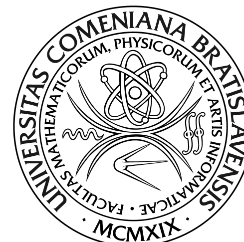
U N I V E R Z I T A   K O M E N S K É H O

Fakulta Matematiky, Fyziky a Informatiky

Katedra Informatiky



Vybrané kapitoly z teoretickej informatiky (1)

Pomocné texty k prednáške 2–AIN–106

(verzia 20. februára 2012)

Bratislava, 2011 Dana Pardubská

Obsah

1 Úvod **5**
1.1 Čo je to Computer Science? . . . 5
1.2 Fascinujúca teória . . . 8
1.3 A teraz už môj úvod . . . 10
1.4 Základné pojmy a definície . . . 11
1.4.1 Analýza zložitosti . . . 11
1.4.2 Analýza problému triedenia . . . 14
1.4.3 Dolný odhad pre problém minMax . . . 16
1.5 Amortizovaná zložitosť . . . 18
1.5.1 Metóda zoskupení . . . 18
1.5.2 Metóda súčtov . . . 19
1.6 Základné výpočtové modely . . . 21
1.6.1 (M)RAM . . . 21
1.6.2 Základný model Turingovho stroja . . . 23

2 Základné metódy tvorby efektívnych algoritmov **27**
2.1 Rozdeľuj a panuj . . . 27
2.1.1 Násobenie veľkých čísel . . . 29
2.1.2 Strassenov algoritmus násobenia matíc . . . 30
2.1.3 Vyhľadávanie k-teho najmenšieho prvku . . . 30
2.1.4 Násobenie Booleovských matíc . . . 31
2.2 Dynamické programovanie . . . 33
2.2.1 Súťaž dvoch družstiev . . . 34
2.2.2 Násobenie n matíc . . . 35
2.2.3 Konštrukcia optimálneho binárneho vyhľadávacieho stromu . . . 36
2.2.4 0/1 Plnenie batoha (0/1 Knapsack problem) . . . 37
2.3 Greedy algoritmy . . . 39
2.3.1 Plnenie batoha (s racionálnymi koeficientami) . . . 40
2.3.2 Úlohy s terminovaním . . . 41
2.3.3 Optimálne zlučovanie súborov . . . 43
2.3.4 Minimálna kostra. . . . 44

3 Metódy založené na prehľadávaní stavového priestoru **47**
3.1 Backtracking-prehľadávanie s návratom . . . 49
3.2 LC Branch and Bound . . . 51
3.3 Problém obchodného cestujúceho. . . . 52
3.4 0/1 Plnenie batoha . . . 54

4 Rozhodnuteľnosť **57**
4.1 Univerzálny TS 57
4.2 Rozhodnuteľné a nerozhodnuteľné problémy 59
4.2.1 Zastavenie TS 62
4.2.2 Postov korešpondenčný problém 63

5 Výpočtové modely a vzťahy medzi nimi **67**
5.1 k-páskový Turingov stroj 67
5.1.1 Redukcia času a redukcia pásky 71
5.2 Vzťah DTS a MRAM 73
5.2.1 Prvá počítačová trieda 76

6 Zložitostné triedy **77**
6.1 Niektoré metódy dolných odhadov 77
6.1.1 Prechodová postupnosť a dolný odhad na čas 77
6.1.2 Prechodová matica a dolný odhad na pamäť 80
6.2 Hierarchia času a priestoru 82
6.2.1 Priestorová hierarchia 83
6.2.2 Hierarchia času 84
6.3 Nedeterministický TS 85
6.4 Vzťahy medzi zložitostnými triedami 87
6.5 Uzavretosť nedeterministického priestoru na komplement 90

7 Trieda NP a NP-úplnosť **93**
7.1 Redukcie, NP-úplnosť 94
7.2 NP-úplnosť problému splniteľnosti BF 95
7.3 Niektoré NP-úplné problémy 99
7.4 P vs. NP* 103
7.4.1 Vzťah tried P, NP 104
7.4.2 Relativizácia problému $P \stackrel{?}{=} NP$ 105

Kapitola 1

Úvod

1.1 Čo je to Computer Science?

Namiesto úvodu pár myšlienok z úvodu knihy Juraja Hromkoviča: Theoretical Computer Science – An Introduction to Automata, Computability, Complexity, Algorithmics, Randomization, Communication, and Cryptography

Na označenie vedy, ktorej sa venuje tento materiál, sa najčastejšie používa označenie Informatika (z anglického computer science, resp. informatics). Každý, kto študuje alebo sa zaoberá touto vednou disciplínou, by sa z času na čas mal zamyslieť nad tým, ako by zadefinoval informatiku, mal by premýšľať o jej prínose pre vedu, vzdelávanie, každodenný život. Je dôležité, aby sme si uvedomili, že získavanie čoraz väčšieho množstva poznatkov o vedeckej disciplíne, prehlbovanie pochopenia jej podstaty vždy vedie k vývoju nášho názoru na úlohu tejto vedy v kontexte všetkých vedeckých disciplín. Je preto najmä pre študentov nesmierne dôležité, aby si neustále premietali svoje vnímanie informatiky ako vedy. Trúfneme si vyprovokovať konflikt medzi vašim terajším vnímaním informatiky a stanoviskom, prezentovaným v tomto úvode; podnietime diskusiu, ktorá môže viesť k vývoju vášho chápania informatiky.

Pokúsme sa najskôr zodpovedať otázku

"Čo je to informatika?"

Je ťažké poskytnúť exaktnú a úplnú definíciu tejto vedeckej disciplíny. Všeobecne akceptovaná definícia je nasledovná:

> *"Informatika je náuka o algoritmickom spracovávaní, reprezentácii, ukladaní a prenose informácií"*

Podľa tejto definície sú hlavnými objektami výskumu informatiky ako vedeckej disciplíny informácia a algoritmus. Táto definícia však zanedbáva úplné odhalenie podstaty a metodológie informatiky. Ďalšou otázkou o podstate informatiky je

> *"Ku ktorej vedeckej disciplíne informatika patrí? Je to metadisciplína ako matematika a filozofia, prírodná veda alebo inžinierska disciplína?"*

Odpoveď na túto otázku slúži nielen na identifikáciu objektu výskumu, musí tiež určiť metodológiu a príspevok informatiky ako vedy. Odpoveďou je, že informatika nemôže byť jednoznačne priradená k žiadnej z týchto disciplín. Informatika v sebe zahŕňa aspekty matematiky, prírodných vied ale aj inžinierskych disciplín. V krátkosti vysvetlíme, prečo.

Podobne ako filozofia a matematika, informatika skúma všeobecné kategórie, ako

determinizmus, nedeterminizmus, náhodnosť, informácia, pravda, nepravda, zložitosť, jazyk, dôkaz, znalosť, komunikácia, aproximácia, algoritmus, simulácia, atď.

a prispieva k ich pochopeniu. Informatika vrhla nové svetlo na tieto kategórie, priniesla nový význam mnohým z nich.
Prírodná veda, na rozdiel od filozofie a matematiky, skúma konkrétne prírodné objekty a procesy, určuje hranicu medzi možným a nemožným, skúma kvantitatívne vzťahy prírodných procesov. Modeluje, analyzuje a dokazuje vierohodnosť hypotetických modelov experimentmi. Tieto aspekty sú bežné aj v informatike. Objektami sú informácie a algoritmy (programy, počítače) a skúmanými procesmi sú (reálne existujúce) výpočty. Presvedčíme sa o tom sledovaním vývoja informatiky. Historicky prvou dôležitou výskumnou otázkou bolo:

> *"Existujú dobre definované problémy, ktoré nemôžu byť riešené automaticky (počítačom, bez ohľadu na silu súčasných a budúcich počítačov)?"*

Snaha o zodpovedanie tejto otázky viedla k základom informatiky ako nezávislej vedy. Odpoveď na otázku je kladná. V súčasnosti sme si o mnohých praktických problémov, ktoré by sme radi riešili algoritmicky, vedomí toho, že algoritmicky riešiteľné nie sú. Tento záver je založený na čisto matematickom dôkaze algoritmickej neriešiteľnosti (inými slovami, na dôkaze neexistencie algoritmu riešiaceho daný problém) a nie na tom, že sa žiadne algoritmické riešenie doteraz nenašlo.
Po tom, ako bola vyvinutá metóda na klasifikáciu problémov podľa ich riešiteľnosti, začali si ľudia klásť nasledujúcu vedeckú otázku:

> *"Ako ťažké sú konkrétne algoritmické problémy?"*

Pritom obtiažnosť nemeriame obtiažnosťou nájsť algoritmické riešenie, či veľkosťou vytvoreného programu. Obtiažnosť meriame množstvom práce potrebnej a postačujúcej k tomu, aby sme pre daný vstup algoritmicky vypočítali riešenie. Dozvedáme sa o existencii ťažkých problémov, na riešenie ktorých treba energiu presahujúcu energiu celého vesmíru. Existujú také algoritmicky riešiteľné problémy, pre ktoré by výpočet ľubovoľného programu na ich riešenie vyžadoval viac času ako uplynul od "Veľkého tresku". Takže číra existencia programu na riešenie nejakého problému ešte nezaručuje, že je tento problém prakticky riešiteľný.

Pokusy o klasifikáciu problémov na prakticky riešiteľné (tractable) a prakticky neriešiteľné viedli k najfascinujúcejším vedeckým objavom teoretickej informatiky.
Ako príklad si vezmime pravdepodobnostné (randomized) algoritmy. Väčšina programov (algoritmov) tak, ako ich poznáme, je deterministická. Determinizmom rozumieme to, že program a vstup úplne určujú všetky kroky spracovávania problému. V každom okamihu je nasledujúca akcia programu jednoznačne určená a závisí iba od momentálnych údajov. Pravdepodobnostné algoritmy môžu nasledujúcu akciu programu vyberať z viacerých možností. Prácu pravdepodobnostného algoritmu si možno predstaviť tak, že algoritmus z času na čas hádže mincou, aby určil nasledujúci krok, napr. vybral nasledujúcu stratégiu pri hľadaní korektnej odpovede. Pravdepodobnostný program tak môže mať pre jeden vstup niekoľko rôznych výpočtov. Na rozdiel od deterministických programov, ktoré vierohodne vrátia pre každý vstup správny výsledok, pravdepodobnostné programy môžu poskytnúť aj nesprávny výsledok. Cieľom je znižovať/potláčať pravdepodobnosť takýchto nesprávnych výsledkov, čo za určitých podmienok znamená znižovať počet nesprávnych výpočtov.
Na prvý pohľad sa pravdepodobnostné programy môžu zdať, na rozdiel od de-

terministických programov, nespoľahlivé. Prečo ich teda potrebujeme? Existuje veľa reálnych problémov, ktorých riešenie najlepšími známymi algoritmami vyžaduje viac počítačovej práce ako je realisticky možné. Takéto problémy sú prakticky neriešiteľné. Môže sa však stať zázrak: tým zázrakom môže byť pravdepodobnostný algoritmus, ktorý rieši problém za niekoľko minút s minimálnou pravdepodobnosťou chyby jednej trilióntiny. Môžeme takýto program zavrhnúť ako nespoľahlivý? Deterministický program, ktorého výpočet trvá niekoľko dní, je menej spoľahlivý ako pravdepodobnostný program bežiaci niekoľko minút, pretože pravdepodobnosť výskytu hardverovej chyby počas 24 hodín je oveľa väčšia ako pravdepodobnosť chyby rýchleho pravdepodobnostného programu. Konkrétnym prípadom prakticky nesmierne dôležitého problému je testovanie prvočíselnosti. Vo všadeprítomných kryptografických systémoch založených na verejných kľúčoch je nevyhnutné, aby sa generovali veľké (500 ciferné) prvočísla. Prvé deterministické algoritmy na testovanie prvočíselnosti boli založené na deliteľnosti vstupu n. Už samotný počet prvočísel, menších ako $\sqrt{n}$, je pre tak veľké čísla väčší, ako počet protónov in the universe. Takéto deterministické algoritmy sú teda prakticky nepoužiteľné. Nedávno sa objavil nový deterministický algoritmus na testovanie prvočíselnosti, ktorého zložitosť je $O(m^{12})$, kde m je dĺžka binárneho zápisu čísla n. Na testovanie 500ciferného čísla však tento algoritmus vyžaduje vykonanie viac ako 10^{32} počítačových inštrukcií; ani na najrýchlejších počítačoch by čas od Veľkého tresku nebol dostatočný na realizáciu tohto výpočtu. Máme však niekoľko randomized algoritmov pomocou ktorých môžeme otestovať prvočíselnosť tak veľkých čísel na bežných PC v priebehu niekoľkých minút, dokonca sekúnd.

Iným exemplárnym príkladom je komunikačný protokol pre porovnanie obsahu dvoch databáz, ktoré sú uložené na dvoch vzdialených počítačoch. Matematicky sa dá dokázať, že každý deterministický protokol, ktorý testuje ekvivalenciu ich obsahov, vyžaduje, aby sa vymenilo toľko bitov, koľko ich je uložených v databázach. Pre databázu veľkosti 10^{16} by to bolo únavné. Pravdepodobnostným komunikačným protokolom môžeme testovať ekvivalenciu dvoch databáz výmenou správy dĺžky približne 2000 bitov. Pravdepodobnosť chyby je pritom menšia ako jedna trilióntina.

Ako je to možné? Bez využitia základných znalostí informatiky sa to vysvetľuje ťažko. Hľadanie vysvetlenia sily pravdepodobnostných algoritmov je fascinujúci výskumný projekt, ktorý zachádza do najhlbších základov matematiky, filozofie a prírodných vied. Príroda je náš najlepší učiteľ a náhoda hrá v prírode oveľa dôležitejšiu úlohu, ako si dokážeme pripustiť. Informatici môžu vymenovať mnoho systémov, ktorých požadované charakteristiky a správanie sa dajú dosiahnuť iba pomocou randomizície. V takýchto prípadoch sú všetky dostupné deterministické systémy zložené z biliónov podsystémov, ktoré musia interagovať korektne. Tak komplexný systém, vysoko závislý od veľkého počtu komponent, je nepraktický. V prípade výskytu chyby by bolo takmer nemožné ju odhaliť. Netreba ani hovoriť, že cena za vývoj takéhoto systému by bola tiež astronomická. Na druhej strane môžeme skonštruovať malý randomized systém s požadovanými vlastnosťami. Vzhľadom na malú veľkosť sú takéto systémy lacné a funkčnosť ich komponent sa ľahko testuje. Kľúčovým pritom ostáva, že pravdepodobnosť chyby takého systému je tak minimálna, až je zanedbateľná.

Popri doteraz prezentovaných vedeckých aspektoch je pre mnohých vedcov informatika typicky problémovo orientovaná a prakticky inžinierska disciplína. Informatika nielenže zahŕňa technické aspekty ako:

> *organizácia procesov (fázy, milestones, dokumentácia), formulácia strategických cieľov a obmedzení, modelovanie, popis, špecifikácia, zabezpečenie kvality, testovanie, integrácia do existujúcich systémov, viacnásobné použitie, podporné nástroje,*

Zahŕňa tiež aspekty manažmentu ako:

> *organizácia a vedenie tímu, odhad cien, plánovanie, produktivita, manažment kvality, odhad časových plánov a termínov, product release, podmienky kontraktov a marketing.*

Informatici by mali byť aj naozaj pragmatickými praktikmi. Pri konštrukcii komplexného softverového alebo hardverového systému musí človek často robiť rozhodnutia na základe vlastnej skúsenosti, pretože nemá možnosť modelovať a analyzovať komplexnú realitu.
Na základe našej definície informatiky možno nadobudnúť dojem, že štúdium informatiky je príliš obtiažne. Človek potrebuje matematické vedomosti, chápanie uvažovania v prírodných vedách a navyše, schopnosť pracovať ako inžinier. Toto naozaj môžu byť vysoké požiadavky, je to však tiež veľká výhoda tohto typu vzdelania. Hlavnou nevýhodou súčasnej vedy je jej vysoká špecializácia, ktorá vedie k vývoju malých nezávislých disciplín. Každá z nich si buduje svoj vlastný jazyk, ktorý je často nezrozumiteľný dokonca aj pre vedcov z príbuznej oblasti. Zašlo to tak ďaleko, že spôsob štandartnej argumentácie jednej oblasti sa považuje za povrchný a neprípustný v inej oblasti. Spomaľuje to vývoj interdisciplinárneho výskumu. A informatika je v svojej podstate interdisciplinárna. Sústreďuje sa na hľadanie riešenia problémov všetkých oblastí vedy a každodenného života; všade tam, kde je predstaviteľné použitie počítača. Súčasne s tým vyvíja široké spektrum metód, počnúc precíznymi matematickými metódami, končiac inžinierskym "know-how", založenom na skúsenosti. Možnosť súbežného učenia sa rôznych jazykov rôznych oblastí a rôznym spôsobom uvažovania v jednej disciplíne, to je najcennejší dar, ktorý študenti informatiky dostávajú.

1.2 Fascinujúca teória

Kniha je úvodom do základov teoretickej informatiky. Teoretická informatika je fascinujúca vedecká disciplína. Svojimi veľkolepými výsledkami a vďaka svojej veľkej interdisciplinarite prispela veľkým dielom k nášmu pohľadu na svet. Ako by štatistiky potvrdili, nie je teoretická informatika najobľúbenejším predmetom študentov.
Niekoľko dôvodov pre jej štúdium:

filozofická hĺbka

Existujú problémy, ktoré sú algoritmicky neriešiteľné? Ak áno, kde leží hranica medzi riešiteľnými a neriešiteľnými problémami?

Sú nedeterministické a náhodou riadené procesy schopné riešiť viac ako deterministické?

Ako definujeme obtiažnosť(zložitosť) problému?

Kde sú hranice praktickej riešiteľnosti?

Čo je to matematický dôkaz? Je ťažšie algoritmicky nájsť dôkaz alebo algoritmicky overiť jeho platnosť?

Ako definujeme náhodné objekty/náhodnosť?

Bez pojmov teoretickej informatiky by sme tie problémy nemohli ani poriadne sformulovať, ani na ne dať odpoveď.

aplikovateľnosť a veľkolepé výsledky

TI súvisí s praxou. Poskytuje metodológie, ktoré aplikujeme pri počiatočnom návrhu, ale aj tie, ktoré využijeme počas celého procesu návrhu, dokazovania, implementácie. Existuje nemálo optimalizačných úloh, kde relaxácia požiadaviek vedie k efektívnejšiemu riešeniu. Veríte, že možno niekoho presvedčiť o znalosti tajného heslo bez toho, aby sme ho povedali? Veríte, že dve osoby môžu zistiť, kto je starší bez toho, aby prezradili svoj vek? Veríte, že môžeme overiť platnosť niekoľko tisíc

stránkového dôkazu bez toho, aby sme ho celý čítali, ale videli iba niektoré jeho náhodne zvolené úseky? Všetko toto sa dá...

živostnosť vedomostí

Zatiaľ čo zhruba polovica vedomostí o software a hardware je po 5 rokoch neaktuálna, metodologické výsledky TI pretrvávajú niekoľko desiatok rokov...

interdisciplinarita

TI je vo svojej podstate interdisciplinárna, nachádzame jej uplatnenie v mnohých iných oblastiach - genetika, medicínska diagnostika, optimalizácia v ekonomických a technických disciplínach, automatické rozpoznávanie reči, prehľadávanie priestoru,...
Nie je to jednostranná spolupráca. TI tiež profituje z tých iných disciplín: kvantova fyzika a kvantove počítače; DNA výpočty; kalenie a metóda simulovaného žíhania;...

spôsob myslenia

TI podporuje vytváranie a analyzovanie matematických modelov, vytváranie pojmov a metodológii na riešenie problémov.

1.3 A teraz už môj úvod

Pri pohľade dozadu si v súvislosti s efektívnou riešiteľnosťou problémov môžme všimnúť niekoľko medzníkov

- Významným krokom bolo formalizovanie pojmu *algoritmus*zhruba v 30-tych rokoch minulého storočia. Vďačíme za to definovaniu *abstraktného* výpočtového modelu - *Turingovho stroja* (TS), ktorý bol všeobecne akceptovaný ako "definícia" pojmu algoritmus. Vďaka tomuto pojmu dochádza k deleniu problémov na *riešiteľné* a neriešiteľné.
- Po vymedzení triedy riešiteľných problémov sa pozornosť presúva k vymedzeniu triedy *prakticky riešiteľných* problémov. Praktická riešiteľnosť sa (väčšinou) vzťahuje na časové nároky potrebné na riešenie toho-ktorého problému.

  Za prakticky riešiteľný sa dlho považoval problém, ktorý sa dá riešiť *sekvenčne* v deterministickom polynomiálnom čase. Na úrovni abstraktného modelu označujeme túto triedu P. Zatiaľ čo pojem riešiteľného problému je zrejme nemenný, definícia praktickej riešiteľnosti sa posúva. V dnešných dňoch sú už za prakticky riešiteľné považované aj problémy, ktoré sú riešiteľné v polynomiálnom čase pravdepodobnostnými algoritmami, aproximačnými algoritmami, rýchlymi heuristickými algoritmami, paralelnými algoritmami, ...
- Pozornosť sa sústredila na porovnávanie problémov z hľadiska náročnosti ich realizácie, skúmajú sa pritom rôzne miery zložitosti (popisná, čas, pamäť, počet porovnaní,...)

Pozrime sa na problematiku z pohľadu teórie a praxe.

Zložitosť konkrétnych výpočtových úloh sa zaoberá riešením konkrétnych výpočtových problémov. Všímame si zložitosť riešenia, ktoré je vyjadrené/vnímané ako algoritmus v nejakom pseudo-programovacom jazyku. Popri hľadaní čo najlepších algoritmov na riešenie problémov, ktoré nás zaujímajú, sa snažíme o získanie všeobecnejších poznatkov. Medzi takéto rozhodne patria

– metódy tvorby efektívnych algoritmov; stretneme sa s metódou rozdeľuj a panuj, dynamickým programovaním, pažravými algoritmami, prehľadávacími algoritmami,...

– dokazovanie zložitosti konkrétnych algoritmov.

Abstraktná teória zložitosti - sa zaoberá riešením problémov trochu inak. V pozadí stoja abstraktné výpočtové modely. Snažíme sa o získanie takých poznatkov o výpočtoch, ktoré sú invariantné vzhľadom na výber počítača (z danej rozumnej triedy). Definovanie toho, čo je to "rozumný počítač" je samo o sebe nie celkom triviálny problém. *Prvá počítačová trieda* obsahuje všetky výpočtové modely, ktoré sú polynomiálne ekvivalentné[1] deterministickému TS. Uvedomme si, že všetky počítače z prvej počítačovej triedy definujú triedu prakticky riešiteľných problémov—teda problémov, riešiteľných v polynomiálnom čase, rovnako. Problém je riešiteľný v polynomiálnom čase bez ohľadu na výber počítača z tejto triedy. *Druhá počítačová trieda* obsahuje všetky také modely, ktoré vedia efektívne využívať priestor. To znamená, že trieda problémov, ktoré sa dajú riešiť v priestore $f(n)$ sa rovná triede problémov, ktoré sa dajú riešiť v čase exponenciálnom od $f(n)$.

Na problematiku sa môžme pozrieť aj na základe spracovania informácie. To vedie k vymedzeniu teórie *sekvenčných* a *paralelných* výpočtov.

[1]Modely sú polynomiálne ekvivalentné, ak zložitosť riešenia každého problému je na oboch rovnaká až na polynóm.

1.4 Základné pojmy a definície

Začnime opakovaním pojmov, s ktorými ste sa už iste stretli na predchádzajúcich prednáškach, napr. na prednáške "Dátové štruktúry a algoritmy".

Problém

- zobrazenie z množiny vstupných reťazcov do množiny výstupných reťazcov
- konečné problémy (z konečnej množiny do konečnej množiny) - napr. booleovské funkcie
- rozhodovacie problémy (odpoveď je áno-nie - je daný reťazec znakov syntakticky správnym programom v danom programovacom jazyku, existuje v danej množine objekt s danou vlastnosťou, sú dve (potenciálne nekonečné) množiny rovnaké,...)
- ...

Algoritmus

- pôvod slova algoritmus je "al-kawarizmi" v mene Perzského matematika(9. storočie)
- konečný návod na riešenie problému s použitím daných elementárnych operácií. Vyžadujeme, aby každá z elementárnych operácií mala presne špecifikovaný vzťah medzi vstupom a výstupom, aby bola vykonateľná v konečnom čase a konečnej pamäti (*strojovo a jazykovo nezávislé inštrukcie*)

Program

- implementácia algoritmu
- postupnosť korektne zapísaných inštrukcií, ktoré majú byť vykonané na počítači
- tieto inštrukcie sú zapísané v programovacom jazyku

1.4.1 Analýza zložitosti

Väčšinou je našou snahou nájsť *čo najlepší algoritmus* na riešenie skúmaného problému. Čo to ale znamená najlepší? Pozor, nejde o správnosť, kvalitu riešenia[2]. Kvalitou algoritmu budeme rozumieť jeho zložitosť. Lenže na riešenie jedného problému existuje viacero prístupov, ktoré vedú k algoritmom rôznej zložitosti. Aká je teda zložitosť nášho problému?

Analýza algoritmu: využitie matematických metód k predikcii času a pamäte potrebnej na realizáciu algoritmu (pri zbehnutí programu)

Uvažujme *mieru zložitosti* X (väčšinou čas, resp. pamäť, počet aritmetických operácií,..) a *model* M (modelom rozumieme vyšpecifikovanie množiny elementárnych inštrukcií, ktoré model poskytuje).

Zložitosť algoritmu A je funkcia veľkosti vstupných dát udávajúca množstvo miery zložitosti X spotrebovanej algoritmom A pri riešení problémov daného rozsahu.

$X_A^M(w)$ — množstvo miery zložitosti X spotrebovanej algoritmom A
pri spracovaní vstupu w

[2]Iná situácia je pri optimalizačných problémoch, keď tzv. aproximačné algoritmy nemusia dávať presné - optimálne - riešenie. Vtedy hovoríme o kvalite riešenia a zložitosti algoritmu

$$X_A^M(n) \quad \begin{cases} max_{w,|w|=n}\{X_A^M(w)\}, & \text{zložitosť najhoršieho prípad} \\ min_{w,|w|=n}\{X_A^M(w)\}, & \text{zložitosť najlepšieho prípadu} \end{cases}$$

Keďže najlepší aj najhorší prípad sú vlastne extremálnymi hodnotami, nedávajú celkom presnú predstavu o zložitosti pre konkrétny prípad. Preto je lepšou charakteristikou **zložitosť priemerného prípadu**

$$\overline{X_A^M(n)} = \sum_{w_i} p(w_i) X_A^M(w_i), \text{ kde}$$

sumu počítame cez všetky slová w_i dĺžky n a $p(w_i)$ je pravdepodobnosť výskytu konkrétneho vstupu w_i. Problémom však je, že

– často nepoznáme pravdepodobnostné rozdelenie vstupných údajov
– samotná suma sa ťažko počíta.

Preto niekedy predpokladáme rovnomerné rozdelenie a počítame priemernú zložitosť ako normálny aritmetický priemer

$$\overline{X_A^M(n)} = \frac{\sum p(w_i) X_A^M(w_i)}{|W_n|}, \text{ kde } W_n \text{ je množina všetkých vstupov veľkosti } n.$$

Uvedomme si význam parametra M! Predpokladajme, že X označuje počet aritmetických operácií. Ako sa zmení zložitosť klasického algoritmu násobenia, ak za základnú aritmetickú operáciu budeme považovať násobenie jednociferným číslom v porovnaní s prípadom, keď základnou aritmetickou operáciou je násobenie ľubovoľne veľkých čísel?

Časovú zložitosť budeme väčšinou označovať T.

Čo je dobrá miera zložitosti?

- **CPU čas**
  strojovo závislá (pre rôzne architektúry môže byť rôzna)
- **počet (vykonateľných) príkazov**
  závisí od programovacieho jazyka, od programátorovho štýlu,..
- **počet opakovaní cyklu**
  závisí od jazyka; dĺžka realizácie jedného behu cyklu sa mení

\+ **počet základných operácií**
  fixujeme množinu elementárnych inštrukcií(model); toto je strojovo nezávislé

problém	**počítané základné operácie**
Vyhľadávanie v zozname	Porovnanie
Násobenie matíc	Násobenie (jednotlivých prvkov)
Triedenie	Porovnanie
Prehľadávanie (BS)	Prehľadanie vrchola

Prečo potrebujeme analyzovať algoritmy?
analýza

- lepšie odráža skutočnosť ako experimentovanie
- umožňuje nám porovnávať kvalitu algoritmov bez ich implementácie

- umožňuje odhad času a pamäťových nárokov implementácie
- umožňuje lepšie pochopenie algoritmu, identifikáciu rýchlych a pomalých častí

Algoritmus	T(n)	max. rozsah	problému
		sek	min
A1	n	1000	60 000
A2	nlogn	141	4 893
A3	n^2	31	244
A4	n^3	10	39
A5	2^n	9	15

Predstavme si, že počítač 10-násobne urýchlime (rýchlosť M je menšia ako rýchlosť N). Ako sa zmení maximálny rozsah problému, ktorý môžme v danom čase riešiť?

Algoritmus	T(n)	max. na M	max. na N
A1	n	S1	10*S1
A2	nlogn	S2	10*S2
A3	n^2	S3	3.16*S3
A4	n^3	S4	2.15*S4
A5	2^n	S5	S5+3.3

Asymptotická zložitosť

Keďže konštanty (rozumnej veľkosti) nemajú podstatný vplyv na celkovú efektívnosť algoritmu, uspokojíme sa často len s asymptotickým odhadom zložitosti (zanedbáme konštanty). Preto sa v súvislosti so zložitosťou algoritmu väčšinou stretneme s pojmom asymptotickej zložitosti. Zopakujme si definície asymptotík:

$g(n) = O(f(n))$	$\exists c, n_o \in N \; \forall n \geq n_0 \;\; g(n) \leq cf(n)$
$g(n) = \Omega(f(n))$	$\exists c, n_o \in N \; \forall n \geq n_0 \;\; g(n) \geq cf(n)$
$g(n) = \Theta(f(n))$	$g(n) = O(f(n)) \wedge g(n) = \Omega(f(n))$
$g(n) = o(f(n))$	$\lim \frac{g(n)}{f(n)} = 0$

A teraz sa môžme vrátiť k tomu, aby sme hovorili o **zložitosti problému**. Ako súvisí zložitosť $X_A^M(n)$ algoritmu A riešiaceho daný problém P so zložitosťou $X_P^M(n)$ daného problému?

- $X_p^M(n) \leq X_A^M(n)$ - každý algoritmus riešiaci daný problém poskytuje **horný odhad zložitosti problému**; pokiaľ máme len tento odhad, nevieme posúdiť kvalitu nami získaného algoritmu.
- $X_p^M(n) \geq f(n) \Leftrightarrow$ neexistuje algoritmus (z danej triedy M) riešiaci problém P so zložitosťou menšou ako $f(n)$. Ak sa nám také niečo podarí ukázať, potom $f(n)$ je **dolný odhad zložitosti problému**.

Keď v nerovnici $f(n) \leq X_p^M(n) \leq X_A^M(n)$ sú hranice $f(n)$ a $X_A^M(w)$ blízko pri sebe, podarilo sa nám nájsť dobrý algoritmus. Ťažko možno očakávať, že by sme dosiahli úplnú rovnosť.

$$\text{algoritmus A je} \begin{cases} \textbf{optimálny}, ak & f(n) = X_A^M(n); \\ \\ \textbf{asymptoticky optimálny}, ak & X_p^M(n) = \Theta(X_A^M(n)) \end{cases}$$

1.4.2 Analýza problému triedenia

Uvažujme problém triedenia. Pýtame sa, aká je zložitosť tohto problému v triede tzv. porovnávacích algoritmov (algoritmov, ktoré nevedia "rozoberať" kľúče, ale jedinú informáciu môžu získať pomocou porovnania dvoch kľúčov).

Všimnime si najprv horný odhad - budeme uvažovať jeden z najznámejších algoritmov za predpokladu, že vstupné hodnoty sú rôzne (ako sa algoritmus a nasledujúca analýza zmenia, keď tento predpoklad odstránime?)

Algorithm QSort(S)
vyber $a \in S$;
$S1 = \{b \in S; b < a\}$;
$S2 = \{b \in S; b > a\}$;
return(QSort(S1) . a . QSort(S2))

Čo vieme povedať o zložitosti tohto algoritmu? Meranú počtom porovnaní ju vo všeobecnosti vyjadríme nasledovne:

$$T(|S|) = T(|S1|) + T(|S2|) + n$$

Ľahko vidíme, že zložitosť podstatne závisí od konkrétnych veľkostí množín S1, S2. Takže zložitosť jednotlivých zaujímavých prípadov dostaneme jednoduchou analýzou vzťahu veľkostí množín S1, S2.

Najhorší prípad nastáva, ak pri každom výbere prvku a zvolíme prvok, ktorý je najmenší alebo najväčší v množine; má to za následok, že zložitosť najhoršieho prípadu je $O(n^2)$.

Skúsme sa zamyslieť nad *priemerným prípadom*. Bez ohľadu na to, aké sú skutočné hodnoty triedených prvkov, na zložitosť algoritmu má najväčší vplyv poradie (po usporiadaní množiny) prvku a - tento má totiž vplyv na mohutnosti množín S1 , S2. Preto zložitosť priemerného prípadu vyjadríme nasledovne:

$$\overline{T(n)} \leq \frac{1}{n}\sum_{i=1}^{n}(T(i-1) + T(n-i) + n)$$
$$T(0) = T(1) = b \quad T(i) = 0 \text{ pre } i < 0$$

Ukážeme, že riešením je $T(n) \approx nlogn$:

$$\begin{aligned} \overline{T(n)} &\leq \frac{1}{n}\sum_{i=0}^{n}(T(i) + T(n-i-1) + n) \\ &= cn + \frac{1}{n}\{T(0) + T(n-1) + T(1) + T(n-2) + \dots T(n-1) + T(0))\} \\ &= cn + \frac{2}{n}\sum_{i=0}^{n-1} T(i) \end{aligned}$$

Dôkaz spravíme matematickou indukciou.
IP: $T(n) \leq knlogn, k = 2c + 2b$

$T(2) \leq cn + \frac{2}{n}\sum_{i=0}^{n}(T(i)) = 2c + \frac{2}{2}(T(0) + T(1)) = 2c + 2b$

$\overline{T(n)} \leq cn + \frac{2}{n}\{T(0) + T(1)\} + \frac{2}{n}\sum_{i=2}^{n-1} kilogi = cn + \frac{4b}{n} + \frac{2k}{n}\sum_{i=2}^{n-1} ilogi$

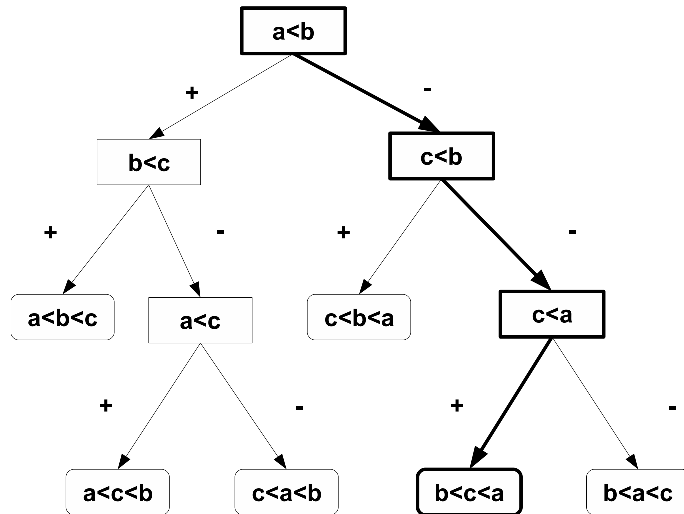
$$\text{Keďže } \sum_{i=2}^{n-1} ilogi \leq \int_2^n xlogxdx = \left[\frac{1}{2}x^2logx - \frac{x^2}{4}\right]_2^n = \left(\frac{1}{2}n^2logn - 2 - \frac{n^2}{4} + 1\right) \leq \frac{1}{2}n^2logn - \frac{n^2}{4}$$

tak $\overline{T(n)} \leq \frac{4b}{n} + \frac{2k}{n}\sum_{i=2}^{n-1} ilogi \leq \underbrace{cn + \frac{4b}{n} - \frac{kn}{2}} + knlogn \leq knlogn$

Uvedomte si, že pre $n \geq 2, k = 2c + 2b$ je $cn + \frac{4b}{n} - \frac{kn}{2} \leq 0$

Ostalo nám ešte nájsť **dolný odhad** na zložitosť problému triedenia. Musíme vyargumentovať, koľko porovnaní minimálne musí použiť každý triediaci algoritmus[3]. Pomôžeme si abstraktným modelom; využijeme rozhodovací strom. Čo to je?

Rozhodovací strom je neuniformný výpočtový model - pre každý rozsah vstupu sa konštruuje iný rozhodovací strom. Rozhodovací strom pre triedenie n-prvkovej množiny prvkov označíme RS(n). Opäť predpokladáme, že vstupné hodnoty sú rôzne. Ako sa RS(n) zmení, ak tento predpoklad vynecháme?



Obrázok 1.1: Rozhodovací strom pre triedenie 3 prvkov. Zvýraznená cesta odpovedá vstupným hodnotám $a = 7, b = 2, c = 3$

- vnútorné vrcholy RS(n) obsahujú porovnanie $?a \leq b?$
- listy RS(n) obsahujú permutáciu $a(i_1), ..., a(i_n)$ vstupných hodnôt (resp. správny výsledok pre daný vstup)

[3]z triedy porovnávacích algoritmov

Ako prebieha výpočet? "Postavíme sa" do koreňa RS(n). V prípade kladnej odpovede na porovnanie "postupujeme" doľava, v prípade zápornej odpovede doprava. Výsledná permutácia (odpoveď) je v liste, do ktorého sme sa dostali.

Zložitosť konkrétneho prípadu je rovná dĺžke realizovanej cesty v rozhodovacom strome. Najlepší prípad je najkratšia cesta od koreňa do listu, najhorší prípad je hĺbka stromu. Takže sa presunieme k analýze binárnych stromov.

Dolný odhad pre zložitosť problému triedenia:

- počet listov každého rozhodovacieho stromu triediaceho n-prvkovú množinu je aspoň $n!$
- počet listov binárneho stromu hĺbky h je nanajvýš 2^h
- počet porovnaní je pre konkrétny rozhodovací strom rovný jeho hĺbke
- pre ľubovoľný algoritmus A možno pre každé n zostrojiť RSA(n)

$$n! \leq \text{počet listov RSA(n)} \leq 2^h \text{ , kde h je hĺbka RSA(n)}$$

$$\Downarrow$$

$$h \geq logn! \approx c * nlogn$$

Keďže tento vzťah platí pre ľubovoľný algoritmus, je cnlogn dolným odhadom pre zložitosť problému triedenia meranú počtom porovnaní.

□

1.4.3 Dolný odhad pre problém minMax

problém minMax

vstup: n-prvková množina S s definovaným usporiadaním
výstup: maximálny a minimálny prvok množiny S

Venujme sa dolnému odhadu (zamyslite sa nad efektívnymi algoritmami).

