

Teorie ICT

Přednášky

1. MNOŽINY

„Naivní“ „definice“ (pojetí): Množina [set] je přesně definovaný soubor prvků, které mají nějakou vlastnost. O čemkoliv je třeba umět rozhodnout, zda do dané množiny patří či nikoliv.

Vztah náležení: „Prvek x patří do množiny A “ značíme $x \in A$.

Dvě množiny jsou totožné tehdy a jen tehdy, mají-li stejné prvky. Každý prvek první množiny je prvkem druhé množiny a každý prvek druhé množiny je prvkem první množiny. Množina prvek buď obsahuje nebo neobsahuje. Nemůže jej obsahovat vícekrát.

Množiny mohou být prvky dalších množin.

Množina může být určena:

1. Jmenovitým výčtem svých prvků.

2. Vlastností (predikátem), který mají prvky splňovat.

Vymezení způsobem č. 2 může být málo „opatrné“. Připouští nejasně určené množiny. Může vést k paradoxům.

Tak zvaný Russelův paradox: Množina je slušná, pokud sama sebe neobsahuje jako prvek. Množina je divná, pokud je sama svým prvkem. Každá množina je buď slušná nebo divná. Slušná množina není divná. Divná množina není slušná. Jaká je množina všech slušných množin?

Jeho varianty „paradox mostu se šibenicí“, „paradox holiče“.

Problém je skrytý logický kruh. Množinu nelze vytvořit pokud k její konstrukci je třeba brát v úvahu i vytvářenou množinu samu.

Matematika potřebuje pracovat s přesně definovanými objekty!

Pojem množina je třeba definovat spolu se základní vlastností náležení „ \in “ axiomaticky.

= Stanovit základní vlastnosti tohoto pojmu.

Matematika však potřebuje pracovat i s méně určitě definovanými soubory objektů. Aby se předešlo paradoxům, tyto objekty (zvané třídy – [class]) nemohou být již prvky dalších tříd.

Přesný výklad teorie množin nutno zavádět souběžně s jazykem teorie množin, kterým je výroková a predikátorová logika. Výroky o množinách lze formulovat jen v jazyce teorie množin. Ne v metajazyce (jazyce o těchto výrocích).

My se omezíme na neformální výklad.

Existuje několik axiomatických systémů popisujících teorii množin.

Nejznámější:

Zermelo – Frankel: Základní pojem je množina a vztah náležení (být prvkem množiny) \in .

Axiomy (pouze pro ilustraci):

- **existence** ($\exists x: x = x$) – existuje aspoň jedna množina

- **extensionality** ($\forall u: u \in x \Leftrightarrow u \in y \Rightarrow x = y$) – množiny, které mají stejné prvky se rovnají

- **vydělení (schéma)** – z každé množiny lze vydělit množinu prvků, které splňují danou formuli – odtud plyne i existence prázdné množiny \emptyset

- **dvojice** – libovolné dvě množiny určují dvouprvkovou množinu

- **sumy** – ke každé množině tvoří všechny prvky, které náležejí do nějakého jejího prvku množinu.

- **potence** – všechny podmnožiny každé množiny tvoří množinu

- **nahrazení** (schéma) – říká v podstatě, že obrazem libovolné množiny při definovaném zobrazení je množina

- **nekonečna** ($\exists z: \emptyset \in z \wedge (\forall x: x \in z \Rightarrow x \cup \{x\} \in z)$) – postuluje existenci potenciálně nekonečné množiny

- **fundovanosti (regularity)** - ($\forall a: a \neq \emptyset \Rightarrow (\exists x: x \in a \wedge x \cap a = \emptyset)$) – nepřipouští například aby množina byla svým vlastním prvkem nebo aby v relaci náležení byly cykly konečné délky.

Uvedené axiomy nejsou nezávislé. Každý axiom schématu vydělení je důsledkem některého axiomu nahrazení. Axiom dvojice je důsledkem axiomu potence a schématu nahrazení.

Třídy jsou soubory množin, definované formulami jazyka teorie množin – predikátorového počtu (vlastností, která má být splněna). Každá množina je třídou. Třída může, ale nemusí být množinou. Třídy, které nejsou množinami, tak zvané „vlastní třídy“ nemohou být prvky dalších tříd (a tedy samozřejmě i množin).

Gödel – Bernays (- von Neumann) : Základní pojem je třída. Axiomy jsou požadovány pro třídy a vztah náležení (být prvkem třídy) \in .

Axiomy jsou požadovány pro třídy a vztah \in . Třída se nazývá množinou, pokud je prvkem nějaké třídy.

Oba způsoby vedou na touž teorii, pokud do soustavy axiomů zařadíme axiom vydělení (Všechny prvky z dané množiny, splňující daný predikát, tvoří množinu).

Její bezespornost nelze dokázat v rámci teorie množin.

Axiomatická teorie množin odstraní paradoxu Russelova typu. Jsou však i jiné „paradoxy“:

Richardův: Existuje nekonečně mnoho přirozených čísel. Pomocí konečně mnoha písmen abecedy lze popsat jen konečně mnoho z nich. Existují tedy čísla, které nelze popsat pomocí nejvýše 100 písmen. Každá neprázdná množina přirozených čísel má nejmenší prvek. Toto tvrzení je ekvivalentní známému principu matematické indukce. Tento prvek je nejmenší přirozené číslo, které nelze popsat pomocí nejvýše 100 písmen. Právě se nám jej popsat ale podařilo (text kurzívou).

Popis je však mimo rámec jazyka teorie množin. Je v metajazyce. Podmínka „být popsateľný s užitím nejvýše 100 písmen“ je formulována v jazyce o výrociích v jazyce teorie množin.

Zenonův (paradox hromady): Kolik zrníček písku smíme uprat z hromady písku, aby zůstala ještě hromadou? Varianta je „paradox plešatého“: Kolik vlasů si musíme vytrhat, abychom byli považováni za plešatého?

Těž: Kolik lidí se vmáčkne do autobusu? Kdy bude zeměkoule přelidněná? ...

Varianta Prof. Vopěnky: Podle teorie Ch. Darwina existuje konečná posloupnost tvorů, na jejímž začátku byl opičák Charlie a na konci ctihodný učenec Charles Darwin. Charlie byl opice, Darwin člověk. Žádný člověk není opice a žádná opice není člověk. Děti opic jsou zase opice, ne lidi. Pokud v popsání posloupnosti tvorů tvoří lidé a opice množiny, existuje první člověk, který má svým přímým předkem opici. Ale pak by byl dítětem opice a tedy opicí.

Pro třídy (a tedy i množiny) je definován vztah **inkluse** (býti částí):

$$A \subseteq B \text{ (} x \in A \Rightarrow x \in B \text{)}$$

$$A \subset B \Leftrightarrow (A \subseteq B \wedge A \neq B)$$

Z axiomu extensionality plyne že $A = B \Leftrightarrow (A \subseteq B \wedge B \subseteq A)$.

Pozor! V některé literatuře může být místo $A \subseteq B$ užito $A \subset B$ a místo $A \subset B$ užito $A \subsetneq B$.

Klasická matematika požaduje (axiomem vydělení), aby každá část množiny byla zase množina ne vlastní třída.

Je-li A třída, $A \subseteq B$ a B je množina, říkáme, že třída A je polomnožina. V klasické matematice je každá polomnožina množinou.

Toto tvrzení však nelze prostředky teorie množin ani dokázat, ani vyvrátit z ostatních axiomů teorie množin.

Pokud je teorie množin bez axiomu vydělení bezesporná, zůstane bezespornou i po přidání axiomu: „každá polomnožina je množinou“, i jeho negace „existuje polomnožina, která není množinou“ (tak zvaná vlastní polomnožina).

V klasické matematice (bez existence vlastních polomnožin) považujeme za třídy, které nejsou množinami (a z kterých nelze vytvářet další třídy pouze „hodně velké soubory“, kde neostrost je dána jejich nadměrným a tedy neurčitým a nejasným rozsahem).

Přijetím axiomu existence vlastních polomnožin lze vybudovat tak zvanou „alternativní matematiku“. Zde kromě neurčitosti co do rozsahu máme ještě neurčitost „uvnitř“ ostře definovaného souboru entit (množiny), danou nejasným rozlišením předmětů, které jsou za „naším obzorem“.

V alternativní matematice existují „nekonečně velká přirozená čísla“ a jejich převrácené hodnoty „nekonečně malá“ ale nenulová reálná čísla. Derivace zde není limitou, ale přímým podílem dvou nekonečně malých hodnot.

V alternativní matematice se podařilo vybudovat vše, o co byl učiněn pokus. Zatím však ne něco nového, co by v klasické nebylo známo také.

Úvahy v alternativní teorii množin a alternativní matematice jsou však zajímavé, ale poněkud „nezvyklé“.

Alternativní teorie množin odstraňuje paradoxy Zenonova typu. Hromady tvoří vlastní polomnožinu, která je částí množiny všech množin zrněk písku. Plešatci vlastní polomnožinu množiny všech lidí a lidé i opice vlastní polomnožiny množiny všech tvorů vývojového řetězce rodičů a dětí od opičáka Charlieho k Ch. Darwinovi.

Neprázdné podmnožiny množiny přirozených čísel mají první prvek i v alternativní teorii množin. Množina přirozených čísel obsahuje ale jako své části i polomnožiny, které nejsou množinami. Polomnožiny nejmenší prvek mít nemusí.

Matematická indukce pro přirozená čísla platí jen pro množiny. Ne pro vlastní polomnožiny.

Otázka která z obou teorií množin (klasická či alternativní) je správnější je špatně položena. Z hlediska logiky jsou rovnoprávné. Pro popis skutečnosti může být někdy vhodnější jedna, jindy druhá. Každá z nich představuje poněkud jiný pohled na svět kolem nás. Bližší o alternativní teorii množin lze nalézt v knihách Prof. Vopěnky.

Značení množin

Množina určená výčtem svých prvků: { <výčet prvků, oddělených čárkami> } Nezáleží na pořadí.

Množina prvek obsahuje nebo neobsahuje. Nemůže jej obsahovat „násobně“.

Příklady:

\emptyset prázdná množina. Neobsahuje žádný prvek.

Množina {1, 3, 5, 7, 9} obsahuje lichá přirozená čísla menší než 10.

Množina { \emptyset , { \emptyset } }; obsahuje prázdnou množinu a množinu o jediném prvku, kterým je prázdná množina.

Množina (třída) určená podmínkou, kterou musí splňovat její prvky: { $x : P(x)$ }, někdy též { $x \mid P(x)$ } nebo, je-li výběr prvků omezen na množinu A : { $x \in A : P(x)$ }, případně { $x \in A \mid P(x)$ }, kde P je podmínka, kterou mají prvky splňovat.

Pokud je A množina a P formule teorie množin, potom v klasické teorii množin je { $x \in A : P(x)$ } též množina.

Označení některých často používaných množin:

Nmnožina všech přirozených čísel

Z množina všech celých čísel

Qmnožina všech racionálních čísel

\mathbb{R}množina všech reálných čísel.

$\{x \in \mathbb{R} ; x > 0\}$ =množina všech kladných reálných čísel \mathbb{R}^+

$\{x \in \mathbb{R} ; x < 0\}$ =množina všech záporných reálných čísel \mathbb{R}^-

$\{x \in \mathbb{R} ; x \geq 0\}$ =množina všech nezáporných reálných čísel \mathbb{R}^+_0

$\{x \in \mathbb{R} ; x \leq 0\}$ =množina všech nekladných reálných čísel \mathbb{R}^-_0

$\{x \in \mathbb{Z} ; x < 0\}$ =množina všech celých záporných čísel \mathbb{Z}^-

$\{x \in \mathbb{Z} ; x > 0\}$ =množina všech celých kladných čísel \mathbb{Z}^+

$\{x \in \mathbb{Z} ; x \geq 0\}$ = \mathbb{Z}_0množina všech celých nezáporných čísel ...

(místo \mathbb{Z} se někdy užívá \mathbb{I} (integer))

Příklady:

Množina $\{n \in \mathbb{N} : (n \bmod 2 \neq 0) \wedge (n < 100)\} = \{1, 3, 5, 7, 9\}$ lichá přirozená čísla menší než 10.

Množina $\{(x, y) : x \in \mathbb{R} \wedge y \in \mathbb{R} \wedge (x^2 + y^2) \leq 1\}$ uzavřený jednotkový kruh v rovině.

Operace se třídami a množinami

Formulujeme pro množiny. Platí však i pro třídy.

Základní operace množinové algebry:

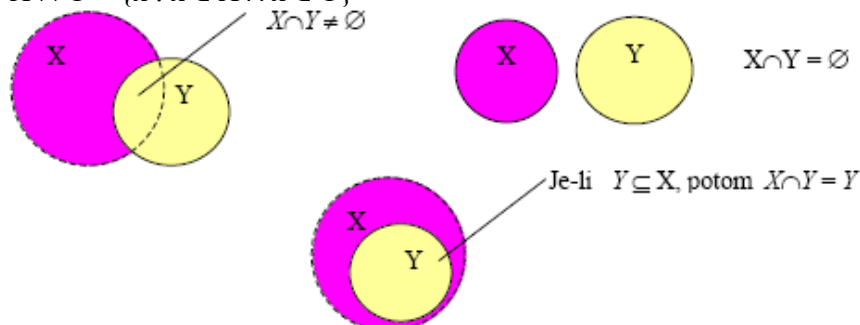
\capprůnik,

\cupsjednocení,

$-$, někdy \div , rozdíl

Průnik množin (tříd) X a Y , píšeme, je množina (třída) všech prvků, které patří zároveň do obou množin.

$X \cap Y = \{x : x \in X \wedge x \in Y\}$



Dále platí: Buď X libovolná množina. Potom $X \cap \emptyset = \emptyset$. Množiny X a Y pro které platí $X \cap Y = \emptyset$ se nazývají **disjunktní**.

Sjednocení dvou množin X a Y , píšeme $X \cup Y$, je množina všech prvků, které patří alespoň do jedné z množin X , Y . $X \cup Y = \{x : x \in X \vee x \in Y\}$

$X \cup Y = \{x : x \in X \vee x \in Y\}$

Platí $X \cup \emptyset = X$

Je-li $X \subset Y$, potom $X \cup Y = Y$

Rozdíl dvou množin X a Y , píšeme $X - Y$ (někdy $X \div Y$), je množina všech prvků množiny X , které nepatří do množiny Y . $X - Y = \{x : x \in X \wedge x \notin Y\}$

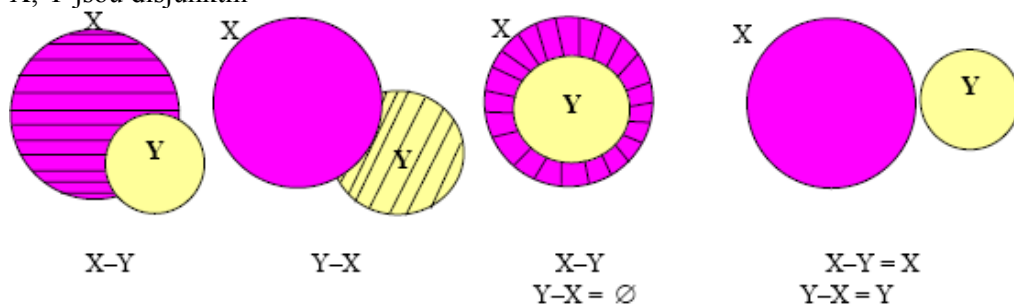
$X - Y = \{x : x \in X \wedge x \notin Y\}$

Někdy se v množinové algebře užívá i termín **doplňěk**. Nechť $A \subset B$. Potom doplňkem množiny A v množině B , píšeme A'_B , je množina všech prvků z B , které nepatří do množiny A . $A'_B = B - A$

$A'_B = B - A$

Následující ukázka ilustruje výsledný (vyšrafovaný) rozdíl dvou množin X , Y s ohledem na jejich vzájemný vztah.

X , Y jsou disjunktní



Dále se někdy užívá pojem **symetrická** difference množin. Symetrickou diferencí množin A a B , píšeme $A \oplus B$, je množina všech prvků, které patří do jedné a jen do jedné z množin A a B . $A \oplus B = A - B \cup B - A$

Platí $A \oplus B = (A \cup B) - (A \cap B)$.

Mocninou, neboli potenci množiny A , značíme $\exp(A)$ nebo 2^A , nazýváme množinu všech podmnožin množiny A .

Příklad: $A = \{1, 2, 3\}$; $\exp(A) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, A\}$. Má-li konečná množina n prvků, má její potenci 2^A 2^n prvků. Později ukážeme, že i pro nekonečné množiny má potenci vždy „více“ prvků než původní množina.

Operace průnik a sjednocení lze rozšířit na libovolný (konečný i nekonečný) systém množin. Tyto množiny označíme indexem vybíraným z nějaké množiny J . Zdůrazněme, že indexy musí tvořit množinu. Ne vlastní třídu, která není množinou.

Průnik $\bigcap_{j \in J} A_j$ je množina prvků, které jsou zároveň ve všech množinách A_j pro $j \in J$.

Sjednocení $\bigcup_{j \in J} A_j$ je množina prvků, které leží aspoň v jedné množině A_j pro $j \in J$.

Uspořádaná dvojice a n-tice prvků (a, b)

Lze zavést jako množinu o dvou prvcích:

- první prvek je množina $\{a, b\}$, obsahující prvky a a b a žádné jiné. Tato množina má dva prvky je-li $a \neq b$ a jediný prvek je-li $a = b$.

- druhý prvek je a . Ten určuje, který prvek z dvojice je v uspořádání na prvním místě.

Uspořádaná n-tice (a_1, a_2, \dots, a_n) se zavede indukcí. Pro $n = 2$ je to uspořádaná dvojice. Pro $n > 2$ uspořádaná dvojice z $(n-1)$ -tice a dalšího prvku.

Kartézský součin $A \times B$ množin A a B je množina všech uspořádaných dvojic prvků z nichž první leží v A a druhý v B
 $A \times B = \{(a, b) : a \in A \wedge b \in B\}$ Podobně pro libovolný konečný počet množin

$A_1 \times A_2 \times \dots \times A_n = \{(a_1, a_2, \dots, a_n) : a_j \in A_j \text{ pro } j = 1, 2, \dots, n\}$

Relace

Nechť A_1, A_2, \dots, A_n jsou množiny. **Potom n-ární relací s doménami A_1, A_2, \dots, A_n nazýváme libovolnou podmnožinu kartézského součinu $A_1 \times A_2 \times \dots \times A_n$.**

Relaci tedy tvoří **některé** uspořádané n-tice. Ty, které vyjadřují daný vztah. Položky relace vyjadřují atributy. Typické užití – relační báze dat.

$n = 1$... unární relace

$n = 2$... binární relace

$n = 3$... ternární relace

Je-li $A_1 = A_2 = \dots = A_n = A$, říkáme relaci **n-ární relace na množině A** . Taková relace vyjadřuje vztah mezi prvky téže množiny A .

Zobrazení

Zobrazení z množiny X do množiny Y je předpis, který některým prvkům $x \in X$ jednoznačně přiřadí jediný prvek $f(x) = y \in Y$.

Ekvivalentní vyjádření: Zobrazení z X do Y je relace f s doménami X a Y pro kterou platí $((x, y) \in f \wedge (x, z) \in f) \Rightarrow y = z$

Pokud množinu Y tvoří nějaký obor čísel, říkáme zobrazení f obvykle **funkce**.

Zobrazení z množiny A nemusí být definováno pro všechny prvky A . Ty, pro které definováno je tvoří definiční obor zobrazení f . Značíme jej $\text{Def}(f)$. $\text{Def}(f) \subseteq X$.

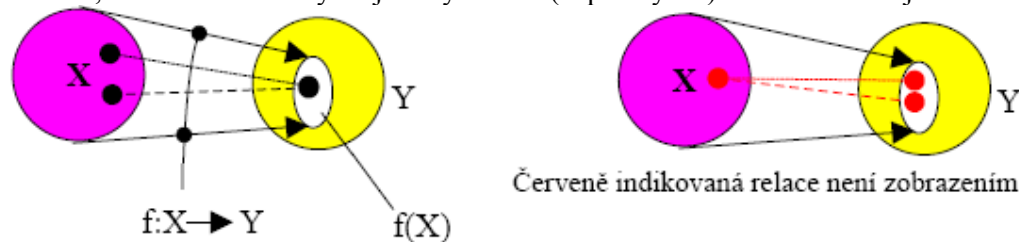
Totální zobrazení neboli zobrazení X do Y (bez předložky „z“) je zobrazení pro které je $\text{Def}(f) = X$.

Takové zobrazení je definováno pro všechny prvky množiny X .

Pokud množinu X tvoří množina přirozených čísel \mathbb{N} , říkáme zobrazení f do Y obvykle **posloupnost** prvků množiny Y .

Vždy platí $f(X) \subseteq Y$. Může být $f(X) = Y$, ale může také být $f(X) \subsetneq Y$.

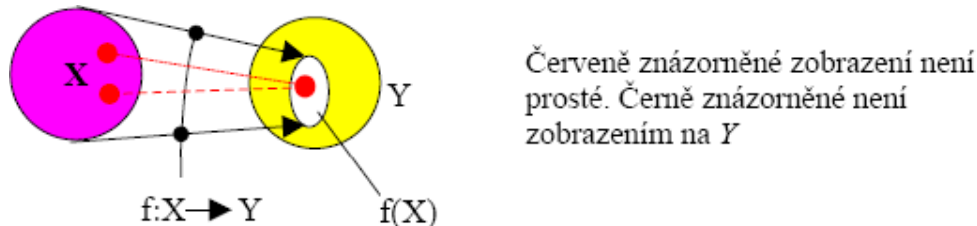
Nevadí, že dva různé vzory mají tentýž obraz (např. $f: y=x^2$). Nemůže však jeden vzor mít dva různé obrazy.



Surjektivní zobrazení neboli zobrazení X na množinu Y je zobrazení X do Y při kterém každý prvek z množiny Y má v X aspoň jeden vzor. Pro zobrazení X na Y je $f(X) = Y$.

Injektivní neboli **prosté** zobrazení množiny X do množiny Y . Je takové zobrazení X do Y pro které platí: $f(x) = f(y) \Rightarrow x = y$, (neboli $x \neq y \Rightarrow f(x) \neq f(y)$).

Různé vzory musí mít různé obrazy. Každý prvek množiny Y má v X nejvýše jeden (tedy jeden nebo žádný) vzor v množině X .



Bijektivní zobrazení X na Y , neboli **vzájemně jednoznačné** zobrazení je takové zobrazení, které je současně surjektivní i injektivní. Je to tedy prosté zobrazení X na Y .

K bijektivnímu zobrazení $f: X \rightarrow Y$ existuje jednoznačně určené inverzní zobrazení $f^{-1}: Y \rightarrow X$, které je rovněž bijektivní. Je definováno vztahem $f^{-1}(y) = x \Leftrightarrow y = f(x)$. Zřejmě je $(f^{-1})^{-1} = f$.

Pojem zobrazení nám umožňuje **rozšířit pojem kartézského součinu i na kartézský součin nekonečně mnoha množin**. Mějme (třeba i nekonečnou) množinu indexů J . Každému prvku $j \in J$ nechť je přiřazena nějaká množina A_j . Zdůrazňujeme, že indexy musí tvořit množinu. Ne vlastní třídu.

Kartézský součin $\times_{j \in J} A_j$ je zobrazení f množiny indexů J do množiny $\cup_{j \in J} A_j$, pro které platí $f(j) \in A_j$. Zdá se, že kartézský součin neprázdného systému neprázdných množin musí být vždy neprázdna množina. Z každé množiny A_j lze vždy vybrat jeden prvek a tak vytvořit prvek jeden kartézského součinu. Jiným výběrem získáme jiný prvek. Tomuto tvrzení se říká **axiom výběru**.

Toto tvrzení však nelze z axiomů teorie množin ani dokázat ani vyvrátit. Je na nich nezávislé. Bylo ukázáno, že pokud jsou axiomy teorie množin nerozporné, zůstanou nerozpornými i při přidání axiomu výběru a rovněž zůstane nerozporná po při přidání jeho negace.

Potíž spočívá v tom, že pokud je množina J nekonečná, výběr nikdy nedokončíme celý a nikdy efektivně nebudeme mít žádaný prvek součinu $\times_{j \in J} A_j$ celý k dispozici.

Bez axiomu výběru se většina moderní matematiky neobejde. Je nutné jej předpokládat ve většině úvah matematické analýzy. Bez něj by byla matematika velmi chudá. Neměli bychom například k dispozici většinu přibližných metod výpočtu běžně užívaných pro řešení úloh, jejichž dopad je velmi praktický. Proto se zpravidla platnost axiomu výběru předpokládá.

Přesto stojí zato uvést, že axiom výběru má řadu velmi překvapivých až kuriózních důsledků.

Snad nejexotičtější a nejméně uvěřitelnější vypadá tak zvaný paradoxní rozklad koule dokázaný polským matematikem Banachem: Kouli o poloměru jedna v trojrozměrném prostoru lze rozložit na 5 navzájem disjunktních částí, s každou z těchto částí provést pohyb (bez jakékoliv deformace, pouhé posunutí a pootočení každé části zvlášť) a složit z těchto částí dvě nové plné koule z nichž každá má stejnou velikost jako původní jediná koule. Tato neuvěřitelná věta je skutečně důsledkem axiomu výběru a lze ji bezchybně a poměrně snadno dokázat.

Problém spočívá v tom, že uvedený rozklad sice existuje, ale nejsme schopni jej efektivně realizovat. Paradoxní rozklad koule ukazuje, že v trojrozměrném prostoru (a ve všech prostorech vyšší dimenze) nelze zavést míru objemu těles tak, aby byla současně neměnná při pohybu a zachovávala se při skládání těles z navzájem disjunktních částí. Je zajímavé, že na přímce a v rovině takovou míru zavést lze.

Mohutnost množin

Tento odstavec se týká pouze množin. Nikoliv tříd, které množinami nejsou.

Pro množiny s konečným počtem prvků platí následující tvrzení:

Pokud množiny A a B mají stejný počet prvků, existuje prosté zobrazení A na B .

Pokud množina A má méně prvků než B , existuje prosté zobrazení A do B , ale neexistuje prosté zobrazení B do A .

Pokud množina B má méně prvků než A , existuje prosté zobrazení B do A , ale neexistuje prosté zobrazení A do B .

Pokud u konečných množin existuje prosté zobrazení jedné na druhou, potom každé jiné prosté zobrazení jedné do druhé je také zobrazením na tuto množinu.

To nás vede k myšlence jak rozšířit pojem „počet prvků“ i na nekonečné množiny. Tomuto zobecnění říkáme **mohutnost množiny** $\text{moh}(A)$ nebo kardinalita množiny $\text{card}(A)$. Hodnotě, která tento zobecněný počet vyjadřuje **kardinální číslo**. Přirozená čísla jsou kardinálními čísly konečných množin. Nekonečná kardinální čísla zavedeme.

Musíme však být opatrní. Pro nekonečné množiny je situace poněkud odlišná. Vyplývá to z příkladu:

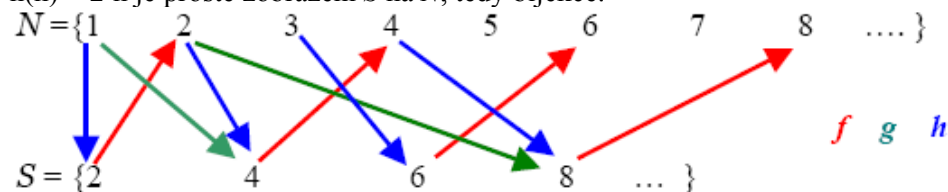
N nechť je množina všech přirozených čísel,

S množina všech sudých přirozených čísel.

$f(n) = n$ je prosté zobrazení S do N , které není zobrazením na N .

$g(n) = 4 \cdot n$ je prosté zobrazení N do S , které není na zobrazením S .

$h(n) = 2 \cdot n$ je prosté zobrazení S na N , tedy bijekce.



U nekonečných množin se tedy musíme smířit s tím, že je budeme považovat za „stejně velké“, přesněji říkat, že mají stejnou mohutnost, pokud existuje aspoň jedno prosté zobrazení jedné na druhou.

U konečných množin není možné zobrazit množinu vzájemně jednoznačně na její vlastní část (aby „něco zbylo“). Každé prosté zobrazení konečné množiny do sebe je zobrazením na sebe.

U nekonečných množin to, jak jsme právě viděli, možné je.

Touto vlastností lze dokonce konečné množiny charakterizovat.

Množina je konečná tehdy a jen tehdy, pokud každé prosté zobrazení této množiny do sebe je zobrazením na sebe.

Abychom mohli mohutnosti porovnávat navzájem potřebujeme následující větu

Cantor - Bernsteinova věta: Jestliže existuje prosté zobrazení množiny A do množiny B a současně existuje prosté zobrazení množiny B do množiny A , potom existuje také prosté zobrazení množiny A na množinu B .

Důkaz této věty vynecháme. Má však pro definici mohutnosti zásadní význam. Cantor – Bernsteinova věta umožňuje vzájemně porovnávat mohutnosti množin.

Říkáme, že množiny A a B mají stejnou mohutnost, pokud existuje prosté zobrazení A na B .

Říkáme, že množina A má mohutnost menší nebo rovnu mohutnosti B , pokud existuje prosté zobrazení A do B , tedy na podmnožinu B .

Říkáme, že mohutnost množiny A je menší než mohutnost množiny B , pokud existuje prosté zobrazení A do B a neexistuje prosté zobrazení A na B .

Kardinální číslo je společná vlastnost množin stejné mohutnosti. Přesněji: Na třídě všech množin (pozor, všechny množiny tvoří množinu, ale vlastní třídu!) je vztah „mít stejnou mohutnost“ ekvivalencí. Ta rozděluje tuto třídu na třídy ekvivalence. Kardinální čísla charakterizují tyto třídy.

Mohutnost množiny A označujeme $\text{card}(A)$, nebo $\text{moh}(A)$, někdy též $|A|$.

Množiny, které mají mohutnost stejnou jako je mohutnost množiny \mathbb{N} všech přirozených čísel se nazývají **spočetné**.

Lze je charakterizovat tím, že **všechny jejich prvky lze seřadit do posloupnosti**.

Mohutnost spočetných množin se označuje hebrejským písmenem \aleph_0 - (čti „alef nula“)

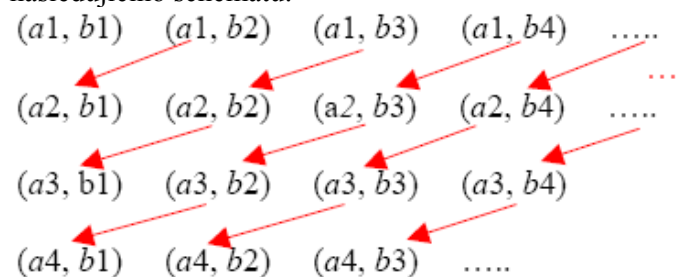
Lze ukázat, že každá nekonečná množina obsahuje spočetnou podmnožinu.

\aleph_0 je tedy nejmenší nekonečné kardinální číslo. Vyjadřuje mohutnost spočetných množin.

Snadno lze dokázat, že každá část spočetné množiny je také spočetná nebo konečná (je „nejvýše spočetná“).

Průnik dvou spočetných množin je spočetná nebo konečná množina. Sjednocení dvou spočetných množin je spočetná množina.

Kartézský součin dvou spočetných množin je rovněž spočetná množina. Jak dvojice seřadit do posloupnosti plyne z následujícího schématu:



Seřadíme: $(a_1, b_1), (a_1, b_2), (a_2, b_1), (a_1, b_3), (a_2, b_2), (a_3, b_1), (a_1, b_4), (a_2, b_3), (a_3, b_2), (a_4, b_1), (a_1, b_5), \dots$

Spočetná je tedy i množina všech racionálních čísel – $\text{moh}(\mathbb{Q}) = \aleph_0$.

Spočetné je i sjednocení spočetně mnoha spočetných množin.

Spočetná je i množina všech konečných řetězců znaků (slov) z libovolné konečné abecedy.

Do posloupnosti je seřadíme takto: Na prvé místo přijde prázdný řetězec. Poté řetězce o jednom znaku, seřazené abecedně. Poté abecedně seřazené řetězce o dvou znacích. Poté trojznaková slova, atd. Vyčerpáme tak všechna slova, libovolné konečné délky.

Jsou však i „větší“ množiny. Ukážeme, že množina všech nekonečných posloupností znaků 0 a 1 spočetná není.

Důkaz naznačíme tak zvanou diagonální metodou, která se v teorii množin užívá často (všimněte si, že do jisté míry připomíná paradox holiče i Russelův paradox). Předpokládejme, že všechny takové posloupnosti seřadit lze a že jsou seřazeny do řádků $p(1), p(2), p(3), \dots$ oboustranně nekonečné matice:

$$p(1) = a_{11}, a_{12}, a_{13}, a_{14}, \dots$$

$$p(2) = a_{21}, a_{22}, a_{23}, a_{24}, \dots$$

$$p(3) = a_{31}, a_{32}, a_{33}, a_{34}, \dots$$

...

$$p(n) = a_{n1}, a_{n2}, a_{n3}, \dots, a_{nn}, \dots$$

sestrojíme nyní posloupnost b_1, b_2, b_3, \dots takto: Je-li $a_{nn} = 0$, bude $b_n = 1$. Je-li $a_{nn} = 1$, bude $b_n = 0$.

Ta musí někde v našem schématu být, protože předpokládáme, že jsme v něm vyčerpali všechny posloupnosti.

Nechť je na místě k . To ale být nemůže, protože $b_k \neq a_{kk}$. Náš předpoklad, že v daném schématu lze vyčerpát všechny posloupnosti byl chybný. Uvažovaná množina je tedy nespočetná.

Zvážíme-li, že jde ve skutečnosti o vyjádření reálných čísel z intervalu $<0, 1>$ v dvojkové soustavě, ukázali jsme, že nespočetná je množina čísel $z <0, 1>$ a zřejmě i z intervalu $(0, 1)$. Ten lze ale snadno zobrazit prostě na množinu \mathbb{R} všech reálných čísel. I ta je tedy nespočetná. Mohutnost množiny reálných čísel říkáme mohutnost kontinua a značíme ji \mathfrak{c} nebo též 2^{\aleph_0}

Diagonální metodu lze užít i na důkaz toho, že množina 2^A všech podmnožin množiny A má vždy větší mohutnost než je $\text{moh}(A)$. Necht' f je prosté zobrazení A na 2^A .

Sestrojíme podmnožinu B množiny A takto:

Pokud pro $x \in A$ je $x \in f(x)$, pak $x \notin B$.
 Pokud $x \notin f(x)$, pak $x \in B$.

$$B = \{x \in A: x \notin f(x)\}$$

B je zřejmě podmnožina A a je tedy obrazem $f(y)$ nějakého prvku $y \in A$.

Položme si otázku, zda je $y \in B$ či nikoliv.

Snadno se přesvědčíme, že ani jedna z těchto .. Náš předpoklad, že existuje prosté zobrazení f množiny A na 2^A byl tedy chybný. Protože A lze prostě zobrazit na jednoprvkové podmnožiny A , je $\text{moh}(A) < \text{moh}(2^A)$.

Snadno se ukáže, že mohutnost množiny 2^{\aleph} všech podmnožin přirozených čísel je také c . To je také důvod toho, proč mohutnost kontinua označujeme též $2.0\aleph$.

Vzniká přirozeně otázka, zda existuje nějaké kardinální číslo, které leží mezi \aleph_0 a 2^{\aleph_0} . Negativní odpověď na tuto otázku představuje tak zvaná hypotéza kontinua. Známe sice množinu, jejíž mohutnost je dána kardinálním číslem, které bezprostředně následuje za \aleph_0 . Toto číslo značíme \aleph_1 . Nevíme však zda $\aleph_1 = 2^{\aleph_0}$ nebo zda $\aleph_1 < 2^{\aleph_0}$. Jde o další tvrzení, které nelze z axiomů teorie množin ani dokázat, ani vyvrátit.

Zda hypotézu kontinua předpokládáme nebo ne také ovlivňuje matematiku. Ne však tak zásadně jako axiom výběru.

Tvrzení o mohutnosti potence množiny ukazuje, že ke každému kardinálnímu číslu můžeme sestavit další kardinální číslo, které je větší.

Mohla by vzniknout otázka: Jaká je mohutnost „množiny“ všech kardinálních čísel?

Tato úvaha však vede jen ke sporu. Pokud pro každé kardinální číslo sestavíme množinu příslušné mohutnosti a všechny tyto množiny sjednotíme, dostaneme množinu, jejíž mohutnost je zřejmě větší nebo rovna mohutnosti všech sjednocovaných množin. Označme ji třeba Θ a její mohutnost \aleph . Číslo 2^{\aleph} je ale větší než \aleph , tedy větší než úplně všechna kardinální čísla, ale přitom je jedním z kardinálních čísel, protože je mohutností množiny 2^{Θ} .

Ve skutečnosti o žádný paradox nejde. Problém je v tom, že kardinální čísla netvoří množinu, ale vlastní třídu, která množinou není. O mohutnosti tříd nemá smysl hovořit.

Mlhavé (fuzzy) množiny (též neostře množiny)

Snaha postihnout neurčitost (vágnost) informace

Rozdílný koncept než je pravděpodobnost (stochastická neurčitost)!

Stochastická neurčitost. Po provedení pokusu dostaneme jednoznačnou odpověď

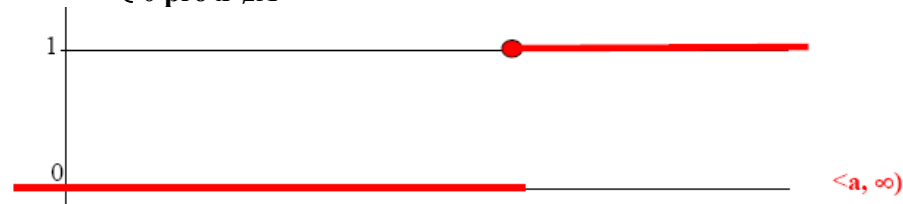
Kvantová neurčitost. Pokus nelze opakovat pro nekompatibilitu jevů

Mlhavost (vágnost). Na otázku nelze odpovědět binárně

Snaha postihnout třetí možnost.

Charakteristická funkce (funkce příslušnosti) množiny

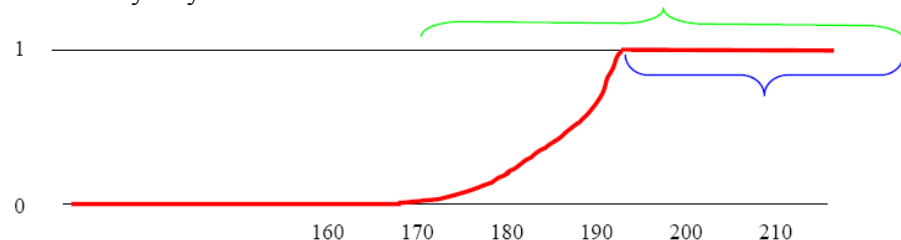
$$\mu_A(x) = \begin{cases} 1 & \text{pro } x \in A \\ 0 & \text{pro } x \notin A \end{cases} \quad \text{užito pro reprezentaci množin například v jazyce PASCAL}$$



Mlhavé množiny

Dáno univerzum U mlhavá množina je určena funkcí příslušnosti (charakteristickou funkcí): $\mu_A : U \rightarrow \langle 0, 1 \rangle$.

Příklad: Vysoký člověk



Nosič $\text{supp}(A) = \{x \in U: \mu_A(x) > 0\}$

Jádro $\text{core}(A) = \{x \in U: \mu_A(x) = 1\}$

Hladiny a řezy:

Pro $0 \leq \alpha \leq 1$ je

α -hladina mlhavé množiny A množina $\{x \in U: \mu_A(x) = \alpha\}$

α -řez mlhavé množiny A množina $\{x \in U: \mu_A(x) \geq \alpha\}$

ostrý α -řez mlhavé množiny A množina $\{x \in U: \mu_A(x) > \alpha\}$

Hladiny a řezy jsou (ostré) množiny.

Mlhavou množinu lze charakterizovat systémem řezů

$R_A: (0, 1) \rightarrow \exp(U)$

$R_A(\alpha) = \{x \in U: \mu_A(x) \geq \alpha\}$.

Inkluze pro mlhavé množiny: A je podmnožinou B ($A \subseteq B$) právě když $\mu_A(x) \leq \mu_B(x)$ pro všechna $x \in U$.

Operace s mlhavými množinami

Stupeň příslušnosti bodu k výsledku operace musí záviset pouze na stupni příslušnosti k operandům. Tím musí být jednoznačně určen. To je rozdíl od operací s pravděpodobnostní neurčitostí, kde navíc záleží na závislosti jevů.

Nejčastěji se používají tak zvané standardní operace:

Doplňěk: $\mu_{U \setminus A}(x) = 1 - \mu_A(x)$.

Sjednocení: $\mu_{A \cup B}(x) = \max(\mu_A(x), \mu_B(x))$.

Průnik: $\mu_{A \cap B}(x) = \min(\mu_A(x), \mu_B(x))$.

Jsou však možné i jiné typy základních operací (více typů doplňků, sjednocení a průniků).

Například průnik:

$\mu_{A \cap B}(x) = \mu_A(x) \cdot \mu_B(x)$ (produktový) nebo

$\mu_{A \cap B}(x) = \max(\mu_A(x) \cdot \mu_B(x) - 1, 0)$ (Lukasiewiczův), případně obecnější

$\mu_{A \cap B}(x) = \max\left(\left((1 - \mu_A(x))^w + (\mu_B(x) - 1)^w\right)^{1/w} - 1, 0\right)$ pro $w > 0$ (Yagerovy) nebo

$\mu_{A \cap B}(x) = \begin{cases} \mu_A(x) & \text{pro } \mu_B(x) = 1 \\ \mu_B(x) & \text{pro } \mu_A(x) = 1 \\ 0 & \text{jinak (tzv. drastický)} \end{cases}$

též pro fuzzy negaci, sjednocení, implikaci i ekvivalenci existuje řada alternativních možností.

S mlhavými množinami úzce souvisí tak zvaná **mlhavá (fuzzy) logika**. V mlhavé logice mohou pravdivostní hodnoty nabývat hodnot z intervalu $(0, 1)$, podle neurčitosti příslušného tvrzení.

V mlhavé logice máme také kromě standardních logických spojek možnost různých variant pro negaci, konjunkci, disjunkci, implikaci. Blíže viz skripta ČVUT: Navara, M. a Olšák, P.: Základy fuzzy množin.

Poznamenejme že

1. - **alternativní teorie množin** a alternativní matematika a

2. - **teorie mlhavých (fuzzy) množin a mlhavá logika**

představují dva velmi odlišné přístupy k popisu neurčitosti našich poznatků a práci s nimi.

První má hlubší filosofický základ a je založen na formálním modelu, který respektuje meze našeho poznání světa. **Nemá přímé výpočetní aplikace**, avšak značný teoretický význam.

Druhý je založen na aparátu klasické matematiky a ideální představě plné poznatelnosti a popisuje neurčitost prostředky klasické matematiky. **Je vhodnější výpočty a pro přímé aplikace. Pro informační techniky mnohem propracovanější.**

2. RELACE, GRAFY, OPERACE, STRUKTURY, TEORIE

n -ární relace s doménami $A_1, A_2, \dots, A_n \equiv$ Podmnožina kartézského součinu $A_1 \times A_2 \times \dots \times A_n$.

Užití v databázích.

n -ární relace na množině $A \equiv$ Podmnožina kartézského součinu $A \times A \times \dots \times A = A^n$.

Užití pro vlastnosti a vztahy mezi prvky téže množiny.

- unární relace na A formalizují vlastnost prvků,
- binární relace na A formalizují vztah mezi dvěma prvky,
- ternární relace na A formalizují vztah mezi 3 prvky, např. být součtem prvních dvou,

Preference a binární relace na množině

Formalizace vztahu „být aspoň tak dobrý“ – levnější, složitější ... (neostré relace) a „být lepší“, ... (ostré relace).

Místo $(a, b) \in R$ píšeme často $a R b$ a místo R používáme značky typu $<, \leq, \geq, \dots$

Některé důležité vlastnosti binárních relací – relace R se nazývá:

Symetrická [symetric], pokud ze vztahu $a R b$ plyne vztah $b R a$.

Nesymetrická [nosymetric], pokud existují a a b tak že $a R b$ ale neplatí $b R a$.

Antisymetrická [antisymmetric], pokud ze vztahů $a R b$ a $b R a$ plyne, že $a = b$.

Asymetrická [asymetric], pokud z platnosti $a R b$ vyplývá, že neplatí $b R a$ (všimněte si rozdílu od předchozích dvou požadavků).

Reflexivní [reflexive], pokud pro všechna $a \in A$ je $a R a$ (definici nereflexivní a antireflexivní relace, která se někdy nazývá též ireflexivní relace, doplňte sami).

Transitivní [transitive], pokud pro všechna $a, b, c \in A$ platí $(a R b \wedge b R c) \Rightarrow a R c$.

Negativně transitivní [negatively transitive], pokud z neplatnosti vztahu $a R b$ a neplatnosti vztahu $b R c$ plyne neplatnost vztahu $a R c$ (jinými slovy, pokud je $x R y$, potom pro libovolné $z \in A$ je buď $x R z$ nebo $z R y$).

Úplná [complete], pokud pro každé dva různé prvky $a, b \in A, a \neq b$, je buď $a R b$ nebo $b R a$.

Silně úplná [strongly complete], pokud pro každé dva prvky $a, b \in A$ je buď $a R b$ nebo $b R a$.

Preference vyžadují vždy transitivnost.

Pokud není ani $a > b$ ani $b > a$ může to popisovat dvě různé situace:

1. a a b jsou z hlediska našeho hodnocení zcela záměnné – ekvivalentní (co je lepší než a je také lepší než b a naopak).
2. a a b jsou z hlediska našeho hodnocení neporovnatelné (vylepším-li a na a^* , nemusí být a^* lepší než b a naopak).

\Rightarrow Ostré preference nemohou popsat situaci úplně. Je třeba dodatečně odlišit neporovnatelné a ekvivalentní prvky.

Popíšeme-li situaci pomocí neostrých preferencí lze neporovnatelné a ekvivalentní prvky odlišit takto:

- Pokud $a \geq b$ a současně $b \geq a$ jsou prvky a a b ekvivalentní,
- Pokud neplatí ani jeden ze vztahů $a \geq b$ a $b \geq a$ jsou prvky a, b neporovnatelné.

Ostrou relaci preference $>$ lze definovat přirozeně vztahem $> = \geq - \leq$.

Ekvivalenci \approx pak vztahem $\approx = \geq \cap \leq$.

Nejčastěji užívaná terminologie pro preferenční relace (pozor ne jediná užívaná!):

Ekvivalence \equiv relace, která je současně reflexivní, symetrická a transitivní. Ekvivalence generuje rozklad svého nosiče na podmnožiny vzájemně ekvivalentních prvků. Tyto podmnožiny jsou po dvou disjunktní a jejich sjednocení je celý nosič.

Pro neostré preference:

Definition: Říkáme, že binární relace \geq na množině A je (neostrým) **kvaziuspořádáním** [quasiorder] na A , **jestliže je reflexivní a transitivní**. Uvedením slova „neostrým“ v závorce rozumíme to, že jej budeme někdy vynechávat. Zde i nadále bude platit, že pokud žádné adjektivum, „neostré“ ani „ostré“ neuvedeme, budeme předpokládat, že jde o neostrý vztah.

Definition: Binární relace se nazývá **částečným uspořádáním** [partial order] na množině A , je-li kvaziuspořádáním na A a ze vztahů $a \geq b$ a $b \geq a$ plyne že $a = b$, to znamená, relace \square je antisymetrická. (Mohou existovat neporovnatelné prvky, ale ne ekvivalentní, ale různé prvky)

Definition: Binární relace na množině A se nazývá **slabým uspořádáním** [weak order] na A , je-li kvaziuspořádáním (to znamená, je-li reflexivní a transitivní) a je-li navíc silně úplná, to znamená pro libovolné dva prvky $a, b \in A$ platí aspoň jeden ze vztahů: $a \geq b$ nebo $b \geq a$.

(Mohou existovat různé ekvivalentní prvky, ale ne různé neporovnatelné prvky).

Definition: Binární relace \geq na množině A se nazývá **uspořádáním** [order], je-li slabým uspořádáním na A a pro libovolné dva prvky $a, b \in A$ platí právě jeden (to znamená aspoň jeden a zároveň nejvýše jeden) ze vztahů: $a \geq b, b \geq a, a = b$. (Někdy lineární uspořádání)

Jinými slovy, jestliže ze vztahu $a \approx b$ vyplývá $a = b$, tedy relace \geq je antisymetrická.

(Neexistují různé neporovnatelné ani různé ekvivalentní prvky)

Pro ostré preference:

Definition: Binární relaci $>$ na A nazýváme **ostrým částečným uspořádáním** [strict partial order] na A , pokud je asymetrická a transitivní.

Je zřejmé, že ostré částečné uspořádání je i antireflexivní, tedy pro všechna $a \in A$ neplatí $a > a$.

Definice: Binární relaci $>$ na A nazýváme **ostrým slabým uspořádáním** [strict weak order] na A , pokud je asymetrická a negativně transitivní.

Definice: Binární relaci $>$ na A nazýváme **ostrým uspořádáním** [strict order] na A , pokud je transitivní, asymetrická a úplná.

To znamená, že ze vztahů: $a > b$, $b > a$, $a = b$ platí u uspořádání **vždy právě jeden** (aspoň jeden a nejvýše jeden).

Přehled vlastností v tabulkové formě:

✓ znamená, že příslušná preferenční relace vlastnost má

✓✓ je použito u vlastností, které mohou sloužit jako definiční

Relace	Kvaziuspořádání	Slabé uspořádání	Uspořádání (lineární)	Ostré uspořádání	Ostré slabé uspořádání	Částečné uspořádání	Ostré částečné uspořádání	Ekvivalence
Reflexivní	✓✓	✓	✓			✓✓		✓✓
Symetrická								✓✓
Tranzitivní	✓✓	✓✓	✓✓	✓✓	✓	✓✓	✓✓	✓✓
Asymetrická				✓	✓✓		✓✓	
Antisymetrická			✓✓	✓✓	✓	✓✓		
Negativně transitivní			✓	✓	✓✓			
Silně úplná		✓✓	✓✓					
Úplná			✓	✓✓				

Zvláštní případy ostrého částečného uspořádání

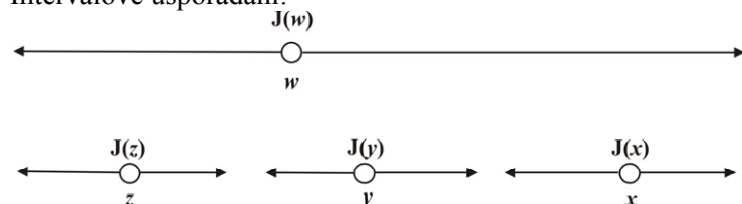
Relace s netransitivní nerozlišitelností. Popisují situace kde “neroznáme velmi malé rozdíly”.

$a \sqsubset b$ modeluje vztah a je zřetelně lepší než b .

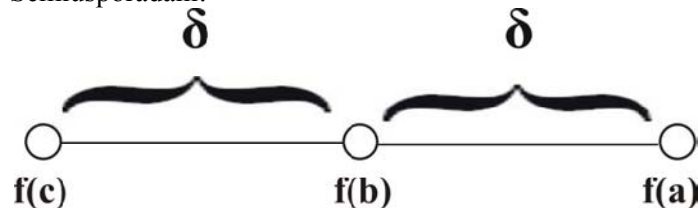
Pokud bychom se snažili nalézt vztah a a b nelze odlišit $a \approx b$ jako ekvivalenci neuspějeme, protože tento vztah není transitivní. Proto je nutné pro \sqsubset volit jiné postupy.

Nejznámější jsou intervalové uspořádání a semiuspořádání.

Intervalové uspořádání:



Semiuspořádání:



Obě relace lze popsat axiomy, které je třeba splnit. Pojem intervalové uspořádání je obecnější. Každé semiuspořádání je intervalovým uspořádáním, ne naopak. V obou případech jde o zvláštní případy ostrého částečného uspořádání.

Na základě teorie mlhavých množin lze vybudovat i teorii mlhavých (fuzzy) relací.

Representace relací:

- Matice z 1 a 0 – řád = počet prvků bazové množiny, $a_{ij} = 1 \Leftrightarrow (i, j) \in R$

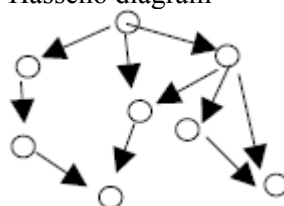
- Orientovaný graf = počet uzlů = mohutnost bazové množiny, uzly i a j propojeny orientovanou hranou $\Leftrightarrow (i, j) \in R$.

Částečné uspořádání (a každá transitivní relace) lze znázornit i tak zvaným Hasseho diagramem. V něm vynecháváme z grafu relace šipky, které již vyplývají z transitivnosti.

Úplný graf relace



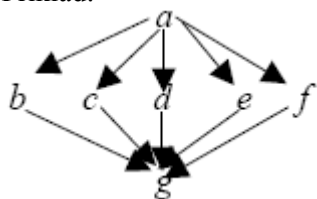
Hasseho diagram



Dimenze částečného ostrého uspořádání

Každé částečné ostré uspořádání lze doplnit na tak zvané topologické uspořádání, které je již (lineárním) ostrým uspořádáním, přidáním nových prvků (vztahů) do relace. Lze to udělat více způsoby.

Příklad:



Topologická uspořádání:

T1: $a > b > c > d > e > f > g$

T2: $a > f > e > d > c > b > g$.

dimense = 2

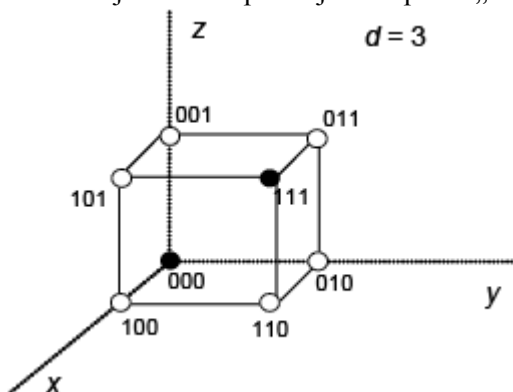
Množina topologických uspořádání se nazývá úplná pro dané částečné ostré uspořádání pokud platí:

Pokud je x před y ve **všech** topologických uspořádáních této množiny, je x před y také v částečném ostrém uspořádání.

Úplná množina topologických uspořádání charakterizuje ostré částečné uspořádání.

Minimální počet topologických uspořádání, které pro dané ostré částečné uspořádání tvoří úplnou množinu nazýváme dimense ostrého částečného uspořádání.

Dimense je důležitá pro nejmenší počet „známek“, kterými lze částečné uspořádání charakterizovat.



Příklad ostré částečné dimense 3:

Lepší \equiv všechny souřadnice jsou \geq a aspoň jedna je $>$.

Grafy

Pojem grafu souvisí úzce s pojmem binární relace. Budeme užívat následující terminologii

Orientovaný multigraf – je určen (formálně to matematici vyjadřují, že je uspořádanou trojicí (V, H, ϕ) :

- (1) konečnou neprázdnou množinou V , uzlů (vrcholů) grafu,
- (2) konečnou množinou orientovaných hran (šipek) H ,
- (3) zobrazením ϕ, H do $V \times V$, které určuje odkud kam hrana směřuje.

(Mezi dvěma vrcholy může vést žádná orientovaná hrana šipka, jedna šipka nebo více šipek.)

Pokud je zobrazení ϕ **prosté**, říkáme, že jde o **orientovaný graf** (šipka je nejvýše jedna).

V případě, že u orientovaného grafu nahradíme hranu h přímo jejím obrazem $\phi(h) = (a, b)$, pojem **orientovaný graf s množinou vrcholů V splyne s pojmem binární relace na V** .

Neorientovaný multigraf a graf vznikne tím, že ve všech definicích uspořádanou dvojici (a, b) nahradíme dvouprvkovou množinou $\{a, b\}$, je-li $a \neq b$ a jednoprvkovou množinou $\{a\}$ pro dvojici (a, a) – tak zvanou smyčku. Tedy zrušením orientace, „přeměnou šipek na neorientované hrany“.

Pozor! Zrušením orientace u orientovaného grafu může vzniknout neorientovaný multigraf, který grafem není (existují-li v orientovaném grafu šipky oběma směry).

Pozor na to, co se v jakémkoliv textu rozumí pod pojmem **graf** (bez přívlastku). Někdy se tím míní orientovaný, jindy neorientovaný graf. Někdy se grafem rozumí i graf bez smyček.

Pro různé aplikace se hodí různé varianty uvedených definic.

Některé další pojmy často užívané pro grafy (vše platí pro orientované i neorientované grafy, pokud není výslovně uvedeno jinak):

Uzlově ohodnocený (nebo též uzlově obarvený) **graf** = Každému uzlu je přiřazen prvek z nějaké množiny – jeho hodnota (Booleovská hodnota, číslo, text, ...).

Hranově ohodnocený graf = hodnota je přiřazena hranám grafu.

Sled [Trail] z uzlu x do uzlu y = posloupnost vrcholů $\{v_j\}_{j=0, \dots, N}$, taková, že $x = v_0$, $y = v_N$ a libovolné dva následující vrcholy jsou spojeny hranou. Tedy $(v_j, v_{j+1}) \in H$ pro $j = 0, \dots, N - 1$. Uzel x se nazývá počáteční, uzel y koncový uzel sledu.

Cesta [Path] z uzlu x do uzlu y – sled z x do y ve kterém se každý uzel, s výjimkou dvojice x a y , kde může být $x = y$, vyskytuje pouze jednou. Tedy $v_j \neq v_k$ jestliže $j \neq k$; $j, k = 1, \dots, N-1$. Pokud existuje sled z x do y , existuje také cesta z x do y .

Délka cesty [Path length] v hranově ohodnoceném grafu číslu = součet vah všech hran na této cestě (případně počet hran, je-li každá ohodnocena 1). **Minimální cesta** z uzlu x do uzlu y je cesta z x to y , která má nejmenší délku ze všech cest z x to y .

Souvislý neorientovaný graph [connected grph] je neorientovaný graf, v kterém mezi libovolnými dvěma vrcholy existuje aspoň jedna cesta. Orientovaný graf nazýváme souvislý, je-li souvislý multigraf vzniklý zrušením jeho orientace. Orientovaný graf je **silně souvislý [strongly connected graph]**, jestliže mezi libovolnými dvěma jeho uzly existuje (orientovaná) cesta.

Cyklus v grafu [cycle or loop] je cesta, která začíná a končí ve stejném uzlu grafu. Graf, který obsahuje aspoň jeden cyklus se nazývá cyklický graf, graf v kterém žádný cyklus není se nazývá acyklický graf.

Hamiltonovský cyklus je cyklus, který obsahuje všechny uzly grafu (a protože jde o cestu, prochází každým uzlem s výjimkou počátečního, který je totožný s koncovým, pouze jednou).

Les [Forest] je neorientovaný acyklický graf.

Strom [Tree] je souvislý les (tedy souvislý neorientovaný acyklický graf). Les s N uzly má $N - 1$ hran. Pokud je graf s N uzly a $N - 1$ hranami souvislý, je to strom.

Vymežíme-li ve stromu jeden jeho uzel jako **kořen [root]**, je tím určena jednoznačně orientace a získáme orientovaný graf - tak zvaný kořenový strom.

Podgraf [Subgraph] grafu $G = (V, E)$ je graf $G' = (V', E')$, pro který je $V' \subseteq V$ a $E' \subseteq E$ a (V', E') je graf. Poznamenejme, že pokud $V' \subseteq V$ a $E' \subseteq E$ může nebýt (V', E') podgrafem. Spolu s každou hranou je nutné vybrat i oba její uzly.

Kostra neorientovaného grafu [Spanning tree or frame] $G = (V, E)$ je jeho podgraf (V, E') , který obsahuje všechny vrcholy původního grafu a je stromem. Jde tedy o acyklický podgraf, obsahující všech N uzlů a $N - 1$ hran. Kostra existuje pro každý souvislý graf. Vynecháním libovolné hrany z kostry získáme graf, který již souvislý není (Ve stromu je každá hrana „mostem“ – jejím odebráním vznikne nesouvislý graf).

Operace

Intuitivně: Operace = Předpis, který dvěma (nebo více) vstupním hodnotám (argumentům) z dané množiny přiřazuje jednoznačně výsledek.

Binární operace na množině $A \equiv$ Ternární relace R na A , která má vlastnost:

Pokud $(x, y, z) \in R$ a současně $(x, y, v) \in R$, potom $z = v$.

Množina A se nazývá **nosič** operace.

Pro operace obvykle užíváme značky připomínající operace s čísly: $+$, $-$, $*$, \cdot , \oplus , \otimes , ...

Obvykle užíváme tak zvanou „infixovou“ notaci: $z = x \oplus y$ místo $(x, y, z) \in \oplus$

Některé důležité vlastnosti binárních operací

Binární operace s nosičem A se nazývá:

Totální [total] (někdy též **úplná [complete]**), pokud pro libovolná $a, b \in A$ existuje $c \in A$ tak, že $a \oplus b = c$. Totální operace tedy musí mít výsledek pro libovolnou dvojici argumentů.

Asociativní [associative], pokud pro libovolná $a, b, c \in A$ platí $a \oplus (b \oplus c) = (a \oplus b) \oplus c$.

Komutativní [commutative], pokud pro libovolná $a, b \in A$ platí $a \oplus b = b \oplus a$.

S neutrálním prvkem [with neutral element] ε , pokud pro libovolné $a \in A$ platí $a \oplus \varepsilon = \varepsilon \oplus a = a$.

S inverzním prvkem [with inverse element], pokud ke každému $a \in A$ existuje $a' \in A$ takové, že platí $a \oplus a' = \varepsilon$, kde ε je neutrální prvek.

