Cornell University, Fall 2021 **CS 6820: Algorithms**
Lecture notes: The multiplicative weights update method **November 2021**

The multiplicative weights update method is a family of algorithms that have found many different applications in CS: algorithms for learning and prediction problems, fast algorithms for approximately solving certain linear programs, and hardness amplification in complexity theory, to name a few examples. It is a general and surprisingly powerful iterative method based on maintaining a vector of state variables and applying small multiplicative updates to the components of the vector to converge toward an optimal solution of some problem. These notes introduce the basic method and explore two applications: online prediction problems and packing/covering linear programs.

1 Investing and combining expert advice

In this section we analyze two inter-related problems. The first problem is an *investment problem* in which there are n stocks numbered $1, \ldots, n$, and an investor with an initial wealth $W(0) = 1$ must choose in each period how to split the current wealth among the securities. The price of each stock then increases by some factor between 1 and $1 + \varepsilon$ (a different factor for each stock, not known to the investor at the time of making her investment) and the wealth increases accordingly. The goal is to do nearly as well as buying and holding the single best-performing stock.

Let's introduce some notation for the investment problem. At time $t = 1, \ldots, T$, the investor chooses to partition her wealth into shares $x_1(t), \ldots, x_n(t)$. These shares must be non-negative (short-selling stocks is disallowed) and they must sum to 1 (the investor's money must be fully invested).

$$\sum_{i=1}^{n} x_i(t) = 1 \quad \forall t$$
$$x_i(t) \geq 0 \quad \forall t \; \forall i$$

We summarize these constraints by saying that the vector $\mathbf{x}(t) = (x_1(t), \ldots, x_n(t))$ belongs to the probability simplex $\mathbf{\Delta}(n)$. As stated earlier, the amount by which the price of stock i appreciates at time t is a number between 1 and $1 + \varepsilon$. Denote this number by $(1 + \varepsilon)^{r_i(t)}$.

If we let $W(t)$ denote the investor's wealth at the end of round t, then the wealth at the start of round t is $W(t-1)$ and the amount invested in stock i is $x_i(t)W(t-1)$. We thus have

$$W(t) = \sum_{i=1}^{n} (1 + \varepsilon)^{r_i(t)} x_i(t) W(t-1).$$

The prediction problem that we will study bears some superficial similarities to the investment problem. (And, as we will see, the similarity extends much deeper.) In this problem there is a gambler and n "experts". At time $t = 1, \ldots, n$, the gambler bets \$1 by dividing it among the experts. Once again, we will use $\mathbf{x}(t) \in \mathbf{\Delta}(n)$ to denote the vector representing how the gambler splits her bet at time t. Each expert generates a payoff at time t denoted by $r_i(t)$, and the gambler's payoff is the dot product $\mathbf{x}(t) \cdot \mathbf{r}(t)$. In other words, placing a bet of $x_i(t)$ on expert i yields a payoff of $x_i(t)r_i(t)$ in round t, and the gambler's total payoff is the sum of these payoffs. The goal is to gain nearly as much payoff as the strategy that always bets on the single best-performing expert.

There are some clear relationships between the two problems, but also some clear differences, chiefly that payoffs accumulate multiplicatively in one problem, and additively in the other. Consequently, the relationship between the problems becomes clearer when we take the logarithm of the investor's wealth. For example, if the investor follows the strategy of buying and holding stock i, her wealth after time t would satisfy

$$\begin{aligned} W(t) &= \prod_{i=1}^{t} (1+\varepsilon)^{r_i(t)} \\ \log_{1+\varepsilon} W(t) &= \sum_{i=1}^{t} r_i(t) = r_i(1:t) \end{aligned}$$

where the last equation should be interpreted as the definition of the notation $r_i(1:t)$. Similarly, if the investor follows the "uniform buy-and-hold strategy" of initially investing $1/n$ in each stock, and never performing any trades after that, then her investment in stock i after time t is given by $\frac{1}{n}(1+\varepsilon)^{r_i(1:t)}$, and her log-wealth after time t satisfies

$$\log_{1+\varepsilon} W(t) = \log_{1+\varepsilon}\left(\frac{1}{n}\sum_{i=1}^{n}(1+\varepsilon)^{r_i(1:t)}\right).$$

Letting i denote an arbitrary stock (e.g. the best-performing one), the wealth of the uniform buy-and-hold strategy satisfies

$$\log_{1+\varepsilon} W(t) > \log_{1+\varepsilon}\left(\frac{1}{n}(1+\varepsilon)^{r_i(1:t)}\right) = r_i(1:t) - \log_{1+\varepsilon}(n).$$

This already gives a useful bound on the additive difference in log-wealth between the uniform buy-and-hold strategy and the strategy that buys and holds the single best-performing stock.

An important relationship between the investment and prediction problems is expressed by the following calculation, which applies to an investor who distributes her wealth at time t using vector $\mathbf{x}(t)$. The log-wealth after time t then satisfies the following.

$$\begin{aligned} \log_{1+\varepsilon} W(t) &= \log_{1+\varepsilon}\left(\sum_{i=1}^{n}(1+\varepsilon)^{r_i(t)}x_i(t)W(t-1)\right) \\ &= \log_{1+\varepsilon} W(t-1) + \log_{1+\varepsilon}\left(\sum_{i=1}^{n}(1+\varepsilon)^{r_i(t)}x_i(t)\right) \\ &\le \log_{1+\varepsilon} W(t-1) + \log_{1+\varepsilon}\left(\sum_{i=1}^{n}(1+\varepsilon r_i(t))x_i(t)\right) \\ &= \log_{1+\varepsilon} W(t-1) + \frac{\ln\left(1+\varepsilon\sum_{i=1}^{n} r_i(t)x_i(t)\right)}{\ln(1+\varepsilon)} \\ &\le \log_{1+\varepsilon} W(t-1) + \frac{\varepsilon}{\ln(1+\varepsilon)}\mathbf{x}(t)\cdot\mathbf{r}(t). \end{aligned}$$

Summing over $t = 1, \ldots, T$, we find that

$$\log_{1+\varepsilon} W(T) \le \frac{\varepsilon}{\ln(1+\varepsilon)}\sum_{t=1}^{T}\mathbf{x}(t)\cdot\mathbf{r}(t),$$

which implies a relation between the log-wealth of an investor using strategy $\mathbf{x}(1), \ldots, \mathbf{x}(T)$ and the payoff of a gambler using the same strategy sequence in the prediction problem.

Recall that the uniform buy-and-hold strategy was actually a pretty good strategy for the investor. This implies that the corresponding prediction strategy is pretty good for the gambler. In the gambling context (also known as the *predicting from expert advice* context) the strategy that corresponds to uniform-buy-and-hold is known as the **multiplicative weights algorithm** or **Hedge**. At time t it predicts the vector $\mathbf{x}(t)$ whose i^{th} component is given by

$$x_i(t) = \frac{(1+\varepsilon)^{r_i(1:t)}}{\sum_{j=1}^{n}(1+\varepsilon)^{r_j(1:t)}}.$$

We have seen that the payoff of the multiplicative weights algorithm satisfies

$$\begin{aligned}\sum_{t=1}^{T} \mathbf{x}(t)\cdot\mathbf{r}(t) &\geq \frac{\ln(1+\varepsilon)}{\varepsilon}\log_{1+\varepsilon} W(T) \\ &\geq \frac{\ln(1+\varepsilon)}{\varepsilon} r_i(1:t) - \frac{\ln(1+\varepsilon)}{\varepsilon}\log_{1+\varepsilon} n \\ &> (1-\varepsilon) r_i(1:t) - \frac{\ln n}{\varepsilon}.\end{aligned}$$

The last line used the identity $\frac{1}{x}\ln(1+x) > 1-x$ which is valid for any $x > 0$. (See the proof in Appendix A.)

