Problem 1 (Interval Scheduling with Revenue, 35 pts)

During the lecture we considered the activity selection/interval scheduling problem, where the goal is to schedule as many non-overlapping activities/intervals as possible.

Now suppose instead of maximizing the number of activities scheduled, we modify the problem by adding "revenue" value r_i to each activity I_i, for all i. The modified goal is to find a set of non-overlapping activities $S' \subseteq S$ that maximizes the total revenue, defined as

$$\sum_{I_i \in S'} r_i.$$

(a) Does the greedy algorithm presented in class correctly solve this problem? If yes, provide a brief justification. If no, provide a counterexample.

(b) What about the improved DP algorithm (the one with $O(n \log n)$ runtime) presented in class? Can it be easily adapted to solve this problem? If yes, briefly describe what changes need to be made. If no, provide a brief justification.

Solution

(a)

The greedy algorithm presented in class does not correctly solve this problem.

Counterexample: Consider the following intervals:

- I_1: starts at time 0, ends at time 2, with revenue $r_1 = 1$.
- I_2: starts at time 2, ends at time 4, with revenue $r_2 = 1$.
- I_3: starts at time 0, ends at time 4, with revenue $r_3 = 3$.

The greedy algorithm based on the earliest finish time would select I_1 and I_2, yielding a total revenue of $1 + 1 = 2$. However, the optimal solution is to select I_3 alone, which gives a revenue of 3.

(b)

Yes, the improved DP algorithm can be adapted to solve this problem.

For this problem, we need to includethe revenue of each interval into the recurrence. After sorting the intervals by finish time, define:

$$\mathrm{DP}[j] = \max\{r_j + \mathrm{DP}[p(j)],\ \mathrm{DP}[j-1]\},$$

- r_j is the revenue of interval I_j.
- $p(j)$ is the index of the rightmost interval that finishes before I_j starts (i.e., the largest index $i < j$ such that $f_i \leq s_j$).

This rucurrence chooses between including interval I_j (and adding its revenue plus the best revenue from compatible intervals) or excluding I_j (and taking the best revenue from the previous intervals). Since $p(j)$ can be computed in $O(\log n)$ time using binary search, the overall runtime remains $O(n \log n)$.

Problem 2 (Min Interval Scheduling, 35 pts)

In the interval scheduling problem, we are given a set of intervals $S = \{[s_1, f_1], \ldots, [s_n, f_n]\}$ and we want to find the largest subset $S' \subseteq S$ such that no two intervals in S' overlap.

Now suppose that instead of wanting to prevent overlap, we want to prevent having any empty areas. That is, we want that for every point $p \in [a, b]$, there is some interval $I = [s, f] \in S'$ such that $s \leq p \leq f$. We say that such a set S' *covers* $[a, b]$.

For example, on input $S = \{[0, 5], [2, 6], [3, 8], [7, 10], [9, 10]\}$, with target $[a, b] = [0, 10]$ an optimal covering is $S' = \{[0, 5], [3, 8], [7, 10]\}$.

Given a set of n intervals S and a target interval $[a, b]$, we want to construct an efficient greedy algorithm to find the smallest $S' \subseteq S$ such that S' covers $[a, b]$.

(a) Consider the greedy strategy where we always choose the interval I which covers the largest amount of uncovered area of the target interval. Construct a counterexample such that this greedy strategy leads to a suboptimal solution.

(b) Give a greedy strategy that is safe. Prove that your greedy choice combined with an optimal solution to the subproblem remaining (after you make your greedy choice) is an optimal solution to the problem.

(c) Describe a full algorithm that implements your greedy strategy and analyze its running time. For full credit, your algorithm should run in time $O(n \log n)$.

Solution

(a) Counterexample: Let

$$[a, b] = [0, 10]$$

and let

$$S = \{\, I_1 = [0, 2], \quad I_2 = [1, 9], \quad I_3 = [2, 10] \,\}.$$

Initially, the uncovered portion is $[0, 10]$. Suppose algorithm breaks ties arbitrarily and selects $I_2 = [1, 9]$ because it covers 8 units. Then the remaining uncovered portions are $[0, 1]$ and $[9, 10]$. To cover these gaps, the algorithm must select at least one interval for each gap. For instance, it might choose:

$$I_1 \text{ to cover } [0, 1] \quad \text{and} \quad I_3 \text{ to cover } [9, 10].$$

Thus, the algorithm returns the covering $\{I_2, I_1, I_3\}$ with 3 intervals. However, observe that the set

$$\{I_1, I_3\}$$

covers $[0, 10]$ since I_1 covers $[0, 2]$ and I_3 covers $[2, 10]$. This is an optimal covering using only 2 intervals. Thus, the given greedy strategy is suboptimal.

