NÁVRH ALGORITMŮ

poznámky pro Algoritmickou matematiku 3

Petr Osička

Univerzita Palackého
v Olomouci

Text obsahuje poznámky ke kurzu Algoritmická matematika 3.

Petr Osička
Olomouc, podzim 2017

Obsah

1 Úvod

Algoritmický problém

Definice 1. *Abeceda* Σ je konečná neprázdná množina. Prvkům Σ říkáme symboly. *Slovo (řetězec)* nad abecedou Σ je uspořádaná sekvence znaků z Σ. Délka slova $x \in \Sigma$, značeno $|x|$, je délka této sekvence. Množinu všech slov nad abecedou Σ označujeme jako Σ^*. Jazyk L nad abecedou Σ je libovolná podmnožina Σ^*.

Příklad 1. Mějme $\Sigma = \{0, 1\}$. Sekvence 0001, 1110, 00 jsou slovy nad touto abecedou, sekvence 12, #43k nejsou. Σ^* je pak množina $\{0, 1, 00, 01, 10, 11, 000, \dots\}$.

Definice 2. Nechť Σ je abeceda. Pak *algoritmický problém* definujeme jako dvojici $\langle L, R\rangle$, kde $L \subseteq \Sigma^*$ je množina vstupů (instancí)[1] a R je relace zachycující vztah mezi instancemi a správnými výsledky. Tedy pokud $\langle x, y\rangle \in R$, pak y je správným řešením x. Formálně je $L \subseteq \Sigma^*$ množina zakódování instancí pomocí řetězců nad abecedou Σ, a $R \subseteq \Sigma^* \times \Sigma^*$, výsledky také kódujeme pomocí Σ.

Pro ukázku si problém testování prvočíselnosti uvedeme ve verzích bez kódování pomocí řetězců i s jeho kódováním.

(a) Bez použití řetězců můžeme problém zavést následovně. Množina instancí je dána $\{n \mid n \in \mathbb{N}, n \geqslant 1\}$, správné řešení je 1 (odpověď *ano*), pokud je n prvočíslo, jinak 0 (odpověď *ne*).

(b) Zavedeme-li kódování nad abecedou $\{0, 1\}$ takové, že zakódováním n je jeho zápis ve dvojkové soustavě, pak je problém popsán pomocí dvojice $\langle L, R\rangle$ takové, že:

$$L = \{x \in \{0, 1\}^* \mid x \text{ je zápis čísla } n \text{ ve dvojkové soustavě}, n \in \mathbb{N}, n \geqslant 1\}$$

a pro každý řetězec $x \in L$ máme, že $\langle x, 1\rangle \in R$ když x je zakódováním prvočísla, a $\langle x, 0\rangle \in R$ pokud x není zakódováním prvočísla.

Proč kódovat vstupy a výstupy?

Velikost instance můžeme zavést jako délku zakódovaní dané instance. Můžeme tak zachytit složitost algoritmu, která by jinak zůstala skrytá.

Naivní algoritmus pro test prvočíselnosti (ten, který zkouší dělitelnost všemi menšími čísly) má pro případ (a) z předchozího příkladu, kde bereme za velikost čísla n opět n, lineární složitost. V případě zakódování z příkladu (b) má algoritmus složitost exponenciální, protože délka zákódování n je $\lceil \lg n \rceil$.

[1] Připouštíme tedy, že může existovat řetězec, který nekóduje žádnou vstupní instanci. To pro nás nebude žádný problém. Snadno lze zařídit, aby každý řetězec kódoval nějaký vstup: řekneme, že původně nesmyslné řetězce kódují jednu specifickou instanci, např. nějakou triviální.

Pokud se v instanci problému vyskytuje množina čísel, bereme často jako velikost instance jejich počet. Zanedbáváme tak ovšem velikost těchto čísel. Některé operace, o kterých při analýze algoritmu předpokládáme, že mají konstantní složitost, mají ve skutečnosti pro velká čísla složitost horší (například sčítání, porovnávání na současném harwaru). Pokud zavedeme velikost instance jako délku zakódování, můžeme do složitosti zahrnout i složitost takových operací, protože větší čísla budou mít delší zakódování.

Typy problémů

Definice 3. Problém $\langle L, R\rangle$ je *rozhodovacím problémem*, pokud je množina všech možných výsledků dvouprvková, tj $|\{y \mid (x, y) \in R \text{ pro } x \in L\}| = 2$.

Rozhodovací problém je problémem zjištění, jestli daná instance má požadovanou vlastnost. Jeden z možných výsledků pak považujeme za kladnou odpověď, druhý za zápornou odpověď. Protože R je v tomto případě jednoduché, můžeme problém reprezentovat pomocí jazyka. Slova, která kódují instance, pro které je odpověď kladná, do jazyka patří, slova, která kódují instance, pro které je odpověď záporná, do jazyka nepatří. [2]

Příklad 2. (a) Problém testování prvočíselnosti je rozhodovací problém.

(b) Typickým rozhodovacím problémem je problém splnitelnosti formule výrokové logiky v konjunktivní normální formě (CNF). Formule je v CNF, pokud je konjunkcí *klausulí*. Klausule je disjunkcí *literálů*, z nichž každý je buď výrokovou proměnnou nebo její negací. Formule $(x \vee y) \wedge (y \vee z)$ je v CNF, kdežto formule $(x \wedge y) \vee (x \to y)$ není v CNF. Každou formuli výrokové logiky lze převést do CNF. Řekneme, že formule φ je splnitelná, pokud existuje ohodnocení výrokových proměnných takové, že φ je pravdivá. Například formule $(x \vee y)$ je splnitelná, protože pro ohodnocení $x = 1, y = 1$ je pravdivá. Oproti tomu formule $(x \wedge \neg x)$ splnitelná není, protože není pravdivá pro žádné ohodnocení proměnných. Problém splnitelnosti formulí výrokové logiky, který označujeme jako SAT (z anglického satisfiability), je problémem určení toho, zda je formule splnitelná. Problém je tedy určen jazykem {zakódování φ | φ je splnitelná formule v CNF}.

Definice 4. *Optimalizační problém* je čtveřice $\langle L, \mathtt{sol}, \mathtt{cost}, \mathtt{goal}\rangle$, kde

- $L \in \Sigma^*$ je množina instancí daného problému,
- $\mathrm{sol} : L \to \mathcal{P}(\Sigma^*)$ je zobrazení přiřazující instanci problému množinu jejích přípustných řešení;
- $\mathrm{cost} : L \times \mathcal{P}(\Sigma^*) \to \mathbb{Q}$ je zobrazení přiřazující každé instanci a přípustnému řešení této instance jeho cenu;
- goal je buď minimum (pak mluvíme o minimalizačním problému) nebo maximum (pak mluvíme o maximalizačním problému).

Řekneme, že $y \in \mathrm{sol}(x)$ je *optimálním řešením* instance x, pokud

$$\mathrm{cost}(x, y) = \mathrm{goal}\{\mathrm{cost}(z, x) \mid z \in \mathrm{sol}(x)\}.$$

[2] V tichosti předpokládáme, že každý řetězec kóduje nějaký vstup, viz poznámka u definice algoritmického problému

Optimalizační problém je problémem výběru řešení s nejlepší cenou z množiny přípustných řešení dané instance.

Příklad 3. *Úloha batohu:* je dána množina položek, každá s přirozenou vahou, a kapacita batohu. Chceme nalézt takovou podmnožinu položek, která se do batohu vejde (součet vah těchto položek není větší než kapacita batohu). Současně chceme batoh zaplnit co nejvíce, tedy hledáme takovou množinu položek, která vede k největšímu možnému součtu (menšímu než kapacita).

Úloha batohu	
Instance:	$\{(b, w_1, \ldots, w_n) \mid b, w_1, \ldots, w_n \in \mathbb{N}\}$
Přípustná řešení:	$\mathrm{sol}(b, w_1, \ldots, w_n) = \{C \subseteq \{1, \ldots, n\} \mid \sum_{i \in C} w_i \leqslant b\}$
Cena řešení:	$\mathrm{cost}(C, (b, w_1, \ldots, w_n)) = \sum_{i \in C} w_i$
Cíl:	maximum

Pro instanci $I = (b, w_1, \ldots, w_5)$, kde $b = 29, w_1 = 3, w_2 = 6, w_3 = 8, w_4 = 7, w_5 = 12$, existuje jediné optimální řešení $C = \{1, 2, 3, 5\}$ s cenou $\mathrm{cost}(C, I) = 29$. Lze snadno ověřit, že všechna ostatní přípustná řešení mají menší cenu.

U optimalizačních problémů, pro které neznáme efektivní algoritmus pro nalezení optimálního řešení, se často spokojíme s efektivním algoritmem, který vrací řešení přípustné, ne nutně optimální. Takový algoritmus nazýváme *aproximačním algoritmem*.

Časová složitost

Časová složitost (dále jen složitost)[3] algoritmu A je funkce $\mathrm{Time}_A : \mathbb{N} \to \mathbb{N}$ přiřazující velikosti vstupní instance počet instrukcí, které musí algoritmus A během výpočtu provést. Protože obecně nemusí algoritmus pro dvě různé stejně velké instance provést stejné množství instrukcí, potřebujeme počty instrukcí pro instance stejné velikosti nějakým způsobem agregovat.

Definice 5. Označme si $t_A(x)$ počet instrukcí, které algoritmus A provede pro instanci $x \in L$. Časová složitost v *nejhorším případě* je definovaná

$$\mathrm{Time}_A(n) = \max\{t_A(x) \mid x \in L, |x| = n\}.$$

Složitost v průměrném případě je pak

$$\mathrm{Time}_A(n) = \frac{\sum_{x \in L, |x|=n} t_A(x)}{|\{x \mid |x| = n\}|}.$$

Jedním z využití složitosti (mimo zřejmé využití k porovnání algoritmů) je rozdělení problémů na prakticky řešitelné a na praktický neřešitelné. Jak ovšem přiřadit složitost k problému?

Definice 6. Nechť U je algoritmický problém.

[3] Analogické úvahy bychom mohli provádět i pro paměťovou složitost.

- $O(g(n))$ je *horním omezením složitosti* problému U, pokud existuje algoritmus A řešící U takový, že $\mathrm{Time}_A(n) \in O(g(n))$.
- $\omega(f(n))$ je *dolním omezením složitosti* problému U, pokud pro každý algoritmus B řešící U platí, že $\mathrm{Time}_B(n) \in \omega(f(n))$

Za horní omezení složitosti problému bereme složitost nejlepšího známého algoritmu. Nalézt dolní omezení složitosti je oproti tomu mnohem komplikovanější. Musíme totiž vyloučit existenci algoritmu se složitostí danou dolní hranicí nebo lepší. To je netriviální a pro naprostou většinu problémů je taková hranice neznámá (vyloučíme-li například hranici danou tím, že algoritmus musí přečíst celý vstup). Příkladem problému, pro který je tato hranice známá je například třídění porovnáváním, pro které neexistuje algoritmus se složitostí lepší než $O(n \log n)$.

Problémy rozdělujeme na prakticky řešitelné a prakticky neřešitelné pomocí jejich horního omezení složitosti. Rozdělení je následující

– problém považujeme za praktický řešitelný, pokud pro je jeho horní omezení složitost $O(p(n))$, kde $p(n)$ je polynom. Tedy, problém je praktický řešitelný, pokud pro něj existuje algoritmus s nejhůře polynomickou složitostí.

– ostatní problémy považujeme za prakticky neřešitelné.

Předchozí rozdělení je nutno brát s určitou rezervou. Například pokud pracujeme s instancemi malé velikosti, lze považovat i algoritmus, který má exponenciální složitost (nebo jinou horší než polynomickou složitost), za praktický. V asymptotické notaci jsou navíc ukryty multiplikativní konstanty. Může se tedy stát, že pro instance malé velikosti je algoritmus s polynomickou složitostí méně efektivní, než ten s exponciální složitostí.

Proč polynomická složitost?

Polynom je největší funkce, která „neroste rychle.“ U tohoto důvodu předpokládáme, že stupeň polynomu nebude extrémně velký (např. 100). U většiny známých algoritmů s polynomickou složitostí, není stupeň polynomu vyšší než 10. Druhým důvodem je to, že polynomy jsou uzavřené vzhledem k násobení (vynásobením dvou polynomů dostaneme zase polynom). Pokud uvnitř algoritmu A spouštíme nejvýše polynomický počet krát algoritmus B s nejhůře polynomickou složitostí, má tento algoritmus také nejhůře polynomickou složitost (přitom samozřejmě předpokládáme, že pokud bychom považovali složitost algoritmu B při analýze A za konstantu, bude složitost A mít nejhůře polynomickou složitost). V podstatě tedy můžeme efektivní algoritmy skládat do sebe (analogicky volání funkcí při programování) a získaný algoritmus bude také efektivní.

Asymptotická notace

Definice 7. $O(f(n))$ je množina všech funkcí $g(n)$ takových, že existují kladné konstanty c a n_0 takové, že $0 \leqslant g(n) \leqslant cf(n)$ pro všechna $n \geqslant n_0$.

$\Omega(f(n))$ je množina všech funkcí $g(n)$ takových, že existují kladné konstanty c a n_0 takové, že $g(n) \geqslant cf(n) \geqslant 0$ pro všechna $n \geqslant n_0$.

$\Theta(f(n))$ je množina všech funkcí $g(n)$ takových, že existují kladní konstanty c_1, c_2 a n_0 takové, že $0 \leqslant c_1 f(n) \leqslant g(n) \leqslant c_2 f(n)$ všechna $n \geqslant n_0$.

Zápis $g(n) = O(f(n))$ je příkladem tzv. jednostranné rovnosti (v následujících úvahách lze zaměnit Ω a Θ za O). Rovnítko v ní interpretujeme jinak, než je obvyklé (což je zřejmé, na obou stranách rovnosti nejsou objekty stejného typu). Korektně by měl být vztah zapsán jako $g(n) \in O(f(n))$, nicméně použití rovnítka se již zažilo a stalo se de facto standardem.

Pro pochopení významu rovností, ve kterých se asymptotická notace nachází na levé i na pravé straně, si stačí uvědomit, že výraz, ve kterém se tato notace nachází představuje množinu funkcí. Tedy například výraz $n^3 + O(n)$ je množina

$$\{n^3 + f(n) \mid f(n) \in O(n)\}.$$

Protože na obou stranách $=$ jsou množiny a jedná se o jednosměrnou rovnost, interpretujeme $=$ jako $\subseteq$. Pro aritmetiku s množinami funkcí platí intuitivní pravidla. Uvážíme-li množiny funkcí S a T, pak platí

$$\begin{aligned} S + T &= \{g + h \mid g \in S, h \in T\}, \\ S - T &= \{g - h \mid g \in S, h \in T\}, \\ S \cdot T &= \{g \cdot h \mid g \in S, h \in T\}, \\ S \div T &= \{g \div h \mid g \in S, h \in T\}. \end{aligned}$$

Skládání funkce s množinou funkcí provedeme obdobně,

$$f(O(g(n))) = \{f(h(n)) \mid h \in O(g(n))\}.$$

Například $\log O(n^2)$ je množina logaritmů z funkcí omezených shora kvadrikou. Pokud jsou asymptotické výrazy vnořeny, pak množiny sjednotíme, tedy například

$$O(g(n) + O(f(n))) = \bigcup \{O(h) \mid h \in g(n) + O(f(n))\}.$$

2 Rozděl a panuj

Základní kostra algoritmu pro vstup I je následující:

1. Pokud je $|I| < b$ pro pevně danou konstantu b, je výsledek triviální nebo jej počítáme jiným algoritmem. Algoritmus A tento výsledek jenom vrátí.
2. Z instance I vygenerujeme instance $I_1, I_2, \ldots, I_k$, takové, že $I > |I_i|$ pro všechna $1 \leqslant i \leqslant k$.
3. Rekurzivně zavoláme algoritmus A pro $I_1, I_2, \ldots, I_k$.
4. Výsledky rekurzivních zavolání zkombinujeme do výsledku aktuálního zavolání a tento výsledek vrátíme.

V algoritmech realizujících techniku Rozděl a panuj můžeme rozpoznat dvě oddělené fáze, které ji dali název. Během první fáze, které říkáme *Rozděl*, algoritmus vygeneruje množinu menších instancí. Během druhé fáze, pojmenované jako *Panuj*, sestavíme ze řešení vypočtených pro menší instance řešení instance původní. Příkladem algoritmu používající strategii rozděl a panuj je MergeSort (který již všichni znáte).

