Společná informatika

Automaty a jazyky

Obecně si definujme nějaké pojmy:

- množina symbolů Σ (abeceda)
- slovo je konečná (i prázdná) posloupnost symbolů $s \in \Sigma$, prázdné slovo značíme λ nebo ϵ
- množina všech slov v abecedě je Σ^*
- množina všech neprázdných slov v abecedě je Σ^+
- jazyk je $L \subseteq \Sigma^*$
- operace se slovy $u, v \in L$:
  - zřetězení $u.v$ nebo uv
  - mocnina u^n, kde n je počet opakování
  - délka slova $|u|$
  - počet výskytů $s \in \Sigma$ ve slově u značíme $|u|_s$

Mezi jazyky můžeme dělat různé operace jako sjednocení, průnik, rozdíl a doplněk, funguje to jako operace množin, takže zbytečné definovat.

Další operace s jazyky:

- pozitivní iterace L^+
- obecná iterace L^*
- otočení jazyka L^R
- levý kvocient L podle M, $M \setminus L = \{v | uv \in L, u \in M\}$
- levá derivace L podle w, $\partial_W L = \{w\} \setminus L$

Regulární jazyky

Regulární jazyky jsou ty jazyky, které jsou generovány regulární gramatikou a jsou rozpoznatelné DFA, NFA nebo λ-NFA. Ty popíšeme později.

Regulární gramatiky

Gramatika je čtveřice $G = (V, T, P, S)$, kde:

1. V je množina neterminálů (variables)
2. T je množina terminálů (terminal symbols)
3. S je počáteční symbol $S \in V$
4. P je konečná množina pravidel (produkcí)

Pravidla mají tvar $\beta A \gamma \to \omega, A \in V, \beta, \gamma, \omega \in (V \cup T)^*$. Tedy levá strana vždy obsahuje alespoň jeden neterminální symbol.

Regulární gramatiky obsahují pouze pravidla tohoto typu: $A \to \omega B, A \to \omega, A, B \in V, \omega \in T^*$. Taková gramatika je pravá lineární. Levá lineární gramatika by byla, kdyby neterminál byl nalevo.

Levé a pravé lineární gramatiky zvlášť generují regulární jazyky, ale dohromady jsou již silnější, například jazyk $L = \{0^n 1^n | n \geq 1\}$ není regulární, ale je generován lineární gramatikou ve tvaru $S \to 0S1|01$.

α se (přímo) přepíše na ω, pokud existuje nějaké vhodné pravidlo, které přepsání splní. Posloupnost takových přepsání nazýváme derivací (odvozením). Z derivací lze postavit derivační strom.

Gramatika je víceznačná, pokud existuje aspoň jeden řetězec w takový, že pro něj můžeme najít dva různé derivační stromy takové, že dávají w. Jinak je gramatika jednoznačná.

Konečný automat lze vždy převést na gramatiku typu 3 a obráceně.

Dvě gramatiky jsou ekvivalentní právě tehdy, když generují stejný jazyk.

Deterministický a nedeterministický konečný automat

DFA $A = (Q, \Sigma, \delta, q_0, F)$ sestává z Q, což je konečná množina stavů, abecedy Σ, přechodové funkce $\delta : Q \times \Sigma \to Q$, počátečního stavu $q_0 \in Q$ a neprázdné množiny koncových (přijímajících) stavů $F \subseteq Q$.

Z definice lze nahlédnout, že pro každý stav a každý znak abecedy je jasné, že existuje právě jeden stav, do kterého se dostaneme.

Používá se také rozšířená přechodová funkce:

1. $\delta^*(q, \lambda) = q$
2. $\delta^*(q, wx) = \delta(\delta^*(q, w), x)$ pro $x \in \Sigma, w \in \Sigma^*$

Jazyk rozpoznávaný konečným automatem je takový jazyk, že všechna slova skončí v přijímajícím stavu. Třída jazyků rozpoznatelných konečnými automaty je nazvána regulární jazyky.

Iterační (pumping) lemma pro regulární jazyky: Mějme regulární jazyk L. Pak existuje konstanta $n \in \mathbb{N}$ (závislá na L) tak, že každé $w \in L, |w| \geq n$ můžeme rozdělit na tři části tak, že $w = xyz$ a platí:

1. $y \neq \lambda$
2. $|xy| \leq n$
3. $\forall k \in \mathbb{N}_0$: slovo $xy^kz \in L$

Příklad použití pumping lemmatu: Jazyk slov se stejným počtem 0 a 1 není regulární. Předpokládejme, že regulární je. Vezměme n z pumping lemmatu. Zvolme $w = 0^n1^n \in L$. Pak $|xy| \leq n$, jenže obsahuje samé nuly. Pak by nešlo pumpovat nuly, porušila by se rovnost.

Mějme konečnou abecedu Σ a relaci ekvivalence $\sim$ na Σ^*. Pak:

1. je pravá kongruence, jestliže $\forall u, v, w \in \Sigma^*$: $u \sim v \implies uw \sim vw$
2. je konečného indexu, má-li rozklad $\Sigma^* / \sim$ konečný počet tříd
3. třídu kongruence $\sim$ obsahující slovo u značíme $[u]_\sim$

Myhill–Nerodova věta: L je rozpoznatelný konečným automatem právě tehdy, když existuje pravá kongruence konečného indexu nad Σ^* tak, že L je sjednocením jistých tříd rozkladu $\Sigma^* / \sim$.

Řekněme, že v automatu je stav dosažitelný, jestliže existuje slovo takové, že po přečtení nějakého slova skončíme v daném stavu.

Dva automaty nad stejnou abecedou jsou ekvivalentní, jestliže rozpoznávají stejný jazyk. Dva stavy v automatu jsou ekvivalentní, pokud pro všechna slova z jazyku platí, že $\forall w; \delta^*(p, w) \in F \iff \delta^*(q, w) \in F$.

