HW7B

Section II

Yanhao Wang 175002614

Zheng Qi 174003950

Mohan Xiao 174005865

7.18

(a) We can add a other source node S which connect to all the other source nodes, and add a other sink node which is only connected by other sink nodes. By this way, we can just simply transfer the graph G which contains sources and sinks to a new Graph G_i which just contain one source node S and one sink node T, the problem is just like original max-flow problem.

(b) we can add a new edge to each node, for example, if there is a edge (u,v) which contains two nodes u and v. Because each vertex also has a capacity on the maximum flow that can enter it, so we can just add a new edge and a new node to each node, (u,v) become ((u_in, u_out),(u,v),(v_in,v_out)) to make a new graph G, and we can use the algorithm in the new graph G.

(c) we can just add a new constraint which limits the lower bound to the origin problem when using the linear programming. $f >= L_i \text{ (i is one of edge, L\_i the lower bound of i)}$

(d) we can just use the same meaning as (c), the constraint of the flow will change to

$\sum_{itou}(1 - e_v) * f_i = \sum_{zoutu} f_z$, so we can just use the linear programming to solve this.

7.19

with a solution to a max-flow problem on some network, we can just get the complementary graph G which get out all the path included in the max-flow solution from the original graph, then we can use DFS to check if there is any path from S to T, if it exist, then the solution is wrong. Else, the solution is right.

7.23

The minimum vertex cover can be solved the same way. Solve for the maximum matching. Consider all of the vertices in the graph besides s and t. Call the set of these vertices that are an endpoint of an edge in the maximum matching A, and call the set of vertices that are not endpoints of any of the maximum matching edges B.

There can be no edges between distinct elements in B. Assume for contradiction that there was. That edge could then have been added to increase the size of the maximum matching, because it runs between two vertices that were untouched by the maximum matching edges. This contradicts our assumption that the matching found was the maximum matching, so it is impossible to have edges between distinct elements of B.

Therefore, all edges in the graph must either be included in the maximum matching or must be be- tween a vertex that is an endpoint of a maximum matching edge and a vertex in B. Note that it is also impossible for a maximum matching e dge to have both endpoints have edges to elements in B, otherwise the size of the matching could have been increased by removing the edge from the maximum matching and adding the two edges from its endpoints to the vertices in B. Therefore, at most one endpoint vertex of every maximum matching edge can have an edge to a vertex in B.

Therefore, when constructing the minimum vertex cover, it is sufficient to take every maximum matching edge and choose one of its endpoints for the minimum vertex cover. Specifically, if one of the endpoints has and edge to a vertex in B, choose that vertex. Otherwise, it doesnt matter which one you choose.

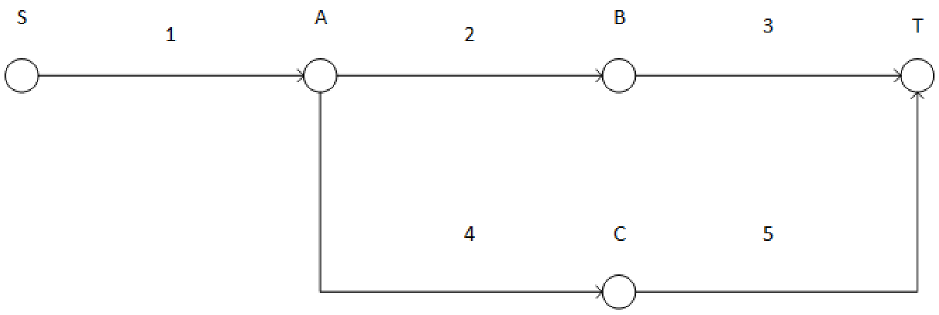
Because an endpoint was selected from every edge in the maximum matching, then the vertex cover must cover all of the maximum matching edges.

Furthermore, if a vertex had an edge to an element in B, then that vertex was chosen. There can be no edges between elements in B, as we proved earlier.

