CS 577: Introduction to Algorithms — Network Flow Review Solutions

Network Flow

1. Let f be a valid flow in the network G, and let G_f be the residual network. Then we claim f is a maximum flow if and only if there is no path (along edges with positive residual capacity) from s to t in G_f. Given this claim, the linear time algorithm for determining whether f is maximal is as follows: construct G_f, and then use DFS (or BFS) to see if there is a path from s to t. If the path exists, f is not maximum; otherwise, f is maximum. The construction of G_f in linear time follows from the definition, and DFS runs in linear time, so the algorithm as a whole takes linear time.

2. Consider a $\max$ flow f and the residual network G_f. From the $\max$ flow $\min$ cut theorem it is easy to get that G_f doesn't contain any edge that goes from S to T. Neither does it contain any edges from S' to T'. Then it also has no edges going from $S \cup S'$ to $T \cap T'$. But if no edge from G_f crosses the cut $(S \cup S', T \cap T')$, then

$$c\left(S \cup S', T \cap T'\right) = f\left(S \cup S', T \cap T\right),$$

and using the $\min$ cut $\max$ flow theorem once again we see that cut $(S \cup S', T \cap T')$ is also minimal.

3. First compute a minimum cut C, and define its volume by $|C|$. Let $e_1, e_2, ..., e_k$ be the edges in C. For each e_i, try increasing the capacity of e_i by 1 and compute a minimum cut in the new graph. Let the new minimum cut be C_i , and denote its volume (in the new graph) as $|C_i|$. If $|C| = |C_i|$ for some i, then clearly C_i is also a minimum cut in the original graph and $C \neq C_i$, so the minimum cut is not unique. Conversely, if there is a different minimum cut C' in the original graph, there will be some $e_i \in C$ that is not in C', so increasing the capacity of that edge will not change the volume of C', thus $|C| = |C_i|$. In conclusion, the graph has a unique minimum cut if and only if $|C| < |C_i|$ for all i. The algorithm takes at most $m+1$ computing of minimum cuts, and therefore runs in polynomial time.

4. First we rule out some edges which can be easily seen to not be among the lower-binding edges. In this question we assume all of the capacities are nonzero and integral. Given an integral maximum flow $f^\star$ in G, if an edge e takes a flow $0 \leq f^\star(e) < c(e)$ the capacity of e, then $c(e)$ is larger than $f^\star(e)$ by a positive integer which is at least 1. In this case, reducing $c(e)$ to $c(e) - 1$ still allows the original flow $f^\star$, so e is not a lower-binding edge. Now we know if an edge e is lower-binding, it has to take full flow in the maximum flow $f^\star$. As an aside, each edge of any minimum cut is lower-binding. However, there could be many minimum cuts and it may not be easy or efficient to find all minimum cuts. Notice that for an edge $e = (u, v)$ with $f^\star(e) = c(e)$, it can only take a flow of value $c(e) - 1$ when its capacity is reduced by 1. If it is not lower-binding, which means the value of maximum flow won't change after this, there must be a way to reroute the unit flow from u to v not using e. On the other hand, if there is way of reroute 1 unit of flow form u to v without using e, then we can see that reducing its capacity by 1 will not decrease the maximum flow. Next we formalize our idea by proving the following statement. Statement: An edge $(u, v) \in E$ is not lower-binding if and only if there is a path from u to v in the residual graph.
Proof. First we prove that for the edge $e = (u, v) \in E$ if there is a path from u to v in the residual graph, then e is not lower-binding. If the path is just the single edge (u, v) in the residual graph, it means that the residual capacity of (u, v) is not zero. That is to say, in flow $f^\star$, edge (u, v) is not fully used. As we noted earlier this means decreasing $c(e)$ by 1 will not decrease the maximum flow. If the residual capacity of (u, v) is 0, and of (v, u) is $c(e)$ it means that the edge of (u, v) is fully occupied by the flow of $f^\star$ .If there is a path from u to v without using e, then we can create a new flow f' by pushing one unit of flow from u to v using the same method as that in the Ford-Fulkerson Algorithm. During this process, the capacity condition at every edge is kept as well. After this operation, if we push back one unit of flow from v to u, the conservation conditions at u and v are fixed. Moreover, the new flow f' has $f'(e) = f^\star(e) - 1 = c(e) - 1$. This means that if we reduce $c(e)$ by 1 we can still find another maximum flow f' with a value the same to $f^\star$, which is to say e is not lowerbinding. Then we prove that if edge $e = (u, v)$ is not lower-binding, there is a path from u to v in the residual graph. If $f^\star(e) < c(e)$ then there must be a path from u to v in the residual graph. So we assume, $f^\star(e) = c(e)$. Since

