Greedy Algorithms

Presented By : Mohammad Saeid Heidari

Supervisor : Dr.Farahani

Mail : Heidari.msaeid@gmail.com

Reference: Introduction to Algorithms CLRS

Greedy Algorithms

- Introduction
- An Activity Selection Problem
- A Recursive Greedy Algorithm
- Greedy Algorithm *vs* Dynamic Programming
- Knapsack Problem
- Huffman Codes

Introduction

Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step. For many optimization problems, using dynamic programming to determine the best choices is overkill; simpler, more efficient algorithms will do. A **greedy algorithm** always makes the choice that looks best at the moment. That is, it makes a locally optimall choice in the hope that this choice will lead to a globally optimal solution. This chapter explores optimization problems for which greedy algorithms provide optimal solutions.

An Activity Selection Problem

The problem of scheduling several competing activities that require exclusive use of a common resource

- *n* ***activities*** require *exclusive* use of a common resource
- For example, scheduling the use of a classroom

Set of activities $S=\{a_1, \ldots, a_n\}$.

a_i needs resource during period $[\, s_i , f_i \,)$

- [ ) is a half-open interval
- s_i = *start time* and f_i = *finish time*

Goal

- Select the largest possible set of *non-overlapping* (***mutually compatible***) activities

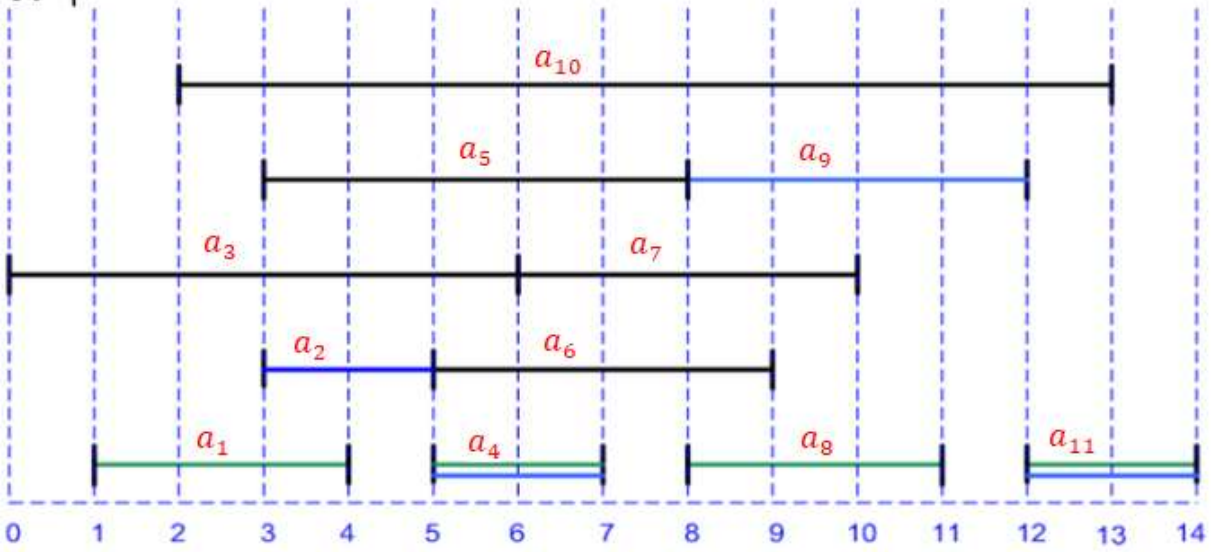
Note: Could have many other objectives

- Schedule room for longest time
- Maximize income rental fees

An Activity Selection Problem

- Here are a set of start and finish times
  - What is the maximum number of activities that can be completed?

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	8	9	10	11	12	13	14

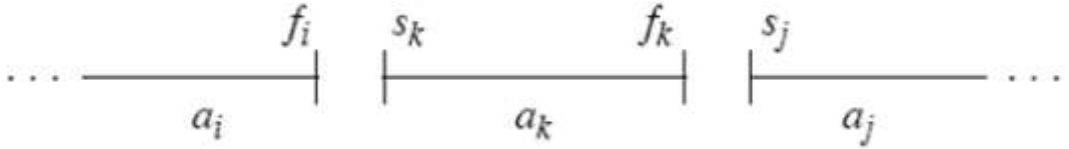


i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

- What is the maximum number of activities that can be completed?
  - $\{a_3, a_9, a_{11}\}$ can be completed
  - But so can $\{a_1, a_4, a_8, a_{11}\}$ which is a larger set
  - But it is not unique, consider $\{a_2, a_4, a_9, a_{11}\}$