Therefore, the vertex cover chosen in this way must cover every edge in the graph. Furthermore, it must be minimal, as removing any of the vertices would mean that neither endpoint vertex of one of the maximum matching edges was in the set of vertices, because only one endpoint vertex was chosen per maximum matching edge. This means that the resultant set of vertices would not cover every edge in the graph. Therefore, if we can solve maximum matching, we can solve minimum vertex cover. Since maximum matching is solved by max flow, then minimum vertex cover reduces to max flow.

7.28

(a) Take the below graph as example. Since size(f) = 1, if we send the flow only through the shortest path (i.e. through S-A-B-T), $\sum f_e l_e = shortest\ path$. If we send flow both through shortest path and other path, or even only through other path(i.e. through S-A-C-T), $\sum f_e l_e$ will be larger than the value of the shortest path. Thus, the shortest path $p_s \leq \sum f_e l_e$. When flow only go through the shortest path, equality holds.



(b)

$$
\begin{aligned}
& min \sum l_e f_e \\
& \sum f_{s,v} = 1 \\
& \sum f_{v,t} = 1 \\
& \forall u \in V, u \neq s, u \neq t \\
& \quad \sum f_{u,v} - \sum f_{u,v} = 0, (w,u) \in E, (e,v) \in E \\
& \forall e \in E : f_e \geq 0
\end{aligned}
$$

(c) In the above linear program, every vertex has a constraint. For each vertex v, multiply x_v on both side of condtraints. Then add up all |V| equations, we'll get $\sum(x_u - x_v)f_{u,v} = x_s - x_t$. Since $x_u - x_v \leq l_{uv}$, so $\sum(x_u - x_v)f_{u,v} = x_s - x_t \leq \sum f_e l_e$. Thus it's the dual LP for the above LP. Or in another approach, we can write $x_s - x_t$ as below:

$$
\begin{aligned}
& x_s - x_{p_1} \leq l_{sp_1} \\
& x_{p_1} - x_{p_2} \leq l_{p_1 p_2} \\
& x_{p_3} - x_{p_3} \leq l_{p_2 p_3} \\
& \ldots \\
& x_{p_n} - x_t \leq l_{p_n p_t}
\end{aligned}
$$

adding all the above inequalities up, we can get $x_s - x_t \leq \sum l_p$, where l_p is all the edges along a path from S to T. Since we want to find the maximum of l_p, we have to find the lowest lowerbound $\sum l_p$ which is exactly the same as finding the shortest path in the graph.

(d) The difference is that the example on page 209 is trying to find the shortest path in an undirected graph, thus, the constraints are $|x_u - x_v \leq w_{uv}|$. Whereas this problem is dealing with a directed graph.

7.31

(a) If the algorithm happened to find path S-A-B-T first, it will send a flow of 1. And then at next step, it could choose path S-B-A-T, and send a flow of 1. This flow over A-B and B-A might last for 1000 iterations for S-A or S-B to achieve capacity.

(b) Using the idea of Dijkstra's algorithm, when finding the path, update the vertex's value as the smallest capacity along the path reaching it from S. And when choosing the next node, always choose the one that its edge has the largest capacity. In this way, we can compute the fattest s-t path.

(c) Max flow is equal to mimum cut. Every cut is represented by an edge, and all the flows on these edges form the max flow. Since there could be no more than $|E|$ cuts, there will be at most $|E|$ individual flows from s to t that form the max flow.

(d) As proved in (c), if the max flow is the sum of individual flows along at most |E| paths from s to t, the fattest flow assigned must be larger or equal to F/|E|. So let f_{t+1} and f_{t} be the max flow in the residual graph after t and t+1 iterations, we then can have the relationship between them:

$$c_{t+1} \leq c_t - \frac{c_t}{|E|}$$

And this recurrence relation will take $O(|E|logF)$