# Empirical Analysis of Algorithms for the $k-$Server and Online Bipartite Matching Problems

Rutvij Sanjay Mahajan

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Masters of Science

in

Computer Engineering

Anil Vullikanti, Chair

Sharath Raghvendra

Pratap Tokekar

June 25, 2018

Blacksburg, Virginia

Keywords: $k$-Server Problem, Work Function Algorithm, Bipartite Matching, Assignment Problem

Copyright 2018, Rutvij Sanjay Mahajan

Empirical Analysis of Algorithms for the $k-$Server and Online Bipartite Matching Problems

Rutvij Sanjay Mahajan

(ABSTRACT)

The $k$–server problem is of significant importance to the theoretical computer science and the operations research community. In this problem, we are given $k$ servers, their initial locations and a sequence of $n$ requests that arrive one at a time. All these locations are points from some metric space and the cost of serving a request is given by the distance between the location of the request and the current location of the server selected to process the request. We must immediately process the request by moving a server to the request location. The objective in this problem is to minimize the total distance traveled by the servers to process all the requests.

In this thesis, we present an empirical analysis of a new online algorithm for $k$-server problem. This algorithm maintains two solutions, online solution, and an approximately optimal offline solution. When a request arrives we update the offline solution and use this update to inform the online assignment. This algorithm is motivated by the Robust-Matching Algorithm [RM-Algorithm, Raghvendra, APPROX 2016] for the closely related online bipartite matching problem. We then give a comprehensive experimental analysis of this algorithm and also provide a graphical user interface which can be used to visualize execution instances of the algorithm. We also consider these problems under stochastic setting and implement a lookahead strategy on top of the new online algorithm.

# Empirical Analysis of Algorithms for the $k-$Server and Online Bipartite Matching Problems

Rutvij Sanjay Mahajan

(GENERAL AUDIENCE ABSTRACT)

Motivated by real-time logistics, we study the online versions of the well-known bipartite matching and the $k$-server problems. In this problem, there are servers (delivery vehicles) located in different parts of the city. When a request for delivery is made, we have to immediately assign a delivery vehicle to this request without any knowledge of the future. Making cost-effective assignments, therefore, becomes incredibly challenging.

In this thesis, we implement and empirically evaluate a new algorithm for the $k$-server and online matching problems.

# Dedication

*Dedicated to my parents*

# Acknowledgments

First and foremost, I wish to thank my advisor Dr. Sharath Raghvendra for his guidance throughout my masters. His consistent support and sound advice during the course of this research will remain as a solid foundation in my future professional career. I am grateful to him for always steering me in the right direction with his innovative ideas, great knowledge, and experience. In the past year, he has not only helped me grow as a researcher but also as a professional with his mentorship. Lastly, I wish to thank him for being so patient with me, as without his support this thesis would not have been possible. I wish to thank the experts on my review committee Dr. Anil Vullikanti and Dr. Pratap Tokekar for taking time out of their busy schedule to read this thesis and providing valuable feedback. I am grateful to Dr. Abhijin Adiga for all his help during the early phases of this project. I will forever be indebted to my family back home for being the constant source of encouragement and support in my life. I would also like to thank all my friends in Blacksburg: Abhishek, Abhijeet, Anand, Kapil, Sahil, Varun, Suraj, Amit, Omkar and Shantanu who made the last two years, the best years of my life.

# Contents


**List of Figures** ix

**List of Tables** xi

**1 Introduction** 1

**2 Problem Description** 5

2.1 $k-$Server Problem . . . . . 5

2.1.1 Online Models of Computation . . . . . 7

2.1.2 Brief History of the Problem . . . . . 8

2.2 Bipartite Matching . . . . . 9

2.2.1 Brief History of the Problem . . . . . 10

**3 Previous Work** 12

3.1 Online Algorithm for Minimum Metric Bipartite Matching . . . . . 12

3.1.1 Background . . . . . 12

3.1.2 RM-Algorithm . . . . . 13

3.2 The Work Function Algorithm . . . . . 14

3.3 The Stochastic Model . . . . . 15

4 **Robust Algorithm for the $k-$Server Problem** 17

4.1 Preliminaries . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

4.2 Algorithm Details . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

4.3 Intuition Behind the Algorithm . . . . . . . . . . . . . . . . . . . . . 26

4.4 Optimizations to reduce time complexity . . . . . . . . . . . . . . . . 29

4.4.1 Bellman-Ford Optimization . . . . . . . . . . . . . . . . . . . 29

4.4.2 Residual Graph Alteration . . . . . . . . . . . . . . . . . . . . 30

4.5 Experiment Details and Results . . . . . . . . . . . . . . . . . . . . . 31

5 **Lookahead Algorithm** 39

5.1 Previous Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

5.2 Algorithm Details . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40

5.3 Experiment Details and Results . . . . . . . . . . . . . . . . . . . . . 41

5.3.1 Lookahead Algorithm for Bipartite Matching Problem . . . . . . . . . 41

5.3.2 Lookahead Algorithm for k-Server Problem . . . . . . . . . . . . . . . 43

6 **Graphical User Interface** 45

6.1 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45

6.1.1 MVC architecture . . . . . . . . . . . . . . . . . . . . . . . . . 45

6.2 GUI Components . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47

6.2.1 Outputs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47

6.2.2 Inputs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48

6.3 Types of Clusters . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49

**7 Conclusion** **51**

**Bibliography** **52**

# List of Figures


2.1 Bipartite graph with set of servers S and requests R. . . . . . . . . . . . . . . 10

4.1 Solution $\sigma = \{C^{(1)}, ...., C^{(5)}\}$ along with final configuration $C^{(f)}$ . . . . . . . 19

4.2 Construction Instance of Residual Graph . . . . . . . . . . . . . . . . . . . . . 21

4.3 Offline optimal solution before $\sigma_i^*$ and after $\sigma_{i+1}^*$ the arrival of $r_{(i+1)}$ . . . . . 22

4.4 Non-trivial alternating path in $\sigma_i \oplus \sigma_{(i+1)}$. All red edges are part of solution $\sigma_i$ and all the blue edges are not part of $\sigma_i$ . . . . . . . . . . . . . . . . . . . 23

4.5 Example of Net-Cost for an alternating . . . . . . . . . . . . . . . . . . . . . . 24

4.6 Example of $t$-Net-Cost for an alternating . . . . . . . . . . . . . . . . . . . . 24

4.7 Execution Instance of the $k$-server problem with $k = 3$ . . . . . . . . . . . . 28

4.8 Graph showing the mean ratio of costs in serving 1000 requests: Random Distribution . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32

4.9 Graph showing the mean ratio of costs in serving 1000 requests: Gaussian Distribution . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

4.10 Graph showing the mean ratio of costs in serving 1000 requests: Gaussian+Random Distribution . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

4.11 Graph showing the mean ratio of costs in serving 1000 requests: Multiple Clusters . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

4.12 Graph showing the mean ratio of costs in serving 1000 requests: Power Law Distribution . . . . . 35

4.13 Graph showing the mean ratio of costs in serving 1000 requests: Exponential Distribution . . . . . 35

4.14 Graph showing the mean ratio of costs in serving 1000 requests . . . . . 36

4.15 Graph showing the mean ratio of costs in serving 150 requests . . . . . 37

4.16 Runtime of the algorithm with and without Optimization . . . . . 38

4.17 Ratio of Cost With Optimization to Cost Without Optimization . . . . . 38

5.1 Graphs showing the mean ratio of costs, averaged across 10 repetitions, for various different values of $\eta$ . . . . . 42

5.2 Graphs showing the mean ratio of costs, averaged across 10 repetitions, for various different values of $\eta$ . . . . . 44

6.1 Three Elements: MODEL, VIEW and CONTROLLER . . . . . 46

6.2 Graphical User Interface . . . . . 47

# List of Tables

6.1 Request distribution in Different Clusters . . . . . . . . . . . . . . . . . . . . . 50

# Chapter 1

# Introduction

Problems arising in logistics such as the bipartite matching problem, the traveling salesman problem, the transportation problem, the transshipment problem, and the $k$-server problem have been widely studied for many decades. These problems find use in a wide array of applications including operating systems, operations research [22], mathematics [6] and economics [21]. With the advent of e-commerce, real-time delivery of goods and services have become an integral part of our lives. Therefore, design of efficient algorithms that produce high-quality results is very important.

For instance, consider the $k$-server problem. In this problem, there are $k$ servers positioned at a fixed set of $k$ points in some metric space, We call this the *initial configuration*. Then in each time step, we get a request at some location of the metric space. We need to move one of the $k$-servers from its current location to the request location. We assume that the new requests appear only after we have processed the current request. The goal in this problem is to minimize the total distance traveled by the servers. [14]. We have formally defined this problem in Chapter 2. There are many real-world applications of this problem. We discuss few of them below.

1. **Ride Sharing Service:** This service exists in the form of two-sided market. For example, Uber has a goal to provide timely and fast service to the *customers* and at the same time they have to generate sizable number of transactions for the drivers.

Furthermore, the customers of these services always outnumber the workers. Therefore, for these services to stay profitable, it is important to minimize the distance traveled by the workers to serve all the customers. This problem of ride-sharing can be seen as an variant of the $k$-server problem with an additional constraint of load balancing. The drivers, in this case, can be seen as the $k$ servers and the location of the customers can be considered as the request locations. Note that, companies such as Uber store historical data using which they can build fairly reliable stochastic model for request generation.

2. **Paging Problem:** This problem arises in the operating systems of a computer. Computers have two types of memories: *cache*, which is small and it provides quick access to the data stored within it, while the other one, *main memory*, is slow to access but has a large size. Each of these memories can store a number of pages. The cache contains a copy of pages present in main memory and is used to access this data quickly. The input, in this case, is a series of data requests. The data request can be mapped to a page reference. Whenever such a request is received, the cache is checked for the corresponding page. If the page is not found we need to pay an extra cost in terms of time to bring the data from slow memory to fast memory. Herein lies the problem, as there is limited space in the fast memory, a decision has to be made to evict a block of fast memory to make space for this newly arrived chunk of data. The goal when dealing with such a problem is to minimize the overall processing time given a sequence of data requests and the challenge is to do that without any knowledge of future data requests. If we are provided with the luxury of knowing the future requests sequence before the computation of the algorithm starts, we can come up with a smart algorithm that takes best decisions at every step. Such an algorithm is called as *offline* algorithm. But unfortunately, we do not have such a luxury. In practice, any cache

eviction algorithm is compelled to make its decisions in an *online* fashion: as soon as data request arrives and corresponding page is not present in the cache, this algorithm must evict data from the cache to make space for this new page before any future request arrives.

3. **Emergency Supply Chain:** The emergency supply chain (ESC) is to locate points of emergency equipments and supplies and to relieve those in need of food and medical care promptly [18]. The key challenge in ESC system is to provide timely service to the affected people given limited resources and uncertainty in the real-time relief demand information. Therefore, the objective of ESC is to improve the performance while minimizing the response time. This can also be seen as a variant of the $k$-server problem, where the service providers can be seen as the $k$-servers and the relief locations can be considered as the request locations. Building stochastic models for requests in emergency situations may be difficult. Furthermore, wrong stochastic assumptions can lead to a substantially increased response time.

These real world adaptations of the problems in logistics offer various challenges in terms of quality and scalability. The user base of services like ride-sharing is constantly increasing, making it important to scale the performance of the existing solutions without affecting the quality of the solutions. Furthermore, these real-time services cannot rely upon future knowledge of inputs. They are required to give instant responses to their customers without full knowledge of the inputs, making it extremely essential to provide online solutions that perform well in the real-time environment.

The theoretical computer science community has studied the online $k$-server problem extensively and they mainly focused on analyzing the algorithm in adversarial settings. To prevent certain worst case instances these solutions, tend to be pessimistic, leading to sub-optimal

performance in realistic setting. The Operational Research community proposes policies based on some weak, yet often practical *request generation model*. However these solutions fail to deliver optimal performance when the requests do not obey this model, which can happen in realistic settings. A major challenge, therefore, is to design robust online and offline algorithms that simultaneous perform optimally under adversarial and stochastic request generation model. In this thesis, we propose a Robust $k$-server Algorithm (RK-Algorithm) that we empirically show to simultaneously perform well in various settings.

In Chapter 2 we define the $k$-server problem and the bipartite matching problem and discuss some of the previous work related to these problems. In Chapter 3, we describe the algorithms and terminologies required to describe the RK-Algorithm. In Chapter 4, we provide an implementation of the RK-Algorithm. We also present a detailed experimental analysis of this algorithm. In Chapter 5, we provide a lookahead algorithm for both the $k$-server problem and the online bipartite matching problem under stochastic setting. In Chapter 6, we give a detailed implementation of the Graphical User Interface, which is used as a framework to test these algorithms. We conclude and state future work in Chapter 7.

# Chapter 2

# Problem Description

## 2.1 $k-$Server Problem

The online $k$-server problem was introduced by Manasse et al. [17], and has since become the cornerstone of the area of online problems.

Consider a *pseudo-metric space* consisting of $k$-servers, i.e., initially, each of the $k$-servers is placed at some point in this metric space. This configuration of servers is called the *initial configuration* $C^{(0)}$. Then at each time step, a request is generated at some point $q$ in this metric space. One of the servers has to move to $q$ in order to process the request. Other servers are also free to move to their desired locations in order to better serve the future request sequence. The cost incurred to serve this request is the distance traveled by the server to move to point $q$ along with the cost incurred by all other servers to move to their desired locations. The cost to serve all the requests in the sequence is the sum of all the costs incurred while serving each request. Our objective is to come up with an online algorithm that minimizes the total cost incurred by the $k$-servers.

Before we define this problem formally we need to look at few notations, terminologies, and definitions.

