CS 341 – Algorithms

Lecture 13 – Dynamic Programming on Trees

30 June 2021

Today's Plan

1. Maximum Independent Sets on Trees

2. Optimal Binary Search Tree

HW 4 posted

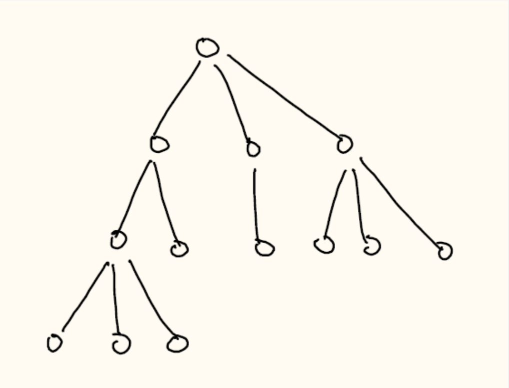
midterm median 72/80

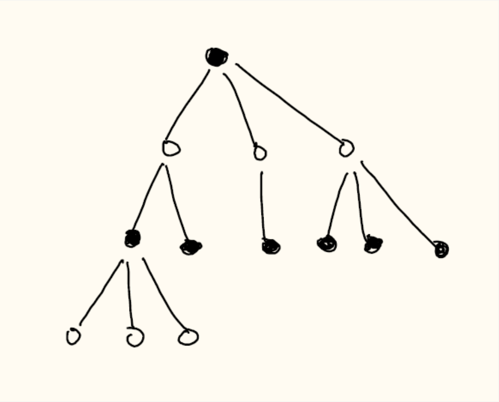
Maximum Independent Sets on Trees

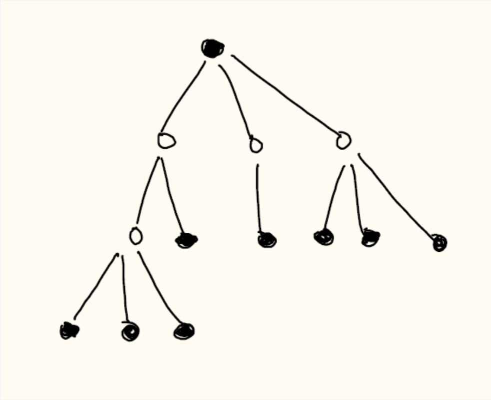
Given a graph $G = (V, E)$, a subset of vertices $S \subseteq V$ is an <u>independent set</u> if $uv \notin E$ for all $u, v \in S$.

<u>Input</u>: a tree $T = (V, E)$.

<u>Output</u>: an independent set of maximum cardinality.







greedy

In the last part of the course, we will see maximum independent set is NP-hard on general graphs.

Tree Structure

Have a tree structure gives a natural way to use dynamic programming.

We will define subproblems on each subtree.

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

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

We have already used this idea before in computing all cut vertices.

The low[] array we maintained is an example of doing dynamic programming on trees.

Recurrence

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

answer : $I(\text{root})$

base cases : $I(\text{leaf}) = 1$

recurrence : $I(v)$

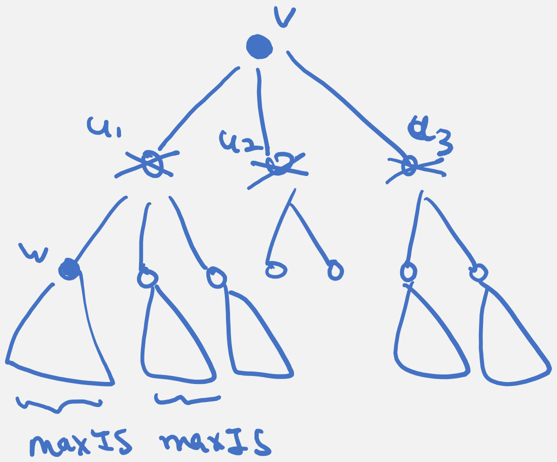
1 include v

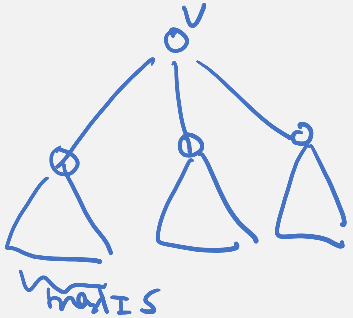
size $1 + \sum_{w:\, w \text{ grandchild of } v} I(w)$

2 not include v

size $\sum_{w:\, w \text{ child of } v} I(w)$

$I(v) = \max\{1, 2\}$.





