CS261: A Second Course in Algorithms
Lecture #14: Online Bipartite Matching*

Tim Roughgarden†

February 18, 2016

1 Online Bipartite Matching

Our final lecture on online algorithms concerns the *online bipartite matching* problem. As usual, we need to specify how the input arrives, and what decision the algorithm has to make at each time step. The setup is:

- The left-hand side vertices L are known up front.
- The right-hand side vertices R arrive online (i.e., one-by-one). A vertex $w \in R$ arrives together with all of the incident edges (the graph is bipartite, so all of w's neighbors are in L).
- The only time that a new vertex $w \in R$ can be matched is immediately on arrival.

The goal is to construct as large a matching as possible. (There are no edge weights, we're just talking about maximum-cardinality bipartite matching.) We'd love to just wait until all of the vertices of R arrive and then compute an optimal matching at the end (e.g., via a max flow computation). But with the vertices of R arriving online, we can't expect to always do as well as the best matching in hindsight.

This lecture presents the ideas behind optimal (in terms of worst-case competitive ratio) deterministic and randomized online algorithms for online bipartite matching. The randomized algorithm is based on a non-obvious greedy algorithm. While the algorithms do not reference any linear programs, we will nonetheless prove the near-optimality of our algorithms by exhibiting a feasible solution to the dual of the maximum matching problem. This demonstrates that the tools we developed for proving the optimality of algorithms (for max flow, linear programming, etc.) are more generally useful for establishing the approximate optimality of algorithms. We will see many more examples of this in future lectures.

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

Online bipartite matching was first studied in 1990 (when online algorithms were first hot), but a new 21st-century killer application has rekindled interest on the problem over the past 7-8 years. (Indeed, the main proof we present was only discovered in 2013!)

The killer application is Web advertising. The vertices of L, which are known up front, represent advertisers who have purchased a contract for having their ad shown to users that meet specified demographic criteria. For example, an advertiser might pay (in advance) to have their ad shown to women between the ages of 25 and 35 who live within 100 miles of New York City. If an advertiser purchased 5000 views, then there will be 5000 corresponding vertices on the left-hand side. The right-hand side vertices, which arrive online, correspond to "eyeballs." When someone types in a search query or accesses a content page (a new opportunity to show ads), it corresponds to the arrival of a vertex $w \in R$. The edges incident to w correspond to the advertisers for whom w meets their targeting criteria. Adding an edge to the matching then corresponds to showing a given ad to the newly arriving eyeball. Both Google and Microsoft (and probably other companies) employ multiple people whose primary job is adapting and fine-tuning the algorithms discussed in this lecture to generate as much as revenue as possible.

2 Deterministic Algorithms

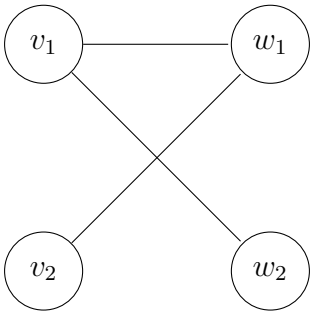

Figure 1: Graph where no deterministic algorithm has competitive ratio better than $\frac{1}{2}$.

We first observe that no deterministic algorithm has a competitive ratio better than $\frac{1}{2}$. Consider the example in Figure 1. The two vertices v_1, v_2 on the left are known up front, and the first vertex w_1 to arrive on the right is connected to both. Every deterministic algorithm picks either the edge (v_1, w_1) or (v_2, w_1).[1] In the former case, suppose the second vertex w_2 to arrive is connected only to v_1, which is already matched. In this case the online algorithm's solution has 1 edge, while the best matching in hindsight has size 2. The other case is symmetric. Thus for every deterministic algorithm, there is an instance where the matching it outputs is at most $\frac{1}{2}$ times the maximum possible in hindsight.

[1]Technically, the algorithm could pick neither, but then its competitive ratio would be 0 (what if no more vertices arrive?).

The obvious greedy algorithm has a matching competitive ratio of $\frac{1}{2}$. By the "obvious algorithm" we mean: when a new vertex $w \in R$ arrives, match w to an arbitrary unmatched neighbor (or to no one, if it has no unmatched neighbors).

