1. Start from the root of the tree. If none of its subtrees has size $s \geq \lfloor N/2 \rfloor + 1$, then it is obvious that the root is the separator. Otherwise, find the subtree that has size $s \geq \lfloor N/2 \rfloor + 1$. Do this recursively until we find the subtree with root $\mu$ which has size $s \geq \lfloor N/2 \rfloor + 1$, but none of the subtrees of $\mu$ has size $s \geq \lfloor N/2 \rfloor + 1$, then $\mu$ is the separator.

Suppose $\mu$ has $m$ subtrees, after removing $\mu$, the tree is divided into $m + 1$ connected components: $m$ subtrees, and the whole tree without the subtree rooted at node $\mu$. The subtrees all have size $s \geq \lfloor N/2 \rfloor + 1$. The whole tree without the subtree rooted at node $\mu$ also has size $s \geq \lfloor N/2 \rfloor + 1$ because the size of the whole tree is $N$, and the size of the subtree rooted at node $\mu$ is at least $\lfloor N/2 \rfloor + 1$.

The algorithm can be completed in linear time if we first run PostOrder to traverse the tree and calculate the size of each subtree.

2. First run PostOrder to calculate the lengths of the longest two paths from each node to the leaves (the two paths should go through different children of the node). If a node has only one child, let the second longest path be 0. Scan the tree again. This time add the lengths of the longest two paths of a node together, and get the largest value for the whole tree. This is the diameter of the tree because the longest path of a tree should be between two leaves (or between a leaf and the root), and it must contain the common ancestor of these two leaves.

The running time is $O(n)$.

3. Dijkstra’s algorithm can not be modified to solve the problem. Because it is a greedy algorithm, it will only find the local optimum. But cost variation is a global optimum. The following is an example.

![Diagram](image)

Figure 1: When a greedy algorithm is picking local optimum from $S$ to $C$, it will choose $S \rightarrow A \rightarrow C$. But the global optimum is $S \rightarrow B \rightarrow C \rightarrow T$.

4. MST is actually what we need. Suppose the maximum weighted edge of the MST is $e_1$. And the maximum weighted edge of another spanning tree $T$ is $e_2$. If $w(e_1) > w(e_2)$, we can remove $e_1$ from MST and cut the graph into two parts. From $T$, we can find an edge $e^*$
linking these two parts. As $e_2$ is the maximum weighted edge in $T$, $w(e^*) \leq w(e_2) < w(e_1)$. Remove $e_1$ from MST and add $e^*$ to it, we get a new spanning tree $T^*$. And the weight of the $T^*$ is less than MST. That’s a contradiction. So the maximum weighted edge in an MST is minimized.

So we can use any algorithm that find the MST to solve the problem.

5. (a) First construct a set $P_i$ for each vertex $i$. Then for each edge $(i, j)$, if $\text{FIND}(i) \neq \text{FIND}(j)$, Union $P_i$ and $P_j$.

(b) Union operation takes constant time, FIND operation takes $O(\log n)$ time. For each edge in the graph, the algorithm runs Union and FIND. So the running time is $O(m \log n)$.

(c) Use DFS to count the connected components. Randomly pick an unvisited vertex and begin DFS. Mark every vertex that are visited until DFS stops. The marked vertices form a connected component. Do this recursively until every nodes are marked.

The algorithm visit each edge and each vertex constant times. So the running time is $O(n + m)$.

6. (a) If the attractor is the $i$-th vertex, then the $i$-th row of the adjacency matrix are all 0’s, and the $i$-th column are all 1’s except the element on the diagonal($a_{ii}$).

(b) Start from $v_1$. Search the first row from $a_{11}$. When we meet $a_{ii} = 0$, this means $v_i$ is not a attractor. When we find the first 1 in the first row, suppose $a_{ik} = 1$, this means $a_i$ is not a attractor. Then we are sure that $v_1$ to $v_k$ are not the attractor and we then look at $v_k$. Search the $k$-th row from $a_{kk}$, and do this recursively until we reach $a_{nj}$ or $a_{jn}$.

Now we have only one possible attractor. Verify it by scan the corresponding row and column.

In the algorithm, when we look at an item other than the diagonal item in the adjacency matrix, we verify that one vertex is not an attractor. And we will not look at that vertex afterwards. So totally we scan at most $2n - 1$ items in the matrix. Then we check $2n - 1$ items to verify an attractor. So the running time is $O(n)$. 