Play with Trees Solutions

Amber
[ADN.cn]
hupo001@gmail.com

Contents

Problem A: Play with a Tree

Brief

Program Name: `PT07A`
Method Summary: Game Theory - Green Hackenbush
Time Complexity: $O(n)$

Editorial

Seven contestants solved this problem. **lozper** (Yixght) is the first one.

Source

IPSC 2003 - Problem G - Got Root?

Solution

This problem is a classical "Impartial Combinatorial Games" - Green Hackenbush [1]. Before reading this solution, you should have some knowledge of the Sprague-Grundy and the Nim game.

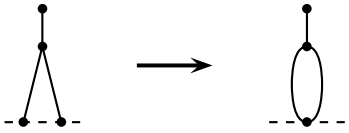


Figure 1: Merge all vertices on the ground

In this problem, there may be multiple vertices on the ground. Actually, we can consider them as just one vertex on the ground, i.e. merge them into one vertex. For example, Figure 1 shows how to merge all vertices on the ground.

1. Bamboo Stalks

Bamboo stalks consist of some segments with one of the ends of segments rooted to the ground. A single bamboo stalk of n segments can be moved into a bamboo stalk of any smaller number of segments from $n-1$ to 0. This movement is similar to the operation of the Nim game, i.e. take away any number > 0 of stones in one pile. So playing Green Hackenbush on Bamboo Stalks is completely equivalent to play the Nim game. Thus, its Sprague-Grundy value is the Nim sum of the Bamboo Stalk's lengths.

[1] Thomas S Ferguson. *Game Theory*. Mathematics Department, UCLA

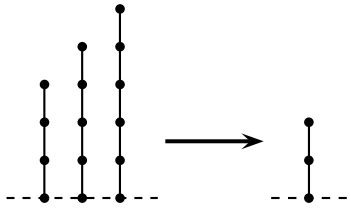


Figure 2: Green Hackenbush on Bamboo Stalks

For example, in Figure 2, there are three stalks on the ground. It is equivalent to the Nim game with three piles of 3, 4 and 5 stones. Its Sprague-Grundy value is $3 \oplus 4 \oplus 5 = 2$.

2. Trees

The task now is to find the Sprague-Grundy value of the tree. We focus on the vertex v in a tree whose children are all sticks. This case is equivalent to consider the vertex v as the ground and play the game of bamboo stalks in its subtree. Thus, we have the following principle.

Lemma A.1 (Colon Principle) *When branches come together at a vertex, one may replace the branches by a non-branching stalk of length equal to their nim sum.*

The colon principle allows us to reduce an arbitrary tree to an equivalent nim pile. Its correctness is obvious based on the result of bamboo stalks and the theory of the game sum.

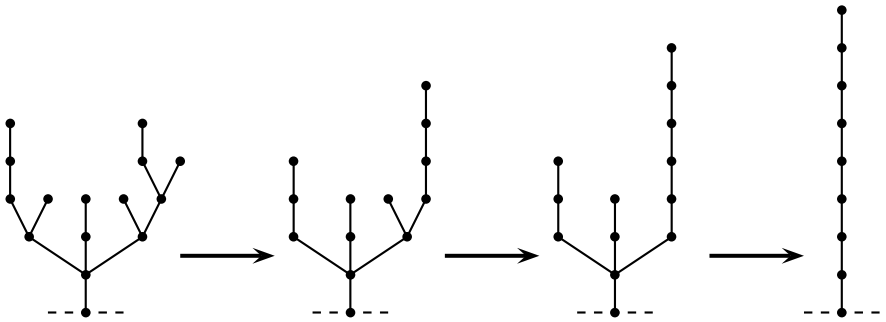


Figure 3: Green Hackenbush on Trees

For example, Figure 3 shows the equivalent transformation of a tree. In the first step, we replace the leftmost branch with one stalk of the length $3 \oplus 1 = 2$. and replace the rightmost branch with one stalk of the length $2 \oplus 1 = 3$. In the second step, we replace the rightmost branch with one stalk of the length $1 \oplus 4 = 5$. In the third step, we merge all three stalks with one stalk of length $3 \oplus 2 \oplus 6 = 7$.

3. General graph

In this section, we should consider how to process the circuits in the graph. Above all, the definition of "fuse" will be given.

Definition A.2 *We **fuse** two neighboring vertices by bringing them together into a single vertex and bending the edge joining them into a loop.*

Lemma A.3 (The Fusion Principle) *The vertices on any circuit may be fused without changing the Sprague-Grundy value of the graph.*

The fusion principle allows us to reduce an arbitrary graph into an equivalent tree which can be further reduced to a nim pile by the colon principle. For example, Figure 4 shows how to fuse a circuit into a vertex with some loops.

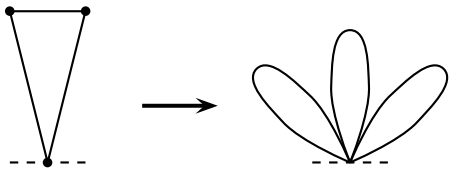


Figure 4: Green Hackenbush on Circuits

For implementing the fusion principle, we need to use strongly connected components algorithm to shrink all circuits in the graph into vertices. **Tarjan's strongly connected components algorithm** [2] and **Kosaraju's algorithm** [3] introduced by CLRS are both okay. After executing strongly connected components algorithm, the resultant graph is a tree with some loops on some vertices. The algorithm of Green Hackenbush is given as follows.

Algorithm 1 Green Hackenbush

Require: Given a tree with some loops on some vertices.
Ensure: Return the Sprague-Grundy value of the subtree at root u.

```
1: function GreenHackenbush(u)
2:     ret ← (the number of loops on the vertex u) mod 2.
3:     for all v ∈ Child(u) do
4:         ret ← ret ⊕ (1 + GreenHackenbush(v))
5:     end for
6:     return ret
7: end function
```

[2] http://en.wikipedia.org/wiki/Tarjan's_strongly_connected_components_algorithm
[3] http://en.wikipedia.org/wiki/Strongly_connected_component

Problem B: The Easiest Problem

Brief

Program Name:	`PT07B`
Method Summary:	Finding the maximal caterpillar in a tree
Time Complexity:	$O(n)$

Editorial

Thirty contestants solved this problem. However, since the strict time limit, some linear methods but with a large constant could not get accepted.

Solution

Definition B.1 *A **caterpillar** is a tree in which every node is on a central stalk (called **path node**) or only one edge away from the stalk (called **leaf**).*

Lemma B.2 (Caterpillar Recognition) [4] *A tree is a caterpillar iff all nodes of degree ≥ 3 are surrounded by at most two nodes of degree two or greater.*

According to the lemma, the task is equal to find the maximal caterpillar in a tree. We only consider the internal path nodes (the path nodes except the head or the tail of a path). The number of leaves which connect to a particular internal path node is the degree of that path node $-$ 2. Thus, we weight the nodes with their degrees $-$ 2, and remove the nodes in the tree whose weights are less than one. The task is converted into finding the maximal node weight path in the tree. The answer is the maximum node weight summation of that path + 2. The 2 denotes the head and the tail of the path.

