Lec21-subset-sum-knapsack

Tuesday, October 22, 2024 3:09 PM

Recall: Dynamic programming is like divide-and-conquer in that you build up big solutions from smaller ones, but is more general / powerful in that small solutions might be used many times.

generally applied in problems where brute-force would be exponential.

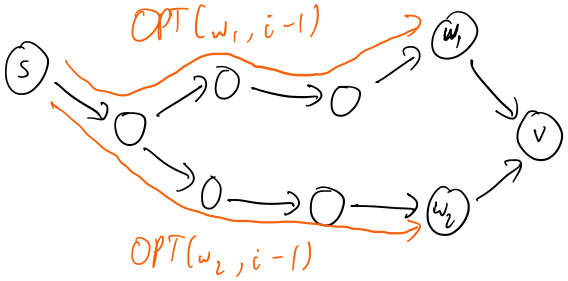
Bellman-Ford shortest path

Let $OPT(v, i)$ = min-cost of $s \to v$ path using $\leq i$ edges.

Can recursively define $OPT(v, i)$:

1. If best $s \to v$ path uses at most $i-1$ edges, then
$$OPT(v, i) = OPT(v, i-1)$$

2. If best $s \to v$ path uses i edges, and the last edge is (w, v), then
$$OPT(v, i) = d(w, v) + OPT(w, i-1)$$
← similar to Ford update rule.



Let $N(w)$ be neighbors of w.

Recurrence

$$OPT(v, i) = \min \begin{cases} OPT(v, i-1) \\ \min_{w \in N(v)} \left[OPT(w, i-1) + d(w, v) \right] \end{cases}$$

$OPT(v, x)$ only depends on $OPT(w, y)$ for $y < x$

Base case: $OPT(v, 1) = d(s, v)$ or ∞ if (s, v) does not exist

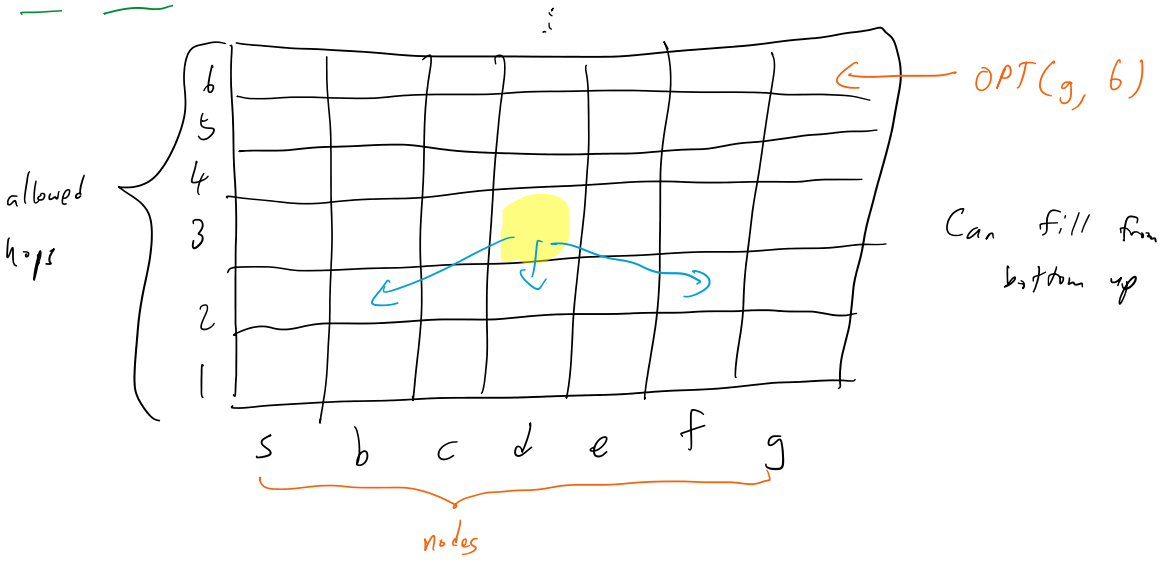
Goal: compute $OPT(t, n-1)$ $|V|$ nodes $|V|-1$ edges (max) here

Goal: compute $OPT(t, n-1)$

$|V|$ nodes, $|V|-1$ allowed hops

Only $|V| \times (|V|-1)$ possible arguments for $OPT(\cdot, \cdot)$

Encode as matrix



Naive runtime: $O(|V|^2)$ subproblems

$O(|V|)$ time to compute each entry in table
(have to search all possible neighbors)

$\Rightarrow O(|V|^3)$ time.

