M. Sc. Engineering Thesis

Online Algorithms for Facility Assignment Problem

By
Abu Reyan Ahmed
Student no.: 0413052001

Submitted to
Department of Computer Science and Engineering
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science & Engineering

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology(BUET)
Dhaka-1000, Bangladesh
July, 2016

The thesis titled **"Online Algorithms for Facility Assignment Problem"**, submitted by Abu Reyan Ahmed, Roll No. 0413052001, Session April 2013, to the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, has been accepted as satisfactory in partial fulfillment of the requirements for the degree of Masters of Science in Computer Science and Engineering and approved as to its style and contents. Examination held on 16th July 2016.

Board of Examiners

1. _____

Dr. Md. Saidur Rahman — Chairman (Supervisor)
Professor
Department of CSE, BUET, Dhaka.

2. _____

Head — Member (Ex-officio)
Department of CSE, BUET, Dhaka.

3. _____

Dr. M. Kaykobad — Member
Professor
Department of CSE, BUET, Dhaka.

4. _____

Dr. Tanzima Hashem — Member
Associate Professor
Department of CSE, BUET, Dhaka.

5. _____

Dr. Muhammad Mahbub Alam — Member (External)
Professor
Department of Computer Science and Engineering
Islamic University of Technology
Board Bazar, Gazipur-1704
Dhaka, Bangladesh.

Candidate's Declaration

This is to certify that the work presented in this thesis entitled "**Online Algorithms for Facility Assignment Problem**" is the outcome of the investigation carried out by me under the supervision of Professor Dr. Md. Saidur Rahman in the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology (BUET), Dhaka. It is also declared that neither this thesis nor any part thereof has been submitted or is being currently submitted anywhere else for the award of any degree or diploma.

Abu Reyan Ahmed

Abu Reyan Ahmed
Candidate

Contents

List of Figures

Acknowledgments

I would like to thank my supervisor Professor Dr. Md. Saidur Rahman for introducing me to the field of graph theory, graph drawing and online problems. I have learned from him how to write, speak and present well. I thank him for his patience in reviewing my so many inferior drafts, for correcting my language, suggesting new ways of thinking and encouraging me to continue my work.

I would like to thank all the members of the examination board, Prof. Dr. M. Sohel Rahman, Prof. Dr. M. Kaykobad, Dr. Tanzima Hashem and Prof. Dr. Muhammad Mahbub Alam, for their valuable comments.

I would also like to thank Mrs. Shaheena Sultana, Mrs. Nazmun Nessa Moon, Mr. Md. Manzurul Hasan, Mr. Mohammad Al-Mahmud, Mr. Md. Mizanur Rahman and all the members of my research group for their valuable suggestions and continual encouragements.

My special thanks goes to Dr. Md. Iqbal Hossain for his helps. My parents, family members and friends also supported me to the best of their ability. My heart-felt gratitude goes to them.

Abstract

Consider an online facility assignment problem where a set of facilities F of equal capacity l is situated on a metric space and customers arrive one by one in an online manner on that space. We assign a customer c_i to a facility f_j before a new customer c_{i+1} arrives. The cost of this assignment is the distance between c_i and f_j. The objective of this problem is to minimize the sum of all assignment costs. In this thesis we first consider F is situated on a straight line having equal distance between adjacent facilities and customers appear anywhere on the same straight line. We propose Algorithm Greedy which is $4|F|$-competitive. After introducing randomization in Algorithm Greedy, we show that it is $\frac{9}{2}$-competitive for a class of input sequences. We provide another method Algorithm Optimal-Fill which is $|F|$-competitive. We also study another greedy method named Algorithm Capacity-Sensitive-Greedy. Consider now instead of a straight line, F is situated on the vertices of a connected unweighted graph G and customers arrive one by one having positions on the vertices of G. We show Algorithm Greedy is $2|E(G)|$-competitive and Algorithm Optimal-Fill is $|E(G)||F|/r$-competitive where r is the radius of G. We introduce service time parameter t in our modeling and show that no

deterministic algorithm is competitive when $t = 2$. We also provide a simulation of the algorithms and define a new parameter named experimental competitive ratio. We analyse the simulation data in terms of experimental competitive ratio.

Chapter 1

Introduction

There are many general classes of problems that arise in theoretical computer science. For instance: given a sequence of integers, find the largest one; given a set, list all its subsets; given a set of integers, put them in increasing order; given a network, find the shortest path between two vertices. When presented with such a problem, the first thing to do is to construct a model that translates the problem into a mathematical context. A method is needed that will solve the general problem using the model. Ideally, what is required is a procedure that follows a sequence of steps that leads to the desired answer. Such a sequence of steps is called an *algorithm*. The term *algorithm* is a corruption of the name *al-Khowarizmi*, a mathematician of the ninth century, whose book on Hindu numerals is the basis of modern decimal notation [29]. Originally, the word algorism was used for the rules for performing arithmetic using decimal notation. *Algorism* evolved into the word *algorithm* by the eighteenth century. With the growing interest in computing machines, the concept of an algorithm was given a more general meaning, to

include all definite procedures for solving problems, not just the procedures for performing arithmetic.

Consider that we have a list of elements and we have to sort the elements in ascending order. The bubble sort is one of the simplest sorting algorithms, but not one of the most efficient. It puts a list into increasing order by successively comparing adjacent elements, interchanging them if they are in the wrong order. To carry out the bubble sort, we perform the basic operation, that is, interchanging a larger element with a smaller one following it, starting at the beginning of the list, for a full pass. We iterate this procedure until the sort is complete.

In order to get correct output from the bubble sort algorithm it is required to provide the whole input data from the beginning. If instead of providing the whole input data at the beginning, one new number is given in every step of the algorithm, the output may become incorrect. Algorithms having such property are called offline algorithms. An *online algorithm* takes decisions based of past events without secure information about the future. It can provide a correct output without knowing the whole input ahead of time. Consider a sorting algorithm which is online in nature. At the beginning of the iteration, the whole input is not given. Instead, the numbers appear in an online manner one by one and the list of current numbers must be sorted before a before a new number appears.

The insertion sort is an online sorting algorithm. To sort a list with n elements, the insertion sort begins with the second element. The insertion

sort compares this second element with the first element and inserts it before the first element if it does not exceed the first element and after the first element if it exceeds the first element. In the jth step of the insertion sort, the jth element of the list is inserted into the correct position in the list of the previously sorted $j-1$ elements. To insert the jth element in the list, a linear search technique is used; the jth element is successively compared with the already sorted $j-1$ elements at the start of the list until the first element that is not less than this element is found or until it has been compared with all $j-1$ elements; the jth element is inserted in the correct position so that the first j elements are sorted. The algorithm continues until the last element is placed in the correct position relative to the already sorted list of the first $n-1$ elements.

1.1 Applications of Online Algorithms

Online algorithms are a natural topic of interest in many disciplines such as computer science, economics, and operations research [7]. Many computational problems are intrinsically online in that they require immediate decisions to be made in real time. Paging in a virtual memory system is perhaps the most studied of such computational problems. Consider the following two-level virtual memory system. Each level can store a number of fixed-size memory units (or "slots") called pages. The first level called the slow memory, stores a fixed set $P = \{p_1, p_2, \ldots, p_N\}$ of N pages. The second level, called the fast memory, can store any k-subset of P where $k < N$.

Given a request for a page p_i, the system must make page p_i available in the fast memory. If p_i is already in the fast memory (called a hit), the system need not do anything. Otherwise (on a miss), the system incurs one page fault and must copy the page p_i from the slow memory to one of the locations in the fast memory. In doing so, the system is faced with the problem of which page to evict from the fast memory to make space for p_i. In order to minimize the number of page faults, the choice of which page to evict must be made wisely. We naturally view paging as an online problem, which means that the decision of how to service the next request for a page can depend only on previous requests.

In the area of parallel processing, processes are assigned and periodically reassigned to processors or machines in order to balance the load over all processors. We can consider this scenario as an online problem. There is a set of (perhaps different) machines and a sequence of jobs is arriving where each job is specified by its processing cost, called the *load*. Each job must be either refused or assigned to one of the machines upon arrival. If the machines are not identical, then each job may have different loads depending on the machine to which the job is assigned. This is called the *load balancing* problem. The *bin packing* problem concerns a different machine assignment optimization. Assume that we have an unbounded number of identical machines (called bins), each having a bounded identical capacity (e.g., without loss of generality, assume that this bound is 1). Given a sequence of requests $r_1, r_2, \ldots$ with each request $r_i \leq 1$ representing a load or size that must be

(permanently) assigned to some bin, the goal in the bin packing problem is to minimize the number of bins needed to assign all the requests without exceeding the capacity of any bin.

Routing in communications networks is another obvious application. A call arrives with a specified bandwidth requirement, and this call must be routed from its origin to its destination in a network where each of the links (i.e., edges) has limited capacity. The call is routed on a dedicated path in the network, thereby consuming the required bandwidth on each edge of the path (i.e., virtually obtaining a dedicated path in the network). For congestion minimization, the goal is to minimize the load on any edge relative to the capacity.

1.2 Facility Assignment Problem

In this thesis, we study an online facility assignment problem where a set of facilities $F = \{f_1, f_2, \ldots, f_{|F|}\}$ of equal capacity l is situated on a metric space and customers arrive one by one in an online manner on that space. We assign a customer c_i to a facility f_j before a new customer c_{i+1} arrives. The cost of this assignment is the distance between c_i and f_j. The objective of this problem is to minimize the sum of all assignment costs. We first consider F is situated on a straight line having equal distance between adjacent facilities and customers appear anywhere on the same straight line. We next consider instead of a straight line, F is situated on the vertices of a connected unweighted graph G and customers arrive one by one having

positions on the vertices of G. We propose some online algorithms for this problem and analyse their performance.

1.3 Motivation

In this section we provide applications of online facility assignment in modeling some real world problems.

1.3.1 Online Food Delivery

Our first example is an application in algorithmic decision making in the field of finance. Consider a restaurants company has some restaurants located on an area. The customers of that company are also located in the same area. There are many competitors of that restaurants company from which the customers enjoy foods too. The company is planning to increase their profit by increasing their sells to customers. In order to increase the number of sells they have decided to provide home delivery service. Now customers can order foods from their home. The delivery cost is proportional to the distance between the location of a customer and the restaurant from which the food will be delivered. We can model the area with a graph such that the restaurants of that company are located on the vertices of that graph. We assume that the restaurants have equal capacity. Customers appear in an online manner on the vertices of the graph. We assign a customer to a restaurant before a new customer appears. The restaurant will deliver the food to the customer. The delivery cost is equal to the distance between the

location of the customer and the restaurant. The objective is to minimize the total delivery cost. This modeling is exactly similar to the online facility assignment problem. Hence we can use the algorithms of online facility assignment to solve this problem.

1.3.2 Parallel Computing

We now provide another application of the facility assignment problem in parallel computing. The role of concurrency in accelerating computing elements has been recognized for several decades. However, their role in providing multiplicity of datapaths, increased access to storage elements (both memory and disk), scalable performance, and lower costs is reflected in the wide variety of applications of parallel computing. Desktop machines, engineering workstations, and computer servers with two, four, or even eight processors connected together are becoming common platforms for design applications. Large scale applications in science and engineering rely on larger configurations of parallel computers, often comprising hundreds of processors. Data intensive platforms such as database or web servers and applications such as transaction processing and data mining often use clusters of workstations that provide high aggregate disk bandwidth. Applications in graphics and visualization use multiple rendering pipes and processing elements to compute and render realistic environments with millions of polygons in real time. Applications requiring high availability rely on parallel and distributed platforms for redundancy.

In parallel computing processors are connected through a network. A

wide variety of network topologies have been used in interconnection networks. These topologies try to trade off cost and scalability with performance. Bus-based networks, crossbar networks, multistage networks, completely-connected network, star-connected network, linear arrays, meshes, and k-d meshes, tree-based Networks are examples of some network topologies used in parallel computing. We can represent these topologies by a graph such that the processors are situated on the vertices of that graph. The time taken to communicate a message between two processors is proportional to the distance between them. The processors are divided into two categories: masters and slaves. Masters are designated to manage the pool of available tasks. Slaves communicate with masters to get tasks and provide the results of those tasks. We can model this scenario using online facility assignment. A set of masters with equal capacity is situated on the vertices of a graph. A set of slaves is also situated on the vertices of the same graph. The slaves communicate with the masters in an online manner to get tasks. The communication cost is proportional to the distance between the master and slave. The objective is to minimize the total communication cost. We can use the algorithms of online facility assignment to solve this problem.

1.3.3 Communication Network

We now provide another important application of online facility assignment in communication networks. In a wireless communication network a set of routers is situated on some specific positions. We represent the network using a graph in which the routers are situated on the vertices of that graph. Clients

appear in the vertices of the same graph and connect to the routers to start communication. We assume that each router has equal capacity l. A router is capable to provide connection to at most l Clients. A client wants to connect to a available router immediately after its appearance. The connection cost is the distance between the client and the router which is equal to the hop count of the vertices on which they are situated. This scenario is similar to the online facility assignment problem. The routers and clients are analogous to the facilities and customers of the facility assignment problem respectively. The algorithms of the online facility assignment problem provide different protocols to get efficient assignments of clients to routers.

1.4 Objectives

The objective of this thesis is to provide algorithms for the online facility assignment problem mentioned in Section 1.2. We will study the problem on some specific metric spaces like straight lines and graphs. We will provide the analysis to determine the performance of the algorithms in these metric spaces. The approach of analyzing online algorithms is different from the traditional approach of offline algorithms. The approach of analyzing online algorithms is called *competitive analysis* where the quality of an online algorithm on each input sequence is measured by comparing its performance to an optimal offline algorithm. We will provide the competitiveness of the algorithms we will provide for the facility assignment problem. Also we will provide an experiment to determine the practical performance of our algo-

rithms.

1.5 Literature Review

Facility location problems are concerned with selecting and/or placing certain facilities to serve given demands efficiently. In this problem, a set of customers/demands are given. A set of facilities need to be located to cover all the customers with some specific objectives like minimizing the cost of covering customers. Many economical decision problems can be solved using facility location modeling. Some examples are manufacturing storage facilities, warehouses, networks, fire stations, base stations for wireless services etc. The Fermat-Weber problem is considered the first facility location problem, studied as early as in the 17th century. Later this problem has been extensively studied by different researchers [11, 9, 14, 18, 19, 25]. It is a MAX-SNP hard problem and Shmoys et al. [33] have provided the first constant approximation algorithm. Charikar [10] and Sviridenko [31] later improved approximation ratio to 1.728 and 1.67 respectively. These approximation algorithms can be divided into three categories: rounding algorithms that rely on linear programming, primal-dual and local search algorithms. All of these previous works are based on an offline environment, where demand points (customers) are known ahead of time. Meyerson [27] have provided an $O(\log n)$-competitive algorithm for an online variant of facility location. In this modeling, a set of demand points appear in online. In order to provide services to these demands some facility centers have to be opened provid-

ing a *facility cost* for each center. Also each demand point has to pay a *service cost* to take service from a center. The objective of this problem is to minimize the total facility cost plus service cost. Fotakis [17] presented the first deterministic online algorithm for the same problem which achieves the optimal competitive ratio of $O(\frac{\log n}{\log \log n})$. Anagnostopoulos [3] et al. have further investigated online facility location for similar modelings.

A recently proposed variant of the facility location problem is the r-gathering problem. An r-gathering of a set of customers C to a set of facilities F is an assignment of C to open facilities $F' \subset F$ such that r or more customers are assigned to each open facility. Armon [1] has given a simple 3-approximation algorithm for this problem. Akagi and Nakano [4] have provided a $O((|C|+|F|) \log |C| + |F|)$ time algorithm to solve the r-gathering problem when all customers in C and facilities in F are on the real line.

The k-server problem is an well known online problem proposed by Manasse, McGeoch, and Sleator [28]. Given a metric space $\mathcal{M}$, an algorithm of the k-server problem controls k mobile servers, which are located on points of $\mathcal{M}$ with $|\mathcal{M}| > k$. The algorithm is presented with a sequence $I = \{r_1, r_2, \ldots, r_n\}$ of requests where a request r_i is a point in the space. We say that a request r is served if one of the servers lies on r. By moving servers, the algorithm must serve all requests sequentially. The objective is to minimize the total distance moved by the servers. Manasse [28] et al. provided a 2-competitive algorithm named residues for the 2-server problem, and a k-competitive algorithm named balance for the k-server problem on

$(k+1)$ points metric spaces. Kleinberg [21] showed a universal lower bound on the competitive ratio of any balancing algorithm for 2-server and the lower bound is equal to $(5+\sqrt{7})/2$.