The role of the parameter $\varepsilon > 0$ in the two problems deserves some discussion. In the investment problem, ε is a parameter of the model, and one can either treat it as an assumption about the way stock prices change in discrete time — never by a factor of more than $1+\varepsilon$ from one time period the next — or one can instead imagine that stock prices change continuously over time, and the parameter ε is determined by how rapidly the investor chooses to engage in trading. In the prediction problem, on the other hand, the model does not define ε and it is instead under the discretion of the algorithm designer. There is a tradeoff between choosing a small or a large value of ε, and the performance guarantee

$$\sum_{t=1}^{T} \mathbf{x}(t)\cdot\mathbf{r}(t) > (1-\varepsilon) r_i(1:t) - \frac{\ln n}{\varepsilon}$$

neatly summarizes the tradeoff. A smaller value of ε allows the gambler to achieve a better multiplicative approximation to the best expert, at the cost of a larger additive error term. In short, ε can be interpreted as a "learning rate" parameter: with a small ε (slow learning rate) the gambler pays a huge start-up cost in order to eventually achieve a very close multiplicative approximation to the optimum; with a large ε the eventual approximation is more crude, but the start-up cost is much cheaper.

2 Solving linear programs with multiplicative weights

This section presents an application of the multiplicative-weights method to solving packing and covering linear programs. When A is a non-negative matrix and p, b are non-negative vectors, the following pair of linear programs are called a *packing* and a *covering* linear program, respectively.

$$\begin{array}{llll} \max & p^{\intercal} y & \min & b^{\intercal} x \\ \text{s.t.} & Ay \preceq b & \text{s.t.} & A^{\intercal} x \succeq p \\ & y \succeq 0 & & x \succeq 0 \end{array}$$

Note that the covering problem is the dual of the packing problem and vice-versa. To develop intuitions about these linear programs it is useful to adopt the following metaphor. Think of the entries a_{ij} of matrix A as denoting the amount of raw material i needed to product one unit of product j. Think of b_i as the total supply of resource i available to a firm, and p_j as the unit price at which the firm can sell product j. If the vector y in the first LP is interpreted as the quantity of each product to be produced, then the vector Ay encodes the amount of each resource required to produce y, the constraint $Ay \preceq b$ says that the firm's production is limited by its resource budget, and the optimization criterion (maximize $p^{\intercal} y$) specifies that the firm's goal is to maximize revenue.

The dual LP also admits an interpretation within this metaphor. If we think of the vector x as designating a unit price for each raw material, then the constraint $A^{\intercal} x \succeq p$ expresses the property that for each product j, the cost of resources required to produce one unit of j exceeds the price at which it can be sold. Therefore, if a vector x is feasible for the dual LP, then the cost of obtaining the resource bundle b at prices x (namely, $b^{\intercal} x$) exceeds the revenue gained from selling any product bundle y that can be made from the resources in b (namely, $p^{\intercal} y$). This reflects weak duality, the assertion that the maximum of $p^{\intercal} y$ over primal-feasible vectors y is less than or equal to the minimum of $b^{\intercal} x$ over dual-feasible vectors x. Strong duality asserts that they are in fact equal; the algorithm we will develop supplies an algorithmic proof of this fact.

The multiplicative weights method for solving packing and covering linear programs was pioneered by Plotkin, Shmoys, and Tardos and independently by Grigoriadis and Khachiyan. The version we present here differs a bit from the Plotkin-Shmoys-Tardos exposition of the algorithm, in order to leverage the connection to the multiplicative weights method for online prediction, as well as to incorporate a "width reduction" technique introduced by Garg and Könemann. We will make the simplifying assuymption that $b = B \cdot \mathbf{1}$, for some scalar $B > 0$. We can always manipulate the linear program so that it satisfies this assumption, by simply changing the units in which resource consumption is measured. Also, after rescaling the units of resource consumption (by a common factor) we can assume that $0 \le a_{ij} \le 1$ for all i, j — possibly at the expense of changing the value of B.

The algorithm is as follows.

Algorithm 1 Multiplicative weights algorithm for packing/covering LP's

```
1: Given: parameters $\epsilon, \delta > 0$.
2: Initialize: $t \leftarrow 0, Y \leftarrow 0$.            // $Y$ is a vector storing $\delta(y_1 + \cdots + y_t)$.
3: while $AY \prec B\mathbf{1}$ do
4:     $t \leftarrow t + 1$.
5:     $\forall i = 1, \ldots, n \quad (x_t)_i \leftarrow \frac{(1+\varepsilon)^{(AY)_i/\delta}}{\sum_{j=1}^{n} (1+\varepsilon)^{(AY)_j/\delta}}$.
6:     $y_t \leftarrow \arg\min_{y \in \mathbf{\Delta}(n)} \left\{ \frac{x_t^{\intercal} A y}{p^{\intercal} y} \right\}$.
7:     $Y \leftarrow Y + \delta\, y_t$.
8: end while
```

The vector x_t is being set using the multiplicative weights algorithm with payoff sequence $r_t = Ay_t$. In the expression defining y_t, the ratio $\frac{x_t^\intercal Ay}{p^\intercal y}$ can be interpreted as the cost-benefit ratio of producing a product randomly sampled from the probability distribution y. The arg min of this ratio will therefore be a point-mass distribution concentrated on the single product with the smallest cost-benefit ratio, i.e. one can always choose the vector y_t to have only one non-zero entry.

To analyze the algorithm, we begin with the performance guarantee of the multiplicative weights prediction algorithm. Let T be the time when the algorithm terminates.

$$\sum_{t=1}^{T} x_t^\intercal Ay_t \geq (1-\varepsilon)\max_i \left\{ \sum_{t=1}^{T} (Ay_t)_i \right\} - \frac{\ln n}{\varepsilon} \geq (1-\varepsilon)\cdot\frac{B}{\delta} - \frac{\ln n}{\varepsilon}. \tag{1}$$

(The second inequality is justified by the stopping condition for the algorithm.)

Next we work on deriving an upper bound on the quantity on the left side of (1). The definition of y_t implies that for any other vector y,

$$\frac{x_t^\intercal Ay}{p^\intercal y} \geq \frac{x_t^\intercal Ay_t}{p^\intercal y_t}. \tag{2}$$

Setting y in this inequality equal to y_*, the optimum solution of the primal linear program, we find that

$$\frac{(x_t^\intercal Ay_*)(p^\intercal y_t)}{p^\intercal y_*} \geq x_t^\intercal Ay_t. \tag{3}$$

Let $\bar{x}$ denote the weighted average of the vectors $x_1, \ldots, x_T$, averaged with weights $\frac{p^\intercal y_t}{p^\intercal Y}$:

$$\bar{x} = \frac{\delta}{p^\intercal Y}\sum_{t=1}^{T}(p^\intercal y_t)x_t. \tag{4}$$

Summing (3) over $t = 1, \ldots, T$ and using the definition of $\bar{x}$, we obtain

$$\frac{1}{\delta}\cdot\frac{p^\intercal Y}{p^\intercal y_*}\cdot\bar{x}^\intercal Ay_* \geq \sum_{t=1}^{T} x_t^\intercal Ay_t. \tag{5}$$

Each of the vectors $x_1, \ldots, x_T$ satisfies $x_t^\intercal \mathbf{1} = 1$, so their weighted average $\bar{x}$ satisfies $\bar{x}^\intercal\mathbf{1} = 1$ as well. Using the inequality $Ay_* \preceq B\mathbf{1}$, which follows from primal feasibility of y_*, we now deduce that

$$B = \bar{x}^\intercal(B\mathbf{1}) \geq \bar{x}^\intercal Ay_*. \tag{6}$$

Combining (1), (5), (6) we obtain

$$\frac{p^\intercal Y}{p^\intercal y_*}\cdot\frac{B}{\delta} \geq (1-\varepsilon)\cdot\frac{B}{\delta} - \frac{\ln n}{\varepsilon} \tag{7}$$

$$\frac{p^\intercal Y}{p^\intercal y_*} \geq 1 - \varepsilon - \frac{\delta \ln n}{\varepsilon B}. \tag{8}$$

Thus, if we want to ensure that the algorithm computes a vector Y which is at least a $(1-2\varepsilon)$-approximation to the optimum of the linear program, it suffices to set $\delta = \frac{\varepsilon^2 B}{\ln n}$.

