Advanced Algorithms March 24, 2022

Lecture 10: Hedging for LPs and Intro to Randomized Algorithms

Notes by Ola Svensson[1]

In this lecture we first show how to solve covering LPs approximately using the Hedge algorithm. We then give an introduction to randomized algorithms.

1 Hedging for LPs

We now turn to a nice application of the Hedge method to that of solving *covering LPs*. We start by defining these LPs.

1.1 Covering LPs

Definition 1 *A linear program of the form :*

$$\begin{aligned} \textbf{\textit{minimize}} \quad & \sum_{j=1}^{n} c_j x_j \\ \textbf{\textit{subject to}} \quad & Ax \geq b \\ & 1 \geq x_j \geq 0 \quad \forall j \end{aligned}$$

is a ***covering linear program*** *if*

$$A \in \mathbb{R}_+^{m \times n} \text{ , } b \in \mathbb{R}_+^m \text{ and } c \in \mathbb{R}_+^n$$

This definition simply ensures that all the coefficients of the constraints and of the objective function are non-negative. Notice that both the set cover relaxation and the vertex cover relaxation that we saw in class were covering LPs.

Let us now introduce an example of a covering LP that we will reuse later:

$$\begin{aligned} \textbf{minimize} \quad & x_1 + 2x_2 \\ \textbf{subject to} \quad & x_1 + 3x_2 \geq 2 \\ & 2x_1 + x_2 \geq 1 \\ & 1 \geq x_1, x_2 \geq 0 \end{aligned} \tag{1}$$

1.2 General idea

The idea of using the Hedge method for linear programming is to associate an expert with each constraint of the LP. In other words, the Hedge method will maintain a weight distribution over the set of constraints of a linear problem to solve, and to iteratively update those weights in a multiplicative manner based on the cost function at each step/day. Of course we need to define the cost function in a smart way later but let us first define the notion of an oracle and then give a small instructive example.

Initially, the Hedge method will give a weight $w_i^{(1)} = 1$ for every constraint/expert $i = 1, \dots, m$ (the number m of constraints now equals the number of experts). And at each day t, it will maintain a

[1] **Disclaimer:** These notes were written as notes for the lecturer. They have not been peer-reviewed and may contain inconsistent notation, typos, and omit citations of relevant works.

convex combination $\vec{p}^{(t)}$ of the constraints (that is defined in terms of the weights). Using such a convex combination $\vec{p}$, a natural easier LP with a single constraint is obtained by summing up all the constraints according to $\vec{p}$. Any optimal solution of the original LP is also a solution of this reduced problem, so the new problem will have at most the same cost as the previous one. We define an oracle for solving this reduced problem:

Definition 2 *An oracle that, given $\vec{p} = (p_1, \ldots, p_m) \geq \mathbf{0}$ such that $\sum_{i=1}^{m} p_i = 1$, outputs an optimal solution x^* to the following reduced linear problem:*

$$\begin{array}{ll} \textbf{\textit{minimize}} & \displaystyle\sum_{j=1}^{n} c_j x_j \\ \textbf{\textit{subject to}} & \displaystyle\left(\sum_{i=1}^{m} p_i A_i\right) \cdot x \geq \sum_{i=1}^{m} p_i b_i \\ & 1 \geq x \geq 0 \end{array}$$

This linear problem is in practice much easier to solve than the original one and we can design a simple greedy algorithm for the oracle.