The famous k-server conjecture that any metric space allows for a deterministic k-competitive k-server algorithm [28], is still an open problem. This conjecture played a significant role for the development of competitive analysis. The k-server conjecture has been proved for some special cases, including uniform spaces [32], lines [12], trees[13], weighted stars (all the requests are placed at the leaves of a weighted star) [8], the 3-server problem in the Manhattan plane [6], and spaces with k + 2 points [24].

Despite the many similarities, there is a major difference between the k-server problem and the online facility assignment problem. The servers of the k-server problem are movable. However the positions of facilities are fixed in the online facility assignment problem. Therefore a customer placed very close to a previous customer is easily served in the server problem which is not true for the facility assignment problem.

The facility assignment problem is related to the matching problem [30] which is one of the most fundamental and well-studied optimization problems. The online variation of the matching problem has been extensively studied in the computer science literature [23, 22, 26, 5, 2]. Kao et al. [26] have provided a randomized lower bound of 4.5911 for the online matching on a line problem. We provide a randomized algorithm which is $\frac{9}{2}$-competitive for a class of input sequences. Antoniadis et al. [2] have provided a $o(n)$-

competitive deterministic algorithm for online matching on a line. They have related the lost cow problem to the matching problem to provide this algorithm. We study online facility assignment directly without using any relation to the search problems.

1.6 Results of this Thesis

We study a problem in a modeling where the location of facilities are defined before the arrival of customers. Let $\mathcal{M}$ be a metric space where $\mathcal{M}$ is a set of points. Let $F = \{f_1, f_2, \ldots, f_{|F|}\}$ be a set of facilities located on points of $\mathcal{M}$ with $|\mathcal{M}| > |F|$. Each facility f_i has an initial capacity limit $capacity_i$. The capacity of a facility reduces by one after providing service to a customer. A facility f_i is called *free* if $capacity_i > 0$. The input sequence $I = \{c_1, c_2, \ldots, c_n\}$ is a set of customers which arrive in an online manner and a customer c_i is a point in the space $\mathcal{M}$. An algorithm assigns each customer c_i to a free facility f_j. The distance between c_i and f_j is the cost of that assignment. For any input sequence I and a facility assignment algorithm ALG, Cost_ALG(I) is defined as the total cost of all assignments made by ALG.

We first assume that the metric space $\mathcal{M}$ is a straight line and a set of facilities F is situated on $\mathcal{M}$ having equal distance between adjacent facilities. The customers arrive in an online manner one by one having position on $\mathcal{M}$ and each point is assigned to a facility before the arrival of next customer. We propose an algorithm named Greedy for this problem which is $4|F|$-

competitive. We also introduce randomization in Greedy method and show that it performs better than that without randomization in particular cases. We propose another algorithm named Optimal-Fill which is $|F|$-competitive. We also study another greedy method named Algorithm Capacity-Sensitive-Greedy. We next assume that $\mathcal{M}$ is a connected unweighted graph G. A set of facilities F is situated on the vertices of G and a set of customers arrive one by one in an online manner having position on the vertices of G. We show Greedy is $2|E(G)|$-competitive and Optimal-Fill is $|E(G)||F|/r$-competitive where r is the radius of G. Next we consider that a customer does not stay in a facility forever after its assignment. Instead it leaves the facility after getting service from the facility. We define *service time* as the amount of time needed to get service from a facility. We study the facility assignment problem with a limited service time. We also provide a simulation of the algorithms and define a new parameter named experimental competitive ratio. Finally we analyse the simulation data in terms of experimental competitive ratio.

1.7 Thesis Organization

The remaining of the thesis is organized as follows. In Chapter 2 we provide the preliminaries used throughout the paper. We show the competitive analysis techniques in the context of the k-server problem. We study the proofs of determining competitive ratio of some k-server algorithms demonstrating the basic methods like averaging technique, potential function which are utilized in many other proofs of different online problems. In Chapter 3 we study

the online facility assignment problem on straight lines. In Chapter 4 we study the problem on connected unweighted graphs. We also introduce service time parameter t in our model and show that no deterministic algorithm is competitive when $t = 2$. Finally, in Chapter 5 we provide a simulation of the algorithms of the online facility assignment problem and show the experimental results.

1.8 Conclusion

In this chapter we have characterized online algorithms in the context of the sorting problem. We have provided some applications of online algorithms like paging in a virtual memory system, load balancing in parallel processing and routing in communications networks. We have defined the facility assignment problem, which is the problem we are going to study in this thesis. We have presented its motivation and the objective of this thesis. We have introduced the similar problems found in the literature. Finally we have provided the results of this thesis and its organization.

Chapter 2

Preliminaries

In this chapter we give necessary definitions and terminologies which will be used throughout the thesis. Most of the contents of this chapter are taken from the existing literature [7] in order to study the basic methodologies of analyzing online algorithms. We study the competitive analysis techniques in the context of the k-server problem, a problem of significant historical, theoretical, and practical interest. Before defining the k-server problem, we describe the basic concepts and definitions that will be used throughout our study.

2.1 Offline and Online Optimization Problems

We begin with a discussion of the concept of an *optimization problem*, which may be one of either cost minimization or profit maximization. An optimization problem $\mathcal{P}$ of cost minimization consists of a set $\mathcal{I}$ of inputs and a cost function C. There is a set of feasible outputs (or solutions) $F(I)$ associated with every input I. There is a positive real, $C(I, O)$, representing the cost

of the output O with respect to the input I. The kind of optimization problems we are typically concerned with are of cost minimization; therefore, the discussion here is primarily in terms of cost problems. It is not difficult to develop the analogous concepts for profit maximization problems.

For example, consider the bin packing problem. This problem presents an unbounded number of uniform bins, each having some fixed size or capacity, say 1. An input is a sequence of items $x_1, x_2, \ldots, x_n$ where x_i represents the size of the ith item. All item sizes satisfy $0 < x_i \leq 1$. The goal is to pack all items into bins in the most compact way. Given an input, a feasible solution is any assignment of all the input items x_i such that the sum of the item sizes assigned to any bin does not exceed the bin capacity (which is 1). The cost of a (feasible) solution is the number of bins used to pack all the items. Originally, this bin packing problem was studied as an offline problem. That is, an algorithm is allowed to consider the entire list of items in order to compute the best solution. In order to view bin packing as an online problem, each item x_i must be assigned without knowledge of items $x_{i+1}, \ldots, x_n$ and, of course, the assignment of items $x_1, \ldots, x_i$ must be a feasible solution for each i; furthermore, when we pack x_i, we are not allowed to alter the bin assignment of previously packed items. Thus, an online feasible solution is a sequence of assignments. Exactly as in the offline problem, the total cost of an online solution is the total number of bins used. Equivalently, we can consider the cost of each online assignment of one item to be either 0 or 1, depending on whether or not the item assignment opened a new bin (during

the online assignment) and the total cost becomes the sum of the individual item assignment costs.

Given any legal input I, an algorithm ALG for an optimization problem $\mathcal{P}$ computes a feasible output (solution) $\text{ALG}[I] \in F(I)$. The cost associated with this feasible output is denoted by $\text{ALG}(I) = C(I, ALG[I])$. An *optimal algorithm* OPT is such that for all legal inputs,

$$\text{OPT}(I) = min_{O \in F(I)} C(I, O)$$

An algorithm ALG is a c-approximation algorithm for a minimization problem $\mathcal{P}$ if there is a constant $\alpha \geq 0$ such that for all legal inputs

$$\text{ALG}(I) - c.\text{OPT}(I) \leq \alpha$$

More precisely, such an algorithm is called an asymptotic c-approximation algorithm; we reserve the term c-approximation algorithm to mean $\alpha = 0$.

In the analogous definition of a maximization problem, we require that $\text{OPT}(I) - c.\text{ALG}(I) \leq \alpha$, where $\text{ALG}(I)$ denotes the profit of ALG. In both cases, the approximation factor is greater than or equal 1; the closer it is to 1, the better the approximation.

Optimization problems in which the input is received in an online manner and in which the output must be produced online are called online problems. The complication inherent in online algorithms is that each online output influences the cost of the overall solution. This suggests a natural partition of optimization problems into online and offline problems (and their respective algorithms). Many problems are intrinsically offline. For example, in most

instances of linear programming, it is natural to assume that the input is given offline. In other optimization problems, such as job scheduling and bin packing, both the online and offline versions of the problem are naturally meaningful. Last, many problem such as paging, telephone circuit switching, and investment planning are intrinsically online. For these problems, offline algorithms are not acceptable. Nevertheless, even for intrinsically online problems, we could hypothesize an offline algorithm (that must then have clairvoyant abilities).

2.2 The Competitive Ratio and Competitiveness

An online algorithm ALG is *c-competitive* if there is a constant α such that for all finite input sequences I,

$$\mathrm{ALG}(I) \leq c.\mathrm{OPT}(I) + \alpha$$

When the additive constant α is less than or equal to zero (i.e., $\mathrm{ALG}(I) \leq c.\mathrm{OPT}(I)$), we may say for emphasis that ALG is *strictly* c-competitive. Allowing a positive constant α reflects the view that for intrinsic online problems such as paging, list accessing, and so on, we have an arbitrarily long input sequence with unbounded cost. The constant α becomes insignificant as we consider longer and longer (and more costly) initial subsequences. Moreover, for finite input sequences, the use of the additive constant α allows for an intrinsic performance ratio that does not depend on initial conditions. However, for bounded cost optimization problems such as (offline or online)

graph coloring where at most N colorings are needed, it is clearly more significant if and how α depends on N, the size of the problem.

A (strictly) c-competitive online algorithm ALG is a c-approximation algorithm with the restriction that ALG must compute online. Thus, for each input I, a c-competitive algorithm is guaranteed to incur a cost within a factor c of the optimal offline cost (up to the additive constant α). We note again that the competitive ratio is always at least 1, and the smaller it is, the better ALG performs with respect to OPT. If ALG is c-competitive, we sometimes say that ALG attains a competitive ratio c. An algorithm is called *competitive* if it attains a constant competitive ratio c. Although c may be a function of the problem parameters, it must be independent of the input I. For example, in a scheduling problem concerning N machines, we might have a competitive ratio c that depends on N, but c must be independent of the number and type of jobs being scheduled. The infimum over the set of all values c such that ALG is c-competitive is called the competitive ratio of ALG and is denoted by $\mathcal{R}(ALG)$.

We make no requirements or assumptions concerning the computational efficiency of a competitive online algorithm. In the more traditional offline complexity studies, we are primarily concerned with approximation algorithms that compute within polynomial time. Thus, strictly speaking, c-competitive online algorithms and polynomial time c-approximation algorithms are not comparable. However, in practice, we usually seek efficient competitive online algorithms and, in particular, algorithms that do compute

within polynomial (in the relevant parameters) time.

2.3 Games and Adversaries

There are several meaningful ways to view the problem of analyzing online algorithms. One way, which we use throughout the text, is to view the problem as a game between an online player and a malicious adversary. The online player runs an online algorithm on an input that is created by the adversary. The adversary, based on the knowledge of the algorithm used by the online player, constructs the worst possible input so as to maximize the competitive ratio. That is, the adversary tries to make the task costly to the online player but, at the same time, inexpensive for the optimal offline algorithm. We sometimes identify the adversary and the optimal offline algorithm as one entity, the offline player. For deterministic online algorithms, the adversary knows exactly what the online players response will be to each input element. In other words, the offline player determines the malicious input sequence in advance. For randomized online algorithms, the nature of the offline player is a more subtle issue. We defer the discussion on randomized competitiveness to Section 2.5.

2.4 The k-Server Problem

In this section we study the k-server problem proposed by Manasse, McGeoch, and Sleator [28]. This model provides an interesting abstraction for a number of problems. Moreover, the model and the k-server conjecture has

been a significant catalyst for the development of competitive analysis.

2.4.1 The formulation of the Model

Let $k > 1$ be an integer, and let $\mathcal{M} = (\mathcal{M}, d)$ be a metric space where $\mathcal{M}$ is a set of points with $|\mathcal{M}| > k$ and d is a metric over $\mathcal{M}$. An algorithm controls k mobile servers, which are located on points of $\mathcal{M}$. The algorithm is presented with a sequence $\sigma = r_1, r_2, \ldots, r_n$ of requests where a request r_i is a point in the space. We say that a request r is served if one of the servers lies on r. By moving servers, the algorithm must serve all requests sequentially. For any request sequence σ and any k-server algorithm ALG, $\mathrm{ALG}(\sigma)$ is defined as the total distance (measured by the metric d) moved by ALG's servers in servicing σ. The k-server problem revolves around the question of finding competitive online k-server algorithms for arbitrary and special metric spaces.

For convenience, we sometimes refer to and specify metric spaces as weighted graphs. In this case, we use the standard graph terminology, and we may interchange points with vertices, distance values with edge weights, and so on. $\mathcal{M}$ may be a finite or infinite space. When the metric space $\mathcal{M}$ is finite, we use $N = |\mathcal{M}|$ to denote the size of $\mathcal{M}$.

We define the (h, k)-server problem the performance of the online algorithm with k servers is compared against that of the optimal offline algorithm, which has only $h \leq k$ servers. In the asymmetric version of the problem the distance function d is not symmetric (therefore, the space is not metric). We assume symmetric problems throughout the chapter. Let $G = (V, E, w)$

be any edge weighted undirected (respectively, directed) graph. G induces a k-server (respectively, asymmetric k-server) problem by letting $d(x, y)$ be $w(x, y)$ if (x, y) is an edge in E; otherwise, it is the distance induced by the transitive closure of the relation w (i.e., the least cost path with respect to the edge weights w).

The k-server model provides an abstraction of various interesting problems. Listed below are some examples of problems that are captured by the k-server model.

- Paging: An instance of the k-server problem with a uniform metric space (all distances are 1) where the k servers represent the k memory slots in the cache and $N = |\mathcal{M}|$ is the number of slow memory pages.

- Weighted paging: A paging problem in which the cost of copying different pages into the cache varies. These problems naturally occurs in computer systems. For example, in a distributed operating system that uses a distributed file system, page access costs may vary depending on communication costs and architectures of the various machines. A virtual memory management system in which pages can have different sizes (e.g., the bitmaps of fonts that must be cached by the display unit) provides another example of weighted paging. Weighted paging is an instance of an asymmetric k-server problem in which the cost of moving a server from point x to point y (i.e., evicting x in order to bring in y) can be different from the cost of moving the server from y to x. Alternatively, we can view weighted paging as a symmetric server

system.

- Two-headed (k-headed) disk: Two (k) read/write heads move along a line segment that is a radius of the disk. When coordinated with the disk spin, they can access every location on each disk track. An algorithm must determine which of the two (k) heads to move in order to service the next read/write request from/to a certain location in a given track. One meaningful way to measure the performance of such a system is to measure the total distance moved by the heads. This is equivalent to the 2-server (k-server) problem on a line segment.

2.4.2 Some Basic Aspects of the k-Server Problem

In this section we present some basic concepts and observations concerning k-server problems from the existing literature [7].

The Optimal Offline Algorithm

For any request sequence σ, the optimal offline cost and the optimal offline schedule to serve σ can be computed by using dynamic programming. The straightforward dynamic programming approach to computing an optimal offline k-server schedule is not the most efficient algorithm. An alternative, faster method of computing the optimal offline cost and schedule is achieved by reduction to a minimum cost/maximization flow problem. When we use this method, the time needed to calculate the optimal offline cost and schedule is $O(kn^2)$, where n is the number of requests in σ.

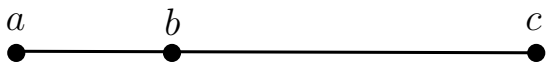

Figure 2.1: A 3-node graph

The Greedy Policy is Not Competitive

For the k-server problem, we can define a greedy online algorithm as any algorithm that processes each request in order to minimize the cost (or maximize the profit) on the input sequence seen thus far. In general, there can be many greedy choices for a given request. It is equivalent to say that greedy algorithms must process each request in order to minimize the individual cost for serving the request; that is, serve each request with a server that is nearest to the request.

It is easy to see why a greedy policy is not necessarily competitive. Consider the 2-server problem when the metric space is the 3 node graph shown in Figure 2.1. In the figure, $d(a, b) < d(b, c)$. No matter what the initial positions of the two server are, the greedy algorithm will service the request sequence $c, b, a, b, a, b, a, \ldots$ as follows: one server will remain at c forever, and all the requests for a and b will be served by a single server incurring an unbounded cost. Clearly, the optimal offline algorithm can serve this request sequence (from any starting configuration) with a total cost no greater than $d(a, b) + 2d(b, c)$.