Finding maximal node weight summation in a tree is equivalent to find the longest path in a tree, which can solved by the algorithm (traverse twice) described in Problem Z Solution. The correctness of the algorithm can be proved in Theorem Z.1.

[4] Weisstein, Eric W. "Caterpillar Tree." From MathWorld - A Wolfram Web Resource. http://mathworld.wolfram.com/CaterpillarTree.html

Problem C: The GbAaY Kingdom

Brief

Program Name:	`PT07C`
Method Summary:	Find the minimum diameter spanning tree
Time Complexity:	$O(n \cdot m)$

Editorial

Only three contestants solved this problem. **mountainking** (Lou Tiancheng) is the first one.

The standard algorithm is $O(n \cdot m)$. The $O(n \cdot m \cdot \log n)$ algorithm is hard to get accepted.

Solution

This problem is a classical problem, called the Minimum Diameter Spanning Tree. The $O(n \cdot m)$ algorithm [5] will be introduced in the following.

Notation C.1 (Basic Graph Glossary) Let $G = (V, E)$ be a connected, undirected, positively real-weighted graph, where the weight of an edge $e = (u, v) \in E$ is given by $\omega_{u,v} \in \mathbb{R}_+$.

- For all vertices u and v, the *distance* from u to v, denoted $d_G(u, v)$, is the shortest path from u to v in G. We call all $d_G(u, v)$s *all-pairs shortest paths* (APSP) of G. Sometimes, $d_G(u, v)$ abbreviates $d(u, v)$ for G.
- The maximal distance from vertex v to all other vertices in V, denoted $\varepsilon(v)$, is the *eccentricity* of v, i.e. $\varepsilon(v) = \max_{u \in V} d(v, u)$.
- $D(G)$ denotes the *diameter* of G, defined as $D(G) = \max_{v \in V} \varepsilon(v)$.
- The *minimum diameter spanning tree* (MDST) of G is a spanning tree T of G minimizing $D(T)$.
- $R(G)$ denotes the *radius* of G, defined as $R(G) = \min_{v \in V} \varepsilon(v)$.
- $\Psi(u)$ represents a *shortest-paths tree* (SPT) rooted at node u, such that $\forall v \in V, d(u, v) = d_{\Psi(u)}(u, v)$. ■

[5] Marc Bui, Franck Butelley, Christian Lavaulty, *A Distributed Algorithm for the Minimum Diameter Spanning Tree Problem*

Suppose that the median of the longest path of the MDST of G is on a certain vertex $v \in V$. Then the SPT $\Psi(v)$ will be a MDST of G. Thus we can enumerate all $v \in V$ and find the SPT $\Psi(v)$ having the minimum diameter $D(\Psi(v))$.

However, in fact, the median of the longest path of the MDST of G can appear in inner of a certain edge. Thus, we suggest the "dummy vertex" (so-called in contrast to actual vertices of V), that is a fictitious vertex may possibly be inserted on any edge $e \in E$.

Definition C.2 *Let $e = (u, v)$ be an edge of weight $\omega_{u,v}$.*

- *A **dummy vertex** γ inserted on e is defined by specifying the weight α of the "dummy edge" (u, γ).*
- *A **general vertex** is an actual vertex in V or a dummy vertex.*

Due to the generalization of the vertex, we should generalize all the concepts in Notation C.1. The domains of the distance, the eccentricity and the SPT are generalized to the general vertices, i.e. $d(\gamma, \gamma'), \varepsilon(\gamma), \Psi(\gamma)$.

If we could enumerate all general vertices in G, the MDST of G would be the SPT $\Psi(\gamma)$ having the minimum diameter. However, because there are infinite general vertices in G, we can't enumerate all of them. Nevertheless, in other way, we will find the "absolute center" of G directly.

Definition C.3 *An **absolute center** γ^* of G is defined as a general vertex γ having the minimal eccentricity $\varepsilon(\gamma)$.*

The absolute center γ^* of G is also the median of a longest path of the MDST $\Psi(\gamma^*)$. Obviously, the MDST problem for a graph G is reducible to the absolute center problem. The following shows how to find the absolute center.

1. For each edge $e \in E$, find a general vertex $\gamma \in e$ having the minimum eccentricity $\varepsilon(\gamma)$, denoted by γ_e.
2. γ^* is a general vertex having the minimum eccentricity among all the above γ_es.

The remaining work is to compute γ_e of a particular edge e.

For edge $e = (u, v) \in E$, let $\alpha = d(u, \gamma)$. Because for each other vertex w, the distance $d(\gamma, w)$ is the length of either a path $\gamma, u, \ldots, w$, or $\gamma, v, \ldots, w$. Hence,

$$\varepsilon(\gamma) = \max_{w \in V} d(\gamma, w) = \max_{w \in V} \min\{\alpha + d(u, w), \omega_{u,v} - \alpha + d(v, w)\}$$

If we plot $d(\gamma, w)$ in Cartesian coordinates for fixed w, the curve of $d(\gamma, w)$ are represented by two line segments with the slope $+1$ and -1. The curve of $d(\gamma, w)$ can be determined by only two numbers $d(u, w)$, $d(v, w)$, denoted by $a_w = d(u, w)$, $b_w = d(v, w)$ respectively. If we plot $d(\gamma, w)$ for all w, the curve of $\varepsilon(\gamma)$ are represented by a broken line that is the upper boundary of the convex cone of all $d(\gamma, w)$. The vertex γ achieving the global minimum value of the broken line represents the absolute center γ_e of the edge e. See Figure 5 illustratively.

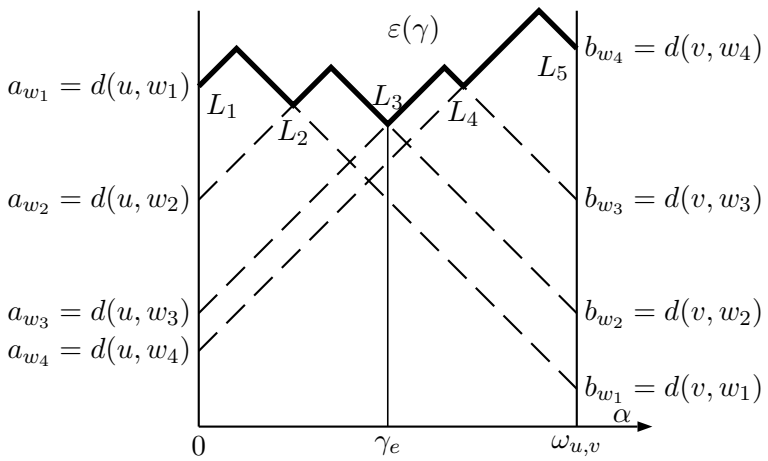


Figure 5: The curve of $\varepsilon(\gamma)$.