Množina s operací, která má všechny zmíněné vlastnosti s výjimkou komutativnosti se nazývá **grupa**. Pokud v grupě platí i komutativní zákon, hovoříme o Abelově grupě. Grupy jsou příkladem struktur, které se v matematice vyskytují často. Celá čísla, racionální čísla, reálná čísla i komplexní čísla tvoří Abelovu grupu vzhledem k operaci sčítání. Ne však vzhledem k operaci násobení (proč?). Permutace konečné množiny a euklidovské pohyby v rovině tvoří grupu vzhledem k operaci skládání. Tyto dvě grupy nejsou Abelovy.

Struktury

V matematice a informatice často potřebujeme vyšetřovat myšlenkové konstrukce, kdy na množině je několik relací (některé z nich mohou být operacemi), které spolu různým způsobem vzájemně souvisejí. Téměř vždy vystačíme s konečným (obvykle nevelkým) počtem takovýchto relací.

Příklady:

- Čísla (z nějakého číselného oboru), relace „být větší nebo rovno“, operace „sčítání“ a „násobení“ čísel.
- Programy pro daný počítač, relace „být složitější“, operace „skládání programových modulů“.

Matematicky se takovéto konstrukce nazývají struktury a definují takto:

Struktura \equiv Množina (nosič struktury) a konečná množina relací na této množině. $\mathbf{S} = (A, \{R_j\}_{j \in J})$.

Relace ve struktuře mohou mít různou aritu. Některé z nich mohou být operacemi.

Nejčastěji se budeme setkávat se strukturami určenými jednou relací (preferencí) a jednou operací skládání prvků této množiny. Tedy $\mathbf{S} = (A, \geq, \oplus)$, kde \geq je obvykle slabé uspořádání na A a \oplus operace skládání prvků z A .

Důležitá je vzájemná souvislost této preferenční relace a operace skládání. Často totiž je třeba, aby nezáleželo na pořadí skládání nebo aby vylepšením části se nepohoršil celek.

Necht' \geq je slabé uspořádání na množině A a \approx je jím generovaná relace ekvivalence a necht' \oplus je totální operace na A . Říkáme, že operace \oplus je:

slabě asociativní, jestliže pro libovolné $a, b, c \in A$ platí $a \oplus (b \oplus c) \approx (a \oplus b) \oplus c$;

slabě komutativní, jestliže pro libovolné $a, b \in A$ platí $a \oplus b \approx b \oplus a$;

zprava monotónní [right monotone], jestliže pro libovolné $a, b, c \in A$ ze vztahu $a \geq b$ vyplývá, že platí také $a \oplus c \geq b \oplus c$;

zleva monotónní [left monotone], jestliže pro libovolné $a, b, c \in A$ ze vztahu $a \geq b$ vyplývá, že platí také $c \oplus a \geq c \oplus b$;

monotónní [monotone], je-li současně zprava i zleva monotónní.

Archimédův axiom

Další důležitou vlastností takovýchto struktur je, zda dostatečně častým přidáváním drobných vylepšení můžeme překonat jakoukoliv počáteční nevýhodu či nikoliv.

„*Gutta cavat lapidem non vi sed saepe cadendo*“. Toto známé latinské přísloví říká zhruba to, že často padající kapka provrtá kámen ne svou silou, ale tím, že padá často. To platí i pro čísla. Přidáváme-li kladné číslo k nějakému základu přičítáním dosti dlouho, po čase převýšíme jakoukoliv hranici, zadanou na počátku. V životě to někdy platí též, někdy však ne. Případy, kdy to neplatí, jsou známy. Některé věci, například život či zdraví, nelze vyrovnat třeba penězi, byť by jich bylo jakkoliv mnoho. Strukturám, které mají tyto vlastnosti říkáme archimédovské.

Archimédův axiom nyní formulujeme přesněji: Necht' \geq je slabé uspořádání na množině A , $>$ jím generované ostré slabé uspořádání a necht' \oplus je totální operace na A . Říkáme že slabé uspořádání \geq je archimédovské vzhledem ke skládání \oplus , pokud platí: je-li $a > b$ a $c > d$, potom existuje přirozené číslo N tak, že $b \oplus N \circ c \geq a \oplus N \circ d$, kde symbol $N \circ x$ je pro libovolné $x \in A$ a libovolné přirozené číslo N definován (indukcí) takto: $1 \circ x = x$, $(N + 1) \circ x = (N \circ x \oplus x)$ pro libovolné přirozené číslo N .

Archimédův axiom lze formulovat i několika jinými ekvivalentními způsoby. Jeden z nich je například tento:

Aritmetickou (nebo též standardní) **posloupností** v A se nazývá každá nekonečná posloupnost prvků $a_1, a_2, \dots \in A$, taková, že $a_{j+1} = a_j \oplus b$, kde $b \in A$ je pevně zvolený prvek. Posloupnost se nazývá omezená, pokud existují prvky $d, h \in A$, tak, že $h > a_j > d$ pro všechna přirozená čísla $j = 1, 2, \dots$.

Archimédův axiom je ekvivalentní s požadavkem, aby neexistovala žádná nekonstantní omezená aritmetická posloupnost. Zda v konkrétní posuzované situaci Archimédův axiom platí či nikoliv, musíme posoudit případ od případu. Jde o to uvážít, zda dostatečně mnoho malých přínosů vyrovná či převáží jeden velký deficit, nebo zda mnoho slabých tušení převáží jeden jakkoliv silný důvod.

Algebry

Strukturu, v kterých uvažujeme převážně pouze operace (nikoliv jiné typy relací) nazýváme obvykle **algebra**.

V algebře můžeme uvažovat jednu operaci (například grupa nebo algebra řetězců nad danou abecedou s operací skládání řetězců), dvě (čísla s operacemi „+“ a „·“, čtvercové matice téhož řádu s operacemi sčítání a násobení) i více operací (celá čísla s operacemi sčítání, násobení, největší společný dělitel a nejmenší společný násobek, relační Coddova algebra) a jejich vzájemné vztahy.

Teorie

V matematice i informatice se často setkáváme s tím, že řadu objektů situací s podobnými vlastnostmi lze popsat tímž modelem. Bývá to zpravidla struktura, u které prvky jejího nosiče a relace mohou být různé, jejich pro nás důležité vlastnosti jsou však tytéž nebo obdobné.

Odvozovat pro každý ze třídy podobných struktur analogickými postupy nové poznatky znovu by bylo plýtváním prací.

Proto je v současné matematice běžný tento postup: Nosič struktury se nespecifikuje konkrétně, ale obecně, jako jakákoliv množina. Právě tak se nedefinují konkrétně relace (a operace), ale pouze se stanoví jejich arita. Specifikují se základní vztahy mezi nosičem a relacemi ve struktuře formou logických formulí, jejichž pravdivost se předpokládá a nedokazuje, tak zvaných „**axiomů**“ **teorie**. A struktura mající tyto vlastnosti se pojmenuje.

Takovéto uspořádané dvojici (S, A) , kde S je struktura a A konečná množina axiomů (logických formulí) se nazývá **teorie**.

Současná matematika pak postupuje tak, že hledá platná tvrzení, které lze odvodit z axiomů teorie a tvrzení, které již z těchto axiomů byly odvozeny. To co je dokázáno v dané teorii pak automaticky platí pro všechny struktury, které axiomy teorie splňují.

Pro odvození matematického tvrzení pak stačí:

1. Zjistit, že jde o tvrzení platné v nějaké teorii.
2. Zjistit, že námi vyšetřovaná struktura splňuje axiomy dané teorie.

Současnou matematiku lze charakterizovat jako vědu o tautologiích na strukturách, platných v dané teorii.

Například: Teorie grup je vymezena algebrou (X, \oplus) a axiomy: $A = \{(1) \text{ totálnosti, } (2) \text{ asociativnosti, } (3) \text{ existence neutrálního prvku, } (4) \text{ existence inverzního prvku ke každému prvku}\}$, platí pro celá čísla se sčítáním, racionální, reálná i komplexní čísla se sčítáním, permutace konečné množiny se skládáním, pohyby v rovině a libovolném n -rozměrném prostoru, geometrické podobnosti, prostá zobrazení konečné množiny do sebe se skládáním, pro projekce v relačních

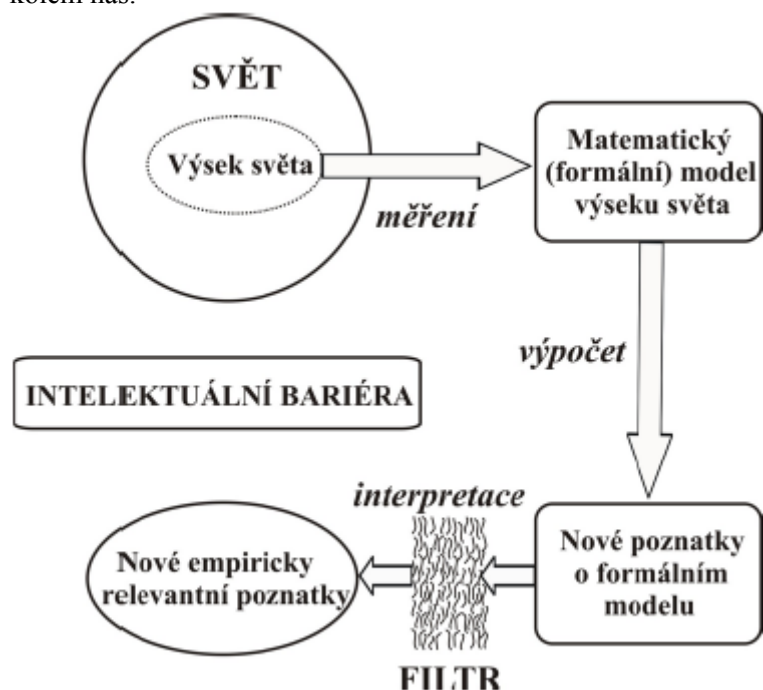
databázích a pro řadu dalších důležitých struktur. Stačí je dokázat pouze jednou pro teorii grup a ověřit, že dané konkrétní struktury splňuje axiomy **A** teorie grup.

Další příklady matematických teorií(některé z mnoha): grupoidy, Abelovy grupy, cyklické grupy, algebraická tělesa, okruhy, svazy, dobře uspořádané množiny, metrické prostory, topologické prostory, Neexistuje jediná ucelená teorie IC. Informatika čerpá z řady různých teorií.

3. MĚŘENÍ

Měření je nástroj pro formální popis vlastností výseku světa, který nás zajímá.

Výsledky měření je možné zpracovávat prostředky matematiky a informatiky a dojít tak k novým poznatkům o světě kolem nás.



Měření nebývá určeno jednoznačně. Zpravidla je zde určitý stupeň volnosti.

Interpretovat lze pouze ty výsledky výpočtu, které jsou nezávislé na tom, jaké měření bylo užito.

Pokud výsledek výpočtu vede k jinému závěru při jiné volbě měření, nemá empirický význam.

Formálně:

Výsek světa je zobrazen empirickou strukturou: $E = (A, \{R_j\}_{j \in J})$,

Kde R_j jsou relace na A , různé arity. Některé z nich mohou být operacemi. Nejčastější případ je struktura $E = (A, \geq, \oplus)$, kde \geq je binární preferenční relace a \oplus operace skládání celku z komponent (ternární relace).

Jí odpovídá formální struktura $F = (\mathbb{R}, \{R'_j\}_{j \in J})$, kde \mathbb{R} je buď množina reálných čísel nebo množina kladných reálných čísel a $\{R'_j\}_{j \in J}$ relace na této množině, nejčastěji pak množina $F = (\mathbb{R}, \geq, +)$ s relací porovnání čísel podle velikosti a operací sčítání čísel.

Měření se nazývá zobrazení μ nosiče A empirické struktury do množiny reálných čísel nebo kladných reálných čísel \mathbb{R} , které **zachovává všechny relace** (a operace).

Tedy ve speciálním případě preferenční relace \geq a operace skládání \oplus musí platit

$x \geq y \Leftrightarrow \mu(x) \geq \mu(y)$, $\mu(x \oplus y) = \mu(x) + \mu(y)$, pro všechna $x, y \in A$.

Takovéto zobrazení struktury E do struktury F se nazývá **homomorfismus** struktur. Homomorfismus nemusí být prostým zobrazením. Různým objektům mohou jako výsledek měření odpovídat různá čísla.

Je-li homomorfismus prostým zobrazením, říkáme mu **isomorfismus**. Isomorfismus struktur má ten význam, že místo toho, abychom vyšetřovali vlastnosti více různých struktur, stačí vyšetřovat jednu z nich. Výsledky lze pak přenést na všechny struktury isomorfní.

Měření [measurement] \equiv **Homomorfní zobrazení μ empirické struktury na strukturu, jejímž nosičem je nějaký číselný obor.**

Zobecněné měření \equiv Formální struktura je **obecnější**, avšak zpracovatelná v rámci nějaké matematické teorie a reprezentovatelná informačními nástroji (vektory, matice, ...).

Hodnota $\mu(x)$ funkce μ pro daný empirický objekt x se nazývá **míra** [measure] x . Míra určuje hodnotu nějakého atributu (měřitelné vlastnosti) objektu x .

Měření nebývá určeno jednoznačně. Při dvou různých měřeních nemůže však být vztah mezi mírami zcela libovolný.

Funkce f , definovaná na $\mu(A)$ se nazývá **přípustná transformace** [admissible transformation] měř, pokud z toho, že μ je měření na struktuře E plyne, že i zobrazení η , definované vztahem $\eta(x) = f(\mu(x))$ je také měření (to je homeomorfismem) E do F .

Měření spolu s množinou (grupou) přípustných transformací \equiv **měřicí stupnice** [scale].

Nejčastější měřicí stupnice (Stevensova hierarchie)

ABSOLUTNÍ [absolute]

Hodnota míry je dána pevně. Přípustnou transformací je pouze identita. Příklad % z celku. V absolutní stupnici lze každý výsledek získaný výpočtem interpretovat jako empiricky smysluplný. Hodnoty měř částí ale nedávají možnost stanovit míru celku.

POMĚROVÁ [ratio]

Hodnota míry je určena jako násobek základní jednotky. Nosičem formální struktury bývají kladná reálná čísla.

Grupu přípustných transformací tvoří všechny funkce $f(x) = c \cdot x$, kde c je kladná konstanta. Příklady: Hmotnost, délka, čas. Ale také počet prvků konečné množiny (může být udán v tisících, kopách, tuctech, nomo-stránkách znaků ...). Výsledky aritmetických operací s mírami bývají „téměř vždy“ interpretovatelné v empirickém světě. Výjimky jsou řídce (sudá či lichá délka, ...). Patrně nejvhodnější měřicí stupnice (pokud ji lze realizovat měřením). Umožňuje zjistit míru celku na základě míry komponent.

INTERVALOVÁ [interval]

Proměnný počátek (nula) a jednotka. Nosičem formální struktury bývají všechna reálná čísla. Grupu přípustných transformací tvoří všechny funkce $f(x) = c \cdot x + d$, kde $c > 0$ a d jsou reálná čísla. Příklad: Teplota ($^{\circ}\text{C}$, $^{\circ}\text{F}$, K).

Lze interpretovat rozdíl hodnot míry, ne však poměr či % (Týž rozdíl teplot má smysl. Poloviční teplota nikoliv). Empirický význam má aritmetický průměr měř, ne však geometrický průměr.

ORDINÁLNÍ [ordinal]

Rozhodující je pouze pořadí (větší hodnota míry = vyšší nebo nižší priorita). Grupu přípustných transformací tvoří všechny **rostoucí** funkce (případně klesající). Příklady: Znamka u zkoušky. Volba „bez výhrad - drobné nedostatky - vážné nedostatky - téměř nevyhovuje - zcela nevyhovuje“ v dotazníku. Interpretovat lze pouze ty výsledky výpočtu, které závisí pouze na pořadí, ne na aritmetických operacích. Tedy empirický význam má maximum, medián, ale již ne aritmetický průměr hodnot měř. Méně výhodné pro měření. V řadě důležitých situací však jsou splněny podmínky pro měření v této stupnici, ne však podmínky pro měření v poměrové stupnici.

NOMINÁLNÍ [nominal]

Čísla užitá pouze pro označení kategorie. Nemají význam priorit ani je nelze užít pro výpočet. Grupa přípustných transformací je tvořena všemi **prostými** funkcemi. Příklad: Číslo domu v adrese. Interpretovat lze pouze shodu a neshodu měř. Případně počet objektů s danou hodnotou míry (modus).

Společný rys:

Čím „bohatší“ je množina přípustných transformací, tím „chudší“ je množina poznatků o formální struktuře (číslech), které lze interpretovat jako smysluplné výroky o empirické struktuře (poznatky o světě kolem nás).

Nerespektování typu měřicích stupnic vede k závažným chybám při interpretaci poznatků získaných výpočtem.

Další časté měřicí stupnice:

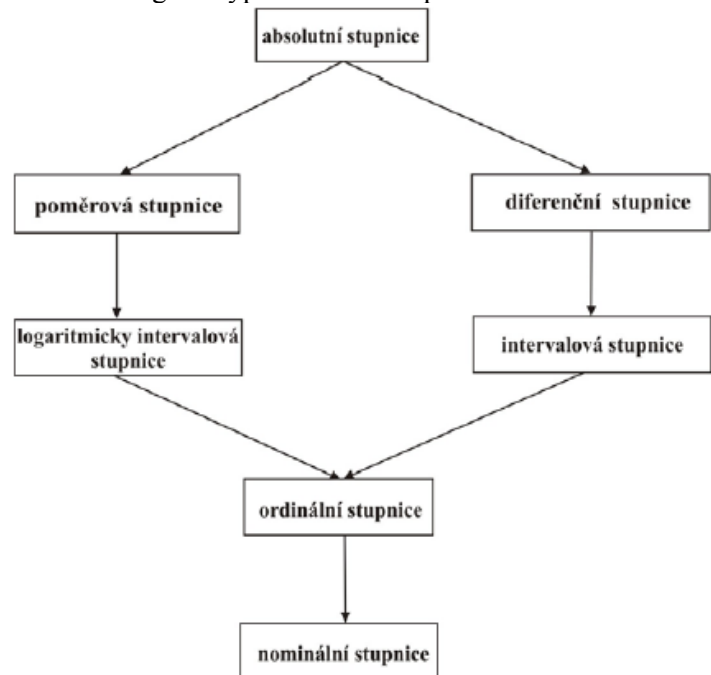
DIFERENČNÍ

Měří odchylku. Grupa přípustných transformací $f(x) = x + d$.

LOGARITMICKY INTERVALOVÁ

Přípustné transformace: $f(x) = \beta \cdot 10^{\alpha x}$, kde $\alpha > 0$ a $\beta > 0$ jsou reálná čísla. Příklady: Hluk v decibelech. Jasnost hvězd na obloze.

Hasseho diagram typu měřicích stupnic:



Pro určení měření jsou důležité podmínky pro existenci měření v dané stupnici. Tak zvané **representační věty**.

Pro ordinální měřicí stupnici (Birkhoff a Milgram):

Věta: Nutná a postačující podmínka pro existenci měření v ordinální měřicí stupnici pro empirickou strukturu $E = (A, \geq)$ je aby platilo současně:

1. Relace \geq je slabým uspořádáním na množině A ,

2. A obsahuje nejvýše spočetnou podmnožinu $B \subset A$, hustou v A , vzhledem k \geq . Množina B je v A hustá, pokud pro každé dva prvky $x, y \in A$, pro které je $x > y$ existoval prvek $z \in B$ takový, že $x \geq z \geq y$.

Podmínka 2 je v informatických aplikacích splněna téměř vždy, protože množina všech konečných řetězců nad konečnou abecedou je spočetná. V informatice pracujeme zpravidla pouze s konečnými a spočetnými množinami. Výjimku tvoří pouze množina všech jazyků nad danou abecedou a tedy množina všech možných „problémů“. Ta má mohutnost c .

Podstatným omezením je splnění podmínky 1. Pokud preferenční relace má neporovnatelné prvky a je tedy pouhým kvaziuspořádáním, ordinální měření (jediným číslem) sestavit nelze. V tomto případě je dimenze n preferenční relace větší než jedna. Preference pak můžeme vyjádřit zobecněným měřením. Sestrojíme n topologických uspořádání, které \geq charakterizují. Pro každé toto uspořádání sestrojíme měření v ordinální stupnici. Výsledkem zobecněného měření bude pak míra vyjádřená n -členným vektorem $\mu(x) = (\mu_1(x), (\mu_2(x), \dots, (\mu_n(x))$.

$x \geq y \Leftrightarrow \mu_j(x) \geq \mu_j(y)$ pro všechna $j = 1, 2, \dots, n$.

Jednoznačnost ordinárního měření:

Míry získané různými měřeními v ordinální stupnici spolu navzájem souvisejí tak, že jsou určeny až na transformaci libovolnou rostoucí funkcí. Jsou-li μ a η ordinární dvě míry, platí $\eta(x) = f(\mu(x))$, kde f je rostoucí funkce a pro libovolnou rostoucí funkci f a ordinární míru μ platí, že míra η , určená vztahem $\eta(x) = f(\mu(x))$ určuje další (stejně hodnotné) měření ordinárního typu.

Při ordinárním měření u kterého je kromě preference \geq bráno v úvahu i slučování \oplus je třeba brát v úvahu, že nelze automaticky předpokládat platnost vztahu $a \approx b \Rightarrow c \oplus a \approx c \oplus b$, ani vztahu $a \geq b \Rightarrow (c \oplus a \geq c \oplus b \wedge a \oplus c \geq b \oplus c)$.

Tyto vztahy jsou pro praktické použití důležité (oprávněně očekáváme, že rovnocenná náhrada části nebo vylepšení části celku nezhorší hodnocení celku). Podmínky, které musí empirická struktura splňovat, aby bylo možné realizovat měření s těmito vysoce žádanými vlastnostmi odvodili Zuse a Vaníček.

Pro poměrovou měřicí stupnici (Roberts a Luce):

Zde je kromě preferenční relace třeba brát v úvahu i operaci skládání \oplus .

Věta: Nutná a postačující podmínka pro existenci měření v ordinální měřicí stupnici pro empirickou strukturu $E = (A, \geq, \oplus)$ je aby platilo současně:

1. Relace \geq je slabým uspořádáním na množině A ,

2. Operace \oplus je slabě asociativní vzhledem k \geq ,

3. Operace \oplus je monotónní vzhledem k \geq ,

4. Operace \oplus je archimédovská vzhledem k \geq (je splněn Archimédův axiom).

Má-li být nosičem formální struktury množina kladných reálných čísel je třeba ještě požadovat, aby platilo navíc ještě

5. Pro všechna $a, b \in A$ je $a \oplus b \geq a$.

Podmínky pro existenci měření v poměrové stupnici jsou velmi tvrdé a při aplikaci měření v informatice často nebývají splněny. V tom případě bývá nutné vystačit s měřením v ordinální stupnici, za cenu větších problémů s interpretací výsledků zpracování měřených hodnot, získaných prostředky informatiky.

Jednoznačnost poměrového měření:

Míry získané různými měřeními v ordinální stupnici spolu navzájem souvisejí tak, že jsou určeny až na kladný násobek. Jsou-li μ a η dvě poměrové míry, platí $\eta(x) = a \cdot \mu(x)$, kde $a > 0$ a pro libovolné $a > 0$ poměrovou míru μ platí, že míra η , určená vztahem $\eta(x) = a \cdot \mu(x)$ určuje další (stejně hodnotné) měření poměrového typu.

Základní a odvozené míry

Základní míry [base measures] jsou míry získané přímo z empirické situace.

Odvozené míry [derived measures] jsou získány výpočtem podle vzorce či algoritmu ze základních nebo již odvozených měř (jedné nebo několika).

Příklady odvozených měř:

$Hustota_poruch_systému = Počet_pozorovaných_poruch / Doba_pozorován_v_hodinách$

$Hustota_chyb_V_programu = Počet_zjištěných_chyb / Rozsah_programy_v_tisících_příkazů$

Hodnotami měř v aplikacích v informatice bývají pouze:

- Velikost něčeho – kladné reálné číslo (například doba nějakého jevu, náklady, ...) – obvykle v poměrové stupnici.
- Počet prvků množiny (rozsah paměti, počet řádků programu, ...) opět v poměrové stupnici.
- Kategorie do které byl objekt zařazen (obvykle ordinární nebo nominální stupnice).

Pro týž atribut (atribut \equiv měřitelná vlastnost entity) může být k dispozici více měř. Která z nich je základní a která odvozená závisí na zvolené metodě měření.

Příklad: Velikost koule lze měřit:

1. Délkou poloměru R .
2. Plochou povrchu S .
3. Objemem V .

Možné vztahy mezi různými mírami téhož atributu závisí na měřicí stupnici.

Pro měření v ordinární stupnici může být vztah mezi základní a odvozenou mírou pouze $\eta(x) = f(\mu(x))$, kde f je libovolná rostoucí funkce.

Pro měření v poměrové stupnici jsou možné vztahy výrazně užší. Jsou dány výsledkem, který odvodil za omezujících předpokladů Luce, v plné obecnosti pak Vaníček:

Věta: Necht' $((A, \geq, \oplus), (\mathbb{R}^+, \geq, +), \mu)$ a $((A, \geq, \oplus), (\mathbb{R}^+, \geq, \bullet), \eta)$ jsou dvě měření v poměrové stupnici. Potom existují kladná reálná čísla α a β , tak, že $\eta(a) = \alpha \cdot \mu(a)^\beta$, pro všechna $a \in A$.

Odtud lze již snadno odvodit reprezentaci operace „ \bullet “ pro slučování měř v odvozeném měření v odvozeném pomocí obvyklého sčítání měř v základních měř. Příslušný vzorec je: $x \bullet y = (x^{1/\beta} + y^{1/\beta})^\beta$.

Ověřte platnost všech tří vztahů na uvedeném příkladu měření „velikosti“ koule užitím měř R , S a V .

Tento vztah má však i méně zřejmé důsledky. Omezuje například možné vztahy mezi rozsahem softwarového díla a náklady na jeho realizaci nebo minimální dobou, která je na jeho vývoj nutná.

4. VÝROKOVÁ LOGIKA

Formální zavedení

Výroková formule: Máme neprázdnou nejvýše spočetnou množinu A výrokových proměnných.

1. Každá proměnná je výroková formule
2. Když α, β jsou formule, potom $(\neg\alpha), (\alpha\wedge\beta), (\alpha\vee\beta), (\alpha\Rightarrow\beta), (\alpha\Leftrightarrow\beta)$, případně i $(\alpha\oplus\beta), \dots$ jsou formule.
3. Nic jiného než to, co vzniklo pomocí konečně mnoha použití bodů 1 a 2, není výroková formule (Konvence: vnější závorky a závorky vyplývající z priorit $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$ lze vynechat)

Tak zvaná rekurzivní či induktivní definice

Základní pojmy

Ohodnocení \equiv_{DEF} zobrazení A do $\{\text{FALSE}, \text{TRUE}\}$.

Ohodnocení formule se řídí běžnými pravidly pro logické spojky.

Výroková formule s n logickými proměnnými má 2^n možných pravdivostních hodnot v závislosti na ohodnocení proměnných

- Formule je **tautologie** právě tehdy, když je TRUE pro všechna možná ohodnocení proměnných.
- Formule je **kontradikce** právě tehdy, když je FALSE pro všechna ohodnocení.
- Formule je **splnitelná** právě tehdy, když existuje alespoň jedno ohodnocení, ve kterém je TRUE

Matematická logika

Matematická logika se zabývá otázkou, co lze z formulí odvodit bez ohledu na jejich význam, pouze podle struktury.

- Formule ϕ je (sémantickým) **důsledkem** množiny formulí $\Psi = \{\psi_1, \psi_2, \dots, \psi_n\}$ právě tehdy když ϕ má ohodnocení TRUE pro každé ohodnocení proměnných, kde každá z formulí v Ψ je TRUE. Značíme $\Psi \models \phi$.
- Formule ϕ a ψ jsou **tautologicky ekvivalentní** právě tehdy když $\psi \models \phi$ a $\phi \models \psi$. Značíme $\phi \equiv \psi$. Formule jsou pravdivé pro stejná ohodnocení.

Všechny logické spojky

x	y	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15
		F	^		x		y	⊕	∨	↓	↔	¬x	←	¬y	⇒	 	T
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	0	0	1	0	1	0	1	0	1

F0 = kontradikce

F1 = AND, konjunkce

F2 = (inhibice)

F4 = (zpětná inhibice)

F6 = XOR, vylučující nebo

F7 = OR, disjunkce

F8 = NOR, Peirceova šipka (arrow)

F9 = ekvivalence

F10 = negace x

F11 = zpětná implikace

F12 = negace y

F13 = implikace

F14 = NAND, Shefferův škrť (stroke)

Jediné dvě logické spojky, které tvoří jednoprvkový úplný systém jsou NOR \downarrow a NAND $|$. Častěji se užívají NOT, OR a AND (Booleova algebra). Jednu ze spojek OR a AND lze vynechat (de Morganova pravidla)

Úplný systém logických spojek

- Nulární spojky TRUE (tautologie) a FALSE (kontradikce)
- Unární spojky Identita a Negace \neg
- Pro log. funkce dvou proměnných máme celkem 2^4 možných log. funkcí (máme 2^2 řádků v tabulce pravdivostního ohodnocení), potřebujeme tedy vyjádřit 14 funkcí (zbylé dvě jsou opět tautologie a kontradikce).
- Úplný systém log. spojek je množina log. spojek, pomocí které můžeme zapsat všechny logické funkce.

Možný úplný systém log. spojek: \neg, \vee, \wedge

Booleova algebra není minimální (de Morganova pravidla)

Normální formy

Konjunktivní normální forma (CNF)

= konjunkce jednoho nebo konečně mnoha formulí, kde každá z nich je literál nebo disjunkce.

Příklad: $(x \vee \neg y) \wedge (\neg y \vee z) \wedge (x \vee \neg r \vee z)$

Disjunktivní normální forma (DNF)

= disjunkce jednoho nebo konečně mnoha formulí, kde každá z nich je literál nebo konjunkce literálů.

Příklad: $(x \wedge \neg y) \vee (\neg y \wedge z) \vee (x \wedge \neg r \wedge z)$

- Formule v konjunkci u CNF je nazývána **klauzule**. Je to tedy **disjunkce literálů nebo literál**. Zavádíme pojem **prázdná klauzule**, která neobsahuje žádný literál a není splnitelná.
- Pro každou log. formuli existuje tautologicky ekvivalentní DNF formule a také tautologicky ekvivalentní CNF formule. Pro každý Booleovský výraz existuje odpovídající CNF i DNF formule.

Přirozená dedukce

- Odvozovací systém umožňující z dané množiny formulí (**předpokladů**) odvodit **závěry**.

- Pro každou logickou spojku máme I-pravidlo pro zavedení a E-pravidlo pro eliminaci

\neg : I-pravidlo: $(\phi \vdash \alpha) \vdash (\neg \alpha \vdash \neg \phi)$;

E-pravidlo: $\neg(\neg \phi) \vdash \phi$ (pravidlo “dvojitěnegace”).

\wedge : I-pravidlo: $\{\phi, \psi\} \vdash \phi \wedge \psi$.

E-pravidla: $\phi \wedge \psi \vdash \phi$; $\phi \wedge \psi \vdash \psi$.

\vee : I-pravidla: $\phi \vdash \phi \vee \psi$; $\psi \vdash \phi \vee \psi$.

E-pravidlo: $\phi \vee \psi$ a $\phi \vdash \alpha$ a $\psi \vdash \alpha \vdash \alpha$ (“důkaz rozбором případů”).

\Rightarrow : I-pravidlo: $(\phi \vdash \psi) \vdash \phi \Rightarrow \psi$;

E-pravidlo: $(\phi \text{ and } \phi \Rightarrow \psi) \vdash \psi$ (pravidlo “modus ponens”).

- Operací odvození nové formule z množiny formulí S značíme \vdash . Touto operací obdržíme novou množinu formulí, která se skládá buď z předpokladů nebo z formulí odvozených pomocí některých odvozovacích pravidel.

- Říkáme, že odvozená formule β je **log. důsledkem** S a log. vyplývá z S . Značíme $S \vdash^* \beta$ (tranzitivníuzávěr). Někdy pouze $S \vdash \beta$.

- Množina formulí S je **nekonzistentní (rozporná)**, právě tehdy když existuje formule α taková, že současně α a $\neg \alpha$ logicky vyplývají z S . Jinak říkáme, že množina S je **konzistentní (bezrozporná, zdravá)**.

Úplnost výrokové logiky

- Formule ϕ je **sémantickým důsledkem** množiny formulí S , pokud platí, že při každém pravdivostním ohodnocení při kterém jsou všechny formule v S pravdivé, je pravdivá také formule ϕ .

- Formule ϕ je **logickým důsledkem** množiny formulí S , pokud platí lze odvodit pomocí odvozovacích pravidel (existuje její důkaz).

Pokud je množina S bezrozporná, je vše, co je logickým důsledkem i sémantickým důsledkem ($S \models \phi \Rightarrow S \vdash^* \phi$).

Pro výrokovou logiku to platí to i naopak. Vše co je sémantickým důsledkem je i logickým důsledkem (lze to dokázat – tedy $S \vdash^* \phi \Rightarrow S \models \phi$). Těto vlastnosti se říká **úplnost dedukčního systému**.

Rezoluční princip

- Mějme S množinu formulí v nějakém odvozovacím systému a necht' α je formule. Zajímá nás otázka, zda-li α je log. důsledkem S .

- Rezoluční princip je založen na faktu, že $S \vdash \alpha$ právě tehdy když $S \cup \{\neg \alpha\}$ není splnitelná.

- Toto je ekvivalentní známému faktu, že $\alpha \Rightarrow \beta$ a $\neg \alpha \vee \beta$ jsou tautologicky ekvivalentní.

- Rezoluční princip je základem logického programování.

Budeme předpokládat CNF, budeme psát $\{x, \neg y, \neg z, v, \neg w\}$ místo $x \wedge \neg y \wedge \neg z \wedge v \wedge \neg w$.

- Prázdná klauzule bude značena $[]$.

- **Rezoluční princip** spočívá v eliminaci dvou komplementárních literálů z klauzulí: $(x \vee y) \wedge (\neg x \vee z) \Rightarrow y \vee z$.

- Obecně budeme říkat, že D je **rezolventa** klauzulí C_1 a C_2 podle literálu p právě tehdy když existuje literál p takový, že $p \in C_1$, $\neg p \in C_2$ a $D = (C_1 \div \{p\}) \cup (C_2 \div \{\neg p\})$.

- Rezoluční princip spočívá v opakovaném vytváření rezolvent: $R_0(S) = S$, $R_{j+1}(S) = R(R_j(S))$ pro $j = 1, 2, \dots$

Necht' $R^*(S) = \bigcup_{j=1}^{\infty} R_j(S)$. $S = R_0(S) \subseteq R_1(S) \subseteq \dots \subseteq R_k(S) \subseteq \dots$. Protože množina proměnných je konečná, lze vytvořit

jen konečně mnoho disjunkcí a existuje takové n , že $R_{n+1}(S) = R_n(S) = R^*(S)$. Množina $R^*(S)$ obsahuje prázdnou klauzuli právě tehdy když S nebo nějaké $R_k(S)$ obsahuje dvě klauzule $\{x\}$ and $\{\neg x\}$ pro nějakou proměnnou x .

- **Rezoluční princip:** Množina klauzulí S je splnitelná právě tehdy když výsledek aplikování rezolvent neobsahuje prázdnou klauzuli $[]$.

- Rozhodnutí, zda-li formule ϕ je sémantickým důsledkem množiny formulí S potom vede na ekvivalentní úlohu rozhodnout, zda $S \cup \{\neg \phi\}$ je **nesplnitelná**, tj. zda z ní lze rezolučním postupem odvodit prázdnou klauzuli $[]$.

Postup:

1. Pro každou formuli v S nalézt tautologicky ekvivalentní CNF formuli. Provedeme náhradu všech formulí z předpokladů. Získáme množinu disjunkcí, které mají platit současně. Tautologie vynecháme. Pokud je výsledná množina prázdná, potom se skládá jen z tautologií a je tedy splnitelná. Jinak aplikujeme rezoluce (hledáme komplementární literály).

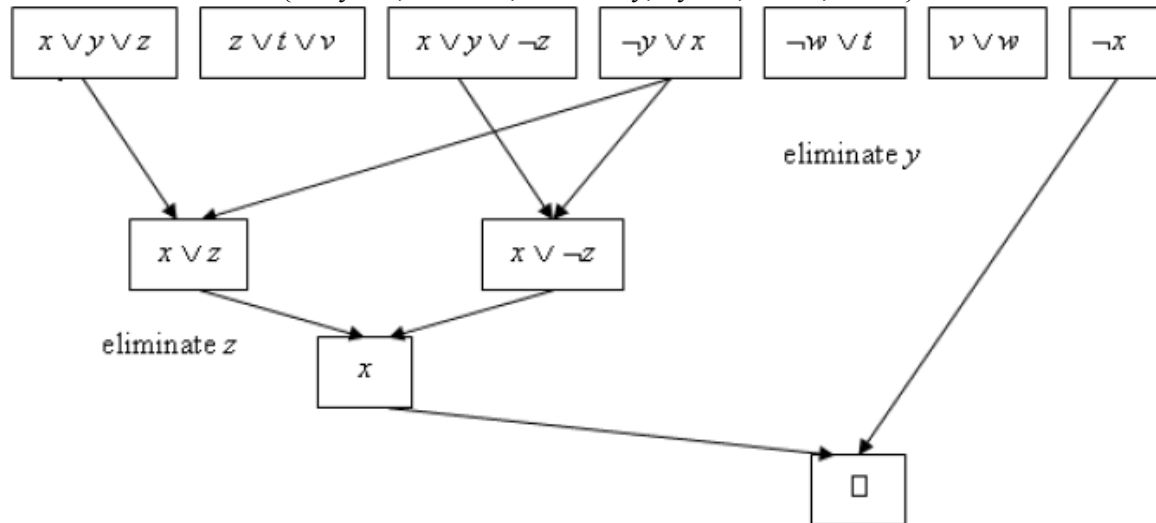
2. Přidáváme postupně (v libovolném pořadí) resolventy.

3. Pokud během postupu nalezneme prázdnou klauzuli, je množina množina S nesplnitelná. Pokud se proces zastaví a $R^*(S)$ prázdnou klausuli neobsahuje, je množina S splnitelná.

Příklad:

- Množina $S = \{x \vee y \vee z, z \vee t \vee v, z \Rightarrow (x \vee y), y \Rightarrow x, w \Rightarrow t, v \vee w\}$
- Je x sémantickým důsledkem? Tj. je pravda, že $S \vdash x$?
- Tuto úlohu převedeme na problém splnitelnosti množiny formulí $S \cup \{\neg x\}$
- Postup:

Převedeme S na CNF: $\{x \vee y \vee z, z \vee t \vee v, \neg z \vee x \vee y, \neg y \vee x, \neg w \vee t, v \vee w\}$



Odvodili jsme prázdnou formuli, množina tedy není splnitelná a tudíž x je sémantickým důsledkem daných klausulí.

5. PREDIKÁTOVÁ LOGIKA [PREDICATE LOGIC]

Přesněji **predikátová logika prvního řádu**.

Formalizuje výroky o vlastnostech předmětů (entit) a vztazích mezi předměty, které patří do dané předmětné oblasti – univerza.

Příklad: Následovník každého lichého přirozeného čísla je sudé číslo. Číslo 7 je liché. \Rightarrow Číslo 8 je sudé.

Predikátové logiky vyšších řádů formalizují vztahy mezi vlastnostmi a vztahy, vztahy mezi vztahy vlastnostmi vztahů a vlastností Výrokovou logiku lze považovat za predikátovou logiku nultého řádu. Formalizuje pouze výroky o entitách.

S výrokovou logikou vědecké disciplíny nevystačí.

S predikátovou logikou prvního řádu se zpravidla vystačí v matematice i informatice.

Jazyk predikátové logiky obsahuje tuto abecedu:

Logické symboly:

1. Nekonečnou spočetnou množinu proměnných (značíme $a, b, \dots, x, y, z, a_1, a_2, \dots$).

2. Logické spojky $\neg, \wedge, \vee, \Rightarrow, (\Leftrightarrow)$.

3. Univerzální kvantifikátor \forall (čti „pro všechna“).

4. Existenční kvantifikátor \exists (čti existuje).

Speciální symboly:

1. **Neprázdnou množinu predikátových symbolů P.** – Různé arity. Vyjadřují vlastnosti a vztahy.

2. **Množinu funkčních symbolů F.** – Různé arity.

3. **Množinu konstantních symbolů K.** Ty lze považovat za funkce arity 0 (nemají žádné proměnné a tedy mají vždy stejnou hodnotu).

Pomocné symboly:

závorky „(, „)“, čárku „,“. Někdy i jiné typy závorek $[,], \{, \}$.

Poznámka:

Univerzální kvantifikátor \forall lze chápat jako zobecnění konjunkce \wedge , existenční kvantifikátor \exists jako zobecnění disjunkce na množiny, které mohou být i nekonečné.

Gramatika predikátové logiky udává jak vytvářet formule

Term (rekurzivní definice)

1. Každý symbol proměnné je term.

2. Každá konstanta je term.

3. Jsou-li t_1, \dots, t_m termy a f je funkční symbol arity m , potom je i $f(t_1, \dots, t_m)$ term.

4. Nic jiného než to, co vznikne aplikací pravidel 1., 2. a 3. již term není.

Atomická formule

Je predikátový symbol aplikovaný na m termů, kde m je arita predikátového symbolu. $p(t_1, \dots, t_m)$.

Formule (rekurzivní definice)

1. Každá atomická formule je formule.

2. Jsou-li ϕ a ψ formule, pak také $\neg\phi, \phi \wedge \psi, \phi \vee \psi, \phi \Rightarrow \psi, \phi \Leftrightarrow \psi$ jsou formule.

3. Je-li ϕ formule, potom i $(\forall\phi)$ a $(\exists\phi)$ jsou formule.

4. Nic jiného než to, co vznikne aplikací pravidel 1., 2. a 3. již formule není.

Závorky lze vynechat, pokud jsou zbytečné vzhledem k obvyklým preferenčním pravidlům pro logické spojky. Vnější závorky se též vynechávají.

Výskyt proměnné x ve formuli A je vázaný, jestliže je součástí nějaké podformule $\forall x B(x)$ nebo $\exists x B(x)$ formule A .

Proměnná x je vázaná ve formuli A , má-li v A vázaný výskyt.

Výskyt proměnné x ve formuli A , který není vázaný, nazýváme **volný**.

Proměnná x je volná ve formuli A , má-li v A volný výskyt.

Formule, v níž každá proměnná má buď všechny výskyty volné nebo všechny výskyty vázané, se nazývá **formulí s čistými proměnnými**.

Formule se nazývá **uzavřenou**, neobsahuje-li žádnou volnou proměnnou. Formule, která obsahuje aspoň jednu volnou proměnnou se nazývá **otevřenou**.

Uzavřená formule se nazývá **větou** [sentence].

Příklady zápisu výroků v predikátové logice:

Univerzum je množina všech lidí.

Nikdo, kdo není zapracován (P), nepracuje samostatně (S).

Ne každý talentovaný (T) spisovatel (Sp) je slavný (Sl).

Někdo je spokojen (Sn) a někdo není spokojen.

Někteří chytrí lidé (Ch) jsou líní (L).

Interpretace

$\forall x (\neg P(x) \Rightarrow \neg S(x)).$

$\neg \forall x ((T(x) \wedge Sp(x)) \vee Sl(x)).$

$\exists x Sn(x) \wedge \exists x \neg Sn(x).$

$\exists x (Ch(x) \wedge L(x)).$

Pro to, abychom rozhodli zda je daná formule pravdivá či ne (má hodnotu TRUE či FALSE), je třeba mít vymezeno univerzum a vědět co znamenají všechny v ní užitě predikáty, funkční symboly a konstanty. Takovému přiřazení říkáme **interpretace**. Formálně je interpretace dvojice (U, I) , kde U je neprázdná množina zvaná univerzum, I je zobrazení které:

- Každé konstantě přiřazuje prvek univerza.
- Každému n -árnímu funkčnímu symbolu přiřazuje funkci n proměnných na univerzu s hodnotami z univerza.
- Každému n -árnímu predikátu přiřazuje n -ární relaci na univerzu, tvořenou všemi n -ticemi prvků univerza, pro které je daný predikát pravdivý.

Pravdivost formule predikátového počtu lze vyhodnotit pouze na základě dané interpretace a daného ohodnocení (valuace) všech volných proměnných.

Přitom:

Výrok $\forall x \phi(x)$ je pravdivý právě když $I(\phi)$ je celé univerzum U .

Výrok $\exists x \phi(x)$ je pravdivý právě když $I(\phi)$ je neprázdná podmnožina univerza.

Formule A je splnitelná v interpretaci I , jestliže existuje aspoň jedno ohodnocení e volných proměnných takové, že vznikne pravdivý výrok.

Formule A je pravdivá v interpretaci I , , jestliže pro všechna možná ohodnocení e volných proměnných vznikne pravdivý výrok.

Formule A je splnitelná, jestliže existuje interpretace I , ve které je splnitelná, tj. jestliže existuje interpretace I a ohodnocení volných proměnných e takové, že vznikne pravdivý výrok. Taková dvojice (I, e) interpretace I a valuace e se nazývá **model** formule.

Formule A je tautologií je-li pravdivá v každé interpretaci.

Formule A je kontradikcí, jestliže nemá model, tedy neexistuje interpretace I , v která by formule A byla splnitelná.

Pozn.: Zjevně platí, že A je kontradikce, právě když negace A je tautologie.

Model množiny formulí $\{A_1, \dots, A_n\}$ je taková interpretace I v kterém jsou všechny formule splnitelné, tedy interpretace I a ohodnocení e volných v kterém jsou všechny formule volných proměnných), která splňuje všechny formule A_1, \dots, A_n pravdivé.

Sémantická a logická dedukce v predikátovém počtu

Oba typy dedukce se definují obdobně jako ve výrokové logice.

Uzavřená formule (věta) ϕ je sémantickým důsledkem (též tautologickým důsledkem – značíme \models) množiny uzavřených formulí S právě tehdy, když každý model S je také modelem ϕ .

Pro **logickou dedukci** (\vdash) přebereme I -pravidla a E -pravidla výrokové logiky a přidáme k nim přirozená pravidla pro kvantifikované výroky.

Jde především o pravidla $\phi(t) \Rightarrow \exists x \phi(x)$ a $\neg \forall x \phi(x) \Leftrightarrow \exists x \neg \phi(x)$

Pro predikátovou logiku platí rovněž **věta o úplnosti**.

Přirozená dedukce je **bezrozporná** (vše co se dá logicky odvodit je i sémantickým důsledkem).

Přirozená dedukce je **úplná**. Vše co je sémantickým důsledkem lze odvodit i logicky.

Tedy $S \models \alpha$ tehdy a jen tehdy když $S \vdash \alpha$. Důkaz tohoto tvrzení však není snadný.

Platí následující důležitá tvrzení:

Větu lze odvodit bez předpokladů, právě když je tautologií.

Množina vět je bezrozporná, právě když je splnitelná (tedy má nějaký model).

Množina vět je rozporná, právě když z ní vyplývá kontradikce.

Mezi výrokovým a predikátovým počtem je následující podstatný rozdíl:

Každý **jazyk predikátové logiky má nekonečně mnoho možných interpretací** (už jenom universum lze stanovit nekonečně mnoha způsoby). Tím se liší od jazyka výrokové logiky, který má vždy jen konečný počet interpretací – ohodnocení TRUE – FALSE výrokových symbolů (jazyk výrokové logiky pracující s n výrokovými symboly má různých 2^n interpretací, je tedy možné, i když časově náročné, ověřit pravdivost všech iterpretací).

Tautologičnost formulí predikátové logiky nelze proto sémanticky dokazovat tak, že ukážeme, že každá možná interpretace jazyka je i modelem dané formule. Tímto způsobem jsme postupovali ve výrokové logice, když jsme zjišťovali pravdivostní hodnotu formule pro každou kombinaci pravdivostních hodnot výrokových symbolů. I zde při velkém n narážel tento postup na exponenciální růst výpočetní složitosti. U predikátového počtu nelze tento způsob užít ani teoreticky, bez ohledu na rostoucí časové nároky na výpočet.

Resoluční princip v predikátové logice

Zavedme některé pojmy:

Literál je atomická formule nebo její negace.

Komplementární literály je dvojice literálů z nichž každý je negací druhého.

Klausule je věta (formule bez volných proměnných), taková, že obsahuje pouze univerzální kvantifikátory na začátku a následuje **disjunkce** konečného počtu literálů.

Zavedeme následující úmluvu: U klausule budeme univerzální kvantifikátory proměnných vynechávat. Protože u disjunkce nezáleží na pořadí, budeme klausule zapisovat pouze jako množiny jejích literálů. Tedy například místo tří

klausulí $P(x, y), \forall a (\neg Q(a) \vee R(a, x) \vee S(f(a), a)), \forall a \forall b S(a, b) \vee \neg Q(b)$ budeme psát pouze množinu tří množin literálů $\{P(x, y)\}, \{\neg Q(a), R(a, x), S(f(a), a)\}, \{S(a, b), \neg Q(b)\}$.

Prázdná klausule neobsahuje žádné literály a je tedy kontradikcí. Obvykle se značí symbolem \square , někdy též **F**.

Princip rezoluční metody u predikátové logiky je analogický jako u výrokové logiky. Je však komplikovanější, protože není k dispozici přímá analogie k konjunktivně disjunktivní normální formě.

Postupně odvozujeme z daných klausulí resolventy tak, že vypouštíme dvojice komplementárních literálů. Původní klausule ponecháme.

Postup je založen na tom, že tautologicky platí $(\phi \vee \eta) \wedge (\psi \vee \neg \eta) \Rightarrow (\phi \vee \psi)$.

Ukážeme to na příkladech:

Příklad 1:

Resolventa klausulí $C1 = \{P(x, y, z), \neg Q(x, y)\}$ a $C2 = \{\neg P(x, y, z), \neg R(x)\}$, kde x, y, z jsou proměnné je klausule $C = \{\neg Q(x, y), \neg R(x)\}$.

Komplementární literály $P(x, y, z)$ a $\neg P(x, y, z)$ lze vynechat. Množiny klausulí $\{C1, C2\}$ a $\{C1, C2, C\}$ jsou tautologicky ekvivalentní. Mají tytéž modely.

Abychom to dokázali, stačí ukázat, že pro každou interpretaci (U, I) , kde $C1$ a $C2$ jsou pravdivé je pravdivé i C .

Nechť a, b, c jsou libovolné konstanty z U . Substitujeme-li a za x , b za y a c za z (označme jako $x/a, y/b, z/c$) odvodíme, že $\{\neg Q(a, b), \neg R(a)\}$ je pravdivé a tedy C je pravdivé v interpretaci (U, I) .

Příklad 2 (již bez podrobného zdůvodnění)

Resolventa klausulí $\{P(x, y, z), \neg Q(x, y)\}$ a $C2 = \{\neg P(a, b, z), \neg R(a)\}$ získaná substitucí $x/a, y/b$ je $\{\neg Q(a, b), \neg R(a)\}$.

Nalézání komplementárních literálů v množině klausulí lze algoritmitizovat.

Tento postup je užit například při ověřování, zda dané tvrzení vyplývá z daných předpokladů.

Jde o ověření tautologičnosti implikace $(p1 \wedge p2 \wedge \dots \wedge pn) \Rightarrow q$, tautologicky ekvivalentní s $\neg(p1 \wedge p2 \wedge \dots \wedge pn) \vee q$, tedy s $\neg p1 \vee \neg p2 \vee \dots \vee \neg pn \vee q$. Takováto klausule se nazývá Hornovou klausulí.

Vyhodnocovací proces (tak zvaný inferenční mechanismus) logického programovacího jazyka PROLOG spočívá v odvozování resolvent z Hornových klausulí. Ty reprezentují fakta a pravidla z databáze. Cílem je ověřit formuli danou dotazem, případně nalézt konstanty, pro které je splněna.

Chceme-li rozhodnout zda je splnitelná jakákoliv množina klausulí S , sestrojíme množinu $S1$, tak, že k S přidáme resolventy prvního řádu. Dále přidáme resolventy $S1$, získáme $S2$ a pokračujeme dokud nenastane rovnost $S_n = S_{n+1}$.

Dostaneme množiny $R_0(S) = S, R_{j+1}(S) = R(R_j(S))$ pro $j = 1, 2, \dots$. Platí: $S = R_0(S) \subseteq R_1(S) \subseteq \dots \subseteq R_k(S) \subseteq \dots$. Položme

$$R^*(S) = \bigcup_{j=1}^{\infty} R_j(S)$$

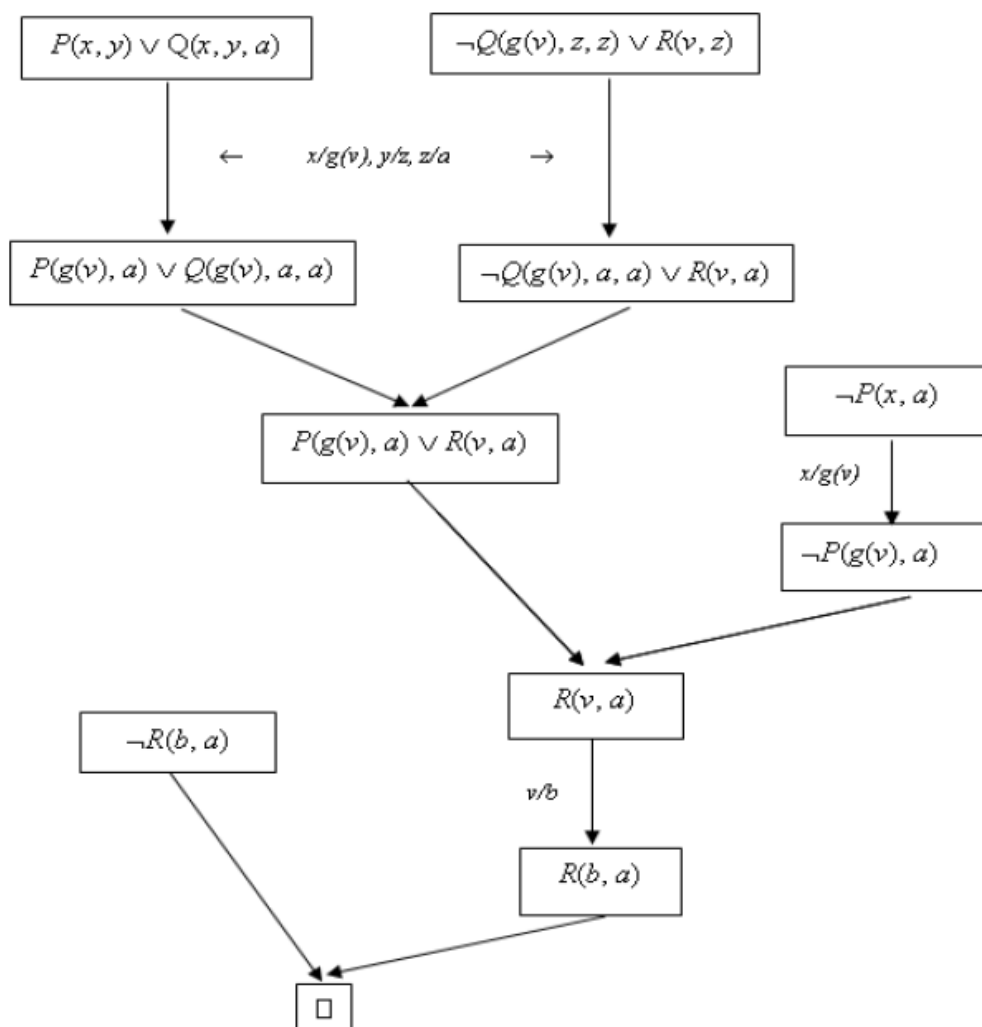
Resoluční princip predikátové logiky říká: Množina S je splnitelná právě když $R^*(S)$ neobsahuje prázdnou klausuli \square .

Chceme-li zjistit zda klausule ϕ je důsledkem (logickým a tedy i sémantickým) množiny klausulí S , vytvoříme množinu $S' = S \cup \{\neg \phi\}$ a zjistíme, zda je splnitelná, či nikoliv. Je-li S' splnitelná ϕ není důsledkem S . Je-li nesplnitelná, je ϕ důsledkem S . To je princip nepřímého důkazu v matematice.

Příklad:

Splnitelnost formulí $S = \{P(x, y), Q(x, y, a)\}, \{\neg Q(g(v), z, z), R(v, z)\}, \{\neg R(b, a), \neg P(x, a)\}$, kde a, b jsou konstantní symboly, x, y, z jsou proměnné:

Sledujte potřebné dosazování konstant za proměnné!



Odvodili jsme prázdnou klausuli. Množina formulí je tedy nespílitelná.

Existuje algoritmický postup jak libovolnou množinu formulí predikátové logiky převést na množinu klausulí.

Lze to provést v těchto po sobě následujících krocích:

1. Přejmenujeme se proměnné tak, aby každý kvantifikátor označoval různou proměnnou. Například $\forall x P(x) \vee \forall x Q(x, a)$ změním na $\forall x \forall y P(x) \vee Q(x, a)$.
2. Spojky $\Rightarrow, \Leftrightarrow$ vyjádříme pouze pomocí \neg, \vee, \wedge užitím tautologických ekvivalencí $\alpha \Rightarrow \beta \equiv \neg \alpha \vee \beta$; $\alpha \Leftrightarrow \beta \equiv (\neg \alpha \vee \beta) \wedge (\alpha \vee \neg \beta)$; ...
3. Zařadíme negace \neg dovnitř až před atomické formule pomocí tautologických ekvivalencí $\neg \exists x \alpha \equiv \forall x \neg \alpha$; $\neg \forall x \alpha \equiv \exists x \neg \alpha$; $\neg(\alpha \vee \beta) \equiv \neg \alpha \wedge \neg \beta$; $\neg(\alpha \wedge \beta) \equiv \neg \alpha \vee \neg \beta$; $\neg \neg \alpha \equiv \alpha$.
4. Zařadíme disjunkce \vee co nehlouběji užitím tautologických ekvivalencí $\alpha \vee (\beta \wedge \gamma) \equiv (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$; $\alpha \vee (\forall x \beta) \equiv \forall x (\alpha \vee \beta)$; $\alpha \vee (\exists x \beta) \equiv \exists x (\alpha \vee \beta)$.
5. Přemístíme univerzální kvantifikátory užitím tautologické ekvivalence $\forall x (\alpha \wedge \beta) \equiv (\forall x \alpha) \wedge (\forall x \beta)$.

Pokud formule neobsahuje existenční kvantifikátory, získali jsme konjunkci klausulí, která je tautologicky ekvivalentní původní formuli.

V případě existenčních kvantifikátorů provedeme tak zvanou **skolemizaci** (název odvozen od norského matematika Thorlafa Skolema). Nahradíme formuli $\exists x P(x)$ formulí $P(a)$, kde a je konstanta. V případě, že předcházejí univerzální kvantifikátory před existenčním, závisí tato konstanta na proměnných univerzálních kvantifikátorů. Musíme tedy užít funkční symbol příslušné arity. Tedy například $\forall x \exists z \forall y P(x, y, z)$ nahradíme $\forall x \forall y P(x, y, c(x))$ a $\forall x \forall y \exists z P(x, y, z)$ nahradíme $\forall x \forall y P(x, y, c(x, y))$. Skolemova „konstanta“ závisí tedy na předchozích univerzálních kvantifikátorech. Je tedy funkčním symbolem arity rovné počtu předchozích univerzálních kvantifikátorů.

Obecně: $\forall x_1, \dots, \forall x_n \exists y \phi(y, x_1, \dots, x_n)$ nahradíme formulí $\forall x_1, \dots, \forall x_n \phi(f(x_1, \dots, x_n), x_1, \dots, x_n)$, kde f je nový funkční symbol arity n . Je-li $n = 0$ užijeme konstantní symbol.

Celý postup ozřejmí následující příklad:

Užitím resoluční metody ověřte správnost následujícího úsudku:

Každý holič holí kohokoliv, kdo se neholí sám.

Žádný holič neholí kohokoliv, kdo se holí sám.

Důsledek: Žádní holiči neexistují.

Převod do predikátové logiky:

Univerzum: Všichni lidé.

$B(x)$ – unární predikát: „člověk je holič“.

$S(x, y)$ – binární predikát “osoba x holí osobu y ”.

Náš úsudek ve formalizovaném tvaru:

$\forall x (B(x) \Rightarrow \forall y (\neg S(y, y) \Rightarrow S(x, y)))$

$\forall x (B(x) \Rightarrow \forall y (S(y, y) \Rightarrow \neg S(x, y)))$

$\neg \exists x B(x)$.

Úsudek bude správný, pokud je nespelnitelná následující množina tří formulí:

$\{\forall x (B(x) \Rightarrow \forall y (\neg S(y, y) \Rightarrow S(x, y))), \forall x (B(x) \Rightarrow \forall y (S(y, y) \Rightarrow \neg S(x, y))), \exists x B(x)\}$.

Tyto formule je třeba transformovat na tautologicky ekvivalentní klausule. Provedeme to standardním algoritmizovatelným postupem, který byl v předchozím odstavci popsán obecně:

Přejmenujeme proměnné a převedeme první dvě formule na klausule. Poslední klausulí je.

