Složitost I

1 Složitost algoritmů a prostředky pro její popis

Definice *(Velikost dat, krok algoritmu)*

Velikost dat je obvykle počet bitů, potřebných k jejich zapsání, např. data D jako pro čísla $a_1, \ldots, a_n$ je velikost dat $|D| = \sum_{i=1}^{n} \lceil \log a_i \rceil$.

Krok algoritmu je jedna operace daného abstraktního stroje (např. Turingův stroj, stroj RAM), zjednodušeně jde o nějakou operaci, proveditelnou v konstantním čase (např. aritmetické operace, porovnání hodnot, přiřazení – pro jednoduché číselné typy).

Definice *(Časová složitost)*

Časová složitost je funkce $f : \mathbb{N} \to \mathbb{N}$ taková, že $f(|D|)$ udává počet kroků daného algoritmu, pokud je spuštěn na datech D.

Definice *(Asymptotická složitost)*

Řekneme, že funkce $f(n)$ je *asymptoticky menší nebo rovna* než $g(n)$, značíme $f(n)$ je $O(g(n))$, právě tehdy, když

$$\exists c > 0 \; \exists n_0 \; \forall n > n_0 : 0 \leq f(n) \leq c \cdot g(n)$$

Funkce $f(n)$ je *asymptoticky větší nebo rovna* než $g(n)$, značíme $f(n)$ je $\Omega(g(n))$, právě tehdy, když

$$\exists c > 0 \; \exists n_0 \; \forall n > n_0 : 0 \leq c \cdot g(n) \leq f(n)$$

Funkce $f(n)$ je *asymptoticky stejná* jako $g(n)$, značíme $f(n)$ je $\Theta(g(n))$, právě tehdy, když

$$\exists c_1, c_2 > 0 \; \exists n_0 \; \forall n > n_0 : 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

Funkce $f(n)$ je *asymptoticky ostře menší* než $g(n)$ ($f(n)$ je $o(g(n))$, když

$$\forall c > 0 \; \exists n_0 \; \forall n > n_0 : 0 \leq f(n) < c \cdot g(n)$$

Funkce $f(n)$ je *asymptoticky ostře větší* než $g(n)$ ($f(n)$ je $w(g(n))$, když

$$\forall c > 0 \; \exists n_0 \; \forall n > n_0 : 0 \leq c \cdot g(n) < f(n)$$

Poznámka

Asymptotická složitost zkoumá chování algoritmů na velkých datech, zařazuje je podle toho do kategorií. Zanedbává multiplikativní a aditivní konstanty.

2 Algoritmy rozděl a panuj

Definice *(Metoda rozděl a panuj)*

Rozděl a panuj je metoda návrhu algoritmů (ne strukturované programování), která má 3 kroky:

1. *rozděl* – rozdělí úlohu na několik podúloh stejného typu, ale menší velikosti
2. *vyřeš* – vyřeší podúlohy a to buď přímo pro dostatečně malé, nebo rekurzivně pro větší
3. *sjednoť* – sjednotí řešení podúloh do řešení původní úlohy

TODO: Strassen

Poznámka *(Vytvoření rekurentní rovnice)*

Pro časovou složitost algoritmů typu rozděl a panuj zpravidla dostávám nějakou rekurentní rovnici.

- $T(n)$ budiž doba zpracování úlohy velikosti n, za předpokladu, že $T(n) = \Theta(1)$ pro $n \leq n_0$.
- $D(n)$ budiž doba na rozdělení úlohy velikosti n na a podúloh stejné velikosti $\frac{n}{c}$.
- $S(n)$ budiž doba na sjednocení řešení podúloh velikosti $\frac{n}{c}$ na jednu úlohu velikosti n.

Dostávám rovnici

$$T(n) = \begin{cases} D(n) + aT(\frac{n}{c}) + S(n) & n > n_0 \\ \Theta(1) & n \leq n_0 \end{cases}$$

Metody řešení rekurentních rovnic

Poznámka

Při řešení rekurentních rovnic:

- Zanedbávám celočíselnost ($\frac{n}{2}$ místo $\lceil \frac{n}{2} \rceil$ a $\lfloor \frac{n}{2} \rfloor$)
- Nehledím na konkrétní hodnoty aditivních a multiplikativních konstant, asymptotické notace používám i v zadání rekurentních rovnic, i v jejich řešení.

Věta *(Substituční metoda)*

1. Uhodnu asymptoticky správné řešení
2. Indukcí ověřím správnost (zvláště horní a dolní odhad)

Věta *(Metoda „kuchařka“ (Master Theorem))*

Nechť $a \geq 1, c > 1, d \geq 0 \in \mathbb{R}$ a nechť $T : \mathbb{N} \to \mathbb{N}$ je neklesající funkce taková, že $\forall n$ tvaru c^k platí

$$T(n) = aT(\frac{n}{c}) + \Theta(n^d)$$

Potom

1. Je-li $\log_c a \neq d$, pak $T(n)$ je $\Theta(n^{\max\{\log_c a, d\}})$
2. Je-li $\log_c a = d$, pak $T(n)$ je $\Theta(n^d \log_c d)$

Věta *(Master Theorem, varianta 2)*

Nechť $0 < a_i < 1$, kde $i \in \{1, \ldots, k\}$ a $d \geq 0$ jsou reálná čísla a nechť $T : \mathbb{N} \to \mathbb{N}$ splňuje rekurenci

$$T(n) = \sum_{i=1}^{k} T(a_i \cdot n) + \Theta(n^d)$$

Nechť je číslo x řešením rovnice $\sum_{i=1}^{k} a_i^x = 1$. Potom

1. Je-li $x \neq d$ (tedy $\sum_{i=1}^{k} a_i^d \neq 1$), pak $T(n)$ je $\Theta(n^{\max\{x,d\}})$
2. Je-li $x = d$ (tedy $\sum_{i=1}^{k} a_i^d = 1$), pak $T(n)$ je $\Theta(n^d \log d)$

Algoritmus *(Hledání mediánu v lin. čase)*

Vstupem je (neuspořádáná) posloupnost n (navzájem různých) čísel. Výstup je $\lfloor \frac{n}{2} \rfloor$-té nejmenší číslo z nich. Složitost budeme měřit *počtem porovnání*.

Algoritmus za použití techniky rozděl a panuj, hledá obecně k-té nejmenší číslo:

1. Vybrat pivota a rozdělit posloupnost pomocí $n-1$ porovnání na 3 oddíly: čísla menší než pivot (a prvků), pivota samotného a čísla větší než pivot ($n - a - 1$ prvků).

2. (a) Pokud je $a+1 < k$, hledám rekurzivně $k-a+1$-tý prvek v $n-a-1$ prvcích
   (b) Pokud je $a+1 = k$, vracím pivot
   (c) Pokud je $a+1 > k$, hledám rekurzivně k-tý prvek v a prvcích

Rekurzivní vzorec $T(n) = T(n-1) + (n-1)$ v nejhorším případě, což dává $\Theta(n^2)$.

Vylepšení, garantující dobrý výběr pivota:

1. Rozdělit posloupnost na pětice (poslední může být neúplná)
2. V každé pětici najít medián
3. Rekurzivně najít medián těchto mediánů
4. Použít ho jako pivot pro dělení celé posloupnosti, protože je větší než alespoň 3 prvky z alespoň $1/2$ pětic (až na zakrouhlení), je větší než alespoň $1/2 \cdot 3/5 = 3/10$ prvků (a také menší než alespoň $3/10$ prvků)
5. Z toho mám vždy zaručeno, že zahodím alespoň $3/10$ prvků pro následující krok.

Vzorec potom vychází: $T(n) = c \cdot \frac{n}{5} + T(\frac{n}{5}) + (n-1) + T(\frac{7}{10}n)$ a podle kuchařky verze 2 nebo substituční metodou se dá vyčíslit jako $T(n) = \Theta(n)$.

3 Hladové algoritmy

Motivace

Problém 1

Dán souvislý neorientovaný graf $G = (V, E)$ a funkce $d : E \to \mathbb{R}^+$, udávající délky hran. Najděte minimální kostru grafu G, tj. kostru $G' = (V, E')$ tak, že $\sum_{e \in E'} d(e)$ je minimální.

Problém 2

Je dána množina $S = \{1, \ldots, n\}$ úkolů jednotkové délky. Ke každému úkolu je dána lhůta dokončení $d_i \in \mathbb{N}$ a pokud $w_i \in \mathbb{N}$, kterou je úkol i penalizován, není-li hotov do své lhůty. Najděte rozvrh (permutaci úkolů od času 0 do času n), který minimalizuje celkovou pokutu.

