1

Úvod do technologie programování a programovacích stylů, objektově orientovaný návrh, základní UML diagramy, psaní programů v Javě

Programování je proces vytváření software splňující určité funkce. K programování využíváme programovací jazyky, dříve procedurální (nestrukturované (asembler) - bez podprogramů, pouze skoky v programu (goto), nebo strukturované (basic, c, pascal) - podprogramy, skoky jen velmi vyjímečně), dnes nejčastěji objektově zaměřené (objekty jsou stavební kameny a obalují funkce, využití technik OOP).

Dříve bylo programování spojeno s testováním, nyní je proces rozdělen do několika fází:

Analýza - Sbírání informací od zákzníka či zadavatele

Návrh - Návrh SW tak, aby splňoval požadavky analýzy

Kódování - Implementace návrhu

Ladění - Testování a odstraňování chyb

Styly

Vodopád - fáze jdou za sebou, nevracíme se, ani neodbočujeme (nejstarší), týmy se zapojují do procesu pokud je na řadě jejich blok

Formální - návrh je matematicky zcela do detailu popsán a jednotlivé bloky modelu jsou poté 1:1 implementovány

Evoluční - začíná se malým funkčním SW, na který nabalujeme další funkce, ale nevíme, jaké funkce bude v budoucnu mít

Iterativní - proces je rozdělen na iterace, v každé iteraci je naplánováno několik funkčností, iterace stejně dlouhé, po iteraci je SW vždy v kompletní spustitelné podobě

Komponentový - využití hotových komponent, či implementace vlastních, po té stavění SW z komponent jako z kostek

Objektově orientovaný návrh

OOP umožňuje, a klade si za cíle:

Znovupoužitelnost - abstraktní části kódu jsou univerzální a je tedy možné je používat stále dokola

Robustnost - návrh předpokládá všechny vstup, nejen ty, které jsou očekávané

Přizpůsobivost - návrh lze pouze s minimálními zásahy do kódu použít na různých platformách

Pilíře OO návrhu

Dědičnost, **Zapouzdření**, **Polymorfismus** (viz PPA2)

Modularita - rozdělení programu na bloky - moduly

Abstrakce - Návrh na abstraktní úrovni je snadno implementovatelný

Jazyk UML

Jazyk pro abstraktní návrh OO SW, modelování SW před implementací, navrhujeme pouze strukturu, nikoliv funkčnost. Typy UML:

diagram tříd a objektů - popisují statickou strukturu systému

modely jednání (diagram případů užití) - dokumentují možné případy použití systému, události, na které musí systém reagovat

scénáře činností (diagramy posloupností) - popisují scénář průběhu určité činnosti, těchto činností je model jednání složen

diagramy spolupráce - zachycují komunikaci spolupracujících objektů

stavové diagramy - popisují dynamické chování objektu nebo systému

diagramy aktivit - průběh aktivit procesu nebo činností

diagramy komponent - popisují rozdělení systému na funkční celky a definují náplň jednotlivých komponent

diagramy nasazení - popisují umístění funkčních celků

Pro návrh diagramu tříd máme následující parametry:

Atributy a metody - definujeme viditelnost a vlastnosti (+ public, # protected, - private, / readonly, in/out směr)

Třída: obdélník, název tučně, pak seznam atributá a metod

Objekt: Název:Třída

Dědičnost - šipka od potomka k rodiči, může být i vícenásobná

Asociace - 0..1 1..1 0..N 1...N --> šipka od první třídy a ta obsahuje odkazy/využívá x..y objektů druhé třídy (např Podvozek 1 ---> 4 Kola), nad šipkou název činnosti (pokud možno lepší než "má")

Kompozice - je li objekt v kompozici s jiným, je na něj silně vázán (například seznam a položka seznamu. seznam může existovat i bez položek, ale položka bez seznamu ne), značka prázdný kosočtverec

Agregace - volná vazba, abstrakce "používá", objekty však mohou fungovat i nezávisle na sobě (například počítače v síti), značka plný kosočtverec

Java

Návrh - UML diagramy, CRC karty (karty, kde je pro každá blok sloupec Má za úkol a Vyžaduje). Je li toho na kartě příliš, je vhodné zvážit její rozdělení

Implementace - podle návrhu, karta odpovídá třídě.

Ladění a testování - kontrolní výpisy, logování, watch na proměnných, ..., po té test pro různá data (generovaná, "nepříjemná") a sledování správnosti, rychlosti, využití zdrojů, uvolnění

Dokumentace - štábní kultura, konvence pojmenovávání, komentáře, Javadoc komentáře a generovaná dokumentace

2
Abstraktní datové typy zásobník fronta, seznamy, řady, vektory a jejich implementace

Zásobník, fronta, seznam viz. PPA2 Obousměrná fronta - výběr a vložení možné na obou koncích (zásobník a fronta dohromady)

Vektor - v podstatě spojový seznam, nebo seznam pomocí pole, kde má každá položka index od začátku, a tedy přístup pomocí indexů, a indexy jsou přerovnány (nevznikají díry). Nejde o pole, pokud do pole na index 3 uložíme hodnotu, bude vždy na indexu 3, ve vektoru se může posunovat prodáváním/mazáním prvků před ním. = ADT seznam, kde místo iterátoru použijeme indexer

Řada - WTF? O tom jsem nikdy neslyšel, v přednáškách to není, na netu nic

- SPEKULACE: Představil bych si ADT, kde je každý prvek stejného typu (do vektoů či seznamu by teoreticky bylo možné uložit i různé prvky, do řady ne. Otázkou je proč by to někdo potřeboval, když pak by bylo možné například ukládat pouze diference od předchozího objektu a možná ušetřit místo??

3
Stromové struktury (Avl, BVS, B, Red-Black) a jejich implementace

Strom je ADT uchovávající prvky v hierarchické struktuře předcůdce-následovník. Použití v reprezentaci znalostí a stavového prostoru, popis scény při zpracování obrazu, v databázových systémech, systémech souborů, pro rozhodovací stromy, komprese dat, raytracing.

- **Obecné stromy** - uzly mohou mít libovolný počet následovníků. Implementace pomocí ukazatelů, ve vrcholech spojový seznam následovníku.
- **N-ární stromy** - uzel má nejvýše n následovníků. Implementace ukazateli, ve vrcholech pole následovníků.
- **Binární stromy** - uzel má nejvýše dva následovníky. Implementace ukazateli a výčtem následovníků, případne polem.

Binární strom, kde nalevo od každého prvku jsou k prvky s nižším klíčem a napravo s vyšším (příp. obráceně), je binární vyhledávací strom (BVS). Kořenem obyčejného BVS je první vložený prvek, další prvky se vkládají nalevo nebo napravo od něj. Při postupném vkládání seřazené posloupnosti ale takový strom degraduje na lineární seznam, což zvyšuje složitost vyhledávání. Proto je na místě snaha o vyvážení stromu.

AVL

(Adelson-Velsky, Landis)

BVS zachovávající pravidlo, že podstromy každého vrcholu se mohou výškou lišit nejvýše o 1. Vyvážení kontrolováno po každém vložení a výberu. Vrchol obsahuje navíc položku informující o rozdílu výšek obou jeho podstromu (-2, -1 levá strana, 0 vyvážený, 1, 2 pravá strana). Vložení stejné jako u BVS, po vložení prepocet rozdílu výšek (v každém kořenu na cestě z vloženého prvku do kořenu stromu sledujeme, jak se jeho vyvýžení změní). Při nevyváženosti v některém vrcholu rotace v tomto vrcholu.

Rotace:

- **Jednoduchá** - nevyváženost způsobuje levá větev levého podstromu nebo pravá větev pravého podstromu. Kontrolujeme, zda kořen podstromu má stejné znamínko jako kořen nadstromu ???
- **Dvojitá** - nevyváženost způsobuje pravá vetev levého podstromu nebo levá větev pravého podstromu. Provedeme nejdříve rotaci spodní větve, a tím se dostaneme na předchozí problém.