For concreteness, let us consider a small example. If we apply the method we described to our example (1), we have two initials weights $w_1 = w_2 = 1$ and thus $p_1 = p_2 = \frac{1}{2}$, and we sum all the constraints:

$$\begin{gathered} p_1(x_1 + 3x_2) + p_2(2x_1 + x_2) \geq p_1 \cdot 2 + p_2 \cdot 1 \Leftrightarrow \\ 1.5x_1 + 2x_2 \geq 1.5 \\ 1 \geq x_1, x_2 \geq 0 \end{gathered}$$

By using the oracle, an optimal solution to this reduced problem is $x_1 = 1, x_2 = 0$ of cost 1. But is this a feasible solution to our original problem? By checking the constraints of the original LP:

$$\begin{array}{ll} 2x_1 + x_2 = 2 \geq 1 & \text{OK} \\ x_1 + 3x_2 = 1 < 2 & \text{not OK} \end{array}$$

We will need to go back to the original problem and *increase the weights of the unsatisfied constraints* to give them more importance and *decrease the weights of the satisfied constraint.* We will perform these updates (as done by Hedge) multiplicatively that we describe in detail in Section 1.4. We first describe how to implement the oracle in general.

1.3 Implementation of the oracle

The oracle is given an objective function **minimize** $\sum_{i=1}^{n} c_i x_i$ and only one constraint that we can rewrite as $\sum_{i=1}^{n} d_i \, x_i \geq b$ (which is the weighted sum of all constraints). We also have $1 \geq x_i \geq 0 \ \forall i$ and $c_i, d_i \geq 0 \ \forall i$ since it is a covering problem.

The idea is to assign the maximum value (namely 1) to the variables that have the highest ratio *constraint coefficient/objective coefficient*. That way we will satisfy the constraint as fast as possible while maintaining a small objective function. Then assign zero to the other variables and possibly an intermediate value for the variable that is at the limit to make the constraint satisfied. This is very similar to the most "bang-for-the-buck" greedy and it is the solution to fractional knapsack. Formally:

- Sort and relabel all variables x_i so that $d_1/c_1 \geq d_2/c_2 \geq \ldots d_n/c_n$.

- Let $k = \min\{j : \sum_{i=1}^{j} d_i \geq b\}$ and $\ell = b - \sum_{i=1}^{k-1} d_i$.
- Set $x_i := 1$ for $i = 1, ..., k-1$.
- Set $x_k := \ell / d_k$.
- Set $x_i := 0$ for $i = k+1, ..., n$.

We have,

$$\sum_{i=1}^{n} d_i \, x_i = \sum_{i=1}^{n} d_i \, x_i = \sum_{i=1}^{k-1} d_i \, x_i + d_k \, x_{\sigma(k)} + \sum_{i=k+1}^{n} d_i \, x_i = \sum_{i=1}^{k-1} d_i + \ell = b$$

Therefore the constraint is exactly satisfied. By the ordering of the variables, this ensures that we minimize the objective function as required. Having implemented the oracle, we proceed to formally define the Hedge algorithm for linear programming.

1.4 Hedge algorithm for covering LPs

As already explained, we associate an expert to each constraint of the covering LP. In addition, as hinted above, we wish to increase the weight of unsatisfied constraints and decrease the weight of satisfied constraints (in a smooth manner depending on the size of the violation or the slack). The Hedge algorithm for covering LPs thus becomes:

- Assign each constraint i a weight $w_i^{(1)}$ initialized to 1.

At each time t:

- Pick the distribution $p_i^{(t)} = w_i^{(t)} / \Phi^{(t)}$ where $\Phi^{(t)} = \sum_{i \in [N]} w_i^{(t)}$.
- Now *we define the cost vector instead of the adversary* as follows:
  - Let $x^{(t)}$ be the solution returned by the oracle on the LP obtained by using the convex combination $\vec{p}^{(t)}$ of constraints. Notice that cost of $x^{(t)}$, i.e., $c^\top x^{(t)}$, is at most the cost of an optimal solution to the original LP.
  - Define the cost of constraint i as

    $$m_i^{(t)} = \sum_{j=1}^{n} A_{ij} x_j - b_i = A_i x - b_i.$$

    Notice that we have a positive cost if the constraint is satisfied (so the weight will be decreased by Hedge) and a negative cost if it is violated (so the weight will be increased by Hedge).
- After observing the cost vector, set $w_i^{(t+1)} = w_i^{(t)} \cdot e^{-\varepsilon \cdot m_i^{(t)}}$.

