Machine Learning

Dias, Schiewe, Pattanaik *Instructor: Pattanaik*

Polyhedral Theory
Dynamic Programming

§ Week IX §

Problem 1: Vending Machines

We have a vending machine with an unlimited supply of coins of denominations $D = \{d_1, d_2, \cdots, d_n\}$ and we want to make change for value V, i.e., we want to get the value V using only the denominations in D. Note that for some instances of the problem, the solution might not exist, e.g., denominations 5 and 10 can make change for 35 but not for 12.

Try to formulate and solve the decision variant of the coin-exchange problem using dynamic programming. You are given the value V – the task is to decide whether the value V can be composed by the denominations from D only.

Solution:

This problem is known as the **change-making problem**, which aims to find the minimum number of coins (in this case, denominations) that add up to a given amount of money. It is a particular case of the **integer knapsack problem** and has applications beyond just currency.

For any $1 \leq t \leq v$, let $C(t) = c$ where c is the minimum number of coins required to create the total t, using the given denominations. If it is not possible for any number of coins, then $C(t) = \infty$.

We calculate $C(t)$ based on $C(s)$ for $1 \leq s \leq t$. The recurrence relation is:

$$C(t) = \min_{\{x \in d1, \cdots, d_n\}} C(t - x) + 1 \tag{1.1}$$

We initialize $C(0) = 0$. We also declare that $C(u) = \infty$ when $u \leq 0$. The final solution is just whether $C(v)$ is less than or equal to k. The evaluation order is to start at t = 1, compute C(t), increment t and repeat.

Problem 2: Shipyard

A shipping company has a cargo ship docked in the harbour onto which it wants to load several shipping containers. There are n containers to choose from. The i-th container weigths $w_P i \geq 0$ tons, and if it is shipped, it will yield a $p_i \geq 0$ dollars profit.

1. Present a recursive DP formulation for the following problem: Given $p[1..n]$ and $w[1..n]$, compute the maximum profit that can be achieved by selecting any subset of the containers, subject to the weight restriction.

   Solution:

LaTeX template for this polyhedral theory
dynamic programming is *SEU-ML-Assign* open source at tvj.one/ml-tex under the MIT License. E-mail me@tvj.one for support.

For $0 \leq i \leq n$ and $0 \leq v \leq W$, define profit(i, v) to be the maximum profit that can be achieved by selecting a subset from among containers $\{1, ..., i\}$ for a ship that has a maximum capacity of v tons. To compute this array, we first consider the basis case. If $i = 0$, then there are no items to select from. If v = 0, then the ship has no capacity to take any containers. Thus, profit(i, v) = 0 if either $i = 0$ or $v = 0$.

Therefore, let us assume that $i \geq 1$ and $v \geq 1$. Then let us consider the i-th container. If we decide not to take this container, we receive no profit and we use up none of the remaining weight capacity. Therefore, the best we can do is to select from the previous $i - 1$ containers but we still have the full v units of weight capacity. Therefore, profit(i, v) = profit($i - 1$, v). On the other hand, if we decide to take the i-th container, then we receive a profit of pi dollars, but the remaining weight capacity is decreased by w_i, and we can select from the remaining $i - 1$ containers. Thus, profit(i, v) = p_i + profit($i - 1$, vw_i). If $w_i \geq v$, this will yield a negative parameter value. To keep the notation simple and avoid multiple cases, we make the assumption that "if any parameter is negative, the profit function returns $-\infty$".

Since we do not know *a priori* which option is better, we try both and take the better of the two options. This yields the following recursive rule:

$$profit(i, v) = \left\{ \begin{array}{ll} -\infty, & \text{if } v \leq 0 \\ 0, & \text{if } i = 0 \text{ or if } v = 0 \\ max(profit(i-1, v), p_i + profit(i-1, v - w_i)), & \text{if } i, v \geq 0 \end{array} \right\}$$

The final answer is the maximum profit to select from all the n containers for a ship of full capacity W, that is, profit(n, W). (Because each entry of the table takes constant time to compute, this can be implemented in O(nW) time.)

2. The shipping company adds the constraint that each ship has a maximum capacity of C containers. Modify your solution to the previous part to handle this constraint.

Solution:

To handle the container limit, we add an additional parameter to the formulation. For $0 \leq i \leq n$, $0 \leq v \leq W$, and $0 \leq c \leq C$, define profit(i, v, c) to be the maximum profit that can be achieved by selecting a subset from among containers $\{1, ..., i\}$ for a ship that has a maximum capacity of v tons, and can hold c containers. The above rule is modified in the following manner.

If $c = 0$, then the profit is zero. Otherwise, whenever we choose to take a container, we decrease the number of allowable containers by one. Thus, we have the following rule:

$$profit(i, v, c) = \left\{ \begin{array}{ll} -\infty, & \text{if } v \leq 0 \\ 0, & \text{if } i = 0 \text{ or if } v = 0 \text{ or } c = 0 \\ max(profit(i-1, v), p_i + profit(i-1, v - w_i, c-1)), & \text{if } i, v \geq 0 \end{array} \right\}$$

As before, if the v_k index is negative, the value is $-\infty$. The final answer is the maximum profit to select from all the n containers for a ship of full weight capacity W, and full container capacity C, that is, profit(n, W, C). (Because each entry of the table takes constant time to compute, this can be implemented in $O(nWC)$ time.)

