Weighted Matchings in General Graphs

Jonathan Turner

March 28, 2013

In the previous section we saw how we could use LP duality theory to develop an algorithm for the weighted matching problem in bipartite graphs. In this section, we'll see how to extend that algorithm to handle general graphs. As in the unweighted case, blossom-shrinking plays a central role. However, in weighted graphs we will handle blossoms a bit differently. In particular, we will maintain blossoms across multiple augmenting path searches.

Let's start by reviewing the matching ILP and its LP relaxation.

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

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. As noted in the previous section, we 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. This property does not hold for general graphs, but if we add some additional constraints to the LP relaxation, we get a linear program that has optimal solutions that correspond to the optimal solutions of the original ILP.

The new contraints take the form

$$\sum_{e \subseteq B} x_e \leq k_B$$

where B is a set of $2k_B + 1$ vertices and $k_B \geq 1$. The LP contains one such constraint for every odd subset B with at least three vertices. Note that although the number of constraints is now exponential in the size of the graph, since we will not be solving the LP directly, this need not interfere with our ability to obtain an efficient algorithm.

We can put this LP in standard matrix form by defining a matrix $G = [g_{B,e}]$ with a row for every odd subset B of the vertices (including those with a single vertex); $g_{B,e} = 1$ if B is a singleton set containing one of e's endpoints or $e \subseteq B$; otherwise $g_{B,e} = 0$. Let $W = [w_e]$ be a column vector containing the edge weights and $X = [x_e]$ be a column vector containing the LP variables. Then, let $K = [k_B]$ be a column vector with an entry for each odd subset of B, where $k_B = 1$ if B is a singleton and $2k_B + 1 = |B|$ if B has more than one vertex. Notice that if B defines an odd blossom, then k_B is the number of matching edges joining vertices in B. With these definitions, the LP becomes maximize $W^T X$ subject to $GX \leq K$.

The dual of this LP is minimize $K^T Z$ subject to $G^T Z \geq W$. Here, the vector $Z = [z_B]$ contains a dual variable for every odd subset. The objective function can be expanded to $\sum_B k_B z_B$ and the constraints can be written $z_e \geq w(e)$. For $e = \{u, v\}$, z_e is defined as $z_u + z_v$ plus the sum of all z_B for which $e \subseteq B$.

The complementary slackness conditions for this pair of problems can be written

$$(G^T Z^* - W)^T X^* = [0] \quad \text{and} \quad (K - GX^*)^T Z^* = [0]$$

where X^* and Z^* are optimal solutions. The first condition means that for every edge in the matching, $z_e = w(e)$. The second means that for every free vertex u, $z_u = 0$ and that for every B with more than one vertex, the number of matching edges connecting vertices in B is k_B. Our weighted matching algorithm finds solutions to the primal and dual that satisfy these condtions, and then uses the complementary slackness property to conclude that the matching defined by the primal variables has maximum weight. The following theorem captures these observations.

Theorem 1 *Let $G = (V, E)$ be a graph with edge weights $w(e)$, let M be a matching in G and let each odd subset B of V have a non-negative label z_B. If*

(1)	$z_e \geq w(e)$	*for all e*
(2)	$z_e = w(e)$	*for $e \in M$*
(3)	$z_B = 0$	*if B is a free vertex or includes $< k_B$ matching edges*

then M is a maximum weight matching.

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

$$\begin{aligned}
\sum_{e\in N} w(e) &\leq \sum_{e\in N} z_e \\
&= \sum_{\{u,v\}\in N} (z_u + z_v) + \sum_{e\in N} \sum_{B:e\subseteq B} z_B \\
&= \sum_{\{u,v\}\in N} (z_u + z_v) + \sum_{B:|B|>1} z_B \left(\begin{array}{c} \text{\# of edges in } N \text{ with} \\ \text{both endpoints in } B \end{array} \right) \\
&\leq \sum_{u\in V} z_u + \sum_{B:|B|>1} z_B k_B
\end{aligned}$$

The first inequality follows from condition (1) in the theorem; the second line follows from the definition of z_e and the third is effectively just reversing the order of the summations. The final inequality follows from the fact that N is a matching and the fact that an odd subset B can contain at most k_B edges in a matching. Continuing, this last expression is

$$\begin{aligned}
&= \sum_{\{u,v\}\in M} (z_u + z_v) + \sum_{B:|B|>1} z_B \left(\begin{array}{c} \text{\# of edges in } M \text{ with} \\ \text{both endpoints in } B \end{array} \right) \\
&= \sum_{e\in M} z_e = \sum_{e\in M} w(e)
\end{aligned}$$

Here, the first equality follows from condition (3). The second follows from the definition of z_e and the final equality follows from condition (2).