DFA je redukovaný, pokud nemá nedosažitelné stavy a žádné dva stavy nejsou ekvivalentní. Redukce je jednoduchá, stačí nalézt ekvivalentní stavy a poté je spojit v jeden stav.

Nedeterministický konečný automat (NFA) $A = (Q, \Sigma, \delta, S_0, F)$, kde se změnila pouze přechodová funkce $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ vracející podmnožinu Q a namísto počátečního stavu můžeme mít více počátečních stavů $S_0 \subseteq Q$.

Tedy vlastně se simuluje průběh ve více stavech najednou.

Dá se NFA rozšířit o λ přechody, pak ho nazývejme λ-NFA. To se můžeme libovolně rozhodnout, jestli už přejdeme do dalšího stavu či ne.

Všechny λ-NFA i NFA se dají převést na DFA.

Regulární výrazy

Regulární výrazy $\alpha, \beta \in RegE(\Sigma)$ nad konečnou neprázdnou abecedou $\Sigma = \{x_1, x_2, ..., x_n\}$ a jejich hodnota $L(\alpha)$ jsou definovány induktivně:

1. $L(\lambda) = \{\lambda\}$
2. $L(\emptyset) = \{\} = \emptyset$
3. $L(a) = \{a\}$

Indukce je poté:

1. $L(\alpha + \beta) = L(\alpha) \cup L(\beta)$
2. $L(\alpha\beta) = L(\alpha)L(\beta)$
3. $L(\alpha^*) = L(\alpha)^*$
4. $L((\alpha)) = L(\alpha)$

Každý jazyk reprezentovaný konečným automatem lze zapsat jako regulární
výraz. Každý jazyk popsaný regulárním výrazem můžeme zapsat jako λ-NFA (a tedy i DFA).

Regulární jazyky jssou uzavřeny na všechny operace, tedy sjednocení, průnik, doplněk, homomorfismus, inverzní homomorfismus.

Poznámka: homomorfismus je zjednodušeně řečeno přeznačení abecedy.

Bezkontextové jazyky

Bezkontextové gramatiky, jazyk generovaný gramatikou

Bezkontextová gramatika je gramatika, kde všechna pravidla jsou ve tvaru $A \to \omega, \omega \in (V \cup T)^*$.

Chomského normální forma (ChNF): Všechna pravidla jsou ve tvaru $A \to BC, A \to a$, kde $A, B, C \in V, a \in T$. Do tvaru Chomského normální formy se dají převést všechny bezkontextové gramatiky postupně eliminací zbytečných symbolů, eliminací λ pravidel a jednotkových pravidel.

Důležité je pumping lemma pro bezkontextové jazyky: Mějme bezkontextový jazyk L. Pak existuje $n \in \mathbb{N}$ takové, že každé slovo $z \in L, |z| > n$ lze rozložit na $z = uvwxy$ tak, že:

1. $|vwx| \leq n$
2. $vx \neq \lambda$
3. $\forall i \geq 0, uv^iwx^iy \in L$

Příklad použití: Jazyk $L = \{0^n1^n2^n | n \geq 1\}$ není bezkontextový. Předpokládáme, že bezkontextový je. Z lemmatu vezmeme číslo n, zvolíme $|0^n1^n2^n| > n$. Pumpovací slovo je $|vwx| \leq n$. Vždy tedy lze pumpovat 2 různé symboly, tím by se porušila rovnost počtu symbolů, tedy nemůže jazyk být bezkontextový.

Existuje algoritmus, který ověří, že slovo je generované CFL, tomu se říká Cocke-Younger-Kasami algorithm.

Zásobníkový automat, třída jazyků přijímaných zásobníkovými automaty

Zásobníkové automaty jsou rozšířením λ-NFA automatů s λ přechody. Přidanou věcí je zásobník. Ze zásobníku můžeme číst, přidávat na vrch a odebírat z vrchu zásobníku. Zásobník je neomezeně velký a má vlastní abecedu. Deterministické zásobníkové automaty přijímají jen vlastní podmnožinu bezkontextových jazyků.

Zásobníkový automat (PDA) je $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, kde:

1. Q je konečná množina stavů
2. Σ je neprázdná konečná množina vstupních symbolů
3. Γ je neprázdná konečná zásobníková abeceda
4. δ je přechodová funkce $\delta : Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \to P_{(FIN}(Q \times \Gamma^*))$. Navíc platí, že $\delta(p, a, X) \ni (q, \gamma)$, kde q je nový stav, γ je řetezec zásobníkových symbolů, který nahradí X na vrcholu zásobníku
5. $q_0 \in Q$ je počáteční stav
6. $Z_0 \in \Gamma$ je počáteční zásobníkový symbol
7. F je množina přijímajících stavů, může být nedefinovaná

Situace zásobníkového automatu je trojice (q, w, γ), kde q je stav, w je zbývající vstup a γ je obsah zásobníku. Situace značíme jako ID. Situace mohou tvořit posloupnosti.

Zásobníkový automat může přijímat dvěma způsoby: Buď prázdným zásobníkem, tedy po odebrání všech symbolů ze zásobníku některého ze stavů, nebo koncovým stavem (stejně jako u DFA). Síla těchto dvou druhů automatů je stejná.

Zásobníkové automaty přijímají stejné jazyky jako jsou ty generované bezkontextovou gramatikou.

Deterministický zásobníkový automat (DPDA) definujeme jako PDA, akorát pro každý stav máme nejvýše jednu kombinaci, tedy $\delta(q, a, X)$ je nejvýše jednoprvková $\forall (q, a, X) \in Q \times (\Sigma \cup \{\lambda\}) \times \Gamma$ a je-li $\delta(q, a, X)$ neprázdná pro nějaké $a \in \Sigma$, pak $\delta(q, \lambda, X)$ musí být prázdná.

Pozor na to, že deterministické zásobníkové automaty jsou slabší, než nedeterministické. Dokážou však třeba přijmout jazyk $L = \{wcw^R | w \in (0 + 1)^*\}$.