If there exists $d(u, w) \leq d(u, w')$ and $d(v, w) \leq d(v, w')$ for two certain vertices w, w', the curve of $d(\gamma, w)$ will be completely under the curve of $d(\gamma, w')$, and then we can remove w and w's curve. We call removing all above-mentioned useless vertices w by "simplification". If we plot $d(u, w) = a_w$, $d(v, w) = b_w$ as a coordinate (a_w, b_w) for all $w \in V$ (See Figure 6), we will find that only the upper boundary of the convex cone of the points can be remained after "simplification".

After "simplification", suppose w has already been ordered by a_w, the local minimum value L_w of $\varepsilon(\gamma)$ will be intersected by the curves of $d(\gamma, w)$ of two adjacent ws (e.g. In Figure 5, L_2 is intersected by the curves of $d(\gamma, w_1)$ and $d(\gamma, w_2)$). We can the global minimum value γ_e among all local minimum values L_w.

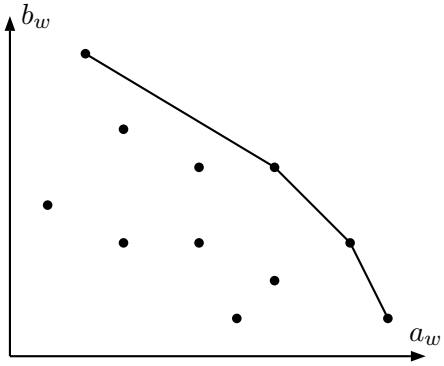


Figure 6: The simplification of w.

Algorithm 2 MDST

Require: Given a connected, undirected, positively real-weighted graph $G = (V, E)$
Ensure: Return the MDST of G

```
1: function MDST(G)
2:     run Floyd-Warshall(f)or finding all d(u, v) (APSP)
3:     for all e = (u, v) ∈ E do
4:         Phase 1: Simplification in O(n) time
5:         S = ∅                          ▷ S stores all w remained after "simplification"
6:         for all w ∈ V ordered by a_w = d(u, w) do
7:             update S by w (a_w, b_w)
8:         end for

9:         Phase 2: Compute γ_e in O(n) time
10:        for all w ∈ S ordered by a_w = d(u, w) do
11:            compute the local minimum value L_w intersected by the curves of w and w + 1
12:            update γ_e by L_w
13:        end for
14:        update γ* by γ_e
15:    end for
16:    return Ψ(γ*)
17: end function
```

The time complexity of the algorithm above is obviously $O(n \cdot m)$.

Problem D: Let us count 1 2 3

Brief

Program Name:	`PT07D`
Method Summary:	Sequence Problem
Time Complexity:	$O(n^2)$

Editorial

It's indeed an easy problem in an *Internet* contest because we can collect enough information online. So seven contestants solved this problem.

Solution

1. Number of labeled unrooted trees with n nodes

Theorem D.1 (Cayley Theorem) [6] *The number of spanning trees of the complete graph K_n is n^{n-2}.*

Due to "labeled" and "unrooted", we can find that the answer is just the number of spanning trees in the complete graph K_n.

Corollary D.2 (A000272) [7] *The number of labeled unrooted trees with n nodes is n^{n-2}.*

2. Number of labeled rooted trees with n nodes

Corollary D.3 (A000169) [8] *The number of labeled rooted trees with n nodes is n^{n-1}.*

Proof For each spanning tree of K_n, we can set any node as a root and get n rooted trees from each spanning tree. According to Corollary D.2, we can get $n^{n-2} \cdot n = n^{n-1}$ rooted tree. ■

3. Number of unlabeled rooted trees with n nodes

Let a_n be the number of unlabeled rooted trees with n nodes.

[6] Douglas B.West, *Introduction to Graph Theory*, Theorem 2.2.3

[7] N. J. A. Sloane, *The On-Line Encyclopedia of Integer Sequences*, Sequences A000272 "Number of labeled trees on n nodes"

[8] N. J. A. Sloane, *The On-Line Encyclopedia of Integer Sequences*, Sequences A000169 "Number of labeled rooted trees with n nodes"

Denote the generating function of a_n by $A(x)$, then

$$A(x) = \sum_{n=0}^{\infty} a_n x^n$$

Let c_i be the number of the subtree, which is a child of the tree root and whose size is equal to i. We have:

$$a_n = \sum_{\sum_{i=1}^{n-1} i \cdot c_i = n-1} \left[\prod_{k=1}^{n-1} \binom{a_k + c_k - 1}{c_k} \right]$$

Here, $\binom{a_k+c_k-1}{c_k}$ denotes the number of ways to put a_k kinds of subtrees into c_k boxes.

Through simplifying $A(x)$ based on a_n, we get

$$A(x) = x \exp\left(\sum_{r=1}^{\infty} \frac{1}{r} A(x^r) \right)$$

Then, we can get the following result.

Theorem D.4 (A000081) [9] *Denote the number of unlabeled rooted trees with n nodes by a_n, then*

$$a_{n+1} = \frac{1}{n} \cdot \sum_{i=1}^{n} \left(i \cdot a_i \cdot \sum_{j=1}^{\lfloor n/i \rfloor} a_{n+1-i \cdot j} \right)$$

4. Number of unlabeled unrooted trees with n nodes

Theorem D.5 (A000055) [10] *Denote the number of unlabeled unrooted trees with n nodes by b_n. If n is odd number, then*

$$b_n = a_n - \sum_{i=1}^{\lfloor n/2 \rfloor} a_i \cdot a_{n-i}$$

If n is even number, then

$$b_n = a_n - \sum_{i=1}^{\lfloor n/2 \rfloor} a_i \cdot a_{n-i} + \binom{a_{n/2} + 1}{2}$$

Remark D.6 *Denote the generating function of b_n by $B(x)$, i.e.*

$$B(x) = \sum_{n=0}^{\infty} b_n x^n$$

We have

$$B(x) = A(x) - \frac{1}{2} \cdot [A^2(x) - A(x^2)]$$

[9] N. J. A. Sloane, *The On-Line Encyclopedia of Integer Sequences*, Sequences A000081 "Number of rooted trees with n nodes"

[10] N. J. A. Sloane, *The On-Line Encyclopedia of Integer Sequences*, Sequences A000055 "Number of trees with n unlabeled nodes"

Implementation Skills

1. Speed up by the partial sum

We use the partial sum in order to speed up the calculation of the formula of a_n.

$$a_{n+1} = \frac{1}{n} \cdot \sum_{i=1}^{n} \left(i \cdot a_i \cdot \sum_{j=1}^{\lfloor n/i \rfloor} a_{n+1-i \cdot j} \right)$$

Let

$$s_{n,i} = \sum_{j=1}^{\lfloor n/i \rfloor} a_{n+1-i \cdot j}$$

In recursion form, we have

$$s_{n,i} = s_{n-i,i} + a_{n+1-i}$$

Then, we rewrite the formula of a_n.

$$a_{n+1} = \frac{1}{n} \cdot \sum_{i=1}^{n} i \cdot a_i \cdot s_{n,i}$$

Thus, we reduce the time complexity from $O(n^2 \cdot \log n)$ to $O(n^2)$.

