1 Recap

In our last class, we discussed the Online Set Cover Algorithm. The Online Set Cover model is as follows :

- U is the huge universe of possible elements.The algorithm knows the universe ahead of time.
- S is the collection of sets whose union is the universe . m is the number of sets.
- $X \subseteq U$ is a small subset of the universe which is the online set of elements that needs to be covered, revealed over time. We have $|X| = n$.
- The algorithm has to pick a set $S_A \subseteq S$ which is the online set cover for X.

  Consider that the sequence of elements arriving over time are $e_1, e_2, \cdots e_n$, where $|X| = n$. So, when e_1 arrives, the algorithm has to include a subset $\in S$ to cover the element, without prior knowledge of the future elements.
- Goal of the algorithm : to minimise the competitive ratio, that is , precisely to pick the minimum number of sets that cover X.

Competitive Ratio = (max over all possible input X) $\left(\frac{\#sets\ chosen\ by\ the\ online\ algorithm\ for\ input\ X}{Optimal\ \#\ sets\ needed\ for\ X}\right)$

We discussed two candidate algorithms as follows:

1. **Greedy Algorithm** : A candidate greedy algorithm was proposed, which picked the largest set to cover an element. A set was chosen with the maximum cardinality, without checking how useful the set is. We discussed in the previous lecture as to why this approach was a bad choice.
2. **Fractional Algorithm** : We discussed a Fractional Algorithm which was Multiplicative-Weights inspired, where we maintained a utility(or confidence) of sets and everytime an element arrived, we increased the confidence of the sets based on their past usefulness. This algorithm had two parts. In one part, we solved the fractional algorithm which maintained the confidence of the sets and in the other, we applied the rounding algorithm to get integer solution.

2 Solving the Online Fractional Algorithm and Online Rounding Scheme

Here, we are going to first create an online fractional algorithm and then provide the online rounding scheme.

Online Fractional Algorithm

- We maintain fractional assignments for all sets in S.

  Say, for instance,the fractional solution is covering the arrived element e_1 with 3 sets S_1, S_2, S_3 in S. So, the fractional assignments are as $x_{S_1} = 0.5$, $x_{S_2} = 0.3$, $x_{S_3} = 0.2$.

- These fractional assignments are updated monotonically over time as the new elements arrive.

- Example:

  → Element e_1 arrives.

  The algorithm tries to cover it using the three sets. So, we have $x_{S_1} = 0.5$, x_{S_2}=0.3, x_{S_3}=0.2 .

  → Element e_2 arrives.

  So, our updated values for the sets are now $x_{S_1} = 0.8$, $x_{S_2} = 0.9$ and $x_{S_3} = 0.2$.

- Also, from the online nature of the algorithm,we cannot delete the sets already included in the set cover.(This can be motivated from the Airtel cell tower example discussed in the last class, where once a cell tower is installed in a location, it later cannot be uninstalled as Airtel has already paid funds to build the tower.)

Online Rounding Scheme

- We try to round the factional solution online after maintaining the fractional assignments for the sets online. We make sure that a given set, say S_1 is included with probability $\propto x_{S_1}$
- The algorithm maintains the invariant that at any time, every set is chosen in the solution with probability = $c.x_{S_i}$ (the current fractional assignment).

- Example:

  (We refer to the same example as used in the Fractional Algorithm) When element e_1 arrives and the fractional assignment is done for the sets S_1, S_2, S_3:-

  → We include set S_1 with probability $\propto c.x_{S_1}$.

  → We include set S_2 with probability $\propto c.x_{S_2}$.

  → We include set S_3 with probability $\propto c.x_{S_3}$.
  So, for each set, the algorithm randomly tosses coins with the probability $c.x_{S_2}$, and includes those sets to the solution.

  When element e_2 arrives:-

  Now, the sets will be included with the difference probability as below:

  → We include set S_1 with probability $\propto c.\Delta x_{S_1}$, where Δx_{S_1} is $0.8 - 0.5 = 0.3$.

  → We include set S_2 with probability $\propto c.\Delta x_{S_2}$, where Δx_{S_2} is $0.9 - 0.3 = 0.6$.

