Weighted Matching and Linear Programming

Jonathan Turner

March 19, 2013

We've seen that maximum size matchings can be found in general graphs using augmenting paths. In principle, this same approach can be applied to maximum weight matchings. The main difference is that we need to choose an augmenting path that produces the largest possible increase in weight at each step. This is illustrated in Figure 1. Note that if we augment the

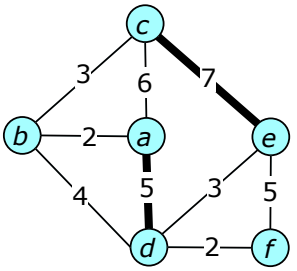

Figure 1: Max weight augmenting paths

matching shown using the path *bcef*, we get a larger matching with a weight that is one larger than that of the original matching. On the other hand, the augmenting path *badf* leads to a *decrease* in the matching weight. The augmenting path that produces the largest increase in weight is *bdacef*.

Define the weight of an augmenting path to be the total weight of its non-matching edges minus the total weight of its matching edges. This path weight is the net gain possible from flipping the edges in the path. Suppose M is a max weight matching of size k (that is, among all matchings with k edges, M has maximum weight) and p is a maximum weight aumenting path with respect to M. If we can show that flipping the edges of p yields a max weight matching of size $k+1$ then we could use this to find maximum weight matchings of arbitrary size.

To shows this, assume that M' is a max weight matching on $k+1$ edges and let N be the set of edges that is in M or M' but not both. Note that N consists of alternating paths and cycles with respect to M, and its cycles all have even length.

Let the weight of a path or cycle in N be the total weight of its edges that are not in M minus the total weight of its edges that are in M. Observe that any even length cycle in N must have weight ≤ 0 since otherwise we could increase the weight of M without increasing its size by flipping the edges in the cycle. The same observation holds for paths of even length.

Now, since N has one more edge from M' than from M, we can pair up all but one of the odd length paths, so that each pair has an equal number of edges from M and M'. By the same argument as before, each such pair of paths must have weight ≤ 0.

The remaining path must be a maximum weight augmenting path with respect to M, since if it were not, we could use a max weight augmenting path to produce a matching of size $k+1$ with larger weight than M'. Hence, we have the following theorem.

Theorem 1 *Let M be a matching of maximum weight among matchings of size k, and let p be a maximum weight augmenting path for M. Then the matching obtained by flipping the edges of p has maximum weight among matchings of size $k+1$.*

This theorem provides the basis for a weighted matching algorithm. Finding a maximum weight augmenting path is difficult to do directly, especially for general graphs. However, we can use linear programming duality to facilitate the search for max weight augmenting paths.

We start by defining a 0-1 selection variable x_e for each edge in the graph; $x_e = 1$ will correspond to the inclusion of edge e in the matching. The matching problem is then equivalent to the integer linear program

$$\begin{array}{lll} \text{maximize} & \mathit{weight}(X) = \sum_e x_e w(e) & \\ \text{subject to} & \sum_{e=\{u,v\}} x_e \leq 1 & \text{for all vertices } u \\ & x_e \in \{0,1\} & \text{for all edges } e \end{array}$$

The objective function maximizes the weight of the edges in the matching, while the constraints limit us to one edge per vertex. If we replace the 0-1 constraints on the selection variables with inequalites of the form $x_e \leq 1$ we get the linear program relaxation of this ILP. We actually don't need to include these inequalities explicitly, since the form of the standard LP already constrains the x_e to be non-negative and consequently, the other

constraints prevent them from being larger than 1. Edmonds showed that for bipartite graphs, the LP relaxation has optimal solutions in which all the selection variables are integers, even though they are not constrained to be. The essential property that makes this true is that the coefficient matrix for the linear program is totally unimodular. Edmonds also showed a similar property for general graphs but we will defer that for now.