As in the bipartite case, an *equality edge* is defined as one for which $z_e = w(e)$, but here we use the more general definition of z_e given above. If we define $z_B = 0$ for any odd subset B with three or more vertices that *does not correspond to a blossom*, we can show that augmenting paths constructed from equality edges have maximum weight. To see why this is true, let $p = u_0, \ldots, u_{2k+1}$ be an augmenting path that does not pass through any blossom. Then, conditions (1) and (2) in the theorem imply that the weight of p satisfies the following.

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

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

Now consider how this argument changes if p does pass through a blossom B with label z_B. The edges in p that are inside the blossom form an even-length sub-path that alternates between edges that are in the matching and edges that are not. If the length of this sub-path be $2r$, we have the following.

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

This argument extends directly to multiple blossoms and nested blossoms. Also, note that if p is constructed entirely from equality edges, its weight is exactly equal to $z_{u_0} + z_{u_{2k+1}}$, as in the bipartite case. 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.

We will search for augmenting paths by building a set of trees using only the equality edges. The labels are chosen to satisfy conditions (1) and (2), and for odd subsets B with at least three vertices, z_B will be non-zero only if B defines a blossom. Note that this means that the only violations to condition (3) in the theorem arise from free vertices u with non-zero z_u.

The algorithm initializes $M = \{\}$ and sets z_u to half the largest edge weight for all vertices u. This satisfies conditions (1) and (2) in the theorem. As it proceeds, the algorithm maintains the validity of (1) and (2) while eliminating violations of (3).

Whenever an augmenting search stalls for lack of eligible equality edges, it adjusts the labels. If this eliminates all violations of condition (3), the algorithm terminates. If it produces additional equality edges, the algorithm resumes its search. For general graphs, there is also a third possibility that we will discuss shortly.

Whenever an augmenting path is found, the edges on the path are flipped as usual, but unlike the bipartite case, the *blossoms B with $z_B > 0$ are preserved* along with their labels. This is illustrated in Figure 1. In this

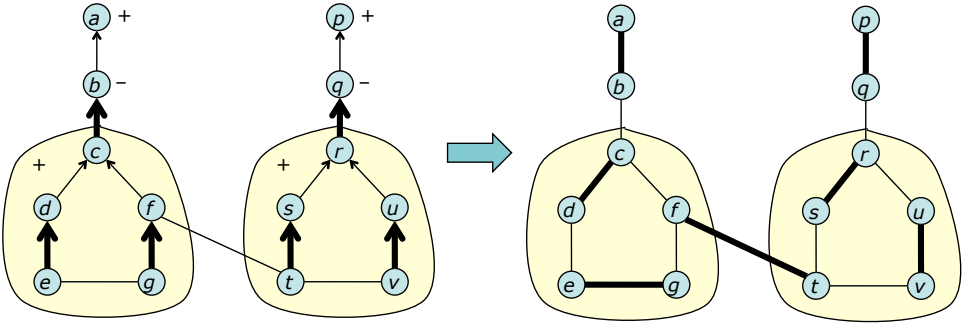

Figure 1: Augmenting matching while preserving blossoms

example, notice that after the edges on the augmenting path are flipped, f is the base of the blossom on the left, while t is the base of the blossom on the right.

Because blossoms may be preserved at the end of a path search, each new search may start with a number of previously-formed blossoms. The path search will operate over the condensed graph that includes these blossoms. Consequently, as blossoms are added to the trees constructed by the path search, they may be either odd or even, unlike the bipartite case, where blossoms are always even. For a vertex contained in a blossom, we say that the vertex is odd, even or unreached, if the outer-most blossom containing the vertex is.

