CS 6363.003.21S Lecture 9—February 25, 2021

Main topics are #dynamic_programming with #example/optimal_binary_search_tree and #example/maximum_independent_set_in_trees.

Optimal Binary Search Trees

- Today we're going to look at another example of dynamic programming that has a bit of a divide-and-conquer flavor to it.
- This problem has to do with binary search trees.
- The time to find a node v in a binary search tree is proportional to the number of ancestors of v.
- So normally, we try to create binary search trees that are as balanced as possible. If every node has depth O(log n), then the worst-case time for a search is also O(log n).
- However, suppose we're not worried about the time to perform individual searches but instead the time to perform lots and lots of searches.
- If a node x is a more frequent search target than a node y, we'd prefer to keep the depth of x really low even if it means the depth of y is high.
- In fact, a balanced tree may not be the best choice if we have highly skewed access frequencies and we can't modify the tree. A tree of depth Omega(n) may actually be the best choice if the nodes near the root are accessed far more often than the others.
- So we're going to consider the following problem: We're given a sorted array of *keys* A[1 .. n] and an array of corresponding *access frequencies* f[1 .. n]. Our goal is to build a binary search tree that minimizes the *total search time* assuming each key A[i] is searched for exactly f[i] times.
- And like before, we'll concentrate on computing the optimal value for a solution.
- But what is the value? For a given binary search tree T, let v_1, v_2, ..., v_n be the nodes of T indexed in sorted order so node v_i stores key A[i]. Ignoring constant factors, the total cost of all accesses is
    - Cost(T, f[1 .. n]) := sum_{i=1}^n f[i] * # ancestors of v_i in T
    - Here, the root has itself as its only ancestor.
- We need a good recursive definition of the tree and its cost so we can do dynamic programming.
- Binary search trees are often defined as a root with two subtrees attached. Suppose v_r is the root of tree T.
- If i < r, then all ancestors of v_i except the root are in the left subtree. If i > r, then all ancestors of v_i except the root are in the right subtree. So

$$Cost(T, f[1..n]) = \sum_{i=1}^{n} f[i] + \sum_{i=1}^{r-1} f[i] \cdot \#\text{ancestors of } v_i \text{ in } left(T) + \sum_{i=r+1}^{n} f[i] \cdot \#\text{ancestors of } v_i \text{ in } right(T)$$

- But those are our original definitions of Cost, so

$$Cost(T, f[1..n]) = \sum_{i=1}^{n} f[i] + Cost(left(T), f[1..r-1]) + Cost(right(T), f[r+1..n])$$

- And the base case can be Cost(T, f[1 .. 0]) = 0, because it costs nothing to search for nothing.
- So we now know the total cost of any tree based on its root and subtrees. If we magically knew the best root v_r to use, we could then try to find the best subtrees over keys less than or greater than the root. In other words, optimal binary search trees have optimal substructure, and we're going to use a backtracking approach of guessing the correct root and recursively building the best two subtrees based on our guess.
- Let's formalize this idea with a recurrence. The goal of each subproblem is to build a subtree of the optimal binary search tree.
- Each subtree contains a contiguous subset of keys. So, we just need to compute OptCost(i, k) which we'll define as the optimal cost of a binary search tree over keys A[i .. k].

$$OptCost(i,k) = \begin{cases} 0 & \text{if } i > k \\ \sum_{j=i}^{k} f[i] + \min_{i \le r \le k} \left\{ \begin{array}{l} OptCost(i, r-1) \\ \quad + OptCost(r+1, k) \end{array} \right\} & \text{otherwise} \end{cases}$$

- Our goal is to compute OptCost(1, n).
- Before we do so, though, let's simplify that recurrence slightly by getting rid of that sum.
- Let F(i, k) denote the total frequency count for all keys in the interval A[i .. k], so F(i, k) := sum_{j = i}^k f[j].
- F(i, k) follows its own simple recurrence:
- F(i, k) =
  - 0 if i > k
  - F(i, k - 1) + f[k] otherwise
- We can actually compute all O(n^2) values for different F(i, k) using dynamic programming! Entries only depend on smaller values of k, so we'll compute them by increasing value of i then increasing value of k.

```
InitF(f[1 .. n]):
    for i ← 1 to n
        F[i, i − 1] ← 0
        for k ← i to n
            F[i, k] ← F[i, k − 1] + f[k]
```