Algoritmus 1 Mergesort

```
1:  procedure MergeSort(A, l, p)
2:      if p < l then
3:          q ← ⌊(l + p)/2⌋                ▷ rozděl
4:          MergeSort(A, l, q)
5:          MergeSort(A, q + 1, p)
6:      Merge(A, l, q, p)                  ▷ panuj
```

```
1:  procedure Merge(A, l, q, p)
2:      n1 ← q − l + 1
3:      n2 ← r − q
4:      for i ← 0 to n1 − 1 do
5:          L[i] ← A[l + 1]
6:      for i ← 0 to n1 − 1 do
7:          R[i] ← A[q + i + 1]
8:      L[n1] ← R[n2] ← ∞
9:      i ← j ← 0
10:     for k ← l to p do
11:         if L[i] ⩽ R[j] then
12:             A[k] ← L[i]
13:             i ← i + 1
14:         else
15:             A[k] ← R[j]
16:             j ← j + 1
```

Analýza časové složitosti

Díky rekurzivním zavoláním se složitost nedá vyjádřit přímo, ale musíme ji vyjádřit pomocí rekurence. Označíme-li tuto složitost $T(n)$, pak

$$T(b) = \Theta(1),$$
$$T(|I|) = \sum_{i=1}^{k} T(|I_k|) + f(|I|).$$

Pro instanci o velikosti menší nebo rovno b je složitostí algoritmu konstanta (první řádek kostry algoritmu). Pro zbylé instance algoritmus spočítá instance menší (řádek 2) a kombinuje výsledky rekurzivních zavolání (řádek 4), složitost těchto řádků je vyjádřena funkcí $f(n)$. Abychom spočetli složitost celého algoritmu, musíme ještě přičíst sumu složitostí rekurzivních zavolání, tedy sumu $\sum_{i=1}^{k} T(|I_k|)$.

Složitost chceme vyjádřit v uzavřené formě, to je ve formě, která neobsahuje rekurentní „volání sebe sama“ na pravé straně rovnosti (hledání uzavřené formy se říká řešení rekurence). Rekurence můžeme vyřešit přesně, nebo–pokud je přesné řešení obtížné– se spokojíme s odhadem uzavřené formy. V takovém případě se spokojíme s výsledkem v asymptotické notaci.

Snadné rekurence

Postup: Uhodni uzavřenou formu a dokaž správnost indukcí.

Příklad 4 (Hanojské věže).

$$T(n) = 2T(n-1) + 1$$
$$T(1) = 1$$

Hodnoty pro prvních pár čísel

n	T(n)
1	1
2	3
3	7
4	15

Tipneme, že $T(n) = 2^n - 1$, pokusíme se výsledek dokázat indukcí. Pro $n = 1$ rovnost triviálně platí. Předpokládejme, že platí pro $n-1$. Pak

$$T(n) = 2T(n-1) - 1 = 2(2^{n-1} - 1) = 2^n - 1.$$

Příklad 5 (Krájení pizzy).

$$T(0) = 1$$
$$T(n) = T(n-1) + n$$

Rekurenci rozvineme:

$$\begin{aligned} T(n) &= T(n-1) + n = \\ &= T(n-2) + (n-1) + n = \\ &\vdots \\ &= 1 + 1 + 2 + \cdots + n = \\ &= 1 + \frac{n(n+1)}{2} \end{aligned}$$

Výsledek ještě dokážeme indukcí. Pro $n = 0$ rovnost triviálně platí. Předpokládejme, že platí pro $n - 1$, pak

$$T(n) = T(n-1) + n = \frac{n(n-1)}{2} + 1 + n = \frac{n(n+1)}{2} + 1$$

Substituční metoda

Postup pro odhad pomocí O notace (pro Ω notaci je postup analogický):

1. Odhadneme uzavřenou formu pomocí asymptotické notace, tj. že uzavřená forma patří do $O(g(n))$ pro nějakou funkci $g(n)$.
2. Vybereme jednu funkci $f(n)$ z asymptotického odhadu, při specifikaci funkce můžeme využít konstant. Pro tuto funkci indukcí dokážeme, že $T(n) \leqslant f(n)$, čímž dokážeme, že $T(n) = O(g(n))$. Během důkazu lze zvolit vhodné hodnoty konstant.

Příklad 6.

$$\begin{aligned} T(n) &= T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1 \\ T(1) &= 1 \end{aligned}$$

Odhadneme jako $O(n)$. Vybereme funkci $cn - b = O(n)$, $c > 0$, b jsou konstanty. Chceme dokázat $T(n) \leqslant cn - b$ pro nějaké hodnoty konstant b, c. To uděláme indukcí. Indukční předpoklady jsou:

$$\begin{aligned} T(\lfloor n/2 \rfloor) &\leqslant c(\lfloor n/2 \rfloor) - b \\ T(\lceil n/2 \rceil) &\leqslant c(\lceil n/2 \rceil) - b \end{aligned}$$

Dokážeme nerovnost pro $T(n)$.

$$\begin{aligned} T(n) &= T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1 \\ &\leqslant c(\lfloor n/2 \rfloor) - b + c(\lceil n/2 \rceil) - b + 1 \\ &\leqslant cn - 2b + 1 = \\ &= cn - b + (-b + 1) \end{aligned}$$

Zvolíme $b = 1$ a tím dostaneme $T(n) \leqslant cn - 1$, což jsme chtěli dokázat. Zbývá ověřit případ, kdy $n = 1$. Nerovnost

$$T(1) = 1 \leqslant cn - 1$$

platí pro jakéhokoliv $c \geqslant 2$. Dohromady jsme dokázali, že $T(n) \leqslant 2n - 1$ a tedy $T(n) = O(n)$.

Příklad 7.

$$\begin{aligned} T(n) &= 2T(\lfloor n/2 \rfloor) + n \\ T(1) &= 1 \end{aligned}$$

Odhadneme je $O(n \lg n)$. Pokud vybereme funkci $cn \lg n$ můžeme dokázat správnost obecného indukčního kroku pro $c \geqslant 1$

$$\begin{aligned} T(n) &\leqslant 2(c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor) + n \\ &\leqslant cn \lg(n/2) + n \\ &= cn \lg n - cn \lg 2 + n \\ &\leqslant cn \lg n \end{aligned}$$

U okrajové podmínky ovšem narazíme na problém.

$$T(1) = 1 \neq 1 \lg 1 = 0$$

Můžeme využít toho, že dokazujeme asymptotický odhad a nerovnost proto může platit až od určitého n_0. Pro $n > 3$ už $T(n)$ nezávisí na $T(1)$, ale stačí znát $T(2)$ a $T(3)$. Volbou $n_0 = 2$ můžeme „posunout“okrajovou podmínku směrem nahoru (čímž případ $n = 1$, který způsobuje problémy, z důkazu vypustíme) a nahradit okrajovou podmínku pomocí

$$\begin{aligned} T(2) &= 4, \\ T(3) &= 5, \end{aligned}$$

s tím, že nyní musí být $c \geqslant 2$.

Odhad pomocí stromu rekurze

Metoda odhadu pomocí stromu rekurze je založená na jednoduché myšlence. Rekurentní vztah

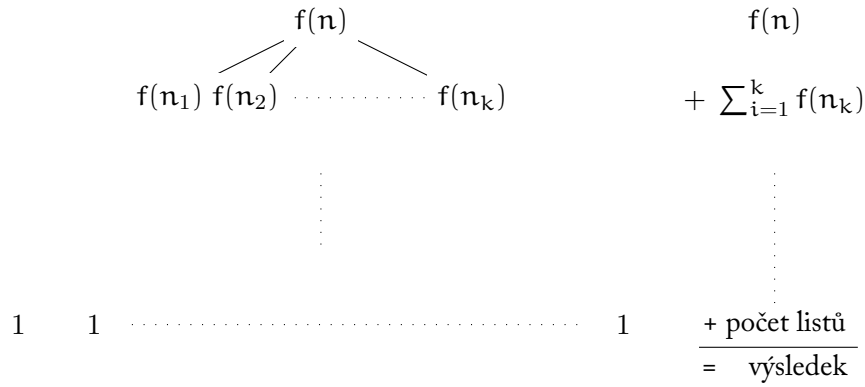
$$T(n) = \sum_{i=1}^{k} T(n_i) + f(n).$$

si představíme jako strom. Uzly stromu odpovídají jednotlivým rekurzivním „voláním.“ Každému uzlu přiřadíme množství práce, kterou pomyslný algoritmus při daném volání provede. Listové uzly tak budou mít přiřazenu konstantu, která odpovídá okrajovým podmínkám rekurentního vztahu. Množství práce vnitřních uzlů pak odpovídá aplikaci funkce f na argument daného rekurzivního volání. V kořenu tedy provedeme $f(n)$ práce, v jeho i-tém potomku pak $f(n_i)$. Přirozeně, pokud sečteme práci přiřazenou všem uzlům, dostaneme řešení rekurence. Součet provedeme ve dvou krocích. Nejdříve pro každou úroveň stromu sečteme práci provedenou v uzlech na oné úrovni, poté sečteme sumy z jednotlivých vrstev.

Příklad 8.

$$\begin{aligned} T(n) &= 2T(n/4) + n^2 \\ T(1) &= 1 \end{aligned}$$

Začněme konstruovat strom rekurze. Kořenu přiřadíme n^2. Potomkům kořene, kteří odpovídají $T(n/4)$ přiřadíme $(n/4)^2$. Uzlům v další vrstvě, odpovídajícím $T(n/16)$ (rekurzivní volání s argumentem $n/4$) přiřadíme $(n/16)^2$. V první vrstvě se nachází 2 uzly, suma



Obrázek 2.1: Idea metody odhadu pomocí stromu rekurze

práce je tedy $2(n/4)^2$, v druhé vrstvě se nachází 4 uzly, suma je tedy $4(n/16)^2$. Nyní si můžeme všimnout určité pravidelnosti. Práce, kterou provedeme v jednom uzlu v i-té vrstvě (kořen je v nulté vrstvě) odpovídá $(n/4^i)^2$, počet uzlů v i-té vrstvě je 2^i, suma přes všechny uzly v této vrstvě je tedy $2^i(n/4^i)^2 = n^2/8^i$.

Listy se nacházejí ve vrstvě $\log_4 n$, jejich počet je tedy $2^{\log_4 n} = n^{1/2}$. Odvození tohoto: víme, že $\log_2 n = \frac{\log_4 n}{\log_4 2}$ a tedy $\log_4 n = \log_2 n^{\log_4 2}$. Dosadíme do $2^{\log_4 n}$ a dostaneme $2^{\log_2 n^{\log_4 2}} = n^{\log_4 2}$.

Pokud nyní sečteme všechny vrstvy, dostaneme

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_4 n-1} (n^2/8^i) + n^{1/2} \\ &= n^2 \sum_{i=0}^{\log_4 n-1} (1/8^i) + n^{1/2} \end{aligned}$$

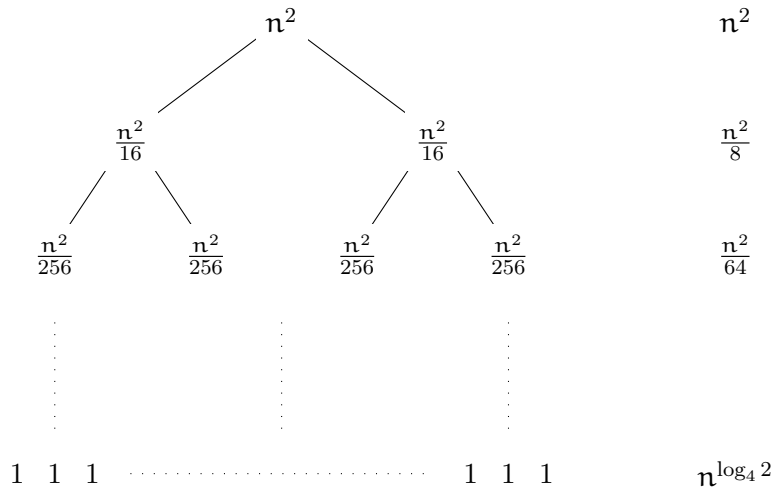
Abychom se zbavili závislosti na n v sumě, nahradíme ji sumou celé geometrické posloupnosti. Dostaneme tedy

$$\begin{aligned} T(n) &= n^2 \sum_{i=0}^{\log_4 n-1} (1/8^i) + n^{1/2} \\ &\leqslant n^2 \sum_{i=0}^{\infty} (1/8^i) + n^{1/2} \\ &= n^2 \frac{1}{1-1/8} + n^{1/2} \end{aligned}$$

Odhad řešení rekurence je $O(n^2)$.

Master theorem

Pečlivé vyřešení metody stromu pro specifický druh rekurencí dává následující větu. Porovnáváme v ní rychlost růstu stromu ($n^{\log_b a}$ je počet listů) s rychlostí růstu funkce $f(n)$ (ze



Obrázek 2.2: Strom rekurze pro rekurenci $T(n) = 2T(n/4) + n^2$

znění věty).

Věta 1. *Nechť* $a \geqslant 1$ *a* $b \geqslant 1$ *jsou konstanty a* $f(n)$ *je funkce a* $T(n)$ *je definovaná na nezáporných celých číslech pomocí rekurence*

$$T(n) = aT(n/b) + f(n),$$

přičemž n/b *interpretujeme jako* $\lfloor n/b \rfloor$ *nebo* $\lceil n/b \rceil$. *Pak můžeme* $T(n)$ *následovně asymptoticky omezit.*

1. *Pokud* $f(n) = O(n^{\log_b a - \epsilon})$ *pro nějaké* $\epsilon > 0$, *pak* $T(n) = \Theta(n^{\log_b a})$.
2. *Pokud* $f(n) = \Theta(n^{\log_b a})$, *pak* $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. *Pokud* $f(n) = \Omega(n^{\log_b a + \epsilon})$ *pro nějaké* $\epsilon > 0$ *a pokud* $af(n/b) \leqslant cf(n)$ *pro konstantu* $0 < c < 1$ *a všechna dostatečně velká* n, *pak* $T(n) = \Theta(f(n))$.

Rychlé násobení velkých čísel

Jsou dána dvě čísla A a B (v binárním zápisu, celý postup lze upravit pro zápis čísel v jakékoliv soustavě), a chceme spočítat AB. Algoritmus násobení pod sebe ze základní školy má složitost $\Omega(n^2)$.

Předpokládáme poziční reprezentaci kladných čísel ve dvojkové soustavě. Číslo A je reprezentováno sekvencí bitů $a_{n-1}a_{n-2}\ldots a_0$ pokud platí, že $A = \sum_{i=0}^{n} a_i 2^i$. V dalším předpokládejme, že n je mocninou 2. Uvažme čísla A_1 dané sekvencí bitů $a_{n-1}\ldots a_{n/2}$ a A_2 dané sekvencí bitů $a_{n/2-1}\ldots a_0$. Pak platí, že $A = A_1 \cdot 2^{n/2} + A_2$. Této vlastnosti můžeme využít a zapsat součin čísel A a B (kde B_1 a B_2 je definována analogicky číslům A_1 a A_2)

$$\begin{aligned} AB &= (A_1 \cdot 2^{n/2} + A_2)(B_1 \cdot 2^{n/2} + B_2) \\ &= A_1B_1 \cdot 2^n + (A_1B_2 + B_1A_2) \cdot 2^{n/2} + A_2B_2 \end{aligned}$$

Mohli bychom navrhnout rekurzivní algoritmus, který pro n-bitová čísla A, B určí A_1,A_2, B_1, B_2 (to jsou všechno $(n/2)$-bitová čísla), rekurzivně spočítá součiny A_1B1, A_1B_2, A_2B_1,

A_2B_2 a s jejich pomocí pak spočte výsledek podle předchozího vztahu. Rekurzi ukončíme v momentě, kdy násobíme jednobitová čísla. Protože násobení mocninou dvou můžeme realizovat pomocí bitového posuvu, který má lineární složitost, a sčítání čísel má také lineární složitost, je složitost navrhovaného algoritmu dána rekurencí

$$\begin{aligned} T(n) &= 4T(n/2) + O(n) \\ T(1) &= O(1), \end{aligned}$$

jejímž řešením je $\Theta(n^2)$. Abychom dosáhli algoritmu s menší složitostí, musíme ubrat rekurzivní zavolání. To můžeme provést díky následující úvaze:

$$\begin{aligned} A_1B_2 + A_2B_1 &= A_1B_2 + A_2B_1 + A_1B_1 - A_1B1 + A_2B2 - A2B2 \\ &= A_1B_1 + A_2B_2 + A_1(B_2 - B_1) + A_2(B_2 - B_1) \\ &= A_1B_1 + A_2B_2 + (A_1 - A_2)(B_2 - B_1), \end{aligned}$$

a proto

$$AB = A_1B_1 \cdot 2^n + [A_1B_1 + A_2B_2 + (A_1 - A_2)(B_2 - B_1)] \cdot 2^{n/2} + A_2B_2$$

Nyní stačí jenom tři rekurzivní zavolání, musíme spočítat A_1B_1, A_2B_2 a $(A_1-A_2)(B_2-B_1)$. Složitost algoritmu tak klesne na $O(n^{\log_2 3})$. Vzniklo nebezpečí, že budeme muset násobit záporná čísla. To lze ale snadno obejít tím, že si nejdříve spočítáme znaménko výsledku, a pak vynásobíme absolutní hodnoty čísel. Operace $\text{shift}(x, n)$ v algoritmu implementuje součin x a 2^n pomocí bitového posuvu a má lineární složitost. Stejně tak i výpočet absolutní hodnoty čísel (např. při reprezentaci záporných čísel doplňkovým kódem) a sčítání (nebo odečítání) čísel. Všimněme si také, že součet na řádku 14 musí být vždycky kladný, protože $\text{shift}(m_1, n)$ je vždy kladný a navíc je řádově (o 2^n) větší než potenciálně záporné číslo $\text{shift}(m_1 + m_2 + m_3, n/2)$.

