1. Převody problémů a NP-úplnost

Všechny úlohy, které jsme zatím potkali, jsme uměli vyřešit algoritmem s polynomiální časovou složitostí. V prvním přiblížení můžeme říci, že polynomialita docela dobře vystihuje praktickou použitelnost algoritmu.⟨1⟩

Existují tedy polynomiální algoritmy pro všechny úlohy? Zajisté ne, jsou dokonce i takové úlohy, jež žádným algoritmem vyřešit nelze. Ale i mezi těmi algoritmicky řešitelnými potkáme spoustu úloh, pro které zatím žádný polynomiální algoritmus není známý (ale ani neumíme dokázat, že neexistuje). Takové úlohy jsou překvapivě časté, proto se na této přednášce podíváme na několik typických příkladů.

Navíc uvidíme, že ačkoliv tyto úlohy neumíme efektivně řešit, jde mezi nimi nalézt zajímavé vztahy a pomocí těchto vztahů obtížnost problémů vzájemně porovnávat. Z těchto úvah vyrůstá i skutečná teorie složitosti se svými hierarchiemi složitostních tříd. Můžete tedy následující kapitolu považovat za malou ochutnávku toho, jak lze k třídám problémů přistupovat.

1.1. Rozhodovací problémy a převody mezi nimi

Aby se nám teorie příliš nerozkošatila, omezíme své úvahy na rozhodovací problémy. To jsou úlohy, jejichž výstupem je jediný bit – máme tedy rozhodnout, zda vstup má či nemá určenou vlastnost. Vstup přitom budeme reprezentovat řetězcem nul a jedniček – libovolnou jinou „rozumnou“ reprezentaci dokážeme na tyto řetězce převést v polynomiálním čase. Formálněji:

Definice: *Rozhodovací problém* (zkráceně *problém*) je funkce z množiny $\{0,1\}^*$ všech řetězců nad binární abecedou do množiny $\{0,1\}$.⟨2⟩

Příklad problému: *Bipartitní párování* – je dán bipartitní graf a číslo $k \in \mathbb{N}$. Zapíšeme je pomocí řetězce bitů: graf třeba maticí sousednosti, číslo dvojkově (detaily kódování budeme nadále vynechávat). Máme odpovědět, zda v zadaném grafu existuje párování, které obsahuje alespoň k hran.

(Otázka, zda existuje párování o právě k hranách, je ekvivalentní, protože můžeme libovolnou hranu z párování vypustit a bude to stále párování.)

Odbočka: Často nás samozřejmě nejen zajímá, zda párování existuje, ale také chceme nějaké konkrétní najít. I to jde pomocí rozhodovací verze problému snadno zařídit. Podobný postup funguje pro mnoho dalších problémů.

⟨1⟩ Jistě vás napadne spousta protipříkladů, jako třeba algoritmus se složitostí $\mathcal{O}(1.001^n)$, který nejspíš je použitelný, ačkoliv není polynomiální, a jiný se složitostí $\mathcal{O}(n^{100})$, u kterého je tomu naopak. Ukazuje se, že tyto případy jsou velmi řídké, takže u většiny problémů náš zjednodušený pohled funguje překvapivě dobře.

⟨2⟩ Ekvivalentně bychom se na problém mohli také dívat jako na nějakou množinu $A \subseteq \{0,1\}^*$ vstupů, na něž je odpověď 1. Tento přístup mají rádi v teorii automatů.

Mějme černou skříňku (fungující v polynomiálním čase), která odpoví, zda daný graf má nebo nemá párování o k hranách. Odebereme z grafu libovolnou hranu a zeptáme se, jestli i tento nový graf má párovaní velikosti k. Když má, pak tato hrana nebyla pro existenci párování potřebná, a tak ji odstraníme. Když naopak nemá (hrana patří do každého párování požadované velikosti), tak si danou hranu poznamenáme a odebereme nejen ji a její vrcholy, ale také hrany, které do těchto vrcholů vedly. Toto je korektní krok, protože v původním grafu tyto vrcholy byly navzájem spárované, a tedy nemohou být spárované s žádnými jinými vrcholy. Na nový graf aplikujeme znovu tentýž postup. Výsledkem je množina hran, které patří do hledaného párování. Hran, a tedy i iterací našeho algoritmu, je polynomiálně mnoho a skříňka funguje v polynomiálním čase, takže celý algoritmus je polynomiální.⟨3⟩

Zpět z odbočky: Jak párovací problém vyřešit? Věrni matfyzáckým vtipům, převedeme ho na nějaký, který už vyřešit umíme. To už ostatně umíme – na toky v sítích. Pokaždé, když se ptáme na existenci párování velikosti alespoň k v nějakém bipartitním grafu, umíme efektivně sestrojit nějakou síť a zeptat se, zda v této síti existuje tok velikosti alespoň k. Chceme tedy přeložit vstup jednoho problému na vstup jiného problému tak, aby odpověď zůstala stejná.

Takovéto převody mezi problémy můžeme definovat i obecněji:

Definice: Jsou-li A, B rozhodovací problémy, říkáme, že A lze *převést* (neboli *redukovat*) 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 polynomiálním čase.

Pozorování: $A \to B$ také znamená, že problém B je alespoň tak těžký jako problém A. Tím myslíme, že kdykoliv umíme vyřešit B, je vyřešit A nejvýše polynomiálně obtížnější. Speciálně platí:

Lemma: Pokud $A \to B$ a B lze řešit v polynomiálním čase, pak i A lze řešit v polynomiálním čase.

Důkaz: Nechť existuje algoritmus řešící problém B v čase $\mathcal{O}(b^k)$, kde b je délka vstupu tohoto problému a k konstanta. Mějme dále funkci f převádějící A na B v čase $\mathcal{O}(a^\ell)$ pro vstup délky a. Chceme-li nyní spočítat $A(x)$ pro nějaký vstup x délky a, spočítáme nejprve $f(x)$. To bude trvat $\mathcal{O}(a^\ell)$ a vyjde výstup délky taktéž $\mathcal{O}(a^\ell)$ – delší bychom v daném čase ani nestihli vypsat. Tento vstup pak předáme algoritmu pro problém B, který nad ním stráví čas $\mathcal{O}((a^\ell)^k) = \mathcal{O}(a^{k\ell})$. Celkový čas výpočtu tedy činí $\mathcal{O}(a^\ell + a^{k\ell})$, což je polynom v délce původního vstupu. □

Relace převoditelnosti jistým způsobem porovnává problémy podle obtížnosti. Nabízí se představa, že se jedná o uspořádání na množině všech problémů. Je tomu doopravdy tak?

Pozorování: O relaci „→“ platí:

⟨3⟩ Zde se skrývá hlavní důvod, proč informatici mají tak rádi polynomiální algoritmy. Třída všech polynomů je totiž nejmenší třídou funkcí, která obsahuje všechny „základní“ funkce (konstanty, n, n^2, …) a je uzavřená na sčítání, odčítání, násobení i skládání funkcí.

- *Je reflexivní* ($A \to A$) – úlohu můžeme převést na tutéž identickým zobrazením.
- *Je tranzitivní* ($A \to B \wedge B \to C \Rightarrow A \to C$) – pokud funkce f převádí A na B a funkce g převádí B na C, pak funkce $g \circ f$ převádí A na C. Složení dvou polynomiálně vyčíslitelných funkcí je zase polynomiálně vyčíslitelná funkce, jak už jsme zpozorovali v důkazu předchozího lemmatu.
- *Není antisymetrická* – například problémy „na vstupu je řetězec začínající nulou“ a „na vstupu je řetězec končící nulou“ lze mezi sebou převádět oběma směry.
- Existují *navzájem nepřevoditelné problémy* – třeba mezi problémy „na každý vstup odpověz 0“ a „na každý vstup odpověz 1“ nemůže existovat převod ani jedním směrem.

Relacím, které jsou reflexivní a tranzitivní, ale obecně nesplňují antisymetrii, se říká *kvaziuspořádání*. Převoditelnost je tedy částečné kvaziuspořádání na množině všech problémů.⟨4⟩

Nyní se již podíváme na příklady několika problémů, které se obecně považují za těžké. Uvidíme, že každý z nich je možné převést na všechny ostatní, takže z našeho „polynomiálního“ pohledu jsou stejně obtížné.

Problém SAT – splnitelnost (satisfiability) logických formulí v CNF

Mějme nějakou logickou formuli s proměnnými a logickými spojkami. Zajímá nás, je-li tato formule *splnitelná,* tedy zda lze za proměnné dosadit 0 a 1 tak, aby formule dala výsledek 1 (byla *splněna*).

Zaměříme se na formule ve speciálním tvaru, v takzvané *konjunktivní normální formě (CNF):*

- *formule* je složena z jednotlivých *klauzulí* oddělených spojkou $\wedge$,
- každá *klauzule* je složená z *literálů* oddělených $\vee$,
- každý *literál* je buďto proměnná nebo její negace.

Vstup problému: Formule ψ v konjunktivní normální formě.

