NÁVRH ALGORITMŮ

Poznámky ke kurzu Algoritmy 3

Univerzita Palackého
v Olomouci

OBSAH

KAPITOLA 1

HLADOVÉ ALGORITMY

Hladový algoritmus při každé své volbě vybírá tu aktuálně nejvýhodnější možnost a ignoruje přitom vliv, který má tato volba na zbytek jeho běhu. Nejdříve si ukážeme tento princip na příkladech algoritmu pro problém batohu.

Vstupem **problému batohu** je $n+1$ přirozených čísel

$$w_1, w_2, \ldots, w_n, b.$$

Čísla $w_1, w_2, \ldots, w_n$ považujeme za váhy položek $1, 2, \ldots, n$, číslo b je kapacita. Chceme nalézt množinu $T \subseteq \{1, 2, \ldots, n\}$ položek takovou, že

$$\sum_{i \in T} w_i \leqslant b, \tag{1.1}$$

a přitom je součet na levé straně nerovnosti (1.1) největší možný.[1] Tedy hledáme množinu položek, tak aby byla suma jí odpovídajích vah nejvýše rovna kapacitě, přitom však chceme tuto sumu maximalizovat. Pro instanci

$$w_1 = 5, w_2 = 1, w_3 = 3, w_4 = 3, b = 7 \tag{1.2}$$

existuje $2^4 = 16$ kandidátů na T, pro množiny

$$\{1,3\},\ \{1,2,3\},\ \{1,4\},\ \{1,2,4\},\ \{1,3,4\},\ \{1,2,3,4\},$$

neplatí (1.1). Zbývají

[1] Existuje varianta problému batohu, ve které navíc uvažujeme přirozená čísla $c_1, c_2, \ldots, c_n$, a chápeme je jako ceny položek $1, 2, \ldots, n$. Cílem je pak najít množinu položek T, pro kterou platí (1.1), maximalizujeme ale součet $\sum_{i \in T} c_i$. V literatuře se někdy označuje souslovím *problém batohu* právě tato rozšířená varianta, původní verzi se pak říká **jednoduchý problém batohu** (anglicky knapsack problem, simple knapsack problem).

$\{2,3,4\}$	součet vah 7,	$\{3\}$, $\{4\}$	součet vah 3,
$\{1,2\}$, $\{3,4\}$	součet vah 6,	$\{2\}$	součet vah 1,
$\{1\}$	součet vah 5,	$\emptyset$	součet vah 0.
$\{2,3\}$ $\{2,4\}$	součet vah 4,		

Najít tedy chceme množinu a $\{2,3,4\}$. Podotkněme, že existují instance, které mají řešení s nejlepší cenou více (viz Cvičení 1.1).

Problém batohu je zástupcem skupiny **optimalizačních problémů**. Takové problémy definujeme tak, že určíme

(a) vstupy problému,

(b) objekty, která považujeme za korektní řešení problému, tzv. **přípustná řešení**,

(c) pro každé přípustné řešení jeho cenu (obvykle nějakým předpisem),

(d) to, jestli chceme najít přípustné řešení s maximální nebo minimální cenou. Řešením s nejlepší cenou pak říkáme **optimální řešení**.

Přípustná řešení problému batohu jsou tedy množiny T položek, které splňují (1.1). Cenou přípustného T řešení je suma vah položek obsažených v T. Za optimální pak považujeme T s maximální cenou.

Hladové algoritmy jsou velmi přirozené. Pokud hledáme algoritmus pro optimalizační problém, často nás jako první napadne nějaká forma hladového algoritmu. Pro úlohu batohu může být základní myšlenkou takového algoritmu

> *Začnu s prázdným řešením a pak, dokud to jde, do něj vždy přidám tu nejtěžší položku, která se do něj ještě vejde.*

Algoritmus je hladový, protože vybírá aktuálně nejlepší možnost (nejtěžší položku) a nijak se nezabývá tím, jaké důsledky to má pro výběr dalších položek. Výše uvedená jednoduchá myšlenka vede k následujícímu algoritmu.

Algoritmus GREEDY-KNAPSACK

1. Nastavíme $T \leftarrow \emptyset$, $s \leftarrow 0$.

2. Nalezneme položku $i \notin T$ takovou, že

$$s + w_i \leqslant b \tag{1.3}$$

a současně je w_i maximální mezi položkami, které splňují (1.3) a nepatří do T *(položka i má tedy mezi položkami, které bychom chtěli přidat do řešení a jejich přidáním nedojde k přesáhnutí kapacity, maximální váhu).*

3. Pokud i neexistuje (protože všechny položky splňující (1.3) už jsou obsaženy v T), algoritmus končí, výsledkem je T.

4. Pokud i existuje, pak nastavíme $T \leftarrow T \cup \{i\}$ a $s \leftarrow s + w_i$ a přejdeme ke kroku 2.

Snadno vidíme, že GREEDY-KNAPSACK vždy vrátí přípustné řešení. Pro T totiž po celý běh algoritmu platí (1.1): Určitě platí na začátku, kdy je T prázdná množina. Díky (1.3) v kroku 3 a tomu, že na začátku je $s = 0$, platí po provedení kroku 4 vždy

$$s = \sum_{j \in T} w_j \leqslant b.$$

Pokud ovšem GREEDY-KNAPSACK spustíme s instancí (1.2), nabude T postupně hodnot

$$\begin{aligned} \{1\} &\quad \text{součet vah } 5, \\ \{1,2\} &\quad \text{součet vah } 6. \end{aligned}$$

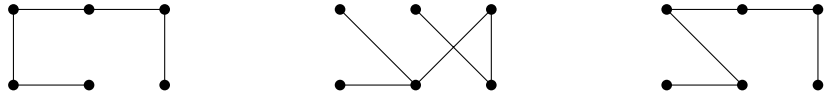
Algoritmus tedy nevrací optimální řešení. Řekneme, že GREEDY-KNAPSACK je **korektní**, protože vždy vrací přípustné řešení. Není ovšem **optimální**, protože nemusí vrátit optimální řešení. S neoptimálním algoritmem se spokojíme například v případě, kdy všechny optimální algoritmy mají mnohem horší časovou složitost než ten neoptimální. Vztahu mezi cenou optimální řešení a cenou řešení vráceného GREEDY-KNAPSACK se věnuje Cvičení 1.2.

