CS 341 - Algorithms, Spring 2017. University of Waterloo

Lecture 13 : Dynamic programming on trees

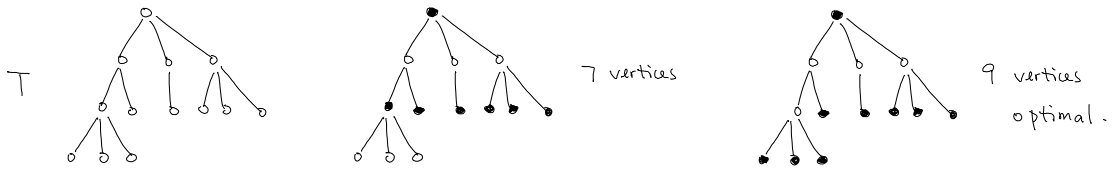
So far, the problems that we solved using dynamic programming has a "line" structure.

Today, we see two examples of using dynamic programming to solve problems with a tree structure.

Independent set on trees [DPV 6.7]

Input: A tree $T=(V,E)$.

Output: A maximum independent set $S \subseteq V$, where a subset of vertices $S \subseteq V$ is an independent set if $uv \notin E$ for all $u,v \in S$ and our objective is to find an independent set of maximum cardinality.



Having a tree structure suggests a natural way to do dynamic programming, by using the subtrees rooted at a vertex as subproblems and write recurrence relation of a parent and its children.

Actually, we have done this before. When we compute the low[] array for computing the cut vertices, we have already done some simple dynamic programming on trees.

Recurrence

For each vertex v in the tree, let $I(v)$ be the size of a maximum independent set in the subtree rooted at v.

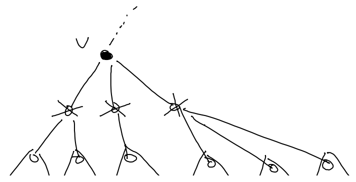
Then $I(\text{root})$ is the answer for the problem.

And $I(\text{leaf})=1$ are the base cases.

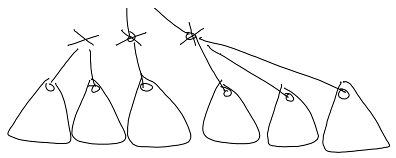
To compute $I(v)$ for an internal node v, we look at its children and its grandchildren and consider two possibilities:

① v is in the solution : Then, all its children cannot be put in the solution, as otherwise the set is not independent.

Then, the optimal way to extend the partial solution is to take the optimal solution for each grandchild.

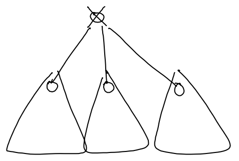


Then, the optimal way to extend the partial solution is to take the optimal solution for each grandchild.



So, in this case, the solution size is $1 + \sum_{w:w \text{ grandchild of } v} I(w)$.

② v is not in the solution: Then, to form an optimal solution, we are free to take any independent set in the subtrees rooted at its children.



Of course, the best way is to take optimal solutions for each child.

So, in this case, the solution size is $\sum_{u:u \text{ child of } v} I(u)$.

Therefore, $I(v) = \max\left\{ 1 + \sum_{w:w \text{ grandchild of } v} I(w) \,,\, \sum_{u:u \text{ child of } v} I(u) \right\}$.

<u>Correctness</u> follows from the justification of the recurrence relation and induction.

<u>Time complexity</u>: There are n subproblems, each subproblem on v requires $O(\#\text{ children} + \#\text{ grandchildren})$ lookups.

Note that $\sum_{v\in T} (\#\text{ children} + \#\text{ grandchildren}) = \sum_{v\in T} (\#\text{ parent} + \#\text{ grandparent}) = \sum_{v\in T} 2 \leq 2n$,

by counting the sum in a different way (i.e. instead of (grand)parent-child, use child-(grand)parent.)

By top-down memorization, the total time complexity is $O(n)$.

<u>Bottom-up implementation</u>: It should be clear how to do it from leaves to the root.

<u>Alternative recurrence relation</u>: Instead of writing a recurrence relation with the grandchildren we can write it in another way so that the recurrence relation involve only the children.

The idea is to use two states for a vertex v.

Let $I^+(v)$ be the size of a maximum independent set with v in the solution, and let $I^-(v)$ be the size of a maximum independent set with v not in the solution.

Then $I^+(v) = 1 + \sum_{u:u \text{ child of } v} I^-(u)$, and $I^-(v) = \sum_{u:u \text{ child of } v} \max\{I^+(u), I^-(u)\}$.

The final answer is $\max\{I^+(\text{root}), I^-(\text{root})\}$, and

the base cases are $I^+(\text{leaf}) = 1$ and $I^-(\text{leaf}) = 0$.

We leave the justifications for these recurrence relations as an exercise.

<u>Dynamic programming on tree-like graphs</u> (optional) [KT 10.4, 10.5]

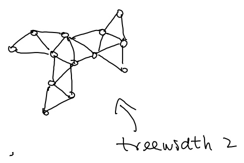
Dynamic programming on tree-like graphs (optional) [KT 10.4, 10.5]

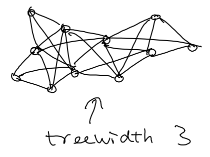
Many optimization problems are hard on graphs but easy on trees using dynamic programming.

The maximum independent set problem is an example, later we will see why it is hard in general.

Generalizing the ideas using dynamic programming on trees, it is possible to show that

dynamic programming on "tree-like" graphs. e.g.





There is a way to define the "tree-width" of a graph,

so as to measure how well a graph can be "approximated" by a tree.

If the tree-width is smaller, then dynamic programming is faster.

