Ch19: Greedy Algorithm

305234

Algorithm Analysis and Design

Jiraporn Pooksook
Naresuan University

Greedy Algorithm

- A greedy algorithm always makes the choice that looks best at the moment.
- It does not always yield optimal solutions, but for many problems it does.

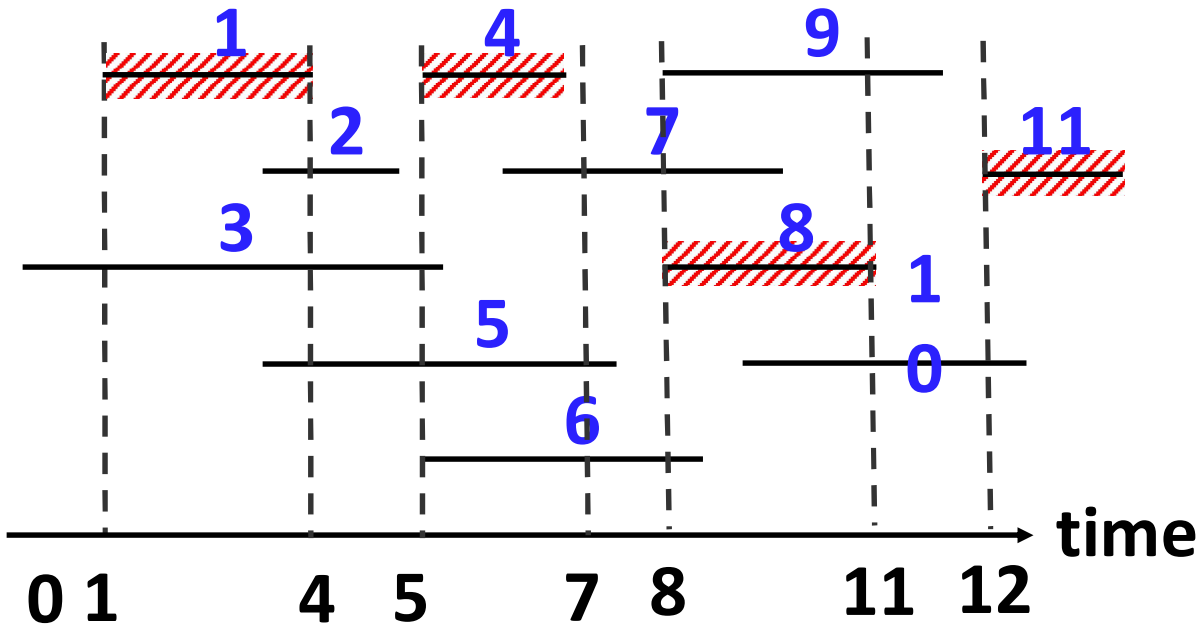
Activity-selection Problem

- It is a problem of how to select a maximum-size subset of mutually compatible activities.
- Activities a_i and a_j are compatible if the interval $[s_i , f_i)$ and $[s_m , f_j)$ do not overlap.

Activity-selection Problem

Input:

i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	9	13
11	12	14



Greedy-Activity-selection(s,f)

```
N = length[s]
A = a_1;
i = 1
For m=2 to n
      do if s[ m ]≥f[ i ]  then
      A = A  ∪ a_m
      i = m
Return A
```

Knapsack Problem

- Consider the sack of capacity 5 Kg.
- We have 3 items
  - Item 1: 10 Kg with value 60$
  - Item 2: 20 Kg with value 100$
  - Item 3: 30 Kg with value 120$

The first i item \capacity	0	5	10	15	20	25	30	35	40	45	50
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	60	60	60	60	60	60	60	60	60
2	0	0	60	60	100	100	160	160	160	160	160
3	0	0	60	60	100	100	160	160	180	180	220