Složitost GREEDY-KNAPSACK závisí na implementačních detailech, hlavně na implementaci bodu 2 (a způsobu reprezentace množiny T). Pokud bychom zvolili naivní implementaci, kdy v bodu 2 hledáme i tak, že pro každou položku testujeme, zda-li je v T a zda-li pro ni platí (1.3), dostaneme složitost GREEDY-KNAPSACK $O(n^3)$, kde n je počet položek (předpokládáme, že aritmetické operace děláme v konstantním čase). Algoritmus lze ovšem implementovat i lépe, čemuž se věnujeme ve Cvičení 1.3.

1.1 NALEZENÍ MINIMÁLNÍ KOSTRY GRAFU

Neorientovaný graf G s množinou vrcholů V a množinou hran E je **souvislý**, pokud pro libovolnou dvojici různých vrcholů $v, w \in V$ existuje alespoň jedna cesta z v do w. Maximálnímu souvislému podgrafu grafu G říkáme **komponenta** grafu G, slovíčkem maximální myslíme to, že do podgrafu nemůžeme přidat žádnou hranu ani vrchol, aniž bychom porušili podmínkou souvislosti. Na následujím obrázku je nalevo souvislý graf, graf napravo souvislý není a obsahuje 3 komponenty.

V grafu se nachází **kružnice**, pokud existuje dvojice různých vrcholů v, w tak, že existují alespoň 2 různé cesty z v do w. **Kostra** souvislého grafu G je jeho souvislý podgraf, který obsahuje všechny vrcholy G a neobsahuje kružnici.[2] Graf může obecně obsahovat více koster, jak vidíme na následujícím obrázku, který zobrazuje některé z koster z grafu z předchozího obrázku vlevo.



Jednou z pro nás užitečných vlastností kostry je vztah mezi jejím počtem uzlů a hran. Ten je zachycen v následujícím tvrzení.

Věta 1. — V souvislém grafu G s množinou vrcholů V a množinou hran E není kružnice, právě když platí $|V| = |E| + 1$.

Důkaz. Předpokládejme, že G neobsahuje kružnici. Ukážeme, že pak nutně platí $|V| = |E| + 1$. Důkaz provedeme indukcí přes $|V|$. Pokud je $|V| = 1$, pak G neobsahuje žádnou hranu a tvrzení platí. Předpokládejme nyní, že $|V| > 1$ a že tvrzení platí pro všechny grafy s $|V| - 1$ uzly. Důležité pozorování nyní je

Protože v G *není kružnice, existuje v něm vrchol se stupněm 1.*

Stačí si vybrat libovolný vrchol v grafu a zjistit, jestli má stupeň 1. Pokud ano, požadovaný vrhol se stupněm 1 jsme našli. Pokud ne, vybereme si jeho libovolného souseda a položíme otázku o stupni pro něj. Pokud budeme tento postup opakovat, nutně přijdeme k vrcholu se stupněm 1. Nemůžeme se totiž nikdy vrátit k vrcholu, pro který už jsme otázku položili, to bychom totiž v G našli kružnici. Řekněme tedy, že v je vrchol stupně 1. Pokud z G odstraníme v a hranu, kterou je spojen s zbytkem grafu, dostaneme graf bez cyklů s $|V| - 1$ uzly a $|E| - 1$ hranami, pro který podle indukčního předpokladu platí

$$|V| - 1 = |E| - 1 + 1$$

a odtud už dostaneme požadované tvrzení.

Abychom dokázali opačnou implikaci, předpokládejme, že platí $|V| = |E| + 1$. Ukážeme, že pak v G není kružnice. Důkaz opět provedeme pomocí indukce přes $|V|$. Pokud $|V| = 1$, pak tvrzení triviálně platí. Předpokládejme tedy, že $|V| > 1$ a že tvrzení platí pro všechny grafy s $|V| - 1$ uzly. Důležité pozorování je potom

Protože platí $|V| = |E| + 1$, *existuje v* G *vrchol se stupněm 1.*

[2] Velmi přirozeně by šlo definovat analogický pojem pro nesouvislé grafy: požadovali bychom existenci kostry v každé komponentě takového grafu. V dalším textu se ovšem tímto zabývat nebudeme.

