**C-15.2**

The claim is not true. We can create a graph with a minimum spanning tree, then we delete all edges not in spanning tree. Now this graph is weighted, connected and undirected, in the same time contain a minimum spanning tree. Whatever edge in this graph now is within the tree, including the largest-weighted edge.

**A-15.2**

We can tweak the Dijkstra’s algorithm to looking for the longest path instead of the shortest path. Then the vertices are switching centers, edges are communication lines with weight of the bandwidth and we need to find most weighted path between a and b. For the algorithm, for a vertex v we use D[v] to hold the weight from a to v. Then to tweak the Dijkstra’s we initialize all vertices but a to zero and a to positive infinity. After that, for a new edge (u,v) in G, if D[v] < min(D[u], edge(u,v)), then we give the value of min(D[u], edge(u,v)) to D[v] as the current bandwidth because the bandwidth for a path is its lowest bandwidth-edge, like this: D[v] = max(0, min(D[u], edge(u,v))) to every edges. Then we need to apply D[v] = max (D[v], min(D[u], edge(u,v))) again to find the path with highest bandwidth. The algorithm is almost the same as Dijkstra’s, which should have the running time of O((n+m)logn).

**A-15.6**

Since we only have two kinds of path, thus for the priority queue operations in Prim-Jarnik’s Algorithm, we can use two linked list whereas one for 10 dollars rooms and the other one for 30 dollars rooms. In this way we can bypass the original O(logn) running time for insertion, removal and key change to O(1) for simply look them up in list. The total running time should be O(n+m) instead of O((n+m)logn).

**R-16.2**

Forward edges: (s, v2), (v2, v3), (v1, v4), (v4, t). Backward edges: (v1, v3)

For augmenting paths, we cannot use (v3, t), (v2, v5) and (v5, t), but all else are fine:

(s, v1, v4, t), (s, v1, v3, v4, t), (s, v2, v3, v4, t), (s, v2, v3, v1, v4, t), (s, v3, v4, t), (s, v3, v1, v4, t)

If we increase the flow in (s, v1, v4, t) by 2, then (v1, v4) reached full capacity and (v4, t) becomes 6/9. Then we can increase the flow in (s, v2, v3, v4, t) by 3, then (v2, v3) reached full capacity and (v4,t) becomes 9/9. Now all path to t, (v4, t), (v3, t) and (v2, v5, t), have reached full capacity, thus, we can say there are no more augmenting paths and the max flow is 9+3+3=15.

**C-16.7**

The cycles in a bipartite must be formed with even number of edges. We can use BFS on each node, and if a node has an edge that not in the BFS tree and joining this node in its depth, then the graph got an odd length cycle which makes the graph not a bipartite graph. If all vertices are traversed and no odd length cycle found, then we can say this graph is bipartite. This is a BFS for all vertices and edges thus the running time is O(n+m).

**A-16.2**

Move all residents into a group A and all puppies into a group B, then this is a bipartite flow network problem where source s points to group A, group A points to group B with at most three links, then group B points to sink t. We call this the Reduction to Maximum Flow Problem according to the course ppt page 16. Since each resident can have at most 3 puppies, (s, A[i]) has a capacity of 3, and (B[j], t) has a capacity of 1 to represent the number of dogs adopted in the flow. Then we simply apply the Maximum Flow of this bipartite graph to find the maximum number of adopted puppies in the end. Since the (s, A[i])’s capacity is fixed to 3, we can assure there are no residents exceed the law and (B[j], t) ’s capacity is fixed to 1, we can both count the puppies adopted in the end and make sure each puppy is only adopted once. The running time should be O(nm) where n is vertices in A and B and m is edges between A and B.