Problém 3

Je dána množina $S = \{1, \ldots, n\}$ úkolů. Ke každému úkolu je dán čas jeho zahájení s_i ukončení f_i, $s_i \leq f_i$. Úkoly i a j jsou kompatibilní, pokud se intervaly $[s_i, f_i)$ a $[s_j, f_j)$ nepřekrývají. Najděte co největší množinu (po dvou) kompatibilních úkolů.

Matroid

Definice *(Matroid)*

Matroid je uspořádaná dvojice $M = (S, I)$, splňující:

- S je konečná neprázdná množina (*prvky matroidu* M)
- I je neprázdná množina podmnožin S (*nezávislé podmnožiny*), která má
  1. *dědičnou vlastnost*: $B \in I \;\&\; A \subseteq B \quad \Rightarrow \quad A \in I$,
  2. *výměnnou vlastnost*: $A, B \in I \;\&\; |A| < |B| \quad \Rightarrow \quad \exists x \in B \backslash A : A \cup \{x\} \in I$.

Věta *(O velikosti maximálních nezávislých podmnožin)*

Všechny *maximální* (maximální vzhledem k inkluzi) nezávislé podmnožiny v matroidu mají stejnou velikost.

Důkaz

Z výměnné vlastnosti, nechť $A, B \in I$ maximální, $|A| < |B|$, pak $A \cup \{x\} \in I, x \notin A$, což je spor.

Definice *(Vážený matroid)*

Matroid $M = (S, I)$ je *vážený*, pokud je dána funkce $w : S \to \mathbb{R}^+$ a její rozšíření na podmnožiny množiny S je definováno předpisem:

$$A \in S \;\Rightarrow\; w(A) = \sum_{x \in A} w(x)$$

Problém 1 a 2 – zobecněný

Pro daný vážený matroid nalezněte nezávislou podmnožinu s co největší vahou (*optimální množinu*). Protože váhy jsou kladné, vždy se bude jednat o maximální nez. množinu.

Problém 1 je spec. případ tohoto, protože můžeme uvažovat *grafový matroid* (nad hranami) – $M_G = (S, I)$, kde $S = E$ a pro $A \subseteq E$ platí $A \in I$, pokud hrany z A netvoří cyklus (tj. indukovaný podgraf tvoří les). Dědičná vlastnost této struktury je zřejmá, výměnnost se dá dokázat následovně: mějme $A, B \subseteq E, |A| < |B|$. Pak lesy z A, B mají $n - a > n - b$ stromů (vč. izolovaných vrcholů).

V B musí být strom, který se dotýká ≥ 2 různých stromů z A, ten obsahuje hranu, která není v žádném stromě A a netvoří cyklus a tak ji můžu k A přidat. Váhovou funkci převedu na hledání maxim: $w(e) = c - d(e)$, kde c je dost velká konstanta, aby všechny váhy byly kladné. Algoritmus pak najde max. množinu, kde hrany netvoří cyklus, implicitně bude mít $n - 1$ hran, takže půjde o kostru a její w bude maximální, tedy původní váha minimální.

Pro problém 2 zavedeme pojem *kanonického rozvrhu* – takového rozvrhu, kde jsou všechny *včasné* úkoly rozvrženy před všemi *zpožděnými* a uspořádány podle neklesající lhůty dokončení (tohle na celkové pokutě nic nezmění a máme bijekci mezi množinou včasných úkolů a kanonickými rozvrhy).

Optimální množinu pak lze hledat jen nad kanonickými rozvrhy – nezávislou množinou úkolů nazvu takovou, pro kterou existuje kanonický rozvrh tak, že žádný úkol v ní obsažený není zpožděný. Potom hledání max. nezávislé množiny při ohodnocení pokutami je hledání nejmenší pokuty (odeberu co nejvíc z možné celkové pokuty).

Pak zbývá dokázat, že takto vytvořená struktura je matroid. Dědičná vlastnost je triviální – vyhodím-li něco z nezávislé množiny, nechám v rozvrhu mezery a bude platit stále. Pro výměnnou vlastnost zavedu pomocnou funkci $N_t(C) = |\{i \in C | d_i \leq t\}|$, udávající počet úkolů z množiny C se lhůtou do času t. Pak množina C je nezávislá, právě když $\forall t \in \{1, \ldots, n\} : N_t(C) \leq t$.

Pak máme-li 2 nezávislé A, B, $|B| > |A|$, označíme k největší okamžik takový, že $N_k(B) \leq N_k(A)$, tj. od $k+1$ dál platí $N_t(A) < N_t(B)$. To skutečně nastane, protože $|B| = N_n(B) > N_n(A) = |A|$. Pak určitě v B je ostře víc úkolů s $d_i = k + 1$ než v A, tj. $\exists x \in B \backslash A$ se lhůtou $k + 1$. $A \cup \{x\}$ je nezávislá, protože $N_t(A \cup \{x\}) = N_t(A)$ pro $t \leq k$ a $N_t(A \cup \{x\}) \leq N_t(B)$ pro $t \geq k + 1$.

Hladový algoritmus na váženém matroidu

Algoritmus *(Greedy)*

Mám zadaný matroid $M = (S, I)$, kde $S = \{x_1, \ldots, x_n\}$ a $w : S \to \mathbb{R}^+$. Potom

1. $A := \emptyset$, setřiď a přeznač S sestupně podle vah
2. pro každé x_i zkoušej: je-li $A \cup \{x_i\} \in I$, tak $A := A \cup \{x_i\}$
3. vrať A jako maximální nezávislou množinu

Pokud přidám nějaké x_i, nikdy nezruším nezávislost množiny; s už přidanými prvky se nic nestane. Časová složitost je $\Theta(n \log n)$ na setřídění podle vah, testování nezávislosti množiny závisí na typu matriodu ($f(n)$), takže celkem něco jako $\Theta(n \log n + n \cdot f(n))$.

Důkaz

- Vyhození prvků, které samy o sobě nejsou nezávislé množiny, nic nezkazí (z dědičné vlastnosti).
- První vybrané x_i je „legální“ (nezablokuje mi cestu), protože mezi max. nez. množinami určitě existuje taková, která jím mohla vzniknout (z výměnné vlastnosti – vezmeme první prvek, který je sám o sobě nez. mn. a s pomocí nějaké max. nez. množiny B ho doplníme, vzniklé $\{x_i\} \cup (B \backslash \{x_j\})$ musí být také maximální).

- Nalezení optimální množiny obsahující pevné (první vybrané) x_i v $M = (S, I)$ je ekvivalentní nalezení optimální množiny v $M' = (S', I')$, kde $S' = \{y \in S | \{x_i, y\} \in I\}$ a $I' = \{B \subset S \backslash \{x_i\} | B \cup \{x_i\} \in I\}$ (je-li S' neprázdná, je to matroid, vlastnosti se přenesou z původního; pokud je S' prázdná, tak algoritmus končí) a tedy když vyberu x_i, nezatarasím si cestu k optimu – stačí ukázat, že $A \cup \{x\}$ je optimální v M právě když A je optimální v M'.
- Takže když budu vybírat (indukcí opakovaně) x_i podle váhy, dojdu k optimální množině.

Algoritmus *(Hladový algoritmus na problém 3)*
Máme $S = \{1, \ldots, n\}$ množinu úkolů s časy startů s_i a konců f_i. Provedeme:

1. $A := \emptyset$, $f_0 := 0, j := 0$
2. setřiď úkoly podle f_i vzestupně a přeznač
3. pro každé i zkoušej: je-li $s_i \geq f_j$, pak $A := A \cup \{i\}$ a $j := i$
4. vrať A jako max. množinu nekryjících se úkolů

Složitost je $n \log n$ na setřídění, zbytek už lineární. Sice to funguje, ale tahle struktura NENÍ matroid – nesplňuje výměnnou vlastnost. Algoritmus má ale stejný důkaz jako předchozí na matroidech (legálnost hladového výběru – existuje max. množina obsahující prvek 1 – a existenci optimální množiny – převod na „menší“ zadání).

4 Grafové algoritmy

Pravidla efektivního průchodu grafem

1. Každou hranou projít max. 1× v každém směru
2. Hranou, kterou do vrcholu přijdu, se smím vrátit, až když z uzlu už nevede neprozkoumaná hrana
3. Hranou, kterou vstoupím do vrcholu již navštíveného, se vracím zpět

Libovolný algoritmus, který toto implementuje, funguje: zabraňuje cyklení; nejde-li použít žádné z pravidel, jsem v počátku a každou hranou jsem prošel oběma směry, tj. je korektní (důkaz indukcí) a má lineární složitost pro konstantní práci na 1 hraně.