Bezkontextové jazyky jsou uzavřené na sjednocení, průnik s RL, homomorfismus a inverzní homomorfismus. Pozor, že nejsou uzavřené na průnik a doplněk.

Deterministické CFL jsou uzavřené na průnik s RL a doplněk s inverzním homomorfismem. Nejsou však uzavřené na sjednocení, průnik ani homomorfismus.

Rekurzivně spočetné jazyky

Gramatika typu 0

Gramatiky typu 0 jsou v obecné formě $\alpha \to \beta, \alpha, \beta \in (V \cup T)^*, \alpha$ obsahuje neterminál. Tedy je to ta nejobecnější forma.

Turingův stroj

Turingův stroj (TM) je sedmice $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ se složkami:

1. Q je konečná množina stavů
2. Σ je konečná neprázdná množina vstupních symbolů
3. Γ je konečná množina všech symbolů pro pásku
4. δ je přechodová funkce $(Q - F) \times \Gamma \to Q \times \Gamma \times \{L, R\}$, platí $\delta(q, x) = (p, Y, D)$, kde $q \in (Q - F)$ je aktuální stav, $X \in \Gamma$ je aktuální symbol na pásce, p je nový stav $p \in Q$, $Y \in \Gamma$ symbol pro zapsání do aktuální buňky, $F \subseteq Q$ množina koncových (přijímacích) stavů
5. $q_0 \in Q$ je počáteční stav
6. $B \in \Gamma - \Sigma$ je symbol pro prázdné buňky
7. $F \subseteq Q$ množina koncových stavů

TM zastaví, pokud vstoupí do stavu q s čteným symbolem X a $\delta(q, X)$ není definováno. Předpokládáme, že v přijímacím stavu $q \in F$ TM zastaví, dokud nezastaví, tak nevíme, jestli přijme či nepřijme slovo.

TM rozhoduje jazyk L, pokud $L = L(M)$ a pro každé $w \in \Sigma^*$ stroj nad w zastaví. Jazyky rozhodnutelné TM nazýváme rekurzivní.

Každý rekurzivně spočetný jazyk je typu 0.

TM můžeme rozšířit tak, že přidáme více pásek, případně tak, že přidáme nedeterminismus. Obě taková rozšíření však mají stejnou sílu jako původní TM.

Algoritmicky nerozhodnutelné problémy

Chceme dojít k důkazu nerozhodnutelnosti jazyka dvojic (M, w) takových, že:

1. M je binárně kódovaný Turingův stroj s abecedou { 0, 1 }
2. $w \in \{0, 1\}^*$
3. M nepřijímá vstup w

Diagonální jazyk $L_d = \{w$, TM reprezentovaný jako w takový, že nepřijímá $w\}$

Neexistuje TM přijímající jazyk L_d, protože by to vedlo k paradoxu.

Definujeme univerzální jazyk L_u jakožto množinu binárních řetězců, které kódují pár (M, w), kde M je TM a $w \in L(M)$. TM rozpoznávající L_u se nazývá Univerzální Turingův stroj.

Problémem P myslíme matematicky/informaticky definovanou množinu otázek
kódovatelnou řetězci nad abecedou Σ s odpověďmi $\in \{ano, ne\}$.

Problém je (algoritmicky) rozhodnutelný, pokud existuje Turingův stroj TM
takový, že pro každý vstup $w \in P$ zastaví a navíc přijme právě když $P(w) = ano$
(tj. pro $P(w) = ne$ zastaví v ne–přijímacím stavu).

Problém, který není algoritmicky rozhodnutelný nazýváme nerozhodnutelný
problém.

L_u je rekurzivně spočetný, ale není rekurzivní.

Instance Postova korespondenčního problému (PCP) jsou dva seznamy slov nad abecedou Σ značené $A = w_1, w_2, ..., w_k$ a $B = x_1, x_2, ..., x_k$ stejné délky k. Pro každé i, dvojice (w_i, x_i) se nazývá odpovídající dvojice.
Instance PCP má řešení, pokud existuje posloupnost jednoho či více přirozených čísel $i_1, i_2, ..., i_m$ tak, že $w_{i_1} w_{i_2} ... w_{i_m} = x_{i_1} x_{i_2} ... x_{i_m}$ tj. dostaneme stejné slovo. V tom případě říkáme, že posloupnost $i_1, i_2, ..., i_m$ je řešení. Postův korespondenční problém je: Pro danou instanci PCP, rozhodněte, zda
má řešení.

Je algoritmicky nerozhodnutelné, zda je bezkontextová gramatika víceznačná.

Chomského hierarchie

Gramatikám typu 0 odpovídají rekurzivně spočetné jazyky, jsou rozpoznatelné Turingovými stroji.

Gramatikám typu 1 odpovídají kontextové gramatiky, jsou rozpoznatelné lineárně omezenými automaty.

Obsahují pouze pravidla typu $\alpha A \beta \to \alpha \omega \beta, A \in V, \alpha, \beta \in (V \cup T)^*, \omega \in (V \cup T)^+$ s výjimkou pravidla $S \to \lambda$, ovšem pak se S neobjevuje nikde jinde.

Příklad kontextového jazyka: $L = \{a^n b^n c^n | n \geq 1\}$.

Poznámka: Lineárně omezené automaty jsou stejné jako TM, ovšem jsou limitovány počtem políček, do kterých mohou psát.

Gramatikám typu 2 odpovídají bezkontextové jazyky, jsou rozpoznávany nějakým PDA.

Gramatikám typu 3 odpovídají regulární/pravé lineární jazyky, jsou rozpoznávány DFA/NFA/λ-NFA.

Schopnost zařazení konkrétního jazyka do Chomského hierarchie (zpravidla sestrojení odpovídajícího automatu či gramatiky)

Tohle bych řekl, že je spíš o trénování, pro regulární jazyky je celkem jednoduché vymyslet nějaký DFA/NFA, pro bezkontextové a výše je lepší vymyslet nějakou gramatiku.

