Readings about DP on Trees

Diego Linares - kiwiAipom

I. DP on Trees Tutorial

Pre-requisites for the reading:

- Basic dynamic programming: optimal substructure property and memoisation.
- Trees (basic DFS, subtree definition, children, etc.)

DP definition: solve problems by breaking them down into overlapping sub-problems which follow the optimal substructure.

We define functions for nodes, which we calculate recursively based on children of a node. One of the states of the **DP** usually is a node i, so we are solving for its subtree.

A. Problem 1

Tree T with N nodes, each node i has C_i coins attached to it. Choose a subset of nodes such as no two nodes are *adjacent*, and the sum of coins is max.

A similar problem was for an $A = \{a_1, a_2, \ldots, a_n\}$. We chose $dp(i)$ as our answer for $a_1, a_2, \ldots, a_i$. The recurrence being $dp(i) = max(a_i + dp(i-2), dp(i-1))$ (choose a_i or not, basically).

In a tree first we need to root it. For example at node 1 and define $dp(V)$ as the answer for subtree with root V, then $dp(1)$ is the answer.

So if we include V we can't include their children, but we can include any grandchildren.

So the recurrency of $dp(V)$ is:

$$max(\sum_{i=1}^{n} dp(v_i), C_V + \sum_{i=1}^{n} \text{sum of } dp(j) \text{ for children } j \text{ of } v_i)$$

We can define two DP, $dp1(V), dp2(V)$, the first one is the number of coins that can be obtained from the subtree, and the second one, if we include the node or not. Final answer being $max(dp1(1) + dp2(1))$

Now the recursion is easier:

$$dp1(V) = C_V + \sum_{i=1}^{n} dp2(v_i)$$

Since the answer of a node is dependent on the one of its children, we need a recursive DFS implementation.

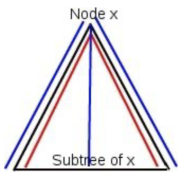
```
vector<int> adj[N] // contains all neighbors
    of i node.
int dp1[N], dp[2];
void dfs(int V, int pV) // pV is the parent of
     node V
{
    // for storing sums of dp1 and max(dp1,dp2
        ) for all children of V
    int sum1 = 0, sum2 = 0;
    // traverse over all children
    for (auto v: adj[V])
    {
        if (v == pV) continue;
        dfs(v,V);
        sum1 += dp2[v];
        sum2 += max(dp1[v], dp2[v]);
    }
    dp1[V] = C[V] + sum1;
    dp2[V] = sum2;
}
int main()
{
    int n;
    cin >> n;
    for(int i=1; i<n; i++)
    {
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    dfs(1,0); // this basically does the DFS
        for all the nodes.
    int ans =  max(dp1[1], dp2[1]);
    cout << ans << endl;
}
```

Complexity: $O(N)$.

B. Problem 2

Given a tree T of N nodes, longest path between two nodes (*diameter of the tree*). If we root node 1, there exists a node x such that:

- Longest path starts from node x and goes into the subtree. Denote this as $f(x)$. Blue
- Starts in the subtree of x, passes through it, and then back into the subtree. Denote this as $g(x)$. Red.



If we get the max from both for all nodes x we can calculate the diameter, but how can we calculate max path length in both cases?

A node V has n children $v_1 \ldots v_n$. $f(i)$ is the length of longest path that starts at node i and ends in its sub tree. So $f(V) = 1 + max(f(v_1), \ldots, f(v_n))$. Since it is the maximum path of their children, plus the father.

As you can see in DP on trees functions are for nodes and recursion is based on values of children.

Case 2: originate from subtree of v_i, go through V and end in the subtree of v_j, where $i \neq j$. To make the path maximum we'll make $f(v_i), f(v_j)$ to be maximum. So $g(V) = 1 +$ sum of two max from set $\{f(v_1), \ldots, f(v_n)\}$.

We calculate f through a DFS and on the go. It's done in $O(N)$

```
vector<int> adj[N];
int f[N],g[N], diameter;
void(int V, int pV)
{
    vector<int> fValues; // f for all children
         of V
    for(auto v: adj[V])
    {
        if(v == pV) continue;
        dfs(v,V);
        fValues.push_back(f[v]);
    }
    //Sort to get the two biggest values
    sort(fValues.begin(),fValues.end());
    f[V] = 1; // update for the current node
    if(!fValues.empty()) f[V] += fValues.back
        ();
    if(fValues.size() >= 2) // more than 2
        kids
        g[V] = 2 + fValues.back() + fValues[
            fValues.size()-2]; // the two
            biggest.
    diameter = max(diameter, max(f[V],g[V]));
}
```

C. Problem 3

Tree T with N nodes. Find number of **sub trees** of size $\leq K$.

Sub Tree (separated): connected subset of nodes from the original tree.

Suppose the tree is rooted at 1, $S(V)$ is the subtree rooted at node V (different definition from the one in the problem), since all the nodes are included.

How to count the number of sub trees of a tree? We define $f(V)$ as the number of sub trees of $S(V)$ (subtree with the node included). So for each child u of V we have the choice of including them in the sub tree (which there are $f(u)$ ways of doing) or not (just 1 way, because we wont choose its children).

