CS 6200: Algorithmics II	**Fall 2020**
Instructor: Avah Banerjee, S&T.	
Lecture: 14	**Date: 9/25**

1 Matchings

Reviewing a bit from the last lecture, we established that given a graph $G(V, E)$, with $u \subseteq V$ and u being independent, then this set system is defined as a matroid. In this lecture we will look at how to compute matchings, and how these matchings are closely related to max-flows. First, we will look at the simple example of bipartite matching. We can break down matchings of all kinds into two cases, the first of which is where all the edges of the graph are unweighted.

1.1 Unweighted Case

The goal in this case of bipartite matching of unweighted graphs is is to maximize the size of the matching. Looking at the following example in figure 1, we know that a matching consists of an edge between one vertex in L and one vertex in R. The question we want to answer is how many matchings do we have in a maximal set. In this example, we have a graph where a *perfect matching* exists (where every vertex is matched with another vertex). We see this matching highlighted below:

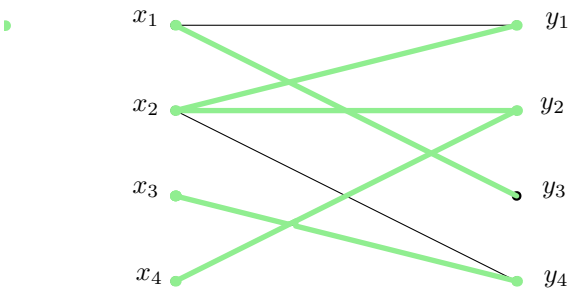

Fig. 1: Perfect Matching of a Bipartite Graph

While this is a good example of perfect matching, this is not the main focus of this lecture. Rather, we are looking to maximize the size of the matching (as described earlier), even if the matching isn't perfect. Of course, in the example in figure 1, the maximum size matching also correlates to a perfect matching

1.2 Weighted Case

The main difference with a weighted graph is that our goal is to find the maximum weight matching, instead of the maximum size matching. We will not be focusing on this very much in general, as the majority of our time will be spent looking at the unweighted case. However, an example of a weighted graph is given below in figure 2.

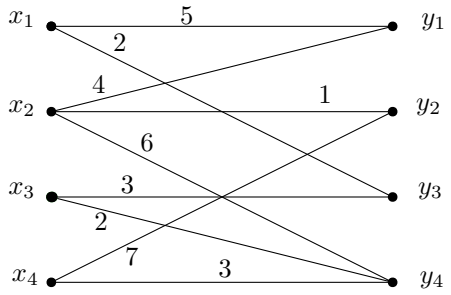

Fig. 2: Weighted Bipartite Graph

Remark 1.1 *Consider an example where we want to find the minimal weight matching, while also maximizing the size of the matching. How would we go about doing this?*

2 Matching Algorithms

In this section we will look at two different algorithms for the bipartite graph situation, one for the unweighted case and one for the weighted case.

2.1 The Unweighted Case

We will give the following example of an unweighted graph, but this time adding a source (s) and a sink (t), which essentially reduces the problem to maximum flow. Every edge on the left side of the graph is attached with edges coming from the source, while every edge on the right has an edge going from it to the sink. These edges are all directed, with a general flow going from the left to the right, or the source to the sink (this can all be seen in figure 3). All of the edges have unit capacity. It is known that since all the capacities are integral, then we can always get a maximum flow with integral capacity. And since every edge in this example has unit capacity, each edge will either be fully saturated or not used at all.

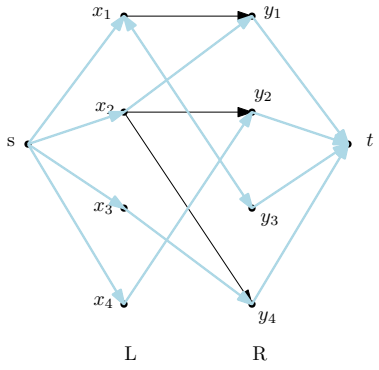

Fig. 3: Unweighted Bipartite Matching with Source and Sink (All edges have unit capacity)

We see in figure 3 that it is possible to saturate all of the edges coming from the source/going into the sink, giving us the size of the matching as 4.

