20. Dynamic Programming II

Subset sum problem, knapsack problem, greedy algorithm vs dynamic programming [Ottman/Widmayer, Kap. 7.2, 7.3, 5.7, Cormen et al, Kap. 15,35.5]

Task

Partition the set of the “item” above into two set such that both sets have the same value.

A solution:

Subset Sum Problem

Consider $n \in \mathbb{N}$ numbers $a_1, \dots, a_n \in \mathbb{N}$.

Goal: decide if a selection $I \subseteq \{1, \dots, n\}$ exists such that

$$\sum_{i \in I} a_i = \sum_{i \in \{1,\dots,n\} \setminus I} a_i.$$

Naive Algorithm

Check for each bit vector $b = (b_1, \dots, b_n) \in \{0,1\}^n$, if

$$\sum_{i=1}^{n} b_i a_i \stackrel{?}{=} \sum_{i=1}^{n} (1 - b_i) a_i$$

Worst case: n steps for each of the 2^n bit vectors b. Number of steps: $\mathcal{O}(n \cdot 2^n)$.

Algorithm with Partition

- Partition the input into two equally sized parts $a_1, \dots, a_{n/2}$ and $a_{n/2+1}, \dots, a_n$.
- Iterate over all subsets of the two parts and compute partial sum $S_1^k, \dots, S_{2^{n/2}}^k$ ($k = 1, 2$).
- Sort the partial sums: $S_1^k \leq S_2^k \leq \dots \leq S_{2^{n/2}}^k$.
- Check if there are partial sums such that $S_i^1 + S_j^2 = \frac{1}{2}\sum_{i=1}^n a_i =: h$
  - Start with $i = 1, j = 2^{n/2}$.
  - If $S_i^1 + S_j^2 = h$ then finished
  - If $S_i^1 + S_j^2 > h$ then $j \leftarrow j - 1$
  - If $S_i^1 + S_j^2 < h$ then $i \leftarrow i + 1$

Example

Set $\{1, 6, 2, 3, 4\}$ with value sum 16 has 32 subsets.

Partitioning into $\{1, 6\}$, $\{2, 3, 4\}$ yields the following 12 subsets with value sums:

$\{1,6\}$				$\{2,3,4\}$							
$\{\}$	$\{1\}$	$\{6\}$	$\{1,6\}$	$\{\}$	$\{2\}$	$\{3\}$	$\{4\}$	$\{2,3\}$	$\{2,4\}$	$\{3,4\}$	$\{2,3,4\}$
0	$\boxed{1}$	6	7	0	2	3	4	5	6	$\boxed{7}$	9

$\Leftrightarrow$ One possible solution: $\{1, 3, 4\}$

Analysis

- Generate partial sums for each part: $\mathcal{O}(2^{n/2} \cdot n)$.
- Each sorting: $\mathcal{O}(2^{n/2} \log(2^{n/2})) = \mathcal{O}(n 2^{n/2})$.
- Merge: $\mathcal{O}(2^{n/2})$

Overal running time

$$\mathcal{O}\left(n \cdot 2^{n/2}\right) = \mathcal{O}\left(n \left(\sqrt{2}\right)^n\right).$$

Substantial improvement over the naive method – but still exponential!

Dynamic programming

Task: let $z = \frac{1}{2}\sum_{i=1}^n a_i$. Find a selection $I \subset \{1, \dots, n\}$, such that $\sum_{i \in I} a_i = z$.

DP-table: $[0, \dots, n] \times [0, \dots, z]$-table T with boolean entries. $T[k, s]$ specifies if there is a selection $I_k \subset \{1, \dots, k\}$ such that $\sum_{i \in I_k} a_i = s$.

