1)
There is an optimal least cost path p. Assume p is length >= n
If the length of this path is >= n then it must contain a cycle.

To prove if the length of the path is >= n then it contains a cycle (well proven fact):
cite Aspnes Corollary 10.10.7. from Notes on Discrete Mathematics

Or simply (not rigorous):
There are n vertices. There are n-1 possible edges to not make a cycle. You can't fit n edges in that without making a cycle (pigeonhole principle).

So we have shown that there is at least one cycle:
We can get rid of all the cycles because they must be a 0 or positive weight cycles (negative weight cycles are not possible).
If it is a 0 weight cycle: if we remove this cycle from the path we will be decreasing the number of edges on the optimal path and not changing total weight.
If it is a positive weight cycle: removing this cycle decreases the number of edges and decreases total weight (in this case the original optimal path p was not optimal)

Therefore, when we remove all these cycles this optimal shortest path must be < n or likewise use at most n - 1 edges in making the least cost path.

2)
We will modify the graph and then run Dijkstra on the modified graph. We will also prove how this modified graph accurately represents the problem.

We will make 2 copies of the graph:
copy1 includes all the Uber edges
copy2 includes all the Uber edges (label all vertices with a ')
then we add directed edges from copy1 to copy2 as hyperloop edges

We then run Dijkstra on this modified graph going from city i to city j'. This takes O(m*log(n)).

We need to prove this models the problem correctly:
There are two stages to the question:
stage 1: you still have your coupon
stage 2: you have used your coupon

The hyperloop can only be used once. Once, one reaches the bottom part of the graph (copy2) there is no way to go back up.
The top part of the graph (copy1) is modeling when one still has the coupon. The bottom part (copy2) is when one doesn't have the coupon anymore.

3.1) A U L acyclic
A is set of connected components. L is set of min weight edges.

Assume there exists a cycle in this union on k nodes: 2 <= k <= n: ci are the components and ei are the edges in between the components. w(ei) is the weight of that edge.
c1 e1 c2 e2 c3 .... ek-1 ck ek c1

all ei are minimum weight edges between the components because for each connected component the algorithm selects the lightest weight edge.

c1 selected e1 as the minimum weight edge between c1 and c2 (instead of ek). This means w(e1) < w(ek)
c2 selected e2 as the minimum weight edge between c2 and c3 (instead of choosing e1). This means: w(e2) < w(e1) so w(e2) < w(e1) < w(ek)
c3 selected e3 as the minimum weight edge between c3 and c4 (instead of e2). This means w(e3) < w(e2) < w(e1) < w(ek)
....
ck-1 selects ek-1 (instead of ek-2): w(ek-1) < w(ek-2) < w(e3) < w(e2) < w(e1) < w(ek)
ck selects ek (instead of ek-1): w(ek) < w(ek-1) < w(ek-2) < w(e3) < w(e2) < w(e1) < w(ek)
This is a contraction. w(ek) can't be less than and greater than the weights of the edges in between in the cycle (all the weights are distinct so equality is not possible)
Therefore there can't exist a cycle in A U L.

3.2 Light Edge:
Cut Property: For any cut C, the unique lightest edge across the cut is in every MST of G. This was proven in class and is in the textbook. We will show that each new edge is the light weight edge across a cut in A. Furthermore, showing that each edge added to A by the algorithm across some cut is the lightest weight edge would imply by the cut property that A constitutes a minimum spanning tree.

Without loss of generality:
Let's say we have any cut that leaves component c0 in one partition and component c1 in the other partition.
Let an edge (u, v) be the minimum weight edge between the two components c0 and c1 coming from c0.
Let's say (u', v') is added to A by the algorithm as the minimum weight between c0 and c1.
This edge must have come from c1 because we know the minimum weight edge from c0 is (u, v). However, we know edge weights are distinct, so this also being a minimum weight edge between c0 and c1 returned from the algorithm is not possible. (u', v') would not have been added by the algorithm. (u, v) will be the only edge returned by the algorithm to cross the cut between c0 and c1. Generally, this means in any cut the algorithm returns the lightest weight edge between any two components separated by the cut. Although, the algorithm could add other light weight edges before adding (u, v), no other edge will be added between c0 and c1 other than (u, v).

