6. Optimalizační problémy

Algoritmizace

Jiří Balun

Obsah

Rekapitulace a motivace

Na minulé přednášce

- datové stuktury: halda a seznam
- Heap Sort – třídění haldou
- Bucket Sort – přihrádkové třídění

Na této přednášce

- optimalizační problémy a problém batohu
- řešení problému hrubou silou
- hladové algoritmy
- dynamické programování

Optimalizační problémy

Motivace: proč studujeme optimalizační problémy?

Intuice: optimalizační problémy se snaží najít řešení s **nejlepší možnou hodnotou** podle daného kritéria (např. maximum zisku, minimum nákladů, nejkratší cesta, atd.).

Motivační příklad – problém batohu:

- máme množinu předmětů, přičemž každý má svou „hodnotu“ (např. váhu nebo cenu)
- batoh má ale omezenou kapacitu
- chceme vybrat podmnožinu předmětů tak, abychom:

maximalizovali celkovou hodnotu a zároveň **nepřekročili kapacitu batohu**

Problém:

- počet možných řešení roste **exponenciálně**
- pro větší vstupy není prakticky možné prohledat všechny možnosti
- **proto potřebujeme optimalizační algoritmy:** chytré postupy, které najdou nejlepší, nebo alespoň dostatečně dobré, řešení v rozumném čase

Optimalizační problém

Motivace: podobně jako jsme definovali obecný pojem problému, musíme nyní formálně definovat i jeho speciální typ – optimalizační problém.

Definice

Optimalizační problém je dán čtveřicí $(I, sol, cost, goal)$, kde:

- I – množina všech **instancí problému**,
- $sol(x)$ – množina **přípustných řešení** pro danou instanci $x \in I$,
- $cost(x, s)$ – **cenová funkce**, která hodnotí kvalitu řešení $s \in sol(x)$,
- $goal$ – je buď MIN nebo MAX, určuje zda se jedná o minimalizaci či maximalizaci.

- **optimální řešení** je maximální/minimální (dle $goal$) ze všech přípustných řešení
- náš cíl je tedy najít takové řešení s instance x, že $cost(x, s)$ je optimální (nebo se mu co nejvíce přiblížit) mezi všemi $s \in sol(x)$

Formální definice: problém batohu

Intuice: máme množinu předmětů, každý má svou váhu, a vybíráme jejich podmnožinu tak, abychom maximalizovali celkovou váhu a zároveň nepřekročili kapacitu batohu.

Definice – problém batohu (angl. *simple knapsack problem*)

- I je dána instancemi ve tvaru: váhy předmětů $\langle w_1, \ldots, w_n \rangle$ a kapacita batohu $b \in \mathbb{N}$,
- $sol(\langle b, w_1, \ldots, w_n \rangle)$ je množina všech množin $T \subseteq \{1, \ldots, n\}$ indexů vah takových, že platí $\Sigma_{i \in T} w_i \leq b$,
- $cost(\langle b, w_1, \ldots, w_n \rangle, T) = \Sigma_{i \in T} w_i$, tj. cena řešení je součet všech vah zahrnutých předmětů,
- $goal = \text{MAX}$.

- existují různé varianty tohoto problému (např. kde každý předmět má váhu i cenu)
- na tomto problému si demonstrujeme další techniky návrhu algoritmu

Příklad

Mějme váhy předmětů $W = \langle w_1, w_2, w_3 \rangle = \langle 3, 4, 5 \rangle$ a kapacitu batohu $b = 7$. Optimální řešení tohoto problému je množina $T = \{1, 2\}$, protože $w_1 + w_2 = 3 + 4 = 7$.

Řešení problému hrubou silou

Řešení problému hrubou silou (brute-force)

Jak algoritmus funguje:

- uvažujeme všechny možné podmnožiny (výběry položek)
- každá volba podmnožin odpovídá binárnímu vzoru/posloupnosti délky n
- spočítáme cenu daného řešení (v případě problému batohu spočítáme celkovou váhu vybraných položek)
- výsledkem je nejlepší přípustné řešení (z těch, které nepřekročí kapacitu batohu)

Vlastnosti:

- zaručuje nalezení **optimálního řešení**
- prochází 2^n možností, od čehož se odvíjí exponeciální časová složitost – i pro poměrně malé instance problému bývá tento algoritmus prakticky nepoužitelný
- používá se především jako referenční metoda nebo pro velmi malé instance

Generování všech podmnožin

Intuice: množina $S = \{s_0, s_1, \ldots, s_{n-1}\}$ má 2^n podmnožin, přičemž k jejich generování použijeme bitovou reprezentaci čísel $k \in \{0, \ldots, 2^n - 1\}$ (například $6 = 110_2$).