  → We include set S_3 with probability $\propto c.\Delta x_{S_3}$.

We discuss two theorems here.

Theorem 21.1. *There exists an online algorithm for maintaining fractional solutions such that the cost of fractional solution* $\leq O(logm).OPT$. *That is :*

$$\sum x_S^{(t)} \leq O(logm)OPT^{(t)}$$

This indicates that every time for every set of elements that arrive, the fractional cost is always less than or equals $O(logm)OPT^{(t)}$. $OPT^{(t)}$ is the optimal set cover of the elements that have arrived till time t.

Theorem 21.2. *There is a rounding scheme to select sets(based on the fractional solution), such that at any time t:*

$$\mathbb{E}[\sum Z_S^{(t)}] \leq O(logn) \sum x_S^{(t)}$$

Here Z is the indicator variable for whether S has been rounded or not.

Our algorithm first maintains a fractional solution online and second, round the fractional solution to get a good integer solution, also done online.So, putting both the theorems **Theorem 21.1** and **Theorem 21.2** together, we get:

$$\mathbb{E}[\text{Cost of Online Algorithm }] \leq O(logm.logn).OPT^{(t)}.$$

Here, m denotes the number of sets and n denotes the number of elements that show up in our online algorithm $= |X|$.

So, for any time t, the inequality holds good,which essentially shows that the competitive ratio is low. This shows that, no matter what the sequence is, the cost of our online algorithm divided by the cost of the optimal solution for that same sequence is atmost a small factor $O(logm.logn)$.

We first discuss the online Fractional Algorithm and then provide a proof for **Theorem 21.1**.

2.1 Online Fractional Algorithm

The steps for our online algorithm are as follows:

We initialise all sets to $x_s^{(0)} = \frac{1}{m}$, where m is the number of sets. Also, $\sum x_s^{(0)} = 1$.

When a new element e_t arrives,

while($\sum_{S:e_t \in S} x_s < 1$)

update $x_s = 2x_s$ if s covers e ($\forall S : e_t \in S$).

endwhile

So, the algorithm looks at the sets that cover the element e_t. If some fractional set already covers it, then we are done. Else the algorithm enters the while loop and double-set values for only those that cover e_t. So, the resulting $\{x_S\}$ values form a fractional solution at time t, called x_t. Note that if a set is useful for many elements, its value increases rapidly, as discussed in the previous class.There is a geometrical increase in value for the **repeatedly useful sets**, as below:

$$\frac{1}{m} \to \frac{2}{m} \to \frac{4}{m} \to \cdots 2.$$

So, atmost $O(logm)$ times, a set can participate in the doubling set algorithm.

We now prove **Theorem 21.1** which says that the fractional cost of online algorithm at time t over all the sets is atmost $O(logm).OPT^{(t)}$. by constructing the LP primal and dual as below.

Proof. We construct the LP primal and dual for the set cover problem as below:

Primal	**Dual**
Min $\sum x_S$	Max $\sum y_e$
subject to:	subject to:
$\sum_{S:e\in S} x_s \geq 1, \quad \forall e_1, e_2, \cdots e_t$	$\sum_{e\in S} y_e \leq 1, \quad \forall s \in S.$

To show **Theorem 1**, we show **Lemma 1**.

Lemma 1: We construct a feasible dual solution $\{\tilde{y}_e^t\}$ such that:

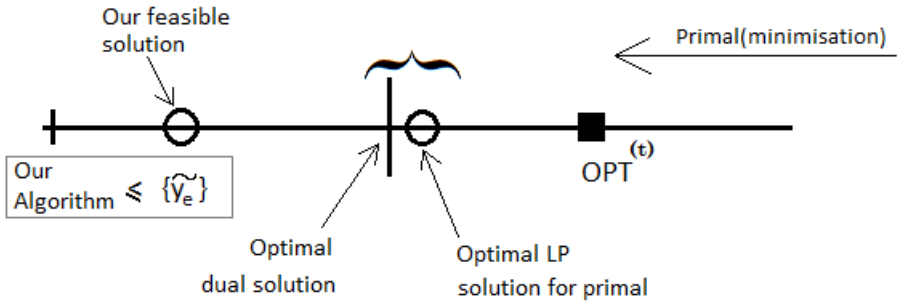
$$\sum x_s^t \ \leq \ O(logm).\sum_e \tilde{y}_e^t \ \leq \ O(logm)OPT^{(t)}.$$

Here $\sum x_s^t$ is the online fractional cost for our algorithm.So, we want to relate our algorithm to the optimal cost using duality.