Node V has children $v_1, \ldots, v_n$, so $f(V) = \prod_{i=1}^{n} 1 + f(v_i)$ (el producto del f de cada uno de los hijos). That function considers the subtrees **rooted** at V. We now define $g(V)$ for the ones not rooted at V, its recursion being:

$$\sum_{i=1}^{n} f(i) + g(i)$$

For each child we add to $g(V)$ number of ways to choose a subtree rooted or not at that child. So the final answer is $f(1) + g(1)$ (for this partial problem).

How many of these don't exceed K size? We add another state to the dp, $f(V,k)$ being the number of sub trees with k nodes and V as root. If a child $S(v_i)$ contributes a_i nodes, k is $\sum_{i=1}^{n} a_i$. If we want to form a sub tree of size $k+1$ (one node contributed by V), $f(V,k)$ is the sum of $\prod_{i=1}^{n} f(v_i, a_i)$ for all distinct sequences $a_1, \ldots, a_n$.

How to compute this? $dp1(i,j)$ is the way to choose j nodes from subtrees $v_1, \ldots, v_i$, and its recurrence is: $dp1(i,j) = \sum_{k=0}^{K} dp1(i-1, j-k) * f(i,k)$ (iteratiing over k assuming that subtree of v_i contains k nodes). Finally:

$$f(V,k) = dp1(n,k)$$

$$\sum_{i=1}^{K} f(V,i) \text{ for all nodes } V$$

In pseudo-code:

```
f[N][K+1]; // K+1 since you will include 0
void rec(int cur_node)
{
    f[cur_node][1] = 1
    dp_buffer[K] = {0}
    dp_buffer[0] = 1

    for(all v such that v is children of
        cur_node)
        rec(v)
        dp_buffer1[K] = {0}
        for i = 0 to K:
            for j = 0 to K - i:
                dp_buffer1[i+j] += dp_buffer[i
                    ]*f[v][j]
        dp_buffer = dp_buffer1
    f[cur_node] = dp_buffer
}
```

Complexity: n children, so $O(N * K^2)$

A similar problem: N nodeds and weight of each, delete enough nodes so it has K leaves. Cost is the weight of the node. Try to find the minimum cost. (*check Indian Computing Olympiad*)

D. Problem 4

Each node i has a cost C_i. Start in root, choose unvisited node at random until there are no unvisited. Cost of travel is the sum of costs so far. *Which node should be the root?*

$f(V)$ is expected total time if we start at node V and visit in its subtree.

$$f(V) = C_V + \frac{\sum_{i=1}^{n} f(v_i)}{n}$$

Now we find a node v such that $f(v)$ is minimized. It depends on the root of the tree. Brute force is too slow.

Idea: Iterate over all nodes V and calculate the value of $f(V)$ if the tree is rooted at it. *We need to see the contribution of $f(parent(V))$ if tree is rooted at V.* $g(V)$ is defined as the expected total time at $parent(V)$, if we don't consider the contribution of the subtree of V.

Total expected time now is:

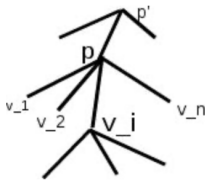
$$C_V = \frac{g(V) + \sum_{i=1}^{n} f(v_i)}{n+1}$$

To calculate $g(V)$ consiedr a node p with parent p' and children $v_1, \ldots, v_n$. To find $g(v_i)$, which means root tree at nodee p and **don't consider** subtree of v_i for calculating $f(p)$.

$$C_p + \frac{g(p) + f(v_i) + \ldots + f(v_{i-1}) + f(v_{i+1}) + \ldots + f(v_n)}{n}$$

$g(p)$ gives us the time at p' without considering the subtree of '. And we divide by the n since it is the number of adjacent

nodes of p, minus the one we are not considering.



f, g can be calculated recursively in $O(N)$.

E. Problem 5

You have to make T_1 as structurally similar to T_2 which can be done by inserting leaves in any of the trees. Minimum number of insertions to do so.
Suppose both are rooted at 1. T_{1_1} with children $u_1, \ldots, u_n$ and T_{2_1} with $v_1, \ldots, v_m$- We map nodes between sets, making $u_i = v_j$ in terms of structure, for some i, j, by addding the required nodes. If $n \neq m$ we add the whole subtree required.
How to choose which node to map? $dp(i, j)$ is the minimum additions required to make the subtree of i to the one of j.
So we gonna assign a node u_i to v_j, the cost gonna be $dp(u_i, v_j)$. We have to assign such that the cost is minimized, **assignment problem**. Cost matrix C in which $C(i, j)$ is the cost of assigning i to j. Solving this matrix can be done in $O(N^3)$, if there are N tasks. $C(i, j) = dp(u_i, v_j)$ so we can get $dp(i, j)$.
Complexity ends up being $O(N^3)$, N being the maximum number of nodes between the trees.

II. Another DP on Trees Tutorial

III. Algorithms Lecture 1: Sums and Expected Value

References

[1] *DP on Trees Tutorial*, darkshadow's blog, From: https://codeforces.com/blog/entry/20935