A pseudo-metric space is a pair $(Q, d)$ where $Q$ is a set of points and $d : Q \times Q \rightarrow [0, \infty)$, is a non-negative distance function satisfying the triangle inequality:

$$d(x,y) + d(y,z) \geq d(x,z) \qquad \text{for all } x,y,z \text{ in } Q$$

Also, $d(x,x) = 0$, and $d(x,y) = d(y,x)$. This differs from a normal metric space in that it does not require $d(x,y) = 0$ to imply $x = y$.

▷ **Definition 2.1.** *Fix some integer $k > 0$ and let $(Q,d)$ be a pseudo-metric space. A $k$-configuration for a pseudo metric space $(Q,d)$ is any set $C$ of size $k$ over $Q$ i.e., $C = \{c_1, ..., c_k\} \subseteq Q$.*

*We define a distance from a request $x$ with respect to a configuration to be the distance to the nearest point in $C$:*

$$d(x,C) = d(x,c_i) \qquad \textit{where } x \in Q \textit{ and } c_i \in C$$

*Given two configurations $C^{(A)}$ and $C^{(B)}$ in $(Q,d)$, the distance $d(C^{(A)}, C^{(B)})$, is the minimum-weight perfect matching of the locations in configurations $C^{(A)}$ and $C^{(B)}$.*

▷ **Definition 2.2.** ($k$-server Problem) *Given a sequence of $n$ requests $R = (r_1, ...., r_n)$ in a metric space $(Q,d)$ and initial location of $k$-servers $C^{(0)} = \{s_1^{(0)}, ...., s_k^{(0)}\} \in Q$, where $s_j^{(0)}$ specifies the initial location of the $j$-th server, a candidate solution is a sequence of configurations $\sigma = \{C^{(1)}, ...., C^{(n)}\}$ such that $r_i \in C^{(i)}$ for all $i = 1, ...., n$. The servers start in configuration $C^{(0)}$ and process requests in $R$ by moving through configurations $\sigma = \{C^{(1)}, ...., C^{(n)}\}$. The objective of the $k$-server problem is to find a candidate solution $\sigma$ that minimizes the following cost function.*

$$w(\sigma) = \sum_{i=0}^{n-1} d(C^{(i)}, C^{(i+1)})$$

The question that lies at the heart of any competitive analysis is how good is an online algorithm when compared to an optimal algorithm. The optimal algorithm has the knowledge of future requests.

▷ **Definition 2.3.** *An online algorithm $A$ for the $k$-server problem is called $\alpha$-competitive if for any initial configuration $C^{(0)}$ and any input request sequence $R = (r_1, ...., r_n)$:*

$$cost_A(R) \leq \alpha \cdot cost_{OPT}(R) + \phi(C^{(0)})$$

*where $\alpha > 0$, $cost_{OPT}(R)$ is the cost that the optimal offline algorithm pays for the processing of $S$ starting with initial configuration $C^{(0)}$. The constant term $\phi(C^{(0)})$, is independent of the sequence of request and is used to eliminate the dependency on the initial configuration $C^{(0)}$.*

### 2.1.1 Online Models of Computation

There are several well-studied models for request generation. In the *adversarial model*, there is an adversary who knows the current server locations as well as the decisions made by the algorithm and it generates a sequence of requests which maximize the competitive ratio $\alpha$. In the closely related *oblivious adversary model*, the adversary generates a sequence of requests without being able to observe the random choices made by the algorithm. Worst case analysis using these models often lead to over-pessimistic results. So Adiga, Friedman and Raghvendra [15] studied the *random arrival model*. In this model the adversary chooses the set of request locations $R$ before the algorithm execution begins, but the arrival order of the requests is a permutation chosen uniformly at random from the set of all possible permutations of $R$. In most of the cases, like the ride sharing problem described in Chapter 1, it may be useful to assume that the request locations are independently and identically distributed from a known or an unknown distribution.

The existing solutions are considered to be optimal if they are able to minimize worst case performance for all input instances. Therefore, it is important to have algorithms for the

$k$-server problem that simultaneously achieves a near-optimal performance for every input instance.

### 2.1.2 Brief History of the Problem

The problem was first defined by Manasse, McGeogh, and Sleator in [17]. Few years down the line Tarjan and Sleator analyzed problems like paging problem and list update problem and came up with a framework of competitive analysis for online algorithm [21]. This framework was used to study snoopy caching during which the term competitive analysis was formalized [10, 11]. Manasse, McGeogh, and Sleator also proved few important results, no online algorithm can have a competitive ratio less than $k$ and this is independent of the metric space as long as the metric space has at least $k+1$ points [17]. Furthermore, based on the proof that competitive ratio for a special case of the $k$-server problem of $k=2$ is 2, they put forth the $k$-server conjecture:

▹ **Conjecture 2.4.**(The $k$-server conjecture). *For every metric space with more than $k$ distinct points, the competitive ratio of the $k$-server problem is exactly $k$.*

A very significant breakthrough was achieved by [16], which put forth a deterministic online algorithm with a competitive ratio of at most $2k-1$ for any metric space. This remains the best known bound. More recent developments for the $k$-server problem are about randomized algorithms. The second conjecture about the competitive ratio of algorithms for the $k$-server problem in the oblivious adversary model was put forth by [13].

▹ **Conjecture 2.5.**(The randomized $k$-server conjecture). *For every metric space, there is a randomized online algorithm for the $k$-server problem with a competitive ratio of $O(\log k)$.*

Two other algorithms that have been widely studied are the Greedy and Retrospective algorithm [16]. Both of these algorithms have an unbounded competitive ratio. Greedy algorithm

moves the closest server to serve the request during each step. While the retrospective algorithm moves the servers so as to minimize the $k$-server problem over the past requests. Both of these algorithms do not have bound on their competitive ratio.

Bansal, Buchbinder, Madry and Naor [2] gave the first polylogarithmic-competitive randomized online algorithm for the $k$-server problem on an discrete metrics that comprise $n$ points. This algorithm achieves a competitive ratio of $O(\log^3 n \log^2 k)$. Adiga, Friedman and Raghvendra [15] gave a $O(\alpha)$ competitive algorithm for the random arrival model where $\alpha$ is a lower bound on the competitive ratio of any online algorithm in this model. Furthermore, Raghavan and Snir in [4] were the first to study the Harmonic Algorithm, a memoryless algorithm that moves the servers with a probability inversely proportional to the distance. This randomized online algorithm has a competitive ratio of $O(2^k \log k)$ [3].

Next we consider the online bipartite matching problem and briefly discuss the related work.

## 2.2 Bipartite Matching

When we consider the $k$-server problem, the capacity of every server is $\infty$. The case where every server has a capacity of 1 is the metric bipartite matching problem.

▷ **Definition 2.6.**(Bipartite Matching) *Consider sequence of requests $R = (r_1, ...., r_n)$, a metric space (Q,d) and servers $S = (s_1, ...., s_n)$ and let $S, R \subseteq Q$. Also consider a complete bipartite graph $G = (V, E)$ of two disjoint sets of vertex $S$ and $R$ where $V = S \cup R$ and $E = S \times R$, see Figure 1.1.*

*A matching $M \subseteq S \times R$ is any set of vertex-disjoint edges of the bipartite graph shown in the aforementioned figure. We denote the cost of server $s$ serving request $r$ by $d(s, r)$. The cost of any matching $M$ is given by:*

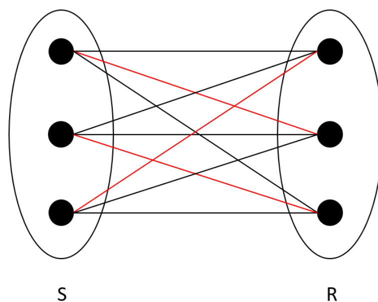


Figure 2.1: Bipartite graph with set of servers S and requests R.

$$w(M) = \sum_{(s,r)\in M} d(s,r)$$

A perfect matching is a matching where every server in S is serving exactly one request in R, i.e., $\mid M \mid = n$. A *minimum-cost perfect matching* is a perfect matching with the minimum cost, it is also know as *minimum cost maximum cardinality matching*.

### 2.2.1 Brief History of the Problem

Same mathematical tools can be applied to both, online $k$-server problem and online bipartite matching problem. The online bipartite matching problem was studied in the adversarial model, oblivious model, and random arrival model. In the adversarial model, there is an optimal $2n-1$ competitive deterministic algorithm by Khuller et al. [12], Kalyansundaram and Pruhs [9], and Raghvendra [20]. While in the oblivious adversary model, there is a $O(poly \log n)$-competitive algorithm. Bansal et al. [1] achieved a $O(\log^2 n)$-competitve algorithm for this problem. Recently $O(\log n)$-competitive algorithm was presented in the random arrival model [20].

Raghvendra [20] presented a deterministic online algorithm called the Robust Matching Algorithm (RM-Algorithm) that simultaneously achieves optimal competitive ratio of $2n-1$ and $2H_n-1$ in the adversarial and random arrival models respectively. Furthermore, Nayyar and Raghvendra [19], analyzed the RM algorithm to show that its competitive ratio is

$O(\mu_M(S)\log^2 n)$ where $\mu_M(S)$ is the worst-case ratio of the cost of the traveling salesman tour and the diameter over every subset of servers $S$ with a positive diameter. For instance when $S$ are points on a line, the competitive ratio is $O(\log^2 n)$. Raghvendra in [22], tightened the analysis to show that the competitive ratio can be improved to $O(\log n)$ for the case when $S$ and $R$ are points on a line.

# Chapter 3

# Previous Work

In this chapter, we will introduce the relevant definitions, describe the algorithm and present some of the details from [20]. We also discuss the stochastic model for the $k$-server problem and the online bipartite matching problem. We then provide describe the Work Function Algorithm which is the best-known algorithm for stochastic $k$-server problem.

## 3.1 Online Algorithm for Minimum Metric Bipartite Matching

The Robust Matching Algorithm (RM-Algorithm) was introduced by Raghvendra in [20]. Before looking at the algorithm we will discuss the necessary terminologies in the following section.

### 3.1.1 Background

Let $S$ and $R$ be the set of server and request locations. A *matching* $M \subseteq S \times R$ is any set of vertex-disjoint edges of the complete bipartite graph $G(S \cup R, S \times R)$ such that the cost of the matching $M$ is given by $w(M) = \sum_{(s,r) \in M} d(s, r)$.

Given a matching $M^*$ on the bipartite graph, an *alternating path* is a simple path whose

edges alternate between those in the matching $M^*$ and those not in the matching $M^*$. Also, any vertex that is not in the matching is referred to as *free vertex*.

Furthermore, an *augmenting path* $P$ is defined as the alternating path between two free vertices. We can augment $M^*$ by one edge along $P$ if we remove the edges of $P \cap M^*$ from $M^*$ and add the edges of $P \setminus M^*$ to $M^*$. The process of augmentation is similar to that of well-known Hungarian method. After augmenting, the new matching can be denoted by $M^* \leftarrow M^* \oplus P$, where $\oplus$ is the symmetric difference operator. This definition of $t$-net cost can be easily extended to alternating paths and cycles as well. For a parameter $t > 1$, we define the $t$-net-cost of an augmenting path $P$ as follows:

$$\phi_t(P) = t \cdot \left( \sum_{(s,r) \in P \setminus M^*} d(s,r) \right) - \sum_{(s,r) \in P \cap M^*} d(s,r)$$

Using these notations, we will now describe the algorithm.

### 3.1.2 RM-Algorithm

For every vertex $v \in S \cup R$, RM-Algorithm maintains a dual weight $y(v)$. At the start of the execution, the dual weight of every vertex is set to 0. During this execution instance, the requests from the set $R$ arrive in an online fashion. Let $r' \in R$ be the request that has not yet arrived. The dual weight for these requests remains to be 0, i.e., $y(r') = 0$. This algorithm also maintains two matchings $M$ and $M^*$; both of these matchings are empty at the start of the execution. Furthermore, $M$ and $M^*$ both match all the request seen so far to servers in $S$. The matching $M^*$ together with the dual weights $y(.)$ is a $t$-approximate matching; for some $t > 1$. We refer to this as the *offline matching*. The set of unmatched servers $S_F$ is the same for both the online matching and offline matching. To process the $i^{th}$ request $r_i$ the algorithm does the following.

**RM-Algorithm:**

1) Computes the minimum t-net-cost augmenting path $P_i$ with respect to the offline matching $M^*$. $P_i$ starts at $r_i$ and ends at some free vertex $s_i \in S_F$.

2) Update $M^*$ by augmenting it along $P$, i.e., $M^* \leftarrow M^* \oplus P$.

3) Update online matching $M$, by matching $s_i$ to $r_i$, i.e., $M \leftarrow M \cup \{(s_i, r_i)\}$.

There is an $O(n^2)$-time primal-dual algorithm to compute such minimum $t$-net-cost path in [20].

The algorithm implemented by us is motivated by the RM-Algorithm. We provide the implementation details and empirical analysis for this algorithm in the Chapter 4.

## 3.2 The Work Function Algorithm

An instance of the $k$-server problem is given by the initial configuration of $k$ servers $C^{(0)} = (s_1^{(0)}, ...., s_k^{(0)})$, where $s_j^{(0)}$ specifies the initial location of the $j$-th server, and with sequence of $n$ requests $R = (r_1, ...., r_n)$, where $r_i$ determines the location of the $i$-th request. In the $i$-th step, an online algorithm is compelled to serve the request $r_i$ by moving a server to the location of $r_i$. There by the current server configuration $C^{(i-1)} = (s_1^{(i-1)}, ...., s_k^{(i-1)})$ transforms to $C^{(i)} = (s_1^{(i)}, ...., s_k^{(i)})$. The server is selected based on the past request or the future distribution or they are based on current request alone. Whenever the algorithm moves a server from a location $p$ to a location $r$, it incurs a cost equal to the distance $d(p, r)$.

