CS 261 Problem Set 1

Luis A. Perez

Problem 1

Solution:

(a) We proof that for every flow f, there exists an acyclic flow f' with the same values as f. We do this by showing a transformation from a cyclic flow f to the acyclic flow f' which preserves the values of f.

To begin, suppose we have a flow f in G whose subgraph of directed edges with positive flow contains at least one directed cycle, $C = \{e \mid e \text{ is in the cycle.}\}$. Now find the minimum flow along along the cycle:

$$f_c = \min_{e \in C} f_e > 0$$

Now, we can construct a new flow f' such that $f'_e = f_e - f_c$ for $e \in C$ (decrease the flow along the cycle by its minimum edge flow). Note that this will effectively destroy the cycle.

The claim is that the value of flow f and f' are equal. This is because f_c is "excess" flow that neither enters not exists the cycle C, so removing has no effect on any external flow values.

For a flow f with multiple cycles, we can simply repeat the above process for each cycle. Since removing a single cycle does not affect the value of the flow, the above construction will create a new flow f' that is acyclic and which has the same flow.

In particular, this implies that some maximum flow is acyclic.

(b) Consider some acyclic flow f as its corresponding subgraph G_f. We show that we can write this acyclic flow as the sum of at most m path flows. We do this by providing a constructive proof.

First, find a simple directed path, p, from $s \to t$ on our flow subgraph G_f. This is possible since f is an acyclic flow, so a simple directed path from $s \to t$ must exists.

Let E_p be the edges contained our flow path p. Then define the flow along p as $f_p = \min_{e \in E_p} f_e$ (the minimum flow along any edge forming a part of p). This defines our first path flow where each edge on the path flow is assigned the flow $f_p > 0$.

We then modify our flow f to subtract out this path flow. We can do this in a straight-forward way, by simple subtracting the path flow value from every edge in the path. Note that this process will zero out at least one edge in our flow, and will leave us with a new acyclic flow.

Repeating this process, we can contruct our second path flow, and so forth. Since each path flow zeros out one f_e, we will end with at most m such path flows.

With the above, we have proven that every acyclic flow can be written as the sum of at most m path flows.

(c) Edmonds-Karp is guaranteed to produce an acyclic maximum flow. We know from lecture that it is guaranteed to produce a maximum flow. Futhermore, Edmonds-Karp always augments along simple directed paths from $s \to t$ (the shortest path in fact), so the final flow it produces is simply the sum of multiple path flows. By (b), this implies that the final flow is a maximum acyclic flow.

(d) We prove that every flow can be written as the sum of at most m path and cycle flows in a straight-forward fashion.

First, find any cycle in our flow graph. If no such cycles exists, continue to the next step. Otherwise, define a the cycle flow c. Let E_c be the edges contained in our cycle flow c. Then define the flow along c as $f_c = \min_{e \in E_c} f_e$ (the minimum flow along any edges forming part of c). This define our first cycle flow where each edge on the cycle flow is assigned the flow $f_c > 0$.

We then modify our flow f to subtract out this cycle flow. We can do this in a straight-forward way, by simply subtracting the cycle flow value from every edge in the cycle. Note that this process will zero out at least one edge in our flow, thereby removing this cycle.

Once all cycles have been removed, we now have an acyclic flow containing at most $m - N_c$ edges, where N_c is the number of cycles we removed. From the results in part (b), this acyclic path flow can be written as the sum of at most $m - N_c$ path flows.

Putting everything together, this implies we can write any flow as the sum of at most m path and cycle flows.

(e) Yes.

The Algorithm

We provide a brief sketch of an algorithm that does exactly this. The basic idea is to use DFS to either (1) find a simple directed path from s to t or (2) find a cycle. Once we've found this path or cycle, we find the minimum flow along it as defined by f. We take the resulting path flow or cycle flow as part of our decomposition, modify f to remove the path flow or cycle flow, and repeat the process on the resulting flow f'. We continue this until there are no simple directed paths from s to t or edges.