- V prvom rade si treba ujasniť, čo budeme považovať za mieru zložitosti. Bude to opäť počet porovnaní - takéto algoritmy sú nezávislé od skutočnej množiny prvkov; treba len správne realizovať porovnanie.
- Pri dolnom odhade si opäť pomôžeme abstrakciou; tentokrát to bude stavový priestor. V každom okamihu riešenia priradíme rozpracovanej úlohe nejaký stav/charakterizáciu. Množina všetkých potenciálnych stavov potom tvorí stavový priestor.
- Počiatočnú situáciu charakterizuje nejaký stav (resp. množina stavov) a výstupná situácia tiež zodpovedá nejakému stavu, resp. množine stavov.
- Stav úlohy sa mení len pri vykonávaní nejakých operácií - v našom prípade sa stav mení následkom porovnania. Takto konkrétnemu priebehu vykonávania algoritmu odpovedá cesta v stavovom priestore.
- Dolný odhad na počet porovnaní získame argumentáciou o dĺžke cesty z počiatočného stavu do koncového stavu. Každé porovnanie môže zmeniť rozloženie prvkov v množinách, a teda stav. Keďže dĺžka cesty tvorí dolný odhad na počet porovnaní, budeme sa zaujímať o dĺžku cesty z počiatočného stavu do koncového. Ide o "hru" medzi nami a algoritmom. My sa snažíme cestu predlžovať a máme k tomu jediný prostriedok - voľbu vstupu. Naproti tomu algoritmus sa snaží cestu skracovať - robí to voľbou krokov (výberom prvkov, ktoré sa v tom ktorom okamihu porovnajú).

Stav (pre problém minMax) je štvorica **(a,b,c,d)**, kde

a je počet prvkov, ktoré sa ešte neporovnávali, A meno tejto množiny

b je počet prvkov, ktoré sa už porovnávali a v každom porovnaní boli väčšie, B meno tejto množiny

c je počet prvkov, ktoré sa už porovnávali a v každom porovnaní boli menšie, C meno tejto množiny

d je počet prvkov, ktoré sa už porovnávali a boli aj menšie aj väčšie, D meno tejto množiny

Stavový priestor množina $\{(a,b,c,d); a,b,c,d \in \{0,1,...,n\}, a+b+c+d=n\}$

Počiatočný stav je charakterizovaný štvoricou $(n,0,0,0)$

Koncový stav je charakterizovaný štvoricou $(0,1,1,n-2)$

Keďže sa potrebujeme dostať z bodu (n,0,0,0) do bodu (0,1,1,n-2), musí

- prvá zložka (mohutnosť množiny A) klesnúť o n-2,
- tretia a štvrtá (mohutnosť množín B, C) stúpnuť o 1,
- posledná zložka (mohutnosť množiny D) stúpnuť o n-2.

Všimnime si, ako jedno porovnanie môže zmeniť stav úlohy:

	porovnávané prvky	výsledok porovnania	starý stav	(nový stav
1	$x \in A, y \in A$	$x < y$ $x > y$	(a,b,c,d)	(a-2, b+1, c+1, d) (a-2, b+1, c+1, d)
2	$x \in A, y \in B$	$x < y$ $x > y$	(a,b,c,d)	(a-1, b, c+1, d) a-1, b, c,d+1)
3	$x \in A, y \in C$	$x < y$ $x > y$	(a,b,c,d)	(a-1, b+1, c, d) a-1, b, c,d+1)
4	$x \in A, y \in D$	$x < y$ $x > y$	(a,b,c,d)	(a-1, b, c+1, d) (a-1, b, c,d+1)
5	$x \in B, y \in B$	$x < y$ $x > y$	(a,b,c,d)	(a, b-1, c, d+1) (a, b-1, c,d+1)
6	$x \in C, y \in C$	$x < y$ $x > y$	(a,b,c,d)	(a, b, c-1, d+1) (a, b, c-1,d+1)
7	$x \in B, y \in C$	$x < y$ $x > y$	(a,b,c,d)	(a, b-1, c-1, d+2) (a, b, c,d)
8	$x \in C, y \in D$	$x < y$ $x > y$	(a,b,c,d)	(a, b, c, d) (a, b, c-1,d+1)
9	$x \in B, y \in D$	$x < y$ $x > y$	(a,b,c,d)	(a, b-1, c, d+1) (a, b, c,d)

Ak si predstavíme, že porovnávané prvky sa po porovnaní presúvajú do príslušnej množiny, musí sa každý prvok, ktorý skončí v množine D, najprv presunúť do jednej z množín B,C a až potom do množiny D. Prvky, ktoré skončia v B, resp. C, sa zúčastnili aspoň jedného porovnania. Dolný odhad teda získame ako súčet počtu porovnaní, ktorými sa prvky presunú z množiny A plus počet porovnaní, ktorými sa prvky presunú z $B \cup C$ do D.

Uvedomme si, ako súvisí naša "hra"s tabuľkou, ktorá zobrazuje možné zmeny stavu. Algoritmus "vyberá"hrubý riadok, my vhodnou voľbou počiatočných hodnôt môžme ovplyvniť, ktorá z možností v zvolenom riadku nastala.

- Pretože algoritmu nemôžme zabrániť, aby prvky z množiny A nepresúval podľa bodu 1, na presun prvkov z A do B alebo C môžme počítať iba n/2 porovnaní - algoritmus vybral pre nás najhoršiu možnosť.
- Koľko porovnaní možno rátať na presuny do množiny D? Vidíme, že posledná zložka sa v jedinom prípade môže zvýšiť o 2 (prípad 7). Nie je ťažké si uvedomiť, že vieme skonštruovať taký vstup, aby každý prvok z množiny B bol väčší ako každý prvok z množiny C (ako?). V takejto situácii prípad 7 nikdy nenastane , a preto na zmenu poslednej súradnice treba minimálne n-2 porovnaní.

Dostávame teda

$$\text{počet porovnaní } \geq \frac{n}{2} + n - 2 = \frac{3n}{2} - 2.$$

□

1.5 Amortizovaná zložitosť

Vieme, že zložitosť je funkcia veľkosti/rozsahu vstupu. Pri analyzovaní zložitosti konkrétneho algoritmu väčšinou postupujeme tak, že najprv zanalyzujeme zložitosť jednotlivých operácií (často v najhoršom prípade) a potom sčítame cez všetky realizované operácie. Keď teda máme cyklus, spočítame (väčšinou) zložitosť trvania jednej iterácie a potom prenásobíme počtom opakovaní. Iným prístupom je tzv.*amortizovaná zložitosť*, keď analyzujeme postupnosť realizovaných operácií ako celok.

1.5.1 Metóda zoskupení

Princíp tejto metódy spočíva v tom, že jednotlivé operácie rozdelíme do skupín a potom analyzujeme skupinu ako celok. Aplikujme na príklad manipulácie so zásobníkom a s binárnym počítadlom.

Príklad 1.1 *Uvažujme údajovú štruktúru zásobník s tromi operáciami*

Push(S,x)
: *vloží do zásobníka S prvok x. Cena/zložitosť realizácie tejto operácie je 1*

Pop(S)
: *(deštruktívne) vráti vrchný prvok zásobníka S. Cena realizácie tejto operácie je tiež 1.*

Multipop(S,k)
: *vyberie zo zásobníka k prvkov, resp. zásobník vyprázdni, ak v ňom bolo menej ako k prvkov. Cena realizácie operácie je prirodzene k (resp. počet naozaj odstránených prvkov)*

Predpokladajme, že máme postupnosť n operácií, ktoré manipulujú so zásobníkom S. Zaujíma nás zložitosť realizácie tejto postupnosti.

Zrejme najhoršia/najťažšia operácia je $Multipop$, ktorého cena je nanajvýš n. Preto klasicky počítaná zložitosť

$$\text{počet realizovaných operácií} \times \text{zložitosť najťažšej} = n \cdot n = O(n^2)$$

Pri použití metódy zoskupení na problém manipulácie so zásobníkom rozdelíme jednotlivé operácie do dvoch skupín.

I	Push	dá sa vždy realizovať, vždy stojí 1, preto je celkový príspevok skupiny n
II	Pop, Multipop	nemôžme vybrať viac prvkov ako sme vložili, preto je sumárny príspevok tejto skupiny tiež n

Získali sme presnejší odhad zložitosti – $2n$

Príklad 1.2 *Uvažujme k-bitové binárne počítadlo realizované ako pole $A[0 \ldots k\text{-}1]$. Hodnota počítadla reprezentovaného poľom $A[a_0 \ldots a_{k-1}]$ je*

$$\sum_{i=0}^{k-1} A(i) \cdot 2^i$$

Pripočítanie jednotky možno vyjadriť takto

INC(A);

$i \leftarrow 0$
while $(i \leq k-1$ *AND* $A(i) = 1)$ ***do*** $A(i) \leftarrow 0; i \leftarrow i+1$
if $i \leq k-1$ ***then*** $A(i) \leftarrow 1$

Ak ako mieru zložitosti uvažujeme počet preklopených bitov, stojí nás jedno pričítanie jednotky zrejme nanajvýš k. Pri realizovaní postupnosti n takýchto inštrukcií sa tak klasicky dostaneme k celkovému počtu

$$n \cdot k$$

Použime metódu zoskupení. Vytvoríme k skupín, pričom i-tu skupinu tvorí preklopenie $A(i)$. Uvedomme si, že $A(i)$ sa preklopí v každom 2^i-tom volaní INC. Preto celkovú zložitosť vyjadríme

$$\sum_{i=0}^{\lfloor \log n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor \leq n \cdot \sum_{i=0}^{\lfloor \log n \rfloor} \frac{1}{2^i} \leq 2n$$

1.5.2 Metóda súčtov

Okrem prirodzenej ceny dostane každá (sledovaná)operácia kredit[4]. Pri realizácii operácie platíme jej kreditom. Ak je kredit väčší ako prirodzená cena operácie, akoby sme si predplatili nejakú ďalšiu manipuláciu s objektami.

cena $\leq$ **kredit** zvyšné kredity (rozdiel medzi kreditmi a cenou) si uložíme na účet k použitiu v budúcnosti

kredit $<$ **cena** rozdiel musí operácia doplatiť z účtu

Kvôli bezproblémovému realizovaniu postupnosti operácií musíme mať účet v priebehu celého výpočtu nezáporný $\Rightarrow$ **nesmieme ísť do debetu**. Ak tomu tak je, potom súčet kreditov dáva horný odhad na zložitosť.

Zásobník

	cena	kredit
Push	1	2
Pop	1	0
Multipop	$\min\{k, \lvert S\rvert\}$	0

[4]prepočítaniu na kredity sa niekedy hovorí amortizovaná cena operácie

Všimnime si, že hoci je cena realizácie Pop jednotková, priradili sme tejto operácii 2 kredity – akoby sme si predplatili aj odstránenie práve vloženého prvku. Je zrejmé, že zložitosť je nanjavýš $2n$.

POČÍTADLO

	cena	kredit
nastavenie bitu na 1	1	2
nastavenie bitu na 0	1	0

Hodnota kreditu 2 pri nastavovaní príslušného bitu na 1 opäť počíta "s budúcnosťou" – keď sa nabudúce dostaneme na toto miesto, budeme ho *musieť* preklopiť. Opäť ľahko vidno, že zložitosť realizovania n binárnych pripočítaní je $2n$. Stačí si uvedomiť, že každé pripočítanie jednotky preklopí len jeden bit z 0 na 1 a na 0 preklápam len taký bit, ktorý som predtým nastavila na 1 (a teda som si na túto operáciu ušetrila).

1.6 Základné výpočtové modely

Rozvoj počítačov spôsobil, že za formálny popis riešenia problémnu môžme považovať riešenie napísané v programovacom jazyku. Počítač potom realizuje program na jednotlivých vstupoch. Hovoríme teda o algoritmickom riešení problémov.

Ak chceme dokázať, že problém je algoritmicky riešiteľný, napíšeme jeho riešenie v programovacom (algoritmickom) jazyku. Na tejto úrovni nie je nutné jazyk fixovať. Potreba formálej definície pojmu *algoritmus* vyvstáva, keď chceme hovoriť o *riešiteľnosti/neriešiteľnosti* problémov, resp. o takých aspektoch zložitosti problémov, ktoré sú nezávislé od výberu konkrétneho z rozumnej triedy modelov.

Stretávame sa s viacerými modelmi výpočtov. Potrebujeme, aby bol model jednoduchý, s malým počtom jednoduchých elementárnych inštrukciíí, ale pritom aby odrážal naše intuitívne vnímanie pojmu algoritmu/algoritmickej riešiteľnosti/vypočitateľnosti. Za takýto model sa berie *Turingov stroj* (TS). V súvislosti so skúmaním zložitosti konkrétnych problémov sa tiež študuje tzv. RAM.

1.6.1 (M)RAM

V tejto časti sa budeme venovať abstraktnému výpočtovému modelu, ktorý pripomína asembler. (M)RAM je skratkou z anglického Random Access Machine, teda počítač s priamym prístupom do pamäti.

$$\text{(M)RAM} = \text{Program} + \text{Dátová štruktúra (pamäť)}$$

(M)RAM je *model* počítača von Neumanovského typu - pamäť, program, čítač inštrukcií. Pri jeho definovaní teda musíme popísať spôsob komunikáciu s okolím – vstupy/výstupy; organizáciu pamäte a spôsob jej adresácie; programovací jazyk, čiže základné inštrukcie. Abstrakcia tohto modelu je najmä v jeho potenciálne nekonečnej pamäti.

Pamäť je pole registrov, v ktorých sa nachádzajú čísla. Pritom predpokladáme
- veľkosť registrov je neobmedzená/operácie nad nekonečne veľkými číslami
- počet registrov je neobmedzený
- priamy prístup do každého registra (indexovanie)

Vstup je konečná postupnosť celých čísel, ktorú čítame zľava doprava. Predpokladáme jednosmerné čítanie, čo znamená, že po vykonaní jednej operácie načítania sa "čítacia hlava"nastaví na ďalšie číslo v poradí.

Program je konečná postupnosť inštrukcií. Nech

i označuje číslo zo vstupu
c(j) označuje obsah registra $R(j)$
P je čítač inštrukcií (číslo práve vykonávaného riadku programu); každá inštrukcia zvýši jeho hodnotu o jedna, ak to skokovou inštrukciou nie je určené inak
x je ľubovoľný operand typu
=j (konštanta),
j (priama adresácia)
*j (nepriame adresovanie)

R(0) označuje akumulátor

adresné módy
$$\begin{cases} x \text{ je číslo } j, & \text{obsah registra } R(j) \\ x \text{ je } *j, & \text{obsah registra, ktorého číslo je uložené v registri } R(j) \\ x \text{ je } = j, & \text{priamo číslo } j \end{cases}$$

Inštrukcie

inštrukcia	operand	sémantika
$READ$	i	$c(i) \leftarrow$ číslo zo vstupu
$READ$	$*i$	$c(c(i)) \leftarrow$ číslo zo vstupu
$STORE$	j	$c(j) \leftarrow c(0)$
$STORE$	$*j$	$c(c(j)) \leftarrow c(0)$
$LOAD$	x	$c(0) \leftarrow x$
ADD	x	$c(0) \leftarrow c(0) + x$
SUB	x	$c(0) \leftarrow c(0) - x$
$JUMP$	j	$P \leftarrow j$
$JPOS$	j	$if\ c(0) > 0\ then\ P \leftarrow j$
$JZERO$	j	$if\ c(0) = 0\ then\ P \leftarrow j$
$HALT$		ukončenie výpočtu
$WRITE$	$= j$	j
$WRITE$	j	$c(j)$
$WRITE$	$*j$	$c(c(j))$

Na RAM sa môžeme pozerať dvomi spôsobmi

1. RAM **počíta funkciu** f. Niekedy sa stretneme s tým, že v takomto prípade sa nepoužívajú inštrukcie WRITE, ale za výstup po zastavení výpočtu sa považuje obsah akumulátora.
2. RAM dáva ako výsledok viacero čísel. Vtedy sa používa aj inštrukcia zápisu. Výstup je postupnosť celých čísel. Po zapísaní čísla na výstupe sa hlava posúva ďalej, čo znamená, že výsledkom je postupnosť čísel zapísaná na výstupnej páske.

Ako je to s definovaním zložitosti? Prirodzené by bolo definovať časovú zložitosť ako počet vykonaných inštrukcií. Vzhľadom k tomu, že máme potenciálne nekonečný register a teda potenciálne nekonečne veľké čísla, asi by to niekedy bolo zavádzajúce. Preto sa rozlišujú dva rôzne prístupy

jednotkové kritérium

Pri **jednotkovej** cene je manipulácia s číslom/registrom jednotkovej zložitosti

- **čas** je počet vykonaných inštrukcií
- **priestor, resp. pamäť** je počet použitých registrov

Je zrejmé, že jednotková cena odráža zložitosť vtedy, ak veľkosť čísel (dĺžku zápisu) môžme ohraničiť konštantou. Vtedy manipuláciu s číslom môžme považovať za konštantu; keďže väčšinou nás zaujíma asymptotické správanie zložitosti, je väčšinou jedno, či uvažujeme $O(f(n))$ alebo $O(cf(n))$, kde c je konštanta.

logaritmické kritérium

Logaritmická cena berie do úvahy veľkosť čísel, s ktorými pracujeme. Cena inštrukcie nie je 1, ale $\lfloor \log i \rfloor + 1$, kde i je najväčšia absolútna hodnota z čísel, s ktorými pri vykonávaní danej inštrukcie manipulujeme (niekedy aj počet bitov, ktoré sa pri vykonaní inštrukcie spracujú; rozdiel v týchto dvoch definíciách je v multiplikatívnej konštante). Analogická úvaha je pre pamäťovú zložitosť, kde veľkosť (cena) použitého registra nie je jednotková, ale je to maximána dĺžka čísla, uloženého počas výpočtu v tom-ktorom registri.

- **čas** je súčet cien vykonaných inštrukcií
- **pamäť** je súčet cien všetkých registrov, použitých v priebehu výpočtu

Vidíme, že logaritmická cena odstraňuje – z hľadiska zložitosti – abstrakciu RAMu, ktorou je dvojrozmerná nekonečnosť pamäte - počet registrov aj ich veľkosť môžu

byť nekonečne veľké. Ak pracujeme s nekonečne veľkými číslami, sú všetky povolené operácie RAMu lineárnej zložitosti vzhľadom na dĺžku binárneho zápisu. Teraz už asi vidno, prečo sme pri definovaní aritmetických operácií nedefinovali násobenie a delenie. Súvisí to s tým, že algoritmy násobenia a delenia nie sú lineárne. Je zrejmé, že pridaním násobenia a delenia

- neovplyvníme výpočtovú silu RAMu
- zložitosť nebude úplne zodpovedať tomu, čo očakávame

Model s násobením a delením je však užitočný, preto ho budeme používať a budeme ho označovať MRAM.

MRAM je RAM, ktorý má navyše inštrukcie *MRAM*

DIV	celočíselné delenie
MULT	násobenie

Keďže nie je jednoduché počítať logaritmickú zložitosť presne, postupujeme väčšinou tak, že vypočítame zložitosť pri jednotkovej cene a prenásobíme ju maximálnou dĺžkou čísla, ktoré sa v priebehu výpočtu použilo. Získame tak horný odhad na logaritmickú zložitosť.

Úloha: Napíšte RAM pre problém triedenia. Začnite s jednoduchým (z hľadiska programovania, nie efektívnosti) algoritmom Bubblesort. Určte zložitosť Vášho programu pri jednotkovej aj logaritmickej cene.

Úloha: Napíšte RAM pre problém Insertsort. Určte zložitosť Vášho programu pri jednotkovej aj logaritmickej cene.

Úloha: Napíšte RAM, ktorý načíta "reťazec"—postupnosť čísel: $n, a_1, \dots, a_n$, kde $a_i \in \{1, 2\}^*$. Na výstup vypíše:

1. $a_2, a_1, a_3, a_4, \dots, a_n, a_{n-1}$ alebo $a_2, a_1, a_3, a_4, \dots, a_{n-2}, a_{n-1}, a_n$ podľa toho, či je n párne alebo nepárne
2. 1 ak tvorí vstupná postupnosť palindróm ($a_1 \dots a_n = a_n \dots a_1$)
3. 1 ak vstupná postupnosť obsahuje ako súvislý podreťazec palindróm dĺžky aspoň $n/2$
4. 1 ak číslo, ktorého desiatkovým zápisom je vstupná postupnosť, je prvočíslo

Určte zložitosť Vášho programu pri jednotkovej aj logaritmickej cene.

1.6.2 Základný model Turingovho stroja

V tejto časti sa budeme venovať z nejakého pohľadu najjednoduchšej verzii Turingovho stroja (TS) - bude to model, ktorého výpočtová sila je ekvivalentná tomu, čo považujeme za algoritmicky riešiteľné. Analogicky ako v prípade (M)RAMu ide o abstaktný výpočtový model, kde abstrakciou je jednak zúženie množiny elementárnych inštrukcií a na strane druhej uvažovanie neobmedzene veľkej pamäte.

Pamäť TS je (jednosmerne)potenciálne nekonečná páska, tvorená potenciálne nekonečnou postupnosťou políčok. Každé políčko môže obsahovať práve jeden z *konečnej* množiny symbolov. Vstupné slovo je na začiatku uložené v súvislom bloku po sebe

idúcich políčok, čítacia hlava je nastavená na prvom symbole vstupného slova. Ľavý okraj pásky budeme označovať špeciálnym symbolom \$. Ostatné políčka obsahujú špeciálny znak, tzv. B (blank), ktorý vyjadruje to, že políčko je prázdne. Prechodová funkcia na základe symbolu pod hlavou (v tomto prípade čítacou i zapisujúcou) a stavu konečnostavovej riadiacej jednotky určuje nový obsah políčka pod hlavou, nový stav a posun hlavy (nanajvýš o jedno políčko doľava, resp. doprava).

Turingov stroj

Definícia 1.1 *Turingov stroj je šestica $T = (\Sigma, \Gamma, K, q_0, \delta, F)$, kde*

Σ; $B, \$ \notin \Sigma$,	*je konečná neprázdna množina symbolov, tzv.* vstupná abeceda,
$\Gamma; \Sigma \subseteq \Gamma$	*je konečná neprázdna množina symbolov, tzv.* pracovná abeceda
K	*je konečná neprázdna množina* stavov
$q_0, q_0 \in K$	*je* počiatočný stav
$F, F \subseteq K$,	*je* množina koncových/akceptujúcich stavov
δ :	$K \times \{\Gamma - \$\} \rightarrow K \times \{\Gamma - B\} \times \{0, 1, -1\} \bigcup K \times \$ \rightarrow K \times \$ \times \{0, 1, \}$ *je* prechodová funkcia

konfigurácia TS

Konfigurácia C Turingovho stroja T je reťazec $\alpha q \beta$, kde $\alpha\beta$ je celá neprázdna časť pásky , q je stav, hlava T je umiestnená na prvom políčku-znaku z β. Uvedomme si, že konfigurácia je len dohodnutým popisom kompletnej informácie o stroji T.

počiatočná konfigurácia

Počiatočná konfigurácia je $C_0 = q_0 a_1 a_2 ... a_n$, kde $a_1 a_2 ... a_n$ je vstupné slovo. Je zrejmé, že C_0 popisuje situáciu na začiatku výpočtu.

krok výpočtu

Elementárnymi inštrukciami stroja T sú zrejme prvky prechodovej funkcie δ. Aplikovaniu prechodovej funkcie hovoríme *krok výpočtu.* Hovoríme, že konfiguráciu C_{i+1} vieme dostať z konfigurácie C_i v jednom kroku, značíme $C_i \rightarrowtail C_{i+1}$, ak

$C_i = a_1 a_2 ... a_j \mathbf{q a_{j+1}} a_{j+2} ... a_m$ a

$$C_{i+1} = \begin{cases} a_1 a_2 ... a_j \mathbf{p b a_{j+2}} ... a_m, & \text{pričom } (p, b, 0) \in \delta(q, a_{j+1}); \text{ hlava stojí} \\ a_1 a_2 ... a_j \mathbf{b p a_{j+2}} ... a_m, & \text{pričom } (p, b, 1) \in \delta(q, a_{j+1}); \text{ hlava sa hýbe doprava} \\ a_1 a_2 ... p a_j \mathbf{b a_{j+2}} ... a_m, & \text{pričom } (p, b, -1) \in \delta(q, a_{j+1}); \text{ hlava sa hýbe doľava} \end{cases}$$

Symbolom $\rightarrowtail^*$ označujeme reflexívny tranzitívny uzáver $\rightarrowtail$.

výpočet

Výpočet Turingovho stroja T je postupnosť konfigurácií $C_0, C_1, C_2, ..., C_m$, pričom C_0 je počiatočná konfigurácia a $C_i \rightarrowtail C_{i+1}$.

TS ako akceptor

Keď TS používame ako akceptor, inými slovami, ak na TS riešime rozhodovacie problémy, zaujíma nás, či vstupné slovo patrí alebo nepatrí do príslušného jazyka. V takom prípade sa zvykne hovoriť o *akceptujúcom výpočte.* Akceptujúci výpočet je výpočet, ktorý končí konfiguráciou so stavom z množiny koncových stavov (ktorému sa vtedy hovorí aj akceptujúci stav). Potom *jazyk* L akceptovaný TS T je množina slov

$$L(T) = \{w | q_0 w \rightarrowtail^* \alpha q \beta, q \in F\}^5$$

TS počítajúci funkciu

[5] Uvedomme si, že sme nešpecifikovali, ako sa má skončiť výpočet na takom vstupe, ktorý nepatrí do daného jazyka. Ak k danému jazyku L vieme skonštruovať taký TS, ktorého výpočet na každom vstupe skončí, vtedy je L rekurzívny jazyk; hovoríme tiež, že TS jazyk L rozhoduje/rozpoznáva.

Ak považujeme TS za model počítača, chceli by sme, aby na vstup reagoval výstupom. Keďže máme (zatiaľ) len jednu pásku, musíme aj výstup napísať na ňu. V takomto prípade od *koncovej konfigurácie* C_m vyžadujeme, aby sme na páske videli/našli výstup. Nech teda $C_0, C_1, C_2, ..., C_m$ je výpočet TS T.

Konfigurácia $C_m = \alpha_m q \beta_m$ je *koncová*, ak

$q \in F$

$\beta_m = \gamma_m \# y$, pričom y je *výstup/výsledok*.

Je zrejmé, že akceptor dostaneme aj tak, že $y \in \{0, 1\}$.

Úloha: Napíšte TS, ktorý rozhoduje jazyk

- $L = \{wcw \mid w \in \{a, b\}^*\}$
- $L = \{ww \in \{a, b\}^*\}$

Ak je príslušný TS taký, že vieme zaručiť konečný výpočet len na slovách z jazyka a pri vstupoch, ktoré z jazyka nie sú sa môže stať, že výpočet je nekonečný, vtedy TS jazyk akceptuje a príslušný jazyk je rekurzívne vyčísliteľný.

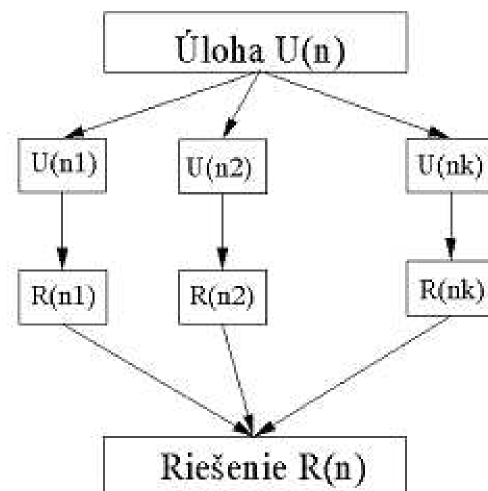
Kapitola 2

Základné metódy tvorby efektívnych algoritmov

2.1 Rozdeľuj a panuj

Táto metóda je jednou z najčastejšie používaných metód. Aplikujeme ju vtedy, ak riešenie úlohy väčšieho rozsahu vieme získať vhodným zložením výsledkov podúloh menšieho rozsahu ale toho istého charakteru. Úlohu potom riešime rekurzívne. Schématicky sa riešenie problémov metódou rozdeľuj a panuj dá popísať nasledovne

- Rozdeľ úlohu $U(n)$ rozsahu n na niekoľko podúloh $U(n_1), \ldots, U(n_k)$ toho istého charakteru ale menšieho rozsahu
- Vyrieš jednotlivé úlohy $U(n_1), \ldots, U(n_k)$
- Ak sú rozsahy úloh $U(n_1), \ldots, U(n_k)$ príliš veľké, použi opakovane princíp rozdeľuj a panuj
- Zlož riešenie $R(n)$ úlohy $U(n)$ z riešení $R(n_1), \ldots, R(n_k)$ úloh $U(n_1), \ldots, U(n_k)$



Ak označíme $T(n)$ zložitosť riešenia nášho problému pre rozsah vstupu n, tak

$$\begin{aligned} T(n) &= \text{zložitosť rozkladu } U(n) \text{ na } U(n_1), \ldots, U(n_k) + \\ &\quad + T(n_1) \ + T(n_2) + T(n_k) + \\ &\quad + \text{zložitosť zloženia výsledku.} \end{aligned}$$

Skôr, ako si povieme o riešení uvedenej rovnice, jeden príklad.

Príklad 2.1 *Majme problém MinMax* *minMax*

vstup:	$S = \{a_1, a_2, \ldots, a_n\}, a_i \in A$, *kde* A *je množina s usporiadaním*
výstup:	*Maximálny a minimálny prvok množiny* S
miera zložitosti:	*počet porovnaní*

Algoritmus 1 Rozdeľuj a panuj

1: **if** $|S| = \{a\}$ **then** return (a,a)
2: **if** $|S| = 2$ **then** return (a, b), kde $a, b \in S$, $a < b$
3: **else** rozdeľ S na dve rovnako veľké disjunktné množiny S_1, S_2;
4: $min1, max1 \leftarrow \mathbf{minMax(S_1)}$;
5: $min2, max2 \leftarrow \mathbf{minMax(S_2)}$;
6: return $(\min\{min1, min2\}, \max\{max1, max2\})$;

Zložitosť algoritmu budeme kvôli jednoduchosti analyzovať pre prípad $n = 2^k$.

$$T(n) = \begin{cases} 1, & \text{ak } n{=}2 \\ 2T(n/2) + 2 & \text{pre } n > 2 \end{cases}$$

Riešením tejto rekurentnej rovnice dostávame

$$\begin{aligned} T(n) &= 2T(n/2) + 2 = 2(2T(n/4) + 2) + 2 = \ldots = \\ &= 2^{k-1}T(2) + \sum_{i=1}^{k-1} 2^i = 2^{k-1} + 2^k - 1 - 1 = \frac{3n}{2} - 2 \end{aligned}$$

Odvodili sme zložitosť najlepšieho i najhoršieho prípadu pre prípad $n = 2^k$. *Dá sa ukázať, že zložitosť meraná počtom porovnaní sa vylepšiť nedá, a tak v tomto prípade použitie metódy rozdeľuj a panuj viedlo dokonca k* optimálnemu algoritmu.

Veta 2.1 (metóda rozdeľuj a panuj) *Nech* a, b, c *sú nezáporné racionálne čísla,* $n = c^k$. *Potom riešením rekurentnej rovnice*

$$T(n) = \begin{cases} b & \text{ak } n{=}1 \\ aT(n/c) + bn & \text{pre } n > 1 \end{cases} \quad \text{je} \quad T(n) = \begin{cases} O(n) & \text{pre prípad } a < c \\ O(n \log n) & \text{pre prípad } a = c \\ O(n^{\log_c a}) & \text{pre prípad } a > c \end{cases}$$

Dôkaz: Všimnime si niekoľko prvých členov súčtu

$$\begin{aligned} T(n) &= aT(n/c) + bn = a(aT(n/c^2) + bn/c) = a^2T(n/c^2) + bna/c + bn = \\ &= a^2(aT(n/c^3) + bn/c^2) + abn/c + bn = \ldots = \\ &= a^{\log_c n} b + bn \sum_{i=0}^{\log_c (n-1)} \left(\frac{a}{c}\right)^i \end{aligned}$$

Rovnicu doriešime podľa vzťahu a, c.

$\mathbf{a < c}$ $\quad \sum_{i=0}^{\log_c n-1} \left(\frac{a}{c}\right)^i$ konverguje $\Rightarrow \quad O(n)$

$\mathbf{a = c}$ $\quad \left(\frac{a}{c}\right) = 1$, preto $\sum_{i=0}^{\log_c n-1} \left(\frac{a}{c}\right)^i = \log n \Rightarrow \quad O(n \log n)$

$\mathbf{a > c}$ $\quad \sum_{i=0}^{\log_c n-1} \left(\frac{a}{c}\right)^i$ je geometrický rad so súčtom $O(n^{\log_c a})$

□

Prečo hovoríme tejto vete, že je to veta o metóde rozdeľuj a panuj?
ak

- delením získame a podúloh rovnakej veľkosti - rozsahu n/c

- rozklad a zloženie výsledku vieme spraviť v lineárnom čase

tak zložitosť získaného algoritmu je vyjadrená práve vzťahom z vety 2.1 a je preto nanajvýš polynomiálna.
Bez dôkazu uvedieme všeobecnejšie znenie vety 2.1.

Veta 2.2 (master theorem) *Nech $a \geq 1, b > 1$ sú konštanty, $f(n)$ je funkcia a nech $T(n)$ je funkcia na nezáporných celých číslach definovaná nasledujúcou rekurenciou:*

$$T(n) = \begin{cases} c & \textit{ak n=1} \\ aT(n/b) + f(n) & \textit{pre n >1} \end{cases}$$

Potom T(n) asymptoticky odhadneme nasledovne:

$$T(n) = \begin{cases} f(n) = O(n^{\log_b a - \varepsilon}), \textit{ pre nejakú konštantu } \varepsilon & \textit{potom } T(n) = \Theta(n^{\log_b a}) \\ f(n) = O(n^{\log_b a}) & \textit{potom } T(n) = \Theta(n^{\log_b a} \ln n) \\ f(n) = O(n^{\log_b a + \varepsilon}), af(n/b) \leq df(n), c < 1 & \textit{potom } T(n) = O(f(n)) \end{cases}$$

2.1.1 Násobenie veľkých čísel

Uvažujme násobenie n-bitových čísel v situácii, keď v jednotkovom čase vieme násobiť len jednociferné čísla (alebo čísla ohraničenej dĺžky). Klasický školský algoritmus vedie k zložitosti $O(n^2)$ aritmetických operácií. Aplikujme metódu rozdeľuj a panuj.

Pri priamočiarom použití metódy rozdeľuj a panuj vyjadríme každý z reťazcov X, Y dĺžky n pomocou dvoch reťazcov dĺžky $n/2$. Ich vynásobením dostávame algoritmus, ktorého zložitosť ostala $O(n^2)$.

priamočiare použitie

$$X = X_1X_2 \qquad X = X_1 2^{n/2} + X_2 \qquad\qquad Y = Y_1Y_2 \qquad Y = Y_1 2^{n/2} + Y_2$$

$$X.Y = (X_1 2^{n/2} + X_2)(Y_1 2^{n/2} + Y_2) = X_1Y_1 2^n + (X_1Y_2 + X_2Y_1)2^{n/2} + X_2Y_2$$

$$\Downarrow$$

$$T(n) = 4T(n/2) + bn = O(n^2)$$

Zdá sa, že nám tento prístup nepomohol. Aby metóda rozdeľuj a panuj viedla k efektívnejšiemu algoritmu, museli by sme použiť menej ako 4 násobenia.

Keď si všimneme, že

rozumnejšie použitie

$$(a + b)(c + d) = ac + (bc + ad) + bd$$

mohli by sme postupovať nasledovne:

$$U \leftarrow X_1Y_1 \quad V \leftarrow X_2Y_2 \quad W \leftarrow (X_1 + X_2)(Y_1 +_Y 2)$$

$$\Downarrow$$

$$X.Y = U2^n + (W - U - V)2^{n/2} + V$$

Pre zložitosť tohto prístupu platí $T(n) = 2T(n/2) + T(n/2 + 1) + cn$. Ostalo násobenie $n/2$ a $n/2 + 1$ bitových čísel. Tu však môžme postupovať podobne — $n/2 + 1$ bitové čísla môžme "rozložiť" na 1 bit a $n/2$ bitov. To vedie k nahradeniu násobenia $n/2 + 1$ bitových čísel jedným násobením $n/2$-bitových čísel.

$$\begin{array}{ll}(X_1+X_2) = A_1A_2, & \text{pričom } A_1 \text{ je jednobitové číslo}\\ (Y_1+Y_2) \;= B_1B_2, & \text{pričom } B_1 \text{ je jednobitové číslo}\end{array}$$

Potom

$$\begin{aligned}(X_1+X_2)(Y_1+Y_2) &= (A_1 2^{n/2}+A_2)(B_1 2^{n/2}+B_2)\\ &= A_1B_1 2^n + A_2B_1 2^{n/2} + A_1B_2 2^{n/2} + A_2B_2\end{aligned}$$

$$T(n) = 3T(n/2) + cn = \mathbf{O}(\mathbf{n}^{\log_2 3})$$

2.1.2 Strassenov algoritmus násobenia matíc

Uvažujme problém násobenia dvoch štvorcových matíc stupňa $n \times n$. Ak postupujeme podľa definície násobenia matíc, získaný algoritmus je zložitosti $O(n^3)$ aritmetických operácií. Ak uvažujeme, že prvky matíc sú z okruhu, môžme sa pokúsiť použiť metódu rozdeľuj a panuj:

$$\begin{pmatrix} A_{11} & A_{12}\\ A_{21} & A_{22}\end{pmatrix}\begin{pmatrix} B_{11} & B_{12}\\ B_{21} & B_{22}\end{pmatrix} = \begin{pmatrix} C_{11} & C_{12}\\ C_{21} & C_{22}\end{pmatrix}$$

$$\begin{aligned}C_{11} &= A_{11}B_{11} + A_{12}B_{21} & C_{21} &= A_{21}B_{11} + A_{22}B_{21}\\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} & C_{22} &= A_{21}B_{12} + A_{22}B_{22}\end{aligned}$$

Priamočiare použitie metódy rozdeľuj a panuj opäť vedie k zložitosti, ktorá nie je vylepšením.

$$T(n) = 8T(n/2) + cn^2 = ... = O(n^3)$$

Vylepšenie získame ušetrením jedného násobenie na úkor nárastu počtu sčítaní a odčítaní. Násobenie dvoch matíc rozmeru 2x2 môžme spraviť nasledovne:

$$\begin{aligned}P &= (a_{11}+a_{22})(b_{11}+b_{22}) & C_{11} &= P+S-T+V\\ Q &= (a_{21}+a_{22})b_{11} & C_{12} &= R+T\\ R &= a_{11}(b_{12}-b_{22}) & C_{21} &= Q+S\\ S &= a_{22}(b_{21}-b_{11}) & C_{22} &= P+R-Q+U\\ T &= (a_{11}+a_{12})b_{22} & &\\ U &= (a_{21}-a_{11})(b_{11}+b_{12}) & &\\ V &= (a_{12}-a_{22})(b_{21}+b_{22}) & &\end{aligned}$$

Aplikovaním uvedených vzťahov a metódy rozdeľuj a panuj dostávame rekurzívny algoritmus, ktorého zložitosť vyjadríme rekurentnou rovnicou

$$T(n) = \begin{cases} 7T(n/2) + an^2, & n > 2\\ b, & n \le 2\end{cases}$$

Riešením tejto rovnice je $O(n^{\log_2 7})$.

Pritom sme predpokladali, že $n = 2^k$. Čo v prípade, ak to nie je pravda

- doplniť nulovými riadkami a stĺpcami na najbližšiu mocninu dvojky;
- pri párnom n priame použitie, pri nepárnom n doplniť jeden riadok a stĺpec
- doplniť na najbližšie také číslo m, že $m = 2^r p$ a potom po rozmer $p \times p$ aplikuj rozdeľuj a panuj, pri rozmere $p \times p$ priamo vynásob

2.1.3 Vyhľadávanie k-teho najmenšieho prvku

vstup:	$S = \{a_1, a_2, \dots, a_n\}, a_i \in A$, kde A je množina s usporiadaním, $k \in \mathbb{N}$
výstup:	k-ty najmenší prvok množiny S
miera zložitosti:	počet porovnaní

Zaujíma nás teda $k-$ty najmenší prvok, nevyžadujeme, aby množina bola utriedená. Na riešenie by sme mohli použiť jednoduché algoritmy.