3. The shipping company has discovered that it has a second ship, identical to the first. Modify your solution to the second part to determine the maximum achievable profit by selecting containers and assigning each to one of these ships.

Solution:

Our solution is to replicate the weight and capacity indices, one set for each of the ships. For $0 \leq i \leq n$, $0 \leq v_1$, $v_2 \leq W$, and $0 \leq c_1$, $c_2 \leq C$, define profit(i, v_1, c_1, v_2, c_2) to be the maximum profit that can be achieved by selecting a subset from among containers $\{1, ..., i\}$ where for $k \in \{1, 2\}$, ship k has a maximum capacity of v_k tons, and can hold c_k containers.

With each container we can decide either to leave it, add it to the first ship, or add it to the second. As before, if the v_k index is negative, the value is $-\infty$.

$$profit(i, v, c_1, v_1, v_2) = \left\{ \begin{array}{ll} -\infty, & \text{if } \min(v_1, v_2) \\ 0, & \text{if any parameter is zero} \\ max(profit(i-1, v_1, v_2, c_2), & \\ \quad p_i + profit(i-1, v_1 - w_i, c-1, v_2, c_2)), & \\ \quad p_i + profit(i-1, v1, c_1, v2 - w_i, c_2 - 1)), & \text{if all indices} \geq 0 \end{array} \right\}$$

The final answer is the maximum profit to select from all the n containers with both ships of full weight capacity W, and full container capacity C, that is, profit(n, W, C, W, C). (Because each entry of the table takes constant time to compute, this can be implemented in $O(nW^2C^2)$ time.)

Problem 3: Pharmacies

A pharmacist has W pills and n empty bottles. Let b_i denote the number of pills that can fit in a bottle i. Let vi denote the cost of purchasing bottle i. Given W, b_i's and v_i's, we wish to compute the cheapest subset of bottles to place all W pills.

- Present an algorithm for this version of the problem.

  Let v_i be the cost of bottle i, and let b_i denote the number of pills it holds. In order to place W pills as inexpensively as possible, sort the bottles in increasing order of cost per pill, that is, $\frac{v_i}{/} b_i$. Then fill the bottles in this order, until all the pills are gone.

  To show that this is optimal, define the incremental cost for a pill to be $\frac{v_i}{/} b_i$, where i denotes the bottle into which this pill was placed. Because we only pay for the portion of the bottle that we use, the total cost of bottling all the pills is equal to the sum of the incremental costs over all W pills. (This is important. For example, we can put a single pill into a last bottle, without paying for the entire bottle.)

  For a given input, let G denote the sorted order of incremental costs for the greedy solution, and let O denote the sorted order of incremental costs for any optimal solution. We will use the usual exchange argument to show that G achieves the same cost as O.

  If $G = O$, we are done. Otherwise, consider the first pill (in sorted order of incremental cost) where O differs from G. Let i denote the bottle into which G puts this pill, and let i denote the bottle used by O. Since both sequences have been sorted, we know that O does not put any more pills into bottle i, even though there is still space remaining there (since G put this pill there). Since G places pills in increasing order of incremental cost, it must be that $\frac{v_i}{/} b_i \geq \frac{v_{i'}}{b_{i'}}$. Let us create a new bottling plan by moving this one pill to bottle i' to bottle i. The incremental change in cost by doing this is $\frac{v_i}{/} b_i - \frac{v_{i'}}{b_{i'}} \geq 0$. Therefore, the total cost cannot increase as a result of the change. (Since O is optimal, it should not decrease.) By repeating this process, we will eventually convert O into G, while never increasing the bottling cost. Therefore, G is optimal.

- Suppose that this assumption does not hold. That is, you must buy the entire bottle, even if only a portion of it is used. Present an algorithm for this version of the problem. (Hint: Use DP. It suffices to give the recursive formulation.) Justify your algorithm's correctness.

For $0 \leq i \leq n$, define $P(i, w)$ to be the minimum amount paid assuming that we place w pills using some subset of the first i bottles. For the basis case, observe that if $w = 0$ and $i = 0$, we can trivially put the 0 pills in 0 bottles for a cost of 0, and thus $P(0, 0) = 0$. If $W \geq 0$, then there is no solution using 0 bottles, and so we have $P(0, W) = \infty$.