Označme pomocí $\deg(v)$ stupeň uzlu v. Víme, že v každém grafu platí

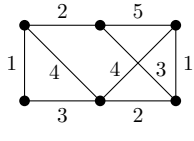
$$\sum_{v \in V} \deg(v) = 2|E|.$$

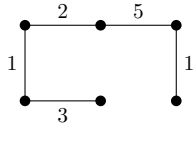
Předpokládáme $|V| = |E| + 1$ a proto po dosazení dostáváme

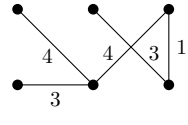
$$\sum_{v \in V} \deg(v) = 2|V| - 2.$$

Vidíme tedy, že musí existovat alespoň jeden uzel se stupněm menším než 2. Současně je G souvislý a proto neobsahuje uzly se stupněm 0. Musí tedy obsahovat alespoň jeden uzel se stupněm 1. Řekněme tedy, že v je vrchol se stupněm 1. Pokud z G odstraníme v a hranu, kterou je spojen s zbytkem grafu, dostaneme graf s $|V| - 1$ vrcholy a $|E| - 1$ hranami, pro který platí $|V| - 1 = |E| - 1 + 1$, a podle indukčního předpokladu neobsahuje kružnici. Přidáním v a hrany, kterou jsme odebrali, zpět k tomuto grafu kružnici vytvořit nemůžeme. □

Ke grafu G přidáme zobrazení $\delta : E \to \mathbb{Q}^+$, které každé hraně přiřadí její ohodnocení (číslu $\delta(e)$ tedy říkáme ohodnocení hrany e). Dvojici G, δ pak říkáme **hranově ohodnocený graf**. Cenu kostry T grafu G definujeme jako součet ohodnocení hran, které se v T nachází. Ohodnocení hran můžeme také namalovat na obrázku grafu, jak lze vidět na následujícím příkladu grafu a jeho dvou koster, kostra vlevo s cenou 12, kostra vpravo s cenou 15.







Vstupem problému nalezení minimální kostry je hranově ohodnocený graf. Cílem je nalézt kostru tohoto grafu s minimální cenou.

1.1.1 Kruskalův algoritmus

Předpokládejme, že na vstupu máme souvislý graf G s množinou vrcholů V a množinou hran E, spolu s ohodnocením hran δ. Protože libovolná kostra grafu G obsahuje všechny jeho vrcholy, k nalezení kostry stačí nalézt množinu jejích hran. Potřeba minimalizovat součet ohodnocení hran v kostře nás vede k následující myšlence.

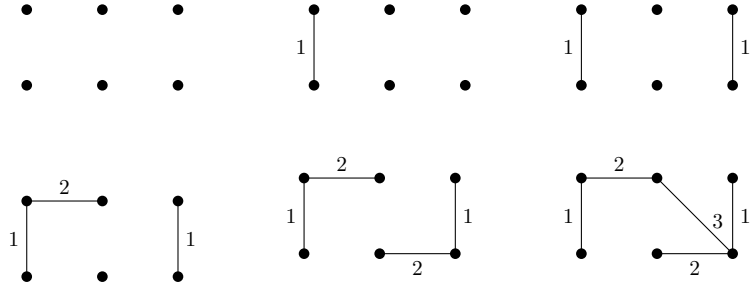
> *Začneme s prázdnou množinou a budeme do ní po jedné přidávat hrany. Pro přidání vybereme hranu jejíž přidání nezpůsobí vznik kružnice, a která má navíc mezi takovými hranami nejmenší ohodnocení.*

Algoritmus KRUSKAL

Vstupem algoritmu je souvislý graf G *s množinou uzlů* V*, množinou hran* E *a hranovým ohodnocením* δ*. Výstupem je množina hran* T*.*

1. Nastavíme $T \leftarrow \emptyset$
2. Pokud je $|T| = |V| - 1$, pak algoritmus končí, výsledkem je T.
3. Nalezneme hranu e, jejíž přidání do T nezpůsobí vznik kružnice (*chceme tedy, aby v grafu s množinou uzlů* V *a množinou hran* $T \cup \{e\}$ *nebyla kružnice*), a která má mezi takovými hranami nejmenší ohodnocení.
4. Nastavíme $T \leftarrow T \cup \{e\}$ a přejdeme ke kroku 2.

Průběh algoritmu na grafu z předchozího příkladu je na dalším obrázku. Namalován je graf s vrcholy V a hranami T, vždy před provedením kroku 2 algoritmu.



O algoritmu KRUSKAL nyní dokážeme, že vždy vrací minimální kostru, tedy že je optimální.

Věta 2. — Algoritmus KRUSKAL vrací optimální řešení.

Důkaz. Nejdříve musíme ukázat, že algoritmus vrací kostru. Z kroku 3 algoritmu plyne, že v průběhu algoritmu neobsahuje graf G' s množinou vrcholů V a množinou hran T kružnici. Z Věty 1 pak plyne, že v momentě, kdy platí $|T| = |V| - 1$ je graf G' souvislý. Pokud by totiž G' souvislý nebyl a bylo by potřeba ještě provést kroky 3 a 4 a doplnit tak do T hrany, aby se souvislým stal, dostali bychom $|T| > |V| - 1$. To je ovšem spor s Větou 1. Vidíme tedy, že G' je na konci algoritmu skutečně kostra. Zároveň také pokud $|T| < |V| - 1$, tak G' souvislý není a v kroku 3 vždy nalezneme nějakou hranu.

Ukážeme, že před každým provedením kroku 2 algoritmu existuje minimální kostra s množinou hran S taková, že $T \subseteq S$ (tj. že T je vždy podmnožinou množiny hran nějaké minimální

kostry). Odtud už vyplývá optimalita KRUSKAL. Tvrzení dokážeme indukcí přes $|T|$. Pro $|T| = 0$ (tj. $T = \emptyset$) tvrzení triviálně platí. Předpokládejme nyní, že $|T| > 1$ a že existuje minimální kostra s množinou hran S' tak, že $T \subseteq S'$. Nechť e je hrana, kterou algoritmus přidá jako další do T. Pokud $e \in S'$, pak tvrzení platí i pro $T \cup \{e\}$. Pokud $e \notin S'$, pak v grafu G' s množinou vrcholů V a množinou hranam $S' \cup \{e\}$ existuje kružnice. Na této kružnici leží hrana e, protože kružnice přidáním této hrany k S' vznikla, a také hrana $f \notin T$. Pokud by totiž f neexistovala, vznikla by přidáním e do T kružnice. Uvažme nyní množinu