Proposition 2.1 *The deterministic greedy algorithm has a competitive ratio of $\frac{1}{2}$.*

Proof: The proposition is easy to prove directly, but here we'll give a more-sophisticated-than-necessary proof because it introduces ideas that we'll build on in the randomized case. Our proof uses a dual feasible solution as an upper bound on the size of a maximum matching. Recall the relevant primal-dual pair ((P) and (D), respectively):

$$\max \sum_{e \in E} x_e$$

subject to

$$\begin{aligned} \sum_{e \in \delta(v)} x_e &\leq 1 \qquad \text{for all } v \in L \cup R \\ x_e &\geq 0 \qquad \text{for all } e \in E, \end{aligned}$$

and

$$\min \sum_{v \in L \cup R} p_v$$

subject to

$$\begin{aligned} p_v + p_w &\geq 1 \qquad \text{for all } (v, w) \in E \\ p_v &\geq 0 \qquad \text{for all } v \in L \cup R. \end{aligned}$$

There are some minor differences with the primal-dual pair that we considered in Lecture #9, when we discussed the minimum-cost perfect matching problem. First, in (P), we're maximizing cardinality rather than minimizing cost. Second, we allow matchings that are not perfect, so the constraints in (P) are inequalities rather than equalities. This leads to the expected modifications of the dual: it is a minimization problem rather than a maximization problem, therefore with greater-than-or-equal-to constraints rather than less-than-or-equal-to constraints. Because the constraints in the primal are now inequality constraints, the dual variables are now nonnegative (rather than unrestricted).

We use these linear programs (specifically, the dual) only for the analysis; the algorithm, remember, is just the obvious greedy algorithm. We next define a "pre-dual solution" as follows: for every $v \in L \cup R$, set

$$q_v = \begin{cases} \frac{1}{2} & \text{if greedy matches } v \\ 0 & \text{otherwise.} \end{cases}$$

The q's are defined in hindsight, purely for the sake of analysis. Or if you prefer, we can imagine initializing all of the q_v's to 0 and then updating them in tandem with the execution

of the greedy algorithm — when the algorithm adds a vertex (v, w) to its matching, we set both q_v and q_w to $\frac{1}{2}$. (Since the chosen edges form a matching, a vertex has its q-value set to $\frac{1}{2}$ at most once.) This alternative description makes it clear that

$$|M| = \sum_{v \in L \cup R} q_v, \tag{1}$$

where M is the matching output by the greedy algorithm. (Whenever one edge is added to the matching, two vertices have their q-values increased to $\frac{1}{2}$.)

Next, observe that for every edge (v, w) of the final graph $(L \cup R, E)$, at least one of q_v, q_w is $\frac{1}{2}$ (if not both). For if $q_v = 0$, then v was not matched by the algorithm, which means that w had at least one unmatched neighbor when it arrived, which means the greedy algorithm matched it (presumably to some other unmatched neighbor) and hence $q_w = \frac{1}{2}$.

This observation does not imply that $\mathbf{q}$ is a feasible solution to the dual linear program (D), which requires a sum of at least 1 from the endpoints of every edge. But it does imply that after scaling up $\mathbf{q}$ by a factor 2 to obtain $\mathbf{p} = 2\mathbf{q}$, $\mathbf{p}$ is feasible for (D). Thus

$$|M| = \frac{1}{2} \underbrace{\sum_{v \in L \cup R} p_v}_{\text{obj fn of } \mathbf{p}} \geq \frac{1}{2} \cdot OPT,$$

where OPT denotes the size of the maximum matching in hindsight. The first equation is from (1) and the definition of $\mathbf{p}$, and the inequality is from weak duality (when the primal is a maximization problem, every feasible dual solution provides an upper bound on the optimum). ■

3 Online Fractional Bipartite Matching

3.1 The Problem

We won't actually discuss randomized algorithms in this lecture. Instead, we'll discuss a deterministic algorithm for the *fractional* bipartite matching problem. The keen reader will object that this is a stupid idea, because we've already seen that the fractional and integral bipartite matching problems are really the same.[2] While it's true that fractions don't help the optimal solution, *they do help an online algorithm*, intuitively by allowing it to "hedge." This is already evident in our simple bad example for deterministic algorithms (Figure 1). When w_1 shows up, in the integral case, a deterministic online algorithm has to match w_1 fully to either v_1 or v_2. But in the fractional case, it can match w_1 50/50 to both v_1 and v_2. Then when w_2 arrives, with only one neighbor on the left-hand side, it can at least be matched with a fractional value of $\frac{1}{2}$. The online algorithm produces a fractional matching