2. Calculate the number modulo prime p

Lemma D.7 *If* $\gcd(a, p) = 1$*, there exists a* ***modular inverse*** a^{-1} *modulo* p *such that* $a \cdot a^{-1} \equiv 1 \pmod{p}$.

Proof Due to the Extended Euclid's algorithm (Algorithm 3), the equation below must have a solution (x, y).

$$a \cdot x + p \cdot y = \gcd(a, p) = 1$$

Let a^{-1} be x constructively. ■

Due to Lemma D.7, we have

$$x = \frac{a}{b} = a \cdot b^{-1} \pmod{p}$$

Algorithm 3 Extended Euclid's algorithm

```
1: function ExtendedGCD(a, b)          ▷ Return a pair (x, y) such that a · x + b · y = gcd(a, b)
2:     if b = 0 then
3:         return (1, 0)
4:     else
5:         (y, x) ← ExtendedGCD(b, a mod b)
6:         y ← y − ⌊a/b⌋ · x
7:         return (x, y)
8:     end if
9: end function
```

3. Exponentiation by repeated squaring

Algorithm 4 shows how to calculate $x^y \mod p$ in $O(\log y)$ time.

Algorithm 4 Exponentiation by repeated squaring

```
1: function ModularPower(x, y, p)                    ▷ Return x^y mod p
2:     if y = 0 then
3:         return 1
4:     else
5:         ret ← [ModularPower(x, ⌊y/2⌋, p)]^2
6:         if y is odd number then
7:             ret ← ret · x
8:         end if
9:         return ret
10:    end if
11: end function
```

Problem E: Yet another computer network problem

Brief

Program Name:	`PT07E`
Method Summary:	Approximation algorithm for MBDST
Time Complexity:	$O(?)$

Editorial

Only six contestants solved this problem, that's why the score is pretty high. After 9 submissions **andyshou** (andyshou) got the first rank, then **mountainking** (Lou Tiancheng) used all of his 10 submissions to get the same place.

Solution

This problem has a well-known name "Minimum bounded degree spanning tree".

Definition E.1 *The Minimum Bounded Degree Spanning Tree (MBDST) problem is given an undirected graph $G = (V, E)$, costs $c : E \to \mathbb{R}_+$ and an integer $B \geq 2$, find a spanning tree of maximum degree at most B and of minimum cost. For $B = 2$, this is the Hamiltonian path problem. The problem is NP-hard for any given B.*

In this thesis [11] , they showed that they can efficiently find a spanning tree of maximum degree at most $B + 2$ whose cost is at most the cost of the optimum spanning tree of maximum degree at most B.

In `PT07E`, all of our tests have $B \geq 3$. Here, we only introduce you some randomized algorithms that easily to implement in real contest.

There are 2 main parts of the idea:

1. Use adjusting tree method of **Kruskal** algorithm to build minimum spanning tree with bounded degree B. If we can't get a tree with maximum degree is at most B, we will slightly increase B to get a tree.
2. Use randomized algorithms to get another result.

Finally, we choose the best answer between them.

[11] Michel X. Goemans, *Minimum Bounded Degree Spanning Trees*, M.I.T

mountainking (Lou Tiancheng) at the first part, he chose a random upper limit for degree (e.g. $B + 20$); next, continued adjusting nodes to get a tree with this new limit and edges' list ordered by the maximum degree between 2 end-points of each edge. The same as the second part, but without using any new limit for degree.

andyshou (andyshou) at the first part he increased B step by step. And at the second part, he increased cost of each edge by one random number, ordered the edges' list by this new cost. Then he computed minimum spanning tree as the first part. After nearly 30 times running the second part he got quite good result.

Problem F: A short vacation in Disneyland

Brief

Program Name: `PT07F`
Method Summary: Find the minimal path cover of a tree
Time Complexity: $O(n)$

Editorial

27 contestants solved this problem.

Solution

There are two methods to solve it. One is dynamic programming, the other is greedy method. Both of them run in $O(n)$ time.

Method 1: Dynamic Programming

The minimum path number of the subtree of the root x denotes $f_1(x)$ if x is an endpoint of a certain covering path and denotes $f_0(x)$ if x is a internal vertex of a certain covering path. We can compute $f_0(x)$ and $f_1(x)$ as follows:

$$f_1(x) = \sum_{c \in \text{Child}(x)} f_0(c) + \min_{c \in \text{Child}(x)} [f_1(c) - f_0(c)] \quad \text{(F.1)}$$

$$f_0(x) = \sum_{c \in \text{Child}(x)} f_0(c) + \min_{c_1, c_2 \in \text{Child}(x)} [f_1(c_1) + f_1(c_2) - f_0(c_1) - f_0(c_2) - 1] \quad \text{(F.2)}$$

Equation F.1 represents to choose a child c as an internal node of a path connected to root x (See Figure 7). Equation F.2 represents to choose two children c_1 and c_2 and form a path from c_1 to x to c_2. (See Figure 8).

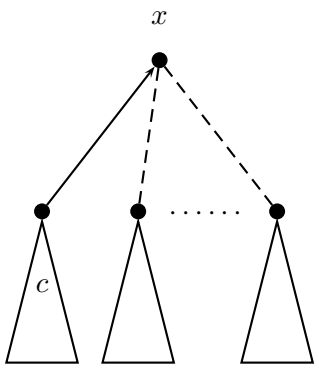


Figure 7: Explanation for Equation F.1

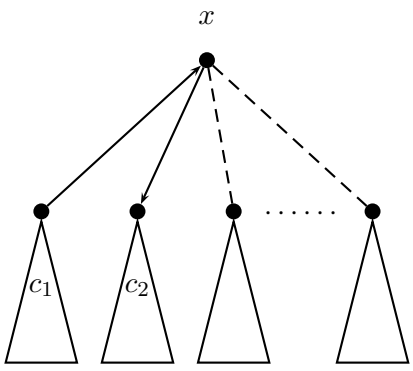


Figure 8: Explanation for Equation F.2

Method 2: Greedy Method

Definition F.1 *A **star** is a tree that has a center with its degree* ≥ 2 *and satisfies the degrees of the other vertices* ≤ 2*.*

The greedy algorithm is given below.

If there exists a star in the tree, which after removing from the tree, the tree won't be disjointed, , we can remove the star from the tree and construct a path cover of the star as follows:

- put an arbitrary path from leaf to leaf that passes the center of the star into the path cover.
- put the remaining paths in the star into the path cover.

Repeat this operation till the tree are empty.

For example, see Figure 9. In 1^{st} step, we find a star $\{2, 5, 6, 7\}$ in the grey frame. It can be decomposed to two paths $\{5, 2, 7\}$ and $\{6\}$. In 2^{nd} step, we find a star $\{3, 8, 9\}$ in the grey frame. It can be decomposed to one path $\{8, 3, 9\}$. Then the remaining graph is a star. It can be decomposed to one path $\{1, 4\}$. So, these 4 paths compose the minimal path cover of the tree in Figure 9.

