Dynamic Programming - 0-1 Knapsack Problem, Parenthesizations

CS313E - Elements of Software Design

Kia Teymourian

11/08/2022

Agenda

1. 0-1 Knapsack Problem

2. Matrix-chain Multiplication - Parenthesization

Dynamic Programming - 0-1 Knapsack Problem

- In 0-1 Knapsack problem, we have a set of items that each have different values and weights.
- We want to collect some of these items in a bag (In German a "Sack") so that we do not go over a maximum weight capacity of our bag
- and maximize the total value of our bag while its weight is smaller or equal to the bag's maximum weight capacity.

In 0-1 Knapsack, items cannot be broken into pieces which means we can pick the item as a whole or leave it.

For example, we have the following items to select:

Items	1	2	3	4
Values	10	40	30	50
Weights	5	4	6	3

We can select each item only once.

Example - 0-1 Knapsack Problem

Items	1	2	3	4
Values	10	40	30	50
Weights	5	4	6	3

The Capacity of our knapsack has max weight of $W = 10$,

- Many options ...
- We can pick up $((4, (v = 50, w = 3)) + (2, (v = 40, w = 4))$ $Knapsack(v = 90, w = 7)$ Optimal solution here.

0-1 Knapsack Problem, Formal Definition

Given two n-tuples of positive numbers $\langle v_1, v_2, \ldots, v_n \rangle$ **and** $\langle w_1, w_2, \ldots, w_n \rangle$, **and** $W > 0$

We want to determine the subset of $T \subseteq 1, 2, \ldots, n$ of items to store in way to maximize the value of the set.

$$\textbf{Maximize } \sum_{i \in T} v_i$$

$$\textbf{Subject to } \sum_{i \in T} w_i \leq W$$

- **This is an optimization problem.**
- We can do brute force and try all 2^n possible subsets of T

0-1 Knapsack Problem, Formal Definition

Given two n-tuples of positive numbers $\langle v_1, v_2, \ldots, v_n \rangle$ **and** $\langle w_1, w_2, \ldots, w_n \rangle$, **and** $W > 0$

We want to determine the subset of $T \subseteq 1, 2, \ldots, n$ of items to store in way to maximize the value of the set.

$$\textbf{Maximize } \sum_{i \in T} v_i$$

$$\textbf{Subject to } \sum_{i \in T} w_i \leq W$$

- **This is an optimization problem.**
- We can do brute force and try all 2^n possible subsets of T

The question as usual is: "Can we do better than brute force?"

0-1 Knapsack Problem, Formal Definition

Given two n-tuples of positive numbers $\langle v_1, v_2, \ldots, v_n \rangle$ **and** $\langle w_1, w_2, \ldots, w_n \rangle$, **and** $W > 0$

We want to determine the subset of $T \subseteq 1, 2, \ldots, n$ of items to store in way to maximize the value of the set.

$$\text{Maximize} \sum_{i \in T} v_i$$

$$\text{Subject to} \sum_{i \in T} w_i \leq W$$

- **This is an optimization problem.**
- We can do brute force and try all 2^n possible subsets of T

The question as usual is: "Can we do better than brute force?"

Yes–Dynamic programming (DP)!

Step-1: Define the Subproblems.

- Sub-problems are possible combinations of all items with their values and weights.
- We can define the given item data as an Array of form $[0 \ldots n, 0 \ldots W]$.
- The value of $V[i, w]$ stores the maximum combination possible, where $1 \leq i \leq n$ **and** $0 \leq w \leq W$
- If we compute all possible combination of such array then one of them $V[n, W]$ will contained the maximum possible that we are looking for it.

Step-2: Recursively define the value of an optimal solution considering the Subproblems of Step-1

We need to think how to tackle the problem by using recursion.
Initialization:

- $V[0, w] = 0$ **for** $0 \leq w \leq W$ when we have no item in the sack.
- $V[0, w] = -\infty$ **for** $w \leq 0$ weights are never negative. This is illegal to have.