[2] In Lecture #9 we used the correctness of the Hungarian algorithm to argue that the fractional problem always has a 0-1 optimal solution (since the algorithm terminates with an integral solution and a dual-feasible solution with same objective function value). See also Exercise Set #5 for a direct proof of this.

with value $\frac{3}{2}$ while the optimal solution has size 2. So this only proves a bound of $\frac{3}{4}$ of the best-possible competitive ratio, leaving open the possibility of online algorithms with competitive ratio bigger than $\frac{1}{2}$.

3.2 The Water Level (WL) Algorithm

We consider the following "Water Level," algorithm, which is a natural way to define "hedging" in general.

Water-Level (WL) Algorithm

Physical metaphor:
think of each vertex $v \in L$ as a water container with a capacity of 1
think of each vertex $w \in R$ as a source of one unit of water

when $w \in R$ arrives:
drain water from w to its neighbors, always preferring the containers with the lowest current water level, until either
(i) all neighbors of w are full; or
(ii) w is empty (i.e., has sent all its water)

See also Figure 2 for a cartoon of the water being transferred to the neighbors of a vertex w. Initially the second neighbor has the lowest level so w only sends water to it; when the water level reaches that of the next-lowest (the fifth neighbor), w routes water at an equal rate to both the second and fifth neighbors; when their common level reaches that of the third neighbor, w routes water at an equal rate to these three neighbors with the lowest current water level. In this cartoon, the vertex w successfully transfers its entire unit of water (case (ii)).

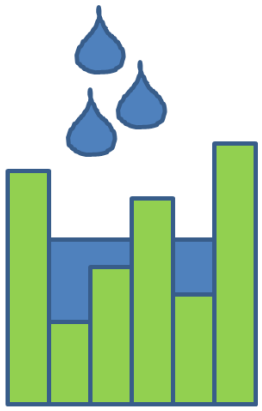

Figure 2: Cartoon of water being transferred to vertices.

For example, in the example in Figure 1, the WL algorithm replicates our earlier hedging, with vertex w_1 distributing its water equally between v_1 and v_2 (triggering case (ii)) and vertex w_2 distributing $\frac{1}{2}$ units of water to its unique neighbor (triggering case (i)).

This algorithm is natural enough, but all you'll have to remember for the analysis is the following key property.

Lemma 3.1 (Key Property of the WL Algorithm) *Let $(v, w) \in E$ be an edge of the final graph G and $y_v = \sum_{e \in \delta(v)} x_e$ the final water level of the vertex $v \in L$. Then w only sent water to containers when their current water level was y_v or less.*

Proof: Fix an edge (v, w) with $v \in L$ and $w \in R$. The lemma is trivial if $y_v = 1$, so suppose $y_v < 1$ — that the container v is not full at the end of the WL algorithm. This means that case (i) did not get triggered, so case (ii) was triggered, so the vertex w successfully routed all of its water to its neighbors. At the time when this transfer was completed, all containers to which w sent some water have a common level ℓ, and all other neighbors of w have current water level at least ℓ (cf., Figure 2). At the end of the algorithm, since water levels only increase, all neighbors of w have final water level ℓ or more. Since w only sent flow to containers when their current water level was ℓ or less, the proof is complete. ■

3.3 Analysis: A False Start

To prove a bound on the competitive ratio of the WL algorithm, a natural idea is to copy the same analysis approach that worked so well for the integral case (Proposition 2.1). That is, we define a pre-dual solution in tandem with the execution of the WL algorithm, and then scale it up to get a solution feasible for the dual linear program (D) in Section 2.

Idea #1:

- initialize $q_v = 0$ for all $v \in L \cup R$;
- whenever the amount x_{vw} of water sent from w to v goes up by Δ, increase both q_v and q_w by $\Delta/2$.

Inductively, this process maintains at all times that the value of the current fractional matching equals $\sum_{v \in L \cup R} q_v$. (Whenever the matching size increases by Δ, the sum of q-values increases by the same amount.)

