Solutions to Homework 2: Greedy Algorithms

Solution 1:

(a) The code tree is shown in Fig. 1.

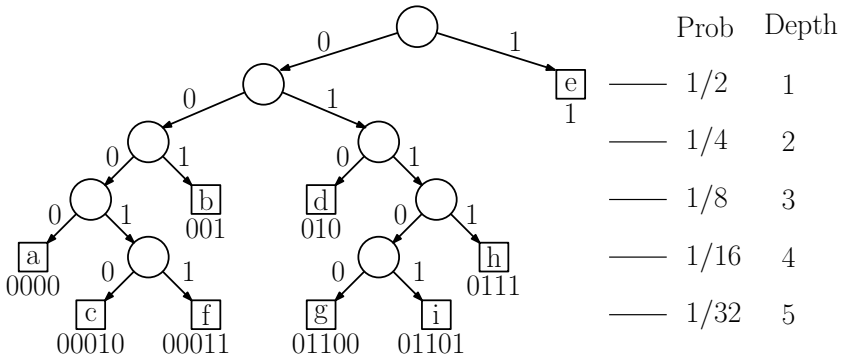

Figure 1: Solution to Problem 1.

The cost is:

$$\begin{aligned} B(T) &= \frac{d(c)+d(f)+d(g)+d(i)}{32} + \frac{d(a)+d(h)}{16} + \frac{d(b)+d(d)}{8} + \frac{d(e)}{2} \\ &= \frac{4\cdot 5}{32} + \frac{2\cdot 4}{16} + \frac{2\cdot 3}{8} + \frac{1}{2} = \frac{5}{8} + \frac{1}{2} + \frac{6}{8} + \frac{1}{2} = 2\frac{3}{8} = 2.375. \end{aligned}$$

(b) By inspection of the tree we have $d_T(x) = \lg(1/p(x))) = -\lg p(x)$.[1]

(c) **Theorem:** Given an alphabet X such that, for each $x \in X$, the associated probability $p(x)$ is a power of 2, the depth of x in the Huffman tree of X is $-\lg p(x)$.

Proof: By induction on $n = |X|$. If $|X| = 1$, that is, X contains a single symbol x, then $p(x) = 1$. The tree consists of a single leaf at depth zero. Thus, we have $d(x) = -\lg p(x) = -\lg 1 = 0$.

For the induction step, let $n \geq 2$, and let us assume the induction hypothesis for any alphabet having $n-1$ symbols whose probabilities are powers of two. Let x and y be the two symbols with the lowest probabilities. Because all the probabilities are powers of 2, we have $p(x) = p(y) = 1/2^k$ for some $k \geq 1$.[2] Huffman's algorithm will replace x and y with a meta-symbol z whose probability is $p(z) = p(x) + p(y) = 1/2^{k-1}$.

[1] By the way, it is noteworthy that the total cost is $B(T) = \sum_{x\in X} p(x)d_T(x) = -\sum_{x\in X} p(x)\lg p(x)$, which is exactly the *entropy* of the probability distribution of the alphabet.

[2] To see why the two smallest probabilities must be equal, suppose that they were not and the lowest was $1/2^k$. Since they are powers of 2, all the other probabilities would be multiples of $1/2^{k-1}$. The sum of all the probabilities must therefore have the form $m/2^{k-1} + 1/2^k = (m+0.5)/2^k$, for some integer m, which cannot equal 1.

After replacing x and y, the result is $n-1$ symbols having probabilities are all still powers of 2. By the induction hypothesis, all the symbols satisfy the depth requirement. Thus, we have $d_T(z) = -\lg p(z) = k-1$. By Huffman's construction, x and y are one level deeper in the tree, and therefore $d_T(x) = k = -\lg p(x)$, as desired. The same is true for y.

Solution 2:

(a) The counterexample involves two files, one slightly longer but with much higher access probability. Let $(s_1, p_1) = (1, 0.1)$ and $(s_2, p_2) = (2, 0.9)$. If we put f_1 before f_2 (size order), the expected access cost is $1 \cdot 0.1 + (2+1) \cdot 0.9 = 2.8$, but if we reverse the order of files the cost is $2 \cdot 0.9 + (1+2) \cdot 0.1 = 2.1$, which is smaller (see Fig. 2(a)).

