$ is the same as $(\Psi_{a_k}, \Psi_{b_k})$.

For any matched point $a_i$ (resp. $b_i$), we refer to $b_i$ (resp. $a_i$) as its partner point. For any $\xi \in \mathbb{G}[\square']$, we partition $A_\xi$ and $B_\xi$ into three types of clusters.

- **Free clusters**: All free points of $A_\xi$ (resp. $B_\xi$) belong to a single cluster

\[ A_\xi^F = A_F \cap \xi, \quad B_\xi^F = B_F \cap \xi. \]
• **Internal clusters**: All points of $A_\xi$ (resp. $B_\xi$) whose partner point is also inside $\Box'$ belong to a single cluster

$$A^I_\xi = \{a_i \in A_\xi \mid (a_i, b_i) \in M, b_i \in B_{\Box}\},$$

$$B^I_\xi = \{b_i \in B_\xi \mid (a_i, b_i) \in M, a_i \in A_{\Box}\}.$$

• **Boundary clusters**: Recollect that $\Box' \in G_j$. All points of $A_\xi$ (resp. $B_\xi$) whose partner points are outside $\Box'$ are partitioned into boundary clusters. Two such vertices belong to the same boundary cluster if the matching edges incident on them belong to the same class. Note that all such matching edges have level at least $j$ and are incident on at least one vertex of $A_\xi \cup B_\xi$. Any such matching edges is captured by one of $\tilde{O}(1)$ many WSPD pairs given by the set $N^*(\xi)$. Since there is at most one class per WSPD pair, there are at most $\tilde{O}(1)$ many boundary clusters per subcell. More specifically, for every $(\Psi_1, \Psi_2) \in N^*(\xi)$, we create a cluster,

$$A^{(\Psi_1, \Psi_2)}_\xi = \{a_i \in A_\xi \mid (a_i, b_i) \in M, b_i \in B_{\Psi_2}\},$$

$$B^{(\Psi_1, \Psi_2)}_\xi = \{b_i \in B_\xi \mid (a_i, b_i) \in M, a_i \in A_{\Psi_2}\}.$$

For every cell $\Box'$ and any subcell $\xi \in G[\Box']$, there are a total of $\tilde{O}(1)$ clusters. Therefore, the total number of clusters at $\Box'$ is $\tilde{O}(\mu_j)$. Let $X_{\Box'}$ be the set of clusters at $\Box$ that are generated from its child $\Box'$. The cluster set at $\Box$ is simply $V_{\Box} = \bigcup_{\Box' \in D(\Box)} X_{\Box'}$. We use $A_{\Box}$ (resp. $B_{\Box}$) to denote the vertices of type $A$ (resp. type $B$) in $V_{\Box}$; $V_{\Box} = A_{\Box} \cup B_{\Box}$.

Next, we partition the clusters in $X_{\Box'}$ into two subsets called the *entry* and *exit* clusters.
respectively,

\[
X_{\rightarrow} = \{ B_{\xi}^{F}, A_{\xi}^{I}, B_{\xi}^{(\Psi_1, \Psi_2)} | \xi \in G[\Box'], (\Psi_1, \Psi_2) \in N^*(\xi) \},
\]

\[
X_{\leftarrow} = \{ A_{\xi}^{F}, B_{\xi}^{I}, A_{\xi}^{(\Psi_1, \Psi_2)} | \xi \in G[\Box'], (\Psi_1, \Psi_2) \in N^*(\xi) \}.
\]

We also denote all the clusters at \( \Box \) from \( \Box' \in D(\Box) \) that contains points of \( A \) and \( B \) as \( A_{\rightarrow} \) and \( B_{\rightarrow} \) respectively; \( X_{\rightarrow} = A_{\rightarrow} \cup B_{\rightarrow} \). We next describe the significance of entry and exit clusters. For any directed path \( \Pi \) in \( G_{\rightarrow} \), let \( \overline{\Pi} \) be a maximal connected sub-path of \( \Pi \) that lies inside \( \Box \). Suppose \( \overline{\Pi} \) contains at least one edge. For the two endpoints \( p, q \) of \( \overline{\Pi} \), we refer to \( p \) as entry and \( q \) as exit point if \( \overline{\Pi} \) is directed from \( p \) to \( q \). Then, it was shown in [51] that the entry point lies in an entry cluster and exit point lies in an exit cluster.

### 7.5.3 Relating parent-child clusters

Let \( \Box \) be any node in \( T_{\rightarrow} \). Note that all internal nodes of the active tree except the root have four children. For any cell \( \Box \) of an active tree \( T_{\rightarrow} \), clusters are defined with respect to the subcells of its children. For any cell \( \Box \) of \( T_{\rightarrow} \), including the root, let \( \xi \) be a subcell of \( \Box' \in D(\Box) \) and let \( \Box \) be a cell of level \( i \). Then, we get the following relationship between clusters of \( \Box \) and \( \Box' \).

\[
A_{\xi}^{F} = \bigcup_{\xi' \in D(\xi)} A_{\xi'}^{F}, \quad B_{\xi}^{F} = \bigcup_{\xi' \in D(\xi)} B_{\xi'}^{F},
\]

\[
A_{\xi}^{(\Psi_1, \Psi_2)} = \bigcup_{\xi' \in D(\xi)} A_{\xi'}^{(\Psi_1, \Psi_2)}, \quad B_{\xi}^{(\Psi_1, \Psi_2)} = \bigcup_{\xi' \in D(\xi)} B_{\xi'}^{(\Psi_1, \Psi_2)},
\]

\[
A_{\xi}^{I} = \bigcup_{\xi' \in D(\xi)} (A_{\xi'}^{I} \cup \bigcup_{(\Psi', \Psi') \in N^i(\xi')} A_{\xi'}^{(\Psi, \Psi')}),
\]

\[
B_{\xi}^{I} = \bigcup_{\xi' \in D(\xi)} (B_{\xi'}^{I} \cup \bigcup_{(\Psi', \Psi') \in N^i(\xi')} B_{\xi'}^{(\Psi, \Psi')}).
\]
Figure 7.6: Examples of edges of $E_\Box$. Internal edges (solid) represent shortest paths between clusters within a child of $\Box$. Bridge edges (dashed) are between clusters in two different children of $\Box$.

For any cluster $X$ defined at a full cell $\Box$, we use the notation $D(X)$ to denote all the clusters at the children that combine to form $X$. Note that if $X$ is a cluster generated at a subcell $\xi$ of a leaf (i.e., sparse) cell $\Box' \in D(\Box)$ of $\mathcal{T}_\Box$, then we set $D(X)$ to be all the points that are contained in $X$. The following lemma whose proof is straightforward states the property of the above hierarchical clustering scheme.

**Lemma 7.9.** For any cell $\Box \in Q$, let $\Box_1, \Box_2$ be two of its children. Let $X \in X_{\Box_1}$ and $Y \in X_{\Box_2}$. Then the net-costs of all edges in $X \times Y$ are the same in $\mathcal{G}_M$ and all such edges are oriented in the same direction — either all are oriented from $B$ to $A$ or all of them are oriented from $A$ to $B$. 
7.5.4 Edges of the Associated Graph

Given a full cell $\square$, we already defined the vertex set $V_{\square}$ for the associated graph. We have the following types of edges in the edge set $E_{\square}$ of the associated graph.

- **Internal Edges:** For any child $\square'$ of $\square$, we add edges from $X$ to $Y$ provided $X \in X_{\square'}^\downarrow$ is an entry cluster of $\square'$ and $Y \in X_{\square'}^\uparrow$ is an exit cluster of $\square'$.

- **Bridge Edges:** For any children $\square' \neq \square''$ of $\square$, for any two clusters $X$ and $Y$ where $X \in X_{\square'}$ and $Y \in X_{\square''}$, suppose $X \in B_{\square'}$ and $Y \in A_{\square''}$. We add an edge directed from $Y$ to $X$ (resp. $X$ to $Y$) if, for every edge $(x, y) \in X \times Y$, $(x, y)$ is a local (resp. non-local) edge. We continue to refer to such edges of the associated graph as local (resp. non-local) edges.

**Bridge Edge Costs** Note that for any local bridge edge from cluster $X$ to $Y$ there is at least one matching edge say $(x, y) \in X \times Y$. We set the cost of $(X, Y)$, denoted by $\phi(X, Y)$ to $\phi(x, y)$. For a non-local bridge edge $(X, Y)$, every edge $(x, y) \in (X \times Y)$ has the same net-cost, which defines the net-cost of $(X, Y)$, i.e., $\phi(X, Y) = \phi(x, y)$. Next, we describe the cost of an internal edge.

**Internal Edge Costs** For any child $\square'$ of $\square$, and any internal edge $(X, Y) \in (X_{\square'}^\downarrow \times X_{\square'}^\uparrow)$ in $E_{\square}$, we define its projection $P(X, Y)$. If $\square'$ is sparse, then $P(X, Y)$ is a minimum net-cost path in $G_{\square'}$ from any $x \in X$ to any $y \in Y$. Otherwise, $\square'$ is full, and the projection $P(X, Y)$ is a minimum net-cost path through $A_{\square'}$ from any $X' \in D(X)$ to any $Y' \in D(Y)$. In either case, the net-cost of $(X, Y)$ is equal to the net-cost of its projection; i.e., $\phi(X, Y) = \phi(P(X, Y))$. The following lemma, which follows from a simple induction on the recursive definition of projection, states that any internal edge $(X, Y) \in (X_{\square'}^\downarrow \times X_{\square'}^\uparrow)$ corresponds to a
minimum net-cost path from $X$ to $Y$ in $G_{□'}$.

**Lemma 7.10.** For any $u, v \in □$ let $Π_{u,v,□}$ be a minimum net-cost alternating path in $G_{□'}$ from $u$ to $v$. For any internal edge $(X, Y) \in (X_{□}^{↓}, X_{□}^{↑})$, consider $(x, y) = \arg\min_{(x', y') \in X \times Y} ϕ(Π_{x',y',□})$. Then $ϕ(X, Y) = ϕ(Π_{u,v,□})$.

### 7.5.5 Compressed Feasibility

Consider any active cell $□^*$ of the quadtree and the active tree $T_{□^*}$ rooted at $□^*$. Consider an assignment of dual weights $y(·)$ to the vertices of $V_{□}$ for all cells $□ \in T_{□^*}$. We say that $M_{□^*}$ along with these dual weights are compressed feasible if for every cell $□$ in $T_{□^*}$.

(C1) For every edge directed from $X$ to $Y$ in $A GX_{□}$,

\[
|y(X)| - |y(Y)| \leq ϕ(X, Y),
\]

\[
|y(X)| - |y(Y)| = ϕ(X, Y) \quad \text{if } (X, Y) \text{ is local with respect to } M_{□}.
\]

(C2) If $□$ is full, then for each exit cluster $X \in X_{□}^{↑}$, for any $X' \in D(X)$, $|y(X')| \leq |y(X)|$.

Note that if $□$ is sparse, then we are at a leaf node of the active tree and only condition (C1) applies. Condition (C1) implies that $M_{□}$ and $y(·)$ are $Q$-feasible when $□$ is sparse.

We define slack of any edge (bridge or internal) directed from $X$ to $Y$, denoted by $s(X, Y)$, as $ϕ(X, Y) - |y(X)| + |y(Y)|$. From (C1), it follows that the slack of any edge is non-negative. We define a slack-weighted associated graph, denoted by $A G'_{□}$, to be identical to the associated graph $A G_{□}$, but where the weight of any edge $(X, Y)$ is its slack $s(X, Y)$.