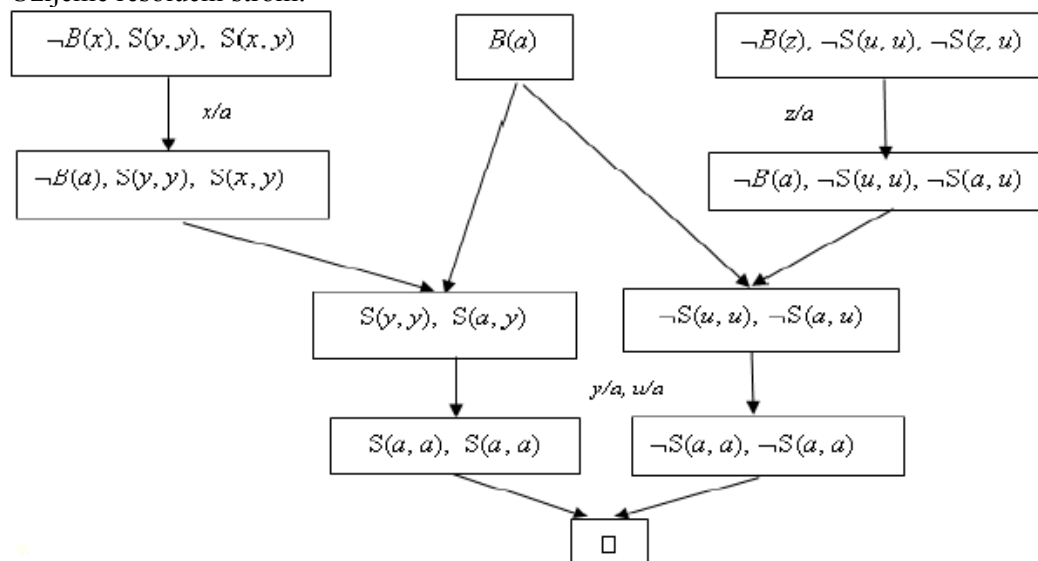
$\forall x (B(x) \Rightarrow \forall y (\neg S(y, y) \Rightarrow S(x, y))) \equiv \forall x (\neg B(x) \vee \forall y ((S(y, y) \vee S(x, y))) \equiv \forall x \forall y (\neg B(x) \vee S(y, y) \vee S(x, y));$

$\forall x (B(x) \Rightarrow \forall y (S(y, y) \Rightarrow \neg S(x, y))) \equiv \forall z (\neg B(z) \Rightarrow \forall u (\neg S(u, u) \vee \neg S(z, u))) \equiv \forall z \forall u (\neg B(z) \vee \neg S(u, u) \vee \neg S(z, u)).$

Poslední klausule obsahuje existenční kvantifikátor Zaměníme jej za klausuli $B(a)$, kde a je Skolemův konstantní symbol.

Úsudek bude správný, pokud bude množina klausulí $S = \{\{\neg B(x), S(y, y), S(x, y)\}, \{\neg B(z), \neg S(u, u), \neg S(z, u)\}, \{B(a)\}\}$ nespelnitelná.

Užijeme resoluční strom:



Náš úsudek byl tedy správný.

6. INFORMACE

Neformálně:

INFORMACE \equiv Poznatek (sdělení), které je **nové** (snižuje neurčitost poznání).

DATA (údaj – datová položka) = ZPRÁVA \equiv konečný řetězec symbolů vybraných z nějaké konečné množiny (abecedy), např.: $\{0,1\}$, $\{a, b, \dots, z\}$.

Data mohou (nemusí) nést informaci po své **interpretaci**.

SÉMANTICKÁ TEORIE INFORMACE

Je dáno **univerzum** U (pevně daná **množina** entit). Jde nám o vlastnosti prvků tohoto univerza. Zda prvek vlastnost má či nemá se určí podle toho, zda do nějaké podmnožiny U patří či nepatří.

POZNATEK (ostrý, či určitý poznatek) o prvku $x \in U \equiv_{DF}$ podmnožina $\delta \subseteq U$

Do podmnožiny δ patří ty a pouze ty prvky, které mají danou vlastnost.

NEOSTRÝ (NEJISTÝ) POZNATEK \equiv_{DF} fuzzy podmnožina U

Funkce příslušnosti $\delta(x)$:
$$\begin{cases} U \rightarrow \{0, 1\} \text{ u ostrých poznatků} \\ U \rightarrow \langle 0, 1 \rangle \text{ u neostrých poznatků} \end{cases}$$

(1) $\delta(x)$ znamená $x \in \delta$,

(0) $\delta(x)$ znamená $x \in U \div \delta$,

(α) $\delta(x)$ znamená příslušnost do δ s mírou jistoty $\alpha \in \langle 0, 1 \rangle$.

$\delta_1 \subseteq \delta_2$ znamená, že poznatek δ_1 je méně nebo stejně obecný než poznatek δ_2 .

Čím „větší“ je množina prvků, které danou vlastnost mají, tím „menší“ (méně určitý) je poznatek o dané vlastnosti.

Všechny poznatky o daném prvku univerza tvoří svaz (tento svaz isomorfní se svazem podmnožin univerza – průseku odpovídá průnik množin, spojení odpovídá sjednocení množin, příslušné částečné uspořádání je množinová inkluze \supseteq).

INFORMACE (ostrá) o prvku $x \in U \equiv_{DF}$ neprázdná množina poznatků $J(x)$ pro které platí:

1. $\delta(x) \in J(x) \Rightarrow \delta \neq \emptyset$, každý poznatek je vlastností aspoň jednoho prvku univerza,

2. $\delta_1(x) \in J(x) \wedge \delta_1 \subseteq \delta_2 \Rightarrow \delta_2(x) \in J(x)$, má-li prvek nějakou vlastnost, má i všechny obecnější vlastnosti (obecnější poznatek informaci nepřináší),

3. $(\delta_1(x) \in J(x)) \wedge (\delta_2(x) \in J(x)) \Rightarrow (\delta_1(x) \cap \delta_2(x) \in J(x))$, mít dvě vlastnosti současně je též poznatek o prvku.

Informace je podrobnější nebo stejně podrobná, obsahuje-li méně nebo stejně obecné poznatky $J_1(x) \subseteq J_2(x) \Leftrightarrow \forall \delta_1 \in J_1 \exists \delta_2 \in J_2 \delta_2 \subseteq \delta_1$, (obecnější poznatky nesou menší informaci než podrobnější poznatky)

Všechny informace o prvku univerza tvoří opět svaz.

NOSIČ informace o prvku x je podmnožina poznatků o prvku x , taková, že pro každý poznatek, který do informace patří existuje poznatek z nosiče, který je méně nebo stejně obecný. Pro určení informace stačí její nosič. Není nutné uvádět všechny poznatky. Nosič může být jednoprvkový, pokud poznatek určuje entitu jednoznačně (klíč). Některé informace nemusí mít žádný jednoprvkový nosič.

Na množině informací o prvku univerza lze vytvářet **koncepty** \equiv **pojmy**.

Množina KONCEPTŮ (POJMŮ) na univerzu $U \equiv_{DF}$ množina L podmnožin univerza taková, že $U \in L$, $\emptyset \in L$ a **L je uzavřená vůči průniku, sjednocení a doplňku.**

Pojmy, které neobsahují jako svoji vlastní podmnožinu žádný pojem kromě prázdné množiny jako svoji část se nazývají **atomy**.

SLOVNÍK POJMŮ \equiv_{DF} množina pojmů z kterých lze všechny pojmy vytvořit užitím operací sjednocení, průniku a doplňku.

Důležité jsou **slovníky pojmů s minimálním a maximálním počtem prvků**.

Maximální slovník = množina všech atomů.

Minimální slovník umožňuje nejúspornější kódování informace daty.

Minimální slovník lze vždy nalézt takový by měl $\lfloor \log_2(A) \rfloor + 1$ prvků

[...] značí celou část čísla a a A je počet atomů (rozměr maximálního slovníku).

Příklad: Studenty plně obsazená posluchárna s 8 řadami lavic, každá má 8 židlí.

Maximální slovník: 64 jednoprvkových, obsahujících jednotlivé studenty - atomy

Minimální slovník: Označení (\check{r} , \check{z}) \check{r} = číslo řady, \check{z} = číslo židle od levého kraje, číslováno vždy od 0 do 7.

podmnožiny (vlastnosti) určené predikáty: $P_1 = \check{r} > 3$, $P_2 = \check{r} \pmod{4} > 1$, $P_3 = \check{r}$ je sudé, $P_4 = \check{z} > 3$, $P_5 = \check{z} \pmod{4} > 1$, $P_3 = \check{z}$ je sudé.

Každého studenta lze popsat šesti odpověďmi ANO/NE na dané predikáty (tedy vytvořit daný pojem – atom z minimálního slovníku pomocí sjednocení, průniků a doplňků. Šesti vlastnostmi však nelze popsat prázdnou množinu studentů.

Shannonova teorie informace

Je ve skutečnost teorie přenosu dat komunikačním kanálem.

Popisuje dopad informace nesené přenášenými daty na snížení neurčitosti našeho poznání (entropie).

Zásada: Současný výskyt dvou nezávislých jevů poskytuje informaci, která je součtem informace o obou jednotlivých jevech.

Jediné spojitě funkce, které mají vlastnost $I(a \cdot b) = I(a) + I(b)$ jsou funkce **$I(p(x)) = -\log_z(p(x))$, pro $z > 1$,**

kde $p(x) \in (0, 1)$ je pravděpodobnost jevu x .

Ty vyjadřují **informační přínos** jevu x .

7. ABECEDY, JAZYKY, GRAMATIKY, PROCEDURY A ALGORITMY

Motivace:

Chceme analyzovat které úlohy lze řešit automatizovaně (= algoritmicky pomocí počítače) a které ne, a jaká jsou omezení pro algoritmická řešení.

K tomu je třeba:

1. Upřesnit co znamená „problém“.
2. Upřesnit co znamená „být řešen algoritmicky“.
3. Protože možnosti reálného počítače jsou málo přehledné, sestavit abstraktní model zařízení s funkcemi analogickými funkcím reálného počítače, avšak tak jednoduchý, aby byl vhodný pro abstraktní úvahu.
4. Aby vytvořený byl vhodný pro abstraktní úvahu.

Základní entity a jevy, které je nutné v modelu vyjádřit:

- Problém
- Algoritmus
- Počítač
- Výpočet

Modely by měly být:

- Matematicky korektní a vhodné pro abstraktní odvozování důsledků (vět).
- Dostatečně dobře odrážet intuitivní vlastnosti modelované oblasti.

ABECEDY A JAZYKY

Jazyk podle Webstrova slovníku: „The body of words and methods of combining words used and understood by considerable community“- „Soubor slov a metod jak slova kombinovat, užitý pro dorozumívání v dané komunitě.“

V informatice: **Abeceda (někdy též slovník) \equiv neprázdná konečná množina symbolů.**

Obvykle aspoň dva prvky. Není již podstatný rozdíl zda 2 nebo více.

Příklady: $\{A, B, C, \dots, Z\}$, $\{\alpha, \beta, \gamma, \dots, \omega\}$, $\{0, 1\}$, $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, *, /\}$,

$\{\text{begin, end, if, then, else, while, repeat, until, for, read, write, } \dots \text{ a další klíčová slova nějakého programovacího jazyka}\}$

Slovo (někdy též věta) nad danou abecedou (slovníkem) \equiv Řetězec prvků abecedy (slovníku) konečné délky.

Řetězec může být i prázdný – prázdné slovo značíme obvykle ϵ .

Značení:

V ... abeceda

V^* ... množina všech slov

V^+ ... množina všech neprázdných slov $= V^* \div \{\epsilon\}$

Příklady:

1. $V = \{0, 1\}$, $V^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, 0000, 0001, \dots\}$,

$V^+ = \{0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, 0000, 0001, \dots\}$,

2. $V = \{a, b, c\}$, $V^* = \{\epsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, aab, aba, \dots\}$

Množina všech slov V^* nad (konečnou) abecedou V je nekonečná spočetná množina.

Mezi prvky množiny V^* je definována operace skládání slov $a_1 a_2 \dots a_m \cdot b_1 b_2 \dots b_n = a_1 a_2 \dots a_m b_1 b_2 \dots b_n$

Jazyk nad danou abecedou \equiv Jakákoliv podmnožina slov (někdy libovolná podmnožina množiny všech vět) nad touto abecedou (slovníkem).

Jazyk může být prázdná, konečná, ale i nekonečná spočetná množina.

Příklady:

Množina všech slov zadané délky. Třeba byte (abeceda, délka 8 – jazyk má 256 slov).

Množina slov nad abecedou $\{0, 1\}$, kde počet jedniček je prvočíslo.

Množina slov nad abecedou $\{0, 1\}$, které mají stejný počet jedniček a nul.

Množina slov nad abecedou $\{a, b, c, \dots, y, z\}$, které obsahují někde řetězec stop.

Množina syntakticky správně vytvořených programů daného programovacího jazyka.

Problém \equiv Otázka, zda dané slovo patří či nepatří do daného programovacího jazyka.

To co považujeme za problém intuitivně lze zpravidla převést na tuto otázku.

Úvahy o mohutnosti (kardinalitě)

• Abeceda je konečná neprázdná množina. Její mohutnost je přirozené číslo.

• Množina všech slov nad abecedou je nekonečná, spočetná. Má mohutnost \aleph_0 .

• Množina všech jazyků je nespočetná. Má mohutnost kontinua. – Jako množina všech podmnožin nekonečné spočetné množiny.

Všech problémů je též $c = 2^{\aleph_0}$.

Všechny (dnes existující i v budoucnu někdy sestrojitelné) počítačové programy v současném intuitivním pojetí lze popsat konečnou posloupností nul a jedniček. Množina všech konečných řetězců nad danou abecedou je nekonečná spočetná množina. Možná vstupní data tvoří rovněž nejvýše spočetnou množinu. Všechna možná užití všech myslitelných

programů tvoří tedy spočetnou množinu. Prosté zobrazení množiny všech možných užití všech myslitelných počítačových programů na množinu všech problémů nemůže existovat. Množiny totiž mají různou mohutnost.

⇒ existují problémy, které dnes ani v budoucnu nelze řešit počítačem (algoritmem).

Provedená úvaha naráží na to, že ukazuje sice existenci takových problémů, ne však nějaký konkrétně.

K dalšímu upřesnění nejprve intuitivně rozlišíme mezi algoritmem a procedurou.

PROCEDURY A ALGORITMY

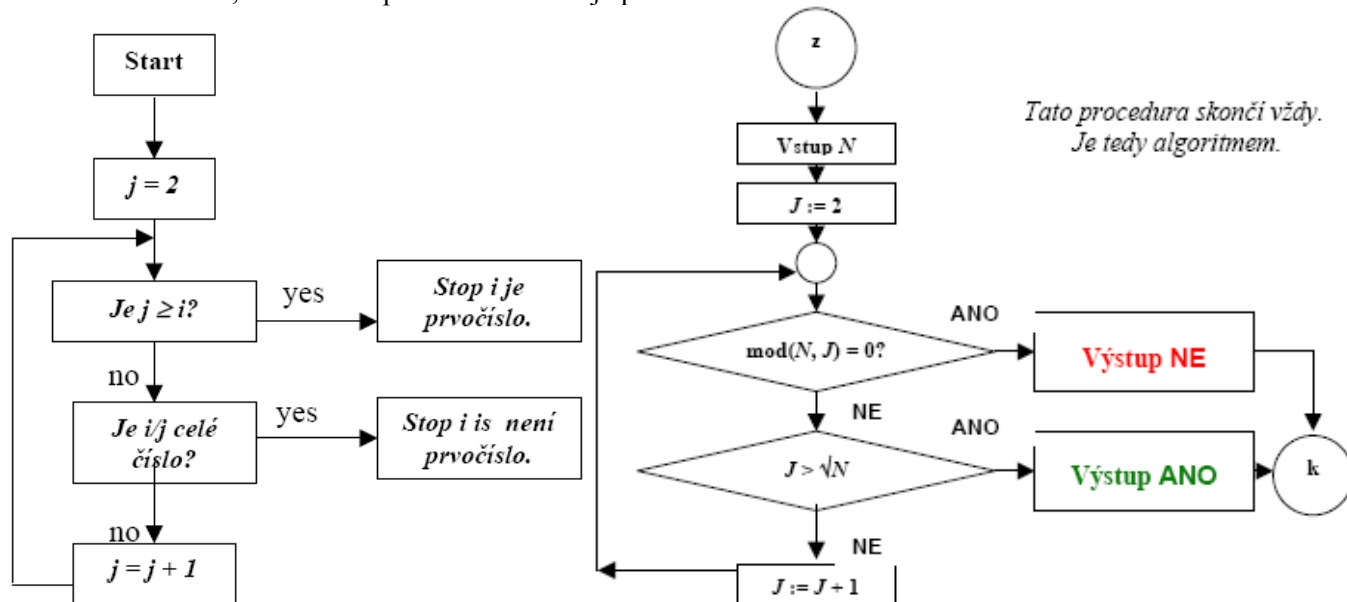
Naivní vymezení pojmu (**definice**):

Procedura je posloupnost jednoduchých kroků proveditelných automaticky (počítačem) pro řešení dané třídy problémů.

Postup musí být diskrétní, deterministický a hromadný.

Algoritmus je procedura, u níž je zaručeno, že po konečném počtu kroků skončí (je finitní).

Příklad 1: Problém, zda zadané přirozené číslo N je prvočíslem či nikoliv.



Někdy

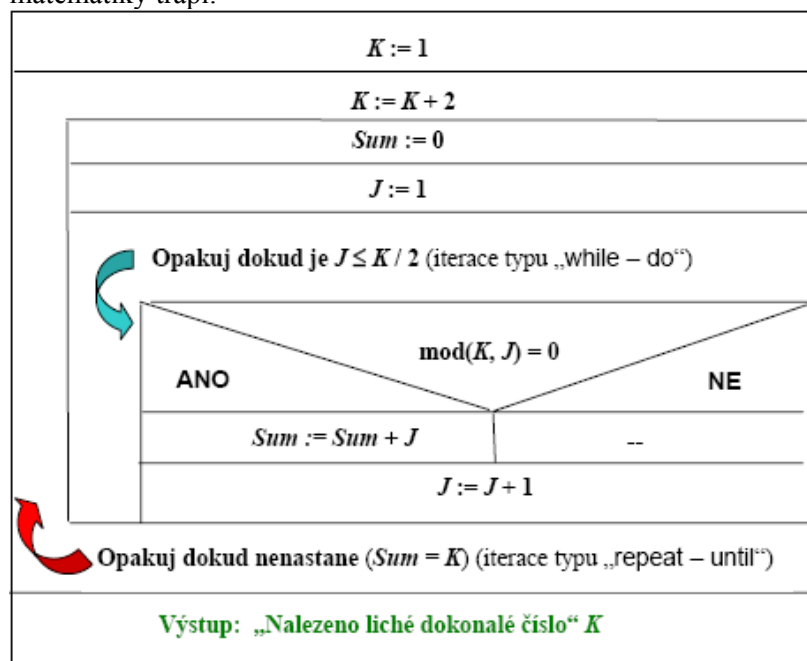
může být důkaz finitnosti procedury obtížný. Někdy nevíme, zda procedura algoritmem je či není (viz další příklad).

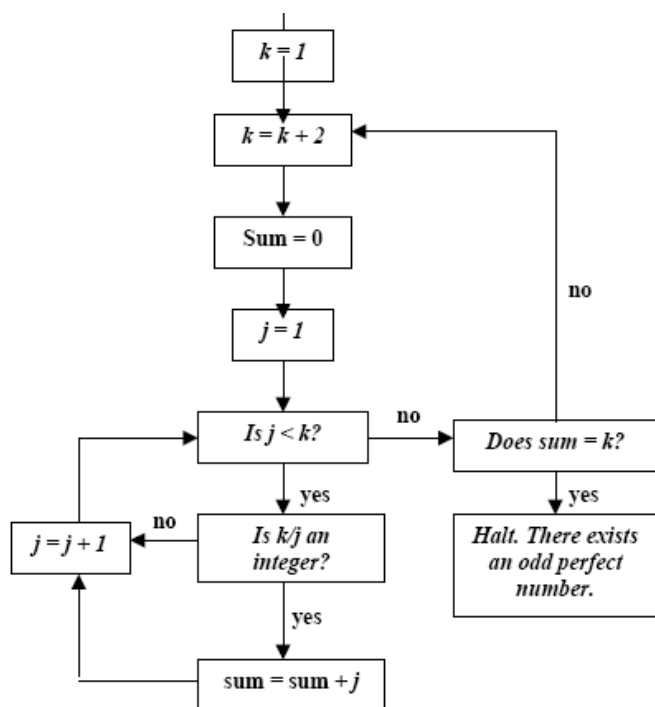
Později ukážeme, že někdy víme jen, že pro daný problém existuje procedura, ne však algoritmus.

Příklad 2:

Přirozené číslo se nazývá dokonalé, jestliže součet všech jeho dělitelů s výjimkou jeho samého je roven tomuto číslu. Nejmenší dokonalá čísla jsou 6 ($3+2+1=6$), 28 ($14+7+4+2+1=28$). Další je 496. Všechna sudá dokonalá čísla jsou známa. Jsou to čísla tvaru $2^n - 1 \cdot 2^{n-1}$, kde $2^n - 1$ je prvočíslo. Jiná sudá dokonalá čísla nejsou. Nikdo však neví, zda existuje nějaké liché dokonalé číslo. Pokud ano, musí být „hodně velké“.

Následující procedura se zřejmě zastaví, pokud takové liché dokonalé číslo existuje. Pokud není, poběží stále a nebude tedy algoritmem. Kdybychom uměli rozhodnout, zda algoritmem je, či není uměli bychom odpovědět na otázku, která matematiky trápí.





Problém příslušnosti slova do jazyka se nazývá **algoritmicky rozhodnutelný**, pokud existuje algoritmus, který vždy skončí a dá na tuto otázku odpověď ANO nebo NE.

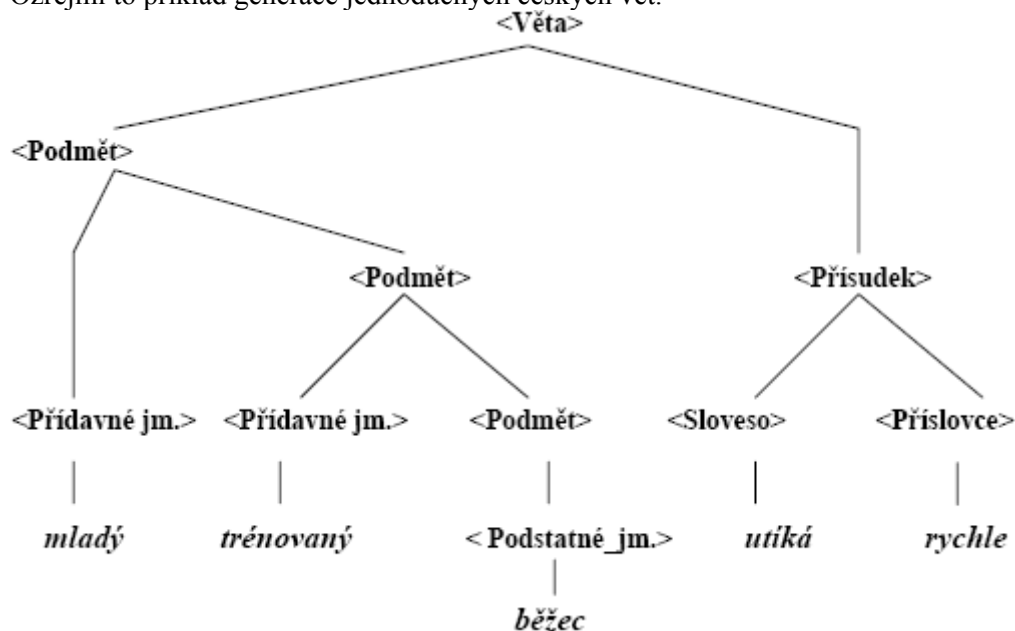
Problém příslušnosti slova do jazyka se nazývá **algoritmicky přechíslitelný**, respektive rekurzivně rozpoznatelný, pokud existuje procedura, která slova rozpoznává, tak, že pokud slovo do jazyka patří, skončí. Pokud neskončí (není algoritmem) slovo do jazyka nepatří. (Problém pouhé procedury, která není algoritmem je pochopitelně v tom, že pokud procedura běží, nevíme zda to znamená odpověď NE nebo zda máme „dále čekat“ na případnou kladnou odpověď).

Existuje mnoho problémů, které jsou pouze rozpoznatelné, ne rozhodnutelné a mnoho problémů, které nejsou ani rozpoznatelné. Ukážeme, že mezi ně patří i takové problémy, které lze konečným způsobem popsat. To je takové, kde příslušný jazyk lze popsat konečným počtem pravidel.

GRAMATIKY

Neformálně generativní gramatika je systém jak pomocí daných pravidel vytvářet z abecedy slova daného jazyka.

Ozřejmí to příklad generace jednoduchých českých vět.



Použitá pravidla v příkladě:

<Věta> → <Podmět> <Přísudek>

<Podmět> → <Přídavné jm.> <Podmět>

<Podmět> → <Podstatné jm.>

<Přísudek> → <Sloveso> <Příslovce>

<Přídavné jm.> → trénovaný

<Přídavné jm.> → mladý

<Podstatné jm.> → běžec

<Sloveso> → utíká

<Příslovce> → rychle

DEFINICE GENERATIVNÍ GRAMATIKY

Generativní gramatika je soubor pravidel jak vytvářet všechna a slova jazyka nad danou abecedou.

K tomu je třeba zadat:

1. **Množinu V_N neterminálních symbolů.** Abecedu proměnných.

2. **Množinu V_T terminálních symbolů.** Abecedu generovaného jazyka.

Množiny V_N a V_T musí být disjunktní ($V_N \cap V_T = \emptyset$). Označme dále $V_N \cup V_T = V$.

3. **Množinu tak zvaných produkčních neboli přepisovacích pravidel P .** Každé pravidlo přiřazuje nějakému řetězci $\alpha \in V^*$, který obsahuje aspoň jeden neterminální symbol (proměnnou) řetězec $\beta \in V^*$ terminálních a neterminálních symbolů.

4. **Počáteční symbol $S \in V_N$,** který je proměnnou (neterminálním symbolem).

Formálně matematicky je generativní gramatika uspořádaná čtveřice $G = (V_N, V_T, P, S)$, kde V_N, V_T, P, S splňují vlastnosti z předchozích bodů 1., 2., 3. a 4.

Je-li $\alpha \rightarrow \beta$ produkční pravidlo, říkáme, že **řetězec $\gamma\alpha\delta$ lze přímo přepsat na řetězec $\gamma\beta\delta$** , (nebo, že $\gamma\alpha\delta$ lze přímo odvodit z $\gamma\beta\delta$) a zapisujeme to $\gamma\beta\delta \rightarrow_G \gamma\alpha\delta$. Přímé přepsání znamená tedy náhradu nějakého podřetězce pravou stranou nějakého přepisovacího pravidla.

Nechť $\alpha_1, \alpha_2, \alpha_n$ jsou řetězce a nechť $\alpha_1 \rightarrow_G \alpha_2, \alpha_2 \rightarrow_G \alpha_3, \dots, \alpha_{n-1} \rightarrow_G \alpha_n$. V tom případě říkáme, že **lze přepsat na** (nebo, že α_n lze odvodit z α_1) a značíme to $\alpha_1 \rightarrow_{G^*} \alpha_n$. Přepsání tedy znamená postupné získání nového řetězce následným užitím konečného počtu přepisovacích pravidel.

Jazyk generovaný gramatikou G je množina všech řetězců, které lze odvodit z počátečního symbolu S (tedy všech slov na které lze počáteční symbol přepsat), které se skládají pouze z terminálních symbolů (abecedy generovaného jazyka, bez proměnných).

Příklady:

1. $V_N = \{S\}, V_T = \{0, 1\}, P = \{S \rightarrow 0S1, S \rightarrow 01\}$.

Generovaný jazyk je $L(G) = \{0^n 1^n, \text{ kde } n = 1, 2, \dots\}$.

2. $V_N = \{S, B, C\}, V_T = \{a, b, c\}, P = \{S \rightarrow aSBC, S \rightarrow aBC, CB \rightarrow BD, aB \rightarrow ab, bB \rightarrow b, bC \rightarrow bc, cC \rightarrow cc\}$.

Generovaný jazyk je: $L(G) = \{a^n b^n c^n, \text{ kde } n = 1, 2, \dots\}$.

CHOMSKÉHO HIERARCHIE GRAMATIK

Typ 0 = Všechny gramatiky – generují jazyky typu 0.

Typ 1 = Kontextové gramatiky [context sensitive] \equiv Pro každé produkční pravidlo $\alpha \rightarrow \beta$ platí, že délka β je větší nebo rovna délce α – generují jazyky typu 1 – kontextové jazyky.

Typ 2 = Bezkontextové gramatiky [context free] \equiv každé produkční pravidlo je tvaru $\alpha \rightarrow \beta$, kde α je buď jednoduchá proměnná nebo β neprázdný řetězec. – generují jazyky typu 2, bezkontextové jazyky.

Typ 3 = Regulární gramatiky \equiv Každé produkční pravidlo je buď tvaru $A \rightarrow aB$ nebo tvaru $A \rightarrow a$, kde A a B jsou proměnná a a je terminál. – Generují jazyky typu 3 – regulární jazyky.

Každá regulární gramatika je bezkontextová,

Každá bezkontextová gramatika je kontextová,

Každá kontextová gramatika je typu 0.

Existuje bezkontextová gramatika, která není regulární,

Existuje kontextová gramatika, která není bezkontextová,

Existuje gramatika typu 0, která není kontextová.

DERIVAČNÍ STROMY BEZKONTEXTOVÝCH GRAMATIK

Nechť $G = (V_N, V_T, P, S)$ je bezkontextová gramatika, $V = V_N \cup V_T$.

Orientovaný strom T se nazývá derivační strom gramatiky G jestliže:

1. Každý uzel stromu T je ohodnocen některým symbolem z V .

2. Uzel stromu T je ohodnocen počáteč. symbolem gramatiky S gramatiky G .

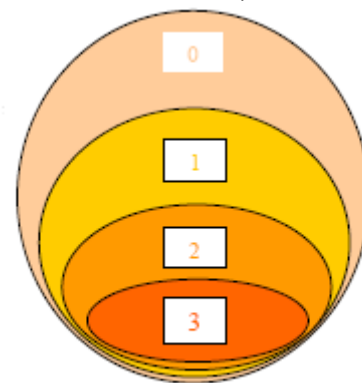
3. Jestliže uzel n má aspoň jednoho následníka různého od sama sebe ohodnoceného symbolem A , potom A musí být proměnná (neterminál) z V_N .

4. jestliže uzly $n_0, n_1, n_2, \dots, n_k$ jsou přímými následníky uzlu n v pořadí zleva doprava a mají po řadě ohodnocení A_1, A_2, \dots, A_k , potom $A \rightarrow A_1 A_2 \dots A_k$ musí být produkční pravidla z P .

5. Příklad:

$G = (\{S, A\}, \{a, b\}, P, S); P = \{S \rightarrow aAS, S \rightarrow SbA, S \rightarrow a, A \rightarrow ba\}$

Nakresleme derivační strom pro slovo aababbaa :



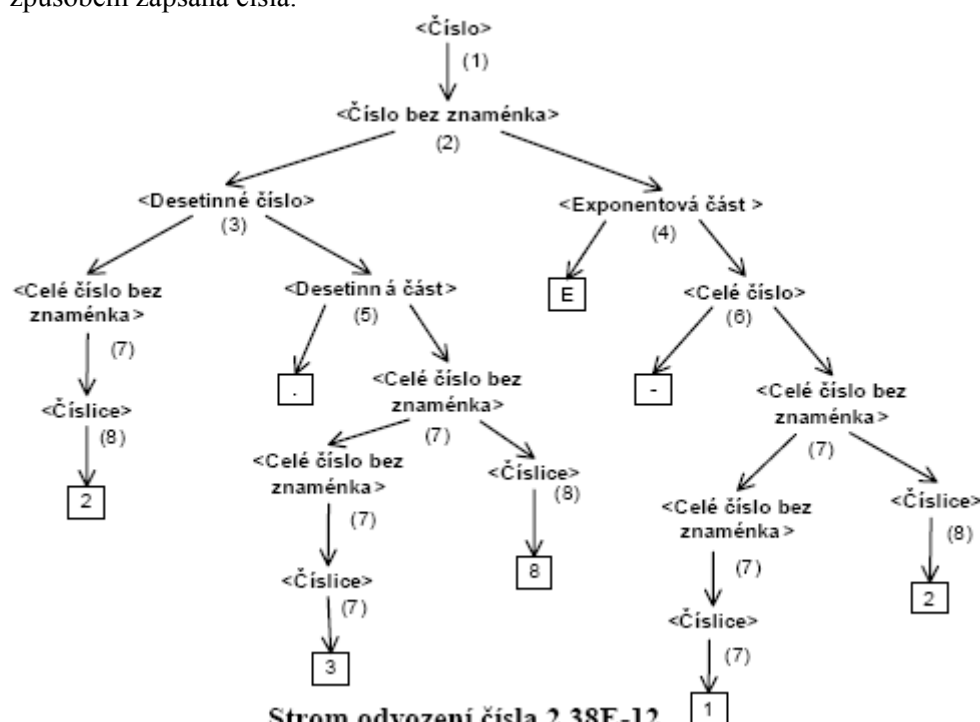
Množinová inkluze jazyků typu 0, 1, 2, 3.

možnost volby. Místo deseti pravidel $\langle \text{Číslice} \rangle ::= 0; \langle \text{Číslice} \rangle ::= 1; \dots, \langle \text{Číslice} \rangle ::= 9$ stačí tak zapsat jediné pravidlo: $\langle \text{Číslice} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$, které reprezentuje všech těchto deset pravidel.

Jazyk obsahující všechny přípustné zápisy reálného čísla může být generován gramatikou, která má počáteční symbol $\langle \text{Číslo} \rangle$, terminální abecedu $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, ., E\}$ a tato přepisovací pravidla:

- $\langle \text{Číslo} \rangle ::= \langle \text{Číslo bez znaménka} \rangle \mid +\langle \text{Číslo bez znaménka} \rangle \mid -\langle \text{Číslo bez znaménka} \rangle$;
- $\langle \text{Číslo bez znaménka} \rangle ::= \langle \text{Desetinné číslo} \rangle \mid \langle \text{Exponentová část} \rangle \mid \langle \text{Desetinné číslo} \rangle \langle \text{Exponentová část} \rangle$;
- $\langle \text{Desetinné číslo} \rangle ::= \langle \text{Celé číslo bez znaménka} \rangle \mid \langle \text{Desetinná část} \rangle \mid \langle \text{Celé číslo bez znaménka} \rangle \langle \text{Desetinná část} \rangle$;
- $\langle \text{Exponentová část} \rangle ::= E \langle \text{Celé číslo} \rangle$;
- $\langle \text{Desetinná část} \rangle ::= . \langle \text{Celé číslo bez znaménka} \rangle$;
- $\langle \text{Celé číslo} \rangle ::= \langle \text{Celé číslo bez znaménka} \rangle \mid +\langle \text{Celé číslo bez znaménka} \rangle \mid -\langle \text{Celé číslo bez znaménka} \rangle$;
- $\langle \text{Celé číslo bez znaménka} \rangle ::= \langle \text{Číslice} \rangle \mid \langle \text{Celé číslo bez znaménka} \rangle \langle \text{Číslice} \rangle$;
- $\langle \text{Číslice} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$;

Na obrázku je znázorněno, jak lze užitím těchto přepisovacích pravidel generovat například zápis čísla 2.38E-12 (tedy $2,38 \cdot 10^{-12}$). Grafu, který znázorňuje postupné používání přepisovacích pravidel, říkáme **strom odvození** nebo též **derivační strom** [derivation tree]. Čtenáři doporučujeme nakreslit si strom odvození pro některá další, přípustným způsobem zapsaná čísla.



Strom odvození čísla 2.38E-12

Navržená gramatika je zřejmě bezkontextová, není však regulární. Užívá i pravidla, která nejsou povolena jako přepisovací pravidla pro regulární gramatiky. Jde o pravidla (2), (3) a (7). To samozřejmě neznamená, že jí generovaný jazyk není regulární. V daném případě tento jazyk regulární je. Lze jej totiž generovat i jinou, tentokrát regulární gramatikou. Tato gramatika by byla o něco méně názorná. Doporučujeme čtenáři, aby ji navrhl jako cvičení.

JAZYKY, PROCEDURY A ALGORITMY

Jazyky typu 0 lze pouze přechísliť pomocí konečně definované procedury to znamená, pokud slovo do jazyka patří, dostaneme po konečném počtu kroků odpověď ANO. Pokud nepatří, procedura nemusí skončit svoji práci nikdy. Takové jazyky a gramatiky se také nazývají **rekurzivně přechísliťelné**.

Příslušnost slova k kontextovému jazyku lze vždy rozhodnout algoritmem. Pro ně existuje algoritmus (procedura, která vždy po konečném počtu kroků skončí), který dá odpověď ANO nebo NE na otázku zda slovo lze generovat danou kontextovou gramatikou, či nikoliv. Takové jazyky a gramatiky se nazývají **rekurzivní**.

Myšlenka důkazu, že jazyky typu 1 jsou rekurzivní je založena na tom, že pokud existuje nějaké odvození daného slova podle pravidel gramatiky, potom existuje i odvození „nepříliš dlouhé“. Tato úvaha se opírá o skutečnost, že slovo v derivačním stromu se v žádném kroku nemůže zkrátit.

Problém může být pouze s „vypouštěcím“ produkčním pravidlem $A \rightarrow \epsilon$.

Tento problém odstraní následující tvrzení:

Je-li L po řadě kontextový, bezkontextový nebo regulární jazyk, potom též jazyk $L \cup \{\epsilon\}$ a $L \div \{\epsilon\}$ je kontextový, bezkontextový nebo regulární.

8. MODELÝ VÝPOČTU

Konečný (deterministický) automat [finite automaton]

Konečný automat je abstraktní matematický model činností (výpočtu) prováděných počítačem = model algoritmu.

Pro specifikaci konečného automatu je třeba zadat:

1. Konečnou neprázdnou množinu stavů automatu K .
2. Konečnou neprázdnou vstupní abecedu automatu Σ .
3. Přechodovou funkci $\delta: K \times \Sigma \rightarrow K$, která každému (současnému) stavu automatu a každému (právě zpracovávanému) symbolu vstupní abecedy přiřazuje nový stav automatu.
4. Počáteční stav automatu $q_0 \in K$. V tomto stavu „práce“ automatu začíná.
5. Neprázdnou množinu koncových stavů $F \subseteq K$. Pokud automat ukončí svoji práci ve stavu, který patří do F , odpovídá kladně na otázku položenou na vstupu (slovo přijímá).

Formálně je konečný automat uspořádaná pětice $(K, \Sigma, \delta, q_0, F)$, kde písmenka označují objekty z předcházejícího výčtu.

Konečný automat ve své základní podobě je považován za nástroj pro řešení problémů.

Problém je modelován jako otázka, zda nějaké slovo (řetězec znaků vstupní abecedy automatu) patří do nějakého jazyka, či nepatří.

Při své práci konečný automat začíná pracovat v počátečním stavu, postupně zpracovává znaky slova (zleva doprava) a podle hodnoty přechodové funkce δ mění svůj stav. Po vyčerpání celého slova dá odpověď.

- Pokud po zpracování celého slova na vstupu je automat v některém z koncových stavů, slovo **přijímá** (problém zda slovo do jazyka patří rozhodne kladně).
- Pokud po zpracování celého slova je automat ve stavu, který do množiny všech koncových stavů nepatří, slovo **odmítá** (problém zda slovo do jazyka patří rozhodne záporně).

Někdy se přechodová funkce chápe pouze jako zobrazení z $K \times \Sigma$ do K . Tedy pro některé stavy a vstupní symboly nemusí být přechod definován. Dojde-li k této situaci, je posuzována tak, že slovo bylo odmítnuto.

Zapsáno formálně:

Funkce $\delta: K \times \Sigma \rightarrow K$ se rozšíří na $\delta^*: K \times \Sigma^* \rightarrow K$ takto:

$\delta^*(q, \epsilon) = q$ pro všechna $q \in K$

$\delta^*(q, xa) = \delta(\delta^*(q, x), a)$ pro všechna $q \in K, x \in \Sigma^*$ a $a \in \Sigma$.

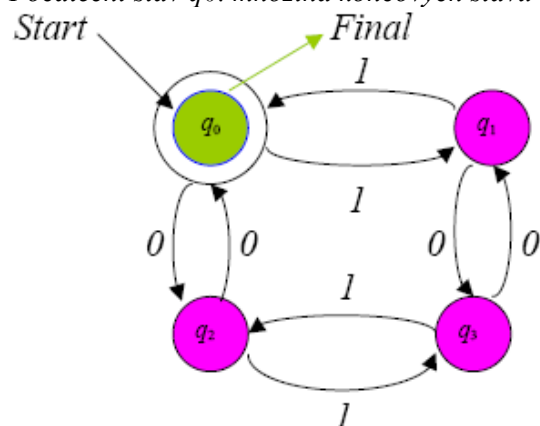
Rozpoznávaný jazyk je $L = \{w \in \Sigma^*: \delta^*(q_0, w) \in F\}$.

Jazyk se nazývá **rozpoznatelný konečným automatem**, pokud existuje konečný automat, který rozpoznává tento jazyk, to znamená pro každé slovo je schopen rozhodnout zda do jazyka patří, či nikoliv.

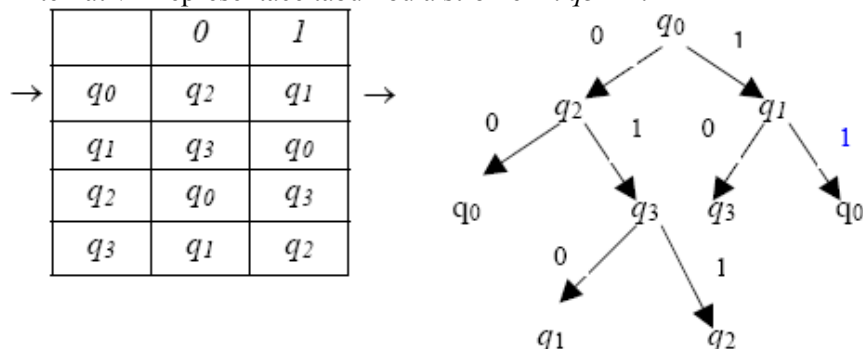
Příklad automatu se vstupní abecedou $\{0, 1\}$ a čtyřmi stavy: $K = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{0, 1\}$,

$\delta(q_0, 0) = q_2$, $\delta(q_1, 0) = q_3$, $\delta(q_2, 0) = q_0$, $\delta(q_3, 0) = q_1$, $\delta(q_0, 1) = q_1$, $\delta(q_1, 1) = q_0$, $\delta(q_2, 1) = q_3$, $\delta(q_3, 1) = q_2$.

Počáteční stav q_0 . množina koncových stavů $\{q_0\}$.



Alternativní reprezentace tabulkou a stromem : $q_3 \in \Sigma$.



Rozpoznává jazyk tvořený všemi slovy, která obsahují sudý počet nul a současně sudý počet jedniček.

MOŽNÁ REALIZACE:



0 ... horizontální posun \longleftrightarrow
1 ... vertikální posun \updownarrow

Modifikace pojmu konečného automatu

Mooreův a Meallyho sekvenční stroj

Někdy je účelnější modelovat počítač ne jako stroj na automatizované řešení problémů hledáním odpovědi ANO / NE, ale jako **stroj na automatizované zpracování dat** (přeměnu vstupů na výstupy).

Toho lze dosáhnout změnou modelu konečného automatu. Místo stanovení množiny F koncových stavů (těch, kdy je odpověď na otázku představující problém ANO) tím, že **automat generuje výstup ve výstupní abecedě**, kterým „dokumentuje“ svoji činnost.

Konečný Mooreův sekvenční stroj je uspořádaná šestice $(Q, p, \Sigma, \Delta, \delta, \mu)$, kde: Q je konečná množina stavů, $p \in Q$ je počáteční stav, Σ je konečná neprázdná vstupní abeceda, Δ je konečná neprázdná výstupní abeceda, $\delta: Q \times \Sigma \rightarrow Q$ je přechodová funkce (dvojici stávající stav, vstupní symbol přiřazuje nový stav), $\mu: Q \rightarrow \Delta$ je značkovácí funkce (každému stavu přiřazuje výstupní symbol).

Mooreův stroj na vstupní posloupnost symbolů vstupní abecedy postupně reaguje generováním symbolů výstupní abecedy. **Tyto symboly závisejí pouze na stavu, v kterém stroj je.**

Konečný automat lze simulovat pomocí Mooreova stroje. Stačí zvolit výstupní abecedu $\Delta = \{0,1\}$ a položit

$$\mu(q) = \begin{cases} 0 & \text{pokud } q \notin F \\ 1 & \text{pokud } q \in F. \end{cases}$$

Konečný Meallyho sekvenční stroj je uspořádaná pětice $(Q, p, \Sigma, \Delta, \delta, \mu)$, kde: Q je konečná množina stavů, $p \in Q$ je počáteční stav, Σ je konečná neprázdná vstupní abeceda, Δ je konečná neprázdná výstupní abeceda, $\delta: Q \times \Sigma \rightarrow Q$ je přechodová funkce (dvojici stávající stav, vstupní symbol přiřazuje nový stav), $\mu: Q \times \Sigma \rightarrow \Delta$ je výstupní funkce (každému stavu a každému vstupnímu symbolu přiřazuje výstupní symbol).

Rozdíl mezi Mooreovým a Meallyho strojem spočívá v tom, že u Mooreova stroje výstup závisí pouze na stavu v kterém stroj právě je. U Meallyho stroje se může výstup v daném stavu lišit v závislosti na právě zpracovávaném symbolu. Tento rozdíl není principiální. Každý Mooreův stroj je zřejmě Meallyho strojem. K Meallyho stroji sestrojíme mooreův stroj s ekvivalentní funkcí, pokud každý stav rozštěpíme do tolika stavů, kolik je prvků vstupní abecedy.

Příklad: Binární sčítanka.

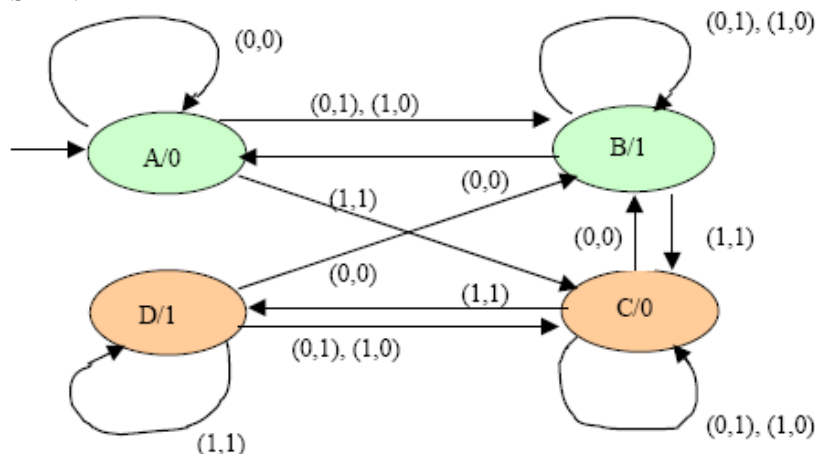
Vstupem jsou postupně uspořádané dvojice binárních číslic sčítaných čísel v pořadí od leva doprava, to je od nejvyššího bitu k nejnižšímu. Výstupem je binárně vyjádřený součet čísel, opět binárně v pořadí z leva doprava. Stav v kterém stroj skončil práci určuje, zda došlo k přetečení či ne.

Mooreův stroj

Tabulka

	STAVY	(0,0)	(0,1)	(1,0)	(1,1)
→	A / 0	A	B	B	C
	B / 1	A	B	B	C
	C / 0	B	C	C	D
	D / 1	B	C	C	D

STAVY

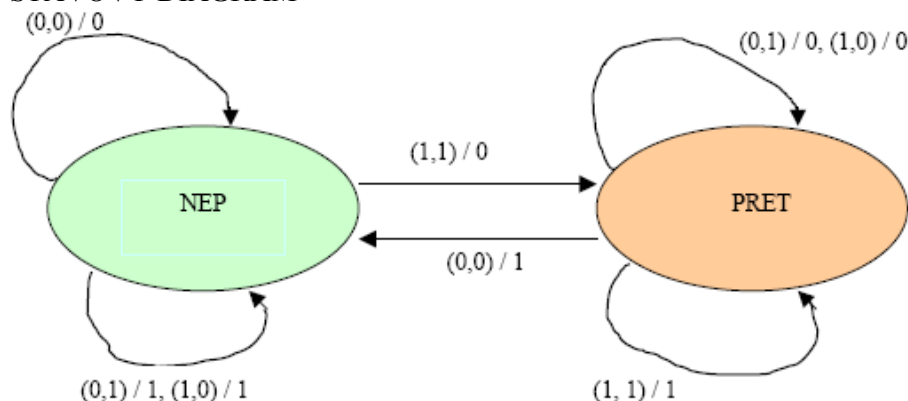


Meallyho stroj

TABULKA

	STAVY	(0,0)	(0,1)	(1,0)	(1,1)
→	NEP	NEP / 0	NEP / 1	NEP / 1	PRET / 0
	PRET	NEP / 1	PRET / 0	PRET / 0	PRET / 1

STAVOVÝ DIAGRAM



Konečný automat, Mooreův a Meallyho sekvenční stroj jsou **zjednodušené modely počítače**. Každý stávající počítač i počítačová síť mají pouze konečný, i když velmi velký počet možných stavů, daných možnými obsahy paměti a vnějších zařízení.

Možnosti konečných automatů, pokud jde o řešení problémů (rozpoznávání jazyků) jsou **značně omezené**. Ukážeme, že ani velmi jednoduché jazyky, například jazyk nad abecedou $\{0, 1\}$, tvořený slovy, která se skládají z určitého počtu nul následovaných stejným počtem jedniček nelze konečným automatem rozpoznat. Pokud totiž počet nul a jedniček není omezen předem, potom při libovolně velkém, avšak předem daném, konečném počtu stavů vždy najdeme slovo, kde počet nul toto omezení překročí a automat si s ním „neporadí“. To platí pro každý jazyk, který je definován na základě vlastnosti, kterou nelze postihnout pouze konečnou pamětí automatu.

V mnoha případech je však počet stavů stávajících výpočetních systémů tak velký, že **je výhodnější uvažovat modely, jejichž možnosti jsou potenciálně neomezené**, i když v každé aktuální situaci se využívá pouze konečný počet možných stavů těchto modelů.

Jazyky rozpoznatelné konečnými automaty

Mějme danou konečnou abecedu Σ a na množině Σ^* všech slov nad touto abecedou relaci R definovanou vztahem $x R y \Leftrightarrow \delta^*(q_0, x) = \delta^*(q_0, y)$,

kde q_0 je počáteční stav a δ přechodová funkce nějakého konečného automatu. Tedy dvě slova jsou ekvivalentní tehdy a jenom tehdy, když se automat po jejich zpracování dostane do téhož stavu.

Tato relace je zřejmě invariantní vzhledem k zřetězení zprava, tedy $x R y \Rightarrow xz R yz$.

Ekvivalence generuje rozklad množiny Σ^* na disjunktní třídy navzájem ekvivalentních slov. Tyto třídy odpovídají koncovým stavům automatu po zpracování jednotlivých slov. Platí to i naopak. Pokud se nám podaří slova abecedy Σ^* rozčlenit ekvivalencí, která generuje konečný počet tříd a je invariantní vzhledem k řetězení zprava, a vymezit daný jazyk jako sjednocení některých tříd této ekvivalence, získáme konečný automat, který tento jazyk rozpoznává.

To přesně říká tak zvaná **Nerodova věta**: Jazyk L nad abecedou Σ je rozpoznatelný konečným automatem tehdy a jenom tehdy, pokud existuje na množině slov nad Σ^* ekvivalence, generující konečný počet tříd ekvivalence a invariantní vzhledem k zřetězení zprava, taková, že L je sjednocením některých tříd této ekvivalence.

Poznámka: Ekvivalenci invariantní vzhledem k zřetězení zprava se říká často „pravá ekvivalence“, ekvivalenci, která generuje rozklad na konečný počet tříd „ekvivalence konečného indexu“. V Nerodově větě se pak krátce hovoří o „pravé ekvivalenci konečného indexu“.

Podobným postupem lze dokázat, **Chomského větu**, která plně charakterizuje jazyky rozpoznatelné konečnými automaty: Jazyk je rozpoznatelný konečným automatem tehdy a jenom tehdy, je-li regulární (typu 3 podle Chomského hierarchie gramatik a jazyků).

K důkazu této věty je vhodné užít toho, že ke každé regulární gramatice, tedy gramatice určené pouze přepisovacími pravidly typu $X \rightarrow wY$, $A X \rightarrow w$

Existuje s ní ekvivalentní gramatika (generující též jazyk), s přepisovacími pravidly pouze typu $X \rightarrow wY$, $X \rightarrow Y$ a $X \rightarrow \varepsilon$. Na základě Nerodovy věty lze přesně dokázat, že jazyk tvořený slovy $0^n 1^n$, tvořený slovy, která mají libovolný počet nul následovaných tímž počtem jedniček nelze rozpoznat konečným automatem a není tedy regulární. Pokud by existovala pravá ekvivalence generující rozklad na N tříd, potom aspoň dvě slova z posloupnosti $N+1$ slov $0, 00, 000, \dots, 0^N, 0^{N+1}$ musí patřit do stejné třídy ekvivalence. Třída 0^J a 0^K , kde $J \neq K$. Doplníme-li zprava obě tato slova J znaky 1, musí také patřit do téže třídy ekvivalence, protože tato ekvivalence je invariantní vůči řetězení zprava. To však možné není, protože jedno z nich však do jazyka patří a druhé nikoliv.

Podobně se dá dokázat, že jazyk tvořený slovy pouze ze symbolů 1, takovými, že jejich délka je druhou mocninou přirozeného čísla také není rozpoznatelný konečným automatem. Důkaz lze opírat o fakt, že mezi K^2 a $(K+1)^2$ je $2K+1$ celých čísel a tento počet tedy neomezeně roste.

Nedeterministické konečné automaty

Návrh konečného automatu na základě definice jazyka nemusí být snadnou záležitostí. Pro snazší návrh i pro usnadnění analýzy některých vlastností regulárních jazyků a pro dokazování vět o nich je účelné zavést pojem nedeterministických konečných automatů.

Jde o automaty, u kterých nemusí být jednoznačně určen počáteční stav a/nebo v některých situacích může být z daného stavu při daném vstupním symbolu povoleno více možných přechodů do nového stavu. Definice je analogická s definicí konečného automatu. Rozdíly jsou vyznačeny červeně.

Pro specifikaci nedeterministického konečného automatu **je třeba zadat:**

1. Konečnou neprázdnou **množinu stavů automatu** K .
2. Konečnou neprázdnou **vstupní abecedu automatu** Σ .
3. **Přechodovou funkci** $\delta: K \times \Sigma \rightarrow 2^K$, která každému (současnému) stavu automatu a každému (právě zpracovávanému) symbolu vstupní abecedy přiřazuje množinu možných nových stavů automatu (podmnožinu množiny K všech stavů). Tato množina může být i prázdná.
4. **Neprázdnou množinu možných počátečních stavů automatu** $I \subseteq K$. V některém z těchto stavů může automat svou práci začít..
5. **Neprázdnou množinu koncových stavů** $F \subseteq K$. Pokud automat ukončí svoji práci ve stavu, který patří do F , odpovídá kladně na otázku položenou na vstupu (slovo přijímá).

Nedeterministický konečný automat přijímá slovo, pokud toto slovo odpovídá nějaké cestě z některého z počátečních stavů do některého koncového stavu. Tedy pokud se nám podaří zvolit počáteční stav a při zpracování každého symbolu vybrat z možných přechodů do nového stavu takový, že po přečtení celého slova automat skončí svoji práci v množině F (v koncovém stavu). Tedy neprázdné slovo $w_1 \dots w_n \in \Sigma^+$ je přijato, pokud existuje posloupnost q_1, \dots, q_{n+1} taková, že $q_1 \in I, q_{n+1} \in F$ a pro všechna $i = 1, \dots, n$ je $q_{i+1} \in \delta(q_i, w_i)$. Prázdné slovo je přijato, pokud $I \cap F \neq \emptyset$.

Nalézt pomocí nedeterministického postupu kladnou odpověď tedy znamená „mít dobrého rádce“ jak postupovat. Tak jak práce deterministického automatu simuluje „výpočet“, tak práce nedeterministického automatu simuluje prověření známého výsledku „zkouškou“.

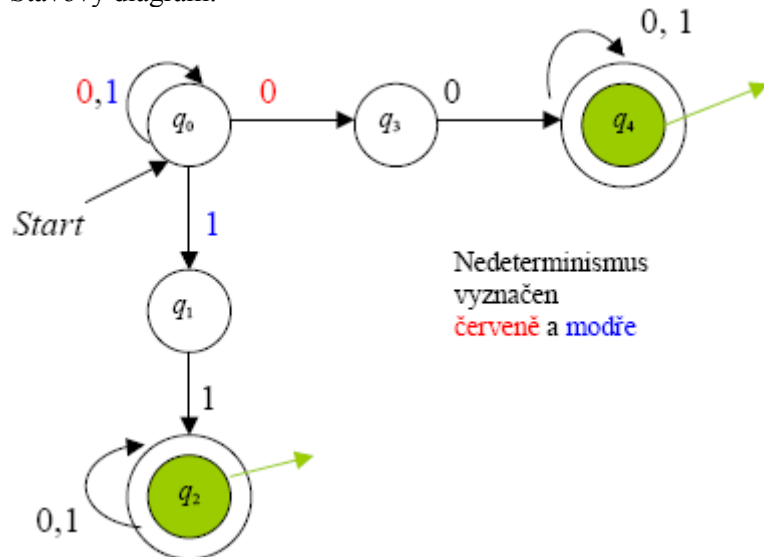
Příklad: Nedeterministický konečný automat, který přijímá všechna slova nad abecedou $\{0,1\}$, která obsahují buď dvě bezprostředně za sebou následující nuly nebo dvě bezprostředně za sebou následující jedničky.

Specifikace: $M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0,1\}, \delta, \{q_0\}, \{q_2, q_4\})$

$\delta(q_0, 0) = \{q_0, q_3\}, \delta(q_1, 0) = \emptyset, \delta(q_2, 0) = \{q_2\}, \delta(q_3, 0) = \{q_4\}, \delta(q_4, 0) = \{q_4\},$

$\delta(q_0, 1) = \{q_0, q_1\}, \delta(q_1, 1) = \{q_2\}, \delta(q_2, 1) = \{q_2\}, \delta(q_3, 1) = \emptyset, \delta(q_4, 1) = \{q_4\}.$

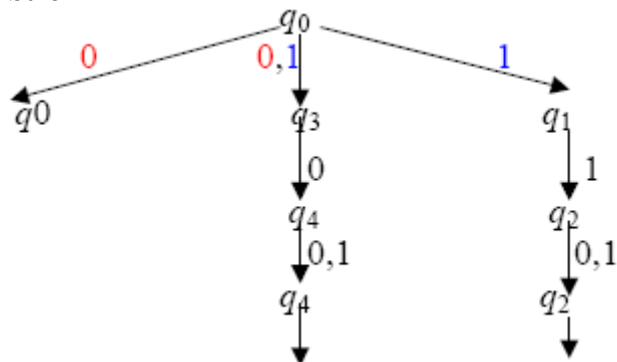
Stavový diagram:



Tabulka:

		0	1	
→	q_0	q_0, q_3	q_0, q_1	
	q_1	\emptyset	q_2	
	q_2	q_2	q_2	→
	q_3	q_4	\emptyset	
	q_4	q_4	q_4	→

Strom



Jazyky rozpoznatelné nedeterministickými konečnými automaty

Konečný automat je zvláštním případem nedeterministického konečného automatu. Abychom uvážili jaké jazyky jsou rozpoznatelné pomocí nedeterministických konečných automatů, provedme tuto úvahu:

Reakci deterministického automatu na zadané slovo z Σ^* lze sledovat pomocí posloupnosti stavů, kterými deterministický automat prochází při postupném zpracovávání symbolů slova. Reakci nedeterministického automatu na zadané slovo z Σ^* budeme sledovat tak, že místo jednoznačně určeného stavu budeme uvažovat **množinu všech stavů, do kterých se může nedeterministický automat dostat** při postupném zpracování symbolů zadaného slova. To je podmnožina množiny K všech stavů tohoto automatu, tedy prvek množiny $2K$. Konečný deterministický automat, jehož stavy tvoří množinu $2K$ s množinou koncových stavů tvořenou podmnožinami $\Phi \in 2K$, takovými, že $\Phi \cap F \neq \emptyset$ zřejmě přijímá dané slovo tehdy a jen tehdy, když toto slovo přijímá daný nedeterministický konečný automat.

Platí tedy:

Ke každému nedeterministickému konečnému automatu existuje deterministický konečný automat, rozpoznávající též jazyk.

Nedeterministické konečné automaty rozpoznávají právě regulární jazyky (jazyky typu 3 podle Chomského hierarchie).

Je však třeba si uvědomit, že transformací nedeterministického automatu na deterministický naznačenou **podmnožinovou konstrukcí** vznikne z automatu o K stavech automat o $2K$ stavech. Vzhledem k prudkému růstu exponenciální funkce to může být vážnou praktickou překážkou.

Praktické postupy při konstrukci konečného automatu pro daný jazyk spočívají v těchto krocích.

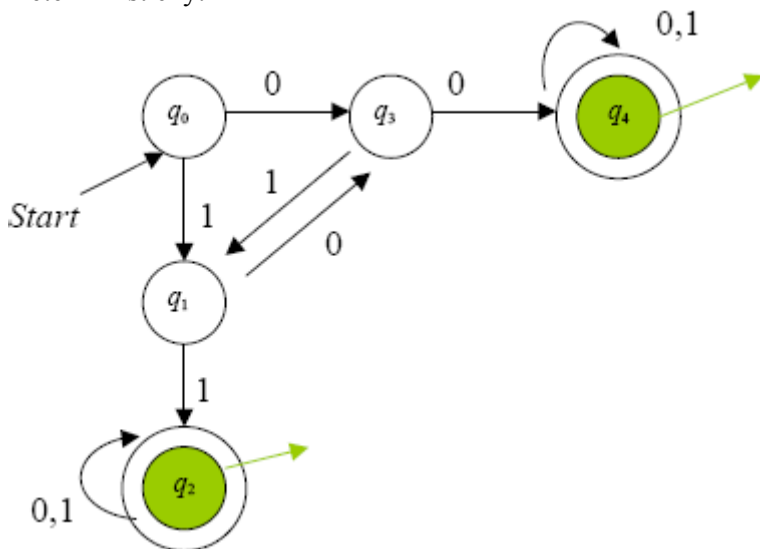
1. Sestrojení nedeterministického konečného automatu rozpoznávajícího daný jazyk
2. Transformace nedeterministického automatu na deterministický podmnožinovou konstrukcí.
3. Postupná redukce stavů deterministického automatu.

