ČVUT FEL – A7B36DSA 15. října 2010

Přednáška 4.

Přednášející: Mgr. Michal Píše *Transkript: Michal Frdlík*

1. Randomizované algoritmy

Je rekapitulována minulá přednáška, zejména algoritmy typu „rozděl a panuj“ a randomizované algoritmy. Randomizované algoritmy se rozdělují na dva typy:

- Las Vegas,
  který využívá pravděpodobnost k tomu, aby se rychleji dobral výsledku, který bude vždy správný (například se jedná o quicksort, kde je pivot volen náhodně, ale výsledkem je vždy setřízené pole). Výhoda tohoto algortimu je ta, že vždy dostaneme správnou odpověď. Nevýhoda je, že ji nedostaneme vždy ve stejném čase (algoritmus někdy svůj čas běhu překročí).
- Monte Carlo,
  jehož doba trvání je vždy předem známá, ale existuje nenulová (vetšinou velmi malá) pravděpodobnost, že vrácená odpověď bude nesprávná. Například máme-li soustavu prvků a chceme z ní vybrat nejčastěji opakující se prvek, vybereme náhodný prvek. Existuje vysoká pravděpodobnost, že jsme vybrali nejčastěji opakující se prvek, ale existuje i pravděpodobnost, že jsme ho nevybrali. Efektivita algoritmu typu Monte Carlo se dá často zvýšit tím, že se nechá proběhnout vícekrát a výsledky se zkombinují, čímž se snižuje pravděpodobnost chyby. Například provedeme-li v našem příkladu více náhodných výběrů, pravděpodobnost správné odpovědi se dramaticky zvyšuje (můžeme spočítat průměr všech vybraných prvků). Výhoda tohoto algoritmu je ta, že lze často zvýšit jeho efektivitu, aniž by se razantně zvýšila jeho výpočetní doba.

Las Vegas se vždy dá převést na Monte Carlo tím, že mu určíme nějaký časový limit, který když překročí, vrátíme libovolnou hodnotu.

2. Třídy P a NP, NP-úplnost

V rámci tříd složitosti O, Θ, Ω a jejich malých variant existují ještě třídy P a NP.

2.1. Rozhodovací problém

Definujeme abecedu Σ a množinu všech slov z této abecedy Σ^*. Můžeme si vybrat nějakou podmnožinu $L \subset \Sigma^*$, následně si vybereme nějaké slovo z Σ^* a ptáme se, zda je nebo není v L. Vzniká rozhodovací problém.

2.2. Redukovatelnost problémů

Problém A je redukovatelný na problém B právě tehdy, když A má nějaká řešení a tato řešení řeší B vždy, když B má nějaká řešení. Řešení problému A tedy nemůže být těžší než řešení problému B.

2.3. Polynomiální převoditelnost

Problém A je polynomiálně převoditelný na problém B právě tehdy, lze-li A redukovat na B v polynomiálním čase.

2.4. Třída P

Třída P obsahuje obsahuje všechny rozhodovací problémy, které mohou být vyřešeny v polynomiálním čase.

2.5. Třída NP

Pokud pro každý vstup x existuje predikát $Q(x, y)$, polynom p a výstup y takový, že platí:

$$|y| > p(|x|) \wedge Q(x, y) \quad \text{a} \quad (Q - \text{běží v polynomiálním čase})$$

Potřebujeme, aby pro každý vstup x z L existoval nějaký výstup y s polynomiální délkou. Úloha je NP právě tehdy, když dostaneme správné řešení a jsme ho schopni ověřit v polynomiálním čase.

2.6. Třída NP-těžkých problémů

Třída NP-těžkých problémů obsahuje všechny problémy (nemusí být nutně rozhodovací), které jsou přinejmenším tak těžké jako nejtěžší NP problémy. Jinými slovy, problém A je NP-těžký právě tehdy, když existuje NP-úplný problém, který je v polynomiálním čase převoditelný na A.

2.7. Třída NP-úplných problémů

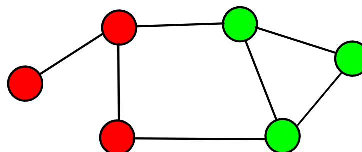
Problém je NP-úplný právě tehdy, je-li NP a zároveň NP-těžký.