Therefore, all added edges will contribute to the formation of a minimum spanning tree.

3.3
Initialize: O(m) time
minEdge[n] //size of all unique components. holds lightest edge for component i.
edge[m] = [u v w] //u and v are endpoints of an edge and w is the weight

edge list

index	endpoints	weight
e1	u1,v1	w1
...		
em	um,vm	wm

```
partitionCount = n  // union function will modify partitionCount accordingly. partitionCount is
                        the number of components.

while (partitionCount > 1) {
        L = []
        clear everything in minEdge. set it all to [empty, empty, infinity]

        for (int i = 0; i < m; i++) {
                u, v, w = edge[i];
                bool usedthisEdge = false;
                comp1 = find[u];
                comp2 = find[v];
                if (comp1 != comp2) {
                        if (minEdge[comp1][2] > w) {
                                [u, v, w] = minEdge[comp1];
                                usedthisEdge = true;
                        }
                        if (minEdge[comp2][2] > w) {
                                [u, v, w] = minEdge[comp2];
                                usedthisEdge = true;
                        }
                }
                if (usedthisEdge) {
                        edge[i][2] = infinity //set this edge weight to infinity if it is used so we
don't use it twice (essentially edge deletion)
                }
        }
```

```
        for (int i = 0; i < n; i++) {
                if (minEdge[i][0] == empty) {
                        continue;
                }
                u, v, w = minEdge[i]
                add {u,v} to L
        }

        for (u,v) in L {
                if (find[u] != find[v]) {
                        union(find(u), find(v)) //relies on something consistent like attach the
                                                smaller size to the larger size
                        //union should decrease partitionCount accordingly
                }
                add {u,v} to A
        }
}

return A
```

Proving this runs in O(m*log(n)) time:

Basic idea:
The outer while loop will only iterate log(n) times
The beginning inner part up until union will take m time (we are looping through the entire edge set every time inside)
This section is clearly m*log(n) time

The union-find part takes n*log(n) over the whole algorithm.
O(mlog(n) + nlog(n) over the whole algorithm) = O(mlog(n))

Proof:
To show the outer while loop will only iterate log(n) times

In the beginning, there are n distinct components.
Let's say at each iteration the number of connected components is ci.

1) Each edge that you add reduces the number of connected components by 1. This is because each edge covers 2 separate connected components (making them one connected component).
2) The number of unique edges that you add in any iteration is at least ci/2. Worst case of ci/2 components left: every component is connected to a partner and that partner is connected to that component.

After each iteration, there are at most half the number of connected components (at most ci/2). Statements 1 and 2 can prove this.
In the beginning, there are ci components. With each edge added, subtract 1 from total number of components. There are at least ci/2 edges that can be added. This leaves at most ci/2 remaining connected components after each iteration.

Therefore, the size of the number of components is halved each time so there can only be log2(n) iterations through the while loop.

It's clear to see how there is a for loop that can only run m times to set minEdge and n times to use minEdge (so max m*log(n)) for this part.

The union-find part takes n*log(n) over the whole algorithm as shown in class.
This can be explained through the worst case:
Say we have the worst case situation in every loop. Every component pairs with itself so there are only half the components left after each iteration.

If this is the case:
There are n components to start. L will worst case first have a size n/2. Union takes log(n) time and find takes O(1) time so this takes
n/2 * log(n) time
n/2 components remain now and L will then have a worst case size n/4.
n/4 * log(n) time then

n/2 * log(n) +
n/4 * log(n) +
n/8 * log(n) +
...
= n * log(n) time
This can be seen because 1/2 + 1/4 + 1/8 + ... converges to 1.

O(mlog(n) + nlog(n) over the whole algorithm) = O(mlog(n))