We introduce two procedures, namely **Sync** and **Construct**. Both these procedures will be used to support **Build**, **GenerateDuals**, **Search** and **Augment** operations.
7.5.6 The Construct Procedure

In this section, we present a procedure called Construct, which will be used to compute the internal edges of an associated graph. The Construct procedure accepts a cell $\Box'$, such that $\Box' \neq \Box^*$ and $\mathcal{AG}_{\Box'}$ has already been computed, along with dual weights $y(\cdot)$ for all vertices of $V_{\Box'}$. It assumes that $M_{\Box'}, y(\cdot)$ satisfy the compressed feasibility conditions. Let $\Box$ be the parent of $\Box'$ in $\mathcal{T}_{\Box'}$. The procedure computes the internal edges of $\mathcal{X}_{\Box'}^\downarrow \times \mathcal{X}_{\Box'}^\uparrow$ in $\mathcal{AG}_{\Box}$. It also assigns dual weights to the vertices of $V_{\Box}$ that correspond to clusters generated for the subcells of $\Box'$.

We describe the process for building the internal edges going out of each cluster $X \in \mathcal{X}_{\Box'}^\downarrow$. We add an additional vertex $s$ to $\mathcal{AG}_{\Box'}'$ and add an edge from $s$ to each cluster $X' \in D(X)$ with a cost equal to $|y(X')|$. After creating this augmented associated graph, we simply execute Dijkstra’s algorithm from $s$ to find the shortest path distance from $s$ to every node in $V_{\Box'}$. Let $d_v$ denote the shortest path distance from $s$ to $v$ in the augmented associated graph. For each exit cluster $Y \in \mathcal{X}_{\Box'}^\uparrow$, we create an internal edge from $X$ to $Y$ in $\mathcal{AG}_{\Box}$ and set its cost to be $\min_{Y' \in D(Y)} (d_{Y'} - |y(Y')|)$. We repeat this procedure for each entry cluster.

This completes the description how to construct the internal edges of $\mathcal{AG}_{\Box}$ in $\Box'$. We next assign dual weights to each cluster $X \in \mathcal{X}_{\Box'}$ as follows: If $X$ is an entry cluster, let $X' = \arg\min_{Y \in D(X)} |y(Y)|$. Otherwise, $X$ is an exit cluster, and we let $X' = \arg\max_{Y \in D(X)} |y(Y)|$. In either case, we set $y(X) \leftarrow y(X')$.

The following lemma shows that the Construct procedure correctly assigns the net-cost of edges in $\mathcal{X}_{\Box'}^\downarrow \times \mathcal{X}_{\Box'}^\uparrow$.

**Lemma 7.11.** Let $X \in \mathcal{X}_{\Box'}^\downarrow$ and $Y \in \mathcal{X}_{\Box'}^\uparrow$ be a pair of clusters of $\Box'$ that form an internal edge $(X, Y) \in E_{\Box}$. Then the Construct procedure ensures that $\phi(X, Y) = \phi(P(X, Y)))$. 


Proof. Consider any \( X' \in D(X) \) and \( Y' \in D(Y) \); let \( P_{X',Y'} \) be the minimum net-cost path from \( X' \) to \( Y' \) in \( \mathcal{A} \mathcal{G}_{\square'} \). For any edge \((u, v) \in P_{X',Y'}\), from the definition of slack, we have \(|y(u)| - |y(v)| + s(u, v) = \phi(u, v)\). When summing over all \((u, v) \in P_{X',Y'}\), we get that each vertex of the path except the first or last vertex has a net-contribution of 0 to the dual weight magnitude total. From the definition of projection,

\[
|y(X')| - |y(Y')| + \sum_{(u, v) \in P_{X',Y'}} s(u, v) = \sum_{(u, v) \in P_{X',Y'}} \phi(u, v). \tag{7.9}
\]

When executing a Dijkstra search from \( X \), the \textsc{Construct} procedure assigns,

\[
\phi(X, Y) = \min_{Y' \in D(Y)} d_{Y'} - |y(Y')| - \sum_{(u,v) \in P_{X',Y'}} s(u, v) + \sum_{(u,v) \in P_{X',Y'}} \phi(u, v)
\]

\[
= \min_{X' \in D(X), Y' \in D(Y)} |y(X')| - |y(Y')| + \sum_{(u,v) \in P_{X',Y'}} s(u, v)
\]

\[
= \phi(P(X, Y)).
\]

Therefore, the \textsc{Construct} procedure correctly computes \( \phi(X, Y) \).

In the following Lemma, we argue that the internal edges of \( \square' \) in \( \mathcal{A} \mathcal{G}_{\square} \) are feasible after \textsc{Construct} is called on \( \square' \).

**Lemma 7.12.** After \textsc{Construct} is called on a cell \( \square' \) with parent \( \square \), then, for every internal edge \((X, Y) \in X_{\square'} \times X_{\square'} \) of \( E_{\square} \), we have, \(|y(X)| - |y(Y)| \leq \phi(X, Y)|.

**Proof.** The edge \((X, Y)\) has some projection \( P(X, Y) \); let \( X' \in D(X) \) (resp. \( Y' \in D(Y) \)) be
the first (resp. last) vertex of $P(X, Y)$. From equation (7.9) and the feasibility of $A\mathcal{G}_{\square}$, we have,

$$|y(X')| - |y(Y')| \leq \phi(P(X, Y)) = \phi(X, Y).$$

From the way the CONSTRUCT procedure assigns dual weights to $X$ and $Y$, we have that $|y(X)| \leq |y(X')|$ and $|y(Y')| \leq |y(Y)|$. Therefore, $|y(X)| - |y(Y)| \leq \phi(X, Y)$. \hfill \qed

**Efficiency of Construct** Next, we bound the time taken for a single call to CONSTRUCT on a cell $\square \in \mathcal{T}_{\square^*}$. Assume that $\square$ appears at level $j$. The CONSTRUCT procedure executes a Dijkstra search from each of the $|\mathcal{X}_\square| = \tilde{O}(\mu_j)$ clusters of $\square$. If $\square$ is full, then each Dijkstra search takes $\tilde{O}(|E_\square|) = \tilde{O}(|V_\square|^2) = \tilde{O}(\mu_j^2)$ time. If $\square$ is sparse, then each Dijkstra search can be executed efficiently in $\tilde{O}(|\mathcal{X}_\square| + |A_\square \cup B_\square|)$ time using the fact that the edges of $\mathcal{G}_M$ outgoing from any vertex $v$ belong to only $\tilde{O}(1)$ different WPSD pairs; the same technique was used for the Hungarian search in Section 7.4. Since $\square$ is sparse, $|A_\square \cup B_\square| \leq \mu_j^2$, and each Dijkstra search takes $\tilde{O}(\mu_j^2)$ time. The CONSTRUCT procedure executes $|\mathcal{X}_\square^k| = \tilde{O}(\mu_j)$ Dijkstra searches, and each Dijkstra search takes $\tilde{O}(\mu_j^2)$ time, so the total time taken by CONSTRUCT is $\tilde{O}(\mu_j^3)$ for any cell of level $j$. This gives the following Lemma.

**Lemma 7.13.** Any execution of CONSTRUCT on a cell $\square$ of layer $j$ takes $\tilde{O}(\mu_j^3)$ time. Furthermore, if $\square$ is sparse, the time taken can be bounded by $\tilde{O}(\mu_j(\mu_j + |A_\square| + |B_\square|))$.

### 7.5.7 Sync Procedure

For an active cell $\square^*$ and a compressed feasible matching $M_{\square^*}$ along with a set of dual weights $y(\cdot)$, the SYNC procedure takes the updated dual weights on clusters of $\mathcal{X}_\square$ at any non-root cell $\square \in \mathcal{T}_{\square^*}$ and uses them to update the dual weights of $V_\square$ such that the matching continues to be compressed feasible, and,
(T1) For any entry cluster $X \in \mathbb{X}_{\Box}$, and for any $X' \in D(X)$, $|y(X')| \geq |y(X)|$.

(T2) For any free or boundary cluster $X \in \mathbb{X}_{\Box}$, and for any $X' \in D(X)$, $y(X') = y(X)$.

The Sync procedure consists of executing the following algorithm for each entry cluster $X \in \mathbb{X}_{\Box}$: We create a new vertex $s$ and add an edge from $s$ to each vertex $X' \in D(X)$. We assign a weight $|y(X')|$ to the edge from $s$ to $X'$. Then, we execute Dijkstra’s algorithm starting from $s$. Let $d_v$ be the shortest path distance from $s$ to $v$ as computed by Dijkstra’s algorithm. For any vertex $v$ with $d_v < |y(X)|$, if $v \in B_{\Box}$ we update the dual weight to $y(v) \leftarrow y(v) + |y(X)| - d_v$. Otherwise, $v \in A_{\Box}$, and we update the dual weight $y(v) \leftarrow y(v) - |y(X)| + d_v$. Note that in both cases the magnitude of the dual weight increases by $|y(X)| - d_v$. The dual weight of every other vertex with $d_v \geq |y(X)|$ does not change. This completes the description of the algorithm initiated with respect to $X$.

Note that for any cluster $X' \in D(X)$ if $|y(X')| \geq |y(X)|$, then the procedure will only further increase the magnitude of $y(X')$ and so, (T1) holds. If, on the other hand, $|y(X')| < |y(X)|$, then the length of the edge from $s$ to $X'$ is $|y(X')|$, and so the shortest path distance $d_{X'} \leq |y(X')| < |y(X)|$. The magnitude of the dual weight of $X'$ increases by $|y(X)| - d_{X'} \geq |y(X)| - |y(X')|$ implying that the new magnitude of $y(X')$ is at least the magnitude of $y(X)$. Therefore (T1) holds for any entry cluster.

After we execute this for all entry clusters, we perform the following dual adjustment: For any $X \in \mathbb{X}_{\Box}$, and for any $X' \in D(X)$, we will explicitly update the dual weight $y(X')$ to match $y(X)$. Therefore, (T2) holds after the execution of Sync.

To prove the correctness of Sync it remains to show that the updated dual weights satisfy compressed feasibility.

**Lemma 7.14.** For any compressed feasible matching $M_{\Box^*}$ on an active cell $\Box^*$, after the execution of Sync at $\Box \in \mathcal{T}_{\Box^*}$, the updated dual weights of $V_{\Box}$ will continue to satisfy (C1)
and (C2).

**Proof.** Assume that the claim holds prior to executing Dijkstra’s algorithm from some vertex $X \in X_D$. We argue that (C1) and (C2) continue to hold after executing the algorithm from $X$. Let $y(\cdot)$ (resp. $y'(\cdot)$) be the dual weights after (resp. before) executing this process with respect to $X$. Let $s(u, v)$ be the slacks with respect to $y(\cdot)$.

We begin by arguing that if $(u, v)$ is local, and $d_u \neq \infty$, then $d_u = d_v$. First, consider if $\square$ is full. Then, the only incoming edge to $v$ is via $u$. Therefore, if $u$ is reached during the Dijkstra search, $d_u = d_v$. Next, consider if $\square$ is sparse. Then, if $(u, v) \in M$, $(u, v)$ is the only edge incoming to $v$, so clearly if $u$ is reached during the search, then $d_u = d_v$. Next, consider if $(u, v) \notin M$. Then, since $(u, v)$ is local, $u \in B$ must be matched to some vertex $v' \in A$ and $v \in A$ must be matched to some vertex $u' \in B$. Furthermore, there must be a non-matching local edge from $u'$ to $v'$. If $u$ is reached, then, since $(u, v)$ has 0 slack, $d_v \leq d_u$. Furthermore, there is a path $\langle v, u', v', u \rangle$ from $v$ to $u$ consisting solely of local edges, which all have 0 slack. Therefore, $d_u \leq d_v$, and we get that $d_u = d_v$.

