Knapsack Revisited: DP Notes

Leon Xu, Lily Bhattacharjee

May 22, 2019

1 What is DP?

Dynamic programming is a technique commonly used in approximation algorithms when a problem can be solved from solving smaller subproblems that compose it. To save time in calculating these smaller solutions (especially when results might be recalculated repeatedly – think recursive Fibonacci), DP algorithms are usually implemented iteratively, with intermediate results being stored in a table / list.

Weakly NP-hard problems: Some can be implemented in polynomial time with DP proportionally to the input size in unary rather than binary (pseudopolynomial algorithms). By rounding all input values so that the number of *unique* input values is polynomial to input size and error parameter $\epsilon > 0$, the pseudopolynomial algorithm runs in polynomial time.

Scheduling problems: These types of problems have "large" and "small" parts w.r.t. processing time. By rounding large input values so that we have a number n unique values that is proportional to the input size and ϵ, we can use DP to approximate a solution to the scheduling problem in polynomial time.

2 Introducing the Knapsack Problem

A traveler has a knapsack with some limited capacity $B > 0$. The parameters of the problem are the following:

- $I = \{1, ..., n\}$ (a list of n items)
- item i has value $v_i > 0$, size $s_i > 0$

The goal: return a list of items $S \in I$ to maximize the value of the items the traveler fits in his knapsack while taking into account the constraint of the knapsack capacity. In formal notation:

$$\max(\textstyle\sum_{i \in S} v_i)$$
$$\text{Constraints: } \textstyle\sum_{i \in S} s_i \leq B$$

Note: We ignore any items that can't fit in the knapsack on their own i.e. $s_i > B$ for some $i \in S$

3 A DP Solution

1. Maintain an array A of length n. Each element of A is a pair (t, w). Here, at location j, t stands for the total space that some subset of items S takes up where $|S| = j$, and w stands for the total value of all items in the aforementioned S. This is based on the assumption that $\sum_{i \in S} s_i = t \leq B$. Formally, $\sum_{i \in S} v_i = w$.

2. There can be many such (t, w) pairs for particular j values, so we'll just keep track of the most space efficient, value-maximizing ones. A pair (t_1, w_1) can dominate a pair (t_2, w_2) for the same j value if $t_1 \leq t_2$ and $w_1 \geq w_2$ i.e. more value for less space. Domination is transitive.

3. At any point in the algorithm, no pair in $A(j)$ should dominate another i.e. if $A(j)$ contains pairs $(t_1, w_1), ..., (t_k, w_k)$, $t_1 < t_2 < ... < t_k$ and $w_1 < w_2 < ... < w_k$.

4. Note that there can be $B + 1$ pairs max in $A(j)$ (0 items ... B items if each item in the knapsack has size 1). Also note that if V is the sum of all item values in the knapsack, there are max $V + 1$ pairs in the list (0 value ... V value by the domination inequality enforced from above).

5. Any subset S s.t. $|S| = j$ with corresponding size $\sum_{i \in S} s_i$ and value $\sum_{i \in S} v_i$ is dominated by some element in $A(j)$.

Pseudocode:

$A(1) = \{(0, 0), (s_1, w_1)\}$
for $j = 2...n$
    $A(j) = A(j-1)$ // accumulates pairs of $\leq j - 1$ items
    for each pair $(t, w) \in A(j-1)$
        if $t + s_j \leq B$ // if the newest item fits in the pack with an existing sequence
            add $(t + s_j, w + v_j)$ to $A(j)$
    remove dominated pairs from $A(j)$ // we won't be considering these low-efficiency pairs anymore
return $max_{(t,w) \in A(n)} w$ // return the maximum value knapsack setup for number of total items $\leq n$

4 DP Correctness

Theorem: Our DP implementation from the pseudocode above will correctly determine the optimal value of the knapsack problem.

Proof: We will prove this statement by induction.

Base case: We consider the list $A(1)$. In the beginning of the program this is set to $\{(0, 0), (s_1, w_1)\}$. Hence, $A(1)$ contains all non-dominated pairs corresponding to feasible sets $S \in \{1\}$ (empty set, set of item 1).

Induction hypothesis: Assume that the statement holds true for list $A(j-1)$.

Inductive step: Consider S, a subset of the first j items. We know that $t = \sum_{i \in S} s_i \leq B$ (fits in the knapsack), $w = \sum_{i \in S} v_i$ (sum of item values). We claim the contrary – that there exists some pair $(t_f, w_f) \in A(j)$ s.t. $t_f \leq t$, $w_f \geq w$. We split this assumption into cases:

Case 1 – $j \notin S$: By the induction hypothesis and the initial setting of $A(j) = A(j-1)$ in the beginning of the iteration, $A(j)$ will contain all non-dominated pairs corresponding to feasible sets $S \in \{1, ..., j\}$.