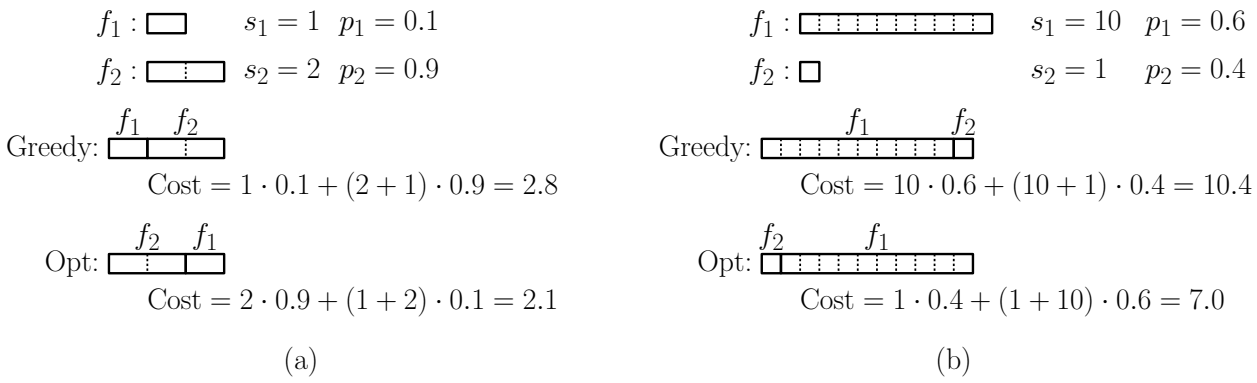

Figure 2: Solution to Problem 2(a) and (b).

(b) The counterexample involves two files, one slightly more likely to be accessed but with much larger size. Let $(s_1, p_1) = (10, 0.6)$ and $(s_2, p_2) = (1, 0.4)$. If we put f_1 before f_2 (decreasing probability order), the expected access cost is $10 \cdot 0.6 + (10+1) \cdot 0.4 = 10.4$, but if we reverse the order of files the cost is $1 \cdot 0.4 + (1+10) \cdot 0.6 = 7.0$, which is smaller (see Fig. 2(b)).

(c) Intuitively, it seems smart to store the most frequently accessed files at the front of the tape, but it also makes sense to store the smallest files at the front of the tape. This suggests that the best way to store the files is in increasing order of s_i/p_i. Let us sort the files according to this statistic and lay them out in this order. (We will make the simplifying assumption that these ratios are distinct for all files.) To simplify notation, let us assume that the files have been renumbered, so that $s_1/p_1 > \cdots > s_n/p_n$. Clearly, this layout can be computed in $O(n \log n)$ time.

Will will prove that this is optimal by contradiction. Suppose that the optimum layout O is different from the greedy layout. If so, there must be two consecutive files of the optimum layout that are not in sorted order. That is, we have $O = \langle \ldots, f_j, f_i, \ldots \rangle$, where $j > i$. Thus, we have $\frac{s_j}{p_j} > \frac{s_i}{p_i}$, or equivalently (because sizes and probabilities are nonnegative), $p_j s_i - p_i s_j < 0$.

Let us consider how the cost changes if these two files are swapped in the layout (see Fig. 3). Call the resulting layout O'. After the swap, file f_j has moved s_i units towards the back of the tape, and so its individual access cost has increased by $p_j s_i$. Similarly, file i has moved

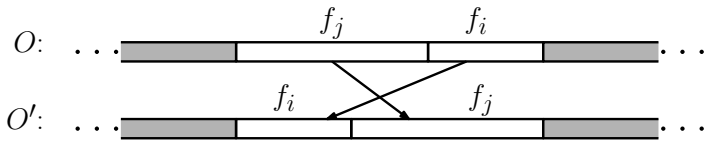

Figure 3: Solution to Problem 2(c), swapping f_j and f_i.

s_j units closer to the front of the tape, so its individual access cost has decreased by $p_i s_j$. All the other files maintain their same placements on the tape, so there are no other changes affecting the total cost. Therefore, the net change in the total access cost is:

$$T(O') - T(O) \;=\; p_j s_i - p_i s_j \;<\; 0.$$

Therefore, $T(O') < T(O)$, which contradicts the optimality of O, and yields the desired contradiction.

