C2142 Návrh algoritmů pro přírodovědce

2. Úvod do složitosti

Tomáš Raček

Jaro 2017

Hledání nejčastějších slov – frequent_words

Zadání. Nalezněte v textu řetězce délky k s nejvyšším počtem výskytů.

frequent_words(text, k)

1. Pro každý podřetězec délky k řetězce text spočítej jeho výskyt pomocí funkce pattern_count(text, pattern)
2. Urči nejvyšší nalezenou četnost
3. Vrať řetězce s touto nejvyšší četností

Praktický test.

- krátké řetězce – frequent_words uspokojivě funguje
- dlouhé řetězce – nepoužitelné, čas výpočtu neodpovídá odhadu

frequent_words – doba výpočtu

Pozorování

- doba výpočtu je úměrná velikosti vstupních dat
- závislost není nutně lineární – výpočet na 1000krát větší úloze nemusí trvat 1000krát déle
- na různých strojích/architekturách různé časy výpočtu
    - Thinkpad T430s: 7 s
    - Thinkpad X200s: 14 s

Důsledek (1). Nutná hlubší analýza frequent_words.

Důsledek (2). Porovnání náročnosti algoritmů podle času výpočtu není vhodné, potřebujeme aparát nezávislý na konkrétním stroji/architektuře.

Složitost

Složitost algoritmu. Zaveďme složitost algoritmu jako funkci $f(n)$, kde n je velikost vstupu.

Návrh. $f(n)$ určuje počet jednoduchých operací daného algoritmu pro vyřešení problému o velikosti n.

- jednoduché operace $\approx$ instrukce CPU (např. sečtení nebo porovnání dvou čísel, AND/OR,...)
- řešení nezávislé na architektuře

Důsledek. Porovnání efektivity algoritmů lze zjednodušeně převést na porovnání jejich složitostí.

Asymptotická složitost – definice

Formální definice

$$\mathcal{O}(g) = \{f \mid \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)\}$$

$f \in \mathcal{O}(g)$ čteme „f roste asymptoticky nejvýše tak rychle jako g“.

Význam konstant

c rozdíl pouze v multiplikativní konstantě nepovažujeme za významný, tj. ztotožňujeme např. n^2 a $4n^2$

n_0 vztah nemusí platit pro prvních n_0 čísel

Poznámka. Analogicky lze definovat další množiny:

- $f \in \Omega(g)$ – f roste asymptoticky alespoň tak rychle jako g
- $f \in \Theta(g)$ – f roste asymptoticky právě tak rychle jako g

Asymptotická složitost – příklady

Rychlost růstu funkcí

$$\log n \ll n \ll n \log n \ll n^2 \ll 2^n \ll n! \qquad \text{pro } n \to \infty$$

Příklady

Funkce	Složitostní třída	Pojmenování
2142	$\mathcal{O}(1)$	konstantní
$2 \log n + 4$	$\mathcal{O}(\log n)$	logaritmická
$0.5n + \log n$	$\mathcal{O}(n)$	lineární
$n^2 - 10n$	$\mathcal{O}(n^2)$	kvadratická
$6n^3$	$\mathcal{O}(n^3)$	kubická
$2^n - 1$	$\mathcal{O}(2^n)$	exponenciální

Poznámka. Ověření, zdali $f \in \mathcal{O}(g)$, lze provést výpočtem limity $\lim_{n\to\infty} f(n)/g(n)$.

Složitost problému

Cíl. Snaha o nalezení efektivních algoritmů pro daný problém.

Otázka. Lze zrychlovat pořád, nebo existuje nějaký dolní limit?

Složitost problému

- minimální počet operací potřebný pro vyřešení libovolné instance problému
- nutno odvodit teoreticky $\rightarrow$ mnohdy netriviální
- odpovídá složitosti optimálního algoritmu pro daný problém

Jak ale poznám optimální algoritmus? Srovnejme odhady složitosti problému $\mathcal{P}_i$ a složitosti algoritmů $\mathcal{A}_j$ řešící tento problém:

$$\mathcal{P}_1(n) < \ldots < \mathcal{P}_k(n) \leq \mathcal{A}_1(n) < \ldots < \mathcal{A}_m(n)$$

$\mathcal{A}$ je optimální algoritmus, pokud $\mathcal{A}(n) = \mathcal{P}_k(n)$.