To compute the optimal solution, we can focus on initial configuration and the set of requests. For every configuration $X$, we define $w(C^{(0)}; r_1, ...., r_t; X)$ to be the cost of the optimal solution which starts at the initial configuration $C^{(0)}$, passes through points $R = (r_1, ...., r_t)$ and ends up at an configuration $X$ [16].

$$w(C^{(0)}; r_1, ...., r_t; X) = min\{\sum_{i=1}^{t+1} d(C^{(i-1)}, C^{(i)})\} \qquad \text{where } C^{(t+1)} = X$$

In its $i^{th}$ step the work function algorithm serves the request $r_i$ by changing the server configuration from $C^{(i-1)}$ to $C^{(i)}$ such that the following cost function is minimum:

$$w(C^{(i)}) = w(C^{(0)}; r_1, ...., r_t; C^{(i)}) + d(C^{(i-1)}, C^{(i)})$$

Thus the objective function $w(C^{(i)})$ is defined as sum of two parts. The first part is the optimal cost of serving requests $R = (r_1, ...., r_i)$ and ending up in $C^{(i)}$. While the second part is the distance traveled by a server to move from $C^{(i-1)}$ to $C^{(i)}$. Furthermore, this algorithm balances the two extremes of the retrospective and greedy algorithm.

This algorithm was implemented by using dynamic programming. Furthermore, we empirically analyze the performance of the work function algorithm against RK-Algorithm in Chapter 4.

## 3.3 The Stochastic Model

In this thesis, we also test our algorithm in stochastic setting. In this setting, we assume that the requests are drawn from some probability distribution. Consider there are $n$ requests. For every $i \in [1, \ldots, n]$, a discrete probability distribution $P_i$ is given in advance from which request $r_i$ will be drawn at time step $i$.

We generate requests from following distributions: Random Distribution, Gaussian Distribution, Power Law Distribution, Exponential Distribution. We also model more realistic scenarios by assuming that requests can arrive from one of many clusters (also called hot-spots). For example, an Uber demand will probably be from *office centers* and *residential complexes* toward the beginning of the day and in the evening. While places like *airport* and

*shopping malls* have requests coming in throughout the day [5]. Apart from these hot-spots, small number of request come from random locations in the city.

In Chapter 5, we give implementation details and empirical analysis of lookahead algorithm based on this stochastic model.

# Chapter 4

# Robust Algorithm for the $k-$Server Problem

In this chapter, we discuss the implementation details of a RK-Algorithm. We also empirically compare this algorithm against greedy, retrospective and work function algorithms. Furthermore, we also discuss various optimizations that can be applied to this algorithm to improve the runtime.

## 4.1 Preliminaries

▷ **Definition 4.1.** For some $k > 0$, a *k-configuration* for a pseudo metric space $(Q, d)$ is any set $C$ of size k over Q i.e., $C = \{s_1, ..., s_k\} \subseteq Q$. We define a distance from a request $r$ with respect to a configuration $C$ to be the distance to the nearest point in $C$: $d(r, C) = d(r, s_i)$ where $r \in Q$ and $s_i \in C$.

Furthermore, if we are given two configurations $C^{(A)}$ and $C^{(B)}$ in $(Q, d)$, the distance $d(C^{(A)}, C^{(B)})$, is the *minimum-weight perfect matching* of the positions in configurations $C^{(A)}$ and $C^{(B)}$.

▷ **Definition 4.2.** In the $k$-server problem, we are given a sequence of $n$ requests $R = (r_1, ...., r_n)$ in a metric space $(Q, d)$ and initial location of $k$-servers $C^{(0)} = \{s_1^{(0)}, ...., s_k^{(0)}\} \in Q$,

where $s_j^{(0)}$ specifies the initial location of the $j$-th server. A candidate solution for this problem is a sequence of configurations $\sigma = \{C^{(1)}, ...., C^{(n)}\}$ such that $r_i \in C^{(i)}$ for all $i = 1, ...., n$. The servers start in configuration $C^{(0)}$ and process requests in $R$ by moving through configurations $\sigma = \{C^{(1)}, ...., C^{(n)}\}$. The objective of the $k$-server problem is to find a candidate solution that minimizes the following cost function: $w(\sigma) = \sum_{i=0}^{n-1} d(C^{(i)}, C^{(i+1)})$.

▷ **Definition 4.3.** For any algorithm, its *competitive ratio* is the largest ratio of the online solution produced by our algorithm to the minimum-cost offline solution for the same set of $n$ requests. An online algorithm A for the $k$-server problem is called c-competitive if for any initial configuration $C^{(0)}$ and any input request sequence $R = (r_1, ...., r_n)$:

$$cost_A(R) \leq c \cdot cost_{OPT}(R) + \phi(C^{(0)})$$

where $c > 0$, $cost_{OPT}(R)$ is the cost that the optimal offline algorithm pays for the processing of S starting with initial configuration $C^{(0)}$. The constant term $\phi(C^{(0)})$, is independent of the sequence of request and is used to eliminate the dependency on the initial configuration $C^{(0)}$.

**Simplifying a solution to the $k$-server problem:** If we consider any solution to the $k$-server problem to be $\sigma = \{C^{(1)}, ...., C^{(n)}\}$, we can convert it to another solution $\sigma' = \{C^{(1)'}, ...., C^{(n)'}\}$ such that $w(\sigma') \leq w(\sigma)$ and $\sigma'$ has following properties:

1. $r_{(i)} \in C^{(i)'}$ for all $i = 1, ...., n$
2. The combinatorial configuration $C^{(i)'}$ and $C^{(i+1)'}$ differ in exactly one request, i.e., $r_{(i+1)}$ is a request in the combinatorial configuration $C^{(i+1)'}$. Infact the server $C^{(i)'} \backslash C^{(i+1)'}$ is precisely one server which serves the request $r_{(i+1)}$.

Given this, we can view the solution $\sigma$ as a set of $k$ paths, one for each server. The path for each server starts from its position defined in the initial configuration $C^{(0)} = \{s_1^{(0)}, ...., s_k^{(0)}\}$.

▷ **Definition 4.4.** We can define *directed edge* as an edge between any two requests $r_i$ and $r_j$ and this edge is directed from $r_i$ to $r_j$ if and only if $i < j$.

▷ **Definition 4.5.** Given a solution $\sigma = \{C^{(1)}, ...., C^{(n)}\}$, we add a *final configuration* $C^{(f)}$ which in terms of locations is identical to $C^{(n)}$. However, combinatorially we assume that $C^{(f)} \cap C^{(n)} = \phi$. For example, in Figure 4.1, the final configuration $C^{(n)}$ is denoted with labels $f_1, f_2, f_3$ and they have zero cost edges from $C^{(5)}$.

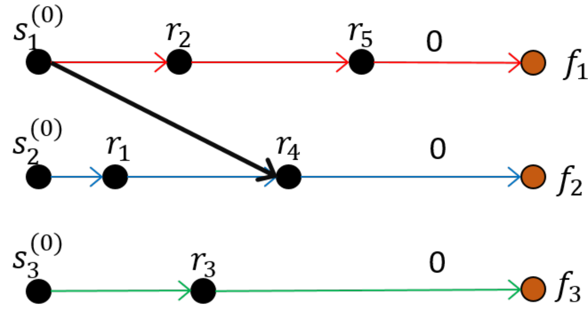


Figure 4.1: Solution $\sigma = \{C^{(1)}, ...., C^{(5)}\}$ along with final configuration $C^{(f)}$

The final configuration provides consistency for the edges in $\sigma$ in a sense that after adding $C^{(f)}$ all the requests have in-degree and out-degree of *one*.

▷ **Definition 4.6.** Consider an edge between two vertices $u$ and $v$, as we walk along that edge we say that edge is *incoming* if it is directed from $v$ to $u$. Otherwise we refer to the edge as *outgoing*. For example, in figure 4.1, edge going from $s_1^{(0)}$ to $r_4$ is an incoming edge with respect to node $r_4$, i.e., if we walk from $r_4$ to $s_1^{(0)}$, the edges between them is directed from $s_1^{(0)}$ to $r_4$.

▷ **Definition 4.7.** Let $\sigma_i$ be a feasible solution to the $k$-server problem. We define an *alternating path* $P$ with respect to $\sigma_i$ as a path that starts from a special vertex $s$ and ends at a vertex $t \in C^{(f)}$. The path $P$ is such that:

1. As we walk along the path from $s$ to $t$, the first edge is an incoming edge, the last edge is an outgoing edge and the edges alternate between incoming and outgoing edges.

2. As we walk along the edges from $s$ to $t$, the first edge is not in $\sigma_i$, the last edge is in $\sigma_i$ and the edges alternate between those in $\sigma_i$ and those not in $\sigma_i$.

*Alternating Cycle* is similar to the alternating path except that the path starts and ends at the same location.

**Switching solution along P:** We can switch the solution from $\sigma_i$ to $\sigma^{'}$ along path P as follows. We add all incoming edges to $\sigma_i$ and remove all the outgoing edges from $\sigma_i$ to obtain $\sigma^{'}$. Furthermore, we add $s$ to the final configuration and remove $t$ from the final configuration. For example, in Figure 4.4, if we switch along the non-trivial augmenting path, we can go from one solution shown in Figure 4.3 to other.

▷ **Definition 4.8.** A *trivial alternating path* is defined as a path with length two, and the end points of this path are from final configurations $C_i^{(f)}$ and $C^{(f)^{'}}$ respectively. $C_i^{(f)}$ and $C^{(f)^{'}}$ represent final configurations before and after switching from $\sigma_i$ to $\sigma^{'}$. The net-cost of this path is always 0.

▷ **Definition 4.9.** *Augmenting path* is an alternating path except that $s$ is a new request that arrived at time step $i+1$. The process of augmentation is similar to that of switching process except that after switching we add an edge from the server at $r_{(i+1)}$ to a new node $q$ in the final configuration. For example, in Figure 4.4, the non-trivial alternating path is also a augmenting path.

▷ **Definition 4.10.** Given a solution $\sigma$, we define *net-cost* of any alternating path $P$ to be:

$$\phi(P) = \left( \sum_{(u,v)\in P\setminus\sigma} d(u,v) \right) - \sum_{(u,v)\in P\cap\sigma} d(u,v)$$

Note that $\sigma^{'}$ is the solution after we switch $\sigma_i$ along the path P. Then $w(\sigma^{'}) - w(\sigma_i) = \phi(P)$, i.e., the change in the cost of the solution is precisely the net-cost of the path $P$.

**Residual Graph Representation:** Next we present a graph $G_\sigma$ which is a directed graph such that any directed path in $G_\sigma$ can be mapped to an alternating path with respect to $\sigma$ and for every alternating path with respect to $\sigma$, there is a directed path in $G_\sigma$.

The vertex set for $G_\sigma$ consists of all requests and vertices in the initial configuration $C^{(0)}$ and final configuration $C^{(f)}$. We also add a special vertex $s$ which will be the start vertex for the alternating path.

For any request $r_i$, let $P$ be the path of the server that contains $r_i$. Let $N(r_i)$ be the request after $r_i$ along this path. If $r_i$ is the last request then $N(r_i)$ is the vertex corresponding to the final configuration $C^{(f)}$. The edge set consists of the following:

1. We add edge from $s$ to $N(r_i)$ for every $i$.

2. For every request $r_i$, we add an edge to $N(r_j)$ for all $j < i$.

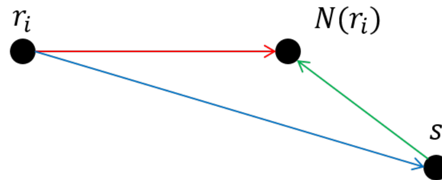

Figure 4.2: Construction Instance of Residual Graph

It is easy to see that any directed path in $G_\sigma$ corresponds to an alternating path with respect to $\sigma$. For example, in Figure 4.2, the edge shown by green color is part of the residual graph and we can map that edge to an alternating path starting from $s$ and ending at $N(r_i)$.

We consider two optimal offline solutions, let $\sigma_i$ be the optimal offline solution for first $i$ requests and let $\sigma_{(i+1)}$ be the optimal offline solution for first $i+1$ requests. Furthermore, let $C_i^{(f)} = \{f_1, ...., f_k\}$ be the final state for solution $\sigma_i$ and let $C_{(i+1)}^{(f)} = \{f_1', ...., f_k'\}$ be the final state for solution $\sigma_{(i+1)}$.

▷ **Lemma 4.11.** *The symmetric difference of $\sigma_i$ and $\sigma_{(i+1)}$ contains only one non-trivial alternating path.*

*Proof.* In-degree of the new request $r_{(i+1)}$ is one, so $r_{(i+1)}$ can be contained only in one alternating path. If there is any other non-trivial alternating path in the symmetric difference, it has to be contained within the first $i$ requests. If there was a non-trivial alternating path in the first $r_i$ requests that reduced the cost of the solution $\sigma_i$; then $\sigma_i$ can improve the overall cost of the solution by switching along this path. But $\sigma_i$ is optimal offline solution, so there cannot be another non-trivial alternating path in the $\sigma_1 \oplus \sigma_{(i+1)}$.

