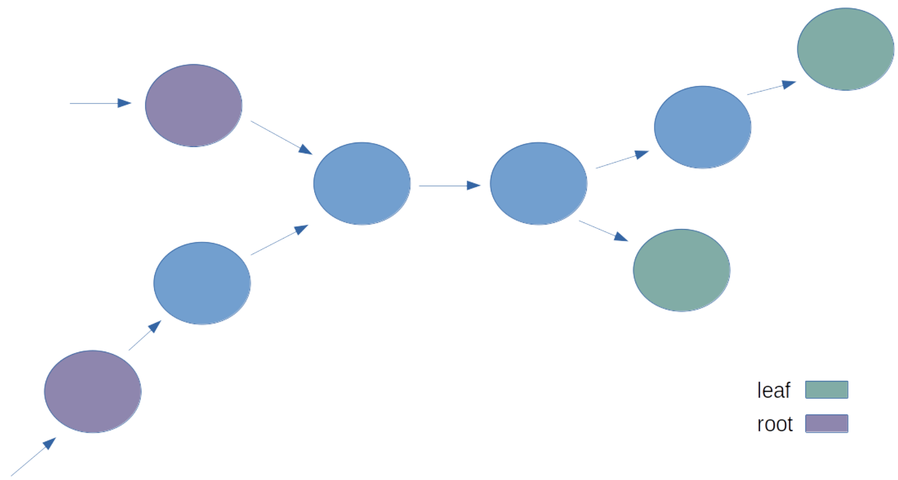
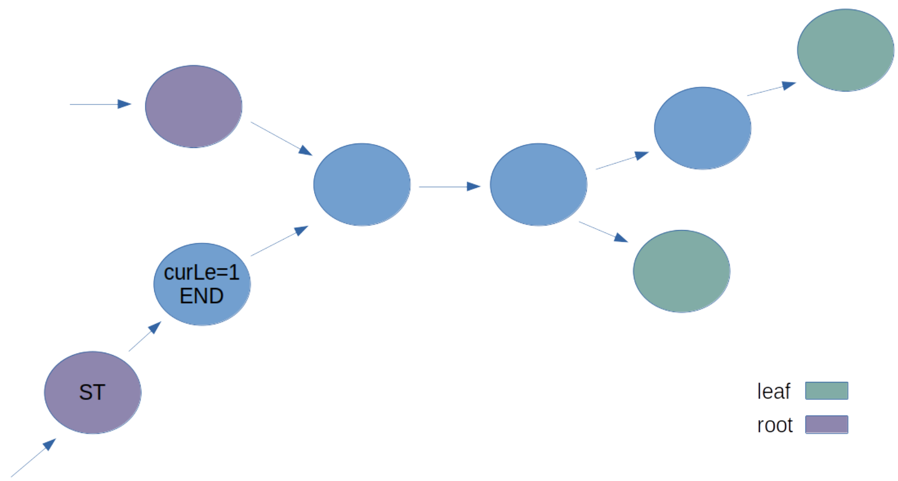
Before showing my solution, It is advised to revise backtracking as my solution is based on it. In fact, It is an enhancement of traditional backtracking where we shall prune already computed branches of the search-tree.

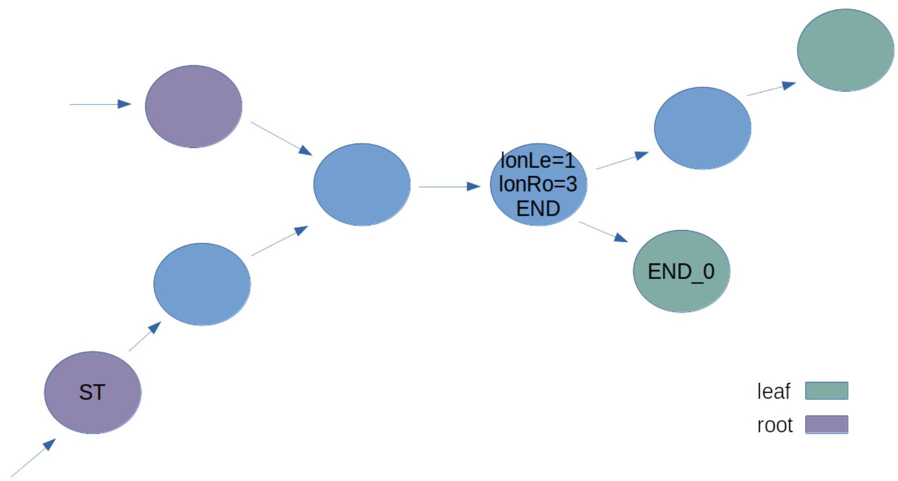


while backtracking, i.e searching within search tree, we maintain a variable *curLe* which indicates how many nodes we traversed beginning from some root. For instance, *curLe* is equal to one as it traversed only one node from root *ST*.

