Advanced Algorithms Analysis and Design

By

Nazir Ahmad Zafar

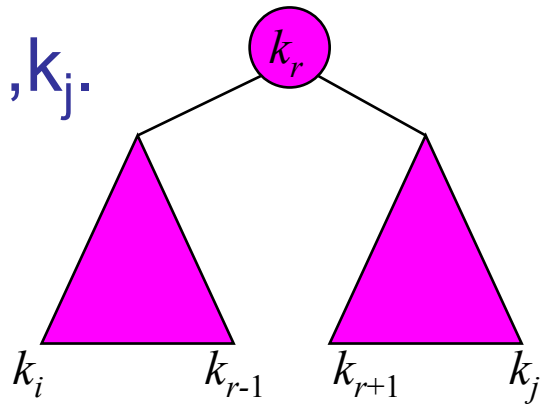
Lecture No 24

Optimal Binary Search Trees

Constructing Dynamic Programming

Construction: Optimal Substructure

- One of the keys in $k_i, \ldots, k_j$, say k_r, $i \le r \le j$, must be the root of an optimal subtree for these keys.
- Left subtree of k_r contains $k_i, \ldots, k_{r-1}$.
- Right subtree of k_r contains $k_{r+1}, \ldots, k_j$.



- To find an optimal BST:
  - Examine all candidate roots k_r , for $i \le r \le j$
  - Determine all optimal BSTs containing $k_i, \ldots, k_{r-1}$ and containing $k_{r+1}, \ldots, k_j$

Construction: Optimal Substructure

- Find optimal BST for $k_i,\ldots,k_j$, where $i \geq 1$, $j \leq n$, $j \geq i-1$
- When $j = i-1$, the tree is empty.
- Define $e[i, j]$ = expected search cost of optimal BST for $k_i,\ldots,k_j$.
- If $j = i-1$, then $e[i, j] = q_{i-1}$.
- If $j \geq i$,
  - Select a root k_r, for some $i \leq r \leq j$.
  - Recursively make an optimal BSTs
    - for $k_i,..,k_{r-1}$ as the left subtree, and
    - for $k_{r+1},..,k_j$ as the right subtree.

Lemma

Prove that when OPT tree becomes a sub-tree of a node then expected search cost increases by

$$w(i,j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$$

Proof

$$Total\ cost \text{ when a tree becomes subtree}$$

$$= \sum_{l=i}^{j} p_l(depth((k_l)+1+1) + \sum_{l=i-1}^{j} q_l(depth((d_l)+1+1)$$

$$= \sum_{l=i}^{j} p_l.(depth(k_l)+1) + \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l(depth(d_l)+1) + \sum_{l=i-1}^{j} q_l$$

$$= \sum_{l=i}^{j} p_l.(depth(k_l)+1) + \sum_{l=i-1}^{j} q_l(depth(d_l)+1) + \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$$

$$= total \text{ cost when tree was not subtree} + \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$$

Construction: Optimal Substructure

- When OPT subtree becomes a subtree of a node:
  - Depth of every node in OPT subtree goes up by 1.
  - Expected search cost increases by

$$w(i,j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$$

- If k_r is the root of an optimal BST for $k_i,..,k_j$

$$e[i, j] = p_r + (e[i, r-1] + w(i, r-1)) + (e[r+1, j] + w(r+1, j))$$
$$= e[i, r-1] + e[r+1, j] + w(i, j).$$

- But, we don't know k_r. Hence,

$$e[i,j] = \begin{cases} q_{i-1} & \text{if } j = i-1 \\ \min_{i \le r \le j} \{e[i, r-1] + e[r+1, j] + w(i,j)\} & \text{if } i \le j \end{cases}$$

Algorithm: Optimal Binary Search

- **OPTIMAL-BST(*p, q, n*)**
- **for** $i \leftarrow 1$ **to** $n + 1$
- **do** $e[i, i-1] \leftarrow q_{i-1}$.
- $w[i, i-1] \leftarrow q_{i-1}$.
- **for** $l \leftarrow 1$ **to** n
- **do for** $i \leftarrow 1$ **to** $n-l+1$
- **do** $j \leftarrow i + l-1$
- $e[i, j] \leftarrow \infty$
- $w[i, j] \leftarrow w[i, j-1] + p_j + q_j$
- **for** $r \leftarrow i$ **to** j
- **do** $t \leftarrow e[i, r-1] + e[r+1, j] + w[i, j]$
- **if** $t < e[i, j]$
- **then** $e[i, j] \leftarrow t$
- $root[i, j] \leftarrow r$
- return *e* and *root*

Consider all trees with *l* keys.

