Online Algorithms

Allan Borodin Denis Pankratov

DRAFT: March 14, 2019

Contents

Preface

In 1998, Allan Borodin and Ran El-Yaniv co-authored the text "Online computation and competitive analysis". As in most texts, some important topics were not covered. To some extent, this text aims to rectify some of those omissions. Furthermore, because the field of Online Algorithms has remained active, many results have been improved. But perhaps most notable, is that the basic adversarial model of competitive analysis for online algorithms has evolved to include new models of algorithm design that are important theoretically as well as having a significant impact on many current applications.

In Part I, we first review the basic definitions and some of the "classical" online topics and algorithms; that is, those already well studied as of 1998. We then present some newer online results for graph problems, scheduling problems, max-sat and submodular function maximization. We also present the seminal primal dual analysis for online algorithms introduced by Buchbinder and Naor that provides an elegant unifying model for many online results. Part I concludes with an update on some recent progress for some of the problems introduced earlier.

The focus of Part II is the study of extensions of the basic online competitive analysis model. In some chapters, we discuss alternative "online-like" algorithmic models. In other chapters, the analysis of online algorithms goes beyond the worst case perspective of the basic competitive analysis in terms of both stochastic analysis as well as alternative performance measures.

In Part III, we discuss some additional application areas, many of these applications providing the motivation for the continuing active interest (and indeed for what we consider is a renaissance) in the study of online and online-like algorithms. We view our text primarily as an introduction to the study of online and online-like algorithms for use in advanced undergraduate and graduate courses. In addition, we believe that the text offers a number of interesting potential research topics.

At the end of each chapter, we present some exercises. Some exercises are denoted by a (*) indicating that the exercise is technically more challenging. Some other exercises are denoted by as (**) indicating that to the best of our knowledge this is not a known result. With the exception of Chapter 1, we will no t mention specific references in the main sections of the chapters. Rather, ee have provided some history and the most relevant citations for results presented in the chapter and often also some relevant related work that is not mentioned in the chapter. Our convention will be to refer to all authors for papers having at most three authors; for paper having more than three authors we will use "author, et al".

We thank ....

Part I

Basics

Chapter 1

Introduction

In this chapter we introduce online problems and online algorithms, give a brief history of the area, and present several motivating examples. The first two examples are the ski rental problem and the line search problem. We analyze several online algorithms for these problems using the classical notion of competitive analysis. The last problem we consider is paging. While competitive analysis can be applied to algorithms for this problem as well, the results significantly deviate from the "practical" performance of these paging algorithms. This highlights the necessity of new tools and ideas that will be explored throughout this text.

1.1 What is this Book About?

This book is about the analysis of online problems. In the basic formulation of an online problem, an input instance is given as a sequence of input items. After each input item is presented, an algorithm needs to output a decision and that decision is final, i.e., cannot be changed upon seeing any future items. The goal is to maximize or minimize an objective function, which is a function of all decisions for a given instance. (We postpone a formal definition of an online problem but hopefully the examples that follow will provide a clear intuitive meaning.) The term "online" in "online algorithms" refers to the notion of irrevocable decisions and has nothing to do with Internet, although a lot of the applications of the theory of online algorithms are in networking and online applications on the internet. The main limitation of an online algorithm is that it has to make a decision in the absence of the entire input. The value of the objective achieved by an online algorithm is compared against an optimal value of the objective that is achieved by an ideal "offline algorithm," i.e., an algorithm having access to the entire input. The ratio of the two values is called the competitive ratio.

We shall study online problems at different levels of granularity. At each level of granularity, we are interested in both positive and negative results. For instance, at the level of individual algorithms, we fix a problem, present an algorithm, and prove that it achieves a certain performance (positive result) and that the performance analysis is tight (negative result). At the higher level of models, we fix a problem, and ask what is the best performance achievable by an algorithm of a certain type (positive result) and what is an absolute bound on the performance achievable by all algorithms of a certain type (negative result). The basic model of deterministic online algorithms can be extended to allow randomness, side information (a.k.a. advice), limited mechanisms of revoking decisions, multiple rounds, and so on. Negative results can often be proved by interpreting an execution of an algorithm as a game between the algorithm and an adversary. The adversary constructs an input sequence so as to fool an algorithm into making bad online decisions. What determines the nature and order of input arrivals? In the standard version of competitive analysis, the input items and their arrival order is arbitrary and can be viewed as being determined by an all powerful adversary. While helpful in many situations, traditional worst-case analysis is often too pessimistic to be of practical value. Thus, it is sometimes necessary to consider limited adversaries. In the *random order model* the adversary chooses the set of input items but then the order is determined randomly; in stochastic models, an adversary chooses an input distribution which then determines the sequence of input item arrivals. Hence, in all these models there is some concept of an adversary attempting to force the "worst-case" behavior for a given online algorithm or the worst-case performance against *all* online algorithms.

A notable feature of the vanilla online model is that it is information-theoretic. This means that there are no computational restrictions on an online algorithm. It is completely legal for an algorithm to perform an exponential amount of computation to make the nth decision. At first, it might seem like a terrible idea, since such algorithms wouldn't be of any practical value whatsoever. This is a valid concern, but it simply doesn't happen. Most of the positive results are achieved by very efficient algorithms, and the absence of computational restrictions on the model makes negative results really strong. Perhaps, most importantly information-theoretic nature of the online model leads to unconditional results, i.e., results that do not depend on unproven assumptions, such as $P \neq NP$.

We shall take a tour of various problems, models, analysis techniques with the goal to cover a selection of classical and more modern results, which will reflect our personal preferences to some degree. The area of online algorithms has become too large to provide a full treatment of it within a single book. We hope that you can accompany us on our journey and that you will find our selection of results both interesting and useful!

1.2 Motivation

Systematic theoretical study of online algorithms is important for several reasons. Sometimes, the online nature of input items and decisions is forced upon us. This happens in a lot of scheduling or resource allocation applications. Consider, for example, a data center that schedules computing jobs: clearly it is not feasible to wait for all processes to arrive in order to come up with an optimal schedule that minimizes makespan. The jobs have to be schedules as they come in. Some delay might be tolerated, but not much. As another example, consider patients arriving at a walk-in clinic and need to be seen by a relevant doctor. Then the receptionist plays the role of an online algorithm, and his or her decisions can be analyzed using the online framework. In online (=Internet) advertising, when a user clicks on a webpage, some advertiser needs to be matched to a banner immediately. We will see many more applications of this sort in this book. In such applications, an algorithm makes a decision no matter what: if an algorithm takes too long to make a decision it becomes equivalent to the decision of completely ignoring an item.

One should also note that the term "online algorithm" is used in a related but different way by many people with respect to scheduling algorithms. Namely, in many scheduling results, "online" could arguably be more appropriately be called "real time computation" where inputs arrive with respect to continuous time t and algorithmic decisions can be delayed at the performance cost of "wasted time". In machine learning, the concept of *regret* is the analogue of the competitive ratio (see Chapter 18). Economists have long studied market analysis within the lens of online decision making. Navigation in geometric spaces and mazes and other aspects of "search" have also been viewed as online computation. Our main perspective and focus falls within the area of algorithmic analysis for discrete computational problems. In online computation, we view input items as arriving in discrete steps and in the initial basic model used in competitive analysis, an irrevocable decision must be made for each input item before the next item arrives. Rather than the concept of a real time clock determining time, we view time in terms of these discrete time steps.

Setting aside applications where input order and irrevocable decisions are forced upon us, in some offline applications it might be worthwhile fixing the order of input items and considering online algorithms. Quite often such online algorithms give rise to conceptually simple and efficient offline approximation algorithms. This can be helpful not only for achieving non-trivial approximation ratios for NP-hard problems, but also for problems in P (such as bipartite matching), since optimal algorithms can be too slow for large practical instances. Simple greedy algorithms tend to fall within this framework consisting of two steps: sorting input items, followed by a single online pass over the sorted items. In fact, there is a formal model for this style of algorithms called the priority model, and we will study it in detail in Chapter 17.

Online algorithms also share a lot of features with streaming algorithms. The setting of streaming algorithms can be viewed figuratively as trying to drink out of a firehose. There is a massive stream of data passing through a processing unit, and there is no way to store the entire stream for postprocessing. Thus, streaming algorithms are concerned with minimizing memory requirements in order to compute a certain function of a sequence of input items. While online algorithms do not have limits on memory or per-item processing time, some positive results from the world of online algorithms are both memory and time efficient. Such algorithms can be useful in streaming settings, which are frequent in networking and scientific computing.

1.3 Brief History of Online Algorithms

It is difficult to establish the first published analysis of an online algorithm but, for example, one can believe that there has been substantial interest in main memory paging since paging was introduced into operating systems. A seminal paper in this regard is Peter Denning's [16] introduction of the working set model for paging. It is interesting to note that almost 50 years after Denning's insightful approach to capturing locality of reference, Albers et al [2] established a precise result that characterizes the page fault rate in terms of a parameter $f(n)$ that measures the number of distinct page references in the next n consecutive page requests.

Online algorithms has been an active area of research within theoretical computer science since 1985 when Sleator and Tarjan [46] suggested that worst-case competitive analysis provided a better (than existing "average-case" analysis) explaination for the success of algorithms such as *move to front* for the list accessing problem (see chapter 4). In fact, as in almost any research area, there are previous worst case results that can be seen as at least foreshadowing the interest in competitive analysis, where one compares the performance of an online algorithm relative to what can be achieved optimally with respect to all possible inputs. Namely, Graham's [29] online greedy algorithm for the identical machines makespan problem and even more explicitly Yao's [51] analysis of online bin packing algorithms. None of these works used the term competitve ratio; this terminology was introduced by Karlin et al. [36] in their study of "snoopy caching" following the Sleator and Tarjan paper.

Perhaps remarkably, the theoretical study of online algorithms has remained an active field and one might even argue that there is now a renaissance of interest in online algorithms. This growing interest in online algorithms and analysis is due to several factors, including new applications, online model extensions, new performance measures and constraints, and an increasing interest in experimental studies that validate or challenge the theoretical analysis. And somewhat ironically, average-case analysis (i.e. stochastic analysis) has become more prominent in the theory, design and analysis of algorithms. We believe the field of online algorithms has been (and will continue to be) a very successful field. It has led to new algorithms, new methods of analysis and a deeper understanding of well known existing algorithms.

1.4 Motivating Example: Ski Rental

Has it ever happened to you that you bought an item on an impulse, used it once or twice, and then stored it in a closet never to be used again? Even if you absolutely needed to use the item, there may have been an option to rent a similar item and get the job done at a much lower cost. If this seems familiar, you probably thought that there has to be a better way for deciding whether to buy or rent. It turns out that many such rent versus buy scenarios are represented by a single problem. This problem is called "ski rental" and it can be analyzed using the theory of online algorithms. Let's see how.

The setting is as follows. You have arrived at a ski resort and you will be staying there for an unspecified number of days. As soon as the weather turns bad and the resort closes down the slopes, you will leave never to return again. Each morning you have to make a choice either to rent skis for r\$ or to buy skis for b\$. By scaling we can assume that $r = 1$\$. For simplicity, we will assume that b is an integer ≥ 1. If you rent for the first k days and buy skis on day $k+1$, you will incur the cost of $k+b$ for the entire stay at the resort, that is, after buying skis you can use them an unlimited number of times free of charge. The problem is that due to unpredictable weather conditions, the weather might deteriorate rapidly. It could happen that the day after you buy the skis, the weather will force the resort to close down. Note that the weather forecast is accurate for a single day only, thus each morning you know with perfect certainty whether you can ski on that day or not before deciding to buy or rent, but you have no information about the following day. In addition, unfortunately for you, the resort does not accept returns on purchases. In such unpredictable conditions, what is the best strategy to minimize the cost of skiing during all good-weather days of your stay?

An optimal offline algorithm simply computes the number g of good-weather days. If $g \leq b$ then an optimal strategy is to rent skis on all good-weather days. If $g > b$ then the optimal strategy is to buy skis on the first day. Thus, the offline optimum is $\min(g, b)$. Even without knowing g it is possible to keep expenses roughly within a factor of 2 of the optimum. The idea is to rent skis for $b-1$ days and buy skis on the following day after that. If $g < b$ then this strategy costs $g \leq (2-1/b)g$, since the weather would spoil on day $g+1 \leq b$ and you would leave before buying skis on day b. Otherwise, $g \geq b$ and our strategy incurs cost $b-1+b = (2-1/b)b$, which is slightly better than twice the offline optimum, since for this case we have $b = \min(g, b)$. This strategy achieves competitive ratio $2-1/b$, which approaches 2 as b increases.

Can we do better? If our strategy is deterministic, then no. On each day, an adversary sees whether you decided to rent or buy skis, and based on that decision and past history declares whether the weather is going to be good or bad starting from the next day onward. If you buy skis on day $i \leq b-1$ then the adversary declares the weather to be bad from day $i+1$ onward. This way an optimal strategy is to rent skis for a total cost of i, but you incurred the cost of $(i-1)+b \geq (i-1)+(i+1) = 2i$; that is, twice the optimal. If you buy skis on day $i \geq b$, then the adversary declares bad weather after the first $2b$ days. An optimal strategy is to buy skis on the very first day with a cost of b, whereas you spent $i-1+b \geq b-1+b = (2-1/b)b$. Thus, no matter what you do an adversary can force you to spend $(2-1/b)$ times the optimal.

Can we do better if we use randomness? We assume a weak adversary — such an adversary knows your algorithm, but has to commit to spoiling weather on some day $g+1$ without seeing your random coins or seeing any of your decisions. Observe that for deterministic algorithms, a weak adversary can simulate the stronger one that adapts to your decisions. The assumption of a weak adversary for the ski rental problem is reasonable because the weather doesn't seem to conspire against you based on the outcomes of your coin flips. It is reasonable to conjecture that even with randomized strategy you should buy skis before or on day b. You might improve the competitive ratio if you buy skis before day b with some probability. One of the simplest randomized algorithms satisfying these conditions is to pick a random integer $i \in [0, b-1]$ from some distribution p and rent for i days and buy skis on day $i+1$ (if the weather is still good). Intuitively, the distribution should allocate more probability mass to larger values of i, since buying skis very early (think of the first day) makes it easier for the adversary to punish such decision. We measure the competitive ratio achieved by a randomized algorithm by the ratio of the expected cost of the solution found by the algorithm to the cost of an optimal offline solution. To analyze our strategy, we consider two cases.

In the first case, the adversary spoils the weather on day $g+1$ where $g < b$. Then the expected cost of our solution is $\sum_{i=0}^{g-1}(i+b)p_i + \sum_{i=g}^{b-1} gp_i$. Since an optimal solution has cost g in this case, we are interested in finding the minimum value of c such that

$$\sum_{i=0}^{g-1}(i+b)p_i + \sum_{i=g}^{b-1} gp_i \le cg.$$

In the second case, the adversary spoils the weather on day $g+1$ where $g \ge b$. Then the expected cost of our solution is $\sum_{i=0}^{b-1} ip_i + b$. Since an optimal solution has cost b in this case, we need to ensure $\sum_{i=0}^{b-1} ip_i + b \le cb$.

We can write down a linear program to minimize c subject to the above inequalities together with the constraint $p_0 + p_1 + \cdots + p_{b-1} = 1$.

$$\begin{array}{lcl}
\text{minimize} & c & \\
\text{subject to} & \displaystyle\sum_{i=0}^{g-1}(i+b)p_i + \sum_{i=g}^{b-1} gp_i \le cg & \text{for } g \in [b-1] \\
& \displaystyle\sum_{i=0}^{b-1} ip_i + b \le cb & \\
& p_0 + p_1 + \cdots + p_{b-1} = 1 &
\end{array}$$

We claim that $p_i = \frac{c}{b}(1-1/b)^{b-1-i}$ and $c = \frac{1}{1-(1-1/b)^b}$ is a solution to the above LP. Thus, we need to check that all constraints are satisfied. First, let's check that p_i form a probability distribution:

$$\sum_{i=0}^{b-1} p_i = \sum_{i=0}^{b-1} \frac{c}{b}(1-1/b)^{b-1-i} = \frac{c}{b}\sum_{i=0}^{b-1}(1-1/b)^i = \frac{c}{b}\frac{1-(1-1/b)^b}{1-(1-1/b)} = 1.$$

Next, we check all constraints involving g.

$$\begin{aligned}\sum_{i=0}^{g-1}(i+b)p_i + \sum_{i=g}^{b-1} g p_i &= \sum_{i=0}^{g-1}(i+b)\frac{c}{b}(1-1/b)^{b-1-i} + \sum_{i=g}^{b-1} g\frac{c}{b}(1-1/b)^{b-1-i} \\ &= (1-1/b)^{b-g}cg + \Big((1-1/b)^g - (1-1/b)^b\Big)(1-1/b)^{-g}cg \\ &= cg\end{aligned}$$

In the above, we skipped some tedious algebraic computations (we invite the reader to verify each of the above equalities). Similarly, we can check that the p_i satisfy the second constraint of the LP. Notably, the solution p_i and c given above satisfies each constraint with equality. We conclude that our randomized algorithm achieves the competitive ratio $\frac{1}{1-(1-1/b)^b}$. Since $(1-1/n)^n \to e^{-1}$, the competitive ratio of our randomized algorithm approaches $\frac{e}{e-1} \approx 1.5819\ldots$ as b goes to infinity.

1.5 Motivating Example: Line Search Problem

A robot starts at the origin of the x-axis. It can travel one unit of distance per one unit of time along the x-axis in either direction. An object has been placed somewhere on the x-axis. The robot can switch direction of travel instantaneously, but in order for the robot to determine that there is an object at location x', the robot has to be physically present at x'. How should the robot explore the x-axis in order to find the object as soon as possible? This problem is known as the line search problem or the cow path problem.

Suppose that the object has been placed at distance d from the origin. If the robot knew whether the object was placed to the right of the origin or to the left of the origin, the robot could start moving in the right direction, finding the object in time d. This is an optimal "offline" solution.

Since the robot does not know in which direction it should be moving to find the object, it needs to explore both directions. This leads to a natural zig-zag strategy. Initially the robot picks the positive direction and walks for 1 unit of distance in that direction. If no object is found, the robot returns to the origin, flips the direction and doubles the distance. We call each such trip in one direction and then back to the origin a phase, and we start counting phases from 0. These phases are repeated until the object is found. If you have seen the implementation and amortized analysis of an automatically resizeable array implementation, then this doubling strategy will be familiar. In phase i robot visits location $(-2)^i$ and travels the distance $2 \cdot 2^i$. Worst case is when an object is located just outside of the radius covered in some phase. Then the robot returns to the origin, doubles the distance and travels in the "wrong direction", returns to the origin, and discovers the object by travelling in the "right direction." In other words when an object is at distance $d = 2^i + \epsilon > 2^i$ in direction $(-1)^i$, the total distance travelled is $2(1+2+\cdots+2^i+2^{i+1})+d \le 2\cdot 2^{i+2}+d < 8d+d = 9d$. Thus, this doubling strategy gives a 9-competitive algorithm for the line search problem.

Typically online problems have well-defined input that makes sense regardless of which algorithm you choose to run, and the input is revealed in an online fashion. For example, in the ski rental problem, the input consists of a sequence of elements, where element i indicates if the weather on day i is good or bad. The line search problem does not have input of this form. Instead, the input is revealed in response to the actions of the algorithm. Yet, we can still interpret this situation as a game between an adversary and the algorithm (the robot). At each newly discovered location, an adversary has to inform the robot whether an object is present at that location or not. The adversary eventually has to disclose the location, but the adversary can delay it as long as needed in order to maximize the distance travelled by the robot in relation to the "offline" solution.

1.6 Motivating Example: Paging

Computer storage comes in different varieties: CPU registers, random access memory (RAM), solid state drives (SSD), hard drives, tapes, etc. Typically, the price per byte is positively correlated with the speed of the storage type. Thus, the fastest type of memory – CPU registers – is also the most expensive, and the slowest type of memory – tapes – is also the cheapest. In addition, certain types of memory are volatile (RAM and CPU registers), while other types (SSDs, hard drives, tapes) are persistent. Thus, a typical architecture has to mix and match different storage types. When information travels from a large-capacity slow storage type to a low-capacity fast storage type, e.g., RAM to CPU registers, some bottlenecking will occur. This bottlenecking can be mitigated by using a cache. For example, rather than accessing RAM directly, the CPU checks a local cache, which stores a local copy of a small number of pages from RAM. If the requested data is in the cache (this event is called "cache hit"), the CPU retrieves it directly from the cache. If the requested data is not in the cache (called "cache miss"), the CPU first brings the requested data from RAM into the cache, and then reads it from the cache. If the cache is full during a "cache miss," some existing page in the cache needs to be evicted. The paging problem is to design an algorithm that decides which page needs to be evicted when the cache is full and cache miss occurs. The objective is to minimize the total number of cache misses. Notice that this is an inherently online problem that can be modelled as follows. The input is a sequence of natural numbers $X = x_1, x_2, \ldots$, where x_i is the number of the page requested by the CPU at time i. Given a cache of size k, initially the cache is empty. The cache is simply an array of size k, such that a single page can be stored at each position in the array. For each arriving x_i, if x_i is in the cache, the algorithm moves on to the next element. If x_i is not in the cache, the algorithm specifies an index $y_i \in [k]$, which points to a location in the cache where page x_i is to be stored evicting any existing page. We will measure the performance by the classical notion of the competitive ratio — the ratio of the number of cache misses of an online algorithm to the minimum number of cache misses achieved by an optimal offline algorithm that sees the entire sequence in advance. Let's consider two natural algorithms for this problem.

<ins>FIFO - First In First Out.</ins> If the cache is full and a cache miss occurs, this algorithm evicts the page from the cache that was inserted the earliest. We will first argue that this algorithm incurs at most (roughly) k times more cache misses than an optimal algorithm. To see this, subdivide the entire input into consecutive blocks $B_1, B_2, \ldots$. Block B_1 consists of a maximal prefix of X that contains exactly k distinct pages (if the input has fewer than k distinct pages, then any "reasonable" algorithm is optimal). Block B_2 consists of a maximal prefix of $X \setminus B_1$ that contains exactly k distinct pages, and so on. Let n be the number of blocks. Observe that FIFO incurs at most k cache misses while processing each block. Thus, the overall number of cache misses of FIFO is at most nk. Also, observe that the first page of block B_{i+1} is different from *all pages* of B_i due to the maximality of B_i. Therefore while processing B_i and the first page from B_{i+1} any algorithm, including an optimal offline one, incurs a cache miss. Thus, an optimal offline algorithm incurs at least $n-1$ cache misses while processing X. Therefore, the competitive ratio of FIFO is at most $nk/(n-1) = k + \frac{k}{n-1} \to k$ as $n \to \infty$.

<ins>LRU - Least Recently Used.</ins> If the cache is full and a cache miss occurs, this algorithm evicts the page from the cache that was accessed least recently. Note that LRU and FIFO both keep timestamps together with pages in the cache. When x_i is requested and it results in a cache miss, both algorithms initialize the timestamp corresponding to x_i to i. The difference is that FIFO never updates the timestamp until x_i itself is evicted, whereas LRU updates the timestamp to j whenever cache hit occurs, where $x_j = x_i$ with $j > i$ and x_i still in the cache. Nonetheless, the two algorithms are sufficiently similar to each other, that essentially the same analysis as for FIFO can

be used to argue that the competitive ratio of LRU is at most k (when $n \to \infty$).

We note that both FIFO and LRU do not achieve a competitive ratio better than k. Furthermore, no deterministic algorithm for paging can achieve a competitive ratio better than k. To prove this, it is sufficient to consider sequences that use page numbers from $[k+1]$. Let A be a deterministic algorithm, and suppose that it has a full cache. Since the cache is of size k, in each consecutive time step an adversary can always find a page that is not in the cache and request it. Thus, an adversary can make the algorithm A incur a cache miss on every single time step. An optimal offline algorithm evicts the page from the cache that is going to be requested furthest in the future. Since there are k pages in the cache, there is at least $k-1$ pages in the future inputs that are going to be requested before one of the pages in the cache. Thus, the next cache miss can only occur after $k-1$ steps. The overall number of cache misses by an optimal algorithm is at most $|X|/k$, whereas A incurs essentially $|X|$ cache misses. Thus, A has competitive ratio at least k.

We finish this section by noting that while competitive ratio gives useful and practical insight into the ski rental and line search problems, it falls short of providing practical insight into the paging problem. First of all, notice that the closer competitive ratio is to 1 the better. The above paging results show that increasing cache size k makes LRU and FIFO perform worse! This goes directly against the empirical observation that larger cache sizes lead to improved performance. Another problem is that the competitive ratio of LRU and FIFO is the same suggesting that these two algorithms perform equally well. It turns out that in practice LRU is *far superior* to FIFO, because of "locality of reference" – the phenomenon that if some memory was accessed recently, the same or nearby memory will be accessed in the near future. There are many reasons for why this phenomenon is pervasive in practice, not the least of which is a common use of arrays and loops, which naturally exhibit "locality of reference." None of this is captured by competitive analysis as it is traditionally defined.

The competitive ratio is an important tool for analyzing online algorithms having motivated and initiating the area of online algorithm analysis. However, being a worst-case measure, it may not not model reality well in many applications. This has led researchers to consider other models, such as stochastic inputs, advice, look-ahead, and parameterized complexity, among others. We shall cover these topics in the later chapters of this book.

1.7 Exercises

1. Fill in details of the analysis of the randomized algorithm for the ski rental problem.

2. Consider the setting of the ski rental problem with rental cost 1\$ and buying cost b\$, $b \in \mathbb{N}$. Instead of an adversary choosing a day $\in \mathbb{N}$ when the weather is spoiled, this day is generated at random from distribution p. Design an optimal deterministic online algorithm for the following distributions p:

   (a) uniform distribution on $[n]$.

   (b) geometric distribution on $\mathbb{N}$ with parameter 1/2.

3. What is the competitive ratio achieved by the following randomized algorithm for the line search problem? Rather than always picking initial direction to be $+1$, the robot selects the initial direction to be $+1$ with probability 1/2 and -1 with probability 1/2. The rest of the strategy remains the same.

4. Instead of searching for treasure on a line, consider the problem of searching for treasure on a 2-dimensional grid. The robot begins at the origin of $\mathbb{Z}^2$ and the treasure is located

at some coordinate $(x, y) \in \mathbb{Z}^2$ unknown to the robot. The measure of distance is given by the Manhattan metric $||(x, y)|| = |x| + |y|$. The robot has a compass and at each step can move north, south, east, or west one block. Design an algorithm for the robot to find the treasure on the grid. What is the competitive ratio of your algorithm? Can you improve the competitive ratio with another algorithm?

5. Consider Flush When Full algorithm for paging: when a cache miss occurs and the entire cache is full, evict *all* pages from the cache. What is the competitive ratio of Flush When Full?

6. Consider adding the power of limited lookahead to an algorithm for paging. Namely, fix a constant $f \in \mathbb{N}$. Upon receiving the current page p_i, the algorithm also learns $p_{i+1}, p_{i+2}, \ldots, p_{i+f}$. Recall that the best achievable competitive ratio for deterministic algorithms without lookahead is k. Can you improve on this bound with lookahead?

Chapter 2

Deterministic Online Algorithms

In this chapter we formally define a general class of online problems, called *request-answer games* and then define the competitive ratio of deterministic online algorithms with respect to request-answer games. The definitions depend on whether we are dealing with a minimization or a maximization problem. As such, we present examples of each. For minimization problems we consider the makespan problem and the bin packing problem.For maximization problems we consider time-series search and the one-way trading problems.

2.1 Request-Answer Games

In the Chapter 1, we gave an informal description of an online problem, one that is applicable to most online problems in the competitive analysis literature including the ski rental, paging, and makespan problems. Keeping these problems in mind, we can define the general request-answer framework that formalizes the class of online problems that abstracts almost all the problems in this text with the notable exception of the line search problem and more generally the navigation and exploration problems in Chapter 23.

Definition 2.1.1. A *request-answer* game for a minimization problem consists of a request set R, an answer set[1] A, and cost functions $f_n : R^n \times A^n \to \mathbb{R} \cup \{\infty\}$ for $n = 0, 1, \ldots$.

Here ∞ indicates that certain solutions are not allowed. For a maximization problem the f_n refer to profit functions and we have $f_n : R^n \times A^n \to \mathbb{R} \cup \{-\infty\}$. Now $-\infty$ indicates that certain solutions are not allowed. We can now provide a template for deterministic online algorithms for any problem within the request-answer game framework.

```
   Online Algorithm Template
1: On an instance $I$, including an ordering of the data items $(x_1, \ldots, x_n)$:
2: $i := 1$
3: While there are unprocessed data items
4:    The algorithm receives $x_i \in R$ and makes an irrevocable decision $d_i \in A$ for $x_i$
      (based on $x_i$ and all previously seen data items and decisions).
5:    $i := i + 1$
6: EndWhile
```

[1]For certain results concerning requst-answer games, it is required that the answer set be a finite set. However, for the purposes of defining a general framework and for results in this text, this is not necessary.

It is important to note that the same computational problem can have several representations as a request answer game depending on the information in the request set and answer set. Often there wlll be a natural request and answer set for a given problem.

2.2 Competitive Ratio for Minimization Problems

In this section we formally define the notion of *competitive ratio*. In a later section we will analyze the performance of Graham's greedy makespan algorithm in terms of its competitive ratio. Let ALG be an online algorithm and $\mathcal{I} = \langle x_1, x_2, \ldots x_n \rangle$ be an input sequence. We shall abuse notation and let $ALG(\mathcal{I})$ denote both the output of the algorithm as well as the objective value of the algorithm when the context is clear. If we want to distinguish the objective value we will write $|ALG(\mathcal{I})|$.

Definition 2.2.1. Let ALG be an online algorithm for a minimization problem and let OPT denote an optimal solution to the problem. *The competitive ratio* of ALG, denoted by $\rho(ALG)$, is defined as follows:

$$\rho(ALG) = \limsup_{|OPT(\mathcal{I})| \to \infty} \frac{ALG(\mathcal{I})}{OPT(\mathcal{I})}.$$

Equivalently, we can say that the competitive ratio of ALG is ρ if for all sufficiently large inputs $\mathcal{I}$, we have $ALG(\mathcal{I}) \leq \rho \cdot OPT(\mathcal{I}) + o(OPT(\mathcal{I}))$. This then is just a renaming of the asymptotic approximation ratio as is widely used in the study of offline optimization algorithms. We reserve the competitive ratio terminology for online algorithms and use *approximation ratio* otherwise. In offline algorithms, when $ALG(\mathcal{I}) \leq \rho \cdot OPT(\mathcal{I})$ for all $\mathcal{I}$, we simply say approximation ratio; for online algorithms we will say that ALG is *strictly ρ competitive* when there is no additive term.

In the literature you will often see a slightly different definition of the competitive ratio. Namely, an online algorithm is said to achieve competitive ratio ρ if there is a constant $\alpha \geq 0$ such that for all inputs $\mathcal{I}$ we have $ALG(\mathcal{I}) \leq \rho \cdot OPT(\mathcal{I}) + \alpha$. We shall refer to this as the *classical definition*. The difference between the two definitions is that our definition allows us to ignore additive terms that are small compared to OPT, whereas the classical definition allows us to ignore constant additive terms. The two definitions are "almost" identical in the following sense. If an algorithm achieves the competitive ratio ρ with respect to the classical definition then it achieves the competitive ratio ρ with respect to our definition as well (assuming $OPT(\mathcal{I}) \to \infty$). Conversely, if an algorithm achieves the competitive ratio ρ with respect to our definition then it achieves the competitive ratio $\rho + \epsilon$ for any constant $\epsilon > 0$ with respect to the classical definition. The latter part is because we have $ALG \leq \rho OPT + o(OPT) = (\rho + \epsilon)OPT + o(OPT) - \epsilon OPT \leq (\rho + \epsilon)OPT + \alpha$ for a suitably chosen *constant* α such that α dominates the term $o(OPT) - \epsilon OPT$. We prefer our definition over the classical one because it makes stating the results simpler sometimes. In any case, as outlined above the difference between the two definitions is minor.

Implicit in this definition is the worst-case aspect of this performance measure. To establish a lower bound (i.e. an *inapproximation* for an online algorithm, there is a game between the algorithm and an adversary. Once the algorithm is stated the adversary creates a nemesis instance (or set of instances if we are trying to establish asymptotic inapproximation results). Sometimes it will be convenient to view this game in extensive form where the game alternates between the adversary announcing the next input item and the algorithm making a decision for this item. However, since the adversary knows the algorithm, the nemesis input sequence can be determined in advance and this becomes a game in normal (i.e., matrix) form. In addition to establishing inapproxination bounds (i.e. lower bounds on the competitive ratio) for specific algorithms, we sometimes wish to establish an inapproximation bound for *all* online algorithms. In this case, we need to show how to derive an appropriate nemesis sequence for any possible online algorithm.

2.3 Minimization Problem Example: Makespan

For definiteness, we consider the makespan problem for identical machines and Graham's online greedy algorithm. As stated earlier, this algorithm is perhaps the earliest example of a competitive analysis argument.

Makespan for identical machines
Input: $(p_1, p_2, \ldots, p_n)$ where p_j is the load or processing time for a job j; m — the number of identical machines.
Output: $\sigma : \{1, 2, \ldots, n\} \to \{1, 2, \ldots m\}$ where $\sigma(j) = i$ denotes that the j^{th} job has been assigned to machine i.
Objective: To find σ so as to minimize $\max_i \sum_{i:\sigma(j)=i} p_j$.

Algorithm 1 THE ONLINE GREEDY MAKESPAN algorithm.

```
procedure GREEDY MAKESPAN
    initialize $s(i) \leftarrow 0$ for $1 \le i \le m$        ▷ $s(i)$ is the current load on machine $i$
    $j \leftarrow 1$
    while $j \le n$ do
        $i' \leftarrow \arg\min_i s(i)$        ▷ The algorithm can break ties arbitrarily
        $\sigma(j) \leftarrow i'$
        $s(i') \leftarrow s(i') + p_j$
        $j \leftarrow j + 1$
    return $\sigma$
```

The makespan of a machine is its current load and the greedy algorithm schedules each job on some least loaded machine. The algorithm is online in the sense that the scheduling of the i^{th} job takes place before seeing the remaining jobs. The algorithm is *greedy* in the sense that at any step the algorithm schedules so as optimize as best as it can given the current state of the computation without regard to possible future jobs. We note that as stated, the algorithm is not fully defined. That is, there may be more than one machine whose current makespan is minimal. When we do not specify a "tie-breaking" rule, we mean that how the algorithm breaks ties is not needed for the correctness and performance analysis. That is the analysis holds even if we assume that an adversary is breaking the ties.

As an additional convention, when we use a WHILE loop we are assuming that the number n of online inputs is not known to the algorithm; otherwise, we will use a FOR loop to indicate that n is known a priori.

We now provide an example of competitive analysis by analyzing the online greedy algorithm for the makespan problem.

Theorem 2.3.1. *Let $G(\mathcal{I})$ denote the makespan of Graham's online greedy algorithm on m machines when executed on the input sequence $\mathcal{I}$.*

Then for all inputs $\mathcal{I}$ we have $G(\mathcal{I}) \le (2 - 1/m) \cdot OPT(\mathcal{I})$. That is, this online greedy algorithm has a strict competitive ratio which is at most $(2 - 1/m)$.