Výstup problému: Existuje-li dosazení 0 a 1 za proměnné tak, aby $\psi(\ldots) = 1$.

Příklad: Formule $(x \vee y \vee z) \wedge (\neg x \vee y \vee z) \wedge (x \vee \neg y \vee z) \wedge (x \vee y \vee \neg z)$ je splnitelná, stačí nastavit například $x = y = z = 1$ (jaká jsou ostatní splňující ohodnocení?). Naproti tomu formule $(x \vee y) \wedge (x \vee \neg y) \wedge \neg x$ splnitelná není, což snadno ověříme třeba vyzkoušením všech čtyř možných ohodnocení.

⟨4⟩ Kdybychom z něj chtěli vyrobit opravdové (byť částečné) uspořádání, mohli bychom definovat ekvivalenci $A \sim B \equiv A \to B \wedge B \to A$ a relaci převoditelnosti zavést jen na třídách této ekvivalence. Taková převoditelnost by už byla slabě antisymetrická. To je v matematice dost běžný trik, říká se mu *faktorizace* kvaziuspořádání.

Poznámka: Co kdybychom chtěli zjistit, zda je splnitelná nějaká formule, která není v CNF? V logice se dokazuje, že ke každé formuli lze najít ekvivalentní formuli v CNF, ale při tom se bohužel formule může až exponenciálně prodloužit. Později ukážeme, že pro každou formuli χ existuje nějaká formule χ' v CNF, která je splnitelná právě tehdy, když je χ splnitelná. Formule χ' přitom bude dlouhá $\mathcal{O}(|\chi|)$, ale budou v ní nějaké nové proměnné.

Problém 3-SAT – splnitelnost formulí s krátkými klauzulemi

Pro SAT zatím není známý žádný polynomiální algoritmus. Co kdybychom zkusili problém trochu zjednodušit a uvažovat pouze formule ve speciálním tvaru?

Povolíme tedy na vstupu pouze takové formule v CNF, jejichž každá klauzule obsahuje nejvýše tři literály. Ukážeme, že tento problém je stejně těžký jako původní SAT.

Převod 3-SATu na SAT: Jelikož 3-SAT je speciálním případem SATu, poslouží tu jako převodní funkce identické zobrazení.

Převod SATu na 3-SAT: Nechť se ve formuli vyskytuje nějaká „špatná“ klauzule o $k > 3$ literálech. Můžeme ji zapsat ve tvaru $(\alpha \vee \beta)$, kde α obsahuje 2 literály a β $k - 2$ literálů. Pořídíme si novou proměnnou x a klauzuli nahradíme dvěma novými $(\alpha \vee x)$ a $(\beta \vee \neg x)$. První z nich obsahuje 3 literály, tedy je dobrá. Druhá má $k - 1$ literálů, takže může být stále špatná, ale aspoň je kratší, takže můžeme postup opakovat.

Takto postupně nahradíme všechny špatné klauzule dobrými, což bude trvat nejvýše polynomiálně dlouho: klauzuli délky k rozebereme po $k-3$ krocích, špatných klauzulí je lineárně s délkou formule.

Zbývá ukázat, že nová formule je splnitelná právě tehdy, byla-li splnitelná formule původní. K tomu stačí ukázat, že každý jednotlivý krok převodu splnitelnost zachovává.

Pokud původní formule byla splnitelná, uvažme nějaké splňující ohodnocení proměnných. Ukážeme, že vždy můžeme novou proměnnou x nastavit tak, aby vzniklo splňující ohodnocení nové formule. Víme, že klauzule $(\alpha \vee \beta)$ byla splněna. Proto v daném ohodnocení:

- Buďto $\alpha = 1$. Pak položíme $x = 0$, takže $(\alpha \vee x)$ bude splněna díky α a $(\beta \vee \neg x)$ díky x.
- Anebo $\alpha = 0$, a tedy $\beta = 1$. Pak položíme $x = 1$, čímž bude $(\alpha \vee x)$ splněna díky x, zatímco $(\beta \vee \neg x)$ díky β.

Ostatní klauzule budou stále splněny.

V opačném směru: pokud dostaneme splňující ohodnocení nové formule, umíme z něj získat splňující ohodnocení formule původní. Ukážeme, že stačí zapomenout proměnnou x. Všechny klauzule, kterých se naše transformace netýká, jsou nadále splněné. Co klauzule $(\alpha \vee \beta)$?

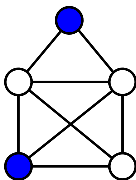
- Buďto $x = 0$, pak musí být $(\alpha \vee x)$ splněna díky α, takže $(\alpha \vee \beta)$ je také splněna díky α.

- Anebo $x = 1$, pak musí být $(\beta \vee \neg x)$ splněna díky β, takže i $(\alpha \vee \beta)$ je splněna.

Tím je převod hotov, SAT a 3-SAT jsou tedy ekvivalentní.

Problém NzMna – nezávislá množina vrcholů v grafu

Definice: Množina vrcholů grafu je *nezávislá,* pokud žádné dva vrcholy ležící v této množině nejsou spojeny hranou. (Jinými slovy nezávislá množina indukuje podgraf bez hran.)



Obr. 1.1: Příklad nezávislé množiny

Na samotnou existenci nezávislé množiny se nemá smysl ptát – prázdná množina či libovolný jeden vrchol jsou vždy nezávislé. Zajímavé ale je, jestli graf obsahuje dostatečně velkou nezávislou množinu.

Vstup problému: Neorientovaný graf G a číslo $k \in \mathbb{N}$.

Výstup problému: Zda existuje nezávislá množina $A \subseteq V(G)$ velikosti alespoň k.

Převod 3-SAT → NzMna: Dostaneme formuli a máme vytvořit graf, v němž se bude nezávislá množina určené velikosti nacházet právě tehdy, je-li formule splnitelná. Myšlenka převodu bude jednoduchá: z každé klauzule budeme chtít vybrat jeden literál, jehož nastavením klauzuli splníme. Samozřejmě si musíme dát pozor, abychom v různých klauzulích nevybírali konfliktně, tj. jednou x a podruhé $\neg x$.

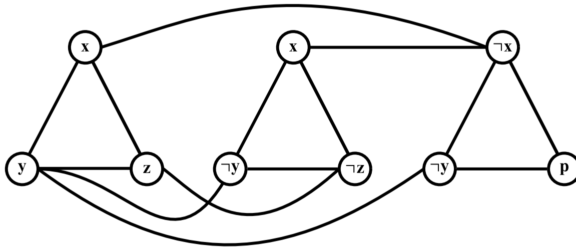
Jak to přesně zařídit: pro každou z k klauzulí zadané formule vytvoříme trojúhelník a jeho vrcholům přiřadíme literály klauzule. (Pokud by klauzule obsahovala méně literálů, prostě některé vrcholy trojúhelníka smažeme.) Navíc spojíme hranami všechny dvojice konfliktních literálů (x a $\neg x$) z různých trojúhelníků.

V tomto grafu se budeme ptát po nezávislé množině velikosti alespoň k. Jelikož z každého trojúhelníka můžeme do nezávislé množiny vybrat nejvýše jeden vrchol, jediná možnost, jak dosáhnout požadované velikosti, je vybrat z každého právě jeden vrchol. Ukážeme, že taková nezávislá množina existuje právě tehdy, je-li formule splnitelná.

Máme-li splňující ohodnocení formule, můžeme z každé klauzule vybrat jeden splněný literál. Do nezávislé množiny umístíme vrcholy odpovídající těmto literálům. Je jich právě k. Jelikož každé dva vybrané vrcholy leží v různých trojúhelnících a nikdy nemůže být splněný současně literál a jeho negace, množina je opravdu nezávislá.

A opačně: Kdykoliv dostaneme nezávislou množinu velikosti k, vybereme literály odpovídající vybraným vrcholům a příslušné proměnné nastavíme tak, abychom tyto literály splnili. Díky hranám mezi konfliktními literály se nikdy nestane, že bychom potřebovali proměnnou nastavit současně na 0 a na 1. Zbývající proměnné ohodnotíme libovolně. Jelikož jsme v každé klauzuli splnili alespoň jeden literál, jsou splněny všechny klauzule, a tedy i celá formule.

Převod je tedy korektní, zbývá rozmyslet, že běží v polynomiálním čase: Počet vrcholů grafu odpovídá počtu literálů ve formuli, počet hran je maximálně kvadratický. Každý vrchol i hranu přitom sestrojíme v polynomiálním čase, takže celý převod je také polynomiální.



Obr. 1.2: Graf pro formuli $(x \vee y \vee z) \wedge (x \vee \neg y \vee \neg z) \wedge (\neg x \vee \neg y \vee p)$

Převod NzMna → SAT: Dostaneme graf a číslo k, chceme vytvořit formuli, která je splnitelná právě tehdy, pokud se v grafu nachází nezávislá množina o alespoň k vrcholech. Tuto formuli sestrojíme následovně.