Next, we show (C1). Consider any edge $(u, v) \in A_G$. We consider four cases:

- $d_u, d_v \geq |y(X)|$: In this case, the dual weights of $u$ and $v$ are unchanged, so the edge remains feasible with respect to (C1).

- $d_u, d_v < |y(X)|$: We first observe that, from the definition of shortest paths, $d_v - d_u \leq$
7.5. Data Structure

$s(u, v)$. If $(u, v)$ is non-local, then,

$$|y'(u)| - |y'(v)| = |y(u)| + (|y(X)| - d_u) - (|y(v)| + (|y(X)| - d_v))$$

$$= |y(u)| - |y(v)| + d_v - d_u$$

$$\leq |y(u)| - |y(v)| + s(u, v)$$

$$= \phi(u, v),$$

and the edge $(u, v)$ satisfies (C1). Furthermore, if $(u, v)$ is local, then $d_v - d_u = 0 = s(u, v)$, and the same equation holds with equality, i.e., $|y'(u)| - |y'(v)| = \phi(u, v)$, which satisfies (C1).

- $d_u < |y(X)|$ and $d_v \geq |y(X)|$: Since $d_u \neq d_v$, $(u, v)$ is non-local. We have:

$$|y'(u)| - |y'(v)| = |y(u)| + (|y(X)| - d_u) - |y(v)|$$

$$\leq |y(u)| - |y(v)| + d_v - d_u$$

$$\leq |y(u)| - |y(v)| + s(u, v)$$

$$= \phi(u, v),$$

and (C1) is satisfied.

- $d_u \geq |y(X)|$ and $d_v < |y(X)|$: Since $d_u \neq d_v$, $(u, v)$ is non-local. We have:

$$|y'(u)| - |y'(v)| = |y(u)| - (|y(v)| + (|y(X)| - d_v))$$

$$\leq |y(u)| - |y(v)|$$

$$\leq \phi(u, v),$$

and (C1) is satisfied.
We conclude that (C1) is satisfied after executing the algorithm from $X$. Next, we argue (C2) continues to hold after executing the algorithm from $X$. Consider any exit cluster $Y \in X_+\boxminus$ and any $Y' \in D(Y)$. If $d_{Y'} \geq |y(X)|$, the claim trivially holds because $y(Y')$ did not change. Otherwise, consider the shortest path $P_{Y'}$ from $X$ to $Y'$ computed by Dijkstra’s algorithm. Let $P_{X',Y'}$ be the path $P_{Y'}$ with the vertex $X$ removed; $X'$ is the first vertex after $X$ on $P_{Y'}$. From the feasibility of $(X,Y)$, and (7.9) we have,

$$ |y(X)| - |y(Y)| \leq \phi(X,Y) \leq \sum_{(u,v) \in P_{X',Y'}} \phi(u,v) = |y(X')| - |y(Y')| + \sum_{(u,v) \in P_{X',Y'}} s(u,v) = d_{Y'} - |y(Y')|. $$

Combining this with the dual weight assignment of the procedure gives:

$$ |y'(Y')| = |y(Y')| + |y(X)| - d_{Y'} \leq |y(Y)|, $$

and (C2) holds.

Finally, we argue that the final step of Sync, which assigns the dual weights of some free and boundary clusters to match their parent cluster’s dual weight, does not violate (C1); it is clear that this operation does not violate (C2). Consider any free or boundary cluster $X \in X_\boxminus$, and any child cluster $X' \in D(X)$. If $X$ is an exit cluster, then $X'$ is of type $A$, and there are no outgoing edges from $X'$ in $V_\boxminus$. For any incoming edge $(Z, X')$, the slack only increases because, from (C2), $|y(X')|$ only increases. Similarly, if $X$ is an entry cluster, then $X'$ is of type $B$, and there are no incoming edges to $X'$ in $V_\boxminus$. For any outgoing edge $(X', Z)$, the slack only increases because, from (T1), $|y(X')|$ only decreases. Therefore, the
7.5. Data Structure

final step of Sync does not violate (C1) or (C2).

Lemma 7.15. Consider any internal edge \((X, Y) \in X^\uparrow \times X^\uparrow\) in the associated graph \(AG^\uparrow\) of the parent \(\Box\) of \(\Box\). Suppose \(s(X,Y) = 0\). After execution of Sync on \(\Box\), \(P(X,Y)\) is an admissible path in \(AG\).

Proof. By its definition, \(P(X,Y)\) is a path from some \(X' \in D(X)\) to some \(Y' \in D(Y)\). Note that the net-cost \(\phi(X,Y) = \phi(P(X,Y))\). From our assumption that \((X,Y)\) is admissible and (7.9),

\[
|y(X)| - |y(Y)| = \phi(X,Y) = \phi(P(X,Y)) = |y(X')| - |y(Y')| + \sum_{(u,v) \in P(X,Y)} s(u,v).
\]

Since \(X\) is an entry cluster, from (T1), \(|y(X')| \geq |y(X)|\). Since \(Y\) is an exit cluster, from (C2), it follows that \(|y(Y')| \leq |y(Y)|\). Therefore,

\[
\sum_{(u,v) \in P(X,Y)} s(u,v) = |y(X)| - |y(X')| + |y(Y')| - |y(Y)| \leq 0.
\]

From Lemma 7.14, the edges of \(P(X,Y)\) satisfy (C1), and every edge of \(P(X,Y)\) has slack at least 0. Therefore, every edge of \(P(X,Y)\) must be admissible.

By recursively applying the above lemma, we get the following.

Corollary 6. Let \(\Box\) be a level \(i\) cell. For any internal edge \((X,Y) \in X^\uparrow \times X^\uparrow\) in the associated graph \(AG\) of the parent \(\Box\) of \(\Box\), suppose \(s(X,Y) = 0\). We can recursively apply Sync on all internal edges of \(P(X,Y)\) to obtain its projection \(\Pi_{u,v,\Box}\) with \(u \in X\) and \(v \in Y\). This projection will be an admissible path. For every vertex \(p\) in \(\Pi_{u,v,\Box}\), let \(P(p)\) be all the clusters for cells of level \(i\) or lower that contain the point \(p\). Then, for every \(v' \in P(p)\), \(y(v') = y(p)\).
Efficiency Analysis of Sync  

Next, we bound the time taken for a single call to \textit{Sync} executed on a cell \( \square \) that updates the dual weights of \( V_\square \). The argument is nearly identical to that used for \textit{Construct}. Assume that \( \square \) appears at level \( j \). The \textit{Sync} procedure executes a Dijkstra search once from each of the \( \tilde{O}(\mu_j) \) entry clusters of \( X_\square \). If \( \square \) is full, then each Dijkstra search takes time \( \tilde{O}(|E_\square|) = \tilde{O}(\mu_j^2) \) time. If \( \square \) is sparse, then each Dijkstra search can be executed efficiently in \( \tilde{O}(|X_\square| + |A_\square \cup B_\square|) \) time. Since \( \square \) is sparse, \( |A_\square \cup B_\square| \leq \mu_j^2 \), and each Dijkstra search takes \( \tilde{O}(\mu_j^2) \) time. This gives the following Lemma.

**Lemma 7.16.** Any execution of \textit{Sync} on a cell \( \square \) of layer \( j \) takes \( \tilde{O}(\mu_j^3) \) time. Furthermore, if \( \square \) is sparse, the time taken can be bounded by \( \tilde{O}(\mu_j(\mu_j + |A_\square| + |B_\square|)) \).

### 7.5.8 Data Structure Operations

For any phase \( i \), we present the implementation of the four operations supported by the data structure using the \textit{Sync} and \textit{Construct} procedures. Before we describe the operations, we will state an additional property that the compressed feasible matching maintained by the data structure satisfies. In any phase \( i \geq 1 \), suppose that \( M_\square^\star \), \( y(\cdot) \) is a compressed feasible matching with the additional condition being satisfied:

\[(J) \text{ For each vertex } b \in B_\square^\star, y(b) \geq 0, \text{ and for each } a \in A_\square^\star, y(a) \leq 0. \text{ Furthermore, let } y_{\text{max}} = \max_{v \in B_\square^\star} y(v). \text{ For every free vertex } b \in B_\square^\star, y(b) = y_{\text{max}} \text{ and } \mu_{i-1}^2 \leq y_{\text{max}} \leq \mu_i^2. \text{ For every free cluster } a \in A_\square^\star, y(a) = 0.\]

As we will later show, a compressed feasible matching that satisfies (J) can be converted to an associated \( Q \)-feasible matching that satisfies (I1) and (I2). Therefore, it suffices to maintain (J) during the execution of our algorithm.
7.5. Data Structure

**Build Operation** As input, the Build operation takes a $Q$-feasible matching $M_{\square^*}$ and set of dual weights $y(\cdot)$ on the vertices of $A_{\square^*} \cup B_{\square^*}$. We execute the Construct procedure on every non-root cell $\square$ of $T_{\square^*}$ in the order of their level in $Q$, processing lower layers first. This ensures that, when Construct is called on $\square$, the associated graph $AG_{\square}$ has already been computed, along with the dual weights for vertices of $V_{\square}$. After Construct is called on all pieces of $\square^*$, the result is an associated graph $AG_{\square}$ for every full cell $\square \in T_{\square^*}$ and dual weights $y(\cdot)$ for all vertices of $\bigcup_{\square \in T_{\square^*}} V_{\square}$. The following lemma argues that this set of dual weights is compressed feasible with respect to $M_{\square^*}$.

**Lemma 7.17.** After executing Construct on all non-root cells of $AG_{\square^*}$, the matching $M_{\square^*}$ and the dual assignment $y(\cdot)$ are compressed feasible.

**Proof.** Consider the circumstances after calling Construct on all children of some full $\square \in T_{\square}$. Inductively assume that, for each child $\square'$ of $\square$, the edges of $AG_{\square'}$ were feasible prior to executing Construct on $\square'$. Then, from Lemma 7.12, the internal edges of $AG_{\square}$ are feasible. It remains to argue that the bridge edges of $AG_{\square}$ are feasible. Consider any such bridge edge $(X, Y)$ in $AG_{\square}$. From a simple inductive argument on the dual assignment of the Construct procedure, it is easy to see that for some point $u \in X$, $y(u) = y(X)$. Similarly, for some point $v \in Y$, $y(v) = y(Y)$. Furthermore, $(u, v)$ is an edge in $G_M$, and $\phi(X, Y) = \phi(u, v)$. Consider the case where $(X, Y)$ is local. Then, from the feasibility of $(u, v)$, we have,

$$|y(X)| - |y(Y)| = |y(u)| - |y(v)| = \phi(u, v) = \phi(X, Y),$$

and $(X, Y)$ is feasible. Similarly, consider if $(X, Y)$ is non-local. Then, from the feasibility
Chapter 7. An Approximation Algorithm for RMS Matching in a Plane

of \((u, v)\), we have,

\[
|y(X)| - |y(Y)| = |y(u)| - |y(v)| \leq \phi(u, v) = \phi(X, Y),
\]

and \((X, Y)\) is feasible. This implies that \(M_{\square^*}, y(\cdot)\) are compressed feasible after executing \textsc{Construct} on all non-root cells of the active tree \(T_{\square^*}\).

\textbf{Corollary 7.} Given a \(Q\)-feasible matching that satisfies (I1) and (I2), upon applying the \textsc{Build} procedure, the compressed feasible matching will satisfy (J).

