Greedy algoritmy

Abstrakt

Tyto poznámky se zabývají technikou návrhu greedy metodou rozděl a panuj. Obsahují jak obecný popis techniky, tak i příklady konkrétních algoritmů.

1 Základní princip

Technika návrhu algoritmů greedy technikou (český překlad je žravou technikou) je použitelná pro algoritmy řešící optimalizační problémy. Algoritmus navržený touto technikou prochází při svém běhu sérií voleb, při každé z nichž vybírá tu možnost, která je v momentálně nejlepší (formulace toho, co znamená nejlepší záleží na konkrétním problému). Volba není založena na jejích možných důsledcích pro další běh algoritmu, je založena pouze na její ceně v momentě volby. Odtud pochází označení greedy (žravé) algoritmy. Algoritmus bez rozmyslu, žravě, vybírá tu aktuálně nejlepší možnost.

Obecné schéma greedy algoritmu se dá shrnout do následujících kroků:

1. Pro vstupní instanci I algoritmus provede volbu x a na jejím základě vytvoří *jednu* menší podinstanci I' . (Volba x je v tomto popisu abstraktní pojem, v reálném algoritmu to je reálný objekt, např. hrana grafu, číslo, množina, posloupnost bitů apod.)
2. Algoritmus poté rekurzivně aplikujeme na I'. Řešení, které získáme pro I' pak zkombinujeme s volbou x z předchozího kroku a obdržíme tak řešení pro I.
3. Rekurze končí v okamžiku, kdy je vstupní instance dostatečně malá.

Protože rekurze ve naznačeném schématu je koncová (a je to tedy v podstatě jenom cyklus), lze algoritmus jednoduše převést do iterativní verze. Algoritmus pak tedy iterativně provádí kroky 1 a 2, přičemž si zapamatuje jednotlivé volby z kroku 1 a zkombinuje je do řešení až po ukončení cyklu (tedy v momentě, kdy už zpracováváme dostatečně malou podinstanci a rekurze by už skončila). Někdy lze jednotlivé volby kombinovat do řešení průběžně (např. pokud je řešením množina prvků a volby v jednotlivých krocích jsou prvky této množiny).

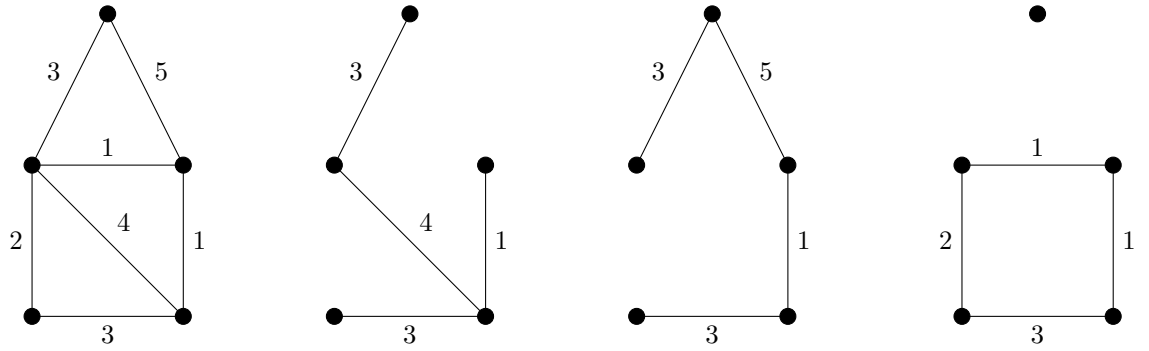
```
1: procedure Greedy(Input)
2:     I ← Input
3:     i ← 0
4:     while |J| ≥ c do                          ▷ Iterujeme dokud je I dostatečně velké
5:         x_i ← Greedy-Choice(I)                ▷ Provedeme volbu x_i
6:         I ← Create-Subinstance(I, x_i)        ▷ Vytvoříme podinstanci a přiřadíme do I
7:         i ← i + 1
8:     end while
9:     x_i ← BasicAlgoritm(I)                    ▷ Řešení malé instance
10:    Zkombinuj volby x_j, (j = 0, ..., i) do řešení x
11:    return x
12: end procedure
```

2 Příklady algoritmů

2.1 Minimální kostra grafu

Definice 1. Nechť $G = (V, E)$ je neorientovaný spojitý graf. *Ohodnocení hran* grafu G je zobrazení $c : E \to \mathbb{Q}$ přiřazující hranám grafu jejich racionální hodnotu, $c(e)$ je pak ohodnocení hrany $e \in E$. Dvojici (G, c) říkáme hranově ohodnocený graf.

Kostra grafu G je podgraf $G' = (V, E')$ (G' má stejnou množinu uzlů jako G!) takový, že



(a) Ohodnocený graf (b) Kostra s cenou 11 (c) Kostra s cenou 12 (d) Neni kostra

Obrázek 1: Příklady pojmů z definice 1.