Solution 3:

(a) Consider any stabbing set $X = \{x_1, \ldots, x_k\}$. If it satisfies the right-endpoint property, we are done. Otherwise, let x_i be the smallest element that is not the right endpoint of some segment. Let b_j be the smallest right interval that exceeds x_i. Replace x_i with b_j. Because we have increased x_i to a value that is not greater than any of the right endpoints stabbed by x_i, every interval stabbed by x_i is also stabbed by b_j. Thus, the modified set is also a stabbing set. (As shown in Fig. 4(a), b_j need not be an endpoint of one of the intervals that x_i stabs, and we may actually stab new intervals.)

After the replacement, the cardinality of the stabbing set is either the same, or it decreases by one (if b_j was in the original stabbing set). After doing this at most k times, X will be converted into a stabbing set of equal or lower cardinality, and all of its endpoints are all right endpoints of some interval, as desired.

(b) This is just one of many possible solutions for computing the right endpoint of maximum depth (pseudo-code below). We visit all the endpoints (a_i and b_i) in left-to-right order. On seeing a left endpoint we increment a depth counter (`currentDepth`), and on seeing a right endpoint we decrement the counter (Fig. 4(b)). Before decrementing, we check whether the count is the largest so far, and if so we save the value (`maxDepth`) and the location (`maxDepthPoint`).

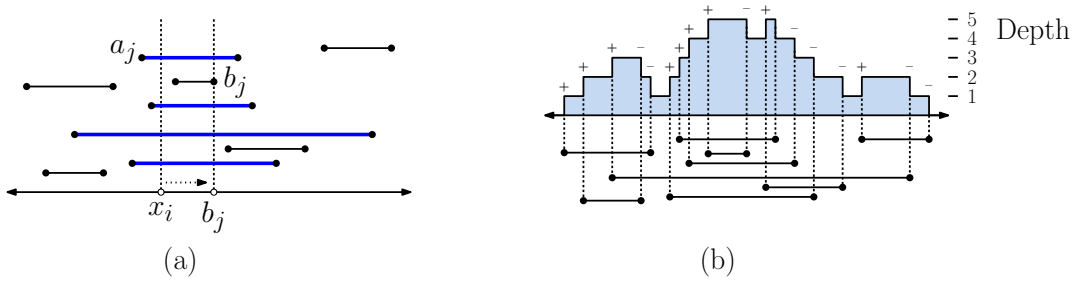

Figure 4: Solution to Problem 3(b).

```
getMaxDepth(a, b) {
  Copy the points of a[1..n] and b[1..n] into a list E of 2*n endpoints
  Sort E
  currentDepth = maxDepth = 0              // current and max depth counts
  foreach (x in E) {
    if (x is a left endpoint) {            // left endpoint?
      currentDepth++                       // increase depth by 1
    }
    else {                                 // right endpoint?
      if (currentDepth > maxDepth) {       // new deepest point?
        maxDepthPoint = x                  // save it
      }
      currentDepth--                       // decrease depth by 1
    }
  }
  return (maxDepth, maxDepthPoint)
}
```

(c) During each iteration, we compute the maximum depth, whose running time is dominated by the $O(n \log n)$ sorting time. Assuming that the intervals are stored in a list, we can enumerate the intervals and delete those that contain the stabbing point in $O(n)$ time. Naively, this would result in a running time $O(kn \log n)$.

(d) Fig. 5(a) presents a counterexample showing that MD is not optimal. The optimum solution involves the right endpoints $\{b_1, b_4\}$. The right endpoints b_1, b_2, and b_4 all have the maximum depth of three. If the algorithm erroneously choses b_2 first, then intervals $[a_1, b_1]$ and $[a_4, b_4]$ are not stabbed. Therefore, it needs to select two additional endpoints, for a total of three.