For the induction, let us assume that $i \geq 1$. There are two cases. Either we do not use the i-th bottle or we do. If not, we put all w pills in the first $i - 1$ bottles, for a cost of $P(i - 1, w)$. Otherwise, we put $\min(b_i, w)$ pills in this bottle, and put the remaining pills in the previous $i - 1$ bottles. The total cost is $v_i + P(i - 1, w - \min(b_i, w))$. We prefer the lower of the two choices, which implies the following recursive rule:

$$P(0, w) = \left\{ \begin{array}{ll} 0, & \text{if } w = 0 \\ \infty, & \text{otherwise} \end{array} \right\}$$

$$P(i, w) = \min(P(i - 1, w), v_i + P(i - 1, w - \min(b_i, w)) \quad \text{for } i \geq 1. \tag{3.1}$$

Problem 4: Ford vs Dijkstra

Use the Bellman-Ford algorithm to calculate the shortest path in this following graph:

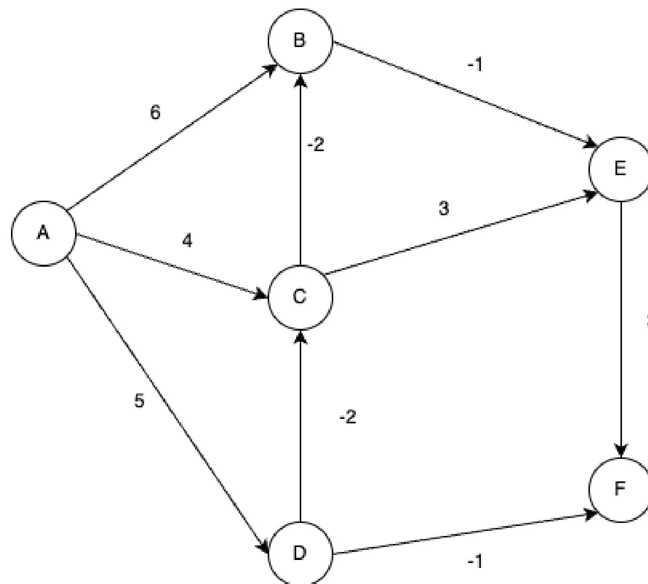

Figure 1: DAG with negative flows

Solution:

In order to calculate the path, we can apply Bellman-Ford algorithm:

- Set distance to A is zero and for all the other nodes equal to ∞;
  $$\begin{pmatrix} & A & B & C & D & E & F \\ 0 & 0 & \infty & \infty & \infty & \infty & \infty \end{pmatrix}$$
- At first iteration, we can update the distance to all nodes based on current information;
  $$\begin{pmatrix} & A & B & C & D & E & F \\ 0 & 0 & \cancel{\infty} & \cancel{\infty} & \cancel{\infty} & \cancel{\infty} & \cancel{\infty} \\ 1st & 0 & 6 & 3 & 5 & 5 & 4 \end{pmatrix}$$

- From the first iteration, we updated the position for C twice, one from A and another from D. This did not happen to B, because C was only updated after B and this algorithm does not work backwards (within an iteration);

- Second iteration goes like this:

$$\begin{pmatrix} & A & B & C & D & E & F \\ 0 & 0 & \cancel{\infty} & \cancel{\infty} & \cancel{\infty} & \cancel{\infty} & \cancel{\infty} \\ 1st & 0 & \cancel{6} & 3 & 5 & 5 & 4 \\ 2nd & 0 & 1 & 3 & 5 & 5 & 4 \end{pmatrix}$$

- Through this iteration, the distance to B was updated.

- Next iteration:

$$\begin{pmatrix} & A & B & C & D & E & F \\ 0 & 0 & \cancel{\infty} & \cancel{\infty} & \cancel{\infty} & \cancel{\infty} & \cancel{\infty} \\ 1st & 0 & \cancel{6} & 3 & 5 & 5 & 4 \\ 2nd & 0 & 1 & 3 & 5 & \cancel{5} & \cancel{4} \\ 3rd & 0 & 1 & 3 & 5 & 0 & 3 \end{pmatrix}$$

- The distance to E and F were updated. Because, E is updated before F, both can be updated in the same iteration.

- One more pass through the graph:

$$\begin{pmatrix} & A & B & C & D & E & F \\ 0 & 0 & \cancel{\infty} & \cancel{\infty} & \cancel{\infty} & \cancel{\infty} & \cancel{\infty} \\ 1st & 0 & \cancel{6} & 3 & 5 & 5 & 4 \\ 2nd & 0 & 1 & 3 & 5 & \cancel{5} & \cancel{4} \\ 3rd & 0 & 1 & 3 & 5 & 0 & 3 \\ \color{red}{4th} & \color{red}{0} & \color{red}{1} & \color{red}{3} & \color{red}{5} & \color{red}{0} & \color{red}{3} \end{pmatrix}$$

The resulting graph is:

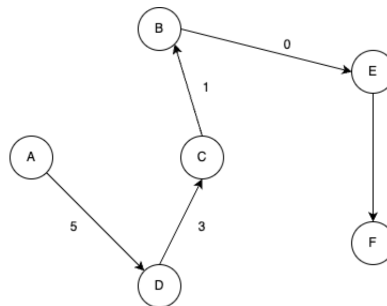

Figure 2: Resulting graph with shortest distance