Better analysis: Let n_v be # in-edges of v.

Actually only $O(n_v)$ time to compute entry for v.

Total time $= O\left(|V| \sum_{v \in V} n_v\right) = O(|V||E|)$.

in each row only $O(|E|)$ computations
Since just need to check each in-edge once.

Subset Sum KT 6.4 (special easy case of knapsack problem)

Given integer bound W, and n items with integer weight $w_i \in \mathbb{N}$

find a subset S of items that maximizes $\sum_{i \in S} w_i$ while keeping $\sum_{i \in S} w_i \leq W$.

Motivation fully schedule a piece of equipment so least downtime
e.g. CPU or mass-spectrometer

Aside: integer weights very important here

Let S^* be an optimal choice of items.

Let $OPT(n, W)$ be the value of the optimal solution $OPT(n, W) = \sum_{i \in S^*} w_i$
└ compute this first, then keep info needed to reconstruct S^*.

Subproblems: Compute $OPT(j, w)$ for optimal value with a knapsack of size $w \leq W$ and the first j items.

Recurrence:

$$OPT(j, w) = \max \begin{cases} OPT(j-1, w) & \text{if } j \notin S^* \\ w_j + OPT(j-1, w - w_j) & \text{if } j \in S^* \end{cases}$$

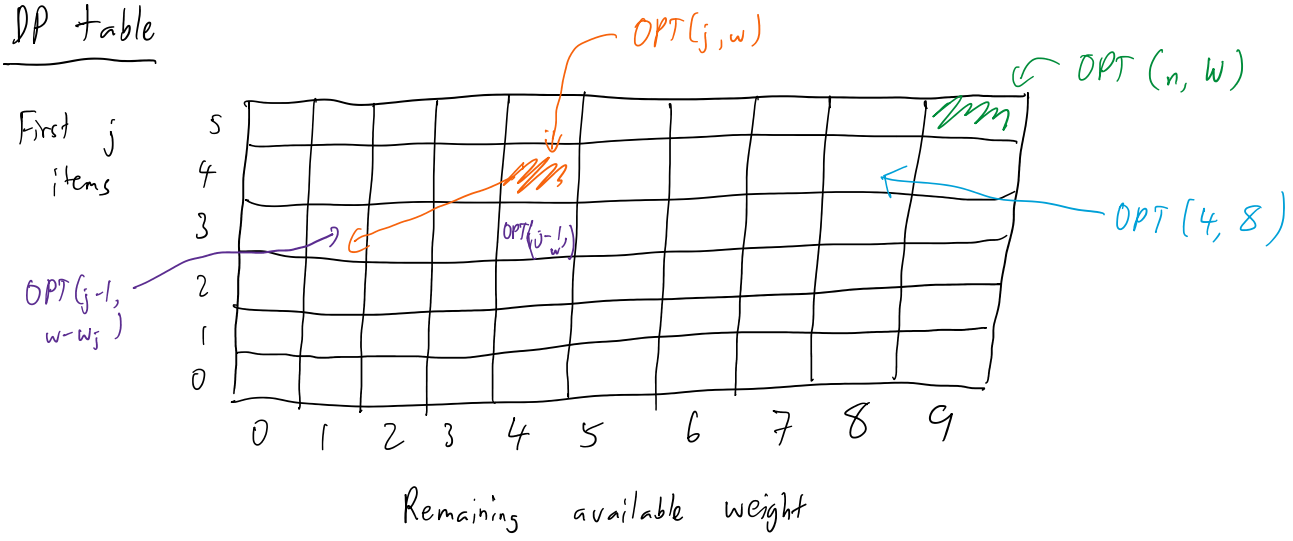
unknown at computation time, so try both using max

Base case:

$OPT(0, W) = 0$ if no items, 0

$OPT(j, 0) = 0$ if no space, 0

DP table



Have to look at two entries in the row below

Pseudocode:

Subset Sum (n, W):