Proof. Let $p_1, p_2, \ldots p_n$ be the sequence of job sizes and let $p_{max} = \max_j p_j$. Lets first establish two necessary bounds for any solution and thus for OPT. The following are simple claims:

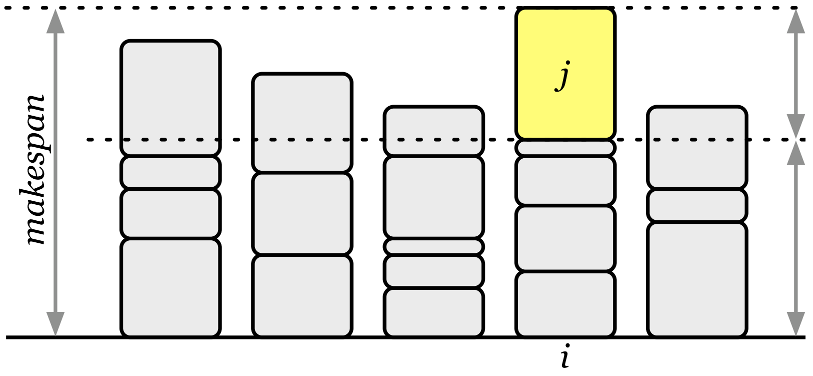

Figure 2.1: Figure from Jeff Erickson's lecture notes

- $OPT \geq (\sum_{k=1}^{n} p_k)/m$; that is the maximum load must be at least the average load.
- $OPT \geq p_{max}$

Now we want to bound the makespan of greedy in terms of these necessary OPT bounds. Let machine i be one of the machines defining the makespan for the greedy algorithm G and lets say that job j is the last item to be scheduled on machine i. If we let q_i be the load on machine i just before job j is scheduled, then $G's$ makespan is $q_i + p_j$. By the greedy nature of G and the fact that the minimum load must be less than the average load (see Figure 2.1), we have $q_i \leq \sum_{k \neq j} p_k/m$ so that

$$G(p_1, \dots p_n) \leq \sum_{k \neq j} p_k/m + p_j = \sum_{k \neq j} p_k/m + p_j/m - p_j/m + p_j = \sum_{k=1}^{n} p_k/m + (1 - 1/m)p_j$$

Since $\sum_{k=1}^{n} p_k/m \leq OPT$ and $p_j \leq p_{max} \leq OPT$ we have our desired competitive ratio. $\square$

We shall see that this *strict competitive ratio* is "tight" in the sense that there exists an input $\mathcal{I}$ such that $G(\mathcal{I}) = (2-1/m) \cdot OPT(\mathcal{I})$. We construct such an input sequence $\mathcal{I}$ with $n = m(m-1)+1$ jobs, comprised of $m(m-1)$ initial jobs having unit load $p_j = 1$ followed by a final job having load $p_n = m$. The optimal solution would balance the unit load jobs on say the first $m-1$ machines leaving the last machine to accommodate the final big job having load m. Thus each machine has load m and $OPT = m$. On the other hand, the greedy algorithm G would balance the unit job of all m machines and then be forced to place the last job on some machine which already has load $m-1$ so that the G's makespan is $m + (m-1)$. It follows that for this sequence the ratio is $\frac{2m-1}{m} = 2 - 1/m$ matching the bound in Theorem 2.3.1.

We note that for $m = 2$ and $m = 3$, it is not difficult to show that the bound is tight for *any* (not necessarily greedy) online algorithm. For example, for $m = 2$, an adversary can either provide the input sequence (1,1) or (1,1,2). If the algorithm spreads the two initial unit jobs, the adversary ends the input having only presented (1,1); otherwise the adversarial input is (1,1,2). Even though it may seem that this problem is pretty well understood, there is still much to reflect upon concerning the makespan problem and the greedy algorithm analysis. We note that the lower bound as given relies on the number n of inputs not being known initially by the algorithm. Moreover, the given inapproximation holds for input sequences restricted to $n \leq 3$ and does not establish an *asymptotic inapproximation.* For $m \geq 4$, there are online (non-greedy) algorithms that improve upon the greedy bound. The general idea for an improved competitive ratio is to leave some space for potentially large jobs. Currently, the best known “upper bound” that holds for all m is 1.901 and the “lower bound” for sufficiently large m is 1.88.

Although the greedy inapproximation is not an asymptotic result, the example suggests a simple greedy (but not online) algorithm. The nemesis sequence for all m relies on the last job being a large job. This suggests sorting the input items so that $p_1 \geq p_2 \ldots \geq p_n$. This is Graham's LPT (“longest processing time”) algorithm which has a tight approximation ratio of $\frac{4}{3} - \frac{1}{3m}$.

2.4 Minimization Problem Example: Bin Packing

Bin packing is extensively studied within the context of offline and online approximation algorithms. Like the makespan problem (even for identical machines), it is an NP-hard optimization problem. In fact, the hardness of both makespan and bin packing is derived by a reduction from the subset sum problem. In the basic bin packing problem, we are given a sequence of items described by their weights $(x_1, x_2, \ldots, x_n)$ such that $x_i \in [0, 1]$. We have an unlimited supply of bins each of unit weight capacity. The goal is to pack all items in the *smallest* number of bins. Formally, it is stated as

Bin Packing

Input: $(x_1, x_2, \ldots, x_n)$; x_i is the weight of an item i.

Output: $\sigma : \{1, 2, \ldots, n\} \to \{1, 2, \ldots m\}$ for some integer m.

Objective: To find σ so as to minimize m subject to the constraints $\sum_{j:\sigma(j)=i} x_j \leq 1$ for each $i \in [m]$.

In this section, we shall analyze the competitive ratios of the following three online algorithms: $NextFit$, $FirstFit$, and $BestFit$.

- $NextFit$: if the newly arriving item does not fit in the *most recently opened* bin, then open a new bin and place the new item in that bin. See Algorithm 2 for pseudocode.

- $FirstFit$: find the first bin among *all opened bins* that has enough remaining space to accommodate the newly arriving item. If such a bin exists, place the new item there. Otherwise, open a new bin and place the new item in the new bin. See Algorithm 3 for pseudocode.

- $BestFit$: find a bin among *all opened bins* that has *minimum* remaining space among all bins that have enough space to accommodate the newly arriving item. If there are no bins that can accommodate the newly arriving item, open a new bin and place the new item in the new bin. See Algorithm 4 for pseudocode.

The algorithms $FirstFit$ and $BestFit$ are greedy in the sense that they will never open a new bin unless it is absolutely necessary to do so. The difference between these two algorithms is in how they break ties when there are several existing bins that could accommodate the new item: $FirstFit$ simply picks the first such bin, while $BestFit$ picks the bin that would result in tightest possible packing. The algorithm $NextFit$ is not greedy — it always considers only the most recently opened bin, and does not check any of the older bins.

The simplest algorithm to analyze is $NextFit$ so we start with it. Later we introduce the weighting technique that is used to analyze both $FirstFit$ and $BestFit$ (in fact, simultaneously).

Theorem 2.4.1.

$$\rho(NextFit) \leq 2.$$

Algorithm 2 The *NextFit* algorithm

```
procedure NextFit
    $m \leftarrow 0$                                  ▷ total number of opened bins so far
    $R \leftarrow 0$                                  ▷ remaining space in the most recently opened bin
    while $j \le n$ do
        if $x_j < R$ then
            $m \leftarrow m + 1$
            $R \leftarrow 1 - x_j$
        else
            $R \leftarrow R - x_j$
        $\sigma(j) \leftarrow m$
        $j \leftarrow j + 1$
```

Algorithm 3 The *FirstFit* algorithm

```
procedure FirstFit
    $m \leftarrow 0$                                  ▷ total number of opened bins so far
    $R$ — a map keeping track of remaining space in all opened bins
    while $j \le n$ do
        $flag \leftarrow False$
        for $i = 1$ to $m$ do
            if $x_j < R[i]$ then
                $R[i] \leftarrow R[i] - x_j$
                $\sigma(j) \leftarrow i$
                $flag \leftarrow True$
                break
        if $flag = False$ then
            $m \leftarrow m + 1$
            $R[m] \leftarrow 1 - x_j$
            $\sigma(j) \leftarrow m$
        $j \leftarrow j + 1$
```

Proof. Define $B[i] = 1 - R[i]$, which keeps track of how much weight is occupied by bin i. Assume for simplicity that $NextFit$ created an even number of bins, i.e., m is even. Then we have $B[1] + B[2] > 1$, since the first item of bin 2 could not fit into the remaining space of bin 1. Similarly we get $B[2i-1] + B[2i] > 1$ for all $i \in \{1, \dots, m/2\}$. Adding all these inequalities, we have

$$\sum_{i=1}^{m/2} B[2i-1] + B[2i] > m/2.$$

Now, observe that the left hand side is simply $\sum_{j=1}^{n} w_j$ and that $OPT \ge \sum_{j=1}^{n} w_j$. Combining these observations with the above inequality we get $OPT > m/2 = NextFit/2$. Therefore, we have $NextFit < 2OPT$. □

Next, we show that $\rho(NextFit) \ge 2$. Fix arbitrary small $\epsilon > 0$ such that $n := 1/\epsilon \in \mathbb{N}$. The input will consist of $3n$ items. The first $2n$ items consist of repeating pairs $1-\epsilon, 2\epsilon$. The remaining n items are all ϵ. Thus, the input looks like this: $1-\epsilon, 2\epsilon, 1-\epsilon, 2\epsilon, \dots, 1-\epsilon, 2\epsilon, \epsilon, \dots, \epsilon$, where the dot dot dots indicate that the corresponding pattern repeats n times. Observe that $NextFit$ on

Algorithm 4 The *BestFit* algorithm

```
procedure BestFit
    $m \leftarrow 0$                                          ▷ total number of opened bins so far
    $R$ — a map keeping track of remaining space in all opened bins
    while $j \leq n$ do
        $ind \leftarrow -1$
        for $i = 1$ to $m$ do
            if $x_j < R[i]$ then
                if $ind = -1$ or $R[i] < R[ind]$ then
                    $ind \leftarrow i$
        if $ind = -1$ then
            $m \leftarrow ind \leftarrow m + 1$
            $R[m] \leftarrow 1$
        $\sigma(j) \leftarrow ind$
        $R[ind] \leftarrow R[ind] - x_j$
        $j \leftarrow j + 1$
```

this instance uses at least $2n$ bins, since the repeating pattern of pairs $1-\epsilon, 2\epsilon$ forces the algorithm to use a new bin on each input item ($1-\epsilon+2\epsilon = 1+\epsilon > 1$). An optimal algorithm can match items of weight $1-\epsilon$ with items of weight ϵ for n bins total, and the remaining n items of weight 2ϵ can be placed into $(2\epsilon)n = (2\epsilon)/\epsilon = 2$ bins. Thus, we have $OPT \leq n+2$ whereas $NextFit \geq 2n$. This means that $\rho(NextFit) \geq 2n/(n+2) \to 2$ as $n \to \infty$.

The weighting technique is a general technique for analyzing *asymptotic* competitive ratios of algorithms for the bin packing problem. The idea behind the weighting technique is to define a weight function $w : [0,1] \to \mathbb{R}$ with the three properties described below. But first we need a few more definitions. Let ALG be an algorithm that we want to analyze. Fix an input sequence $x_1, \ldots, x_n$ and suppose ALG uses m bins. Define S_i to be the set of indices of items that were placed in bin i by ALG, and extend the weight function to index sets $S \subseteq [n]$ as $w(S) := \sum_{i \in S} w(x_i)$. Now, we are ready to state the properties:

1. $w(x) \geq x$;
2. there exist an absolute constant $k_0 \in \mathbb{N}$ (independent of input) and numbers $\beta_j \geq 0$ (dependent on input) such that for all $j \in [m]$ we have $w(S_j) \geq 1 - \beta_j$ and $\sum_{j=1}^m \beta_j \leq k_0$;
3. for all $k \in \mathbb{N}, y_1, \ldots, y_k \in [0,1]$ we have $\sum_{i=1}^k y_i \leq 1$ implies that $\sum_{i=1}^k w(y_i) \leq \gamma$.

In the above $\gamma \geq 1$ is a parameter. If such a weight function w exists for a given ALG and with a given γ then we have $\rho(ALG) \leq \gamma$. This is easy to see, since by property 2 we have

$$w([n]) = \sum_{i=1}^n w(x_i) = \sum_{i=1}^m w(S_i) \geq \sum_{i=1}^m (1-\beta_i) = m - \sum_{i=1}^m \beta_i \geq m - k_0.$$

Let Q_i denote the indices of items that are placed in bin i by OPT and suppose that OPT uses m' bins. Then by property 3, we have

$$w([n]) = \sum_{i=1}^{m'} w(Q_i) \leq \gamma m'.$$

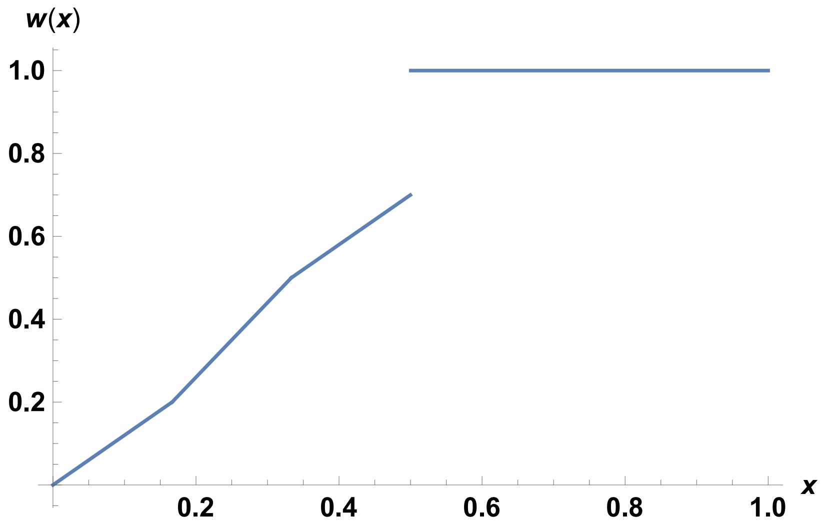

Figure 2.2: The graph of the weight function w used to analyze the competitive ratio of the $FirstFit$ and $BestFit$ algorithms.

Combining the above two inequalities we get that $\gamma m' \geq m - k_0$. That is $ALG \leq \gamma\,\mathrm{OPT} + k_0$, i.e., $\rho(ALG) \leq \gamma$.

We use the weighting technique to prove the following.

Theorem 2.4.2.

$$\rho(FirstFit) \leq 17/10; \qquad \rho(BestFit) \leq 17/10.$$

The proof follows by verifying the above properties for the following weight function w for $FirstFit$ and $BestFit$:

$$w(x) = \begin{cases} \frac{6}{5}x & \text{for } 0 \leq x \leq \frac{1}{6}, \\ \frac{9}{5}x - \frac{1}{10} & \text{for } \frac{1}{6} < x \leq \frac{1}{3}, \\ \frac{6}{5}x + \frac{1}{10} & \text{for } \frac{1}{3} < x \leq \frac{1}{2}, \\ 1 & \text{for } \frac{1}{2} < x \leq 1. \end{cases}$$

The first property is easy to check and becomes obvious by looking at the graph of the function w shown in Figure 2.2.

The following lemma establishes the third property. The proof of this lemma is left as an instructive exercise that can be done by a case analysis.

Lemma 2.4.3. *For the w function defined above we have for all $k \in \mathbb{N}$ and all $y_1, \ldots, y_k \in [0,1]$*

$$\textit{if } \sum_{i=1}^{k} y_i \leq 1 \textit{ then } \sum_{i=1}^{k} w(y_i) \leq 1.7.$$

Thus, it is left to verify the second property for $FirstFit$ and $BestFit$. Recall that our goal is to show that for every bin i we have $w(S_i) \geq 1 - \beta_i$, such that the sum of β_i converges. Observe that the solutions constructed by $FirstFit$ and $BestFit$ cannot contain two bins that are at most half full (i.e., $R[i] \geq 1/2$) — if two such bins are present, then why weren't the items from the later bin inserted into an earlier bin? Suppose that bin i is the unique bin with $R[i] \geq 1/2$, then we can set $\beta_i = 1$ to guarantee that this bin satisfies the second property. This adds at most 1 to $\sum_j \beta_j$. Thus, we can perform the entire argument with respect to bins that are more than half full. Therefore, from now on we assume that $R[i] < 1/2$ for all $i \in [m]$.

To complete the argument we introduce the idea of *coarseness*. Coarseness of bin i, denoted by α_i, is defined as the maximum remaining space in an earlier bin, i.e., there is an earlier bin $j < i$ with remaining space $R[j] = \alpha_i$. Observe that $0 = \alpha_1 \leq \alpha_2 \leq \cdots \leq \alpha_m < 1/2$. The proof of the second property follows from the following lemma.

Lemma 2.4.4. *1. If $R[i] \leq \alpha_i$ then $w(S_i) \geq 1$, i.e., we can set $\beta_i = 0$.*

2. If $R[i] > \alpha_i$ then $w(S_i) \geq 1 - \frac{9}{5}(R[i] - \alpha_i)$, i.e., we can set $\beta_i = \frac{9}{5}(R[i] - \alpha_i)$.

Proof. Let $y_1, \ldots, y_k$ be the weights of the items in bin i. Recall that $B[i] = \sum_{j=1}^k y_j$. By renaming variables, if necessary, we can assume that $y_1 \geq y_2 \geq \cdots \geq y_k$.

1. If $k = 1$ then $y_1 > 1/2$ and $w(y_1) = 1$ and therefore $w(S_i) \geq 1$. Thus, we can assume that $k \geq 2$. In particular, we must have $y_1 \geq y_2 \geq \alpha_i$ (otherwise, each of the two items could have been placed into an earlier bin corresponding to the coarseness α_i). We consider several cases depending on the range of α_i.

   Case 1: $\alpha_i \leq 1/6$. Then we have $B[i] = 1 - R[i] \geq 1 - \alpha_i \geq 5/6$. Observe that on interval $[0, 1/2)$ the slope of w is at least $6/5$, therefore we have $w(S_i)/B[i] \geq 6/5$, so $w(S_i) \geq (6/5)B[i] \geq (6/5)(5/6) = 1$.

   Case 2: $1/3 < \alpha_i < 1/2$. Since $y_1 \geq y_2 > \alpha_i$, we have $y_1, y_2 > 1/3$, so $w(y_1) + w(y_2) \geq 2W(1/3) = 2((9/5)(1/3) - 1/10) = 1$.

   Case 3: $1/6 < \alpha_i \leq 1/3$.

   Subcase (a): $k = 2$. If $y_1, y_2 \geq 1/3$ then it is similar to Case 2 above. Similarly, note that both items cannot be less than 1/3. Then it only remains to consider $y_1 \geq 1/3 > y_2 > \alpha_i$. Thus, we have

   $$w(y_1) + w(y_2) = (6/5)y_1 + 1/10 + (9/5)y_2 - 1/10 = (6/5)(y_1 + y_2) + (3/5)y_2.$$

   Since $y_1 + y_2 \geq 1 - \alpha_i$ and $y_2 > \alpha_i$ we have $w(y_1) + w(y_2) \geq (6/5)(1 - \alpha_i) + 3/5\alpha_i = 1 + 1/5 - 3/5\alpha_i \geq 1$, where the last inequality follows since $\alpha_i \leq 1/3$.

   Subcase (b): $k > 2$. As in subcase (a) if $y_1, y_2 \geq 1/3$ then we are done. If $y_1 < 1/3$ then we have

   $$\begin{aligned} w(y_1) + w(y_2) + \sum_{j=3}^k w(y_j) &= (9/5)(y_1 + y_2) - (1/5) + (6/5)\sum_{j=3}^k y_j \\ &\geq (6/5)B[i] + (3/5)(y_1 + y_2) - (1/5) \\ &= (6/5)(1 - \alpha) + (3/5)(2\alpha) - 1/5 = 1. \end{aligned}$$

   If $y_1 \geq 1/3 > y_2$ then we have

   $$\begin{aligned} w(y_1) + w(y_2) + \sum_{j=3}^k w(y_j) &= (6/5)y_1 + (1/10) + (9/5)y_2 - (1/10) + \sum_{j=3}^k w(y_j) \\ &= (6/5)(y_1 + y_2) + (3/5)y_2 + (6/5)\sum_{j=3}^k y_j \\ &\geq (6/5)(1 - \alpha) + (3/5)\alpha = 1 + (1/5) - 3/5\alpha \geq 1. \end{aligned}$$

2. We have $B[i] = 1 - R[i] = 1 - \alpha_i + (R[i] - \alpha_i)$. Consider $z_1 = y_1 + (R[i] - \alpha_i)/2$ and $z_2 = y_2 + (R[i] - \alpha_i)/2$. Then we have $z_1 + z_2 + \sum_{j=3}^{k} y_k = 1 - \alpha_i$. Then by the first part of this theorem, we have $w(z_1) + w(z_2) + \sum_{j=3}^{k} w(y_k) \geq 1$. Note that $w(z_1) + w(z_2) \geq w(y_1) + w(y_2) + (9/5)(R[i] - \alpha_i)$ since the slope of w on this range does not exceed 9/5. Combining the two inequalities we get $w(y_1) + w(y_2) + (9/5)(R[i] - \alpha_i) + \sum_{j=3}^{k} w(y_k) \geq 1$. Collecting the $w(y_j)$ together we get $w(S_i) \geq 1 - (9/5)(R[i] - \alpha_i)$.

□

It is just left to verify that $\sum_{i=1}^{m} \beta_i$ is bounded by an absolute constant. From Lemma 2.4.4 we see that those bins with $R[i] \leq \alpha_i$ do not contribute anything to this sum. Thus, we assume that we only deal with bins such that $R[i] > \alpha_i$ from now on. Observe that for such bins we have

$$\alpha_i \geq R[i-1] = \alpha_{i-1} + (R[i-1] - \alpha_{i-1}) = \alpha_{i-1} + \frac{5}{9}\beta_{i-1}.$$

Alternatively, we write $\beta_{i-1} \leq \frac{9}{5}(\alpha_i - \alpha_{i-1})$. Thus, we have

$$\sum_{i=1}^{m-1} \beta_i = \frac{9}{5}\sum_{i=2}^{m}(\alpha_i - \alpha_{i-1}) = \frac{9}{5}(\alpha_m - \alpha_0) < \frac{9}{5} \cdot \frac{1}{2} < 1.$$

Since $\beta_m \leq 1$ and due to the fact that we have ignored a possible bin that is at most half full, we have established that

$$\sum_{j=1}^{m} \beta_j \leq 3.$$

This finishes the analysis of *FirstFit* and *BestFit*.

2.5 Competitive Ratio for Maximization Problems

As defined, competitive ratios and approximation ratios for a minimization problem always satisfy $\rho \geq 1$ and equal to 1 if and only the algorithm is (asymptotically) optimal. Clearly, the closer ρ is to 1, the better the approximation.

For maximization problems, there are two ways to state competitive and asymptotic approximation ratios for a maximization algorithm *ALG*.

1. $\rho(ALG) = \liminf_{OPT(\mathcal{I})\to\infty} \frac{ALG(\mathcal{I})}{OPT(\mathcal{I})}$.

2. $\rho(ALG) = \limsup_{OPT(\mathcal{I})\to\infty} \frac{OPT(\mathcal{I})}{ALG(\mathcal{I})}$.

There is no clear consensus as to which convention to use. In the first definition we always have $\rho \leq 1$. This is becoming more of the standard way of expressing competitive and approximation ratios as the fraction of the optimum value that the algorithm achieves especially if the ratio is a non-parameterized constant. (For example, see the results in Chapter 7 for bipartite matching and Chapter 8) With this convention, we have to be careful in stating results as now an "upper bound" is a negative result and a "lower bound" is a positive result. Using the second definition we would be following the convention for minimization problems where again $\rho \geq 1$ and upper and lower bounds have the standard interpretation for being (respectively) positive and negative results. For both conventions, it is unambiguous when we say, for example, "achieves approximation ratio ..." and "has an inapproximation ratio ...". We will use a mixture of these two conventions, stating constant ratios as fractions while, as in this section and many of the results in Chapter 7), we will use ratios $\rho > 1$ when ρ is expresseed as a function of some input parameter.

2.6 Maximization Problem Example: Time-Series Search

As an example of a deterministic algorithm for a maximization problem, we first consider the following time-series search problem. In this problem one needs to exchange the entire savings in one currency into another, e.g., dollars into euros. Over a period of n days, a new exchange rate p_j is posted on each day $j \in [n]$. The goal is to select a day with maximally beneficial exchange rate and exchange *the entire savings* on that day. If you have not made the trade before day n, you are forced to trade on day n. You might not know n in advance, but you will be informed on day n that it is the last day. Without knowing any side information, an adversary can force any deterministic algorithm to perform arbitrarily badly. There are different variations of this problem depending on what is known about currency rates a-priori. We assume that before seeing any of the inputs, you also have access to an upper bound U and a lower bound L on the p_j, that is $L \le p_j \le U$ for all $j \in [n]$. We also assume no transaction costs. Formally, the problem is defined as follows.

Time-series search
Input: $(p_1, p_2, \ldots, p_n)$ where p_j is the rate for day j meaning that one dollar is equal to p_j euros, $U, L \in \mathbb{R}_{\ge 0}$ such that $L \le p_j \le U$ for all $j \in [n]$
Output: $i \in [n]$
Objective: To compute i so as to maximize p_i.

We introduce a parameter $\phi = U/L$ — the ratio between a maximum possible rate and minimum possible rate. Observe that any algorithm achieves competitive ratio ϕ. Can we do better?

The following deterministic *reservation price* online algorithm is particularly simple and improves upon the trivial ratio ϕ. The algorithm trades all of its savings on the first day that the rate is at least $p^* = \sqrt{UL}$. If the rate is always less than p^*, the algorithm will trade all of its savings on the last day. This algorithm is also known as the reservation price policy or RPP for short.

Algorithm 5 The RESERVATION PRICE POLICY
```
procedure RESERVATION PRICE
    $p^* \leftarrow \sqrt{UL}$
    $flag \leftarrow 0$
    $j \leftarrow 1$ and
    while $j \le n$ and $flag = 0$ do
        if $j < n$ and $p_j \ge p^*$ then
            Trade all savings on day $j$
            $flag \leftarrow 1$
        else if $j = n$ then
            Trade all savings on day $n$
```

Theorem 2.6.1. *Let $\phi = \frac{U}{L}$. The reservation price algorithm achieves competitive ratio $\sqrt{\phi}$ whether n is known or unknown in advance.*

Proof. Consider the case where $p_j < p^*$ for all $j \in [n]$. Then the reservation price algorithm trades all dollars into euros on the last day achieving the objective value $p_n \ge L$. The optimum is $\max_j p_j \le p^* = \sqrt{UL}$. The ratio between the two is

$$\frac{OPT}{ALG} \le \frac{\sqrt{UL}}{L} = \sqrt{\phi}.$$

Now, consider the case where there exists $p_j \ge p^*$ and let j be earliest such index. Then the reservation price algorithm trades all dollars into euros on day j achieving objective value

$p_j \geq \sqrt{UL}$. The optimum is $\max_j p_j \leq U$. The ratio between the two is

$$\frac{OPT}{ALG} \leq \frac{U}{\sqrt{UL}} = \sqrt{\phi}.$$

□

We can also show that $\sqrt{\phi}$ is optimal among all deterministic algorithms for time-series search.

Theorem 2.6.2. *No deterministic online algorithm for time-series search can achieve competitive ratio smaller than $\sqrt{\phi}$ even if n is known.*

Proof. The adversary specifies $p_i = \sqrt{UL}$ for $i \in [n-1]$. If the algorithm trades on day $i \leq n-1$, the adversary then declares $p_n = U$. Thus, OPT trades on day n. In this case, the competitive ratio is $U/\sqrt{UL} = \sqrt{U/L} = \sqrt{\phi}$.

If the algorithm does not trade on day $i \leq n-1$, the adversary declares $p_n = L$. Thus the algorithm is forced to trade on day n with exchange rate L, while OPT trades on an earlier day with exchange rate $\sqrt{UL}$. In this case, the competitive ratio is $\sqrt{UL}/L = \sqrt{U/L} = \sqrt{\phi}$. □

A similar argument presented in the following theorem shows that knowing just ϕ is not sufficient to improve upon the trivial competitive ratio.

Theorem 2.6.3. *Suppose that instead of U and L only $\phi = \frac{U}{L}$ is known to an algorithm a priori. Then competitive ratio of any algorithm for time-series search is at least ϕ.*

Proof. The adversary declares ϕ and presents the input sequence $(p_1, \ldots, p_n)$, where $p_i = 1$ for $i \in [n-1]$ to a reservation price algorithm ALG.

If ALG trades on a day $i \leq [n-1]$, then the adversary declares $p_n = \phi$. In this case, an optimal solution is to trade on day p_n achieving objective value p_n, and the algorithm traded when the exchange rate was 1.

If ALG doesn't trade on day $n-1$ or earlier, then the adversary declares $p_n = 1/\phi$. In this case, an optimal solution is to trade on day 1 (for example) achieving objective value 1, and the algorithm trades on the last day achieving $1/\phi$.

In either case, the adversary can achieve competitive ratio ϕ. □

2.7 Maximization Problem Example: One-Way Trading

A natural generalization of the time-series search is the one-way trading problem. In this problem, instead of requiring the algorithm to trade *all of its savings* on a single day, we allow an algorithm to trade a fraction $f_i \in [0,1]$ of its savings on day i for $i \in [n]$. An additional requirement is that by the end of the last day all of the savings have been traded, that is $\sum_{i=1}^n f_i = 1$. As before, the bounds U and L on exchange rates are known in advance. Also as before, we assume that the algorithm is forced to trade all remaining savings at the specified rate on day n.

One-way currency trading

Input: $(p_1, p_2, \ldots, p_n)$ where p_j is the rate for day j meaning that one unit of savings is equal to p_j units of new currency; $U, L \in \mathbb{R}_{\geq 0}$ such that the rates must satisfy $L \leq p_i \leq U$ for each day i.

Output: $f_1, \ldots f_n \in [0,1]$ where f_i indicates that the fraction f_i of savings are traded on day i and $\sum_{i=1}^n f_i = 1$.

Objective: To compute f so as to maximize $\max_i f_i p_i$; that is, to maximize the aggregate exchange rate.

The ability to trade a fraction of your savings on each day is surprisingly powerful — one can *almost* achieve competitive ratio $\log \phi$ instead of $\sqrt{\phi}$ achievable without this ability. Algorithm 6 shows how this is done. To simplify the presentation we assume that $\phi = 2^k$ for some positive integer k. Consider exponentially spaced reservation prices of the form $L2^i$ where $i \in \{0, 1, \dots, k-1\}$. We refer to $L2^i$ as the ith reservation price. Upon receiving p_1, the algorithm computes index i of the largest reservation price that is exceeded by p_1 and trades $(i+1)/k$ fraction of its savings. This index i is recorded in i^*. For each newly arriving exchange rate p_j we compute index i of the reservation price that is exceeded by p_j. If $i \le i^*$ the algorithm ignores day j. Otherwise, the algorithm trades $(i - i^*)/k$ fraction of its savings on day j and updates i^* to i. Thus, we can think of i^* as keeping track of the best reservation price that has been exceeded so far, and whenever we have a better reservation price being exceeded we trade the fraction of savings proportional to the difference between indices of the two reservation prices. Thus, the algorithm is computing some kind of *mixture of reservation price policies*, and it is called the Mixture of RPPs algorithm.

Algorithm 6 The Mixture of RPPs

```
procedure RESERVATION PRICE
    ▷ $U, L$, and $\phi = U/L = 2^k$ are known in advance
    $i^* \leftarrow -1$
    for $j \leftarrow 1$ to $n$ do
        $i \leftarrow \max\{i \mid L2^i \le p_j\}$
        if $i = k$ then
            $i \leftarrow k - 1$
        if $i > i^*$ then
            Trade fraction $(i - i^*)/k$ of savings on day $j$
            $i^* \leftarrow i$
    Trade all remaining savings on day $n$
```

Theorem 2.7.1. *The competitive ratio of the Mixture of RPPs algorithm is $c(\phi) \log \phi$ where $c(\phi)$ is a function such that $c(\phi) \to 1$ as $\phi \to \infty$.*

Proof. Consider the input sequence $(p_1, \dots, p_n)$ and let $i_1, \dots, i_n$ be the indices of the corresponding reservation prices. That is, i_j is the largest index such that $L2^{i_j} \le p_j$ for all $j \in [n]$. Let ℓ be the day number with the highest exchange rate, that is $p_\ell = \max\{p_j\}$. Clearly, $OPT = p_\ell \le L2^{i_\ell + 1}$.

Note that the i^*s form a non-decreasing sequence during the execution of the algorithm. Consider only those values of i^* that actually change value, and let $i_0^* < i_1^* < \cdots < i_p^*$ denote that sequence of values. We have $i_0^* = -1$ and $i_p^* = i_\ell$. The algorithm achieve at least the following value of the objective function:

$$\sum_{j=1}^{p} \frac{i_j^* - i_{j-1}^*}{k} L2^{i_j^*} + \frac{k - i_\ell}{k} L,$$

where the first term is the lower bound on the contribution of trades until day ℓ and the second term is the contribution of trading the remaining savings on the last day.

In order to bound the first term, we note that if we wish to minimize $\sum_{j=1}^{p} (i_j^* - i_{j-1}^*) 2^{i_j^*}$ (E) over all increasing sequences i_j^* with $i_0^* = -1$ and $i_p^* = i_\ell$ then we have $i_j^* = j - 1$. That is the unique minimizer of expression (E) is the sequence $-1, 0, 1, 2, \dots, i_\ell$, i.e., it doesn't skip any values. In this case we have $\sum_{j=1}^{p} (i_j^* - i_{j-1}^*) 2^{i_j^*} = \sum_{j=0}^{i_\ell} 2^j = 2^{i_\ell + 1} - 1$. Why is this a minimizer? We will show that an increasing sequence that skips over a particular value v cannot be a minimizer. Suppose that you have a sequence such that $i^*_{j-1} < v < i^*_j$ and consider the jth term in (E) corresponding to this sequence. It is $(i^*_j - i^*_{j-1})2^{i^*_j} = (i^*_j - v + v - i^*_{j-1})2^{i^*_j} = (i^*_j - v)2^{i^*_j} + (v - i^*_{j-1})2^{i^*_j} > (i^*_j - v)2^{i^*_j} + (v - i^*_{j-1})2^v$ — that is if we change our sequence to include v we strictly decrease the value of (E). Thus, the unique minimizing sequence is the one that doesn't skip any values.

From the above discussion we conclude that we can lower bound ALG as follows:

$$ALG \geq \frac{2^{i_\ell+1}-1}{k}L + \frac{k-i_\ell}{k}L.$$

Finally, we can bound the competitive ratio:

$$\frac{OPT}{ALG} = \frac{L2^{i_\ell+1}}{(2^{i_\ell+1}-1)L/k + (k-i_\ell)L/k} = k\frac{2^{i_\ell+1}}{2^{i_\ell+1}+k-i_\ell-1}.$$

The worst-case competitive ratio is obtained by maximizing the above expression. We can do so analytically (taking derivatives, equating to zero, etc.), which gives $i_\ell = k - 1 + 1/\ln(2)$. Thus, the competitive ratio is $\log\phi = k$ times a factor that is slightly larger than 1 and approaching 1 as $k \to \infty$. □

We saw that with time-series search knowing U or L was crucial and knowing just ϕ was not enough. It turns out that for one-way trading one can prove a similar result to the above assuming that the algorithm only knows ϕ and doesn't know U or L. One of the exercises at the end of this chapter deals is dedicated to this generalization. Another generalization is that we don't need to assume that ϕ is a power of 2. Proving this is tedious and has low pedagogical value, thus we state it here without a proof.