Odebrání stejné jako u BVS - nahrazení prvku symetrickým následovníkem, po odstranění prvku prepočty rozdílu výšek kontrola vyváženosti smerem ke kořeni (od původní pozice symetrického následovníka - prvky prohozeny a zde odstraňován).

// přidat obrázek

Red-Black

Další modifikace BVS. Vrcholy obsahují navíc položku s informací o barve (cervená nebo cerná). Pravidla: koren je vždy cerný, cesty od korene do každého listu obsahují stejný pocet cerných vrcholu (cerná výška stromu), cervený vrchol má pouze cerné následovníky.

Vkládaný vrchol je na zacátku vždy cervený, postupne od jeho pozice ke koreni kontrola pravidel a prípadná náprava

- přebarvením vrcholů: pokud je rodič červený a jeho bratr také, oba přebarvíme na černé a jejich rodiče na černé) - pokud bratr rodiče červený není, přebarvíme na černo pouze rodiče a provedeme rotace - pokud je rodič levým synem, a vkládaný také, provedeme rotaci do prava (rodič se stane kořenem) - pokud je rodič levý, ale vkládaný pravý, provedeme nejprve rotaci do leva (vkládaný se stane rodičem) a tím získáme předchozí problém, tedy přebarvíme a zarotujeme do prava podle vkládaného prvku, který se stane kořenem.

Rotace jsou stejné jako u AVL stromu. Odebrání - nahrazení symetrickým následovníkem, ten ale obarven barvou odebíraného vrcholu, a po té postupujeme stejně jako při vkládání, tedy od místa odebrání kontrola pravidel smerem ke koreni.

B-strom

B-strom řádu m má v každém vrcholu nejvýše m následovníků.

Pravidla:

- klíčů ve vrcholu je o 1 méně než následovníků
- všechny listy jsou stejně hluboko
- všechny vrcholy mají nejvýše m následovníků
- všechny vrcholy kromě kořene mají alespoň m^2 následovníků
- kořen je buď list nebo má alespoň 2 následovníky

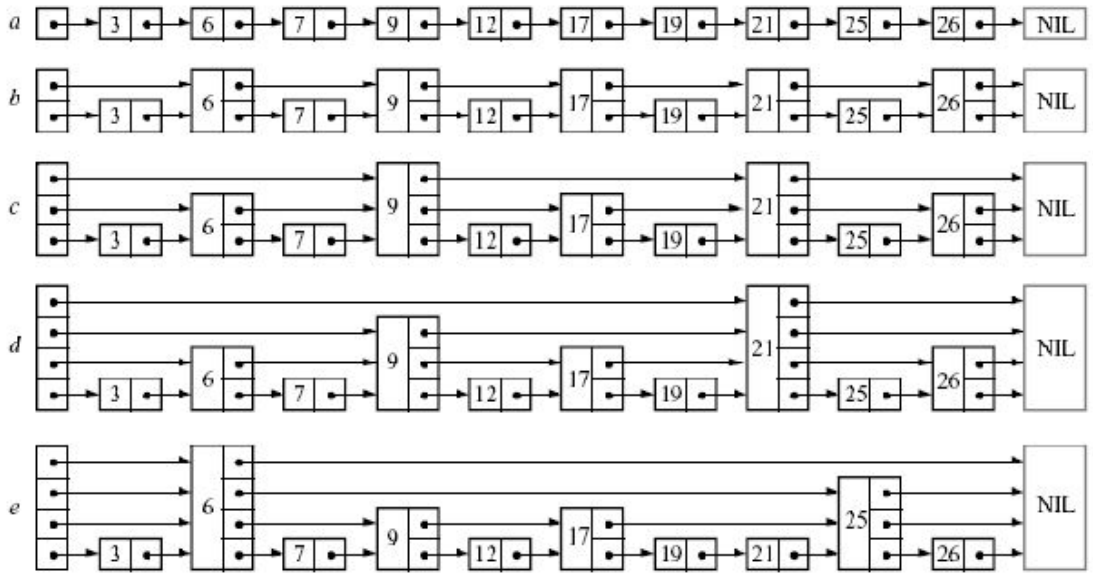
Ve vrcholech klíce serazené, vrcholy razeny jako u BVS (menší klíce vlevo, vetší vpravo). Pri vkládání se pridávají klíce do listu, pokud je v listu prekrocen maximální pocet (pretecení stránky), je list rozdelen uprostred a prostrední klíc vložen do rodice (zde opet kontrola prekrocení poctu). Odebereme-li klíc z listu a ten má potom príliš málo podtecení stránky), muže si vypujcit od sousedu (klíc z rodice do vrcholu, klíc ze souseda do rodice), pokud sousedi mají minimální pocet, dojde ke sloucení vrcholu s jedním sousedem a klícem z rodice. Pri odebírání z vnitrních vrcholu je nutné místo vždy zaplnit - klíc od potomka. Nemají-li potomci dost klícu, sloucí se dva potomci do jednoho. Implementace ukazateli s následovníky seznamem nebo polem, prípadne pomocí dvourozmerných polí (jedno pro klíce, jedno pro indexy následovníku, index korenu).

4
Skip-list - použití a implementace

Skiplist je alternativa ke stromům, která je snadno implementovatelná. Operace se skiplistem jsou podobné složitosti jako u stromů (O(n log n)), ale paměťová náročnost může být o něco větší.

Skiplist si lze představit následovně: seznam, ve kterém některé prvky ukazují i na vzdálenější prvky než jsou prvky bezprostředně následující. Tyto ukazatele "přeskakují" položky seznamu, odtud skiplist.

Maximální výška uzlu = Maxlevel, každý prvek může mít 1 <= k <= Maxlevel ukazatelů na další prvky



Popisky (obrázek je z přednášky):

- **a)** vidíme nejjednoduší skiplist se skokem 1, tedy lineární spojový seznam
- **b)** skiplist se skokem 2. Každý druhý prvek obsahuje kromě hodnoty a ukazatele na následníka také ukazatel na následovníka následovníka (next[next[x]]). Už touto změnou zkrátíme vyhledávání na polovinu, jelikož abychom se dostali na prvek např. 25, stačí nám pouze 5 skoků (> 6, >9, >17, >21, >25) na rozdíl od seznamu, kde jich je 9
- **c)** skiplist se skokem 4. na prvek 25 se dostaneme pouze 2 skoky
- **d)** skiplist se skokem 8. na prvek se dostaneme 1 skokem => ideální skip-list: každý 2i-tý prvek má ukazatel, který ukazuje o 2i prvků dopředu
- **e)** skiplist s náhodným skokem - ?? skoků, ale ve velkém skiplistu je rychlejší, než lineární seznam, a operace přidání a odebírání jsou jednodušší

Výhody:

- Skiplist je extrémně rychlý (vyhledávání je log n)
- Velmi snadná implementace (pouze pole ukazatelů v každém prvku)

Nevýhody:

- Pomalé operace insert a delete (je nutné reorganizovat skiplist)

Řešení: skiplist s náhodnými velikostmi uzlů. Potom:

- Vložení - najdeme místo, kam prvek patří, pomatujeme si všechny předchůdce. Po nalezení vložíme prvek s náhodnou výškou 1 < k < Maxlevel, a napojíme všechny předchůdce do výšky tohoto prvku (samozřejmě na obě strany, předchůdci ukazují na tento prvek, a ten ukazuje na prvky, na které předchůdci ukazovali původně)
- Mazání - opačný proces, najdeme prvek a pomatujeme si předchůdce. Pro všechny předchůdce do výšky hledaného prvku

upravíme ukazatele (předchůdci budou ukazovat na prvky, na které ukazoval hledaný prvek). Hledaný prvek smažeme.