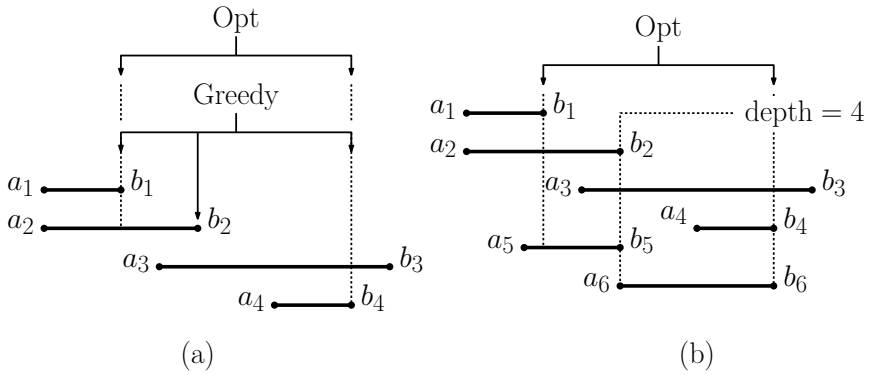

Figure 5: Solution to Problem 3(d).

This example is not very satisfying, since the algorithm might get lucky and select either b_1 or b_4 first. A more robust counterexample is shown in Fig. 5(b). Again, the optimal solution consists of $\{b_1, b_4\}$. In this case, $b_2 = b_5$ has the (unique) maximum depth of four and all other are strictly smaller. After selecting either b_2 or b_5, the algorithm must still select b_1 and b_4, leading to a total cost of three.

(e) We claim that it is possible to reduce the stabbing set problem to an equivalent instance of

the set cover problem. Given any set of n intervals, we construct a set system $\Sigma = (X, S)$ as follows. Let X be the interval indices, that is, the set $\{1, \ldots, n\}$. For each i, $1 \leq i \leq n$, define s_i to be the indices of the intervals that contain the point b_i, and let S be the resulting collection of sets (see Fig. 6(a)).

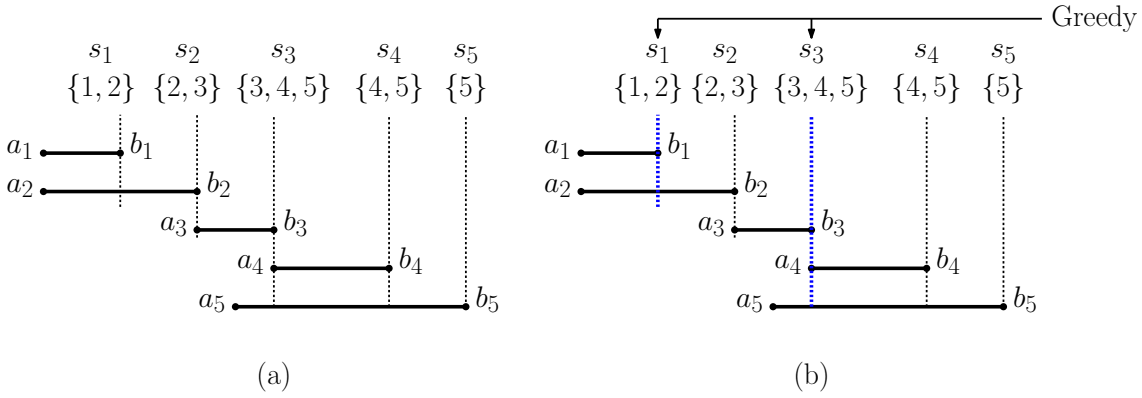

Figure 6: Solution to Problem 3(e).

To establish the equivalence, observe that a set of right endpoints is a stabbing set iff every interval (say, $[a_i, b_i]$) contains at least one point of the stabbing set (say, b_j). This is true iff every interval (say the ith interval) lies within some set of our set system (in this case the set s_j corresponding to b_j). Therefore, a set of k points $\{b_1, \ldots, b_k\}$ is a stabbing set for the intervals iff the associated collection of k sets $\{s_1, \ldots, s_k\}$ form a set cover of X. Thus, an optimal solution to the set cover problem yields an optimal solution to the stabbing set problem.