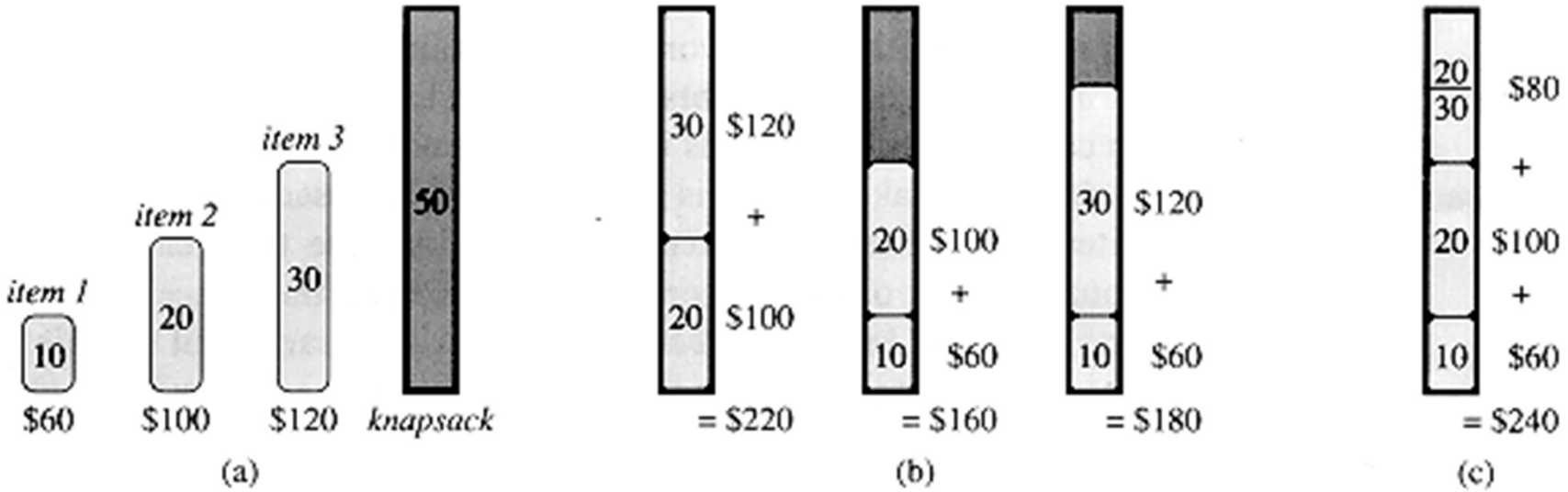
0-1 and Fractional Knapsack Problem

- Constraints of 2 variants of the knapsack problem:
  - *0-1 knapsack problem*: each item must either be taken or left behind.
  - *Fractional knapsack problem*: the thief can take fractions of items.
- The greedy strategy of taking as mush as possible of the item with greatest v_i / w_i only works for the fractional knapsack problem.

Greedy vs Dynamic programming

- 0-1 knapsack problem cannot be solved by greedy algorithm because it cannot deliver the optimal solution.
- Fractional knapsack problem can be solved by greedy strategy.

Greedy Knapsack problem



Huffman codes

- Huffman codes are a widely used and very effective technique for compressing data; savings of 20% to 90% are typical, depending on the characteristics of the file being compressed.
- Huffman's greedy algorithm uses a table of the frequencies of occurrence of each character to build up an optimal way of representing each character as a binary string.

Huffman codes

- There are many ways to represent such a file of information. We consider the problem of designing a ***binary character code.***
- A data file of 100,000 characters contains only the characters `a–f`, with the frequencies indicated. If each character is assigned a 3-bit codeword, the file can be encoded in 300,000 bits. Using the variable-length code shown, the file can be encoded in 224,000 bits.