Analysis

correctness : recurrence

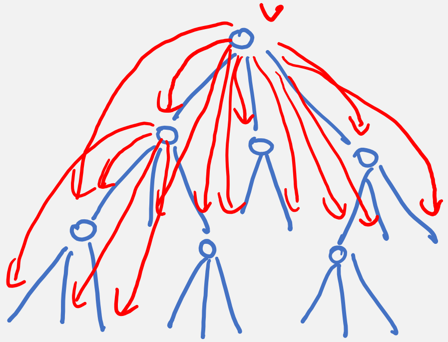
time complexity : $O(n)$ subproblems

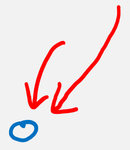
$$\sum_v \left(\underbrace{\#\text{children}(v)}_{\deg(v)} + \#\text{grandchildren}(v) \right)$$

$$= \sum_v \left(\#\text{parent}(v) + \#\text{grandparent}(v) \right)$$

$$\leq \sum_v 2 = 2n$$

total time complexity $O(n)$.





<u>Exercise</u>: Extend the algorithm to solve the maximum **weighted** independent set problem on trees.

Alternative Recurrence

We can also write a recurrence involving children only (i.e. no grandchildren).

<u>Subproblems</u>: Let $I^+(v)$ be the size of a maximum independent set with v included.

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

answer: $\max\{I^+(\text{root}), I^-(\text{root})\}$

base cases: $I^+(\text{leaf}) = 1$ $\quad I^-(\text{leaf}) = 0$ $\quad \forall$ leaf

recurrence: $$I^+(v) = 1 + \sum_{\substack{w:\, w \text{ child} \\ \text{of } v}} I^-(w)$$

$$I^-(v) = \sum_{\substack{w:\, w \text{ child} \\ \text{of } v}} \max\{I^+(w), I^-(w)\}$$

time: $O\left(\sum_v \deg(v)\right) = O(m+n) = O(n)$. $\square$

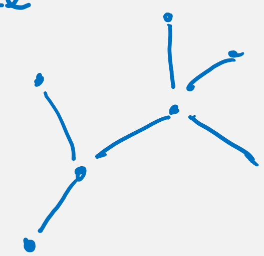
Dynamic Programming on Tree-Like Graphs

(optional)

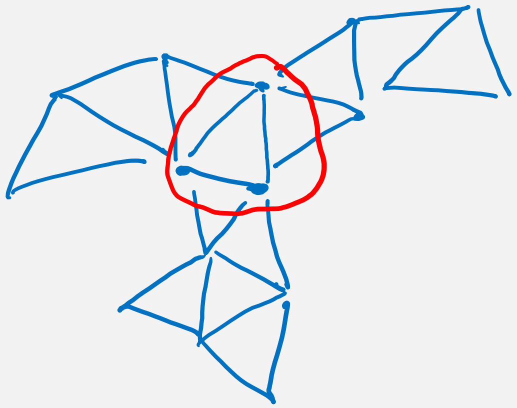
It is possible to generalize the idea to use dynamic programming on “tree-like” graphs.

There is a notion called “tree-width” which is very popular in theoretical computer science.

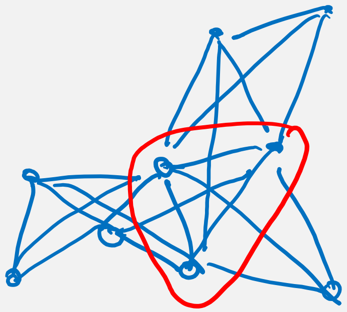
tree



2-tree



3-tree



Subgraph of k-tree = treewidth k

runtime $\approx n^k$

Today’s Plan

1. Independent Sets on Trees

2. Optimal Binary Search Tree

Setup

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

Consider the scenario we have n commonly search strings, e.g. French vocabularies.