Lazy Algorithms

A straightforward observation is that we may restrict our attention to server algorithms that move at most one server in response to each request, and we

may do so only if the request is not presently covered by a server. We call such algorithms *lazy*. The reduction is easily justified: since each request can be serviced by one server, if an algorithm wants to move other servers to some new locations, it can store these locations in memory and move a server (according to the stored locations) only when it is its turn to service a request. According to the triangle inequality, the total distance incurred by the modified algorithm is no more than the total distance incurred by the original one.

The k-Server Conjecture

The following conjecture was formulated by Manasse, McGeoch, and Sleator [28]:

The k-server conjecture: *Any metric space allows for a deterministic k-competitive, k-server algorithm.*

After considerable effort, researchers have "nearly" solved the k-server conjecture. Koutsoupias and Papadimitriou [24] have shown that there is a generic k-server algorithm (the work function algorithm) that is $(2k-1)$-competitive in any metric space. In addition, there are various examples of special metric spaces that allow for k-competitive server algorithms. Considering these results, we may believe that the conjecture is true. On the other hand, it may appear somewhat strange if this conjecture holds: most other natural generalizations of the results on paging (that hold for paging) are known to be false for k servers. Here are two examples:

- It is not true that any metric space allows for a deterministic $\frac{k}{k-h+1}$-

competitive k-server algorithm for the (h, k)-server problem (in contrast, LRU and many other algorithms are $\frac{k}{k-h+1}$-competitive for paging).

- It is not true that in any metric space a randomized competitive ratio of H_k is attainable against an oblivious adversary. In contrast, there is an H_k-competitive for paging.

In addition, the k-server conjecture does not generalize to asymmetric spaces.

A Deterministic Lower Bound

In this section we present a lower bound of k on the competitive ratio of any deterministic k-server algorithm in a metric space with at least $k + 1$ points proved by Manasse et al. [28]. In fact, we have a stronger result, a lower bound of $\frac{k}{k-h+1}$ on the competitive ratio of any server algorithm for the (h, k)-server problem. The proof of the following lower bound will make use of an averaging technique that is utilized in many other proofs of different online problems. In this technique we make use of a cruel adversary that causes the online player to pay the maximum for every request.

Theorem 1. [28] Let $\mathcal{M}$ be any metric space with at least $k + 1$ points. For any $1 \leq h \leq k$, $\frac{k}{k-h+1}$ is a lower bound on the competitive ratio of any online (h, k)-server algorithm for $\mathcal{M}$.

Proof. Let ALG be any k-server algorithm. We show that there exists an arbitrarily long (cruel) request sequence σ such that $\mathrm{ALG}(\sigma) \geq (\frac{k}{k-h+1}).\mathrm{OPT}(\sigma)$.

Without loss of generality, we assume that ALG is lazy. For the construction of the cruel request sequence, fix any set M of $k+1$ points in the space.

Let the initial configuration of ALG be $\{1, 2, \ldots, k\}$. Since ALG is lazy, it always occupies exactly k distinct points in M (assuming that ALG's initial configuration contains k distinct points). Let $\sigma = r_1, r_2, \ldots, r_n$ be a sequence requesting at each step the (unique) point unoccupied by ALG. This means that ALG serves r_i with the server positioned on the point r_{i+1} incurring a cost of $d(r_{i+1}, r_i)$. For such a (cruel) request sequence σ of length $n \geq 2$, the total cost incurred by ALG is

$$\begin{aligned} \text{ALG}(\sigma) &= \sum_{i=1}^{n-1} d(r_{i+1}, r_i) \\ &= \sum_{i=1}^{n-1} d(r_i, r_{i+1}) \quad \text{(using symmetry)} \end{aligned}$$

We now prove the existence of an offline h-server algorithm that serves σ while incurring a cost of no more than $\text{ALG}(\sigma)/(\frac{k}{k-h+1})$.

We define a set $\mathcal{B}$ of particular h-server algorithms. For each h-subset $S \subset M$ with $r_1 \in S$ (recall that $r_1 = 0$ is the unique point not occupied by ALG in the initial configuration), let B_S be an offline algorithm that operates as follows:

Algorithm B_S starts in the initial configuration S and serves an uncovered request $r_i (i = 2, 3, \ldots, n)$ with the server occupying r_{i-1} (by definition $r_1 \in S$). Note that B_S is well defined, since one of its servers must occupy r_{i-1} when r_i is requested. *Note:* If one assumes that the adversary starts from the same configuration as ALG, then B_S will pay at most $\max_{p \in M} d(p, 0)$ to move to the initial configuration.

Let $\mathcal{B}$ be the set of all such algorithms B_S. We now claim that the number of algorithms in the set $\mathcal{B}$ is fixed throughout the game; that is, after each request is processed, different algorithms will always be in different configurations. To prove this claim, it is useful to use the following notation: for each algorithm B_S (with an initial configuration S), let $S^i, i = 0, 1, 2, \ldots$ denote the configuration of B_S after it serves the ith request ($S^0 = S$). We prove the claim by induction on i, where the induction statement is:

Induction statement: If $S_1 \neq S_2$, then the algorithms B_{S_1} and B_{S_2} are such that $S_1^i \neq S_2^i$ for all $i = 0, 1, \ldots, n$.

The base case ($i = 0$) is true by definition. A brief case analysis proves the induction step:

- *Case 1:* r_i is in S_1^{i-1} and S_2^{i-1}. In this case, both algorithms do not move; therefore, according to the induction hypothesis, $S_1^i = S_1^{i-1} \neq S_2^{i-1} = S_2^i$.

- *Case 2:* r_i is in, for example, S_1^{i-1} but not in S_2^{i-1}. B_{S_1} does not

move, and B_{S_2} serves the request with the server located at r_{i-1}. Thus, $S_1^i = S_1^{i-1}$ contains r_{i-1} but S_2^i does not.

- *Case 3:* r_i is in neither S_1^{i-1} not S_2^{i-1}. Clearly, $S_1^{i-1} - \{r_{i-1}\} \neq S_2^{i-1} - \{r_{i-1}\}$, and both algorithms serve the request with the server from r_{i-1}, so $S_1^i \neq S_2^i$.

We conclude that the configurations of the algorithms in $\mathcal{B}$ remain distinct throughout the game. Hence, at all times $\mathcal{B}$ contains exactly $\binom{k}{h-1}$ algorithms corresponding to all h-subsets of M that include the first request. Now, what is the total cost incurred by all algorithms in $\mathcal{B}$?

For each request r_i (except the first one), exactly $\binom{k-1}{h-1}$ algorithms move a server (from r_{i-1}) in order to serve this request. These algorithms correspond to all $(h-1)$-subsets of M that include r_{i-1} but not r_i. Hence, the total cost incurred by all algorithms in $\mathcal{B}$ is

$$\binom{k-1}{h-1} . \sum_{i=2}^{n} d(r_{i-1}, r_i) = \binom{k-1}{h-1} . \sum_{i=1}^{n-1} d(r_i, r_{i-1})$$

It follows that the average performance of these algorithms is

$$\frac{\binom{k-1}{h-1}}{\binom{k}{h-1}} . \sum_{i=1}^{n-1} d(r_i, r_{i+1}) = \binom{k-h+1}{k} . \sum_{i=1}^{n-1} d(r_i, r_{i+1})$$

Since at least one of these algorithms incurs costs no greater than this average, the proof is complete. □

2.4.3 k-Servers on a Line

For most natural geometric structures (e.g., the Euclidean plane of the circle), no simple k-competitive deterministic algorithms are known. In contrast, there is a simple k-competitive algorithm for the Euclidean line [12], and this algorithm can be generalized to metric spaces defined by trees.

We only consider the real line with Euclidean metric. The metric space allows for a simple deterministic k-competitive k-server algorithm. The algorithm is one of the most elegant k-server algorithms. This line algorithm

naturally generalizes to trees and subsequently has several interesting applications. We now describe the algorithm for the real line called DOUBLE-COVERAGE or DC for short.

Algorithm DC: If the request falls outside the convex hull of the servers, serve it with the nearest server. Otherwise, the request is in between two adjacent servers. In this case, move both these servers toward the request at equal speeds until (at least) one server reaches it. (If two servers occupy the same point, then choose one arbitrarily.)

Notice that DC is not a lazy algorithm; it may move one or two servers as a response to some requests. Of course, it may operate as a lazy algorithm by remembering the virtual locations of the servers as described before. However, sometimes (and, indeed, in this case) it is more intuitive and easier to analyze nonlazy algorithms, especially since the nonlazy moves have some intuitive meaning.

Consider the response of DC (with two servers) to the request sequence that defeated the greedy algorithm (see Figure 2.1). Suppose that the two servers, s_1 and s_2, of DC are positioned on b and c, respectively. Assume, for this example, that $d(a,b) << d(b,c)$. Consider the behavior of DC in response to the sequence $a, b, a, b, a, b, \ldots$ of requests. The first request for a is served by s_1; then, the request for b is also served by s_1, but this time s_2 moves a distance $d(a,b)$ toward b. This pattern of response continues; consequently, DC does not fall for the greedy trap by gradually shuttling the server s_2 to help server s_1 on the western front. Clearly, after sufficiently

many such repetitions, DC positions its two servers on a and b.

Theorem 2. [12] DC is k-competitive.

Proof. Let M_{min} denote the minimum cost matching between OPT's and DC's servers. For $i = 1, 2, \ldots, k$, denote by s_i DC's servers. Let $\sum_{DC}$ denote the sum of all interpoint distances between DC's servers: $\sum_{DC} = \sum_{i<j} d(s_i, s_j)$. The proof uses the following potential function:

$$\Phi = k.M_{min} + \sum_{DC}$$

Note that Φ is nonnegative and, hence, bounded below as required for a potential function. Using the potential function method it is sufficient to prove the following. (i) If the adversary moves a distance d, the potential is increased by at most kd. (ii) If DC moves d, the potential is decreased by at least d.

Proving that (i) holds is almost trivial (notice that the adversary's move does not affect the $\sum_{DC}$-component and, clearly, M_{min} cannot increase by more than d). To prove that (ii) holds, we consider two possible kinds of moves by DC. First, suppose DC moves a single server, say, a distance d. According to the definition of DC, this server is an extreme point of the convex hull of all the servers. Since this server is moving away from all other servers, $\sum_{DC}$ increases by $(k-1)d$. However, there exists a minimum weight matching in which this moving server is matched to the request, so M_{min} decreases by at least d. Thus, the total decrease of potential is at least $kd - (k-1)d = d$, which is exactly the cost incurred by DC.

In the second case, suppose servers s_1 and s_2 move toward the request from opposite sides. Suppose also that each of these servers moves a distance d. Clearly, one of these servers is matched to the request (in some minimum weight matching), so its move decreases M_{min} by at least d. The other moving server may move away a distance d from its match. Hence, M_{min} does not increase overall. Now consider the change in $\sum_{DC}$. The change in the sum of distances from s_1 and s_2 to any other server q is zero: one of them is moving away a distance d from q, and the other is moving a distance d toward q. However, the distance between s_1 and s_2 is shortened by $2d$, which is the total online move. This prove (ii). □

2.5 Randomized Algorithms

When dealing with deterministic online algorithms, we have one natural model for the adversary. This adversary knows the online algorithm and chooses the worst input sequence in order to maximize the competitive ratio. This is no longer the case with randomized algorithms. In randomized algorithms online players use randomness. The adversary does not exactly know the outcomes of the random choices made by the online player. By concealing such knowledge from the adversary, the online player can fool the adversary so that there is uncertainty as to what is the worst possible request sequence.

As in the deterministic case, we should like to measure the quality of a randomized online algorithm by a quantity similar to competitive ratio. That is, we consider a game between an online player (algorithm) and an adversary that constructs the input sequence in order to maximize the ratio of the expected online cost to the adversary cost. However, this adversary cost may have various forms depending on the exact nature of the adversary model [7].

2.5.1 Adversary Models

The key issue that requires distinction between possible adversary models is the extent to which the adversary knows (and can exploit) the outcomes of the random choices made by the online player. We first demonstrate the simplest adversary model which is called the oblivious adversary.

Let ALG be a randomized online algorithm. Based on the knowledge of the probability distribution(s) ALG uses, the oblivious adversary must choose a finite request sequence σ in advance. ALG is c-competitive against an oblivious adversary if for every such σ,

$$E[\mathrm{ALG}(\sigma)] \leq c.\mathrm{OPT}(\sigma) + \alpha$$

Here α is a constant independent of σ and $E[.]$ is the mathematical expectation operator taken with respect to the random choices made by ALG. Since the offline player does not have information about the outcomes of the random choices made by the online player, $\mathrm{OPT}(\sigma)$ is not a random variable; consequently, there is no need to take its expectation. As is true for deterministic algorithms, the infimum c, such that ALG is c-competitive against an oblivious adversary, is called ALG's expected competitive ratio against an oblivious adversary. We write $\overline{\mathcal{R}}(\mathrm{ALG}) = c$.

We now introduce two other types of adversaries. In all three type, we assume that the adversary knows the online algorithm (including, of course, the probability distributions used by the algorithm).

The first distinction is between oblivious and adaptive adversaries: at each time, an adaptive adversary knows all the actions taken by the online player for servicing the requests thus far. The adaptive adversary may choose the next request based on this knowledge. In contrast, an oblivious adversary must choose the entire request sequence in advance, without any knowledge of the actions taken by the online player. Oblivious adversaries are more standard in that they correspond to the adversaries that we use in the analysis

of offline randomized algorithms. Adaptive adversaries can be motivated in several ways. First, we can envisage online problems where the actions of the online algorithm do influence future requests. For example, consider a paging algorithm being used by a real time application that reads the system clock to determine appropriate actions. A random choice by the paging algorithm will influence the timing and, consequently, the actions (including future page requests) of the real time application. Another motivation for studying adaptive adversaries is that in some cases it may be easy to design a randomized algorithm that is competitive against an adaptive adversary.

The oblivious adversary cost is measured exactly as in the deterministic case: via the optimal offline cost. Measuring the adversary cost of the adaptive adversary is a bit more subtle, and we make a further distinction between two types of adaptive adversaries. The first type is called adaptive-offline, and its adversary cost is, again, the optimal offline cost (on the request sequence that is created online by this adversary). There is also an intermediate type of adversary called adaptive-online, which is less powerful. This adversary must service each request it generates before the online player services the request: in a sense, then, the adaptive online adversary is also performing in an online fashion except that it knows its own strategy for generating requests as well as the description of the online algorithm and all its actions taken thus far. To summarize, we consider the following three types of adversaries:

- OBL (oblivious): must construct the request sequence in advance and

pays optimally.

- ADON (adaptive-online): serves the current request online and then chooses the next request based on the online algorithm's actions so far.

- ADOF (adaptive-offline): chooses the next request based on the online algorithm's actions thus far, but pays the optimal offline cost to service the resulting request sequence.

Having defined these three adversary types, we have yet to define the competitive ratio with respect to each adversary. In general, let ADV be an adversary (of type OBL, ADON, or ADOF). We say that the online algorithm ALG is c-competitive against ADV if there exists a constant α such that for all request sequences, σ,

$$E[\mathrm{ALG}(\sigma) - c.\mathrm{ADV}(\sigma)] \leq \alpha$$

Here the expectation is taken over the random choices made by ALG, and $\mathrm{ADV}(\sigma)$ is the adversary cost. As usual, α is referred to as the adaptive constant.

Both $\mathrm{OBL}(\sigma)$ and $\mathrm{ADOF}(\sigma)$ are exactly $\mathrm{OPT}(\sigma)$. That is, the adversary cost in the case of oblivious or adaptive-offline adversaries is exactly the optimal offline cost to serve σ. However, there is a fundamental difference between $\mathrm{OBL}(\sigma)$ and $\mathrm{ADOF}(\sigma)$. $\mathrm{OBL}(\sigma)$ is a fixed quantity, since the oblivious adversary constructs the request sequence independently of the random choices made by ALG. On the other hand, $\mathrm{ADOF}(\sigma)$ is a random variable, since the choice of σ depends on the random choices made by ALG. That

is, σ is a random variable and, therefore, $\text{ADOF}(\sigma) = \text{OPT}(\sigma)$ is a random variable as well.

In the case of an adaptive-online adversary, σ and therefore $\text{ADON}(\sigma)$ are again, random variables. However, in this case, we do not have a concise characterization of the adversary cost (such as the optimal offline cost).

