Problem 1

Part (a)

I argue that the running time of Dial's algorithm is $O(m + D)$, where D is the maximum distance, and that the algorithm still works if some edges have length 0.

Dial's algorithm keeps a pointer to the min element in the array, starting at index 0. This pointer only advances. Once the last node has been processed and its (shortest path) distance calculated, the algorithm can stop. This happens when the pointer is at index D. Therefore, the pointer to the min only advances D indices. As Dial's algorithm processes each edge once, its running time is $O(m + D)$.

Dial's algorithm still works with some edges having length 0. An edge from v to w of length 0 will be processed when v is processed, that is when v is the current min (at index d_v). If w has not yet been processed (so $d_w \geq d_v$), d_w is reduced to d_v and placed in the array at the index of the current min. So once v is processed, w (and any other elements sitting at the min) will be processed. Basically, an edge of length 0 might just decreases the key of an element to the current min, but not below it, and so doesn't disturb the invariant of Dial's algorithm that the min pointer only increases.

Part (b)

I show that the *reduced edge lengths* l^d_{vw} defined by the rule $l^d_{vw} = l_{vw} + d_v - d_w$ are all nonnegative integers. For any edge from v to w, I can form a path from the source s to w by going from s to v and then through that edge from v to w. Therefore, $l_{vw} + d_v \geq d_w$, so $l^d_{vw} = l_{vw} + d_v - d_w \geq 0$. Hence, all $l^d_{vw} \geq 0$.

Part (c)

By telescoping, the length of a path P from s to t under the reduced length function is:

$$\sum_{(v,w)\in P} l^d_{vw} = \sum_{(v,w)\in P} l_{vw} + d_s - d_t = \sum_{(v,w)\in P} l_{vw} - d_t$$

If the length of the path P under l^d is 0, then $\sum_{(v,w)\in P} l_{vw} = d_t$, so this path is a shortest path under l.

If the length of the path P under l^d is greater than 0, then $\sum_{(v,w)\in P} l_{vw} > d_t$, so this path is not a shortest path under l.

Therefore, the shortest paths under the reduced length function have length 0 and are the same as those under the original length function.

Part (d)

I devise a scaling algorithm for shortest paths.

I start with $d_v = 0, \forall v$. Then, I repeat until no bits are left:

- Shift next higher-order bit into l_{vw}.
- Left shift (i.e. double) $d_v, \forall v$.
- Calculate all l^d_{vw} using the l_{vw}'s shifted so far and the d_v's.
- Run Dial's algorithm, collecting new $d'_v, \forall v$.
- Update $d_v \leftarrow d_v + d'_v, \forall v$.

The algorithm is correct, because our analysis from parts (b) and (c) hold as the distance function d satisfies $l_{vw} + d_v \geq d_w$ at all time since it is just an under-estimate of the actual distances.

Each iteration of the algorithm takes $O(m+n) = O(m)$, since the maximum distance D is $\leq n$, as, by bit-scaling, each edge along a path might contribute at most 1 to the distance. Overall, I have $\lfloor \log C \rfloor + 1$ iterations. Therefore, the running time is $O(m \log C)$.

Part (e)

Suppose I use base-b scaling. Then, at each iteration the algorithm takes $O(m + n(b-1))$ because the maximum distance D is $\leq n(b-1)$ as each edge along a path might contribute at most $b-1$ to the distance. By solving for b in $m = n(b-1)$, I get $b = \frac{n+m}{n} = 1 + \frac{m}{n}$. Therefore, I can achieve the slightly better running time of $O(m \log_{(2+\lfloor \frac{m}{n} \rfloor)} C)$ by scaling with the base $2 + \lfloor \frac{m}{n} \rfloor$.