We can use duality to develop an efficient algorithm for the bipartite case. The dual problem is

$$\begin{array}{ll} \text{minimize} & \sum_u z_u \\ \text{subject to} & z_u + z_v \geq w(e) \qquad \text{for all edges } e = \{u, v\} \end{array}$$

The dual variables are referred to as vertex labels. For convenience, we define $z_e = z_u + z_v$ for $e = \{u, v\}$. The complementary slackness condition tells us that for optimal values of the primal and dual variables, if $z_e > w(e)$ then x_e must be zero, and if $\sum_{e=\{u,v\}} x_e \leq 1$, z_u must be zero. This is equivalent to saying that for any edge e in the optimal matching, $z_e = w(e)$ and for any free vertex u, $z_u = 0$.

Moreover, if we can find values of the x_e and z_u that satisfy the complementary slackness conditions, these must correspond to optimal solutions, hence the x_e values define a maximum weight matching. We can use this to construct an algorithm that systematically adjusts the primal and dual variables until the complementary slackness conditions are satisfied. The following theorem captures these observations.

Theorem 2 *Let $G = (V, E)$ be a bipartite graph with edge weights $w(e)$, let M be a matching in G, let each vertex u have a non-negative label z_u and let $z_e = z_u + z_v$ for $e = \{u, v\}$. If*

$$\begin{array}{lcll} (1) & z_e \geq w(e) & \textit{for all } e \\ (2) & z_e = w(e) & \textit{for } e \in M \\ (3) & z_u = 0 & \textit{if } u \textit{ is free} \end{array}$$

then M is a maximum weight matching.

Although, we arrived at Theorem 2 using LP duality, it can actually be proved independently, without reference to linar programming at all. If N is any matching, then

$$\sum_{e \in N} w(e) \leq \sum_{e \in N} z_e \leq \sum_u z_u \leq \sum_{e \in M} z_e = \sum_{e \in M} w(e)$$

The first inequality follows from condition (1) in the theorem; the second follows from the fact that N is a matching; the third follows from condition (3); and the final equality follows from condition (2).

An edge $e = \{u, v\}$ with $z_e = z_u + z_v = w(e)$ is called an *equality edge*. Note that if $p = u_0, \ldots, u_{2k+1}$ is an augmenting path, then conditions (1) and (2) in the theorem imply that the weight of p satisfies the following.

$$\begin{aligned}
\mathit{weight}(p) &= (w(u_0, u_1) + w(u_2, u_3) + \cdots + w(u_{2k}, u_{2k+1})) \\
&\quad -(w(u_1, u_2) + w(u_3, u_4) + \cdots + w(u_{2k-1}, u_{2k})) \\
&= (w(u_0, u_1) + w(u_2, u_3) + \cdots + w(u_{2k}, u_{2k+1})) \\
&\quad -(z_{u_1} + z_{u_2} + \cdots + z_{u_{2k}}) \\
&\leq (z_{u_0} + z_{u_2} + \cdots + z_{u_{2k+1}}) - (z_{u_1} + z_{u_2} + \cdots + z_{u_{2k}}) \\
&= z_{u_0} + z_{u_{2k+1}}
\end{aligned}$$

Also, note that if p is constructed entirely from equality edges, its weight is exactly equal to $z_{u_0} + z_{u_{2k+1}}$. Hence, any augmenting path that uses only equality edges is a max weight path, while any augmenting path that uses at least one non-equality edge is not a max weight path.

This leads directly to an algorithm for finding a max weight matching. At all times, we maintain labels that satisfy conditions (1) and (2) and we construct a collection of trees using only equality edges. Whenever we find such an augmenting path, we flip the edges, as usual. Note that this maintains the validity of condition (2) because all edges in the augmenting path are equality edges. At various points in the algorithm, we may use up all the equality edges, preventing us from expanding the collection of trees any further. When this happens, we adjust the labels so as to create additional equality edges, taking care to maintain the validity of conditions (1) and (2). Eventually, the label adjustments cause condition (3) to be satisfied, causing the algorithm to terminate.

