Combinatorial Algorithms for Server Allocation Problem

Rachita Sowle

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
Liqing Zhang
Chittaranjan Tripathy

August 2, 2024
Blacksburg, Virginia

Keywords: Some Keywords, Subject matter, etc.

Combinatorial Algorithms for Server Allocation Problem

Rachita Sowle

(ABSTRACT)

Motivated by problems in logistics, image recognition, and statistics, we consider the server allocation problem. In this problem, we are given k servers (with capacities) and n requests, which are points in a metric space. A server serves a request by moving to the request location, and the goal is to serve all requests while minimizing the total movement of servers, subject to the constraint that the number of requests served by a server cannot exceed its capacity. When the server capacity is 1, and for the Euclidean metric, the problem reduces to the Euclidean bipartite matching problem. When the capacity is ∞, suppose we are also provided with the order in which requests are to be served, the problem is the k-first come first served routing problem. We also consider a generalization of the k-first come first served routing problem to the taxi allocation problem, where each request is associated with a pickup location, dropoff location, and pickup time, and the server's velocity is also given as input.

We present new algorithms for the Euclidean bipartite matching problem, showing improvements over existing algorithms. In particular, for two point sets $A, B \subset \mathbb{R}^d$ with $|A| = |B| = n$ and dimension $d > 1$ being constant, we developed:

- A faster algorithm that computes an ε-approximate minimum-cost perfect matching in $O(n(\varepsilon^{-O(d^3)} \log\log n + \varepsilon^{-O(d)} \log^4 n \log^5 \log n))$ time. This is an improvement over previous algorithms, which took $n(\varepsilon^{-1} \log n)^{\Omega(d)}$ time.

- An algorithm that boosts the accuracy of any ε-additive approximation algorithm, achieving an expected additive error of $\min\{\varepsilon, (d \log\log n)w^*\}$ from the optimal matching cost w^* in $O(T(n, \varepsilon/d) \log\log n)$ time, where $T(n, \varepsilon)$ is the time complexity of any given ε-additive approximation algorithm.

For the k-first come first served routing problem, we present the following results.

- The online version of the k-first come first served routing problem is the celebrated k-server problem. The best-known online algorithm for this problem is the Work Function algorithm. We present a new implementation of the work function algorithm, where processing the ith request takes $O((i+k)^2)$ time, improving on the previous methods that take $\Omega(k(i+k)^2)$ time [93], [91].

- For the offline setting, we show that the k-first come first served routing problem and the taxi allocation problem can be reduced to the minimum-cost bipartite matching problem. Using this reduction,

  - we develop a time-based divide-and-conquer algorithm to obtain an optimal solution in $\tilde{O}(kn^2)$ time, which can be further improved to $\tilde{O}(kn)$ when the requests and servers are in two-dimensional Euclidean space, and,

  - we apply a recent geometric divide-and-conquer algorithm presented in [55] to obtain an optimal solution for the taxi routing problem in a two-dimensional Euclidean space. As a result, we obtain significant empirical performance improvements for the taxi allocation problem in a two-dimensional space where the cost of moving from one location to another is lower bounded by the Euclidean cost.

Combinatorial Algorithms for Server Allocation Problem

Rachita Sowle

(GENERAL AUDIENCE ABSTRACT)

In the server allocation problem, we are given n requests and k servers, both as points in a space. A server can serve a request by moving to the request location, and each request can be served by exactly one server. The objective is to optimize the allocation of servers to requests such that the total distance traveled by the servers is minimized. In this thesis, we present efficient algorithms for three specific problems in the server allocation problem framework. First, we consider the case when the servers are restricted to serving up to one request each. This problem reduces to the well-known minimum cost maximum cardinality bipartite matching problem. When the underlying distance is Euclidean, this problem is called the Euclidean bipartite matching problem. We present two efficient algorithms that improve the state-of-the-art for the Euclidean bipartite matching problem. Second, we consider the case when the servers can handle multiple requests. Assuming the requests are given as ordered sequences, a server can serve a subsequence of the request sequence. We devise two efficient algorithms for this problem and show empirical performance improvements on the New York Taxi data set. Third, we consider the scenario when the requests appear in an online fashion such that on the arrival of each request, a server must be allocated to it immediately and irrevocably. This problem is the celebrated k-server problem. The work function algorithm is an online algorithm that solves the k-server problem with the best-known competitive ratio. We present a new, faster implementation of the work function algorithm.

Dedication

To my mother, who always believed in me and encouraged me every step of the way.

Though you couldn't see this completion, I know you are proud.

And to my husband, thank you for being my rock throughout.

Acknowledgments

First, I want to express my deepest gratitude to my PhD advisor, Dr. Sharath Raghvendra, for being everything I could have hoped for in my advisor. Your support, advice, and commitment to my growth as a researcher have been invaluable. I am very grateful for your guidance and encouragement, which have shaped my academic and professional development in countless ways. I would also like to thank the rest of my dissertation committee for their valuable feedback and support. Your insights helped me refine my work and broaden my perspective. Additionally, I am grateful to the faculty and staff at Virginia Tech for their continuous support. Your mentorship, administrative assistance, and encouragement have greatly contributed to my success, and I am fortunate to have been part of such a supportive environment. I am thankful for the collaboration with my co-authors. Working with you has been a valuable learning experience, and I appreciate the knowledge and ideas you brought to our work together. To my fellow graduate students at Virginia Tech, your passion, ideas, and friendship have been a constant source of motivation. Thank you for being my community and support system. I look forward to our continued friendships and future collaborations. I am deeply thankful for the financial support provided by various sources. This work was supported in part by the National Science Foundation grant CCF-1909171 and by the continuous support from Virginia Tech's Graduate School. This financial assistance was crucial to let me focus on my research and achieve my academic goals. Finally, I wish to acknowledge the encouragement and love of my family and friends, who have been my foundation.

Contents

List of Figures

Chapter 1

Introduction

The field of combinatorial optimization is continuously growing in its practical importance. It relates to the methods that identify the best solution from a large finite set of possibilities. The rapid growth in new technology and digital transformation are a few of the many reasons there is a growing interest in building efficient algorithms. This includes many interesting problems from various fields, including logistics, image recognition, operating systems, and statistics. Across all these sectors, we frequently deal with scenarios that include a set of requests and servers (with capacity) as points in space, and each request must be served by a server by moving the server to the location of the request. Each server has a capacity that suggests the maximum number of requests it can serve. To serve any request, one of the servers with positive capacity is moved to the location of this request, and after this movement, the server capacity is reduced by one. This raises really interesting questions, such as how one can assign servers to the requests to minimize the total movement of servers. We refer to this problem as the *Server Allocation Problem*. An allocation of a server for each request is called a solution to the problem.

In most applications related to pattern recognition, shape matching, statistics, and VLSI, there are a set of unit supply points and a set of unit demand points, and each demand can be served by moving a supply point to the demand's location. A supply point, in this case, can serve only one demand. We can state the problem as a server allocation problem where supply points are the set of servers, the demand points are the set of requests, and the

capacity of each server is 1. The goal of the problem is to allocate a server to each request so that the total cost of serving the requests is minimized. This is a fundamental combinatorial optimization problem called the Assignment Problem, also known as the minimum-cost Bipartite Matching Problem. When the points are in Euclidean space, we call it *Euclidean Bipartite Matching Problem* (EBM). We discuss the details of this problem in Section 1.1 of this chapter.

On the other hand, consider the case where there is no restriction on the number of requests a server can serve, i.e., the capacity of each server is ∞. Now consider the *online* case where requests arrive over time and one has to immediately assign a server to it. More specifically, given a set of k servers and a request sequence $R = \langle r_1, r_2, \ldots, r_n \rangle$ arranged in the order of arrival, then on the arrival of each request $r_i \in R$, an online algorithm must serve it by moving one of the servers to the location of r_i irrevocably, and a new request appears only after the current request has been served. This problem is well known as the k-server problem. For any online algorithm, its competitive ratio is a measure for identifying how well the algorithm performs compared to the optimal offline algorithm, which has complete knowledge of the sequence of requests in advance. The work function algorithm is a classical online algorithm that has been shown to achieve the best-known bound on the competitive ratio for the k-server problem [68]. We will expand on the specifics of the work function algorithm for the k-server problem in Section 1.2 of this chapter.

We also consider the problem of finding the optimal offline solution to the k-server problem and refer to it as the k-first come first served routing problem (k-FCFSRP). More precisely, in the k-FCFSRP problem, given a set of servers and a request sequence $R = \langle r_1, r_2, \ldots, r_n \rangle$, we have to assign a server to serve each of the request in the sequence with the additional constraint that, for any two requests $r_i r_j \in R$ served by the same server, if $i < j$, then r_i must be served before r_j.

An area where the k-FCFSRP naturally finds application is transportation and logistics. For instance, a taxi service agency that only allows pre-booking and gathers customer requests before dispatching the taxis would be interested in determining a set of routes for their taxis that serve all the requests at the requested time while minimizing the total distance traveled by taxis. We extend the k-FCFSRP by incorporating logistical constraints that such taxi service agencies will likely have, leading to the formulation of the *taxi allocation problem* (TAP). Additional constraints in the taxi allocation problem that are not present in the k-FCFSRP are as follows.

- Unlike in the k-FCFSRP, where each request is a single point, in the taxi allocation problem, each request is associated with a pickup location, a dropoff location, and a pickup time. The requests must be served at their pickup times.
- In the k-FCFSRP, it is assumed that a server can immediately serve a request. As a result, given two requests r_i and r_j with $i < j$, a server can always serve r_j after serving r_i. However, in the taxi allocation problem, all the servers travel with a fixed velocity, which is given as input. Therefore, a taxi cannot serve r_j after r_i if the pickup time of r_i and the pickup time for r_j are close (say just one second) to each other but the distance between the dropoff location of r_i and the pickup location of r_j is high (say 10 kilometers).

The k-FCFSRP is a special case of the taxi allocation problem (TAP). An instance of TAP is k-FCFSRP when the given taxis have infinite speed, and the pickup and dropoff points are co-located for each request. We describe these problems formally in section 1.3 of this chapter.

Our results. Our main results are the following,

(1) For the Euclidean bipartite matching, we present a faster algorithm that computes a ε-approximate minimum-cost perfect matching. To be more specific, for two point sets $A, B \subset \mathbb{R}^d$, with $|A| = |B| = n$ and dimension $d > 1$ a constant, and for a parameter $\varepsilon > 0$, our algorithm runs in $O(n(\varepsilon^{-O(d^3)} \log\log n + \varepsilon^{-O(d)} \log^4 n \log^5 \log n))$ time. All previous algorithms take $n(\varepsilon^{-1} \log n)^{\Omega(d)}$ time.

(2) For Euclidean bipartite matching when points are in high dimensional space, we present an algorithm that boost the accuracy of any ε-additive approximation algorithm that runs in $T(n, \varepsilon)$ time to an expected additive error of $\min\{\varepsilon, (d \log\log n)w^*\}$ from the optimal matching cost w^* in $O(T(n, \varepsilon/d) \log\log n)$ time.

(3) We present a new implementation of the work function algorithm for the k-server problem. Our method serves the ith request in $O((i+k)^2)$ time which is faster than previous methods that take $\Omega(k(i+k)^2)$ time [93], [91].

(4) We show that the k-FCFSRP and the TAP can be reduced to the minimum-cost bipartite matching problem. Using this reduction,

   – we develop a time-based divide-and-conquer algorithm to obtain an optimal solution in $\tilde{O}(kn^2)$ time, which can be further improved to $\tilde{O}(kn)$ in a two-dimensional space by utilizing dynamic weighted nearest neighbor data structures, and,

   – we applied a recent geometric divide-and-conquer algorithm presented in [55] to obtain an optimal solution in a two-dimensional Euclidean space. As a result, we are able to achieve a sub-quadratic time complexity for the k-FCFSRP in a two-dimensional Euclidean space and significant empirical performance improvements for the TAP compared to the state-of-the-art in a two-dimensional space where the cost of moving from one location to another is lower bounded by the Euclidean cost.

1.1 Euclidean Bipartite Matching (EBM)

In this section, we describe the Euclidean bipartite matching problem and discuss certain related challenges. For simplicity, we limit our discussion to the case where the number of requests equals to the number of servers. All the definitions and methods can be easily extended to cases when the number of requests and servers is unequal. Next, we describe the problem formally.

Let $A, B \subset \mathbb{R}^d$ be two point sets of size n each, where $d > 1$ is a constant. Let the distance function $\mathsf{d}(r, s)$ be defined as the Euclidean distance between any two points r and s where $r, s \in \mathbb{R}^d$. Let $\mathcal{G} = (A \cup B, A \times B)$ be a weighted complete bipartite graph where the weight of an edge $(a, b) \in A \times B$ is $\mathsf{d}(a, b)$. A set of vertex disjoint edges in $\mathcal{G}$ is called a *matching*. A matching in $\mathcal{G}$ that has a cardinality n is called a *perfect matching* in $\mathcal{G}$. The cost of a matching M, denoted by $\mathscr{c}(M)$, is $\mathscr{c}(M) = \sum_{(a,b)\in M} \mathsf{d}(a, b)$. An optimum EBM in $\mathcal{G}$, denoted by M^*, is a perfect matching in $\mathcal{G}$ of the minimum cost.

All existing exact solutions, such as the Hungarian algorithm([71]) to solve the bipartite matching problem, are too slow to be practical. There have been improvements in the efficiency of exact algorithms when the points are in Euclidean space using geometric properties, but these algorithms are still very slow for practical applications. This has shifted the focus towards the design of approximate algorithms. Any approximation can be classified fundamentally as a multiplicative approximation (also called relative approximation) or additive approximation which we describe next.

For any $\varepsilon > 0$, a perfect matching M in $\mathcal{G}$ is called an *ε-approximate matching* if $\mathscr{c}(M) \leq (1 + \varepsilon)\mathscr{c}(M^*)$ and M is called an *δ-close matching* if $\mathscr{c}(M) \leq \mathscr{c}(M^*) + \delta$.

For a parameter $\delta > 0$, a δ-close approximation of the matching has a cost that is bounded by $\mathscr{c}(M^*) + nC_{\max}\delta$, where C_{max} is the largest cost of an edge in the complete bipartite graph.

A generalization of the EBM problem is the 1-Wasserstein problem. Informally, given two (possibly continuous) probability distributions μ and ν whose support is a unit square in the Euclidean space, the *optimal transport* is the cheapest way of transporting mass from μ to ν; here the cost of transporting δ mass from points a to b is the $\delta\|a-b\|$. The cost of this optimal transport is also known as the 1-Wasserstein distance. For discrete distributions, the optimal transport plan can be computed by solving a generalization of the EBM called the *Eucldiean transportation problem*.

When μ and ν are continuous distributions, one can draw two sets A and B of n samples each from μ and ν, respectively. One can approximate the 1-Wasserstein distance between μ and ν by solving the EBM Problem between A and B. This approximation has been extensively used in generative models [56, 94], robust learning [49], and parameter estimation [25, 77].

The additive approximation algorithms are good when $\mathrm{¢}(M^*)/n \approx C_{\max}\delta$. However, when $\mathrm{¢}(M^*)/n$ becomes significantly smaller than $C_{\max}\delta$, these algorithms produce sub-optimal transport plans where the error of $+nC_{\max}\delta$ dominates the cost. As an example, when A and B are samples drawn from the same distribution, as n approaches ∞, $\mathrm{¢}(M^*)/n$ approaches 0. In such scenarios, additive approximation methods compute transport plans with large errors. In contrast, relative approximation algorithms will return a more accurate solution. One of our contribution is to design an approximation algorithm that combines the guarantees of relative and additive approximation methods.

This completes our discussion on the motivation for developing the above EBM algorithms. Next, we describe the work function algorithm for the k-server problem.

1.2 k-server problem

In this section, we describe the k-server problem and the work function algorithm, which is an online algorithm for the k-server problem.

Problem Statement: Consider a vertex set V and a weighted complete graph where each edge $(u, v) \in V \times V$ has a cost $d(u, v)$. We assume that $d(\cdot, \cdot)$ is a metric. Given any two multi-sets A and B of points with $A, B \subseteq V$ and $|A| = |B|$, we use $d(A, B)$ to denote the minimum-cost bipartite matching of the points in A to points in B under the distance $d(\cdot, \cdot)$. For an integer $k > 0$, we are given k identical servers and their initial locations, also called the *initial configuration* $\mathcal{C}^0 = \{s_1^0, \ldots s_k^0\}$ in the metric space. A *configuration* is simply any multi-set $\mathcal{C} \subset V$, with $|\mathcal{C}| = k$. We use configurations to denote the locations of the k servers. For any request r_i, a configuration $\mathcal{C}$ *serves* r_i if the location of r_i is contained in the multi-set $\mathcal{C}$. In other words, a server $s \in \mathcal{C}$ that is co-located with r_i serves r_i at zero cost. We are also given a sequence of n requests $R = \langle r_1, \ldots, r_n \rangle$ that arrive over time with r_i arriving at time $t = i$. After r_i arrives, we move the servers to a configuration $\mathcal{C}^i = \{s_1^i, \ldots, s_k^i\}$ that serves i. The input to the k-server problem is simply the initial configuration $\mathcal{C}^0$ and the request sequence R.

A *valid* solution to the problem is any sequence of configurations $\sigma = \langle \mathcal{C}^0, \mathcal{C}^1 \ldots, \mathcal{C}^n, \mathcal{C}^{n+1} \rangle$ where $\forall 1 \leq i \leq n, \mathcal{C}^i$ serves request r_i. Note that, we set the *final configuration* $\mathcal{C}^{n+1}$ to be the same as $\mathcal{C}^n$ unless otherwise specified. Furthermore, we define the points within the final configuration as *anchor nodes*. The cost of σ, denoted by $w(\sigma) = \sum_{i=0}^{n} d(\mathcal{C}^i, \mathcal{C}^{i+1})$. The *optimal solution*, denoted by $\sigma^*_{\mathcal{C}^0,R}$ is a valid solution with the smallest possible cost when the input is the initial configuration $\mathcal{C}^0$ and the request sequence is R. We denote $\sigma^*_{\mathcal{C}^0,R}$ as σ^* when the request sequence R and the initial configuration $\mathcal{C}^0$ are obvious from the context. Let R_i be the sequence of first i requests, i.e., $R_i = \langle r_1, \ldots, r_i \rangle$. We use σ_i^* to

denote $\sigma^*_{\mathcal{C}^0,R_i}$. Suppose, in addition to the initial configuration $\mathcal{C}^0$ and the request sequence R_i, the problem also specifies a final configuration $\mathcal{C}$, then $\sigma^*_i(\mathcal{C})$ denotes the minimum-cost solution that places the k servers in the initial configuration $\mathcal{C}^0$ after which it serves the i requests, and ends with $\mathcal{C}$ as the final configuration.

In the k-server problem, when a request r_i arrives, one has to immediately and irrevocably commit to a configuration $\mathcal{C}^i$ that serves request r_i. For any algorithm $\mathcal{A}$, let $\sigma_{\mathcal{A}} = \sigma_{\mathcal{A},\mathcal{C}^0,R}$ be the sequence of configurations that $\mathcal{A}$ chooses for an input request sequence R. Then, for a constant $\beta > 0$, we say that $\mathcal{A}$ has a competitive ratio of α if $w(\sigma_{\mathcal{A},\mathcal{C}^0,R}) \leq \alpha w(\sigma^*_{\mathcal{C}^0,R}) + \beta$, for all possible request sequences R. The competitive ratio indicates how well an online algorithm performs in the worst-case scenario compared to the best possible solution that knows all requests in advance. The work function algorithm is a classical online algorithm that has been shown to archive the best-known bound on the competitive ratio for the k-server problem [68].

At any time step i, a *lazy* algorithm will move only one server, say from some location r_j to the location of r_i. Due to the metric properties of cost, any valid solution can be converted to a *lazy solution* without increasing the cost (proof is available in Section 5.6.1 where we discussed the properties of a lazy and valid solution). Therefore, we restrict ourselves to lazy algorithms. Next, we describe the greedy, retrospective, and work function algorithms for the k-server problem.

Greedy Algorithm: Given a request r_i, the greedy algorithm will find a configuration $\mathcal{C}^i$ that serves request r_i such that $d(\mathcal{C}^{i-1}, \mathcal{C}^i)$ is as small as possible. A greedy algorithm is also a lazy algorithm, and configuration $\mathcal{C}^i$ can be obtained from $\mathcal{C}^{i-1}$ by moving the server $s \in \mathcal{C}^{i-1}$ that is nearest to r_i. The greedy algorithm is computationally inexpensive and seems to produce good-quality assignments when data is drawn from some known distributions. This approach is fairly popular [13, 78, 102, 105, 106] in real-world applications. However,

it has two major drawbacks. First, an outlier request may draw a service provider away from a request hotspot, leading to underutilization of this server. Second, real-world request patterns evolve throughout the day. For instance, a new request hotspot may emerge after the end of a musical concert. Greedy methods do not learn and relocate servers to better serve such hotspots, leading to higher costs.

Retrospective Algorithm: Given a request r_i, the retrospective algorithm chooses $\mathcal{C}^i$ to be the final configuration of the optimal solution σ_i^* of the first i requests. It can be shown that the retrospective algorithm is also a lazy algorithm (Lemma 5.19). The retrospective algorithm can potentially counter the drawbacks of the greedy algorithm. This is because the optimal solution σ_i^* would have moved the servers to match the request hotspots, and the retrospective algorithm benefits by doing the same. Unfortunately, the optimal solutions σ_i^* and σ_{i+1}^* and their final configurations may differ, causing a high server relocation cost.

Work Function Algorithm: Given a request r_i, the work function algorithm combines the greedy and retrospective algorithms as follows. The algorithm chooses a configuration $\mathcal{C}^i$ that serves the request r_i and minimizes the sum of greedy and retrospective costs:

$$\mathcal{C}^i = \operatorname*{argmin}_{\mathcal{C}}(w(\sigma_i^*(\mathcal{C})) + d(\mathcal{C}^{i-1}, \mathcal{C})). \tag{1.1}$$

Note that the minimum is over all possible configurations, i.e., every multi-set of size k. However, the work function algorithm is a lazy algorithm. A simple proof to show that the Work Function Algorithm is lazy can be found in [68] (ending in a discussion after Definition 2.2 of [68]) on pages 974–975. Several subsequent papers, for instance [93] have used this property. So we can conclude that the configuration $\mathcal{C}^i$ that minimizes equation 1.1 is obtained

by moving one server in $\mathcal{C}^{i-1}$ to the location of r_i. Therefore, we can rewrite equation 1.1 as

$$\mathcal{C}^i = \underset{s \in \mathcal{C}^{i-1}, \mathcal{C} = \mathcal{C}_{i-1} \setminus \{s\} \cup \{r_i\}}{\operatorname{argmin}} \left(w(\sigma_i^*(\mathcal{C})) + d(s, r_i) \right). \tag{1.2}$$

The k-server problem and its variants, such as the paging problem, have found applications in statistics [88], mathematics [45], operations research [85], layered graph traversal [31], metrical task systems [36], and page migration [39]. The k-server problem has also played a crucial role in developing competitive analysis for online algorithms [29, 101].

We design a polynomial combinatorial algorithm that iteratively processes each request in order such that after processing the ith request, we obtain a solution that serves the first i requests and satisfies Equation equation 1.2. Our algorithm is a new implementation of the Work Function algorithm. Our algorithm processes the ith request by executing a single Dijkstra's shortest path search on a carefully defined weighted graph in $O((i+k)^2)$ time, which is faster than previous methods that take $\Omega(k(i+k)^2)$ time [93], [91].

1.3 k-First-come First-server Routing Problem

The k-FCFSRP is the offline version of the k-server problem. Rather than directly adopting the description of the k-server problem, we describe the k-FCFSRP problem statement as follows.

Problem Statement. We are given k servers and their initial locations; let server i be located at s_i. Let $\mathcal{R} = \langle r_1, \ldots, r_n \rangle$ denote the sequence of requests that are in increasing order of the time when they must be served. For any two locations p and q, let $w(p, q)$ denote the cost of moving from p to q. A *routing plan* Γ for any server $s \in \mathcal{S}$ is a sub-sequence

$\Gamma(s) = \langle r_{i_1}, \ldots, r_{i_t} \rangle$ of the requests where server s serves these requests in the order specified by the subsequence. The cost of a routing plan $\Gamma(s)$ of server s is defined as the total cost of the movement of server s and is denoted as $w(\Gamma(s))$. A set of k routing plans, one for each of the k servers, is *complete* if every request in $\mathcal{R}$ is served by exactly one server. A complete set of routing plans is also called a *valid* solution to the problem. For a valid solution σ, the cost of σ, $w(\sigma)$, is the total distance moved by all servers, i.e., $w(\sigma) = \sum_{s \in \mathcal{S}} w(\Gamma(s))$. The optimal solution for the k-FCFSRP problem is a valid solution with the minimum cost.

The k-FCFSRP and its variants have applications in page replacement problems in operating systems [100], mathematics and statistics [44, 100], logistics [40] to name a few. The k-FCFSRP can be carefully modeled as a minimum-cost flow problem [37], where each edge has a unit capacity. Every unit of flow corresponds to a path taken by a server. In this flow network, every request is represented by two nodes connected by an edge of weight $-\infty$. This forces any minimum-cost flow to visit every request. The optimal solution to this $(2n + k + 2)$ node flow network can be found in $\tilde{O}(kn^2)$ time.

Next, we describe the taxi allocation problem (TAP), which is a generalization of the k-FCFSRP.

Taxi Allocation Problem (TAP). We extend the k-FCFSRP by incorporating logistical constraints that real-world taxi service agencies are likely to have, leading to the formulation of the taxi allocation problem (TAP). For this problem, we use the terms taxi and server interchangeably. To describe the TAP, we introduce some additional constraints and terminologies and modify the definitions of some existing terminologies of the k-FCFSRP problem description to model it as the TAP. All the other definitions remain the same as described in k-FCFSRP. We highlight these enhancements as follows.

A request $r \in R$ is associated with a pickup time $\textsc{pickupTime}(r)$, pickup location $\textsc{src}(r)$, and

dropoff location $\textsc{dest}(r)$. We assume the set R of requests is given as an ordered sequence $\langle r_1, r_2, \ldots, r_n \rangle$ ordered in a non-decreasing order of their pickup times. We are also given a fixed velocity $\vee$ as input that indicates the velocity of the given taxis. For each taxi s, we define its current location, denoted as $\textsc{curr}(s)$. Initially, the current location of each taxi is its initial location, i.e., for the ith taxi s, $\textsc{curr}(s)$ is initially set to s_i. A taxi s *serves* a request r by,

- moving from its current location, $\textsc{curr}(s)$, to the pickup location of r ($\textsc{src}(r)$), and,
- updating the current location of s to the dropoff location of r, i.e., by assigning $\textsc{curr}(s) \leftarrow \textsc{dest}(r)$.

For any two requests r_i and r_j where $i < j$, we say the pair of requests (r_i, r_j) is *admissible* if a taxi, traveling at the given velocity, can start moving from the dropoff location of r_i at the pickup time of r_i and reach the pickup location of r_j no later than the pickup time of r_j. More specifically, a pair of requests (r_i, r_j) is admissible if,

$$\textsc{pickupTime}(r_j) - \textsc{pickupTime}(r_i) \geq \frac{\mathsf{d}(\textsc{dest}(r_i), \textsc{src}(r_j))}{\vee}.$$

A solution to the TAP is valid only if for every routing path Γ in the solution, suppose $r_i, r_j \in \Gamma$ such that $i < j$, then the pair (r_i, r_j) is admissible.

The k-FCFSRP is a special case of the TAP. A TAP instance is identical to the k-FCFSRP when the pickup and dropoff locations are co-located for each request, and the given taxis have infinite velocity, making the pair (r_i, r_j) admissible for every i and j where $i < j$.

We show that the optimal solution to the k-FCFSRP and TAP can be computed by solving an instance of the minimum-cost bipartite matching problem. This reduction can be used

to obtain scalable solutions to the k-FCFSRP by using existing scalable exact, approximate, or parallel algorithms for the minimum-cost matching problem.

Using this reduction, we introduce two divide-and-conquer strategies to solve the k-FCFSRP. First, we present a new time-based divide-and-conquer strategy that computes the optimal solution to the k-FCFSRP and the TAP in $\tilde{O}(kn^2)$ time. The asymptotic running time of this algorithm is comparable to existing methods [38]. In two-dimensional space, one can use a dynamic weighted nearest neighbor data structure to improve the execution time of this algorithm to $\tilde{O}(kn)$. Second, in two-dimensional Euclidean space, we show that we can solve the k-FCFSRP using a geometric divide and conquer algorithm introduced by Gattani, Raghvendra, and Shirzadian [55] (GRS-algorithm) and we show its running time to be $\tilde{O}(n^{1.8})$. The GRS algorithm achieves sub-quadratic complexity for the Euclidean bipartite matching for points generated from a fixed but unknown distribution and when the given bipartite graph is complete. However, in the k-FCFSRP, points are not generated from a fixed distribution, and the bipartite graph representing the k-FCFSRP is not a complete bipartite graph. We still show that it has a sub-quadratic complexity. The GRS algorithm can also be extended straightforwardly to solve the taxi allocation problem in a two-dimensional space where the cost of moving from one location to another is lower bounded by the Euclidean cost. In such a setting, we show that the empirical performance of the GRS algorithm significantly improves over state-of-the-art algorithms for the taxi allocation problem.

In some real-world applications, such as ride-hailing services or serving calls to the police department, a server (such as a taxi or a police officer) can serve a limited number of requests; in such cases, $k = \Theta(n)$ and the GRS algorithm would be preferable. On the other hand, in some applications, such as the paging problem, the number of servers (such as the number of cache slots) can be much lower than the number of requests and $k \ll n$. In these scenarios,

our time-based divide and conquer algorithm performs faster.

Chapter 2

Background and Literature Review

In this chapter, we present the background on bipartite matching. We also discuss the prior work related to the problems under consideration in this thesis. We start by describing the bipartite matching in a general bipartite graph.

2.1 Basics of Bipartite Matching

In this section, we describe the preliminary definitions related to matching in a bipartite graph. A *matching* is a set of vertex disjoint edges in a graph. Let $G(V, E)$ be a graph and let M be a matching in G. A vertex in V is called a matched vertex if it is an endpoint of some edge in M. Otherwise, it is called a free vertex. If all the vertices of a graph are matched, the the matching is called *perfect*. A path P(resp. cycle) in G is called an alternate path(resp. cycle) with respect to matching M if the edges in P alternate between edges in M and edges not in M. An augmenting path is an alternating path that begins with and ends at a free vertex. For any given augmenting path P with respect to M, we can augment M by removing all edges in $M \cup P$ from M and adding all edges in $P - M$ to M. More formally, we can obtain a new matching M' by finding the symmetrical difference between M and P, i.e., $M' = M \bigoplus P$. After applying this operation, we get the matching M with $\|M\| = \|M\| + 1$. This process is called augmenting the matching M along P(defined first in [54]). The difference in the cost of matching before and after augmentation, i.e.

the quantity $(\mathcal{c}(M') - \mathcal{c}(M))$ is called the *net-cost* of the augmenting path along which the augmentation was performed. In a given matching M in G, we define a directed graph called the *residual graph* G_M with respect to matching M which has a vertex and edge set identical to G except for the edges in G_M are directed as follows: For any edge $(a, b) \in A \times B$, if $(a, b) \notin M$ it is directed from a to b otherwise the direction of the edge is b to a. Any directed path in the residual graph is an alternating path with respect to the underlying matching.

Primal-dual approach. The *primal-dual* approach is a commonly used technique for matching problems where there is a weight $y(v)$ associated with each vertex v of G_M which is called the dual weight of v. These weights then define a feasibility condition for the matching M in G. The matching M is feasible if the feasibility condition is met. A feasible perfect matching can be shown to be a minimum weight perfect matching. The Hungarian algorithm [69] is one of the earliest known algorithms to use the primal-dual approach for computing minimum weight bipartite matching. For any edge (a, b) in the given graph, the feasibility condition for the Hungarian algorithm is as follows:

$$y(a) + y(b) \leq d(a, b).$$
$$y(a) + y(b) = d(a, b) \quad \text{if } (a, b) \in M.$$

For any given set of dual weights, a *slack* $s(a, b)$is defined for each edge (a, b) as $s(a, b) = d(a, b) - y(a) - y(b)$. An edge is called an *admissible* edge if it has 0 slack. The Hungarian algorithm computes a minimum cost matching by iteratively finding an augmenting path P containing admissible edges and augmenting along it. Augmenting along this path will not violate the feasibility of the existing matching. The dual weights are updated between each iteration to make sure more admissible paths are found if they exist while maintaining feasibility. We have a perfect feasible matching with minimum cost when no free vertices

exist.

2.2 Prior Work

This section will discuss the state of the art for the EBM problem, k-server problem, and the taxi allocation problem.

Euclidean Bipartite Matching Problem (EBM).

A classical method for computing a maximum cardinality matching is to reduce the problem into a maximum-flow problem, which can be solved in $O(mn)$ time using the well-known Ford Fulkerson (FF) algorithm [51] (1956). The FF algorithm iteratively finds augmenting paths in the graph to increase the cardinality of matching in each iteration. This forms the basis for many advanced algorithms that developed over the years. The Hopcroft-Karp (HK) algorithm [60](1973) runs in $O(m\sqrt{n})$ time, and it is an improvement on FF-algorithm for computing a maximum cardinality matching. The HK algorithm achieves this speed up by finding multiple vertex disjoint augmenting paths per iteration and reducing the number of iterations from $O(n)$ to $O(\sqrt{n})$. For the problem of computing a minimum weight maximum cardinality matching in a weighted graph, the Hungarian algorithm designed by Kuhn and Munkres in 1956 [69] gave a primal-dual approach that has a similar structure as the FF algorithm in a weighted setting. In the Hungarian algorithm, each iteration finds a minimum cost augmenting path in $O(m + nlogn)$ time using Dijkstra's algorithm, so total time is $O(mn + n^2logn)$. The running time of the Hungarian algorithm was improved by Gabow and Tarjan (GT) in 1989 [53] for graphs with non-negative integer weights upper bounded by C. The GT Algorithm used a bit scaling approach where the algorithm of each scale is

similar to HK algorithm in a weighted graph setting. The GT algorithm can be seen as a generalization of the HK algorithm because if all the edge weights are 1, it works identically to the HK algorithm. The HK algorithm achieves a faster running time than the FF algorithm by computing multiple vertex disjoint shortest paths in each iteration. Similarly, one scale of the GT algorithm computes more augmenting paths in each phase than one iteration of the Hungarian algorithm. Both the Hungarian Algorithm and GT algorithm iteratively find augmenting paths with minimum net-cost and augment the matching along them. Thus, after each iteration, it is maintained that the increase in the matching cost is minimized. Alternate approaches such as the electrical flow [104] method and the matrix multiplication based methods [82] can be used to obtain fast matching algorithms. The current best-known execution time is $\tilde{O}(m + n^{1.5})$.

When $A \cup B$ is a 2-dimensional point set in the Euclidean space, a minimum weights Euclidean bipartite matching can be computed in $O(n^2 \operatorname{polylog} n)$ time [6, 8, 65], and in $O(n^{3/2} \operatorname{polylog} n)$ time when the points have integer coordinates [96, 97]. A widely used approach for computing approximate matching is approximating the distances between points by dividing the set of points using a hierarchical structure like a quadtree. When we want to compute an approximate matching in expectation, randomly shifted quadtrees can be used to approximate the distance between points. When the given points have integer coordinates, it is easy to compute a $O(\log n)$-approximate matching in expectation using a randomly shifted quad-tree [33, 50]. Agarwal and Varadarajan [2] build on this method and use a shifted tree structure similar to quad-tree to compute an $O(\log 1/\delta)$-approximate solution in $O(n^{1+\delta})$ time. Following this, several results were obtained that used such a decomposition. For instance, Indyk [62] approximated the minimum weight EBM by a constant factor without calculating the underlying matching in $O(n\text{poly}\log n)$ time. See [18, 52] for more related results that used this decomposition.

Recently, Raghvendra and Agarwal [89] presented an approximation algorithm for computing EBM, which computes an ε-approximate matching with high probability in $n(\varepsilon^{-1}\log n)^{O(d)}$ time. In their algorithm, each cell $\square$ of a randomly shifted quad-tree Q is decomposed by a uniform grid into $(\log n/\varepsilon)^{O(d)}$ subcells. The Euclidean distance between any pair of points u, v with $\square$ as their least common ancestor in Q is ε-approximated by the distance between the subcells of $\square$ that contain u and v respectively. Their algorithm uses Q to compute a minimum net-cost augmenting path P with respect to the new distance and augment the matching along this path, both in time $O(|P|\text{poly}\log n)$. They obtain a near-linear execution time by bounding the total length of all augmenting paths by $O(\varepsilon^{-1}n\log n)$. To compute these paths quickly, they compress the residual graph inside $\square$ into a graph of $(\log n/\varepsilon)^{O(d)}$ size and execute the Bellman-Ford algorithm on this graph.

Lahn and Raghvendra [73] extended this framework to approximate the 2-Wasserstein distance of planar point sets, i.e., an approximate minimum-cost matching when $\mathsf{d}(u, v)$ is $\|u - v\|_2^2$. Unlike Euclidean distance, approximating the squared-Euclidean distance using Q results in a polynomial-sized compressed residual graph at each cell. Since using the Bellman-Ford algorithm on such a compressed graph can be prohibitively expensive, they introduce a novel primal-dual framework and define *compressed feasibility* on the compressed residual graph. Using this framework, they are able to find an augmenting path and augment it along this path in sub-linear time. Consequently, they achieve an $O(n^{5/4}\text{poly}(\log n, 1/\varepsilon))$ time algorithm for the 2-Wasserstein distance between planar point sets. Recently, Agarwal *et al.* [9] have designed a deterministic algorithm that uses multiple quadtrees to compute a $(1+\varepsilon)$-approximate Euclidean matching in $n(\varepsilon^{-1}\log n)^{O(d)}$ time.

The time complexity to compute an ε-approximate matching by Raghvendra and Agarwal [89] is recently improved to $O(n(\varepsilon^{-O(d^3)}\log\log n + \varepsilon^{-O(d)}\log^4 n\log^5\log n))$ [12] by using a randomly-shifted tree, with a polynomial branching factor and $O(\log\log n)$ height, to de-

fine a tree-based distance function that ε-approximates the L_p metric as well as to compute the matching hierarchically. They apply the primal-dual framework on a compressed representation of the residual graph to obtain an efficient implementation of the Hungarian-search and augment operations.

In fixed dimensional settings, i.e., $d = O(1)$, there is extensive work on the design of near-linear time $(1+\varepsilon)$-relative approximation for EBM problem. The execution time of these algorithms are $\Omega(n(d\varepsilon^{-1}\log n)^d)$ ([10, 52, 66, 90]). A recent algorithm presented by [11] improved the dependence on d slightly and achieved an execution time of $n(d\varepsilon^{-1}\log\log n)^d$. Nonetheless, the exponential dependence on d make it unsuitable for higher dimensions.

For higher dimensions, a quad-tree based greedy algorithm provides an $O(d\log n)$ approximation in $O(nd)$ time. [4] combined the hierarchical greedy paradigm with an exact solver to get an $O(d^2\log 1/\varepsilon)$-approximation in $\tilde{O}(n^{1+\varepsilon})$ time. [61] combined the hierarchical greedy framework with importance sampling to estimate the cost of the EBM within a constant factor of the optimal. Additionally, there are $\tilde{O}(nd)$ time algorithms that approximate the matching cost with a factor of $O(\log^2 n)$ ([19]).

There are also exact and approximation algorithms that run in $\tilde{O}(n^2)$ time; however, these algorithms rely on several black-box reductions and at present, there are no usable implementations of these algorithms ([3, 99]). The lack of fast exact and relative approximations that are also implementable have motivated machine learning researchers to design additive approximation algorithms which we discuss next.

When it comes to δ-close Euclidean matching, the best-known algorithms are the algorithms that are designed for computing 1-Wasserstein distance, also called Earth mover's distance(EMD) between two distributions. Algorithms designed for computing EMD are valid algorithms for solving EBM problem since EBM between two sets of points is a special

case of EMD between two distributions. When the distributions are discrete and demand or supply at every node is 1, the problem becomes the bipartite matching problem. There are a number of algorithms that produce a δ-close solution such as the algorithm presented by Altschuler *et al.* [15] that runs in $\tilde{O}(n^2(C/\delta)^3)$ time, Dvurechensky *et al.* [48] that runs in $\tilde{O}(\min(n^{9/4}\sqrt{C}/\delta, n^2C/\delta^2))$, Lin *et al.* [76] that runs in $\tilde{O}(\min(n^2C\sqrt{\gamma}/\delta, n^2(C/\delta)^2))$, Quanrud [87] and Blanchet *et al.* [27] both with running time $\tilde{O}(n^2C/\delta)$. All of these algorithms have a factor of $\log n$ in their execution time and, hence, are too slow to be used in practical implementation. For faster execution in practice, there are results that use Sinkhorn projection technique for instance by Cuturi [42], Altschuler *et al.* [16], and, Dvurechensky *et al.* [48] which are faster and practical to implement. One can also adapt graph theoretic approaches including the algorithm by [53] to obtain an ε-close solution in $O(n^2\sqrt{d}/\varepsilon + nd/\varepsilon^2)$ time for points within the unit hypercube ([74]). Some of the additive approximation methods, including the Sinkhorn method, are also highly parallelizable. For instance, the algorithm by [63] has a parallel depth of $\tilde{O}(1/\varepsilon)$; see also [14, 17, 28, 47, 57, 86]

These implementations, however, only perform well for larger values of δ as they either have a significantly high running time or numerical instabilities when δ is set too small. Lahn et al. [74] addressed this by introducing an algorithm that computes a δ-close 1-Wasserstein distance in $O(n^2C/\delta + nC^2/\delta^2)$ time. Along with these theoretical bounds, they provided an implementation and showed that the empirical results of their algorithm is better than the worst case bounds.

k-server problem.

The k-server problem is central to the theory of online algorithms. For a survey of the problem, see [67]. The problem was first posted by Manasse *et al.* [80] who established a

lower bound that as long as a metric space has $k+1$ points, no deterministic algorithm can achieve a competitive ratio better than k. They also showed the competitive ratio of 2 for the 2-server problem. With this as evidence, they conjectured that in fact there is a k-competitive algorithm for this problem for any metric space. This conjecture is the celebrated *k-server conjecture*. Since then, the k-server conjecture has also been shown to be true for the line metric (1-dimensional Euclidean space) [37] and the tree metric [35]. It was shown that the Work Function Algorithm achieves a competitive ratio of $2k-1$ on any metric space [68]. There has not been any significant progress on this conjecture since then. On the other hand, there has been substantial work on the randomized version of the k-server conjecture; see, for instance, Bubeck et al. [30] and Lee [75].

The analysis of the WFA in [68] was based on an exponential time dynamic programming implementation which processes the ith request r_i by solving equation equation 1.2.

The problem of finding the offline optimal solution for the k-server problem can be carefully modeled as a minimum-cost flow problem [37], where each edge has a unit capacity. Every unit of flow corresponds to a path taken by a server. In this flow network, every request is represented by two nodes connected by an edge of weight $-\infty$. This forces any minimum-cost flow to visit every request. The optimal solution to this flow network of $2n+k+2$ nodes can be found in $O(n^2k)$ time.

For the online case, processing the ith request r_i requires solving equation equation 1.2. Similar to the offline case, evaluating equation 1.2 explicitly can be modelled as computing k distinct minimum-cost flow values, each of which can take $\Theta((i+k)^2k)$ time [37]. This observation leads to an $O((i+k)^2k^2)$ time algorithm for the WFA [92]. Using clever observations, evaluation of equation 1.2 can be reduced to computation of a single minimum-cost flow which takes $O(k(i+k)^2)$ time [93]. Rudec and Manger [91] presented an alternative approach for computing an offline solution to the k-server problem. Instead of creating a

flow network with edges of $-\infty$ cost, they define a graph in the original metric space and define the notion of *regular flow* to be any flow in which each request is served by at least one server. One can move from any regular flow to another one using the so-called *up-down cycles*. Upon adding a new request, they show that finding the minimum-cost regular flow can be done by finding the most negative up-down cycle, which they accomplish by conducting an exhaustive search. They argue that there is an empirical benefit to this approach despite the worst-case execution time of this algorithm is slower than that of [93]. They also extend this approach to the work function algorithm.

Taxi allocation problem.

Typically, when the requirement is to deliver goods from one location to another, the routing problem is called a delivery problem. When the vehicle moves people, the routing problem is referred to as *dial-a ride* in [24]. The dial-a-ride problem also allows car-pooling or taxi sharing. In our discussion, the problem in context, i.e. the taxi allocation problem is a special case of dial-a-ride where any taxi has to complete the service of a customer before starting to serve the next customer. It is easy to see that a good practical solution to the taxi allocation problem should have minimum distance routes that incur reasonable wait times for the requests. Variations to the Vehicle routing problem (VRP) indirectly address this. For instance, VRP with Time Windows (VRP-TW) [1] is the same as Vehicle Routing with an added restriction that a customer can only be served within a given time window associated with the customer. The VRP-TW is typically formulated as an Integer Programming (IP) problem. There are diverse applications of routing problems where the requests are associated with time windows such as [21, 34, 43]. The formulations typically used in these problems generate an underlying graph with requests as nodes and a directed edge from nodes u to v if a vehicle can serve v after u. When the graph size is large, the

integer programming algorithms typically run very slowly. In [26], the authors presented ways to improve running time by deleting the graph edges that are not likely to be in the optimal solution. In some problem statements, the time windows can be relaxed to obtain a solution with a better cost that does not strictly obey the time window constraint. For example, in [59], the algorithm adds some penalty in the output for violating the time window constraint. Other than this, there have been results that focus on binding requests to one another from a group and enforcing that the requests in a group will be served by the same taxi. [79, 83, 84, 95] implements such a strategy when taxi-pooling is allowed, i.e., a taxi can carry multiple customers at the same time. These approaches are used for large-scale applications since binding requests in a group reduces the need to choose the best taxi for each request. This approach, however, focuses mainly on how well the requests are grouped together and not much on the overall quality of routes generated for taxis.

The design of an algorithm for any routing problem depends on the objective and the known information. In the offline case, the known information is the entire request sequence. In contrast, in the online scenario, at any time during the algorithm's execution, we only know the past requests that have already been processed. Many online algorithms, for instance, work function algorithm, make use of the known requests by maintaining an offline solution for the requests that have been revealed, and on the arrival of each new request r, the offline maintained solution is updated to incorporate r, and this maintained offline solution also plays a part in deciding the server it picks for serving r. A similar strategy can also be used with integer programming optimization, where an offline solution is maintained by repeatedly finding a solution with available requests. This is known as rolling-horizon optimization. Rolling-horizon optimization is popular in online algorithms, and [59] proved that the better the cost of the offline solution maintained after each new request, the better the quality of the online solution. [24] and [107] are examples of results that utilize the rolling-

horizon optimization for routing problems. In addition, numerous online routing algorithms predict future requests based on a combination of the specific application in context and the real-life data available. For instance, [22, 23] analyze dynamic VRP-TW with stochastic requites and makes use of the predictions of the future request locations along with rolling-horizon. [81] also included methods to improve long-term benefits based on predicted future requests to solve the Pickup and Delivery Problem such as courier delivery service. A problem description similar to the taxi allocation problem is discussed in [108], which uses network flow mixed integer formulation and aims at load balancing of delivery trucks and cost optimization. Such formulation considers every combination of feasible routes and hence very slow when the input size is as large as the number of Uber taxi requests per day. [26] made the underlying graph sparse and used a commercial solver to achieve improvement on running time for large scale applications.

Chapter 3

An Improved ε-Approximation Algorithm for EBM

In this chapter, we will describe our first result, which is a faster computation of an ε-approximate Euclidean bipartite matching. Our algorithm runs in algorithm which runs in $O(n(\varepsilon^{-O(d^3)} \log\log n + \varepsilon^{-O(d)} \log^4 n \log^5 \log n))$ time. The best-known algorithm prior to this result has a running time of $n(\varepsilon^{-1} \log n)^{\Omega(d)}$. Note that in this time complexity, d appears in the exponent of the running time. We remove the dimension dependency on the $\log n$ term in our time complexity. The following theorem states our main result.

Theorem 3.1. *Let A, B be two point sets in $\mathbb{R}^d$ of size n each, for a constant $d > 1$, and let $0 < \varepsilon \leq 1$ be a parameter. With probability at least $1/2$, an ε-approximate matching under any L_p-metric can be computed in $O(n(\varepsilon^{-O(d^3)} \log\log n + \varepsilon^{-O(d)} \log^4 n \log^5 \log n))$ time.*

For the sake of simplicity, we describe the algorithm for the Euclidean metric. It can be extended to other L_p-metrics in a straight forward manner. For any two points a and b, we use $\|a - b\|$ to denote the Euclidean distance between them. Using standard techniques [73, 89], we can preprocess the input points in $O(n \log n)$ time so that the point sets A and B satisfy the following conditions: (P1) All input points have integer coordinates bounded by $n^{O(1)}$. (P2) No integer grid point contains points of both A and B. (P3) $\mathfrak{c}(M^*) \in \left[\frac{3\sqrt{d}n}{\varepsilon}, \frac{9\sqrt{d}n}{\varepsilon}\right]$. Details of how we preprocess A and B is given in Section 3.1.

Assuming A and B satisfy (P1)–(P3), we present an algorithm that, with probability 1/2, computes an $(\varepsilon/2)$-approximate matching in $O(n(\varepsilon^{-O(d^3)}+\varepsilon^{-O(d)}\log^4 n \log^4 \log n))$ time. The preprocessing step adds an additional $\log\log n$ factor to the running time of the algorithm, resulting in the running time mentioned in Theorem 3.1. In the following, we provide an overview of our approach and its comparison with existing work.

As in [73, 89], we also define a tree based distance $\mathsf{d}_{\mathsf{T}}(\cdot,\cdot)$ that approximates the Euclidean distance (Section 3.2). Unlike [73, 89] that use a quad-tree of height $O(\log n)$, we build a tree T of height $O(\log\log n)$ (see Section 3.2.1). Each cell of T at level i (root is assigned level 0) with a side length of ℓ_i is partitioned using a randomly-shifted grid into children whose side-length $\ell_{i+1} = \ell_i^c$ where $c < 1$ is a constant that depends only on d. Given that the point set have integer coordinates bounded by $n^{O(1)}$ (from (P1)), the height of T is $h = O(\log\log n)$. For any pair of points (u, v) with a cell $\square$ of level i as its least common ancestor, let $\square_u$ and $\square_v$ be the children of $\square$ that contain u and v respectively. As in the case of a randomly shifted-quadtree where we get an $O(h) = O(\log n)$ approximation, one can show that the distance between the centers of $\square_u$ and $\square_v$ is a $O(h) = O(\log\log n)$ approximation of the Euclidean distance (in expectation). We obtain a refined $(1+\varepsilon)$-approximation of the Euclidean distance by partitioning $\square_u$ and $\square_v$ into finer subcells and then using the distance between the centers of those sub-cells that contain u and v. As in [73, 89], one can divide each cell into $O(h^d)$ many subcells and obtain a $(1+\varepsilon)$-approximation of the Euclidean distance. With $h = \log\log n$, this will result in an execution time of $\Omega(n\log^4 n(\varepsilon^{-1}\log\log n)^{d^3})$. Instead, we partition a cell into subcells more carefully (See the definition of subcells in Section 3.2.2). Intuitively, we make the number of subcells a function of the height of the cell, i.e., smaller cells have significantly fewer than $\log^{O(d)}\log n$ subcells. As a result, we are able to improve the dependence of our algorithm from $\log^{O(d^3)}\log n$ to $\log^{O(d)}\log n$. Interestingly, we show that the expected distortion is higher for cells that are

closer to the leaves. Nonetheless, we are able to show a bound $\varepsilon\|u-v\|$ on the expected error of the distance from a node u to a node v (See Lemma 3.3).