e is not lower-binding, the maximum flow does not decrease if we reduce $c(e)$ by 1. However, this means that the new maximum flow f' has $f'(e) \leq c(e) - 1$. We can think of changing from $f^\star(e)$ to $f'(e)$ as pushing $f^\star(e) - f'(e)$ units of flow from v back to u through edge (v, u) in the residual graph. Now the conservation conditions at u and v no longer hold. Since f' is a maximum flow just like $f^\star$, there must be a way to propagate at least 1 unit of flow from u to v in $G_{f^\star}$ to counteract the flow pushed back through edge (v, u). This means that there is a path from u to v.
Algorithm:
1. Get the residual graph.
2. For each vertex $v \in V$, run BFS or DFS to find all the edges $e = (u, v)$ such that there is a path from u to v in the residual graph.
3. For all other edges $e' = (u, v')$ with $f^\star(e') = c(e')$ return e' as a lower binding edge.
Correctness follows from the Statement we provided.
Running time: Obtaining the residual graph takes $O(m)$; running BFS DFS takes $O(m + n) = O(m)$ and we need to run it for every vertex so it takes $O(mn)$. In total, the algorithm runs in $O(mn)$.

Applications of Network Flow

5. In this solution we reduce the problem to project selection, which we know can be solved efficiently via a reduction to maximum flow. The reduction is as follows: create a project p_v for each vertex v in G, and create a project q_e for each edge e in G. Each project p_v has no dependencies, but a profit of -α . The projects q_e have profit 1, but depend on p_v and p_u, where $e = \{u, v\}$. This is the entire reduction. We now claim that there exists a set S of vertices with $\frac{f(S)}{|S|} > \alpha$ if and only if there is a choice of projects with positive profit. Consequently, solving the project selection problem gives a way to solve our current problem. To see the claim, let S be a set of vertices with $\frac{f(S)}{|S|} > \alpha$. Then $f(S) - \alpha|S| > 0$. Accordingly, we can consider the choice of projects p_v for $v \in S$ and q_e for edges $e = \{u, v\}$ with both u and v in S. Then the profit is precisely $1 \cdot f\{S\} + (-\alpha) \cdot |S| > 0$. For the other direction, suppose we have some choice of projects which gives a positive profit. Let P denote the set of projects p_v which are selected, and Q denote the set of projects q_e which are selected. We know that $|Q| - \alpha|P| > 0$. Now set $S = \{v : p_v \in P\}$. Since all projects q_e in Q have all their dependencies in P, it follows that $f(S) \geq |Q|$, while $|S| = |P|$. Therefore we have the inequality $f(S) - \alpha|S| \geq |Q| - \alpha|P| > 0$ as desired.

6. Algorithm: Form as a bipartite matching problem. Create a node for every grid square and put all the white square nodes in the set L and all the black square nodes in the set R. For every white grid square (which has a corresponding vertex in L), create an edge connecting its vertex in L to the vertices corresponding to the black squares adjacent to it in the grid. Create a node s that is connected to every element of L with edge weight 1, and create a node t that is connected to every element of R with edge weight 1. Run FF to get $\max(s, t)$ flow, and if the $\max$ flow is the number of squares divided by 2 (let's call this number m), then return true. Proof of Correctness:
(1) First we must prove that a solution to this bipartite matching reduction gives a solution to our problem. The flow can be used to construct the solution as follows: outgoing flow from a node in L only goes to one node in R (which corresponds to an adjacent black tile). Also, each node in R can only take in flow from one node in L, because there is only one outgoing edge from each node in R to t with capacity 1. Altogether, these facts imply that each square is uniquely tiled. For each pair of nodes in L and R such that there is flow pushed between the nodes from L to R, place a tile covering these two adjacent nodes.
(2) Second we must prove that a solution to our problem (if it exists) gives a solution to this bipartite matching reduction. Each tile covers a white tile and a black tile. In the network, flow 1 unit from s to a white tile's node to a black tile's node to t if there is a tile present.