2.8 Exercises

1. Consider the makespan problem for temporary jobs, where now each job has both a load p_j and a duration d_j. When a job arrives, it must be scheduled on one of the m machines and remains on that machine for d_j time units after which it is removed. The makespan of a machine is the maximum load of the machine at any point in time. As for permanent jobs, we wish to minimize (over all machines) the maximum makespan.
   Show that the greedy algorithm provides a 2-approxmimation for this problem.

2. Suppose that in the time-series search problem the algorithm only knows L beforehand. Does there exist a deterministic algorithm with competitive ratio better than ϕ?

3. Suppose that in the one-way trading problem the algorithm only knows L beforehand. Does there exist a deterministic algorithm with competitive ratio better than ϕ?

4. (HARD) Suppose that in the one-way trading problem the algorithm only knows ϕ beforehand. Design a deterministic algorithm with competitive ratio as close to $\log\phi$ as possible.

5. Prove Lemma 2.4.3.

6. Which properties of $FirstFit$ and $BestFit$ were needed for the proof of Theorem 2.4.2? Can you find any other algorithms with those properties? That is find an algorithm other than $FirstFit$ and $BestFit$ such that the proof of Theorem 2.4.2 applies to the new algorithm *without any modifications.*

7. Prove that $\rho(FirstFit) \geq 1.7$ and that $\rho(BestFit) \geq 1.7$ (give an adversarial strategy).
8. Prove that no online algorithm can achieve competitive ratio better than 4/3 for the Bin Packing problem.

2.9 Historical Notes and References

Request-answer games were introduced in Ben-David et al [6]. As mentioned, request-answer games serve as a very general abstract model that applies to almost all optimization problems problems that we will be considering in this text. Moreover, as we will see in Chapter 3, within this model the different types of adversaries (with respect to randomized online algorithms) can be compared.

The makespan problem for identical machines was studied by Graham [29, 30]. These papers present online and offline greedy approximation algorithms for the makespan problem on identical machines as well as presenting some surprising anomolies. The papers preceed Cook's seminal paper [15] introducing NP completeness but still Graham conjectures that it will not be possible to have an efficient optimal algorithm for this problem. This work also preceeds the explicit introduction of competitive analysis by Sleator and Tarjan [46] and does not emphasize the online nature of the greedy algorithm but still the $2 - \frac{1}{m}$ appears to be the first approximation and competitive bound to be published.

The makespan problem belongs to a wider class of scheduling problems called load balancing problems, including other performance measures (e.g., where the laod on a machine is its L_p norm for $p \geq 1$) as well as other measures, and scheduling and routing problems. In particular, the so-called "Santa Claus problem" [4] considers the $max - min$ performance measure for scheduling jobs the unrelated machines model (see Chapter 4. The name of the problem derives from the motivation of distributing n presents amongst m children (where now $p_{i,j}$ is the value of the j^{th} present when given to the i^{th} child) so as to maximize the the value of the least happy child.

Competitive algorithms for the time series and one-way trading problems were introduced and anlayzed in El-Yaniv-et al [23].

The bin packing problem is a classic NP-hard optimzation problem and one that continues to be a subject of interest for both the offline and online settings. Early work on the bin packing problem popularized the field of approximation algorithms although Graham's makespan results preceeded the results for bin packing. This chapter was restricted to the classical one-dimensional bin packing problem. Until recently, the most precise results for the first fit (FF) and best fit (BF) online bin packing algorithms appeared in Johnsoni et al [35] following an earlier conference version by Garey,Graham and Ullman [27] which iteself was preceded by a technical report for first fit by Ullman [47]. Namely, these results showed the competitive ratio to be 1.7. After more than 40 years, the bounds for FF and BF were proven to be *strict* competitive ratios for FF and BF by Dósa and Sgall ([19] and [20]). It what might be the first explicit study of competitive anlayis (before the term was introduced), Yao [52] provided an improved online algorithm called *refined first fit* with competitive ratio $\frac{5}{3}$. Furthermore, Yao gave the first negative result for competitive analysis showing that no online bin packing algorithm can have a competitve ratio better than $\frac{3}{2}$. Since there early bin-packing online algorithms, there have been a numebr of improvements in the competitive ratio based on the Harmonic algorithm of Lee and Lee [41]; in partitcular, the current best ratio is 1.57829 due to Balogh et al [3]. A comprehensive review of the many papers and ideas leading up to the current best ratio can be found in [3]. This upper bound can be compared to 1.54278, the current best lower bound in Balogh et al [3].

Given the applications and historical interest in bin packing, it is not surprising that there are many varants of the problem. The most important variant is arguably the two dimensional problem, which itself comes in different versions; for example, can the items be rotated or must they fit according to the given axis parallel orientation. The two dimensional problem was introduced in Chung, Garey and Johmson [45]. As in the case of one-dimensional online bin packing, there is also a substantial sequence of results for axis parallel 2-dimensional bin packing leading up to the current best 2.5545 competitive ratio in Han et al [32] whereas the most recent lower bound, due to Epstein [25], is slightly above 1.91.

Chapter 3

Randomized Online Algorithms

The central theme of this chapter is how much randomness can help in solving an online problem. To answer this question, we need to extend the notion of competitive ratio to randomized algorithms. Measuring the performance of randomized algorithms is not as straightforward as measuring the performance of deterministic algorithms, since randomness allows for different kinds of adversaries. We look at the classical notions of **oblivious**, **adaptive online**, and **adaptive offline** adversaries, and explore relationships between them. We cover Yao's minimax theorem, which is a useful technique for proving lower bounds against an oblivious adversary. Lastly, we briefly discuss some issues surrounding randomness as an expensive resource and the topic of derandomization.

3.1 Randomized Online Algorithm Template

We recall that according to our convention random variables are denoted by capital letters and particular outcomes of random variables are denoted by small case letters. For example, suppose that B is the Bernoulli random variable with parameter p. Then B is 1 with probability p and 0 with probability $1-p$, and when we write B the outcome hasn't been determined yet. When we write b we refer to the outcome of sampling from the distribution of B, thus b is fixed to be either 0 or 1. In other words B is what you know about the coin before flipping it, and b is what you see on the coin after flipping it once.

A randomized online algorithm generalizes the deterministic paradigm by allowing the decision in step 5 of the template to be a randomized decision. We view the algorithm as having access to an infinite tape of random bits. We denote the contents of the tape by R, i.e., $R_i \in \{0,1\}$ for $i \geq 1$, and the R_i are distributed uniformly and independently of each other.

```
   Randomized Online Algorithm Template
1: $R \leftarrow$ infinite tape of random bits
2: On an instance $I$, including an ordering of the data items $(x_1, \ldots, x_n)$:
3: $i := 1$
4: While there are unprocessed data items
5:     The algorithm receives $x_i$ and makes an irrevocable randomized decision $D_i := D_i(x_1, \ldots, x_i, R)$ for $x_i$
       (based on $x_i$, all previously seen data items, and $R$).
6:     $i := i + 1$
7: EndWhile
```

Remark 3.1.1. You might wonder if having access to random bits is enough. After all, we often want random variables distributed according to more complicated distributions, e.g., Gaussian with parameters μ and σ. Turns out that you can model any reasonable random variable to any desired accuracy with access to R only. For example, if you need a Binomial random variable with parameters $1/2$ and n, you can write a subprocedure that returns $\sum_{i=1}^{n} R_i$. If you need a new independent sample from that distribution, you can use fresh randomness from another part of the string R. This can be done for all other standard distributions — Bernoulli with parameter p (how?), Binomial with parameters p and n, exponential, Gaussian, etc. We will often skip the details of how to obtain a particular random variable from R and simply assume that we have access to the corresponding subroutine.

Note that the decision in step 4 is now a function not only of all the previous inputs but also of randomness R. Thus each decision D_i is a random variable. However, if we fix R to be particular infinite binary string $r \in \{0,1\}^{\mathbb{N}}$, each decision becomes deterministic. This way, we can view an online randomized algorithm ALG as a distribution over deterministic online algorithms ALG_r indexed by randomness r. Then ALG samples r from $\{0,1\}^{\mathbb{N}}$ uniformly at random and runs ALG_r. This viewpoint is essential for the predominant way to prove inappoximation results for randomized algorithm, namely the use of the von Neumann-Yao principle.

3.2 Types of Adversaries

For this section we recall the view of an execution of an online algorithm as a game between an adversary and the algorithm. In the deterministic case, there is only one kind of adversary. In the randomized case, we distinguish between three different kinds of adversaries: **oblivious**, **adaptive offline**, and **adaptive online**, depending on the information that is available to the adversary when it needs to create the next input item.

Oblivious adversary: this is the weakest kind of an adversary that only knows the (pseudocode of the) algorithm, but not the particular random bits r that are used by the algorithm. The adversary has to come up with the input sequence $x_1, x_2, \ldots, x_n$ in advance — before learning any of the decisions made by the online algorithm on this input. Thus, the oblivious adversary knows the *distribution* of $D_1, D_2, \ldots, D_n$, but it doesn't know which particular decisions $d_1, d_2, \ldots, d_n$ are going to be taken by the algorithm. Let $OBJ(x_1, \ldots, x_n, d_1, \ldots, d_n)$ be the objective function. The performance is measured as the ratio between the expected value of the objective achieved by the algorithm to the offline optimum on $x_1, \ldots, x_n$. More formally it is

$$\frac{\mathbb{E}_{D_1,\ldots,D_n}\left(OBJ(x_1,\ldots,x_n,D_1,\ldots,D_n)\right)}{OPT(x_1,\ldots,x_n))}.$$

Observe that we don't need to take the expectation of the optimum, because input items $x_1, \ldots, x_n$ are *not random.*

Adaptive offline adversary: this is the strongest kind of an adversary that knows the (pseudocode of the) algorithm and its online decisions, but not R. Thus, the adversary creates the first input item x_1. The algorithm makes a decision D_1 and the adversary learns the outcome d_1 prior to creating the next input item x_2. We can think of the input items as being defined recursively $x_i := x_i(x_1, d_1, \ldots, x_{i-1}, d_{i-1})$. After the entire input sequence is created we compare the performance of the algorithm to that of an optimal offline algorithm that knows the entire sequence $x_1, \ldots, x_n$ in advance. More formally it is

$$\frac{\mathbb{E}_{D_1,\ldots,D_n}\left(OBJ(x_1,\ldots,x_n,D_1,\ldots,D_n)\right)}{\mathbb{E}_{D_1,\ldots,D_n}(OPT(x_1,\ldots,x_n))}.$$

Observe that we have to take the expectation of the optimum in this case, because input items $x_1, \ldots, x_n$ are random as they depend on $D_1, \ldots, D_n$ (implicit in our notation).

Adaptive online adversary: this is an intermediate kind of an adversary that creates an input sequence and an output sequence adaptively. As before the adversary knows the (pseudocode of the) algorithm, but not R. The adversary creates the first input item x_1 and makes its own decision on this item d'_1. The algorithm makes a random decision D_1, the outcome d_1 of which is then revealed to the adversary. The adversary then comes up with a new input item x_2 and its own decision d'_2. Then the algorithm makes a random decision D_2, the outcome d_2 of which is then revealed to the adversary. And so on. Thus, the order of steps is as follows: $x_1, d'_1, d_1, x_2, d'_2, d_2, x_3, d'_3, d_3, \ldots$ We say that the adaptive online adversary can create the next input item based on the previous decisions of the algorithm, but *it has to serve this input item immediately itself.* The performance of an online algorithm is measured by the ratio of the objective value achieved by the adversary versus the objective value achieved by the algorithm. More formally, it is

$$\frac{\mathbb{E}_{D_1,\ldots,D_n}\left(OBJ(x_1,\ldots,x_n,D_1,\ldots,D_n)\right)}{\mathbb{E}_{D_1,\ldots,D_n}\left(OBJ(x_1,\ldots,x_n,d'_1,\ldots,d'_n)\right)}.$$

Observe that we have to take the expectation of the objective value achieved by the adversary, since both input items $x_1, \ldots, x_n$ and adversary's decisions $d'_1, \ldots, d'_n$ depend on random decisions of the algorithm $D_1, \ldots, D_n$ (implicit in our notation).

Based on the above description one can easily define competitive ratios for these different kinds of adversaries for both maximization and minimization problems (using lim infs and lim sups in the obvious way). We denote the competitive ratio achieved by a randomized online algorithm ALG with respect to the oblivious adversary, adaptive offline adversary, and adaptive online adversary by $\rho_{\text{OBL}}(ALG), \rho_{\text{ADOFF}}(ALG)$, and $\rho_{\text{ADON}}(ALG)$, respectively.

The most popular kind of adversary in the literature is the oblivious adversary. When we analyze randomized online algorithms we will assume the oblivious adversary unless stated otherwise. The oblivious adversary often makes the most sense from a practical point of view, as well. This happens, when performance of the algorithm is independent of the input. We already saw it for the ski rental problem. Whether you decide to buy or rent skis should (logically) have no affect on the weather — this is precisely modelled by an oblivious adversary. However, there are problems for which decisions of the algorithm can affect the behaviour of the future inputs. This happens, for example, for paging. Depending on which pages are in the cache, the future pages will either behave as cache misses or as cache hits. In addition, one can write programs that alter their behaviour completely depending on cache miss or cache hit. One real-life example of such (nefarious) programs are Spectre and Meltdown that use cache miss information together with speculative execution to gain read-only access to protected parts of computer memory. Thus, there are some real-world applications which are better modelled by adaptive adversaries, since decisions of the algorithm can alter the future input items.

3.3 Relationships between Adversaries

We start with a basic observation that justifies calling oblivious, adaptive offline, and adaptive online adversaries as weak, strong, and intermediate, respectively.

Theorem 3.3.1. *For a minimization problem and a randomized online algorithm ALG we have*

$$\rho_{OBL}(ALG) \le \rho_{ADON}(ALG) \le \rho_{ADOFF}(ALG).$$

An analogous statement is true for maximization problems.

The following theorem says that the adaptive offline adversary is so powerful that any randomized algorithm running against it cannot guarantee a better competitive ratio than the one achieved by deterministic algorithms.

Theorem 3.3.2. *Consider a minimization problem given as a request-answer game. Assume that the set of possible answers/decisions is finite (e.g., Ski Rental) and consider a randomized online algorithm ALG for it. Then there is a deterministic online algorithm ALG' such that*

$$\rho(ALG') \leq \rho_{ADOFF}(ALG).$$

An analogous statement is true for maximization problems.

Proof. We refer to $(x_1, \ldots, x_k, d_1, \ldots, d_k)$ as a *position in the game*, where the x_i are input items, provided by an adversary, and d_i are decisions, provided by an algorithm. We say that a position $(x_1, \ldots, x_k, d_1, \ldots, d_k)$ is *immediately winning for adversary* if $f_k(x_1, \ldots, x_k, d_1, \ldots, d_k) > \rho_{\text{ADOFF}}(ALG) OPT(x_1, \ldots, x_k)$, where f_k is the objective function. We call a position *winning for adversary* if there exists $t \in \mathbb{N}$ and an adaptive strategy of choosing requests such that an immediately winning position is reached *no matter what answers are chosen by an algorithm* within t steps.

Note that the initial empty position cannot be a winning position for the adversary. Suppose that it was, for contradiction. The randomized algorithm ALG is a distribution on deterministic algorithms ALG_z for some $z \sim Z$. If the initial empty position was winning for the adversary, then for every z we would have a sequence of requests and answers (depending on z) such that $ALG_z(I_z) > \rho_{\text{ADOFF}}(ALG) OPT(I_z)$. Taking the expectation of both sides, we get $\mathbb{E}_Z(ALG_Z(I_Z)) > \rho_{\text{ADOFF}}(ALG) \mathbb{E}_Z(OPT(I_Z))$, contradicting the definition of $\rho_{\text{ADOFF}}(ALG)$.

Observe that a position $(x_1, \ldots, x_n, d_1, \ldots, d_n)$ is winning if and only if there exists x_{n+1} such that for all d_{n+1} the position $(x_1, \ldots, x_n, x_{n+1}, d_1, \ldots, d_n, d_{n+1})$ is also winning. Thus, if a position is not winning, then for any new input item x_{n+1} there is a decision d_{n+1} that leads to a position that is also not winning. This precisely means that there is a deterministic algorithm ALG' that can keep the adversary in a non-winning position for as long as needed. Since the game has to eventually terminate, it will terminate in a non-winning position, meaning that after any number t of steps of the game we have $f_t(x_1, \ldots, x_t, d_1, \ldots, d_t) \leq \rho_{\text{ADOFF}} OPT(x_1, \ldots, x_t)$, where d_i are the *deterministic* choices provided by ALG. □

The gap between offline adaptive adversary and online adaptive adversary can be at most quadratic.

Theorem 3.3.3. *Consider a minimization problem and a randomized online algorithm ALG for it. Then*

$$\rho_{ADOFF}(ALG) \leq \left(\rho_{ADON}(ALG)\right)^2.$$

An analogous statement is true for maximization problems.

Proof. Let ADV be an arbitrary adaptive offline adversary against ALG. Let R denote the randomness used by ALG, and let R' be its independent copy. Then we can represent ALG as a distribution on deterministic algorithms ALG_r where $r \sim R$. Let $x(R)$ be requests given by ADV when it runs against ALG_R. Let $d(R)$ denote ALG_R responses when it runs against ADV.

Consider a fixed value of r and the well-defined sequence of requests $x(r)$. In order to avoid ambiguity, we are going to label randomness of ALG by a copy of R, namely R'. Since $ALG_{R'}$ is $\rho_{\text{ADON}}(ALG)$-competitive against any adaptive online adversary, it is also $\rho_{\text{ADON}}(ALG)$-competitive against any oblivious adversary (by Theorem 3.3.1). In particular $ALG_{R'}$ is ρ_{ADON}-competitive against the oblivious adversary that presents request sequence $x(r)$:

$$\mathbb{E}_{R'}(ALG_{R'}(x(r))) \leq \rho_{\text{ADON}}(ALG)OPT(x(r)).$$

Taking the expectation of both sides with respect to r we get

$$\mathbb{E}_R\mathbb{E}_{R'}(ALG_{R'}(x(R))) \leq \rho_{\text{ADON}}(ALG)\mathbb{E}_R(OPT(x(R))). \tag{3.1}$$

Let f denote the objective function. Now, consider a fixed value of r'. Define an adaptive online strategy working against ALG_R that produces a request sequence $x(R)$ and provides its own decision sequence $d(r')$, while ALG_R provides decision sequence $d(R)$. Since ALG is $\rho_{\text{ADON}}(ALG)$-competitive against this adaptive online strategy, we have:

$$\mathbb{E}_R(ALG_R(x(R)) \leq \rho_{\text{ADON}}(ALG)\mathbb{E}_R(f(x(R), d(r'))).$$

Taking the expectation of both sides with respect to r' we get

$$\mathbb{E}_R(ALG_R(x(R)) \leq \rho_{\text{ADON}}(ALG)\mathbb{E}_{R'}\mathbb{E}_R(f(x(R), d(R'))).$$

The right hand side can be written as $\mathbb{E}_{R'}\mathbb{E}_R(f(x(R), d(R'))) = \mathbb{E}_R\mathbb{E}_{R'}(ALG_{R'}(x(R))$. Combining this with (3.1) we get

$$\mathbb{E}_R(ALG_R(x(R))) \leq \rho_{\text{ADON}}(ALG)^2\mathbb{E}(OPT(x(R))).$$

The left hand side is the expected cost of the solution produced by ALG running against *the adaptive offline adversary ADV*. □

In the following section we establish that the gap between ρ_{OBL} and ρ_{ADOFF} (as well as between ρ_{OBL} and ρ_{ADON}) can be arbitrary large.

3.4 How Much Can Randomness Help?

We start by showing that the gap between the best competitive ratio achieved by a randomized algorithm and a deterministic algorithm can be arbitrary large. We begin by fixing a particular gap function $g : \mathbb{N} \to \mathbb{R}$. Consider the following maximization problem:

Modified Bit Guessing Problem

Input: $(x_1, x_2, \ldots, x_n)$ where $x_i \in \{0, 1\}$.
Output: $z = (z_1, z_2, \ldots, z_n)$ where $z_i \in \{0, 1\}$
Objective: To find z such that $z_i = x_{i+1}$ for some $i \in [n-1]$. If such i exists the payoff is $g(n)/(1 - 1/2^{n-1})$, otherwise the payoff is 1.

In this problem, the adversary presents input bits one by one and the goal is to guess the bit arriving in the next time step based on the past history. If the algorithm manages to guess at least one bit correctly, it receives a large payoff of $g(n)/(1 - 1/2^{n-1})$, otherwise it receives a small payoff of 1.

Theorem 3.4.1. *Every deterministic algorithm ALG achieves objective value* 1 *on the Modified Bit Guessing Problem.*

There is a randomized algorithm that achieves expected objective value $g(n)$ against an oblivious adversary on inputs of length n for the Modified Bit Guessing Problem.

Proof. For the first part of the theorem consider a deterministic algorithm ALG. The adversarial strategy is as follows. Present $x_1 = 0$ as the first input item. The algorithm replies with z_1. The adversary defines $x_2 = \neg z_1$. This continues for $n-2$ more steps. In other words, the adversary defines $x_i = \neg x_{i-1}$ for $i = \{2, \ldots, n\}$ making sure that the algorithm does not guess any of the bits. Thus, the algorithm achieves objective function value 1.

Consider the randomized algorithm that selects z_i uniformly at random. The probability that it picks $z_1, \ldots, z_{n-1}$ to be different from $x_2, \ldots, x_n$ in each coordinate is exactly $1/2^{n-1}$. Therefore with probability $1 - 1/2^{n-1}$ it guesses at least one bit correctly. Therefore the expected value of the objective function is at least $g(n)/(1-1/2^{n-1}) \cdot (1-1/2^{n-1}) = g(n)$. □

Corollary 3.4.2. *The gap between ρ_{OBL} and ρ_{ADOFF} can be arbitrarily large.*

Thus, there are problems for which randomness helps a lot. What about another extreme? Are there problems where randomness does not help at all? It turns out "yes" and, in fact, we have already seen such a problem, namely, the One-Way Trading problem.

Theorem 3.4.3. *Let ALG be a randomized algorithm for the One-Way Trading problem. Then there exists a deterministic algorithm ALG' for the One-Way Trading problem such that*

$$\rho(ALG') \leq \rho_{OBL}(ALG).$$

Proof. Recall that ALG is a distribution on deterministic algorithms ALG_R indexed by randomness R. For each r consider the deterministic algorithmALG_r running on the input sequence $p_1, \ldots, p_n$. Let $f_i(r, p_1, \ldots, p_i)$ be the fraction of savings exchanged on day i. We can define the *average* fraction of savings exchanged on day i, where the average is taken over all deterministic algorithms in the support of ALG. That is

$$\widetilde{f_i}(p_1, \ldots, p_i) := \int f_i(r, p_1, \ldots, p_i)\, dr.$$

Observe that $\widetilde{f_i}(p_1, \ldots, p_i) \geq 0$ and moreover

$$\sum_{i=1}^{n} \widetilde{f_i}(p_1, \ldots, p_i) = \sum_{i=1}^{n} \int f_i(p_1, \ldots, p_i)\, dr = \int \sum_{i=1}^{n} f_i(p_1, \ldots, p_i)\, dr = \int 1\, dr = 1.$$

Therefore, $\widetilde{f_i}$ form valid fractions of savings to be traded on n days. The fraction $\widetilde{f_i}$ depends only on $p_1, \ldots, p_i$ and is independent of randomness r. Thus, these fractions can be computed by a deterministic algorithm (with the knowledge of ALG) in the online fashion. Let ALG' be the algorithm that exchanges $\widetilde{f_i}(p_1, \ldots, p_i)$ of savings on day i. It is left to verify the competitive ratio of ALG'. On input $p_1, \ldots, p_n$ it achieves the value of the objective

$$\begin{aligned}\sum_{i=1}^{n} p_i \widetilde{f_i}(p_1, \ldots, p_i) &= \sum_{i=1}^{n} p_i \int f_i(r, p_1, \ldots, p_i)\, dr = \int \sum_{i=1}^{n} p_i f_i(r, p_1, \ldots, p_i)\, dr \\ &= \mathbb{E}_R(ALG_R(p_1, \ldots, p_n)) \geq OPT(p_1, \ldots, p_n)/\rho_{\text{OBL}}(ALG).\end{aligned}$$

□

The Modified String Guessing problem provides an example of a problem where using randomness improves competitive ratio significantly. Notice that the randomized algorithm uses n bits of randomness to achieve this improvement. Next, we describe another extreme example, where a *single bit* of randomness helps improve the competitive ratio.

Proportional Knapsack

Input: $(w_1, \ldots, w_n)$ where $w_i \in \mathbb{R}_{\geq 0}$; W — bin weight capacity, known in advance.
Output: $z = (z_1, z_2, \ldots, z_n)$ where $z_i \in \{0, 1\}$
Objective: To find z such that $\sum_{i=1}^n z_i w_i$ is maximized subject to $\sum_{i=1}^n z_i w_i \leq W$.

In this problem the goal is to pack maximum total weight of items into a single knapsack of weight capacity W (known in advance). Item i is described by its weight w_i. For item i the algorithm provides a decision z_i such that $z_i = 1$ stands for packing item i, and $z_i = 0$ stands for ignoring item i. If an algorithm produces an infeasible solution (that is total weight of packed items exceeds W), the payoff is $-\infty$. Thus, without loss of generality, we assume that an algorithm never packs an item that makes the total weight exceed W. Our first observation is that deterministic algorithms cannot achieve constant competitive ratios.

Theorem 3.4.4. *Let $\epsilon > 0$ be arbitrary and let ALG be a deterministic online algorithm for the Proportional Knapsack problem. Then we have*

$$\rho(ALG) \geq \frac{1-\epsilon}{\epsilon}.$$

Proof. Let $n \in \mathbb{N}$. We describe an adversarial strategy for constructing inputs of size n. First, let $W = n$. Then the adversary presents inputs ϵn until the first time ALG packs such an input. If ALG never packs an input item of weight ϵn, then ALG packs total weight 0, while $OPT \geq \epsilon n$, which leads to an infinitely large competitive ratio.

Suppose that ALG packs $w_i = \epsilon n$ for the first time for some $i < n$. Then the adversary declares $w_{i+1} = n(1-\epsilon) + \epsilon$ and $w_j = 0$ for $j > i + 1$. Therefore ALG cannot pack w_{i+1} since $w_i + w_{i+1} = n + \epsilon > W$. Moreover, packing any of w_j for $j > i+1$ doesn't affect the value of the objective function. Thus, we have $ALG = \epsilon n$, whereas $OPT = w_{i+1} = n(1-\epsilon) + \epsilon$. We get the competitive ratio of $\frac{n(1-\epsilon)+\epsilon}{\epsilon n} \geq \frac{n(1-\epsilon)}{\epsilon n} = \frac{1-\epsilon}{\epsilon}$. □

Next we show that a randomized algorithm, which we call SimpleRandom, that uses only 1 bit of randomness achieves competitive ratio 4. Such 1 bit randomized algorithms have been termed "barely random". Algorithm 7 provides a pseudocode for this randomized algorithm. The algorithm has two modes of operation. In the first mode, the algorithm packs items greedily — when a new item arrives, the algorithm checks if there is still room for it in the bin and if so packs it. In the second mode, the algorithm waits for an item of weight $\geq W/2$. If there is such an item, the algorithm packs it. The algorithm ignores all other weights in the second mode. The algorithm then requires a single random bit B, which determines which mode the algorithm is going to use in the current run.

Algorithm 7 Simple randomized algorithm for Proportional Knapsack

```
procedure SimpleRandom
    Let $B \in \{0,1\}$ be a uniformly random bit        ▷ $W$ is the knapsack weight capacity
    if $B = 0$ then
        Pack items $w_1, \ldots, w_n$ greedily, that is if $w_i$ still fits in the remaining weight knapsack
    capacity, pack it; otherwise, ignore it.
    else
        Pack the first item of weight $\geq W/2$ if there is such an item. Ignore the rest of the items.
```

Theorem 3.4.5.

$$\rho_{OBL}(SimpleRandom) \leq 4.$$

Proof. The goal is to show that $OPT \leq 4\mathbb{E}(\text{SimpleRandom})$ on any input sequence $w_1, \ldots, w_n$. We distinguish two cases.

Case 1: for all $i \in [n]$ we have $w_i < W/2$. Subcase 1(a): $\sum_{i=1}^{n} w_i \leq W$. In this subcase, SimpleRandom running in the first mode packs all of the items. This happens with probability 1/2, thus we have $\mathbb{E}(\text{SimpleRandom}) \geq 1/2 \sum_i w_i$ and $OPT = \sum_i w_i$. Therefore, it follows that $OPT \leq 2\mathbb{E}(\text{SimpleRandom})$ in this subcase. Subcase 1(b): $\sum_i w_i > W$. Consider SimpleRandom running in the first mode again. There is an item that SimpleRandom does not pack in this case. Let w_i be the first item that is not packed. The reason w_i is not packed is that the remaining free space is less than w_i, but we also know that $w_i < W/2$. This means that SimpleRandom has packed total weight at least $W/2$ by the time w_i arrives. Since SimpleRandom runs in the first mode with probability 1/2 we have that $\mathbb{E}(\text{SimpleRandom}) \geq (1/2)(W/2) = W/4 \geq OPT/4$, where the last inequality follows from the trivial observation that $OPT \leq W$. Rearranging we have $OPT \leq 4\mathbb{E}(\text{SimpleRandom})$ in this subcase.

Case 2: there exists $i \in [n]$ such that $w_i \geq W/2$. Consider SimpleRandom running in the second mode: it packs the first w_i such that $w_i \geq W/2$. Since SimpleRandom runs in the second mode with probability 1/2 we have $\mathbb{E}(\text{SimpleRandom}) \geq (1/2)(W/2) = W/4 \geq OPT/4$. Thus, it follows that $OPT \leq 4\mathbb{E}(\text{SimpleRandom})$.

This covers all possibilities. We got that in all cases the competitive ratio of SimpleRandom is at most 4. □

3.5 Derandomization

Randomness is a double-edged sword. On one hand, when we are faced with a difficult problem for which no good deterministic algorithm exists, we do hope that adding randomness would allow one to design a much better (and often simpler) randomized algorithm. On another hand, when we have an excellent randomized algorithm, we hope that we can remove its dependence on randomness, since randomness as a resource is quite expensive. If we can design a deterministic algorithm with the same guarantees (e.g., competitive ratio) as the randomized algorithm ALG, we say that ALG has been "derandomized." Whether all algorithms can be derandomized or not depends on the computational model. For example, we have seen that the general-purpose derandomization is not possible for online algorithms versus an oblivious adversary, but it is possible for online algorithms versus an adaptive offline adversary. In the Turing machine world, it is a big open problem whether all of bounded-error polynomial time algorithms (i.e., the class BPP) can be derandomized or not. Many believe that such derandomization of BPP should be possible. When general-purpose derandomization is not possible, it is still an interesting question to see if derandomization is possible for a particular problem. We saw one such example for the One-Way Trading problem. Derandomization is a big topic and it will come up several times in this book.

In the remainder of this section, we would like to discuss why randomness as a resource can be expensive. Best scientists in human history have argued whether "true randomness" exists, or if randomness is simply a measure of our ignorance. The latest word on the subject is given by quantum mechanics, which says that, indeed, to the best of our understanding of how the world works there are truly random events in the nature. In principle, one could build machines that generate truly random bits based on quantum effects, but there are no cheap commercially available solutions like that at this moment (to the best of our knowledge). Instead, random bit generators implemented on the off-the-shelf devices are pseudo-random. The pseudo-random bits can come from different sources — they can be mathematically generated, or they can be generated from some physical processes, such as coordinates of the latest mouse click on the desktop, or voltage noise in the CPU. Standard programming libraries take some combination of these techniques to produce random-looking bits. How can one detect if the bits that are being generated are truly random or pseudo-random? For that we could write a program, called a test, that receives a string of bits and outputs either "YES" for truly random or "NO" for pseudo-random. The test could be as simple as checking sample moments (mean and variance, for example) of the incoming bits and seeing if it falls not too far from the true moments of the distribution. The test could also compute autocorrelation, and so on. Typically, it is not hard to come up with pseudo-random generators that would "fool" such statistical tests to believe that the bits are truly random. But it is again a big open problem to come up with a pseudo-random generator that would provably fool *all* reasonable tests. Fortunately, typically all we need for a randomized algorithm to run correctly is for the pseudo-random bits to pass statistical tests. This is why Quicksort has an excellent empirical performance even with pseudo-random generators. In addition, even if the bits are not truly random the only side-effect that your program might experience is a slight degradation in performance, which is not critical. However, there are cases where breaking truly random guarantee can result in catastrophic losses. This often happens in security, where using pseudo-random tactics (such as generating randomness based on mouse clicks) introduces a vulnerability in the security protocol. Since this book doesn't deal with the topic of security, we will often permit ourselves to use randomness freely. Nonetheless, we realize that random bits might be expensive and we shall often investigate whether a particular randomized algorithm can be derandomized or not.

3.6 Lower Bound for Paging

In this section we revisit the Paging problem. We shall prove a lower bound on the competitive ratio achieved by any randomized algorithm. In the process, we shall discover a general-purpose technique called Yao's minimax principle. In the following section, we will formally state and discuss the principle. To state the result we need to introduce the *nth harmonic number.*

Definition 3.6.1. The nth Harmonic number, denoted by H_n, is defined as

$$H_n = 1 + \frac{1}{2} + \cdots + \frac{1}{n} = \sum_{i=1}^{n} \frac{1}{i}.$$

An exercise at the end of this chapter asks you to show that $H_n \approx \ln(n)$. Now, we are ready to state the theorem:

Theorem 3.6.1. *Let ALG be a randomized online algorithm for the Paging problem with cache size k. Then we have*

$$\rho_{OBL}(ALG) \geq H_k.$$

Proof. We will first show that there is a distribution on input sequences $x_1, \ldots, x_n$ such that every *deterministic* algorithm achieves average competitive ratio greater than H_k with respect to this distribution. We will then see how this implies the statement of the theorem.

