CS261: A Second Course in Algorithms Lecture #12: Applications of Multiplicative Weights to Games and Linear Programs[*]

Tim Roughgarden[†]

February 11, 2016

1 Extensions of the Multiplicative Weights Guarantee

Last lecture we introduced the multiplicative weights algorithm for online decision-making. You don't need to remember the algorithm details for this lecture, but you should remember that it's a simple and natural algorithm (just one simple update per action per time step). You should also remember its regret guarantee, which we proved last lecture and will use today several times as a black box.[1]

Theorem 1.1 *The expected regret of the multiplicative weights algorithm is always at most $2\sqrt{T \ln n}$, where n is the number of actions and T is the time horizon.*

Recall the definition of regret, where A denotes the action set:

$$\underbrace{\max_{a \in A} \sum_{t=1}^{T} r^t(a)}_{\text{best fixed action}} - \underbrace{\sum_{t=1}^{T} r^t(a^t)}_{\text{our algorithm}} .$$

The expectation in Theorem 1.1 is over the random choice of action in each time step; the reward vectors $\mathbf{r}^1, \ldots, \mathbf{r}^T$ are arbitrary.

The regret guarantee in Theorem 1.1 applies not only with with respect to the best fixed action in hindsight, but more generally to the best fixed probability distribution in

*©2016, Tim Roughgarden.

†Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: `tim@cs.stanford.edu`.

[1]This lecture is a detour from our current study of online algorithms. While the multiplicative weights algorithm works online, the applications we discuss today are not online problems.

hindsight. The reason is that, in hindsight, the best fixed action is as a good as the best fixed distribution over actions. Formally, for every distribution $\mathbf{p}$ over A,

$$\sum_{t=1}^{T}\sum_{a\in A} p_a \cdot r^t(a) = \sum_{a\in A} \underbrace{p_a}_{\text{sum to 1}} \underbrace{\left(\sum_{t=1}^{T} r^t(a)\right)}_{\le \max_b \sum_t r^t(b)} \le \max_{b\in A} \sum_{t=1}^{T} r^t(b).$$

We'll apply Theorem 1.1 in the following form (where time-averaged just means divided by T).

Corollary 1.2 *The expected time-averaged regret of the multiplicative weights algorithm is at most ϵ after at most $(4\ln n)/\epsilon^2$ time steps.*

As noted above, the guarantee of Corollary 1.2 applies with respect to any fixed distribution over the actions.

Another useful extension is to rewards that lie in $[-M, M]$, rather than in $[-1, 1]$. This scenario reduces to the previous one by scaling. To obtain time-averaged regret at most ϵ:

1. scale all rewards down by M;
2. run the multiplicative weights algorithm until the time-averaged expected regret is at most $\frac{\epsilon}{M}$;
3. scale everything back up.

Equivalently, rather than explicitly scaling the reward vectors, one can change the weight update rule from $w^{t+1}(a) = w^t(a)(1 + \eta r^t(a))$ to $w^{t+1}(a) = w^t(a)(1 + \frac{\eta}{M} r^t(a))$. In any case, Corollary 1.2 implies that after $T = \frac{4M^2 \ln n}{\epsilon^2}$ iterations, the time-averaged expected regret is at most ϵ.

2 Minimax Revisited (Again)

Last lecture we sketched how to use the multiplicative weights algorithm to prove the minimax theorem (details on Exercise Set #6). The idea was to have both the row and the column player play a zero-sum game repeatedly, using their own copies of the multiplicative weights algorithm to choose strategies simultaneously at each time step. We next discuss an alternative thought experiment, where the players move sequentially at each time step with only the row player using multiplicative weights (the column player just best responds). This alternative has similar consequences and translates more directly into interesting algorithmic applications.