Algoritmus 2 Násobení velkých čísel

```
 1: procedure MULTIPLY(A,B,n)
 2:     s ← sign(X) · sign(Y)
 3:     A ← abs(X)
 4:     B ← abs(Y)
 5:     if n = 1 then
 6:         return X · Y · s
 7:     A1 ← číslo tvořené n/2 levými bity X
 8:     A2 ← číslo tvořené n/2 pravými bity X
 9:     B1 ← číslo tvořené n/2 levými bity Y
10:     B2 ← číslo tvořené n/2 pravými bity Y
11:     m1 ←Multiply(A1, B1, n/2)
12:     m2 ←Multiply(A1 − A2, B2 − B1, n/2)
13:     m3 ←Multiply(A2, B2, n/2)
14:     return s · (shift(m1, n) + shift(m1 + m2 + m3, n/2) + m3)
```

Rychlé násobení polynomů

Součin funkcí f a g je funkce $f \cdot g$ definovaná jako $(f \cdot g)(x) = f(x)g(x)$ pro každé x.

Polynom je funkce $A(x) = a_0 + a_1 x + a_2 x^2 + \ldots$. Číslům $a_0, a_1, \ldots$ říkáme koeficienty. Exponent nejvyšší mocniny mezi členy s nenulovým koeficientem je stupeň polynomu. Při zápisu polynomu můžeme všechny jeho členy s vyššími mocninami než je jeho stupeň vynechat.

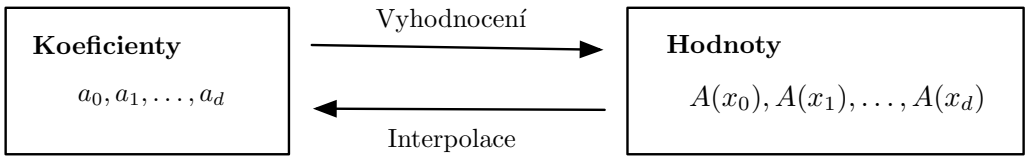
Součin $AB(x)$ polynomů je definován jako součin funkcí. Algoritmus pro násobení polynomů roznásobením závorek má složitost $\Omega(n^2)$, kde n je maximum ze stupňů polynomů. Polynomy totiž mají maximálně $n + 1$ členů a násobíme každý člen s každým.

Reprezentace polynomu

Polynom můžeme reprezentovat pomocí posloupnosti jeho koeficientů, tj. polynom $A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_d x^d$ je lze reprezentovat jako $[a_0, a_1, a_2 \ldots a_d]$, přičemž napravo můžeme přidat libovolný počet nul, například $[a_0, a_1, a_2 \ldots a_d, 0, 0, 0]$ je také reprezentací polynomu $A(x)$. Alternativní reprezentace polynomu je dána následující větou.

Věta 2. *Polynom stupně d je jednoznačně reprezentován svými hodnotami v alespoň $d+1$ různých bodech.*

Polynom lze tedy reprezentovat i hodnotami tohoto polynomu v potřebném počtu různých bodů. Tento počet musí být alespoň tak velký jako je počet koeficientů daného polynomu v jeho nejúspornější reprezentaci (kde je počet koeficientů roven stupni polynomu zvětšenému o jedna), ale může být větší. Mezi oběma reprezentacemi můžeme přecházet (později uvidíme jak).



Příklad 9. Polynom $x^3 + x + 2$ je reprezentován pomocí koeficientů jako $[2, 1, 0, 3]$. Pro reprezentaci pomocí hodnot musíme zvolit alespoň 4 různé body, zvolme tedy $1, 2, 3, 4$. Reprezentace hodnotami potom je $[4, 12, 32, 70]$.

Máme-li polynomy A a B reprezentovány hodnotami ve *stejných* bodech, můžeme získat reprezentaci polynomu AB tak, že reprezentace polynomu A a B po složkách vynásobíme. Tato operace má lineární složitost. Musíme si ovšem dát pozor na to, aby výsledná reprezentace měla dostatečný počet položek. Tedy, pokud má A stupeň s a B stupeň t, a proto má AB stupeň $s + t$, pak podle předchozí věty potřebujeme hodnoty polynomu AB v alespoň $s + t + 1$ různých bodech. Odtud plyne, že i pro polynomy A a B musíme mít hodnoty v alespoň $t + s + 1$ bodech.

Idea algoritmu

Algoritmus pro rychlé násobení by měl pracovat následovně. Pro vstupní polynomy A a B (stupňů s a t):

1. vybereme alespoň $s + t + 1$ různých bodů a spočteme hodnoty polynomů A a B v těchto bodech.
2. pro každý bod pak vynásobíme hodnotu polynomu A v tomto bodě a hodnotu polynomu B v tomto bodě. Dostaneme tak hodnotu polynomu AB v tomto bodě.
3. z hodnot polynomu AB interpolujeme koeficienty a vrátíme je jako výsledek.

Krok 2 lze provést s lineární složitostí. Abychom dostali složitost lepší než $\Omega(n^2)$, potřebujeme provést kroky 1. a 3. se složitostí lepší než $\Omega(n^2)$.

Vyhodnocení polynomu a rychlá fourierova transformace

Problém vyhodnocení polynomu je následující: vstupem je polynom v reprezentaci pomocí n koeficientů. Výsledkem je jeho reprezentace pomocí hodnot v n různých bodech. Tytu body musí být různé, ale jiné omezení na ně neklademe.

Algoritmus, ve kterém do polynomu příslušné hodnoty dosadíme a spočítáme výsledek, má složitost $\Omega(n^2)$. Pro každý bod totiž musíme spočítat hodnoty n členů polynomu a bodů je n. Pro naše účely je tedy tento algoritmus nevyhovující.

Pro rychlý výpočet hodnot polynomu i interpolaci lze použít Rychlou Fourierovu Transformaci (FFT). V dalším budeme předpokládat, že n je mocninou dvou. To nás nijak neomezuje: (a) reprezentaci pomocí koefieficientů můžeme rozšířit přidáním nul na konec, (b) pro reprezentaci pomocí hodnot můžeme použít větší než potřebný počet bodů.

Protože body, pro které chceme hodnoty polynomu spočítat si můžeme zvolit libovolně, můžeme zvolit body

$$\pm x_0, \pm x_1, \cdots \pm x_{n/2-1}. \tag{2.1}$$

Později uvidíme, proč je taková volba výhodná.

Polynom rozdělíme na členy se sudými a s lichými mocninami

$$A(x) = (a_0 + a_2x^2 + \dots) + x(a_1 + a_3x^2 + \dots)$$

Všimneme si, že oba výrazy v závorkách jsou opět polynomy. Výraz $(a_0 + a_2x^2 + \dots)$ je polynom $A_S(z) = a_0 + a_2z + \dots$, do kterého dosadíme x^2 za z, a tedy

$$(a_0 + a_2x^2 + \dots) = A_S(x^2).$$

Stejnou úvahou dostaneme pro polynom v pravé závorce polynom $A_L(z) = a_1 + a_3z + \dots$, pro který platí

$$(a_1 + a_3x^2 + \dots) = A_L(x^2).$$

Polynomy A_S a A_L jsou reprezentovány $n/2$ koeficienty. Hodnoty polynomu A pro $\pm x_i$ pak lze vyjádřit jako

$$A(x_i) = A_s(x_i^2) + x_i \cdot A_l(x_i^2), \tag{2.2}$$

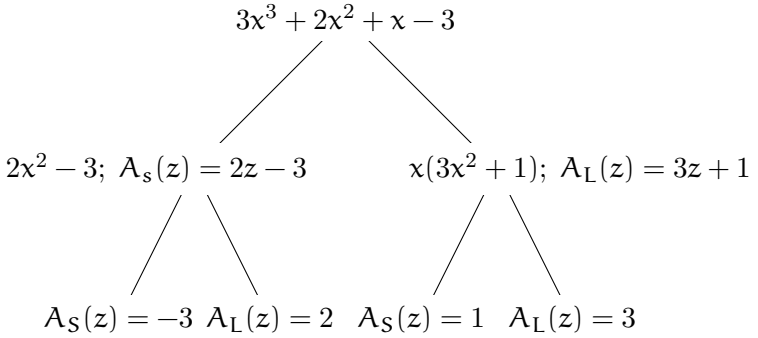
$$A(-x_i) = A_s(x_i^2) - x_i \cdot A_l(x_i^2). \tag{2.3}$$

Na vztazích vidíme, proč bylo výhodné zvolit si dvojice opačných bodů. Platí totiž, že $x^2 = (-x)^2$ a tedy i $A_S(x^2) = A_S((-x)^2)$ a $A_L(x^2) = A_L((-x)^2)$.

K výpočtu hodnot polynomu A v n bodech tedy potřebujeme vypočítat hodnoty polynomů A_S a A_L v $n/2$ bodech. Na základě této úvahy můžeme sestavit algoritmus, který pracuje následovně:

1. určíme A_S a A_L
2. rekurzivně spočítáme hodnoty polynomů A_S a A_L v bodech $x_0^2, x_1^2, \ldots, x_{n/2-1}^2$.
3. pomocí (2.2) a (2.3) spočteme hodnoty polynomu A v bodech $\pm x_0, \pm x_1, \cdots \pm x_{n/2-1}$

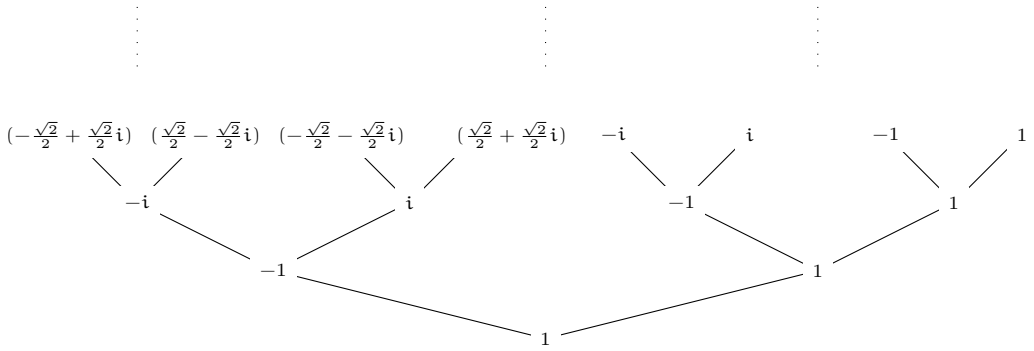
Příklad 10. Uvažujme polynom $3x^3 + 2x^2 + x - 3$. Pak algoritmus projde postupně následující strom polynomů (A_S vlevo, A_L vpravo). Body, ve kterých se hodnoty počítají ještě neuvažujeme.



Řekněme, že v předchozím příkladě budeme chtít spočítat hodnoty polynomu pro body $\pm x_0, \pm x_1$. Při rekurzivním volání pak musíme spočítat hodnoty polynomů $(2z - 3)$ a $(3z + 1)$ pro x_0^2 a x_1^2. Problém je, že x_0^2 a x_1^2 nemohou být opačná čísla, protože druhé mocniny reálných čísel jsou vždycky kladné. Abychom našli takové body, jejichž druhá mocnina je záporná, musíme přejít od reálných čísel ke komplexním.

Nejjednodušší je případ, kdy je polynom stupně 1 a hodnoty počítáme pouze pro dvojici bodů (když je polynom jenom konstanta, je jeho hodnotou vždy tato konstanta a tedy nemá smysl volit body dvojici opačných bodů). Zvolme pro tuto úroveň rekurze (tedy úroveň stromu rekurze, na které se vyskytují polynomy stupně 1) body ± 1. O úroveň výše (úroveň, na které se vyskytují polynomy stupně 3) potřebujeme 2 páry bodů, druhá mocnina jednoho páru se musí rovnat 1, druhá mocnina druhého páru se musí rovnat -1. Přirozeně tyto body jsou $\pm 1, \pm i$, kde i je imaginární jednotka. Opakováním této konstrukce získáme body, pro které potřebujeme spočítat hodnoty polynomu v kořeni stromu rekurze (viz obrázek 2.3).

Každá z úrovní stromu na obrázku 2.3 obsahuje právě všechny odmocniny z 1. Přesněji, je-li v dané úrovni n uzlů, obsahuje právě všechny $\sqrt[n]{1}$. Algoritmus FFT využívá následujících vlastností $\sqrt[n]{1}$.



Obrázek 2.3: Idea nalezení bodů pro výpočet hodnoty polynomu pomocí FFT. Potomci každého uzlu jsou jeho druhé odmocniny.

Pro $n = 2^k$ platí, že

(a) n-té odmocniny jedné jsou $\omega^0 = 1, \omega, \omega^2, \ldots, \omega^{n-1}$, kde $\omega = e^{\frac{2\pi i}{n}}$.

(b) $\omega^{\frac{n}{2}+j} = -\omega^j$

(c) Množina druhých mocnin $\sqrt[n]{1}$ jsou právě $\{1, \omega^2, (\omega^2)^2, \ldots, (\omega^2)^{n/2-1}\}$, tedy $n/2$-té odmocniny 1.

Tyto vlastnosti nám umožňují iterovat přes všechny $\sqrt[n]{1}$. Stačí spočítat ω a pak iterovat přes její mocniny. Konec iterace poznáme podle toho, že se dostaneme na hodnotu 1. Například pro $n = 4$ je $\omega = i$ a jednotlivé mocniny jsou postupně $\omega^2 = -1$, $\omega^3 = -i$, $\omega^4 = 1$. Vlastnost (b) říká, že množina všech $\sqrt[n]{1}$ tvoří opravdu dvojice opačných bodů a k jejich nalezení stačí iterovat pouze přes polovinu mocnin. Druhá polovina jsou právě opačné body. Například pro $n = 4$ máme $\omega = i$ a opačný bod je $\omega^{1+2} = \omega^3 = -i$, pro $\omega^2 = -1$ je opačný bod $\omega^{2+2} = \omega^4 = 1$. Díky (c) v rekurzivních voláních skutečně počítáme s druhými mocninami bodů.

Algoritmus 3 Rychlá Fourierova transformace

```
1:  procedure FFT(A[0, ..., n − 1], ω)          ▷ n je mocnina dvou
2:      if ω = 1 then                           ▷ V tomto případě už |A| = 1
3:          return A
4:      for i ← 0 to n/2 − 1 do
5:          A_S[i] ← A[i · 2]                   ▷ Koeficienty pro sudé mocniny
6:          A_L[i] ← A[i · 2 + 1]               ▷ Koeficienty pro liché mocniny
7:      S ← FFT(A_S, ω^2)
8:      L ← FFT(A_L, ω^2)
9:      x ← 1                                   ▷ Začínáme od ω^0
10:     for j ← 0 to n/2 − 1 do
11:         R[j] ← S[j] + x · L[j]              ▷ Rovnice (2.2)
12:         R[j + n/2] ← S[j] − x · L[j]        ▷ Rovnice (2.3)
13:         x ← x · ω                           ▷ Další mocnina ω
14:     return R
```

Složitost FFT vyjádříme pomocí rekurence. V algoritmu jsou dvě rekurzivní volání, každému z nich předáváme instanci o velikosti $n/2$. Zbývající část algoritmu (tedy cykly na

řádcích 4 až 6 a 10 az 13) má složitost $O(n)$. Rekurence je tedy

$$T(n) = 2T(n/2) + O(n)$$
$$T(1) = O(1)$$

Podle master theoremu je pak složitost FFT vyjádřena $\Theta(n \log n)$. Připomeňme si, že algoritmus přímým dosazením do polynomu má složitost $\Omega(n^2)$.

Nyní si můžeme ukázat rychlý algoritmus pro násobení polynomů. Nechť A a B jsou polynomy, které chceme násobit se stupni s a t. Výsledný polynom AB bude mít stupeň $s + t$. Abychom tedy mohli interpolovat z hodnot výsledného polynomu jeho koeficienty, potřebujeme jich znát nejméně $s + t + 1$. Pro potřeby FFT ovšem musí být počet hodnot mocninou dvou, proto je skutečně počítaný počet hodnot roven

$$n = 2^{\lceil \log_2(s+t+1) \rceil},$$

tedy nejbližší větší mocnině dvou. Algoritmus nejdříve volá FFT, aby spočítal hodnoty A a B v n bodech, poté tyto body vynásobí a získá hodnoty v těchto bodech pro AB. Posledním krokem je interpolace koeficientů. Díky vlastnostem $\sqrt[n]{1}$ ji lze vypočítat také pomocí FFT. Pro hodnoty $AB(\omega^0), AB(\omega), AB(\omega^2), \ldots, AB(\omega^{n-1})$ dostaneme koeficienty $ab_0, ab_1, \ldots, ab_{n-1}$ pomocí

$$[ab_0, ab_1, \ldots, ab_{n-1}] = \frac{1}{n}\text{FFT}([AB(\omega^0), AB(\omega), AB(\omega^2), \ldots, AB(\omega^{n-1})], \omega^{-1}).$$

Algoritmus 4 Rychlé násobení polynomů

```
1: procedure FastPolyMultiply(A[0,...,s], B[0,...,t])
2:     n ← 2^⌈log2(s+t+1)⌉
3:     ω ← e^(2πi/n)
4:     Doplň pomocí nul A i B na n prvků.          ▷ Nuly přidávám na konec
5:     V_A ←FFT(A, ω)                              ▷ Hodnoty pro A
6:     V_B ←FFT(B, ω)                              ▷ Hodnoty pro B
7:     for i ← 0 to n − 1 do
8:         V_AB[i] ← V_A[i] · V_B[i]               ▷ Násobení polynomů
9:     return (1/n)FFT(V_AB, ω^−1)                 ▷ Interpolace pomocí FFT
```

Algoritmus třikrát volá FFT se složitostí $O(n \log n)$. Zbylé části mají složitost $O(n)$. Celkově je tedy složitost dominována FFT.