Output: the average $\bar{x} = \frac{1}{T} \sum_{t=1}^{T} x^{(t)}$ of the constructed solutions.

Analysis. Since the analysis for the Hedge algorithm works with an adversarial construction of the cost vectors, it definitely holds for the cost vectors we constructed. Let

$$\rho = \max_{1 \leq i \leq m} \{\max(b_i, A_i \mathbf{1} - b_i)\},$$

be (an upper bound on) the width of our constructed cost vectors. We use

Corollary 3 *Suppose $\epsilon \leq 1$ and for $t \in [T]$, $\vec{p}^{(t)}$ is picked by Hedge and cost vectors satisfy $\vec{m}^{(t)} \in [-\rho, \rho]^N$. If $T \geq (4\rho^2 \ln N)/\epsilon^2$, then for any expert i:*

$$\frac{1}{T}\sum_{t=1}^{T} \vec{p}^{(t)} \cdot \vec{m}^{(t)} \leq \frac{1}{T}\sum_{t=1}^{T} m_i^{(t)} + 2\epsilon\,.$$

By Corollary 3, we thus have for $\epsilon \in [0,1], T \geq (4\rho^2 \ln m)/\epsilon^2$ and any constraint i that

$$\frac{1}{T}\sum_{t=1}^{T} \vec{p}^{(t)} \cdot \vec{m}^{(t)} \leq \frac{1}{T}\sum_{t=1}^{T} m_i^{(t)} + 2\epsilon\,. \tag{2}$$

Let us first consider the left-hand-side of the above expression. We have that

$$\begin{aligned}\sum_{t=1}^{T} \vec{p}^{(t)} \cdot \vec{m}^{(t)} &= \sum_{t=1}^{T}\left(\sum_i p_i^{(t)} \cdot m_i^{(t)}\right)\\ &= \sum_{t=1}^{T}\left(\underbrace{\sum_i p_i^{(t)} \cdot (A_i x^{(t)} - b_i)}_{(*)}\right)\end{aligned}$$

which is non-negative because $(*)$ is non-negative for every t since the oracle outputs a feasible solution. Using that the left-hand-side of (2) is non-negative, we get

$$-2\epsilon \leq \frac{1}{T}\sum_{t=1}^{T} m_i^{(t)} = \frac{1}{T}\sum_{t=1}^{T}\left(A_i x^{(t)} - b_i\right) = A_i\bar{x} - b_i$$

which implies that

$$A_i\bar{x} \geq b_i - 2\epsilon\,,$$

for every constraint i. So our solution $\bar{x}$ is **almost feasible**. Moreover the cost $c^\top\bar{x} = c^\top(\frac{1}{T}\sum_{t\in[T]} x^{(t)})$ is **at most the cost of an optimal solution** to the original LP since each $x^{(t)}$ has no higher cost than an optimal solution (by the properties of the oracle).

Setting the parameters for solving the set cover relaxation. For set cover we have that $\rho \leq n$ and therefore, it is sufficient to set $T = (4n^2 \ln m)/\epsilon^2$ (this can in fact be improved by a better analysis to $\approx n \ln m/\epsilon^2$). This gives a solution $\bar{x}$ that satisfies $\sum_{e\in S} x_e \geq 1 - 2\epsilon$ for every set S and the cost of $\bar{x}$ is at most that of an optimal LP solution. We can obtain a feasible (approximately optimal) solution by simply defining $x^* = \frac{1}{1-2\epsilon}\bar{x}$.

2 Introduction to Randomized Algorithms

Deterministic algorithms take input and produce output. In Randomized Algorithms, in addition to input algorithms take a source of random bits and makes random choices during execution - which leads behavior to vary even on a fixed input. For many problems a randomized algorithm is the simplest or fastest or both.