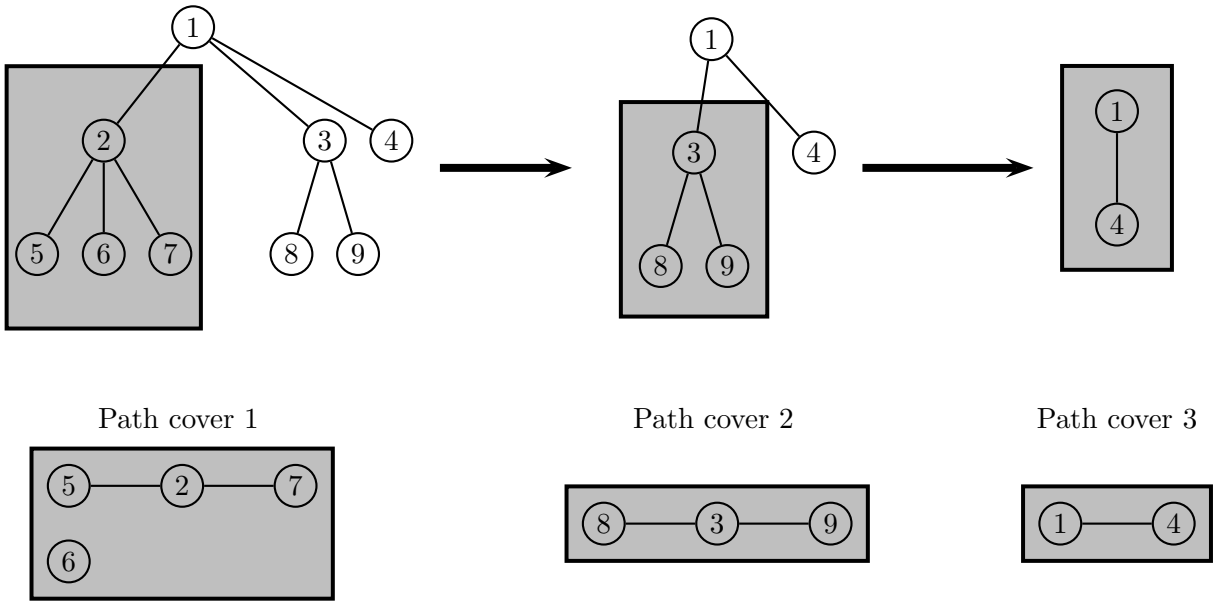


Figure 9: Example for Greedy Method

Problem G: Colorful Lights Party

Brief

Program Name: `PT07G`
Method Summary: Find the graceful labeling of a tree
Time Complexity: $O(?)$

Editorial

Nobody solved this problem. However, it's actually an easy problem. The most of people only dare solve that kinds of problems, whose complexity can be estimated. The simple search techniques will be used for this problem. Certainly the complexity of the search can't be estimated. But in fact, the search will be much faster than your imagination. It, as it were, just challenges your courage.

Solution

The concept of graceful labeling of trees and graphs was introduced by Rosa [12]. Afterward, the graceful tree conjecture was suggested. So far, no proof of the truth or falsity of the conjecture has been found.

Conjecture G.1 (The Graceful Tree Conjecture, Ringel-Kötzig Conjecture, Ringel 1964) [13] *Any tree is graceful, i.e. any tree has a graceful labeling.*

This problem gives two kinds of queries. One is general trees, another is a kind of special trees.

1. General trees

The search techniques could be better than thinking out a construction method for this "hard" problem. Especially, n in this kind of queries is equal to or less than 27, which is quite a small number. Due to approach the solution very fast, Hill-climbing techniques would be an effective method.

Hill climbing is an optimization technique which belongs to the family of Local search (optimization). Hill climbing attempts to maximize (or minimize) a function $f(x)$, where x are discrete states. In each step of hill climbing, all successors x' of x are generated and the one x' that is

[12] Rosa, A. 1967, *On certain valuations of the vertices of a graph*, in *Theory of Graphs*(International Symposium, Rome, July 1966), Gordon and Breach, New York, pp. 349-355.

[13] Ringel, G. 1964, *Problem 25*, in *Theory of Graphs and its Applications*, *Proceedings Symposium Smolenice*, Prague.

closest to the solution will be chosen as the next x. This process is like hill-climbing in steepest ascent.

Notation G.2 For a given tree T and labeling L of the vertices, let $f(T, L)$ be the number of distinct edge labels. ■

The aim of the search is to find L such that $f(T, L) = |T| - 1$, i.e. to maximize $f(T, L)$. The hill climbing algorithm is given below.

Algorithm 5 Hill-climbing and Tabu Search algorithm

Require: T is a tree. M is the tabu searching limit step.

```
1: function GracefulLabeling(T, M)                    ▷ return the graceful labeling L.
2:     L = Any labeling of T.
3:     while f(T, L) < |T| − 1 do
4:         Let Q be a set of all pairs (u, v) which has not been chosen during the most recent M
   times.
5:         if there exists a pair (u, v) ∈ Q, such that f(T, Swap(L, (u, v))) > f(T, L) then
6:             L ← Swap(L, (u, v))
7:         else
8:             Find a pair (u, v) ∈ Q that maximizes f(T, Swap(L, (u, v)))
9:             L ← Swap(L, (u, v))
10:        end if
11:    end while
12:    return L
13: end function

14: function Swap(L, (u, v))
15:            ▷ return the labeling L after swapping the labeling of the vertex u and the vertex v.
16:    swap the values of L_u and L_v
17:    return L
18: end function
```

The parameter M of the tabu searching limit step should be chosen from 10 to 20 for such small n. The purpose of M is to prevent the algorithm from repeatedly cycling around within some small set of labelings. For the most of testcases, this algorithm can find the solution in only < 0.1 seconds.

2. Trees of m-star

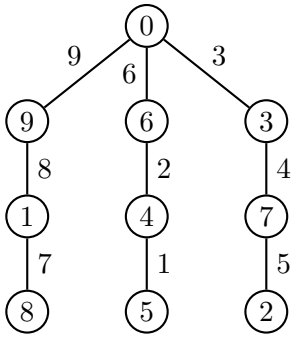


Figure 10: Here is the graceful labeling of the m-star with $m = 2, n = 3$

An m-star has a single root node with n paths of length m attached to it.

The construction method of the graceful labeling of m-star is easy to think out. Figure 10 shows how the algorithm given below works.

Algorithm 6 Graceful labeling on m-star

Require: T is a m-star tree.

```
1:  procedure GracefulLabelingOnMStar(T)          ▷ Print the graceful labeling of T.
2:      head ← 1
3:      tail ← (m + 1) · n
4:      print 0
5:      for i = 0 to m do
6:          if i is even number then
7:              for j = 0 to n − 1 do
8:                  print tail − j · (m + 1)
9:              end for
10:             tail ← tail − 1
11:         else
12:             for j = 0 to n − 1 do
13:                 print head + j · (m + 1)
14:             end for
15:             head ← head + 1
16:         end if
17:     end for
18: end procedure
```

Problem H: Search in XML

Brief

Program Name: `PT07H`
Method Summary: Tree Pattern Matching
Time Complexity: $O(n \cdot m)$

Editorial

Five contestants solved this problem. They all used the naive $O(n \cdot m)$ algorithm. In fact, it's hard to generate the pretty tough testdata.