Here is the distribution: pick each x_i uniformly at random from $[k+1]$, independently of all other x_j. For every deterministic algorithm, the expected number of page faults is $k + (n-k)/(k+1) \geq n/(k+1)$ — there are k initial page faults, and there is a $1/(k+1)$ chance of selecting a page not currently in the cache in each step after the initial k steps. Next, we analyze the expected value of OPT. As in Section 1.6, let's subdivide the entire input sequence into blocks $B_1, \ldots, B_T$, where block B_1 is the maximal prefix of $x_1, \ldots, x_n$ that contains k distinct pages, B_2 is obtained in the same manner after removing B_1 from $x_1, \ldots, x_n$, and so on. Arguing as in Section 1.6, we have a bound on $OPT \geq T - 1$. Note that T is a random variable, and $\mathbb{E}(OPT) \geq \mathbb{E}(T) - 1$. Thus, the asymptotic competitive ratio is bounded below by $\lim_{n\to\infty} \frac{n}{(k+1)\mathbb{E}(T)}$. If $|B_i|$ were i.i.d., then we could immediately conclude that $\mathbb{E}(T) = n/\mathbb{E}(|B_1|)$. Unfortunately, the $|B_i|$ are not i.i.d., since they have to satisfy $|B_1| + \cdots + |B_T| = n$. For increasing n, $|B_i|$ start behaving more and more as i.i.d. random variables. Formally, this is captured by the Elementary Renewal Theorem from the theory of renewal processes, which for us implies that the competitive ratio is bounded below by $\lim_{n\to\infty} \frac{n}{(k+1)n/\mathbb{E}(W)} = \mathbb{E}(W)/(k+1)$, where W is distributed as $|B_1|$ in the limit (i.e., $n = \infty$).

Thus, let's consider $n = \infty$ and compute $|B_1|$. Computing $\mathbb{E}(|B_1|)$ is known as *the coupon collector problem*. We can write $|B_1| = Z_1 + \cdots + Z_k + Z_{k+1} - 1$, where Z_i is the number of pages we see before seeing an ith *new* page (see it for the first time). The last term (-1) means that we terminate B_1 one step before seeing $k+1$st new page for the first time. Then $Z_1 = 1$, i.e., any page that arrives first is the new first page. After that, we have the probability $k/(k+1)$ of seeing a page different from the first one in each consecutive step. Therefore, Z_2 is a geometrically distributed random variable with parameter $p_2 = k/(k+1)$, hence $\mathbb{E}(Z_2) = 1/p_2 = (k+1)/k$. Similarly, we get Z_i is geometrically distributed with parameter $p_i = (k-i+2)/(k+1)$, hence $\mathbb{E}(Z_i) = 1/p_i = (k+1)/(k-i+2)$. Thus, we have $\mathbb{E}(|B_1|) = \mathbb{E}(Z_1) + \mathbb{E}(Z_2) + \cdots + \mathbb{E}(Z_k) = 1 + (k+1)/k + \cdots + (k+1)/(k-i+2) + \cdots + (k+1)/2 + ((k+1)/1 - 1) = (k+1)(1/k + \cdots + 1) = (k+1)H_k$. Combining this with above, we get the bound on competitive ratio of any deterministic algorithm: $(k+1)H_k/(k+1) = H_k$.

Let X^n denote the random variable which is the input sequence generated as above of length n. Also note that ALG is a distribution on deterministic algorithms ALG_R. We have proved above that for each deterministic algorithm ALG_r it holds that

$$\mathbb{E}_{X^n}(ALG_r(X^n)) \geq H_k \mathbb{E}_{X^n}(OPT(X^n)) + o(\mathbb{E}_{X^n}(OPT(X^n)).$$

Taking the expectation of the inequality over r, we get

$$\mathbb{E}_R\mathbb{E}_{X^n}(ALG_R(X^n)) \geq H_k \mathbb{E}_{X^n}(OPT(X^n)) + o(\mathbb{E}_{X^n}(OPT(X^n)).$$

Exchanging the order of expectations, it follows that

$$\mathbb{E}_{X^n}\mathbb{E}_R(ALG_R(X^n)) \geq H_k \mathbb{E}_{X^n}(OPT(X^n)) + o(\mathbb{E}_{X^n}(OPT(X^n)).$$

By the definition of expectation, it means that there exists a sequence of inputs x^n such that

$$\mathbb{E}_R(ALG_R(x^n)) \geq H_k OPT(x^n) + o(OPT(x^n)).$$

□

3.7 Yao's Minimax Principle

In the proof of Theorem 3.6.1 we deduced a lower bound on *randomized algorithms against oblivious adversaries* from a lower bound on *deterministic algorithms against a particular distribution of inputs*. This is a general technique that appears in many branches of computer science and is often referred to as Yao's Minimax Principle. In words, it can be stated as follows: the expected cost of a randomized algorithm on a worst-case input is at least as big as the expected cost of the best deterministic algorithm with respect to a random input sampled from a distribution. Let ALG denote an arbitrary randomized algorithm, i.e., a distribution over deterministic algorithms ALG_R. Let μ denote an arbitrary distribution on inputs x. Then we have

$$\max_x \mathbb{E}_R(cost(ALG_R, x)) \geq \min_{\widetilde{ALG}:\text{deterministic}} \mathbb{E}_{X\sim\mu}(cost(\widetilde{ALG}, X)). \tag{3.2}$$

Observe that on the left-hand side ALG is fixed in advance, and x is chosen to result in the largest possible cost of ALG. On the right-hand side the input distribution μ is fixed in advance, and $\widetilde{ALG}$ is chosen as the best deterministic algorithm for μ. Thus, to apply this principle, we fix some distribution μ and show that the expected cost of every deterministic algorithm with respect to μ has to be large, e.g., at least ρ. This immediately implies that any randomized algorithm has to have cost at least ρ. In the above, *cost*() is some measure function. For example, in online algorithms *cost*() is the competitive ratio (strict or asymptotic), in offline algorithms *cost*() is the approximation ratio, in communication complexity *cost*() is the communication cost, and so on. Yao's minimax principle is by far the most popular technique for proving lower bounds on randomized algorithms. The interesting feature of this technique is that it is often *complete*. This means that under mild conditions you are guaranteed that there is a distribution μ that achieves equality in (3.2) for the best randomized algorithm ALG. This way, not only you can establish a lower bound on performance of all randomized algorithms, but you can, in principle, establish the strongest possible such lower bound, i.e., the tight lower bound, provided that you choose the right μ.

In order to state Yao's minimax principle formally, we need to have a formal model of algorithms. This makes it a bit awkward to state for online algorithms, since there is no single model. We would have to state it separately for request-answer games, for search problems, and for any other "online-like" problems that do not fit either of these categories. In addition, for a maximization problem (for example, see the poof of Theorem 7.4.5 in Chapter 7), the statement of the Yao minimax principle is different than for minimization games (i.e., the inequality is reversed) . Moreover, completeness of the principle depends on finiteness of the answer set in the request-answer game formulation. Therefore, we prefer to leave Yao's Minimax Principle in the informal way stated above, especially considering that it's application in each particular case is usually straightforward (as in Theorem 3.6.1) and doesn't warrant a stand-alone black-box statement.

3.8 Upper Bound for Paging

In this section we present an algorithm called Mark that achieves the competitive ratio $\leq 2H_k$ against an oblivious adversary for the Paging problem. In light of Theorem 3.6.1, this algorithm is within a factor of 2 of the best possible *online* algorithm.

The pseudocode of Mark appears in Algorithm 8 and it works as follows. The algorithm keeps track of cache contents, and it associated a Boolean flag with each page in the cache. Initially the cache is empty and all cache positions are unmarked. When a new page arrives, if the page is in the cache, i.e., it's a page hit, then the algorithm marks this page and continues to the next request. If the new page is not in the cache, i.e., it's a page miss, then the algorithm picks an unmarked page uniformly at random, evicts it, brings the new page in its place, and sets the status of the new page to *marked*. If it so happens that there are no unmarked pages at the beginning of this process, then the algorithm *unmarks all pages* in the cache prior to processing the new page.

By tracing the execution of the algorithm on several sample input sequences one may see the intuition behind it: pages that are accessed frequently will be often present in the cache in the marked state, and hence will not be evicted, while other pages are evicted uniformly at random from among all unmarked pages. In the absence of any side information about the future sequence of requested pages, all unmarked pages seem to be equally good candidates. Therefore, picking a page to evict from a uniform distribution is a natural choice.

Theorem 3.8.1.

$$\rho_{OBL}(Mark) \leq 2H_k.$$

Algorithm 8 Randomized algorithm for Paging against oblivious adversaries

```
procedure Mark
    $C[1...k]$ stores cache contents
    $M[1...k]$ stores a Boolean flag for each page in the cache
    Initialize $C[i] \leftarrow -1$ for all $i \in [k]$ to indicate that cache is empty
    Initialize $M[i] \leftarrow False$ for all $i \in [k]$
    $j \leftarrow 1$
    while $j \leq n$ do
        if $x_j$ is in $C$ then                                  ▷ page hit!
            Compute $i$ such that $C[i] = x_j$
            if $M[i] = False$ then
                $M[i] \leftarrow True$
        else                                                     ▷ page miss!
            if $M[i] = True$ for all $i$ then
                $M[i] \leftarrow False$ for all $i$
            $S \leftarrow \{i \mid M[i] = False\}$
            $i \leftarrow$ uniformly random element of $S$
            Evict $C[i]$ from the cache
            $C[i] \leftarrow x_j$
            $M[i] \leftarrow True$
        $j \leftarrow j + 1$
```

Proof. Let $x_1, \ldots, x_n$ be the input sequence chosen by an oblivious adversary. As in Section 1.6, subdivide the entire input sequence into blocks $B_1, \ldots, B_t$, where block B_1 is the maximal prefix of $x_1, \ldots, x_n$ that contains k distinct pages, B_2 is obtained in the same manner after removing B_1 from $x_1, \ldots, x_n$, and so on.

Pages appearing in block B_{i+1} can be split into two groups: (1) new pages that have not appeared in the previous block B_i, and (2) those pages that have appeared in the previous block B_i. Clearly, the case where all pages of type (1) appear *before* all pages of type (2) results in the worst case number of page faults of algorithm Mark. Let m_i be the number of pages of type (1) in block B_i, then block B_i contains $k - m_i$ pages of type (2).

It is easy to see that while processing the first page from each block B_i all existing pages in the cache become unmarked. Every new page of type (1) that is brought in results in a page fault and becomes marked in the cache. A page of type (2) may or may not be present in the cache. If it is not present in the cache, then it is brought in marked; otherwise, it becomes marked. A marked page is never evicted from the cache while processing block B_i.

Consider the first page of type (2) encountered in block B_i. Since all m_i pages of type (1) have already been processed, there are $k - m_i$ unmarked pages of type (2) currently in the cache. Moreover, since the choice of an unmarked page to evict during a page fault is uniform, then the $k - m_i$ unmarked pages of type (2) currently in the cache are equally likely to be any of the original k jobs of type (2) in the cache. Thus, the probability that the first page of type (2) is present (unmarked) in the cache is $\binom{k-1}{m_i}/\binom{k}{m_i} = (k - m_i)/k$. Consider the second page of type (2) encountered in block B_i. We can repeat the above analysis by disregarding the first job of type (2) and pretending that cache size is $k - 1$ (since the first job of type (2) has been marked and for during of block B_i it will never be evicted). Therefore, the probability that the second page of type (2) is present (unmraked) in the cache is $(k - m_i - 1)/(k - 1)$. Proceeding inductively, the probability that the jth job of type (2) in block B_i is present (unmarked) in the cache when it is encountered for the first time is $(k - m_i - j + 1)/(k - j + 1)$. Therefore, the expected number of page faults in block B_i is

$$m_i + \sum_{j=1}^{k-m_i} \left(1 - \frac{k - m_i - j + 1}{k - j + 1}\right) = m_i \sum_{j=1}^{k-m_i} \frac{m_i}{k - j + 1} = m_i + m_i(H_k - H_{m_i}) \le m_i H_k.$$

Note that the number of distinct pages while processing B_{i-1} and B_i is $k + m_i$. Therefore, OPT encounters at least m_i page faults. The number of page faults of OPT in B_1 is m_1. Thus, OPT encounters at least $\left(\sum_i m_i\right)/2$ page faults overall, whereas Mark has expected number of page faults at most $H_k \sum_i m_i$. □

3.9 Exercises

1. Prove Theorem 3.3.1.

2. Prove that $\ln(n) \le H_n \le \ln(n) + 1$.

3. The deterministic Algorithm 6 in Chapter 2 for the One-Way Trading problem was obtained from some randomized algorithm (essentially following the steps of Theorem 3.4.3). Find such randomized algorithm and show that after applying the conversion in the proof of Theorem 3.4.3, you get back Algorithm 6.

4. Prove that $\rho_{\text{OBL}}(Mark) > H_k$.

5. (Harder) Prove that $\rho_{\text{OBL}}(Mark) \ge 2H_k - 1$.

6. Prove that if requested pages are restricted to come from $[k+1]$ then Mark is H_k-competitive.

7. Give an example of an online problem where there is a gap between strict competitive ratio and asymptotic competitive ratio. First, do it for deterministic algorithms, then do it for randomized algorithms against oblivious adversaries. Try to achieve the gap that is as large as possible.

3.10 Historical Notes and References

The different types of adversaries and their relative power is studied in Ben-David et al citeBen-DavidBKTW94.

The modified bit guessing game is an adaption of the Böckenhauer el al [9] string guessing game that was used to establish inapproximations for online algorithms with advice. e

The $2H_k$ *Mark* algorithm for randomized paging and the H_k lower bound for any randomized online algorithm is due to Fiat et al [26]. This was followed by McGeoch and Sleator who otanined the optimal H_k competitive *Partitioning* algorithm. Randomized paging is the most prominent natural example for which randomization leads to a signiifcant improvement in the competitive ratio. Achlioptas, Chrobak and Noga [1] provide a simpler and more efficient H_k competitive algorithm *Equitable* in which the time complexity for each request does not depend on the number of previous requests.

The minimax theorem was originally proved by John von Neumann [48] in the context of zero-sum games, and it was adapted to randomized algorithms by Andrew Yao [50]. It is the central technique used for proving negative results about randomized algorithms. Yao's application was in proving negative results concerning the minimum time needed for a problem in a given computational model. As we have seen in this chapter, it applies equally well to proving negative results concerning competitive ratios. Since paging is a minimization problem, the application of Yao's minimax principle in Section 3.6 follows the statement given in Section 3.7. For a maximization problem, we have the ambiguity as to whether or not competitive ratios are stated to be at most or at least equal to one. In applying the principle when dealing with ratios $\rho < 1$, we need a different statement of the principle so as to prove an "upper bound" (i.e., a negative result) on the competitive ratio.

Chapter 4

Some Classical Problems

As mentioned in the introduction, there were some early seminal papers that began what became known as competitive analysis. In particular, Paging and List Accessing are fundamental online problems that attracted much attention and furthermore motivated the Metrical Task Systems and the k-Server problems. These problems have provided frameworks that have led to innovative algorithmic approaches. As we have also mentioned in the introduction, other classical offline optimization problems, namely the Makespan and Bin Packing problems, were previously studied in the online framework. All of these "classical problems" continue to motivate substantial research activity. In this section, we will mainly review results that have been known since the initial interest in competitive analysis. In Chapter 9, we will present relatively new results providing some indication of the continued interest in these problems.

4.1 Potential Function Method Primer

The potential function method is an important technique in the analysis of algorithms. In this section we provide a high level description of this method, give a rather straightforward application of this method in the area of dynamic data structures, and discuss how this method is applied in the setting of online algorithms. In the next section, we apply this method to the List Accessing Problem.

In the setting of dynamic data structures, the task is to design a data structure that supports different types of operations so as to minimize the total cost (or equivalently, the average/amortized cost) of processing a sequence of operations $op_1, op_2, \ldots, op_m$ chosen by an adversary. There are typically only finitely many different types of operations, e.g., insert, delete, update, etc. It is usually easy to bound the worst-case cost of the i^{th} operation op_i *if we assume that it is of a given type.* For example, we might know that if op_i is of type 1 it costs 1, but if it is of type 2 it costs i. The simplest way to bound the total cost is to prove a *uniform worst-case bound* $cost(op_i) \leq c_i$ that is independent of the type of op_i. Then we could conclude that $\sum_{i=1}^{m} cost(op_i) \leq \sum_{i=1}^{m} c_i$. Proceeding naively, we could obtain the uniform worst-case bound on the cost of op_i by the *maximum* cost over all possible types, since we don't know the type of op_i chosen by the adversary. However, most of the time this leads to an *extremely* pessimistic estimate of the total cost, since not all sequences of operations are possible. It could happen (and often does) that every expensive operation has to be preceded by many inexpensive operations. By taking the maximum possible cost and applying it to every single operation, we are pretending as if every single operation can be an expensive operation.

The idea behind the potential method is to smoothen costs of individual operations: introduce a virtual cost/amortized cost of an operation with the goal of making cheap operations have slightly higher amortized costs and making expensive operations have significantly lower amortized costs. This allows you to get a uniform bound on each operation. If you manage to define amortized costs so that the total amortized cost doesn't differ too much from the total real cost then you get a reasonable bound by the naive approach described above, but applied to the *amortized costs* rather than the real costs. Intuitively, this is similar to financial maintenance funds of buildings – everyone living in the building contributes slightly more than the real on-going month-to-month expenses, so that when the building requires a big repair resulting in a huge and sudden expense, it can be covered out of the accumulated reserve.

Let's look at it more formally in the setting of dynamic data structures. Let S be the set of all possible internal states of the data structure (e.g., cache contents, arrays, pointers, etc.). Let $s_i \in S$ be the state of the data structure at time i. Note that s_0 is the initial state prior to seeing any operations. Define a function, called *the potential function*, $\Phi : S \to \mathbb{R}$, and define the virtual/amortized cost $\widetilde{cost}(op_i) = cost(op_i) + \Phi(s_i) - \Phi(s_{i-1})$. Then we have

$$\sum_{i=1}^{m} \widetilde{cost}(op_i) = \sum_{i=1}^{m} (cost(op_i) + \Phi(s_i) - \Phi(s_{i-1})) = \left(\sum_{i=1}^{m} cost(op_i) \right) + \Phi(s_m) - \Phi(s_0),$$

where the last equality holds since the sum of terms $\Phi(s_i) - \Phi(s_{i-1})$ is telescoping. Thus, if $\Phi(s_m) - \Phi(s_0)$ is $o(OPT)$ then the sum of amortized costs is a good approximation of the sum of real costs. Since you control Φ, this gives you the possibility to define $\widetilde{cost}$s that are more well-behaved than the original *cost*s.

As an example we briefly mention a vector, that is, the data structure for a dynamically re-sizeable array. The goal in this problem is to have quick random access to stored elements, but also allow the storage to grow as much as needed to accommodate new elements that are being inserted. Maintaining a single array is not an option, because arrays are of fixed size. Maintaining a balanced binary search tree is also not an option, because we would like to have $O(1)$ access time to the ith stored element. The idea is to maintain an array A of capacity N that is dynamically resized when the capacity is reached. Thus, when A becomes full, you create a new array of twice the capacity, i.e., $N \leftarrow 2N$, and you copy the contents of the old array over to the new array. The internal state of this data structure is a pair (n, N) where N is the capacity and n is how many elements are actually stored with $n \le N$.

Consider a sequence of operations $op_1, \dots, op_m$ where each operation is either *insert* or *access*. The insert operation inserts a new element at the end of the array, and each access operation returns an element at the specified position. Clearly, each access operation costs 1). However, insert operations are of two types: (1) if we insert an element when $n < N$ then the operation costs 1, and (2) if we insert an element when $n = N$, then the operation costs N, since we need to double the capacity and copy the contents. The worst-case cost of an operation is then $O(N)$ and N can be as large as $\Theta(m)$. If we estimate the cost of each operation by $O(m)$, then we get that the cost of all operations is $O(m^2)$. This is too pessimistic, because we won't be doubling the size of array at each step. Let (n_i, N_i) be the state at time i.

We introduce $\Phi(n_i, N_i) = 2n_i - N_i$. Then we define $\widetilde{cost}(op_i) = cost(op_i) + \Phi(n_i, N_i) - \Phi(n_{i-1}, N_{i-1})$. If op_i is an insertion and $n_{i-1} < N_{i-1}$, then $N_i = N_{i-1}$ and $n_i = n_{i-1} + 1$. So $\Phi(n_i, N_i) - \Phi(n_{i-1}, N_{i-1}) = (2n_i - N_i) - (2n_{i-1} - N_{i-1}) = (2(n_{i-1}+1) - N_{i-1}) - (2n_{i-1} - N_{i-1}) = 2$, and $cost(op_i) = 1$; therefore, $\widetilde{cost}(op_i) = 3$. If op_i is insertion and $n_{i-1} = N_{i-1}$ then $n_i = n_{i-1} + 1$ and $N_i = 2N_{i-1} = 2n_{i-1}$. So $\Phi(n_i, N_i) - \Phi(n_{i-1}, N_{i-1}) = (2n_i - N_i) - (2n_{i-1} - N_{i-1}) = (2(n_{i-1} + 1) - 2N_{i-1}) - (2n_{i-1} - N_{i-1}) = 2 - N_{i-1} = -n_{i-1} + 2$, whereas $cost(op_i) = n_{i-1}$. Therefore we have $\widetilde{cost}(op_i) = n_{i-1} + (-n_{i-1} + 2) = 2$. In all cases, we get that the amortized cost of an operation is $O(1)$. Hence the total amortized cost is $O(m)$. Combining this with the observation that $\Phi(n_m, N_m) \leq N_m \leq m$ and $\Phi(n_0, N_0) = 0$ and the telescoping sum trick, we get that the actual total cost is $O(m)$; that is, linear instead of quadratic in the number of operations!

In the case of online algorithms, the internal state of the online algorithm often does not have enough information to define a useful potential function. Instead, we fix an offline optimal algorithm and pretend that it runs on the same input sequence (with the knowledge of the future) in parallel with our online algorithm. The potential function can then depend on both internal states – that of our online algorithm and that of the offline optimal solution.

4.2 List Accessing Problem

In this section, we consider the static version of the famous List Accessing problem with unit costs. In this problem, you maintain an *ordered* list of m elements. Each online request is to a particular element in the list. To service this request, you have to access the element in the list. If the requested element is at position i in the list, the cost of accessing that element is i. At any point in time, you can swap two adjacent elements at a cost of 1. The goal is to maintain the ordered list in such a way as to minimize the total cost of processing all requests.

Static List Accessing Problem with Unit Costs

Input: $(r_1, \ldots, r_n); m$ — m is the number of elements labelled by $[m]$; $r_i \in [m]$ is the ith request for an element.

Output: $(\sigma_1, \ldots, \sigma_n)$ where $\sigma_i : [m] \to [m]$ is a permutation denoting the ordering of the list immediately prior to processing request r_i: $\sigma_i(j)$ is the position of element j in the list prior to processing item i; σ_1 is the identity permutation.

Objective: To find $\sigma_1, \ldots, \sigma_n$ so as to minimize total cost, which consists of the number of swaps of two adjacent elements required to transform σ_i into σ_{i+1} plus the cost of processing item r_i, given by $\sigma_i(r_i)$, summed over all i.

More general versions of the List Accessing problem have also been considered in the literature. Although we do not cover these more general problems here, we mention a few popular versions. The first generalization is to the dynamic version of the problem, where new elements can be added to the list and old elements can be removed from the list. The new operations incur costs proportional to the position of the element to be removed or the position where the new element is to be inserted. The second generalization is to consider two types of swaps: (a) swaps that involve r_i immediately after processing it, and (b) swaps that involve elements $\neq r_i$ immediately following accessing r_i. The idea is that type (a) swaps should intuitively be cheaper than type (b) swaps — this is because in accessing r_i we would have computed a pointer to r_i within a list, while swapping other elements incurs an additional cost of finding those elements within the list. Thus in this second generalization, one assigns different costs to swaps of type (a) and type (b).

4.2.1 Deterministic Algorithms

The following are three classical algorithms for the static List Accessing problem:

Move-to-Front (MTF): after processing an item, move the item to the front of the list.

Transpose (TRANS): after processing an item, swap it with the immediately preceding item (if there is one).

Frequency Count (FC): maintain an array $F : [m] \to \mathbb{Z}_{\geq 0}$ such that $F[j]$ is the number of times element j has been accessed so far. Maintain σ_i such that the elements are ordered in the order of non-increasing $F[j]$.

Exercises 4-7 asks you to formalize TRANS and FC and analyze their competitive ratios (spoiler: they are not good). In the next two sections we analyze the performance of MTF. The pseudocode for MTF is presented in Algorithm 9. The following easy observation is used a few times throughout the pseudocode: the inverse permutation σ_i^{-1} has the meaning $\sigma_i^{-1}(j)$ = the name of the element (in $[m]$) located at position j in the list immediately preceding the ith request.

Algorithm 9 Move-to-Front algorithm for the List Accessing problem.

```
procedure MTF
    $\sigma_1 \leftarrow$ the identity permutation on $[m]$
    $i \leftarrow 1$
    while $i \leq n$ do
        Process the new request $r_i \in [m]$
        ▷ In the following steps we construct the new permutation $\sigma_{i+1}$
        $\sigma_{i+1}(r_i) \leftarrow 1$                          ▷ move $r_i$ to the front of the list
        for $j = 1$ to $j < \sigma_i(r_i)$ do
            $\ell \leftarrow \sigma_i^{-1}(j)$                        ▷ element at position $j$ in $\sigma_i$
            $\sigma_{i+1}(\ell) \leftarrow j + 1$                ▷ shift this element over by one position
        ▷ Positions of other elements are unchanged
        for $j = \sigma_i(r_i) + 1$ to $j \leq m$ do
            $\ell \leftarrow \sigma_i^{-1}(j)$
            $\sigma_{i+1}(\ell) \leftarrow j$
        $i \leftarrow i + 1$
```

Observe that the cost of moving r_i to the front of the list is $\sigma_i(r_i) - 1$ since the element r_i needs to be swapped with each of the preceding elements. Together with the cost $\sigma_i(r_i)$ of accessing element r_i, we get that the overall cost of processing r_i and preparing the new permutation σ_{i+1} is $2\sigma_i(r_i) - 1$.

The MTF algorithm is very important in theoretical and practical aspects of computer science. In addition to the obvious applications of MTF to memory management, there are much less obvious applications of MTF to efficient encoding of information for communication (almost matching the Shannon's bound), as well as efficient compression that has been shown to empirically outperform *gzip*.

4.2.2 Upper Bound on MTF via the Potential Method

In this subsection we prove that MTF is 4-competitive using the potential function method as outlined in Section 4.1.

Theorem 4.2.1.

$$\rho(MTF) \leq 4.$$

Proof. Let $r_1, \ldots, r_n$ be the input sequence, $(\sigma_1, \ldots, \sigma_n)$ be the list orders of MTF where σ_1 is the identity permutation and σ_i is the order of the list immediately prior to the ith item arrival. Lastly, let $(\sigma'_1, \ldots, \sigma'_n)$ denote the list orders of some optimal algorithm. We say that a pair of elements j and k form an *inversion* with respect to σ_i and σ'_i if either (1) $\sigma_i(j) < \sigma_i(k)$ and $\sigma'_i(j) > \sigma'_i(k)$, or

(2) $\sigma_i(j) > \sigma_i(k)$ and $\sigma'_i(j) < \sigma'_i(k)$. In words, a pair of elements form an inversion w.r.t. σ_i and σ'_i if the two elements appear in different orders in σ_i and σ'_i. The value of the potential function at step i, denoted by Φ_i, is defined as follows:

$$\Phi_i = \text{twice the number of inversions w.r.t } \sigma_{i+1} \text{ and } \sigma'_{i+1}.$$

Let $cost(i) = 2\sigma_i(r_i) - 1$ denote the cost that MTF incurs while processing request r_i. We define the amortized cost $\widetilde{cost}(i)$ as usual:

$$\widetilde{cost}(i) = cost(i) + \Phi_i - \Phi_{i-1}.$$

Observe that $\Phi_0 = 0$ since $\sigma_1 = \sigma'_1 =$ the identity permutation. Also, we have $\Phi_i \geq 0$ for all i. Thus, by the standard application of the potential function method we have $\sum_{i=1}^n \widetilde{cost}(i) = \Phi_n - \Phi_0 + \sum_{i=1}^n cost(i) \geq \sum_{i=1}^n cost(i)$. Hence to prove the statement of the theorem it suffices to show that $\sum_{i=1}^n \widetilde{cost}(i) \leq 4OPT(r_1, \ldots, r_n)$. We establish this claim by showing that for each i, the amortized cost is at most 4 times the cost incurred by OPT in processing r_i. We first analyze how the potential function changes when $\sigma'_{i+1} = \sigma'_i$, i.e., when OPT doesn't change its permutation while processing r_i. Later, we will see that the invariant $\widetilde{cost}_i \leq 4OPT(r_i)$ continues to hold even if $\sigma'_i \neq \sigma'_{i+1}$.

For now, assume that $\sigma'_i = \sigma'_{i+1}$. To see how the value of the potential function changes in processing r_i we need to see how many pairs of elements j, k change their *inversion status*: j, k was an inversion w.r.t. σ_i and σ'_i and j, k becomes not an inversion w.r.t. σ_{i+1} and σ'_{i+1}, and vice versa. First of all, observe that if $j \neq r_i$ and $k \neq r_i$ then their relative orders are exactly the same in σ_i and σ_{i+1}, hence they do not change their inversion status. Thus, we only need to consider pairs that involve r_i. For the following argument it is helpful to consult Figure 4.1. Let k be an element that appears after r_i in the MTF's list, i.e., $\sigma_i(k) > \sigma_i(r_i)$. Moving r_i to the front of the list does not affect the relative order of r_i and k in the MTF's list, therefore r_i and k do not change their inversion status and such pairs do not need to be considered as well. Consider an element j that appears before r_i in σ_i. Moving r_i to the front of the list *flips* the inversion status of j, r_i. In order for j, r_i to flip the status from being an inversion to not being an inversion, j has to appear before r_i in σ'_i, as well. Thus, out of all $\sigma_i(r_i) - 1$ elements preceding r_i in the MTF's list, there can be at most $\sigma'_i(r_i) - 1$ pairs that switch from being non-inversion to inversion. The remaining at least $\sigma_i(r_i) - \sigma'_i(r_i)$ pairs switch from being inversion to non-inversion. Thus, we get that

$$\begin{aligned}
\widetilde{cost}(i) &= cost(i) + \Phi_i - \Phi_{i-1} \\
&\leq 2\sigma_i(r_i) - 1 && \text{(the true cost of the operation)} \\
&\quad + 2(\sigma'_i(r_i) - 1) && \text{(bound on the positive change in the potential function)} \\
&\quad - 2(\sigma_i(r_i) - \sigma'_i(r_i)) && \text{(upper bound on the negative change in the potential function)} \\
&\leq 4\sigma'_i(r_i)
\end{aligned}$$

Observe that in this case the cost of processing r_i by OPT is $OPT(r_i) = \sigma'_i(r_i)$, hence we get the desired statement that $\widetilde{cost}(i) \leq 4OPT(r_i)$.

To handle the general case of $\sigma'_{i+1} \neq \sigma'_i$ we observe that we can treat transformation of σ'_i into σ'_{i+1} one swap of adjacent items at a time. If OPT swaps any two adjacent items j, k then it contributes 1 to its own cost and it can create at most 1 inversion leading to the change in the potential function of at most 2. □

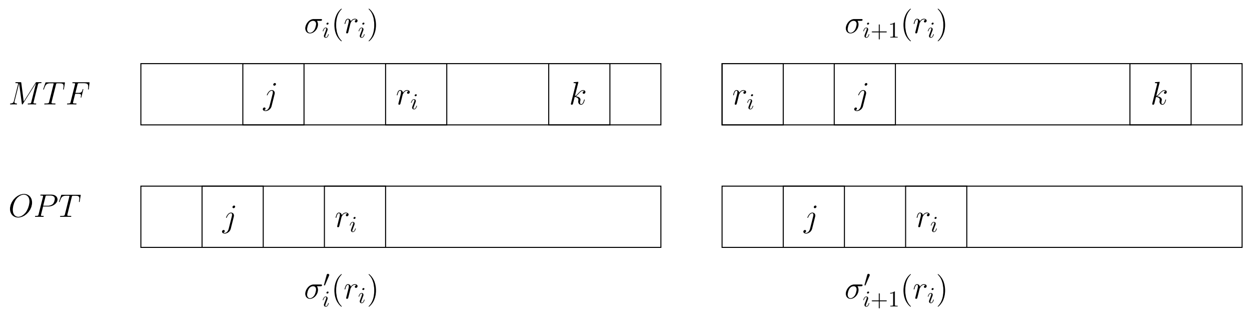

Figure 4.1: States of MTF and OPT at times i and $i+1$ for the case where $\sigma'_{i+1} = \sigma'_i$.

4.2.3 Lower Bound on MTF via the Averaging Technique

In this section we prove a lower bound on the asymptotic competitive ratio of MTF for the List Accessing problem. Asymptotically (in terms of m and n) this lower bound that matches the upper bound of Theorem 4.2.1. We begin by describing the averaging technique that allows us to establish a non-constructive upper bound on the performance of OPT on *any* sequence of input requests.

Lemma 4.2.2. *Let $r_1, \ldots, r_n$ be an arbitrary sequence of requests for the List Accessing problem with m elements. Then we have*

$$OPT(r_1, \ldots, r_n) \le m + \binom{m}{2} + \frac{m+1}{2}(n-1).$$

Proof. We use an averaging technique to prove a good upper bound on OPT.

Let S denote the set of all permutations of $[m]$, hence $|S| = m!$. For each $\sigma \in S$ consider the following simple algorithm ALG_σ: it sets $\sigma_i = \sigma$ for all $i \ge 2$ regardless of the request sequence. Observe that the cost of processing the first request is $\sigma_1(r_1) \le m$ plus the number of swaps of consecutive elements needed to transform the identity permutation into σ, which is upper bounded by $\binom{m}{2}$ – the number of pairs of elements. After that σ_i is never transformed, so the incurred costs are exactly $\sigma(r_i)$ for all $i \ge 2$. The overall cost of ALG_σ is bounded by $\binom{m}{2} + m + \sum_{i=2}^{n} \sigma(r_i)$.

Although for each σ the simple algorithm ALG_σ can perform poorly on the input sequence, the key insight of the averaging technique is to observe that *on average* ALG_σs perform very well, where the average is taken over all values of σ. The crucial observation in the computation of this average is that no matter what r is we have:

$$\sum_{\sigma \in S} \sigma(r) = \sum_{i=1}^{m} i(m-1)! = \frac{m(m+1)}{2}(m-1)!$$

The reason the equality holds is that here $(m-1)!$ permutations that place r in position i for each $i \in [m]$. Now we can compute the average (over σ) cost of ALG_σ

$$\begin{aligned}
\frac{1}{m!}\sum_{\sigma\in S} ALG_\sigma(r_1,\ldots,r_n) &\le \frac{1}{m!}\left(\sum_{\sigma\in S}\binom{m}{2} + m + \sum_{i=2}^{n}\sigma(r_i)\right)\\
&= \binom{m}{2} + m + \frac{1}{m!}\sum_{i=2}^{n}\sum_{\sigma\in S}\sigma(r_i)\\
&= \binom{m}{2} + m + \frac{1}{m!}\sum_{i=2}^{n}\frac{m(m+1)}{2}(m-1)!\\
&= \binom{m}{2} + m + \frac{m+1}{2}(n-1).
\end{aligned}$$

By the definition of the average, the above computation means that there is some σ such that $ALG_\sigma(r_1,\ldots,r_n) \le \binom{m}{2} + m + \frac{m+1}{2}(n-1)$. Since OPT performs at least as well as ALG_σ, the claim follows. □

Now, we are ready to prove the main result of this subsection.

