Notes 20 for CS 170

1 Duality

As it turns out, the max-flow min-cut theorem is a special case of a more general phenomenon called *duality*. Basically, duality means that a maximization and a minimization problem have the property that any feasible solution of the min problem is greater than or equal any feasible solution of the max problem (see Figure 1). Furthermore, and more importantly, *they have the same optimum*.

Consider the network shown in Figure 1 below, and the corresponding max-flow problem. We know that it can be written as a linear program as follows ($f_{xy} \geq 0$ in the last line is shorthand for all 5 inequalities like $f_{s,a} \geq 0$, etc.):

$$P: \begin{cases} \max f_{sa} + f_{sb} & & & & & & \\ f_{sa} & & & & & \leq & 3 \\ & f_{sb} & & & & \leq & 2 \\ & & f_{ab} & & & \leq & 1 \\ & & & f_{at} & & \leq & 1 \\ & & & & f_{bt} & \leq & 3 \\ f_{sa} & & -f_{ab} & -f_{at} & & = & 0 \\ & f_{sb} & +f_{ab} & & -f_{bt} & = & 0 \\ & & & & f_{xy} & \geq & 0 \end{cases}$$

Consider now the following linear program (where again $y_{xy} \geq 0$ is shorthand for all inequalities of that form):

$$D: \begin{cases} \min 3y_{sa} + 2y_{sb} + y_{ab} + y_{at} + 3y_{bt} & & & & & & & \\ y_{sa} & & & & & + u_a & & \geq & 1 \\ & y_{sb} & & & & & + u_b & \geq & 1 \\ & & y_{ab} & & & - u_a & + u_b & \geq & 0 \\ & & & y_{at} & & - u_a & & \geq & 0 \\ & & & & y_{bt} & & - u_b & \geq & 0 \\ & & & & & & y_{xy} & \geq & 0 \end{cases}$$

This LP describes the min-cut problem! To see why, suppose that the u_A variable is meant to be 1 if A is in the cut with S, and 0 otherwise, and similarly for u_B (naturally, by the definition of a cut, S will always be with S in the cut, and T will never be with S). Each of the y variables is to be 1 if the corresponding edge contributes to the cut capacity, and 0 otherwise. Then the constraints make sure that these variables behave exactly as

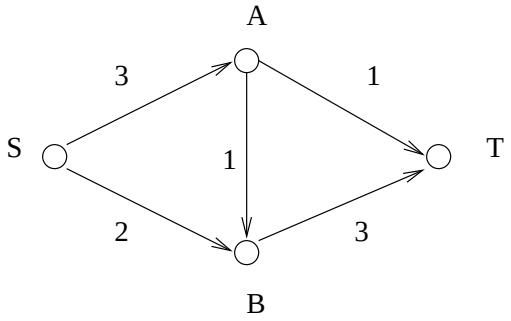

Figure 1: A simple max-flow problem.

they should. For example, the first constraint states that *if A is not with S, then SA must be added to the cut.* The third one states that *if A is with S and B is not* (this is the only case in which the sum $-u_A + u_B$ becomes -1), *then AB must contribute to the cut.* And so on. Although the y and u's are free to take values larger than one, they will be "slammed" by the minimization down to 1 or 0 (we will not prove this here).

Let us now make a remarkable observation: These two programs have strikingly symmetric, *dual*, structure. Each variable of P corresponds to a constraint of D, and vice-versa. Equality constraints correspond to unrestricted variables (the u's), and inequality constraints to restricted variables. Minimization becomes maximization. The matrices are transposes of one another, and the roles of right-hand side and objective function are interchanged. Such LP's are called *dual* to each other.

By the max-flow min-cut theorem, the two LP's P and D above have the same optimum. *In fact, this is true for general dual LP's!* This is the *duality theorem*, which can be stated as follows.

Theorem 1 (Duality Theorem)
Consider a primal *LP problem written in standard form as* "maximize $c * x$ subject to $A \cdot x \leq b$ and $x \geq 0$". *We define the* dual *problem to be* "minimize $b * y$ subject to $A^T y \geq c$ and $y \geq 0$". *Suppose the primal LP has a finite solution. Then so does the dual problem, and the two optimal solutions have the same cost.*

The theorem has an easy part and a hard part. It is easy to prove that for every feasible x for the primal and every feasible y for the dual we have $c * x \leq b * y$. This implies that the optimum of the primal is at most the optimum of the dual (this property is called *weak duality*. To prove that the costs are equal for the optimum is the hard part.

Let us see the proof of weak duality. Let x be feasible for the primal, and y be feasible for the dual. The constraint on the primal impose that $a_1 * x \leq b_1$, $a_2 * x \leq b_2$, ..., $a_m * x \leq b_m$, where $a_1, \ldots, a_m$ are the rows of A and $b_1, \ldots, b_m$ the entries of b.

Now, the cost of y is $y_1 b_1 + \cdots + y_m b_n$, which, by the above observations, is at least $y_1(a_1 * x) + \cdot + y_m(a_m * x)$, that we can rewrite as

$$\sum_{i=1}^{m} \sum_{j=1}^{n} a_{i,j} y_i x_j \ . \tag{1}$$

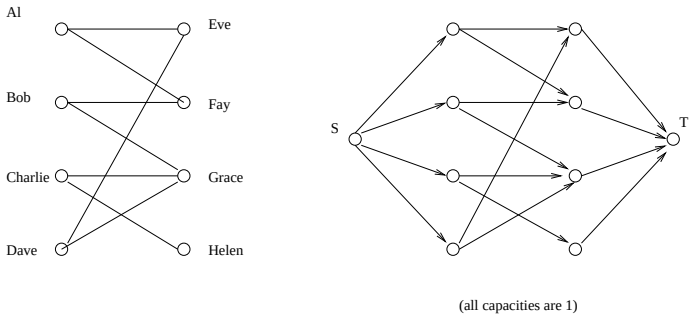

Figure 2: Reduction from matching to max-flow.

