Lower Bounds for Online Algorithms

Scribe: Hossein Esfandiari
Lecturer: Vahid Liaghat

November 4th

1 Introduction

What is an oline problem Unlike the offline problems that we get the whole input at the beginning, in an online problem we just have some part of the input in advance. The rest of the input are queries that arrive one by one. Upon the arrival of each request, we need to process as it is received.

In analyzing online algorithms, we usually, do not really care about the ruing time. The source of hardness here is lack of information, not complexity assumptions like $P \neq NP$. Since the algorithm does not know the rest of the input it may not be able to make the optimum decisions.

In online problems, we want to have a "good solution". How to measure a "solution"? We compare the outcome of an online algorithm with the best possible offline solution. This is the notion of "competitive ratio", which is defined below:

Definition 1.1 (competitive ratio)**.** *Suppose, someone knows the whole input and find an optimum solution (OPT). we want to compare ourselves with this optimum.*
Let OPT denote the cost of an optimal offline solution
Let Alg denote the cost of our algorithm.
α-competitive ratio is defined as follow:

$$\textit{(in minimization): } \forall \sigma, Alg(\sigma) \leq \alpha OPT(\sigma) + \gamma$$
$$\textit{(in maximisation): } Alg(\sigma) \geq \frac{1}{\alpha} OPT(\sigma).$$

Where γ is some constant independent of σ.

2 Caching Problem

When we work with data, the RAM is slow. A usual trick to speed up the access to data is to have a cache in between to work with.

Definition 2.1 (caching problem)**.** *Suppose, we have n pages of RAM and k pages of cache. The input is a sequence of request of pages in the RAM. In each step, if you read a page from the RAM, you need to store in it in the cache. Cost of a request from cache is 0 and a request from RAM is 1.*

Indeed, an online algorithm decide what to put in cach and what to remove form it.

There are several algorithms that one may consider the caching problem such as:
FIFO: First In, First Out.
LIFO: Last In First Out.
LRU: Least Recently Used.
LFFO: Least Friquency First Out.

Theorem 2.2. *LRU is k-competitive.*

Proof. Consider that, if in the first part of the input i have $k+1$ different requests, any algorithm has at least 1 fault (because size of the cache is k and it it not possible to have $k+1$ different pages in it). In the rest, we show that if LRU pays for k faults, the optimum pays for at least 1 fault. In order to prove this, we need to know the notion of phase.

Definition 2.3 (Phase:)**.** *Phase is an interval of input σ that Alg pays for exactly k faults.*

Consider an arbitrary phase $[\sigma_i, \sigma_j]$, we have two cases.

- **Case 1:** In this case we assume, for some page A, Alg pays at least 2 faults in this phase. Say, this happens at σ_s and σ_r. Remark that, from σ_s to σ_r we had k different requests excluding A. Therefore, including A we have $k+1$ different requests, and thus OPT pays at least 1.
- **Case 2:** In this case we assume that, all k faults that Alg pays for, is belong to different pages. Let A be the last request of the previous phase. Both OPT and Alg have A in their caches. If in this interval Alg pays a penalty for A, same as case 1, OPT has to pay 1 in this phase. If Alg does not have A, OPT does not have all k pages, and needs to pay at least 1.

□

Theorem 2.4. *There is no [deterministic] algorithm with competitive ratio better than k for the caching problem.*

Proof. Consider the following input:
We have exactly $k+1$ pages in the RAM. Given an algorithm Alg, request all $k+1$ pages, and then at each step request the page that Alg had recently removed form the cache. Alg never have the requested page in the cache. Therefore, the cost of Alg is $|\sigma| - k$

Now consider the following Optimum:
Opt: kick out the page that is going to be requested furthest in the future.

Suppose that *Opt* pays penalties at steps i and j. We show that $j - i + 1$. This means that $OPT \leq (|\sigma| + 1)/k$.

Consider that, *Opt* have k item in the cache at step i. In fact, it can keep the items $\sigma_{i+1}, \ldots, \sigma_{i+k-1}$. Thus, the next item that OPT may pay for is σ_{i+k}. This says that, $j >= i + k$.