Case 2 – $j \in S$: By the induction hypothesis, there exists some $(\hat{t}, \hat{w})$ pair in $A(j-1)$ that dominates $(\sum_{i \in S - \{j\}} s_i, \sum_{i \in S - \{j\}} v_i)$ s.t. $\hat{t} \leq \sum_{i \in S - \{j\}} s_i$ and $\hat{w} \geq \sum_{i \in S - \{j\}} v_i$. This occurs when the if condition in the pseudocode holds true, so the pair $(\hat{t} + s_j, \hat{w} + v_j)$ will be added to $A(j)$ if $\hat{t} + s_j \leq t \leq B$ and $\hat{w} + v_j \geq w$. Hence, the pair $(t_f, w_f) \in A(j)$ that dominates (t, w) will always exist.

Analyzing the pseudocode, the DP algorithm will take $O(n\min(B, V))$ time ($\min(B, V)$) for the maximum list size, n for the number of lists). Note that this not polynomial time even though it is linear w.r.t. n because all inputs are binary-encoded i.e. the size of input B is actually $\log_2 B$. This means that runtime $O(nB)$ is exponential w.r.t. B. Going back to binary vs. unary-encoded inputs, if the input was in unary, $O(nB)$ would be polynomial in input size.

5 A Polynomial-Time Approximation

pseudopolynomial: an algorithm that has a running time polynomial to the size of the input when the input values are unary-encoded

Note: If max possible item value V were polynomial in n, the runtime would be polynomial to input size. We can get a polynomial time approximation of the knapsack problem by rounding item values. While this won't give the optimal solution in most cases, it will be close.

polynomial-time approximation scheme (PTAS): a group of algorithms $\{A_\epsilon\}$ where for each $\epsilon > 0$, $A_{1+\epsilon}$ is a minimization approximation algorithm or $A_{1-\epsilon}$ is a maximization approximation algorithm

Note: Runtime of A_ϵ is dependent on $\frac{1}{\epsilon}$ (potentially exponential).

fully polynomial-time approximation scheme (FPAS, FPTAS): approximation algorithm A_ϵ with runtime bounded by a polynomial in $\frac{1}{\epsilon}$ (removes exponential dependence from consideration)

A general approach:

1. Item values will be measured in multiples of some μ. Round each value v_i to the nearest integer multiple $q\mu$.

2. The new values v_i' will be $\lfloor \frac{v_i}{\mu} \rfloor$ for each item i.

3. Run our previously-specified DP algorithm on the new values.

Note: The rounding loss doesn't have a huge effect but at the same time, we can now run our algorithm in polynomial time. Because we are rounding down to the nearest multiple of μ, each value v_i' will be off from v_i by max μ. Hence, each feasible solution may differ by max $n\mu$.

We want to specify our error as $\epsilon * \text{lower\_bound}(OPT)$. If M is the max item value, we can have M be a potential OPT lower bound because we could just pack the max-value item by itself. Therefore, to specify μ, we have the following relationship: $n\mu = \epsilon M$, or $\mu = \frac{\epsilon M}{n}$.

With the changed item values v_i', we have $V' = \sum_{i=1}^{n} v_i' = \sum_{i=1}^{n} \lfloor \frac{v_i}{\epsilon M/n} \rfloor$ by substituting for μ. From this, we see that V' is on the order of $O(\frac{n^2}{\epsilon})$. Because the iteration in the original DP knapsack is run n times, the runtime of the new algorithm is $O(\frac{n^3}{\epsilon})$ (bounded by a polynomial w.r.t. $\frac{1}{\epsilon}$.

Pseudocode:
$M = \max_{i \in I} v_i$ // item with the maximum value
$\mu = \frac{\epsilon M}{n}$
$v_i' = \lfloor \frac{v_i}{\mu} \rfloor$ for all items in I run the DP algorithm on the new scaled knapsack instance with values v_i'

6 Approximation Optimality

Theorem: Our polynomial-time approximation for for knapsack algorithm returns a solution $\leq (1-\epsilon)$ times OPT.

Below, we list our definitions:

S – set of items returned by our approximation algorithm
O – optimal set of items fulfilling knapsack constraints

We know $M \leq OPT$ (only putting the max value item in the knapsack). Also, based on the definition

of v_i', we know that v_i is bounded by $\mu v_i'$ and $(\mu+1)v_i'$ (rounded down to the nearest multiple of μ). From this, we can state the following:

$$\begin{gathered}
\sum_{i\in S} v_i \geq \mu \sum_{i\in S} v_i', \text{ because we rounded the values down} \\
\geq \mu \sum_{i\in O} v_i' \\
\geq \sum_{i\in O} v_i - |O|\mu \\
\geq \sum_{i\in O} v_i - n\mu, \text{ because } |O| \text{ is max } n \text{ (all items in OPT)} \\
= \sum_{i\in O} v_i - \epsilon M, \text{ because we found earlier that } n\mu = \epsilon M \\
\geq OPT - \epsilon OPT = (1-\epsilon)OPT
\end{gathered}$$