To summarize, in the case of an oblivious adversary, the inequality defining the competitive ratio can be written as

$$E[\text{ALG}(\sigma)] - c.\text{OPT}(\sigma) \leq \alpha$$

In the case of the adaptive-offline adversary,

$$E[\text{ALG}(\sigma) - c.\text{OPT}(\sigma)] \leq \alpha$$

Given any problem and a randomized online algorithm ALG for this problem, we define $\overline{\mathcal{R}}_{ADV}(\text{ALG})$, the competitiveness of ALG against an adversary of type ADV, as the infimum over all numbers c such that ALG is c-competitive against ADV. Whenever the type of the adversary ADV is clear, we may omit the subscript ADV from the functional $\overline{\mathcal{R}}$.

2.5.2 Randomized k-Server Algorithm

In this section we present a k-server algorithm for the circle [20]. The algorithm works against oblivious adversary. The algorithm works in any metric space that can be embedded in a circle and attains a competitive ratio of $2k$. We now describe the algorithm called CIRC for a circle with circumference C.

Algorithm CIRC: The online player chooses randomly and uniformly a point P on the circumference of the circle. Think about the point P, which remains unknown to the (oblivious) adversary, as a roadblock that breaks the circle into a line segment. On this line segment the online player plays the k-server game according to the optimal (deterministic) line algorithm DC (DOUBLE-COVERAGE) of Section 2.4.3.

Clearly, the above algorithm is a mixed algorithm; it makes only a single random choice. Note, however, that this single random choice is a real number. A more important feature is that this algorithm introduces a general approach for constructing randomized algorithms against an oblivious adversary.

Theorem 3. [20] *CIRC is $2k$-competitive against an oblivious adversary.*

Proof. Call OPT-LINE the optimal offline algorithm that serves a request within the line segment (as defined by P). OPT is the optimal offline algorithm for the circle. Clearly, for any request sequence σ,

$$\text{CIRC}(\sigma) \leq k.\text{OPT-LINE}(\sigma) \tag{2.1}$$

We now bound OPT-LINE(σ) from above in terms of OPT(σ). Consider an algorithm OPT' that behaves exactly as OPT does except that whenever OPT crosses the point P, OPT' makes a detour along the circle, paying C. Since OPT' is an algorithm for the line segment, OPT-LINE(σ) $\leq$ OPT'(σ). Every distance d_i traveled by OPT's servers crosses P with probability at most $\frac{d_i}{C}$. Hence the expected cost for all detours is at most $D = \sum_i \frac{d_i}{C}.C =$ OPT(σ). To summarize, we obtained

$$\text{OPT-LINE}(\sigma) \leq E[\text{OPT'}(\sigma)] \leq \text{OPT}(\sigma) + D = 2.\text{OPT}(\sigma) \tag{2.2}$$

From equations 2.1 and 2.2,

$$E[\text{CIRC}(\sigma)] \leq 2k\text{OPT}(\sigma)$$

□

2.6 Conclusion

In this chapter we have studied the competitive analysis techniques in the context of the k-server problem. The competitive analysis can be viewed as a game between an online player and a malicious adversary. The adversary tries to make the task costly to the online player but, at the same time, inexpensive for the optimal offline algorithm. There are different kinds of adversaries when the online player uses randomized algorithm. We have studied these models of adversaries. We have studied the proofs of determining competitive ratio of some k-server algorithms demonstrating the basic methods like averaging technique, potential function which are utilized in many other proofs of different online problems.

Chapter 3

Facility Assignment on Straight Lines

In this chapter, we study online facility assignment on straight lines. We assume a set of facilities F of equal capacity l is situated on a straight line having equal distance between adjacent facilities and customers arrive one by one having position anywhere on the straight line of facilities. In Section 3.1 we first propose Algorithm Greedy which is $4|F|$-competitive. In Section 3.2 we introduce randomization in the greedy method and show that it is $\frac{9}{2}$-competitive for a class of input sequences. In Section 3.3 we provide another method Algorithm Optimal-Fill which is $|F|$-competitive.

3.1 A Greedy Approach

It is hard, if not impossible, to define precisely what is meant by a *greedy algorithm*. An algorithm is greedy if it builds up a solution in small steps, choosing a decision at each step myopically to optimize some underlying criterion. One can often design many different greedy algorithms for the same

problem, each one locally, incrementally optimizing some different measure on its way to a solution.

3.1.1 Algorithm Greedy

In this section we propose Algorithm Greedy to solve the problem. Algorithm Greedy assigns a customer to the nearest free facility. The algorithm is given below.

Algorithm Greedy

Input: $F = \{f_1, f_2, \ldots, f_{|F|}\}$:facilities, l:capacity limit,
$I = \{c_1, c_2, \ldots, c_n\}$:input customers
Output: An assignment of the customers to the facilities and the total cost of that assignment

```
for i ← 1 to f do
    capacity_i = l;
for i ← 1 to n do
    min ← ∞;
    index ← −1;
    for j ← 1 to f do
        if capacity_j > 0 and distance(f_j, c_i) < min then
            min ← distance(f_j, c_i);
            index ← j;
    assign c_i to f_index;
    capacity_index ← capacity_index − 1;
    sum ← sum + min;
```

Result: An assignment of the customers having total cost equal to *sum*

3.1.2 Competitive Analysis

There are several meaningful ways to view the problem of analyzing online algorithms. A problem can be viewed as a game between an *online player* and a malicious *adversary*. The online player runs an online algorithm on an input that is created by the adversary. The adversary, based on the

knowledge of the algorithm used by the online player, constructs the worst possible input so as to maximize the competitive ratio. In the context of facility assignment problem, the online player runs Algorithm Greedy. The adversary runs the optimal algorithm OPT. The adversary tries to make the task costly for Algorithm Greedy but, at the same time, inexpensive for the OPT. The following lemma provides a lower bound for the expense of OPT.

Lemma 1. *Let d be the distance between two adjacent facilities. If the assignments of OPT and Algorithm Greedy are not same, then optimal cost is at least $\frac{d}{2}$.*

Proof. Let c_x be the first customer for which the assignments of OPT and Algorithm Greedy are different. The optimal cost for assigning c_x is at least $\frac{d}{2}$. Hence the total optimal cost is at least $\frac{d}{2}$. □

The following theorem determines the worst input sequence an adversary can construct for Algorithm Greedy and provides the competitive ratio.

Theorem 4. *Let $\mathcal{M}$ be a straight line and a set of facilities F is situated on $\mathcal{M}$ having equal distance between two adjacent facilities. Then $\mathcal{R}$(Algorithm Greedy) $\leq 4|F|$.*

Proof. We can consider two types of input sequences an adversary can construct; in the optimum assignment all customers have cost less than d (distance between two adjacent facilities) and k customers have distance greater than d. For both cases, we can simply assume that the facilities have unit capacity since the analysis is similar for capacity l, where $l > 1$. Also we consider all points are placed between the first and last facilities since competitive ratio does not increase if points are placed outside of this area.

We first consider all customers have cost less than d in the optimum assignment. If an input I of customers has this property then we say I is *well distributed*. In the worst case, the adversary places all the customers very close to the facilities except the first customer c_1 as illustrated in Figure 3.1. The total cost of Algorithm Greedy is no more than $2|F|d$. In optimum assignment all customers c_i have cost ε_i except c_1. The cost of first customer c_1 is γ, where $\gamma > \frac{d}{2}$ (lemma 1).

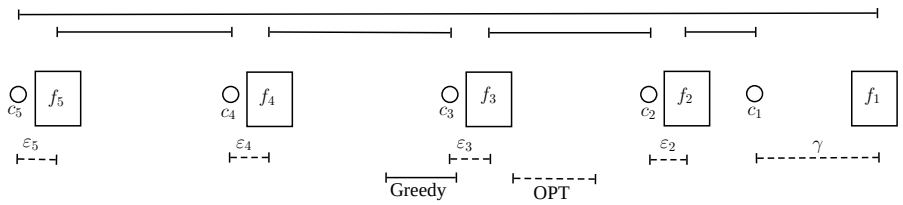

Figure 3.1: The configurations of Algorithm Greedy and OPT

$$\frac{\text{Cost\_Algorithm\_Greedy}(I)}{\text{Cost\_OPT}(I)} \leq \frac{2|F|d}{\frac{d}{2}} = 4|F|$$

In the second case, k customers have cost greater than d in optimum assignment. Hence total cost of optimum assignment is at least kd. It has been assumed that the customers having distance less than d are assigned with cost zero by optimal algorithm. There are $|F| - k$ such customers. In the assignment of Algorithm Greedy, each of these customers would have cost at most d. Notice that if any of these customers have cost greater than d then that assignment can be easily transformed to an equivalent assignment such that total cost is same as original assignment and $|F| - k$ customers have cost no more than d. Each of the remaining k points would have a cost at most $(|F| - 1)d$.

$$\begin{aligned} \frac{\text{Cost\_Algorithm\_Greedy}(I)}{\text{Cost\_OPT}(I)} &\leq \frac{(|F|-1)dk+(|F|-k)d}{kd} \\ &= \frac{|F|(k+1)-2k}{k} \\ &= \frac{(k+1)|F|}{k} - 2 \end{aligned}$$

□

3.2 Introducing Randomization in Greedy Approach

In this section we introduce randomness to the greedy method of previous section and show that smaller competitive ratios are attainable if the online player uses randomness. When dealing with deterministic online algorithms, the adversary knows the online algorithm and chooses the worst input sequence in order to maximize the competitive ratio. This is no longer the case

with randomized algorithms. In this case, the adversary does not exactly know the outcomes of the random choices made by the online player. By concealing such knowledge from the adversary, the online player can fool the adversary so that there is uncertainty as to what is the worst possible request sequence.

Let ALG be a randomized online algorithm. Based on the knowledge of the probability distribution(s) ALG uses, the oblivious adversary must choose a finite request sequence I in advance. ALG is c-competitive against an oblivious adversary if for every such I,

$$E[\text{Cost\_ALG}(I)] \leq c.\text{Cost\_OPT}(I) + \alpha$$

where α is a constant independent of I, and $E[.]$ is the mathematical expectation operator taken with respect to the random choices made by ALG. Since the offline player does not have information about the outcomes of the random choices made by the online player, Cost_OPT(I) is not a random variable; consequently, there is no need to take its expectation.

3.2.1 Algorithm σ-Randomized-Greedy

We introduce randomness in the Greedy method described in previous section and call the new method Algorithm σ-Randomized-Greedy. Let f_x be a facility which is nearest to the the customer c_y and σ is a real number. σ-Randomized-Greedy checks whether distance between c_y and f_x is less than σ or not. If it is less than σ, c_y is assigned to f_x. Otherwise σ-Randomized-Greedy tosses a fair coin before assigning a customer to a facility. It chooses

the nearest free facility at right (left) side when head (tail) appears. Our algorithm is given below.

Algorithm σ-Randomized-Greedy

Input: $F = \{f_1, f_2, \ldots, f_{|F|}\}$:facilities, l:capacity limit,
$I = \{c_1, c_2, \ldots, c_n\}$:input customers

Output: An assignment of the customers to the facilities and the total cost of that assignment

for $i \leftarrow 1$ **to** $|F|$ **do**
  $capacity_i = l$;

for $i \leftarrow 1$ **to** n **do**
  $min \leftarrow \infty$;
  $index \leftarrow -1$;
  for $j \leftarrow 1$ **to** $|F|$ **do**
    if $capacity_j > 0$ *and* $distance(f_j, c_i) < min$ **then**
      $min \leftarrow distance(f_j, c_i)$;
      $index \leftarrow j$;
  if $min \geq \sigma$ **then**
    randomly select a facility f_k from the nearest left and right free facilities;
    $min \leftarrow distance(f_k, c_i)$;
    $index \leftarrow k$;
  assign c_i to f_{index};
  $capacity_{index} \leftarrow capacity_{index} - 1$;
  $sum \leftarrow sum + min$;

Result: An assignment of the customers having total cost equal to sum

3.2.2 Competitive Analysis

Algorithm σ-Randomized-Greedy performs better than Algorithm Greedy as shown in the following theorem.

Theorem 5. *Let I be a well distributed request sequence for Algorithm Greedy. Let γ be the optimal cost of first customer and ε_i be the optimal cost of i^{th} customer where $i > 1$. If $\sigma > \varepsilon_i$ for all i and $\sigma \leq \gamma$ then Algorithm σ-Randomized-Greedy is $\frac{9}{2}$-competitive for I.*

Proof. In Theorem 4, a well distributed request sequence for Algorithm Greedy has been defined. The first customer c_1 is placed relatively closer to f_2 (Figure 3.1) in order to fool Algorithm Greedy. Algorithm Greedy assigns c_1 to f_2. Except the first customer c_1, the adversary places every customer c_k very close to facility f_k. Since Algorithm Greedy has already assigned c_1 to f_2, it can not assign c_2 to the same facility. Similarly for every customer c_k, Greedy assigns it to f_{k+1} although it is very close to f_k. Algorithm σ-Randomized-Greedy overcomes this situation by using randomness. Consider the first customer c_1 which is close to f_2. Algorithm σ-Randomized-Greedy chooses either f_1 or f_2 with equal probability $\frac{1}{2}$. Similarly for every customer c_k, Algorithm σ-Randomized-Greedy chooses either f_{k+1} or f_k with equal probability $\frac{1}{2}$.

$$\begin{aligned} E[\text{Cost\_}\sigma\text{-Randomized-Greedy}(I)] &= \tfrac{d}{4} + \textstyle\sum_{i=1}^{|F|-2}\{\tfrac{1}{2^{i+1}}(2id - \tfrac{d}{2})\} \\ &\quad + \tfrac{1}{2^{|F|-1}}\{2(|F|-1)d - \tfrac{d}{2}\} \\ &< \tfrac{d}{4} + d\textstyle\sum_{i=1}^{|F|-2}\tfrac{i}{2^i} \\ &< \tfrac{d}{4} + 2d \\ &= \tfrac{9d}{4} \end{aligned}$$

Since optimum cost is at least $d/2$, Algorithm σ-Randomized-Greedy is $\frac{9}{2}$-competitive for I. □

3.3 A More Complex Approach

In Section 3.1, it has been shown that Algorithm Greedy can be easily fooled by placing all the customers very close to the facilities except the first customer. In this section, Algorithm Optimal-Fill is proposed which is more efficient than Algorithm Greedy.

3.3.1 Algorithm Optimal-Fill

When a new customer c_i is placed, Algorithm Optimal-Fill finds out the new facility f_j that would be selected by an optimal assignment of the customers $c_1, c_2, \ldots, c_i$. Algorithm Optimal-Fill then assigns c_i to f_j. The algorithm is

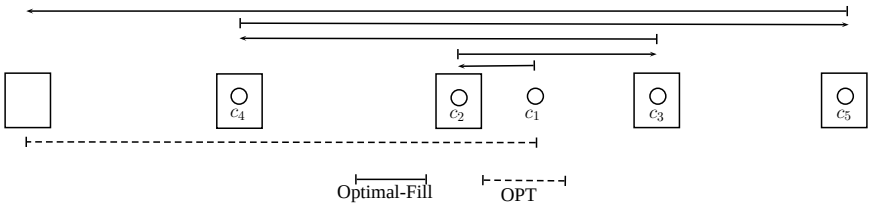

Figure 3.2: Worst case for Algorithm Optimal-Fill

given below.

Algorithm Optimal-Fill

Input: $F = \{f_1, f_2, \ldots, f_{|F|}\}$:facilities, $I = \{c_1, c_2, \ldots, c_n\}$:input customers

Output: An assignment of the customers to the facilities and the total cost of that assignment

for $i \leftarrow 1$ **to** n **do**

  let f_j be the new facility chosen by an optimal assignment of the customers $c_1, c_2, \ldots, c_i$;

  assign c_i to f_j;

  $sum \leftarrow sum + distance(f_j, c_i)$;

Result: An assignment of the customers having total cost equal to sum

3.3.2 Competitive Analysis

The following theorem shows that Algorithm Optimal-Fill performs better than deterministic greedy method.

Theorem 6. *Let $\mathcal{M}$ be a straight line and a set of facilities F is situated on $\mathcal{M}$ having equal distance between two adjacent facilities. Then $\mathcal{R}$(Algorithm Optimal-Fill) $\leq |F|$.*

Proof. In the worst case, the adversary places the customers such that it can assign them with out any cost. However Algorithm Optimal-Fill has to pay a large amount of cost for each of these customers. The adversary pays only for the first customer (see Figure 3.2).