Therefore, the competitive ratio is bounded by:

$$\frac{\sigma - k}{(|\sigma| + 1)/k} \simeq k.$$

□

How do we prove a lower bound of deterministic algorithms for a problem?
We assume that every deterministic algorithm A makes a sequence of decisions. Since A is deterministic, we can anticipate its decisions. We design our hard instance BASED on the deterministic decisions of A.
Can we have a fixed instance that is hard for every deterministic algorithm? A fixed hard instance that does not depends on the algorithm?
No! An oblivious hard instance does not exist! For every input, there is a 'perfect' algorithm that just outputs the optimum of that instance.
BUT, an oblivious distribution P over hard instances may exist s.t. for every deterministic algorithm A, $E[A(I)]$ is bad. Such P, destroys every deterministic algorithm. **How do we prove a lower bound of randomized algorithms for a problem?**
In this case, for each randomized algorithm R, we need to find an instance I s.t. $E_R[R(I)]$ is bad, where by E_R we mean the expectation over the randomness in R.

Lemma 2.5 (**Yao's Lemma**(maximization version))**.** *Let $S(P)$ denote the support of P. For every randomized algorithm R*

$$\min_{I \in S(p)} \{E[R(I)]\} \leq \max_{A} \{E_{I\ P}[A(I)]\}$$

where the max *is over all deterministic algorithms.*

Intuitively Yao's Lemma says that "the worst input in $P \leq$ the best deterministic algorithm w.r.t P".

3 Online Matching

In the online matching problem, we have bipartite graph $G((U, V), E)$. We have the set V in advance, which are called offline vertices. The vertices in U arrive one by one, which are called online vertices. At the time we get

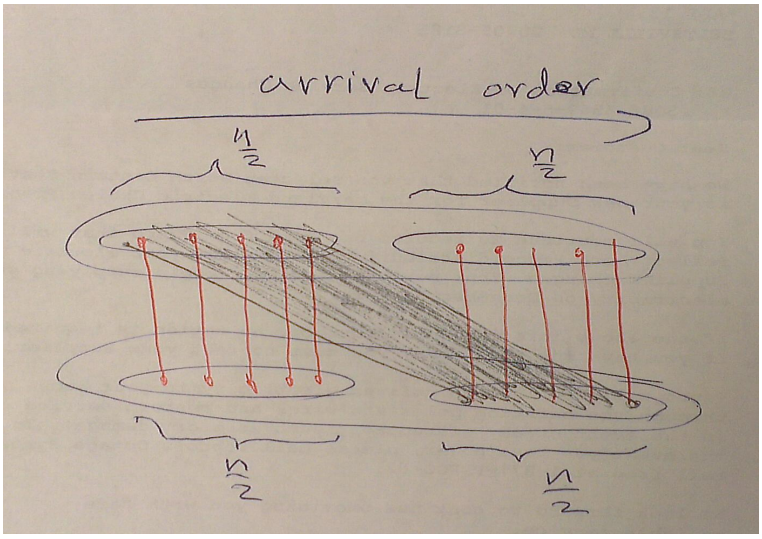
u_i, *we get all of its neighbour as well. When we receive an online vertex* u_i, we need to match u_i to one of the offline vertices. Indeed, this decision is irrevocable.

The greedy algorithm gives a maximal matching which is a two approximation. The following example shows that the greedy algorithm is not better than two competitive.

Example 3.1. *Suppose that we have two offline vertices and two online vertices. The first online vertex is connected to both offline vertices. The second online vertex is connected to one of the offline vertices, the one that is matched to the first online vertex. One can see that the outcome of greedy is* 1*, however the optimum algorithm gives a matching of size* 2*.*

One can see that the above example works for all deterministic algorithms and says that, there is no deterministic algorithm with competitive ratio better than 2 for online matching problem.

What about a randomize algorithm? What if we select a vertex uniformly at random? This algorithm is 4/3 competitive for the above example. However it is still 2 competitive in the example of figure 3.



$$Pr[u_1 \text{ is good}) = \frac{1}{n/2+1}$$

$$Pr[u_2 \text{ is good}) = Pr[u_1 \text{ is good}) \frac{1}{n/2+1} + Pr[u_1 \text{ is bad}) \frac{1}{n/2} \leq \frac{1}{n/2}$$

$$Pr[u_i \text{ is good}) = 1/(n/2 - i + 2).$$

Thus we have:

$$E(Alg) \leq \sum \frac{1}{n/2 - i + 2} + n/2 = O(logn) + n/2.$$

Remark that there is a perfect matching of size n in this graph and thus for large n the competitive ratio is 2.