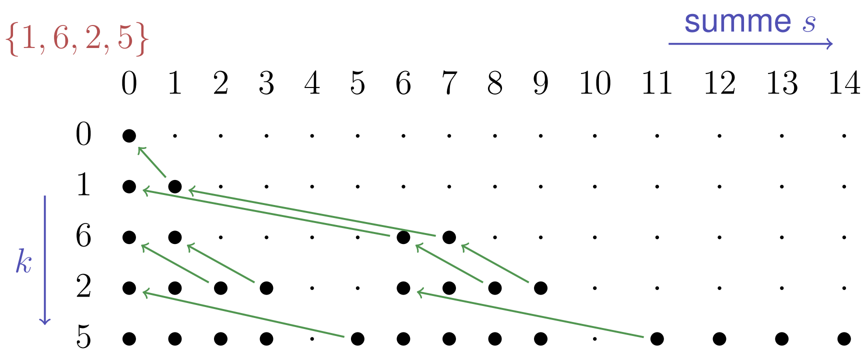
Initialization: $T[0, 0]$ = true. $T[0, s]$ = false for $s > 1$.

Computation:

$$T[k, s] \leftarrow \begin{cases} T[k-1, s] & \text{if } s < a_k \\ T[k-1, s] \vee T[k-1, s-a_k] & \text{if } s \geq a_k \end{cases}$$

for increasing k and then within k increasing s.

Example



Determination of the solution: if $T[k, s] = T[k-1, s]$ then a_k unused and continue with $T[k-1, s]$, otherwise a_k used and continue with $T[k-1, s-a_k]$.

That is mysterious

The algorithm requires a number of $\mathcal{O}(n \cdot z)$ fundamental operations.

What is going on now? Does the algorithm suddenly have polynomial running time?

Explained

The algorithm does not necessarily provide a polynomial run time. z is an *number* and not a *quantity*!

Input length of the algorithm $\cong$ number bits to *reasonably* represent the data. With the number z this would be $\zeta = \log z$.

Consequently the algorithm requires $\mathcal{O}(n \cdot 2^\zeta)$ fundamental operations and has a run time exponential in ζ.

If, however, z is polynomial in n then the algorithm has polynomial run time in n. This is called *pseudo-polynomial*.

NP

It is known that the subset-sum algorithm belongs to the class of *NP*-complete problems (and is thus *NP-hard*).

P: Set of all problems that can be solved in polynomial time.

NP: Set of all problems that can be solved Nondeterministically in Polynomial time.

Implications:

- NP contains P.
- Problems can be *verified* in polynomial time.
- Under the not (yet?) proven assumption[41] that NP $\neq$ P, there is *no algorithm with polynomial run time* for the problem considered above.

[41] The most important unsolved question of theoretical computer science.

The knapsack problem

We pack our suitcase with ...

- toothbrush
- dumbell set
- coffee machine
- uh oh – too heavy.

- Toothbrush
- Air balloon
- Pocket knife
- identity card
- dumbell set
- Uh oh – too heavy.

- toothbrush
- coffe machine
- pocket knife
- identity card
- Uh oh – too heavy.

Aim to take as much as possible with us. But some things are more valuable than others!

Knapsack problem

Given:

- set of $n \in \mathbb{N}$ items $\{1, \dots, n\}$.
- Each item i has value $v_i \in \mathbb{N}$ and weight $w_i \in \mathbb{N}$.
- Maximum weight $W \in \mathbb{N}$.
- Input is denoted as $E = (v_i, w_i)_{i=1,\dots,n}$.

Wanted:

a selection $I \subseteq \{1, \dots, n\}$ that maximises $\sum_{i \in I} v_i$ under $\sum_{i \in I} w_i \leq W$.

Greedy heuristics

Sort the items decreasingly by value per weight v_i/w_i: Permutation p with $v_{p_i}/w_{p_i} \geq v_{p_{i+1}}/w_{p_{i+1}}$

Add items in this order ($I \leftarrow I \cup \{p_i\}$), if the maximum weight is not exceeded.

That is fast: $\Theta(n \log n)$ for sorting and $\Theta(n)$ for the selection. But is it good?