Algoritmy a datové stuktury

Časová složitost algoritmů

Časová a prostorová složitost algoritmu

Časová složitost se dá počítat různými způsoby, například počet cyklů algoritmu, počet instrukcí v počítači, ...

Nechť $f, g : \mathbb{N} \to \mathbb{R}$ jsou dvě funkce. Řekneme, že funkce $f(n)$ je třídy $O(g(n))$ (poznámka: správně by se mělo značit $\mathcal{O}$, ale pro jednoduchost značíme i O), jestliže existuje taková kladná reálná konstanta c, že pro skoro všechna n platí $f(n) \leq cg(n)$. Skoro všemi n se myslí, že nerovnost může selhat pro konečně mnoho výjimek, tedy že existuje nějaké přirozené n_0 takové, že nerovnost platí pro všechna $n \geq n_0$. Funkci $g(n)$ se pak říká asymptotický horní odhad funkce $f(n)$.

Zjednodušeně řečeno nezáleží na konstantách. Pozor však na to, že v realitě může být n^3 algoritmus lepší než třeba $5000n^2$, i když asymptoticky je horší.

Mějme dvě funkce $f, g : \mathbb{N} \to \mathbb{R}$. Řekneme, že funkce $f(n)$ je třídy $\Omega(g(n))$, jestliže existuje taková kladná reálná konstanta c, že $f(n) \geq cg(n)$ pro skoro všechna n. Tomu se říká asymptotický dolní odhad.

Řekneme, že funkce $f(n)$ je třídy $\Theta(g(n))$, jestliže $f(n)$ je jak třídy $O(g(n))$, tak třídy $\Omega(g(n))$.

Podobně bychom definovali prostorovou složitost, která prostě počítá, kolik datových buňek náš program zabírá, tedy kolik místa v paměti algoritmus spotřebuje.

Měření velikosti dat

Záleží na tom, jak se na tohle díváme, časově se to měří podle velikosti vstupu - tedy počet dat, která musíme zpracovat, paměťově záleží na použitých typech, ... Nehledal bych v tom vědu, často se to dá prostě aproximovat přes asymptotickou složitost.

Složitost v nejlepším, nejhorším a průměrném případě

Viz definice nahoře, jinak může se stát, že nejlepší/nejhorší/průměrný případ se může asymptoticky lišit. Většinou se bere nejhorší a průměrný případ.

Asymptotická notace

Viz nahoře.

Třídy složitosti

Rozhodovací problém (zkráceně problém) je funkce z množiny $\{0,1\}^*$ všech řetězců nad binární abecedou do množiny $\{0,1\}$.

Jsou-li A, B rozhodovací problémy, říkáme, že A lze převést na B (píšeme $A \to B$) právě tehdy, když existuje funkce $f : \{0,1\}^* \to \{0,1\}^*$ taková, že pro všechna $x \in \{0,1\}^*$ platí $A(x) = B(f(x))$, a navíc lze funkci f spočítat v čase polynomiálním vzhledem k $|x|$. Funkci f říkáme převod nebo také redukce.

Třídy P a NP

P je třída rozhodovacích problémů, které jsou řešitelné v polynomiálním čase. Jinak řečeno, problém L leží v P právě tehdy, když existuje nějaký algoritmus A a polynom f, přičemž pro každý vstup x algoritmus A doběhne v čase nejvýše $f(|x|)$ a vydá výsledek $A(x) = L(x)$.

NP je třída rozhodovacích problémů, v níž problém L leží právě tehdy, pokud existuje nějaký problém $K \in$ P a polynom g, přičemž pro každý vstup x je $L(x) = 1$ právě tehdy, pokud pro nějaký řetězec y délky nejvýše $g(|x|)$ platí $K(x, y) = 1$. Co to znamená? Algoritmus K řeší problém L, ale kromě vstupu x má k dispozici ještě polynomiálně dlouhou nápovědu y. Přitom má platit, že je-li $L(x) = 1$, musí existovat alespoň jedna nápověda, kterou algoritmus K schválí. Pokud ovšem $L(x) = 0$, nesmí ho přesvědčit žádná nápověda.

Třída P leží uvnitř NP. Pokud totiž problém umíme řešit v polynomiálním čase bez nápovědy, tak to zvládneme v polynomiálním čase i s nápovědou. Algoritmus K tedy bude ignorovat nápovědy a odpověď spočítá přímo ze vstupu.

Problém L nazveme NP-těžký, je-li na něj převoditelný každý problém z NP. Pokud navíc L leží v NP, budeme říkat, že L je NP-úplný.

Nevíme, zda je pravda, že P = NP. Jde o jeden z nevyřešených a asi nejznámějších problémů informatiky.

Převoditelnost problémů, NP-těžkost a NP-úplnost

Převoditelnost je nahoře, NP-těžkost a úplnost je definovaná taktéž nahoře.

Obecně se mezi sebou dají převádět problémy třídy P. Poté zvlášť problémy třídy NP.

Příklady NP-úplných problémů a převodů mezi nimi

Logické problémy:

1. SAT: splnitelnost logických formulí v CNF
2. 3-SAT: každá klauzule obsahuje max. 3 literály
3. 3,3-SAT: navíc se každá proměnná vyskytuje nejvýše třikrát
4. SAT pro obecné formule: nejen v CNF
5. Obvodový SAT: splnitelnost booleovského obvodu

Grafové problémy:

1. Nezávislá množina: existuje množina alespoň k vrcholů, mezi nimiž nevede žádná hrana?
2. Klika: existuje úplný podgraf na k vrcholech?
3. Barvení grafu: lze obarvit vrcholy k barvami (přidělit každému vrcholu číslo od 1 do k) tak, aby vrcholy stejné barvy nebyly nikdy spojeny hranou? To je NP-úplné už pro $k = 3$.
4. Hamiltonovská cesta: existuje cesta obsahující všechny vrcholy?
5. Hamiltonovská kružnice: existuje kružnice obsahující všechny vrcholy?
6. 3D-párování: máme tři množiny se zadanými trojicemi; zjistěte, zda existuje taková množina disjunktních trojic, ve které jsou všechny prvky právě jednou? (Striktně vzato, není to grafový problém, ale hypergrafový – hrany nejsou páry, ale trojice.)