Fix a zero-sum game $\mathbf{A}$ with payoffs in $[-M, M]$ and a value for a parameter $\epsilon > 0$. Let m denote the number of rows of $\mathbf{A}$. Consider the following thought experiment, in which the row player has to move first and the column player gets to move second:

Thought Experiment

- At each time step $t = 1, 2, \ldots, T = \frac{4M^2 \ln m}{\epsilon^2}$:
  - The row player chooses a mixed strategy $\mathbf{p}^t$ using the multiplicative weights algorithm (with the action set equal to the rows).
  - The column player responds optimally with the deterministic strategy $\mathbf{q}^t$.[2]
  - If the column player chooses column j, then set $r^t(i) = a_{ij}$ for every row i, and feed the reward vector $\mathbf{r}^t$ into the multiplicative weights algorithm. (This is just the payoff of each row in hindsight, given the column player's strategy at time t.)

We claim that the column player get at least its minimax payoff, and the row player gets at least its minimax payoff minus ϵ.

Claim 1: In the thought experiment above, the negative time-averaged expected payoff of the column player is at most

$$\max_{\mathbf{p}} \left(\min_{\mathbf{q}} \mathbf{p}^T \mathbf{A} \mathbf{q} \right).$$

Note that the benchmark used in this claim is the more advantageous one for the column player, where it gets to move second.[3]

Proof: The column player only does better than its minimax value because, not only does the player get to go second, but the player can tailor its best responses on each day to the row player's mixed strategy on that day. Formally, we let $\hat{\mathbf{p}} = \frac{1}{T}\sum_{t=1}^{T} \mathbf{p}^t$ denote the time-averaged row strategy and $\mathbf{q}^*$ an optimal response to $\hat{\mathbf{p}}$ and derive

$$\begin{aligned}
\max_{\mathbf{p}} \left(\min_{\mathbf{q}} \mathbf{p}^T \mathbf{A} \mathbf{q} \right) &\geq \min_{\mathbf{q}} \hat{\mathbf{p}}^T \mathbf{A} \mathbf{q} \\
&= \hat{\mathbf{p}}^T \mathbf{A} \mathbf{q}^* \\
&= \frac{1}{T} \sum_{t=1}^{T} (\mathbf{p}^t)^T \mathbf{A} \mathbf{q}^* \\
&\geq \frac{1}{T} \sum_{t=1}^{T} (\mathbf{p}^t)^T \mathbf{A} \mathbf{q}^t,
\end{aligned}$$

with the the last inequality following because $\mathbf{q}^t$ is an optimal response to $\mathbf{p}^t$ for each t. (Recall the column player wants to minimize $\mathbf{p}^T \mathbf{A} \mathbf{q}$.) Since the last term is the negative

[2]Recall from last lecture that the player who goes second has no need to randomize: choosing a column with the best expected payoff (given the row player's strategy $\mathbf{p}^t$) is the best thing to do.

[3]Of course, we've already proved the minimax theorem, which states that it doesn't matter who goes first. But here we want to reprove the minimax theorem, and hence don't want to assume it.

time-averaged payoff of the column player in the thought experiment, the proof is complete. ■

Claim 2: In the thought experiment above, the time-averaged expected payoff of the row player is at least

$$\min_{\mathbf{q}} \left(\max_{\mathbf{p}} \mathbf{p}^T \mathbf{A} \mathbf{q} \right) - \epsilon.$$

We are again using the stronger benchmark from the player's perspective, here with the row player going second.

Proof: Let $\hat{\mathbf{q}} = \frac{1}{T} \sum_{t=1}^{T} \mathbf{q}^t$ denote the time-averaged column strategy. The multiplicative weights guarantee, after being extended as in Section 1, states that the time-averaged expected payoff of the row player is within ϵ of what it could have attained using any fixed mixed strategy $\mathbf{p}$. That is,

$$\begin{aligned} \frac{1}{T} \sum_{t=1}^{T} (\mathbf{p}^t)^T \mathbf{A} \mathbf{q}^t &\geq \max_{\mathbf{p}} \left(\frac{1}{T} \sum_{t=1}^{T} \mathbf{p}^T \mathbf{A} \mathbf{q}^t \right) - \epsilon \\ &= \max_{\mathbf{p}} \mathbf{p}^T \mathbf{A} \hat{\mathbf{q}} - \epsilon \\ &\geq \min_{\mathbf{q}} \left(\max_{\mathbf{p}} \mathbf{p}^T \mathbf{A} \mathbf{q} \right) - \epsilon. \end{aligned}$$

