Chapter 6

Greedy Algorithms

6.1 Storing the Huffman codeword tree T on a file

We briefly describe a possible way of storing a compact representation of the tree T corresponding to the optimal prefix code on a file. Such a representation is essential and must be stored together with the encoding of a given text file so that a decompression routine can reconstruct it and use it for decoding.

Define the *preorder sequence* $\text{pre}(T)$ of a given full binary tree T as a string containing all nodes of T ordered as follows:

1. If T is a single-node tree, then $\text{pre}(T) = \langle \text{root}(T) \rangle$, where $\text{root}(T)$ denotes the only node of T (which is also its root).

2. If T contains more than one node, let $\text{left}(T)$ and $\text{right}(T)$ denote, respectively, its left and right subtree (note that these subtrees are both nonempty since T is full). Then $\text{pre}(T) = \langle \text{root}(T), \text{pre}(\text{left}(T)), \text{pre}(\text{right}(T)) \rangle$

We will represent the codeword tree T in the file by means of its preorder sequence. Recall that internal nodes do not have to carry any information, while for each leaf f we must store the character $\text{char}(f)$ associated with it. Therefore, in the preorder sequence, we identify each internal node with a single bit, 0, while a leaf f is identified with the pair of symbols $1, \text{char}(f)$. Under this representation, the preorder sequence associated to the optimal tree T of figure 16.4.(b) (CLRS, page 387) is

$$\langle 0, 1, \text{ASCII}(a), 0, 0, 1, \text{ASCII}(c), 1, \text{ASCII}(b), 0, 0, 1, \text{ASCII}(f), 1, \text{ASCII}(e), 1, \text{ASCII}(d) \rangle$$

Note that, since the tree t is full, in the sequence there are always $n-1$ 0's and n pairs

$1, \text{char}(f)$.

When decompressing, it is very simple to reconstruct T from pre(T). Indeed, it is sufficient to create the root node with the statement $r \leftarrow$ new_node() and then call the following routine with parameter r:

```
BUILD_TREE(r)
b ← read()
if (b = 0)
  then
        s ← new_node()
        left[r] ← s
        BUILD_TREE(left[r])
        t ← new_node()
        right[r] ← t
        BUILD_TREE(right[r])
  else { b = 1 }
        c ← read()
        char[r] ← ASCII(c)
        left[r] ← right[r] ← nil
return
```

Exercise 6.1 Let $S = \{[a_1, b_1], [a_2, b_2], \ldots, [a_n, b_n]\}$ be a set of closed intervals on the real line. We say that $C \subseteq S$ is a *covering subset* for S if, for any interval $[a, b] \in S$, there exists an interval $[a', b'] \in C$ such that $[a, b] \subseteq [a', b']$ (that is, $a \geq a'$ and $b \leq b'$).

(a) Write a greedy $O(n \log n)$ algorithm that, on input S, returns a covering subset C^* of minimum size.

(b) Prove the greedy choice property for the above algorithm.

(c) Prove the optimal substructure property for the problem.

Answer:

(a) We first sort the intervals in nondecreasing order of their left endpoint. Ties are broken as follows: if $a_i = a_j$ then $[a_i, b_i]$ precedes $[a_j, b_j]$ in the sorted sequence if $b_i > b_j$. Otherwise, if $b_i < b_j$, $[a_j, b_j]$ precedes $[a_i, b_i]$ (note that it cannot be the case that $b_i = b_j$ or the two intervals would be the same). In other words, ties are broken in favor of the larger interval. Once the intervals are so sorted, we perform a linear scan of the intervals. The greedy choice is to select the first interval. At the i-th iteration, let $[a_k, b_k]$ denote the last interval that was selected. Interval $[a_i, b_i]$ is discarded if it is contained in $[a_k, b_k]$, and is selected otherwise.

We assume that the intervals are given in input stored into an array of records S, with $S[i]$.left $= a_i$ and $S[i]$.right $= b_i$. In the code, procedure SORT(S) performs the required sorting of the intervals, so that, on its termination, $S[i]$ stores the i-th interval of the sorted sequence. Note that a trivial adaptation of MERGE_SORT suffices to yield an $O(n \log n)$ running time for SORT. The algorithm follows:

```
MIN_COVERING(S)
n ← length(S)
SORT(S)
C[1] ← S[1]
j ← 2
{ j stores the index of the first empty location in vector C }
    k ← 1
    { k stores the index of the last selected interval }
    for i ← 2 to n do
        if S[i].right > S[k].right
          then C[j] ← S[i]
          j ← j + 1
          k ← i
          { S[i] is now the last selected interval }
    return C
```

The running time $T_{\text{M\_C}}(n)$ of MIN_COVERING(S) is clearly dominated by the time required by the initial sort, hence $T_{\text{M\_C}}(n) = O(n \log n)$.

