CS 341 Algorithms. Spring 2025. University of Waterloo

Lecture 13: Dynamic programming on trees

We will see two examples to use dynamic programming to solve problems on trees.

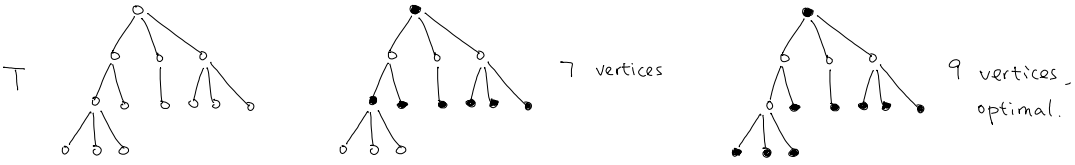
Independent Sets on Trees [DPV 6.7]

Given a graph $G=(V,E)$, a subset of vertices $S \subseteq V$ is called an independent set if $ij \notin E \ \forall i,j \in S$.

We will see that the problem of finding a maximum cardinality independent set is NP-Complete, but we can use dynamic programming to solve the problem in polynomial time on trees.

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

Output: An independent set $S \subseteq V$ of maximum cardinality.



Having a tree structure suggests a natural way to use dynamic programming.

We will define a subproblem for each subtree rooted at a vertex v.

The key point is that since there are no edges between different subtrees, we can solve the problem on each subtree separately and thus reduce to smaller subproblems.

Then we can write a recurrence relation between a parent and its children.

Actually, we have used this idea before.

When we compute the early array to compute all cut vertices, we have already used dynamic programming on trees.

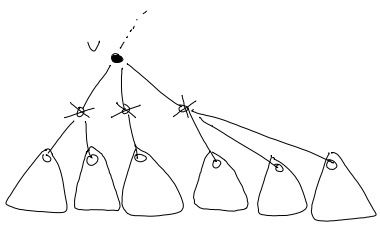
Dynamic Programming

Subproblems: Let $I(v)$ be the size of a maximum independent set in the subtree rooted at vertex v.

Answer: $I(\text{root})$.

Base cases: $I(\text{leaf})=1$ for all leaves in the tree.

Recurrence: To compute $I(v)$, we consider two possibilities.



① v is in the independent set.

Then all its children cannot be included in the independent set.

The optimal way to extend the current partial solution is to take a maximum independent set in each subtree rooted at its grandchildren.

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

② v is not in the independent set.

The optimal way to extend the current partial solution is to take a maximum independent set in each subtree rooted at its children.

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

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

Correctness follows from the explanation of the recurrence relation and induction.

Time complexity: There are n subproblems.

Each subproblem requires (# children + # grandchildren) lookups.

Note that $\sum_{v \in V} (\#\text{children} + \#\text{grandchildren}) = \sum_{v \in V} (\#\text{parent} + \#\text{grandparent}) \leq \sum_{v \in V} 2 = 2n$, by counting the sum in a different way (i.e. counting upward instead of counting downward).

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

Bottom-up implementation and printing out solution: Exercises.

Alternative recurrence relation: We can also write a recurrence relation involving children only.

The idea is to use two subproblems on a vertex v.

Let $I^+(v)$ be the size of a maximum independent set with v included, and

$I^-(v)$ be the size of a maximum independent set with v excluded.

Then the answer is $\max\{ I^+(\text{root}), I^-(\text{root}) \}$.

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

The recurrences are $I^+(v) = 1 + \sum_{w: w \text{ child of } v} I^-(w)$ and $I^-(v) = \sum_{w: w \text{ child of } v} \max\{ I^+(w), I^-(w) \}$.

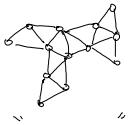
We leave the justification of these recurrences as an exercise.

Exercise: Extend the algorithm to solve the maximum weighted independent set problem on trees.

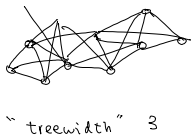
Dynamic Programming on "Tree-like" Graphs (optional) [KT 10.4, 10.5]

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

Generalizing the ideas using dynamic programming on trees, it is possible to show that dynamic programming also works on "tree-like" graphs, e.g.