Let $p > 0$ be the probability that a (randomized) algorithm generates the correct (optimal) solution. It turns out that, even if p is much smaller than 1 we can obtain an algorithm with that succeeds with high probability just by executing the original algorithm independently many times. In particular, if we run the original algorithm k times, the probability that at least one copy succeeds is at least $1-(1-p)^k$. For small values of p one can always approximate $1-p$ with e^{-p}, and during this course we always use such an approximation. It follows that for $k = 100/p$, at least one copy finds the correct (optimal) solution with probability at least

$$1 - e^{-pk} = 1 - e^{-100}.$$

In other words, even if we have an algorithm with a small probability of success we can boost the success probability to a number very close to 1.

In this lecture, we describe a randomized algorithm for the minimum cut problem. Let us start with the definition of minimum cut problem. Let $G = (V, E)$ be a graph with $n = |V|$ vertices. Let

$$E(S, \overline{S}) := \{(u, v) : u \in S, v \notin S\},$$

be the set of edges that have exactly one endpoint in S. In the minimum cut problem, we want to partition the into two subsets which are joined together with *minimum* number of edges, i.e.,

$$\min_{\emptyset \subset S \subset V} |E(S, \overline{S})|.$$

3 Karger's Algorithm

In this lecture we will discuss Karger's [**Kar93**] algorithm for the minimum cut problem. We will show that the it finds a minimum cut in time $O(n^4 \log n)$ with high probability.

Before describing these algorithms, let us define a contraction procedure. Contraction of an edge (u, v) in G, merges the endpoints u and v to create a new (super) node uv. This reduces the total number of nodes in the graph by 1. All other edges which were previously attached to u or v are attached to the new (super) node uv. Note that this might lead to multiple parallel edges; in particular, if a node z has a edges to u and b edges to v, after contraction it will have $a + b$ edges to uv.

When we contract u, v we just remove the edge connecting u to v from the graph; in other words, we do not include the loops.

Let k be the size of the minimum cut of G; fix a minimum cut $(S^*, \overline{S^*})$. In this section we design an algorithm that finds $(S^*, \overline{S^*})$ with probability at least $1/\binom{n}{2}$.

Let us start by describing the main idea. Suppose we choose a uniformly random edge e in G. What is the probability that $e \in E(S^*, \overline{S^*})$?

Claim 4 *The probability that a uniformly random edge is in $E(S^*, \overline{S^*})$ is at most $2/n$.*

Proof Let e be a uniformly random edge in G. Obviously,

$$\mathbb{P}\left[e \in E(S^*, \overline{S^*})\right] = \frac{|E(S^*, \overline{S^*})|}{|E|} = \frac{k}{|E|}.$$

To give an upper bound we need to lower bound $|E|$. Here we use the hand-shake lemma. Let $d(v)$ be the degree of a vertex v. The hand-shake lemma says that in any graph G,

$$\sum_{v \in V} d(v) = 2|E|.$$

This is because in the LHS we count each edge twice. Now, we can lower bound $d(v)$ for any vertex v by k. This is because, for any v,

$$d(v) = |E(\{v\}, \overline{\{v\}})| \geq k.$$

In other words, the size of the cut separating v from the rest of the graph is at least k. It follows from the above two equations that $|E| \geq nk/2$. So,

$$\mathbb{P}\left[e \in E(S^*, \overline{S^*})\right] = \frac{k}{|E|} \leq \frac{k}{nk/2} = \frac{2}{n}.$$

■

Using the above lemma if we choose $n/4$ edges of G uniformly at random with probability $1/2$ none of them is in the min-cut $(S^*, \overline{S^*})$. Now, we can contract these $n/4$ edges. The new graph will have at most $3n/4$ vertices and we can recurse. After $O(\log n)$ steps, we get to a graph with just two super nodes. With probability at least $(1/2)^{O(\log n)}$ we do not contract any of the edges of $(S^*, \overline{S^*})$ throughout the process. So, the size of the cut separating the final two super-nodes is exactly k. In fact, one of these two super-nodes corresponds to S^* being contracted and the other one corresponds to $\overline{S^*}$.