3 Dynamické programování

Princip demonstrujeme na problému výpočtu n-tého Fibonacciho čísla. Je to příklad specifický v tom, že ukazuje princip dynamického programování v kontrastu k principu rozděl a panuj. Není ovšem obecné pravidlo, že dynamické programování lze použít pouze na problémy, pro které princip rozděl a panuj vede k neefektivnímu algoritmu.

Připomeňme, že n-té Fibonacciho číslo je definováno následujícím rekurentním vztahem

$$F(n) = \begin{cases} F(n-1) + F(n-2) & n \geqslant 2 \\ n & n \in \{0, 1\} \end{cases} \tag{3.1}$$

Nabízí se použít algoritmus rozděl a panuj, který následuje

Algoritmus 5 n-té Fibonacciho číslo pomocí rozděl a panuj

```
1: procedure FibDQ(n)
2:     if n = 0 or n = 1 then
3:         return n
4:     return FibDQ(n − 1) + FibDQ(n − 2)
```

Složitost FibDQ můžeme vyjádřit jako rekurenci

$$T(n) = T(n-1) + T(n-2) + O(1),$$

odhadem jejíhož řešení je $O(2^n)$ (lze snadno vidět metodou stromu). Algoritmus má tedy velmi nevýhodnou složitost. Pokud si visualizujeme strom rekurzivního volání, zjistíme, že problém je v opakovaném volání procedury pro stejný argument, viz. obrázek 3.1 (a). Problém odstraníme tak, že si pro každé číslo, pro které jednou zavoláme FibDQ, zapamatujeme výsledek, který vrátil. Při opakovaném rekurzivním volání pro tuto podinstanci zapamatovanou hodnotu použijeme.

Složitost FibTab je lineární. Lze snadno vidět, že k rekurzivnímu volání na řádku 10 dojde pro každé číslo (a tedy pro každou podinstanci) maximálně jedenkrát. Ve zbylých případech je vrácena v tabulce zapamatovaná hodnota. Strom rekurzivních volání pak vypadá jako na obrázku 3.1 (b).

Složitost algoritmu můžeme ještě zlepšit: časovou složitost o faktor 2, který z pohledu asymptotické notace nezanedbatelný (dostaneme ale mnohem hezčí algoritmus); paměťovou složitost zlepšíme z lineární (velikost tabulky) na konstantní.

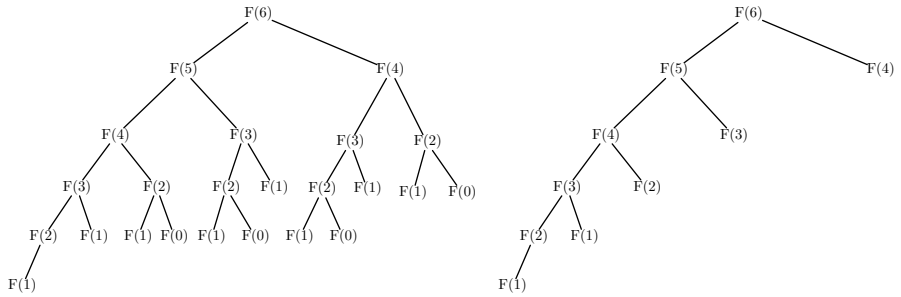
Pozorování z předchozího příkladu:

1. *Omezíme opakovaný výpočet pro stejné instance*, ke kterému dochází u přístupu rozděl a panuj tak, že si výsledky pamatujeme (například v tabulce).

2. *instance, které se v průběhu výpočtu vyskytnou, vhodně uspořádáme*. Označme jako $I_i \lesssim I_j$ situaci, kdy k výpočtu výsledku instance I_j potřebujeme znát řešení instance I_i. Dynamické programování můžeme použít pouze tehdy, pokud v množině všech instancí uspořádané podle $\lesssim$ nejsou cykly. To znamená, že neexistuje žádná instance, k jejímuž výpočtu potřebujeme znát výsledek jí samotné. Situaci lze visualizovat pomocí

Algoritmus 6 n-té Fibbonaciho číslo s pamatováním si mezivýsledků

```
1:  procedure PrepareTable(n)
2:      t[0] ← 0
3:      t[1] ← 1
4:      for i ← 2 to n do
5:          t[i] ← −1
6:      return t
7:
8:  procedure FibHelp(n, t)
9:      if t[n] = −1 then
10:         t[n] ←FibTab(n − 1) + FibTab(n − 2)
11:     return t[n]
12:
13: procedure FibTab(n)
14:     return FibHelp(n, PrepareTable(n))
```



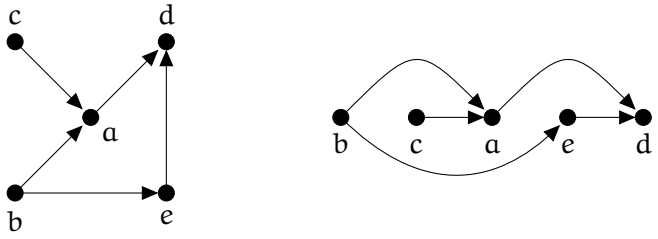
(a) FibDQ (b) FibTab

Obrázek 3.1: Rekurzivní strom volání výpočtu Fibbonaciho čísla

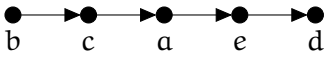
orientovaného grafu. Za vrcholy grafu vezmeme všechny instance, z I_i vede hrana do I_j právě když $I_i \lesssim I_j$. Pak lze dynamické programování použít, právě když nejsou v tomto grafu cykly. Graf bez cyklů totiž lze linearizovat. To znamená, že nalezneme takové lineární uspořádání $\leqslant$ instancí, že pro každé dvě instance platí, že pokud $I_i \lesssim I_j$, pak i $I_i \leqslant I_j$. Situace je demonstrována na obrázku 3.2.

Algoritmus 7 n-té Fibonacciho číslo pomocí dynamického programování

```
1:  procedure FibIdeal(n)
2:      if n = 0 then
3:          return n
4:      a ← 0
5:      b ← 1
6:      for n ← 2 to n − 1 do
7:          c ← a + b
8:          a ← b
9:          b ← c
```



(a) Orientovaný graf bez cyklů (b) Lineární uspořádání vrcholů



(c) Jeden z možných průběhů výpočtu

Obrázek 3.2: Linearizovaná podoba grafu

3. *Instance řešíme od nejmenší* podle jejich lineárního uspořádání. Předchozí bod zajišťuje, že vždy známe řešení všech instancí, které jsou k sestavení řešení pro aktuální instanci potřeba.

4. *Ze vstupní instance jsme schopni najít nejmenší instance*[1], přesněji pro takové, které jsou podle $\lesssim$ minimální (v grafu do nich nevede žádná hrana). Toto je potřeba, abychom mohli výpočet spustit. Hledat další instance není nutné, lze je sestavovat až v průběhu výpočtu.

Pokud pro výpočet výsledku pro instanci I potřebujeme znát výsledky instancí I_j (pro nějaká j, těchto instancí může být více), tak se řešení instance I *překrývá* s řešeními instancí I_j.

Například pro problém Fibonacciho čísel:

$$\begin{aligned}
F(2) &= F(0) + F(1) = 0 + 1 \\
F(3) &= F(1) + F(2) = 1 + (0 + 1) \\
F(4) &= F(2) + F(3) = (0 + 1) + (1 + (0 + 1)) \\
F(5) &= F(3) + F(4) = [1 + (0 + 1)] + [(0 + 1) + (1 + (0 + 1))]
\end{aligned}$$

$F(5)$ se překrývá s $F(4)$ a $F(3)$ (překrývá se i s menšími instancemi, to ale pro výpočet $F(5)$ není podstatné).

V průběhu výpočtu, kdy počítáme směrem od menších instancí k větším (ve smyslu grafu závislostí mezi instancemi, viz obrázek 3.2), můžeme využít řešení menších instancí (které už máme spočítané) jako části řešení větších instancí.

[1]Minimální instance můžeme určit už při navrhování algoritmu.

Úloha batohu

Úloha batohu, jejíž definice následuje, je optimalizačním problémem. Formálně je úloha batohu určena následovně:

Úloha batohu	
Instance:	$\{(b, w_1, \ldots, w_n) \mid b, w_1, \ldots, w_n \in \mathbb{N}\}$
Přípustná řešení:	$\mathtt{sol}(b, w_1, \ldots, w_n) = \{C \subseteq \{1, \ldots, n\} \mid \sum_{i \in C} w_i \leqslant b\}$
Cena řešení:	$\mathtt{cost}(C, (b, w_1, \ldots, w_n)) = \sum_{i \in C} w_i$
Cíl:	maximum

Pro instanci $I = (b, w_1, \ldots, w_5)$, kde $b = 29, w_1 = 3, w_2 = 6, w_3 = 8, w_4 = 7, w_5 = 12$, existuje jediné optimální řešení $C = \{1, 2, 3, 5\}$ s cenou $\mathtt{cost}(C, I) = 29$. Lze snadno ověřit, že všechna ostatní přípustná řešení mají menší cenu.

Uvažujme vstupní instanci $I = (b, w_1, \ldots, w_n)$. Jako $I(i, c)$, kde $0 \leqslant i \leqslant n$ a $0 \leqslant c \leqslant b$, označíme instanci $(c, w_1, \ldots, w_i)$. Přitom platí, že $I = I(n, b)$. Hodnota $i = 0$ znamená, že instance $I(i, c)$ neobsahuje žádnou položku.

Pro optimální řešení instance $I(i, c)$ s $i > 0$ platí následující:

1. Pokud položka i nepatří do optimálního řešení instance $I(i, c)$, pak je toto řešení shodné s optimálním řešením instance $I(i - 1, c)$.
2. Pokud položka i patří do optimálního řešení instance $I(i, c)$ (a tedy nutně platí $w_i \leqslant c$), pak toto řešení obdržíme tak, že k optimálnímu řešení instance $I(i - 1, c - w_i)$ přidáme položku i.

Pokud známe optimální řešení instancí $I(i - 1, c)$ a $I(i - 1, c - w_i)$, můžeme spočítat optimální řešení instance $I(i, c)$. K tomu musíme rozhodnout, jestli položka i do tohoto řešení patří. To můžeme udělat na základě cen optimálních řešení instancí $I(i - 1, c)$ a $I(i - 1, c - w_i)$. Označme jako $\mathrm{OPT}(i, c)$ cenu optimálního řešení instance $I(i, j)$. Pak platí:

1. Pokud položka i do optimálního řešení nepatří, pak $\mathrm{OPT}(i, c) = \mathrm{OPT}(i - 1, c)$
2. Pokud položka i do optimálního řešení patří, pak $\mathrm{OPT}(i, c) = \mathrm{OPT}(i - 1, c - w_i) + w_i$.

Pokud $w_i > c$, pak položku i do řešení nemůžeme dát, protože se tam nevejde. Jinak se rozhodneme na základě toho, které z čísel $\mathrm{OPT}(i - 1, c)$ a $\mathrm{OPT}(i - 1, c - w_i) + w_i$ je větší. Protože úloha batohu je maximalizační problém, vybereme tu možnost, která vede k řešení s větší cenou.

Z předchozího jde vidět, že instance lze uspořádat bez cyklů. Platí totiž, že $I(i - 1, c - w_i) \lesssim I(i, c)$ a současně $I(i - 1, c) \lesssim I(i, c)$. Minimální prvky vzhledem k tomuto uspořádání jsou vždycky

- $I(0, d)$ pro nějakou kapacitu d, což odpovídá instanci, kde nemáme žádné položky pro vkládání do batohu.

- $I(i, 0)$ pro nějaké i, což odpovídá prázdné kapacitě (a nemá cenu řešit, které prvky z $w_1, \ldots, w_{i-1}$ dám do řešení, protože kapacita je prázdná)

Algoritmus můžeme rozdělit do dvou částí:

1. Spočítám $OPT(i, j)$ pro $0 \leqslant i \leqslant n$ a $0 \leqslant c \leqslant b$ a výsledky uložím do tabulky.
2. Na základě tabulky spočítám optimální řešení.

Z cen optimálního řešení instancí $I(i-1, c)$ a $I(i-1, c-w_i)$ dostaneme cenu optimálního řešení instance $I(i, c)$:

$$OPT(i, c)) = \begin{cases} \max(OPT(i-1, c), OPT(i-1, c-w_i) + w_i) & \text{pokud } w_i \leqslant c \\ OPT(i-1, c) & \text{jinak} \end{cases} \tag{3.2}$$

Dále pro minimální prvky platí, že $cost(I(0, d)) = 0$ (nemám prvky, které bych do řešení zařadil) a $cost(I(i, 0)) = 0$ (prázdná kapacita a nemůžu do řešení zařadit žádné prvky).

Nyní si už můžeme napsat algoritmus, s pomocí kterého spočítáme cenu řešení $I(n, b)$. Algoritmus si vytváří tabulku, jejíž řádky odpovídají jednotlivým položkám, které chceme vložit do batohu, uvažujeme i jeden řádek odpovídající žádné položce. Sloupce tabulky odpovídají číslům $0, 1, \ldots, b$, tedy všem číslům potenciálně se vyskytujícím jako kapacita některé instance. Na místo v tabulce na řádku odpovídajícím položce i a číslu c spočítáme $OPT(i, c)$. To můžeme udělat tak, že první řádek a první sloupec vyplníme nulami. Poté můžeme po řádcích doplňovat další místa tabulky podle vztahu (3.2). Nakonec na místě odpovídajícím $I(n, b)$ je $OPT(n, b)$.

Zbývá si říci, jak nalézt skutečné řešení, nikoliv jeho cenu. To lze provést opět podle vztahu (3.2). Vezmeme $OPT(n, b)$ (tj. políčko tabulky na řádku n a ve sloupci c), pokud $b - w_n < 0$, pak n do řešení nepatří a posuneme se v tabulkce k $OPT(n-1, b)$; v opačném případě se podíváme na ceny $x = OPT(n-1, b-w_n)$ a $y = OPT(n-1, b)$. Pokud platí, že $x + w_n > y$, pak w_n do řešení patří a přejdeme k $OPT(n-1, b-w_n)$, pokud $x + w_n < y$, pak w_n do řešení nepatří a přejdeme k $OPT(n-1, b)$. Jsou-li si x a y rovny, můžeme zvolit libovolně. Poté postup opakujeme pro políčko, ke kterému jsme přešli. Končíme v případě, že se dostaneme na okraj tabulky (první sloupec nebo první řádek).

KnapsackDP vytváří tabulku o rozměrech $n \cdot b$. Pro vyplnění každého políčka je potřeba konstantní počet operací. Složitost nalezení řešení v hotové tabulce je je lineární vzhledem k n. Můžeme tedy konstatovat, že složitost algoritmu je $O(n \cdot b)$. Složitost není závislá jenom na velikosti instance měřené jako počet čísel $w_1, \ldots, w_n$, ale i na velikosti největšího čísla se v instanci vyskytujícího. Toto je zajímavá vlastnost. Pokud bychom například vynásobili všechna čísla v instanci, kterou jsme si uvedli jako příklad, milionem, KnapsackDP by počítal řešení milionkrát déle. Intuitivně bychom však řekli, že obě dvě instance jsou stejně komplikované.

Hledání nejkratších cest v grafu

Definice 8. Cesta z uzlu s do uzlu t v orientovaném grafu G je posloupnost vrcholů a hran $P = u_0, e_0, \ldots, e_{n-1}, u_n$ taková, že $e_i = (u_{i-1}, u_i)$, $u_0 = s$, $u_n = t$, a každý uzel $u \in P$ se vyskytuje v cestě právě jednou.

Definice 9. Cena $c(P)$ cesty P v ohodnoceném grafu $(G = (V, E), c)$ je suma ohodnocení všech hran vyskytujících se v cestě.

Algoritmus 8 Úloha batohu pomocí dynamického programování

```
1:  procedure KnapsackDP(w_1, ..., w_n, b)
2:      for i ← 0 to b do
3:          M[0, i] ← 0
4:      for i ← 0 to n do
5:          M[i, 0] ← 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:     S ← ∅
13:     c ← b
14:     for i ← n to 0 do
15:         if c − w_i ⩾ 0 and M[i − 1, c] < M[i − 1, c − w_i] + w_i then
16:             S ← S ∪ {i}
17:             c ← c − w_i
18:     return S
```

Definice 10. Nejkratší cesta z uzlu s do uzlu t v grafu G je taková cesta P, pro kterou platí, že $c(P) \leqslant c(R)$ pro každou cestu R z uzlu s do uzlu t.