\textit{Proof.} Consider any cluster \(X \in V_{\square^*}\). Then, there must be some \(u \in X\) for which \(y(u) = y(X)\) after \textsc{Build}. Since (I1) and (I2) were satisfied for \(u\), it is easy to see that (J) holds for \(X\). \qed

\textbf{Execution Time for \textsc{Build}} We show that the time taken by \textsc{Build} during phase \(i\) is \(\tilde{O}(n\mu_i^{2/3})\). During, the \textsc{Build} procedure, \textsc{Construct} is called on all non-root cells of \(T_{\square^*}\) for each full active cell \(\square^*\). We assign each non-root cell \(\square \in T_{\square^*}\) to one of four categories:

(a) \(\square\) is full.

(b) \(\square\) is sparse and the parent of \(\square\) in \(T_{\square^*}\) is a full cell that is not the root \(\square^*\).

(c) \(\square\) is sparse, its parent in \(T_{\square^*}\) is the root \(\square^*\), and \(|A_{\square} \cup B_{\square}| \leq \mu_i^{2/3}\).

(d) \(\square\) is sparse, its parent in \(T_{\square^*}\) is the root \(\square^*\), and \(|A_{\square} \cup B_{\square}| > \mu_i^{2/3}\).

We separately bound the total time taken for a single \textsc{Construct} call on every cell in each of the four categories, over all active cells for phase \(i\), showing that the time taken is \(\tilde{O}(n\mu_i^{2/3})\).
7.5. Data Structure

First, we bound the time taken by cells of category (a). We bound the the time for a Construct call on all full cells in some grid $G_j$. Since these full cells together contain at most $n$ points, the total number of full cells in $G_j$ is bounded by $\tilde{O}(n/\mu_j^2)$. A Construct call on a full cell $\Box \in G_j$ takes $\tilde{O}(\mu_j^2)$ time. Therefore, the total time taken for all full cells of $G_j$ is $\tilde{O}(n\mu_j)$. During phase $i$, Construct is only called on cells of $G_j$ where $j \leq \lfloor 2i/3 \rfloor$. The time taken by $G_j[2i/3]$ dominates, taking $\tilde{O}(n\mu_j^{2/3})$ time. This completes the bound on cells in category (a).

Next, we bound the time taken for category (b). If a sparse cell $\Box$ of level $j$ in an active tree has a non-root parent $\Box'$ in level $j + 1$, then its parent $\Box'$ must fall into category (a). The time taken for a call to Construct on $\Box$ is $\tilde{O}(\mu_j^2) = \tilde{O}(\mu_{j+1}^2)$, which can be taxed on the time taken to execute Construct on the parent $\Box'$. Specifically, since each non-root cell $\Box'$ in the active tree has at most 4 children in the active tree, the time taken for a Construct call on all sparse children of $\Box'$ is $\tilde{O}(\mu_j^{3})$, which is also the bound on the time taken for Construct on $\Box'$ itself. Therefore, the total time taken by category (b) is bounded by the time taken by (a), and is $\tilde{O}(n\mu_i^{2/3})$.

Now we bound the time for category (c). All cells of category (c) are pieces of some active tree, so we begin by bounding the total number of pieces over all active cells. Since these cells together contain at most $n$ points, the number of full active cells during phase $i$ is $\tilde{O}(n/\mu_i^2)$. Each active cell $\Box^*$ has its pieces in grid $G_{[2i/3]}$. Since $\Box^*$ has diameter $\tilde{O}(\mu_i^2)$ and each piece of $\Box^*$ has diameter $\Omega(\mu_{[2i/3]}^2/poly\{\log n, 1/\varepsilon\}) = \Omega(\mu_i^{4/3}/poly\{\log n, 1/\varepsilon\})$, $\Box^*$ has $\tilde{O}(\mu_i^{2/3}) = \tilde{O}(\mu_i^{4/3})$ pieces. Summing over all phase $i$ active full cells gives a total of $\tilde{O}(n/\mu_i^{2/3})$ pieces. The time taken by a single Construct call on one of these pieces $\Box$ is $\tilde{O}(\mu_{[2i/3]}|A_{\Box} \cup B_{\Box}| + \mu_{[2i/3]}^2) = \tilde{O}(\mu_i^{2/3}|A_{\Box} \cup B_{\Box}| + \mu_i^{4/3})$. However, since $|A_{\Box} \cup B_{\Box}| \leq \mu_i^{2/3}$, we can rewrite the time taken for a single Construct call as $\tilde{O}(\mu_i^{4/3})$. Summing over all $\tilde{O}(n/\mu_i^{2/3})$ pieces of category (c) gives a total time of $\tilde{O}(n\mu_i^{2/3})$ as desired.
Finally, we bound the time taken for category (d). The time taken for a single \textsc{Construct} call on a cell \(\Box\) of category (d) is \(\tilde{O}(|A_\Box \cup B_\Box|\mu_i^{2/3} + \mu_i^{4/3})\). However, since \(|A_\Box \cup B_\Box| > \mu_i^{2/3}\), the first term dominates, and we can rewrite the time taken by \textsc{Construct} on \(\Box\) as \(\tilde{O}(|A_\Box \cup B_\Box|\mu_i^{2/3})\). Summing over all such cells of category (d) gives a total time of \(\tilde{O}(n\mu_i^{2/3})\).

\textsc{GenerateDuals} Operation  The \textsc{GenerateDuals} procedure simply consists of recursively calling the \textsc{Sync} procedure on all non-root cells of \(\mathcal{T}_\Box^*\), processing cells closest to the root of \(\mathcal{T}_\Box^*\) first. This process generates a set of dual weights \(y(\cdot)\) for the vertices of \(A_{\Box^*} \cup B_{\Box^*}\). Next, we show that after executing this \textsc{GenerateDuals} procedure, \(M_{\Box^*}, y(\cdot)\) are \(Q\)-feasible, meaning (I1) holds.

**Lemma 7.18.** After executing \textsc{Sync} on all non-root cells of \(\mathcal{T}_\Box^*\), starting with the cells closest of the root of \(\mathcal{T}_\Box^*\), let \(y(\cdot)\) be the dual weights of vertices in \(\bigcup_{\Box \in \mathcal{T}_\Box^*} V_\Box\). Then \(M_{\Box^*}, y(\cdot)\) are \(Q\)-feasible.

**Proof.** We consider any edge \((u, v)\) in \(G_{\Box^*}\). The vertices \(u\) and \(v\) must each appear in some sparse cell in \(\mathcal{T}_\Box^*\). If \(u\) and \(v\) are in the same sparse leaf cell \(\Box \in \mathcal{T}_\Box^*\), then \((u, v) \in G_\Box\), and \((u, v)\) is \(Q\)-feasible because \(M_\Box, y(\cdot)\) are compressed feasible. Otherwise, \(u\) and \(v\) appear at different leaves of \(\mathcal{T}_\Box^*\), and there is some pair of clusters \((X, Y)\) and some cell \(\Box \in \mathcal{T}_\Box^*\) such that \((X, Y)\) is a bridge edge in \(E_\Box\), \(u \in X\), and \(v \in Y\). If \((X, Y)\) is local, then \((u, v)\) is also local, and we have that \(|y(X)| - |y(Y)| = \phi(X, Y)\) from (C1). From (T2) we get that \(y(u) = y(X)\) and \(y(v) = y(Y)\). Since \(\phi(X, Y) = \phi(u, v)\), we get that \((u, v)\) is \(Q\)-feasible.

Otherwise, we consider the case where \((u, v)\) is non-local. We claim that \(|y(u)| \leq |y(X)|\) (resp. \(|y(v)| \geq |y(Y)|\)). If \(X\) (resp. \(Y\)) is a boundary or free cluster, then, from (T2), \(y(u) = y(X)\) (resp. \(y(v) = y(Y)\)) and the claim holds. The remaining case is when \(X\) (resp. \(Y\)) is an internal cluster. Since the edge \((X, Y)\) is directed from \(X\) to \(Y\), the cluster \(X\) (resp. \(Y\)) must be an internal exit cluster of \(B\) (resp. internal entry cluster of \(A\)). From
conditions (C2) (resp. (T1)), we immediately get that \(|y(u)| \leq |y(X)|\) (resp. \(|y(v)| \geq |y(Y)|\)). Therefore, the claim holds. From condition (C1), \(|y(X)| - |y(Y)| \leq \phi(X,Y)|. Combining this with the facts that \(|y(u)| \leq |y(X)|\) and \(|y(v)| \geq |y(Y)|\) immediately implies that \((u, v)\) is \(Q\)-feasible.

**Lemma 7.19.** Consider any compressed feasible matching that satisfies (J), then \textsc{GenerateDuals} generates an associated \(Q\)-feasible matching that satisfies invariant (I2).

**Proof.** Consider any free vertex \(v \in A_F \cup B_F\) of \(G_{\square^*}\). Then there must be some free vertex cluster \(X \in X_{\square}\) for some piece \(\square\) of \(\square^*\) in \(T_{\square^*}\), such that \(v \in X\). From (T2), \(y(v) = y(X)\) after \textsc{GenerateDuals}. Since (J) holds for \(X\), (I2) holds for \(v\). Next, consider the case where \(v \in A \cup B\) is not free. The \textsc{GenerateDuals} procedure only modifies the dual weight of \(v\) via the \textsc{Sync} procedure, which ensures that \(y(v)\) remains non-positive (resp. non-negative) if \(v \in A\) (resp. \(v \in B\)).

**Execution Time for \textsc{GenerateDuals}** Since \textsc{Construct} and \textsc{Sync} have the same time bounds from Lemmas 7.13 and 7.16, and both procedures are called on all non-root cells of the active tree \(\mathcal{A}G_{\square^*}\), it is easy to see that the time taken by \textsc{GenerateDuals} can be bounded in a fashion identical to the argument used for the efficiency of \textsc{Build}. Therefore, \textsc{GenerateDuals} takes \(\tilde{O}(n\mu_i^{2/3})\) time for phase \(i\).

**Search Operation** This procedure takes a compressed feasible matching \(M_{\square^*}, y(\cdot)\) that also satisfies (J) as input. It then conducts a search identical to Hungarian search on the associated graph of \(\square^*\). The search procedure adjusts the dual weights of the vertices of \(V_{\square}\) so that we have a path consisting of admissible edges. Once an admissible path is found in \(\mathcal{A}G_{\square^*}\), the procedure projects this path to find an augmenting path of admissible edges in \(G_{\square}\) by recursively applying the \textsc{Sync} procedure. We describe the details of the procedure
in two parts. First, we describe the dual adjustments conducted by the \textsc{Search}, and then we describe how the procedure projects the path. We show that (J) continues to hold after the execution of \textsc{Search}.