An The Optimal Substructure of the Activity Selection Problem

- $S_{ij} = \{a_k \in S : f_i \le s_k < f_k \le s_j\}$
  = activities that start after a_i finishes and finish before a_j starts



- Activities in S_{ij} are compatible with
  - all activities that finish by f_i and
  - all activities that start no earlier than s_j
- To represent the entire problem, add fictitious activities:
  - $a_0 = [-\infty, 0)$
    $a_{n+1} = [\infty, \text{“}\infty+1\text{”})$
  - We don't care about $-\infty$ in a_0 or “$\infty+1$” in a_{n+1}
- Then $S = S_{0, n+1}$
- Range for S_{ij} is $0 \le i, j \le n + 1$

An The Optimal Substructure of the Activity Selection Problem

- Assume that activities are sorted by monotonically increasing finish time
  - $f_0 \le f_1 \le f_2 \le \cdots \le f_n < f_{n+1}$
- Then $i \ge j \Rightarrow S_{ij} = \varnothing$
  - If there exists $a_k \in S_{ij}$: $f_i \le s_k < f_k \le s_j < f_j \Rightarrow f_i < f_j$
  - But $i \ge j \Rightarrow f_i \ge f_j$. Contradiction
- So only need to worry about S_{ij} with $0 \le i < j \le n + 1$
- All other S_{ij} are $\varnothing$

An The Optimal Substructure of the Activity Selection Problem

- If an optimal solution to S_{ij} includes a_k, then the solutions to S_{ik} and S_{kj} used within this solution must be optimal as well
- Let A_{ij} = optimal solution to S_{ij}
- So $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$ assuming:
  - S_{ij} is nonempty
  - we know a_k

An The Optimal Substructure of the Activity Selection Problem

- $c[i, j]$: size of maximum-size subset of mutually compatible activities in S_{ij}
  - $i \geq j \Rightarrow S_{ij} = \varnothing \Rightarrow c[i, j] = 0$
- If $S_{ij} \neq \varnothing$, suppose we know that a k is in the subset
  - $c[i, j] = c[i, k] + 1 + c[k, j]$.
- But, we don't know which k to use, and so
  - $$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset , \\ \max\limits_{\substack{i<k<j \\ a_k \in S_{ij}}} \{c[i, k] + c[k, j] + 1\} & \text{if } S_{ij} \neq \emptyset . \end{cases}$$

A Recursive Activity Selector

- Assumes activities already sorted by monotonically increasing finish time
  - If not, then sort in $O(n \lg n)$ time
  - Return an optimal solution for $S_{i,n+1}$
  - REC-ACTIVITY-SELECTOR(s, f, i, n)

```
m ← i + 1
while m ≤ n and s_m < f_i                ▷ Find first activity in S_{i,n+1}
    do m ← m + 1
if m ≤ n
  then return {a_m} ∪ REC-ACTIVITY-SELECTOR(s, f, m, n)
  else return ∅
```

- *Initial call:* REC-ACTIVITY-SELECTOR*(s, f,* 0*, n)*
- *Time:* $\Theta(n)$ — each activity examined exactly once

A Recursive Greedy Algorithm

GREEDY - ACTIVITY - SELECTOR(s,f)

```
1.  n = s.length
2.  A = {a_1}
3.   i = 1
4.   for m 2 to n
5.         if s[m]>= f[i]
6.                A = A ∪ {a_m}
7.                 i = m
8.   Return A
```