Similar to [73], our algorithm compactly stores the residual graph (Section 3.5.2) as well as the dual weights (Section 3.5.3) and uses this compact representation to efficiently find augmenting paths. The size of the compressed residual graph inside any cell is bounded by the side-length of its child, i.e., smaller cells have a smaller compressed graph (Lemma 3.15). As a result, finding augmenting paths in smaller cells is significantly faster than that in larger ones. In our analysis, we show that most of the augmenting paths in the algorithm are found in smaller cells which can be computed quickly. In particular, only $O(\frac{n}{\varepsilon\ell_{i+1}})$ augmenting paths are found inside a compressed graph at level i, each of which can be found in $O(\ell_{i+1}\log^2 n)$ time. Combining across all $O(\log\log n)$ levels, we get a near-linear execution time.

Typical matching algorithms that are based on a compressed residual graph modify the dual weights and find an augmenting path with respect to current matching M. The algorithms presented in [73, 89] classify edges into *local* and *non-local* which they use critically in computing a minimum net-cost augmenting path. We remove the need for this classification and make our algorithm and its analysis simpler. Instead of using the classification, our algorithm carefully updates the dual weights, possibly modifies a matching M to another matching M' of the same size and cost, and finds an augmenting path with respect to the new matching M'.

3.1 Preprocessing Step

Similar to some of the earlier algorithms [73, 89], we perform the following preprocessing step that makes the input "well-conditioned" at a slight increase in the cost of the optimal matching. Using a quad-tree based greedy algorithm [33], we compute a $c_1 \log n$-approximate

matching M_0 of A and B, in $O(n \log n)$ time, for some constant $c_1 \geq 0$ [50]. Let $w_0 = \mathfrak{c}(M_0)$. Then $\frac{w_0}{c_1 \log n} \leq \mathfrak{c}(M^*) \leq w_0$.

For any integer $i \in [0, \log(c_1 \log n)]$, define $\beta_i = w_0/2^i$; there is an i such that $\beta_i \leq \mathfrak{c}(M^*) \leq \beta_{i-1}$. Set $t(n) = c_0 n(\varepsilon^{-O(d^3)} + \log^3 n(\varepsilon^{-1} \log\log n)^{O(d)} + \varepsilon^{-O(d)} \log^4 n \log\log^4 n)$ for some sufficiently large constant c_0. We run the algorithm described in Section 3.4 for at most $t(n)$ steps on each choice of β_i. In the i-th iteration, either the algorithm returns a perfect matching of A and B or terminates without computing a perfect matching. Among the perfect matchings computed by the algorithm, we return the one with the smallest cost. Theorem 3.11 ensures that if $\beta_i \leq \mathfrak{c}(M^*) \leq \beta_{i-1}$ then with probability at least $1/2$, the algorithm returns an ε-approximate matching within $t(n)$ time. Now forward, we assume that we have computed a value $\beta > 0$ such that $\mathfrak{c}(M^*) \leq \beta \leq 2\mathfrak{c}(M^*)$. As in [73, 89], by scaling $A \cup B$ and snapping points to an integer grid, we can assume A and B satisfy the following conditions: (P1) All input points have integer coordinates bounded by $n^{O(1)}$. (P2) No integer grid point contains points of both A and B. (P3) $\mathfrak{c}(M^*) \in \left[\frac{3\sqrt{d}n}{\varepsilon}, \frac{9\sqrt{d}n}{\varepsilon}\right]$. We compute an $(\varepsilon/2)$-approximate matching of A and B satisfying $(P1) - (P3)$ in $t(n)$ time.

3.2 Hierarchical Partition and the Distance Function

In this section, we present a randomized hierarchical partitioning of space, used to define a new distance function $\mathsf{d}_\mathsf{T}(\cdot, \cdot)$ that approximates the Euclidean metric (in expected sense) as well as to guide the construction of matching (in a hierarchical manner).

3.2.1 Hierarchical Partitioning

For a value $\ell > 0$, let $\mathbb{G}[\ell]$ be the d-dimensional uniform grid with cell side-length ℓ, i.e., $\mathbb{G}[\ell] = (\ell\mathbb{Z})^d + [0, \ell]^d$. For a point $x \in \mathbb{R}^d$, we use $\mathbb{G}[\ell] + x$ to denote the translate of $\mathbb{G}[\ell]$ by x. For any rectangle R, let $A_{\mathsf{R}} = A \cap \mathsf{R}$ and $B_{\mathsf{R}} = B \cap \mathsf{R}$. We say that R is *non-empty* if $A_{\mathsf{R}} \cup B_{\mathsf{R}} \neq \emptyset$. Given R and a grid $\mathbb{G}$, let $\mathsf{C}[\mathsf{R}, \mathbb{G}]$ denote the set of non-empty rectangles in the rectangular subdivision of R induced by $\mathbb{G}$. If $\mathbb{G}$ is fixed or clear from the context, we use $\mathsf{C}[\mathsf{R}]$ to denote $\mathsf{C}[\mathsf{R}, \mathbb{G}]$. By abusing the notation slightly, we use $\mathsf{C}[\mathsf{R}]$ to denote the subdivision as well as the set of non-empty rectangles in the subdivision. For a non-empty rectangle R, we designate one of the points in $A_{\mathsf{R}} \cup B_{\mathsf{R}}$, say r_{R}, as its *representative*.

Let $\square_0$ be the smallest axis-aligned hypercube that contains $A \cup B$. Let ℓ_0 be its side length. By property (P1), $\ell_0 = n^{O(1)}$. We construct a hierarchical partition and the associated tree T, as follows. Each node in T is associated with a non-empty rectangle which we refer to as a *cell* and we will not distinguish between the two. The *level* of a node (and the corresponding rectangle) is the length of the path in T from the root. The root of T is $\square_0$ and its level is 0. Set $\alpha = 1 - \frac{1}{8d+2}$. For $i > 0$, set $\ell_i = {\ell_{i-1}}^{\alpha}$. Let $h > 0$ be the smallest integer such that $\ell_h \leq (\varepsilon^{-1} d)^{\frac{\alpha}{(1-\alpha)^2}}$. Any cell $\square$ in T of level h is designated as a leaf node. By construction, $h = O(\log \log n)$ The choice for the condition of the leaf node will become apparent in Section 3.2.2. Otherwise, we choose a random point, $\xi_{\square} \in [0, \ell_{i+1}]^d$ and set $\mathbb{G} = \mathbb{G}[\ell_{i+1}] + \xi_{\square}$. Let $\mathsf{C}[\square] := \mathsf{C}[\square, \mathbb{G}]$ be the subdivision of $\square$ induced by $\mathbb{G}$. Each rectangle $\square' \in \mathsf{C}[\square]$ is a level $i + 1$ node. We create a child of $\square$ in T for each non-empty rectangle $\square' \in \mathsf{C}[\square]$ and recursively construct the partition and associated sub-tree of $\square'$. For any $0 \leq i \leq h - 1$, let $\Delta[i]$ denote all cells of T of level i.

3.2.2 Euclidean Distance Approximation

For any pair of points $(a, b) \in A \times B$, let $\square$ be the least common ancestor of a and b in T, i.e., the cell with the highest level that contains a and b. Let the level of $\square$ be i. We define the *level* of (a, b), $\text{lev}(a, b)$, to be i and refer to (a, b) as *a level i edge*. Let $\square_a$ and $\square_b$ be the children of $\square$ that contain a and b respectively. We divide $\square_a$ and $\square_b$ into $O(\varepsilon^{-d}(h-i)^{2d})$ *subcells* and show that the distance between any two points in the subcells that contain a and b is a $(1+\varepsilon)$-approximation of $\|a-b\|$. Note that the smaller cells, i.e., those at a higher level, have fewer subcells. Using this and the bound on the side-length of any leaf node of T, we can bound the number of subcells of the children of $\square$ by ℓ_{i+1} (independent of $\log\log n$). In Section 3.5, we use this fact to compress the residual graph more efficiently.

Subcells: Any cell $\square$ is divided into subcells as follows. For each $0 < i < h$, set

$$\mu_i = \frac{\varepsilon}{c_2\sqrt{d}(h-i)^2}\ell_i, \tag{3.1}$$

where $c_2 = 24\pi^2$. We set $\mu_0 = \infty$ and $\mu_h = 0$. A *subcell* is formed by combining a subset of children of $\square$ such that the diameter of points in these children is no more than μ_i. Set $\bar{\mu_i} = \left\lfloor \frac{\mu_i}{\sqrt{d}\ell_{i+1}} \right\rfloor \ell_{i+1}$ (By Lemma 3.2 below, $\left\lfloor \frac{\mu_i}{\sqrt{d}\ell_{i+1}} \right\rfloor$ is a positive integer). For any non-leaf and non-root cell of level i in the tree T, let $\mathbb{G}_\square = \mathbb{G}[\bar{\mu_i}] + \xi_\square$, where $\xi_\square$ is the random shift for the grid constructed in $\square$. By construction, the boundary of grid cells in $\mathbb{G}_\square$ are aligned with those of $\mathbb{G}[\ell_{i+1}] + \xi_\square$ (See Figure 3.1). Therefore, each non-empty child cell of $\square$ lies in exactly one subcell of subdivision $\mathsf{C}[\square, \mathbb{G}_\square]$. Let $\mathsf{S}_\square$ be this set of non-empty subcells, and let $\mathsf{R}_\square$ be the set of representative points of $\mathsf{S}_\square$, i.e., $\mathsf{R}_\square = \{r_\mathsf{R} \mid \mathsf{R} \in \mathsf{S}_\square\}$. It is clear that the diameter of each subcell is at most μ_i. The following lemma relates the side-length of the children of a cell to the diameter of the subcells of that cell.

Lemma 3.2. *For any* $0 \leq i < h$, $\ell_{i+1} \leq \mu_i/\sqrt{d}$.

WSPD: For any cell $\square$, the number of pairs of children subcells of $\square$ can be prohibitively large. We use a well-separated pair decomposition (WSPD) to compactly store these pairs. For simplicity of the algorithm, similar to [73], we use WSPD to define our Euclidean distance approximation. For a point set X, let $\textsc{Diam}_X$ be the distance between the farthest pair of points in X. For any $\varepsilon > 0$, two point sets X and Y are called *ε-well-separated* if for all $x \in X$ and $y \in Y$, $\max\{\textsc{Diam}_X, \textsc{Diam}_Y\} \leq (\varepsilon/12)\|x - y\|$. Given a point set $X \in \mathbb{R}^d$ of size n and a parameter $\varepsilon > 0$, an *ε-WSPD* (or simply WSPD for brevity) of X is a set $\mathcal{W} = \{(R_1, S_1), \cdots, (R_k, S_k)\}$ such that (i) each (R_i, S_i) is ε-well separated, (ii) for every two points $(u, v) \in X \times X$, there is a distinctive pair $(R_i, S_i) \in \mathcal{W}$ such that $(u, v) \in R_i \times S_i$ or $(u, v) \in S_i \times R_i$, and (iii) $k = O(\varepsilon^{-d} n)$. Also, if the largest divided by the smallest pairwise distances, also known as the spread of X, is within $n^{O(1)}$, then every point of X participates in $O(\varepsilon^{-d} \log n)$ pairs of $\mathcal{W}$. One can construct a $\mathcal{W}$ in $O(n \log n + \varepsilon^{-d} n)$ time [32, 58]. For any $(R_i, S_i) \in \mathcal{W}$, we choose an arbitrary pair $(x_i, y_i) \in R_i \times S_i$ and make this pair its *representative pair*.

For any non-leaf cell $\square \in T$, let $X_\square = \bigcup_{\square' \in \mathsf{C}[\square]} \mathsf{R}_{\square'}$ be the set of representative points of all non-empty subcells of the children of $\square$. We build an ε-WSPD $\mathcal{W}_\square = \{(R_1, S_1), \ldots, (R_k, S_k)\}$ on $X_\square$. For a leaf cell $\square$, we construct an ε-WSPD $\mathcal{W}_\square$ on $A_\square \cup B_\square$.

Distance function: We are now ready to define the distance function $\mathsf{d}_\mathsf{T} : A \times B \mapsto \mathbb{R}_{\geq 0}$. For any pair of points $(a, b) \in A \times B$ of level i with $\square$ as its least common ancestor, if $i = h$, we set $\delta_{ab} = 0$ and if $i < h$, we set $\delta_{ab} = \mu_{i+1}$. For $i = h$, we set (R_j, S_j) to be the pair in $\mathcal{W}_\square$ such that $(a, b) \in R_j \times S_j$. For $i < h$, (R_j, S_j) is defined as follows. Let $\square_a$ (resp. $\square_b$) be the child of $\square$ that contains a (resp. b), and let ξ^a_{ab} (resp. ξ^b_{ba}) be the subcell of $\square_a$ (resp.

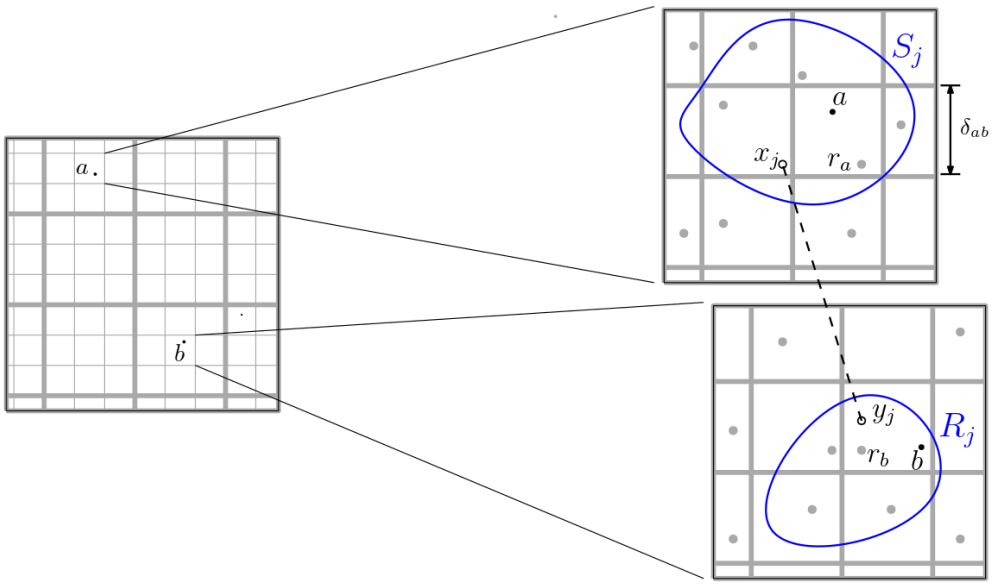


Figure 3.1: Euclidean distance approximation: The grey rectangular subdivision shows the children of $\square$ (on left) and the bold grey rectangular subdivision shows the subcells.

$\square_b$) that contains a (resp. b). Let r_a and r_b be the representatives of ξ^a_{ab} and ξ^b_{ba} respectively. Let (R_j, S_j) be the unique ε-well separated pair of $\mathcal{W}_\square$ such that $(r_a, r_b) \in R_j \times S_j$. We say that (a, b) is *covered* by (R_j, S_j). Now let (x_j, y_j) be the representative pair of (R_j, S_j) (See Figure 3.1). Then, we define

$$\mathsf{d}_\mathsf{T}(a, b) = (1 + \varepsilon/4)\|x_j - y_j\| + 2\delta_{ab}. \tag{3.2}$$

Unlike in [73, 89], we create fewer subcells for cells that are closer to the leaves, which results in a larger distortion for the edges within these cells. However, Lemmas 3.3 and 3.4 together establish that the expected distortion on any pair of points $(a, b) \in A \times B$ is still proportional to $\varepsilon\|a - b\|$.

Lemma 3.3. *For any pair of points* $(a, b) \in A \times B$, $\mathbb{E}[\delta_{ab}] \leq \frac{\pi^2}{6c_2}\varepsilon\|a - b\|$.

Proof. If the edge (a, b) intersects the boundary of a cell at level $i + 1$, then, $\mathrm{lev}(a, b) \leq i$.

Therefore, $\Pr[\text{lev}(a,b)=i] \leq \frac{\sqrt{d}\|a-b\|}{\ell_{i+1}}$. As a result,

$$\mathbb{E}[\delta_{ab}] \leq \sum_{i=0}^{h-1} \Pr[\text{lev}(a,b)=i].\mu_{i+1} \leq \sum_{i=0}^{h-1} \frac{\sqrt{d}\|a-b\|}{\ell_{i+1}}.\frac{\varepsilon\ell_{i+1}}{c_2\sqrt{d}(h-i)^2} = \frac{\varepsilon}{c_2}\|a-b\| \sum_{i=0}^{h-1} \frac{1}{(h-i)^2}.$$

$\sum_{i=1}^{\infty} \frac{1}{i^2} = \zeta(2) = \frac{\pi^2}{6}$, where $\zeta(\cdot)$ is the *Riemann zeta function*. Therefore, $\mathbb{E}[\delta_{ab}] \leq \frac{\pi^2}{6c_2}\varepsilon\|a-b\|$. □

Using Lemma 3.3, we show in Lemma 3.4 that our distance function, in expectation, approximates the Euclidean distance within a factor of $(1+\varepsilon)$.

Lemma 3.4. *For any pair of points* $(a,b) \in A \times B$, $\mathsf{d_T}(a,b) \geq \|a-b\|$. *Furthermore,* $\mathbb{E}[\mathsf{d_T}(a,b)] \leq (1+(\frac{5\pi^2}{6c_2}+\frac{11}{24})\varepsilon)\|a-b\|$.

3.3 Algorithm Preliminaries

We begin by presenting notations pertaining to the distance function that will help us describe our algorithm. For any subset $E \subseteq A \times B$ of edges, we use $w(E) = \sum_{(u,v)\in E} \mathsf{d_T}(u,v)$ to denote the sum of the weights of the edges in E with respect to $\mathsf{d_T}(\cdot,\cdot)$.

Next, we will recall the definitions related to matching that will assist us in presenting our algorithm. Let M be any matching in $\mathcal{G}$. Recall that a vertex is *free* with respect to M if it has no edges of M incident on it. An *alternating* path with respect to M is a simple path in $\mathcal{G}$ whose edges alternate between those in M and not in M. An *augmenting* path is an alternating path whose endpoints are free. We can *augment* M along an augmenting path P by simply setting $M \leftarrow M \oplus P$. For any matched vertex $u \in A \cup B$, let $m(u)$ denote the vertex to which u is matched in M. For any edge (u,v), we define the *net-cost* $\phi(u,v)$ of (u,v) as follows: $\phi(u,v) = \mathsf{d_T}(u,v) + \delta_{uv}$ if $(u,v) \notin M$, and $\phi(u,v) = -\mathsf{d_T}(u,v)$ if $(u,v) \in M$.

For a set $E \subseteq A \times B$ of edges, we define its *net-cost* as $\phi(E) = \sum_{(u,v)\in E} \phi(u,v)$.

A *residual graph* $\mathcal{G}_M$ is a directed bipartite graph that has the same set of edges as $\mathcal{G}$ and for any matching (resp. non-matching) edge $(a,b) \in A \times B$, it is directed from a to b (resp. b to a). We refer to the *weight* of (a,b) in $\mathcal{G}_M$ to be its net-cost. Any simple directed path P in $\mathcal{G}_M$ alternates between being a matching and a non-matching edge implying that P is an alternating path. For any rectangle R, let M_{R} be the subset of the edges of M with both endpoints inside R, and let $\mathcal{G}_M^{\mathsf{R}}$ denote the residual graph on $A_{\mathsf{R}} \cup B_{\mathsf{R}}$ for the matching M_{R}.

Similar to the Hungarian algorithm, our algorithm assigns a dual weight $y(v) \geq 0$ to each vertex $v \in A \cup B$. We say that a matching M along with an assignment of non-negative dual weights $y(\cdot)$ are *feasible* if, for every directed edge (u,v) of $\mathcal{G}_M$, $y(u) - y(v) \leq \phi(u,v)$. The presence of δ_{uv} in the definition of $\phi(u,v)$ makes our feasibility conditions a relaxed form of the feasibility conditions of the Hungarian algorithm. It can be shown that a feasible perfect matching is (in expectation) a $(1+\varepsilon/2)$-approximation of the minimum-cost Euclidean matching.

Lemma 3.5. *Suppose a perfect matching M along with a set of dual weights $y(\cdot)$ are feasible and let M^* denote an optimal matching with respect to the Euclidean cost $\mathfrak{c}(\cdot)$. Then, $\mathbb{E}[\mathfrak{c}(M)] \leq \mathbb{E}[w(M)] \leq (1+\varepsilon/2)\mathfrak{c}(M^*)$.*

For any directed edge (u,v) of $\mathcal{G}_M$, we define its *slack* as $s(u,v) = \phi(u,v) - y(u) + y(v)$. Based on the definition of feasibility, it is clear that $s(u,v) \geq 0$. An edge (u,v) of $\mathcal{G}_M$ is called *admissible* if $s(u,v) = 0$. Given a feasible matching M, we can use the definition of slack to relate the weight of any directed path P from u to v in $\mathcal{G}_M$ to the slack on its edges:

$$\phi(P) = \sum_{(u',v')\in P} (y(u') - y(v') + s(u',v')) = y(u) - y(v) + \sum_{(u',v')\in P} s(u',v'). \tag{3.3}$$

We present a slow yet simple implementation to find a $(1+\varepsilon)$-approximate matching, which is basically the Hungarian algorithm. Initialize for every $v \in A \cup B$, $y(v) = 0$ and $M \leftarrow \emptyset$. At each step, we find an augmenting path in the residual graph as follows. We set the edge weights in the residual graph to be their slacks. Next, starting from the free vertices of B, we execute the Dijkstra's shortest-path algorithm (also called the *Hungarian Search*) in this graph. For any $v \in A \cup B$, let κ_v be the shortest path from any free vertex of B to v. The algorithm returns the augmenting path P to a free vertex a of A that minimizes κ_a; i.e, the minimum net-cost augmenting path. We update the dual weight of every vertex v with $\kappa_v < \kappa_a$ by setting $y(v) \leftarrow y(v) - \kappa_v + \kappa_a$, making all edges of P admissible. Finally, we set $M = M \oplus P$. Augmenting the matching along P keeps the matching feasible. In the following lemma, we state two observations of this algorithm.

Lemma 3.6. *During the execution of the algorithm described above,*

(i) augmenting paths are computed in increasing order of their net-costs; and

(ii) if the net-cost of an augmenting path P is less than μ_i, then P does not contain any edge of level lower than i (Recollect that any such edge has a cost of at least μ_i).

Lemma 3.6 suggests that we can search for augmenting paths in residual graph inside the cells of $\Delta[i]$ until the net-cost of the augmenting path reaches μ_i.

The implementation described above requires n executions of Dijkstra's algorithm, each taking $\Theta(n^2)$ time. In the next two sections, we use the properties of this algorithm (Lemma 3.6) to present an efficient implementation of a variant of the above algorithm.

3.4 Overview of the Algorithm

We present our algorithm assuming the existence of three procedures, namely, Build, HungarianSearch, and Augment procedures. The details of these procedures is deferred to Section 3.5.

Our algorithm computes a feasible perfect matching by processing the cells of T in decreasing order of their levels. Initially $i \leftarrow h-1$ and $M \leftarrow \emptyset$ (no matching is computed at level h). For any cell $\square \in \Delta[i]$, the algorithm executes the following steps:

- If $i < h-1$, the algorithm calls the Build$(\square, M)$ procedure. This step builds a compact representation of the residual graph (defined in Section 3.5.2).
- The algorithm does the following iteratively: It calls the HungarianSearch$(\square, M)$ procedure. This procedure returns NULL if there is no augmenting path in $\mathcal{G}_M^{\square}$ of a net-cost less than μ_i (see equation 3.1). In this case, the algorithm stops processing $\square$. Otherwise, if there is an augmenting path, the procedure updates the dual weights $y(\cdot)$ and may update $M \leftarrow M'$ where M' is another matching of equal size such that $y(\cdot), M'$ is feasible and $w(M) = w(M')$. Then it returns a minimum net-cost augmenting path P with respect to the updated matching. The algorithm calls Augment$(\square, P, M)$ to augment M along P.

After all cells in $\Delta[i]$ are processed, if $i = 0$, the algorithm returns the matching M. Otherwise, it sets $i \leftarrow i-1$ and continues.

Execution time of the procedures: The Build, HungarianSearch, and Augment procedures presented in Section 3.5 have the following execution time. For any cell $\square$, let $n_{\square} =$

$|A_\square \cup B_\square|$. If $\square$ has a level $i < h-1$, the BUILD procedure takes $n_\square \log^3 n(\varepsilon^{-1}\log\log n)^{O(d)}$ time. Next, we present the running time of HUNGARIANSEARCH and AUGMENT procedures.

For any cell $\square$, if $\square$ is at level $h-1$ of T, then the HUNGARIANSEARCH and AUGMENT procedures takes $\varepsilon^{-O(d^3)}$ time. Otherwise, let $i = \text{lev}(\square) < h-1$. For $j > i$, let k_j be the number of level j cells that contains at least one vertex of P. If HUNGARIANSEARCH returns NULL, $k_j = 0$. HUNGARIANSEARCH takes

$$O(\mu_{i+1}\varepsilon^{-O(d)}\log^3 n \log^2\log n + \sum_{j=i+1}^{h-1} k_j\mu_{j+1}\varepsilon^{-O(d)}\log^3 n\log^3\log n)$$

time and the total time taken by AUGMENT is $O(\sum_{j=i+1}^{h-1} k_j\mu_{j+1}\varepsilon^{-O(d)}\log^3 n\log^3\log n)$.

Invariants: In Section 3.5, we also show that the three procedures maintain the following two invariants while processing cells at level i. At any point, for the matching M, there is a set of dual weights $y(\cdot)$ such that

(J1) $M, y(\cdot)$ is feasible, and,

(J2) For any vertex $u \in B$, $y(u) \le \mu_i$. Furthermore, if u is a free vertex of B, its dual weight $y(u) \ge \mu_{i+1}$. If u is a free vertex of A, its dual weight $y(u) = 0$.

Our procedures will not maintain dual weights $y(\cdot)$ explicitly but only guarantee the existence of dual weights that satisfy (J1) and (J2). From (J1), (J2), and equation 3.3, we get the following:

Corollary 3.7. *For any $i \ge 0$, while the algorithm is processing level i cells, the net-cost of any augmenting path in $\mathcal{G}_M$ is at least μ_{i+1}.*

3.4.1 Analysis of the algorithm

Note that at the root cell $\square_0$, $\mu_0 = \infty$ and therefore, the second step of the algorithm terminates only when there are no more augmenting paths in $\mathcal{G}_M$; i.e, M is prefect. Since $M, y(\cdot)$ is also feasible, from Lemma 3.5, it is (in expectation) a $(1+\varepsilon)$-approximate matching. Let W be the cost of the matching returned by our algorithm. From (P3) and the fact that $\varepsilon \leq 1$, we get that, with probability at least $1/2$,

$$W = \Theta(n/\varepsilon). \tag{3.4}$$

Next, using the execution time of the procedures and the invariants they maintain, we bound the execution time of our algorithm.

We introduce a few notation that helps us analyze our algorithm. Recollect that $n_\square = |A_\square \cup B_\square|$ and M^* denotes the optimal matching of $A \cup B$ with respect to the Euclidean costs. Let $\mathbb{P} = \langle P_1, \ldots, P_n \rangle$ be the sequence of augmenting paths computed by the algorithm in the order in which they were computed. Let $M_0 = \emptyset$ and let M_i be the matching after augmenting along the path P_i.

Efficiency analysis: We begin by bounding the time taken by $\textsc{Build}(\square, M)$ across all $O(\log \log n)$ levels of T by

$$\sum_{i=0}^{h-1} \sum_{\square \in \Delta[i]} n_\square \log^3 n_\square (\varepsilon^{-1} \log \log n)^{O(d)} = O(n \log^3 n (\varepsilon^{-1} \log \log n)^{O(d)}). \tag{3.5}$$

This equality follows from the fact that $\sum_{\square \in \Delta[i]} n_\square = n$ and $h = O(\log \log n)$. Next, we bound the execution time of HungarianSearch and Augment procedures. For any cell $\square$ of level $h-1$, the HungarianSearch and Augment procedures take $(1/\varepsilon)^{O(d^3)}$ time. The total

time taken across all level $h-1$ cells is $n/\varepsilon^{O(d^3)}$. Next, we bound the running time for cell at levels less than $h-1$.

Lemma 3.8. *For $i < h$, the number of free vertices after processing level i cells is $O\left(\frac{W}{\mu_i}\right)$.*

Therefore, there are $O(\frac{W}{\mu_{i+1}})$ executions of HungarianSearch on level i cells. The time taken by all HungarianSearch executions that return a NULL is at most

$$\left(\frac{W}{\mu_{i+1}}\right) \times O(\mu_{i+1}\varepsilon^{-O(d)} \log^3 n \log^2 \log n) = O(W\varepsilon^{-O(d)} \log^3 n \log^2 \log n).$$

Otherwise, the HungarianSearch procedure returns a minimum net-cost augmenting path P and the Augment procedure augments the matching along P. The time taken by each such execution of HungarianSearch and Augment procedure is

$$\begin{aligned} &O(\mu_{i+1}\varepsilon^{-O(d)} \log^3 n \log^2 \log n + \sum_{j=i+1}^{h} k_j \mu_{j+1}\varepsilon^{-O(d)} \log^3 n \log^3 \log n) \\ &\quad = O(\mu_{i+1}\varepsilon^{-O(d)} \log^3 n \log^4 \log n + \sum_{j=i+1}^{h} (k_j - 1)\mu_{j+1}\varepsilon^{-O(d)} \log^3 n \log^3 \log n). \end{aligned} \tag{3.6}$$

The equality follows from the fact that $\mu_{i+1} > \mu_{j+1}$. While processing $\Delta[i]$, $O(W/\mu_{i+1})$ augmenting paths are found (see Lemma 3.8), so the first term in the RHS of equation 3.6 is $O(W\varepsilon^{-O(d)} \log^3 n \log^5 \log n)$ over all augmenting paths. Next, we bound the second term over all augmenting paths.

Any augmenting path P has at least $k_j - 1$ edges of level at most $j-1$. Furthermore, for any $j' \geq j$, every level $j-1$ edge on P will contribute at most two new cells to $k_{j'}$. Suppose γ_j is the number of level $j-1$ edges across all augmenting paths. The second term of the RHS of equation 3.6, when summed across all augmenting paths, can be written as $O(\sum_{j=1}^{h} \gamma_j \mu_{j+1}\varepsilon^{-O(d)} \log^3 n \log^3 \log n)$. Lemma 3.9 establish that $\gamma_j = O\left(\frac{W \log n}{\mu_{j+1}}\right)$.

Lemma 3.9. *For any* $1 \leq j \leq h-1$, $\gamma_j = O(W \log n/\mu_{j+1})$.

Proof. To prove this lemma, first, we bound the total additive error across all augmenting paths computed during the execution of our algorithm.

Lemma 3.10. $\sum_{P_i \in \mathbb{P}} \sum_{(u,v) \in P_i \cap M_i} \delta_{uv} = O(W \log n)$.

Every level $j-1$ edge in γ_j appears as a matching or a non-matching edge. Furthermore, any matching edge will first appear as a non-matching edge in an augmenting path and carry an additive error of μ_{j+1}. Therefore, we charge at most two level $j-1$ edges in γ_j to the additive error of any non-matching edge. So, the total additive error on all non-matching edges of γ_j is at least $\gamma_j \mu_{j+1}/2$. Combining with Lemma 3.10, we get $\gamma_j = O(W \log n/\mu_{j+1})$. □

Therefore, the second term of the RHS of equation 3.6 over all augmenting paths is $O(W\varepsilon^{-O(d)} \log^4 n \log^4 \log n)$ and the total running time is $O(W(\varepsilon^{-O(d^3)} + \varepsilon^{-O(d)} \log^4 n \log^4 \log n))$. Since $W = \Theta(n/\varepsilon)$ with probability at least $1/2$, we get the following:

Lemma 3.11. *Let* $A, B \in \mathbb{R}^d$ *be two point sets of size* n *each and a parameter* $0 < \varepsilon \leq 1$ *that satisfy* $(P1) - (P3)$. *With probability at least* $1/2$, *an* ε*-approximate matching of* A, B *can be computed in time* $O(n(\varepsilon^{-O(d^3)} + \varepsilon^{-O(d)} \log^4 n \log^4 \log n))$.

3.5 Algorithm Details

In this section, we present the details of the Build, HungarianSearch and Augment procedures for some level i. For any cell, we cluster the points inside the cell into $O(\mu_{i+1}^{1/4} \varepsilon^{-d} \log n \log \log n)$ clusters (Section 3.5.1). We use these clusters to compress the residual graph (Section 3.5.2) and feasibility conditions (Section 3.5.3). Next, we describe

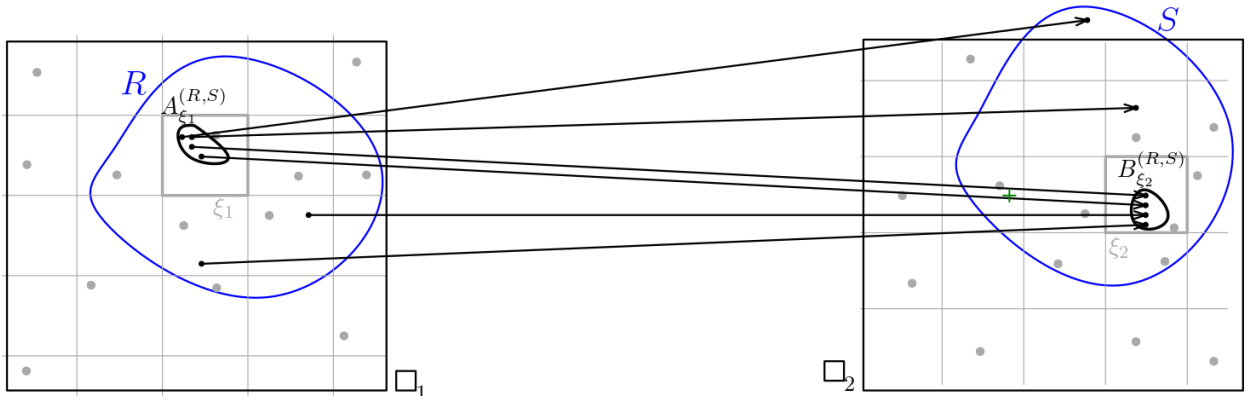


Figure 3.2: Illustrates two boundary clusters $A_{\xi_1}^{(R,S)}$ and $B_{\xi_2}^{(R,S)}$. $\square_1$ and $\square_2$ are siblings. Grey subdivision represents the subcells and matching edges are shown as solid black.

an efficient implementation of the Build, HungarianSearch, and Augment procedures using the compressed graph and feasibility conditions (Section 3.5.4). Our compressed graph is different from [73, 89] as it does not recursively expand an augmenting path in this compressed graph to an admissible augmenting path with respect to M. Instead, our algorithm may modify the matching M to another matching M' of the same cost and size and returns an admissible path with respect to M'.

We begin by introducing a few notations that will assist in describing the procedures. For any cell $\square$ and $\xi \in \mathsf{S}_\square$, let $K \subseteq \mathsf{C}[\square]$ be the subset of children of $\square$ that are contained inside ξ. Let $D(\xi) = \bigcup_{\square'' \in K} \mathsf{S}_{\square''}$ be the set of subcells of the children cells of $\square$ that lie inside ξ. For any cell $\square$ with $\mathrm{lev}(\square) < h - 1$ and for any $j < \mathrm{lev}(\square)$, let $\square^j$ be the ancestor of $\square$ at level j. Consider any $\square' \in \mathsf{C}[\square]$ and a subcell $\xi \in \mathsf{S}_{\square'}$ of $\square'$. Then, it can be shown that each level-j edge $(a, b) \in A_{\square^j} \times B_{\square^j}$ with one endpoint in ξ is covered by at least one WSPD pair from a subset $\mathcal{W}_\xi^j \subseteq \mathcal{W}_{\square^j}$, with $|\mathcal{W}_\xi^j| = O(\varepsilon^{-1} \log n)$.

Lemma 3.12. *For any cell $\square$ with $\mathrm{lev}(\square) < h - 1$, for any $j < \mathrm{lev}(\square)$, and for any subcell $\xi \in \mathsf{S}_\square$, $|\mathcal{W}_\xi^j| = O(\varepsilon^{-1} \log n)$. Furthermore, for any non-empty subcell $\xi' \in D(\xi)$, $\mathcal{W}_\xi^j = \mathcal{W}_{\xi'}^j$.*

3.5.1 Clustering points

For any cell $\square$ of level $k \in [i, h-1)$, we partition $A_\square \cup B_\square$ into a set of clusters denoted by $V_\square$. For a subcell ξ of a child $\square'$ of $\square$ (i.e, $\xi \in \mathsf{S}_{\square'}$), we partition $A_\xi \cup B_\xi$ into three types of clusters. We create one *free cluster* A_ξ^F (resp. B_ξ^F) for all free points of A_ξ (resp. B_ξ) and one *internal cluster* A_ξ^I (resp. B_ξ^I) for all points $a \in A_\xi$ (resp. $b \in B_\xi$) such that $m(a) \in \square'$ (resp. $m(b) \in \square'$). Additionally, we create *boundary clusters* as follows: For any level $j \in [i, k]$, recall that $\mathcal{W}_\xi^j$ is the set of WSPD pairs that cover all level-j edges with at least one endpoint in ξ. For every pair $(R, S) \in \mathcal{W}_\xi^j$, we create one cluster $A_\xi^{(R,S)}$ (resp. $B_\xi^{(R,S)}$) of A (resp. B) that contains all points $a \in A_\xi$ (resp. $b \in B_\xi$) whose matching edge $(a, m(a))$ (resp. $(m(b), b)$) is a level-j edge that is covered by the well-separated pair (R, S) (See Figure 3.2). For each level j, there are $O(\varepsilon^{-d} \log n)$ WSPD pairs in $\mathcal{W}_\xi^j$ and there are $O(\log \log n)$ levels. Therefore, there are $O(\varepsilon^{-d} \log n \log \log n)$ many clusters of this type.

For any cell $\square$ of level $k \in [i, h-1)$, and for any child $\square' \in \mathsf{C}[\square]$, every subcell $\xi \in \mathsf{S}_{\square'}$ is formed by combining the children of $\square'$. For any subcell $\xi \in \mathsf{S}_\square$, the free and internal clusters of ξ are formed as in Equation equation 3.7.

$$A_\xi^F = \bigcup_{\xi' \in D(\xi)} A_{\xi'}^F, \qquad B_\xi^F = \bigcup_{\xi' \in D(\xi)} B_{\xi'}^F. \tag{3.7}$$

$$A_\xi^I = \bigcup_{\xi' \in D(\xi)} \{(A_{\xi'}^I \cup \bigcup_{(R,S) \in \mathcal{W}_{\xi'}^{k+1}} A_{\xi'}^{(R,S)})\}, \qquad B_\xi^I = \bigcup_{\xi' \in D(\xi)} \{(B_{\xi'}^I \cup \bigcup_{(R,S) \in \mathcal{W}_{\xi'}^{k+1}} B_{\xi'}^{(R,S)})\}.$$

For any $i \le j \le k$ and any $(R, S) \in \mathcal{W}_\xi^j$, the boundary cluster of ξ corresponding to (R, S) is formed as follows.

$$A_\xi^{(R,S)} = \bigcup_{\xi' \in D(\xi)} A_{\xi'}^{(R,S)}, \qquad B_\xi^{(R,S)} = \bigcup_{\xi' \in D(\xi)} B_{\xi'}^{(R,S)}. \tag{3.8}$$

Based on these relations, we extend the definition of $D(\cdot)$ for any cluster X to denote the clusters that combine to form X as $D(X)$. We refer to each cluster $X' \in D(X)$ as a *child-cluster* of X.

If $\square$ is a level $h-1$ node, for every child $\square' \in \mathsf{C}[\square]$, we add $A_{\square'} \cup B_{\square'}$ to $V_\square$ and appropriately classify them as free, internal, or boundary cluster depending on whether they are free, matched inside $\square'$, or outside $\square'$.

We present an upper bound on the number of subcells of the children of any cell in the following lemma.

Lemma 3.13. *For any cell $\square$ at level $i < h-1$, $\sum_{\square' \in \mathsf{C}[\square]} |\mathsf{S}_{\square'}| = O(\mu_{i+1}^{1/4})$.*

Proof. By construction, the number of subcells of a cell $\square'$ is bounded by the number of children of $\square'$; i.e, $|\mathsf{S}_{\square'}| \leq |\mathsf{C}[\square']|$. Furthermore, for any cell $\square'$ at level j, $|\mathsf{C}[\square']| \leq (\frac{\ell_j}{\ell_{j+1}})^d = \ell_j^{d(1-\alpha)}$. As a result,

$$\sum_{\square' \in \mathsf{C}[\square]} |\mathsf{S}_{\square'}| \leq \sum_{\square' \in \mathsf{C}[\square]} |\mathsf{C}[\square']| \leq \ell_i^{d(1-\alpha)} \ell_{i+1}^{d(1-\alpha)} \leq \ell_i^{\frac{2d}{8d+2}}. \tag{3.9}$$

From Lemma 3.2, $\mu_{i+1} \geq \ell_{i+2}$. Furthermore, from the construction of the tree and since $d \geq 2$, $\ell_{i+2} = \ell_i^{(\frac{8d+1}{8d+2})^2} \geq \ell_i^{\frac{8d}{8d+2}}$. Plugging this in equation 3.9, we get $\sum_{\square' \in \mathsf{C}[\square]} |\mathsf{S}_{\square'}| = O(\mu_{i+1}^{1/4})$. $\square$

For each cell $\square$ at level $i < h-1$ and each child $\square' \in \mathsf{C}[\square]$, any subcell $\xi \in \mathsf{S}_{\square'}$ contributes $O(\varepsilon^{-d} \log n \log\log n)$ clusters to $V_\square$. Therefore, by Lemma 3.13,

Corollary 3.14. *For any cell $\square$ at level $i < h-1$, $|V_\square| = O(\mu_{i+1}^{1/4} \varepsilon^{-d} \log n \log\log n)$.*

3.5.2 Compressed residual graph

At every non-leaf node $\square$ in the tree T, we create a *compressed residual graph* $\mathcal{AG}_\square$ of $\mathcal{G}_M^\square$ with $V_\square$ being its vertex set. For any non-leaf node $\square$, the vertex set of $\mathcal{AG}_\square$ consists of one vertex for each cluster in $V_\square$. For any cell $\square$ and its child $\square' \in \mathsf{C}[\square]$, we use $V_\square(\square')$ to denote the clusters of $V_\square$ that are inside $\square'$. Next, we define $E_\square$, the set of edges of $\mathcal{AG}_\square$, and a weight $\Phi(X, Y)$ for every edge $(X, Y) \in E_\square$.

If $\text{lev}(\square) = h - 1$, we simply set the edges of $\mathcal{AG}_\square$ to be the edges of $\mathcal{G}_M^\square$ and use its net-cost as the weight, i.e., for any edge (u, v) in $\mathcal{G}_M^\square$, $\Phi(u, v) = \phi(u, v)$. If $\text{lev}(\square) < h - 1$, then we define *internal* and *bridge* edges between vertices of $V_\square$ as follows:

Bridge edges: For any two children $\square_1 \neq \square_2 \in \mathsf{C}[\square]$, let $X_1 \in V_\square(\square_1)$ and $X_2 \in V_\square(\square_2)$ be two clusters in those children, such that X_1 (resp. X_2) is a cluster of type A (resp. B). If there is at least one non-matching edge $(b, a) \in X_2 \times X_1$, we add a directed edge from X_2 to X_1 and assign it a weight equal to $\Phi(X_2, X_1) = \phi(b, a)$. We refer to this edge as a *non-matching* arc. If there is a matching edge $(a, b) \in X_1 \times X_2$, we add an edge from X_1 to X_2 and set its cost to be $\Phi(X_1, X_2) = \phi(a, b)$. We call this edge from X_1 to X_2 a *matching arc.*

We classify clusters as entry and exit clusters, and define internal edges from an entry to an exit cluster: The free cluster B_ξ^F, the internal cluster A_ξ^I, and every boundary cluster $B_\xi^{(R,S)}$, for $i \leq j \leq k$ and $(R, S) \in \mathcal{W}_\xi^j$, is designated as an *entry cluster.* The free cluster A_ξ^F, the internal cluster B_ξ^I, and every boundary cluster $A_\xi^{(R,S)}$, for $i \leq j \leq k$ and $(R, S) \in \mathcal{W}_\xi^j$, is designated as an *exit cluster.* For any cell $\square$ and its child $\square' \in \mathsf{C}[\square]$, we use $V_\square^\downarrow(\square')$ and $V_\square^\uparrow(\square')$ to denote the entry and exit clusters of $V_\square(\square')$.

We classify entry and exit clusters in this way for the following reason. Consider any augmenting path P. For any cell $\square'$, consider any edge (u, v) of P such that (u, v) is in $\mathcal{G}_M^{\square'}$ but

the edge preceding (u, v) (resp. succeeding (u, v)) is not. Then u has to be a point in an entry (resp. exit) cluster.

Internal edges: Let $\square'$ be any child of $\square$. For any pair of clusters $(X_1, X_2) \in V_{\square}^{\downarrow}(\square') \times V_{\square}^{\uparrow}(\square')$, we create an *internal edge* (X_1, X_2) in $\mathcal{AG}_{\square}$. Let $E_{\square}(\square')$ denote the set of these edges. For any $(X_1', X_2') \in V(\square') \times V(\square')$, define $P_{\square'}(X_1', X_2')$ to be the minimum-weight path between X_1' and X_2' in $\mathcal{AG}_{\square'}$. Define $\textsc{Proj}X_1, X_2 = \arg\min_{X_1' \in D(X_1), X_2' \in D(X_2)} \Phi(P_{\square'}(X_1', X_2'))$. We set $\Phi(X_1, X_2)$ to be the weight of the path $\textsc{Proj}X_1, X_2$ in $\mathcal{AG}_{\square'}$.

For consistency, if $\square$ is a cell of level $h-1$, any edge that lies completely inside a child of $\square$ becomes an internal edge and edges that go between two points in two different children is referred to as a bridge edge.

We abuse notation and refer to any directed path P between two free clusters in the compressed residual graph $\mathcal{AG}_{\square}$ as an *augmenting path*. For efficiency reasons, we only store the internal edges of $E_{\square}$. To compute the bridge edges and their costs efficiently, we construct an ε-WSPD as described in Section 3.2. In the following lemma, we bound the total size of all compressed residual graphs across all cells.

Lemma 3.15. *The total size of all compressed residual graphs across all cells of T is $O(n \log n (\varepsilon^{-1} \log\log n)^{O(d)})$.*

Proof. By construction, for any cell $\square$ and any child $\square' \in \mathsf{C}[\square]$ and for each pair of clusters $(X, Y) \in V_{\square}^{\downarrow}(\square') \times V_{\square}^{\uparrow}(\square')$, there exists at most one internal edge in $E_{\square}$. Therefore, each cluster $X \in V_{\square}(\square')$ has degree at most $|V_{\square}(\square')| = O(|\mathsf{S}_{\square'}| \varepsilon^{-d} \log n \log\log n) = O(\varepsilon^{-O(d)} \log n \log^{O(d)} \log n)$, where the last equality is resulted since $|\mathsf{S}_{\square'}| = O(\varepsilon^{-d} \log^{2d} \log n)$. Summing over all clusters in $V_{\square}$,

$$|E_{\square}| = O(\varepsilon^{-O(d)} |V_{\square}| \log n \log^{O(d)} \log n).$$

Summing across all cells in T, we get

$$\sum_{\square \in T} |E_{\square}| = \sum_{\square \in T} |V_{\square}| \times O(\varepsilon^{-O(d)} \log n \log^{O(d)} \log n). \tag{3.10}$$

Since each point participates in a single cluster per level and $O(\log \log n)$ clusters overall, we get $\sum_{\square \in T} |V_{\square}|$ by $O(n \log \log n)$. Plugging this in equation 3.10, we can conclude that the total size of the compressed residual graph across all cells is $O(n \log n(\varepsilon^{-1} \log \log n)^{O(d)})$. $\square$

Lemma 3.16. *For any cell $\square$ and for any augmenting path P from u to v in $\mathcal{G}_M^{\square}$, there is an augmenting path P' in $\mathcal{AG}_{\square}$ that goes from the cluster of u to the cluster of v and $\Phi(P') \leq \phi(P)$.*

3.5.3 Compressed Feasibility

We use the compressed residual graph to compute an augmenting path. To assist us with the computation of this path, we describe a set of dual weights of $V_{\square}$ and a set of feasibility conditions for the edges of the compressed graph. Let $\square$ be a cell of level i. For every $X \in V_{\square}$, we assign a dual weight $y(X)$. We say that a matching and dual weights are *compressed feasible* with respect to $\mathcal{AG}_{\square}$ if, for any directed edge $(X, Y) \in E_{\square}$,

$$y(X) - y(Y) \leq \Phi(X, Y). \tag{3.11}$$

Next, we define slack on any compressed edge (X, Y) to be $s(X, Y) = \Phi(X, Y) - y(X) + y(Y)$. Note that $s(X, Y) \geq 0$ for a compressed feasible matching. We say that an edge (X, Y) is *admissible* if $s(X, Y) = 0$. Similar to equation 3.3, one can express the weight of any path

P in $\mathcal{AG}_{\square}$ from X to Y using the slacks on its edges as follows.

$$\Phi(P) = \sum_{(X',Y')\in P} (y(X') - y(Y') + s(X',Y')) = y(X) - y(Y) + \sum_{(X',Y')\in P} s(X',Y'). \quad (3.12)$$

After any execution of Build, HungarianSearch, or Augment procedures at $\square$ within our algorithm, in addition to (J1) and (J2), our algorithm also satisfies a third invariant. Let $\square'$ be either $\square$ or any descendant of $\square$ in T.

(J3) There exists a set of dual weights $y(\cdot)$ on $V_{\square'}$ that satisfy compressed feasibility conditions for edges in $\mathcal{AG}_{\square'}$. In addition, for any $X \in V_{\square'}$, if X is a free cluster of A, then $y(X) = 0$ and if X is a free cluster of B, then $y(X) = \max_{X'\in V_{\square'}} y(X')$. Furthermore, for any cluster $X \in V_{\square'}$,

(a) if X is an internal entry (resp. exit) cluster, $y(X) \leq \min_{X'\in D(X)} y(X')$ (resp. $y(X) \geq \max_{X'\in D(X)} y(X')$), and

(b) if X is a free or a boundary cluster then for every cluster $X' \in D(X)$, $y(X') = y(X)$.

3.5.4 Details of the Procedures

Assume that the algorithm has executed until level $i+1$ and (J1)–(J3) hold at the end. We describe the implementation of Build, HungarianSearch and Augment procedures and show that (J1)–(J3) continue to hold after their executions.

Build procedure

For any cell $\square$ of level i and for every $\square' \in \mathsf{C}[\square]$, we have a set of compressed feasible dual weights on $V_{\square'}$. The **Build** procedure creates a cluster X at $\square$ by simply combining the clusters $D(X)$ of its children and set its dual weight $y(X)$ to $\min_{X' \in D(X)} y(X')$ (resp. $\max_{X' \in D(X)} y(X')$) provided X is an entry (resp. exit) cluster.