Fix the first key.

Fix the last key

Determine the root of the optimal sub-tree

Greedy Algorithms

Today Covered

- Introduction to Greedy Algorithms
  - Why Greedy Algorithm?
  - Where Greedy?
  - Where Greed Algorithm do not work?
- Activity Selection Problem
- Steps developing activity selection algorithm
  - Dynamic programming solution
  - Greedy choice
  - Recursive algorithm
  - Iterative algorithm
- Conclusion

Why Greedy Algorithm?

The algorithms we have studied in dynamic programming are relatively inefficient, for example

- Cost of 0-1 knapsack problem: O(nW)
- Cost of matrix chain multiplication: O(n3)
- Cost in longest common subsequence: O(mn)
- Optimal binary search trees: O(n3)

This is because

- We have many choices computing optimal solution.
- We check all of them in dynamic programming.
- We must think which choice is the best or
- At least restrict the choices we have to try.

What are Greedy Algorithm?

- In greedy algorithms, we do the same thing
- Mostly optimization algorithms go through a sequence of steps, with a set of choices at each step.
- In dynamic programming best choices is ignored
- Some times a simpler and efficient algorithm required.
- Greedy algorithms make best choice at a moment.
- It makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.
- Greedy algorithms do not always yield an optimal solutions, but mostly they do
- They tend to be easier to implement.

Where Greedy Algorithms do not work?

- Greedy choice:
  - We make the choice that looks best at a moment.
  - Every time make a choice, greedily maximize value
- An example where this does not work
  - Find longest monotonically increasing subsequence
  - Given sequence <3 4 5 17 7 8 9>
  - Longest such subsequence is <3 4 5 7 8 9>.
  - The greedy choice after choosing <3 4 5> is to choose 17, which is an unwanted element, results in the sequence <3 4 5 17> which is suboptimal.

Activity Selection Problem

Some Definitions

Closed Interval =

$$[a, b] = \{x \in R \mid a \leq x \leq b\}$$

Open Interval =

$$(a, b) = \{x \in R \mid a < x < b\}$$

Left Semi Open =

$$(a, b] = \{x \in R \mid a < x \leq b\}$$

Right Semi Open =

$$[a, b) = \{x \in R \mid a \leq x < b\}$$

Activity Selection Problem

- The problem involves scheduling of several competing activities that require exclusive use of common resource

Problem Statement

- The problem can be stated as, suppose we have a set:
  $S = \{a_1, a_2, ..., a_n\}$ of n proposed activities.
  - Each activity wish to use a resource which can be used by only one activity at a time.
  - Each activity a_i has starting time s_i, finishing time f_i