This is a classical problem, which can be indeed solved in linear time by a complicated algorithm [14]. Some near-linear algorithm were also suggested [15] [16] [17]. The $O(nm^{0.5} \log m)$ method was suggested earlier [18].

Source

J. T. Yao, M. Zhang. *A Fast Tree Pattern Matching Algorithm for XML Query.* Department of Computer Science, University of Regina, Regina, Saskatchewan, Canada S4S 0A2

Solution

At least 7 characters (`<?></?>`) are needed for a vertex in tree. The number of vertices is about $100k \,/\, 7 \approx 15000$. In worst case ($n = 10000, m = 5000$), the quantity of operation is about $(n - m) \cdot m = 25 \times 10^6$. In fact, due to the restriction of the labeling of vertices, the quantity of operation may be much less than the theoretical value. So the naive algorithm is rather reasonable for this problem.

Firstly, we bulid the tree in $O(n \log n)$ time because we should sort each vertices' children ordered by their values. Only in this way, we can check whether two trees match in linear time. Secondly, we enumerate all the vertices in text tree as roots and check whether pattern tree matches it.

[14] R. Cole, R. Hariharan. *Tree Pattern Matching to Subset Matching in Linear Time*
[15] R. Cole, R. Hariharan. *Tree Tree Pattern Matching and Subset Matching in Deterministic $O(n \log^3 m)$ Time*
[16] R. Cole, R. Hariharan. *Tree Pattern Matching and Subset Matching in Randomized $O(n \log^3 m)$ Time*
[17] R. Cole, R. Hariharan, P. Indyk. *Fast Algorithms for Subset Matching and Tree Pattern Matching*
[18] M.Dubiner. Z Galil, E.Magen, *Faster tree pattern matching*, Journal of the ACM (1994) vol.41, no.2, p.205-213

Algorithm 7 Tree Matching

Ensure: return whether the subtree of u in text tree matches the subtree of v in pattern tree

```
1: function MATCH(u, v)
2:     i ← the first element of Child(u).
3:     ret ← True
4:     for all j ∈ Child(v) do                ▷ Child(v) is a sorted sequence
5:         while L_i ≠ L_j do                 ▷ Seek the child i of u that matches j
6:             i ← i + 1
7:             if i is out of Child(u) then
8:                 return False
9:             end if
10:        end while
11:        ret ← ret and MATCH(i, j)
12:    end for
13:    return ret
14: end function
```

Problem I: The Ants in a Tree

Brief

Program Name: `PT07I`
Method Summary: Least Common Ancestor
Time Complexity: $O(n \log n + m^2)$

Editorial

It's a partly scoring problem. Five contestants got full mark. Because of the large testdata and the strict time limit, few contestants could get the full mark even though they are correct.

Solution

Each ant has its traveling path. If two ants can meet, they must meet on the common part of their paths. Thus, we can determine that the main issues of our algorithm are to enumerate each pair of ants, then to find the common path of their paths and to judge whether they meets on that common path.

Now we consider two ants who travel from a to b and from c to d respectively. Above all, we suppose their common path is not empty and ignore the path direction. The task now is to find the endpoint u, v of the common path from u to v.

Definition I.1 *We call u is the **entry** of c to the path from a to b, such that u is the vertex on the path from a to b, which is nearest to c.*

Lemma I.2 *If the common path of the path from a to b and the path from c to d is not empty, the entries of c and d must be the endpoints of the common path.*

Due to the lemma, the problem reduces to find the entries of c and d respectively. Now we concentrate finding c's entry u on the path from a to b.

Let $r = \mathrm{LCA}(a, b)$.

1. If c is not in the subtree rooted by r, i.e. $\mathrm{LCA}(c, r) \neq r$, then the entry $u = r$. (See Figure 11)

2. Else if the entry u is on the path from r to b except r, i.e. $\mathrm{LCA}(c, b) \neq r$, then the entry $u = \mathrm{LCA}(c, b)$ (See Figure 12)

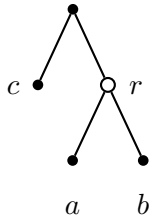


Figure 11: c is not in the subtree rooted by r

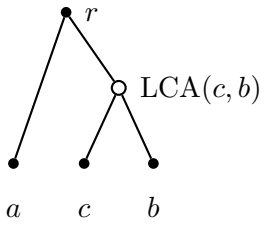
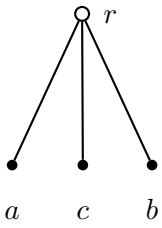


Figure 12: the entry u is on the path from r to b except r



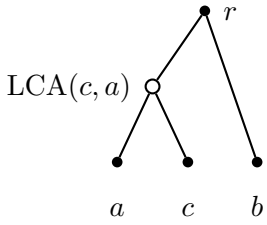


Figure 13: the entry u is on the path from r to a

3. Else the entry $u = \text{LCA}(c, a)$ (See Figure 13)

In the same way, we also get d's entry v. Thus, both endpoints of common path can be found. However, we still don't know whether the entries are on the path from c to d because we supposed the common path is not empty before. In next step, we should check whether u and v on the path from c to d.

Let $r' = \text{LCA}(c, d)$. If u is in the subtree leaded by r', i.e. $\text{LCA}(r', u) = r'$ and u is on the path from c to r' or on the path from d to r', i.e. $\text{LCA}(u, c) = u$ or $\text{LCA}(u, d) = u$, then u is on the path from c to d.

The remain work is easy. Just judge whether two ants meet on the path.

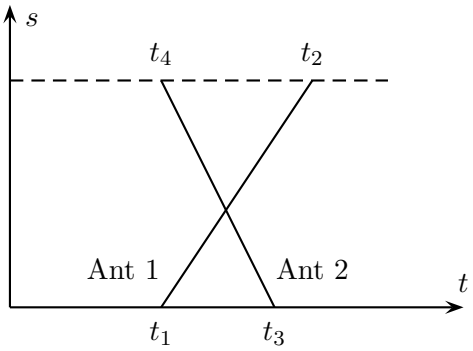


Figure 14: Judge whether two ants meet on the path.

Let

$$t_1 = d(a, u) \,/\, \text{Speed}_{\text{Ant 1}}$$
$$t_2 = d(a, v) \,/\, \text{Speed}_{\text{Ant 1}}$$
$$t_3 = d(c, u) \,/\, \text{Speed}_{\text{Ant 2}}$$
$$t_4 = d(c, v) \,/\, \text{Speed}_{\text{Ant 2}}$$

In Figure 14, the axis s denotes the distance to u and the axis t denotes the time passed from beginning. The movement of the ant is represented as a segment in the figure. If two ants meet on the path, there must exist one intersecting point of two movement segments of two ants.

