אור אטיאס 207953308 פתרון ש"ב מס' 7 + 8 בקורס בינה מלאכותית

לתאר בעיית התרמיל במספרים שלמים

לכתוב אלגוריתם תכנון דינאמי של בעיית התרמיל

Each item has a value (bi) and a weight (wi)

• **Objective:** maximize value

• **Constraint:** knapsack has a weight limitation (W)

Given: Knapsack has weight limit W A set S of n items, items labeled 1, 2, …, n (arbitrarily), each item i has - bi - a positive benefit value - wi - a positive weight (assume all weights are integers)

Goal: Choose items with maximum total benefit but with weight at most W.

Let T denote the set of items we take –

Objective: maximize – $i \in T \ \sum bi$

Constraint: $i \in T \ \sum wi \leq W$

Problem, in other words, is to find

max $i=n \sum xibi$ subject to $i \ \sum xiwi \leq W$; where $xi \epsilon \{0,1\}$

- bi - a benefit value

- wi - a weight

Recursive Formula:

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

• The best subset of Sk that has the total weight ≤ w, either contains item k or not.

• First case: wk>w. Item k can’t be part of the solution, since if it was, the total weight would be > w, which is unacceptable.

• Second case: wk ≤ w. Then the item k can be in the solution, and we choose the case with greater value.

אור אטיאס 207953308 פתרון ש"ב מס' 7 + 8 בקורס בינה מלאכותית

The Algorithm of 0/1 Knapsack - Calculation of complexity

```
for w = 0 to W                                    O(W)
    V[0,w] = 0
for i = 1 to n
    V[i,0] = 0                                    Repeat n times
for i = 1 to n
    for w = 1 to W
        if w_i <= w // item i can be part of the solution      O(W)
            if b_i + V[i-1,w-w_i] > V[i-1,w]
                V[i,w] = b_i + V[i-1,w- w_i]
            else
                V[i,w] = V[i-1,w]
        else V[i,w] = V[i-1,w]  // w_i > w
```

What is the running time of this algorithm? $O(n*W)$

Not a polynomial-time algorithm if W is large

This is a pseudo-polynomial time algorithm

Remember: the brute-force algorithm takes $O(2^n)$

אור אטיאס 207953308 פתרון ש"ב מס' 7 + 8 בקורס בינה מלאכותית

Programming By Python

https://github.com/or1610/Knapsack-Implementation-Python

#Example usage (WEIGHT, VALUE OR PROFIT)

items = [(30, 20), (25, 18), (20, 17), (18, 15), (17, 15), (11, 10), (5, 5), (2, 3), (1, 1), (1, 1)]

	dynamic programming	δ-approximation algorithm with δ=2	greedy algorithm
capacity = 55	50	50.3	38
capacity = 90	75	77.8	70
capacity = 60	52	54.5	43
capacity = 65	57	58.6	48

Here is an example of the knapsack problem with a capacity of 70 and a list of items:

Greedy Algorithm Solution:

Items:

1. Book (weight: 10, value: 60)
2. Food (weight: 20, value: 100)
3. Water bottle (weight: 30, value: 120)
4. Sleeping bag (weight: 40, value: 200)
5. Tent (weight: 50, value: 300)

To solve this problem using the greedy algorithm, we first sort the items in descending order based on their value-to-weight ratio:

Items:

1. Tent (weight: 50, value: 300, ratio: 6)
2. Sleeping bag (weight: 40, value: 200, ratio: 5)
3. Water bottle (weight: 30, value: 120, ratio: 4)
4. Food (weight: 20, value: 100, ratio: 5)
5. Book (weight: 10, value: 60, ratio: 6)

Starting with the most valuable item (the tent), we try to fit it into the knapsack. Since its weight (50) is less than the capacity of the knapsack (70), we add it to the knapsack and remove its weight from the remaining capacity, leaving us with a remaining capacity of 20.

Next, we try the sleeping bag. Since its weight (40) is less than the remaining capacity (20), we add it to the knapsack and remove its weight from the remaining capacity, leaving us with a remaining capacity of 0.

At this point, the knapsack is full and we have used all the items, so our solution is to include the tent and the sleeping bag, with a total value of 500 and a total weight of 90.

Note that this solution may not be the optimal one, as we did not consider the other items when making our decisions. For example, if we had chosen the book and the food instead of the tent and the sleeping bag, we would have a lower weight but the same total value. However, the greedy algorithm only considers the most valuable item at each step and does not consider the overall consequences of its choices.

Dynamic Programming Solution:

To solve the knapsack problem using dynamic programming, we can use the following steps:

1. Define a two-dimensional array "M" with rows representing the items and columns representing the weights. M[i][j] will represent the maximum value that can be achieved using the first i items with a knapsack of capacity j.
2. Initialize the array with base cases: M[0][j] = 0 for all j, and M[i][0] = 0 for all i. This represents the case where there are no items or the knapsack has a capacity of 0, in which case the maximum value is 0.
3. Iterate through the array and fill in the values using the following formula: M[i][j] = max(M[i-1][j], M[i-1][j-w[i]] + v[i]) where w[i] is the weight of the i-th item and v[i] is its value. The first part of the formula represents the case where the i-th item is not included in the knapsack, and the second part represents the case where it is included.
4. Once the array is filled in, the maximum value that can be achieved with a knapsack of capacity j is M[n][j], where n is the number of items.

