Algorithms
Greedy and DP

Computing Lab
CS 69201
Department of Computer Science and Engineering
IIT Kharagpur

Question

There are **N** pieces of items Dom wants to steal.
Each of those items “**i**” has a weight(**wi**) and a value(**vi**).
Dom wants to make sure he takes the maximum total value but also has a constraint on the amount of weight(**W**) he can steal in his car.
What is the maximum profit he can make and get away in his car?

Example 1:
N = 3, W = 30
w : [25,10,12,15]
v : [100, 30,35,90]

Greedy Approach

- Algorithm design which tries to find a local optimal choice solution at each step.
- After such steps, a solution is obtained.
- May or may not be optimal.
- Examples
    - Fractional Knapsack
    - Huffman Coding
    - Dijkstra's Algorithm

Greedy Algorithm

- Steps to solve problems using Greedy Algorithms
  - Define the problem statements and problem objective.
  - Determine the locally optimal choice at each step based on the current state.
  - Select the greedy choice and update the current state.
  - Continue the above steps till a solution is reached.

Problem 1

If Dom can divide the items into fractional parts. How can he obtain the maximum profit if the value of each item is directly proportional to its weight?

Greedy Problem 1

Solution :

- ➔ For each item, calculate the value per unit weight by dividing the value by the weight. Let call this Density (**di**)
- ➔ Sort all items in descending order based on their **density**.
- ➔ Iterate through the sorted list of items:
  - ◆ If the entire item can fit into the car, Dom takes it.
  - ◆ If the entire item cannot fit, he add as much of the item as possible. **Since he can take fractions of an item, he fills his car to its full capacity.**

Greedy Problem 1

Example 1 walkthrough:

- ➔ Calculate Value-to-Weight Ratio
  - ◆ Item1 : 100/25 = 4
  - ◆ Item2 : 30/10 = 3
  - ◆ Item3 : 35/12 = 2.92
  - ◆ Item4 : 90/15 = 6
- ➔ Sort the items in descending order based on their value-to-weight ratio : [Item4, Item1, Item2, Item3]
- ➔ Max_value = 0, total_capacity = 0
- ➔ Now iterating through the list, he adds item4 first, Max_value becomes 90, total_capacity = 15. Then he adds fractional part of Item1 to make total_capacity = 30, Max_value = 90 + 4*(30-15) = 150

So the total value he can take is 150

Problem 2

Now, unlike before, Dom cannot divide the items into fractional parts. He must either take an entire item or leave it. How can he obtain the maximum total value without exceeding the car's capacity?

Here are three greedy approaches Dom can take:

- ➔ Greedy by value-to-weight ratio
- ➔ Greedy by value
- ➔ Greedy by weight

Greedy Problem 2 (Greedy by value-to-weight ratio)

Select items based on the highest value-to-weight ratio (density) first.

Solution :

- ➔ Calculate the value-to-weight ratio for each item.
- ➔ Sort items in descending order based on this ratio.
- ➔ Starting from the top of the sorted list, add items to the car if they fit, until no more items can be added without exceeding the capacity.

For Example 1, after obtaining the sorted list [Item4, Item1, Item2, Item3], he can take item4(w = 15), has to skip item Item1(w = 25) and take Item2(w = 10).

So the total value he can take is 90+30=120, resulting capacity= 15+10=25

Greedy Problem 2 (Greedy by value)

This approach selects items based on their value.

Solution :

- ➔ Sort items in descending order by their value.
- ➔ Add items to the car in this order, until the capacity is reached.

For Example 1, after obtaining the sorted list :
[Item1(100,25), Item4(90,15), Item3(35,12), Item2(30,10)],

Dom can take item1(w = 25), and then has to skip all the remaining items.
So the total value he can take is 100, resulting capacity = 25

Greedy Problem 2 (Greedy by weight)

This approach focuses on minimizing the total weight while still maximizing the value. Choose the items with the smallest weight first, which allows more items to be added.

Solution :

- ➔ Sort items in ascending order by their weight.
- ➔ Add items to the car in this order, until the capacity is reached.

For Example 1, after obtaining the sorted list:
[Item2(30,10), Item3(35,12), Item4(90,15), Item1(100,25)],

Dom can take item2(w = 10), item3(w =12) and then has to skip all the remaining items.
So the total value he can take is 30+35=65, resulting capacity=10+12=22

Problem with the Greedy approach

In all of the greedy strategies we followed above :

- **Greedy by value-to-weight ratio**
  - Max_value = **120**, total_capacity = **25**
- **Greedy by value**
  - Max_value = **100**, total_capacity = **25**
- **Greedy by weight**
  - Max_value = **65**, total_capacity = **22**

List : [Item1(100,25),Item2(30,10), Item3(35,12), Item4(90,15)]
We can see that **we can get** a **Max_value=125** with **total_capacity=27**.
If we take **Item3** and **Item4**.