LCA can be implemented by RMQ online algorithm in $O(n \log n) - O(1)$ time. Thus, the problem can be solved totally in $O(n \log n + m^2)$ time.

Problem J: Query on a tree III

Brief

Program Name:	`PT07J`
Method Summary:	Application of data structures
Time Complexity:	$O(n \log n + m \log^3 n)$ or $O((n+m) \log n)$

Editorial

It's not an easy problem. But to our surprise, 27 contestants solved this problem.

There are many methods for this problem. The most of contestants used online algorithm using segment tree and binary search in $O(n \log n) - O(\log^3 n)$ time. **chenqifeng** (chenqifeng) and **a123** (A123) both used off-line algorithm using the balanced trees in $O((n+m) \log n)$ time.

Solution

Method 1 - Segment Tree and Binary Search

Above all, find the sequence S of the depth-first searching of the tree. One query (x, k) is equivalent to find the k^{th} largest element in the continuous interval in the sequence that consists of x and all its posterities.

We can binary search for the answer's value v in $O(\log n)$ iterations. In each iteration during binary search, we should know how many elements are less than or equal to v (called v's rank) in that segment in the sequence. This operation can be implemented by a segment tree. Each segment $[l, r]$ in the segment tree stores a sorted sequence, which consists of the continuous interval of S from S_l to S_r. Thus, for each segment, we can binary search for v's rank in its sorted sequence in $O(\log n)$ time. Totally, there are $O(\log n)$ segments in the segment tree for the interval of a query. According to above stated, each query can be answered in $O(\log^3 n)$ time.

Now on the part of building the segment tree. If we sort the sequence in each segment using quicksort, the time complexity in total will be $O(n \log^2 n)$. However, due to the property of the segment tree, we can sort the sequence in each segment by merge its two children's sorted sequences like mergesort. Thus, only $O(n \log n)$ time is needed to build the segment tree.

Method 2 - Balanced Tree

For each subtree rooted at u, if we know the balanced tree that consists of u and all its posterities, we will answer all queries on u easily. The algorithm below may be very obvious.

Algorithm 8 Balanced Tree Method

```
1: function DFS(u)                ▷ return a balanced tree that consists of u and all its posterities
2:     ret ← ∅
3:     for all v ∈ Child(u) do
4:         ret.Union(DFS(v))
5:     end for
6:     for all (u,k) ∈ Query(u) do          ▷ Query(u) denotes the set of the queries on u
7:         answer ret.Locate(k)
8:     end for
9:     return ret
10: end function
```

But the difficulty is to merge the balanced trees. The usual balanced trees don't have "merge" operation. We have to insert vertices one by one. The algorithm will perhaps degenerate to $O(n^2 \log n)$.

chenqifeng (chenqifeng) used size balanced tree, which doesn't have "merge" operation. But he controlled the merging direction based on the principle that the small tree should be merged into the large tree. Namely, each time, the other children's posterities are inserted to the largest child's balanced tree. Thus, each vertex will be inserted $O(\log n)$ times at most. The time complexity $O((n+m)\log n)$ can be guaranteed.

In other ways, we can also use some special balanced trees, e.g. splay tree. **a123** (A123) used *join* operation of splay tree directly. The time complexity $O((n+m)\log n)$ can be guaranteed in amortized analysis of splay tree, either. However, Splay tree with *join* operation may be hard to implement for a majority of contestants.

Problem K: Balloons of JiaJia

Brief

Program Name:	`PT07K`
Method Summary:	Tree edit distance
Time Complexity:	$O(n^5)$

Editorial

Only Two contestants solved this problem. They are **mountainking** (Lou Tiancheng) and **codrut** (Grosu Codrut).

Solution

This problem is also a classical problem [19] [20] [21] [22].

Notation K.1 A *tree* T is a node (called the root) connected to an ordered sequence of disjoint trees.

A *forest* F is an ordered sequence of trees, denoted by $F = \{F_1, F_2, \cdots, F_n\}$.

- F_i denotes the i^{th} tree of the forest F.
- $|F|$ denotes the number of the trees in the forest F.
- $C(T)$ denotes a forest composed of all children of the tree T.
- $r(T)$ denotes the root of the tree T.
- $F - T$ denotes to remove the tree T in the forest F. ■

Theorem K.2 *Let F and G be two forests. The edit distance between F and G, denoted $d(F, G)$, is the minimal cost of edit operations needed to transform F into G. We have:*

[19] Philip Bille, *Tree Edit Distance, Alignment Distance and Inclusion*

[20] Philip Bille, *A Survey on Tree Edit Distance and Related Problems*

[21] Erik D. Demaine, Shay Mozes, Benjamin Rossman, and Oren Weimann, *An $O(n^3)$-Time Algorithm for Tree Edit Distance*

[22] Serge Dulucq and Hélène Touzet, *Analysis of Tree Edit Distance Algorithms*

$$d(F,G) = \min \begin{cases} c_{rel}(r(F1), r(G1)) + d(F - F_1, G - G_1) + d(C(F_1), C(G_1)) & \text{(K.1)} \\ c_{del}(r(F1)) + \min\limits_{1 \le i \le |G|} \left[\begin{array}{l} d(C(F_1), \{G_1, G_2, \ldots, G_i\}) \\ +d(F - F_1, \{G_{i+1}, G_{i+2}, \ldots, G_{|G|}\}) \end{array} \right] & \text{(K.2)} \\ c_{ins}(r(G1)) + \min\limits_{1 \le i \le |F|} \left[\begin{array}{l} d(\{F_1, F_2, \ldots, F_i\}, C(G_1)) \\ +d(\{F_{i+1}, F_{i+2}, \ldots F_{|F|}\}, G - G_1) \end{array} \right] & \text{(K.3)} \end{cases}$$

■

1. Equation K.1 represents the "**relabel**" operation (**relabel** $r(F)$ to $r(G)$). $d(C(F_1), C(G_1))$ denotes the edit distance between their children forests. $d(F - F_1, G - G_1)$ denotes the edit distance between the remaining trees in the forests. Figure 15 shows how it works.

