Cornell University, Fall 2010	**CS 6820: Algorithms**
Lecture notes: Primal-dual min-cost bipartite matching	**August 27–30**

1 Linear programming relaxation

Recall that in the bipartite minimum-cost perfect matching problem, we are given a pair of vertex sets X, Y, each having cardinality n, and a matrix of edge costs $c(x, y)$, also denoted as $c(e)$ for edge $e = (x, y)$. We assume that all edge costs belongs to $\mathbb{R}_+ \cup \{\infty\}$ and let m denote the number of edges whose cost is finite.

A perfect matching M can be described by a matrix (m_{xy}) of 0's and 1's, where $m_{xy} = 1$ if and only if $(x, y) \in M$. The sum of the entries in each row and column of this matrix equals 1, since each vertex belongs to exactly one element of M. Conversely, for any matrix with $\{0, 1\}$-valued entries, if each row sum and column sum is equal to 1, then the corresponding set of edges is a perfect matching. Thus, the bipartite minimum-cost matching problem can be expressed as follows.

$$\begin{array}{lll} \min & \sum_{x,y} c(x,y) m_{xy} & \\ \text{s.t.} & \sum_y m_{xy} = 1 & \forall x \\ & \sum_x m_{xy} = 1 & \forall y \\ & m_{xy} \in \{0,1\} & \forall x, y \end{array}$$

This is a discrete optimization problem because of the constraint that $m_{xy} \in \{0, 1\}$. Although we already know how to solve this discrete optimization problem in polynomial time, many other such problems are not known to have any polynomial-time solution. It's often both interesting and useful to consider what happens when we relax the constraint $m_{xy} \in \{0, 1\}$ to $m_{xy} \geq 0$, allowing the variables to take any non-negative real value. This turns the problem into a continuous optimization problem, in fact a *linear program.*

$$\begin{array}{lll} \min & \sum_{x,y} c(x,y) m_{xy} & \\ \text{s.t.} & \sum_y m_{xy} = 1 & \forall x \\ & \sum_x m_{xy} = 1 & \forall y \\ & m_{xy} \geq 0 & \forall x, y \end{array}$$

How should we think about a matrix of values m_{xy} satisfying the constraints of this linear program? We've seen that if the values are integers, then it represents a perfect matching. A general solution of this constraint set can be regarded as a *fractional perfect matching.* What does a fractional perfect matching look like? An example is illustrated in Figure 1. Is it possible that this fractional perfect matching achieves a lower cost than any perfect matching? No, because it can be expressed as a convex combination of perfect matchings (again, see Figure 1) and consequently its cost is the weighted average of the costs of those perfect matchings. In particular, at least one of those perfect matchings costs no more than the fractional perfect matching illustrated on the left side of the figure. This state of affairs is not a coincidence. The *Birkhoff-von Neumann Theorem* asserts that every fractional perfect matching can be decomposed as a convex combination of perfect matchings. (Despite the eminence of its namesakes, the theorem is actually quite easy to prove. You should try finding a proof yourself, if you've never seen one.)

Now suppose we have an instance of bipartite minimum-cost perfect matching, and we want to prove a *lower bound* on the optimum: we want to prove that every fractional perfect matching

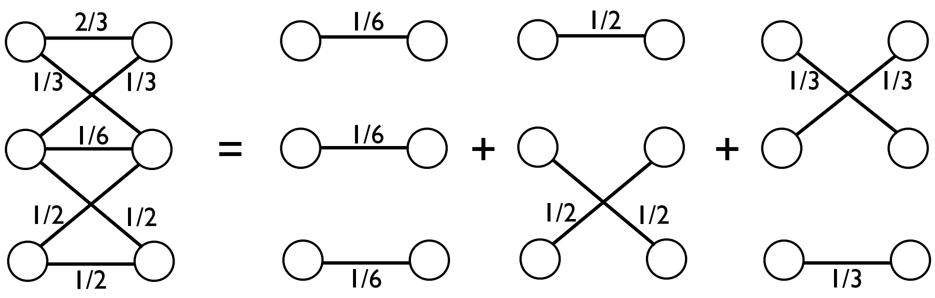

Figure 1: A fractional perfect matching.