To bound the number of iterations of the algorithm's while loop, let γ be a parameter such that every column of A has an entry bounded below by γ. Then, in every iteration some entry of the vector AY increases by at least $\gamma\delta$. Since the algorithm stops as soon as some entry of AY exceeds B, the number of iterations is bounded above by $nB/(\gamma\delta)$. Substituting $\delta = \frac{\varepsilon^2 B}{\ln n}$, this means that the number of iterations is bounded by $(n \log n)/(\varepsilon^2\gamma)$.

3 Multicommodity Flow

Now it's time to see how these ideas are applied in the context of a concrete optimization problem, multicommodity flow, which is a generalization of network flow featuring multiple source-sink pairs.

3.1 Problem definition

A multicommodity flow problem is specified by a graph (directed or undirected) G, a collection of k source-sink pairs $\{(s_i, t_i)\}_{i=1}^k$, and a non-negative capacity $c(e)$ for every edge $e = (u, v)$. A *multicommodity flow* is a k-tuple of flows $(f_1, \dots, f_k)$ such that f_i is a flow from s_i to t_i, and the superposition of all k flows satisfies the edge capacity constraints in the sense that for every edge $e = (u, v)$ we have

$$\textit{[Undirected case]} \qquad c(e) \geq \sum_{i=1}^{k} |f_i(u, v)|$$

$$\textit{[Directed case]} \qquad c(e) \geq \sum_{i=1}^{k} \max\{0, f_i(u, v)\}$$

There are two different objectives that are commonly studied in multicommodity flow theory.

Maximum throughput: Maximize $\sum_{i=1}^k |f_i|$.

Maximum concurrent flow: Maximize $\min_{1 \leq i \leq k} |f_i|$.

3.2 The case of uniform edge capacities

It is fairly straightforward to apply the multiplicative-weights algorithm to solve multicommodity flow problems in graphs where all edges have identical capacity. (We will consider the general case, in which edges don't necessarily have identical capacity, in the next section of these notes.) Letting B denote the capacity of each edge, the multicommodity flow problem can be expressed by the following linear program with exponentially many variables y_P, where P ranges over all paths that join some source-sink pair (s_i, t_i).

$$\begin{array}{ll} \max & \sum_P y_P \\ \text{s.t.} & \sum_{P: e \in P} y_P \leq c(e) \quad \forall e \\ & y_P \geq 0 \quad \forall P \end{array} \tag{9}$$

This problem is a packing linear program. The objective function of the packing problem has coefficient vector $p = \mathbf{1}$, and the constraint matrix A has entries $a_{ij} = 1$ if edge e_i belongs to path P_j. Note that every column of A contains at least one entry equal to 1, so this problem has $\gamma = 1$. Thus, the multiplicative weights algorithm finds a $(1 - 2\varepsilon)$-approximation of the optimal solution in at most $m \log n / \varepsilon^2$ iterations, where m denotes the number of edges. (In previous sections we referred to the number of constraints in the packing LP as n rather than m, but it would be too confusing to use the letter n to denote the number of edges in a graph, which is always denoted by m. Accordingly, we have switched to using m in this section.)

In any iteration of the algorithm, we must solve the minimization problem $\arg\min\{(x_t^\intercal Ay)/(\mathbf{1}^\intercal y) \mid y \in \mathbf{\Delta}(\text{paths})\}$, where $\mathbf{\Delta}(\text{paths})$ denotes the set of all probability distributions over paths that join some (s_i, t_i) pair. Recalling that the minimum is always achieved at a distribution y that assigns probability 1 to one path and 0 to all others, and that the vector Ay in this case is a $\{0,1\}$-vector that identifies the edges of the path, we see that the expression $x_t^\intercal Ay$ can be interpreted as the combined cost of the edges in path y, when edge costs are given by the entries of the vector x_t. The expression $\mathbf{1}^\intercal y$ is simply equal to 1, so it can be ignored. Thus, the minimization problem that we must solve in one iteration of the algorithm is to find a minimum-cost path with respect to the edge costs given by x_t. This is easily done by running Dijkstra's algorithm to find the minimum cost (s_i, t_i) path for each $i = 1, \dots, k$.

In summary, we have derived the following algorithm for approximately solving the maximum-throughput multicommodity flow problem in graphs whose edges all have identical capacity B. The algorithm reduces computing a $(1 - 2\varepsilon)$-approximate maximum multicommodity flow to solving $km \ln m/\varepsilon^2$ shortest-path problems. In the pseudocode, the variable z_e for an edge $e = e_i$ keeps track of the amount of flow we have sent on edge e, and $x_e = (1+\varepsilon)^{z_e/\delta}$ is a variable whose value in loop iteration t is proportional to (but not equal to) the i^{th} entry of the vector x_t in the above discussion. The algorithm's validity is unaffected by the fact that the vector $(x_e)_{e \in E}$ is a scalar multiple of the vector x_t in the above discussion, because the outcome of the min-cost path computation with respect to edge cost vector $\mathbf{x}$ is unaffected by rescaling the costs.

Algorithm 2 Max-throughput multicommodity flow algorithm, uniform-capacity case.

```
1: Given: Parameter $\varepsilon > 0$.
2: Initialize: $\delta = \varepsilon^2 B/(\ln m)$, $x = \mathbf{1}$, $f_1 = \cdots = f_k = 0$, $z = 0$.
3: while $z \prec B\mathbf{1}$ do
4:     for $i = 1, \dots, k$ do
5:         $P_i \leftarrow$ minimum cost path from $s_i$ to $t_i$, with respect to edge costs $x_e$.
6:     end for
7:     $i \leftarrow \arg\min_{1 \le j \le k}\{\text{cost}(P_j)\}$.
8:     Update flow $f_i$ by sending $\delta$ units of flow on $P_j$.
9:     for all $e \in P_j$ do
10:        $x_e \leftarrow (1+\varepsilon)x_e$.
11:        $z_e \leftarrow z_e + \delta$.
12:    end for
13: end while
```

Note that in this example, the fact that the packing linear program has exponentially many variables did not prevent us from designing an efficient algorithm to solve it. That is because, although the matrix A and vector Y in the multiplicative-weights algorithm have exponentially many entries, the algorithm never explicitly stores and manipulates them. This theme is quite common in applications of the multiplicative-weights method: the space requirement of the algorithm scales linearly with the number of *constraints* in the primal LP, but we can handle exponentially many variables in polynomial space and time, provided that we have a subroutine that efficiently solves the minimization problem $\arg\min\{(x_t^\intercal Ay)/(p^\intercal y)\}$.

3.3 General edge capacities

When edges have differing capacities, a small modification to the foregoing algorithm permits us to use it for computing an approximate maximum-throughput multicommodity flow.

The issue is that the multiplicative-weights algorithm we have presented in these notes requires a packing LP in which all of the constraints have the same number, B, appearing on their right-hand side. As a first step in dealing with this, we can rescale both sides of each constraint:

$$\sum_{P:e\in P} y_P \le c(e) \quad \iff \sum_{P:e\in P} \frac{1}{c(e)} y_P \le 1.$$

The trouble with this rescaling is that now the constraint matrix entry a_{ij} is equal to $1/c(e_i)$ if edge e_i belongs to path P_j. Our algorithm requires $0 \le a_{ij} \le 1$ and this could be violated if some edges have capacity less than 1.

The simplest way to deal with this issue is to preprocess the graph, scaling all edge capacities by $1/c_{\min}$ where $c_{\min}$ denotes the minimum edge capacity, to obtain a graph whose edge capacities are bounded below by 1. Then we can solve for an approximate max-flow in the rescaled graph, and finally scale that flow down by $c_{\min}$ to obtain a flow that is feasible — and still approximately throughput-maximizing — in the original graph. To bound the number of iterations that this algorithm requires, we must determine the value of γ for the rescaled graph. The rescaled capacity of edge e is $c(e)/c_{\min}$, so the matrix entry a_{ij} is $c_{\min}/c(e_i)$ if edge e_i belongs to path P_j. Thus, the maximum entry in column j of the constraint matrix is $c_{\min}/c_{\min}(P_j)$, where $c_{\min}(P_j)$ denotes the minimum edge capacity in P_j. Thus $\gamma = \min_j\{c_{\min}/c_{\min}(P_j)\}$ and the number of iterations is

$$\frac{m \ln m}{\gamma \varepsilon^2} = \frac{m \ln m}{\varepsilon^2} \cdot \max_j \left\{ \frac{c_{\min}(P_j)}{c_{\min}} \right\}.$$

This could be a very large number of iterations, if the graph contains some very "fat" paths whose minimum-capacity edge has much more capacity than the globally minimum edge capacity.