Definition 3.2 (Ranking)**.** *select a random permutation π uniformly at random. At each point choose the unmatched vertex with the smallest rank.*

It is known that the ranking algorithm is $\frac{e}{e-1}$ competitive (not covered in the class).

3.1 The hardness result of online matching

Consider the following instance:

- For all i and $j >= i$, we have the edge (u_i, v_j).
- For every permutation π, $I(\pi)$ relabels the vertices in V by π, i.e. we have edges (u_i, v_{π_i}).
- The input is $I(\pi)$, where π is chosen uniformly at random.

Here is the high level approach
We define the randomized algorithm *Rand* as follow:

Definition 3.3 (Rand)**.** *At any step choose an unmatched neighbour uniformly at random.*

We show the randomize algorithm Rand that is better (or the same as) any deterministic algorithm on $I(\pi)$

Lemma 3.4. *For any algorithm A we have, $E[A(P)] \leq E[Rand(I)]$.*

Lemma 3.5. $E[Rand(I)] \leq n(1 - 1/e)$

These two together with Yao's lemma shows that for any randomize algorithm R, the approximation ratio of R is at most $e/(e-1)$.

of Lemma 3.4. Consider an arbitrary iteration i. At (the beginning of) step i, let the set of eligible vertices be $Q(i) = \{v_{\pi_i} | j \geq i$.

a) Suppose that, A (or *Rand*) have k unmatched eligible vertices. Any two subsets of size k from $Q(i)$ are the set of unmatched eligible vertices, with the same probability.

Let $P(i, k)$ be the probability that at iteration i, the number of unmatched eligible vertices is k. In fact, we have $Pr_{A(P)}(i, k) = Pr_{Rand(I)}(i, k)$.

This shows that the expected number of steps that makes the the number of unmatched eligible vertices 0 are the same in A and *Rand*. This proves Lemma 3.4. □

of Lemma 3.5. Consider the algorithm *Rand*. For each iteration i, define the following two random variables:

- $x(i) = n - i + 1 = |Q(i)|$

- $y(i)$ =number of unmatched eligible vertices.

Consider that we have:

- $\Delta x = -1$.
- $\Delta y = -2$ if v_{π_i} is unmatched and u_i will not me matched to it.
- $\Delta y = -1$ otherwise.

By Lemma 3.4(a), we have $Pr[v_{\pi_i} \text{ is unmatched}] = \frac{y(i)}{|Q(i)|}$. Therefore

$$Pr[y(i+1) - y(i) = -2] = \frac{y(i)}{x(i)} \frac{y(i)-1}{y(i)}$$

Thus, we have

$$E[\Delta y] = -1 - \frac{y(i)-1}{x(i)}$$

which gives us

$$\frac{E[\Delta y]}{E[\Delta x]} = 1 + \frac{y(i)-1}{x(i)}$$

When n goes to infinity, this can be approximated by the solution of the following differential equation:

$$\frac{dy}{dx} = 1 + \frac{y-1}{x}$$

which gives us

$$y(n+1) = \frac{n}{e} - o(n).$$

This completes the proof of the lemma. □

4 Online Set Cover

Definition 4.1 (Set Cover)**.** *we have a universe of elements E and a collection of subsets of elements of E, named F. We are asked to choose the minimum number of sets in F to cover E.*

Definition 4.2 (Online Set Cover)**.** *The offline part is (F, E). In each online step, we receive e_i with the sets covering e_i. At each step, we need to cover all elements we have seen so far. Consider that, we may not receive all of the elements in E.*

This is one of the few example that we prove a good hardness result using a time complexity assumption. Here we assume that NP can not be solved in time $n^{O(loglog(n))}$. This gives us that there is no $(1-\epsilon)log(n)$ approximation algorithm for offline set cover.