Prohledávání do hloubky (DFS) a do šířky (BFS) tyto podmínky splňují. Předpokládáme zadání grafu seznamem sousedů.

Aplikace DFS na neorientovaném grafu

Algoritmus *(DFS)*
DFS samotné je napřed obarvení všech hran na bílo a potom postupně od 1 spouštění procedury NAVŠTIV z těch, které jsou stále bílé. Procedura NAVŠTIV je rekurzivní průchod do všech dosažitelných vrcholů z daného začátku, postupně barví hrany na šedo (při procházení dál) a na černo (při opouštění) a označuje stromové a zpětné hrany.

Algoritmus *(Hledání komponent souvislosti)*
Jen před každým prvním zavoláním NAVŠTIV (v hlavním cyklu) zvedneme o 1 speciální čítač KOMP (na zač. je 0), tak dostaneme počet komponent. Do NAVŠTIV přidáme označení právě zpracovávané hrany číslem právě procházené komponenty (tj. hodnotou čítače KOMP).

Definice *(Dvousouvislost, artikulace, 2-souvislá komponenta)*
V grafu $G = (V, E)$:

- Vrchol $v \in V$ nazveme *artikulací*, pokud po odstranění v přestane graf G být souvislý.
- G je *dvousouvislý*, pokud nemá žádné artikulace.
- $K \subseteq E$ je *dvousouvislá komponenta*, pokud je to maximální (vzhledem k inkluzi) množina taková, že každé dvě hrany z K leží na prosté kružnici (kružnici bez opakování vrcholů).

Poznámka *(Vlastnosti artikulace)*
$i \in V$ je artikulace, právě když

1. i není kořen DFS stromu a má potomka j takového, že z podstromu s kořenem j nevede žádná zpětná hrana do nějakého předchůdce i, nebo
2. i je kořen DFS stromu a má aspoň 2 potomky (vedou z něj ≥ 2 stromové hrany)

Definice *(Funkce* low(i)*)*
Každému vrcholu přiřadíme pořadí jeho objevení (přebarvení z bílé na šedou) $d(i)$. Cestu z vrcholu i v DFS stromě nazveme *přípustnou*, pokud vede z i dál po libovolném počtu stromových hran a případně končí jednou zpětnou hranou. Definujeme funkci:

$$\text{low}(i) = \min\{d(j) | \text{ z } i \text{ do } j \text{ vede přípustná cesta}$$

Ekvivalentní rekurzivní definice:

$$\text{low}(i) = \min\{x, y, z\}$$

kde $x = d(i)$, y je minimum $d(j)$ přes zpětné hrany z i a z je minimum low(j) přes stromové hrany z i.

Algoritmus *(Dvousouvislost s pomocí DFS)*
Při procházení grafu pomocí DFS ještě počítám low(i), a to následovně:

- low$(i) := d(i)$ při objevení vrcholu
- při procházení sousedů $(\forall j)$:
  - j je bílý – rekurze jako DFS, (i, j) je stromová
  - j je šedý – pokud je j rodič i, tak nic; jinak je (i, j) zpětná a low$(i) := \min\{\text{low}(i), d(j)\}$
  - j je černý – ((j, i) byla zpětná), pak nic
- při backtrackingu – návrat z j do i po stromové hraně (i, j): low$(i) := \min\{\text{low}(i), \text{low}(j)\}$

Artikulací je potom takové i, že

1. není kořen DFS stromu a existuje potomek j takový, že low$(j) \geq d(i)$
2. je kořen DFS stromu a má alespoň 2 potomky

Jako artikulaci tedy označíme i při návratu po stromové hraně (i, j), pokud bude platit tamta rovnice – a u kořenu nejdřív zrušíme příznak, že jde o kořen a při druhém průchodu bude určitě označen jako artikulace.

Pokud chceme navíc rozdělit hrany do dvousouvislých komponent, ukládáme je do zásobníku (stromové i zpětné), a pokud zjistíme při backtrackingu po nějaké hraně (i, j), že i je artikulace, tak uřízneme všechno ze zásobníku od (i, j) (včetně) výš a označíme jako jednu komponentu dvousouvislosti.

Aplikace DFS na orientovaném grafu

Algoritmus *(DFS na orientovaném grafu)*
Běží úplně stejně jako na neorientovaném, k tomu počítá čas a označuje $d(i)$ čas navštívení a $f(i)$ čas opuštění každého vrcholu. Označuje hrany jako stromové, zpětné a navíc ještě dopředné a příčné (když cílový vrchol hrany má černou barvu a $d(i) < d(j)$, resp. naopak).

Potom

1. Stromové hrany tvoří orientovaný les (DFS les).
2. Vrchol j je následníkem i v DFS stromě, právě když v čase $d(i)$ vedla z i do j cesta pouze po bílých vrcholech.
3. Intervaly $[d(i), f(i)]$ tvoří dobré uzávorkování; j je následníkem i, právě když $[d(j), f(j)] \subset [d(i), f(i)]$.
4. Hrana (i, j) je příčná, právě když $f(j) < d(i)$ – všechny příčné hrany jdou „proti pořadí stromů“.

Algoritmus *(Topologické třídění)*

1. Graf je obsahuje cyklus, právě když DFS najde zpětnou hranu.
2. Setřídění vrcholů acyklického grafu podle klesajících $f(i)$ je topologické.

Důkaz

Když graf obsahuje cyklus, vezmu i, který je objeven z cyklu jako první a j jeho předchůdce v cyklu. Pak z i do j je v čase $d(i)$ cesta po bílých vrcholech, takže j je následníkem i a proto (j, i) se stane zpětnou. Pokud najdu zpětnou hranu, z definice zpětné hrany mám v grafu cyklus.

Pro libovolnou hranu (i, j) musí platit $f(i) > f(j)$. Rozbor případů podle barvy j v okamžiku prohlížení (i, j):

1. j je bílý $\Rightarrow$ (i, j) je stromová, takže j je následník i, z vlastnosti 3
2. j je šedý, pak jde o zpětnou hranu, takže v grafu je cyklus
3. j je černý – z toho triviálně plyne $f(j) < f(i)$

Definice *(Silně souvislá komponenta, transponovaný graf)*

Silně souvislá komponenta je taková podmnožina vrcholů grafu, kde pro každé dva vrcholy existují cesty v obou směrech a navíc je tato množina maximální vzhledem k inkluzi.

Mějme orientovaný graf $G = (V, E)$. *Transponovaný graf* označme $G^T = (V, E')$, kde $(u, v) \in E' \equiv (v, u) \in E$. Je zřejmé, že transponované grafy mají identické silně souvislé komponenty.

Algoritmus *(Silně souvislé komponenty)*

1. DFS(G) a vytvoření spojáku vrcholů podle klesajícího $f(i)$
2. vytvoření G^T
3. DFS(G^T), které zpracovává vrcholy dle seznamu z fáze 1

Důkaz

- Každá SSK je obsažena v jediném DFS stromě fáze 1 a navíc v něm tvoří podstrom (tohle se dá ukázat obrázkem). Ukážeme, že stromy z 3. fáze odpovídají SSK. Nechť x je kořen T_1 (1. stromu z fáze 3), potom je kořenem S_l, posledního stromu z fáze 1. Vezmu $y \in T_1$ libovolné, pak z x vede cesta do y v G^T, tedy z y vede cesta do x v G. Musí tedy nutně být $y \in S_l$ (jinak by x nebyl kořen), a tím pádem x a y jsou ve stejné SSK.
- Protože y bylo libovolné, celé T_1 je podmnožinou jedné SSK. Žádný vrchol z $T_2, T_3, \ldots$ ale v této SSK už nemůže být, protože z x do něj v G^T nevede cesta, takže T_1 je právě jedna SSK.
- T_1 tvoří v S_l kořenový podstrom. Po jeho odstranění mohu dostat další stromy, které přidám do seznamu a mám stejnou situaci. Indukcí mohu pokračovat, tj. každý strom fáze 3 je SSK.
- Algoritmus je lineární, protože DFS je lineární a vytvoření transpozice grafu taky – vezmu seznam sousedů G, založím nový prázdný seznam sousedů a pro všechny sousedy j nějakého vrcholu i v G zapíšu i do seznamů sousedů j v G^T.

Rovinné grafy, věta o planárním separátoru

Definice

Neorientovaný graf $G = (V, E)$ se nazývá *rovinný*, pokud existuje jeho nakreslení (vnoření) do roviny ($\mathbb{E}_2$) takové, že se žádné 2 hrany neprotínají (rovinnost a nakreslení lze vyřešit v lineárním čase).