Číselné problémy:

1. Součet podmnožiny: má daná množina přirozených čísel podmnožinu s daným součtem?
2. Batoh: jsou dány předměty s váhami a cenami a kapacita batohu, chceme najít co nejdražší podmnožinu předmětů, jejíž váha nepřesáhne kapacitu batohu. Aby se jednalo o rozhodovací problém, ptáme se, zda existuje podmnožina s cenou větší nebo rovnou zadanému číslu.
3. Dva loupežníci: lze rozdělit danou množinu čísel na dvě podmnožiny se stejným součtem?
4. $Ax = 1$ (soustava nula-jedničkových lineárních rovnic): je dána matice $A \in \{0,1\}^{m \times n}$. Existuje vektor $x \in \{0,1\}^n$ takový, že Ax je rovno vektoru samých jedniček?

Problém SAT jsme si už představili v logice, jde o splnitelnost CNF formule. Některé převody jsou jednoduché, třeba 3-SAT je varianta SAT. SAT na 3-SAT se dá převést zavedením nové proměnné. 3-SAT se dá převést na nezávislou množinu jednoduše, tedy znázorněním proměnných jako vrcholy a hrany jsou stejné jako klauzule (s tím, že každý literál je spojený ještě s opačným). Nezávislá množina se dá převést na SAT nějakou šikovnou reprezentací hran. Klika je prakticky identická jako nezávislá množina. 3-SAT se dá převést na 3,3-SAT zavedením nových proměnných. 3,3-SAT se dá převést na 3D-párování.

Metoda rozděl a panuj

Chceme problém rozdělovat na podproblémy, které budou potom tak malé, že je umíme vyřešit samostatně. To se může hodit v různých algoritmech, ale třeba i při paralelním programování.

Typicky se to používá při rekurzivním programování.

Princip rekurzivního dělení problému na podproblémy

Existuje explicitní vzorec, který dokáže určit složitost rekurzivního algoritmu, říkáme mu Master theorem. Popíšeme ho později. Dá se to odhadnout také stromem rekurze, že uděláme sumu přes jednotlivé složitosti.

Celkově jsou různé problémy, které se dají přes rekurzi řešit. Nějaké příklady mohou být třeba Hanojské věže, třídění sléváním (MergeSort), rychlé násobení čísel, hledání k-tého nejmenšího prvku (QuickSelect), rychlé třídění (QuickSort), k-tý nejmenší prvek v lineárním čase apod.

Výpočet složitosti pomocí rekurentních rovnic

Složitost se dá odvodit podle toho, kolik času trávíme na jednotlivých hladinách. Uvedeme si příklad - MergeSort. Jako první si tedy napíšeme čas strávený na první hladině, to je $T(1) = 1$. Obecně na n-té hladině trávíme $T(n) = 2 \cdot T(n/2) + cn$ času. Všimneme si, že pokud bychom dosadili za $T(n/2)$, dostaneme postupem $T(n) = 2^k \cdot T(n/2^k) + kcn$. Pak zvolíme k tak, aby bylo rovno jedné. Tedy $k = \log_2 n$. Dosadíem do vzorce a dostaneme $T(n) = 2^{\log_2 n} \cdot T(1) + log_2 n \cdot cn = n + cn \log_2 n$. Časová složitost je tedy $\Theta(n \log n)$.

Kromě rekurentních rovnic se to dá řešit i právě stromem rekurze.

Master theorem (kuchařková věta) (bez důkazu)

Aby se nám složitost dobře počítala, dá se použít tzv. Master theorem pro výpočet složitosti. Mějme tedy obecně zadanou rekurenci:

$T(1) = 1, T(n) = a \cdot T(n/b) + \Theta(n^c)$. Dokazovat si to tedy nebudeme, každopádně platí, že výsledek závisí na kvocientu $q = a/b^c$. Potom platí:

1. $q = 1 \implies T(n) = \Theta(n^c \log n)$
2. $q < 1 \implies T(n) = \Theta(n^c)$
3. $q > 1 \implies T(n) = \Theta(n^{\log_b a})$

Samozřejmě musíme ošetřit to, aby konstanty byly rozumné, logicky $a \geq 1, b > 1, c \geq 0$.

Aplikace

Některé aplikace jsme si už zmínili nahoře.

Mergesort

Algoritmus MergeSort (rekurzivní třídění sléváním)
Vstup: Posloupnost $a_1, ..., a_n$ k setřídění

1. Pokud $n = 1$, vrátíme jako výsledek $b_1 = a_1$ a skončíme.
2. $x_1, ..., x_{n/2} \leftarrow MergeSort(a_1, ..., a_{n/2})$
3. $y_1, ..., y_{n/2} \leftarrow MergeSort(a_{n/2} + 1, ..., a_n)$
4. $b_1, ..., b_n \leftarrow Merge(x_1, ..., x_{n/2}; y_1, ..., y_{n/2})$
   Výstup: Setříděná posloupnost $b_1, ..., b_n$

Merge je procedura slévání. To je jednoduché, začneme se dvěma setřídenými posloupnostmi, pointery jsou na začátku každé z nich. Postupně porovnáváme prvky po dvou a vždy posuneme jeden pointer podle toho, který prvek jsme zpracovali. To je lineární procedura.

Buď z analýzy přes Master theorem nebo jinak nahlédneme, že celková časová složitost tohoto algoritmu je $\Theta(n \log n)$. Obecně se dělá MergeSort tak, že potřebuje nějakou lineární paměť navíc, ale dá se udělat i in-place, jen by se to nedělalo rekurzivně.