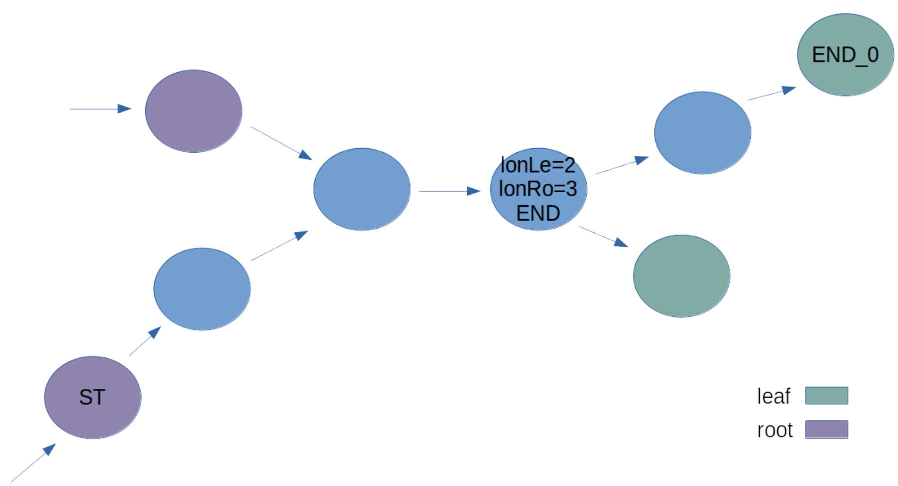


We prune search-tree branches via dynamic programming whereby we store the results of our computations. In case we needed to re-compute while going through the search-tree, We return the stored result. Particularly, Each node contains two variables, *longestRoot* and *longestLeaf*, indicating longest path length from that node to any root, and longest path length from that node to any leaf, respectively.

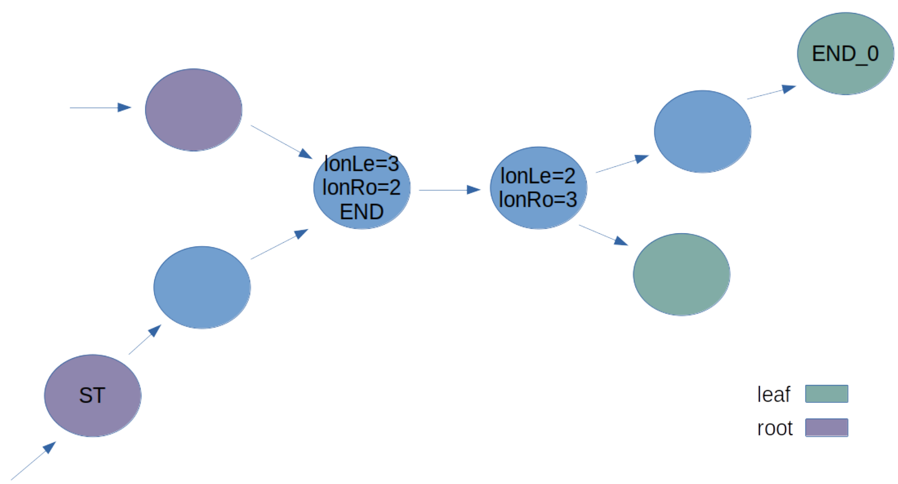
For instance, assume we started from ST root, reached *END_0*, backtracked and ended-up in *END* node. $lonLe = 1$, As there is one node we backtracked beginning from some leaf. Note that *lonLe* value in this snapshot is not the final possible value, As there are still unvisited leafs. *lonRo* is already equal to 3 via our *curLe* as illustrated in the previous figure.



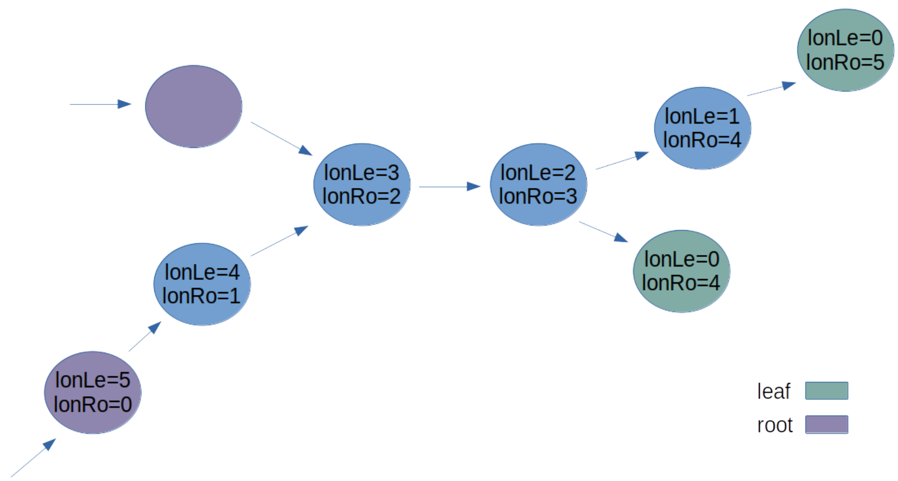
Now, Assume we visited the other leaf *END_0*, then backtracked back again to *END*. In this case we have new leaf length value. Only if it is greater than *lonLe*, we update *lonLe* to it. As we visited all node *END* children, We are sure that *lonLe* is the final correct answer.