7. Our solution will reformulate the escape problem as a network flow problem. We can think of vertex-disjoint paths through the grid as distinct paths carrying flow through a network. Since we want to determine whether

there are m disjoint paths from the terminals to the boundaries, we will try to push m units of flow through the terminals to the boundaries, bounding the flow along each path to 1 unit. Some caution is required, though. If we just turn the grid into a network in the obvious way (by connecting a source to the terminals, the boundaries to a sink, and setting directed edge capacities of 1 throughout), it is possible that we could achieve a flow value of m even if there are no m vertex-disjoint paths. This is because the flow paths need not be vertex-disjoint: two units of flow could enter the same vertex and then continue along different out edges. We need to modify the "obvious" network to make sure that each vertex has at most one unit of flow sent through it. One way to accomplish this is by adding a single "internal" edge of capacity 1 inside each vertex. This way, we can't send more than one unit of flow through any vertex because the internal edge will be saturated. While internal edges aren't one of the basic building blocks of graphs or networks, we can achieve the same effect by splitting each vertex v into two vertices v_{in} and v_{out} and including a directed edge $(v_{in}; v_{out})$ of capacity 1. The intuition here is that v_{in} is where flow enters v , while v_{out} is where it exits, and $(v_{in}; v_{out})$ plays the part of the desired internal edge. With this construct in mind, we are now ready to present our algorithm. Escape(G):
1. Construct a flow network G' as follows:
a. Create a source s and a sink t
b. For each vertex v in G, create a pair of vertices v_{in} and v_{out} with a directed edge (v_{in}, v_{out}) of capacity 1.
c. For each pair v, w of adjacent verticies in G, create (in G') directed edges of capacity 1 from v_{out} to w_{in} and from w_{out} to v_{in}
d. For each terminal vertex v in G, create (in G') a directed edge of capacity 1 from s to v_{in}
e. For each boundary vertex w in G, create a directed edge of capacity 1 from w_{out} to t
2. Run Ford Fulkerson on G' to determine the value of a maximum flow
3. If that $\max$ flow is m, return true (escape problem can be solved); else, return false.
Runtime Analysis: There are $2n^2 + 2$ vertices in the network G'. Each of s and t has $O\left(n^2\right)$ edges, and each other vertex has at most 5 edges. The number of edges in G' is therefore $O\left(n^2\right)+O\left(n^2\right)+5O\left(n^2\right) = O\left(n^2\right)$. So the time required for network construction is $O\left(|V| + |E|\right) = O\left(n^2\right) + O\left(n^2\right) = O\left(n^2\right)$. The runtime of Ford-Fulkerson is $O\left(|E|^\star|f^\star|\right)$. We have $|E| = O\left(n^2\right)$ and the maximum flow value cannot be more than m, so this runtime is $O\left(mn^2\right)$. This dominates $O\left(n^2\right)$ term for networkconstruction so we conclude that the overall runtime is $O\left(mn^2\right)$.
Proof of Correctness: It suffices to show that the escape problem is solvable if and only if G' has a flow of total weight m. First suppose the escape problem is solvable, so that there exist m vertex-disjoint paths from all of the terminals to boundary vertices. Since s has an edge to each terminal and each boundary vertex has an edge to t, this means that there are m vertex-disjoint paths from s to t in G'. All edges in all of these paths have capacity 1, so by sending a single unit of flow through each path, we obtain a flow of total weight m. Now suppose that G' has a flow of weight m. Since all edge capacities are integers, we know that there exists a flow $f^\star$ with only integral flow values such that $|f^\star| = m$. The flow $f^\star$ is obtainable (via Ford-fulkerson) by sequentially sending one more unit of flow along an augmenting path. All edge capacities are 1, so no single path can carry more than a single unit of f low; to have a total flow of m, there must therefore be m different s, t paths in G'. These paths are edgedisjoint-if any two of them shared an edge, then the shared edge would carry at least 2 units of flow, violating the edge capacity of 1.
Each of the m paths in G' can be specified by listing the visited vertices in order and each path will have the form $\left(s, v^1_{in}, v^1_{out}, ..., v^k_{in}, v^k_{out}, t\right)$, where $\left(v^1_{in}, v^1_{out}\right)$ corresponds to a terminal vertex v^1 and $\left(v^k_{in}, v^k_{out}\right)$. Corresponds to a boundary vertex v_k. We can "collapse" this path into the terminal boundary path $(v_1, ..., v_k)$ in G by eliminating s, t and the internal edges. No 2 of the paths obtained in this way can share any vertices. If 2 of the paths shared a vertex v, then the internal edge (v_{in}, v_{out}) would be used by both corresponding paths in G', and so that edge would be filled over capacity $f^\star$. We conclude that there are m vertex-disjoint paths in G from terminal vertices to boundary vertices, so the escape problem is solvable.