Problem 2

Let d be the distance between the source and the sink. Let l_i be the number of nodes in the ith layer in the admissible graph (the number of nodes in source layer being l_0 and the number of nodes in the sink layer, l_{d+1}). Then, $\sum_{i=1}^{d} l_i + l_{i+1} \leq 2n$. Indeed, each node is counted at most twice. Since we have d numbers that sum to less than $2n$, one of these numbers must be $\leq \frac{2n}{d}$. So there exist an i, such that $l_i + l_{i+1} \leq \frac{2n}{d}$. In the best case for max flow, l_i has $\frac{n}{d}$ vertices and l_{i+1} has $\frac{n}{d}$ vertices, and the number of edges between the two layers is $\leq \frac{n^2}{d^2}$. Therefore, the maximum flow is at most n^2/d^2.

After k blocking flows, the distance between the source and the sink is at least k, in which case the maximum remaining flow is at most n^2/k^2. As each blocking flow finds at least a unit of flow, $O(k + \frac{n^2}{k^2})$ blocking flows suffice to find a maximum flow. Let $k = n^{\frac{2}{3}}$. Then $\frac{n^2}{k^2} = \frac{n^2}{n^{\frac{4}{3}}} = n^2 \cdot n^{-\frac{4}{3}} = n^{\frac{2}{3}}$. Therefore $O(n^{\frac{2}{3}})$ blocking flows suffice to find a maximum flow.

Problem 3

I transform to the standard maximum flow problem the network problem in which, in addition to arc capacities, each node i other than the source and the sink, might have an upper bound, say $w(i)$ on the amount of that can pass through it.

I create a standard maximum flow network G' from G as follows:

- For each vertex i in G: if i has some $w(i)$, in G', add vertices in_i and out_i and add an edge from in_i to out_i with capacity $w(i)$. If i doesn't have a $w(i)$, simply add a vertex i to G'.

- For each edge from v to w with capacity c in G: add an edge from out_v (or just v) to in_w (or just w) in G'.

Now, in order to solve the network problem with node capacities, I simply look for the maximum flow from s to t in G'.

From the perspective of work-case complexity, the maximum flow problem with node capacities is not more difficult to solve than the standard flow problem. Indeed, let m and n be the number of edges and nodes in G. In the worse-case, G' will have $m' = m + n = O(m)$ edges and $n' = 2n = O(n)$ nodes. Using Fold-Fulkerson, the running time is $O(m|f|)$ in both cases (and $|f|$ can only get smaller by adding more constraints).

Problem 4

Part (a)

To solve the minimum flow problem, I first find a feasible flow on the graph G and then convert it to a minimum flow.

To find a feasible flow on the graph G, I use an auxiliary graph G', which I construct as follows:

- For each vertex v in G: add a vertex v in G'.
- Add a super-source s' and a super-sink t'.
- Add an edge of infinite capacity from t to s.
- For each edge from i to j with capacity u_{ij} and lower bound l_{ij}: add an edge from i to j with capacity $u_{ij} - l_{ij}$, add an edge from i to t' with capacity l_{ij}, add an edge from s' to j with capacity l_{ij}.

Now, I run a maximum flow algorithm on G' from s' to t'. If all the edges from s' and all edges to t' are saturated, then I have a feasible flow. I convert this feasible flow f' in G' to a feasible flow f in G by letting the gross flow between i and j be $f_{ij} = f'_{ij} + l_{ij}$.

To convert this feasible flow f into a minimum flow, I construct a residual graph G'' by having an edge from i to j with capacity $u_{ij} - f_{ij} + f_{ji} - l_{ji}$ if there is an edge from i to j or an edge from j to i in G. Now, I run a maximum flow algorithm on G'' from t to s obtaining a flow f''. I decompose f'' into $f''_{ij} = g_{ij} + h_{ij}$, where $0 \le g_{ij} \le u_{ij} - f_{ij}$ and $0 \le h_{ij} \le f_{ji} - l_{ji}$. I get the minimum flow f^m by $f^m_{ij} = f_{ij} + g_{ij} - h_{ji}$.

