CHAPTER

TWELVE

Combinatorial Applications of Network Flows

12.1 Introduction

Before delving into the particular combinatorial problems we wish to consider in this chapter, we will state a key theorem. When working with network flow problems, our examples thus far have always had integer capacities and we always found a maximum flow in which every edge carried an integer amount of flow. It is not, however, immediately obvious that this can always be done. Why, for example, could it not be the case that the maximum flow in a particularly pathalogical network with integer capacities is $23/3$? Or how about something even worse, such as $\sqrt{21\pi}$? We can rule out the latter because network flow problems fall into a larger class of problems known as linear programming problems, and a major theorem tells us that if a linear program is posed with all integer constraints (capacities in our case), the solution must be a rational number. However, in the case of network flows, something even stronger is true.

Theorem 12.1. *In a network flow problem in which every edge has integer capacity, there is a maximum flow in which every edge carries an integer amount of flow.*

Notice that the above theorem does not guarantee that every maximum flow has integer capacity on every edge, just that we are able to find one. With this theorem in hand, we now see that if we consider network flow problems in which the capacities are all 1 we can find a maximum flow in which every edge carries a flow of either 0 or 1. This can give us a combinatorial interpretation of the flow, in a sense using the full edges as edges that we "take" in some useful sense.

12.2 Matchings in Bipartite Graphs

Recall that a bipartite graph $\mathbf{G} = (V, E)$ is one in which the vertices can be properly colored using only two colors. It is clear that such a coloring then partitions V into two independent sets V_1 and V_2, and so all the edges are between V_1 and V_2. Bipartite graphs have many useful applications, particularly when we have two distinct types of objects and a relationship that makes sense only between objects of distinct types. For example, suppose that you have a set of workers and a set of jobs for the workers to do. We can consider the workers as the set V_1 and the jobs as V_2 and add an edge from worker $w \in V_1$ to job $j \in V_2$ if and only if w is qualified to do j.

For example, the graph in Figure 12.1 is a bipartite graph in which we've drawn V_1 on the left and V_2 on the right.

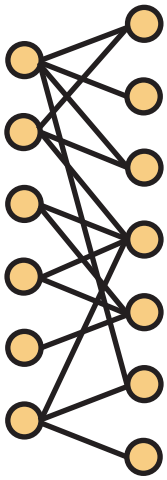

FIGURE 12.1: A BIPARTITE GRAPH

By a *matching* M in a graph $\mathbf{G}$, we mean that M is a subset of edges such that no two edges of M share an endpoint. When $\mathbf{G}$ is bipartite, a matching is thus a set of pairings of an element of V_1 with an element of V_2. We're usually interested in finding a *maximum matching*, which is a matching that contains the largest number of edges possible, and in bipartite graphs we usually fix the sets V_1 and V_2 and seek a maximum matching from V_1 to V_2. In our workers and jobs example, the matching problem thus becomes trying to find an assignment of workers to jobs such that each worker is assigned to a job for which he is qualified (meaning there's an edge), each worker is assigned to at most one job, and each job is assigned at most one worker. Clearly for there to be any hope of this to happen, there must be at least as many jobs as there are workers. In fact, there is a general theorem due to Hall that says that this is almost enough to

guarantee a matching that includes all the workers. (The theorem actually requires that if we restrict our attention to any subset of workers, there must be at least as many jobs for which they are collectively qualified as there are workers in the subset.)

As an example, in Figure 12.2, the red edges form a matching from V_1 to V_2. Suppose that you're the manager of these workers (on the left) and must assign them to the jobs (on the right). Are you really making the best use of your resources by only putting four of six workers to work? How can you either find a more efficient assignment or else justify to your boss that there's no better assignment of workers to jobs?