(b)

1. Let x denote the leftmost point in $[a, b]$ that is not yet covered.
2. Among all intervals $I = [s, f]$ with $s \leq x$, choose the interval I^* that has the *largest* finish time f (i.e., that extends farthest to the right).

3. Add I^* to the solution and update $x := f$.

Proof:

Suppose there is an optimal solution S^* that covers $[a, b]$ and let $I' \in S^*$ be the interval that covers the leftmost uncovered point x. By our greedy rule, we chose an interval

$$I^* = [s^*, f^*] \quad \text{with } s^* \leq x \quad \text{and} \quad f^* \geq f',$$

where $I' = [s', f']$. Replace I' with I^* in S^*; the modified solution still covers $[a, b]$ and does not use more intervals. Then, the problem reduces to covering $[f^*, b]$. By applying the same argument recursively, we see that our greedy choice at each step leads to an overall optimal solution.

(c)

Algorithm:

1. **Sort:** Sort the intervals in S by their starting times. (This takes $O(n \log n)$ time.)
2. **Initialize:** Set current $\leftarrow a$ and $S' \leftarrow \emptyset$.
3. **While** current $< b$:
   (a) Among all intervals $I = [s, f]$ with $s \leq$ current, choose the one with the maximum f. (This can be done in a single pass using a pointer or by binary search as the intervals are sorted.)
   (b) If no such interval exists, then there is no covering for $[a, b]$ and the algorithm terminates with failure.
   (c) Add the chosen interval I^* to S' and update current $\leftarrow f^*$.
4. **Return** S'.

Running Time Analysis:

- Sorting the intervals takes $O(n \log n)$.
- The while-loop makes one pass through the intervals. By using a pointer (or maintaining an index) to track the intervals with $s \leq$ current, we can find the interval with the maximum f in $O(1)$ time per interval (amortized). Hence, the loop runs in $O(n)$ time.

Thus, the overall running time of the algorithm is $O(n \log n)$.

Problem 3 (Huffman Encoding Practice, 15 points)

Consider a file that uses the symbols $\{A, B, C, D, E, F, G\}$ with the corresponding frequencies:

f_A	f_B	f_C	f_D	f_E	f_F	f_G
0.08	0.12	0.14	0.15	0.28	0.13	0.1

Find an optimal prefix code based on Huffman's algorithm (using 0 and 1 only). You should give out the following (no justification needed):

(a) The derived Huffman tree.

(b) The mapping from symbols to bit strings. (You can fill in the chart below)

Symbol	Encoding (e.g. 1101)
A	
B	
C	
D	
E	
F	
G	

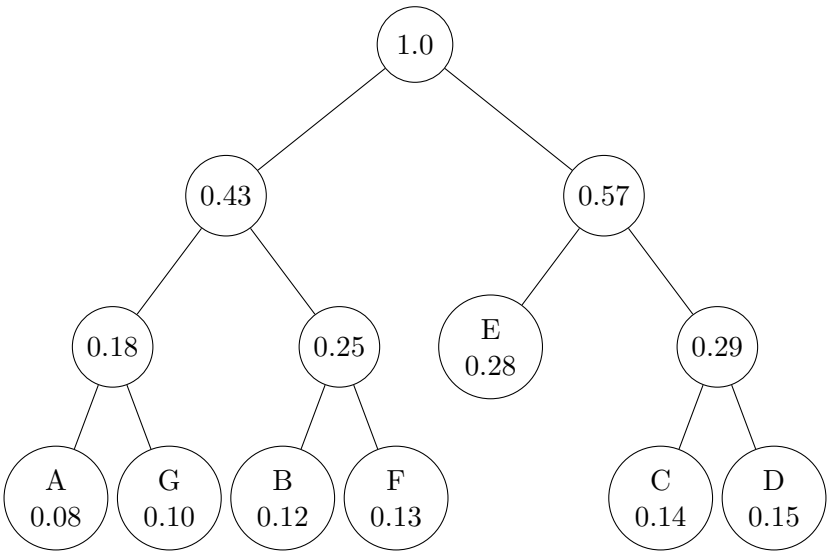
(c) ABL (Average Bits per Letter) of your encoding.