Vrcholy grafu očíslujeme od 1 do n a pořídíme si pro ně proměnné $v_1, \ldots, v_n$, které budou indikovat, zda byl příslušný vrchol vybrán do nezávislé množiny (příslušné ohodnocení proměnných tedy bude odpovídat charakteristické funkci nezávislé množiny).

Aby množina byla opravdu nezávislá, pro každou hranu $ij \in E(G)$ přidáme klauzuli $(\neg v_i \vee \neg v_j)$.

Ještě potřebujeme zkontrolovat, že množina je dostatečně velká. To neumíme provést přímo, ale použijeme lest: vyrobíme matici proměnných X tvaru $k \times n$, která bude popisovat očíslování vrcholů nezávislé množiny čísly od 1 do k. Konkrétně x_{ij} bude říkat, že v pořadí i-tý prvek nezávislé množiny je vrchol j. K tomu potřebujeme zařídit:

- Aby v každém sloupci byla nejvýše jedna jednička. Na to si pořídíme klauzule $(x_{i,j} \Rightarrow \neg x_{i',j})$ pro $i' \neq i$. (Jsou to implikace, ale můžeme je zapsat i jako disjunkce, protože $a \Rightarrow b$ je totéž jako $\neg a \vee b$.)
- Aby v každém řádku ležela právě jedna jednička. Nejprve zajistíme nejvýše jednu klauzulemi $(x_{i,j} \Rightarrow \neg x_{i,j'})$ pro $j' \neq j$. Pak přidáme klauzule $(x_{i,1} \vee x_{i,2} \vee \ldots \vee x_{i,n})$, které požadují alespoň jednu jedničku v řádku.

- Vztah mezi očíslováním a nezávislou množinou: přidáme klauzule $x_{i,j} \Rightarrow v_j$. (Všimněte si, že nezávislá množina může obsahovat i neočíslované prvky, ale to nám nevadí. Důležité je, aby jich měla k očíslovaných.)

Správnost převodu je zřejmá, ověřme ještě, že probíhá v polynomiálním čase. To plyne z toho, že vytvoříme polynomiálně mnoho klauzulí a každou z nich stihneme vypsat v lineárním čase.

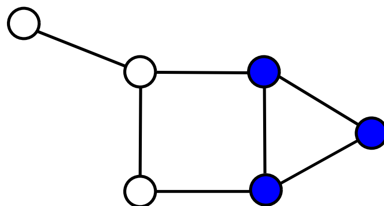
Dokázali jsme tedy, že testování existence nezávislé množiny je stejně těžké jako testování splnitelnosti formule. Pojďme se podívat na další problémy.

Problém Klika – úplný podgraf

Podobně jako nezávislou množinu můžeme v grafu hledat i *kliku* – úplný podgraf dané velikosti.

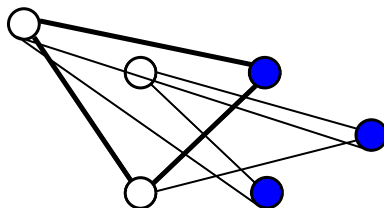
Vstup problému: Graf G a číslo $k \in N$.

Výstup problému: Existuje-li úplný podgraf grafu G na alespoň k vrcholech.



Obr. 1.3: Příklad kliky

Tento problém je ekvivalentní s hledáním nezávislé množiny. Pokud v grafu prohodíme hrany a nehrany, stane se z každé kliky nezávislá množina a naopak. Převodní funkce tedy zneguje hrany a ponechá číslo k.



Obr. 1.4: Prohození hran a nehran

Problém 3,3-SAT – splnitelnost s malým počtem výskytů

Než se pustíme do dalšího kombinatorického problému, předvedeme ještě jednu speciální variantu SATu, se kterou se nám bude pracovat příjemněji.

Již jsme ukázali, že SAT zůstane stejně těžký, omezíme-li se na formule s klauzulemi délky nejvýše 3. Teď budeme navíc požadovat, aby se každá proměnná vyskytovala v maximálně třech literálech. Tomuto problému se říká 3,3-SAT.

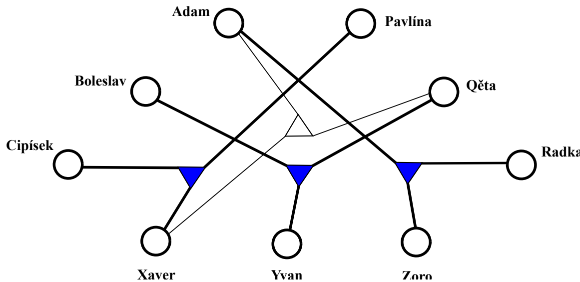
Převod 3-SATu na 3,3-SAT: Pokud se proměnná x vyskytuje v $k > 3$ literálech, nahradíme její výskyty novými proměnnými $x_1, \ldots, x_k$ a přidáme klauzule, které zabezpečí, že tyto proměnné budou vždy ohodnoceny stejně: $(x_1 \Rightarrow x_2), (x_2 \Rightarrow x_3), (x_3 \Rightarrow x_4), \ldots, (x_{k-1} \Rightarrow x_k), (x_k \Rightarrow x_1)$.

Zesílení: Můžeme dokonce zařídit, aby se každý literál vyskytoval nejvýše dvakrát (tedy že každá proměnná se vyskytuje alespoň jednou pozitivně a alespoň jednou negativně). Pokud by se nějaká proměnná objevila ve třech stejných literálech, můžeme na ni také použít náš trik a nahradit ji třemi proměnnými. V nových klauzulích se pak bude vyskytovat jak pozitivně, tak negativně (opět připomínáme, že $a \Rightarrow b$ je jen zkratka za $\neg a \vee b$).

Problém 3D-párování

Vstup problému: Tři množiny, např. K (kluci), H (holky), Z (zvířátka) a množina $T \subseteq K \times H \times Z$ kompatibilních trojic (těch, kteří se spolu snesou).

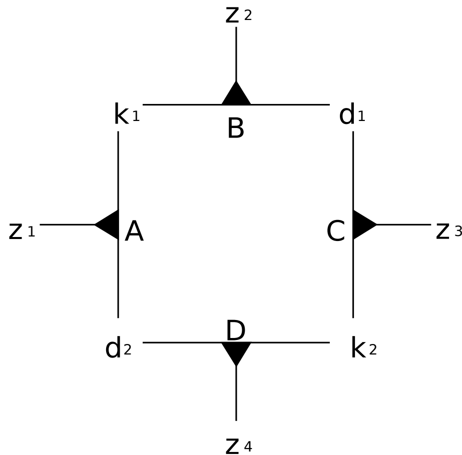
Výstup problému: Zda existuje perfektní podmnožina trojic, tedy taková, v níž se každý prvek množin K, H a Z účastní právě jedné trojice.



Obr. 1.5: Ukázka 3D-párování

Převod 3,3-SATu na 3D-párování: Uvažujme trochu obecněji. Pokud chceme ukázat, že se na nějaký problém dá převést SAT, potřebujeme obvykle dvě věci: Jednak konstrukci, která bude simulovat proměnné, tedy něco, co nabývá dvou stavů *true*/*false*. Pak také potřebujeme cosi, co umí zařídit, aby každá klauzule byla splněna alespoň jednou proměnnou. Jak to provést u 3D-párování?

Uvažujme následující konfiguraci:

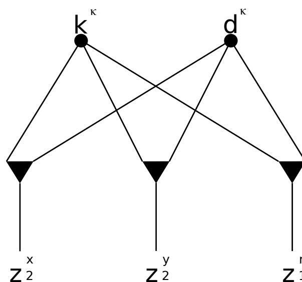


V ní se nacházejí 4 zvířátka (z_1 až z_4), 2 kluci (k_1 a k_2), 2 dívky (d_1 a d_2) a 4 trojice (A, B, C a D). Zatímco zvířátka se budou moci účastnit i jiných trojic, kluky a děvčata nikam jinam nezapojíme.

Všimneme si, že existují právě dvě možnosti, jak tuto konfiguraci spárovat. Abychom spárovali kluka k_1, tak musíme vybrat buď trojici A nebo B. Pokud si vybereme A, k_1 i d_2 už jsou spárovaní, takže si nesmíme vybrat B ani D. Pak jediná možnost, jak spárovat d_1 a k_2, je použít C. Naopak začneme-li trojicí B, vyloučíme A a C a použijeme D (situace je symetrická).

Vždy si tedy musíme vybrat dvě protější trojice v obrázku a druhé dvě nechat nevyužité. Tyto možnosti budeme používat k reprezentaci proměnných. Pro každou proměnnou si pořídíme jednu kopii obrázku. Volba $A + C$ bude odpovídat nule a nespáruje zvířátka z_2 a z_4. Volba $B + D$ reprezentuje jedničku a nespáruje z_1 a z_3. Přes tato nespárovaná zvířátka můžeme předávat informaci o hodnotě proměnné do klauzulí.

Zbývá vymyslet, jak reprezentovat klauzule. Mějme klauzuli tvaru řekněme $(x \vee y \vee \neg r)$. Potřebujeme zajistit, aby x bylo nastavené na 1 nebo y bylo nastavené na 1 nebo r na 0.