2.8. Příklady známých NP problémů

Prvním příkladem je tzv. SAT (boolean SATisfability) problém, tj. vyšetřování, zda je formule v konjunktivní normální formě splnitelná (tj. existuje takové pravdivostní ohodnocení všech proměnných, že ohodnocení celé formule je pravdivé). Pro příklad formule:

$$(A \vee B \vee \neg C) \wedge (B \vee C) \wedge (\neg A \vee \neg B \vee C).$$

Dalším příkladem je hledání kliky (takového podgrafu, který je sám o sobě úplným grafem) o velikosti n v nějakém grafu.



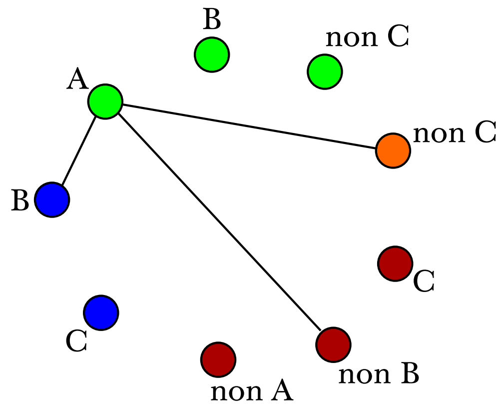
Obr. 1. – Klika o velikosti 3 (vyznačena zeleně)

U těchto problémů pravděpodobně neexistuje žádný způsob, jak ověřit správnost řešení v polynomiálním čase.

Úlohy 1 a 2 jsou navzájem polynomiálně převoditelné (tj. když se mi podaří najít polynomiální řešení pro úlohu 1, mám ho i pro úlohu 2 a naopak), což ukážeme v následující podkapitole.

2.9. Polynomiální převoditelnost SAT → hledání kliky

Abychom ukázali, že úlohy 1 a 2 jsou polynomiálně převoditelné, převeďme si naši SAT úlohy z podkapitoly 2.8. na graf podle obrázku 2.



Obr. 2. – SAT → hledání kliky

Každému atomu v každé klauzuli formule z příkladu 1 přiřadíme jeden vrchol grafu (na obrázku jsou klauzule barevně odděleny). K tomu, abychom zjistili, zda-li je formule splnitelná, musíme najít takové ohodnocení logických proměnných, že ohodnocení všech klauzulí bude 1, a tudíž i ohodnocení formule bude 1. To nastane právě tehdy, když v každé z klauzulí bude alespoň jeden atom ohodnocený 1. Spojíme vrcholy, které nejsou ve sporu a jsou v různých klausulích (podle vzoru v obrázku) a pokračujeme pro všechny vrcholy. Vznikne graf, v němž hledat kliku velikosti n je přesně stejně složité jako hledat v booleovské rovnici v konjunktivní normální formě takové ohodnocení n proměnných, že ohodnocení formule bude 1, a tudíž bude splnitelná.

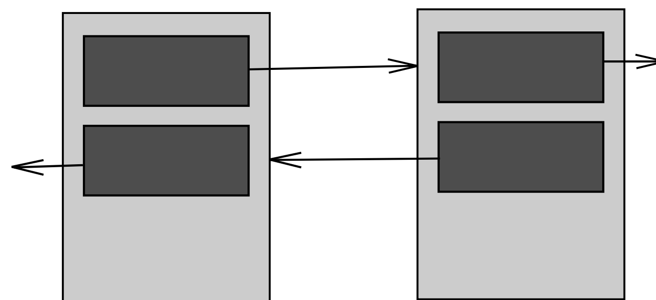
2.10 Zásadní otázka: „P = NP?“

Zásadní otázka zní: Jdou NP úlohy řešit v polynomiálním čase? Většina teoretiků předpokládá, že nejdou, ale čas od času se objeví důkaz, který říká, že jdou. Tato otázka je velice zásadní, protože pokud by tyto úlohy šly řešit v polynomiálním čase, okamžitě by přestaly fungovat například kryptografické úlohy, které jsou založeny na opaku (na tvrzení, že P ≠ NP se zakládá mnoho životně důležitých aplikací).

3. Pole a spojové seznamy

3.1. Obousměrné spojové seznamy

Spojové seznamy jsou v knize ukazovány na příkladu obousměrných spojových seznamů, které mají kromě ukazatele na následující prvek i ukazatel na předchozí prvek.