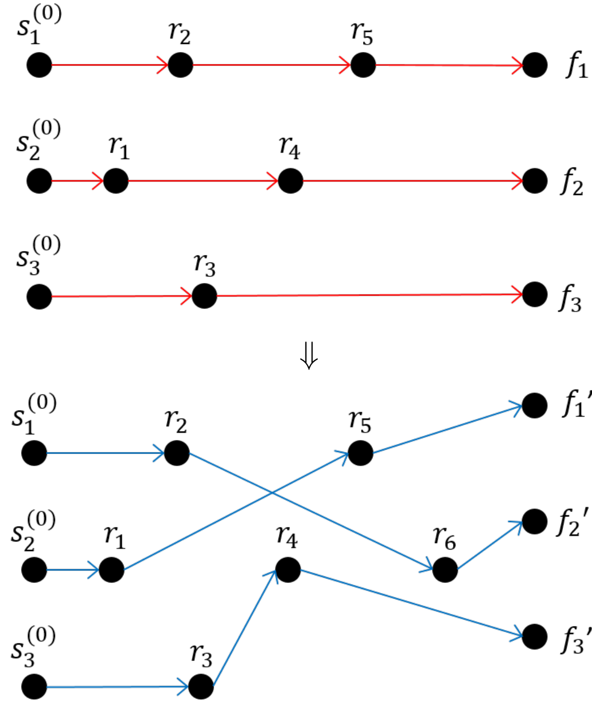


Figure 4.3: Offline optimal solution before $\sigma_i^*$ and after $\sigma_{i+1}^*$ the arrival of $r_{(i+1)}$

**Optimal Offline Solution:** Let the solution $\sigma^*$ be the optimal offline solution. To begin with, $\sigma^*$ is empty, i.e., $\sigma^* = \phi$. Given a new request $r_{(i+1)}$, we compute the minimum net-cost augmenting path $P$ with respect to the solution $\sigma^*$. This augmenting path starts at $r_{(i+1)}$ and ends at $f$, where $f \in C^{(f)}$ and $C^{(f)}$ is the final configuration. We then update the solution $\sigma^*$ by augmenting it along $P$, i.e., $\sigma^* \leftarrow \sigma^* \oplus P$.

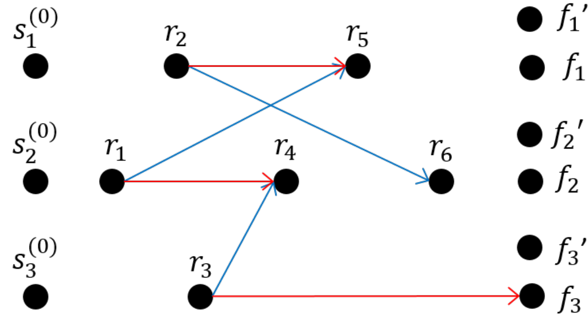


Figure 4.4: Non-trivial alternating path in $\sigma_i \oplus \sigma_{(i+1)}$. All red edges are part of solution $\sigma_i$ and all the blue edges are not part of $\sigma_i$

To compute the minimum net-cost augmenting path we construct a weighted residual graph $G_\sigma$. Consider request $r_t$, for all $t \leq i$, let $N(r_t)$ be the next request. If $r_t$ is the last request on any of the $k$-paths in $\sigma^*$, then $N(r_t) \in C^{(f)}$. Cost of the edge between the new request $r_{(i+1)}$ and $N(r_t)$ is given by: $d(r_{(i+1)}, N(r_t)) = d(r_t, r_{(i+1)}) - d(r_t, N(r_t))$. We can then run Bellman-Ford algorithm on $G_\sigma$ with $r_{(i+1)}$ as the source vertex to find minimum net-cost augmenting path $P$.

**Retrospective Solution:** We can construct an online solution $\sigma$ using this optimal offline solution $\sigma^*$. To make the online assignment we move the server at location $f$ to $r_{(i+1)}$, i.e., $\sigma \leftarrow \sigma \cup \{(f, r_{(i+1)})\}$. This online assignment is based on the optimal offline solution is the *retrospective solution*. The retrospective algorithm for the $k$-server problem has an unbounded competitive ratio.

To improve bound on this algorithm, parameter $t$ is introduced. The algorithm described in this chapter uses, for some $t > 1$, a $t$-approximate offline solution to guide the online server assignment. This $t$-approximate solution is derived from a linear program that relaxes the constraint for all the edges not in the solution by a multiplicative factor of $t$. The value of $t$ is fixed for the entire execution instance of the algorithm.

▷ **Definition 4.12.** For a parameter $t > 1$, we define $t$-net cost of any path $P$ with respect

to solution $\sigma^*$ to be:

$$\phi_t(P) = t \cdot \left( \sum_{(u,v) \in P \setminus \sigma^*} d(u,v) \right) - \sum_{(u,v) \in P \cap \sigma^*} d(u,v)$$

For example, if we consider the path shown in Figure 4.5, it has a Net-Cost of 2. The $t$-Net-Cost for the same path will be 22 if the value of $t = 2$.

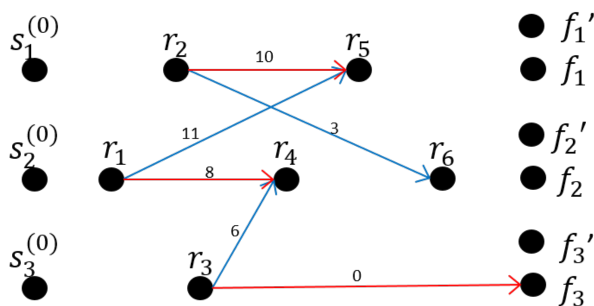


Figure 4.5: Example of Net-Cost for an alternating

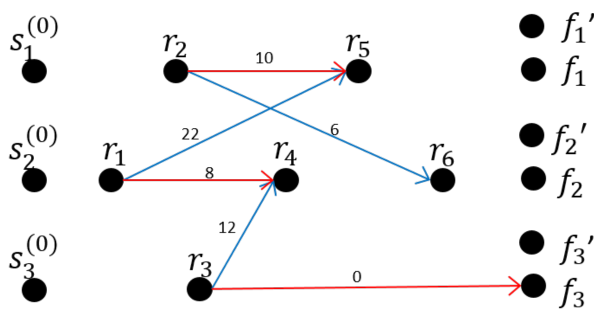


Figure 4.6: Example of $t$-Net-Cost for an alternating

Given a residual graph, when a new request arrives, the algorithm computes the minimum $t$-net cost augmenting and then augments along that path to compute the $t$-approximate offline solution.

## 4.2 Algorithm Details

This algorithm maintains two solutions $\sigma$ and $\sigma^*$; both of these solutions are empty to begin with, i.e., they are initialized to $\phi$ at the start. Solution $\sigma^*$ is $t$–approximate solution; we refer to this as the *offline solution*. Solution $\sigma$ is the *online solution*. Both online and offline solutions process all the requests seen so far in the metric space.

**Input:** Metric space $(Q, d)$, a sequence of requests $R = \{r_1, ...., r_n\}$ , initial server positions $C^{(0)} = \{s_1^{(0)}, ...., s_k^{(0)}\}$ and constant parameter $t$

**Output:** Online solution $\sigma$

1 Initialize online solution $\sigma$ and offline solution $\sigma^*$ to $\phi$;
2 Initialize final configuration $C^{(f)} = \{f_1, ...., f_k\}$;
3 **for** *Request* $r_i$ *in* $R$ **do**
4     Compute weighted residual graph $G_\sigma$;
5     Run Bellman-Ford algorithm on $G_\sigma$ and find augmenting path $\overrightarrow{P}$ with minimum $t$-net cost, with source: $r_i$ and destination $f \in C^{(f)}$;
6     Compute the offline solution $\sigma^* \leftarrow \sigma^* \oplus \overrightarrow{P}$;
7     Remove $f$ from $C^{(f)}$ and add $r_i$ to $C^{(f)}$;
8     Compute the online solution $\sigma \leftarrow \sigma \cup \{(f, r_i)\}$;
9 **end**

**Algorithm 1:** Robust Algorithm for the $k$-Server Problem Based on $t$-Net Cost

Given a new request $r_{(i+1)}$, this algorithm computes the minimum $t$-net-cost augmenting path $P$ with respect to the solution $\sigma^*$. The augmenting path starts at $r_{(i+1)}$ and ends at $f$, where $f \in C^{(f)}$. The algorithm updates offline solution $\sigma^*$ by augmenting it along $P$, i.e., $\sigma^* \leftarrow \sigma^* \oplus P$. For the online solution $\sigma$, the algorithm will move the server at location $f$ to $r_{(i+1)}$, i.e., $\sigma \leftarrow \sigma \cup \{(f, r_{(i+1)})\}$. We also remove $f$ from the final configuration $C^{(f)}$ and add $r_{(i+1)}$ to $C^{(f)}$.

To compute the minimum $t$-net-cost augmenting path we construct a weighted residual graph $G_\sigma$. Consider request $r_a$, for all $a \leq i$, let $N(r_a)$ be the next request. If $r_a$ is the last request on any of the $k$-paths in $\sigma^*$, then $N(r_a) \in C^{(f)}$. Cost of the edge between the new request $r_{(i+1)}$ and $N(r_a)$ is given by: $d(r_{(i+1)}, N(r_a)) = t \cdot d(r_a, r_{(i+1)}) - d(r_a, N(r_a))$. We can then run Bellman-Ford algorithm on $G_\sigma$ with $r_{(i+1)}$ as the source vertex and find minimum net-cost augmenting path $P$.

## 4.3 Intuition Behind the Algorithm

The Greedy Algorithm moves the nearest server to process each request, is the most natural online algorithm. In other words, it picks the configuration of $k$-servers at step $i$ that minimizes the cost of $d(C^{(i-i)}, C^{(i)})$. But this algorithm has an unbounded competitive ratio. Consider a small metric space with three vertices, two of the vertices $A$ and $B$ are quite close and the third vertex $C$ is somewhat more distance from both $A$ and $B$. Consider servers to be located at $B$ and $C$ and the requests in this metric space alternate between locations $A$ and $B$. The greedy algorithm will keep moving the server between locations $A$ and $B$, this leads to arbitrarily large cost. On the other hand, an optimal offline solution would simply have moved the server from $C$ to location $A$, resulting in finite overall cost.

Another well-known algorithm for the $k$-server problem is the Retrospective Algorithm. At each time step, it chooses to move the servers into the configuration that minimizes the cost $w_i(C^{(i)}) = \sum_{i=1}^{t+1} d(C^{(i-1)}, C^{(i)})$. This algorithm also has an unbounded competitive ratio. Consider a metric space with four points, $(A, B, C, D)$. The points are such that $A$ and $B$ are close to each other and $C$ and $D$ are close to each other, while these two clusters are slightly far away. To begin with, consider three servers to be placed at $A$,$B$, and $C$. Now consider a request sequence: $\{A, B, C, D, C, D\}$. This optimal offline algorithm will move one

server from $A$ or $B$ to $D$ and the online assignment will reflect the same. Now if the following requests were added to the sequence such that new sequence is $\{A, B, C, D, C, D, A, B, A, B\}$. The retrospective algorithm will again move one server from $C$ of $D$ to $B$. If the request keep coming according to the aforementioned sequence, online assignment made by the algorithm will keep moving one server between two clusters, leading to unbounded competitive ratio.

The Work Function Algorithm described in Section 3.2 attempts to balance between these two unbounded algorithms. At time step $i$, it chooses configuration $C^{(i)}$ that minimizes the cost function: $w_i(C^{(i)}) + d(C^{(i-1)}, C^{(i)})$. But this is a exponential time algorithm, making it difficult to run for larger inputs. Furthermore, this algorithm requires a finite metric space for execution.

The algorithm described in Section 4.2 works similar to the work function algorithm and tried to eliminate the drawback put forth by the work function algorithm. Instead of computing cost of work function, it uses $t$-approximate offline solution to guide the online assignment.

It can be observed that for larger values of t, the algorithm picks an augmenting path P of a smaller length leading to lower cost of the online solution. On the other hand, larger values of $t$ cause the offline solution to be a weaker approximation which leads to weaker bound for the online matching.

Furthermore, for $t = 1$, the offline solution computed by this algorithm is the optimal offline solution and the online solution corresponds to the retrospective solution which has an unbounded competitive ratio. Therefore, we can say that the best trade-off between these observations can be achieved at some finite value of $t > 1$.

Consider an example of 3 servers and 2 clusters of requests presented in figure 4.7. Cluster 1 is present at the top left corner in the space and the Cluster 2 is present in the bottom right corner. Two servers, $s_0$ (black) and $s_1$ (red) are placed close to cluster 1 and $s_2$ (red) is

placed to cluster to 2. Cluster 2 is generating a greater number of requests during the start of the execution, our algorithm learns the same and allocates one server to serve requests from cluster 2. Later the frequency of requests in cluster 2 slows down and cluster 1 generates requests with larger frequency, learning this fact our algorithm again reallocates resources such that cluster 1 has a greater number of servers.

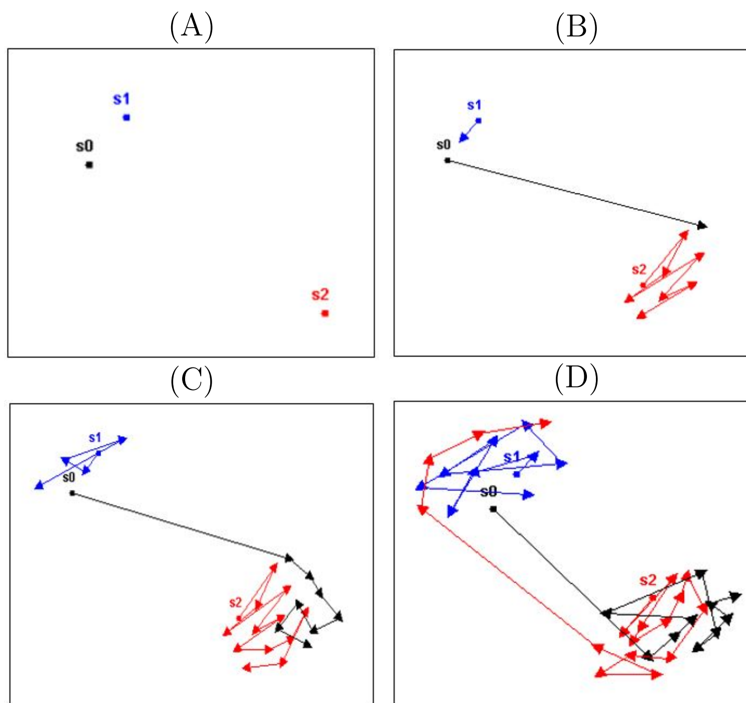


Figure 4.7: Execution Instance of the $k$-server problem with $k = 3$

The promptness of the algorithm to make the necessary adjustment when the request pattern changes can be controlled by controlling $t$. Larger the value of $t$, shorter is the augmenting path selected by the algorithm. This is because all the incoming edges (edges which are not a part of the current solution) are penalized by a factor of $t$, so the cost of any incoming edge is $t$ times the weight of the edge. So for each incoming edge in the augmenting path we are paying extra cost. So we tend to select an augmenting path containing fewer number of edges.