Theorem 4.2.3.

$$\rho(MTF) \xrightarrow{m\to\infty} 4.$$

Recall that m is the size of the static list.

Proof. Consider an adversarial sequence $r_1, r_2, ..., r_n$ that always requests the element at the last position of the current list of MTF. Observe that each request always costs $2m-1$ to process. Therefore, the overall cost of MTF is $(2m-1)n$. The overall cost of OPT is bounded by $m + \binom{m}{2} + \frac{m+1}{2}(n-1)$ by Lemma 4.2.2. Thus, the competitive ratio can be computed as follows:

$$\begin{aligned}
\lim_{n\to\infty}\frac{ALG(r_1,\ldots,r_n)}{OPT(r_1,\ldots,r_n)} &\le \lim_{n\to\infty}\frac{ALG(r_1,\ldots,r_n)}{ALG_\sigma(r_1,\ldots,r_n)}\\
&\le \lim_{n\to\infty}\frac{(2m-1)n}{\binom{m}{2} + m + ((m+1)/2)(n-1)}\\
&= \frac{4m-2}{m+1} = 4 - \frac{6}{m+1}.
\end{aligned}$$

□

Lemma 4.2.2 is quite helpful in the analysis of List Accessing problem. It can be used to show that any deterministic algorithm for the List Accessing problem has to have competitive ratio ≥ 2 as $m \to \infty$ by considering only the cost of looking up an element during the requests (see Exercises at the end of this section).

4.2.4 A Simple Randomized Algorithm BIT

Consider the following randomized algorithm called BIT for the static List Accessing problem with unit costs. The algorithm maintains a Boolean array B of size m, where $B[i] \in \{0,1\}$ for all i. Initially, $B[i]$ is set to a uniformly random bit independently for all $i \in [m]$. When an element r_i is requested, complement its bit ($B[i] \leftarrow 1 - B[i]$), and if $B[i] = 1$ move r_i to the front of the list and otherwise ($B[i] = 0$) leave r_i in place. The pseudocode is given in Algorithm 10. Observe that this

Algorithm 10 Simple randomized algorithm for the List Accessing problem.

```
procedure BIT
    for $i \leftarrow 1$ till $i = m$ do
        $B[i] \leftarrow$ a uniformly random bit from $\{0, 1\}$
    $\sigma_1 \leftarrow$ the identity permutation on $[m]$
    $i \leftarrow 1$
    while $i \leq n$ do
        Process the new request $r_i \in [m]$
        $B[r_i] \leftarrow 1 - B[r_i]$
        if $B[r_i] = 1$ then
            $\sigma_{i+1} \leftarrow$ the permutation obtained from $\sigma_i$ by moving $r_i$ to the front — see Algorithm 9
        else
            $\sigma_{i+1} \leftarrow \sigma_i$
        $i \leftarrow i + 1$
```

algorithm uses m bits of randomness that is independent of the input sequence length n. During the execution of BIT, the values of $B[i]$ are never resampled after the initial phase; instead, the values simply keep flipping on relevant requests.

Next we analyze BIT against an oblivious adversary and show that it improves upon the MTF algorithm: the asymptotic competitive ratio of BIT is bounded by $11/4 = 2.75$ instead of the 4 of MTF.

Theorem 4.2.4.

$$\rho_{OBL}(BIT) \leq \frac{11}{4}.$$

Proof. We prove the statement by adapting the proof of Theorem 4.2.1. Thus, the proof is going to be based on the potential function method. We shall reuse the notation of Theorem 4.2.1 and its proof, so the reader should familiarize themselves with that proof first.

Consider a particular time step i. Suppose that a pair of elements $\{j_1, j_2\}$ form an inversion. Then we define its type as $B[\arg\max(\sigma_i(j_1), \sigma_i(j_2))]$. In words, the type of an inversion is the current state of the bit corresponding to the element of the pair that appears later according to σ_i. Let Φ_b^i denote the number of inversions of type $b \in \{0, 1\}$ with respect to σ_{i+1} and σ'_{i+1}. Define the value of the potential function as follows:

$$\Phi_i = 2\Phi_0^i + 3\Phi_1^i.$$

Observe that $\Phi_0 = 0$ and $\Phi_i \geq 0$ for all i. The amortized cost is defined as usual:

$$\widetilde{cost}_i = cost_i + \Phi_i - \Phi_{i-1}.$$

To establish that $\rho_{\text{OBL}}(BIT) \leq \frac{11}{4}$, it suffices to show that

$$\mathbb{E}(\widetilde{cost}_i) \leq (11/4)\Delta OPT_i,$$

where $\widetilde{cost}_i = cost_i + \Delta\Phi_i$ denotes the change in the amortized cost while processing request i, and ΔOPT_i denotes the change in OPT while processing request i. We write $\Delta\Phi_i = A_i + D_i + C_i$, where A_i is the change in potential due to new inversions being *added* in processing r_i, D_i is the change in potential due to inversions being *deleted*, and C_i is the change in potential due to inversions *changing* type. This accounts for all changes that happen during timestep i. Thus, we have:

$$\mathbb{E}(\widetilde{cost_i}) = \mathbb{E}(cost_i + A_i + D_i + C_i) = \mathbb{E}(cost_i + D_i + C_i) + \mathbb{E}(A_i).$$

The plan for the rest of the proof consists of 2 parts: (1) analyze $\mathbb{E}(cost_i + D_i + C_i)$ by looking the move of BIT alone, i.e., without OPT's move; and (2) analyze $\mathbb{E}(A_i)$ by looking at the move of OPT and BIT. Observe that OPT's move does not affect $cost_i$ and C_i, and can only affect D_i favourably for our goal; therefore, we can ignore its affect in part (1). To simplify the presentation of the two parts, we introduce some notation. Let $k = \sigma_i(r_i)$, $\ell = \sigma'_i(r_i)$, $\ell' = \sigma'_{i+1}(r_i)$. **For now, assume that $\ell' \le \ell$.** Let N_i denote the number of inversions j, r_i such that $\sigma_i(j) < k$. This means that there are $k - N_i - 1$ elements preceding r_i in BIT's list that do not form an inversion. Hence $\ell - 1 \ge k - N_i - 1$, so $k \le \ell + N_i$.

PART (1). Consider first the move of BIT alone. We claim that $cost_i + C_i + D_i \le (B'[r_i] + 1)\ell$, where $B'[r_i]$ is the new value of B after processing r_i. Consider subcase $B[r_i] = 1$ immediately prior to processing r_i: no inversions are created or destroyed, so the change in potential only comes from inversions changing type. Exactly N_i inversions change their type from 1 to 0, resulting in the change in potential $-N_i$. In this subcase we have $cost_i = k \le \ell + N_i$ since BIT only accesses r_i and doesn't perform any swaps. Hence, we have $cost_i + C_i + D_i \le \ell + N_i + 0 - N_i = \ell = (B'[r_i]+1)\ell$. Now, consider subcase $B[r_i] = 0$ immediately prior to processing r_i: N_i inversions of type 0 are destroyed. No new inversions are created. Hence $C_i + D_i \le -2N_i$. We also have $cost_i = 2k - 1 \le 2k \le 2\ell + 2N_i$. Thus, we get $cost_i + C_i + D_i \le 2\ell + 2N_i - 2N_i + 0 \le 2\ell = (B'[r_i] + 1)\ell$. It follows that

$$\mathbb{E}(\widetilde{cost_i}) = \mathbb{E}(cost_i + A_i + C_i + D_i) \le \mathbb{E}((B'[r_i] + 1)\ell) + \mathbb{E}(A_i) = \frac{3}{2}\ell + \mathbb{E}(A_i). \tag{4.1}$$

PART (2). Now, we need to compute $\mathbb{E}(A_i)$. Consider the step of OPT. Let $x_1, \ldots, x_{\ell-1}$ denote the elements preceding r_i in OPT's list immediately before processing r_i. A new inversion is created only when x_j precedes r_i in BIT's list and either BIT or OPT, but not both, moves r_i in front of x_j. Let X_j denote a contribution to A_i of the inversion x_j, r_i if such an inversion is created. OPT moves r_i from ℓ to $\ell' < \ell$. If $B[r_i] = 0$ then r_i is moved to the front of the list of BIT and we have $X_j = 2 + B[X_j]$ for $1 \le j < \ell'$ and $X_j = 0$ for $j \ge \ell'$. If $B[r_i] = 1$ then r_i is not moved and $B[r_i]$ is flipped to zero resulting in $X_j = 0$ for $1 \le j < \ell'$ and $X_j \le 2$ for $j > \ell'$. The value of $B[r_i]$ is a random variable that is distributed uniformly throughout the runtime of the algorithm (why?). Thus, we have

$$\mathbb{E}(A_i) = \sum_{j=1}^{\ell-1} \mathbb{E}(X_j) \le \sum_{j=1}^{\ell'-1} \frac{1}{2}\left(\frac{1}{2}2 + \frac{1}{2}3\right) + \sum_{j=\ell'}^{\ell-1} \frac{1}{2}2 \le \frac{5}{4}\ell. \tag{4.2}$$

Combing equations (4.1) and (4.2), we get that the change in the amortized cost is bounded as follows:

$$\mathbb{E}(\widetilde{cost_i}) \le \frac{3}{2}\ell + \frac{5}{4}\ell = \frac{11}{4}\ell.$$

Since $\Delta OPT_i = \ell + (\ell - \ell') \ge \ell$, the claim follows in this case.

If OPT makes any other move not covered by the above (e.g., $\ell' > \ell$), the move costs OPT exactly 1, and it creates at most one transposition which results in the expected change in the potential of $\frac{1}{2}2 + \frac{1}{2}3 = \frac{5}{2} \le \frac{11}{4}$. □

4.3 k-Server Problem

The k-Server problem is among the most famous and influential problems in the area of online algorithms. Actually, it is not even a single problem, but rather a whole family of problems. Each particular problem in the family is characterized by the underlying metric space. To state the k-Server problem formally takes some preparation, which we do in Section 4.3.1. Readers familiar with metric spaces, configurations, and min-cost matchings can skip ahead to Section 4.3.2. We introduce the notion of a "work function" and discuss its properties in Section 4.3.5. The work function can be used to design an optimal offline algorithm, but more importantly it is used in the best known online algorithm for the k-Server problem on general metrics — the Work Function algorithm. We present the algorithm in Section 4.3.6. We defer the discussion of randomized algorithms for the k-Server problem till a later chapter on recent progress.

4.3.1 Preliminaries

We begin with the definition of a metric space, followed by some example.

Definition 4.3.1. A *metric space* $\mathcal{M}$ is a pair (X, d) where X is a set of points and $d : X \times X \to \mathbb{R}_{\geq 0}$ is a *metric/distance* function that satisfies the following *metric space axioms*:

Positivity. $d(x, y) > 0$ for all $x, y \in X$ such that $x \neq y$.

Reflexivity. $d(x, x) = 0$ for all $x \in X$.

Symmetry. $d(x, y) = d(y, x)$ for all $x, y \in X$.

Triangle Inequality. $d(x, y) \leq d(x, z) + d(z, y)$ for all $x, y, z \in X$.

The above definition is the mathematical abstraction of a space where you can measure the distance. If X is finite then we call $\mathcal{M}$ a *finite* metric space.

Example 4.3.2. The standard 1-dimensional real line is a metric space $\mathcal{M} = (\mathbb{R}, |\cdot|)$, where the distance between two real numbers is simply the absolute value of their difference.

Example 4.3.3. Generalizing the previous example, the n-dimensional real space is a metric space $\mathcal{M} = (\mathbb{R}^n, ||\cdot||_2)$, where the distance between two vectors x and y is the standard Euclidean distance $||x - y||_2 = \sqrt{(x_1 - y_1)^2 + \cdots + (x_n - y_n)^2}$. Such $\mathcal{M}$ is called the standard Euclidean space. Observe that $\mathbb{R}^n$ can be paired with other notions of distance to create metric spaces on $\mathbb{R}^n$ different from the standard Euclidean space.

Example 4.3.4. Previous examples gave infinite metric spaces. For an example of a finite metric space, consider a weighted undirected graph $G = (V, E, w)$, where $w : E \to \mathbb{R}_{\geq 0}$ is a non-negative weight function. Define $d(x, y) =$ the weight of shortest path between x and y. Then one can check that $\mathcal{M} = (V, d)$ is a finite metric space.

Example 4.3.5. The following is an example of a simple distance function d that can be defined on *any* set of points X (finite or infinite):

$$d(x, y) = \begin{cases} 1 & \text{if } x \neq y \\ 0 & \text{if } x = y \end{cases}$$

The above is known as the *discrete or uniform* metric on X.

For the following definitions, fix some metric space $\mathcal{M}$ and a natural number $k \in \mathbb{N}$.

Definition 4.3.6. A multiset C of k points from $\mathcal{M}$ is called a *configuration*.

Definition 4.3.7. Consider two configurations C_1 and C_2. A *perfect matching* between C_1 and C_2 is a bijection $\sigma : C_1 \to C_2$. Thus a matching between C_1 and C_2 associates exactly one unique point of C_2 to each point of C_1.

Definition 4.3.8. Given a perfect matching σ between configurations C_1 and C_2, *the cost* of σ, is defined as $\sum_{x \in C_1} d(x, \sigma(x))$.

Definition 4.3.9. The *distance* between configurations C_1 and C_2, denoted by $d(C_1, C_2)$ is defined as the minimum cost of a perfect matching between C_1 and C_2. Intuitively, $d(C_1, C_2)$ is the least distance that needs to be travelled to transform C_1 into C_2 (or vice versa, which is the same thing by symmetry).

Definition 4.3.10. Let $x \in C$ and $y \in X$. We write $C - x + y$ to denote the new configuration that is obtained from C by replacing x with y.

4.3.2 Formulation of the Problem

For concreteness, suppose that you run a consulting company. You have k consultants on staff. When a new request comes in, you have to dispatch a consultant to the location of the customer. Once the consultant finishes the job, they can be assigned to travel to the location of a new request. Consultants travel from a location of one customer immediately to a location of a new customer, because there is no "home base" of operations of your company. The consultants are completely interchangeable, and the questions is how to assign consultants to customers so that to minimize the total travelled distance.

In the more formal statement of the problem, you fix a metric space $\mathcal{M} = (X, d)$ and you control k servers (correspond to consultants), which are initially located at a multiset of points from X. We use C_0 to denote this initial configuration. A sequence of requests is simply a sequence of points $x_i \in X$ (correspond to locations of customers). To process each x_i you can reposition the k servers into a new configuration C_i such that at least one of the servers is located at x_i, i.e., $x_i \in C_i$. The goal is to minimize the total travel distance, as measured by $\sum_{i=1}^{n} d(C_i, C_{i-1})$ (see Definition 4.3.9).

k-Server Problem
Input: $(x_1, \ldots, x_n); \mathcal{M} = (X, d); k; C_0$ — $\mathcal{M}$ is a metric space; k is the number of servers; C_0 is the initial configuration of the servers; $x_i \in X$ is the ith request.
Output: $(C_1, \ldots, C_n)$ where C_i is a multiset of k points from X denoting the ith configuration of the servers.
Objective: To find $C_1, \ldots, C_n$ so as to minimize the total distance $\sum_{i=1}^{n} d(C_i, C_{i-1})$ subject to servicing all requests, i.e., $x_i \in C_i$ for all $i \in [n]$.

This problem generalizes the Paging problem (see Exercises 10 and 11). One algorithm immediately comes to mind:

Greedy: to process x_i relocate the closest server to x_i breaking ties arbitrarily. If we let y^i_j denote the location of jth server in C_i, then to process x_i we pick the server $\arg\min_j \{d(x_i, y^i_j)\}$.

It is easy to see that the competitive ratio of the greedy algorithm is unbounded (see Exercise 9). In general, the best achievable competitive ratio could depend on $\mathcal{M}$. However, the following conjecture that is quite widely believed to be true states that the deterministic competitive ratio is k regardless of the metric space (as long as $\mathcal{M}$ has at least $k + 1$ points).

Conjecture 4.3.1 ((Deterministic) k-Server Conjecture). Consider the k-Server problem with metric space $\mathcal{M}$. Let ALG be a best deterministic online algorithm for this problem. If $\mathcal{M}$ has at least $k + 1$ points, then

$$\rho(ALG) = k.$$

There has been a lot of progress on the above conjecture, but it is still open. The best bounds for general metric spaces are $\rho(ALG) \geq k$, which we show in Section 4.3.3, and $\rho(ALG) \leq 2k-1$, which we outline in Section 4.3.6.

Conjecture 4.3.2 (Randomized k-Server Conjecture). For every metric space there is a randomized online algorithm for the k-Server problem that achieves competitive ratio $O(\log k)$ against oblivious adversaries.

There is also a randomized version of the k-Server conjecture that is also wide open, but there were important recent developments, which we will cover in Chapter 9.

The formulation of the k-Server problem allows an online algorithm to relocate several servers in between requests. A natural class of algorithms, called "lazy algorithms", consists of those algorithms that never move servers that they do not have to move to serve a request. This means that if the new request is already covered by the current configuration then *no servers* are moved, and if the new request is not covered by the current configuration then *exactly 1 server* is moved. Formally, we have

Definition 4.3.11. An online algorithm for the k-Server problem is called *lazy* if its configurations $C_1, \ldots, C_n$ satisfy the property:

- if $x_i \in C_{i-1}$ then $C_i = C_{i-1}$;
- if $x_i \notin C_{i-1}$ then there exists one $y \in C_{i-1}$ such that $C_i = C_{i-1} - y + x_i$, i.e., we reposition exactly one server.

By Exercise 12 it suffices to consider only lazy algorithms for the k-Server problem.

4.3.3 Deterministic Lower Bound

In this section we show that no algorithm can be better than k-competitive for the k-Server problem with respect to *any* metric space $\mathcal{M}$, as long as $\mathcal{M}$ has at least $k+1$ distinct points.

Theorem 4.3.1. *Let $\mathcal{M}$ be an arbitrary metric space with at least $k+1$ distinct points. Let ALG be a deterministic online algorithm for the k-Server problem with respect to $\mathcal{M}$. Then, we have*

$$\rho(ALG) \geq k.$$

Proof. Fix a set $\widetilde{X} = \{p_1, \ldots, p_{k+1}\}$ of $k+1$ points from $\mathcal{M}$. Set $C_0 = \{p_1, \ldots, p_k\}$. The adversary will construct a sequence of requests using only points from $\widetilde{X}$. Since ALG has k servers and $\widetilde{X}$ has $k+1$ points, for each configuration C_{i-1} there is a point x_i that is not covered by C_{i-1}, e.g., $x_1 = p_{k+1}$. The adversary presents the sequence $x_1, \ldots, x_n$, i.e., the adversary keeps requesting an uncovered point from among $\widetilde{X}$. We assuming that ALG is lazy, without loss of generality. Let y_i denote the original position of the server that got moved to process x_i. Clearly, we have $ALG(x_1, \ldots, x_n) = \sum_{i=1}^{n} d(y_i, x_i)$. Observe that $x_{i+1} = y_i$ — if a server is moved, the adversary requests its previous position in the very next step. Thus, we can rewrite $ALG(x_1, \ldots, x_n) = \sum_{i=1}^{n-1} d(x_{i+1}, x_i) + d(y_n, x_n) \geq \sum_{i=1}^{n-1} d(x_{i+1}, x_i)$.

To bound OPT we use the averaging technique. Consider k algorithms ALG_i for $i \in [k]$ defined as follows. Initially, ALG_i starts at configuration C_0. To service $x_1 = p_{k+1}$ algorithm ALG_i uses the server at p_i. Observe that there exists exactly one location that is covered by all ALG_i, namely, p_{k+1}. Algorithms ALG_i are lazy and they behave so as to maintain this invariant. We illustrate it with an example. Suppose that $x_2 = p_5$ arrives. Since x_2 is covered by all ALG_i with $i \neq 5$, none of these algorithms move a server. The only algorithm that has to move a server is ALG_5.

Since we want to maintain that at all times there is *exactly one point* from $\mathcal{X}$ that is covered by *all* algorithms ALG_i, ALG_5 has to move the server that is presently at location p_{k+1}. Next, suppose that $x_3 = p_{10}$. Arguing as before, only ALG_{10} has to move a server, and it has to move the server that is presently at location p_5. And so on.

Let $T(x_1, \ldots, x_n) = \sum_{i=1}^k ALG_i(x_1, \ldots, x_n)$ denote the sum of costs of all algorithms ALG_i. Following the example in the previous paragraph, it is easy to see by induction that at each time step j only one algorithm has to move a server and moreover the algorithm moves a server that is at location x_{j-1}. Thus, we have $T(x_1, \ldots, x_n) = \sum_{j=2}^n d(x_j, x_{j-1}) + \sum_{i=1}^k d(p_i, p_{k+1})$, where the second term is from the initialization procedure during processing x_1. Therefore, the average cost over the k algorithms is

$$\frac{1}{k}T(x_1, \ldots, x_n) = \frac{1}{k}\sum_{j=1}^{n-1} d(x_{j+1}, x_j) + \frac{1}{k}\sum_{i=1}^{k} d(p_i, p_{k+1}).$$

In particular, one of the algorithms achieves this cost, hence $OPT(x_1, \ldots, x_n) \le \frac{1}{k}\sum_{j=1}^{n-1} d(x_{j+1}, x_j) + \frac{1}{k}\sum_{i=1}^k d(p_i, p_{k+1})$. Note that $\frac{1}{k}\sum_{i=1}^k d(p_i, p_{k+1}) = o(OPT)$. Thus, we have

$$ALG(x_1, \ldots, x_n) \ge \sum_{i=1}^{n-1} d(x_{i+1}, x_i) \ge kOPT(x_1, \ldots, x_n) + o(OPT).$$

□

4.3.4 k-Server on a Line

In this section we present an elegant algorithm for the special case of the k-Server problem, where we take $\mathcal{M}$ to be the standard Euclidean 1-dimensional space, a.k.a., a real line (see Example 4.3.2). The algorithm is called DoubleCoverage or DC, for short. The pseudocode is presented in Algorithm 11. We use $p_1 \le \ldots \le p_k$ to denote positions of the k servers of the algorithm during its runtime. The algorithm works as follows. Consider a new request x_j. If x_j is to the right of the right-most server then the algorithm uses the right-most server to serve x_j. If x_j is to the left of the left-most server then the algorithm uses the left-most server to process x_j. The remaining case is when x_j is in between two servers i^* and i_*, i.e., $p_{i_*} \le x_j \le p_{i^*}$ and there are no other servers between p_{i_*} and x_j, and x_j and p_{i^*}. Then DC starts moving *both* servers i^* and i_* towards x_j at the same speed until one of the servers reaches x_j.

Notice that DC, as stated, is not a lazy algorithm. Although by Exercise 12 we could modify DC to be lazy, it is conceptually easier to analyze the non-lazy version of the algorithm. Also observe that the relative order of servers $p_1 \le p_2 \le \cdots \le p_k$ can be easily preserved during the execution of the algorithm.

Next, we show that DC is k-competitive. In light of Theorem 4.3.1, this ratio is tight and it is the best possible deterministic competitive ratio for the real line.

Theorem 4.3.2.

$$\rho(DC) \le k.$$

Proof. Consider an input instance $x_1, \ldots, x_n$. As discussed before, we use $p_1 \le p_2 \le \cdots \le p_k$ to denote the sorted positions of the servers during the execution of the DC algorithm. Similarly, let $q_1 \le q_2 \le \cdots \le q_k$ denote the sorted positions of the servers during the execution of OPT. We use

Algorithm 11 DoubleCoverage algorithm for the k-Server problem on a line.

```
procedure DC
    ▷ $C_0$ is the initial pre-specified configuration.
    Initialize $p_i \in \mathbb{R}$ to the initial coordinate of server $i$ according to $C_0$
    ▷ we have $p_1 \leq p_2 \leq \cdots \leq p_k$, which is maintained during execution
    $j \leftarrow 1$
    while $j \leq n$ do
        The new request $x_j \in \mathbb{R}$ arrives
        if $x_j > p_k$ then                    ▷ request is to the right of the right-most server
            $p_k \leftarrow x_j$                      ▷ use the right-most server to process $x_j$
        else if $x_j < p_1$ then                  ▷ request is to the left of the left-most server
            $p_1 \leftarrow x_j$                        ▷ use the left-most server to process $x_j$
        else            ▷ request is in between the right-most and left-most servers
            $i^* \leftarrow \arg\min_i\{p_i \mid p_i \geq x_j\}$      ▷ find a server that is immediately to the right of $x_j$
            $i_* \leftarrow \arg\max_i\{p_i \mid p_i \leq x_j\}$      ▷ find a server that is immediately to the left of $x_j$
            $\delta \leftarrow \min(|p_{i^*} - x_j|, |p_{i_*} - x_j|)$    ▷ distance until one of the servers reaches $x_j$
            ▷ Move the two servers
            $p_{i^*} \leftarrow p_{i^*} - \delta$
            $p_{i_*} \leftarrow p_{i_*} + \delta$
        $C_j \leftarrow$ the multiset formed by $p_1, \ldots, p_k$
        $j \leftarrow j + 1$
```

the potential function method to establish the result. We use the following potential function:

$$\Phi = k \sum_{i=1}^{k} |p_i - q_i| + \sum_{i<j} |p_i - p_j|.$$

Thus, we can write $\Phi = \Phi_1 + \Phi_2$, where $\Phi_1 = k \sum_{i=1}^{k} |p_i - q_i|$ and $\Phi_2 = \sum_{i<j} |p_i - p_j|$. Since we shall analyze how the potential function changes in each step j, we assume that Φ refers to the current step under consideration and so do the p_i and the q_i. Thus, we drop index j from all the variables to reduce clutter in our notation. Let $\Delta\Phi$ denote the change in potential, ΔDC denote the true cost incurred by DC, and ΔOPT denote the cost incurred by OPT. In processing x_j both OPT and DC have to move. By the potential function method, our goal is to show that after both moves we have

$$\Delta DC + \Delta\Phi \leq k\Delta OPT. \tag{4.3}$$

We can prove this by showing that the inequality holds for the move of OPT in isolation followed by the move of DC, i.e., we can analyze the inequality one move at a time. We provide the summary of how different moves can affect DC, OPT, and $\Delta\Phi$ in Table 4.1. It is relatively straightforward to verify that the table is correct and that equation (4.3) follows from the summary in the table. □

4.3.5 Work Function

Let $x = (x_1, \ldots, x_n)$ be the request sequence. We write $x_{\leq t}$ to denote the subsequence consisting of the first t elements, i.e., $x_{\leq t} = (x_1, \ldots, x_t)$. Thus, $x_{\leq 0}$ is the empty sequence and $x_{\leq n} = x$. Recall that C_0 denotes the initial pre-specified configuration.

Move type	ΔDC	$\Delta\Phi_1$	$\Delta\Phi_2$	$\Delta DC + \Delta\Phi$	ΔOPT
OPT moves	0	$\leq k\Delta OPT$	0	$\leq k\Delta OPT$	ΔOPT
DC moves right-most or left-most server	ΔDC	$-k\Delta DC$	$(k-1)\Delta DC$	0	0
DC moves two servers for an "in-between" request	ΔDC	≤ 0	$-\Delta DC$	0	0

Table 4.1: Summary of changes to DC, OPT, and potential function during one step.

Definition 4.3.12. *The work function* $w_x : X^k \to \mathcal{R}_{\geq 0}$ is defined with respect to the request sequence $x = (x_1, \ldots, x_n)$ and initial configuration C_0. The work function maps configurations in the metric space $\mathcal{M}$ to real numbers with the following meaning: $w_x(C)$ is the minimum total distance that is needed to process all requests x (in order) **and** end up in configuration C. Note that C can be an arbitrary configuration and is not required to contain x_n or, in fact, any x_i. Formally, $w_x(C)$ is defined as follows

$$w_x(C) = \min_{C_1,\ldots,C_n} \left\{ \sum_{i=0}^{n} d(C_i, C_{i+1}) : \forall i \in [n] \;\; r_i \in C_i \text{ and } C_{n+1} = C \right\}.$$

We will consider work functions with respect to prefixes $x_{\leq t}$ of x, i.e., $w_{x_{\leq t}}$. To simplify notation we will denote these work functions by $w_{\leq t}$. Note $w_x = w_{\leq n}$.

The following lemma collects a number of observations regarding the work function.

Lemma 4.3.3. *1. $w_{\leq 0}(C) = d(C_0, C)$.*

2. $OPT(x) = \min_C\{w_{\leq n}(C)\}$.

3. $w_{\leq t}(C) \geq w_{\leq t-1}(C)$.

4. If $x_t \in C$ then $w_{\leq t}(C) = w_{\leq t-1}(C)$.

5. If $x_t \notin C$ then $w_{\leq t}(C) = \min_{x \in C}\{w_{\leq t-1}(C - x + x_t) + d(x_t, x)\}$.

6. $w_{\leq t}(C) = \min_{x \in C}\{w_{\leq t-1}(C - x + x_t) + d(x_t, x)\}$.

Proof. 1. The meaning of $w_{\leq 0}(C)$ is the minimum distance to end up in C without processing any requests. Since the initial configuration is C_0 we have to move from C_0 to C, the answer is given precisely by $d(C_0, C)$.

2. OPT is the minimum total distance required to process the entire sequence of requests starting from C_0 and ending in some configuration C.

3. Clear, since in $w_{\leq t}$ you need to process more requests than in $w_{\leq t-1}$.

4. There are two possible strategies to process $x_1, \ldots, x_t$ and end up in C. The first strategy is to process $x_1, \ldots, x_{t-1}$ and end up in C and stay in C, since $x_t \in C$. The total distance of the first strategy is $w_{\leq t-1}(C)$. Another strategy is to process $x_1, \ldots, x_{t-1}$ and end up in $C' \neq C$ and then move from C' to C. The total distance of the second strategy is $w_{\leq t-1}(C') + d(C, C')$. By the definition of w, we have $w_{\leq t-1}(C) \leq w_{\leq t-2}(C') + d(C', C) \leq w_{\leq t-1}(C') + d(C', C)$, where the last step is by the previous item.

5. In this case x_t is not in C but we still have to process it before moving to configuration C. Consider the time immediately after some server processed x_t. The order in which we reposition servers into configuration C does not matter. Thus, we can assume that the server that processed x_t is repositioned *last*. The other servers occupy all but one of the positions in C, say, x. The optimal distance of achieving this step is $w_{\le t}(C-x+x_t) = w_{\le t-1}(C-x+x_t)$ (by the previous item). Then it is left to move the server from x_t to x, which adds distance $d(x_t, x)$ to the total cost. Overall, processing x_t and moving to C costs $w_{\le t-1}(C-x+x_t) + d(x_t, x)$. Since we don't know which server is the best to process x_t, we have to minimize over all choices of x.

6. This is a straightforward consequence of the previous two items.

□

The observations in the lemma can be used to come up with an optimal offline algorithm for the k-server problem based on dynamic programming: item (1) gives the base case and item (6) shows how to fill out the table. Exercise 15 asks you to fill in the details.

4.3.6 Work Function Algorithm

In this section we describe the online algorithm that achieves the best known upper bound for the k-Server problem — the Work Function algorithm, or WFA for short. The algorithm is easy to describe: it is a lazy algorithm that processes request x_t by moving the server $x \in C_{t-1}$ that minimizes $w_{\le t-1}(C_{t-1} - x + x_t) + d(x_t, x)$. Algorithm 12 provides the pseudocode.

Algorithm 12 The Work Function algorithm for the k-Server problem on general metrics.

```
procedure WFA
    ▷ $C_0$ is the initial pre-specified configuration.
    $j \leftarrow 1$
    while $j \le n$ do
        The new request $x_j$ arrives
        $x \leftarrow \arg\min_{x \in C_{j-1}} \{w_{\le j-1}(C_{j-1} - x + x_j) + d(x_j, x)\}$
        $C_j \leftarrow C_{j-1} - x + x_j$
        $j \leftarrow j + 1$
```

We state the best known upper bound on the performance of WFA in the following theorem. We omit the proof of this result, as it is quite long and involved and there are already several excellent expositions listed in the historical notes at the end of this chapter.

Theorem 4.3.4.

$$\rho(WFA) \le 2k - 1.$$

It is possible that $\rho(WFA) = k$, but no one has been able to prove it yet. The claim that $\rho(WFA) = k$ has only been established for certain special cases of metrics, e.g., $\mathcal{M} = (X, d)$ such that $|X| = k + 1$.

4.4 Exercises

1. Consider the following randomized version of MTF: after processing an item request, move the item to the front of the list with probability 1/2 and leave the item in its original place

with probability 1/2. What is the asymptotic competitive ratio of this randomized version of MTF?

2. Fix $c \geq 1$. Analyze the competitive ratio of MTF (in terms of c) for the generalized version of the List Accessing problem, where swaps of type (a) cost 1 and swaps of type (b) cost c. See Section 4.2 for the relevant definitions.

3. In the case of unit costs of *all types of swaps* in the static List Accessing problem, is it necessary for OPT to use type (b) swaps? If it is, then give an instance where OPT cannot be achieved without using type (b) swaps and prove it. If it is not necessary, give a general conversion procedure (and prove that it works) that transforms any optimal algorithm that uses type (b) swaps into an algorithm that uses only type (a) swaps.

4. Write down pseudocode for the algorithm TRANS for the List Accessing problem.

5. Prove that the algorithm TRANS for the List Accessing problem does not achieve any constant competitive ratio.

6. Write down pseudocode for the algorithm FC for the List Accessing problem.

7. Does the algorithm FC for the List Accessing problem achieve a constant competitive ratio?

8. Prove that the metric spaces in Examples 4.3.2-4.3.5 satisfy the metric space axioms.

9. Write down pseudocode for the greedy algorithm for the k-Server problem. Give an adversarial input sequence that shows that the the competitive ratio of the greedy algorithm is unbounded.

10. Prove that Paging is a special case of the k-Server problem. Specifically, define $\mathcal{M}$ such that the corresponding k-Server problem is *exactly* the Paging problem.

11. Is List Accessing a special case of the k-Server problem? Specifically, can you define $\mathcal{M}$ such that the corresponding k-Server problem is *exactly* the List Accessing problem? If not, what's the main difficulty?

12. Prove that any online algorithm for the k-Server problem can be converted into a lazy algorithm without hurting its competitive ratio.

13. Show that the result of Exercise 9 can be achieved for the 2-Server problem with respect to the Euclidean 1-dimensional space and by considering only requests coming from 3 possible locations. Trace how the DC algorithm works on such an adversarial instance and see why it is able to outperform greedy.

14. Verify Table 4.1.

15. Design an offline optimal algorithm for the k-Server problem based on Lemma 4.3.3. Describe it in pseudocode, prove its correctness, and analyze its running time.

16. Design a more efficient offline optimal algorithm for the k-Server problem than the one suggested in Exercise 15.

17. Use Lemma 4.2.2 to prove that any deterministic algorithm for the List Accessing problem has to have competitive ratio ≥ 2 as $m \to \infty$. Can you prove a stronger lower bound?

4.5 Historical Notes and References

NOTE: The history of the these more classical online problems is extensive. Like the history of the bin packing problem, we will only give a partial list of references. The interested reader can find reasonably comprehensive history in the papers we cite. And very recent results have made very significant advances for the MTS and k-server problems.