Nalezení nejkratší cesty	
Instance:	orientovaný graf $(G = (V, E), c), s, t \in V$
Přípustná řešení:	$\mathtt{sol}(G, s, t) = \{P \mid P$ je cesta z uzlu s do uzlu $t\}$
Cena řešení:	$\mathtt{cost}(P, G, s, t) = c(P)$
Cíl:	minimum

Floyd-Warshall algoritmus

Vrcholy grafu označíme $V = \{1, 2, \ldots, n\}$. Jako $I(i, j, k)$ označíme instanci, s následujícími vlastnostmi:

- Grafem je podgraf grafu G indukovaný vrcholy $\{1, \ldots, k\} \cup \{i, j\}$. Podgraf grafu G indukovaný množinou vrcholů F je graf s množinou vrcholů F a všemi hranami z grafu G, které mají oba koncové vrcholy v množině F.
- Počáteční vrchol je i, cílový vrchol je j.

Cenu optimálního řešení $I(i, j, k)$ označíme jako $\mathsf{OPT}(i, j, k)$. Optimální řešení instance $I(i, j, k)$ můžeme určit na základě následující úvahy. Pokud je nejkratší cesta pro $I(i, j, k)$ rozdílná (tj. kratší) než nejkratší cesta pro $I(i, j, k - 1)$, pak musí nutně vést přes uzel k a platí

$$\mathsf{OPT}(i, j, k) = \mathsf{OPT}(i, k, k - 1) + \mathsf{OPT}(k, j, k - 1),$$

protože nová cesta se skládá z cesty z uzlu i do uzlu k a z cesty z uzlu k do uzlu j. Pokud tedy známe nejkratší cesty (a jejich ceny) pro instance $I(s, t, k-1)$ pro všechna s a t, můžeme rozhodnout o tom, jestli nejkratší cesta pro $I(s, t, k)$ vede přes uzel k. Je tomu tak, právě když platí, že

$$OPT(i, k, k-1) + OPT(k, j, k-1) < OPT(i, j, k-1). \tag{3.3}$$

V tomto případě je totiž cesta, kterou obdržíme spojením optimálního řešení instancí $I(i, k, k-1)$ a $I(k, j, k-1)$, kratší než cesta, která nevede přes uzel k.

Musíme se vyrovat se situací, kdy cesta z i do j v instanci $I(i, j, k)$ neexistuje. Uděláme to následovně: pokud cesta z i do j v instanci $I(i, j, k)$ neexistuje, pak nastavíme $OPT(i, j, k) = \infty$. Při reálné implementaci nahradíme ∞ nějakým dostatečně velkým číslem, například číslem větším než suma ohodnocení všech hran v grafu.

Představme si nyní situaci, kdy cesta v instanci $I(i, j, k-1)$ neexistuje, tedy platí $OPT(i, j, k-1) = \infty$. Pak

- Pokud $OPT(i, k, k-1) \neq \infty$ a $OPT(k, j, k-1) \neq \infty$ pak nerovnost (3.3) platí, a najdeme cestu.
- Ve všech zbývajících případech nerovnost neplatí a dostaneme, že $OPT(i, j, k) = \infty$.

Nerovnost (3.3) se tedy chová správně vzhledem k neexistujícím cestám.

Z předchozí diskuze je vidět, že instance lze uspořádat podle třetí komponenty: k výpočtu pro instanci $I(,,k)$, potřebujeme znát řešení instancí $I(,,k-1)$. Minimální prvky jsou pak instance $I(i, j, 0)$. Pokud existuje hrana mezi uzly i a j, máme $OPT(i, j, 0) = c((i, j))$, jinak nastavíme $OPT(i, j, 0) = \infty$.

Algoritmus Floyd-Warshall pro vstupní graf s n uzly vrátí matici D o velikosti $n \times n$, kde je na pozici (i, j) cena nejkratší cesty mezi uzly i a j. Dále vrací matici P o rozměrech $n \times n$, která na pozici (i, j) obsahuje uzel, přes který nejkratší cesta z i do j vede. Tuto matici poté využívá procedura GetPath, která s její pomocí cestu sestaví. Lze snadno vidět, že algoritmus má kubickou složitost

Algoritmus nefunguje pro grafy, které obsahují tzv. záporné cykly. Záporný cyklus je takový cyklus, jehož suma hran je menší než 0. Opakováním tohoto cyklu můžeme získat potenciálně nekonečný tah. Záporný cyklus lze detekovat tak, že zkontrolujeme diagonálu matice D. Pokud se na ní nachází záporné číslo, leží uzel, který danému místu v matici odpovídá, na záporném cyklu.

Problém obchodního cestujícího

Problém si lze představit následovně. Je dána množina měst a pro každá dvě města vzdálenost mezi nimi. Úkolem je nalézt takové uspořádání měst, že pokud je v tomto pořadí navštívíme a pak se vrátíme do počátečního města, urazíme nejkratší možnou vzdálenost.

Algoritmus 9 Nejkratší cesty v grafu

```
 1: procedure FLOYD-WARSHALL(G = (V, E), c)
 2:     for i ← 1 to n do:
 3:         for j ← 1 to n do:
 4:             D[i, j] = ∞
 5:             P[i, j] = ∞
 6:     for (i, j) ∈ E do:
 7:         D[i, j] = c((i, j))
 8:     for k ← 1 to n do:
 9:         for i ← 1 to n do:
10:             for j ← 1 to n do:
11:                 x ← D[i, k] + D[k, j]
12:                 if x < D[i, j] then
13:                     D[i, j] ← x
14:                     P[i, j] ← k
15:     return ⟨D, P⟩
16:
17: procedure GETPATH(P,i,j)
18:     mid ← P[i, j]
19:     if mid = ∞ then                          ▷ i a j jsou spojeny jednou hranou
20:         return []
21:     else
22:         return GetPath(P, i, mid) + [mid]+GetPath(P, mid, j)      ▷ + je zřetezení
    seznamů
```

Problém obchodního cestujícího	
Instance:	úplný neorientovaný graf s množinou uzlů $V = \{c_1, \ldots, c_n\}$ a ohodnocením hran δ
Přípustná řešení:	kružnice (cyklus) procházející přes všechny uzly (tzv. Hamiltonovy kružnice)
Cena řešení:	součet ohodnocení hran na dané kružnici
Cíl:	minimum

Algoritmus využívá následujícího poznatku. Pokud pro každý uzel $c \in V - \{c_1\}$ označíme cenu nejkratší cesty z c_1 do c vedoucí přes všechny uzly jako $\mathrm{OPT}[V - \{c_1\}, c]$, pak jistě můžeme cenu optimální cesty obchodního cestující nalézt jako

$$\min\{\mathrm{OPT}[V - \{c_1\}, c] + \delta(c, c_1) \mid c \in V - \{c_1, c\}\}.$$

Označme jako S libovolnou neprázdnou podmnožinu množiny $V - \{c_1\}$. Jako $\mathrm{OPT}[S, c]$ označíme délku nejkratší cesty začínající v uzlu c_1, procházející všemi uzly v S a končící v uzlu c (přičemž předpokládáme, že $c \in S$). Hodnotu $\mathrm{OPT}[S, c]$ můžeme spočítat podle

následujícího vztahu

$$\mathrm{OPT}[S, c] = \begin{cases} \delta(c_1, c) & \text{pokud } |S| = 1 \ \ (\text{tj. } S = \{c\}), \\ \min\left\{\mathrm{OPT}[S - \{c\}, c_j] + \delta(c_j, c) \mid c_j \in S - \{c\}\right\} & \text{jinak.} \end{cases} \tag{3.4}$$

První řádek předchozího vztahu je triviální. Z uzlu c_1 do uzlu c existuje jenom jedna cesta (hrana z c_1 do c) a její délka je $\delta(c_1, c)$. Ve druhém řádku předpokládáme, že známe délky nejkratších cest z c_1 do c_j pro každé c_j (tyto cesty vedou přes všechny uzly v množině $S - \{c\}$). Tedy nevedou přes c. Každou z takových cest můžeme prodloužit na cestu z c_1 do c (teď cesta vede přes uzly v S) tak, že přidáme hranu z c_j do c. Z takto vzniklých cest (pro každý uzel c_i máme jednu) vybereme tu nejkratší (tj. ve vztahu určíme její cenu).

Předchozí úvahy vedou k následujícímu algoritmu k výpočtu ceny optimální cesty obchodního cestujícího. Vlastní cestu lze určit pomocí pamatování si uzlu c_j, pro který byl druhý řádek (3.4) minimální, pro každou dvojici S, c a poté zpětné projití tabulky (analogicky předchozím dvěma algoritmům).

Algoritmus 10 Problém obchodního cestujícího

```
1: procedure DynamicTSP(V = {c_1, ..., c_n}, δ)
2:     for all c ∈ V − {c_1} do
3:         OPT[{c}, c] = δ(c_1, c)
4:     for j ← 2 to n − 1 do
5:         for all S ⊆ V − {c_1} such that |S| = j do
6:             for all c ∈ S do
7:                 OPT[S, c] = min{OPT[S − {c}, d] + δ(d, c) | d ∈ S − {c}}
8:     return min{OPT[V − {c_1}, c] + δ(c, c_1) | c ∈ V − {c_1}}
```

Protože algoritmus prochází všechny podmnožiny množiny $V - \{c_1\}$ a pro každou z nich provede operace se složitostí omezenou $O(n^2)$, je složitost algoritmu $O(n^2 2^n)$.

4 Hladové algoritmy

Algoritmus navržený hladovou 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 hladové (žravé, anglicky greedy) algoritmy. Algoritmus bez rozmyslu, hladově, vybírá tu aktuálně nejlepší možnost.

Obecné schéma hladového algoritmu se dá shrnout do následujících kroků:

1. Pro vstupní instanci I algoritmus provede volbu x a na jejím základě vytvoří *jednu* jednodušší (ve smyslu toho, že jsme poté blíže řešení) instanci 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).

Nalezení Minimální kostry grafu

Definice 11. Nechť $G = (V, E)$ je neorientovaný spojitý graf (Pro každou dvojici uzlů u, v platí, že mezi nimi existuje cesta). *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.

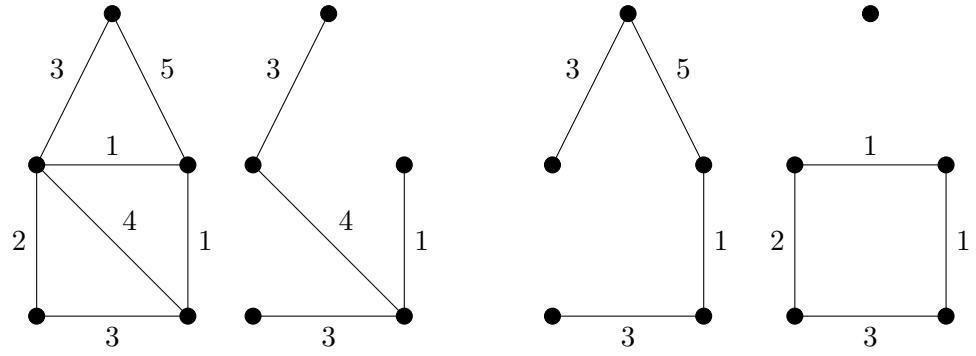
Kostra grafu G je podgraf $G' = (V, E')$ (G' má stejnou množinu uzlů jako G!) takový, že

(a) G' neobsahuje kružnici,

(b) G' je spojitý graf.

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

Na obrázku 4.1 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. Formální definice problému je následující.



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

Obrázek 4.1: Příklady pojmů z definice 11.

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

Minimální kostru lze nalézt *Kruskalým algoritmem.* 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.

Testování vzniku kružnice - disjoint set structure

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 (DSS)*. 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).
- FINDSET(x) vrátí reprezentanta množiny obsahující x.

Jednotlivé množiny v systému reprezentujeme pomocí kořenových stromů. V každém uzlu stromu uchováváme jeden prvek, ukazatel p na předka ve stromu (u kořene tento ukazatel ukazuje opět na kořen) a $\mathtt{rank}$, 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 11 Operace nad disjoint set structure

```
1: procedure MAKESET(x)
2:     x.p ← x
3:     x.rank ← 0

1: procedure FINDSET(x)
2:     if x ≠ x.p then
3:         x.p ← FINDSET(x.p)
4:     return x.p
```

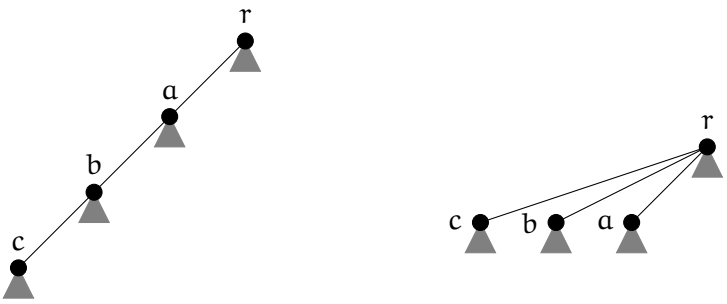
```
1: procedure UNION(x, y)
2:     r_x ← FINDSET(x)
3:     r_y ← FINDSET(y)
4:     if r_x.rank > r_y.rank then
5:         r_y.p ← r_x
6:     else
7:         r_x.p ← r_y
8:         if r_y.rank = r_x.rank then
9:             r_y.rank = r_y.rank + 1
```

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 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 $\mathtt{rank}$ 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 5.

Kruskalův algoritmus

Kruskalův algoritmus využívá disjoint set structure pro ukládání množin uzlů grafu. Na začátku algoritmu přidáme do systému pro každý vrchol jednoprvkovou množinu. Vždy, když



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

Obrázek 4.2: Heuristika v proceduře FindSet.

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 $\{0, 1, 2, \ldots |V| - 1\}$.

Algoritmus 12 Kruskalův algoritmus

```
 1: procedure Kruskal((G = (V, E), c))
 2:     Vytvoř prioritní frontu hran Q
 3:     Vytvoř |V| prvkové pole uzlů pro DSS A
 4:     for i ← 1 to |V| − 1 do
 5:         MakeSet(A[i])
 6:     E′ ← ∅
 7:     while |E′| < |V| − 1 do
 8:         Odeber z Q hranu (u, v)
 9:         if FindSet(A[u]) ≠ FindSet(A[v]) then
10:             E′ ← E′ ∪ {(u, v)}
11:             Union(A[u], A[v])
12:     return (V, E′)
```

Věta 3. Kruskal *vrací kostru grafu.*

Důkaz. Množina E' obsahuje $|V| - 1$ hran (řádek 7 algoritmu) a neobsahuje cykly (řádek 9 algoritmu + diskuze v předchozím odstavci). Tvrzení pak plyne ze souvislosti grafu G. □

Věta 4. 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 $c(e_i) \leqslant c(e_j)$, 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 $c(e_j) \leqslant c(e_i)$, odtud již dostáváme požadovanou rovnost. □

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 $|E| \geqslant |V| - 1$, můžeme konstatovat, že složitosti dominuje $O(|E| \log |E|)$.

Sestavení Huffmanova kódu

Definice 12. 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 11. (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 13. Nechť Σ je abeceda. Kód γ je *prefixový*, pokud pro všechna $x, y \in \Sigma$ platí, že $\gamma(x)$ není prefixem $\gamma(y)$.

Věta 5. *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 12. 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	<u>001</u>0000011101	c
2	<u>000</u>0011101	e
3	<u>001</u>1101	c
4	<u>11</u>01	a
5	<u>01</u>	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 13. (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 14. 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*, jesliže pro všechny ostatní prefixové kódy γ' platí, že $ABL(\gamma) \leqslant ABL(\gamma')$. Optimální kód také nazýváme Huffmanův kód.

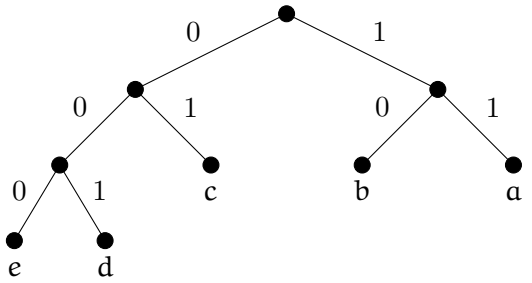
Nalezení Huffmanova kódu	
Instance:	abeceda Σ, frekvence výskytu jednotlivých znaků f_x pro $x \in \Sigma$
Přípustná řešení:	$\{\gamma \mid \gamma$ je prefixový kód pro $\Sigma\}$
Cena řešení:	$cost(\gamma, \Sigma, \{f_x \mid x \in \Sigma\}) = ABL(\gamma)$
Cíl:	minimum

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 již listem

Příklad 14. 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 $e_1, \ldots, e_m$ (předp. že 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, kterou označíme $\mathtt{depth}_T(x)$. Průměrnou délku kódového slova můžeme vyjádřit

$$ABL(T) = \sum_{x \in \Sigma} f_x \cdot \mathtt{depth}_T(x)$$

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

Věta 6. *Pro každý optimální prefixový kód γ platí, že každý nelistový uzel v 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 7. *Nechť γ je optimální prefixový kód. Pak pro každé dva listy y, z ve stromu T_γ platí, že pokud $\mathtt{depth}_{T_\gamma}(z) > \mathtt{depth}_{T_\gamma}(y)$, pak $f_y \geqslant f_z$.*

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 $\sum_{x \in \Sigma} f_x \cdot \mathrm{depth}_{T_\gamma}(x)$ pro $x \in \{y, z\}$, pak zjistíme, že

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

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

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

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

Věta 8. *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. □

Věta 9. *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) $\mathrm{ABL}(T') = \mathrm{ABL}(T) - f_w$

(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 hloubka w v T'. Odtud máme, že

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

(b) Sporem. Předpokládejme, že kód odpovídající T není optimální. Pak existuje optimální kód se stromem Z tak, že $\mathrm{ABL}(Z) < \mathrm{ABL}(T)$. Podle věty 8 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. □

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 13. 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*.