For the following customers the adversary does not pay any cost, because they are situated exactly on their facilities. However Algorithm Optimal-Fill have to pay at least d for these customers.

$$\begin{aligned}\frac{\text{Cost\_Algorithm\_Optimal-Fill}(I)}{\text{Cost\_OPT}(I)} &= \frac{d+2d+\cdots+(|F|-1)d+\frac{d}{2}}{\frac{|F|d}{2}} \\ &= \frac{|F|(|F|-1)+1}{|F|} \\ &< \frac{|F|(|F|-1)+|F|}{|F|} \\ &= |F|\end{aligned}$$

Algorithm Optimal-Fill pays the maximum cost possible in each step. Hence the total cost of Algorithm Optimal-Fill does not increase in other cases. However the total cost of OPT may increase. Therefore $\mathcal{R}$(Algorithm Optimal-Fill) $\leq |F|$.

□

3.4 Capacity Sensitive Greedy Approach

In Section 3.1 we provide Algorithm Greedy which assigns a customer to the nearest free facility. The algorithm divides the whole metric space in some *cover areas*. Consider a facility f_i has two adjacent free facilities f_j and f_k. Let p_1 (p_2) be the middle point of f_i and f_j (f_k). The cover area of f_i is the line segment from p_1 to p_2. If a customer c_i appears in the cover area of f_i then c_i is assigned to f_i. The cover area of a facility in Algorithm Greedy is not sensitive to the amount of capacity. The distance between a customer and a facility is defined as the euclidean distance between them. The assignment of a customer to a facility is made based on this distance and the capacity of facilities does not play a significant role in this assignment. Also the competitive analysis of these algorithms does not depend on the value of initial capacity of the facilities. The analysis is similar for the cases where initial capacity is greater than one and equal to one. The following lemma provides an upper bound for Algorithm Greedy.

Lemma 2. *Let $\mathcal{M}$ be a straight line and two facilities having distance d are*

situated on $\mathcal{M}$. *Then Cost_Algorithm_Greedy*$(I) \leq$ *Cost_OPT*$(I) + \frac{|I|d}{2}$.

Proof. We assume that the facilities have initial capacity equal to l. Hence the number of customers $|I|$ is at most $2l$. In the worst case, the adversary places the first l customers on the middle of two facilities. Let Algorithm Greedy assigns the customers to f_1. The adversary places the next l customers on f_1. Cost_OPT(I) is equal to $\frac{ld}{2}$. Cost_Algorithm_Greedy(I) is equal to $\frac{3ld}{2} =$ Cost_OPT$(I) + \frac{|I|d}{2}$. □

We now introduce the idea of capacity sensitive approach. In this approach the cover area of a facility changes with its capacity.The algorithm is given below.

Algorithm Capacity-Sensitive-Greedy

Input: $F = \{f_1, f_2, \ldots, f_{|F|}\}$:facilities, l:capacity limit,
$I = \{c_1, c_2, \ldots, c_n\}$:input customers

Output: An assignment of the customers to the facilities and the total cost of that assignment

for $i \leftarrow 1$ **to** f **do**
  $capacity_i = l$;
let d be the distance between adjacent facilities;
for $i \leftarrow 1$ **to** f **do**
  $left\_cover\_area_i = \frac{d}{2}$;
  $right\_cover\_area_i = \frac{d}{2}$;
for $i \leftarrow 1$ **to** n **do**
  $index \leftarrow -1$;
  for $j \leftarrow 1$ **to** f **do**
    if $capacity_j > 0$ *and* c_i *appears in the cover area of* f_j **then**
      $index \leftarrow j$;
  assign c_i to f_{index};
  $capacity_{index} \leftarrow capacity_{index} - 1$;
  let f_l be the left adjacent free facility of f_{index};
  if $capacity_{index} > capacity_l$ **then**
    $left\_cover\_area_{index} = \frac{distance(f_{index}, f_l)}{2^{1+(capacity_{index} - capacity_l)}}$;
    $right\_cover\_area_l = distance(f_{index}, f_l) - left\_cover\_area_{index}$;
  else
    $right\_cover\_area_l = \frac{distance(f_{index}, f_l)}{2^{1+(capacity_l - capacity_{index})}}$;
    $left\_cover\_area_{index} = distance(f_{index}, f_l) - right\_cover\_area_l$;
  similarly update cover area of right adjacent facility;
  $sum \leftarrow sum + distance(f_{index}, c_i)$;

Result: An assignment of the customers having total cost equal to sum

We name this new approach Algorithm Capacity-Sensitive-Greedy. We use CSG to denote Algorithm Capacity-Sensitive-Greedy. The following lemma provides a upper bound for CSG.

Lemma 3. *Let $\mathcal{M}$ be a straight line and two facilities having distance d are situated on $\mathcal{M}$. Then $Cost\_CSG(I) \leq Cost\_OPT(I) + \frac{|I|d}{4}$.*

Proof. We assume that the facilities have initial capacity equal to l. Hence

the number of customers $|I|$ is at most $2l$. In the worst case, the adversary places a customer on the middle of two facilities. Let CSG assigns the customers to f_1. The adversary places the next customer on f_2. This process continues for the next $l-2$ customers. The adversary places the next l customers on f_1. Cost_OPT(I) is equal to $\frac{ld}{4}$. Cost_CSG(I) is equal to Cost_OPT(I) + $\frac{|I|d}{4}$. □

3.5 Conclusion

We have proposed some algorithms (Greedy, Randomized-Greedy, Optimal-Fill and Capacity-Sensitive-Greedy) for the facility assignment problem. It has been assumed that the facilities are situated on a straight line having equal distance between adjacent facilities. Exploring this problem in a more complex network would be interesting. Also analyzing different randomized approaches to solve this problem may lead to many prominent results.

Chapter 4

Facility Assignment on Connected Unweighted Graphs

We now study the facility assignment problem on connected unweighted graphs. A set of facilities F of equal capacity l is situated on the vertices of a connected unweighted graph. A set of customers arrive one by one having position on the vertices of the same graph. We assign a customer to a facility before a new customer arrives. In Section 4.1 we provide the graph terminologies used throughout this chapter. We show the competitive analysis of Algorithm Greedy and Algorithm Optimal-Fill in Section 4.2 and 4.3 respectively.

4.1 Graph Terminology

In this section we give necessary definitions and terminologies which will be used throughout the paper. A graph G is a tuple (V, E) which consists of a finite set V of vertices and a finite set E of edges; each edge is an unordered pair of vertices. We often denote the set of vertices G by $V(G)$

and the set of edges by $E(G)$. We say G is *unweighted* if every edge of G has equal weight. Let u and v be two vertices of G. If G has a u, v-path, then the distance from u to v is the length of a shortest u, v-path. The distance from u to v in G is denoted by $d_G(u, v)$ or simply by $d(u, v)$. If G has no u, v-path then $d(u, v) = \infty$. The *diameter* of G is the longest distance among the distances of all pair of vertices in G. The *eccentricity* of a vertex u in G is $max_{v \in V(G)} d(u, v)$ and denoted by $\epsilon(u)$. The *radius* r of G is $min_{u \in V(G)} \epsilon(u)$. The *center* of G is the subgraph of G induced by vertices of minimum eccentricity.

4.2 Competitive Analysis of Algorithm Greedy

In Section 3.1 we show competitive analysis of Algorithm Greedy when the metric space is a straight line. The following theorem determines the performance of Algorithm Greedy when the space is a graph.

Theorem 7. *Let $\mathcal{M}$ be a connected unweighted graph. Then $\mathcal{R}$(Algorithm Greedy) $\leq 2|E(\mathcal{M})|$.*

Proof. We assume that the facilities have unit capacity since the analysis is similar for capacity l, where $l > 1$. Two facilities f_i and f_j are *adjacent* if there exists a path P from f_i to f_j such that no other facilities are situated on P. We say an input I is *well distributed* if there is at least one customer between two adjacent facilities. We first prove the claim for an input I which is well distributed or we can transfer I to I' such that I' is well distributed and the competitive ratios of I and I' are same.

We can consider two cases; $\mathcal{M}$ is a tree and $\mathcal{M}$ contains at least one cycle. If $\mathcal{M}$ is a tree, we assume that every leaves contain a facility since $\mathcal{R}$(Algorithm Greedy) does not increase in other cases. In the worst case Cost_Greedy(I) is less than $2|E(\mathcal{M})|$ and Cost_OPT(I) is equal to one as

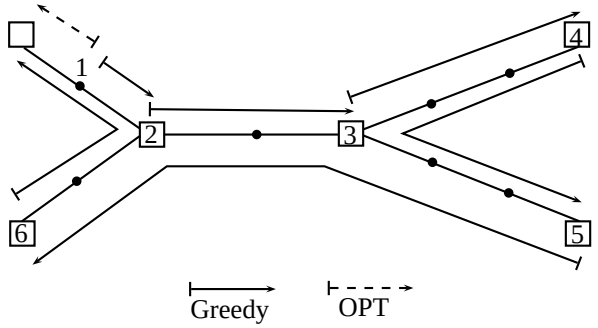

Figure 4.1: The configurations of Algorithm Greedy and OPT

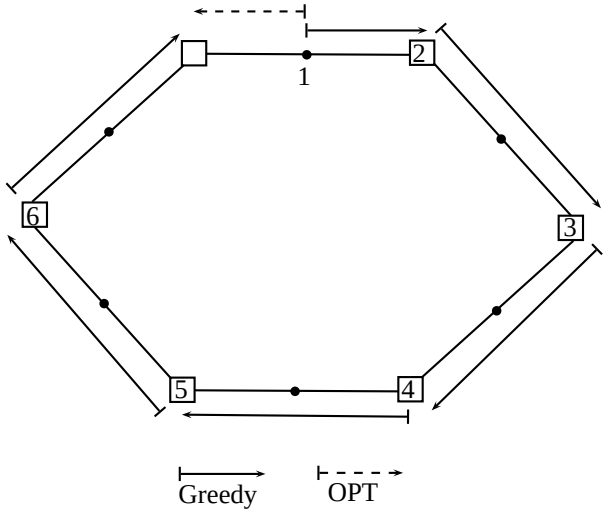

Figure 4.2: The configurations of Algorithm Greedy and OPT for a cycle

shown in Figure 4.1. A square box represents a facility and the input customers are shown by their sequence numbers. In this case competitive ratio is less than $2|E(\mathcal{M})|$.

If $\mathcal{M}$ contains a cycle, $\mathcal{R}$(Algorithm Greedy) does not increase. Consider a set of facilities F is situated on a cycle. In the worst case Cost_Greedy(I) is less than $|E(\mathcal{M})|$ and Cost_OPT(I) is equal to one as shown in Figure 4.2. In this case competitive ratio is less than $|E(\mathcal{M})|$.

We now assume that I is not an well distributed input. Let $\mathcal{M}'$ be the minimum subgraph of $\mathcal{M}$ such that all customers are situated on $\mathcal{M}'$. There is a set of facilities situated on $\mathcal{M}'$. In the worst case the customers assigned to those facilities by Algorithm Greedy incur total cost less than $2|E(\mathcal{M}')|$ and OPT incurs only unit cost. If OPT incurs x amount of cost to assign

a customer to a remaining facility, then Algorithm Greedy incurs at most $x + |E(\mathcal{M}')|$ amount of cost to assign a customer to that facility. Hence,

$$\text{Cost\_Greedy}(I) \leq \text{Cost\_OPT}(I) - 1 + |E(\mathcal{M}')|(|E(\mathcal{M})| - |E(\mathcal{M}')|) + 2|E(\mathcal{M}')|$$

According to this equation, if $|E(\mathcal{M}')|$ is small then Algorithm Greedy will perform similar to OPT. The larger the value of $|E(\mathcal{M}')|$ the more well distributed the input I becomes. Hence $\mathcal{R}$(Algorithm Greedy) is no more than $2|E(\mathcal{M})|$.

□

Theorem 7 immediately yields the following corollary.

Corollary 1. *Let $\mathcal{M}$ be a connected unweighted graph and a set of facilities F is situated on the vertices of $\mathcal{M}$ having equal distance between two adjacent facilities. Then $\mathcal{R}(\textit{Algorithm Greedy}) \leq 4|F|$.*

Proof. Let the distance between two adjacent facilities be d. In the worst case, each path between two adjacent facilities is visited no more than twice. Hence Cost_Greedy(I) is less than $2|F|d$. According to lemma 1 Cost_OPT(I) is greater than $d/2$. Hence $\mathcal{R}$(Algorithm Greedy) is no more than $4|F|$. □

4.3 Competitive Analysis of Algorithm Optimal-Fill

In Section 3.3 we show Algorithm Optimal-Fill is more efficient than Algorithm Greedy when the metric space is a straight line. However when the metric space is a connected unweighted graph, it is not straight forward to find whether Algorithm Optimal-Fill is better than Algorithm Greedy. In this case it depends on the number of edges, facilities and the radius of the graph. The following theorem determines the performance of Algorithm Optimal-Fill when the space is a graph.

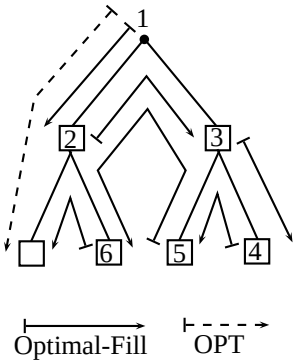

Figure 4.3: Worst case of Algorithm Optimal-Fill

Theorem 8. *Let $\mathcal{M}$ be a connected unweighted graph and a set of facilities F is situated on the vertices of $\mathcal{M}$. Then $\mathcal{R}$(Algorithm Optimal-Fill) $\leq \frac{|E(\mathcal{M})||F|}{r}$.*

Proof. The proof is similar to the analysis of theorem 7. It is sufficient to consider the case when $\mathcal{M}$ is a tree and I is well distributed. Let x be a vertex in the center of $\mathcal{M}$ which is not a facility. If no such vertex exists, the first customer c_1 is placed on a vertex which is not a facility and the distance from the center of $\mathcal{M}$ is minimum. Otherwise, c_1 is placed on x. In the worst case, Algorithm Optimal-Fill pays a cost equal to the distance between two facilities for each customers except the first one (see Figure 4.3). The adversary pays a cost which is no more than *radius* only for the first customer. Optimal-Fill traverses an edge no more than $|F|$ times. Hence, $\mathcal{R}$(Algorithm Optimal-Fill) is no more than $\frac{|E(\mathcal{M})||F|}{r}$.

□

4.4 Facility Assignment with Finite Service Time

Until now we have considered that if a customer c_w is assigned to a facility, then c_w stays there forever. In other words, service time of an assignment is infinite. Hence a facility with capacity l can provide service to at most l customers. If there are f facilities, total number of customers is limited

to fl. In this section we study the facility assignment problem with a finite service time t. It can be assumed that a customer arrives after one unit time of its previous customer. When $t = 1$, service time is unit. Let c_w be assigned to f_x in this scenario. Consider that a facility center can give service to only one customer ($l = 1$). If c_y is next to c_w then we can also assign c_y to f_x although c_w was assigned to f_x. For unit service time, both Algorithm Greedy and Algorithm Optimal-Fill provide optimal solution. When service time is two ($t = 2$), we can not assign c_y to f_x. However if c_z arrives just after c_y, then we can assign c_z to f_x. The following theorem demonstrates the scenario when $t = 2$.

Theorem 9. *Let t be the time needed to provide service to a assigned customer. Then no deterministic algorithm ALG is competitive for $t = 2$.*

Proof. Let $I = (c_1, c_2, \ldots, c_n)$ be the input sequence. The adversary places the first customer c_1 between any two adjacent facilities f_i and f_{i+1}. Suppose ALG has assigned c_1 to f_i. The adversary now places $c_2, c_3, \ldots, c_n$ exactly on the facilities assigned for $c_1, c_2, \ldots, c_{n-1}$. The adversary runs the optimal algorithm. It assigns c_1 to f_{i+1}, which incurs cost less than x, the distance between f_i and f_{i+1}. The adversary does not pay any cost for the later assignments, because each customer is placed exactly on a facility. However ALG pays x for each assignment except the first one. □

4.5 Conclusion

We have studied the online facility assignment problem on connected unweighted graphs and explored two algorithms (Greedy, Optimal-Fill) for this problem. We also introduced service time parameter to that problem. It would be an interesting problem to find out other deterministic and randomized algorithms which can perform better than the given methods.

Chapter 5

Experimental Competitive Ratio

In this chapter we perform an experiment to understand the performance of different algorithms of online facility assignment in practical scenarios.

