**R-13.7**

1. Adj Matrix uses n^2 = 1E8 space, Adj List uses n+m = 3E4 space. In this case I choose Adj List for its smaller space occupation.
2. Adj Matrix uses n^2 = 1E8 space, Adj List uses n+m = 2.001E7 space. Adj Matrix uses 5 times more space than Adj List, which is not by far. Thus, assuming the space requirement is strict, I would choose Adj List, else both is fine, and have their own advantages for specific area.
3. Adj Matrix uses O(1) for areAdjacent(v,w) operation, when Adj List uses O(min(deg(v),deg(w))). Thus, I choose Adj Matrix.

**C-13.11**

/\* forward edges, which point from a node of the tree to one of its descendants, back edges, which point from a node to one of its ancestors, and cross edges, which do neither. \*/

We need to use Euler tour to traversal T first and mark the nodes in visiting order as 1,2,3,4…. Note that if we go back to the same node we simply add that node’s number again. For example, [1,2,1,3,1,4] in tree: 1

2 3 4

Then for each vertical we mark them as Fi for first visit and Li for last visit. In this case Fi is the first number and Li is the 2nd last number. Between Fi and Li should be a sub-Set of the Euler tour. Say we got an edge [2,1], then it’s contained in the sub-Set of [Fi, Li]: [1,2,1,3,1] and in [2,1] 2 > 1, thus, we can say that this edge is a back edge.

Now we can just traversal all vertices and result in O(n) to label each vertices, then traversal through edges labeled in e(x,y), where x and y are the two vertices connected by edge e. We determine if e in [F(y), L(y)] (the first and last visits at vertical y) and if (x>y), then we can say it’s a back-edge. After the scan through we simply label the rest of the edges as cross edges. This comparing should take O(m). Thus, the total running time is O(m+n).

**A-13.6**

All we need to do is make sure the depth of the tree of stations with root as the current station is no more than 4. At the “create an empty list Li+1” in BFS Algorithm, we can check i+1 and stops when i+1=5. In this way we can make sure the Tree with current station as root would not exceed depth of 4, and so we get the set of stations using no more than 4 links.

**C-14.7**

The normal Dijkstra’s Algorithm would take O((n+m)logn) where the O(logn) comes from the insertion and removal from the priority queue. Thus, we need to modify the lookup process to O(1).

We can do this with creating a table T where T[i] is the group of vertices that have the same distance from themselves to v(the single-source vertical). In each iteration of Dijkstra’s Algorithm, we are looking for one shortest distance from nodes within the “cloud”, thus, all new nodes in each iteration have the same distance to v, which makes sense to push the new set of nodes into T[i+1]. This only takes O(1).

Thus, when we are trying to do insertion and removal from the priority queue in Dijkstra’s Algorithm we can use table T instead, which result in O(1) and makes the whole algorithm O(n+m).

**A-14.2**

We just need to adjust the weight of each edges from the distance to some 0 and 1. In this case, all safe edges should be weighted 0 and all danger edges that might be eavesdropped should be weighted 1. Then just run the Dijkstra’s Algorithm to find the shortest path. This should take O((n+m)logn).

**A-14.5**

This is a DAG problem. We only need to change the weight to the amount of gold, then apply DAGShortestPaths Algorithm. Since this Algorithm can take negative input, we can just add a minus sigh on all amounts of gold and thus find the smallest value without modify the algorithm. The running time should be O(n+m).