Algorithm 1: generate-subsets

Input: set $S = \{s_0, s_1, \ldots, s_{n-1}\}$
Output: all subsets of S

```
1 A ← index-based array representation of set S
2 for k ← 0 to 2^n − 1 do
3 |   T ← ∅
4 |   for i ← 0 to n − 1 do
5 |   |   if bit i of k is 1 then
6 |   |   |   T ← T ∪ {A[i]}
7 |   output and process T
```

- T na řádku 7 hned zpracujeme, ať neukládáme všech 2^n podmnožin do paměti
- vnější cyklus 2^n iterací, vnořený cyklus n iterací $\Rightarrow$ časová složitost $\Theta(n \cdot 2^n)$

Příklad: generování všech podmnožin

Příklad

Vstup: množina $S = \{a, b, c\}$ obsahující $n = 3$ prvky.

Algoritmus prochází všechna čísla od 0 **do** $2^n - 1$:

k	binárně	vygenerovaná podmnožina T
0	000	$\emptyset$
1	001	$\{c\}$
2	010	$\{b\}$
3	011	$\{b, c\}$
4	100	$\{a\}$
5	101	$\{a, c\}$
6	110	$\{a, b\}$
7	111	$\{a, b, c\}$

- první bit zleva udává přítomnost prvku a, druhý bit udává prvek b, a tak dále...,
- každý binární vzor určuje jednu podmnožinu: **8 vzorů** $\Rightarrow$ **8 podmnožin** (pro $n = 3$).

Brute-force algoritmus: průchod všech řešení

Intuice: projdeme všechny možné podmnožiny položek a vybereme tu, jejíž součet nepřesáhne kapacitu b a zároveň je maximální ze všech platných řešení.

Algorithm 2: brute-force-knapsack

Input: weights $W = \langle w_1, \ldots, w_n \rangle$ and knapsack capacity $b \in \mathbb{N}$
Output: set of indexes T (optimal solution)

```
1 T ← ∅
2 bestSum ← 0
3 for S in generate-subsets({1, ..., n}) do
4     sum ← Σ_{i∈S} w_i
5     if sum ≤ b and sum > bestSum then
6         T ← S
7         bestSum ← sum
8 return T
```

- v cyklu na řádku 3 množina S postupně prochází všechny podmnožiny (množiny indexů $\{1, \ldots, n\}$) generované pomocí generate-subsets

Příklad: řešení batohu hrubou silou

Příklad

Vstup: váhy $W = \langle w_1, w_2, w_3 \rangle = \langle 3, 4, 5 \rangle$ a kapacita batohu $b = 7$.

Všechny podmnožiny položek:

Množina S	Součet
$\emptyset$	0
$\{1\}$	3
$\{2\}$	4
$\{3\}$	5
$\{1, 2\}$	7
$\{1, 3\}$	8
$\{2, 3\}$	9
$\{1, 2, 3\}$	12

Výstup: z platných řešení vybíráme podmnožinu s největším součtem, což je $T = \{1, 2\}$ se součtem $\sum_{i \in T} w_i = 3 + 4 = 7$.

Hladové algoritmy

Motivace pro hladové algoritmy

Co je to heuristika?

- heuristika je metoda řešení problémů, která používá intuitivní „mentální zkratky“ namísto složitých algoritmů
- i když obvykle **nezaručují optimální řešení** (jedná se o tzv. aproximační algoritmy), často dávají rychlá, jednoduchá a v praxi velmi dobrá řešení

Hladové algoritmy (z angl. *greedy algorithm*)

- mnohé optimalizační problémy (např. batoh, plánování, atd.) jsou obecně výpočetně obtížné – nalezení optimálního řešení může trvat příliš dlouho
- hladový algoritmus je heuristická metoda, která používá jednoduchou lokální volbu: **v každém kroku vyberou „nejlepší“ dostupnou možnost**
- v některých speciálních případech (např. hledání minimální kostry grafu) hladové algoritmy poskytují dokonce optimální řešení daného problémů

Hladový algoritmus

Algorithm 3: greedy-knapsack

Input: weights $W = \langle w_1, w_2, \ldots, w_n \rangle$ and knapsack capacity $b \in \mathbb{N}$
Output: set of indexes T

```
1 sort weights W in descending order (so that w_1 ≥ w_2 ≥ ... ≥ w_n)
2 T ← ∅
3 c ← 0
4 for i ← 0 to n − 1 do
5     if c + w_i ≤ b then
6         T ← T ∪ {i}
7         c ← c + w_i
8 return T
```

- časová složitost algoritmu je $\Theta(n \log n)$, odvíjí se od třídění vah na řádku 1
- lze dokázat, že greedy-knapsack vrátí řešení, jehož cena (součet vah) je nejhůře poloviční v porovnání s optimálním řešením

Příklad: hladový algoritmus

Příklad

Vstup: váhy $W = \langle 8, 5, 6, 2, 4 \rangle$ a kapacita batohu $b = 17$.