(a) G' neobsahuje kružnici,

(b) G' je spojitý graf. Pro každou dvojici uzlů u, v platí, že mezi nimi existuje cesta

Cena kostry $G' = (V, E')$ v ohodnoceném grafu je součet ohodnocení jejích hran, tj. $\sum_{e \in E'} c(e)$. □

Na obrázku 1 můžeme vidět konkrétní příklady pojmů z předchozí definice. V příkladu jsou dvě kostry s různými cenami. Vyřešit problém minimální kostry grafu znamená najít takovou kostru, jejíž cena je minimální, tj. takovou, že neexistuje jiná kostra s menší cenou. Následuje formální definice problému:

Minimální kostra grafu	
Instance:	Hranově ohodnocený graf (G, c), kde $G = (V, E)$.
Přípustná řešení:	$sol(G, c) = \{(V, E') \mid (V, E') \text{ je kostra grafu } G\}$
Cena řešení:	$cost((V, E'), G, c) = \sum_{e \in E'} c(e)$
Cíl:	minimum

Pro nalezení minimální kostry existují poměrně přímočaré algoritmy využívající greedy přístup. Ukážeme si jeden z nich, tzv. *Kruskalův algoritmus.* Protože nalezení minimální kostry můžeme chápat jako nalezení množiny hran E' z původního grafu, lze v jednotlivých iteracích kroků 1 a 2 z obecného popisu techniky, vybírat vždy jednu hranu. To znamená, že začneme s $E' = \emptyset$ a v každém kroku do E' jednu hranu přidáme. Při výběru přidané hrany se řídíme jednoduchým pravidlem:

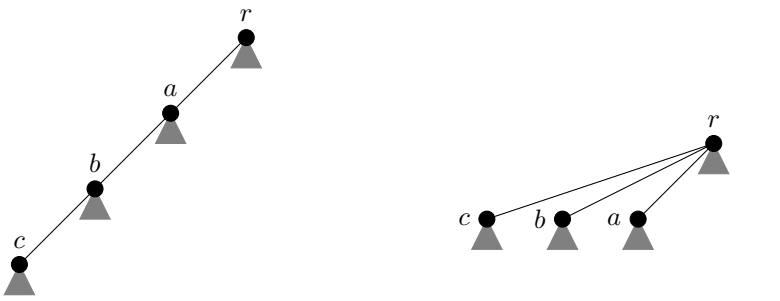
Vyber hranu s nejmenší cenou, jejíž přídání do E' nevytvoří v (V, E') kružnici.

Z původního grafu poté tuto hranu a všechny hrany s menší cenou, jejichž výběr by vedl k vytvoření kružnice, odstraníme. Iterujeme tak dlouho, dokud nenalezneme kostru.

Při implementaci tohoto algoritmu je potřeba nalézt efektivní způsob hledání hrany s minimální cenou a také ověření existence kružnice. První problém lze vyřešit tím, že si na začátku algoritmu hrany vzestupně setřídíme. Efektivně hledat kružnici lze s pomocí vhodné datové struktury pro reprezentaci grafu. Touto strukturou je tzv. *disjoint set structure*. Jak napovídá její název, tato struktura slouží k uložení kolekce $\mathcal{S} = \{S_1, \ldots, S_n\}$ disjunktních množin. Každou z množin $S_1, \ldots, S_n$ lze identifikovat pomocí *reprezentanta*, vybraného prvku z dané množiny. Volba reprentanta je závislá na konkrétním použití struktury, pro účely Kruskalova algoritmu na volbě reprezentanta nezáleží. Jedinou podmínkou je, že pokud strukturu nezměníme nějakou operací, je volba reprezentanta jednoznačná (tzn. pro dva dotazy na reprezentanta dostaneme stejnou opověď).

Nad strukturou lze provádět následující operace.

- MakeSet(x) přidá do systému novou množinu, jejímž jediným prvkem je x.
- Union(x, y) v systému množin sjednotí množinu, která obsahuje prvek x s množinou obsahující prvek y (původní množiny ze systému odstraní a nahradí je jejich sjednocením).



(a) Původní strom (b) Strom po provedení FindSet(c)

Obrázek 2: Heuristika v proceduře FindSet.

- FindSet(x) vrátí reprezentanta množiny obsahující x.

Jednotlivé množiny v systému reprezentujeme pomocí kořenových stromů. V každém uzlu uchováváme jeden prvek, ukazatel *parent* na předka ve stromu (u kořene tento ukazatel ukazuje opět na kořen) a *rank*, což je horní limit výšky podstromu generovaného daným uzlem. V následujícím pseudokódu předpokládáme, že argumenty procedur jsou uzly stromu. Je tedy vhodné udržovat všechny uzly odpovídající prvkům z množin, které jsou v systému, i v jiném struktuře s přímým přístupem, např. v poli. Funkce pro operace se strukturou pak pouze mění ukazatele a ranky.