■

Letting $\epsilon \to 0$, Claims 1 and 2 provide yet another proof of the minimax theorem. (Recalling the "easy direction" that $\max_{\mathbf{p}} \min_{\mathbf{q}} \mathbf{p}^T \mathbf{A} \mathbf{q} \leq \min_{\mathbf{q}} \max_{\mathbf{q}} \mathbf{p}^T \mathbf{A} \mathbf{q}$.) The next order of business is to translate this thought experiment into fast algorithms for approximately solving linear programs.

3 Linear Classifiers Revisited

3.1 Recap

Recall from Lecture #7 the problem of computing a linear classifier — geometrically, of separating a bunch of "+"s and "-"s with a hyperplane (Figure 1).

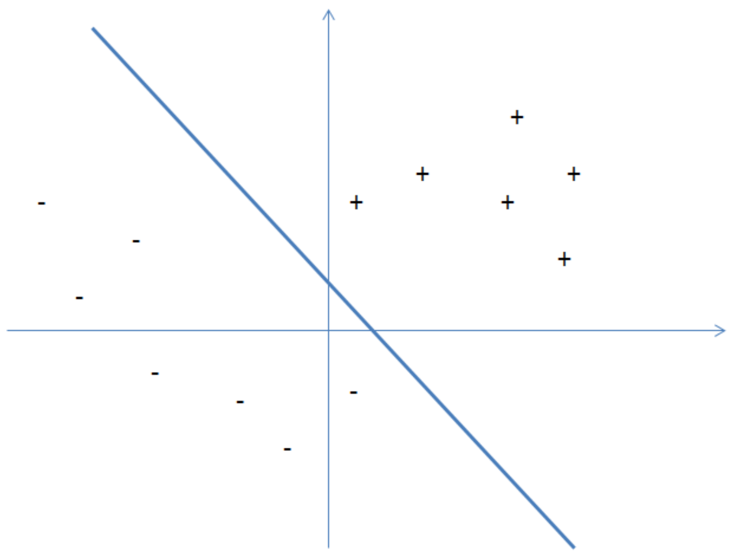

Figure 1: We want to find a linear function that separates the positive points (plus signs) from the negative points (minus signs)

Formally, the input consists of m "positive" data points $\mathbf{p}^1, \ldots, \mathbf{p}^m \in \mathbb{R}^d$ and m' "negative" data points $\mathbf{q}^1, \ldots, \mathbf{q}^{m'} \in \mathbb{R}^d$. This corresponds to labeled data, with the positive and negative points having labels +1 and -1, respectively.

The goal is to compute a linear function $h(\mathbf{z}) = \sum_{j=1}^{d} a_j z_j + b$ (from $\mathbb{R}^d$ to $\mathbb{R}$) such that

$$h(\mathbf{p}^i) > 0$$

for all positive points and

$$h(\mathbf{q}^i) < 0$$

for all negative points. In Lecture #7 we saw how to compute a linear classifier (if one exists) via linear programming. (It was almost immediate; the only trick was to introduce an additional variable to turn the strict inequality constraints into the usual weak inequality constraints.)

We've said in the past that linear programs with 100,000s of variables and constraints are usually no problem to solve, and sometimes millions of variables and constraints are also doable. But as you probably know from your other computer science courses, in many cases we're interested in considerably larger data sets. Can we compute a linear classifier faster, perhaps under some assumptions and/or allowing for some approximation? The multiplicative weights algorithm provides an affirmative answer.

3.2 Preprocessing

We first execute a few preprocessing steps to transform the problem into a more convenient form.

First, we can force the intercept b to be 0. The trick is to add an additional $(d+1)$th variable, with the new coefficient a_{d+1} corresponding to the old intercept b. Each positive and negative data point gets a new $(d+1)$th coordinate, equal to 1. Geometrically, we're now looking for a hyperplane separating the positive and negative points that passes through the origin.