Pro třetí bod platí, že problém ekvivalence (generování téhož jazyk') lze pro konečné automaty řešit algoritmicky (existuje konečný automat, který rozhoduje problém, zda jsou dva konečné automaty ekvivalentní či nikoliv).

Všechny konečné automaty ekvivalentní s daným a s minimálním počtem stavů jsou navzájem isomorfní a existuje algoritmitizovatelný postup jak je získat.

Pro některé regulární jazyky je návrh deterministického automatu, který je rozpoznává intuitivní cestou poměrně snadný úkol. Tak například pro jazyk nad abecedou $\{0,1\}$, která obsahují buď dvě bezprostředně za sebou následující nuly nebo dvě bezprostředně za sebou následující jedničky, pro který jsme navrhli nedeterministický konečný automat je ekvivalentním deterministickým automatem konečný automat s stavovým diagramem:

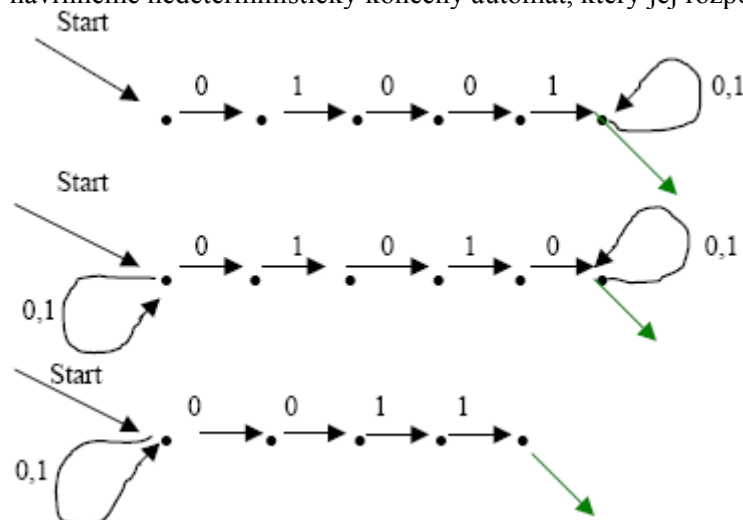
Deterministický:



Pro jazyk nad abecedou $\{0,1\}$, složený ze slov, která:

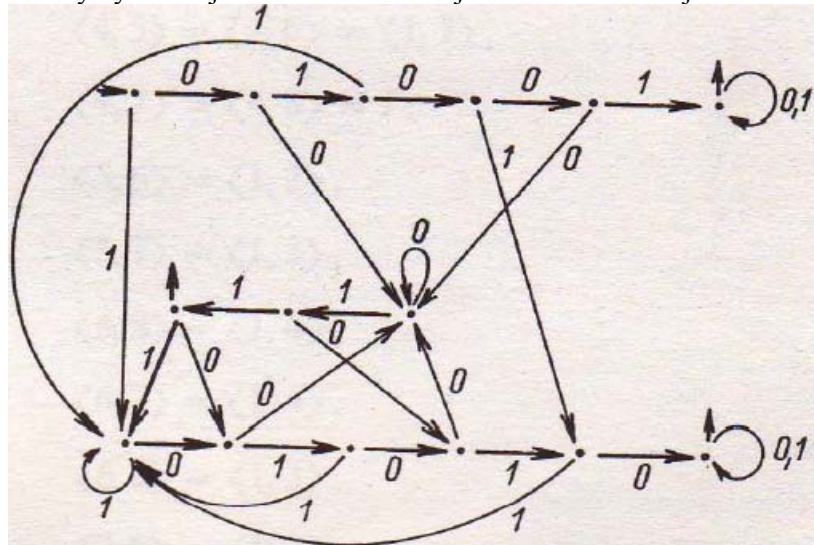
- buď začínají úsekem 01001
- nebo obsahují (někde) úsek 01010 jako poslovo
- nebo končí úsekem 0011

navrhne nedeterministický konečný automat, který jej rozpoznává snadno. Je to automat:



Přímý návrh deterministického automatu rozpoznávajícího tento jazyk by si vyžádal poměrně značnou dávku intuice a zvýšenou pozornost.

Možný výsledek je uveden na následujícím obrázku. Ani jeho kontrola není zcela triviální.



I postup převodem nedeterministického automatu na deterministický množinovou konstrukcí a následné redukce naráží na problémy. 217 je opravdu velké číslo.

Uzávěrové vlastnosti regulárních jazyků

Skutečnost, že množina všech regulárních jazyků nad danou abecedou je totožná s množinou všech jazyků nad touto abecedou, které lze rozpoznat konečnými automaty nám dává možnost poměrně snadno odvodit řadu důležitých vlastností množiny regulárních jazyků.

Označme v dalším Σ pevně zvolenou abecedou a $L_j, j = 1, 2, \dots$ jazyky nad touto abecedou (podmnožiny Σ^*). Potom platí:

• **Je-li L regulární jazyk, je i jeho doplněk $L' = \Sigma^* \div L$ regulární jazyk.** *Náznak proč tomu tak je:* V konečném automatu, který rozpoznává L zaměníme všechny koncové stavy za nekoncové a nekoncové za koncové. Automat s koncovými stavy $K \div F$ bude rozpoznávat $\Sigma^* \div L$.

• **Jsou-li L_1 a L_2 regulární jazyky je i jejich sjednocení $L_1 \cup L_2$ regulární jazyk.** *Náznak proč tomu tak je:* Necht' konečné automaty rozpoznávající po řadě L_1 a L_2 mají po řadě množiny stavů K_1 a K_2 a množiny koncových stavů F_1 a F_2 . Sestrojíme konečný automat s množinou stavů $K_1 \times K_2$ a s množinou koncových stavů $(K_1 \times F_2) \cup (F_1 \times K_2)$. (stačí tedy, když bude koncového stavu dosaženo pouze pro jednu složku). Takový automat rozpoznává $L_1 \cup L_2$.

• **Jsou-li L_1 a L_2 regulární jazyky je i jejich průnik $L_1 \cap L_2$ regulární jazyk.** *Náznak proč tomu tak je:* Důvod je obdobný jako u předchozího bodu. Je však třeba aby koncového stavu bylo dosaženo u obou složek. Množina koncových stavů automatu rozpoznávajícího $L_1 \cap L_2$ bude $F_1 \times F_2$.

Zřetěžením (někdy též součinem) $L_1 \cdot L_2$ jazyků L_1 a L_2 se nazývá jazyk, jehož všechna slova vzniknou zřetěžením slova z jazyka L_1 se slovem z jazyka L_2 .

Mocnina jazyka, jejímž exponentem je přirozené číslo N je definována indukci takto: $L^0 = \{\varepsilon\}$, $L^1 = L$, $L^{N+1} = L^N \cdot L$ pro všechna přirozená čísla $N \geq 1$.

Jazyk L^N tedy obsahuje právě ta slova, která vzniknou zřetěžením N slov jazyka L .

Označme dále $L^* = \bigcup_{j=0}^{\infty} L^j$. Jazyk L^* se říká také řetězový **uzávěr** neboli **iteraci** jazyka L . Je to jazyk tvořen všemi slovy,

kteřé vzniknou zřetěžením libovolného počtu slov jazyka L . Platí:

- Jsou-li L a L_j , $j = 1, 2, \dots$ regulární jazyky, potom je i zřetězení dvou regulárních jazyků, $L_1 \cdot L_2$ regulární jazyk, libovolná přirozená mocnina L^N pro $N \geq 0$ regulárního jazyka regulární jazyk a iterace L , to je jazyk L^* vzniklý zřetěžením libovolného počtu slov jazyka L regulární jazyk.

Názna proč tomu tak je uvedeme pouze pro zřetězení dvou jazyků. Pro každý z nich sestojíme konečný automat, který je rozpoznává. Poté doplníme přechody ze všech koncových stavů prvního automatu na počáteční stavy druhého. Získáme automat, obecně nedeterministický, který rozpoznává zřetězení.

Regulární jazyky (= jazyky rozpoznatelné konečnými automaty) nad danou abecedou tvoří tedy Booleovu algebru, uzavřenou vůči operaci zřetězení.

Mezi regulární jazyky patří všechny jazyky, které obsahují konečný počet slov.

Z jazyků s nekonečným počtem slov, jak jsme viděli, jsou regulární pouze některé.

Lze dokázat i že množina všech regulárních jazyků je „nejmenší“ množinou, která obsahuje všechny jazyky s konečným počtem slov a je uzavřená vůči operacím je uzavřená vůči operacím doplnění, sjednocení (a tedy i průniku), řetězení a řetězového uzavěru (iterace). Slovu „nejmenší“ je třeba rozumět tak, že každá jiná množina jazyků, která tyto vlastnosti má obsahuje množinu všech regulárních jazyků jako svoji podmnožinu.

Množina regulárních jazyků je uzavřená i vůči operaci T obrácení pořadí symbolů ve slově. Je-li L regulární jazyk je i L^T regulární jazyk.

Úlohy:

- jsou dva konečné automaty ekvivalentní automat (to je zda rozpoznávají též jazyk),
 - zda je jazyk rozpoznávaný daným automatem prázdný či neprázdný,
 - zda jazyk rozpoznávaný daným konečným automatem je konečnou nebo nekonečnou množinou,
- jsou všechny algoritmicke řešitelné. To znamená existuje konečný automat, který na ně dává odpověď.

Regulární výrazy

Regulární výrazy představují užitečný formalismus pro zápis regulárních jazyků nad danou abecedou $\Sigma = \{a_1, a_2, \dots, a_N\}$.

Abeceda regulárních výrazů vznikne doplněním abecedy Σ o speciální symboly s tímto významem:

- + ve významu sjednocení, respektive logické disjunkce (náhrada za \cup , respektive za \vee),
- . pro označení zřetězení symbolů a slov (někdy se vynechává), - . má vyšší prioritu než +,
- * pro označení řetězového uzavěru (opakování v libovolném počtu $N \geq 0$, tedy včetně možnosti žádného opakování) – tento znak má nejvyšší prioritu.

Dále abeceda regulárních výrazů užívá symboly „ \emptyset “ pro prázdnou množinu a „ ε “ pro prázdné slovo a levé a pravé kulaté závorky „(“ a „)“.

Pravidla pro gramatiku regulárních výrazů jsou tato:

- $\emptyset \in RV(\Sigma)$,
- $\varepsilon \in RV(\Sigma)$,
- $a \in RV(\Sigma)$ pro každé $a \in \Sigma$,
- $\alpha, \beta \in RV(\Sigma) \Rightarrow (\alpha + \beta) \in RV(\Sigma)$, $(\alpha \cdot \beta) \in RV(\Sigma)$, $\alpha^* \in RV(\Sigma)$.

Užívají se i běžné konvence o vynechávání závorek.

Příklady ($\Sigma = \{a, b\}$):

1. $b \cdot a^*$ - popisuje množinu slov (jazyk) tvořený slovy začínajícími symbolem b , následovaným libovolným počtem symbolů a . Opakování může být i prázdné. Tedy za b žádné a být nemusí. Slovo b do jazyka patří. Tento jazyk lze zapsat zkráceně i jako ba^* . Dále budeme užívat již jen zkrácený zápis.

2. baa^* - popisuje jazyk obsahující slova začínající b , za kterým následuje jedno nebo více a . Slovo s jediným symbolem b do něj nepatří. To je rozdíl oproti jazyku popsanému v příkladě 1.

3. $a^*ba^*ba^*$ - označuje jazyk obsahující slova v který se vyskytuje b právě dvakrát.

4. $(a+b)^*(aa+bb)(a+b)^*$ - jazyk tvořený slovy, která obsahují buď aspoň dvě bezprostředně po sobě následující a nebo aspoň dvě po sobě bezprostředně následující b .

5. $(b+abb)^*$ - všechna slova u kterých po každém a bezprostředně následují aspoň dvě b .

6. $((a+b)(a+b)(a+b))^*$ - všechna slova, jejichž délka je dělitelná třemi.

7. $(aa + bb + (ab + ba)(aa + bb)^*(ab + ba))^*$ - všechna slova, která obsahují sudý počet výskytů a a současně sudý počet výskytů b .

Kleeneova věta: Třída jazyků, které lze generovat pomocí regulárních výrazů je totožná s třídou regulárních jazyků. (Tedy jazyků rozpoznatelných konečnými automaty a jazyků, které lze generovat gramatikami typu 3 podle Chomského hierarchie).

Někdy je pohodlnější obohatit symboliku regulárních výrazů o další symboly pro průnik jazyků a doplněk.

Rozšířené regulární výrazy

Užívají kromě abecedy regulárních výrazů symboly

α & β – pro průnik jazyků \cap (logickou konjunkci \wedge - tečka ve smyslu „součin“ se zde nepoužívá pro možnou záměnu s operací zřetězení),

α' – čárku nebo $\bar{\alpha}$ - pruh nad výrazem pro komplement $\Sigma^* \div L$ (logickou negaci).

To pomůže zjednodušit zápis některých jazyků.

Například jazyk nad abecedou $\{0, 1\}$ tvořený všemi slovy, které obsahují aspoň jednu jedničku lze zapsat regulárním výrazem $(0 + 1)^* 1 (0 + 1)^*$. Pomocí rozšířených regulárních výrazů snáze jako $(0^*)^1$. Jazyk z příkladu 7. lze v rozšířené notaci zapsat jednodušeji jako: $(0^*10^*10^*)^* \& (1^*01^*01^*)^*$.

Pro úpravu rozšířených regulárních výrazů lze užít známá de Morganova pravidla:

$$(a + b)' = a' \& b', \quad (a \& b)' = a' + b',$$

$$a + b = (a' \& b')', \quad a \& b = (a' + b')'.$$

Některá vybraná rozšíření regulárních jazyků a konečných automatů

Pojem deterministických i nedeterministických konečných automatů lze rozšířit na tak zvané **dvousměrné konečné automaty**. U nich je reakcí na přečtení symbolu ze vstupního slova nejen možná změna stavu automatu, ale i rozhodnutí, zda automat provede jednu z následujících tří akcí:

- Posune „čtecí hlavu“ o jedno místo doprava (vpřed).
- Posune „čtecí hlavu“ o jedno místo vlevo (zpět).
- Ponechá hlavu na místě a bude v dalším kroku zpracovávat týž znak v jiném stavu.

Slovo je přijato, tehdy a jenom tehdy, když automat opustí vstupní slovo na jeho pravém konci v některém z koncových stavů. Pokud jej opustí v jiném stavu než koncovém nebo překročí levý okraj slova nebo se zacyklí, slovo odmítá. Lze ukázat, že dvousměrné konečné automaty rozpoznávají stejnou množinu jazyků jako (jednosměrné) konečné automaty. Regulární jazyky.

Regulární jazyky jsou definovány omezením prepisovacích pravidel generativní gramatiky pouze na pravidla dvou typů:

$$A \rightarrow aB \text{ a } A \rightarrow a.$$

Někdy se takovým gramatikám říká také **levé lineární gramatiky**. Již jsme uvedli, že ta na stejné omezení vede zúžení prepisovacích pravidel pouze na typy: $A \rightarrow aB$, $A \rightarrow B$ a $A \rightarrow \varepsilon$.

Pravé lineární gramatiky jsou definovány omezením prepisovacích pravidel pouze na typy: $A \rightarrow Ba$ a $A \rightarrow a$, případně $A \rightarrow Ba$, $A \rightarrow B$ a $A \rightarrow \varepsilon$.

Lze ukázat, že **pravé lineární gramatiky generují také množinu všech regulárních jazyků**.

Pokud však připustíme užití obou pravidel, pravého i levého, tedy prepisovací pravidla omezíme na typy: $A \rightarrow aB$, $A \rightarrow Ba$ a $A \rightarrow a$, případně $A \rightarrow aBb$ a $A \rightarrow a$, dostaneme již jazyk, který není regulární. Takovým jazykem je již analyzovaný jazyk $\{0^n1^n\}$, složený ze všech slov obsahujících na začátku nuly, následované stejným počtem jedniček.

Gramatiky s tímto omezením na typy prepisovacích pravidel se nazývají **lineární gramatiky**. Jimi generované jazyky **lineární jazyky**. Jde o důležitý zvláštní případ bezkontextových gramatik a jazyků.

Zásobníkové automaty [Pushdown automata]

Tato myšlenková konstrukce (model výpočtu) obohacuje možnosti konečných automatů o potenciálně nekonečnou paměť, z které však automat v každém okamžiku využívá pouze konečnou část. Velikost této paměti „pro poznámky“ však není omezena předem. Této paměti se říká zásobník. Jde o paměť s obslužnou disciplinou LIFO [last in - first out]. Tato paměť není přímo přístupná celá. V každém okamžiku je k dispozici pouze vrchol zásobníku, tedy naposledy uložený symbol. Na základě vrcholu zásobníku může automat měnit svůj stav, tento symbol může přepsat jiným symbolem, vrchol zásobníku odstranit nebo do zásobníku symbol uložit. Symboly ukládané do zásobníku tvoří tak zvanou zásobníkovou abecedu, která může být, ale nemusí být, stejná se vstupní abecedou zásobníku.

Formálně je zásobníkový automat uspořádaná sedmice: $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, kde

Q = je neprázdná, konečná množina možných stavů automatu,

Σ = je neprázdná, konečná vstupní abeceda automatu,

Γ = je neprázdná konečná zásobníková abeceda automatu,

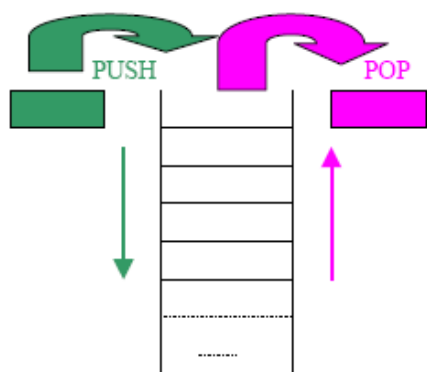
$q_0 \in Q$ = je počáteční stav automatu,

$Z_0 \in \Gamma$ = je počáteční symbol zásobníku (na počátku práce zásobník obsahuje jako jedinou položku tento symbol),

$F \subseteq Q$ = je množina všech koncových stavů automatu,

δ = Je zobrazení množiny $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$ do množiny všech konečných podmnožin množiny $Q \times \Gamma^*$.

Zobrazení δ tedy provádí akci na základě stavu automatu, vstupního symbolu a vrcholu zásobníku, při čemž připouští, že v daném kroku může i vstupní symbol nezpracovat. Výsledek kroku je nedeterministický výběr nového stavu automatu, doprovázený uložením konečně dlouhého slova (žádného, jednoho nebo více symbolů) zásobníkové abecedy na vrchol zásobníku.



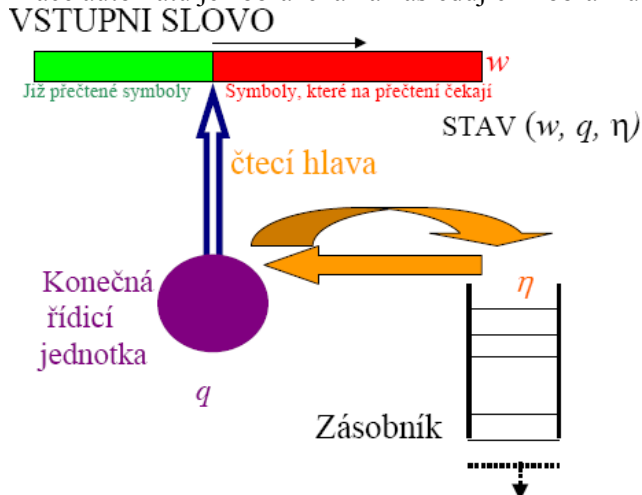
Zásobníkový automat zahájí svoji práci v počátečním stavu a provádí jednotlivé kroky až do vyčerpání celého vstupního slova, zleva doprava. V každém kroku může přečíst symbol ze vstupu a posunout čtecí hlavu o jedno místo vpravo, ale nemusí. V druhém případě považuje za vstup prázdné slovo a čtecí hlavu ponechá na místě. Na základě takto získaného vstupu, aktuálního vrcholu zásobníku a stavu v kterém právě je zvolí jednu z možností jak změnit svůj stav a uložit na vrchol zásobníku žádný, jeden nebo více, ale konečně mnoho symbolů zásobníkové abecedy.

Výběr nového stavu může být i prázdný. V tom případě, právě tak, jako když je prázdný zásobník automat svoji práci ukončí neúspěchem.

Obecně je tedy chování zásobníkového automatu nedeterministické.

Je uvažována i deterministická varianta, spočívající v tom, že není výběr zda vstupní znak přečíst, či provádět činnost „na místě“ a následný stav a slovo ukládané na vrchol zásobníku je určeno jednoznačně. Podobně jako u konečných automatů nedeterminismus nezvyšuje rozpoznávací schopnosti zásobníkových automatů. Uspadňuje však návrh a snižuje počet stavů, které je nutné uvažovat.

Práce automatu je zobrazena na následujícím obrázku:



Nehledíme-li na přirozená, ale dosud „ještě vzdálená“ omezení daná pravděpodobně konečným počtu atomů, jejich nenulovým rozměrem, omezené rychlosti přenosu signálu, lze zásobníkový automat považovat a model současných i všech budoucích možností výpočtu.

Řešení problému přijímání slov zásobníkovými automaty

Existují dva způsoby jak rozhodnout zda zásobníkový automat přijímá či odmítá slovo:

1. Přijímání podle koncového stavu. Slovo je přijato, pokud existuje aspoň jeden možný režim práce zásobníkového automatu, při kterém je přečteno celé slovo a po jeho přečtení se zásobníkový automat dostane do nějakého koncového stavu $q \in F$.

2. Přijímání podle prázdného zásobníku. Slovo je přijato, pokud existuje aspoň jeden možný režim práce zásobníkového automatu, při kterém je přečteno celé slovo a po jeho přečtení je zásobník prázdný. Množina F všech koncových stavů je v tomto případě nepodstatná.

Je-li P zásobníkový automat, označme:

$F(P)$ množinu jazyků, které lze rozpoznávat pomocí P koncovým stavem.

$E(P)$ množinu jazyků, které lze rozpoznávat pomocí P prázdným zásobníkem.

Platí následující věta: Necht' L je jazyk. Potom $L \in F(P)$ pro nějaký (nedeterministický) zásobníkový automat P tehdy a jenom tehdy, když $L \in E(P)$.

Oba typy rozpoznávání jsou tedy pro nedeterministické zásobníkové automaty ekvivalentní.

Pro deterministické zásobníkové automaty přímá analogie neplatí. Konečný jazyk $\{a, aa\}$ je regulární, tedy i rozpoznatelný zásobníkovým automatem, koncovým stavem. Nemůže být ale rozpoznatelný prázdným zásobníkem. Při čtení slova aa totiž po přečtení prvního znaku chybí prostředek jak zjistit, zda již bylo přečteno celé slovo či nikoliv. Platí

však, že pro každý deterministický zásobníkový automat P_1 existuje deterministický zásobníkový automat P_2 takový, že $F(P_1) = E(P_2)$. Množiny regulárních jazyků a jazyků rozpoznatelných deterministickými zásobníkovými automaty prázdným zásobníkem jsou však množinovou inkluzí neporovnatelné.

Jazyky rozpoznatelné deterministickým zásobníkovým automatem pomocí koncového stavu se nazývají **deterministické jazyky**. Deterministický jazyk se nazývá bezprefixový, pokud jej lze rozpoznat deterministickým zásobníkovým automatem pomocí prázdného zásobníku. Dodatečným přidáním zarážky na konec slova se rozdíl mezi množinami deterministických jazyků a bezprefixových deterministických jazyků setře. Platí: Je-li $L \subseteq \Sigma^*$ deterministický jazyk a $\$ \notin \Sigma$, potom $L.\{\$ \}$ je bezprefixový deterministický jazyk.

Pro jazyky rozpoznatelné zásobníkovými automaty platí následující **věta**:

Pro libovolný jazyk L jsou ekvivalentní následující tvrzení:

- L je bezkontextový jazyk (jazyk typu 2 z Chomského hierarchie jazyků).
- L je rozpoznatelný nějakým zásobníkovým automatem pomocí prázdného zásobníku (vystačí se i s automatem, který má jediný stav).
- L je rozpoznatelný nějakým zásobníkovým automatem pomocí koncového stavu.

Na základě podobných úvah jako pro regulární jazyky lze ukázat, že množina bezkontextových jazyků je uzavřená vůči operacím sjednocení, průniku, doplňku, zřetězení i řetězového uzávěru (iterace).

Příklady:

1. Jazyk L se skládá ze všech slov nad abecedou $\{0, 1\}$ se stejným počtem nul a jedniček.

Sestrojíme zásobníkový automat, který přijímá slova tohoto jazyka prázdným zásobníkem. Automat bude mít jediný stav Q , vstupní abecedu $\{0, 1\}$, výstupní abecedu $\{Z, M, C\}$. Pro větší názornost si prvky zásobníkové abecedy můžeme představit jako zelené, modré a červené talíře, které ukládáme do zásobníku, sloupce s pohyblivým dnem dole. Na počátku práce bude v zásobníku jediný zelený talíř. Ten máme možnost kdykoliv odstranit a ukončit tak činnost automatu. Modrý talíř na vrhu bude znamenat, že v dosud přečteném vstupu převládají nuly. O kolik, to udává počet modrých talířů. Počet červených talířů naopak udává o kolik bylo v dosud přečtené části slova více jedniček. Automat pracuje tak, že v případě vstupu nuly se posívá na vrchol zásobníku. Je-li tam modrý talíř nebo zelený talíř, přidá další modrý. Pokud tam je červený tak jeden červený odebere. Zrcadlově postupuje při vstupu jedničky. Pokud je vrchol červený nebo zelený, jeden červený talíř přidá. Je-li modrý, jeden červený talíř odebere.

Formálně zapsáno automat sedmice ($\{Q\}, \{0, 1\}, \{Z, M, C\}, Q, \delta, Z, \emptyset$), kde funkce δ je dána tabulkou:

Řádky: Symbol na vrcholu zásobníku.

Sloupce: Čtené vstupní symboly.

Obsah tabulky: Řetězec symbolů ukládaných do zásobníku.

		0	1	ϵ
\rightarrow	Z	MZ	CZ	Z ϵ
	M	MM	ϵ	--
	C	ϵ	CC	--

Jediný nedeterminismus potřebujeme proto, že v okamžiku, kdy vznikne rovnováha počtu nul a jedniček nevíme, zda jsme na konci slova a máme výpočet ukončit nebo zda slovo ještě pokračuje.

Tento problém se standardně řeší přidáním speciálního koncového znaku $\$ \notin \Sigma$ na konec slova v roli zarážky.

Daný jazyk generuje například bezkontextová gramatika ($\{S, A, B\}, \{0, 1\}, P, S$) s pravidly $P = \{S \rightarrow 1A, A \rightarrow 0, A \rightarrow 0S, A \rightarrow 1AA, A \rightarrow 0B, B \rightarrow 1, B \rightarrow 1S, B \rightarrow 0BB\}$.

Například slovo 001101 má v této gramatice dvě nejlevější odvození (nakreslete si derivační grafy!):

$S \rightarrow 0B \rightarrow 00BB \rightarrow 001SB \rightarrow 0011AB \rightarrow 00110B \rightarrow 001101$ a

$S \rightarrow 0B \rightarrow 00BB \rightarrow 001SB \rightarrow 0011AB \rightarrow 00110B \rightarrow 001101$.

Pokud bychom v definici gramatiky zvolili jiná přepisovací pravidla, například $P = \{S \rightarrow 0BS, S \rightarrow 0B, S \rightarrow 1AS, S \rightarrow 1A, A \rightarrow 1AA, A \rightarrow 0, B \rightarrow 0BB, B \rightarrow 1\}$, bylo by levé odvození jednoznačné. 20

2. Pro slovo $w = v_1v_2...v_{N-1}v_N$, označme $w^T = v_Nv_{N-1}...v_2v_1$ slovo zapsané v „opačném pořadí“ symbolů.

Jazyk L nad abecedou $\{0, 1, c\}$ obsahuje všechna slova tvaru wcw^T .

Navrhněte pro oba jazyky zásobníkový automat, který je rozpoznává a regulární gramatiku, která je generuje!

Jazyk je zřejmě generován gramatikou ($\{S\}, \{0, 1, c\}, \{S \rightarrow 0S0, S \rightarrow 1S1, S \rightarrow \epsilon\}, S$).

Při konstrukci zásobníkového automatu vystačíme s jediným stavem. Zásobníková abeceda bude mít tři symboly. Zvolme je stejné jako u vstupní abecedy: $\Sigma = \Gamma = \{0, 1, c\}$. Chceme-li užít analogie s předchozí úlohou, ozřejmíme roli zásobníkových symbolů takto

c ... bude počáteční symbol v zásobníku (odpovídá „červenému talíři“),

0 ... se před přečtením centrálního symbolu do zásobníku zapíše (jako se ukládal modrý talíř“), po přečtení centrálního symbolu bude jako vrchol zásobníku porovnávána se vstupem.

1 ... bude obsluhována analogicky („zelený talíř“).

Zásobníkový automat bude mít dva stavy:

PRED – před přečtením centrálního symbolu *c* a

PO – po přečtení centrálního symbolu *c*.

Tabulka přechodové funkce δ je tedy následující. Řádky jsou uspořádané dvojice (stav, vrchol zásobníku), sloupce čtené symboly. Obsah tabulky je řetězec ukládaný do zásobníku a případná změna stavu:

		0	1	C
→	(<i>PRED</i> , <i>c</i>)	Přidej 0, neměň stav	Přidej 1, neměň stav	Přejdi do stavu <i>PO</i>
	(<i>PRED</i> , 0)	Přidej 0, neměň stav	Přidej 1, neměň stav	Přejdi do stavu <i>PO</i>
	(<i>PRED</i> , 1)	Přidej 0, neměň stav	Přidej 1, neměň stav	Přejdi do stavu <i>PO</i>
	(<i>PO</i> , <i>c</i>)	Přejdi do stavu <i>PO</i>	Přejdi do stavu <i>PO</i>	--- = STOP - NE
	(<i>PO</i> , 0)	Odstraň vrchol, neměň stav	--- = STOP - NE	--- = STOP - NE
	(<i>PO</i> , 1)	--- = STOP - NE	Odstraň vrchol, neměň stav	--- = STOP - NE

Turingovy stroje [Turing machines]

Tento nejobecnější model výpočtu předpokládá, že vstupní slova jsou na pásce, která je potenciálně oboustranně nekonečná, i když (podobně jako u zápisníku) je v každém okamžiku práce využívána pouze její konečná část. Toto omezení však není stanoveno předem. V průběhu zpracování slova má čtecí hlava možnost symboly na pásce přepisovat a pohybovat se po pásce oběma směry, tedy i mimo oba okraje zadaného slova.

Formálně je Turingův stroj definován jako uspořádaná šestice $T = (K, \Sigma, \Gamma, \delta, q_0, F)$, kde:

K = je neprázdná konečná množina stavů,

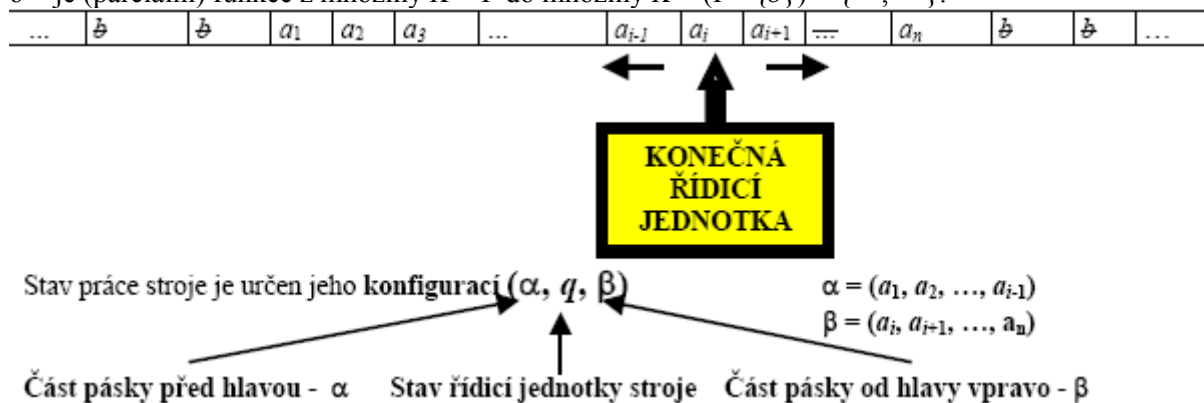
Γ = je neprázdná konečná množina páskových symbolů obsahující speciální prázdný symbol, označovaný b ,

$\Sigma \subset \Gamma$ = je neprázdná množina vstupních symbolů, která prázdný symbol b neobsahuje,

$q_0 \in K$ = je počáteční stav stroje,

$F \subseteq K$ = je neprázdná množina koncových stavů stroje a

δ = je (parciální) funkce z množiny $K \times \Gamma$ do množiny $K \times (\Gamma - \{b\}) \times \{\leftarrow, \rightarrow\}$.



Vstupní páska je rozdělena na políčka obsahující symboly páskové abecedy. Na počátku práce je na pásce zapsáno zkoumané slovo a čtecí hlava je nastavena na jeho levý okraj. Na zbytku pásky jsou zapsány prázdné symboly. V každém kroku práce stroje má řídicí jednotka s konečným počtem svých vnitřních stavů (čtecí hlava) přístup k jednomu políčku pásky. V závislosti na obsahu tohoto políčka a stavu v kterém je může řídicí jednotka provést některé nebo všechny tyto akce:

- přepsat obsah tohoto políčka nějakým symbolem páskové abecedy,
- posunout se o jedno místo vpravo (\rightarrow) nebo vlevo (\leftarrow),
- změnit svůj vnitřní stav.

Při své práci se čtecí hlava může dostat i mimo okraje původně zapsaného slova a to v obou směrech. Vpravo i vlevo. Potřeba místa v průběhu výpočtu není předem omezena.

Počáteční konfigurací stroje je konfigurace $P = (\varepsilon, q_0, w)$, kde $w \in \Sigma^*$. Je-li stroj v konfiguraci (ux, q, yv) , potom konfigurace **po ní bezprostředně následující** bude: (uxy', q', v) v případě, že $\delta(q, x) = (q', x', \rightarrow)$ a $(u, q', xy'v)$ v případě, že $\delta(q, x) = (q', x', \leftarrow)$.

Pokud funkce δ není definována, stroj se zastaví a svoji práci ukončí.

Relace „**konfigurace K následuje konfiguraci P** “ je definována jako tranzitivní uzávěr relace bezprostředního následování. Tedy tak, že existuje konečný řetězec konfigurací počínající P a končící konfigurací K , tak, že každá z nich bezprostředně následuje předchozí konfiguraci.

Koncovou konfigurací nazýváme konfiguraci tvaru (u, q, v) , kde $q \in F$ je nějaký koncový stav. Připomeňme, že pro koncové stavy není přechodová funkce definována a tedy pokud stroj dosáhne na koncovou konfiguraci, nemůže ve své práci pokračovat a zastaví se.

Říkáme, že Turingův stroj přijímá slovo w , tehdy a jenom tehdy, jestliže po počáteční konfiguraci následuje koncová.

Přijetí slova Turingovým strojem tedy znamená jeho zastavení v některém z koncových stavů.

K nepřijetí slova Turingovým strojem může dojít dvěma způsoby:

1. Stroj se zastaví ve stavu, který do množiny koncových stavů nepatří.

2. Stroj se nikdy nezastaví.

Tyto dva typy nepřijetí slova jsou podstatně odlišné.

- V prvním případě získáme v konečném čase kladnou odpověď na otázku zda slovo do jazyka patří, či nikoliv.
- V druhém případě dostaneme v konečném čase (který však nelze předem shora odhadnout) pouze kladnou odpověď. Záporné odpovědi se nedočkáme. *Absence kladné odpovědi může znamenat, že odpověď je záporná, ale i to, že „jsme příliš netrpěliví“ a máme ještě vyčkat.*

Říkáme, že Turingův stroj rozhoduje jazyk L , jestliže se pro každé slovo $w \in \Sigma^*$ jeho výpočet vždy v konečném čase zastaví (a podle stavu v kterém se zastaví rozhodneme, zda slovo do jazyka patří, či nikoliv). Jazyky rozhodnutelné Turingovým strojem se nazývají **rekurzivní jazyky**.

Říkáme, že Turingův stroj přechísľuje nebo též rozpoznává jazyk L , jestliže přijímá všechna slova tohoto jazyka a nepřijímá slova, která do jazyka nepatří (u těchto slov však nemusí výpočet zastavit). Jazyky rozpoznatelné Turingovým strojem se nazývají **rekurzivně přechísľitelné jazyky**, někdy též **rekurzivně vyčísľitelné jazyky** nebo **rekurzivně spočítelné jazyky**.

Snadno se dokáže tak zvaná **Postova věta**: Je-li jazyk L rekurzivně přechísľitelný a jeho doplněk $\Sigma \div L$ také rekurzivně přechísľitelný, je jazyk L (i jeho doplněk) rekurzivní.

Názna proč tomu tak je: Máme-li dva Turingovy stroje T a T' , pro rozpoznání jazyků L a $\Sigma \div L$, můžeme sestrojit Turingův stroj, který „střídá kroky“ obou strojů. Tedy liché kroky provádí podle T a sudé podle T' . Ten svoji práci skončí vždy a oba jazyky rozhodne.

Analogicky jako u konečných automatů a zásobníkových automatů lze uvažovat také **nedeterministický Turingův stroj**. V definici stačí připustit více možných počátečních stavů a funkci δ nahradit relací. Při práci pak stroj může při daném stavu a daném obsahu čteného políčka mít více možností jakým symbolem čtený symbol nahradí, jak změnit svůj stav a zda posunout hlavu o jedno místo vpravo, či vlevo. Podobně jako u předchozích modelů výpočtu mají nedeterministické Turingovy stroje stejnou rozpoznávací schopnost jako deterministické. Přesněji řečeno, každý nedeterministický Turingův stroj lze nahradit deterministickým, který rozpoznává či rozhoduje též jazyk.

Vyšetřují se i různé modifikace Turingových strojů. Uvedme například:

- Turingovy stroje s jednostranně ohraničenou páskou.
- Turingovy stroje s více páskami.
- ...

Rozlišovací schopnost těchto modelů je stejná jako u modelu, který jsme popsali.

Stejnou rozpoznávací schopnost jako Turingovy stroje mají i Zásobníkové automaty s dvěma nebo více zásobníky.

Omezení však představuje podmínka, že Turingův stroj během své práce nesmí překročit hranice původně zadaného slova. Turingův stroj, který **nikdy neopustí políčka pásky na které bylo zapsáno vstupní slovo** (může však jeho symboly libovolně přepisovat) se nazývá **ohraničený automat** (někdy lineárně ohraničený automat).

Pro vztah mezi Chomského hierarchií gramatik a jazyků a modely výpočtu platí následující tvrzení:

Jazyky rozpoznatelné Turingovým strojem (rekurzivně přechísľitelné jazyky) jsou ty a pouze ty jazyky, které lze generovat nějakou generativní gramatikou (gramatikou typu 0 podle Chomského hierarchie).

Jazyky rozpoznatelné ohraničeným automatem jsou kontextové jazyky a pouze kontextové jazyky, tedy jazyky typu 1 podle Chomského hierarchie.

Znamé algoritmy lze realizovat pomocí Turingova stroje. Není to ale právě snadná práce. 24

Turingovy stroje a algoritmická řešitelnost a přechísľitelnost problémů

Turingovy stroje jsou nejobecnějším známým modelem algoritmu a procedury.

Platí tak zvaná **Turingova teze**. Tato teze není matematickou větou, pouze obecně přijímaným *názorem*, že pojem algoritmu, tak jak jej chápeme intuitivně odpovídá možnostem Turingových strojů. Turingova teze říká:

Každý Turingův stroj reprezentuje nějaký algoritmus a každý algoritmus lze realizovat nějakým Turingovým strojem.

Zároveň přijetí této teze umožňuje rozlišit dva případy:

- Řešení problému **algoritmem**, kdy Turingův stroj **rozhoduje** daný jazyk (odpoví ANO i NE v konečném čase).
- Řešení problému **procedurou**, kdy Turingův stroj daný jazyk pouze **rozpoznává** (v konečném čase dá pouze kladnou odpověď).

Pro podporu Turingovy teze lze uvést další model výpočtu.

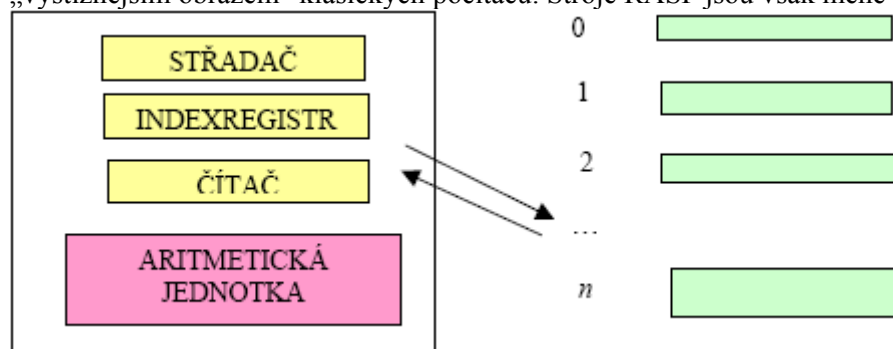
Stroje RASP [Random Access Stored Program]

Tyto modely pracují s čísly. Formalizují v podstatě velmi zjednodušeně von Neumannův model počítače. Pracují s aritmetickou jednotkou, čítačem instrukcí, střadačem, indexregistrem a potenciálně neomezenou přímo adresovatelnou množinou paměťových registrů pro data i program. Realizují velmi omezený operační kód obsahující instrukce výběru z

paměti (LOAD), ukládání do paměti (STORE) základní aritmetické operace a podmíněné a nepodmíněné (v závislosti na obsahu střadače) skoky a operaci zastavení stroje HALT. Připouštějí i možnost nepřímého adresování.

Je zajímavé, že (pochopitelně po náležitém upřesnění předchozího odstavce, naznačujícího velmi nepřesně možnosti strojů RASP) lze dokázat, že **možnosti strojů RASP jsou plně ekvivalentní možnostem Turingových strojů**. Každému Turingovu stroji odpovídá určitý „program“ pro stroj RASP a naopak, každému programu pro RASP lze sestrojit Turingův stroj realizující též výpočet.

Realizovat daný algoritmus pro stroj RASP bývá obvykle snazší než pro Turingův stroj. Tento model je samozřejmě „výstižnějším obrazem“ klasických počítačů. Stroje RASP jsou však méně vhodné pro dokazování obecných vět.



Algoritmicky nerozhodnutelné problémy

Každý Turingův stroj lze popsat konečným počtem symbolů nějaké abecedy. Stačí zakódovat vhodně jeho stavy, páskovou abecedu a přechodovou funkci. Množina všech Turingových strojů je tedy nekonečná, spočetná. Uvažujme pro jednoduchost pouze stroje pracující s abecedou $\{0, 1\}$. Kódy všech Turingových strojů lze uspořádat do posloupnosti T_1, T_2, \dots

Tato úvaha umožňuje myšlenkovou konstrukci tak zvaného **univerzálního Turingova stroje**. Tento stroj provede postupně:

Prvý krok T_1 ,

Druhý krok T_1 , Prvý krok T_2 ,

Třetí krok T_1 , Druhý krok T_2 , Prvý krok T_3 ,

Čtvrtý krok T_1, \dots

Tento stroj rozpoznává všechny jazyky, které rozpoznává libovolný Turingův stroj a rozhoduje všechny jazyky, které rozhoduje libovolný Turingův stroj. *Realizuje tedy „všechny současné i budoucí“ algoritmy programy.*

Všechna slova nad konečnou neprázdnou abecedou tvoří také nekonečnou spočetnou množinu. Její prvky lze též seřadit do nekonečné posloupnosti. Postupujme nyní diagonální metodou. Sestrojme jazyk L tak, že do něj zařadíme ta a jen ta slova, pro která se Turingův stroj umístěný na téže místě posloupnosti se při své práci nezastaví. Tento jazyk není žádným Turingovým strojem rozpoznatelný. Kdyby byl, musel by být umístěn na nějakém místě posloupnosti a musel by dané slovo přijmout. Existují tedy jazyky, které nelze žádným Turingovým strojem rozpoznat. Ty představují nepřečísitelné problémy.

Tyto problémy lze i konkretizovat. Jeden takový uvedeme:

Formulujme tak zvaný **problém zastavení Turingova stroje**:

Je dán Turingův stroj v nějaké své konfiguraci. Existuje algoritmus, který rozhoduje zda se tento stroj zastaví či nikoliv?

Odpověď na tuto otázku je negativní. Lze to dokázat opět postupem v které se užije diagonální metoda. *Tento poznatek je důležitý pro prokázání nemožnosti algoritmicky prověřit v obecném případě úplnou korektnost programů při pokusech rigorózně dokázat jejich správnost.*

V teorii formálních jazyků je řada algoritmicky nerozhodnutelných problémů. Uveďme spíše namátkou některé z nich:

- Je jazyk generovaný kontextovou gramatikou prázdný? Konečný? Nekonečný?
- Je jazyk generovaný gramatikou typu 0 prázdný? Konečný? Nekonečný?
- Platí množinová inkluze a tedy i rovnost mezi jazyky, které jsou generovány oba bezkontextovou gramatikou? Kontextovou gramatikou? Gramatikou typu 0? *Důsledkem algoritmické neřešitelnosti je nemožnost rozhodnout zda dva různé programy mají vždy tutéž funkci.*
- Je jazyk generovaný kontextovou gramatikou, bezkontextovou gramatikou, gramatikou typu 0 regulární?
- ...

Existují i poměrně snadno formulovatelné kombinatorické problémy u kterých lze ukázat, že jsou algoritmicky nerozhodnutelné.

Nejznámější z nich je tak zvaný **Postův korespondenční problém**.

Mějme dvě stejně dlouhé posloupnosti neprázdných slov nad danou abecedou Σ :

$x_1, x_2, \dots, x_k, x_j \in \Sigma^+, j = 1, \dots, k,$

$y_1, y_2, \dots, y_k, y_j \in \Sigma^+, j = 1, \dots, k.$

Řekneme, že tento problém má řešení, pokud „lze z obou těchto posloupností slov sestavit stejné slovo“, přesněji pokud existuje posloupnost přirozených čísel i_1, i_2, \dots, i_m tak, že $x_{i_1}x_{i_2} \dots x_{i_m} = y_{i_1}y_{i_2} \dots y_{i_m}$

Například je-li $\Sigma = \{0, 1\}$ a máme dány posloupnosti slov:

	Posloupnost A	Posloupnost B
i	x_i	y_i
1	1	111
2	10111	10
3	10	0

jde o řešitelný problém, protože při volbě $m = 4$, $i_1 = 2$, $i_2 = 1$, $i_3 = 1$, $i_4 = 3$ máme $x_2x_1x_1x_3 = y_2y_1y_1y_3 = 10111110$.



Při jiné volbě posloupností, například

	Posloupnost A	Posloupnost B
i	x_i	y_i
1	10	101
2	011	11
3	101	011

Lze ale ukázat, že pokusy o takovéto sestavení stejného řetězce by byly marné. Problém řešení nemá.

Lze ukázat, že obecný algoritmus, který by pro každou volbu posloupností slov nad danou abecedou rozhodoval, zda má Postův korespondenční problém řešení, neexistuje. Jazyk definovaný těmi posloupnostmi posloupností, pro které Postův problém řešení má je sice rozpoznatelný, ale není rozhodnutelný. Jazyk určený posloupnostmi posloupností, pro které Postův problém řešení nemá není ani rozpoznatelný.

Petriho síť

Kromě modelů výpočtu o kterých jsme se zmínili existuje ještě řada dalších. Všechny dosud probírané simulovali vždy v podstatě jen sekvenční výpočet. Zmíníme se ještě velmi stručně o jednom, který se hodí i pro modelování paralelních a distribuovaných procesů. Je pojmenován po německém matematikovi Petrim, který jej formuloval v šedesátých letech minulého století.

Proces je modelován orientovaným grafem. Jeho uzly jsou dvojího druhu: MÍSTO (STAVY) a PŘECHODY.

Graf je **bipartitní**. To znamená, že orientovaná hrana může vést pouze ze stavu do přechodu nebo z přechodu do stavu. Ne mezi místy navzájem a mezi přechody navzájem. Je zvykem označovat místa kruhy a přechody obdélníky. Z místa může vést několik orientovaných hran do více stavů, ze stavu několik orientovaných hran do více míst. Místa i přechody mohou mít více vstupních hran.

Formálně je Petriho síť uspořádaná trojice (P, T, F) , kde P a T jsou disjunktní konečné a neprázdné množiny, P množina míst [places] a T množina [transitions] přechodů a $F \subseteq (P \times T) \cup (T \times P)$ je relace přechodů v síti.

Stav sítě je určen polohou **značek** – „pešků“ [token] umístěvaných do míst sítě. V jednom místě může být umístěna žádná značka, jedna značka nebo několik značek, pokud to ohodnocení (kapacita) místa dovolí. Při uskutečnění přechodu se značky přemísťují do nového místa sítě.

Stav sítě je určen počty značek v jednotlivých místech sítě. Výchozí stav je dán rozmístěním značek v místech sítě..

Místa v síti jsou ohodnocena přirozenými čísly 1, 2, ... nebo symbolem ∞ . Ohodnocení znamená maximální počet značek, které mohou být do místa umístěny. Ohodnocení místa symbolem ∞ znamená, že kapacita místa omezena není. Hodnota ∞ se někdy považuje za implicitní.

Ohodnoceny jsou i hrany spojující místa s přechody a přechody s místy. Pro toto ohodnocení se užívají přirozená čísla 1, 2, ... Pokud ohodnocení uvedeno není, předpokládá se jeho implicitní hodnota 1.

U hran MÍSTO \rightarrow PŘECHOD má ohodnocení hran ten význam, že k přesunu značek dojde, pokud všechny vstupní místa tohoto přechodu obsahují aspoň takový počet značek, který je dán ohodnocením těchto hran.

U hran PŘECHOD \rightarrow MÍSTO určuje ohodnocení hran počet značek, která v cílovém místě vzniknou. Počet značek umístěných v celé síti se tedy může měnit.

Práce sítě probíhá evolucí, pokud jsou přechody možné. Síť se vyvíjí obecně nedeterministicky.

Existuje řada podmínek na síť například pro to, aby síť byla „bezpečná“, „ohraničená“, „živá“, „reverzibilní“ apod.

Typické užití Petriho sítě je pro modelování a synchronizaci paralelních výpočtů. Petriho síť umožňují také elegantně hlídat přeplnění vyrovnávacích pamětí v kterých si paralelní procesy předávají data. Například u procesů typu producent \rightarrow konsument.

Další možné užití je pro řešení konfliktních situací, například pro prevenci uváznutí [death lock] při přidělování prostředků paralelně probíhajícím procesům.

9. VÝPOČETNÍ SLOŽITOST

Produkty a jejich jakost

Pro hodnocení softwaru, jako každého produktu je důležitá jeho **jakost (=kvalita)**.

Produkt je výrobek nebo služba nebo kombinace výrobku a služby určený pro dodávku. Software je specifický druh produktu.

Jakost každého produktu je vymezena jako **míra, kterou** vnitřní charakteristiky produktu **splňují požadavky**.

Požadavky mohou být stanoveny předpisy, být obecně předpokládány nebo mohou být stanoveny zvlášť pro daný případ.

Požadavky mají odrážet potřeby uživatele produktu ale též všech dalších zainteresovaných stran.

Jakost informačních systémů a softwaru

Software je zvláštní typ produktu. Pro hodnocení jeho jakosti bylo rozhodnuto sledovat šest charakteristik:

- Funkčnost [functionality].
- Bezporuchovost [reliability].
- Použitelnost [usability].
- Účinnost [efficiency].
- Udržovatelnost [maintainability]
- Přenositelnost [portability].

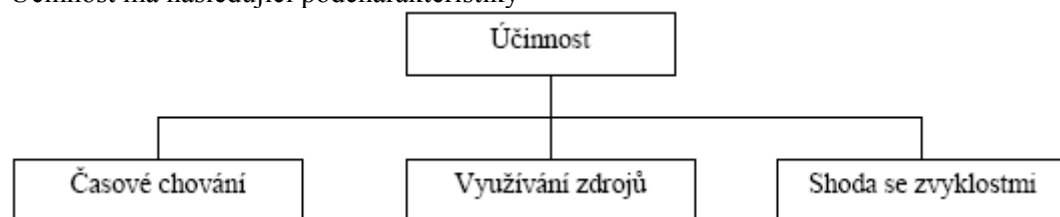
Pro každou z těchto charakteristik je vhodné stanovit požadavky zvlášť a jejich splnění hodnotit odděleně. Tyto charakteristiky lze užít i pro hodnocení jakosti informačních systémů. Jakost informačních systémů však ovlivňují i další složky (hardware, lidé, kteří s IS pracují, organizační opatření, ...).

Každá z uvedených šesti charakteristik se dělí na několik **podcharakteristik**. Měřitelné vlastnosti ovlivňující jeho jakost se nazývají **atributy jakosti**. Atributy jakosti **obvykle vypovídají o několika podcharakteristikách a charakteristikách jakosti zároveň**.

Účinnost softwaru jako podcharakteristika jakosti

V tomto tématu se budeme zabývat především charakteristikou **účinnost**. Ta totiž **závisí v rozhodující míře na užitém algoritmu**.

Účinnost má následující podcharakteristiky



Časové chování [time behavior] je schopnost softwaru poskytnout při provádění požadované funkce v požadovaném čase, poskytovat v požadovaném čase odezvu a zabezpečit požadovanou propustnost výpočtu.

Využívání zdrojů [resource utilization] je schopnost softwaru zajistit výpočet s přijatelnými požadavky na typy a rozsah zařízení výpočetního systému a užitých medií (například rozsah potřebné vnitřní a vnější paměti).

Shoda se zvyklostmi [compliance] je schopnost softwaru respektovat lokálně platné zákony, nařízení, normy a zvyklosti.

Časové chování a potřeba paměti pro software závisí především na užitém algoritmu.

Požadavkům na čas potřebný pro výpočet (jeho dokončení v dávkovém režimu nebo přípravu odezvy v interakčním režimu) se obvykle říká **časová složitost výpočtu (algoritmu)**.

Požadavkům na rozsah paměti potřebné pro výpočet se obvykle říká **prostorová složitost výpočtu algoritmu**.

Časová a prostorová složitost se obvykle souhrnně nazývá **výpočetní složitost**.

Poznámka: Výpočetní složitost je „složitost výpočtu pro stroj“. Nelze ji zaměňovat se za projekční složitost softwaru „(složitost pro člověka)“, která je dána mírou úsilí potřebného k návrhu, implementaci, ověření a údržbě softwaru. Ta je měřena „člověkoměsíci“ práce nebo náklady na vytvoření softwaru. Vytvořit software s vysokou výpočetní složitostí může být snazší, než vytvořit účinný software, který může mít projekční složitost vyšší. Není tomu tak ale vždy.

Složitost algoritmů

Role algoritmu je pro účinnost softwaru **rozhodující, ne však jediná**.

Účinnost informačního systému může být i při vhodném algoritmu degradována řadou faktorů. Uvedme některé nejčastější:

- Málo výkonný hardware užitý v systému (nízká rychlost, malá paměť, ...).
- Nevhodný operační systém výrazně zpomalující práci systému a/nebo s vysokými nároky na zdroje.
- Nevhodné předpisy pro využívání systému vedoucí k jeho málo efektivnímu využívání.
- Nekvalifikovaná obsluha.

Ani vynikající úroveň všech dalších faktorů ovlivňujících účinnost systému však nemůže „dohnat“ nedostatky způsobené nevhodným algoritmem s nepřijatelnou časovou nebo prostorovou složitostí. Jeden z nejvážnějších problémů současné

informatiky je však skutečnost, že pro řešení řady důležitých otázek není algoritmus s přijatelnou výpočetní složitostí k dispozici.

Odhlédneme-li od realizace algoritmu na konkrétním hardwaru a v konkrétním prostředí informačního systému, lze **časovou složitost hodnotit počtem kroků, které algoritmus musí provést než problém rozhodne. Prostorovou složitost pak rozsahem paměti, kterou pro realizaci výpočtu potřebuje.**

Toto hodnocení lze provést na různých úrovních. Krajní přístupy jsou:

- **Rigorózní přístup:** Algoritmus uvažujeme zcela nezávisle na konkrétním počítači jako proces probíhající na TURINGově stroji. Hodnotíme **počet kroků Turingova stroje**, od počátku jeho práce do okamžiku jeho zastavení a rozhodnutí problému. Prostorová složitost je pak dána maximální délkou pásky, které během své práce (od počátku do zastavení) Turingův stroj použil pro záznam znaků abecedy.
- **Praktický přístup:** Odhadujeme počet operací reálného stroje s tím, že (v rozporu se skutečností) považujeme všechny operace za „stejně časově náročné“, respektive odhadujeme pouze počet operací nějakého vybraného typu (třeba aritmetické operace u numerických výpočtů, operace porovnávání položek u algoritmů řazení dat – „třídění“) a ostatní operace zanedbáváme. Vzhledem ke skutečnosti, že při odhadech složitosti jde vždy pouze o trend růstu nároků na čas s růstem rozsahu úlohy a vzhledem k tomu, že délku každé operace lze zhora odhadnout násobkem délky základního taktu stroje, je toto „zjednodušení“ oprávněné. Prostorová složitost je pak dána rozsahem paměti měřené v počtu bytů či bitů, bez ohledu na realizaci paměťových míst.

Ve svých úvahách budeme sledovat spíše praktický pohled s tím, že budeme mít na zřeteli, že pro vyvození přesných závěrů je rigorózní přístup potřebný.

Pesimistická a průměrná výpočetní složitost

Je zřejmé, že u **téhož algoritmu i při stejném rozsahu zpracovávaných dat může výpočet trvat různou dobu** v závislosti na konkrétních hodnotách vstupních dat. Bude totiž pro různá data potřeba provést různý počet kroků. Totéž může platit pro potřebu paměti.

Příkladem může být třídící algoritmus bubblesort, který pokud data na vstupu jsou již setříděna, potřebuje pouze jeden průchod polem pro kontrolu uspořádání. Pokud je vstup délky n uspořádán inverzně k požadovanému, bude potřebovat $n - 1$ průchodů. Obdobný příklad lze nalézt i pro prostorovou složitost.

Je tedy třeba rozlišovat

- **Pesimistickou výpočetní složitost** – definovanou jako složitost (časovou nebo prostorovou), které je dosaženo v „nejhorším možném případě“ pro daný rozsah zpracovávaných dat.

- **Průměrnou výpočetní složitost** – definovanou jako aritmetický průměr složitostí v pro konkrétní vstupy daného rozsahu s přihlédnutím k pravděpodobnostnímu rozložení těchto možných vstupů.

Odhad pesimistické výpočetní složitosti bývá jednodušší než odhad složitosti průměrné.

Nelze jednoznačně zodpovědět otázku, která z obou složitostí je „důležitější“. U časové složitosti náročných výpočtů, které se často opakují, ale kde není doba pro ukončení konkrétního výpočtu kritická, dá jistě lepší obrázek o účinnosti průměrná složitost. U časově kritických procesů, například přímá regulace zařízení, kde prodleva může způsobit havarii je rozhodující znát pesimistickou časovou složitost. U prostorové složitosti je zpravidla rozhodující též pesimistická varianta. Překročení prostoru totiž znamená odepření funkce.

V dalších úvahách, pokud budeme hovořit o výpočetní složitosti bez uvedení zda nám jde o průměrnou či pesimistickou, budeme mít vždy na mysli **pesimistickou složitost**.

Dále se soustředíme především na časovou složitost. Při praktických aplikacích bývá totiž častěji omezujícím faktorem než složitost prostorová.

Úvahy o prostorové složitosti jsou však analogické.

Míry pro časovou složitost

Časové chování jako podcharakteristika účinnosti informačního systému nebo softwarového produktu lze obvykle charakterizovat různými atributy v závislosti na typu softwaru. Pro systémy pracující v interakčním režimu to bývá atributem a příslušnou mírou:

- **Doba odezvy** [Response time], definovaná jako doba mezi ukončením uživatelské interakce se systémem a reakcí systému po provedení požadované operace. Přitom uživatelem může být člověk i technické zařízení (například v případě automatizovaného řízení nějakého výrobního procesu). Důležitá může být jak průměrná tak i maximální doba odezvy.

V případě dávkového zpracování bývá atributem a mírou:

- **Doba obrátky** [Turnaround time], definována jako doba mezi zavedením vzájemně souvisejících úloh [task] nebo dávky [job] a jejím ukončením. Opět může být důležitá průměrná doba obrátky i maximální doba obrátky.

Pokud nás zajímá množství práce, kterou může výpočetní systém zastat, bývá měřeným atributem a mírou:

- **Propustnost** [Throughput], definovaný jako počet úloh, které systém v daném časovém intervalu dokončí. Lepší obrázek ovšem dá tak zvaný „vážený počet“, kdy každou dokončenou úlohu hodnotíme potřebným počtem jednotek podle její obtížnosti, respektive důležitosti. Tento atribut se hodí spíše pro charakteristiku systémového prostředí (účinnosti hardwaru a systémového softwaru – operačního systému a jeho nástaveb) než pro hodnocení uživatelského softwaru a jím užívaných algoritmů. Bývá sledována průměrná hodnota za dlouhá období.