Correctness

The correctness of the algorithm follows from (d), where we proved that every flow can be written as the sume of path and cycle flows. Since finding these paths/cycle flows and removing them one-by-one is exactly what the proposed algorithm does, by (d) it will compute the decomposition.

Running Time

For the running time analysis, we first consider how many times we repeat the DFS search. Since everytime we remove either a path flow or a cycle flow with maximal flow, then by (d) we repeat this process at most m times.

In each iteration, we effectively do a modified version of DFS. Note that we stop as soon as we find a cycle, which means that our DFS will visit at most $n + 1$ nodes (once we've visited this many, we're guaranteed to have discovered a cycle). As such, the running time of this early stopping version of DFS is actually $O(n)$. Computing the minimum flow along the discovered path/cycle and updating the original flow and graph will take at most $O(n)$ time, since we look at only $O(n)$ edges.

Putting it all together, this gives an algorithm with running time $O(mn)$ as desired.

Problem 2

Solution:

(a) We proof that the running time of the provided algorithm is $O(mn)$, following the provided hints.

We note that **Initialize** is $O(m)$.

We note that **Retreat** can be called at most n times, since on each call we delete at least one node which is an $O(1)$ operation.

Similarly, **Augment** can be called at most m times, since on each call we saturate (and delete) one edge, which is an $O(1)$ operation.

Lastly, **Advance** can be called at most $d(f) = O(n)$ times without a call to **Retreat** or **Augment**, since on each such call our path P increases in size by 1, and it can be at most length $d(f)$.

Putting the above together, the running time is $O(m + (n + m)n\})$, since it takes $O(m)$ time to **Initialize** and **Advance** is called at most $O(n)$ times between each a call to either **Retreat** or **Augment**. As such, the final running time of a reasonable implementation is $O(mn)$.

(b) We proof that the provided algorithm terminates with a blocking flow in L_f. We proof by contradiction.

Suppose the algorithm terminates with a flow g in L_f which is not blocking. This means that there exists an s-t path P of L_f such that every edge of P has $g_e < u_e$. However, this implies our algorithm would have continued. Consider the final call to **Augment**, where be backtrack to the last unsaturated edge. Either this edge is contained in P, or we are at s. In either situation, since P exists, the call to **Advance** would have followed the outgoing edges along P until reaching t, and calling **Augment** one more time. This contradict the fact that our algorithm halted.

(c) We now suppose that every edge of L_f has capacity of 1, and prove that the algorithm has running time $O(m)$. As the hint indicates, let us consider how many times **Advance** can be called over the entirety of the algorithm. Since each edge only has capacity of 1, a call to **Augment** will saturate all edges along P. As such, an edge (v, w) can only ever be included once in a path P, implying that **Advance** can only be called at most $O(m)$ times over the entirety of the algorithm.

As such, the total runtime is then given by $O(m + n + m) = O(m)$.

Problem 3

Solution:

(a) The modification is straight-forward. The traditional version of Dijkstra's shortest-path algorithmn orders paths based on the sum of the lengths of the edges in the path and seeks to find the minimal length. In our case, we wish to order paths based on the minimum of the residual edge capacities in the path, and we wish to find the maximal such path. As such, we modify Dijkstra's by changing all sums to mins and minimization to maximization using the edge capacities as edge lengths.

Withouth any further modifications, running this modified version of Dijkstra's will find the maximum possible minimum residual edge capacity s-t path. The fact that the running time remains unchanged is immediately obvious, so we focus now on providing an argument for correctness.

We focus on proving only the critical part of Dijkstra's algorithm, with our modification. Let us consider some intermediate iteration i, where we remove some vertex v from our queue. Let u be v's predecessor. By induction, since u was previously removed from the queue, we know the maximum-possible minimum residual edge capacity path from $s \to u$. As such, the minimum between this residual edge capacity and the residual edge capacity of $u \to v$ gives us the minimum residual edge capacity path from $s \to v$ through u.