The hope is that, for some constant $c > \frac{1}{2}$, the scaled-up vector $\mathbf{p} = \frac{1}{c}\mathbf{q}$ is feasible for (D). If this is the case, then we have proved that the competitive ratio of the WL algorithm is at least c (since its solution value equals c times the objective function value $\sum_{v \in L \cup R} p_v$ of the dual feasible solution $\mathbf{p}$, which in turn is an upper bound on the optimal matching size).

To see why this doesn't work, consider the example shown in Figure 3. Initially there are four vertices on the left-hand side. The first vertex $w_1 \in R$ is connected to every vertex of L, so the WL algorithm routes one unit of water evenly across the four edges. Now every container has a water level of $\frac{1}{4}$. The second vertex $w_2 \in R$ is connected to v_2, v_3, v_4. Since all neighbors have the same water level, w_2 splits its unit of water evenly between the three

containers, bringing their water levels up to $\frac{1}{4}+\frac{1}{3}=\frac{7}{12}$. The third vertex $w_3 \in R$ is connected only to v_3 and v_4. The vertex splits its water evenly between these two containers, but it cannot transfer all of its water; after sending $\frac{5}{12}$ units to each of v_3 and v_4, both containers are full (triggering case (i)). The last vertex $w_4 \in R$ is connected only to v_4. Since v_4 is already full, w_4 can't get rid of any of its water.

The question now is: by what factor to we have to scale up $\mathbf{q}$ to get a feasible solution $\mathbf{p} = \frac{1}{c}\mathbf{q}$ to (D)? Recall that dual feasibility boils down to the sum of p-values of the endpoints of every edge being at least 1. We can spot the problem by examining the edge (v_4, w_4). The vertex v_4 got filled, so its final q-value is $\frac{1}{2}$ (as high as it could be with the current approach). The vertex w_4 didn't participate in the fractional matching at all, so its q-value is 0. Since $q_{v_4} + q_{w_4} = \frac{1}{2}$, we would need to scale up by 2 to achieve dual feasibility. This does not improve over the competitive ration of $\frac{1}{2}$.

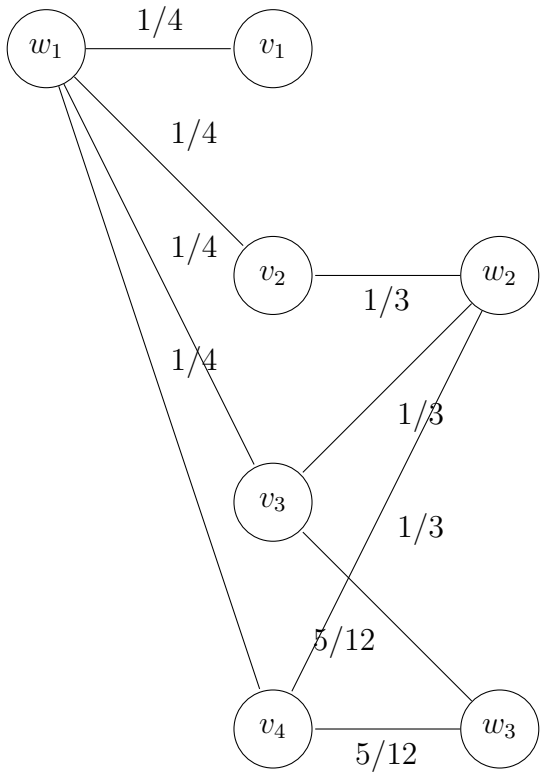

Figure 3: Example showcasing why Idea #1 does not work.

On the other hand, the solution computed by the WL algorithm for this example, while not optimal, is also not that bad. Its value is $1 + 1 + \frac{5}{6} + 0 = \frac{17}{6}$, which is substantially bigger than $\frac{1}{2}$ times the optimal solution (which is 4). Thus this is a bad example only for the analysis approach, and not for the WL algorithm itself. Can we keep the algorithm the same, and just be smarter with its analysis?

3.4 Analysis: The Main Idea

Idea #2: when the amount x_{vw} of water sent from w to v goes up by Δ, split the increase *unequally* between q_v and q_w.