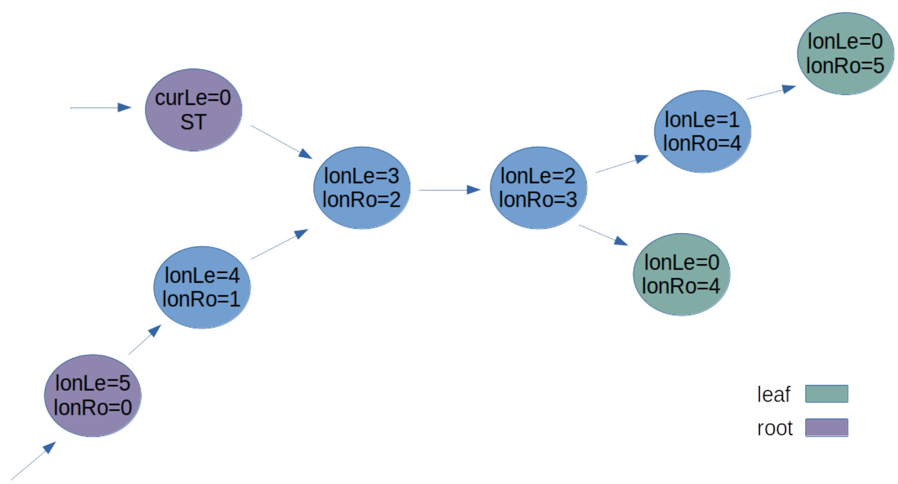
As we backtrack, we could conclude *lonLe* value by adding one. Clearly, its *lonLe* is equal to the node's ahead of it plus one. Note that, *lonRo* is already computed via *curLen* as illustrated earlier.



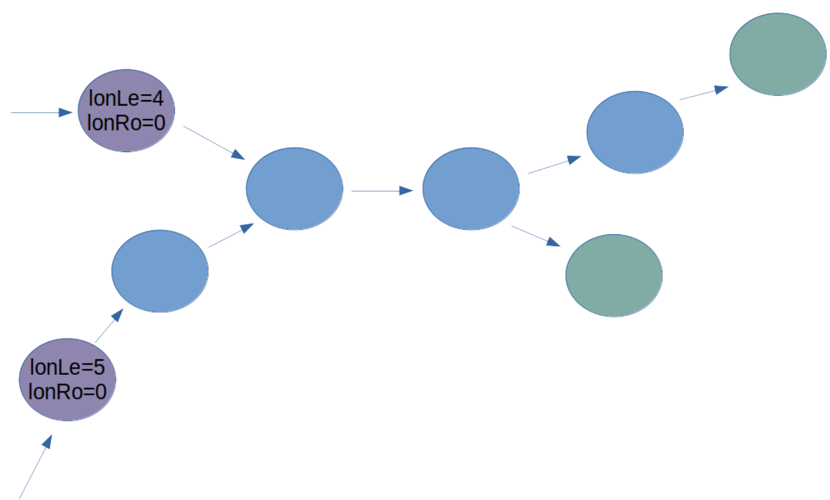
That is how our graph looks like after completing searching from root *ST*. However, That is not the end of searching, as there is still another root we need to begin searching from.



Now, Are we going to re-compute longest leaf and longest root for each node ahead of root ST again?



In fact, No. The node ahead of ST has $lonLe$ computed, So we do not need to re-compute it again. We already know the longest path from this node to any leaf is 3. In addition, its *lonRo* = 2, while the length from *ST* up to it is 1. So, we discovered a shorter path from another root. However, since we are interested in longest paths, we do not update *lonRo* with 1. But if we discovered a longer path from another root, we would have updated *lonRo*. In either cases, *lonLe* is left with no modifications. That is how our graph ends up



Finally, we return max(lonLe) among all roots. In this case, the answer is 5.

As requested by grant, I am going to write a psuedo-code for the algorithm illustrated above.

input: directed acyclic graph G, list of G's roots, curLe=0 output: longest path from any route to any leaf, a.k.a given graph's diameter

```
findDiameter(graph G, list children, curLe):
    for vertix in children:
        if G.vertex.lonRo > curLe:
            return G.vertix.lonLe + 1
        G.vertix.lonRo=curLe

        if G.lonLe !=1:      #already computed before
            return G.vertix.lonLe + 1

        leafCou = findDiameter(G, vertix.children, curLe+1, &leafCou "passed by reference")
```

```
        G.vertix.lonLe = max(leafCou, G.vertix.lonLe)
        return G.vertix.lonLe + 1

    return 0
```

After completing traversing the graph, we output max found leaf

```
max = -1
for vertix in G's roots:
    if G.lonRo > max:
        max = G.lonRo

output: max
```