Tento skip-list není zcela log n, a nelze (díky náhodnosti) jeho složitost určit. S vysokou pravděpodobností však všechny operace budou rychlejší než O(n) (samozřejmě můžeme mít smůlu a všechny prvky budou náhodně výšky 1).

5
Tabulky s rozptýlenými položkami, vyhledávání v tabulkách

To samé jako PPA2 14 a 15,

jen doplňky/opakování/upřesnění:

Vnitřní zřetězení - synonyma (položky s kolizemi) se ukládají do volných míst

Vnější zřetězení - synonyma se ukládají do seznamů pod klíči

Obsah

Otevřené rozptýlení - hash funkce

Kolize:

1) nedefinované chování při kolizi - nutno dodefinovat

2) lineární posun - při kolizi vkládáme na první volné místo lineárně (na indexech 1,2,3,4 od původní polohy

3) kvadratický posun - při kolizi hledáme první volné místo na indexech podle kvadratické funkce (vzdálenosti 1, 4, 9, 16...)

4) opakovaný hash - hashovací fci použijeme víckrát pokud dojde ke kolizi

Uzavřené rozptýlení - přímé adresování

metody 2) a 3) nejsou vhodné, vzniká shlukování (clustering), proto je dobré použít způsob 4) nebo lepší hash funkci

Tabulky s uzavřeným rozptýlením a vnějším zřetězením

= kombinace obou

- adresujeme přímo, při kolizi si ale do tabulky na konec uložíme, kam budeme vnitžně zřetězovat. Hash funkce tedy může být při kolizi (třeba náhodná), nebo tabulku rozdělíme na 2 části, jednu pro vyhledávání klíčů, a druhou pro ukládání synonym. ta už může být dynamicky alokovaná.

Metoda rozptýlených indexů

To co bylo v předchozím případě v tabulce vyjmeme, a vytvoříme vedle. Uděláme si mapu synonym (synonyma v seznamu pro každý klíč) a tyto pak ukazují na index do pole.

Vyhledávání

Má samozřejmě smysl jen pokud nemáme orákulum (= hashovací fci). Pokud orákulum máme, není třeba hledat protože ideální orákulum vždy ví, kam se podívat pro správná data (The all-knowing Oracle is never surprised. How can she be, she knows everything. -- The Matrix: Revolution), reálné ví aspoň kde hledat.

Jelikož je při přímém adresování orákulum pouhý index do tabulky (mno, to moc orákulum není, orákulum vždy ví, kam jít, aby nebyla kolize), alespoň nám napoví odkud hledat.

Sekvenční:

Hledáme klíč lineárně, pokud na něj narazíme, vrátíme hodnotu

+ snadná implementace

+ tabulka nemusí být uspořádaná

- pomalé hledání

Binární

Jako sekvenční, ale klíče musí být seřazeny, a nad nimi vytvoříme BVS, tedy hledáme vždy v půlce intervalu

+ rychlejší (O(log n))

- nutnost udržovat tabulku seřazenou

Fibonacciho

Jako binární, ale nerozdělujeme v polovině, ale podle Fibonacciho posloupnosti (tedy ne 1:1 -> 1:(1:1) -> 1:((1:1): 1) ale (1:1) -> (1: (1:2)) - > (1:((2:3):2))

Prvky vybíráme jako dvojice, první z dvojice je ve Fibonacciho posloupnosti na pozici odpovídající úrovni vnožení stromu (výška větve, ve které hledáme)

Fibonacciho ppst: a1 = 1, a2 = 1, ai = ai-1 + ai - 2 , tedy 1,1,2,3,5,8,13,21,34,55, ...

Podle sekundárního klíče

Jen pouze pokud sekundární klíč existuje v datech, např seznam jmen, jméno je primární (není, ale pro vysvětlení řekněme, že je) a rok je sekundární. Vyhledáme jména v sekundárním klíči a v nich pak hledáme jméno podle primárního klíče.

Sekundární klíče máme v seznamu, který obsahuje seznam primárních klíčů zabulky pro každý sekundární klíč

6

Algoritmy zpracování textů – operace s řetězci, porovnání se vzorem (KMP, Boyer-Moore algoritmus), nejdelší společný podřetězec (LCS algoritmus), vzdálenost mezi řetězci, datová struktura Trie a použití

Tak tahle otázka je očistec. Kdo tohle všechno vysvětlí za 15 minut je borec, i když bych řekl, že je tam rozsekaj na menší otázky.

Obsah

Obecně

Vyhledáváme podřetězec P v řetězci T. T je string s textem, P je string s maskou (pattern)

Podřetězec je část řetězce [i..j]. T je délky n, P je délky m.

Prefix je část řetězce (či podřetězce) od začátku, suffix je část řetězce (či podřetězce) od konce.

Brutální síla

Indexem jdeme po T a kontrolujeme, jestli podřetězec začínající na tomto indexu neodpovídá P (tedy kontrolujeme m prvků od indexu směrem do prava. V nejhorším případě O(nm), což je pomalé

T: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaah

P: aah

P v T najdeme až úplně na konci, pro každé 'a' v T (kromě třetího od konce) znovu hledáme písmena P v řetězci T zcela zbytečně

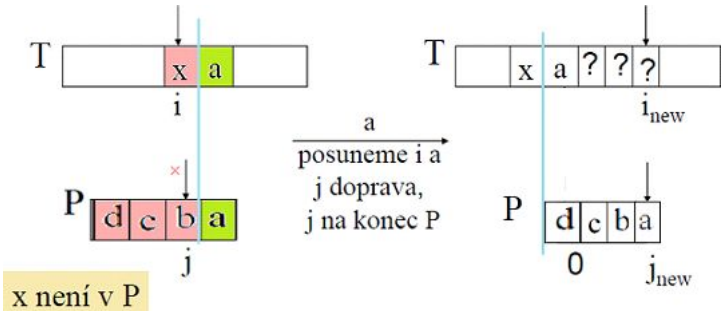
Boyer-Moore (BM) algoritmus

V tomto algoritmu prohledáváme P od konce, a hledáme výskyt v T. Do T ukazuje index i, do P index j.

Mohou nastat 4 případy:

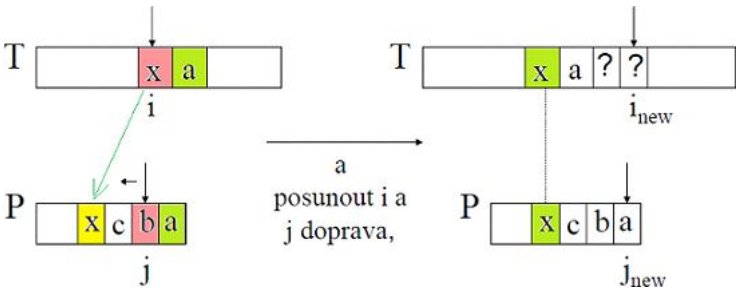
1) T[i] není v P vůbec, posuneme i dále o délku P (zarovnáme P na další písmeno v T, tedy T[i+1]

(obrázky v přednáškách jsou křivé, tak se mi to snad povedlo odstranit)

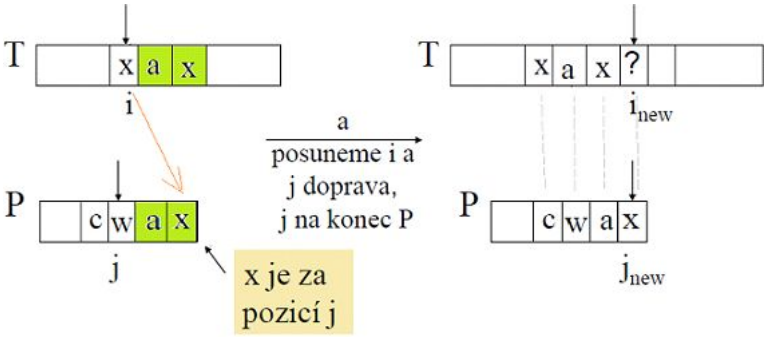


2) T[i] odpovídá P[j] - posuneme se v obou do leva a opakujeme (jako v brutální síle)

3) T[i] není P[j], ale T[i] je v P před indexem j -> zarovnáme P do prava tak, aby T[i] odpovídal jeho výskytu v P a opakujeme



4) T[i] není P[j], ale T[i] je v P za indexem j -> posuneme se do prava o 1 a opakujeme (nemůžeme se vracet, ale ani posunout o více dále)



Pokud nalezneme celý pattern, vrátíme i. Algoritmus je pro texty rychlejší, než brute force, i tak je však jeho složitost O(mn + A), kde A je velikost abecedy. Pro velké abecedy funguje rychle, ale např pro binární soubory funguje pomalu.