## 4.4 Optimizations to reduce time complexity

In this section we will discuss some of the optimizations techniques implemented to increase the speed of computation.

### 4.4.1 Bellman-Ford Optimization

Given a graph $G = (V, E)$ and source vertex $p$, the Bellman-Ford algorithm gives the shortest distance to all the vertices from $p$. The algorithm first define the distance matrix *dist* of size $\mid V \mid$ and initialize all values as $\infty$ except $dist[p] = 0$. Then we iterate for $\mid V - 1 \mid$ and for each edge $\in E$ between vertices, say $(r, s)$, if $dist[r] > dist[s] + d(r, s)$ then $dist[r] = dist[s] + d(r, s)$.

▷ **First Optimization:** The outer most iteration runs for $\mid V - 1 \mid$ times as the number of longest connect edges could be $\mid V - 1 \mid$. It assumes that during each iteration at least one of the edges will get relaxed. But turns out that the longest connected edges are significantly less than $\mid V - 1 \mid$. So we can terminate the outer loop when we no longer see any improvements in the distance matrix.

▷ **Second Optimization:** Also during each iteration, the algorithm updates one or more vertex weights and in the next iteration, it visits every single vertex to check if its weight needs an update. But it can be seen that unless a vertex weight was updated in the last iteration, the next connected vertices to it will not have any effect on weights. In this optimization, we use a boolean flag for each vertex to check if it was updated in the last iteration.

These optimizations nearly give 20 times improvement over the original algorithm. But the time complexity of the algorithm is still a function of number of requests.

### 4.4.2 Residual Graph Alteration

Before we go into details, we need to understand the process of augmentation. When we augment along any path, we force a server to stop at some request in the past. Consider a case where we were serving request $r_i$ and to serve this request in the $t$-optimum offline solution we were required to augment along some path. This augmenting path forced one of the servers to stop at an earlier request, say $r_t$. So in the $t$-optimum solution the requests that come after $r_t$ till request $r_i$ were served by only $k-1$ servers, as we forced one of the servers to stay put at $r_t$. So the probability of selecting an augmenting path that has old requests is less, as having that server in the path would require of some server to stop at that point and this would put burden on the remaining $k-1$ servers to serve all the requests that come after that particular request till the current request.

We utilize this fact and remove the older requests while constructing the residual graph. For each server, we maintain a list of a fixed number of requests that it has recently served. Any request older than that fixed number is not considered while constructing the residual graph. Before we go into details, we need to understand the process of augmentation. When we augment along any path, we force a server to stop at some request in the past. Consider a case where we were serving request $r_i$ and to serve this request in the offline solution we were required to augment along some path. This augmenting path forced one of the servers to stop at an earlier request, say $r_t$. So in the $t$-optimum solution the requests that come after $r_t$ till request $r_i$ were served by only $k-1$ servers, as we forced one of the servers to stay put at $r_t$. So the probability of selecting an augmenting path that has old requests is less, as having that server in the path would require of some server to stop at that point and this would put burden on the remaining $k-1$ servers to serve all the requests that come after that particular request till the current request.

We utilize this fact and remove the older requests while constructing the residual graph. For each server, we maintain a list of a fixed number of requests that it has recently served. Any request older than that fixed number is not considered while constructing the residual graph.

## 4.5 Experiment Details and Results

During these experiments we consider a 2-dimensional Euclidean space with servers and requests to be locations in this space. We analyze this algorithm using following synthetic datasets: Random Distribution, Gaussian Distribution, Power Law Distribution, Exponential Distribution. The locations of the requests are randomly generated from these distributions. We also consider more realistic setting described below.

The locations of the requests in a city hardly follow a single distribution. The request locations in a city can be divided in to two categories, requests coming from hot spots like airport, office center, shopping complexes while other requests often come from random locations in the city. This can be viewed as a combination of random distribution, representing the location of the request coming from random parts in the city, and multiple clusters with Gaussian distribution, representing the locations of the hot spots in and around the city. Also, according to recent studies, the movement of taxies in a city follow Power Law and Exponential Distribution, so we have also considered these distributions for our analysis.

We implemented the RK-Algorithm, allowing for computation of sequence of configurations start from the initial locations of $k$-servers for given sequence of requests. The code was written in Java 8 and all the computations were performed on an Intel 2.60 GHz $i5$ processor running windows with 16 GB memory.

We present our results for Greedy, Retrospective and RK-Algorithm in various settings. For the random distribution refer to Figure 4.8. We observer that the Greedy Algorithm performs 2-3% better than the RK-Algorithm, while the RK-Algorithm outperforms Retrospective Algorithm by 3-20%. Greedy Algorithm is known to do well for data that is chosen uniformly at random, whereas, Retrospective Algorithm is very sensitive to minor changes in input.

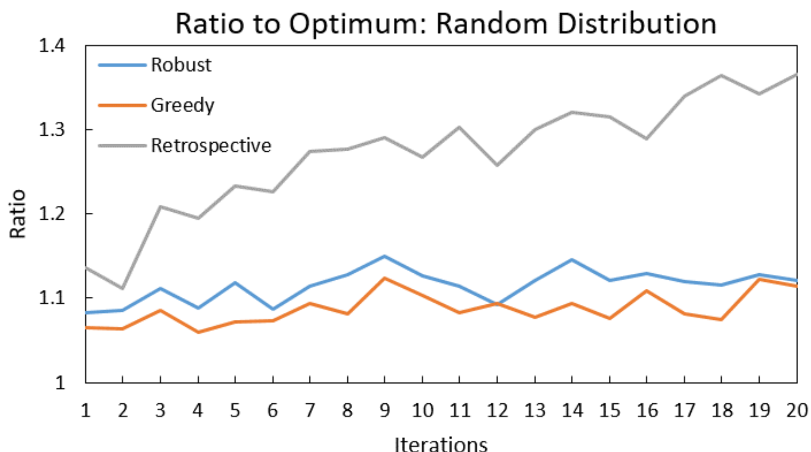


Figure 4.8: Graph showing the mean ratio of costs in serving 1000 requests: Random Distribution

For the Gaussian distribution refer to Figure 4.9. We observer that the Retrospective Algorithm does as good as the RK-Algorithm, while the RK-Algorithm outperforms Greedy Algorithm by 4-15%. In Gaussian distribution most of the requests are close to the center. Greedy Performs poorly since, one request that is far from the center may pull a server out. This server may remain out of play for several future requests leading to an overall increased cost.

For combination of both Random and Gaussian Distribution refer to Figure 4.10. During this experiment we have 10% of the total requests arriving from random locations, while the remaining requests arrive from the Gaussian centers. We observer that RK-Algorithm outperforms both Greedy and Retrospective Algorithm. The greedy algorithm pulls servers

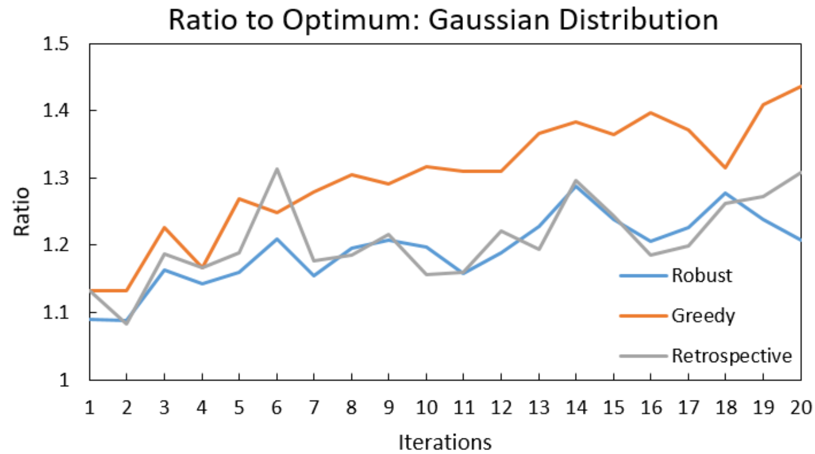


Figure 4.9: Graph showing the mean ratio of costs in serving 1000 requests: Gaussian Distribution

away from the center, while the retrospective algorithm keeps adjusting to minor changes in the input this leads to poor performance of these algorithms. RK-Algorithm balances these two extremes by making it difficult to make a retrospective choice and if the servers are too far away from the center it eventually brings them back, closer to the center.

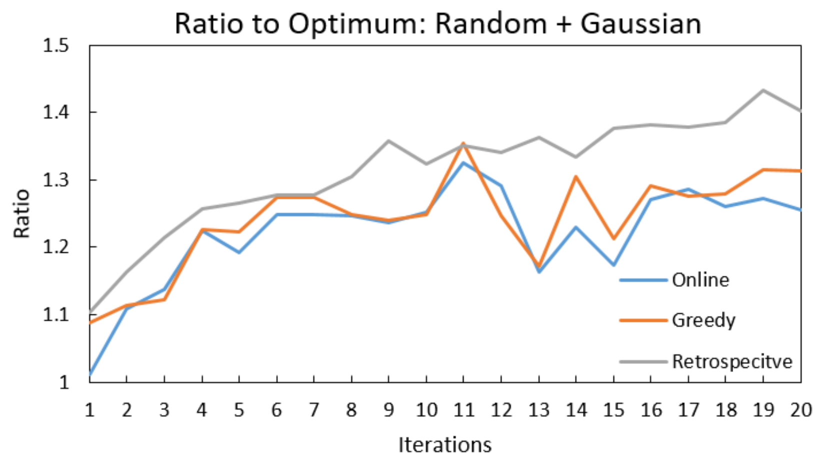


Figure 4.10: Graph showing the mean ratio of costs in serving 1000 requests: Gaussian+Random Distribution

We also consider a setting with multiple clusters. These clusters use Gaussian distribution

and may have different rate at which they generate requests. For analysis under this setting refer to Figure 4.11. We observer that RK-Algorithm outperforms both Greedy and Retrospective Algorithm. The Greedy Algorithm performs poorly since, it fails to take in to account the different rate of request generation for different clusters and so it does not move the servers between the clusters. While the retrospective algorithm keeps adjusting to minor changes in the input by moving servers from one cluster to other more than necessary.

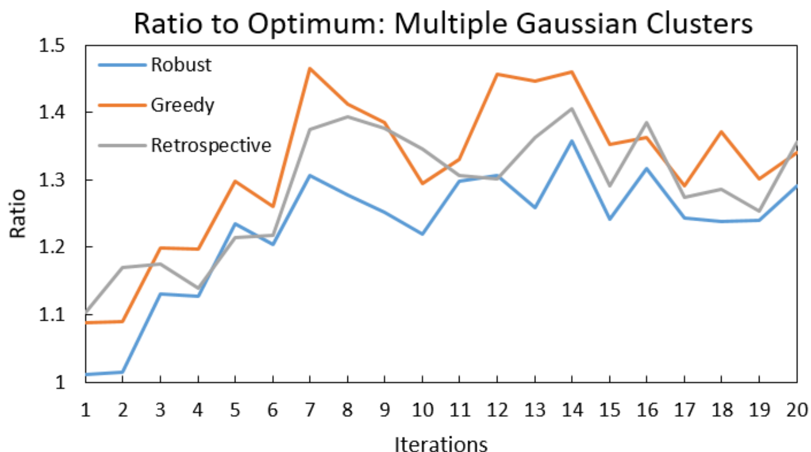


Figure 4.11: Graph showing the mean ratio of costs in serving 1000 requests: Multiple Clusters

We also observe that the Greedy algorithm does not perform well for Power Law (Figure 4.12) and Exponential Distribution (Figure 4.13). RK-Algorithm performs as good as or in some cases better than Greedy Algorithm under Power Law Distribution, while it outperforms greedy algorithm by 5-10% if we consider Exponential Distribution. Furthermore, RK-Algorithm performs better than the Retrospective Algorithm under both Power Law as well as Exponential distributions.