In order to compute the weight of any internal edge $(X, Y) \in V_{\square}^{\downarrow}(\square') \times V_{\square}^{\uparrow}(\square')$, we observe that from equation 3.12, the minimum-weight path $P(X, Y)$ is also the path with the smallest total slack between any two clusters $X', Y' \in D(X) \times D(Y)$ in $\mathcal{AG}_{\square'}$. For every entry cluster $X \in V_{\square}^{\downarrow}(\square')$, this can be found in a straight-forward way using an execution of Dijkstra's shortest-path algorithm. More specifically, we add a source s to $\mathcal{AG}_{\square'}$, connect s to all $X' \in D(X)$ with an edge of weight of $y(X')$, and replace the weight of every other edge in $\mathcal{AG}_{\square'}$ with its slack. Then, we execute Dijkstra's algorithm in $\mathcal{AG}_{\square'}$ from s to find the distance of each cluster Y' from s, denoted by $\kappa_{Y'}$. For any exit cluster $Y \in V_{\square}^{\uparrow}(\square')$, we set the weight of the edge (X, Y) in $\mathcal{AG}_{\square}$, to be $\Phi(X, Y) = \min_{Y' \in D(Y)} \{\kappa_{Y'} - y(Y')\}$.

The **Build** procedure does not affect the invariants (J1) and (J2). The following lemma states that the invariant (J3) holds after the execution of **Build** procedure.

Lemma 3.17. *The dual weights assigned to the clusters of $V_{\square}$ by the* **Build** *procedure satisfy (J3).*

Efficiency of the Build procedure: To compute the internal edges incident on the entry cluster X, instead of using an $O(|V_{\square'}|^2)$ time Dijkstra's algorithm, as in [73], we use the WSPDs in a straight-forward way to compute the shortest path in $O(|E_{\square'}| \log |E_{\square'}|)$ time. Given that the number of entry clusters in each cell is $\log n(\varepsilon^{-1} \log \log n)^{O(d)}$ and since $|E_{\square'}| = O(\varepsilon^{-O(d)} n_{\square'} \log n \log^{O(d)} \log n))$ (from Lemma 3.15), the total running time of the

Build procedure is

$$O\left(\sum_{\square' \in \mathsf{C}[\square]} \sum_{X \in V_{\square}^{\downarrow}(\square')} \varepsilon^{-O(d)} n_{\square'} \log^2 n \log^{O(d)} \log n\right) = O(n_{\square} \log^3 n_{\square} (\varepsilon^{-1} \log \log n)^{O(d)}).$$

HungarianSearch procedure

Let $\square$ be a cell of level i. The HungarianSearch procedure on $\square$ consists of two parts. In the first part, the algorithm modifies the dual weights of $V_{\square}$ and make the edges on the minimum-weight augmenting path in $\mathcal{AG}_{\square}$ admissible (Search step). Then, the algorithm updates the dual weights of the clusters and points and may modify the matching M to another feasible matching M' with $w(M') = w(M)$. Then, it returns an admissible augmenting path in $\mathcal{G}_{M'}^{\square}$ with respect to M' (Update step). This path is also the minimum net-cost augmenting path with respect to M'.

Search Step: From equation 3.12, the path from a free cluster F' to a free cluster F with the smallest total slack is also the minimum-weight path between F' and F in $\mathcal{AG}_{\square}$. To compute this path, we replace the cost of every edge in $\mathcal{AG}_{\square}$ with its slack and then execute a Dijkstra's algorithm that starts at the free clusters of B. Let P_X be the shortest path from a free cluster of B to any cluster X and κ_X be its cost. Let F be a free cluster of A that has the smallest shortest path. If there are no free clusters of A in $\mathcal{AG}_{\square}$, then the HungarianSearch returns NULL. Let the path P_F start at some free cluster F' of B. P_F is the minimum-weight augmenting path in $\mathcal{AG}_{\square}$ and $y(F) = 0$ (from (J3)). Therefore, from equation 3.12, the augmenting path P_F has a weight of $\Phi(P_F) = y(F') + \kappa_F$. If $\Phi(P_F) \leq \mu_i$, let $U \subseteq V_{\square}$ be the subset of clusters whose shortest-path distances from s is less than κ_F. We update the dual weights of any cluster $X \in U$ by setting $y(X) \leftarrow y(X) - \kappa_X + \kappa_F$. If $\Phi(P_F) > \mu_i$, the algorithm sets $\kappa_F = \mu_i - y(F')$, updates the dual weights as described above and then

returns NULL. The dual updates to the clusters of $V_\square$ in the search step ensures that the dual weights of free clusters of B do not exceed μ_i.

For any cell $\square_1$ and its child $\square_2 \in \mathsf{C}[\square_1]$, we say that the dual weights of $V_{\square_1}$ *dominates* the dual weights of $V_{\square_2}$ if for each exit cluster $X \in V^{\uparrow}_{\square_1}(\square_2)$, $y(X) \geq \max_{X' \in D(X)} y(X')$. During the search step, the dual weight of any cluster $X \in V_\square$ is non-decreasing. Therefore, after the search step, for each child $\square' \in \mathsf{C}[\square]$, the dual weights of $V_\square$ dominates the dual weights of $V_{\square'}$. Furthermore, the updated dual weights are compressed feasible and the edges of P_F is admissible.

Lemma 3.18. *For any cell $\square$, after the execution of the search step of the* HungarianSearch *procedure, the updated dual weights of $V_\square$ are compressed feasible and every edge on the minimum-weight path computed by the search step is admissible. Furthermore, for any child $\square' \in \mathsf{C}[\square]$, the dual weights of $V_\square$ dominate the dual weights of $V_{\square'}$.*

After the search step, the updated dual weights of $V_\square$ remain compressed feasible and the edges of P_F are admissible. In the Update Step, we expand the path P_F into an admissible augmenting path in the residual graph. We describe a procedure called Sync that assists in expanding this path. In particular, consider any admissible edge (X, Y) on P_F where (X, Y) is an internal edge for some child $\square' \in \mathsf{C}[\square]$. The Sync procedure updates the dual weight of $V_{\square'}$ so that the path $P(X, Y)$ becomes admissible.

Sync Procedure: The Sync procedure takes any cell $\square_1$ and its child $\square_2 \in \mathsf{C}[\square_1]$ along with a set of compressed feasible dual weights of $V_{\square_1}$ and $V_{\square_2}$ such that the dual weights of $V_{\square_1}$ dominates the dual weights of $V_{\square_2}$. This procedure then generates a set of compressed feasible dual weights of $V_{\square_2}$ such that the dual weights of $V_{\square_1}$ and $V_{\square_2}$ satisfy conditions (a) and (b) of (J3). Furthermore, if any internal edge $(X, Y) \in E_{\square_1}(\square_2)$ is admissible, then the path $P(X, Y)$ is also admissible with respect to the dual weights of $V_{\square_2}$. We will next

describe the Syncprocedure in more detail.

For any entry cluster $X \in V_{\square}^{\downarrow}$ and any cluster $X' \in D(X)$, the dual weight $y(X')$ needs to be no less than the updated $y(X)$ (In the case of free or boundary clusters, it should be equal). We define $\mathrm{inc}(X')$ to be the value by which $y(X')$ should be increased, i.e., $\mathrm{inc}(X') = y(X) - y(X')$. Note that $\mathrm{inc}(X')$ can be negative. Let $\rho_X = \max_{X' \in D(X)} \mathrm{inc}(X')$ and let $\rho = \max\{0, \max_{X \in V_{\square}^{\downarrow}(\square')} \rho_X\}$. The value ρ corresponds to the largest increase in dual weights we desire across all child-clusters in each entry cluster. So, for any cluster $X \in V_{\square}^{\downarrow}(\square')$ and $X' \in D(X)$, $-\infty < \mathrm{inc}(X') \leq \rho$.

Let $\mathcal{AG}'$ be an augmented compressed residual network that is created by adding a vertex s to $\mathcal{AG}_{\square'}$ and connecting s to every $X' \in D(X)$ for any entry cluster $X \in V_{\square}^{\downarrow}(\square')$. We set the weight of (s, X') to be $\rho - \mathrm{inc}(X')$. Since $\mathrm{inc}(X') \leq \rho$, the weight on the edge will be non-negative. For every other edge (U, V), we set its weight to be the slack $s(U, V)$. We then execute Dijkstra's algorithm on $\mathcal{AG}'$ from the source s. For any cluster V, let κ_V denote the weight of the shortest-path distance from s to V. The dual updates are done in an identical fashion to the HungarianSearch. Let U denote the set of all clusters $V \in V_{\square'}$ with $\kappa_V < \rho$. For any $V \in U$, we update the dual weight $y(V) \leftarrow y(V) - \kappa_V + \rho$.

After updating these dual weights, for every boundary and free exit cluster $X \in V_{\square}^{\uparrow}(\square')$ and any $X' \in D(X)$, we set $y(X') \leftarrow y(X)$. This step will not decrease the dual weight of any cluster. This completes the description of Sync procedure.

Let $\square_1$ be a cell of level j of T. The execution of the Sync procedure on $\square_2$ requires an execution of Dijkstra's algorithm on $\mathcal{AG}_{\square_2}$ and takes a total of $O(\mu_{j+1}\varepsilon^{-O(d)} \log^3 n \log^2 \log n)$ time. The following lemma states the important properties of the Sync procedure.

Lemma 3.19. *For any cell $\square_1$ and any child $\square_2 \in \mathsf{C}[\square_1]$, suppose the set of dual weights of $V_{\square_1}$ and $V_{\square_2}$ are compressed feasible and the set of dual weight of $V_{\square_1}(\square_2)$ dominates the set*

of dual weights of $V_{\square_2}$. After applying the Sync *procedure on $\square_2$, we have that:*

(i) The updated dual weights on $\mathcal{AG}_{\square_2}$ are compressed feasible,

(ii) For any cluster $X \in V_{\square_1}(\square_2)$, the dual weight of X and clusters in $D(X)$ satisfies the conditions (a) and (b) of (J3), and,

(iii) If $\square_2$ is not a leaf cell, for any $\square_3 \in \mathsf{C}[\square_2]$, the dual weights of $V_{\square_2}(\square_3)$ dominates the set of dual weights of $V_{\square_3}$.

Using Lemma 3.19 part (iii), it is clear that one can recursively apply the Sync procedure on all descendants of a cell $\square$. The following lemma shows that after the search step, if we apply the Sync procedure on all descendants of every cell $\square \in \Delta[i]$, the dual weights assigned to all clusters and points satisfy (J1)–(J3).

Lemma 3.20. *After executing the search step of* HungarianSearch *on $\mathcal{AG}_{\square}$, for every $\square \in \Delta[i]$, suppose we apply* Sync *to $\square$ and all its descendants. The resulting up-to-date dual weights satisfy (J1)–(J3).*

The following lemma helps in converting the minimum-weight path obtained by the search step into an admissible augmenting path.

Lemma 3.21. *For any admissible internal edge $(X, Y) \in E_{\square}(\square')$, let $X' \in D(X)$ and $Y' \in D(Y)$ be the clusters containing the first and the last vertex of some $P(X, Y)$. Then, after calling the* Sync *procedure on $\square'$, the path $P(X, Y)$ is admissible, $y(X') = y(X)$, and $y(Y') = y(Y)$.*

Using the Sync procedure, the Update step converts the augmenting path P_F returned by the Search step to an admissible augmenting path $\tilde{P}_F$ in the residual graph. In this process, the Update step might change the matching M to another matching M' with the same weight

and size. We describe the Update step as a recursive procedure that initially takes P_F as the input.

Update step: For any cell $\square'$, the Update step takes any admissible path $P = \langle X = X_1, X_2, \ldots, X_m = Y \rangle$ in $\mathcal{AG}_{\square'}$ as input and returns an admissible alternating path from a point p to p' in $\mathcal{G}_M^{\square'}$ with the property that $y(p) = y(X)$ and $y(p') = y(Y)$.

If $\square'$ is a cell of level $h-1$, then P is also an admissible path in $\mathcal{G}_M^{\square'}$ and the procedure returns this path. Otherwise, let $k = \text{lev}(\square') < h-1$ be the level of $\square'$. Let $\mathcal{I}$ denote the set of all internal edges on the path P. Note that $\mathcal{I}$ is a set of vertex-disjoint edges. Let $\mathcal{B}$ be the set of all clusters X_t on P that do not participate in any edge of $\mathcal{I}$. It is easy to see that $\mathcal{B}$ is a set of boundary clusters.

For any internal edge $(X_j, X_{j+1}) \in \mathcal{I}$, let $\square_j \in \mathsf{C}[\square']$ such that $(X_j, X_{j+1}) \in E_{\square'}(\square_j)$. We execute the Sync procedure on $\square_j$. Since (X_j, X_{j+1}) is an admissible edge, the path $P(X_j, X_{j+1})$ is admissible with $y(X'_j) = y(X_j)$ and $y(X'_{j+1}) = y(X_{j+1})$ (From Lemma 3.21). We recursively apply the Update step on $P(X_j, X_{j+1})$.

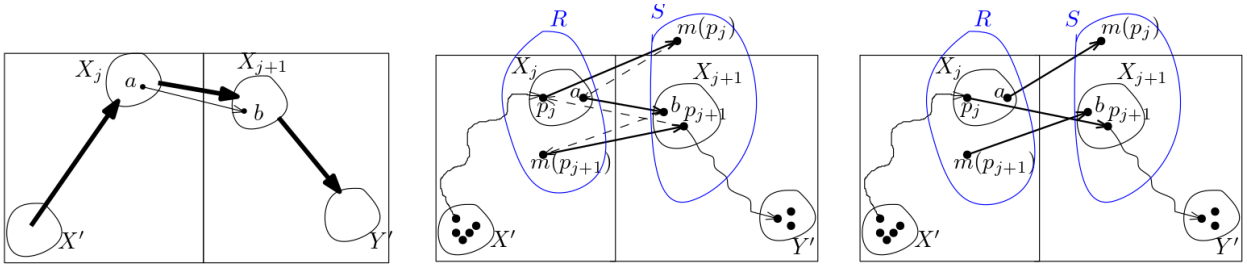
Assume that for each internal edge $(X_j, X_{j+1}) \in \mathcal{I}$, this recursive call has returned an admissible path Π_j in the residual graph from a point $p_j \in X_j$ to a point $p_{j+1} \in X_{j+1}$ with $y(p_j) = y(X_j)$ and $y(p_{j+1}) = y(X_{j+1})$. We *select* the point p_j for the cluster X_j and the point p_{j+1} for the cluster X_{j+1}. For any boundary cluster $X_t \in \mathcal{B}$, we select an arbitrary point p_t. Let Z be the set of all ancestors of p_t in T of level greater than k. First, we iteratively apply Sync on every cell in Z in increasing order of their level. After all executions of the Sync procedure, from Lemma 3.19 (ii), $y(p_t) = y(X_t)$. Thus, from every cluster X_j on P, we have selected one point p_j with $y(p_j) = y(X_j)$.

Next, we construct the admissible alternating path $\tilde{P}$ corresponding to the path P as follows. For every internal edge $(X_j, X_{j+1}) \in \mathcal{I}$, we replace (X_j, X_{j+1}) with the path Π_j. For every

bridge edge (X_j, X_{j+1}), if (X_j, X_{j+1}) is a non-matching arc, we simply add an edge from p_j to p_{j+1} to the path. If (X_j, X_{j+1}) is a matching arc, then both X_j and X_{j+1} are boundary clusters. While there may not be a matching edge between p_j and p_{j+1}, we know that there is some matching edge (a, b) such that $a \in X_j$ and $b \in X_{j+1}$. Let Z (resp. Z') be the set of all ancestors of a (resp. b) of level greater than k. We apply the Sync procedure on the cells in Z (resp. Z') in increasing order of their level. Then, we modify our matching M to M' as follows: Add matching edges $(p_j, p_{j+1}), (a, m(p_j))$, and $(b, m(p_{j+1}))$ to the matching and remove the edges $(a, b), (p_j, m(p_j))$, and $(p_{j+1}, m(p_{j+1}))$ from the matching (See Figure 3.3). The new matching continues to be feasible since the dual weights of a (resp. b) and p_j (resp. p_{j+1}) are identical. This is because X_j (resp. X_{j+1}) is a boundary cluster and therefore, from Lemma 3.19 (ii), the application of the Sync procedure on the ancestors of p_j (resp. p_{j+1}) and a (resp. b) will make $y(a) = y(p_j)$ (resp. $y(b) = y(p_{j+1})$). Note that, for every internal edge (X_j, X_{j+1}), the path Π_j consists of only admissible edges. Furthermore, for every bridge edge (X_j, X_{j+1}), the edge (p_j, p_{j+1}) added to the path is admissible. This follows from the fact that $y(p_j) = y(X_j)$, $y(p_{j+1}) = y(X_{j+1})$, and (X_j, X_{j+1}) is admissible. Finally, the dual weight of the first (resp. last) point p_1 (resp. p_m) is equal to $y(X)$ (resp. $y(Y)$) as desired. This completes the description of the Update step. Let $\tilde{P}_F$ be the admissible augmenting path in $\mathcal{G}_\square$ returned by the Update step with P_F as input.

Lemma 3.22. *The matching M can be modified to another matching M' so that, $w(M) = w(M')$, $M', y(\cdot)$ is feasible and the compressed residual graph at each node remains unchanged. Furthermore, there is an admissible augmenting path in the residual graph $\mathcal{G}_{M'}$.*

Recollect that, we only require the existence of a set of dual weights that satisfy the conditions in (J1)–(J3). For efficiency reasons, the Update step does not maintain the up-to-date dual weights explicitly. Instead, it computes the up-to-date dual weights for all cells $\square'$ whose $V_{\square'}$ or $M_{\square'}$ may change after augmenting M along $\tilde{P}_F$. For every other cell $\square''$, from Lemma 3.23



(a) (b) (c)

Figure 3.3: (a) Compact minimum-weight path P in $\mathcal{AG}_{\square}$. (b) and (c) show how the matching M is modified to a matching M' to obtain an augmenting path.

and Lemma 3.20, we can always retrieve the up-to-date dual weights for these cells satisfying (J1)–(J3) by recursively applying Sync on $\square''$ and all its descendants.

Lemma 3.23. *During the execution of our algorithm, consider a sequence of consecutive applications of the* Sync *procedure on a cell* $\square$*. If* $M_{\square}$ *and the clusters in* $V_{\square}$ *remain unchanged, then, this sequence of* Sync *executions can be replaced with the last one while producing the same set of dual weights of* $V_{\square}$*.*

Efficiency of the HungarianSearch procedure: The search step requires execution of a single Dijkstra's algorithm on $\mathcal{AG}_{\square}$ which takes $O(\mu_{i+1}\varepsilon^{-O(d)}\log^3 n \log^2 \log n)$ time. Applying Sync procedure for a cell $\square'$ of level j requires an execution of Dijkstra's algorithm on $\mathcal{AG}_{\square'}$, which takes $O(\mu_{j+1}\varepsilon^{-O(d)}\log^3 n \log^2 \log n)$ time. Recall that for any level j of T and an augmenting path $\tilde{P}_F$ on the residual graph, k_j denotes the number of level j cells containing at least one point of $\tilde{P}_F$. In the Update step, For each level $j > i$, we execute the Sync procedure on the cell of level j containing at least one point of $\tilde{P}_F$. Furthermore, during the Update step, for each bridge matching arc (X_j, X_{j+1}) the algorithm may also apply the Sync procedure on an additional $O(\log \log n)$ cells. This is done before the algorithm modifies the matching. These executions of the Sync procedure can be charged to the $O(\log \log n)$ Sync procedures executed for the ancestors of points $p_j \in X_j$ and $p_{j+1} \in X_{j+1}$.

Therefore, there are at most $O(k_j \log\log n)$ executions of the Sync procedure during the execution of the Update step. The execution time of HungarianSearch procedure, therefore, is $O(\mu_{i+1}\varepsilon^{-O(d)} \log^3 n \log^2 \log n + \sum_{j=i+1}^{h-1} k_j \mu_{j+1} \varepsilon^{-O(d)} \log^3 n \log^3 \log n)$.

Augment procedure

Given an augmenting path $P = \langle b_0, a_0, b_1, \cdots, b_k, a_k \rangle$ with respect to the matching M, the Augment procedure will simply update $M \leftarrow M \oplus P$. After augmentation, any edge (a_i, b_i) is a matching edge and any edge (b_{i+1}, a_i) is a non-matching edge (Note that the direction of the edges are reversed after the augmentation). For a new matching edge (a_i, b_i) after an augmentation, suppose $\square_i$ is the least common ancestor of a_i and b_i. Let $X, Y \in V_{\square_i}$ be the pair of boundary clusters such that $(a_i, b_i) \in X \times Y$. If there exists another matching edge $(a'_i, b'_i) \in (X \times Y) \setminus P$, then a_i and b_i inherit their dual weights, i.e., $y(a_i) \leftarrow y(a'_i)$ and $y(b_i) \leftarrow y(b'_i)$. Otherwise, their dual weight remains unchanged. This completes the description of the Augment procedure. For every new matching edge (a_i, b_i), the procedure may inherit the dual weights from another matching edge (a'_i, b'_i) that did not participate in P. Since (J2) holds prior to augmentation, $y(a'_i)$ and $y(b'_i)$ satisfy (J2). Therefore, post augmentation, $y(a_i)$ and $y(b_i)$ also satisfy (J2).

The following lemma shows that the dual weights of the points after the Augment procedure remains feasible and (J1) holds.

Lemma 3.24. *After augmenting the matching and updating the dual weights by the* Augment *procedure, the dual weights of the points are feasible with respect to the new matching.*

The vertex and the edge sets of the compressed residual graph change after augmentation. The Augment procedure creates the new clusters at all ancestors of every point in P in a straight-forward way. In a bottom-up fashion, for any cluster $X \in V_{\square}$, if X is an exit (resp.

entry) cluster, it assigns $\max_{X' \in D(X)} y(X')$ (resp. $\min_{X' \in D(X)} y(X')$) as $y(X)$. For each edge on P, the procedure will update the bridge edges in $\mathcal{AG}_\square$ in a straight-forward way. The following lemma shows that the updated dual weights are compressed feasible with respect to $\mathcal{AG}_\square$.

Lemma 3.25. *After augmenting the matching, the new set of clusters and their dual weights $y(\cdot)$ satisfy (J3).*

Next, for any $\square' \in \mathsf{C}[\square]$, in order to update the weights on the internal edges $E_\square(\square')$ in $\mathcal{AG}_\square$, we apply the **Build** procedure. From Corollary 3.14, if $\square$ is a cell of level i, then $|V_\square| = O(\mu_{i+1}^{1/4} \varepsilon^{-d} \log n \log\log n)$. Since there at most $O(|V_\square|)$ entry clusters in $V_\square$ and the **Build** procedure executes that many Dijkstra's algorithm to construct the internal edges, the total time taken is bounded by $O(\mu_{i+1} \varepsilon^{-O(d)} \log^3 n \log^3 \log n)$.

The **Augment** procedure executes **Build** procedure on all ancestors of any vertex of P in a bottom-up fashion. Therefore, the total time taken is $O(\sum_{j=1}^{i} (k_j \mu_{i+1} \varepsilon^{-O(d)} \log^3 n \log^3 \log n))$.

This completes the description of our first result for the EBM problem. We will next describe our second result that consists of an algorithm to compute a higher precision matching in certain applications.

Chapter 4

A Higher Precision Algorithm for Computing the EBM

In this chapter, we present an algorithm that boosts the accuracy of any ε-additive approximation algorithm, achieving an expected additive error of $\min\{\varepsilon, (d \log\log n)w^*\}$ from the optimal matching cost w^* in $O(T(n, \varepsilon/d) \log\log n)$ time, where $T(n, \varepsilon)$ is the time complexity of any given ε-additive approximation algorithm. We start by describing the problem.

4.1 Problem Definition

In the 1-Wasserstein problem, we are given two discrete distributions μ and ν. Let A and B be the points that define μ and ν, respectively. For the distribution μ (resp. ν), suppose each point $a \in A$ (resp. $b \in B$) has a probability of μ_a (resp. ν_b) associated with it, where $\sum_{a\in A} \mu_a = \sum_{b\in B} \nu_b = 1$. Let $G(A, B)$ denote the complete bipartite graph where, for any pair of points $a \in A$ and $b \in B$, there is an edge from a to b of cost $\|a-b\|$; i.e, the Euclidean distance between a and b.

For each point $a \in A$ (resp. $b \in B$), we assign a weight $\eta(a) = -\mu_a$ (resp. $\eta(b) = \nu_b$). We refer to any point $v \in A \cup B$ with a negative (resp. positive) weight as a *demand point* (resp. *supply point*) with a demand (resp. supply) of $|\eta(v)|$. Given any subset of points $V \subseteq A \cup B$, the weight $\eta(V)$ is simply the sum of the weights of its points; i.e, $\eta(V) = \sum_{v\in V} \eta(v)$. For

any edge $(a, b) \in A \times B$, let the cost of transporting a supply of β from b to a be $\beta\|a-b\|$. In this problem, our goal is to transport all supplies from supply points to demand points with a minimum cost. More formally, a transport plan is a function $\sigma : A \times B \to \mathbb{R}_{\geq 0}$ that assigns a non-negative value to each edge of $G(A, B)$ indicating the quantity of supply transported along the edge. The transport plan σ is such that the total supplies transported into (resp. from) any demand (resp. supply) point $a \in A$ (resp. $b \in B$) is equal to $-\eta(a)$ (resp. $\eta(b)$). $w(\sigma)$ is the total cost of the transport plan σ which is computed as $\sum_{(a,b)\in A\times B} \sigma(a, b)\|a-b\|$. The objective is to find a transport plan that has a minimum cost.

If two points $a \in A$ and $b \in B$ are co-located (i.e, they share the same coordinates), then due to the metric property of the Euclidean distances, if $\eta(b) = -\eta(a)$, we can match the supplies to the demands at zero cost and remove the points from the input. Otherwise, if $\eta(b) \neq -\eta(a)$, we replace the two points with a single point of weight $\eta(a) + \eta(b)$. By definition, if the weight of the newly created point is negative (resp. positive), we consider it a demand (resp. supply) point. In our presentation, we always consider A and B to be the point sets obtained after replacing all the co-located points. Observe that the total supply $U = \eta(B)$ need not be 1. However, it is easy to see that $\eta(B) = -\eta(A)$; i.e, the problem instance defined on $A \cup B$ is *balanced*. We say that a transport plan σ is an ε-close transport plan if $w(\sigma) \leq \mathcal{W}(\mu, \nu) + \varepsilon U$.

In many applications, the distributions μ and ν are continuous or large (possibly unknown) discrete distributions. In such cases, it might be impossible to compute $\mathcal{W}(\mu, \nu)$. Instead, one can draw two sets A and B of n samples each from μ and ν, respectively. Each point $a \in A$ (resp. $b \in B$) is assigned a weight of $\eta(a) = -1/n$ (resp. $\eta(b) = 1/n$). One can approximate the 1-Wasserstein distance between the distributions μ and ν by simply solving the 1-Wasserstein problem defined on $G(A, B)$. This special case where every point has the same demand and supply is identical to the Euclidean Bipartite Matching (EBM)

problem. A matching M here is a set of vertex-disjoint edges in $G(A, B)$ and has a cost $1/n \sum_{(a,b)\in M} \|a - b\|$. For the special case of the EBM problem, the optimal transport plan is simply a minimum-cost matching of cardinality n.

For any point set P in the Euclidean space, let $C_{\max}(P) := \max_{(a,b)\in P\times P} \|a - b\|$ denote the distance of its farthest pair and $C_{\min}(P) := \min_{(a,b)\in P\times P, a\neq b} \|a - b\|$ denote the distance of its closest pair. The *spread* of the point set, denoted by $\Delta(P)$, is the ratio $\Delta(P) = C_{\max}(P)/C_{\min}(P)$. When P is obvious from the context, we simply use $C_{\min}$, $C_{\max}$, and Δ to denote the distance of its closest and farthest pair and its spread.

4.2 Our Results

Let $T(n, \varepsilon)$ be the time taken by an ε-additive approximation algorithm on an input of n points in the unit hypercube. In Theorem 4.1 and Theorem 4.2, we present new algorithms that improve the accuracy of any additive approximation algorithm for the 1-Wasserstein problem and the EBM problem, respectively.

Theorem 4.1. *There exists a randomized algorithm that, given two discrete distributions μ and ν in the d-dimensional unit hypercube with a spread of $\Delta = n^{O(1)}$ and a parameter $\varepsilon > 0$, computes a transport plan with an expected additive error of $\min\{\varepsilon, (d \log_{\sqrt{d}/\varepsilon} n)\mathcal{W}(\mu, \nu)\}$ in $O(T(n, \varepsilon/d) \log_{\sqrt{d}/\varepsilon} n)$ time; here, $\mathcal{W}(\mu, \nu)$ is the 1-Wasserstein distance between μ and ν.*

Theorem 4.2. *There exists an algorithm that, given two sets of samples A and B each from distributions μ and ν in the d-dimensional unit hypercube , where $|A| = |B| = n$, and a parameter $\varepsilon > 0$, computes, with high probability, a matching that is within an expected additive error of $\min\{\varepsilon, (d \log\log n)w^*\}$ from the optimal matching cost w^* in $O(T(n, \varepsilon/d) \log\log n)$ time.*

Typical additive approximation algorithms run in $T(n, \varepsilon) = \tilde{O}(n^2)$ time and compute an ε-close transport plan for any arbitrary cost function; i.e, they make no assumption about the distance between points. The inputs to our algorithm, on the other hand, are point sets in the Euclidean space. Therefore, one can use an approximate dynamic nearest neighbor data structure to improve the execution time of such additive approximation algorithms. In particular, the algorithm by [74] executes in $O(1/\varepsilon)$ phases, where in each phase, they execute one iteration of Gabow-Tarjan's algorithm. As shown by [3], one can use a $\frac{1}{\sqrt{\varepsilon}}$-approximate dynamic nearest neighbor data structure with a query/update time of $O(n^{\varepsilon} + d \log n)$ ([20]) to execute each iteration of Gabow-Tarjan's algorithm in $\tilde{O}(n^{1+\varepsilon})$ time. Combining with the algorithms from Theorem 4.1 and Theorem 4.2, we obtain the following relative approximation algorithms. The details are provided in Section 4.2.1.

Theorem 4.3. *There exists a randomized algorithm that, given two discrete distributions μ and ν in the d-dimensional unit hypercube in Euclidean space with a spread of $\Delta = n^{O(1)}$ and a parameter $\varepsilon > 0$, computes a $O(d/\varepsilon^{3/2})$-approximate transport plan in $O(d^2 n^{1+\varepsilon}/\varepsilon)$ time.*

Theorem 4.4. *There exists an algorithm that, given two sets of samples A and B each from distributions μ and ν in the d-dimensional unit hypercube in Euclidean space, where $|A| = |B| = n$, and a parameter $\varepsilon > 0$, computes, with high probability, an $O(\frac{d}{\sqrt{\varepsilon}} \log \frac{d}{\varepsilon})$-approximate matching in $O(dn^{1+\varepsilon} \log \frac{d}{\varepsilon})$ time.*

In contrast to our results, for the EBM problem, [4] compute an $O(d^2 \log \frac{1}{\varepsilon})$-approximate matching in the same running time. Therefore, our algorithm computes a more accurate EBM when $d > \frac{1}{\sqrt{\varepsilon}}$. For instance, consider the case where $d = \sqrt{\log n}$. For any arbitrarily small constant $\varepsilon > 0$, the algorithm of Theorem 4.4 will run in $\tilde{O}(n^{1+\varepsilon})$ time and return an $O(\sqrt{\log n})$-approximation. In contrast, all previous methods that achieve sub-logarithmic approximation require $\Omega(n^{5/4})$ time ([3]).

We also note that all of our algorithms extend to any ℓ_p norm in a straight-forward way[1]. For simplicity, we restrict our presentation only to the Euclidean case.

Overview of the Algorithm

Our algorithm uses the hierarchical greedy paradigm. In our presentation, we refer to a hypercube as a *cell*. For any cell $\square$, let $V_\square$ be the set of all points of $A \cup B$ that lies inside $\square$. Unlike in the quad-tree greedy algorithm, instead of splitting each cell $\square$ into 2^d cells, we split it into $\min\{|V_\square|, (4\sqrt{d}/\varepsilon)^d\}$ cells. Thus, the height of the resulting tree T reduces from $O(\log n)$ to $O(\log_{\sqrt{d}/\varepsilon} n)$. For any cell $\square$ of T and any child $\square'$ of $\square$, we move any excess supply or demand inside $\square'$ to its center. Let $\mathcal{A}_\square$ (resp. $\mathcal{B}_\square$) be a set consisting of the center points of all children of $\square$ with excess demand (resp. supply). For any child $\square'$ of $\square$ with excess demand (resp. supply), we assign a weight of $\eta(V_{\square'})$ to its center point in $\mathcal{A}_\square$ (resp. $\mathcal{B}_\square$). Using an additive approximation algorithm, we compute an (ε/d)-close transport cost between $\mathcal{A}_\square$ and $\mathcal{B}_\square$ in $T(|V_\square|, \varepsilon/d)$ time. We report the sum of the transport costs computed at all cells of T as an approximate 1-Wasserstein distance. This simple algorithm guarantees improvement in the quality of the solutions produced by both additive and relative approximation algorithms.

From the stand-point of relative approximation algorithms, [4] as well as [61] have utilized a very similar hierarchical framework to design approximation algorithms. However, unlike our algorithm, they used an exact solver that takes $\Omega(|V_\square|^3)$ time. As a result, to obtain near-quadratic execution time, they needed every cell to be divided into at most $|V_\square|^{2/3}$ children, i.e., $1/\varepsilon^d \leq n^{2/3}$ or $d \leq \log_{1/\varepsilon} n$. This also forces the height of the tree to be $O(d \log_{1/\varepsilon} n)$ leading to an $O(d^2 \log_{1/\varepsilon} n)$-factor approximation. We replace the $\Omega(n^3)$ time

[1]Owing to a higher complexity of LSH in arbitrary ℓ_p norms, the approximation factor in Theorem 4.3 and Theorem 4.4 is slightly higher for arbitrary ℓ_p norms.

exact solver in their algorithm with a $T(|V_\square|, \varepsilon) = \tilde{O}(|V_\square|^2)$ time additive approximation algorithm. Therefore, each instance (regardless of the number of non-empty children) can be solved in $\tilde{O}(|V_\square|^2)$ time. As a result, we are able to improve the approximation factor from $O(d^2 \log_{1/\varepsilon} n)$ to $O(d \log_{\sqrt{d}/\varepsilon} n)$ and also remove restrictions on the dimension. Our algorithm now works for any dimension!

Technical Challenge: By using an additive approximation algorithm to solve each instance, we have an increased error that may be difficult to bound. We use the following observation to overcome this challenge. In Section 4.3.3, we show that for any point set with spread Δ, an additive approximation algorithm can be used to compute a 2-relative approximation in $T(n, 1/\Delta)$ time. Since the spread of the point set at each cell $\square$ is d/ε, we get a 2-relative approximation by using an additive approximation algorithm in $T(n, \varepsilon/d)$ time.

Improvements for EBM: For the EBM problem, as mentioned in Theorem 4.2, we obtain an improvement in the approximation ratio as follows: Instead of dividing each cell into a fixed number $\min\{n, (4\sqrt{d}/\varepsilon)^d\}$ of children, we divide them into $\min\{n, n^{d/2^i}\}$ children at each level i; here, level of any cell in the tree is equal to the length of the path from the root to this cell. By doing so, we reduce the height of the tree to $O(\log \log n)$. To analyse the running time, we show that the number of remaining unmatched points over all level i cells is $\tilde{O}(n^{1-\frac{1}{2^i}})$. Since there are only sub-linearly many points remaining, we can afford a larger spread of $O(\frac{d}{\varepsilon} n^{\frac{1}{2^i}})$ and the resulting execution time of the additive approximation will continue to be $O(T(n, \frac{\varepsilon}{d}))$ per level.

4.2.1 An Improved Relative Approximation Algorithm

In Lemma 4.10, we show that any ε-additive approximation algorithm for the 1-Wasserstein problem with a running time of $T(n, \varepsilon)$ can be used to obtain a $(1+\varepsilon)$-relative approximation

algorithm in $T(n, \varepsilon/\Delta)$ time; here, Δ is the spread of the input points. Both of our algorithms in Section 4.4 and Section 4.5 use this lemma to obtain a 2-approximate transport plan for the instance created for each cell. Plugging the existing additive approximation algorithms, such as the algorithm by [74], in Lemma 4.10 results in a quadratic running time for our algorithms for 1-Wasserstein and EBM problems. In this section, using the ideas mentioned in ([3, 72]), we show that one can use an approximate dynamic nearest neighbor data structure (ANN) to improve the running time of our algorithms.

For demand points A and supply points B, let w^* denote the cost of the optimal transport plan on $A \cup B$. Let (α, ε)-*approximate transport plan* on $A \cup B$ be a transport plan σ satisfying $w(\sigma) \leq \alpha w^* + \varepsilon$. Using an identical discussion as in Section 4.3.3, when points have a unit diameter, by setting $\varepsilon = 1/\Delta$, the additive error of the transport plan will be at most w^*, and as a result, any $(\alpha, 1/\Delta)$-approximate transport plan is an $(\alpha + 2)$-relative approximate transport plan. In this section, we modify the algorithm by [74] (we refer to this algorithm by the LMR-Algorithm) to obtain an (α, ε)-approximate transport plan in $\tilde{O}(n\Phi(n, \alpha)/\varepsilon)$ time; here, $\tilde{O}$ hides factors of $\operatorname{poly}\log n$.

Overview of the LMR-Algorithm: For input points $A \cup B$ of size n and diameter C, the LMR-Algorithm computes an ε-close transport plan on $A \cup B$ as follows. Initially, the distances are scaled by $4/\varepsilon$ and rounded down so that the distance between any two points is an integer bounded by $O(C/\varepsilon)$. The LMR-Algorithm maintains a transport plan $\sigma : A \times B \to \mathbb{R}_{\geq 0}$ and a set of integer dual weights $y : A \cup B \to \mathbb{Z}_{\geq 0}$ for the points satisfying the following feasibility conditions.

$$\begin{aligned} y(b) - y(a) &\leq d(a, b) + 1, && \text{if } \sigma(a, b) < \min\{\eta(a), \eta(b)\}, \\ y(b) - y(a) &\geq d(a, b), && \text{if } \sigma(a, b) > 0. \end{aligned} \tag{4.1}$$

The LMR-Algorithm runs in phases, where in each phase, similar to the algorithm by [53], it executes a Hungarian search procedure followed by an execution of the DFS procedure to find a maximal set of vertex-disjoint admissible augmenting paths. The algorithm then augments the transport plan σ using the set of computed augmenting paths. The algorithm guarantees to increase the dual weight of each free point of B at each phase by at least 1. Additionally, each free point of A will have a zero dual weight at each phase. Therefore, after $\frac{4C}{\varepsilon}+2$ phases, no point of B can be a free point, because otherwise, if there exists a free point $b' \in B$ and a free point $a' \in A$, $y(b')-y(a') > d(a',b')+1$, which violates the feasibility conditions. Therefore, after $\frac{4C}{\varepsilon}+2$ phases, the LMR-Algorithm computes a complete ε-close transport plan (see Lemma 2.3 in ([74])).

Implementation of the LMR-Algorithm using the Partial DFS: As discussed by [72], each phase of the Gabow-Tarjan's algorithm can be implemented using an execution of the partial DFS procedure on the admissible graph. In this implementation, in each phase, the algorithm iteratively initiates a partial DFS procedure from each free point b of B and maintains the search path $Q = \langle b = u_0, \ldots, u_k \rangle$. To process the point u_k, if u_k is a free point of A, the algorithm has found an admissible augmenting path Q. In this case, it stops the partial DFS, augments the transport plan along Q as described in Section 2.1 in ([74]), and initiates the next partial DFS. Otherwise, if there is an admissible outgoing edge (u_k, v) from u_k, if v has not been processed yet, the algorithm adds v to the end of Q as u_{k+1} and starts processing v. Finally, if there are no admissible outgoing edges from u_k, the algorithm sets $y(u_k) \leftarrow y(u_k)+1$, marks u_k as processed, remove u_k from Q, and continue the search from u_{k-1}.

Implementation of each phase using Approximate Nearest Neighbor data structure: Next, we discuss how to use an α-approximate dynamic nearest neighbor data struc-

ture (α-ANN) to implement the partial DFS procedures more efficiently while introducing a factor of α in the approximation factor. Note that the dual weight of each point is an integer between 0 and $O(C/\varepsilon)$; i.e, there are only $O(C/\varepsilon)$ unique dual weights. Similar to [3], we define a set of α-feasibility conditions as follows.

$$\begin{aligned} y(b) - y(a) &\leq \alpha d(a,b), && \text{if } \sigma(a,b) < \min\{\eta(a), \eta(b)\}, \\ y(b) - y(a) &\geq d(a,b), && \text{if } \sigma(a,b) > 0. \end{aligned} \tag{4.2}$$

Any complete α-feasible transport plan is an α-approximate transport plan. Next, we describe the modified implementation of the partial DFS. We construct, for any $y' \in [0, \lfloor \frac{4C}{\varepsilon} + 1 \rfloor]$, an α-ANN for the subset of points of A with dual weight y'. Suppose the partial DFS procedure has maintained a path $Q = \langle u_0, \ldots, u_k \rangle$ and is processing u_k. If u_k is a free point of A, then we have found an augmenting path Q. We augment the transport plan along Q and add every point of A in Q to the α-ANN. Otherwise, if $u_k \in A$ is not a free point, then we simply check all the neighbors of u_k to find an admissible outgoing edge. Otherwise, $u_k \in B$. For any $y' \in [0, \lfloor \frac{4C}{\varepsilon} + 1 \rfloor]$, we query the α-ANN corresponding to the points of A with dual weight y' to find an approximate nearest neighbor a of u_k. If $\frac{y(u_k) - y'}{\alpha} \leq d(a, u_k) \leq y(u_k) - y'$ (successful query), we consider (a, u_k) as an admissible edge and add a as u_{k+1} to the search path Q. We remove a from the α-ANN and continue the search. Otherwise, the returned point a has distance $d(a, u_k) > y(u_k) - y'$ (unsuccessful query). If all $O(C/\varepsilon)$ queries are unsuccessful, we set $y(u_k) \leftarrow y(u_k) + 1$, mark u_k as processed, and remove it from Q.

Next, we give a discussion on the running time of this algorithm. While processing any point $b \in B$, for any successful query returning a point a, two cases might happen: (1) the algorithm removes the neighbor point a from Q without finding any augmenting paths, or (2) the algorithm finds an augmenting path. The first case happens once for each point of

A in each phase and requires $O(nC/\varepsilon)$ queries in total. The second case requires at most one query for each edge of each augmenting path, and by Lemma 2.4 in ([74]), requires $O(nC^2/\varepsilon^2)$ queries in total. Furthermore, in each phase, the search from b might result in at most $O(C/\varepsilon)$ unsuccessful queries and $O(nC^2/\varepsilon^2)$ queries in total. Processing all points of A requires searching for admissible edges among the edges with a positive flow, which by Lemma 2.4 in ([74]) is at most $O(nC^2/\varepsilon^2)$ over all phases. Finally, in each iteration, each point undergoes a dual-weight update at most once and there are $O(nC/\varepsilon)$ updates in total. Therefore, if $\Phi(\alpha, n)$ denotes the query/update time of an α-ANN on n points, this algorithm computes an (α, ε)-approximate transport plan in $O(\frac{nC^2\Phi(\alpha,n)}{\varepsilon^2})$ time.

Extensions of our algorithms: In our algorithms, as in Section 4.3.3, we assume $C = 1$. Plugging $\varepsilon = 1/\Delta$, we get the following lemma.

Lemma 4.5. *There exists an algorithm that, given two point sets A and B in the d-dimensional unit hypercube with a spread of Δ, computes an $O(\alpha)$-approximate transport plan in $O(n\Delta^2\Phi(n, \alpha))$ time.*

First, we extend our 1-Wasserstein algorithm from Section 4.4 using the α-ANN data structure. Recall that for each cell $\square$, our algorithm creates instances $\mathcal{I}_\square$ with a spread $\Delta \leq d/\varepsilon$. Recall that $n_\square$ denotes the number of points in the instance $\mathcal{I}_\square$. Using Lemma 4.5, we compute an $O(\alpha)$-approximate transport cost on $\mathcal{I}_\square$ in $O(d^2 n_\square \varepsilon^{-2}\Phi(n, \alpha))$ time. For any level of the hierarchical partitioning, the total size of instances created at all cells of that level is at most n. Therefore, summing over all cells of the tree, we get the following result.

Theorem 4.6. *There exists a randomized algorithm that, given two discrete distributions μ and ν in the d-dimensional unit hypercube with a spread of $\Delta = n^{O(1)}$ and a parameter $\varepsilon > 0$, computes a $O(d\alpha \log_{\sqrt{d}/\varepsilon} n)$-approximate transport plan in $O(d^2 n\Phi(n, \alpha)\varepsilon^{-2} \log_{\sqrt{d}/\varepsilon} n)$ time.*

Next, we discuss the extension of our EBM algorithm. To do so, first, we modify our hierarchical partitioning to obtain T', as follows. Define $\delta' := 10d^2\varepsilon^{-1}\log n$. Similar to Section 4.5, we define the root cell of T' as a randomly-shifted cell $\square^*$ that contains all points of $A \cup B$ and has a side-length of $2\max\{C_{\max}, \ell_1/\varepsilon\}$; here, $\ell_1 = \delta' n$. For any cell $\square$ of T', if $\square$ contains at most $n^{\varepsilon/2}$ points of $A \cup B$, then we designate $\square$ as a leaf cell. Otherwise, let $\square$ be a cell of level i. Define the grid $\mathbb{G}_\square = \mathbb{G}(\square, \delta' n^{(2/3)^i})$ and add the non-empty cells of $\mathbb{G}_\square$ to T' as the children of $\square$. The height of T' is $h = O(\log \frac{d}{\varepsilon})$. Using a similar proof as in Lemma 4.8, we can show that with probability at least $1/2$, no cell of level 1 is a surplus or deficit cell.

For any non-root cell $\square$ of level i of T', we construct an instance $\mathcal{I}_\square$ identical to the one in Section 4.5. The spread of the points in $\mathcal{I}_\square$ is $O(\sqrt{d}\frac{\ell_i}{\ell_{i+1}}) = O(\sqrt{d}n^{\frac{1}{3}\cdot(\frac{2}{3})^i})$. Additionally, by invoking Lemma 4.13 on level i cells, we have $\mathbb{E}[n_i] = O(\frac{\sqrt{d}w^*}{\ell_{i+1}}) = O(n^{1-(\frac{2}{3})^{i+1}})$. Therefore, plugging Lemma 4.5 as the approximate solver of the instances, we get the following theorem.

Theorem 4.7. *There exists an algorithm that, given two sets of samples A and B each from distributions μ and ν, where $|A| = |B| = n$, and a parameter $\varepsilon > 0$, computes, with high probability, an $O(d\alpha \log \frac{d}{\varepsilon})$-approximate matching in $O(dn\Phi(n,\alpha)\log\frac{d}{\varepsilon} + n^{1+\varepsilon})$ time.*

Lemma 4.8. *Let M^* be an optimal matching on the point set $A \cup B$. Then, with probability at least $1 - \varepsilon/\sqrt{d}$, no edge of M^* crosses $\mathbb{G}_1$.*

Proof. Recall that $\ell_1 = 5d^{5/2}n\log n/\varepsilon^2$. For any pair of points $(a,b) \in A \times B$, the probability that the edge (a,b) crosses the grid $\mathbb{G}_1$ is upper-bounded as follows.

$$\Pr((a,b) \text{ crosses } \mathbb{G}_1) \le \frac{\sqrt{d}\|a-b\|}{5d^{5/2}n\log n/\varepsilon^2}. \tag{4.3}$$

Let $X_{(a,b)}$ be an indicator random variable where $X_{(a,b)} = 1$ if (a,b) crosses $\mathbb{G}_1$ and $X_{(a,b)} = 0$ otherwise. Clearly, $\mathbb{E}[X_{(a,b)}] = \Pr((a,b) \text{ crosses } \mathbb{G}_1)$. Define $X = \sum_{(a,b)\in M^*} X_{(a,b)}$ to be the

number of edges of M^* crossing the grid $\mathbb{G}_1$. By linearity of expectation and Equation 4.3,

$$\mathbb{E}[X] = \sum_{(a,b)\in M^*} \mathbb{E}[X_{(a,b)}] \leq \sum_{(a,b)\in M^*} \frac{\sqrt{d}\|a-b\|}{5d^{5/2}n\log n/\varepsilon^2} = \frac{w(M^*)}{5d^2 n\log n/\varepsilon^2} \leq \frac{\varepsilon}{\sqrt{d}},$$

where the last inequality follows from property (T3) in the input transformation. Therefore, with probability at least $1-\varepsilon/\sqrt{d}$, $X=0$ and no edge of M^* crosses the grid $\mathbb{G}_1$. □

4.3 Preliminaries

We begin by introducing notations that are used in presenting our algorithm. For any cell $\square$, we denote the side-length of $\square$ by $\ell_\square$ and the *center* of $\square$ by $c_\square$. For a parameter ℓ, let $\mathbb{G}(\square,\ell)$ denote a grid that partitions $\square$ into smaller cells with side-length ℓ. Recall that $V_\square$ denotes the subset of $A\cup B$ that lies inside $\square$. We say that $\square$ is *non-empty* if $V_\square$ is non-empty; i.e, $\square$ contains at least one point of $A\cup B$. We define the weight of $\square$ to be $\eta(V_\square)$ and denote it by $\eta(\square)$. For each cell $\square$, we call it a *deficit cell* if $\eta(\square)<0$, a *surplus cell* if $\eta(\square)>0$, and a *neutral cell* if $\eta(\square)=0$.

In this section, we provide a simple transformation of the input for achieving an additive approximation of the 1-Wasserstein distance. Furthermore, for point sets with a spread of Δ, we show how an additive approximation algorithm can be used to obtain a $(1+\varepsilon)$-relative approximation of the 1-Wasserstein problem in $T(n,\varepsilon/\Delta)$ time.

4.3.1 Additive Approximation in Euclidean Space

In this section, for any $\varepsilon>0$, given an additive approximation algorithm that runs in $T(n,\varepsilon)$ time, we present an algorithm to compute an ε-close transport cost for distributions inside

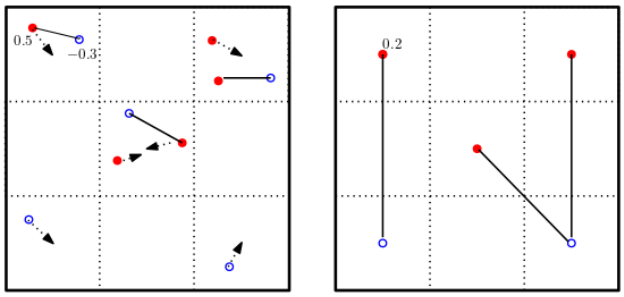


Figure 4.1: (a) The algorithm transports supplies (red disks) to demands (blue circles) within each cell and create an instance by moving any excess supplies or demands to the center of the corresponding cells, (b) An $\varepsilon/2$-close transport plan is computed on the new problem instance.

the d-dimensional unit hypercube $\square^*$ in $O(\min\{T(n, \frac{\varepsilon}{2}), n + T((\frac{2\sqrt{d}}{\varepsilon})^d, \frac{\varepsilon}{2})\})$ time.

The algorithm executes in two steps. In the first step, the algorithm builds a grid $\mathbb{G}(\square^*, \frac{\varepsilon}{2\sqrt{d}})$ on the unit hypercube $\square^*$ and constructs a transport plan σ_1 as follows. For each non-empty neutral cell of the grid, σ_1 arbitrarily transports all supplies to demands within the cell. Similarly, for any deficit (resp. surplus) cell, σ_1 arbitrarily transports supplies from (resp. to) all supply (resp. demand) points inside the cell to (resp. from) some arbitrary demand (resp. supply) points within the cell.