Násobení dlouhých čísel

Mějme n-ciferná čísla X, Y, která chceme vynásobit. Rozdělíme je na horních $n/2$
a dolních $n/2$ cifer (pro jednoduchost opět předpokládejme, že n je mocnina dvojky). Platí tedy:

1. $X = A \cdot 10^{n/2} + B$
2. $Y = C \cdot 10^{n/2} + D$
   pro nějaká $(n/2)$-ciferná čísla A, B, C, D. Hledaný součin XY můžeme zapsat takto: $XY = AC \cdot 10^n + (AD + BC) \cdot 10^{n/2} + BD$.

To by nám však nestačilo, protože by násobení bylo stále kvadratické. Proto můžeme ještě formuli upravit: $XY = AC \cdot 10^n + ((A + B)(C + D) - AC - BD) \cdot 10^{n/2} + BD$.

Tím se změní složitost na hladině na $T(n) = 3 \cdot T(n/2) + \Theta(n)$. To podle Master theoremu sníží složitost na $O(n^{\log_2(3)-1})$, což už je lepší a vyplatí se to.

Binarní vyhledávací stromy

Definice vyhledávacího stromu

Strom nazveme binární, pokud je zakořeněný a každý vrchol má nejvýše dva syny, u nichž rozlišujeme, který je levý a který pravý. Stromy si můžeme představit stejně jako v diskrétce, akorát tady máme jeden kořen a postupně hrany vedou dolů.

Pro vrchol v binárního stromu T značíme:

1. $T(v)$ - podstrom obsahující vrchol v a všechny jeho potomky
2. $l(v), r(v)$ - levý a pravý syn vrcholu v
3. $L(v), R(v)$ - levý a pravý podstrom vrcholu v, tedy $T(l(v))aT(r(v))$
4. $h(v)$ - hloubka stromu $T(v)$, čili maximum z délek cest z v do listů

Pokud vrchol nemá levého syna, položíme $l(v) = r(v) = \emptyset$. Pak se hodí dodefinovat, že $T(\emptyset)$ je prázdný strom a $h(\emptyset) = -1$.

Binární vyhledávací strom (BVS) je binární strom, jehož každému vrcholu v přiřadíme unikátní klíč $k(v)$ z univerza. Přitom musí pro každý vrchol v platít:

1. Kdykoliv $a \in L(v)$, pak $k(a) < k(v)$
2. Kdykoliv $b \in R(v)$, pak $k(b) > k(v)$

Tedy vrchol v odděluje klíče v levém a pravém podstromu.

Operace s nevyvažovanými stromy

Základní operací je projít strom. To uděláme rekurzivně, jde o in-order průchod. Tedy nejprve voláme rekurzivně levou stranu, pak vypíšeme sebe, pak voláme pravou stranu.

Další operací je najít klíč ve stromu. Opět to jde přímočaře rekurzivně - pokud je hodnota aktuálního klíče moc velká, zavoláme proceduru na levou stranu, pokud moc malá, tak na pravou stranu. Pokud se klíč rovná hledanému, pak jsme vrchol nalezli.

Minimum nalezneme jednoduše - půjdeme stále doleva. Obdobně maximum.

Vkládání do stromu funguje jako vyhledávání. Pokud bychom měli prvek vyhledat, přejdeme do neexistujícího vrcholu. Místo toho, abychom oznámili neexistenci prvku, jednoduše prvek přidáme.

Mazání prvku je složitější - pokud mažeme list, je to jednoduché, tím se stávající struktura neporuší. Pokud to list nebyl, tak musíme rozlišit, jestli mažeme zleva nebo zprava a dát do vrcholu nejbližší možnou hodnotu. Dá se to celkem jednoduše rozmyslet.

Pozor na to, že tyto operace mají sice v průměrném případě složitost $\Theta(\log n)$, ale v nejhorším případě $\Theta(n)$ - stačí si představit příklad, že začneme s číslem 1 a vždy přidáváme prvek o 1 větší, pak budeme mít strom jako jednu dlouhou větev doprava.

AVL stromy (definice)

Definice: Binární vyhledávací strom nazveme dokonale vyvážený, pokud pro každý jeho
vrchol v platí $||L(v)| - |P(v)|| \leq 1$. Jinými slovy počet vrcholů levého a pravého podstromu se smí lišit nejvýše o 1.

Dokonale vyvážený strom má tedy hloubku $\log_2 n$.

Binární vyhledávací strom nazveme hloubkově vyvážený, pokud pro každý jeho
vrchol v platí $h(l(v)) - h(r(v)) \leq 1$. Jinými slovy, hloubka levého a pravého podstromu se vždy liší nejvýše o jedna.

Stromy, které jsou hloubkově vyvážené, se nazývají AVL stromy.

AVL stromy mají logaritmickou hloubku, tedy $\Theta(\log n)$.

AVL stromy se musí vyvažovat tzv. rotacemi stromu, tedy nějaké operace s vrcholy, aby zůstal hloubkově vyvážený a zároveň stále splňoval definici BVS. Strom se musí vyvážit vždy, když přidáme nebo smažeme vrchol.

Operace přidání a smazání vrcholu u AVL nejsou tak jednoduché, protože mohou nastat různé případy prohlubování, někdy se vyvažovat vůbec nemusí. Do detailu to rozepisovat nebudu.

Třídění

Primitivní třídicí algoritmy (Bubblesort, Insertsort)

Jako první si uvedeme bublinkové třídění (Bubblesort). Jeho základem je myšlenka nechat stoupat větší prvky v poli podobně, jako stoupají bublinky v limonádě. V algoritmu budeme opakovaně procházet celé pole. Jeden průchod postupně porovná všechny dvojice sousedních prvků $P[i]$ a $P[i+1]$. Pokud dvojice není správně uspořádaná (tedy $P[i] > P[i+1]$), prvky prohodíme. V opačném případě necháme dvojici na pokoji. Menší prvky se nám tak posunou blíže k začátku pole, zatímco větší prvky „bublají“ na jeho konec. Pokaždé, když pole projdeme celé, začneme znovu od začátku. Tyto průchody opakujeme, dokud dochází k prohazování prvků. V okamžiku, kdy výměny ustanou, je pole setříděné. Technicky tedy jde jen o 2 for loopy. Časová složitost je tedy $O(n^2)$.