Pro $G = (V, E)$ rovinný graf máme *duální (multi)graf* $G^* = (V^*, E)$: vrcholy G^* odpovídají stěnám rovinného nakreslení G a hrany G^* jsou stejné jako hrany G. Pokud mají 2 stěny společnou více než 1 hranu, je G^* multigraf. G^* je vždy souvislý, a pro G souvislý jsou G a G^{**} izomorfní. Konstrukce duálního grafu je také možná v lin. čase.

Rovinný graf je *triangulovaný*, pokud je každá jeho stěna omezená 3 hranami. Jeho G^* je pak 3-regulární; (nějakou) triangulaci grafu je také možné zkonstruovat v lin. čase.

Věta *(O planárním separátoru)*
Nechť $G = (V, E)$ je neorientovaný rovinný graf. Pak existuje rozdělení množiny V na disjunktní množiny A, B, S takové, že:

- $|A| \leq \frac{2}{3}n$, $|B| \leq \frac{2}{3}n$
- $|S| \leq 4\sqrt{n}$
- $(A \times B) \cap E = \emptyset$ (tj. S separuje A od B)

Takové rozdělení je navíc možné zkonstruovat v čase $O(m + n)$.
Důkaz

- BÚNO nechť je G souvislý (pokud není, přidáním hran nic nezkazím). Zkonstruujeme rovinné nakreslení G, vybereme libovolné $s \in V$ a pomocí BFS z vrcholu s rozdělíme V na vrstvy $L_0, L_1, \ldots$ (kde $L_0 = \{s\}$). Dodefinujeme $L_{q+1} = \emptyset$ a $l_i = |L_i|$ pro $i \in \{0, \ldots, q+1\}$.
- Pak pro každou hranu (x, y) platí, že $\exists i : x, y \in L_i$ nebo $\exists i : x \in L_i, y \in L_{i\pm1}$. Z toho je vidět, že $\forall i$ je L_i separátor, oddělující $A = \cup_{j<i} L_j$ a $B = \cup_{j>i} L_j$.
- Označme t_1 index takový, že L_{t_1} obsahuje $\frac{n}{2}$-tý vrchol, navštívený BFS. Předpokládejme, že $l_{t_1} > 4\sqrt{n}$, jinak není co dokazovat. Pak existují indexy $t_0 \leq t_1$ a $t_2 \leq t_1$ takové, že $l_{t_0} \leq \sqrt{n}$ a $l_{t_2} \leq \sqrt{n}$ a $t_2 - t_0 \leq \sqrt{n}$ (vezmu $t_0 = \max_t\{t < t_1, |L_t| < \sqrt{n}\}$ a t_2 symetricky, pak kdyby byly od sebe dál než o $\sqrt{n}$, muselo by $\cup_{t_0<t<t_2} L_t$ mít víc než n vrcholů).
- Označíme $C := \cup_{t<t_0} L_t$, $D := \cup_{t_0<t<t_2} L_t$, $E := \cup_{t>t_2} L_t$. Víme, že $|C|, |E| < \frac{n}{2}$, předpokládáme, že $|D| > \frac{2}{3}n$, jinak není co dokazovat ($A :=$ větší z C, E, $B :=$ druhý z nich $\cup D$, $S := L_{t_0} \cup L_{t_2}$).
- Nyní nalezneme separátor S' o velikosti $\leq 2\sqrt{n}$, který rozděluje D v poměru 2 : 1 nebo lepším, pak $S := S' \cup L_{t_0} \cup L_{t_2}$ a $A :=$ větší z $C, E \cup$ menší kus D a $B :=$ menší z $C, E \cup$ větší kus D. Tím budeme hotovi, protože

$$|A| \leq (n - |D|) + \frac{1}{2}|D| \text{ (hodně hrubý odhad většího z } C, E + \text{ menší kus } D \text{)} \qquad = n - \frac{1}{2}|D| \leq \frac{2}{3}n$$
$$|B| \leq \frac{1}{2}(n - |D|) + \frac{2}{3}|D| \text{ (odhad menšího z } C, E + \text{ větší kus } D \text{)} \qquad \leq \frac{1}{2}n + \frac{1}{6}n = \frac{2}{3}n$$

- Vezmu samotné D spolu s vrcholem s, a s spojím hranou s každým vrcholem vrstvy L_{t_0+1} (graf zůstane rovinný). Vyrobím jeho kostru T – pro každý vrchol $v \in L_{t_2-1}$ vyberu hranu, jdoucí do L_{t_2-2} a postupně hrany po vrstvách až do s. Protože $t_2 - t_0 \leq \sqrt{n}$, platí, že cesta v kostře T odkudkoliv kamkoliv je dlouhá max. $2\sqrt{n}$. Graf ztrianguluji, výsledek triangulace nechť se jmenuje $G' = (V', E')$. Veškeré hrany mimo kostru, tj. $E' \backslash T$ nazveme příčky, a tyto příčky reprezentují všechny cykly v triangulaci.
- Potom vezmu G^* duální ke G', jehož kostru T' tvoří právě příčky. Hledám cestu v kostře T, která tvoří požadovaný separátor, pomocí DFS nad duálním grafem. Vyberu list kostry duálu $E' \backslash T - v_0^*$ a orientuji hrany $E' \backslash T'$ směrem od něj. Pak při DFS nad $E' \backslash T$ pro každou hranu $e = (f, g)$ počítám:
  - $I_e =$ počet vrcholů grafu G' uvnitř cyklu, odpovídajícímu hraně
  - $d_e =$ počet vrcholů G^* na onom cyklu
  - $c_e =$ seznam vrcholů G^* na onom cyklu

  Postup tohoto výpočtu závisí na hraně, rozlišují se 4 případy:

  1. g je list kostry $E' \backslash T$, tj. stěna g grafu G' je omezena 2 hranami kostry T a jednou příčkou (přes ní vede hrana e): položím $I_e = 0$, $d_e = 3$, $c_e = (u, x, v)$ pro u, x, v vrcholy grafu G', mezi kterými leží stěna g – tedy konstantní práce.
  2. e má jednoho následníka v DFS stromě e', tj. stěna g je omezena 1 stromovou hranou ($\in T$) a 2 příčkami (přes druhou příčku vede e'):
     (a) pokud stromová hrana je součástí cyklu, položím $I_e = I'_e$, $d_e = d_{e'} + 1$ a $c_e = c_{e'} \circ (u)$ (kde u je „krajní“ vrchol stromové hrany),
     (b) když není, položím $I_e = I_{e'} + 1$, $d_e = d_{e'} - 1$ a $c_e = c_{e'} \backslash (u)$ (a u je na kraji seznamu, takže jde stále o konstantní práci).
  3. g je omezena 3 příčkami, tj. e má 2 následníky v DFS stromě – e' a e''. Nechť e vede mezi vrcholy u a v grafu G' a e', e'' mezi u, v a nějakým vrcholem y. Nechť cykly odpovídající e'' a e' mají společnou cestu p z y do

nějakého vrcholu x (ta může být nulově dlouhá). Pak položím $I_e = I_{e'} + I_{e''} + |p| - 1$, $d_e = d_{e'} + d_{e''} - 2|p| + 1$ (odečítám společnou cestu a zpátky přičítám x) a $c_e = (c_{e'} \backslash p) \circ (x) \circ (c_{e''} \backslash p)$.

Konstantnost práce se dokáže pro celý algoritmus dohromady, „amortizovaně“ – jakmile 1 vrchol vyhodím $1\times$ (protože je součástí p), už ho zpět nedostanu, jeden vrchol může být jen ve 2 seznamech – max. $2n$ vyhození vrcholů.

- Existuje příčka e taková, že $I_e \leq \frac{2}{3}n'$ a $n' - (I_e + d_e) \leq \frac{2}{3}n'$. Navíc e najdeme v průběhu práce DFS na kostře $E' \backslash T$, tedy v lineárním čase. Důkaz: nechť e je první hrana $E' \backslash T$, pro kterou platí během DFS $I_e + d_e \geq \frac{1}{3}n'$. Takové e
  - existuje, protože v listech je $I_e + d_e = 3$, ale v kořeni je $I_e + d_e = n'$ a BÚNO $n' \geq 9$ (malé grafy jsou nezajímavé).
  - splňuje požadovanou vlastnost – ověříme rozborem podle typu hrany:
    1. $I_e = 0 \leq \frac{2}{3}n'$
    2. (a) $I_e + d_e = I_{e'} + d_{e'} + 1 \leq \frac{1}{3}$ (nerovnost z výběru e), takže pokud $n' \geq 3$, platí
       (b) $I_e + d_e = I_{e'} + d_{e'}$, tj. platí „samo“
    3. $I_e + d_e = I_{e'} + I_{e''} + |p| - 1 + d_{e'} + d_{e''} - 2|p| + 1 = (I_{e'} + d_{e'}) + (I_{e''} + d_{e''}) - |p| \leq \frac{1}{3}n' + \frac{1}{3}n' - |p|$, takže platí
- Nyní zvolíme: $S' = c_e \backslash \{s\}$, $A' = I_e$ a $B' = V' \backslash \{I_e \cup c_e\}$ (když I_e chápeme jako množinu) a jsme hotovi.