We can define the recursive step to be the following, when we start backwards from the optimal solution:

$$V[i, w] = max(V[i-1, w], v_i + V[i-1, w-w_i])$$
$$1 \leq i \leq n \text{ and } 0 \leq w \leq W$$

Time per sub-problem would be then $O(1)$

Step-3: Think of a possible Bottom-Up Computation of $V[i, w]$.

- We want to use iterations and no more recursions.
- $\boldsymbol{V[i,w] = max(V[i-1,w], v_i + V[i-1,w-w_i])}$
- Bottom is $\boldsymbol{V[0,w] = 0 \text{ for all } 0 \leq w \leq W}$

Back to our Example -

Items	1	2	3	4
Values	10	40	30	50
Weights	5	4	6	3

Example: We have a Knapsack with Capacity $\boldsymbol{W = 8}$.

i/w	0	1	2	3	4	5	6	7	8
$i = 0$	0	0	0	0	0	0	0	0	0
$i = 1$	0	0	0	0	0	10	10	10	10
$i = 2$	0	0	0	0	40	40	40	40	40
$i = 3$	0	0	0	0	40	40	40	40	40
$i = 4$	0	0	0	50	50	50	50	**90**	**90**

Back to our Example -

Items	1	2	3	4
Values	10	40	30	50
Weights	5	4	6	3

Example: We have a Knapsack with Capacity $\boldsymbol{W = 8}$.

i/w	0	1	2	3	4	5	6	7	8
$i = 0$	0	0	0	0	0	0	0	0	0
$i = 1$	0	0	0	0	0	10	10	10	10
$i = 2$	0	0	0	0	40	40	40	40	40
$i = 3$	0	0	0	0	40	40	40	40	40
$i = 4$	0	0	0	50	50	50	50	**90**	**90**

- Our final output would be then $V[4, 7] = 90$
- Here, we do not tell which subset makes this optimal solution. The optimal solution is $[2, 4]$

Knapsack Bottom UP

Algorithm 1 KnapSack(v, w, n, W)

```
for w = 0 to W do
    V[0, w] = 0
end for
for i = 1 to n do
    for w = 0 to W do
        if w[i] ≤ w then
            V[i, w] = max{V[i − 1, w], v[i] + V[i − 1, w − w[i]]}
        else
            V[i, w] = V[i − 1, w]
        end if
    end for
end for
```

The running time is $O(nW)$

Knapsack Bottom UP

- So far, we do not know which subset gives the optimal subset.
- We can use a helper array of $keep[i, w]$ to keep track of the items that we want to keep in the optimal solution. It would be 1 if we keep it and 0 if we drop the item.

Knapsack Bottom UP with optimal set output

Algorithm 2 KnapSack(v, w, n, W)

for $w = 0$ to W **do**
    $V[0, w] = 0$
end for
for $i = 1$ to n **do**
    for $w = 0$ to W **do**
        if $w[i] \leq w$ and $v[i] + V[i-1, w-w[i]] \geq V[i-1, w]$ **then**
            $V[i, w] = v[i] + V[i-1, w-w[i]]$
            $keep[i, w] = 1$ ▷ Keep it!
        else
            $V[i, w] = V[i-1, w]$
            $keep[i, w] = 0$ ▷ Drop it!
        end if
    end for
end for
$K = W$
for $i = n$ DownTo 1 **do**
    if $keep[i, k] == 1$ **then**
        Print(i); K=K-W[i]; ▷ Here we output the elements of optimal Set
    end if
end for
return $V[n, W]$

Matrix-Chain Multiplication - Parenthesization

Matrix-Chain Multiplication - Parenthesization

In Matrix Multiplication the order of computation controls the computation complexity.
For example consider the following case:
$\boldsymbol{D = A_{n\times 1} \cdot B_{1\times n} \cdot C_{n\times 1}}$
We have the following two options here:

1. $D = (A_{n\times 1} \cdot B_{1\times n})_{n\times n} \cdot C_{n\times 1}$ This computation will take $\Theta(n^2)$ Quadratic Time
2. $D = A_{n\times 1} \cdot (B_{1\times n} \cdot C_{n\times 1})_{1\times 1}$ This computation will take $\Theta(n)$ Linear Time

MATRIX-MULTIPLY

Algorithm 3 MATRIX-MULTIPLY

```
if A.columns ≠ B.rows then
    error "incompatible dimensions"
else
    let C be a new A.rows × B.columns matrix
    for i = 1 to A.rows do
        for j = 1 to B.columns do
            c_ij = 0
            for k = 1 to A.columns do
                c_ij = c_ij + a_ik.b_kj
            end for
        end for
    end for
    return C
end if
```

Dynamic Programming Steps - Parenthesization

1. **Sub-Problems.** number of possible combinations of sub-strings $(A_1(A_2A_3))$ or $((A_1A_2)A_3)$
2. **Guessing.** What is the outermost multiplication. How we go up/back from there.
3. **Building Recursive implementation and Recurrence.** If I have *itoj* number of array multiplications, it would be $A_i \dots A_{k-1} \cdot A[k] \dots A_{j-1}$ Cost per subproblem $O(n)$ if the subproblem has n elements.
4. **DAG.** This problem is again a DAG that we need to work on topological order of it. Total run time for it $O(n^3)$
5. **Solve the main Problem.** Merge the solutions of the subproblem and solve the original problem.

MATRIX-CHAIN-ORDER - Computing the Optimal Costs

Algorithm 4 MATRIX-CHAIN-ORDER

```
n = p.length − 1
let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
for i = 1 to n do
    m[i, i] = 0
end for
for l = 2 to n do                          ▷ Here l is the chain length of matrices.
    for i = 1 to n − l + 1 do
        j = i + l − 1
        m[i, j] = ∞
        for k = i to j − 1 do
            q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
            if q < m[i, j] then
                m[i, j] = q
                s[i, j] = k
            end if
        end for
    end for
end for
return m and s
```

Optimal Paranthesis.

MATRIX-CHAIN-ORDER determines the optimal number of scalar multiplications needed to compute a matrix-chain product, it does not directly show how to multiply the matrices.

Algorithm 5 PRINT-OPTIMAL-PARENS(s, i, j)

```
if i == j then
    print("A")
else
    print("(")
    PRINT-OPTIMAL-PARENS (s, i, s[i, j])
    PRINT-OPTIMAL-PARENS (s, s[i, j] + 1, j)
    print(")")
end if
```

- $s[1..n-1, 2..n]$ The table that provides info that we need to do so.
- Each entry $s[i,j]$ records a value of k such that an optimal parenthesization of $A_i A_{i+1} \dots A_j$ splits the product between A_k and A_{k+1} .

Cost of Matrix-chain multiplication problem

- We can solve the matrix-chain multiplication problem by either a **top-down, memoized dynamic-programming algorithm or a bottom-up dynamic-programming** algorithm in $\boldsymbol{O(n^3)}$.
- We take advantage of the overlapping subproblems property.
- The number of parenthesizations is at least $T(n) \geq T(n-1) + T(n-1)$.
  Since the number with the first element removed is $T(n-1)$, which is also the number with the last removed
  Thus the number of parenthesizations is $\Omega(2^n)$
  Number of distinct sub-problems is $\boldsymbol{\Theta(n^2)}$.

References and Readings List

- More details about Matrix Chain parenthesizations
  `https://sites.radford.edu/~nokie/classes/360/dp-matrix-parens.html`

Readings from CLRS Book (Introduction to Algorithms, 3rd Edition)

- Section 15 Dynamic Programming
- Section 15.1 Rod Cutting
- Section 15.2 Matrix-chain multiplication
- Section 15.3 Elements of dynamic programming
- Section 15.5 Optimal binary search trees