Algoritmus 13 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:     while SIZE(S) > 1 do
10:         x ← DEQUEUE(S)                         ▷ x, y jsou stromy s nejnižšími frekvencemi
11:         y ← DEQUEUE(S)
12:         Vytvoř uzel w
13:         w.freq ← x.freq + y.freq
14:         w.left ← x
15:         w.right ← y
16:         ENQUEUE(S, w)                          ▷ Vložím nově vytvořený strom do fronty
17:     return DEQUEUE(S)                          ▷ Vrátím jediný strom v frontě
```

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.

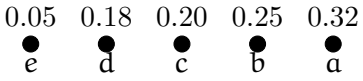
Věta 10. HUFMANN *vrací optimální kód.*

Důkaz. Stačí si všimnout, že jeden krok algoritmu odpovídá konstrukci z věty 9. □

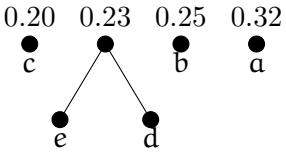
Příklad 15. Uvažujme abecedu s frekvencemi výskytu znaků danou tabulkou

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

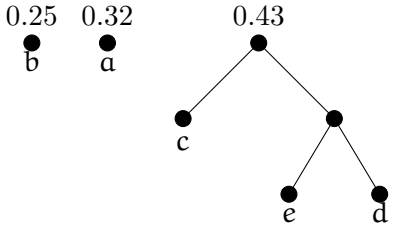
Fronta stromů v algoritmu pak postupně projde následujícími stavy.



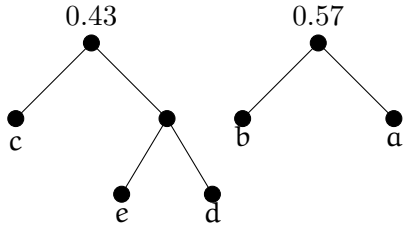
(a) inicializace



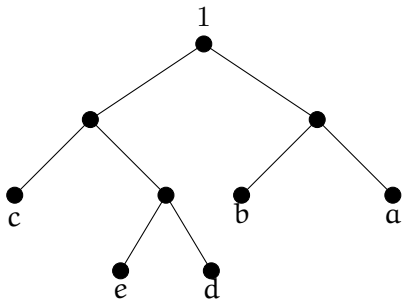
(b) Po 1. iteraci



(c) Po 2. iteraci



(d) Po 3. iteraci



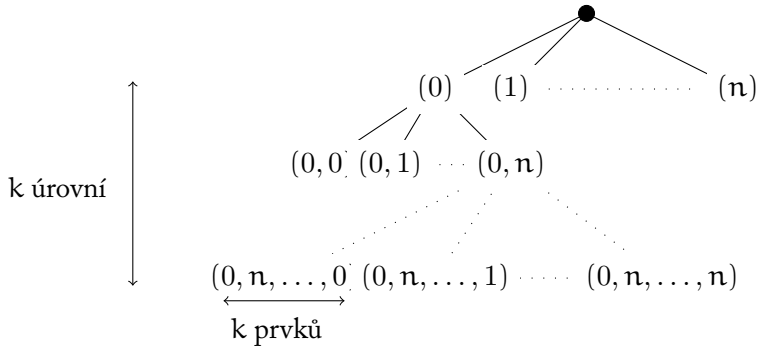
(e) Po 4. iteraci

□

5 Metody založené na hrubé síle

Metoda hrubé síly se dá popsat jednoduchou větou „zkus všechny možnosti.“ U optimalizačních problémů to znamená, že k dané vstupní instanci algoritmus vygeneruje všechna přípustná řešení a pak z nich vybere to optimální. Technika hrubé síly je použitelná i pro rozhodovací problémy. Pokud pro danou instanci x rozhodovacího problému L (zde chápaného jako jazyk L), existuje vhodný certifikát, na jehož základě víme, že $x \in L$ (tedy, odpověd je ano), algoritmus fungující hrubou silou vygeneruje všechny možné kandidáty na takový certifikát a poté se jej v této množině pokusí najít. Uvažme například problém SAT. Zde můžeme za certifikát považovat takové ohodnocení výrokových proměnných, při kterém je vstupní instance, tj. formule výrokové logiky, pravdivá. Algoritmus vygeneruje všechna možná ohodnocení výrokových proměnných této formule a poté ověří, jestli je pro některé z nich formule pravdivá. Pokud takové ohodnocení najde, je odpověď *ano*.

Generování všech možností často vede ke generování základních kombinatorických struktur jako jsou permutace a variace. Způsob jejich generování si můžeme ukázat na klasickém problému tahání barevných míčků z pytlíku. Předpokládejme, že máme míčky n barev označených jako $\{0, 1, \ldots, n-1\}$ a chceme vytáhnout k míčků. Algoritmus pro vygenerování všech možných výběrů těchto míčků je rekurzivní, strom rekurzivních zavolání vypadá následovně.



Uzly stromu odpovídají vytažení jednoho míčku. Podle úrovní stromu určíme, kolikátý míček v pořadí taháme. Označíme-li si generovanou sekvenci jako $\langle a_1, \ldots, a_k \rangle$, pak 1. úroveň (uzly v hloubce 1) odpovídá volbě a_1, 2. úroveň odpovídá volbě pro a_2 a tak dále. Pro každý uzel platí, že uzly cestě od kořene k tomuto uzlu jednoznačně určují odpovídající část generované sekvence. Například u uzlu v hloubce 3 víme, které míčky jsme vytáhli pro a_1, a_2 a a_3. Na základě této informace můžeme rozhodnout, jaké míčky zvolíme v další vrstvě (jestli chceme, aby se míček v sekvenci opakoval, nebo ne). Uzly v hloubce k už nemají potomky (a jsou tedy listy). Sekvence odpovídající cestám z kořene do listů jsou pak vygenerované sekvence.

Algoritmus pro generování prochází strom do hloubky, můžeme proto použít rekurzi. Argumenty procedury GENERATE jsou množina míčků X, doposud sestavená část sekvence $\langle a_1, \ldots, a_i \rangle$ a počet míčků v úplné sekvenci k. Procedura FILTER slouží k výběru toho, které míčky lze dosadit za a_{i+1}. Pokud FILTER vždy vrátí X, GENERATE generuje k-prvkové

variace s opakováním, pokud Filter vrátí $X - \{a_1, \ldots, a_i\}$, Generate generuje variace bez opakování. Generování spustíme zavoláním Generate s prázdnou sekvencí a.

Algoritmus 14 Generování variací a permutací

```
1: procedure Generate(X, ⟨a_1, ..., a_i⟩, k)
2:     if i = k then                          ▷ Sekvence má k prvků, skonči rekurzi
3:         Zpracuj a_1, ..., a_i
4:     S ← Filter(X, ⟨a_1, ..., a_i⟩)         ▷ Vyfiltruj prvky, které lze dosadit za a_{i+1}
5:     for x ∈ S do
6:         Generate(X, ⟨a_1, ..., a_i, x⟩, k)  ▷ Doplň sekvenci o x a pokračuj v rekurzi
```

Pojmenování *backtracking* je inspirováno principem, na kterém Generate pracuje. Když algoritmus nalezne jednu sekvenci (dostane se do listu stromu), vystoupí z rekurze o úroveň nahoru (tj. posune se ve stromě po cestě směrem ke kořenu) a generuje další sekvence rekurzivním voláním na řádku 7. Tento „návrat nahoru“ má anglické jméno backtrack.

Pokud používáme backtracking pro řešení konkrétního problému, často nemusíme generovat všechny možnosti. Předpokládejme že máme vygenerovánu část sekvence $a = a_1, \ldots, a_i$. Potom můžeme Generate obohatit o následující testy.

I. test na řádku 2 můžeme nahradit testem, který rozhodne, zda-li je $a_1, \ldots, a_i$ už řešením, nebo se dá rychle doplnit na řešení (například libovolným výběrem zbylých prvků sekvence).

II. před rekurzivním voláním na řádku 7. můžeme testovat, jestli $a_1, \ldots, a_i, x$ je prefixem sekvence, kterou chceme vygenerovat (tj. to, jestli má smysl pokračovat v generování zbytku sekvence). Pokud ne, Generate už rekurzivně nevoláme.

Oba předchozí testy se dají použít k ořezání stromu rekurzivního volání.

Jednoduchý SAT solver

Jako ukázku si uvedeme algoritmus pro problém SAT. Připomeňme, že SAT je definován jako

SAT	
Instance:	formule výrokové logiky v konjunktivní normální formě φ
Řešení:	1 pokud je φ splnitelná, jinak 0.

K sestavení algoritmu pro SAT stačí upravit Generate. Algoritmus totiž generuje postupně všechna možná ohodnocení výrokových proměnných. Uvažme formuli φ, která je konjunkcí m literálů $C_1, \ldots, C_m$ a obsahuje k výrokových proměnných $x_1, \ldots, x_k$. Ohodnocení těchto proměnných můžeme chápat jako sekvenci $\langle a_1, \ldots, a_k \rangle$ složenou z 1 a 0, přitom a_i je ohodnocení proměnné x_i. Dále si pro klauzuli C_j definujeme následující dva predikáty

- $\mathcal{F}(C_j, \langle a_1, \ldots, a_i \rangle)$ je pravdivý, právě když proměnné obsažené v C_j patří do $\{x_1, \ldots, x_i\}$ a vzhledem k ohodnocení $\langle a_1, \ldots, a_i \rangle$ neobsahuje C_j žádný pravdivý literál,

Algoritmus 15 Jednoduchý SAT solver

```
1:  procedure EasySat(φ = C_1 ∨ ··· ∨ C_m, ⟨a_1, ..., a_i⟩, k)
2:      E ← 1
3:      for j ← 1 to m do
4:          if not T(C_j, ⟨a_1, ..., a_i⟩) then
5:              E ← 0
6:              break
7:      if E then
8:          return 1                                  ▷ Všechny klauzule jsou pravdivé
9:      for x ∈ {0, 1} do
10:         E ← 1                                     ▷ Hledám nesplnitelnou klausuli
11:         for j ← 1 to m do
12:             if F(C_j, ⟨a_1, ..., a_i, x⟩) then
13:                 E ← 0
14:                 break
15:         if E then
16:             if EasySat(φ, ⟨a_1, ..., a_i, x⟩, k) then
17:                 return 1                          ▷ φ je pravdivá, algoritmus končí
18:     return 0                                      ▷ Obě rekurzivní volání vrátila 0
```

- $\mathcal{T}(C_j, \langle a_1, \ldots, a_i \rangle)$ je pravdivý, pokud literál alespoň jedné proměnné z C_j patřící do $\{x_1, \ldots, x_i\}$ je při ohodnocení $\langle a_1, \ldots, a_i \rangle$ pravdivý.

S pomocí $\mathcal{F}$ a $\mathcal{T}$ můžeme pro SAT jednoduše implementovat testy zmíněné v bodech I a II. Víme, že formule φ je pravdivá, právě když jsou pravdivé všechny klausule, což platí, právě když je v každé klausuli alespoň jeden pravdivý literál. Pokud je tedy $\mathcal{T}(C_j, \langle a_1, \ldots, a_i \rangle)$ pravdivý pro všechny klausule, víme, že φ je pravdivá pro libovolné doplnění $\langle a_1, \ldots, a_i \rangle$. Naopak, pokud je splněno $\mathcal{F}(C_j, \langle a_1, \ldots, a_i \rangle)$ alespoň pro jednu klausuli, je tato klausule pro libovolné doplnění $\langle a_1, \ldots, a_i \rangle$ nepravdivá, v důsledku čehož je nepravdivá i φ.

Časová složitost EasySat je v nejhorším případě exponenciální vhledem k počtu proměnných k. Pokud každá klausule nesplnitelné formule φ obsahuje literál proměnné x_k, pak EasySat vygeneruje všechna ohodnocení. Těchto ohodnocení je 2^k.

SAT je důležitý nejen teoreticky (jak zjistíte v kurzu složitosti), ale také pro praktické nasazení. Mnoho v praxi důležitých úloh přechází na SAT (ověřování návrhu logických obvodů, ověřování správnosti modelů apod). Existuje dokonce každoroční soutěž programů, tzv. SAT solverů, pro řešení SAT problému. Mnohé z nich jsou vyvíjené velkými firmami. EasySat je z tohoto pohledu velmi naivním algoritmem (i když většina SAT solverů je více či méně založena na backtrackingu). Jednoduše ale demonstruje základní princip ořezání stromu rekurze.

Úloha n dam

Úloha n dam je problém, který se často vyskytuje v programovacích nebo matematických soutěžích. Úkolem je spočítat kolika způsoby lze umístit n dam na šachovnici tak, aby se vzájemně neohrožovaly. Přitom předpokládáme, že dáma může táhnout jako v šachu, tedy posunout se o libovolný počet polí vodorovně, svisle nebo diagonálně.

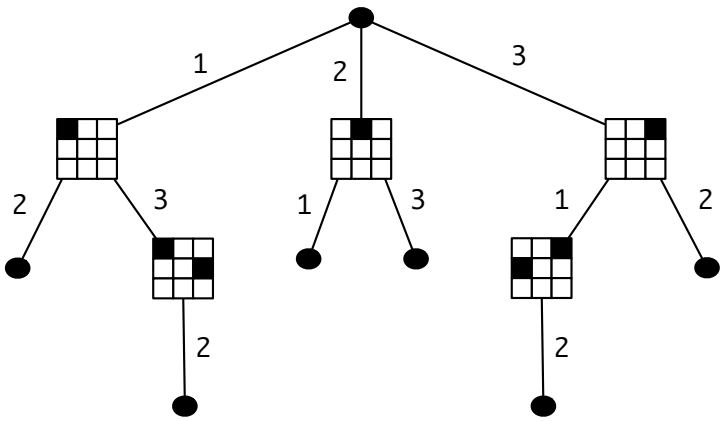
Naivní přístup k řešení tohoto problému by bylo vygenerovat všechna možná rozmístění dam na šachovnici a poté pro každé rozmístění otestovat, jestli je dámy neohrožují. Tento přístup je ale velmi neefektivní, protože takových rozmístění je $\binom{n^2}{n} = \frac{n^2!}{n!(n^2-n)!}$.

Ukážeme si sofistikovanější přístup, ve kterém využijeme znalostí šachových pravidel už během generování. Sloupce a řádky šachovnice si označíme čísly $1 \dots n$. Protože dáma táhne horizontálně, ve správném rozestavení musí být na každém řádku právě jedna dáma. Pozice tedy můžeme generovat jako sekvence $\langle a_1, \dots, a_n \rangle$, kde a_i je sloupec, ve kterém se nachází dáma na i-tém řádku. Protože v každém sloupci může být právě jedna dáma, můžeme jak kostru algoritmu využít GENERATE pro generování permutací. Jediná věc, kterou můžeme přidat, je test odpovídají podmínce II. Pro $a_1, \dots, a_i$ otestujeme, jestli jestli se dámy v prvních i řádcích neohrožují diagonálně.

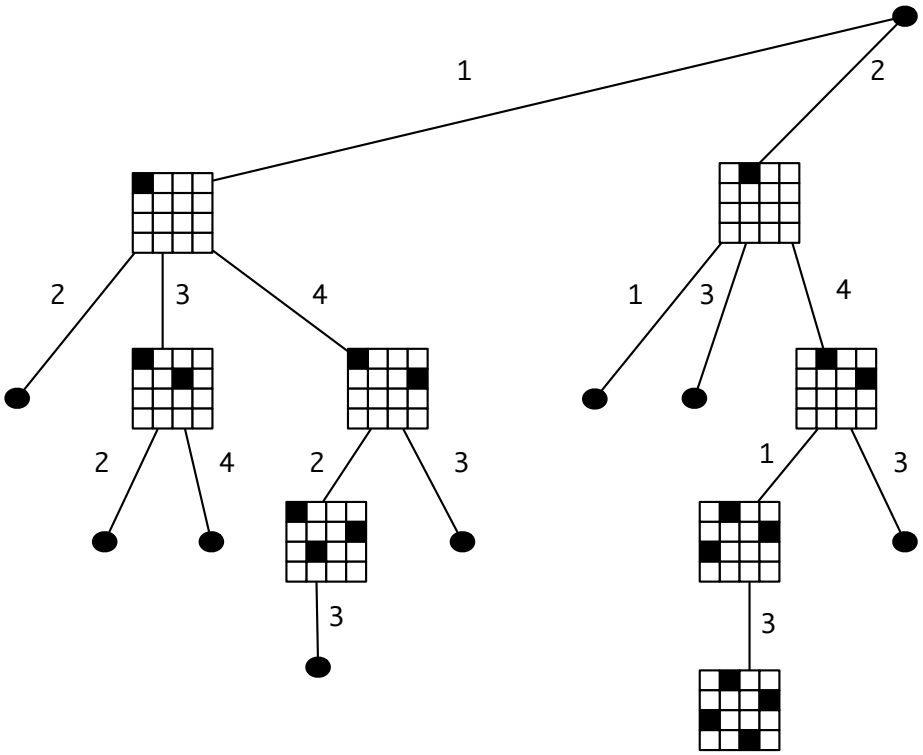
Algoritmus 16 Úloha n dam