Obr. 3. – Uzly obousměrného spojového seznamu

Invariant této struktury zní: Když pro nějaký uzel `A` a `B` platí `A.next==B`, potom platí také `B.prev==A`, kde `next` je ukazatel na následující uzel a `prev` je ukazatel na předchozí prvek.

První uzel takového spojového seznamu je zpravidla prázdný (neobsahuje žádnou hodnotu). Dělá se to proto, že se to jednodušeji programuje.

3.2. Procedura `splice`

Procedura `splice` „vyřízne“ z obousměrného spojového seznamu n po sobě jdoucích uzlů, které následně připojí mezi jiné uzly. Podobnou proceduru lze implementovat i na jednosměrném spojovém seznamu. Tato procedura je důležitá, protože se pomocí ní dá nadefinovat mnoho jiných procedur.

V následující implementaci předpokládáme, že `a` a `b` paří do stejného seznamu, `a’` je prvek předcházející `a`, `b’` je prvek následující po `b`, `t` je prvek, za který má být výsek zařazen a `t’` je prvek následující po `t`. Dále předpokládáme, že `b` není v seznamu umístěno před `a`.

```
0: a’ = a.prev
1: b’ = b.next
2: a’.next = b’
3: b’.prev = a’
4: t’ = t.next
5: b.next = t’
6: a.prev = t
7: t.next = a
8: t’.prev = b
```

Výpis 1. – Procedura splice

Operace na řádcích 0–3 vyjímají úsek spojového seznamu a operace na řádcích 4–8 tento úsek vkládají za požadovaný prvek.

3.3. Pole

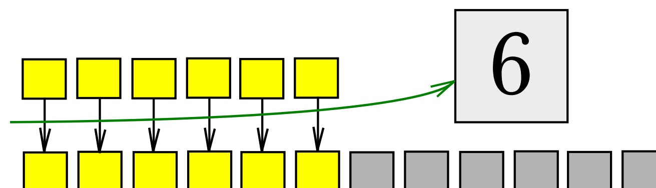
Občas potřebujeme mít pole, které dynamicky zvětšuje nebo zmenšuje svoji kapacitu. Zvětšování probíhá tak, že se původní pole překopíruje do nového zvětšeného pole. Otázka zní: Jak to, že i s těmito zdánlivě pomalými operacemi je pole i tak rychlé?

3.3.1. Metoda bankovního účtu

Uvažujme pole, které obsahuje tyto přístupové metody:

- `[n]` – přístup na n-tý prvek,
- `pushBack` – přidání prvku na konec pole,
- `popBack` – odebrání prvku z konce pole.

Lemma 3.1. potom říká, že pokud máme prázdné pole, potom výše zmíněné operace a jejich libovolné sekvence mají konstantní složitost.
Dále uvažujme, že si někde držíme „bankovní účet“, kam střádáme volné výpočetní kroky (které jsme mohli využít, ale nevyužili). Dále uvažujme, že pro každou z výše uvedených operací máme k dispozici tři kroky, přičemž každá operace trvá pouze jeden krok. Všechny nevyužité kroky ukládáme do „bankovního účtu“.



Obr. 4. – Metoda bankovního účtu

Pokud budeme chtít překročit původní délku pole, vytvoříme nové a překopírujeme do něj původní pole (přičemž tím spotřebujeme dříve naštřádané kroky) a zbyde nám počet kroků,

který bude obecně vždy větší nebo roven nule (v tomto případě větší nebo roven třem). Opakovaným použitím tohoto postupu zjistíme, že počet uložených kroků neklesá, a tedy amortizovaná složitost je 3, tedy konstantní. Že bude mít bankovní účet vždy nezápornou hodnotu dokáži v následující podkapitole.