$$S = (S' - \{f\}) \cup \{e\}\,.$$

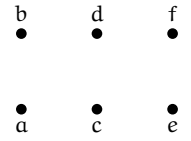
Jistě platí, že $T \subseteq S$. Dále víme, že S je množina hran nějaké kostry, protože $S' \cup \{e\}$ obsahuje kružnici, a odstraněním f, která na této kružnici leží, dostaneme kostru. Protože S' je množina hran minimální kostry, máme $\delta(f) \leqslant \delta(e)$. Na druhou stranu, algoritmus si vybral e dříve než f a přitom přidáním f do T nevznikne kružnice. To je proto, že $T \subseteq S'$, $f \in S'$ a v S' není kružnice. Z bodu 3 algoritmu tedy plyne, že $\delta(e) \leqslant \delta(f)$. Dohromady tedy $\delta(e) = \delta(f)$ a S je množinou hran nějaké minimální kostry. □

K tomu, abychom algoritmus KRUSKAL popsali úplně, musíme doplnit detaily ke kroku 3, ve kterém hledáme hranu s nejmenším ohodnocením nepatřící do T, jejímž přidáním do T nevznikne kružnice. Před prvním provedením kroku 2 si množinu hran E uspořádáme podle ohodnocení do vzestupné posloupnosti. Tu potom v průběhu algoritmu procházíme a pro každou hranu testujeme, jestli jejím přidáním vznikne kružnice. Test toho, jestli přidáním hrany e do T vznikne kružnice, je založen na následujícím pozorování:

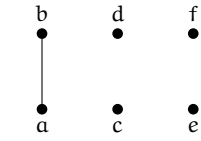
> *Žádná z komponent grafu s množinou vrcholů* V *a množinou hran* T *neobsahuje kružnici. Pokud má přidáním* e *kružnice vzniknout, musí oba její vrcholy ležet ve stejné komponentě.*

Z předchozího pozorování také plyne, že pokud přidáním hrany e do T nevznikne kružnice, musí být její vrcholy v různých komponentách. Přidáním e pak tyto komponenty spojíme do jedné.

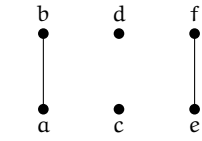
V průběhu algoritmu budeme jednotlivé komponenty reprezentovat jako množiny vrcholů. K testu toho, jestli přidáním hrany vznikne kružnice pak stačí ověřit, jestli se oba její vrcholy nachází ve stejné množině. Přidání hrany, a tedy spojení dvou komponent, pak provedeme sjednocením odpovídajících množin. Na následujícím obrázku je zobrazen předchozí příklad běhu algoritmu spolu s komponentami reprezentovanými jako množiny vrcholů.



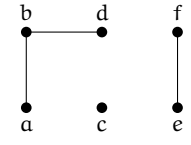
$\{a\},\{b\},\{c\},\{d\},\{e\},\{f\}$



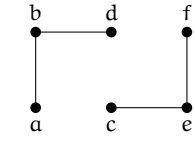
$\{a,b\},\{c\},\{d\},\{e\},\{f\}$



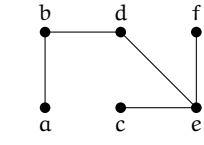
$\{a,b\},\{c\},\{d\},\{e,f\}$



$\{a,b,d\},\{c\},\{e,f\}$



$\{a,b,d\},\{c,e,f\}$



$\{a,b,c,d,e,f\}$

K uložení systému disjuktních množin, ve kterém potřebujeme v algoritmu KRUSKAL ukládat vrcholy, slouží datová struktura **disjoint set structure** (tento název budeme zkracovat na dss). Obecně slouží dss k uložení systému množin

$$\mathcal{S} = \{S_1, S_2, \ldots, S_n\},$$

kde pro libovolné $S_i \neq S_j$ platí $S_i \cap S_j = \emptyset$., tedy množiny v $\mathcal{S}$ jsou po dvojicích disjunktní. Navíc žádná z množin v $\mathcal{S}$ není prázdná. Množinu S_i ze systému identifikujeme pomocí reprezentanta, k tomuto účelu určeného prvku $r \in S_i$. Nad dss jou definovány následující operace.

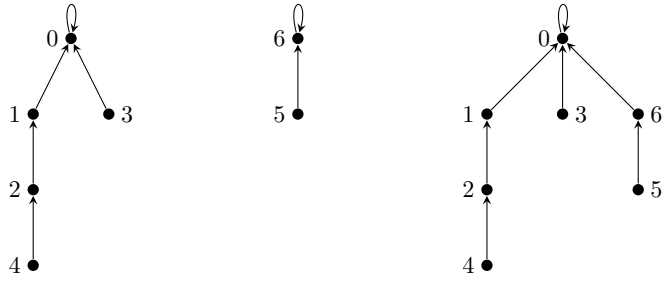
- MAKE-SET (x). Přidá do $\mathcal{S}$ množinu $\{x\}$. Předpokládáme, že x se před použitím operace nenachází v žádné z množin v systému $\mathcal{S}$.
- FIND (x). Vrátí reprezentanta množiny ze systému $\mathcal{S}$, do které patří prvek x. Před použitím operace očekáváme, že x se v některé z množin ze systému nachází.
- UNION (x, y). Z $\mathcal{S}$ odstraní množiny, do kterých patří prvky x a y, a přidá tam jejich sjednocení. Před použitím operace očekáváme, že x a y patří do různých množin.