Ve všech těchto případech je zřejmé, že sledovaná doba i počet úloh nezávisí pouze na algoritmu, ale velmi výrazně i na rozsahu dat, s kterými algoritmus pracuje. Jde buď o vstupní data úlohy nebo o rozsah datové základny s kterou program pracuje, případně o oboje.

Tomuto údaji budeme říkat **rozměr vstupu** [imput size, imput volume]. Bývá charakterizován přirozeným číslem.

Časovou složitost nelze tedy korektně měřit základní měřením, kde hodnotou míry je číslo. **Časovou složitost lze měřit pouze zobecněným měřením, kde mírou složitosti je funkce, která rozměru vstupu přiřazuje čas, respektive počet operací potřebných pro provedení výpočtu.**

Totéž platí pro míry prostorové složitosti. Zde opět není mírou číslo ale funkce, která přiřazuje rozměru vstupu **rozměr paměti nezbytné pro realizaci výpočtu.**

U obou těchto funkcí se zajímáme především o jejich růst v závislosti na růstu rozměru vstupu. Tedy o tak zvané **asymptotické chování** této funkce při neomezeném růstu rozměru vstupu.

Toto chování posuzujeme porovnáním s chováním běžně známých funkcí o jejichž růstu máme určitou představu.

Odhad výpočetní složitosti. Symboly O , o a Θ

Pro odhady funkce f funkcí g v intervalu $(0, \infty)$ zavedeme tyto definice:

Nechť f a g jsou funkce definované na množině přirozených čísel N do množiny přirozených čísel N nebo funkce definované na množině kladných reálných čísel R^+ s hodnotami rovněž v R^+ . Řekneme, že:

- $f \in O(g)$ (čteme: "f je ve velkém O g", nebo méně přesně zkráceně "f je velké O g") právě když $\exists k \in N \exists n_0 \in N : \forall n > n_0 : f(n) \leq k \cdot g(n)$;
- $f \in o(g)$ (čteme: "f je v malém o g", nebo méně přesně zkráceně "f je malé o g") právě když $\exists k \in N \exists n_0 \in N : \forall n > n_0 : f(n) < k \cdot g(n)$;
- $f \in \Theta(g)$ (čteme: "f je v theta g", nebo méně přesně zkráceně "f je theta g") právě když je současně $f \in O(g)$ a $g \in O(f)$.

Názorně řečeno:

- $f \in O(g)$ označuje že funkce f asymptoticky roste nejvýše tak rychle jako funkce g .
- $f \in o(g)$ označuje, že funkce f asymptoticky roste pomaleji než funkce g .
- $f \in \Theta(g)$ označuje, že asymptotický růst funkcí f a g je srovnatelný

Podmínku " $\exists n_0 \in N : \forall n > n_0$:" lze vyjádřit jako „pro skoro všechna n “ to znamená „pro všechny hodnoty n s výjimkou konečně mnoha n “.

Někdy se můžeme setkat i se zápisem „ $f = O(g)$ “ místo $f \in O(g)$, se zápisem $f = o(g)$ místo $f \in o(g)$ a se zápisem $f = \Theta(g)$ místo $f \in \Theta(g)$. Tyto zápisy se běžně užívají, nejsou však zcela korektní, protože na levé straně rovností jsou funkce a na pravé množiny funkcí daných vlastností. Je totiž $O(g) = \{f : \exists k \in N \exists n_0 \in N : \forall n > n_0 : f(n) \leq k \cdot g(n)\}$

Analogicky pro $o(g)$ a $\Theta(g)$.

Někdy se pro odhad asymptotického růstu funkcí užívají i symboly: Ω (a také ω) **pro odhady zdola**. Zde nebývá význam vždy jednotný. Někteří autoři užívají definici:

- $f \in \Omega(g)$ právě když $\exists k \in N \exists n_0 \in N : \forall n > n_0 : f(n) \geq k \cdot g(n)$.

Jiní autoři

- $f \in \Omega(g)$ právě když neplatí $f \in o(g)$, tedy právě když $\exists k \in N \forall n_0 \in N \exists n > n_0 : f(n) > k \cdot g(n)$.

Užití symbolů Ω a ω však není nezbytné. Se symboly O , o a Θ vystačíme.

Zřejmě je $f \in \Theta(g)$ právě když $\exists c \in N \exists n_0 \in N : \forall n > n_0 : 1/c \cdot g(n) \leq f(n) \leq k \cdot g(n)$.

Jasně jsou i množinové inkluze, například: $\Theta(g) \subseteq O(g)$ a $o(g) \subseteq O(g)$ pro každou funkci g .

Dále zřejmě platí: $(f_1 \in \Theta(g_1) \wedge f_2 \in \Theta(g_2)) \Rightarrow f_1 + f_2 \in \Theta(\max(g_1 + g_2))$ a $(f_1 \in \Theta(g_1) \wedge f_2 \in \Theta(g_2)) \Rightarrow f_1 \cdot f_2 \in \Theta(g_1 \cdot g_2)$.

Obdobně pro O a o .

Dále platí:

Jestliže $\lim_{n \rightarrow \infty} f(n) / g(n) = 0$, potom $f \in o(g)$.

Jestliže $\lim_{n \rightarrow \infty} f(n) / g(n) = c \neq 0$, potom $f \in \Theta(g)$.

Jestliže $\lim_{n \rightarrow \infty} f(n) / g(n) = +\infty$, potom $g \in o(f)$.

Definici symbolů O , o a Θ však v plném rozsahu nelze nahradit limitami, protože tyto limity nemusí existovat.

Typické třídy výpočetní složitosti

Pro porovnání funkce časové, respektive prostorové, složitosti se známými funkcemi se nejčastěji užívají tyto třídy složitosti:

$\Theta(1)$ – růst nezáleží na rozměru vstupu

$\Theta(\log n)$ – logaritmický růst (základ logaritmu není podstatný – proč?)

$\Theta(n)$ – lineární růst (složitost je přímo úměrná rozměru dat)

$\Theta(n \cdot \log n)$ – tento růst dosahují „chytřejší“ algoritmy řazení („třídění“)

$\Theta(n^2)$ – kvadratický růst, dosahují jednoduché algoritmy řazení

$\Theta(n^3)$ – kubický růst typický pro některé operace s maticemi a algoritmy řešení soustav lineárních rovnic

$\Theta(n^k)$ pro nějaké přirozené číslo $k \in N$ – polynomiální růst

$\Theta(k^n)$, $k > 1$ – exponenciální růst $k > 1$ (nejčastěji $k = 2$, tedy $\Theta(2^n)$)

$\Theta(n!)$ – faktoriální růst.

Samozřejmě, pokud je součástí algoritmu vstup dat, je minimální možná časová složitost algoritmu $\Theta(n)$. Pokud nás však zajímá pouze časová složitost nějakého kroku algoritmu (například pro odhad doby odezvy pro kterou hraje podstatnou roli vyhledání záznamu podle klíče v souboru s přímým přístupem, který je podle klíče uspořádán), může být složitost nižší. (V uvažovaném případě výběru ze seřazených přímo přístupných dat $\Theta(\log n)$).

Minimální prostorová složitost u algoritmů, které vyžadují mít v paměti současně celý vstup je samozřejmě $\Theta(n)$.

Pro analýzu výpočetní složitosti algoritmů je důležitá otázka **co je vhodnou mírou pro rozměr vstupu?**

Uvažme případ běžně známého algoritmu pro výpočet řešení soustavy n lineárních rovnic pro n neznámých eliminační metodou. Pro vyloučení jedné neznámé je třeba úpravami rozšířené matice soustavy rovnic získat nuly do všech $(n - 1)$ položek příslušného sloupce s výjimkou jediného řádku. Poté lze proměnnou vyloučit. Toho lze dosáhnout odečtením určitého násobku řádku z kterého proměnnou vylučujeme ode všech $(n - 1)$ zbývajících řádků. Odečtení vektoru o $(n + 1)$ složkách představuje vždy $\Theta(n)$ operací. Z naznačené úvahy (kterou je možné upřesnit) vyplývá, že počet potřebných aritmetických operací (a snadno nahlédneme, že i všech operací) je $f(n) \in \Theta(n^3)$, kde n je počet rovnic a neznámých řešené soustavy. Vstupní data pro algoritmus však kromě zadání řádu soustavy n představují zadat matici soustavy a pravé strany rovnic, tedy $n \cdot (n + 1)$ reálných čísel. Jaká je tedy složitost algoritmu?

$\Theta(n^3)$ nebo $\Theta(n^{3/2})$?

Zdá se, že korektní je druhá možnost. Přesto se častěji užívá prvá. Vždy je tedy nutné si ověřit jak je měřen rozměr vstupu algoritmu.

Další skutečnost, na kterou je třeba brát zřetel, je ta, že operace, které bereme v úvahu, mají argumenty omezené. Pokud jde o práci s čísly, zpracovávají vždy pouze hodnoty daného rozsahu. Pokud má algoritmus pracovat s „libovolně velkými čísly“, je třeba operace interpretovat pomocí operací, které jsou k dispozici. Počet strojových instrukcí potřebných na jednu aritmetickou operaci s libovolně velkým číslem je pak úměrný počtu bitů potřebných k reprezentaci tohoto čísla, tedy logaritmu velikosti tohoto čísla. To je třeba brát v úvahu například při zkoumání, zda je libovolně velké číslo prvočíslem, respektive při pokusu rozkladu libovolně velkých přirozených čísel na součin. Časová složitost takovýchto algoritmů je důležitou otázkou při zkoumání některých důležitých šifrovacích algoritmů.

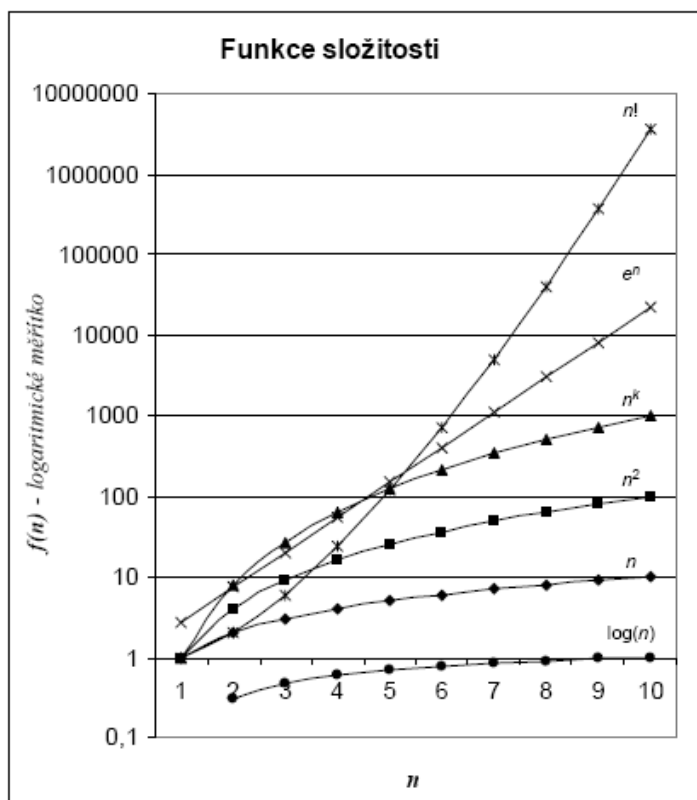
Růst funkcí užívajících pro asymptotické chování je uveden v následující tabulce. Řádky představují rozměr vstupu. Sloupce typické funkce. Hodnoty v tabulce udávají čas při rychlosti 10 μ s pro jeden krok.

	$\log_2 n$	n	$N \cdot \log_2 n$	n^2	n^3	n^4	2^n	$n!$
2	10 μ s	20 μ s	20 μ s	40 μ s	80 μ s	160 μ s	40 μ s	20 μ s
5	23,1 μ s	50 μ s	116 μ s	250 μ s	1,25ms	6,25ms	320 μ s	1,2ms
10	33,2 μ s	100 μ s	332 μ s	1ms	10ms	100ms	10,2 ms	1,17s
15	39,1 μ s	150 μ s	587 μ s	2,25ms	33,8ms	507ms	328ms	15,1 days
20	43,2 μ s	200 μ s	864 μ s	4ms	80ms	1,6s	10,5s	771000 years
25	46,4 μ s	250 μ s	1,16ms	6,25ms	156ms	3,91s	5,59min	∞
30	19,1 μ s	300 μ s	5,73ms	9ms	270ms	37,5s	2,98h	∞
50	56,4 μ s	500 μ s	28,2ms	25ms	1,25s	1,04min	357years	∞
100	66,4 μ s	1ms	6,64ms	100ms	10s	16,7min	∞	∞
200	76,4 μ s	2ms	15,3ms	400ms	1,34min	4,47h	∞	∞
500	89,4 μ s	5ms	44,4ms	2,5s	4,17min	13,9h	∞	∞

Následující tabulka je ještě výmluvnější. Zachycuje se možné rozšíření rozměru vstupu v případě, že algoritmus je zpracováván na rychlejší stroji. Sloupce udávají kolikrát je nové zpracování rychlejší než původní. Řádky odpovídají třídám složitosti. Předpokládá se, že původní přípustný rozsah úlohy (na „starém“ počítači) odpovídá rozměru vstupu 100. V tabulce je nový přípustný rozsah (omezení na „novém, rychlejším“ počítači).

	Původní	2-krát	5-krát	10-krát	100-krát	1 000-krát
$\Theta(n)$	100	200	500	1 000	10 000	100 000
$\Theta(n^2)$	100	141	223	316	1 000	3 162
$\Theta(n^3)$	100	125	170	215	464	1 000
$\Theta(2^n)$	100	101	102	103	107	110
$\Theta(n!)$	100	100	101	101	101	101

Doplňme pro ilustraci ještě grafické znázornění růstu funkcí nejčastěji používaných pro stanovení tříd složitosti. Připomeňme, že graf má logaritmické měřítko na ose y, která udává čas.



Z obou tabulek i z grafu je zřejmé, že mezi polynomiální složitostí ($O(n^k)$ pro nějaké $k \in \mathbb{N}$) a exponenciální složitostí je propastný rozdíl. Pro vysoké k může být růst $O(n^k)$ nepříjemný, nicméně o užití takového algoritmu na vysoce rychlých strojích lze uvažovat. Pro algoritmy s exponenciálním nebo faktoriálním růstem časové složitosti však zrychlení výkonu nic nezachrání. **Tedy algoritmy s exponenciální nebo ještě vyšší časovou složitostí jsou pro netriviální rozměr vstupu prakticky nepoužitelné.**

Je třeba ještě připomenout, že časová složitost neříká nic o času, který je potřeba pro vyřešení úlohy pro konkrétní vstup poměrně malého rozsahu. **Postihuje pouze trend růstu pro velké objemy dat.** Potřebný čas pro výpočet v reálném případě je dán odhadem $f(n) \leq k \cdot g(n)$, který závisí nejen na hodnotě funkce g , ale též na konstantě k . Může se stát, že „horší“ (z hlediska růstu funkce g) algoritmus dá pro relativně malý rozměr vstupu výsledek v kratším čase (díky nižší hodnotě konstanty k). Tak například algoritmus quicksort má pesimistickou složitost $\Theta(n^2)$, pro „malé“ úlohy však zpravidla pracuje rychleji než algoritmus třídění haldou heapsort, který má garantovanou pesimistickou časovou složitost $\Theta(n \cdot \log n)$, je však implementačně náročnější.

Analýza složitosti konkrétních algoritmů

U konkrétních algoritmů je stanovení počtu všech operací, které je třeba provést velmi pracné. Proto se zpravidla započítávají pouze vybrané typy operací. Například

☉ U numerických algoritmů **pouze aritmetické operace** (+, -, *, /).

☉ U operací řazení dat **pouze operace porovnávání hodnot**.

Tento postup je korektní, pokud je splněna následující podmínka:

Existuje přirozené číslo K , takové, že pokud označíme $C_SELECTED$ počet provedení všech operací z vybrané množiny a C_ALL je počet provedení všech operací algoritmu, platí: $C_ALL \leq K \cdot C_SELECTED$.

Tento předpoklad bývá splněn.

Dále je třeba vzít v úvahu omezení na délku operandů.

Pokud takové omezení není z povahy algoritmu apriori dáno, je třeba každou operaci počítat s vahou, která je úměrná součtu délek operandů, nebo reálněji součtu logaritmů těchto délek.

Tak například jednoduchý algoritmus pro testování zda přirozené číslo N je či není prvočíslem vyžaduje postupně testovat dělitelnost čísla od 2 do \sqrt{N} . Jeho časová složitost v závislosti na N není však $\Theta(\sqrt{N})$, protože žádný počítač nemůže mít ve svém kódu operaci celočíselného dělení libovolně velkým číslem.

Příklady analýzy časové složitosti některých vybraných algoritmů budou naším dalším tématem.

10. TŘÍDY VÝPOČETNÍ SLOŽITOSTI

Toto téma navazuje na teoretický výklad výpočetní složitosti algoritmů. Přesná matematická odvození výsledků by bylo potřeba provádět užitím matematických modelů výpočtu, třídy Turingova stroje nebo stroje RASP pro program s přímým přístupem k paměti.

Rozměr vstupu [size of input] při tomto rigorózním přístupu by byl počet míst na pracovní pásce Turingova stroje v jeho základní konfiguraci, popsaných jinými znaky než mezerami \square , u stroje RASP pak počtem čísel omezené velikosti na vstupu úlohy. Pokud by algoritmus měl pracovat s daty, jejichž rozsah není apriorně omezen, museli bychom tato data pro Turingův stroj kódovat v pevně dané konečné abecedě, v případě čísel zpracovávaných strojem RASP pak v nějaké (nejspíše binární) číselné soustavě. To by vedlo k potřebě $\log_2(x + 1) + 1$ paměťových míst pro jeden vstup, který může nabývat nejvýše x různých hodnot. Pro algoritmus zpracovávající celá čísla bez omezení délky přispívá tedy jedno číslo x na rozsah vstupu hodnotou $\log_2(|x| + 1) + 1$.

Prostorovou složitost algoritmu budeme hodnotit stejně jako rozsah vstupních dat s tím rozdílem, že budeme brát v úvahu maximální počet položek pásky Turingova stroje na kterých se kdykoliv v průběhu výpočtu vyskytnou nemezerové znaky. U modelu RASP pak opět maximální počet paměťových míst, která budou kdykoliv v průběhu výpočtu používána.

Časovou složitost pak budeme hodnotit podle počtu kroků, které provede Turingův stroj od začátku práce do svého zastavení. U modelu RASP pak jako počet operací, které se provedou do okamžiku, kdy výpočet narazí na instrukci **HALT**, při čemž ovšem každá instrukce bude započítávána s váhou úměrnou součtu rozměrů všech svých operandů.

Algoritmus považujeme za prostředek k řešení problémů na některém z těchto modelů výpočtu.

Výpočetní problém chápeme jako uspořádanou trojici (IN, OUT, P) , kde vstup IN je množina konečných posloupností nad danou abecedou (slov) splňujících danou podmínku C_{IN} , výstup OUT je množina posloupností nad nějakou (touž nebo jinou než je abeceda pro IN) abecedou, reprezentující výstup. Relace $P \in IN \times OUT$ je podmínka, která vstupům přiřazuje „povolené - správné“ výstupy. Pokud má být pro daný vstup výstup definován jednoznačně, je relace P funkcí na množině IN , která každému povolenému vstupu přiřazuje jediný výstup, řešení problému, které splňuje podmínku $P(IN, OUT)$. Můžeme pak psát $p(X) = Y \Leftrightarrow (\forall X: X \in IN \Rightarrow P(X, Y))$.

Speciální typ výpočetního problému je **rozhodovací problém** (problém ANO / NE). V něm je množina OUT dvouprvkovou množinou booleovských pravdivostních hodnot $OUT = \{TRUE, FALSE\}$.

Časová složitost problému je pak definována takto:

Řekneme, že problém (IN, OUT, P) patří do třídy složitosti $T(f(n))$, kde f je funkce $N \rightarrow N$ (N je množina všech přirozených čísel), který může být vyřešen **nějakým** Turingovým strojem nebo strojem RASP v čase $O(f(n))$ **pro všechny** přípustné vstupy o rozměru n z IN . Analogicky problém (IN, OUT, P) je prostorové složitosti $S(f(n))$, je-li možné jej vyřešit **nějakým** Turingovým strojem nebo strojem typu RASP **pro všechny** přípustné vstupy o rozměru n s prostorovými nároky $O(f(n))$.

Třídy časové a prostorové složitosti tedy představují vždy pouze **horní odhady složitosti** a to složitosti v **pesimistickém případě**. Týkají se **problému**, nikoliv konkrétního algoritmu. Dokázat, že problém patří do třídy složitosti $O(f(n))$ znamená najít **aspoň jeden** algoritmus jeho řešení, realizovatelný Turingovým strojem nebo strojem RASP, příslušné (časové nebo prostorové) složitosti $O(f(n))$. Prokázat, že problém do třídy složitosti $T(f(n))$, respektive $S(f(n))$ nepatří znamená dokázat, že **žádný** takový algoritmus s takovým horním odhadem pesimistické složitosti pro Turingův stroj nebo stroj RASP existovat nemůže. To znamená nalézt dolní odhad pesimistické složitosti pro všechny algoritmy řešící daný problém. To může být nesnadný úkol. Často takový algoritmus přes velké úsilí neznáme. To však ještě zdaleka není důkazem jeho neexistence.

Třídy složitosti poněkud závisí na volbě modelu výpočtu. Nemusí být totožné pro Turingovy stroje a stroje RASP. Turingovy stroje pracují vzhledem k sekvenčnímu přístupu k políčkům na pásce u některých algoritmů „pomaleji“. Pro posuzování, zda je algoritmus z hlediska složitosti akceptovatelný či nikoliv však tento rozdíl není příliš podstatný.

Zatím jsme uvažovali deterministické Turingovy stroje. Nedeterministické Turingovy stroje mají sice v principu tutéž rozhodovací a rozpoznávací schopnost vzhledem k formálním jazykům (oba modely rozpoznávají každý jazyk generovaný gramatikou typu 0), může zde být však být podstatný rozdíl pokud jde o potřebný počet kroků stroje.

Třídy nedeterministické složitosti

Nedeterministický Turingův stroj řeší rozhodovací problém (ANO / NE) jestliže pro všechny přípustné vstupy z množiny IN existuje aspoň jedna posloupnost po sobě následujících konfigurací tohoto stroje, taková, že se Turingův stroj po přečtení celého vstupu zastaví v závislosti na stavu v kterém se zastaví odpověď ANO / NE.

Jestliže **existuje nějaký nedeterministický Turingův stroj**, který řeší rozhodovací problém pro všechny přípustné vstupy z množiny IN o rozměru n za $O(f(n))$ kroků (při vhodné volbě posloupnosti svých po sobě bezprostředně následujících konfigurací), potom říkáme, že tento rozhodovací problém patří **do třídy nedeterministické časové složitosti** $NT(f(n))$.

Třídy nedeterministické prostorovou složitostí lze definovat obdobně.

Názorně si můžeme situaci přiblížit takto:

Třídy (deterministické) časové a prostorové složitosti $T(f(n))$ a $S(f(n))$ představují horní odhady **složitosti nalezení** řešení problémů „bez nápovědi“. Tedy čas a prostor pro nalezení neznámého řešení problémů.

Třídy nedeterministické časové a prostorové složitosti $NT(f(n))$ a $NS(f(n))$ představují horní odhady **složitosti ověření**, zda nalezené řešení skutečně řešením je. Tedy třídy složitosti „zkoušky“ správnosti řešení, které již známe.

Je zřejmé, že problémy, které mají neakceptovatelnou NT nebo NS složitost nejsou pro praxi příliš zajímavé. Nemůžeme-li je v prakticky realizovatelné době zkontrolovat nebo nestačí-li nám místo na jejich zápis, nebudeme je asi ani v životě k ničemu potřebovat. Z nich nás tedy „hlava bolet nemusí“.

U problémů, které mají přijatelnou NT i NS složitost můžeme řešení na přijatelném rozsahu můžeme zapsat i zkontrolovat. Můžeme je tedy využít. U řady důležitých problémů tohoto typu však neznáme algoritmus řešení, který by patřil do přijatelné třídy časové nebo prostorové deterministické složitosti. Řešení tedy neumíme při netriviálním rozměru vstupu nalézt.

P-těžké, NP-těžké a NP-úplné problémy

V neformálních úvahách o časové složitosti jsme ukázali, že je propastný rozdíl mezi případem, kdy lze složitost odhadnout shora nějakou mocninou (nebo mnohočlenem) rozměru vstupu a případem, kdy tomu tak nelze a kdy čas roste rychleji než libovolný polynom, například exponenciálně.

Jako horní hranice pro algoritmus, aby jej bylo možné pokládat (byť někdy s výhradami) za „rozumně použitelný“ pro netriviální rozměr vstupu se zpravidla bere omezení času i prostoru nějakou mocninou rozměru vstupu. Tedy omezení na

$$\bigcup_{k=0}^{\infty} O(n^k).$$

To vede také na rozdělení problémů na problémy řešitelné v polynomiálním čase a v polynomiálním prostoru. Tyto třídy problémů jsou definovány takto.

Polynomiálně časově těžké nebo je **časově P-těžké problémy** jsou problémy třídy $P_{TIME} = \bigcup_{k=0}^{\infty} T(n^k)$

Třída P_{TIME} je tatáž, nezávisle na tom zda algoritmus modelujeme na Turingově stroji nebo na stroji RASP. Lze totiž ukázat, že pro každý stroj MR typu RASP existuje Turingův stroj MT , který při stejných podmínkách na IN a mezi IN a OUT realizuje výpočet tak, že platí pro nějakou „malou“ konstantu c vztah $T_{MR}(n) \leq (T_{MT}(n))^c$. Totéž platí i naopak pro realizaci výpočtu stroje RASP na Turingově stroji.

Problémy, které do této třídy nepatří je třeba považovat za „prakticky neřešitelné“.

Samozřejmě problém časové složitosti $n^{1000000}$ je podle této klasifikace také P-těžký, ale řešení algoritmem složitosti $\theta(n^{1000000})$ bychom se nejspíše nedočkali. Obvykle však u takovýchto problémů lze nalézt polynomiální algoritmus omezený nižší mocninou n (zpravidla stačí $n \leq 6$).

Podobně lze problémy klasifikovat z hlediska nedeterministické časové složitosti. Třída $NP_{TIME} = \bigcup_{k=0}^{\infty} NT(n^k)$

Zahrnuje všechny problémy, které lze nedeterministicky řešit (známé řešení zkontrolovat) pomocí nějakého nedeterministického stroje v konečném čase. Těmto problémům se říká **časově NP-těžké problémy**.

Je třeba upozornit na to, že písmenko „N“ v tomto kontextu nelze číst jako „not“. Znamená nedeterministicky. Každý P-těžký problém je samozřejmě i NP-těžký. Není známo zda existují problémy, které jsou NP-těžké, ale nejsou P-těžké. Soudí se že ano, důkaz však podán nebyl. Třída **NOT** P_{TIME} by označovala třídu problémů, o kterých je známo, že v polynomiálním čase je řešit nelze. Pokud tyto problémy nepatří zároveň do třídy NP_{TIME} , nejsou příliš zajímavé.

Zcela analogicky lze definovat třídy pro prostorovou složitost: $P_{SPACE} = \bigcup_{k=0}^{\infty} T(n^k)$ a $NP_{SPACE} = \bigcup_{k=0}^{\infty} NT(n^k)$

Třídy NP_{TIME} , P_{SPACE} a NP_{SPACE} také nezávisí na volbě modelu výpočtu (Turingův stroj nebo stroj RASP).

Pokud se hovoří o třídách P a NP nebo o P-těžkých a NP-těžkých problémech, bez uvedení zda jde o časovou nebo prostorovou složitost, míníme tím zpravidla složitost časovou. Ta totiž bývá pro praktické aplikace kritičtější než prostorová.

Obecně plat, že pokud lze problém řešit nedeterministickým Turingovým strojem s prostorovou složitostí $O(n^k)$, lze jej

řešit i deterministickým Turingovým strojem s prostorovou složitostí $O(n^{2k})$. Tedy $P_{SPACE} = NP_{SPACE}$.

Pro časovou složitost se však nic podobného dokázat nepodařilo. Samozřejmě platí, že $P_{TIME} \subset P_{SPACE}$.

Máme tedy $P_{TIME} \subseteq NP_{TIME} \subset P_{SPACE} = NP_{SPACE}$.

Prvá inkluze $P_{TIME} \subseteq NP_{TIME}$ představuje vážný problém teorie výpočtů. Přes velké a dlouhodobé soustředění řady vynikajících vědců na otázku zda je tato inkluze ostrá nebo nikoliv dodnes neznáme odpověď na otázku zda platí $P_{TIME} \subsetneq NP_{TIME}$ nebo $P_{TIME} = NP_{TIME}$?

Je známo mnoho prakticky důležitých problémů třídy NP, pro které neznáme žádný algoritmus, který by je řešil v polynomiálním čase. Důkaz, že takový algoritmus neexistuje však podán nebyl. Nevíme zda neexistuje nebo zda jsme jej dosud nenalezli. Většina vědců z této oblasti však již ztratila naději, že takový algoritmus bude objeven. Některé takové problémy uvedeme v dalším odstavci.

Zajímavý výsledek představuje nalezení tak zvaných NP-úplných problémů. Tyto problémy patří do třídy NP-těžkých problémů, pro které polynomiální deterministický algoritmus řešení znám není. Mají navíc tu vlastnost, že kdybychom

aspoň pro jediný z těchto problémů polynomiální algoritmus objevili, dovedli bychom v polynomiálním čase řešit všechny NP-těžké problémy a dokázali tak rovnost $P_{TIME} = NP_{TIME}$. Proto jsou problémy této třídy tak zajímavé.

Řekneme, že problém **P lze převést v polynomiálním čase na problém Q** a budeme to zapisovat $P \leq Q$, jestliže existuje Turingův stroj s polynomiální časovou složitostí, který transformuje každý přípustný vstup u pro problém P na jím generovaný výstup v, který je vstupem pro problém Q tak, že Turingův stroj odpovídá ANO / NE na vstup v problému Q tehdy a jenom tehdy, když odpovídá ANO / NE na vstup u problému P a odpověď je stejná.

Zřejmě je:

$$(P \leq Q) \wedge (Q \leq R) \Rightarrow (P \leq R);$$

$$(P \leq Q) \wedge (Q \in P_{TIME}) \Rightarrow (P \in P_{TIME});$$

$$(P \leq Q) \wedge (Q \in NP_{TIME}) \Rightarrow (P \in NP_{TIME}).$$

Problém P se nazývá NP-úplný, je-li NP-těžký a pro všechny NP-těžké problémy Q platí, že $Q \leq P$.

Zřejmě jestliže je P NP-těžký problém a $Q \leq P$ pro nějaký NP-úplný problém Q, potom je také P NP-úplný problém. NP-úplné problémy jsou tedy ty „nejobtížnější“ problémy třídy NP_{TIME} . Kdybychom uměli kterýkoliv z nich řešit v polynomiálním čase, bylo by $NP_{TIME} = P_{TIME}$ a věděli bychom si rady se všemi problémy, které dosud vzdorují. Naneštěstí se zdá, že moc naděje takové řešení nalézt není.

Na druhé straně, řešení některého z NP-úplných problémů a tedy důkaz, že $P_{TIME} = NP_{TIME}$ by znamenalo velmi vážné ohrožení bezpečnosti dat. Problém dešifrování dat zašifrovaných pomocí asymetrických šifer a problém prolomení elektronického podpisu jsou totiž NP-těžké problémy.

V dalším odstavci uvedeme některé NP-úplné problémy.

Některé příklady NP-úplných problémů

Nejčastěji citovaným NP-úplným problémem je **problém splnitelnosti booleovské formule** [SAT-problem]. Pro formuli zapsanou v konjunktivně disjunktivní normální $\bigwedge_{j=1}^n \bigvee_{i=1}^{m_j} p_{i,j}$, kde $p_{i,j}$ jsou literály (logické proměnné nebo jejich negace)

jde o to zjistit, zda existuje nějaké pravdivostní ohodnocení TRUE / FALSE všech proměnných ve formuli, tak, aby výsledná pravdivostní hodnota formule byla TRUE, či nikoliv. Pro splnitelnost disjunktivně konjunktivní normální formy je formulace analogická. Právě tak problémy zda je daná formule tautologií nebo kontradikcí.

Tento problém je zřejmě NP-těžký. Pokud takové ohodnocení existuje a někdo nám jej „chytře poradí“, můžeme v lineárním čase prověřit, že opravdu dostaneme jako výsledek logických operací hodnotu TRUE.

Důkaz, že každý NP-těžký problém lze převést na problém SAT je obtížnější. Zakládá se na myšlence, že pro každý Turingův stroj lze v polynomiálním čase sestavit booleovskou formuli, která je splněna tehdy a jenom tehdy, pokud se posloupnost konfigurací tohoto stroje zastaví v koncovém stavu.

Je zajímavé, že pokud bychom se v definici SAT problému omezili jen na formule s nejvýše dvěma literály v každé klausuli (disjunkci), byl by již tento tak zvaný 2-SAT problém řešitelný v polynomiálním čase. Při omezení na nejvýše 3 literály příslušný 3-SAT problém je již NP-úplný.

Pro řešení problému splnitelnosti booleovské formule neznáme rychlejší algoritmus než je algoritmus postupného prověřování všech 2^N možností pravdivostních ohodnocení TRUE / FALSE všech booleovských proměnných ve formuli. Tento algoritmus má ovšem exponenciální složitost $\theta(2^n)$.

NP-úplné jsou i následující dva známé problémy teorie grafů:

Problém úplného podgrafu: Je dán neorientovaný graf bez smyček $G = (V, E)$ a přirozené číslo k .

Rozhodovací ANO / NE problém je: Existuje v grafu G úplný podgraf $G' = (V', E')$ (takový, že každé dva uzly tohoto podgrafu jsou spojeny hranou) o k uzlech či nikoliv?

Varianta tohoto problému je tak zvaný problém maximálního úplného podgrafu (kliky) [clique]: Nalezněte maximální počet uzlů v grafu tak, aby z každého vedla hrana do každého dalšího vrcholu!

Problém nezávislé množiny vrcholů v grafu: Je dán neorientovaný graf bez smyček $G = (V, E)$ a přirozené číslo k .

Rozhodovací ANO / NE problém je: Existuje podmnožina množiny uzlů grafu o k uzlech taková, že žádný uzel z této množiny není spojen hranou grafu s žádným jiným?

Variantou je samozřejmě určit pro daný graf maximální možný počet „nezávislých“ uzlů.

Ukážeme že každý z dosud uvedených tří problémů lze transformovat v polynomiálním čase na kterýkoliv jiný ze tří. Protože o SAT problému jsme uvedli, že je NP-úplný, ukážeme tím, že i oba grafové problémy jsou NP-úplné.

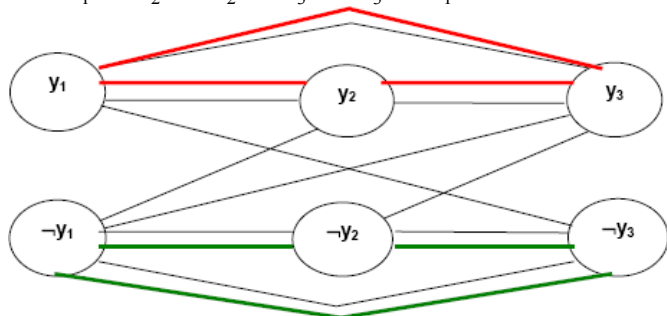
Transformace mezi oběma grafovými problémy je snadná. Stačí nahradit daný graf $G = (V, E)$ duálním grafem $G = (V, E^*)$ v kterém jsou dva uzly spojeny hranou tehdy a jenom tehdy, když v původním grafu spojeny hranou nebyly. K tomu stačí položit $E^* = V_2 \div E$, kde V_2 je množina všech dvouprvkových podmnožin množiny V . To, co byl v jednom grafu úplný podgraf (kliky) je v druhém nezávislá množina uzlů a to, co byla v jednom grafu nezávislá množina uzlů, je v druhém kliky. Transformace je zřejmě proveditelná v lineárním čase vzhledem k počtu hran a tedy v čase nejvýše kvadratickém vzhledem k počtu uzlů v grafu.

Transformace na problém SAT a naopak transformace SAT na jeden z obou grafových problémů vyžaduje trochu složitější úvahu:

Nechť F je booleovská formule délky n zapsaná v konjunktivně disjunktivní normální formě. Jako konjunkce disjunkcí literálů $\bigwedge_{j=1}^n \bigvee_{i=1}^{m_j} x_{i,j}$. Sestrojíme graf tak, že jeho uzly $v_{i,j}$ budou odpovídat i -tému výskytu logické proměnné v j -té disjunkci. Uzel $v_{i,j}$ spojíme hranou s uzlem $v_{p,q}$ tehdy a jenom tehdy, když bude $i \neq p$ a proměnné $x_{i,j}$ a $x_{p,q}$ ve formuli F nebudou opačné, to je když $x_{i,j} \neq \text{NOT } x_{p,q}$. Hrany tedy reprezentují dvojice literálů v různých klausulích, které nejsou opačné a které tedy mohou nabýt hodnotu **TRUE** současně. Formule F je splnitelná právě tehdy a jenom tehdy, když takto sestrojený graf G má úplný podgraf o k uzlech, kde k je počet konjunktí ve formuli F . Je tomu tak proto, že graf o k uzlech musí obsahovat právě jeden uzel pro každou elementární disjunkci a žádné dvě proměnné v dané formuli, které odpovídají uzlům úplného podgrafu nemohou být navzájem komplementární.

Na následujícím obrázku je nakreslen graf pro booleovskou formuli

$$F = (y_1 \vee \neg y_2) \wedge (y_2 \vee \neg y_3) \wedge (y_3 \vee \neg y_1).$$



Skutečně formule je splnitelná pro volby $y_1 = y_2 = y_3 = \text{TRUE}$ a $y_1 = y_2 = y_3 = \text{FALSE}$.

Dále uvedeme jen výčet některých dalších NP-úplných problémů. Tento výčet není zdaleka vyčerpávající. Jsou jich známy stovky.

Problémy nebudou formulovány detailně. Při detailních formulacích může mít týž problém řadu variant.

Problémy budeme formulovat pro rozhodovací variantu ANO / NE. Většinou lze vytvořit i varianty, v kterých se má vyhledat určitá hodnota.

- Problém uzlového pokrytí grafu: Podmnožina S množiny všech uzlů neorientovaného grafu se nazývá uzlové pokrytí grafu, pokud každá hrana obsahuje aspoň jeden uzel z množiny S . Existuje pro dané přirozené číslo k uzlové pokrytí o k uzlech nebo nikoliv?
- Problém hamiltonovského cyklu v grafu: Je dán orientovaný graf. Existuje v grafu uzavřená orientovaná cesta, která prochází každým uzlem právě jednou nebo neexistuje?
- Problém okružní hamiltonovské cesty: Je dán neorientovaný graf. Existuje v tomto grafu neuzavřená cesta, na které leží každý vrchol právě jednou nebo neexistuje?
- Problém obchodního cestujícího: Je dána množina míst. Vzdálenosti mezi libovolnými dvěma místy jsou dány jako celá kladná čísla. Existuje hamiltonovská cesta taková, že součet všech vzdáleností mezi sousedními místy je nejvýše roven zadanému kladnému číslu nebo neexistuje?

Tři předchozí problémy mají varianty, v kterých hledáme optimální cesty a cykly (ty pro které je součet ohodnocení hran minimální ze všech hamiltonovských cest (cyklů)).

- Problém třírozměrného sdružování: Necht' A, B, C jsou disjunktní konečné množiny, a necht' je dána množina uspořádaných trojic $S \subseteq A \times B \times C$ a q necht' je dané přirozené číslo. Existuje podmnožina množiny S o q prvcích, taková, že každé dva prvky z této množiny se liší ve všech třech souřadnicích nebo neexistuje?
- Problém spravedlivého rozdělování: Je dána množina N přirozených čísel. Je možné tuto množinu rozdělit na dvě disjunktní podmnožiny tak, aby součet čísel v obou těchto podmnožinách byl stejný nebo to možné není?
- Problém barevného čísla grafu: Přiřaďme barvy nebo čísla každému vrcholu daného grafu. Necht' k je dané přirozené číslo. Lze graf obarvit k barvami tak, aby každé dva vrcholy, které jsou spojeny hranou, byly obarveny jinou barvou nebo nelze? Pro $k \geq 3$ je tento problém NP-úplný. Pro $k = 2$ je P-těžký.
- Problém nejdelší společné podposloupnosti: Necht' u a v jsou dvě posloupnosti symbolů nad touž abecedou a k je dané přirozené číslo. Existuje posloupnost délky k , která je podposloupností posloupnosti u i posloupnosti v nebo neexistuje?
- Problém ruksaku: Necht' je dána množina dvojic čísel $S = \{(s_1, v_1), \dots, (s_N, v_N)\}$. Proměnné s označují rozměr jednotlivých předmětů, proměnné v jejich cenu. Necht' b je celkový objem, který je k dispozici a k necht' je požadovaná celková cena. Lze vybrat podmnožinu S , tak, aby součet $\sum_{j=1}^N s_j \leq b$ a $\sum_{j=1}^N s_j \geq k$ nebo nelze? Tedy tak,

aby se nám do ruksaku vešly předměty aspoň zadané úhrnné hodnoty

- Problém optimalizace programu: Je možné daný algoritmus realizovat pouze užitím k paměťových registrů nebo nelze, je-li k zadané číslo (rozsah paměti, která je k dispozici)?
- Problém plánování multiprocessorového zpracování: Je dána konečná množina T kladných čísel označujících doby které jsou zapotřebí pro realizaci daných úloh, dále je dáno m procesorů a pevný čas d , v kterém mají být všechny

úlohy dokončeny. Existuje rozdělení úloh mezi m procesorů tak, aby byl dodržen požadovaný čas nebo takové rozdělení neexistuje?

- **Problém alokace paměti:** Je dán počet datových souborů, každý daného rozsahu a určitý počet paměťových medií, každé dané kapacity. Lze všechny soubory uložit na daná media tak, aby kapacita každého z nich nebyla překročena nebo nelze?
- **Problém plánování úloh:** Je dána konečná množina uspořádaných trojic přirozených čísel $\{(t_1, d_1, p_1), \dots, (t_N, d_N, p_N)\}$. Proměnné t označují doby potřebné na řešení jednotlivých úloh, d jsou konečné časy v kterých mají být úlohy dokončeny a p jsou pokuty, které bude třeba zaplatit, pokud se úlohu v požadovaném čase ukončit nepodaří. Lze naplánovat plnění úkolů tak, aby celková pokuta nepřekročila dané číslo b nebo takový plán neexistuje?
- **Problém řešení kvadratické diofantické rovnice:** Jsou dána přirozená čísla a, b a c . Existují přirozená čísla x a y tak, že $a \cdot x^2 + b \cdot y = c$ nebo neexistují?
- **Problém celočíselného programování:** Je dána matice celých čísel A typu (m, n) a celočíselný sloupcový vektor pravých stran o m složkách b . Existuje celočíselný m -rozměrný vektor x takový že $A \cdot x \leq b$ nebo neexistuje?
- ... (je ještě mnoho dalších, prakticky důležitých)...

Mnoho dalších NP-úplných problémů souvisí s problémy rozvrhování. Například i ve většině exaktních formulací je problém nalezení školního rozvrhu tak, aby byly splněny přirozené požadavky učitelů i studentů v rámci daných prostorových kapacit NP-úplný problém.

Poněkud specifický je následující problém:

Problém prvočísel: Pro dané přirozené číslo rozhodnout, zda je prvočíslem, či nikoliv. Rozměr vstupu (zkoumané přirozené číslo) nelze považovat za jednotkový, protože velikost tohoto čísla není omezená. Musíme rozměr považovat počet bitů potřebných pro jeho vyjádření. Tedy $n = \log_2 k$ (přesněji $\log_2(k + 1) + 1$). Jednoduchý (avšak nestrukturovaný, protože obsahuje předčasný výskok z cyklu) algoritmus pro prověření prvočíselnosti je:

upper_margin := \sqrt{k}

for i := 2 to upper_margin do if (k mod i = 0) then return NO;

return YES;

V tomto algoritmu se v pesimistickém případě provádí tělo $\sqrt{k} = k^{0,5} = 2^{0,5 \cdot \log_2 k} = 2^{\theta(n)}$ -krát. Algoritmus má tedy exponenciální časovou složitost. Jsou známy „rychlejší“ algoritmy, zlepšení však není podstatné. Žádný algoritmus s polynomiální složitostí znám není. Problém prvočíselnosti však „pravděpodobně“ není NP-úplný (pochoptelně za předpokladu, že $NP_{\text{TIME}} \neq P_{\text{TIME}}$). Pokud by se snad ukázalo, že $NP_{\text{TIME}} = P_{\text{TIME}}$, byl by NP-úplný i N-těžký.

Některé alternativní postupy pro problémy u kterých neznáme polynomiální algoritmus jejich řešení

V předchozím odstavci byla uvedena řada problémů, pro které není znám žádný algoritmus polynomiální časové složitosti. Tedy žádný algoritmus, který by byl pro větší rozměr vstupních dat reálně použitelný. Patrně není naděje takový algoritmus pro NP-úplné problémy nalézt.

Protože řada z uvedených problémů má značný praktický význam, je důležité umět nabídnout nějaké „náhradní“ řešení, jak v dané situaci postupovat. Takových postupů je řada. Zmíníme se o některých.

Žádný z těchto postupů však není univerzální. Každý má své vady a omezení. Někdy je vhodné i nutné je zkoušet alternativně.

V principu jsou možné dvě cesty jak problém časové složitosti obejít:

1. Nahradit daný problém jiným problémem, který v polynomiálním čase řešit umíme a jehož řešení „není příliš odlišné“ od řešení původního problému, které nás zajímá, nebo se od něj příliš neliší „v převážné většině případů“. Tedy aproximovat řešení časově nezvládnutelného problému řešením problémů, které zvládnout umíme a doufat, že získané řešení bude hledanému „blízko“.
2. Užít algoritmus pro řešení původního problému, jehož pesimistická časová složitost sice není polynomiální, nalézt však takovou jeho modifikaci, při které k časově neúnosně dlouhému výpočtu dochází spíše výjimečně a ve „většině“ případů je potřebná doba přijatelná. Doufat, že k výjimečné situaci, pokud jde o délku výpočtu, nedojde a výsledek dostaneme v přijatelném čase.

V obou případech používáme tak zvaná **heuristická pravidla** (heuristiky). To jsou pravidla, která sice neoplatí obecně, nezaručují úspěch vždy, ale pouze ve „většině případů“.

Zmíníme se **pouze rámcově** o dvou skupinách algoritmů, které se používají. Tento výčet však není zdaleka vyčerpávající.

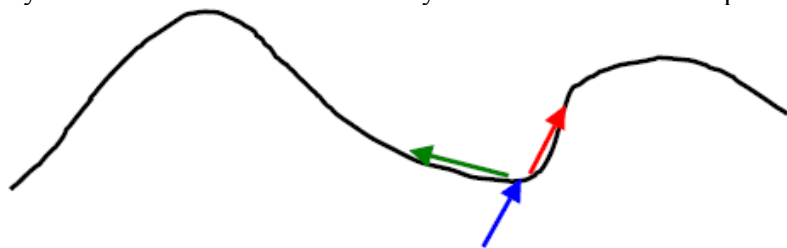
Algoritmy větví a hranic (prořezávání stromu)

Typický postup představuje tak zvaný **princip větví a hranic [branch and bound idea]**, někdy též nazývaný princip **prořezávání stromu**. Princip spočívá v tom, že v každé situaci, kdy musíme vyšetřit více možností se věnujeme pouze těm, které jsou z nějakého důvodu perspektivní. Ty, které se jeví jako málo nadějně pro nalezení řešení vynecháme (prořezeme).

Typická aplikace metody větví a hranic prvního typu (kdy si nemůžeme být jisti zda výsledek získaný v přijatelném čase je blízko hledanému, je tomu však tak „obvykle“) bývá užít v programech pro hru šachy. Zde program nezkoumá všechny možnosti vývoje pozic na šachovnici mnoho tahů dopředu, ale každou pozici hodnotí. Pokud se zdá neperspektivní (například proto, že došlo k velkým ztrátám materiálu), dále ji nerozvíjí a zvažuje jen pozice „nadějně“. Tato strategie je

úspěšná většinou. Ne však vždy. Šachisté vědí, že některé „oběti materiálu“ mohou být výhodné a vést třeba k matu soupeře o několik tahů později. Tak se může stát že optimální cesta bude odříznuta předčasně.

Podobný princip, kombinovaný s principem hladových algoritmů, vede k tak zvanému **gradientním principu**, zvaném někdy i **princip horolezce** [mountain climbing principle]. V řadě optimalizačních algoritmů je vhodné volit metodu postupného přibližování k optimu tak, že přiblížení volíme „tím směrem“, kde se sledovaná hodnota zlepšuje nejrychleji. Je to jako když horolezec chce dosáhnout vrcholu hory tak, že leze tím směrem, kterým je svah nejpráhřejší. V řadě případů to k cíli vede. Ne však vždy. Může se snadno stát, že horolezec, který si dal za cíl zdolat nejvyšší vrchol pohoří vyleze do sedla mezi dvěma vrcholy a na základě zvoleného principu vyleze na ten nižší z obou.



Tak i gradientní algoritmy často naleznou jen lokální extrém, ne hledaný globální extrém.

Příkladem užití principu větví a hranic druhého typu (kdy je sice zaručeno nalezení hledaného řešení původního problému, ne však čas v kterém bude řešení získáno) je následující varianta algoritmu pro problém obchodního cestujícího bez návratu do výchozího uzlu:

Je dán neorientovaný graf $G = (V, E)$. Hamiltonovská cesta je neuzavřená cesta, která každým vrcholem grafu prochází právě jednou. Graf je hranově ohodnocený (například cenami transportu z jednoho uzlu do druhého). Problém je nalézt hamiltonovskou cestu s minimálním součtem ohodnocení hran.

Již sám problém nalezení hamiltonovské cesty je NP-úplný. Tím spíše náš optimalizační problém. Uvažujme následující (neformálně naznačený algoritmus):

1. Najdeme minimální kostru G (například Borůvkovým algoritmem). Je-li to cesta, je to minimální hamiltonovská cesta, protože hamiltonovská cesta je kostrou grafu.
2. Není-li K cesta, je délka minimální hamiltonovské cesty větší nebo rovna součtu ohodnocení hran v K a K přitom obsahuje aspoň jeden uzel stupně aspoň 3, kde se kostra „větví“ (protože není cestou). Necht'v je takový uzel a uzly $w_1, w_2, w_3, \dots, w_n$ jsou jeho sousedi.
3. Sestrojíme grafy $G_1, G_2, G_3, \dots, G_n$ z původního grafu G po řadě vynecháním hran $(v, w_1), (v, w_2), (v, w_3), \dots, (v, w_n)$, a pro každý z těchto grafů nalezneme po řadě jejich minimální kostru $K_1, K_2, K_3, \dots, K_n$.
4. Pokud je některá z těchto koster $K_1, K_2, K_3, \dots, K_n$ cestou, máme minimální dosud nalezenou hamiltonovskou cestu.
5. Jestliže žádná z koster $K_1, K_2, K_3, \dots, K_n$ cestou není, vybereme tu, která má minimální součet ohodnocení a opakujeme bod 3 algoritmu (vyšetřujeme další větev).
6. Nalezneme-li takto nějakou hamiltonovskou cestu, mohou nastat následující případy:
 - o Má-li tato hamiltonovská cesta délku menší než dosud nalezená nejkratší hamiltonovská cesta, zapamatujeme si ji jako dosud nejkratší nalezenou hamiltonovskou cestu
 - o Jde-li o cestu delší než je dosud nalezená nejkratší hamiltonovská cesta, vyřadíme ji
 - o Jde-li o kostru, která není cestou, ale součet ohodnocení hran této kostry je nižší než délka dosud nalezené hamiltonovské cesty, opakujeme bod 3 (vyšetřujeme další větev).
 - o Jde-li o kostru, která není cestou a součet ohodnocení hran této kostry je větší než délka dosud nalezené nejkratší cesty, vyřadíme ji, protože nemůže již vést k nalezení kratší hamiltonovské cesty.
7. Pokračujeme tak dlouho, dokud všechny větve algoritmu nejsou ukončeny.

Tento algoritmus je v převážné většině případů mnohem rychlejší než je řešení hrubou silou, tedy testováním všech $n!$ možných cest. Jeho pesimistická časová složitost však polynomiální není. Existují grafy, u kterých tento algoritmus testuje také všechny permutace. Jsou však poměrně „vzácné“.

Genetické algoritmy

V poslední době je velmi populární následující třída tak zvaných genetických algoritmů.

Genetické algoritmy jsou založeny na Darwinově evolučním principu vývoje populace živých organismů přirozeným výběrem. V něm se genetická informace nové generace vytváří kombinováním (křížením) genetické informace úspěšných jedinců generace předchozí s přípustěním určitých samovolně vznikajících mutací.

V informatické formulaci jde o snahu maximalizovat danou funkci zvanou **funkce úspěšnosti** [fitness function], na množině konečných řetězců (slov) nad danou abecedou, která charakterizují příslušné entity (individa v populaci). V biologické analogii těmto řetězcům odpovídají řetězce chromozomů v buňce, tvořící genetickou informaci definující individuum.

Pro dva řetězce je binární operace **křížení** [crossing] definována jako vytvoření řetězce převzetím některých částí z jednoho a jiných z druhého kříženého řetězce. Operace **mutace** [mutation] je definována jako náhodná náhrada některého symbolu v řetězci symbolem (chromozomem) dané abecedy.

Genetické algoritmy patří do první třídy klasifikace uvedené na začátku tohoto odstavce. **Nalezení řešení původního problému (maxima funkce úspěšnosti) genetické algoritmy negarantují.** „Často“ se k němu však přibližují. Jde spíše o „experimentální přístup“.

Hrubé schéma genetických algoritmů je následující:

Instalace: Zřídíme náhodně počáteční populaci slov – individuí POP[0] o rozměru N a určíme hodnoty funkce úspěšnosti f pro všechny prvky této populace. Nastavíme ukazatel pořadového čísla populace j na hodnotu j := 0.

Hlavní cyklus: Dokud nebudeme spokojeni s maximální hodnotou funkce f v dané populaci, opakujeme tyto kroky:

- 1.1 Výběr: Vyber některá individua z dané populace POP[j] s „dobrou“ hodnotou funkce úspěšnosti a převed' je přímo do následující populace POP[j + 1].
- 1.2 Křížení: Pro všechny páry „dobrých rodičů“ provedeme křížení a „děti“ zařadíme do nové populace POP[j + 1].
- 1.3 Mutace: Pro některá vybraná individua dosud zařazená do POP[j + 1] provedeme mutaci a mutované prvky zařadíme do POP[j + 1].
- 1.4 Hodnocení: Pro všechna individua v POP[j + 1] vypočteme hodnotu funkce úspěšnosti f.
- 1.5 Redukce: V závislosti na hodnotě funkce úspěšnosti zredukujeme populaci POP[j + 1] na rozměr N (neúspěšní jedinci se dále nekříží).

Popsané schéma není deterministické. Má mnoho stupňů volnosti. Například vybírat, které hodnoty funkce úspěšnosti budeme považovat za „dobré“? Jak budeme vybírat rodiče pro křížení? Jedna z užívaných možností je třeba výběr rodičů,

každého s pravděpodobností $f(x) / \sum_{i=1}^N f(t_i)$, kde i probíhá celou populací. Užívají se i jiné strategie výběru. Stejná

volnost je pro redukci populace. Pokud budeme uvažovat absolutní preference, riskujeme blíženi pouze k lokálnímu extrému funkce úspěšnosti. Pokud dáme více na náhodu, nemusí postup konvergovat, populace může degenerovat. Další volnost je v míře povolených mutací. Jde tedy spíše o experimenty než o matematicky rigorózní postupy.

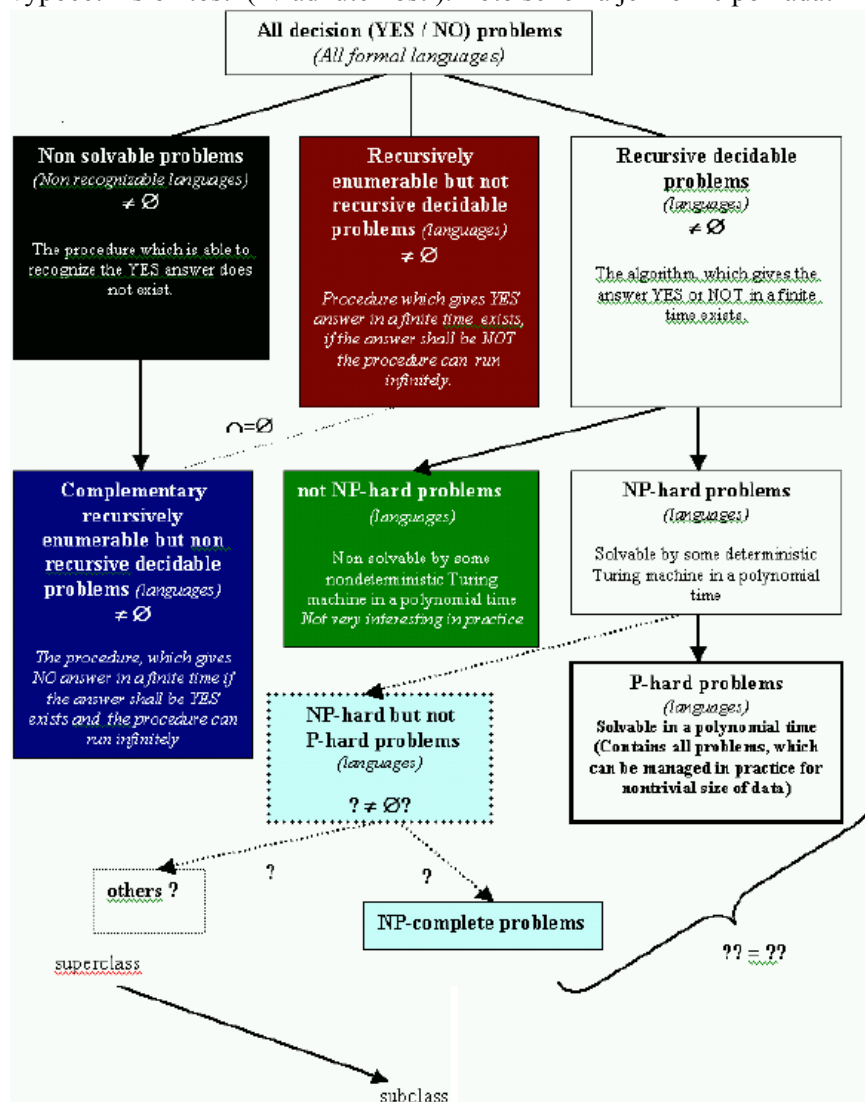
Velkým problémem je i výběr chromozomů pro charakterizaci individuí.

Genetické algoritmy jsou často v praxi úspěšné. Nelze však garantovat ani správnost výsledku, ani jejich rychlost.

Další velmi důležitý přístup k řešení nezládnutelných problémů představují **neuronové sítě**.

Klasifikace problémů

Na závěr uvedeme schéma představující taxonomii problémů z hlediska jejich rozhodnutelnosti, rozpoznatelnosti a výpočetní složitosti (zvládnutelnosti). Toto schéma je možné pokládat i za taxonomii formálních jazyků.



11. SLOŽITOST VYBRANÝCH NĚKTERÝCH ALGORITMŮ

V celém tomto tématu budeme předpokládat, že pracujeme pouze s čísly omezeného rozsahu. Například s celými čísly, které lze reprezentovat typem **int** nebo **longint**. Čísla převyšující tento rozsah nebudeme připouštět. Následkem toho můžeme počet kroků během výpočtu považovat za lineární funkci počtu operací s takovými čísly.

Nejprve uvedeme pro ilustraci příklad, který ukáže, že týž problém lze řešit různými algoritmy, jejichž časová složitost se může značně lišit.

Čtyři různá řešení téhož problému

Mějme časovou řadu reprezentovanou polem celých čísel A_1, A_2, \dots, A_N , která mohou být kladná i záporná. Úkol je nalézt v této řadě úsek s maximálním součtem $\sum_{k=1}^j A_k$. Například v případě, že $N = 10$ a pole má složky $-1, 11, -5, 10, 3, -12, 0, 5, 6, -8$, bude řešením součet 19, získaný sčítáním od A_2 do A_5 . Pro jednoduchost doplníme zadání tak, že pokud v poli není žádné kladné číslo, prohlásíme za hledané maximum nulu, jako součet přes prázdný úsek pole.

Praktickou interpretaci si můžeme představit třeba jako zjištění období nejvyššího růstu nějakého atributu, třeba teploty či směnného kurzu, jsou-li hodnoty v poli rozdíly hodnot atributu ze dne na den.

Možná řešení popíšeme algoritmy v jazyce C. V dalších odstavcích budeme algoritmy již pouze naznačovat pseudokódem pascalského typu.

Ukážeme, že časová složitost různých algoritmů se může drasticky lišit.

První algoritmus je primitivní a dosti „hloupý“. Zakládá se na výpočtu součtů pro všechny možné začátky a všechny možné konce vybraného úseku a vybrání maxima pro každou volbu dvojice (počátek, konec).

První algoritmus je primitivní a dosti „hloupý“. Zakládá se na výpočtu součtů pro všechny možné začátky a všechny možné konce vybraného úseku a vybrání maxima pro každou volbu dvojice (počátek, konec).