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

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

We could build a full binary tree to support queries in $O(\log_2 n)$ time.

As in Huffman coding, suppose we know the frequencies of each search string.

Can we design a better binary search tree so as to minimize average search time?

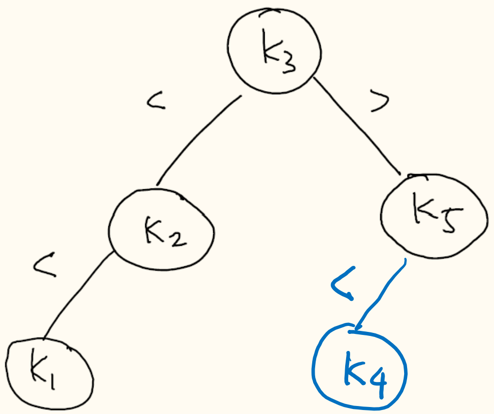
Optimal Binary Search Tree

Input: n keys $k_1 < k_2 < \cdots < k_n$, frequencies $f_1, f_2, \ldots, f_n \geq 0$ with $\sum_{i=1}^{n} f_i = 1$.

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

= expected query time

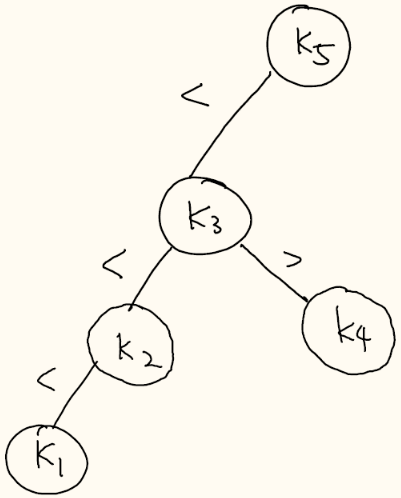
Example: $f_1 = 0.1$, $f_2 = 0.2$, $f_3 = 0.25$, $f_4 = 0.05$, $f_5 = 0.4$.



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

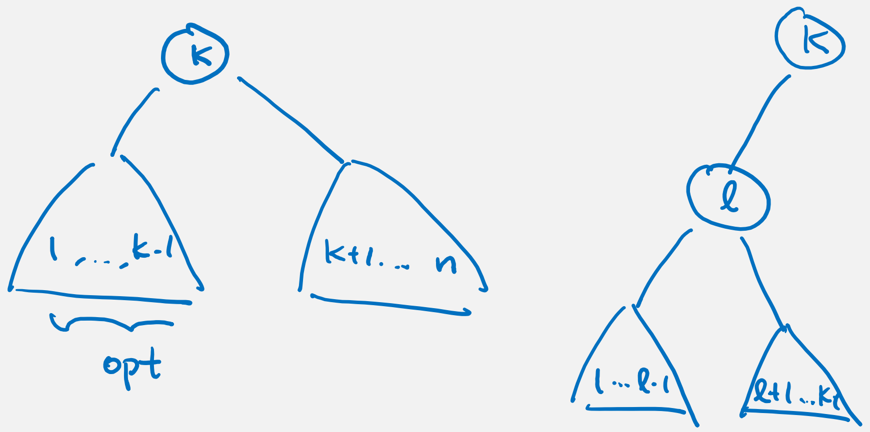
$= 0.4 \times 1 + 0.25 \times 2 + 0.2 \times 3 + 0.05 \times 3 + 0.1 \times 4$

$= 2.05$

Dynamic Programming

The restriction of maintaining a binary tree structure leads to a nice recurrence relation.

idea: try the root



<u>Subproblems</u>: Let $C(i, j)$ be the objective value of an optimal binary search tree with keys $k_i < \cdots < k_j$.