Here's a more complete description. First, initialize $M = \{\}$ and for all u, let z_u be half of the largest edge weight. This clearly satisfies conditions (1) and (2). We search for augmenting paths by building a set of trees rooted at the free vertices, as we have done before. At each step, we consider a previously unexamined equality edge $e = \{u, v\}$ with u even. If v is unreached, we extend the tree by adding e and the matching edge incident to v. If v is even, then we have found an augmenting path that we can use to extend the tree. Every time we extend the matching, we discard the current set of trees and the odd/even status information. However, we retain the vertex labels.

If the search examines all equality edges without finding an augmenting path, we adjust the vertex labels as follows. Let

$$\begin{aligned}
\delta_1 &= \min\{z_u \mid u \text{ is even}\} \\
\delta_2 &= \min\{z_e - w(e) \mid e = \{u, v\},\ u \text{ is even},\ v \text{ is unreached}\} \\
\delta_3 &= \min\{(z_e - w(e))/2 \mid e = \{u, v\},\ u \text{ is even},\ v \text{ is even}\} \\
\delta &= \min\{\delta_1, \delta_2, \delta_3\}
\end{aligned}$$

Note that while the algorithm is running, there must be at least one even vertex, so δ_1 is always defined. However, δ_2 and δ_3 may be undefined due to the lack of an edge that satisfies the condition. If either or both are not defined, the definition of δ is adjusted to ignore the undefined δ_i values.

We adjust the labels by subtracting δ from the labels of even vertices and adding δ to the label of odd vertices. Notice that this maintains the validity of condition (2), since matching edges either have both endpoints unreached or have one odd endpoint and one even endpoint. It also maintains existing equality edges that are part of a tree, since these edges also connect an even vertex to and an odd one. In addition, it maintains the validity of condition (1), since no z_u is reduced enough to create a violation. Note however, that equality edges with two odd endpoints may become inequality edges, as a result of a label adjustment.

Observe that if $\delta = \delta_1$, then after the adjustment, condition (3) is satisfied, so the current matching is a max weight matching. If $\delta = \delta_2 < \delta_1$ or $\delta = \delta_3 < \delta_1$, the adjustment produces at least one new equality edge, allowing the search for an augmenting path to resume.

Let's look at an example of the algorithm in action. In Figure 2 we see

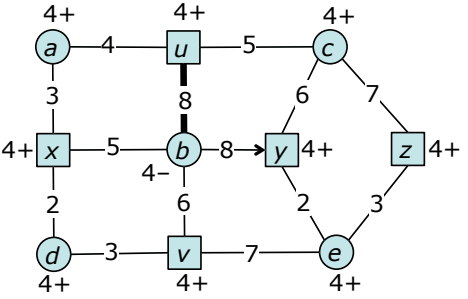

Figure 2: Getting started

the state of the algorithm after the first two steps. The initialization set all vertex labels equal to 4. Then, the first step discovered the augmenting path