Second, if we multiply all the coordinates of each negative point $\mathbf{y}^i \in \mathbb{R}^{d+1}$ by -1, then we can write the constraints as

$$h(\mathbf{x}^i), h(\mathbf{y}^i) > 0$$

for all positive and negative data points. (For this reason, we will no longer distinguish positive and negative points.) Geometrically, we're now looking for a hyperplane (through the origin) such that all of the data points are on the same side of the hyperplane.

Third, we can insist that every coefficient a_j is nonnegative. (Don't forget that the coordinates of the $\mathbf{x}^i$'s can be both positive and negative.) The trick here is to make two copies of every coordinate (blowing up the dimension from $d+1$ to $2d+2$), and interpreting the two coefficients a'_j, a''_j corresponding to the jth coordinate as indicating the coefficient $a_j = a'_j - a''_j$ in the original space. For this to work, each entry $\mathbf{x}^i_j$ of a data point is replaced by two entries, $\mathbf{x}^i_j$ and $-\mathbf{x}^i_j$. Geometrically, we're now looking for a hyperplane, through the origin and with a normal vector in the nonnegative orthant, with all the data points on the same side (and the same side as the normal vector).

For the rest of this section, we use d to denote the number of dimensions after all of this preprocessing (i.e., we redefine d to be what was previously $2d+2$).

3.3 Assumption

We assume that the problem is feasible — that there is a linear function of the desired type. Actually, we assume a bit more, that there is a solution with some "wiggle room."

Assumption: There is a coefficient vector $\mathbf{a}^* \in \mathbb{R}^d_+$ such that:

1. $\sum_{j=1}^d a^*_i = 1$; and
2. $\sum_{j=1}^d a^*_j x^i_j > \underbrace{\epsilon}_{\text{"margin"}}$ for all data points $\mathbf{x}^i$.

Note that if there is any solution to the problem, then there is a solution satisfying the first condition (just by scaling the coefficients). The second condition insists on wiggle room after normalizing the coefficients to sum to 1.

Let M be such that $|x^i_j| \leq M$ for every i and j. The running time of our algorithm will depend on both ϵ and M.

3.4 Algorithm

Here is the algorithm.

1. Define an action set $A = \{1, 2, \ldots, d\}$, with actions corresponding to coordinates.

2. For $t = 1, 2, \ldots, T = \frac{4M^2 \ln d}{\epsilon^2}$:

   (a) Use the multiplicative weights algorithm to generate a probability distribution $\mathbf{a}^t \in \mathbb{R}^d$ over the actions/coordinates.

   (b) If $\sum_{j=1}^d a_j^t x_j^i > 0$ for every data point $\mathbf{x}^i$, then halt and return $\mathbf{a}^t$ (which is a feasible solution).

   (c) Otherwise, choose some data point $\mathbf{x}^i$ with $\sum_{j=1}^d a_j^t x_j^i \leq 0$, and define a reward vector $\mathbf{r}^t$ with $r^t(j) = x_j^i$ for each coordinate j.

   (d) Feed the reward vector $\mathbf{r}^t$ into the multiplicative weights algorithm.

To motivate the choice of reward vector, suppose the coefficient vector $\mathbf{a}^t$ fails to have a positive inner product $\sum_{j=1}^d a_j^t x_j^i$ with the data point $\mathbf{x}^i$. We want to nudge the coefficients so that this inner product will go up in the next iteration. (Of course we might screw up some other inner products, but we're hoping it'll work out OK in the end.) For coordinates j with $x_j^i > 0$ we want to increase a_j; for coordinates with $x_j^i < 0$ we want to do the opposite. Recalling the multiplicative weight update rule ($w^{t+1}(a) = w^t(a)(1 + \eta r^t(a))$), we see that the reward vector $\mathbf{r}^t = \mathbf{x}^i$ will have the intended effect.

3.5 Analysis

We claim that the algorithm above halts (necessarily with a feasible solution) by the time it gets to the final iteration T.