Algoritmus *(Postup konstrukce planárního separátoru)*

1. Vytvoř rovinné nakreslení grafu G (lineární algoritmus Hoperofe/Tarjan)
2. Proveď BFS a rozděl V do vrstev $L_0, L_1, \ldots$
3. Najdi t_1 a pokud $l_{t_1} \leq 4\sqrt{n}$, skonči ($S = L_{t_1}, A = \cup_{t<t_1} L_t, B = \cup_{t>t_1} L_t$)
4. Najdi t_0 a t_2 takové, že $l_{t_0} \leq \sqrt{n}$, $l_{t_2} \leq \sqrt{n}$, $t_2 - t_0 \leq \sqrt{n}$ a rozděl V na C, D, E. Pokud $|D| \leq \frac{2}{3}n$, skonči ($S = L_{t_0} \cup L_{t_2}$, $A =$ největší z C, D, E, $B = \cup$ zbylých dvou).
5. Zkonstruuj graf $G' = (V', E')$ (přidáním vrcholu s k D a triangulací) a jeho kostru T o průměru $\leq 2\sqrt{n}$
6. Zkonstruuj graf $G^* = (V^*, E')$ duální k G' a jeho kostru $E' \backslash T$
7. Proveď DFS(V^*, $E' \backslash T$) a $\forall e \in (E' \backslash T)$ spočti I_e, d_e, c_e
8. Najdi příčku $e \in E' \backslash T$, pro kterou platí $I_e \leq \frac{2}{3}n'$ a $n' - (I_e + d_e) \leq \frac{2}{3}n'$. Nechť $X = \{I_e\}$ a $Y = V' \backslash (I_e \cup c_e)$, skonči ($S = L_{t_0} \cup L_{t_2} \cup c_e$, $A =$ větší z $\{X, Y\} \cup$ menší z $\{C, E\}$, $B =$ menší z $\{X, Y\} \cup$ větší z $\{C, E\}$

Celé běží v $\Theta(m + n)$.

5 Dolní odhady složitosti problémů

Definice *(Složitost problému)*

Složitost problému je složitost asymptoticky nelepšího **možného** algoritmu, který řeší daný problém (ne nejlepšího známého).

Každý konkrétní algoritmus dává horní odhad složitosti. Dolní odhady (až na triviální – velikost vstupu, výstupu) jsou složitější.

Věta *(Dolní odhad složitosti mediánu)*

Pro výběr k-tého z n prvků je třeba alespoň $n - 1$ porovnání, tj. problém je $\Omega(n)$.

Důkaz

TODO

Věta *(Dolní odhad složitosti třídění)*

Pro každý třídící algoritmus, založený na porovnávání prvků, existuje vstupní posloupnost, pro kterou provede $\Omega(n \log n)$ porovnání.

Důkaz

Nakreslím si rozhodovací strom jako model algoritmu – všechny vnitřní uzly odpovídají nějakému porovnání, které algoritmus provedl, jejich synové jsou operace, které nasledovaly po různých výsledcích toho porovnání. Listy odpovídají setříděným posloupnostem. Aby byl algoritmus korektní, musí mít strom listy se všemi $n!$ možnými pořadími prvků. Počet porovnání je výška stromu, proto stačí ukázat, že strom s $n!$ listy má výšku $\Theta(n \log n)$.

Označím výšku jako h, pak počet listů je $\leq 2^h$ a tedy $n! \leq 2^h$, tj. $h \geq \log n!$, stačí odhad faktoriálu jako $n^{\frac{n}{2}}$.

6 Amortizovaná složitost

Definice *(Amortizovaná složitost)*

Typicky se používá pro počítání časové složitosti operací nad datovými strukturami, počítá *průměrný čas na 1 operaci při provedení posloupnosti operací.* Dává realističtější odhad složitosti posloupnosti všech operací, než měření všech nejhorším případem.

Známe 3 metody amortizované analýzy (a používat budeme první dvě):

- agregační
- účetní
- potenciálová (je podobná účetní, vynecháme)

Problémy:

- *Inkrementace binárního čítače* – do binárního čítače délky k postupně přičteme n-krát jedničku. Počet bitových operací na 1 přičtení je v nejhorším případě $O(\log n)$, ale amortizovaně dojdeme k $O(1)$.
- *Vkládání do dynamického pole* – začnu s prázdným polem délky 1 a postupně vkládám n prvků. Pokud je stávající pole plné, alokujeme dvojnásobné a kopírujeme prvky. Počet kopírování prvků na jedno vložení je až $O(n)$, ale amortizovaně opět $O(1)$.

Algoritmus *(Agregační metoda)*

Postup: spočítáme nejhorší čas pro celou posloupnost n operací – $T(n)$, amortizovaný čas na 1 operaci je pak $\frac{T(n)}{n}$.

1. v průběhu n přičtení se i-tý bit překlopí $\lfloor \frac{n}{2^i} \rfloor$-krát, takže celková cena překlopení je $\leq n \cdot \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$, tj. amortizovaně na jedno přičtení $\frac{2n}{2} = \Theta(1)$
2. cena i-tého vložení do pole je $c_i = \begin{cases} i \text{ pokud } \exists k : i - 1 = 2^k \\ 1 \text{ jinak} \end{cases}$. Celkem dostávám $T(n) = \sum_{i=1}^{n} c_i = n + \sum_{j=0}^{\lfloor \log n \rfloor} 2^j \leq n + 2n = 3n$, takže na jedno vložení vyjde $\frac{3n}{n} = \Theta(1)$.

Algoritmus *(Účetní metoda)*

Postup: Od každé operace vyberu urč. pevný „obnos“, kterým onu operaci „zaplatím“. Pokud něco zbyde, dám to na účet, pokud bude oprace naopak dražší než onen obnos, z účtu vybírám. Zůstatek na účtu musí být stále nezáporný – pokud uspěji, obnos je amortizovaná cena 1 operace.

1. Při každém přičtení je právě jeden bit překlopen z 0 na 1. Proto každému bitu zavedeme účet a za přičtení budeme vybírat 2 jednotky. Jedna je použita na překlopení daného bitu z 0 na 1 a druhá uložena na právě jeho účet; překlápění z 1 na 0 jsou hrazeny z účtů (protože každý bit, který má nastavenu 1 má na účtu právě 1 jednotku, projde to). Amortizovaná cena tedy vyjde $2 = \Theta(1)$.
2. Od každého vložení vyberu 3 jednotky – na vlastní vložení, na překopírování právě vloženého prvku při příštím vložení a na příští překopírování odpovídajícího prvku v levé polovině pole (na pozici $n - \frac{s}{2}$), který obnos ze svého vložení vyčerpal. Po expanzi je celkem na všech účtech 0, jindy víc, tj. amortizovaná cena operace je $3 = \Theta(1)$.

Binomiální a Fibonacciho haldy

Motivace

Složitost Dijkstrova algoritmu závisí na operacích EXTRACT-MIN a DECREASE-KEY nad datovou strukturou, v níž jsou uchovávany vrcholy podle vzdálenosti od zdroje (celkově se provádí n-krát EXTRACT-MIN a m-krát DECREASE-KEY). Chceme dosáhnout složitost $O(n \log n + m)$, tedy konstantní DECREASE-KEY a logaritmický EXTRACT-MIN.

Definice *(Binomiální halda)*

Binomiální strom B_i sestává z kořene a jeho dětí $B_0, B_1, \ldots, B_{i-1}$ (rekurzivně), navíc se zachovává vlastnost haldy. *Řád uzlu* je počet jeho synů, *řád stromu* řád jeho kořene.

Binomiální halda je soubor binomiálních stromů různých řádů, doplněný o ukazatel MIN na strom s nejmenším kořenem. Implementovatelné např. v poli, kde i-tý prvek je buď ukazatel na strom řádu i nebo NIL.

