0-1 Knapsack Problem

Linnea Caraballo

CS 341 Professor Pinto

May 7, 2022

What is the 0-1 Knapsack Problem?

- A problem that requires one to fill a knapsack that has a specific weight capacity with items that have two conditions, value and weight.
- **Goal:** Fill the knapsack in a way that optimizes the value fo the bag, but does no exceed the weight capacity.
- **Use Case:** Resource allocation problems.

Brute Force

- Let $S = \{item_1, item_2, item_3, ..., item_k\}$, $w_i =$ weight of $item_i$, $v_i =$ value of $item_i$, and $W =$ weight capacity of knapsack.
- Brute force requires us to consider all subsets of S which we will call N.
- We first need to calculate every single N
- We will then discarde those that exceed W, the weight limit of the knapsack.
- This leaves us to then find the subset with the largest value.

> We will need to calculate 2^n subsets, so $O(2^n)$ which is undesirable.

Greedy Approach 1

- Pick the highest value first and then continue in decreasing order of item values.

Table: Example from CodesDope [2]

Item Number $item_i$	**Weight** w_i	**Value** p_i
1	12	100
2	32	200
3	30	50
4	5	60
5	34	150

$$W = 50$$

- Using this method gets us $N_1 = \{item_2, item_1\}$ in that order for a total value of 300 and weight of 44 [**Highest value: 360**].

Greedy Approach 2

- Pick the lowest weight item first and then go in ascending order of weight.

Table: Example from CodesDope [2]

Item Number $item_i$	**Weight** w_i	**Value** p_i
1	12	100
2	32	200
3	30	50
4	5	60
5	34	150

$$W = 50$$

- Using this method gets us $N_2 = \{item_4, item_3, item_1\}$, which gets us a total value of 210 and a total weight of 47 [**Highest value: 360**].

Greedy Approach 3

- First pick the items with the largest value per weight.

Table: Example from CodesDope [2]

Item Number $item_i$	**Weight** w_i	**Value** p_i
1	5	50
2	10	60
3	20	140

$$W = 30$$

- Using the same method we get a value of 190 [**Highest Value:** 200].

Dynamic Programming: Bottom-Up

- **Goal:** Put values into a table where the rows represent the weight limit and the columns represent the items.
- This means that if we pick a cell (i, j) it would contain the optimal value for the first i items for a weight limit of j.
- **Order:** $O(n * W)$

Bottom-Up Example Setup

Table: Example from CodesDope [2]

Item Number $item_i$	Weight w_i	Value p_i
1	3	8
2	2	3
3	4	9
4	1	6

$$W = 5$$

Bottom-Up Example Cont.

Table: Example from CodesDope [2]

$w \rightarrow$ $item_i \downarrow$	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

Observe that for a maximum weight of 0 that we cannot take any items, so the whole column $j = 0$ would have a total value of 0. Similarly note how if we take 0 items that the total value would be 0, so the whole row $i = 0$ is also 0.

Bottom-Up Example Cont.

Table: Example from CodesDope [2]

$w \rightarrow$ $item_i \downarrow$	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	0	8	8	8
2	0					
3	0					
4	0					

Recall: $w_1 = 3$ with a value $v_1 = 8$. This means that for all $w < 3$ that we cannot take an item, so these cells have a 0. For any $w \geq 3$ we can put item 1 into our knapsack, so these cell values will be 8

Bottom-Up Example Cont.

Table: Example from CodesDope [2]

$W \rightarrow$ $item_i \downarrow$	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	0	8	8	8
2	0	0	3	8	8	11
3	0	0	3	8	8	11
4	0	6	6	9	14	15

Observe: $F(4, 1) = 6$ because $w_4 = 1$ and $v_4 = 6$, so we can carry that item with a $w = 1$. Also observe that $F(4, 5) = 15$ becuase we can take $item_3$ and $item_4$, such that $w_3 + w_4 = 5$ and $v_3 + v_4 = 9 + 6$.

Refinement of Dynamic Programming Approach (Recursion)

- The order using bottom-up can exceed $O(2^n)$ is for example given $W = 2!$ leading to $O(n * n!)$.
- We can instead simply consider $F[i][w]$ and $F[i-1][W]$ and $F[i-1][W - w_i]$ where n is the number of items and i is the item we are talking about/the row we are in.
- We start from n and end when either $n = 1$ or $w \leq 0$. Leaving us with the equation.

$$F[i][W] = \begin{cases} maximum(P[i-1][W],\ v_i + P[i-1][w - w_i]) & \text{if } w_i \leq W \\ P[n-1][W] & \text{if } w_i > W \end{cases}$$

- **Order:** $O(minimize((n * W), (2^n)))$

Refinement of Dynamic Programming Example Setup

Table: Example from *Foundations of Algorithms* [4]

Item Number $item_i$	**Weight** w_i	**Value** p_i
1	5	50
2	10	60
3	20	140

$$W = 30$$

Refinement of Dynamic Programming Example Cont.

- **Determine the entries need:**
  - Row 3 we need $F[3][30]$
  - Row 2 we compute
    $F[3][30]$, thus we need $F[2][30]$ and $F[3-1][30-w_3] = F[2][10]$
  - Row 2 we compute $F[2][30]$, so we need $F[1][30]$ and $F[1][20]$
  - Row 1 we compute $F[1, 10]$ and $F[1][0]$
  - Stop because $n = 1$ as well as $w \leq 0$

Refinement of Dynamic Programming Example Cont.

Step 1:

$$F[1][w] = \begin{cases} maximum(P[0][w],\ 50 + P[0][w-5]) & \text{if } w_1 = 5 \leq w \\ P[0][5] & \text{if } w_1 = 5 > w \end{cases}$$

$$= \begin{cases} 50 & \text{if } 5 \leq w \\ 0 & \text{if } 5 > w \end{cases}$$

Step 2:

$$F[2][10] = \begin{cases} maximum(P[1][10],\ 60 + P[1][0]) & \text{if } w_2 = 10 \leq 10 \\ P[1][10] & \text{if } w_2 = 10 > 10 \end{cases}$$

$$= 60$$

Refinement of Dynamic Programming Cont. (Steps)

Step 3:

$$F[2][30] = \begin{cases} maximum(P[1][30],\ 60 + P[1][20]) & \text{if } w_2 = 10 \leq 30 \\ P[1][30] & \text{if } w_2 = 10 > 30 \end{cases}$$
$$= 60 + 50 = 110$$

Step 4:

$$F[3][30] = \begin{cases} maximum(P[2][30],\ 140 + P[2][10]) & \text{if } w_3 = 20 \leq 30 \\ P[2][30] & \text{if } w_3 = 20 > 30 \end{cases}$$
$$= 140 + 60 = 200$$

Resources Used

- The Knapsack Problem — Data Structures and Algorithms.
  https://stevenschmatz.gitbooks.io/data-structures-and-algorithms/content/281/lecture_20.html.
- Knapsack Programming Using Dynamic Programming and its Analysis.
  https://www.codesdope.com/course/algorithms-knapsack-problem/.
- Matthew Martin.
  0/1 Knapsack Problem Fix using Dynamic Programming Example.
  https://www.guru99.com/knapsack-problem-dynamic-programming.html, March 2020.
- Richard E. Neopolitan.
  Foundations of Algorithms.
  Jones & Bartlett Learning, the fifth edition.