Centroid Decomposition of Tree - GeeksforGeeks

Source: https://www.geeksforgeeks.org/centroid-decomposition-of-tree/

Courses Tutorials Practice Jobs DSA Tutorial Interview Questions Quizzes Must Do Advanced DSA System Design Aptitude Puzzles Interview Corner DSA Python Technical Scripter 2026 Explore DSA Fundamentals Logic Building Problems Analysis of Algorithms Data Structures Array Data Structure String in Data Structure Hashing in Data Structure Linked List Data Structure Stack Data Structure Queue Data Structure Tree Data Structure Graph Data Structure Trie Data Structure Algorithms Searching Algorithms Sorting Algorithms Introduction to Recursion Greedy Algorithms Tutorial Graph Algorithms Dynamic Programming or DP Bitwise Algorithms Advanced Segment Tree Binary Indexed Tree or Fenwick Tree Square Root (Sqrt) Decomposition Algorithm Binary Lifting Geometry Interview Preparation Interview Corner GfG160 Practice Problem GeeksforGeeks Practice - Leading Online Coding Platform Problem of The Day - Develop the Habit of Coding DSA Course 90% Refund Centroid Decomposition of Tree Last Updated : 23 Jul, 2025 Background : What is centroid of Tree? Centroid of a Tree is a node which if removed from the tree would split it into a ‘forest’, such that any tree in the forest would have at most half the number of vertices in the original tree. Suppose there are n nodes in the tree. ‘Subtree size’ for a node is the size of the tree rooted at the node. Let S(v) be size of subtree rooted at node v S(v) = 1 + ? S(u) Here u is a child to v (adjacent and at a depth one greater than the depth of v). Centroid is a node v such that, maximum(n - S(v), S(u 1), S(u 2), .. S(u m)) <= n/2 where u i is i'th child to v. Finding the centroid Let T be an undirected tree with n nodes. Choose any arbitrary node v in the tree. If v satisfies the mathematical definition for the centroid, we have our centroid. Else, we know that our mathematical inequality did not hold, and from this, we conclude that there exists some u adjacent to v such that S(u) > n/2. We make that u our new v and recurse. We never revisit a node because when we decided to move away from it to a node with subtree size greater than n/2, we sort of declared that it now belongs to the component with nodes less than n/2, and we shall never find our centroid there. In any case we are moving towards the centroid. Also, there are finitely many vertices in the tree. The process must stop, and it will, at the desired vertex. Algorithm : Select arbitrary node v Start a DFS from v, and setup subtree sizes Re-position to node v (or start at any arbitrary v that belongs to the tree) Check mathematical condition of centroid for v If condition passed, return current node as centroid Else move to adjacent node with ‘greatest’ subtree size, and back to step 4 Theorem: Given a tree with n nodes, the centroid always exists. Proof: Clear from our approach to the problem that we can always find a centroid using above steps. Time Complexity Select arbitrary node v: O(1) DFS: O(n) Reposition to v: O(1) Find centroid: O(n) Centroid Decomposition : Finding the centroid for a tree is a part of what we are trying to achieve here. We need to think how can we organize the tree into a structure that decreases the complexity for answering certain ‘type’ of queries. Algorithm Make the centroid as the root of a new tree (which we will call as the ‘centroid tree’) Recursively decompose the trees in the resulting forest Make the centroids of these trees as children of the centroid which last split them. The centroid tree has depth O(log n), and can be constructed in O(n lg n), as we can find the centroid in O(n). Illustrative Example Let us consider a tree with 16 nodes. The figure has subtree sizes already set up using a DFS from node 1. We start at node 1 and see if condition for centroid holds. Remember S(v) is subtree size for v. We make node 6 as the root of our centroid, and recurse on the 3 trees of the forest centroid split the original tree into. NOTE : In the figure, subtrees generated by a centroid have been surrounded by a dotted line of the same color as the color of centroid. We make the subsequently found centroids as the children to centroid that split them last, and obtain our centroid tree. NOTE : The trees containing only a single element have the same element as their centroid. We haven’t used color differentiation for such trees, and the leaf nodes represent them. C++ // C++ program for centroid decomposition of Tree #include <bits/stdc++.h> using namespace std ; #define MAXN 1025 vector < int > tree [MAXN]; vector < int > centroidTree [MAXN]; bool centroidMarked [MAXN]; /* method to add edge between to nodes of the undirected tree */ void addEdge (int u , int v) { tree [u]. push_back (v); tree [v]. push_back (u); } /* method to setup subtree sizes and nodes in current tree */ void DFS (int src , bool visited [], int subtree_size [], int * n) { /* mark node visited */ visited [src] = true ; /* increase count of nodes visited */ * n += 1 ; /* initialize subtree size for current node*/ subtree_size [src] = 1 ; vector < int >:: iterator it ; /* recur on non-visited and non-centroid neighbours */ for (it = tree [src]. begin (); it != tree [src]. end (); it ++) if (! visited [* it] && !