has to cost at least a certain amount. How might we prove this? One way is to run a minimum-cost perfect matching algorithm (such as the one we saw in the previous lecture), look at its output, and declare this to be a lower bound on the cost of any fractional perfect matching. By the Birkhoff-von Neumann Theorem, this produces a valid lower bound, but it's not very satisfying. There's another, much more direct, way to prove lower bounds on the cost of every fractional perfect matching, by directly combining constraints of the linear program. To illustrate this, consider the graph with edge costs as shown in Figure 2. Clearly, the minimum cost perfect

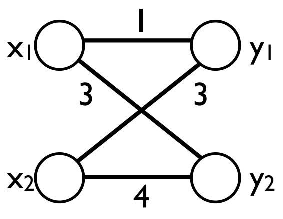

Figure 2: An instance of bipartite minimum cost perfect matching.

matching has cost 5. To prove that no fractional perfect matching can cost less than 5, we combine some constraints of the linear program as follows.

$$\begin{aligned} 2m_{11} + 2m_{21} &= 2 \\ -m_{11} - m_{12} &= -1 \\ 4m_{12} + 4m_{22} &= 4 \end{aligned}$$

Adding these constraints, we find that

$$m_{11} + 3m_{12} + 2m_{21} + 4m_{22} = 5 \tag{1}$$
$$m_{11} + 3m_{12} + 3m_{21} + 4m_{22} \geq 5 \tag{2}$$

Inequality (2) is derived from (1) because the only change we made on the left side was to increase the coefficient of m_{21} from 2 to 3, and we know that $m_{21} \geq 0$. The left side of (2) is the cost of the fractional perfect matching $\vec{m}$. We may conclude that the cost of every fractional perfect matching is at least 5.

What's the most general form of this technique? For every vertex $v \in X \cup Y$, the linear program contains a "degree constraint" asserting that the degree of v in the fractional perfect matching is equal to 1. For each degree constraint, we multiply its left and right sides by some coefficient to obtain

$$\sum_y p_x m_{xy} = p_x$$

for some $x \in X$, or

$$\sum_x q_y m_{xy} = q_y$$

for some $y \in Y$. Then we sum all of these equations, obtaining

$$\sum_{x,y} (p_x + q_y) m_{xy} = \sum_x p_x + \sum_y q_y. \tag{3}$$

If the inequality $p_x + q_y \geq c(x, y)$ holds for every edge (x, y), then in the final step of the proof we (possibly) increase some of the coefficients on the left side of (3) to obtain

$$\sum_{x,y} c(x, y) m_{xy} \geq \sum_x p_x + \sum_y q_y,$$

thus obtaining a lower bound on the cost of every fractional perfect matching. This technique works whenever the coefficients p_x, q_y satisfy $p_x + q_y \leq c(x, y)$ for every edge (x, y), regardless of whether the values p_x, q_y are positive or negative. To obtain the strongest possible lower bound using this technique, we would set the coefficients p_x, q_y by solving the following linear program.

$$\begin{array}{ll} \max & \sum_x p_x + \sum_y q_y \\ \text{s.t.} & p_x + q_y \leq c(x, y) \quad \forall x, y \end{array}$$

This linear program is called the *dual* of the min-cost-fractional-matching linear program. We've seen that its optimum constitutes a lower bound on the optimum of the min-cost-fractional-matching LP. For any linear program, one can follow the same train of thought to develop a dual linear program. (There's also a formal way of specifying the procedure; it involves taking the transpose of the constraint matrix of the LP.) The dual of a minimization problem is a maximization problem, and its optimum constitutes a lower bound on the optimum of the minimization problem. This fact is called **weak duality**; as you've seen, weak duality is nothing more than an assertion that we can obtain valid inequalities by taking linear combinations of other valid inequalities, and that this sometimes allows us to bound the value of an LP solution from above or below. But actually, the optimum value of an LP is always *exactly equal* to the value of its dual LP! This fact is called **strong duality** (or sometimes simply "duality"), it is far from obvious, and it has important ramifications for algorithm design. In the special case of fractional perfect matching problems, strong duality says that the simple proof technique exemplified above is actually powerful enough to prove the *best possible* lower bound on the cost of fractional perfect matchings, for *every* instance of the bipartite min-cost perfect matching problem.