Counterexample

$$\begin{array}{lll} v_1 = 1 & w_1 = 1 & v_1/w_1 = 1 \\ v_2 = W - 1 & w_2 = W & v_2/w_2 = \frac{W-1}{W} \end{array}$$

Greed algorithm chooses $\{v_1\}$ with value 1.
Best selection: $\{v_2\}$ with value $W - 1$ and weight W.

Greedy heuristics can be arbitrarily bad.

Dynamic Programming

Partition the maximum weight.

Three dimensional table $m[i, w, v]$ ("doable") of boolean values.

$m[i, w, v] =$ true if and only if

- A selection of the first i parts exists ($0 \le i \le n$)
- with overal weight w ($0 \le w \le W$) and
- a value of at least v ($0 \le v \le \sum_{i=1}^{n} v_i$) .

Computation of the DP table

Initially

- $m[i, w, 0] \leftarrow$ true für alle $i \ge 0$ und alle $w \ge 0$.
- $m[0, w, v] \leftarrow$ false für alle $w \ge 0$ und alle $v > 0$.

Computation

$$m[i, w, v] \leftarrow \begin{cases} m[i-1, w, v] \vee m[i-1, w-w_i, v-v_i] & \text{if } w \ge w_i \text{ und } v \ge v_i \\ m[i-1, w, v] & \text{otherwise.} \end{cases}$$

increasing in i and for each i increasing in w and for fixed i and w increasing by v.

Solution: largest v, such that $m[i, w, v] =$ true for some i and w.

Observation

The definition of the problem obviously implies that

- for $m[i, w, v] =$ true it holds:
  $m[i', w, v] =$ true $\forall i' \ge i$,
  $m[i, w', v] =$ true $\forall w' \ge w$,
  $m[i, w, v'] =$ true $\forall v' \le v$.
- fpr $m[i, w, v] =$ false it holds:
  $m[i', w, v] =$ false $\forall i' \le i$,
  $m[i, w', v] =$ false $\forall w' \le w$,
  $m[i, w, v'] =$ false $\forall v' \ge v$.

This strongly suggests that we do not need a 3d table!

2d DP table

Table entry $t[i, w]$ contains, instead of boolean values, the largest v, that can be achieved[42] with

- items $1, \dots, i$ ($0 \le i \le n$)
- at maximum weight w ($0 \le w \le W$).

[42] We could have followed a similar idea in order to reduce the size of the sparse table.

Computation

Initially

- $t[0, w] \leftarrow 0$ for all $w \geq 0$.

We compute

$$t[i, w] \leftarrow \begin{cases} t[i-1, w] & \text{if } w < w_i \\ \max\{t[i-1, w], t[i-1, w-w_i] + v_i\} & \text{otherwise.} \end{cases}$$

increasing by i and for fixed i increasing by w.

Solution is located in $t[n, w]$

Example

$E = \{(2,3), (4,5), (1,1)\}$

$i \downarrow$ \ $w \rightarrow$	0	1	2	3	4	5	6	7
$\emptyset$	0	0	0	0	0	0	0	0
$(2,3)$	0	0	3	3	3	3	3	3
$(4,5)$	0	0	3	3	5	5	8	8
$(1,1)$	0	1	3	4	5	6	8	9

Reading out the solution: if $t[i, w] = t[i-1, w]$ then item i unused and continue with $t[i-1, w]$ otherwise used and continue with $t[i-1, s-w_i]$.

Analysis

The two algorithms for the knapsack problem provide a run time in $\Theta(n \cdot W \cdot \sum_{i=1}^{n} v_i)$ (3d-table) and $\Theta(n \cdot W)$ (2d-table) and are thus both pseudo-polynomial, but they deliver the best possible result.

The greedy algorithm is very fast butmight deliver an arbitrarily bad result.

Now we consider a solution between the two extremes.