Operace na binomiální haldě („pilná“ implementace):

- MERGE – slije dvě haldy podobně jako v binárním sčítání, jsou-li v obou haldách stromy řádu i, slepí je do jednoho stromu řádu $i+1$ a přenáší dál. MIN nové haldy nastaví na menší z obou MIN. Složitost $O(\log n)$
- EXTRACT-MIN – utrhne kořen, jeho potomky prohlásí za další haldu a zmergeuje je s ostatními stromy. $O(\log n)$.
- MAKEHEAP – z jednoho prvku udělá haldu. $O(1)$.
- INSERT – kombinace MAKEHEAP a MERGE. Amortizovaně $O(1)$ – analogicky jako binární přičítání jedničky.

Operace v „líné“ implementaci: povolím (dočasnou) existenci více stromů stejného řádu, navíc místo pole stromů použiji kruhový spoják.

- MERGE – je proste napojení spojáků za sebe. $O(1)$.
- INSERT – to samé, ale volá líný MERGE, takže je $O(1)$.
- EXTRACT-MIN – oddělí minimum, do spojáku zavede všechny syny minima, to je $O(\log n)$. Pak provede KONSOLIDACE – upraví haldu do podoby, kdy je každý řád zastoupen nejvýše jedním stromem. To běží taky v $O(\log n)$:
  Stromy, které prodělají konsolidaci, rozdělím na dvě skupiny:
  - Ty, co zaniknou (jsou slité pod jiným uzlem) zaplatí z účtu svého kořene: za odstranění ze spojáku a přepojení do stromu jde o konst. práci.
  - Za ty, co nezaniknou (jejich kořeny zůstávají kořeny i po konsolidaci) zaplatí EXTRACT-MIN, což jde, protože jich je logaritmicky.

Definice *(Fibonacciho halda)*

Je stejná jako Binomiální s línou implementací, navíc ale povolím i jiné než binomiální stromy; řád stromu je pořád definován stejně, slévám stále pouze stromy stejného řádu. Stromy mají stále exponenciálně uzlů vzhledem k řádu (důkaz viz níže).

- MERGE, INSERT, EXTRACT-MIN – implementovány stejně jako u binomiálních, mám navíc:
- DECREASE-KEY – sníží klíč i o δ, pak podstrom s i jako kořenem odřízne a zavede do spojáku jako samostatný strom. Od každého vrcholu x můžou být odříznuti max. dva synové, tj. pokud odřezávám druhého, odříznu hned po něm i x. Pracuje amortizovaně v $O(1)$, i když může jedna operace trvat až $O(\log n)$.
  K prvkům se musím dostat v konstantním čase. Každé odříznutí platí 4 jednotky – jednu za vlastní práci, jednu na účet nového kořene a dvě na účet otce – když jsou otci odříznuti 2 synové, má na zaplacení svého odříznutí.

Lemma *(Minimální řád vrcholu ve Fibonacciho haldě)*

Nechť x je nějaký vrchol a $y_1, \ldots, y_m$ jsou jeho synové v pořadí podle slití. Pak $\forall i \in \{1, \ldots, N\}$ je řád y_i aspoň $i-2$.

Důkaz
V okamiku, kdy y_i byl „slit“ pod x, byl řád x aspoň $i-1$ (protože všichni synové s menším indexem byli na x už přidáni) a řád y_i taktéž (slévám jen stromy stejného řádu), navíc y_i mohl být odříznut jen 1 syn, jinak by byl odříznutý sám.

Věta *(O počtu vrcholů ve Fibonacciho stromech)*
Strom řádu i ve Fibonacciho haldě má velikost alespoň φ^i pro nějaké $\varphi > 1$.
Důkaz
F_i nechť je nejmenší možný Fibonacciho strom řádu i, splňující omezení na odřezávání, počet jeho vrcholů buď f_i (odpovídá Fibonacciho posloupnosti). F_i musí vzniknout „slitím“ F_{i-1} a F_{i-2} (z omezení na řád vrcholu v lemmatu). To skutečně jde dostat posloupností operací – mám 2x F_i (z indukce $F_{i-2} \cup F_{i-3}$) a F_0 (dummy s nejmenším klíčem), zahodím F_0 v extract-min a výsledná konsolidace mi dá F_i ze dvou F_{i-1}. Následně provedu DECREASE-KEY na kořen jednoho z F_{i-3} a tím mám F_i jako spojení F_{i-1} a F_{i-2}. Dostávám tedy rekurentní vzorec $f_i = f_{i-1} + f_{i-2}$ a jako u Fibonacciho čísel vyčíslím $\varphi = \frac{1+\sqrt{5}}{2}$.

7 Třídy P a NP, polynomiální převody, NP-úplnost

Definice *(Úloha)*

- *Úloha* je situace, kdy pro daný vstup (*instanci úlohy*) chceme získat výstup se zadanými vlastnostmi.
- *Optimalizační úloha* je úloha, kde cílem je získat optimální (zpravidla největší nebo nejmenší) výstup s danými vlastnostmi.
- *Rozhodovací problém* je úloha, jejímž výstupem je ANO/NE.

Definice *(Kódování vstupů)*
Každá instance problému Q je kódována jako posloupnost 0 a 1, tj. instance je slovo v abecedě $\{0,1\}^*$. Kódy všech instancí problému Q tvoří jazyk $L(Q)$ nad abecedou $\{0,1\}^*$, který se dělí na

- $L(Q)_Y$ – kódy instancí s odpovědí ANO (jazyk kladných instancí)
- $L(Q)_N$ – kódy instancí s odpovědí NE (jazyk záporných instancí)

Rozhodovací problém pak je rozhodnutí, zda $x \in L(Q)_Y$ nebo $x \in L(Q)_N$ (kde x je kód nějaké instance Q), když předpokládáme, že rozhodnutí $x \in L(Q)$ lze udělat v polynomiálním čase vzhledem k $|x|$.

Definice *(Deterministický Turingův stroj)*
DTS obsahuje řídící jednotku, čtecí/zápisovou hlavu a (nekonečnou) pásku. Program sestává z:

1. Konečné množiny Γ páskových symbolů, $\Sigma \subset \Gamma$ vstupních symbolů a $* \in \Gamma$ prázdného symbolu
2. Konečné množiny Q stavů řídící jednotky, která obsahuje startovní stav q_0 a 2 terminální stavy q_Y, q_N
3. Přechodové funkce $\delta : (Q \backslash \{q_Y, q_N\}) \times \Gamma \to Q \times \Gamma \times \{\leftarrow, \rightarrow\}$

DTS s programem M *přijímá* $x \in \Sigma^*$, právě když pro vstup x se M zastaví ve stavu q_Y. *Jazyk rozpoznávaný programem* M je $L(M) = \{x \in \Sigma^* | M \text{ přijímá } x\}$.

DTS s programem M *řeší* problém Q, právě když výpočet M skončí pro každý vstup $x \in \Sigma^*$ a platí $L(M) = L(Q)_Y$.

Nechť M je program pro DTS, který skončí pro $\forall x \in \Sigma^*$. Časová složitost programu M je dána funkcí $T_M(n) = \max\{m | \exists x \in \Sigma^*, |x| = n,$ výpočet na DTS s programem M a vstupem x skončí po m krocích stroje$\}$. Pokud existuje polynom p tak, že $T_M(n) \leq p(n)\ \forall n$, pak M je *polynomiální DTS program.*

Definice *(Třída P)*
Problém Q je ve třídě P, právě když existuje polynomiální DTS program M, který řeší Q.

Definice *(Nedeterministický Turingův stroj)*
Stejný jako DTS, ale místo přechodové funkce δ je zde zobrazení δ, které každé dvojici z $Q \times \Gamma$ přiřazuje množinu možných pokračování výpočtu, tj. trojic z $Q \times \Gamma \times \{\leftarrow, \rightarrow\}$.

NTS s programem M *přijímá* $x \in \Sigma^*$, právě když existuje přijímající výpočet programu M (tj. běh M, kdy na vstupu je x a končí se ve stavu q_Y). *Jazyk rozpoznávaný programem* M je $L(M) = \{x \in \Sigma^* | M \text{ přijímá } x\}$.

Čas, ve kterém M přijímá $x \in \Sigma^*$ definujeme jako počet kroků nejkratšího přijímajícího výpočtu nad daty x.

Časová složitost programu je dána funkcí:

$$T_M(n) = \begin{cases} 1 \text{ neexistuje } x \text{ délky } n, \text{ které je přijímáno} \\ \max\{m | \exists x \in \Sigma^*, |x| = n, M \text{ přijímá } x \text{ v čase } m\} \end{cases}$$

Pokud existuje polynom p takový, že $T_M(n) \leq p(n)$, pak M je polynomiální NTS program.

Definice *(Třída NP)*
Problém Q je ve třídě P, právě když existuje polynomiální NTS program M, který řeší Q. Na rozdíl od deterministického případu netrváme na tom, že výpočet musí skončit i pro nepřijímané instance.