```
1:  procedure QUEENS(⟨a_1, ..., a_i⟩, n)
2:      if i=n then
3:          return 1
4:      S ← {1, ..., n} \ {a_1, ..., a_i}
5:      c ← 0
6:      for a_{i+1} ∈ S do
7:          p ← 1
8:          for j ← 1 to i do
9:              if |j − (i + 1)| = |a_j − a_{i+1}| then
10:                 p ← 0
11:                 break
12:         if p then
13:             c ← c + QUEENS(⟨a_1, ..., a_i, a_{i+1}⟩, n)
14:     return c
```

Příklad 16. (a) pro $n = 3$ je výsledek 0.



(b) pro $n = 4$ existují 2 pozice, strom na následujícím obrázku je symetrický



Složitost Queens není známa, nicméně je obrovská. Jenom počet možných správných pozic a tedy i vygenerovaných úplných sekvencí roste obrovsky rychle. Přesný vzorec není zatím znám, nicméně pro $n = 5$ je správných pozic 10, pro $n = 24$ je jich už více než $227 \cdot 10^{12}$.

Set Cover

Na problému *Set Cover* si ukážeme jednoduchou techniku, jak prořezat strom rekurze při řešení optimalizačního problému. Řekněmě, že hledáme řešení instance I. Idea je, že budeme generovat prvky $x \in \mathtt{sol}(I)$, počítat pro ně cenu $\mathtt{cost}(x, I)$ a vybereme ten optimální. Pokud si v průběhu algoritmu pamatujeme cenu zatím nejlepšího nalezeného řešení, lze za určitých podmínek prořezat strom rekurze a vyhnout se výpočtu některých prvků $\mathtt{sol}(I)$. Nutnou podmínkou je monotonie funkce $\mathtt{cost}$ vzhledem k tomu, jak rozšiřujeme hledanou sekvenci. Předpokládejme tedy, že $\mathtt{cost}$ je neklesající. To znamená, že umíme spočítat cenu částečného řešení reprezentovaného doposud vygenerovanou sekvencí (pro sekvenci $\langle a_1, \ldots, a_i \rangle$ tuto cenu označíme $\mathtt{cost}(\langle a_1, \ldots, a_i \rangle, I)$), a že vždycky platí

$$\mathtt{cost}(\langle a_1, \ldots, a_i \rangle, I) \leqslant \mathtt{cost}(\langle a_1, \ldots, a_i, a_{i+1} \rangle, I).$$

Pak, pokud je I instance minimalizačního problému a $\mathtt{cost}(\langle a_1, \ldots, a_i \rangle, I)$ je větší než cena doposud nejlepšího nalezeného řešení, tak $\langle a_1, \ldots, a_i \rangle$ není možné doplnit na optimální řešení (cena optimální řešení je menší nebo rovna ceně nejlepšího doposud nalezeného řešení).

Pokud existuje efektivní aproximační algoritmus k danému optimalizačnímu problému, je cenu řešení, které tento algoritmus vrátí, možné použít k inicializaci ceny doposud nejlepšího nalezeného řešení. Popsaný přístup označujeme jako *branch-and-bound*.

Set cover je optimalizační problém, hojně studovaný především v oblasti aproximačních algoritmů. Jeho zadání je jednoduché. Jsou dány množina X a systém jejích podmnožin $\mathcal{S}$, který tuto množinu pokrývá (tj. platí $\bigcup \mathcal{S} = X$). Úkolem je nalézt co nejmenší podmnožinu $\mathcal{S}$ tak, aby stále pokrývala X.

Set Cover	
Instance:	konečná množina X, systém podmnožin $\mathcal{S} = \{S_i \mid S_i \subseteq X\}$ takový, že $\bigcup_{S_i \in \mathcal{S}} S_i = X$
Přípustná řešení:	$\mathcal{C} \subseteq \mathcal{S}$ takové, že $\bigcup_{S_i \in \mathcal{C}} S_i = X$
Cena řešení:	$\mathtt{cost}(X, \mathcal{S}, \mathcal{C}) = \lvert\mathcal{C}\rvert$
Cíl:	minimum

Idea algoritmu je tedy generovat podmnožiny $\mathcal{S}$, které pokrývají X a vybrat tu s nejmenším počtem prvků. K tomuto účelu můžeme opět použít jako kostru Generate. Podmnožiny n prvkové množiny totiž jednoznačně odpovídají n prvkovým variacím nad $\{0, 1\}$. Uvažme množinu Y. Pak pro každou podmnožinu Z této množiny můžeme definovat její *charakteristickou funkci* $\mu_Z : Y \to \{0, 1\}$ jako

$$\mu_Z(y) = \begin{cases} 0 & y \notin Z \\ 1 & y \in Z \end{cases}$$

Mezi charakteristickými funkcemi a podmnožinami je jednoznačný vztah. Každá podmnožina Z indukuje unikátní charakteristickou funkci, a naopak, je-li dána charakteristická funkce $\mu : Y \to \{0, 1\}$ pak množinu $\mathtt{Set}(\mu)$, která tuto funkci indukuje nalezneme jako

$$\mathtt{Set}(\mu) = \{y \in Y \mid \mu(y) = 1\}.$$

Pokud zafixujeme pořadí prvků v množině X, tj $X = \{x_1, x_2, \ldots, x_n\}$, dá se každá podmnožina $Z \subseteq X$ jednoznačně zapsat jako sekvence $\langle \mu_Z(x_1), \mu_Z(x_2), \ldots, \mu_Z(x_n)\rangle$. Ke vygenerování všech podmnožin $\mathcal{S}$ tedy stačí generovat všechny n-prvkové sekvence nad $\{0, 1\}$. V dalším budeme značit jako $\mathtt{Set}(\langle a_1, \ldots, a_i\rangle)$ podmnožinu, která odpovídá sekvenci $\langle a_1, \ldots, a_i\rangle$ doplněné na konci potřebným počtem 0.

Zamysleme se nad podmínkami, pomocí nichž lze ořezat strom rekurze. Uvažujme situaci, kdy máme vygenerovánu sekvenci $\langle a_1, \ldots, a_i\rangle$. Potom můžeme využít následující (nezapomínejme, že $\mathcal{S}$ je systém množin (prvky jsou podmnožiny X) takže $\mathtt{Set}(\langle a_1, \ldots, a_i\rangle)$ je také systém množin, je to totiž podmnožina $\mathcal{S}$.

- Pokud $\bigcup \mathtt{Set}(\langle a_1, \ldots, a_i\rangle) \cup \bigcup(\mathcal{S} \setminus \{S_1, \ldots, S_i\}) \neq X$, pak můžeme rekurzi ukončit, protože $\langle a_1, \ldots, a_i\rangle$ není možné doplnit tak, aby pokryla celou množinu X.
- Pokud $\bigcup \mathtt{Set}(\langle a_1, \ldots, a_i\rangle) = X$, tak končíme rekurzi, protože hledáme minimální pokrytí X a přidávat další prvky do $\mathtt{Set}(\langle a_1, \ldots, a_i\rangle)$ proto nemá smysl.
- Pokud je $\lvert\mathtt{Set}(\langle a_1, \ldots, a_i\rangle)\rvert$ větší než velikost doposud nejmenšího nalezeného pokrytí, končíme rekurzi. Hledáme minimální pokrytí a pokračovat v přidávání prvků do $\mathtt{Set}(\langle a_1, \ldots, a_i\rangle)$ nemá smysl.

Algoritmus 17 Set Cover pomocí branch-and-bound

```
1: C ← S          ▷ Globální proměnná pro uchování zatím nejlepšího nalezeného pokrytí
2:
3: procedure OptimalSetCover(⟨a_1, ..., a_i⟩, F, X)
4:     Y ← ⋃ Set(⟨a_1, ..., a_i⟩)
5:     if Y = X then                    ▷ test, jestli ⟨a_1, ..., a_i⟩ už neindukuje pokrytí
6:         C ← Set(⟨a_1, ..., a_i⟩)
7:         return                       ▷ ⟨a_1, ..., a_i⟩ je zatím nejlepší pokrytí
8:     if Y ∪ ⋃ F ≠ X or |Set(⟨a_1, ..., a_i⟩)| = |C| − 1 then
9:         return                       ▷ nemůžu vytvořit lepší pokrytí než je C
10:    OptimalSetCover (⟨a_1, ..., a_i, 1⟩, F \ {S_{i+1}}, X)
11:    OptimalSetCover (⟨a_1, ..., a_i, 0⟩, F \ {S_{i+1}}, X)
```

Výpočet optimálního řešení spustíme pomocí OptimalSetCover$(\langle\rangle, \mathcal{S}, X)$. Algoritmus si uchovává v globální proměnné $\mathcal{C}$ zatím nejlepší nalezené pokrytí, které na začátku nastaví na celou množinu $\mathcal{S}$ Na řádku 5 pak testujeme, jestli jsme nalezli pokrytí. Pokud ano, je toto pokrytí určitě lepší než $\mathcal{C}$. Na řádku 9 totiž testujeme velikost doposud vygenerované podmnožiny. Pokud je jenom o 1 menší než $\mathcal{C}$, tak ukončíme rekurzi. Přidáním dalšího prvku bychom totiž mohli dostat jenom pokrytí, které je alespoň tak velké jako $\mathcal{C}$. Na stejném řádku také testujeme, zda-li lze aktuální podmnožinu rozšířit na pokrytí. Pokud ne, rekurzi ukončíme.

Úloha batohu

Použití backtrackingu vede většinou (u všech příkladů, které jsme si ukázali, tomu tak bylo) k algoritmu s horší než polynomickou složitostí. Toto je přirozené, backtracking většinou používáme k nalezení (optimálního) řešení problému, které považujeme za prakticky neřešitelný (jako SAT, Set Cover nebo úloha n dam). Mohlo by se tedy zdát, že bactracking není pro větší instance příliš použitelný. V této kapitole si ale ukážeme, jak se dá backtracking zkombinovat s algoritmem navrženým jiným způsobem a vylepšit tak jeho výkon (tak, že algoritmus spočítá řešení, které je bližší optimálnímu řešení). Ukážeme si to na příkladu úlohy batohu. Tento optimalizační problém definovaný jako

Úloha batohu	
Instance:	$\{(b, w_1, \ldots, w_n) \mid b, w_1, \ldots, w_n \in \mathbb{N}\}$
Přípustná řešení:	$sol(b, w_1, \ldots, w_n) = \{C \subseteq \{1, \ldots, n\} \mid \sum_{i \in C} w_i \leqslant b\}$
Cena řešení:	$cost(C, (b, w_1, \ldots, w_n)) = \sum_{i \in C} w_i$
Cíl:	maximum

Idea algoritmu je následující. Pro $k \leqslant n$ nejdříve vygenerujeme všechny podmnožiny množiny $\{1, \ldots, n\}$ do velikosti k, takové, že pro každou z nich je suma vah jim odpovídajících prvků menší než b. V druhé fázi algoritmu se každou z podmnožin pokusíme rozšířit tak, že budeme žravým způsobem přidávat další prvky $\{1, \ldots, n\}$. Vybereme takovou rozšířenou množinu, suma jejíchž prvků je největší.

Algoritmus 18 Kombinace backtrackingu a greedy přístupu pro úlohu batohu

```
1:  procedure GenerateCandidates(⟨a_1, ..., a_i⟩, (b, w_1, ..., w_n), k)
2:      if Σ_{i∈Set(⟨a_1,...,a_i⟩)} w_i > b then
3:          return ∅
4:      if |Set(⟨a_1, ..., a_i⟩)| = k or i = n then
5:          return {Set(⟨a_1, ..., a_i⟩)}
6:      X ← GenerateCandidates(⟨a_1, ..., a_i, 1⟩, (b, w_1, ..., w_n), k)
7:      Y ← GenerateCandidates(⟨a_1, ..., a_i, 0⟩, (b, w_1, ..., w_n), k)
8:      return X ∪ Y
9:
10: procedure ExtendCandidate(C, (b, w_1, ..., w_n))
11:     Vytvoř prioritní frontu Q z prvků 1, ..., n uspořádanou sestupně podle odpovída-
        jících prvků w_1, ..., w_n
12:     while Q není prázdná do
13:         Odeber z Q prvek x
14:         if x ∉ C and Σ_{i∈C} +w_x ⩽ b then
15:             C ← C ∪ {x}
16:     return C
17:
18: procedure KnapsakScheme((b, w_1, ..., w_n), k)
19:     X ← GenerateCandidates(⟨⟩, (b, w_1, ..., w_n), k)
20:     B ← ∅
21:     for x ∈ X do
22:         A ← ExtendCandidate(x, (b, w_1, ..., w_n))
23:         if Σ_{x∈A} w_x > Σ_{x∈B} w_x then
24:             B ← A
25:     return B
```

Složitost algoritmu závisí mimo velikosti vstupní instance i na volbě k. Pokud $k = 0$ je KnapsackScheme jenom obyčeným greedy algoritmem se složitostí $O(n \log n)$. Situace je podobná, pokud uvažujeme k jako pevně danou konstantu a počítáme složitost algoritmu závislou jenom na n. Počet kandidátů je pak totiž roste polynomicky, jejich počet je vždy omezen $O(n^k)$ (kandidáty totiž počítáme jako k prvkové variace bez opakování) a tedy složitost celého algoritmu je polynomická v závisloti na n. Z toho také plyne, že v závislosti na k roste složitost algoritmu exponenciálně.

6 Iterativní zlepšování

Algoritmus implementující iterativní zlepšování si po celou dobu běhu udržuje jedno aktuální přípustné řešení. Na začátku si může za aktuální řešení vybrat libovolné z přípustných řešení dané instance (např. náhodně vygenerované). Pak iterativně přechází na další přípustná řešení, které vybírá z okolí toho aktuálního. Okolí aktuálního řešení jsou jiná přípustná řešení, která se z toho aktuálního dají spočítat pomocí jednoduchého a efektivního algoritmu. Pro potřeby algoritmu je nutné přesně určit, jak vypadá okolí libovolného přípustného řešení a také, jak z tohoto okolí jedno řešení vybrat. Iterace končí po splnění vhodné podmínky. Často používané podmínky jsou například: cena aktuálního řešení a řešení, na které přecházíme, se už moc neliší; nemůžeme přejít na řešení s lepší cenou apod. Princip můžeme shrnout v následujícím pseudokódu.