Rather than rescaling all of the edge capacities in the graph by the same common factor, a smarter solution is to rescale the *flow* on path P_j by the factor $c_{\min}(P_j)$. More precisely, define the "P-saturating flow " to be the flow that sends $c_{\min}(P)$ units on every edge of P, and zero on all other edges. Our LP will have variables y_P for every path P that joins s_i to t_i for some $i = 1, \dots, k$, and a primal-feasible solution will correspond to a multicommodity flow that is a weighted sum of P-saturating flows, scaled by the values y_P.

This leads to the following linear programming formulation of maximum-throughput multicommodity flow.

$$\begin{array}{ll} \max & \sum_P c_{\min}(P) y_P \\ \text{s.t.} & \sum_{P:e\in P} \frac{c_{\min}(P)}{c(e)} y_P \le 1 \quad \forall e \\ & y_P \ge 0 \quad \forall P \end{array} \tag{10}$$

The constraint matrix has entries $a_{ij} = \frac{c_{\min}(P_j)}{c(e_i)}$ if e_i belongs to P_j. By the definition of $c_{\min}(P_j)$, this implies that all entries are between 0 and 1, and that every column of A has at least one entry equal to 1. Thus the multiplicative weights method, applied to this LP formulation, yields a $(1-2\varepsilon)$-approximate solution after at most $\frac{m \ln m}{\varepsilon^2}$ iterations. To conclude the discussion of this algorithm, we should specify a procedure for solving the minimization problem $\arg\min\{(x_t^{\mathsf{T}} Ay)/(p^{\mathsf{T}} y)\}$ in every iteration of the while loop. If y is the indicator vector for a path P, then $p^{\mathsf{T}} y = c_{\min}(P)$ while Ay is the vector whose ith entry is $\frac{c_{\min}(P)}{c(e_i)}$ if e_i belongs to P, and 0 otherwise. Thus,

$$x_t^\intercal Ay = \sum_{e \in P} \frac{c_{\min}(P)x_{ti}}{c(e_i)}$$
$$\frac{x_t^\intercal Ay}{p^\intercal y} = \sum_{e \in P} \frac{x_{ti}}{c(e_i)}$$

so the minimization problem that must be solved in each loop iteration is merely finding a minimum-cost path with respect to the edge costs $\mathrm{cost}(e_i) = \frac{x_{ti}}{c(e_i)}$.

Summarizing this discussion, we have the following algorithm which finds a $(1-2\varepsilon)$-approximate maximum-throughput multicommodity flow in general graphs using $\frac{m \ln m}{\varepsilon^2}$ loop iterations, each of which requires k minimum-cost path computations. In the pseudocode, the variable z_e for an edge $e = e_i$ keeps track of the *fraction* of e's capacity that has already been consumed by the flow sent in previous loop iterations. The variable $x_e = (1+\varepsilon)^{z_e/\delta}$ is a variable whose value in loop iteration t is proportional to (but not equal to) the i^{th} entry of the vector x_t in the above discussion. As in the preceding section, the algorithm's validity is unaffected by the fact that the vector $(x_e)_{e \in E}$ is a scalar multiple of the vector x_t in the above discussion, because the outcome of the min-cost path computation with respect to edge cost vector $\mathbf{x}$ is unaffected by rescaling the costs.

Algorithm 3 Max-throughput multicommodity flow algorithm, general case.

```
 1: Given: Parameter $\varepsilon > 0$.
 2: Initialize: $\delta = \varepsilon^2/(\ln m)$, $x = \mathbf{1}$, $f_1 = \cdots = f_k = 0$, $z = 0$.
 3: while $z \prec \mathbf{1}$ do
 4:     for $i = 1, \ldots, k$ do
 5:         $P_i \leftarrow$ minimum cost path from $s_i$ to $t_i$, with respect to edge costs $x_e/c(e)$.
 6:     end for
 7:     $i \leftarrow \arg\min_{1 \le j \le k}\{\mathrm{cost}(P_j)\}$.
 8:     Update flow $f_i$ by sending $\delta c_{\min(P_j)}$ units of flow on $P_j$.
 9:     for all $e \in P_j$ do
10:         $r \leftarrow \frac{c_{\min}(P_j)}{c(e)}$.
11:         $x_e \leftarrow (1+\varepsilon)^r x_e$.
12:         $z_e \leftarrow z_e + \delta\, r$.
13:     end for
14: end while
```

3.4 Maximum concurrent flow

The maximum concurrent flow problem can be solved using almost exactly the same technique. While the packing formulation of maximum-throughput multicommodity flow involves packing individual paths, each of which connects one source-sink pair, the natural packing formulation of maximum concurrent multicommodity flow involves packing k-tuples of paths, one for each source-sink pair. In the following LP, Q is an index that ranges over all such k-tuples. (As before, this means that there are exponentially many variables y_Q. Likewise, as before, this will not inhibit our ability to design an efficient algorithm for approximately solving the LP, because the algorithm need not explicitly represent all of the entries of the constraint matrix A or the vector Y.) The notation $n_Q(e)$ refers to the number of paths in the k-tuple Q that contain edge e;

thus, its value is always an integer between 0 and k. The notation $c_{\min}(Q)$ refers to the minimum capacity of an edge e such that $n_Q(e) > 0$.

$$\begin{array}{ll} \max & \sum_Q c_{\min}(Q) y_Q \\ \text{s.t.} & \sum_{Q: n_Q(e)>0} \frac{n_Q(e) c_{\min}(Q)}{k \cdot c(e)} y_Q \le 1 \quad \forall e \\ & y_Q \ge 0 \quad \forall Q \end{array} \tag{11}$$

For a path-tuple Q, the "Q-saturating flow" is a multicommodity flow that sends $c_{\min}(Q)/k$ units of flow on each of the k paths in Q. (The scaling by $1/k$ is necessary, to ensure that the Q-saturating flow doesn't exceed the capacity of any edge, even if the minimum-capacity edge of Q belongs to all k of the paths in Q.) A primal-feasible vector for the linear program 11 can be interpreted as a weighted sum of Q-saturating flows, weighted by y_Q. The coefficients in the capacity constraint for each e are justified by the observation that a Q-saturating flow sends a total of $n_Q(e) c_{\min}(Q)/k$ units of flow on edge e.

The value of γ for this linear program is $1/k$, so after at most $km \ln(m)/\varepsilon^2$ we obtain a $(1-2\varepsilon)$-approximation to the maximum concurrent multicommodity flow. The minimization problem $\arg\min\{(x_t^\intercal Ay)/(p^\intercal y)\}$ has the following interpretation: when y is the indicator vector of a path-tuple Q, then $p^\intercal y = c_{\min}(Q)$, while Ay is the vector whose i^{th} component is $\frac{n_Q(e_i)\, c_{\min}(Q)}{k\, c(e_i)}$. Thus, letting $Q_1, \ldots, Q_k$ denote the k paths that make up Q, we have

$$x_t^\intercal A y = \sum_i \frac{x_{t,i}\, n_Q(e_i)\, c_{\min}(Q)}{k\, c(e_i)} = \frac{c_{\min}(Q)}{k} \sum_i \frac{x_{t,i}}{c(e_i)} \cdot n_Q(e_i) = \frac{c_{\min}(Q)}{k} \sum_{j=1}^{k} \sum_{e_i \in Q_j} \frac{x_{t,i}}{c(e_i)}$$

$$\frac{x_t^\intercal A y}{p^\intercal y} = \frac{1}{k} \sum_{j=1}^{k} \sum_{e_i \in Q_j} \frac{x_{t,i}}{c(e_i)}.$$

Hence, the ratio $\frac{x_t^\intercal A y}{p^\intercal y}$ is minimized by choosing Q to be the k-tuple consisting of the minimum-cost path from s_j to t_j, for each $j = 1, \ldots, k$, with respect to the edge costs $\frac{x_{t,i}}{c(e_i)}$.