Při implementaci je výhodné jednotlivé množiny reprezentovat jako stromy. Uzel u takového stromu je struktura `node` s položkami.

- `parent`. Odkaz na předka u ve stromu. Pokud je uzel kořenem stromu, je předkem sám svým předkem.
- `rank`. Odhad výšky podstromu s kořenem u.

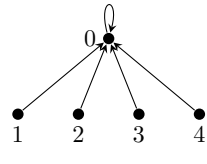
Budeme také předpokládat, že prvky množin jsou přímo struktury `node` a že tyto struktury jsou také argumenty operací nad dss. Pro jednoduchost budeme totiž počítat s tím, že do dss

ukládáme prvky $0, 1, 2, \ldots$. Jednotlivé uzly pak lze uchovávat v poli, přičemž na indexu i v tomto poli se nacházet uzel odpovídající prvku i. Pokud potřebujeme do dss ukládat jiné objekty než $0, 1, 2, \ldots$, lze si vazbu mezi těmito objekty a $0, 1, 2, \ldots$ uchovávat bokem. Na dalším obrázku jsou příklady stromů pro množiny $\{0, 1, 2, 3, 4\}$, $\{5, 6\}$, $\{0, 1, 2, 3, 4, 5, 6\}$.



Třetí strom na obrázku vznikl nastavením kořene prvního stromu jako předka kořene druhého stromu. Takové spojení je základem operace UNION . Argumenty UNION ovšem nemusí nutně být kořeny stromů, ty nejdříve potřebujeme najít. Protože reprezentantem stromu je jeho kořen a operace FIND , která dostane na vstupu uzel stromu, tedy musí pomocí odkazů na předka projít stromem až do kořene a ten vrátí, můžeme v operaci UNION k nalezení kořenů použít právě FIND .

Časová složitost FIND závisí na výšce stromu. Tu můžeme snížit jednoduchým trikem: když FIND prochází stromem ke kořeni, přepojíme všechny uzly, které navštíví, jako přímé potomky kořene. Zavoláme-li tedy FIND pro uzel odpovídající 4 ve stromu předchozím příkladě vlevo, bude tento strom po skončení FIND vypadat následovně.



V operaci UNION se snažíme udržovat výšku stromů co nejnižší tím, že kořen stromu s větší výškou nastavíme jako předka kořene stromu s menší výškou. Tím je výška výsledného stromu rovna výšce stromu s větší výškou. Bohužel zjišťovat skutečné výšky stromů v operaci UNION by bylo příliš výpočetně náročné. Můžeme ale v položce `rank` v kořenovém uzlu stromu udžovat odhad jeho výšky: `rank` nastavíme na začátku na 0. V případě, že v operaci UNION spojujeme dva stromy se stejnou hodnotou `rank`, zvýšíme ji v kořeni výsledného stromu o 1. V operaci FIND položku `rank` neupravujeme, protože nevíme, jestli jsme výšku stromu skutečně zmenšili. Proto je `rank` jenom horní odhad výšky stromu, výška stromu může být menší.

Nyní už můžeme uvést kompletní Kruskalův algoritmus, který používá dss.

Algoritmus FIND

Vstupem je struktura node x. *Výstupem je struktura* node, *která je kořenem stromu, ve kterém se nachází* x.

1. Pokud je $x = x.\texttt{parent}$, pak algoritmus končí a výsledek je x.
2. Nastavíme $x.\texttt{parent} \leftarrow$ FIND $(x.\texttt{parent})$. Výsledek algoritmu je $x.\texttt{parent}$.

Algoritmus UNION

Vstupem jsou struktury node x, y.

1. Nastavíme $r_x \leftarrow$ FIND (x), $r_y \leftarrow$ FIND (y).
2. Pokud je $r_x.\texttt{rank} > r_y.\texttt{rank}$, pak nastavíme $r_y.\texttt{parent} \leftarrow r_x$ a algoritmus končí.
3. Nastavíme $r_x.\texttt{parent} \leftarrow r_y$ a pokud $r_x.\texttt{rank} = r_y.\texttt{rank}$ tak nastavíme $r_y.\texttt{rank} \leftarrow r_y.\texttt{rank} + 1$.

Algoritmus MAKE-SET

Vstupem je struktura node x

1. Nastavíme $x.\texttt{rank} \leftarrow 0$ a $x.\texttt{parent} \leftarrow x$.

Algoritmus KRUSKAL-DSS

Vstupem algoritmu je souvislý graf G *s množinou uzlů* V, *množinou hran* E *a hranovým ohodnocením* δ. *Předpokládáme, že* $V = \{0, 1, \ldots, n-1\}$. *Výstupem je množina hran* T.

1. Nastavíme $T \leftarrow \emptyset$, $c \leftarrow 0$ a inicializujeme pole A struktur typu node o délce n.
2. Pro $i = 0, 1, \ldots, n-1$ zavoláme MAKE-SET $(A[i])$.
3. Vytvoříme pole hran H, ve kterém budou hrany setřízeny vzestupně podle ohodnocení δ.
4. Pokud je $T = |V| - 1$, pak algoritmus končí, výsledkem je T.
5. Označme vrcholy hrany $H[c]$ pomocí x, y. Pokud FIND $(A[x]) \neq$ FIND $(A[y])$, nastavíme $T \leftarrow T \cup \{H[c]\}$ a zavoláme UNION $(A[x], A[y])$.
6. Nastavíme $c \leftarrow c + 1$ a přejdeme ke kroku 2.