Algoritmus 1 Operace nad disjoint set structure

```
1: procedure MakeSet(x)
2:     x.parent ← x
3:     x.rank ← 0
4: end procedure

1: procedure FindSet(x)
2:     if x ≠ x.parent then
3:         x.parent ←FindSet(x.parent)
4:     end if
5:     return x.parent
6: end procedure

1: procedure Union(x, y)
2:     r_x ←FindSet(x)
3:     r_y ←FindSet(y)
4:     if r_x.rank > r_y.rank then
5:         r_y.parent ← r_x
6:     else
7:         r_x.parent ← r_y
8:         if r_y.rank = r_x.rank then
9:             r_y.rank = r_y.rank + 1
10:        end if
11:    end if
12: end procedure
```

Z pohledu složitosti je kritická operace FindSet. Hledáme v ní kořen stromu průchodem od uzlu ke kořenu. Složitost této operace tedy závisí na výšce stromu, který reprezentuje množinu (a v nejhorším případě by mohla být lineární). V operacích proto používáme heuristiky, které zajišťují, aby strom nebyl příliš vysoký. První z nich se nachází v proceduře FindSet. Nejdříve řetězem rekurzivních volání najdeme kořen stromu a poté při návratu z rekurze (řádek 3) nastavíme rodiče všech navštívených uzlů na kořen a snížíme tak výšku podstromu, ve kterém se nachází prvek v argumentu FindSet. Druhá heuristika se uplatňuje v proceduře Union. Tato procedura sjednotí dvě množiny tak, že jednoduše připojí kořen stromu reprezentující první množinu jako potomka ke kořenu stromu druhé množiny. Předpokládejme nyní, že chceme spojit stromy s kořeny x a y. Pokud je výška x větší než výška y, pak připojením x jako potomka y způsobí to, že výška výsledného stromu o jedna větší než výška x. Pokud ale kořeny připojíme naopak, bude výškou výsledného stromu výška x. Heuristika tedy spočívá v připojení nižšího stromu jako potomka vyššího. Zajímavé je, že položky rank v jednotlivých uzlech nemusí odpovídat reálné výšce, protože FindSet, která mění výšky některých uzlů, tuto položku vůbec nemění. Ranky jsou tak pouze horním omezením reálné výšky uzlů.

Složitost sekvence m operací se strukturou, z nichž n operací je MakeSet, je $O(m \cdot \alpha(n))$, kde α je *velmi* pomalu rostoucí funkce (asi jako inverzní Ackermanova funkce), která má pro n vyskytující v prakticky představitelných aplikacích hodnotu menší než 4.

Kruskalův algoritmus využívá disjoint set structure pro ukládání množin uzlů grafu. Na začátku algoritmu přidáme do systému pro každý vrchol jednoprvkovou množinu. Vždy, když v průběhu algoritmu přidáme do řešení novou hranu, sjednotíme množiny, které obsahují koncové vrcholy této hrany. Indukcí lze dokázat, že v

libovolném okamžiku obsahuje každá množina právě vrcholy jedné komponenty grafu tvořeného algoritmem zatím vybranými hranami. Protože žádná z těchto komponent neobsahuje kružnici (protože hrany tvořící kružnici do řešení nepřidáváme) a protože komponenty jsou spojité, lze test toho, jestli přidáním nové hrany vznikne kružnice provést jednoduše tak, že otestujeme, jestli jsou oba koncové uzly této hrany ve stejné množině. Pokud ano, přidáním hrany by kružnice vznikla. Po přidání takové hrany by totiž existovali dvě cesty mezi jejími koncovými uzly, první z nich tvořená právě touto hranou, druhá díky tomu, že uzly byli před přidáním ve stejné komponentě.

Nyní si již můžeme uvést pseudokód Kruskalova algoritmu. Pro jednoduchost zápisu předpokládejme, že množina vrcholů V je tvořena $\{0, 1, 2, \dots |V| - 1\}$.

Algoritmus 2 Kruskalův algoritmus

```
1:  procedure KRUSKAL((G = (V, E), c))
2:      Vytvoř prioritní frontu hran Q                ▷ hrany jsou setřízené vzestupně podle ohodnocení
3:      Vytvoř |V| prvkové pole A      ▷ každý prvek obsahuje položky rank a parent, A[i] odpovídá vrcholu i
4:      for i ← 1 to |V| − 1 do
5:          MAKESET(A[i])                              ▷ inicializujeme jednoprvkové komponenty
6:      end for
7:      E' ← ∅
8:      while |E'| < |V| − 1 do                                  ▷ iteruji dokud nemám kostru
9:          Odeber z Q hranu (u, v)
10:         if FINDSET(A[u]) ≠ FINDSET(A[v]) then     ▷ test toho, jestli jsou uzly ve stejné komponentě
11:             E' ← E' ∪ {(u, v)}
12:             UNION(A[u], A[v])                      ▷ spojím komponenty nově přidanou hranou
13:         end if
14:     end while
15:     return (V, E')
16: end procedure
```

Následující věta ukazuje, že KRUSKAL vrací přípustné řešení.