Algorithm 4 Maximum concurrent multicommodity flow algorithm

```
1:  Given: Parameter $\varepsilon > 0$.
2:  Initialize: $\delta = \varepsilon^2/(\ln m)$, $x = \mathbf{1}$, $f_1 = \cdots = f_k = 0$, $z = 0$.
3:  while $z \prec \mathbf{1}$ do
4:      for $i = 1, \ldots, k$ do
5:          $Q_i \leftarrow$ minimum cost path from $s_i$ to $t_i$, with respect to edge costs $x_e/c(e)$.
6:          Update flow $f_i$ by sending $\delta\, c_{\min(P_j)}/k$ units of flow on $Q_i$.
7:      end for
8:      for all edges $e$ do
9:          $n_Q(e) \leftarrow$ the number of $i$ such that $e \in Q_i$.
10:         $r \leftarrow \frac{c_{\min}(Q)\, n_Q(e)}{k\, c(e)}$.
11:         $x_e \leftarrow (1+\varepsilon)^r x_e$.
12:         $z_e \leftarrow z_e + \delta\, r$.
13:     end for
14: end while
```

4 The sparsest cut problem

Given the importance of the max-flow min-cut theorem in discrete mathematics and optimization, it is natural to wonder if there is an analogue of this theorem for multicommodity flows.

If one adopts the interpretation that "a minimum cut is an edge set whose capacity certifies an upper bound on the maximum flow," then the next question is: what upper bounds on throughput or concurrent multicommodity can be certified by an edge set?

Definition 1. Let G be a graph with edge capacities $c(e) \geq 0$ and source-sink pairs $\{(s_i, t_i)\}_{i=1}^k$. An edge set A is said to *separate* a source-sink pair (s_i, t_i) if every path from s_i to t_i contains an edge of A. A *cut* is an edge set that separates at least one source-sink pair. A *multicut* is an edge set that separates every source-sink pair. The *sparsity* of a cut is its capacity divided by the number of source-sink pairs it separates.

If G contains a multicut A of capacity c, then the throughput of any multicommodity flow cannot exceed c, since each unit of flow must consume at least one unit of capacity on one of the edges in A. A similar argument shows that if G contains a cut A with sparsity c, then the maximum concurrent flow rate cannot exceed c.

Unlike in the case of single-commodity flows, it is *not* the case that the maximum throughput is equal to the minimum capacity of a multicut, nor is it the case that the maximum concurrent flow rate is equal to the sparsest cut value. In both cases, the relevant cut-defined quantity may exceed the flow-defined quantity, by only by a factor of $O(\log k)$ in undirected graphs. This bound is known to be tight up to constant factors. In directed graphs the situation is worse: the minimum multicut may exceed the maximum throughput by $\Theta(k)$ and this is again tight in terms of k, but the way this gap depends on n (the number of vertices) in the worst case remains an open question.

In this section we will present a randomized algorithm to construct a cut whose (expected) sparsity is within a $O(\log k)$ factor of the maximum concurrent flow rate, in undirected graphs. Thus, we will be giving an algorithmic proof of the $O(\log k)$-approximate max-flow min-cut theorem for concurrent multicommodity flows.

4.1 Fractional cuts

To start designing the algorithm, let us recall the sparsest cut LP and its dual. (In the following linear programs, the index Q ranges over k-tuples of paths joining each source to its sink.)

$$\begin{array}{lll} \max & \sum_Q y_Q & \\ \text{s.t.} & \forall e & \sum_Q n_Q(e) y_Q \leq c(e) \\ & \forall Q & y_Q \geq 0 \end{array} \qquad \begin{array}{lll} \min & \sum_e c(e) x_e & \\ \text{s.t.} & \forall Q & \sum_e n_Q(e) x_e \geq 1 \\ & \forall e & x_e \geq 0 \end{array}$$

If one interprets $x = (x_e)_{e \in E}$ as a vector of edge lengths, then the expression $\sum_e n_Q(e) x_e$ in the dual LP represents the sum of the lengths of all paths in Q. Thus, a feasible solution of the dual LP is an assignment of a length to each edge of G, such that the sum of shortest-path lengths between all source-sink pairs is at least 1. For example, if there is a cut A whose sparsity is C/p because it separates p source-sink pairs and has capacity C, then we obtain a dual-feasible vector by setting $x_e = 1/p$ if e belongs to A and $x_e = 0$ otherwise. For each of the p source-sink pairs separated by A, their distance in the graph with edge lengths defined by x is at least

$1/p$, and therefore the combined distance of all source-sink pairs is at least 1 as required by the dual feasibility condition. For this particular dual-feasible vector x, the dual objective function is $\sum_{e \in A} c(e)/p = C/p$, which matches the sparsity of A. Thus, we have confirmed that the optimum of the dual LP is a lower bound on the sparsest cut value. (Which we knew anyhow, because the optimum of the dual LP coincides with the maximum concurrent multicommodity flow rate, and we already knew that was a lower bound on the sparsest cut value.)

Owing to these considerations, a dual-feasible vector x is often called a *fractional cut* and $\sum_e c(e)x_e$ is called the sparsity of the fractional cut. Our randomized algorithm for the sparsest cut problem starts by computing an optimal (or approximately optimal) solution x to the dual LP — for example, using the multiplicative weights algorithm developed in the preceding section — and then "rounding" x to produce a cut whose sparsity exceeds the sparsity of x by a factor of at most $O(\log k)$, in expectation.

4.2 Dependent rounding

One natural idea for transforming a fractional cut into a genuine cut is to sample a random edge set by selecting each edge e independently with probability x_e. This turns out to be a terrible idea. For example, consider that case that $k = 1$ (a single-commodity flow problem) and G is the complete bipartite graph $K_{2,n}$; the two nodes on the left side of the bipartition are the source and sink, s and t. In this graph there is a fractional cut defined by setting $x_e = 1/2$ for every edge e. However, if we construct a random edge by sampling every edge independently with probability $1/2$, the probability of separating s from t is exponentially small in n.

Rather than independent randomized rounding, a better plan is to do some sort of *dependent rounding*. Once again, the case of single-commodity flows is a fertile source of intuition. Suppose x is a fractional cut for a single-commodity flow problem with source s and sink t. Using x, we will construct a random cut based on a sort of "breadth-first search" starting from s. For every vertex u let $d(s, u)$ denote the length of the shortest path from s to u when edge lengths are defined by x. Choose a uniformly random number $r \in [0, 1]$, and cut all edges (u, v) such that $d(s, u) \le r < d(s, v)$. This random cut always separates s from t: on every path from s to t there is an earliest vertex whose distance from s exceeds r, and the edge leading into this vertex belongs to the cut. The expected capacity of the cut can be computed by linearity of expectation: for any edge $e = (u, v)$, the probability that the random cut contains e is $|d(s, u) - d(s, v)|$, which is bounded above by x_e. Hence the expected capacity of the random cut is bounded above by $\sum_e c(e)x_e$. We have thus shown that in the special case of single-commodity flow, for any fractional cut of capacity C, there is a simple randomized algorithm to compute a cut whose expected capacity is at most C.

The randomized sparsest cut algorithm that we will develop uses a similar dependent rounding scheme based on breadth-first search, but this time starting from a set of sources rather than just one source. The precise sampling procedure looks a little bit strange at first sight. Here it is:

1. Sample t uniformly at random from the set $\{0, 1, \dots, \lfloor \log(2k) \rfloor\}$.
2. Sample a random set W by selecting each element of the set $\{s_1, t_1, s_2, t_2, \dots, s_k, t_k\}$ independently with probability 2^{-t}.
3. Sample a uniformly random r in $[0, 1]$.

4. Cut all edges (u, v) such that $d(u, W) < r < d(v, W)$, where the expression $d(u, W)$ refers to the minimum of $d(u, w)$ over all $w \in W$.

Why does this work? We have to estimate two things: the expected capacity of the cut, and the expected number of source-sink pairs that it separates.