\textbf{Dual Adjustments} Recall that $A_{\square^\ast}$ (resp. $B_{\square^\ast}$) denotes the set of vertices of type $A$ (resp. type $B$) in $V_{\square^\ast}$. Let $A_F$ (resp. $B_F$) be the set of free vertex clusters of $A_{\square^\ast}$ (resp. $B_{\square^\ast}$). We add a vertex $s$ to the graph $A_G'$ and add an edge from $s$ to every free cluster of $B_F$. The weight associated with this edge is 0. We set $\ell_{\text{max}} = \mu_i^2 - \max_{X \in B_{\square^\ast}} y(X)$. We then execute a Dijkstra’s search to compute the shortest path distance from $s$ to every vertex in $V_{\square^\ast}$. For any $v \in V_{\square^\ast}$, let $\ell_v$ be the shortest path distance from $s$. Let $\ell = \min_{X \in A_F} \ell_X$. If $\ell > \ell_{\text{max}}$, we set $\ell = \ell_{\text{max}}$ and continue. For every vertex $v \in V_{\square^\ast}$ with $\ell_v \leq \ell$, we update the dual weight as follows. If $v \in B_{\square^\ast}$, we increase the dual weight $y(v) \leftarrow y(v) + \ell - \ell_v$. Otherwise, if $v \in A_{\square^\ast}$, we reduce the dual weight $y(v) \leftarrow y(v) - \ell + \ell_v$. This completes the description of the dual weight changes. These dual adjustments will make some of the edges on the shortest path tree have a zero slack. If $\ell = \ell_{\text{max}}$, the dual weight of every free cluster of type $B$ would be updated to $\mu_i^2$ and we return without finding an augmenting path. Otherwise, the dual adjustments will maintain compressed feasibility and create an admissible path $P$ from a free cluster $Z \in B_F$ to a free cluster $Z'$ of $A_F$ inside the associated graph $A_G'$. Using a relatively straight-forward and standard argument very similar to that used in Lemma 7.14, one can show that these dual adjustments do not violate the compressed feasibility conditions.

\textbf{Projecting an Augmenting Path} The dual adjustment ensures that there is some admissible augmenting path $P$ in $A_G'$. We create an augmenting admissible augmenting path in $G_{\square^\ast}$ from some free vertex $b \in Z$ to $a \in Z'$ as follows: For any internal edge $(U, V)$ in $P$, we can recursively use \textsc{Sync} (Corollary 6) to retrieve an admissible path $\Pi_{u',v',\square^\ast}$ where
u' ∈ U and v' ∈ V. We make u' (resp. v') the representative of U (resp. V) and denote it by r(U) (resp. r(V)). For every vertex Y on the path P that does not have a representative, we choose an arbitrary vertex p ∈ Y as its representative, p = r(Y). Note that P cannot have any vertex with two internal edges incident on it. Next, for any bridge edge (x, y) in P, we show how to connect their representatives. Suppose the bridge edge (x, y) is non-local edge. Then, we connect r(x) and r(y) directly by a non-local edge in 𝒢. Otherwise, suppose (x, y) is a local bridge edge. In this case, if r(x) is matched to r(y), we simply add the matching edge between them. Otherwise, if r(x) is matched to x' and r(y) is matched to y', the edges (r(x), x'), (x', y') and (y', r(y)) are all local and admissible. We add them the three edges in this order to connect r(x) to r(y). The resulting path obtained is a compact admissible path from a free vertex in BF to a free vertex in AF as desired.

Note that the input compressed feasible matching satisfied (J) and the dual weight of every free cluster v in BF is y_max. The dual adjustments conducted by the SEARCH procedure will not decrease the dual weights of any vertex v ∈ BF and will not increase the dual weight of any vertex v ∈ AF. Furthermore, each dual adjustment conducted by the SEARCH procedure increases the dual weight of all free clusters of BF by ℓ which is the largest increase among all clusters. Therefore, the new dual weight of free clusters is y_max + ℓ which is the largest among all vertices of BF. Finally, by definition, every free vertex cluster v of AF has ℓ_v ≥ ℓ and, therefore, y(v) remains 0. In conclusion, after the execution of SEARCH procedure (J) continues to hold.

**Efficiency of Search** Next, we bound the time taken by the SEARCH procedure. First, we bound the time taken for the Dijkstra search over AG during some phase i. The root cell □ has a diameter of ∼(μ²i), and each of its pieces have a diameter of ∼(μ²i/3i). Therefore, there are ∼((μ²/μi⁴/3)²) = ∼(μi⁴/3) pieces of □. Each piece contains ∼(μi²/3)
vertices in \( V \) and \( \tilde{O}(\mu_i^{4/3}) \) internal edges in \( E \). The number of bridge edges in \( E \) could be much higher, but we observe that, by using the WSPD, the bridge edges incident on every vertex of \( V \) can be divided into only \( \tilde{O}(1) \) groups where the edges of each group have the same net-cost and direction. A similar technique is used for the Hungarian search described in Section 7.4. Therefore, the Dijkstra search over \( AG' \) can be executed in time near-linear in the number of internal edges and vertices of \( AG' \), i.e., \( \tilde{O}(\mu_i^{8/3}) \) time.

After executing the Dijkstra search over \( AG' \), the SEARCH procedure executes the SYNC procedure to produce an admissible augmenting path \( P \) in \( G_M \). During this process, SYNC only needs to be executed once per affected cell \( \square \in \mathcal{A}(P) \). From Lemma 7.16, each execution of SYNC on a cell of level \( j \) takes \( \tilde{O}(\mu_j^{3}) \) time. Recall that SYNC is not called on any cell with level higher than \( \lceil 2i/3 \rceil \), i.e., the level of the pieces of \( \square \). Therefore, the total time taken by the executions of the SYNC procedure can be expressed as:

\[
\tilde{O}(\sum_{j=0}^{\lceil 2i/3 \rceil} |\mathcal{A}_j(P)| \mu_j^{3}).
\]

Combining this with the time taken by the Dijkstra search gives the following bound on the time taken by the SEARCH procedure.

\[
\tilde{O}(\mu_i^{8/3}) + \sum_{j=0}^{\lceil 2i/3 \rceil} |\mathcal{A}_j(P)| \mu_j^{3}).
\]

### 7.5.9 Augment Operation

The AUGMENT procedure accepts an admissible augmenting path \( P \) in \( G_M \). It then augments \( M \) along \( P \), and updates the data structure accordingly. To augment \( M \) along \( P \), we set \( M \leftarrow M \oplus P \) and perform very similar dual weight changes to those described in Section 7.4. For any edge \( (a, b) \) that was non-local prior to augmentation and became local after
augmentation, let \( \Box \) be the least common ancestor of \( a \) and \( b \) in \( Q \). If there is a local bridge edge \((X, Y) \in E_{\Box}\) prior to augmentation such that \( a \) enters \( X \) and \( b \) enters \( Y \) through augmentation, we simply set \( y(a) \leftarrow y(X) \) and \( y(b) \leftarrow y(Y) \). Otherwise, if no such local edge existed, we set \( y(a) \leftarrow y(a) - \mu_{ab}^2 \). Using similar arguments to those given in Section 7.4, it can be shown that this dual weight assignment only decreases the dual weights of \( y(a) \) and \( y(b) \).

After augmenting along \( P \), the data structure must perform updates to account for the changes to the matching. Recall that the set \( A(P) \) of affected cells contains all non-root cells of \( T_{\Box^*} \) that contain at least one vertex of \( P \). To update the data structure, the procedure executes the CONSTRUCT procedure on all cells of \( A(P) \), processing cells at lower layers of \( Q \) first.

**Efficiency of Augment** To bound the efficiency of the AUGMENT procedure, we consider the most expensive portion, which is the time taken for the calls to the CONSTRUCT procedure on all affected pieces. Consider an execution of AUGMENT that produced an augmenting path \( P \). Recall that, from Lemma 7.13, the time taken for a single call to CONSTRUCT on a cell of level \( j \) is \( \tilde{O}(\mu_j^3) \), which matches the time taken for the calls to the SYNC procedure during the execution of SEARCH that generated \( P \). Using an identical argument, we can conclude that the total time taken by AUGMENT is:

\[
\tilde{O}\left( \sum_{j=0}^{\lfloor 2i/3 \rfloor} |A_j(P)|\mu_j^3 \right).
\]

Next, we show that the AUGMENT operation will not violate compressed feasibility. For any point \( p \) on the augmenting path, all clusters that contain \( p \) have the same dual weight as \( p \); this follows from Corollary 6 and the fact that \( P \) was found by recursively applying SYNC
on an admissible path in $\mathcal{AG}_\square$. As was the case in Section 7.4, the Augment procedure only reduces the dual weights of vertices in $A \cup B$. The Build procedure, when applied at the ancestors, may reduce the dual weights of some clusters. Recollect that the dual updates are done so that the local edges that they participate in satisfy (C2). Reducing the the dual weight of any cluster of type $B$ or reducing (i.e., increasing the magnitude of) the dual weight of any cluster of type $A$ only increases the slack on non-local edges. As a result, the compressed feasibility conditions holds continue to hold.

7.5.10 Simple Augmenting Paths

When the algorithm executes the Search procedure to generate an admissible augmenting path $P$, it is important that this augmenting path is simple, having no self-intersections. In this section, we argue that all augmenting paths generated by the algorithm are simple. We begin by specifying a useful property of any cycle in $\mathcal{G}_\square$.

**Lemma 7.20.** Let $C$ be any cycle in $\mathcal{G}_\square$. Then $s(C) = \phi(C)$.

**Proof.** Observe that,

$$s(C) = \sum_{(u,v) \in C} \phi(u,v) - |y(u)| + |y(v)| = \phi(C) + \sum_{(u,v) \in C} -|y(u)| + |y(v)|.$$ 

Since each vertex of $C$ occurs as the head and tail of exactly one edge of $C$, the net contribution of each dual weight to $s(C)$ is 0. Therefore, for any alternating cycle $C$ in $\mathcal{AG}_\square$, we have $s(C) = \phi(C)$. 

Note that any path $P$ returned by the Search procedure must contain at least one non-local edge. Therefore, to argue that the algorithm never produces an admissible cycle, it is
sufficient to argue that the graph $G_{\square^*}$ does not contain any cycles with both 0 net-cost and at least one non-local edge. It is worth noting that, since local edges are admissible, any cycle consisting solely of local edges of the same class is admissible. However, the SEARCH procedure will never return such a cycle as part of a path.

At the beginning of the algorithm, all edges are non-local, and have a positive net-cost. Therefore, we can assume the claim holds initially. The only operation performed by the algorithm that changes net-costs in $G_{\square^*}$ is augmentation. So, it is sufficient to argue that, if there were no 0 net-cost cycles with a non-local edge prior to some augmentation, there are also no such cycles after augmentation. Lemma 7.20 implies that any cycle has zero net-cost iff it is admissible with respect to every possible $Q$-feasible dual assignment. Therefore, it suffices to argue that every cycle with at least one non-local edge after augmentation is inadmissible with respect to any single $Q$-feasible dual assignment.

The algorithm does not explicitly maintain a $Q$-feasible set of dual weights, but it does implicitly maintain an associated $Q$-feasible matching. Namely, the GENERATEDUALS procedure accepts a compressed feasible matching as input and returns a $Q$-feasible set of dual weights $y(\cdot)$ for the points of $A_{\square^*} \cup B_{\square^*}$. Instead of generating this set of dual weights in its entirety, the algorithm only generates the dual weights that may change during augmentation, i.e., those along the augmenting path $P$. Furthermore, the augmenting path $P$ produced by the algorithm is admissible w.r.t. $y(\cdot)$ from Corollary 6. We can describe a set of $Q$-feasible dual weights $y'(\cdot)$ after the augmentation; for any vertex $v$ not on $P$, $y'(v) = y(v)$, and for any vertex on $P$, the new dual weight $y'(v)$ is assigned explicitly by the AUGMENT procedure. Since the AUGMENT procedure only reduces dual weights, $y'(v)$ is a $Q$-feasible matching with respect to the matching $M'$ after augmentation. Therefore, we simply need to argue that there are no admissible cycles w.r.t. $y'(\cdot)$ in $G_{M'}$ that have at least one non-local edge. We argue this in the following lemma.
Lemma 7.21. Let $M, y(\cdot)$ be any $Q$-feasible matching such that $G_M$ does not contain any admissible cycles with at least one non-local edge, and let $P$ be an admissible compact augmenting path with respect to $y(\cdot)$. Consider the matching $M' = M \oplus P$ and the set of dual weights $y'(\cdot)$ assigned during augmentation. Then $G_{M'}$ does not contain any admissible cycles w.r.t. $y'(\cdot)$ with at least one non-local edge.