When a label adjustment is needed, we compute a value δ and adjust the labels as follows. For each vertex u, we subtract δ from z_u if u is even, and we add δ to z_u if u is odd. For each top-level blossom B, we add 2δ to z_B if B is even and subtract 2δ if z_B is odd. Notice that for any edge $e = \{u, v\}$ contained within a blossom, this leaves z_e unchanged, since the changes to z_u and z_v are balanced by the changes to the label of the top-level blossom containing e. Similarly, if e is a tree edge that is not contained within any blossom, z_e is unchanged since one of e's endpoints is odd and the other is even. Finally, if both endpoints are unreached z_e is also unchanged. If e is not contained in a blossom and has an even endpoint that is not balanced by an odd endpoint, z_e may change and we need to take care to ensure that the label adjustment does not lead to a violation of condition (1).

We compute $\delta = \min\{\delta_1, \delta_2, \delta_3, \delta_4\}$ where the δ_i are defined as follows.

$$
\begin{aligned}
\delta_1 &= \min\{z_u \mid u \text{ is even}\} \\
\delta_2 &= \min\{z_e - w(e) \mid e = \{u, v\},\ u \text{ is even},\ v \text{ is unreached}\} \\
\delta_3 &= \min\left\{(z_e - w(e))/2 \;\middle|\; \begin{array}{l} e = \{u, v\},\ u,\ v \text{ are both even and} \\ e \text{ is not contained in any blossom} \end{array}\right\} \\
\delta_4 &= \min\{z_B/2 \mid B \text{ is a top-level odd blossom}\}
\end{aligned}
$$

Whenever any of the δ_i are undefined, they are simply omitted from the definintion of δ. Note that while the algorithm is running, δ_1 is always is defined, so δ is also.

Now, observe that the label adjustments maintain the validity of condition (1) and ensure that all labels remain non-negative. Also note 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.

If $\delta = \delta_4 < \delta_1, \delta_2, \delta_3$ then the label adjustment makes $z_B = 0$ for some odd blossom B. The algorithm then expands all such odd blossoms. If this does not produce any new equality edges, another label adjustment is performed. Note that expanding such a blossom B cannot lead to any violations of condition (1).

When an odd blossom is expanded, the algorithm must update the state variables and parent pointers for vertices and sub-blossoms contained within the blossom, as illustrated in Figure 2. Notice that this will typically make

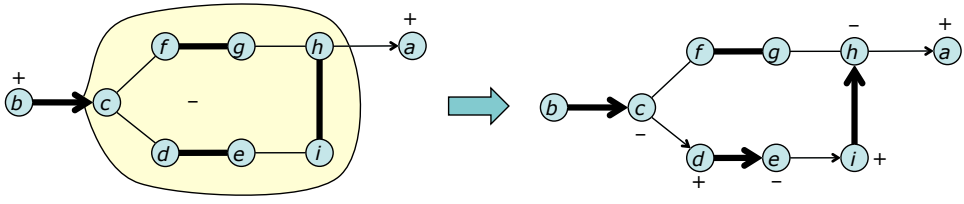

Figure 2: Expanding an odd blossom

some blossoms (and the vertices they contain) unreached. Such blossoms may be added to a tree at some later stage in the some augmenting path search as odd blossoms. Consequently, a vertex may alternate between being odd and unmatched multiple times during the course of a single path search. This behavior requires more complicated data structures than are needed for the bipartite case. We'll discuss the implications of this below.

Before we do that, however, let's look at the number of basic steps needed to complete one augmenting path search.

- At most $n/2$ steps expand a tree. Each such step may make one unreached blossom (or vertex) odd and one unreached blossom even. The edges incident to vertices in the even blossom must all be examined, but since a blossom only makes the transition from unreached to even once per augmenting path search, the number of edges that are examined during one augmenting path search is $O(m)$.

- At most $n/2$ steps form new blossoms. This is true because every new blossom formed is even, and even blossoms are not expanded during a search. In addition, the total length of these blossoms is at most $3n/2$. During the formation of the blossom, the edges incident to its odd sub-blossoms must be examined, but since each odd blossom joins an even blossom at most once in a path search, the number of edges examined during one augmenting path search is $O(m)$.