In the second step, the algorithm constructs a set of demand points $\mathcal{A}$ and supply points $\mathcal{B}$ as follows: For any deficit cell $\square$ (resp. surplus cell $\square'$), the point $c_\square$ (resp. $c_{\square'}$) is added to $\mathcal{A}$ (resp. $\mathcal{B}$) with a weight of $\eta(\square)$ (resp. $\eta(\square')$). Note that $\mathcal{A} \cup \mathcal{B}$ is a balanced instance for the 1-Wasserstein problem. The algorithm computes an $\frac{\varepsilon}{2}$-close transport plan σ_2 on the instance $\mathcal{A} \cup \mathcal{B}$ in $T(|\mathcal{A}|+|\mathcal{B}|, \varepsilon/2)$ time (See Figure 4.1). The algorithm returns $w(\sigma_1)+w(\sigma_2)$ as an ε-close transport cost on $A \cup B$.

We provide a discussion on the accuracy of the algorithm in Section 4.3.2.

The following lemma follows from the fact that $|\mathcal{A}| + |\mathcal{B}|$ is bounded by $\min\{2n, (2\sqrt{d}/\varepsilon)^d\}$.

Lemma 4.9. *There exists an algorithm that, given two point sets A and B in the d-dimensional unit hypercube and a value $\varepsilon > 0$, computes an ε-close transport cost in $O(\min\{T(n, \frac{\varepsilon}{2}), n + T((\frac{2\sqrt{d}}{\varepsilon})^d, \frac{\varepsilon}{2})\})$ time.*

Instead of transporting supplies inside cells arbitrarily, our algorithm in Section 4.4 recursively applies the same algorithm in each cell and obtains a higher accuracy.

4.3.2 Quality of the Geometric Additive Approximation Algorithm

In this section, we analyze the additive error of the 1-Wasserstein distance. First, we define some notations that are used in the analysis.

For a transport plan σ, the *residual graph* is a graph $\mathcal{G}(A, B)$ on the point set $A \cup B$ such that for any pair of points $(a, b) \in A \times B$, (1) if $\sigma(a, b) > 0$, there is a *backward edge* from a to b of capacity $\sigma(a, b)$ and (2) if $\sigma(a, b) < \min\{-\eta(a), \eta(b)\}$, then there is a *forward edge* from b to a with a capacity $\min\{-\eta(a), \eta(b)\} - \sigma(a, b)$. An *alternating path* is any simple path P in $\mathcal{G}(A, B)$; i.e, a path P where the edges alternates between forward edges and backward edges. Any demand point $a \in A$ (resp. supply point $b \in B$) is called a *free point* with respect to σ if the total supplies transported into a (resp. from b) in σ is less than $-\eta(a)$ (resp. $\eta(b)$). An *augmenting path* P with respect to σ is simply an alternating path from a free supply point to a free demand point. Let $\beta(P)$ denote the amount by which P increases the total supplies transported by σ.

Recall that the algorithm of Section 4.3.1 computes a transport plan in two steps. In the first step, it computes a transport plan σ_1 by arbitrarily transporting supplies to demands within each non-empty cell of the grid. Suppose U' supplies are transported in this step. In the

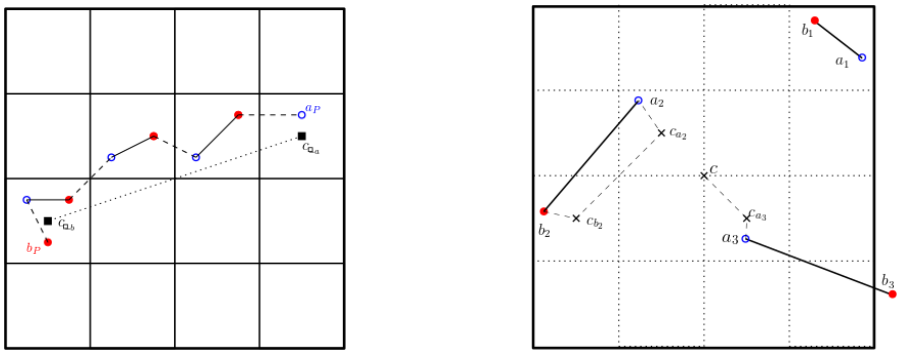
second step, for each surplus (resp. deficit) cell $\square$, the algorithm moves the excess supplies (resp. demands) of the points inside $\square$ to the center point $c_{\square}$ and creates the input instance $\mathcal{A} \cup \mathcal{B}$. Then, it computes an $\varepsilon/2$-close transport plan σ_2 on $\mathcal{A} \cup \mathcal{B}$, which transports $U - U'$ supplies. The algorithm reports $w(\sigma_1) + w(\sigma_2)$ as an approximate 1-Wasserstein distance.

Since the diameter of each cell of the grid is $\varepsilon/2$, each edge carrying a positive supply in σ_1 has a length of at most $\varepsilon/2$; therefore, $w(\sigma_1) \leq \varepsilon U'/2$. Next, we bound the cost of σ_2. Let σ^* be any optimal transport plan on $A \cup B$. Let $\mathcal{P}$ denote the set of augmenting paths in the symmetric difference $\sigma^* \oplus \sigma_1$ with respect to σ_1. For any $P \in \mathcal{P}$, suppose P is an augmenting path with a supply point $b_P \in B$ and a demand point $a_P \in A$ as its endpoints. Note that a_P (resp. b_P) lies inside a deficit (resp. surplus) cell of the grid, since a_P (resp. b_P) is a free point with respect to σ_1.

Consider a transport plan σ_2^* constructed as follows. For each augmenting path $P \in \mathcal{P}$, let $\square_{a_P}$ (resp. $\square_{b_P}$) denote the cell of the grid containing a_P (resp. b_P). The transport plan σ_2^* transports $\beta(P)$ supplies from $c_{\square_{b_P}}$ to $c_{\square_{a_P}}$ (see Figure 4.2(a)). Since the augmenting paths in $\mathcal{P}$ transport all excess supplies from the surplus cells to the deficit cells, σ_2^* is a valid transport plan for $\mathcal{A} \cup \mathcal{B}$. Furthermore, since σ_2 is an $\varepsilon/2$-close transport plan, $w(\sigma_2) \leq w(\sigma_2^*) + \frac{\varepsilon}{2}(U - U')$. Next, we bound $w(\sigma_2^*)$. Since the diameter of each cell is $\varepsilon/2$,

$$w(\sigma_2^*) = \sum_{P \in \mathcal{P}} \beta(P) \| c_{\square_{a_P}} - c_{\square_{b_P}} \| \leq \sum_{P \in \mathcal{P}} \beta(P) (\|a_P - b_P\| + \frac{\varepsilon}{2}).$$

The total supplies transported by σ_2^* is $\sum_{P \in \mathcal{P}} \beta(P) = U - U'$. Furthermore, for each augmenting path P, by the triangle inequality, $\|a_P - b_P\| \leq \sum_{(u,v) \in P} \|u - v\|$. Therefore,



(a) (b)

Figure 4.2: (a) An example of an augmenting path from a point b_P to a point a_P. In σ_2^*, we add a flow from $c_{\square_b}$ to $c_{\square_a}$. (b) An example showing edges of first case (a_1, b_1), second case (a_2, b_2), and third case (a_3, b_3). The budget assigned to them are shown as dashed lines.

$$w(\sigma_2^*) \leq \frac{\varepsilon}{2}(U - U') + \sum_{P \in \mathcal{P}} \sum_{(u,v) \in P} \beta(P)\|u - v\|.$$

Since $\mathcal{P}$ is the set of augmenting paths in $\sigma^* \oplus \sigma_1$, the total cost of edges of the paths in $\mathcal{P}$ is at most $w(\sigma^*) + w(\sigma_1)$. Therefore,

$$w(\sigma_2^*) \leq \frac{\varepsilon}{2}(U - U') + w(\sigma^*) + w(\sigma_1).$$

Combining with $w(\sigma_1) \leq \frac{\varepsilon}{2}U'$ and $w(\sigma_2) \leq \frac{\varepsilon}{2}(U - U') + w(\sigma_2^*)$,

$$w(\sigma_1) + w(\sigma_2) \leq 2w(\sigma_1) + w(\sigma^*) + \varepsilon(U - U') \leq w(\sigma^*) + \varepsilon U.$$

Thus, the reported cost is an ε-close 1-Wasserstein distance.

4.3.3 Relative Approximation for Low Spread Point Sets

In this section, we show that an ε-additive approximation algorithm can be used to obtain a $(1+\varepsilon)$-relative approximation algorithm for the 1-Wasserstein problem in $T(n, \varepsilon/\Delta)$ time; here, Δ is the spread of the points in $A \cup B$. Since relative approximations are scale invariant, without loss of generality, we assume that the input has $C_{\max} = 1$ and $C_{\min} = 1/\Delta$.

Suppose $-\eta(A) = \eta(B) = U$. The minimum distance between any two points $(a, b) \in A \times B$ is $1/\Delta$. Therefore, the cost of transporting a supply of U is at least U/Δ. We obtain a $(1+\varepsilon)$-relative approximate transport plan by simply executing an additive approximation algorithm with an additive error of $U(\varepsilon/\Delta)$ in time $T(n, \varepsilon/\Delta)$.

Lemma 4.10. *There exists an algorithm that, given two point sets A and B in the d-dimensional unit hypercube with a spread of Δ, computes a $(1+\varepsilon)$-approximate transport plan in $T(n, \varepsilon/\Delta)$ time.*

4.4 An $O(d \log_{\sqrt{d}/\varepsilon} n)$-approximation algorithm for 1-Wasserstein problem

In this section, we present an algorithm satisfying the bounds claimed in Theorem 4.1. We begin by defining a hierarchical partitioning and a tree T associated with it. Each node in the tree corresponds to a non-empty cell in our hierarchical partition and we do not distinguish between the two. Our tree partitions each cell of side-length ℓ into $\lceil 4\sqrt{d}/\varepsilon \rceil^d$ many cells of side-length at most $(\varepsilon/4\sqrt{d})\ell$. We construct our hierarchical partition in a randomly-shifted fashion as follows.

Hierarchical Partitioning: First, we pick a point ξ uniformly at random from the unit hypercube $[0,1]^d$ and set $\square^* = [-1,1]^d + \xi$. Note that $\square^*$ is a hypercube of side-length 2 containing all points in $A \cup B$. We designate $\square^*$ as the root of T. Let $\kappa = \lceil 4\sqrt{d}/\varepsilon \rceil$. For any cell $\square$, if only one point of $A \cup B$ lies inside $\square$, we designate $\square$ as a leaf cell in T. For any non-leaf cell $\square$, we construct its children by partitioning $\square$ using a grid $\mathbb{G}_\square = \mathbb{G}(\square, \ell_\square/\kappa)$ into $\lceil 4\sqrt{d}/\varepsilon \rceil^d$ cells and create a child node for each non-empty cell of this grid. For any non-leaf cell $\square$ of T, we denote the set of children of $\square$ by $\mathsf{C}[\square]$. Assuming the spread $\Delta = n^{O(1)}$, the height of T, h, is $O(\log_{\sqrt{d}/\varepsilon} n)$.

Similar to quadtrees, our hierarchical partitioning can be seen as a sequence of grids $\langle \mathbb{G}_0, \mathbb{G}_1, \ldots, \mathbb{G}_h \rangle$, where $\mathbb{G}_0$ is the root and each grid $\mathbb{G}_i$ refines the cells of grid $\mathbb{G}_{i-1}$. For each grid $\mathbb{G}_i$, we denote the cell-side-length of $\mathbb{G}_i$ by ℓ_i. Next, we describe our algorithm.

Computing an approximate 1-Wasserstein distance: In this section, we present our algorithm for computing an approximate cost of an optimal transport plan on $A \cup B$.

1) Creating an instance at each cell: For each cell $\square$ of T, we create an instance $\mathcal{I}_\square$ of the 1-Wasserstein problem as follows. If $\square$ is a deficit or surplus cell, we add the point $c_\square$ with a weight $-\eta(\square)$ to $\mathcal{I}_\square$. Additionally, if $\square$ is a leaf cell, then it contains a single point u of $A \cup B$, which we add to $\mathcal{I}_\square$. Otherwise, $\square$ is a non-leaf cell and for any child $\square' \in \mathsf{C}[\square]$, if $\square'$ is a deficit or surplus cell, we add $c_{\square'}$ with a weight $\eta(\square')$ to $\mathcal{I}_\square$, which represents the excess demands or supplies of the points in $V_{\square'}$. The instance $\mathcal{I}_\square$ is balanced and has a spread of $O(d/\varepsilon)$.

2) Estimating the 1-Wasserstein distance: In this step, for each cell $\square$, using the algorithm from Lemma 4.10, we compute a 2-approximate transport plan $\sigma_\square$ on $\mathcal{I}_\square$. Our algorithm then reports the total cost of the transport plans computed at all cells of T, i.e, $w := \sum_{\square \in T} w(\sigma_\square)$, as an approximate 1-Wasserstein distance.

Retrieving an approximate transport plan: We retrieve an approximate transport plan σ on the point set $A \cup B$ by processing grids $\langle \mathbb{G}_h, \ldots, \mathbb{G}_0 \rangle$ in decreasing order of their level. First, for each non-empty cell $\square$ of $\mathbb{G}_h$, $\square$ is a leaf cell and $V_\square$ contains only one point. We map the only point in $V_\square$ to $c_\square$. For some $i < h$, assume (inductively) that after processing the non-empty cells of the grid $\mathbb{G}_{i+1}$, the following conditions (i)–(iii) hold for the current transport plan σ within any cell $\square$ in $\mathbb{G}_{i+1}$: (i) if $\square$ is a neutral cell, then σ transports every supply to some demand point inside $\square$, (ii) if $\square$ is a deficit (resp. surplus) cell, then σ transports all supplies (resp. demands) inside $\square$ to (resp. from) some demand (resp. supply) point within $\square$, and, (iii) if $\square$ is a deficit (resp. surplus) cell, the excess demand (resp. supply) is mapped to $c_\square$. Given this, we show how to process any non-empty cell $\square$ of $\mathbb{G}_i$ so that (i)–(iii) holds for $\square$.

Recollect that $\sigma_\square$ is a transport plan computed by our algorithm on $\mathcal{I}_\square$. By condition (iii), the excess supplies or demands at any child $\square'$ of $\square$ is mapped to $c_{\square'}$. Therefore, for any pair of children $\square_1, \square_2 \in \mathsf{C}[\square]$, where $\square_1$ is a surplus cell and $\square_2$ is a deficit cell, the transport plan σ transports $\sigma_\square(c_{\square_1}, c_{\square_2})$ supplies from $c_{\square_1}$ to $c_{\square_2}$. In addition, for any child $\square_1$ (resp. $\square_2$) of $\square$, suppose $\square_1$ (resp. $\square_2$) is a surplus (resp. deficit) cell. If $\sigma_\square(c_{\square_1}, c_\square) > 0$ (resp. $\sigma_\square(c_{\square_2}, c_\square) > 0$), then we map the supplies (resp. demands) from $c_{\square_1}$ (resp. $c_{\square_2}$) to $c_\square$. It is easy to confirm that after processing $\square$, (i)–(iii) holds for $\square$. From triangle inequality, $w(\sigma)$ is upper-bounded by the total cost of the transport plans computed for each cell of T; i.e, $w(\sigma) \leq \sum_{\square \in T} w(\sigma_\square)$.

Efficiency: For any i, let C_i denote the set of non-empty cells of T at level i. For each cell $\square \in \mathsf{C}_i$, let $n_\square$ be the number of points in $\mathcal{I}_\square$. Since the spread of the points in $\mathcal{I}_\square$ is $O(d/\varepsilon)$, executing the algorithm from Lemma 4.10 on $\mathcal{I}_\square$ takes $O(T(n_\square, \varepsilon/d))$ time. Since $\mathcal{I}_\square$ contains at most one point for each non-empty child of $\square$, $n_\square \leq \min\{|V_\square|, (4\sqrt{d}/\varepsilon)^d\}$. Therefore, $\sum_{\square \in \mathsf{C}_i} n_\square \leq \sum_{\square \in \mathsf{C}_i} |V_\square| = n$. Since $T(n, \varepsilon) = \Omega(n)$, the running time of our

algorithm on cells at level i is $O(\sum_{\square \in \mathsf{C}_i} T(n_\square, \varepsilon/d)) = O(T(n, \varepsilon/d))$. Summing over all levels, the running time of our algorithm is $O(T(n, \varepsilon/d) \log_{\sqrt{d}/\varepsilon} n)$.

When the dimension is a small constant, we get an improved running time as follows. For each level i of T, there are at most n non-empty cells at level i and the instance created at each cell has a size of at most $(4\sqrt{d}/\varepsilon)^d$. Therefore, $O(\sum_{\square \in \mathsf{C}_i} T(n_\square, \varepsilon/d)) = O(nT((4\sqrt{d}/\varepsilon)^d, \varepsilon/d)) = n(\frac{d}{\varepsilon})^{O(d)}$ and the overall running time will be improved to $n(\frac{d}{\varepsilon})^{O(d)} \log_{\sqrt{d}/\varepsilon} n$.

In the next section, assuming $\varepsilon \leq 1/2$, we show that the cost w computed by the algorithm is an $O(d \log_{\sqrt{d}/\varepsilon} n)$-approximation of the cost of the optimal transport on $A \cup B$.

Quality of Approximation: In this section, we analyze the approximate 1-Wasserstein distance computed by our algorithm. After scaling the points to have a unit diameter, the algorithm of Lemma 4.10 obtains a 2-approximate transport plan by computing a $(1/\Delta)$-close transport plan. When scaled back to original, for the root cell $\square^*$ of T, $\sigma_{\square^*}$ is an $(\varepsilon/4\sqrt{d})$-close transport plan on $\square^*$. The remaining demands and supplies are recursively transported within the children. Since the diameter of each child of $\square^*$ is $\varepsilon/2$, similar to Section 4.3.1, we can argue that our algorithm reports an ε-close 1-Wasserstein distance. Next, we show that the reported cost is an $O(d \log_{\sqrt{d}/\varepsilon} n)$-approximation of the 1-Wasserstein distance.

Recall that for each level i, C_i denotes the set of non-empty cells of T at level i. We show that the expected distortion of mapping all points to the centers of the leaf cells is $O(d \log_{\sqrt{d}/\varepsilon} n) w(\sigma^*)$, where the expectation is over the choice of the random shift of the hierarchical partitioning. Additionally, for each $i < h$, we show that $\mathbb{E}[\sum_{\square \in \mathsf{C}_i} w(\sigma_\square)] = O(d) w(\sigma^*)$; here, σ^* is an optimal transport plan on $A \cup B$. The result then follows since T has $O(\log_{\sqrt{d}/\varepsilon} n)$ levels.

For any level $i < h$, we bound $\mathbb{E}[\sum_{\square \in \mathsf{C}_i} w(\sigma_\square)]$ in two steps as follows. In the first step, we assign a budget to every edge (a, b) with $\sigma^*(a, b) > 0$ and show that the total budget

assigned to all such edges is (in expectation) at most $O(d)w(\sigma^*)$. In the second step, we redistribute this budget to the cells of level i in a way that the budget received by any cell $\square$ is at least $w(\sigma_\square)/2$. Theorem 4.1 follows from combining the two steps.

4.5 An $O(d \log\log n)$-approximation algorithm for EBM problem

Given two sets of samples A and B from distributions μ and ν, respectively, where $|A| = |B| = n$ and each point $a \in A$ (resp. $b \in B$) has a weight $\eta(a) = -1/n$ (resp. $\eta(b) = 1/n$), we present an approximate algorithm for the EBM problem satisfying the bounds claimed in Theorem 4.2. For simplicity in presentation, we scale all the $\eta(\cdot)$ values from $-1/n$ (resp. $1/n$) to -1 (resp. 1) and show that our algorithm computes a matching that is within an additive error of $n \min\{\varepsilon, (d \log\log n)(w(M^*)/n)\}$ from the optimal matching cost $w(M^*)$.

Input Transformation: Let $A' \cup B'$ be the input point set. We transform $A' \cup B'$ into another point set $A \cup B$ such that

(T1) The coordinates of the points in $A \cup B$ are positive integers bounded by $n^{O(1)}$,

(T2) Any optimal matching with respect to $A \cup B$ is a $(1+\varepsilon)$ approximation for $A' \cup B'$, and,

(T3) The cost of the optimal matching of A and B is at least $5\sqrt{d}n/\varepsilon$ and at most $5d^{3/2}n \log n/\varepsilon$.

Similar transformations have been applied in several papers in the literature ([7, 11, 73]). The details of this transformation are as follows. Suppose the hierarchical greedy paradigm

mentioned in 1, when applied to the input points $A' \cup B'$, returns a value θ which is, in expectation, an $O(d \log n)$-approximation of the cost of the optimal matching on $A' \cup B'$. Using θ, we can transform any point $p' = (p_{x_1}, \ldots, p_{x_d}) \in A' \cup B'$ into another point $p = (\lceil \frac{5d^{3/2} n \log n}{\varepsilon \theta} p_{x_1} \rceil, \ldots, \lceil \frac{5d^{3/2} n \log n}{\varepsilon \theta} p_{x_d} \rceil)$ to obtain a set of points $A \cup B$. If any pair of points $(a, b) \in A \times B$ map to the same location, we match the two and remove them from $A \cup B$. The resulting point set satisfies (T1)–(T3).

As before, we can match and remove any co-located points $a \in A$ and $b \in B$.

Next, assuming $T(n, \varepsilon) = O(\frac{n^k}{\varepsilon} \text{poly}\{d, \log n, \log \frac{1}{\varepsilon}\})$ for some $k \geq 1$, we describe our EBM algorithm. Our algorithm is easily adaptable to any additive approximate algorithm with a running time of $T(n, \varepsilon) = O(n^k \varepsilon^{-t} \text{poly}\{d, \log n, \log \frac{1}{\varepsilon}\})$, where $k \geq 1$ and t is a fixed constant.

Overview of the algorithm: Similar to Section 4.4, our EBM algorithm constructs a hierarchical partitioning and the associated tree T, and executes the algorithm from Lemma 4.10 on the instance created for each cell of T. In contrast to Section 4.4, the tree T constructed by our EBM algorithm has a lower height of $O(\log \log n)$, resulting in an improved approximation factor. The hierarchical partitioning of this section differs from the one in Section 4.4 in two ways. First, we partition the root cell into a grid $\mathbb{G}_1$ with cell-side-length of $\Theta(d^{5/2} n \log n / \varepsilon^2)$ as the first level. The grid $\mathbb{G}_1$ may result in a high branching factor at the root; however, we show that, with probability at least $1 - \varepsilon/\sqrt{d}$, no edges of an optimal matching will cross $\mathbb{G}_1$. Therefore, with that probability, all cells of $\mathbb{G}_1$ are neutral cells and the problem instance for the root is an empty instance; i.e, the branching factor of the root will not impact the running time of our algorithm. Second, for any cell $\square$ of level i, instead of splitting $\square$ into a fixed number $\min\{n, (4\sqrt{d}/\varepsilon)^d\}$ of children, we divide $\square$ into $\min\{n, n^{d/2^i}\}$ children. Although this results in a higher spread of $\tilde{O}(n^{1/2^i})$ for the problem instance $\mathcal{I}_\square$, we show that the expected number of remaining unmatched points over all cells of level i is

$\tilde{O}(n^{1-1/2^i})$. Therefore, the total execution time of our algorithm remains $T(n, \varepsilon/d)$ per level. These modifications result in a tree T of height $O(\log \log n)$. We describe the details below.

Hierarchical Partitioning: Define $\delta := 5d^2\varepsilon^{-1} \log n$. Similar to Section 4.4, we define a cell $\square^*$ as a randomly-shifted hypercube that contains all points of $A \cup B$ and has a side-length of $2 \max\{C_{\max}, \ell_1/\varepsilon\}$, where $\ell_1 = \frac{\sqrt{d}}{\varepsilon}\delta n$. We designate $\square^*$ as the root of T ($\square^*$ is at level 0 of T). Define a grid $\mathbb{G}_1 := \mathbb{G}(\square^*, \ell_1)$. We add each non-empty cell of $\mathbb{G}_1$ to the tree as the children of $\square^*$. We construct the hierarchical partitioning in a recursive fashion as follows. For any non-root cell $\square$ of T, if $\square$ contains only one point of $A \cup B$, then we designate $\square$ as a leaf cell. Otherwise, let $\square$ be a cell of level i. Define the grid $\mathbb{G}_{\square} = \mathbb{G}(\square, \delta n^{1/2^i})$[2] and add the non-empty cells of $\mathbb{G}_{\square}$ to T as the children of $\square$. For any cell $\square$, denote the set of children of $\square$ in T by $\mathsf{C}[\square]$. The height of T, denoted by h, is $O(\log \log n)$.

Similar to Section 4.4, our hierarchical partitioning is also a sequence of grids $\langle \mathbb{G}_0, \mathbb{G}_1, \ldots, \mathbb{G}_h \rangle$, where $\mathbb{G}_0$ is the root cell $\square^*$, $\mathbb{G}_1$ has a cell-side-length of $\ell_1 = \frac{\sqrt{d}}{\varepsilon}\delta n$, and for each $2 \le i \le h$, the cell-side-length of $\mathbb{G}_i$ is $\ell_i = \delta n^{1/2^{i-1}}$.

Computing an approximate matching cost: To estimate the matching cost, similar to Section 4.4, our algorithm creates an instance of the 1-Wasserstein problem for each cell of the tree T. Using the algorithm from Lemma 4.10, our algorithm computes a 2-approximate transport plan for the instance created for each cell and returns the total cost of such transport plans as an approximate matching cost. This completes the description of the algorithm.

We describe the details of retrieving a matching in Section 4.5.1, the quality of approximation in Section 4.5.2, and the efficiency of our algorithm in Section 4.5.3.

[2]For simplicity in presentation, we assume $n^{1/2^i}$ is an integer.

4.5.1 Retrieving an approximate bipartite matching

In this section, we show how to retrieve a matching M on the point set $A \cup B$ using an identical approach as described in Section 4.4. We process the grids $\langle \mathbb{G}_h, \dots, \mathbb{G}_1 \rangle$ in the decreasing order of their level. For any cell $\square$ of $\mathbb{G}_h$, we map the unique point in $V_\square$ to the center point $c_\square$. Inductively, assume that for a level $i < h$, after processing all non-empty cells of $\mathbb{G}_{i+1}$, conditions (i)–(iii) defined in Section 4.4 hold for each cell of that grid. For each non-empty cell $\square$ of $\mathbb{G}_i$, we show how to process $\square$ so that conditions (i)–(iii) hold for $\square$ as well. For any two children $\square_1, \square_2 \in \mathsf{C}[\square]$, we match $\sigma_\square(c_{\square_1}, c_{\square_2})$ many unmatched points that are mapped to $c_{\square_1}$ to the unmatched points that are mapped to $c_{\square_2}$. Additionally, for any child $\square_1 \in \mathsf{C}[\square]$ with $\sigma_\square(c_{\square_1}, c_\square) > 0$, we map $\sigma_\square(c_{\square_1}, c_\square)$ unmatched points from $c_{\square_1}$ to $c_\square$.

4.5.2 Quality of Approximation

In this section, we show that the cost returned by the algorithm is within an additive error of $\min\{\varepsilon, (d \log\log n) w(M^*)\}$ from the optimal matching cost. Consider any cell $\square$ of $\mathbb{G}_1$. Our algorithm first computes some matching $M_1^\square$ by matching the points within each child of $\square$. Then, it moves the unmatched points to the centers of the children of $\square$ and compute a matching $M_2^\square$ on this instance. Let $w_\square$ be the cost of the matching computed inside $\square$ after transforming the points back to the original ones. To show that the algorithm reports a ε-close matching cost, we show that for each non-empty cell $\square$ of $\mathbb{G}_1$, $w(M_1^\square) + w(M_2^\square)$ is within an additive error of $|V_\square| \ell_1 \varepsilon$ from the optimal matching cost on $V_\square$. Therefore, when transformed back to the original point sets in the unit hypercube, the additive error would be at most $|V_\square|\varepsilon$, as desired.

We use an identical discussion as Section 4.3.2. Let $m_1^\square$ be the size of the matching $M_1^\square$.

Since the diameter of each child of $\square$ is $\sqrt{d}\ell_2$, $w(M_1^{\square}) \leq m_1^{\square}\sqrt{d}\ell_2$. Furthermore, since the spread of the points in the instance created in the second step, as described above, is $\sqrt{d}\ell_1/\ell_2$, the matching $M_2^{\square}$ is obtained by computing a $\frac{1}{\sqrt{d}\ell_1/\ell_2}$-close transport plan on points that are scaled by a factor of at most $\frac{1}{\sqrt{d}}$, which adds an additive error of $(|V_{\square}| - m_1^{\square})\frac{\ell_1}{\ell_1/\ell_2}$. Finally, the cost of moving the unmatched points inside the children of $\square$ to their centers is at most $(|V_{\square}| - m_1^{\square})\sqrt{d}\ell_2$. Since $\ell_1 = \frac{\sqrt{d}}{\varepsilon}\delta n$ and $\ell_2 = \delta\sqrt{n}$, and considering the discussion in Section 4.3.2, the additive error of our algorithm would be at most

$$2m_1^{\square}\sqrt{d}\ell_2 + (|V_{\square}| - m_1^{\square})\ell_2 + (|V_{\square}| - m_1^{\square})\sqrt{d}\ell_2 \leq |V_{\square}|\ell_1\varepsilon.$$

Therefore, it suffices to show that the matching M is an $O(d \log\log n)$-approximate matching. To do so, using an argument identical to Section 4.4, we can show that the total cost of the transport plans computed at all cells of level i (in expectation) is at most $O(d)w(M^*)$.

Lemma 4.11. $\mathbb{E}[\sum_{\square\in\mathsf{C}_i} w(\sigma_{\square})] \leq O(d)w(M^*)$.

Summing over all levels of T, we get the following corollary.

Corollary 4.12. $\mathbb{E}[w(M)] \leq \sum_{i=1}^{h-1} \mathbb{E}[\sum_{\square\in\mathsf{C}_i} w(\sigma_{\square})] \leq O(d \log\log n)w(M^*)$.

4.5.3 Efficiency of our Algorithm

To analyze the running time of our algorithm, we first show that, with probability at least $1 - \frac{\varepsilon}{\sqrt{d}}$, no edges of an optimal matching will cross $\mathbb{G}_1$ and consequently, each child of $\square^*$ contains an equal number of points of A and B. In other words, all children of $\square^*$ are neutral cells and the problem instance $\mathcal{I}_{\square^*}$ is an empty instance.

Therefore, by constructing $O(\log_{\sqrt{d}/\varepsilon} n)$ randomly-shifted hierarchical partitions, with high probability, in at least one partition, all children of the root are neutral cells. Assuming

such partition, we show that for each $0 < i \leq h$, the algorithm requires $O(T(n, \varepsilon/d))$ time to process all cells at level i. Summing over all levels, the total running time of our algorithm is $O(T(n, \varepsilon/d) \log \log n)$.

Recall that C_i denotes the set of cells of T at level i. For any cell $\square$, recall that $n_\square$ denotes the number of points in $\mathcal{I}_\square$. Define $n_i = \sum_{\square \in \mathsf{C}_i} n_\square$. For any cell $\square \in \mathsf{C}_i$, the spread of the points in $\mathcal{I}_\square$ is $O(\frac{d}{\varepsilon} n^{1/2^i})$. Therefore, since $T(n, \varepsilon) = \tilde{O}(n^k/\varepsilon)$ with $k \geq 1$, the execution time of our algorithm over all cells of level i is $\sum_{\square \in \mathsf{C}_i} T(n_\square, \frac{\varepsilon}{dn^{1/2^i}}) \leq T(n_i, \frac{\varepsilon}{dn^{1/2^i}})$. Additionally, since $n_i \leq n$, we have $\mathbb{E}[n_i^k] \leq n^{k-1}\mathbb{E}[n_i] = \left(\frac{\mathbb{E}[n_i]}{n}\right) n^k$. Therefore,

$$\sum_{\square \in \mathsf{C}_i} T(n_\square, \frac{\varepsilon}{dn^{1/2^i}}) \leq T\left(n_i, \frac{\varepsilon}{dn^{1/2^i}}\right) \leq \frac{n^{1/2^i}\mathbb{E}[n_i]}{n} T\left(n, \frac{\varepsilon}{d}\right). \tag{4.4}$$

Next, we bound $\mathbb{E}[n_i]$. For any cell $\square$ and any child $\square' \in \mathsf{C}[\square]$, the weight of $c_{\square'}$ in $\mathcal{I}_\square$ is the minimum number of matching edges connecting a point inside $\square'$ to a point out of it in any maximum matching. Summing over all cells of level i, n_i is upper-bounded by the number of edge of M^* crossing the grid $\mathbb{G}_{i+1}$. The following lemma bounds this number as a function of $w(M^*)$ and ℓ_{i+1}.

Lemma 4.13. *For any $0 < j < h$, the expected number of matching edges in M^* crossing $\mathbb{G}_j$ is at most $\sqrt{d}w(M^*)/\ell_j$.*

Proof. For any edge $(a, b) \in M^*$, the probability that the edge (a, b) crosses the grid $\mathbb{G}_j$ is $\Pr((a, b) \text{ crosses } \mathbb{G}_j) \leq \frac{\sqrt{d}\|a-b\|}{\ell_j}$. Let $X_j(a, b)$ be an indicator random variable such that $X_j(a, b) = 1$ if (a, b) crosses $\mathbb{G}_j$ and $X_j(a, b) = 0$ otherwise. Thus, $\mathbb{E}[X_j(a, b)] \leq \frac{\sqrt{d}\|a-b\|}{\ell_j}$. Define X_j to be a random variable indicating the number of edges of M^* crossing $\mathbb{G}_j$; i.e,

$X_j = \sum_{(a,b)\in M^*} X_j(a,b)$. Using the linearity of expectation, the expected value of X_j is

$$\mathbb{E}[X_j] = \sum_{(a,b)\in M^*} \mathbb{E}[X_j(a,b)] \leq \frac{\sqrt{d}w(M^*)}{\ell_j}.$$

□

By invoking Lemma 4.13 with $j = i+1$, we get $\mathbb{E}[n_i] \leq \sqrt{d}w(M^*)/\ell_{i+1}$. By definition, $\ell_{i+1} = \delta n^{1/2^i}$ and from property (3) of the input transformation, $w(M^*) \leq 5d^{3/2}n\log n/\varepsilon$. Plugging into Equation 4.4, the time taken to process all cells of level i is

$$\frac{\mathbb{E}[n_i]}{n^{1-\frac{1}{2^i}}} T\left(n, \frac{\varepsilon}{d}\right) \leq \frac{\sqrt{d}w(M^*)}{\ell_{i+1}n^{1-\frac{1}{2^i}}} T\left(n, \frac{\varepsilon}{d}\right) \leq T\left(n, \frac{\varepsilon}{d}\right),$$

as desired.

4.6 Experiments

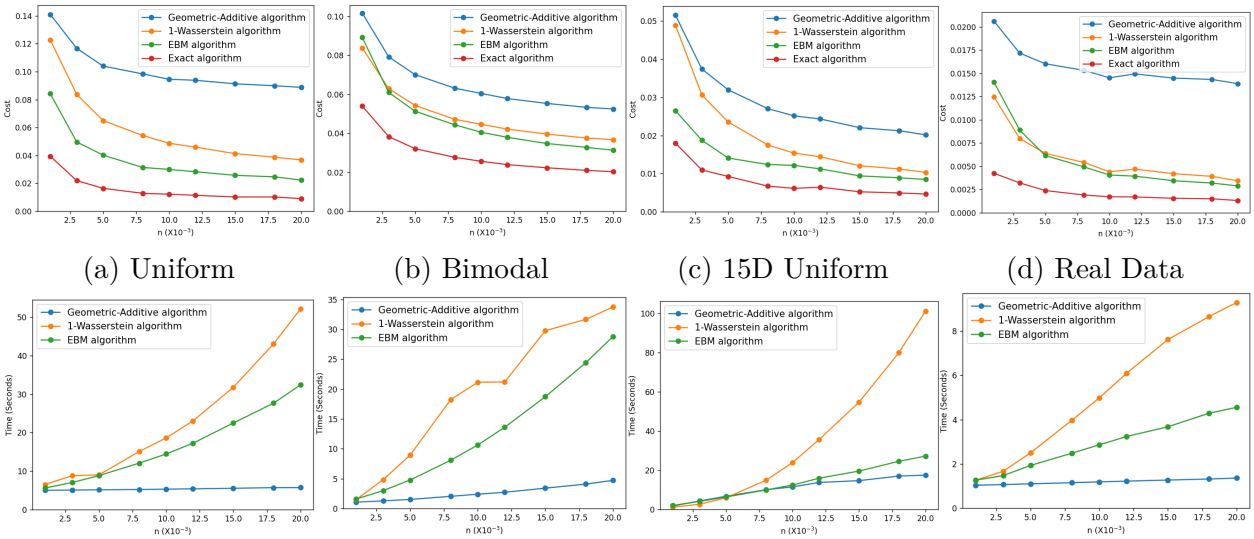
In this section, we conduct experiments to show that our algorithms from Section 4.4 and Section 4.5 improve the accuracy of the additive approximation algorithms with an increase in execution time. We test an implementation of our algorithm, written in Python, on discrete probability distributions derived from real-world and synthetic data sets. All tests are executed on a computer with a 2.50 GHz Intel Core i7 processor and 8GB of RAM using a single computation thread.

Datasets: We test our algorithms on n samples taken from synthetic (distribution) and real data sets. In all our experiments, the sample size n varies from 1000 to 20000. For each data set, we present our results averaged over 10 executions. To generate the synthetic data,

we sample from (i) a uniform distribution inside a unit square (Uniform), (ii) 3-dimensional Gaussian mixture with 2 mean points centered at $(0.24, 0.24, 0.24)$ and $(0.72, 0.72, 0.72)$ inside a unit cube (Bimodal), and (iii) a uniform distribution from inside a 2-dimensional unit square placed on a random plane in 15-dimensional space (15-D Uniform). For a real dataset, we use the *Adult Census Data* (UCI repository) which is a point cloud in $\mathcal{R}^6$ with continuous features for $35,000$ individuals, divided into two categories by income [46].

Implementation Details: We compare the quality of the approximation as well as the execution time of our higher precision algorithms from Section 4.4 (1-Wasserstein algorithm) and Section 4.5 (EBM algorithm) to the additive approximation algorithm from Section 4.3.1 (Geometric-Additive algorithm). Each of these three algorithms uses an additive approximation algorithm with an execution time of $T(n, \varepsilon)$ as a black box. We use the Sinkhorn method for this purpose.

Results: Figure 4.3 illustrates the estimated 1-Wasserstein distance and the execution times resulting from applying the algorithms to the four data sets. We summarize our findings. First, we observe that our algorithms produce more accurate results than the Geometric-Additive algorithm in all of our experiments (see Figure 4.3a, 4.3b, 4.3c and 4.3d). This improvement is more significant on the real dataset while the execution time of the algorithms only increased slightly (see Figure 4.3h). Second, we observe that as the sample size increases, the costs returned by our algorithms converge to the optimal cost, whereas the Geometric-Additive approach does not. In each case, both samples are drawn from an identical distribution, and as a result, the Wasserstein distance between them approaches zero as the sample size increases. Since our algorithm provides a relative approximation, we converge toward 0, whereas the Geometric-Additive does not.



(a) Uniform (b) Bimodal (c) 15D Uniform (d) Real Data

(e) Uniform-Time (f) Bimodal-Time (g) 15D Uniform-Time (h) Real Data-Time

Figure 4.3: (a)–(d) estimated 1-Wasserstein distance, and (e)–(h) execution times of the algorithms

We have now completed the description of our results related to EBM problem. We will now describe our result related to the k-server problem in Chapter 5.

Chapter 5

A scalable work function algorithm for k-Server Problem

This chapter presents a new implementation of the classical work function algorithm for the k-server problem.

For every server $s \in \mathcal{C}^{i-1}$, and $\mathcal{C} = \mathcal{C}^{i-1} \setminus \{s\} \cup \{r_i\}$, our algorithm computes equation 1.2 explicitly and then computes the minimum across all k choices of s. We show that the symmetric difference between $\sigma = \sigma^*_{i-1}(\mathcal{C}^{i-1})$ and $\sigma' = \sigma^*_i(\mathcal{C})$ is a trail[1] T whose edges alternate between those in σ and σ'. We refer to this as an *augmenting trail* and define its net-cost to be $w(\sigma') - w(\sigma)$. To find $w(\sigma')$, we must identify the minimum net-cost augmenting trail that starts at r_i and ends at s. Our augmenting trails can be seen as a variant of the up-down cycles maintained by Rudec and Manger [91]. However, instead of conducting an exhaustive search, we describe an efficient algorithm (similar to the Kuhn-Munkres algorithm [71]) to find this minimum net-cost augmenting trail.

Using a graph search algorithm to find a minimum net-cost augmenting trail in the residual graph can be difficult since these algorithms find simple paths and not trails. To assist in the search for an augmenting trail, we define a weighted graph called the *alternating graph*. Any augmenting trail in the residual graph maps to a directed path in the alternating graph, and every directed path in the alternating graph corresponds to an alternating trail in the

[1]Recollect that a trail is a (possibly non-simple) path that does not repeat edges

residual graph (See Lemma 5.2 and Figure 5.1).

Critically, we also store a set of weights on the vertices of the alternating graph. These weights satisfy a set of feasibility constraints, one for each edge in the alternating graph. Vertex weights have been used to speed-up computation for a shortest path in a graph with negative edge weights, for instance, in Johnson's algorithm [64]. These weights allow us to reduce the problem of finding minimum net-cost augmenting trail from r_i to every server $s \in \mathcal{C}^{i-1}$ to a single execution of Dijkstra's search procedure. Consequently, one can find the optimal choice in equation 1.2 in $O((i+k)^2)$ time. After identifying the optimal choice, we augment the solution to a solution that serves request r_i. This may create many new edges, delete existing edges and also change the cost of some edges in the alternating graph. Somewhat surprisingly, despite the many updates to the alternating graph, we show that the vertex weights maintained by our algorithm continue to satisfy the feasibility constraints for all edges.

5.1 Preliminaries

Recollect that a valid solution is provided by a sequence of configurations $\sigma = \langle \mathcal{C}^0, \mathcal{C}^1, \ldots, \mathcal{C}^n, \mathcal{C}^{n+1} \rangle$. However, lazy valid solutions can also be represented as a set of k paths $\{\Gamma_1, \ldots, \Gamma_k\}$ taken by each of the k servers. More precisely, for any $1 \leq i \leq n$ and $1 \leq j \leq k$, these paths satisfy the following properties:

(P1) For each server s_j, its path Γ_j starts at the location of s_j in the initial configuration. After the first vertex, Γ_j consists of a sequence of requests served by s_j in increasing order of their arrival time. Finally, the last vertex of Γ_j is the location of s_j in the final configuration.

(P2) Every request in R_i participates in exactly one of the k paths.

Furthermore, it can be shown that any set of paths $\{\Gamma_1, \ldots, \Gamma_k\}$ that satisfies (P1) and (P2) will be a valid solution .

The work function algorithm is a lazy algorithm. Therefore, we can represent the solution it produces as k paths satisfying (P1) and (P2). Next, we introduce the notations that are needed to describe an efficient implementation of the work function algorithm.

Notations: Throughout the rest of our discussion, we consider directed graphs. We assume that any edge (u, v) is directed from u to v unless otherwise stated. Let σ_i be a valid solution to the first i requests of the k-server problem. Let $\{\Gamma_1, \Gamma_2, \ldots, \Gamma_k\}$ be the set of k paths satisfying (P1) and (P2). For $1 \leq j \leq k$, and for any vertex v on a path Γ_j where v is not an anchor node, let $f(v)$ denote the location of the next vertex on the path. Similarly, for any vertex v on a path Γ_j where v is not a location in the initial configuration, let $p(v)$ denote the vertex that precedes v in the path Γ_j. In our algorithm, for any given valid solution σ_i, we create a directed graph G_i called the *residual graph* as follows. The vertex set V_i of G_i contains all vertices participating in any of the k paths, i.e., $V_i = \bigcup_{l=1}^{k} \left(\bigcup_{v \in \Gamma_l} v \right)$. There are two types of edges in the edge set E_i of G_i

- *Forward edges:* For each of the k paths and any vertex $v \in V$ where v is not an anchor node, we add a directed edge from v to $f(v)$ denoting that the server at v moves to $f(v)$.
- *Backward edges:* For every request r_j, we add a backward edge to $r_{j'}$ provided $j' < j$ and $j \neq f(j')$. This edge is directed from r_j to $r_{j'}$. We also add a backward edge directed from r_j to the k vertices of the initial configuration.

We refer to the residual graph with respect to σ_i as G_i. The k paths $\{\Gamma_1, \ldots, \Gamma_k\}$ are

represented as k directed paths consisting of all the forward edges in G_i. The backward edges, on the other hand, are not in the solution.

The set of all forward edges of a residual graph G_i corresponds to a valid solution if and only if

(Q1) The forward edges are directed from an earlier request to a later request, and,

(Q2) Every request r has exactly one incoming forward edge and one outgoing forward edge, every vertex from the initial configuration has one outgoing forward edge and every anchor node has one incoming forward edge.

One can prove this by showing their equivalence to (P1) and (P2) . Next, we define alternating and augmenting trails that play a critical role in processing a request.

Alternating Trails: Recollect that, in graph theory, a *trail* T is a path that is not necessarily a simple path but it does not repeat edges. We define an *alternating trail* T in G_i as a directed trail that alternates between forward and backward edges and ends at an anchor node.

When a new request r_{i+1} arrives, we include the request r_{i+1} and extend the residual graph to create an *extended* graph G^0_{i+1} from G_i as follows. The new vertex set V_{i+1} is $V_i \cup \{r_{i+1}\}$. The edges incident on r_{i+1} are as follows: for each vertex $v \in V_i$, if v is not an anchor node, we add a backward edge directed from r_{i+1} to v. Figure 5.1(a) shows an example of an extended graph where $r_{i+1} = r_6$ with i_1, i_2, i_3, i_4 being the nodes in initial configuration and a_1, a_2, a_3, a_4 are the anchor nodes. Any alternating trail T in the extended graph G^0_{i+1} that starts at r_{i+1} is an *augmenting trail*. Every edge going out of r_{i+1} in the extended graph G^0_{i+1} is a backward edge. Therefore, an augmenting trail T starts with a backward edge and ends at an anchor node. For example, in Figure 5.1(a), $\langle r_6, i_1, r_2, i_2, r_3, r_2, r_4, r_1, a_3 \rangle$ is an augmenting trail.

Alternating Graph: Finding augmenting trails can be tricky. Typical graph search algorithms only find paths and not trails. In order to assist us in finding an augmenting trail efficiently, we define a different directed graph called the *alternating graph* for G^0_{i+1} and denote it by $\mathcal{G}^0_{i+1}(\mathcal{V}^0_{i+1}, \mathcal{E}^0_{i+1})$. Every directed simple path in this alternating graph $\mathcal{G}^0_{i+1}$ maps to a unique alternating trail in G^0_{i+1} and every augmenting trail T in G^0_{i+1} maps to a unique simple path in the alternating graph $\mathcal{G}^0_{i+1}$ which we refer to as the *augmenting path* (Lemma 5.2).

Thus, finding augmenting trails in G^0_{i+1} reduces to finding augmenting paths in $\mathcal{G}^0_{i+1}$, which can be done via graph search algorithms. We describe the alternating graph next. The vertex set $\mathcal{V}^0_{i+1}$ of the alternating graph is the same as that of G^0_{i+1}, i.e., $\mathcal{V}^0_{i+1} = V_{i+1}$. The edge set of the alternating graph, $\mathcal{E}^0_{i+1}$, is defined as follows. For every vertex v, if v has a backward edge to a node v', then we add a directed edge from v to $f(v')$ in $\mathcal{E}^0_{i+1}$. Figure 5.1(a) is an extended graph and Figure 5.1(b) is its alternating graph. For any directed edge $(v, f(v'))$ in $\mathcal{G}^0_{i+1}$, denoted by $\textsc{Proj}(v, f(v'))$ is the backward edge (v, v') concatenated with the forward edge $(v', f(v'))$, i.e., $\textsc{Proj}(v, f(v')) = \langle (v, v'), (v', f(v')) \rangle$. For example, the projection of an edge (r_3, r_4) (Figure 5.1(b)) in the alternating graph consists of the edges $\langle (r_3, r_2), (r_2, r_4) \rangle$ (Figure 5.1(a)) of the residual graph. For any path P in the alternating graph, its projection is simply the concatenation of the projection of the individual edges. The highlighted augmenting path $\langle r_6, r_2, r_3, r_4, a_3 \rangle$ (Figure 5.1(b)) when projected gives the highlighted augmenting trail $\langle r_6, i_1, r_2, i_2, r_3, r_2, r_4, r_1, a_3 \rangle$ (Figure 5.1(a)). The construction of alternating graphs and the definition of projection will also extend to the residual graph G_{i+1} in a straightforward way. The alternating graph for G_{i+1} will be referred to as $\mathcal{G}_{i+1}(\mathcal{V}_{i+1}, \mathcal{E}_{i+1})$.

Lemma 5.1. *For any two edges (u, v) and (u', v') in $\mathcal{G}^0_i$, their projections are edge-disjoint if and only if the head of both of the edges are distinct, i.e., $v \neq v'$.*

Proof. The projection of (u, v) is $\langle (u, p(v)), (p(v), v) \rangle$ and the projection of (u', v') is

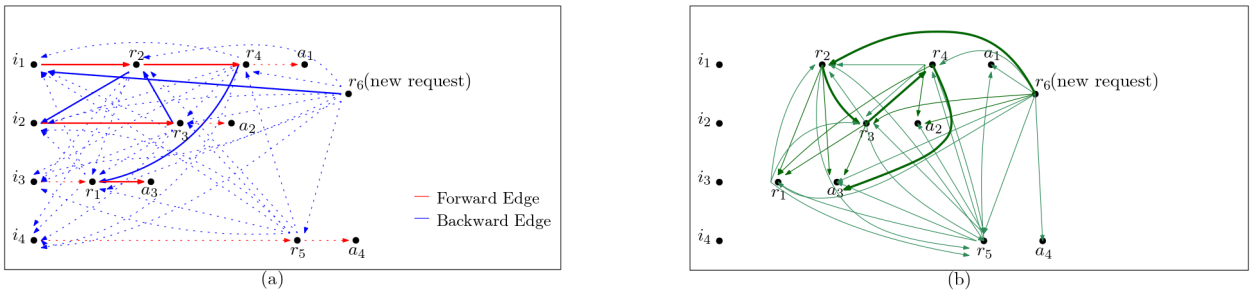


Figure 5.1: Example of an (a) Extended graph G_6^0 and (b) its alternating graph $\mathcal{G}_6^0$

$\langle (u', p(v')),$
$(p(v'), v') \rangle$. Note that for every vertex s, there is a unique previous vertex $p(s)$. Therefore, if these projections are edge-disjoint and $(p(v), v)$ and $(p(v'), v')$ are distinct edges, then v and v' must be distinct points, i.e., $v \neq v'$. If $v \neq v'$, then $p(v) \neq p(v')$. Therefore, the forward edges $(p(v), v)$ and $(p(v'), v')$ are two vertex-disjoint edges implying that the projections of (u, v) and (u', v') are edge-disjoint. □

Lemma 5.2. *For every directed simple path P in the alternating graph $\mathcal{G}_i^0$ that ends at an anchor node, its projection $T = \textsc{Proj}(P)$ is an alternating trail. Furthermore, for every alternating trail T in G_i^0 where the first edge of T is a backward edge, there is a directed simple path P in $\mathcal{G}_i^0$ such that $\textsc{Proj}(P) = T$.*

Proof. The in-degree of any vertex on a simple directed path is at most one. Therefore, for any two edges (u, v) and (u', v') on a simple directed path P, $v \neq v'$. From Lemma 5.1, the projections of (u, v) and (u', v') are edge-disjoint. Therefore, the projection T of P, which is simply the concatenation of projections of all the edges of P, will be a path that does not repeat any edges, i.e., T is a trail. By construction, T starts with a backward edge, alternates between backward and forward edges, and ends at an anchor node, i.e., T is an alternating trail.