To see the motivation for this idea, consider the bottom edge in Figure 3. The WL algorithm never sends any water on any edge incident to w_4, so it's hard to imagine how its q-value will wind up anything other than 0. So if we want to beat $\frac{1}{2}$, we need to make sure that v_4 finishes with a q-value bigger than $\frac{1}{2}$. A naive fix for this example would be to only increase the q-values for vertices of L, and not from R; but this would fail miserably if w_1 were the only vertex to arrive (then all q-values on the left would be $\frac{1}{4}$, all those on the right 0). To hedge between the various possibilities, as a vertex $v \in L$ gets more and more full, we will increase its q-value more and more quickly. Provided it increases quickly enough as v becomes full, it is conceivable that v could end up with a q-value bigger than $\frac{1}{2}$. Summarizing, we'll use unequal splits between the q-values of the endpoints of an edge, with the splitting ratio evolving over the course of the algorithm.

There are zillions of ways to split an increase of Δ on x_{vw} between q_v and q_w (as a function of v's current water level). The plan is to give a general analysis that is parameterized by such a "splitting function," and solve for the splitting function that leads to the best competitive ratio. Don't forget that all of this is purely for the analysis; the algorithm is always the WL algorithm.

So fix a nondecreasing "splitting function" $g : [0,1] \to [0,1]$. Then:

- initialize $q_v = 0$ for all $v \in L \cup R$;
- whenever the amount x_{vw} of water sent from w to v goes up by an infinitesimal amount dz, and the current water level of v is $y_v = \sum_{e \in \delta(v)} x_e$:
  - increase q_v by $g(y_v)dz$;
  - increase q_w by $(1 - g(y_v))dz$.

For example, if g is the constant function always equal to 0 (respectively, 1), then only the vertices of R (respectively, vertices of L) receive positive q-values. If g is the constant function always equal to $\frac{1}{2}$, then we recover our initial analysis attempt, with the increase on an edge split equally between its endpoints.

By construction, no matter how we choose the function g, we have

$$\text{current value of WL fractional matching} = \text{current value of } \sum_{v \in L \cup R} q_v,$$

at all times, and in particular at the conclusion of the algorithm.

For the analysis (parameterized by the choice of g), fix an arbitrary edge (v, w) of the final graph. We want a worst-case lower bound on $q_v + q_w$ (hopefully, bigger than $\frac{1}{2}$).

For the first case, suppose that at the termination of the WL algorithm, the vertex $v \in L$ is full (i.e., $y_v = \sum_{e \in \delta(v)} x_e = 1$). At the time that v's current water level was z, it accrued q-value at rate $g(z)$. Integrating over these accruals, we have

$$q_v + q_w \geq q_v = \int_0^1 g(z)dz. \tag{2}$$

(It may seem sloppy to throw out the contribution of $q_w \geq 0$, but Figure 3 shows that when v is full it might well be the case that some of its neighbors have q-value 0.) Note that the bigger the function g is, the bigger the lower bound in (2).

For the second case, suppose that v only has water level $y_v < 1$ at the conclusion of the WL algorithm. It follows that w successfully routed its entire unit of water to its neighbors (otherwise, the WL algorithm would have routed more water to the non-full container v). Here's where we use the key property of the WL algorithm (Lemma 3.1): whenever v sent water to a container, the current water level of that container was as most y_v. Thus, since the function g is nondecreasing, whenever v routed any water, it accrued q-value at rate at least $1 - g(y_v)$. Integrating over the unit of water sent, we obtain

$$q_w \geq \int_0^1 (1 - g(y_v))dz = 1 - g(y_v).$$

As in the first case, we have

$$q_v = \int_0^{y_v} g(z)dz$$

and hence

$$q_v + q_w \geq \left(\int_0^{y_v} g(z)dz \right) + 1 - g(y_v). \tag{3}$$

Note the lower bound in (3) is generally larger for smaller functions g (since $1 - g(y_v)$ is bigger). This is the tension between the two cases.

For example, if we take g to be identically 0, then the lower bounds (2) and (3) read 0 and 1, respectively. With g identically equal to 1, the values are reversed. With g identically equal to $\frac{1}{2}$, as in our initial attempt, the right-hand sides of both (2) and (3) are guaranteed to be at least $\frac{1}{2}$ (though not larger).