In the algorithm, the reward vectors are nefariously defined so that, at every time step t, the inner product of $\mathbf{a}^t$ and $\mathbf{r}^t$ is non-positive. Viewing $\mathbf{a}^t$ as a probability distribution over the actions $\{1, 2, \ldots, d\}$, the means that the expected reward of the multiplicative weights algorithm is non-positive at every time step, and hence its time-averaged expected reward is at most 0.

On the other hand, by assumption (Section 3.3), there exists a coefficient vector (equivalently, distribution over $\{1, 2, \ldots, d\}$) $\mathbf{a}^*$ such that, at every time step t, the expected payoff of playing $\mathbf{a}^*$ would have been $\sum_{j=1}^d a_j^* r^t(j) \geq \min_{i=1}^m \sum_{j=1}^d a_j^* x_j^i > \epsilon$.

Combining these two observations, we see that as long as the algorithm has not yet found a feasible solution, the time-averaged regret of the multiplicative weights subroutine is strictly more than ϵ. The multiplicative weights guarantee says that after $T = \frac{4M^2 \ln d}{\epsilon^2}$, the time-averaged regret is at most ϵ.[4] We conclude that our algorithm halts, with a feasible linear classifier, within T iterations.

[4] We're using the extended version of the guarantee (Section 1), which holds against every fixed distribution (like $\mathbf{a}^*$) and not just every fixed action.

3.6 Interpretation as a Zero-Sum Game

Our last two topics were a thought experiment leading to minimax payoffs in zero-sum games and an algorithm for computing a linear classifier. *The latter is just a special case of the former.*

To translate the linear classifier problem to a zero-sum game, introduce one row for each of the d coordinates and one column for each of the data points $\mathbf{x}^i$. Define the payoff matrix $\mathbf{A}$ by

$$\mathbf{A} = \left[a_{ji} = x^i_j\right]$$

Recall that in our thought experiment (Section 2), the row player generates a strategy at each time step using the multiplicative weights algorithm. This is exactly how we generate the coefficient vectors $\mathbf{a}^1, \ldots, \mathbf{a}^T$ in the algorithm in Section 3.4. In the thought experiment, the column player, knowing the row player's distribution, chooses the column that minimizes the expected payoff of the row player. In the linear classifier context, given $\mathbf{a}^t$, this corresponds to picking a data point $\mathbf{x}^i$ that minimizes $\sum_{j=1}^d a^t_j x^i_j$. This ensures that a violated data point (with nonpositive dot product) is chosen, provided one exists. In the thought experiment, the reward vector $\mathbf{r}^t$ fed into the multiplicative weights algorithm is the payoff of each row in hindsight, given the column player's strategy at time t. With the payoff matrix $\mathbf{A}$ above, this vector corresponds to the data point $\mathbf{x}^i$ chosen by the column player at time t. These are exactly the reward vectors used in our algorithm for computing a linear classifier.

Finally, the assumption (Section 3.3) implies that the value of the constructed zero-sum game is bigger than ϵ (since the row player could always choose $\mathbf{a}^*$). The regret guarantee in Section 2 translates to the row player having time-averaged expected payoff bigger than 0 once T exceeds $\frac{4M^2 \ln m}{\epsilon^2}$. The algorithm has no choice but to halt (with a feasible solution) before this time.

4 Maximum Flow Revisited

4.1 Multiplicative Weights and Linear Programs

We've now seen a concrete example of how to approximately solve a linear program using the multiplicative weights algorithm, by modeling the linear program as a zero-sum game and then applying the thought experiment from Section 2. The resulting algorithm is extremely fast (faster than solving the linear program exactly) provided the margin ϵ is not overly small and the radius M of the ℓ_∞ ball enclosing all of the data points $\mathbf{x}^i_j$ is not overly big.

This same idea — associating one player with the decision variables and a second player with the constraints — can be used to quickly approximate many other linear programs. We'll prove this point by considering one more example, our old friend the maximum flow problem. Of course, we already know some pretty good algorithms (faster than linear programs) for maximum flow problems, but the ideas we'll discuss extend also to multicommodity flow problems (see Exercise Set #6 and Problem Set #3), where we don't know any exact algorithms that are significantly faster than linear programming.