$$\text{where, } 0 \leq s_i < f_i < \infty$$

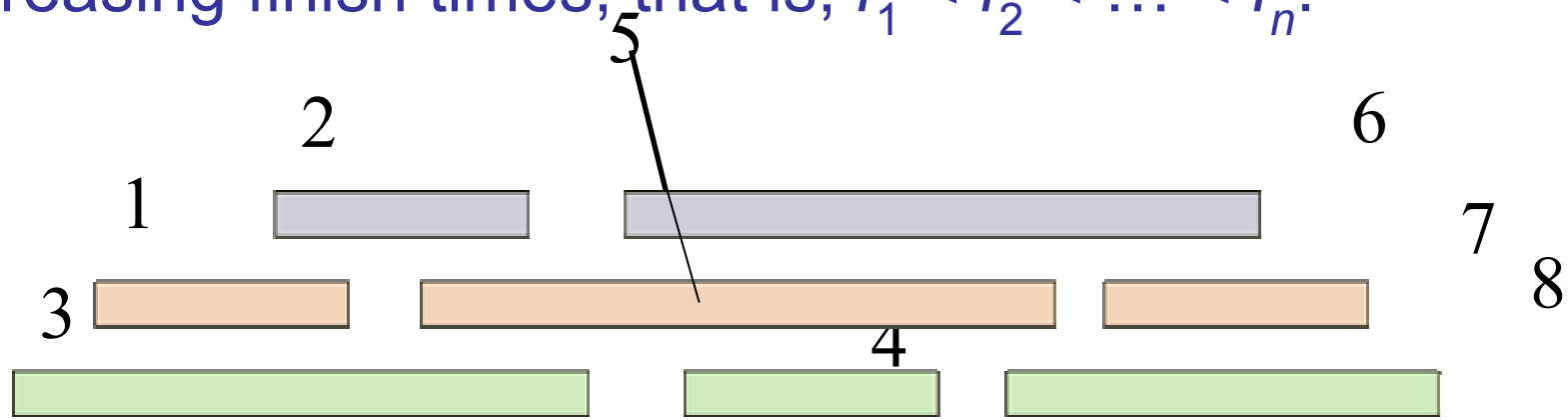
- Objective in activity-selection problem is to select a maximum-size subset of mutually compatible activities.

Steps in Designing Activity Selection Algorithm

- Steps to solve the problem
  - We will formulate the dynamic-programming solution in which
    - Combine optimal solutions to two subproblems to form an optimal solution to original problem
    - Check several choices when determining which subproblems to use in an optimal solution
  - Then needed to make a greedy choice in which
    - One of the subproblems guaranteed empty, so that only one nonempty subproblem remains
  - Then a recursive greedy algorithm is developed and converted to an iterative one

Application : Scheduling Problem

- A classroom can be used for one class at a time.
- There are *n* classes that want to use the classroom.
- Every class has a corresponding time interval $I_j = [s_j, f_j)$ during which the room would be needed for this class.
- Our goal is to choose a maximal number of classes that can be scheduled to use the classroom without two classes ever using the classroom at the same time.
- Assume that the classes are sorted according to increasing finish times; that is, $f_1 < f_2 < \ldots < f_n$.



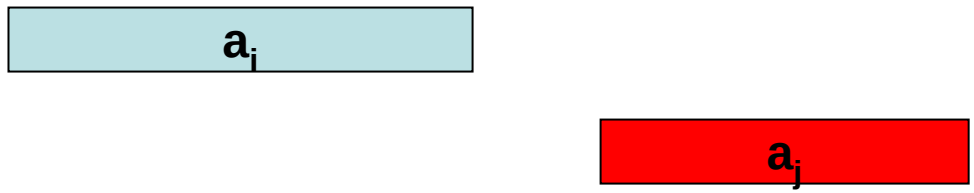
Designing Activity Selection Problem

Compatible Activity

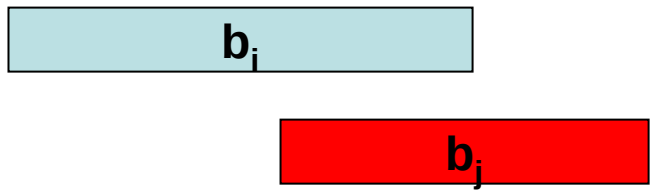
- If a selected activity a_i *is required to* take place during the half-open time interval $[s_i, f_i)$. And activity a_j *is required to* take place during the half-open time interval $[s_j, f_j)$. Then the activities a_i and a_j are compatible if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap i.e

$$s_i \geq f_j \text{ or } s_j \geq f_i$$

Compatible Activities



Not Compatible Activities



Compatible not Maximal

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

A set S of activities sorted in increasing order of finish time

- The subset consisting of mutually compatible activities are
  - $\{a_3, a_9, a_{11}\}$ but it is not a maximal subset,
  - $\{a_1, a_4, a_8, a_{11}\}$ is larger.
  - $\{a_2, a_4, a_9, a_{11}\}$ is another largest subset.

Optimal Substructure of Activity Selection Problem

- The first step is to find optimal substructure and then use it to construct an optimal solution to the problem from optimal solutions to subproblems.
- Let us start by defining sets

$$S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$$

S_{ij} is the subset of activities in S that can start after activity a_i finishes and finish before activity a_j starts.