5.1 Experimental Setup

In order to perform the simulation we take a graph G of 100 vertices. The structure of G varies for different algorithms. For Algorithm Greedy and Algorithm Optimal-Fill G is a connected unweighted graph. The edges of G are generated randomly such that G remains connected. For Algorithm Randomized-Greedy G is a path. For both cases the number of vertices of G is 100. We take n facilities situated on the vertices of G. The vertices are selected randomly. Similarly we also generate n customers located on the vertices of G. After generating the facilities and the customers we run all algorithms (Greedy, Randomized-Greedy and Optimal-Fill). We also run the optimal algorithm. When G is not a path, in optimal algorithm we generate

all possible assignment of the customers to the facilities. The assignment cost is equal to the distance between the customer and the facility on which the customer has been assigned. We take the assignment which have minimum total assignment cost. This is the optimal assignment. When G is a path, we assign the customers to the facilities in monotonic order to get the optimal assignment [4]. We take the ratio of the assignment cost of online algorithms (Greedy, Randomized-Greedy and Optimal-Fill) and the assignment cost of optimal assignment. We call these values *experimental competitive ratios*. After that we again generate the positions of facilities and the customers randomly and determine the experimental competitive ratios. We continue this process for 20 times. We take the average and the maximum value of the experimental competitive ratios. We vary the value of n from 1 to 15 and analyse the data for each algorithms. This process is briefly demonstrated in Figure 5.1.

5.2 Simulation Results

In this section we study the simulation result to understand the performance of different algorithms of online facility assignment. The experimental competitive ratios of all algorithms are shown in Table 5.1. We vary the value of n (number of facilities) and draw graphs to see the relationship between the experimental competitive ratios with the number of facilities in the following sections.

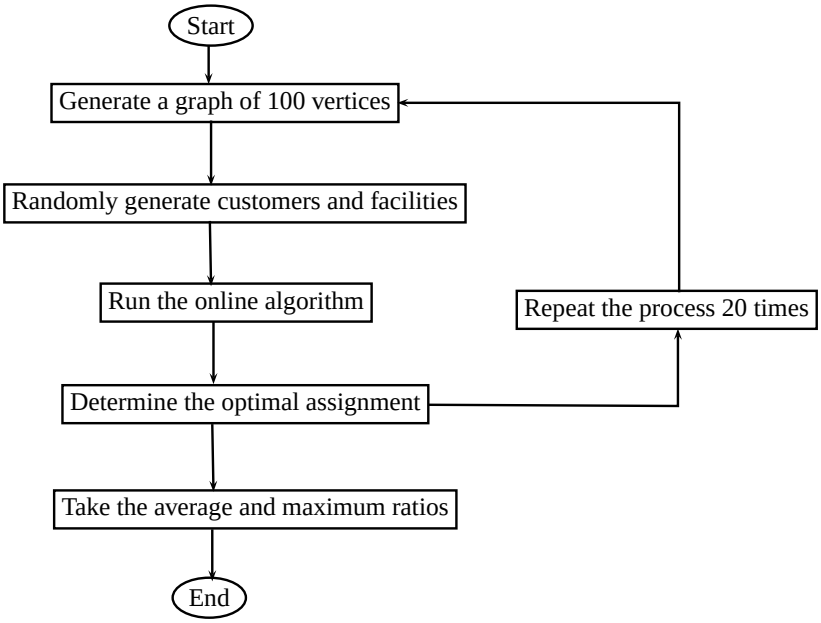

Figure 5.1: The flow of simulation

Facilities	Greedy		Randomized-Greedy		Optimal-Fill	
	Average	Maximum	Average	Maximum	Average	Maximum
1	1.00	1.00	1.00	1.00	1.00	1.00
2	1.02	1.44	1.22	3.32	1.02	1.44
3	1.03	1.29	1.12	1.67	1.03	1.29
4	1.19	2.20	1.46	2.87	1.19	2.20
5	1.13	1.89	1.29	2.87	1.15	1.89
6	1.11	1.76	1.50	3.63	1.16	1.90
7	1.18	1.45	1.62	3.96	1.18	1.45
8	1.18	1.84	1.34	3.75	1.16	1.74
9	1.13	1.42	1.38	2.29	1.15	1.56
10	1.17	1.96	1.55	3.72	1.13	1.46
11	1.26	2.01	1.46	2.29	1.21	1.45
12	1.29	2.20	1.60	3.92	1.30	1.82
13	1.23	1.89	1.76	3.69	1.26	2.11
14	1.21	1.48	1.66	3.39	1.25	1.81
15	1.18	1.62	1.64	3.72	1.32	2.33

Table 5.1: Experimental Competitive Ratios

5.2.1 Performance of Algorithm Greedy

Algorithm Greedy is the simplest approach among other algorithms. It assigns a customer to the nearest free facility. The performance of Algorithm Greedy is shown in Figure 5.2. The average ratio increases a little bit as the number of facilities increases. According to Theorem 7 the maximum ratio should increase linearly as the number of facilities increases. In the worst case the graph is a tree and all the facilities are situated in the leaves of that tree. Also the customers must appear in a particular order. However the graph, the positions of facilities and customers are generated randomly and there is no guarantee that the worst case is generated. Hence the experimental competitive ratios deviate from theoretical results.

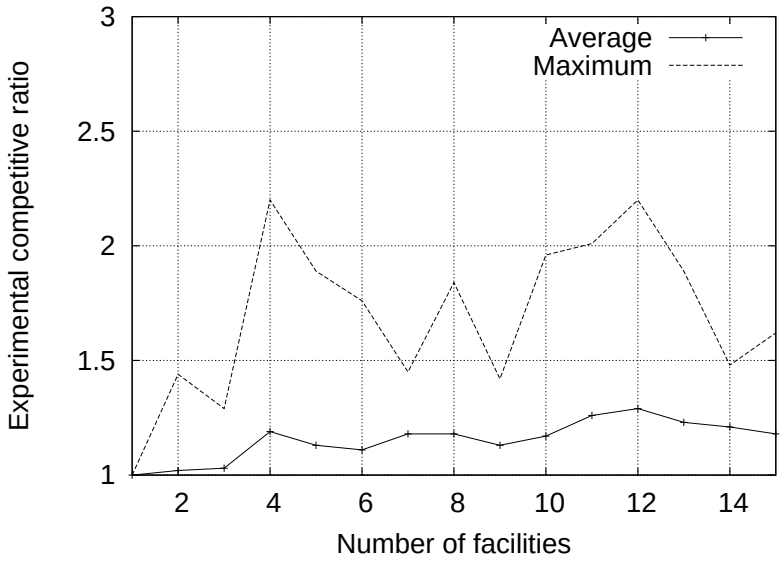

Figure 5.2: Algorithm Greedy

5.2.2 Performance of Algorithm Randomized-Greedy

We introduce randomness in the Algorithm Greedy and call the new method Algorithm σ-Randomized-Greedy. Let f_x be a facility which is nearest to the the customer c_y and σ is a real number. σ-Randomized-Greedy checks whether distance between c_y and f_x is less than σ or not. If it is less than σ, c_y is assigned to f_x. Otherwise σ-Randomized-Greedy tosses a fair coin before assigning a customer to a facility. It chooses the nearest free facility at right (left) side when head (tail) appears. In this simulation we assume σ is equal to one and call it Algorithm Randomized-Greedy. The performance of Algorithm Randomized-Greedy is shown in Figure 5.3. Both the average and maximum ratios are relatively larger than the ratios of Algorithm Greedy. Note that the theoretical result of Algorithm Randomized-Greedy in Theorem 5 assumes that the input of customers has a special characteristics. However the simulation generates the input sequence randomly. Hence the ratio is different from theoretical result.

5.2.3 Performance of Algorithm Optimal-Fill

In Algorithm Optimal-Fill when a new customer c_i is placed, Algorithm Optimal-Fill finds out the new facility f_j that would be selected by an optimal assignment of the customers $c_1, c_2, \ldots, c_i$. Algorithm Optimal-Fill then assigns c_i to f_j. The performance of Algorithm Optimal-Fill is shown in Figure 5.4. Practically it performs similar to Algorithm Greedy.

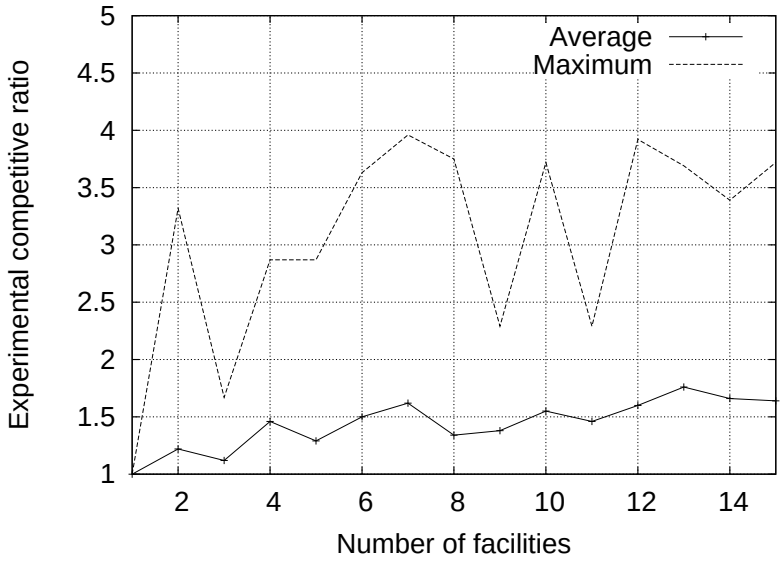

Figure 5.3: Algorithm Randomized-Greedy

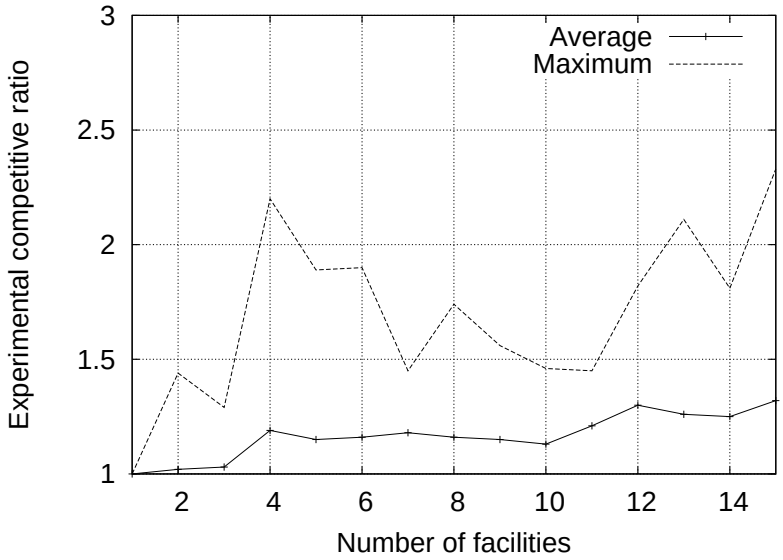

Figure 5.4: Algorithm Optimal-Fill

5.2.4 Comparison

We have plotted all the curves of average experimental competitive ratio in Figure 5.5. The performance of Algorithm Greedy and Algorithm Optimal-

Fill are similar. The perfermonce of Algorithm Randomized-Greedy is worst. In Theorem 5 we have shown that Algorithm Randomized-Greedy performs very well when the input is distributed in the whole metric space. However from the experimental results it seems that, Algorithm Randomized-Greedy does not perform well in all cases.

Figure 5.5: Comparison

5.3 Facility Assignment with Preference

Till now we have considered that all facilities are same which is similar to the assumption of the theoretical results of previous chapters. We have defined the assignment cost as the distance between the customer and facility and it does not depend on any other criteria. In this section we consider that the customers have a preference list of facilities. The assignment cost depends both on the distance and the preference of the customer. We conduct

Facilities	Greedy		Randomized-Greedy		Optimal-Fill	
	Average	Maximum	Average	Maximum	Average	Maximum
1	1.00	1.00	1.00	1.00	1.00	1.00
2	1.02	1.45	1.15	3.00	1.02	1.45
3	1.05	1.67	1.10	1.62	1.06	1.66
4	1.18	1.57	1.32	4.61	1.20	1.57
5	1.11	1.32	1.29	1.87	1.15	1.49
6	1.15	1.64	1.46	2.22	1.16	1.45
7	1.15	1.55	1.36	2.67	1.19	1.55
8	1.16	1.50	1.43	2.01	1.26	1.76
9	1.17	1.40	1.41	2.05	1.26	1.53
10	1.25	1.48	1.68	2.14	1.32	1.48
11	1.22	1.52	1.59	2.46	1.32	1.69
12	1.18	1.45	1.73	2.83	1.32	1.54
13	1.21	1.50	1.67	2.34	1.39	1.96
14	1.19	1.38	1.69	2.34	1.32	1.63
15	1.23	1.42	1.81	2.57	1.45	1.78

Table 5.2: Experimental Competitive Ratios

a simulation to provide the performance of different algorithms in this circumstance. The experimental competitive ratios of all algorithms are shown in Table 5.2.

5.3.1 Performance of Algorithm Greedy

The algorithm is similar to Algorithm Greedy. However the definition of distance has been changed. Previously the distance between a customer and a facility was defined as the euclidean distance between them. Now the distance depends on both the euclidean distance and the preference of customer. The preference of customer is generated randomly. The performance of Algorithm Greedy is shown in Figure 5.6.

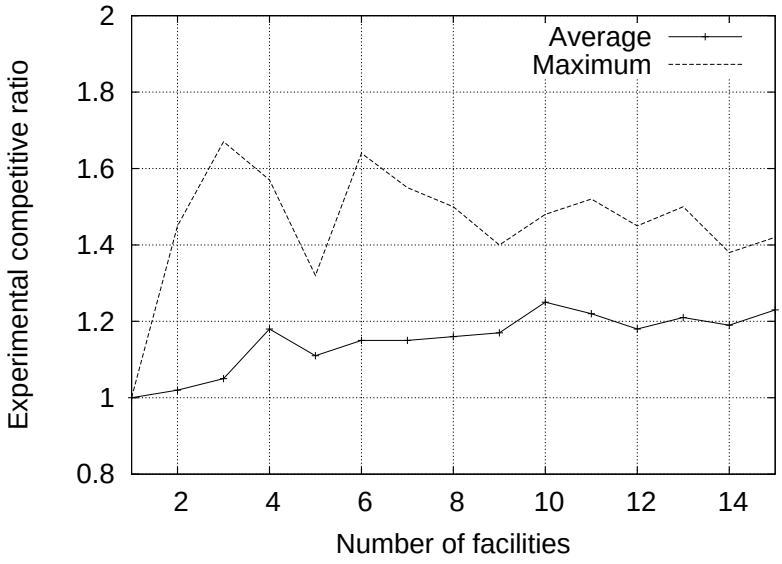

Figure 5.6: Algorithm Greedy with preference

5.3.2 Performance of Algorithm Randomized-Greedy

The algorithm is similar to Algorithm σ-Randomized-Greedy except the definition of distance. In this simulation we assume σ is equal to one and call it Algorithm Randomized-Greedy. The performance of Algorithm Randomized-Greedy is shown in Figure 5.7. Both the average and maximum ratios are relatively larger than the ratios of Algorithm Greedy.

5.3.3 Performance of Algorithm Optimal-Fill

The performance of Algorithm Optimal-Fill is shown in Figure 5.8. Practically it performs similar to Algorithm Greedy.

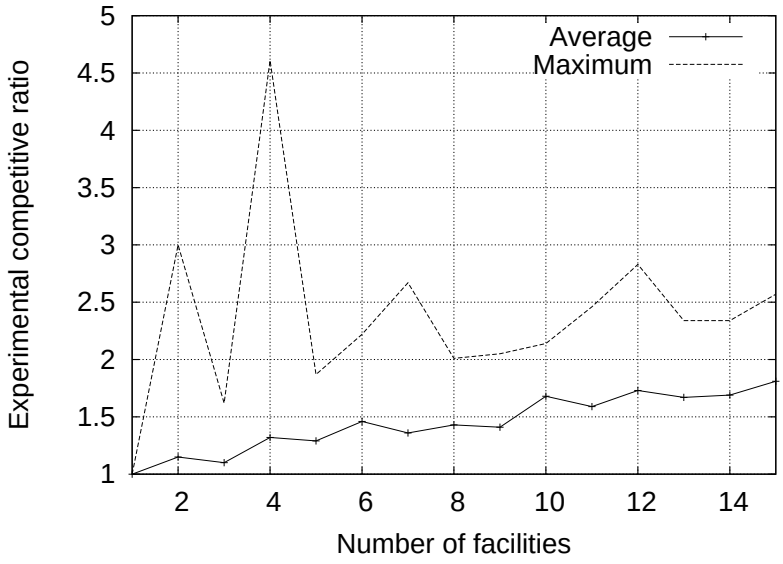

Figure 5.7: Algorithm Randomized-Greedy with preference

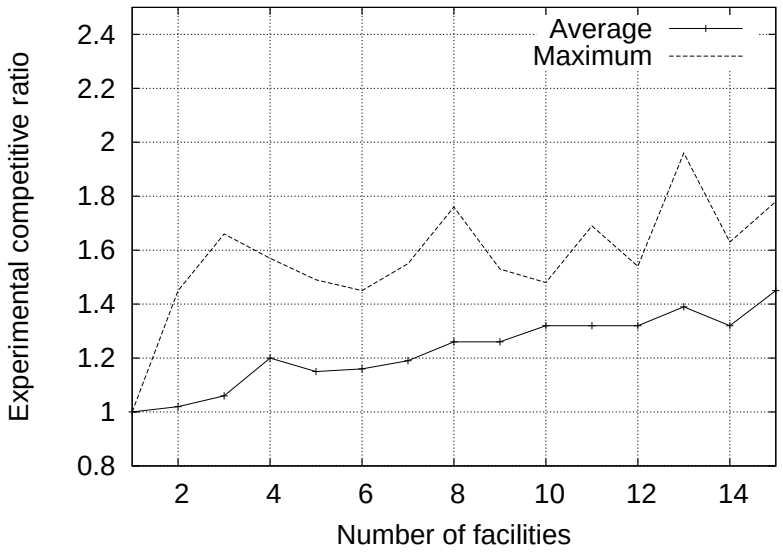

Figure 5.8: Algorithm Optimal-Fill with preference

5.3.4 Comparison

We have plotted all the curves of average experimental competitive ratio in Figure 5.9. The performance of Algorithm Greedy and Algorithm Optimal-

Fill are similar. However the ratio of Algorithm Greedy is smallest. The perfermonce of Algorithm Randomized-Greedy is worst.