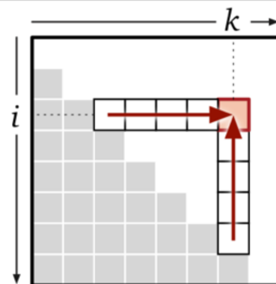
- We'll run this procedure as an initialization subroutine. Now we can rewrite our OptCost recurrence.

$$OptCost(i,k) = \begin{cases} 0 & \text{if } i > k \\ F[i,k] + \min\limits_{i \le r \le k} \left\{ \begin{array}{l} OptCost(i, r-1) \\ \quad + OptCost(r+1, k) \end{array} \right\} & \text{otherwise} \end{cases}$$

- Time to do the rest of the process.

Memoization

- Subproblems: Each recursive subproblem is specified by two integers i and k such that 1 ≤ i ≤ n + 1 and 0 ≤ k ≤ n.
- Memoization: So we'll store all values in an array OptCost[1 .. n + 1, 0 .. n].
- Dependencies: each entry depends upon multiple other entries where the second parameter is smaller or the first parameter is larger. So we only need to look left or below each entry.

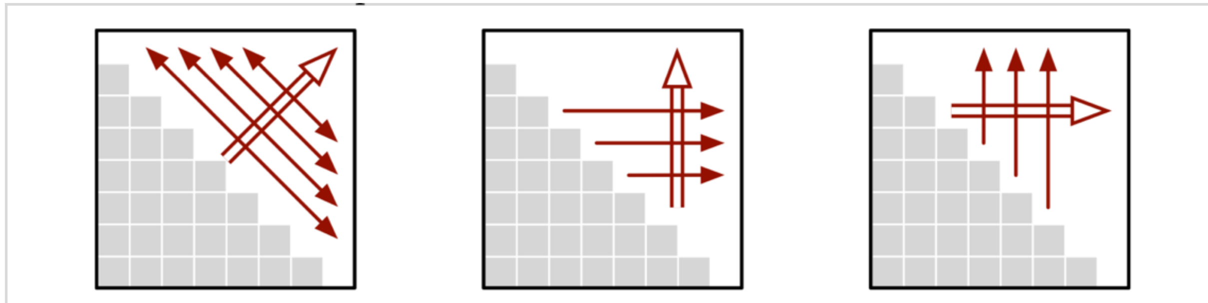


- To simplify filling in each entry OptCost[i, j] given its dependencies are already computed, we'll use a subroutine ComputeOptCost(i, k).

```
ComputeOptCost(i, k):
    OptCost[i, k] ← ∞
    for r ← i to k
        tmp ← OptCost[i, r − 1] + OptCost[r + 1, k]
        if OptCost[i, k] > tmp
            OptCost[i, k] ← tmp
    OptCost[i, k] ← OptCost[i, k] + F[i, k]
```

- Evaluation order: There are a few different orders we can use to fill in this array. We could go diagonal by diagonal from the middle to the top right, but it's a bit nasty to write

correct code for that case.



- Instead, let's go row by row from bottom up, column by column from left to right. **[Use the left pseudocode.]**

```
OptimalBST2(f[1..n]):
  InitF(f[1..n])
  for i ← n+1 downto 1
      OptCost[i, i−1] ← 0
      for j ← i to n
          ComputeOptCost(i, j)
  return OptCost[1, n]
```

```
OptimalBST3(f[1..n]):
  InitF(f[1..n])
  for j ← 0 to n+1
      OptCost[j+1, j] ← 0
      for i ← j downto 1
          ComputeOptCost(i, j)
  return OptCost[1, n]
```

- Time and space: The memoization structure uses **O(n^2) space**. ComputeOptCost(i, j) takes O(n) time at worst and lies in a doubly nested for loop. So **the total time is O(n^3)**.
- Note however, that we didn't even need the code to predict the space or time. We get O(n^2) space from the number of subproblems. Each subproblem takes O(n) time to compute given its dependencies for O(n^3) time total.