3.5 Solving for the Optimal Splitting Function

With our lower bounds (2) and (3) on the worst-case value of $q_v + q_w$ for an edge (v, w), our task is clear: we want to solve for the splitting function g that makes the minimum of these two lower bounds as large as possible. If we can find a function g such that the right-hand sides of (2) and (3) (for any $y_v \in [0, 1]$) are both at least c, then we will have proved that the WL algorithm is c-competitive. (Recall the argument: the value of the WL matching is $\sum_v q_v$, and $\mathbf{p} = \frac{1}{c}\mathbf{q}$ is a feasible dual solution, which is an upper bound on the maximum matching.)

Solving for the best nondecreasing splitting function g may seem an intimidating prospect — there are an infinite number of functions to choose from. In situations like this, a good strategy is to "guess and check" — try to develop intuition for what the right answer might look like and then verify your guess. There are many ways to guess, but often in an optimal analysis there is "no slack anywhere" (since otherwise, a better solution could take advantage of this slack). In our context, this corresponds to guessing that the optimal function g equalizes the lower bound in (2) with that in (3), and with the second lower bound tight simultaneously for all values of $y_v \in [0,1]$. There is no a priori guarantee that such a g exists, and if such a g exists, its optimality still needs to be verified. But it's still a good strategy for generating a guess.

Let's start with the guess that the lower bound in (3) is the same for all values of $y_v \in [0,1]$. This means that

$$\left(\int_0^{y_v} g(z)dz\right) + 1 - g(y_v),$$

when viewed as a function of y_v, is a constant function. This means its derivative (w.r.t. y_v) is 0, so

$$g(y_v) - g'(y_v) = 0,$$

i.e., the derivative of g is the same as g.[3] This implies that $g(z)$ has the form $g(z) = ke^z$ for a constant $k > 0$. This is great progress: instead of an infinite-dimensional g to solve for, we now just have the single parameter k to solve for.

Now let's use the guess that the two lower bounds in (2) and (3) are the same. Plugging ke^z into the lower bound in (2) gives

$$\int_0^1 ke^z dz = k\left[e^z\right]_0^1 = k(e-1),$$

which gets larger with k. Plugging ke^z into the lower bound in (3) gives (for any $y \in [0,1]$)

$$\int_0 ke^z dz + 1 - ke^y = k(e^y - 1) + 1 - ke^y = 1 - k.$$

This lower bound is independent of the choice of y — we knew that would happen, it's how we chose $g(z) = ke^z$ – and gets larger with smaller k. Equalizing the two lower bounds of $k(e-1)$ and $1-k$ and solving for k, we get $k = \frac{1}{e}$, and so the splitting function is $g(y) = e^{y-1}$. (Thus when a vertex $v \in L$ is empty it gets a $\frac{1}{e}$ share of the increase of an incident edge; the share increases as v gets more full, and approaches 100% as v becomes completely full.) Our lower bounds in (2) and (3) are then both equal to

$$1 - \frac{1}{e} \approx 63.2\%.$$

This proves that the WL algorithm is $(1 - \frac{1}{e})$-competitive, a significant improvement over the more obvious $\frac{1}{2}$-competitive algorithm.

[3] I don't know about you, but this is pretty much the only differential equation that I remember how to solve.

3.6 Epilogue

In this lecture we gave a $(1 - \frac{1}{e})$-competitive (deterministic) online algorithm for the online *fractional* bipartite matching problem. The same ideas can be used to design a randomized online algorithm for the original *integral* online bipartite matching problem that always outputs a matching with expected size at least $1 - \frac{1}{e}$ times the maximum possible. (The expectation is over the random coin flips made by the algorithm.) The rough idea is to set things up so that the probability that a given edge is included the matching plays the same role as its fractional value in the WL algorithm. Implementing this idea is not trivial, and the details are outlined in Problem Set #4.

But can we do better? Either with a smarter algorithm, or with a smarter analysis of these same algorithms? (Recall that being smarter improved the analysis of the WL algorithm from a $\frac{1}{2}$ to a $1-\frac{1}{e}$.) Even though $1-\frac{1}{e}$ may seem like a weird number, the answer is negative: *no* online algorithm, deterministic or randomized, has a competitive ratio better than $1 - \frac{1}{e}$ for maximum bipartite matching. The details of this argument are outlined in Problem Set #3.