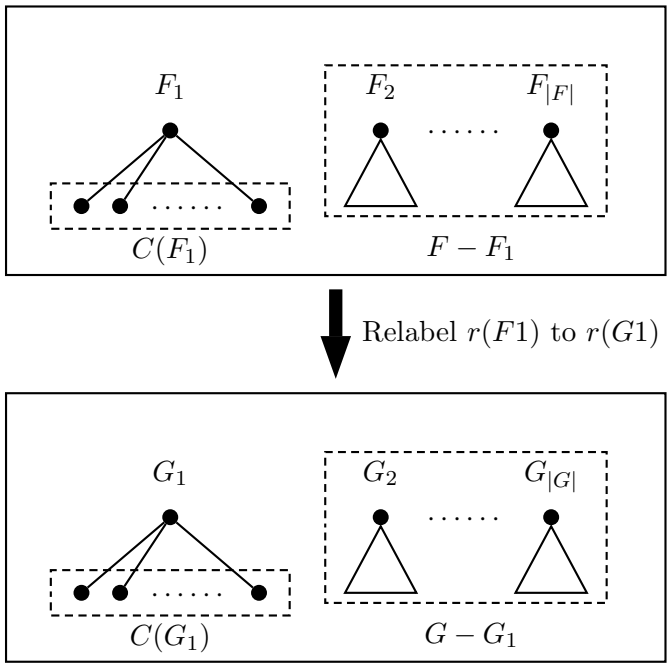


Figure 15: Explanation for Equation K.1

2. Equation K.2 represents the "**delete**" operation (**delete** $r(F)$). Then we should enumerate i, let $\{G_1, G_2, \ldots, G_i\}$ correspond to the children forest of F_1. $d(F - F_1, \{G_{i+1}, G_{i+2}, \ldots, G_{|G|}\})$ denotes the edit distance between the remaining trees in the forests. Figure 16 shows how it works.

3. Equation K.3 represents the "**insert**" operation (**insert** $r(G)$ into F, equivalent to **delete** $r(G)$ from G). In the same way as Equation K.2, we should enumerate i, let $\{F_1, F_2, \ldots, F_i\}$ correspond to the children forest of G_1. $d(\{F_{i+1}, F_{i+2}, \ldots F_{|F|}\}, G - G_1)$ denotes the edit distance between the remaining trees in the forests.

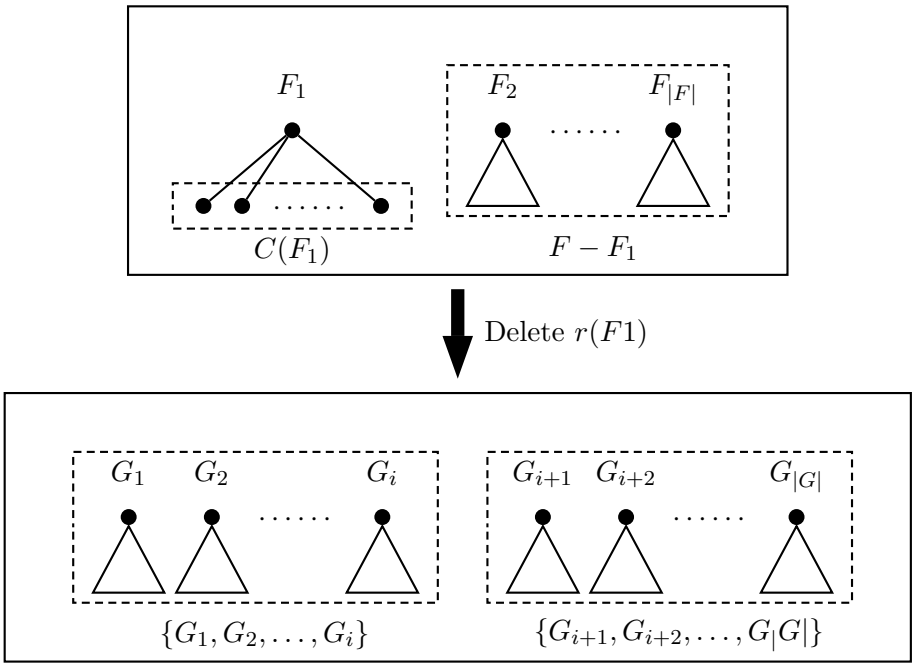


Figure 16: Explanation for Equation K.2

Consider the state $d(F, G)$. F and G must be the continuous siblings in the original trees. Therefore, the number of the total states is $O(n^2) \times O(n^2) = O(n^4)$. Time for each transformation is $O(n)$. So the time complexity is $O(n^5)$. In most of cases, the degree of the tree is less than $O(n)$, the actual effect is better than theoretical value.

Problem X: Vertex Cover

Brief

Program Name: `PT07X`
Method Summary: Minimal vertex cover set in a tree
Time Complexity: $O(n)$

Solution

We use dynamic programming to solve this problem. The minimum cover number of the subtree of the root x denotes $f_1(x)$ if x is in the cover set and denotes $f_0(x)$ if x is not in the cover set.

$$f_1(x) = \sum_{c \in \text{Child}(x)} \min\{f_0(c), f_1(c)\} \tag{X.1}$$

$$f_0(x) = \sum_{c \in \text{Child}(x)} f_1(c) \tag{X.2}$$

If x is in the cover set, whether the children of x is in the cover set or not, the edges between x and its children are all covered (Equation X.1). If x is not in the cover set, the children of x must be in the cover set (Equation X.2).

Consequently, we can solve it in a linear time.

Problem Y: Is it a tree

Brief

Program Name: `PT07Y`
Method Summary: Tree Recognition
Time Complexity: $O(n)$

Solution

Lemma Y.1 (Tree Recognition) [23] *A tree with n nodes has $n-1$ graph edges. Conversely, a connected graph with n nodes and $n-1$ edges is a tree.*

If the graph doesn't have $n-1$ edges, print "NO". In next step, we should check the connectivity of the graph. Both of the following methods are okay.

1. Traverse (*Depth First Search*) the graph from any node. If all nodes are visited, print "YES". Otherwise, print "NO".
2. Use *Disjoint Set* to merge the end nodes of each edge. If the end nodes of a certain edge are in the same set before merging, print "NO".

[23] Weisstein, Eric W. "Tree." From MathWorld - A Wolfram Web Resource. http://mathworld.wolfram.com/Tree.html

Problem Z: Longest path in a tree

Brief

Program Name: `PT07Z`
Method Summary: Find the longest path in a tree
Time Complexity: $O(n)$

Solution

Based on the tree's property, we just traverse (*Depth First Search*) the tree twice to find the longest path. Two steps below are needed.

1. Traverse the tree from any node u, and find the farthest node v away from the node u (the longest path from the node u).
2. Traverse the tree from the node v that we get last time, and find the farthest node w away from the node v (the longest path from the node v).

The path between v and w is just the longest path in the tree.

Theorem Z.1 *According to the algorithm above, the node v we get in the first step is one of the end nodes of the longest path.*

Proof Denote the distance between u and v in the tree by $d_{u,v}$. Let the path between v and w is the longest path in the tree.

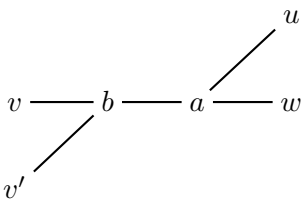


Figure 17: Algorithm for finding the longest path

Suppose we get the node v' but not the node v in the first step, i.e. v' is farther than v from u ($d_{u,v'} > d_{u,v}$). Because $d_{u,b}$ is the common part of two paths, we get $d_{b,v'} > d_{b,v}$. Thus, we have

$$d_{v,w} = d_{v,b} + d_{b,w} > d_{v',b} + d_{b,w} = d_{v',w}$$

Namely, the path between v' and w is longer than the path between v and w. It's conflicted with the supposition. ■

The correctness of the algorithm is proved above.