Složitost sekvence m operací nad dss, přičemž n operací je MAKE-SET , je $O(m \cdot \alpha(n))$, kde α je *velmi* pomalu rostoucí funkce (je to *inverzní Ackermanova funkce*), která má pro n vyskytující se v prakticky představitelných aplikacích hodnotu menší než 5. V algoritmu KRUSKAL-DSS je $m = 3 \cdot |E| + |V|$ a $n = |V|$. Krok 3 lze provést v čase $O(|E| \log |E|)$, což je i celková složitost KRUSKAL-DSS. Je tomu tak proto, že $\alpha(n) \in O(\log n)$ a že u souvislých grafů je není počet hran menší než počet vrcholů (zanedbáme-li situaci, že je graf kostrou, tam je počet hran o 1 menší).

1.2 SESTAVENÍ HUFFMANOVA KÓDU

V této kapitole se budeme zabývat nalezením Huffmanova kódu. Huffmanův kód je využíván pro bezeztrátovou kompresi dat[3]. Pro naše účely si stačí představit, že komprimujeme text v anglickém jazyce.

Zafixujme tedy abecedu Σ obsahující znaky komprimovaného textu. Kód je zobrazení γ, které znaku x z abecedy Σ přiřadí neprázdnou konečnou posloupnost bitů $\gamma(x)$, které jsou jeho *zakódováním*. Přitom přirozeně požadujeme, aby γ různé znaky kódovalo různě: musí tedy platit, že pro znaky $x \neq y$ máme $\gamma(x) \neq \gamma(y)$ (tj. γ je prosté zobrazení). Například pro $\Sigma = \{a, b, c\}$ je následující zobrazení kódem.

$$\gamma(a) = 0,\ \gamma(b) = 1,\ \gamma(c) = 10, \tag{1.4}$$

Text zakódujeme tak, že jej procházíme zleva doprava, na znaky aplikujeme γ a výsledky řetězíme za sebe. Výsledkem zakódování slova $z_1 z_2 \dots z_n$ (z_i jsou znaky ze Σ) je tedy posloupnost bitů

$$\gamma(z_1)\gamma(z_2)\dots\gamma(z_n),$$

kterou budeme značit také $\gamma(z_1 z_2 \dots z_n)$. Například slovo `bbac` zakódujeme pomocí (1.4) na `11010`. Posloupnosti, která vznikla zakódováním nějaké slova, budeme říkat *kódové slovo*. Pokud bychom chtěli z `11010` získat zpět slovo, ze kterého jsme vyšli, zjistíme, že existuje více možností. Toto kódové slovo totiž mohlo vzniknout i zakódovaním slova `bcc`. Protože přirozeně chceme, abychom dekódováním získali slovo, které jsme zakódovali, takové chování kódu pro nás není přijatelné. Budeme se proto zabývat pouze *jednoznačně dekódovatelnými kódy*. Kód γ je jednoznačně dekódovatelný, pokud zakódováním různých slov dostaneme různá kódová slova. To v předchozím případě neplatilo. Mezi jednoznačně dekódovatelné kódy patří *blokové* kódy, ve kterých mají všechna zakódovaní znaků stejný počet bitů. Příkladem takového kódu je ASCII tabulka. Dalším typem kódu, který je jednoznačně dekódovatelný, je kód pro který platí následující vlastnost:

pro všechna $x, y \in \Sigma$ platí, že $\gamma(x)$ není prefixem (= začátkem slova) $\gamma(y)$

Takové kódy označujeme jako *prefixové*. Například kód

$$\gamma(a) = 11,\ \gamma(b) = 01,\ \gamma(c) = 001,\ \gamma(d) = 10. \tag{1.5}$$

Blokové kódy mohou být neefektivní v tom, že zanedbávají frekvenci výskytů znaků a tím je zakódování textu 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. Text w nad Σ, který obsahuje

[3] Je využit např. jako součást v kompresi/dekompresi formátu jpg.

10 000 krát a,
50 000 krát b,
35 000 krát c,
5 000 krát c,

(celkem má tedy 100 000 znaků), zakódujeme pomocí zmíněného blokového kódu do 200000 bitů dlouhého kódového slova. Pokud bychom ovšem sestrojili kód, který často vyskytujímu se znaku přiřadí kratší zakódování než málo se vyskytujícím znakům, můžeme w zakódovat pomocí méně bitů. Použijeme-li například kód (1.5), ve kterém a zakódujeme pomocí 3 bitů, b pomocí 1 bitu, c pomocí 2 bitů a d pomocí 3 bitů, dostaneme kódové slovo, které má

$$3 \cdot 10000 + 1 \cdot 50000 + 2 \cdot 35000 + 3 \cdot 5000 = 165000$$

bitů. Ušetříme tak 35000 bitů, což je 17.5 procentní úspora.

Každému kódu můžeme přiřadit jeho efektivitu. Přirozeně, chceme-li zakódovat slovo w, je mírou efektivity kódu γ délka kódového slova $\gamma(w)$. Ze situace můžeme vypustit závislost na délce w nahrazením počtu výskytů jednotlivých znaků jejich frekvencí. *Frekvence* f_x znaku x ve slově w je

$$f_x = \frac{\text{počet výskytů } x}{|w|},$$

kde $|w|$ značí délku slova w (značení délky slova pomocí $|.|$ budeme používat obecně). Efektivitu kódu pak vyjádříme jako[4]

$$\textsc{pdz}(\gamma) = \sum_{x \in \Sigma} f_x \cdot |\gamma(x)|,$$

Snadno vidíme, že $n \cdot \textsc{pdz}(\gamma) = |\gamma(w)|$.