For any alternating trail T in G_i^0, let the backward edges be $(u_1, v_1), (u_2, v_2), \ldots, (u_j, v_j)$ in

the order in which they appear on the trail. Similarly let the forward edges be $(v_1, f(v_1)), (v_2, f(v_2)), \ldots, (v_j, f(v_j))$, i.e., $T = \langle (u_1, v_1), (v_1, f(v_1)), (u_2, v_2), (v_2, f(v_2), \ldots, (u_j, v_j), (v_j, f(v_j)) \rangle$. By our assumption, the first edge of the alternating trail (u_1, v_1) must be a backward edge, and $f(v_j)$ must be an anchor node, and for $1 \leq t < j$, $f(v_t) = u_{t+1}$. For each $1 \leq t \leq j$, the pair of edges $(u_t, v_t)(v_t, f(v_t))$ is represented by a unique directed edge $(u_t, f(v_t))$ in $\mathcal{G}_i^0$. We can therefore *lift* T to a path P in $\mathcal{G}_i^0$ by simply replacing successive pairs $(u_t, v_t)(v_t, f(v_t))$ with $(u_t, f(v_t))$. The resulting sequence of edges is $P = \langle (u_1, f(v_1)), (u_2, f(v_2)), \ldots, (u_j, f(v_j)) \rangle$. Since for $1 \leq t < j$, $f(v_t) = u_{t+1}$, P is precisely the directed path $\langle u_1, u_2, \ldots, u_j, f(v_j) \rangle$. Furthermore, since T is a trail and does not repeat any edges, for any two edges (u, v) and (u', v') in P, its projections will be edge-disjoint. Therefore, from Lemma 5.1, $v \neq v'$ and so, P is a simple path. □

Augmentation: Consider any augmenting trail T in the extended graph G_{i+1}^0 that starts at r_{i+1} and ends at an anchor node a. We *augment* a valid solution σ_i (represented by the extended graph G_{i+1}^0) along an augmenting trail T to produce a solution σ_{i+1} and the residual graph G_{i+1} as follows:

To obtain the residual graph G_{i+1} from the extended graph G_{i+1}^0, we can simply reverse the direction of all the edges on the augmenting trail T and relabel the forward edges as backward and all backward edges as forward. Finally, we remove the incoming forward edge to the anchor node a and add a new forward edge from r_{i+1} to the anchor node a and update the location of a to that of r_{i+1}.

Equivalently, one can consider modifying the k paths $\{\Gamma_1, \Gamma_2, \ldots, \Gamma_k\}$ of σ_i by removing all forward edges of T and adding all backward edges of T to obtain the k paths $\{\Gamma'_1, \Gamma'_2, \ldots, \Gamma'_k\}$ of σ_{i+1}. It can be shown that the solution σ_{i+1} is a valid solution that also serves request r_{i+1}. One can also generalize the augment operation for alternating trails and cycles. We refer to this generalized operation as the *flip* operation and use it in Section 5.3.2.

We define the net-cost of an augmenting trail T to be

$$\Phi(T) = \sum_{(u,v) \text{ is backward}} d(u,v) - \sum_{(u,v) \text{ is forward}} d(u,v).$$

Note that the net-cost of an augmenting trail T with respect to σ_i is the change in the cost due to augmenting σ_i along T. Therefore, one can express net-cost as $\Phi(T) = w(\sigma_{i+1}) - w(\sigma_i)$.

For any edge (u, v) in the alternating graph, we set its cost $c(u, v) = d(u, p(v)) - d(p(v), v)$ and for any simple path P, let $c(P) = \sum_{(u,v)\in P} c(u, v)$ denote its *net-cost*. Note that the cost of any edge in the alternating graph can be negative. For any simple augmenting path P in the alternating graph $\mathcal{G}^0_{i+1}$, its net-cost $c(P)$ is simply the net-cost of its projection $\Phi(\textsc{Proj}(P))$.

Lemma 5.3. *An augmenting path P in the alternating graph and the projection of P have equal net-costs.*

Proof. Given an augmenting path P in the alternating graph, let the augmenting trail P' be its projection. Note that the set $B = \{(a, p(b)) \mid (a, b) \in P\}$ is the set of all backward edges of P'. Similarly, the set $F = \{(p(b), b) \mid (a, b) \in P\}$ is the set of all forward edges of P'. Therefore, the net-cost of P is $\sum_{(a,b)\in P} c(a, b) = \sum_{(a,b)\in P}(d(a, p(b)) - d(p(b), b)) = \sum_{(u,v)\in B} d(u, v) - \sum_{(u,v)\in F} d(u, v) = \Phi(P')$. □

Let $y(\cdot)$ be a weight associated with every vertex of the alternating graph. We say that any valid solution σ_i and the weight function $y(\cdot)$ is *feasible* if for any edge (a, b) directed from a to b

$$y(a) - y(b) \leq c(a, b). \tag{5.1}$$

We say that any edge satisfying this inequality is *feasible*. We define *slack* of any edge (a, b) directed from a to b to be $c(a, b) + y(b) - y(a)$ and denote it by $s(a, b)$. Given these notations,

we are ready to describe our algorithm.

5.2 The Algorithm

After processing i requests, our algorithm will maintain a feasible, valid solution $\sigma = \sigma_i$. We refer to this as the *offline* solution. Initially, the weight $y(v)$ for every vertex $v \in \mathcal{C}^0$ is set to 0 and the offline solution σ is empty. For $i \geq 0$, using the alternating graph $\mathcal{G}^0_{i+1}$, our algorithm will identify an appropriate augmenting trail T in the extended graph G^0_{i+1}. Recollect that T ends at an anchor node a. The algorithm then moves the server located at a and that served request $p(a)$ to serve request r_{i+1}. The offline solution σ is updated by augmenting σ_i along T leading to a valid solution $\sigma = \sigma_{i+1}$. The algorithm consists of four steps[2]:

(1) *Augmenting path search:* Let G' be identical to this alternating graph $\mathcal{G}^0_{i+1}$ except the cost of any edge (a, b) is replaced by its slack $s(a, b)$. Note that G' is a graph with only nonnegative edge-costs. The algorithm executes Dijkstra's algorithm on G' with r_{i+1} as the source. Dijkstra's algorithm returns the shortest path from r_{i+1} to every other vertex in $\mathcal{V}^0_{i+1}$. Let, for any vertex $v \in \mathcal{V}^0_{i+1}$, ℓ_v be its shortest path cost as returned by Dijkstra's algorithm from the source r_{i+1}.

(2) *Determine net-cost:* Next, we compute the minimum net-cost augmenting path from r_{i+1} to each of the k anchor nodes $\{a_1, \ldots, a_k\}$. For any anchor node $a_j \in \{a_1, \ldots, a_k\}$, we set the minimum net-cost to be $\Phi_j = \ell_{a_j} - y(a_j)$ and the path P_j corresponding to this net-cost is the shortest path from r_{i+1} to a_j in G' as returned by Dijkstra's algorithm (Step 1).

[2]An implementation of this algorithm is available here: https://github.com/RachitaS/ScalableWorkFunction_Public

(3) *Choose server:* Let $a_m = \arg\min_{a_j \in \{a_1,\ldots,a_k\}}(d(a_j, r_{i+1}) + \Phi_j)$ and let s be the server located at a_m with $p(a_m)$ as its last served request. We assign s to serve request r_{i+1}.

(4) *Update offline solution:* We update the offline solution as follows: **(a)** For any vertex $v \in V_{i+1}$, if $\ell_v < \ell_{a_m}$, we set its weight $y(v) \leftarrow y(v) + \ell_{a_m} - \ell_v$. **(b)** After updating the weights, we augment σ_i along P_{a_m} to obtain $\sigma = \sigma_{i+1}$ and update the edges of the alternating graph to reflect the new solution σ. We also set $y(a_m) \leftarrow 0$.

Our algorithm maintains the following two invariants at all times:

(I1): The offline solution σ along with the weights $y(\cdot)$ is a valid and feasible solution, and, **(I2):** Let $\mathcal{C}^i$ be the final configuration of σ_i. Then, $\sigma_i = \sigma_i^*(\mathcal{C}^i)$. Furthermore, for every anchor node $a_j \in \{a_1, \ldots, a_k\}$, let $\mathcal{C}_j^{i+1} = \mathcal{C}^i \setminus \{a_j\} \cup \{r_{i+1}\}$. Then $\Phi_j = w(\sigma_{i+1}^*(\mathcal{C}_j^{i+1})) - w(\sigma_i^*(\mathcal{C}^i))$. The proofs of these invariants are given in Section 5.3. Note that, after each request is processed the set of edges in the alternating graph can change substantially. Despite this, our weight updates guarantee that every newly added edge in the alternating graph continues to be feasible (Section 5.3.1).

Efficiency Note that the $|V_{i+1}| = i + 1 + 2k$ and $|E_{i+1}| = O((i + k)^2)$. The extended graph and alternating graphs also have identical bounds. Step 1 of the algorithm requires computation of G' and an execution of Dijkstra's algorithm on G' which takes $O((i + k)^2))$ time. Step 2 of the algorithm requires constant time computation for each anchor node and, therefore, takes $O(k)$ time. The paths P_j computed in Step 2 can be compactly represented using the shortest path tree that is returned by Dijkstra's algorithm. Therefore, computing P_j does not require any additional time. Choosing the server in Step 3 can be performed by simply accessing the cost between the r_{i+1} and each of the k servers and computing the one that minimizes $\Phi_j + d(a_j, r_{i+1})$. Step 3, therefore, takes only $O(k)$ time. Step 4(a) requires us to update the weight at each vertex, which can be done in $O(i + k)$ time. Step 4(b)

requires augmenting and updating the residual and alternating graphs, each of which can be performed in $O((i+k)^2)$. Therefore, the time taken to process each request is dominated by $O(i^2+k^2) = O(n^2)$.

Next, assuming the invariants hold, we will show that the algorithm picks the same server as the Work Function Algorithm.

Correctness: The following lemma establishes a link between the net-cost of an augmenting path and the sum of the slacks along its edges.

Lemma 5.4. *Suppose σ_i and the weights $y(\cdot)$ form a feasible solution. For any augmenting path P in the alternating graph that starts at r_{i+1} and ends at an anchor node a, its net-cost is*

$$\Phi(P) = y(r_{i+1}) - y(a) + \sum_{(u,v)\in P} s(u,v). \tag{5.2}$$

Proof. Every vertex $v' \in P$ with the exception of the first vertex r_{i+1} and the last vertex a will have an incoming edge (u', v') and an outgoing edge (v', w') in P. The weight of v', $y(v')$ is added with respect to (v', w') and subtracted with respect to the edge (u', v') and therefore, the net-contribution of v' to Equation equation 5.2 is zero. The first vertex r_{i+1} participates in the first edge of P and contributes $+y(r_{i+1})$ to Equation equation 5.2. The last vertex a participates only in the last edge and contributes $-y(a)$ to Equation equation 5.2. □

From Invariant (I1), Equation 5.2 and since $y(r_{i+1}) = 0$, the minimum net-cost path P_j from r_{i+1} to some anchor node a_j in the alternating graph $\mathcal{G}^0_{i+1}$ is also the augmenting path that minimizes the sum of slacks along its edges. From invariant (I1) all slacks are non-negative and so, P_j will be the augmenting path returned by the execution of Dijkstra's algorithm in Step 1 of the algorithm. Furthermore, $\ell_{a_j} = \sum_{(u,v)\in P_j} s(u,v)$. Therefore, from Equation 5.2, we conclude that $\Phi_j = \Phi(P_j) = \ell_{a_j} - y(a_j)$. Therefore, Step 2 of the algorithm will correctly

compute the minimum net-cost augmenting path to every anchor node $a_j \in \{a_1, \ldots, a_k\}$.

Step 3 of the algorithm selects the server located at the anchor node $a_m = \text{argmin}_{a_j \in \{a_1,\ldots,a_k\}}$ $(d(a_j, r_{i+1}) + \Phi_j)$. Let $X_j = \mathcal{C}_j^{i+1} = \mathcal{C}^i \setminus \{a_j\} \cup \{r_{i+1}\}$. By Invariant (I2), $(d(a_j, r_{i+1}) + \Phi_j) = d(a_j, r_{i+1}) + w(\sigma_{i+1}^*(X_j)) - w(\sigma_i^*(\mathcal{C}^i))$ and, $\text{argmin}_{a_j \in \{a_1,\ldots,a_k\}}(d(a_j, r_{i+1}) + \Phi_j)$ is

$$
\begin{aligned}
&= \underset{j \in \{1,\ldots k\}}{\text{argmin}} (d(a_j, r_{i+1}) + w(\sigma_{i+1}^*(X_j)) - w(\sigma_i^*(\mathcal{C}^i))) \\
&= \underset{j \in \{1,\ldots,k\}}{\text{argmin}} (d(a_j, r_{i+1}) + w(\sigma_{i+1}^*(X_j)).
\end{aligned}
$$

The last equality follows from the fact that $w(\sigma_i^*(\mathcal{C}^i))$ is the same for every choice of j. Thus, we choose the same server as required by the work function algorithm.

5.3 Proof of Invariants

5.3.1 Proof of Invariant (I1)

Recall the definition of feasibility. Any valid solution σ_i with the weight function $y(\cdot)$ associated with each vertex of its alternating graph is *feasible* if for every edge (a, b) directed from a to b in its alternation graph satisfies the following equation.

$$y(a) - y(b) \leq c(a, b). \tag{5.3}$$

Furthermore, any edge satisfying the above inequality is *feasible*.

We prove a slightly stronger version of Invariant (I1)

(I1): The offline solution σ along with the weights $y(\cdot)$ maintained by the algorithm is a

valid and feasible solution. Furthermore, for any forward edge (u, v) in the residual graph of σ,

$$y(v) \geq d(u, v). \tag{5.4}$$

We can prove this invariant by induction. At time $t = 0$, we have the servers in the initial configuration. The vertex set of the initial residual graph G_0 only contains the vertices for the initial configuration and the anchor nodes. The edge set of the initial residual graph G_0 contains only forward edges directed from each server in the initial configuration to its anchor node. Since there are no backward edges in G_0, the alternating graph $\mathcal{G}_0$ does not have edges. Therefore, σ_0 and the vertex weights $y(\cdot)$ are trivially feasible.

Let the solution σ_i after serving i requests be a valid and feasible solution and let $y(\cdot)$ at the end of processing request r_i satisfy equation 5.4. Given this, we will now prove that the solution σ_{i+1} is valid and feasible and the updated vertex weight satisfies equation 5.4. Note that each new request arrives with a default weight 0. Given a valid feasible solution σ_i along with the vertex weight function $y(\cdot)$, the algorithm first constructs the extended graph G^0_{i+1}. Lemma 5.6 shows that the corresponding alternating graph $\mathcal{G}^0_{i+1}$ with weight function $y(\cdot)$ continues to be feasible. Steps 1, 2 and 3 do not modify σ_i or the vertex weights. Therefore, σ_i continues to remain valid and feasible till the end of Step 3.

In Step 4(a) of the algorithm, the weights on the vertices of $\mathcal{G}^0_{i+1}$ are updated. Let $y(\cdot)$ be the weights prior to executing step 4(a) and let $y'(\cdot)$ be the weights after executing step 4(a). Lemma 5.8 proves that $\mathcal{G}^0_{i+1}$ along with the updated vertex weights $y'(\cdot)$ remain feasible. In Step 4(b), the algorithm augments the solution σ_i along the augmenting trail T (as determined in Step 3) leading to σ_{i+1}. Lemma 5.10 shows that the solution σ_{i+1} remains valid and feasible. Finally, in Lemma 5.11, we argue that the vertex weights at the end of Step 4(b) satisfies equation 5.4.

Lemma 5.5. *Suppose P is the augmenting path from r_{i+1} to the anchor node a_m chosen in step 3 of the algorithm, then the slack $s(u,v)$ on every edge (u,v) of P after the vertex weight update in step 4(a) is zero.*

Proof. Let $y(\cdot)$ be the vertex weight prior to Step 4(a) and $y'(\cdot)$ be the vertex weight after Step 4(a). By our choice in Step 3, P is the shortest path computed by Dijkstra's algorithm from r_{i+1} to a_m in G'. Since ℓ_{a_m} is the cost of P in G', every vertex $v \in P$ has a shortest path cost at most ℓ_{a_m}, i.e., $\ell_v \leq \ell_{a_m}$. Furthermore, by the optimal sub-structure property of shortest paths, for any edge $(u,v) \in P$, $\ell_v - \ell_u = s(u,v) = c(u,v) + y(v) - y(u)$ or $\ell_v - \ell_u = c(u,v) + y(v) - y(u)$.

$$\begin{aligned} c(u,v) + (y(v) - \ell_v) - (y(u) - \ell_u) &= 0, \\ c(u,v) + (y(v) - \ell_v + \ell_{a_m}) - (y(u) - \ell_u + \ell_{a_m}) &= 0, \\ c(u,v) + y'(v) - y'(u) &= 0. \end{aligned}$$

The second to last equality is obtained by simply adding and subtracting ℓ_{a_m} to the LHS. The last equality follows from the fact that $\ell_v \leq \ell_{a_m}$ and $\ell_u \leq \ell_{a_m}$ and the update of the vertex weights defined in Step 4(a). □

Lemma 5.6. *Given a valid feasible solution σ_i, all the edges of the alternating graph $\mathcal{G}^0_{i+1}$ are feasible.*

Proof. From the inductive hypothesis, σ_i is a feasible solution i.e. the edges of $\mathcal{G}_i$ satisfy equation 5.3, and, the weights $y(\cdot)$ satisfies equation 5.4. The extended graph G^0_{i+1} is created by the addition of r_{i+1} to the vertex set of G_i in the vertex set. Furthermore, for every $j \leq i$ a backward edge is added from r_{i+1} to r_j and an edge $(r_{i+1}, f(r_j))$ is added to $\mathcal{G}^0_{i+1}$. We show that for every such edge, $(r_{i+1}, f(r_j)) \in \mathcal{G}^0_{i+1}$, the feasibility condition equation 5.3 holds.

The cost of the edge $(r_{i+1}, f(r_j))$ is

$$c(r_{i+1}, f(r_j)) = d(r_{i+1}, r_j)) - d(r_j, f(r_j)) \tag{5.5}$$

$$\geq -d(r_j, f(r_j)). \tag{5.6}$$

Note that the vertex weight of r_{i+1}, $y(r_{i+1})$ is set to 0. By the inductive hypothesis, $-y(f(r_j)) \leq -d(r_j, f(r_j))$. Adding $y(r_{i+1})$ to the LHS and 0 to the RHS, we get $y(r_{i+1}) - y(f(r_j)) \leq -d(r_j, f(r_j))$ or $y(r_{i+1}) - y(f(r_j)) \leq c(r_{i+1}, f(r_j))$. The last inequality follows from 5.6. This implies that the edge (r_{i+1}, v) satisfies equation 5.3. □

Feasibility after Step 4(b): Step 4(b) in the algorithm augments the solution along an augmenting trail T. In doing so, the alternating graph $\mathcal{G}^0_{i+1}$ is updated to $\mathcal{G}_{i+1}$. The edges $\mathcal{E}_{i+1}$ of $\mathcal{G}_{i+1}$ may include several new edges that were not in $\mathcal{G}^0_{i+1}$. Furthermore, there may be edges whose costs $c(\cdot, \cdot)$ change due to augmentation. We classify all such edges in the alternating graph $\mathcal{G}_{i+1}$ as *affected* edges. The following two lemmas establishes important properties of the affected edges.

Lemma 5.7. *Let P be the augmenting path from r_{i+1} to an anchor node a_m in the alternating graph $\mathcal{G}^0_{i+1}$ that is computed in Step 3 of our algorithm. Given any affected edge (u, v) in $\mathcal{G}_{i+1}$, let $\langle (u, x), (x, v) \rangle$ be its projection. Then, $v \neq a_m$ and (v, x) is a backward edge on the augmenting trail* $\textsc{Proj}(P)$.

Proof. Let P be the augmenting path from r_{i+1} to an anchor node a_m in the alternating graph $\mathcal{G}^0_{i+1}$ that is computed in Step 3 of our algorithm. And let T be the augmenting trail in G^0_{i+1} such that $T = \textsc{Proj}(P)$.

First we will prove that given any affected edge (u, v) in $\mathcal{G}_{i+1}$, $v \neq a_m$. After augmentation along T, we add the forward edge (r_{i+1}, a_m). Since all backward edges are directed from a

later request to an earlier request and since r_{i+1} is the latest request in the residual graph G_{i+1}, there are no in-coming backward edges to r_{i+1}. Therefore, by its description, there will not be any incoming edges to a_m in the alternating graph $\mathcal{G}_{i+1}$ and so $v \neq a_m$.

Since (u, v) is an affected edge, at least one of the edges in its projection $\langle (u, x), (x, v) \rangle$ is newly introduced by the augment operation. We claim that the forward edge (x, v) was added by the augment operation in Step 4(b) of the algorithm. Suppose, for the sake of contradiction, (x, v) was not added in Step 4(b), i.e., (x, v) is a forward edge in G^0_{i+1}. Therefore, (u, x) must be the backward edge that was newly added by the augment operation. We can conclude that the augmenting trail T contains the forward edge (x, u), implying (x, u) is a forward edge in G^0_{i+1}. Note that (x, u) and (x, v) are both forward edges in G^0_{i+1} which contradicts the fact that σ_i is a valid solution. Therefore, we conclude that the forward edge (x, v) was introduced by the augment operation in Step 4(b), i.e., (v, x) is a backward edge in the augmenting trail T. □

Lemma 5.8. *The edges of the alternating graph $\mathcal{G}^0_{i+1}$ remains feasible after the vertex weight update in step 4(a).*

Proof. The alternating graph $\mathcal{G}^0_{i+1}$ before the execution of Step 4(a) is feasible. Let $y(\cdot)$ (resp. $y'(\cdot)$) denote the vertex weights before (resp. after) the weight update of step 4(a). For each edge $(u, v) \in \mathcal{G}^0_{i+1}$, let $s(u, v)$ (resp. $s'(u, v)$) represent the slack on (u, v) before (resp. after) weight update in step 4(a). Let (u, v) be any directed edge in $\mathcal{G}^0_{i+1}$. Note that every edge in $\mathcal{G}^0_{i+1}$ along with the weights $y(\cdot)$ satisfies equation 5.1 and so the slack $s(u, v) \geq 0$. Let a_m be the anchor node chosen by algorithm in Step 3 and for any vertex $w \in \mathcal{G}^0_{i+1}$, recollect that ℓ_w is the shortest path cost returned by Dijkstra's algorithm in Step 1 of the algorithm. For the edge (u, v), there are four possibilities after step 4(a):

Case (i) $\ell_u \geq \ell_{a_m}$ and $\ell_v \geq \ell_{a_m}$. In this case, Step 4(a) will not update the vertex weights

for u and v. So, $y'(u) = y(u)$, $y'(v) = y(v)$, and $s'(u,v) = s(u,v)$. Therefore, (u,v) remains feasible with respect to $y'(\cdot)$.

Case (ii) $\ell_v < \ell_{a_m}$ and $\ell_u \geq \ell_{a_m}$. In this case, Step 4(a) will update the vertex weights to $y'(v) = y(v) + (\ell_{a_m} - \ell_v) > y(v)$ and $y'(u) = y(u)$. Furthermore, from feasibility of (u,v) with respect to $y(\cdot)$, we have $s(u,v) \geq 0$. Therefore, $s'(u,v) = c(u,v) - y'(u) + y'(v) \geq c(u,v) - y(u) - y(v) \geq 0$ implying that (u,v) remains feasible with respect to the updated vertex weight $y'(\cdot)$.

Case (iii) $\ell_u < \ell_{a_m}$ and $\ell_v \geq \ell_{a_m}$. In this case, Step 4(a) updates the vertex weight to $y'(u) = y(u) + (\ell_{a_m} - \ell_u)$ and $y'(v) = y(v)$. From the property of shortest path distances, the shortest path distance from r_{i+1} to v is bounded by the shortest path distance from r_{i+1} to u and the slack of the edge from (u,v), i.e., $\ell_v - \ell_u \leq s(u,v)$. Using the definition of slack we get,

$$\begin{aligned}
\ell_v - \ell_u &\leq c(u,v) - y(u) + y(v), \\
(\ell_{a_m} - \ell_u) - (\ell_{a_m} - \ell_v) &\leq c(u,v) - y(u) + y(v), \\
[(\ell_{a_m} - \ell_u) + y(u) - & \\
((\ell_{a_m} - \ell_v) + y(v))] &\leq c(u,v), \\
y'(u) - ((\ell_{a_m} - \ell_v) + y'(v)) &\leq c(u,v)
\end{aligned}$$

The last inequality follows from the fact that $y'(u) = (\ell_{a_m} - \ell_u) + y(u)$. Furthermore, since $\ell_{a_m} < \ell_v$, we get $y'(u) - y'(v) \leq c(u,v)$, i.e., the edge (u,v) remains feasible.

Case (iv) $\ell_u < \ell_{a_m}$ and $\ell_v < \ell_{a_m}$. In this case, Step 4(a) sets $y'(u) = y(u) + (\ell_{a_m} - \ell_u)$ and $y'(v) = y(v) + (\ell_{a_m} - \ell_v)$. Again, the shortest path from r_{i+1} to v is of cost bounded by the shortest path from r_{i+1} to u and the slack on the edge (u,v). Therefore, $\ell_v \leq \ell_u + s(u,v)$

i.e. $\ell_v - \ell_u \leq s(u,v)$. Using the definition of slack we get

$$\begin{aligned}
\ell_v - \ell_u &\leq && c(u,v) - y(u) + y(v), \\
(\ell_{a_m} - \ell_u) - (\ell_{a_m} - \ell_v) &\leq && c(u,v) - y(u) + y(v), \\
[(\ell_{a_m} - \ell_u) + y(u) - & && \\
((\ell_{a_m} - \ell_v) + y(v))] &\leq && c(u,v), \\
y'(u) - y'(v) &\leq && c(u,v).
\end{aligned}$$

implying (u,v) is feasible with respect to $y'(\cdot)$. □

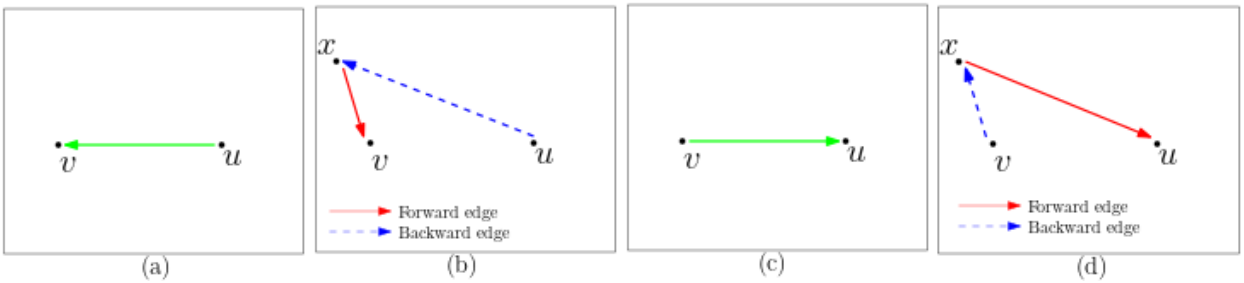
Lemma 5.9. *Let P be the augmenting path computed in Step 3 that goes from request r_{i+1} to an anchor node a_m in the alternating graph $\mathcal{G}^0_{i+1}$. For any vertex v on the path P where $v \neq a_m$, let $n(v)$ denote the vertex that succeeds v on P. Given any affected edge (u,v) in $\mathcal{G}_{i+1}$, let $\langle (u,x),(x,v)\rangle$ be its projection. Then,*

(i) Either the edge (v,u) is on the augmenting path P with $\textsc{Proj}(v,u) = \langle (v,x),(x,u)\rangle$, or

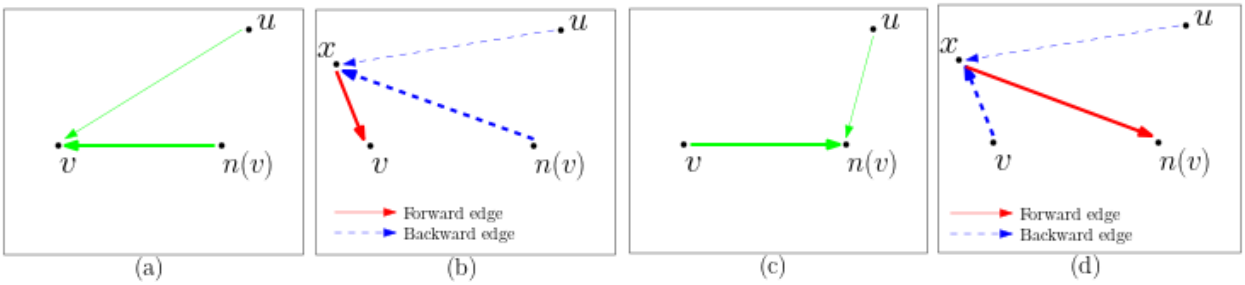
(ii) There is an edge $(u,n(v))$ in $\mathcal{G}^0_{i+1}$ with its projection $\textsc{Proj}(u,n(v)) = \langle (u,x),(x,n(v))\rangle$.

Proof. From Lemma 5.7, we know that (v,x) is a backward edge in the augmenting trail T. There are two possibilities for the edge (u,x): (a) (u,x) was also added as a backward edge by the augment process, or (b) (u,x) was also an edge in the extended graph G^0_{i+1}.

For case (a), the edge (x,u) was a forward edge prior to augmentation. Since both the backward edge (v,x) and the forward edge (x,u) are in the augmenting trail T, we will have an edge (v,u) in the augmenting path P, implying (i). An instance of this case is shown in Figure 5.2a where directed edge (u,v) is the affected edge.



(a) Demonstrating Case(a): (u, v) is an affected edge with projection $\langle (u, x), (x, v) \rangle$ such that (x, u) was a forward edge prior to augmentation.
(a) Affected edge $(u, v) \in \mathcal{G}_{i+1}$. (b) Projection of the affected edge (u, v), $\textsc{Proj}(u, v) = \langle (u, x), (x, v) \rangle$. (c) Alternating graph edge $(v, u) \in \mathcal{G}^0_{i+1}$ before augmentation. (d) Projection of the edge (v, u), $\textsc{Proj}(v, u) = \langle (v, x), (x, u) \rangle$ before augmentation.



(b) Demonstrating Case(b): (u, v) is an affected edge with projection $\langle (u, x), (x, v) \rangle$ such that (u, x) was a backward edge prior to augmentation as well.
(a) Affected edge $(u, v) \in \mathcal{G}_{i+1}$. (b) Projection of the affected edge $(u, v) \in \mathcal{G}_{i+1}$, $\langle (u, x), (x, v) \rangle$. (c) Alternating graph edges of $\mathcal{G}^0_{i+1}$ before augmentation where $(v, n(v)) \in P$ and $(u, n(v)) \in \mathcal{G}^0_{i+1}$ (d) Projection of the edge $(u, n(v)) \in \mathcal{G}^0_{i+1}$, $\langle (u, x), (x, n(v)) \rangle$ before augmentation.

Figure 5.2

In case (b), (u, x) is also a backward edge in G^0_{i+1} (Figure 5.2b(b)). Since (v, x) is a backward edge in T, the projection of the edge $(v, n(v))$ will contain the backward edge (v, x) followed by the forward edge $(x, n(v))$. Since (u, x) is a backward edge and $(x, n(v))$ is a forward edge in G^0_{i+1}, we will have an edge $(u, n(v))$ in the alternating graph $\mathcal{G}^0_{i+1}$. An example of this case is demonstrated in Figure 5.2b where the directed edge (u, v) in Figure 5.2b(a) is the affected edge. Figure 5.2b(b) shows the projection of (u, v). Figure 5.2b(c) shows the scenario before augmentation where $(v, u) \in \mathcal{G}^0_{i+1}$ and Figure 5.2b(d) shows (v, u)'s projection in G^0_{i+1}. □

Lemma 5.10. *σ_{i+1} is a valid and feasible solution after the Augment operation in Step 4(b).*

Proof. Let P be the augmenting path from r_{i+1} to the anchor node chosen in Step 3. To prove that σ_{i+1} is a feasible solution after Step 4(b), we need to show that the edges of alternating graph $\mathcal{G}_{i+1}$ that are affected by the augment operation along the path P continue to be feasible and satisfy equation 5.3.

Let $c(\cdot, \cdot)$ be the cost function of edges in the alternating graph $\mathcal{G}^0_{i+1}$, i.e., prior to augmentation and let $c'(\cdot, \cdot)$ be the cost function of edges in the alternating graph $\mathcal{G}_{i+1}$, i.e., after augmentation. From Lemma 5.9, one of the following cases is true.

(i) Edge (v, u) is on the augmenting path P with $\textsc{Proj}(v, u) = \langle (v, x), (x, u) \rangle$, or

(ii) There is an edge $(u, n(v))$ in $\mathcal{G}^0_{i+1}$ with its projection $\textsc{Proj}(u, n(v)) = \langle (u, x), (x, n(v)) \rangle$.

We will consider both these cases separately.

Case (i): For the edge (v, u), $c(v, u) = d(v, x) - d(x, u)$. After augmentation, $\textsc{Proj}(u, v) = \langle (u, x), (x, v) \rangle$ and the cost of the affected edge $(u, v) \in \mathcal{G}_{i+1}$ is $c'(u, v) = d(u, x) - d(x, v) = -c(v, u)$. From Lemma 5.5, after Step 4(a), the slack on every edge of the augmenting path P, including $(v, u) \in P$ is 0 i.e. $s(v, u) = (c(v, u) + y(u) - y(v)) = 0$. The slack on the

affected edge $(u, v) \in \mathcal{G}_{i+1}$ can be calculated as,

$$\begin{aligned} s(u,v) &= c'(u,v) - y(u) + y(v) \\ &= -(c(v,u) + y(u) - y(v)) \\ &= 0. \end{aligned}$$

Hence, the affected edge $(u, v) \in \mathcal{G}_{i+1}$ is feasible.

Case (ii): There is an edge $(u, n(v))$ in $\mathcal{G}^0_{i+1}$ and $\textsc{Proj}(u, n(v)) = \langle (u, x), (x, n(v)) \rangle$. From Lemma 5.5, after Step 4(a), every edge on P including $(v, n(v)) \in P$ has a slack of 0. Hence, $y(v) - y(n(v)) = c(v, n(v)) = d(v, x) - d(x, n(v))$, or

$$y(n(v)) = y(v) + d(x, n(v)) - d(v, x). \tag{5.7}$$

Since $(u, n(v))$ is an edge in $\mathcal{G}^0_{i+1}$, it is a feasible edge after the step 4(a) (Lemma 5.8) and we get $y(u) - y(n(v)) \leq c(u, n(v))$. Substituting $c(u, n(v))$ as $d(u, x) - d(x, n(v))$, we get $y(u) - y(n(v)) \leq d(u, x) - d(x, n(v))$. From equation equation 5.7, we can rewrite this inequality as

$$\begin{aligned} y(u) - y(v) - d(x, n(v)) + d(v, x) &\leq d(u, x) - d(x, n(v)), \\ y(u) - y(v) &\leq d(u, x) - d(x, v), \\ y(u) - y(v) &\leq c'(u, v), \end{aligned}$$

implying that the affected edge (u, v) is a feasible edge.

Next, we show that σ_{i+1} is a valid solution after Step 4(b). To show that σ_{i+1} remains a valid solution, we need to show that the forward edges of G_{i+1} satisfies (Q1) and (Q2). By construction, every backward edge is directed from a later request to an earlier one. During

augmentation, for every backward edge on the augmenting trail, we reverse its direction and label it as a forward edge. Therefore, every newly introduced forward edge is from an earlier request to a later one implying (Q1).

Next, we will show (Q2)

The first vertex of P is r_{i+1}. Let a be the anchor node at the end of P. Consider any edge (u, v) of the simple directed path P. Note that its projection is $\langle (u, p(v)), (p(v), v) \rangle$ where $(u, p(v))$ is a backward edge and $(p(v), v)$ is a forward edge. Due to augmentation, this projection is modified as follows: $(v, p(v))$ becomes a backward edge and $(p(v), u)$ is now a forward edge. We abuse notation and refer to the modifications made by augmentation along the projection of (u, v) as augmentation along the edge (u, v).

Augmentation along (u, v) modifies the outgoing forward edge from $p(v)$. It also removes the incoming forward edge to v and adds an incoming forward edge to u. Therefore, augmentation along (u, v) does not change the number of forward edges coming in and going out of $p(v)$, i.e., $p(v)$ continues to satisfy (Q2). However, it increases the number of forward edges coming into u by 1 and reduces the number of incoming forward edges incident on v by 1.

Every vertex v' along P, except for the first and the last vertex, i.e., $v' \notin \{r_{i+1}, a\}$, will be the tail for some edge $(u', v') \in P$ and the head for some edge $(v', w') \in P$. Augmentation along (u', v') will reduce the incoming forward edge on v' by one. On the other hand, augmenting along (v', w') will increase the incoming forward edge on v' by one. As a result the incoming and outgoing forward edges incident on v' remain unchanged. Therefore, for every vertex except r_{i+1} and a, we can conclude that (Q2) holds.

The first vertex r_{i+1} is the tail of the first edge in P, augmentation will result in a new incoming forward edge in G_{i+1}. Finally, the last vertex $a \in P$ is the head of the last edge.

Therefore, augmentation causes removal of the only incoming forward edge to a. Instead, we add a forward edge from r_{i+1} to a. This guarantees that r_{i+1} contains exactly one incoming forward edge and one outgoing forward edge satisfying (Q2). Furthermore, the anchor node a will have exactly one incoming forward edge satisfying (Q2).

□

Finally, we will show that the updated vertex weights satisfy equation 5.4.

Lemma 5.11. *Consider any solution σ_i maintained by the algorithm with residual graph G_i, for any forward edge (u, v) in the residual graph,*

$$y(v) \geq d(u, v). \tag{5.8}$$

Proof. The initial solution σ_0 is a trivially valid solution and the only edges in the residual graph are forward edges that go from a vertex in the initial configuration to an anchor node. For any such forward edge (u, v), v is an anchor node with $y(v) = 0$. The cost $d(u, v)$ is also 0 since the anchor node v and the vertex u share the same location. Therefore, inequality equation 5.8 holds.

Let us assume that the inequality equation 5.8 holds after request r_i is processed by our algorithm. To complete the proof, we will show that the inequality will continue to be satisfied after request r_{i+1} is processed. To do so, we will show that the any of the changes made by the algorithm will not violate inequality equation 5.8.

At the start of the algorithm, we add r_{i+1} to G_i to create G^0_{i+1}. Since all the edges incident on r_{i+1} in G^0_{i+1} are backward edges, all forward edges will continue to satisfy inequality equation 5.8. Steps 1, 2 and 3 do not alter the weights $y(\cdot)$ or the alternating graph. Therefore, the inequality equation 5.8 continues to hold for every forward edge during these three steps.

In Step 4(a), we modify the vertex weights for every vertex v with $\ell_v < \ell$. Consider any such vertex and let (u, v) be the in-coming forward edge to v. Recollect that $y(v)$ is the weight prior to the execution of Step 4(a) and $y'(v)$ is the weight after Step 4(a). Since inequality equation 5.8 is true prior to execution of Step 4(a), we have $y(v) \geq d(u, v)$. In Step 4(a), the weight is updated to $y'(v) \leftarrow y(v) - \ell_v + \ell$ provided $\ell_v < \ell$. Since $\ell_v < \ell$, it follows that $y'(v) \geq y(v) \geq d(u, v)$ and inequality equation 5.8 continues to hold.

Next, we show that the inequality equation 5.8 continues to hold after Step 4(b). In Step 4(b), we apply the augment operation along an augmenting trail T (computed in Step 3). Recollect that the first vertex of T is r_{i+1} and the last vertex of T is an anchor node a. Note that the weights of every vertex, except the anchor node a remains unchanged. However, for any v along the alternating trail T, its incoming forward edge may change. As a result of augmentation along T, every backward edge (v_2, v_1) in T changes to a forward edge (v_1, v_2). Let P be the augmenting path in the alternating graph $\mathcal{G}^0_{i+1}$ such that trail $T = \textsc{Proj}(P)$. Let (v_2, v_1) be a backward edge in T. Let (v_2, w) be the edge in P such that $\textsc{Proj}(v_2, w)$ contains the backward edge (v_2, v_1), i.e., $\textsc{Proj}(v_2, w) = \langle (v_2, v_1), (v_1, w) \rangle$. By definition of slack,

$$\begin{aligned} s(v_2, w) &= c(v_2, w) - y(v_2) + y(w) \\ &= d(v_2, v_1) - d(v_1, w) - y(v_2) + y(w). \end{aligned}$$

Since $(v_2, w) \in P$, at the end of Step 4(a), the slack $s(v_2, w)$ becomes 0 (Lemma 5.5). Therefore, we have $(d(v_2, v_1) - d(v_1, w)) - y'(v_2) + y'(w) = 0$ which can be rearranged as

$$y'(v_2) = d(v_2, v_1) - d(v_1, w) + y'(w). \tag{5.9}$$

Since, (v_1, w) is a forward edge in T, after Step 4(a), we have $y'(w) \geq d(v_1, w)$. Substituting

this in 5.9, we get

$$y'(v_2) \geq d(v_1, v_2). \tag{5.10}$$

After augmentation, the backward edge $(v_2, v_1) \in \textsc{Proj}(v_2, w)$ changes to a forward edge $(v_1, v_2) \in G_{i+1}$. Hence, inequality equation 5.9 implies inequality equation 5.8 continues to hold. Therefore, inequality equation 5.8 continues to hold after the augment operation in Step 4(b).

Finally, step 4(b) also adds a forward edge from r_{i+1} to a_m and modifies the vertex weight of the anchor node a_m to 0. Since the cost of this forward edge $d(r_{i+1}, a_m)$ is 0, inequality equation 5.8 holds for the vertex a_m. This concludes the argument that inequality equation 5.8 holds after Step 4(b). □

5.3.2 Proof of Invariant (I2)

Before we present the proof for (I2), we would like to remind the reader of the following discussion.

From Invariant (I1), Equation equation 5.2 and since $y(r_{i+1}) = 0$, the minimum net-cost path P_j from r_{i+1} to some anchor node a_j in the alternating graph G^0_{i+1} is also the augmenting path that minimizes the sum of slacks along its edges. From invariant (I1) all slacks are non-negative and so, P_j will be the augmenting path returned by the execution of Dijkstra's algorithm in Step 1 of the algorithm. Furthermore, $\ell_{a_j} = \sum_{(u,v)\in P_j} s(u, v)$. Therefore, from Equation equation 5.2, we conclude that $\Phi_j = \Phi(P_j) = \ell_{a_j} - y(a_j)$. Therefore, Step 2 of the algorithm will correctly compute the minimum net-cost augmenting path to every anchor node $a_j \in \{a_1, \ldots, a_k\}$.

Invariant (I2): Let $\mathcal{C}^i$ be the final configuration of σ_i. Then, $\sigma_i = \sigma^*_i(\mathcal{C}^i)$. Furthermore, for

every anchor node $a_j \in \{a_1, \ldots, a_k\}$, let $\mathcal{C}_j^{i+1} = \mathcal{C}^i \setminus \{a_j\} \cup \{r_{i+1}\}$. Then $\Phi_j = w(\sigma_{i+1}^*(\mathcal{C}_j^{i+1})) - w(\sigma_i^*(\mathcal{C}^i))$.

Proof: Prior to processing any request, all the servers are in their initial configuration and σ_0 with zero cost is indeed the optimal solution. Assume that, after processing i requests, $\sigma_i = \sigma_i^*(\mathcal{C}^i)$. We will use this to show that $\Phi_j = w(\sigma_{i+1}^*(\mathcal{C}_j^{i+1})) - w(\sigma_i^*(\mathcal{C}^i))$ and $\sigma_{i+1} = \sigma_{i+1}^*(\mathcal{C}^{i+1})$.

Consider $\sigma_{i+1}^j = \sigma_{i+1}^*(\mathcal{C}_j^{i+1})$ to be the smallest cost solution that serves $i+1$ requests and ends in $\mathcal{C}_j^{i+1}$. If there are many minimum-cost solutions, we set σ_{i+1}^j to be the one that has the fewest edges in the symmetric difference with $\sigma_i = \sigma_i^*(\mathcal{C}^i)$. Consider the symmetric difference of the edges of σ_i and σ_{i+1}^j. Since their final configurations $\mathcal{C}^i$ and $\mathcal{C}_j^{i+1}$ differ in only the locations of a_j and r_{i+1}, the symmetric difference will include exactly one augmenting trail from r_{i+1} to a_j and possibly a set $\mathbb{C}$ of alternating cycles.

First, we show that $\mathbb{C}$ is an empty set. For the sake of contradiction, assume $\mathbb{C}$ is not empty. Let C be an alternating cycle with respect to G_{i+1}^0 (extended graph for solution σ_i) in the symmetric difference. The net-cost of C cannot be zero, since otherwise applying the flip operation on the cycle C in $G_{\sigma'}$ will lead to another valid solution σ'' whose cost is identical to that of σ_{i+1}^j and the final configuration is $\mathcal{C}_j^{i+1}$. However, the flip operation will reduce the size of the symmetric difference and so, σ_{i+1}^j has more edges that σ'' in the symmetric difference with σ_i. This contradicts our assumption that σ_{i+1}^j is the minimum-cost valid solution that ends in configuration $\mathcal{C}_j^{i+1}$ and has the smallest symmetric difference with σ_i.

Similarly, the net-cost $\Phi(C)$ cannot be negative, since otherwise applying the flip operation along the cycle C on G_{i+1}^0 will lead to another valid solution that ends in $\mathcal{C}^i$ and has a smaller cost than σ_i . This contradicts the fact that $\sigma_i = \sigma_i^*(\mathcal{C}^i)$ is a minimum-cost solution.

If the net-cost $\Phi(C)$ with respect to G_{i+1}^0 is positive, i.e., $\Phi(C) > 0$, then let C' be the

alternating cycle corresponding to C in $G_{\sigma^j_{i+1}}$ (the residual graph with respect to σ^j_{i+1}). From Corollary 5.13 presented in Section 5.3.3, it follows that the net-cost $\Phi(C') = -\Phi(C) < 0$. Again, applying the flip operation along the cycle C' in $G_{\sigma^j_{i+1}}$ will lead to a valid solution whose cost is smaller than σ^j_{i+1} contradicting the fact that σ^j_{i+1} is the smallest cost solution.

From the above discussion, it follows that the symmetric difference of σ_i and σ^j_{i+1} is an augmenting trail T'. We claim that T' is in fact the minimum net-cost augmenting trail that starts at r_{i+1} and ends at a_j. For the sake of contradiction, suppose T' is not the minimum net-cost augmenting trail and $\Phi(T') > \Phi_j$. Let T be some minimum net-cost augmenting trail in G^0_{i+1} that starts at r_{i+1} and ends at an anchor node a_j. Note that $\Phi_j = \Phi(T)$. Let $\overline{\sigma}^j_{i+1}$ be the valid solution obtained by augmenting σ_i along T in the extended graph G^0_{i+1}. The final configuration of $\overline{\sigma}^j_{i+1}$ is $\mathcal{C}^j_{i+1}$. Then, by its definition,

$$\begin{aligned} w(\sigma^j_{i+1}) - w(\sigma_i) &> w(\overline{\sigma}^j_{i+1}) - w(\sigma_i), \\ w(\sigma^j_{i+1}) &> w(\overline{\sigma}^j_{i+1}), \end{aligned}$$

contradicting the fact that σ^j_{i+1} is a minimum cost solution to serve $i+1$ requests and end in configuration $\mathcal{C}^j_{i+1}$. Thus the net-cost of the augmenting trail T' in the symmetric difference of σ_i and σ^j_{i+1} is Φ_j. From the definition of net-cost, $\Phi_j = w(\sigma^j_{i+1}) - w(\sigma_i) = w(\sigma^*_{i+1}(\mathcal{C}^{i+1}_j)) - w(\sigma^*_i(\mathcal{C}^i))$.

Our algorithm chooses the minimum net-cost path from r_{i+1} to a_m and augments the valid solution along this path. As a result, the cost of the solution σ_{i+1} increases precisely by $w(\sigma^*_{i+1}(\mathcal{C}^{i+1}_m)) - w(\sigma^*_i(\mathcal{C}^i))$ and the new valid solution will end in configuration $\mathcal{C}^{i+1}_m = \mathcal{C}^{i+1}$ and have a cost equal to $w(\sigma^*_{i+1}(\mathcal{C}^{i+1}))$ proving invariant (I2).

5.3.3 Symmetric Difference of Valid Solutions

Next, we introduce the properties of the symmetric difference between two valid solutions. These properties are used in the proof of invariant (I2).

Let σ and σ' be two valid solutions where σ serves the first i requests and σ' serves the first $i+1$ requests. Let G^0_σ be the extended graph with respect to σ and $G_{\sigma'}$ be the residual graph with respect to σ'. We show that the edges in the symmetric difference of σ and σ' can be decomposed into an edge-disjoint set of alternating trails, augmenting trails, and alternating cycles.

Let $\mathcal{C}$ and $\mathcal{C}'$ be the final configurations of σ and σ'. Let X denote the edges in the symmetric difference of σ and σ'. For any edge in X, there is a corresponding directed edge in G^0_σ. We assign the same direction for edge in X. Therefore, by construction, the edges of $X \cap \sigma$ will be forward edges and the edges of $X \cap \sigma'$ will be backward edges in G^0_σ. Figures 5.3(a), (b) and (c) highlight the edges of σ, σ' and $(X \cap G^0_\sigma)$ respectively. For each vertex $v \in V_i$, suppose v is not an anchor node (resp. if v is not a vertex in the initial configuration), let $f(v)$ (resp. $p(v)$) denote the vertex that appears after (resp. before) v in σ. Similarly, for all $v \in V_{i+1}$, where v is not a vertex from the initial configuration (resp. not an anchor node), let $p'(v)$ (resp. $f'(v)$) denote the vertex that appears before (resp. after) v in $G_{\sigma'}$. Let $\{a_1, \ldots, a_k\}$ denote the anchor nodes of G^0_σ and $\{a'_1, a'_2, \ldots, a'_k\}$ denote the anchor nodes of $G_{\sigma'}$. Let $v'_j = p'(a'_j)$ in $G_{\sigma'}$ and $v_j = p(a_j)$ in G^0_σ. Let $Y = \{v_1, \ldots, v_k\}$ and let $Y' = \{v'_1, \ldots, v'_k\}$. We use these notations throughout this section.

Next, consider any edge $(u, v) \in X$. Suppose (u, v) is a forward edge and $v \notin \{a_1, \ldots, a_k\}$. Since (u, v) is in the symmetric difference and since v is not an anchor node, there is a different in-coming forward edge to v, namely (u', v) in $G_{\sigma'}$ with $u \neq u'$. The backward edge (v, u') will be in X and we denote the edge (v, u') as $\text{next}(u, v)$ in G^0_σ. For example,

in Figure 5.3(c), the backward edge (r_9, r_8) is next(r_6, r_9). For the forward edge $(u, v) \in X$, there is a different out-going forward edge (u, v') in $G_{\sigma'}$. This edge appears as the backward edge (v', u) in X provided $v' \notin \{a'_1, a'_2, \ldots, a'_k\}$. Therefore, we define the backward edge (v', u) to be the prev(u, v) in X. For example, in Figure 5.3(c), $(r_3, r_1) = \text{prev}(r_1, r_7)$.