2.2 Weak Duality

So, the Primal is a minimisation problem, trying to get as low cost as possible. We denote the real optimal offline solution (feasible to the primal LP) as $OPT^{(t)}$ as is shown in Figure 21.1 below.And our optimal LP solution for primal will only be smaller than the real optimal solution as LP is a relaxation of the real integer constraint.And by the strong duality theorem, the optimal dual and the optimal primal are equal.Our feasible dual is $\{\tilde{y}_e\}$ as shown in the figure,and the dual problem is a maximisation problem. So, if we find a feasible dual solution, it will only be lesser than the actual optimal dual solution, which is again less than the real integer optimal solution.So, if we can show that our primal solution is atmost $(logm)$ times our constructed dual solution,then by duality,our primal solution is atmost $(logm)$ times our actual optimal solution. So, here we are bounding the optimal solution using the constructed feasible dual solution.

Figure 21.1: Weak Duality



Now we prove **Lemma 1**:

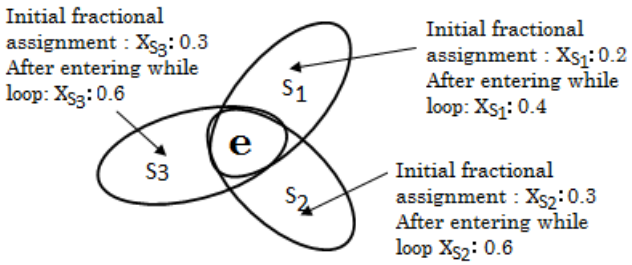
Proof. Now, how do we construct the dual? Constructing the dual has a two fold goal.

- The dual should be feasible
- The dual should help us bound the cost of the online algorithm

We reconsider the online algorithm to see how we could construct the dual from it.In the algorithm, we increase the cost whenever it enters the while loop, that is , the overall cost increases.Also, we enter the while loop when the element is not fractionally covered.The question is, how much can this cost increase be in a single iteration of the while loop?

Consider an example where there is an element e, which is covered by 3 sets with their fractional assignments as given as shown in Figure 21.2:

Figure 21.2: Example to show increase in Fractional Assignment in a single while loop iteration



So, initially, we have $\sum_{S \text{ covering } e} x_S = 0.8$.So, now while loop is entered and x_S doubles, and overall new solution has $\sum_{S \text{ covering } e} x_S = 1.6$. Also, note that no other x_S was increased, as we considered only the useful sets covering this current element e.So, overall fractional increase = 0.8 (in a single iteration for this example).

Claim 1: *In any iteration, fractional cost increases by* ≤ 1.

Using this intuition, for every while loop iteration, for the element e that caused the iteration, we increase the total objective function by 1: $y_e \rightarrow y_e + 1$,and at the same time the fractional cost incurred by our algorithm increases by atmost 1.

Let $\{\tilde{y}_e^t\}$ denote final dual variables.

By design,the final fractional cost $= \sum x_s^t \leq \sum_e \tilde{y}_e^t$. However, it may also happen that the dual solution may not be feasible.So, some y_e can be > 1 (y becomes 2 on increasing twice), and the point we need to worry about here is that the dual is subject to the constraint that $\sum_{e \in S} y_e \leq 1$.So, it may not be true that $\sum_{e \in S} \tilde{y}_e^t \leq 1$, $\forall S$.

So, we here determine that how much can this be over-violated. The saving grace is that the constraint wont be violated too much. Infact , we claim the following:

Claim 2: $\sum_{e \in S} \tilde{y}_e^t \leq (logm + 1)$, $\forall S$.