Greedy-Choice Property

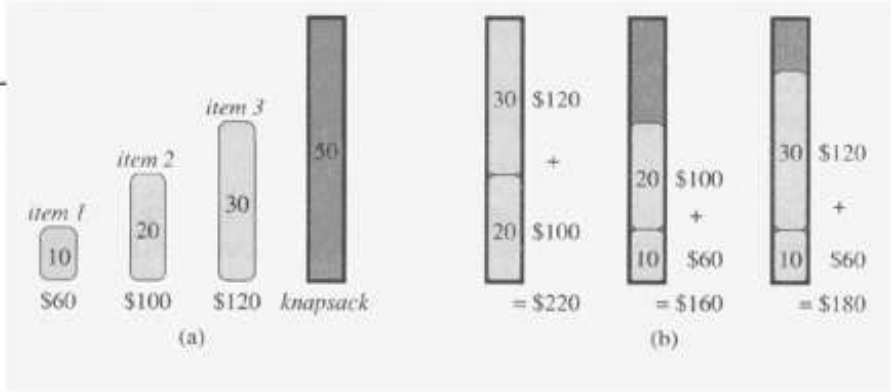
- A globally optimal solution can be arrived at by making a *locally optimal (greedy) choice*
- *Dynamic programming*
  - Make a choice at each step
  - Choice depends on knowing optimal solutions to subproblems
  - Solve subproblems first
  - Bottom-up manner
- *Greedy algorithm*
  - Make a choice at each step
  - Make the choice before solving the subproblems
  - Top-down fashion

Greedy Algorithm *vs* Dynamic Programming

- The knapsack problem
  - Good example of the difference
- 0-1 knapsack problem: not solvable by greedy
  - n items
  - Item i is worth $\$v_i$, weighs w_i pounds
  - Find a most valuable subset of items with total weight ≤ W
  - Have to either take an item or not take it
    - can't take part of it

Knapsack Problem

i	1	2	3
v_i	60	100	120
w_i	10	20	30
v_i / w_i	6	5	4



- $W = 50$
- Greedy solution
  - Take items 1 and 2
  - value = 160, weight = 30
  - Have 20 pounds of capacity left over
- Optimal solution
  - Take items 2 and 3
  - value = 220, weight = 50
  - No leftover capacity