1. utrieď a vyber k-ty najmenší prvok $O(n \log n)$
2. k-krát hľadaj minimum $O(kn)$
3. resp. porovnaním vstupných hodnôt $k, \log n$ zistiť, ktorý z uvedených algoritmov je pre daný vstup výhodnejší. $O(\min\{k, \log n\} \cdot n)$

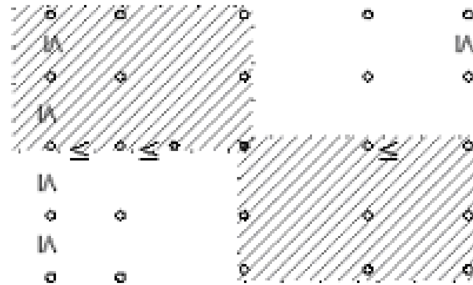
Využitím metódy rozdeľuj a panuj získame algoritmus $Select(k, S)$.

Algoritmus 2 Select(k,S)

```
1: if S < 50 then utrieď a najdi k-ty najmenší
2: else
3:     rozdeľ S do 5-tíc a utrieď v rámci 5-tíc
4:     vytvor množinu M stredných prvkov
5:     m := SELECT(M/2, M)
6:     vytvor množiny S1, S2
7:     S1 = {a ∈ S; a < m}
8:     S2 = {a ∈ S; a > m}
9:     if |S1| ≥ k then return SELECT(k, S1)
10:    else
11:        if |S1 + 1| = k then return m
12:        else return SELECT(k, S2)
```

Korektnosť algoritmu ľahko ukážeme indukciou vzhľadomk veľkosti množiny S.

Zložitosť algoritmu: označme $T(n)$ zložitosť pre $|S| = n$. Potom



3,4	zložitosť $O(n)$
5	$T(n/5)$
6-8	$O(n)$
9-12	$O(\max\{T(\lvert S_1\rvert), T(\lvert S_2\rvert)\})$

Keďže $|S_1 1, |S_2| \leq 3|S|/4$, dostávame

$$T(n) \leq \begin{cases} T(n/5) + T(3n/4) + cn, & 50 \leq n \\ cn, & n \leq 50 \end{cases}$$

Matematickou indukciou sa dá ukázať: $T(n) \leq 20cn$.

2.1.4 Násobenie Booleovských matíc

Majme problém násobenia dvoch booleovských matíc. Nakoľko booleovské matice netvoria okruh, nemôžme priamo použiť Strassenov algoritmus násobenia matíc. Môžme však vnoriť booleovské matice do okruhu, tam použiť Strassenov algoritmus a potom sa vrátiť naspäť.

Majme booleovské matice A, B. Nech súčinom $A \cdot B = C$. Predstavme si, že budeme matice A a B násobiť v **okruhu** Z_{n+1}, kde n je rozmer matíc A, B. Výsledkom tohto násobenia nech je matica D. Aký je vzťah medzi maticami C a D?

$$\text{D(i,j)=0} \quad \Leftrightarrow \quad \text{C(i,j)=0}$$
$$\text{D(i,j)} \neq 0 \quad \Leftrightarrow \quad \text{C(i,j)=1}$$

Takže z matice D vieme požadovanú maticu C získať v čase úmernom veľkosti matíce C, D.

Nasleduje algoritmus, ktorý je lepší ako klasický školský algoritmus násobenia, hoci väčšej zložitosti ako Strassenov algoritmus. Rozdeľme maticu A na stĺpce $A_1, \ldots, A_m$ hrúbky $\log n$ a maticu B na riadky $B_1, \ldots, B_m$ hrúbky $\log n$, kde $m = n/\log n$. Potom

$$A \cdot B = \sum_{i=1}^{m} A_i \cdot B_i = \sum_{i=1}^{m} C_i$$

Pokiaľ by sa nám podarilo získať súčin $A_i \cdot B_i$ v čase $O(n^2)$, získame algoritmus zložitosti $O(n^3/\log n)$. Uvedomme si, že j-ty riadok matice C_i vznikol tak, že sa sčítali tie riadky matice B_i , ktoré odpovedali jednotkám v j-tom riadku matice A_i. Keďže všetkých možných riadkov matice A_i je n, je aj počet všetkých možných riadkov matice C_i len n. Najprv všetky tieto potenciálne riadky predvypočítame do tabuľky TAB, a potom použijeme ten, ktorý potrebujeme.

Pri fixovanom i TAB počítame nasledovne:

$TAB(0) = 0000...0$
$TAB(j) = TAB(j - 2^k) + B_i(k + 1)$, kde $2^k \leq j < 2^{k+1}$ a $B_i(t)$ označuje t-ty riadok odspodu matice B_i(každé číslo považujeme za binárny reťazec dĺžky $\log n$. Potom odčítanie 2^k odpovedá nahradeniu vodiacej jednotky v binárnom zápise čísla j nulou)

Ako pomocou TAB získame maticu C? Nech $bin(v)$ označuje číslo, ktorého binárnym zápisom je binárny reťazec v. Potom

$$C_i(j) = TAB(bin(A_i(j)))$$

Algoritmus 3 BoolNásobenie

```
1: rozdeľ maticu A na stĺpcové podmatice A_1, A_2, ..., A_m
2: rozdeľ maticu B na riadkové podmatice B_1, B_2, ..., B_m
3: vytvor nulovú maticu C
4: for i=1 to m do
5:     vypočítaj TAB
6:     for j = 1 to n do
7:         C(j) = C(j) ⊕ TAB(bin(A_i(j)))
```

Aká je zložitosť tohto algoritmu?

– riadok 5 vieme realizovať so zložitosťou $O(n^2)$
– cyklus na riadku 6 sa realizuje v čase $O(n^2)$
– cyklus na riadku 4 opakujeme m, čiže $n/\log n$ krát

Preto je výsledná zložitosť $\mathbf{O(n^3/\log n)}$.

2.2 Dynamické programovanie

Začnime s príkladom.

súťaž družstiev

Príklad 2.2 *Uvažujme súťaž dvoch rovnako silných družstiev A, B, ktoré hrajú proti sebe zápasy. Pre každý zápas je rovnako pravdepodobné víťazstvo družstva A i B. Nech (i, j) je dvojica prirodzených čísel. Hovoríme, že* družstvo A vyhralo súťaž s parametrami (i, j), ak A dosiahne i víťazstiev skôr, ako B dosiahne j víťazstiev. *Našou úlohou je vypočítať pravdepodobnosť $P(i, j)$ toho, že A vyhrá súťaž s parametrami (i, j).*

Analýzou dostávame

- $P(0, j) = 1, \ j > 0$
- $P(i, 0) = 0, \ i > 0$
- $P(i, j) = \frac{1}{2}(P(i, j-1) + P(i-1, j)), \ i, j > 0$[1]

Tento rekurentný vzťah navádza na použitie metódy rozdeľuj a panuj. Ak za veľkosť vstupu berieme $n = i + j$ a ako mieru zložitosti uvažujeme počet priradení a aritmetických operácií, potom metóda rozdeľuj a panuj vedie k zložitosti

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2T(n-1) + 3 \ = \ 2(2T(n-2) + 3) + 3 = 2^2 T(n-2) + 2 \cdot 3 + 3 \\ &= 2^n + 3 \cdot \textstyle\sum_{i=0}^{n-1} 2^i \end{aligned}$$

Ľahko vidíme, že horný odhad je exponenciálny.

Čím je spôsobená táto veľká zložitosť?

- Vieme síce z riešenia menších podproblémov poskladať riešenie väčšieho podproblému, ale pri riešení sa vygeneruje exponenciálne veľa menších podproblémov menšieho rozsahu.
- Vieme, že je len polynomiálne veľa problémov menšieho rozsahu. To ale znamená, že niektoré sme počítali niekoľkokrát.

Pri metóde rozdeľuj a panuj sme veľký problém rozdelili na menšie, ktoré mohli byť riešené nezávisle. V prípade metódy dynamického programovania je tento princíp dotiahnutý do extrému:

- riešime postupne VŠETKY problémy (toho istého charakteru) MENŠIEHO rozsahu; postupujeme systematicky od menšieho rozsahu k väčšiemu
- riešenia si ukladáme k ďalšiemu použitiu
- používame, keď je počet problémov menšieho rozsahu rozumný

Nevýhodou je, že môžeme potrebovať veľkú pamäť.

Časté použitie tejto metódy je pre taký typ optimalizačných úloh, kde si riešenie možno predstaviť ako postupnosť nejakých rozhodnutí. V tejto situácii my hľadáme takú postupnosť rozhodnutí, ktorá niečo optimalizuje.

Abysme mohli použiť dynamické programovanie, musí sa optimalita dediť. Znamená to, že bez ohľadu na to, aké bolo prvé rozhodnutie, zvyšné rozhodnutia musia

tvoriť optimálne riešenie pre situáciu, ktorá nastala po prvom rozhodnutí. Inými slovami, ak súčasťou optimálneho riešenia väčšieho problému je riešenie menšieho problému, je toto riešenie optimálnym pre daný podproblém.

dedenie optimality

Formálnejšie. Nech

- S_0 je situácia na začiatku.
- $\{r_1, \ldots, r_k\}$ je množina rozhodnutí, ktoré môžme urobiť v prvom kroku
- S_i je situácia po aplikovaní rozhodnutia r_i
- Γ_i je optimálna postupnosť rozhodnutí pre situáciu S_i

Hovoríme, že úloha (problém) spĺňa princíp dedenia optimality, ak o optimálnom riešení situácie S_0 platí, že vznikne výberom najlepšieho z riešení $r_1\Gamma_1,\ r_2\Gamma_2, \ldots,\ r_k\Gamma_k$.

2.2.1 Súťaž dvoch družstiev

Použime na výpočet hodnoty $P(i,j)$ metódu dynamického programovania.

- $P(1,0) = P(2,0) = \ldots = P(n,0) = 0$
- $P(0,1) = P(0,2) = \ldots = P(0,n) = 1$
- $P(i,j) = \frac{P(i-1,j)+P(i,j-1)}{2}$

Hodnoty budeme počítať postupne, začneme s rozsahom problému 1, potom prejdeme k rozsahu 2, ... Vypočítané hodnoty uložíme do tabuľky.

P(0,4)=1				
P(0,3)=1	P(1,4)			
P(0,2)=1	P(1,2)	P(2,2)		
P(0,4)=1	P(1,1)	P(2,1)	P(3,1)	
	P(1,0)=0	P(2,0)=0	P(3,0)=0	P(4,0)=0

Obrázok 2.1: Výpočet $P(i,j)$

Vzhľadom k závislosti väčšieho podroblému na menších je v tomto prípade zrejmé, že pri vypĺňaní tabuľky môžme postupovať po diagonálach, ktoré odpovedajú jednotlivým rozsahom problému.

Algoritmus 4 DP-1

```
1: for s ← 1 to n do
2:     P(0,s) ← 1; P(s,0) ← 0;
3:     for r ← 1tos − 1 do
4:         P(r,s) ← (P(r − 1,s) + P(r,s − 1))/2
```

zložitosť

Analyzujme teraz zložitosť algoritmu $DP-1$

čas Počítame $2+3+\ldots+(n+1)$ hodnôt. Na výpočet každej z nich stačí konštantne veľa času. Celkovo teda máme $\mathbf{O(n^2)}$.

[1]S pravdepodobnosťou 1/2 vyhrá prvú hru A. Potom mu ostáva vyhrať $i-1$ hier skôr, ako B vyhrá j hier. Analogicky, s pravdepodobnosťou 1/2 vyhrá prvú hru B. Potom A musí vyhrať i hier skôr, ako B vyhrá $j-1$ hier

pamäť všimnime si rozmer tabuľky

- tabuľka je veľkosti $O(n^2)$
- pri výpočte sa hodnoty zapisujú po diagonálach, pričom nová vzniká z informácií v starej; ľahko vidno, že stačí priestor dvoch diagonál
- pri lepšej manipulácii stačí len priestor jedinej diagonály[2]

2.2.2 Násobenie n matíc

Vstup: matice $M_1, \ldots, M_n$, pričom rozmer matice M_i je $r_i \times s_i,\ r_{i+1} = s_i$. *problém*
Výstup: matica $M = M_1 \times \ldots \times M_n$

Zamyslime sa nad tým, čím môžme ovplyviť zložitosť vypočítania tohto súčinu. Sú to

- efektívnosť algoritmu použitého na vynásobenie dvoch matíc
- poradie, v ktorom matice budeme násobiť

$M_1 = 10 \times 20$	$M_1 \times ((M_2 \times M_3) \times M_4) = 12200$
$M_2 = 20 \times 5$	$(M_1 \times M_2) \times (M_3 \times M_4) = 1550$
$M_3 = 5 \times 100$	$M_1 \times (M_2 \times (M_3 \times M_4)) = 800$
$M_4 = 100 \times 1$	$(M_1 \times (M_2 \times M_3)) \times M_4 = 31000$

Ako vidíme, rôzne ozátvorkovanie/poradie násobenia má vplyv na výslenú zložitosť. Má preto zmysel najprv predvypočítať vhodné uzátvorkovanie vstupnej postupnosti matíc tak, aby sme pri ich násobení použili (pri fixovanom algoritme násobenia dvoch matíc) čo najmenší počet aritmetických operácií. Dostali sme sa takto k optimalizačnej úlohe, ktorá by sa mohla dať riešiť metódou dynamického programovania. Presvedčíme sa o tom :

- riešenie hľadáme ako postupnosť rozhodnutí – ktoré dve matice majú byť v tom ktorom kroku násobené?
- platí princíp dedenia optimality
  Nech M_{ik} označuje maticu, ktorá vznikla vynásobením $M_i \times M_{i+1} \times \ldots \times M_k$. Nech A je optimálne poradie násobení, ktoré je také, že posledným násobením je násobenie $M_{1k} \times M_{k+1,n}$. Je zrejmé, že v A museli matice M_{1k} aj $M_{k+1,n}$ vzniknúť optimálnym násobením, lebo v opačnom prípade by sme ich výpočet v A nahradili poradím, ktoré by zmenšilo celkovú zložitosť.

Označme $cena(i,j)$ – počet operácií na získanie súčinu matíc $M_i \times M_{i+1} \times \ldots \times M_j$. Potom zrejme

$$cena(i,j) = cena(i,k) + cena(k+1,j) + \text{ zložitosť vynásobenia } M_{1k} \times M_{k+1,n}.$$

$$\underbrace{(M_i \times M_{i+1} \times \ldots \times M_k)}_{M_{ik}} \times \underbrace{(M_{k+1} \times \ldots \times M_j)}_{M_{k+1,j}}$$

Označme $m_{i,j}$ optimálny počet sledovaných operácií postačujúcich na získanie súčinu matíc $M_i \times M_{i+1} \times \ldots \times M_j$ keď predpokladáme, že pri násobení dvoch matíc používame klasický školský algoritmus. Potom

$$m_{ij} = \begin{cases} \min_{i \le k < j} \{(m_{ik} + m_{k+1,j} + r_i \times s_k \times s_j)\}, & \text{i < j} \\ 0, & \text{i=j} \end{cases}$$

[2]premyslite a naprogramujte

$m(1,4) = 800$ $k = 1$		
$m(1,3) = 6000$ $k = 2$	$m(2,4) = 600$ $k = 2$	
$m(1,2) = 1000$ $k = 1$	$m(2,3) = 10000$ $k = 2$	$m(3,4) = 500$ $k = 3$

Obrázok 2.2: Výpočet zátvorkovania

Uvedomme si, že m_{ij} je len optimálnym počtom operácií, potrebných na získanie výsledku M_{ij}. Nás ale zaujíma to, ktoré je to správne poradie násobenia matíc, pri ktorom dosiahneme optimálny počet operácií; potrebujeme do výrazu správne vložiť zátvorky. Preto si v tabuľke budeme pamätať nielen hodnotu m_{ij}, ale aj hodnotu $k(i,j)$, pri ktorej sa toto minimum m_{ij} dosahovalo. Hodnota $k(i,j)$ totiž určuje, ***za ktorú*** z matíc máme v danom násobení zložiť ***prvú pravú*** zátvorku.

$$\begin{array}{lll} (M_1 \times M_2 \times M_3 \times M_4),\ k(1,4) = 1 & \Rightarrow & (M_1) \times (M_2 \times M_3 \times M_4) \\ k(2,4) = 2 & \Rightarrow & (M_1) \times ((M_2) \times (M_3 \times M_4)) \end{array}$$

DÚ

Úloha:

- Napíšte program, ktorý pre vstupnú postupnosť rozmerov n matíc vypočíta, v akom poradí treba matice násobiť, aby pre daný algoritmus násobenia dvoch matíc bola výsledná zložitosť optimálna.
- Aká je časová a pamäťová zložitosť Vášho algoritmu?

2.2.3 Konštrukcia optimálneho binárneho vyhľadávacieho stromu

Keď používame binárny vyhľadávací strom (BVS) na reprezentáciu množiny, ktorú budeme často prezerať, má zmysel optimalizovať nielen čas na realizáciu jedného prístupu do BVS (teda zložitosť najhoršieho prípadu), ale v prípade, že poznáme (resp. odhadneme) početnosť prístupov do jednotlivých prvkov reprezentovanej množiny, môžme skonštruovať BVS, ktorý bude minimalizovať čas, strávený prehľadávaním BVS počas celého výpočtu (čo je vlastne optimalizovanie zložitosti priemerného prípadu). Predpokladajme, že BVS má reprezentovať n-prvkovú množinu, pričom vieme, že početnosť prvku a_i je p_i. Bez ujmy na všeobecnosti budeme predpokladať, že prvky sú utriedené, teda $a_1 \leq a_2 \leq \ldots \leq a_n$. Koľko vrcholov BVS T prezrieme, ak budeme p_1 razy hľadať prvok a_1, p_2 razy prvok a_2, $\ldots$?

$$c(T) = \sum_{i=1}^{n} h_i p_i$$

kde h_i je počet vrcholov v BVS T na ceste z koreňa do vrchola reprezentujúceho a_i. Je celkom prirodzené, že sa budeme snažiť konštruovať OBVS T, ktorý je optimálny vzhľadom na mieru $c(T)$.

Pri konštrukcii OBVS[3] robíme v každom kroku rozhodnutie, ktorý z prvkov má byť v danom vrchole. Nech koreňom OBVS je prvok a_i. Potom ľavý (T_L) aj pravý (T_R) podstrom OBVS sú optimálnymi BVS pre množinu prvkov, ktorú reprezentujú. Preto je princíp dedenia optimality splnený a my môžme použiť na konštrukciu

[3] OBVS konštruujeme postupne od koreňa smerom k listom

OBVS metódu dynamického programovania.

Čo vieme povedať o cene stromu T? Platí

$$c(T) = c(T_L) + c(T_R) + \sum_{i=1}^{n} p_i$$

nakoľko každá z ciest sa v porovnaní so stromom T_L, resp. T_R predĺžila o 1.

- nech $cena(i, j)$ označuje cenu optimálneho BVS pre množinu prvkov $a_i, a_{i+1}, \ldots, a_j$
- nech $cena(i, i-1) = 0$
- potom

$$cena(i, j) = \min_{i \leq k \leq j} \{cena(i, k-1) + cena(k+1, j) + \sum_{k=i}^{j} p_k\}$$

- kvôli možnosti konštrukcie OBVS potrebujeme poznať aj hodnotu $k(i, j)$, pri ktorej sa dosahuje $cena(i, j)$.

Úloha:

- Naprogramujte popísaný algoritmus a analyzujte jeho časovú a priestorovú zložitosť. *DÚ*
- Ako sa zmení uvedený algoritmus ak budeme predpokladať, že navyše poznáme aj početnosť vyhľadávania prvkov, ktoré sa v množine, reprezentovanej BVS, nenachádzajú? Teda keď poznáme $q(i)$– početnosť vyhľadávania prvku, ktorý je medzi a_{i-1} a a_i. Pritom predpokladáme, že $q(1)$ je početnosť vyhľadávania prvkov menších ako a_1, $q(n+1)$ je početnosť vyhľadávania prvkov väčších ako a_n.

2.2.4 0/1 Plnenie batoha (0/1 Knapsack problem)

problém Uvažujme nasledujúci problém. Daných je n objektov s váhami w_i a batoh kapacity M. Ak umiestnime do batoha objekt w_i, získame profit p_i. Našou úlohou je naplniť batoh tak, aby sme získali čo možno najväčší zisk. Formálne:

Vstup: $w_i,\ p_i,\ M,\ 1 \leq i \leq n$
Výstup: $\max \sum_{i=1}^{n} x_i p_i$, pričom $\sum_{i=1}^{n} x_i w_i \leq M$ a $x_i \in \{0, 1\}$

Aby sme mohli na riešenie použiť metódu dynamického programovania, musíme riešenie vyjadriť ako postupnosť rozhodnutí a overiť platnosť dedenia optimality.

Označme $KNAP(a, b, Y)$ problém 0/1 plnenia batoha pri vstupoch $w_i,\ p_i,\ a \leq i \leq b,\ M = Y$. Zrejme pôvodný problém 0/1 plnenia batoha je $KNAP(1, n, M)$. Za rozhodnutie možno považovať zaradenie, resp. nezaradenie objektu do batohu.

Nech $y_1, y_2, \ldots, y_n$ je optimálne priradenie hodnôt premenným $x_1, x_2, \ldots, x_n$.
Ak $y_1 = 0$, tak $y_2, \ldots, y_n$ musí byť optimálne riešenie $KNAP(2, n, M)$.
Ak $y_1 = 1$, tak $y_2, \ldots, y_n$ musí byť optimálne riešenie $KNAP(2, n, M - w_1)$.

Označme $g_j(y)$ hodnotu $KNAP(j+1,\ n,\ y)$. Zrejme $g_0(M)$ je hodnota optimálneho riešenia $KNAP(1, n, M)$ a

$$g_0(M) = \max\{g_1(M),\ g_1(M - w_1) + p_1\} \qquad (*)$$

Nech $y_1, y_2, \ldots, y_n$ je optimálne riešenie pre $KNAP(1, n, M)$. Potom $\forall j, 1 \leq j \leq n$,

- $y_1, \ldots, y_j$ je optimálne riešenie pre $KNAP(1, j, \sum_{1 \le i \le j} w_i y_i)$
- $y_{j+1}, \ldots, y_n$ je optimálne riešenie pre $KNAP(j+1, n, M - \sum_{1 \le i \le j} w_i y_i)$

Preto môžme (*) zovšeobecniť:

$$g_i(y) = \max\{g_{i+1}(y), g_{i+1}(y - w_{i+1}) + p_{i+1}\} \qquad (**)$$

Ak si uvedomíme, že $g_n(y) = 0\ \forall y$, môžme postupne počítať $g_{n-1}(y), \ldots$. Tento postup vlastne odpovedal postupnému priraďovaniu hodnôt $x_n, x_{n-1}, \ldots$

Nemusíme postupovať odzadu, ale sa dá aj spredu. Ak označíme $f_j(X)$ hodnotu optimálneho riešenia pre $KNAP(1, j, X)$, tak

$$\begin{aligned} f_n(M &= \max\{f_{n-1}(M),\ f_{n-1}(M - w_n) + p_n\}, \text{ resp.} \\ f_i(X) &= \max\{f_{i-1}(X), f_{i-1}(X - w_i) + p_i\}, \text{ čo súčasne s faktom } f_0(X) = 0 \text{ a} \\ & \quad f_i(X) = -\infty, X < 0 \text{ dáva návod, ako postupne počítať } f_1, f_2, \ldots \end{aligned}$$

Úloha:

DÚ

- Domyslite detaily a naprogramujte riešenie problému 0/1 plnenia batoha.
- Nech C je matica cien ohodnoteného grafu s vrcholmi $1, 2, \ldots, n$; $C[i, j]$ udáva kladnú dĺžku hrany medzi vrcholmi i a j. Použite metódu dynamického programovania na výpočet matice S dĺžok najkratších ciest; $S[i, j]$ bude cena najkratšej cesty z vrchola i do vrchola j, pričom pod dĺžkou cesty rozumieme súčet cien hrán, ktoré cestu tvoria.
- Uvažujme problém hľadania *najdlhšej spoločnej podpostupnosti* dvoch postupností $X = X_1, \ldots, X_n,\ Y = Y_1, \ldots, Y_m$, ktorá vznikne vynechaním niektorých znakov z každej z nich. Označme $LCS(i, j)$ dĺžku najdlhšej spoločnej podpostupnosti postupností $X_1, \ldots, X_i$ a $Y_1, \ldots, Y_j$; zujíma nás zrejme $LCS(n, m)$. (LCS(ABAKUS, AUTOBUS)=4; najdlhšia spoločná podpostupnosť je ABUS)Základom algoritmu pre výpočet $LCS(n, m)$ založenom na dynamickom programovaní je nasledujúci vzťah:

  $$LCS(i, j) = \begin{cases} 1 + LCS(i-1, j-1), & \text{ak } X_i = Y_j \\ \min\{LCS(i-1, j), LCS(i, j-1)\}, & \text{inak} \end{cases}$$

  Napíšte program, ktorý nielen vypočíta $LCS(n, m)$, ale aj príslušnú najdlhšiu spoločnú podpostupnosť. Aká je zložitosť Vášho programu?

- Majme ohodnotený graf $G = (V, E)$, zadaný maticou cien: $C[i, j]$ je cena hrany z vrchola i do vrchola j. Našou úlohou je nájsť takú permutáciu π vrcholov $2, 3, \ldots, n-1$, ktorá minimalizuje dĺžku kružnice $1, \pi, 1$; pod cenou kružnice rozumieme súčet cien jej hrán. Problému sa hovorí problém obchodného cestujúceho (TSP)

  Nech $S = \{1, 2, \ldots, n\},\ j \in S$. Označme $C(S, j)$ dĺžku najkratšej cesty $1\pi j$, kde π je permutácia množiny $S - \{1, j\}$.

  Základom riešenie problému TSP dynamickým proramovaním je nasledujúci vzťah

  $$C(S, j) = \min_i \{C(S - \{j\}, i) + C[i, j]\}$$

  Napíšte program, ktorý rie3i TSP dynamickým programovaním. Aká je zložitosť Vášho programu?

2.3 Greedy algoritmy

Je to jedna z veľmi rozšírených metód, ktorá sa väčšinou používa pri riešení optimalizačných úloh. Tieto úlohy často možno charakterizovať nasledovne:

- Vstupom je n-prvková množina, resp. postupnosť
- Cieľom je vytvoriť podmnožinu (podpostupnosť), ktorá spĺňa nejaké podmienky (odrážajúce charakter problému). Každé riešenie, ktoré spĺňa tieto ohraničenia, sa nazýva *prípustné riešenie.*
- Navyše je daná je účelová funkcia, definovaná na množine prípustných riešení.

Výstupom je to prípustné riešenie, ktoré optimalizuje účelovú funkciu. Takéto riešenie voláme *optimálne riešenie.*

Snažíme sa nájsť *rýchly* algoritmus, ktorý generuje priamo optimálne riešenie. Greedy metóda vedie k algoritmom, ktoré pracujú v taktoch. V každom takte sa vyberie najvhodnejší kandidát na zaradenie do optimálneho riešenia. Ak vybratý kandidát nemôže byť do riešenia zaradený (nespĺňa podmienky prípustného riešenia, dokázateľne nemôže byť súčasťou optimálneho riešenia, ...), definitívne sa vyradí z množiny potenciálnych kandidátov. Pri výbere kandidátov zväčša aplikujeme "pažravý" prístup - snažíme sa zohľadňovať optimalizačné kritérium.

Algoritmus 5 Greedy(A,n)

1: riešenie $\leftarrow \emptyset$;
2: **for** i=1 to n **do**
3: $\quad X \leftarrow SELECT(A)$;
4: $\quad$ **if** PRÍPUSTNÉ(riešenie, X) **then** riešenie $\leftarrow$ UNION(riešenie, X)
5: return(riešenie)

Uvedomme si, čo ovplyvňuje úspešnosť takejto metódy

- SELECT – má vplyv nielen na zložitosť, ale i na to, či získané riešenie bude optimálne
- PRÍPUSTNÉ – predpokladáme, že vieme rozhodnúť, či daný prvok bude súčasťou (optimálneho) riešenia. Zrejme ovplyvňuje nielen zložitosť, ale aj kvalitu.

Príklad 2.3 *Daná je množina hodnôt mincí a suma, ktorú treba zaplatiť. Hodnotou účelovej funkcie je počet mincí, ktorý sa snažíme minimalizovať.* *mince*

Greedy metóda vedie k nasledovnému postupu — začni s bankovkami s maximálnou možnou hodnotou a plať, kým sa dá. Potom vezmi bankovky s menšou hodnotou a plať, kým sa dá,...

Ak vezmeme ako základnú množinu mincí 1,2,5,10,..., vedie greedy metóda k optimálnemu riešeniu. Čo však, ak vezmeme ako množinu mincí 1,9, 11, a suma, ktorú potrebujeme zaplatiť, je 19?

Greedy algoritmus vedie k prípustnému riešeniu – 11,1,1,1,1,1,1,1,1. Optimálne riešenie je však 9,9,1.

Úloha: Pre aké hodnoty mincí vedie greedy metóda k optimálnemu riešeniu? *DÚ*

2.3.1 Plnenie batoha (s racionálnymi koeficientami)

Uvažujme nasledovný problém *problém*

Vstup: n objektov určených váhou a ziskom a kapacita batoha
- M kapacita batoha
- w_i váha i-teho objektu
- p_i profit; ak prenesieme časť x_i objektu i, $0 \leq x_i \leq 1$, tak získame cenu (profit) $p_i x_i$

Úloha Naplniť batoh tak, aby sme maximalizovali zisk $Z = \sum_{i=1}^{n} p_i x_i$, pričom
1. $\sum_{i=1}^{n} w_i x_i \leq M$
2. $0 \leq x_i \leq 1,\ p_i, w_i > 0, 1 \leq i \leq n$

Prípustným je každé také riešenie $(x_1, x_2, \ldots, x_n)$, ktoré spĺňa (1.) a súčasne (2.) Optimálnym riešením je to prípustné riešenie, ktoré maximalizuje zisk Z.

Príklad 2.4 *Majme vstup*

$$n = 3,\ M = 20,\ (p_1, p_2, p_3) = (25, 24, 15), (w_1, w_2, w_3) = (18, 15, 10)$$

Prípustné riešenia:	$\sum_{i=1}^{n} w_i x_i$	$\sum_{i=1}^{n} p_i x_i$
1. $(1/2, 1/3, 1/4)$	16.5	24.25
2. $(1, 2/15, 0)$	20	28.2
3. $(0, 2/3, 1)$	20	31
4. $(0, 1, 1/2)$	20	31.5

Zrejme ak $\sum_{i=1}^{n} w_i \leq M$, tak položíme $x_i = 1$ a máme optimálne riešenie. Preto predpokladajme, že $\sum_{i=1}^{n} w_i > M$.

Existuje viacero greedy stratégií, ktoré môžme skúsiť použiť:

1. plníme podľa maximálneho zisku, ktorý by priniesol celý objekt – nie vždy dáva optimálne riešenie
2. podľa najmenšej váhy
3. podľa maximálneho relatívneho profitu p_i/w_i – toto vedie k optimálnemu riešeniu

Všimli sme si, že sme mohli brať do úvahy tri miery – *váhu, profit, relatívny zisk* – podľa toho sa menilo poradie, v ktorom sme objekty skúmali. Pritom len jeden z prístupov vedie k optimálnemu riešeniu.

korektnosť greey pre batoh

Fakt 2.3 *Greedy algoritmus založený na maximálnom relatívnom zisku dáva optimálne riešenie.*

Dôkaz: Pre jednoduchosť predpokladajme, že $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \frac{p_3}{w_3} \geq \ldots \geq \frac{p_n}{w_n}$. Majme

$X = (x_1, x_2, \ldots, x_n)$ riešenie algoritmu greedy
$Y = (y_1, y_2, \ldots, y_n)$ optimálne riešenie

pričom predpokladáme, že sú rôzne.

Ukážeme, ako modifikáciami optimálneho riešenia, ktoré nezhoršujú jeho kvalitu, transformujeme toto optimálne riešenie na greedy riešenie. Z toho bude vyplývať, že greedy riešenie je optimálne.

Nech j je minimálny index taký, že $x_j \neq 1$. Ak X nie je optimálne riešenie, tak pre optimálne riešenie Y platí $\sum p_i x_i < \sum p_i y_i$ a navyše môžme predpokladať, že $\sum w_i x_i = M$. Nech k je minimálny index taký, že $x_k \neq y_k$.

$k < j$	$x_k = 1,\ y_k < 1 \Rightarrow x_k > y_k$	
$k = j$	$\Rightarrow x_k > y_k$	
$j < k$	nie je možné, lebo by sme dostali $\sum w_i y_i > M$	

Preto $\mathbf{x_k} > \mathbf{y_k}$. Nech $Z = (z_1, z_2, \ldots, z_n)$ je riešenie, ktoré získame nasledovnou úpravou optimálneho riešenia:

$$z_i = x_i, \quad i \leq k;$$
$$z_i, \qquad \sum_{k<i\leq n} w_i(y_i - z_i) = w_k(z_k - y_k).$$

Dostávame

$$\begin{aligned}\sum p_i z_i &= \sum p_i y_i + \tfrac{(z_k-y_k)w_k p_k}{w_k} - \sum_{k<i\leq n}(y_i - z_i)\tfrac{w_i p_i}{w_i}, \quad \tfrac{p_i}{w_i} \leq \tfrac{p_k}{w_k} \qquad \Rightarrow \\ &\geq \sum p_i y_i + \underbrace{[(z_k - y_k)w_k - \sum_{k<i\leq n}(y_i - z_i)w_i]}_{\geq 0}\tfrac{p_k}{w_k} \\ &= \sum p_i y_i\end{aligned}$$

– Ak $X = Z$, vtedy je X optimálne
– Ak $Z \neq X$, vtedy $X \leftarrow Z$ a aplikuj uvedený postup. Po konečnom počte krokov musí nastať situácia, že $X = Z$.

2.3.2 Úlohy s terminovaním

Majme n úloh a jeden procesor. O každej úlohe u_i vieme, dokedy musí byť spočítaná (deadline $d(i)$) a aký zisk (profit $p(i)$) budeme mať, ak ju načas zrátame. Predpokladáme, že každá úloha sa počíta jednotku času.

Úloha: Vybrať podmnožinu úloh, ktoré všetky môžu byť spočítané načas tak, aby sme maximalizovali zisk.
Prípustné riešenie: Podmnožina J úloh (resp. ich indexov), ktoré sa dajú spracovať pred deadlinom (vrátane)
Hodnota súčet získaných profitov
Optimum: maximálna hodnota

Príklad 2.5 *Majme*

	p_i	d_i
1	*100*	*2*
2	*10*	*1*
3	*15*	*3*
4	*27*	*1*

prípustné riešenie	*postup*	*zisk*
1,2	*2,1*	*110*
1,3,2	*2,1,3*	*125*
1,4,3	*4,1,3*	*142*

Ako máme hľadať optimálne riešenie? Jednou možnosťou je pre všetky podmnožiny úloh overiť, či sa dajú usporiadať tak, aby bola každá vypočítaná načas a ak áno, spočítať získaný zisk. Potom stačí vybrať tú podmnožinu, ktorá prináša najväčší zisk. Takéto riešenie ale vedie k exponenciálnej zložitosti.

Použijeme preto greedy metódu a dokážeme, že dáva optimálne riešenie. K popisu stačí určiť, akú statégiu na výber kandidáta volíme a ako rozhodujeme o tom, či je daný kandidát vhodný:

SELECT

- Vyberáme prvok, ktorý má maximálny profit. Pri rovnosti profitov zachovávame relatívne poradie zo vstupu.

PRÍPUSTNÉ
Ako overiť, či skúmaná podmnožina indexov odpovedá takej množine úloh, ktoré sa dajú zrátať načas? Prezeranie všetkých usporiadaní je neefektívne. Ľahko vidno, že ak má byť úloha dopočítaná v čase t, musí byť nanajvýš $t-1$ úloh počítaných pred ňou.

- Majme nejaké poradie úloh π. Ak pre každú úlohu je poradové číslo úlohy v π nanajvýš rovné jej deadlinu, dajú sa úlohy spočítať v poradí π a všetky budú načas. Preto usporiadame úlohy podľa deadlinov neklesajúco, pričom pri rovnosti deadlinov zachovávame relatívne poradie zo vstupu.
- Potom v čase $O(n)$ vieme overiť, či *Prípustné(Riešenie, X)* dáva hodnotu TRUE.

V našom prípade:

Riešenie = $\{1\}$
Riešenie = $\{1\}$, $X = 4$, Prípustné(1,4) = True $\Rightarrow$ Riešenie = $\{1, 4\}$
Riešenie = $\{1, 4\}$, $X = 3$, Prípustné(1,4,3) = True $\Rightarrow$ Riešenie = $\{1, 4, 3\}$
Riešenie = $\{1, 4, 3\}$, $X = 2$, Prípustné(1,4,3,2) = False $\Rightarrow$ Riešenie = $\{1, 4, 3\}$

Odpovedajúci postup: 4,1,3

korektnosť greedy riešenia pre úlohy s terminovaním

Dôkaz: Musíme vyargumentovať optimalitu riešenia, získaného greedy metódou. Nech
$G = \{g_1, \ldots, g_k\}$ je riešenie získané greedy metódou
$O = \{o_1, \ldots, o_\ell\}$ je optimálne riešenie.

Predpokladáme

- G, O sú usporiadané nerastúco podľa profitov
  $p(g_1) \geq p(g_2) \geq \ldots \geq p(g_k)$ a $p(o_1) \geq p(o_2) \geq \ldots \geq p(o_\ell)$
- ak dve úlohy majú rovnaký profit, tak je zachované relatívne poradie zo vstupu
  $p(i_q) = p(i_{q+1}) \Rightarrow i_q < i_{q+1}$

Majme $\mathbf{G} \neq \mathbf{O}$

$\mathbf{G} \subset \mathbf{O}$ V tomto prípade optimálne riešenie O obsahuje úlohu u, ktorú neobsahuje greedy riešenie G. Keďže greedy metódou bola každá úloha uvažovaná ako potenciálny kandidát na zaradenie do G, bola zvažovaná aj úloha u. Jej nezaradenie do G znamená, že jej pridanie k G znemožnilo zrátania úloh načas. To znamená, že by tomu muselo byť tak aj v prípade riešenia O. Preto **nemôže nastať**

$\mathbf{O} \subset \mathbf{G}$ Tento fakt by implikoval, že O nie je optimálne riešenie. Preto **nemôže nastať**

Z uvedeného vyplýva, že **existuje a (minimálne také), že $\mathbf{g_a} \neq \mathbf{o_a}$**. Ukážeme, že ak zaradíme g_a do O, budeme vedieť vylúčiť z O také o_s, že takouto zámenou nezhoršíme kvalitu riešenia O a zachováme jeho riešiteľnosť. Po konečnom počte takýchto krokov skončíme v situácii, keď $O = G$, čo bude dokazovať optimalitu riešenia G.

- môže patriť g_a do O? Nie, pretože:

  ak $p(g_a) > p(o_a)$, tak ďalej sú už len úlohy s menším profitom

  ak $p(g_a) = p(o_a)$, tak $g_a < o_a$

- ako hľadať o_s?
  o_s odpovedá prvej úlohe spomedzi úloh s indexom $o_t, t \geq a$, ktorá má najmenší deadline

Keďže $p(g_a) \geq p(o_a) \geq p(o_s)$, kvalitu sme nepokazili. Ostáva ukázať, že sme zachovali riešiteľnosť. Inými slovami, že $O' = O \setminus \{o_a\} \cup \{o_s\}$ je množina indexov odpovedajúca úlohám, ktoré sa dajú spočítať tak, že každá z nich je spočítaná načas.