```
1: procedure LocalSearch(I)
2:     S ← InitialSolution(I)                    ▷ Vygeneruj počáteční řešení
3:     while True do
4:         C ← Neighbourhood(S, I)               ▷ Vygeneruj okolí aktuálního řešení
5:         S′ ← SelectNeighbour(C, S, I)         ▷ Najdi další řešení
6:         if End(I, S′, S) then                 ▷ Test konce iterace
7:             return S′
8:         S ← S′
```

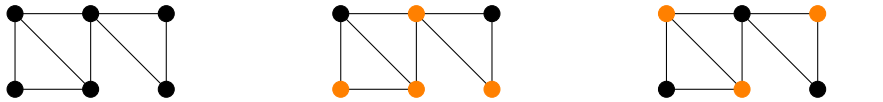
Pro konkrétní algoritmus musíme doplnit implementace procedur Neighbourhood, SelectNeighbour a End. Řádky 4 a 5 lze někdy spojit do jednoho, pokud dokážeme nalézt následující řešení bez potřeby celé okolí vygenerovat. Pomocné procedury obecně nejsou vzájemně nezávislé, například způsob volby dalšího řešení může mít vliv na způsob, kterým určujeme konec algoritmu.

Vrcholové pokrytí

Techniku si ukážeme na problému *vrcholového pokrytí grafu* (Vertex Cover). Vstupem je neorientovaný graf. Cílem je najít minimální množinu uzlů takovou, že pokrývá všechny hrany grafu. To znamená, že každá hrana grafu inciduje s některým uzlem z této množiny. Formální definice následuje.

Vrcholové pokrytí grafu	
Instance:	Neorientovaný graf $G = (V, E)$.
Přípustná řešení:	$sol(G) = \{V' \mid V' \subseteq V$ a pro všechny hrany $\{u, v\} \in E$ platí, že buď $u \in V'$ nebo $v \in V'\}$
Cena řešení:	$cost(V', G) = \lvert V' \rvert$
Cíl:	minimum

Jako počáteční pokrytí můžeme zvolit libovolné pokrytí, výpočetně nejjednoduší je zvolit všechny uzly grafu, které tvoří pokrytí vždycky. Jako další je nutné zvolit jak bude vypadat



(a) Neorientový graf (b) Vrcholové pokrytí grafu (c) Vrcholy netvořící pokrytí

Obrázek 6.1: Pokrytí grafu

okolí aktuálního řešení. Při volbě je nutné mít na paměti, že v okolí chceme efektivně hledat následující řešení (tedy čím menší okolí, tím lépe). Na druhou stranu musí být okolí dostatečně velké, aby se v něm potenciálně nacházelo dobré řešení. V algoritmu, který si ukážeme, za okolí aktuálního pokrytí zvolíme všechna taková pokrytí, která dostaneme přídáním nebo odebráním vrcholu. Pro pokrytí C potom máme

$$\textsc{Neighbourhood}(C,(V,E)) = \{C\setminus\{v\} \mid v \in C,\ C\setminus\{v\} \text{ je pokrytí}\} \cup \{C\cup\{v\} \mid v \in E\setminus C\}. \tag{6.1}$$

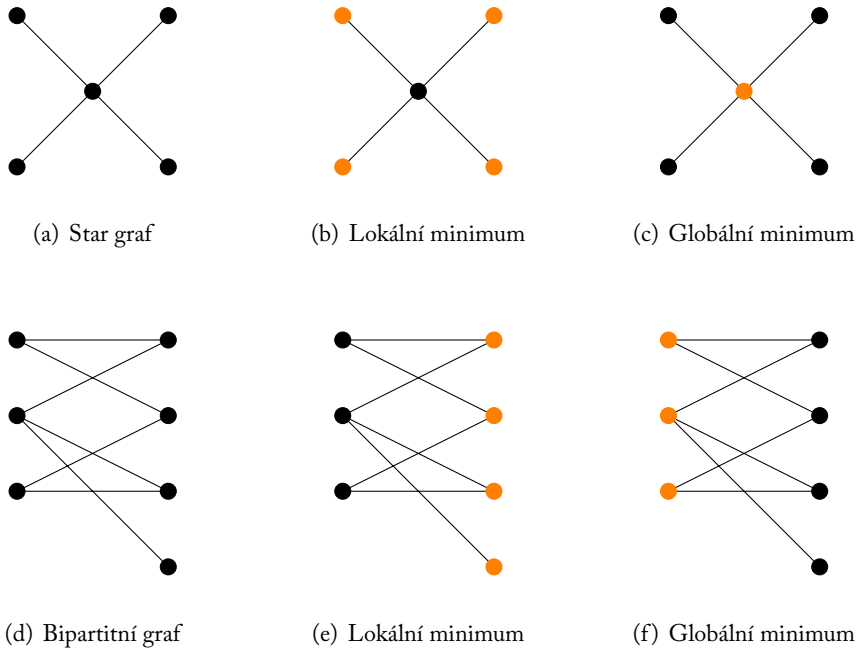
Pro výběr prvku z okolí existuje několik strategií, z nichž nejjednoduší je vybrat prvek s nejlepší cenou. Této strategii se říká *gradientní metoda.* V případě vrcholového pokrytí tedy vybíráme pokrytí s nejmenším počtem prvků. Protože okolí je definováno jako (6.1), vybíráme jeden z prvků, které dostaneme odebráním vrcholu z aktuálního pokrytí. Který z nich vybereme je jedno, můžeme vybrat náhodný (pozor, výběr prvku z okolí sice ovlivňuje další běh algoritmu, ale v momentě výběru nemůžeme rozhodnout, který z prvků se stejnou cenou vede k lepšímu finálnímu řešení, takže lze vybrat náhodný prvek). Přirozeně, algoritmus končí v momentě, kdy se v okolí aktuálního řešení nevyskytuje žádné řešení s lepší cenou.

Pseudokód algoritmu následuje:

Algoritmus 19 Vrcholové pokrytí gradientní metodou

```
1:  procedure VertexCoverGradient((V, E))
2:      C ← V
3:      F ← ∅                                   ▷ Vrcholy, ktere nelze odebrat
4:      while C \ F ≠ ∅ do
5:          Z C \ F vyber náhodný vrchol u
6:          x ← true
7:          for {v, w} ∈ E do                   ▷ Pokrývá C \ {u} všechny hrany?
8:              if v ∉ C \ {u} and w ∉ C \ {u} then
9:                  x ← false
10:                 F ← F ∪ {u}                 ▷ u nelze odebrat
11:                 break
12:         if x then
13:             C ← C \ {u}                     ▷ Přejdu na další řešení
14:     return C
```

Časová složitost VertexCoverGradient je v nejhorším případě $O(|V| \cdot |E|)$. Cyklus na řádku 4 proběhne maximálně $|V|$ krát, cyklus na řádku 7 maximálně $|E|$ krát.



(a) Star graf (b) Lokální minimum (c) Globální minimum

(d) Bipartitní graf (e) Lokální minimum (f) Globální minimum

Obrázek 6.2: Lokální a globální minima pro vybrané typy grafů

Problém uváznutí v lokálním minimu

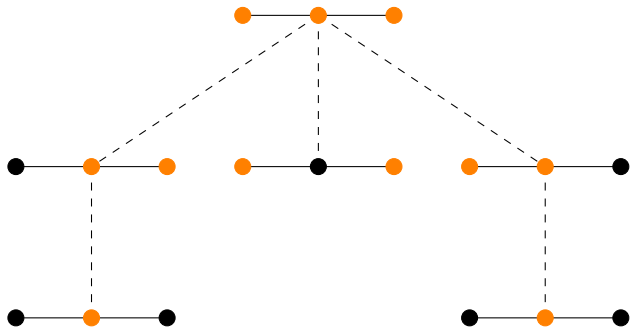
Gradientní metoda nemusí vést k nalezení optimálního řešení. Pokud se algoritmus dostane do situace, kdy se v okolí aktuálního řešení nenachází řešení s lepší cenou, označujeme aktuální řešení jako *lokální minimum*. Optimální řešení pak označujeme jako *globální minimum*. V obecném případě se lokální a globální minimum nemusí shodovat. Může se tedy stát, že algoritmus vrátí řešení, které není optimální. Ukažme si několik příkladů pro VertexCoverGradient.

Prvním příkladem je rodina grafů, které mají tvar hvězdy (star graf). V těchto grafech existuje jeden uzel — střed, který sousedí se všemi ostatními uzly, přičemž ostatní uzly už s žádným jiným uzlem nesousedí. Graf se dá zobrazit tak, že připomíná hvězdu. Pokud VertexCoverGradient vybere v první iteraci jako uzel k odebrání střed, nevyhnutně skončí v lokálním minimu, které není globálním minimem. Pokud vybere v první iteraci libovolný jiný uzel, skončí v globálním minimu.

Dalším příkladem jsou bipartitní grafy. Bipartitní graf je takový graf, jehož uzly můžeme rozdělit do dvou disjuktních množin, přičemž pro každý uzel platí, že může sousedit pouze s uzly z množiny, do které nepatří. Pokud tyto množiny nemají stejný počet uzlů, je globálním minimem menší množina. Lokálních minim může existovat více, příkladem jednoho z nich je větší množina uzlů.

Protože počet iterací cyklu na řádku 4 procedury VertexCoverGradient je konečný (je omezen seshora počtem uzlů v grafu), lze všechny možné průběhy této procedury zakreslit do konečného stromu. Listy tohoto stromu odpovídají lokálním minimům (pozor, více listů může odpovídat stejnému lokálnímu minimu!). Pak platí, že čím hlouběji se list nachází, tím je blíže globálnímu minimu. Listy, které jsou v maximální hloubce odpovídají globálním minimům. Zobrazíme si takový strom pro příklad tříprvkového grafu.

Vidíme, existuje jedno globální minimum (prostřední uzel) a jedno lokální minimum,



které není globálním (okrajové uzly).

Techniky, které se používají k částečnému řešení problému uváznutí v lokálním minimu, jsou většinou založeny na fyzikálních analogiích. Pokud při výběru dalších řešení z okolí aktuálního řešení umožníme zvolit i řešení s horší cenou než aktuální, můžeme někdy zabránit uváznutí tím, že „vyskočíme“ z lokálního minima (nebo z cesty do něj). V dalším průběhu algoritmu se pak lokálnímu minimu, ze kterého jsme vyskočili, můžeme vyhnout. U vrcholového pokrytí by to na přechozím obrázku znamenalo „přeskočit“ do uzlu stromu, který je ve stromu výše (tedy z lokálního minima uprostřed bychom se dostali opět do kořene).

Výběr řešení z okolí vypadá následovně. Pro jednoduchost předpokládejme, že řešíme minimalizační problém.

1. Pro aktuální řešení x vybereme náhodné řešení y z okolí x (s co nejvíce uniformním rozložením pravděpodobnosti),
2. pokud $\mathrm{cost}(x) < \mathrm{cost}(y)$ (y je horší řešení než x), pak s pravděpodobností

$$e^{-(|\mathrm{cost}(y)-\mathrm{cost}(x)|)/T}, \tag{6.2}$$

   kde T je konstanta, vybereme jako další řešení y (alternativou je ponechání x),
3. pokud je $\mathrm{cost}(x) \geqslant \mathrm{cost}(y)$, je dalším řešením je y.

Konstanta T určuje míru nestability výběru, čím je T vyšší, tím vyšší je i (6.2). Všimněte si také, že čím menší je rozdíl mezi cenami x a y, tím je pravděpodobnost výběru y vyšší. Vysvětlením může být to, že pokud se blížíme k lokálnímu minimu, většinou je rozdíl v cenách aktuálního řešení a prvků z jeho okolí menší než pokud jsme dále od něj. Existuje tedy větší šance, že se blížíme k uváznutí v lokálním minimu. Popsaný přístup označujeme jako *Metropolis algoritmus*. Vzhledem ke změně výběru dalšího řešení musíme také změnit podmínku ukončení algoritmu. Algoritmus nemůžeme ukončit stejně jako u gradientní metody (můžeme se totiž nacházet v lokálním minimu a ukončením algoritmu bychom zabránili možnosti z něj vyskočit). Mezi často používané podmínky ukončení algoritmu patří zejména:

- algoritmus provede předem daný počet iterací,
- algoritmus dostatečný počet iterací za sebou nevybere jiné řešení než aktuální.

Metropolis algoritmu obecně trvá poměrně dlouhou dobu, než se ustálí (tj. trvá poměrně hodně iterací, než se dostane k lokálnímu (globálnímu) minimu). Důvodem je fakt, že i když se nachází blízko minima, pravděpodobnost (6.2) je poměrně vysoká. Tento problém se dá vyřešit tak, že s rostoucím počtem iterací bude (6.2) klesat (pochopitelně pro stejné x a y). V počátečních iteracích tak exituje poměrně velká šance, že algoritmus unikne z lokálního

Algoritmus 20 Vrcholové pokrytí simulovaným žíháním

```
1:  procedure VertexCoverAnnealing((V, E), k)
2:      C ← V
3:      i ← 0
4:      ch ← true                          ▷ ch je true, pokud jsem změnil v minulé iteraci C
5:      while i ⩽ k do
6:          if ch then                                                  ▷ Spocitam okoli C
7:              F ← ∅
8:              for u ∈ C do                          ▷ Odebírání hran, musím testovat na pokrytí
9:                  x ← true
10:                 for {v, w} ∈ E do
11:                     if v ∉ C \ {u} and w ∉ C \ {u} then
12:                         x ← false
13:                         break
14:                 if x then
15:                     F ← F ∪ {C \ {u}}
16:             for u ∈ V \ C do                                        ▷ Přidávání hran
17:                 F ← F ∪ {C ∪ {u}}
18:         Vyber náhodný prvek S ∈ F                          ▷ s uniformním rozložením
19:         if |S| > |C| then
20:             S pravděpodobností e^{−1/T(i)} proveď C ← S a ch ← true
21:             Jinak ch ← false
22:             continue
23:         i ← i + 1
24:     return C
```

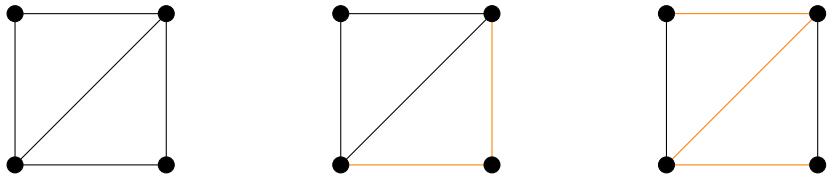
minima se špatnou cenou, ale v pozdějších iteracích už bude (6.2) dostatečná malá pro to, aby se algoritmus zastavil v minimu s lepší cenou. Technicky to lze zajistit tak, že konstantu T nahradíme klesající funkcí závislou na počtu iterací. Popsaná úprava má jméno *simulované žíhání*.

Na závěr kapitoly si ukážeme úpravu VertexCoverGradient na techniku simulovaného žíhání (Algoritmus 2). Abychom dosáhli uniformního rozložení pravděpodobnosti při volbě na řádku 18, vygeneruje algoritmus na řádcích 5 až 17 okolí aktuálního řešení C. Toto okolí se generuje jenom v případě, že se v předchozí iteraci C změnilo (proměnná ch). Za $T(i)$ je možné zvolit libovolnou klesající funkci takovou, aby $0 \leqslant e^{-1/T(i)} \leqslant 1$.

Maximální řez

Problém nalezení maximálního řezu v grafu je problémem, který lze snadno řešit pomocí lokálního vyhledávání. Zajímavé je, že pro výběr následujícího prvku z okolí aktuálního řešení se nepoužívá gradientní metoda ani simulované žíhání. Tato vylepšení nemají totiž na výkon algoritmu (tj. jak dobrý řez vrací) nijak zásadní vliv.

Definice 15. Nechť $G = (V, E)$ je neorientovaný graf a V_1, V_2 jsou disjunktní podmnožiny V takové, že $V_1 \cup V_2 = V$. Pak množinu hran $C = \{(u, v) \mid u \in V_1, v \in V_2\}$ označujeme jako *řez grafu*. Cenou řezu C je $|C|$.



(a) Ohodnocený graf (b) Řez s cenou 2 (c) Řez s cenou 3

Obrázek 6.3: Řez grafu

Příklady pojmu z předchozí definice jsou na obrázku 6.3. Problém nalezení maximálního řezu v grafu je definován následovně

Maximální řez grafu	
Instance:	Neorientovaný graf $G = (V, E)$.
Přípustná řešení:	$\{C \subseteq E \mid C \text{ je řez grafu } G\}$
Cena řešení:	$\mathtt{cost}(C, G) = \lvert C \rvert$
Cíl:	maximum

Aproximační algoritmus, který využívá techniku lokálního vyhledávání, si v každém kroku uchovává množiny V_1 a V_2 a jim odpovídající řez C. Na začátku algoritmus zvolí tyto množiny V_1 a V_2 libovolně, například $V_1 = V$, $V_2 = \emptyset$. Okolí aktuálního řezu C jsou všechny řezy, které vzniknou přesunem jednoho uzlu takového, že počet hran, které vedou z uzlu do množiny, ve které se nachází, je větší, než počet hran, které vedou do opačné množiny, mezi množinami V_1 a V_2. Z okolí pak algoritmus vybírá libovolný řez.

V pseudokódu používáme následující značení. Pro uzel v je V_v ta z množin V_1, V_2, do které tento prvek patří.

Věta 11. *Složitost algoritmu* MaxCut *je* $O(|V| \cdot |E|)$.

Důkaz. Cyklus na řádku 7 se dá provést se složitostí $O(|V|)$. Protože maximální cena řezu je $|E|$ a v každé iteraci cyklu na řádku 5 zvýšíme cenu řezu minimálně o 1, je maximální počet iterací tohoto cyklu $|E|$. □

Algoritmus 21 Maximální řez pomocí lokálního vyhledávání

```
1:  procedure MaxCut((V, E))
2:      V_1 ← V
3:      V_2 ← ∅
4:      x ← TRUE
5:      while x do
6:          x ← FALSE
7:          for u ∈ V do
8:              if |{{u, v} | V_v = V_u}| > |{{u, v} | V_u ≠ V_u}| then
9:                  Přesun u mezi V_1 a V_2
10:                 x ← TRUE
11:                 break
12:     return C odpovídající V_1 a V_2.
```

Literatura

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*, third edition, MIT press, 2009.

[2] Jon Kleinberg, Éva Tardos. *Algorithm Design*. Pearson, 2005.

[3] Christos H. Papadimitriou, Sanjoy Dasgupta, Umesh Vazirani. *Algorithms*. McGraw-Hill Education, 2006.

[4] Ronald L. Graham, Donald E. Knuth, Oren Patashnik. Concrete Mathematics: A Foundation for Computer Science. Addison-Wesley Professional, 1994.