We need to proof that this path through u is the maximum-possible such path. For a contradiction, suppose another path p exists with a higher minimum residual edge capacity starting at s and ending at u. Since s is no longer in the queue but u is, there exists an edge (a, b) such that a is outside the queue and b is inside. By induction, a had the correct maximum-possible minimum residual value from $s \to a$, and when processing its neighbors, we found b to have a larger or equal maximum-possible minimum residual value than either the path $s \to a$ or the residual capacity of $a \to b$. But this would mean that its residual capacity is at least that of p, while the residual capacity of $s \to u$ is smaller than p, which is a contradiction since we chose u to be the element with the largest residual capacity among all elements in the queue.

(b) Following the hint, let Δ be the maximum amout of flow that can be pushed along any s-t path in G_f. Then this defines an s-t cut as described in the hint. The minimum capacity of this s-t cut is at most $m\Delta$ (all edges crossing the cut must have less than Δ residual capacity and there at at most m such edges). This value is also an upper bound on the maximum flow in the residual graph, given by $F^* - F$. Recalling from lecture that the maximum flow is at most the minimum s-t cut, this leads to the following inequality:

$$\underbrace{F^* - F}_{\text{maxflow in } G_f} \leq \underbrace{m\Delta}_{\text{maximum capacity of our s-t cut}}$$
$$\implies \Delta \geq \frac{F^* - F}{m}$$

As such, we have the maximum amout of flow that can be pushed on any $s-t$ path is at least $(F^* - F)/m$, which means that there is an augmented path in G_f such that every edge has residual capacity of at least this amount.

(c) We wish to proof that this variant of Edmonds-Karp will terminate within $O(m \log F^*)$ iterations. To see this, let us consider what happens at each iteration i. Using our modified algorithm, a single iteration increases the current flow by at least $(F^* - F_i)/m$ (see results from (b)). Furthermore, the algorithm will halt once there is no more flow to push at a particular iteration i.

Let us define R_i as the amount of flow that can still be pushed at the beginning of iteration i. Then we have the following:

$$\begin{aligned}
R_i &= F^* - F_i && \text{(Definition of residual flow)} \\
&\leq F^* - \left(F_{i-1} + \frac{R_{i-1}}{m}\right) && \text{(Follows from the fact that } F_i \geq F_{i-1} + \tfrac{R_{i-1}}{m}\text{)} \\
&= (F^* - F_{i-1}) - \frac{R_{i-1}}{m} && \text{(distribute)} \\
&= R_{i-1}\left(1 - \frac{1}{m}\right)
\end{aligned}$$

We know that $R_0 = F^*$, so the above leads to:

$$R_i \leq F^* \left(1 - \frac{1}{m}\right)^i$$

Where R_i is the residual capacity at iteration i and F^* is the maximum flow. Following the hint, we can rewrite the above as:

$$R_i \leq F^* e^{-\frac{i}{m}}$$

The algorithm will terminate at i such that $R_i \leq 1$ (assuming integer capacities). As such, solving for i under these conditions we have:

$$\begin{aligned}
1 &\leq F^* e^{\frac{-i}{m}} && \text{(Termination condition)} \\
\implies e^{\frac{i}{m}} &\leq F^* \\
\implies \frac{i}{m} &\leq \log F^* \\
\implies i &\leq m \log F^*
\end{aligned}$$

As such, we have that the algorithm must terminate within $O(m \log F*)$ iterations.