Věta 1. KRUSKAL *vrací kostru grafu.*

Důkaz. Množina E' obsahuje $|V|-1$ hran (řádek 8 algoritmu) a neobsahuje cykly (řádek 10 algoritmu + diskuze v předchozím odstavci). Tvrzení pak plyne ze souvislosti grafu G. □

Nyní si můžeme dokázat optimalitu algoritmu.

Věta 2. KRUSKAL *vrací minimální kostru.*

Důkaz. Dokážeme, že po každém přidání hrany do E' existuje minimální kostra $T = (V, B)$ taková, že obsahuje doposud algoritmem nalezené hrany. Důkaz provedeme indukcí přes velikost E'. Označme si jako E'_i i-prvkovou množinu hran, kterou získáme po přidání i-té hrany, kterou si označíme e_i.

Pro $E'_0 = \emptyset$ je situace triviální, stačí vybrat libovolnou minimální kostru.

Předpokládejme, že tvrzení platí pro E'_{i-1} a $T = (V, B)$ je odpovídající minimální kostra. Pokud $e_i \in B$, je tvrzení triviální. Pokud $e_i \notin B$, pak přidáním e_i do B vytvoříme v T kružnici. Pak ale B obsahuje hranu $e_j \notin E'_i$, která leží na této kružnici (jinak by algoritmus nemohl e_i přidat do E'_{i-1}, v E'_i by vznikla kružnice). Potom $(V, B - \{e_j\} \cup \{e_i\})$ tvoří kostru se stejnou cenou jako T. Stačí si uvědomit, že po přidaní e_i do B leží obě hrany e_i a e_j na kružnici, a tudíž odstraněním jedné z nich dostaneme kostru. Dále platí, že $c(e_i) \leq c(e_j)$, protože v opačném případě by si algoritmus vybral e_j místo e_i. Skutečně, protože $E'_{i-1} \subseteq B$ tak by přidáním e_j do E'_{i-1} nevnikla kružnice (e_j totiž neleží na kružnici ani v T) a algoritmus tedy e_j nemohl v předchozích iteracích vynechat. Současně T je minimální kostra, takže $c(e_j) \leq c(e_i)$, odtud již dostáváme požadovanou rovnost. □

Zamysleme se nad složitostí algoritmu. Vytvoření prioritní fronty má za předpokladu použití třídění porovnáváním složitost $O(|E| \log |E|)$. Řádek 3 se dá provést s lineární složitostí $O(|V|)$. Poté algoritmus provede nejhůře $|V|+3|E|$ operací nad disjoint sets structure, z nichž $|V|$ operací je MAKESET. Pokud budeme považovat $\alpha(|V|)$

za konstantu (viz diskuze ke složitosti operací nad disjoint sets structure), dostáváme složitost $O(|E|)$. Řádky 9 a 11 mají konstatní složitost. Protože u spojitého grafu je $|E| \geq |V| - 1$, můžeme konstatovat, že složitosti dominuje $O(|E| \log |E|)$.

2.2 Sestavení Huffmanova kódu

Huffmanův kód je klíčem k efektivnímu uložení řetězců nad určitou abecedou pomocí sekvence bitů (a tedy efektivní uložení tohoto řetezce v počítači).

Definice 2. Kód nad abecedou Σ je injektivní zobrazení $\gamma : \Sigma \to \{0,1\}^*$. Řekneme, že kód γ je jednoznačný, pokud existuje jednoznačný způsob, kterým lze libovolné slovo $w = w_1 w_2 \dots w_k \in \Sigma^*$ dekódovat z jeho zakódování $\gamma(w) = \gamma(w_1)\gamma(w_2)\dots\gamma(w_k)$. Tato podmínka je ekvivalentní tomu, že každá dvě různá slova mají různé zakódování. Kód se nazývá *blokový*, pokud pro každé dva znaky $a, b \in \Sigma$ platí, že $|\gamma(a)| = |\gamma(b)|$.

Příklad 1. (a) Uvažujme jednoduchou abecedu $\Sigma = \{x, y, z\}$ a kód γ daný $\gamma(x) = 0, \gamma(y) = 1, \gamma(z) = 01$. Je snadno vidět, že kód není blokový. Také není jednoznačný. Uvažme například slovo xyz. Jeho zakódování $\gamma(xyz) = 0101$ lze dekódovat také jako $xyxy$.