Chapter 5

Game Theory Analysis for Online Algorithms

5.1 Two-Person Zero-Sum Games

5.2 Application to Competitive Analysis

5.3 Paging

Is Paging
ple of the
ciple so n
is the bes
this. Mat
a good ex
analysis.

5.4 Online Algorithms and Pricing

5.5 Exercises

5.6 Historical Notes and References

5.7 Additional Related Results

Chapter 6

Primal-Dual Method for Online Problems

6.1 Linear Programming and Duality

6.2 Primal-Dual Approach in the Online Setting-Pricing Revisited

6.3 Online Set Cover

Not sure
here so I
it out an
introdyce
4.

6.4 Exercises

6.5 Historical Notes and References

6.6 Additional Related Results

Chapter 7

Graph Problems

In this chapter we study graph problems in an online setting. More specifically, we consider problems where the input contains a *graph that is not known in its entirety in advance* and is revealed in an online fashion. These types of problems are not to be confused with online problems that are defined in terms of graphs that are known in advance (e.g., the k-Server problem with respect to graph metrics). From complexity theory, we know that many graph problems are not only NP-hard but, moreover, hard (under various complexity assumptions) to approximate much beyond what are trivial approximations. This is the case, for example, for computng a maxmum independent set, and for computing a minimum graph coloring. However, these complexity based hardness results do not preclude the possibility that there can be an online algorithm with a good approximation since we do not impose complexity assumptions in competitive analysis. We shall see, however, that many graph programs cannot be well approximated using only the myopic information theoretic constraints.

There are several issues involved in the study of online graph problems. The first issue is in defining a representation of an online input item. Turns out that there are at least four different natural input models. The input model that should be used for a particular graph problem depends on the application domain. In addition to the four input models for general graphs, there is also another natural input model that is specific to bipartite graphs. The second issue arises from the fact that many online graph problems do not admit any non-trivial deterministic or randomized algorithms under any of the input models. In other words, many graph problems are completely hopeless for online algorithms under worst-case input models. Thus, the literature on worst-case analysis of online graph problems is rather scarce in comparison with scheduling, routing, and packing problem domains. In order to get meaningful results, one often has to restrict inputs to come from a special class of graphs, e.g., bipartite, small degree, or k-colorable for small values of k. There are some success stories with regards to studying more specialized graph problems. In this chapter, we will study two such success stories: Bipartite Maximum Matching and Bipartite Graph Coloring.

We begin this chapter by introducing the different input models and discussing relationships between them. Then, we show that several natural graph problems are completely hopeless for online algorithms on general graphs. We then move on to studying Bipartite Maximum Matching, and we finish the chapter by studying Graph Coloring.

7.1 Input Models for General Graphs

In the online world, we are always faced with the question as to what is an input item. For some problem areas (e.g. Paging, Makespan, Bin Packing), there is a seemingly most natural choice for an input item. For graph problems, there are a number of reasonable choices which we discuss in this section, but first we introduce some notation.

We shall consider an undirected and unweighted graph $G = (V, E)$, where $E \subseteq \binom{V}{2}$. We typically use n to denote $|V|$ and m to denote $|E|$. Most definitions in this section generalize to directed and weighted graphs in a rather straightforward way. We use $\sim$ to denote adjacency, i.e., $u \sim v$ is the same thing as $\{u, v\} \in E$. We define the neighborhood of a set of vertices $S \subseteq V$, denoted by $N(S)$, as those nodes outside S that have a neighbor in S, i.e.:

$$N(S) := \{v \in V : \exists u \in S \text{ such that } u \sim v\} \setminus S.$$

For a single vertex $v \in V$ we denote its neighborhood $N(\{v\})$ by $N(v)$ for short.

A graph is composed of two kinds of objects — vertices and edges. An adversary will present corresponding input items in a certain order. Thus, we will often assume that the sets V and E are totally ordered. We will use the symbol $\prec$ to indicate the order. For example, if $\prec$ is a total order defined on V, then for two vertices $u, v \in V$ we write $u \prec v$ to indicate that the input item associated with u is revealed before the input item associated with v.

Now, we are ready to describe the basic online input models for graphs.

The Edge Model (EM). Each input item is an edge given by its two endpoints. Thus, the input graph G is presented as a sequence of edges $\{u_1, v_1\}, \{u_2, v_2\}, \dots, \{u_n, v_n\}$. The algorithm does not know V a priori and it is understood that $V = \bigcup_{i=1}^{n} \{u_i, v_i\}$.

The Vertex Adjacency Model, Past History (VAM-PH). Each input item consists of a vertex together with a set of neighbors among the vertices that have appeared before. Suppose that input items are revealed in the order given by $\prec$ on V, then each new input item can be described as follows:

$$(v; N(v) \cap \{u : u \prec v\}).$$

The Edge Adjacency Model (EAM) In this model, we associate with each edge $e = \{u, v\}$ a label ℓ_e. Note that the label does not carry the information about the endpoints of an edge e. An input item in this model consists of a vertex together with a set of labels of edges that are incident on that vertex. Formally, each input item is of the form:

$$(v; \{\ell_e : v \in e\}).$$

Thus, if a neighbor $u \sim v$ has appeared before v, i.e., $u \prec v$, then an online algorithm can recover the information that $u \sim v$, since both data items corresponding to u and v will contain the same label $\ell_{\{u,v\}}$. If a neighbor $u \sim v$ appears after v, i.e., $v \prec u$, then an online algorithm only knows that some neighbor is going to appear later in the input sequence, but it does not know the identity of u at the time of processing v.

The Vertex Adjacency Model, Full History (VAM-FH). Each input item consists of a vertex together with the set of all its neighbors (even the neighbors that have not appeared before):

$$(v; N(v)).$$

For two models M1 and M2 we use the notation M1≤M2 to indicate that any algorithm that works in model M1 can be converted into an algorithm that works in model M2 without any deterioration in the performance, as measured by the worst-case competitive ratio. Intuitively, M1≤M2 means that M1 is a harder model for online algorithms, while M2 is harder for adversaries. In light of this definition, we have the following lemma.

Lemma 7.1.1. *We have the following relationships*

$$EM \leq VAM\text{-}PH \leq EAM \leq VAM\text{-}FH$$

Proof. We will show the first relation. The rest are delegated to exercises.

Let ALG be an algorithm for the EM model. We show how to convert it into an algorithm ALG' that operates in the VAM-PH model. Suppose that an input item in VAM-PH model arrives: $(v; \{u_1, \dots, u_k\})$, where $\{u_1, \dots, u_k\} = N(v) \cap \{u : u \prec v\}$. Then ALG' splits this input item into a sequence of edges $\{v, u_i\}$ and feeds these edges into ALG. It uses responses of ALG to create its own response for $(v; \{u_1, \dots, u_k\})$ in a consistent manner. Thus, ALG' achieves the same value of the objective function as ALG, which is at least the value of the objective function achieved by ALG on the worst-case ordering of edges (v, u_i). This finishes the proof.

Note that this proof essentially says that VAM-PH can be thought of as presenting edges of graph G in groups, where edges are grouped together by a vertex on which they are incident. Thus we can think of running ALG in VAM-PH as running ALG in EM, but on restricted kinds of input sequences. □

The relationship ≤ used in Lemma 7.1.1 is transitive: M1≤M2 and M2≤M3 implies that M1≤M3. Thus, EM is the hardest model for the algorithms and VAM-FH is the easiest model for online algorithms, as expected.

We note that the model that makes the most sense for online applications is often (but not always) the VAM-PH model, especially when a graph problem requires decisions about nodes. Suppose we want to apply the theory of online algorithms to graphs that grow dynamically, for example, graphs arising out of social networks. In those real-life scenarios, consider the graph growing because of new users joining the network (as opposed to new friendships being established between existing users). When a new user joins the network, they establish connections with friends that are already using the social network. Let $\prec$ denote the order in which the users join the network. Then when user v arrives we learn about v together with its friends $u_1, \dots, u_k$ such that $u_i \prec v$. Therefore, VAM-PH is the most suitable model for such applications. In contrast, consider what it means to analyze social networks in VAM-FH, for example. When a user v joins the network, not only do we learn about its friends who are already in the network, but we also learn about all its friends that are not yet in the network, but are *guaranteed* to join the network later. This is a less realistic scenario for these kinds of applications.

In some graph problems, it might be more appropriate to make decisions about edges rather than about nodes. For instance, suppose that a solution to a problem is a matching or a path or a tree (as in the MST problem). Then such a solution can be represented as a sequence of 0/1 decisions about edges – whether or not to include an edge in a matching or path or tree. In such scenarios, the EM input model might be more natural than the VAM-PH input model. Often, there are graph problems where either model can be considered natural. For example, consider a maximum matching problem. A matching can either be thought of as a collection of vertex-disjoint edges, which leads to 0/1 decisions in the EM model, or it can be viewed as a map going from a vertex to its (potentially matched) neighbor, which leads to decisions labelled by names of neighbors in the VAM-PH model.

The EAM and VAM-FH models provide partial information about the future, so these models are not so easily justified. However, in some applications you do have side-information. For instance, you sometimes either know the true degree of each arriving node or you have an upper bound on the degree of each arriving node. The EAM and VAM-FH models can be viewed as stronger versions of this degree side-knowledge.

7.2 Special Input Model for Bipartite Graphs

Let $G = (U, V, E)$ be an unweighted bipartite graph, where U and V are the sets of vertices in the two parts and $E \subseteq U \times V$ is the set of edges going between the two parts. We can always "forget" the bipartite structure of the graph and consider G as a general graph. Thus, we can always consider G given in one of the input models of Section 7.1. However, in many applications a special kind of input model can be assumed. This model is specific to bipartite graphs only and it is defined as follows:

The Bipartite Vertex Arrival Model (BVAM). In this model, one part of the graph is revealed to an algorithm in advance, and vertices from another part arrive in the vertex adjacency format. We assume that vertices in V are revealed in advance. Thus, V is also called the set of "known" or "offline" vertices. Vertices from U arrive one at a time in a certain order chosen by an adversary. Thus, U is also called the set of "online" vertices. When an online node $u \in U$ arrives, all of its incident offline vertices, $N(u)$, are revealed as well.

Lemma 7.2.1. *For bipartite graphs we have*

$$\textit{VAM-PH} \le \textit{BVAM}$$

Proof. Let ALG be an algorithm that works on bipartite graphs presented in VAM-PH model. We shall design ALG' with the same performance guarantees, but it will work in the BVAM model. Before arrival of any of the vertices $u \in U$, ALG' feeds vertices V into ALG without any edges. Observe that vertices in V do not have edges between each other, thus this step simulates graph G in VAM-PH correctly so far. Now, when ALG' receives an item $(u, N(u))$, it can feed this item into ALG, because $N(u) \subseteq V$ and V have already appeared in the simulated input to ALG. Lastly, ALG' uses the decisions of ALG verbatim. $\square$

The relationships between BVAM and EAM, as well as between BVAM and VAM-FH are not so straightfoward. For example, we almost have BVAM≤VAM-FH, because when VAM-FH receives an input item corresponding to the "offline" side V, it can ignore the input item; and when VAM-FH receives an input item corresponding to the "online" side U, it can use an algorithm for BVAM to make a decision about the input item. The reason that this reduction does not quite work is that in VAM-FH nothing is known about the set of nodes in advance, while in BVAM the entire offline side V is known in advance. Suppose that an online algorithm in BVAM model makes online decisions that depend not only on data items corresponding to $u \in U$ but also on V. Then such decisions cannot be simulated in VAM-FH without additional a priori knowledge.

7.3 Hard Online Graph Problems

In this section we present several online graph problems under various input models from Section 7.1. The results are pessimistic in the sense that all the problems that we consider do not admit non-trivial online algorithms. Unfortunately, this is a standard state of affairs when it comes to online graph problems. The main take-away point of this section is that to study online graph problems, we need to either restrict input graphs to special classes, or change the input model from worst-case online to say average-case, or change the algorithmic model to some other online-like models, e.g., streaming, online with advice, priority, dynamic data structures, and so on.

7.3.1 Maximum Independent Set

A subset of vertices $S \subseteq V$ is called an independent set if there are no edges between any pair of vertices in S. In the Maximum Independent Set problem, we wish to find an independent set of largest possible cardinality. In the online version of the problem, we need to decide whether to include a newly arriving vertex into an independent set or not. We consider this problem in the VAM-PH input model. More formally, the problem is stated as follows:

Maximum Independent Set

Input: $G = (V, E, \prec)$; G is an unweighted and undirected graph; $\prec$ is the total order on V; $(v_1, N_1), \ldots, (v_n, N_n)$ is the sequence of input items, where $v_i \prec v_{i+1}$ and $N_i = N(v_i) \cap \{v_j : j < i\}$.
Output: $d_1, \ldots, d_n$ — such that $d_i \in \{0, 1\}$ indicates whether to include v_i into an independent set or not.
Objective: To find $d_1, \ldots, d_n$ so as to maximize the size of the constructed set $S = \{v_i : d_i = 1\}$ subject to S being an independent set: $v_i \neq v_j \in S$ implies $v_i \not\sim v_j$.

We begin by showing that every deterministic online algorithm ALG can be fooled by an adversary in the following strong sense: the algorithm can be forced to find an independent set of size 1 in a graph with an independent set $n-1$, where $n = |V|$. In the standard notation, we have $\rho(ALG) \geq n - 1$.

Theorem 7.3.1. *Let ALG be a deterministic online algorithm for the Maximum Independent Set problem under the VAM-PH input model. Then we have*

$$\rho(ALG) \geq n - 1.$$

Proof. As long as decisions of the algorithm ALG are 0, an adversary keeps presenting isolated vertices. That is, the first few items are of the form (v_i, N_i) such that $N_i = \emptyset$ and $d_i = 0$. This happens for $i = 1$ to j. Then for some j an adversary presents (v_j, N_j) with $N_j = \emptyset$ and ALG responds with $d_j = 1$ for the first time. First, we claim that $j < \infty$, since otherwise ALG finds an independent set of size 0, whereas OPT includes all the isolated vertices into the solution, leading to an infinite competitive ratio. Thus, an adversary has to encounter $d_j = 1$ eventually. After that point onward the adversary presents items that have v_j as their sole neighbor. That is, items (v_i, N_i) for $i > j$ have $N_i = \{v_j\}$. This continues until the nth item is presented. Note that ALG cannot answer $d_i = 1$ on any of the items with $i > j$, since then the solution constructed by ALG is not an independent set. Thus, $ALG = 1$. Also, observe that the maximum independent set includes all the vertices, except for v_j. Thus, $OPT = n - 1$. The result follows. □

Observe that any greedy algorithm would find an independent set of size at least 1 in non-empty graphs. The above theorem says that this is the best one can hope for! The following theorem shows that randomized algorithms cannot do much better.

Theorem 7.3.2. *Let ALG be a randomized online algorithm for the Maximum Independent Set problem under the VAM-PH input model. Then we have*

$$\rho_{OBL}(ALG) = \Omega(n).$$

Proof. We will use Yao's minimax principle (see Section 3.7) to prove the claim. Thus, we shall present a distribution on inputs and show that every *deterministic* algorithm achieves competitive ratio $\Omega(n)$ on average with respect to that distribution.

The distribution is as follows. The input is generated two vertices at a time. The first two vertices v_0 and v_1 are isolated. Thus, the first two items are $(v_0, \emptyset)$ and $(v_1, \emptyset)$. To generate the next two vertices, first sample a random bit $b_1 \in \{0,1\}$ uniformly at random. Then the next two vertices $v_{b_1 0}$ and $v_{b_1 1}$ are presented as having v_{b_1} as their neighbor. Thus, the two items corresponding to these vertices are $(v_{b_1 0}, \{v_{b_1}\})$ and $(v_{b_1 1}, \{v_{b_1}\})$. To generate the next two vertices, again sample a random bit $b_2 \in \{0,1\}$ uniformly at random. The next two vertices $v_{b_1 b_2 0}$ and $v_{b_1 b_2 1}$ are presented as having v_{b_1} and $v_{b_1 b_2}$ as their neighbors. Thus, the two items corresponding to these vertices are $(v_{b_1 b_2 0}, \{v_{b_1}, v_{b_1 b_2}\})$ and $(v_{b_1 b_2 1}, \{v_{b_1}, v_{b_1 b_2}\})$. This process continues for $n/2$ rounds. During the ith round we generate two vertices $v_{b_1 \ldots b_{i-1} 0}$ and $v_{b_1 \ldots b_{i-1} 1}$ with neighbors $v_{b_1}, v_{b_2}, \ldots, v_{b_{i-1}}$. At the end of the process (after $n/2$ rounds) there are n vertices in total. See Figure 7.1 for an example.

First, observe that the vertices $v_{(\neg b_1)}, v_{b_1(\neg b_2)}, v_{b_1 b_2(\neg b_3)}, \ldots, v_{b_1 b_2 \ldots (\neg b_{n/2-1})}$ form an independent set. Thus, $OPT \geq n/2 - 1$. Second, consider an arbitrary *deterministic* algorithm running on such a random instance. Observe that in each round the algorithm is given two vertices: $v_{b_1 \ldots b_{i-1} 0}$ and $v_{b_1 \ldots b_{i-1} 1}$. We say that the algorithm participates in a round if it chooses at least one of the vertices. Note that choosing one of these vertices, namely the vertex $v_{b_1 \ldots b_i}$, prevents the algorithm from choosing any future vertices since all the future vertices contain $v_{b_1 \ldots b_i}$ in the neighborhood. We will call this vertex as the incorrect vertex and the other vertex in that round is called the correct vertex. Thus, if the algorithm participates in a round by choosing both vertices, the algorithm cannot participate in any future rounds because it is guaranteed to choose the incorrect vertex. If the algorithm participates in a round by choosing a single vertex, it has a 50-50 chance of choosing the correct vertex $v_{b_1 \ldots b_{i-1}(\neg b_i)}$ that will not prevent the algorithm from participating in future rounds. The reason it is 50-50 is because b_i is not yet decided at the time the algorithm has to make a choice. Thus, on average the algorithm participates in 2 rounds until it picks one of the "incorrect" vertices. Therefore the independent set constructed by this deterministic algorithm is of size at most 2 on average. Together with the first observation about OPT this results in the competitive ratio of $\Omega(n)$. $\square$

7.3.2 Maximum Clique

A subset of vertices S is called a clique if every two distinct vertices from S are adjacent. In the Maximum Clique problem the goal is to find a clique of maximum cardinality. We consider this problem in the VAM-PH input model. Formally this problem is stated as follows (it looks very similar to Maximum Independent Set):

Maximum Clique Problem

Input: $G = (V, E, \prec)$ — G is an unweighted and undirected graph; $\prec$ is the total order on V; $(v_1, N_1), \ldots, (v_n, N_n)$ is the sequence of input items, where $v_i \prec v_{i+1}$ and $N_i = N(v_i) \cap \{v_j : j < i\}$.
Output: $d_1, \ldots, d_n$ — such that $d_i \in \{0,1\}$ indicates whether to include v_i into a clique or not.
Objective: To find $d_1, \ldots, d_n$ so as to maximize the size of the constructed set $S = \{v_i : d_i = 1\}$ subject to S being a clique: $v_i \neq v_j \in S$ implies $v_i \sim v_j$.

In this section we establish results analogous to Theorems 7.3.1 and 7.3.2. Rather than proving the analogous statements from scratch, we will use the *reduction* technique. We say that an online problem P1 *reduces* to another online problem P2 if any online algorithm that achieves competitive ratio ρ for P2 can be transformed into an online algorithm that achieves competitive ratio ρ for P1. This implies that lower bounds proved on competitive ratios of online algorithms for P1 immediately

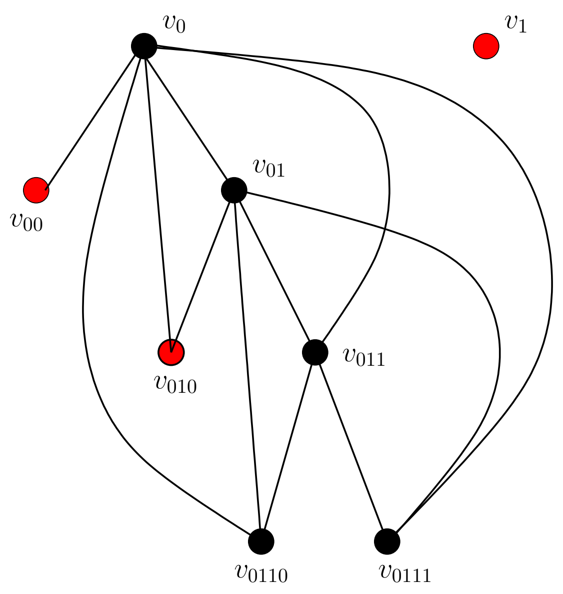

Figure 7.1: Example of an input instance used in Theorem 7.3.2. In this example, we have $b_1 = 0$, $b_2 = 1$, $b_3 = 1$. The vertices $v_1 = v_{(\neg b_1)}$, $v_{00} = v_{b_1(\neg b_2)}$ and $v_{010} = v_{b_1 b_2(\neg b_3)}$ form an independent set (shown in red).

carry over to the same lower bounds for P2. Thus, we need to prove the following:

Theorem 7.3.3. *Maximum Independent Set reduces to Maximum Clique under the VAM-PH input model.*

Proof. We start with a simple observation: S is a clique in $G = (V, E)$ if and only if S is an independent set in the complement graph of G, i.e., $G^c = (V, \binom{V}{2} \setminus E)$.

Let ALG be an online algorithm for Maximum Clique. We need to construct an algorithm ALG' for Maximum Independent Set that has the same competitive ratio as ALG. The idea behind the reduction is that while ALG' is receiving G in the VAM-PH model it can generate the VAM-PH representation of the complement graph G^c. Thus, it can feed G^c into ALG and use ALG's decisions d_i as its own. While ALG finds a clique in G^c, ALG' finds an independent set of exactly same size in G. In addition OPT is exactly the same whether we look for a maximum clique in G^c or for a maximum independent set in G. Thus, the competitive ratio is preserved.

It is left to see how ALG' can generate G^c online. ALG' receives an input item $(v; \{u : u \prec v \text{ and } u \sim_G v\})$, where $u \sim_G v$ denotes the adjacency of u and v in G. Observe u and v are adjacent in G^c if and only if they are not adjacent in G. Therefore, to generate an input item corresponding to v in G^c, the algorithm ALG' can simply declare that the neighborhood of v consists of those vertices that have already appeared and are not adjacent to v in G. That is ALG' creates an input item $(v; \{u : u \prec v \text{ and } u \not\sim_G v\})$. □

Observe that the above reduction works for both deterministic and randomized algorithms. Thus, we have an immediate corollary:

Corollary 7.3.4.

1. *Let ALG be a deterministic online algorithm for the Maximum Clique problem under the VAM-PH input model. Then we have*

$$\rho(ALG) \geq n - 1.$$

2. *Let ALG be a randomized online algorithm for the Maximum Clique problem under the VAM-PH input model. Then we have*
$$\rho_{OBL}(ALG) = \Omega(n).$$

7.3.3 Longest Path

A path in G is called simple if it does not contain any repeated vertices. In the Longest Path problem the goal is to find a simple path that is as long as possible. We will consider this problem in the EM input model. Formally, it is defined as follows:

Longest Path

Input: $G = (V, E), \prec)$ — G is an unweighted and undirected graph; $\prec$ is the total order on E; $e_1 \prec \cdots \prec e_m$ is the sequence of input items, where $e_i = \{v_i, u_i\} \in E$ is an edge, and $\bigcup_i \{e_i\} = E$.
Output: $d_1, \ldots, d_m$ — such that $d_i \in \{0, 1\}$ indicates whether to include e_i into a path or not.
Objective: To find $d_1, \ldots, d_m$ so as to maximize the length of the constructed path $P = \{e_i : d_i = 1\}$ subject to P forming a simple path.

We begin by proving a strong lower bound on the competitive ratio of deterministic algorithms.

Theorem 7.3.5. *Let ALG be a deterministic algorithm for the Longest Path problem in the EM model. Then*
$$\rho(ALG) \geq n - 3.$$

Proof. We give an adversarial argument. The adversary fixes $V = \{v_1, \ldots, v_n\}$ and presents the first edge $e_1 = \{v_1, v_2\}$. If ALG does not take this edge, i.e., $d_1 = 0$, then the adversary presents $e_2 = \{v_2, v_3\}$. If ALG does not take this edge, i.e., $d_2 = 0$, then the adversary presents $e_3 = \{v_3, v_4\}$. And so on. If ALG does not accept any edges in this process, then the input terminates after e_{n-1}. In this case, we have $ALG = 0$ and $OPT = n - 1$, since the entire graph is just a simple path of length $n - 1$. This leads to an infinite competitive ratio.

Suppose that ALG accepts the first edge $e_1 = \{v_1, v_2\}$ in the above process, i.e., $d_1 = 1$. Then the adversary is going to switch its strategy from the above one in the next step. The adversary will present $e_2 = \{v_3, v_4\}$, $e_3 = \{v_4, v_5\}$ and so on until $e_{n-2} = \{v_{n-1} v_n\}$. Observe that ALG cannot accept any e_i with $i > 1$ since this would lead to a disconnected solution. Also observe that the entire graph in this case consists of two disjoint simple paths: one of length 1 and another of length $n - 3$. Hence, $OPT \geq n - 3$ and $ALG = 1$. This leads to a competitive ratio $n - 3$.

Lastly, suppose that ALG accepts an edge $e_i = \{v_i, v_{i+1}\}$ for some $i > 1$ in the process described in the first paragraph. Then the adversary is going to switch its strategy: instead of $\{v_{i+1}, v_{i+2}\}$ it will present $e_{i+1} = \{v_{i-1}, v_{i+2}\}$, $e_{i+2} = \{v_{i+2}, v_{i+3}\}$, $e_{i+3} = \{v_{i+3}, v_{i+4}\}$, and so on until e_{n-1}. Observe that ALG cannot accept any of the edges e_j with $j > i$, because it would lead to a disconnected solution. Thus, $ALG = 1$. Meanwhile OPT can take the entire graph without edges e_i, e_{i-1} as a solution, since it forms a path of length $n - 3$. This leads to a competitive ratio $n - 3$.

See Figure 7.2 for an example of each of the above scenarios. □

Note that the above lower bound is very strong, it says that the simplest algorithm that just accepts the first edge it is presented is essentially optimal when it comes to worst-case analysis. The Longest Path problem is also difficult for the randomized algorithms. Exercise 3 asks you to prove the following theorem.

Theorem 7.3.6. *Let ALG be a randomized algorithm for the Longest Path problem in the EM model. Then*
$$\rho_{OBL}(ALG) = \Omega(n).$$

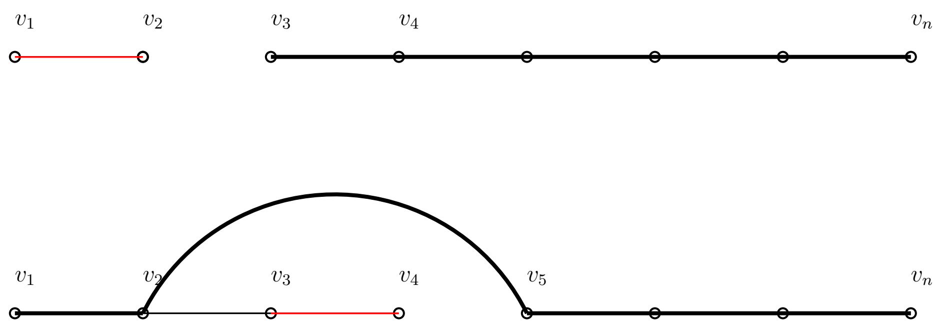

Figure 7.2: Example of adversarial instances used in Theorem 7.3.5. At the top we have the input graph constructed by the adversary when ALG accepts the first edge. At the bottom we have the input graph constructed by the adversary when ALG accepts the third edge. Red edges correspond to ALG solution and bold edges correspond to OPT.

7.3.4 Minimum Spanning Tree

Given a graph $G = (V, E)$, a subgraph $T = (V, E' \subseteq E)$ is called a spanning tree if T is connected and acyclic. If the graph G is edge-weighted, i.e., there is a weight function $w : E \to \mathbb{R}$, then we can also define the weight of the tree T as $w(T) := \sum_{e \in E'} w(e)$. In the Minimum Spanning Tree (MST, for short) problem the goal is to find a spanning tree of minimum weight. We consider this problem in the EM input model. In the weighted version of the EM model, when an edge e is revealed its weight $w(e)$ is revealed as well. Formally, the problem is defined as follows:

Minimum Spanning Tree

Input: $G = (V, E, w, \prec)$; G is an edge-weighted undirected graph; $w : E \to \mathbb{R}$ is the weight function; $\prec$ is the total order on E; $(e_1, w(e_1)) \prec \cdots \prec (e_m, w(e_m))$ is the sequence of input items, where $e_i = \{v_i, u_i\} \in E$ is an edge and $w(e_i)$ is its weight; we have $\bigcup_i \{e_i\} = E$.

Output: $d_1, \ldots, d_m$ — such that $d_i \in \{0, 1\}$ indicates whether to include e_i into a tree or not.

Objective: To find $d_1, \ldots, d_m$ so as to minimize the total weight $w(T)$ of the constructed tree $T = (V, E')$, where $E' = \{e_i : d_i = 1\}$, subject to T forming a spanning tree.

In this section, we show that even a randomized online algorithm cannot perform well for this problem. More specifically, we prove:

Theorem 7.3.7. *Let ALG be a randomized algorithm for the MST problem in the EM model. Then*

$$\rho_{OBL}(ALG) = \Omega(n).$$

Proof. We present a distribution on inputs and show that any *deterministic* algorithm has competitive ratio $\Omega(n)$ on average with respect to this distribution. The claim then follows by Yao's minimax principle.

Let n be even and let $V = \{v_0, v_1, \ldots, v_{n-1}\}$. We place all vertices with an even index in a row, and all vertices with an odd index in another row immediately above. Vertices in the bottom row form a path where each edge is of weight ϵ. Each vertex in the bottom row is connected to a vertex directly above it in the top row via an edge of weight 1. Lastly, we select an index $i \in \{0, 1, \ldots, n/2 - 1\}$ uniformly at random. We connect vertices in the top row from v_1 to v_{2i-1} via a path where each edge is of weight ϵ, and we connect vertices in the top row from v_{2i+3} to v_{n-1} via another path where each edge is of weight ϵ. This construction is depicted in Figure 7.3. Formally, it can be described as follows: set $E_1 := \{\{v_{2j}, v_{2j+2}\} : j \in \{0, \ldots, n/2 - 2\}\}$, $E_2 = \{\{v_{2j}, v_{2j+1}\} : j \in \{0, \ldots, n/2 - 1\}\}$, $E_3 = \{\{v_{2j+1}, v_{2j+3}\} : j \neq i, i - 1\}$, and let $E = E_1 \cup E_2 \cup E_3$. All edges in E_1 and E_3 have weight ϵ, while all edges in E_2 have weight 1. We call the edge $\{v_{2i}, v_{2i+1}\}$ an "isolated column edge," since its top vertex is isolated from all other top vertices.

Let ALG be a deterministic algorithm working on this distribution. The order of arrival of edges is as follows: edges from E_1 and E_2 are presented first in an interleaved fashion, followed by all edges from E_3. Thus, the first few terms of the input sequence look like this: $\{v_0, v_2\}, \{v_0, v_1\}, \{v_2, v_4\}, \{v_2, v_3\}, \{v_4, v_6\}, \{v_4, v_5\}, \ldots$ Observe that the random choice i can be deferred until all of E_1 and E_2 edges have been presented. We claim that in order for the algorithm to guarantee a feasible solution that it has to accept all E_2 edges. Suppose that an algorithm decides not to pick edge $\{v_{2j}, v_{2j+1}\}$, then in the case that $i = j$ the algorithm cannot cover vertex v_{2j+1} in the tree. This leads to an infeasible solution and an infinitely bad payoff. This happens with a positive probability, i.e., $2/n$, hence this decision would contribute infinity to the expected cost of a solution. Thus, if ALG hopes to achieve any bounded competitive ratio, it has to accept all of E_2. This means that $ALG \geq n/2$. Meanwhile, as can be seen from Figure 7.3, OPT can achieve a solution of cost $3 + \epsilon(n/2 - 2)$. By taking $\epsilon > 0$ sufficiently small, we get that OPT's cost is at most 4. This leads to the competitive ratio $\Omega(n)$. □

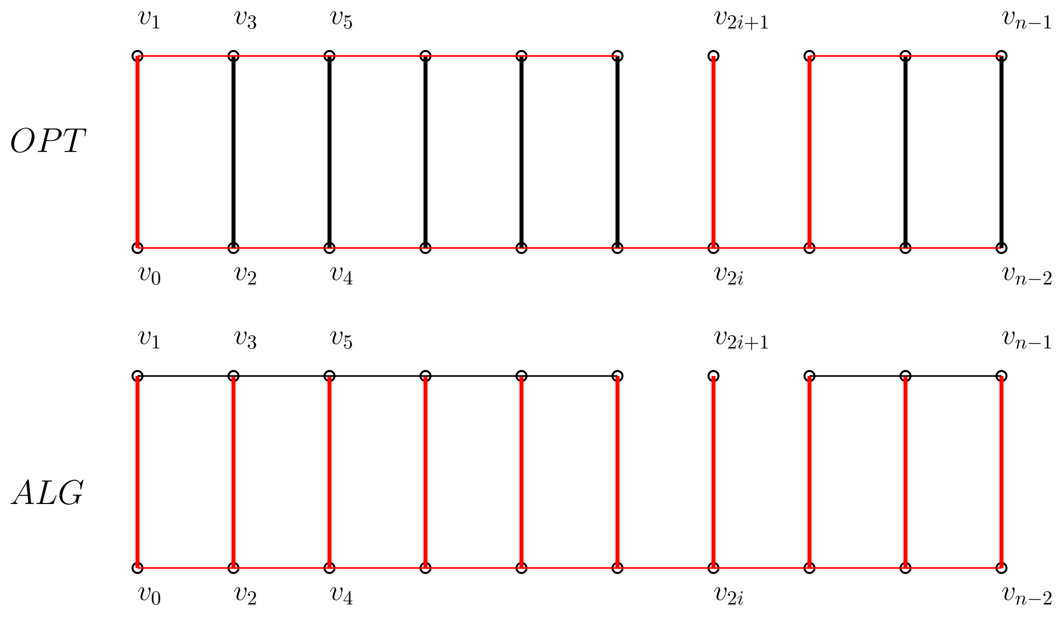

Figure 7.3: Example of an adversarial instance used in Theorem 7.3.7. In the instance bold (vertical) edges have weight 1 and thin (horizontal) edges have weight ϵ. The position of the "isolated column edge" $\{v_{2i}, v_{2i+1}\}$ is random. The solution of OPT is shown at the top in red, while the solution of ALG is shown at the bottom in red. The blue edges are edges that are not in the matching.

7.3.5 Travelling Salesperson Problem

In the Travelling Salesperson Problem (TSP, for short), you are given an edge-weighted complete graph $G = (V, E = \binom{V}{2}); w : E \to \mathbb{R}_{\geq 0} \cup \{\infty\}$, and the goal is to find a minimum-weight cycle that visits every node exactly once. We consider this problem in the EM input model. Formally, this problem is defined as follows.