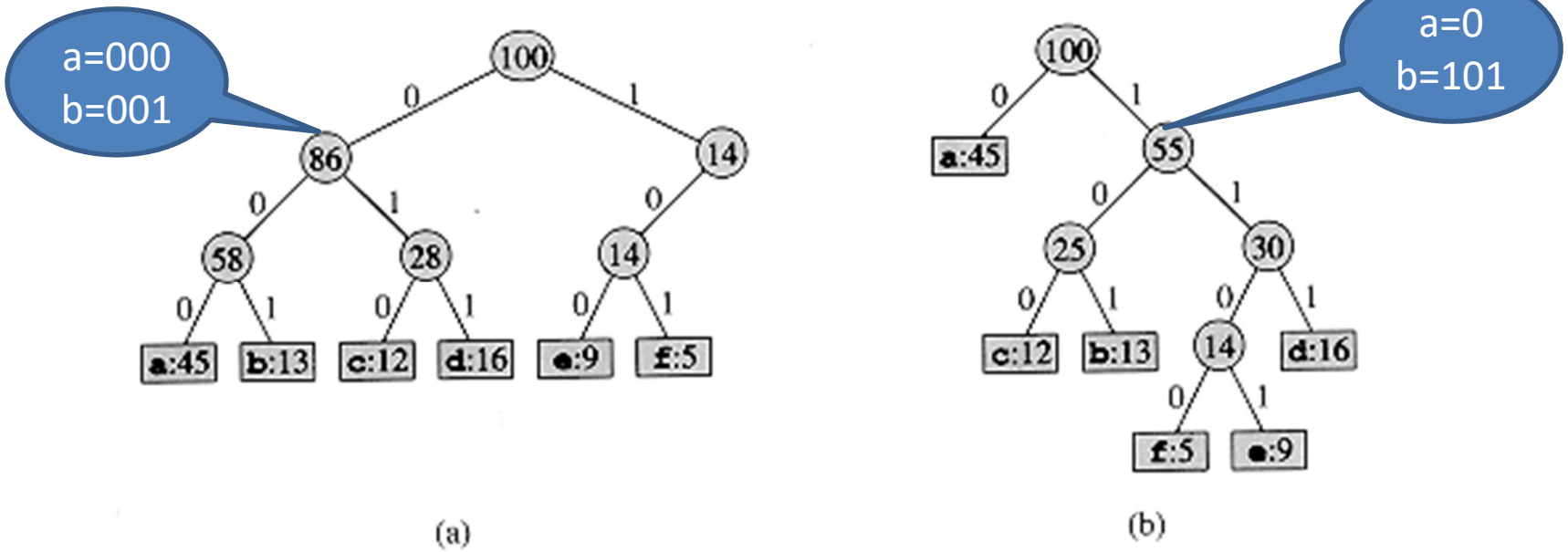
	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Prefix codes and Coding tree

- Prefix codes are codes in which no codeword is also a prefix of some other codeword.
- Decoding is simple. We can identify the initial codeword, translate it back to the original character, remove it from the encoded file, and repeat the decoding process on the remainder of the encoded file.
- The string 001011101 parses uniquely as
  0 0 101 1101, which decodes to aabe.

Prefix codes and Coding tree

- A binary tree are used as a presentation. We interpret the binary codeword for a character as the path from the root to that character, where 0 means "go to the left child" and 1 means "go to the right child."
- Each leaf is labeled with a character and its frequency of occurrence. Each internal node is labeled with the sum of the weights of the leaves in its subtree.



Optimal coding trees

- Given a tree T corresponding to a prefix code, it is a simple matter to compute the number of bits required to encode a file.
- For each character c in the alphabet C, let $f(c)$ denote the frequency of c in the file and let $d_T(c)$ denote the depth of c's leaf in the tree. Note that $d_T(c)$ is also the length of the codeword for character c. The number of bits required to encode a file is thus

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

- **We want to find a coding tree with minimum B(T)**

Constructing a Huffman code

```
Psuedo code: Huffman(C)
n = |C|
Q=C
For i = 1 to n-1
        do allocate a new node z
             left[z] = x = Extract-Min(Q)
             right[z] = y = Extract-Min(Q)
             f[z] = f[x]+f[y]
             Insert(Q,z)
Return Extract-Min(Q)
```

Example: Constructing Huffman code