Here is how we can apply these steps to solve the example problem with a capacity of 70 and the following list of items:

Items:

1. Book (weight: 10, value: 60)
2. Food (weight: 20, value: 100)
3. Water bottle (weight: 30, value: 120)
4. Sleeping bag (weight: 40, value: 200)
5. Tent (weight: 50, value: 300)
6. Define the array M with 5 rows (for the 5 items) and 71 columns (for the weights from 0 to 70).
7. Initialize the array with base cases: M[0][j] = 0 for all j M[i][0] = 0 for all i
8. Iterate through the array and fill in the values using the formula: M[i][j] = max(M[i-1][j], M[i-1][j-w[i]] + v[i])
9. Once the array is filled in, the maximum value that can be achieved with a knapsack of capacity 70 is M[5][70], which represents the maximum value that can be achieved using all the items.

Here is the final array M:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**...**	**70**
0	0	0	0	0	0	0	0	0	0	0	...	0
1	0	0	0	0	0	0	0	0	0	60	...	60
2	0	0	0	0	0	0	0	0	0	100	...	100
3	0	0	0	0	0	0	0	0	0	120	...	120
4	0	0	0	0	0	0	0	0	0	200		

אור אטיאס 207953308 פתרון ש"ב מס' 7 + 8 בקורס בינה מלאכותית

Advanced Greedy Algorithms Solution:

To solve the knapsack problem with a performance guarantee of δ-approximation, where δ is the desired approximation factor, we can use the following steps:

1. Sort the items in descending order based on their value-to-weight ratio (value/weight). This will ensure that the most valuable items are considered first.
2. Initialize a variable "total value" to 0 and a variable "total weight" to 0.
3. Iterate through the items and do the following for each item: a. If the item's weight plus the current total weight is less than or equal to the capacity of the knapsack (c), add the item to the knapsack and update the total value and total weight accordingly. b. If the item's weight plus the current total weight is greater than the capacity of the knapsack (c), compute the fraction of the item's value that can be added to the total value, based on the remaining capacity and the item's weight: value fraction = (c - total weight) / weight * value c. Add the value fraction to the total value and set the total weight to the capacity of the knapsack (c).
4. Return the total value as the solution to the knapsack problem.

For a δ-approximation algorithm, the value fraction in step 3b is computed as follows: value fraction = δ * (c - total weight) / weight * value. Here is how we can apply these steps to solve the example problem with a capacity of 70 and a desired approximation factor of 2, using the following list of items:

Items:

1. Book (weight: 10, value: 60)
2. Food (weight: 20, value: 100)
3. Water bottle (weight: 30, value: 120)
4. Sleeping bag (weight: 40, value: 200)
5. Tent (weight: 50, value: 300)
6. Sort the items in descending order based on their value-to-weight ratio:

Items:

1. Tent (weight: 50, value: 300, ratio: 6)
2. Sleeping bag (weight: 40, value: 200, ratio: 5)
3. Water bottle (weight: 30, value: 120, ratio: 4)
4. Food (weight: 20, value: 100, ratio: 5)
5. Book (weight: 10, value: 60, ratio: 6)
6. Initialize total value to 0 and total weight to 0.
7. Iterate through the items and do the following for each item: a. If the item's weight plus the current total weight is less than or equal to the capacity of the knapsack (70), add the item to the knapsack and update the total value and total weight accordingly. b. If the item's weight plus the current total weight is greater than the capacity of the knapsack (70), compute the fraction of the item's value that can be added to the total value, based on the remaining capacity and the item's weight: value fraction = 2 * (70 - total weight) / weight * value c. Add the value fraction to the total value and set the total weight to the capacity of the knapsack (70).
8. Return the total value as the solution to the knapsack problem.

Advanced Greedy Algorithm Solution:

Items:

1. Tent (weight: 50, value: 300)
2. Sleeping bag (weight: 40, value: 200)
3. Water bottle (weight: 30, value: 120)
4. Food (weight: 20, value: 100)
5. Book (weight: 10, value: 60)

Initialize total_value to 0 and remaining_capacity to 70.

Iterate through the sorted list of items and do the following for each item:

- For the tent (weight: 50, value: 300): Since the tent fits in the remaining capacity (70), add its value (300) to total_value and remove its weight (50) from remaining_capacity, setting total_value to 300 and remaining_capacity to 20.
- For the sleeping bag (weight: 40, value: 200): Since the sleeping bag fits in the remaining capacity (20), add its value (200) to total_value and remove its weight (40) from remaining_capacity, setting total_value to 500 and remaining_capacity to 0.
- For the water bottle (weight: 30, value: 120): Since the water bottle does not fit in the remaining capacity (0), but its value (120) is at least 2 times the remaining capacity (0), add 2 times the remaining capacity (0) to total_value and set the remaining_capacity to 0, setting total_value to 500 and remaining_capacity to 0.
- For the food (weight: 20, value: 100): Since the food does not fit in the remaining capacity (0) and its value (100) is less than 2 times the remaining capacity (0), skip it and move on to the next item.
- For the book (weight: 10, value: 60): Since the book does not fit in the remaining capacity (0) and its value (60) is less than 2 times the remaining capacity (0), skip it and move on to the next item.

At this point, we have iterated through all the items and our solution is to include the tent and the sleeping bag, with a total value of 500 and a total weight of 90.

Note that this solution is a 2-approximation of the optimal solution, as it guarantees that the value of the items included in the knapsack is at least half of the optimal value. However, it may not be the optimal solution itself, as it only considers the most valuable items and does not consider the overall weight of the items.