Poznámka *(Jiný model NTS)*
Přidáme další pásku (orákulum) a stroj pracuje ve 2 fázích:

1. Nedeterministicky hádá – zapíše problém do orákula.
2. Deterministicky ověřuje obsah orákula – práce DTS na původním vstupu plus obsahu orákula.

Je to ekvivalentní s původním – omezíme-li počet možných přechodů NTS na 2 (tím ho jen zpomalíme) a zapisujeme-li do orákula větve pokračování výpočtu (pak stačí na jednu jeden bit), převedeme veškerý nedeterminismus čistě na naplnění orákula.

Definice *(Třída co-NP)*
Problém Q je ve třídě co-NP, právě když existuje polynomiální NTS program M takový, že $L(M) = L(Q)_N$. O poměru množin co-NP a NP nevíme nic, jen to, že podmnožinou jejich průniku je P.

Převody a NP-úplnost

Definice *(Polynomiálně vyčíslitelná funkce)*
Funkce $f : \{0,1\}^* \rightarrow \{0,1\}^*$ je *polynomiálně vyčíslitelná*, právě když existuje polynom p a algoritmus A takový, že pro každý výstup $\{0,1\}^*$ dává výstup $f(x)$ v čase nejvýše $p(|x|)$.

Definice *(Polynomiální převoditelnost)*
Jazyk L_1 je *polynomiálně převoditelný* na jazyk L_2 (píšeme $L_1 \propto L_2$), právě když existuje polynomiálně vyčíslitelná funkce f taková, že

$$\forall x \in \{0,1\}^* : x \in L_1 \equiv f(x) \in L_2$$

Definice *(NP-těžký, NP-úplný problém)*

- Problém Q je *NP-těžký*, právě když $\forall Q' \in \text{NP} : L(Q')_Y \propto L(Q)_Y$.
- Problém Q je *NP-úplný*, právě když je Q NP-těžký a $Q \in \text{NP}$.

Je-li nějaký NP-těžký problém převoditelný na jiný, pak ten musí být také NP-těžký.

8 Příklady NP-úplných problémů a převody mezi nimi

TODO!

9 Silná NP-úplnost, pseudopolynomiální algoritmy

Příklad

SP není exponenciální, ale polynomiální v počtu a velikosti čísel. Algoritmus:

1. Nechť $a_1 \leq a_2 \leq \cdots \leq a_n$ a A je bitové pole délky b (kde 1 na pozici i bude indikovat možnost vytvoření podmnožiny se součtem i).
2. Všechny prvky pole A nastav na 0.
3. Pro i od 1 do n opakuj (hl. cyklus):
   - $A[a_i] := 1$
   - Pro j od a_{i-1} do b zkoušej: když $A[j] = 1$ a $j + a_i \leq b$, nastav $A[j + a_i] := 1$
4. Je-li $A[b] = 1$, podmnožina se součtem rovným b existuje.

Po i-tém průchodu hlavním cyklem obsahuje A jedničky právě u všech součtů neprázdných podmnožin $\{a_1, \ldots, a_i\}$. Důkaz – indukcí. Celk. složitost je $O(n \cdot b)$.

Definice *(Pseudopolynomiální algoritmus)*

Nechť je dán rozhodovací problém Π a jeho instance I. Pak definujeme:

- *kód(I)* – délka zápisu (počet bitů) instance I v binárním kódování (či jiném na něj polynomiálně převoditelném)
- *max(I)* – velikost největšího čísla, vyskytujícího se v I (NE délka jeho binárního zápisu!)

Algoritmus se nazývá *pseudopolynomiální*, pokud je jeho časová složitost omezena polynomem v proměnných $kód(I)$ a $max(I)$. Každý polynomiální algoritmus je tím pádem pseudopolynomiální.

Poznámka *(O číselných problémech)*

Pokud pro nějaký problém Π platí, že $\forall I : max(I) \leq p(kód(I))$ pro nějaký polynom p, pak všechny pseudopolynomiální algoritmy, řešící tento problém, jsou zároveň polynomiální.

Všechny problémy, kde tato rovnice neplatí(tj. neexistuje p, že by platila), nazýváme *číselné problémy.*

Věta *(O pseudopolynomialitě a NP)*

Nechť Π je NP-úplný problém a není číselný. Pak pokud P≠NP, nemůže být Π řešen pseudopolynomiálním algoritmem.

Poznámka

Ani ne každý číselný problém je řešitelný pseudopolynomiálním algoritmem.

Věta *(O pseudopolynomialitě a podproblémech)*

Nechť Π je rozhodovací problém a p polynom. Potom Π_p označme množinu instancí (podproblém) problému Π, pro které platí $max(I) \leq p(kód(I))$. Potom máme-li pseudopolynomiální algoritmus A, který řeší problém Π, určitě existuje polynomiální algoritmus, řešící Π_p. Toto platí pro libovolné p.

Důkaz

Algoritmus A', řešící Π_p v polynomiálním čase, otestuje x na přítomnost v Π_p (spočítá $kód(x)$ a $max(x)$) a pokud $x \in \Pi_p$, chová se stejně jako A, takže běží v čase $q(kód(x), max(x)) \leq q(kód(x), p(kód(x))) = q'(kód(x))$.

Definice *(Silně NP-úplný problém)*

Rozhodovací problém Π je *silně NP-úplný*, pokud $\Pi \in$NP a existuje polynom p takový, že podproblém Π_p je NP-úplný.

Věta *(O silné NP-úplnosti)*

Nechť problém Π je silně NP-úplný. Potom, pokud P≠NP, neexistuje pseudopolynomiální algoritmus, který by řešil Π.

Důkaz
Plyne z předchozí věty.

Příklady

- TSP je silně NP-úplný. Je to číselný problém, protože váhy hran nejsou omezené. Když váhy na hranách omezím, dostanu NP-úplný podproblém (jde na něj převést HK).
- 3-ROZDĚLENÍ je silně NP-úplné. Problém: máme $a_1, \dots a_{3m}, b \in \mathbb{N}$ takové, že $\forall j : \frac{1}{4}b \leq a_j \leq \frac{1}{2}b$ a navíc $\sum_{j=1}^{3m} a_j = mb$. Existuje $S_1, \dots S_m$ disjunktní rozdělení množiny $\{1, \dots, 3m\}$ takové, že $\forall i : \sum_{j \in S_i} a_j = b$?
  Důkaz se provádí převodem z 3DM (třídimenzionální párování na tripartitních grafech), všechna čísla konstruovaná pro převod jsou polynomiálně velká vzhledem ke $|G|$ (v převodu $VP \propto SP$ byla exponenciálně velká).

10 Třída #P, #P-úplné úlohy

Definice *(Početní úloha)*
Narozdíl od rozhodovacího problému, kde ptáme na existenci řešení, se v *početní úloze* ptáme na počet všech řešení.

Definice *(Certifikační funkce, certifikát)*
Funkce $w : \Sigma^* \to \mathcal{P}(\Gamma^*)$ (kde $\mathcal{P}$ znamená potenční množinu) se nazývá *certifikační funkce* pro Σ (vstupní abeceda) a Γ (abeceda certifikátů). Každý prvek množiny $w(x)$ je *certifikát* pro instanci x.

Rozhodovací problém A_w spojený s certif. funkcí w je definován předpisem $A_w = \{x \in \Sigma^* | w(x) \neq \emptyset\}$, tj. A_w je množina kódů těch instancí, pro které existuje nějaký certifikát.

Definice *(Třída #P)*
Třída #P je množina certif. funkcí w takových, že

- existuje polynomiální algoritmus (v $|x| + |y|$), který pro libovolné dané $x \in \Sigma^*$ a $y \in \Gamma^*$ ověří, zda $y \in w(x)$,
- existuje polynom p takový, že $\forall y \in w(x) : |y| \leq p(|x|)$ (a p může záviset na w).

Tedy ověření je polynomiální, ne výpočet certifikátu.

Věta *(O vztahu #P a NP)*

1. $w \in \#\mathrm{P} \Rightarrow\ A_w \in \mathrm{NP}$
2. $A \in \mathrm{NP} \Rightarrow\ \exists w \in \#\mathrm{P}\text{: } A = A_w$

Důkaz

1. Chci ukázat, že k A_w existuje NTS, rozpoznávající jazyk v polynomiálním čase. Ten nedeterministicky vygeneruje certifikát a ověří ho, a díky tomu, že $w \in \#\mathrm{P}$, je ověření polynomiální, tj. $A_w \in \mathrm{NP}$.
2. Mám NTS program M pro A, definuji pro $x \in \Sigma^*$ množinu

$$w(x) := \{y \in \Gamma^* | y \text{ kóduje přijímající výpočet } M \text{ nad vstupem délky } \leq p(x)\}$$

Pak určitě $A = A_w$. Ověřím podmínky #P: příslušný ověřovací algoritmus otestuje polynomiální velikost vygenerovaného certifikátu y a pak simuluje práci M, y je polynomiální z definice $w(x)$.