Expected capacity. Estimating the expected capacity is surprisingly easy. It closely parallels the argument in the single-commodity case. For an edge $e = (u, v)$, no matter what set W is chosen, we have

$$\Pr(d(u, W) < r < d(v, W)) = |d(u, W) - d(v, W)| \leq x_e$$

so the expected combined capacity of the edges in the cut, by linearity of expectation, is at most $\sum_e c(e)x_e$, the value of the fractional cut x. Recall that this is equal to the maximum concurrent flow rate, if x is an optimal solution to the dual of the maximum concurrent flow LP.

Expected number of separated pairs. For a source-sink pair (s_i, t_i), the probability that the cut separates s_i from t_i is

$$\int_0^1 [\Pr(d(s_i, W) < r < d(t_i, W)) + \Pr(d(t_i, W) < r < d(s_i, W))] \, dr$$

To prove a lower bound on this integral, we will show that for $0 < r < \frac{1}{2}d(s_i, t_i)$, the integrand is bounded below by $\Omega(1/\log(2k))$. This will imply that the integral is bounded below by $\Omega(1/\log(2k))d(s_i, t_i)$. Recall that dual-feasibility of x implies that $\sum_{i=1}^k d(s_i, t_i) = 1$. Thus, the expected number of source-sink pairs separated by our random cut is $\Omega(1/\log(2k))$.

For $0 < r < \frac{1}{2}d(s_i, t_i)$ let S and T denote the subsets of $\{s_1, t_1, \ldots, s_k, t_k\}$ consisting of all terminals within distance r of s_i and t_i, respectively. Note that S and T are non-empty (they contain s_i and t_i, respectively) and they are disjoint, because $r < \frac{1}{2}d(s_i, t_i)$. The event that $d(s_i, W) < r < d(t_i, W)$ is the same as the event that $S \cap W$ is nonempty but $T \cap W$ is not, and similarly for the event $d(t_i, W) < r < d(s_i, W)$. Hence the integrand $\Pr(d(s_i, W) < r < d(t_i, W)) + \Pr(d(t_i, W) < r < d(s_i, W))$ is equal to the probability that precisely one of the sets $S \cap W, T \cap W$ is non-empty. Note that whenever $|(S \cup T) \cap W| = 1$, it is always the case that precisely one of the sets $S \cap W, T \cap W$ is non-empty. Let $h = |S \cup T|$. There is a unique $t \in \{0, 1, \ldots, \lfloor \log(2k) \rfloor\}$ such that $2^t < h \leq 2^{t+1}$. Assuming this value of t is sampled in the first step of our sampling algorithm, the probability that W contains exactly one element of $S \cup T$ is precisely

$$h \cdot 2^{-t} \cdot (1 - 2^{-t})^{h-1} = \frac{h}{2^t} \cdot \left(1 + \frac{1}{2^t - 1}\right)^{-(h-1)} > e^{-(h-1)/(2^t-1)} \geq e^{-3}.$$

So the integrand is bounded below by $e^{-3} \cdot \frac{1}{\log(2k)}$ when $0 < r < \frac{1}{2}d(s_i, t_i)$, which completes the proof.

4.3 Rejection sampling

You may notice that we promised a sampling algorithm that produces a random cut whose expected sparsity is $O(\log k)$ times the maximum concurrent flow rate $\sum_e c(e)x_e$. Instead we have given a sampling algorithm that produces a random cut A such that

$$\frac{\mathbb{E}[\text{cap}(A)]}{\mathbb{E}[\text{sep}(A)]} \leq e^3 \log(2k) \sum_e c(e)x_e. \tag{12}$$

which is not quite the same thing. (Here, cap(A) denotes the capacity of A and sep(A) denotes the number of source-sink pairs that it separates.) To fix this problem, we rewrite (12) as follows, using the formula $\text{cap}(A) = \text{sep}(A) \cdot \text{sparsity}(A)$ along with the definition of the expected value of a random variable:

$$e^3 \log(2k) \sum_e c(e) x_e \geq \frac{\sum_A \Pr(A) \ \text{cap}(A)}{\sum_A \Pr(A) \ \text{sep}(A)} = \frac{\sum_A \Pr(A) \ \text{sep}(A) \cdot \text{sparsity}(A)}{\sum_A \Pr(A) \ \text{sep}(A)} .$$

So, if we adjust our sampling rule so that the probability of sampling a given cut A is scaled up by sep(A) (and then renormalized so that probabilities sum up to 1) we get a random cut whose expected sparsity is at most $e^3 \log(2k) \sum_e c(e) x_e$, as desired. One way to adjust the probabilities in this way is to use *rejection sampling*, which leads to the following algorithm.

Algorithm 5 Rounding a fractional cut to a sparse cut.

```
1: Given: fractional cut $x$ defining shortest-path distances $d(\cdot,\cdot)$.
2: repeat
3:     Sample $t$ uniformly at random from the set $\{0, 1, \dots, \lfloor \log(2k) \rfloor\}$.
4:     Sample a random set $W$ by selecting each element of the set $\{s_1, t_1, s_2, t_2, \dots, s_k, t_k\}$
       independently with probability $2^{-t}$.
5:     Sample a uniformly random $r$ in $[0, 1]$.
6:     $A = \{(u, v) \mid d(u, W) < r < d(v, W)\}$.
7:     Sample a uniformly random $j \in \{1, \dots, k\}$.
8: until $j \leq \text{sep}(A)$
```

Why does this work? Let $\Pr(A)$ denote the probability of sampling A under the previous algorithm. Imagine that we modified the algorithm to run a single iteration of the **repeat** loop and either output A if it passes the test $j \leq \text{sep}(A)$ at the end of the loop, or else the algorithm simply fails and outputs nothing. For any cut A, the probability that this modified algorithm outputs A would be $\Pr(A) \cdot \frac{\text{sep}(A)}{k}$. In other words, conditional on succeeding, the modified algorithm samples a cut from exactly the rescaled distribution that we wanted to sample from. By repeating the loop until it succeeds, we guarantee that the algorithm draws one sample from this conditional distribution.

5 Application: zero-sum games

Definition 2. A *two-player zero-sum game* is one in which $\mathcal{I} = \{1, 2\}$ and $u_2(a_1, a_2) = -u_1(a_1, a_2)$ for all pure strategy profiles (a_1, a_2).

A famous theorem of von Neumann illustrates that the equilibria of two-player zero-sum games are much simpler than the equilibria of general two-player games.

Theorem 1 (von Neumann's minimax theorem)**.** *For every two-player zero-sum game with finite strategy sets A_1, A_2, there is a number $v \in \mathbb{R}$, called the* game value*, such that:*

1.

$$v = \max_{p \in \boldsymbol{\Delta}(A_1)} \min_{q \in \boldsymbol{\Delta}(A_2)} u_1(p, q) = \min_{q \in \boldsymbol{\Delta}(A_2)} \max_{p \in \boldsymbol{\Delta}(A_1)} u_1(p, q)$$

2. *The set of mixed Nash equilibria is nonempty. A mixed strategy profile (p, q) is a Nash equilibrium if and only if*

$$\begin{aligned} p &\in \arg\max_p \min_q u_1(p, q) \\ q &\in \arg\min_q \max_p u_1(p, q) \end{aligned}$$

3. *For all mixed Nash equilibria (p, q), $u_1(p, q) = v$.*

5.1 The main lemma

The hardest step in the proof of von Neumann's minimax theorem is to prove that

$$\max_p \min_q u_1(p, q) \geq \min_q \max_p u_1(p, q).$$

We will prove this fact using online learning algorithms. The basic idea of the proof is that if the players are allowed to play the game repeatedly, using **Hedge** to adapt to the other player's moves, then low-regret property of **Hedge** guarantees that the time-average of each player's mixed strategy is nearly a best response to the time-average of the other player's mixed strategy.

Lemma 2. *For any two-player zero-sum game,*

$$\max_{p \in \boldsymbol{\Delta}(A_1)} \min_{q \in \boldsymbol{\Delta}(A_2)} u_1(p, q) \geq \min_{q \in \boldsymbol{\Delta}(A_2)} \max_{p \in \boldsymbol{\Delta}(A_1)} u_1(p, q).$$

Proof. Assume without loss of generality that $0 \leq u_1(a_1, a_2) \leq 1$ for all strategy profiles (a_1, a_2). Let $\delta > 0$ be an arbitrarily small positive number, and define n, T, ε as above. Player 1 still uses **Hedge**(ε) to define a sequence of mixed strategies $p_1, p_2, \ldots, p_T$ in response to the payoff function induced by the opponent's sequence of strategies. But player 2 now chooses its strategies adversarially, according to the prescription

$$q_t \in \arg\min_q u_1(p_t, q). \tag{13}$$

Note that the set of mixed strategies minimizing player 1's payoff always contains a pure strategy, so we may assume q_t is a pure strategy if desired.