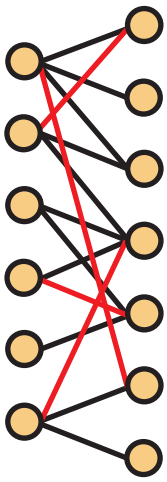

FIGURE 12.2: A MATCHING IN A BIPARTITE GRAPH

How could we go about finding an optimal assignment of workers to jobs in our problem? The approach we will consider makes use of a network flow. This algorithm, while decent, is not the most efficient algorithm for the problem, so it is not likely to be the one used in practice. However, it is a nice example of how network flows can be used to solve a combinatorial problem. The network that we use is formed from a bipartite graph **G** by placing an edge from the source S to each vertex of V_1 and an edge from each vertex of V_2 to the sink T. The edges between V_1 and V_2 are oriented from V_1 to V_2, and *all* edges are given capacity 1. Figure 12.3 contains the network corresponding to our graph from Figure 12.1. Edges in this network are all oriented from left to right and all edges have capacity 1. We can translate between a matching in **G** and a flow in the associated network simply by putting one unit of flow on the edges of the matching and running one unit of flow into those edges' ends in V_1 from S and routing one unit of flow out of those edges' ends in V_2 to T. Conversely, the edges between V_1 and V_2 that carry a flow of 1 in an integer-valued flow in the network are a matching in **G**. Thus, we can find a maximum matching from V_1 to V_2 by simply running the labelling algorithm

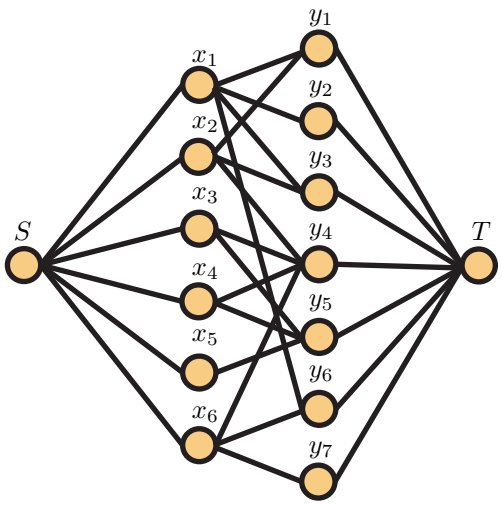

FIGURE 12.3: THE NETWORK CORRESPONDING TO A BIPARTITE GRAPH

on the associated network in order to find a maximum flow.

In Figure 12.4, we show the flow (using red edges to indicated edges that have flow 1 and black for flow 0 edges) corresponding to our guess at a matching from Figure 12.2. Now with priority sequence $S, T, x_1, x_2, \ldots, x_6, y_1, y_2, \ldots, y_7$, the labelling algorithm produces the labels shown below.

$$
\begin{array}{rl}
S & (*, +, \infty) \\
x_3 & (S, +, 1) \\
x_5 & (S, +, 1) \\
y_4 & (x_3, +, 1) \\
y_5 & (x_3, +, 1) \\
x_6 & (y_4, -, 1) \\
x_4 & (y_5, -, 1) \\
y_6 & (x_6, +, 1) \\
x_1 & (y_6, -, 1) \\
y_1 & (x_1, +, 1) \\
y_2 & (x_1, +, 1) \\
y_3 & (x_1, +, 1) \\
x_2 & (y_1, -, 1) \\
T & (y_2, +, 1)
\end{array}
$$

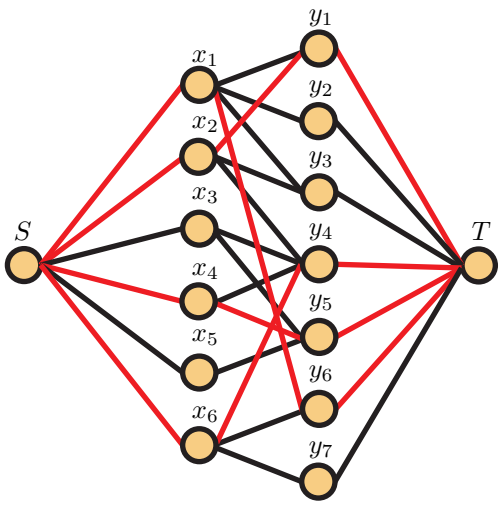

FIGURE 12.4: THE FLOW CORRESPONDING TO A MATCHING

This leads us to the augmenting path $S, x_3, y_4, x_6, y_6, x_1, y_2, T$, which gives us the flow shown in Figure 12.5. Is this a maximum flow? Another run of the labelling algorithm produces

$$\begin{array}{rl} S & (*, +, \infty) \\ x_5 & (S, +, 1) \\ y_5 & (x_5, +, 1) \\ x_4 & (y_5, -, 1) \\ y_4 & (x_4, +, 1) \\ x_3 & (y_4, -, 1) \end{array}$$

and then halts. Thus, the flow in Figure 12.5 is a maximum flow, and therefore, the corresponding matching is a maximum matching. Now you've made an improvement in how many workers are kept busy, and you can justify to your boss that you can't put all six workers to work.