Dodatek: Algoritmus si v preprocesingu zjistí, na které pozici má které písmena (směrem od leva). Je-li patern tedy například "ABRAKADABRA", A dostane index 10, B dostane index 8, K = 4, D = 6 a R = 9. Pro ostatní písmena přiřadíme -1. Naimplementujeme tedy funkci Last(char input), která podle písmene vrátí tento index, a pak pro případy 3 a 4 porovnáváme Last(T[i]) a j, a tím pak víme, jestli i posunout na Last(T[i]) (pokud je Last(T[i]) < j) nebo pouze i++

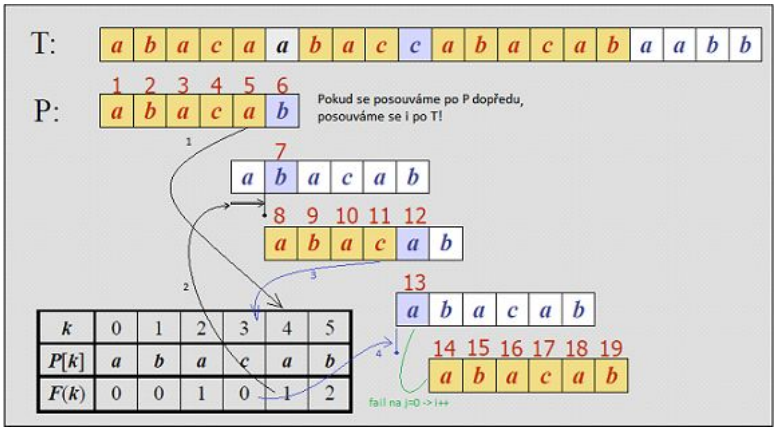
Knuth-Morris-Pratt (KMP) algoritmus

Jako brutal force jde zleva do prava, ale nedělá otrocky všechny porovnání. Pokud narazíme na neshodu, je možné se posunout o více než o jedno písmeno.

Otázka zní o kolik?

Odpověď, o maximální prefix P(0 .. j-1), který je suffixem P(1 .. j-1). Suffix a prefix se nesmí překrývat.

Proč? Protože najdeme-li kus P (od začátku, tedy prefix), víme, že znaky tohoto prefixu odpovídají textu. Je tedy netřeba je kontrolovat znovu. Může se ale stát, že konec nalezeného podřetězce je také obsažen v začátku tohoto podřetězce. Takovou shodou je samozřejmě celá nalezená část P, proto hledáme od P+1. Takže jdeme od konce nalezeného kusu P zleva a zprava, a ve chvíli, kdy nenalezneme shodu, víme, o kolik se můžeme posunout. Tohle se samozřejmě dá předpočítat do tabulky, a pak je vše O(1)



(i.e. P[j] != T[i]), pak k = j-1; j = F(k); // získání nové hodnoty j, kde F(k) je právě předpočítaná tabulka společných částí P

Trie

Trie je struktura pro předspracování textu pro pozdější rychlé vyhledávání. V každém uzlu standartní Trie je písmeno a délka cesty k vrcholu je také pozicí písmene ve slově, tedy je-li hloubka písmene 'E' v Trie rovna 3, víme, že je to třetí písmeno (např. slova Trie)

Standartní trie má v každém uzlu právě jedno písmeno. Vyhledávání je O(d * m) kde d je velikost abecedy (v každém uzlu musíme projít všechna písmena, abychom zjistili, že tam písmeno je), m je délka slova. Paměťově je Trie nejvýše složitosti O(n) kde n je délka původního textu. Pokud se slova překrývají, paměťi je potřeba méně.

Rychlost se dá na úkor paměti vylepšit tak, že v každém uzlu připravíme celou abecedu a při hledání převedeme písmeno na index v této abecedě, tím pádem získáme v O(1) čase pointer na další písmeno ve slově.

Komprimovaná trie

Jako trie, ale shlukujeme písmena. Tedy ve vrcholech stromu jsou všechna písmena, která jsou společná pro všechna slova. Tedy pro slova 'ahoj' a 'ahojky' bude mít trie 2 vrcholy, 'ahoj' a pod ním 'ahojky'. Tuto Trie lze převést na standardní pouhým rozdělením shluků, standardní Trie převedeme na kompaktní sloučením vrcholů s rodiči, kteří mají pouze jeden odkaz na vrchol (tento vrchol).

Pak je ještě suffixová Trie, která ukládá všechny sufixy slov v komprimované podobě. Umožňuje vyhledávat i uprostřed slova.

LCS - Longest common subsequence

Tohle je skoro nemožný vysvetlit na statickym papiru....

Cílem algoritmu je najít nejdelší společný podřetězec libovolného počtu řetězců. To je problém NP-těžký, což je masakr. My se ale omezíme na porovnávání 2 řezězců, což je v polynomiálním čase řešitelné (dynamicky).

Brutální síla by dělala to, že by našla všechny podřetězce obou řetězců, a ty porovnávala (také brutální silou) Podřetězců je však řekněme n!, brutální síla je n2, takže (n!)2. Dynamicky to lze dělat rychleji. Použijeme tedy rovnou dynamické programování, protože to nám dovoluje pamatovat si so už jsme našli, a neztrácet čas znovuobjevováním nalezeného.

Nechť Li,j = maximální délka společných řetězců končících v Ai a Bj . Vytvoříme si tedy dvojrozměrnou tabulku, do které si tyto Li,j budeme zapisovat.

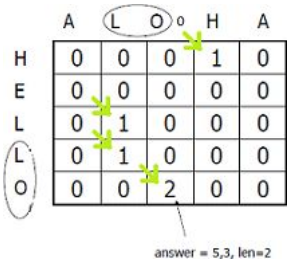
Nyní, je-li písmeno na pozici i v řetězci A stejné, jako písmeno na pozici j v řetezci B, víme, že jde o společný podřetězec (délky 1). Nevíme však, zda předchozí písmena nebyla také podřetězcem (zatím)

Víme ale následující: Pokud existoval pořetezec pro indexy i-1 a j-1 (tedy pro předchozí písmena v obou řetězcích), bude v tabulce na příslušné pozici, tedy vlevo nahoře od Li,j tedy Li-1,j-1. Na této pozici je tím pádem dosavadně nalezená délka společného podřetězce, který končí na indexech i-1 a j-1. Pokud by i předchozí indexy daly stejné písmeno, musíme tuto délku dále diagonálně propagovat, a tedy

Ai = Bj -> Li,j = Li-1,j-1 + 1 -> společný řetězec končící na indexu i-1,j-1 pokračuje

Ai <> Bj -> Li,j = 0 -> není společný (pokud na i-1 a j-1 byl, na tomto indexu končí)

Takto vyplníme celou tabulku, přičemž nulté řádky budou nulové (abychom nevyšli pro první písmena z hranice pole)



Problém: Jak zjistit, kde je řešení nejlepší? Ano, můžeme najít maximum v tabulce, ale to je další O(M + N) operace, a máme tabulku plnou nul, toho lze využít. Algoritmus tedy modifikujeme tak, že bude zachovávat i dosavadně nejlepší nalezenou délku společného podřetězce v každém bodě. toho docílíme úpravou druhého pravidla, tedy