Travelling Salesperson Problem

Input: $G = (V, E, w, \prec)$; G is an edge-weighted complete graph; $E = \binom{V}{2})$; $w : E \to \mathbb{R}_{\geq 0} \cup \{\infty\}$

is the weight function; $\prec$ is the total order on E; $(e_1, w(e_1)) \prec \cdots \prec (e_m, w(e_m))$ is the sequence of input items, where $e_i = \{v_i, u_i\} \in E$ is an edge and $w(e_i)$ is its weight.
Output: $d_1, \ldots, d_m$ — such that $d_i \in \{0, 1\}$ indicates whether to include e_i into a cycle.
Objective: To find $d_1, \ldots, d_m$ so as to minimize the total weight $w(C)$ of the constructed cycle $C = \{e_i : d_i = 1\}$, subject to C forming a well-defined cycle.

Like some other problems in this section, TSP is a very difficult problem for online (and offline) algorithms. We start with deterministic algorithms.

Theorem 7.3.8. *Let ALG be a deterministic algorithm for the TSP problem in the EM model. Then*

$$\rho(ALG) = \infty.$$

Proof. Let $V = \{v_1, \ldots, v_n\}$. The adversary presents edges incident on vertex v_1 first: $\{v_1, v_2\}, \{v_1, v_3\}, \ldots, \{v_1,$
The adversary declares the weight of each edge to be 1 until one of two things happen: (1) either ALG accepts such an edge, or (2) there is only one edge left to present, namely $\{v_1, v_n\}$.

In case (1), the adversary declares all other edges (even those not incident on v_1) to have weight 0. In this case, $ALG = 1$ and $OPT = 0$ leading to competitive ratio ∞.

In case (2), ALG is forced to take edge $\{v_1, v_n\}$ to maintain feasibility, so the adversary declares the weight of this edge to be ∞. All other edges are revealed in an arbitrary order and with weight 0. Thus, we have that $OPT = 1$ and $ALG = \infty$ leading to competitive ratio ∞, as well. □

In Exercise 5 you are asked to establish a similar result for randomized algorithms.

7.4 Bipartite Maximum Matching

Online Bipartite Maximum Matching (BMM for short) provides one of the most interesting online graph problems both because of its relevance to the applications in Auctions and Online Advertising (see Chapters 19 and 20), and because the bipartite matching problem has led to an number of new algorithmic developments. Indeed, we shall be considering BMM in a number of other online and “online-like” models in Chapters 10, 11, 12, 14, 16 and 17.

In this chapter we only consider the basic adversarial online framework for the BMM problem in the BVAM input model while deferring alternative computational models to later chapters. Given a graph $G = (U, V, E)$ a subset of edges $M \subseteq E$ is called a matching if the edges in M do not share any vertices, i.e., for all $e, e' \in M$ we have $e \cap e' = \emptyset$. In the unweighted matching problem, the goal is to find a matching that is as large as possible. In Chapter 20, we will consider various weighted versions of bipartite matching which more realistically model applications in online advertising. The size of a maximum matching in G is denoted by $\mu(G)$. We now restrict attention to the BMM problem, that is, the “one sided” online bipartite maximum matching problem where one-sided refers to the setting where only one side of the bipartite graph is considered as the online nodes. More formally it is defined as follows:

Bipartite Maximum Matching
Input: $G = (U, V, E, n \prec)$; G is an unweighted bipartite graph; $n = |U|$ is the number of online vertices; $\prec$ is the total order on online vertices U; V consists of offline vertices that are known to an algorithm in advance; $(u_1, N(u_1)) \prec \cdots \prec (u_n, N(u_n))$ is the sequence of input items, where $u_i \in U$ is an online vertex and $N(u_i) \subseteq V$ is its neighborhood.
Output: $d_1, \ldots, d_n$ — such that $d_i \in N(u_i) \cup \{\perp\}$ indicates how to match vertex u_i to its neighbor ($\perp$ indicates u_i remains unmatched).
Objective: To find $d_1, \ldots, d_n$ so as to minimize the size of the constructed matching $M = \{(u_i, d_i) : d_i \neq \perp\}$, subject to M forming a well-defined matching.

Before discussing algorithms for BMM, we introduce some notation. We often consider neighborhoods of vertices when a partial matching has been constructed. Then we write $N_c(u)$ to denote the neighbors of u that are still available, i.e., they do not participate in the partial matching. Let $\sigma : V \to [n]$ be a permutation. The rank of $v \in V$ is $\sigma(v)$. A vertex of rank t refers to $\sigma^{-1}(t)$. Vertex v is said to have a better rank than u if $\sigma(v) < \sigma(u)$. If we imagine V sorted in the order given by σ, then v having better rank that u means that v appears before u. Vertex v is said to have best rank in $S \subseteq V$ if it has a better rank than all other vertices in S.

7.4.1 Deterministic Algorithms

We begin with what is arguably the simplest algorithm for the online BMM in the adversarial setting. Namely, we consider the natural greedy algorithm, called SimpleGreedy, where an arriving online node u is matched to the first available neighbor according to some fixed predetermined order of V. If there are no available neighbors, the simple online greedy algorithm must leave u unmatched. The pseudocode is presented in Algorithm 13.

Algorithm 13 Simple greedy algorithm for BMM.

```
procedure SimpleGreedy
    $V$ – set of offline vertices
    Fix a ranking $\sigma$ on vertices $V$
    $M \leftarrow \emptyset$
    $i \leftarrow 1$
    while $i \leq n$ do
        New online vertex $u_i$ arrives according to $\prec$ together with $N(u_i)$
        if $N_c(u) \neq \emptyset$ then                    ▷ if there is an unmatched vertex in $N(u_i)$
            ▷ select an unmatched vertex $v$ of best rank in $N_c(u_i)$
            $v \leftarrow \arg\min\{\sigma(v) : v \in N_c(u)\}$
            $M \leftarrow M \cup \{(u_i, v)\}$                          ▷ match $u_i$ with $v$
        $i \leftarrow i + 1$
```

We begin by showing that this algorithm achieves competitive ratio $\frac{1}{2}$ via a classical argument that we expect will be familiar to most readers.

Theorem 7.4.1.

$$\rho(\mathit{SimpleGreedy}) = \frac{1}{2}.$$

Proof. First, we show that the competitive ratio is no better than $\frac{1}{2}$. A matching $M \subseteq E$ is *maximal* if there is no edge $e \in E \setminus M$ that can be added to M without violating the vertex disjointness constraint. Recall, that a subset of vertices $S \subseteq U \cup V$ is a *vertex-cover* if every edge is incident to some vertex of S. The size of the smallest vertex cover is denoted by $\nu(G)$. By definition, each maximal matching gives rise to a vertex cover by including both endpoints of each edge in the matching. Thus, we have $\nu(G) \leq 2|M|$. In addition, every vertex cover has size at least the size of a matching (since edges in a matching are vertex-disjoint). In particular, we have $\nu(G) \geq \mu(G)$. Thus, we have $\mu(G) \leq 2|M|$. This part of the proof is complete by observing that the simple greedy algorithm always constructs a maximal matching.

Next, we show that the competitive ratio is at least 2. Let $V = \{v_1, v_2\}$ and suppose that the order in which the simple greedy tries to match neighbors is v_1 followed by v_2. Consider the behavior of the algorithm on the input where $U = \{u_1, u_2\}$ and $E = \{(u_1, v_1), (u_1, v_2), (u_2, v_1)\}$. Upon seeing

u_1, the simple greedy matches it to v_1, so when u_2 arrives it cannot be matched. Thus, the simple greedy finds a matching of size 1, whereas an optimal matching is $M = \{(u_1, v_2), (u_2, v_1)\}$. This construction can be replicated arbitrarily many times to obtain an asymptotic bound. □

An argument similar to the second half in the above proof can be used to show that no deterministic algorithm can achieve a competitive ratio better than 2 in the adversarial setting.

Theorem 7.4.2. *Let ALG be a deterministic algorithm for the BMM problem in the BVAM input model. Then*

$$\rho(ALG) \leq \frac{1}{2}.$$

7.4.2 A Simple Randomized Algorithm

Arguably the most natural randomized greedy algorithm matches an arriving online node u to a random available neighbor. We shall refer to this algorithm as the *natural randomized greedy* algorithm. Consider the following family of graphs. The vertices are $U = \{u_1, \ldots, u_{2n}\}$ and $V = \{v_1, \ldots, v_{2n}\}$. Each node in the first half of U is connected to each node in the second half of V by an edge. In addition, each "parallel" edge, that is (u_i, v_i), is present. Formally, we have: $E = \{(u_i, v_j) \mid i \in [n], j \in [n+1, \ldots, 2n]\} \cup \{(u_i, v_i) \mid i \in [2n]\}$. Exercise 9 asks you to prove that the natural randomized greedy algorithm achieves competitive ratio $\frac{1}{2}$ on this instance. Thus, this algorithm provides no improvement over the simple greedy algorithm. This could lead one to believe that maybe randomness does not add power in the adversarial setting. Surprisingly, there is another simple greedy strategy that, although somwehat less natural, provides a significant improvement over the competitive ratio of the natural randomized greedy.

7.4.3 The Ranking Algorithm

In this section we present and analyze the randomized Ranking algorithm. We will see that Ranking achieves competitive ratio $1 - \frac{1}{e} \approx .632$ and that this ratio is the best possible for randomized algorithms for BMM. For the purpose of this algorithm it is useful to take $V = [n]$, which can be assumed without loss of generality. With this notation, the description of Ranking is easy: initially, pick a uniformly random permutation of V and fix it for the whole duration of the online phase. In the online phase, when a vertex u arrives, match it with a vertex of best rank among the unmatched neighbors of u. The pseudocode of Ranking is given in Algorithm 14.

Algorithm 14 The Ranking algorithm for BMM.

```
procedure RANKING
    $V$ – set of offline vertices
    Pick a ranking $\sigma$ on vertices $V$ uniformly at random
    $M \leftarrow \emptyset$
    $i \leftarrow 1$
    while $i \leq n$ do
        New online vertex $u_i$ arrives according to $\prec$ together with $N(u_i)$
        if $N_c(u) \neq \emptyset$ then            ▷ if there is an unmatched vertex in $N(u_i)$
            ▷ select the vertex of best rank in $N(u_i)$
            $v \leftarrow \arg\min\{\sigma(v) : v \in N(u)\}$
            $M \leftarrow M \cup \{(u_i, v)\}$                    ▷ match $u_i$ with $v$
        $i \leftarrow i + 1$
```

Let's compare Ranking with SimpleGreedy and the natural randomized greedy. Ranking differs from SimpleGreedy in a single line: rather than picking a fixed ranking σ of offline vertices V, it picks one at random. Also while the natural randomized greedy uses fresh random coins to make decisions for each arriving online node, the Ranking algorithm "shares" the random coins among all online decisions. This sharing improves the competitive ratio, but also makes the algorithm harder to analyze. Ranking has an interesting discovery and analysis history (see the historical notes at the end of this chapter). The remainder of this section is dedicated to proving the following theorem.

Theorem 7.4.3.

$$\rho_{OBL}(\mathit{Ranking}) = 1 - \frac{1}{e} \approx .632.$$

We begin by establishing the positive result, i.e., $\rho_{\text{OBL}}(\mathit{Ranking}) \geq 1 - \frac{1}{e}$. Let $G = (U, V, E)$ be a given bipartite graph with $U = V = [n]$ (disjoint copies). Exercise 10 asks you to prove that we can assume G has a perfect matching without loss of generality. Let M^* denote some perfect matching. We will use notation $M^*(x)$ to denote the neighbor of x as given by the matching M^*. We note that a vertex of rank t is a random variable. Let p_t denote the probability over σ that the vertex of rank t in V is matched by Ranking. We are interested in computing the expected size of the matching returned by Ranking, which is given by $\sum_{t=1}^n p_t$. Our analysis of Ranking will be centered around establishing the following lemma:

Lemma 7.4.4. *For all $t \in [n]$ we have $1 - p_t \leq (1/n) \sum_{s=1}^t p_s$.*

We first assume Lemma 7.4.4 to prove that Ranking has competitive ratio $1 - \frac{1}{e}$ as follows. Observe that $p_1 = 1$, since G has a perfect matching. By induction and the lemma, it follows that $p_t \geq (1 - 1/n)(n/(n+1))^{t-1}$ for all $t \geq 2$. Thus, we have

$$\sum_{t=1}^n p_t = p_1 + \sum_{t=2}^n p_t \geq \frac{1}{n} + \left(1 - \frac{1}{n}\right) \sum_{t=1}^n \left(\frac{n}{n+1}\right)^{t-1} \geq \frac{1}{n} + \left(1 - \frac{1}{n}\right) \frac{1 - \left(\frac{n}{n+1}\right)^n}{1 - \left(\frac{n}{n+1}\right)}$$
$$= \frac{1}{n} + \left(n - \frac{1}{n}\right) \left[1 - \left(\frac{n}{n+1}\right)^n\right] \geq n \left[1 - \left(1 - \frac{1}{n+1}\right)^n\right],$$

where the first inequality follows by representing $p_1 = 1/n + (1 - 1/n)$ and absorbing the second $(1 - \frac{1}{n})$ term into the sum, and the last inequality follows since $1/n \geq 1/n(1 - (n/(n+1))^n)$. Lastly, to conclude the proof of Theorem 7.4.3, we observe that $1 - \left(1 - \frac{1}{n+1}\right)^n \to 1 - 1/e$ as $n \to \infty$.

Next, we show how to prove the lemma. Let A_t denote the set of permutations such that a vertex of rank t is matched by Ranking. Let $S_{[n]}$ denote the set of all permutations $V \to V$ and define $B_t = S_{[n]} \setminus A_t$; that is, B_t is the set of permutations such that a vertex of rank t is not matched by Ranking. We shall construct an injection of the form $[n] \times B_t \to \bigcup_{i=1}^t A_i$. This will prove the lemma. (See Exercise 7.)

The injection is defined as follows. Let $\sigma \in B_t$ and $i \in [n]$. Let v be the vertex of rank t in σ, define σ_i to be the permutation obtained by moving v into a position i and shifting other elements accordingly, so that the rank of v in σ_i is i. It is not difficult to see that this map is injective, so it remains to show that it is well defined, i.e.: for all $\sigma \in B_t$ and $i \in [n]$ we have $\sigma_i \in \bigcup_{s=1}^t A_s$. There are two cases to consider, namely, when $i \geq t$ (see Exercise 11) and when $i < t$ which we will now consider. Let u be neighbor of v in the perfect matching that we initially fixed, i.e., $M^*(u) = v$. Note that since v is not matched by Ranking with respect to σ, it follows that u *is* matched by Ranking to some vertex of rank $< t$ with respect to σ, otherwise when u arrives it would have been matched to an available neighbor of rank t, namely, v. In fact, an even stronger statement is true: u *is* matched by Ranking to a vertex of rank $\leq t$ with respect to σ_i for all $i \in [n]$. This is clearly true for $i > t$, because by moving v to a worse rank than t has no affect on how vertices of rank $< t$ are matched. Thus u remains matched to a vertex of rank $< t$. Moving v to a better rank than t, results in a new matching constructed by Ranking. The difference between the new matching and the old matching is given by an alternating path with every vertex from U being matched to a vertex of rank that is at most 1 worse than before (this is because moving v to rank $i < t$ increases the ranks of vertices between i and t by 1). Figure 7.4 is helpful in following this argument.

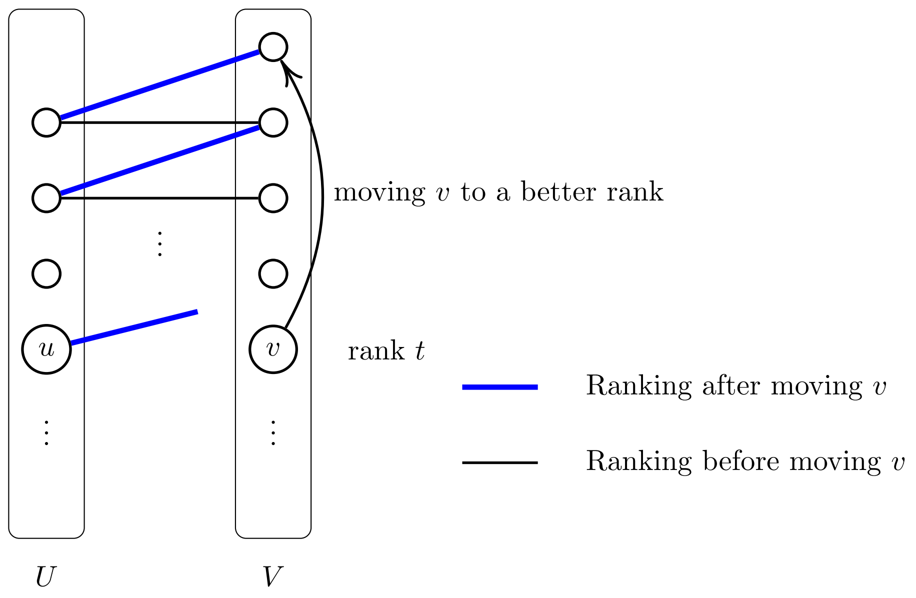

Figure 7.4: The difference between the matching constructed by Ranking on σ and the matching constructed by Ranking on σ_i with $i < t$.

This completes the proof of the positive result concerning Ranking. In the remainder of this section, we argue that the competitive ratio of any randomized algorithm for bipartite matching cannot be better than $e/(e-1)$.

Theorem 7.4.5. *Let ALG be a randomized algorithm for BMM in the BVAM input model. Then*

$$\rho_{OBL}(ALG) \leq 1 - \frac{1}{e}.$$

Proof sketch. We prove this result by exhibiting a family of bipartite graphs, which have a perfect matching, but a given randomized algorithm matches at most a $1 - 1/e$ fraction of nodes in expectation (asymptotically). In particular, this would imply that the above analysis of Ranking is tight (asymptotically).

A given randomized algorithm can be viewed as a distribution, denoted by $\mathcal{A}$, on deterministic algorithms. Let $\mathcal{P}$ denote a distribution on inputs G. For a maximization problem, Yao's minimax lemma is as follows: for any $\mathcal{A}$ and $\mathcal{P}$ we have:

$$\min_G \mathbb{E}_{A \sim \mathcal{A}}(A(G)) \leq \max_A \mathbb{E}_{G \sim \mathcal{P}}(A(G)).$$

Thereefore, to prove a negative result, it suffices to exhibit a distribution on inputs such that the best deterministic algorithm does not perform very well with respect to that distribution. Next, we define the distribution. We fix the vertex sets to be $U = \{u_1, \ldots, u_n\}$ and $V = \{v_1, \ldots, v_n\}$. For a permutation $\pi : [n] \to [n]$ we define a graph $G_\pi := (U \times V, E_\pi)$, where $E_\pi = \{(u_i, v_j) \mid \pi^{-1}(j) \geq i\}$. The distribution $\mathcal{P}$ is the distribution of G_π when π is chosen uniformly at random among all possible permutations. Observe that when π is the identity permutation, G_π is the so-called triangular graph, because the biadjacency matrix where rows are indexed by U and columns are indexed by V is the upper-triangular matrix. Any other graph from the support of $\mathcal{P}$ is obtained by permuting the columns of the upper-triangular matrix. Observe that G_π has a perfect matching given by matching u_i with $v_{\pi(i)}$.

Fix a deterministic algorithm A. By Exersize 6 we can assume that A is greedy. The intuition behind the negative result is that by choosing π randomly we hide the "true" identities of offline nodes. By "true" identities of offline nodes we mean their ranks in the triangular graph. The algorithm sees an offline node v_i, but the true identity of the node is given by $v_{\pi^{-1}(i)}$. By an easy induction, when an online node u_i arrives with k available neighbors, the true identities of those neighbors form a uniformly random set of size k from among $\{v_i, \ldots, v_n\}$. It follows that the performance of A on $\mathcal{P}$ is exactly equal to the performance of the natural randomized greedy algorithm on the triangular graph. The analysis of the natural randomized greedy on the triangular graph is carried out via a technique of partial differential equations. The technique is somewhat cumbersome to state formally, but it is very simple to apply. We give a high level overview here. The idea is to introduce random variables that track the progress of the algorithm. Let X_i denote the number of remaining online nodes at time i (it's not random, but will be useful) and Y_i denote the number of unmatched neighbors of u_i (they are among $\{v_i, \ldots, v_n\}$). We are interested in index i when Y_i drops to zero, because from i onward the u_i won't be matched. In order to keep track of Y_i we analyze how X_i and Y_i change in time. First of all, $X_{i+1} - X_i$ is always -1. Second of all, $Y_{i+1} - Y_i$ is -2 if v_i is available and was not matched, and -1 otherwise. By our earlier observations, the available neighbors of u_i form a uniformly random set of size Y_i from among $\{v_i, \ldots, v_n\}$, thus we have the following expected difference equation:

$$\mathbb{E}(Y_{i+1} - Y_i \mid Y_i) = (-2) \times \frac{Y_i}{X_i} \times \frac{Y_i - 1}{Y_i} + (-1) \times \left(1 - \frac{Y_i}{X_i} \times \frac{Y_i - 1}{Y_i}\right) = -1 - \frac{Y_i - 1}{X_i}.$$

Thus, we have $\mathbb{E}\left(\frac{Y_{i+1} - Y_i}{X_{i+1} - X_i} \mid Y_i\right) = 1 + \frac{Y_i - 1}{X_i}$ since $X_{i+1} - X_i = -1$. As the next step in the method of partial differential equations, you syntactically replace the finite difference equation with a continuous differential equation, i.e., $dy/dx = 1 + (y-1)/x$, and solve it with initial conditions $x = y = n$ to get $y = 1 + x\left(\frac{n-1}{n} + \ln \frac{x}{n}\right)$. When $y = 1$ (i.e., there is only one available neighbor) we get that the number of remaining unmatched online nodes is $x \approx n/e$. The theorems of Kurtz and Wormald show that as n goes to ∞, the difference equation is closely approximated by the differentai equation. Hence we can conclude that for large n the expected number of online nodes that do not have available neighbors is $n/e \pm o(n)$. Thus, the size of the matching found by the natural randomized greedy is bounded by $n\left(1 - \frac{1}{e}\right) \pm o(n)$. $\square$

7.5 Coloring

Given a graph $G = (V, E)$, a function $c : V \to [k]$ is called a valid k-coloring of G if for every edge $\{u, v\} \in E$ we have $c(u) \neq c(v)$. The value $c(v)$ is called a color of the vertex v. A graph is called k-colorable if there exists a valid k-coloring. For concreteness, imagine that G represents a political map of the world, where V is the set of countries and $u \sim v$ if and only if countries u and v share a border. You would like to color the countries in such a way that any two countries that share a border are colored differently[1]. The goal is to minimize k — the total number of colors used. This is the Graph Coloring problem. In this section we consider this problem in the VAM-PH input model. Formally, the problem is defined as follows:

Graph Coloring

Input: $G = (V, E, \prec)$; G is an unweighted and undirected graph; $\prec$ is the total order on V; $(v_1, N_1), \ldots, (v_n, N_n)$ is the sequence of input items, where $v_i \prec v_{i+1}$ and $N_i = N(v_i) \cap \{v_j : j < i\}$.
Output: $c : V \to [k]$ — such that $c(v_i)$ indicates the color assigned to vertex v_i.
Objective: To find c so as to minimize k – the number of colors used, subject to c being a valid coloring, i.e., $\forall \{u, v\} \in E \;\; c(u) \neq c(v)$.

Trivially, every graph on $n = |V|$ vertices is n-colorable. Some graphs, in fact, require that many colors, e.g., the complete graph; however, many graphs can be colored with much fewer colors. The question we are interested in is the following: if G is k-colorable, how many colors do you need to color G *online*? Although there are nontrivial online coloring algorithms for large values of k, e.g., $k = n^c$ for some $c \in (0, 1)$, tight bounds are not known for all values of k. In this chapter, we shall tackle one of the most basic versions of this question: case $k = 2$. In other words, in the rest of this chapter we are interested in how many colors does a deterministic algorithm need to color a 2-colorable graph online. Observe that 2-colorable graphs are precisely the bipartite graphs, where the two parts of the partition correspond to the two colors. We begin with a lower bound.

Theorem 7.5.1. *Let ALG be a deterministic online algorithm for Graph Coloring problem restricted to bipartite graphs in the VAM-PH input model. Then*

$$\rho(ALG) \geq \frac{logn}{2}.$$

Proof. We prove the following statement by induction on k: given an arbitrary sequence of input items $I_1, \ldots, I_m$, the adversary can extend the sequence with disjoint trees $T_1, T_2, T_3, \ldots$ such that ALG colors roots of the trees with k different colors and the combined size of the trees is $\leq 2^k - 1$.

Base case is $k = 1$: the adversary presents T_1 consisting of a single node. No matter what the previous items were, ALG clearly has to use 1 color to color the root of T_1. Moreover, the size of T_1 is 1.

Inductive assumption. Fix $k \geq 1$. We assume that any sequence of items $I_1, \ldots, I_m$ can be extended by an adversary with disjoint trees $T_1, T_2, \ldots$ such that ALG colors roots of the trees with k different colors and the combined size of the trees is $\leq 2^k - 1$.

Inductive step: constructing trees for $k + 1$. We are given an arbitrary sequence of items $I_1, \ldots, I_m$. We apply the inductive assumption and extend it with a sequence of trees $T_1^1, T_2^1, \ldots$ such that the roots of $T_1^1, T_2^1, \ldots$ are colored with k different colors, and the combined size of all trees is $\leq 2^k - 1$. Let S denote the set of colors used by ALG to color roots of all the T_i^1, so we have $|S| = k$. We apply the inductive assumption again after all the items corresponding to the last of T_i^1 have been presented, and start presenting $T_1^2, T_2^2, \ldots$. One of two things can happen: (1) either we present a tree in the second sequence T_i^2 such that the root of that tree is colored with a color that is not in S; or (2) we present the whole sequence of T_i^2 guaranteed by the inductive assumption, such that roots of the T_i^2 are colored with k different colors and moreover, the colors are precisely S. In case (1), we are done, since roots of the combined sequence $T_1^1, T_2^1, \ldots, T_1^2, T_2^2, \ldots$ are colored with $k+1$ distinct colors, and the total size of the sequence is $2^k - 1 + 2^k - 1 = 2^{k+1} - 2 \leq 2^{k+1} - 1$. In case (2), let $r_1, \ldots, r_\ell$ denote roots corresponding to the $T_1^2, \ldots, T_\ell^2$. The adversary then presents a brand new vertex v connected to each of $r_1, \ldots, r_\ell$. Since ALG used colors S to color $r_1, \ldots, r_\ell$ and v is connected to all of them, ALG has to use a new $(k + 1)$st color to color v in order to maintain

[1]Of course, the example here is that of a planar graph which, therefore, can be colored with at most 4 colors.

feasibility. Note that $T_1^2, \ldots, T_\ell^2$ together with the new item form a new tree of size $2^k - 1 + 1 = 2^k$. Thus, the combined size of $T_1^1, \ldots$ together with the new tree obtained from $T_1^2, \ldots$ is at most $2^k - 1 + 2^k = 2^{k+1} - 1$. This completes the inductive step.

Applying the statement to the initial empty sequence, we get that the adversary can present trees $T_1, T_2, \ldots$ such that the combined size of trees is $\le 2^k - 1$ and ALG uses at least k distinct colors to color the roots of the T_i. Letting $n = 2^k - 1$ denote the number of vertices, we see that ALG uses $k = \log(n+1) \ge \log(n)$ colors. Lastly, note that trees are bipartite which are bipartite and hence can be colored with 2 colors. □

Consider the following algorithm, called CBIP, for online Graph Coloring of bipartite graphs. When a vertex v arrives, CBIP computes the connected component C_v (so far) to which v belongs. Since the entire graph is bipartite, C_v is also bipartite. CBIP computes a partition of C_v into two blocks: S_v that contains v and $\widetilde{S}_v$ that does not contain v. In other words, $C_v = S_v \cup \widetilde{S}_v$. Note that neighbors of v are only among $\widetilde{S}_v$. Let i denote the smallest color that does not appear in $\widetilde{S}_v$. CBIP colors v with color i. Next, we show that this algorithm is $2 \log n$ competitive.

Theorem 7.5.2.

$$\rho(CBIP) \le 2 \log n.$$

Proof. Let $n(i)$ denote the minimum number of nodes that have to be presented to CBIP in order to force it to use color i for the first time. We will show that $n(i) \ge \lceil 2^{i/2} \rceil$ by induction on i.

Base cases: clearly we have $n(1) = 1$ and $n(2) = 2$.

Inductive assumption: assume that the claim holds for $i \ge 2$.

Inductive step: let v be the first vertex that is colored with color $i+1$ by CBIP. Consider C_v, S_v, and $\widetilde{S}_v$ as defined in the paragraph immediately preceding this theorem. In particular, *all colors* $1, 2, \ldots, i$ appear among $\widetilde{S}_v$. Let u be a vertex in $\widetilde{S}_v$ that is colored i. Let $C_u, S_u, \widetilde{S}_u$ be defined as before, but for the vertex u at the time that it appeared. Since u was assigned color i, then all colors $1, 2, \ldots, i-1$ appeared in $\widetilde{S}_u$. Observe that $\widetilde{S}_u \subseteq S_v$. Therefore, there exists vertex $u_1 \in \widetilde{S}_v$ colored $i-1$ and there exists vertex $u_2 \in S_v$ colored $i-1$, as well. Without loss of generality assume that $u_1 \prec u_2$. At the time that u_2 was colored the connected component of u_2, i.e., C_{u_2}, had to be disjoint from the connected component of u_1, i.e., C_{u_1}, for otherwise u_2 would not have been colored with the same color as u_1. Thus, we have $C_{u_1} \cap C_{u_2} = \emptyset$. Furthermore, we can apply the inductive assumption to each of C_{u_1} and C_{u_2} to get that $|C_{u_1}|, |C_{u_2}| \ge \lceil 2^{(i-1)/2} \rceil$. Thus, the number of vertices that have been presented prior to v is at least $|C_{u_1}| + |C_{u_2}| \ge 2\lceil 2^{(i-1)/2} \rceil \ge \lceil 2^{(i+1)/2} \rceil$. See Figure 7.5. □

7.6 Exercises

1. Finish the proof of Lemma 7.1.1. That is, prove $VAM - PH \le EAM \le VAM - FH$.

2. Does there exists a deterministic algorithm ALG for the Maximum Independent Set problem in the VAM-PH model that achieves competitive ratio strictly better than $n - 1$ ($\rho(ALG) < n - 1$) when input graphs are restricted to be trees?

3. Prove Theorem 7.3.6.

4. For both deterministic and randomized online MST algorithms, provide a specific lower bound cn instead of the asymptotic bound $\Omega(n)$ in Theorem 7.3.7.

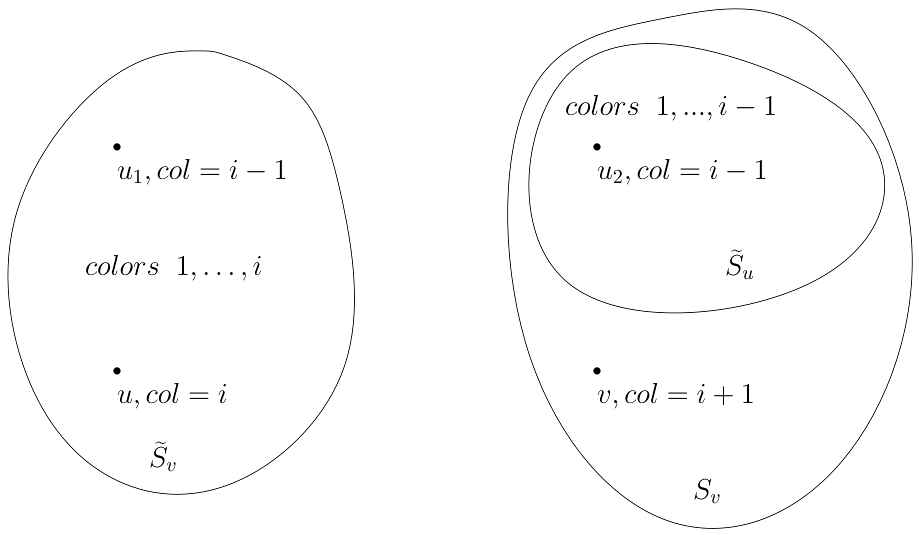

Figure 7.5: Schematic representation of the inductive step in Theorem 7.5.2.

5. What is the best competitive ratio achievable by a randomized algorithm for the TSP in the EM model?

6. Show that if there is an algorithm (either deterministic or randomized) achieving competitive ratio ρ for BMM, then there is a *greedy* algorithm (respectively, deterministic or randomized) achieving competitive ratio at least ρ.

7. Explain why it is sufficent to construct an injection $\gamma : [n] \times B_t \to \bigcup\bigcup_{i=s}^{t} A_s$ top prove Lemma 7.4.4.

8. Prove Theorem 7.4.2.

9. Write down pseudocode for the natural randomized greedy algorithm for the BMM problem (see Section 7.4.2). Prove that the natural randomized greedy algorithm has competitive ratio $\frac{1}{2}$ in the adversarial setting.

10. Prove that for every $\sigma : V \to [n]$ removing a vertex from G (could be either from U or V) either leaves the matching size returned by Ranking unchanged or reduces it by 1. Conclude that the worst-case competitive ratio of Ranking is witnessed by graphs with perfect matchings.

11. Show that the injective mapping in the proof of Lemma 7.4.4 is well defined when $i \geq t$.

7.7 Historical Notes and References

Many of the graph theoretic problems in this chapter are known to be hard to approximate even in the offline setting (under well accepted complexit assumptions). Hastad [33] shows that unless $NP = co-RP$, max clique (and therefore max independent set) cannot be efficiently approximated to within a factor $\Omega(n^{1-\epsilon})$ for any $\epsilon > 0$. The same inapproximation also holds for the min coloring problem. However, as we have indicated, these complexity based inapproximations do not necessarily imply similar negative results for online algorithms (that can have unlimited time bounds) nor do they hold for restricted classes of graphs.

The seminal $1 - \frac{1}{e}$ competitive Ranking algorithm is due to Karp, Vazirani and Vazirani [37] who also showed that the simple randomized algorithm does not asymptototically improve upon the $\frac{1}{2}$ approximation achieved by any deterministic greedy online algorithm. Several years after this publication, it was discoved that there was a technical error in the proof and this was independently observed in Goel and Mehta [28] who provide a correct proof, and Krohn and Varadarajan [39]. Subsequent alternative proofs appear in Birnbaum and Mathieu [7], Devanur, Jain and Kleinberg [17] and Alon et al [22]. We have presented the proof as provided in [7]. The proofs that the competitive ratio for the simple randomized algorithm is $\frac{1}{2}$ and that the competitive ratio for Ranking is tight appear in the seminal [37] paper. The latter proof relies on the differential equation technique initially due to Kurtz [40] and adapted to discrete probability spaces by Wormhold [49].