Inicializace:

- váhy třídíme sestupně: $W = \langle w_1, w_2, w_3, w_4, w_5 \rangle = \langle 8, 6, 5, 4, 2 \rangle$,
- nastavíme $T \leftarrow \emptyset, \quad c \leftarrow 0$.

Průběh cyklu kde testujeme podmínku $c + w_i < b$:

- testujeme $w_1 = 8$: $0 + 8 \leq 17 \Rightarrow$ přidáme do řešení, $T \leftarrow \{1\}, \ c \leftarrow 8$,
- testujeme $w_2 = 6$: $8 + 6 = 14 < 17 \Rightarrow$ přidáme do řešení, $T \leftarrow \{1, 2\}, \ c \leftarrow 14$,
- testujeme $w_3 = 5$: $14 + 5 = 19 > 17 \Rightarrow$ přeskočíme,
- testujeme $w_4 = 4$: $14 + 4 = 18 > 17 \Rightarrow$ přeskočíme,
- testujeme $w_5 = 2$: $14 + 2 = 16 < 17 \Rightarrow$ přidáme do řešení, $T \leftarrow \{1, 2, 5\}, \ c \leftarrow 16$.

Výstup: $T = \{1, 2, 5\}$ s celkovou váhou 16 (ale existuje lepší, optimální řešení $T' = \{1, 3, 4\}$ s celkovou váhou 17, které algoritmus nenajde).

Dynamické programování

Motivace pro dynamické programování

Pozorování: naivní přístup k řešení některých problémů vede k opakovanému řešení stejných podproblémů – a tedy k exponenciální době běhu.

Kdy jednoduché techniky nestačí:

- hladové algoritmy často selžou – nalezené řešení nemusí být globální optimum
- backtracking (užitečná rekurzivní technika, kterou více rozebírat nebudeme) najde optimální řešení, ale může vyžadovat exponenciální čas
- řešení hrubou silou řeší mnoho překrývajících se podproblémů

Hlavní princip dynamického programování:

- výpočet rozdělíme na menší podproblémy
- každý podproblém vyřešíme **pouze jednou** a **řešení si zapamatujeme**
- složitý problém pak řešíme skládáním známých dílčích výsledků

Dvourozměrné pole jako tabulka

Intuice: dvourozměrné pole si můžeme představit jako **tabulku**, kde každý prvek má **řádek** (index i) a **sloupec** (index j).

Vlastnosti:

- přístup k libovolnému prvku $M[i, j]$ je v konstantním čase, tj. $\Theta(1)$,
- každý řádek může nést informace pro jeden „stav“ algoritmu,
- každý sloupec představuje jinou hodnotu parametru (např. kapacity batohu).

Příklad

Dvourozměrné pole $M[0..3, 0..4]$ použité v následujícím algoritmu, kde řádek reprezentuje použití prvních i předmětů a každý sloupec udává kapacitu batohu j.

$M[i,j]$	0	1	2	3	4
0	0	0	0	0	0
1	0	1	1	1	1
2	0	1	2	2	3
3	0	1	2	3	3

Budování tabulky M

Intuice: postupně počítáme nejlepší dosažitelnou váhu z prvních i předmětů při kapacitě batohu c, a přitom budujeme dvourozměrnou tabulku $M[0\,..\,n, 0\,..\,b]$.

