6.046 Recitation 6 Handout

March 14, 2008

1 Review: Greedy Algorithms

Optimal substructure	An optimal solution contains within it optimal solutions to subproblems.
Greedy choice property	A globally optimal solution can be arrived at by making a locally optimal choice.
Greedy algorithm	Algorithm that makes the choice that looks the best at the moment.

1.1 Design/Correctness

1. Cast the problem as one where you make choices and each choice results in a smaller subproblem to solve.
2. Prove that the greedy choice property holds.
3. Demonstrate that the solution to the subproblem can be combined with the greedy choice to get an optimal solution for the original problem.

2 Knapsack (Shoplifting) Problem

2.1 0-1 Knapsack Problem

Given n items,

- Each item i has value v_i, size s_i
- You have a bag of size S
- You can only choose to take all or nothing of a particular item (hence the 0-1)
- Goal: maximize the value of the items you put in the bag

2.2 Fractional Knapsack Problem

Variation: you can take any *fractions* of the items.

2.3 Optimal Substructure?

0-1 Knapsack

Yes. If we remove item j from an optimal solution, then the remaining items must be optimal for the subproblem where the bag size is $S - s_j$ and the available items don't include item j. Otherwise, we would be able to construct an even better solution than the optimal one.

Fractional Knapsack

Yes by the same argument. If we remove s of item j from an optimal solution, then the remaining items must be optimal for the subproblem where the bag size is $S - s$ and the available items include only $s_j - s$ of item j and all the other items as before. Otherwise, we would be able to construct an even better solution than the optimal one.

2.4 Greedy Algorithm

Greedy thing to do:

Sort by most rewarding items with respect to the ratio of value to size, and take as many of the top most valuable items as will fit.

Algorithm:

1. Compute value per size *density* for each item $d_i = v_i/s_i$.
2. Sort each item by their value density.
3. Take as much as possible of the densest item not already in the bag

Running Time: $O(n \log n)$ for sorting, $O(n)$ for greedy choices

In one of the recitations, it was suggested that the items be entered into a priority queue and retrieved one by one until either the bag is full or all items have been picked. This actually has a better runtime of $O(n + c \log n)$ where c is the number of items that actually get picked in the solution. There is a savings in runtime if $c = o(n)$, but otherwise it doesn't get any worse than just sorting beforehand.

2.5 Does it work?

Example with $n = 3$ items and a bag of size $S = 5$:

Item i	Value v_i	Size s_i	Density d_i
1	$6	1	6
2	$10	2	5
3	$12	3	4

0-1 Knapsack

Greedy algorithm chooses: items 1 and 2 for a total value of 16

Optimal solution: items 2 and 3 for a total value of 22

Greedy algorithm doesn't work!

Fractional Knapsack

Greedy algorithm chooses: items 1, 2, and 2/3 of item 3 for a total value of 24

Optimal solution: the same as the greedy solution

The greedy choice property holds. In particular, we claim that for any instance of the fractional knapsack problem, the optimal solution must include as much of the densest item as will fit. We can prove this by contradiction. Suppose the optimal solution included some amount s of a less dense item i and did not include all that was available of the densest item j, of which there is some amount t that is not included in the bag. Then we could switch $\min(s, t)$ of the less dense item with the densest item and get an even more optimal solution, which is a contradiction. By a similar argument, the optimal solution also cannot have empty space in the bag when there are items still available to put in the bag. (Just imagine instead of a less dense item i, you have an imaginary item k that has no value and is infinitely available.)

3 File Merging

Problem: Need to merge a set $S = \{f_1, \ldots, f_n\}$ of sorted files of different lengths using an optimal merging pattern where merging two files f_i and f_j costs the sum of their lengths $|f_i|+|f_j|$. See Figure 1 for an example.

Total merging cost for a tree T: sum of the internal nodes.

$$C(T) = \sum_{k=1}^{n} |f_k| \cdot depth(f_k)$$

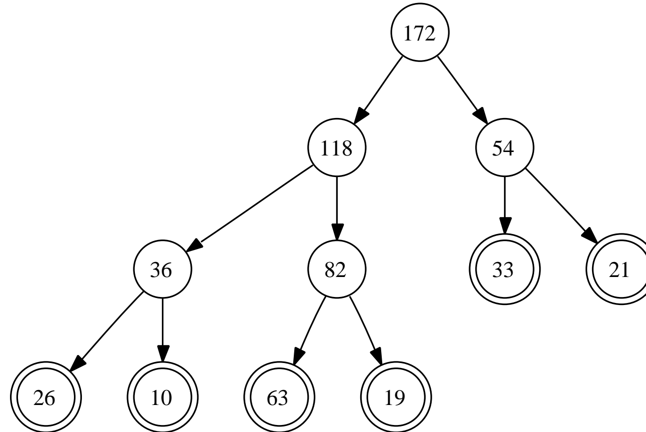

Figure 1: Each leaf is a file, and each internal node is the merge of its children.

3.1 Optimal Substructure

If we look at any subtree T' in the merging tree T, it represents the cost of merging the files that are at its leaves. The cost of the optimal merge tree includes the cost of this subtree $C(T')$ plus the cost of merging the files in T' with the rest of the files in T. The merging tree of T' must be optimal in the optimal solution or else we would be able to construct an even better solution.

3.2 Greedy Choice Property

Claim: There exists an optimal merging pattern such that the two shortest files are merged together. If this is true, we can therefore greedily pick the two shortest files and merge them together and get an optimal solution.

Proof: Suppose the two shortest files f_i and f_j are not merged together in some optimal merging pattern. Find the two deepest nodes that are merged together and swap them with f_i and f_j. The cost of the resulting merge pattern is at most that of the merging pattern we had before since the cost is $C(T) = \sum_{k=1}^{n} |f_k| \cdot depth(f_k)$. We have now constructed an optimal merging pattern that merges the two smallest files.

3.3 Greedy Algorithm

1. Store files in a priority queue, keyed by their lengths.
2. Repeat the following until there is only one file:
   (a) Extract two smallest elements f_i and f_j.
   (b) Merge f_i and f_j and insert this new file in the priority queue.

Running time: $O(n \lg n)$ time using heaps to find best merging pattern plus the optimal cost of merging the files
An observation pointed out in one of the recitations was that if you wanted to gamble, you could naively merge the files together without spending this preprocessing time trying to figure out the best order. For example, if you knew they were all of the same size, you could perform the greedy algorithm without having to actually check which ones have the smallest size.