But none of the greedy approaches he took gave this result

Problem

Help Dom obtain the maximum total value without exceeding the car’s capacity by trying all possible combinations of items

Optimal Solution w/ Recursion

- Solve the problem by identifying the recurrence relation
- Recursively, solve the recurrence relation :
  - Identify the base cases
  - Solve the subproblems of the recurrence relations recursively
- Guarantees to provide optimal solution.

Trying all possible combinations

We can try all possible combinations and take the result which gives the maximum total value.

Let **maxValue(i, w)** be the maximum value obtainable using the first **i** items and a car capacity of **w**.

Base case :

➔ If **w = 0**, then **return 0**

➔ If **i = 0**, If i = 0, **return 0** if the item's weight > w else **return V[0]**

Try all possible combination:

➔ **Option1 = maxValue(i-1,w)** // Here we are skipping the current item

➔ If W[i]<=w : **Option2 = V[i] + maxValue(i-1,w-W[i])** // Take the item if possible

return max(Option1,Option2)

Optimal Solution w/ Recursion

(Trying all possibilities)

- However there are pitfalls
  - Redundant solving of subproblems. (Overlapping Subproblems)
  - Memory Issues (Stack Overflow)

Dynamic Programming to the Rescue

Problem

Help Dom obtain the maximum total value without exceeding the car's capacity by trying all possible combinations of items by storing the solution of subproblems and using it as precompute for some other subproblem.

Dynamic Programming

- Solve complex problems by breaking them down into simpler subproblems.
- **Avoids redundant computations** (overlapping subproblems)
- Has two types of approaches :
  - Top Down Approach
  - Bottom Up Approach

Dynamic Programming(Top-down)

Store the calculated value of a state so that we need not recalculate it later.

Define: **dp[N][W+1];** where N = number of items, W=max capacity of car
(Fill dp with a number that can never be a solution to our problem,let it be -1 here)

Now, in our previous recursive function, we make the following changes to store and check for calculated values :
Base case :

- ➔ If w = 0, then return 0
- ➔ If i = 0, return 0 if the item's weight > w else return V[0]
- ➔ **If dp[i][w]!=-1 : return dp[i][w]**

Try all possible combination:

- ➔ Option1 = maxValue(i-1,w) // Here we are skipping the current item
- ➔ If W[i]<=w : Option2 = V[i] + maxValue(i-1,w-W[i]) // Take the item if possible
- ➔ **dp[i][w] = max(Option1,Option2)**

return dp[i][w]

Dynamic Programming(Bottom-up)

To convert the memoization approach to a tabulation one, create a dp array with the same size as done in memoization. We can initialize it as 0.

First, we need to initialize the base conditions of the recursive solution.

- ➔ At ind==0, we are considering the first element, if the capacity of the car is greater than the weight of the first item, we store val[0] as answer. We will achieve this using a for loop.
- ➔ Next, we are done for the first row, so our ‘ind’ variable will move from 1 to n-1, whereas our ‘cap’ variable will move from 0 to ‘W’. We will set the nested loops to traverse the dp array.
- ➔ Inside the nested loops we will apply the recursive logic to find the answer of the cell.
- ➔ **dp[ind][cap] = max(dp[ind-1][cap],dp[ind-1][cap-weight[ind]])**
- ➔ When the nested loop execution has ended, we will return dp[N-1][W] as our answer.

Resources

- Divide and Conquer - https://en.wikipedia.org/wiki/Divide-and-conquer_algorithm
- Greedy - https://en.wikipedia.org/wiki/Greedy_algorithm
- Dynamic Programming - https://www.geeksforgeeks.org/dynamic-programming/?ref=lbp#when-to-use-dynamic-programming-dp
- Algorithms - https://www.geeksforgeeks.org/introduction-to-algorithms/
- Cormen, Leiserson, Rivest, Stein Algorithms Book CLRS https://dl.ebooksworld.ir/books/Introduction.to.Algorithms.4th.Leiserson.Stein.Rivest.Cormen.MIT.Press.9780262046305.EBooksWorld.ir.pdf

Exam on Dynamic Programming

- Tuesday August 6, 2024
- Duration : 2 hrs 30 minutes
- Exam Syllabus : Dynamic Programming Paradigm based Problems


INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR
भारतीय प्रौद्योगिकी संस्थान खड़गपुर
1951
योगः कर्मसु कौशलम्

Thank You!