Pro takovouto klauzuli si pořídíme novou dvojici kluk a dívka, kteří budou figurovat ve třech trojicích se třemi různými zvířátky, což budou volná zvířátka z obrázků

pro příslušné proměnné. Zvolíme je tak, aby byla volná při správném nastavení proměnné. Dokážeme přitom zařídit, že každé zvířátko bude použité v maximálně jedné klauzuli, neboť každý literál se vyskytuje nejvýše dvakrát a máme pro něj dvě volná zvířátka.

Ještě nám určitě zbude $2p - k$ zvířátek, kde p je počet proměnných a k počet klauzulí. Každá proměnná totiž dodá 2 volná zvířátka a každá klauzule použije jedno z nich. Přidáme proto ještě $2p - k$ párů lidí, kteří milují úplně všechna zvířátka; ti vytvoří zbývající trojice.

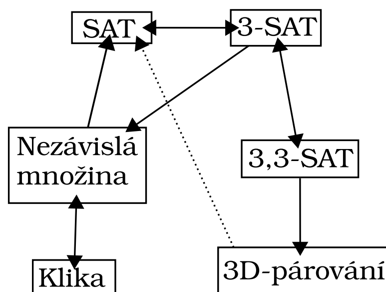
Snadno ověříme, že celý převod pracuje v polynomiálním čase, rozmysleme si ještě, že je korektní.

Pokud formule byla splnitelná, z každého splňujícího ohodnocení můžeme vyrobit párování v naší konstrukcí. Obrázek pro každou proměnnou spárujeme podle ohodnocení (buď $A+C$ nebo $B+D$). Pro každou klauzuli si vybereme trojici, která odpovídá některému z literálů, jimiž je klauzule splněna.

A opačně: Když nám někdo dá párovaní v naší konstrukci, dokážeme z něj vyrobit splňující ohodnocení dané formule. Podíváme se, v jakém stavu je proměnná, a to je všechno. Z toho, že jsou správně spárované klauzule, už okamžitě víme, že jsou všechny splněné.

Ukázali jsme tedy, že na 3D-párování lze převést 3,3-SAT, a tedy i obecný SAT. Převod v opačném směru ponecháme jako cvičení, můžete ho provést podobně, jako jsme na SAT převáděli nezávislou množinu.

Jak je vidět z následujícího schématu, ukázali jsme, že všechny problémy, které jsme zkoumali, jsou navzájem převoditelné.



Obr. 1.6: Problémy a převody mezi nimi

Cvičení:

1. Vrcholové pokrytí grafu je množina vrcholů, která obsahuje alespoň jeden vrchol z každé hrany. (Chceme na křižovatky rozmístit strážníky tak, aby každou ulici alespoň jeden hlídal.) Ukažte vzájemné převody mezi problém nezávislé množiny a problémem „Existuje vrcholové pokrytí velikosti nejvýše k?“.

2. Zesilte náš převod SATu na nezávislou množinu tak, aby vytvářel grafy s maximálním stupněm 4.

1.2. NP-úplné problémy

Všechny problémy, které jsme zatím zkoumali, měly jednu společnou vlastnost. Šlo v nich totiž o to, zda existuje nějaký objekt. Například splňující ohodnocení formule nebo klika v grafu. Kdykoliv nám přitom někdo takový objekt ukáže, umíme snadno ověřit, že má požadovanou vlastnost. Ovšem najít ho už tak snadné není. Podobně se chovají i mnohé další „vyhledávací problémy", pojďme se na ně tedy podívat obecněji.

Definice: **P** je třída[5] 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 takové, že pro každý vstup x algoritmus A doběhne v čase nejvýše $f(|x|)$ a vydá výsledek $A(x) = L(x)$.

Třída **P** tedy zachycuje naši představu o efektivně řešitelných problémech. Nyní definujeme třídu **NP**, která bude odpovídat naší představě vyhledávacích problémů.

Definice: **NP** je třída rozhodovacích problémů, v níž problém L leží právě tehdy, pokud existuje nějaký problém $K \in \mathbf{P}$ a polynom g, přičemž pro každý vstup x je $L(x) = 1$ právě tehdy, pokud pro nějaký řetězec y délky nejvýše $g(|x|)$ platí $K(x, y) = 1$.[6]

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.

Jinými slovy y je jakýsi *certifikát,* který stvrzuje kladnou odpověď, a problém K má za úkol certifikáty kontrolovat. Pro kladnou odpověď musí existovat alespoň jeden schválený certifikát, pro zápornou musí být všechny certifikáty odmítnuty.

Pozorování: Splnitelnost logických formulí je v **NP**. Stačí si totiž nechat napovědět, jak ohodnotit jednotlivé proměnné, a pak ověřit, je-li formule splněna. Nápověda je polynomiálně velká (dokonce lineárně), splnění zkontrolujeme také v lineárním čase. Podobně to můžeme to dokázat i o ostatních rozhodovacích problémech, se kterými jsme se zatím potkali.

Pozorování: 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.

---

[5] Formálně vzato je to množina, ale v teorii složitosti se pro množiny problémů vžil název *třídy.*

[6] Rozhodovací problémy mají na vstupu řetězec bitů. Tak jaképak x, y? Máme samozřejmě na mysli nějaké binární kódování této dvojice.

Otázka: Jsou třídy **P** a **NP** různé? Na to se teoretičtí informatici snaží odpovědět už od 70. let minulého století a postupně se z toto stal asi vůbec nejslavnější otevřený problém informatiky.

Například o žádném z našich problémů nevíme, zda se nachází v **P**. Brzy uvidíme, že to jsou v jistém smyslu nejtěžší problémy v **NP**.

Definice: Problém L nazveme **NP**-*těžký*, je-li na něj převoditelný každý problém z **NP**. Pokud navíc $L \in$ **NP**, budeme říkat, že je **NP**-*úplný.*

NP-úplné problémy jsou tedy nejtěžšími problémy v **NP**, aspoň v našem uspořádání převoditelností.

Lemma: Pokud nějaký **NP**-těžký problém L leží v **P**, pak **P** = **NP**.

Důkaz: Již víme, že **P** $\subseteq$ **NP**, takže stačí dokázat opačnou nerovnost. Vezměme libovolný problém $A \in$ **NP**. Z úplnosti problému L víme, že A lze převést na L. Ovšem problémy převoditelné na něco z **P** jsou samy také v **P**. □

Z definice **NP**-úplnosti ale vůbec není jasné, že nějaký **NP**-úplný problém doopravdy existuje. Odpověď je překvapivá:

Věta (Cookova): SAT je **NP**-úplný.

Důkaz této věty je značně technický a alespoň v hrubých rysech ho předvedeme v závěru teto kapitoly. Jakmile ale máme jeden **NP**-úplný problém, můžeme velice snadno dokazovat i **NP**-úplnost dalších:

Lemma: Mějme dva problémy $L, M \in$ **NP**. Pokud L je **NP**-úplný a $L \rightarrow M$, pak M je také **NP**-úplný.

Důkaz: Jelikož M leží v **NP**, stačí o něm dokázat, že je **NP**-těžký, tedy že na něj lze převést libovolný problém z **NP**. Uvažme tedy nějaký problém $Q \in$ **NP**. Jelikož L je **NP**-úplný, musí platit $Q \rightarrow L$. Převoditelnost je ovšem tranzitivní, takže z $Q \rightarrow L$ a $L \rightarrow M$ plyne $Q \rightarrow M$. □

Důsledek: Cokoliv, na co jsme uměli převést SAT, je také NP-úplné. Například nezávislá množina, různé varianty SATu, klika v grafu …

Dva možné světy: Jestli je **P** = **NP** nevíme a nejspíš ještě dlouho nebudeme vědět. Nechme se ale na chvíli unášet fantazií a zkusme si představit, jak by vypadaly světy, v nichž platí jedna nebo druhá možnost:

- **P** = **NP** – to je na první pohled idylický svět, v němž jde každý vyhledávací problém vyřešit v polynomiálním čase, nejspíš tedy i prakticky efektivně. Má to i své stinné stránky: například jsme přišli o veškeré efektivní šifrování – rozmyslete si, že pokud umíme vypočítat nějakou funkci v polynomiálním čase, umíme efektivně spočítat i její inverzi.
- **P** $\neq$ **NP** – tehdy jsou **P** a **NP**-úplné dvě disjunktní třídy. SAT a ostatní **NP**-úplné problémy nejsou řešitelné v polynomiálním čase. Je ale stále možné, že aspoň na některé z nich existují prakticky použitelné algoritmy, třeba o složitosti $\Theta((1+\varepsilon)^n)$ nebo $\Theta(n^{\log n/100})$.

Ví se, že třída **NP** by pak obsahovala i problémy, které leží „mezi“ **P** a **NP**-úplnými.