- No edge becomes an equality edge more than once, and consequently the number of label adjustments that add equality edges is at most m.

- Any odd blossom that gets expanded must have originally been formed before the start of the current path search, and consequently the number of steps that expand odd blossoms is at most $n/2$ and the total length of their blossom cycles is at most $3n/2$.

A straightforward implementation of Edmond's algorithm takes $O(mn^2)$ time. This can be improved to $O(mn \log n)$ with the help of some additional data structures. In the remainder of this section we give an overview of the required data structures, while omitting some details.

The first data structure that we need is a *blossom structure forest*. This is a forest with a node for every vertex and blossom (including sub-blossoms) in the current graph. For each vertex or blossom that is contained in another blossom, its parent in the tree is the node for the inner-most blossom containing it. Vertices and blossoms that are not contained within a blossom correspond to tree roots in the forest. The tree is implemented using doubly-linked circular sibling lists connecting all nodes with a common parent. Each tree node has a child pointer, to one of the nodes in its list of children (if any). Tree roots are also joined in a circular *root list*. The blossom structure tree is also used to define a total order on the vertices and blossoms/sub-blossoms. This order is simply the order in which the nodes in

the blossom-structure forest would be visited in a post-order traversal (with different trees visited in the order defined by the root list).

To determine the top-level blossom that contains a vertex u, we could simply go up to the root of the tree in the blossom-structure forest that contains u. Unfortunately, this can take $\Omega(n)$ time, which is too slow to support the most efficient implementation. We can speed this up by adding another data structure that implements *split-join sets*. This is a data structure that defines a set of disjoint sets on a base set of elements, where the elements are ordered relative to one another. This data structure supports three operations.

- *split*(s, j) divides a set s into two subsets, one containing items that are $\leq j$ (using the underlying order of the elements) and the other containing items that are $> j$.
- *join*(s_1, s_2) joins two sets where the elements of s_1 are all strictly less than the elements in s_2.
- *find*(j) returns the id of the set containing item j.

The *join* and *split* operations could be implemented using a collection of doubly-linked circular lists. However, this does not enable an efficient implmentation of the *find* operation. To make all three efficient, the data structure can be implemented as a collection of binary search trees, where the left-right order of nodes in the search tree corresponds to the underlying order of the elements. With this representation, all three operations can be implemented to run in $O(\log n)$ time, where n is the number of elements in the underlying set. This data structure can be used in conjunction with the blossom-structure forest to allow us to quickly determine the top-level blossom containing a vertex. As blossoms are formed and expanded, the data structure is modified to reflect the changes in the set of blossoms.

In the previous section, we described an extension to the d-heap data structure that supports the *addtokeys* operation, and showed how it could be used to speed up the computation of δ and the updating of vertex and blossom labels, for the version of the algorithm that handles bipartite graphs. Unfortunately, it is not sufficient to handle the case of general graphs. We finish with a high level summary of a general heap data structure that can be used for this purpose.

A *group heap* allows the heap elements to be divided into *groups*, where each group can be *active* or *inactive*. The *addtokeys* operation changes the key values of the elements that are in active groups, while leaving the keys

of the other elements unchanged. The *findmin* operation returns the active item with the smallest key. The group heap also supports a group-splitting operation, similar to the split operation in the split-join sets data structure. To support this operation, it requires all the heap elements to be totally-ordered, independent of their key values.

One way in which thie group heap is used is to keep track of the labels for the odd vertices and the unreached vertices. There is a group corresponding to each top-level blossom that is either odd or unreached. The groups for the odd blossoms are active, while the others are inactive. Each vertex belongs to the group for its top-level blossom (each vertex not contained in a blossom has its own group). This group heap allows us to efficiently update the label z_u for each odd vertex u using the *addtokeys* operation. When an unreached blossom becomes odd, we simply make its group active. When expanding an odd blossom, we also need to split its group into sub-groups corresponding to the sub-blossoms. The split operation in the group heap allows us to do this.