Define $\overline{p}, \overline{q}$ as above. We find that

$$\begin{aligned} \max_p \min_q u_1(p, q) &\geq \min_q u_1(\overline{p}, q) \\ &= \min_q \frac{1}{T} \sum_{t=1}^{T} u_1(p_t, q) \\ &\geq \frac{1}{T} \sum_{t=1}^{T} \min_q u_1(p_t, q) \\ &= \frac{1}{T} \sum_{t=1}^{T} u_1(p_t, q_t) \\ &\geq \max_p \frac{1}{T} \sum_{t=1}^{T} u_1(p, q_t) - \delta \\ &= \max_p u_1(p, \overline{q}) - \delta \\ &\geq \min_q \max_p u_1(p, q) - \delta. \end{aligned}$$

$\square$

5.2 Proof of Theorem 1

In this section we complete the proof of von Neumann's minimax theorem.

Proof of Theorem 1. For any mixed strategy profile $(\hat{p}, \hat{q})$ we have

$$u_1(\hat{p}, \hat{q}) \le \max_p u_1(p, \hat{q}).$$

Taking the minimum of both sides as $\hat{q}$ ranges over $\mathbf{\Delta}(A_2)$ we find that

$$\min_q u_1(\hat{p}, q) \le \min_q \max_p u_1(p, q).$$

Taking the maximum of both sides as $\hat{p}$ ranges over $\mathbf{\Delta}(A_1)$ we find that

$$\max_p \min_q u_1(p, q) \le \min_q \max_p u_1(p, q).$$

The reverse inequality was proven in Lemma 2. Thus we have established part (1) of Theorem 1.

Note that the sets $B_1 = \arg\max_p \min_q u_1(p, q)$ and $B_2 = \arg\min_q \max_p u_1(p, q)$ are both nonempty. (This follows from the compactness of $\mathbf{\Delta}(A_1)$ and $\mathbf{\Delta}(A_2)$, the contintuity of u_1, and the finiteness of A_1 and A_2.) If $p \in B_1$ and $q \in B_2$ then

$$v = \min_q u_1(p, q) \le u_1(p, q) \le \max_p u_1(p, q) = v$$

hence $u_1(p, q) = v$. Moreover, since $q \in B_2$, player 1 can't achieve a payoff greater than v against q by changing its mixed strategy. Similarly, since $p \in B_1$, player 2 can't force player 1's payoff to be less than v by changing its own mixed strategy. Hence (p, q) is a Nash equilibrium. Conversely, if (p, q) is a Nash equilibrium, then

$$u_1(p, q) = \max_p u_1(p, q) \ge v \tag{14}$$

$$u_1(p, q) = \min_q u_1(p, q) \le v \tag{15}$$

and this implies that in each of (14), (15), the inequality on the right side is actually an *equality*, which in turn implies that $p \in B_1$ and $q \in B_2$. This completes the proof of (2) and (3). $\square$

The proof of the minimax theorem given here, using online learning, differs from the standard proof which uses ideas from the theory of linear programming. The learning-theoretic proof has a few advantages, some of which are spelled out in the following remarks.

Remark 1. The procedure of using **Hedge** to approximately solve a zero-sum game is remarkably fast: it converges to within δ of the optimum using only $O(\log(n)/\delta^2)$ steps, provided the payoffs are between 0 and 1. (By "converges to within δ", we mean that it outputs a pair of mixed strategies, $(\overline{p}, \overline{q})$ such that $\min_q u_1(\overline{p}, q) \ge v - \delta$ and $\max_p u_1(p, \overline{q}) \le v + \delta$, where v is the game value.) This is especially important when one of the players has a strategy set whose size is exponentially larger than the size of the natural representation of the game. See Example 1 below for an example of this.

Recall that in the second proof of Lemma 2 we remarked that player 2's strategies q_t could be taken to be pure strategies. Note also that the **Hedge** algorithm used by player 1 only needs to look at the scores in a column of the payoff matrix if the corresponding strategy has been used by player 2 some time in the past. Thus, as long as we have an oracle for finding player 2's best response to any mixed strategy, we need only look at a very sparse subset of the payoff matrix — a set of $O(\log(n)/\delta^2)$ columns — to compute a mixed strategy for player 1 which obtains an additive δ-approximation to the game value. Again, see Example 1 for an example in which it is reasonable to assume that we don't have an explicit representation of the payoff matrix, but we can examine any desired column and we have an oracle for finding player 2's best response to any mixed strategy.

Example 1 (The VPN eavesdropping game)**.** Let $G = (V, E)$ be an undirected graph. In the "VPN eavesdropping game", player 1 chooses an edge of G, and player 2 chooses a spanning tree of G. For any edge e and spanning tree T, the payoff of player 1 is

$$u_1(e, T) = \begin{cases} 1 & \text{if } e \in T \\ 0 & \text{otherwise.} \end{cases}$$

(We can think of player 1 as an eavesdropper who can listen on any single edge of G, and player 2 as someone who is setting up a virtual private network on the edges of T, to join together all the nodes of G. The game is a win for player 1 if he or she eavesdrops on an edge which is part of the VPN.)

Note that, in general, the cardinality of player 2's strategy set is exponential in the size of G. Thus the parameter n appearing in the proof of Lemma 2 will be exponential in the size of the game's natural representation. However, it is easy to examine any particular column of the payoff matrix u_1: the column corresponding to a spanning tree T will be a vector of 0's and 1's, with 1's in the rows corresponding to the edges of T. Moreover, it is easy to compute player 2's best response to any mixed strategy of player 1: one simply computes a minimum spanning tree of G, where the weight of each edge is equal to the probability of player 1 picking that edge.

Consequently, there is an algorithm for approximately solving the game (up to an additive error of δ) which requires only $O(\log(M)/\delta^2)$ minimum spanning tree computations, where M is the total number of minimum spanning trees of G. (If G has V vertices, then by Cayley's formula $M \leq V^{V-2}$. Hence $\log(M)$ is always polynomial — in fact, nearly linear — in the number of vertices of G.)

Remark 2. Another consequence of the second proof of Lemma 2 is that player 2 has a mixed strategy which has *sparse support* — i.e. at most $O(\log(n)/\delta^2)$ strategies have positive probability — yet it achieves an additive δ-approximation to the game value. By symmetry, player 1 also has a mixed strategy with sparse support which achieves an additive δ-approximation to the game value. Hence the game has an approximate Nash equilibrium in which both players use sparsely-supported mixed strategies.

Remark 3. If player 2 is not playing rationally, by using **Hedge** player 1 comes close to achieving the best possible payoff against whatever distribution of strategies player 2 happens to be using. This property would not be ensured in repeated play if player 1 instead solved the game offline, picked a strategy in $\arg\max_p \min_q u_1(p, q)$, and always used this strategy.

Remark 4. If we think about our intuition of how human beings learn to play games against each other, the process is probably more similar to a learning algorithm such as **Hedge** than to a linear programming algorithm such as the simplex method. Hence another benefit of the learning-theoretic proof is that it gives an intuitive justification for why human beings are able to find the equilibria of zero-sum games.

5.3 Yao's lemma

The von Neumann minimax theorem has an important consequence in computer science. Suppose we have a computational problem with a finite set of possible inputs I, and we are considering a finite set of possible algorithms A. For example, I might be the set of all n-bit binary strings, and A might be the set of all Boolean circuits of size at most n^3 which accept an n-bit input and return a valid output for the problem under consideration. Suppose we have a parameter $t(i, a)$ which corresponds to the cost of running algorithm a on input i. For example, $t(i, a)$ could denote the algorithm's running time, or the cost of the solution it computes.