Katalog NP-úplných problémů

Pokud se setkáme s problémem, který neumíme zařadit do **P**, hodí se vyzkoušet, zda je **NP**-úplný. K tomu se hodí mít alespoň základní zásobu „učebnicových“ **NP**-úplných problémů, abychom si mohli vybrat, z čeho převádět. Ukážeme tedy katalog několika nejběžnějších **NP**-úplných problémů. O některých jsme to dokázali během této kapitoly, u ostatních alespoň naznačíme, jak na ně.

- *Logické:*
  - SAT (splnitelnost logických formulí v CNF)
  - 3-SAT (každá klauzule obsahuje max. 3 literály)
  - 3,3-SAT (a navíc každá proměnná se vyskytuje nejvýše 3×)
  - SAT pro obecné formule (nejen CNF; ukážeme níže)
  - Obvodový SAT (místo formule booleovský obvod; viz níže)
- *Grafové:*
  - Nezávislá množina (existuje množina alespoň k vrcholů taková, že žádné dva nejsou propojeny hranou?)
  - Klika (existuje úplný podgraf na k vrcholech?)
  - 3D-párování (tři množiny se zadanými trojicemi, existuje taková množina disjunktních trojic, ve které jsou všechny prvky právě jednou?)
  - Barvení grafu (lze obarvit vrcholy k barvami tak, aby vrcholy stejné barvy nebyly nikdy spojeny hranou? **NP**-úplné už pro $k = 3$)
  - Hamiltonovská cesta (cesta obsahující všechny vrcholy)
  - Hamiltonovská kružnice (opět obsahuje všechny vrcholy)
- *Číselné:*
  - Batoh (nejjednodušší verze: má daná množina čísel podmnožinu s daným součtem?)
  - Batoh – optimalizace (podobně jako u předchozího problému, ale místo množiny čísel máme množinu předmětů s vahami a cenami, chceme co nejdražší podmnožinu, jejíž váha nepřesáhne zadanou kapacitu batohu)
  - Dva loupežníci (lze rozdělit danou množinu čísel na dvě podmnožiny se stejným součtem?)
  - $\mathbf{Ax} = \mathbf{b}$ (soustava celočíselných lineárních rovnic; je dána matice $\mathbf{A} \in \{0,1\}^{m \times n}$ a vektor $\mathbf{b} \in \{0,1\}^m$, existuje vektor $\mathbf{x} \in \{0,1\}^n$ takový, že $\mathbf{Ax} = \mathbf{b}$?)

Náznak důkazu Cookovy věty

Zbývá dokázat Cookovu větu. Technické detaily si odpustíme, ale aspoň načrtneme základní myšlenku důkazu.

Potřebujeme tedy dokázat, že SAT je **NP**-úplný, a to z definice. Ukážeme to ale nejprve pro jiný problém, pro takzvaný *obvodový SAT.* V něm máme na vstupu booleovský obvod (hradlovou síť) s jedním výstupem a ptáme se, zda jí můžeme přivést na vstupy takové hodnoty, aby vydala výsledek 1.

To je obecnější než SAT pro formule (dokonce i neomezíme-li formule na CNF), protože každou formuli můžeme přeložit na lineárně velký obvod. (Platí i opačně, že každému obvodu s jedním výstupem můžeme přiřadit ekvivalentní formuli, ale ta může být až exponenciálně velká.)

Nejprve tedy dokážeme **NP**-úplnost obvodového SATu a pak ho převedeme na obyčejný SAT v CNF. Tím bude důkaz Cookovy věty hotový. Začněme lemmatem, v němž bude koncentrováno vše technické.

Lemma: Nechť L je problém ležící v **P**. Potom existuje polynom p a algoritmus, který pro každé n sestrojí v čase $p(n)$ hradlovou síť B_n s n vstupy a jedním výstupem, která řeší L. Tedy pro všechny řetězce x musí platit $B_n(x) = L(x)$.

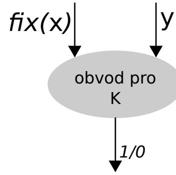
Důkaz: Náznakem. Na základě zkušeností z Principů počítačů intuitivně chápeme počítače jako nějaké složité booleovské obvody, jejichž stav se mění v čase. Uvažme tedy nějaký problém $L \in \mathbf{P}$ a polynomiální algoritmus, který ho řeší. Pro vstup velikosti n algoritmus doběhne v čase T polynomiálním v n a spotřebuje $\mathcal{O}(T)$ buněk paměti. Stačí nám tedy „počítač s pamětí velkou $\mathcal{O}(T)$“, což je nějaký booleovský obvod velikosti polynomiální v T, a tedy i v n. Vývoj v čase ošetříme tak, že sestrojíme T kopií tohoto obvodu, každá z nich bude odpovídat jednomu kroku výpočtu a bude propojena s „minulou“ a „budoucí“ kopií. Tím sestrojíme booleovský obvod, který bude řešit problém L pro vstupy velikosti n a bude polynomiálně velký vzhledem k n.

Ekvivalentní definice NP: Pro důkaz následující věty si dovolíme drobnou úpravu v definici třídy **NP**. Budeme chtít, aby nápověda měla pevnou velikost, závislou pouze na velikosti vstupu (tedy: $|y| = g(|x|)$ namísto $|y| \leq g(|x|)$). Proč je taková úprava bez újmy na obecnosti? Stačí původní nápovědu doplnit na požadovanou délku nějakými „mezerami“, které budeme při ověřování nápovědy ignorovat.

Věta: Obvodový SAT je **NP**-úplný.

Důkaz: Obvodový SAT evidentně leží v **NP** – stačí si nechat poradit vstup, síť topologicky setřídit a v tomto pořadí počítat hodnoty hradel.

Mějme nyní nějaký problém L z **NP**, o němž chceme dokázat, že se dá převést na obvodový SAT. Když nám někdo předloží nějaký vstup x délky n, spočítáme velikost nápovědy $g(n)$. Víme, že algoritmus K, který kontroluje, zda nápověda je správně, je v **P**. Využijeme předchozí lemma, abychom získali obvod, který pro konkrétní velikost vstupu n počítá to, co kontrolní algoritmus K. Vstupem tohoto obvodu bude x (vstup problému L) a nápověda y. Na výstupu se dozvíme, zda je nápověda správná. Velikost tohoto obvodu bude činit $p(g(n))$, což je také polynom.



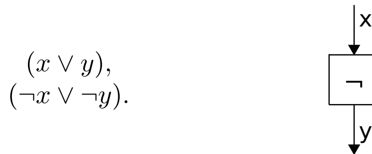
V tomto obvodu zafixujeme vstup x (na místa vstupu dosadíme konkrétní hodnoty z x). Tím získáme obvod, jehož vstup je jen y, a chceme zjistit, zda za y můžeme dosadit nějaké hodnoty tak, aby na výstupu bylo *true*. Jinými slovy, ptáme se, zda je tento obvod splnitelný.

Ukázali jsme tedy, že pro libovolný problém z **NP** dokážeme sestrojit funkci, která pro každý vstup x v polynomiálním čase vytvoří obvod, jenž je splnitelný právě tehdy, když odpověď tohoto problému na vstup x má být kladná. To je přesně převod z daného problému na obvodový SAT. □

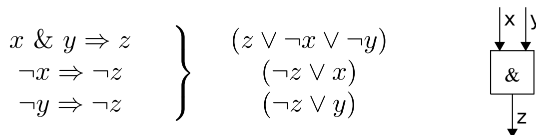
Lemma: Obvodový SAT se dá převést na 3-SAT.

Důkaz: Budeme postupně budovat formuli v konjunktivní normální formě. Každý booleovský obvod se dá v polynomiálním čase převést na ekvivalentní obvod, ve kterém se vyskytují jen hradla AND a NOT, takže stačí najít klauzule odpovídající těmto hradlům. Pro každé hradlo v obvodu zavedeme novou proměnnou popisující jeho výstup. Přidáme klauzule, které nám kontrolují, že toto hradlo máme ohodnocené konzistentně.

Převod hradla NOT: Na vstupu hradla budeme mít nějakou proměnnou x (která přišla buďto přímo ze vstupu celého obvodu, nebo je to proměnná, která vznikla na výstupu nějakého hradla) a na výstupu proměnnou y. Přidáme klauzule, které nám zaručí, že jedna proměnná bude negací té druhé:



Převod hradla AND: Hradlo má vstupy x, y a výstup z. Potřebujeme přidat klauzule, které nám popisují, jak se má hradlo AND chovat. Tyto vztahy přepíšeme do konjunktivní normální formy:



Tím v polynomiálním čase vytvoříme formuli, která je splnitelná právě tehdy, je-li splnitelný zadaný obvod. Ve splňujícím ohodnocení formule bude obsaženo jak splňující ohodnocení obvodu, tak výstupy všech hradel obvodu. □

Poznámka: Tím jsme také odpověděli na otázku, kterou jsme si kladli při zavádění SATu, tedy zda omezením na CNF o něco přijdeme. Teď už víme, že nepřijdeme – libovolná booleovská formule se dá přímočaře převést na obvod a ten zase na formuli v CNF. Zavádíme sice nové proměnné, ale nová formule je splnitelná právě tehdy, kdy ta původní.