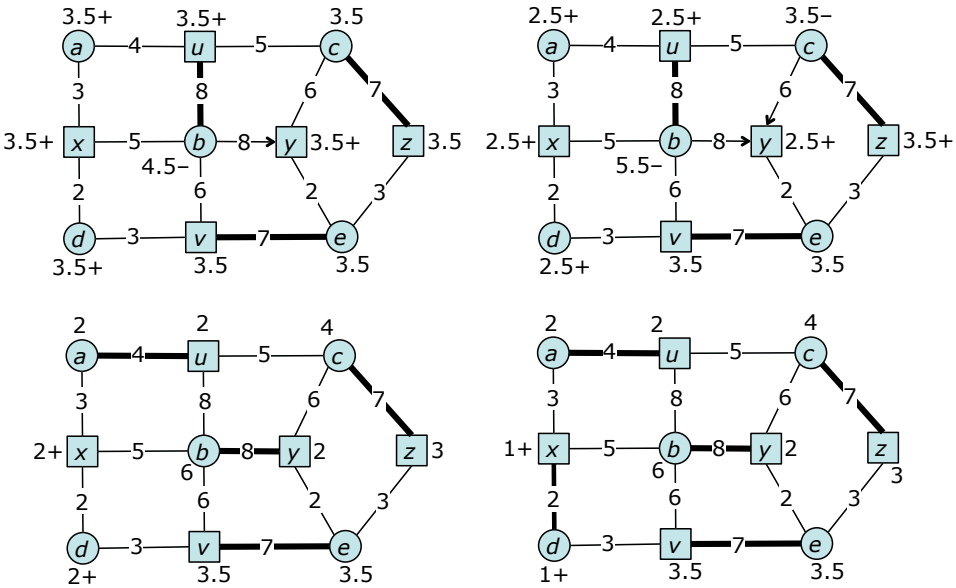

Figure 3: Example of labeling algorithm

from b to u and added $\{b, u\}$ to the matching. The second step added the edges $\{b, y\}$ and $\{b, u\}$ to the tree rooted at y. Note that these are the only equality edges at this point, so after the second step, the algorithm relabels vertices. In this case, $\delta = 0.5$ and edges $\{c, z\}$ and $\{e, v\}$ become equality edges. These are also augmenting paths, so the algorithm finds them and augments the matching. It then rebuilds the tree consisting of edges $\{b, u\}$ and $\{b, v\}$ giving the state shown in the top left of Figure 3.

At this point, we must again pause to relabel the vertices. In this case, $\delta = 1$, making $\{c, y\}$ an equality edge, allowing the tree to expanded further, giving us the state shown in top right of the figure. Once more, we must relabel the vertices, this time using $\delta = 0.5$. This, makes $\{a, u\}$ an equality edge and leads to the augmenting path $auby$ and the state shown in the bottom left of the figure. Two more steps produces the final matching shown in the bottom right of the figure.

Now let's consider how to implement this most efficiently. We can use heaps to quickly determine the δ_i values when a relabeling is required.

- Let $h_{1,e}$ be a heap containing all the even vertices with $z(u)$ as the key for vertex u. We'll maintain a second heap $h_{1,o}$ for the odd vertices.

- Let h_2 be a heap containing all edges e with one even and one un-

reached endpoint, and let $key(e) = z_e - w(e)$.

- Let h_3 be a heap containing all edges e with two even endpoints, and let $key(e) = z_e - w(e)$.

We can also use these heaps to update the labels efficiently if we use a heap with an efficient *addtokeys* operation. This operation adds a value x to the keys of all items in a heap and allows the labels to be adjusted implicitly. When we remove a vertex u from $h_{1,e}$ or $h_{1,o}$, its key value is used to update the label z_u, but while u is in either of the heaps, its label value is represented implicitly by the key of the heap item. We'll explain later how to extend the d-heap to implement *addtokeys* in constant time. This makes each label adjustment a constant time operation.

Now, there are at most $n/2$ augmenting path searches and within each search, all but the last step either extends the tree or performs a relabeling. When extending the tree, we make one unreached vertex odd and another one even. These must also be added to either $h_{1,o}$ or $h_{1,e}$. In addition, the edges incident to the new even vertex must be examined, and added to h_2 or h_3 as appropriate (we may have to remove an edge from h_2 here also). Since no vertex becomes even more than once in an augmenting path search, these heap operations contribute $O(m \log n)$ to the time for each path search and $O(mn \log n)$ altogether.

The time for each relabeling operation is $O(1)$. During any single augmenting path search, an edge can become an equality edge at most one time. Since all but the last relabeling operation creates at least one new equality edge, the number of relabeling operations per augmenting path search is at most m and so the run time for all relabeling operations is $O(mn)$. The remaining operations and hence the overall run time is $O(mn \log n)$.

What remains is to explain how the d-heap can be extended to implement *addtokeys* in constant time. This can be done by adding a single variable Δ to the heap. Each *addtokeys*(x) adds x to Δ. When an item is inserted into the heap with a key of k, we substitute $k - \Delta$ as the internal key value stored in the heap. Subsequent *addtokeys* operation do not change this stored key value, just the value of Δ. To recover the "true" value of a heap item's key, we simply add the current value of Δ to the stored key.