$M[0, w] = 0 \quad \forall w \in 0, \ldots, W$

$M[j, 0] = 0 \quad \forall j \in 1, \ldots, n$

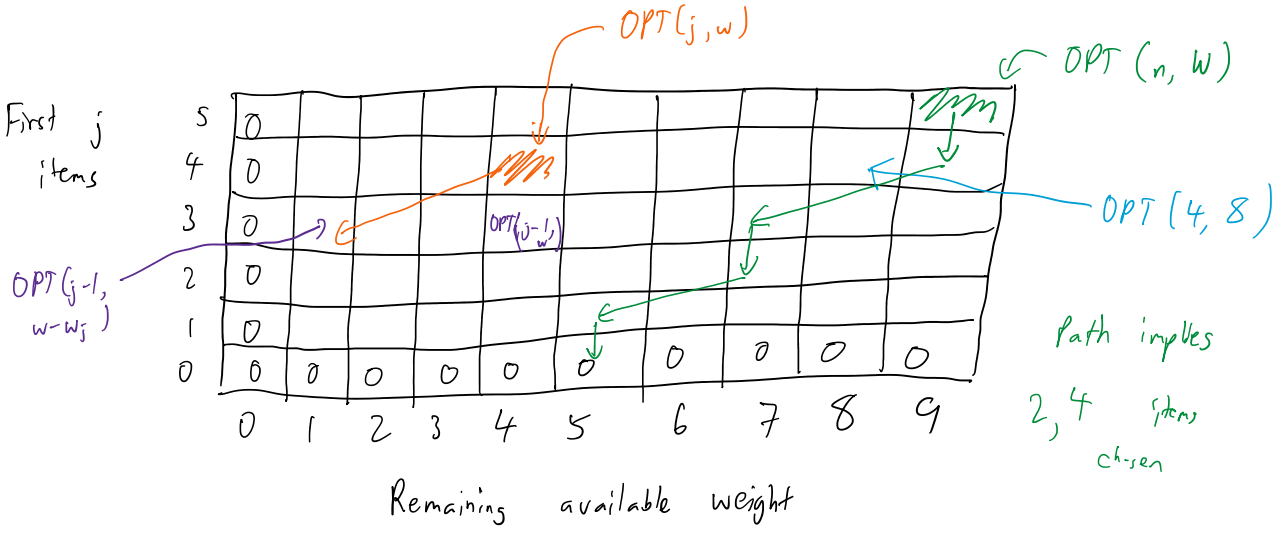
Define $M[j, r] = 0 \quad \forall r < 0$

For $j \in 1, \ldots, n$:

For $w \in 0, \ldots, W$:

$$M[j, w] = \max \begin{cases} M[j-1, w] \\ w_j + M[j-1, w - w_j] \end{cases}$$

} also keep track of which entry was the max so we can reconstruct S^* at the end.



nW cells in matrix

Each cell takes $O(1)$ time to fill.

$O(n)$ time time to follow path backwards

$\Rightarrow O(nW + n) = O(nW)$ total running time.

General principles:

- Opt can be easily computed from subproblems
- Only poly number of subproblems.
- "Natural" ordering of problems from small to large, such that to answer large problem, need only small sols.

Knapsack problem

- A bound W
- collection of n items, each with weight w_i, and a value v_i.

Find a subset S that

maximizes $\sum_{i \in S} v_i$ while keeping $\sum_{i \in S} w_i \leq W$.

want to maximize value instead of weight, as in subset sum.

Greedy doesn't work

Idea Sort by value per weight $p_i = \frac{v_i}{w_i}$, and pick highest value density.

1 lb bottle of ambrosia,	$30	$p_1 = \$30/lb$
2 lb jade paperweight,	$40	$p_2 = \$20/lb$
3 lb salt,	$45	$p_3 = \$15/lb$
4 lb iron crowbar,	$100	$p_4 = \$25/lb$

6 lb carrying capacity knapsack.

Greedy ambrosia 1 lb, crowbar 4 lb, ~~jade~~ 2 lb Total value: $130

Better crowbar, jade Total value: $140

} problem is can't divide objects

DP solution: $OPT(i, W) = \max \{ OPT(j-1, W)$, if $j \notin S^*$

DP solution:

$$OPT(j, W) = \max \begin{cases} OPT(j-1, W) & \text{if } j \notin S^* \\ v_j + OPT(j-1, w - w_j) & \text{if } j \in S^* \end{cases}$$