Cvičení:

1. Dokažte **NP**-úplnost problému $\mathbf{Ax} = \mathbf{b}$ z katalogu.
2.* Dokažte **NP**-úplnost problému barvení grafu z katalogu.
3. Ukažte, že barvení grafu jednou nebo dvěma barvami je snadné.
4. Převeďte batoh na dva loupežníky a opačně.
5. Dokažte **NP**-úplnost problému batohu.
6. Pokud bychom definovali **P**-úplnost analogicky k **NP**-úplnosti, které problémy z **P** by byly **P**-úplné?
7.* Dokažte lemma o vztahu mezi problémy z **P** a hradlovými sítěmi pomocí výpočetního modelu RAM.

1.3. Co si počít s těžkým problémem

V předchozí kapitole jsme zjistili, že leckteré rozhodovací problémy jsou **NP**-úplné. Z toho plyne, že jsou ekvivalentní, ale bohužel také, že ani jeden z nich zatím neumíme vyřešit v polynomiálním čase.

Často se stane, že problém, který v životě potkáme, patří mezi **NP**-úplné. Přesněji řečeno spíš než s rozhodovacím problémem se potkáme s problémem *optimalizačním,* ve kterém jde o nalezení *nejlepšího* objektu s danou vlastností. To může být třeba největší nezávislá množina v grafu nebo obarvení grafu nejmenším možným počtem barev. Kdybychom uměli efektivně řešit optimalizační problém, umíme samozřejmě řešit i příslušný rozhodovací, takže pokud $\mathbf{P} \neq \mathbf{NP}$, jsou i optimalizační problémy těžké.

Ale co naplat, svět nám takové úlohy předkládá a my je potřebujeme vyřešit. Naštěstí situace není zase tak beznadějná. Nabízejí se tyto možnosti, co si počít:

1. *Spokojit se s málem.* Nejsou vstupy, pro které problém potřebujeme řešit, dostatečně malé, abychom si mohli dovolit použít algoritmus s exponenciální složitostí? Zvlášť když takový algoritmus vylepšíme prořezáváním neperspektivních větví výpočtu a třeba ho i paralelizujeme.
2. *Vyřešit speciální případ.* Nemají naše vstupy nějaký speciální tvar, kterého bychom mohli využít? Grafové problémy jsou často v **P** třeba pro stromy nebo i obecněji pro bipartitní grafy. U číselných problémů zase někdy pomůže, jsou-li čísla na vstupu dostatečně malá.

3. *Řešení aproximovat.* Opravdu potřebujeme optimální řešení? Nestačilo by nám o kousíček horší? Často existuje polynomiální algoritmus, který nalezne nejhůře c-krát horší řešení než je optimum, kde c je konstanta.
4. *Použít heuristiku.* Neumíme-li nic lepšího, můžeme sáhnout po některé z mnoha heuristických technik, které sice nic nezaručují, ale obvykle nějaké uspokojivé řešení najdou. Může pomoci třeba hladový algoritmus nebo genetické algoritmy. Často platí, že čím déle heuristiku necháme běžet, tím lepší řešení najde.
5. *Kombinace přístupů.* Mnohdy lze předchozí přístupy kombinovat: například použít aproximační algoritmus a poté jeho výsledek ještě heuristicky vylepšovat. Tak získáme řešení, které od optima zaručeně není moc daleko, a pokud budeme mít štěstí, bude se od něj lišit jen velmi málo.

Nyní si některé z těchto technik předvedeme na konkrétních příkladech.

Největší nezávislá množina ve stromu

Ukážeme, že hledání největší nezávislé množiny je snadné, pokud graf je strom.

Lemma: Buď T zakořeněný strom a ℓ jeho libovolný list. Pak alespoň jedna z největších nezávislých množin obsahuje ℓ.

Důkaz: Mějme největší nezávislou množinu M, která list ℓ neobsahuje. Podívejme se na otce p listu ℓ (kdyby neexistoval, je celý strom jednovrcholový a tvrzení triviální). Leží p v M? Pokud ne, mohli bychom do M přidat list ℓ a dostali bychom větší nezávislou množinu. V opačném případě z M odebereme otce p a nahradíme ho listem ℓ, čímž dostaneme stejně velkou nezávislou množinu obsahující ℓ. □

Algoritmus bude přímočaře používat toto lemma. Dostane na vstupu strom, ten zakoření a zvolí libovolný list. Tento list umístí do nezávislé množiny a jeho otce odebere, protože se nemůže v nezávislé množině vyskytovat. Toto bude opakovat, dokud nějaké vrcholy zbývají. (Graf se v průběhu může rozpadnout na více komponent, ale to nevadí.)

Tento algoritmus jistě pracuje v polynomiálním čase. Šikovnou implementací můžeme složitost snížit až na lineární. Například tak, že budeme udržovat seznam listů. My si ukážeme jinou lineární implementaci založenou na prohledávání do hloubky. Bude pracovat s polem značek M, v němž na počátku bude všude *false* a postupně obdrží *true* všechny prvky hledané nezávislé množiny.

Algoritmus NzMnaVeStromu

Vstup: Strom T s kořenem v, pole značek M.

1. $M[v] \leftarrow$ *true*.
2. Pokud je v list, skončíme.
3. Pro všechny syny w vrcholu v:
4. Zavoláme se rekurzivně na podstrom s kořenem w.
5. Pokud $M[w] =$ *true*, položíme $M[v] \leftarrow$ *false*.

Barvení intervalového grafu

Mějme n přednášek s určenými časy začátku a konce. Chceme je rozvrhnout do co nejmenšího počtu poslucháren tak, aby nikdy neprobíhaly dvě přednášky naráz v jedné místnosti.

Chceme tedy obarvit co nejmenším počtem barev graf, jehož vrcholy jsou časové intervaly a dvojice intervalů je spojena hranou, pokud má neprázdný průnik. Takovým grafům se říká *intervalové* a pro jejich barvení existuje pěkný polynomiální algoritmus.

Podobně jako jsme geometrické problémy řešili zametáním roviny, zde budeme „zametat přímku bodem“, tedy procházet ji zleva doprava, a všímat si událostí, což budou začátky a konce intervalů. Pro jednoduchost předpokládejme, že všechny souřadnice začátků a konců jsou navzájem různé.

Kdykoliv interval začne, přidělíme mu barvu. Až skončí, o barvě si poznamenáme, že je momentálně volná, a dalším intervalům budeme přednostně přidělovat volné barvy. Řečeno v pseudokódu:

Algoritmus BarveníIntervalů
Vstup: Intervaly $[x_1, y_1], \ldots, [x_n, y_n]$.

1. $b \leftarrow 0$ *(počet zatím použitých barev)*
2. $B \leftarrow \emptyset$ *(které barvy jsou momentálně volné)*
3. Setřídíme množinu všech x_i a y_i.
4. Procházíme všechna x_i a y_i ve vzestupném pořadí:
5. Narazíme-li na x_i:
6. Je-li $B \neq \emptyset$, odebereme jednu barvu z B a uložíme ji do c_i.
7. Jinak $b \leftarrow b + 1$ a $c_i \leftarrow b$.
8. Narazíme-li na y_i:
9. Vrátíme barvu c_i do B.

Výstup: Obarvení $c_1, \ldots, c_n$.

Analýza: Tento algoritmus má časovou složitost $\mathcal{O}(n \log n)$ kvůli třídění souřadnic. Samotné obarvování je lineární.

Ještě ovšem potřebujeme dokázat, že jsme použili minimální možný počet barev. Uvažujme okamžik, kdy proměnná b naposledy vzrostla. Tehdy začal interval a množina B byla prázdná, což znamená, že jsme $b - 1$ předchozích barev museli přidělit intervalům, jež začaly a dosud neskončily. Existuje tedy b různých intervalů, které mají společný bod (v grafu tvoří kliku), takže každé obarvení potřebuje alespoň b barev.

Problém batohu s malými čísly

Připomeňme si *problém batohu.* Jeho optimalizační verze vypadá takto: Je dána množina n předmětů s hmotnostmi $h_1, \ldots, h_n$ a cenami $c_1, \ldots, c_n$ a nosnost batohu H. Hledáme podmnožinu předmětů $P \subseteq \{1, \ldots, n\}$, která se vejde do batohu (tedy $h(P) = \sum_{i \in P} h_i \leq H$) a její cena $c(P) = \sum_{i \in P} c_i$ je největší možná.

Ukážeme algoritmus, jehož časová složitost bude polynomiální v počtu předmětů n a součtu všech cen $C = \sum_i c_i$.

Použijeme dynamické programování. Představme si problém omezený na prvních k předmětů. Označme $A_k(c)$ (kde $0 \leq c \leq C$) minimum z hmotností těch podmnožin jejichž cena je právě c; pokud žádná taková podmnožina neexistuje, položíme $A_k(c) = \infty$.