Co reprezentuje $M[i, c]$:

- nejlepší dosažitelná celková váha pomocí prvních i položek
- přičemž máme k dispozici (omezenou) kapacitu c
- řešíme prefixové instance: *buď prvek i přidáme do částečného řešení, nebo ne*

Jak tabulku vyplňujeme:

- základní případy: $M[0, c] = 0$ (žádné položky), $M[i, 0] = 0$ (kapacita je nulová)
- pro $i \geq 1$ a $c \geq 1$:

$$M[i, c] = \begin{cases} M[i-1, c], & c < w_i \\ \max\{M[i-1, c],\ M[i-1, c-w_i] + w_i\}, & c \geq w_i \end{cases}$$

- pokud $c < w_i$, pak w_i nelze přidat do řešení, protože je větší než dostupná kapacita c
- v opačném případě tedy volíme lepší možnost: **použít nebo nepoužít** w_i

Algorithm 4: initialize-knapsack-table

Input: weights $W = \langle w_1, w_2, \ldots, w_n \rangle$ and knapsack capacity $b \in \mathbb{N}$
Output: two-dimensional array M

```
1  initialize new two-dimensional array M[0..n, 0..b]
2  for i ← 0 to n do
3      M[i, 0] ← 0
4  for c ← 0 to b do
5      M[0, c] ← 0
6  for i ← 1 to n do
7      for c ← 1 to b do
8          if c < w_i then
9              M[i, c] ← M[i − 1, c]
10         else
11             M[i, c] ← max(M[i − 1, c], M[i − 1, c − w_i] + w_i)
12 return M
```

Rekonstrukce řešení z tabulky M

Cíl: tabulka M po inicializaci obsahuje největší dosažitelnou váhu, ale pořád potřebujeme najít samotnou množinu vybraných předmětů (nejen jejich celkovou váhu).

Postup (zpětný průchod tabulkou):

- začneme v pravém dolním rohu tabulky: pozice $M[n, b]$
- pro každý index $i = n, n-1, \ldots, 1$:
  - zjišťujeme, zda jsme použili w_i
  - porovnáváme:
$$M[i, c] \stackrel{?}{=} M[i-1, c-w_i] + w_i$$
  - pokud rovnost platí, položka i byla zahrnuta v optimálním řešení
  - když předmět zahrneme, snížíme i dostupnou kapacitu: $c \leftarrow c - w_i$

Rekonstrukce řešení z tabulky M

Intuice: provádíme „zpětné inženýrství“ použitých hodnot w_i z jejich celkového součtu.

Algorithm 5: dp-knapsack

Input: weights $W = \langle w_1, w_2, \ldots, w_n \rangle$ and knapsack capacity $b \in \mathbb{N}$
Output: set of indexes T (optimal solution)

```
1 M ← initialize-knapsack-table(W, b)
2 T ← ∅
3 c ← b
4 for i ← n downto 1 do
5     if c − w_i ≥ 0 and M[i, c] = M[i − 1, c − w_i] + w_i then
6         T ← T ∪ {i}
7         c ← c − w_i
8 return T
```

- v podmínce na řádku 5 testujeme, která volba (použít/nepoužít w_i) vedla k optimální hodnotě v tabulce

Příklad: budování tabulky M

Příklad (část 1/2)

Vstup:

$$W = \langle w_1, w_2, w_3, w_4 \rangle = \langle 3, 4, 6, 5 \rangle, \quad b = 8$$

Inicializace:

- tabulka M má rozměry $(n+1) \times (b+1)$, tj. $M[0..4,\ 0..8]$:,
- první řádek $M[0, c] = 0$ (bez položek),
- první sloupec $M[i, 0] = 0$ (kapacita 0),
- další hodnoty dopočítáváme po řádcích, zleva do prava.

První dva řádky:

$i \backslash c$	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1 ($w_1 = 3$)	0	0	0	3	3	3	3	3	3

Logika: pokud $c < w_i$, nemůžeme položku přidat.

Příklad (část 2/2)

Celá tabulka M pro $W = \langle 3, 4, 6, 5 \rangle$, $b = 8$:

$i \backslash c$	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1 ($w_1 = 3$)	0	0	0	3	3	3	3	3	3
2 ($w_2 = 4$)	0	0	0	3	4	4	4	7	7
3 ($w_3 = 6$)	0	0	0	3	4	4	6	7	7
4 ($w_4 = 5$)	0	0	0	3	4	5	6	7	8

Příklad výpočtu konkrétních buněk:

- pro $i = 1, c = 2$ platí $c < w_1$ a proto $M[1, 2] = M[0, 2] = 0$,
- pro $i = 1, c = 3$ přidáme větší z $M[0, 3] = 0$ a $M[0, 0] + 3 = 3$, a tedy $M[1, 3] = 3$,
- pro $i = 2, c = 8$ přidáme větší z $M[1, 8] = 3$ a $M[1, 4] + 4 = 7$, a tedy $M[2, 8] = 7$,
- pro $i = 3, c = 4$ platí $c < w_3$ a proto $M[3, 4] = M[2, 4] = 4$,
- pro $i = 4, c = 8$ přidáme větší z $M[3, 8] = 7$ a $M[3, 3] + 5 = 8$, a tedy $M[4, 8] = 8$.