12.3 Chain partitioning

In Chapter 4, we discussed Dilworth's theorem, which told us that for any poset $\mathbf{P}$, there is a partition of $\mathbf{P}$ into w chains, where w is the width of $\mathbf{P}$. However, we were only able to devise an algorithm to find this chain partition (and a maximum antichain) in the special case where $\mathbf{P}$ was an interval order. Now, through the magic of network flows, we

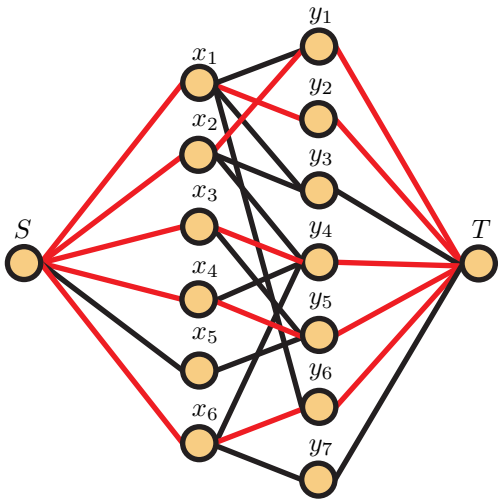

FIGURE 12.5: THE AUGMENTED FLOW

will be able to devise an efficient algorithm that works in general for all posets. However, to do so, we will require a slightly more complicated network than we devised in the previous section.

Suppose that the points of our poset $\mathbf{P}$ are $\{x_1, x_2, \ldots, x_n\}$. We construct a network from $\mathbf{P}$ consisting of the source S, sink T, and two points x_i' and x_i'' for each point x_i of $\mathbf{P}$. All edges in our network will have capacity 1. We add edges from S to x_i' for $1 \leq i \leq n$ and from x_i'' to T for $1 \leq i \leq n$. Of course, this network wouldn't be too useful, as it has no edges from the single-prime nodes to the double-prime nodes. To resolve this, we add an edge directed from x_i' to x_j'' if and only if $x_i < x_j$ in $\mathbf{P}$.

We can now turn on our labelling algorithm for this network, which will halt and give us a maximum flow in which all edges either carry one unit of flow or are empty. We also make a record of which vertices were labelled in the last run of the labelling algorithm, when it haled with the sink unlabelled, as this will be important, too. In the bipartite matching problem above, it was easy to tell how to turn a flow into a matching; here, however, it's definitely more challenging to see what's going on. We form the chains of our chain partition by putting points x_i and x_j in the same chain if $x_i'x_j''$ is full or $x_j'x_i''$ is full in the flow. You can think of how to read this off the network flow as starting at a node x_{i_1}', following any edge out of it that has flow on it to x_{i_2}'', then jumping back to x_{i_2}' and seeing if it has flow going out of it and continuing the process in this manner. When the single-prime vertex has no flow going out of it, you've reached the top of the chain. If, in building a chain, you reach a vertex that you've already built up from, the existing chain is extended by adding the new one on at the end.

Even once we see that the above process does in fact generate a chain partition, it is

not immediately clear that it's a minimum chain partition. For this, we need to find an antichain of as main points as there are chains in our partition. To do this, look at pairs of points (x_i', x_i'') and if the last run of the labelling algorithm (you did save the list of which vertices it labelled when it didn't reach the sink, right?) labelled the vertices or not. In each of the chains that we've built up, it cannot be the case that all of the vertices corresponding to the points in the chain are labelled or all of them are unlabelled, as otherwise there would be an augmenting path or we would have an invalid flow. Therefore, when traversing the chain, the vertices must switch from being labelled to being unlabelled. This will always happen by creating a pair (x_i', x_i'') in which one of the vertices is labelled and the other is unlabelled. The set of such vertices forms an antichain, and since there is one from each chain, we have that the antichain is maximum and the chain partition is minimum.

Example 12.2. Consider the poset in Figure 12.6. We'll discuss the points of the poset as

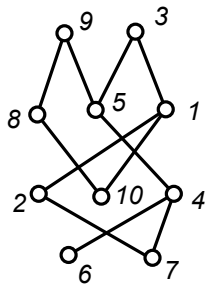

FIGURE 12.6: A PARTIALLY ORDERED SET

x_i where i is the number printed next to the point in the diagram. We wish to apply our algorithm to find a chain partition. First, we must construct the network, which is shown in Figure 12.7. In this network, all capacities are 1, edges are directed from left to right, the source is the vertex on the far left, the sink on the far right, the first column of ten vertices is the x_i' with x_1' at the top and x_{10}' at the bottom, and the second column of ten vertices is the x_i'' in increasing order of index.

