[CS203] Design and Analysis of Algorithms

Course Instructor: Dr. Dibyendu Roy — Autumn 2023-24

Scribed by: Prakhar Jain (202251099) — Lecture (Weak 9)

Greedy Algorithms: An Introduction

- Optimization algorithms often involve choices at each step.
- Greedy algorithms make locally optimal decisions, aiming for a globally optimal solution.
- Effective for various optimization problems.
- Chapter explores greedy algorithms, comparing them to dynamic programming.

Activity-Selection Problem

- Introduction to a nontrivial problem solved by a greedy algorithm.
- Initial dynamic programming approach, followed by the derivation of a greedy solution.

Fundamentals of Greedy Approach

- Section highlights the basics of greedy algorithms.
- Direct method for proving the correctness of greedy algorithms.

Applications

- Data-compression through Huffman codes as a practical use case.
- Optimality of the ¡furthest-in-future¿ strategy for cache block replacement.

Versatility of Greedy Method

- Greedy method proves powerful in various algorithms.
- Examples include minimum-spanning-tree algorithms, Dijkstra's algorithm, and set-covering heuristic.

Example: Activity-Selection Problem

Our first example is the activity-selection problem, where we aim to schedule competing activities that require exclusive use of a common resource. Consider a set S of proposed activities, each denoted by a_i with start time s_i and finish time f_i.

To solve this problem, we define a dynamic programming approach. Let $c[i, j]$ represent the size of an optimal solution for the set S_{ij}, i.e., activities starting after a_i finishes and finishing before a_j starts.

Activity	Start Time	Finish Time
a_1	1	4
a_2	3	5
a_3	0	6
a_4	5	7
a_5	3	9
a_6	5	9
a_7	6	10
a_8	7	11
a_9	8	12
a_{10}	2	14
a_{11}	12	16

Figure 1: Set of activities

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ c[i,k] + c[k,j] + 1 & \text{otherwise, where } k \text{ is the first activity in } S_{ij} \end{cases}$$

This recurrence relation captures the optimal substructure of the activity-selection problem.

Greedy Method for Activity-Selection

- **Greedy Choice:** Choose the activity that finishes first.
- **Reasoning:** Selecting the activity with the earliest finish time maximizes the availability of the resource for subsequent activities.
- **Top-Down Approach:** The algorithm works top-down, making a choice and then solving a subproblem, instead of a bottom-up technique.

Recursive Greedy Algorithm

The following algorithm, `RECURSIVE-ACTIVITY-SELECTOR`, provides a top-down, recursive approach to solving the activity-selection problem.

Algorithm 1 RECURSIVE-ACTIVITY-SELECTOR

```
function RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
    m ← k + 1
    while m ≤ n and s[m] < f[k] do          ▷ Find the first activity in S_k to finish
        m ← m + 1
    end while
    m ← m + 1
    if m ≤ n then
        return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
    else
        return ∅
    end if
end function
```

Huffman Codes

Huffman codes compress data well: savings of 20% to 90% are typical, depending on the characteristics of the data being compressed. The data arrive as a sequence of characters. Huffman's greedy algorithm uses a table giving how often each character occurs (its frequency) to build up an optimal way of representing each character as a binary string.

Suppose that you have a 100,000-character data file that you wish to store compactly and you know that the 6 distinct characters in the file occur with the frequencies given by Figure 15.4. The character a occurs 45,000 times, the character b occurs 13,000 times, and so on.

You have many options for how to represent such a file of information. Here, we consider the problem of designing a binary character code (or code for short) in which each character is represented by a unique binary string, which we call a codeword. If you use a fixed-length code, you need $\lceil \log_2 n \rceil$ bits to represent $n - 2$ characters. For 6 characters, therefore, you need 3 bits: $a = 000$, $b = 001$, $c = 010$, $d = 011$, $e = 100$, and $f = 101$. This method requires 300,000 bits to encode the entire file. Can you do better?

A variable-length code can do considerably better than a fixed-length code. The idea is simple: give frequent characters short codewords and infrequent characters long codewords. Figure 15.4 shows such a code. Here, the 1-bit string 0 represents a, and the 4-bit string 1100 represents f. This code requires $.45 \times 1 + 13 \times 3 + 12 \times 3 + 16 \times 3 + 9 \times 4 + 5 \times 4 = 224,000$ bits to represent the file, a savings of approximately 25%. In fact, this is an optimal character code for this file, as we shall see.

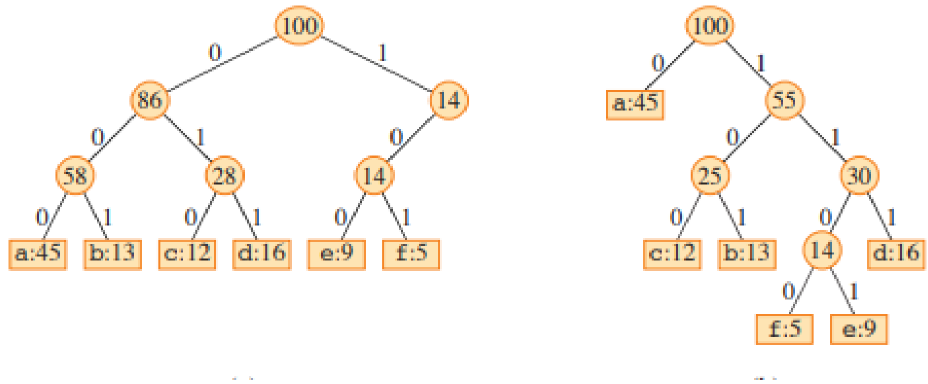

Figure 2: Huffman Tree

Huffman Tree Construction

Codewords

$$
\begin{aligned}
a &: 0 \\
b &: 100 \\
c &: 101 \\
d &: 110 \\
e &: 1110 \\
f &: 1111
\end{aligned}
$$

Huffman Encoding

Original Text: `aaabbbcccdddeeeff`
Huffman Encoded: `00000000011111111110101010101111111100000`

Representation of Prefix-Free Code:

1. A binary tree is used to represent the preûx-free code.
2. Leaves of the tree correspond to characters.
3. Codewords are formed by the path from the root to the leaves:
   - 0 means "go to the left child."
   - 1 means "go to the right child."

Optimal Code and Full Binary Trees:

1. Optimal code is represented by a full binary tree.

2. In a full binary tree:
    - Every non-leaf node has two children.
    - The tree has exactly $|C|$ leaves (characters) and $|C| - 1$ internal nodes.

Number of Bits Required:

1. For each character c in the alphabet C, let $c : \text{freq}$ denote its frequency.
2. Let $dT(c)$ be the depth of c's leaf in the tree.
3. The number of bits required to encode a file is given by:

$$B(T) = \sum_{c \in C} c : \text{freq} \cdot dT(c)$$

Huffman Code Construction Algorithm:

Algorithm 2 Huffman Code Construction

```
procedure HUFFMAN(C)
    n ← |C|
    Q ← C
    for i ← 1 to n − 1 do
         ← Allocate a new node
        x ← EXTRACT-MIN(Q)
        y ← EXTRACT-MIN(Q)
         : left ← x
         : right ← y
         : freq ← x : freq + y : freq
        INSERT(Q, )
    end for
    return EXTRACT-MIN(Q)                ▷ Root of the tree is the only node left
end procedure
```