Výsledek tabulky: maximální váha, kterou lze do batohu pobrat, je $M[4, 8] = 8$.

Příklad: rekonstrukce řešení z tabulky

Příklad

Procházíme inicializovanou tabulku M od pravého dolního rohu, tj. $i = 4,\ c = 8$:

Krok 1: pro $i = 4,\ c = 8$ porovnáváme váhu $w_4 = 5$ (platí $c - w_4 \geq 0$):

$$M[4, 8] = 8 \quad \text{a} \quad M[3, 3] + 5 = 3 + 5 = 8$$

$\Rightarrow$ **přidáme** index 4 do řešení (pro položku $w_4 = 5$). Nová kapacita: $c = 8 - 5 = 3$.

Krok 2: pro $i = 3$ neplatí $c - w_3 \geq 0$, tj. $w_3 = 6$ se nevejde $\Rightarrow$ **nepřidáme** index 3,

Krok 3: pro $i = 2$ neplatí $c - w_2 \geq 0$, tj. $w_2 = 4$ se nevejde $\Rightarrow$ **nepřidáme** index 2,

Krok 4: pro $i = 1,\ c = 3$ porovnáváme váhu $w_1 = 3$ (platí $c - w_1 \geq 0$):

$$M[1, 3] = 3 \quad \text{a} \quad M[0, 0] + 3 = 0 + 3 = 3$$

$\Rightarrow$ **přidáme** index 1 do řešení (pro položku $w_1 = 3$). Nová kapacita: $c = 3 - 3 = 0$.

Výsledek: množina optimálního řešení $T = \{1, 4\}$ s celkovou váhou $w_1 + w_4 = 3 + 5 = 8$.

Časová složitost dynamického algoritmu

Intuice: jedná se o *pseudopolynomiální* algoritmus: je rychlý, pokud je kapacita b malá.

Analýza výpočtu tabulky:

- tabulka M má $(n+1) \times (b+1)$ položek
- pro každou dvojici (i, c) provádíme konstantní počet operací: porovnání, sčítání, atd.
- celkový čas potřebný na vyplnění tabulky: $\Theta(n \cdot b)$

Analýza celého algoritmu:

- při rekonstrukci řešení procházíme tabulku zpět po řádcích $i = n, n-1, \ldots, 0$
- v každém kroku provedeme jen konstanní počet operací: $\Theta(n)$
- asymptoticky dominuje čas na vyplnění tabulky
- **celková časová složitost:** $T(n, b) = \Theta(n \cdot b)$

Pseudopolynomiální čas

Intuice: některé algoritmy nejsou polynomiální v obvyklém smyslu, ale jsou efektivní, pokud jsou číselné hodnoty vstupu malé.

Zajímavé pozorování:

- uvažujme instanci problému, kde vynásobíme každou váhu w_i a kapacitu b číslem 100
- dp-knapsack bude tuto instanci řešit 100-krát délší dobu, než před vynásobením

Kdy říkáme, že algoritmus běží v pseudopolynomiálním čase?

- jeho časová složitost je polynomiální v „aritmetické velikosti“ vstupu (např. kapacitě b), nikoliv v délce binárního zápisu vstupu

Proč to není skutečně polynomiální algoritmus?

- kapacita batohu b je číslo, které má binární délku $\Theta(\log b)$
- tedy čas $\Theta(n \cdot b)$ je ve skutečnosti **exponenciální** v délce vstupu ($b = 2^{\Theta(\log b)}$)
- pseudopolynomiální čas závisí přímo na hodnotě čísla, ne na jeho zápisu

Závěrem

Optimalizační problémy:

- optimalizační problém – speciální typ problému, který povoluje více možných řešení, přičemž našim úkolem je najít to nejlepší z nich
- problém batohu: maximalizujeme hodnotu vybraných předmětů do daného limitu

Techniky řešení problému batohu:

- řešení problému hrubou silou
  - najde optimální řešení, ale má superexponenciální časovou složitost $\Theta(n \cdot 2^n)$
  - ideální jako refenrenční řešení
- hladové algoritmy
  - aproximační algoritmus – obecně nevrací optimální řešení
  - výsledné řešení může mít v nejhorším případě poloviční cenu optimálního řešení
  - časová složitost $\Theta(n \log n)$
- dynamické programování
  - chytrá technika, která si předpočítá mezivýsledky, ze kterých poté skládá finální řešení
  - pseudopolynomiální časová složitost $\Theta(n \cdot b)$