(b) ASCII tabulka je příkladem jednoznačného blokového kódu. Každý z 256 možný znaků, které se v tabulce nacházejí má přidělen unikátní sekvenci 8 bitů. Všimněme si, že blokový kód je vždy jednoznačný. □

ASCII tabulka je příkladem široce používaného kódu, který je standardem. Díky tomu je univerzální a textové soubory v ASCII nemusí obsahovat kódovací tabulku. Na druhou stranu je neefektivní (jako každý jiný blokový kód) v tom, že zanedbává frekvenci výskytů znaků v řetezci a tím je zakódování řetezce delší než je nutné. Uvážíme-li například čtyřprvkovou abecedu $\Sigma = \{a, b, c, d\}$, pak lze jednoduše vytvořit blokový kód s délkou zakódování jednoho slova 2 bity. Uvažme řetezec w nad Σ, který obsahuje 100 000 znaků, a znak a v něm má 10 000 výskytů, b má 50 000 výskytů, c má 35 00 výskytů a d má 5 000 výskytů. Pokud zakódujeme w pomocí zmíněného blokového kódu, dostaneme 200000 bitů. Pokud bychom ovšem sestrojili kód, který často vyskytujímu se znaku přiřadí kratší zakódování než málo se vyskytujícím znakům, můžeme w zakódovat s pomocí méně bitů. Například pokud a zakódujeme pomocí 3 bitů, b pomocí 1 bitu, c pomocí 2 bitů, d pomocí 3 bitů. Zakódování w pomocí takového kódování má pak

$$3 \cdot 10000 + 1 \cdot 50000 + 2 \cdot 35000 + 3 \cdot 5000 = 165000 \text{ bitů.}$$

Ušetřili jsme tedy 35000 bitů, což je 17.5 procenta z původní velikosti souboru. Při použití kódu, který není blokový si ovšem musíme dát pozor na jednoznačnost kódu.

Definice 3. Nechť Σ je abeceda. Kód γ je *prefixový*, pokud pro všechna $x, y \in \Sigma$ platí, že $\gamma(x)$ není prefixem $\gamma(y)$.

Věta 3. *Každý prefixový kód je jednoznačný.*

Důkaz. Stačí ukázat, že existuje procedura pro jednoznačné dekódování. Slovo $w = w_1 w_2 \dots w_n$ dekódujeme ze sekvence $\gamma(w) = b_1 \dots b_m$ následujícím postupem.

1. čteme sekvenci zleva doprava
2. když přečteme sekvenci $b_1 \dots b_j$ takovou, že $\gamma(w_1) = b_1 \dots b_j$ pro dekódujeme w_1
3. smažeme $b_1 \dots b_j$ ze sekvence a pokračujeme bodem 1. Iterujeme dokud není sekvence prázdná.

Díky tomu, že je γ prefixový kód, nemůžeme v bodě 1 dekódovat jiný znak než w_1 (a v dalších iteracích $w_2, w_3, \dots$). □

Příklad 2. Uvažujme prefixový kód γ daný následující tabulkou.

Symbol	γ
a	11
b	01
c	001
d	10
e	000

Řetez *cecab* pak kódujeme pomocí 0010000011101. Procedura z předcházejícího důkazu pak provede následující kroky

Krok	$\gamma(cebab)$	dekódovaný znak
1	0010000011101	c
2	0000011101	e
3	0011101	c
4	1101	a
5	01	b

□

Pro $x \in \Sigma$ je *frekvence* f_x znaku x v textu $w \in \Sigma^*$ o n znacích je podíl

$$f_x = \frac{\text{počet výskytů } x}{n}.$$

Všimněme si, že $n \cdot f_x$ je počet výskytů znaku x v textu, a jelikož součet počtů výskytů jednotlivých znaků z textu je n, je suma všech frekvencí znaků vyskytujících se v textu rovna 1. Efektivitu kódu měříme délkou zakódování vstupního řetězce. Protože

$$|\gamma(w)| = \sum_{x \in S} n \cdot f_x \cdot |\gamma(x)| = n \sum_{x \in S} f_x \cdot |\gamma(x)|$$

můžeme vypustit závislost na délce $|w| = n$ a měřit efektivitu γ pomocí průměrné délky zakódování jednoho znaku

$$ABL(\gamma) = \sum_{x \in S} f_x \cdot |\gamma(x)|.$$

Příklad 3. (a) Uvažme prefixový kód daný následující tabulkou

x	$\gamma(x)$	f_x
a	11	.32
b	01	.25
c	001	.20
d	10	.18
e	000	.05

Průměrná dékla slova tohoto kódu je

$$ABL(\gamma) = 0.32 \cdot 2 + 0.25 \cdot 2 + 0.20 \cdot 3 + 0.18 \cdot 2 + 0.05 \cdot 3 = 2.25$$

Oproti blokovému kódu, který by měl délku ABL rovno 3 (proč?), jsme ušetřili 0.75 bitu.