Proof. Assume for the sake of contradiction that $G_{M'}$ contains an admissible alternating cycle $C'$ with at least one non-local edge. Observe that any vertex that experiences a dual weight change during augmentation only has its dual weight strictly reduce. This causes all non-matching edges incident on it to accumulate a strictly positive slack with respect to $M', y'(\cdot)$. Since we assumed that $C'$ is admissible, $C'$ cannot contain any such non-matching edge, which implies that $C'$ does not contain a vertex that experienced a dual weight change.

Now, consider any edge $(u, v)$ shared between $C'$ and $P$ that is in $M'$ and was a non-local non-matching edge w.r.t. $M'$. Since $(u, v)$ was an admissible non-local edge, $y(u) + y(v) = d_Q(u, v) + \mu_{uv}$. After augmentation, $(u, v)$ is a feasible matching edge with $y'(u) + y'(v) = d_Q(u, v)$. Therefore, the dual weight of one of the endpoints of $(u, v)$ decreased. Since $C'$ cannot use any vertex that experienced a dual weight decrease, $C'$ cannot use any edge $(u, v)$ that is in $M'$ but was non-local w.r.t. $M$.

We conclude that any edge $(u, v)$ of $M'$ on $C'$ must have been a local non-matching edge prior to augmentation. Since $(u, v)$ was local in $M$, $u$ was matched to a vertex $v'$, $v$ was matched to a vertex $u'$, and there must have been another non-matching local edge directed from $u'$ to $v'$ in $G_M$. Therefore, $G_M$ contains an admissible path $P_{u,v} = \langle v, u', v', u \rangle$ from $v$ to $u$ in $G_M$.

Using this fact, we can craft an admissible cycle $C$ in $G_M$ as follows: For any edge $(u, v)$ on $C'$ that is not on $P$, we add $(u, v)$ to $C$. Since neither $u$ nor $v$ experienced a dual weight
change, any such edge is admissible w.r.t. $G_M$. For any edge $(u, v)$ on $C'$ that is also on $P$, we add the edges of $P_{u,v}$ to $C$. Note that all edges of $P_{u,v}$ were admissible local edges in $G_M$. Thus, $C$ forms an admissible cycle in $G_M$. Furthermore, any non-local edge of $C'$ is also a non-local edge in $C$. Since we assumed that $C'$ contains at least one non-local edge, this contradicts the assumption that $G_M$ did not contain any admissible cycles with at least one non-local edge.

\[\square\]

\section*{7.6 Transforming Input}

In this section, we show how to transform point sets $A', B' \subseteq \mathbb{R}^2$ of $n$ points each, into point sets $A$ and $B$ with $n$ points each so that properties (A1)–(A3) hold, i.e.,

(A1) Every point in $A \cup B$ has integer coordinates bounded by $\Delta = n^{O(1)}$,

(A2) The optimal matching of $A$ and $B$ has a cost of at most $O(n/\varepsilon^2)$, and,

(A3) Any optimal matching of $A$ and $B$ corresponds to a $(1 + \varepsilon)$-approximate matching of $A'$ and $B'$.

We start by computing an $n^{O(1)}$-approximation of the optimal matching cost. First, we compute a $2n^2$-approximate bottleneck matching $M_B$ of $A', B'$ in $O(n \log n)$ time using the algorithm of [1]; see Lemma 2.2 in their paper. Let $\beta$ be the optimal bottleneck distance; then each edge of $M_B$ has a length of at most $2n^2 \beta$. Therefore, the cost of $M_B$ under squared-Euclidean distance is at most $2n^5 \beta^2$. On the other hand, at least one edge of the optimal squared-Euclidean matching $M_{OPT}$ must have a length that is at least $\beta^2$; otherwise, there is a smaller bottleneck matching distance than $\beta$. Therefore, the optimal squared Euclidean cost is at least $\beta^2$. We conclude that $w(M_B) \leq 2n^5 w(M_{OPT})$. 

Next, let $\Gamma = c(M_B)$. Then for each integer $i$ such that $2^i \in [\Gamma/(2n^5), \Gamma]$, let $\gamma = 2^i$. For at least one of these $O(\log n)$ values of $\gamma$, we will have

$$\gamma \leq c(M_{OPT}) \leq 2\gamma. \quad (7.10)$$

For a sufficiently large constant $c_1$, we execute $c_1 n^{5/4} \text{poly} \{\log n, 1/\varepsilon\}$ steps of the algorithm for each value of $\gamma$. Therefore, we can assume that our algorithm has a value of $\gamma$ that satisfies (7.10). We rescale the point set by dividing all coordinates by $\sqrt{\frac{\gamma \varepsilon^2}{64n}}$. Let $\hat{A}$ and $\hat{B}$ be the resulting scaled points from $A'$ and $B'$ respectively. Since the scaling was uniform, the optimal matching $\hat{M}_{OPT}$ with respect to the scaled points $\hat{A}, \hat{B}$ is also optimal with respect to the original point sets $A', B'$. Similarly, a $(1 + \varepsilon)$-approximate matching with respect to $\hat{A}, \hat{B}$ is also a $(1 + \varepsilon)$-approximate matching with respect to the original points $A', B'$. As a result of the scaling, it is easy to see from (7.10) that the resulting optimal matching cost can be bounded by,

$$64n/\varepsilon^2 \leq c(\hat{M}_{OPT}) \leq 128n/\varepsilon^2. \quad (7.11)$$

Next, we explain how to ensure that the diameter of the point set is polynomial in $n$. We construct a randomly shifted grid $\hat{G}$, where each cell in the grid has side-length $512 \cdot n^3/\varepsilon^2$. Since each edge of the optimal matching has cost at most $128n\varepsilon^2$, each edge of $\hat{M}_{OPT}$ has a probability of at most $1/n^2$ of crossing between two different cells of $\hat{G}$. The probability that at least one of the $n$ optimal matching edges crosses between different cells is at most $1/n$. Therefore, we can split the point set using cells of $\hat{G}$, treating the points within each cell as a separate problem, and combine the resulting matchings together. With probability at least $1 - 1/n$, this splitting of points will not destroy any edges of the optimal matching. Therefore, we can assume that $\Delta = n^{O(1)}$.

Finally, we round the point sets $\hat{A}, \hat{B}$ to the nearest integer coordinates; let $A, B$ be the
resulting point sets, and let $M_{\text{OPT}}$ be the optimal matching with respect to these rounded coordinates. The following lemma shows that $M_{\text{OPT}}$ is a $(1+\varepsilon)$-approximation of the optimal matching $\hat{M}_{\text{OPT}}$ prior to rounding. Combining this fact with (7.11) in turn gives properties (A1)-(A3).

**Lemma 7.22.**

$$c(M_{\text{OPT}}) \leq (1 + \varepsilon)c(\hat{M}_{\text{OPT}}).$$

**Proof.** To see this, first consider that the Euclidean length of any edge is distorted by at most $\sqrt{2}$ from rounding. For any point $\hat{a}$ prior to rounding, let $a$ be the corresponding point after rounding. We have,

$$c(M_{\text{OPT}}) = \sum_{(a,b)\in M_{\text{OPT}}} \|a - b\|^2 \leq \sum_{(\hat{a},\hat{b})\in \hat{M}_{\text{OPT}}} \|\hat{a} - \hat{b}\|^2 \leq \sum_{(\hat{a},\hat{b})\in \hat{M}_{\text{OPT}}} (\|\hat{a} - \hat{b}\|^2 + \sqrt{2})^2 = \sum_{(\hat{a},\hat{b})\in \hat{M}_{\text{OPT}}} (\|\hat{a} - \hat{b}\|^2 + \sqrt{2}\|\hat{a} - \hat{b}\| + 2) = c(\hat{M}_{\text{OPT}}) + \sum_{(\hat{a},\hat{b})\in \hat{M}_{\text{OPT}}} (\sqrt{2}\|\hat{a} - \hat{b}\| + 2).$$

We must show that $\sum_{(\hat{a},\hat{b})\in \hat{M}_{\text{OPT}}} (\sqrt{2}\|\hat{a} - \hat{b}\| + 2) \leq \varepsilon c(\hat{M}_{\text{OPT}})$. Since $c(\hat{M}_{\text{OPT}}) \geq 64 n/\varepsilon^2$, it is sufficient to show that the quantity $\sum_{(\hat{a},\hat{b})\in \hat{M}_{\text{OPT}}} (\sqrt{2}\|\hat{a} - \hat{b}\| + 2)$ is at most $64 n/\varepsilon$, or that, $\sum_{(\hat{a},\hat{b})\in \hat{M}_{\text{OPT}}} \|\hat{a} - \hat{b}\| \leq 31 n/\varepsilon$. To bound this quantity, we divide the edges of $\hat{M}_{\text{OPT}}$ into two groups. First, consider that all the edges $(\hat{a},\hat{b}) \in \hat{M}_{\text{OPT}}$ with $\|\hat{a} - \hat{b}\| \leq 16/\varepsilon$ contribute a total value of at most $16 n/\varepsilon$. Next, consider the edges $(\hat{a},\hat{b}) \in \hat{M}_{\text{OPT}}$ with $\|\hat{a} - \hat{b}\| > 16 n/\varepsilon$. For each such edge, we have, $\|\hat{a} - \hat{b}\| \leq \varepsilon \|\hat{a} - \hat{b}\|^2/16$. Since $c(\hat{M}_{\text{OPT}}) \leq 128 n/\varepsilon^2$, the
total contribution from these edges is at most $8n^2/\varepsilon$. Thus, $\sum_{(\hat{a}, \hat{b}) \in \hat{M}_{\text{OPT}}} \|a - b\| \leq 31n/\varepsilon$, completing the proof. \qed

7.7 Omitted Proofs

7.7.1 Proof of Lemma 7.2

For any cell $\square$ of $Q$ with level $i$, the total number of subcells is $\tilde{O}(\mu_i)$. 

Proof. We bound the number of subcells by giving an upper bound on the number of leaves of $Q_{\square}$. Let $p$ be the center of $\square$ and let $j = \lfloor i/2 \rfloor - 2\log \frac{\log \Delta}{\varepsilon} - c_1$ be such that $\mu_i = 2^j$ is the minimum subcell size for $\square$. Consider a set of concentric axis-parallel squares $S'_1, \ldots, S'_t$, where each $S'_r$ is centered at $p$ with a side-length $2^i - (\frac{144}{\varepsilon} + 1)2^{j+r}$. Note that $t < i - j$. Consider all cells of $G_{j+r}$ that are completely contained inside $S'_r$. Let $S_r$ be the bounding square of these cells. Note that the distance $\ell_{\text{min}}(S_r, \square)$ is at least $\frac{144}{\varepsilon}2^{j+r}$ and therefore, all subcells of $G[\square]$ inside the square $S_r$ (by condition (b) for subcell construction) are cells of $G_k$ for some $k \geq j + r$. The total number of subcells in the region $S_r \setminus S_{r+1}$ can be bounded by the maximum number of cells of $G_{j+r}$ that can fit inside this region. The side length of $S_r$ is at least $2^i - (\frac{144}{\varepsilon} + 1)2^{j+r}$ and the side length of $S_{r+1}$ is at most $2^i - (\frac{144}{\varepsilon})2^{j+r+1}$. Therefore, the total number of cells of $G_{j+r}$ that can fit inside this region is $\tilde{O}(2^{i-j-r})$. The values for $r$ can range from 1 to $i - j$. Therefore, the total number of subcells is at most $\tilde{O}(i2^{i-j}) = \tilde{O}(\mu_i)$. \qed
7.7.2 Proof of Lemma 7.3

In the following lemma, we use the $Q$-feasibility conditions to upper bound the cost of any $Q$-optimal matching by $\sum_{(a,b)\in M_{OPT}} d_Q(a,b) + \mu_{ab}^2$. This will assist in proving Lemma 7.3.