Finally, we define next(u, v) and prev(u, v) for the case where (u, v) is a backward edge in X. Since (u, v) is in the symmetric difference, the edge (v, u) is an out-going forward edge from v in $G_{\sigma'}$. Since (u, v) is in the symmetric difference, there is a different out-going forward edge from v, namely $(v, v') \in X$ with $v' \neq u$. We denote the forward edge (v, v') as next(u, v) in G^0_σ. For instance, in Figure 5.3(c), $(r_1, r_7) = \text{next}(r_3, r_1)$. Similarly, for the backward edge $(u, v) \in X$, if $u \neq r_{i+1}$, there is a different in-coming forward edge to u (v', u) in G_σ. This edge appears as a forward edge (v', u) in X. Therefore, we define the forward edge (v', u) to be the prev(u, v) in X. For instance, in Figure 5.3(c), $(r_3, r_5) = \text{prev}(r_5, i_4)$.

Thus, every edge in X have a unique next$(\cdot, \cdot)$ edge except those that are directed towards an anchor nodes $\{a_1, \ldots, a_k\}$. Edges directed towards the anchor nodes $\{a_1, \ldots, a_k\}$ do not have any next$(\cdot, \cdot)$ edge.

Similarly, every edge in X has a unique prev$(\cdot, \cdot)$, except the edges going out of $\{v'_1, v'_2, \ldots, v'_k\}$. Edges going out of $\{v'_1, v'_2, \ldots, v'_k\}$ do not have any previous edge.

Decomposing X into augmenting trail, alternating trails and cycles: Consider any vertex in $v \in Y \cap Y'$. The forward edge (v, a) is in X. However, since $v \in Y'$, the prev(v, a) does not exist. Similarly, since $a \in \{a_1, \ldots, a_k\}$, next$(v, a)$ does not exist. So, we create a trivial alternating trail with one edge (v, a).

For every vertex $v \in Y' \setminus Y$, let (v, v') be the outgoing edge in X. We initialize T to be (v, v') We construct an alternating trail incrementally by concatenating the last edge (u, v) of T with next(u, v). This construction stops when we reach some edge (u', a) for which

next(u', a) is not defined. Suppose the vertex $v \neq r_{i+1}$, then this trail starts with a forward edge and ends at an anchor node. On the other hand, suppose $v = r_{i+1}$, this trail is an augmenting trail that starts with a backward edge and ends at an anchor node. Any edge (u, v) of X that did not participate in the alternating and augmenting trails have a well defined next(u, v). Therefore, we can construct an alternating cycle that contains (u, v) by repeatedly concatenating the last added edge (u', v') with its next(u', v'). The construction stops when next$(u', v') = (u, v)$ and we get an alternating cycle. Figure 5.3(c) illustrates an example such decomposition. Vertex r_{10} and r_5 are in $Y \cap Y'$, so $\langle r_{10}, a_1 \rangle$ and $\langle r_5, a_3 \rangle$ are trivial trails. Vertex r_6 and r_3 are in $Y' \setminus Y$ and therefore we have two alternating trails $\langle r_6, r_9, r_8, a_2 \rangle$ and $\langle r_3, r_5, i_4, r_4, i_3, r_3, r_1, r_7, r_4, a_4 \rangle$. The vertex r_{11} is r_{i+1}, therefore we have an augmenting trail $\langle r_{11}, r_9, r_{10}, r_7, a_5 \rangle$. All the remaining edges form an alternating cycle $\langle r_2, r_6, i_1, r_2, i_2, r_8, r_2 \rangle$.

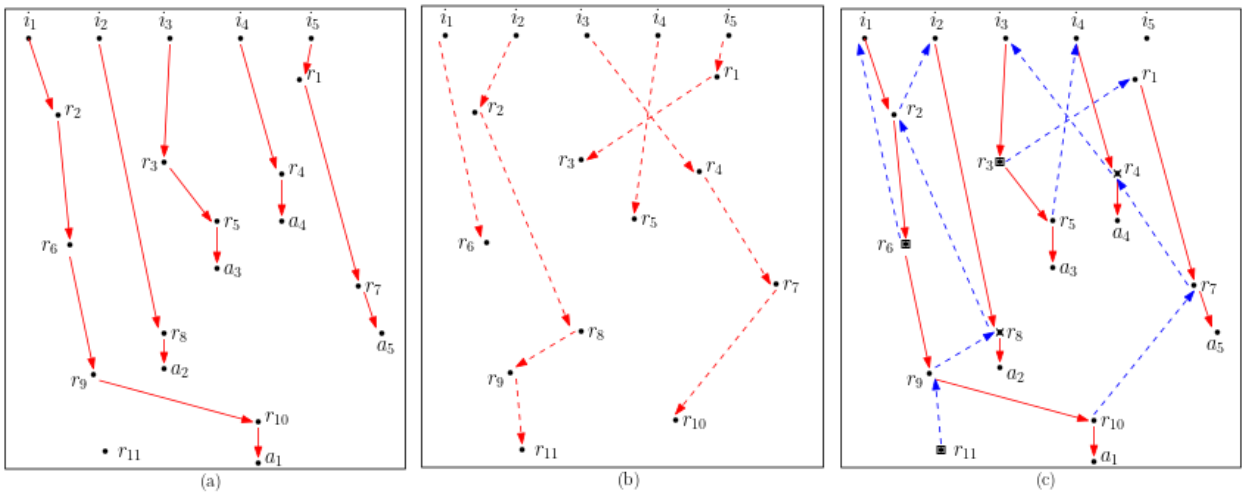


Figure 5.3: (a) Forward edges of G^0_σ, representing σ, (b) Forward edges of $G_{\sigma'}$, representing σ', (c) Edges in the symmetric difference of σ and σ'. Edges of σ are shown as forward edges in G_σ (red edges), and edges of σ' are shown as the backward edges in G_σ (dashed blue edges). This graph has an augmenting trail $\langle r_{11}, r_9, r_{10}, r_7, a_5 \rangle$, two alternating trails $\langle r_6, r_9, r_8, a_2 \rangle$, $\langle r_3, r_5, i_4, r_4, i_3, r_3, r_1, r_7, r_4, a_4 \rangle$ and one directed alternating cycle $\langle r_2, r_6, i_1, r_2, i_2, r_8, r_2 \rangle$

Note that the construction described above also extends to the residual graph $G_{\sigma'}$ and we

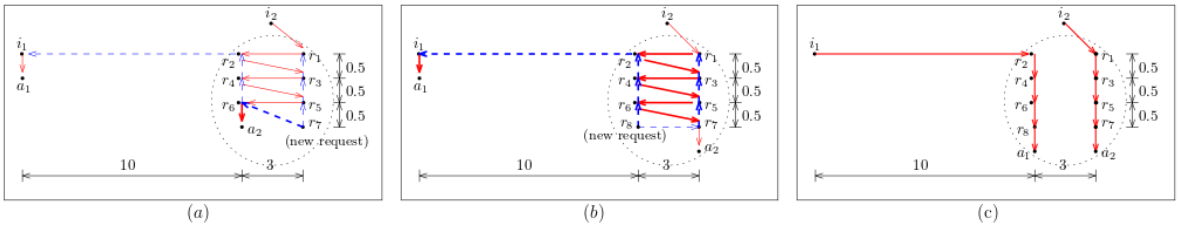


Figure 5.4: Example that highlights the benefit of WFA over Greedy Algorithm

obtain the following lemma.

Lemma 5.12. *The edges of symmetric difference X of two valid solutions σ and σ' in G^0_σ can be decomposed into (a) one augmenting trail A, (b) a set $\mathbb{T}$ of $|Y' \setminus Y| - 1$ non-trivial alternating trails that start with a forward edge, (c) a set $\mathbb{T}'$ of $|Y' \cap Y|$ trivial alternating trails and (d) a set $\mathbb{C}$ of alternating cycles. Similarly, the edges of X in $G_{\sigma'}$ can be decomposed into alternating trails and cycles each of which are obtained by simply applying the flip operation to the trails and cycles in A, $\mathbb{T}$, $\mathbb{T}'$ and $\mathbb{C}$*

Corollary 5.13. *Given the set X of edges in the symmetric difference, consider the decomposition of X into $\{A\} \cup \mathbb{T} \cup \mathbb{C}$ in G^0_σ as described in Lemma 5.12. Then the edges of X in $G_{\sigma'}$ can be decomposed into alternating trails such that for each alternating trail (resp. cycle) $T \in \{A\} \cup \mathbb{T} \cup \mathbb{C}$, there is an alternating trail T' (resp. cycle) induced by the edges of X in $G_{\sigma'}$ such that T' is obtained by applying the flip operation on T and $\Phi(T) = -\Phi(T')$.*

5.4 Truncated Work Function Algorithm

In this section, using the insights from the Work function algorithm, we describe a highly scalable variant called the Truncated Work Function Algorithm (T-WFA). Prior to introducing the algorithm, we illustrate the benefits of the WFA over the greedy algorithm through an example.

In Figure 5.4, we illustrate an example in the two dimensional space equipped with the ℓ_∞-metric, i.e., the distance between two points (x, y) and (x', y') is $\max\{|x - x'|, |y - y'|\}$. In this example, we have two servers and eight requests $\langle r_1, \ldots, r_8 \rangle$ that arrive in increasing order of their index. The eight requests are closely knit and close to server s_2. Server s_1, on the other hand is far away. Initially, for the first seven requests, server s_2 is picked by the Work Function Algorithm. For request r_8, server s_2 has a minimum net-cost augmenting trail $T_2 = \langle r_8, r_7, a_2 \rangle$ with a net-cost Φ_2 of 3 and $w_2 = \Phi_2 + d(r_8, r_7) = 6$. However, the augmenting trail $T_1 = \langle r_8, r_6, r_7, r_5, r_6, r_4, r_5, r_3, r_4, r_2, r_3, r_1, r_2, s_1 \rangle$ has a net-cost $\Phi_1 = -5$ and $w_1 = \Phi_1 + d(i_1, r_8) = 5$ (See Figure 5.4(b)). Therefore, we pull s_1 to the hotspot and serve this request. All future requests in the hotspot will benefit from the two servers close to them.

Since much of the benefits of the Work Function Algorithm seems to accrue from the most recent requests, we propose the following simple modification resulting in the *Truncated Work Function Algorithm*(T-WFA). We choose a parameter $t > 0$ as the *window size* and restrict the algorithm to consider only the last t requests. We further restrict the window to include only those requests that arrived after the *least recently used* server moved to its current location. More precisely, for each anchor node $a_j \in \{a_1, \ldots, a_k\}$, recollect that $p(a_j)$ is the location of the last request that was served by the server with anchor node a_j. Let $\text{ind}(j)$ denote the index of this request in the request sequence R. When $p(a_j)$ is a vertex from the initial configuration, we set $\text{ind}(j)$ to 0. Let, $\text{ind}^* = \min_{a_j \in \{a_1, \ldots, a_k\}} \text{ind}(a_j)$. Note that ind^* also denotes the location of the *least recently used* server. Suppose the current time step is i. Let $i^* = \max\{\text{ind}^*, i - t\}$. The T-WFA will consider the request sequence $R'_i = \langle r_{i^*}, \ldots, r_i \rangle$ to compute the minimum net-cost augmenting trails. Note that the window size will be at most t. However, when all the servers are frequently engaged, the window size becomes substantially smaller.

5.5 Experimental Results

In this section, we compare the performances of the WFA, the T-WFA, the retrospective and greedy algorithms on synthetic and real-world data sets. All algorithms were implemented with Java, and testing code is written in MATLAB which calls the Java code. Our experiment are executed on a machine with 2.1GHz Intel Xeon E5-2683v4 and 64 GB of RAM.

Synthetic Data: We generate 10 different data sets for each value of $N = 3000, 5000, 10000$ and 15000 requests. All requests are drawn from within a unit square. We set the number of servers $k = 10$. Additionally, we also have a *noise rate* nr which is set to $0, 0.2$ or 1 in our experiments. The requests are generated in four groups of $N/4$ requests each. The requests of the first group appear first followed by the requests from groups two, three and four in that order. Within each group, we choose a random location (x, y) as the mean of a normal distribution with $\delta^2 = 0.1$ and generate $(1 - nr)N/4$ many requests (hotspot) from this distribution. The remaining $nr \times N/4$ requests (noise) are chosen uniformly at random from within the unit square. We generate the request sequence by taking a random permutation of requests in this group. When $nr = 0$, all requests are arriving from the hotspots. When $nr = 1$, there is data is generated uniformly at random and there are no hotspots. When $nr = 0.2$, there is a 80% of the requests come from hotspots with 20% requests coming uniformly at random from the unit square.

Experiments on Synthetic Data: Experimental results on synthetic data are provided in Figure 5.5. We compare the performances of WFA, T-WFA, the greedy and retrospective algorithms. The x-axis denotes the value of N and the y-axis denotes the average distance between a request and the server allocated to it. The error bar shows the largest and smallest average distance recorded for the 10 different data sets. First, our experiments suggest no difference in the quality of solution produced by WFA and the T-WFA for the synthetic

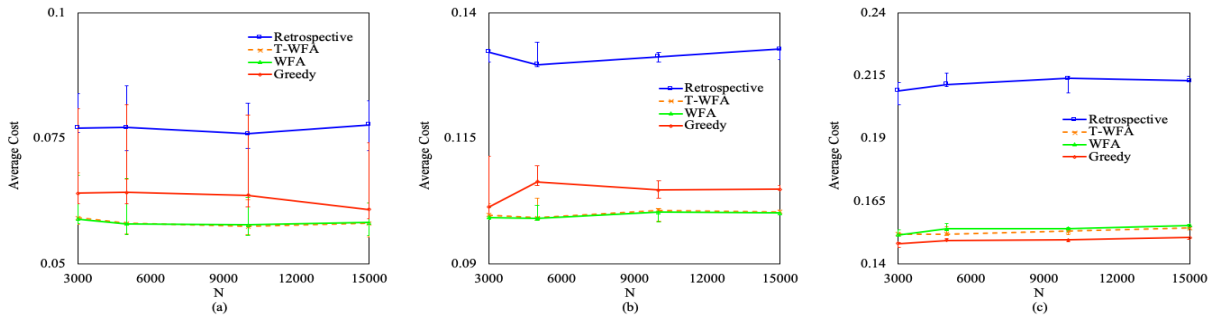


Figure 5.5: Synthetic Data Experimental Results for (a) $nr = 0$, (b) $nr = 0.2$, and (c) $nr = 1$.

.

data sets. For the T-WFA, we set $t = N$. While, WFA takes approximately N^2 steps per request, the T-WFA executes in square of the window size. In our experiments, we observed the average window size to be between 50 and 100 and the max window size to be between 650 and 700. This means that the T-WFA scales substantially better than WFA without any notable loss in accuracy.

Next, when $nr = 0$, the requests are only coming from the hotspots. We observed that the retrospective method does better than greedy. The T-WFA and WFA algorithms tend to outperform retrospective algorithm. When $nr = 0.2$, the greedy algorithm outperforms retrospective algorithm. Again WFA and T-WFA perform better than greedy. Finally, when $nr = 1$, we observe that greedy only marginally outperforms WFA and T-WFA. This is because, when $nr = 1$, the set of requests is uniformly random and there are no emerging patterns in the requests. Consequently, WFA does not benefit from re-organizing the servers. However, occasionally, a few consecutive requests may be generated from a small region making the WFA detect a false pattern causing it to relocate a server to that region. However, this happens infrequently and causes only a slight increase in cost.

Real Data: We use the taxi-calling records from NYC trip record website [41] for the month of January, 2016. Note that taxi requests contain a source and a destination given by their

longitude and latitude. There are about 10 million request records for this month with an average of 320 thousand requests per day. Each trip request contains two locations – a source and a destination. We removed any request with a service time of $< 10s$ and any request with a travel distance greater than 100 kilo-meters. We converted the longitude and latitude of locations to the three dimensional coordinates and used the Euclidean distance between the locations as the distance between them.

Experiments on Real Data: In our experiments, we set $k = 1024$ and the number of requests are approximately $n = 320000$ per day. We extend the WFA and T-WFA in a straight-forward way to handle requests that contain a source and a destination. In this extended implementation, when our algorithm allocates a server to a request, it is assumed that the server will immediately serve the request and move to the destination. The cost of this allocation is simply set to the distance between the server's location (at the time the allocation was made) to the source location of the request. We also implement retrospective and greedy algorithms to process requests in the same manner. Executing the WFA and retrospective methods for one request requires at least one scan of the entire graph, i.e., $\Omega(n^2)$ time. Therefore, $n = 320000$ requests take $\Omega(n^3 + nk^2)$ time (Appendix Lemma 1.18 and Section 1.10) which can take months to process all requests rendering real-time decisions meaningless. Therefore, as proposed in [93], we focus on the last t requests for both WFA as well as the retrospective methods. We conduct two sets of experiments. In our first experiment, we show the affect of window size t on the quality of the decisions made by the WFA. We vary t from 250 to 15000. By having an increased window size of 15000, we are able to reduce the distance between a request and its matched taxi by 21% (vs $t = 250$) and 16% (vs $t = 500$). Increasing window size from $t = 5000$ to $t = 15000$, however, only leads to a marginal 3% improvement in the matching cost suggesting diminishing return (see Figure 5.6) for a significantly increased processing time. This suggests that $t = 5000$ may

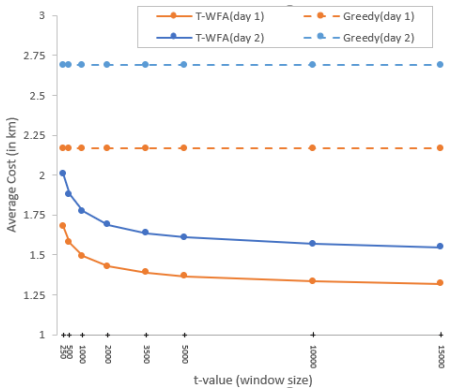


Figure 5.6: Impact of Window size on T-WFA

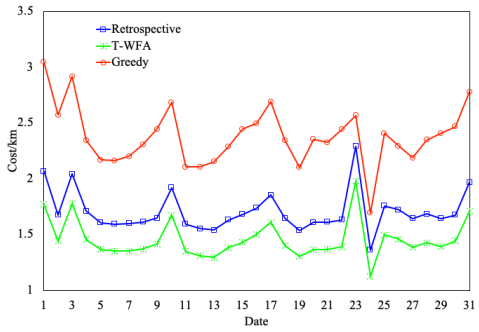


Figure 5.7: NYC Taxi Data Experiment

be sufficient to obtain a large fraction of the benefits of work function algorithm. In the next experiment, we set $t = 5000$ and compare the performance of greedy and retrospective methods for 10 million requests of the month of January, 2016. In Figure 5.7, we compare the performances of T-WFA with our implementation of greedy and retrospective. The x-axis is the day of the month of January 2016. The y-axis is the average allocation distance between the request source and the location of the server allocated to it. The results show that the T-WFA outperforms greedy by 40% and retrospective algorithm by 20%. It is noteworthy that the retrospective algorithm outperforms greedy algorithms for all days.

5.6 Appendix

Organization: In this section, we present proofs of several claims that are discussed in this chapter. In Section 5.6.1, we prove properties of a lazy and valid solution. In Section 5.6.2, we generalize the augment operation and show that this generalized augment operation (also called the flip operation) does not affect the validity of the solution. In Section 5.3.1, we provide a proof for Invariant (I1). In Section 5.3.3, we introduce a few important properties of a symmetric difference of two valid solutions. These properties, along with the flip operation, (Section 5.6.2) are used to prove invariant (I2) (Section 5.3.2). Finally, we discuss implementation details of the WFA for the NY Taxi data set in Section 5.6.5.

5.6.1 Relating Valid solutions to Lazy solutions

Recollect the properties (P1) and (P2)

(P1) For each server s_j, its path Γ_j starts at the location of s_j in the initial configuration. After the first vertex, Γ_j consists of a sequence of requests served by s_j in increasing order of their arrival time. Finally, the last vertex of Γ_j is the location of s_j in the final configuration.

(P2) Every request in R_i participates in exactly one of the k paths.

Lemma 5.14. *Any solution is a lazy valid solution if and only if the paths* $\{\Gamma_1, \ldots, \Gamma_k\}$ *satisfy (P1) and (P2).*

Proof. We begin by showing that a lazy valid solution will satisfy (P1) and (P2). At the start the k servers are in the initial configuration and (P1) and (P2) hold trivially. Inductively assume that i requests have been served and the lazy valid solution and their paths

$\{\Gamma_1, \ldots, \Gamma_k\}$ satisfies (P1) and (P2). When request r_{i+1} arrives, the lazy valid solution will move only one server s to r_{i+1}. Let Γ_j be the path taken by s for the first i requests. Every other path apart from Γ_j do not change and so they satisfy (P1). In addition, Γ_j is extended by one vertex (r_{i+1}) implying that Γ_j satisfies (P1) as well. Inductively, the paths covered the paths cover i requests and Γ_j also covers r_{i+1}. Therefore, (P2) holds.

We also show that any valid solution that satisfies (P1) and (P2) is a lazy valid solution. Let $\{\Gamma_1, \ldots, \Gamma_k\}$ be the k paths that serve all the requests of R and satisfy (P1) and (P2). Let for any $1 \leq i \leq n$, $\Gamma_j(i)$ denote the last request r_t served on the path Γ_j such that $t \leq i$. We set the configuration $\mathcal{C}^i$ to be simply the multiset $\{\Gamma_j(i) \mid 1 \leq j \leq k\}$. Let m be such that Γ_m serves request r_{i+1}. Note that for every $1 \leq j \leq k$, $j \neq m$, by its definition, $\Gamma_j(i) = \Gamma_j(i+1)$. $\Gamma_m(i+1) = r_{i+1}$ and therefore, $\mathcal{C}^{i+1}$ serves request r_{i+1}. Furthermore, for every i, $\mathcal{C}^{i+1} \setminus \mathcal{C}^i = r_{i+1}$ implying that the valid solution is also lazy. □

Recollect the properties (Q1) and (Q2). The set of all forward edges of a residual graph G_i corresponds to a valid solution if and only if

(Q1) The forward edges are directed from an earlier request to a later request, and,

(Q2) Every request r has exactly one incoming forward edge and one outgoing forward edge, every vertex from the initial configuration has exactly one outgoing forward edge and every anchor node has exactly one incoming forward edge.

Proof that (Q1) and (Q2) implies validity: From (Q2), the sub-graph induced by the forward edges are either (i) k paths that start at a vertex of the initial configuration and end at an anchor node, or, (ii) a cycle. Since forward edges are directed from an earlier to a later request only, they cannot participate in a cycle and therefore, the sub-graph induced by the forward edges does not have any cycles. This implies that all the forward edges participate

in the k paths. Each of these paths start at a vertex from the initial configuration, serve a set of requests in increasing order of their arrival time and end at an anchor node, implying (P1). Also, every request will participate in exactly one path, implying (P2). Therefore, every path that satisfies (Q1) and (Q2) also satisfies (P1) and (P2) and is a valid solution.

Lemma 5.15. *A valid solution can be converted to a lazy solution without increasing the cost.*

Proof. In any valid solution, each server s_j will follow a path Γ'_j that starts at a location in the initial configuration, visits several locations and ends at a vertex of the final configuration. Furthermore, each request will lie on one of these k paths. Therefore, (P2) is satisfied. However, in addition to the requests along the path Γ'_j, there may also be other intermediate vertices that do not correspond to any request. We can modify Γ'_j as follows: Let r and r' be two successive requests served by s_j along the path Γ'_j and let $\langle r, i_1, i_2, \ldots, i_t, r' \rangle$ be be the sub-path between r and r'. We remove the edges $(r, i_1), (i_1, i_2), \ldots, (i_t, r')$ from the path and add the edge (r, r') to the path. From triangle inequality, this modification will not increase the total cost of the path Γ'_j. Repeatedly applying this, we can eliminate all intermediate nodes on the paths. The resulting path satisfies (P1) implying that this modified solution is a lazy solution. □

5.6.2 Proofs related to the Augment Operation

Recollect that an augmenting trail is an alternating trail in the extended graph that starts with request r_{i+1} and ends at an anchor node. We described the process of augmentation to simply remove the forward edges from the solution and add the backward edges into the solution (See Section 5.2 for precise details). In this section, we extend this process to certain other alternating trails and cycles are refer to it as the *flip* operation on its edges. We also

prove that after applying the flip operation the solution remains valid. Since, the augment operation is a special case of the flip operation, as a corollary, we can also show that the augment operation preserves validity of the solution.

Recollect the definition of residual graph and alternating trails (given in Section 5.2). First, we define *alternating cycle* in the residual graph to be an alternating trail that starts and ends at the same vertex. Note that any cycle can repeat multiple vertices but does not repeat any edges. Furthermore, there will be an even number of edges in the alternating cycle. This is because the edges of the cycle should alternate between backward and forward edges.

Flip Operation: The flip operation is defined on a trail T that is: (a) An augmenting trail that starts at $r = r_{i+1}$ and ends at an anchor node a, or (b) an alternating trail that starts at some vertex $r \in V_i$, has a forward edge as its first edge and ends with an anchor node, or, (c) an alternating cycle. We describe the operation below:

For any residual graph (resp. extended graph) G_{i+1} (resp. G^0_{i+1}), the flip operation will simply reverse the direction of all the edges on the trail T and relabel the forward edges of T as backward and all the backward edges of T as forward. If T is not a cycle, the flip operation will remove the incoming forward edge to the anchor node a, add a new forward edge from r to the anchor node a, and update the location of a to that of r.

Note that when executed on an augmenting trail, the flip operation is exactly the same as an augment operation. Also note that if T is a single edge (r, a) then the flip operation will not modify the valid solution σ. For this reason, we refer to alternating trails containing only one edge as *trivial* and the other alternating trails as *non-trivial*. Note that the definition of net-cost extends to alternating cycles and alternating trails as well. We define the net-cost

of any alternating trail or an alternating cycle T to be

$$\Phi(T) = \sum_{(u,v) \text{ is backward}} d(u,v) - \sum_{(u,v) \text{ is forward}} d(u,v).$$

The next lemma shows that the flip operation produces a valid solution.

Lemma 5.16. *Let σ_i be the current valid solution with a configuration identical to $\mathcal{C}^i$ as its final configuration. For any alternating trail T that starts at r, has a forward edge as its first edge and ends with an anchor node a, let $p(a)$ be the node prior to a on T. The flip operation will produce a new valid solution σ with the location of the final configuration being identical to $\mathcal{C}^i \setminus \{p(a)\} \cup \{r\}$. The cost of the new solutions $w(\sigma) = w(\sigma_i) + \Phi(T)$.*

Proof. Backward edges are directed from a later request to an earlier one. After the flip operation, a backward edge becomes a forward edge and reverses its direction. As a result, the newly introduced forward edge will be directed from an earlier request to a later one implying that (Q1) holds. Next, we show that (Q2) holds.

Since the first vertex of T is r and its last vertex is an anchor node a and its first and last edges are a forward edges, there are an odd number of edges along the trail. Let the trail T be the sequence $T = \langle f_1, e_1, f_2, \ldots, f_{t-1}, e_{t-1}, f_t \rangle$ of edges where $f_i = (b_i, a_i)$ and $e_i = (a_i, b_{i+1})$. Consequently, we can also write the alternating trail $T = \langle b_1 = r, a_1, b_2, \ldots, a_{t+1} = a \rangle$ as a sequence of vertices.

The alternating trail starts at r and ends at an anchor node a. We begin by showing that (Q2) holds for every node v on T where $v \notin \{r, a\}$. After that, we show (Q2) also holds for the vertices r and a.

(Q2) holds for b_j for $j > 1$: For $1 < j \leq t+1$, any vertex b_j has an outgoing forward edge (b_j, a_j) and an incoming backward edge (a_{j-1}, b_j) on the trail T. After the flip operation,

the edge (b_j, a_j) is removed from the solution and (b_j, a_{j-1}) is added as a forward edge to the solution. Consequently, the number of out-going forward edge from b_j remains unchanged and (Q2) holds for every b_j except b_1

(Q2) holds for a_j for $j < t+1$: For any $1 \leq j < t+1$, a_j has an incoming forward edge (b_j, a_j) and an out-going backward edge (a_j, b_{j+1}) prior to the flip operation. After the flip operation, the incoming forward edge (b_j, a_j) is removed from the solution and another incoming forward edge (b_{j+1}, a_j) is added to the solution. Therefore, the incoming forward edge to a_j remains unchanged and (Q2) holds for every a_j except a_{t+1}.

(Q2) holds for $b_1 = r$ and $a_{t+1} = a$: Finally, note that for b_1, we remove the out-going forward edge (b_1, a_1) from the solution and the in-coming forward edge (b_{t+1}, a_{t+1}) from the solution. However, we add the forward edge from $b_1 = r$ to $a_{t+1} = a$. Therefore, the number of incoming forward edges to a_{t+1} and the number of out-going forward edges to b_1 remains unchanged.

Note that any anchor node has only one in-coming forward edge and no backward edge going out of it. Therefore, any alternating trail that enters an anchor node cannot leave it. Therefore, the alternating trail T does not contain any anchor node except the end vertex a. Let $v = p(a)$ prior to the flip operation. After the flip operation, $p(a)$ changes from v to r and the new configuration is $\mathcal{C}' = \mathcal{C}^i \setminus \{v\} \cup \{r\}$.

Finally, the flip operation will remove the forward edges and add the backward edges to the solution. Therefore, the change in cost of the solution is precisely $\Phi(T)$. So, we conclude $w(\sigma') = w(\sigma) + \Phi(T)$. □

We can extend the claim to augmenting paths and alternating cycles as well. We present the slightly modified proof for augmenting paths next.

Lemma 5.17. *Let σ_i be the current valid solution with $\mathcal{C}^i$ also as its final configuration. For*

any augmenting trail T that starts at $r = r_{i+1}$ and ends at an anchor node a, let $p(a)$ be the vertex that appears prior to a on T. The flip operation will produce a new valid solution σ that serves $i+1$ requests and has a final configuration $\mathcal{C}^i \setminus \{p(a)\} \cup \{r_{i+1}\}$. The cost of the new solution σ is $w(\sigma) = w(\sigma_i) + \Phi(T)$.

Proof. Backward edges are directed from a later request to an earlier one. After the flip operation, a backward edge becomes a forward edge and reverses its direction. As a result, the newly introduced forward edge will be directed from an earlier request to a later one implying that (Q1) holds. Next, we show that (Q2) holds.

Since the first vertex of T is r_{i+1} and its last vertex is an anchor node a and its first edge is a backward edge and the last edge is a forward edges, there are an even number of edges along the trail. Let the trail T be the sequence $T = \langle e_1, f_2, \ldots, f_{t-1}, e_{t-1}, f_t \rangle$ of edges where $e_i = (b_i, a_i)$ and $f_i = (a_i, b_{i+1})$ are backward and forward edges respectively. Consequently, we can also write the alternating trail $T = \langle b_1 = r_{i+1}, a_1, b_2, \ldots, a_{t+1} = a \rangle$ as a sequence of vertices.

The augmenting trail starts at r and ends at an anchor node a. We begin by showing that (Q2) holds for every node v on T where $v \notin \{r_{i+1}, a\}$. After that, we show (Q2) also holds for the vertices r_{i+1} and a.

(Q2) holds for b_j for $j > 1$: For $1 < j \leq t+1$, any vertex b_j has an outgoing backward edge (b_j, a_j) and an incoming forward edge (a_{j-1}, b_j) on the trail T. After the flip operation, the edge (a_{j-1}, b_j) is removed from the solution and (a_j, b_j) is added as a forward edge to the solution. Consequently, the number of in-going forward edge to b_j remains unchanged and (Q2) holds for every b_j except b_1

(Q2) holds for a_j for $j < t+1$: For any $1 \leq j < t+1$, a_j has an incoming backward edge (b_j, a_j) and an out-going forward edge (a_j, b_{j+1}) prior to the flip operation. After the

flip operation, the outgoing forward edge (a_j, b_{j+1}) is removed from the solution and another outgoing forward edge (a_j, b_j) is added to the solution. Therefore, the outgoing forward edge from a_j remains unchanged and (Q2) holds for every a_j except a_{t+1}.

(Q2) holds for $b_1 = r_{i+1}$ and $a_{t+1} = a$: Finally, note that there were no in-coming forward edge to $b_1 = r_{i+1}$ prior to the flip operation. In the flip operation, we add an incoming forward edge (a_1, b_1) to b_1 and remove an in-coming forward edge $(b_{t+1}, a_{t+1}$ to a_{t+1}. Instead, we add a forward edge from b_1 to a_{t+1}. As a result, $b_1 = r_{i+1}$ has exactly one in-coming and one out-going forward edge and satisfies (Q2). The anchor node $a_{t+1} = a$ has exactly one in-coming anchor node satisfying (Q2).

Note that any anchor node has only one in-coming forward edge and no backward edge going out of it. Therefore, any alternating trail that enters an anchor node cannot leave it. As a result the augmenting trail T does not contain any anchor node except the end vertex a. Let $v = p(a)$ prior to the flip operation. After the flip operation, $p(a)$ changes from v to r and the new configuration is $\mathcal{C}' = \mathcal{C}^i \setminus \{v\} \cup \{r\}$.

Finally, the flip operation will remove the forward edges and add the backward edges to the solution. Therefore, the change in cost of the solution is precisely $\Phi(T)$. So, we conclude $w(\sigma') = w(\sigma) + \Phi(T)$. □

Alternating cycles do not have an anchor node and a start vertex. Therefore, using arguments identical to the ones used in previous lemma, we can prove the following for alternating cycles.

Lemma 5.18. *Let σ_i be the current valid solution with $\mathcal{C}^i$ also as its final configuration. For an alternating cycle C, the flip operation will produce a valid solution with a final configuration that is identical to $\mathcal{C}^i$. The cost of the new solution σ is $w(\sigma) = w(\sigma_i) + \Phi(C)$.*

5.6.3 Demonstration of the Algorithm

We will demonstrate the execution of the work-function algorithm by using an example with 15 requests, $\langle r_1, \ldots, r_{15} \rangle$. We use two servers s_1 and s_2 in this example. Let the nodes at the initial locations of s_1 and s_2 be i_1 and i_2 respectively. The anchor nodes corresponding to the two servers are referred as a_1 and a_2. This example is constructed in the Euclidean plane with an origin and x and y axis, i.e., the distance $d(p, q)$ between two points p and q is the straight line distance between the locations of p and q. We will describe the graph after processing each request. For describing this example, we will refer to the forward edges as red edges and the backward edges as blue edges. We also use the colors red and blue in the figures to show forward and backward edges respectively. Recollect that the location of any anchor node is same as the coordinates of the node it is connected to by a red edge. However, for readability of the figure, we show the anchor node is slightly displaced from its original position. For any extended graph after the arrival of a new request r, we describe the values of four variables ϕ_1, ϕ_2, d_1 and d_2. ϕ_1 refers to the net-cost of the minimum net-cost path from r to a_1, ϕ_2 refers to the net-cost of the minimum net-cost path from r to a_2, d_1 refers to the direct distance from r to a_1 and d_2 refers to the direct distance from r to a_2.

Initial graph. The initial graph before any request is served contains 2 nodes i_1 and i_2 representing the initial server locations. This graph also contains the two anchor nodes a_1 and a_2 connected to i_1 and i_2 respectively by a red edge.

Arrival of first request r_1. Figure 5.8(a) shows the extended graph G_1^0 when the first request arrives. Since, $\phi_1 + d_1 < \phi_2 + d_2$, server s_1 moves from i_1 to serve r_1. We augment along the path $< r_1, i_1, a_1 >$ and the red edge in figure 5.8(b) shows the solution we maintain after r_1 is served.

Arrival of the second request r_2. Figure 5.8(b) shows the extended graph G_2^0 when the second request arrives. Since, $\phi_1 + d_1 < \phi_2 + d_2$, server s_1 moves to serve r_2. The red edge in figure 5.8(c) shows the solution we maintain after r_2 is served.

Arrival of the third request r_3. Figure 5.8(c) shows the extended graph G_3^0 when the third request arrives. Since, $\phi_1 + d_1 < \phi_2 + d_2$, server s_1 moves to serve r_3. The red edge in figure 5.8(d) shows the solution we maintain after r_3 is served.

Arrival of the fourth request r_4. Figure 5.8(d) shows the extended graph G_4^0 when the fourth request arrives. Since, $\phi_2 + d_2 < \phi_1 + d_1$, server s_2 moves from i_2 to serve r_4. We augment along the path $< r_4, i_2, a_2 >$ and the red edge in figure 5.8(e) shows the solution we maintain after r_4 is served.

Arrival of the fifth request r_5. Figure 5.8(e) shows the extended graph G_5^0 when the fifth request arrives. Since, $\phi_2 + d_2 < \phi_1 + d_1$, server s_2 moves to serve r_5. We augment along the path $< r_5, r_4, a_2 >$ and the red edge in figure 5.8(f) shows the solution we maintain after r_5 is served.

Arrival of the sixth request r_6. Figure 5.8(f) shows the extended graph G_6^0 when the sixth request arrives. The shortest augmenting path from r_6 to a_1 is $< r_6, r_4, r_5, r_3, a_1 >$ having net-cost of 5.1 and the shortest augmenting path from r_6 to a_2 is $< r_6, r_5, a_2 >$ with net-cost of 5.4. Here, $d_1 = 5.4$ and $d_2 = 5.4$. Since $\phi_1 + d_1 < \phi_2 + d_2$, server s_1 moves to the position of r_6. But since the chosen augmenting path is not trivial, the augmentation process will change some of the existing edges color and direction and therefore, the solution we maintain will rearrange the solution such that r_6 lies in the path followed by server s_2. Figure 5.8(g) shows the graph after augmenting along the shortest net-cost path $<$

$r_6, r_4, r_5, r_3, a_1 >$ from r_6 to a_1.

Arrival of the seventh request r_7. Figure 5.8(g) shows the extended graph G_7^0 when the seventh request arrives. The shortest augmenting path from r_7 to a_1 is the trivial path $< r_7, r_5, a_1 >$ with net-cost equal to d_1 and the shortest augmenting path from r_7 to a_2 is also the trivial path $< r_7, r_5, a_2 >$ with net-cost equal to d_2. Since, $\phi_2 + d_2 < \phi_1 + d_1$, server s_2 moves to serve r_7. The red edge in figure 5.8(h) shows the solution we maintain after r_7 is served.

Arrival of the eighth request r_8. Figure 5.8(h) shows the extended graph G_8^0 when the eighth request arrives. Please note that for clarity of the graph, we have only shown those blue (or backward) edges of G_8^0 that participate in the shortest augmenting path from r_8 to any anchor node. The shortest augmenting path from r_8 to a_1 is $< r_8, r_5, a_1 >$ and the shortest augmenting path from r_8 to a_2 is $< r_8, r_7, a_2 >$. Since, $\phi_2 + d_2 < \phi_1 + d_1$, server s_2 moves to serve r_8. The red edge in figure 5.8(i) shows the solution we maintain after r_8 is served.

Arrival of the ninth request r_9. Figure 5.8(i) shows the extended graph G_9^0 when the eighth request arrives. Please note that for clarity of the graph, we have only shown those blue (or backward) edges of G_9^0 that participate in the shortest augmenting path from r_9 to any anchor node. The shortest augmenting path from r_9 to a_1 is $< r_9, r_5, a_1 >$ and the shortest augmenting path from r_9 to a_2 is $< r_9, r_8, a_2 >$. Since, $\phi_2 + d_2 < \phi_1 + d_1$, server s_2 moves to serve r_9 and the path $< r_9, r_8, a_2 >$ is augmented in our maintained solution.

Processing requests r_{10} to r_{14}. The location of requests $r_{10}, r_{11}, r_{12}, r_{13}$ and r_{14} is shown in figure 5.8(j). For each of these requests from r_{10} to r_{14}, we get $\phi_2 + d_2 < \phi_1 + d_1$. Therefore,

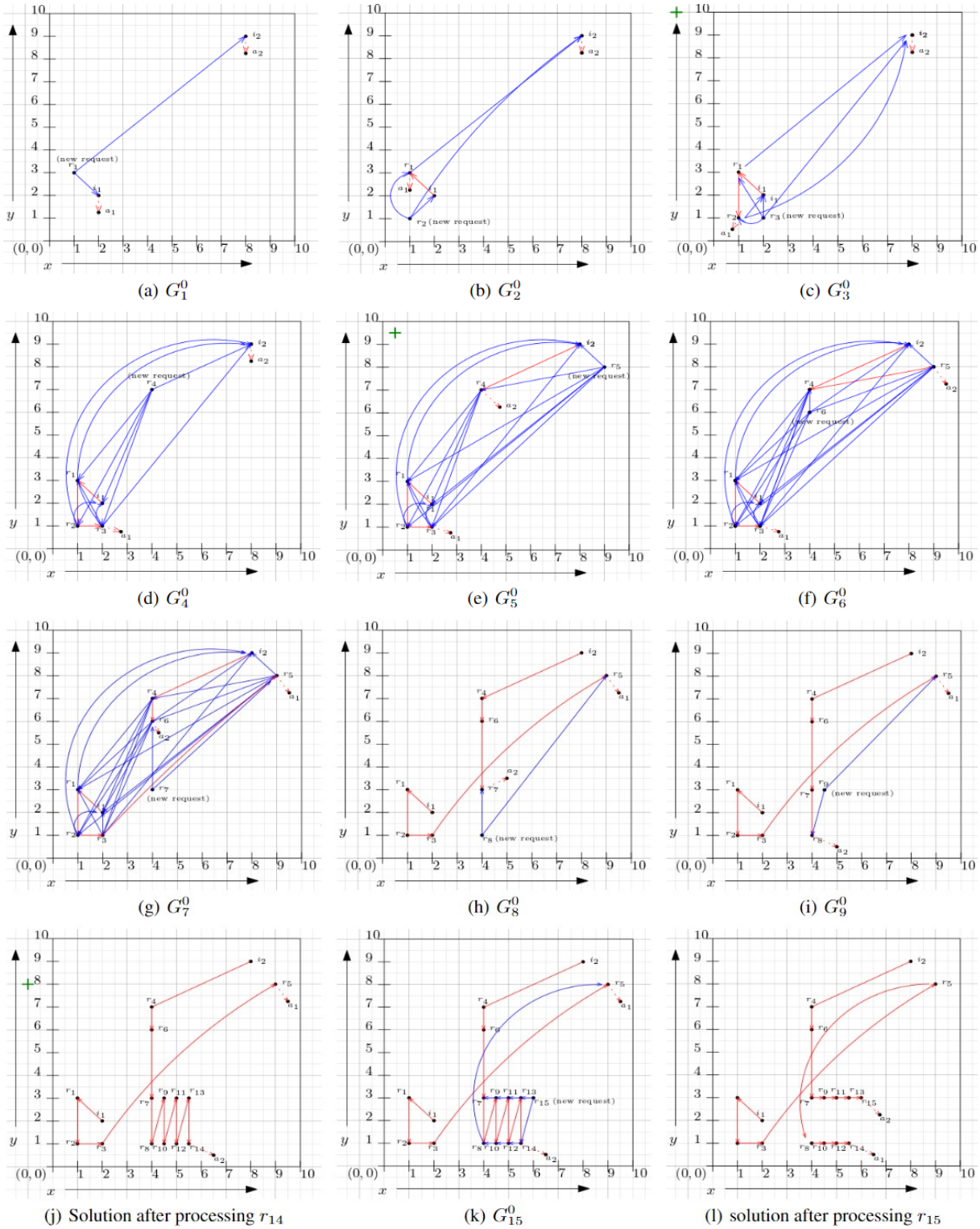
server s_2 serves all the requests from r_{10} to r_{14} in sequence and the solution we maintain after serving the fourteenth request r_{14} is sown in figure 5.8(j).

Arrival of the fifteenth request r_{15}. Figure 5.8(k) shows the extended graph G^0_{15} when the fifteenth request arrives. Please note that for clarity of the graph, we have only shown those blue (or backward) edges of G^0_{15} that participate in the shortest augmenting path from r_{15} to any anchor node. The shortest net-cost augmenting path from r_{15} to a_1 is $< r_{15}, r_{13}, r_{14}, r_{12}, r_{13}, r_{11}, r_{12}, r_{10}, r_{11}, r_9, r_{10}, r_8, r_9, r_7, r_8,$
$r_5, a_1 >$. The net-cost of this path is, $\phi_1 = d(r_{15}, r_{13}) - d(r_{13}, r_{14}) + d(r_{14}, r_{12}) - d(r_{12}, r_{13}) + d(r_{13}, r_{11}) - d(r_{11}, r_{12}) + d(r_{12}, r_{10}) - d(r_{10}, r_{11}) + d(r_{11}, r_9) - d(r_9, r_{10}) + d(r_{10}, r_8) - d(r_8, r_9) + d(r_9, r_7) - d(r_7, r_8) + d(r_8, r_5) - d(r_5, a_1) = 8.6 - 10.8 = -2.2$. And the shortest net-cost augmenting path from r_{15} to a_2 is $< r_{15}, r_{14}, a_2 >$ having the net cost $\phi_2 = 2.1$. Here, $d_1 = d(r_{15}, a_1) = 5.8$ and $d_2 = d(r_{15}, a_2) = 2.1$. Since, $\phi_1 + d_1 < \phi_2 + d_2$, server s_1 moves to serve r_{15}. We augment along the shortest augmenting path from r_{15} to a_1 and Figure 5.8(l) shows the solution we maintain after r_{15} is served.

In the above example, it can be seen that requests r_7 to r_{15} were arriving near to each forming a hot-spot. Since server s_2 was present in the newly forming hot-spot, it served the requests from r_7 to r_{14}. Then when request r_{15} arrived in the same hot-spot, our algorithm detected the pattern and pulled the server s_1 to serve r_{15} and hence pulling it in the emerging hot-spot for an efficient utilization of available servers.

5.6.4 Retrospective Algorithm

Below, we present a proof to show that retrospective algorithm is a lazy algorithm and can be implemented in $\mathcal{O}(n^2)$ time per request.



(a) G_1^0 (b) G_2^0 (c) G_3^0

(d) G_4^0 (e) G_5^0 (f) G_6^0

(g) G_7^0 (h) G_8^0 (i) G_9^0

(j) Solution after processing r_{14} (k) G_{15}^0 (l) solution after processing r_{15}

Figure 5.8: Demonstration of the Algorithm with an Example

.

Lemma 5.19. *Retrospective algorithm is a lazy algorithm and can be implemented in $\mathcal{O}(n^2)$ time per request.*

Proof. Let $\sigma = \sigma_i^*$ (resp. $\sigma' = \sigma_{i+1}^*$) be the optimal solution for the first i (resp. $i+1$) requests. If there are multiple minimum-cost solution for processing the first $i+1$ requests, we choose σ' to be a minimum-cost valid solution whose symmetric difference with σ has the fewest edges. Let $\mathcal{C}$ (resp. $\mathcal{C}'$) be the final configurations of σ (resp. σ'). To show that retrospective is a lazy algorithm, we will argue that the symmetric difference of σ and σ' contains one augmenting trail A and it does not contain any alternating trails or cycles. Suppose, for the sake of contradiction, in addition to the augmenting trail A, let there also be an alternating trail T in the symmetric difference of σ and σ' with respect to the extended graph G_σ^0. The net-cost of T cannot be zero, since otherwise, from Corollary 5.13, there is an alternating trail T' with net-cost 0 and applying the flip operation along T' in $G_{\sigma'}$ will lead to a valid solution with a cost equal to that of σ' (Lemma 5.16) and whose symmetric difference with σ is smaller leading to a contradiction.

Next, we show that T cannot have a negative net-cost. Otherwise, we can apply the flip operation along T on the extended graph G_σ^0 to produce a valid solution that serves i requests and has a smaller cost than σ (Lemma 5.16) leading to a contradiction.

Finally, we show that T cannot have a positive net-cost either. This is because, from Corollary 5.13, there is an alternating trail T' in the residual graph $G_{\sigma'}$ such that $\Phi(T') = -\Phi(T) < 0$. This implies that if we apply the flip operation along T' in the residual graph $G_{\sigma'}$, we will get another valid solution with a smaller cost (Lemma 5.16), contradicting the fact that σ' is the minimum-cost solution to serve the first $i+1$ requests. This implies that the symmetric difference of σ and σ' consists of exactly one augmenting trail and no non-trivial alternating trails. Using a similar argument, we can also show that the symmetric difference does not

contain any alternating cycles. Therefore, we can conclude that the symmetric difference of σ and σ' is exactly one augmenting trail A.

Let this augmenting trail A be from r_{i+1} to some anchor node a. Then, the retrospective algorithm will move the server located at a to the request r_{i+1} implying that the algorithm is a lazy algorithm.

We also present a simple implementation of the retrospective algorithm. Recollect that Φ_j is the minimum net-cost path from r_{i+1} to a_j. We claim that the augmenting trail A chosen has a net-cost $\Phi(A) = \min_{a_j \in \{a_1,\ldots,a_k\}} \Phi_j$. For the sake of contradiction, assume $\Phi(A) > \min_{a_j \in \{a_1,\ldots,a_k\}} \Phi_j$. Let $\Phi_q = \min_{a_j \in \{a_1,\ldots,a_k\}} \Phi_j$, a_q be the corresponding anchor node and T' be the augmenting path from r_{i+1} to a_q with a net-cost of Φ_q. Augmenting σ along T' gives a valid solution σ'' (Lemma 5.17). The cost of σ'', $w(\sigma'') = w(\sigma) + \Phi(T') < w(\sigma) + \Phi(A) = w(\sigma')$. This implies that $w(\sigma'') < w(\sigma')$ reaching a contradiction to the assumption that σ' is the smallest cost solution to serve $i+1$ requests.

Using this observation, we modify the description of work function algorithm only slightly and obtain the retrospective algorithm: While processing request r_{i+1}, Step 3 will pick the anchor node a_j and the augmenting path from r_{i+1} to a_j such that the net-cost of this path Φ_j is the smallest among all $1 \leq j \leq k$. One can modify Step 3 of the algorithm to achieve this. The rest of the algorithm remains unchanged and the algorithm continues to have an execution time of $\mathcal{O}(n^2 + k^2)$ per request. □

5.6.5 WFA Implementation Details for Taxi Requests

For our experiments on NYC taxi data, each request contains a pickup location $\text{src}(r)$ and a drop-off location $\text{dest}(r)$. In this section, we explain the modifications to our implementation of WFA that allows us to handle such source-destination requests.

(a) (b)

Figure 5.9: Comparing T-WFA and WWFA w.r.t. (a) Average Window Size and (b)Average Cost for different t-values

Each vertex (except the vertices of the initial configuration and the anchor nodes) represent requests and have two associated locations for them, i.e., the pickup location $\text{src}(v)$ and the drop-off location $\text{dest}(v)$. For any vertex v in the initial configuration, we set $\text{src}(v) = \text{dest}(v)$ to be the location associated with this vertex. When v is an anchor node, we set $\text{src}(v) = \text{dest}(v)$ to be $\text{dest}(p(v))$, where $p(v)$ is the vertex that appears before v in the solution.

In the residual graph and the extended graph, for any forward edge (u, v), we define its cost $d(u, v)$ as the distance between $\text{dest}(u)$ and $\text{src}(v)$. For any backward edge (u, v) directed from u to v, we define its cost $d(u, v)$ as the distance between $\text{src}(u)$ and $\text{dest}(v)$. The rest of the algorithm remains unchanged.

5.6.6 Additional Experimental Results

In this section, we present an additional experiment on the real data set. We maintain the same experimental setup for the real data as in our experimental results (Section 5.6.5).

The windowed version of the WFA (Rudec,Baumgartner, and Manger 2013) with window size t is an algorithm that executes the Work Function Algorithm but only on the last t requests. We execute our faster implementation of the WFA on the last t requests in $\mathcal{O}(t^2 + k^2)$ time

and refer to it as the Windowed WFA (WWFA).