21. Dynamic Programming III

FPTAS [Ottman/Widmayer, Kap. 7.2, 7.3, Cormen et al, Kap. 15,35.5]

Approximation

Let $\varepsilon \in (0,1)$ given. Let I_{opt} an optimal selection.

No try to find a valid selection I with

$$\sum_{i \in I} v_i \geq (1-\varepsilon) \sum_{i \in I_{\text{opt}}} v_i.$$

Sum of weights may not violate the weight limit.

Different formulation of the algorithm

Before: weight limit $w \to$ maximal value v

Reversed: value $v \to$ minimal weight w

$\Rightarrow$ **alternative table** $g[i,v]$ provides the minimum weight with

- a selection of the first i items ($0 \leq i \leq n$) that
- provide a value of exactly v ($0 \leq v \leq \sum_{i=1}^n v_i$).

Computation

Initially

- $g[0,0] \leftarrow 0$
- $g[0,v] \leftarrow \infty$ (Value v cannot be achieved with 0 items.).

Computation

$$g[i,v] \leftarrow \begin{cases} g[i-1,v] & \text{falls } v < v_i \\ \min\{g[i-1,v], g[i-1,v-v_i]+w_i\} & \text{sonst.} \end{cases}$$

incrementally in i and for fixed i increasing in v.

Solution can be found at largest index v with $g[n,v] \leq w$.

Example

$E = \{(2,3),(4,5),(1,1)\}$

$v \rightarrow$

$i \downarrow$	0	1	2	3	4	5	6	7	8	9
$\emptyset$	0	∞	∞	∞	∞	∞	∞	∞	∞	∞
$(2,3)$	0	∞	∞	2	∞	∞	∞	∞	∞	∞
$(4,5)$	0	∞	∞	2	∞	4	∞	∞	6	∞
$(1,1)$	0	1	∞	2	3	4	5	∞	6	7

Read out the solution: if $g[i,v] = g[i-1,v]$ then item i unused and continue with $g[i-1,v]$ otherwise used and continue with $g[i-1, b-v_i]$.

The approximation trick

Pseduopolynomial run time gets polynmial if the number of occuring values can be bounded by a polynom of the input length.

Let $K > 0$ be chosen *appropriately*. Replace values v_i by "rounded values" $\tilde{v}_i = \lfloor v_i/K \rfloor$ delivering a new input $E' = (w_i, \tilde{v}_i)_{i=1\dots n}$.

Apply the algorithm on the input E' with the same weight limit W.

Idea

Example $K = 5$

Values

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 10, \dots, 98, 99, 100$$
$$\to$$
$$0, 0, 0, 0, 1, 1, 1, 1, 1, 2, \dots, 19, 19, 20$$

Obviously less different values

Properties of the new algorithm

- Selection of items in E' is also admissible in E. Weight remains unchanged!
- Run time of the algorithm is bounded by $\mathcal{O}(n^2 \cdot v_{\max}/K)$ ($v_{\max} := \max\{v_i | 1 \le i \le n\}$)

How good is the approximation?

It holds that

$$v_i - K \le K \cdot \left\lfloor \frac{v_i}{K} \right\rfloor = K \cdot \tilde{v}_i \le v_i$$

Let I'_{opt} be an optimal solution of E'. Then

$$\left(\sum_{i \in I_{\text{opt}}} v_i \right) - n \cdot K \overset{|I_{\text{opt}}| \le n}{\le} \sum_{i \in I_{\text{opt}}} (v_i - K) \le \sum_{i \in I_{\text{opt}}} (K \cdot \tilde{v}_i) = K \sum_{i \in I_{\text{opt}}} \tilde{v}_i$$
$$\underset{I'_{\text{opt}} \text{optimal}}{\le} K \sum_{i \in I'_{\text{opt}}} \tilde{v}_i = \sum_{i \in I'_{\text{opt}}} K \cdot \tilde{v}_i \le \sum_{i \in I'_{\text{opt}}} v_i.$$

Choice of K

Requirement:

$$\sum_{i \in I'} v_i \geq (1-\varepsilon) \sum_{i \in I_{\text{opt}}} v_i.$$

Inequality from above:

$$\sum_{i \in I'_{\text{opt}}} v_i \geq \left(\sum_{i \in I_{\text{opt}}} v_i \right) - n \cdot K$$

thus: $K = \varepsilon \frac{\sum_{i \in I_{\text{opt}}} v_i}{n}$.