Insertsort neboli třídění přímým vkládáním funguje takto: Udržujeme dvě části pole - na začátku leží setříděné prvky a v druhé části pak zbývající nesetříděné. V každém kroku vezmeme jeden prvek z nesetříděné části a vložíme jej na správné místo v části setříděné. Složitost je stejná, tedy kvadratická, protože opakovaně procházíme setříděnou posloupnost.

Quicksort

Začínáme s nesetříděným polem hodnot. Pro 1 prvek skončíme, to je zřejmé. Některý z prvků zvolíme jako pivota (je dobré to volit například náhodně). Poté rozdělíme posloupnost následovně na 3 části: prvky posloupnosti menší než pivot, prvky posloupnosti rovny pivotu, prvky posloupnosti větší než pivot. Rekurzivně setřídíme levou a pravou část stejným algoritmem a slepíme za sebe 3 části dohromady. Dostaneme setříděnou posloupnost.

Při dobré volbě pivota jsme schopni třídit průměrně v čase $O(n \log n)$, v nejhorším případě v $O(n^2)$. Pokud volíme jako pivot nějaké mediány nebo skoromediány, pak se časová složitost zlepší, ovšem v kontrastu samozřejmě nějakou dobu zabere i zjistit, který prvek je meidánem apod.

Dolní odhad složitosti porovnávacích třídicích algoritmů

Nemůžeme třídit lépe než v čase $\Theta(n \log n)$, pokud tedy nepočítáme takové algoritmy jako CountingSort apod. (ty nevyužívají porovnávání). Vyhledávat lze totiž nejlépe v čase $\log n$ binárně, na každý prvek se musíme podívat alespoň jednou.

Grafové algoritmy

Prohledávání do šířky a do hloubky

BFS (prohledávání do šířky)

Základním stavebním kamenem většiny grafových algoritmů je nějaký způsob prohledávání grafu. Tím myslíme postupné procházení grafu po hranách od určitého počátečního vrcholu. Možných způsobů prohledávání je víc, zatím ukážeme ten nejjednodušší: prohledávání do šířky. Často se mu říká zkratkou BFS z anglického breadth-first search.

Na vstupu dostaneme konečný orientovaný graf a počáteční vrchol v_0. Postupně nacházíme následníky vrcholu v_0, pak následníky těchto následníků, a tak dále, až objevíme všechny vrcholy, do nichž se dá z v_0 dojít po hranách. Obrazně řečeno, do grafu nalijeme vodu a sledujeme, jak postupuje vlna.

Během výpočtu rozlišujeme 3 stavy vrcholů:

1. nenalezené - zatím jsme je nepotkali na cestě grafem
2. otevřené - už jsme je viděli, ale ještě jsme neprozkoumali všechny hrany, které z něj vedou
3. uzavřené - už jsme je viděli a prozkoumali jsme i hrany

Předpokladem je, že všechny vrcholy až na první považujeme za uzavřené. Funguje to tak, že do fronty postupně dáváme vrcholy, jak je potkáme podle hran a vždy z fronty odebereme vrchol a zařadíme jeho sousedy, jsou-li nenalezené.

BFS se vždy zastaví a korektně najde nejkratší (neohodnocenou) cestu mezi dvěma vrcholy. Časová složitost algoritmu je $O(|V| + |E|)$, prostorovou taky.

Pozor na různé reprezentace sousednosti, například u matice sousednosti by to bylo až $O(n^2)$, lepší je seznam sousedů.

DFS (prohledávání do hloubky)

Dalším důležitým algoritmem k procházení grafů je prohledávání do hloubky, anglicky depth-first search čili DFS. Je založeno na podobném principu jako BFS, ale vrcholy zpracovává rekurzivně: kdykoliv narazí na dosud nenalezený vrchol, otevře ho, zavolá se rekurzivně na všechny jeho dosud nenalezené následníky, načež původní vrchol zavře a vrátí se z rekurze. Mimochodem se dá udělat i přes zásobník.

DFS je vhodná ve chvíli, kdy chceme dostat třeba libovolnou cestu, případně je nám jedno, jaké bude řešení, ale chceme ho rychle. Má časovou složitost $O(|V| + |E|)$, prostorovou taky.

Hrany v DFS se dají klasifikovat. Mohou nastat následující případy:

1. stromová hrana - na DFS cestě
2. dopředná hrana - na cestě, ovšem ne přímo (už jsme vrchol nalezli)
3. zpětná hrana - na cestě, ovšem opačným směrem
4. příčná hrana - už jsme tam v rekurzi byli, takže prostě napříč

Topologické třídění orientovaných grafů

Častým případem orientovaných grafů jsou acyklické orientované grafy neboli DAGy (z anglického directed acyclic graph). Pro ně umíme řadu problémů vyřešit efektivněji než pro obecné grafy. Mnohdy k tomu využíváme existenci topologického pořadí vrcholů.

Jak přijdeme na to, že v grafu je cyklus? Použijeme DFS. Pokud DFS nalezne zpětnou hranu, pak graf není DAG.

DAG lze uspořádat tak, aby všechny hrany vedly po směru tohoto uspořádání (to je třeba zleva doprava). Orientovaný graf má topologické uspořádání právě tehdy, když je to DAG. V grafu s cykly by nebylo možné hrany seřadit, ale v acyklickém to jde.

V každém neprázdném DAGu existuje vrchol, do kterého nevede žádná hrana (zdroj).

Pořadí, v němž DFS opouští vrcholy, je opačné topologické.