centroidMarked [* it]) { DFS (* it , visited , subtree_size , n); subtree_size [src] += subtree_size [* it]; } } int getCentroid (int src , bool visited [], int subtree_size [], int n) { /* assume the current node to be centroid */ bool is_centroid = true ; /* mark it as visited */ visited [src] = true ; /* track heaviest child of node, to use in case node is not centroid */ int heaviest_child = 0 ; vector < int >:: iterator it ; /* iterate over all adjacent nodes which are children (not visited) and not marked as centroid to some subtree */ for (it = tree [src]. begin (); it != tree [src]. end (); it ++) if (! visited [* it] && ! centroidMarked [* it]) { /* If any adjacent node has more than n/2 nodes, * current node cannot be centroid */ if (subtree_size [* it] > n / 2) is_centroid = false ; /* update heaviest child */ if (heaviest_child == 0 || subtree_size [* it] > subtree_size [heaviest_child]) heaviest_child = * it ; } /* if current node is a centroid */ if (is_centroid && n - subtree_size [src] <= n / 2) return src ; /* else recur on heaviest child */ return getCentroid (heaviest_child , visited , subtree_size , n); } /* function to get the centroid of tree rooted at src. * tree may be the original one or may belong to the forest */ int getCentroid (int src) { bool visited [MAXN]; int subtree_size [MAXN]; /* initialize auxiliary arrays */ memset (visited , false , sizeof visited); memset (subtree_size , 0 , sizeof subtree_size); /* variable to hold number of nodes in the current tree */ int n = 0 ; /* DFS to set up subtree sizes and nodes in current tree */ DFS (src , visited , subtree_size , & n); for (int i = 1 ; i < MAXN ; i ++) visited [i] = false ; int centroid = getCentroid (src , visited , subtree_size , n); centroidMarked [centroid] = true ; return centroid ; } /* function to generate centroid tree of tree rooted at src */ int decomposeTree (int root) { //printf("decomposeTree(%d)\n", root); /* get centroid for current tree */ int cend_tree = getCentroid (root); printf ("%d " , cend_tree); vector < int >:: iterator it ; /* for every node adjacent to the found centroid * and not already marked as centroid */ for (it = tree [cend_tree]. begin (); it != tree [cend_tree]. end (); it ++) { if (! centroidMarked [* it]) { /* decompose subtree rooted at adjacent node */ int cend_subtree = decomposeTree (* it); /* add edge between tree centroid and centroid of subtree */ centroidTree [cend_tree]. push_back (cend_subtree); centroidTree [cend_subtree]. push_back (cend_tree); } } /* return centroid of tree */ return cend_tree ; } // driver function int main () { /* number of nodes in the tree */ int n = 16 ; /* arguments in order: node u, node v * sequencing starts from 1 */ addEdge (1 , 4); addEdge (2 , 4); addEdge (3 , 4); addEdge (4 , 5); addEdge (5 , 6); addEdge (6 , 7); addEdge (7 , 8); addEdge (7 , 9); addEdge (6 , 10); addEdge (10 , 11); addEdge (11 , 12); addEdge (11 , 13); addEdge (12 , 14); addEdge (13 , 15); addEdge (13 , 16); /* generates centroid tree */ decomposeTree (1); return 0 ; } Java import java.util.ArrayList ; import java.util.Arrays ; import java.util.List ; public class CentroidDecomposition { static final int MAXN = 1025 ; static List < Integer >[] tree = new ArrayList [MAXN] ; static List < Integer >[] centroidTree = new ArrayList [MAXN] ; static boolean [] centroidMarked = new boolean [MAXN] ; // Method to add an edge between two nodes in the undirected tree static void addEdge (int u , int v) { tree [u] . add (v); tree [v] . add (u); } // Method to set up subtree sizes and nodes in the current tree static void DFS (int src , boolean [] visited , int [] subtreeSize , int [] n) { visited [src] = true ; n [0]++ ; subtreeSize [src] = 1 ; for (int neighbor : tree [src]) { if (! visited [neighbor] && ! centroidMarked [neighbor]) { DFS (neighbor , visited , subtreeSize , n); subtreeSize [src] += subtreeSize [neighbor] ; } } } // Get the centroid of the tree rooted at 'src' static int getCentroid (int src) { boolean [] visited = new boolean [MAXN] ; int [] subtreeSize = new int [MAXN] ; Arrays . fill (visited , false); Arrays . fill (subtreeSize , 0); int [] n = { 0 }; // Number of nodes in the current tree DFS (src , visited , subtreeSize , n); Arrays . fill (visited , false); int centroid = getCentroid (src , visited , subtreeSize , n [0]); centroidMarked [centroid] = true ; return centroid ; } // Get the centroid of the subtree rooted at 'src' static int getCentroid (int src , boolean [] visited , int [] subtreeSize , int n) { boolean isCentroid = true ; visited [src] = true ; int heaviestChild = 0 ; for (int neighbor : tree [src]) { if (! visited [neighbor] && ! centroidMarked [neighbor]) { if (subtreeSize [neighbor] > n / 2) isCentroid = false ; if (heaviestChild == 0 || subtreeSize [neighbor] > subtreeSize [heaviestChild]) heaviestChild = neighbor ; } } if (isCentroid && n - subtreeSize [src] <= n / 2) return src ; return getCentroid (heaviestChild , visited , subtreeSize , n); } // Generate the centroid tree of the tree rooted at 'root' static int decomposeTree (int root) { int cendTree = getCentroid (root); System . out . print (cendTree + " "); for (int neighbor : tree [cendTree]) { if (! centroidMarked [neighbor]) { int cendSubtree = decomposeTree (neighbor); centroidTree [cendTree] . add (cendSubtree); centroidTree [cendSubtree] . add (cendTree); } } return cendTree ; } // Driver function public static void main (String [] args) { // Number of nodes in the tree int n = 16 ; for (int i = 0 ; i < MAXN ; i ++) { tree [i] = new ArrayList <> (); centroidTree [i] = new ArrayList <> (); } // Add edges to the tree addEdge (1 , 4); addEdge (2 , 4); addEdge (3 , 4); addEdge (4 , 5); addEdge (5 , 6); addEdge (6 , 7); addEdge (7 , 8); addEdge (7 , 9); addEdge (6 , 10); addEdge (10 , 11); addEdge (11 , 12); addEdge (11 , 13); addEdge (12 , 14); addEdge (13 , 15); addEdge (13 , 16); // Generate the centroid tree decomposeTree (1); } } Python3 import

collections MAXN = 1025 tree = collections . defaultdict (list) centroidTree = collections . defaultdict (list) centroidMarked = [False] * MAXN # method to add edge between to nodes of the undirected tree def addEdge (u , v): tree [u] . append (v) tree [v] . append (u) # method to setup subtree sizes and nodes in current tree def DFS (src , visited , subtree_size , n): # mark node visited visited [src] = True # increase count of nodes visited n [0] += 1 # initialize subtree size for current node subtree_size [src] = 1 # recur on non-visited and non-centroid neighbours for it in tree [src]: if not visited [it] and not centroidMarked [it]: DFS (it , visited , subtree_size , n) subtree_size [src] += subtree_size [it] def getCentroid (src , visited , subtree_size , n): # assume the current node to be centroid is_centroid = True # mark it as visited visited [src] = True # track heaviest child of node, to use in case node is # not centroid heaviest_child = 0 # iterate over all adjacent nodes which are children # (not visited) and not marked as centroid to some # subtree for it in tree [src]: if not visited [it] and not centroidMarked [it]: # If any adjacent node has more than n/2 nodes, # current node cannot be centroid if subtree_size [it] > n / 2 : is_centroid = False # update heaviest child if heaviest_child == 0 or subtree_size [it] > subtree_size [heaviest_child]: heaviest_child = it # if current node is a centroid if is_centroid and n - subtree_size [src] <= n / 2 : return src # else recur on heaviest child return getCentroid (heaviest_child , visited , subtree_size , n) # function to get the centroid of tree rooted at src. # tree may be the original one or may belong to the forest # function to get the centroid of tree rooted at src. # tree may be the original one or may belong to the forest def getCentroidTree (src): visited = [False] * MAXN subtree_size = [0] * MAXN # initialize auxiliary arrays n = [0] # DFS to set up subtree sizes and nodes in current tree DFS (src , visited , subtree_size , n) visited = [False] * MAXN centroid = getCentroid (src , visited , subtree_size , n [0]) centroidMarked [centroid] = True return centroid # function to generate centroid tree of tree rooted at src def decomposeTree (root): # get centroid for current tree cend_tree = getCentroidTree (root) print (cend_tree , end = " ") # for every node adjacent to the found centroid, # decompose the tree rooted at that node for it in tree [cend_tree]: if not centroidMarked [it]: decomposeTree (it) # driver code if __name__ == "__main__" : # number of nodes in the tree n = 16 # arguments in order: node u, node v # sequencing starts from 1 addEdge (1 , 4) addEdge (2 , 4) addEdge (3 , 4) addEdge (4 , 5) addEdge (5 , 6) addEdge (6 , 7) addEdge (7 , 8) addEdge (7 , 9) addEdge (6 , 10) addEdge (10 , 11) addEdge (11 , 12) addEdge (11 , 13) addEdge (12 , 14) addEdge (13 , 15) addEdge (13 , 16) # generates centroid tree decomposeTree (1) C# using System ; using System.Collections.Generic ; namespace CentroidDecomposition { class Program { static List < int > [] tree = new List < int > [1025]; static List < int > [] centroidTree = new List < int > [1025]; static bool [] centroidMarked = new bool [1025]; /* method to add edge between to nodes of the undirected * tree */ static void AddEdge (int u , int v) { tree [u]. Add (v); tree [v]. Add (u); } /* method to setup subtree sizes and nodes in current * tree */ static void DFS (int src , bool [] visited , int [] subtree_size , ref int n) { /* mark node visited */ visited [src] = true ; /* increase count of nodes visited */ n ++ ; /* initialize subtree size for current node*/ subtree_size [src] = 1 ; /* recur on non-visited and non-centroid neighbours */ foreach (int neighbour in tree [src]) { if (! visited [neighbour] && ! centroidMarked [neighbour]) { DFS (neighbour , visited , subtree_size , ref n); subtree_size [src] += subtree_size [neighbour]; } } } static int GetCentroid (int src , bool [] visited , int [] subtree_size , int n) { /* assume the current node to be centroid */ bool is_centroid = true ; /* mark it as visited */ visited [src] = true ; /* track heaviest child of node, to use in case node * is not centroid */ int heaviest_child = 0 ; /* iterate over all adjacent nodes which are * children (not visited) and not marked as centroid * to some subtree */ foreach (int neighbour in tree [src]) { if (! visited [neighbour] && ! centroidMarked [neighbour]) { /* If any adjacent node has more than n/2 * nodes, current node cannot be centroid */ if (subtree_size [neighbour] > n / 2) { is_centroid = false ; } /* update heaviest child */ if (heaviest_child == 0 || subtree_size [neighbour] > subtree_size [heaviest_child]) { heaviest_child = neighbour ; } } } /* if current node is a centroid */ if (is_centroid && n - subtree_size [src] <= n / 2) { return src ; } /* else recur on heaviest child */ return GetCentroid (heaviest_child , visited , subtree_size , n); } /* function to get the centroid of tree rooted at src. * tree may be the original one or may belong to the * forest */ static int GetCentroid (int src) { bool [] visited = new bool [1025]; int [] subtree_size = new int [1025]; /* initialize auxiliary arrays */ Array . Fill (visited , false); Array . Fill (subtree_size , 0); /* variable to hold number of nodes in the current * tree */ int n = 0 ; /* DFS to set up subtree sizes and nodes in current * tree */ DFS (src , visited , subtree_size , ref n); Array . Fill (visited , false); int centroid = GetCentroid (src , visited , subtree_size , n); centroidMarked [centroid] = true ; return centroid ; } /* function to generate centroid tree of tree rooted at * src */ static int DecomposeTree (int root) { /* get centroid for current /* get centroid for current tree */ int cend_tree = GetCentroid (root); Console . Write (cend_tree + " "); /* for every node adjacent to the found centroid * and not already

marked as centroid */ foreach (int adjNode in tree [cend_tree]) { if (! centroidMarked [adjNode]) { /* decompose subtree rooted at adjacent node */ int cend_subtree = DecomposeTree (adjNode); /* add edge between tree centroid and * centroid of subtree */ centroidTree [cend_tree]. Add (cend_subtree); centroidTree [cend_subtree]. Add (cend_tree); } } /* return centroid of tree */ return cend_tree ; } // driver function public static void Main (string [] args) { /* number of nodes in the tree */ int n = 16 ; for (int i = 0 ; i < 1025 ; i ++) { tree [i] = new List < int > (); centroidTree [i] = new List < int > (); } /* arguments in order: node u, node v * sequencing starts from 1 */ AddEdge (1 , 4); AddEdge (2 , 4); AddEdge (3 , 4); AddEdge (4 , 5); AddEdge (5 , 6); AddEdge (6 , 7); AddEdge (7 , 8); AddEdge (7 , 9); AddEdge (6 , 10); AddEdge (10 , 11); AddEdge (11 , 12); AddEdge (11 , 13); AddEdge (12 , 14); AddEdge (13 , 15); AddEdge (13 , 16); /* generates centroid tree */ DecomposeTree (1); } } } JavaScript const MAXN = 1025 ; const tree = {}; const centroidTree = {}; const centroidMarked = new Array (MAXN). fill (false); // method to add edge between two nodes of the