(d) If we use a Fibonnaci heap for the modified version of Dijkstra's, we can have it run in $O(m + n\log n)$ running time. From (c), we know that the total number of iterations is $O(m \log F^*)$. Given that all edge capacities are integers in $\{1, 2, \cdots, U\}$, we must have $F^* = O(U)$, giving the number of iterations as $O(m \log U)$. Putting all of this together, with a Fibonacci heap implementatin, we can achieve a runtime of:

$$O([m + n\log n]m \log U) = O(m^2 \log U + n \log n \log U) = O(m^2 \log U)$$

The running time above is polynomia in the input size, since it takes only $\log U$ bits to represent the edge capacities.

Problem 4

Solution:

(a) We now propose an efficient algorithm that rounds A in the fashion described. We do this by reducing to a maximum flow problem.

The Algorithm:

The first step is to compute the sums of each row and each column with the given real numbers from the matrix. The next step is to compute the sums of each row and column with all entries in the matrix rounded down to the nearest integer. More precisely, we compute the following important "capacities":

$$R_i = \sum_{j=1}^{n} A[i][j] - \sum_{j=1}^{n} \lfloor A[i][j] \rfloor$$
$$C_j = \sum_{i=1}^{m} A[i][j] - \sum_{i=1}^{m} \lfloor A[i][j] \rfloor$$

Note that all R_i and C_j are non-negative ($\lfloor x \rfloor \leq x$) and integral since we're told that all row and columns sums of A are integral. With the above computations completed, we now construct a graph $G = (V = R \cup C \cup \{s,t\}, E)$ with $m + n + 2$ vertices. We have m vertices corresponding to rows (R) in A and n vertices corresponds to columns (C) in A. Additionally, we have source and sink vertices, s, t.

We now define the connectivity and capacities for our graph, G.

First, for every $r \in R$ we have an edge (s, r) such that $u_{s \to r} = R_r$ (the capacity of the edge is the R_r quantity computed above, which indicates the number of entries in row r which need to be rounded up in order to achive the sum required for row r).

Similarly, for every $c \in C$ we have an edge (c, t) such that $u_{c \to t} = C_c$ (the capacity of the edge is the C_c quantity computed above, which indicates the number of entries in column c which need to be rounded up in order to achieve the sum required for column c).

Lastly, for all $r \in R$ and $c \in C$, we add the edge (r, c) such that $u_{r \to c} = 1$ (the capacity is 1, which allows this "entry" to be selected as the entry to be rounded up).

We now compute the maximum flow f of G. Then for each entry $A[i][j]$ in the matrix, we set the output as per

$$A'[i][j] = \begin{cases} \lceil A[i][j] \rceil & f_{i \to j} = 1 \\ \lfloor A[i][j] \rfloor & \text{otherwise} \end{cases}$$

Basically, if there's flow on the edge (i, j), we round up. If there's not, we round down.

Running Time The algorithm is just a reduction to the maximum flow problem with $O(m + n)$ vertices and $O(mn)$ edges, where n is the number of columns and m the number of rows in A. The reduction itself takes $O(mn)$ time.

Proof of Correctness The correctness of the algorithm follows almost immediately from the description. By construction, the entries in the result matrix must be either rounded down or rounded up from their orignal values.

Furthermore, note that the maximum flow of the constructed graph saturates the capacity of all (s, r) and (c, t) edges. This is due to the fact that all (r, c) edges exists, and as such do not impede the flow from the set R to the set C at all.

Since these edge capacities are simply the residuals from rounding down, this implies that the sums of the rows and columns remain the same as in the original matrix. Similarly, any edge (r, c) which has flow through it corresponds to the entry $A[r][c]$ being rounded up in order to saturate the sums.

(b) The proof that such a rounding is guaranteed to exists basically follows from (a). From class, we know that an integral maximum flow is always achievable if all edge capacities are integral. As such, since (a) reduces to an integral maximum flow problem, a rounding as described is always guaranteed to exists because the reduction constructs a graph with all integer capacities (since all row and column sums of A are integral). See (a) for more detail.