Věta *(O početních a rozhodovacích úlohách)*
Pokud $w \in \#\mathrm{P}$ je taková, že $\forall x \in \Sigma^*$ lze polynomiálním čase spočítat $|w(x)|$, tak lze v polynomiálním čase rozhodnout, zda $x \in A_w$ (tj. vyřešit příslušný problém třídy NP) i zda $x \in \overline{A_w}$ (tj. vyřešit přísl. problém z třídy co-NP).

Díky tomu jsou početní úlohy, odpovídající NP-úplným problémům (jako #SAT, #KLIKA, #SP, ...) alespoň tak těžké jako odpovídající rozhodovací problémy (SAT, KLIKA, SP, ...).

Definice *(Polynomiální redukce)*

Nechť $w : \Sigma^* \to \mathcal{P}(\Gamma^*)$ a $v : \Pi^* \to \mathcal{P}(\Delta^*)$ jsou certifikační funkce. *Polynomiální redukce* z w na v je dvojice funkcí, vyčíslitelných v polynomiálním čase $\sigma : \Sigma^* \to \Pi^*$ a $\tau : \mathbb{N} \to \mathbb{N}$ takových, že $\forall x \in \Sigma^* : |w(x)| = \tau(|w(\sigma(x))|)$.

Pokud je navíc τ identita, redukce se nazývá *šetrná*.

Definice *(#P-úplná funkce)*

Certifikační funkce w je #P-úplná, pokud $w \in$ #P a zároveň $\forall v \in$ #P existuje polynomiální redukce z v na w.

Věta *(O #KACHL)*

#KACHL je #P-úplný.

Důkaz

Mám dány definice barev, síť a typy kachlíků, úkolem je spočítat, kolik různých legálních vykachlíkování existuje.

Nechť $w \in$ #P, potom existuje program M pro NTS, který pro zadání x nedeterministicky vygeneruje certifikát y a pak ověří, zda $y \in w(x)$. Přijímající výpočty M odpovídají $1:1$ počtu certifikátů y. Podle důkazu Cooke-Lewinovy věty pro každou dvojici (program, přijímající výpočet) existuje právě jedno legální vykachlíkování (pro dva různé výpočty jsou dvě různá kachlíkování).

Díky tomu existuje polynomiální šetrná redukce $\sigma : \Sigma^* \to L(\text{KACHL})$ taková, že A_w se zobrazí na $L(\text{KACHL})_Y$.

Věta *(O #P-úplných úlohách)*

#SAT, #3-SAT, #KLIKA a #SP jsou #P-úplné

Důkaz

Pro #SAT přímo transformace KACHL $\to$ SAT definuje šetrnou redukci. Pro ostatní se redukce dají získat mírnými úpravami převodů rozhodovacích úloh.

Příklad

Úloha, která jako rozhodovací problém (rozhodnout $x \in A_w$) není NP-úplná, ale její početní úloha ($|w(x)|$) je #P-úplná:

Nechť $G = (U \cup V, E)$ je bipartitní graf, kódovaný řetězcem x a $y \in w(x)$, pokud y kóduje perfektní párování v G. Potom $A_w = \{G|$ v G existuje perf. párování$\}$ a platí:

1. Rozhodnout, zda $x \in A_w$ lze v polynomiálním čase
2. Spočítat $|w(x)|$ je #P-úplná úloha

Příklad

Spočítat permanent matice s prvky 0 a 1 je také #P-úplná úloha. Permanent matice $A = (a_{ij})$ typu $n \times n$ je definován jako $\text{Perm}(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{i\sigma(i)}$.

Důkaz

Matici lze považovat za matici sousednosti grafu, v níž hledám perfektní párování (řádky: jedna partita, sloupce: druhá, permanent je počet perf. párování).

11 Aproximace NP-úplných problémů

Aproximační algoritmus

Definice *(Aproximační algoritmus)*

Aproximační algoritmus běží v polynomiálním čase a vrací řešení „blízká“ optimu. Je nutné mít nějakou míru kvality řešení. Označme:

- C^* hodnotu optimálního řešení
- C hodnotu nalezenou aproximačním algoritmem

A předpokládejme nezáporné hodnoty řešení.

Definice *(Poměrová chyba)*

Řekneme, že algoritmus řeší problém s *poměrovou chybou* $\rho(n)$, pokud pro každé zadání velikosti n platí:

$$\max\{\frac{C^*}{C}, \frac{C}{C^*}\} \leq \rho(n)$$

Definice *(Relativní chyba)*

Řekneme, že algoritmus řeší problém s *relativní chybou* $\varepsilon(n)$, pokud pro každé zadání velikosti n platí:

$$\frac{|C - C^*|}{C^*} \leq \varepsilon(n)$$

Poznámka *(O převodu chyb)*

Z jedné chyby se dá vyjádřit druhá:

- V případě maximalizační úlohy: $\varepsilon(n) = \frac{C-C^*}{C^*} = \frac{C}{C^*} - 1 = \rho(n) - 1$
- V případě minimalizační úlohy: $\varepsilon(n) = \frac{C^*-C}{C^*} == 1 - \frac{1}{\rho(n)}$

Příklad

Aproximační algoritmy pro vrcholové pokrytí:

- „Brát vrcholy od nejvyššího stupně, dokud nemám celé pokrytí“ nemá konstantní relativní chybu – ex. protipříklad, kdy $\rho(k) \leq a \cdot \ln k$
- „Vzít libovolnou hranu, dát do pokrytí její dva konce, odstranit její incidentní hrany a projít tak celé E“ má relativní chybu 2 – žádné 2 hrany nemají společný vrchol, tj. mám pokrytí o velikosti $2 \times$ |mn. disj. hran|. Každé vrcholové pokrytí je ale $\geq$ |mn. disj. hran|.

Příklad

Aproximační algoritmy pro TSP

- *Omezení na trojúhelníkovou nerovnost* – pořád je NP (převod HK→TSP zachovával trojúhelníkovou nerovnost). Algoritmus:
  1. Najdi minimální kostru T
  2. Zvol lib. vrchol a pomocí DFS nad T (slajdy: chyba) v PRE-ORDERu očísluj vrcholy.
  3. Cesta (s opakováním) po kostře T přes všechny vrcholy := X. Pak $w(X) = 2w(T)$ (každou hranou kostry jdu tam a zpět).
  4. Výslednou HK H vyrobím zkrácením cesty (vypouštěním již navštívených vrcholů), z trojúhelníkové nerovnosti $w(H) \leq w(X)$.

  Celkem tedy dává $w(H) \leq w(X) \leq 2w(H^*)$, protože $w(T) \leq w(H^*)$ (H^* je kostra, bez jedné hrany).
- *Bez omezení* – pro žádné konstantní ρ neexistuje polynomiální algoritmus, řešící obecný TSP s poměrovou chybou ρ. Mohu totiž mít HK v grafu $G = (V, E)$, pak zadání TSP zkonstruovat jako $K_{|V|}(V, V \times V)$, kde

$$w(e) = \begin{cases} 1 & e \in E \\ n\rho & e \notin E \end{cases}$$

  Pak by aproximační algoritmus s chybou ρ musel určitě vždy vrátit přesné řešení, takže by musel být NP-těžký.

Aproximační schéma

Definice *(Aproximační schéma)*

Aproximační schéma pro optimalizační úlohu je aproximační algoritmus, který má jako vstup instanci dané úlohy a číslo $\varepsilon \geq 0$, a který pro libovolné ε pracuje jako aproximační algoritmus s relativní chybou ε. Doba běhu může být exponenciální jak vzhledem n – velikosti vstupní instance, tak vzhledem k $\frac{1}{\varepsilon}$.

Definice *(Polynomiální aproximační schéma)*

Polynomiální aproximační schéma je takové aproximační schéma, které pro každé pevné $\varepsilon \geq 0$ běží v polynomiálním čase vzhledem k n (ale stále může být exponenciální vzhledem k $\frac{1}{\varepsilon}$.

Definice *(Úplně polynomiální aproximační schéma)*

Úplně polynomiální aproximační schéma je polynomiální aprox. schéma, běžící také v polynomiálním čase vzhledem k $\frac{1}{\varepsilon}$ (tj. algoritmus s konstantně-krát menší relativní chybou běží v konstantně-krát delším čase).

TODO: APPROX-SP