Advanced Analysis of Algorithms

- Dr. Qaiser Abbas
- Department of Computer Science & IT,
- University of Sargodha, Sargodha, 40100, Pakistan
- qaiser.abbas@uos.edu.pk

Wednesday, January 20, 2021

Greedy Algorithms

- Builds up a solution **piece by piece**, always choosing the next piece that offers the most obvious and immediate benefit.
- An **optimization problem** can be solved using Greedy if the problem has the following property:
  - ***At every step****, we can make a choice that looks* ***best*** *at the moment, and we get the optimal solution of the* ***complete*** *problem*.

Greedy Algorithms

- It generally becomes a better approach if applicable as the Greedy algorithms are **more efficient than other techniques like DP**.
- Greedy algorithms cannot always be applied.
  - Fractional Knapsack (can be solved using **Greedy**)
  - 0-1 Knapsack (**cannot** be solved using Greedy).

Greedy Algorithms

- Following are some Greedy algorithms.
  - **Kruskal's Minimum Spanning Tree (MST):** We create an MST by **picking edges one by one**. The Greedy choice is to pick the smallest weight edge that doesn't cause a cycle in the MST constructed so far.
  - **Prim's Minimum Spanning Tree:** We create an MST by **picking edges one by one**. We maintain **two sets**: set of the vertices already included in MST and the set of the vertices not yet included. The Greedy choice is to pick the smallest weight edge that **connects the two sets**.

Greedy Algorithms

- **Dijkstra's Shortest Path: Similar to Prim's algorithm**. Shortest path tree is built up, edge by edge. We maintain two sets: set A of the vertices already included in the tree and the set B of the vertices not yet included. The Greedy Choice is to pick the edge that connects the two sets and is on the smallest weight path from source to the set B vertices.
- **Huffman Coding:** Huffman Coding is a loss-less compression technique. It assigns variable length bit codes to different characters. The Greedy Choice is to assign **least bit length code to the most frequent character**.

Greedy Algorithms

- Can be used to get an **approximation for hard optimization problems**. Traveling Salesman Problem is a NP hard problem. Picking nearest unvisited city from the current city at every step is a greedy approach but this doesn't always produce best optimal solution, however, can be used to get an approximate optimal solution.

Activity Selection Problem

- Problem statement:
  - *You are given **n activities** with their **start and finish times**. Select the maximum number of activities that can be performed **by a single person**, if a person can only work on a **single activity at a time**.*
- Example:
  - Consider the following 6 activities.
    - Act[]={a0,a1,a2,a3,a4,a5}
    - start[] = {1, 3, 0, 5, 8, 5};
    - finish[] = {2, 4, 6, 7, 9, 9};
    - The maximum set of activities that can be executed by a single person is **{a0, a1, a3, a4}**

Activity Selection Problem

- The greedy choice is to always pick the next activity whose finish time is least among the remaining activities and the start time is more than or equal to the finish time of previously selected activity.
- We can sort the activities according to their finishing time so that we always consider the next activity as minimum finishing time activity.
  - Sort the activities according to their finishing time
  - Select the first activity from the sorted array and print it.
  - Do following for remaining activities in the sorted array.
    - If the start time of this activity is greater than the finish time of previously selected activity then select this activity and print it.

Activity Selection Problem

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

```
GREEDY-ACTIVITY-SELECTOR(s, f)
1  n = s.length
2  A = {a_1}
3  k = 1
4  for m = 2 to n
5      if s[m] ≥ f[k]
6          A = A ∪ {a_m}
7          k = m
8  return A
```

Activity Selection Problem

- GREEDY-ACTIVITY-SELECTOR() schedules a set of n activities in O(n) time, assuming that the activities are sorted initially by their finish times.
- In fact, {a1; a4; a8; a11} is a largest subset of mutually compatible activities; another largest subset is {a2;a4;a9;a11}.

Assignment # 3

16.1-2

Suppose that instead of always selecting the first activity to finish, we instead select the last activity to start that is compatible with all previously selected activities. Describe how this approach is a greedy algorithm, and prove that it yields an optimal solution.

16.1-5

Consider a modification to the activity-selection problem in which each activity a_i has, in addition to a start and finish time, a value v_i. The objective is no longer to maximize the number of activities scheduled, but instead to maximize the total value of the activities scheduled. That is, we wish to choose a set A of compatible activities such that $\sum_{a_k \in A} v_k$ is maximized. Give a polynomial-time algorithm for this problem.

Assignment # 3

16.2-2

Give a dynamic-programming solution to the 0-1 knapsack problem that runs in $O(n\,W)$ time, where n is the number of items and W is the maximum weight of items that the thief can put in his knapsack.

16.2-6 ★

Show how to solve the fractional knapsack problem in $O(n)$ time.

16.2-7

Suppose you are given two sets A and B, each containing n positive integers. You can choose to reorder each set however you like. After reordering, let a_i be the ith element of set A, and let b_i be the ith element of set B. You then receive a payoff of $\prod_{i=1}^{n} {a_i}^{b_i}$. Give an algorithm that will maximize your payoff. Prove that your algorithm maximizes the payoff, and state its running time.