Huffmanův kód je prefixový kód γ, který má pro danou abecedu a dané frekvence výskytu znaků minimální $\textsc{pdz}(\gamma)$. Nalezení Huffmanova kódu je tedy z tohoto pohledu minimalizační problém, kde vstupem je abeceda a frekvence výskytu znaků, přípustná řešení jsou prefixové kódy a cena kódu je dána PDZ.

Prefixový kód lze jednoznačně reprezentovat kořenovým stromem. Tato reprezentace je výhodnější při dekompresi a je také použita v algoritmu, který sestaví Huffmanův kód.

Strom prefixového kódu je binární, jeho listy jsou tvořeny prvky Σ. Hrany z nelistových uzlů k levému potomku jsou označeny `0`, k pravému potomku `1`. Ke kódu γ setrojíme následovně.

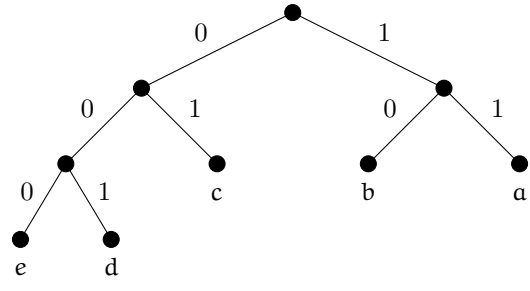
1. začínáme s kořenem
2. všechny znaky x, jejichž první bit $\gamma(x)$ je `0` jsou listy v levém podstromu, ostatní (první bit $\gamma(x)$ je `1`) jsou listy v pravém podstromu.

[4] PDZ je zkratka pro *průměrná délka zakódování*

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 list.

Pro znak x najdeme ve stromě $\gamma(x)$ tak, že jdeme od kořene do listu x, zřetězíme bity čtené na navštívených hranách, a takto získanou posloupnost bitů otočíme. Příklad prefixového kódu a jemu odpovídajícího stromu je na následujícím obrázku.

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



Do konce kapitoly budeme prefixové kód uvažovat v reprezentaci stromem. Pro takový strom T zavedeme analogii PDZ, ve které nahradíme délku zakódování znaku, hloubkou listu odpovídajícího daném znaku v T, tedy

$$\text{PDZ}(T) = \sum_{x \in \Sigma} f_x \cdot \text{HLB}(x, T),$$

kde pomocí $\text{HLB}(x, T)$ značíme hloubku listu (odpovídajícího znaku) x ve stromě T.

Věta 3. — (i) Pro každý strom T Huffmanova kódu platí, že nelistové vrcholy v T mají právě dva potomky.

(ii) Pro každý strom T Huffmanova kódu platí, že pokud pro listy y, z máme $\text{HLB}(z, T) > \text{HLB}(y, T)$, pak $f_y \geqslant f_z$. (*Listy, které jsou ve větší hloubce, nemohou mít větší frekvenci než ty v menší hloubce.*)

(iii) Existuje strom T Huffmanova kódu, ve kterém jsou listy x, y znaků s nejmenšími frekvencemi v maximální hloubce a mají společného rodiče.

Důkaz. (i) viz Cvičení 1.10

(ii) Pokud $f_y < f_z$, pak prohozením uzlů z a y získáme strom s menší průměrnou hloubkou, což je spor. Podíváme-li se na členy sumy $\text{PDZ}(T) = \sum_{x \in \Sigma} f_x \cdot \text{HLB}(x, T)$ pro $x \in \{y, z\}$, pak zjistíme, že

$$\text{násobek } f_y \text{ vzroste z } \text{HLB}(y, T) \text{ na } \text{HLB}(z, T),$$
$$\text{násobek } f_z \text{ klesne z } \text{HLB}(z, T) \text{ na } \text{HLB}(y, T).$$

Protože $f_y < f_z$ (to je náš předpoklad, který chceme dovést ke sporu), prohozením x a z ve stromu součet odpovídajících členů sumy v $\textsc{pdz}(T)$ klesne. Tím ovšem zmenšíme i $\textsc{pdz}(T)$.

(iii) To že jsou x, y v maximální hloubce plyne z (i) a (ii). Protože prohazováním uzlů ve stejné hloubce ve stromě neměníme PDZ, můžeme x, y prohodit tak, aby měli společného rodiče. □

Pokud v T existují listy x, y, které mají stejného rodiče, můžeme provést operaci FOLD. Tu provedeme tak, že ze stromu odstraníme listy x, y. Tím se stane z rodiče vrcholů x, y list. Ten nastavíme tak, aby odpovídal novému symbolu z. Vzniklý strom $\textsc{fold}(T)$ je poté stromem (prefixového kódu) pro abecedu $\Sigma' = \Sigma - \{x, y\} \cup \{z\}$. Navíc nastavíme $f_z = f_x + f_y$. Pak platí

$$\textsc{pdz}(T) = \textsc{pdz}(\textsc{fold}(T)) + f_z \tag{1.6}$$

Platí totiž, že $\textsc{hlb}(z, \textsc{fold}(T)) + 1 = \textsc{hlb}(x, T) = \textsc{hlb}(y, T)$ a tedy

$$\begin{aligned}(f_a + f_b) \cdot \textsc{hlb}(x, T) &= f_z \cdot (\textsc{hlb}(z, \textsc{fold}(T)) + 1) \\ &= f_z \cdot \textsc{hlb}(z, \textsc{fold}(T)) + f_z.\end{aligned}$$

Ostatní členy v součtu v PDZ zůstanou shodné. Na druhou stranu, získáme-li strom T' pro abecedu Σ', můžeme provést opačnou operaci UNFOLD, ve které k vrcholu z připojíme jako potomky x, y a dostaneme tak strom (prefixového kódu) pro Σ. Přitom platí rovnice

$$\textsc{pdz}(T') + f_z = \textsc{pdz}(\textsc{unfold}(T')), \tag{1.7}$$

kterou lze odvodit analogicky (1.6). Pokud je navíc T' strom optimálního kódu pro Σ', pak je $\textsc{unfold}(T')$ stromem optimálního kódu pro Σ. Pokud by totiž $\textsc{unfold}(T')$ optimální nebyl, existoval by optimální strom S, pro který by platilo

$$\textsc{pdz}(\textsc{unfold}(T')) > \textsc{pdz}(S).$$

Podle věty 3 (iii) můžeme provést $\textsc{fold}(S)$ a s použitím (1.6) a (1.7) dostaneme

$$\textsc{pdz}(T') > \textsc{pdz}(\textsc{fold}(S)),$$

což je ale spor s tím, že T' je optimální.