(b) Uvažme prefixový kód daný následující tabulkou

x	$\gamma(x)$	f_x
a	11	.32
b	10	.25
c	01	.20
d	001	.18
e	000	.05

Průměrná dékla slova tohoto kódu je

$$ABL(\gamma) = 0.32 \cdot 2 + 0.25 \cdot 2 + 0.20 \cdot 2 + 0.18 \cdot 3 + 0.05 \cdot 3 = 2.23$$

Oproti blokovému kódu, který by měl délku 3, jsme ušetřili 0.77 bitu. Tento kód je také o 0.02 bitu lepší než ten z předchozího příkladu. □

Definice 4. Nechť Σ je abeceda znaků vyskytujících se v textu s frekvencemi f_x pro $x \in \Sigma$. Řekneme, že prefixový kód γ kódující Σ je *optimální*, jestliže pro všechny ostatní prefixové kódy γ' platí, že $ABL(\gamma) \leq ABL(\gamma')$. Optimální kód také nazýváme Huffmanův kód.

Nalezení Huffmanova kódu je optimalizační problém

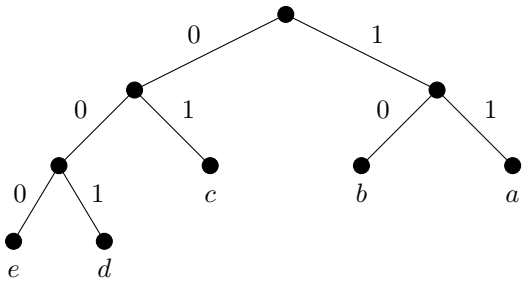
Nalezení Huffmanova kódu	
Instance:	abeceda Σ, frekvence výskytu jednotlivých znaků f_x pro $x \in \Sigma$
Přípustná řešení:	$\{\gamma \mid \gamma$ je prefixový kód pro $\Sigma\}$
Cena řešení:	$cost(\gamma, \Sigma, \{f_x \mid x \in \Sigma\}) = ABL(\gamma)$
Cíl:	minimum

V algoritmu pro nalezení Huffmanova kódu je tento kód reprezentován kořenovým stromem (a tato reprezentace je také výhodnější v důkazech správnosti a optimality algoritmu). Pro prefixový kód γ nad abecedou Σ označíme takový strom T_γ. Tento strom je binární, jeho listy jsou tvořeny prvky Σ. Hrany z nelistových uzlů k levému potomku jsou označeny 0, k pravému potomku 1. T_γ lze setrojit následovně:

1. začínáme s kořenem
2. všechny znaky x, jejichž první bit $\gamma(x)$ je 0 jsou listy v levém podstromu, ostatní (první bit $\gamma(x)$ je 1) jsou listy v pravém podstromu.
3. předchozí krok opakujeme pro levý a pravý podstrom (s množinami znaků rozdělených podle bodu 2), pro $\gamma(x)$ v nichž vynecháme první bit kódu
4. pokud je $\gamma(x)$ prázná sekvence, x je kořenem stromu

Příklad 4. Prefixový kód a jím indukovaný strom.

x	$\gamma(x)$
a	11
b	10
c	01
d	001
e	000



□

Je-li dán T_γ je snadné zpět zpočítat γ. Pro každý $x \in \Sigma$ vypočteme $\gamma(x)$ tak, že najdeme ve stromu T_γ cestu od listu x do kořene. Dostaneme tak sekvenci hran $e_1, \ldots, e_m$ (x má hloubku m), kterou podle označení hran převedeme na sekvenci bitů $b_1, \ldots, b_m$. Převrácením této sekvence obdržíme $\gamma(x)$.

Délka kódového slova $\gamma(x)$ odpovídá v T_γ hloubce listu x (ozn. $depth_T(x)$). Průměrnou délku kódového slova můžeme vyjádřit

$$ABL(T) = \sum_{x \in \Sigma} f_x \cdot depth_T(x)$$

Abychom nalezli algoritmus pro sestavení stromu odpovídajícího Huffmanově kódu, projdeme si několik vlastností takových stromů.

Věta 4. *Pro každý optimální prefixový kód γ platí, že každý nelistový uzel v T_γ má stupeň 2.*

Důkaz. Dokážeme sporem. Předpokládejme, že ve stromě T_γ existuje uzel v s jedním potomkem u. Pokud uzel v ze stromu smažeme a nahradíme jej uzlem u, obdržíme strom s menší průměrnou hloubkou (všechny listy v podstromu generovaném uzlem u budou mít o 1 menší hloubku). Tento strom odpovídá prefixového kódu pro stejnou abecedu jako T_γ, protože jsme neodstranili žádný list. To je ale spor s tím, že γ je optimální kód. □

Věta 5. *Nechť γ je optimální prefixový kód. Pak pro každé dva listy y, z ve stromu T_γ platí, že pokud $depth_{T_\gamma}(z) > depth_{T_\gamma}(y)$, pak $f_y \geq f_z$.*

