Yinfeng Cong
CS 344
mid 2
Problem 1. (a).

We have known that $(S, V-S)$ is a cut with zero cut edges. This means that G is an unconnected graph, and it has two components S and V-S. (we have known this from lecture video) [if G has a cut, but no cut edges, then it's unconnected] So, we have $u \in S$ and $v \in V-S$, and they are from two compoments. When we connect them, which means we connect the graph and there's one cut edge now.

We want to prove there's no cycle. So start from one vertex a in S and go to its neighbors abitrarily (but try to get to vertex u) until it gets to u. Then we repeat this process. go the edge (u, v), and from v to other vertex in V-S. Since (u,v) is the only ~~edge~~ cut edge of $(S, V-S)$. So, it won't go back from V-S to S by other cut edges. It means that if we pass (u, v) we'll never go back to the previous part. So there will not be a cycle.

Problem 1 (b).

For any arbitrary MST T of G and suppose by contradiction that f is a part of T.

Then we add f to T, which is $T+f$. Since T is a tree, so $T+f$ will have one cyle. Let e be the heaviest weight of this cycle (except f) and $w_f > w_e$.

Now consider the subgraph $T+f-e$, it has $n-1$ edges and is connected. And it has no cycle (it's a spaning tree) because we move an edge from it. Since $w_f > w_e$. weight of the spanning tree is strictly larger than T, a contradiction with T being a MST of G that contains the edge f.

Problem 2 (a).

Algorithm: ①. Initialize an array count [1:n] with 0.

②. create a queue data structure Q and insert s to Q

③. While Q is not empty:

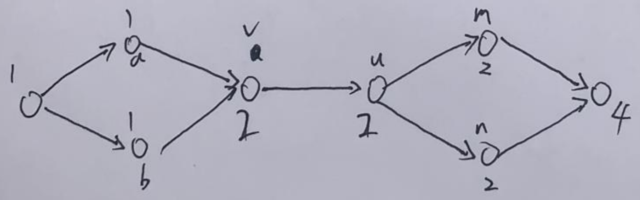
(a) Let v be the first vertex of Q and dequeue this vertex from Q

(b). let count [v] = count [v] +1 and for $u \in N(v)$. insert u to the end of Q.

④. After we go through the graph.

```
if count [u] = count [v] {
    if count [v] > count [ preneighbor of v ] {
        if count [u] = count [N(u)] {
    then the edge uv is a bottleneck edge.
        }
    }
}
```

example



count [v] = count [u] here.
and count [v] > count [b] / count [a]
count [u] = count [m] / count [n]
So uv is a bottleneck edge.
I tried many examples and finally find this works.

(b). Proof of correctness:
This question is a kind of transformation of BFS, we change mark [1:n] to count [1:n] and that's the only difference for the above part. Also, when we are in BFS, we mark the correctness of s to make sure if its passed yet. Here we can pass a vertex several times and record the value as the time we passed this vertex. Then, we know that bottleneck edges are the edges that connect two part of the graph (if this edge is deleted, the graph will be disconnected) So if we pass a bottleneck edge e, we will go through the maximum time of ~~we go~~ what we go in the previous graph. So count of bottleneck edge's vertices are bigger than the previous ones. Also, bottleneck edge is the edge that we have to go.

So there will not be an edge x parallel to it ~~a~~ (which x and e directs to the same edge is impossible) So Count of bottleneck edge must be equal to the count of its neighbor's vertices. At last, to make sure it is a bottleneck edge, the two vertices of it must have the same count (if first ~~edge~~ vertex < second ~~means~~ that there's another vertex direct to second vertex, then this edge will not be a edge that has to be passed through) As we make the ~~three~~ factors work, we prove the correctness of this problem.

(c). runtime analysis:

Let n be the number of vertices and m be the edges, we have O(n) and O(m). (This is just the runtime of BFS) . So the total run time is O(m+n).

Problem 3 (a). ①. we construct a weighted undirected graph $G = (V, E)$

Algorithm: ① Sort the roads in decreasing order of their costs.

② . Let Cost = sum of all the c_e.

④. For $i = 1$ to m :

if delete e_i from the roads makes there's no cycle in the roads, let Cost = Cost − c_i, and stop the for loop.

Otherwise, delete e_i and Cost = Cost − c_i

⑤. return Cost

(b). Proof of correctness:

To make the city connected and has the minimum cost, we need to make the cycle becomes an spanning tree. Among all spanning trees, MST will have the minimum cost by definition. So we just remove edges to make the cycle becomes an MST, thus proving the correctness of the algorithm. (we have know Kruskal's Algorithm and MST in lectures)

(c). Runtime analysis: We create G in $O(m+n)$ time (it has m edges and n vertices). The runtime of Kruskal's algorithm is $O(m \cdot \log m)$.

Since $m \geq n-1$, we have runtime is $O(m \log m)$

Problem 4. (a)

Algorithm: We run Dijkstra's Algorithm from s to t, and get shortest path named dist [s,t] (in G)

And we recorded all the vertices that has been passed.

For i = 1 to k,

we get L[i], and its weight is w_i. We set the vertices of L[i] are x and y

If vertice x and y is also in the dist[s] and there's no path between x and y,

we calculate the $\boxed{\text{sum of weight from } x \text{ to } y}$ in dist[s], and name it as "cost." (weight of xm, mn, ... zy, many edges)

let $d_{new} = |cost - w_i|$, if the new d is bigger than the old one, then $d = d_{new}$ (otherwise, $d = d_{old}$) [because d[t] = d[t] - cost + w_i, so the smaller w_i - cost is, the smaller d[t] is]

the old one

Then return L[i] as the edge should be added.

(b) Proof of correctness:

In lecture 10, we have proved the correctness of dijkstra's algorithm by induction. So we know dist [s,t] is the original shortest path. Here we want to have an edge xy to replace the old shortest path, to make a new shortest path dist [s,x] + dist [y,t] + w_{xy}. dist [s,t] can be written as dist[s,x] + dist[y,t] + dist[x,y]

So the different part is w_{xy} and dist [x,y]. So what we need to do is make sure $w_{xy} \leq$ dist [x,y]. So when we find we have edge xy in L[i] we can compare w_{xy} and dist [x,y] to see if it can make a new shortest path.

Also, to make sure we pick the best L[i], we need to make sure we decrease the weight as much as possible. So if we find another Edge in L (we call it ab) and dist (s,a) + w_{ab} + dist (b,y) $\leq$ dist [s,x] + dist [y,t] + w_{xy} then we change the goal as edge ab. By iteration of this, we will get the edge that have the shortest path. Hence, the correctness is proved

(c) runtime analysis:

Runtime of dijkstra's algorithm is $O(n + m\log m)$ and the time we go through L is $O(k)$, so total $= O(k + n + m\log m)$