"treewidth" 2



"treewidth" 3

There is a way to define the "tree-width" of a graph so as to measure how "close" a graph is to a tree.

If the tree-width is small, then dynamic programming works faster, usually with runtime $\sim O(n^{\text{treewidth}})$.

This has become an important paradiam to deal with hard problems on graphs, at least in research papers.

Read [KT 10.4, 10.5] for an introduction to this approach.

Optimal Binary Search Tree [CLRS 15.5]

This problem is a bit similar to the Huffman coding problem, also about finding an optimal binary tree.

Imagine the scenario where there are n commonly searched strings, e.g. some French vocabularies for English meanings.

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

And somehow we have decided to use a binary search tree (say instead of using hashing).

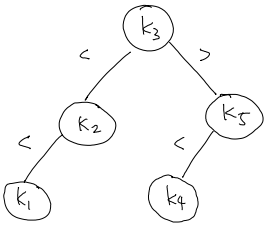
As there are n strings, we could build a balanced binary search tree to answer the queries in $O(\log_2 n)$ time.

As in Huffman coding, suppose we know the frequencies of the searched strings. Can we use this information to design a better binary search tree so as to minimize the average query time?

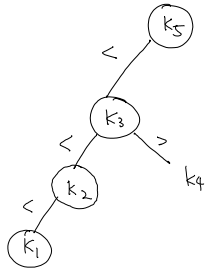
Input: n keys $k_1 < k_2 < \dots < k_n$, frequencies $f_1, f_2, \dots, f_n$ with $\sum_{i=1}^{n} f_i = 1$.

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

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
$= 0.25 \times 1 + 0.2 \times 2 + 0.4 \times 2 + 0.1 \times 3 + 0.05 \times 3$
$= 1.9$



objective value
$= 0.4 \times 1 + 0.25 \times 2 + 0.2 \times 3 + 0.05 \times 3 + 0.1 \times 4$
$= 2.05$

In the above example, even though k_5 has the highest frequency, it is not necessarily optimal to put k_5 at the root for it to have the minimum depth.

So, this is unlike the prefix coding problems, in which keys with higher frequencies will have smaller depth.

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

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

Dynamic Programming

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

In the following, we let $F_{i,j} = \sum_{\ell=i}^{j} F_\ell$. To handle boundary cases, we let $F_{i,j} = 0$ for $i > j$.

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

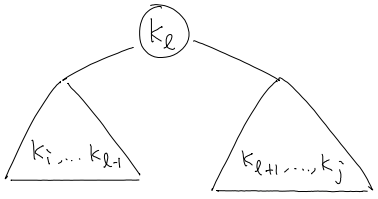
Answer : $C(1,n)$.

Base cases : $C(i,i) = f_i$ for $1 \le i \le n$. To handle boundary cases, we also set $C(i,i-1) = 0$ $\forall i$.

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

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

The two subtrees can be computed independently of each other, as there are no more constraints between the two subtrees.



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

Therefore, $C(i,j) = \min_{i \le \ell \le j} \{ \underbrace{f_\ell}_{\text{root}} + \underbrace{F_{i,\ell-1} + C(i,\ell-1)}_{\text{left subtree}} + \underbrace{F_{\ell+1,j} + C(\ell+1,j)}_{\text{right subtree}} \}$

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

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

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

Time Complexity There are no more than n^2 subproblems

Each subproblem looks up no more than n values.

Using top-down memorization, the total time complexity is $O(n^3)$.

Bottom-up implementation This problem requires more 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 ≤ i ≤ n                    // boundary cases
Compute F_{i,j} for all 1 ≤ i,j ≤ n              // can be done in O(n^2) time using partial sums
For 1 ≤ width ≤ n-1 do                           // from short intervals to long intervals
    For i from 1 to (n-width) do
        j = i + width.   C(i,j) = ∞.
        For ℓ from i to j do
            C(i,j) ← min { C(i,j), F_{i,j} + C(i,ℓ-1) + C(ℓ+1,j) }.
```

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

Tracing out solution : Exercise.

Faster Algorithm : With additional observations. Knuth used the same subproblems but showed how to solve the problem in $O(n^2)$ time!