Preusporiadajme úlohy O podľa deadlinov nerastúco. Získame postupnosť α, o_s, β. Vzhľadom k voľbe indexu s je zrejmé, že všetky úlohy s indexom z α sa nachádzajú aj v riešní G získanom greedy metódou. Existuje preto usporiadanie α' postupnosti indexov α, o_s, ktoré odpovedá nerastúcim deadlinom. Navyše, ak by sme úlohy riešili v tomto poradí, bude každá z nich vyriešená načas. Preto

$$\alpha', g_a, \beta$$

je poradie, v ktorom je každá z úloh s indexom z O' spočítaná načas.

Úloha: Zamyslite sa nad implementáciou a zložitosťou.

DÚ

2.3.3 Optimálne zlučovanie súborov

Majme n utriedených súborov $F(1), F(2), \ldots, F(n)$. Úlohou je napísať zlučovací algoritmus na vytvorenie jediného utriedeného súboru F. Pritom vieme, že pre zlučovanie dvoch utriedených súborov poznáme efektívny algoritmus (lineárnej zložitosti). Vzhľadom k tomu, že porovnanie dvoch prvkov je zanedbateľné vzhľadom k prístupu do súboru, je prirodzenou mierou zložitosti celkový počet prístupov k prvkom. Presnejšie: $\sum p(i)|F(i)|$, kde $|F(i)|$ je počet prvkov v súbore $F(i)$ a $p(i)$ je počet zlučovaní, v ktorých sa zúčastňujú prvky pôvodne uložené v súbore $F(i)$.

Majme nejaký zlučovací algoritmus. K nemu možno priradiť binárny zlučovací strom nasledovným spôsobom:

- listy stromu reprezentujú súbory, ktoré máme zlučovať
- vnútorné vrcholy reprezentujú súbory, ktoré vznikajú v priebehu zlučovania; $F(i)$ je súbor, ktorý vznikol v $(i-n)$-tom zlučovaní, $i > n$.

Ľahko vidno, že k jednému algoritmu je zlučovací strom priradený jednoznačne až na izomorfizmus. Na druhej strane jeden zlučovací strom reprezentuje viacero algoritmov.

Zrejme zložitosť algoritmu možno vyjadriť ako $\sum h(i) \cdot |F(i)|$, kde $h(i)$ je hĺbka listu reprezentujúceho súbor $F(i)$; $\sum h(i) \cdot |F(i)|$ sa volá vážená dĺžka ciest. Optimálny zlučovací algoritmus je taký, pre zlučovací strom ktorého je $\sum h(i) \cdot |F(i)|$ minimálne.

Riešme greedy metódou.

SELECT
V každom kroku zlučuj dva súbory s najmenšou veľkosťou.

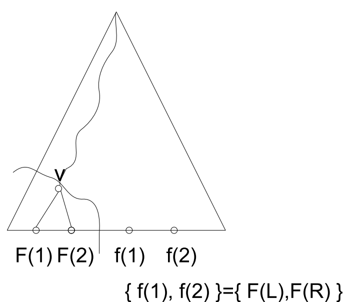
Kvôli *korektnosti* predpokladajme, že máme dva stromy - TO(optimálny zlučovací algoritmus) a TG(greedy). Indukciou ukážeme, že z hľadiska váženej dĺžky ciest zlučovacieho stromu je TG rovnako dobrý ako TO.

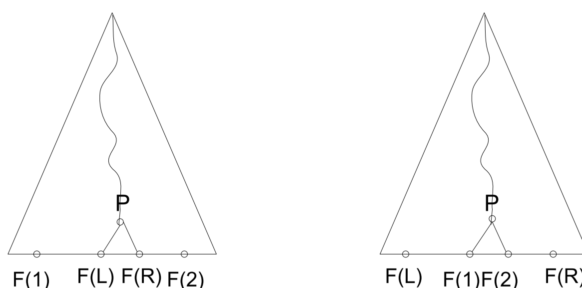
korektnosť

Pre triviálne prípady $n = 1, 2$ tvrdenie zrejme platí

Báza indukcie

Nech tvrdenie platí pre zlučovanie m súborov, $m < n$. *IP*





Indukčný krok:

- Nech $|F(1)| \leq |F(2)| \leq \ldots \leq |F(n)|$. Po prvom kroku greedy algoritmu sa vytvorí nový vrchol v so synmi $F(1), F(2)$.
- Nech v zlučovacom strome TO je P vnútorný vrchol najďalej od koreňa. Nech $F(L), F(R)$ sú súbory reprezentované ľavým, resp. pravým synom vrchola P. Nech $f(1), f(2)$ sú vrcholy stromu TO reprezentujúce súbory $F(1), F(2)$. Spravme v strome TO zámenu tak, aby sa $f(1)$ stalo ľavým synom vrchola P, $f(2)$ pravým synom vrchola P. Tým sme dostali nový zlučovací strom NTO.

  Všimnime si , ako sa zmenila cena $c(NTO)$ vážených ciest stromu NTO oproti cene $c(TO)$ stromu TO.

  $$c(NTO) - c(TO) = h(L)F(L) + h(R)F(R) + h(1)F(1) + h(2)F(2)$$
  $$- \{h(L)F(1) + h(R)F(2) + h(1)F(L) + h(2)F(R)\}$$

  Keďže
  $$h(L) \leq h(1),\ h(R) \leq h(2),\ F(1) \leq F(L),\ F(2) \leq F(r)$$
  je $c(NTO) - c(TO) \leq 0$. Oba stromy TG aj na NTO sú zlučovacie stromy pre zlučovanie n-1 súborov $F(1)+F(2), F(3), ..., F(n)$. Preto je podľa IP cena TG rovnaká ako cena NTO a teda greedy algoritmus dáva optimálne riešenie.

□

2.3.4 Minimálna kostra.

Posledným príkladom použitia greedy metódy bude algoritmus konštrukcie minimálnej kostry v grafe.

problém

Daný je súvislý neorientovaný ohodnotený graf. Kostra je podgraf tohto grafu, ktorý obsahuje všetky vrcholy a tvorí strom. Cena kostry je súčet cien priradených hranám, ktoré tvoria kostru. Chceme skonštruovať kostru s minimálnou cenou.

riešenie

SELECT: sú dva základné princípy

- **PRIM**: vyber hranu s minimálnou cenou tak, aby čiastočné riešenie tvorilo strom
- **KRUSKAL**: vyber hranu s minimálnou cenou tak, aby čiastočné riešenie tvorilo les

Ukážeme *korektnosť* Kruskalovho algoritmu:

korektnosť

Nech K je kostra vytvorená Kruskalovým algoritmom, O optimálna kostra. Označme

$E(K)$ množinu hrán kostry K
$E(O)$ množinu hrán kostry O.

Ak $E(O) = E(K)$, potom majú obe kostry rovnakú cenu. Predpokladajme teda, že $\mathbf{E(O)} \neq \mathbf{E(K)}$ a uvažujme hranu $e \in E(K) \setminus E(O)$ s *minimálnou* cenou. Pridanie hrany e k $E(O)$ spôsobí vznik jediného cyklu $e, e(1), e(2), \ldots, e(k)$ v O. Zrejme aspoň jedna z hrán $e(1), e(2), \ldots, e(k)$ nepatrí do $E(K)$[4]. Nech je to hrana $e(j)$. Potom $cena(e(j)) \geq cena(e)$(ak by to neplatilo, bol by Kruskalov algoritmus uvažoval hranu $e(j)$ skôr ako hranu e.)

Zmena $E(O) \leftarrow E(O) \setminus \{e\} \cup \{e(j)\}$ nezväčší cenu takto vzniknutej kostry NO. Pritom

$$|E(K) \setminus E(NO)| < |E(K) \setminus E(O)|$$

Takto postupnými modifikáciami, ktoré nezhoršujú cenu, dostaneme z kostry O kostru K.

□

[4] prečo?

Kapitola 3

Metódy založené na prehľadávaní stavového priestoru

Uvažujme taký typ úloh, ktoré možno charakterizovať nasledovným spôsobom. Výstupom úlohy je n-tica (množina n-tíc, prípadne postupnosť), ktorá spĺňa nejaké ohraničenia. Pritom požadované riešenie možno napísať v tvare $(x_1, x_2, \ldots, x_n)$, kde $x_i \in S_i$, kde S_i je **konečná** množina.

Príklad 3.1 ***Problém n-dám.*** *Na šachovnicu $n \times n$ chceme umiestniť n dám tak, aby v každom riadku a v každom stĺpci stála dáma a aby sa navzájom neohrozovali. Riešenie hľadáme vo forme vektora dĺžky n, kde na s-tej pozícii je číslo riadku, v ktorom je v s-tom stĺpci umiestnená dáma.*

n dám na šachovnici

Iným, nevhodným problémom, je aj problém triedenia, ktorý možno naformulovať aj takto: keď triedime n-prvkovú množinu, výsledkom môže byť n-tica indexov $i_1, i_2, \ldots, i_n$, kde i_j je index j-teho najmenšieho prvku zo vstupu.

Všetky potenciálne n-tice tvoria potenciálny *stavový priestor* úlohy/*priestor riešení*. Vďaka konečnosti jednotlivých množín S_i je tento priestor konečný. Keďže však stavový priestor úlohy obsahuje minimálne všetky potenciálne riešenia, je jeho veľkosť aspoň $|S_1| \cdot |S_2| \cdot \ldots \cdot |S_m|$.

Konečnosť priestoru riešení umožňuje hľadať požadované riešenie napríklad

metódou hrubej sily – vygenerujeme všetky prípustné riešenia a z nich vyberieme tie, ktoré spĺňajú nami požadované požiadavky. Zložitosť je určite aspoň veľkosť stavového priestoru, čo je veľa.

Budeme sa snažiť vektor budovať postupne, pričom pre každý vygenerovaný začiatok vektora zistíme, či sa dá predĺžiť na riešenie. Má to tú výhodu, že ak pre nejaký začiatok $x_1, x_2, \ldots, x_k$ zistíme, že sa nedá predĺžiť, potom minimálne

$$|S_{k+1}| \cdot \ldots \cdot |S_n|$$

prípadov nemusíme generovať.

Podmienky, ktoré má riešenie spĺňať, sú väčšinou dvojakého charakteru

– *explicitné*, ktoré hovoria o x_i ako takom. Špecifikujú hodnoty, ktoré môže x_i nadobúdať ($x_i \in S_i, x_i \leq 20, \ldots$). Tieto podmienky definujú stavový priestor problému

– *implicitné*, ktoré hovoria o vzťahu jednotlivých x_i medzi sebou (dámy sa nemajú ohrozovať, ...), resp. ktoré spĺňajú nejakú ohraničujúcu funkciu

čiastočné súčty

Príklad 3.2 ***Problém čiastočných súčtov***

Vstup: $w_1, w_2, \ldots, w_n,\ M; w_i, M \geq 0$
Výstup: *množina indexov* $I = \{i_1, i_2, \ldots, i_k\}$ *takých, že* $\sum_{t \in I} w_t = M$.

Čo sú *explicitné* podmienky? $1 \leq i_j \leq n$

Implicitné podmienky sa viažu na medzisúčty: $\sum_{t=i_1}^{i_j} w_t \leq M$

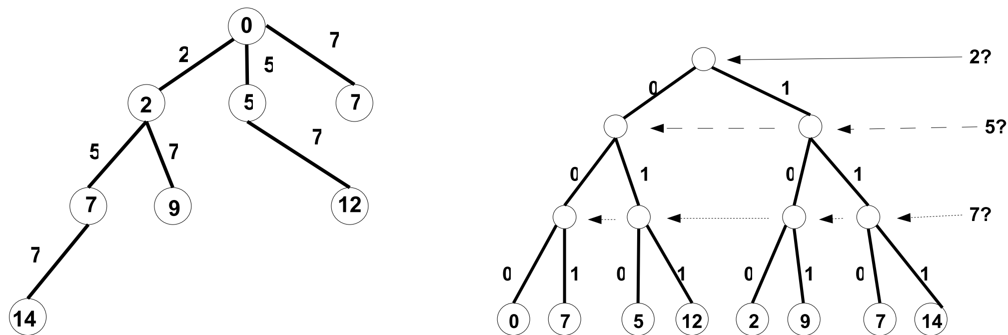
Kvôli efektívnosti algoritmov pri **prehľadávaní stavového priestoru** vyžadujeme **systematickosť** + **efektívnosť**, čo vedie k použitiu **stromov**.

Zamyslime sa nad tým, ako máme **organizovať** strom stavového priestoru[1].

typ stromu

statický : konštrukcia stromu nezávisí od od konkrétneho vstupu; vrchol vyzerá na každej úrovni rovnako – na i-tej úrovni rozhodujeme o zaradení alebo nezaradení w_i do príslušnej sumy. Každý vnútorný vrchol má teda práve dvoch synov. Riešeniu odpovedá cesta z koreňa do príslušného listu

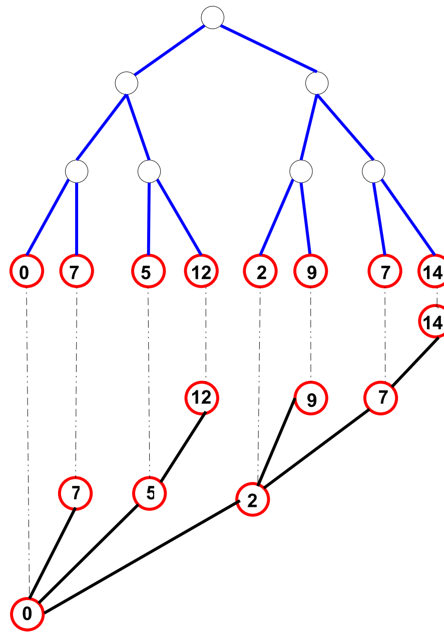
dynamický : konštrukcia stromu je závislá na vstupe; "vzhľad" vrcholov sa líši s úrovňou v strome. Napr: ak budeme (bez ujmy na všeobecnosti) predpokladať, že $i_1 \leq i_2 \leq \ldots \leq i_k$, tak vrchol, do ktorého sme prišli po hrane označenej h, má $n - h$ synov, do ktorých ideme po hranách $h+1, h+2, \ldots, n$. V tomto prípade potenciálnemu riešeniu odpovedá každý vrchol vytvoreného stromu.



Na obrázku vidíme dynamický (vľavo) a statický (vpravo) strom pre problém čiastočných súčtov pri vstupe $w_1 = 2, w_2 = 5, w_3 = 7$. V prípade dynamického stromu tvoria potenciálne riešenia vrcholy, ktoré sú pospájané do stromu. Statický strom je vybudovaný *nad* listami, v ktorých sú uložené riešenia.

Keď máme predstavu o tom, ako by vyzeral strom stavového priestoru problému, musíme systematicky "generovať" vrcholy tohto stromu. K lepšiemu popisu budeme uvažovať 3 kategórie vrcholov:

[1] Strom stavového priestoru nevytvoríme na začiatku; jeho časti budeme v priebehu prehľadávania generovať, resp. navštevovať.



kategórie vrcholov

E(expand) – vrchol, ktorý sa práve vybral na spracovanie

L(live) – živý vrchol, teda taký, ktorý sme už vygenerovali, ale ešte sa doň budeme vracať

D(dead) – mŕtvy vrchol, ktorý už má spracované všetky deti, resp. sme zistili, že sa z neho nevieme dostať do prípustného riešenia.

"Notoricky známe" sú dva systematické postupy prehľadávania stromov a grafov. Z nich vychádzajú aj dve základné metódy:

Branch & Bound	Backtracking
⇓	⇓
do šírky + orezanie	do hĺbky + orezanie

Dobre si všimnime slovíčko *orezenie*; bez neho by tieto metódy boli metódou hrubej sily...

3.1 Backtracking-prehľadávanie s návratom

Spomedzi "priestor prehľadávajúcich metód" začneme metódou prehľadávania s návratom – tzv. backtrackingom.

metóda

Nech $T(X(1), \dots, X(k-1))$ je množina potenciálnych hodnôt pre $X(k)$

$B_k(X(1), \dots, X(k))$ je booleovská funkcia, ktorá dáva hodnotu true práve vtedy keď $X(1), ..., X(k)$ je alebo môže byť predĺžené na riešenie.

Potom backtracking možno schématicky znázorniť algoritmom 6, pričom efektívnosť tejto metódy zrejme záleží od
– výpočtu $T(\cdot)$, mohutnosti $T(\cdot)$
– výpočtu $B_k(\cdot)$, počtu prvkov, spĺňajúcich $B_k(\cdot)$.

Zamyslime sa nad tým, či a ako môžme ovplyvniť počet vrcholov stromu, ktoré pri prehľadávaní stavového priestoru vygenerujeme.

Algoritmus 6 BACKTRACK

1: $k \leftarrow 1$
2: **while** $k > 0$ **do**
3:    **if** $\exists\ X(k);\ X(k) \in T(X(1), ..., X(k-1))$ a $B_k(X(1), ..., X(k))$ **then**
4:       **if** X(1),...,X(k) je cesta do odpoveďového vrchola **then**
5:          **return** X(1),...,X(k)
6:       $k \leftarrow k+1$
7:    **else** $k \leftarrow k-1$

skúšame zmenšiť veľkosť prezretého priestoru

- ak nezáleží na poradí dopĺňaných položiek výsledného vektora, zdá sa rozumné generovať položky v poradí, ktoré zodpovedá neklesajúcemu usporiadaniu mohutností príslušných S_i. Intuícia za tým – ak zistíme, že sa prvých k položiek *nedá* doplniť na riešenie, potom $S(k,m) = |S_{k+1}| \cdot \ldots \cdot |S_m|$ prípadov nemusíme generovať. Chceli by sme, aby $S(k,m)$ bolo čo možno najväčšie číslo.

- pre zvolenú postupnosť $i_1, i_2, \ldots, i_n$ skúsime odhadnúť, koľko vrcholov bude treba vygenerovať. Ako? Nech pre vrchol v hĺbke 1 vygenerujeme m_1 synov spĺňajúcich B_1. Náhodne sa presuňme do jedného z nich a zistime, koľko má synov, spĺňajúcich B_2. Nech je to $m_2 \ldots$ Nech m_i je počet synov vrchola v hĺbke i, ktorí spĺňajú B_i. Potom počet vygenerovaných vrcholov odhadneme ako
$$m_1 + m_1m_2 + m_1m_2m_3 + \ldots + m_1m_2\ldots m_{n-1}$$
Zopakovaním pre niekoľko zvolených permutácií môžme vybrať tú permutáciu, pre ktorú nám odhad vychádza najlepšie.

opäť problém súčtov

Vráťme sa k problému čiastočných súčtov. Predpokladáme, že
$$w_1 \leq w_2 \leq \ldots \leq w_n, M \in \mathbb{N}$$

Budeme aplikovať statický prístup. Hľadáme teda vektor dĺžky n, ktorého i-ta položka hovorí o prítomnosti, resp. neprítomnosti w_i v súčte. Ako bude vyzerať ohraničujúca funkcia?

Predpokladajme, že $\sum\limits_{i=1}^{k-1} x_iw_i < M$

- Kedy môžme uvažovať, že $\mathbf{x_k = 1}$?

  – Ak $\sum_{i=1}^{k} w_ix_i \leq M$ a $\sum_{i=1}^{k} w_ix_i + \sum_{i=k+1}^{n} w_i \geq M$, tak áno

  – Ak $\sum_{i=1}^{k} w_ix_i < M$ a $\sum_{i=1}^{k} w_ix_i + w_{k+1} > M$, vtedy možnosť $x_k = 1$ nemôže viesť k riešeniu a preto ju neaplikujeme.

- Kedy môžme uvažovať, že $\mathbf{x_k = 0}$?
Vtedy, ak $\sum_{i=1}^{k-1} w_ix_i + \sum_{i=k+1}^{n} w_i \geq M$

Označme
$$s = \sum_{i=1}^{k-1} w_ix_i \qquad\qquad r = \sum_{i=k+1}^{n} w_i$$

Algoritmus 7 SumSub(s,k,r)

$x(k) \leftarrow 1$;
if s+w(k) = M **then return** $(x(1), ..., x(k))$
else if s + w(k) + w(k+1) $\leq$ M **then**
  SumSub(s+w(k), k+1, r-w(k))
if s+r-w(k)$\geq$ M a s+w(k+1) $\leq$ M **then**
  $X(k) \leftarrow 0$
  Sum-of-sub(s,k+1,r-w(k))

3.2 LC Branch and Bound

Prehľadávanie do hĺbky i do šírky sú vlastne "slepé" metódy – postup prehľadávania je daný dopredu a príliš nezohľadňuje kvalitu získavaného riešenia. Metóda LC Branch and Bound (LC-B&B) sa snaží túto "sleposť" odstrániť. Rieši sa to určením inteligentnej funkcie $c(x)$, ktorou ohodnotíme živé vrcholy. Hodnotu tejto funkcie potom využijeme pri voľbe nového E-vrchola.

ohodnocujeme vrcholy

Čo by táto funkcia mala odrážať? Ideálne by bolo, keby hovorila o úsilí, ktoré ešte treba vynaložiť na to, aby sme sa z daného vrchola dostali k odpovedi (resp. o hodnote účelovej funkcie, ktorú z daného vrchola môžme dosiahnuť).

- počet vrcholov, ktoré ešte treba vygenerovať
- počet úrovní, ktoré nás delia od odpovede

Keď chceme presnú hodnotu, znamenalo by to prezretie celého stavového priestoru, preto budeme používať radšej len odhad.

$\widehat{c}(x) = f(h(x)) + \widehat{g}(x)$, kde
  $h(x)$ je cena dosiahnutia x z koreňa
  $f()$ je nejaká neklesajúca funkcia
  $\widehat{g}(x)$ je odhad úsilia do najbližšieho odpoveďového listu

Ak zvolíme $f(x) = 0$, tak vlastne vôbec neuvažujeme úsilie vynaložené na príchod do vrchola. Zodpovedá to tomu, že celkom prirodzene predpokladáme, že $\widehat{g}(x) \leq \widehat{g}(y)$ ak x je synom y. Pritom sa snažíme ísť stále hlbšie a hlbšie a do vedľajšieho podstromu sa už nikdy nedostaneme. Keby $\widehat{g}(x)$ bola reálna hodnota a nie len odhad, bol by to dobrý prístup. Preto $f(x) \neq 0$ dáva možnosť prechodu do vedľajšieho podstromu.

Vyhľadávanie, ktoré zo živých vrcholov vyberá podľa minimálnej hodnoty $\widehat{c}(x)$ sa volá **LC-SEARCH**.

BFS $\widehat{g}(x) = 0,\ f(h(x)) =$ úroveň x
DFS $f(h(x)) = 0,\ \widehat{g}(x) < \widehat{g}(y)$ pre x dieťa y

Často riešime optimalizačné úlohy. Vieme zaručiť, že pomocou metódy LC-B&B dosiahneme minimum (optimum)?

vlastnosti metódy LC-B &B

Uvažujme nasledovnú funkciu

$$\mathbf{c(x)} = \begin{cases} \text{cena cesty z koreňa do x}, & \text{ak x je list s odpoveďou} \\ \infty, & \text{ak x je list bez odpovede} \\ \min\{c(y) \mid y \text{ je list stromu s koreňom x}\}, & \text{ak x je vnútorný vrchol} \end{cases}$$

Ľahko vidno, že ak by LC-B&B používalo pri rozhodovaní funkciu $c(x)$, dosiahli by sme optimálne riešenie. Keďže (z hľadiska zložitosti) efektívny výpočet $c()$ pri riešení ťažkých problémov nemáme, používame odhad $\widehat{c}(x)$. Použitie $\widehat{c}(x)$ namiesto $c(x)$ vo všeobecnosti optimálne riešnie nedáva. Platí ale nasledujúca veta.

Veta 3.1 *$\widehat{c}(x)$ nech je odhad $c(x)$ taký, že*

$$\widehat{c}(x) < \widehat{c}(z) \Leftrightarrow c(x) < c(z)$$

Potom algoritmus LC používajúci $\widehat{c}(x)$ namiesto $c(x)$ nájde riešenie minimálnej ceny a skončí.

ohraničovanie

Ak používame odhad $\widehat{c}(x)$ taký, že $\widehat{c}(x) \leq c(x)$, tak $\widehat{c}(x)$ vlastne dáva dolný odhad. Ak by sme mali aj horný odhad U, tak všetky živé vrcholy s $\widehat{c}(x) > U$ možno vylúčiť. Ako získame U?
– heuristika
– začínajúc s ∞ vylepšujeme podľa toho, čo dosahujeme

Ukážeme použitie pre optimalizačnú úlohu. Ako c budeme používať účelovú funkciu, resp. ako $\widehat{c}(x)$ jej odhad.

3.3 Problém obchodného cestujúceho.

Použitie metódy LC-B &B ilustrujeme na probléme obchodného cestujúceho. O tomto probléme je známe, že je ťažký[2] Daný je ohodnotený orientovaný graf reprezentujúci mestá so vzdialenosťami. Treba prejsť cez všetky mestá, v každom, s výnimkou štartovacieho, treba byť práve raz. Zo štartovacieho mesta výjdeme a potom sa doň vrátime. Treba minimalizovať dĺžku prejdenej cesty, pričom pod dĺžkou cesty myslíme súčet ohodnotení hrán, ktoré tvoria cestu. Bez ujmy na všeobecnosti môžme predpokladať, že cesta začína a končí v meste č.1.

Aké hodnoty pre $\widehat{c}, c, U$?

c(A)

presná cena vrchola

– ak A je list, potom $c(A)$ je dĺžka cesty definovanej cestou z koreňa do listu A
– ak A nie je list, tak $c(A)$ je cena minimálnej ceny listu v podstrome s koreňom A

definovanie odhadu

Jednoduchý odhad $\widehat{c}(x)$ je založený na tzv. redukovanej matici. Hovoríme, že

- riadok(stĺpec) matice je redukovaný, ak obsahuje aspoň jednu 0 a ostatné prvky sú nezáporné;
- **matica je redukovaná**, ak každý riadok a stĺpec, ktorý obsahuje aspoň jeden prvok rôzny od ∞, je redukovaný.

Prečo nás zaujíma redukovaná matica? Nech A je matica cien grafu. Uvedomme si, že každá cesta obchodného cestujúceho (OC) obsahuje práve jednu hranu $A(i,j)$ pre $i = k$ a práve jednu hranu $A(i,j)$ pre $j = k$. Preto keď odpočítame konštantu t od každého prvku v jednom riadku (resp. stĺpci) matice cien A, každá cesta OC sa skráti práve o t. Nemení sa ale relatívny vzťah ciest. Minimálna cesta ostáva minimálnou.

Každému bodu stavového priestoru priradíme redukovanú maticu a cenu nasledovným spôsobom:

[2] Tento prblém je NP-úplný, čo znamená, že máme dosť dôvodov sa domnievať, že preň neexistuje polynomiálny algoritmus.

Algoritmus 8 LC-B&B - Redukcia

Nech A je (aktuálna)matica cien grafu, r označuje cenu redukcie

1: nájdi minimum r_i v riadku $i = 1, \dots, n$. Nech je to prvok $A(i,j)$ na pozícii (i,j)
2: $r \leftarrow r + r_i$
3: odpočítaj r_i od každého prvku v riadku i. Tým na mieste $A(i,j)$ vznikne hodnota 0
4: ak nevznikla redukovaná matica, postupuj analogicky ďalej po tých stĺpcoch, ktoré nie sú redukované, až kým nezískaš redukovanú maticu

- Koreňu priradíme maticu $red(A)$, ktorá vznikla z matice cien vyššie popísaným Algoritmom 8. Cena tohto vrchola je cena redukcie, ktorou sme vyrobili redukovanú maticu v koreni. Uvedomme si, že uvedená hodnota odpovedá tomu, že sa snažíme z každého vrchola odísť po najkratšej hrane.

- Nech O je otec a S jeho syn taký, že odpovedá zaradeniu hrany (i,j) do cesty OC. Zaradenie hrany (i,j) do cesty OC znamená, že

  - nesmieme viac použiť hranu odchádzajúcu z i. Preto každý prvok v i-tom riadku nastavíme na ∞
  - nesmieme viac použiť hranu prichádzajúcu do j. Preto každý prvok v stĺpci j nastavíme na ∞
  - ak ešte nechceme ísť do 1, nastavíme $A(j,1) = \infty$

  Takto vznikla nová matica cien $A_{i,j}$, ktorá nemusí byť redukovaná. Zredukujme ju, čím získame maticu $red(A_{ij})$, ktorú priradíme vrcholu S. Nech $r(i,j)$ je cena redukcie matice A_{ij} na $red(A_{ij})$ (redukujeme riadky a stĺpce okrem tých, ktoré obsahujú samé ∞).
  Potom $\widehat{c}(S) = \widehat{c}(O) + A(i,j) + r(i,j)$

Postupujeme teda tak, že vypočítame redukovanú maticu a cenu pre každého syna (branch) a pohneme sa do toho syna, ktorý má minimálnu cenu (LC).

Poznámka: Namiesto písania hodnôt ∞ môžme odpovedajúci riadok a stĺpec z matice odstrániť. V tomto prípade bude matica odpovedať už len podgrafu indukovanému doteraz nezaradených vrcholov.
Analogicky môžme zvážiť rozhodnutie o nezaradení hrany (i,j) do cesty OC.

- ak X je otec a P jeho syn taký, že odpovedá nezaradeniu hrany (i,j) do cesty OC, tak maticu A musíme modifikovať nasledovne

  - keďže hranu (i,j) viac nemáme použiť, nastavíme $A(i,j) = \infty$ redukcia.

Takto vznikla nová matica B, ktorú musíme zredukovať. Uvedomme si, že budeme redukovať len v tom prípade, ak pôvodná hodnota $A(i,j) = 0$. Redukovanú maticu B' priradíme vrcholu P a cenou bude $\widehat{c}(P) = \widehat{c}(X) + r(i,j)$, kde $r(i,j)$ je cena redukcie potrebná na získanie redukovanej matice B'.

Teraz už môžme uvažovať aj iné stratégie

- vyber hranu, ktorej nezaradenie vedie k maximálnej cene

- vyber hranu, pri ktorej je maximálny rozdiel medzi cenou zaradenia $\widehat{c}(L)$ a cenou $\widehat{c}(P)$ nezaradenia danej hrany do cesty OC

3.4 0/1 Plnenie batoha

Vysvetlili sme, ako metódu LC-B&B používame na riešenie minimalizačných problémov. Je možné ju použiť aj na riešenie problém 0/1 plnenia batoha, ktorý je maximalizačný? Ľahko vidno, že áno. Maximalizovať $\sum_{i=1}^{n} p_i x_i$ je ekvivalentné minimalizovaniu $-\sum_{i=1}^{n} p_i x_i$. Budeme vysvetľovať maximalizáciu, transformáciu na minimalizáciu zvládne každý sám.

Použime statickú reprezentáciu stavového priestoru (zaradenie x_i odpovedá 1-hrane, nezaradenie x_i 0-hrane). Zrejme

$$\mathbf{c(x)} = \begin{cases} \sum_{i=1}^{n} x_i p_i, & \text{ak list x reprezentuje prípustné riešenie} \\ \infty, & \text{ak list x neodpovedá prípustnému riešeniu} \\ \max\{c(0syn(x)), c(1syn(x))\}, & \text{ak x je vnútorný vrchol} \end{cases}$$

Prirodzene sa núkajú dva odhady - $\widehat{c}(x)$ a $u(x)$:

$$u(x) \le c(x) \le \widehat{c}(x)$$

Ako tieto odhady získame? Ak x je vrchol na úrovni j, tak cesta z koreňa doň určuje nastavenie pre $x_i, 1 \le i < j$. Zrejme

$$c(x) \ge \sum_{i=1}^{j-1} x_i p_i$$

a preto ako dolný odhad určite môžme použiť $u(x) = \sum_{i=1}^{j-1} x_i p_i$. Vylepšený dolný odhad získame, ak uvažujeme greedy doplnenie s koeficientami 0,1 (algoritmus 9); pre $q = \sum_{1 \le i < j} p_i x_i$ použijeme

$$u(x) = UBOUND(q, \sum_{1 \le i < j} w_i x_i, j-1, M)$$

Algoritmus 9 UBOUND(p,w,k,M)

$P \leftarrow p; W \leftarrow w$
for $i \leftarrow k+1$ to n **do**
  if $W + w_i \le M$ **then** $W \leftarrow W + w_i; P \leftarrow P - p_i$
return(P)

Horný odhad $\widehat{c}(x)$ v príslušnej vetve je zas daný greedy doplnením s racionálnymi koeficientami za predpokladu, že $\frac{p_i}{w_i} \ge \frac{p_{i+1}}{w_{i+1}}$ (Algoritmus 10). Ľahko vidno, že

$$BOUND(q, \sum_{1 \le i < j} w_i x_i, j-1, M) \le c(x)$$

preto ako horný odhad použijeme

$$\widehat{c}(x) = BOUND(q, \sum_{1 \le i < j} w_i x_i, j-1, M)$$

Ak sme už získali nejaké riešenie ceny c^*, môžme orezať každú vetvu, v ktorej $\widehat{c} < c^*$

Algoritmus 10 BOUND(p,w,k,M)

p aktuálny profit
w aktuálny súčet váh
k index posledne odstráneného prvku
M veľkosť batoha

výsledkom je nový profit

$P \leftarrow p; W \leftarrow w$
for $i \leftarrow k$ to n **do**
  $W \leftarrow W + w_i$
  if $W < M$ **then** $P \leftarrow P + p_i$
  else return$\left(P + \left(1 - \frac{(W-M)}{w_i}\right) \times p_i\right)$
return(P)

Kapitola 4

Rozhodnuteľnosť

4.1 Univerzálny TS

Zamyslime sa nad existenciou Turingovho stroja, ktorý by bol schopný simulovať výpočet ľubovoľného iného Turingovho stroja. Nazvime ho *univerzálny Turingov stroj* a označme UTS. Čo rozumieme simuláciou ľubovoľného Turingovho stroja? Je to schopnosť UTS simulovať výpočet Turingovho stroja T na vstupe w *ak pozná* kód T a w. Inými slovami – UTS *dostane na vstupe* kód nejakého TS T a vstupné slovo w a výstupom UTS bude presne výstup T na vstupe w. Konštrukciou UTS ukážeme, že takýto stroj existuje. Je zrejmé, že jeho existencia úzko súvisí s možnosťou zakódovať program/prechodovú reláciu Turingovho stroja takým spôsobom, aby mu výpočtový model Turingov stroj rozumel.

Dôkaz existencie UTS je konštruktívny - pozostáva z konštrukcie požadovaného UTS. Skôr, ako napíšeme samotný popis UTS, treba sa rozhodnúť pre *rozumné* kódovanie Turingových strojov[1].

kódovanie TS

Keďže

- pásková abeceda UTS je fixovaná a pásková abeceda TS T je konečná, ale ľubovoľne veľká - jej veľkosť pri konštrukcii UTS nepoznáme
- počet stavov UTS je fixovaný, ale počet stavov TS T je síce konečný, ale dopredu neznámej ľubovoľnej veľkosti

musíme na kódovanie používať rozumnú fixovanú abecedu, ktorou môžeme kódovať akúkoľvek konečnú ľubovoľne veľkú abecedu. Pritom treba dať pozor, aby zvolené kódovanie bolo jednoznačné, pretože dekódovanie potrebujeme jednoznačné.

Zvoľme

- $\{0, 1, B\}$ **na kódovanie páskových symbolov** Bez ujmy na všeobecnosti môžeme predpokladať, že abeceda páskových symbolov každého TS pozostáva, okrem špeciálneho symbolu B, ktorý označuje tzv. "blank" =prázdny symbol, len zo symbolov $0, 1$. Nech by nejaká konečná abeceda Σ obsahovala k rôznych symbolov: $\Sigma = \{a_1, \ldots, a_k\}$. Potom každý zo symbolov a_i môžme zakódovať/identifikovať indexom, teda reťazcom núl a jednotiek dĺžky $m = \log k + 1$. Prvky štvorprvkovej množiny a_1, a_2, a_3, a_a, resp. a, b, c, d môžme kódovať reťazcami dĺžky 2:

$$a_1 = a \rightsquigarrow 00 \quad a_2 = b \rightsquigarrow 01 \quad a_3 = c \rightsquigarrow 10 \quad a_4 = d \rightsquigarrow 11$$

[1] UTS má dostať ako vstup kód TS, treba ho rozumným spôsobom zapísať

- 1^s **na kódovanie stavov** Nech množina stavov má k stavov, bez ujmy na všeobecnosti nech $K = \{q_1, \ldots, q_k\}$. Potom reťazcom 1^s kódujeme/označujeme stav q_s.

Každý TS T možno jednoznačne určiť/definovať tabuľkou rozmeru *počet stavov* $\times$ *počet páskových symbolov*, kde obsah príslušného políčka $(stav, symbol)$ určuje hodnotu $\delta(stav, symbol)$. Aby sme tabuľku (ako vstup) uložili do jednorozmernej pásky, budeme ju zapisovať na pásku po riadkoch - blokoch. Pritom jeden riadok popisuje činnosť TS, ktorý v odpovedajúcom stave číta príslušný symbol z pásky. Predpokladáme, že poradie symbolov páskovej abecedy je fixované - napr. 0,1,B.

kódTS T — $\sharp\sharp\sharp$ tabuľka $\sharp\sharp\sharp$

celá tabuľka — $\sharp\sharp\sharp$ riadok $\sharp\sharp \ldots \sharp\sharp$ riadok $\sharp\sharp\sharp$

riadok tabuľky — $\sharp\sharp 1^j \sharp 1^{S(0)} P_0 Z_0 \sharp 1^{S(1)} P_1 Z_1 \sharp 1^{S(B)} P_B Z_B \sharp\sharp$, pričom
1^j znázorňuje, že nasleduje riadok pre spracovanie j-teho stavu,
nasledujú podbloky pre spracovanie symbola $0, 1, B$ (v poradí)

jedno políčko j-teho riadku tabuľky (uložené vo viacerých políčkach vstupnej pásky)
$\sharp 1^{S(a)}, P_a, Z_a \sharp$, kde
$\delta(j, a) = (S(a), P_a, Z_a)$,
$a \in \{0, 1, B\}$ je symbol čítaný pod hlavou
$S(a)$ je stav, do ktorého má TS T po spracovaní (j, a) prejsť
$P_a \in \{1, 0, -1\}$ je posun hlavy pri spracovaní (j, a)
$Z_a \in \{0, 1\}$ je symbol, ktorý TS T zapíše po spracovaní (j, a) na snímané políčko

ak prechod na daný znak nie je definovaný, bude príslušné políčko tabuľky 0.

simulácia zakódovaného TS

UTS dostane na vstupe kód Turingovho stroja T a vstupné slowo w v tvare

$$\sharp\sharp\sharp \underbrace{\text{tabuľka}}_{\text{kód TS T}} \sharp\sharp\sharp \quad \underbrace{w}_{\text{vstup do T}}$$

Obsah pásky tak pozostáva z dvoch častí - časť kódu a časť stroja T. Na začiatku pásky je kód, ktorý si potrebujeme počas celého výpočtu uchovať, druhú časť tvorí obsah pásky zakódovaného stroja T v priebehu výpočtu na vstupnom slove w. V oboch častiach si značkami budeme uchovávať informáciu, ktorá nám simuláciu uľahčí; predpokladáme preto, že páska je v priebehu výpočtu dvojstopá.

Stav stroja T si pamätáme tak, že máme v časti kódu značku stavu S umiestnenú v riadku, ktorý popisuje príslušný stav. Polohu hlavy na páske stroja T si pamätáme umiestnením značky hlavy H v spodnej stope strojom T snímaného políčka.