The graph coloring problem has a long history and indeed is a foundational problem within graph theory. The coloring problem is also a fundamental problem in the analysis of algorithms and, in particular, in online analysis. Gyárfás and Lehel [31] show that the first fit online algorithm (i.e., color each online node with the lowest numbered color available) colors trees with $O(\log n)$ colors and Bean [5] shows that online algorithms require $\Omega(\log n)$ colors establishing the asymptotic competitive ratio for trees. Irani [34] generalizes the results for trees to the much broader class of d-inductive graphs. These are graphs for which there is an ordering of the vertices $v_1, v_2, \ldots, v_n$ such that for each v_i, there are at most d vertices adjacent to v_i when restricted to vertices v_j with $j > i$. In [34], it is shown that first fit colors every d-inductive graph with at most $O(d \log n)$ colors and that there exists such graphs that require $\Omega(d \log n)$ colors establishing $\Theta(\log n)$ as the competitve ratio for this class of graphs. Note that trees are d inductive for $d = 1$, graphs with treewidth d are d-inductive, planar graphs are d-inductive for $d = 5$, and k-colorable chordal graphs are $d = k - 1$ inductive so that this is indeed a broad class of graphs. Although there are bipartite graphs for which first fit uses $\Omega(n)$ colors, Lovász, Saks and Trotter provide the $O(\log n)$ competitve algorithm shown in Theorem **??**. Whereas we now have an asymptotically tight bound for the competitive ratio of bipartitie graph coloring (i.e., online coloring of a 2-colorable graph), the most glaring open problem is to determine the competitive ratio for coloring a k-colorable graphs for small $k \geq 3$. Perhaps this is not surprising as the Blum and Karger [8] $\tilde{O}(n^{3/14})$ approximation is the best known offline approximation for 3-colorable graphs. The current state of the art for the online competitive ratio is $O(n^{1-\frac{1}{k!}})$ (and an improved $\tilde{O}(n^{2/3})$ for 3-coloroable graphs) for coloring k-colorable due to Kierstead [38].

Chapter 8

Online MaxSat and Submodular Maximization

We wil now consider two problems which are not naturally thought of as online problems, namely the max-sat problem and the unconstrained maximization of an arbitrary non montone submodular set function. We will see that these problems are related and, furthermore, the current best known *combinatorial algorithms* [1] for these two problems are the online algorithms that will be presented. We will start with the max-sat problem as it is a more well known and well studied problem.

8.1 Max-sat

The satisfiability problem (SAT) was the first problem to be shown to be *NP-complete*. While the theory of NP-completeness argues that there can not be a polynomial time algorithm for the satisfiability decision problem, it is believed that SAT can often be solve "in practice' and is therefore a common way to solve many problems that can be encoded as a SAT problem.

The corresponding NP-hard optimization problem is the max-sat problem and is perhaps the most basic constraint satisfaction problem. Like SAT, although the problem is NP hard (and there is a known $\frac{7}{8}$-hardness of approximation), there is much evidence to suggest that "in practice", max-sat problems are solved at or close to optimality. We consider the weighted max sat problem.

Max-sat
Input: Clauses $C_1 \wedge C_2 \ldots, \wedge C_m$ where each clause C_i has a weight w_i; each clause is a disjuntion of literals over propositional variables $x_1, \ldots x_n$.
Output: A truth assignment $\tau \to \{True, False\}$.
Objective: To find τ so as to maximize $\sum_{i:\tau \text{ satisfies } C_i} w_i$.

If every clause has at most (resp., exactly) k literals, the problem is called max-k-sat (resp., exact max-k-sat). In the online version of this problem, we assume that the online input items are the propositional variables $\{x_i\}$ where each input propositional variable x_i is represented by information about the clauses in which x_i appears positively and the clauses where it appears negatively. We can consider the following possible input representations for each variable x_i:

- (Input model 0): For each x_i only the names of the clauses in which x_i occurs positively and those in which it appears negatively.

[1] We have not defined the concept of a combinatorial algorithm but suffice it to say, the algorithms are relatively easy to state and implement

- (Input model 1): Input model 0 plus the lengths of those clauses.
- (Input model 2): Input model 0 plus the names of the oher varaiables occuring in each of those clauses but not their signs.
- (Input model 3): A complete description of each of the clauses in which x_i occurs.

Clearly, Input model 3 is the most general input representation and input model 0 is effectively a minimal representation. We begin with the most elementary algorithm, namely the naive randomized algorithm for which input model 0 suffices.

Algorithm 15 The NAIVE ONLINE GREEDY MAX-SAT algorithm.

```
procedure NAIVE ONLINE MAX=SAT
    $i \leftarrow 1$
    while $i \leq n$ do
        With probability $\frac{1}{2}$ set $\tau(x_i)$ to $True$ (or $False$)
    return $\tau$
```

Rather than analyze the competitive ratio of Algorithm 15, consider its (expected) *totality ratio.* The totality ratio (for a max-sat algorithm) is the worst case ratio of the (expected) value of the algorithm compared to $\sum_j w_j$, whether or not the formula is satisfiable. Clearly, the competitive ratio is at least as good as the totality ratio and equal if and only if the formula is satififsiable.

Theorem 8.1.1. *For exact max-k-sat (with just input model* 0*), the totality ratio of Algorithm 15 is equal to* $1 - \frac{1}{2^k} = \frac{2^k-1}{2^k}$. *In particular, for a formula consisting only of unit clauses, the totality ratio is* $\frac{1}{2}$.

Proof. Let $F = C_1 \wedge C_2 \wedge \ldots \wedge C_m$ with clause weights $\{w_i\}$. Let τ be a random setting of the variables and let $\mathbb{E}_\tau[F]$ denote expectation of $\sum_{i:C_i \text{ is satisfied by } \tau} w_i$. Therefore, if every clause has length k, the expected totality ratio will be $\sum_j \frac{w_j \cdot \mathbb{E}_\tau[C_j]}{w_j}$ from which the result immediately follows. Note that the method requires that the branching probabilities (in this case $\frac{1}{2}$ for each x_i) do not depend on the previous settings of x_j for $j < i$.

□

8.1.1 Derandomization by the method of conditional expectations; Johnson's algorthm rediscovered.

Theorem 8.1.2. *Using input model* 1*, the naive randomized algorithm can be used to derive a deterministic online algorithm that constructs a specific truth assignment resulting in a totality ratio is at least as good as the expected totality ratio of Algorithm 15.*

Proof. In order to de-randomize Algorithm 8.1.1, we will need input model 1. In the method of conditional expectations, we consider the randomization tree where at each node, a propositional variable x_i is being set randomly. Consider the first online input x_1. Before we set x_1 randomly, consider a clause C_j in which x_1 occurs positively and say C_j has length ℓ. As already stated we know the contribution of this clause to the expected value of the algorithm, namely $w_j \cdot (1 - \frac{1}{2^\ell})$. If we set x_1 to $TRUE$, the contribution to the expected value of the formula increases to w_j since C_j is now satisfied. On the other hand, if we set x_1 to $FALSE$, then the expected contribution decreases to $w_j \cdot (1 - \frac{1}{2^{\ell-1}})$. We can similarly determine the gain and loss for a clause in which x_1 occurs negatively. Thus when setting the value of x_1, we can determine the change in the expectation by knowing the lengths of the clauses in which x_1 occurs posiively and negatively (i.e., given the representation of x_1 using input model 1).

The method of conditional expectations is based on the observation that

$$\mathbb{E}_\tau[F] = \frac{1}{2}\mathbb{E}_{\tau'}[F|x_1 = TRUE] + \frac{1}{2}\mathbb{E}_{\tau'}[F|x_1 = FALSE]$$

where τ' is a random setting of all variables except x_1. One of the two terms must be at least as great as $\mathbb{E}_\tau[F]$ and since we can determine which term is at least as good as the other, we can know which branch to follow in the randomization tree when setting x_1. We can do this for each variable and hence find a path in the randomization tree (and hence a fixed setting for a truth assignment τ^*) which gives an approximation that is at least as good as the expected totality ratio.

□

The de-randomization (by the method of conditional expectations) leaves open a question as to the competitive ratio (in contrast to the totoality ratio) achieved by this deterministic algorihm for an arbitrary max-sat formula. Namely, one expects that it should be better than $\frac{1}{2}$. Remarkably, the de-randomized algorithm turns out to be equivalent to Johnson's algorithm, a deterministic online algorithm known since 1974.

Algorithm 16 The JOHNSON'S DETERMINISTIC MAX-SAT algorithm.

```
procedure JOHNSON'S ALGORITHM
    For all clauses $C_i$, set $w'_i := \frac{w_i}{2^{|C_i|}}$
    Let $L$ be the set of clauses in formula $F$ and $X$ the set of variables
    for $x \in X$ (or until $L$ empty) do
        Let $P = \{C_i \in L$ such that $x$ occurs positively$\}$
        Let $N = \{C_i \in L$ such that $x_i$ occurs negatively$\}$
        if $\sum_{C_i \in P} w'_i \geq \sum_{C_j \in N} w'_j$ then
            $x := true; L := L \setminus P$
            for $C_r \in N$ do
                $w'_r := 2w'_r$
        else
            $x := false; L := L \setminus N$
            for $C_r \in P$ do
                $w'_r := 2w'_r$
```

Theorem 8.1.3. *No deterministic online algorithm for max-2-sat can achieve a competitive ratio better than $\frac{2}{3}$. The competitive ratio of Johnson's max-sat algorithm is $\frac{2}{3}$*

Proof. It is easy to see why a deterministic algorithm cannot exceed competitive ratio $\frac{2}{3}$. Consider followng two instances of max-2-sat:

- $C_1 = x \vee y, C_2 = \bar{x} \vee \bar{y}, C_3 = x$
- $C_1 = x \vee y, C_2 = \bar{x} \vee \bar{y}, C_3 = \bar{x}$

Note that in each instance all three clauses can be satisfied once we know whether x or $\bar{x}$ is the third clause. However, a deterministic algorithm can be forced to satisfy at most two clauses, since satisfying both C_1 abd C_2 forces the assignment of the variable x which allows an adversary to choose C_3.

□

8.2 A randomized max-sat algorithm with competitive ratio $\frac{3}{4}$

The underlying idea for improving upon the "naive" randomized algorithm is that in setting the variables, we want to balance the weight of clauses satisfied with that of the weight of clauses that can no longer be satsified.

Let S_i be the assignment to the first i variables and let $w(SAT_i) = w(S_i)$ be the weight of satisfied clauses with respect to the partial assignment S_i of propositional variables. Let $UNSAT_i$ be the set of clauses that can no longer be satisfied given the assignment S_i; that is, the clauses that are unsatsifed by S_i and containing only variables in $\{x_1, \ldots, x_i\}$. Let $w(UNSAT_i)$ be the weight of clauses in $UNSAT_i$. In order to balance the weights $w(SAT_i)$ and $w(UNSAT_i)$, we define $B_i = \frac{1}{2}(SAT_i + W - UNSAT_i)$ where W is the total weight of all clauses, so that $w(B_i) = \frac{1}{2}(w(SAT_i) + W - w(UNSAT_i))$. Note that $w(S_0) = w(UNSAT_0) = 0$ so that $B_0 = \frac{1}{2}W$; note also that $w(S_n) = W - w(UNSAT_n) =$ the weight of the clauses satsified by the algorithm upon termination which implies $w(B_n) = w(S_n)$.

As in the de-randomization analysis of the naive randomized algorithm in Theorem 8.1.2, an online algorithm can calculate the changes in SAT_i and $UNSAT_i$ and hence B_i when setting x_i to true and when setting x_i to false. The algorithm's plan is to randomly set variable x_i so as to increase $\mathbb{E}[B_i - B_{i-1}]$ where the expectation is conditioned on the setting of the propositional variables $x_1, \ldots, x_{i-1}$ in the previous iterations. To that end, let t_i (resp. f_i) be the value of $w(B_i) - w(B_{i-1})$ when x_i is set to true (resp. false). In order to insure that the value ofthe partial solution cannot decrease in any iteration, we will need to eventually prove the basic Lemma 8.2.1 below which establishes that $f_i + t_i \geq 0$. We then have the following randomized online $\frac{3}{4}$ competitive algorithm (without explicity showing the calculations for SAT_i, $UNSAT_i$ and B_i).

Algorithm 17 The $\frac{3}{4}$ RANDOMIZED MAX-SAT algorithm.

```
procedure RANDOM 3/4 APPROXIMATION FOR ONLINE MAX-SAT
    $B_0 \leftarrow 0$
    for $i \leq n$ do
        $B_i(true) \leftarrow$ value of $B_i$ if $x_i$ is set true; $B_i(false) \leftarrow$ value of $B_i$ if $x_i$ is set false
        $t_i \leftarrow B_i(true) - B_{i-1}$; $f_i \leftarrow B_i(false) - B_{i-1}$
        if $f_i \leq 0$ then
            $\tau(x_i) := True$
        else if $t_i \leq 0$ then
            $\tau(x_i) := False$
        else if $f_i$ and $t_i$ are both positive then
            With probability $t_i$ set $\tau(x_i)$ to $True$ and $B_i \leftarrow B_i(true)$
            With probability $f_i$ set $\tau(x_i)$ to $False$ and $B_i \leftarrow B_i(false)$
                                        ▷ If $f_i = t_i = 0$, the algorithm is (arbitrarily) setting $x_i = TRUE$
                                                            ▷ Since $f_i + t_i \geq 0$, $B_i$ is non-decreasing.
    return $\tau$
```

Lemma 8.2.1. *The basic lemma*
$f_i + t_i \geq 0$

We now establish the competitive ratio of Algorithm 17.

Theorem 8.2.2. *Using input model 1, Algorithm 17 is a randomized $\frac{3}{4}$-approximation algorithm for max-sat. Furthermore, this bound is the precise competitive raio for Algorithm 17.*

For any integer program (IP) for max-sat (.e., constraining variables $x_i = TRUE = 1$ and $x_i = FALSE = 0$), we consider its fractional relaxation LP for the constraints $0 \leq x_i \leq 1$. In proving the approximation bound, we want to consider the progress of the algorithm in terms of how much an optimal fractional solution OPT_{LP} deteriorates as more and more variables become fixed to 0 or 1.

Let $OPT_{LP} = (x_1^*, \ldots, x_n^*)$ denote an optimal fractional solution for the given max-sat instance. Clearly $w(OPT_{LP}) \geq w(OPT)$ where OPT denotes an optimal solution. Let $OPT_i = (x_1, \ldots, x_i, x_{i+1}^*, \ldots, x_n^*)$ be the assignment coinciding with the partial assignment S_i of the algorithm and completed by the assignment in OPT_{LP}.

The proof of Theorem 8.2.2 relies on a key lemma showing in every iteration that (in expectation) the increase in B_i is no worse than the decrease in OPT_i whereas the key lemma depends on a supporting lemma showing that the decrease in OPT_i is bounded as a function of f_i and t_i. We will first prove the theorem given the key lemma and then prove the key lemma using the supporting lemma.

Lemma 8.2.3. *The key lemma*
$\mathbb{E}[w(OPT_{i-1}) - w(OPT_i)] \leq \mathbb{E}[B_i - B_{i-1}]$.

Lemma 8.2.4. *A supporting lemma*
If $f_i + t_i > 0$ then

$$\mathbb{E}[w(OPT_{i-1} - w(OPT_i)] \leq \max\{0, \frac{2f_it_i}{f_i + t_i}\}$$

Proof. (Theorem 8.2.2)

We first note that $S_0 = OPT_0 = OPT_{LP}$ and that $S_n = OPT_n$ is the final assignment of the algorithm. We recall that $W = \sum_j w_j$ is the total weight of all clauses, $w(B_0) = \frac{1}{2}W$ and that $B_n = S_n$. The proof establishes something stronger than stated. Namely,

$$\mathbb{E}[w(S_n)] \geq \frac{2w(OPT_{LP}) + W}{4} \geq \frac{3}{4}w(OPT_{LP}) \geq \frac{3}{4}w(OPT)$$

Summing the inequalities provided by the key lemma 8.2.3, we have

$$\sum_{i=1}^{n} \mathbb{E}[w(OPT_{i-1}) - w(OPT_i)] \leq \sum_{i=1}^{n} \mathbb{E}[B_i - B_{i-1}]$$

This then implies (by linearity of expectations) and telescoping that

$$\mathbb{E}[w(OPT_0)] - w(OPT_n)] \leq \mathbb{E}[B_n] - \mathbb{E}[B_0]$$

Restated:

$$w(OPT_{LP}) - \mathbb{E}[w(S_n)] \leq \mathbb{E}[w(S_n)] - \frac{1}{2}W$$

Rearranging and dividing by 2::

$$\frac{1}{2}(OPT_{LP}) + \frac{1}{4}W \leq \mathbb{E}[w(S_n)]$$

Which (since $OPT_{LP} \leq W$) implies the desired result:

$$\frac{3}{4}w(OPT_{LP}) \leq \frac{1}{2} + \frac{1}{4}W \leq \mathbb{E}w(S_n)]$$

□

Proof. (The key Lemma 8.2.3)
We recall the definition of t_i (resp. f_i) as the increase/decrease in B_i when setting $x_i = true$ (resp. false); namely,
$t_i = B_i(true) - B_{i-1}$ and $f_i = B_i(false) - B_{i-1}$.
There are two cases to consider:

- $f_i \leq 0$ or $t_i \leq 0$ in which case x_i is set determiniistically so that $B_i - B_{i-1} \geq 0$. The lemma follows since $f_i + t_i \geq 0$ and hence we must have $f_i t_i \leq 0$.
- $f_i > 0$ and $t_i > 0$ so that $f_i t_i > 0$ and $f_i + t_i > 0$.
  By definition
  $\mathbb{E}[B_i - B_{i-1}] = \frac{f_i}{f_i+t_i}\mathbb{E}[B_i(false) - B_{i-1}] + \frac{t_i}{f_i+t_i}\mathbb{E}[B_i(true) - B_{i-1}]$

  $\geq \frac{f_i^2+t_i^2}{f_i+t_i}$ by the supporting lemma

  $\geq \frac{2f_it_i}{f_i+t_i}$ since $f_i^2 - 2f_it_i + t_i^2 = (f_i - t_i)^2 \geq 0$.

□

Proof. (The supporting Lemma 8.2.4)
We are considering an optimal fractional solution $x_1^*, \ldots, x_n^*)$ for a given max-sat instance. Similar to the definition of $B_i(true)$ and $B_i(false)$, we can define $SAT_i(true), SAT_i(false), UNSAT_i(true)$ and $UNSAT_i(false)$.

We need to bound the change in $w(OPT_i)$ when the algorithm replaces x_i^* by $x_i = true$ and when x_i^* is replaced by $x_i = false$. Lets consider $w(OPT_{i-1}) - w(OPT_i)$ when we reverse setting $x_i = true = 1$ back to x_i^*. Any clause C_j that contributes positively to $w(OPT_{i-1}) - w(OPT_i)$ must be a clause in which $\bar{x}_i$ occurs and the increase due to such a clause C_j is bounded by $(1-x_i^*)$ times the weight of C_j. That is, the total possible increase is bounded by $(1 - x_i^*) \cdot (w(SAT_i(false)) - w(SAT_{i-1a})$. On the other hand, any clause C_j containing x_i will cause a decrease of exactly $(i-x_i^*)$ times the weight of C_j. That is, the total decrease is $(1-x_i^*)\cdot(w(UNSAT_i(false)) - w(UNSAT_{i-1})$. Hence the total change in $w(OPT_{i-1} - w(OPT_i)$ is at most

$$(1 - x_i^*) \cdot [(w(SAT_i(false)) - w(Sat_{i-1}) - w(UNSAT_i(false)) - w(UNSAT_{i-1})] = (1 - x_i^*) \cdot 2f_i.$$

By the same type of reasoning, when setting of $x_i = false$, we obtain $w(OPT_{i-1}) - w(OPT_i) \leq x_i^* \cdot 2t_i$.

So to conclude the supporting lemma, we just need to recall how the algorithm is probabilistically setting $x_i = true$ (resp. false) with probability $\frac{t_i}{t_i+f_i}$ (resp. with probability $\frac{f_i}{t_i+f_i}$. We then obtain the desired bound

$$\mathbb{E}[w_(OPT_{i-1}) - w(OPT_i)] \leq \frac{t_i}{t_i + f_i} \cdot (1 - x_i^*) \cdot 2f_i + \frac{f_i}{t_i + f_i} \cdot x_i^* \cdot 2t_i = \frac{2f_it_i}{t_i + f_i}$$

.

□

Finally we need to prove the basic fact that $f_i + t_i \geq 0$

Proof. The basic Lemma 8.2.1

□

With some care, Algorithm 8.2.2 can be implemented so as to have time complexity linear in $|F|$ where $|F|$ is the encoded length of the input formula F.

8.3 The unconstrained submodular maximization problem and the “two-sided” online algorithm

Submodular set functions occur in many applications and have been the motivation for a number of novel algorithmic ideas. Let U be a set or universe of elements. There are two alternative definitions for submodular functions. The following is perhaps the standard definition and the one that we will rely upon for this section. Namely, a submodular function $f : U \to \mathbb{R}$ satisfies the following property for all $S, T \subseteq U$:

$$f(S \cup T) + f(S \cap T) \leq f(S) + f(T), \quad \forall S \subseteq T \subset U$$

An equivalent definition which is often the more useful characterization for applications such as auction is that submodular set functions are functions which satisfy the property of diminishing marginal gains. More precisely, a function f is submodular if it satisfies the following property for all $S \subseteq T \subseteq U$ and $x \notin T$:

$$f(S \cup \{x\}) \geq f(T \cup \{x\})$$

In considering submodular functions, we usually have the additional properties that the functions are normalized (i.e., $f(\emptyset) = 0$) and monotone (i.e. $f(S) \leq f(T)$ whenever $S \subseteq T$). A common optimization problem is to find a subset S that maximizes $f(S)$ subject to some constraint. There are also examples of non monotone submodular functions and in the non-monotone case, it is also meaningful to study the unconstrained maximization problem. The two most prominent examples of non-monotone submodular functions are finding maximum cuts in graphs and directed graphs; that is, given a graph (or digraph) $G = (V, E)$, find a subset S that maximizes $|\{(u, v) \in E$ such that $u \in S, v \in V \setminus S\}|$. In the edge weighted case, the objective is to maximize $\sum_{u \in S, v \in V \setminus S} w(u, v)$ where $w(u, v)$ is the weight of an edge (u, v).

For an arbitrary (possibly non-monotone) submodular function f, the problem of finding a subset S so as to maximize $f(S)$ is referred to as the unconstrained submodular maximization problem (USM) and the problem has an interesting history as discussed in Section 8.6. For a universe U of n elements, the input for the problem consists of the 2^n possible subset values. Hence to be able to consider more efficient algorithms, we need to have a model that allows access to information about the function f without having to explicitly list all subset values. The most common type of access is called a *value oracle* which allows an algorithm to ask for the value of $f(S)$ for any specified subset S. The complexity of algorithms that utilize a value oracle is often measured by the number of oracle calls, ignoring other computational steps. For an explicity defined function (such as max cut), we do not have to assume access to an oracle and time is measured in the usual way by counting all time steps.

For the general USM problem there is an elegant “linear time” deterministic $\frac{1}{3}$ algorithm which is the best known deterministic approximation. Moreover, there is a natural “canonical randomization” of the algorithm that achieves a $\frac{1}{2}$ approximation. This is the best possible approximation in the oracle model in the sense that exponentially many value oracle calls are necessary in order to achieve a $\frac{1}{2} + \epsilon$ approximation for any $\epsilon > 0$. (There is also an explicitly given non monotone submodular function for which a $\frac{1}{2} + \epsilon$ approximation is not possible unless $RP \neq NP$.). It should be noted that the "natural" (forward) greedy and reverse greedy algorithms both fail to provide any constant approximation for the USM problem. Instead, it is possible in some sense to run the forward and reverse greedy algorithms at the same time, and even to do so in an online manner.

The $\frac{1}{3}$ deterministic and $\frac{1}{2}$ randomized approximation algorithms can be viewed as online algorithms in the sense that for both algorithms, the input items are elements in the universe U and they arrive in an adversarial online order. In principle, a value oracle algorthm can obtain the value $f(S)$ for any subset $S \subseteq U$. However, to better capture the online model as given in the templates for explicitly given problems, we might want to restrict an online oracle model so that when the i^{th} element u_i arrives, we restrict oracle calls to subsets $S \subseteq \{u_1, u_2, \ldots, u_i\}$. This turns out to be too restrictive to model a reverse greedy algorithm where we start with a solution consisting of all elements in the universe U and "greedily" eliminate elements. To capture the deterministic and randomized algorithms for USM, we can also allow "compliment oracle" calls to $\bar{f}(S) = f(U \setminus S)$.

The randomized algorithm for USM is conceptually similar to the $\frac{3}{4}$ approximation for max-sat which is not a coincidence as the randomized algorithm for USM becomes the max-sat algorithm when applied to that problem. Indeed, the randomized algorithm can be extended to privde a $\frac{3}{4}$ approximation for *submodular max-sat* where the value of a solution is a submodular function of the satisfied clauses (rather than a weighted sum). We first state the deterministic algorithm.

Algorithm 18 The DETERMINISTIC "DOUBLE-SIDED" GREEDY USM algorithm.

```
procedure DETERMINISTIC 1/3 APPROXIMATION FOR "ONLINE" USM
    $X_0 \leftarrow \emptyset; Y_0 \leftarrow U$
    ▷ The sets $X_i$ will build up a solution while the sets $Y_i$ eliminate elements in such a way that $X_i$ and $Y_i$ will agree with respect to the first $i$ elements.
    for $i \leq n$ do
        $a_i \leftarrow f(X_{i-1} \cup \{u_i\}) - f(X_{i-1})$; $b_i \leftarrow f(Y_{i-1} \setminus \{u_i\}) - f(Y_{i-1})$
        if $a_i \geq b_i$ then
            $X_i \leftarrow X_{i-1} \cup \{u_i\}; Y_i \leftarrow Y_{i-1}$    ▷ Add an element to $X$ if that is at least as good as taking away an element from $Y$
        else if $b_i > a_i$ then
            $X_i \leftarrow X_{i-1}; Y_i \leftarrow Y_{i-1} \setminus \{u_i\}$
    return $X_n = Y_n$
```

Theorem 8.3.1. *Using a value oracle that in the i^{th} iteration allows calls to $f(S)$ and $\bar{f}(S)$ for $S \subseteq U \cap \{u_1, u_2, \ldots, u_i\}$, Algorithm 18 is a deterministic $\frac{1}{3}$-approximation algorithm for the USM problem. In such a value oracle model, the algorithm can be viewed as an online algorithm. This $\frac{1}{3}$ approximation is the precise competitive raio for Algorithm 18.*

Proof. Intuitively, it seems important (if not necessary) that Algorithm 18 requires at least one of a_i or b_i to be positive if the algorithm is going to make "progress" in each iteration (or at least be non-negative so as to not not "lose ground"). The following basic fact (analogous to Lemma 8.2.1) gives us that assuarance.

Lemma 8.3.2. *$a_i + b_i \geq 0$ for $1 \leq i \leq n$.*

Proof. Note that $Y_{i-1} = (X_{i-1} \cup \{u_i\}) \cup (Y_{i-1} \setminus \{u_i\})$ and $X_{i-1} = (X_{i-1} \cup \{u_i\}) \cap (Y_{i-1} \setminus \{u_i\})$. Since f is submodular (and letting $S = X_{i-1} \cup \{u_i\}$ and $T = Y_{i-1} \setminus \{u_i\}$), it follows from the "standard submodular definition" that

$$f(Y_{i-1}) + f(X_{i-1}) \leq f(X_{i-1} \cup \{u_i\}) + f(Y_{i-1} \setminus \{u_i\})$$

Therefore, $a_i + b_i = [f(X_{i-1} \cup \{u_i\}) - f(X_{i-1})] + [f(Y_{i-1} \setminus \{u_i\}) - f(Y_{i-1})]$
$= [f(X_{i-1} \cup \{u_i\}) + f(Y_{i-1} \setminus \{u_i\})] - [f(X_{i-1}) + f(Y_{i-1})] \geq 0.$ □

Note that for all i, X_i and Y_i agree on elements $\{u_1, u_2, \ldots, u_n\}$, $X_0 = \emptyset$, $Y_0 = U$, and $X_n = Y_n$ is the solution returned by Algorithm 8.3.1. Let OPT be an optimal solution. In order to monitor the progress of the algorithm, we define $OPT_i = (OPT \cup X_i) \cap Y_i$ so that OPT_i agrees with X_i and Y_i on the first i elements and agrees with OPT on the remaining elements $\{u_{i+1}, \ldots, u_n\}$. Thus $OPT_n = X_n = Y_n$. As i increases OPT_i is non increasing while X_i is non-decreasing (using the Lemma 8.3.2.

The crux of the proof of Theorem 8.3.1 is that the decrease in $f(OPT_i)$ in any iteration is bounded by the sum of the increases in $f(X_i)$ and $f(Y_i)$. That is, we need to prove the following:

Lemma 8.3.3. *For* $(1 \leq i \leq n),\ f(OPT_{i-1}) - f(OPT_i) \leq [f(X_i) - f(X_{i-1})] + [f(Y_i) - f(Y_{i-1})]$

Proof. There are two cases to consider, namely when $a_i \geq b_i$ and $b_i \geq a_i$. The proof is similar in both cases, so we will just do the proof for $a_i \geq b_i$ and leave the other case for exercise 1.

When $a_i \geq b_i$, we have $X_i = X_{i-1} \cup \{u_i\}$ and $Y_i = Y_{i-1}$. The lemma then can be restated as:

$$f(OPT_{i-1} - f(OPT_i) \leq f(X_i) - f(X_{i-1})$$

We once again have two cases:

Case 1: $u_i \in OPT$.
This is the easier case, as the left side of the inequality is 0, while the right hand side is $a_i \geq 0$ since $a_i \geq b_i$ and $a_i + b_i \geq 0$.

Case 2: $u_i \notin OPT_i$
$u_i \notin OPT$ and hence $u_i \notin OPT_{i-1}$. We note that in this case, we also have

$$OPT_{i-1} = ((OPT \cup X_{i-1}) \cap Y_{i-1}) \subseteq Y_{i-1} \setminus u_i$$

Using the definition of submodularity, we have: we have:

$$f(Y_{i-1} \setminus \{u_i\}) + f(OPT_{i-1} \cup \{u_i\}) \geq f[((Y_{i-1} \setminus \{u_i\}) \cup (OPT_{i-1} \cup \{u_i\})] + f[((Y_{i-1} \setminus \{u_i\}) \cap (OPT_{i-1} \cup \{u_i\})]$$

$$= f(Y_{i-1}) \qquad\qquad + f(OPT_{i-1})$$

from which the required inequality follows, namely that:

$$f(OPT_{i-1}) - f(OPT_{i-1} \cup \{u_i\}) \leq f(Y_{i-1} \setminus \{u_i\}) - f(Y_{i-1}) = b_i \leq a_i$$

□

We can now conclude the proof of Theorem 8.3.1.

Using Lemma 8.3.3, we have

$$\sum_{i=1}^{n}[f(OPT_{i-1}) - f(OPT_i)] \leq \sum_{i=1}^{n}[f(X_i) - f(X_{i-1})] + \sum_{i=1}^{n}[f(Y_i) - f(Y_{i-1})]$$

These sums telescope so that

$$f(OPT_0) - f(OPT_n) \leq [f(X_n - f(X_0)] + [f(Y_n - Y_0)] \leq f(X_n) + f(Y_n)$$

Recalling the measning of OPT_0 and OPT_n) we have

$$f(OPT_0) = f(OPT) \leq f(OPT_n) + f(X_n) + f(Y_n) = 3f(X_n)$$

This concludes the proof of the deterministic approximation ratio (or competitive ratio when viewing the double-sided algorithm as an online algorithm in terms of an appropriate value oracle).
□

8.4 The natural randomization of the double sided-greedy algorithm

We now want to consider the "natural randomization" of algorithm 8.3.1 as presented in Algorithm 8.4. That is, rather than making a deterministic decision for choosing A over B based on whether or not $a \geq b$, one decides randomly in proportion to relative values of a and b.

Algorithm 19 The RANDOMIZED "DOUBLE-SIDED" GREEDY USM algorithm.

```
procedure RANDOMIZED 1/2 APPROXIMATION FOR "ONLINE" USM
    $X_0 \leftarrow \emptyset; Y_0 \leftarrow U$
    for $i \leq n$ do
        $a_i \leftarrow f(X_{i-1} \cup \{u_i\}) - f(X_{i-1})$; $b_i \leftarrow f(Y_{i-1} \setminus \{u_i\}) - f(Y_{i-1})$
        $a' \leftarrow \max\{a_i, 0\}; b' \leftarrow \max\{b_i, 0\}$
        if $a' = b' = 0$ then
            $X_i \leftarrow X_{i-1} \cup \{u_i\}$
        else if $a' + b' > 0$ then
            With probability $\frac{a'}{a'+b'}$, $X_i \leftarrow X_{i-1} \cup \{u_i\}, Y_i \leftarrow Y_{i-1}$; and
            with probabiity $\frac{b'}{a'+b'}$, $X_i \leftarrow X_{i-1}, Y_i \leftarrow Y_{i-1} \setminus \{u_i\}$
    return $X_n = Y_n$
```

We have the following Theorem proving that Algorithm 19 achieves an optimal competitve ratio (under either of the assumptions stated in Section 8.3).

Theorem 8.4.1. *The randomized algorithm 19 achieves a $\frac{1}{2}$ competitive ratio.*

Using the same definition of OPT_i, the proof will parallel the proof of Theorem 8.3.1 using the same idea of bounding the decrease in $f(OPT_{i-1} - f(OPT_i)$. Here we now need to show the following analogue of Lemma 8.3.3:

Lemma 8.4.2. *For* $(1 \leq i \leq n)$, $\mathbb{E}[f(OPT_{i-1})-f(OPT_i)] \leq \frac{1}{2}\mathbb{E}[f(X_i)-f(X_{i-1})]+[f(Y_i)-f(Y_{i-1})]$

Of course, now OPT_i, X_i and Y_i are random variables. Looking back at section 8.2, we will see that the proof of Lemma 8.4.2 will then have analogues to the lemmas used to establish Lemma 8.2.3, the key lemma in the proof of Theorem 8.2.2 which gave a $\frac{3}{4}$ competitive ratio for max-sat.

As in the proof of the $\frac{1}{3}$ deterministic bound in Theorem 8.3.1, the same telescoping argument will establish the proof for the $\frac{1}{2}$ randomized bound. That is, assuming Lemma 8.4.2, we have:

Proof.

$$\mathbb{E}[\sum_{i=1}^{n} f(OPT_{i-1}) - f(OPT_i)] \leq \frac{1}{2}\mathbb{E}[\sum_{i=1}^{n} f(X_i) - f(X_{i-1}) + \sum_{i=1}^{n}[f(Y_i) - f(Y_{i-1})]$$

By telescoping then:

$$\mathbb{E}[f(OPT_0) - f(OPT_n)] \leq \frac{1}{2}\mathbb{E}[f(X_n - f(X_0) + f(Y_n - Y_0)] \leq \frac{\mathbb{E}[f(X_n) + f(Y_n)]}{2}$$

And as before, we have $OPT_0 = OPT$ and $OPT_n = X_n = Y_n$, so that we obtain the desired competitive ratio.

□

8.5 Exercises

1. Provide the proof for the case $b_i \geq a_i$ in Lemma 8.3.3.
2. Consider the implementation (as an online algorithm) of Algorithm 18 for the explicitly defined maximum directed cut (DICUT) problem. Here the online input items are the nodes of the directed graph. In particular, what information must be provided in the representation of each node?

8.6 Historical Notes and References

8.7 Additional Related Results