Algorithm #1:

```
Int MaxSubsequenceSum( const int A[ ], int N )
{
    int ThisSum, MaxSum, i, j, k;
    /*1*/ MaxSum = 0;
    /*2*/ for(i = 0; i < N; i++ )
    /*3*/ for ( j = i; j < N; j++ )
    {
        /*4*/ ThisSum = 0;
        /*5*/ for( k=i; k<= j; k++ )
        /*6*/ ThisSum += A[ k ];
        /*7*/ if ( ThisSum > MaxSum )
        /*8*/ MaxSum = ThisSum
    }
    /*9*/ return MaxSum;
}
```

V algoritmu je příkaz na řádce 6 vnořen do tří vnořených cyklů. Časová složitost tohoto příkazu je $\Theta(1)$. Cyklus na řádce 2 má N kroků, cyklus na řádce 3 $N - 1$ kroků a cyklus na řádce 5 má $j - i + 1$ kroků. Rozměr všech tří cyklů je nutné odhadnout číslem N . Celkový počet provedení příkazů na řádce 6 bude tedy $\mathbf{O}(1 \cdot N \cdot N \cdot N) = \mathbf{O}(N^3)$. Test na řádce 7 a příkazy na řádkách 4 a 8 se provedou jen $\mathbf{O}(N^2)$ -krát, příkaz na řádce 1 pouze jednou.

Celková časová složitost algoritmu #1 tedy bude $\mathbf{O}(N^3)$. Lze ukázat, že dokonce $\Theta(N^3)$.

Přesná analýza se získá analýzou součtu $S = \sum_{i=0}^{N-1} \sum_{j=i}^{N-1} \sum_{k=i}^j 1$, který udává kolikrát se provádí příkaz 6. Užitím vzorce pro součet kvadratické řady dostaneme

$$S = \sum_{i=0}^{N-1} \sum_{j=1}^{N-1} \sum_{k=i}^j 1 = \sum_{i=0}^{N-1} ((N-i+1)(N-i))/2 = \sum_{i=0}^N ((N-i+1)(N-i))/2 = (N^3 + 3N^2 + 2N)/2$$

Tedy celková složitost je $\Theta(N^3)$.

Je jasné, že užijeme-li vzorec $\sum_{k=i}^j A_k = \sum_{k=i}^{j-1} A_k + A_j$, můžeme se snadno zbavit nejvnitřnějšího cyklu. Získáme tak následující algoritmus:

Algorithm #2:

```
int
MaxSubsequenceSum( const int A[ ], int N )
{
    int ThisSum, MaxSum, i, j;
    /*1*/ MaxSum = 0;
    /*2*/ for( i=0; i < N; i++ )
    {
        /*3*/ ThisSum = 0;
```



```

/*4*/ for( j = I; j < N; j++ )
{
/*5*/ ThisSum += A[ j ];
/*6*/ if( ThisSum > MaxSum )
/*7*/ MaxSum = ThisSum;
}
}
/*8*/ return MaxSum;
}

```

Analogickou úvahou jako u algoritmu #1 se dá ukázat, že příkazy na řádcích 5, 6 a 7 se provádějí nejvýše N^2 -krát. **Složitost algoritmu #2 je tedy $\Theta(N^2)$.**

Z hlediska složitosti návrhu je náročnější následující algoritmus, založený na principu typu „rozděl a panuj“ [„divide at impera“].

Algorithm #3:

```

static int
MaxSubSum( const int A[ ], int Left, int Right )
{
int MaxLeftSum; MaxRightSum; MaxLeftBorderSum;
int MaxRightBorderSum; LeftBorderSum; RightBorderSum;
int Center; I;
/*1*/ if(Left == Right ) /* Base Case*/
/*2*/ if A[ Left ] > 0
/*3*/ return A [ Left ];
else
/*4*/ return 0;
/*5*/ Center = ( Left + Right ) / 2;
/*6*/ MaxLeftSum = MaxSubSum ( A, Left, Center );
/*7*/ MaxRightSum = MaxSubSum ( A, Center + 1, Right);
/*8*/ MaxLeftBorderSum = 0; LeftBorderSum = 0;
/*9*/ for i = Center; i >= Left; i-- )
{
/*10*/ LeftBorderSum += A[ i ];
/*11*/ if LeftBorderSum > MaxLeftBorderSum )
/*12*/ MaxLeftBorderSum = LeftBorderSum;
}
/*13*/ MaxRightBorderSum = 0; RightBorderSum = 0;
/*14*/ for i = Center + 1; i <= Right; i++ )
{
/*15*/ RightBorderSum += A[ i ];
/*16*/ if RightBorderSum > MaxRightBorderSum )
/*17*/ MaxRightBorderSum = RightBorderSum;
}
/*18*/ return Max3(MaxLeftSum, MaxRightSum,
MaxLeftBorderSum + MaxRightBorderSum );
}
int MaxSubSequenceSum (const int A[ ], int N )
{
return MaxSubSum ( A, 0, N - 1 );
}

```

Idea algoritmu #3 (jako u každého postupu typu rozděl a panuj) spočívá v rozdělení problému analýzy časové řady na dva (nebo více) „skoro stejně rozsáhlých“ úloh, které se řeší rekursivním postupem (dalším dělením až dospějeme k triviálnímu problému) a následným „dáním dohromady“ řešení těchto „menších“ úloh. Fáze „rozděl“ představuje dělení problému na části, fáze „panuj“ slučování dílčích řešení.

V našem případě rozdělíme-li řadu na skoro stejně velké (stejně velké, je-li rozměr pole sudé číslo, lišící se délkou o 1, je-li rozměr pole liché číslo) části nazvané „Levá polovina“ a „Pravá polovina“, může být maximální úsek buď:

- Celý v Levé polovině nebo
 - Celý v Pravé polovině nebo
 - Uprostřed mezi oběma polovinami, realizovaný úsekem, který v levé polovině začíná a v pravé polovině končí.
- Prvé dva případy lze řešit rekursivně. Rozdělením Levé poloviny na Levou-levou a Levou-pravou a Pravé poloviny na Pravou-levou a Pravou-pravou.

Třetí případ je třeba ošetřit zvlášť. Provede se to tak, že nalezneme úsek s největším součtem v levé polovině, který končí posledním prvkem této poloviny a úsek s největším součtem v Pravé polovině, který prvním prvkem této poloviny začíná a následným sečtením těchto dvou součtů.

Řádky 1 – 4 obsluhují triviální případ. Je-li $\text{Left} = \text{Right}$, má pole jediný prvek. Je-li nezáporný, je roven hledanému maximálnímu součtu. Je-li záporný, je hledaný součet 0, protože jde o součet přes prázdný úsek.

Řádky 6 a 7 provádějí rekurzi. Řádky 8 – 12 a 13 – 17 počítají součty polí, které se dotýkají prostředního rozhraní.

Analýza časové složitosti algoritmu #3 je následující: Označme $T(N)$ počet potřebných kroků algoritmu. Je-li $N = 1$ je třeba provést pouze konstantní počet kroků na řádcích 1 – 4. Tedy $T(1) = \Theta(1)$. Pro $N > 1$ dojde ke dvěma rekurzivním voláním cyklů mezi řádky 9 to 17 a provedení některých dalších operací (řádky 5 a 18). Složitost uvnitř cyklu je $\Theta(N)$. Složitost kódu na řádcích 1 – 5, 8, 13, 18 je konstantní a lze ji vzhledem k složitosti $\Theta(N)$ zanedbat. Zbývá složitost provedení řádků 6 a 7. Jde o dva problémy, každý složitosti $N / 2$ (je-li N sudé).

Pro celkový odhad složitosti tedy dostáváme rekurentní vzorec: $T(1) = 1$, $T(N) = 2 \cdot T(N/2) + O(N)$.

Odtud plyne, že je-li $N = 2^k$, je $T(N) = N \cdot (k + 1) = N \cdot \log N + N = \Theta(N \cdot \log N)$.

Tato analýza je zcela korektní, je-li N přirozená mocnina dvou. Odhad platí ale i obecně. To lze nahlédnout například tak, že doplníme pole nulami do délky M tvaru 2^k . Dostaneme

$T(N) \leq \Theta(M \cdot \log M) = \Theta(2N \cdot \log 2N) = \Theta(N \cdot \log N)$, ale přitom

$T(N) \geq \Theta(M/2 \cdot \log (M/2)) = \Theta(N \cdot \log N)$.

Algoritmus #4 je pro velká N rychlejší než algoritmy #1 a #2, nicméně není optimální. Existuje algoritmus řešící náš problém pouze užitím jediného průchodu daty v poli, tedy algoritmus časové složitosti $\Theta(N)$.

Tento optimální algoritmus lze odvodit na základě následující úvahy o vstupních datech:

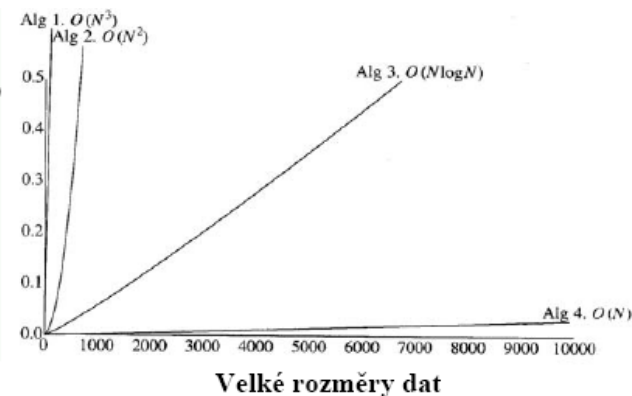
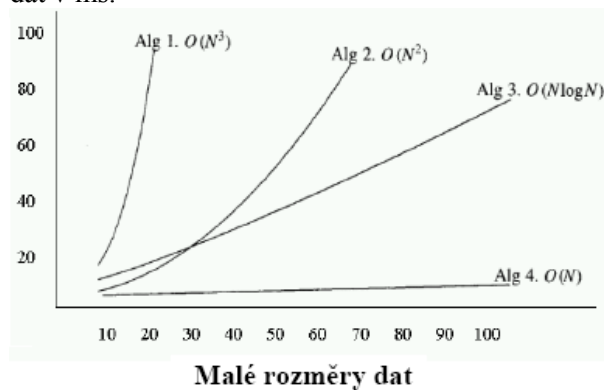
Maximální úsek musí začínat kladným číslem (jinak bychom vynecháním tohoto začátku součet zvětšili). Dále musí tento vybraný úsek končit dříve než vyhodnocovaný součet, shromažďovaný v proměnné **ThisSum** nabude s záporné hodnoty. V tom případě musíme začít s novým pokusem zjistit možné nové maximum.

Algorithm #4:

```
int
MaxSubsequenceSum( const int A[ ], int N )
{
    int ThisSum, MaxSum, j ;
    /*1*/ ThisSum = MaxSum = 0;
    /*2*/ for( j = 0; j < N, j++ )
    {
        /*3*/ ThisSum += A[ j ];
        /*4*/ if( ThisSum > MaxSum )
        /*5*/ MaxSum = ThisSum;
        /*6*/ else if( ThisSum < 0 )
        /*7*/ ThisSum = 0;
    }
    /*8*/ return MaxSum;
}
```

Algoritmus má zřejmě časovou složitost $\Theta(N)$.

Následující grafy znázorňují časovou náročnost analyzovaných algoritmů pro malé rozsahy dat v μs a pro velké rozsahy dat v ms .



Princip „rozděl a panuj“ a rekurze

Algoritmus #3 in 2.1 byl příkladem algoritmu využívajícího postup, který se obvykle nazývá „rozděl a panuj“ [divide-and-conquer], latinsky “divide at impera”. Algoritmy této třídy mají často „dobrou časovou složitost. Princip spočívá v následujících krocích:

1. Původní „velký“ problém se rozdělí na (obvykle dva) „menší“ problémy přibližně stejného rozsahu, stejného typu
2. Podproblémy se řeší samostatně, pokud nejsou triviální, opět rozkladem podle bodu 1.
3. Výsledné řešení „velkého“ problému se získá vhodným sloučením řešení „malých“ problémů.

Řešení dílčích problémů stejného typu jako problém původní se získá **rekurzivním voláním příslušné procedury**. Toto volání musí být pouze alternativní částí procedury, která se uplatní pouze je-li problém dosud netriviální. Musí být zabezpečena konečnost rekurze, tedy to, že po konečném počtu kroků již k dalšímu rekurzivnímu volání nedojde a

procedura se ukončí. Rekurzivní procedura tedy musí mít dvě větve. V jedné se tato procedura volá znova, v druhé již k volání nedojde. Prvá větev se užije pouze konečně-krát.

Platí následující hrubý odhad pro časovou složitost:

Pokud se podaří problém rozložit vždy na „téměř“ stejně rozměrné problémy algoritmem složitosti $O(1)$, násobí se výsledná složitost faktorem $O(\log N)$.

Pokud je rozklad algoritmem složitosti $O(1)$, typu $(N - 1, 1)$, násobí se výsledná složitost faktorem $O(N)$.

Přesně popisuje situaci následující věta, kterou uvedeme bez důkazu:

Věta: Necht' $a \geq 0$, $b > 1$ jsou reálné konstanty a f a T jsou funkce zobrazující množinu N všech přirozených čísel do N . Nechť pro funkci T platí rekurentní vztah: $T(n) = a \cdot T(n/b) + f(n)$.

Potom:

If $f(n) = \Theta(n^c)$ and $c < \log_b a$, then $T(n) = \Theta(n^{\log_b a})$

If $f(n) = \Theta(n^{\log_b a})$, then $\Theta(n^{\log_b a} \cdot \log n)$

If $f(n) = \Theta(n^c)$ and $c > \log_b a$, then $T(n) = \Theta(n^c)$

Větu můžeme použít pro analýzu složitosti algoritmů třídy „rozděl a panuj“ tak, že b je počet dílčích úloh na které úlohu rozdělíme a a a f bude rovno složitosti práce spojené s kompletací dílčích řešení na řešení celku.

U algoritmu #3 pro maximální součet úseku posloupnosti je

$a = 2$, $b = 2$, $f(n) = \Theta(n) = \Theta(n)$. Tedy $T(n) = \Theta(n \cdot \log n)$. $2 \log 2$

Rekurze a princip „rozděl a panuj“ není ovšem lék na cokoli. Někdy nelze problém rozdělit na „skoro stejné“ části.

Například u známého algoritmu pro výpočet faktoriálu:

```
long int
Factorial ( int N )
{
    if ( N <= 1 )
        return 1
    else
        return N * Factorial( N - 1 );
}
```

Složitost je $\Theta(N)$, protože krok rekurze snižuje N vždy pouze o 1. Další problém spočívá v tom, že velmi brzo překročíme omezení i pro typ **long int**.

Analogický algoritmus pro prvky Fibonacciho posloupnosti definované vztahy:

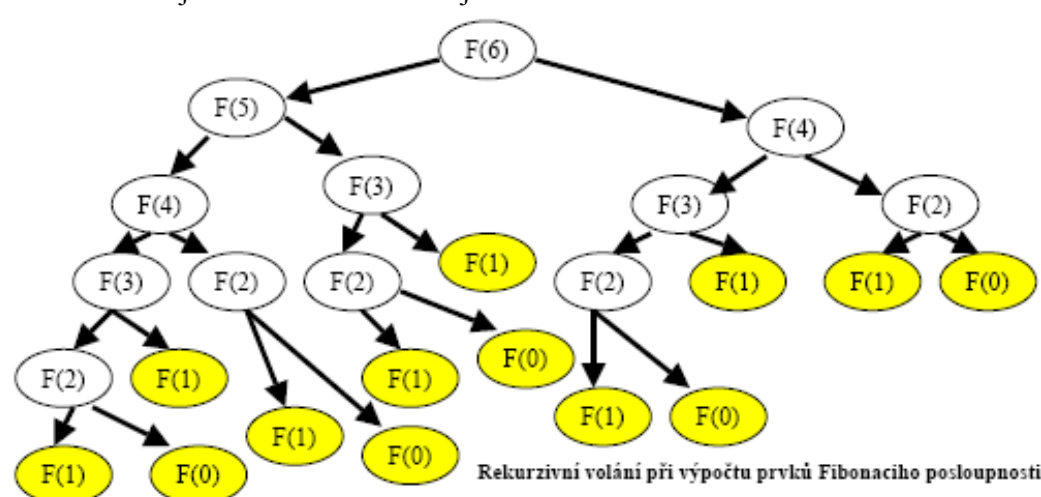
$FIB(0) = FIB(1) = 1$; $FIB(N) = FIB(N - 1) + FIB(N - 2)$ pro $N = 2, 3, \dots$

```
long int
FIB ( int N )
{
    /*1*/ if ( N <= 1 )
    /*2*/ return 1
    else
    /*3*/ return FIB( N - 1 ) + FIB( N - 2 );
}
```

, vypadá na první pohled elegantně. Ve skutečnosti je však velmi nevýhodný. Provedme jeho analýzu:

Pro $N = 0$ a pro $N = 1$ je potřebný čas konstantní. Jde jen o test na řádce 1 a návrat. Pro $n \geq 2$ se potřebný čas skládá z konstantního času pro provedení řádku 1 a času pro řádek 3. Na řádce 3 se sčítají výsledky dvou opětovných volání funkce **FIB**. Prvé volání potřebuje $T(N - 1)$ operací, druhé $T(N - 2)$ operací. Celkem tedy je třeba $T(N - 1) + T(N - 2) + 2$ operací. Je tedy $T(N) \geq FIB(N)$. Lze ukázat, že pro $N > 4$ je $FIB(N) \geq (3/2)^N$. Přesný důkaz by bylo možné provést matematickou indukcí. Časová složitost je tedy exponenciální. Například pro $N > 30$ již není reálné výpočet ukončit v rozumném čase.

Průběh volání je znázorněn na následujícím obrázku



Přitom není obtížné navrhnout algoritmus, který hodnoty posloupnosti počítá v jediném cyklu a ukládá je do dvou pomocných proměnných, v kterých jsou vždy předchozí hodnoty $FIB(N - 1)$ a $FIB(N - 2)$. Realizujte jej!

Tento algoritmus bude mít přijatelnou časovou složitost $\Theta(N)$.

Algoritmy vyhledávání [search]

Formulace problému vyhledávání:

Nechť $A_0, A_1 \dots A_{n-1}$ jsou jednoduché položky v paměti, uložené jako pole nebo proud (například lineární spojový seznam nebo zásobník). Obě tyto struktury budeme nazývat společným termínem „seznam“.

Nechť X je daná položka. Problém je určit i tak, že $A_i = X$, případně indikovat, že takové i neexistuje (například $i = -1$). V obecnějším případě může jít o složené položky obsahující klíč a o hledání položky se zadanou hodnotou klíče. Pro jednoduchost předpokládejme, že postačí vyhledat jakýkoliv výskyt položky se zadanou vlastností, pokud se vyskytuje v prohledávaných datech takových položek více než jedna.

V případě, že **nemáme k dispozici žádné doplňující informace o uspořádání** zadaného pole nebo proudu položek (například, že položky jsou uspořádané podle klíče, který slouží k vyhledávání), je zřejmé, že pole či proud musíme postupně prohledávat, dokud na hledaný klíč nenarazíme. Je zřejmé, že časová složitost takového algoritmu v pesimistické i průměrné variantě bude - $\Theta(n)$.

Pokud pole bude uspořádáno (utříděno) podle hodnoty klíče, můžeme algoritmus urychlit na základě principu „rozděl a panuj“. Hledanou hodnotu porovnáme s „prostředním“ prvkem pole. Nastává-li rovnost, je prvek nalezen. Pokud je hledaná hodnota menší než „střední“, použijeme stejnou strategii pro polovinu „menších“ prvků, je-li menší, hledáme ve střední polovině „větších“. Tento postup opakujeme rekurzivně, dokud na prvek nenarazíme nebo dokud nezjistíme, že hledaná hodnota je „ostře mezi“ dvěma sousedními prvky v poli. V tom případě prvek s hledanou hodnotou indexu v poli není. Pro možnost užít tento postup je **podstatné, že k prvkům pole je náhodný přístup** [random access] prostřednictvím indexu. Tento algoritmus se nazývá binární vyhledávání [binary search].

Nerekurzivní varianta algoritmu binárního vyhledávání je:

```
int
BinarySearch (const ElementType A[ ], ElementType X, int N )
{
    int Low, Middle , High;
    Low = 0; High = N - 1;
    While ( Low <= High )
    {
        Middle = ( Low + High ) / 2;
        if ( A[ Mid ] < X
        Low = Mid + 1;
        else
        if ( A[Mid] > X)
        High = Mid - 1;
        else return Mid /* Found */
    }
    return NotFound /* NotFound is defined as -1 */
}
```

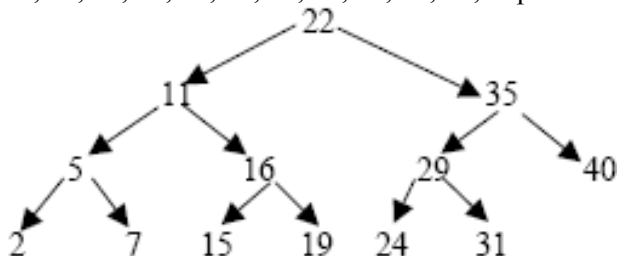
Rekurzivní variantu sestavte sami v rámci procvičování.

Z názoru je patrné, že časová složitost rekurzivní i nerekurzivní varianty $\Theta(\log n)$. Přesný důkaz tohoto a dalších obdobných tvrzení by bylo možné provést užitím obecné věty z předchozího odstavce.

Vyšetřujeme nyní úlohu **zařadit nový prvek do uspořádaného seznamu** tak, aby uspořádání bylo zachováno. Užijeme opět princip binárního vyhledávání:

Porovnáme nový prvek X s „prostředním“ prvkem seznamu A_{middle} . Je-li $X > A_{\text{middle}}$, pokusíme se prvek zařadit do „horní poloviny“ pole, v opačném případě do „dolní poloviny“ pole. Časová složitost algoritmu až po rozhodnutí kam prvek zařadit je také $\Theta(\log n)$, nyní však nastane problém vlastního zařazení do pole. Pokud jde o statickou strukturu, musíme pro X „uvolnit místo“, postupným posunutím prvků počínaje posledním až po nejbližší větší než je zařazovaný o jedno místo vlevo. To si však vyžádá $\Theta(n)$ operací. V případě dynamické struktury (jako například lineární spojový seznam) stačí provést pouze omezený konečný počet změn ukazatelů. Vlastní zařazení má tedy složitost $\Theta(1)$. Zde však vyvstane jiný problém. K prvkům lineárního spojového seznamu není přímý přístup.

Kompromisem, který umožní užít jak výhod přímého přístupu tak výhod rychlé reorganizace struktury při doplňování prvků je **užití stromové struktury pro organizaci seznamu**. Pro reprezentaci stromu užijeme pořadí **inorder** (levý podstrom obsahuje prvky s nižší hodnotou klíče, pravý s vyšší hodnotou klíče). Například pro seznamu s klíči 2, 5, 7, 11, 15, 16, 19, 22, 24, 26, 28, 29, 31, 35, 40, odpovídá binární „inorder“ strom:



Bude sice vytvoření struktury složitější a prostorově náročnější (budeme potřebovat u každé položky 3 ukazatele: na levého syna, na pravého syna a zpět na otce), vyhledání místa pro nový prvek však bude mít složitost $\Theta(\log n)$ a reorganizace struktury pro zařazení nového prvku též složitost $\Theta(\log n)$. celková složitost zařazení nového prvku do uspořádaného seznamu realizovaného tímto binárním stromem bude tedy pouze $\Theta(\log n)$.

Algoritmy řazení („třídění“) [sort]

Úloha je najít tak zvanou třídící permutaci $(\pi(1), \pi(2), \dots, \pi(N))$ daného seznamu (pole nebo proudu) A_1, A_2, \dots, A_N tak, aby posloupnost $A_{\pi(1)}, A_{\pi(2)}, \dots, A_{\pi(N)}$ byla uspořádaná to je, aby $A_{\pi(1)} \leq A_{\pi(2)} \leq \dots \leq A_{\pi(N)}$.

Analogicky můžeme požadovat řazení v opačném pořadí. Od největšího klíče k nejmenšímu. Nelézt permutaci $\pi(1), \pi(2), \dots, \pi(N)$, čísel $1 \dots, N$ tak, že $A_{\pi(1)} \geq A_{\pi(2)} \geq \dots \geq A_{\pi(N)}$. Dál se omezíme na prvou z obou variant. Druhá je analogická.

Veškeré algoritmy řazení lze rozdělit do dvou tříd:

- Algoritmy adresního řazení.
- Algoritmy asociativního řazení.

Algoritmy adresního řazení jsou založeny na myšlence definovat zobrazení mezi hodnotou klíče a indexem místa, kterou bude mít záznam v seřazeném souboru. Jejich časová složitost je nízká. Pokud bychom mohli o umístění rozhodnout v jediném průchodu, byla by pouze $\Theta(N)$. Tyto algoritmy však vyžadují znát informace o rozložení hodnot klíče v seznamu, které obvykle k dispozici nejsou. Pokud je nemáme, byl by takový algoritmus neúnosně prostorově složitý. Proto se užívají jen zřídka.

Algoritmy asociativního řazení jsou založeny na postupném porovnávání hodnot klíče. Jimi se budeme zabývat podrobněji.

Omezíme se dále pouze na tak zvané **vnitřní řazení**. Budeme předpokládat, že data jsou všechna současně umístěna v jedné úrovni paměti s přímým přístupem. Pro velké objemy dat nemusí být snadné tento požadavek splnit. Pokud jsou data v pamětech různé úrovně (též na vnějších pamětech), je třeba užívat speciálních algoritmů, založených zpravidla na vytváření seřazených úseků ve formě sekvenčních souborů (monotonií) a jejich následném slévání [merge]. Pro vytváření monotonií se užívají algoritmy vnitřního řazení.

Dále se budeme věnovat blíže asociativním algoritmům pro vnitřní řazení.

Algoritmus vycházející přímo z definice úlohy, založený na postupném generování všech permutací a testování, zda po provedení permutace je seznam seřazen lze realizovat snadno. Procedura **next_permutation(A)** může být navržena tak, že vybere kterýkoliv prvek seznamu na prvé místo a rekurzivně doplní všemi možnými permutacemi ostatních prvků. Poté se vždy otestuje, zda jde již o seřazený seznam pomocí booleovské funkce **ordered**, která postupem složitosti $\Theta(N)$ testuje sekvenčně splnění nerovností $A_{\pi(1)} \leq A_{\pi(2)} \leq \dots \leq A_{\pi(N)}$. Jádrem algoritmu je cyklus:

```
if ORDERED(A) then return(A)
else while not ORDERED(A) do
  next_permutation(A);
return(A);
```

Tento algoritmus je však extrémně špatný. Kroky časové složitosti $\Theta(N)$ se provádí v pesimistickém případě $n!$ -krát, v průměrném $(n!/2)$ -krát. Celková složitost je tedy $\Theta(N \cdot N!)$, což je zcela nevyhovující.

Všechny „rozumné“ algoritmy vnitřního asociativního řazení jsou založeny na nějaké modifikaci postupu „rozděl a panuj“.

Lze je klasifikovat podle dvou nezávislých binárních kritérií:

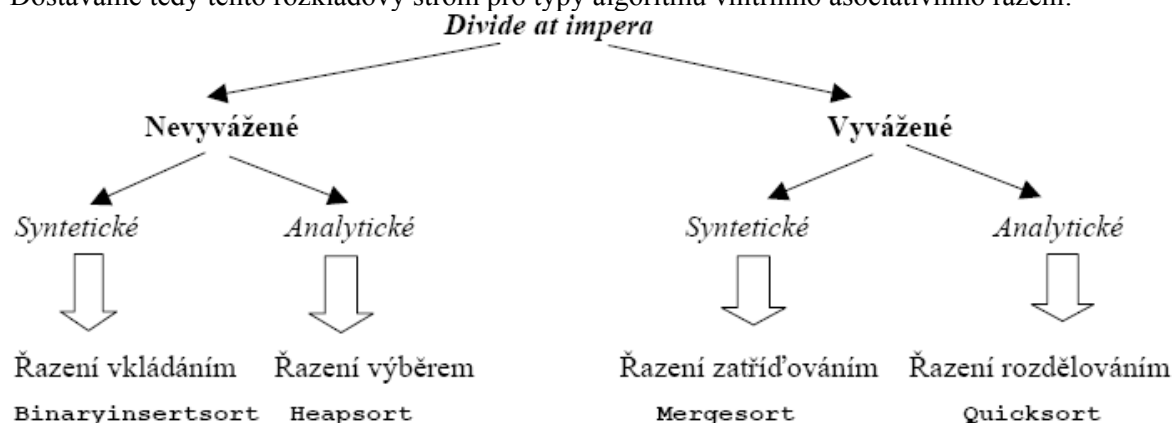
1. **Vyvážené**, které rozkládají problém na přibližně stejně velké díly.
2. **Nevyvážené**, kde má jeden z podproblémů pouze rozměr 1 a druhý $N - 1$.

Dále, podle dalšího kritéria na:

A. **Syntetické**, kde hlavní díl práce spočívá v sloučení řešení obou podproblémů.

B. **Analytické**, kde hlavní díl práce spočívá v rozkladu problému na podproblémy.

Dostáváme tedy tento rozkladový strom pro typy algoritmů vnitřního asociativního řazení:



Na posledním řádku uvádíme příklady algoritmů této třídy, jejichž časová složitost je $\Theta(N \cdot \log N)$ nebo se této složitosti blíží.

Řazení vkládáním [Insert sorting]

Algoritmy jsou založeny na sekvenčním vkládání prvků do seřazeného seznamu. Uvedeme schéma nerekurzivní a rekurzivní varianty:

```
begin
for j = 2 to N do
vlož A[j + 1] do seřazené posloupnosti A[1] ... A[j]
end
recursive procedure INSERTSORT (j)
begin if j > 1
then begin INSERTSORT(j - 1);
vlož A[j] do seřazené posloupnosti A[1] ... A[j]
end
end
```

Pro vložení může být užit sekvenční algoritmus a získáme tak algoritmus časové složitosti $\Theta(N^2)$. Užijeme-li binární vkládání do pole, bude mít opět algoritmus složitost $\Theta(N^2)$, protože v každém kroku potřebujeme $\Theta(N)$ operací na posun prvků v poli. Při jednosměrném lineárním spojovém seznamu budeme mít problém s přímým přístupem k prvkům proudu a složitost bude opět $\Theta(N^2)$. Použijeme-li ale binární strom typu „inorder“ pro reprezentaci seřazených seznamů, bude složitost každého kroku v cyklu $\Theta(\log N)$ a celková složitost takto navrženého algoritmu **binaryinsertsort** pouze $\Theta(N \cdot \log N)$. Algoritmus bude však náročnější na paměť.

Řazení výběrem [Select sorting]

Princip těchto algoritmů spočívá v postupném výběru nejmenších prvků z dosud neseřazené části seznamu. To se provede postupným porovnáváním kandidáta na minimum s ostatními prvky dosud neseřazené části seznamu. Každý krok má zřejmě složitost $\Theta(N)$, takže při jednoduchém uspořádání algoritmu bude výsledná složitost $\Theta(N^2)$.

Modifikací tohoto postupu, která porovnává a zaměňuje vždy pouze sousední prvky je algoritmus tak zvaného bublinkového řazení:

```
while NOTORDERED do
for i := 1 to n - 1 do
if A[i] > A[i + 1] then swap( A[i], A[i + 1]);
{NOTORDERED je Booleovská funkce, která má na počátku hodnotu TRUE a změní ji na FALSE,
pokud v cyklu nedojde k žádnému přehození - přehození se realizuje funkcí swap}
```

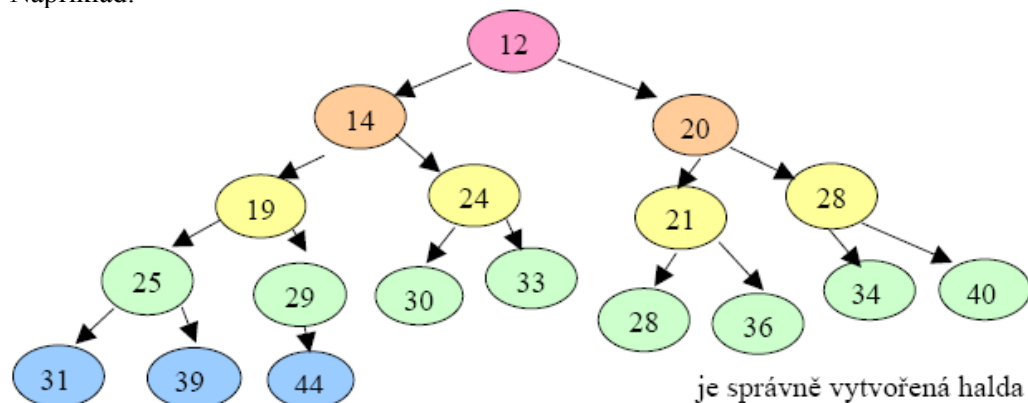
Pesimistická časová složitost bublinkového řazení je také $\Theta(N^2)$.

Abychom dostali účinný algoritmus tohoto typu, musíme opět užít speciální datovou strukturu pro reprezentaci seřazených seznamů. Jednou z nich je struktura zvaná **halda** (hromada) [heap], která umožňuje odebírání a vkládání prvků v logaritmickém čase. Princip její organizace je založen na principu vylučovacího sportovního turnaje.

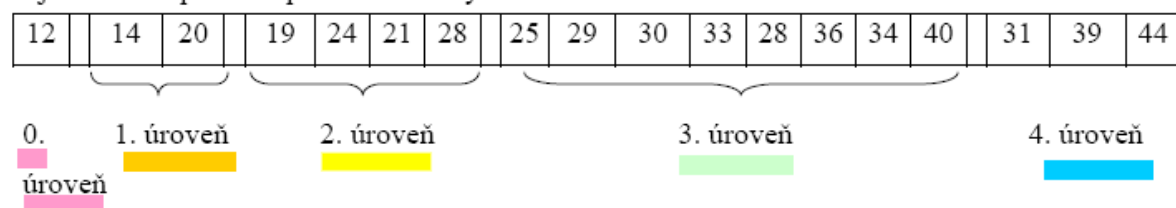
Přesněji: **Halda** (též „hromada“) [heap] je **binární strom** těchto vlastností:

- Všechny listy stromu mají hloubku vnoření od kořene h nebo $h - 1$.
- Všechny listy hloubky $h - 1$ leží vlevo od listů hloubky h .
- Synové každého uzlu ve stromu odpovídají vždy prvkům s větším klíčem než je klíč otce.

Například:



Její realizace polem v paměti může být



Pseudokód algoritmu heapsort může být (vstupem je proud, čtený do položky **item** s využitím zásady „čtení s předstihem“):

```

{variable H denotes the heap (var H: array[1 .. N] of integer)}
{variable END defines the actual end index of the heap}
END := 0; {the heap is empty}
read (item);
while item ≠ eof do
begin
insert_to_the_heap (item);
read (item)
end
while END > 0 {the heap is not empty} do
begin
select_min_from_the_heap (item);
write (item);
end
procedure insert_to_a_the_heap (K);
begin
END := END + 1; H[END] := K; P := END;
while (P > 1 and H[P div 2] > H[P] do
begin
swap (H[P div 2], H[P]); P := P div 2
end
end;
procedure select_min_from_the_heap (var: min);
begin
min := H[1];
if END > 1 then H[1] := H[END];
END := END - 1; P := 1; {heap reorganisation follows}
while (2 * P + 1 ≤ END and H[P] > H[P * 2] or H[P] > H[2 * P + 1]) do
if ( H[2 * P] > H[2 * P + 1] then
begin
swap (H[P], H[2 * P]); P := 2 * P
end
else
begin
swap (H[P], H[2 * P + 1]); P := 2 * P + 1;
if (2 * P = END and H[P] > H[2 * P] then swap (H[P], H[2 * P]
end;

```

Každá z obou procedur **insert_to_the_heap** a **select_min_from_the_heap** potřebuje $\Theta(\log n)$ porovnání a další počet operací shora omezený pevně daným násobkem těchto porovnání. Pro n prvků je tedy celková $\Theta(n \cdot \log n)$.

Razení sléváním [Merge sorting]

Algoritmy této skupiny jsou přímou aplikací principu „rozděl a panuj“. Seznam se rozdělí na dvě přibližně stejné části.

Každá se seřadí rekurzivně. Poté se užije algoritmus slévání monotonií **merge**.

```

recursive procedure MERGESORT(S);
begin
if |S| = 1
then return (S) { |S| denotes the cardinality - (size) of S }
else split S into disjoint sets S1 and S2, such that
|S1| = |S| div 2;
return MERGE( MERGESORT (S1), MERGESORT (S2));
end;

```

Slévání (zatříd'ování) procedurou **MERGE** dvou seřazených posloupností $x_1 \leq x_2 \leq \dots \leq x_M$ a $y_1 \leq y_2 \leq \dots \leq y_N$ se provede vždy porovnáním prvních prvků x_1 a y_1 . Menší je přesunut na výstup. Po uprázdnění některé z posloupností se výstup doplní zbytkem té druhé.

```

procedure MERGE(M, N);
begin
i, j, k := 1;
while (i ≤ M and j ≤ N) do
begin
if X[i] ≤ Y[j] then begin
Z[k] := X[i]; i := i + 1
end
else begin
Z[k] := Y[j]; j := j + 1
end;
k := k + 1

```



```

end;
while i ≤ M do begin Z[k] := X[i]; i := i + 1; k := k + 1 end;
while j ≤ N do begin Z[k] := Y[j]; j := j + 1; k := k + 1 end
end;

```

Pro počet kroků $T(N)$ algoritmu **MERGESORT** užitého na seznam délky N platí:

$T(1) = \Theta(1)$,

$T(N) = T(\lfloor N/2 \rfloor) + T(\lceil N/2 \rceil) + f(N)$, pro $N \geq 2$,

Dále je zřejmě $f(N) = \Theta(N)$.

Po zaokrouhlení zlomků dostáváme: $T(N) = 2 \cdot T(N/2) + \Theta(N)$

A tedy $T(N) = \Theta(N \cdot \log N)$.

Algoritmus **MERGE** pro slévání monotonií, užívaný při externím řazení, má zřejmě pouze lineární časovou složitost $\Theta(N)$.

Řazení rozdělováním [Divide sorting]

Princip této třídy algoritmů spočívá v rozdělení řazeného seznamu na dvě části „pokud možno stejné nebo podobné“ (rovnovážné) délky tak, že všechny prvky v jedné části jsou menší nebo rovné než prvky v druhé části. Tento krok se použije rekurzivně.

Rozdělení se realizuje výběrem prvku zvaného **pivot**. Do jedné části se zařadí prvky s klíčem menším než pivot, do druhé prvky s klíčem větším nebo rovným než pivot. Pivot je zařazen mezi obě části.

Pseudokód algoritmu quicksort na tomto principu je následující:

```

recursive procedure QUICKSORT(S);
begin
if |S| ≤ 1 then return (S) { |S| is the cardinality of S }
else
begin
select "anywise" a pivot P from S;
create sets S1 = {x ∈ S: x < P},
S2 = {x ∈ S: x = P},
S3 = {x ∈ S: x > P};
return (QUICKSORT(S1), S2, QUICKSORT(S3))
end;

```

Problém je samozřejmě ve vymezení „anywise“ – nějak. Na této volbě záleží časová složitost algoritmu. Při náhodném výběru je pesimistická složitost $\Theta(n^2)$. Vyplývá to z následující úvahy. Vybereme-li jako pivot nejmenší nebo největší prvek, bude jedna z částí prázdná. Seznam délky M tak rozdělíme v poměru $1 : M - 1$. Pro výchozí rozměr dat N budeme tedy potřebovat $N - 1$ kroků, každý s časovou $\Theta(N)$. Celková časová složitost tedy bude $\Theta(N^2)$.

Na druhé straně, pokud se podaří za pivot zvolit medián, bude potřeba $\log N$ rekurzivních volání. Celková složitost algoritmu bude pouze $\Theta(N \cdot \log N)$. Lze dokázat (důkaz není příliš snadný), že i průměrná složitost algoritmu **quicksort** při náhodné volbě pivotů $\Theta(N \cdot \log N)$.

Zdá se, že volbou mediánu lze vyloučit případy s nepříznivou časovou složitostí. Tudy však cesta nevede. Nalezení mediánu je totiž úloha stejně časově složitá jako řazení. Někdy se tento problém obchází rozdělováním podle aritmetického průměru, místo podle mediánu. Výpočet aritmetického průměru má složitost $\Theta(N)$. Hodnota aritmetického průměru se však může od mediánu dosti lišit a rozdělení podle průměrného klíče nemusí být rovnovážné.

Maticová algebra a soustavy lineárních rovnic

V mnoha odvětvích aplikované matematiky a v řadě výpočetních metod se často pracuje maticemi. Tyto operace bývají časově náročné. Proto je třeba věnovat příslušným algoritmům zvýšenou pozornost.

Matice typu (m, n) je definována jako tabulka s m řádky a n sloupci. V dalším budeme předpokládat, že prvky matice jsou reálná čísla.

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \dots & \dots & \dots & \dots \\ a_{m,1} & a_{m,2} & \dots & a_{m,n} \end{pmatrix} = (a_{i,j})_{i=1,\dots,m}^{j=1,\dots,n} = (a_{i,j})$$

Řádky matice budeme značit $\mathbf{r}_1(\mathbf{A})$, ..., $\mathbf{r}_m(\mathbf{A})$, sloupce $\mathbf{c}_1(\mathbf{A})$, ..., $\mathbf{c}_n(\mathbf{A})$. Řádky matice jsou n -dimensionální vektory, sloupce m -dimensionální vektory.

Součet dvou matic $\mathbf{A} + \mathbf{B}$ je definován pouze pro matice téhož typu (musí mít týž počet řádků a týž počet sloupců). Součtová matice má na daném místě součet prvků obou sčítanců. Časová složitost výpočtu součtu dvou matic je $\Theta(m \cdot n)$ a tento odhad nelze zlepšit.

Součin dvou matic $A \cdot B$ je definován tehdy a jenom tehdy, je-li první matice typu (m, n) a druhá typu (n, p) , tedy když počet sloupců první matice je týž jako počet řádků druhé matice. Výsledný součin je matice (m, p) a pro prvky součinu C

$$= (c_{i,j}) \text{ platí: } c_{i,j} = \sum_{k=1}^n a_{i,k} \cdot b_{k,j}$$

Každý prvek součinu je tedy skalárním součinem $r_i(A) \cdot c_j(B)$ vektorů $r_i(A)$ a $c_j(B)$. Skalární součin lze zřejmě vypočítat v čase $\Theta(n)$. Časová složitost násobení matic je tedy $\Theta(m \cdot n \cdot p)$.

Čtvercové matice typu (n, n) , kterým říkáme také matice řádu n , lze tedy přímo podle definice násobit s časovou složitostí $\Theta(n^3)$, ovšem pokud za míru rozsahu vstupu algoritmu vezmeme řád n . Pokud bychom za rozměr vstupu brali skutečný počet položek, který je $2 \cdot n^2$, bylo by ovšem tato složitost pouze $\Theta(n^{3/2})$. Běžně se však užívá první způsob a za míru vstupu se považuje n .

Poznamenejme, že čtvercové matice s operacemi součtu a součinu tvoří algebru. Platí pro ni podobná pravidla jako pro algebru čísel. Existují však odlišnosti. V algebře matic existují netriviální dělitelé nuly. Součin dvou nenulových matic může být matice tvořená samými nulami. Na rozdíl od algebry reálných čísel zde také neplatí, že ke každému prvku existuje inverzní prvek vzhledem k násobení. Inverzní matice existují pouze k regulárním maticím (takovým, které mají hodnotu rovnou svému řádu, respektive nenulovou hodnotu svého determinantu).

Naznačíme, jak lze užít metody „rozděl a panuj“ pro získání algoritmu pro násobení matic s nižší časovou složitostí.

Pro jednoduchost předpokládejme, že řád matice je mocnina dvou, $n = 2^r$. Pokud by tomu tak nebylo, doplnili bychom řádky i sloupce na matici tohoto řádu. Takovou matici můžeme reprezentovat jako matici řádu 2, složenou z bloků, které jsou matice řádu $n/2$. Zřejmě je:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \cdot \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} A.E + B.G & A.F + B.H \\ C.E + D.G & C.F + D.H \end{pmatrix}$$

Pro časovou složitost dostáváme rekurentní $T(n) = 8 \cdot T(n/2) + \Theta(n^2)$,

Který vede opět na složitost $\Theta(n^3)$, protože $\log_2 8 = 3$. Zdá se tedy, že princip „rozděl a panuj“ nám zde nepomůže.

Urychlení algoritmu spočívá na nápadu Strassena, který publikoval algoritmus násobení čtvercových matic druhého řádu, který používá pouze sedm násobení místo osmi, avšak 18 sčítání místo 4. Platí i to pro blokové matice. Lze se přesvědčit, že:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \cdot \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} S_1 + S_2 - S_4 + S_6 & S_4 + S_5 \\ S_6 + S_7 & S_2 - S_3 + S_5 - S_7 \end{pmatrix}$$

kde

$$S_1 = (B - D) \cdot (G + H), \quad S_2 = (A + D) \cdot (E + H),$$

$$S_3 = (A - C) \cdot (E + F), \quad S_4 = H \cdot (A + B),$$

$$S_5 = A \cdot (F - H), \quad S_6 = D \cdot (G - E),$$

$$S_7 = E \cdot (C + D).$$

Rekurentní vztahy pro tento algoritmus jsou $T(n) = 7 \cdot T(n/2) + \Theta(n^2)$.

Tedy podle naší obecné věty je časová složitost Strassenova algoritmu $T(n) = \Theta(n^\alpha)$, kde $\alpha = \log_2 7 \approx 2,81 < 3$,

Což je lepší výsledek než u standardního algoritmu vycházejícího přímo z definice. Vzhledem k tomu, že kroky rekurse obsahují mnohem více sčítání, jejichž počet je ovšem v každém kroku rekurse konstantní, projeví se zrychlení až u matic vyšších řádů. To je situace typická pro mnoho sofistikovaných algoritmů směřujících k zlepšení časové složitosti výpočtu. Pokusy kdy se zlepšení projeví mohou být zajímavé.

Spíše jako zajímavou ilustraci uveďme následující: Pokud se pokusíme užít též princip „rozděl a panuj“ tak, že rozdělíme matici na 3×3 blokové matice, nejlepší známý algoritmus pro součin potřebuje $23 > 3 = 21,8 \dots$ násobení. Výsledek tedy není lepší než při rozdělení na bloky 2×2 . Až při dělení na 70×70 bloků je znám algoritmus potřebující pouze 143640 násobení. Ten vede na složitost $\Theta(n^7 \log_2^{2,795} \dots)$, která je lepší než $\Theta(n^{2,81 \dots})$. Podle informací, které by měly být (snad, a bez záruky) čerstvé má nejlepší dosud publikovaný algoritmus pro násobení matic časovou složitost $\Theta(n^{2,376 \dots})$. Je však založen na mnohem složitějších úvahách než jsou ty, které jsme provedli.

Soustavy lineárních rovnic a inverze matice

Problém je při dané čtvercové matici $A = (a_{i,j})_{i=1, \dots, m}^{j=1, \dots, n}$ a vektoru $b = (b_1, b_2, \dots, b_n)$ nalézt vektor $x = (x_1, x_2, \dots, x_n)$ takový,

že

$$a_{1,1} \cdot x_1 + a_{1,2} \cdot x_2 + \dots + a_{1,n} \cdot x_n = b_1$$

$$a_{1,2} \cdot x_1 + a_{2,2} \cdot x_2 + \dots + a_{2,n} \cdot x_n = b_2$$

...

$$a_{1,n} \cdot x_1 + a_{n,2} \cdot x_2 + \dots + a_{n,n} \cdot x_n = b_n$$

$$\text{Budeme předpokládat, že matice soustavy } A = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \dots & \dots & \dots & \dots \\ a_{m,1} & a_{m,2} & \dots & a_{m,n} \end{pmatrix} = (a_{i,j})_{i=1,\dots,m}^{j=1,\dots,n} = (a_{i,j})$$

je regulární, a tedy soustava rovnic má jediné řešení.

Soustavu lze napsat maticově ve tvaru $A \cdot x = b$, kde vektor b pravých stran a hledaný vektor neznámých jsou považovány za matice řádu $(n, 1)$ o n řádcích a jedním sloupci ("sloupcové" vektory).

Vyšetřujeme tak zvanou rozšířenou matici soustavy typu $(n, n+1)$.

$a_{1,1}, a_{1,2}, \dots, a_{1,n}, b_1$

$a_{2,1}, a_{2,2}, \dots, a_{2,n}, b_2$

$A^* = \dots$

$a_{n,1}, a_{n,2}, \dots, a_{n,n}, b_n$

a označme řádky této matice A^* jako r_1, r_2, \dots, r_n a sloupce jako c_1, c_2, \dots, c_n, b .

Nejčastěji užívaná eliminační metoda zobecňuje postup užívaný již na základní škole pro řešení soustav dvou lineárních rovnic pro dvě neznámé. Je založena na tak zvaných **ekvivalentních transformacích** rozšířené matice soustavy A^* . Tyto ekvivalentní transformace nemění řešení soustavy. Jsou to:

- Vynásobení libovolného řádku matice nenulovým číslem.
- Přičtení násobku libovolného řádku k jinému řádku.
- Záměna pořadí (prohození) libovolných dvou řádků.

Užitím těchto ekvivalentních transformací postupně změňme rozšířenou matici soustavy A^* tak, aby její levá část, matice A , byla jednotková. Tedy aby měla v diagonále jedničky ($a_{i,i} = 1$ pro $i = 1, \dots, n$) a mimo ni nuly ($a_{i,j} = 0$ pro $i \neq j$). Po této transformaci bude vektor b představovat řešení soustavy.

Transformace provedeme tak, že postupně, sloupec po sloupci, budeme dostávat ekvivalentními transformacemi nuly do všech míst, kromě diagonály.

Algoritmus by selhal, v případě, kdy by se v diagonále objevila nula. V tom případě zaměníme celý řádek r_i , pro který je $a_{i,i} = 0$ s nějakým r_j pro $j > i$, pro který je $a_{i,j} \neq 0$. Protože matice A je regulární, takový nenulový prvek určitě existuje. Pokud by neexistoval, byla by matice singulární a soustava rovnic by buď neměla žádné řešení nebo by měla řešení nekonečně mnoho. Z důvodu numerické stability algoritmu lze doporučit záměnu s takovým řádkem, pro který se diagonální prvek co nejvíce liší od nuly, tedy s prvkem, pro který je $|a_{i,j}|$ je maximální.

Schéma algoritmu je následující:

```
procedure eliminate(var: A*);
begin
  for i := 1 to n do
  begin
    find j ≥ i with the maximal abs(a[i, j]);
    swap (r[i], r[j]);
    r[i] := r[i] / a[i, i];
    for k = 1 to n do
      if k ≠ i then r[k] := r[k] - r[i]*a[k, i];
    end
  end;
{b is the vector of solutions in the end of the algorithm}
```

Každý řádek, který pracuje s celým vektorem (obsahuje červené písmeno r) má časovou složitost $\Theta(n)$. Tedy vnitřní cyklus má složitost $\Theta(n^2)$. V vnějším cyklu **for** jsou tedy tři příkazy složitosti $\Theta(n)$ a jeden složitosti $\Theta(n^2)$. Výsledný algoritmus bude tedy mít časovou složitost $\Theta(n^3)$, samozřejmě vzhledem k počtu rovnic. Složitost vzhledem k objemu vstupních dat, která představují $n \cdot (n-1)$ položek bude $\Theta(n^3)$.

Princip „rozděl a panuj“ nám zde zřejmě nemůže pomoci

Pro algoritmus **nižší časové složitosti** se někdy užívají **přibližné iterační metody**.

Někdy lze pro řešení soustavy $A \cdot x = b$ užít formuli typu: $x^{(k+1)} = x^{(k)} - \tau \cdot (A \cdot x^{(k)} - b)$

v iteracích začínajících počátečním odhadem řešení $x^{(0)}$. Pokud matice A splňuje určité podmínky, lze dokázat, že posloupnost postupných iterací $x^{(0)}, x^{(1)}, x^{(2)}, \dots$ konverguje k řešení x soustavy $A \cdot x = b$.

Každý krok iterace má zřejmě $\Theta(n^2)$. Pokud proces konverguje (tak tomu ovšem není vždy), záleží časová složitost na tom, jaká je rychlost této konvergence. Tedy na tom po kolika krocích se přiblížíme k přesnému řešení tak blízko, že přiblížení můžeme za řešení považovat. Pokud nemáme k dispozici přesný horní odhad chyby, spokojíme se obvykle tím, že dvě nebo více „posledních“ aproximací se od sebe liší již jen nepatrně. Počet potřebných kroků iterace může pochopitelně záviset na řádu soustavy n . Vhodnou volbou parametru τ , lze někdy (ne vždy) dosáhnout toho, že počet potřebných kroků je nižší než $\Theta(n)$, například $\Theta(n^{1/2})$. V tomto případě má celý algoritmus složitost nižší než $\Theta(n^3)$. Nevýhodou ovšem je, že konvergence metody závisí na vlastnostech matice A a nelze ji garantovat obecně. Dále pak to, že získané řešení je pouze přibližné.

Jednoduchá modifikace eliminačního algoritmu pro řešení soustavy lineárních rovnic, složitosti $\Theta(n^3)$ může být užita i pro **výpočet inverzní matice** k dané regulární matici.

Inverzní matice A^{-1} matici A je matice pro kterou $A \cdot A^{-1} = I$, kde I je jednotková matice, která má v diagonále jedničky a mimo diagonálu samé nuly.

Eliminaci zahájíme s maticí $A = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} & 1 & 0 & \dots & 0 \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} & 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ a_{m,1} & a_{m,2} & \dots & a_{m,n} & 0 & 0 & \dots & 1 \end{pmatrix}$

a proceduru **eliminate** provedeme s řádky délky $2 \cdot n$ (řešíme tedy současně n soustav rovnic se stejnými pravými stranami). Po ukončení procedury bude v “pravé části” transformované matice $A^\#$ matice A^{-1} . Časová složitost tohoto algoritmu je zřejmě opět $\Theta(n^3)$.

Hladové algoritmy [Greedy algorithms]

Tyto algoritmy představují “nejistý”, avšak často velmi užitečný návod jak získat řešení řady optimalizačních úloh s dobrou časovou složitostí.

Jsou založeny na principu v každém okamžiku volit mezi různými možnostmi **tu variantu, která se lokálně jeví jako nejvýhodnější**. Tato zásada je pouze heuristická. K nejlepšímu řešení vede často, ne však vždy.

Uvedeme několik typických příkladů.

Optimální plánování aktivit

Mějme konečný počet aktivit $1, 2, \dots, N$, které nelze provozovat současně. Každá z nich má přesně určený začátek a přesně určený konec.

$(s_1, f_1), (s_2, f_2), \dots, (s_N, f_N)$, $s_i < f_i$ pro každé $i = 1, \dots, N$.

Úkol je nalézt největší počet aktivit, které by bylo možné uskutečnit bez vzájemného časového překrytí (Například nalézt maximální možné využití dané učebny při pevných požadavcích rozvrhu výuky, či v daném období stihnout sledování co nejvíce televizních pořadů).

Řešení „hrubou silou“, které nalezení optima garantuje, spočívá v prověření všech možných podmnožin množiny $\{1, 2, \dots, N\}$ a výběru té, která má nejvíce prvků. Takovýto algoritmus má ovšem složitost $\Theta(2^N)$ a je tedy prakticky bezcenný.

Následující „hladový“ algoritmus je založen na principu „co nejnadějnějšího kroku“, který spočívá v zásadě ponechat vždy co nejvíce volného času pro umístění dalších dosud neumístěných aktivit. Idea je:

“Pokud to lze, opakuj následující krok: Vyber aktivitu, kterou umístit lze a která končí co nejdříve. Aby co nejvíce volného času zbylo pro další aktivity.”

Pro realizaci tohoto algoritmu je vhodné aktivity seřadit podle koncových časů tak, aby $f_1 \leq f_2 \leq \dots \leq f_N$.

Toho lze dosáhnout se složitostí $\Theta(N \cdot \log N)$, některým z „inteligentních“ algoritmů pro řazení. Dál můžeme postupovat podle následujícího schématu, ve kterém dvojité složené závorky $\{\{ \dots \}\}$ označují množinu (pro odlišení od poznámky v jednoduchých složených závorkách $\{ \dots \}$).

```
A :=  $\{\{1\}\}$ ; j := 1
for i := 2 to n do
  if  $s[i] \geq f[j]$  then
    begin
      A :=  $A \cup \{\{i\}\}$ ; j := j + 1
    end; {A is the set of selected activities}
```

Protože vlastní algoritmus výběru aktivit ze seřazeného seznamu aktivit má časovou složitost pouze $\Theta(N)$, je celková složitost algoritmu, včetně řazení $\Theta(N \cdot \log N)$. Takto nalezené řešení je skutečně optimální, jak lze snadno dokázat matematickou indukcí. Poznamenejme však, že to není vždy úplná samozřejmost. Optimálnost řešení ke kterému vedou hladové algoritmy je třeba vždy případ od případu zvážit a dokázat.

Poznamenejme, že při jiné interpretaci požadavku „volit lokálně co nejvýhodnější dílčí řešení“, například, kdybychom umísťovali kamkoliv nejkratší dosud neumístěnou aktivitu, nemuseli bychom nalézt optimum.

Prohledávání grafu do hloubky a do šířky

Mnoho úloh aplikované matematiky je výhodné formulovat pomocí grafů (neorientovaných i orientovaných). Pro návrh algoritmů řešení těchto úloh je potřeba reprezentovat graf jako datovou strukturu.

Pro průchod algoritmu grafem, který je uložen v paměti jako datová struktura se užívají nejčastěji dvě strategie, které umožňují prohledat graf v lineárním čase a zjistit potřebné informace o něm. Tyto strategie odpovídají dvěma disciplinám organizace fronty.

- **Prohledávání do hloubky** [dept-first search] odpovídá organizaci zásobníku jako fronty LIFO s operacemi **push** pro uložení na vrchol zásobníku a **pop** pro odebrání vrcholu zásobníku.
- **Prohledávání do šířky** [breadth-first search] odpovídá organizaci na základě discipliny FIFO pro nalezení dalšího uzlu v grafu.

Typickou úlohou, která vyžaduje prohledat graf je tato úloha:

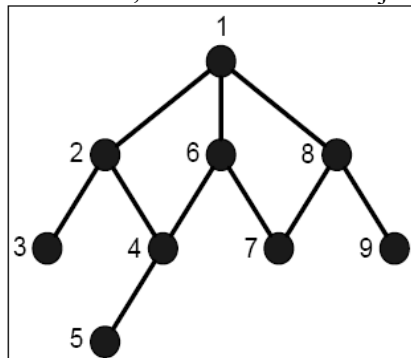
Je dát uzlově ohodnocený graf. To je každému uzlu je přiřazena nějaká hodnota (číslo, symbol, text, ...). Úloha je **zjistit, zda z daného uzlu existuje nějaká cesta, která jej spojuje s nějakým uzlem ohodnoceným danou hodnotou. V případě, že taková cesta existuje, nalézt ji.** Problémy tohoto typu se vyskytují v úlohách automatické dedukce a hledání řešení nejrůznějších problémů.

Procedura neorientovaného prohledávání souvislého grafu do hloubky (DFS) očíslovává uzly v grafu takto:

Vyjde z daného uzlu u , který prohlásí za kořen stromu. Všechny hrany $(u, v) \in E$ uloží **do zásobníku**. Uzlu u přiřadí DFS-číslo 0 a nastaví DFS-počítadlo c na hodnotu 1. Poté opakuje následující akci, dokud zásobník není prázdný:

Nechť (x, y) je na vrcholu zásobníku. Potom uzel x je již očíslován. Pokud uzel y dosud žádné DFS-číslo nemá, přiřadíme mu hodnotu c DFS-počítadla, zvětšíme hodnotu tohoto počítadla o 1 a zařadíme do zásobníku všechny hrany (y, z) , které vycházejí z uzlu y . Hrana (x, y) bude hranou námi konstruovaného stromu. V opačném případě, pokud uzel y již označen je, pouze odstraníme operaci pop hranu (x, y) ze zásobníku.

Očíslování, které takto vznikne je na následujícím obrázku:

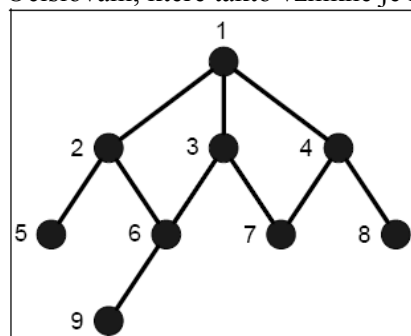


Očíslování uzlů grafu do hloubky

Vzniklý strom je kostrou [spanning tree] původního grafu. Algoritmus má lineární složitost $\Theta(N)$, kde N je počet hran v grafu.

Procedura neorientovaného prohledávání souvislého grafu do šířky (BFS) je analogická. Vyjdeme z daného uzlu u . V prvním kroku postupně očíslovujeme všechny uzly do kterých z uzlu u vede hrana a zařadíme je do fronty. Poté postupně odebereme z fronty tyto uzly a na konec fronty zařadíme všechny dosud neočíslované uzly, do kterých vede hrana. Takto postupujeme až do vyprázdnění celé fronty a očíslování všech uzlů v grafu.

Očíslování, které takto vznikne je na následujícím obrázku:



Očíslování uzlů grafu do šířky

Postup ilustrujeme na příkladu řešení známé hádanky.

Příklad řešení problému: Vlk, koza a hlávka zelí

Farmář jde z trhu. Má s sebou vlka, kozu a hlávku zelí. Má překročit řeku. K dispozici je loďka. Může však na ni vzít jen jedno zvíře nebo věc. Vlk s kozou ani kozu ze zelím nesmí nechat na stejném břehu bez dozoru. Jak má postupovat?

Řešení: Je celkem 16 možných situací, které je možné znázornit grafem nebo tabulkou. Řádky v tabulce jsou uzly stavového grafu. Písmenka znamenají: (F) - farmář, (V) - vlk, (K) - koza a (Z) - zelí.

Prohledáváním grafu do hloubky (DFS) dostaneme řešení 1, 9, 1, 10, 2, 13, 3, 11, 3, 15, 4, 12, 4, 14, 6, 16.

Prohledáváním do šířky (BFS) řešení 1, 9, 10, 11, 12, 3, 13, 15, 2, 5, 4, 7, 14, 6, 16.