As is the case with bipartite matching, running the labelling algorithm by hand on this from the zero flow would be tedious. Thus, we attempt to eyeball in a reasonably good flow, such as the one shown in Figure 12.8, where the red edges carry a flow of 1 and the black edges are empty. We now run the labelling algorithm to see if we can find an augmenting path or if we've actually got a maximum flow here. When we do so, we

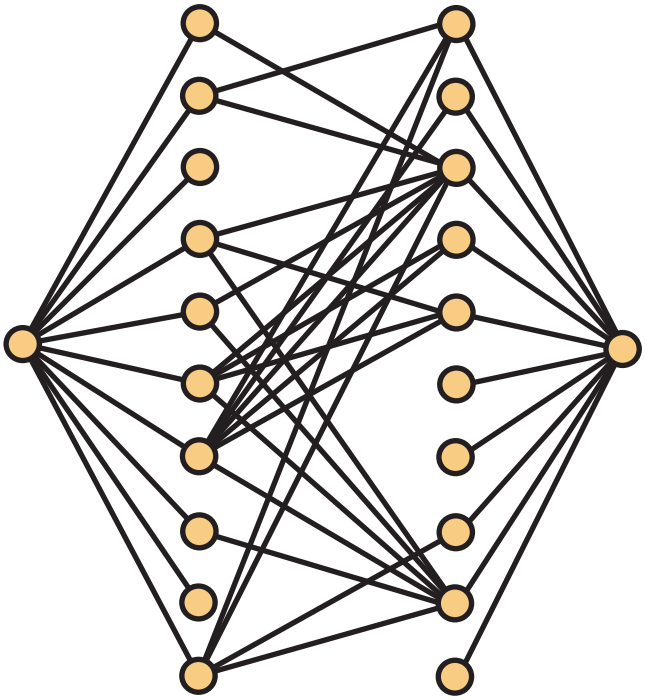

FIGURE 12.7: THE NETWORK OF THE POSET FROM FIGURE 12.6

get the labels below.

S	$(*,+,\infty)$	x_5''	$(x_6',+,1)$
x_3'	$(S,+,1)$	x_1'	$(x_3'',-,1)$
x_5'	$(S,+,1)$	x_8'	$(x_9'',-,1)$
x_6'	$(S,+,1)$	x_7'	$(x_4'',-,1)$
x_9'	$(S,+,1)$	x_3'	$(S,+,1)$
x_3''	$(x_5',+,1)$	x_1''	$(x_7',+,1)$
x_9''	$(x_5',+,1)$	x_2''	$(x_7',+,1)$
x_4''	$(x_6',+,1)$	x_2'	$(x_7',+,1)$
		T	$(x_2'',+,1)$

Thus, we find the augmenting path $S, x_6', x_4'', x_7', x_2'', T$, and the updated flow can be seen in Figure 12.9. Are we done? We run the labelling algorithm again and assign the labels below, with the algorithm halting with the sink unlabelled, so we do have a maximum

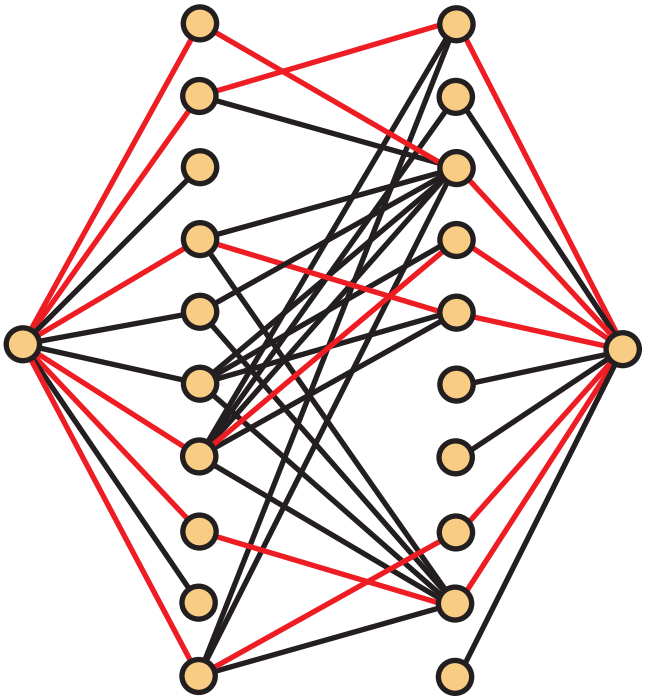

FIGURE 12.8: AN INITIAL FLOW

flow.

