[CSE3081(2반)] 알고리즘 설계와 분석

2020학년도 2학기

강의자료

(2020.11.05 목요일)

서강대학교 공과대학 컴퓨터공학과
임 인 성 교수

[주제 4]
Dynamic Programming

A Variation of the 0-1 Knapsack Problem

- **Problem**

Decision Problem

Given n items of length $l_1, l_2, \cdots, l_n$, is there a subset of these items with total length exactly L?

- **Example**

{ 1, 2, 7, 14, 49, 98, 343, 686, 2409, 2793, 16808, 17206, 117705, 117993 },
L = 138457

→ {1, 2, 7, 98, 343, 686, 2409, 17206, 117705}

- **Dynamic programming approach**
  - Let $P(i, w)$ be **the maximized profit** obtained when choosing items **only from the first** i **items** under the restriction that **the total weight cannot exceed** w.
  - If we let A^* be an optimal subset of $\{1, 2, \ldots, n\}$,

$$1.\ \ n \in A^* : \ P(n, W) = p_n + P(n-1, W - w_n)$$
$$2.\ \ n \notin A^* : \ P(n, W) = P(n-1, W)$$

  - **Optimal substructure**

fill(i, j)

A Divide-and-Conquer Approach

- Let ***fill(i, j)*** return TRUE if and only if there is a subset of the first i items that has total length j.
- When *fill*(i, j) returns TRUE,

  ① If the ith item is used, *fill*(i - 1, j - l_i) must return TRUE.

  ② If the ith item is not used, *fill*(i - 1, j) must return TRUE.

```
int fill(int i, int j) {
  // l[i]: global variable
  if (i == 0) {
    if(j == 0) return TRUE;
    else return FALSE;
  }
  if (fill(i-1, j))
    return TRUE;
  else if (l[i] <= j)
    return fill(i-1, j-l[i]);
}
```

- To solve `fill(int n, int L)`,

$$T(n) \leq \begin{cases} c, & \text{if } n = 0 \\ 2T(n-1) + d, & \text{if } n > 0 \end{cases}$$

$$\Downarrow$$

$$T(n) = \Theta(2^n)$$

A Dynamic Programming Approach

- **The optimal substructure**

$$F(i,j) = \begin{cases} \text{FALSE}, & \text{if } i = 0 \text{ and } j \neq 0 \\ \text{TRUE}, & \text{if } i = 0 \text{ and } j = 0 \\ F(i-1,j) \ \textbf{or} \ ((l_i \leq j) \ \textbf{and} \ F(i-1, j-l_i)), & \text{if } i > 0 \end{cases}$$

- ***O*(*nL*)-time implementation**

```
…
F[0][0] = TRUE;
for (ll = 1; ll <= L; ll++) F[0][ll] =FALSE;
for (i = 1; i <= n; i++) {
  for (ll = 0; ll <= L; ll++) {
    F[i][ll] = F[i-1][ll];
    if (ll - l[i] >= 0)
      F[i][ll] = F[i][ll] || F[i-1][ll-l[i]];
  }
}
return (F[n][L]);
```

- **Example**
  - L = 15, $(l_1, l_2, l_3, l_4, l_5, l_6, l_7)$ = (1, 2, 2, 4, 5, 2, 4)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
1	T	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F
2	T	T	T	T	F	F	F	F	F	F	F	F	F	F	F	F
3	T	T	T	T	T	T	F	F	F	F	F	F	F	F	F	F
4	T	T	T	T	T	T	T	T	T	T	F	F	F	F	F	F
5	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	F
6	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
7	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T

Subset Sum

- **Problem**

  Given a set of positive integers $\{w_1, w_2, \cdots, w_n\}$ of size n and a positive integer W, find a subset A of $\{1, 2, \cdots, n\}$ that maximizes $\sum_{i \in A} w_i$ subject to $\sum_{i \in A} w_i \leq W$.

- **Example**

  $\{w_1, w_2, \cdots, w_9\} = \{20, 30, 14, 70, 40, 50, 15, 25, 80, 60, 10, 95\}$, $W = 99$
  $\longrightarrow \{20, 14, 40, 25\}$

- **Application**
  - There are n jobs, each of which takes w_i time. Now we have a CPU with W free cycles, and want to choose the set of jobs that minimizes the number of idle cycles.

- **Relation to the 0-1 Knapsack problem**

> Given two sets of positive integers $\{w_1, w_2, \cdots, w_n\}$ and $\{p_1, p_2, \cdots, p_n\}$ of size n and a positive integer W, find a subset A of $\{1, 2, \cdots, n\}$ that maximizes $\sum_{i \in A} p_i$ subject to $\sum_{i \in A} w_i \leq W$.