	#	#	1	1	#			∘ ∘ ∘	#	#	#	1	0	1	B	B
					S								H			

simulovaný TS je v stave q_2 a sníma symbol 0

prípravná fáza

- vytvor na vstupnej páske druhú stopu, pričom prepíš vstup do hornej stopy
- daj značku stavu pod $\sharp$ nasledujúci za 1^1
  (T začína svoj výpočet v počiatočnom stave 1)
- daj značku hlavy pod prvý symbol vstupného slova w. Zapamätaj si v riadiacej jednotke symbol a, ktorý sa nachádza pod hlavou
  (výpočet stroja T začína s hlavou umiestnenou na prvom políčku)

- nastav hlavu UTS na najľavejší symbol $\sharp$ a hodnotu *pokračuj* na true

simulačná fáza

Keďže stroj UTS si v stave pamätá strojom T snímaný symbol a má označený stav, v ktorom sa stroj T "momentálne" nachádza, môže z kódu T vyčítať a následne aplikovať príslušný krok T.

while pokračuj **do**
  nájdi v tabuľke značku stavu
  nájdi podblok $\sharp 1^{S(a)} P_a Z_a \sharp$ odpovedajúci spracovaniu symbola a pod hlavou [2]
  if nie je definovaný(rovná sa $\sharp 0 \sharp$) **then**
    pokračuj $\leftarrow$ false
  else
    zapamätaj si v riadiacej jednotke $S(a), P_a, Z_a$
    prepíš symbol pod značkou hlavy na Z_a, zapamätaj si Z_a v riadiacej jednotke ako nový symbol pod hlavou
    posuň značku hlavy o P_a
    presuň hlavu stavu pod $\sharp$ nasledujúci za $1^{S(a)}$

□

Poznámka: Skonštruovaný univerzálny TS má fixovaný počet stavov a abecedu, ktorá je 4 (počet symbolov na hornej stope) x 3 (počet symbolov na spodnej stope) prvková. Opäť nie je problém prerobiť tento stroj tak, aby jeho abeceda bola len trojprvková $\Sigma = \{0, 1, B\}$. Uvedomme si, že samotný kód používa len symboly 0,1 - je nad dvojprvkovou abecedou $\Sigma_{bin} = \{0, 1\}$. Preto aj UTS má svoj kód, ktorý môže byť vstupom do UTS . Navyše, existuje usporiadanie na Σ_{bin}^* a má zmysel hovoriť o i-tom reťazci x_i, a teda aj i-tom TS, resp. i-tom kóde TS, či kóde i-teho TS.

$$\Sigma_{bin}^* = 0, 1, 00, 01, 10, 11, 000, \ldots$$

$$x_7 = \mathbf{?} \qquad 000101011 = x_{\mathbf{?}}$$

Zrejme nie každý reťazec zo Σ^* je zmysluplným kódom TS tak, ako sme ho popísali. Zistiť to však nie je problém. Prípadne sa môžme dohodnúť, že reťazec, ktorý nie je správnym kódom TS, budeme považovať za kód takého TS, ktorý rozpoznáva prázdny jazyk; potom je i-ty reťazec kódom i-teho TS.

Úloha: Napíšte program v ľubovoľnom programovacom jazyku, ktorý overí, či vstupný reťazec je syntakticky korektným kódom nejakého TS. Skúste ten istý problém riešiť na TS.

Úloha: Napíšte program (v ľubovoľnom programovacom jazyku), ktorý bude simulovať UTS. Zo vstupu načíta reťazec $\#\#\#kod\#\#\#w$ a na výstup vypíše výstup zakódovaného TS.

Úloha: Zamyslite sa nad usporiadaním PASCALovských programov.

4.2 Rozhodnuteľné a nerozhodnuteľné problémy

Doteraz ste sa zrejme venovali takým problémom, ktoré sa dali riešiť. Dá sa ukázať, že existujú problémy, ktoré sú neriešiteľné. Dôkaz je jednoduchý:

$\Rightarrow$ Ak existuje riešenie, je napísané v nejakom jazyku; nezáleží na tom, či je to program, matematická formula alebo text. Vo všetkých prípadoch sa jedná o *reťazec*

[2] pamätáme si ho v riadiacej jednotke

nad konečnou abecedou. Keďže konečnú abecedu môžme kódovať binárnymi reťazcami fixnej dĺžky, môžme každému reťazcu *text* nad konečnou abecedou priradiť binárny reťazec τ. No a teraz už nie je problém vnímať τ ako binárny zápis čísla t.

$$text \leadsto \tau \leadsto t$$

Z toho dostávame, že *algoritmov/riešní/dôkazov/.. je toľko, ako prirodzených čísel.*

$\Rightarrow$ Na druhej strane problémov je aspoň toľko, ako reálnych čísel. Všimnime si len jednu konkrétnu množinu problémov, a to rozpoznávanie formálnych jazykov. Formálnym jazykom môžme rozumieť ľubovoľnú množinu reťazcov L nad konečnou abecedou Σ; $L \subseteq \Sigma^*$. Nech a^n označuje reťazec pozostávajúci z n symbolov a. Potom

$$L = \{aab, aaabb, \ldots a^n b^{n-1} \mid n \in \mathbb{N}\}$$

je jazykom nad abecedou $\Sigma = \{a, b\}$. My sa obmedzíme na jazyky nad abecedou $\{0, 1\}$. Ukážeme, že jazykov je toľko, ako reálnych čísel v intervale $< 0, 1 >$.
Každému binárnemu reťazcu τ vieme priradiť jeho poradové číslo i_τ v postupnosti Σ_{bin}. Potom reálnemu číslu $\alpha = 0.\alpha_1\alpha_2\ldots,\ 0 < \alpha < 1$ priradíme najprv číslo $\beta, 0 < \beta < 1$ tak, že α_i nahradíme binárnym zápisom $bin(\alpha_i)$ čísla α_i.

$$0.23 \leadsto 0.\underbrace{0010}_{2}\underbrace{0011}_{3} \qquad 0.451 \leadsto 0.\underbrace{0100}_{4}\underbrace{0101}_{5}\underbrace{0001}_{1}$$

$\beta = 0.\beta_1\beta_2\ldots$ Potom

$$\begin{array}{lll} \beta & \leadsto & L_\beta = \{\tau \in \{0,1\}^*, \beta_{i_\tau} = 1\} \\ L \subseteq \{0,1\}^* & \leadsto & \beta = 0.\beta_1\beta_2\ldots..., \quad \beta_i = 1 \Leftrightarrow \tau_i \in L \end{array}$$

$\Rightarrow$ Keďže prirodzených čísel je menej ako reálnych, existujú problémy, pre ktoré neexistuje riešenie.

Na programovaní ste písali programy na riešenie rôznorodých problémov, na M ste skúmali uzavretosť, resp. neuzavretosť triedy objektov na niektoré operácie,... Je tu však množstvo iných otázok, napr:

- je napísaný program korektný?
- je výpočet daného programu na konkrétnom vstupe konečný?
- ...

Toto sú otázky, na ktoré sa ťažko odpovedá. Chceli by sme nájsť algoritmy, ktoré nám odpoveď vypočítajú. Pre niektoré z uvedených problémov sa však hľadaný algoritmus nedarí nájsť. Existuje vôbec? Dostávame sa k problémom rozhodnuteľnosti.

rozhodovací problém

Definícia 4.1 *Problém, ktorý zahrňuje nekonečný počet prípadov a v každom prípade je odpoveď buď áno alebo nie, je rozhodovací problém.*

rozhodnuteľný problém

Definícia 4.2 *Rozhodovací problém je rozhodnuteľný, ak existuje algoritmus, ktorý pre daný vhodný kód ľubovoľného prípadu dá správnu odpoveď pre tento prípad. Ak algoritmus neexistuje, je problém nerozhodnuteľný.*

Pri kódovaní treba dávať pozor, aby nami predpokladané kódovanie existovalo; aby sme ho efektívne vedeli vyrábať a používať.

Ak chceme dokázať, že problém je rozhodnuteľný, napíšeme algoritmus, ktorý ho rieši. Ako ale dokázať, že problém nie je rozhodnuteľný, ale je nerozhodnuteľný? Musíme vyargumentovať, že neexistuje TS na jeho riešenie. Inými slovami, neexistuje TS, ktorý na každom vstupe zastane a správne odpovie; každý potenciálny kandidát sa na nejakom vstupe musí "pomýliť".

Ako prvý nerozhodnuteľný problém uvažujme jazyk $L_{DIAG} \in \Sigma^*_{bin}$

$$L_{DIAG} = \{x_i \mid T_i \text{ neakceptuje } x_i\}$$

Pripomeňme si, že každý reťazec núl a jednotiek možno chápať ako vstupné slovo, ale tiež ako kód TS. A tiež to, že keď zafixujeme postupnosť reťazcov nad abecedou $\{0, 1\}$, tak má zmysel hovoriť o i-tom vstupe x_i, resp. i-tom TS T_i, ktorý zodpovedá i-temu slovu v tejto postupnosti (chápanom ako jeho kód).

Veta 4.1 *Jazyk L_{DIAG} je nerozhodnuteľný.*

nerozhodnuteľnosť L_{DIAG}

Dôkaz: Nerozhodnuteľnosť jazyka L_{DIAG} ukážeme sporom. Predpokladajme, že existuje TS T, ktorý jazyk L_{DIAG} rozpoznáva. Nech tento TS je i-ty v kódovaní TS, teda vlastne T_i. Zoberme slovo x_i a dajme ho na vstup TS $T = T_i$. Aká je situácia?

$$\left.\begin{array}{lll} x_i \in L(T_i) & \Rightarrow & x_i \notin L_{DIAG} = L(T_i) \\ x_i \notin L(T_i) & \Rightarrow & x_i \in L_{DIAG} = L(T_i) \end{array}\right\} x_i \in L(T_i) \Leftrightarrow x_i \notin L(T_i)$$

V oboch prípadoch máme spor, preto bol predpoklad o existencii TS, ktorý rozpoznáva L_{DIAG} nesprávny.

□

Použitému jazyku sa niekedy hovorí diagonalizačný a metóde dôkazu *diagonalizácia*. Je tomu tak (aj) preto, že keď spravíme tabuľku, ktorej jedným indexom je slovo, druhým indexom stroj a hodnota

$$\text{tab(Index-slova, Index-stroja)} = 1 \Leftrightarrow \text{stroj akceptuje slovo}$$

tak (diagonalizačný) jazyk L_{DIAG} je vznikne tak, že hodnoty na diagonále zmeníme.

$$x_i \in L(T_i) \Leftrightarrow tab(i, i) = 0$$

Úloha: Metódu diagonalizácie môžme použiť napríklad aj pri argumentácii o existencii funkcie, ktorá sa nedá vypočítať žiadnym PSCALovským programom. Skúste.

metóda redukcie

V okamihu, keď už máme nejaký nerozhodnuteľný problém, môžeme ho využiť pri dôkaze nerozhodnuteľnosti iného problému. Ide vlastne o dôkaz sporom, ktorý je zachytený v nasledujúcej schéme.

Nech
N je známy nerozhodnuteľný problém
K problém - kandidát na to, aby bol nerozhodnuteľný

Postupujeme sporom:

1. Predpokladajme, že problém K je rozhodnuteľný a $\mathcal{A}$ je algoritmus, ktorý ho rozhoduje.
2. Skonštuujeme algoritmus $\mathcal{B}$, ktorý za predpokladu existencie $\mathcal{A}$ rieši N.

3. Keďže N je nerozhodnuteľný, dostaneme spor s predpokladom existencie $\mathcal{A}$. *Preto je K nerozhodnuteľný.*

Uvedená schéma je vlastne **metódou redukcie**, keď sa neriešiteľnosť jedného problému získa redukciou na iný problém. Formálne:

rekurzívna redukovateľnosť

Definícia 4.3 *Nech $L_1 \subseteq \Sigma_1^*, L_2 \subseteq \Sigma_2^*$ sú jazyky. Hovoríme, že jazyk L_1 je* rekurzívne redukovateľný *na jazyk L_2, $\mathbf{L_1 \leq_R L_2}$, ak*

$$L_2 \in \mathcal{L}_R \Longrightarrow L_1 \in \mathcal{L}_R$$

Neformálne $L_1 \leq_R L_2$ vyjadruje fakt, že jazyk L_2 je z hľadiska algoritmickej riešiteľnosti aspoň tak ťažký, ako jazyk L_1.

Pre aplikovanie spomínanej schémy potrebujeme nájsť vhodný nerozhodnuteľný problém N, ktorého riešenie by nám pomohlo vyriešiť náš problém K. Ako však využiť algoritmus jedného problému pre riešenie iného problému? Jedným zo spôsobov je redukcia/transformácia jedného problému na druhý. Formálnejšie:

many-one redukovateľnosť

Definícia 4.4 *Nech $L_1 \subseteq \Sigma_1^*, L_2 \subseteq \Sigma_2^*$ sú jazyky. Hovoríme, že jazyk L_1 je* (many-one) redukovateľný *na jazyk L_2, $\mathbf{L_1 \leq_m L_2}$, ak existuje TS M, ktorý počíta zobrazenie*

$$f_M : \Sigma_1^* \to \Sigma_2^*$$

také, že

$$\forall x \in \Sigma_1^* \qquad x \in L_1 \Leftrightarrow f_M(x) \in L_2$$

Funkcii f_m zvykneme hovoriť redukcia L_1 na L_2.

O vzťahu spomenutých redukcií hovorí nasledujúca lema, dôkaz ktorej necháme ako cvičenie.

Lema 4.2 *Nech $L_1 \subseteq \Sigma_1^*, L_2 \subseteq \Sigma_2^*$ sú jazyky.*

$$\text{ak } L_1 \leq_R L_2 \quad \text{tak } L_1 \leq_m L_2$$

A teraz sa vráťme k dôkazom nerozhodnuteľnosti.

4.2.1 Zastavenie TS

problém zastavenia

Definícia 4.5 Problém zastavenia*: pre daný kód TS T a konfiguráciu C určiť, či T zastane pri výpočte štartujúcom z konfigurácie C.*

Ukážeme, že tento problém je nerozhodnuteľný. Napriek tomu, že je tento problém formulovaný ako problém zastavenia TS, zrejme nie je ťažké si predstaviť, že k danému programu v nejakom programovacom jazyku by sme napísali TS, ktorý by simuloval jeho činnosť. Potom sa problém zastavenia TS stáva problémom toho, či daný program vôbec niekedy skončí. Neznamená to nutne, že z toho priamo vyplýva nerozhodnuteľnosť problému zastavenia programu, ale určite si uvedomíme, že je to ťažký problém. A to je problém zastavenia zrejme ľahší, ako požadovať odpoveď na otázku, či je daný program korektný...

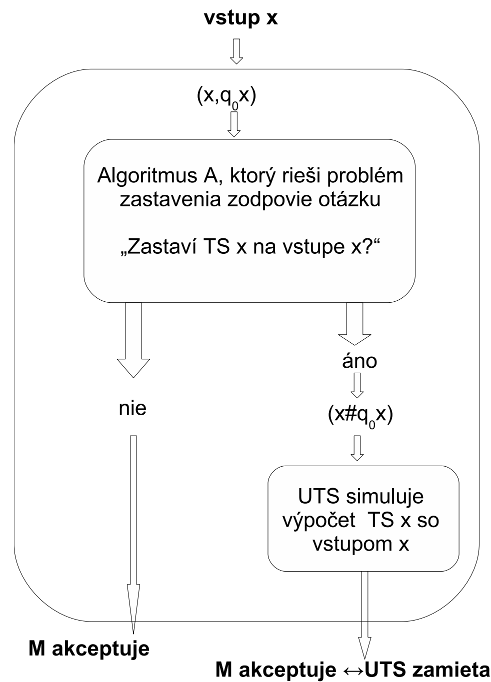
nerozhodnuteľnosť zastavenia

Veta 4.3 *Problém zastavenia TS je nerozhodnuteľný.*

Dôkaz: Pri dôkaze využijeme nerozhodnuteľnosť jazyka L_{DIAG},

$$L_{DIAG} = \{x_i \mid T_i \text{ neakceptuje } x_i\}$$

ktorý použijeme podľa vysvetlenej schémy:



Obrázok 4.1: Rozpoznávanie L_{DIAG} pomocou algoritmu, ktorý rieši problém zastavenia

1. Predpokladajme, že je problém zastavenia TS rozhodnuteľný, $\mathcal{A}$ je TS, ktorý ho rozhoduje.
2. Uvažujme nasledovný algoritmus/TS M na rozpoznávanie L_{DIAG}:
   - M vytvorí počiatočnú konfiguráciu $C = q_0x$ a chápe x ako kód TS. Vytvorí vstup pre problém zastavenia

$$(\underbrace{x}_{\text{kód TS}}, \underbrace{q_0x}_{\text{počiatočná konfigurácia}})$$

   - M odovzdá riadenie TS $\mathcal{A}$, ktorý rozhodne, či TS x so vstupom x zastane alebo nie
   - ak TS x zastane, tak M odovzdá riadenie UTS, ktorý odsimuluje výpočet TS x na vstupe x a odpovie naopak
   - ak TS x na vstupe x nezastane, tak M akceptuje.

Ak vieme, že jazyk L_{DIAG} sa nedá rozpoznávať, bol predpoklad o existencii TS $\mathcal{A}$ rozhodujúceho problém zastavenia nesprávny. □

4.2.2 Postov korešpondenčný problém

Definícia 4.6 *Nech Σ je konečná abeceda, A a B sú dva zoznamy slov zo Σ^*, pričom oba majú* rovnaký *počet slov.* *PKP*

A $= w_1, w_2, \ldots, w_n$

B $= x_1, x_2, \ldots, x_n$

Hovoríme, že Postov korešpodenčný problém *(PKP, presnejšie inštancia/prípad PKP)* má riešenie*, ak existuje postupnosť indexov* $i_1, \ldots, i_m$ *taká, že*

$$w_{i_1} w_{i_2} \ldots w_{i_m} = x_{i_1} x_{i_2} \ldots x_{i_m}$$

V takom prípade je postupnosť $i_1, \ldots, i_m$ riešením *tohto* prípadu *PKP.*

Ak vyžadujeme, aby $i_1 = 1$*, ide o* modifikovaný Postov korešpondenčný problém.

Nech (A,B) je vstup do PKP. Fakt, že tento prípad má riešenie zvykneme označovať $(A, B) \in PKP$*. Analogicky pre MPKP.*

Nasledujú dva príklady konkrétnych vstupov do PKP. Prvý je príkladom PKP, ktorý riešenie má.

Príklad 4.1

i	w_i	x_i
1	1	111
2	10111	10
3	10	0

Riešenie

2,	1,	1,	3
10111	1	1	10
10	111	111	0

A teraz príklad prípadu PKP, ktorý riešenie nemá.

Príklad 4.2

i	w_i	x_i
1	10	101
2	011	11
3	101	011

Riešenie

Aby x_{i_1}, w_{i_1} *mohli byť začiatkom riešenia, musí byť jeden druhému prefixom. Preto do úvahy pripadá len* $i_1 = 1$*. Takto dostaneme*
A: 10
B: 101

Ak chceme správne pokračovať, musí byť $i_2 = 3$*. Avšak potom*
A: 10101
B: 101011

Opäť je zoznam A kratší a jediné možné pokračovanie túto situáciu zachováva. Preto tento prípad PKP riešenie nemá.

Uvedené dve verzie Postovho korešpondenčného problému sa z hľadiska definície líšia len minimálne. Čo vieme povedať o riešení týchto problémov? Líšia sa z hľadiska zložitosti? Uvidíme, že nie. Najprv ukážeme, že PKP je rozhodnuteľný práve vtedy ak je rozhodnuteľný MPKP. Následne potom redukciou na problém zastavenia ukážeme, že MPKP(a teda aj PKP!) je nerozhodnuteľný.

$MPKP \to PKP$

Fakt 4.4 *Ak by bol rozhodnuteľný MPKP, tak by bol rozhodnuteľný aj PKP.*

Dôkaz: Majme inštanciu PKP danú zoznamami A, B
$A = w_1, w_2, \ldots, w_n$
$B = x_1, x_2, \ldots, x_n$

Predpokladajme, že existuje algoritmus $\mathcal{A}$ na riešenie MPKP. Ľahko vidno, že PKP má riešenie práve vtedy, ak ho má niektoré z n prípadov *modifikovaného* MPKP, ktoré vzniknú shiftami zoznamov A,B. Nech sú to PKP_1, $\text{PKP}_2, \ldots, \text{PKP}_n$, kde $\text{PKP}_i = (A_i, B_i)$,
$A_i = w_i, \ldots w_n, w_1, \ldots, w_{i-1}$
$B_i = x_i, \ldots x_n, x_1 \ldots, x_{i-1}$
Rozhodnuteľnosť PKP sme takto zredukovali na nanajvýš n volaní algoritmu $\mathcal{A}$.
□

Veta 4.5 *Ak by bol rozhodnuteľný PKP, tak by bol rozhodnuteľný aj MPKP.*

$PKP \to MPKP$

Dôkaz: Aby sme pri riešení MPKP mohli využiť algoritmus, ktorý rieši PKP, budeme postupovať takto:
Nech (A,B) je prípad MPKP
$A = w_1, w_2, \ldots, w_n$
$B = x_1, x_2, \ldots, x_n$.
Zostrojíme (k nemu) taký prípad (A', B') PKP, pre ktorý bude platiť

$$(A, B) \in MPKP \Leftrightarrow (A', B') \in PKP$$

Konštrukcia PKP (A', B')

Nech Σ je najmenšia abeceda obsahujúca všetky symboly zo zoznamov A,B a nech $\$, \# \notin \Sigma$. Uvažujme nasledujúce homomorfizmy:
$h_L(a) = \$a$
$h_R(a) = a\$$

Potom hľadané zoznamy sú:

	w'_i	x'_i
1	$\$h_R(w_1)$	$h_L(x_1)$
i+1	$h_R(w_i)$	$h_L(x_i)$
n+2	§	$§

PKP má riešenie ⇔ MPKP má riešenie

Nech $1, i_1, i_2, \ldots, i_m$ je riešenie MPKP (A,B). Potom $1, i_1+1, i_2+1, \ldots, i_m+1, n+2$ je riešenie PKP (A', B').

$MPKP \to PKP$

Nech $i_1, i_2, \ldots, i_r$ je riešenie PKP (A', B'). Potom $i_1 = 1$ a $i_r = n + 2$. Nech j je najmenší taký index, že $i_j = n + 2$. Potom zrejme $i_1, i_2, \ldots, i_j$ je tiež riešenie $\text{PKP}(A', B')$. Riešenie MPKP(A,B) potom je $1, i_2 - 1, i_3 - 1, \ldots, i_{j-1} - 1$. □

$PKP \to MPKP$

Veta 4.6 *Modifikovaný Postov korešpondenčný problém je nerozhodnuteľný.*

nerozhodnuteľnosť MPKP

Dôkaz: Pri dôkaze využijeme nerozhodnuteľnosť problému zastavenia — ak by bol rozhodnuteľný MPKP, tak by bol rozhodnuteľný aj problém zastavenia TS.
K danému TS T a vstupnému slovu w – teda vstupu do problému zastavenia – skonštruujeme prípad MPKP. Pritom skonštruovaný prípad MPKP bude mať tú

vlastnosť, že má riešenie práve vtedy, ak sa TS T vo výpočte z počiatočnej konfigurácie so vstupom w zastaví.

Zoznamy MPKP preto budeme konštruovať tak, aby sme mali možnosť "simulovať" výpočet TS T na slove w.

- Nech TS $T = (K, \Sigma, \Gamma, \delta, q_0, F)$. Predpokladáme, že $\forall q \in F\ \forall a \in \Sigma$ je $\delta(q, a)$ nedefinované; po dosiahnutí akceptujúceho stavu sa TS zastaví.
- Konfiguráciu (q, α, i) budeme reprezentovať reťazcom $\alpha_1 q \alpha_2$, kde $|\alpha_1| = i - 1$ô stav vkladáme pred symbol snímaný hlavou TS.
- Ak $q_0 w, \alpha_1 q_1 \beta_1, \alpha_2 q_2 \beta_2, \ldots, \alpha_k q_k \beta_k$ bude možným výpočtom, $q_k \in F$, tak začiatok každého riešenia prípadu MPKP bude

$$\sharp q_0 w \sharp \alpha_1 q_1 \beta_1 \sharp \alpha_2 q_2 \beta_2 \sharp \ldots \sharp \alpha_k q_k \beta_k \sharp ; \sharp \notin K \cup \Gamma$$

Na zozname B simulujeme výpočet TS T, zoznamom A sme o jeden krok pozadu. Takto na základe konfigurácie C_i v zozname A máme možnosť budovať konfiguráciu C_{i+1} v zozname B.

čiastočné riešenie

Označenie: Hovoríme, že (x, y) je *čiastočné riešenie* MPKP, ak x je prefixom y a x, y sú potenciálnym začiatkom riešenia prípadu MPKP.

zvyšok

Ak $xz = y$, potom hovoríme, že z je *zvyšok* čiastočného riešenia x, y.

	Zoznam A	Zoznam B	
vytváranie pásky skupina 1	$\sharp$	$\sharp q_0 w \sharp$	
	X	X	$X \in \Gamma - B$
	$\sharp$	$\sharp$	
simulácia T vytváranie úseku v okolí hlavy skupina 2	qX	Yp	δ(q,X)=(p,Y,R)
	ZqX	pZY	δ(q,X)=(p,Y,L)
	q$\sharp$	Yp$\sharp$	δ(q,B)=(p,B,R)
	Zq$\sharp$	pZY$\sharp$	δ(q,B)=(p,Y,L)
A postupne "dobieha"B skupina 3	XqY	q	$q \in F$, X,Y $\in \Gamma - \{B\}$
	Xq$\sharp$	q$\sharp$	
	$\sharp$qy	$\sharp$q	
záver–skupina 4	q$\sharp\sharp$	$\sharp$	$q \in F$

Matematicou indukciou sa ukáže:

Nech $q_0 w, \alpha_1 q_1 \beta_1, \alpha_2 q_2 \beta_2, \ldots, \alpha_k q_k \beta_k$ je platná postupnosť konfigurácií taká, že neobsahuje koncový stav. Potom existuje čiastočné riešenie

$$\text{(x,y)} = (\sharp q_0 w \sharp \alpha_1 q_1 \beta_1 \sharp \alpha_2 q_2 \beta_2 \sharp \ldots \sharp \alpha_{k-1} q_{k-1} \beta_{k-1} \sharp,\ \sharp q_0 w \sharp \alpha_1 q_1 \beta_1 \sharp \alpha_2 q_2 \beta_2 \sharp \ldots \sharp \alpha_k q_k \beta_k \sharp)$$

Od okamihu, keď sa v zozname B objaví koncový stav, vhodným výberom indexov skupiny 1 a 3 a na záver skupiny 4 dotiahneme riešenie do konca. □

Kapitola 5

Výpočtové modely a vzťahy medzi nimi

Dôvodom pre definovanie abstraktného výpočtového modelu je snaha o dokazovanie vlastností, ktoré hovoria o problémoch ako takých. Chceme hovoriť o zložitosti problémopv a tá nemá byť závislá na výbere konkrétneho počítača. Model však musí byť dostatočne silný, aby odpovedal nášmu intuitívnemu vnímaniu toho, čo je a čo nie je vypočitateľné. Zatiaľ sme mali dva modely - TS a (M)RAM.

Pre dokazovanie toho, čo sa na TS *nedá*, je jeho definícia vyhovujúca, pretože je dostatočne jednoduchá, stroj sa dá jednoducho popísať a má malú množinu elementárnych inštrukcií, ktoré môže používať. Ak chceme TS ako výpočtový model "programovať", je to nepohodlné. Zadefinujeme preto viacero jeho modifikácií, ktoré nám programovanie uľahčia. Ukážeme, že naše modifikácie nemajú vplyv na samotnú výpočtovú silu TS; len na zložitosť. Tiež si budeme všímať vzťah TS a (M)RAMu, ktorý je predsalen bližší počítaču...

Ako prvú modifikáciu dovolíme TS používať viac pások.

5.1 k-páskový Turingov stroj

Existuje viacero možností, ako definovať k-páskový TS. V podstate nám ide o to, že namiesto jednej pásky, na ktorej mohol čítať i zapisovať, uvažujeme k pások a každá má "svoju" hlavu. V jednom kroku sa TS "pozrie" na obsah políčok pod hlavami, na aktuálny stav a na základe toho zmení stav, obsah políčok pod hlavami a patrične pohne jednotlivými hlavami.

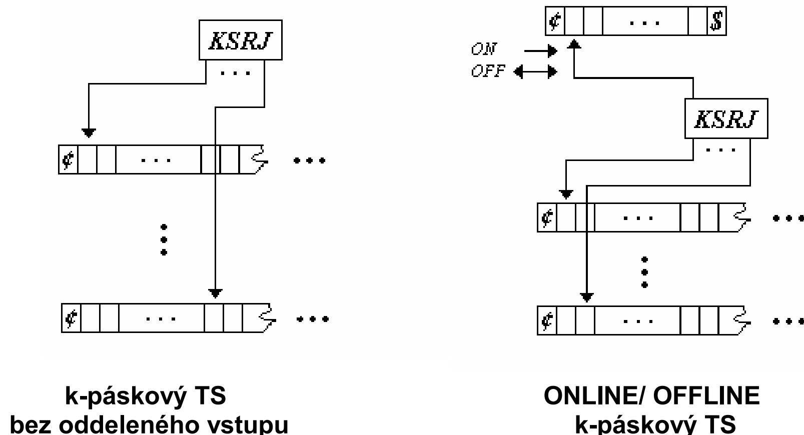
Otázkou je, kde sa nachádza vstup. Presnejšie, či ho možno prepisovať. Máme dva základné prístupy:

(i) vstupná páska sa môže prepisovať a je teda pracovná

(ii) jedna z pások je len vstupom; vstup sa z nej iba číta, nesmie sa prepisovať. V tomto prípade rozpoznávame 2 modely ONLINE (jednosmerný pohyb čítacej hlavy) a OFFLINE (obojsmerný pohyb čítacej hlavy)

Vo všetkých prípadoch budeme predpokladať, že všetky pásky sú jednosmerne nekonečné smerom doprava, s ľavým okrajom označeným špeciálnym znakom ç (v prípade ONLINE/OFFLINE modelu je vstup označený aj z prava značkou \$).

Nasleduje definícia deterministického k-páskového TS, prvá páska obsahuje na začiatku výpočtu vstup.



k-páskový TS bez oddeleného vstupu — **ONLINE/ OFFLINE k-páskový TS**

Definícia 5.1 *Deterministický k-páskový TS je sedmica* $(\Sigma, \Gamma, K, \delta, B, q_0, F)$*, kde* $\Sigma, \Gamma(\Sigma \subseteq \Gamma), K, B, q_0$ *a* F *majú význam ako pri definícii TS,*

$$\delta : (\Gamma^k \times K) \rightarrowtail (\Gamma^k \times K \times \{0, 1, -1\}^k)$$

je prechodová funkcia.

Pojem konfigurácie, kroku odvodenia, výpočtu,.. sa definuje analogicky ako v prípade TS. Pritom platí:

Ak $\delta(c_1, c_2, .., c_k, k) = (c'_1, c'_2, .., c'_k, p, pos_1, .., pos_k)$
tak $\forall\ i$ také, že $c_i =$ç je $pos_i \in \{0, 1\} \wedge c'_i = c_i$. (Pripúšťame prepisovanie vstupu.)

Formálnu definíciu ONLINE,OFFLINE k-páskového TS prenechávam čitateľovi (vstupná páska sa nesmie prepisovať, pozor na pohyb hlavy na vstupe).
Základné miery zložitosti definované pre TS sú časová a pamäťová zložitosť. Definujú sa analogicky ako v prípade zložitosti konkrétnych algoritmov.

Uvedomme si, že ak uvažujeme do pamäťovej zložitosti aj priestor vstupu, je najmešia možná pamäťová zložitosť lineárna. Pritom vieme, že v prípade rozpoznávania regulárnych jazykov je vlastne pamäť konštantná. Zdá sa preto rozumné rozlišovať medzi vstupom a tým, čo naozaj musí byť v pamäti. Na úrovni abstraktného modelu to riešime takým modelom TS, ktorý má jedinú vstupnú pásku a niekoľkých pamäťových pások, ktoré možno prepisovať. Do priestorovej–pamäťovej zložitosti počítame len pamäťové pásky. Pri definovaní zložitosti uvažujeme základný model TS - deterministický viacpáskový TS s k jednosmerne nekonečnými páskami.

čas – počet krokov, resp. dĺžka výpočtu; označujeme T(n)

pamäť – maximále číslo políčka navštívené počas výpočtu na niektorej z pamäťových pások; označujeme S(n), resp. niekedy L(n)

Definícia 5.2 *Hovoríme, že*

čas

TS je T(n) - časovo ohraničený*, ak pre žiadne vstupné slovo dĺžky n nespraví viac ako T(n) krokov; jazyk/problém je* časovej zložitosti T(n)*, ak existuje T(n)-časovo ohraničený TS, ktorý ho rozpoznáva/rieši*

priestor

TS je S(n) -priestorovo (páskovo) ohraničený*, ak pri výpočte na žiadnom vstupnom slove dĺžky n nepoužije viac ako S(n) políčok pásky; jazyk je* priestorovej zložitosti S(n)*, ak existuje S(n) priestorovo ohraničený TS, ktorý ho rozpoznáva*

Príklad 5.1 *Napíšte TS, ktorý rozpoznáva nasledujúce jazyky*

1. $L = \{x1y|\ x, y \in \{a, b\}^*, |x| = |y|\}$
2. $L = \{wcw|\ w \in \{a, b\}^*\}$
3. $L = \{a^n b^n|\ n \geq 0\}$

Aká je zložitosť vami navrhnutých TS?

Veta 5.1 *Ku každému k-páskovému $S(n)$-páskovo ohraničenému TS existuje ekvivalentný jednopáskový $S(n)$-páskovo ohraničený TS . Ku každému k-páskovému $T(n)$-časovo ohraničenému TS existuje ekvivalentný jednopáskový $O(T^2(n))$-časovo ohraničený TS.*

k→1

Dôkaz: : Obsahy k-pások T_k si jednopáskový T_1 uloží do stôp. Budeme mať $2k$ stôp. V každej dvojici stôp jedna stopa uchováva obsah simulovanej pásky, druhá slúži na uchovanie informácie o polohe hlavy na tejto páske.

simulácia

Takže: v stope $2i$ bude uložený obsah i-tej pásky. Stopa $2i-1$ slúži na zapamätanie pozície hlavy na i-tej páske; j-te políčko $2i-1$ -ej stopy obsahuje špeciálny znak $\sharp$ práve vtedy ak i-ta hlava simulovaného T_k je na j-tom políčku.

Simulácia prebieha v taktoch. V jednom takte odsimuluje T_1 jeden krok T_k:

- T_1 prejde hlavou zľava doprava, pričom si do stavu zapamätá symboly $a_1, \ldots, a_k$ pod hlavami stroja T_k. V priebehu tohto posunu je T_1 v stave, ktorý vyzerá nasledovne:

  $[q, x_1, .., x_k]$, pričom

  q je stav simulovaného T_k

  $x_i = a_i$ ak v priebehu tohto posunu bola hlava T_1 na políčku, ktoré v stope $2i$ obsahovalo symbol a_i a v stope $2i-1$ bol symbol pre polohu hlavy

  $x_i = -$ ak sme v priebehu tohto posunu ešte neboli na políčku, ktoré v stope $2i-1$ obsahovalo symbol pre polohu hlavy

  Je zrejmé, že v okamihu, keď dosiahne stav $[q, a_1, .., a_k]$, má T_1 informáciu potrebnú pre "aplikovanie" prechodovej funkcie T_k.

- V stave aplikuje prechodovú funkciu. Nech $\delta_k(q, a_1, \ldots, a_k) = (p, b_1, \ldots, b_k, pos_1, \ldots, pos_k)$; T_1 si v stave uchová $(p, b_1, \ldots, b_k, pos_1, \ldots, pos_k)$

- Prechodom sprava doľava upraví pásku tak, aby odpovedala kroku T_k - prepíše a_i symbolom b_i a posunie "hlavou" j-tej pásky podľa pos_j, nastaví aktuálny stav T_k na p.

Uvedomme si, že "posun hlavy" znamená posun značky označujúcej polohu hlavy a môže znamenať, že sa T_1 posunulo vždy o jedno políčko doprava (ak sa hlava má posunúť doprava).

Teraz sa pozrime na zložitosť tejto simulácie. Nech $T_k(n), T_1(n), S_k(n), S_1(n)$ označujú časovú, resp. priestorovú zložitosť i-páskového TS $T_i, i = 1, k$.

zložitosť simulácie

Reprezentácia viacerých pások v stopách jedinej pásky spôsobí, že priestorová zložitosť stroja je daná stopou, ktorá simuluje pôvodne najdlhšiu pásku.

pamäť

Simulácia jedného taktu stroja T_1 je (lineárne)úmerná veľkosti pásky stroja T_k. Keďže pásková zložitosť je vždy zhora ohraničená časovou zložitosťou, dostávame:

čas

$$T_1(n) = O(T_k(n) \times S_k(n)) = O(T_k^2(n))$$

□

Situácia je trochu iná, ak uvažujeme simuláciu k-páskového TS na dvojpáskovom TS. Možnosť používania pomocnej pásky sa odrazí na zložitosti tejto simulácie.

k→2

Veta 5.2 *Ku každému k-páskovému $T(n)$-časovo ohraničenému TS T_k existuje ekvivalentný dvojpáskový $O(T_k(n) \log T_k(n))$ časovo ohraničený TS T_2.*

Dôkaz: (Náčrt) Jednu z pások T_2 budeme používať ako pamäťovú, druhá bude pomocná. Keďže nárast času pri simulácii bol spôsobený tým, že sme v zložitosti $O(T_k(n))$ zisťovali obsah políčok pod hlavami simulovaného T_k, budeme "šetriť" tým, že po skončení každého taktu bude obsah pásky upravený tak, aby sa všetky simulované hlavy nachádzali na jednom políčku pamäťovej pásky (každá v inej stope). Popíšeme, akým spôsobom sa pomocou pomocnej pásky realizuje prepísanie čítaného symbola a posun na jednej simulovanej páske. Celkový krok T_k potom pozostáva z k po sebe nasledujúcich prepisov a posunov; zmenu na každej z k pások realizujeme zvlášť, jednu po druhej.
Pri simulácii

- predpokladáme, že páska je obojstranne nekonečná
- každá z k pások T_k je na pamäťovej páske T_2 reprezentovaná dvomi stopami. Stopy sú rozdelené (až v okamihu, keď sa na príslušné políčko dostaneme) do blokov $\ldots, B_{-i}, \ldots, B_{-1}, B_0, B_1, \ldots, B_i, \ldots$, pričom $|B_i| = |B_{-i}| = 2^{i-1}, |B_0| = 1$
- predpokladáme, že na začiatku výpočtu sú symboly uložené v spodnej stope

uloženie pásky v 2 stopách

j-ta páska simulovaného T_k je v dvoch stopách T_2 uložená nasledovne:

Pre bloky B_i a $B_{-i}, i > 0$, platí

ak je jeden plný (používa obe stopy), je druhý prázdny

oba používajú len spodnú stopu

Obsahy blokov sú po sebe idúce políčka reprezentovanej pásky, pričom

v B_i sú políčka na hornej stope pred políčkami na spodnej stope

v B_{-i} sú políčka na hornej stope pred políčkami na spodnej stope

B_i reprezentuje políčka naľavo od $B_j, 0 \leq i < j$

B_0 reprezentuje políčko pod hlavou T_k

posun hlavy vľavo

Ak simulovaný T_k chce posunúť hlavou doľava, my musíme "potiahnuť" pásku doprava. potrebujeme do bloku B_0 "dostať" prvý symbol, ktorý sa nachádza naľavo od neho. To súčasne znamená, že blok B_0 musíme uvoľniť. Budeme ho presúvať do časti napravo.
Algoritmus 11 popisuje simuláciu jedného kroku T_k. Podľa hodnoty i budeme jednému vykonaniu tohto algoritmu hovoriť B_i−operácia.