Dynamic Programming on Trees

- We just built a tree using dynamic programming. Now, let's run a dynamic programming algorithm on a tree. It's a good example of when a multidimensional array isn't always the best option for a memoization structure.
- An *independent set* of a graph is a subset of vertices with no edges between them. The *maximum independent set* problem asks for the largest independent set in a graph. This problem is really really hard. In fact, it's one of the examples I'll use near the end of the semester for a problem that most people believe doesn't have a polynomial time algorithm *at all*.
- But when you're guaranteed the graph is special in certain ways, you can find the maximum independent set pretty quickly.
- Suppose we're given a rooted tree T with n vertices. Let's compute the size of the maximum independent set in T.
- Again, we'll start with a backtracking approach. Like before, we can decide to do something with the root (which in this case is given to us) and then recurse on the subtrees.
- We'll try to decide if the root belongs to the maximum independent set or not.

- Suppose we decide the root does not belong to the maximum independent set. We can treat each subtree of the root as its own version of the problem, because they do not share edges. We'll just ask for their maximum independent sets.
- And if we do decide to include the root, we can't include the children nodes, but we can include all the grandchildren. We can ask the Recursion Fairy to find maximum independent sets of the grandchildren's subtrees.
- So let MIS(v) denote the size of the maximum independent set in the subtree rooted at v. Let w (downarrow) v mean “w is a child of v”. **[First recursive call should be on w]**

a child of v”.

$$MIS(v) = \max \left\{ \sum_{w \downarrow v} MIS(v),\ 1 + \sum_{w \downarrow v} \sum_{x \downarrow w} MIS(x) \right\}$$

- We need to compute MIS(r) where r is the root of T.
  - Subproblems: Each recursive subproblem takes a node.
  - Memoization: The surprise here is that we don't make a new array. Instead, we'll use the tree itself by storing each MIS(v) in a new field v.MIS.
  - Dependencies: Each entry MIS(v) depends upon the children and grandchildren of v.
  - Evaluation order: And you've likely seen a way to process children and grandchildren before a node. We'll use a standard *post-order* traversal of the tree.
  - Space and time: We're storing one number per vertex, so we use **O(n)** space. Time is more subtle. The time taken to compute MIS(v) for each vertex varies depending on how many children and grandchildren it has. So let's turn the analysis around. Each vertex contributes a constant amount of time to the evaluation of its parent and grandparent's MIS. There’s at most one parent and grandparent per vertex, so the algorithm will run in **O(n)** time.
- And here's the algorithm. It's still recursive, but that's the easiest way to implement post-order tree traversal. **[Maybe do MIS(w) separately from using its value]**

```
MIS(v):
  withoutv ← 0
  for each child w of v
      withoutv ← withoutv + MIS(w)
  withv ← 1
  for each grandchild x of v
      withv ← withv + x.MIS
  v.MIS ← max{withv, withoutv}
  return v.MIS
```

- There's another way we could have solved this problem so that we don't have to worry about grandchildren.
- If we take the root into our independent set, we cannot include the roots of the children subtrees. So we could ask for the maximum independent subsets of the children subtrees that *don't* include their roots.

- Similarly, we could ask for maximum independent sets that *must* include roots.
- So let MISyes(v) denote the size of the maximum independent subset of the subtree rooted at v that *includes* v.
- And MISno(v) is defined the same except we exclude v.
- Now we just want to know max{MISyes(r), MISno(r)}, and we can use the following recurrence.

$$MISyes(v) = 1 + \sum_{w \downarrow v} MISno(w)$$

$$MISno(v) = \sum_{w \downarrow v} \max\{MISyes(w), MISno(w)\}$$

- Nearly all the details are the same as before, but we get a simpler dynamic programming algorithm.

```
MIS(v):
   v.MISno ← 0
   v.MISyes ← 1
   for each child w of v
         v.MISno ← v.MISno + MIS(w)
         v.MISyes ← v.MISyes + w.MISno
   return max{v.MISyes, v.MISno}
```