$\updownarrow$

> Given a set of positive integers $\{w_1, w_2, \cdots, w_n\}$ of size n and a positive integer W, find a subset A of $\{1, 2, \cdots, n\}$ that maximizes $\sum_{i \in A} w_i$ subject to $\sum_{i \in A} w_i \leq W$.

- **참고**
  - If it is possible to solve the 0-1 knapsack problem **in polynomial time**, the subset sum problem can be solved **in polynomial time** too.
  - Somebody has already proven that the subset sum problem is very hard. In other words, the subset sum problem is **NP-complete**.

  ⇨ Hence, the 0-1 knapsack problem is also a very hard problem. In other words, the 0-1 knapsack problem is also **NP-complete**.

[주제 5]
Greedy Methods

Algorithm Design Techniques

- Divide-and-Conquer Method
- Dynamic Programming Method
- Greedy Method
- Backtracking Method
- Local Search Method
- Branch-and-Bound Method
- Etc.

The Fractional Knapsack Problem

- **Problem**

  Given two sets of positive integers $\{w_1, w_2, \cdots, w_n\}$ and $\{p_1, p_2, \cdots, p_n\}$ of size n and a positive integer W, find a set of ratios $\{r_1, r_2, \cdots, r_n\}$ $(0 \le r_i \le 1)$ that maximizes $\sum_i r_i \cdot p_i$ subject to $\sum_i r_i \cdot w_i \le W$.

Given two sets of positive integers $\{w_1, w_2, \cdots, w_n\}$ and $\{p_1, p_2, \cdots, p_n\}$ of size n and a positive integer W, find a subset A of $\{1, 2, \cdots, n\}$ that maximizes $\sum_{i \in A} p_i$ subject to $\sum_{i \in A} w_i \le W$.

- A **greedy** approach
  ① Sort the items in nonincreasing order by profits per unit weight $\frac{p_i}{w_i}$.
  ② Choose the items, possibly partially, one by one until the knapsack is full.

- **Example:** $\{w_1, w_2, w_3\} = \{5, 10, 20\}$, $\{p_1, p_2, p_3\} = \{50, 60, 140\}$, $W = 30$

  $\frac{p_1}{w_1} = 10,\ \frac{p_2}{w_2} = 6,\ \frac{p_3}{w_3} = 7$
  - Choose all of the 1st item: (5, 50)
  - Choose all of the 3rd item: (20, 140)
  - Choose half of the 2nd item: (10/2, 60/2)

A **greedy** approach $\sum_{i \in A} p_i$ subject to $\sum_{i \in A} w_i \le W$.

① Sort the items in nonincreasing order by profits per unit weight $\frac{p_i}{w_i}$.

② Choose the items, possibly partially, one by one until the knapsack is full.

- **Implementation 1**
  - Sort the items → $O(n \log n)$
  - Repeat the choice → $O(n)$

$$O(n + n \log n) = O(n \log n)$$

- **Implementation 2**
  - Put the items in a heap → $O(n)$
  - Repeat the choice → $O(k \log n)$
  - ☞ Could be faster if only a small number of items are necessary to fill the knapsack.

$$O(n + k \log n) = ?$$

✓ The greedy method always find an optimal solution to the fractional Knapsack problem! ← **Correctness**

- Does the greedy approach always find an optimal solution to the 0-1 Knapsack problem?

0-1 Knapsack Example 2: n = 6, W = 10

- **0-1 knapsack (dynamic programming)**

	1	2	3	4	5	6
pi	4	5	**12**	**3**	4	3
wi	4	2	**9**	**1**	6	2

Selected items: i = 3, 4
Obtained profit: 15
Time Complexity: O(nW)

- **Fractional knapsack (greedy)**

	4	2	6	3	1	5
pi	**3**	**5**	**3**	**12**	4	4
wi	**1**	**2**	**2**	**9**	4	6
pi/wi	**3.000**	**2.500**	**1.500**	**1.333**	1.000	0.667

Selected items: i = 4, 2, 6, 3(5)
Obtained profit: 17.67
Time Complexity: O(n log n)

- **0-1 knapsack (greedy 1)**

	4	2	6	3	1	5
pi	**3**	**5**	**3**	12	4	4
wi	**1**	**2**	**2**	9	4	6
pi/wi	**3.000**	**2.500**	**1.500**	1.333	1.000	0.667

Selected items: i = 4, 2, 6
Obtained profit: 11
Time Complexity: O(n log n)

- **0-1 knapsack (greedy 2)**

	4	2	6	3	1	5
pi	3	5	3	**12**	4	4
wi	1	2	2	**9**	4	6
pi/wi	3.000	2.500	1.500	**1.333**	1.000	0.667

Selected items: i = 3
Obtained profit: 12
Time Complexity: O(n log n)