Ai = Bj -> Li,j = Li-1,j-1 + 1 Ai <> Bj -> Li,j = max (Li-1,j, Li,j-1)

Nejlepší nalezené řešení je nahoře, nebo vlevo. Tím se tabulka naplní, a my budeme znát nejlepší délku v pravém spodním rohu tabulky

	C	O	M	P	U	T	E	R
H	0	0	0	0	0	0	0	0
O	0	1	1	1	1	1	1	1
U	0	1	1	1	2	2	2	2
S	0	1	1	1	2	2	2	2
E	0	1	1	1	2	2	3	3
B	0	1	1	1	2	2	3	3
O	0	1	1	1	2	2	3	3
A	0	1	1	1	2	2	3	3
T	0	1	1	1	2	3	3	3

Nyní ale nevíme, kde ta písmena jsou. Je tedy nutné si to ukládat do druhé tabulky (označit si, kudy jsme na které pole přišli, z leva, ze shora (při zpětném průchodu pouze posun), nebo šikmo (při zpětném průchodu výpis a posun)) a následně zrekonstruovat zpětným průchodem. V přednáškách jsou šipky, ukazující kudy se budeme vracet. Při provádění pravidla 2 si samozřejmě tuto informaci musíme také propagovat, pomatujeme si tedy, odkud jsme na každé pole přišli (v druhé tabulce si to lze označit třeba 0, 1 pro nahoru a do leva, 2 pro šikmo)

A konečně...

Vzdálenost mezi řetězci

Hammingova metrika: na stejně dlouhé řetězce, jen je porovnáme, a v O(n) čase víme, kolik změn písmen musíme provést (Auto, Zutá = 2)

Levensteinova metrika: někdy také edit distance, máme operace změnit/přidat/odebrat písmeno, abychom dostali z jednoho řetězce ten druhý. Jde o celočíselné vyjádření podobnosti řetězců, například

P=abcdefghijkl, T=bcdeffghixkl, P odpovídá T po 3 změnách (v T přidat a, ubrat f a zaměnit x za j)

Algoritmus: Postup podobný jako u LCS, ale matice je D[i,j] a prvky vyplňujeme takto

máme 3 možné způsoby úpravy řetězce

1. P[i]=T[j] , s = D[i-1,j-1] (shoda) jinak D[i-1,j-1] + 1 (substituce, platíme cenu za změnu znaku)

2. v = D[i-1,j] + 1 (znak navíc v P, neposouváme pointer v T (při úpravách) a platíme cenu za vkládání)

3. r = D[i,j-1] + 1 (znak navíc v T, neposouváme pointer v P (při úpravách) a platíme cenu za rušení)

minimem z s, v a r získáme pole D[i,j]

		b	c	d	e	f	f	g	h	i	x	k	l
	0	1	2	3	4	5	6	7	8	9	10	11	12
a	1	1	2	3	4	5	6	7	8	9	10	11	12
b	2	1	2	3	4	5	6	7	8	9	10	11	12
c	3	2	1	2	3	4	5	6	7	8	9	10	11
d	4	3	2	1	2	3	4	5	6	7	8	9	10
e	5	4	3	2	1	2	3	4	5	6	7	8	9
f	6	5	4	3	2	1	2	3	4	5	6	7	8
g	7	6	5	4	3	2	2	2	3	4	5	6	7
h	8	7	6	5	4	3	3	3	2	3	4	5	6
i	9	8	7	6	5	4	4	4	3	2	3	4	5
j	10	9	8	7	6	5	5	5	4	3	3	4	5
k	11	10	9	8	7	6	6	6	5	4	4	3	4
l	12	11	10	9	8	7	7	7	6	5	5	4	3

Řešení je opšt ve spodním pravém rohu a zpět postupujeme po stejných číslech (žádná změna) nebo menších (změna, jdeme-li šikmo, substituce, jdeme-li nahoru rušení, do leva vkládání)

7
Komprese dat, rozdělení kompresních metod, princip kompresních metod (Huffmann, aritmetické kódování, LZW, JPG, fraktálová komprese)

Algoritmy pro kompresi dat používáme pro snížení velikosti přenášených dat

- z paměťových důvodů
- ušetření šířky pásma
- urychlení přenosu a zkrácení doby odezvy
- archivace dat, zálohy
- obrana proti virům
- distribuce software
- ...

Kvalita komprese - úroveň zmenšení, poměr komprese = před/po. Symetrické metody potřebují stejný čas ke komprimaci jako k dekomprimaci

Obsah

Rozdělení kompresních metod

- **Bezztrátová** - po komprimaci a dekomprimaci je výsledek k nerozeznání od originálu, 100% shoda. Metody jsou méně efektivní. Použijeme na text, obecné soubory
- **Ztrátová** - po komprimaci a dekomprimaci je výsledek rozdílný, cílem je však aby byl co nejvíce podobný. Více efektivní. Použijeme na zvuk, obraz, video.

Metody komprese

- **Jednoduché** - založené na kódování opakujících se posloupností znaků (Run-Length Encoding - RLE)
- **Statistické** - založené na četnosti výskytu znaků v komprimovaném souboru (Huffmanovo kódování, Aritmetické kódování)
- **Slovníkové** - založené na kodování všech vyskytujících se posloupností (Lempel-Ziv-Welch - LZW)
- **Transformační** - založené na ortogonálních popř. jiných transformacích (JPEG, waveletová komprese, fraktálová komprese)

RLE

RLE (Run Length Encoding) – kódování délkou běhu

Nejjednodušší komprimační algoritmus, nahrazuje opakující se znaky jedním znakem a číslem vyjadřujícím počet okpakování (AAAAABBBBBCCD -> 5A5B2C1D) Pro text k ničemu, pro obraz celkem dobrá komprese (zvláště obrazy používající malou paletu barev) Problém: pokud se znak neopakuje, dochází k expanzi (ABC -> 1A1B1C) - Řešení pomocí escape sekvencí (označíme si, co je délka a co ne) a poté komprimujeme pouze opakující se znaky. Escape znak nesmí být obsažen v původním textu.

00000000000000000001110000000000000000000000000111	19 3 26 3
00000000000000000001110000000000000000000000000111	19 3 26 3
00000000000000000001110000000000000000000000000111	19 3 26 3
00000000000000000001110000000000000000000000000111	19 3 26 3
00000000000000000001111000000000000000000000001110	20 4 23 3 1
00000000000000000000011100000000000000000000111000	22 3 20 3 3

Ukázka komprese bitového souboru pomocí RLE. Ukládáme vždy počet nul a jedniček v řadě.

Huffmanovo kódování

Statistická metoda. Vyhledáme četnosti všech znaků v řetězci a tyto znaky podle četnosti zakódujeme tak, aby nejkratší kód odpovídal nejčastějšímu znaku. Kódy jsou binární, nejčastější znak bude tedy reprezentován 1 bitem na rozdíl od 8(16) bitů potřebných pro uložení ASCII(UTF) znaku. Jená se o prefixový kód, takže žádný znak není prefixem jiného znaku -> kódy pro jednotlivé znaky mají různou délku a není potřeba označovat jejich konec (každý znak kreprezentuje jeden list stromu).