undirected tree function addEdge (u , v) { if (! tree [u]) tree [u] = []; if (! tree [v]) tree [v] = []; tree [u]. push (v); tree [v]. push (u); } // method to setup subtree sizes and nodes in current tree function DFS (src , visited , subtree_size , n) { // mark node visited visited [src] = true ; // increase count of nodes visited n [0] += 1 ; // initialize subtree size for current node subtree_size [src] = 1 ; // recur on non-visited and non-centroid neighbours for (let i = 0 ; i < tree [src]. length ; i ++) { const it = tree [src][i]; if (! visited [it] && ! centroidMarked [it]) { DFS (it , visited , subtree_size , n); subtree_size [src] += subtree_size [it]; } } } function getCentroid (src , visited , subtree_size , n) { // assume the current node to be centroid let is_centroid = true ; // mark it as visited visited [src] = true ; // track heaviest child of node, to use in case node is // not centroid let heaviest_child = 0 ; // iterate over all adjacent nodes which are children // (not visited) and not marked as centroid to some // subtree for (let i = 0 ; i < tree [src]. length ; i ++) { const it = tree [src][i]; if (! visited [it] && ! centroidMarked [it]) { // If any adjacent node has more than n/2 nodes, // current node cannot be centroid if (subtree_size [it] > n / 2) { is_centroid = false ; } // update heaviest child if (heaviest_child == 0 || subtree_size [it] > subtree_size [heaviest_child]) { heaviest_child = it ; } } } // if current node is a centroid if (is_centroid && n - subtree_size [src] <= n / 2) { return src ; } // else recur on heaviest child return getCentroid (heaviest_child , visited , subtree_size , n); } // function to get the centroid of tree rooted at src. // tree may be the original one or may belong to the forest function getCentroidTree (src) { const visited = new Array (MAXN). fill (false); const subtree_size = new Array (MAXN). fill (0); // initialize auxiliary arrays const n = [0]; // DFS to set up subtree sizes and nodes in current tree DFS (src , visited , subtree_size , n); visited . fill (false); const centroid = getCentroid (src , visited , subtree_size , n [0]); centroidMarked [centroid] = true ; return centroid ; } // function to generate centroid tree of tree rooted at src function decomposeTree (root) { // get centroid for current tree const cend_tree = getCentroidTree (root); process . stdout . write (cend_tree + " "); // mark centroid as visited in original tree centroidMarked [cend_tree] = true ; // for every node adjacent to the found centroid, // decompose the tree rooted at that node for (let i = 0 ; i < tree [cend_tree]. length ; i ++) { const it = tree [cend_tree][i]; if (! centroidMarked [it]) { decomposeTree (it); } } } // driver code // number of nodes in the tree const n = 16 ; // arguments in order: node u, node v // sequencing starts from 1 addEdge (1 , 4); addEdge (2 , 4); addEdge (3 , 4); addEdge (4 , 5); addEdge (5 , 6); addEdge (6 , 7); addEdge (7 , 8); addEdge (7 , 9); addEdge (6 , 10); addEdge (10 , 11); addEdge (11 , 12); addEdge (11 , 13); addEdge (12 , 14); addEdge (13 , 15); addEdge (13 , 16); // generates centroid tree decomposeTree (1); Output : 6 4 1 2 3 5 7 8 9 11 10 12 14 13 15 16 Application: Consider below example problem Given a weighted tree with N nodes, find the minimum number of edges in a path of length K, or return -1 if such a path does not exist. 1 <= N <= 200000 1 <= length(i;j) <= 1000000 (integer weights) 1 <= K <= 1000000 Brute force solution: For every node, perform DFS to find distance and number of edges to every other node Time complexity: O(N 2) Obviously inefficient because N = 200000 We can solve above problem in O(N Log N) time using Centroid Decomposition . Perform centroid decomposition to get a "tree of subtrees" Start at the root of the decomposition, solve the problem for each subtree as follows Solve the problem for each "child tree" of the current subtree. Perform DFS from the centroid on the current subtree to compute the minimum edge count for paths that include the centroid Two cases: centroid at the end or in the middle of the path Time complexity of centroid decomposition based solution is O(n log n) Reference : https://www.students.cs.ubc.ca/~cs-490/2014W2/pdf/jason.pdf This article is contributed by Yash Varyani . Comment Article Tags: Article Tags: Advanced Data Structure DSA