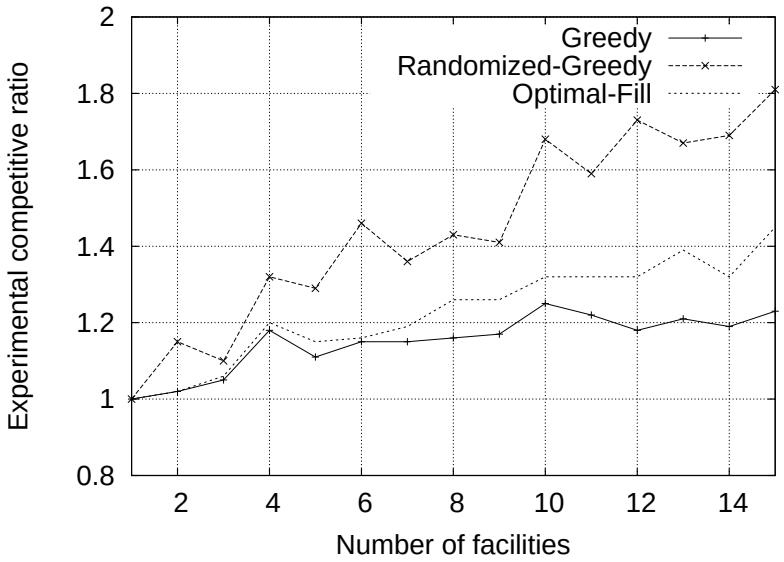

Figure 5.9: Comparison

5.4 Conclusion

In this chapter we have conducted a simulation to see the practical performance of different algorithms of online facility assignment problem. We have studied the theoretical results in Chapter 3 and Chapter 4. Most of the theoretical results consider the worst case performances. We have generated the metric spaces, customers and facilities randomly. In this situation the probability of the worst case scenario is very low. Hence the practical performances are better than the worst cases found in theoretical results. According to the theoretical results the competitive ratio increases as the number of facility increases. However this in not necessarily true for an experiment where the

scenarios are generated randomly. Hence we have found the curves as zigzag lines. We have also considered that a customer has a preference list of facilities. From the results of the simulation it seems that the experimental competitive ratio does not change significantly with this modification in the problem. The code of this simulation is provided in the appendix.

Chapter 6

Conclusion

In this thesis we have studied an online facility assignment problem. We have given some algorithms for this problem and provided their competitive analysis. We have also conducted a simulation of these algorithms. We now briefly describe the content of each chapter of this thesis.

In Chapter 1 we have characterized online algorithms in the context of the sorting problem. We have provided some applications of online algorithms like paging in a virtual memory system, load balancing in parallel processing and routing in communications networks. We have defined the facility assignment problem, which we have studied in this thesis. We have presented its motivation and the objective of this thesis. We have introduced the similar problems found in the literature. Finally we have provided the results of this thesis and its organization.

In Chapter 2 we have studied the competitive analysis techniques in the context of the k-server problem. The competitive analysis can be viewed as a game between an online player and a malicious adversary. The adver-

sary tries to make the task costly to the online player but, at the same time, inexpensive for the optimal offline algorithm. There are different kinds of adversaries when the online player uses randomized algorithm. We have studied these models of adversaries. We have studied the proofs of determining competitive ratio of some k-server algorithms demonstrating the basic methods like averaging technique, potential function which are utilized in many other proofs of different online problems.

In Chapter 3 we have given some algorithms (Greedy, Randomized-Greedy, Optimal-Fill and Capacity-Sensitive-Greedy) for the facility assignment problem. It has been assumed that the facilities are situated on a straight line having equal distance between adjacent facilities. We have showed that Algorithm Greedy is $4|F|$-competitive and Algorithm Optimal-Fill is $|F|$-competitive. Kao et al. [26] have provided a randomized lower bound of 4.5911 for the online matching on a line problem. We have proved that Algorithm Randomized-Greedy is $\frac{9}{2}$-competitive for a class of input sequences. We have also provided another method named Algorithm Capacity-Sensitive-Greedy.

In Chapter 4 we have analyzed the online facility assignment problem on connected unweighted graphs. We have proved Algorithm Greedy is $2|E(G)|$-competitive and Algorithm Optimal-Fill is $|E(G)||F|/radius$-competitive. We have also introduced service time parameter t in our modeling and show that no deterministic algorithm is competitive when $t = 2$.

Finally in Chapter 5 we have conducted a simulation to see the practical

performance of different algorithms of online facility assignment problem. We have also considered that a customer has a preference list of facilities. From the results of the simulation it seems that the experimental competitive ratio does not change significantly with this modification in the problem. The code of this simulation is provided in the appendix.

We now present some open problems and future research scopes related to this thesis.

1. In Theorem 4 we have shown that Algorithm Greedy is $4|F|$-competitive when the input I is well distributed. In Theorem 5 we have proved that Algorithm Randomized-Greedy is $\frac{9}{2}$-competitive when I is well distributed. However the competitive ratio for input sequences which are not well distributed is not known. It is an interesting problem to determine the competitive ratio of Algorithm Randomized-Greedy for general input sequences.

2. We have provided the competitive analysis of Algorithm Randomized-Greedy when the metric space is a line. It would be interesting to analyse it on a more complex metric space like graph.

3. In Section 3.4 we have provided Algorithm Capacity-Sensitive-Greedy and shown a relation between the cost of this algorithm and the optimal algorithm. Determining the competitiveness of Algorithm Capacity-Sensitive-Greedy is an open problem.

4. In this thesis we have considered that we can always assign a customer

to a facility when it is free. We can introduce a probability of the availability of each facility. It would be really interesting to analyse the facility assignment problem with uncertainty.

5. We can analyse the facility assignment problem with an objective to maximize the total assignment cost.

6. We can analyse the facility assignment problem when the metric space is a weighted graph.

Bibliography

[1] Armon, A., "On min-max r-gatherings", Theoretical Computer Science, Vol. 412, No. 7, pp. 573-582, 2011.

[2] Antoniadis, A., Barcelo, N., Nugent, M., Pruhs, K. and Scquizzato, M. "A $o(n)$-competitive deterministic algorithm for online matching on a line", Proc. of Approximation and Online Algorithms: 12th International Workshop (WAOA 2014), Lecture Notes in Computer Science, Vol. 8952, pp. 11-22, 2015.

[3] Anagnostopoulos, A., Bent, R., Upfal, E., and Hentenryck P. V., "A simple and deterministic competitive algorithm for online facility location", Information and Computation, Vol. 194, No. 2, pp. 175-202, 2004.

[4] Akagi, T., and Nakano, S., "On r-gatherings on the line", Proc. of Frontiers in Algorithmics Workshop (FAW 2015), Lecture Notes in Computer Science, Vol. 9130, Springer, pp. 25-32, 2015.

[5] Bansal, N., Buchbinder, N., Gupta, A., and Naor, J. S., "An $O(\log^2 k)$-competitive algorithm for metric bipartite matching", Algorithmica, Vol. 68, No. 2, pp. 390-403, 2012.

[6] Bein, W. W., Chrobak, M., and Larmore, L. L., "The 3-server problem in the plane", Theoretical Computer Science, Vol. 289, No. 1, pp. 335-354, 2002.

[7] Borodin, A., and El-Yaniv, R., "Online Computation and Competitive Analysis", Cambridge University Press, 1998.

[8] Bartal, Y., and Koutsoupias, E., "On the competitive ratio of the work function algorithm for the k-server problem", Theoretical Computer Science, Vol. 324, No. 23, pp. 337-345, 2004.

[9] Chudak, F. A., "Improved algorithms for uncapacitated facility location problem", Proc. of the 10th ACM-SIAM Symposium on Discrete Algorithms, Lecture Notes in Computer Science, Vol. 1412, pp. 180-194, 1998.

[10] Charikar, M., and Guha, S.,"Improved combinatorial algorithms for the facility location and k-median problems", Proc. of the 40th Annual IEEE Symposium on Foundations of Computer Science, pp. 378-388, 1999.

[11] Charikar, M., Guha, S., Tardos, É., and Shmoys, D. B., "A constant-factor approximation algorithm for the k-median problem", Journal of Computer and System Sciences, Vol. 65, No. 1, pp. 129-149, 2002.

[12] Chrobak, M., Karloff, H., Payne, T., and Vishwanathan, S., "New results on server problems", SIAM Journal on Discrete Mathematics, Vol. 4, No. 2, pp. 172-181, 1991.

[13] Chrobak, M., and Larmore, L. L., "An optimal on-line algorithm for k-servers on trees", SIAM Journal on Computing, Vol. 20, No. 1, pp. 144-148, 1991.

[14] Chudak, F. A., and Shmoys., D. B., "Improved approximation algorithms for capacitated facility location problem", Mathematical Programming, Vol. 102, No. 2, pp. 207-222, 2005.

[15] Drezner, Z., "Facility Location: A Survey of Applications and Methods", Springer, 1995.

[16] Drezner, Z., and Hamacher, H. W., "Facility Location: Applications and Theory", Springer, 2006.

[17] Fotakis, D., "On the competitive ratio for online facility location", Algorithmica, Vol. 50, No. 1, Springer, pp. 1-57, 2008.

[18] Guha, S., and Khuller, S., "Greedy strikes back: improved facility location algorithms", Journal of Algorithms, Vol. 31, No. 1, pp. 228-248, 1999.

[19] Jain, K., and Vazirani, V. V., "Approximation algorithms for metric facility location and k-median problems using the primal-dual schema and lagrangian relaxation", Journal of the ACM, Vol. 48, No. 2, pp. 274-296, 2001.

[20] Karp, R. M., "A 2k-competitive algorithm for the circle", 1989. Unpublished manuscript.

[21] Kleinberg, J. M., "A lower bound for two-server balancing algorithms", Information Processing Letters, Vol. 52, No. 1, pp. 39-43, 1994.

[22] Khuller, S., and Mitchell, S. G., and Vazirani, V. V., "On-line algorithms for weighted bipartite matching and stable marriages", Theoretical Computer Science, Vol. 127, No. 2, pp. 255-267, 1994.

[23] Kalyanasundaram, B., and Pruhs, K., "Online weighted matching", Journal of Algorithms, Vol. 14, No. 3, pp. 478-488, 1993.

[24] Koutsoupias, E. and Papadimitriou, C., "The 2-evader problem", Information Processing Letters, Vol. 57, No. 5, pp. 249-252, 1996.

[25] Korupolu, M. R., Plaxton, C. G., and Rajaraman R., "Analysis of a local search heuristic for facility location problems", Journal of Algorithms, Vol. 37, No. 1, pp. 146-188, 2000.

[26] Kao, M., Reif, J. H., and Tate, S. R., "Searching in an unknown environment: an optimal randomized algorithm for the cow-path problem", Information and Computation, Vol. 131, No. 1, pp. 63-79, 1996.

[27] Meyerson, A., "Online facility location", Proc. of the 42nd IEEE symposium on Foundations of Computer Science (FOCS '01), IEEE Computer Society, pp. 426-431, 2001.

[28] Manasse, M. S., McGeoch, L. A., and Sleator, D. D., "Competitive algorithms for server problems", Journal of Algorithms, Vol. 11, No. 2, pp. 208-230, 1990.

[29] Rosen, K. H., "Discrete Mathematics and Its Applications", McGraw-Hill, 2012.

[30] Schrijver, A., "Combinatorial Optimization: Polyhedra and Efficiency", Springer, 2003.

[31] Sviridenko, M.,"An improved approximation algorithm for the metric uncapacitated facility location problem", Integer Programming and Combinatorial Optimization, Lecture Notes in Computer Science, Vol. 2337, Springer, pp. 240-257, 2002.

[32] Sleator, D. D., and Tarjan, R. E., "Amortized efficiency of list update and paging rules", Communications of the ACM, Vol. 28, No. 2, pp. 202-208, 1985.

[33] Shmoys, D. B., Tardos, É., and Aardal, K.,"Approximation algorithms for facility location problems", Proc. of the twenty-ninth annual ACM symposium on Theory of computing (STOC '97), pp. 265-274, 1997.

Appendix A

Code of Simulation

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_CAPACITY 1
#define STRAIGHT_LINE 1
#define GRAPH 2
#define MAX_VERTICES 200
#define EXTRA 10
#define INFINITY 1000000
#define EDGES_PER_STEP 10
int adjacency_matrix[MAX_VERTICES + EXTRA][MAX_VERTICES + EXTRA];
int number_of_vertices;
int number_of_facilities;
int capacity;
int number_of_customers;
int facilities[MAX_VERTICES + EXTRA];
int sorted_facilities[MAX_VERTICES + EXTRA];
int capacities[MAX_VERTICES + EXTRA];
int last_capacities[MAX_VERTICES + EXTRA];
int marker[MAX_VERTICES + EXTRA];
int customers[MAX_VERTICES * MAX_CAPACITY + EXTRA];
int assigned_facilities[MAX_VERTICES * MAX_CAPACITY + EXTRA];
int minimum_assignment[MAX_VERTICES * MAX_CAPACITY + EXTRA];
int brute_force_cost;
int minimum_brute_force_cost;
int sorted_customers[MAX_VERTICES * MAX_CAPACITY + EXTRA];
int distance[MAX_VERTICES + EXTRA][MAX_VERTICES + EXTRA];
```

```
int total_cost_of_greedy;
int total_cost_of_optimal;
int total_cost_of_optimal_fill;
int total_cost_of_randomized_greedy;
FILE * output_file;
void clean_marker()
{
 int i;
 for(i=0;i<number_of_vertices;i++)
 {
  marker[i]=0;
 }
}
void clean_previous_data()
{
 int i;
 for(i=0;i<number_of_vertices;i++)
 {
  int j;
  for(j=0;j<number_of_vertices;j++)
  {
   adjacency_matrix[i][j]=0;
   distance[i][j]=0;
  }
 }
 clean_marker();
 total_cost_of_greedy = 0;
 total_cost_of_optimal = 0;
 total_cost_of_optimal_fill = 0;
 total_cost_of_randomized_greedy = 0;
 number_of_customers = 0;
 capacity = 0;
 number_of_facilities = 0;
 number_of_vertices = 0;
}
void depth_first_search(int v)
{
 int i;
 for(i=0;i<number_of_vertices;i++)
 {
  if(adjacency_matrix[v][i])
```

```
  {
   if(!marker[i])
   {
    marker[i] = 1;
    depth_first_search(i);
   }
  }
 }
}
int is_connected()
{
 depth_first_search(0);
 //is all visited?
 int count = 0;
 int i;
 for(i=0;i<number_of_vertices;i++)
 {
  if(marker[i])count++;
 }
 clean_marker();
 if(count == number_of_vertices)return 1;
 else return 0;
}
void generate_graph()
{
 do{
  int edges = EDGES_PER_STEP;
  while(edges--)
  {
   int generated = 0;
   while(!generated)
   {
    int u = rand() % number_of_vertices;
    int v = rand() % number_of_vertices;
    while(v==u)
    {
     v = rand() % number_of_vertices;
    }
    if(!adjacency_matrix[u][v])
    {
     adjacency_matrix[u][v] = 1;
```