- The fictitious activities a_0 and a_{n+1} are added and adopt the conventions that $f_0 = 0$ and $s_{n+1} = \infty$. Then $S = S_{0.n+1}$, and the ranges for i and j are given by

$$0 \leq i, j \leq n + 1.$$

- Let us assume that *A_{ij} is a solution to S_{ij}*

Optimal Substructure of Problem

- Let us assume that the activities are sorted in monotonically increasing order of finish times of the activities:

$$f_0 \leq f_1 \leq f_2 \leq \ldots \leq f_n < f_{n+1}$$

- Assuming that we have sorted set of activities, our space of subproblems is
  - to select a maximum-size subset of mutually compatible activities from S_{ij}, for $0 \leq i < j \leq n + 1$,
  - knowing that all other S_{ij} are empty.

Decomposition: Optimal Substructure of Problem

- Suppose that a solution to S_{ij} *is non-empty and* includes some activity a_k, so that

  $$f_i \leq s_k < f_k \leq s_j.$$

- After choosing activity a_k, *it* decomposes S_{ij} into two subproblems, S_{ik} and S_{kj}

  S_{ik} = activities that start after a_i finishes and finish before a_k starts and

  S_{kj} = activities that start after a_k finishes and finish before a_j starts,

- Each of S_{ik} and S_{kj} are subset of the activities in S_{ij}

Solution to $S_{i,j}$: Optimal Substructure of Problem

- Our solution to S_{ij} is the union of the solutions to S_{ik} and S_{kj}, along with the activity a_k.
- Thus, the number of activities in our solution to S_{ij} is the size of our solution to S_{ik}, plus the size of our solution to S_{kj} , plus a_k.

  $\text{Solution}(S_{ij}) = \text{Solution}(S_{ik}) \cup \text{Solution}(S_{kj}) \cup \{a_k\}$

  $A_{ij} = A_{ik} \cup A_{kj} \cup \{a_k\}$
- Suppose we now that an optimal solution A_{ij} *includes* activity a_k, then the solutions A_{ik} and A_{kj} used within this optimal solution must be optimal.

Why $A_{i\,k}$ and $A_{k\,j}$ are Optimal?

Proof Why A_{ik} and A_{kj} are Optimal?

- If we had a solution A'_{ik} to S_{ik} that included more activities than A_{ik}, we could cut out A_{ik} from A_{ij} and paste A'_{ik} in A_{ij}, thus producing another solution A'_{ij} to S_{ij} with more activities than A_{ij}.
- Because we assumed that A_{ij} is an optimal solution, we have derived a contradiction.
- Similarly, if we had a solution A'_{kj} to S_{kj} with more activities than A_{kj}, we could replace A_{kj} by A'_{kj} to produce solution to S_{ij} with more activities than A_{ij}

Further Decomposition to Find $S_{0, n+1}$

- Now we can build a maximum-size subset of mutually compatible activities in S_{ij}
  - by splitting the problem into two subproblems, mutually compatible activities in S_{ik} and S_{kj}
  - finding maximum-size subsets A_{ik} and A_{kj} of theses activities for these subproblems and then
  - forming maximum-size subset A_{ij} as

$$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$$

- Optimal solution to entire problem is: $A_{0,n+1}$.

A Recursive Solution

- Let c[i, j] = number of activities in a maximum-size subset of mutually compatible activities in S_{ij}.
- We have c[i, j] = 0 whenever $S_{ij} = \emptyset$;

  In particular we have, c[i, j] = 0 for i ≥ j.
- Since $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$

  Therefore the recurrence relation for the problem is

$$c[i, j] = c[i, k] + c[k, j] + 1.$$

- Since value of k varies between i + 1, ..., j – 1, and hence there are (j - 1) - (i - 1) + 1 possible values of k, i.e., j - 1 – i - 1 + 1 = j – i - 1

A Recursive Solution

- Since maximum-size subset of S_{ij} must use one of these values for k, we check them all to find the best.
- Thus, our full recursive definition of $c[i, j]$ becomes

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \phi \\ \max\limits_{i<k<j} \{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \neq \phi \end{cases}$$

- Now it is a straightforward exercise to write a tabular, bottom-up, dynamic programming algorithm based on recurrence relation defined in previous slide.