Proof. So, we fix a set S and look at all the iterations which increased the LHS $\sum_{e \in S} \tilde{y}_e^t$.The common property of all the iterations that increases the LHS is that, S must be covering the current element that caused the iteration.Also, in each such iteration, x_s value (≤ 2) is doubled.And we have already seen earlier that if x_S value exceeds 1,there wont be any further iterations for elements that are covered by S.

So, we have $\frac{1}{m} \rightarrow \frac{2}{m} \rightarrow \frac{4}{m} \rightarrow \cdots \frac{m}{m} \rightarrow \frac{2m}{m}$

So, number of iterations $= logm + 1$.

□

So, we have seen that our fractional cost at any time $t = \sum_{all\ sets} x_s^t \leq \sum_e \tilde{y}_e^t$. $\cdots\cdots(a)$

Also, note that the fractional solution $\{\frac{\tilde{y}_e^t}{logm+1}\}$ is feasible for the dual program, as it satisfies all the constraints.$\cdots\cdots(b)$

Hence (a) and (b) together implies that our fractional cost is atmost $(logm + 1)$ times the cost of feasible fractional dual solution . This implies our **Lemma 1** and in turn proves **Theorem 21.1**.

□

□

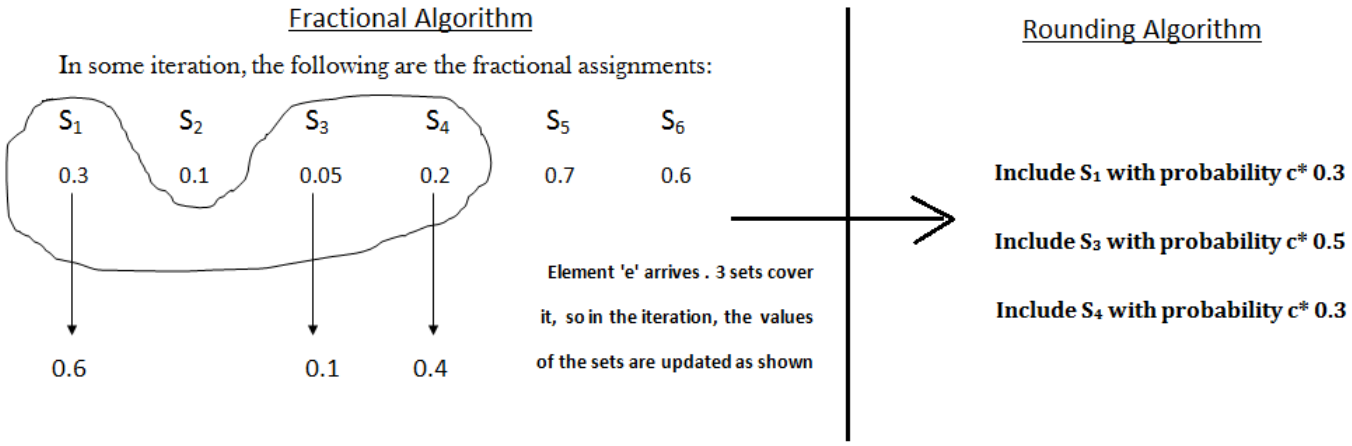
3 Summary

- ✓ So, our online algorithm is Multiplicative Weight inspired (more aggressive for more useful sets historically, rapidly increasing confidence for repeatedly useful sets).
- ✓ To show that our algorithm works, we used a dual based analysis. We analysed algorithm cost versus dual solution constructed (upto $O(logm)$).
- ✓ Duality implies that dual optimum $\leq$ Real set cover.

4 Analysis of Online Rounding Algorithm

Whenever fractional variables double, our rounding algorithm tosses coins and includes those sets randomly.We illustrate the online rounding algorithm with the example below in Figure 21.3:

Figure 21.3: Example to illustrate Rounding Algorithm



At time t, Probability[set S is included by now] $= c.x_s^t$, which is the fractional assignment.Now, we consider any element that arrived already, and we claim that:

$$\mathbb{E}[\#\text{ sets in my rounded solution which covers } e] \geq \text{C.}[x_s^t] \geq \text{C.}$$

Our rounding algorithm ensures that every element is covered atleast C times in expectation. Also note that:

$$\mathbb{E}[\text{cost of rounding solution}] = \text{c. } \sum_{all\ sets\ S} x_s^t$$

So, since we are doing independent roundings, we apply Chernoff bounds:-

Probability[a fixed element which has arrived is not covered by rounding solution] $\leq e^{-c}$.