Tato A_k spočteme indukcí podle k: Pro $k = 0$ je určitě $A_0(0) = 0$ a $A_0(1) = \ldots = A_0(C) = \infty$. Pokud již známe A_{k-1}, spočítáme A_k následovně: $A_k(c)$ odpovídá nějaké podmnožině předmětů z $1, \ldots, k$. V této podmnožině jsme buďto k-tý předmět nepoužili, a pak je $A_k(c) = A_{k-1}(c)$, nebo použili, a tehdy bude $A_k(c) = A_{k-1}(c - c_k) + h_k$ (to samozřejmě jen pokud $c \geq c_k$). Z těchto dvou možností si vybereme tu, která dává množinu s menší hmotností:

$$A_k(c) = \min(A_{k-1}(c), A_{k-1}(c - c_k) + h_k).$$

Přechod od A_{k-1} k A_k tedy trvá $\mathcal{O}(C)$, od A_1 až k A_n se dopočítáme v čase $\mathcal{O}(Cn)$.

Jakmile získáme A_n, známe pro každou cenu příslušnou nejlehčí podmnožinu. Maximální cena množiny, která se vejde do batohu, je tedy největší c^*, pro něž je $A_n(c^*) \leq H$. Jeho nalezení nás stojí čas $\mathcal{O}(C)$.

Zbývá zjistit, které předměty do nalezené množiny patří. Upravíme algoritmus, aby si pro každé $A_k(c)$ pamatoval ještě $B_k(c)$, což bude index posledního předmětu, který jsme do příslušné množiny přidali. Pro nalezené c^* tedy bude $i = B_n(c^*)$ poslední předmět v nalezené množině, $i' = B_{i-1}(c^* - c_i)$ ten předposlední a tak dále. Takto v čase $\mathcal{O}(n)$ rekonstruujeme celou množinu od posledního prvku k prvnímu.

Máme tedy algoritmus, který vyřeší problém batohu v čase $\mathcal{O}(nC)$. Tato funkce ovšem není polynomem ve velikosti vstupu, jelikož reprezentujeme-li vstup binárně, C může být až exponenciálně velké vzhledem k délce jeho zápisu. To je pěkný příklad tzv. *pseudopolynomiálního* algoritmu, tedy algoritmu, jehož složitost je polynomem v počtu čísel na vstupu a jejich velikosti. Pro některé **NP**-úplné problémy takové algoritmy existují, pro jiné (např. pro nezávislou množinu) by z jejich existence plynulo **P** = **NP**.

Verze bez cen: Jednodušší verzi problému batohu, která nerozlišuje mezi hmotnostmi a cenami, zvládneme i jiným algoritmem, opět založeným na dynamickém programování.

Indukcí podle k vytváříme množiny Z_k obsahující všechny hmotnosti menší než H, kterých nabývá nějaká podmnožina prvních k prvků. Jistě je $Z_0 = \{0\}$. Podobnou úvahou jako v předchozím algoritmu dostaneme, že každou další Z_k můžeme zapsat jako sjednocení Z_{k-1} s kopií Z_{k-1} posunutou o h_k, ignorujíce hodnoty větší než H. Nakonec ze Z_n vyčteme výsledek.

Všechny množiny přitom mají nejvýše $H + 1$ prvků, takže pokud si je budeme udržovat jako setříděné seznamy, spočítáme sjednocení sléváním v čase $\mathcal{O}(H)$ a celý algoritmus doběhne v čase $\mathcal{O}(Hn)$.

Aproximace problému obchodního cestujícího

V *problému obchodního cestujícího* je zadán neorientovaný graf G, jehož hrany jsou ohodnoceny délkami $\ell(e) \geq 0$. V tomto grafu chceme nalézt nejkratší z hamiltonovských kružnic, tedy těch, které navštíví všechny vrcholy.

Není překvapivé, že tento problém je těžký – už sama existence hamiltonovské kružnice je **NP**-úplná. Ukážeme ovšem, že pokud je graf úplný a platí v něm trojúhelníková nerovnost (tj. $\ell(x,z) \leq \ell(x,y)+\ell(y,z)$ pro všechny trojice vrcholů x,y,z), můžeme problém obchodního cestujícího 2-aproximovat, tedy najít v polynomiálním čase kružnici, která je přinejhorším dvakrát delší než ta optimální.

Grafy s trojúhelníkovou nerovností přitom nejsou nijak neobvyklé – odpovídají totiž konečným metrickým prostorům.

Algoritmus bude snadný: Najdeme nejmenší kostru a obchodnímu cestujícímu poradíme, ať ji obejde. To můžeme popsat například tak, že kostru zakořeníme, prohledáme ji do hloubky a zaznamenáme, jak jsme procházeli hranami. Každou hranou kostry přitom projdeme dvakrát – jednou dolů, podruhé nahoru. Tím však nedostaneme kružnici, nýbrž jen nějaký uzavřený sled, protože vrcholy navštěvujeme vícekrát. Sled tedy upravíme tak, že kdykoliv se dostává do již navštíveného vrcholu, přeskočí ho a přesune se až do nejbližšího dalšího nenavštíveného. Tím ze sledu vytvoříme hamiltonovskou kružnici a jelikož v grafu platí trojúhelníková nerovnost, celková délka nevzrostla. (Pořadí vrcholů na kružnici můžeme získat také tak, že během prohledávání budeme vypisovat vrcholy v preorderu. Rozmyslete si, že je totéž.)

Věta: Nalezená kružnice není delší než dvojnásobek optima.

Důkaz: Označme T délku minimální kostry, A délku kružnice vydané naším algoritmem a O (optimum) délku nejkratší hamiltonovské kružnice. Z toho, jak jsme kružnici vytvořili, víme, že $A \leq 2T$. Platí ovšem také $T \leq O$, jelikož z každé hamiltonovské kružnice vznikne vynecháním hrany kostra a ta nemůže být menší než minimální kostra. Složením obou nerovností získáme $A \leq 2T \leq 2O$. □

Sestrojili jsme tedy 2-aproximační algoritmus pro problém obchodního cestujícího. Dodejme ještě, že trochu složitějším trikem lze tento problém 1.5-aproximovat a že v některých metrických prostorech (třeba v euklidovské rovině) lze v polynomiálním čase najít $(1+\varepsilon)$-aproximaci pro libovolné $\varepsilon > 0$. Ovšem čím menší ε, tím déle algoritmus poběží.

Trojúhelníková nerovnost nicméně byla pro tento algoritmus naprosto nezbytná. To není náhoda – hned dokážeme, že bez tohoto předpokladu je libovolná aproximace stejně těžká jako přesné řešení.

Věta: Pokud pro nějaké reálné $t \geq 1$ existuje polynomiální t-aproximační algoritmus pro problém obchodního cestujícího bez trojúhelníkové nerovnosti, pak je **P** = **NP**.

Důkaz: Ukážeme, že pomocí takového aproximačního algoritmu dokážeme v polynomiálním čase zjistit, zda v grafu existuje hamiltonovská kružnice, což je **NP**-úplný problém.

Dostali jsme graf G, ve kterém hledáme hamiltonovskou kružnici (zkráceně HK). Doplníme G na úplný graf G'. Všem původním hranám nastavíme délku na 1, těm novým na nějaké dost velké číslo c. Kolik to bude, určíme za chvíli.

Graf G' je úplný, takže v něm určitě nějaké HK existují. Ty, které se vyskytují i v původním grafu G, mají délku přesně n. Jakmile ale použijeme jedinou hranu, která z G nepochází, vzroste délka kružnice alespoň na $n - 1 + c$.

Podle délky nejkratší HK tedy dokážeme rozpoznat, zda existuje HK v G. Potřebujeme ovšem zjistit i přes zkreslení způsobené aproximací. Musí tedy platit $tn < n - 1 + c$. To snadno zajistíme volbou hodnoty c větší než $(t-1)n + 1$.

Naše konstrukce přidala polynomiálně mnoho hran s polynomiálně velkým ohodnocením, takže graf G' je polynomiálně velký vzhledem ke G. Rozhodujeme tedy existenci HK v polynomiálním čase a $\mathsf{P} = \mathsf{NP}$. □

Poznámka: Podobně můžeme dokázat, že pokud $\mathsf{P} \neq \mathsf{NP}$, neexistuje pro problém obchodního cestujícího ani pseudopolynomiální algoritmus. Stačí původním hranám přiřadit délku 1 a novým délku 2.

Aproximační schéma pro problém batohu

Již víme, jak optimalizační verzi problému batohu vyřešit v čase $\mathcal{O}(nC)$, pokud jsou hmotnosti i ceny na vstupu přirozená čísla a C je součet všech cen. Jak si poradit, pokud je C obrovské? Kdybychom měli štěstí a všechny ceny byly násobky nějakého čísla p, mohli bychom je tímto číslem vydělit. Tak bychom dostali zadání s menšími čísly, jehož řešením by byla stejná množina předmětů jako u zadání původního.

Když nám štěstí přát nebude, můžeme přesto zkusit ceny vydělit a výsledky nějak zaokrouhlit. Optimální řešení nové úlohy pak sice nemusí odpovídat optimálnímu řešení té původní, ale když nastavíme parametry správně, bude alespoň jeho dobrou aproximací.