(b) MIN_COVERING clearly exhibits the greedy choice property. In fact, *every* covering subset must contain $S[1]$, since by construction $S[1]$ is not contained in any other interval in S.

(c) Let $S = \{[a_1, b_1], [a_2, b_2], \ldots, [a_n, b_n]\}$ and assume that the intervals in S have been renumbered so to comply with the ordering enforced by procedure SORT above. Let $C^\star = \{[a_1, b_1]\} \cup \overline{C}^\star$ be a covering subset of minimum size for S. The optimal substructure property that the algorithm MIN_COVERING exploits is the following: letting k be the maximum index such that $b_j \leq b_1$, for $1 \leq j \leq k$ (note that $a_j \geq a_1$ also holds, because of the ordering), then $\overline{C}^\star$ must be an optimal solution to $\overline{S} = S - \{[a_j, b_j] : 1 \leq j \leq k\}$ (observe that $\overline{S}$ could be empty). To prove this, let us first prove that $\overline{C}^\star$ is indeed a covering subset for $\overline{S}$. If $\overline{S} = \emptyset$, then $\overline{C}^\star$ is clearly empty, since $\{[a_1, b_1]\}$ is a covering subset for S. If $\overline{S}$ is not empty, then $\overline{C}^\star$ must be a covering subset for all intervals in S not contained in $[a_1, b_1]$, since $C^\star = \{[a_1, b_1]\} \cup \overline{C}^\star$ is a covering subset. Therefore, $\overline{C}^\star$ must contain $[a_{k+1}, b_{k+1}]$, which is not covered by any other interval in S. Note, however, that $\overline{S}$ might also contain intervals which are contained in $[a_1, b_1]$. Let $[a_s, b_s]$ be one of such intervals. First note that it must be $s > k+1$, since the interval is in $\overline{S}$. Then, the following chain of inequalities is easily established:

$$a_1 \leq a_{k+1} \leq a_s \leq b_s \leq b_1 < b_{k+1},$$

hence $[a_s, b_s]$ is covered by $[a_{k+1}, b_{k+1}] \in \overline{C}^\star$. To finish the proof it suffices to observe that if $\overline{C}^\star$ were not a covering subset of $\overline{S}$ of minimum size, then $C^\star$ would not be a covering subset of minimum size for S. □

Exercise 6.2 Let $C = \{1, 2, \ldots, n\}$ denote a set of n rectangular frames. Frame i has base $d[i].b$ and height $d[i].h$. We say that Frame i *encapsulates* Frame j if $d[i].b \geq d[j].b$ and $d[i].h \geq d[j].h$. (Note that a frame encapsulates itself). An *encapsulating subset* $C' \subseteq C$ is such that for each $j \in C$ there exists $i \in C'$ such that i encapsulates j.

(a) Design and analyze a greedy algorithm that, on input the vector $\boldsymbol{d}$ of sizes of a set C of n frames, returns a *minimum size* encapsulating subset $C' \subseteq C$ in $O(n \log n)$ time.

(b) Prove the greedy choice property.

(c) Prove the optimal substructure property.

Answer: We first sort the entries of vector $\boldsymbol{d}$ in nonincreasing order of their $d[\cdot].b$ field. Ties are broken as follows: if $d[i].b = d[j].b$ then $d[i]$ precedes $d[j]$ in the sorted sequence if $d[i].h \geq d[j].h$. In other words, ties are broken in favor of the higher frame. Once the frames are so sorted, we perform a linear scan starting from the first frame in the sorted order. The greedy choice is to select the first frame. At the beginning of the i-th iteration, $i \geq 2$, let $j < i$ denote the last frame that was selected. Frame i is discarded if it is encapsulated by frame j, and is selected otherwise.