Each term $\sum_{i=1}^{m} a_{i,j} y_j$ is at least c_j, because this is one of the constraints of the dual, and so we have that Expression (1) is at least $\sum_i x_i c_i$, that is the cost of the primal solution.

2 Matching

2.1 Definitions

A *bipartite graph* is a (typically undirected) graph $G = (V, E)$ where the set of vertices can be partitioned into subsets V_1 and V_2 such that each edge has an endpoint in V_1 and an endpoint in V_2.

Often bipartite graphs represent relationships between different entities: clients/servers, people/projects, printers/files to print, senders/receivers ...

Often we will be given bipartite graphs in the form $G = (L, R, E)$, where L, R is already a partition of the vertices such that all edges go between L and R.

A *matching* in a graph is a set of edges that do not share any endpoint. In a bipartite graph a matching associates to some elements of L precisely one element of R (and vice versa). (So the matching can be seen as an *assignment*.)

We want to consider the following optimization problem: given a bipartite graph, find the matching with the largest number of edges. We will see how to solve this problem by reducing it to the maximum flow problem, and how to solve it directly.

2.2 Reduction to Maximum Flow

Let us first see the reduction to maximum flow on an example. Suppose that the *bipartite* graph shown in Figure 2 records the compatibility relation between four straight boys and four straight girls. We seek a maximum matching, that is, a set of edges that is as large as possible, and in which no two edges share a node. For example, in the figure below there is a *perfect* matching (a matching that involves all nodes).

To reduce this problem to max-flow we do this: We create a new source and a new sink, connect the source with all boys and all girls with the sinks, and direct all edges of the original bipartite graph from the boys to the girls. All edges have capacity one. It is easy to see that the maximum flow in this network corresponds to the maximum matching.

Well, the situation is slightly more complicated than was stated above: What is easy to see is that the optimum *integer-valued* flow corresponds to the optimum matching. We

would be at a loss interpreting as a matching a flow that ships .7 units along the edge Al-Eve! Fortunately, what the algorithm in the previous section establishes is that *if the capacities are integers, then the maximum flow is integer.* This is because we only deal with integers throughout the algorithm. Hence *integrality comes for free in the max-flow problem.*

In general, given a bipartite graph $G = (L, R, E)$, we will create a network $G' = (V, E')$ where $V' = L \cup R \cup \{s, t\}$, and E' contains all directed edges of the form (s, u), for $u \in L$, (v, t), for $v \in R$, and (u, v), for $\{u, v\} \in E$. All edges have capacity one.

2.3 Direct Algorithm

Let us now describe a direct algorithm, that is essentially the composition of Ford-Fulkerson with the above reduction.

The algorithm proceeds in phases, starting with an empty matching. At each phase, it either finds a way to get a bigger matching, or it gets convinced that it has constructed the largest possible matching.

We first need some definitions. Let $G = (L, R, E)$ be a bipartite graph, $M \subseteq E$ be a matching. A vertex is *covered* by M if it is the endpoint of one of the edges in E.

An *alternating path* (in G, with respect to M) is a path of odd length that starts with a non-covered vertex, ends with a non-covered vertex, and alternates between using edges not in M and edges in M.

If M is our current matching, and we find an alternating path with respect to M, then we can increase the size of M by discarding all the edges in of M which are in the path, and taking all the others.

Starting with the empty matching, in each phase, the algorithm looks for an alternating path with respect to the current matching. If it finds an alternating path, it updates, and improves, the current matching as described above. If no alternating path is found, the current matching is output.

It remains to prove the following:

1. Given a bipartite graph G and a matching M, an alternating path can be found, if it exists, in $O(|V| + |E|)$ time using a variation of BFS.

2. If there is no alternating path, then M is a maximum matching.

We leave part (1) as an exercise and prove part(2).

Suppose M is not an optimal matching, that is, some other matching M^* has more edges. We prove that M must have an alternating path.

Let $G = (L, R, E')$ be the graph where E' contains the edges that are either in M or in M^* *but not in both* (i.e. $E' = M \oplus M^*$).

Every vertex of G has degree at most two. Furthermore if the degree is two, then one of the two incident edges is coming from M and the other from M^*.

Since the maximum degree is 2, G is made out of paths and cycles. Furthermore the cycles are of even length and contain each one an equal number of edges from M and form M^*.

But since $|M^*| > |M|$, M^* must contribute more edges than M to G, and so there must be some path in G where there are more edges of M^* than of M. This is an augmenting path for M.

This completes the description and proof of correctness of the algorithm. Regarding the running time, notice that no matching can contain more than $|V|/2$ edges, and this is an upper bound to the number of phases of the algorithm. Each phase takes $O(|E| + |V|) = O(|E|)$ time (we can ignore vertices that have degree zero, and so we can assume that $|V| = O(|E|)$), so the total running time is $O(|V||E|)$.