Složitost problému – příklady

Nalezení nejmenšího prvku pole

- je nutné projít všechny prvky pole – $\Omega(n)$ operací
- algoritmus se složitostí $\mathcal{O}(n)$ jistě existuje $\rightarrow$ složitost problému (= složitost optimálního algoritmu) je lineární

Násobení matic

- potřeba $\Omega(n^2)$ operací
- naivní algoritmus – $\mathcal{O}(n^3)$
- Strassenův algoritmus – $\mathcal{O}(n^{\log_2 7}) \doteq \mathcal{O}(n^{2{,}81})$
- aktuálně nejlepší algoritmus (2014) – $\mathcal{O}(n^{2{,}372\ldots})$
- nalezení optimálního algoritmu je otevřený problém

Prostorová složitost

Prostorová složitost. Vedle časové náročnosti algoritmů lze určit i množství paměti, které algoritmus potřebuje pro svůj výpočet.

- velikost vstupních (a výstupních) dat neuvažujeme
- vyjadřujeme také $\mathcal{O}$-notací

In situ algoritmus vyžaduje navíc pouze $\mathcal{O}(1)$ paměti.

- výpočet průměrné hodnoty prvků v poli
- naivní násobení matic
- ...

Otázka. Je lepší in situ algoritmus s časovou složitostí $\mathcal{O}(n^2)$ než algoritmus s časovou složitostí $\mathcal{O}(n \log n)$ a prostorovou složitostí $\mathcal{O}(n)$?

Vztah mezi časem a prostorem

Teze. Někdy lze snížit časovou složitost algoritmu zvýšením jeho prostorové složitosti (a naopak).

↑ prostor ↓ čas

- softwarová cache
- předpočítání (mezi)výsledků

↑ čas ↓ prostor

- komprese
- zvýšení abstrakce

Složitost v praxi

Tabulka časů výpočtu algoritmů o složitostech $\log n$, n, n^2, 2^n a pro vstup velikosti 10, 20, 50 a 1000. Předpokládejme, že jedna iterace algoritmu trvá $1\mu s$.

	10	20	50	1000
$\log n$	0,000001 s	0,000001 s	0,000002 s	0,000003 s
n	0,00001 s	0,00002 s	0,00005 s	0,001 s
n^2	0,0001 s	0,0004 s	0,0025 s	1 s
2^n	0,001024 s	1,048576 s	35,7 let	$3,4 \cdot 10^{287}$ let

Poznámka. Stáří vesmíru je odhadováno na $13,8 \cdot 10^9$ let.

pattern_count – analýza

```
def pattern_count(text, pattern):
    count = 0
    for i in range(0, 1  + len(text) - len(pattern)):
        if text[i : i + len(pattern)] == pattern:
            count += 1

    return count
```

Pozorování

- procházíme celkem $|text| - |pattern| + 1$ možných umístění
- každé porovnání dvou řetězců obnáší nejvýše $|pattern|$ porovnání jednotlivých znaků

Závěr. Počet kroků, které vykoná funkce pattern_count, lze vyjádřit jako $\mathcal{O}(|pattern| \cdot (|text| - |pattern| + 1))$.

frequent_words – analýza I

```
def frequent_words(text, k):
    counts = dict()
    frequent_patterns = set()

    for i in range(0, len(text) - k + 1):
        pattern = text[i : i + k]
        counts[pattern] = pattern_count(text, pattern)

    max_count = max(counts.values())
    for (pattern, count) in counts.items():
        if count == max_count:
            frequent_patterns.add(pattern)

    return frequent_patterns
```

Pozorování

- počet volání pattern_count je $|text| - k + 1$
- další příkazy nejsou určující pro dobu běhu

frequent_words – analýza II

Složením předchozích informací dostáváme:

frequent_words(text, k)

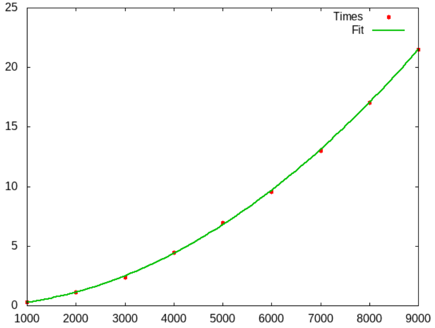
- složitost funkce pattern_count je $\mathcal{O}(|pattern| \cdot (|text| - |pattern| + 1))$
- počet volání pattern_count je $|text| - k + 1$
- platí $k = |pattern|$
- počet kroků celkem:

$$k \cdot (|text| - k + 1) \cdot (|text| - k + 1) = k \cdot (|text| - k + 1)^2$$

V praxi platí $k \ll |text|$, asymptotická složitost funkce frequent_words je tedy $\mathcal{O}(k \cdot |text|^2)$.

frequent_words – praxe

Měření. Doba výpočtu funkce frequent_words(text, k) pro $k = 9$ a $|text| = \{1000, \dots, 9000\}$.