Let's relate the greedy set-cover heuristic for set cover with the MD heuristic for the stabbing set problem. Observe that at each step, the greedy heuristic for the set cover problem selects the set s_i that covers the maximum number of uncovered elements. This is equivalent to selecting the corresponding right endpoint b_i that stabs the maximum number of unstabbed intervals. MD then removes all the intervals that have been stabbed, and this is equivalent to the greedy set cover heuristic marking the corresponding intervals as covered. Thus, the sets chosen by the greedy set-cover heuristic correspond exactly to the right endpoints chosen by the MD heuristic. (For example, in Fig. 6(b), we would first select $b_3 \equiv s_3$ since it stabs the greatest number of intervals $\{3, 4, 5\}$, and next it would select $b_1 \equiv s_1$ since it covers the most of the remaining intervals $\{1, 2\}$.)

Because the greedy set-cover heuristic returns a collection of sets that is larger than the optimum by a factor of at most $\ln |X| = \ln n$, it follows that the MD heuristic does the same. (Note that because of the special nature of our problem, in particular the fact that the sets are derived from intervals, it may be the greedy algorithm performs better than this. But our reduction only implies this result.)

Solution 4:

(a) The LRE algorithm selects the interval with the earliest finish time and then deletes all overlapping intervals. Rather than explicitly deleting these intervals, it suffices to just skip

over them. Remarkably, we presented an algorithm in class that does exactly this.[3] This was our implementation of the earliest finish first (EFF) heuristic for interval-scheduling problem (see Lecture 7, page 2). We argued in class that this algorithm runs in $O(n \log n)$, as desired.

(b) **Claim:** The LRE heuristic is optimal.

Proof: Let $G = \{g_1, \ldots, g_k\}$ denote the stabbing set generated by the LRE heuristic, and let $O = \{x_1, \ldots, x_k\}$ denote any optimum stabbing set. From Problem 3(a), we may assume that O consists only of right endpoints (and G does as well by definition of LRE). Let us assume that the elements of both G and O are sorted in increasing order.

If $G = O$, then we are done. Otherwise, let x_i denote the first element where G and O differ, and let g_i denote the corresponding element of G. First, observe that because g_i intersects the leftmost right endpoint of an interval that is not stabbed by one of the earlier elements of the stabbing set, we can infer that $x_i < g_i$ (since otherwise the optimum would not stab this interval, contradicting the fact that it is a stabbing set).

We assert that any interval stabbed by x_i is stabbed by either g_i or one of its predecessors. Consider any interval $[a_j, b_j]$ that is stabbed by x_i. If it is not stabbed by one of g_i's predecessors then by definition of greedy $g_i \leq b_j$. (Greedy selects the leftmost right endpoint that is not already stabbed.) Also, since the interval contains x_i, we have $a_j \leq x_i$. In conclusion, we have $a_j \leq x_i < g_i \leq b_j$, implying that g_i stabs this interval, as desired.

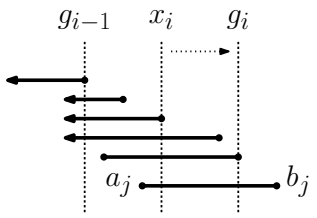

Figure 7: Solution to Problem 4(b).

Now, let us define a new set O' by replacing x_i with g_i in O. The resulting set is still a stabbing set, since every interval stabbed by x_i is also stabbed by g_i or one of its predecessors. Clearly, $|O'| \leq |O|$, implying that O' is also optimal. After repeating this a sufficient number of times, we will convert O into G, without increasing the size of the stabbing set.

Solution to Challenge Problem: See Fig. 5(b) for a solution. (If it bothers you that some of the endpoints are duplicates, it is possible to break these ties without disrupting the example. For example, b_5 could be perturbed slightly to the right and a_6 slightly to the left. Similarly, b_5 could be slid slightly to the right.)

[3] I say that this is *remarkable* because the stabbing set problem is a minimization problem and interval scheduling is a maximization problem. Nonetheless, the same algorithm solves both of them optimally!