```
      adjacency_matrix[v][u] = 1;
      generated = 1;
     }
    }
    if(is_connected())break;
   }
  }while(!is_connected());
}
void generate_path()
{
 int i;
 for(i=1;i<number_of_vertices;i++)
 {
  adjacency_matrix[i][i-1] = 1;
  adjacency_matrix[i-1][i] = 1;
 }
}
void randomly_generate_facilities()
{
 printf("facilities:");
 fprintf(output_file, "facilities:");
 int i;
 for(i=0;i<number_of_facilities;i++)
 {
  int next_facility;
  do{
   next_facility = rand() % number_of_vertices;
  }while(marker[next_facility]);
  facilities[i] = next_facility;
  marker[next_facility]=1;
  printf(" %d", next_facility);
  fprintf(output_file, " %d", next_facility);
 }
 printf("\n");
 fprintf(output_file, "\n");
 clean_marker();
}
void generate_customers()
{
 printf("customers:");
 fprintf(output_file, "customers:");
```

```
 int i;
 for(i=0;i<number_of_customers;i++)
 {
  customers[i] = rand() % number_of_vertices;
  printf(" %d", customers[i]);
  fprintf(output_file, " %d", customers[i]);
 }
 printf("\n");
 fprintf(output_file, "\n");
}
void find_all_pair_shortest_path()
{
 int i;
 for(i=0;i<number_of_vertices;i++)
 {
  int j;
  for(j=0;j<number_of_vertices;j++)
  {
   if(adjacency_matrix[i][j])distance[i][j] = 1;
   else distance[i][j] = INFINITY;
  }
 }
 for(i=0;i<number_of_vertices;i++)
 {
  int j;
  for(j=0;j<number_of_vertices;j++)
  {
   int k;
   for(k=0;k<number_of_vertices;k++)
   {
    if((distance[j][i] + distance[i][k]) < distance[j][k])
     distance[j][k] = distance[j][i] + distance[i][k];
   }
  }
 }
 /*for(i=0;i<number_of_vertices;i++)
 {
  int j;
  for(j=0;j<number_of_vertices;j++)
  {
   printf("%d ",distance[i][j] );
```

```
   }
   printf("\n");
  }*/
  for(i=0;i<number_of_vertices;i++)
  {
   distance[i][i]=0;
  }
}
void algorithm_greedy()
{
 int i;
 for(i=0;i<number_of_vertices;i++)
 {
  capacities[i]=capacity;
 }
 for(i=0;i<number_of_customers;i++)
 {
  //find minimum cost
  int min = INFINITY;
  int f;
  int j;
  for(j=0;j<number_of_facilities;j++)
  {
   if(capacities[facilities[j]]>0)
   {
    if(distance[customers[i]][facilities[j]] < min)
    {
     min = distance[customers[i]][facilities[j]];
     f=facilities[j];
    }
   }
  }
  total_cost_of_greedy += min;
  capacities[f]--;
  //printf("customer %d assigned to facility %d with cost %d\n",
  customers[i],f,min );
 }
}
void algorithm_randomized_greedy(int sigma)
{
 int i,j,k;
```

```
int sorted_facilities[MAX_VERTICES + EXTRA];
for(i=0;i<number_of_facilities;i++)
{
 sorted_facilities[i] = facilities[i];
}
// sort the facilities
for(j=1;j<number_of_facilities;j++)
{
 i=0;
 while(sorted_facilities[j] > sorted_facilities[i])
  i++;
 long int temp = sorted_facilities[j];
 for(k=0; k<=(j-i-1); k++)
  sorted_facilities[j-k] = sorted_facilities[j-k-1];
 sorted_facilities[i] = temp;
}

for(i=0;i<number_of_vertices;i++)
{
 capacities[i]=capacity;
}
for(i=0;i<number_of_customers;i++)
{
 int assigned = 0;
 int min = INFINITY;
 int f=-1;
 for(k=0;k<number_of_facilities;k++)
 {
  if((capacities[sorted_facilities[k]]>0)&&
  (distance[customers[i]][sorted_facilities[k]] <= sigma))
  {
   if(distance[customers[i]][sorted_facilities[k]] < min)
   {
    f = sorted_facilities[k];
    min = distance[customers[i]][sorted_facilities[k]];
   }
  }
 }
 if(f != -1)
 {
  assigned = 1;
```

```
 capacities[f]--;
 total_cost_of_randomized_greedy += min;
}
if(!assigned)
{
 for(j=0;j<number_of_facilities;j++)
 {
  if(customers[i] <= sorted_facilities[j])
   break;
 }
 if((rand()%2)==1)
 {
  for(k=j;k<number_of_facilities;k++)
  {
   if(capacities[sorted_facilities[k]]>0)
   {
    assigned = 1;
    capacities[sorted_facilities[k]]--;
    total_cost_of_randomized_greedy +=
    distance[customers[i]][sorted_facilities[k]];
    break;
   }
  }
 }
 else
 {
  for(k=j-1;k>=0;k--)
  {
   if(capacities[sorted_facilities[k]]>0)
   {
    assigned = 1;
    capacities[sorted_facilities[k]]--;
    total_cost_of_randomized_greedy +=
    distance[customers[i]][sorted_facilities[k]];
    break;
   }
  }
 }

 if(!assigned)
 {
```

```
      min = INFINITY;
      f = -1;
      for(k=0;k<number_of_facilities;k++)
      {
       if(capacities[sorted_facilities[k]]>0)
       {
        if(distance[customers[i]][sorted_facilities[k]] < min)
        {
         f = sorted_facilities[k];
         min = distance[customers[i]][sorted_facilities[k]];
        }
       }
      }
      capacities[f]--;
      total_cost_of_randomized_greedy += min;
     }
    }
   }
}
void optimal_algorithm_for_path()
{
 int i;
 for(i=0;i<number_of_customers;i++)
 {
  sorted_customers[i] = customers[i];
 }
 // sort the customers
 int j,k;
 for(j=1;j<number_of_customers;j++)
 {
  i=0;
  while(sorted_customers[j] > sorted_customers[i])
   i++;
  int temp = sorted_customers[j];
  for(k=0; k<=(j-i-1); k++)
   sorted_customers[j-k] = sorted_customers[j-k-1];
  sorted_customers[i] = temp;
 }


 for(i=0;i<number_of_facilities;i++)
```

```
 {
  sorted_facilities[i] = facilities[i];
 }
 // sort the facilities
 for(j=1;j<number_of_facilities;j++)
 {
  i=0;
  while(sorted_facilities[j] > sorted_facilities[i])
   i++;
  int temp = sorted_facilities[j];
  for(k=0; k<=(j-i-1); k++)
   sorted_facilities[j-k] = sorted_facilities[j-k-1];
  sorted_facilities[i] = temp;
 }



 for(i=0;i<number_of_vertices;i++)
 {
  capacities[i]=capacity;
 }
 for(i=0;i<number_of_customers;i++)
 {
  for(j=0;j<number_of_facilities;j++)
  {
   if(capacities[sorted_facilities[j]]>0)
   {
    total_cost_of_optimal +=
    distance[sorted_customers[i]][sorted_facilities[j]];
    capacities[sorted_facilities[j]]--;
    break;
   }
  }
 }
}
void brute_force(int nCustomers, int ci)
{
 //printf("inside burute force\n");
 //if all customer assigned update minimum if needed and return
 if(ci == nCustomers)
 {
```

```
  //printf("inside end condition\n");
  if(brute_force_cost < minimum_brute_force_cost)
  {
   int i;
   for(i=0;i<number_of_customers;i++)
   {
    minimum_assignment[i] = assigned_facilities[i];
   }
   //printf("last_capacities:");
   for(i=0;i<number_of_vertices;i++)
   {
    last_capacities[i] = capacities[i];
    //printf(" %d",last_capacities[i] );
   }
   //printf("\n");
   minimum_brute_force_cost = brute_force_cost;
  }
  return;
 }
 //for all available facility f
 int i;
 for(i=0;i<number_of_facilities;i++)
 {
  //printf("recursively\n");
  //assign the first unassigned customer to f
  if(capacities[facilities[i]]>0)
  {
   if((brute_force_cost+distance[customers[ci]][facilities[i]])
   > minimum_brute_force_cost)
    continue;
   assigned_facilities[ci] = facilities[i];
   brute_force_cost += distance[customers[ci]][facilities[i]];
   capacities[facilities[i]]--;
   //call recursively
   brute_force(nCustomers, ci+1);
   //unassign the customer
   brute_force_cost -= distance[customers[ci]][facilities[i]];
   capacities[facilities[i]]++;
  }
 }
}
```

```
void optimal_algorithm_with_some_customers(int nCustomers)
{
 int i;
 /*for(i=0;i<number_of_customers;i++)
 {
  assigned_facilities[i] = -1;
 }*/
 for(i=0;i<number_of_vertices;i++)
  capacities[i] = capacity;
 brute_force_cost = 0;
 minimum_brute_force_cost = INFINITY;
 brute_force(nCustomers, 0);
}
void optimal_algorithm()
{
 optimal_algorithm_with_some_customers(number_of_customers);
}
void optimal_algorithm_for_graph()
{
 optimal_algorithm();
}
void algorithm_optimal_fill()
{
 int optimal_fill_capacities[MAX_VERTICES + EXTRA];
 int i;
 for(i=0;i<number_of_vertices;i++)
  optimal_fill_capacities[i] = capacity;
 for(i=0;i<number_of_customers;i++)
 {
  optimal_algorithm_with_some_customers(i+1);
  int j;
  //printf("optimal_fill_capacities:");
  for(j=0;j<number_of_vertices;j++)
  {
   //printf(" %d",optimal_fill_capacities[j] );
   if(optimal_fill_capacities[j] != last_capacities[j])
    break;
  }
  //printf("\n");
  total_cost_of_optimal_fill += distance[customers[i]][j];
  optimal_fill_capacities[j]--;
```

```
  //printf("customer %d assigned to facility %d with cost %d\n",
  customers[i],j,distance[customers[i]][j] );
 }
}
void start_simulation(int number_of_simulations,
int type, int preference)
{
 time_t t;
 srand((unsigned) time(&t));

 int fi;
 for(fi=1;fi<=15;fi++)
 {
  unsigned int start_time = (unsigned)time(NULL);
  double avg_greedy = 0, avg_rand = 0, avg_optfill = 0,
  max_greedy = 0, max_rand = 0, max_optfill = 0;

  int i;
  for(i=0;i<number_of_simulations;i++)
  {
   do{
    //number_of_vertices = rand() % (MAX_VERTICES+1);
    number_of_vertices = 100;
   }while(!number_of_vertices);
   //number_of_vertices = 5;
   printf("number of vertices:%d\n",number_of_vertices);
   fprintf(output_file, "number of vertices:%d\n",
   number_of_vertices);
   do{
    //number_of_facilities = rand() % (number_of_vertices+1);
    number_of_facilities = fi;
   }while(!number_of_facilities);
   printf("number of facilities:%d\n",number_of_facilities);
   fprintf(output_file, "number of facilities:%d\n",
   number_of_facilities);
   if(type == GRAPH)generate_graph();
   else if(type == STRAIGHT_LINE)generate_path();
   randomly_generate_facilities();
   do{
    //capacity = rand() % (MAX_CAPACITY+1);
    capacity = 1;
```

```
}while(!capacity);
printf("capacity:%d\n",capacity);
fprintf(output_file, "capacity:%d\n",capacity);
number_of_customers = number_of_facilities * capacity;
printf("number of customers:%d\n",number_of_customers);
fprintf(output_file, "number of customers:%d\n",number_of_customers);
generate_customers();
find_all_pair_shortest_path();
if(preference)
{
 int preference_unit = 5;
 int j;
 for(j=0;j<number_of_customers;j++)
 {
  clean_marker();
  int k;
  for(k=0;k<number_of_facilities;k++)
  {
   int rand_facility;
   do
   {
    rand_facility = rand() % number_of_facilities;
   }while(marker[rand_facility]);
   marker[rand_facility]=1;
   distance[customers[j]][facilities[rand_facility]] +=
  (k+1) * preference_unit;
  }
 }
 clean_marker();
}
//run algorithms
algorithm_greedy();
if(type == STRAIGHT_LINE)algorithm_randomized_greedy(1);
algorithm_optimal_fill();
if((type == GRAPH) || (preference))
{
 optimal_algorithm_for_graph();
 total_cost_of_optimal = minimum_brute_force_cost;
}
else if(type == STRAIGHT_LINE)optimal_algorithm_for_path();
//optimal_algorithm();
```

```
//print result
printf("total cost of greedy:%d\n",
total_cost_of_greedy);
fprintf(output_file, "total cost of greedy:%d\n",
total_cost_of_greedy);
printf("total cost of randomized greedy:%d\n",
total_cost_of_randomized_greedy);
fprintf(output_file, "total cost of randomized greedy:%d\n",
total_cost_of_randomized_greedy);
printf("total cost of optimal fill:%d\n",
total_cost_of_optimal_fill);
fprintf(output_file, "total cost of optimal fill:%d\n",
total_cost_of_optimal_fill);
printf("total cost of optimal:%d\n",
total_cost_of_optimal);
fprintf(output_file, "total cost of optimal:%d\n",
total_cost_of_optimal);
double temp=0;
if(total_cost_of_optimal>0)temp =
(total_cost_of_greedy*1.0/total_cost_of_optimal);
avg_greedy += temp;
if(max_greedy < temp)max_greedy = temp;
printf("ratio of greedy:%lf\n", temp);
fprintf(output_file, "ratio of greedy:%lf\n", temp);
if(total_cost_of_optimal>0)temp =
(total_cost_of_randomized_greedy*1.0/total_cost_of_optimal);
avg_rand += temp;
if(max_rand < temp)max_rand = temp;
printf("ratio of randomized greedy:%lf\n", temp);
fprintf(output_file, "ratio of randomized greedy:%lf\n", temp);
if(total_cost_of_optimal>0)temp =
(total_cost_of_optimal_fill*1.0/total_cost_of_optimal);
avg_optfill += temp;
if(max_optfill < temp)max_optfill = temp;
printf("ratio of optimal fill:%lf\n", temp);
fprintf(output_file, "ratio of optimal fill:%lf\n", temp);
//printf("total cost of brute force:%d\n",
minimum_brute_force_cost);
clean_previous_data();
}
```

```
  printf("avg_greedy:%lf\n",avg_greedy );
  printf("number_of_simulations:%d\n",number_of_simulations);

  avg_greedy = avg_greedy/number_of_simulations;
  printf("average ratio of greedy for %d facilities:%lf\n",
  fi, avg_greedy );
  printf("maximum ratio of greedy for %d facilities:%lf\n",
  fi, max_greedy );
  fprintf(output_file, "average ratio of greedy for %d facilities:%lf\n",
  fi, avg_greedy );
  fprintf(output_file, "maximum ratio of greedy for %d facilities:%lf\n",
  fi, max_greedy );


  avg_rand = avg_rand/number_of_simulations;
  printf("average ratio of randomized greedy for %d facilities:%lf\n",
  fi, avg_rand );
  printf("maximum ratio of randomized greedy for %d facilities:%lf\n",
  fi, max_rand );
  fprintf(output_file, "average ratio of randomized greedy for
  %d facilities:%lf\n", fi, avg_rand );
  fprintf(output_file, "maximum ratio of randomized greedy for
  %d facilities:%lf\n", fi, max_rand );

  avg_optfill = avg_optfill/number_of_simulations;
  printf("average ratio of optimal fill for %d facilities:%lf\n",
  fi, avg_optfill );
  printf("maximum ratio of optimal fill for %d facilities:%lf\n",
  fi, max_optfill );
  fprintf(output_file, "average ratio of optimal fill for
  %d facilities:%lf\n", fi, avg_optfill );
  fprintf(output_file, "maximum ratio of optimal fill for
  %d facilities:%lf\n", fi, max_optfill );

  printf("time taken:%u\n", ((unsigned)time(NULL) - start_time));
  fprintf(output_file, "time taken:%u\n",
  ((unsigned)time(NULL) - start_time));
  printf("**********************************\n");
  fprintf(output_file, "**********************************\n");
 }
}
```

```
int main()
{
 char file_name[2000];
 strcpy(file_name, "output-");
 char temp_str[200];
 sprintf(temp_str, "%u", (unsigned)time(NULL));
 strcat(file_name, temp_str);
    output_file = fopen(file_name, "w");

 start_simulation(20, STRAIGHT_LINE, 0);
 //start_simulation(20, STRAIGHT_LINE, 1);
 //start_simulation(100, GRAPH);

 fclose(output_file);
 return 0;
}
```

Index