It turns out that there is a polynomial-time algorithm to solve linear programs. As you can imagine, this fact also has extremely important ramifications for algorithm design, but that's the topic of another lecture.

2 Primal-dual algorithm

In this section we will construct a fast algorithm for the bipartite minimum-cost perfect matching algorithm, exploiting insights gained from the preceding section. The basic plan of attack is as follows: we will design an algorithm that simultaneously computes two things: a minimum-cost perfect matching, and a dual solution (vector of p_x and q_y values) whose value (sum of p_x's and q_y's) equals the cost of the perfect matching. As the algorithm runs, it maintains the a dual solution $\vec{p}, \vec{q}$ and a matching M, and it preserves the following invariants:

1. Every edge (x, y) satisfies $p_x + q_y \leq c(x, y)$. If $p_x + q_y = c(x, y)$ we say that edge $e = (x, y)$ is *tight*.
2. The elements of M are a subset of the tight edges.
3. The cardinality of M increases by 1 in each phase of the algorithm, until it reaches n.

Assuming the algorithm can maintain these invariants until termination, its correctness will follow automatically. This is because the matching M at termination time will be a perfect matching satisfying

$$\sum_{(x,y)\in M} c(x, y) = \sum_{(x,y)\in M} p_x + q_y = \sum_{x\in X} p_x + \sum_{y\in Y} q_y,$$

where the final equation holds because M is a perfect matching. The first invariant of the algorithm implies that $\vec{p}, \vec{q}$ is a feasible dual solution, hence the right side is a lower bound on the cost of any fractional perfect matching. The left side is the cost of the perfect matching M, hence M has the minimum cost of any fractional perfect matching.

So, how do we maintain the three invariants listed above while growing M to be a perfect matching? We initialize $M = \emptyset$ and $\vec{p} = \vec{q} = 0$. Note that the three invariants are trivially satisfied at initialization time. Now, as long as $|M| < n$, we want to find a way to either increase the value of the dual solution or enlarge M without violating any of the invariants. The easiest way to do this is to find an M-augmenting path P consisting of tight edges: in that case, we can update M to $M \oplus P$ without violating any invariants, and we reach the end of a phase. However, sometimes it's not possible to find an M-augmenting path consisting of tight edges: in that case, we must adjust some of the dual variables to make additional edges tight.

The process of adjusting dual variables is best described as follows. The easiest thing would be if we could find a vertex $x \in X$ that doesn't belong to any tight edges. Then we could raise p_x by some amount $\delta > 0$ until an edge containing x became tight. However, maybe every $x \in X$ belongs to a tight edge. In that case, we need to raise p_x by δ while lowering some other q_y by the same amount δ. This is best described in terms of a vertex set T called the *Hungarian tree*, which will have the property that if one endpoint of an edge $e \in M$ belongs to T, then both endpoints of e belong to T. Whenever T has this property, we can set

$$\delta = \min\{c(x, y) - p_x - q_y \mid x \in X \cap T, y \in Y \setminus T\} \tag{4}$$

and adjust the dual variables by setting $p_x \leftarrow p_x + \delta, q_y \leftarrow q_y - \delta$ for all $x \in X \cap T, y \in Y \cap T$. This preserves the feasibility of our dual solution $\vec{p}, \vec{q}$ (by the choice of δ) and it preserves the tightness of each edge $e \in M$ because every such edge has either both or neither of its endpoints in T.

Define a *free vertex* to be one that doesn't belong to any element of M. T will be constructed by a sort of breadth-first search along tight edges, starting from the set of free vertices in X. We initialize T to be the set of free vertices in X. Since $|M| < n$, T is nonempty. Define δ as in (4); if $\delta > 0$ then adjust dual variables as explained above. Call this a *dual adjustment step*. If $\delta = 0$ then there is at least one tight edge $e = (x, y)$ from $X \cap T$ to $Y \setminus T$. If y is a free vertex, then we have discovered an augmenting path P consisting of tight edges (namely, P consists of a path in T that starts at a free vertex in X, walks to x, then crosses edge e to get to y) and we update M to $M \oplus P$ and finish the phase. Call this an *augmentation step.* Finally, if y is not a free vertex then we identify an edge $e = (x', y) \in M$ and we add both y and x' to T and call this a *tree growing step.* Notice that the left endpoint of an edge of M is always added to T at the same time as the right endpoint, which is why T never contains one endpoint of an edge of M unless it contains both.