In Figure 4.14, we analyze the effect of changing parameter $t$ when the algorithm is given a constant input. We see that the performance of this algorithm improves with the value of $t$

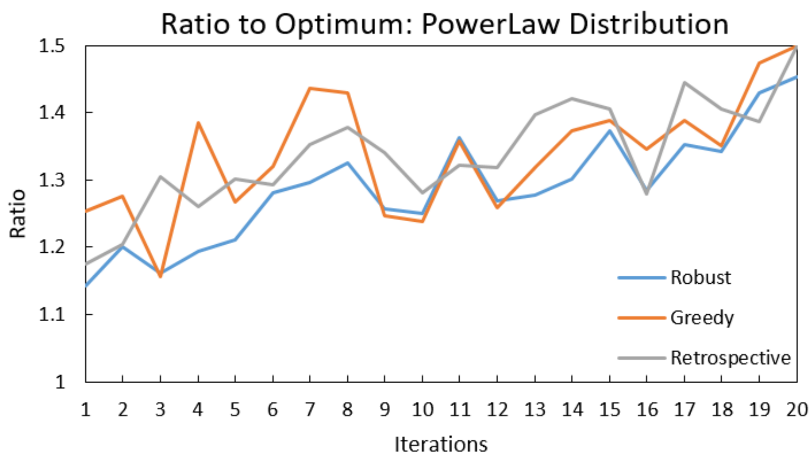


Figure 4.12: Graph showing the mean ratio of costs in serving 1000 requests: Power Law Distribution

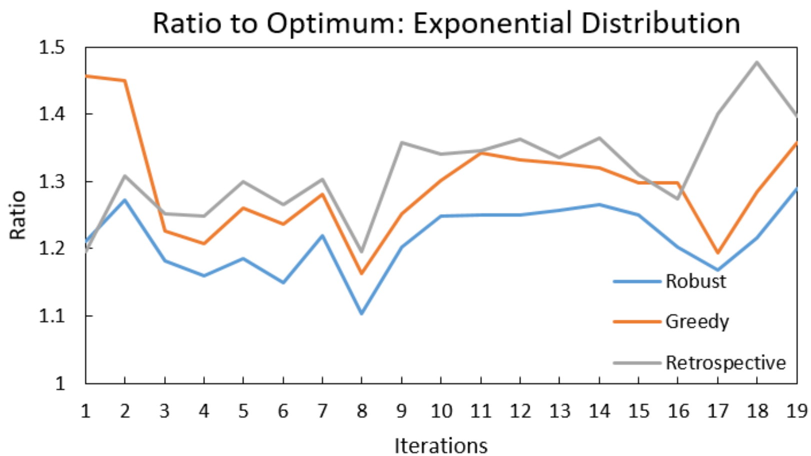


Figure 4.13: Graph showing the mean ratio of costs in serving 1000 requests: Exponential Distribution

up to a certain value after which it starts degrading. We know that for $t = 1$ RK-Algorithm produces the same result as the retrospective algorithm. But is interesting to see that as we increase the value of $t$, the cost of RK-Algorithm approaches the cost of the greedy algorithm.

We can conclude that the parameter $t$ acts as a balance between retrospective and greedy algorithm. Furthermore, we select the value of $t = 2.5$ for all the instances of the $t$-Net

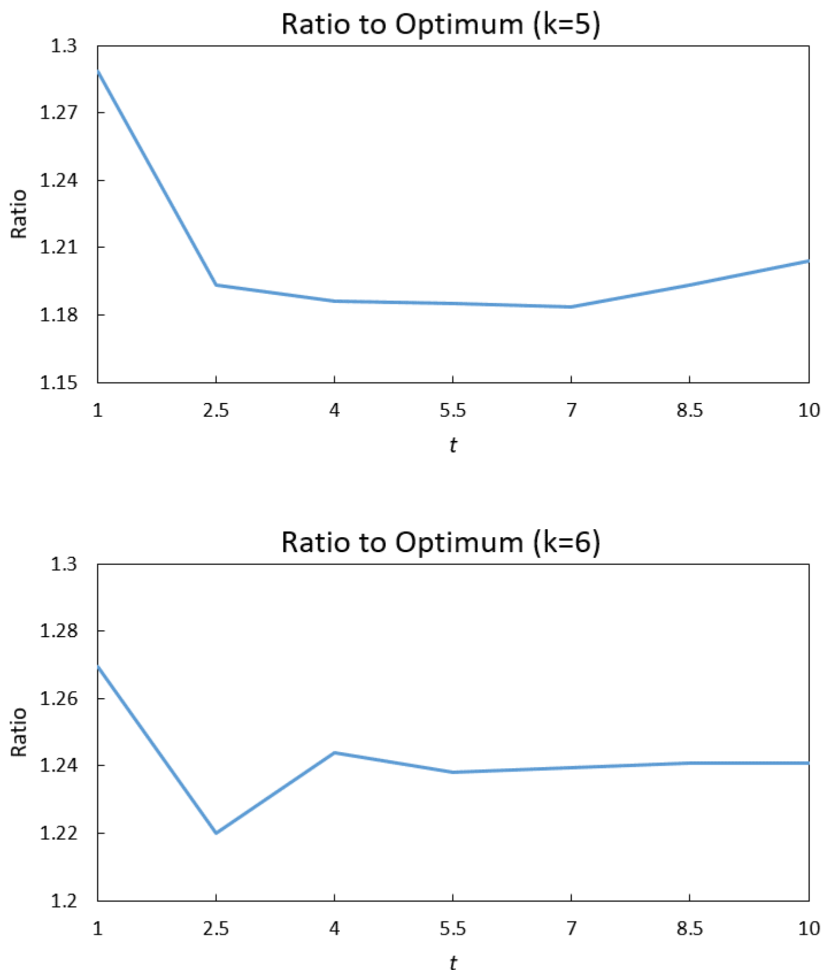


Figure 4.14: Graph showing the mean ratio of costs in serving 1000 requests

Algorithm as it empirically gives the best possible result for numerous values of $k$.

RK-Algorithm performs as good as the best-known algorithm for $k$–server problem, the Work Function Algorithm. Code for WFA was written in Python 2.7. The execution time to run a single instance of WFA increases exponentially with increase in server or request count. So we have only included results for $k = 3$ and $k = 4$ and $n = 150$ (Figure 4.15).

The process of computing the shortest $t$-net augmenting path uses Bellman-Ford algorithm, which is $O(n^3)$ algorithm. So to reduce the run-time of we suggested few modifications in

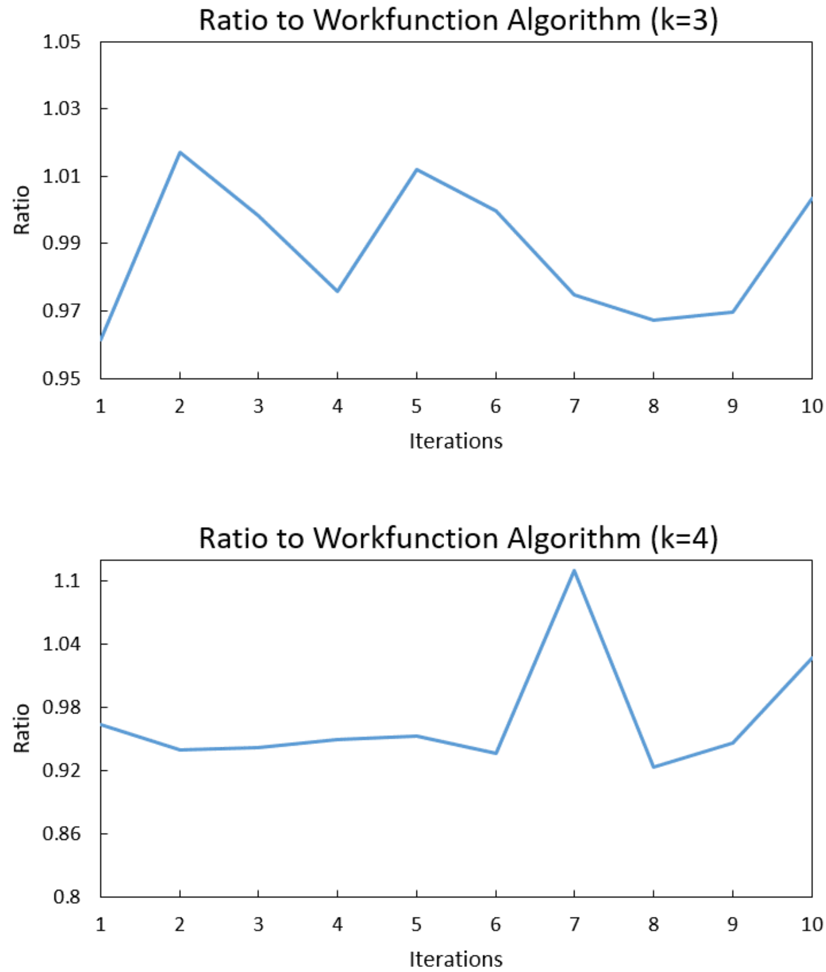


Figure 4.15: Graph showing the mean ratio of costs in serving 150 requests

4.4. We analyze the run-time improvement and effect on the performance of this algorithm after we incorporate these optimizations.

In Figures 4.16 and 4.17, we can see that there is nearly 50 times improvement in the run-time and the performance of the algorithm is not affected because of these optimizations.

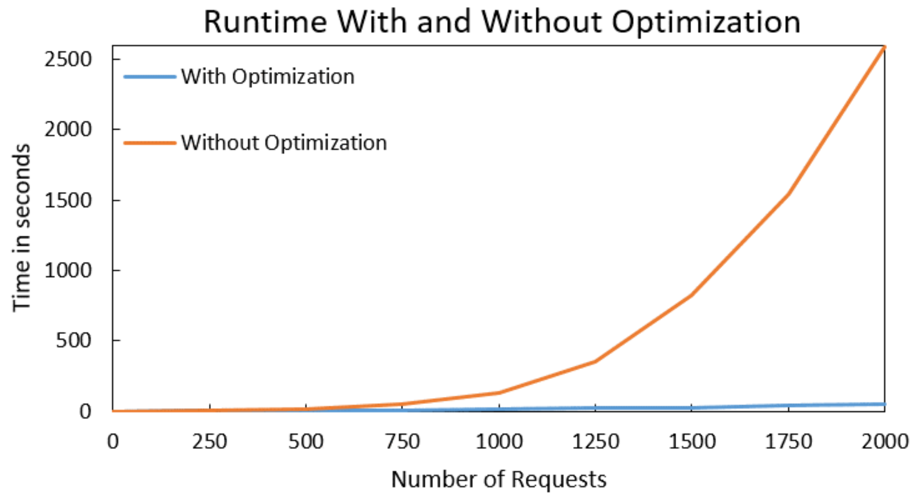


Figure 4.16: Runtime of the algorithm with and without Optimization

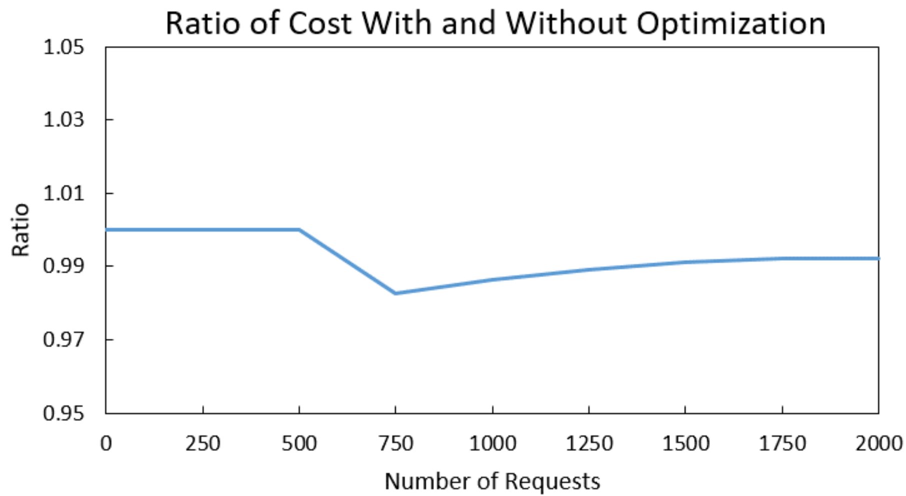


Figure 4.17: Ratio of Cost With Optimization to Cost Without Optimization

# Chapter 5

# Lookahead Algorithm

In this chapter, we give the lookahead based algorithm for the online bipartite matching problem and the $k$-server problem under stochastic setting. Furthermore, we also provide an empirical analysis of these algorithms.

## 5.1 Previous Work

Before diving into the details of the implemented algorithm it is important to understand the lookahead algorithm and related terminologies. So in this subsection, we will cover the required background for lookahead algorithm.

As mentioned in Section 2.1 there are numerous algorithms that have been studied in relation to $k$-server problem in both deterministic and randomized setting. One such algorithm is the work function algorithm that balances two unbounded algorithms greedy and retrospective. Lookahead does much of the same, instead of the retrospective cost, it uses expected minimum distance.

▷ **Lookahead Algorithm:** Given request $r$, servers $S = \{s_1, ...., s_k\}$ and constant parameter $\eta$, the lookahead algorithm outputs new server configuration $C^{(i)}$. This algorithm matches the server $s_i$ to request r with the goal to minimize the following cost function[8]:

$$C^{(i)} = d(r, s_i) + \eta E[C^{(i)}]$$

$E[C^{(i)}]$ is the expected cost to serve the future requests based on the resulting server configuration after matching server $s_i$ to server request r. The value of $\eta$ in the aforementioned equation is fixed before the computation of this algorithm begins.

This algorithm, like work function algorithm, works on finite metric spaces as the complexity of computing the expected minimum distance increases exponentially with increase in the number of points in the metric space. So to reduce the time complexity to a certain extent we have implemented a one-lookahead and two-lookahead variations of this algorithm. The one-lookahead algorithm minimizes expected cost just for the next round of requests, i.e., the $(i+1)^{th}$ request.

## 5.2 Algorithm Details

Lookahead algorithm optimizes the performance based on the distribution of the metric space, while the RK-Algorithm seen in Chapter 4, optimizes the performance based on the requests that we have seen so far. In other words, one algorithm looks at the future requests while the other looks at the past requests. The algorithm we suggest in this chapter tries to harness the positives of both of these algorithms.

Similar to the algorithm described in Chapter 4, we maintain two solutions $\sigma$ and $\sigma^*$; both of these solutions are empty to begin with. Solution $\sigma^*$ is $t$–approximate offline solution.