It has become an important paradigm to deal with hard problems on graphs, at least in

research papers.

See [KT 10.4, 10.5] if you are interested in this approach.

Optimal binary search tree [CLRS 15.5]

This problem is a little similar to the Huffman coding problem, searching for an optimal binary tree.

Imagine the following scenario: there are n commonly searched strings, say some French vocabularies,

where users look for their English meanings.

We would like to build a good data structure to support these queries.

And somehow we have decided to use binary search tree (as mentioned in piazza, one could

implement a static dictionary by using perfect hashing).

Since there are n strings, we could build a balanced binary search tree, so that each query

can be answered using at most $\log_2 n$ comparisons.

But, suppose we know that some strings are searched more frequently than others, can we

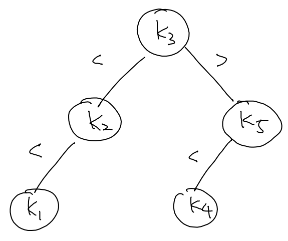
use this extra information to design a better binary search tree, so that the average

number of comparisons done is minimized?

Input: n keys $k_1 < k_2 < \dots < k_n$, frequencies $f_1, f_2, \dots, f_n$, with $\sum_{i=1}^{n} f_i = 1$.

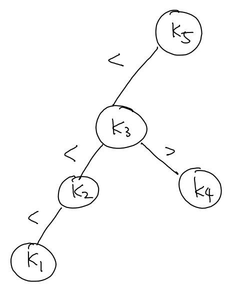
Output: a binary search tree T that minimizes the objective value $\sum_{i=1}^{n} f_i \, \text{depth}_T(k_i)$.

For example, given $f_1 = 0.1$ $f_2 = 0.2$ $f_3 = 0.25$ $f_4 = 0.05$ $f_5 = 0.4$,



objective value $= 0.25\times1 + 0.2\times2 + 0.4\times2 + 0.1\times3 + 0.05\times3$

$= 1.9$



objective value

$= 0.4\times1 + 0.25\times2 + 0.2\times3 + 0.05\times3 + 0.1\times4$

$= 2.05$

As the above example shown, even though k_5 has the highest frequency, it is not necessarily optimal to put k_5 at the root (that has the minimum depth).

So, it is unlike the prefix coding problem, that keys have higher frequencies will have smaller depth while keys with lower frequencies will have larger depth.

This is because we have to maintain the binary search tree structure, so that smaller keys have to be put on the left subtree and larger keys have to be on the right.

This restriction turns out to be useful in setting up the recurrence relation.

Recurrence

The subproblem structure is slightly different from those that we have seen before.

Let $C(i,j)$ be the objective value of an optimal binary search tree for keys $k_i < \dots < k_j$.

The answer that we are looking for is $C(1,n)$.

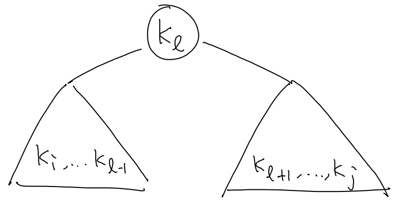
The base cases are $C(i,i) = f_i$, and $C(i,i-1) = 0$ for handling boundary cases.

Let $F_{i,j} = \sum_{l=i}^{j} f_l$. For $j < i$, we set $F_{i,j} = 0$ to handle boundary cases.

To compute $C(i,j)$, we try all the possible root of the binary search tree:

For $i \le l \le j$, if we set k_l to be the root, then all keys $k_i, \dots, k_{l-1}$ must be on the left subtree of the root, and $k_{l+1}, \dots, k_j$ must be on the right subtree of the root.

The two subtrees are independent of each other.



So, the best way is to find an optimal binary search tree for $k_i, \dots, k_{l-1}$ on the left, and an optimal binary search tree for $k_{l+1}, \dots, k_j$ on the right.

Therefore, $C(i,j) = \min_{i \le l \le j} \{ f_l + F_{i,l-1} + C(i,l-1) + F_{l+1,j} + C(l+1,j) \}$.

$$= \min_{i \le l \le j} \{ F_{i,j} + C(i, l-1) + C(l+1, j) \}.$$

Note that the terms $F_{i,l-1}$, $F_{l+1,j}$ are added because all the keys in $k_i, \dots, k_{l-1}$ and $k_{l+1}, \dots, k_j$ are put one level lower, and the two terms account for the increase in the objective value because of that.

Correctness follows from the justification of the recurrence formula and by induction.

Time Complexity There are at most n^2 subproblems, and each subproblem looks up at most n values. so the total time complexity is $O(n^3)$ by top down memorization.

Bottom-up implementation It requires some care to write it correctly.

We will solve the subproblems with $j-i=1$ first, and then $j-i=2$, and so on.

```
$C[i,i-1] = 0$   for   $1 \le i \le n$.             // boundary cases
Compute $F_{i,j}$ for all $1 \le i,j \le n$        // can be done in $O(n^2)$ time using partial sums.
for $1 \le width \le n-1$ do                        // from short intervals to long intervals
    for $i$ from 1 to $(n - width)$ do
        $j = i + width$.   $C(i,j) = \infty$.
        for $l$ from $i$ to $j$ do
            $C(i,j) \leftarrow \min\{C(i,j),\ F_{i,j} + C(i,l-1) + C(l+1,j)\}$.
```

It is clear that the time complexity is $O(n^3)$ since there are three for loops.

Tracing out solution: We leave it as an exercise to print out an optimal binary search tree.

Remark: It starts out as a tree problem, but eventually it becomes a problem on a line because we have to respect the key ordering. It is still interesting because it is the first problem where we use $C[i,j]$ as subproblems.