Všimnime si, že

- realizácia B_i operácie je úmerná veľkosti $|B_i|$
- po realizácii B_i−operácie používajú bloky $B_1, \ldots, B_{i-1}$ len spodnú stopu. Preto každú B_i operáciu môžme vykonať nanajvýš raz za 2^{i-1} krokov stroja T_k (musíme najprv zaplniť hornú stopu, ktorú sme pri poslednej B_i operácii vyprázdnili.)
- maximálny index i bloku, do ktorého sa počas simulácie dostane hlava T_2 je zhora ohraničený $i \leq \log T_k(n) + 1$

Algoritmus 11 posun hlavy doľava

1: nech $i > 0$ je najmenšie také, že blok B_i nie je plný;
2: **if** B_i nie je úplne prázdny (používa spodnú stopu) **then** prekopíruj $B_0, B_1, \ldots, B_i$ na pomocnú pásku
3: ulož obsah pomocnej pásky do spodnej stopy blokov $B_1, \ldots, B_{i-1}$ a oboch stôp bloku B_i
4: pomocou pomocnej pásky ulož obsah bloku B_{-i} do spodnej stopy blokov $B_{-(i-1)}, \ldots, B_0$
5: **else** (B_i je úplne prázdny) prekopíruj $B_0, B_1, \ldots, B_{i-1}$ na pomocnú pásku
6: ulož obsah pomocnej pásky do spodnej stopy blokov $B_1, \ldots, B_i$
7: pomocou pomocnej pásky prekopíruj obsah bloku B_{-1} do spodnej stopy blokov $B_{-i}, \ldots, B_0$

Časovú zložitosť stroja T_2 teda môžme odhadnúť nasledovne

$$T_2(n) \leq \sum_{i=1}^{\log T_k(n)+1} m \cdot 2^i \cdot \frac{T_k(n)}{2^{i-1}} \leq l \cdot T_k(n) \log T_k(n) = O(T_k(n) \log T_k(n))$$

□

5.1.1 Redukcia času a redukcia pásky

V prípade zložitosti konkrétnych algoritmov sme sa uspokojili s "rádovou" zložitosťou; používali sme asymptotiku "O", multipkikatívnu konštantu sme zanedbávali. Ako je to pri zložitosti definovanej na TS? Ukážeme, že aj v prípade TS môžme multiplikatívne konštanty zanedbať.

redukcia pásky

Veta 5.3 *Ku každému $f(n)$-priestorovo ohraničenému TS M a konštante $0 < c < 1$ existuje ekvivalentný $cf(n)$-ohraničený TS N.*

Dôkaz: Základom dôkazu je zväčšenie abecedy páskových symbolov. Nech M je jednopáskový TS bez oddeleného vstupu, ktorého pásková zložitosť je $f(n)$; v tomto prípade zrejme $f(n) \geq n$.

$$M = (\Sigma, \Gamma, K, \delta, q_0, F)$$

Skonštruujeme nový TS N, ktorého pásková zložitosť bude nanajvýš $c \cdot f(n)$. Konštrukcia TS N je založená na tom, že jedno políčko stroja N bude simulovať m políčok stroja M. Polohu hlavy vrámci simulovanej m-tice si stroj N bude pamätať v stave ako index $i, 1 \leq i \leq m$. To umožní, aby N simuloval M krok za krokom. Presnejšie:

$$N = (\Sigma, \{B\} \cup \Gamma^m, K \cup K \times \{1, \ldots, m\} \cup K', \delta_N, q_0, F \times \{1, \ldots, m\})$$

kde prechodová funkcia δ_N je určená nie celkom formálnym popisom činnosti N.

činnosť N

1 TS N najprv "skomprimuje" vstup; využitím stavov z množiny K' zakóduje vstup $w = w_1 \ldots w_n$ do prvých $r, r = \lceil n/m \rceil$, políčok pásky. Nech b_i je obsah $i-$teho políčka pásky, $b_i = [w_{(i-1)m+1} w_{(i-1)m+2} \cdots a_{im}], a_j = B$ pre $j > n$.

2 Počiatočnú konfiguráciu stroja M vytvorí N tak, že nastaví hlavu na prvé políčko a stav KSRJ na $[q_0, 1]$.

3 A teraz už N simuluje krok za krokom výpočet TS M. Položka i stavu určuje, kedy sa hlava simulovaného stroja hýbe v rámci snímaného políčka a nespôsobuje pohyb hlavy stroja N a kedy hlavou pohne aj N.

$\delta_N(B,[q,i]) = ([B,\ldots,B],[q,i],0) \qquad q \in K$

Ku každému prvku $\boldsymbol{\delta(x,q) = (y,p,pos)}$ prechodovej funkcie stroja M vytvoríme niekoľko prvkov prechodovej funkcie stroja N. Jednotlivé prípady odpovedajú rôznym hodnotám parametra i.

$x = x_i, 1 < i < m$ $\quad \delta_N([x_1 \ldots x_i \ldots x_m],[q,i]) = ([x_1 \ldots y \ldots x_m],[p,i+pos],0)$

$x = x_i,\ i = 1$

$$\delta_N([x_1 \ldots x_m],[q,1]) = \begin{cases} ([y \ldots x_m],[p,1+pos],0), & pos \neq -1; \\ ([y \ldots x_m],[p,m],-1), & pos = -1. \end{cases}$$

$x = x_i,\ i = m$

$$\delta_N([x_1 \ldots x_m],[q,m]) = \begin{cases} ([x_1 \ldots y],[p,m+pos],0), & pos \neq 1; \\ ([x_1 \ldots y],[p,1],1), & pos = 1. \end{cases}$$

Aká je pamäťová zložitosť TS N? $S_N(n) = \max\{n, \lceil S(n)/m \rceil\}$. Je zrejmé, že konštantu m vieme nastaviť podľa konštanty c tak, aby $S_N(n) \leq c \cdot S(n)$. Aký má byť vzťah c a m?

Ak M je TS s oddeleným vstupom a jednou pamäťovou páskou, kumulovanie m symbolov do jediného budeme aplikovať len na pamäťovej páske. Formálny zápis nechávame ako cvičenie.

□

redukcia času

Veta 5.4 *Nech M je $f(n)$-časovo ohraničený TS, c konštanta, $0 < c < 1$, $f(n)/n \to 0$. Potom existuje ekvivalentný TS N, ktorý je $cf(n)$-časovo ohraničený.*

Dôkaz: Ak pôvodný TS M je $k-$páskový, tak ekvivalentný TS N bude $(k+1)$-páskový. Pre jednoduchosť budeme predpokladať, že M bol jednopáskový. Dôkaz pre $k > 1$ je analogický.

Z predchádzajúceho dôkazu sa necháme inšpirovať kompresiou pásky. Kompresiu využijeme k tomu, aby sme mali možnosť v jednom kroku odsimulovať m krokov výpočtu pôvodného stroja M. Uvedomme si, že k tomu, aby sme mohli odsimulovať m krokov stroja M, potrebujeme poznať nielen stav, ale aj pásku veľkosti $2m-1$, $m-1$ políčok naľavo od terajšej polohy a $m-1$ políčok napravo.

príprava simulácie

1 TS N skomprimuje vstup $a_1 \ldots a_n$ do prvých $\lceil n/m \rceil$ políčok druhej pásky. Nech b_i označuje obsah $i-$teho políčka pásky, $b_i = [a_{(i-1)m+1}a_{(i-1)m+2} \cdots a_{im}]$, pričom $a_j = B$ pre $j > n$. Od tohto okamihu pásku, na ktorej bol pôvodne vstup, nepoužíva a preto ju nebudeme uvažovať.

2 TS N nastaví hlavu na prvé políčko pásky a stav KSRJ na $[q_0, 1]$; rovnako ako v predchádzajúcom dôkaze hodnota 1 indikuje, že hlava simulovaného stroja sníma prvý z m symbolov, ktoré sa nachádzajú na políčku pod hlavou.

1 takt simulácie

3 TS N pracuje **v taktoch**: nech stav N na začiatku taktu je $[q,i], 1 \leq i \leq m$, obsah snímaného políčka $a_1, \ldots, a_m$

– TS N v 3 krokoch prečíta obsah domáceho aj susedných políčok; v stave si teda pamätá

$$[l_1, \ldots, l_m, a_1, \ldots, a_m, r_1, \ldots, r_m, q, m+i]$$

– aplikovaním svojej prechodovej funkcie (v jednom kroku) v stave odsimuluje m krokov TS M - tieto sa realizujú na úseku vymedzenom domácim a susednými políčkami, preto sa to dá. Zmenený stav bude

$$[L_1, \ldots, L_m, A_1, \ldots, A_m, R_1, \ldots, R_m, p, j, k]$$

– nanajvýš v 6 krokoch (podľa info v stave) prepíše N hodnoty domácich a susedných políčok a pohne správne hlavou

Časová zložitosť stroja N je daná časom potrebným na komprimovanie vstupu a zložitosťou samotnej simulácie. *čas*

$$T_N(n) \leq 2n + 10 \cdot \lceil T_M(n)/m \rceil$$

Pri vhodnej voľbe m (akej?) môžme tvrdiť

$$2n + 10 \cdot \lceil T_M(n)/m \rceil \leq c \cdot T_M(n)$$

□

5.2 Vzťah DTS a MRAM

Ukážeme, že výpočtové modely (M)RAM a TS sú z hľadiska výpočtovej sily ekvivalentné. Sústredíme sa na zmenu zložitosti pri prechode od jedného modelu k druhému. Pri označovaní časovej zložitosti TS, resp. (M)RAMu budeme dodržiavať nasledujúce značenie

$T(n)$	časová zložitosť TS
$T_U(n)$	časová zložitosť (M)RAMu pri jednotkovej cene
$T_{\log}(n)$	časovú zložitosť (M)RAMu pri logaritmoickej cene

Veta 5.5 *Nech $M = (\Sigma, \Gamma, K, q_0, \delta, F)$ je jednopáskový TS rozhodujúci jazyk L v čase $T(n)$. Potom existuje ekvivalentý RAM, pre časovú zložitosť ktorého platí $T_U(n) = O(T(n))$, resp. $T_{\log}(n) = O(T(n) \log T(n))$* *$TS \longrightarrow RAM$*

Dôkaz: Konfiguráciu TS v RAMe[1] reprezentujeme tak, že obsah pásky uložíme do pamäte RAMu a stav si budeme pamätať návestím programu. Nezabúdajme, že pri simulovaní kroku TS potrebuje poznať nielen stav, ale aj symbol pod hlavou. Tento zistíme pomocou zapamätaného indexu registra, v ktorom sa nachádza.

Pri simulácii TS postupuje RAM nasledovne:

1 prekopíruje vstup do registrov $R(2), \ldots, R(n+1)$

2 do $R(1)$ uloží polohu hlavy na páske (číslo registra, v ktorom je uložený symbol z toho políčka pásky, na ktorom je nastavená hlava)

3 krok za krokom nasleduje simulácia TS. Pre každý stav $q \in K$ máme sadu inštrukcií, ktorá pozostáva z $|\Gamma|$ podpostupností. Nech j-ta podpostupnosť q-tej sady začína inštrukciou $N(q, j)$. Nech $\delta(\mathbf{q}, \mathbf{s_j}) = (\mathbf{p}, \mathbf{s_{j'}}, \mathbf{D})$. Potom inštrukcie j-tej podpostupnosti sú nasledujúce:

[1] RAM počíta funkciu Φ_L. Ak L je jazyk, tak Φ_L označuje charakteristickú funkciu tohto jazyka. Na slovách z jazyka dáva hodnotu 1, na slovách, ktoré nie sú z jazyka, dáva hodnotu 0.

$N(q,j)$	$LOAD * 1$
$N(q,j)+1$	$SUB = j$
$N(q,j)+2$	$JZERO\ N(q,j)+4$
$N(q,j)+3$	$JUMP\ N(q,j+1)$
$N(q,j)+4$	$LOAD\ = j'$
$N(q,j)+5$	$STORE * 1$
$N(q,j)+6$	$LOAD\ 1$
$N(q,j)+7$	$ADD\ = D$
$N(q,j)+8$	$STORE\ 1$
$N(q,j)+9$	$JUMP\ N(p,1)$

– simulácia začína v počiatočnom stave

4 koncovým stavom *áno, nie* odpovedajú jednoduché postupnosti inštrukcií, ktoré uložia do akumulátora 1, resp. 0 a ukončia výpočet RAMu.

Časová a pamäťová zložitosť.

Uvedomme si, že uvedená simulácia platí aj pre (M)RAM. Keďže je jeden krok výpočtu TS simulovaný počtom krokov, ktoré sa dajú ohraničiť konštantou, a keďže jediný register $R(1)$ sa konštantou ohraničiť nedá ale závisí od polohy hlavy, dostávame:

TS	(M)RAM-jednotková	(M)RAM-logaritmická
T(n)	O(T(n))	O(T(n)log n)
S(n)	O(S(n))→ O(T(n))	O(S(n)log(S(n))) ⇓ O(T(n)log(T(n)))

□

Cvičenie 5.1 *Modifikujte simuláciu RAMu v prípade, že TS bol $k-$páskový. Zachovajte zložitosť simulácie.*

(M)RAM → TS

Veta 5.6 *Ku každému (M)RAMu existuje ekvivalentný TS.*

Dôkaz: Vychádzame z (M)RAMu, ktorý počíta funkciu $f(n)$. Konštruovaný TS bude 5-páskový. Pamäť–registre (M)RAMu budú uložené na pamäťovej páske, program spolu s čítačom inštrukcií si pamätáme v stave. Simulácia prebieha inštrukciu za inštrukciou.

1 Prvá páska obsahuje vstup

2 Druhá páska slúži na simuláciu akumulátora

3 Tretia (pamäťová) páska slúži na simuláciu pamäte (M)RAMu. Jej obsahom je postupnosť reťazcov $\sharp\sharp b(i)\sharp b(R(i))$, kde
$b(i)$ je binárne zapísaná adresa
$b(R(i))$ je binárne zapísaný obsah registra $R(i)$

4 V stave TS si pamätáme program (M)RAMu a hodnotu čítača inštrukcií

5 Simulácia (M)RAMu prebieha v taktoch; jeden takt simuluje vykonanie jednej inštrukcie (M)RAMu. Na uskutočnenie ľubovoľnej inštrukcie stačia dva operandy; tieto si môžeme uložiť na štvrtú a piatu pásku. Potom už odsimulovanie jednotlivých inštrukcií nie je problém.

Príklad simulácie $STORE * 4$

- na pamäťovej páske nájdi $\sharp\sharp b(4)\sharp b(R(4))$
- prepíš $b(R(4))$ na pomocnú pásku a nájdi na pamäťovej páske $\sharp\sharp b(R(4))\sharp b(b(R(4)))$; pamäťová páska je teda v tvare $\alpha\sharp\sharp b(R(4)\sharp b(b(R(4)))\sharp\sharp\beta$
- presuň obsah pamäťovej pásky $\sharp\sharp\beta$ na pomocnú pásku
- prekopíruj obsah akumulátora za $\alpha\sharp\sharp b(R(4)\sharp$ a na záver opäť presuň $\sharp\sharp\beta$ z pomocnej pásky

Časovú zložitosť TS získame, ak počet simulovaných inštrukcií (M)RAMu prenásobíme časom, potrebným na odsimulovanie jednej inštrukcie. Pritom *Časová zložitosť.*

- čas potrebný na simuláciu jednej RAMovskej inštrukcie je úmerný veľkosti pamäťovej pásky
- keďže násobenie a delenie sú na TS kvadratickej zložitosti, v prípade MRAMu musíme na simuláciu jednej inštrukcie uvažovať s časom kvadratickým od veľkosti pamäťovej pásky
- počet simulovaných inštrukcií je zhora ohraničený časovou zložitosťou (M)RAMu

Ostáva sa zamyslieť nad tým, *aká je veľkosť pamäťovej pásky*?

- pri logaritmickej cene $O(T(n))$ - stačí si uvedomiť, že logaritmická cena inštrukcie, ktorou (na páske vzniká, resp.) sa modifikuje ten-ktorý register, je úmerná počtu políčok, ktoré na pamäťovej páske zápis bloku zaberá
- pri jednotkovej cene - O(počet blokov $\times$ veľkosť bloku)
  počet blokov je nanajvýš $T(n)$
  veľkosť bloku ohraničíme na základe nasledujúceho faktu:

Fakt 5.7 *Nech I je vstup (M)RAMu dĺžky n, $l(B)$ maximálna dĺžka čísla použitého v programe. Po t krokoch výpočtu (M)RAMu má obsah každého registra dĺžku nanajvýš*

– $n + l(B) + t$ *v prípade RAMu;*
– c^t *v prípade MRAMu, pričom* $c = \max\{n, l(B)\}$

K dôkazu si stačí uvedomiť, že keď máme dve binárne čísla s dĺžkami binárneho zápisu a, resp. b, tak
– pre dĺžku c súčtu týchto čísel platí: $c \leq \max\{a, b\} + 1$
– pre dĺžku c súčinu týchto čísel platí: $c \leq a + b$

	časová zložitosť		TS
RAM	jednotková	$T_U(n)$	$O(T_U^3(n))$
	logaritmická	$T_{\log}(n)$	$O(T_{\log}^2(n))$
MRAM	jednotková	$T_U(n)$	$O(d^{T_U(n)})$
	logaritmická	$T_{\log}(n)$	$O(T_{\log}^3(n))$

$\square$

5.2.1 Prvá počítačová trieda

Definícia 5.3 *Nech* M, N *sú dva rôzne modely. Hovoríme, že* M *je* polynomiálne redukovateľný *na* N, *ak existuje polynóm* $p(n)$ *taký, že ku každému výpočtu* C_M *na* M *časovej zložitosti* $T_M(n)$ *existuje ekvivalentný výpočet* C_N *na* N *časovej zložitosti* $O(p(T_M(n)))$

Definícia 5.4 *Modely* M, N *sú* polynomiálne ekvivalentné, *ak* M *je polynomiálne redukovateľný na* N *a naopak.*

Definícia 5.5 Prvú počítačovú triedu *tvoria tie výpočtové modely, ktoré sú polynomiálne ekvivalentné viacpáskovému DTS.*

Zhrnutím predchádzajúcich výsledkov dostávame.

Veta 5.8 *RAM, MRAM s logaritmickou cenou, jednopáskový a viacpáskový DTS sú polynomiálne ekvivalentné modely, ktoré patria do prvej počítačovej triedy.*

Kapitola 6

Zložitostné triedy

V tejto časti sa budeme zaoberať zložitosťou na TS. Bude nás zaujímať, aký vplyv má množstvo príslušnej miery zložitosti na triedu problémov, ktoré sa s danou zložitosťou dajú riešiť. Pritom sa sústredíme nielen na výsledky, ale aj na metódy, ktoré sa na ich riešenie používajú.
V časti 6.1 sa budeme venovať dolným odhadom; vysvetlíme použitie prechodovej postupnosti pri získaní dolného odhadu na čas a prechodovej matice pri dolnom odhade na priestor. Potom vysvetlíme dôvody pre zavedenie pojmu konštruovateľnej/počítateľnej funkcie a ukážeme, že existuje nekonečná hierarchia času(6.2.2) a priesotru(6.2.1). V časti 6.3 zavedieme nedeterministický TS a napokon v časti 6.4 sa budeme venovať vzťahu determinizmu a nedeterminizmu a hierarchii zložitostných tried.

6.1 Niektoré metódy dolných odhadov

Najprv uvedieme použitie prechodovej postupnosti pre získanie dolného odhadu na čas pri rozpoznávaní jazyka na jednopáskovom TS. Pri dolných odhadoch na priestor sa zas využíva pojem prechodovej matice.
Zopakujme si:

$$\sup_{n\to\infty} f(n) \text{ je limita najnižšej hornej hranice postupnosti } f(n), f(n+1), \ldots$$

$$\inf_{n\to\infty} f(n) \text{ je limita najvyššej dolnej hranice postupnosti } f(n), f(n+1), \ldots$$

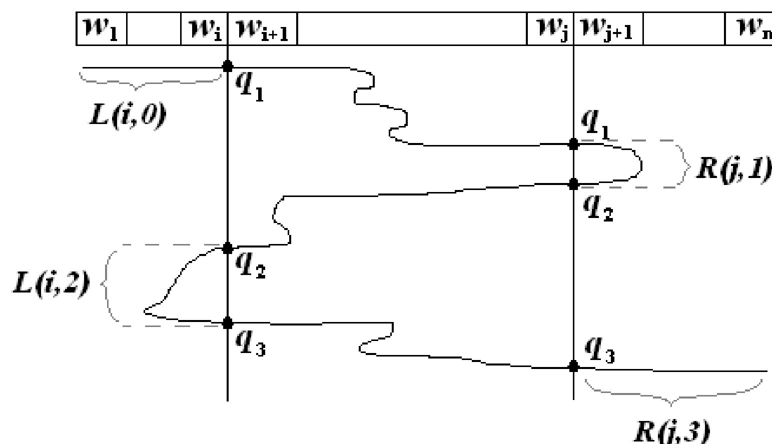
Tieto pojmy sa používajú v prípade, keď funkcia limitu nemá, napriek tomu sup, resp. inf existuje. Napr.

$$f(n) = \begin{cases} \frac{1}{n} & \text{pre n párne;} \\ \\ n & \text{pre n nepárne.} \end{cases}$$

$$\inf_{n\to\infty} f(n) = 0, \sup_{n\to\infty} f(n) = \infty \text{ ale limita funkcie neexistuje.}$$

6.1.1 Prechodová postupnosť a dolný odhad na čas

Majme jednopáskový TS, ktorého jediná jednosmerne nekonečná páska je čítacia aj prepisovacia zároveň. Predpokladajme, že stroj najprv zmení stav a až potom pohne hlavou. Potom pre fixovaný výpočet a každé prirodzené i má zmysel uvažovať o postupnosti stavov, v ktorých stroj prechádza hranicu medzi políčkom i a $i+1$. Túto postupnosť stavov (pre fixovaný výpočet) nazveme prechodová postupnosť medzi i-tym a $(i+1)$-ým políčkom.



Obrázok 6.1: Prechodová postupnosť

Lema 6.1 *Nech $w = a_1a_2 \dots a_n$ je akceptované jednopáskovým TS T a nech prechodová postupnosť medzi $i, i+1$ je rovnaká ako medzi $j, j+1$ $(i < j)$. Potom T akceptuje aj slovo $\omega = a_1a_2 \dots a_ia_{j+1} \dots a_n$.*

Dôkaz: Bez ujmy na všeobecnosti predpokladáme, že stroj končí v konfigurácii s hlavou umiestnenou na poslednom symbole vstupu.

Majme uvažovaný akceptujúci výpočet $A(w)$ na slove $w = a_1a_2 \dots a_n$. Nech uvažovaná prechodová postupnosť medzi $i, i+1$ (resp. $j, j+1$) je $q_1, q_2, \dots, q_s$.

Označme $L(i,t)$ tú časť výpočtu $A(w)$, ktorá leží medzi príchodom sprava na políčko a_i v stave q_t a odchodom doprava z políčka a_i v stave q_{t+1}, t párne
($L(i,0)$ je začiatok výpočtu $A(w)$ do okamihu, keď stroj prvýkrát prekročí hranicu medzi $i, i+1$).

Nech $R(i,t)$ označuje tú časť výpočtu, ktorá začína príchodom zľava na políčko a_{i+1} v stave q_t a odchodom z tohto políčka doľava v stave q_{t+1}, t nepárne.
Potom akceptujúci výpočet na slove $a_1a_2 \dots a_ia_{j+1} \dots a_n$ môže prebiehať nasledovne:

$$L(i,0), R(j,1), L(i,2), R(j,3), \dots$$

□

Analogicky sa dá dokázať nasledovné tvrdenie.

Lema 6.2 *Nech $v = a_1a_2 \dots a_n$, $u = b_1b_2 \dots b_m$ sú akceptované jednopáskovým TS T a nech prechodová postupnosť medzi a_i, a_{i+1} je rovnaká ako medzi b_j, b_{j+1}. Potom T akceptuje aj slovo $a_1a_2 \dots a_ib_{j+1} \dots b_m$, resp. $b_1b_2 \dots b_ja_{i+1} \dots a_n$.*

Prechodová postupnosť akýmsi spôsobom zachytáva "komunikáciu" medzi dvomi časťami vstupného slova, resp. vstupných slov.

použitie prechodovej postupnosti

Cvičenie 6.1 *Ukážte, že žiadny jednopáskový TS rozhodujúci jazyk $\{a^nb^n \mid n \geq 0\}$ nemá dĺžku prechodových postupností ohraničenú konštantou.*

Ako ale súvisia prechodové postupnosti s odhadom na čas?

Fakt 6.3 *Ak T je $T(n)$ časovo ohraničený TS, potom súčet dĺžok prechodových postupností je najviac $T(n)$.*

Je zrejmé, že ak chceme zdola odhadnúť čas, stačí zdola odhadnúť súčet dĺžok prechodových postupností.

Veta 6.4 *Ak je jazyk $L = \{wcw^R | w \in \{0,1\}^*\}$ rozpoznávaný jednopáskovým TS T, tak pre jeho časovú zložitosť $T(n)$ platí*

dolný odhad na čas

$$\sup_{n\to\infty} \frac{T(n)}{n^2} > 0$$

Dôkaz: Na základe predchádzajúcich liem je zrejmé, že nemôžu existovať také slová $w_1w_2cw_2^Rw_1^R$, $w_3w_4cw_4^Rw_3^R$, pre ktoré platí (*):

- $|w_1| = |w_3|$
- $w_1 \neq w_3$
- v akceptujúcich výpočtoch na slovách $w_1w_2cw_2^Rw_1^R$, $w_3w_4cw_4^Rw_3^R$ je prechodová postupnosť medzi w_1, w_2 rovnaká ako medzi w_3, w_4.

V opačnom prípade by totiž stroj akceptoval[1] aj slová $w_1w_4cw_4^Rw_3^R$, resp. $w_3w_2cw_2^Rw_1^R$, ktoré nie sú z jazyka.

Zafixujme dĺžku slova n a označme $p(i)$ priemernú dĺžku prechodovej postupnosti medzi i-tym a $(i+1)$-ým políčkom – pritom priemer uvažujeme cez všetky slová dĺžky n z jazyka. Potom platí, že aspoň polovica slov (dĺžky n) z jazyka má dĺžku prechodovej postupnosti medzi $i, i+1$ nanajvýš $2p(i)$. Týchto slov (označme ich W_n) je aspoň

$$\frac{1}{2}2^{\frac{(n-1)}{2}} = 2^{\frac{(n-1)}{2}-1} \tag{6.1}$$

Koľko je všetkých možných prechodových postupností dĺžky nanajvýš $2p(i)$?

$$\sum_{j=1}^{2p(i)} s^j \leq s^{2p(i)+1} \tag{6.2}$$

Ak teda spravíme rozklad množiny slov W_n podľa relácie - mať rovnakú prechodovú postupnosť medzi $i, i+1$ - vieme odhadnúť priemerný počet prvkov v jednej triede tohto rozkladu. Tento počet dostaneme ako počet slov z W_n (6.1) lomeno počet prechodových postupností dĺžky nanajvýš $2p(i)$ (6.2). Pritom tento počet nesmie byť väčší ako počet všetkých možných dokončení medzi $i+1$ a $(n-1)/2$. V opačnom prípade by sme totiž v jednej triede mali dve slová, ktoré spĺňajú (*).

Teda

$$\frac{2^{\frac{(n-1)}{2}-1}}{s^{2p(i)+1}} \leq 2^{\frac{(n-1)}{2}-i}$$

$$2^{i-1} \leq s^{2p(i)+1}$$

$$(i-1)\log 2 \leq (2p(i)+1)\log s \leq 3p(i)\log s$$

[1] vyplýva z Lemy

Pre priemernú dĺžku prechodovej postupnosti teda dostávame:

$$\frac{i-1}{3\log s} \leq p(i)$$

Ak sčítame priemerné dĺžky, dostaneme odhad na priemerný čas. Keďže aspoň jeden výpočet musí mať zložitosť aspoň priemernú, poskytuje dolný odhad na priemerný čas aj dolný odhad na čas ako taký.

$$\sum_{i=1}^{(n-1)/2} \frac{i-1}{3\log s} = \frac{1}{3\log s}\left(\sum_{i=1}^{(n-1)/2} i - \sum_{i=1}^{(n-1)/2} 1\right)$$
$$= \frac{1}{3\log s}\left(\frac{(n-1)(n+1)}{8} - \frac{1}{2}(n-1)\right) = \frac{1}{24\log s}(n^2-4n+3)$$

$$\sup_{n\to\infty} \frac{T(n)}{n^2} \geq \frac{1}{24\log s} > 0$$

□

6.1.2 Prechodová matica a dolný odhad na pamäť

Na riešenie niektorých problémov pamäť, presnejšie pamäťovú pásku, nepotrebujeme; vystačíme si s konštantnou pamäťou, ktorú nám poskytuje stav. Ako je to však s pamäťovou zložitosťou v prípade, keď nám konštantná pamäť nestačí?

stav pamäte

Pre TS s jednou vstupnou a jednou pamäťovou páskou je *stav pamäte* daný

- obsahom pamäťovej pásky
- polohou hlavy na pamäťovej páske
- stavom konečnostavovej riadiacej jednotky

Uvedomme si, že ak $L(n)$ je pásková zložitosť TS, tak počet rôznych stavov pamäte je zhora ohraničený hodnotou

$t^{L(n)}L(n)s$, kde t označuje mohutnosť páskovej abecedy
s je počet stavov

Majme vstupné slovo dĺžky n. Nech r je počet stavov pamäte, ktoré stroj pri spracovaní slova dĺžky n môže dosiahnuť. Môžeme predpokladať, že jednotlivé stavy pamäte sú očíslované $1, 2, \dots, r$.

prechodová matica

Prechodová matica pre $L(n)$ páskovo ohraničený TS je matica $T^n_{r\times r}$, ktorej prvky sú z množiny $\{00, 01, 10, 11\}$.
Nech w je vstupné slovo dĺžky n, u jeho koncové podslovo (sufix), $r = t^{L(n)}L(n)s$.
Hovoríme že *prechodová matica $T^n_{r\times r}$ popisuje koncové podslovo u* ak platí

- ak TS začne čítať u v stave pamäti i a dosiahne stav pamäte j bez toho, aby opustil koncové podslovo u, tak $T^n(i,j) = 1*$ (inak $T^n(i,j) = 0*$)[2]

- ak TS začne čítať u v stave pamäti i a prvýkrát opustí u doľava v stave pamäti j, tak $T^n(i,j) = *1$ (inak $T^n(i,j) = *0$)

Veta 6.5 *Nech TS je $L(n)$ páskovo ohraničený TS s jednou vstupnou a jednou pamäťovou páskou. Nech w_1, w_2 sú vstupné slová dĺžky n, ktoré TS akceptuje. Nech $w_i = v_iu_i$, $i = 1, 2$ a nech nejaká prechodová matica T popisuje aj u_1 aj u_2. Potom TS akceptuje aj slovo v_2u_1, resp. v_1u_2.*

[2] $*$označuje ľubovoľný zo symbolov $\{0, 1\}$

Dôkaz: Môžeme predpokladať, že po akceptovaní stroj neurobí žiadne kroky. Majme postupnosť konfigurácií C stroja TS, ktorá vedie k akceptovaniu slova v_1u_1. Nech $Q_1, Q_2, \dots, Q_p$ je postupnosť stavov pamäte, v ktorých pri tomto výpočte prekračuje hranicu medzi v_1 a u_1. Potom C možno napísať

$$C = \alpha_1\beta_1\alpha_2\beta_2\dots,$$

kde

α_i je časť výpočtu na časti slova v_1 pred i-tym prekročením hranice medzi v_1 a u_1
β_i je časť výpočtu na časti slova u_1 pred i-tym prekročením hranice medzi u_1 a v_1

Všimnime si výpočet TS pri vstupe na v_1u_2. Výpočet začína α_1. Potom stroj v stave pamäte Q_1 vstúpi na časť slova u_2. Keďže T popisuje aj u_1 aj u_2 a keďže sa TS pri výpočte C vracia na časť slova v_1 v stave pamäte Q_2, existuje výpočet γ_1 TS, ktorý začína vstúpením TS na u_2 v stave pamäte Q_1 a jeho opustením smerom doľava v stave pamäte Q_2. Analogicky ďalej. Výpočet D TS na slove v_1u_2 teda možno popísať nasledovne

$$D = \alpha_1\gamma_1\alpha_2\gamma_2\dots$$

Ak pri výpočte C stroj akceptoval s hlavou v časti v_1, bude tomu tak aj v prípade D. Ak pri výpočte C stroj akceptoval s hlavou v časti u_1, tak hranicu medzi u_1 a v_1 poslednýkrát prekračoval v stave pamäte Q_p. Z tohto stavu pamäte sa bez opustenia slova u_1 vedel dostať do stavu pamäte Q_A odpovedajúcom akceptovaniu. Druhá položka $T(Q_p, Q_A)$ je teda 1 a zaručuje, že ak stroj vstúpi na u_2 v stave pamäte Q_p, môže bez opustenia slova u_2 dosiahnuť stav pamäte Q_A - môže teda akceptovať.

□

Analogické tvrdenie vieme dokázať aj pre prípad, keď prechodová matica popisuje dve rôzne koncové podslová toho istého vstupného slova.

Teraz už môžme prejsť k využitiu prechodovej matice pri dolnom odhade na priestor.

Veta 6.6 *Nech T je $L(n)$ páskovo ohraničený, kde $L(n)$ nie je zhora ohraničená žiadnou konštantou. Potom*

dolný odhad na pamäť

$$\sup_{n\to\infty} \frac{L(n)}{\log\log n} > 0$$

Dôkaz:

- Keďže hodnota $L(n)$ nie je ohraničená konštantou, potom pre každé k existuje najmenšie také n_k, že $L(n_k) \geq k$
- Nech r je počet stavov pamäti, ktoré neobsahujú viac ako $L(n_k)$ políčok
- Nech $w_k, |w_k| = n_k$ je *najkratšie* také, že sa pri jeho spracovaní použije aspoň k políčok

1. Ukážeme, že w_k nemôže mať dve rôzne koncové podslová popísané tou istou prechodovou maticou. Inak by sme totiž vedeli skonštruovať slovo kratšie ako n_k, pri spracovaní ktorého sa použije aspoň k políčok.
2. Potom spočítame počet všetkých možných prechodových matíc, ktorý nesmie byť menší, ako počet všetkých rôznych koncových podslov slova w_k.

K bodu 1 Nech $w = v_1v_2v_3$, $v_2 \neq \epsilon$, $T^{(n_k)}$ popisuje aj v_2v_3, aj v_3. Potom stroj pri spracovaní slova v_1v_3 použije aspoň k políčok pásky.
Prečo? Dôkaz je podobný ako v prípade Veta 6.6, treba len akceptujúcu konfiguráciu nahradiť konfiguráciou, ktorá použije k políčok pásky. To je ale spor s predpokladom, že w_k bolo najkratšie také, pri spracovaní ktorého sa použilo aspoň k políčok pásky.

K bodu 2 Musí teda existovať aspoň $n_k - 1$ rôznych prechodových matíc typu $r \times r$. Takýchto matíc je 4^{r^2}, kde $r = s \cdot t^{L(n_k)L(n_k)}$. Preto

$$\begin{aligned} n_k - 1 &\leq 4^{r^2} \\ \log(n_k - 1) &\leq r^2 \log 4 = (s \cdot t^{L(n_k)} \cdot L(n_k))^2 \cdot \log 4 \\ \log\log(n_k - 1) &\leq 2(\log s + L(n_k)\log t + \log L(n_k)) + 1 \\ \tfrac{1}{2}\log\log n_k &\leq 3L(n_k)\log 2st \end{aligned}$$

[3] Pre nekonečne veľa n takých, že $n = n_k$(pre nejaké k) teda platí

$$\frac{L(n)}{\log\log n} \geq \frac{1}{6\log 2st}$$

□

6.2 Hierarchia času a priestoru

Pri definovaní výpočtového modelu definujeme aj miery zložitosti, ktoré sú preň relevantné. Potom má zmysel uvažovať o triede problémov, ktoré sa dajú riešiť s nejakým obmedzením na tú-ktorú mieru zložitosti. Prirodzene sa tak dostávame k zložitostným triedam. Hlavným výpočtovým modelom je pre nás TS, pričom základné zložitostné triedy sú definované ohraničením jeho mier zložitosti - času a priestoru. Vo všeobecnosti sa používa značenie $DTIME(f(n))$ na označenie triedy problémov, ktoré sa dajú na deterministickom TS riešiť s časovou zložitosťou $f(n)$ a $DSPACE(f(n))$ označuje tú triedu problémov, ktoré sa dajú na deterministickom TS riešiť s priestorovou zložitosťou $f(n)$.

Majme teda TS, ktorý je $f(n)$ – či už časovo, alebo priestorovo – ohraničený. Ak by sme nejakéme inému TS túto informáciu nejako vedeli odovzdať, mohol by ju možno využiť. Bol by napríklad schopný spočítať počet všetkých možných konfigurácií, do ktorých sa pôvodný TS pri výpočte na danom vstupe (dĺžky n) môže dostať a tak by vhodnou simuláciou a počítaním krokov mohol identifikovať, že sa pôvodný TS dostal do cyklu. Teraz už zrejme budeme rozumieť, prečo uvádzame nasledujúce definície, zavádzame definované pojmy.

páskovo konštruovateľná funkcia

Definícia 6.1 *Funkcia $L(n)$ sa nazýva páskovo konštruovateľná, ak existuje deterministický Turingov stroj T, ktorý je $L(n)$ priestorovo ohraničený a ktorý na každom vstupe dĺžky n použije na prvej páske* presne *$L(n)$ políčok.*

časovo konštruovateľná funkcia

Definícia 6.2 *Funkcia $T(n)$ sa nazýva časovo konštruovateľná, ak existuje deterministický Turingov stroj T, ktorý na každom vstupe dĺžky n spraví presne $T(n)$ krokov.*

Veta 6.7 *Ku každej konštruovateľnej funkcii $f(n)$ existuje rekurzívny jazyk LT, resp. LS taký, že $LT \notin DTIME(f(n))$ a $LS \notin DSPACE(f(n))$.*

[3] $\log\log(n_k - 1) \geq \frac{1}{2}\log\log n_k$ pre $n_k \geq 3$

Dôkaz: Uvedieme dôkaz len pre prípad času; dôkaz pre priestor je analogický. Nech $f(n)$ je konštruovateľná funkcia a M_f TS, ktorý ju konštruuje. Nech x je binárny reťazec. Potom označme M_x ten DTS, ktorého kódom je práve x; ak x nie je kódom TS, môžme ho považovať za kód TS rozhodujúceho prázdny jazyk. Definujme jazyk L_f nasledovne:
$L_f = \{x|$ výpočet M_x na vstupe x sa zastaví po nanajvýš $f(|x|)$ krokoch v zamietajúcej konfigurácii$\}$

Jazyk L_f je rekurzívny. Dôkaz spravíme konštrukciou DTS T, ktorý ho rozhoduje.

- T so vstupom w najprv odsimuluje M_f
- chápe w ako kód M_w a simuluje $f(|w|)$ krokov výpočtu M_w so vstupom w
- ak M_w zastavil v zamietajúcej konfigurácii, akceptuje.

$\mathbf{L_f \notin DTIME(f(n))}$ vyargumentujeme sporom. Nech existuje DTS M rozhodujúci L_f v čase $f(n)$. Nech $M = M_x$ pre nejaké x.

$$x \in L_f = L(M) \Leftrightarrow x \notin L(M_x) = L(M)$$

□

6.2.1 Priestorová hierarchia

Veta 6.8 *Nech $f_1(n), f_2(n)$ sú konštruovateľné funkcie a nech $f_1(n) = o(f_2(n))$. Potom existuje jazyk L, ktorý patrí do $DSPACE(f_2(n))$, ale nepatrí do $DSPACE(f_1(n))$.*

Dôkaz: Jazyk L popíšeme konštrukciou TS T, ktorý ho rozhoduje. Tento budeme konštruovať tak, aby bol $f_2(n)$ priestorovo ohraničený. Potom ukážeme, že L sa nedá rozhodovať žiadnym TS s priestorovou zložitosťou $f_1(n)$.