f^m is still a feasible flow, because $f^m_{ij} \le u_{ij}$ and $f^m_{ij} \ge l_ij$:

$$\begin{aligned} g_{ij} &\le u_{ij} - f_{ij} \\ f_{ij} + g_{ij} &\le u_{ij} \\ f^m_{ij} = f_{ij} + g_{ij} - h_{ji} &\le u_{ij} - h_{ji} \le u_{ij} \end{aligned}$$

$$\begin{aligned} -h_{ji} &\ge l_{ij} - f_{ij} \\ f_{ij} - h_{ji} &\ge l_{ij} \\ f^m_{ij} = f_{ij} + g_{ij} - h_{ji} &\ge l_{ij} \end{aligned}$$

In addition, f^m must be a minimum flow, because, otherwise, the flow could be reduced further by finding some path from t to s in G, which corresponds to finding some augmenting path in G'' from t to s, which would mean that the flow f'' isn't maximum. Since the flow f'' is maximum, the flow f^m is minimum.

Part (b)

I show that the minimum value of all feasible flows from node s to node t equals to the maximum lower bound on cut capacity of all s-t cuts.

Let $L(S) = \sum_{(i,j) \in S \times T} l_{ij} - \sum_{(i,j) \in T \times S} u_{ij}$ be the lower bound on the cut capacity of an s-t cut S.

I show that $\min_f |f| = \max_S L(S)$ by showing that $\min_f |f| \geq \max_S L(S)$ and $\min_f |f| \leq \max_S L(S)$.

I show that $\min_f |f| \geq \max_S L(S)$, by showing that $|f| \geq L(S), \forall f, \forall S$. Indeed, suppose we have a feasible flow f and an s-t cut S. Since the flow f is feasible, it must satisfy the minimum capacities, so the flow from S to T is at least $\sum_{(i,j) \in S \times T} l_{ij}$. Since the flow f is legal, it must satisfy the maximum capacities, so the flow from T to S is at most $\sum_{(i,j) \in T \times S} u_{ij}$. Therefore, the flow across the cut is $|f| \geq \sum_{(i,j) \in S \times T} l_{ij} - \sum_{(i,j) \in T \times S} u_{ij} = L(S)$.

I show that $\min_f |f| \leq \max_S L(S)$ by contradiction. Suppose that we have a minimum flow f and that, for the sake of contradiction, $f > \max_S L(S)$. Then, for all cuts $f > L(S) = \sum_{(i,j) \in S \times T} l_{ij} - \sum_{(i,j) \in T \times S} u_{ij}$, so that, for all cuts, either we are sending more than the minimum flow from S to T or we are not sending the maximum amount back from T to S. This means that in the residual graph from part (a), there is extra capacity available from T to S on all cuts, which means, by the max-flow min-cut theorem, that there is an augmenting path in the residual graph from t to s, so f is not a minimum flow. Contradiction.

Part (c)

I develop a flow-based algorithm for identifying the minimum number of students needed to cover all the lectures. Construct a minimum flow network G as follows (all edge capacities are ∞):

- For each lecture i: add 2 vertices, in_i and out_i, and add an edge from in_i to out_i with a lower bound of 1.
- For each pair of lectures i and j such that $b_i + r_{ij} \leq a_j$: add an edge between out_i and in_j.
- Add a source s and, $\forall i$, edges from s to in_i.
- Add a sink t and, $\forall i$, edges from out_i to t.

To solve the problem, simply search for a minimum flow between s and t. The value of the flow corresponds to the minimum number of students needed to cover all lectures.

Problem 5

Part (a)

I construct a bipartite matching graph with n professors and n students. There is a source s, with an edge from s to each professor, and a sink t, with an edge from each student. Each professor has d outgoing edges to different students and each student has d incoming edges from different professors. All edges have unit capacity.

I show that the minimum cut of this graph is n, indicating that there are n pairs of professor/student, so one can schedule a single slot in which every professor is meeting with a different student.

I show that the minimum cut has value at $\leq n$, by demonstrating a cut of value n: the cut where S consists of only the source. Indeed, this cut has value n because there are links from the source s to each of the n professors in T, and no other links across the cut.