Given a new request $r_{i+1}$, our algorithm computes the minimum $t$-net-cost augmenting paths $P = \{p_1, ...., p_k\}$ with respect to the solution $\sigma^*$ for each of the $k$ server. This algorithm selects the augmenting path $p_i$ with the goal to minimize the following cost function:

$$\phi_t(p_i) + \eta E[C^{(i)}]$$

where $E[C^{(i)}]$ is the expected cost to serve the next request based on the resulting server configuration after augmenting along the path $p_i \in P$. $\eta$ in the aforementioned equation is a constant parameter and is fixed before the computation of this algorithm begins. The algorithm updates offline solution $\sigma^*$ by augmenting it along $p_i$, i.e., $\sigma^* \leftarrow \sigma^* \oplus p_i$. For the on-line solution $\sigma$, the algorithm will match the server $s$ to $r_{i+1}$, i.e., $\sigma \leftarrow \sigma \cup \{(s, r_{i+1})\}$.

To reduce the runtime of this algorithm we use the 1-lookahead algorithm described in 5.1. The metric space in this problem is continuous but for computing the expected minimum distance we assume finite metric space based on the distribution of requests.

## 5.3 Experiment Details and Results

### 5.3.1 Lookahead Algorithm for Bipartite Matching Problem

The code was written in Python 2.7.

We calculated the ratio of total costs of lookahead to greedy, presenting the mean across repetitions. Note that at $\eta = 0$ the lookahead is equivalent to greedy, and hence should always be greater than 1.

Though no clear pattern for values of emerge, the lookahead algorithm appears to outperform greedy on average by $3-8\%$ for a range of values of $\eta$ (Figure 5.1). This offers some empirical evidence that greedy is not optimal.

It is worth noting that this is merely a presentation of mean results. There were instances where greedy outperformed the lookahead.

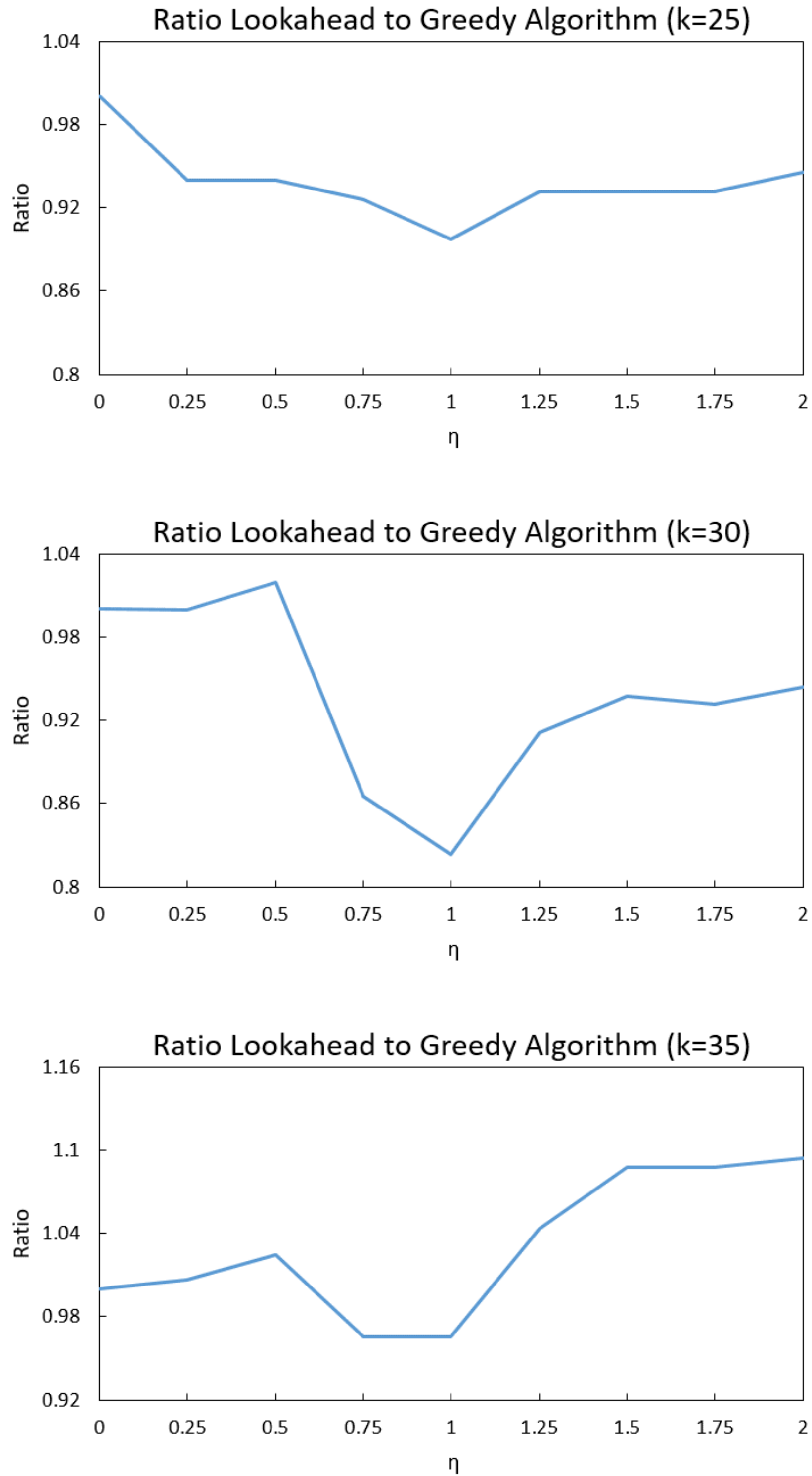


Figure 5.1: Graphs showing the mean ratio of costs, averaged across 10 repetitions, for various different values of $\eta$

### 5.3.2 Lookahead Algorithm for k-Server Problem

The code for lookahead algorithm was written in Java and for testing we use the same GUI framework that we used for the RK-Algorithm.

We calculated the ratio of total costs of this new algorithm to the RK-Algorithm. Note that at $\eta = 0$ this new algorithm is equivalent to the RK-Algorithm, and hence $\eta$ should always be greater than 1.

Our new lookahead algorithm performs better than the RK-Algorithm on average by $3-12\%$ for a range of values of $\eta$ (Figure 5.2). Lookahead step allows the algorithm to get a sense of future requests. Unlike RK-Algorithm, which only relies on requests seen so far while making the assignment for the new request, lookahead algorithm manages the server assignment more effectively by using future knowledge along with previous data.

Note that the decisions based an inaccurate stochastic model would adversely affect the performance of this algorithm.

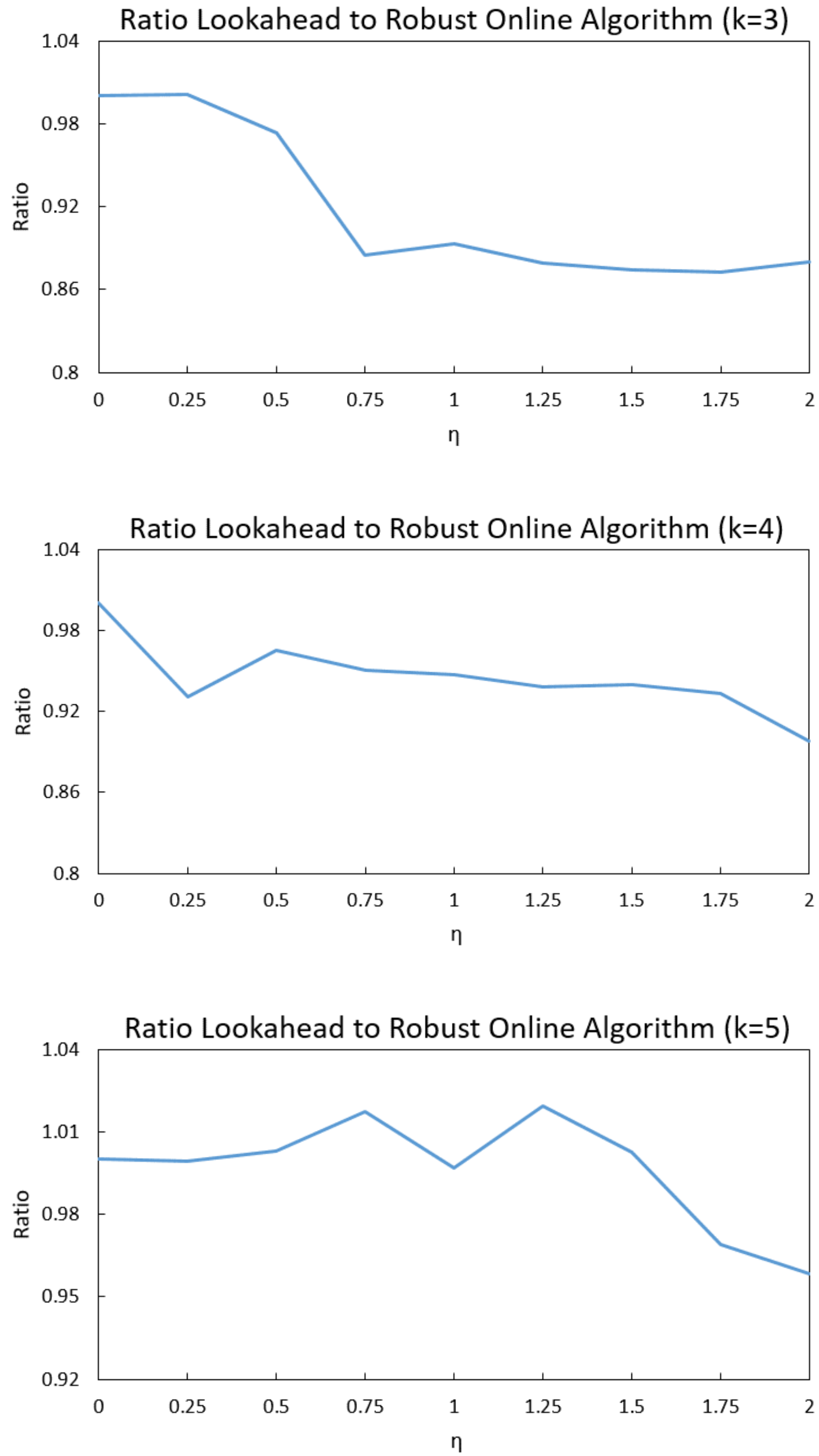


Figure 5.2: Graphs showing the mean ratio of costs, averaged across 10 repetitions, for various different values of $\eta$

# Chapter 6

# Graphical User Interface

In this chapter, we present an overview of the graphical user interface (GUI) developed for simulations of the RK-Algorithm. We will also look into various features of this GUI, like the *fast mode* and the *lookahead mode*.

## 6.1 Background

The GUI was implemented using the Swing framework. Swing is a GUI widget toolkit for Java applications and its components are entirely written in Java, therefore they are platform-independent.

### 6.1.1 MVC architecture

Swing uses the model-view-controller architecture (MVC) as the fundamental design behind each of its components. MVC divides the GUI components into three elements and each of these elements are important to how this component behaves [7].

▹ **Model:** Model encompasses the state data for each component. For example, the model of a scrollbar component might contain information about the current position of the marker, its maximum, and minimum values. Model data always exist and is independent of the component's visual representation.

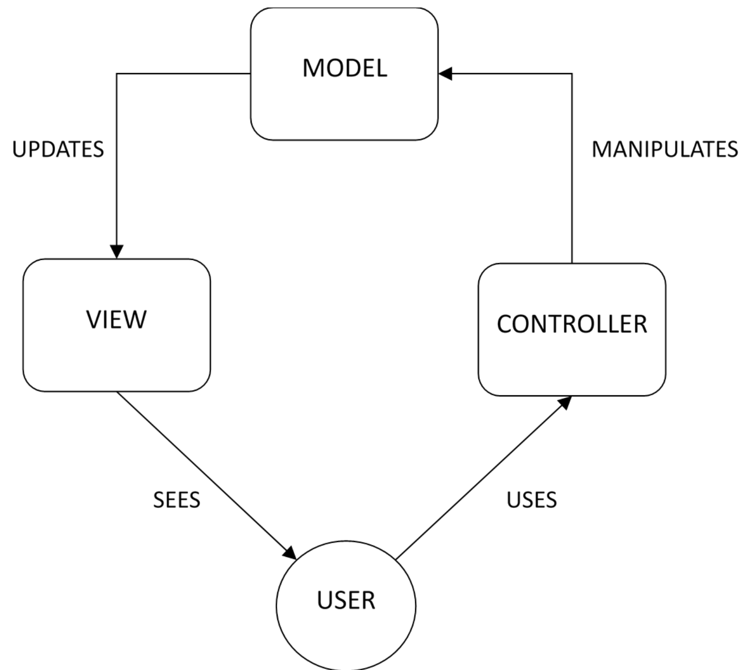


Figure 6.1: Three Elements: MODEL, VIEW and CONTROLLER

▷ **View:** A View defines how we see the component on the screen. A good example of how views can differ may be the close box in an application window that we open. The close box appears on the right side for few platforms like Linux, while for other platforms like Windows the close box appears on the left side.

▷ **Controller:** A Controller dictates how the component interacts with events. Events may come in the form of mouse clicks, keystroke on a keyboard.

Swing uses a simplified variant of the MVC design. In swing the view and the controller objects are combined into a single element the draws the component to the screen and handles GUI events. Note that the separation of model from the other two elements is beneficial. For example, we can tie multiple views to a single model.