In above we described the gist of the idea of Karger's algorithm. There are several missing points in the above description. First of all, when we recursively call the algorithm, we are recursively using the above claim. So, we need to make sure the size of the min-cut of G does not decrease when we contract an edge.

Fact 5 *For any graph G, when we contract an edge (u, v) the size of the minimum cut does not decrease.*

We leave the proof of the above fact as an exercise.

Secondly, when we choose $n/4$ edges uniformly at random many of them may be parallel, so after contraction the number of vertices of G may go down only by 1. To avoid running into these special cases, it suffices to contract edges one by one. That is, each time we choose a uniformly random edge of G and we contract it.

Algorithm 1 Karger's Algorithm

```
for $i = 1 \to n - 2$ do
    Choose a uniformly random edge $(u, v)$ and contract it, i.e., remove $u, v$, add a new node $uv$,
    connect all edges that go to $u$ or $v$ to the new node $uv$, also remove all loops.
end for
Return the number edges between the final two super-nodes as the size of the min-cut.
```

Theorem 6 *For any graph $G = (V, E)$ with n nodes and any min-cut $(S^*, \overline{S^*})$, Algorithm 1 returns $(S^*, \overline{S^*})$ with probability at least $\frac{2}{n(n-1)}$.*

Proof Let, A_i be the event that the edge picked in step i of the loop is not in $E(S^*, \overline{S^*})$. Observe that the algorithm succeeds in finding $(S^*, \overline{S^*})$ if $A_1, A_2, \ldots, A_{n-2}$ occur, i.e., if we never contract an edge of $E(S^*, \overline{S^*})$. So, we just need to lower bound $\mathbb{P}[A_1, A_2, \ldots, A_{n-2}]$. By Bayes rule we have,

$$\mathbb{P}[A_1, \ldots, A_{n-2}] = \mathbb{P}[A_1]\,\mathbb{P}[A_2|A_1]\,\mathbb{P}[A_3|A_1, A_2] \ldots \mathbb{P}[A_{n-2}|A_1, A_2, \ldots, A_{n-3}].$$

Now, by Theorem 4 and Theorem 5, for all i,

$$\mathbb{P}[A_i|A_1, \ldots, A_{i-1}] \geq 1 - \frac{2}{n - i + 1}.$$

Therefore,

$$\begin{aligned} \mathbb{P}[A_1, \ldots, A_{n-2}] &\geq \left(1 - \frac{2}{n}\right)\left(1 - \frac{2}{n-1}\right) \cdots \left(1 - \frac{2}{3}\right) \\ &= \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdots \frac{3}{5} \cdot \frac{2}{4} \cdot \frac{1}{3} \\ &= \frac{2}{n(n-1)} = 1/\binom{n}{2}. \end{aligned}$$

■

The following corollary is immediate from the above theorem.

Corollary 7 *Any graph has at most $\binom{n}{2}$ min-cuts.*

Proof Suppose there is a graph G that has more than $\binom{n}{2}$ min-cuts. Then, for one of those cuts, say $(S, \overline{S})$ the probability that Algorithm 1, finds $(S, \overline{S})$ is less than $1/\binom{n}{2}$. But, this is a contradiction. ■

In fact, the above bound is tight: you will give an example showing that it is tight in the exercise session.

Runtime and Boosting Probability of Success. It is not hard to see that one can run Algorithm 1 in time $O(n^2)$, but as we proved above the success probability of each execution is just $O(1/n^2)$. Using the boosting idea, we can boost the probability of success to $1 - 1/n$ by running $O(n^2 \log n)$ independent copies of the above algorithm and returning the best cut that any of the copies find.

Note that, running Karger's algorithm $n^2 \log n$ times produces a min cut with high probability; but this way we have to spend time $O(n^4 \log n)$ to find the min-cut. In the next lecture we will describe a much faster algorithm due to Karger and Stein.

References

[Kar93] D. R. Karger., "Global min-cuts in RNC, and other ramifications of a simple min-cut algorithm," in *SODA.* , 1993, pp. 21–30.