Problem 5

Solution:

(a) We prove that in every graph $G = (V, E)$, the minimum size of a vertex cover is at least the size of a maximum matching. Let $M \subseteq E$ be a maximum matching. This means that there are maximally $|M|$ edges with no shared endpoints. Since a vertex cover must cover every edge, all vertex covers S must contain at least $|M|$ vertices to cover the $|M|$ edges in the maximal matching.

(b) The simplest such example is simply a triangle. More formally, we have an undirected graph $G = (V, E)$ where $V = \{a, b, c\}$ and $E = \{(a, b), (b, c), (c, d)\}$. Then we must have that any vertex cover S must be such that $|S| \geq 2$, since we have three edges but all vertices only have degree 2. Furthermore, we have all matching M such that $|M| \leq 1$, since any set of two edges share an endpoint.

This is an example where the minimum vertex cover is strictly bigger than the size of the maximum matching.

(c) As per the hint, we perform a reduction the maximum flow. The reduction is as follows.

Given bipartite graph $G = (U \cup V, E)$, construct the graph $G' = (U \cup V \cup \{s, t\}, E')$ where s and t are two new vertices. We have the following directed edges: (1) (s, u) for all $u \in U$ with capacity 1, (2) (v, t) for all $v \in V$ with capacity 1, and (3) (u, v) for all $(u, v) \in E$ with capacity ∞ (same reduction we used for maximal matching).

We now compute the maximum flow, which also gives us the minimum capacity cut S in G'. We output as a vertex cover, C, which consists of the union of the following elements:

- All nodes in U that are not in S (left-side nodes not reachable from s in our maxflow)
- All nodes in V that are in S (right-side nodes reachable from s in our maxflow)
- The set of vertices in V that are not in S (right-sid enodes not reachable from s in our maxflow) that have neighbors in the set of vertices U reachable from s. Call this set T.

We claim that $C = (U \setminus S) \cup (V \cap S) \cup T$ is a minimum vertex cover of G. We proof this in two steps. First, let us argue that C is a vectex cover. That is to say, C is a subset of $U \cap V$ such that for every edge $e \in E$, at least one vertex is included in C. By construction, this is immediately true for all edges with endpoints in $(U \setminus S) \cup (V \cap S)$, since these nodes are directly included in C. Then note that we cover the remaining set of edges with T since it consists of all verticies in $V \setminus S$ that have neighbors in $U \cap S$.

This shows that C is a vertext cover. Next, we need to show that C is a minimum such cover.

To see this, note that the capacity of our s-t cut is given by $|U \setminus S| + |V \cap S| +$ #edges from $(U \cap S)$ to $(V \setminus S)$. Furthermore, we must have that the #edges from $(U \cap S)$ to $(V \setminus S)$ is at least $|T|$ (in the worst-case, theres only on edge per vertex). Putthing this together, we have:

$$|C| = |U \setminus S| + |V \cap S| + |T| \leq \Delta$$

where Δ is the capacity of our s-t cut, which is the minimum-cut in G, which is equal to the maximum flow in G. From class, we know that this maximum flow corresponds to a maximal bipartite matching, and from (a), we know that the minimum size of a vertex cover is at least the size of the maximum matching Δ. As such, $|C| = \Delta$ must be a minimal vertex cover.

(d) The proof follows almost immediately from the algorithm proposed above.

Consider a bipartite G, and a transformation into a flow network G' as defined previously (and in lecture). We know that the size of the maximal matching of G is equal to the maxflow in G' (from lecture), and the proof of correctness of (c), we know that the size of the minimum vertex cover of G is equal to the minimum capacity of an (s-t)-cut in G'.

Therefore, by the Max-Flow/Min-Cut Theorem, we have that:

$$\begin{aligned}
\text{size of maximum matching in } G &= \text{ maxflow value in } G' \\
&= \text{capacity of minimum (s-t)-cut in } G' \\
&\qquad\qquad \text{(Max-Flow/Min-Cut Theorem)} \\
&= \text{size of minimum vertex cover in } G' \quad \text{(By (c))}
\end{aligned}$$

This concludes our proof.