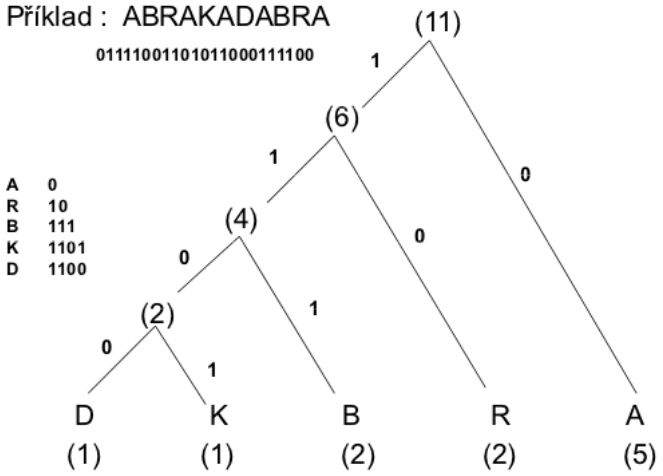
Kódy vytvoříme Huffmannovým stromem:

- Jednotlivé znaky označíme jako listy stromu
- Tyto listy uložíme do seznamu S (S obsahuje uzly stromu, na začátku tedy pouze listy)
- Seznam S seřadíme podle četnosti

- Vybereme z S dva prvky M,N s nejnižšími četnostmi m, n, m < n
- Vytvoříme uzel p, kde vlevo je M a vpravo je N, a jeho četnost bude m+n
- Vložíme p do S a zpět na bod 3 dokud v S není jen jeden uzel

Strom samozřejmě musíme přenášet spolu s komprimovanými daty. Strom můžeme reprezentovat řetězcem tak, že jej projdeme DFS algoritmem a pro haždý vrchol uložíme 0 a pro každý list 1 a znak tohoto listu. Při načítání čteme nuly a vytváříme uzly (směrem doleva). Pokud narazíme na 1, zapíšeme písmeno a vrátíme se do nejbližšího pravého uzlu (pravý uzel rodiče, případně nejvíce levý prvek podstromu - u rodiče vpravo)

Při dekompresi procházíme strom a rekonstruujeme zprávu.



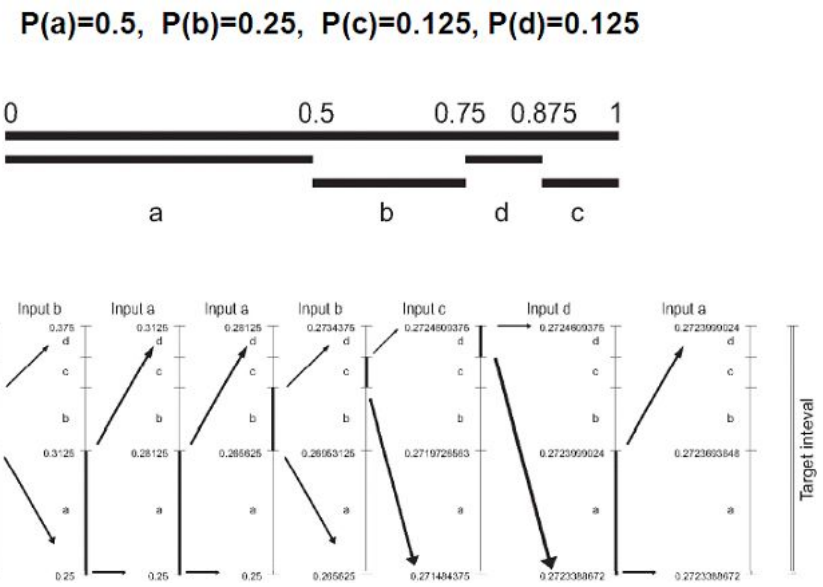
Aritmetické kódování

Také statistická metoda. Aritmetické kódování na rozdíl od jiných metod nepracuje na principu nahrazování vstupního znaku specifickým kódem. Místo toho se kódovaný vstupní proud znaků nahradí jediným reálným číslem z intervalu <0,1).

Na základě pravděpodobnosti výskytu jednotlivých symbolů vstupní abecedy je každému symbolu přiřazena odpovídající poměrná část intervalu <0,1). Při kódování je pak celý interval <0,1) postupně omezován z obou stran na základě postupně přicházejících symbolů (Interval ořízneme jen na tu část, které odpovídá pravděpodovnost písmene v textu, a tento interval opět rozdělíme podle pravděpodobností pro další písmeno).

Kódovaná hodnota se reprezentuje libovolným reálným číslem, které leží ve výsledném intervalu získaném po přečtení všech vstupních symbolů. Vzhledem k tomu, že z takto reprezentované hodnoty nelze při dekódování určit konec zprávy, je třeba navíc ke zprávě přidat zpeciální znak označující konec, případně musí být uložena i délka původní posloupnosti.

Příklad: kódujeme zprávu 'abaabcda'



Co z toho číst?

- První řádka ukazuje pravděpodovnosti písmen (ty vypočteme jako četnost písmene / písmen celkem)
- Dále vidíme, jak bude interval (a podintervaly) rozdělený. 'a' jako nejpravděpodobnější písmeno dostane největší část intervalu atd...
- A dále už kódujeme. Jdeme zleva do prava. Přijde písmeno 'a', vybereme tedy spodní interval. Tento opět rozdělíme na stené části jako interval původní a přijde 'b'. Opět vybereme odpovídající interval, který rozdělíme atd.
- Zvýrazněná část intervalu se na obrázku vždy zazoomuje ;) (jsou tam šipky).
- Na konci máme jen jeden interval, vybereme tedy číslo , které do něj patří. Zpráva je zakódována v desetinné části tohoto čísla (a ano, celá zpráva je reprezentována jedním číslem)

LZW

Z přednášek, doplněno: Slovníkový algoritmus. Využívá skupin znaků označených kódy a přenos kódů, které mají menší velikost než původní skupiny. Princip: Jednoprůchodová metoda (nevyžaduje předběžnou analýzu souboru. Ve výsledku se jen posunujeme po souboru jedním ukazatelem.) Vyhledávání opakujících se posloupností znaků, ukládání těchto posloupností do slovníku pro další použití a přiřazení jednoznakového kódu těmto posloupnostem. Kódy musí mít délku (počet bitů) větší než délka původních znaků (např. pro ASCII znaky (8 bitů) se obvykle používá délka kódu 12 bitů popř. větší. Na osmi bitech kódu by se dal uložit pouze slovník o 256 slovech(posloupnostech znaků), což odpovídá pouze počtu písmen, což je k ničemu.) Při průchodu komprimovaným souborem se vytváří slovník (počet položek slovníku odpovídá hodnotě 2(počet bitů kódu), kde prvních 2(počet bitů původních znaků) položek jsou znaky původní abecedy a zbývající položky tvoří posloupnosti znaků obsažené v komprimovaném souboru (používáme tedy i kódy z již komprimovaného souboru k vytvýření nových kódů). Při dekompresi procházíme kódy, a podle tabulky je rozebíráme na menší a menší, až se dostaneme na elementární znaky, které vypíšeme.

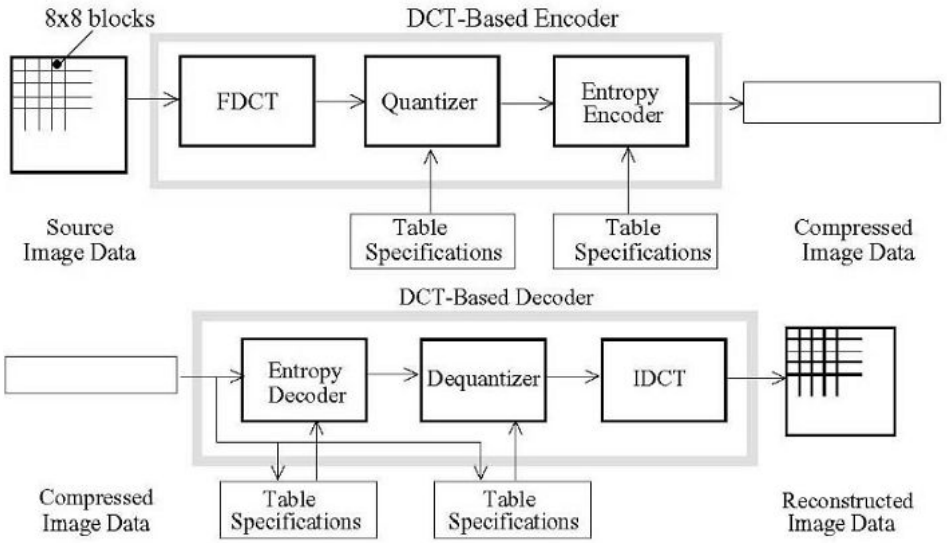
Aby to celé fungovalo, musíme slovník na dekomprimační straně "naučit" slova od menších po větší. Musíme proto na začátku souboru definovat elementární znaky, a z nich poté skládat slovník. Pro ASCII tedy prvních 256 znaků tabulky může odpovídat 1:1 ASCII abecedě, až poté následuje slovník. Pokud při dekompresi narazíme na neznámý kód, víme, že je složen z předchozích 2 kódů. Proto se jej naučíme, a příště už víme, co zapsat (případně jak jej rozložit). Tabulka na obou stranách tedy je stavěna stejně, jen používána z jiné strany (při komprimaci ukládáme kódy, pri dekomprimaci jejich znaky)

Pro obrázky viz přednášky z PT, jsou celkem snadno pochopitlené, i když místo "kód" občas autor používá "znak" či "nový znak", což je trochu matoucí.