In the code below, procedure SORT($\boldsymbol{d}$) performs the required sorting of the frames, so that, on its termination, $d[i]$ stores the i-th frame of the sorted sequence. Note that a trivial adaptation of MERGE_SORT suffices to yield an $O(n \log n)$ running time for SORT. The algorithm follows:

```
MIN_ENCAPSULATING_SUBSET(d)
n ← length(d)
SORT(d)
C′[1] ← d[1]
k ← 2
{ k stores the index of the first empty location in vector C′ }
j ← 1
{ j stores the index of the last selected frame }
for i ← 2 to n do
    if d[i].h > d[j].h
      then C′[k] ← d[i]
      k ← k + 1
      j ← i
      { i is now the last selected frame }
return C′
```

The running time $T_{\text{M\_E\_S}}(n)$ of MIN_ENCAPSULATING_SUBSET($\boldsymbol{d}$) is clearly dominated by the time required by the initial sort, hence $T_{\text{M\_E\_S}}(n) = O(n \log n)$.

In the following, we assume that the frames in C have been renumbered so to comply with the ordering enforced by procedure SORT.

(b) Greedy Choice Consider an arbitrary optimal solution $C^\star \subseteq C$. Such subset must contain a frame t which encapsulates Frame 1. Then, either $t = 1$, in which case $C^\star$ contains the greedy choice, or $t > 1$ and (1) $d[t].b \geq d[1].b$ and (2) $d[t].h \geq d[1].h$. In the latter case, due to the sorting, both (1) and (2) must be equalities, hence all frames encapsulated by Frame t are also encapsulated by Frame 1. Therefore $(C^\star - \{t\}) \cup \{1\}$ is an optimal solution containing the greedy choice.

(c) Optimal Substructure Let $C^\star = \{1\} \cup \overline{C}^\star$ be an encapsulating subset of minimum size for C containing the greedy choice. The optimal substructure property featured by the problem is the following: let $j \leq n+1$ be the maximum value such that $d[i].h \leq d[1].h$ for $1 \leq i < j$. Then $\overline{C}^\star$ must be an optimal solution to $\overline{C} = C - \{i : 1 \leq i < j\}$ (observe that $\overline{C}$ could be empty). To prove this, let us first prove that $\overline{C}^\star$ is indeed an encapsulating subset for $\overline{C}$. If $\overline{C} = \emptyset$, then $\overline{C}^\star$ is clearly an encapsulating subset. If $\overline{C}$ is not empty, then $\overline{C}^\star$ must be an encapsulating subset for all frames in C which are not encapsulated by Frame 1, since $C^\star = \{1\} \cup \overline{C}^\star$ is an encapsulating subset for C. Therefore, $\overline{C}^\star$ must contain Frame j, which is not encapsulated by any other frame in C (indeed, $\overline{C}^\star$ could contain a frame j' with the same base and height as j, but then we can substitute j' with

j in $\overline{C}^{\star}$ and proceed with the argument). Note, however, that $\overline{C}$ might also contain frames which are encapsulated by Frame 1. Let s be one of such frames. First note that it must be $s \geq j$, since the frame is in $\overline{C}$. Then, $d[j].b \geq d[s].b$ (by the ordering) and

$$d[j].h > d[1].h \geq d[s].h,$$

hence Frame s is also encapsulated by $j \in \overline{C}^{\star}$.

To finish the proof it suffices to observe that if $\overline{C}^{\star}$ were not an encapsulating subset of $\overline{C}$ of minimum size, then $C^{\star}$ would not be an encapsulating subset of C of minimum size. $\square$

Exercise 6.3 Consider a variant of the *Activity Selection Problem*, where the input set of intervals $S = \{[s_1, f_1), ..., [s_n, f_n)\}$ is sorted by *non decreasing* values of the s_i's, that is, $s_1 \leq s_2 ... \leq s_n$. As in the original problem, we want to determine a maximum set of pairwise disjoint activities.

(a) Design an $O(n)$ algorithm for the above problem.

(b) Prove that the greedy choice and the optimal substructure properties hold.

Exercise 6.4 Let $X = \{x_1, x_2, \ldots, x_N\}$ be a set of points on the real line.

(a) Design and analyze a *greedy* algorithm to determine a minimum cardinality set I of closed intervals of unit length (that is, $i = [a, b] \in I \Rightarrow (b - a) = 1$) such that, for any $x \in X$, there exists an interval $i \in I$ which contains x.

(b) Prove that your algorithm has the greedy choice property.

(c) Prove that the problem enjoys the optimal substructure property.