Knapsack Problem

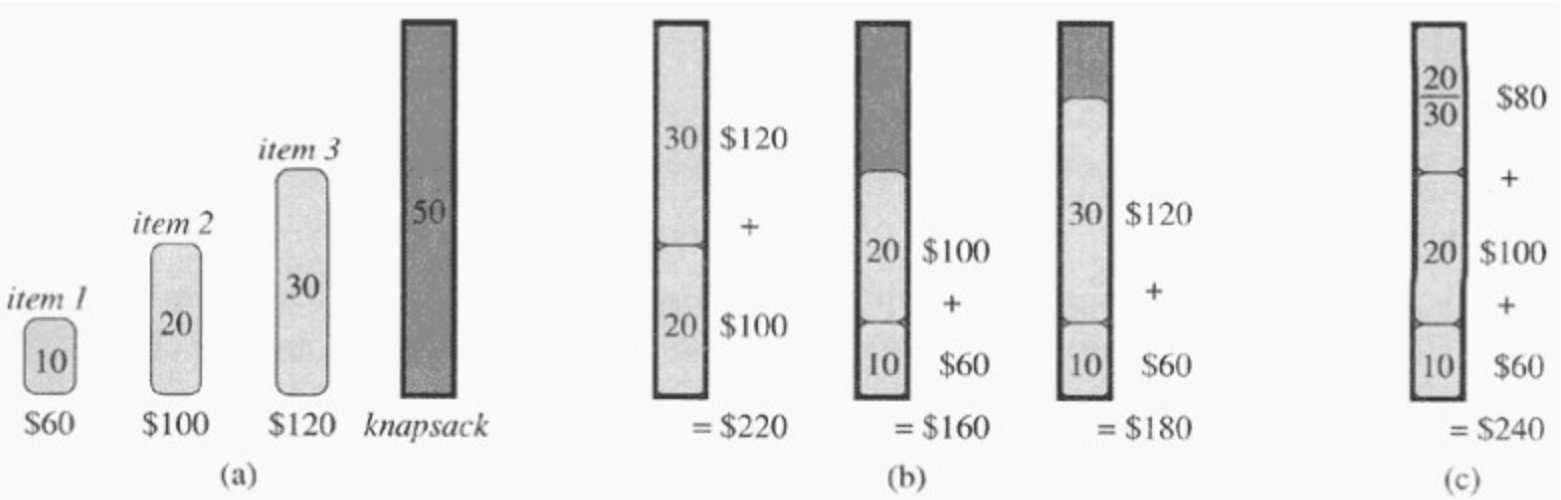
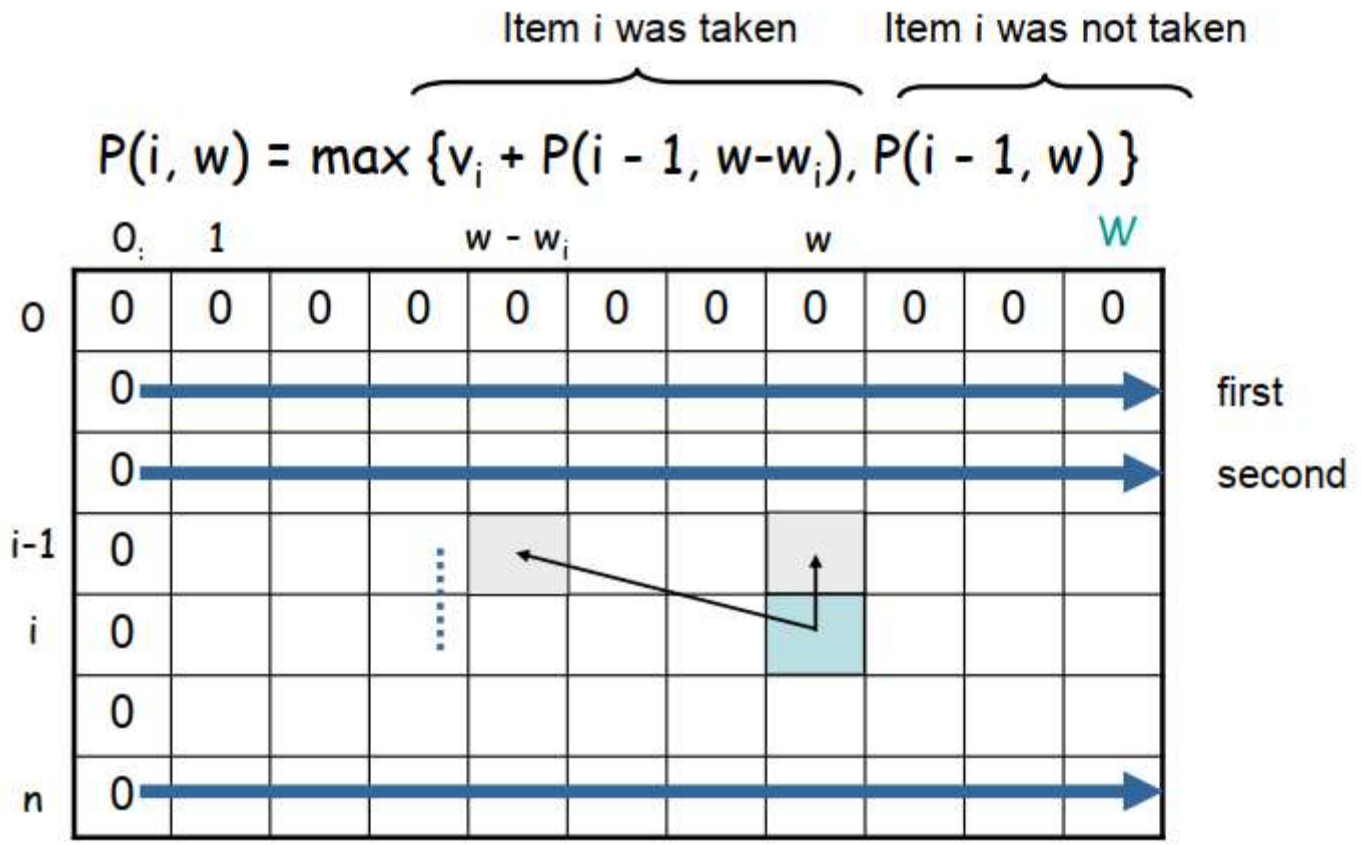


Figure 16.2 The greedy strategy does not work for the 0-1 knapsack problem. **(a)** The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. **(b)** The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound. **(c)** For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.

Knapsack
Dynamic Programming

- P(i, w) – the maximum profit that can be obtained from items 1 to i, if the knapsack has size w
- Case 1: thief takes item i
  - P(i, w) = $v_i + P(i - 1, w-w_i)$
- Case 2: thief does not take item i
  - P(i, w) = $P(i - 1, w)$

Knapsack
Dynamic Programming



Example:

$P(i, w) = \max \{v_i + P(i - 1, w-w_i), P(i - 1, w) \}$

W = 5

Item	Weight	Value
1	2	12
2	1	10
3	3	20
4	2	15

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

P(1, 1) = P(0, 1) = 0

P(1, 2) = max{12+0, 0} = 12

P(1, 3) = max{12+0, 0} = 12

P(1, 4) = max{12+0, 0} = 12

P(1, 5) = max{12+0, 0} = 12

P(2, 1)= max{10+0, 0} = 10

P(2, 2)= max{10+0, 12} = 12

P(2, 3)= max{10+12, 12} = 22

P(2, 4)= max{10+12, 12} = 22

P(2, 5)= max{10+12, 12} = 22

P(3, 1)= P(2,1) = 10

P(3, 2)= P(2,2) = 12

P(3, 3)= max{20+0, 22}=22

P(3, 4)= max{20+10,22}=30

P(3, 5)= max{20+12,22}=32

P(4, 1)= P(3,1) = 10

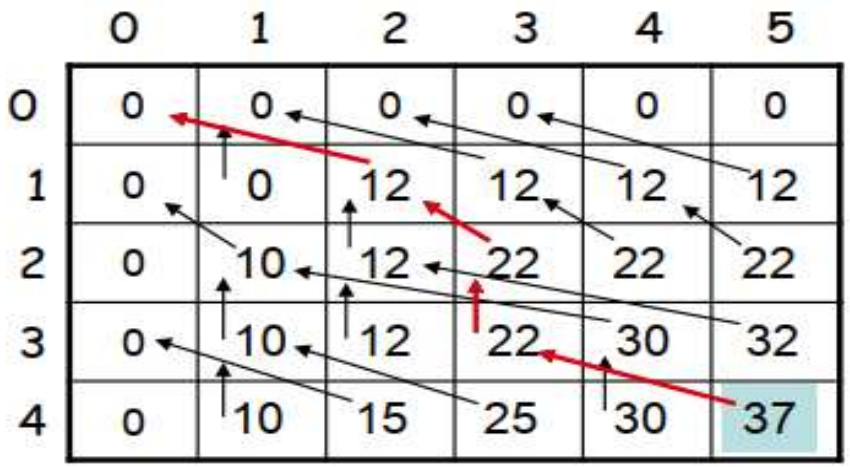
P(4, 2)= max{15+0, 12} = 15

P(4, 3)= max{15+10, 22}=25

P(4, 4)= max{15+12, 30}=30

P(4, 5)= max{15+22, 32}=37

Reconstructing the Optimal Solution



	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

- Item 4
- Item 2
- Item 1

- Start at P(n, W)
- When you go left-up
  - item i has been taken
- When you go straight up
  - item i has not been taken

Knapsack
Dynamic Programming

$$P[k,w] = \begin{cases} P[k-1,w] & \text{if } w_k > w \\ \max\{P[k-1,w-w_k]+v_k, P[k-1,w]\} & \text{else} \end{cases}$$

- Define P[k,w] to be the best selection from S_k with weight at most w
- Since P[k,w] is defined in terms of P[k-1,*], we can use two arrays of instead of a matrix
- Running time: O(nW)
- Not a polynomial-time algorithm since W may be large
- This is a pseudo-polynomial time algorithm

Algorithm ***0-1Knapsack*****(*S*, *W*)**:

Input: set S of n items with benefit v_i and weight w_i; maximum weight W

Output: benefit of best subset of S with weight at most W

let A and B be arrays of length $W + 1$

for $w \leftarrow 0$ **to** W **do**

  $B[w] \leftarrow 0$

for $k \leftarrow 1$ **to** n **do**

  copy array B into array A

  for $w \leftarrow w_k$ **to** W **do**

    if $A[w-w_k] + v_k > A[w]$ **then**

      $B[w] \leftarrow A[w-w_k] + v_k$

return $B[W]$

Huffman codes

- Any binary tree with edges labeled with 0's and 1's yields a prefix-free code of characters assigned to its leaves

- Optimal binary tree minimizing the expected (weighted average) length of a codeword can be constructed as follows:

Huffman Algorithm

- **Initialize *n* one-node trees** with alphabet characters and the tree weights with their frequencies.

- Repeat the following step *n* -1 times: **join two binary trees with smallest weights into one** (as left and right subtrees) and make its weight equal the sum of the weights of the two trees.

- **Mark edges** leading to left and right subtrees with 0's and 1's, respectively.

Example

characters	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101

average bits per character: **2.25**

for fixed-length encoding of 3 bits, the *compression ratio is*:

((3-2.25)/3)*100% = 25%


Thank you