Pozorování. Naměřená data lze úspěšně proložit parabolou, což odpovídá odhadnuté složitosti $\mathcal{O}(k \cdot |text|^2)$.

Hledání nejčastějších slov v textu

Dosavadní řešení. Jsme schopni navrhnout a implementovat algoritmus se složitostí $\mathcal{O}(k \cdot |text|^2)$.

Zásadní otázka. Jde to i lépe?

Alternativní návrh. Počítání četností podřetězců při průchodu textem

1. Procházej vstupní text postupně po podřetězcích délky k
   1.1 Pokud se konkrétní podřetězec vyskytl poprvé, nastav jeho četnost na 1, jinak ji zvyš o 1
2. Urči nejvyšší nalezenou četnost
3. Vrať řetězce s touto nejvyšší četností

faster_frequent_words

Jak ukládat pro každý řetězec jeho četnost?

- počet různých řetězců délky k z písmen A, C, G, T je 4^k
- každému tomuto řetězci lze přiřadit číslo od 0 do $4^k - 1$

Příklad pro $k = 3$:

$$\text{AAA} \to 0, \text{AAC} \to 1, \text{AAG} \to 2, \ldots, \text{TTT} \to 63$$

Příklad převodu řetězce na číslo. A $\to 0$, C $\to 1$, G $\to 2$, T $\to 3$.

$$\begin{aligned} \text{ACCTG} &\to A \cdot 4^4 + C \cdot 4^3 + C \cdot 4^2 + T \cdot 4^1 + G \cdot 4^0 \\ \text{ACCTG} &\to 0 + 64 + 16 + 12 + 2 \\ \text{ACCTG} &\to 94 \end{aligned}$$

Implementace. Vytvořím pole o velikosti 4^k, kde budu ukládat četnosti jednotlivých řetězců.

Převod řetězce na číslo – implementace

```
def pattern2number(pattern):
    characters = "ACGT"

    if pattern == "":
        return 0
    else:
        return 4 * pattern2number(pattern[:-1]) \
                + characters.index(pattern[-1:])


def number2pattern(number, k):
    characters = "ACGT"

    if k == 0:
        return ""
    else:
        divisor = 4 ** (k - 1)
        return characters[number // divisor] \
                + number2pattern(number % divisor, k - 1)
```

faster_frequent_words – implementace

```python
def computing_frequencies(text, k):
    frequency_array = [0] * (4 ** k)

    for i in range(len(text) - k + 1):
        pattern = text[i: i + k]
        frequency_array[pattern2number(pattern)] += 1

    return frequency_array


def faster_frequent_words(text, k):
    frequent_patterns = set()
    frequency_array = computing_frequencies(text, k)
    max_count = max(frequency_array)

    for i in range(0, 4 ** k):
        if frequency_array[i] == max_count:
            frequent_patterns.add(number2pattern(i, k))

    return frequent_patterns
```

Složitost faster_frequent_words

Složitost jednotlivých fází algoritmu

- inicializace pole četností $\mathcal{O}(4^k)$
- převod řetězce na číslo (a naopak) $\mathcal{O}(k)$
- průchod vstupním textem, počítání četností $\mathcal{O}(k \cdot (|text| - k + 1))$
- nalezení nejvyšší četnosti $\mathcal{O}(4^k)$
- výběr řetězců s nejvyšší četností $\mathcal{O}(k \cdot 4^k)$

Celková složitost faster_frequent_words. Po úpravě dostáváme složitost $\mathcal{O}(k \cdot |text| + k \cdot 4^k)$, přičemž paměťová složitost je $\mathcal{O}(4^k)$.

Závěr. Pro $k \ll n$ je faster_frequent_words výrazně rychlejší než frequent_words.

A jde to ještě lépe? ;-)