JPG

V současné době patří mezi nejvíce používané komprese u obrázků Je vhodná pro komprimaci fotek, nevhodná pro např. technické výkresy (čarové výkresy) - dochází k viditelnému rozmazání (na webové aplikace pomalu převládá png (zejména kvůli neztrátovosti a průhlednosti))

Princip: části obrazu se transformují do frekvenční oblasti (výsledkem je matice „frekvenčních“ koeficientů, bezztrátový převod) z matice koeficientů se odstraní koeficienty odpovídající vyšším frekvencím (rychlejší změny jasu - např. hrany v obraze, zde jsou ztáty) zbývající koeficienty se vhodným způsobem zkomprimují (huffman, aritmeticky)



Fraktálová komprese

Fraktál je obraz, který je matematicky popsatelný a tím jej lze libovolně přibližovat a oddalovat bez stráty informace. Také lze fraktál popsat jako soubor matematicky popsaných sebepodobných částí, jejichž spojováním dostáváme také sebepodobný obraz ve větším detailu (jako když děláme koláž z fotek, které dohromady a z dálky vypadají jako fotka)

Toho lze využít pro kompresi. Pokud se podíváme na libovolný obrázek, často zjistíme, že některé jeho části jsou si podobné

Tyto malé kousky (frakce) lze pomocí rotace, změny velikosti a posunu otočit tak, že si budou velmi podobné (nebo dokonce budou stejné). V obrazu tedy můžeme na takový kousek mít pouze jednu, a v ostatních výskytech pouze uložit tyto operace.

V praxi obrázek rozřežeme na shluky 2x2 pixely (range bloky), a pak hledáme, kde jinde v obrazu je můžeme najít (domain bloky). Domain bloky se mohou překrývat, range bloky se nepřekrývají. Postupujeme tak, že Domain blok vytvoříme jako dvojnásobek range bloku, a v obrazu postupujeme zleva do prava. V bloku uděláme průměr hodnot pixelů pod ním tak, abychom získali blok velikosti range bloku, a poté hledáme, kterému range bloku je takto zmenšený domain blok podobný. Uložíme si operace, které s domainblokem musíme udělat, aby byl range bloku podobný (rotace, překlopení, posun...) Pokud je domainblok složen z domainbloků, postupujeme rekurzivně. Tím, pomocí podobnosti, získáme komprimovaný obraz.

8

Grafové algoritmy (cesta – Dijkstra, Floyd-Warshal, kostra – Prim-Jarnik), reprezentace grafu (matice, seznam sousednosti), základy kryptografie (symetrické, asymetrické šifrování)

//Detaily viz PPA 11, 12, 13

Obsah

Reprezentace grafu

Seznam sousednosti - každý vrchol má seznam svých sousedů. Je-li graf neorientovaný, hrany jsou v seznamech dvakrát

Matice sousednosti - matice v * v, kde v je pořet vrcholů, a v ní jsou '1' tam, kde je hrana. Na diagonále jsou smyčky, případně ohonocení vrcholů. Pro neorientovaný hraf je matice symetrická.

Graf může mít v*v-1 hran, podle očekávaného počtu se tedy rozhodneme, kterou možnost zvolíme.

Dijkstrův algoritmus

= algoritmus pro hledání nejkratší cesty v ohodnoceném hrafu grafu.

- hrany mají kladné hodnocení

- většinou se bavíme o neorientovaném grafu, funguje však i pro orientované

- probírán 1000x, takže jen opakování:

Máme ohodnocený graf a dva vrcholy, u a v, a hledáme cestu z u do v

Začneme v u, a tomu přiřadíme vzdálenost 0

Podíváme se na všechny okolní body všech již označených bodů

Posuneme se na ten, který má nejkratší vzdálenost od u (tedy musíme do vzdálenosti připočítat i vzdálenost předchozího bodu od počátku. Tohle je extrémně důležité, protože jinak bychom nalezli MST (konkrétně primův algoritmus), a tomu přiřadíme vzdálenost odpovídající součtu vzdálenosti předchozího bodu (od u) a délky cesty tohoto bodu od jeho předchůdce.

Opakujeme 3 a 4, dokud nenalezneme cílový bod v.

kombinace BFS (pro zjištění všech okolních nejkratších cest) a DFS (pro posun po prvcích). Algoritmus zastaví, pokud najde žádanou cestu.

Implementace

tabulkou, kdy ke každému vrcholu zapisujeme vzdálenosti, vybíráme v každé řádce nejmenší a tu označíme. Vyhledávání minima by bylo však O(v) (projdeme všechny sousedy vrcholu a hledáme nejbližší) pro všechny hrany, tedy celkem O(v2)

jinak, například pomocí prioritní fronty s možností úpravy. všechny nalezené a neoznačené okolní vrcholy umístíme do prioritní fronty s prioritou jejich vzdálenosti (opět jde o součet délky cesty do předchozího bodu a délky cesty od předchozího bodu do vkládaného bodu), vybereme zepředu fronty, ten prvek označíme, zapamatujeme si, odkud jsme na něj přišli pro pozdější rekonstrukci cesty, a vložíme jeho okolní vrcholy. pokud již ve frontě jsou, upravíme jejich prioritu, je li to nutné. Složitost, pokud je fronta implementována haldou, je O(h + v log v), h protože procházíme všechny hrany a v log v protože pro každý vrchol musíme haldu obnovit.

Floyd-Warshallův algoritmus

= algoritmus pro hledání nejkratší cesty pro všechny páry (u, v) v grafu.

- Rychlejší, než Dijsktra pro všechny výchozí body (Dijsktra najde od zadaného vrcholu všechny nejkratší cesty, pokud jej na cílovém vrcholu nezastavíme), pokud je graf hustý (tedy hran je v2) -> Dijsktra by byl (pomocí haldy) O(h + v log v), tedy O(v2 + v log v), tedy horší než v3, při implementaci polem dokonce v4

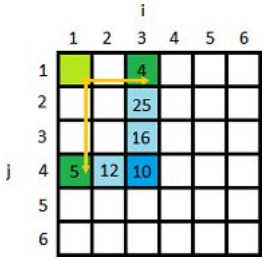
- Složitost O(v3)

Algoritmus

```
FLOYD-WARSHALL(G,d,mezi)
    vstup  :  souvislý graf G=(V, H) s nezáporným ohodnocení
    výstup :  d[i,j], i,j ∈V;
              mezi[i,j], i,j ∈V;
    for i := 1 to |V| do
        for j := 1 to |V| do
            begin d[i,j] := w[i,j];
                  mezi[i,j] := null
            end;
    for k := 1 to |V| do
        for i := 1 to |V| do
            for j := 1 to |V| do
                if d[i,k]+d[k,j] < d[i,j]
                    then begin d[i,j] := d[i,k]+d[k,j];
                               mezi[i,j] := k
                         end;
```