Nech $T2 = (\Sigma_2, \Gamma_2, K_2, \delta_2, B, q_{02}, F_2)$ je TS, ktorý konštruuje funkciu $f_2(n)$. Stroj T, $T = (\Sigma, \Gamma, K, \delta, B, q_0, F)$, rozhodujúci jazyk L, pracuje nasledovne:

1. abeceda $\Sigma = \{a_x | a \in \Sigma_2, x \in \{0, 1\}\}$
2. nech w je vstupné slovo; označme $bin(w)$ binárny reťazec tvorený indexami vstupného slova w
3. ignorujúc indexy T simuluje $T2$ a vyhradí na páske priestor veľkosti $f_2(n)$; odteraz zastaví bez akceptovania vždy, keď by chcel opustiť tento vyhradený priestor. Takto sme zabezpečili, že $L \in DSPACE(f_2(n))$
4. T chápe $bin(w)$ ako binárny zápis čísla i; vygeneruje kód i-teho TS M_i
5. T simuluje M_i so vstupom $bin(w)$
6. T akceptuje vtedy a len vtedy ak M_i zastane a zamieta, resp. ak sa výpočet M_i zacyklí

Diagonalizáciou ukážeme, že neexistuje $f_1(n)$ priestorovo ohraničený TS rozhodujúci $L = L(T)$.
Nech takýto TS existuje. Nech je v číslovaní strojov j-ty, teda M_j. Uvažujme vstupné slovo w také, že $bin(w)$ je dvojkovým zápisom čísla j. Uvedomme si, že takýchto slov je vlastne nekonečne veľa, lebo ak $bin(w)$ je zápisom čísla j, tak aj $0^*bin(w)$ je zápisom čísla j. Potrebujeme vyargumentovať, že sa T v svojej práci úspešne dostane cez body 3.,4.,5. Potom v bode 6. nastane spor.

K bodu 4. Nech $N4$ je také, že $f_2(N4) \geq$ dĺžka kódu M_j

K bodu 5. Vzhľadom k tomu, že abeceda stroja T je fixná, musí abecedu stroja M_j kódovať. Na kódovanie znakov použije $c(j)$ znakov(je to konštanta závislá od j). Takto sa priestorová zložitosť zmení z $f_1(w)$ na $c(j)f_1(w)$. Kedže $f_1(n) = o(f_2(n))$, existuje N5 také, že $\forall n \geq N5$: $c(j)f_1(n) \leq (f_2(n))$.

K bodu 6. Aby sme zistili, či sa M_j zacyklil, potrebujeme vedieť (stačí) počet všetkých možných konfigurácií stroja M_j na slove dlžky |w|. Na výpočet tejto hodnoty H stačí znalosť $f_1(n)$ - to vieme, mohutnosť pracovnej abecedy – to vyčítame z kódu, dlžka vstupného slova – tú vieme zistiť.
Kedže $f_1(n)$ je konštruovateľná, nie je problém H vypočítať (napísať TS, ktorý bude H počítať). Pri simulovaní M_j počítame počet odsimulovaných krokov a pri presiahnutí hodnoty H vieme, že stroj sa zacyklil.$H = |\Gamma|^{f_1}(n)f_1(n)|K_j|$, čo znamená, že existuje N6 také, že počet bitov, potrebných na zapísanie hodnoty H je menší ako $f_2(N6)$.

Nech $N = max\{N4, N5, N6, \lfloor logj \rfloor + 1\}$. Uvažujme slovo $v = O^*bin(w), |v| \geq N$. Pre takto zvolené slovo sa T dostane až do bodu 6 a platí: $v \in L = L(T) = L(M_j) \Leftrightarrow v \notin L(M_j)$

□

6.2.2 Hierarchia času

Veta 6.9 *Nech $f_2(n)$ je konštruovateľná funkcia a nech $f_1(n)logf_1(n) = o(f_2(n))$. Potom existuje jazyk, ktorý patrí do $DTIME(f_2(n))$, ale nepatrí do $DTIME(f_1(n))$.*

Dôkaz: Analogicky ako v prípade priestorovej hierarchie. Navrhneme $f_2(n)$-časovo ohraničený TS M, ktorý rozpoznáva jazyk $L \in \{0,1\}^*$.
Stroj M na vstupe w, $|w| = n$,

1. vypočíta $f_2(|w|)$
2. $w = 1^*0v$ v chápe ako kód TS M_v
3. M simuluje prvých nanajvýš $f_2(|w|)$ krokov stroja M_v
4. M akceptuje w práve vtedy, ak M úspešne ukončil simuláciu M_v, pričom M_v neakceptoval w

Predpokladajme, že M je $f_1(n)$-časovo ohraničený. Potom existuje taký vstup α, že $M = M_\alpha$. Kedže M_α môže mať ľubovoľný počet pások a my ho simulujeme na stroji s fixovaným počtom pások, môže dôjsť k logaritmickému spomaleniu. To je jeden zdroj spomalenia. Druhý zdroj je počítanie krokov simulovaného stroja - počítadlo je veľkosti $4logf_1(n)$ - teda tiež logaritmické spomalenie.

Vzhľadom na platnosť podmienky $f_1(n)logf_1(n) = o(f_2(n))$ má M_α dostatočne veľa času na simuláciu M_α a v bode 4. dochádza k sporu, keď M akceptuje vstupné slovo α práve vtedy, keď ho M_α neakceptuje. Pritom však $L(M) = L(M_\alpha)$. □

Požiadavka na konštruovateľnosť funkcie v predchádzajúcej vete je podstatná. Ak by sme od nej upustili, dostaneme pomerne prekvapivý výsledok:

Veta 6.10 *Existuje rekurzívna funkcia $f : N \to N$ taká, že $DTIME(f(n)) = DTIME(2^{f(n)})$*

Dôkaz: Dôkaz je založený na vhodnej konštrukcii funkcie $f(n)$. Zadefinujeme funkciu f tak, že výpočet každého DTS na vstupe dĺžky n sa zastaví alebo nanajvýš po $f(n)$ krokoch, alebo po viac ako $2^{f(n)}$ krokoch, alebo sa nezastaví vôbec.

Uvažujme rastúce usporiadanie kódov TS - $M_0, M_1, \ldots$. Pre každú dvojicu prirodzených čísel i, k definujme vlastnosť $P(i, k)$

$\mathbf{P(i, k)} = 1 \Leftrightarrow$ výpočet každého spomedzi strojov $M_0, M_1, \ldots, M_i$ na každom vstupe dĺžky i sa buď zastaví po nanajvýš k krokoch, alebo sa zastaví po viac ako 2^k krokoch, alebo sa nezastaví vôbec.

Napriek tomu, že niektoré spomedzi uvažovaných výpočtov môžu byť nekonečné, *je vlastnosť $P(i, k)$ rozhodnuteľná* - pre každý zo strojov $M_0, M_1, \ldots, M_i$ stačí odsimulovať nanajvýš 2^{k+1} krokov výpočtu na každom vstupe dĺžky i.

Uvažujme postupnosť čísel *definícia f(i)*

$k_1 = 2i$ a pre $j = 2, 3, \ldots$ $k_j = 2^{k_{j-1}} + 1$

Uvedomme si, že každý zo vstupov dĺžky i môže narušiť platnosť vlastnosti $P(i, k_j)$ pre nanajvýš jednu hodnotu j, resp. k_j. Ak totiž stroj na danom vstupe zastal po t krokoch, pričom $k_j < t < 2^{k_j}$, tak zastal po nanajvýš $2^{k_{j+1}}$ krokoch.

Existuje číslo **c** také, že $\mathbf{P(i, c) = 1}$.

Potom pomocou tejto hodnoty definujeme hodnotu našej funkcie v bode i - $\mathbf{f(i)} := \mathbf{k_c}$

Keďže $DTIME(f(n)) \subseteq DTIME(2^{f(n)})$, stačí ukázať iba platnosť opačnej inklúzie.

$\mathbf{L \in DTIME(2^{f(n)}) \Rightarrow L \in DTIME(f(n))}$ Jazyk L je rozhodovaný nejakým TS, nech je to stroj M_j, v čase $2^{f(n)}$. Potom pre každý vstup w dĺžky $|w| \geq j$ (teda pre všetky vstupy až na konečne veľa) nie je možné, aby sa výpočet stroja M_j zastavil po viac ako $f(|w|)$ a menej ako $2^{f(|w|)}$ krokoch. [4] Keďže ale stroj určite zastaví po nanajvýš $2^{f(|w|)}$ krokoch, tak vlastne zastane po nanjvýš $f(|w|)$ krokoch.

Táto vlastnosť ale platí len pre slová dĺžky aspoň j. Pre ostatné - ktorých je ale *konečne veľa*, stačí na ich rozpoznanie konečný stav. Pre túto konečnú množinu slov teda na rozpoznanie stačí lineárny čas.

□

6.3 Nedeterministický TS

Poslednou modifikáciou Turingovho stroja je pridanie nedeterministického rozhodovania. Nedeterministickým rozhodovaním myslíme potenciálnu možnosť výberu nasledujúceho kroku z *konečnej množiny* možností. Ak pridáme TS možnosť nedeterministického rozhodovania, dostaneme nedeterministický TS.

Formálnejšie: *Nedeterministický k-páskový TS* (NTS)je sedmica $(\Sigma, \Gamma, K, \delta, B, q_0, F)$, kde *NTS*

$\Sigma, \Gamma, K, B, q_0, F$ majú význam ako pri definícii TS,

[4] pri definovaní hodnoty $f(d), d \geq j$ sme brali do úvahy aj stroj M_j

$\delta: \quad (\Sigma \times \Gamma^k \times K) \rightarrow 2^{\Gamma^k \times K \times \{0,1,-1\}^k \times \{0,1,-1\}}$ je prechodová funkcia[5].
Pojmy konfigurácia, krok, výpočet, akceptujúci výpočet ostávajú rovnaké ako v prípade deterministického Turingovho stroja. Čo si treba vyjasniť, je akceptovanie. Vzhľadom k tomu, že z danej konfigurácie existuje viacero možností, ako pokračovať vo výpočte, máme možnosť realizovať pre jedno vstupné slovo viacero výpočtov. Pritom niektoré z týchto výpočtov môžu končiť v akceptujúcej konfigurácii, iné v zamietajúcej. Čo v takom prípade?

NTS *akceptuje* práve vtedy, keď existuje akceptujúci výpočet a *zamieta*, ak akceptujúci výpočet neexistuje. To znamená, že zamieta, ak všetky výpočty na danom vstupe končia v zamietajúcej konfigurácii (resp. neskončia)

Zamyslime sa nad tým, čo v prípade TS nedeterminizmus prináša. Ukážeme, že v porovnaní s deterministickým TS sa výpočtová sila nemení. Mení sa len zložitosť výpočtov.

Veta 6.11 *Nech N je nedeterministický TS rozhodujúci jazyk L. Potom existuje ekvivalentný deterministický TS D.*

Dôkaz: Bez ujmy na všeobecnosti môžeme predpokladať, že NTS N je jednopáskový. Uvažujme strom výpočtu $\mathcal{T}$ stroja N. V koreni stromu je počiatočná konfigurácia, vrchol–konfigurácia C má ako syna vrchol–konfiguráciu C' práve vtedy, keď $C \mapsto_N C'$.

DTS D bude dvojpáskový a bude simulovať výpočet NTS N "prechádzaním" stromu výpočtu do šírky. V každom okamihu simulácie je na jednej z pások DTS uložená postupnosť konfigurácií z $i-$tej úrovne $\mathcal{T}$ (stará), na druhej páske sa vytvára $(i+1)-$á úroveň stromu výpočtu $\mathcal{T}$ (nová). Úlohu pások (ktorá je nová a ktorá stará) rozlišujeme stavom.

$i \longrightarrow i+1$

Nech obsahom starej pásky je $i-$ta úroveň $\mathcal{T}$.

- hlavy DTS D stoja na začiatku pások
- kým nie je koniec starej pásky
  - nech C je konfigurácia zo starej pásky;
  - vytvor na novej páske konfigurácie $C_1, \dots, C_k$ také, že $C \mapsto_N C_k$
- posuň hlavy na oboch páskach doľava
- vymeň úlohy starej a novej pásky

ukončenie

Ukončenie práce DTS nastane z dvoch dôvodov:
– ak DTS D vytvorí akceptujúcu konfiguráciu, zastane a akceptuje
– ak vytvorená úroveň obsahuje len listy, pričom každý odpovedá zamietajúcej konfiguráci, DTS D zastane a zamieta.

Časová aj priestorová zložitosť touto simuláciou narastie exponenciálne. Efektívnejšie simulácie v nasledujúcej časti.

□

[5] Pre množinu A označíme symbolom 2^A množinu všetkých podmnožín množiny A

6.4 Vzťahy medzi zložitostnými triedami

V tejto časti nás budú zaujímať vzťahy medzi zložitostnými triedami. Špeciálne si budeme všímať vzťah nedeterministického a deterministického času a priestoru, resp. to, koľko "zaplatíme" za prechod od nedeterminizmu k determinizmu.

Pre niektoré (najčastejšie používané) zložitostné triedy používame špeciálne označenie:

$$\begin{array}{llll} P = & = \bigcup_{k\geq 0} DTIME(n^k) & DLOG & = DSPACE(logn) \\ NP = & = \bigcup_{k\geq 0} NTIME(n^k) & NLOG & = NSPACE(logn) \\ ETIME & = \bigcup_{k\geq 0} DTIME(2^{kn}) & PSPACE & = \bigcup_{k\geq 0} DSPACE(n^k) \\ NETIME & = \bigcup_{k\geq 0} NTIME(2^{kn}) & NPSPACE & = \bigcup_{k\geq 0} NSPACE(n^k) \\ EXPTIME & = \bigcup_{k\geq 0} DTIME(2^{n^k}) & & \\ NEXPTIME & = \bigcup_{k\geq 0} NTIME(2^{n^k}) & & \end{array}$$

Veta 6.12 *Nech $f(n)$ je konštruovateľná funkcia. Potom*

1. $DSPACE(f(n)) \subseteq NSPACE(f(n))$ *a* $DTIME(f(n)) \subseteq NTIME(f(n))$
2. $NTIME(f(n)) \subseteq DSPACE(f(n))$
3. $NSPACE(f(n)) \subseteq \bigcup_{c>0} DTIME(c^{logn+f(n)})$
4. $NTIME(f(n)) \subseteq \bigcup_{c>0} DTIME(c^{f(n)})$
5. $NSPACE(f(n)) \subseteq DSPACE(f(n)^2)$ *pre funkciu* $f(n) \geq logn$***[Savitchova veta]***

Dôkaz:

1. platí triviálne – determinizmus je špeciálnym prípadom nedeterminizmu
2. Nech N je NTS, $f(n)$ jeho časová zložitosť, d miera nedeterminizmu (odchádzajúci stupeň vrchola v strome výpočtu; max počet možností výberu na pravej strane prechodovej funkcie). Uvažujme nasledovnú simuláciu NTS N deterministickým strojom D:
   - D si na prvej páske pamätá vstupné slovo
   - na druhej páske systematicky generuje postupnosti $a_1 a_2 \ldots a_f(n), a_i \in \{1, 2, \ldots, d\}$
   - na tretej páske simuluje výpočet N so vstupom w, pričom v i-tom kroku vyberá a_i-tu možnosť z množiny možností; ak ich toľko nie je, prechádza na dalšiu postupnosť
   - ak sa pri simulácii dostane D do akceptujúcej konfigurácie N, akceptuje a zastane. Inak zamieta.
3. Nech N je $f(n)$ priestorovo ohraničený TS. Každá konfigurácia stroja N je jednoznačne určená pozíciou hlavy na vstupe, obsahom pásky a polohou hlavy na nej, stavom. Preto je počet #(CN) všetkých možných konfigurácií stroja N možno ohraničiť nasledovne

$$\#(CN) \leq n|K|log(f(n))|\Gamma|^{f(n)} \leq d^{logn+f(n)}$$

Ak chceme vedieť, či N akceptuje vstupné slovo w, pýtame sa, či sa zo vstupnej konfigurácie (ktorá je jednoznačná) vie N dostať do akceptujúcej konfigurácie (konfigurácia s akceptujúcim stavom; bez ujmy na všeobecnosti predpokladáme, že N má jediný akceptujúci stav). Ak si predstavíme, že všetky konfigurácie tvoria (orientovaný) konfiguračný graf :

– množina vrcholov je tvorená množinou konfigurácií

– C1 je synom C2 práve vtedy keď z C1 sa dá v jednom kroku dostať do C2, zredukovali sme náš problém na problém dosiahnuteľnosti akceptujúcej konfigurácie z počiatočnej konfigurácie. Samotné prehľadávanie grafov je zložitosti $O(m) = O(\#(CN)^2)$. Predpokladá ale, že poznáme susedov. My ich vždy, ked treba, vygenerujeme. Toto generovanie spôsobí zdržanie $O(logn + f(n))$. Preto je celkový čas $O((logn + f(n))(\#(CN))^2) = O(c^{logn+f(n)})$

4. Analýza času simulácie z bodu 2.

5. **dôkaz Savitchovej vety** Nech N je NTS priestorovej zložitosti $S(n)$, kde $S(n)$ je konštruovateľná funkcia, $S(n) \geq \log n$. Nech T je DTS, ktorý konštruuje $S()$.

   Popíšeme konštrukciu DTS M ekvivalentného NTS N. Keďže

   $$w \in L(N) \Leftrightarrow \exists \text{ akceptujúci výpočet na slove } w$$

   základom simulácie je overenie existencie takéhoto výpočtu. Pri overovaní existencie akceptujúceho výpočtu využijeme nasledujúcu funkciu $OVER(C_1, C_2, t)$, kde C_1, C_2 sú konfigurácie, t je čas

   $$OVER(C_1, C_2, t) = \begin{cases} 1, & \text{ak sa z } C_1 \text{ do } C_2 \text{ dostaneme za } t \text{ krokov;} \\ 0, & \text{inak.} \end{cases}$$

   Ako využijeme $OVER()$?

   Nech $\#(CN)$ je počet všetkých možných konfigurácií nedeterministického TS N s priestorovou zložitosťou $S(n)$. Uvedomme si, že ak existuje ekceptujúci výpočet TS na vstupe w dĺžky n, tak na tomto vstupe existuje aj akceptujúci výpočet, ktorého dĺžka je nanajvýš $\#(CN)$. Vzhľadom ku konštruovateľnosti $S(n)$ teda platí $T(n) \leq \#(CN)$. Bez ujmy na všeobecnosti predpokladajme, že každý výpočet tohto stroja trvá presne $T(n)$ krokov (prečo môžme?) Potom

   $w \in L(N) \Leftrightarrow \sum_{C_A} OVER(C_0, C_A, T(n)) = 1$, kde
   - C_O je počiatočná konfigurácia na vstupe w
   - C_A je akceptujúca konfigurácia pre vstup dĺžky $|w|$
   - $T(n) = \#(CN)$ je maximálny čas pre vstup dĺžky $|w|$
   - $\sum_{C_A}$ označuje disjunkciu cez všetky akceptujúce konfigurácie.

   Uvedomme si, že k tomu, aby simulácia pre vstup w mohla byť založená na pozorovaní $w \in L(N) \Leftrightarrow \sum_{C_A} OVER(C_0, C_A, T(n)) = 1$ musí simulujúci TS vedieť počítať nielen funcku $OVER$, ale musí pre ňu vedieť najpr vypočítať potrebné parametre C_0, C_A, T. Zamyslite sa, ako DTS M počíta tieto parametre!

   Funkciu $OVER(C_1, C_2, t)$ môžeme počítať rekurzívne:

   $t = 0$, $OVER(C_1, C_2, t) = 1 \Leftrightarrow C_1 = C_2$;
   $t = 1$, $OVER(C_1, C_2, t) = 1 \Leftrightarrow C_1 \mapsto C_2$;
   $t \geq 2$, $OVER(C_1, C_2, t) = \sum_C OVER(C_1, C, t/2) \bigwedge OVER(C, C_2, t/2)$,
   kde disjunkcia ide cez všetky konfigurácie C stroja N.

Výpočet rekurzívnej funkcie simulujeme na TS simuláciou systémového zásobníka. Potrebujeme si uchovávať jednotlivé volania = parametre C_1, C_2, t. O priestorovej zložitosti $S_D(n)$ simulujúceho DTS M platí:

$$\begin{aligned} S_D(n) &= O(\text{hĺbka vnorenia} \times \text{ veľkosť hlavičky volania}) \\ &= O(\log T(n) \times S(n)) \end{aligned}$$

Preto $S_D(n) = O(S^2(n))$.

□

Zhrnutím dostávame

$$DLOG \subseteq NLOG \subseteq P \subseteq NP \subseteq NPSPACE = PSPACE \subseteq EXPTIME \subseteq NEXPTIME$$

Pri všetkých inklúziách predpokladáme, že sú ostré, ale ani pre jednu z nich sa to nepodarilo ani dokázať ani vyvrátiť. Z výsledkov o hierarchiách vieme, že

$$\begin{aligned} DLOG &\subset PSPACE & \qquad NLOG &\subset NPSPACE \\ P &\subset EXPTIME & \qquad NP &\subset NEXPTIME \end{aligned}$$

Ukázať ostrosť niektorej z inklúzií je centrálnym problémom teórie zložitosti. Prezentujeme techniku, ktorá by mohla byť nápomocná.

Veta 6.13 *Ak $DSPACE(n) \subseteq P$, tak $P = PSPACE$*

Dôkaz: Kedže $P \subseteq PSPACE$, potrebujeme vlastne ukázať, že za predpokladu, že $DSPACE(n) \subseteq P$, platí aj opačná inklúzia $PSPACE \subseteq P$.

Nech teda $DSPACE(n) \subseteq P$ a $L \in PSPACE$. Ukážeme, že potom $L \in P$.

Uvažujme ten DTS M polynomiálnej priestorovej zložitosti $p(n)$, ktorý rozhoduje L, $L(M) = L$. Skonštruujme nový jazyk L' nasledovne:

$L' = \{w10^{p(|w|)} | w \in L\}$
Konštrukciou TS $M', L(M') = L'$, lineárnej časovej zložitosti ukážeme, že $L' \in DSPACE(n)$. M' pracuje (napríklad) takto:

- M' nájde najpravší symbol 1 = tým pozná aj vstupné slovo w, aj má vymedzený priestor veľkosti $p(|w|)$
- M' simuluje M so vstupom w, na čo mu stačí priestor $p(|w|)$, čo je však lineárne vzhľadom na veľkosť vstupu do M'!

Preto M' patrí do $DSPACE(n)$ a teda aj P. To znamená, že existuje polynomiálne časovo ohraničený TS M'', ktorý rozpoznáva L'. Tento stroj M'' využijeme na rozpoznanie pôvodného jazyka L v polynomiálnom čase nasledovne:

- v polynomiálnom čase vyrobí zo vstupu w na pracovnú pásku $w10^{p(|w|)}$
- simuluje výpočet M'' na vstupe $w10^{p(|w|)}$

Tým sme ukázali, že $PSPACE \subseteq P$.

□

Dôsledok 6.14 $P \neq DSPACE(n)$

6.5 Uzavretosť nedeterministického priestoru na komplement

V poslednej časti tejto kapitoly sa budeme venovať otázke uzavretosti nedeterministického priestoru na komplement. Otázka uzavretosti kontextových jazykov, a teda lineárneho priestoru, na komplement bola otvorená veľmi dlho. Napokon sa ju podarilo vyriešiť v rovnakom čase nezávisle dvom ľuďom—Neilovi Immermanovi a Robertovi szelepcsenyimu[6]. Získaný výsledok je všeobecnejší, hovorí o uzavretosti nedeterministického priestoru na komplement.

Veta 6.15 (Immerman-Szelepcsenyi) *Nech funkcia $f(n)$ je konštruovateľná, pričom $f(n) \geq \log n$. Potom trieda $NSPACE(f(n))$ je uzavretá na komplement.*

Dôkaz: Majme jazyk L rozpoznávaný $f(n)$-priestorovo ohraničeným TS T. Kedy slovo w patrí/nepatrí do jazyka L? Slovo w patrí do $L(T)$ práve vtedy, ak existuje akceptujúci výpočet T na w a nepatrí do $L(T)$ práve vtedy, ak žiadna konfigurácia, dosiahnuteľná výpočtom začínajúcom v počiatočnej konfigurácii s w na páske, nie je akceptujúca. Na tomto jednoduchom fakte je založený algoritmus, ktorý rozhoduje jazyk $L^C = L^C(T)$. Označme

KON množinu všetkých konfigurácií stroja T. Môžme predpokladať, že na tejto množine existuje usporiadanie (napr. lexikografické na reťazcoch odpovedajúcich konfiguráciám)

K(n) množinu všetkých konfigurácií, do ktorých sa T môže dostať pri spracovaní slova dĺžky n

D(w) množinu konfigurácií, dosiahnuteľných strojom T pri spracovávaní slova w

Pri konštrukcii stroja TC, ktorý rozhoduje komplement jazyka $L(T)$ sa zameriame na prehľadávanie množiny $D(w)$—platí $w \in L^C \iff D(w)$[7] neobsahuje akceptujúcu konfiguráciu. Problém sa teda zredukoval na prehľadávanie $D(w)$ v priestore $f(|w|)$. Označme **card(M)** mohutnosť množiny M

Predpokladajme, že $card(D(w))$ poznáme. Potom konštrukcia NTS, ktorý nielenže postupne vypíše všetky prvky z $D(w)$, ale aj správne odpovie na otázku $w \in L^C$? je jednoduchá:

1. card:=0
2. postupne generuj konfigurácie z $K(|w|)$. Pre každú vygenerovanú konfiguráciu k nedeterministicky uhádni, či $k \in D(w)$. Odpoveď áno prever nedeterministickou simuláciou pôvodného stroja T so vstupom w; pri dosiahnutí konfigurácie k je odpoveď potvrdená - $card \leftarrow card + 1$.
3. ak po skončení je $card = card(D(w))$, mali sme "v ruke" všetky dosiahnuteľné konfigurácie a môžme korektne odpovedať
   NIE - ak niektorá z konfigurácií v $D(w)$ bola akceptujúca, ÁNO v opačnom prípade

Vzhľadom k tomu, že nepotrebujeme všetky konfigurácie naraz, stačí nám k takému výpočtu priestor $f(|w|)$. Ostáva však **výpočet card(D(w))**.

- Nech $D_i(w)$ označuje množinu tých konfigurácií, ktoré sú strojom T dosiahnuteľné zo vstupného slova w maximálne v i krokoch.

[6] Robert bol v tom čase študentom informatiky na našej fakulte

[7] D(w) sa vzťahuje k pôvodnému stroju T pre jazyk L

- $D_0(w)$ obsahuje len počiatočnú konfiguráciu, preto poznáme $card(D_0(w))$, $Max(D_0(w))$.

- **výpočet card($\mathbf{D_{i+1}}$(w))**
  Zrejme $card(D_{i+1}(w)) = card(D_i(w)) + \Delta$, kde Δ je počet tých konfigurácií $k \notin D_i(w)$, pre ktoré existuje konfigurácia $k_0 \in D_i(w)$ taká, že $k_0 \rightarrow k$. Preto

  - vyššie popísaným spôsobom začne stroj vymenovávať všetky konfigurácie $k_0 \in D_i(w)$. Pri overení príslušnosti k $D_i(w)$ simuluje len i krokov výpočtu.
  - nech $k_0 \in D_i(w)$, $k_0 \rightarrow k$ a *count* je momentálna hodnota počítadla konfigurácií z $D_i(w)$. Ostáva overiť, či sa má k započítať do Δ. Odpoveď je áno, ak
    (a) $k \notin D_i(w)$
    (b) neexistuje konfigurácia $l \in D_i(w)$ *menšia* ako k_0 taká, že $l \rightarrow k$. Overenie spravíme opäť postupným vymenovávaním - tentokrát konfigurácií z $D_i(w)$, ktoré sú menšie ako k_0. Pre každú z nich overíme, či sa z nej v jednom kroku nedostaneme do k. Ak taká neexistuje a nám sa podarilo vymenovať všetky konfigurácie z $D_i(w)$ menšie ako k_0 (dopočítali sme sa do momentálnej hodnoty *count* počítadla pre porovnanie s hodnotou $card(D_i(w))$), tak konfiguráciu k "zaradíme" do $D_{i+1}(v)$, teda zvýšime hodnotu Δ, prípadne zmeníme hodnotu $Max(D_{i+1}(w))$

Nie je ťažké si uvedomiť, že popísanú konštrukciu môžme na viacpáskovom TS robiť v pamäti $f(|w|)$.

□

Kapitola 7

Trieda NP a NP-úplnosť

O význame triedy P ako triede prakticky riešiteľných problémov sme už hovorili. Videli sme, že polynomiálny čas je taká vlastnosť problémov, ktorá je nezávislá od výberu výpočtového modelu, pokiaľ sa hýbeme v prvej počítačovej triede. Triedu P rozhodne môžme považovať za triedu prakticky riešiteľných úloh. (Neskôr došlo k posunu – za prakticky riešiteľné považujeme aj pravdepodobnostné a aproximačné algoritmy polynomiálnej zložitosti.)

Je veľa takých úloh, pre ktoré nepoznáme deterministcké polynomiálne riešenie; všetky známe deterministické algoritmy sú exponenciálnej zložitosti, často založené na preberaní všetkých možností. Mnohé z nich sú však také, že ak *poznáme riešenie*, jeho správnosť vieme overiť v polynomiálnom čase. Konštrukcia nedeterministického stroja na riešenie tohto problému metódou "uhádni a over" tak v prípade, že riešenie je polynomiálne od veľkosti vstupu, vedie k nedeterministickému polynomiálnemu času. Dostali sme sa tak trochu inak k triede NP - sú to tie problémy, ktorých riešenie vieme overiť v čase polynomiálnom od veľkosti vstupu[1]. Z tohto pohľadu je otázka vzťahu $P?NP$ vlastne otázkou— je ľahšie nájsť riešenie alebo overiť jeho správnosť?

Nedeterminizmus priniesol do riešenia problémov novú kvalitu. Hoci pochopenie tohto konceptu nie je jednoduché, nedeterministické riešenia—algoritmy—sú často zrozumiteľnejšie, elegantnejšie. Vzhľadom na simulácie z predchádzajucej časti vieme, že prechod od nedeterminizmu k determinizmu nás môže stáť až exponenciálny nárast času. Je to nutné? Sú problémy riešiteľné v nedeterministicom polynomiálnom čase tak ťažké, že sa nedajú riešiť v polynomiálnom deterministickom čase? Otázka vzťahu tried P a NP patrí k tým najvýznamnejším. Väčšina informatikov predpokladá, že $P \neq NP$ a namiesto riešenia vzťahu týchto dvoch tried sa tak sústreďujeme skôr na to, ako riešiť tie problémy, ktoré sú ťažké.

Kedy hovoríme, že problém je ťažký? Napr. vtedy, ak nepatrí do P. Dokázať však o konkrétnom probléme, že nepatrí do triedy P, je nesmierne obtiažne, čo vlastne znamená, že takáto "definícia" pojmu ťažký je nám zbytočná. Uvidíme však, že existuje "nástroj", pomocou ktorého pre mnohé problémy pomerne ľahko ukážeme, že sa za predpokladu $P \neq NP$ v polynomiálnom deterministickom čase riešiť nedajú. Týmto nástrojom sú NP-úplné problémy. No a to, v situácii, keď akceptujeme $P \neq N$, poskytuje dobrú formuláciu toho, čo je ťažké.

K čomu je to dobré? Ak ukážeme, že problém je NP-úplný, je ťažký a teda máme dobrý dôvod sa domnievať, že sa nedá riešiť v P. No a keď to nejde deterministicky polynomiálne, treba z niečoho, konkrétne nejakých požiadaviek na riešenie, zľaviť.

[1] Uvedomme si, že tým implicitne hovoríme, že riešenie je polynomiálne vzhľadom na veľkosť vstupu.

Dostávame sa tak k *aproximačným* algoritmom (v prípade optimalizačných úloh upúšťame od požiadavky na optimálne riešenie a uspokojíme sa s približným, ktoré je rýchlo a vieme ohraničiť, ako ďaleko od optimálneho), *pravdepodobnostným* algoritmom (upúšťame od toho, že riešenie je vždy správne, vyžadujeme však, aby bolo rýchlo a aby pravdepodobnosť korektnej odpovede bola vysoká; väčšinou vyššia ako 0,5), ***heuristickým algoritmom*** (algoritmus sleduje nejakú rozumnú stratégiu, ktorá však negarantuje, že dostaneme korektné riešenie; môže sa stať, že riešenie vôbec nedostaneme napriek tomu, že existuje.).

7.1 Redukcie, NP-úplnosť

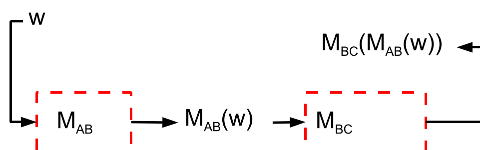
Aby sme mohli hovoriť o ťažších a ľahších problémoch, potrebujeme mať možnosť ich porovnávať. V snahe o vymedzenie najťažších problémov v triede NP začneme definíciami redukcií, ktoré umožňujú obtiažnosť problémov porovnávať.

polynomiálna redukcia $L \rightsquigarrow L'$

Definícia 7.1 *Jazyk L sa nazýva* polynomiálne redukovateľný *na jazyk L', ak existuje polynóm p a $p(n)$-časovo ohraničený DTS M taký, že M prepíše každý vstupný reťazec $w \in \Sigma_L^*$ na reťazec $w' \in \Sigma_{L'}^*$ tak, že $w \in L \Leftrightarrow M(w) = w' \in L'$.*

tranzitivita $\rightsquigarrow$

Ľahko vidno, že relácia polynomiálnej redukcie je tranzitívna: Nech $A \rightsquigarrow B$ a $B \rightsquigarrow C$, M_{AB}, M_{BC} sú DTS počítajúce príslušné redukcie a p_{AB}, p_{BC} sú polynómy ohraničujúce ich časovú zložitosť. Nech α, β je vstup do M_{AB}, resp. M_{BC}. Potom $|M_{AB}(\alpha)| \leq p_{AB}(\alpha)$; analogicky $|M_{BC}(\beta)| \leq p_{BC}(\beta)$.
Redukciu $A \rightsquigarrow C$ ľahko realizujeme napr. zreťazením strojov M_{AB}, M_{BC}: $M_{AC}(w) = M_{BC}(M_AB(w))$.



Pre čas $t(w)$ výpočtu redukcie $M_{AC}(w)$ zrejme platí

$$t(w) \leq p_{AB}|w| + p_{BC}(M_{AB}(w)) \leq p_{AB}|w| + p_{BC}(|p_{AB}(|w|)|)$$

Keďže skladaním polynómu dostávame opäť polynóm, existuje polynóm p_{AC} taký, že

$$t(w) \leq p_{AC}(|w|)$$

A teraz už môžme definovať NP-úplnosť.

NP-ťažký, NP-úplný jazyk

Definícia 7.2 *Jazyk L_0 sa nazýva* NP-ťažký,*ak $\forall L \in NP$: L je polynomiálne redukovateľný na L_0. Ak navyše $L_0 \in NP$ tak je jazyk* NP-úplný*.*
Triedu NP-úplných problémov budeme označovať NPU.

O význame NP-úplných problémov hovorí nasledujúca veta, najmä podmienka 2. Jej dôkaz prenechávame čitateľovi.

Veta 7.1

1. *Ak $NPU \cap P \neq \emptyset$, tak $P = NP$*
2. *Ak $P \neq NP$, tak $NPU \subset NP \setminus P$*

Zo znenia vety vidno, že ak chceme ukázať rovnosť tried $P = NP$, stačí pre jediný NP-úplný problém nájsť deterministický polynomiálny algoritmus.

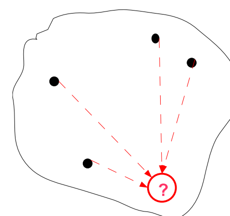
Naopak – ak prijmeme hypotézu , že $P \neq NP$, je dôkaz NP-úplnosti dôkazom toho, že problém je z $NP \setminus P$. Takto nám pojem NP-úplnosti poskytuje možnosť alternatívneho "dôkazu" toho, že problém nie je prakticky riešiteľný.

Existuje aj iná definícia NP-úplnosti, s ktorou sa stretávame v prípade problémov, ktoré nie sú rozhodovacie.

Definícia 7.3 *Problém L_0 z NP je NP-úplný, ak z existencie algoritmu polynomiálnej zložitosti $T(n)$ riešiaceho L_0 vyplýva existencia algoritmu zložitosti $p_L(T(n))$ pre každý problém $L \in NP$, pričom p_L je polynóm, ktorý závisí od L.*

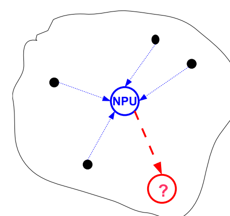
Pri dokazovaní NP-úplnosti robí problém najmä dôkaz toho, že problém je NP-ťažký. Tu využijeme tranzitivitu relácie "byť polynomiálne redukovateľný".

NP-úplnosť redukciou



Nech problém $L_0 \in NP$ je NP-úplný (na obr. NPU). Potom ho môžme využiť pri dôkaze NP-úplnosti problému L (na obr. ?) nasledovne:

1. ukážeme, že $L \in NP$
2. ukážeme, že L_0 je polynomiálne transformovateľný na L; toto dokazuje polynomiálnu redukovateľnosť ľubovoľného L' na L (prečo?)



K použitiu tohto prístupu však potrebujeme nejaký problém, o ktorom už vieme, že je NP-úplný. NP-úplnosť prvého problému musíme dokázať podľa definície.

7.2 NP-úplnosť problému splniteľnosti BF

Hlavným výsledkom tejto časti je Cookova veta, ktorá o probléme splniteľnosti boolovskej formuly dokáže, že je NP-úplný.

Definícia 7.4 Boolovskou formulou *(BF) nazveme výrazy*

- $x, \neg x$
- *ak p, q sú boolovské formuly, tak aj $(p \vee q), (p \wedge q), \neg p$ sú boolovské formuly*
- *iné boolovské formuly nie sú*

Nech BF je Boolovská formula n premenných $x_1, \dots, x_n$; označujeme $BF(x_1, \dots, x_n)$. Hovoríme, že BF je splniteľná*, ak existuje také priradenie hodnôt $0, 1$ premenným $x_1, \dots, x_n$, pri ktorom BF nadobúda hodnotu 1*

Problém splniteľnosti BF

Vstup: Boolovská formula BF, resp. reťazec, ktorý ju kóduje
Výstup: 1, ak je formula splniteľná
0 inak

Podľa potreby a kontextu bude SAT označovať alebo problém splniteľnosti BF alebo triedu splniteľných BF.

Veta 7.2 (Cook) *Problém splniteľnosti BF je NP-úplný*

Dôkaz: Príslušnosť problému splniteľnosti do NP ukážeme konštrukciou NTS, ktorý ho rieši.

- nech vstupom je $BF(x_1, \ldots, x_n)$; nech m označuje dĺžku vstupu
- DTS uhádne priradenie hodnôt $\alpha_1, \ldots, \alpha_n$ premenným $x_1, \ldots, x_n$ (prechádzaním po vstupnej páske si na pracovnú pásku zapisuje premenné a im uhádnuté hodnoty; časová zložitosť $O(m^2)$)
- dosadí uhádnuté priradenie hodnôt $\alpha_1, \ldots, \alpha_n$ do BF (prechádzaním po vstupnej páske opisuje na druhú pracovnú pásku BF zo vstupu, pričom namiesto jednotlivých premenných píše hodnoty; časová zložitosť $O(m^2)$)
- vyhodnotením BF s dosadenými hodnotami overí, či $BF(\alpha_1, \ldots, \alpha_n) = 1$; časová zložitosť $O(m^2)$

$L \in NP \Rightarrow$ L **je polynomiálne redukovateľný na SAT** K dôkazu tejto časti treba skonštruovať DTS polynomiálnej zložitosti, ktorý pri vstupe w v čase polynomiálnom skonštruuje BF tak, že $w \in L \Leftrightarrow BF \in SAT$. Ak pri konštrukcii DTS využijeme len tie vlastnosti/znalosti o jazyku L, ktoré sú spoločné všetkým jazykom z NP, bude táto konštrukcia platná pre všetky jazyky z NP.

