I implemented my algorithm for search based on Breadth First Search using a recursive function named "h(i, j, dist)". First paramaters "i", "j" and "dist" are passed, "i" and "j" represents ith and jth elements on network. "dist" represents the hopping distance which will be calculated by this function. It is passed as a parameter and changed in each call. First of all function has 2 base cases to stop it from recursing further:

- If function already reaches jth node which means i and j are same, stop and return calculated distance.
- If function has exceeded maximum number of nodes to hop, then function is simply looping over same path which results in function stack to overflow. So function should notify chosen path is never ending by returning a unique value in this case maximum integer as this means the path is so long it can never be the minimum path. Choose another closer path on the previous function call in stack.

So if the base cases are not true then function still haven't reached element it is searching for or circling in a endless recursive function call loop. Then we should get all possible nearby elements function can hop into (recurse).
"sorted" array here will hold indexes of these nearby possible elements, this array is built in order:
- Initialize with total number of connections (nearby elements) current element "i" has. This is found from adjacency matrix of whole network which is created before with "buildNetwork()" function. The adjacency matrix has total number of connections on each nodes on diagonal of matrix and on other entries of matrix whether it is connected or not. For ex: non-diagonal (i, j) entry is 0 if there is no direct connection (link) between ith and jth node, if it is 1 then there is connection between them. Get number of connections from corresponding diagonal entry. This will give the size needed for "sorted" array.
- Then check for nearby element connections just by iterating through corresponding row of adjacency matrix. Don't check for diagonal entries. Skip non-connected elements (elements with entries 0). If the target laptop "j" is reachable then hop into it by returning "dist + 1". If all this cases fail then it is a possible path to hop into so add it to "sorted" array. Then bubblesort this sorted array based on distance ordered minumum to maximum.

Finally assume minimum path after current element to be not reachable (maximum integer), function must try every and each path in "sorted" array. Iterating through it and calling recursively, don't forget to pass in already calculated distance "dist", calculate distance from each called path. By comparing each one of the calculated paths with "min" we get the minimum distance from current element "i". Then returning this minimum value we compare each one of the possible paths on the call stack and after last "h()" function returns it will be the minimum path from "i" to "j".

In conclusion:
- Filling "sorted" array takes O(n) time,
- Sorting "sorted" using bubblesort takes O(n^2) time,
- Last loop where "h()" is called recursively takes O(n) time normally but there is a recursive call for "h()" so if "h()" takes O(x) time then this loop takes O(n^x) and the "h()" itself takes O(n^2) because of procedures mentioned above. So in total "h()" should take O(n^n) on worst case cause the function can be called on n times on each call of itself and the maximum recurrence is n because of second base case stopping function from looping in circles repeating the same path.
Total Big O estimate:
Around O(n^n)

Erhan Yalnız - 150117905