Pracuje s maticí sousednosti w, kde jsou hrany označeny jejich délkou (cenou, vzdáleností, ...) a nehrany označeny nekonečnem. d je matice aktuálne spoctených nejkratších vzdáleností mezi je matice nejkratších mezicest, je nainicializována na -1 (null)

V každém kroku algoritmu zjištuji, jestli existuje mezi vrcholy i,j kratší cesta pres vrchol k, pokud ano, nastavím vzdálenost v d[i][j] na novou velikost a do mezi[i][j] zaznamenám vrchol k. V podstatě procházím všechny buňky matice D v kříži, který protíná prvek (i, j). Pokud jsem dříve nalezl optimální cestu do bodu, je tato cesta ideální, pokud neexistuje cesta přímá. (Víme, že v bodě D[i,j] jsou na začátku všechny hrany. Může se ale stát, že existuje kratší cesta přes jiný vrchol, než po hraně v matici sousednosti. Kde tu cestu najdu? Buď končí ve vrcholu i, nebo ve vrcholu j, každopáně bude v řádce j, nebo sloupci i. a jde přes jiný vrchol. Pokud tedy znám cesty z tohoto vrcholu do ostatních vrcholů, můžu najít cestu z vrcholu i do jiného, třeba x, a z x do j tak, že délka z i -> x + x -> j je kratší, než doposud nalezená nejkratší cesta)



i = 3, j = 4, tedy cesta z i do j. V matici sousednosti byla 10, ale pokud půjdeme z vrcholu 4 do vrcholu 1 a z něj do vrcholu 3, tato cesta bude kratší (9 < 10). Do matice mezi si tedy uložíme vrchol 1 a v matici D (tato) změníme délu nejlepší cesty z i do j na 9 (tmavomodré pole)

// Nechápu, proč je to v přednáškách vysvětleno tak přes ruku. Vždycky. Stačí vždy jen říct jak a proč, a je to hned jasné. Výpis programu v polopascalovskym kodu nikdo nechápe.

Prim-Jarníkův algoritmus

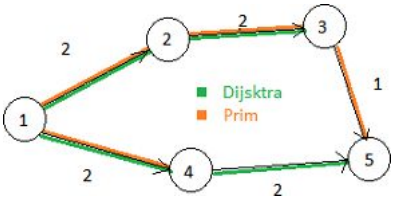
= algoritmus pro řešení úlohy minimal spanning tree (MST), tedy problému, kdy pro zadaný graf ((ne)orientovaný, ohodnocený) hledáme kostru (strom), která bude mít nejmenší možné celkové ohodnocení (součet ohodnocení hran bude minimální)

- užití - nejlevnější spojení všech uzlů grafu (elektrifikace domů, stavění silnic po povodni, ...). Nejkratší vzdálenost od elektrárny nemusí nutně být nejlepší (pokud je na mapě kratší vzdálenost, ale v cestě je hora, určitě to není nejlepší řešení)

= hledáme pro každá vrchol nejkratší cestu ke stromu

=> neplést si Dijktrou, ten také hledá nejkratší cestu, ale z výchozího vrcholu

Obrázek naznačí rozdíl:



Dijsktra použije hranu 4->5, protože pak bude vždálenost 5 od 1 = 4 (s hranou 3 -> 5 by byla 5) Prim použije hranu 3->5, protože je levnější, než hrana 4 -> 5, a tedy hodnota stromu je 7, což je menší než 8.

Tím je vlastně i popsán algoritmus. Je podobný jako Dijsktra, ale v bodě 4 kontrolujeme pouze váhu hrany, nikoliv součet cesty do předchozího bodu a váhy. Implementace je stejná, řadíme podle vah hran (opět ignorujeme předchozí cesty)

Základy kryptografie

Šifrujeme, protože nechceme, aby třetí osoba věděla, o co se jedná

ochrana citlivých přenášených dat

ochrana před odposlechem

ochrana před neautorizovaným přístupem

....

Šifrování = transformace dat do nečitelné podoby

Dešifrování = zpětný proces rekonstrukce šifrované zprávy pomocí tajného klíče

Kryptografie - věda o tvorbě šifer

Kryptoanalýza - věda o prolamování šifer (nejjednodušší: frekvenční analýza. Anglické texty obsahují velké množství výskytů slova THE, můžeme tedy dedukovat, že nejčastější 3 písmenná slova substitučně šifrovaného textu jsou právě THE (například PQX), a můžeme tyto písmena použít (T->P, H->Q, E->X)

Dělění

Z hlediska zpracování zprávy:

Blokové šifry - pracují s celými bloky dat (obvykle 8-128 bytů)

Proudové šifry (streamové) - pracují s jednotlivými bitu zprávy zvlášť, jsou považovány za méně bezpečné, jsou pomalejší než šifry blokové

Z hlediska šifrování:

Symetrické šifry - odesilatel i příjemce sdílí jedno tajemství (klíč) nutné k šifrování a zašifrování zprávy

Asymetrické šifry - odesilatel a příjemci šifrují a dešifrují zprávu různými klíči, nemusí spolu sdílet žádné tajemství. Nevýhoda: je o několik řádů pomalejší než symetrická kryptografie

Symetricky

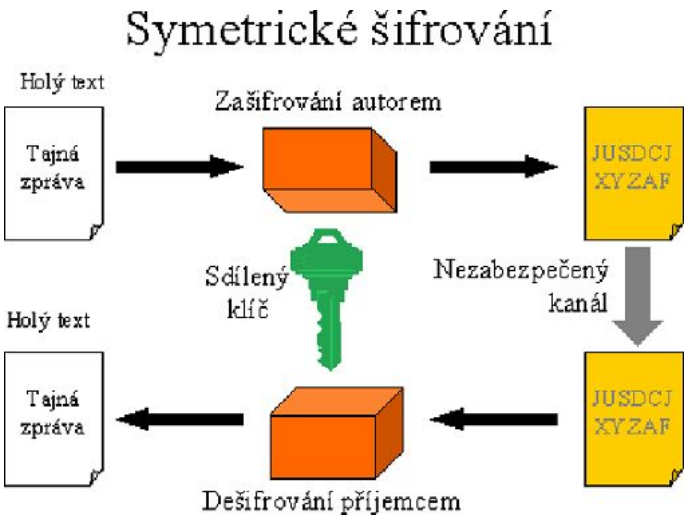
Je nejpoužívanějším typem šifrovacího algoritmu

Používá stejný šifrovací klíč k šifrování i dešifrování- což je jeho největší slabina

Je velmi rychlý a používá se při velkém množství dat

Klíč se musí dostat od odesilatele k adresátovi bezpečným kanálem (cestou), aby adresát mohl zprávu dešifrovat. Často osobní předání. Pokud takový bezpečný kanál existuje, je často jednodušší zprávu nešifrovat a poslat ji rovnou tímto kanálem.

Příklady: AES (ve wifi, je nutné fyzicky zadat klíč do zařízení i do pořítače), A5 (v mobilních telefonech, klíč je na SIM a u uperátora) a DES (nejpoužívanější šifra na světě)



Nesymetricky

Používá jiný klíč k zašifrování a jiný klíč zpátky k dešifrování

První z nich se nazývá veřejný, ostatní ho musejí znát. Druhý klíč se nazývá privátní

Asymetrický šifrovací systém (systém s veřejným klíčem) je založen na principu jednocestné funkce, což jsou operace, které lze snadno provést pouze v jednom směru: ze vstupu lze snadno spočítat výstup, z výstupu však je velmi obtížné nalézt vstup. Nejběžnějším příkladem je například násobení: je velmi snadné vynásobit dvě i velmi velká čísla, avšak rozklad součinu na činitele (tzv. faktorizace) je velmi obtížný. (Na tomto problému je založen např. algoritmus RSA.)