4.2 A Zero-Sum Game for the Maximum Flow Problem

Recall the primal-dual pair of linear programs corresponding to the maximum flow and minimum cut problems (Lecture #8):

$$\max \sum_{P \in \mathcal{P}} f_P$$

subject to

$$\underbrace{\sum_{P \in \mathcal{P}\,:\, e \in P} f_P}_{\text{total flow on } e} \leq 1 \qquad \text{for all } e \in E$$

$$f_P \geq 0 \qquad \text{for all } P \in \mathcal{P}$$

and

$$\min \sum_{e \in E} \ell_e$$

subject to

$$\sum_{e \in P} \ell_e \geq 1 \qquad \text{for all } P \in \mathcal{P}$$

$$\ell_e \geq 0 \qquad \text{for all } e \in E,$$

where $\mathcal{P}$ denotes the set of s-t paths. To reduce notation, here we'll only consider the case where all edges have unit capacity ($u_e = 1$). The general case, with u_e's on the right-hand side of the primal and in the objective function of the dual, can be solved using the same ideas (Exercise Set #6).[5]

We begin by defining a zero-sum game. The row player will be associated with edges (i.e., dual variables) and the column player with paths (i.e., primal variables). The payoff matrix is

$$\mathbf{A} = \left[a_{eP} = \begin{cases} 1 \text{ if } e \in P \\ 0 \text{ otherwise} \end{cases} \right]$$

Note that all payoffs are 0 or 1. (Yes, this a huge matrix, but we'll never have to write it down explicitly; see the algorithm below.)

Let OPT denote the optimal objective function value of the linear programs. (The same for each, by strong duality.) Recall that the value of a zero-sum game is defined as the expected payoff of the row player under optimal play by both players ($\max_{\mathbf{x}} \min_{\mathbf{y}} \mathbf{x}^T \mathbf{A} \mathbf{y}$ or, equivalently by the minimax theorem, $\min_{\mathbf{y}} \max_{\mathbf{x}} \mathbf{x}^T \mathbf{A} \mathbf{y}$).

Claim: The value of this zero-sum game is $\frac{1}{OPT}$.

[5] Although the running time scales quadratically with ratio of the maximum and minimum edge capacities, which is not ideal. One additional idea ("width reduction"), not covered here, recovers a polynomial-time algorithm for general edge capacities.

Proof: Let $\{\ell_e^*\}_{e\in E}$ be an optimal solution to the dual, with $\sum_{e\in E}\ell_e^* = OPT$. Obtain x_e's from the ℓ_e^*'s by scaling down by OPT — then the x_e's form a probability distribution. If the row player uses this mixed strategy $\mathbf{x}$, then each column $P \in \mathcal{P}$ results in expected payoff

$$\sum_{e\in P} x_e = \frac{1}{OPT}\sum_{e\in P}\ell_e^* \geq \frac{1}{OPT},$$

where the inequality follows the dual feasibility of $\{\ell_e^*\}_{e\in E}$. This shows that the value of the game is at least $\frac{1}{OPT}$.

Conversely, let $\mathbf{x}$ be an optimal strategy for the row player, with $\min_{\mathbf{y}} \mathbf{x}^T\mathbf{A}\mathbf{y}$ equal to the game's value v. This means that, no matter what strategy the column player chooses, the row player's expected payoff is at least v. This translates to

$$\sum_{e\in P} x_e \geq v$$

for every $P \in \mathcal{P}$. Thus $\{x_e/v\}_{e\in E}$ is a dual feasible solution, with objective function value $(\sum_{e\in E} x_e)/v = 1/v$. Since this can only be larger than OPT, $v \leq \frac{1}{OPT}$. ■

4.3 Algorithm

For simplicity, assume that OPT is known.[6] Translating the thought experiment from Section 2 to this zero-sum game, we get the following algorithm:

1. Associate an action with each edge $e \in E$.
2. For $t = 1, 2, \ldots, T = \frac{4OPT^2 \ln |E|}{\epsilon^2}$:
   (a) Use the multiplicative weights algorithm to generate a probability distribution $\mathbf{x}^t \in \mathbb{R}^E$ over the actions/edges.
   (b) Let P^t be a column that minimizes the row player's expected payoff (with the expectation with respect to $\mathbf{x}^t$). That is,
   $$P^t \in \operatorname*{argmin}_{P\in\mathcal{P}} \sum_{e\in P} x_e^t. \tag{1}$$
   (c) Define a reward vector $\mathbf{r}^t$ with $r^t(e) = 1$ for $e \in P^t$ and $r^t(e) = 0$ for $e \notin P^t$ (i.e., the P^tth column of $\mathbf{A}$). Feed the reward vector $\mathbf{r}^t$ into the multiplicative weights algorithm.

[6]For example, embed the algorithm into an outer loop that uses successive doubling to "guess" the value of OPT (i.e., take $OPT = 1, 2, 4, 8, \ldots$ until the algorithm succeeds).

4.4 Running Time

An important observation is that this algorithm never explicitly writes down the payoff matrix $\mathbf{A}$. It maintains one weight per edge, which is a reasonable amount of state. To compute P^t and the induced reward vector $\mathbf{r}^t$, all that is needed is a subroutine that solves (1) — that, given the x_e^t's, returns a shortest s-t path (viewing the x_e^t's as edge lengths). Dijkstra's algorithm, for example, works just fine.[7] Assuming Dijkstra's algorithm is implemented in $O(m \log n)$ time, where m and n denote the number of edges and vertices, respectively, the total running time of the algorithm is $O(\frac{OPT^2}{\epsilon^2} m \log m \log n)$. (Note that with unit capacities, $OPT \le m$. If there are no parallel edges, then $OPT \le n-1$.) This is comparable to some of the running times we saw for (exact) maximum flow algorithms, but more importantly these ideas extend to more general problems, including multicommodity flow.

4.5 Approximate Correctness

So how do we extract an approximately optimal flow from this algorithm? After running the algorithm above, let $P^1, \ldots, P^T$ denote the sequence of paths chosen by the column player (the same path can be chosen multiple times). Let f^t denote the flow that routes OPT units of flow on the path P^t. (Of course, this probably violates the edge capacity constraints.) Finally, define $f^* = \frac{1}{T}\sum_{t=1}^{T} f^t$ as the "time-average" of these path flows. Note that since each f^t routes OPT units of flow from the source to the sink, so does f^*. But is f^* feasible?

Claim: f^* routes at most $1+\epsilon$ units of flow on every edge.

Proof: We proceed by contradiction. If f^* routes more than $1+\epsilon$ units of flow on the edge e, then more than $(1+\epsilon)T/OPT$ of the paths in $P^1, \ldots, P^T$ include the edge e. Returning to our zero-sum game $\mathbf{A}$, consider the row player strategy $\mathbf{z}$ that deterministically plays the edge e. The time-averaged payoff to the row player, in hindsight given the paths chosen by the column player, would have been

$$\frac{1}{T}\sum_{t=1}^{T} \mathbf{z}^T \mathbf{A} \mathbf{y}^t = \frac{1}{T}\sum_{t\,:\,e \in P^t} 1 > \frac{1+\epsilon}{OPT}.$$

The row player's guarantee (Claim 1 in Section 2) then implies that

$$\frac{1}{T}\sum_{t=1}^{T} (\mathbf{x}^t)^T \mathbf{A} \mathbf{y}^t \ge \frac{1}{T}\sum_{t=1}^{T} \mathbf{z}^T \mathbf{A} \mathbf{y}^t - \frac{\epsilon}{OPT} > \frac{1+\epsilon}{OPT} - \frac{\epsilon}{OPT} = \frac{1}{OPT}.$$

But this contradicts the guarantee that the column player does at least as well as the minimax value of the game (Claim 2 in Section 2), which is $\frac{1}{OPT}$ by the Claim in Section 4.2. ■

Scaling down f^* by a factor of $1+\epsilon$ yields a feasible flow with value at least $OPT/(1+\epsilon)$.

[7] This subroutine is precisely the "separation oracle" for the dual linear program, as discussed in Lecture #10 in the context of the ellipsoid method.