Homework 7 - Solution

C-15.6

The main modification in this case is to implement the priority queue, Q, as an unordered doubly linked list. This allows us to insert and update the key for any vertex in O(1) time and remove the minimum in O(n) time. We perform O(m) insertions and updates and O(n) removeMin operations in the algorithm. Since m is O(n^2), the total running time in the worst case is O(n^2).

Alternate Solution:

We have studied Prim-Jarnik algorithm (Algorithm 15.8) that has running time $O(m \log n)$ (Theorem 15.6) because we are implementing priority queue as a heap that can extract vertex u in each iteration in $O(\log n)$ time. We can implement Prim-Jarnik algorithm in $O(n^2)$ time by representing the graph $G = (V, E)$ as an adjacency matrix A. So, to find adjacent vertices of vertex u, we need to search the row of u in the adjacency matrix. We assume that the adjacency matrix stores the edge weights and unconnected edges have weight 0. Below is the modified Prim Jarnik's algorithm.

Algorithm modifiedPrimJarnikMST(G, r):
Input: A weighted connected graph $G(V, E)$ and r is the set of vertices in the current tree.
Output: A maximum spanning tree T for G

```
$for\ each\ u \in V[G]\ do$
$key[u] \leftarrow \infty;$
        $\pi[u] \leftarrow NIL;$

$key[r] \leftarrow 0;$
$Q \leftarrow V[G];$
$while\ Q \neq \emptyset\ do$
        $u \leftarrow EXTRACT - MIN(Q);$
$for\ each\ v \in V[G]\ do$
        $if\ A[u,v] \neq 0\ and\ v \in Q\ and\ A[u,v] < key[v]\ then$
                $\pi[v] \leftarrow u;$
                $key[v] \leftarrow A[u,v];$

$return\ the\ tree\ T$
```

Running time: $O(n^2)$ because outer loop as well as inner loop has $|V|$ variables. i.e. total number of vertices here n.

Note:
There are several ways to implement Prim-Jarnik's algorithm in $O(n^2)$.

1. Implementing priority queue as above, assuming G is a connected graph or there is a path from a vertex v to all other vertices.
2. Converting adjacency matrix to adjacency list representation in $O(n^2)$ time.
3. Using an array so that each time while extracting the minimum, we need to scan whole array once which takes $O(n)$ time. So, the total running time will be $O(n^2)$.

A-15.1
This similar to finding out the minimum spanning tree, the difference here is we will pick up the vertex with the highest weight edge instead of the one connected by the lowest weight. We will modify Kruskal's algorithm to generate a maximum spanning tree.

Algorithm maximumSpanningTree (*G*):
Input: A simple connected weighted graph *G* with *n* vertices and *m* edges
Output: A maximum spanning tree *T* for *G*

```
$for\ each\ vertex\ v\ in\ G\ do$
    $Define\ an\ elementary\ cluster\ C(v) \leftarrow \{v\}$
$Let\ Q\ be\ a\ priority\ queue\ storing\ the\ edges\ in\ G, using\ edge\ weights\ as\ keys$
$T \leftarrow \emptyset$
$while\ T\ has\ fewer\ than\ n-1\ edges\ do$
    $(u, v) \leftarrow Q.removeMax()$
    $Let\ C(v)\ be\ the\ cluster\ containing\ v$
    $Let\ C(u)\ be\ the\ cluster\ containing\ u$
    $if\ C(v)\ != \ C(u)\ then$
        $Add\ edge(v, u)\ to\ T$
        $Merge\ C(v)\ and\ C(u)\ into\ one\ cluster$
$return\ tree\ T$
```

The only change is that the priority queue is now a max heap instead of a min heap and thus the running time will be similar to that of Kruskal's algorithm. Thus, the running time will be $O((n+m)\log n)$, which is $O(m \log n)$.

Alternate Solution:

We can perform a simple transformation of the edge weights to turn this into a minimum spanning tree problem. Namely, replace each edge weight, w(e), with -w(e). Now a minimum spanning tree for the transformed graph will be a maximum spanning tree for the original graph.

A-15.4
Here, we need to reconstruct a minimum spanning tree for the computer network. Since the weight of the edge *e* has been reduced, we need to find out the edge should be removed from the tree and push edge *e* to the MST. We will have one cycle in the tree, then we have to use the Cycle Property of an MST, to remove one edge from the tree *T*.

We can traverse all the *m* edges in MST to find out all the possible edges and record their weight to be reconsidered. This will take $O(m)$ time. We replace the greatest edge in the cycle with *e*. Now we need to generate a new tree. Since there are *n* vertices in the graph, this step takes $O(n)$ time. So the total running time will be $O(m+n)$.

C-16.3
This problem can be solved using the Dijkstra's algorithm because maximum residual capacity is the same as minimum weighted path.

We keep a variable $d[v]$ associated with each vertex. We initialize the d values of all vertices to 0, except for the value of the source (the vertex corresponding to a) which is initialized to ∞. Assume that we have an edge (u, v). If $min\{d[u], w(u, v)\} > d[v]$, then we should update $d[v]$ to $min\{d[u], w(u, v)\}$ (because the path from a to u and then to v has bandwidth $min\{d[u], w(u, v)\}$, which is more than the one we have currently). The total running time is $O\ ((n + m) \log n)$.

Alternate Solution:

The residual capacity of an augmenting path is the minimum capacity of one of its edges. Thus, we are interested in finding a maximum capacity path from *s* to *t*. We can perform this computation by using a maximum spanning tree algorithm (such as in A-15.1 above), which is just like a minimum spanning tree algorithm with all the weights multiplied by -1. The path from *s* to *t* in this tree will be a maximum capacity path (using a similar argument used to prove the important fact about minimum spanning trees).

A-16.3
This problem can be solved with a modified version of the Maximum Bipartite Matching algorithm. Let *G* be a flow network with *s* and *t* as the source and terminal nodes respectively. Node *s* is directed towards *n* nodes of people with the capacity of each edge being 3. There will also be *b* nodes of different breeds of dogs presented in the problem. Each person node is directed towards the dog breeds nodes representing the kinds of dog that each person is willing to adopt with each edge has the capacity of 1. All of the dog breeds nodes are directed towards the terminal node *t* with each respective edge having the capacity equal the number of available dogs in each breed.

We can now use a standard Ford-Fulkerson algorithm to compute a maximum flow. Suppose there are *n* vertices and *m* edges in such a graph, the running time will be $O(nm)$.

A-16.7
We can use Minimum-Cost Flow to solve this problem. We create a source vertex *s* and connects to all limos (*n* vertices) giving each directed edge a cost 0, and capacity as 1. Next, create a sink vertex *t* (airport) and connect all pickup locations (*n* vertices) to *t*, giving each directed edge a cost 0, and capacity as 1.

Finally, we can set directed edges from limos to locations, $n*n$ edges with cost as d_{ij} and capacity as 1.

Thus, the problem is a minimum-cost flow and we apply MinCostFlow which runs in $O(|f|n^2 \log n)$. Since $|f|$ here is n, so it runs in $O(n^3 \log n)$.