We may interpret this scenario as a two-player zero-sum game in which player 1 specifies an input, player 2 specifies an algorithm, and $t(i, a)$ is the payoff for player 1. Let $\mathcal{D} = \mathbf{\Delta}(I)$ denote the set of all probability distributions on inputs, and let $\mathcal{R} = \mathbf{\Delta}(A)$ denote the set of all probability distributions on algorithms, i.e. the set of all randomized algorithms. We can extend the function t to mixed strategy profiles in the usual way, i.e.

$$t(d, r) = \sum_{i \in I} \sum_{a \in A} t(i, a) d(i) r(a).$$

Lemma 3 (Yao's Lemma)**.**

$$\max_{d \in \mathcal{D}} \min_{a \in A} t(d, a) = \max_{d \in \mathcal{D}} \min_{r \in \mathcal{R}} t(d, r) = \min_{r \in \mathcal{R}} \max_{d \in \mathcal{D}} t(d, r) = \min_{r \in \mathcal{R}} \max_{i \in I} t(i, r).$$

Proof. The second equality is a restatement of von Neumann's minimax theorem. The first and third equalities follow from the fact that for any mixed strategy of one player, the other player always has a best response which is a pure strategy, i.e.

$$\begin{aligned} \forall d \in \mathcal{D} \ \min_{r \in \mathcal{R}} t(d, r) &= \min_{a \in A} t(d, a) \\ \forall r \in \mathcal{R} \ \max_{d \in \mathcal{D}} t(d, r) &= \max_{i \in I} t(i, r). \end{aligned}$$

□

6 Boosting

The multiplicative weights algorithm was applied to machine learning in a famous and influential paper by Freund and Schapire, that presented an algorithm called AdaBoost that shows how to train a very accurate classifier, given access to a subroutine called a "weak learner" that is capable of outputting classifiers that perform only slightly better than chance.

In this section we will present and analyze a simplified form of AdaBoost, in a setting where there are n training examples numbered $i = 1, \ldots, n$, and each has a *label* $y(i) \in \{\pm 1\}$. The weak learner is represented by an oracle WL whose input is a probability distribution $\pi \in \mathbf{\Delta}(n)$ and whose output is a *hypothesis* $h : [n] \to \{\pm 1\}$.

The error of hypothesis h on example i is defined to be

$$\mathbf{Err}_i(h) \triangleq \tfrac{1}{2} [1 - h(i)y(i)] = \begin{cases} 1 & \text{if } h(i) \neq y(i) \\ 0 & \text{otherwise.} \end{cases}$$

More generally, the *training error* of h with respect to a distribution π is

$$\mathbf{Err}_\pi(h) = \mathbb{E}_{i\sim\pi}[\mathbf{Err}_i(h)] = \frac{1}{2}\left(1 - \sum_{i=1}^{n} \pi(i)h(i)y(i)\right).$$

The training error $\mathbf{Err}_\pi(h)$ represents the probability that h makes an error when an example i is randomly sampled from π and classified according to h.

Let us assume that the weak learner satisfies the following guarantee for some $\varepsilon > 0$: for every distribution π, the hypothesis $WL(\pi)$ has training error less than $\frac{1}{2} - \varepsilon$. The boosting algorithm works as follows: it runs a sequence of T iterations, where T is an odd number greater than $2\ln(n)/(\varepsilon^2 - \varepsilon^3)$. Each iteration t constructs a distribution π_t using the multiplicative weights algorithm, where the "cumulative reward" $r_i(1:t-1)$ is interpreted as the number of times i was misclassified by the first $t-1$ hypotheses:

$$r_i(1:t-1) = \frac{1}{2}\left(1 - \sum_{s=1}^{t-1} h_s(i)y(i)\right)$$
$$\pi_t(i) = \frac{(1+\varepsilon)^{r_i(1:t-1)}}{\sum_{j=1}^{n}(1+\varepsilon)^{r_j(1:t-1)}}.$$

Iteration t ends by calling the weak learner to obtain a hypothesis $h_t = WL(\pi_t)$. After completing all T iterations, the boosting algorithm then classifies each example by taking a majority vote of the hypotheses $h_1, \ldots, h_T$:

$$h(i) = \textsc{maj}(h_1(i), \ldots, h_T(i)) = \operatorname{sgn}\left(\sum_{t=1}^{T} h_t(i)\right).$$

Theorem 4. *If each hypothesis h_t has training error less than $\frac{1}{2} - \varepsilon$ with respect to distribution π_t, i.e.*

$$\forall t \;\; \mathbf{Err}_{\pi_t}(h_t) < \tfrac{1}{2} - \varepsilon,$$

then the classifier h defined by the boosting algorithm classifies every training example correctly.

Proof. The performance guarantee for the multiplicative weights algorithm ensures that

$$\forall i \;\; (1-\varepsilon)r_i(1:T) - \frac{\ln(n)}{\varepsilon} < \sum_{t=1}^{T}\sum_{i=1}^{n} \pi_t(i)\left[\frac{1}{2} - \frac{1}{2}h_t(i)y_t(i)\right] = \sum_{t=1}^{T} \mathbf{Err}_{\pi_t}(h_t) < \left(\tfrac{1}{2} - \varepsilon\right)T.$$

Rearranging terms and dividing both sides by $1 - \epsilon$ we conclude that

$$\forall i \;\; r_i(1:T) < \left(\frac{\frac{1}{2} - \varepsilon}{1-\varepsilon}\right)T + \frac{\ln(n)}{\varepsilon(1-\varepsilon)} < \left(\frac{1}{2} - \frac{\varepsilon}{2}\right)T + \frac{\ln(n)}{\varepsilon(1-\varepsilon)} < \frac{T}{2} \tag{16}$$

since our choice of T ensures that $\frac{\ln(n)}{\varepsilon(1-\varepsilon)} < \frac{\varepsilon T}{2}$. Recalling that $r_i(1:T)$ is equal to the number of times i is misclassified by the hypotheses $h_1, \ldots, h_T$, we see that (16) implies that i is classified correctly by the majority of hypotheses in $h_1, \ldots, h_T$. □

Remark 5. Theorem 4 is really the start of the boosting story, not the end of the story. In applications of boosting, the goal is not to learn a hypothesis with zero training error but to learn a hypothesis with low *test error* or *generalization error*, meaning that it performs well on previously unseen examples, as long as those examples and their labels were sampled from the same distribution that produced the training set. Running the boosting algorithm until it attains zero error on the training set can potentially result in *overfitting*: learning a hypothesis that "memorizes the labels of the training examples" but doesn't generalize well when presented with unseen examples. One of the key objectives of research on boosting, therefore, was to understand the circumstances under which one could prove upper bounds on the generalization error of the procedure.

Remark 6. Often it is not necessary to learn a hypothesis that classifies every training example correctly, and it would suffice to learn one whose training error (under the uniform distribution) is less than δ. A modification of the proof of Theorem 4 shows that $O(\log(1/\delta)/\epsilon^2)$ training iterations suffice to achieve this objective. The key ingredient is a modified performance guarantee for **Hedge** that says that for any $r_* \geq 0$, if at least δn of the "experts" $i \in [n]$ have cumulative payoff greater than r_*, then the cumulative payoff of the **Hedge** algorithm is greater than $(1-\varepsilon)r_* - \frac{\ln(1/\delta)}{\varepsilon}$. The proof is the same, except that when bounding $\log_{1+\varepsilon} W(t)$ from below, rather than using $\frac{1}{n}(1+\varepsilon)^{r_i(1:t)}$ where i is the best-performing expert, we use the combined weight of the δn best-performing experts to improve this bound to $\frac{\delta n}{n}(1+\varepsilon)^{r_*}$.

A An inequality involving logarithms

Lemma 5. *For $x > 0$,*

$$\frac{1}{x}\ln(1+x) > 1 - x. \tag{17}$$

Proof. We have

$$\ln\left(\frac{1}{1+x}\right) = \ln\left(1 - \frac{x}{1+x}\right) < -\left(\frac{x}{1+x}\right).$$

Multiplying both sides by $-1/x$,

$$\frac{1}{x}\ln(1+x) > \frac{1}{1+x}.$$

Finally, the inequality $1 > (1-x)(1+x)$ implies that $\frac{1}{1+x} > 1 - x$, which concludes the proof of the lemma. $\square$