The T-WFA we proposed is slightly different from WWFA since, among the last t requests, T-WFA considers only those requests that arrived after the least recently used server moved to its current location. Therefore, the average window size of T-WFA can be smaller than t.

In our first experiment, we set the number of servers $k = 10$ and plot the ratio of the average window size of T-WFA to the window size of WWFA (See Figure 5.9a). Figure 5.9b compares the cost of the assignments made by T-WFA and WWFA. Our experiments suggest that T-WFA and WWFA incur almost identical costs (See figure 5.9b). The average window size of T-WFA, however, is significantly smaller than that of WWFA (See figure 5.9a) which means that T-WFA processes the request significantly faster than WWFA.

When we increase the number of servers to $k = 1024$, the window size of T-WFA does not reduce and remains identical to that of the WWFA, even when $t = 15000$. Due to a large number of servers being available, we have at least one server that does not serve any of the last $t = 15,000$ requests. Therefore, the least recently used server remains outside the window of 15000. As a result, the T-WFA will have a window size identical to that of the WWFA and the assignment cost and processing time of T-WFA and WWFA become identical.

Chapter 6

Divide and Conquer Algorithms for the Taxi Routing Problem

In this chapter, we adapt our algorithms for the k-first come first served routing problem (k-FCFSRP) to solve the taxi allocation problem (TAP).

We begin by showing that the optimal solution to the k-FCFSRP can be computed by solving an instance of the minimum-cost bipartite matching problem.

6.1 Reduction to Bipartite Matching

To reduce the k-FCFSRP to the minimum-cost maximum bipartite matching problem, we first construct a graph representation of the instance of the k-FCFSRP. We then show that any minimum-cost maximum matching on this graph corresponds to an optimal solution for the k-FCFSRP.

Constructing the graph representation We construct a bipartite graph defined over two point sets A and B representing the instance of the k-FCFSRP, where each vertex in the set A (resp. B) can be interpreted as the departure (resp. arrival) gate of a request. More formally, given a sequence of n requests $\mathcal{R} = \langle r_1, \ldots, r_n \rangle$ and a set of k servers $\mathcal{S} = \{s_1, \ldots, s_k\}$, we construct a bipartite graph $\mathcal{G} = (A \cup B, E)$ as follows. For each request

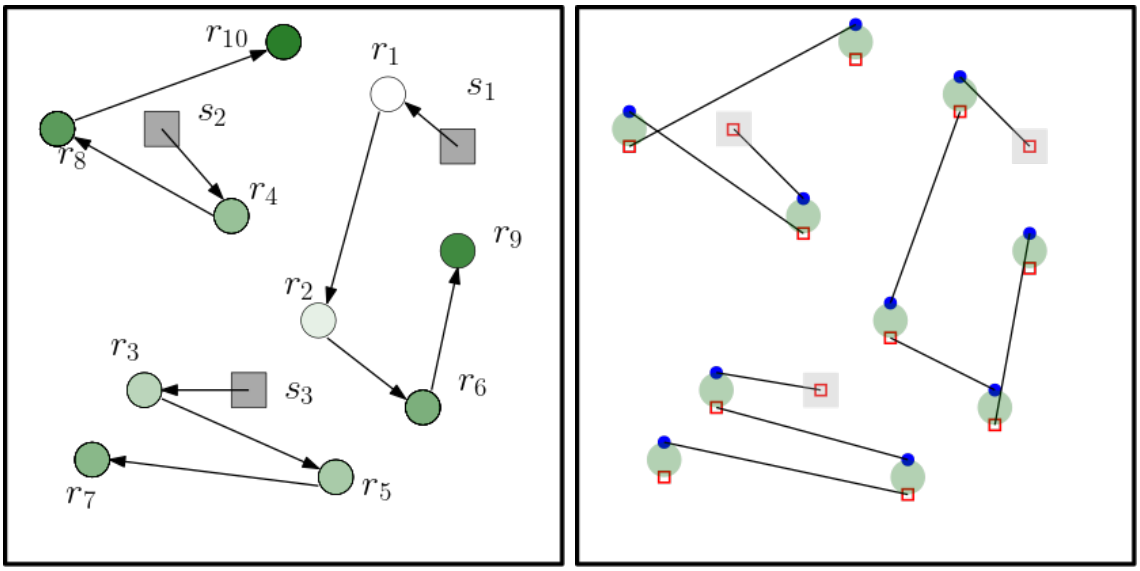


Figure 6.1: (Left) A valid solution (black arrows) for the k-FCFSRP and (right) corresponding maximum matching on the graph representation of the problem.

$r \in \mathcal{R}$, we add points a_r and b_r at the location of r to A and B, respectively. Furthermore, for each server $s \in \mathcal{S}$, we add a point a_s at the location of s to A.

For any two requests r_i and r_j, where $i < j$, we add the edge (a_{r_i}, b_{r_j}) to E. Additionally, for each server $s \in \mathcal{S}$ and each request $r \in \mathcal{R}$, we add the edge (a_s, b_r) to E. This completes the construction of the graph $\mathcal{G}$. Note that the set A contains $n + k$ points, and the set B contains n points.

Relating maximum matchings to a valid solution of k-FCFSRP. Next, we show that any valid solution to a k-FCFSRP instance uniquely maps to a maximum matching on $\mathcal{G}$ with the same cost and vice versa. We use this observation to show that k-FCFSRP can be reduced to the problem of computing a minimum-cost maximum matching on $\mathcal{G}$.

Intuitively, as noted above, the points in A represent the departure gate of a request, whereas the points in B exemplify the arrival gate of a request. Therefore, a possible movement of a server from a request r_i to a request r_j is represented in $\mathcal{G}$ by the edge (a_{r_i}, b_{r_j}). Similarly, for

any server s, a possible movement of s from its initial location to a request r is represented by the edge (a_s, b_r). One can use this observation to construct a matching M_σ for each valid solution σ with the same cost by simply collecting the edges of $\mathcal{G}$ representing the server movements in σ (Lemma 6.1). One can also use a similar construction to show how a maximum matching M can be transformed into a valid solution σ with the same cost (Lemma 6.2).

Lemma 6.1. *Computing an optimal solution for the k-FCFSRP reduces to computing a minimum-cost maximum matching in the graph representation $\mathcal{G}$.*

Proof. Consider any valid solution σ to the k-FCFSRP. We construct a matching M_σ representing σ as follows. For any server $s \in \mathcal{S}$, if the routing plan of s is $\Gamma(s) = \langle r_{i_1}, \ldots, r_{i_t} \rangle$, then add the edge $(a_s, b_{r_{i_1}})$ and for each $1 \leq j < t$ add the edge $(a_{r_{i_j}}, b_{r_{i_{j+1}}})$. Since each request in $\mathcal{R}$ is served by exactly one server according to σ, exactly one server arrives at each request, and therefore, each point $b \in B$ is incident to exactly one edge in M_σ; similarly, at most one server departs each request location and therefore, each point $a \in A$ is incident to at most one edge of M_σ. Hence, M_σ is a matching of size n (i.e., a maximum matching), which, by the definition of the cost of the routing plans and matchings, has the same cost as σ.

Next, given a maximum matching M on $\mathcal{G}$, we construct a valid solution σ_M with the same cost. Our construction relies on the following observation: since M is a maximum matching, all points in B are matched in M. For each server $s \in \mathcal{S}$, we construct the sequence of requests that need to be served by s by following the matching edges as described next. We construct a graph $\mathcal{G}' = (A \cup B, E')$, where we add an edge (a, b) to E' for each matching edge $(a, b) \in M$. Additionally, for each request $r \in \mathcal{R}$, we add the edge (b_r, a_r) to E'. For each server $s \in \mathcal{S}$, let $\sigma_M(s)$ denote the sub-sequence of all requests $r \in \mathcal{R}$ where b_r is reachable from s in $\mathcal{G}'$.

Note that $\mathcal{G}'$ is a bipartite graph, where (i) each point $b \in B$ has a degree exactly 2, (ii) for each server $s \in \mathcal{S}$, the vertex a_s has a degree at most 1, and (iii) for each request $r \in \mathcal{R}$, the degree of a_r is either 1 or 2. Since M is a matching of size n, the graph $\mathcal{G}'$ consists of k disjoint paths (possibly of length 0), where each path starts at a vertex a_s for a server $s \in \mathcal{S}$ and ends at a vertex $a \in A$, and each point $b \in B$ lies in exactly one such path. Therefore, each point $b \in B$ is reachable from exactly one server and each request r is included in exactly one set $\sigma_M(s)$ for a unique server $s \in \mathcal{S}$. Furthermore, for each consecutive pair of requests $(r_{i_1}, r_{i_2}) \in \sigma_M(s)$, since the edge $(a_{r_{i_1}}, b_{r_{i_2}})$ exists in $\mathcal{G}$, the pair (r_{i_1}, r_{i_2}) is feasible; hence, each sub-sequence $\sigma_M(s)$ for each server $s \in \mathcal{S}$ is feasible and σ_M is a feasible routing plan. Finally, by the definition of the matching cost as well as the cost of the routing plans, it can be easily confirmed that M and σ_M have the same cost. □

This completes our proof that the k-FCFSRP can be reduced to the minimum-cost maximum bipartite matching problem.

The above reduction can be extended in a straightforward way to show the TAP can also be reduced to the minimum-cost maximum bipartite matching problem. To show this reduction, we adjust the construction of the graph representation of the instance of the k-FCFSRP to describe the graph representation of an instance of the TAP. All remaining claims and proofs for the reduction remain unchanged and can be adopted as is.

Constructing the graph representation $\mathcal{G}'$ for the TAP. For constructing the bipartite graph defined over two point sets A and B to represent the TAP, each vertex in the set A (resp. B) can be interpreted as the dropoff location(resp. pickup location) of a request. More formally, given a sequence of n requests $\mathcal{R} = \langle r_1, \ldots, r_n \rangle$ and a set of k servers $\mathcal{S} = \{s_1, \ldots, s_k\}$, we construct a bipartite graph $\mathcal{G} = (A \cup B, E)$ as follows. For each request $r \in \mathcal{R}$, we add point a_r at location $\textsc{dest}(r)$ and point b_r at the location $\textsc{src}(r)$ to A and B,

respectively. Furthermore, for each server $s \in \mathcal{S}$, we add a point a_s at the location of s to A.

For any two requests r_i and r_j, if the pair (r_i, r_j) is admissible then we add the edge (a_{r_i}, b_{r_j}) to E. Additionally, for each server $s \in \mathcal{S}$ and each request $r \in \mathcal{R}$, we add the edge (a_s, b_r) to E. This completes the construction of the graph $\mathcal{G}$ for the instance of TAP.

Let us next discuss some of the preliminaries of the minimum-cost maximum matching problem.

Definitions and preliminaries for the minimum-cost maximum matching problem.

Suppose G is a bipartite graph defined over two point sets A and B with an edge set $E \subseteq A \times B$. Let M denote a matching on the graph G.

Primal-Dual Framework. Given a bipartite graph $G = (A \cup B, E)$, a matching M on G along with a set of dual weights $y : A \cup B \to \mathbb{R}_{\geq 0}$ is *feasible* if

$$y(b) - y(a) \leq \mathrm{d}(a, b), \qquad \forall (a, b) \in E, \tag{6.1}$$

$$y(b) - y(a) = \mathrm{d}(a, b), \qquad \forall (a, b) \in M. \tag{6.2}$$

A maximum matching $M, y(\cdot)$ is *dual-optimal* if (i) $M, y(\cdot)$ is feasible, (ii) for each free point $a \in A$, $y(a) = 0$, and (iii) for each free point $b \in B$, $y(b) = \max_{b' \in B} y(b')$.

Lemma 6.2. *Given a bipartite graph G and a dual-optimal maximum matching $M, y(\cdot)$ on G, the matching M is a minimum-cost maximum matching on G.*

Proof. Let A_F (resp. B_F) denote the set of points of A (resp. B) that are free in M. Let $y_{\max} := \max_{b \in B} y(b)$. By dual optimality conditions, for each point $a \in A_F$, $y(a) = 0$ and for each point $b \in B_F$, $y(b) = y_{\max}$. Using the feasibility condition equation 6.2, we can rewrite the cost of the matching M as follows.

$$\begin{aligned} w(M) &= \sum_{(a,b) \in M} \mathrm{d}(a,b) = \sum_{(a,b) \in M} y(b) - y(a) \\ &= \left(\sum_{b \in B} y(b) - \sum_{a \in A} y(a) \right) - \sum_{b \in B_F} y(b) + \sum_{a \in A_F} y(a) \\ &= \left(\sum_{b \in B} y(b) - \sum_{a \in A} y(a) \right) - |B_F| \cdot y_{\max}. \end{aligned} \tag{6.3}$$

Let M^* denote any minimum-cost maximum matching on G. Let A_F^* (resp. B_F^*) denote the set of points of A (resp. B) that are free in M^*. Since both M and M^* are maximum matchings, $|B_F| = |B_F^*|$. Using the feasibility condition equation 6.1,

$$\begin{aligned} w(M^*) &= \sum_{(a,b) \in M^*} \mathrm{d}(a,b) \geq \sum_{(a,b) \in M^*} y(b) - y(a) \\ &= \left(\sum_{b \in B} y(b) - \sum_{a \in A} y(a) \right) - \sum_{b \in B_F^*} y(b) + \sum_{a \in A_F^*} y(a) \\ &\geq \left(\sum_{b \in B} y(b) - \sum_{a \in A} y(a) \right) - |B_F^*| \cdot y_{\max}, \end{aligned} \tag{6.4}$$

where the last inequality holds since $y(b) \leq y_{\max}$ for each point $b \in B$ and $y(a) \geq 0$ for each point $a \in A$. Combining Equations equation 6.3 and equation 6.4,

$$w(M) = \sum_{b \in B} y(b) - \sum_{a \in A} y(a) - |B_F| \cdot y_{\max} \leq w(M^*).$$

Since M^* is a minimum-cost maximum matching, $w(M) = w(M^*)$ and M is also a minimum-

cost maximum matching. □

For any feasible matching $M, y(\cdot)$ and any edge $(a, b) \in E$, the *slack* of (a, b), denoted by $s(a, b)$, is defined as $s(a, b) := \mathrm{d}(a, b) - y(b) + y(a)$. The edge (a, b) is *admissible* if $s(a, b) = 0$. Any alternating path P is admissible if all edges in P are admissible.

Residual Graph. The *residual graph* of G with respect to a matching M is a graph defined over $A \cup B$ in addition to a source vertex s. For any pair $(a, b) \in E$, if (a, b) is a matching (resp. non-matching) edge, there is an edge directed from a to b (resp. from b to a) with a weight $s(a, b)$ in the residual graph. Furthermore, a zero-weight edge exists from s to every free point in B. This completes the construction of the residual network. The construction of the residual graph helps in computing admissible augmenting paths.

Hungarian Algorithm. The Hungarian algorithm [70] computes a dual-optimal maximum matching by initializing an empty matching M and a zero dual weight assignment to points in $A \cup B$. The algorithm iteratively executes the HungarianSearch procedure (described below), which updates the dual weights while maintaining conditions (i)–(iii) and returns an admissible augmenting path P. The algorithm then augments the matching along the path P and continues the iterations until M is a maximum matching.

HungarianSearch procedure Execute the Dijkstra's shortest path algorithm from the source s to compute, for each point $v \in A \cup B$, the distance ℓ_v from s to v. Let A_F denote the set of free points of A with respect to M, and define $\ell := \min_{a \in A_F} \ell_a$. For each point $v \in A \cup B$ with $\ell_v < \ell$, set $y(v) \leftarrow y(v) - \ell_v + \ell$. Define $a^* := \min_{a \in A_F} \ell_a$, and let P denote the path from s to a^* in the Dijkstra's shortest path tree. Let P' denote the path obtained from removing s from P. The path P is an admissible augmenting path from a free point

$b \in B$ to a^*. The HungarianSearch procedure returns P'. This completes the description of the HungarianSearch procedure. The following lemma states a well-known property of the HungarianSearch procedure.

Lemma 6.3. *Let $M, y(\cdot)$ denote a feasible matching on G. Let $y'(\cdot)$ denote the updated dual weights after executing the* HungarianSearch *procedure, and let P denote the path returned by the procedure. Then, $M, y'(\cdot)$ is a feasible matching for G and the path P is an admissible augmenting path.*

The Hungarian algorithm computes a minimum-cost maximum matching on a graph of m vertices in $O(m^3)$ time [70]. In the geometric settings, one can use a weighted nearest neighbor data structure with query and update time of $\Phi(m)$ to improve the efficiency of the Hungarian algorithm to $\tilde{O}(m^2\Phi(m))$ [5, 98, 103].

6.2 Time-Based Divide-and-Conquer Algorithm

Using the reduction shown in the previous section, we present a new time-based divide-and-conquer strategy that computes the optimal solution to the k-FCFSRP. In the following subsections, we first discuss the background required to present our algorithm. We then present the details of the algorithm and finally show that the running time of our algorithm is $\tilde{O}(kn^2)$. This result also extends to the TAP in a straightforward manner and the same algorithm, analysis, and discussion apply directly to the TAP as well. To simplify the presentation, we demonstrate the algorithm with k-FCFSRP, but the algorithm remains valid for the TAP.

6.2.1 Preliminaries

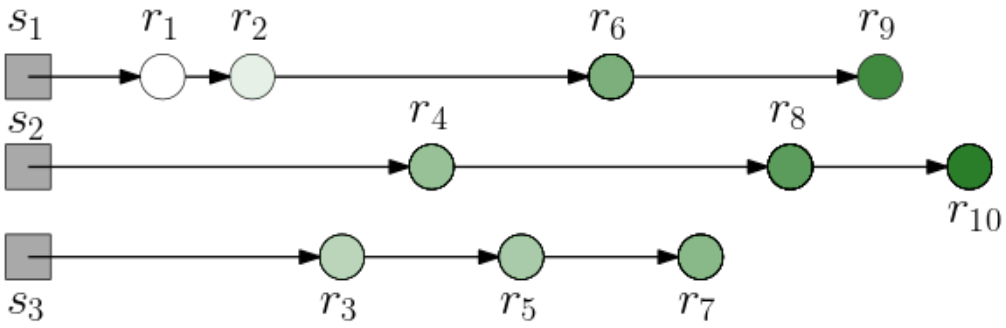
Let $\mathcal{R} = \langle r_1, \ldots, r_n \rangle$ denote the set of n requests in the order of their arrival time, and let $\mathcal{S} = \{s_1, \ldots, s_k\}$ denote the set of k servers. For each request $r_i \in \mathcal{R}$, $t(r_i)$ denotes the arrival time of r_i; we also define an arrival time of 0 for each server $s \in \mathcal{S}$, i.e., $t(s) := 0$. In this algorithm, we also introduce a set of k anchor nodes $\mathcal{C} = \{c_1, \ldots, c_k\}$ and assign an arrival time of $t(c_i) := n + 1$ to each anchor node $c_i \in \mathcal{C}$. We use the function $t(\cdot)$ to split a problem instance into two sub-problems based on the arrival times.

Recall that $\mathcal{G}$ denotes the graph representation of the k-FCFSRP instance, a bipartite graph defined over two point sets A and B. For our time-based divide-and-conquer algorithm, we add a node b_c for each anchor node $c \in \mathcal{C}$ to the set B, and add a zero-weight edge from each vertex $a \in A$ to b_c. Using an identical discussion as in Section 6.1, one can easily show that any optimal matching over this new graph representation (including the anchor nodes) corresponds to an optimal solution with the same cost and contrariwise. In the following, we abuse notation and use $\mathcal{G}$, A, and B to refer to the graph representation in the presence of the anchor nodes. Note that the graph $\mathcal{G}$ is a bipartite graph over point sets A and B, where $|A| = |B| = n + k$, and assuming the existence of a valid solution for the instance of the k-FCFSRP, $\mathcal{G}$ has a perfect matching of size $n + k$. See Figure 6.2.

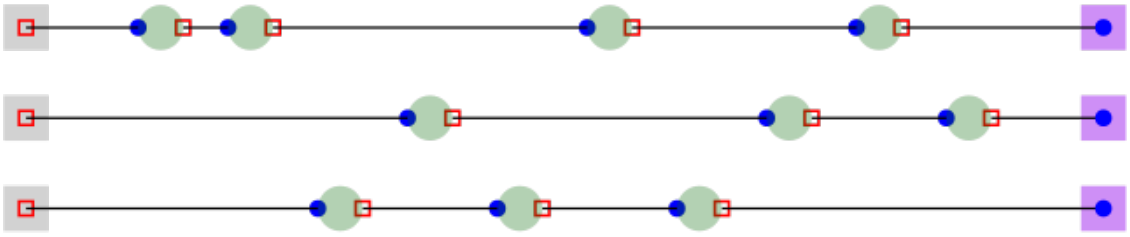
We extend the arrival times $t(\cdot)$ to the vertices of $\mathcal{G}$ in a straightforward manner as follows. For each requests $r \in \mathcal{R}$, $t(a_r) = t(b_r) = t(r)$. Furthermore, for each server $s \in \mathcal{S}$, $t(a_s) = t(s) = 0$. Finally, for each anchor node $c \in \mathcal{C}$, we set $t(b_c) = t(c) = n + 1$. For any i and j with $0 \leq i \leq j \leq n + 1$, define

$$A_{i,j} := \{a \in A \mid i \leq t(a) \leq j\}, \qquad B_{i,j} := \{b \in B \mid i \leq t(b) \leq j\}.$$

Define $\mathcal{G}_{i,j}$ as the subgraph of $\mathcal{G}$ induced by $A_{i,j} \cup B_{i,j}$.



(a)



(b)

Figure 6.2: (a) An instance of the k-FCFSRP, (b) its corresponding instance of the bipartite matching problem.

6.2.2 Algorithm

At a high level, our algorithm decomposes the problem defined over $A \cup B$ into two sub-problems $\mathcal{G}_{0,\lceil n/2 \rceil - 1}$ and $\mathcal{G}_{\lceil n/2 \rceil, n+1}$, and recursively solves them independently to compute feasible maximum matchings M_- and M_+ along with dual weights $y(\cdot)$, respectively (See Figure 6.3(a)). The algorithm then combines M_- and M_+ in the conquer step and iteratively augments the combined matching along an admissible augmenting path to obtain a feasible maximum matching for $A \cup B$ (See Figure 6.3(b)). We provide the details next.

For some i and j with $0 \leq i \leq j \leq n+1$, our time-based divide-and-conquer algorithm computes a feasible maximum matching $M_{i,j}, y_{i,j}(\cdot)$ in $\mathcal{G}_{i,j}$ as follows.

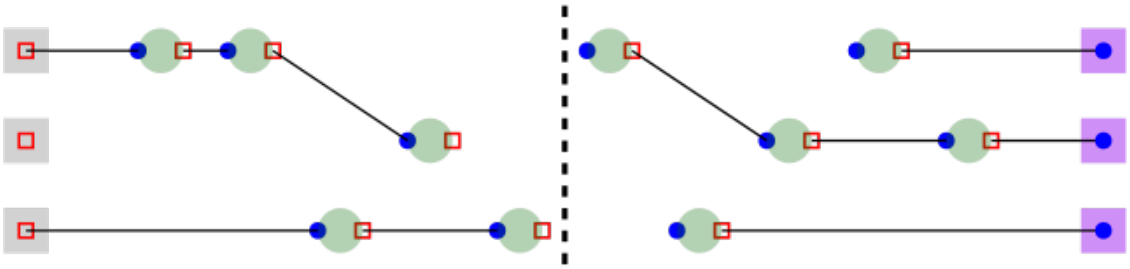
Base case If $i = j$, simply set $M_{i,j} = \emptyset$ and $y_{i,j}(v) = 0$ for each point $v \in A_{i,j} \cup B_{i,j}$.

Divide step If $i < j$, let $t := \lceil \frac{i+j}{2} \rceil$. Divide the problem into two sub-problems $\mathcal{G}_{i,t-1}$ and $\mathcal{G}_{t,j}$. Recursively solve the sub-problem defined for $\mathcal{G}_{t,j}$ to compute a feasible maximum matching $M_{t,j}, y_{t,j}(\cdot)$ on $\mathcal{G}_{t,j}$. For each point $a \in A_{i,t-1}$, set an initial dual weight of

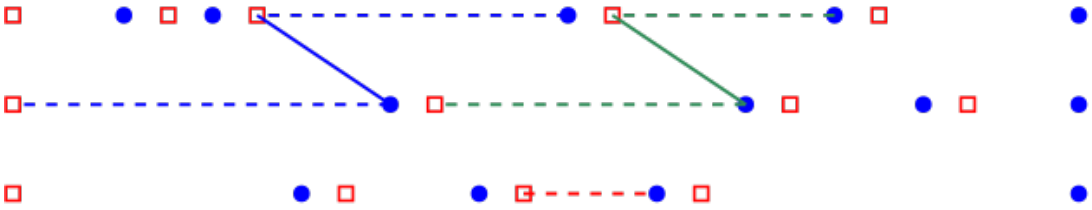
$$y_{i,t-1}(a) := \max_{b \in B_{t,j}} y_{t,j}(b). \tag{6.5}$$

Recursively solve the sub-problem defined for $\mathcal{G}_{i,t-1}$ to compute a feasible maximum matching $M_{i,t-1}, y_{i,t-1}(\cdot)$ on $\mathcal{G}_{i,t-1}$.

Conquer step Define $M_{i,j} := M_{i,t-1} \cup M_{t,j}$, and dual weights $y_{i,j}(\cdot)$ that assigns $y_{i,j}(v) := y_{i,t-1}(v)$ for each $v \in A_{i,t-1} \cup B_{i,t-1}$ and $y_{i,j}(v) := y_{t,j}(v)$ for each $v \in A_{t,j} \cup B_{t,j}$. The matching $M_{i,j}, y_{i,j}(\cdot)$ is a feasible (possibly not maximum) matching. The algorithm then iteratively executes the HungarianSearch procedure, which updates the dual weights $y_{i,j}(\cdot)$



(a)



(b)

Figure 6.3: (a) Feasible matchings computed inside the sub-problems, (b) three admissible augmenting paths (red, blue, and green paths).

and returns an admissible augmenting path P or concludes that the matching $M_{i,j}$ is a maximum matching. Upon returning an admissible augmenting path P, the algorithm augments $M_{i,j}$ along P and continues the iterations. This completes the description of the conquer step.

The following lemma shows that the matching $M_{i,j}, y_{i,j}(\cdot)$ constructed at the beginning of the conquer step is a feasible matching.

Lemma 6.4. *For any $i, j \in [0, n+1], i \leq j$, when processing the sub-problem defined by $\mathcal{G}_{i,j}$, the matching $M_{i,j}, y_{i,j}(\cdot)$ constructed at the beginning of the conquer step of the algorithm is feasible.*

From Lemma 6.3, the matching $M_{i,j}, y_{i,j}(\cdot)$ remains feasible after each iteration of the HungarianSearch procedure. Furthermore, augmenting $M_{i,j}$ along an admissible augmenting

path returned by the HungarianSearch procedure does not violate the feasibility conditions; therefore, the maximum matching $M_{i,j}, y_{i,j}(\cdot)$ returned by the algorithm after processing $\mathcal{G}_{i,j}$ is a feasible matching. Since the graph $\mathcal{G}_{0,n+1} = \mathcal{G}$ that defines the main problem has a perfect matching, the final matching $M_{0,n+1}, y_{0,n+1}(\cdot)$ computed by our algorithm is a dual-optimal matching, which by Lemma 6.2, is a minimum-cost perfect matching, as desired.

6.2.3 Efficiency Analysis

In this section, we show that the running time of our time-based divide-and-conquer algorithm is $\tilde{O}(kn^2)$. To achieve this, we show that for any sub-problem $\mathcal{G}_{i,j}$ for some $i, j \in [0, n+1], i < j$, the conquer step of the algorithm executes $O(k)$ augmentations. We then argue that each execution of the Sync and HungarianSearch procedures on a sub-problem of n' vertices takes $O(n'^2)$ time. We finally conclude that the running time of the conquer step for any sub-problem consisting of n' vertices is bounded by $O(kn'^2)$, leading to a total running time of $O(kn^2)$. The details are provided next.

Consider any sub-problem $\mathcal{G}_{i,j}$ for some $i, j \in [0, n+1], i < j$, and let $t \in [i+1, j]$. Let M^* denote a minimum-cost maximum matching on $\mathcal{G}$ corresponding to the optimal solution for the k-FCFSRP. Let M_t^* denote the subset of edges of the optimal matching M^* over $\mathcal{G}$ with one endpoint in $\mathcal{G}_{0,t-1}$ and the other endpoint in $\mathcal{G}_{t,j}$. In the following, we first show in Lemma 6.5 that the number of edges of M^* with one endpoint in $\mathcal{G}_{0,t-1}$ and the other in $\mathcal{G}_{t,n}$ is at most k.

Lemma 6.5. *For $t \in [1, n-1]$, let M_t^* denote the subset of edges of the optimal matching M^* over $\mathcal{G}$ with one endpoint in $\mathcal{G}_{0,t-1}$ and the other endpoint in $\mathcal{G}_{t,n}$. Then, $|M_t^*| \leq k$.*

Proof. An optimal solution to the k-FCFSRP consists of k routing plans. The routing plans can be interpreted as k disjoint paths starting from the initial locations of the servers, where

each path denotes the path traveled by each taxi. If a matching edge from $a_p \in A$ to $b_q \in B$ is present in the optimal solution, it implies that one of the given taxis served the request r_p and then proceeded to serve the request r_q in the optimal solution. At any time instance t, a taxi can participate in no more than 1 edge. Hence, the total number of edges in $|M_t^*|$ can not exceed k. □

Note that by the construction of the graph $\mathcal{G}$, for each edge $(a, b) \in M_t^*$, we know $a \in A_{0,t-1}$ and $b \in B_{t,j}$. Let $M_{i,t-1}^*$ (resp. $M_{t,j}^*$) denote the subset of edges of M^* induced by the vertices of $\mathcal{G}_{i,t-1}$ (resp. $\mathcal{G}_{t,j}$). Let $B_{t,j}^*$ denote the subset of vertices of $B_{t,j}$ that are matched in $M_{t,j}^*$. By the construction of $\mathcal{G}$, all vertices $b \in B_{t,j} \setminus B_{t,j}^*$ are vertices that are matched to a point $a \in A_{0,t-1}$ in M^* and hence, are incident on an edge in M_t^*. Therefore, using Lemma 6.5, the matching $M_{t,j}^*$ is a matching over $\mathcal{G}_{t,j}$ that matches all except

$$|B_{t,j} \setminus B_{t,j}^*| \leq |M_t^*| \leq k$$

vertices of $B_{j,t}$. As a result, any maximum matching over $\mathcal{G}_{j,t}$ leaves at most k points of $B_{j,t}$ unmatched, and the conquer step of the algorithm when processing $\mathcal{G}_{i,j}$ needs to computes at most k augmenting paths, one for each unmatched point of $B_{t,j}$.

Corollary 6.6. *For any $i, j \in [0, n+1], i < j$, when processing $\mathcal{G}_{i,j}$, the conquer step of our time-based divide-and-conquer algorithm runs $O(k)$ iterations.*

Let n' denote the number of vertices in the graph $\mathcal{G}_{i,j}$. Each iteration of the conquer step executes the HungarianSearch procedure that takes $O(n'^2)$ time followed by an augmentation along an augmenting path that takes $O(n')$ time. Since there are $O(k)$ iterations, the running time of the conquer step when processing $\mathcal{G}_{i,j}$ is $O(kn'^2)$. Since the total size of all sub-problems at each level of recursion is $2n + k = O(n)$, the total running time of the conquer step across all sub-problems of each level of recursion is $O(kn^2)$. Since there are

$O(\log n)$ levels, the total running time of the algorithm is $\tilde{O}(kn^2)$. When the points are in two-dimensional space, we can utilize a dynamic weighted nearest neighbor data structure to execute the HungarianSearch procedure in $\tilde{O}(n')$ time and consequently improving the execution time of our algorithm to $\tilde{O}(kn)$.

6.3 Geometric Divide-and-Conquer Algorithm

In this section, we show that, in two-dimensional Euclidean space, we can solve the k-FCFSRP using the GRS algorithm. The GRS algorithm also applies directly to the TAP for a two-dimensional Euclidean space. This is because both the k-FCFSRP and the TAP can be reduced to the minimum-cost maximum matching problem, which we then solve using our algorithm. We demonstrate the algorithm with the k-FCFSRP to simplify the presentation. We then show that for k-FCFSRP, this algorithm achieves a running time of $\tilde{O}(n^{1.8})$, and for the TAP, we show significant empirical performance improvement when compared to the state-of-the-art. We next describe the GRS algorithm.

6.3.1 The GRS Algorithm

Suppose A and B are two point sets of size n inside a square $\square$, and let M be a (possibly partial) matching of A and B. Gattani et al. [55] defined a $\square$-constrained cost for M (Equation equation 6.6 below) and showed that when the points in $A \cup B$ are far enough from the boundaries of $\square$, any minimum $\square$-constrained cost matching will be a minimum-cost perfect matching. They then designed a primal-dual framework for computing a minimum $\square$-constrained cost matching. Using this primal-dual framework, they devised a quadtree-based divide-and-conquer algorithm for computing a minimum $\square$-constrained cost matching.

The details are provided next.

We begin by defining some notations. Let $\mathcal{Q}$ be a randomly shifted quadtree that hierarchically partitions a square into four equal-sized child squares. For any point u and any square $\square$ of $\mathcal{Q}$, let $\mathrm{d}(u, \square)$ be the distance of u to the boundaries of $\square$. For any square $\square$, let $A_\square$ and $B_\square$ denote the subset of A and B that lie inside $\square$, respectively.

Constrained Cost. For a square $\square$ of $\mathcal{Q}$, let $M_\square$ be a matching of the points in $A_\square \cup B_\square$. Let $A_\square^F$ (resp. $B_\square^F$) denote the set of free points of $A_\square$ (resp. $B_\square$) with respect to $M_\square$. The $\square$-*constrained cost* of the matching $M_\square$ is defined as

$$w_\square(M_\square) := \sum_{(a,b)\in M_\square} \mathrm{d}(a,b) + \sum_{b\in B_\square^F} \mathrm{d}(b,\square). \tag{6.6}$$

The *minimum-cost* $\square$-*constrained matching*, or simply $\square$-*MCM*, is defined as a matching with a minimum $\square$-constrained cost. It is shown that if the root cell $\square^*$ of the quadtree is large enough, then any $\square^*$-MCM would be a minimum-cost perfect matching on $A \cup B$ [55, Lemma 2.2].

Constrained Feasibility. For any square $\square$ of $\mathcal{Q}$, a matching M and a set of non-negative dual weights $y(\cdot)$ for the points in $A_\square \cup B_\square$ is $\square$-*feasible* if,

$$y(b) - y(a) \le \mathrm{d}(a,b), \qquad \forall (a,b) \in A_\square \times B_\square, \tag{6.7}$$

$$y(b) - y(a) = \mathrm{d}(a,b), \qquad \forall (a,b) \in M, \tag{6.8}$$

$$y(b) \le \mathrm{d}(b,\square), \qquad \forall b \in B_\square, \tag{6.9}$$

$$y(a) = 0, \qquad \forall a \in A_\square^F. \tag{6.10}$$

Let $M, y(\cdot)$ be a $\square$-feasible matching. For any pair of points $(a, b) \in A_\square \times B_\square$, the *slack* of (a, b) is defined as $s(a, b) := \mathrm{d}(a, b) - y(b) + y(a)$. Furthermore, for any point $b \in B_\square$, the slack of b is defined as $s_\square(b) := \mathrm{d}(b, \square) - y(b)$. Any point $b \in B_\square$ is called a $\square$-*free* point if it is unmatched in M and $s(b) > 0$. Any $\square$-feasible matching with no $\square$-free points of $B_\square$ is called a $\square$-*optimal matching*. An *admissible path* is defined as an alternating path that starts from a $\square$-free point $b \in B_\square$, contains only zero-slack edges, and ends at either (i) a free point of $A_\square$, or (ii) a zero-slack point of $B_\square$.

Gattani et al. showed that any $\square$-optimal matching $M, y(\cdot)$ is a $\square$-MCM [55, Lemma 2.3]. Using this observation, they designed a quadtree-based divide-and-conquer algorithm as follows. To compute a $\square$-MCM, their algorithm first partitions $\square$ into four equal-sized squares $\square_1, \square_2, \square_3$, and $\square_4$ and recursively computes a $\square_i$-MCM for each $i \in [1, 4]$. Then, the algorithm combines these matchings and iteratively computes admissible paths to augment the matching. The details are described next.

Divide-and-Conquer Algorithm. The GRS algorithm first builds a randomly-shifted quadtree $\mathcal{Q}$. For any square $\square \in \mathcal{Q}$, let $\mathsf{C}[\square]$ be the set of children of $\square$ in $\mathcal{Q}$. The GRS algorithm processes any square $\square \in \mathcal{Q}$ as follows.

Base case If $\square$ contains only one point v from $A \cup B$, then if $v \in B$, set $y(v) \leftarrow \mathrm{d}(v, \square)$ and $M_\square \leftarrow \emptyset$; otherwise, if $v \in A$, then set $y(v) \leftarrow 0$ and $M_\square \leftarrow \emptyset$.

Divide step If $\square$ contains more than one point, for each child $\square' \in \mathsf{C}[\square]$, recursively compute the $\square'$-optimal matching $M_{\square'}, y(\cdot)$ on $A_{\square'} \cup B_{\square'}$.

Conquer step Set $M_\square := \bigcup_{\square' \in \mathsf{C}[\square]} M_{\square'}$. While a $\square$-free point exists in $B_\square$, execute a Hungarian search-style procedure called HungarianSearch procedure on the residual graph to find an admissible path P and augment $M_\square$ along P.

HungarianSearch procedure. Suppose $G'_\square$ is the residual graph of $G_\square$, where the source vertex has zero-weight edges only to the $\square$-free points of $B_\square$ (and not all free points of $B_\square$). This procedure executes Dijkstra's shortest path algorithm from the source s to compute the distance κ_v of each point $v \in A_\square \cup B_\square$ from s. Define

$$\kappa = \min\{\min_{a \in A^F_\square} \kappa_a, \min_{b \in B_\square} \kappa_b + s_\square(b)\}.$$

For any $v \in A_\square \cup B_\square$, if $\kappa_v < \kappa$, update its dual weight to $y(v) \leftarrow y(v) + \kappa - \kappa_v$. Furthermore, let u be the point with this minimum distance. Let P_u denote the shortest path from s to u. The HungarianSearch procedure returns the path obtained by removing s from P_u as an admissible path.

Efficiency. Gattani et al. relied on the cost of the optimal matching to bound the number of iterations of their algorithm at each square of the quadtree. In particular, they argued that at any time during the execution of their algorithm, the sum of the dual weights of the free points of B in each square is upper-bounded by the cost of the optimal matching in that square. Using this observation, they bounded the number of free points of B after processing the children of a square (and consequently, the number of iterations of the conquer step of the algorithm) by $O(n^{3/4})$, leading to a total running time of $\tilde{O}(n^{7/4} \log \Delta)$ for the algorithm; here, Δ is the ratio of the maximum to minimum pairwise distance of points in $A \cup B$.

6.4 GRS Algorithm for the k-FCFSRP

We extend the GRS algorithm to the k-FCFSRP in a straightforward way. To achieve this, we show that the constrained cost of a matching and the constrained feasibility conditions can be applied to the graph representation of the k-FCFSRP and be used to compute a minimum-cost maximum matching over $\mathcal{G}$. We then show that the GRS algorithm can be easily extended to compute an optimal constrained matching and, consequently, a minimum-cost maximum matching over $\mathcal{G}$. Finally, we use Lemma 6.1 to conclude that the matching computed by the GRS algorithm corresponds to an optimal solution for the k-FCFSRP.

Given a sequence of n requests $\mathcal{R}$ and a set of k servers $\mathcal{S}$ inside the unit square, let $\mathcal{G}$ be their graph representation as constructed in Section 6.1. Let ξ be a point selected uniformly at random from the unit square $[0,1]^2$ and define $\square^* = [-3,3]^2 + \xi$ to be a square of side-length 6 that is shifted by ξ. The following lemma shows that computing a minimum-cost maximum matching on $\mathcal{G}$ reduces to computing a $\square^*$-MCM on $\mathcal{G}$.

Lemma 6.7. *Let M be a $\square^*$-MCM for the graph $\mathcal{G}$. Then, M is a minimum-cost maximum matching for $\mathcal{G}$.*

For any square $\square$, let $\mathcal{G}_\square$ denote the subgraph of $\mathcal{G}$ induced by $A_\square$ and $B_\square$ and let $E_\square$ denote the set of edges of $\mathcal{G}_\square$. We extend the $\square$-constrained feasibility conditions to $\mathcal{G}_\square$ by simply taking conditions equation 6.8–equation 6.10 the same way and considering the condition of Equation equation 6.7 only for the edges in $E_\square$, i.e., we replace Equation equation 6.7 with

$$y(b) - y(a) \leq d(a,b), \qquad \forall (a,b) \in E_\square. \tag{6.11}$$

We also extend the definitions of slack, $\square$-free point, $\square$-optimal matching, and admissible paths in a straightforward way. The following lemma shows that any $\square$-optimal matching is

a $\square$-MCM of $\mathcal{G}_\square$.

Lemma 6.8. *For any square $\square$ and any $\square$-optimal matching $M, y(\cdot)$ over $\mathcal{G}_\square$, the matching M is a $\square$-MCM.*

Next, we show that combining the matchings computed at the children of $\square$ does not violate the $\square$-feasibility conditions.

Lemma 6.9. *For a square $\square$ of $\mathcal{Q}$, let $\mathsf{C}[\square]$ denote the set of children of $\square$ in $\mathcal{Q}$, and let $M_{\square'}, y(\cdot)$ denote a $\square'$-MCM for each child $\square' \in \mathsf{C}[\square]$. Then, the matching $\bigcup_{\square' \in \mathsf{C}[\square]} M_{\square'}$ along with dual weights $y(\cdot)$ is a $\square$-feasible matching.*

Finally, we note that the GRS algorithm computes a minimum-cost perfect matching on a complete bipartite graph defined over point sets A and B inside the unit square. The algorithm uses the residual graph to find admissible paths (i.e., minimum-net-cost augmenting paths) and augment the matching along those paths. It is shown that if we augment a feasible matching along an admissible path, the matching remains feasible, and also, the number of $\square$-free points of $B_\square$ reduces by one [55, Lemma 2.5].

To apply the GRS algorithm to the graph $\mathcal{G}$, one can simply construct a *residual representing graph* as a graph $\mathcal{G}'$ defined over $A_\square \cup B_\square$ in addition to a source vertex s. For any edge $(a, b) \in E_\square$, if (a, b) is a matching (resp. non-matching) edge, we add an edge from a to b (resp. b to a) with weight $s(a, b)$ to $\mathcal{G}'$. In addition, for each $\square$-free point $b \in B_\square$, we add a zero-weight edge from s to b. Using this construction, in each iteration of the conquer step, we execute the HungarianSearch procedure on $\mathcal{G}'$ to compute an admissible path and augment the matching. One can use an identical proof as in [55, Lemma 2.5] to show that the augmentation does not violate the feasibility conditions; furthermore, each iteration reduces the number of $\square$-free points by one.

6.4.1 Efficiency Analysis of GRS algorithm for the k-FCFSRP

In this section, we show that the GRS algorithm computes an optimal solution for the k-FCFSRP in $\tilde{O}(n^{1.8})$ time in expectation. To achieve this, we show that for any square $\square$ with $n_\square$ requests and $k_\square$ servers inside $\square$, the conquer step of the GRS algorithm runs $\tilde{O}(n^{0.8})$ iterations in expectation, where each iteration takes $\tilde{O}(n_\square + k_\square)$ time. We then give an argument to conclude that the total computation at each level of the quadtree takes $\tilde{O}(n^{1.8})$ time. In the bipartite graph representing the k-FCFSRP instance, an edge exists from $a_i \in A$ to $b_j \in B$ only if $i < j$. Since our given graph is not a complete bipartite graph, we can not directly use the efficiency analysis by Gattani et al. [55]. We provide a different analysis to achieve a subquadratic bound. Gattani et al. relied on the cost of the optimal matching to bound the number of iterations of their algorithm at each square of the quadtree. In particular, they argued that at any time during the execution of their algorithm, the sum of the dual weights of the free points of B in each square is upper-bounded by the cost of the optimal matching in that square. Using this observation, they bounded the number of free points of B after processing the children of a square (and consequently, the number of iterations of the conquer step of the algorithm) by $O(n^{3/4})$, leading to a total running time of $\tilde{O}(n^{7/4} \log \Delta)$ for the algorithm.

In our problem instance, too, it is true that the sum of the dual weights of the free points of B in each square is upper-bounded by the cost of the optimal matching in that square. However, since our given graph is not a complete bipartite graph, obtaining a tight enough upper bound on the cost of an optimum solution that guarantees the number of free points of B would be sublinear is not straightforward. Therefore, in our analysis, instead of relying on the cost of the minimum-cost maximum matching, we use the cost of an optimal partial matching. More specifically, for any square $\square$ of $\mathcal{Q}$ with a side-length $\ell_\square$, we show that there exists a partial matching M that matches all except $O(n^{0.8})$ points and has a squared

Euclidean cost of $O(\ell_\square n^{0.6})$. Using this observation, we show that at any time during the execution of the GRS algorithm, the sum of the dual weights of the subset of free points of B that is matched in M is bounded by the cost of M (i.e., $O(\ell_\square n^{0.6})$. We then use this bound to show that the expected number of iterations of the conquer step of the GRS algorithm on any square is $O(n^{0.8})$, which leads to a total running time of $\tilde{O}(n^{1.8} \log \Delta)$; here, Δ is the ratio of the maximum to minimum cost of an edge in $\mathcal{G}$. Next, we will present the details of the proof.

Notations. For a square $\square \in \mathcal{Q}$, let $A_\square$ (resp. $B_\square$) denote the subset of points of A (resp. B) that lie inside $\square$. Define $n_\square := |A_\square \cup B_\square|$ and $\ell_\square$ as the side-length of $\square$. Let $M_\square, y(\cdot)$ be the $\square$-feasible matching computed at the beginning of the conquer step. For any matching M', let $F(M')$ denote the set of free points of $B_\square$ with respect to M'. For each level i of $\mathcal{Q}$, let $\mathcal{L}[i]$ denote the set of all level i squares of $\mathcal{Q}$.

Proof of efficiency. We show that the expected number of free points of $B_\square$ with respect to $M_\square$ (i.e., $|F(M_\square)|$) is $\tilde{O}(n^{0.8})$. Since the conquer step executes one iteration for each free point in $F(M_\square)$, where each iteration takes $\tilde{O}(n_\square)$ time, the execution time of the conquer step when processing $\square$ would be $\tilde{O}(n^{0.8} n_\square)$. Therefore, for any level i of the quadtree, since $\sum_{\square \in \mathcal{L}[i]} n_\square \leq n$, the total execution time of the conquer step across all squares at level i is $\tilde{O}(n^{1.8})$. Finally, summing over all $O(\log \Delta)$ levels of the quadtree, we get an execution time of $\tilde{O}(n^{1.8} \log \Delta)$. The details of bounding the number of free points in $F(M_\square)$, which is the only non-trivial step, are provided next.

To show that $|F(M_\square)| = \tilde{O}(n^{0.8})$, we first show in Lemma 6.10 below that there exists a matching M over $A_\square \cup B_\square$ that, with a high probability, matches all except $\tilde{O}(n^{0.8})$ points of $B_\square$ and has a cost $O(\ell_\square n^{0.6})$. We then use the set of augmenting paths and alternating

paths in the symmetric difference $M \oplus M_\square$ to bound the number of free points of $M_\square$. We split the free points $F(M_\square)$ into those that are endpoints of alternating paths (referred to by $F_{\text{alt}}(M_\square)$) and those that are are endpoints of augmenting paths (referred to by $F_{\text{aug}}(M_\square)$). For the first set, since each alternating path in the symmetric difference has one free endpoint in $F(M_\square)$ and the other free endpoint in $F(M)$, $|F_{\text{alt}}(M_\square)| \leq |F(M)| = \tilde{O}(n^{0.8})$. For the second set, we show in Equation equation 6.13 below that the total dual weights of the points in $F_{\text{aug}}(M_\square)$ is at most $w(M)$. Using this observation, we show in Equations equation 6.14 and equation 6.15 that the number of points in $F_{\text{aug}}(M_\square)$ is $\tilde{O}(n^{0.8})$. Combining the two bounds, $|F(M_\square)| = \tilde{O}(n^{0.8})$, as claimed.

(i) Construction of partial matching M.

Lemma 6.10. *For any square $\square$ of $\mathcal{Q}$, there exists a matching M over the representing graph $\mathcal{G}_\square$ that matches all except $O(n^{0.8})$ points of $B_\square$ and has a cost $O(\ell_\square n^{0.6})$.*

Proof. Define $\mathbb{G}$ to be a grid of cell side-length $\ell_\square n^{-0.4}$ that partitions $\square$ into $O(n^{0.8})$ equal-sized squares. For each square ξ of the grid $\mathbb{G}$, let $\mathcal{R}_\xi = \langle r'_1, \ldots, r'_m \rangle$ denote the sub-sequence of requests in $\mathcal{R}_\square$ that lie inside ξ, and let A_ξ (resp. B_ξ) denote the subset of $A_\square$ (resp. $B_\square$) that lie inside ξ. Define $\mathcal{G}_\xi$ as the sub-graph of $\mathcal{G}_\square$ induced by $A_\xi \cup B_\xi$. Let M_ξ denote the set of edge $(a_{r'_i}, b_{r'_{i+1}})$ for all $i \in [1, m-1]$. M_ξ is a matching on $\mathcal{G}_\xi$ that matches all except one point of B_ξ. Furthermore, since each matching edge in M_ξ has a cost at most $O(\ell_\square n^{-0.4})$, the cost of M_ξ would be $O(\ell_\square |B_\xi| n^{-0.4})$.

Define $M := \bigcup_{\xi \in \mathbb{G}} M_\xi$ as the union of the matchings computed inside each cell of the grid $\mathbb{G}$. Since $|\mathbb{G}| = O(n^{0.8})$ and each cell leaves at most one point of $B_\square$ unmatched, the matching

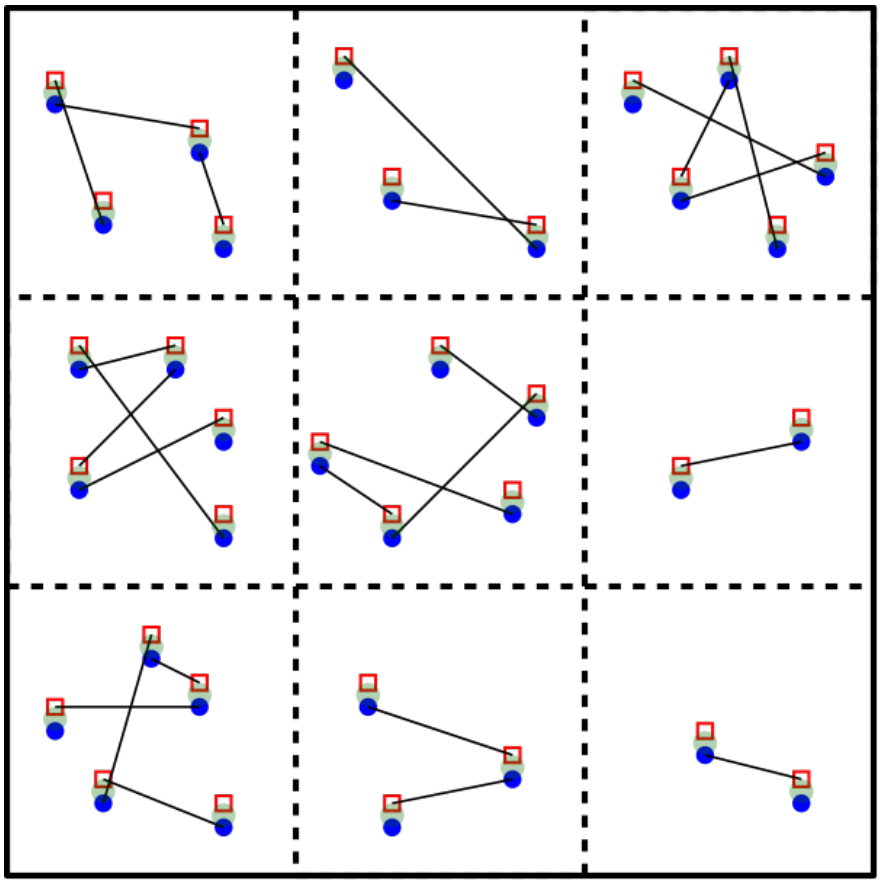


Figure 6.4: A matching M_ξ is computed inside each square ξ by connecting the requests in their arrival order.