Důkaz. Sporem Pokud $f_y < f_z$, pak prohozením uzlů z a y získáme strom s menší průměrnou hloubkou, což je spor. Skutečně: Podíváme-li se na členy sumy $\sum_{x\in\Sigma} f_x \cdot depth_{T_\gamma}(x)$ pro $x \in \{y, z\}$, pak zjistíme, že

- násobek f_y vzroste z $depth_{T_\gamma}(y)$ na $depth_{T_\gamma}(z)$
- násobek f_z klesne z $depth_{T_\gamma}(z)$ na $depth_{T_\gamma}(y)$

Změna je tedy (rozdíl sumy před a po výměně pro x, y):

$$(f_y \cdot depth_{T_\gamma}(y) - f_y \cdot depth_{T_\gamma}(z)) + (f_z \cdot depth_{T_\gamma}(z) - f_z \cdot depth_{T_\gamma}(y)) = (depth_{T_\gamma}(y) - depth_{T_\gamma}(z))(f_y - f_z)$$

Poslední výraz je vždy kladný, proto má nový strom menší průměrnou hloubku. □

Věta 6. *Existuje optimální prefixový kód γ takový, že listy x, y v T_γ takové, že f_x a f_y jsou dvě nejmenší frekvence, jsou*

(a) v maximální hloubce

(b) sourozenci

Důkaz. (a) Plyne z předchozí věty.

(b) Prohazováním listů, které jsou ve stejné hloubce se nezmění průměrná hloubka listů. Protože v T_γ mají všechny nelistové uzly stupeň 2, musí existovat v maximální hloubce dva listy, které jsou sourozenci. Tyto listy pak můžeme prohodit s x a y. □

Předchozí věta je základní myšlenkou greedy algoritmu pro konstrukci T_γ. Algoritmus si udržuje množinu stromů, které postupně spojuje do výsledného stromu T_γ. Kořen každého takového stromu má přiřazeno číslo — sumu frekvencí znaků abecedy, které jsou listy stromu. Na začátku algoritmu vytvoříme pro každý znak z abecedy jednoprvkový strom, frekvence nastavíme na frekvence výskytů odpovídajích znaků. Poté algoritmus greedy strategií sestavuje výsledný strom:

1. Vybere dva stromy x, y s nejnižšími frekvencemi kořenů f_x a f_y a spojí je do nového stromu tak, že vytvoří nový kořen w, nastaví jeho frekvenci na $f_w = f_x + f_y$. Kořeny stromů x a y se pak stanou potomky w.
2. Opakuje předchozí krok, dokud nespojí všechny stromy do jednoho.

Aby byl algoritmus jasnější, uvedeme ho i v pseudokódu jako Algoritmus 3. Budeme předpokládat, že vstupem je pole znaků abecedy Σ a pole frekvencí výskytu těchto znaků F, a dále že uzly stromů mají položky *symbol*, *freq*, *left* a *right*.

Složitost Huffman je $O(|\Sigma| \log |\Sigma|)$. Prioritní frontu (řádky 2 až 9) zkonstruujeme v čase $O(|\Sigma| \log |\Sigma|)$. Poté $|\Sigma|-1$ krát opakujeme cyklus (řádky 10 až 18), ve kterém dvakrát odebereme a jednou přidáme prvek do prioritní fronty, tyto operace mají logaritmickou složitost. Ostatní operace v tomto cyklu mají konstantní složitost.

Zbývá dokázat, že Huffman skutečně konstruuje strom pro optimální kód.

Věta 7. *Nechť $S = \{x_1, \ldots, x_n\}$ je abeceda znaků s frekvencemi $f_{x_1} \ldots f_{x_n}$ a $S' = S - \{x_i, x_j\} \cup \{w\}$, kde x_i, x_j jsou znaky s nejmenšími frekvencemi a w je nový znak s frekvencí $f_w = f_{x_i} + f_{x_j}$. Nechť T' je strom optimálního kódu pro S'. Pak pro strom T, který dostaneme z T' tak, že nahradíme w vnitřním uzlem s potomky x_i, x_j platí:*

(a) $ABL(T') = ABL(T) - f_w$

(b) T je strom optimálního kódu pro abecedu S.

Důkaz. (a) Hloubky všech listů mimo x_i a x_j sou stejné v T i T'. Hloubka listů x_i a x_j v T je o 1 větší než