S	$(*,+,\infty)$	x_3''	$(x_5',+,1)$
x_3'	$(S,+,1)$	x_9''	$(x_5',+,1)$
x_5'	$(S,+,1)$	x_1'	$(x_3'',-,1)$
x_9'	$(S,+,1)$	x_8'	$(x_9'',-,1)$

This information is helpful, as it allows us to find our maximum antichain as well. In Figure 12.10, the green vertices are those labelled in the final run of the labelling algorithm and the yellow ones are unlabelled. Now from Figure 12.10, we can read off what we need, The chains of our chain partition are

$$C_1 = \{x_8, x_9, x_{10}\}$$
$$C_2 = \{x_1, x_2, x_3, x_7\}$$
$$C_3 = \{x_4, x_5, x_6\},$$

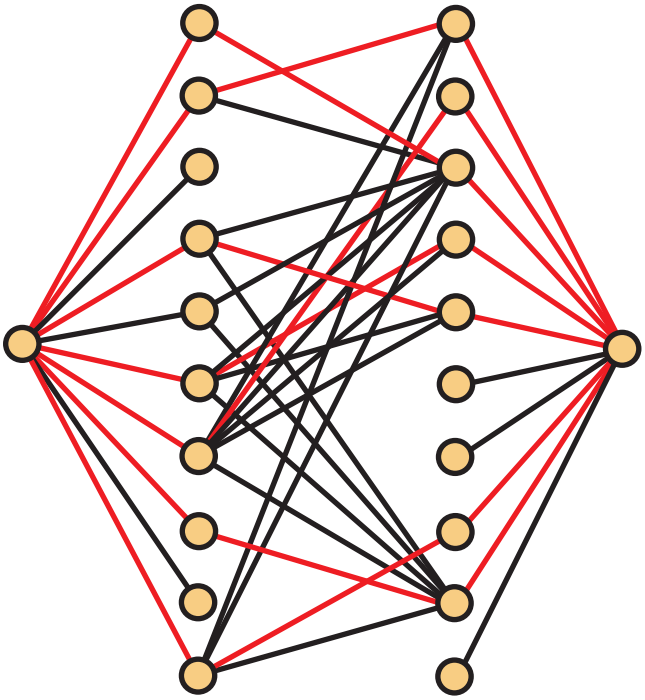

FIGURE 12.9: A BETTER FLOW

and a maximum antichain is $\{x_1, x_5, x_8\}$, the vertices which have x_i' labelled and x_i'' unlabelled.

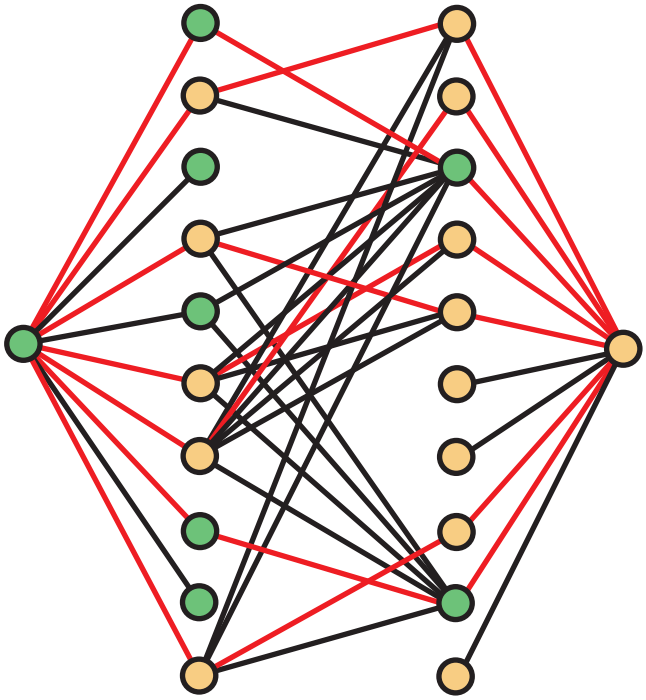

FIGURE 12.10: A MAXIMUM FLOW WITH LABELLED VERTICES IN GREEN

12