M matches all except $O(n^{0.8})$ points of $B_\square$. In addition, the cost of M is at most

$$w(M) = \sum_{\xi \in \mathbb{G}} w(M_\xi) = O\left(\ell_\square n^{-0.4} \sum_{\xi \in \mathbb{G}} |B_\xi|\right) = O(\ell_\square n^{0.6}).$$

□

(ii) Bounding the Total Dual Weights. We next show that the total dual weights of the free points in $F_{\text{aug}}(M_\square)$ is at most $w(M)$. To achieve this, we begin by stating an important property of □-optimal matchings.

(P1) For any augmenting path P from a free point $b \in B_\square$ to a point $a \in A_\square$, $\phi(P) \geq y(b)$.

Proof. By the definition of the slack of an edge, for any non-matching edge $(a, b) \in A_\square \times B_\square$, $\phi(a, b) := \mathrm{d}(a, b) = s(a, b) + y(b) - y(a)$. Furthermore, for each matching edge $(a, b) \in M$, since the slack of (a, b) is zero, $\phi(a, b) := -\mathrm{d}(a, b) = s(a, b) - y(b) + y(a)$. Therefore, for any augmenting path P from a free point $b \in B_\square$ to a free point $a \in A_\square$,

$$\phi(P) = \sum_{(a',b') \in P} \phi(a', b') = \sum_{(a',b') \in P} s(a', b') + y(b) - y(a) \geq y(b),$$

where the last inequality holds since $s(a', b') \geq 0$ for any $(a', b') \in A_\square \times B_\square$ and $y(a) = 0$ due to feasibility condition equation 6.10. □

Let $\mathcal{P}$ denote the set of all augmenting paths in the symmetric difference $M \oplus M_\square$, i.e., $\mathcal{P}$ consists of all paths $P = \langle b_1, a_1, b_2, \ldots, b_t, a_t \rangle$ such that for all $i \in [1, t]$, $(b_i, a_i) \in M$ and for

all $i \in [1, t-1]$, $(a_i, b_{i+1}) \in M_\square$. From the definition of net-costs,

$$\sum_{p \in \mathcal{P}} \phi(P) = \sum_{p \in \mathcal{P}} \left(\sum_{(a,b) \in P \cap M} \mathrm{d}(a,b) - \sum_{(a,b) \in P \cap M_\square} \mathrm{d}(a,b) \right)$$
$$\leq \sum_{p \in \mathcal{P}} \left(\sum_{(a,b) \in P \cap M} \mathrm{d}(a,b) \right) \leq w(M). \tag{6.12}$$

For each $P \in \mathcal{P}$, let $b_P \in B_\square$ and $a_P \in A_\square$ denote the two end-points of P. Using property (P1) and Equation equation 6.12,

$$w(M) \geq \sum_{P \in \mathcal{P}} \phi(P) \geq \sum_{P \in \mathcal{P}} y(b_P) = \sum_{b \in F_{\mathrm{aug}}(M_\square)} y(b). \tag{6.13}$$

(iii) Bounding the Number of Iterations. Define $\theta := \ell_\square n^{-0.2}$. For each point $b \in B_\square$, let $\square_b \in \mathsf{C}[\square]$ denote the child of $\square$ that contains b. Each free point $b \in F(M_\square) \setminus F(M)$ is called a close (resp. far) point if $\mathrm{d}(b, \square_b) \leq \theta$ (resp. $\mathrm{d}(b, \square_b) > \theta$), and let X (resp. Y) denote the set of all close (resp. far) points of $F(M_\square) \setminus F(M)$. By [55, Lemma 2.1],

$$\mathbb{E}[|X|] = O(n^{-0.2} \mathbb{E}[n_\square]) = O(n^{0.8}). \tag{6.14}$$

Combining Equations equation 6.13 and using Lemma 6.10,

$$|Y| \leq \frac{w(M)}{\theta} = O\left(\frac{\ell_\square n^{0.6}}{\ell_\square n^{-0.2}} \right) = O(n^{0.8}). \tag{6.15}$$

Combining Equations equation 6.14 and equation 6.15, $\mathbb{E}[|F(M_\square) \setminus F(M)|] = O(n^{0.8})$, and as a result, $\mathbb{E}[|F(M_\square)|] = O(n^{0.8})$, i.e., the conquer step of the algorithm when processing $\square$ executes $O(n^{0.8})$ iterations. Each iteration of the conquer step executes a HungarianSearch procedure followed by an augmentation along an augmenting path, which takes $\tilde{O}(n_\square + k_\square)$.

For any level i of the quadtree, recall that $\mathcal{L}[i]$ denotes the set of all squares of the quadtree at level i. Note that $\sum_{\square' \in \mathcal{L}[i]} n_{\square} + k_{\square} \leq n + k$; therefore, the total running time of the conquer step of the algorithm across all squares of any level i of the quadtree would be $\tilde{O}(n^{0.8} \sum_{\square' \in \mathcal{L}[i]} n_{\square} + k_{\square}) = \tilde{O}(n^{1.8})$. Summing over all $O(\log \Delta)$ levels, the total running time of the geometric divide-and-conquer algorithm would be $\tilde{O}(n^{1.8} \log \Delta)$.

6.4.2 Empirical results of GRS algorithm for the TAP

In this section, we present the results of our experiments comparing the execution time of the GRS algorithm to that of the Hungarian algorithm to solve the taxi allocation problem (TAP). We also compare the two algorithms for some slight variations of the problem to make it more practical for real-world scenarios. Our dataset uses the Uber taxi data for NYC, collected on various weekdays in 2021. The dataset includes the locations of pickups and dropoffs and pickup times, which we use as input to the TAP.

The additional variants we incorporated into the TAP for our testing are as follows.

- Recall that in TAP, each request is a pair of pickup and a drop-off location instead of a single point. In some applications, like grocery delivery services, the requests will be a single location where the delivery has to be made at the requested time. Therefore, for such cases, we assume that the pickup and dropoff points are co-located. When the pickup and dropoff location is same for each request, we refer to the TAP as the *grocery delivery problem (GDP)*. Note that the grocery delivery problem (GDP) is a special case of the taxi allocation problem (TAP). We tested our implementation for both problems.

- To compute the cost of moving a taxi between two locations, we use the Haversine distance between the two points, which approximates (in kilometers) the distance

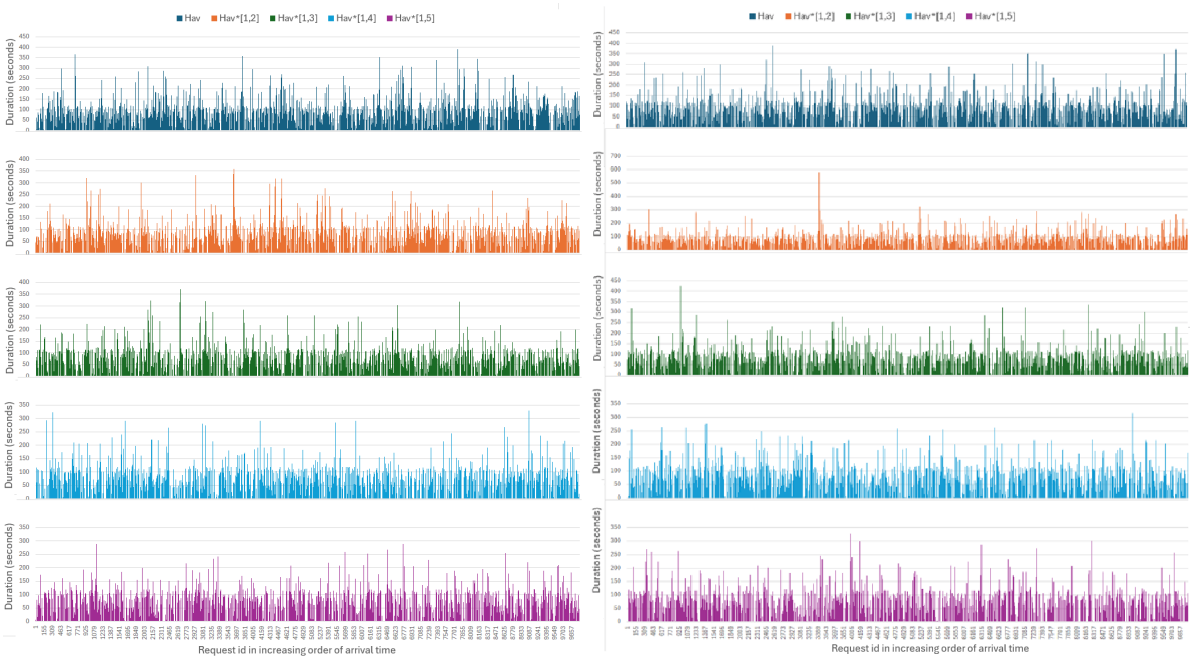


Figure 6.5: Pickup delays in optimum solution for $\sigma = 2$

between the points along the surface of the earth.In a practical scenario, a city would connect two locations via a road network. Let the distance a taxi would cover from a location a to b in a road network be the *driving* distance from a to b. The driving distance could be greater than the Haversine distance between the two locations but can not be smaller than that. Additionally, in a well-planned city with a dense road network, the driving distance between two locations can be assumed to be within a reasonably small multiple of the Haversine distance. Therefore, to evaluate the performance of our algorithm in the presence of road networks, we create four scenarios where we predefined the distance between every two locations by randomly increasing the Haversine distance between them up to two, three, four, and five times. We test the performance of our algorithm for each of the distortions. Let us denote these distance functions as **HAV*[1,2]**, **HAV*[1,3]**, **HAV*[1,4]**, and **HAV*[1,5]**, respectively, and let **HAV**denote the original Haversine distance.

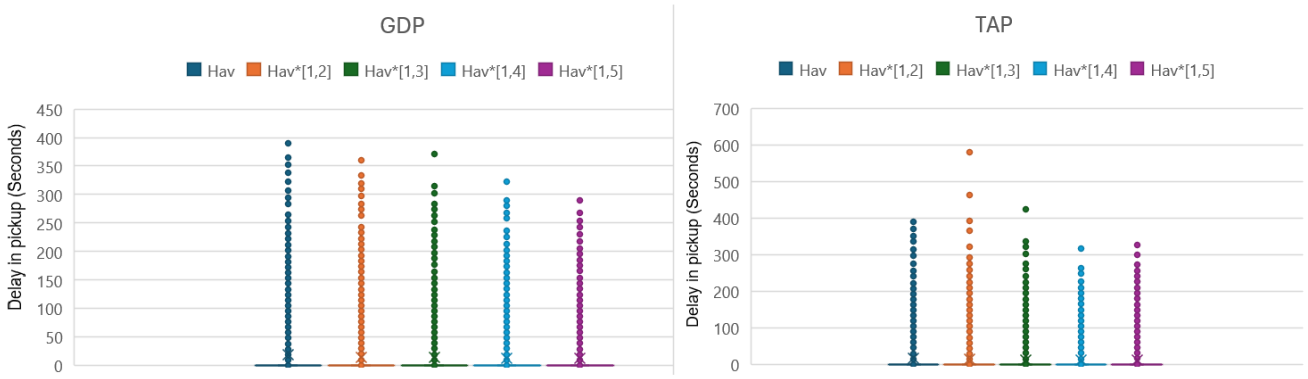


Figure 6.6: Boxplots to analyze delay in pickups in optimum solution for $\sigma = 2$

- Recall that in TAP, a pair (r_i, r_j) with $i < j$, is called *admissible* if a taxi that can serve r_i and reach in time to serve request r_j and a valid solution can only consist of admissible pairs. We introduce a variation to the definition of an admissible pair, called σ-admissible pair, where σ is a time duration in minutes. An edge (r_i, r_j) is σ-admissible if a taxi that starts traveling from the dropoff location of r_i at $r_i's$ pickup time can reach the pickup location of r_j no later than $\sigma+$ $r_j's$ pickup time. For a small value of σ, allowing only σ-admissible edges in a valid solution mimics a practical scenario where a customer can wait a few minutes after arrival before being served. In the worst case, the cumulative wait time of the last request may become very high. However, we observe that when we allow σ to be up to 5 minutes, no request accumulates a high delay in pickup, as one might suspect, even with as large as 10^5 number of requests. This is because for many request times the taxis reach well before the pickup time of a request thereby cancelling the prior accumulated wait times. Figure 6.5 shows the request pickup delays for 10^5 requests throughout the day for an optimum route when σ is set to 2 minutes. For each problem instance, we report the delays when different distance functions are used as the ground distance. As one might suspect, these delays are not compounded to be very high, especially towards the end of the day, making it a suitable constraint for real-world data. These delays are also plotted in boxplots to observe how they affect the delay in pickup for requests overall (see Figure 6.5). The plots show the interquartile range and the median near the lower end of the delay, indicating that most requests bear almost no delay. The short whiskers further support this by showing low variability in delay for most requests. It also shows that although very few requests (outliers) experience slightly higher delays, those delays are still well within practical ranges (i.e. less than 10 minutes). We also notice that, for the TAP, as we go towards a more realistic distance function (i.e., HAV*[1,4] and HAV*[1,5]), the delays tend to be smaller, and the maximum delay is

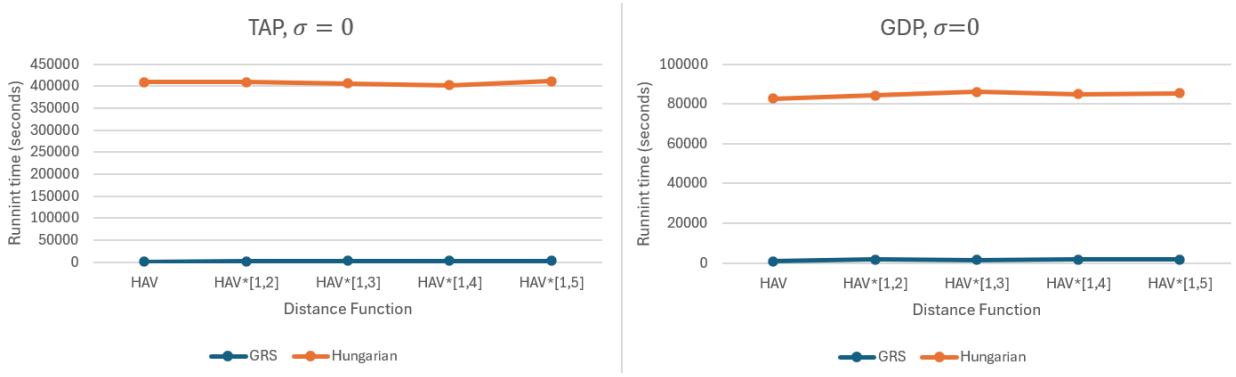


Figure 6.7: Running time of GRS vs Hungarian Search algorithm for $\sigma = 0$

only a few minutes. Hence, we can say that these constraints are well suited for making the problem instance very similar to a real-world scenario. We test our implementation for σ set as $0, 1, 2$, and 5 minutes. Note that when σ is set as 0, the problem instance is the original TAP.

We pre-process the input data to obtain optimum routes in all scenarios mentioned.

All shortest paths are computed using the original Dijkstra's shortest path algorithm taking $O(n^2)$ time. We do not implement any additional optimization such as the use of dynamic weighted nearest neighbors. All computations are performed on a computer with an i7-7660U Core CPU @ 2.50GHz and 8 GB of RAM. We repeated each experiment on eight different datasets to ensure the reliability of the results. We will show our algorithm's empirical performance compared to the Hungarian algorithm, which is currently the state-of-the-art algorithm for computing an optimal solution for our problem statements.

We first compare the time the GRS and Hungarian algorithms take for 20k requests. Figure 6.7 shows the results for the TAP and GDP when the distance functions are set to **HAV**, **HAV*[1,2]**, **HAV*[1,3]**, **HAV*[1,4]**, and **HAV*[1,5]**. We observe that the GRS algorithm runs almost 375 times faster than the Hungarian search algorithm for TAP when the distance function is **HAV**. In Figures 6.8, 6.9 and 6.10 we show the comparison when we set σ to

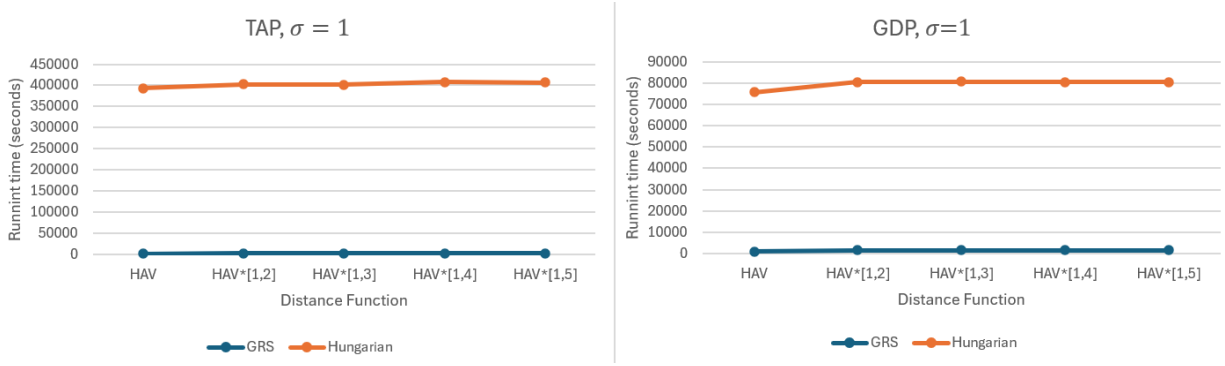


Figure 6.8: Running time of GRS vs Hungarian Search algorithm for $\sigma = 1$

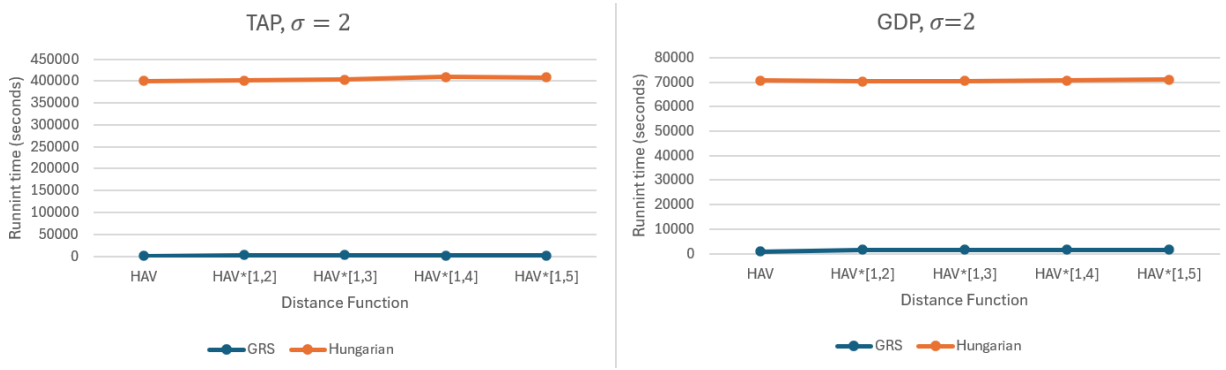


Figure 6.9: Running time of GRS vs Hungarian Search algorithm for $\sigma = 2$

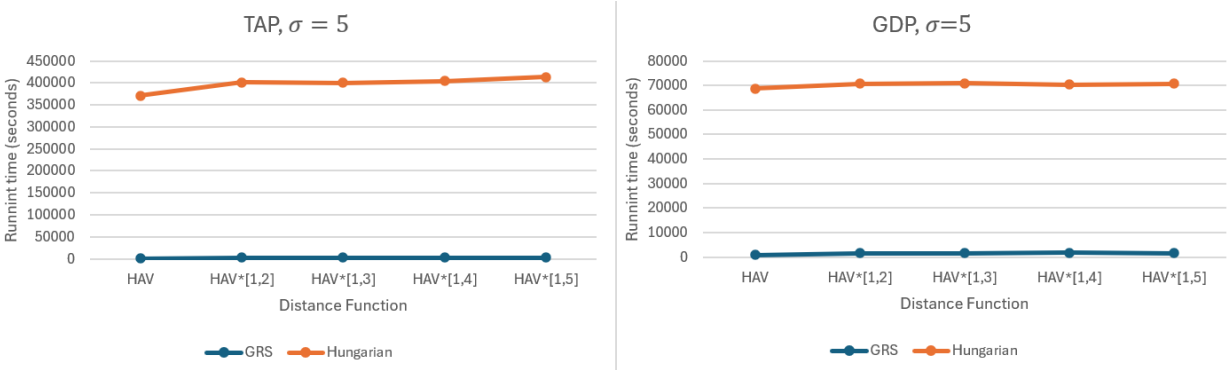


Figure 6.10: Running time of GRS vs Hungarian Search algorithm for $\sigma = 5$

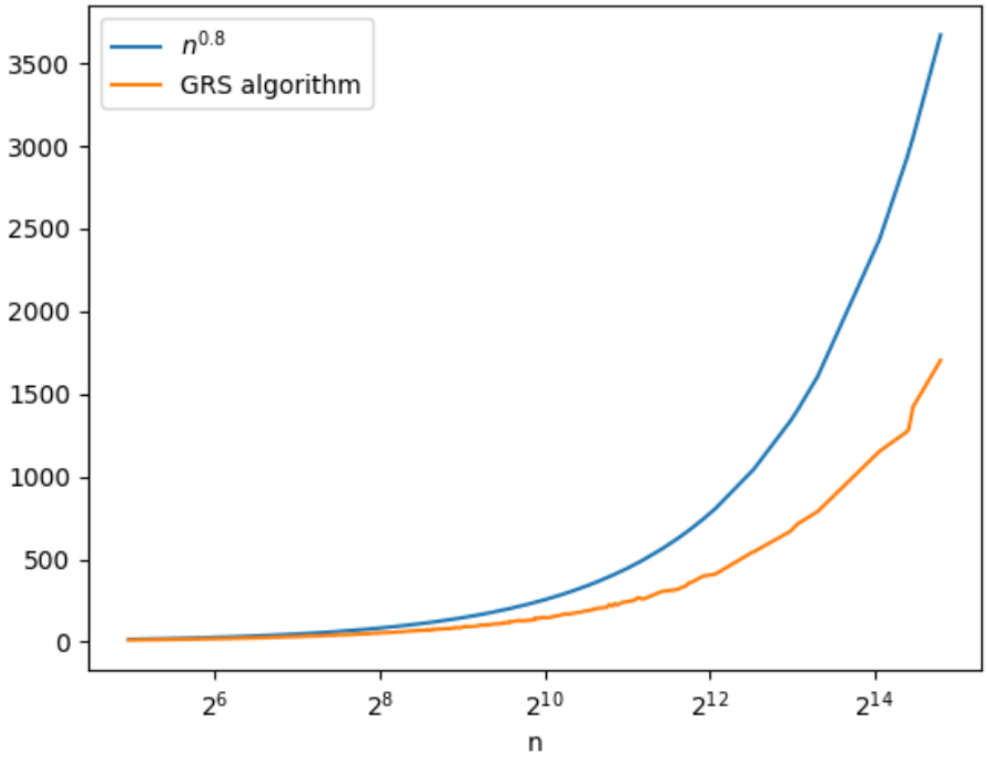


Figure 6.11: Number of iterations in conquer step of a cell with n points

1, 2 and 5. In other words, when we allow 1-admissible, 2-admissible, and 5-admissible edges to appear in the solution. Since the GRS algorithm uses a quad-tree based on Euclidean distance, it is expected to slow down slightly when the distance between points increases. However, since the increase in the distance is bounded by 2,3,4, and 5 times, we observe that the algorithm slows down to up to 2 times for these distance functions which is not significant. More importantly, the GRS algorithm runs drastically faster than the Hungarian search algorithm for all distance functions as noted earlier.

We summarize our findings as follows. First, our experiments indicate that the GRS algorithm outperforms the Hungarian algorithm in running time across all experiments. The GRS algorithm scales easily to significantly larger inputs. Figure 6.12 shows the time it takes for the GRS algorithm to solve TAP when n is set to $50K$. Running the Hungarian

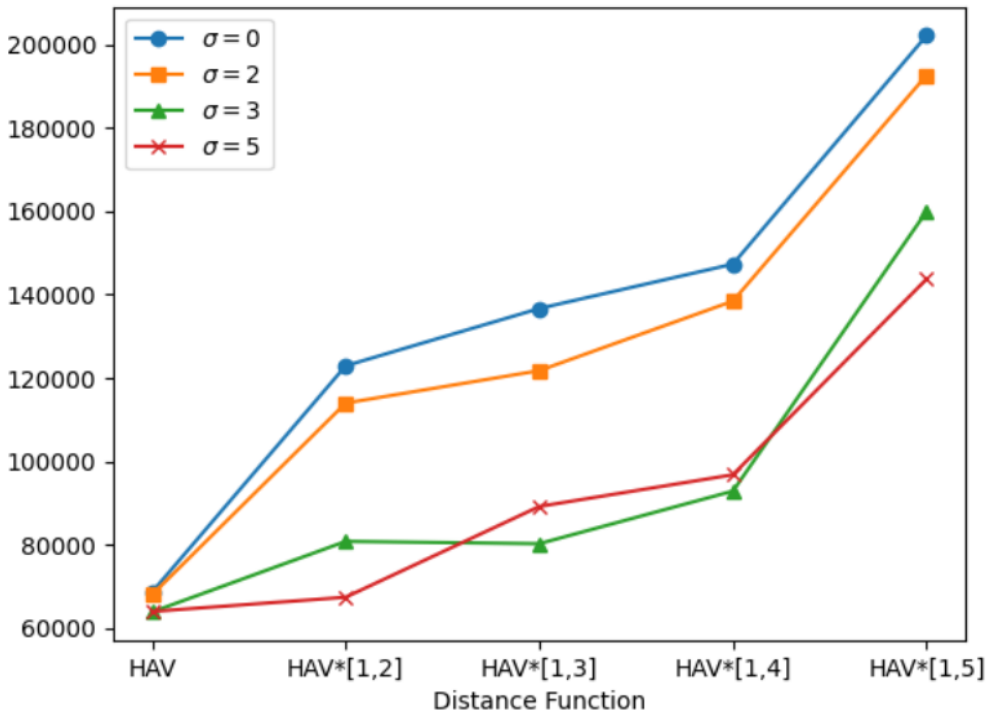


Figure 6.12: n=50K

search algorithm on such a large input size would require a continuous processing time that is out of the scope of our available resources.

We proved in section 6.4.1, for k-FCFSRP, that for any square $\square$ with $n_\square$ requests, the conquer step of the GRS algorithm runs $\tilde{O}(n^{4/5})$ iterations in expectation. To estimate the performance of the GRS algorithm for large-scale input with practical instances of TAP, we plot the number of Hungarian search iterations executed in a cell during the conquer step as a function of the total number of requests in the cell. 6.11 shows the comparison of the exact number of iterations vs $O(n_\square^{4/5})$ for TAP. We see that the number of iterations executed by the GRS algorithm is bounded by $n^{4/5}$, which implies that the GRS algorithm beats the theoretical bounds of the Hungarian algorithm for the practical instances of the taxi allocation problem.

The results suggest that the GRS algorithm is highly effective for the taxi allocation problem where distance and time constraints are critical. The ability of this algorithm to efficiently handle various limitations added to the problem instance highlights its potential for real-world use.

Bibliography

[1] Vehicle routing problem with time windows | or-tools | google developersn. https://developers.google.com/optimization/routing/vrptw.

[2] Pankaj Agarwal and Kasturi Varadarajan. A near-linear constant-factor approximation for Euclidean bipartite matching? In *Proceedings of the twentieth annual symposium on Computational geometry*, page 247, 2004.

[3] Pankaj K. Agarwal and R. Sharathkumar. Approximation algorithms for bipartite matching with metric and geometric costs. *In Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, page 555–564, 2014.

[4] Pankaj K. Agarwal and Kasturi R. Varadarajan. A near-linear constant-factor approximation for Euclidean bipartite matching? *In Proceedings of the 20th annual symposium on Computational geometry*, pages 247–252, 2004.

[5] Pankaj K Agarwal, Alon Efrat, and Micha Sharir. Vertical decomposition of shallow levels in 3-dimensional arrangements and its applications. In *Proc. eleventh Annual Sympos. Comput. Geom.*, pages 39–50, 1995.

[6] Pankaj K. Agarwal, Alon Efrat, and Micha Sharir. Vertical decomposition of shallow levels in 3-dimensional arrangements and its applications. *SIAM J. Comput.*, 29(3), December 1999.

[7] Pankaj K. Agarwal, Kyle Fox, Debmalya Panigrahi, Kasturi R. Varadarajan, and Allen Xiao. Faster algorithms for the geometric transportation problem. In *Proc. 33rd International Symposium on Computational Geometry*, pages 7:1–7:16, 2017.

[8] Pankaj K. Agarwal, Hsien-Chih Chang, and Allen Xiao. Efficient algorithms for geometric partial matching. In Gill Barequet and Yusu Wang, editors, *35th International Symposium on Computational Geometry (SoCG 2019)*, volume 129 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:14, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-104-7. doi: 10.4230/LIPIcs.SoCG.2019.6.

[9] Pankaj K. Agarwal, Hsien-Chih Chang, Sharath Raghvendra, and Allen Xiao. A deterministic near-linear ε-approximation algorithm for geometric bipartite matching. *submitted for publication*, 2021.

[10] Pankaj K Agarwal, Hsien-Chih Chang, Sharath Raghvendra, and Allen Xiao. Deterministic, near-linear ε-approximation algorithm for geometric bipartite matching. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*, pages 1052–1065, 2022.

[11] Pankaj K Agarwal, Sharath Raghvendra, Pouyan Shirzadian, and Rachita Sowle. An improved ε-approximation algorithm for geometric bipartite matching. In *18th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[12] Pankaj K. Agarwal, Sharath Raghvendra, Pouyan Shirzadian, and Rachita Sowle. An Improved ε-Approximation Algorithm for Geometric Bipartite Matching. In Artur Czumaj and Qin Xin, editors, *18th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2022)*, volume 227 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:20, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-236-5. doi: 10.4230/LIPIcs.SWAT.2022.6. URL https://drops.dagstuhl.de/opus/volltexte/2022/16166.

[13] Alonso-Mora, Samaranayake, Wallar, Frazzoli, and Rus. On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment. *Proc Natl Acad Sci U S A*, January 2017.

[14] Jason Altschuler, Jonathan Niles-Weed, and Philippe Rigollet. Near-linear time approximation algorithms for optimal transport via sinkhorn iteration. *Advances in neural information processing systems*, 30, 2017.

[15] Jason Altschuler, Jonathan Weed, and Philippe Rigollet. Near-linear time approximation algorithms for optimal transport via sinkhorn iteration. In *Advances in Neural Information Processing Systems 30*, pages 1964–1974, 2017. URL http://papers.nips.cc/paper/6792-near-linear-time-approximation-algorithms-for-optimal-transport-via-sinkhorn-

[16] Jason Altschuler, Jonathan Weed, and Philippe Rigollet. Near-linear time approximation algorithms for optimal transport via sinkhorn iteration. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 1964–1974. Curran Associates, Inc., 2017. URL http://papers.nips.cc/paper/6792-near-linear-time-approximation-algorithms-for-optimal-transport-via-sinkhorn-pdf.

[17] Jason Altschuler, Francis Bach, Alessandro Rudi, and Jonathan Niles-Weed. Massively scalable sinkhorn distances via the nyström method. *Advances in neural information processing systems*, 32, 2019.

[18] A. Andoni, K. D. Ba, P. Indyk, and D. P. Woodruff. Efficient sketches for earth-mover distance, with applications. In *Proc. 50th Annual IEEE Sympos. Foundations of Comp. Sc.*, pages 324–330, 2009.

[19] Alexandr Andoni, Piotr Indyk, and Robert Krauthgamer. Earth mover distance over high-dimensional spaces. In *SODA*, volume 8, pages 343–352, 2008.

[20] Alexandr Andoni, Piotr Indyk, Huy L Nguyen, and Ilya Razenshteyn. Beyond locality-sensitive hashing. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 1018–1028. SIAM, 2014.

[21] Roberto Baldacci, Aristide Mingozzi, and Roberto Roberti. Recent exact algorithms for solving the vehicle routing problem under capacity and time window constraints. *European Journal of Operational Research*, 218(1):1–6, 2012. ISSN 0377-2217. doi: https://doi.org/10.1016/j.ejor.2011.07.037. URL https://www.sciencedirect.com/science/article/pii/S0377221711006692.

[22] Russell Bent and Pascal Van Hentenryck. Scenario-based planning for partially dynamic vehicle routing with stochastic customers. *Operations Research*, 52:977–987, 12 2004. doi: 10.1287/opre.1040.0124.

[23] Russell Bent and Pascal Van Hentenryck. Waiting and relocation strategies in online stochastic vehicle routing. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, IJCAI'07, page 1816–1821, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

[24] Gerardo Berbeglia, Jean-François Cordeau, and Gilbert Laporte. Dynamic pickup and delivery problems. *European Journal of Operational Research*, 202(1):8–15, 2010. ISSN 0377-2217. doi: https://doi.org/10.1016/j.ejor.2009.04.024. URL https://www.sciencedirect.com/science/article/pii/S0377221709002999.

[25] Espen Bernton, Pierre E. Jacob, Mathieu Gerber, and Christian P. Robert. On parameter estimation with the wasserstein distance. *Information and Inference: A Journal of the IMA*, 8(4):657–676, 2019.

[26] Dimitris Bertsimas, Patrick Jaillet, and Sebastien Martin. Online Vehicle Routing: The Edge of Optimization in Large-Scale Applications. *Operations Research*, 67(1): 143–162, January 2019. doi: 10.1287/opre.2018.1763. URL https://ideas.repec.org/a/inm/oropre/v67y2019i1p143-162.html.

[27] Jose Blanchet, Arun Jambulapati, Carson Kent, and Aaron Sidford. Towards optimal running times for optimal transport. *arXiv preprint arXiv:1810.07717*, 2018.

[28] Jose Blanchet, Arun Jambulapati, Carson Kent, and Aaron Sidford. Towards optimal running times for optimal transport. *arXiv preprint arXiv:1810.07717*, 2018.

[29] Allan Borodin and Ran El-Yaniv. *Online computation and competitive analysis*. cambridge university press, 2005.

[30] Sébastien Bubeck, Michael B. Cohen, Yin Tat Lee, James R. Lee, and Aleksander Madry. k-server via multiscale entropic regularization. In Ilias Diakonikolas, David Kempe, and Monika Henzinger, editors, *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*, pages 3–16. ACM, 2018. doi: 10.1145/3188745.3188798. URL https://doi.org/10.1145/3188745.3188798.

[31] William R Burley. Traversing layered graphs using the work function algorithm. *Journal of Algorithms*, 20(3):479–511, 1996.

[32] Paul B. Callahan and S. Rao Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *J. ACM*, 42 (1):67–90, January 1995. ISSN 0004-5411. doi: 10.1145/200836.200853. URL https://doi.org/10.1145/200836.200853.

[33] Moses Charikar. Similarity estimation techniques from rounding algorithms. In *Proc. 34th Annu. ACM Sympos. Theory Comput.*, pages 380–388, 2002.

[34] Zhi-Long Chen and Han Xu. Dynamic column generation for dynamic vehicle routing with time windows. *Transp. Sci.*, 40:74–88, 2006.

[35] Marek Chrobak and Lawrence L. Larmore. An optimal on-line algorithm for k-servers on trees. *SIAM J. Comput.*, 20(1):144–148, 1991. doi: 10.1137/0220008. URL https://doi.org/10.1137/0220008.

[36] Marek Chrobak and Lawrence L Larmore. Metrical task systems, the server problem and the work function algorithm. *Online algorithms: the state of the art*, pages 74–96, 2005.

[37] Marek Chrobak, Howard J. Karloff, T. H. Payne, and Sundar Vishwanathan. New results on server problems. *SIAM J. Discret. Math.*, 4(2):172–181, 1991. doi: 10.1137/0404017. URL https://doi.org/10.1137/0404017.

[38] Marek Chrobak, H Karloof, Tom Payne, and Sundar Vishwnathan. New results on server problems. *SIAM Journal on Discrete Mathematics*, 4(2):172–181, 1991.

[39] Marek Chrobak, Lawrence L Larmore, Nick Reingold, and Jeffery Westbrook. Page migration algorithms using work functions. *Journal of Algorithms*, 24(1):124–157, 1997.

[40] Christian Coester and Elias Koutsoupias. The online k-taxi problem. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2019, page 1136–1147, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367059. doi: 10.1145/3313276.3316370. URL https://doi.org/10.1145/3313276.3316370.

[41] NYC Taxi Limousine Commission. Tlc trip record data. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page, 2020. [Online; accessed 02-September-2020].

[42] Marco Cuturi. Sinkhorn distances: Lightspeed computation of optimal transport. In *Advances in neural information processing systems*, pages 2292–2300, 2013.

[43] Guy Desaulniers, Fausto Errico, Stefan Irnich, and Michael Schneider. Exact algorithms for electric vehicle-routing problems with time windows. *Operations Research*, 64, 10 2016. doi: 10.1287/opre.2016.1535.

[44] D. Z. Du and Frank K. Hwang. Combinatorial group testing and its applications. In *Combinatorial Group Testing and Its Applications*, 1993.

[45] Ding-Zhu Du and Frank K Hwang. Competitive group testing. *Discrete Applied Mathematics*, 45(3):221–232, 1993.

[46] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017. URL http://archive.ics.uci.edu/ml.

[47] Pavel Dvurechensky, Alexander Gasnikov, and Alexey Kroshnin. Computational optimal transport: Complexity by accelerated gradient descent is better than by sinkhorn's algorithm. In *International conference on machine learning*, pages 1367–1376. PMLR, 2018.

[48] Pavel Dvurechensky, Alexander Gasnikov, and Alexey Kroshnin. Computational optimal transport: Complexity by accelerated gradient descent is better than by sinkhorn's algorithm. In *Proceedings of the 35th International Conference on Machine Learning*, pages 1367–1376, 2018.

[49] Peyman Mohajerin Esfahani and Daniel Kuhn. Data-driven distributionally robust optimization using the wasserstein metric: Performance guarantees and tractable reformulations. *Mathematical Programming*, 171(1):115–166, 2018.

[50] Jittat Fakcharoenphol, Satish Rao, and Kunal Talwar. A tight bound on approximating arbitrary metrics by tree metrics. In *In Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, pages 448–455, 2003.

[51] L. R. Ford Jr and D. R. Fulkerson. A simple algorithm for finding maximal network flows and an application to the hitchcock problem. *No. RAND/P-743*, 1955.

[52] Kyle Fox and Jiashuai Lu. A near-linear time approximation scheme for geometric transportation with arbitrary supplies and spread. In *Proc. 36th Annual Symposium on Computational Geometry*, pages 45:1–45:18, 2020.

[53] H. N. Gabow and R.E. Tarjan. Faster scaling algorithms for network problems. *SIAM J. Comput.*, 18:1013–1036, October 1989. ISSN 0097-5397. doi: 10.1137/0218069. URL http://portal.acm.org/citation.cfm?id=75795.75806.

[54] Harold N. Gabow and Robert E. Tarjan. Faster scaling algorithms for network problems. *SIAM Journal on Computing*, 18(5):1013–1036, October 1989. ISSN 0097-5397, 1095-7111. doi: 10.1137/0218069.

[55] Akshaykumar Gattani, Sharath Raghvendra, and Pouyan Shirzadian. A robust exact algorithm for the euclidean bipartite matching problem. *Advances in Neural Information Processing Systems*, 36, 2024.

[56] Aude Genevay, Gabriel Peyre, and Marco Cuturi. Learning generative models with sinkhorn divergences. *In International Conference on Artificial Intelligence and Statistics*, page 1608–1617, 2018.

[57] Wenshuo Guo, Nhat Ho, and Michael Jordan. Fast algorithms for computational optimal transport and wasserstein barycenter. *In International Conference on Artificial Intelligence and Statistics*, pages 2088–2097, 2020.

[58] Sariel Har-Peled. *Geometric approximation algorithms*. American Mathematical Soc., 2011.

[59] Hideki Hashimoto, Toshihide Ibaraki, Shinji Imahori, and Mutsunori Yagiura. The vehicle routing problem with flexible time windows and traveling times. *Discrete Applied Mathematics*, 154(16):2271–2290, 2006. ISSN 0166-218X. doi: https://doi.org/10.1016/j.dam.2006.04.009. URL https://www.sciencedirect.com/science/article/pii/S0166218X06001879. Discrete Algorithms and Optimization, in Honor of Professor Toshihide Ibaraki at His Retirement from Kyoto University.

[60] J. Hopcroft and R. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[61] P. Indyk. A near linear time constant factor approximation for Euclidean bichromatic matching (cost). In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 39–42, 2007.

[62] Piotr Indyk. A near linear time constant factor approximation for euclidean bichromatic matching (cost). In *SODA 2007*, page 4, 2007.

[63] Arun Jambulapati, Aaron Sidford, and Kevin Tian. A direct $\tilde{O}(1/\epsilon)$ iteration parallel algorithm for optimal transport. *arXiv preprint arXiv:1906.00618*, 2019.

[64] Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *J. ACM*, 24(1):1–13, 1977. doi: 10.1145/321992.321993. URL https://doi.org/10.1145/321992.321993.

[65] Haim Kaplan, Wolfgang Mulzer, Liam Roditty, Paul Seiferth, and Micha Sharir. Dynamic planar Voronoi diagrams for general distance functions and their algorithmic applications. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2495–2504. Society for Industrial and Applied Mathematics, January 2017. ISBN 978-1-61197-478-2. doi: 10.1137/1.9781611974782.165.

[66] Andrey Boris Khesin, Aleksandar Nikolov, and Dmitry Paramonov. Preconditioning for the geometric transportation problem. *arXiv preprint arXiv:1902.08384*, 2019.

[67] Elias Koutsoupias. The k-server problem. *Comput. Sci. Rev.*, 3(2):105–118, 2009. doi: 10.1016/j.cosrev.2009.04.002. URL https://doi.org/10.1016/j.cosrev.2009.04.002.

[68] Elias Koutsoupias and Christos H. Papadimitriou. On the k-server conjecture. *Journal of the ACM (JACM)*, 42(5):971–983, 1995.

[69] Harold Kuhn. Variants of the hungarian method for assignment problems. *Naval Research Logistics*, 3(4):253–258, 1956.

[70] Harold W Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quart.*, 2(1-2):83–97, 1955.

[71] Harold W. Kuhn. The hungarian method for the assignment problem. In Michael Jünger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A. Wolsey, editors, *50 Years of Integer Programming 1958-2008 - From the Early Years to the State-of-the-Art*, pages 29–47. Springer, 2010. doi: 10.1007/978-3-540-68279-0_2. URL https://doi.org/10.1007/978-3-540-68279-0_2.

[72] Nathaniel Lahn and Sharath Raghvendra. A faster algorithm for minimum-cost bipartite matching in minor-free graphs. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 569–588. SIAM, 2019.

[73] Nathaniel Lahn and Sharath Raghvendra. An $O(n^{5/4})$ time ε-approximation algorithm for RMS matching in a plane. *In Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms*, pages 869–888, 2021.

[74] Nathaniel Lahn, Deepika Mulchandani, and Sharath Raghvendra. A graph theoretic additive approximation of optimal transport. In *Advances in Neural Information Processing Systems*, pages 13813–13823, 2019.

[75] James R. Lee. Fusible hsts and the randomized k-server conjecture. In Mikkel Thorup, editor, *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018*, pages 438–449. IEEE Computer Society, 2018. doi: 10.1109/FOCS.2018.00049. URL https://doi.org/10.1109/FOCS.2018.00049.

[76] Tianyi Lin, Nhat Ho, and Michael I. Jordan. On efficient optimal transport: An analysis of greedy and accelerated mirror descent algorithms. *CoRR*, abs/1901.06482, 2019. URL http://arxiv.org/abs/1901.06482.

[77] Huidong Liu, G. U. Xianfeng, and Dimitris Samaras. A two-step computation of the exact gan wasserstein distance. *In International Conference on Machine Learning*, pages 3159–3168, 2018.

[78] Lyft. Matchmaking in lyft line. https://eng.lyft.com/matchmaking-in-lyft-line-9c2635fe62c4, 2016. [Online; accessed 02-September-2020].

[79] Shuo Ma, Yu Zheng, and Ouri Wolfson. Real-time city-scale taxi ridesharing. *IEEE*

Trans. on Knowl. and Data Eng., 27(7):1782–1795, jul 2015. ISSN 1041-4347. doi: 10.1109/TKDE.2014.2334313. URL https://doi.org/10.1109/TKDE.2014.2334313.

[80] Mark S. Manasse, Lyle A. McGeoch, and Daniel Dominic Sleator. Competitive algorithms for server problems. *J. Algorithms*, 11(2):208–230, 1990. doi: 10.1016/0196-6774(90)90003-W. URL https://doi.org/10.1016/0196-6774(90)90003-W.

[81] Snežana Mitrović-Minić, Ramesh Krishnamurti, and Gilbert Laporte. Double-horizon based heuristics for the dynamic pickup and delivery problem with time windows. *Transportation Research Part B: Methodological*, 38(8):669–685, 2004. ISSN 0191-2615. doi: https://doi.org/10.1016/j.trb.2003.09.001. URL https://www.sciencedirect.com/science/article/pii/S019126150300095X.

[82] Marcin Mucha and Piotr Sankowski. Maximum matchings via gaussian elimination. *45th Annual IEEE Symposium on Foundations of Computer Science*, pages 248–255, 2004.

[83] Masayo Ota, Huy Vo, Claudio Silva, and Juliana Freire. A scalable approach for data-driven taxi ride-sharing simulation. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 888–897, 2015. doi: 10.1109/BigData.2015.7363837.

[84] Masayo Ota, Vo Huy, Claudio Silva, and Juliana Freire. Stars: Simulating taxi ride sharing at scale. *IEEE Transactions on Big Data*, PP:1–1, 11 2016. doi: 10.1109/TBDATA.2016.2627223.

[85] Caroline Privault and Gerd Finke. k-server problems with bulk requests: an application to tool switching in manufacturing. *Annals of Operations Research*, 96(1):255–269, 2000.

[86] Kent Quanrud. Approximating optimal transport with linear programs. *arXiv preprint arXiv:1810.05957*, 2018.

[87] Kent Quanrud. Approximating optimal transport with linear programs. In *2nd Symposium on Simplicity in Algorithms*, volume 69, pages 6:1–6:9, 2019.

[88] Prabhakar Raghavan. A statistical adversary for on-line algorithms. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 7:79–83, 1991.

[89] Sharath Raghvendra and Pankaj K. Agarwal. A near-linear time ε-approximation algorithm for geometric bipartite matching. *Journal of the ACM*, 67(3):1–19, June 2020. ISSN 0004-5411, 1557-735X. doi: 10.1145/3393694.

[90] Sharath Raghvendra and Pankaj K. Agarwal. A near-linear time ε-approximation algorithm for geometric bipartite matching. *J. ACM*, 67(3):18:1–18:19, 2020. doi: 10.1145/3393694. URL https://doi.org/10.1145/3393694.

[91] Tomislav Rudec and Robert Manger. A new approach to solve the k-server problem based on network flows and flow cost reduction. *Comput. Oper. Res.*, 40(4):1004–1013, 2013. doi: 10.1016/j.cor.2012.11.006. URL https://doi.org/10.1016/j.cor.2012.11.006.

[92] Tomislav Rudec, Alfonzo Baumgartner, and Robert Manger. A fast work function algorithm for solving the k-server problem. *Central European Journal of Operations Research*, 21:187–205, 2013.

[93] Tomislav Rudec, Alfonzo Baumgartner, and Robert Manger. A fast work function algorithm for solving the k-server problem. *Central Eur. J. Oper. Res.*, 21(1): 187–205, 2013. doi: 10.1007/s10100-011-0222-7. URL https://doi.org/10.1007/s10100-011-0222-7.

[94] Tim Salimans, Han Zhang, Alec Radford, and Dimitris Metaxas. Improving gans using optimal transport. *In International Conference on Learning Representations*, 2018.

[95] Paolo Santi, Giovanni Resta, Michael Szell, Stanislav Sobolevsky, Steven H. Strogatz, and Carlo Ratti. Quantifying the benefits of vehicle pooling with shareability networks. *Proceedings of the National Academy of Sciences*, 111(37):13290–13294, 2014. doi: 10.1073/pnas.1403657111. URL https://www.pnas.org/doi/abs/10.1073/pnas.1403657111.

[96] R. Sharathkumar. A sub-quadratic algorithm for bipartite matching of planar points with bounded integer coordinates. In *29th International Symposium on Computational Geometry*, pages 9–16, 2013. doi: 10.1145/2462356.2480283. URL https://doi.org/10.1145/2462356.2480283.

[97] R. Sharathkumar and P. K. Agarwal. Algorithms for transportation problem in geometric settings. In *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 306–317, 2012.

[98] R Sharathkumar and Pankaj K Agarwal. Algorithms for the transportation problem in geometric settings. In *Proc. 23rd Annual ACM-SIAM Sympos. Discrete Algo.*, pages 306–317. SIAM, 2012.

[99] Jonah Sherman. Generalized preconditioning and undirected minimum-cost flow. In *In Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 772–780, 2017.

[100] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, feb 1985. ISSN 0001-0782. doi: 10.1145/2786.2793. URL https://doi.org/10.1145/2786.2793.

[101] Daniel D Sleator and Robert E Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[102] Uber. Better matching matters. https://marketplace.uber.com/matching, 2020. [Online; accessed 02-September-2020].

[103] Pravin M Vaidya. Geometry helps in matching. *SIAM J. on Comput.*, 18(6):1201–1225, 1989.

[104] Jan van den Brand, Danupon Nanongkai Yin-Tat Lee, Richard Peng, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. Bipartite matching in nearly-linear time on moderately dense graphs. *IEEE 61st Annual Symposium on Foundations of Computer Science*, pages 919–930, 2020.

[105] M.M. Vazifeh and P. Santi. Addressing the minimum fleet problem in on-demand urban mobility. *Nature 557, 534–538*, May 2018. URL https://doi.org/10.1038/s41586-018-0095-1.

[106] Dominic Widdows, Jacob Lucas, Muchen Tang, and Weilun Wu. Grabshare: The construction of a realtime ridesharing service. In *2017 2nd IEEE International Conference on Intelligent Transportation Engineering (ICITE)*, pages 138–143. IEEE, 2017.

[107] K. I. Wong and Michael G. H. Bell. The optimal dispatching of taxis under congestion: A rolling horizon approach. *Journal of Advanced Transportation*, 40(2):203–220, 2006. doi: https://doi.org/10.1002/atr.5670400207. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/atr.5670400207.

[108] Jian Yang, Patrick Jaillet, and Hani Mahmassani. Real-time multivehicle truckload pickup and delivery problems. *Transportation Science*, 38:135–148, 05 2004. doi: 10.1287/trsc.1030.0068.