Problem 1 (DPV 4.11)

First, we will implement a modified version of Dijkstra's algorithm. We will make it such that the algorithm not only returns dist() values, but also paths of the cycles if they exist. This can be done easily because Dijkstra's algorithm already keeps in track of the vertices that had visited, so we introduce a queue that can store the previous vertices for each path and remember them. For instance, if the algorithm goes through u -> a, a -> b, b -> u, then we determine that there exists a cycle with some dist() length and we store and output the path u -> a -> b -> u. Repeat these steps for every vertices (actually just the source nodes) and find cycles. Finally return the cycle with the shortest dist().

We will run the Dijkstra's algorithm on all the source nodes, therefore the runtime takes $O(|V|^2 * |V|) = O(|V|^3)$
Implementing queue and doing operations (enqueue, dequeue, etc) takes constant time, therefore this doesn't affect the overall runtime.

Problem 2 (DPV 4.13 all parts)

(a) Using DFS, we can find whether there is a path from s to t. When the algorithm is run, we set a variable *l* that is initialized to the size of the tank. During the traversal, whenever we encounter an edge whose weight is greater than the size of the tank, we discard the path and search for another one, again using DFS. Repeat this process and check if there is a feasible route from s to t. If not, then return false. Since we employ the DFS algorithm, the runtime is essentially O(|V|+|E|).

(b) In order to find every feasible path, we will run a greedy algorithm, more specifically Dijkstra's algorithm, with a bit of modification.

--- part of Dijkstra's algorithm from the textbook (pg. 110) ---

```
H = makequeue(V)
While H is not empty:
   u = deletemin(H)
   for all edges (u, v) ∈ E:
      if dist(v) > dist (u) + l(u, v):
         dist(v) = dist(u) + l(u, v)
         prev(v) = u
         decreasekey(H, v)
```

Let p_i indicate every feasible path from *s* to *t*. When the above algorithm runs, we will store a value e_i for each path p_i such that e_i is the weight of the longest edge in each p_i. Then once the algorithm is done running, we will return the minimum of all the e_i's, that is going to be the minimum fuel tank capacity required to get to a farthest city while traveling from s to t.

Problem 3 (DPV 4.20) -undirected graph, efficient algo

The underlying idea is that we will compute the distances between u,v, and all the other reachable nodes from them. Written more formally, we compute the dist(u -> v_i) and dist(t -> v_i) where $v_i \in V$ are all the vertices such that there exists a path from u -> v_i and t -> v_i, respectively. And, we also remember the distances of each e'. Let each $e_i' \in E'$ be written as $e_i' = (a_i , b_i)$. Then, we will calculate the net distances of u -> a_i -> b_i -> t by doing dist(u -> a_i) + dist(a_i -> b_i) + dist(b_i -> t). This gives us the distance from u to t when each of e_i has been added to the graph.

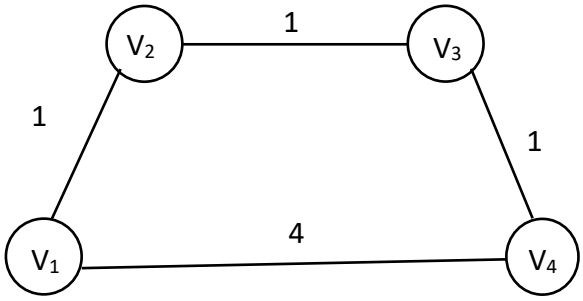
Finally, we just return the path that has the greatest $\{dist(u \rightarrow t) - dist(u \rightarrow a_i \rightarrow b_i \rightarrow t)\}$ value i.e., addition of the e' that resulted in the maximum decrease in the driving distance between two cities u and t in the network.

For computing dist(u -> v_i) and dist(t -> v_i), Dijkstra's algorithim can do the job by running twice, *dijkstra(G, l, u)* and *dijkstra(G, l, t)*. Additionally, we compute $|E'|$ many times in order to get the maximum $dist(u \rightarrow t) - dist(u \rightarrow a_i \rightarrow b_i \rightarrow t)$ value, which will aggregate to the initial Dijstra's algorithm. Therefore, this solution has an efficient runtime of $O(|V|^2)$ overall.

Problem 4 (DPV 5.5) all parts

(a) No. Whether you use Kruskal's or Prim's algorithm, the process of selecting edges does not get affected. For instance, say you have two edges e_i and e_j. Let $e_i < e_j$. Then, the equivalence relation $(e_i + 1) < (e_j + 1)$ still holds true. Therefore, under any circumstances, both Kruskal's algorithm and Prim's will choose $(e_i + 1)$ over $(e_j + 1)$ in their edge-selection processes.

(b) Yes, the shortest path can change.



For example, the initial shortest path from v_1 to v_4 is $\{v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4\}$ and the distance is 3. However, once we increment all the distance by 1, we can now see that the path $\{v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4\}$ yields the distance of 6. Then, the shortest path now will be directly from v_1 to v_4 whose distance is 5 (4+1). Therefore, yes.

Problem 5 (DPV 5.6)

Proof by contradiction. Let there be graph G with all edge weights being distinct. Also, assume there exist two unique MSTs. $M = (V_M, E_M)$ and $N = (V_N, E_N)$. Let's find an edge e' of the graph N such that e' has the minimum weight out of all edges in E_N that doesn't belong to M. Notice $e \in (E_N \setminus E_M)$. If you add e' to M, then this results in another MST because there must be an edge in E_M that is greater than e' (since each edge length is unique). This results in a contradiction because this shows how M was not a valid MST in the first place. Therefore, M = N should be accepted to be true.