Problem 5

Solution:

(a) The intuition for this is that if a negative cycle exist in G_f, then we can divert some flow to go through this cycle and thereby reduce the cost of the flow (since the cycle has net negative cost).

We proof this by contradiction. Suppose that G_f is a residual network of flow f with a negative cycle, where f is a minimum-cost flow. However, we can construct a feasible flow f' with value d with lower cost than f, contradicting the fact that f is a minimum-cost flow.

For the construction, note we can always increase the flow along the cycle C without impacting the value of f. This is because the additional flow is essentially trapped within the cycle. However, since C is a negative cycle, this additional flow along C will give f' a lower cost that f.

(b) Suppose that the residual network of G_f of a flow f has no negative cycles (where f is feasible and has value d). We wish to show that f is the minimum-cost flow.

We do this by proving the contrapositive. That is, we proof that if f is not minimum-cost (but has value d), then the residual network G_f has a negative cycle. Let f^* be the minimum-cost flow. Then note that $f^* - f$ must contain a cycle, since by assumption the value of $f^* - f = 0$ (the values of f and f^* are both d) but the flows are not the same. Furthermore, because the cost of f^* is smaller than the cost of f, this cycle must have a negative cost. Therefore G_f contains a negative cycle.

(c) We can make use of Bellman-Ford Shortest Path algorithm to detect a negative cycle in G_f. Its runtime is $O(mn)$. We use the edge costs instead of edge lengths. We use the fact that after $n-1$ iterations of the inner-loops, Bellman-Ford is guaranteed to have computed the minimum cost source s and all other nodes. If we then perform the iteration again and the minimum cost of some vertex decreases, we know that a negative cycle exists. Otherwise, no negative cycles exists.

(d) We can follow a relatively simple approach using the results from above.

The Algorithm

Our first step is to find a feasible flow f with value d. We can do this by finding the maximum flow on G using any maximum-flow algorithm. In our case, let us use the short-path s-t path heuristic for Edmonds Karp. If the value of this maximum flow is less than d, we exit and report the problem as unfeasible.

Otherwise, we can modify the maximum flow (decrease the flow by 1 along some simple s-t path) until we arrive at a flow with value d. Let this be our flow f.

Next, while there exists a negative cycle C in G_f, let $c = \min_{(u,v)\in C} u_{u\to v}$ (find the minimum flow that can be pushed along C). Then send c units of flow along C.

Correctness

The correctness of the algorithm follows almost immediately. It's clear that f is a feasible flow with value d. Furthermore, the algorithm does not terminate until G_f has no negative cycles. By our results from (b), when the algorithm terminates, f is a minimum-cost flow.

Running Time

We now argue the running time is $O(nm^2M^2)$, which is polynomial in n, m, M as required.

The first step requires computing the maximum flow using shortest-path s-t heuristic for Edmonds Karp, which has running time $O(m^2n)$

Once the maximum flow is found, we decrease the flow by at least 1 along some simple s-t path (which can be found using BFS). This will take at most $O(mM)$ iterations (mM is the maximum possible flow), so the total runtime of this part is $O(m^2M)$.

Now we enter the cycle detection section. Using the algorithm discussed in (c), cycle detection will take $O(mn)$ time per iteration (this also finds the cycle). On each iteration, we decrease the cost by at least 1 (since C is a negative cycle). Note that the cost of any flow f is at most mM^2 (every edge has maximum flow M and has cost M per unit of flow), and is at least $-mM^2$ (every edge has maximum flow M and has cost $-M$ per unit of flow). Since each iteration decreases the cost by at least 1, we will terminate in at most $2mM^2$ iterations. This gives the total running time of this part of the algorithm as $O(nm^2M^2)$.

Putting everything together, we have the total runtime as $O(m^2n+m^2M+nm^2M^2) = O(nm^2M^2)$.