Choice of K

Choose $K = \varepsilon \frac{\sum_{i \in I_{\text{opt}}} v_i}{n}$. The optimal sum is unknown. Therefore we choose $K' = \varepsilon \frac{v_{\max}}{n}$.[43]

It holds that $v_{\max} \leq \sum_{i \in I_{\text{opt}}} v_i$ and thus $K' \leq K$ and the approximation is even slightly better.

The run time of the algorithm is bounded by

$$\mathcal{O}(n^2 \cdot v_{\max}/K') = \mathcal{O}(n^2 \cdot v_{\max}/(\varepsilon \cdot v_{\max}/n)) = \mathcal{O}(n^3/\varepsilon).$$

[43] We can assume that items i with $w_i > W$ have been removed in the first place.

FPTAS

Such a family of algorithms is called an *approximation scheme*: the choice of ε controls both running time and approximation quality.

The runtime $\mathcal{O}(n^3/\varepsilon)$ is a polynom in n and in $\frac{1}{\varepsilon}$. The scheme is therefore also called a *FPTAS - Fully Polynomial Time Approximation Scheme*

21. Dynamic Programming III

Optimal Search Tree [Ottman/Widmayer, Kap. 5.7]

Optimal binary Search Trees

Given: search probabilities p_i for each key k_i ($i = 1, \dots, n$) and q_i of each interval d_i ($i = 0, \dots, n$) between search keys of a binary search tree. $\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$.

Wanted: optimal search tree T with key depths $\mathrm{depth}(\cdot)$, that minimizes the expected search costs

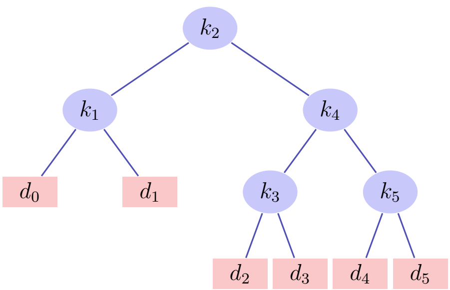
$$C(T) = \sum_{i=1}^{n} p_i \cdot (\mathrm{depth}(k_i) + 1) + \sum_{i=0}^{n} q_i \cdot (\mathrm{depth}(d_i) + 1)$$
$$= 1 + \sum_{i=1}^{n} p_i \cdot \mathrm{depth}(k_i) + \sum_{i=0}^{n} q_i \cdot \mathrm{depth}(d_i)$$