Problem 3: Huffman Encoding Practice

Consider a file that uses the symbols $\{A, B, C, D, E, F, G\}$ with the corresponding frequencies:

f_A	f_B	f_C	f_D	f_E	f_F	f_G
0.08	0.12	0.14	0.15	0.28	0.13	0.10

(a) See the derived Huffman tree on next page.



(b)

Symbol	**Encoding**
A	000
G	001
B	010
F	011
E	10
C	110
D	111

(c)

$$ABL = \sum_{x \in \{A,B,C,D,E,F,G\}} f_x \cdot \ell(x),$$

$$\begin{aligned} ABL &= 0.08 \times 3 + 0.10 \times 3 + 0.12 \times 3 + 0.13 \times 3 \\ &\quad + 0.28 \times 2 + 0.14 \times 3 + 0.15 \times 3 \\ &= 0.24 + 0.30 + 0.36 + 0.39 + 0.56 + 0.42 + 0.45 \\ &= 2.72. \end{aligned}$$

Problem 4 (Huffman Encoding, Short Answers, 15 points)

(a) In general, for a given set of letters and corresponding frequencies, is Huffman encoding always "length-unique"? That is, for any letter, will the length of its encoding always be the same across all valid Huffman trees constructed from the frequencies? Note that different letters may have the same frequency in the general case. If yes, justify in a few sentences. Otherwise, provide a counterexample.

(b) Is it possible that in an *optimal* prefix code, a letter with a lower frequency has a shorter encoding than a letter with a higher frequency? If no, explain why not in a few sentences. Otherwise, provide an explicit example of the letter frequencies and the corresponding prefix code.

Solution

(a)

Huffman encoding is *not* always length-unique.

Counterexample: Consider the set of symbols with frequencies:

$$A : 1, \quad B : 1, \quad C : 2, \quad D : 2.$$

Proceed with Huffman's algorithm as follows:

1. **Initialization:** List the symbols and their frequencies:

$$A(1), \quad B(1), \quad C(2), \quad D(2).$$

2. **First Merge:** Combine the two smallest frequencies. Since A and B both have frequency 1, merge them to form a new node AB with weight $1 + 1 = 2$.

   Now, the available nodes are:

$$AB(2), \quad C(2), \quad D(2).$$

At this point, different valid merge choices lead to different trees:

- **Tree 1:**

  1. Merge $AB(2)$ and $C(2)$ to form node ABC with weight $2 + 2 = 4$.
  2. Merge $ABC(4)$ with $D(2)$ to form the root node with weight $4 + 2 = 6$.

  In this tree, one possible assignment of codeword lengths is:

$$\ell(A) = \ell(B) = 3, \quad \ell(C) = 2, \quad \ell(D) = 1.$$

- **Tree 2:**

  1. Instead of merging AB with C, merge $C(2)$ and $D(2)$ to form node CD with weight $2 + 2 = 4$.
  2. Merge $AB(2)$ with $CD(4)$ to form the root node with weight $2 + 4 = 6$.

In this tree, the codeword lengths become:

$$\ell(A) = \ell(B) = 2, \quad \ell(C) = \ell(D) = 2.$$

Notice that in Tree 1, symbol B has a codeword length of 3, while in Tree 2, symbol B has a codeword length of 2. This demonstrates that the same frequency set can yield valid Huffman trees with different codeword lengths for some letters. Therefore, Huffman encoding is not necessarily length-unique.

(b)

No, in an optimal prefix code it is not possible for a letter with a lower frequency to have a shorter encoding than a letter with a higher frequency.
Suppose by contradiction that a letter x with a lower frequency f_x has a shorter encoding than a letter y with a higher frequency f_y (i.e. $f_x < f_y$ but $\ell(x) < \ell(y)$). Swapping the encodings of x and y would yield a new code with a lower overall weighted length:

$$f_x \cdot \ell(y) + f_y \cdot \ell(x) > f_x \cdot \ell(x) + f_y \cdot \ell(y),$$

which contradicts the optimality of the original code. Hence, in an optimal prefix code, if $f_i > f_j$ then it must be that $\ell(i) \leq \ell(j)$.