## 6.2 GUI Components

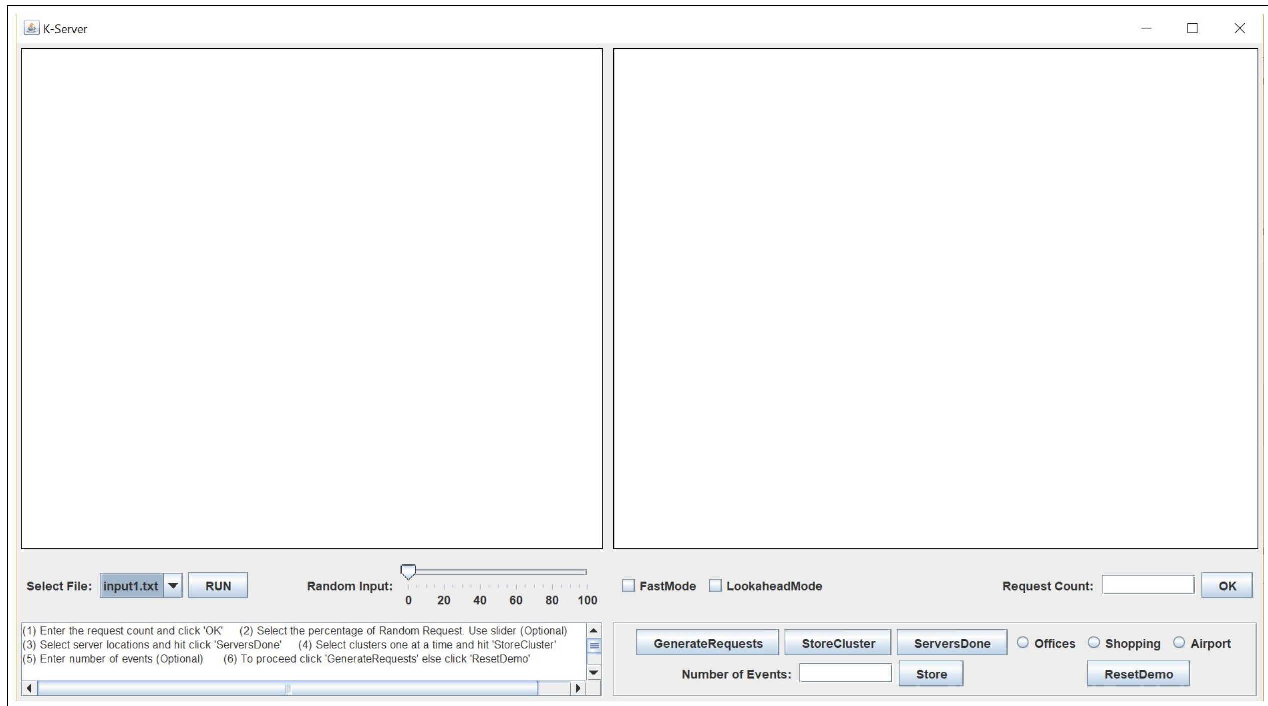


Figure 6.2: Graphical User Interface

### 6.2.1 Outputs

▷ **Online Solution and Offline Solution:** The GUI consists of views for both online and offline solution. Also the view for online solution acts as a controller to receive inputs from the user via mouse commands. These views are updated in real time so it is possible to see how the allocation of servers is made in accordance to the changing input pattern.

▷ **Results:** The results and instructions to be followed to execute the algorithm are displayed in the bottom left view in the GUI. The cost of the online and greedy solution is updated in this view after every 100 requests. At the end of the execution Retrospective and Optimum cost is also displayed in this view.

### 6.2.2 Inputs

▷ **Server Selection:** View for online solution also acts a metric space in which user can position servers. Each server is represented using a different color, this helps to visualize the movement of the servers. Once the server allocation is completed the user can press an action switch indicated by *ServerDone* to mark the end of server selection.

▷ **Request Count:** A user can input the number of requests for the current instance of the $k$-server problem. Depending on the clusters selected and the percentage of random requests the requests are generated before the algorithm begins computation. These requests are feed to the algorithm in an online manner.

The user also has the freedom to select the request location once the server selection is completed. The user can use the view for the online solution to select the request location, and these requests are processed as soon as the location is selected by the user. Once the request is processed the online and offline solution views are updated to show which server has moved to serve the request.

Once all the parameters relevant to the experiment are set user can ask the inbuilt algorithm auto-generate requests by pressing an action switch indicated by *GenerateRequests*.

▷ **Cluster Selection:** This user interface supports four types of Gaussian clusters elaborated in Section 6.3. User can use the online solution view to draw rectangular clusters. Center of the rectangular cluster is used as the mean value and the height and the width of the rectangular region is used as the variances for the Gaussian clusters. User can confirm the cluster selection by pressing an action switch indicated by *StoreCluster*.

▷ **Random Input:** User has an option to set the percentage of the random requests. These requests are independent of all the clusters and can pop up at any location in the metric

space. The default value of random requests is set to zero percent.

▷ **Reset:** Reset controller allowed the user to reset all the execution parameters, it also clears the online and offline solution views. A user can run multiple instances of the problem one after the other by using this reset functionality.

▷ **Fast Mode:** By selecting this controller, user can run the optimized version of the RK-Algorithm as describe in Section 4.4.

▷ **Lookahead Mode:** By selecting this controller, user can run the lookahead version of the RK-Algorithm as describe in Section 5.2. We can run this algorithm along with the fast mode.

## 6.3 Types of Clusters

We consider the execution instance of the k-server problem to be divided into 10 slots of one hour each. These 10 hours represent the time between 8 am and 6 pm. Also, we consider each server to be a taxi. The goal of these cabs is to serve the requests that can occur at any location in the metric space at any point during these 10 hours.

In any city, the requests for cab's have a specific pattern. There are few hot spots in the city which generate many more requests than other. Furthermore, the rate at which requests are generated depends on the time of the day. To represent this city environment, we divide these hot spots into four categories:

▷ **Airports:** The intensity of request generation in this cluster is constant throughout the day.

▷ **Office Centers:** The intensity of request generation in this cluster has two peaks one in

| Distribution of Requests | | | |
|---|---|---|---|
| Time | Office Centers | Airports | Movie Theaters |
| 8 am to 9 am | 50 % | 10 % | 0 % |
| 9 am to 10 am | 0 % | 10 % | 20 % |
| 10 am to 11 am | 0 % | 10 % | 0 % |
| 11 am to 12 pm | 0 % | 10 % | 20 % |
| 12 pm to 1 pm | 0 % | 10 % | 0 % |
| 1 pm to 2 pm | 0 % | 10 % | 20 % |
| 2 pm to 3 pm | 0 % | 10 % | 0 % |
| 3 pm to 4 pm | 0 % | 10 % | 20 % |
| 4 pm to 5 pm | 50 % | 10 % | 0 % |
| 5 pm to 6 pm | 0 % | 10 % | 20 % |

Table 6.1: Request distribution in Different Clusters

the morning and the other one in the evening. 50 percent of the requests are generated in between 8 am and 10 am while the remaining requests are generated between 4 pm and 6 pm.

▷ **Movie Theaters:** The intensity of request generation in this cluster has five peaks with two hours between each peak.

▷ **Unknown Events:** The intensity of request generation in this cluster has a single peak. But the time and the location of these clusters are not known to the algorithm. The time at which this event occurs is selected randomly and the algorithm has to adapt to the service these requests which are not known before.

All these clusters generate requests according to the Gaussian distribution. The mean and the variance of the distribution is defined by the user, as mentioned in Section 6.2. A user can have multiple clusters of the same type and location of these clusters can be chosen by the user.

# Chapter 7

# Conclusion

We considered different algorithms for the k-server and online bipartite matching problems. First, we presented the RK-Algorithm for the $k$-server problem and discussed various optimization techniques to significantly speed-up the implementation of this algorithm. We then compared the performance of this RK-Algorithm with the greedy and retrospective algorithms. Furthermore, we empirically show that this algorithm performs as good as WFA for our test cases.

We then extended the RK-Algorithm for stochastic setting by using a lookahead strategy. A similar algorithm was also designed for online bipartite matching problem. We also provide an experimental analysis showing that this algorithm performs better than the RK-Algorithm with out lookahead. Finally, we presented a graphical user interface that was used to test these algorithms and also provide a chance for the users to interact with the testing framework.

These algorithms provide many different avenues of future research. Of particular importance, we hope to address the following:

1. Provide theoretical analysis to determine the competitive ratio and provide bounds for these algorithms.

2. Can we implement the RK-Algorithm using the Primal Dual Method?

# Bibliography

[1] Nikhil Bansal, Niv Buchbinder, Anupam Gupta, and Joseph (Seffi) Naor. A randomized o(log2 k)-competitive algorithm for metric bipartite matching. *Algorithmica*, 68(2): 390–403, Feb 2014. ISSN 1432-0541. doi: 10.1007/s00453-012-9676-9. URL https://doi.org/10.1007/s00453-012-9676-9.

[2] Nikhil Bansal, Niv Buchbinder, Aleksander Madry, and Joseph (Seffi) Naor. A polylogarithmic-competitive algorithm for the k-server problem. *J. ACM*, 62(5):40:1–40:49, November 2015. ISSN 0004-5411. doi: 10.1145/2783434. URL http://doi.acm.org/10.1145/2783434.

[3] Yair Bartal and Eddie Grove. The harmonic k-server algorithm is competitive. *J. ACM*, 47(1):1–15, January 2000. ISSN 0004-5411. doi: 10.1145/331605.331606. URL http://doi.acm.org/10.1145/331605.331606.

[4] Don Coppersmith, Peter Doyle, Prabhakar Raghavan, and Marc Snir. Random walks on weighted graphs and applications to on-line algorithms. *J. ACM*, 40(3):421–453, July 1993. ISSN 0004-5411. doi: 10.1145/174130.174131. URL http://doi.acm.org/10.1145/174130.174131.

[5] Sina Dehghani, Soheil Ehsani, MohammadTaghi Hajiaghayi, Vahid Liaghat, and Saeed Seddighin. Stochastic k-Server: How Should Uber Workl. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming (ICALP 2017)*, volume 80 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 126:1–126:14, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-

041-5. doi: 10.4230/LIPIcs.ICALP.2017.126. URL http://drops.dagstuhl.de/opus/volltexte/2017/7480.

[6] D. Du and H. Park. On competitive group testing. *SIAM Journal on Computing*, 23(5): 1019–1025, 1994. doi: 10.1137/S0097539793246690. URL https://doi.org/10.1137/S0097539793246690.

[7] Robert Eckstein, Marc Loy, and Dave Wood. *Java Swing*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1998. ISBN 1-56592-455-X.

[8] Alexander D. Friedman. Various approaches to the stochastic k-server and stacker-crane problems. Master's thesis, Virginia Polytechnic Institute and State University, May 2017.

[9] Bala Kalyanasundaram and Kirk Pruhs. Online weighted matching. *J. Algorithms*, 14 (3):478–488, May 1993. ISSN 0196-6774. doi: 10.1006/jagm.1993.1026. URL http://dx.doi.org/10.1006/jagm.1993.1026.

[10] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator. Competitive snoopy caching. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 244–254, Oct 1986. doi: 10.1109/SFCS.1986.14.

[11] Anna R. Karlin, Mark S. Manasse, Larry Rudolph, and Daniel D. Sleator. Competitive snoopy caching. *Algorithmica*, 3(1):79–119, Nov 1988. ISSN 1432-0541. doi: 10.1007/BF01762111. URL https://doi.org/10.1007/BF01762111.

[12] Samir Khuller, Stephen G. Mitchell, and Vijay V. Vazirani. On-line algorithms for weighted bipartite matching and stable marriages. In Javier Leach Albert, Burkhard Monien, and Mario Rodríguez Artalejo, editors, *Automata, Languages and Program-*

*ming*, pages 728–738, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg. ISBN 978-3-540-47516-3.

[13] E. Koutsoupias. Weak adversaries for the k-server problem. In *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*, pages 444–449, 1999. doi: 10.1109/SFFCS.1999.814616.

[14] Elias Koutsoupias. The k-server problem. *Comput. Sci. Rev.*, 3(2):105–118, May 2009. ISSN 1574-0137. doi: 10.1016/j.cosrev.2009.04.002. URL http://dx.doi.org/10.1016/j.cosrev.2009.04.002.

[15] Elias Koutsoupias and Akash Nanavati. The online matching problem on a line. In Roberto Solis-Oba and Klaus Jansen, editors, *Approximation and Online Algorithms*, pages 179–191, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-24592-6.

[16] Elias Koutsoupias and Christos Papadimitriou. On the k-server conjecture. *Journal of the ACM*, 42:507–511, 1995.

[17] Mark S. Manasse, Lyle A. McGeoch, and Daniel D. Sleator. Competitive algorithms for server problems. *J. Algorithms*, 11(2):208–230, May 1990. ISSN 0196-6774. doi: 10.1016/0196-6774(90)90003-W. URL http://dx.doi.org/10.1016/0196-6774(90)90003-W.

[18] B.S. Manoj and Alexandra Hubenko Baker. Communication challenges in emergency response. *Commun. ACM*, 50(3):51–53, March 2007. ISSN 0001-0782. doi: 10.1145/1226736.1226765. URL http://doi.acm.org/10.1145/1226736.1226765.

[19] K. Nayyar and S. Raghvendra. An input sensitive online algorithm for the metric

bipartite matching problem. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 505–515, Oct 2017. doi: 10.1109/FOCS.2017.53.

[20] Sharath Raghvendra. A Robust and Optimal Online Algorithm for Minimum Metric Bipartite Matching. In Klaus Jansen, Claire Mathieu, José D. P. Rolim, and Chris Umans, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2016)*, volume 60 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:16, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-018-7. doi: 10.4230/LIPIcs.APPROX-RANDOM.2016.18. URL http://drops.dagstuhl.de/opus/volltexte/2016/6641.

[21] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, February 1985. ISSN 0001-0782. doi: 10.1145/2786.2793. URL http://doi.acm.org/10.1145/2786.2793.

[22] Kyle Treleaven, Marco Pavone, and Emilio Frazzoli. Asymptotically optimal algorithms for pickup and delivery problems with application to large-scale transportation systems. *CoRR*, abs/1202.1327, 2012. URL http://arxiv.org/abs/1202.1327.