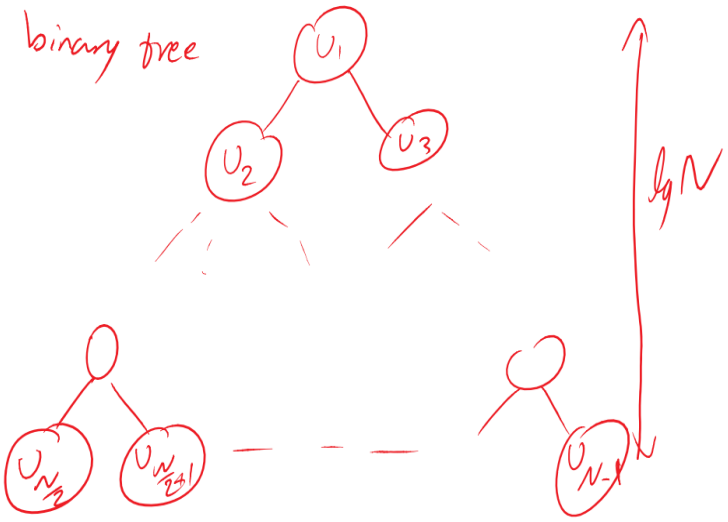
4.1 Reduction form set cover to online set cover

Let (E, F) be an offline hard instance with an optimal solution of size k, but any algorithm that runs in polynomial time, can not compute a solution better than k' (we know $k' \in \Omega(log(n))k$). We will construct an online instance (U', F', E') with an optimal solution of size k s.t. an online algorithm with solution of size better than $k'O(log(n))$ requires solving (E, F) with a cost better than k'.

This implies an $\Omega(log^2(n))$-hardness for online algorithms that run in polynomial time.

We will shortly see that for a size N, we have $|E| = m, F = n, |U'| = (N'-1)m, |F'| = \frac{N}{2}n, |E'| = m.log(N)$.

We depict these elements in the binary tree showed in figure 4.1. For every copy, at $i \in [1 \dots N-1]$, we have a path from root to that copy.



Let the ordered vector of indices P_i, denote the indices of the path form root to i.

- $P_1 =< 1 >$
- $P_i =< P_{[i/2]}, i >$

Constructing the sets:

Substantively, every leaf $i \in [N/2, \dots, N-1]$, has a copy of offline subsets F, that covers the same set of elements form $U_i, U_{i/2}, \dots, U_1$.

For every $f \in F$ and $i \in [1 \dots N-1]$, let $U_i(f)$ denote the elements of U_i that correspond to the copies of f.

For $\forall i \in [N/2, \dots, N-1]$, let F_i denote a copy of F s.t. $\forall f \in F, f_i = \{j \in [1, .., log(N) | U_{p_i(j)}(f)\}$.

For every leaf i, we construct an online sequence E'_i as follow: $E'_i =< U_{p_i(1)}, U_{p_i(2)}, \dots, U_{p_i(log(N))} >$.

By Yao's lemma, it is sufficient to show that if we pick a leaf i uniformly at random, then every online deterministic algorithm that runs in polynomial time uses $\Omega(log(n)k')$ sets to cover the online requests E'_i.

consider an arbitrary step $j \in [1, \ldots, log(N)]$ in which we receive $U_{p_i(j)}$. Consider the subtree T rooted at $U_{p_i(j)}$. Only the sets that are in the leaves of T can be useful.

We may assume that the online algorithm has to choose at least k' sets among these sets to cover $U_{p_i(j)}$. Let k_1 denote the number of selected sets in the left subtree of T, while k_2 denote the number of those at the right subtree. Consider that $k_1 + k_2 \geq k'$ which gives us $max(k_1, k_2) \geq \frac{k'}{2}$.

With out loss of generality assume that $k_2 \geq k_1$. With probability $\frac{1}{2}$ the next U-node might go to the left subtree. Hence, all the sets that contribute to k_2 will be redundant. Therefore the online solution wastes at least $\frac{k'}{2}$ sets with probability $\frac{1}{2}$. Thus, we have:

$$E[\text{size of an online solution} \geq \frac{log(N)}{2}\frac{k'}{2}$$

as desired.

Note that *Opt* is still k. we can simply choose the optimal solution at the final leaf.