8. We begin with two observations. First, we can forget about the integral part of each element in the matrix$A$, and focus only on the fractional part. This is because a rounding preserves row- and column-sums in the simplified matrix iff it also preserves the sums in the original matrix. So from here on $0 \leq A[i, j] \leq 1$ for each i, j.

   Second, if any of the rows or columns in A sum to a non-integer, then a rounding does not exist. This is because

the rounding is supposed to preserve the row- and column-sums, and after the rounding those sums are integers. Therefore, we first calculate the sum all the rows and columns and return infeasible if any of them are not integral. So from here on all row- and column-sums of A are integers.

Next, we reduce the problem to a matching-style flow problem as follows.

- We setup a bipartite graph, with a node on the left side for each row in A (row nodes), and a node on the right side for each column in A (column nodes).
- For every row node i and column node j, we introduce the edge (i, j) of capacity equal to 1 iff $A[i][j]$ is not an integer. This represents the fact that we can round numbers up as long as they are not already integers.
- We add a source node s and connect it to all row nodes. The capacity of edge (s, i) equals the sum of row i.
- We add a sink node t and connect all column nodes to it. The capacity of edge (j, t) equals the sum of col j.

We claim that there is a rounding of A iff there is an integral flow in this network that saturates all edges leaving s and all edges entering t.

First off, notice that the flow saturating all edges leaving s versus saturating all edges entering t are actually equivalent conditions. This is because if all s-leaving edges are saturated, then the flow has value equal to the sum of all rows, which equals the sum of all columns, and which implies that all t-entering edges are also saturated. The converse direction follows similarly.

Now suppose that a valid rounding exists. Then construct a flow by sending 1 unit along edge (i, j) if $A[i][j]$ is rounded up; otherwise send no flow along (i, j). Observe that in row i, the number elements that are rounded up must equal to the sum of row i; this implies that the flow we just defined saturates all the edges leaving s (hence all edges entering t).

For the opposite direction, suppose that we have an integral flow that saturates all s-leaving edges and all t-entering edges. Round $A[i][j]$ up if the edge (i, j) carries flow; round it down otherwise. Consider a row node i; because the flow saturates the edge (s, i), and because of flow conservation, we know that the number of j such that $(i, j) = 1$ is exactly the sum of row i, meaning that the rounding preserves all row-sums. A similar argument applies to column sums.

Although not required for the solution, we derive another interesting fact. Namely, if the sum of the elements of all rows and columns are integers then there will always be a rounding that exists for the matrix. The proof is as follows - in the graph we setup, we initially send flow equal to the capacity of the edge to all the row nodes. Then for a row node i and a column node j we will send the flow equal to the fractional amount of $A[i][j]$ from node i to node j. Note that doing this will saturate the edges from source to the row nodes and also corresponds to a flow equal to the sum of all the fractional values (we can do this for any matrix with integral sums). But since all the capacities are integral, if a flow exists with some value, there must also exist a integral flow (flow through all edges being integers) with the same flow amount. This, in our graph means there will be a flow with the same values such that the values are either only 0 or 1. This implies that a rounding must exist.