Existuje i řešení na menší počet přejezdů řeky (nejméně na 7). Například 1, 11, 3, 13, 2, 14, 6, 16.

O tom jak hledat nejkratší cestu mezi dvěma uzly v grafu se zmíníme v následujícím odstavci.

Číslo stavu	Levý břeh	Pravý břeh	Stav je bezpečný	Mohou následovat stavy
1.	FVKZ	--	+	9. 10. 11. 12.
2.	FÍK	Z	+	10. 13. 14.
3.	FVZ	K	+	11. 13. 15.
4.	FKZ	V	+	12. 14. 15.
5.	FV	KZ	-	13. 16.
6.	FK	VZ	+	14. 16.
7.	FZ	VK	-	15. 16.
8.	F	VKZ	-	16.
9.	WKZ	F	-	1.
10.	VK	FZ	-	1. 2.
11.	VZ	FK	+	1. 3.
12.	KZ	FV	-	1. 4.
13.	V	FKZ	+	2. 3. 5.
14.	K	FVZ	+	2. 4. 6.
15.	Z	FVK	+	3. 4. 7.
16.	--	FVKZ	+	5. 6. 7. 8.

Jako cvičení lze řešit podobnou úlohu o třech misionářích a třech lidojedech. V cestě je opět řeka a do lodičky se vejdou jenom dva lidé. Na žádném břehu nesmí nikdy (z pochopitelných důvodů) nastat převaha lidojedů nad misionáři.

Problém nejkratší cesty v grafu

Mějme orientovaný nebo neorientovaný hranově ohodnocený graf (V, E) . Necht' x a y jsou dva jeho uzly. Problém je nalézt cestu z uzlu x do uzlu y , takovou, aby součet ohodnocení (délek) všech hran na této cestě byl nejmenší ze všech cest z uzlu x do uzlu y . Pokud mají všechny hrany ohodnocení rovno jedné, dostaneme jako zvláštní případ této úlohy úlohu nalézt cestu z uzlu x do uzlu y s minimálním počtem hran. Pokud je graf souvislý nebo když uzly x a y leží v jedné souvislé komponentě nesouvislého grafu, má úloha vždy řešení. Prohledáváním grafu do hloubky nebo do šířky cestu sice nalezneme, nebude však obecně nejvýhodnější.

Tento problém má mnoho aplikací. Například nalézt nejkratší nebo nejlevnější cestu. Někdy je zajímavá i podobná úloha pro orientovaný graf (například cesta autem v městě, kde jsou jednosměrky).

Řešení „hrubou silou“ znamená testovat všechny podmnožiny množiny E všech hran v grafu. Vyloučit ty podmnožiny, které netvoří cestu z x do y , pro ostatní (pro ty, které cestu tvoří) vypočítat součet ohodnocení hran a nalézt minimum. Takový algoritmus má zřejmě časovou složitost $\Theta(n \cdot 2^n)$, kde n je počet hran v grafu, a je tedy nevhodný. Také kdybychom cestu popisovali jako možnou permutaci množiny uzlů, dostaneme $m!$ možností, kde m je počet uzlů v grafu a algoritmus složitosti $\Theta(m!)$, rovněž nepoužitelný.

Musíme tedy hledat jiný algoritmus. Princip konstrukce hladových algoritmů lze využít takto:

Budeme sestavovat nejprve cesty z uzlu x postupně, nejprve do uzlů, které leží v „blízkém okolí“ uzlu x , tak dlouho, dokud nedorazíme do uzlu y . V každém kroku vybereme vždy minimální cestu po které lze do daného uzlu dospět. V případě, že při tomto postupu „objevíme“ cestu kratší, provedeme opravu cest do všech v předchozích krocích dosud dosažených uzlů.

Necht' w je váhová funkce ohodnocení hran v daném grafu. Budeme předpokládat, že toto **ohodnocení je vždy kladné číslo** ($w: E \rightarrow \mathbb{R}^+$). Hledejme nejkratší cestu z uzlu a do uzlu b . Postup uvedeme pro orientovaný graf. Pro neorientovaný je analogický.

Zvolíme tento postup, zvaný Dijkstrův algoritmus:

1. Každému uzlu $z \neq a \in V$ grafu přiřadíme dočasnou hodnotu $dh(z) = \infty$, pouze uzlu a přiřadíme trvalou hodnotu $th(a) = 0$.
2. Je-li poslední uzel, kterému byla přiřazena trvalá hodnota $th(z)$, potom opravíme u všech uzlů z' , které s ním jsou spojeny nějakou hranou $(z, z') \in E$ jejich dočasnou hodnotu tak, že položíme $dh(z') := \min\{dh(z'), th(z) + w((z, z'))\}$. (zkrátíme si cestu).
3. Pro (nějaký) uzel z' s nejmenší dosaženou hodnotou $dh(z')$ položíme $th(z') = dh(z')$ (prohlásíme jeho dočasnou hodnotu za trvalou).

Kroky 2. a 3. střídavě opakujeme dokud b nedosáhne trvalou hodnotu, která je délkou nejkratší cesty.

Možná varianta je zapsána v pseudokódu:

```

begin
d(x) := 0; d(u) := ∞ for all other nodes from V;
R := ∅;
repeat
select such a node in R for which d(u) is minimal;

```

```

for all descendants v of u do
begin
if  $v \notin R$  and  $d(u) + w(u, v) < d(v)$  then
begin
 $d(v) := d(u) + w(u, v)$ ;
store u as a predecessor of v on the minimal path;
end;
 $R := R \cup \{v\}$ 
end
until  $y \in R$ 
end;
{The minimal length is  $d(y)$ , the minimal path can be reconstructed from the predecessors
of the nodes on the minimal path}

```

V tomto algoritmu potřebujeme vyšetřovat každou hranu grafu pouze jednou. Neorientovaný graf s m uzly má nejvýše $n = (m \cdot (m - 1)) / 2$ hran. Orientovaný graf s m uzly má nejvýše $n = m^2$ hran. Časová složitost tohoto algoritmu je tedy $\Theta(m^2)$.

Procedura má jednu „skrytou vadu“. Předpoklad kladného ohodnocení je podstatný. Pokud graf obsahuje smyčku (cyklus) u které je součet ohodnocení hran záporný, nikdy neskončí svoji práci. Nebude tedy algoritmem. Hladový postup bude stále nalézat „kratší“ a „kratší“ cesty a pracovat v nekonečném cyklu. Pro ohodnocení, u kterého se připouštějí i záporné hodnoty není obecně znám efektivní postup určení nejkratší cesty mezi dvěma uzly v grafu. Lze dokonce ukázat, že takto zobecněný problém nejkratší cesty je z hlediska složitosti ekvivalentní s problémem hamiltonovské cesty v grafu. Pro tento problém algoritmus s polynomiální složitostí neznáme.

Postup ilustrujme na příkladě grafu, v kterém hledáme cestu z uzlu x do uzlu y:

Postup ukazuje tabulka:

Krok	T	u	v	x	y	Z	Poznámka
0				0			Počáteční ohodnocení
1a	<i>1</i>			0		2	Sousedé počátečního uzlu
1b	1			0		2	Nejmenší dočasnou hodnotu změním na trvalou
2a	1		5	0		2	Přidáme další uzel
2b	1		5	0		2	Nejmenší dočasnou hodnotu změním na trvalou
3a	1	3	4	0		2	Další uzly a oprava hodnocení uzlu v
3b	1	3	4	0		2	Nejmenší dočasnou hodnotu změním na trvalou
4a	1	3	4	0	7	2	Poslední uzel
4b	1	3	4	0	7	2	Nejmenší dočasnou hodnotu změním na trvalou
5a	1	3	4	0	6	2	Oprava
5b	1	3	4	0	6	2	Nejkratší cesta má délku 6. Je to (x, z), (z, v), (v, y).

Dočasné hodnoty v tabulce jsou označeny kurzívou, konečné **tučně**.

Problém minimální kostry [Minimal spanning tree problem]

Další důležitý optimalizační problém pro grafy je problém nalezení minimální kostry (neorientovaného) grafu. Kostra souvislého grafu je jeho souvislý podgraf, obsahující všechny uzly grafu a žádný cyklus. Každá hrana kostry je „most“. Jejím odstraněním získáme již nesouvislý graf. Kostra je zřejmě strom.

Předpokládejme, že graf je hranově ohodnocený. Problém minimální kostry je úloha nalézt kostru daného souvislého grafu takovou, že součet ohodnocení jejích hran je minimální.

Nechť $G = (V, E)$ je souvislý hranově ohodnocený graf a f je váhová funkce zobrazující E do množiny kladných čísel. Hledáme kostru grafu tak, aby $\sum_{e \in E'} f(e)$ přes všechny hrany E' byl minimální na množině všech koster grafu G .

Problém má praktický význam například při návrhu minimální transportní sítě (uzly jsou města, hrany železnice či silnice, ohodnocení náklady pro vybudování spojení) nebo pro propojování počítačů do sítě.

Řešení „hrubou silou“ by znamenalo generovat všechny podmnožiny množiny všech hran daného grafu, které mají $m - 1$ prvků, kde m je počet uzlů v grafu. Počet hran stromu je totiž vždy o jedničku menší než počet jeho uzlů. Pro tyto výběry by bylo třeba testovat, zda tvoří kostru a mezi těmi, které kostru tvoří hledat minimum. Počet všech podmnožin o $m - 1$ prvcích vybraných z množiny mohutnosti n , kde n je počet hran grafu G , je ale

$$C(m, n) = m! / (n! \cdot (m - n)!),$$

Což vede k nepřijatelné nepolynomiální výpočetní složitosti.

Jeden z užívaných algoritmů „na hladovém principu“ pro vyhledání minimální kostry lze v „železniční terminologii“ popsat takto:

1. Vystavěj nádraží v libovolném městě.

2. Opakuj následující krok, pokud to lze: Z libovolného města, kam již železnice vede vystavěj železnici do dosud do města, do kterého zatím žádná železnice nevede, je-li více možností, vyber tu, která přijde nejlevněji.

Tento algoritmus, zvaný obvykle Primův algoritmus, je ve své podstatě příbuzný s Dijkstrovým algoritmem pro hledání nejkratší cesty. I zde je třeba v průběhu zahrnování dalších hran do kostry počítat s možností oprav. Jeho zápis v pseudokódu může být následující:

```
select some  $u \in V$ ;
 $X := \{\{u\}\}$ ,  $Y := \emptyset$ ;  $M := V \div \{\{u\}\}$ ;
for each  $v \in M$  do
  if  $(u, v) \in E$  then begin
    father( $v$ ) :=  $u$ ;  $d(v) := f((u, v))$ 
  end
  else  $d(v) := \infty$ ;
while  $M \neq \emptyset$  do
  begin
    select  $v \in M$  such that  $d(v) = \min\{(d(w) : w \in M)\}$ ;
     $X := X \cup \{\{v\}\}$ ;
     $Y := Y \cup \{(v, \text{father}(v))\}$ ;
     $M := M \div \{\{v\}\}$ 
    for each  $(v, w) \in M$  do
      if  $((u, w) \in E \text{ and } f((v, w)) < d(w))$ 
      then begin
        father( $w$ ) :=  $v$ ;
         $d(w) := f((v, w))$ 
      end
    end
  end
return  $H = (X, Y)$ ;
```

Takto nalezneme minimální kostru. Není to jasné „na prvý pohled, ale přesný důkaz je poměrně snadný. (Pokuste se o něj!).

Má-li graf m uzlů, má tento algoritmus časovou složitost $\Theta(m^2)$. Pro počet n hran souvislého grafu o m uzlech platí $m - 1 \leq n \leq (m \cdot (m - 1)) / 2$. Dolní meze se dosáhne jde-li o strom, horní jde-li o úplný graf.

Existuje však jiný algoritmus nalezení minimální kostry hranově ohodnoceného grafu, který má časovou složitost $\Theta(n \cdot \log n)$, kde n je počet hran v grafu. Tento algoritmus navrhl v roce 1926 český matematik profesor Borůvka a je znám jako „Borůvkův algoritmus“. Který z obou algoritmů pro minimální kostru je výhodnější záleží na „hustotě“ grafu. Borůvkův algoritmus je podstatně výhodnější pro „řidké“ grafy.

Borůvkův algoritmus (někdy zvaný též Kruskalův, ačkoliv tento autor jej publikoval mnohem později) spočívá v tom, že se nejprve seřadí všechny hrany grafu vzestupně podle velikosti. To lze provést „chytrým“ algoritmem řazení se složitostí $\Theta(n \cdot \log n)$, kde n je počet hran grafu. Poté se postupně sestavuje les z těchto hran tak, že se z seřazeného seznamu vybírají zleva doprava ty, které neuzavírají cyklus. Po $m - 1$ krocích tento proces končí nalezením jediného stromu, který je kostrou.

```
begin
  sort all edges on the graph depending on there weights;
  { This can be realized in the time  $\Theta(n \cdot \log n)$  }
   $S := \emptyset$ ;
  while  $S$  contains less the  $n - 1$  elements do
    go through sorted path list and select first path, which does not close any circle and
    insert this path to the set  $S$ , if such a path does not exists the graph  $G$  is not
    connected;
    {after finishing this while loop  $S$  is the minimal spanning tree}
  end;
```

Problém může nastat pouze s tím, jak rozhodnout, zda hrana, kterou chceme doplnit neuzavírá nějaký cyklus. V tom případě bychom ji totiž museli při zařazování do kostry odmítnout. Při řešení „pomocí obrázku“ je takovéto rozhodnutí snadné metodou „kouknu se a vidím“. Vyjádření tohoto testu algoritmem může však působit menší potíže. Užívá se tento postup. Definujme na množině všech uzlů grafu ekvivalenci tak, že na počátku bude každý uzel ekvivalentní pouze sám se sebou. Bude tedy m tříd ekvivalence po jediném prvku. Při pokusech zařadit hranu se vždy dotážeme, zda oba její vrcholy neleží ve stejné třídě ekvivalence. Pokud ano, hranu odmítneme do kostry zařadit. Pokud ne, obě třídy ekvivalence sloučíme a hranu akceptujeme. Invariant tohoto algoritmu bude skutečnost, že ekvivalentní budou vždy ty a jen ty uzly, které patří v postupně vytvářeném lese do jediné souvislé komponenty (stromu). Při této disciplíně zařazování nemůže vzniknout cyklus. Pokud budeme třídy ekvivalence reprezentovat poli nebo lineárními seznamy, může při provádění testu příslušnosti do třídy a/nebo slučování tříd vzniknout pro každý krok dodatečná výpočetní složitost $\Theta(m)$, kde m je počet uzlů grafu, která může být ve svém výsledku vyšší než je složitost řazení hran podle délky.

Algoritmy analogickými jako jsou algoritmy pro řazení (mergesort a heapsort) lze však při vhodné volbě struktury dat jeden test, zda hranu zařadit, či ne provést s časovou složitostí $\Theta(\log m)$, kde m je počet uzlů. Je pouze třeba uzly jedné třídy ekvivalence (tedy ty, které patří do stejné souvislé komponenty) propojit směrničky tak, aby vznikl binární strom

nebo jemu „blízký“ graf. Každá třída je pak charakterizována kořenem tohoto stromu. Známe-li uzel, lze příslušný kořen získat s logaritmickou složitostí a tak rozhodnout zda jsou dva uzly již v jedné komponentě nebo nikoliv. Pokud nejsou, vytvořit z jednoho stromu větev druhého. „Delikátní“ podrobnosti vynecháme.

Protože horní odhad počtu hran grafu $n = (m \cdot (m - 1)) / 2$ je vyšší než počet m jeho uzlů, je složitost Borůvkova algoritmu $\Theta(n \cdot \log n)$, kde n je počet jeho hran. To může být méně než $\Theta(m^2)$. Výpočet lze ještě urychlit organizací hran do haldy, což umožní, že řazení hran podle déky bude možné ukončit dříve, než se zpracují všechny. Pro konstrukci kostry (stromu) totiž stačí seřadit pouze prvních $m - 1$ (m je počet uzlů) hran a ty hrany, které byly vyloučeny protože způsobovaly cyklus.

Problém maximálního toku v síti

Další grafový problém s řadou aplikací je problém maximálního toku [maximal flow problem] daným grafem. Jedna z možných aplikací je rozvod plynu nebo kapaliny potrubím, které tvoří síť s omezenou kapacitou každého spoje. Jiná síť komunikačních kanálů s omezenou propustností danou objemem zpráv nebo síť elektrických vodičů s omezenou intenzitou proudu. Problém je nalézt maximální možný tok z jednoho uzlu (zdroje) do jiného uzlu.

Přesná formulace úlohy je následující:

Nechť V je konečná množina uzlů a nechť c je kapacitní funkce zobrazující $V \times V$ do množiny \mathbb{R}^+ všech kladných reálných čísel. Hodnota funkce $c(x, y)$ určuje maximální možný tok hranou z uzlu x do uzlu y . Nejsou-li uzly x a y spojeny hranou, je $c(x, y) = 0$. Nechť dále s a t jsou dva různé uzly, nazývané po řadě zdroj [source] a ponor [sink]. Situaci odpovídá hranově ohodnocený orientovaný graf (V, E) , kde množina E je určena jako $E = \{(u, v) : c((u, v)) > 0\}$.

Tento graf označíme $G = (V, E, c)$.

Funkce $f : V \times V \rightarrow \mathbb{R}$, kde \mathbb{R} je množina všech reálných čísel nazveme **tok** [flow], jestliže platí současně tyto podmínky:

1. Antisymetrie: pro všechna $u, v \in V$ je $f((u, v)) = -f((v, u))$.
2. Konzervativní zákon: pro všechny vnitřní uzly $u \notin \{s, t\}$ je $\sum_{v \in V} f(u, v) = 0$

3. Kapacitní omezení: pro všechna u, v je $f((u, v)) \leq c((u, v))$.

Říkáme, že orientovaná hrana (u, v) je saturována tokem f , jestliže je $f((u, v)) = c((u, v))$.

Je-li f tok grafem G s kapacitními omezeními c , budeme ohodnocení $r : c - f$ říkat residuální kapacita toku f . Residuální graf grafu $G = (V, E, c)$ vzhledem k toku f je graf $G_f(V, E_f, c)$, kde $E_f = \{(u, v) : r((u, v)) > 0\}$.

Residuální kapacita $r((u, v))$ určuje tok, který může procházet hranou (u, v) „navíc“ oproti toku f , aniž by došlo k narušení kapacitních omezení hran. V případě záporného toku $f((u, v))$ může být residuální kapacita větší než původní $c(u, v)$ o tok f z v do u . Residuální graf tedy může mít i jiné hrany než původní graf G . V residuálním grafu G_f však nemůže být hrana (u, v) , pokud v původním grafu nebyla ani hrana (u, v) , ani hrana ponor (v, u) . Maximální počet hran v množině E_f je $2 \cdot n$, kde n je počet hran v množině E .

Je-li dán graf G a tok f tímto grafem, říkáme, že cesta ze zdroje s do ponoru t je **nenasycená cesta** [augmenting path], pokud je i cestou v residuálním grafu G_f . Nenasycená cesta je posloupnost hran, jejichž kapacita je větší než tok a kterými lze tedy tok zvýšit.

Řez [cut] **v grafu** je dvojice disjunktních množin $A \subset V$ a $B \subset V$, taková, že $s \in A$ a $t \in B$.

Kapacita řezu je definována jako $c(A, B) = \sum_{u \in A, v \in B} f(u, v)$

Snadno se dokáže následující tvrzení nazývané „Maximální tok – Minimální řez“:

Následující výroky jsou ekvivalentní:

- a) f je maximální tok v grafu $G = (V, E, c)$,
- b) Existuje řez grafu takový, že má kapacitu f . Tedy: $c(A, B) = |f|$.
- c) V grafu G neexistuje žádná nenasyčená cesta.

Tato úvaha nám dává možnost navrhnout na hladovém principu algoritmus pro nalezení maximálního toku grafem v případě, že kapacity hran jsou celá nebo racionální čísla.

V případě iracionálních kapacit bychom museli omezení aproximovat racionálními. Jinak by procedura nemusela být resultativní. Po převedení na společného jmenovatele (změně jednotky) můžeme předpokládat, že kapacity hran v grafu G jsou celá čísla. Za poznámku stojí, že v případě iracionálních kapacit nelze proceduru, kterou navrhne užit ani pro postupná přiblížení k výsledku. Mohla by konvergovat k toku, který není maximální.

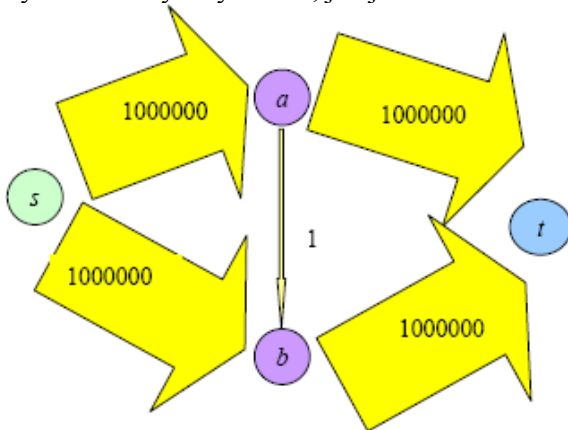
Neformální popis postupu zvaného **postupné nasycování cest** následuje:

1. Začneme nulovým tokem a libovolnou cestou z s do t v grafu G .
2. Pokud lze nalézt nějakou dosud nenasyčenou cestu p z s do t určíme nejmenší kapacitu $d > 0$ orientované hrany na této a zvýšíme tok po této cestě o hodnotu d . Tím získáme na cestě p maximální možný průtok. Opakujeme znovu bod 2.
3. Pokud již žádná nenasyčená cesta neexistuje, máme maximální tok.

Jsou-li kapacity celá čísla, zvýší se při každém kroku tok aspoň o 1. Maximální tok tedy získáme po nejvýše tolika krocích, kolik je (celočíslný) maximální tok.

Popsaný postup není striktně řečeno algoritmem, protože není deterministický. Výběr z více možných nenasyčených cest je „náhodný“. To lze snadno odstranit určením strategie jak budeme cesty vybírat. Například prohledáváním do hloubky nebo do šířky.

Časová složitost je zřejmě $O(|f^*|)$, kde f^* je maximální tok. Konkrétní složitost může velmi silně záviset na strategii výběru nenasyčených cest, jak je vidět z následujícího obrázku:



Kapacitní omezení jsou $c(s, a) = c(s, b) = c(a, t) = c(b, t) = 1000000$, $c(a, b) = c(b, a) = 1$.

Pokud zvolíme prvou cestu (s, a, t) pro 1000000 jednotek, bude mít residuální graf pouze jedinou cestu (s, b, t) pro dalších 1000000 jednotek a maximální tok 2000000 jednotek dostaneme ve dvou krocích. Pokud však zvolíme alternativně cesty (s, a, b, t) a (s, b, a, t) , budeme potřebovat 2000000 kroků pro dosažení maximálního toku.

Proto na volbě cest velmi záleží. Je třeba užít nějaké pravidlo, které úspěch „zcela nezaručuje“, většinou však k úspěchu vede. Taková pravidla se nazývají heuristická pravidla nebo heuristiky. Takovým pravidlem může být volit vždy nenasyčenou cestu s největší residuální kapacitou. Lze ukázat, že při této volbě je možné snížit odhad časové složitosti na $O(n \cdot \log |f^*|)$ kroků, kde n je počet hran v grafu a f^* maximální tok.

Jiná možná heuristika je volit vždy cestu o minimální délce. Tak dostáváme algoritmus časové složitosti $\Theta(n^2 \cdot m)$, kde n je počet hran a m počet uzlů v grafu. V tomto případě složitost nezáleží na kapacitách. Tato volba je zřejmě lepší.

Poznamenejme, že jsou známy ještě „lepší“, avšak podstatně složitější algoritmy řešení tohoto problému složitosti $\Theta(n \cdot m^2)$ a $\Theta(m^3)$.

Algoritmus nalezení maximálního toku v síti má některé další poměrně „nečekané“ použití pro řešení jiných důležitých problémů. Uvedeme dva. U obou těchto problémů vede řešení „hrubou silou“ na algoritmus jehož složitost není polynomiální a tyto algoritmy jsou tedy pro reálné situace bezcenné.

Problém bipartitního párování [bipartite matching]

Bipartitní graf $G = (U, V, E)$ je neorientovaný graf $(U \cup V, E)$, kde $U \cap V = \emptyset$, jehož uzly jsou dvojího. Hraný spojují pouze uzly z množiny U s uzly z množiny V . Žádná hrana nespojuje dva uzly U ani dva uzly z V . Párování je taková podmnožina $M \subseteq E$ hran grafu, pro kterou žádná hrana, která leží v M , nemá společný žádný vrchol s jinou hranou v M . Problém maximálního párování je problém nalézt pro daný graf párování o maximálním počtu prvků („uspokojených dvojic“). Typické použití si můžeme představit v sňatkové kanceláři, ale i při sestavování spolupracujících týmů zaměstnanců.

Problém lze převést na problém maximálního toku v síti přidáním dvou nových uzlů. Zdroje s a ponoru t . Zdroj s spojíme orientovanými hranami se všemi uzly v množině U . Všechny uzly v množině V spojíme orientovanými hranami s ponorem t . Všechny hrany v původním grafu G nahradíme orientovanými hranami vedoucími z množiny U do množiny V . Všem hranám v takto doplněném grafu, původním i doplněným, přiřadíme každé kapacitu rovnou 1. Maximální tok v této síti představuje řešení problému maximálního párování.

Problém minimální souvislosti grafu [Minimum connectivity problem]

Je dán souvislý neorientovaný graf $G = (V, E)$. Problém je kolik hran v tomto grafu lze vynechat, aby graf zůstal ještě souvislý. Tento problém má zřejmě použití v teorii spolehlivosti komunikačních a počítačových sítí.

Problém lze převést na problém maximálního toku v síti takto: Zaměníme každou neorientovanou hranu dvěma orientovanými hranami v obou směrech. Každé hraně přiřadíme kapacitu 1. Vybereme pevně uzel s jako zdroj toku a za ponor t budeme považovat postupně všechny uzly z množiny V , různé od s . Pro každý z těchto $m - 1$ (kde m je počet uzlů grafu G) problémů maximálního toku v síti najdeme maximální tok a nalezneme minimum všech takto zjištěných hodnot. Tak zjistíme minimální počet hran, po jejichž odstranění přestane být graf souvislý.

Problém minimální souvislosti grafu je tak převeden na řešení $n - 1$ problémů maximálního toku v grafu.

12. PETRIHO SÍTĚ

Co jsou Petriho sítě

Za "otce" Petriho sítě je pokládán německý matematik Prof. Dr. Carl Adam Petri napsal v období roků 1960 – 62 svou disertační práci s názvem „Kommunikation mit Automaten“, ve které se zabýval komunikací s automaty a právě zde definoval koncept tzv. Petriho sítě.

Petriho sítě jsou ve své podstatě modelovací nástroj (prostředek). V současnosti je mohou spojovat především s návrhem, analýzou a modelováním paralelních či distribuovaných systémů, zejména v technických oblastech lidské činnosti (např. automatizace v průmyslu, pomoc při návrhu hardwaru a softwaru počítačů apod.).

Základní názvosloví a charakteristika

Neoznačená Petriho síť je *orientovaný ohodnocený bipartitní* graf (Bayer, 2000). Skládá z těchto třech základních objektů :

1. **Místa (Places)**, graficky reprezentována kruhy, tvoří konečnou množinu míst Petriho sítě.
2. **Přechody (Transitions)**, graficky reprezentována obdélníky, která tvoří konečnou množinu přechodů Petriho sítě.
3. **Hrany (Arcs)**, graficky reprezentována šipkami, přívlastek *orientovaný* vyjadřuje skutečnost, že hrany grafu jsou orientované a přívlastek *ohodnocený* vyjadřuje skutečnost, že hranám jsou přiřazeny **Váhy (Weight)**. Váha udává násobnost (mohutnost) každé hrany (viz. obr.č. 1).

Slovo *bipartitní* vyjadřuje skutečnost, že vrcholy grafu mohou být prvky dvou množin (míst a přechodů), přičemž místa a přechody se vždy v průběhu cesty (grafu) střídají (viz. obr.č. 1).

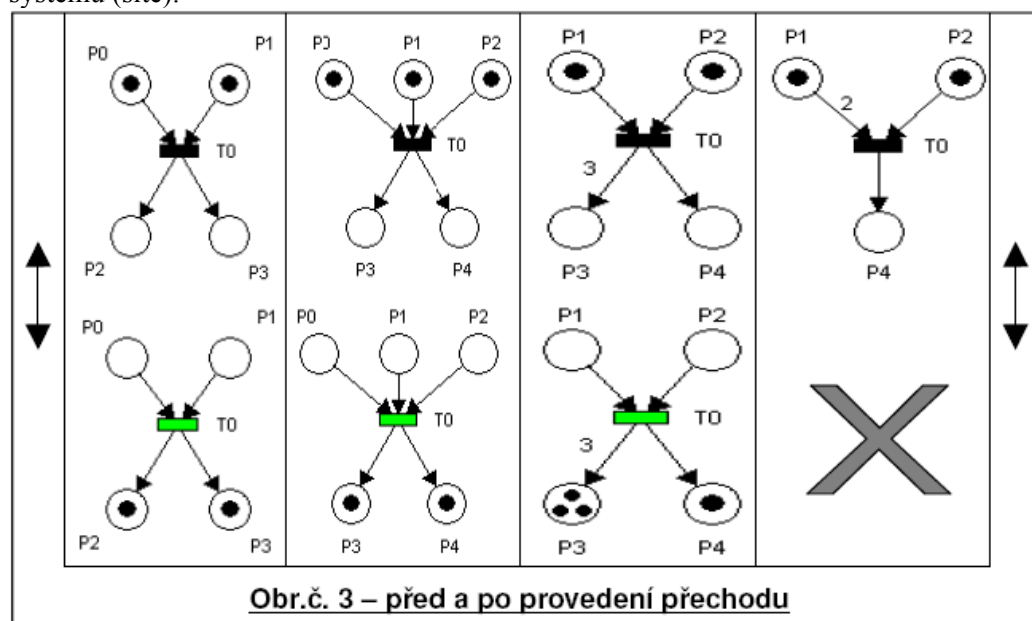
Označená Petriho síť vznikne umístěním **Značek (Marks, Tokens)** (viz. obr.č. 2) do míst neoznačené Petriho sítě [2]. Rozmístění značek v Petriho síti před první *aktivací* některého z přechodů se nazývá *počáteční značení* Petriho sítě.

Aktivace přechodu a kapacita místa v Petriho síti

Přechod může mít jedno nebo více vstupních a výstupních *míst*. **Místa** mohou obsahovat *značky*. Nyní definujeme za jakých podmínek dochází k přesunům *značek* v síti, tj. za jakých podmínek dochází k **Aktivaci přechodů**.

Přechod může být *aktivován* pouze tehdy, jestliže všechna vstupní *místa* obsahují počet značek **rovný nebo větší** než je **váha** hrany spojující vstupní místo a přechod.

Počet odebraných (vložených) *značek* přímo závisí na *váze* vstupních (výstupních) *hran* (viz. obr.č. 3). Pro zjednodušení si představme, že *značky* ve výstupních *místech* vznikají jakoby znovu a doba, která je nutná pro aktivaci přechodu, je rovna nule (přechod se aktivuje pokaždé za výše uvedených podmínek). Dostáváme se tak k vývoji (evoluci) celého systému (sítě).

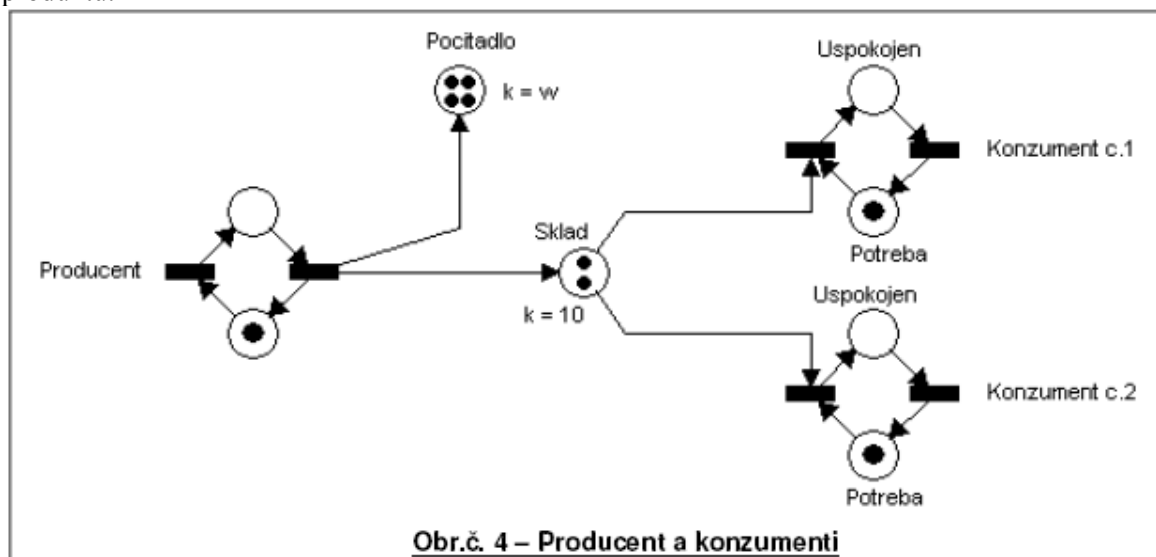


Z předchozího obrázku je zřejmé, že místa mohou obsahovat žádnou, jednu nebo více značek.

Existuje tedy další vlastnost (atribut), kterou mohou mít místa v Petriho síti, tím je **Kapacita** místa. Tu lze definovat následně: **Kapacita** místa je přirozené číslo nebo symbol Ω , ten značí nekonečnou kapacitu (značení nekonečna se často v praxi vůbec neuvádí). **Kapacita** místa se pak značí písmenem **k** (např. $k=100$).

Kapacitní omezení se hojně využívá při modelových situacích, kdy potřebujeme nutně odstranit nebezpečí přetečení (přeplnění), například fronty, vyrovnávací paměti atd. Pokud tedy při modelování využijeme kapacitní omezení, pak si v takovém případě musíme uvědomit, že je nutné přidat další podmínku pro aktivaci přechodu. Ta zní: „Přechod může být proveden (aktivován), jestliže nebude překročena **kapacita** výstupního místa [2].“

Příklad Petriho sítě s využitím omezení kapacity některých míst (viz. obr.č. 4) ukazuje vztah producent (produkuje) – konzumenti (nakupují, odebírají) s využitím vyrovnávací paměti (skladu, kde $k = 10$) a počítadla ($k = \Omega$) vyrobených produktů.



Petriho síť lze chápat jako automat, který definuje chování systému posloupnostmi událostí a odpovídajícími stavovými změnami systému. V případě konečných deterministických automatů byl stav systému jednoznačně určen prvkem množiny stavů. V případě Petriho sítě je okamžitý stav modelovaného systému určen určitými parciálními stavy spojenými s příslušnými místy sítě. Na otázku, jakou cestou je dosaženo určitého konkrétního parciálního (dílčího) stavu? Je odpověď, že k tomuto účelu slouží značení míst(a). Připomínáme, značení místa je celé nezáporné číslo, které bývá obvykle graficky vyznačeno počtem černých teček (značek), které jsou umístěny uvnitř příslušného místa (pokud by informace měla příliš velkou hodnotu, může být vyjádřena číselně).

Formální matematická definice Petriho sítě

Uvádím základní matematickou definici Petriho sítě. Ta definuje pouze základní prvky, vazby mezi nimi a zavádí slovní označení. Je do určité míry zjednodušující, protože se nijak nedotýká definicí značení, kapacity místa a nedefinuje ani váhu hrany v Petriho síti.

Definice 1. [1]

Uspořádanou trojici $N = (P, T, F)$ nazýváme sítí, jestliže

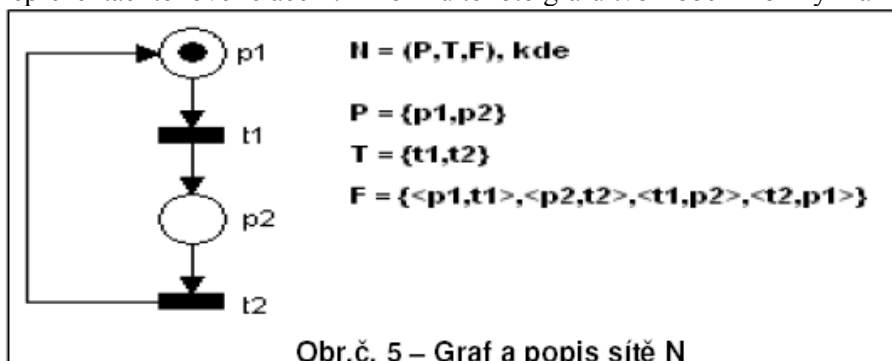
- (1) P a T jsou disjunktní množiny a
- (2) $F \subseteq (P \times T) \cup (T \times P)$ je binární relace.

P se nazývá množinou míst (Places) sítě N .

T se nazývá množinou přechodů (Transitions) sítě N .

F se nazývá tokovou relací (Flow relation) sítě N .

Jak již bylo řečeno v úvodu, grafem sítě je orientovaný (případně ohodnocený) bipartitní graf. Ten vzniká právě grafovou reprezentací tokové relace F . Množinu tohoto grafu tvoří obě množiny P a T , tedy $P \cup T$ (viz. obr.č. 5).

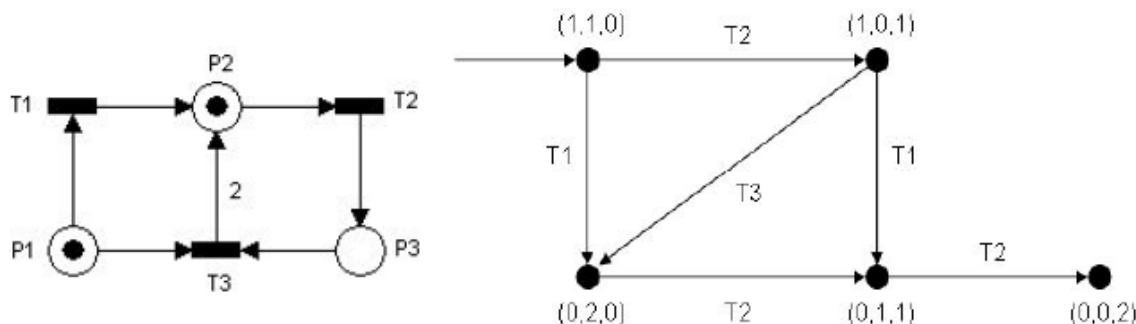


Stavový graf

Stavový graf je určitou abstrakcí Petriho sítě. Diagramem (zobrazením) metody je orientovaný ohodnocený graf. Šipka přicházející do počátečního vrcholu („odnikud“) je počátkem grafu. Mezi jednotlivými vrcholy jsou přechody vyjádřeny jako orientované ohodnocené hrany (resp. jde o uspořádanou trojici (m, t, m')).

Následuje příklad konstrukce stavového grafu.

Příklad:



Ze stavového grafu je dobře vidět, jaké značení (stavy) generuje tato Petriho síť a jaké přechody na tyto značení vedou (resp. nevedou).

Vlastnosti Petriho sítě

Každá Petriho síť má řadu vlastností, které lze popsat formálně a které vyjadřují podstatné typy chování modelovaného systému. Umožňují prokázat použitelnost a bezchybnost ještě před případnou praktickou realizací. Tyto vlastnosti jsou:

1. **Bezpečnost – bezpečné Petriho sítě**, zda je síť bezpečná může být definováno jak pro jednotlivá místa, tak pro celou síť. O místě říkáme, že je bezpečná právě tehdy, pokud pro všechna možná značení (stavy) počet značek v daném místě sítě nikdy nepřekročí hodnotu jedna. Neboli bezpečná Petriho síť je ohraničená pro $k = 1$.

2. **Ohraničenost – ohraničené Petriho sítě**, zda je Petriho síť ohraničená může být opět definováno buď pro jednotlivá místa nebo celou síť. Ohraničení je obecnější nežli první vlastnost. O místě říkáme, že je ohraničené pokud pro všechna možná značení počet značek v daném místě sítě nikdy nepřekročí hodnotu k (kapacitu daného místa). O celé síti říkáme, že je ohraničená pokud všechna místa v dané síti jsou bezpečná. Jinak řečeno, pokud síť

není ohraničená, počet značek roste v některém z jejích míst nad všechny meze a tím pádem stavový diagram roste také nad všechny meze, takový systém nelze realizovat konečným automatem (Bayer, 2000). Například místo, které v síti reprezentuje vyrovnávací paměť (buffer), by mělo mít ohraničenou kapacitu.

3. **Živá Petriho síť**, tato třetí vlastnost je velice důležitou pro aplikaci Petriho sítě. Skrývá v sobě koncept, který zaručí, že systém bude funkční kontinuálně (nepřetržitě). Tj., že nenastane situace, kterou v počítačových vědách označujeme jako uváznutí (deathlock). Síť je živá, pokud je pro všechna možná značení je aktivován přechod (přechod může být „kdykoli“ přeskočen), neustále tedy dochází k vývoji (evoluci) sítě (systému). Živá síť modeluje systém, ve kterém se nevyskytuje deathlock [2]. Uplatní se například při modelování komunikačních protokolů, operačních systémů atd.

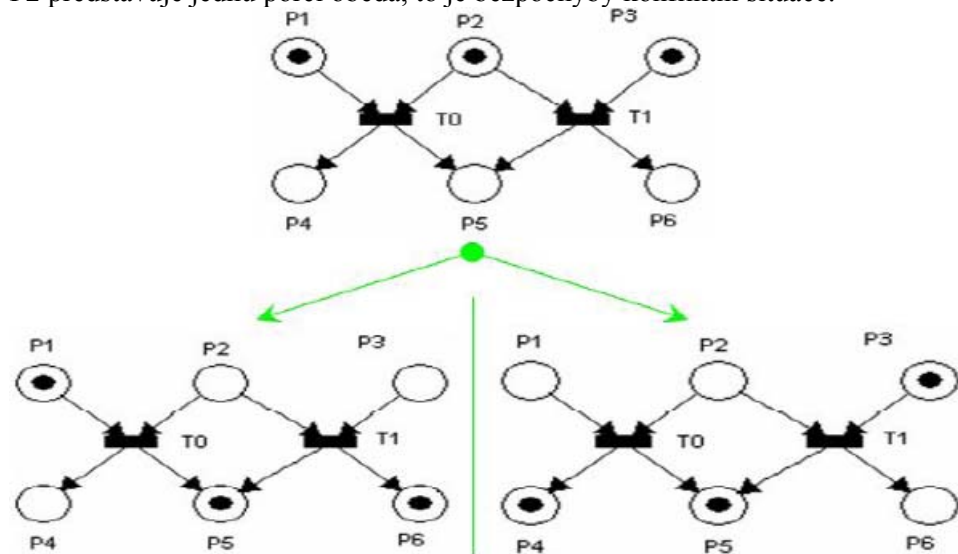
4. Reverzibilita reverzibilní

Petriho sítě, síť je reversibilní, jestliže pro libovolné dosažitelné značení M (nějaký možný stav) existuje aktivační sekvence S , která uvede síť do původního značení M_0 (počáteční značení).

Jednou z dalších důležitých vlastností Petriho sítě, na kterou by se nemělo zapomínat, je možnost modelování (a simulace) konfliktních situací (stavů). Právě o tom krátce pojednává následující odstavec.

Situace, kdy Petriho síť modeluje nějaký konflikt, se v odborné literatuře často nazývá tzv. *efektivním konfliktem* (Bayer, 2000). Je nutné jasně říci, že úkolem je konflikt modelovat, ale nikoli však řešit (například pomocí priority přechodů) [2].

Efektivní konflikt znamená, že se systém může vyvíjet více způsoby, následující příklad na obrázku č. 7 (podle [2]) nám jednu takovou konfliktní situaci pěkně ukazuje. Uvedený příklad se může vyvíjet dvěma různými způsoby ($M1=(1, 0, 0, 0, 1, 1)$ a $M2=(0, 0, 1, 1, 1, 0)$). Můžeme si například představit, že místa $P1$ a $P3$ představují hladové strážníky a místo $P2$ představuje jednu porci oběda, to je bezpochyby konfliktní situace.

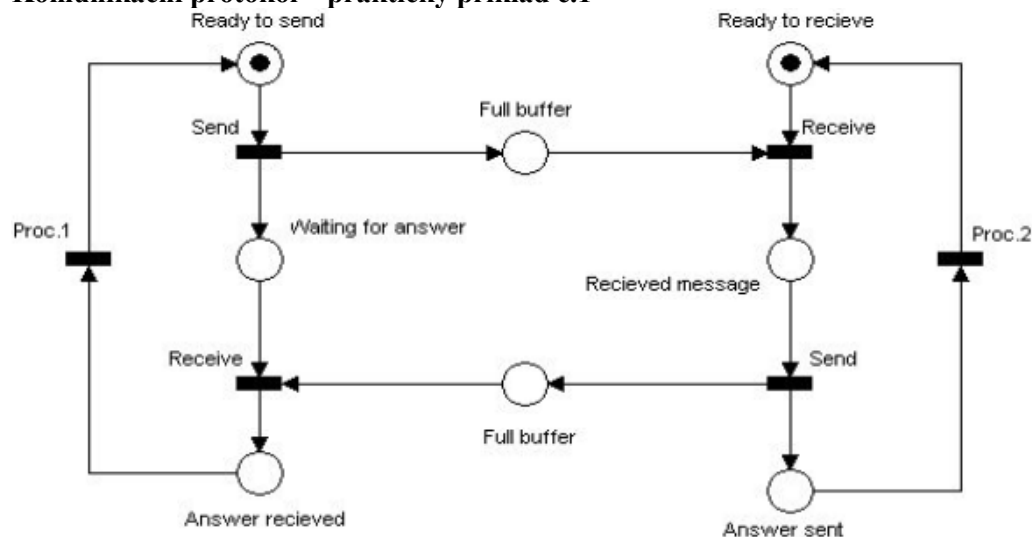


Obr.č. 7 – Efektivní konflikt v Petriho síti

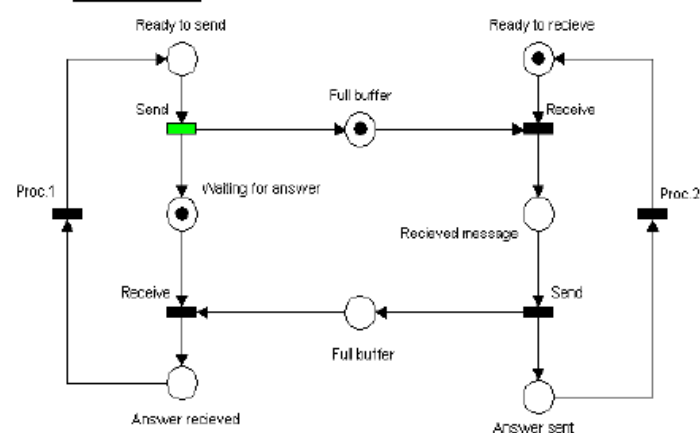
Na závěr snad jen dodáme, že Petriho síť má mimo jiné i za cíl simulovat konfliktní situace a tím

předcházet problémům, které se mohou vyskytnout v průběhu vývoje celého systému.
V závěru jsou ukázány dva praktické příklady Petriho sítě a jejich simulace.

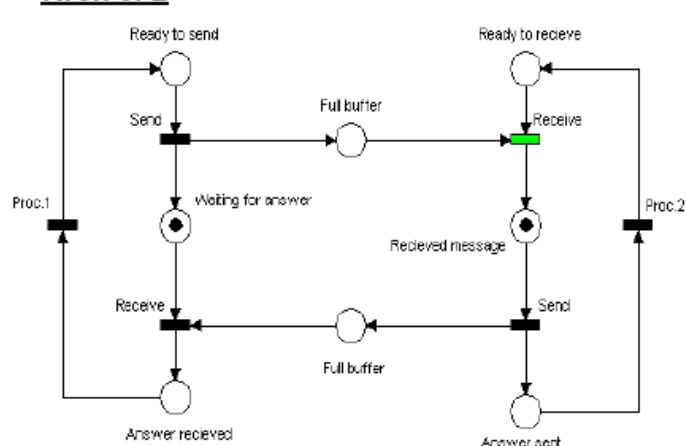
Komunikační protokol – praktický příklad č.1



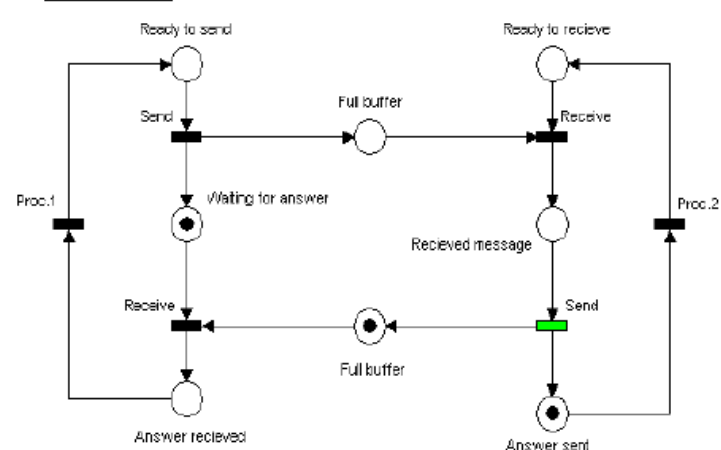
Krok č. 1



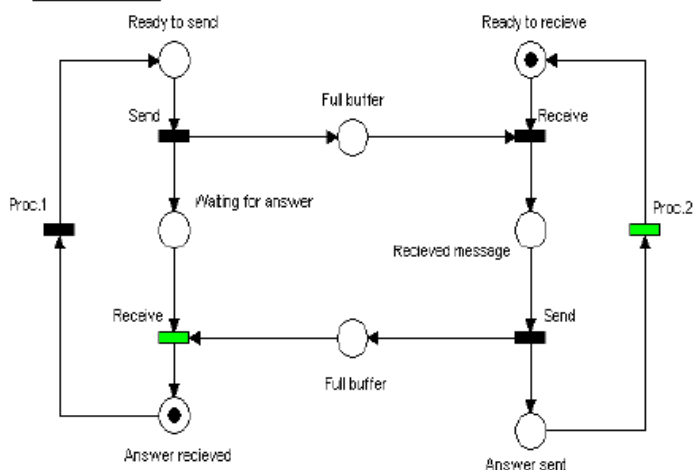
Krok č. 2



Krok č. 3

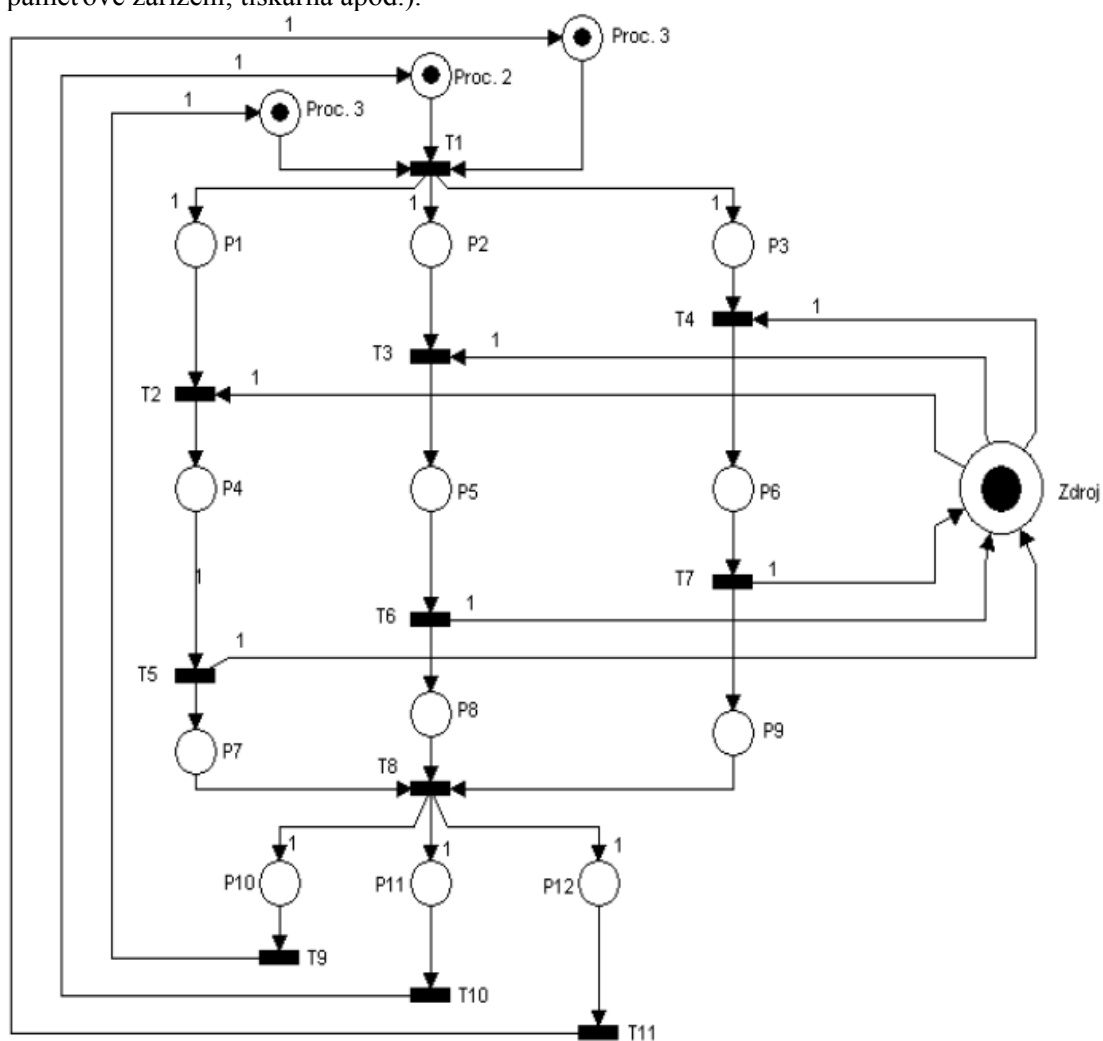


Krok č. 4



Synchronizované paralelní procesy – praktický příklad č.2

Příklad modelování třech paralelních synchronizovaných procesů, které se dělí o jeden společný zdroj (např. vnější paměťové zařízení, tiskárna apod.).



13. ALGORITMY VYHLEDÁVÁNÍ A DOPLŇOVÁNÍ

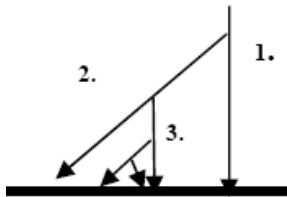
Vyhledání položky v neseřazeném souboru délky N – vždy složitost $\Omega(N)$, bez ohledu na organizaci

Seřazení může pomoci:

Organizace s přímým přístupem – pole $a_i \leq a_j$ pro $i \leq j$

Vyhledání umožňuje přístup půlením intervalu

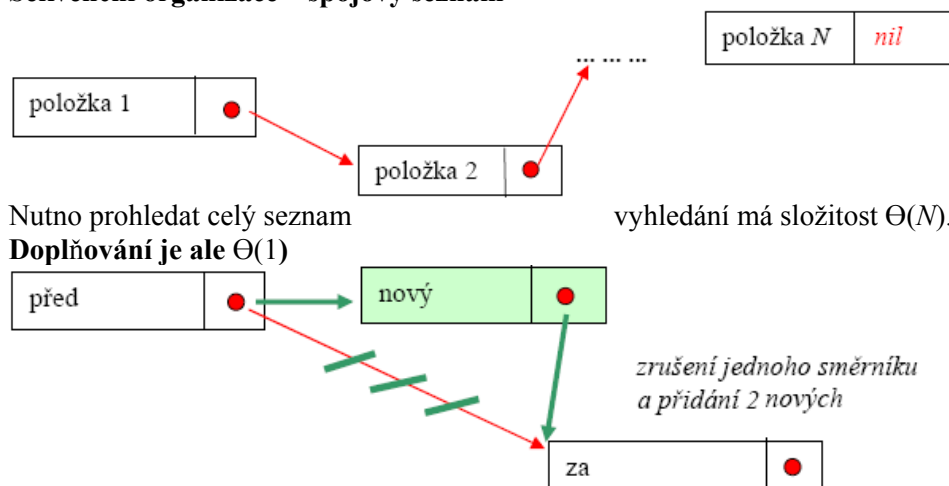
$\log_2(N) + 1$ kroků složitost $\Theta(\log(N))$



Problém je doplňování.

Nutno prvky posunout – složitost $\Theta(N)$

Sekvenční organizace – spojový seznam



Analogicky vynechání.

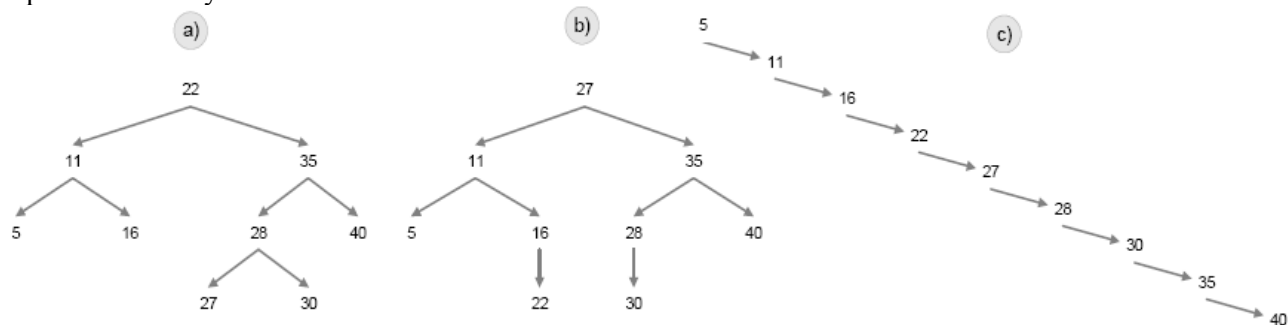
Proto snaha o lepší uložení dat, kombinující oba požadavky:

- Snadné vyhledávání
- Snadné doplňování a vypouštění

BINÁRNÍ STROM

Podmínka: Levý syn předchází před otcem a otec před pravým synem

Způsobů může být více.



Vyvážený strom (Dokonale vyvážený strom)

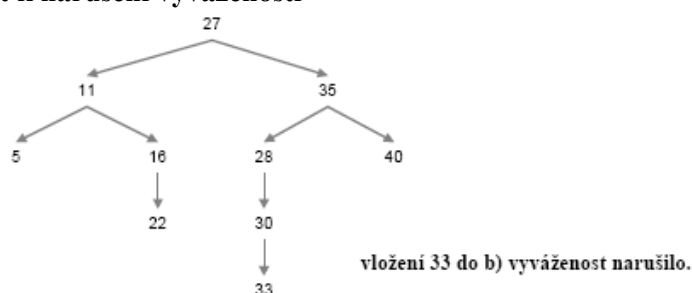
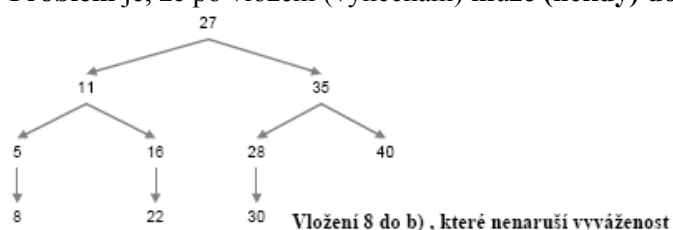
\equiv_{DEF} Počet uzlů v levém a pravém podstromu každého vrcholu se liší nejvýše o 1

Ze stromů a) b) c) v předchozím obrázku je vyvážený pouze strom b)

Umožňuje vyhledávání se složitostí složitosti $\Theta(\log(N))$

Vložení a vypuštění je složitosti $\Theta(1)$ – při užití směrnic pouze změna jediného směřníku

Problém je, že po vložení (vynechání) může (někdy) dojít k narušení vyváženosti



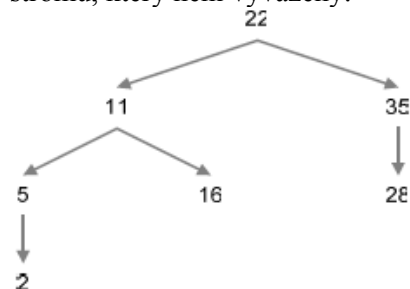
Rekonstrukce nevyváženého stromu na vyvážený je poměrně náročná.

Vyžaduje $\Theta(N^2)$ operací.

Proto se častěji užívají AVL-stromy (Adelson Veselský a Landis)

AVL- strom \equiv_{DEF} Pro každý vrchol se hloubka levého a pravého podstromu liší nejvýše o 1.

Tato podmínka je slabší než dokonalá vyváženost. Každý vyvážený strom je AVL- strom, ne naopak. Příklad AVL- stromu, který není vyvážený:



Minimální počty vrcholů, které obsahuje AVL-strom jsou dány Fibonacciho posloupností.

Proto se AVL-stromům říká též Fibonacciho stromy.

Vyhledávání v AVL stromech má stále složitost $\Theta(\log(N))$

Doplnění prvku může (nemusí) opět vlastnost AVL narušit.

Na rozdíl od vyvážených stromů je však **náprava mnohem jednodušší.**

Provede se tak zvanou rotací, která vyžaduje **přesměrování nejvýše 4 směrniců.** - $\Theta(1)$.

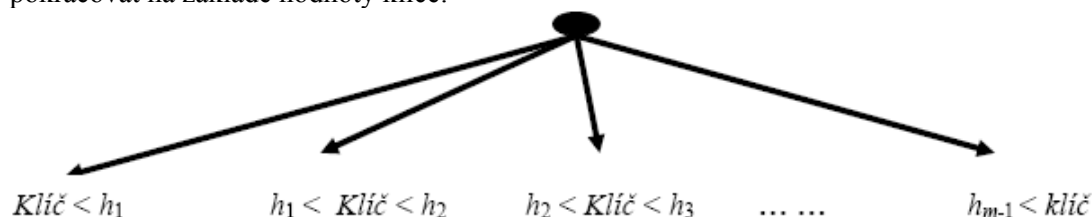
Složitost doplnění i vypuštění, včetně nápravy na AVL-strom je tedy $\Theta(\log(N))$.