3.3.2. Explicitní vzorec a důkaz

Chceme-li vyjádřit hodnotu bankovního účtu v jakémkoli kroku, musíme se matematicky zamyslet nad stavy, které během přidávání prvků nastávají. Začněme polem o velikosti 0. V prvním kroku se pole allokuje na $2^0 = 1$ prvků a do prvního prvku se přiřadí hodnota. Bankovní účet má v tomto okamžiku hodnotu 2, protože při allokaci nového pole se žádné předchozí prvky nekopírovaly, pouze jsme uložili 2 kroky při přidání prvku. Ve druhém kroku se nové pole allokuje na $2^1 = 2$ prvků, překopíruje se jeden prvek ze starého pole a přidá se další. Hodnota bankovního účtu bude 3, protože při kopírování jednoho prvku jsme jeden token odebrali a dva přidali. Přidáme-li třetí prvek, nové pole se allokuje na $2^2 = 4$ prvků, překopírují se dva předchozí a přidá se jeden nový. Hodnota bankovního účtu je v tomto okamžiku opět 3. Ve čtvrtém kroku allokace neprobíhá, pouze se přidává prvek a hodnota bankovního účtu vzroste na 5. Tímto způsobem lze pokračovat do nekonečna.

Povšimněme si nyní, že ve výpočtu hodnoty bankovního účtu existují dva hlavní děje. Jeden probíhá při každém kroku – přidávání 2 tokenů za každý z n prvků. S jistotou můžeme tedy říci, že v našem vzorci bude figurovat číslo $2n$. Další krok probíhá pouze jednou za určitou dobu – odebírání n tokenů, když pole překročí mocninu dvojky. Takový děj proběhne právě tehdy, bude-li aktuální krok vyjádřitelný jako $2^k + 1$ (čili čísla 3, 5, 9, 17...), protože pouze tehdy potřebujeme provést kopírování do nového pole. Jak ale ve vzorci vyjádřit fakt, že takové kopírování proběhne pouze v těchto situacích? Potřebujeme funkci, která by měnila svůj výstup pouze, kdybychom na vstupu překlenuli mocninu dvojky a která bude vypadat takto:

$$
\begin{aligned}
f(1) &= 0 \\
f(2) &= 1 \\
f(3) &= 2 \\
f(4) &= 2 \\
f(5) &= 3 \\
f(6) &= 3 \\
f(7) &= 3 \\
f(8) &= 3 \\
f(9) &= 4 \\
&\dots
\end{aligned}
$$

čili pro n bude vracet dvojkový logaritmus čísla nejbližší horní mocniny 2^n. Úvahami o vlastnostech elementárních funkcí nyní lehce dospějeme k výsledku. Hledaná funkce má tvar:

$$\lceil \log_2 n \rceil,$$

Pokud chceme výstup této funkce převést na odpovídající hraniční čísla formátu $2^k + 1$, upravíme funkci takto:

$$2^{\lceil \log_2 n \rceil} + 1.$$

To je vše, co potřebujeme k sestavení explicitního vzorce pro výpočet hodnoty bankovního účtu v n-tém kroku. Vzorec vypadá následovně (lze empiricky ověřit).

$$\mathrm{B}n = 2n - 2^{\lceil \log_2 n \rceil} + 1.$$

Zbývá ověřit, že pro všechna $n \in \mathrm{N}$ je $\mathrm{B}n$ větší nebo roven nule. Nasnadě je slabý princip matematické indukce. Provedeme základní krok pro $n = 1$:

$$\mathrm{B}n = 2 - 2^{\lceil \log_2 1 \rceil} + 1 \geq 0,$$

$$2 - 1 + 1 \geq 0,$$

$$2 \geq 0.$$

Formulujeme indukční předpoklad.

$$\forall n \in \mathrm{N} : 2n - 2^{\lceil \log_2 n \rceil} + 1 \geq 0.$$

Provedeme indukční krok.

$$2n - 2^{\lceil \log_2 n \rceil} + 1 \geq 0,$$

$$2n + 1 \geq 2^{\lceil \log_2 n \rceil},$$

$$\log_2(2n) + \log_2(1) \geq \lceil \log_2 n \rceil,$$

$$\log_2(2) + \log_2(n) + \log_2(1) \geq \lceil \log_2 n \rceil,$$

$$1 + \log_2(n) \geq \lceil \log_2 n \rceil.$$

Nyní již vidíme, že jsme dospěli k platné nerovnici, protože horní celá část se dá shora omezit přesně jako levá strana rovnice (horní celá část z čísla a bude nejhůře $a + 1$).

$$1 + \log_2(n) \geq 1 + \log_2(n).$$

Důkaz je hotov.