**Lemma 7.23.** For any $Q$-optimal matching $M$ and set of dual weights $y(\cdot)$ on the vertices of $A\cup B$, then $w(M) \leq \sum_{(a,b)\in M_{OPT}} d_Q(a,b) + \mu_{ab}^2$.

**Proof.** For any edge $(a,b)$ in the matching $M$, from equation (7.2), $\|a - b\|^2 \leq d_Q(a,b)$ and so,

$$w(M) = \sum_{(a,b)\in M} \|a - b\|^2 \leq \sum_{(a,b)\in M} d_Q(a,b). \quad (7.12)$$

If $(a,b) \in M$, then $(a,b)$ is local and from (7.5) we have $y(a) + y(b) = d_Q(a,b)$. Since $M$ is a perfect matching,

$$\sum_{(a,b)\in M} d_Q(a,b) = \sum_{(a,b)\in M} (y(a) + y(b)) = \sum_{v\in A\cup B} y(v).$$

Finally, consider the edges of the optimal matching $M_{OPT}$. From the fact that $M_{OPT}$ is a perfect matching, and from the $Q$-feasibility conditions,

$$\sum_{v\in A\cup B} y(v) = \sum_{(a,b)\in M_{OPT}} y(a) + y(b) \leq \sum_{(a,b)\in M_{OPT}} d_Q(a,b) + \mu_{ab}^2. \quad (7.13)$$

Combining equations (7.5), (7.12), and (7.13) completes the proof. \qedsymbol

To prove Lemma 7.3, we first need to show the following auxiliary claim.

**Lemma 7.24.** For any two points $p, q \in A\cup B$, let $\Box$ be the least common ancestor of $p$ and $q$ in $Q$, where $\Box$ is a cell in $G_i$, and let $(\Psi_p, \Psi_q)$ be the WSPD pair in $W_{\Box}$ that contains $p$ and $q$ respectively.
(i) If \( \|p - q\| \geq (144/\varepsilon)\mu_i \), then \( d_Q(p, q) + \mu_{pq}^2 \leq (1 + 3\varepsilon/8)\|p - q\|^2 \).

(ii) If \( \|p - q\| < (144/\varepsilon)\mu_i \), then the subcells that contain \( p \) and \( q \), i.e., \( \xi_p, \xi_q \), have a side-length of \( \mu_i \).

**Proof.** Let \( j = \lfloor i/2 \rfloor - 2 \log \frac{\log \Delta}{\varepsilon} - c_1 \). Then the minimum subcell size \( \mu_i \) is \( 2^j \). Let \( \Box_1 \) and \( \Box_2 \) be the two children of \( \Box \) such that \( p \) is inside \( \Box_1 \) and \( q \) is inside \( \Box_2 \). Let \( t \) be an integer such that \( \frac{144}{\varepsilon}2^t \leq \|p - q\| \leq \frac{144}{\varepsilon}2^{t+1} \). For (i), \( t \geq j \) and \( \xi_p \) (resp. \( \xi_q \)) is a cell of grid \( G_k \) (resp. \( G_{k'} \)) such that \( k \leq t + 1 \) (resp. \( k' \leq t + 1 \)). Let \( (\Psi_p, \Psi_q) \in \mathcal{W}_\Box \) be the representative pair of \( (p, q) \). The diameters of \( \xi_p \) and \( \xi_q \) are at most \( \sqrt{2} \times 2^{t+1} \leq \frac{\varepsilon}{36\sqrt{2}}\|p - q\| \), and, therefore,

\[
(1 + \varepsilon/12)\ell_{\text{max}}(\xi_p, \xi_q) \leq (1 + \varepsilon/12)(\|p - q\| + \sqrt{2} \times 2^{t+2}) \\
\leq \|p - q\|(1 + \frac{\varepsilon}{18\sqrt{2}})(1 + \varepsilon/12) \\
\leq (1 + \varepsilon/12)(1 + \varepsilon/24)\|p - q\| \\
\leq (1 + \varepsilon/8)\|p - q\|.
\]

Similarly, we can bound

\[
\mu_{pq}^2 \leq 2^{2t+2} \leq (\varepsilon/72)^2\|p - q\|^2 \leq (\varepsilon/72)\|p - q\|^2.
\]

Combining the previous two bounds together with (7.1) gives the following:

\[
d_Q(p, q) + \mu_{pq}^2 = \ell_{\text{max}}(\Psi_p, \Psi_q)^2 + \mu_{pq}^2 \\
\leq (1 + \varepsilon/12)^2\ell_{\text{max}}(\xi_p, \xi_q)^2 + \varepsilon/72\|p - q\|^2 \\
\leq (1 + \varepsilon/8)^2\|p - q\|^2 + \varepsilon/72\|p - q\|^2 \\
\leq (1 + 3\varepsilon/8)\|p - q\|^2.
\]
For (ii), observe that the distances of $\xi_p$ and $\xi_q$ to the boundaries of $\square_1$ and $\square_2$ respectively are less than $\frac{144}{\varepsilon}\mu_i$. From the subcell construction procedure, $\xi_p$ and $\xi_q$ should be cells of the minimum subcell size $\mu_i$. 

Finally, in the following Lemma, we argue that, for any edge $(p, q)$ the expected value of the quadtree distance $d_Q(p, q)$ plus the additional additive error $\mu_{pq}^2$ is at most $(1 + \varepsilon)/2$ times the squared-Euclidean distance $\|p - q\|^2$. Combining this with Lemma 7.23 and applying linearity of expectation immediately gives Lemma 7.3.

**Lemma 7.25.** Given a randomly shifted quad tree $Q$, for any pair of points $(p, q) \in A \times B$, 

$$
\mathbb{E}[d_Q(p, q) + \mu_{pq}^2] \leq (1 + \varepsilon/2)\|p - q\|^2.
$$

**Proof.** Let $\square \in G_i$ be the least common ancestor of $p$ and $q$ in $Q$, and let $\xi_p, \xi_q$ be the subcells that contain $p$ and $q$ respectively. Let $\mathbb{E}_i[d_Q(p, q)]$ be the expected value of of the distance given that $\square \in G_i$.

$$
\mathbb{E}[d_Q(p, q)] = \sum_{i=1}^{\log \Delta} \Pr[\square \in G_i] \mathbb{E}_i[d_Q(p, q) + \mu_{pq}^2]
$$

$$
\leq \sum_{i=1}^{\log \Delta} \Pr[\square \in G_i] \left( \mathbb{E}_i \left[ d_Q(p, q) + \mu_{pq}^2 \mid \|p - q\| \geq (144/\varepsilon)\mu_i \right] + \mathbb{E}_i \left[ d_Q(p, q) + \mu_{pq}^2 \mid \|p - q\| < (144/\varepsilon)\mu_i \right] \right)
$$

$$
\leq (1 + 3\varepsilon/8)\|p - q\|^2 + \sum_{i=1}^{\log \Delta} \Pr[\square \in G_i] \left( \mathbb{E}_i \left[ d_Q(p, q) + \mu_{pq}^2 \mid \|p - q\| < (144/\varepsilon)\mu_i \right] \right).
$$

To complete the proof, we upper bound the second term of the RHS by $(\varepsilon/8)\|p - q\|^2$. First, note that

$$
\Pr[\square \in G_i] \leq \|p - q\|_1/2^{i-1} \leq \|p - q\|^2/2^{i-1}.
$$
Since \(\|p - q\| < (144/\varepsilon)\mu_i\), by Lemma 7.24(ii), \(\xi_p\) and \(\xi_q\) have a side-length of the minimum subcell size \(\mu_i\) and therefore, we can bound

\[
d_Q(p, q) + \mu_{pq}^2 \leq (1 + \varepsilon/12) \ell_{\text{max}}(\xi_p, \xi_q)^2 + \mu_{pq}^2
\leq (1 + \varepsilon/12)(\|p - q\| + 2\sqrt{2}\mu_i)^2 + \mu_i^2
\leq (1 + \varepsilon/12)((144/\varepsilon)\mu_i + 2\sqrt{2}\mu_i)^2 + \mu_i^2
\leq 900\mu_i^2/\varepsilon^2.
\]

Recall that the minimum subcell size \(\mu_i = 2^{(i/2) - 2\log \log \Delta - c_1}\), where \(c_1 > 0\) is a constant. By setting \(c_1\) to be sufficiently large, we get \(\mu_i^2 \leq \varepsilon^4 2^i/(16 \log^4 \Delta)\). Therefore, \(d_Q(p, q) + \mu_{pq}^2 \leq \varepsilon 2^i/(16 \log \Delta)\), and we finally have,

\[
\sum_{i=1}^{\log \Delta} \Pr[\square \in G_i] \left( \mathbb{E}_i \left[ d_Q(p, q) + \mu_i^2 \mid \|p - q\| < (144/\varepsilon)\mu_i \right] \right) \leq \sum_{i=1}^{\log \Delta} \left( (\|p - q\|^2/2^{i-1}) \left( \frac{\varepsilon}{16 \log \Delta} \right)^2 \right)
\leq \varepsilon/8 \|p - q\|^2,
\]

as desired.

7.7.3 Computing an \((1+\varepsilon)\)-Approximate Matching with High Probability

From Lemma 7.3, we have

\[
\mathbb{E} \left[ \sum_{(p, q) \in M_{\text{OPT}}} d_Q(p, q) + \mu_{pq}^2 \right] \leq (1 + \varepsilon/2)w(M_{\text{OPT}}).
\] (7.14)
However, it is desirable to remove the need for expected values. Instead, we explain how to ensure that
\[
\sum_{(p,q)\in M^{\text{OPT}}} d_Q(p,q) + \mu_{pq}^2 \leq (1 + \varepsilon)w(M^{\text{OPT}}) \tag{7.15}
\]
with high probability. We can then design a $\tilde{O}(n^{5/4})$ time algorithm for computing a $Q$-optimal matching under the assumption that (7.15) holds. To ensure this assumption, we can execute our algorithm $\log_2(n)$ times, and among all executions that terminate in $\tilde{O}(n^{5/4})$ time, use the one that produces the smallest cost. Within each execution, from (7.2) we have
\[
\begin{align*}
\textstyle w(M^{\text{OPT}}) & = \sum_{(p,q)\in M^{\text{OPT}}} \|p - q\|^2 \\
\textstyle & \leq \sum_{(p,q)\in M^{\text{OPT}}} d_Q(p,q) + \mu_{pq}^2.
\end{align*}
\]

Furthermore, by combining this with (7.14) we have that (7.15) holds with probability at least $1/2$. Therefore, the probability that (7.15) is satisfied by at least one of the $\log_2(n)$ random shifts is at least $1 - 1/n$. We present an $\tilde{O}(n^{5/4})$ time algorithm for computing a $Q$-optimal matching under the assumption that (7.15) holds. By combining this assumption, with Lemma 7.23, we have that any $Q$-optimal matching $(1 + \varepsilon)$-approximates the optimal RMS matching. Therefore, to prove Theorem 7.1, it is sufficient to give an $\tilde{O}(n^{5/4})$ time algorithm for computing a $Q$-optimal matching.
Chapter 8

Conclusion

The goal of this work was to investigate new techniques for exploiting graph separators to design asymptotically faster matching algorithms. As part of this work, we have successfully designed new techniques and demonstrated their applicability towards the design of matching algorithms in the settings of planar graphs, minor-free graphs, and even arbitrary graphs with recursively-computable balanced separators. Furthermore, we have demonstrated the applicability of these techniques towards the design of asymptotically faster algorithms for geometric matching problems.