Základní myšlenka: Označíme $c_{\max}$ maximum z cen c_i. Zvolíme nějaké přirozené číslo $M < c_{\max}$ a zobrazíme interval cen $[0, c_{\max}]$ na $\{0, \ldots, M\}$ (tedy každou cenu znásobíme poměrem $M/c_{\max}$ a zaokrouhlíme). Jak jsme tím zkreslili výsledek? Všimněme si, že efekt je stejný, jako kdybychom jednotlivé ceny zaokrouhlili na násobky čísla $c_{\max}/M$ (prvky z intervalu $[i \cdot c_{\max}/M, (i+1) \cdot c_{\max}/M)$ se zobrazí na stejný prvek). Každé c_i jsme tím tedy změnili o nejvýše $c_{\max}/M$, celkovou cenu libovolné podmnožiny předmětů pak nejvýše o $n \cdot c_{\max}/M$. Navíc odstraníme-li ze vstupu předměty, které se samy nevejdou do batohu, má optimální řešení původní úlohy cenu $c^* \geq c_{\max}$, takže chyba naší aproximace nepřesáhne $n \cdot c^*/M$. Má-li tato chyba být shora omezena $\varepsilon \cdot c^*$, musíme zvolit $M \geq n/\varepsilon$.

Na této myšlence „kvantování cen“ je založen následující algoritmus.

Algoritmus AproximaceBatohu

1. Odstraníme ze vstupu všechny předměty těžší než H.
2. Spočítáme $c_{\max} = \max_i c_i$ a zvolíme $M = \lceil n/\varepsilon \rceil$.
3. Kvantujeme ceny: Pro $i = 1, \ldots, n$ položíme $\hat{c}_i \leftarrow \lfloor c_i \cdot M/c_{\max} \rfloor$.

4. Vyřešíme dynamickým programováním problém batohu pro upravené ceny $\hat{c}_1, \ldots, \hat{c}_n$ a původní hmotnosti i kapacitu batohu.
5. Vybereme stejné předměty, jaké použilo optimální řešení kvantovaného zadání.

Analýza: Kroky 1–3 a 5 jistě zvládneme v čase $\mathcal{O}(n)$. Krok 4 řeší problém batohu se součtem cen $\hat{C} \leq nM = \mathcal{O}(n^2/\varepsilon)$, což stihne v čase $\mathcal{O}(n\hat{C}) = \mathcal{O}(n^3/\varepsilon)$. Zbývá dokázat, že výsledek našeho algoritmu má opravdu relativní chybu nejvýše ε.

Označme P množinu předmětů použitých v optimálním řešení původní úlohy a $c(P)$ cenu tohoto řešení. Podobně Q bude množina předmětů v optimálním řešení nakvantované úlohy a $\hat{c}(Q)$ jeho hodnota v nakvantovaných cenách. Potřebujeme odhadnout ohodnocení množiny Q v původních cenách, tedy $c(Q)$, a srovnat ho s $c(P)$.

Nejprve ukážeme, jakou cenu má optimální řešení P původní úlohy v nakvantovaných cenách:

$$\hat{c}(P) = \sum_{i \in P} \hat{c}_i = \sum_{i \in P} \left\lfloor c_i \cdot \frac{M}{c_{\max}} \right\rfloor \geq \sum_{i \in P} \left(c_i \cdot \frac{M}{c_{\max}} - 1 \right) \geq$$
$$\geq \left(\sum_{i \in P} c_i \cdot \frac{M}{c_{\max}} \right) - n = c(P) \cdot \frac{M}{c_{\max}} - n.$$

Nyní naopak spočítejme, jak dopadne optimální řešení Q nakvantovaného problému při přepočtu na původní ceny (to je výsledek našeho algoritmu):

$$c(Q) = \sum_{i \in Q} c_i \geq \sum_{i \in Q} \hat{c}_i \cdot \frac{c_{\max}}{M} = \left(\sum_i \hat{c}_i \right) \cdot \frac{c_{\max}}{M} = \hat{c}(Q) \cdot \frac{c_{\max}}{M} \geq \hat{c}(P) \cdot \frac{c_{\max}}{M}.$$

Poslední nerovnost platí proto, že $\hat{c}(Q)$ je optimální řešení kvantované úlohy, zatímco $\hat{c}(P)$ je nějaké další řešení téže úlohy, které nemůže být lepší.⟨7⟩ Teď už stačí složit obě nerovnosti a dosadit za M:

$$c(Q) \geq \left(\frac{c(P) \cdot M}{c_{\max}} - n \right) \cdot \frac{c_{\max}}{M} \geq c(P) - \frac{n \cdot c_{\max}}{n/\varepsilon} \geq c(P) - \varepsilon c_{\max} \geq$$
$$\geq c(P) - \varepsilon c(P) = (1 - \varepsilon) \cdot c(P).$$

Na přechodu mezi řádky jsme využili toho, že každý předmět se vejde do batohu, takže optimum musí být alespoň tak cenné jako nejcennější z předmětů.

Shrňme, co jsme dokázali:

⟨7⟩ Zde nás zachraňuje, že ačkoliv u obou úloh leží optimum obecně jinde, obě mají stejnou množinu *přípustných řešení,* tedy těch, která se vejdou do batohu. Kdybychom místo cen kvantovali hmotnosti, nebyla by to pravda a algoritmus by nefungoval.

Věta: Existuje algoritmus, který pro každé $\varepsilon > 0$ nalezne $(1-\varepsilon)$-*aproximaci* problému batohu s n předměty v čase $\mathcal{O}(n^3/\varepsilon)$.

Dodejme ještě, že algoritmům, které dovedou pro každé $\varepsilon > 0$ najít v polynomiálním čase $(1-\varepsilon)$-aproximaci optimálního řešení, říkáme *polynomiální aproximační schémata* (PTAS – Polynomial-Time Approximation Scheme). V našem případě je dokonce složitost polynomiální i v závislosti na $1/\varepsilon$, takže schéma je *plně polynomiální* (FPTAS – Fully Polynomial-Time Approximation Scheme).

Cvičení:

1. Popište polynomiální algoritmus pro hledání nejmenšího vrcholového pokrytí stromu. (To je množina vrcholů, která obsahuje alespoň jeden vrchol z každé hrany.)
2.* Nalezněte polynomiální algoritmus pro hledání nejmenšího vrcholového pokrytí bipartitního grafu.
3. Ukažte, jak v polynomiálně najít největší nezávislou množinu v intervalovém grafu.
4.* Vyřešte v polynomiálním čase 2-SAT, tedy splnitelnost formulí zadaných v CNF, jejichž klauzule obsahují nejvýše 2 literály.
5. Problém E3,E3-SAT je zesílením 3,3-SATu. Chceme zjistit splnitelnost formule v CNF, jejíž každá klauzule obsahuje právě tři různé proměnné a každá proměnná se nachází v právě třech klauzulích. Ukažte, že tento problém lze řešit efektivně z toho prostého důvodu, že každá taková formule je splnitelná.
6. Pokusíme se řešit problém dvou loupežníků hladovým algoritmem. Probíráme předměty od nejdražšího k nejlevnějšímu a každý dáme tomu loupežníkovi, který má zrovna méně. Je nalezené řešení optimální?
7. Problém tří loupežníků: Je dána množina předmětů s cenami, chceme ji rozdělit na 3 části o stejné ceně. Navrhněte pseudopolynomiální algoritmus.
8. Problém MaxCut: vrcholy zadaného grafu chceme rozdělit do dvou množin tak, aby mezi množinami vedlo co nejvíce hran. Jinými slovy chceme nalézt bipartitní podgraf s co nejvíce hranami. Rozhodovací verze tohoto problému je **NP**-úplná, zkuste jej v polynomiálním čase 2-aproximovat.
9.* V problému MaxE3-SAT dostaneme formuli v CNF, jejíž každá klauzule obsahuje právě 3 různé proměnné, a chceme nalézt ohodnocení proměnných, při němž je splněno co nejvíce klauzulí. Rozhodovací verze je **NP**-úplná. Ukažte, že při náhodném ohodnocení proměnných je splněno v průměru 7/8 klauzulí. Z toho odvoďte deterministickou 7/8-aproximaci v polynomiálním čase.
10. Hledejme vrcholové pokrytí následujícím hladovým algoritmem. V každém kroku vybereme vrchol nejvyššího stupně, přidáme ho do pokrytí a odstraníme ho z grafu i se všemi již pokrytými hranami. Je nalezené pokrytí nejmenší? Nebo alespoň $\mathcal{O}(1)$-aproximace nejmenšího?
11.* Uvažujme následující algoritmus pro nejmenší vrcholové pokrytí grafu. Graf projdeme do hloubky, do výstupu vložíme všechny vrcholy vzniklého DFS stro-

mu kromě listů. Dokažte, že vznikne vrcholové pokrytí a že 2-aproximuje to nejmenší.

12.* V daném orientovaném grafu hledáme acyklický podgraf s co nejvíce hranami. Navrhněte polynomiální 2-aproximační algoritmus.