Algoritmus 3 Sestavení Huffmanova kódu

```
1: procedure HUFFMAN(Σ, F)
2:     Inicializuj prioritní frontu S uspořádanou podle frekvencí
3:     for i ← 0 to |Σ| − 1 do
4:         Vytvoř uzel x                          ▷ Vytvoříme jednoprvkový strom
5:         x.symbol ← Σ[i]
6:         x.freq ← F[i]
7:         x.left ← x.right ←NIL
8:         ENQUEUE(S, x)                          ▷ Vložíme strom do fronty
9:     end for
10:    while SIZE(S) > 1 do
11:        x ←DEQUEUE(S)                          ▷ x, y jsou stromy s nejnižšími frekvencemi
12:        y ←DEQUEUE(S)
13:        Vytvoř uzel w
14:        w.freq ← x.freq + y.freq
15:        w.left ← x
16:        w.right ← y
17:        ENQUEUE(S, w)                          ▷ Vložím nově vytvořený strom do fronty
18:    end while
19:    return DEQUEUE(S)                          ▷ Vrátím jediný strom v frontě
20: end procedure
```

hloubka w v T'. Odtud máme, že

$$\begin{aligned} ABL(T) &= \sum_{x\in S} f_x \cdot depth_T(x) \\ &= f_{x_i} \cdot depth_T(x_i) + f_{x_j} \cdot depth_T(x_j) + \sum_{x\neq x_i,x_j} f_x \cdot depth_{T'}(x) \\ &= (f_{x_i} + f_{x_j})(1 + depth_{T'}(w)) + \sum_{x\neq x_i,x_j} f_x \cdot depth_{T'}(x) \\ &= f_w + f_w \cdot depth_{T'}(w) + \sum_{x\neq x_i,x_j} f_x \cdot depth_{T'}(x) \\ &= f_w + \sum_{x\in S'} f_x \cdot depth_{T'}(x) = f_w + ABL(T') \end{aligned}$$

(b) Sporem. Předpokládejme, že kód odpovídající T není optimální. Pak existuje optimální kód se stromem Z tak, že $ABL(Z) < ABL(T)$. Podle věty 6 můžeme bez obav předpokládat, že x_i a x_j jsou v Z sourozenci. Označme jako Z' strom, který získáme ze Z náhradou podstromu generovaným rodičem uzlů x_i a x_j pomocí novéhu uzlu s frekvencí $f_w = f_{x_i} + f_{x_j}$. Pak podle (a) máme:

$$ABL(Z') = ABL(Z) - f_w < ABL(T) - f_w = ABL(T').$$

To je ale spor s tím, že T' je optimální. □

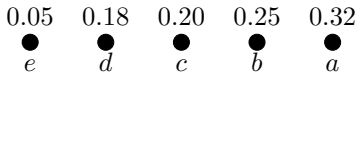
Věta 8. HUFMANN *vrací optimální kód.*

Důkaz. Stačí si všimnout, že jeden krok algoritmu odpovídá konstrukci z věty 7. □

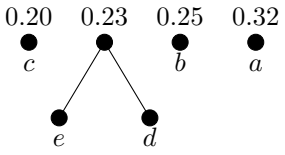
Příklad 5. Uvažujme abecedu s frekvencemi výskytu znaků danou tabulkou

x	f_x
a	.32
b	.25
c	.20
d	.18
e	.05

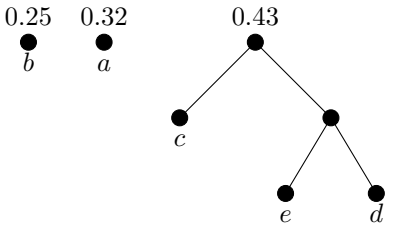
Fronta stromů v algoritmu pak postupně projde následujícími stavy.



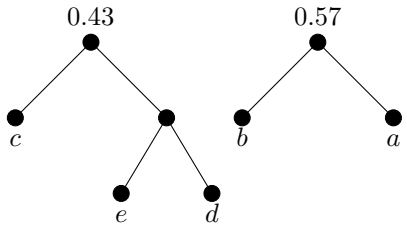
(a) inicializace



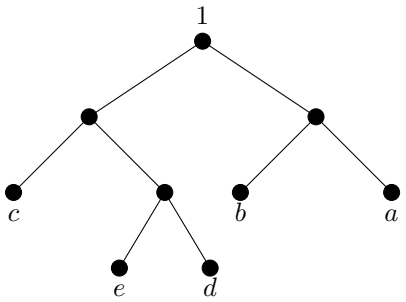
(b) Po 1. iteraci



(c) Po 2. iteraci



(d) Po 3. iteraci



(e) Po 4. iteraci

□

Reference

[1] Donald Knuth. The Art of Computer Programming, Volume I. Addison-Wesley, 1997

[2] Donald Knuth. Selected papers on analysis of algorithms. Center for the study of language an information, 2000

[3] Cormen et. al. Introduction to algorithms. The MIT press. 2008.

[4] Donald Knuth Concrete mathematics: A foundation for computer science. Addison-Wesley professional, 1994

[5] John Kleinberg, Éva Tardes. Algorithm design. Addison-Wesley, 2005.

[6] U. Vazirani et. al. Algorithms. McGraw-Hill, 2006.

[7] Steve Skiena. The algorithm design manual. Springer, 2008.

[8] Juraj Hromkovič. Algorithmics for hard problems. Springer, 2010.

[9] Oded Goldreich. Computational complexity: a conceptual perspective. Cambridge university press, 2008.