The novelty of this work primarily arises from a deeper understanding of the analyses of the Hopcroft-Karp and Gabow-Tarjan matching algorithms. Perhaps the most important contribution of this work is a new strategy for bounding the number of affected pieces, a key component of each algorithm in Chapters 4–7. In addition, this work demonstrates the power that can be gained from a non-uniform distribution of error on the dual feasibility conditions, a technique that is very flexible in the design of matching algorithms.

Among the new results presented in this work, the main algorithm of Chapter 4 stands out for its simplicity and practicality, despite being, in some ways, the most general result. Despite being the simplest algorithm, it was neither the first, nor the second, but the third algorithm designed as part of this work, being written after the algorithms of Chapters 5 and 6.
An optimized, slight modification of the bottleneck matching algorithm of Section 4.5 has been implemented as part of the Master’s degree thesis work of Jiacheng Ye [60]. His experimental results suggest that our algorithm performs significantly faster than Hopcroft-Karp on large random inputs. In addition, there are ongoing efforts to apply observations from this work towards the design of practical approximation algorithms for optimal transport, which has many applications in machine learning. Lahn et al. [32] presented a fast additive approximation algorithm for computing the optimal transport on a complete graph with arbitrary costs, but this algorithm is not easily parallelizable. However, when the ground metric is given by geometric costs, it seems possible to apply techniques from, for example, the algorithm of Chapter 7 to design even faster parallelizable versions of the algorithm of [32]. These ongoing works demonstrate the practical implications of the current work in the applications domain.

Much of the difficulty of this work arose from the complexity of integrating our new primal-dual techniques into domain-specific contexts. While at its core, all presented algorithms are executing either exact or approximate searches for minimum net-cost augmenting paths in a residual graph, the introduction of a set of dual feasibility conditions is a necessary component for the sake of efficiency.

The most in aspect of the work was the careful definition and maintenance of a set of compressed feasibility conditions, a key component of the results in Chapters 5, 6, and 7. Despite these compressed feasibility conditions being similar at a high level, each set was technically unique from all previous work. For example, the algorithm of Chapter 6 is, to this author’s knowledge, the first one to exploit a relaxed set of admissibility conditions in order to find many more augmenting paths per phase, augmenting paths that were only partially disjoint instead of fully disjoint. While this observation was the key insight, integrating it into the existing framework of the Chapter 5 algorithm required the introduction of new
subsidiary techniques, such as the detection and removal of admissible alternating cycles. Each of these three results required a different set of procedures for converting between the compressed feasibility and the underlying feasibility conditions, as well as a different procedure for facilitating augmenting path searches over the compressed graph.

This work falls at the intersection of algorithmic domains, combining insights on classical bipartite matching algorithms for arbitrary graphs with more recent advances in planar graphs and geometric settings. Most notably, this work showed how strong the relationship is between algorithms on planar graphs, and geometric graphs. In particular, the RMS matching result of Chapter 7 was heavily influenced by techniques for planar graph matching presented in Chapter 5. It seems conceivable that similar insights from the geometric settings may be useful in the planar graph context as well.

In the RMS matching algorithm, it was observed that the majority of augmenting paths found during earlier phases of the algorithm could be localized to smaller active cells. Once cells of layer $i$ become active, there are roughly only $\tilde{O}(n/2^i)$ unmatched vertices remaining, so fewer Hungarian searches operate at higher levels of the quad-tree. Interestingly, a similar layered approach, in the style of the Lipton-Tarjan algorithm, can be applied to the algorithm of Chapter 5. By doing so, the number of Hungarian searches at the top-most subproblem can be reduced to $O(\sqrt{n})$ instead of the $O(n/\sqrt{r})$ searches that occur after the preprocessing step of Chapter 5. This observation can be used to obtain an $\tilde{O}(n^{5/4} \log nC)$ time algorithm for minimum-cost perfect matching on planar graphs, but this technique was discovered in conjunction with the discovery of the $\tilde{O}(n^{6/5} \log nC)$ algorithm from Chapter 6. Furthermore, the technique seems to offer no additional benefit to the algorithm of Chapter 6.

For some of the results presented here it is unclear how any further improvements can occur without the introduction of fundamentally new and very different techniques. For example, it is unclear how the the main result of Chapter 4 can be improved a set of techniques similar
to those presented here. However, in other respects, it is likely that several more results will result from this framework. In the remainder of the chapter, we discuss the prospects for further applications of this framework towards future algorithmic results, both in the fashion of improving the complexities of the current work, as well as extending the techniques to more general settings.

8.1 Open Questions for Planar Graphs

This work establishes an $\tilde{O}(n^{6/5} \log(nC))$ time algorithm for computing a minimum-cost perfect matching in a bipartite planar graph. However, it seems very unlikely this is the end of the story. There are a few open questions remaining for planar graphs:

- For several years prior to this work, there was a near-linear time algorithm for the unweighted matching problem on planar graphs [9], while the best algorithm for the weighted version was, more or less, an algorithm for general graphs [18], with a running time of $\Omega(n^{3/2})$. It seems reasonable to expect that there is some way to close this polynomial gap between the weighted and unweighted versions of the problem. While the current work has significantly narrowed the gap, it seems likely that $\tilde{O}(n^{6/5} \log(nC))$ is not the right bound for weighted matching. Is a near-linear time algorithm possible?

- The current $\tilde{O}(n^{6/5} \log(nC))$ time result for minimum-cost matching is only able to compute a perfect matching; it is not currently clear how to trivially modify our algorithm to work for the maximum-cardinality matching case. The reason for this is somewhat subtle. Roughly speaking, it is likely that the current algorithm within a scale can be modified to work for the maximum-cardinality case, with some modifications to the analysis. However, the correctness argument for the maximum-cardinality
case requires certain assumptions on the dual weights, which are satisfied with respect to the dual weights within a scale (i.e., with respect to the reduced costs), but are not necessarily satisfied by the dual weights between the two scales (i.e., with respect to the original costs). Gabow and Tarjan gave a reduction from the minimum-cost maximum-cardinality matching problem [18], but the reduction inherently violates planarity. An extensive follow-up work for the minimum-cost maximum-cardinality matching case was presented by Ramshaw and Tarjan [46], but more investigation is needed to figure out how applicable these techniques are to our context. Can we generalize the algorithm of Chapter 6 to the minimum-cost maximum-cardinality matching problem?

- All existing algorithms for both weighted and unweighted matching problems on non-bipartite planar graphs take $\Omega(n^{3/2})$ time. Can our algorithms for planar graphs be extended to the non-bipartite case (see Section 8.5)?

## 8.2 Open Questions for Minor-Free Graphs

The weighted matching algorithm of Chapter 6 is actually not restricted to minor-free graphs, but generalizes to any graph with sufficiently small, recursively computable balanced separators. However, unlike the main algorithm of Chapter 4, which breaks the $\Omega(n^{3/2})$ bound for any sub-linear size separator, the algorithm of Chapter 6 has a much higher dependency on the separator size and does not run in $o(n^{3/2})$ time, unless the separator size is a smaller polynomial in $n$. For example, the algorithm of Chapter 6 takes $\Omega(n^{3/2})$ time when the separator size is $\Omega(n^{2/3})$. As a result, there remains a huge running time gap between the weighted algorithm of Chapter 6 and the unweighted algorithm of Chapter 4. Can this gap be closed?
8.3 Open Questions for Geometric Matching

The current work on RMS matching, described in Chapter 7, gives a $\tilde{O}(n^{5/4}\text{poly}(1/\varepsilon))$ time algorithm for computing a $(1 + \varepsilon)$ approximation for two-dimensional points. However, it seems as if a near-linear time relative approximation algorithm should be possible. Specifically, the current algorithm searches over the associated graph (the RMS algorithm’s equivalent of the planar dense distance graph) in time linear in the number of internal edges. Is it possible to apply a search technique, similar to FR-Dijkstra [15], on this associated graph to reduce the search time to near-linear in the number of vertices in the associated graph? Doing so would result in a near-linear-time approximation algorithm for two-dimensional points. Furthermore, is it possible to generalize the approaches discussed here to yield an $o(n^{3/2})$ algorithm for any $d > 2$?

The two geometric matching algorithms presented in this work are both approximation algorithms. Currently, there are no known $o(n^{3/2})$ exact algorithms for the Euclidean bipartite matching problem, the RMS matching problem, or the bottleneck matching problem. Can the approaches discussed here be applied to exact geometric matching problems?

Finally, there is a question of whether the techniques presented in this work can be extended to arbitrary metric spaces. Given a set $V$ of $n$ points with arbitrary metric costs $d(\cdot, \cdot)$ between them, there is a notion of a $\beta$-padded decomposition, which can be used to partition the graph in a similar fashion to graph separators. A $\beta$-padded decomposition of $(V, d)$ is a partition of $V$ into pieces $V_1, \ldots, V_g$ such that (i) for any piece $V_j$, the longest edge of $V_j \times V_j$ is at most some value $\Delta$, and (ii) the probability that any edge $(a, b)$ is split between two different pieces is at most $d(a, b)\beta/\Delta$. In other words, each piece is small in terms of diameter and short edges have a low probability of being split between pieces. Given an arbitrary $n$ point metric space, it is possible to compute a $\beta$-padded decomposition for $\beta = O(\log n)$ [7].
Is it possible to use $\beta$-padded decomposition to extend the approaches used in this work to arbitrary metric costs?

### 8.4 Extensions to Flow Problems

It is well-known that bipartite matching problems can be solved using network-flow algorithms, implying that network flow problems are more general than matching problems. Can the approaches described in this work be extended to the minimum-cost maximum-flow problem? After the work described in Chapters 5 and 6, it was shown by Karczmarz and Sankowski [24] how to compute a minimum-cost flow in a planar multigraph with unit capacities and maximum edge cost $C$ in $\tilde{O}((mn)^{2/3} \log C)$ time, which can be seen as analogous to the $\tilde{O}(n^{4/3} \log nC)$ bound achieved by the minimum-cost perfect matching algorithm of Chapter 5. However, their algorithm does not make explicit use of dual weights. Is it possible to improve this algorithm by integrating the additional techniques introduced in Chapter 6?

### 8.5 Extensions to Non-Bipartite Graphs

For the algorithms discussed in this work, it seems likely that some extension to the non-bipartite matching case is possible. Most notably, the algorithm of Chapter 4 has a natural high-level interpretation in the context of non-bipartite graphs, but the actual implementation of the algorithm for a phase is highly nontrivial, a trend that has permeated the history of bipartite vs. non-bipartite matching algorithms. The main difficulty introduced by applying this framework in the non-bipartite setting results from the fact that internal edges within a piece can be reused multiple times during a phase. Modifying existing algorithms for non-bipartite matching on general graphs, which are already quite complicated,
to account for this change seems possible, but technically very difficult. A similar trend applies to prospects of extending the results of Chapters 5 and 6 to non-bipartite planar and minor-free graphs.
Bibliography