2.2 The Weighted Case

We now want to find the matching with maximum weight. This problem can be reduced the minimum cost/maxflow problem. In this example, each edge from the source and every edge going into the sink have a cost of 0. For the rest of the edges, whatever the weight of the edge was, we will turn in the cost into the negative of that value. For example, if the weight of one edge was 5, the cost in our modified graph will be -5. It is also established that every edge in this graph has a unique capacity. This means that the integral flow f will equal the size of the matching $|f|$, with a weight of $-cost(f)$. Thus, this means we can use the our minimum cost/maxflow algorithm we have covered already to solve our new problem of maximizing the weight of this bipartite graph.

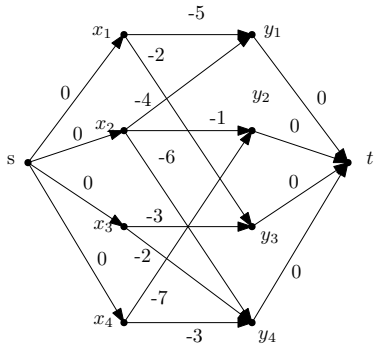

Fig. 4: Weighted Bipartite Matching Network

Normally, finding a polynomial-time algorithm to solve this problem for generic graphs is difficult. But in the special case of bipartite graphs, we do have one that we are able to use to find the maximum weight. Let's take the example we see in figure 4. Currently, the flow of this network is 0, and we can see that we also currently do not have any negative cost cycles, which is what we are looking for in this algorithm. So our goal is to find a augmenting path along the graph which has the minimum cost. Of course we want to keep adding to our path with the minimum cost until we can no longer augment it any more (and still hold the required properties).

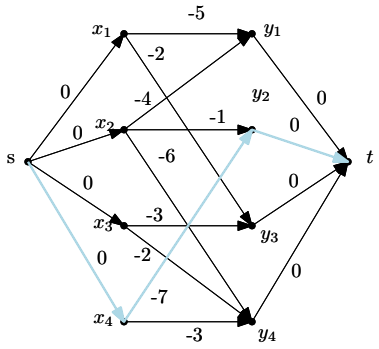

Fig. 5: Weighted Bipartite Matching Algorithm Example - Part 1

We start our augmenting path by picking the initial one with the lowest value, in this case $s \to x_4 \to y_2 \to t$ (seen above in figure 5). After picking this path, in our updated graph we will

remove those edges and replace them with edges in the opposite direction (and switch the cost of the edge between x_4 and y_2 to 7 instead of -7) (seen below in figure 6). We still do not have a cycle of negative cost, so we continue the algorithm. Thus we would continue doing this, find the edges that have the minimum cost.

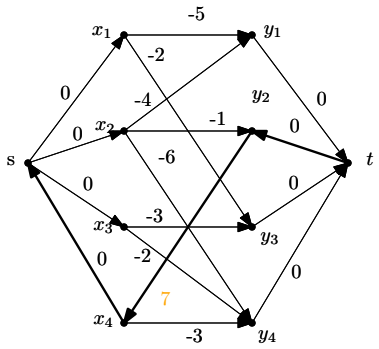

Fig. 6: Weighted Bipartite Matching Algorithm Example - Part 2

This algorithm can be summarized into two basic lines such as the following: As long as G has no negative cost cycles then we:

1. Start with zero flow
2. Find an augmenting path of mincost

To prove that this algorithm is true, we can set up a basic proof by contradiction.

Proof: If f is a flow with minimum cost and p is an augmented path with minimum cost, then suppose $f' = f \uparrow p$ is not a minimum cost flow. Then their must be some negative cost cycle in G_f. We define our cycle as c and we can say that $p \oplus c$ represents all edges in p or c but not in both. Thus we know that:

$$Cost(p \oplus c) = Cost(p) + Cost(c) \tag{1}$$

Since c is a negative cost cycle, the quantity is negative (less then zero). Thus $Cost(p \oplus c) < Cost(p)$ and $p \oplus c$ will be an augmenting path. This represents a contradiction as path p was supposed to the path of minimum cost, but now we have found an augmenting path that has a smaller cost then that of p. ■

This two-step algorithm that we laid out above has a time complexity of $O(n(m+nlgn))$, which is strongly polynomial of course.