A phase can contain at most n tree growing steps and at most one augmentation step. Also, there can never be two consecutive dual adjustment steps (since the value of δ drops to zero after the first such step) so the total number of steps in a phase is $O(n)$. Let's figure out the running time of one phase of the algorithm by breaking it down into its component parts.

1. There is only one augmentation step and it costs $O(n)$.

2. There are $O(n)$ tree growing steps and each costs $O(1)$.

3. There are $O(n)$ dual adjustment steps and each costs $O(n)$.

4. Finally, every step starts by computing the value δ using (4). Thus, the value of δ needs to be computed $O(n)$ times. Naïvely it costs $O(m)$ work each time we need to compute δ.

Thus, a naïve implementation of the primal-dual algorithm takes $O(mn^2)$ which is no better than the algorithm we saw in the previous lecture.

However, we can do better using some clever book-keeping combined with efficient data structures. For a vertex $v \in T$, let $s(v)$ denote the number of the step in which v was added to T. Let δ_s denote the value of δ in step s of the phase, and let Δ_s denote the sum $\delta_1 + \cdots + \delta_s$. Let $p_{x,s}, q_{y,s}$ denote the values of the dual variables associated to vertices x, y at the end of step s. Note that

$$p_{x,s} = \begin{cases} p_{x,0} + \Delta_s - \Delta_{s(x)} & \text{if } x \in X \cap T \\ p_{x_0} & \text{if } x \in X \setminus T \end{cases} \tag{5}$$

$$q_{y,s} = \begin{cases} q_{y,0} - \Delta_s + \Delta_{s(y)} & \text{if } y \in Y \cap T \\ q_{y,0} & \text{if } y \in Y \setminus T \end{cases} \tag{6}$$

Consequently, if $e = (x, y)$ is any edge from $X \cap T$ to $Y \setminus T$ at the end of step s, then

$$c(x,y) - p_{x,s} - q_{y,s} = c(x,y) - p_{x,0} - \Delta_s + \Delta_{s(x)} - q_{y,0}$$

The only term on the right side that depends on s is $-\Delta_s$, which is a global value that is common to all edges. Thus, choosing the edge that minimizes $c(x,y) - p_{x,s} - q_{y,s}$ is equivalent to choosing the edge that minimizes $c(x,y) - p_{x,0} + \Delta_{s(x)} - q_{y,0}$. Let us maintain a priority queue containing all the edges from $X \cap T$ to $Y \setminus T$. An edge $e = (x, y)$ is inserted into this priority queue at the time its left endpoint x is inserted into T. The value associated to e in the priority queue is $c(x,y) - p_{x,0} + \Delta_{s(x)} - q_{y,0}$, and this value never changes as the phase proceeds. Whenever the algorithm needs to choose the edge that minimizes $c(x,y) - p_{x,s} - q_{y,s}$, it simply extracts the minimum element of this priority queue, repeating as necessary until it finds an edge whose right endpoint does not belong to T. The total amount of work expended on maintaining the priority queue throughout a phase is $O(m \log n)$.

Finally, our gimmick with the priority queue eliminates the need to actually update the values p_x, q_y during a dual adjustment step. These values are only needed for computing the value of δ_s, and for updating the dual solution at the end of the phase. However, if we store the values $s(x), s(y)$ for all x, y as well as the values Δ_s for all s, then one can compute any specific value of $p_{x,s}$ or $q_{y,s}$ in constant time using (5)-(6). In particular, it takes $O(n)$ time to compute all the values p_x, q_y at the end of the phase, and it only takes $O(1)$ time to compute the value $delta_s = c(x, y) - p_x - q_y$ once we have identified the edge $e = (x, y)$ using the priority queue. Thus, all the work to maintain the values p_x, q_y amounts to only $O(n)$ per phase.

In total, the amount of work in any phase is bounded by $O(m \log n)$ and consequently the algorithm's running time is $O(mn \log n)$.