Huffman Codes

- Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters.
- Based on the frequencies of corresponding characters, The most frequent character gets the smallest code, and the least frequent character gets the largest code.

Huffman Codes

- Codes assigned to input characters are Prefix Codes, means the code assigned to one character is not prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream.

Huffman Codes

- **Example:** Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be “cccd” or “ccb” or “acd” or “ab”.
- There are mainly two major parts in Huffman Coding
  - Build a Huffman Tree from input characters.
  - Traverse the Huffman Tree and assign codes to characters.

Huffman Codes

- ***Steps to build Huffman Tree***
  Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.
  1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)
  2. Extract two nodes with the minimum frequency from the min heap.
  3. Create a new internal node with frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
  4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node, and the tree is complete.

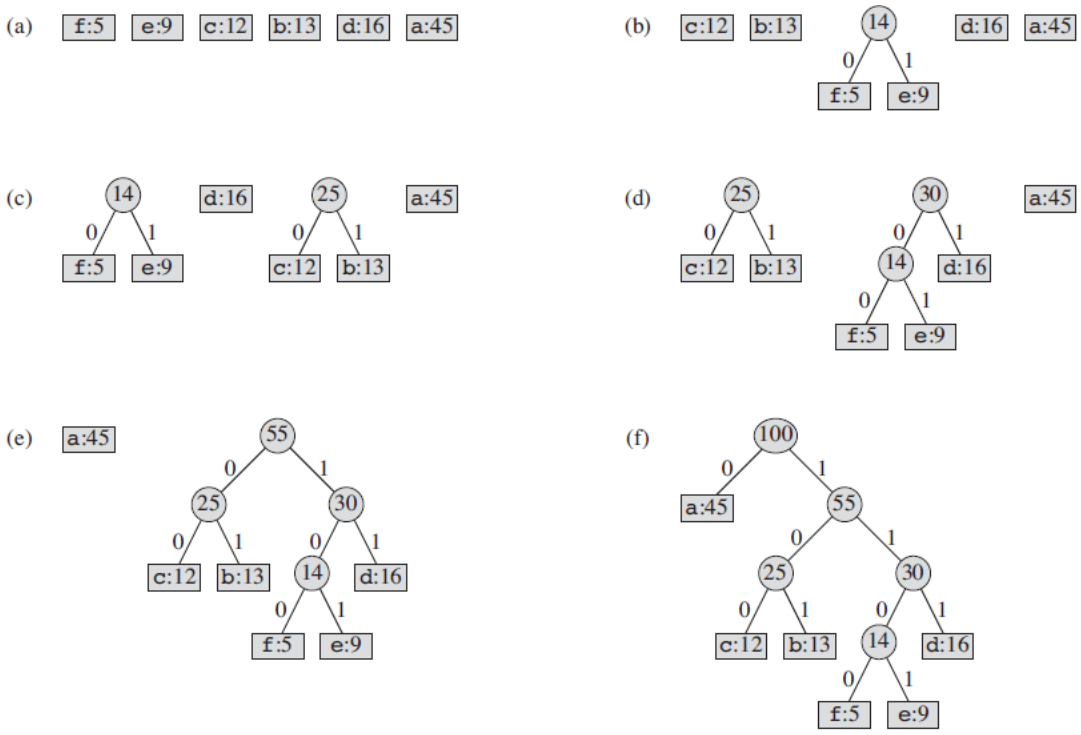
Huffman Codes

- Let us understand the algorithm with an example:

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Figure 16.3 A character-coding problem. A data file of 100,000 characters contains only the characters `a`–`f`, with the frequencies indicated. If we assign each character a 3-bit codeword, we can encode the file in 300,000 bits. Using the variable-length code shown, we can encode the file in only 224,000 bits.

Huffman Codes



Huffman Codes Algorithm

```
HUFFMAN(C)
1  n = |C|
2  Q = C
3  for i = 1 to n − 1
4       allocate a new node z
5       z.left = x = EXTRACT-MIN(Q)
6       z.right = y = EXTRACT-MIN(Q)
7       z.freq = x.freq + y.freq
8       INSERT(Q, z)
9  return EXTRACT-MIN(Q)      // return the root of the tree
```

Huffman Codes

- **Time Complexity:**
  - Q is implemented as binary min-heap(ch6)
  - For a set of C of n characters, the initialization of Q in line 2 can be performed in O(n) time using the BUILD-MIN-HEAP procedure (section 6.3)
  - The for loop in lines 3-8 is executed exactly n-1 times, and since each heap operation requires time O(logn), so the loop contributes O(nlogn) to the running time of Huffman algorithm on a set of n characters.

Assignment # 3

16.3-3

What is an optimal Huffman code for the following set of frequencies, based on the first 8 Fibonacci numbers?

`a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21`

Can you generalize your answer to find the optimal code when the frequencies are the first n Fibonacci numbers?

16.3-7

Generalize Huffman's algorithm to ternary codewords (i.e., codewords using the symbols 0, 1, and 2), and prove that it yields optimal ternary codes.

16.3-9

Show that no compression scheme can expect to compress a file of randomly chosen 8-bit characters by even a single bit. (*Hint:* Compare the number of possible files with the number of possible encoded files.)