Example

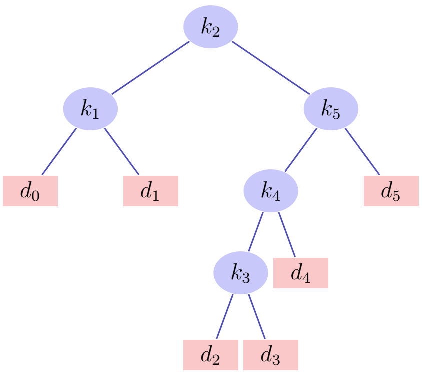
Expected Frequencies

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Example



Search tree with expected costs 2.8



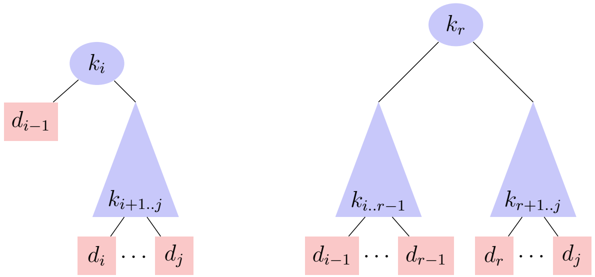
Search tree with expected costs 2.75

Structure of a optimal binary search tree

- Subtree with keys $k_i, \dots, k_j$ and intervals $d_{i-1}, \dots, d_j$ must be optimal for the respective sub-problem.[44]
- Consider all subtrees with roots k_r and optimal subtrees for keys $k_i, \dots, k_{r-1}$ and $k_{r+1}, \dots, k_j$

[44] The usual argument: if it was not optimal, it could be replaced by a better solution improving the overal solution.

Sub-trees for Searching



empty left subtree

non-empty left and right subtrees

empty right subtree

Expected Search Costs

Let $\text{depth}_T(k)$ be the depth of a node k in the sub-tree T. Let k be the root of subtrees T_r and T_{L_r} and T_{R_r} be the left and right sub-tree of T_r. Then

$$\text{depth}_T(k_i) = \text{depth}_{T_{L_r}}(k_i) + 1,\ (i < r)$$
$$\text{depth}_T(k_i) = \text{depth}_{T_{R_r}}(k_i) + 1,\ (i > r)$$

Expected Search Costs

Let $e[i,j]$ be the costs of an optimal search tree with nodes $k_i, \ldots, k_j$.

Base case $e[i, i-1]$, expected costs d_{i-1}

Let $w(i,j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$.

If k_r is the root of an optimal search tree with keys $k_i, \ldots, k_j$, then

$$e[i,j] = p_r + (e[i, r-1] + w(i, r-1)) + (e[r+1, j] + w(r+1, j))$$

with $w(i,j) = w(i, r-1) + p_r + w(r+1, j)$:

$$e[i,j] = e[i, r-1] + e[r+1, j] + w(i,j).$$

Dynamic Programming

$$e[i,j] = \begin{cases} q_{i-1} & \text{if } j = i-1, \\ \min_{i \le r \le j}\{e[i, r-1] + e[r+1, j] + w[i,j]\} & \text{if } i \le j \end{cases}$$

Computation

Tables $e[1 \dots n+1, 0 \dots n], w[1 \dots n+1, 0 \dots m], r[1 \dots n, 1 \dots n]$
Initially

- $e[i, i-1] \leftarrow q_{i-1}$, $w[i, i-1] \leftarrow q_{i-1}$ for all $1 \leq i \leq n+1$.

We compute

$$
\begin{aligned}
w[i, j] &= w[i, j-1] + p_j + q_j \\
e[i, j] &= \min_{i \leq r \leq j} \{e[i, r-1] + e[r+1, j] + w[i, j]\} \\
r[i, j] &= \arg\min_{i \leq r \leq j} \{e[i, r-1] + e[r+1, j] + w[i, j]\}
\end{aligned}
$$

for intervals $[i, j]$ with increasing lengths $l = 1, \dots, n$, each for $i = 1, \dots, n-l+1$. Result in $e[1, n]$, reconstruction via r. Runtime $\Theta(n^3)$.

Example

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

w

j \ i	1	2	3	4	5	6
0	0.05					
1	0.30	0.10				
2	0.45	0.25	0.05			
3	0.55	0.35	0.15	0.05		
4	0.70	0.50	0.30	0.20	0.05	
5	1.00	0.80	0.60	0.50	0.35	0.10

e

j \ i	1	2	3	4	5	6
0	0.05					
1	0.45	0.10				
2	0.90	0.40	0.05			
3	1.25	0.70	0.25	0.05		
4	1.75	1.20	0.60	0.30	0.05	
5	2.75	2.00	1.30	0.90	0.50	0.10

r

j \ i	1	2	3	4	5
1	1				
2	1	2			
3	2	2	3		
4	2	2	4	4	
5	2	4	5	5	5