Předchozí odstavec je současně odvozením následujícího algoritmu i důkazem jeho správnosti a optimality.

Algoritmus HUFFMAN

Vstupem algoritmu je abeceda Σ a frekvence znaků. Výstupem je strom optimálního prefixového kódu.

1. Pokud obsahuje Σ pouze dva znaky, vytvoříme strom s kořenem a těmito znaky jako listy.

2. Nalezneme dva znaky s nejmenšími frekvencemi, označme je x, y. Vytvoříme abecedu $\Sigma' = \Sigma - \{x, y\} \cup \{z\}$, kde z je nový znak s frekvencí $f_x + f_y$. (*Toto je odvozeno z operace* FOLD.)

3. Rekurzivně zavoláme HUFFMAN(Σ') a získáme strom T.

4. Vrátíme strom UNFOLD(T).

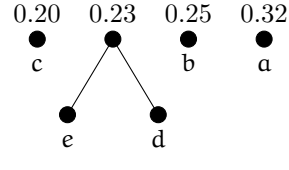
Při implementaci lze v kroku 2 nový za nový znak z vzít přímo strom s kořenem a listy x, y. Potom není nutné provádět krok 4, a algoritmus nemusí být rekurzivní (resp. je koncově rekurzivní, a takovou rekurzi je snadné převést na cyklus). Symboly v abecedě poté můžeme udržovat uspořádané podle frekvencí v prioritní frontě. Složitost algoritmu je pak $O(n \log n)$.

Nakonec uveďme příklad běhu algoritmu (ve iterativní formě popsané v předchozím odstavci). Pro abecedu

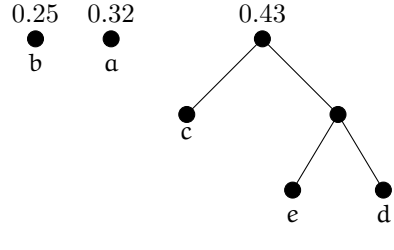
x	f_x
a	.32
b	.25
c	.20
d	.18
e	.05

Dostaneme postupně následující abecedy (a na konci strom optimální kódu), kde znaky zobrazujeme jako stromy s frekvencí uvedenou nad vrcholem.

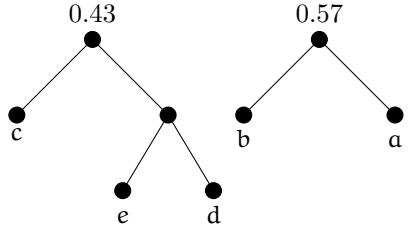
Po první iteraci.



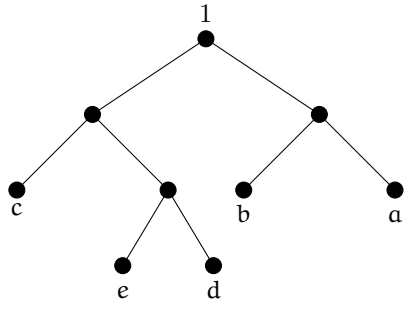
Po druhé iteraci.



Po třetí iteraci.



Po čtvrté iteraci.



CVIČENÍ

1.1. (a) Nalezněte instanci úlohy batohu, ve které existuje více optimálních řešení. (b) Existuje instance úlohy batohu, která nemá žádné přípustné řešení? (c) Nalezněte instanci úlohy batohu s položkou, která nepatří do žádného přípustného řešení.

1.2. Dokažte, že cena řešení vráceného algoritmem GREEDY-KNAPSACK je vždy větší nebo rovna polovině kapacity. S pomocí tohoto tvrzení dokažte, že cena řešení vráceného GREEDY-KNAPSACK je nejhůře dvakrát menší než cena optimálního řešení.

1.3. Doplňte implementační detaily k algoritmu GREEDY-KNAPSACK tak, aby jeho složitost byla $O(n \log n)$.

1.4. Z rovnosti $\sum_{v \in V} \deg(v) = 2|V| - 2$ použité v důkazu Věty 1 plyne nejenom existence vrcholu stupně 1, ale i pravidlo o závislosti počtu vrcholů se stupněm 1 na počtu vrcholů s vyšším stupněm. Toto pravidlo formulujte.

1.5. Formulujte a dokažte analogii Věty 1 pro nesouvislé grafy. (*Nápověda: důležitý pojem je komponenta*).

1.6. Navrhněte algoritmus, který spočítá počet komponent v grafu. Algoritmus by měl pracovat v lineárním čase vzhledem k počtu hran v grafu.

1.7. Je ASCII tabulka prefixový kód?

1.8. Dokažte, že každý prefixový kód je jednoznačně dekódovatelný.

1.9. Stromovou reprezentaci prefixového kódu lze výhodně využít při dekódování kódového slova. Najděte algoritmus dekódování, který tak činí.

1.10. Dokažte Větu 3 (i).