

1 Zoo Tycoon

Algorithm: Construct a source s and a sink t . For each food j , create a vertex x_j , and add an edge from s to x_j with capacity T_j . For each animal i , add a vertex y_i , and for each food j not in S_i , add an edge from x_j to y_i with infinite capacity. Finally, for each animal i , add a vertex z_i . For each food $j \in S_i$, add an edge from x_j to z_i with infinite capacity. Additionally, add an edge from y_i to z_i with capacity D_i , and from z_i to t with capacity F_i . Now run a network flow algorithm on this flow network, and return true if and only if there is a flow of size $F = \sum_i F_i$.

Correctness: We will show that our algorithm returns true if and only if a feasible feeding plan exists.

First, assume there is a feasible feeding plan. Define p_j to be the amount of food j used by the plan, and p_{ij} to be the amount of food j fed to animal i . We will construct a flow of size F . To do so, for each food, route p_j units of flow on the edge (s, x_j) . For each animal i , for each food not in S_i , route p_{ij} units of flow from x_j to y_i . Then, for each animal, route $\sum_{j \notin S_i} p_{ij}$ units of flow from y_i to z_i . For each $j \in S_i$, route p_{ij} unit of flow from x_j to z_i . Finally, for each animal i , route F_i units of flow from z_i to t . We now verify that this is a flow of size F :

- Conservation constraints are satisfied: by construction. For any node other than s and t , it is easy to check that we routed exactly as much flow out as we routed in.
- Capacity constraints are satisfied:
 - Because the feeding plan didn't use more than T_j of any food j , the capacity constraints are satisfied on all edges from s to each x_j .
 - Because the feeding plan didn't feed any animal more than D_i total units of food not in S_i , the capacity constraints on each edge (y_i, z_i) are satisfied.
 - Because each animal consumes exactly F_i units of food, the capacity constraints are satisfied on all edges from each z_i to t .
- The size of the flow is F : this follows from the fact that each animal eats exactly F_i units of food, so the amount of flow going into t will be exactly F .

Conversely, assume there is a flow f of size F in our network (that is, assume our algorithm returns T). For each edge e from x_j to z_i or y_i , feed animal i f_e tons of food j . We now check that this is a feasible feeding plan:

- No more than T_j tons of each food j are used: the amount of food j used is the total flow out of x_j . By the conservation constraint, this is exactly the amount of flow routed on (t, x_j) . By the capacity constraint on that edge, this is at most T_j .
- No animal i is fed more than D_i total tons of food not in S_i : the amount of food not in S_i being fed to animal i is the total flow into y_i . By the conservation constraint, this is equal to the total flow on (y_i, z_i) . By the capacity constraints, this is at most D_i .

- Each animal i is being fed exactly F_i tons of food: the amount of food being fed to animal i is the total flow into y_i plus the total flow into z_i . By the conservation constraints, this is equal to the flow leaving z_i . By the capacity constraints, this is at most F_i . Consider the $s - t$ cut where the source side is just t . The capacity of this cut is F . Since the size of our flow is F , this cut must be saturated. But then every edge from a z_i to t must have flow F_i , implying that each animal is fed exactly F_i tons of food.

Runtime: The graph constructed by this algorithm has $O(n + m)$ vertices and $O(nm)$ edges - it therefore takes $O(nm)$ time to construct. We then call a network flow algorithm on this graph. The runtime will be $O(F(n + m, nm))$, where $F(x, y)$ is the runtime of your favorite network flow algorithm on a graph with x vertices and y edges.

2 Disrupting Shipping

Algorithm: Find the min cut in G . Destroy d edges crossing this cut, or all the edges, if there are fewer than d .

Correctness: If the algorithm completely disconnects s and t , we've clearly done the best we can. Otherwise, we delete d edges, and the graph is still connected.

Note that by the duality of Max Flow and Min Cut, the Max Flow of a graph can only be reduced by as much as you can reduce the size of the min cut. By cutting d edges with unit capacity, you can only decrease the weight of the min cut by d .

Our algorithm does exactly this. Let C be the cut computed by our algorithm. After cutting d edges from C , C must remain the min cut - other cuts have had their capacity reduced by at most d , and C 's capacity has been reduced by exactly d . It follows that the max flow is reduced by d as well. By the previous paragraph, this is optimal.

Runtime: The runtime of our algorithm is just the time required to compute a max flow on the input graph.

3 No Child Left Behind

- Algorithm:** Construct a flow graph from the matching graph in the standard way. [We will think of putting orphans on the source side, and mentors on the sink side.] On this graph, construct a flow corresponding to M , and let G_M be the residual graph from such a flow. Run the Ford-Fulkerson algorithm, starting from residual graph G_M (rather than the usual starting residual graph, which is just G). Once you have obtained the max flow f in this way, return the matching implied by f .

Correctness: We have already shown in class that there is a one-to-one correspondence between max flows and max matchings in the network used by our algorithm. In particular, the flow at the end of each iteration of Ford-Fulkerson can be thought of as a matching. We will show that none of these successive matchings un-matches orphans who are matched in M . We argue by induction on the number of iterations.

Base Case: Before we run the first iteration of Ford-Fulkerson, we have M .

Inductive Hypothesis: Let M_k be the matching after k iterations of the algorithm, and let G_k be the residual graph for that matching. Assume that all the matched orphans for M are still matched in M_k .

Inductive Case: Consider the $k + 1$ st iteration of Ford-Fulkerson. The algorithm finds an augmenting path in G_k . This path must start at s , and go to an orphan node. Because there are no edges between orphans, it must then go to a mentor node via a forward edge. It then either continues on to t , or goes back to the orphan size via a back edge. It then goes back to the mentors, and so on. In other words, an augmenting path starts at s , goes to an orphan, alternates orphan-mentor-orphan-mentor etc. before finally ending up at t . Every orphan-mentor edge in this path is a forward edge, and every mentor-orphan edge is a back edge.

We now show that every orphan i who was assigned after step k will remain assigned.

- * If i is not in the augmenting path, their mentorship relationship doesn't change - in particular, they stay assigned if they were assigned before.
- * Now assume i is in the augmenting path. Note that after the flow is augmented according to this path, i has one unit of out-flow, and one unit of in-flow. It follows that every orphan in the augmenting path is assigned.

Now note that the proof of correctness, which argues by finding a matching min-cut, holds regardless of the starting residual graph used. It follows that our algorithm in fact finds a max flow, and thus a maximum matching. This answers part (b) as well.

- b. See the end of the previous part. Note: given a bipartite graph with sets A and B , the sets of elements of A which can be simultaneously matched form a matroid, called a *transversal matroid*. The subset property is obvious, and the augmentation property follows from a similar argument to the one in part (a).