Nejkratší cesty v ohodnocených grafech (Dijkstrův a Bellmanův-Fordův algoritmus)

Dijkstrův algoritmus používáme ve chvíli, kdy máme nezáporně ohodnocené hrany grafu. Mějme tedy orientovaný graf, jehož hrany jsou ohodnocené celými kladnými čísly. Každou hranu podrozdělíme na tolik jednotkových hran, jaká byla cena cesty (ohodnocení hrany). Pro každý vrchol si pořídíme budík, tedy jakmile k vrcholu zamíří vlna, nastavíme jeho budík na čas, kdy do něj má vlna dorazit, uvažujeme ten nejnižší. Dokud existují nějaké otevřené vrcholy, vybíráme vždy otevřený vrchol s nejmenší hodnotou budíku a spočítáme budíky pro všechny jeho sousedy. Dokud existuje otevřený vrchol, nezastavujeme. Inicializace trvá $O(n)$, Dijkstra funguje celkem v čase $O(n^2)$. Dá se zlepšit, pokud pracujeme s binární haldou, pak bychom měli $O((n + m) \log n)$.

Dijkstrův algoritmus na grafu bez záporných hran uzavírá všechny dosažitelné vrcholy v pořadí podle rostoucí vzdálenosti od počátku (každý právě jednou). V okamžiku uzavření je ohodnocení rovno této vzdálenosti a dále se nezmění.

Bellman-Fordův algoritmus používáme ve chvíli, kdy máme měřit vzdálenost v grafech se zápornými hranami. Rozdíl je v tom, že oproti Dijkstrovi budeme uzavírat nejstarší z otevřených vrcholů namísto toho s nejnižší hodnotou budíku. Pokud graf neobsahuje záporné cykly, tak zastaví. V grafu bez záporných cyklů nalezne Bellman-Ford všechny vzdálenosti z v_0 v čase $O(mn)$.

Minimální kostra grafu (Jarníkův a Borůvkův algoritmus)

Potřebujeme si zadefinovat pár pojmů, které nám pomůžou pochopit, co chceme.

Nechť $G = (V, E)$ je souvislý neorientovaný graf a $w : E \to \mathbb{R}$ váhová funkce, která přiřazuje hranám čísla – jejich váhy. n, m nechť jako obvykle značí počet vrcholů a hran grafu G. Váhovou funkci můžeme přirozeně rozšířit na podgrafy: Váha $w(H)$ podgrafu $H \subseteq G$ je součet vah jeho hran. Kostra grafu G je podgraf, který obsahuje všechny vrcholy a je to strom. Kostra je minimální, pokud má mezi všemi kostrami nejmenší váhu.

Graf může mít více minimálních koster, pokud jsou váhy všech hran navzájem různé, pak existuje pouze jedna minimální kostra.

Jarníkův algoritmus: Mějme souvislý graf s unikátními vahami. Pak začneme s libovolným vrcholem grafu a grafem T obsahující pouze tento vrchol. Dokud existuje hrana taková, že jeden z vrcholů leží v T a druhý tam neleží, přidáme nejlehčí z nich. Algoritmus běží v čase $O(mn)$.

Borůvkův algoritmus: Je to taková paralelní verze Jarníkova algoritmu. Začínáme s nesouvislými vrcholy. Dokud T není souvislý, rozložíme ho na komponenty souvislosti. Pro každou komponentu nalezneme takového souseda s nejlehčí hranou, se kterým ještě nebyl spojený. Tímto rozšiřujeme kostru po mocninách dvojky v každé iteraci. Lze tedy nahlédnout, že takový algoritmus pracuje v čase $m \log n$.

Toky v sítích (Ford-Fulkerson algoritmus)

Tok v síti jsme si už definovali u teorie grafů v matematice. Pro připomenutí máme orientovaný graf s nějakými kapacitami, zdrojem, stokem a nějakou funkcí, které říkáme tok. Pro tok platí, že je shora omezen kapacitami a platí Kirchhoffův zákon, tedy co přiteče do vrcholu, to z něj odteče pro všechny kromě zdroje a stoku.

Navíc můžeme ještě definovat přítok, odtok a přebytek z vrcholu:

1. $f^+(v) = \sum_{u:uv \in E} f(uv)$ - přítok
2. $f^-(v) = \sum_{u:vu \in E} f(uv)$ - odtok
3. $f^\Delta(v) = f^+(v) - f^-(v)$ - přebytek

Velikost toku se značí $|f|$ a je to vlastně přebytek stoku.

Ford-Fulkerson: Nejjednodušší z algoritmů na hledání maximálního toku je založen na prosté myšlence: začneme s nulovým tokem a postupně ho vylepšujeme, až dostaneme maximální tok. Uvažujme, jak by vylepšování mohlo probíhat. Nechť existuje cesta P ze z do s taková, že po všech jejích hranách teče méně, než dovolují kapacity. Takové cestě budeme říkat zlepšující, protože po ní můžeme tok zvětšit. Zvolíme $\epsilon = \min_{e \in P}(c(e) - f(e))$. Poté tok na každé hraně, která je ovlivněná tímto zlepšením může být zlepšena. Pozor na to, že to ještě nestačí, mohl by vzniknout tok, který není maximální.

Definice: Rezerva hrany uv je číslo $r(uv) = c(uv) - f(uv) + f(vu)$. Hraně s nulovou rezervou budeme říkat nasycená, hraně s kladnou rezervou nenasycená. O cestě řekneme, že je nasycená, pokud je nasycená alespoň jedna její hrana; jinak mají všechny hrany kladné rezervy a cesta je nenasycená. Roli zlepšujících cest tedy budou hrát nenasycené cesty. Budeme je opakovaně hledat a tok po nich zlepšovat.

Dokud tedy existuje nenasycená cesta P ze z do s, opakujeme, že spočítáme rezervu celé cesty, vybereme minimum, pro všechny hrany spočítáme, kolik můžeme odečíst v protisměru a zbytek přičteme po směru. Dostaneme maximální tok.

Side note: Velikost maximálního toku je stejná jako velikost minimálního řezu. O tom však více v té matematické části.

Programovací jazyky

Viz specializace.

Architektura počítačů a operačních systémů

Viz specializace.