Ak $L \in NP$, potom existuje NTS M rozpoznávajúci L; $L = L(M)$. O NTS M môžme bez ujmy na všeobecnosti predpokladať:

- $NTSM$ je jednopáskový
- vstupná abeceda $\Sigma = \{0, 1\}$
- abeceda symbolov $\Gamma = \{1, 2, \ldots, s\}$; pritom predpokladáme, že (napr) 1 označuje B (blank)
- množina stavov je $K = \{1, 2, \ldots, q\}$; nech 1 je počiatočný stav, q koncový akceptujúci
- časová zložitosť stroja M je zhora ohraničená polynómom $p(n)$

O výpočte $C = C_1, C_2, \ldots, C_T$ NTS M môžme bez ujmy na všeobecnosti predpokladať, že

- je dĺžky *presne* $p(n)$
- každá konfigurácia je veľkosti $p(n)$ - uvažujeme teda aj prípadne "blanky"

Booleovská formula BF_w vytvorená DTS k vstupnému slovu w (pri znalosti popisu NTS M) bude používať tri skupiny premených. Ak zafixujeme nejaký konkrétny výpočet NTS M $C = C_1, C_2, \ldots, C_{p(n)}$, možno sa na tieto premenné pozerať ako na predikáty.

$C(i, j, t)$,	$1 \leq i \leq p(n), 1 \leq j \leq s,$ $0 \leq t \leq p(n)$	?je pravda, že vo výpočte C je v čase t na i-tom políčku symbol j?
$S(k, t)$,	$1 \leq k \leq q, \leq t \leq p(n)$	?je pravda, že vo výpočte C je v čase t stroj v stave k?
$H(i, t)$,	$1 \leq i \leq p(n), 0 \leq t \leq p(n)$	?je pravda, že vo výpočte C je v čase t hlava na i-tom políčku?

Máme tak $O(p^2(n))$ premenných, na zápis každej z nich stačí priestor $O(\log n)$.

Nech $Q_0, Q_1, \ldots, Q_{p(n)}$ je popis postupnosti konfigurácií. Tento popis je korektným *popisom akceptujúceho výpočtu*, ak platí nasledujúcich 7 podmienok:

1. Q_0 je počiatočná konfigurácia
2. V každom čase je hlava na práve jednom políčku
3. V každom čase je stroj práve v jednom stave
4. V každom čase je na každom políčku pásky práve jeden symbol
5. Mení sa len to políčko, na ktorom je nastavená hlava
6. Zmena sa deje podľa prechodovej funkcie
7. Posledná konfigurácia je akceptujúca

Teraz popíšeme výslednú formulu BF_w. Táto vznikne ako logický súčin formúl ABCDEFG. Pri konštrukcii týchto podformulí využijeme predikát

$$U(y_1, y_2, \ldots, y_n) = (y_1 \vee y_2 \vee \ldots \vee y_r) \cdot \prod_{i \neq j} (\neg y_i \vee \neg y_j)$$

ktorý nadobúda hodnotu 1 práve vtedy, keď *práve jedna* z premenných $y_1, \ldots, y_r$ nadobúda hodnotu 1. Uvedomme si, že zápis $U(y_1, \ldots, y_r)$ je len označenie formuly, ktorá používa $O(r^2)$ premenných a symbolov.

K 1. $\mathbf{Q_0}$ je počiatočná konfigurácia

$$A = S(1,0) \wedge H(1,0) \wedge \prod_{1 \leq i \leq n} C(i, w_i, 0) \wedge \prod_{n < i \leq p(n)} C(i, 1, 0)$$

K 2. V každom čase je hlava na práve jednom políčku Zafixujme čas t. Potom

$$B_t = U(H(1,t), \ldots, H(p(n),t))$$

Keďže podmienka má platiť v každom časovom okamihu,

$$B = \prod_{0 \leq t \leq p(n)} B_t$$

K 3. V každom čase je stroj práve v jednom stave

$$C = \prod_{0 \leq t \leq p(n)} U(S(1,t), \ldots, S(q,t))$$

K 4. V každom čase je na každom políčku pásky práve jeden symbol Zafixujme najprv čas t a políčko i

$$D_{i,t} = U(C(i,1,t), \ldots, C(i,s,t))$$

Podmienka má platiť v každom časovom okamihu a na každom políčku, preto

$$D = \prod_{\substack{1 \leq p(n) \\ 0 \leq t \leq p(n)}} D_{i,t}$$

K 5. Mení sa len to políčko, na ktorom je nastavená hlava
To znamená, že alebo je obsah políčka v dvoch po sebe idúcich časových okamihoch rovnaký, alebo na ňom bola nastavená hlava. Zafixujme najprv čas, políčko aj symbol

$$E_{i,j,t} = (C(i,j,t) \equiv C(i,j,t+1)) \vee H(i,t)$$

Podmienka má platiť v každom čase (pre každé políčko a symbol)

$$E = \prod_{\substack{1 \le i \le p(n) \\ 0 \le t \le p(n) \\ 1 \le j \le s}} E_{i,j,t}$$

K 6. Zmena sa deje podľa prechodovej funkcie
Zafixujme čas t, políčko i, symbol j a stav k. Tieto štyri hodnoty spolu alebo súvisia – v čase t je stroj v stave k, hlavu má na políčku i a na tomto políčku je symbol j alebo spolu nesúvisia. Nech

$$\delta(j,k) = \{(j_1, k_1, pos_1), \dots (j_{m(j,k)}, k_{m(j,k)}, pos_{m(j,k)})\}$$

Potom

$$F_{i,j,k,t} = \neg C(i,j,t) \vee \neg S(k,t) \vee \neg H(i,t) \vee \sum_{1 \le l \le m(j,k)} C(i,j_l,t{+}1) \wedge H(i{+}pos_l,t{+}1) \wedge S(k_l,t{+}1)$$

Podmienka platí pre všetky hodnoty i, j, k, t

$$F = \prod_{\substack{1 \le i \le p(n) \\ 0 \le t \le p(n) \\ 1 \le j \le s,\ 1 \le k \le q}} F_{i,j,k,t}$$

K 7. Posledná konfigurácia je akceptujúca
Pri platnosti predchádzajúcich podmienok to znamená, že psledný stav má byť akceptujúci

$$G = S(q, p(n))$$

Výstupom DTS je teda výsledná formula $BF_w = A \wedge B \wedge C \wedge D \wedge E \wedge F \wedge G$. Vzhľadom k tvaru podformúl A, B, C, D, E, F, G je zrejmé, že časová zložitosť DTS je polynomiálna vzhľadom na veľkosť výslednej BF_w. Sústredíme sa preto len na veľkosť BF_w.

A	$O(p(n) \log n)$
B	$O(p^3(n) \log n)$
C	$O(q^2 p(n) \log n)$
D	$O(s^2 p(n) \log n)$
E	$O(p^2(n) s \cdot \log n)$
F	$O(p^2(n) s \cdot q \cdot M \cdot \log n)$, kde $M = \max\{m(i,j)\}$
G	$O(\log n)$

Výsledná veľkosť BF_w je $O(p^3(n) \log n)$. Ešte ostáva ukázať, že $w \in L \Leftrightarrow BF_w \in SAT$.

$$w \in L$$

$$\Updownarrow$$

$$\exists \text{ akceptujúci výpočet } Q = Q_0, Q_1, \dots, Q_{p(n)}$$

$$\Updownarrow$$

priradenie hodnôt premenným $C(i,j,t), S(k,t), H(i,t),$ podľa predpisu

$C(i,j,t) = 1 \Leftrightarrow$ je pravda, že vo výpočte Q je na i-tom políčku v čase t symbol j

$S(k,t) = 1 \Leftrightarrow$ je pravda, že vo výpočte Q je v čase t stroj v stave q

$H(i,t) = 1 \Leftrightarrow$ je pravda, že vo výpočte Q je v čase t hlava na i-tom políčku

zabezpečuje, že $BF_w(C(i,j,t), S(k,t), H(i,t)) = 1$

□

Keď že vytvorenú formulu BF_w vieme v polynomiálnom čase previesť na formulu v tvare KNF, dostávame nasledujúci dôsledok.

Dôsledok 7.3 *Problém splniteľnosti formuly v tvare KNF je z NP.*

7.3 Niektoré NP-úplné problémy

Keď už máme NP-úplný problém, metódou redukcie ukážeme NP-úplnosť aj o niektorých ďalších.

Problém 3 splniteľnosti

Vstup: formula $3BF$ v tvare $3KNF = F_1 \wedge F_2 \wedge \ldots \wedge F_q,\ F_i = x_{i_1}^{\sigma i_1} \vee x_{i_2}^{\sigma_{i_2}} \vee x_{i_3}^{\sigma_{i_3}}$
Výstup: $1 \Leftrightarrow 3BF \in SAT$

Problém k-kliky

Vstup: $G = (V,E),\ k \in \mathbb{N}$
Výstup: $1 \Leftrightarrow G$ obsahuje úplný podgraf o k vrcholoch

Problém k-pokrytia

Vstup: $G = (V,E),\ k \in \mathbb{N}$
Výstup: $1 \Leftrightarrow$ ak existuje množina vrcholov $S,\ S \subseteq V$, mohutnosti k tak, že $\forall (u,v) \in E : (u \in S$ alebo $v \in S)$

Problém k-zafarbiteľnosti

Vstup: $G = (V,E),\ k \in \mathbb{N}$
Výstup: 1 ak množinu vrcholov V vieme zafarbiť k farbami tak, že žiadna hrana nemá oba konce rovnakej farby.

Problém HK

Vstup: $G = (V,E)$
Výstup: $1 \Leftrightarrow G$ obsahuje hamiltonovskú kružnicu (jednoduchý cyklus prechádzajúci cez každý vrchol grafu)

Veta 7.4 *Problém 3-splniteľnosti je NPÚ.*

Dôkaz: Tvrdenie dokazujeme redukciou z problému splniteľnosti formuly v tvare KNF.
3-Splniteľnosť $\in NP$ vlastne netreba dokazovať, pretože všebecný problém splniteľnosti je z NP.

Polynomiálna redukcia KNF splniteľnosti na 3 splniteľnosť: Nech KNF je formula v tvare KNF. Bez ujmy na všeobecnosti môžme predpokladať, že je to formula tvaru $(x_1 \vee \ldots \vee x_n)$. Uvažujme formulu

$$3BF = (x_1 \vee x_2 \vee y_1) \wedge (\overline{y_1} \vee x_3 \vee y_2) \wedge \ldots \wedge (\overline{y_{i-2}} \vee x_i \vee y_{i-1}) \wedge \ldots \wedge (\overline{y_{n-3}} \vee x_{n-1} \vee x_n)$$

kde y_j sú nové premenné. Túto formulu zrejme na TS vytvoríme v čase polynomiálnom od veľkosti pôvodnej formuly BF.

BF ⟶ 3BF

BF $\in$ ***SAT*** Nech $\alpha = (\alpha_1, \ldots, \alpha_n)$ je spĺňajúci vektor hodnôt, na ktorom $BF(\alpha) = 1$. Potom existuje také i, že v α je $x_i = 1$. Definujme vektor hodnôt β premenných $y_1, \ldots, y_{n-3}$ $\begin{cases} y_j = 1 & \text{pre } j \le i-2 \\ y_j = 0 & \text{pre } i-2 < j \le n-3, \end{cases}$

Potom $3BF(\alpha, \beta) = 1$.

3BF⟶BF

3*BF* $\in$ ***SAT*** Nech $\alpha = (\alpha_1, \ldots, \alpha_n)$ je vektor hodnôt, na ktorom $3BF(\alpha) = 1$. Stačí ukázať, že existuje i také, že pre $\alpha = \alpha_1, \ldots, \alpha_n,\ i = 1$. Postupujme sporom. Nech by $x_i = 0$ pre každé i. Keďže $3BF(\alpha) = 1$, musí platiť:

$$y_{n-3} = 0 \Rightarrow y_{n-2} = 0 \Rightarrow \ldots \Rightarrow y_1 = 0$$

To ale znamená, že $(x_1 \vee x_2 \vee y_1) = 0$, čo je spor s predpokladom, že $3BF(\alpha) = 1$.

všeobecný prípad

V prípade, keď formula F nie je $(x_1 \vee \ldots \vee x_n)$ ale $(x_1^{\sigma_1} \vee \ldots \vee x_n^{\sigma_n})$, zameníme v konštruovanej 3BF výskyt x_i za $x_i^{\sigma_i}$.
Ak F je konjunkcia viacerých elementárnych disjunkcií, každá konjunkcia pracuje s novými pomocnými premennými.

dorob

□

Veta 7.5 *Problém k-kliky je NPÚ.*

Dôkaz: Tvrdenie dokazujeme redukciou z problému splniteľnosti formuly v tvare KNF. Časť **problém k-kliky je z NP** prenechávame čitateľovi.

Redukcia spočíva v dvoch krokoch - k vstupnej formule $F = F_1 \wedge F_2 \wedge \ldots \wedge F_q$ zostrojíme v polynomiálnom čase graf $G = (V, E)$, o ktorom dokážeme, že obsahuje q-kliku práve vtedy, keď formula F je splniteľná.

Konštrukcia grafu G Nech $F = F_1 \wedge F_2 \wedge \ldots \wedge F_q,\ F_i = x_{i1} \vee x_{i2} \vee \ldots \vee x_{ik(i)}$.

Množinu vrcholov vytvoríme tak, že ku každému výskytu literálu vo formule F vytvoríme jeden vrchol. Hranu medzi dva vrcholy pridáme vtedy, ak príslušné literály *môžu* mať v priradení hodnôt rovnakú hodnotu. Inými slovami nepridáme hranu medzi také vrcholy, ktorých literály *nemôžu* mať rovnakú hodnotu (x a $\neg x$ nemôžu mať rovnakú hodnotu).

Formálnejšie: množinu vrcholov V tvoria dvojice

$$V = \{[i,j] \mid 1 \leq i \leq q, 1 \leq j \leq k(i)\}$$

Prvá zložka vrchola $[i,j]$ ukazuje na i-tu elementárnu disjunkciu formuly F, druhá zložka na literál na j-tej pozícii v nej.

Množina hrán E je tvorená množinou dvojíc vrcholov

$$E = \{([i,j],[k,l]) \mid i \neq k, x_{ij} \neq \neg x_{kl}\}$$

Ak literály odpovedajúce hranou spojeným vrcholom obsahujú rovnakú premennú, potom sú tieto literály rovnaké:

$$(([i,j],[k,l]) \in E, x_{ij} = x_m^{\sigma_m}, x_{kl} = x_t^{\sigma_t} \wedge m = t) \Rightarrow (\sigma_m = \sigma_t)$$

Je zrejmé, že tento popis vieme na DTS vytvoriť v polynomiálnom čase.

F je splniteľná Nech formula F je splniteľná. Potom existuje súbor hodnôt α taký, že $F(\alpha) = 1$. Nech x_{i,m_i} je ten literál elementárnej disjunkcie F_i, ktorý sa vyhodnocuje na 1. Tvrdíme, že množina vrcholov $\{[i,m_i], 1 \leq i \leq q\}$ tvorí kliku v grafe G. K dôkazu stačí overiť, že $\forall i \neq j$ tvorí dvojica vrcholov $[i,m_i],[j,m_j]$ hranu v grafe G. Nech by existovali také $i \neq j$, že $([i,m_i],[j,m_j]) \notin E$. Keďže $i \neq j$, znamená to, že $x_{i,m_i} = \neg x_{j,m_j}$. To je ale v spore s faktom, že $x_{i,m_i} = x_{j,m_j} = 1$

$F \longrightarrow G$

G obsahuje q-kliku Nech q-kliku tvorí množina vrcholov $\{[i,m_i] 1 \neq i \neq q\}$. Vytvorme dve množiny premenných

$S1 = \{y \mid y = x_{i,m_i}, y \text{ je premenná}, 1 \leq i \leq q\}$
$S0 = \{y \mid y = \neg x_{i,m_i}, y \text{ je premenná}, 1 \leq i \leq q\}$

Sporom ukážeme, že $S1 \cap S0 = \emptyset$. Ak by $\exists i \neq j$ také, že $x_{i,m_i} = y = \neg x_{j,m_j}$, potom by platilo, že $([i,m_i],[j,m_j]) \notin E$.
Potom pre vektor hodnôt α, ktorý vznikne tak, že $y = h$ ak $y \in Sh,\ h = 0,1$ platí: $F(\alpha) = 1$

$G \longrightarrow F$

□

Cvičenie 7.1 *Redukciou z problému k-kliky dokážte, že problém k-pokrytia je NPÚ.*

Veta 7.6 *Problém k-zafarbiteľnosti je NPÚ.*

Dôkaz: Tvrdenie dokazujeme redukciou z problému 3splniteľnosti. To, že problém k-zafarbiteľnosti patrí do NP prenechávame čitateľovi.

Redukciu spravíme v dvoch krokoch - k danej vstupnej formule $F = F_1 \wedge \ldots \wedge F_t$ nad premennými $x_1, \ldots, x_n$ zostrojíme graf $G = (V, E)$, o ktorom ukážeme, že F je splniteľná práve vtedy keď G je $(n+1)$-zafarbiteľný.

Graf G budeme konštruovať tak, aby sme vynútili použitie n farieb (úplný graf na n vrcholoch) a aby stačila jedna ďalšia farba práve vtedy, ak je formula splniteľná. Farbenie touto novou farbou bude odpovedať priradeniu hodnoty 0 príslušnej premennej.

konštrukcia G

$V = \{x_i, \neg x_i, v_i, 1 \leq i \leq n\} \cup \{F_i, 1 \leq i \leq t\}$

$$\begin{aligned} E = \ & \{(v_i, v_j)\ i \neq j\} \cup \\ & \{(v_i, x_j), (v_i, \neg x_j)\ i \neq j\} \cup \\ & \{(x_i, \neg x_i)\} \cup \\ & \{(x_i, F_j) \mid x_i \text{ nie je termom } F_j\} \cup \\ & \{(\neg x_i, F_j) \mid \neg x_i \text{ nie je termom } F_j\} \end{aligned}$$

Všimnime si, že vrcholy $v_1, \dots, v_n$ tvoria úplný podgraf, preto na ich zafarbenie potrebujeme n farieb. Nech teda v_i je zafarbené i-tou farbou.
Keďže oba vrcholy $x_i, \neg x_i$ sú spojené s každým $v_j, j \neq i$, môžme pri farbení vrcholov $x_i, \neg x_i$ používať i-tu farbu. Navyše je ale každý vrchol x_i spojený s vrcholom $\neg x_i$, preto potrebujeme novú farbu – nazvime ju s. Takže z dvojice $x_i, \neg x_i$ je jeden vrchol zafarbený i-tou farbou a druhý farbou s. Uvedomme si, že ak chceme graf farbiť minimálnym počtom farieb, je popísané farbenie "jednoznačné".
Ostáva farbenie vrcholov/termov. Čo vieme o F_j? F_j je elementárna disjunkcia troch termov. Ak teda $n \geq 4$, pre každé j existuje $i(j)$ také, že F_j je spojený aj s $x_{i(j)}$ aj s $\neg x_{i(j)}$. To vylučuje použitie farby s na farbenie vrchola F_j.

Ukážeme, že **pri farbení vrcholov F_j nepotrebujeme ďalšiu farbu práve vtedy, ak F je splniteľná formula.**
Nech F_j obsahuje literál y, pričom $\neg y$ je zafarbený farbou s. Potom F_j nie je spojený so žiadnym iným vrcholom, ktorý má rovnakú farbu ako y (y má i-tu farbu - rovnako ako v_i, ktoré je spojené s každým ďalším x_r, resp. $\neg x_r$. Preto žiaden z vrcholov, s ktorými je spojený F_j, nemôže mať farbu ako v_i). V takomto prípade môžeme F_j zafarbiť rovnakou farbou ako y. Ak budeme zafarbenie farbou s považovať za priradenie hodnoty 0 a zafarbenie farbou $1, 2, \dots, n$ za priradenie hodnoty 1, dostávame

F_j môže byť zafarbené bez použitia ďalšej farby

$\Updownarrow$

$\exists$ priradenie $(n+1)$ farieb literálom tak, že $\forall$ elementárna disjunkcia obsahuje literál y taký, že $\neg y$ má priradenú farbu s

$\Updownarrow$

ak možno priradiť hodnoty premenným tak, že $\forall$ elementárna disjunkcia obsahuje $y = 1$ ($\neg y$ má farbu s, $\neg y = 0$)

$\Updownarrow$

F je splniteľná

□

Veta 7.7 *Problém HK je NPÚ*

Dôkaz: Tvrdenie budeme opäť dokazovať redukciou - tentokrát z problému vrcholového pokrytia. Príslušnosť problému HK do NP nechávame čitateľovi ako cvičenie.

Redukciu problému spravíme v dvoch krokoch – k vstupnému neorientovanému grafu $G = (V, E)$ a prirodzenému číslu k zostrojíme orientovaný graf $G_D = (V_D, E_D)$, o ktorom ukážeme, že G_D má HK práve vtedy, keď G má vrcholové pokrytie mohutnosti k.

konštrukcia G_D

$\mathbf{V_D} = \{a_1, a_2, \dots, a_k\} \cup \{[v, e, b] \mid v \in V, e \in E, b \in \{0, 1\}, v$ incidentný s $e\}$

$\mathbf{E_D}$: Ku každému vrcholu $v_i \in V$ môžme priradiť usporiadaný zoznam hrán $f_{i1}, \dots, f_{ip(i)}$ s ním incidentných; pre vrchol v_i nech $p(i)$ označuje počet hrán incidentných s v_i. Potom do E_D pridávame hrany

$\forall j, i \quad a_j \to [v_i, e_{i1}, 0],$ $\qquad e_{i1}$ je prvá na zozname pre v_i

$\qquad [v_i, e_{ip(i)}, 1] \to a_j,$ $\qquad e_{ip(i)}$ je posledná na zozname pre v_i

Každá hrana $(v_i, v_j) \in E$ spôsobí vznik podgrafu o 4 vrcholoch

$$\begin{array}{ccc} [v_i, e_{ij}, 0] & \rightleftarrows & [v_j, e_{ji}, 0] \\ \downarrow & & \downarrow \\ [v_i, e_{ij}, 1] & \rightleftarrows & [v_j, e_{ji}, 1] \end{array} \qquad \begin{array}{ccc} \mathrm{A} & \rightleftarrows & \mathrm{B} \\ \downarrow & & \downarrow \\ \mathrm{C} & \rightleftarrows & \mathrm{D} \end{array}$$

pokrytie $\longrightarrow$ *HK*

Nech $v_1, \ldots, v_k$ tvoria pokrytie v G. Ukážeme, že v grafe G_D existuje HK. Uvažujme najprv kružnicu

$$\begin{array}{lll} K = & a_1, & [v_1, f_{11}, 0], [v1, f_{11}, 1], \ldots, [v_1, f_{1p(1)}, 0], [v_1, f_{1p(1)}, 1], \\ & a_2, & [v_2, f_{21}, 0], \ldots, , [v_2, f_{2p(2)}, 1], \\ & \ldots & \\ & a_k, & [v_k, f_{k1}, 0], \ldots, [v_k, f_{kp(k)}, 1], a_1 \end{array}$$

Ak ostali nejaké nezaradené vrcholy $[v_j, e_{ij}, 0], [v_j, e)ij, 1]$, tak vrchol v_i je z pokrytia (prečo?). Vtedy modifikujeme K tak, že úsek

$$[v_i, e_{ij}, 0], [v_i, e_{ij}, 1]$$

nahradíme

$$[v_i, e_{ij}, 0], [v_j, e_{ij}, 0], [v_j, e_{ij}, 1], [v_i, e_{ij}, 1]$$

Týmto spôsobom zaradíme všetky doteraz nezaradené vrcholy a na záver bude K tvoriť HK.

HK $\longrightarrow$ *pokrytie*

Nech v G_D existuje HK K. Vrcholy $a_1, \ldots, a_k$ rozdeľujú K na k úsekov (kvôli jednoduchosti predpokladajme, že $a_1, \ldots, a_k$ je to poradie, v ktorom sa tieto vrcholy vyskytujú na K). Vzhľadom k tomu, že ak vojdeme do vrchola A (obr. vyššie) je prechod štvoricou vrcholov A, B, C, D v HK možný jedine dvomi spôsobmi - $ABCD$ alebo AD, definuje každý úsek medzi a_i a a_{i+1} vlastne jeden vrchol z pokrytia; každému takémuto úseku možno priradiť vrchol $v_t \in V$ taký, že každý vrchol z tohto úseku je tvaru $[v_t, e_{tj}, 0], [v_t, e_{tj}, 1]$ alebo $[v_i, e_{tj}, 0], [v_i, e_{tj}, 1]$. Týmto spôsobom HK K definuje v grafe G vrcholové pokrytie mohutnosti nanajvýš k.

□

7.4 P vs. NP*

Povedali sme, že na triedu NP sa možno pozerať ako na triedu problémov, pre ktoré vieme v polynomiálnom čase overiť, že dané riešenie je správnym riešením našej úlohy. Táto alternatívna charakterizácia je formálne zachytená prostredníctvom polynomiálne rozhodnuteľnej relácie.

Definícia 7.5 *Reláciu $R \subseteq \Sigma^* \times \Sigma^*$ nazveme* polynomiálne rozhodnuteľnou*, ak existuje TS rozhodujúci jazyk $L = \{(x, y) \mid (x, y) \in R\}$ v polynomiálnom čase. R nazveme* polynomiálne vyváženou*, ak $(x, y) \in R$ implikuje $|y| < |x^k|$ pre nejaké $k \geq 1$.*

Lema 7.8 *Nech $L \subseteq \Sigma^*$. Potom $L \in NP \Leftrightarrow$ ak existuje polynomiálne rozhodnuteľná a polynomiálne vyvážená relácia R taká, že $L = \{x \mid \exists y : (x, y) \in R\}$*

$\Longleftarrow$

Dôkaz: Ak máme spomínanú polynomiálne rozhodnuteľnú a polynomiálne vyváženú reláciu R, môže NTS rozhodujúci jazyk L pracovať nasledovne:

– uhádne y

– overí, že $(x, y) \in R$

Nech $p(n) = n^k$ je časovou zložitosťou TS, ktorý rozhoduje L. Potom definujeme požadovanú reláciu R nasledovne: $(x, y) \in R \quad \Leftrightarrow \quad y$ je kód akceptujúceho výpočtu na x. $\Longrightarrow$

□

Máme teda iný pohľad na triedu NP - jazyk $L \in NP$ práve vtedy, ak $\forall x \in L$ existuje krátky certifikát, dosvedčujúci, že x je z L. Navyše sa ten certifikát dá ľahko nájsť. Takže otázka vzťahu $P?NP$ môže byť aj otázkou, či je rovnako ťažké nájsť dôkaz alebo overiť dôkaz.

7.4.1 Vzťah tried P, NP

Z hľadiska zložitosti máme tri dôležité skupiny problémov - triedy P, NP, NPU. Aký je vzťah týchto tried? Potenciálne máme tri možnosti

1. $P = NP$
2. $P \neq NP$ a $NPU = NP/P$
3. $P \neq NP$ a $NPU \neq NP/P$

Ukážeme, že možnosť 2. nepripadá do úvahy.

Veta 7.9 *Ak $P \neq NP$, tak existuje jazyk $L \in NP$ taký, že $L \notin P$ a $L \notin NPU$.*

Dôkaz:

- Predpokladajme, že $M_1, M_2, \ldots$ je efektívne očíslovanie polynomiálne ohraničených TS[2].
- Analogicky nech $R_1, R_2, \ldots$ je efektívne očíslovanie polynomiálne ohraničených strojov počítajúcich redukcie.
- Nech S je deterministický stroj rozhodujúci SAT (zrejme nie je polynomiálnej zložitosti, keďže predpokladáme $P \neq NP$)

A teraz už môžme popísať požadovaný jazyk $L \in NP/\{P \cup NPU\}$. Popíšeme ho definovaním stroja K, ktorý ho rozhoduje.
if $S(x) =$ 'áno' a $f(|x|)$ je párne, then $K(x)$='áno'
else $K(x)$='nie'
Jadrom je definícia funkcie f. Túto tiež budeme definovať popisom stroja F, ktorý ju počíta. Pritom F počíta $f(n)$ ak na vstupe dostane n ako 1^n. Pri výpočte $f(n)$ pracuje stroj F v dvoch etapách, pričom v každej etape spraví presne n krokov.

1. F počíta postupne hodnoty $f(0), f(1), \ldots$. Nech posledná dopočítaná hodnota je $f(j) = k$. Potom hodnota $f(n)$ bude alebo k alebo $k+1$, a to podľa výsledku výpočtu v druhej etape.

2. Rozlišujeme dva prípady
k = 2i F postupne simuluje $M_i(z), S(z), F(|z|)$, kde z sú binárne reťazce dosadzované postupne zo $\{0,1\}* = 0, 1, 00, 01, 10, 11, \ldots$. Pritom sa snaží **nájsť také** z, **aby** $\mathbf{K(z) \neq M_i(z)}$. Uvedomme si, že hľadáme z, pre ktoré

[2] $M_{\langle i,j\rangle}$ označuje stroj, ktorý simuluje $|w|^j$ krokov (v štandardnom číslovaní) $i-$teho TS. Pritom párovacia funkcia $\langle i, j\rangle$ je definovaná takto $\langle i, j\rangle = 1 + 2 + \ldots + (i + j) + j$

$$(*)\begin{cases} (M_i(z) = \text{'áno'}) \quad \wedge \quad (S(z) = \text{ 'nie' alebo } f(|z|) \text{ je nepárne }), \text{ alebo} \\ (M_i(z) = \text{ 'nie'}) \quad \wedge \quad (S(z) = \text{ 'áno'}) \quad \wedge \quad (f(|z|) \text{ je párne }) \end{cases}$$

$\mathbf{k = 2i - 1}$ F postupne simuluje $Z, R_i(z), S(R_i(z)), F(|R_i(z)|)$, kde z sú opäť postupne dosadzované binárne reťazce. Pritom sa **snaží nájsť z také, aby** $\mathbf{K(R_i(z)) \neq S(z)}$. Hľadáme teda z tak, aby

$$(**)\begin{cases} (S(z) = \text{'áno'} \quad \wedge \quad (S(R_i(z)) = \text{'nie'} \quad \vee \quad f(|R_i(z)|) \text{ je nepárne }), \text{ alebo} \\ (S(z) = \text{'nie'} \quad \wedge \quad (S(R_i(z)) = \text{ 'áno'} \quad \wedge \quad f(|R_i(z)|) \text{ je párne}) \end{cases}$$

Ak F nájde také z, tak $f(n) = k + 1$, inak $f(n) = k$.

Ostáva ukázať, že popísaný jazyk má požadované vlastnosti.
$\boldsymbol{F \in NP}$ je zrejmé.

$\boldsymbol{L \in NP}$ – stačí uhádnuť spĺňajúce priradenie hodnôt premenných a vypočítať hodnotu $f(|x|)$

$\boldsymbol{L \notin P \wedge L \notin NPU}$ dôkaz rozdelíme na dve časti, v oboch prípadoch budeme postupovať sporom.

$L \notin P$

Predpokladajme, že $\mathbf{L \in P}$. Potom existuje index i a polynomiálne ohraničený stroj M_i taký, že $L(M_i) = L$. To ale znamená, že $K(z) = M_i(z)$ pre každé z. Preto v druhej etape svojho výpočtu, keď $\mathbf{k = 2i}$ nenájde z také, aby platila podmienka (*). Preto $f(n) = 2i \forall n \geq n_0$. Následne $f(n)$ je párne až na konečne veľa n. Preto odpoveď K koinciduje so SAT až na konečne veľa vstupov. To ale znamená, že $SAT \in P$ a to je spor s predpokladom $P \neq NP$.

$L \notin NPU$

Nech by $\mathbf{L \in NPU}$. Potom existuje redukcia R_j SAT na L, a teda $K(R_j(z)) = S(z) \forall z$. Potom v druhej etape, keď $\mathbf{k = 2j}$ **-1** stroj F nemôže nájsť z spĺňajúce (**). Preto $f(n) = 2j - 1$ až na konečne veľa n. To znamená, že jazyk L je konečný, a preto máme spor s predpokladom $L \in NPU, P \neq NP$

□

7.4.2 Relativizácia problému $P \stackrel{?}{=} NP$

Definícia 7.6 TS s orákulom $M^?$ *je viacpáskový TS, ktorý má špeciálnu pásku (dotazovacia, query) a 3 špeciálne stavy* $q?, q_{ano}, q_{nie}$.

Nech $A \in \Sigma^*$. *Výpočet orákulovského stroja* M^A *prebieha ako výpočet normálneho TS dovtedy, kým sa* M^A *nedostane do konfigurácie so stavom* $q?$. *Z tejto konfigurácie prejde stroj do stavu* $\begin{cases} q_{ano}, & \text{ak obsah } x \text{ dotazovacej pásky patrí do } A \\ q_{nie}, & \text{ak obsah } x \text{ dotazovacej pásky nepatrí do } A \end{cases}$

Definícia 7.7 *Nech* $C = DTIME(f(n))$, *A je jazyk. Potom*

$$C^A = \{L \mid L = L(M^A), \text{ časová zložitosť } M^A \text{ je } f(n)\}$$

Potom otázka $P \stackrel{?}{=} NP$ je vlastne otázkou $P^\emptyset \stackrel{?}{=} NP^\emptyset$. Ukážeme, že platí protichodná relativizácia, pretože

$$\exists A \quad P^A = NP^A \qquad \wedge \qquad \exists B \quad P^B \neq NP^B$$

Veta 7.10 *Existuje orákulm A také, že* $P^A = NP^A$

Dôkaz: V tomto prípade potrebujeme zoslabiť silu nedeterminizmu. Keďže DTS a NTS sú si v polynomiálnom priestore rovné, zoberieme ako A ľubovoľný PSPACE-úplný jazyk[3]. Ukážeme, že

$$PSPACE \subseteq P^A \subseteq NP^A \subseteq NPSPACE = PSPACE$$

PSPACE$\subseteq P^A$ Nech $L \in PSPACE$. Potom existuje redukcia L na A. Orákulovský stroj na rozhodovanie L pracuje nasledovne:
– najprv odsimuluje redukciu, pričom výstup $r(w)$ ukladá na dotazovaciu pásku. Uvedomme si, že redukcia v logaritmickom priestore sa počíta v polynomiálnom čase
– potom dotazom na orákulum zistí, či výsledok $r(w)$ patrí do A
– nakoniec akceptuje práve vtedy ak $r(w)$ patrí do A

$NP^A \subseteq$NPSPACE Nech $L \in NP^A$. Potom existuje orákulovský stroj M^A polynomiálnej časovej zložitosti rozpoznávajúci L. Preto je M^A aj polynomiálnej priestorovej zložitosti. Keďže A je PSPACE úplný, je aj z PSPACE, a preto existuje TS T_A, ktorý rozpoznáva A a je navyše polynomiálnej priestorovej zložitosti.
NTS rozpoznávajúci L v polynomiálnom priestore pracuje tak, že simuluje výpočet M^A, ale každý dotaz na orákulum zodpovie odsimulovaním T_A so vstupom, ktorý je na dotazovacej páske.

□

Veta 7.11 *Existuje orákulum B také, že $P^B \neq NP^B$.*

Dôkaz: V tomto prípade potrebujeme orákulum, ktoré umocní silu nedeterminizmu. Pomocou tohto orákula zadefinujeme jazyk L, ktorý bude z NP^B, a ukážeme o ňom, že nepatrí do P^B.

$$L = \{0^n \mid \exists x \in B,\ |x| = n\}$$

$L \in NP^B$ Ak poznáme orákulum B, je rozpoznávanie jazyka L jednoduché. Pre vstup $w, |w| = n$
– stroj najprv nedeterministicky uhádne x, $|x| = n$
– potom dotazom na orákulum overí, že $x \in B$

orákulum B Pri definovaní orákula B budeme myslieť na to, že časť $L \notin P^B$ budeme dokazovať diagonalizáciou.
Nech $M_1^?, \ldots$ je číslovanie polynomiálne ohraničených orákulovských DTS[4]; každý stroj je v tejto postupnosti nekonečne veľakrát (so zbytočnými stavmi, so zbytočne veľkým stupňom polynómu). B definujeme iteratívne

$B_0 = X = \emptyset$

Keď máme
X - tie slová, ktoré nesmú patriť do B
B_{i-1} - tie slová z B, ktorých dĺžka je menšia ako i
tak B_i budujeme nasledovne:

Simulujeme $i^{\log i}$ krokov stroja M_i^B so vstupom 0^i. Ak bol počas simulácie vznesený dotaz na orákulum B – $x \in ?B$, tak odpovedáme podľa toho, čo vieme o budovanom orákule B doteraz

[3] redukcia v logaritmickom priestore
[4] Ak i je poradové číslo orákulovského stroja $OTS(i)$ bez obmedzenia času, j je stupeň polynómu, $k(i,j)$ je (jednoznačný) kód priradený dvojici (i,j). Potom $M_{k(i,j)}^?$ je orákulovský stroj, ktorý simuluje n^j krokov stroja $OTS(i)$.

$|x| \leq i-1$ $\begin{cases} q_{ano}, & \text{ak } x \in B_{i-1} \\ q_{nie}, & \text{ak } x \notin B_{i-1} \end{cases}$

$|x| \geq i$ potom q_{nie} a x pridaj do X.

Ak v priebehu $i^{\log i}$ krokov simulovaný stroj

1. zastane a zamieta, tak $B_i \leftarrow B_{i-1} \cup \Delta(i),\ \Delta(i) = \{x \in \{0,1\}^i, x \notin X\}$
2. zastane a akceptuje, tak $B_i \leftarrow B_{i-1}$
3. nezastane, tak $B_i \leftarrow B_{i-1}$

Pri dôkaze $L \notin P^B$ postupujeme sporom. Predpokladajme, že $M_i^B, i \in \{i1, i2, \ldots\}$[5], je postupnosť orákulovských strojov, ktoré rozhodujú L v polynomiálnom čase. Nech 0^i je vstupné slovo. Podľa definície jazyka L $0^i \in L$ práve vtedy, keď v B existuje slovo dĺžky i. Všimnime si preto vytváranie B_i; ak pri simulovaní nastal prípad

$L \notin P^B$

1. tak vstupné slovo nepatrí do L, čo znamená, že v B neexistuje slovo dĺžky i. Lenže množina $\Delta(i)$ je neprázdna ($|X| \leq \sum_{j=1}^{i} j^{\log j} < 2^i$) a preto sme do B pridali aspoň jedno slovo dĺžky i. To je spor.

2. tak vstupné slovo patrí do L, čo znamená, že v B by malo existovať slovo dĺžky i; my sme však definovali B_i tak, že $B_i \leftarrow B_{i-1}$ – neobsahuje teda slovo dĺžky i. Opäť máme spor.

3. V tomto prípade nemáme priamo spor. Túto možnosť však môžme vylúčiť. Prečo? To, že $L \in P^B$ znamená, že existuje polynóm ohraničujúci časovú zložitosť orákulovského stroja, ktorý L rozhoduje. Nech stupeň tohto polynómu je s. Potom existuje také I (I je index jedného z orákulovských strojov s časovou zložitosťou nanajvýš n^s), že $I^{\log I} \geq I^s$. Ak uvažujeme $i = I$, simulovaný počet krokov je dostatočný na to, aby stroj ukončil simulovaný výpočet a zastal.

□

[5] už sme spomínali, že ich je nekonečne veľa