I show that, in general, the value of a cut is $\geq n$. Suppose a cut where S has the source, i professors and j students. The value of this cut is $(n - i)$ [for the links between s and the professors in T] plus j [for the links between the students in S and t] plus the number of cross links (i.e. links between professors in S and students in T and professors in T and students in S). If $j \leq i$, in the best case (smallest number of cross links), all j students in S are connected with professors within S and all $n - i$ professors in T are connected with students within T and the remaining $i - j$ professors in S have links to the remaining $i - j$ students in T, so the number of cross links is $\geq d(i - j)$. Therefore, the cut value is $\geq (n - i) + j + d(i - j) \geq n + (d - 1)(i - j) \geq n$ (since $j \leq i$). The case where $i \leq j$ is symmetric.

Since the value of the minimum cut is simultaneously $\leq n$ and $\geq n$, it is exactly n. Therefore, one can schedule a single slot in which every professor is meeting with a different student.

Part (b)

It is possible to schedule all the meetings to take place in d time slots. I start with the graph with the n professors and n students, each professors with links to d students, and each student with links to d professors. I find n matching pairs of professors and students for the first time slot, and delete those n edges from the graph. Since each professor and each student loses an edge (because each one is matched), each professor now links to $d-1$ students and each student to $d - 1$ professor. Now, I can find matches for the second time slot. After the dth time slot, each professor links to 0 student, and so all the meetings will have been scheduled in d time slots.

Part (c)

I consider an arbitrary set of desired meetings. Let s be the number of links of the professor or student with the most number of links. I show that it is possible to arrange all meetings in no more than s time slots by transforming this problem into the problem of part (b) by adding "dummy" nodes and "dummy" edges.

I add as many "dummy" nodes as necessary to make the number of students and the number of professors equal. I also want $d = s$, so I go through each professor and, if it has less than s nodes, I add as many "dummy" edges to students, each time choosing a student with less than s connections. By this process (I allow multiple edges between the same pair of professor / student, only one of which might be not "dummy"), when there will be ns connections between professors and students, each professor will have s links to students and each student will have s links to professors.

Now, I ran my algorithm of part (b), except that when a match involves a "dummy" node or a "dummy" edge, I schedule no meeting.

Part (d)

We saw in class that we can perform bipartite matching using blocking flows in $O(m\sqrt{n})$ time. Here, we're performing s bipartite matching sequentially, so we can achieve a running time of $O(sm\sqrt{n})$.

Problem 6

I develop an efficient algorithm for deciding if a team can still win the league pennant. I construct a maximum flow problem with node capacities. Of course, as seen in Problem 3, this problem can easily be transformed into a standard maximum flow algorithm, but I will leave it in the form of a problem with node capacities for the sake of clarity.

Suppose that I am interested in knowing whether team x can still win the league pennant. I calculate the maximum total number of games that team x can win: $W = w_x + \sum_i q_{ix} + \sum_j q_{xj}$. For each team i other than x, I calculate $w'_i = W - w_i - 1$. If any $w'_i < 0$, then team x cannot still win the pennant. Otherwise, I construct a graph G with node capacities as follows (all edges have infinite capacities):

- Add a source s and a sink t.
- For each team i other than x: add a node i with upper bound w'_i and add an edge from i to t.
- For each $q_{ij} > 0$ where $i \neq x$ and $j \neq x$: add a node v_{ij} with upper bound q_{ij} and add edges from s to v_{ij}, from v_{ij} to i and from v_{ij} to j.

Now, I look for the maximum flow f in G between s and t. If the value of this flow $|f| = \sum_{i\neq x, j\neq x} q_{ij}$, then team x can still win the pennant. On the other hand, if $|f| < \sum_{i\neq x, j\neq x} q_{ij}$, then that means that all the games cannot be played while still satisfying the w'_i's constraints, so team x cannot win the pennant no matter what.

Analysis

Let there be k teams. Then, the number of nodes in the graph is $n = O(k^2)$ and the number of edges in the graph is $m = O(k^2)$. By using dynamic trees, maximum flow can achieve a running time of $O(mn \log n)$. So this algorithm can run in $O(k^4 \log k)$.