So, if we set $c = 2logn$, the above probability $= \frac{1}{n^2}$. Thus, we see that our rounding algorithm has a very small probability that it will not cover an element that arrived. This implies **Theorem 21.2** that the above rounding algorithm maintains an online set cover , such that, at time t, it covers all elements with probability $1 - \frac{1}{n}$ and has expected cost $\leq O(logn).\sum x_s^{(t)}$.

5 Lower Bound Example

We show that no online algorithm can do better than some α factor.So, consider the following instance.

- We have a huge universe of elements.$U = \{e_1, e_2, \cdots e_L\}$. $L = l^2$.
- $S =$ all possible l-sized subsets of U; $|S| = \binom{L}{l}$

We now provide a bad input sequence to the algorithm:

We present e_1, and algorithm chooses $S_1 \rightarrow e_1$.

Again, we present e_2 not in S_1, so that forces the algorithm to choose a new set, say s_2.

Now,we present e_3 not in s_1 or s_2, so that again forces the algorithm to choose new set, say s_3.

Similarly, continue and we finally present e_l not in $s_1, s_2, \cdots s_{l-1}$, so that forces the algorithm to choose new set, say s_l.

So, overall, the algorithm cost $= l$ as l sets are picked,whereas the optimum cost for $\{e_1, e_2, \cdots e_l\} = 1$(as S must definitely have one such subset covering the elements.This means, the competitive ratio is as bad as l.

In offline set cover, if we knew all the elements ahead of time, the randomised rounding algorithm had a performance of $\leq O(logn)$. Notice that, in the above example,l is worse than $(logn)$, where gap $= l = \Omega(n)$.Hence there is a huge gap between online and offline set cover.

6 Online Routing

Consider we have a large graph G with unit capacities on edges.We can simulate high capacities by parallel links between nodes.We have online requests arriving, where a request is : a source vertex, destination vertex and their unit bandwidth requirement.

Goal of the algorithm is :

$\rightarrow$ to fix a path to route the bandwidth, which we cannot change later to ensure QoS.Once we find a path, we commit to it.

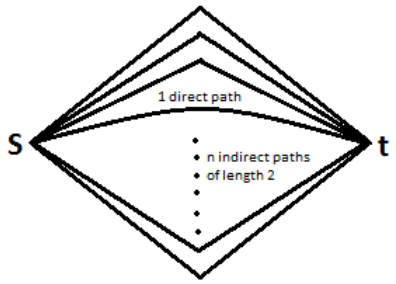
$\rightarrow$ Minimise the maximum congestion on all the edges, again to ensure QoS.We dont want the edges to be overused more than its capacity.

The algorithm is an online algorithm and hence the requests arrive with time. We suggest three candidate algorithms for this:

- We choose the shortest path to route the bandwidth, which can fail as it is an absolute shortest path.So, we fix a source and sink and all the requests are continuously routed through the same source to the same sink.

  Consider the example in Figure 21.4, which has 1 path of length 1 and n indirect paths, of length 2.So, we choose the shortest (direct) path. Suppose , 1^{st} request is $(s, t, 1)$, 2^{nd} request is $(s, t, 1)$, and so on and the n^{th} request is $(s, t, 1)$ .So, all the requests are routed through the same shortest path, and the congestion of the Shortest Path Algorithm = n.

  Figure 21.4: Graph with single direct path and 'n' indirect paths

  But optimal congestion is 1 (by routing using the indirect paths of length 2).

- We will greedily pick a path which minimises the maximum congestion after that path is chosen.One way to find this is to to throw out high congestion edges and find a new shortest path in the remaining graph.

- The right approach is to give exponential weights to the edges,and then choose the shortest path. We show the correctness of this approach in the next lecture.

  So,for instance, we can set the edge to $w(e) = \left(\frac{3}{2}\right)^{current\ load(e)}$, and then find the shortest path.This is a soft form of maximum congestion(softmax objective), and a nice middle ground between maximum congestion and length of path.