answer: $C(1, n)$

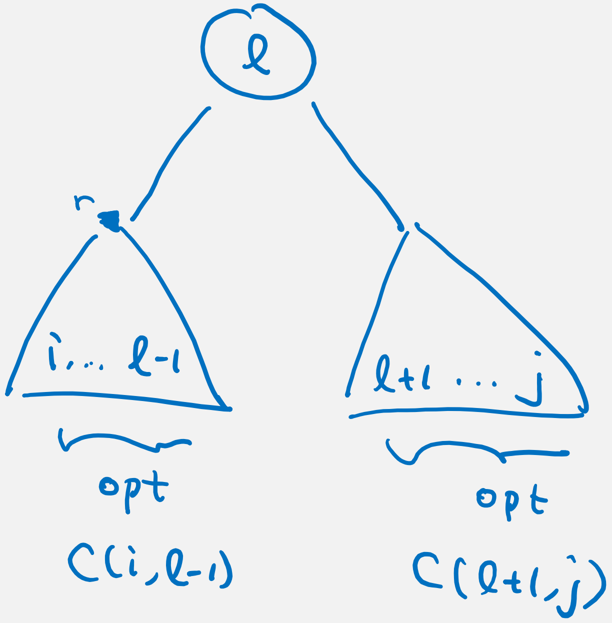
base cases: $C(i,i) = f_i \quad \forall i$, $\quad C(i, i-1) = 0$

Recurrence

$$C(i,j)$$

$$= \min_{i \le \ell \le j} \left\{ f_\ell + \left(C(i,\ell-1) + \sum_{k=i}^{\ell-1} f_k \right) + \left(C(\ell+1,j) + \sum_{k=\ell+1}^{j} f_k \right) \right\}$$

$$= \underbrace{\sum_{k=i}^{j} f_k}_{\text{fixed term}} + \min_{i \le \ell \le j} \left\{ C(i,\ell-1) + C(\ell+1,j) \right\}$$



Analysis

time complexity: $< n^2$ subproblems

$\min_{i \le l \le j}$ each subproblem $O(n)$ time

total time complexity $O(n^3)$

Faster Algorithm: Knuth gave a $O(n^2)$-time algorithm using the same subproblems,

but proved additional properties to achieve faster computation.

Bottom-Up Implementation

$C(i, i-1) = 0$ for $1 \le i \le n$

$C(i,i) = f_i \;\; \forall i$ base case

Compute $F_{i,j}$ for all $1 \le i, j \le n$

$F_{i,j} = \sum_{\ell=i}^{j} f_\ell$

For $1 \le \text{width} \le n-1$ do

For i from 1 to $(n - \text{width})$ do

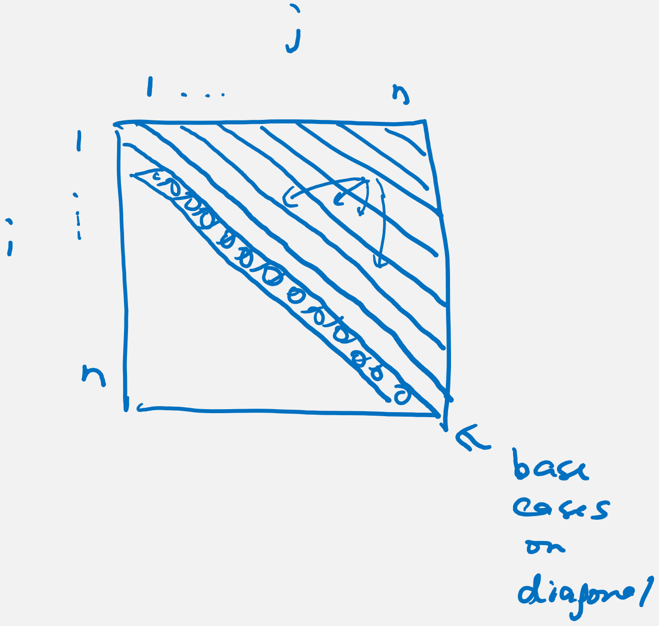
$j = i + \text{width}$. $C(i,j) = \infty$.

$j - i = \text{width}$

min

For ℓ from i to j do

$C(i,j) \leftarrow \min\{ C(i,j),\; F_{i,j} + C(i,\ell-1) + C(\ell+1, j)\}$.



Time Complexity: $O(n^3)$

Exercise: Trace out an optimal solution.