Užívají se i obecnější vyhledávací stromy než binární

Položky ukládáme jako listy stromu. Vrcholy stromu, které nejsou listy označují skupiny položek, jejichž klíče jsou v určitém rozmezí.

(a, b) - STROM \equiv_{DEF} Každý vrchol, který není listem má aspoň a a nejvýše b synů.

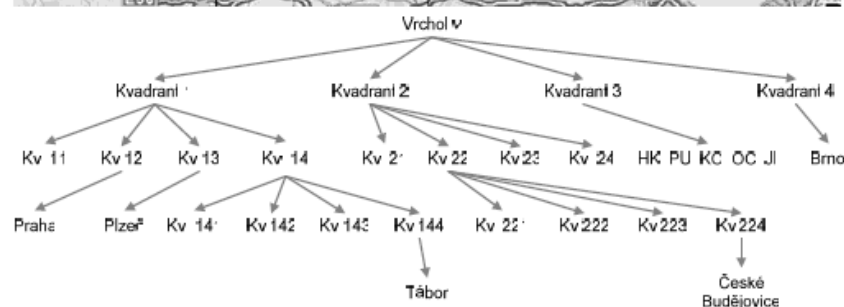
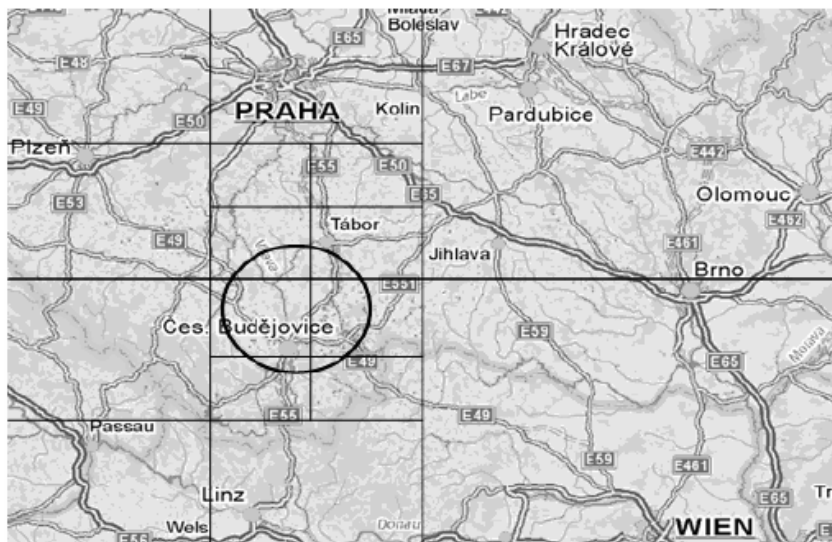
Má-li vrchol m následníků, je určeno $m - 1$ hraničních hodnot $h_1 < h_2 < \dots < h_{m-1}$, podle kterých se rozhodne jak pokračovat na základě hodnoty klíče:



Velmi užívané jsou (2, 3)-stromy

Vyhledávání a doplňování u nich má složitost $\Theta(\log(N))$

Pro vyhledávání objektů v rovině může být vhodný 4-strom (4, 4) - strom quadtree



14. ŘAZENÍ (SORT)

Řazení = uspořádávání – nevhodně „třídění“.

Řadíme podle hodnoty klíče (číslo nebo abecedně či jiný ordinární typ).

Musí jít o slabé uspořádání klíčů.

Vzestupné / Sestupné ... Teorie je analogická. Budeme uvažovat dále jen **vzestupné**.

Dáno N prvků: a_1, a_2, \dots, a_N

Jde o nalezení tak zvané **řadící permutace P** množiny čísel $\{1, 2, \dots, N\}$, aby $a_{P(1)} \leq a_{P(2)} \leq \dots \leq a_{P(N)}$ respektive jde-li o čísla $a_{P(1)} \leq a_{P(2)} \leq \dots \leq a_{P(N)}$.

Omezíme se pro jednoduchost nadále na označení preference \leq i když častěji řadíme celé záznamy podle jedné nebo více číselných položek. *Užívat \neg by bylo „čistší“.*

Někdy je vhodné aby při řazení u prvků se stejnou hodnotou klíče nedocházelo k přehazování – tak zvané **stabilní řazení**.

Vhodné chceme-li stávající seřazení zachovat u prvků se stejnou hodnotou nového řadícího klíče.

Dva základní postupy:

ŘAZENÍ Adresové – na základě klíče se určí umístění v seřazené posloupnosti. Varianta je tak zvané přihrádkové řazení – na základě klíče se zatřídí do přihrádky, poté se přihrádky znovu roztřídí na přihrádky nižší úrovně... + *Je rychlé. Lze se přiblížit lineární složitosti.* – *Musíme znát údaje o rozložení klíčů. Jinak plýtvá místem.*

Asociativní – klíče navzájem porovnává a položky přehazuje + *Je univerzální. Umí řadit „na místě“ (in situ).*
– *Minimální teoretická složitost je $\Theta(n \cdot \log(n))$, jednoduché algoritmy mají složitost $\Theta(n^2)$.*

Hybridní – kombinuje oba přístupy. Adresově roztřídí do přihrádek, poté uvnitř přihrádek asociativně dokončí řazení.

ASOCIATIVNÍ ŘAZENÍ

Hodnotí se počtem potřebných porovnání klíčů.

Metoda hrubé síly: Generování všech permutací postupně a kontrola zda je splněna podmínka seřazení.

PROLOG:

```
% procedura del vynechá postupně všechny prvky seznamy.
del(X, [X|T], T). % vynech hlavu
del(X, [H|T], [H|NoveT]) :- del(X, T, NoveT). %nebo něco z těla
% procedura permut postupně generuje všechny permutace seznamu tak, že
% vynechá postupně všechny prvky, dá je na prvé místo jako hlavu a ostatní
% prvky těla libovolně permutuje.
permut([], []). % V prázdném seznamu není co přehazovat
permut(L, [X|PT]) :- del(X, L, T), permut(T, PT).
% Procedura serazen testuje, zda je seznam čísel seřazen vzestupně.
serazen([]).
serazen([_]). % Prázdný a jednoprvkový určitě seřazen jr.
serazen([X|Y|T]) :- X <= Y, serazen(T). % Jinak je třeba aby
% první prvek (hlava) byl nejvýše roven druhému a celý zbytek, počínaje
% druhým prvkem byl rovněž seřazen.
hloupy_sort(List, SList) :- permut(List, SList), serazen(SList).
% Budeme tak dlouho seznam permutovat, až nám vyjde seřazený.
```

Složitost je $\Theta(N!)$ \Rightarrow nepoužitelné.

Pro asociativní řazení se užívá **princip „rozděl a panuj“** – Divide et impera – *Divide and conquer*.

OBEZNÉ SCHÉMA POSTUPU ROZDĚL A PANUJ:

- Úloha se rozdělí na dvě části.
- V každé části se úloha vyřeší samostatně (případně rekurzivním voláním téže procedury).
- Z obou částí se vytvoří společné řešení.

? \rightarrow ?1 - Jak úlohu rozdělit? *Analytická část*

\rightarrow ?2 - Jak dát řešení dohromady? *Syntetická část*

ROZDĚLIT JAK ?	NEVYROVNANĚ 1 : N - 1	VYROVNANĚ Pokud možno na poloviny
KDE JE PRÁCE ?		
V ANALYTICKÉ ČÁSTI	Řazení výběrem <i>selectsort, bubblesort, $\Theta(N^2)$</i> heapsort (halda)	Řazení rozdělčováním quicksort
V SYNTETICKÉ ČÁSTI	Řazení vkládáním <i>Insertsort, $\Theta(N^2)$</i> Úprava s klesajícím krokem Shellovo řazení	Řazení sléváním (zatřídčováním) mergesort

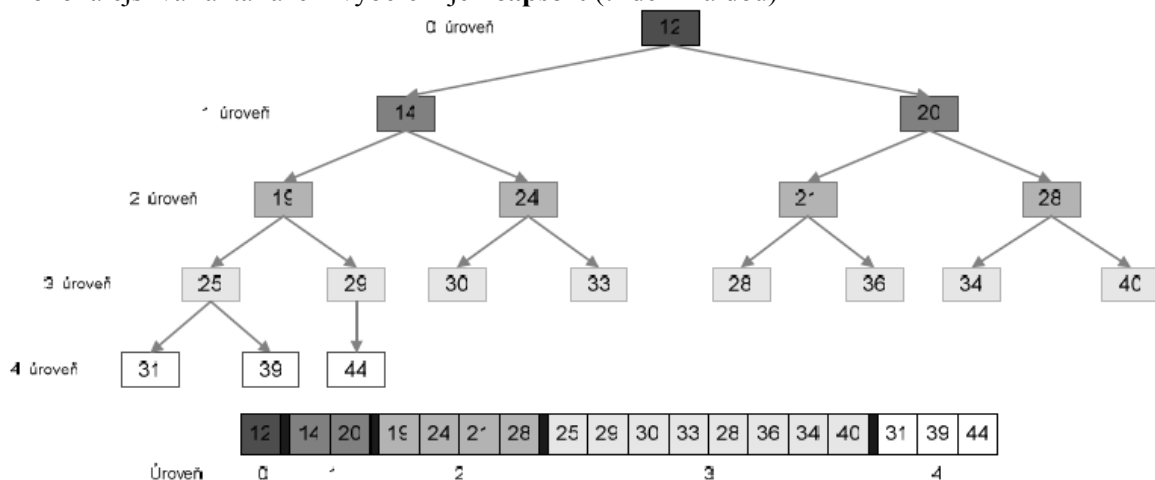
Řazení výběrem:

Selectsort : vyber minimum, pak minimum ze zbytku, ...

$(N-1) + (N-2) + \dots + 1 = N \cdot (N-1) / 2$ porovnání Složitost $\Theta(N^2)$.

Bubblesort (bublínkové řazení) : v pesimistickém i průměrném případě potřebuje též $\Theta(N^2)$ porovnání. Výhodnější je pouze v případě „přetříděných“ dat (s málo inverzemi).

Dokonalejší varianta řazení výběrem je **heapsort** (třídění haldou)



Řazení probíhá takto:

1. Vytvoří se haldy – složitost $\Theta(N \cdot \log(N))$

2. Odebere se vrchol, nahradí se novým prvkem a ten se „nechá propadnout do haldy“, tedy haldy se rekonstruuje. Na rekonstrukci haldy je třeba nejvýše $\log(N)$ porovnání, kde N je počet prvků v haldě. To umožňuje seřadit i větší objem dat než je v operační paměti. Složitost jednoho kroku je $\Theta(\log(N))$.

Celková složitost je $\Theta(N \cdot \log(N))$

Řazení vkládáním:

Insertsort: Procházíme sekvenčně již seřazená data a zařazujeme nové prvky (cykl nebo rekurze). Opět třeba $(N-1) + (N-2) + \dots + 1 = N \cdot (N-1) / 2$ porovnání.

⇒ **Složitost $\Theta(N^2)$.**

Vkládání s klesajícím krokem

Při vhodné organizaci dat, např. vyvážený binární strom nebo AVL-strom, lze místo kam prvek zařadit nalézt za $\log(N)$ porovnání klíčů. Složitost je pak $\Theta(N \cdot \log(N))$.

Oblíbená varianta na tomto principu, upravená pro práci s polem je známa pod názvem **Shellovo řazení – shellsort**.

Řazení rozdělčováním - quicksort

Princip:

1. Vybere se prvek – **pivot**

2. Data se rozdělí na ta s klíčem – s menším nebo rovným klíči pivotu – **M**
– s klíčem větším než pivot – **V**.

Výsledné pořadí je **M, pivot, V**

3. Krok 1 se opakuje pokud nejsou části prázdné nebo jednoprvkové.

Časová složitost jednoho rozdělení podle jednoho pivotu je lineární – $\Theta(N)$

Pokud jsou vždy množiny **M** a **V** „skoro stejně“ velké je třeba $\Theta(\log(N))$ kroků.

Pokud je jedna z množin **M** a **V** vždy prázdná, je třeba N kroků.

⇒ **Pesimistická složitost je $\Theta(N^2)$.** Tento nepříznivý případ nastane, jsou-li zadaná data již seřazena nebo seřazena opačně.

Průměrná složitost je však $\Theta(N \cdot \log(N))$. Jde o velmi rychlý a oblíbený algoritmus řazení.

Důležité je zabránění nepříznivého případu.

Zajištěno, pokud se za pivota zvolí **medián**. Medián se však hledá obtížně.

Proto se místo mediánu volí často **aritmetický průměr** a rozdělení je podle něj.

Případně se vybere náhodně několik prvků a pivot je medián či průměr z nich.

(Práce průměrem není vždy 100 % spolehlivá – průměr se může od mediánu dost lišit.)

Řazení sléváním

Založeno na **slévání monotonií** = vytváření seřazené posloupnosti z dvou (více) seřazených.

Princip slévání dvou posloupností:

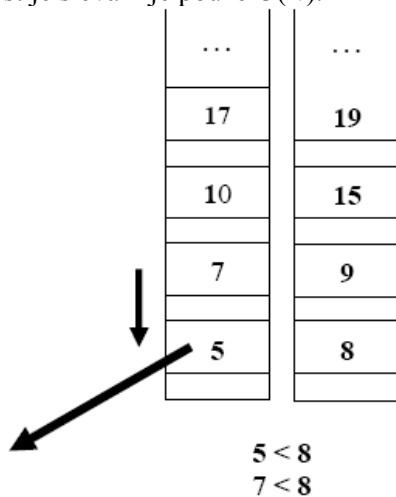
Porovnají se vždy oba nejmenší prvky v obou posloupnostech.

Menší (jsou-li si rovné tak libovolný) se vybere, zařadí a ukazatel v ní se posune.

To se opakuje, dokud se jedna z posloupností nevyprázdní.

Jakmile k tomu dojde, zařadí se celý zbytek druhé na závěr.

Složitost je slévání je pouze $\Theta(N)$.



5, 7, 9, 10, 15, 17, ...

...

Princip řazení sléváním:

1. Je-li posloupnost jednoprvková, končí. Jinak rozděl posloupnost na 2 stejně dlouhé nebo lišící se rozměrem o 1.
2. Obě posloupnosti seřaď rekurzivně touto procedurou.
3. Slej obě seřazené posloupnosti v jednu.

Pro realizaci algoritmu je potřeba $\log(N)$ kroků složitosti $\Theta(N)$. \Rightarrow **Složitost celku je $\Theta(N \cdot \log(N))$.**

Nevýhoda: Algoritmus nepracuje in situ (je zapotřebí dvojnásobná paměť).

15. LINEÁRNÍ ALGEBRA

Maticová algebra (časté v ekonomických a plánovacích aplikacích)

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & K & a_{1,n} \\ a_{2,1} & a_{2,2} & K & a_{2,n} \\ K & & & \\ a_{m,1} & a_{m,2} & K & a_{m,n} \end{pmatrix} = \left(a_{i,j} \right)_{\substack{j=1,K,n \\ i=1,K,m}} = (a_{i,j})$$

Sčítání matic (analogicky odčítání) $C = A + B \Leftrightarrow_{\text{DEF}} a_{ij} + b_{ij} = c_{ij}$

Deterministická i nedeterministická složitost je:

Lineární vzhledem k počtu prvků matice.

U čtvercových matic **kvadratická vzhledem k řádu matice.**

Nelze zlepšit.

Násobení matic

Součin matic $A \cdot B$ je definován tehdy a jenom tehdy, je-li první matice typu (m, n) a druhá typu (n, p) , tedy když počet sloupců první matice je též jako počet řádků druhé matice.

Výsledný součin je matice (m, p) a pro prvky součinu $C = (c_{ij})$ platí: $c_{i,j} = \sum_{k=1}^n a_{i,k} \cdot b_{k,j}$

Každý prvek součinu je tedy skalárním součinem $r_i(A) \cdot c_j(B)$ vektorů $r_i(A)$ a $c_j(B)$. Skalární součin lze zřejmě vypočítat v čase $\Theta(n)$. Časová složitost násobení matic je tedy $\Theta(m \cdot n \cdot p)$.

Čtvercové matice typu (n, n) , kterým říkáme také matice řádu n , lze tedy přímo podle definice násobit s časovou složitostí $\Theta(n^3)$, ovšem pokud za míru rozsahu vstupu algoritmu vezmeme řád n . Pokud bychom za rozměr vstupu V brali skutečný počet je složitost $\Theta(n^{3/2})$.

Pokus vylepši složitost užitím rozdělení a panuj:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \cdot \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} A \cdot E + B \cdot G & A \cdot F + B \cdot H \\ C \cdot E + D \cdot G & C \cdot F + D \cdot H \end{pmatrix}$$

nevede k cíli. Je třeba 8 násobení matic polovičního řádu. $\log_2 8 = 3 \Rightarrow$ složitost vzhledem k řádu matice je opět $\Theta(n^3)$.

Strassen:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \cdot \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} S_1 + S_2 - S_4 + S_6 & S_4 + S_5 \\ S_6 + S_7 & S_2 - S_3 + S_5 - S_7 \end{pmatrix}, \text{ kde } S_1 = (B - D) \cdot (G + H), S_2 = (A + D) \cdot (E + H), S_3 =$$

$$(A - C) \cdot (E + F), S_4 = H \cdot (A + B), S_5 = A \cdot (F - H), S_6 = D \cdot (G - E), S_7 = E \cdot (C + D).$$

Pro výpočet součinu potřebuje jen 7 násobení matic (na úkor 8 sčítání navíc).

Algoritmus má složitost $\Theta(n^{\log_2 7}) = B\Theta(N^{2,81\dots})$. Tedy lepší!

(vylepšit lze až při rozkladu na 70 bloků. Zatím nejlepší výsledek, získaný daleko složitějším postupem je $\Theta(n^{2,376\dots})$).

Výhoda těchto algoritmů se projevuje až u extrémně rozsáhlých matic.

Soustavy lineárních rovnic

$$a_{1,1} \cdot x_1 + a_{1,2} \cdot x_2 + \dots + a_{1,n} \cdot x_n = b_1$$

$$a_{1,2} \cdot x_1 + a_{2,2} \cdot x_2 + \dots + a_{2,n} \cdot x_n = b_2$$

...

$$a_{1,n} \cdot x_1 + a_{n,2} \cdot x_2 + \dots + a_{n,n} \cdot x_n = b_n$$

Lze v maticovém vyjádření formulovat jako problém pro danou čtvercovou matici $A = (a_{i,j})_{i,j=1,\dots,n}$ a vektor $b = (b_1, b_2, \dots$

$b_n)$ nalézt vektor $x = (x_1, x_2, \dots, x_n)$ takový, že $A \cdot x = b$, kde vektor b pravých stran a hledaný vektor neznámých jsou považovány za matice řádu $(n, 1)$ o n řádcích a jediném sloupci („sloupcové“ vektory).

Výpočet pomocí determinantů (Crammerovo pravidlo) má složitost $n!$ \Rightarrow je nepoužitelný.

Eliminační metoda řešení:

Transformuje matici A elementárními úpravami na jednotkovou. Užívá se:

- Vynásobení libovolného řádku matice nenulovým číslem.
- Přičtení násobku libovolného řádku k jinému řádku.
- Záměna pořadí (prohození) libovolných dvou řádků.

Tyto operace jsou všechny složitosti $\Theta(N)$. Pro vyloučení jedné proměnné je třeba je provést se zbylými $N - 1$ řádky. To je potřeba pro každou z N proměnných.

\Rightarrow **složitost eliminace je $\Theta(N^3)$, kubická vzhledem k řádu matice.**

Vyhledání přesného řešení rychlejším algoritmem možné není. Výpočet lze však často urychlit **iteračními algoritmy**, kterými se lze k přesnému řešení libovolně přiblížit.

Jeden krok iterace má časovou složitost obvykle $\Theta(N^2)$, je totiž nutné vektor, který je odhadem řešení násobit celou maticí.

Zda je iterační metoda výhodnější záleží na tom, zda se řešení přiblížíme dostatečně v počtu kroků, který s řádem matice roste výrazně pomaleji než lineárně. Aby bylo dosaženo výsledné časové složitosti výhodnější než $\Theta(N^3)$.

Nevýhodou iteračních metod může být, že pro jejich konvergenci musí matice splňovat některé dodatečné podmínky. Ty nemusí být splněny vždy. Metoda pak nemusí konvergovat.

Na stejném principu se určuje inverzní matice.

Aplikuje se eliminace na matici

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & K & a_{1,n} & 1 & 0 & K & 0 \\ a_{2,1} & a_{2,2} & K & a_{2,n} & 0 & 1 & K & 0 \\ K & & & & & & & \\ a_{m,1} & a_{m,2} & K & a_{m,n} & 0 & 0 & K & 1 \end{pmatrix}$$

Po eliminaci získáme matici (E, A^{-1}) .

\Rightarrow **složitost inverze je $\Theta(N^3)$, kubická vzhledem k řádu matice.**

Výpočet pomocí vzorce s determinanty je opět nepoužitelný. Má složitost $n!$.

Úlohy lineárního programování

Je speciální variantou obecnější **optimalizační úlohy**:

Obecná optimalizační úloha:

Nalézt maximum (minimum) dané funkce při daných omezeních (na dané množině).

Omezení jsou určena splněním daných M predikátů (podmínek).

U lineárního programování je oborem hodnot n -rozměrný vektorový prostor Θ_n :

Veškeré omezení jsou určena lineárními neostrými nerovnicemi nebo rovnicemi.

Tedy podmínkami typu:

$$a_1x_1 + a_2x_2 + \dots + a_nx_n \triangleq b,$$

kde \triangleq nahrazuje některé ze znamének \leq , \geq , nebo $=$.

Kriteriální (účelová) funkce, jejíž extrém (pro určitost maximum) hledáme je rovněž lineární

$$f(x_1, x_2, \dots, x_n) = c_1x_1 + c_2x_2 + \dots + c_nx_n.$$

Množinou přípustných řešení (vyhovujících všem omezením) je jak známo vždy konvexní (omezený nebo neomezený) polyedr v Θ_n , případně prázdná množina.

Z teorie je známo, že pokud má tato optimalizační úloha lineárního programování řešení, potom **účelová funkce nabývá svého maxima v jednom z vrcholů tohoto polyedru.**

Simplexová metoda řešení spočívá v nalezení jednoho z vrcholů polyedru a postupném přechodu k dalším vrcholům v kterých má účelová funkce vyšší hodnotu.

Jestliže platí:

- Všechna omezení mají tvar lineárních nerovností se znakem nerovnosti \leq ,
- všechny pravé strany nerovností b_j jsou kladná reálná čísla,
- mezi omezeními jsou podmínky $x_i \geq 0$ pro všechna $i = 1, 2, \dots, n$,

potom je jedním z takových vrcholů vždy bod $(0, 0, \dots, 0)$.

Po nalezení vrcholu zkoumáme všechny jeho sousedy a přejdeme do toho z nich, kde je hodnota největší. Pokud takový již takový vrchol není, našli jsme maximum.

(To ovšem platí jen za předpokladu, že jde o lineární problém, ne pro obecnou optimalizační úlohu. Tam může být situace složitější.)

Pokud má úloha n proměnných a M podmínek (včetně podmínek $x_i \geq 0$), je jistě $M > n$.

Každý vrchol polyedru je průsečíkem n podmínek vybraných z M daných. Naopak to sice neplatí, ale lepší obecně platný odhad počtu vrcholů polynomu nemáme.

Musíme tedy počítat s tolika vrcholy, kolik je možných kombinací jak vybrat n prvků z M daných. To je $\text{Comb}(M, n) = M! / ((n!) \cdot (M-n)!)$.

Tato funkce vzhledem k M roste rychleji než jakákoliv mocnina M .

V pesimistickém případě tedy algoritmus simplexové metody není polynomiálně složitý.

V praktických případech však většinou určíme optimum v přijatelném čase.

Úloha celočíselného lineárního programování je NP-úplný problém.

Pro nelineární programování (konvexní p.) se užívají většinou přibližné (gradientní) postupy.

16. ALGORITMY ZALOŽENÉ NA PRINCIPU HOROLEZCE, HLADOVÉ ALGORITMY

Tento rozsáhlý a různorodý soubor algoritmů je založen *zhruba* na myšlence provést v každém okamžiku takovou akci, která zajišťuje co největší okamžitou (lokální) výhodu.

Obecně se užívá pro řešení optimalizačních úloh.

Obecná optimalizační úloha:

Nalézt maximum (minimum) dané funkce při daných omezeních (na dané množině). Pro určitost v dalším hledáme **max**. Pro **min** je vše obdobné.

Postup může ale nemusí vést k cíli.

Vždy je nezbytné ověřit (dokázat), že daná varianta postupu v daném speciálním případě nalezne opravdu (globální) řešení úlohy.

Příklad:

Úloha výběru maximálního počtu aktivit ze zadaných částečně se překrývajících aktivit.

Dáno M aktivit, každá má svůj začátek a konec.

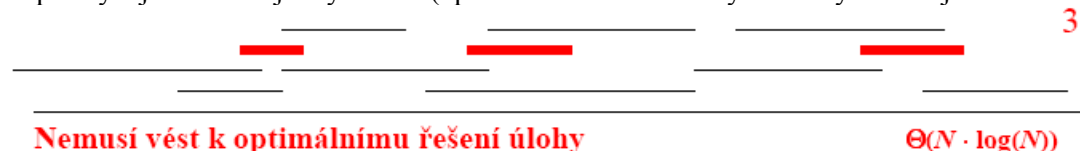
Trvají od z_j do k_j , zaplní uzavřené intervaly $\langle z_j, k_j \rangle, j = 1, \dots, M$.

Úkol: Vybrat jich co nejvíce aby se nepřekrývaly (i konec jedné musí předcházet začátku další)

Řešení hrubou silou: Postupně pro $N = 1, 2, 3, \dots$ vyšetřovat všechny podmnožiny aktivit o N prvcích a zjišťovat, zda se nepřekrývají. Poslední úspěšný výběr je řešením

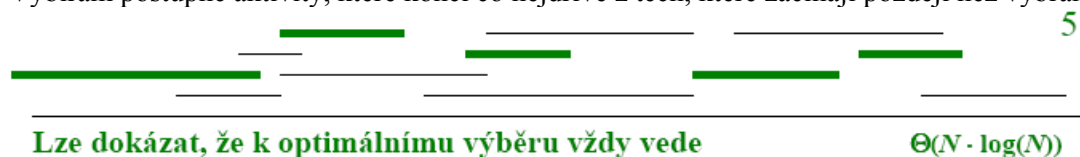
Složitost řešení hrubou silou není polynomiální, ale v pesimistickém případě exponenciální.

VARIANTA ŘEŠENÍ ÚLOHY č. 1: Budeme vybírat postupně vždy ty nejkratší z dosud nevybraných, které se nepřekrývají s žádnou již vybranou (optimalizační úvaha : aby nám zbylo co nejvíce volného času pro výběr dalších.



VARIANTA ŘEŠENÍ ÚLOHY č. 2:

Vybírám postupně aktivity, které končí co nejdříve z těch, které začínají později než vybrané.



Jiný příklad na to, že hladový princip může zklamat

NP-úplná ÚLOHA O BATOHU:

Dáno N usprádaných dvojic kladných čísel $(v_1, c_1), (v_2, c_2), \dots, (v_N, c_N)$ a kladná číslo V . Úkol je vybrat takovou podmnožinu B množiny $\{1, 2, \dots, N\}$, aby byl součet $\sum_{j \in B} c_j$ maximální při zachování omezení $\sum_{j \in B} v_j < V$.

Jak v batohu do kterého se vejde jen V odnést předměty co nejvyšší ceny, jsou-li jednotlivé předměty nedělitelné?

Pro úlohu není znám efektivnější algoritmus, než prozkoumat všech 2^N podmnožin množiny $\{1, 2, \dots, N\}$ (všechny možné výběry, které v batohu unesu a vybrat nejvýhodnější).

Je dokázáno, že problém je NP-úplný. Kdybychom znali polynomiální algoritmus jeho řešení, uměli bychom na tento problém transformovat v polynomiálním čase a též v polynomiálním čase vyřešit všechny nedeterministicky polynomiální (tedy v polynomiálním čase kontrolovatelné a využitelné) problémy.

Navrháme dvě heuristiky založené na hladovém principu:

H1 Vybíráme postupně vždy ty nejlcennější dosud neumístěné předměty.

H2 Vybíráme postupně předměty, které mají nejlepší „měrnou hodnotu“, poměr ceny k objemu, tedy c_j/v_j .

Uvedme příklad:

Unesu 25 kg - V

Mohu si vybrat z:

A: Cena 100 peněz, váží 12 kg 8,33 peněz za kg

B: Cena 100 peněz, váží 12 kg 8,33 peněz za kg

C: Cena 180 peněz, váží 15 kg 12,0 peněz za kg

D: Cena 190 peněz, váží 20 kg 9,5 peněz za kg

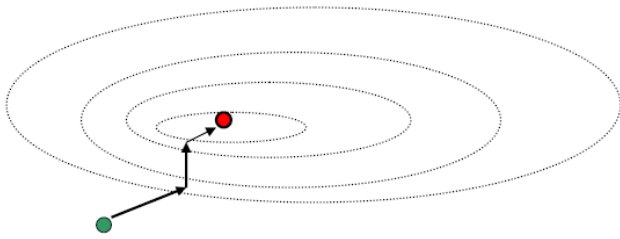
H1 - velí dát do batohu D. Pakuj nic dalšího neunesu. Odnesu si věci za 190 peněz

H2 - dá ještě horší výsledek. Odnesu si jen C za 180 peněz.

Kdybych vzal A a B, byl bych na tom lépe. Měl bych zisk 200 peněz.

Spojité případy – GRADIENTNÍ METODY (princip horolezce)

Hledá se maximum hladké (spojité parciální derivace) funkce N proměnných $f(x_1, x_2, \dots, x_N)$ na uzavřené podmnožině \mathbb{R}^N

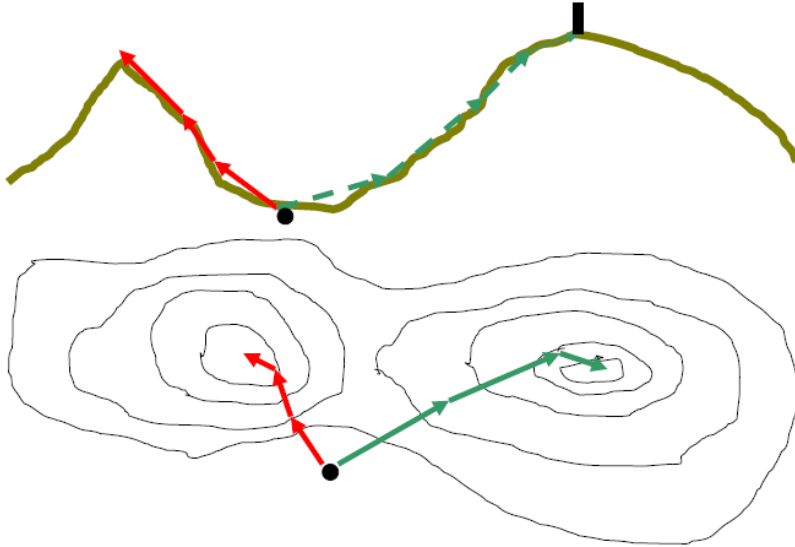


Pokračujeme vždy směrem největšího spádu, v postupných krocích $j = 0, 1, 2, \dots$.

Ten je určen vektorem $\text{grad } f(x) = \left(\frac{\partial f(x)}{\partial x_1}, \frac{\partial f(x)}{\partial x_2}, \dots, \frac{\partial f(x)}{\partial x_N} \right)$

Problémem může být volba délky kroku.

Postup může skončit v lokálním maximu, které není globálním maximem.



17. OPTIMALIZACE NA GRAFECH (HLADOVÉ ALGORITMY)

Algoritmy založené na principu hladovce, aplikovaný na **diskrétní úlohy** se nazývají často **hladové algoritmy**. V aplikacích z oblasti ekonomie a řízení jsou typickými úlohami tohoto typu optimalizační problémy na grafech. Uvedeme výběr několika typických základních úloh tohoto typu.

Každá z úloh má řadu modifikací.

Problém optimální cesty mezi vrcholy grafu

Je dán **neorientovaný nebo orientovaný souvislý graf**, **hranově ohodnocený** nezápornými čísly a **dva jeho vrcholy** (u orientovaného grafu je určeno, který je počáteční a který cílový). **Úkol je nalézt** (orientovanou) **cestu** mezi nimi, takovou, **aby součet ohodnocení hran na této cestě byl minimální** (ze všech cest, které tyto vrcholy spojují).

Pod ohodnocením hran si lze představit:

vzdálenost, (hledáme nejkratší spoj),

čas potřebný na transport, (hledáme nejrychlejší spoj),

cenu přepravy, (hledáme nejlevnější spoj),

apod.

Lze si představit že ohodnocení znamená například i **spolehlivost úspěšného přenosu dat**. V tom případě ovšem při hledání nejspolehlivější cesty nelze užít sčítání ohodnocení hran, ale, je-li x_j procentuální pravděpodobnost poruchy při přenosu či přepravy hranou h_j , bude procentuální pravděpodobnost chybného přenosu po cestě $v_1, h_1, v_2, h_2, \dots, h_{n-1}, v_n$

rovna $100 \cdot \prod_{k=1}^{n-1} (x_k / 100)$. Tuto hodnotu musíme minimalizovat, abychom získali nejspolehlivější cestu.



Při výpočtu ohodnocení cesty tedy musíme operaci sčítání čísel y v tomto případě nahradit jinou operací skládání ohodnocení hran.

V případě, kdy ohodnocení představuje **omezení kapacity jednotlivých hran** bychom při hledání (jediné) přenosové cesty s největší kapacitou místo součtu museli uvažovat minimum.

Řešení hrubou silou by spočívalo ve výpočtu ohodnocení všech cest od začátku do cíle.

Vybrat všechny podmnožiny množiny M vrcholů, vyloučit ty, které netvoří cestu a vybrat cestu s minimálním ohodnocením má složitost 2^M , vyšetřovat všechny permutace vrcholů nejméně $\Theta(M!)$. Obé je **nepoužitelné**. **Dijkstrův algoritmus** (na hladovém principu): Optimální cesty se hledají nejprve v blízkém okolí výchozího vrcholu. Toto okolí se postupně rozšiřuje, až se dostaneme do cíle.


Každý vrchol má dočasné a trvalé ohodnocení (ohodnocení nejkratší cesty z počátku do něj)

 **Doc-ohod**  **Trv-ohod**

Nejprve získají všechny vrcholy kromě počátku p **Doc-ohod** $= \infty$. **Trv-ohod**(p) $= 0$.

Lze-li si z posledně získaného vrcholu s trvalým ohodnocením zkrátit cestu, do některého z jeho sousedů opraví se dočasné ohodnocení těchto sousedů.

Do toho (těch) vrcholu, který má ze všech dočasné ohodnocení nejnižší, již určitě kratší cesta nevede. Jeho dočasné ohodnocení tedy prohlásíme za trvalé.

 Opakujeme, dokud jsme nezískali pro cílový vrchol trvalé ohodnocení.

Při algoritmu se každá hrana v grafu vyšetřuje pouze jednou.

Neorientovaný graf s M vrcholy má nejvýše $M \cdot (M - 1) / 2$ hran, orientovaný nejvýše M^2 orientovaných hran.)

Složitost Dijkstrova algoritmu pro graf s N hranami a M vrcholy je tedy $\Theta(N)$, či $O(M^2)$.

Poznamenejme, že pro tento algoritmus je **požadavek**, **aby hrany byly ohodnoceny kladnými čísly** (při hledání minimálně ohodnocené cesty) **je podstatný**. Zaručuje ukončení cyklu v Dijkstrově algoritmu.

Problém minimální kostry souvislého ohodnoceného grafu

Problém: Je dán neorientovaný, **hranově ohodnocený** souvislý graf G . **Kostra** (někdy též napnutý strom) grafu G je **jeho faktor, tedy podgraf, který obsahuje všechny vrcholy grafu G , je souvislý a není v něm žádný cyklus**. Kostra je zřejmě strom. Každá její hrana je most (jejím vynecháním již získáme nesouvislý podgraf). Má-li graf G M vrcholů, má jeho kostra též M vrcholů a $M - 1$ hran.

Úkol je nalézt ze všech koster grafu tu, která má minimální součet ohodnocení hran.

Ohodnocení může znamenat (jako u nejkratší cesty): délku, cenu, pravděpodobnost poruchy,

Význam:

Nejlevnější dopravní síť.

Nejlevnější, nejprůchodnější, či nejspolehlivější propojení počítačů do sítě, ...

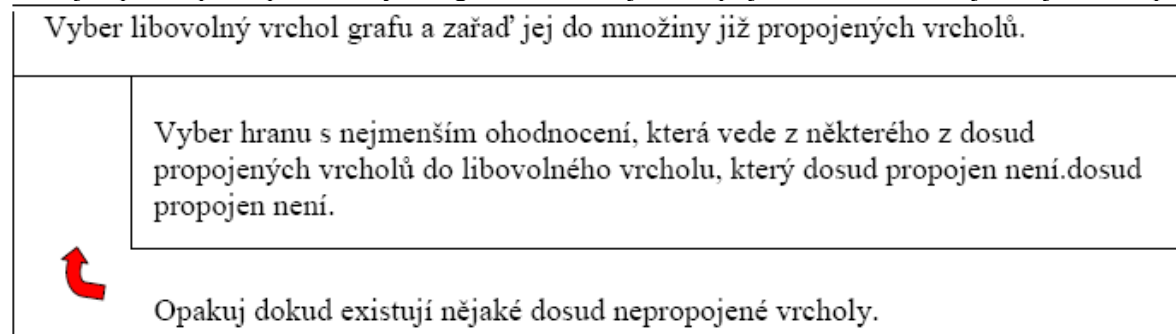
Řešení hrubou silou: Generovat všechny podmnožiny množiny všech hran grafu G o $M - 1$ prvcích. Vyloučit ty, které netvoří kostru, ze zbývajících vybrat minimální. Má-li graf M vrcholů a N hran, je takových výběrů $\text{Comb}(N, M - 1) = N! / ((M - 1)! \cdot (N - M + 1)!)$.

Algoritmus nemá polynomiální složitost \Rightarrow je nepoužitelný.

Nepomohlo by, ani kdybychom uměli snadno generovat přímo kostry. Úplný graf o M vrcholech má M^{M-2} koster.

Analogie Dijkstrova algoritmu je tak zvaný **Priamův algoritmus**.

Pracuje opět na principu hladových algoritmů. „Dělej to, co přijde momentálně nejlevněji!“ Jeho princip je:

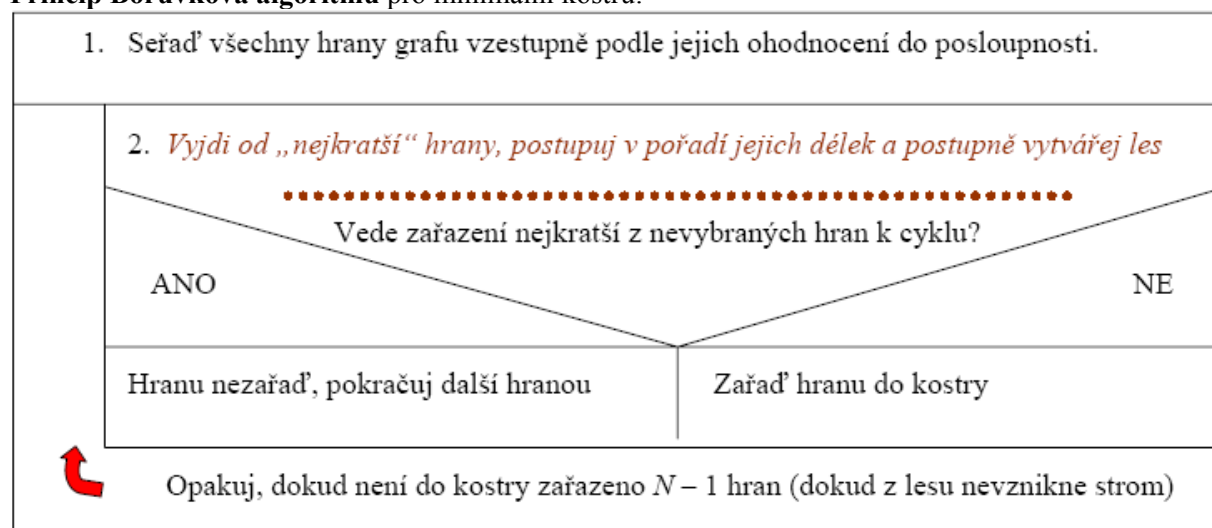


Algoritmus má M kroků. V každém z nich je třeba zkoumat dosud nepropojené vrcholy. Má **složitost** $\Theta(M^2)$.

Zajímavý algoritmus navrhl v roce 1926 český (moravský) matematik Prof. Borůvka, tehdy ještě bez jakéhokoliv zaměření na počítače a jejich programy.

Dnes je ve světě znám (nezaslouženě) jako „Kruskalův algoritmus“

Princip Borůvkova algoritmu pro minimální kostru:



Prvý krok má při užití vhodného algoritmu řazení složitost $\Theta(N \cdot \log(N))$, kde N je počet hran grafu G .

Ze složitostí druhého kroku může být problém. Při „ruční“ realizaci algoritmu je zda cyklus vzniká či nikoliv vidět „na první pohled“. S algoritmizací tohoto ověření může však být menší problém. Je třeba vést evidenci o jednotlivých postupně vznikajících souvislých komponentách vznikající kostry (stromech v lese). Pokud se informace o vrcholech v jednotlivých komponentách ukládají v binárních vyvážených stromech (eventuálně AVL-stromech), lze o jednom zařazení rozhodnout a informace aktualizovat v čase $O(\log(M))$. Tedy celý cyklus 2. má složitost $O(M \cdot \log(M))$. To je nižší než krok 1.

Celková složitost Borůvkova algoritmu je tedy $\Theta(N \cdot \log(N))$, kde N je počet hran grafu.

Otázka: „Který z obou algoritmů je výhodnější? Priamův, či Borůvkův?“

Těžko dát jednoznačnou odpověď. Situace je typická. Vzniká často v případech, kdy je pro týž problém k dispozici více algoritmů.

Je-li M počet vrcholů souvislého grafu G a M počet jeho hran, platí vždy:

$M - 1 \leq N \leq M \cdot (M - 1)$.

Dolní meze je dosaženo, je-li G strom.

Horní meze je dosaženo, je-li G úplný graf.

\Rightarrow Pro „řidké“ grafy je výhodnější algoritmus Borůvkův, pro „husté“ grafy algoritmus Priamův.

Problém maximálního toku v síti

Problém: Je dán orientovaný graf, hranově ohodnocený kladnými čísly. Tato čísla mají význam maximálního možného toku hranou (propustnost hrany). Jsou vymezeny dva vrcholy:

- **Zdroj z .**

- **Spotřebič (ponor) s .**

Hledáme **maximální možný souhrnný tok ze zdroje k spotřebiči**, který je možný, aniž by byla překročena maximální kapacita jednotlivých spojů (hran v grafu).

Představa a aplikace může být tok nějaké materie (ropy, plynu) potrubím, zboží v dopravní síti, dat v komunikační síti apod.

Graf lze pro jednoduchost považovat za úplný. Pokud vrcholy spojeny hranou nejsou, doplníme hranu s kapacitou 0.

Je-li V množina vrcholů grafu, lze tok charakterizovat reálnou funkcí f , definovanou na $V \times V$.

$f(u, v)$ označuje objem toku z vrcholu u do vrcholu v . Pro tuto funkci musí platit:

- Antisymetrie: $f(u, v) = -f(v, u)$.

- Pro všechny vrcholy u , s výjimkou zdroje a spotřebiče Kirchhoffův zákon: $\sum_{v \in V} f(u, v) = 0$ (.) (Co tam přiteklo, to taky

odtud odeče).

- Je dodrženo kapacitní omezení $f(u, v) \leq c(u, v)$.

Úkol je maximalizovat tok od zdroje k spotřebiči, tedy součty $\sum_{u \in V} f(z, u) = -\sum_{u \in V} f(u, s)$

Úloha může mít varianty například varianta s rozšířením o dolní omezení toku každou hranou nebo úloha o cirkulaci (není zdroj a spotřebič, Kirchhoffův zákon platí pro každý vrchol).

Jiné zobecnění: úloha s více zdroji a/nebo více spotřebiči.

Některé pojmy:

Jestliže je f tok sítě a c je kapacita hrany, pak $c - f$ se nazývá **zbytková (residuální) kapacita**. V případě záporného toku hranou může být zbytková kapacita větší než byla původní kapacita.

⇒ V dalším kroku se pracuje se zbytkovým (reziduálním) grafem

Cesta z z do s je nasycená, pokud ve zbytkovém grafu již cestu netvoří. (Nelze již nic tou cestou poslat navíc).

Fordův a Fulkersonův algoritmus řešení:

1. Začneme nulovým tokem.

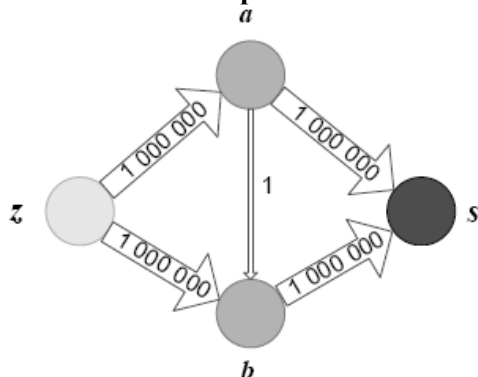
2. Pokud existuje nějaká nenasyčená cesta z z do s , určíme hranu na této cestě, která má nejmenší kapacitu a o tuto kapacitu zvýšíme toky po této cestě. Cestu tím nasatíme a určíme zbytkový graf. Tento bod opakujeme, dokud existují *nějaké* nenasyčené cesty. Pokud jsou již všechny cesty nasyceny, máme již dosažen maximální tok sítě f^* .

Tento „algoritmus“ není formulován deterministicky. Nenasyčených cest může totiž být více. Je třeba doplnit nějaké kritérium výběru nenasyčené cesty.

Složitost algoritmu můžeme odhadnout pro případ že toky a kapacitní omezení jsou celá čísla. Jsou-li racionální, lze úlohu převést na celočíselnou (společný jmenovatel).

Při každém kroku se tok zvýší aspoň o jedna ⇒ časová složitost je $O(|f^*|)$.

Skutečná složitost podstatně závisí na volbě nenasyčené cesty.



Při volbě nenasyčených cest $z \rightarrow a \rightarrow s$, $z \rightarrow b \rightarrow s$ získáme řešení za 2 kroky.

Při volbě cest: $z \rightarrow a \rightarrow b \rightarrow s$, $z \rightarrow b \rightarrow a \rightarrow s$, $z \rightarrow a \rightarrow b \rightarrow s$, $z \rightarrow b \rightarrow a \rightarrow s$, ... bude potřeba 2 000 000 kroků!

Pro volbu vhodného pořadí výběru z několika nenasyčených cest se užívají **heuristiky**.

Heuristika je (nejisté) pravidlo, které ve „většině případů“ umožní řešení nebo usnadní jeho průběh.

Heuristiky se velmi často opírají opět o hladový princip.

Možné heuristiky pro výběr nenasyčených cest:

1. Vyber z nenasyčených cest vždy tu, která má největší residuální tok! Vedena na algoritmus složitosti $O(N \cdot \log(f^*))$. (N je počet hran v síti.)

2. Vyber z nenasyčených cest vždy tu nejkratší! Vedena na algoritmus složitosti $O(N^2 \cdot M)$. (N je počet hran v síti, M počet vrcholů.) Nezávisí tedy na velikosti maximálního toku.

Jsou známy ještě „lepší“ avšak „složitější“ heuristiky pro výběr nenasyčené cesty – složitosti $O(N \cdot M^2)$ či $O(M^3)$.

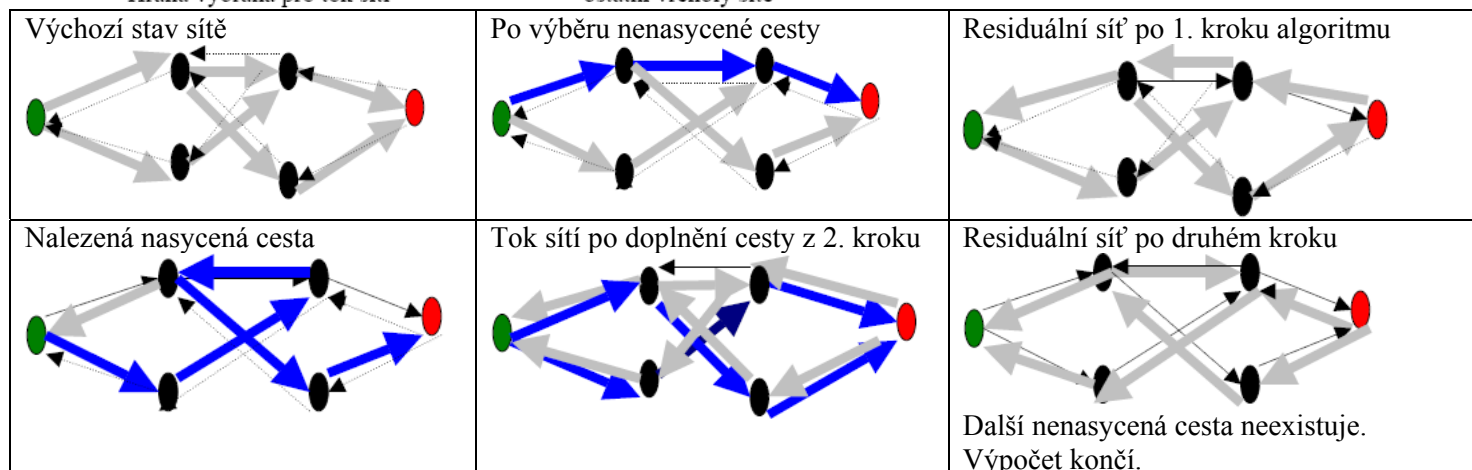
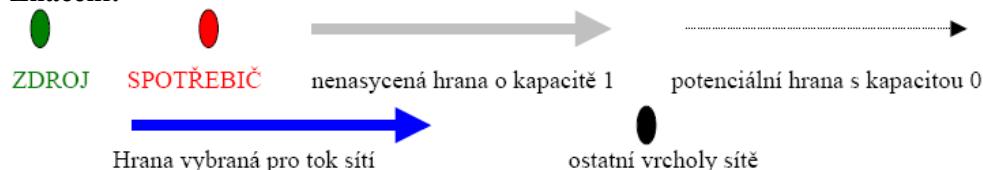
Na úlohu maximálního toku lze převést některé další důležité problémy.

Jejich řešení „hrubou silou“ by mělo nepřijatelnou časovou složitost.

Příklad užití Fordova a Fulkersonova algoritmu pro postupné nasycování cest v síti

Aneb: Jak napravit nevhodnou volbu cesty

Značení:



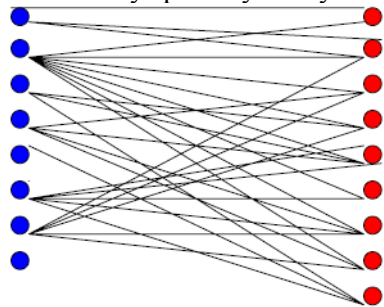
Problém maximálního párování

Problém: Jsou dány dvě disjunktní množiny A a B a symetrická relace na $A \cup B$ v níž jsou pouze dvojice prvků z nichž jeden leží v A , druhý v B . Mají se najít podmnožiny $A^* \subseteq A$ a $B^* \subseteq B$ s co největším počtem prvků takové, aby každý prvek A^* byl v relaci s jedním a pouze jedním prvkem B^* .

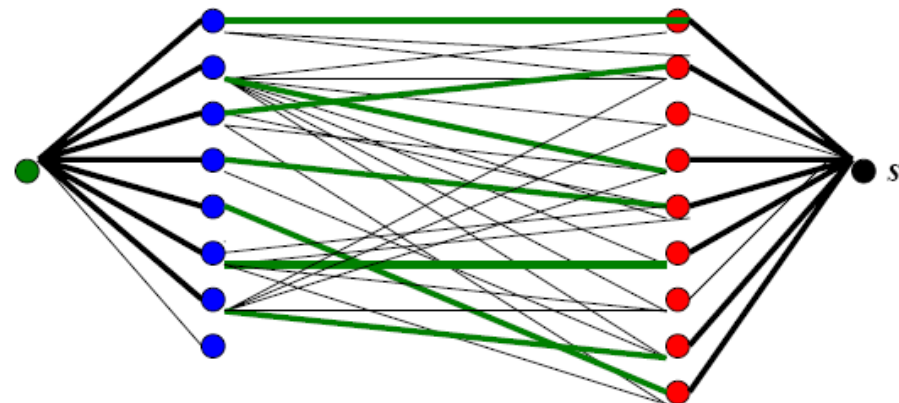
Úlohu si lze představit jako problém seznamovací kanceláře pro osoby různého pohlaví, kde každý zadá nezbytné požadavky na partnera a má smysl inicializovat seznámení pouze když jsou tyto požadavky oboustranně splněny. Úkol je dát šanci co nejvíce párům.

Jiná interpretace je vytváření týmů pracovníků dvou různých profesí, kdy ne každý je schopen a ochoten spolupracovat s kterýmkoliv partnerem.

Ve formulaci teorie grafů jde o bipartitní symetrický neorientovaný graf množina vrcholů je $A \cup B$, $A \cap B = \emptyset$ a neexistují hrany mezi dvěma prvky z A ani hrany mezi dvěma prvky z B . Hledáme podmnožinu hran takovou, aby žádné dvě neměly společný žádný vrchol.



Řešení lze na maximální tok sítě převést tak, že doplníme další dva „umělé“ vrcholy grafu. Zdroj z , který spojíme se všemi vrcholy z A , a spotřebič s , který spojíme se všemi vrcholy z B . Všem hranám zadáme kapacitu 1 a hledáme maximální tok touto sítí.



Ten určí maximální párování. $f^* = 7$

Problém minimální souvislosti grafu

Problém: Je dán souvislý graf. **Otázka je, kolik hran lze v tomto grafu odstranit, abychom měli jistotu, že nebude narušena souvislost grafu?**

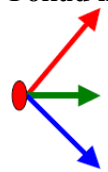
Problém má zřejmý význam pro teorii spolehlivosti. Zajištění funkceschopnosti při poruchách za cenu případného snížení výkonu.

Řešení:

1. Každé hraně (spoji) přiřadíme kapacitní ohodnocení 1.
2. Zvolíme jako zdroj libovolný vrchol.
3. Jako spotřebič volíme postupně všechny další vrcholy a řešíme tak $M - 1$ úloh o maximálním toku v síti.
4. Minimum z takto získaných $M - 1$ hodnot maximálních toků je minimální počet hran, které mohou být odstraněny, aby zůstala zachována souvislost grafu.

18. HEURISTIKY A NETRADIČNÍ POSTUPY

Pokud není znám algoritmus s přijatelnou časovou složitostí, jsou v principu tři možnosti:

- 
1. **Místo původní úlohy řešit jinou, skromnější**, například hledat místo optimálního řešení pouze „dobré“ řešení (třeba nepříliš vzdálené od optimálního). To lze pro řadu NP-úplných úloh.
 2. Řešit původní úlohu postupem, který „**ve většině případů**“ **nalezneme řešení v přijatelném čase** a riskovat, že v nepříznivém případě bude výpočet nepřijatelně dlouhý.
 3. Hledat řešení problému **jiným modelem výpočtu než je model von Neumannův**.

Prvé dva náhradní postupy (1. a 2.) bývají **založeny na heuristikách**.

HEURISTIKA je souhrn empirických (získaných ze zkušenosti) pravidel a zásad, které „fungují ve většině případů“, **nedávají však plnou záruku úspěchu**.

Typickou heuristikou je **princip hladových algoritmů**.

Jiná častá heuristika je známa pod názvem:

Prořezávání stromu

Někdy se algoritmy založené na tomto principu nazývají též **algoritmy větví a mezí** (branch and bound).

Vysvětlíme je pro optimalizační úlohu.

Jsou založeny na myšlence **uspořádat všechna přípustná řešení do orientovaného stromu, jehož listy budou všechna přípustná řešení**, tedy řešení odpovídající omezujícím podmínkám. Vnitřní vrcholy stromu budou odpovídat podmnožinám množiny všech přípustných řešení s nějakou společnou vlastností, která umožní odhad hodnotící funkce pro listy v příslušném podstromu.

Hledáme-li na listech (přípustných řešeních) minimum f , pak tento odhad může být funkce $V(u)$ taková, že $V(u) \leq \min_{x \in S(u)} f(x)$, kde minimum je přes všechny listy podstromu s kořenem u .

Větve stromu, kde je dolní odhad $V(u)$ nejmenší dávají „nejlepší šanci“ nalézt hledané minimum. Prohledáme je tedy prioritně. Větve, kde je dolní odhad $V(u)$ vysoký, dočasně nebo trvale odřízneme a tato možná řešení nebudeme uvažovat. Dočasné odřezání větve vede na 2. typ náhradního řešení, trvalé odřezání na 1. typ.

Pokud touto metodou nalezneme řešení s „přijatelnou“ hodnotou účelové funkce (i když si nemůžeme být jisti, že je tato hodnota optimální), můžeme definitivně odříznout ty větve, kde je dolní odhad vyšší nebo roven dosažené hodnotě.

Na tomto principu pracuje například **Littlův algoritmus** pro hledání Hamiltonovské cesty v grafu. **Typ 1. Správné řešení vždy nalezne**. Často však, v **někdy však pozdě**.

Jiný typický případ: **Programy na hraní šachů**. Hodnotí pozice a vylučují ty, které jsou málo nadějně. **Typ 2.** V omezeném čase nutno rozhodnout **vždy**. **Občas však rozhodne špatně** (může vyloučit například varianty s dočasnou obětí materiálu, vedoucí k matu).

Evoluční algoritmy

Algoritmy se snaží hledat řešení **maximalizační či minimalizační úlohy** na základě analogie „přirozeného výběru“ (Darwinova teorie).

Poznámka: Na minimalizační (max.) úlohu lze převést řadu dalších problémů. Například řešení rovnic. Rovnici $F(x) = c$ řešíme tak, že nalezneme $\min((F(x) - c)^2)$. Je-li 0 máme řešení, je-li nenulové, řešení neexistuje.

Heuristika evoluce:

Darwin: Úspěšní (lépe přizpůsobení prostředí) jedinci žijí déle a mají větší šance zplodit potomstvo. Tento vývoj vede k zdokonalování celé populace a k jejímu lepšímu přizpůsobení prostředí.

Pro problémy, pro které neznáme deterministické řešení v přijatelném čase:

Formulace pro maximum funkce F . (Pro minimum zcela analogické)

Zvolme odhady maxima náhodně. Na základě těch odhadů, které mají „dobrou“, to je vysokou hodnotu funkce F (funkce úspěšnosti – „fitness“) vytvářejme analogií biologických postupů nové odhady. Přiblížíme se tak nejvyšší možné hodnotě funkce úspěšnosti. (Snaha o napodobení přírody)

SCHÉMA ALGORITMU:

1. **Vytvoř náhodně výchozí populaci** P_0 , náhodným výběrem konečného (dosti velkého) počtu jedinců, bodů z definičního oboru funkce F , jejíž maximum hledáme. Pro všechny prvky populace P_0 vypočti hodnotu funkce úspěšnosti F . Její maximum je nultým odhadem řešení problému.
2. **Dokud se odhady řešení v jednotlivých krocích podstatněji liší, opakuj tyto kroky:**
 - 2.1 **KŘÍŽENÍ:** Vyber z populace náhodně nebo s preferencí těch jedinců, které mají vysokou hodnotu funkce úspěšnosti dva jedince a vytvoř nového jedince s vlastnostmi, které jsou kombinací vlastností obou „rodičů“.
 - 2.2 **MUTACE:** Proveď náhodnou změnu některých vlastností nově vytvořených jedinců.
 - 2.3 **SELEKCE:** Vypočti hodnotu funkce úspěšnosti pro nové členy populace a poté vyluč z populace ty nové či staré prvky, které mají nízkou hodnotu funkce úspěšnosti, aby populace měla stále „zhruba“ stejný počet členů. Vypočti nejvyšší hodnotu funkce F jako nový odhad řešení.

Rámcový algoritmus má **mnoho stupňů volnosti** (otázek které nutno dořešit pro konkrétní případ a které mohou mít rozhodující vliv na úspěšnost a rychlost nalezení řešení):

- Jak charakterizovat vlastnosti? (Co ovlivňuje hodnotu funkce úspěšnosti)
- Jak silně a důsledně preferovat úspěšné jedince při výběru pro křížení?
- Jak postupovat při křížení?
- Jak velké připouštět mutace?
- Jak dlouho nechat přežít staré populace, pokud jsou jedinci úspěšní?
- Kdy zastavit proces generace nových populací?

Obecně platí:

O úspěšnosti metody nelze dokázat žádné obecně platné matematické věty. Vše velmi závisí na konkrétním nastavení procesu.

V řadě aplikací bylo dosaženo pozoruhodných úspěchů.

Nic však není zaručeno. Kombinací vlastností úspěšných jedinců může vzniknout neúspěšný.

Některé zkušenosti:

Omezení mutací a dlouhé přežívání přestárých může proces urychlit, zvýší však nebezpečí uváznutí v lokálním maximu.

Varianta evolučních algoritmů:

Genetické algoritmy

Jedinec je reprezentován jako slovo pevné délky nad danou abecedou – chromozóm

Funkce úspěšnosti zobrazuje množinu všech možných chromozómů (jazyk) do množiny čísel, zpravidla do intervalu $\langle 0, 1 \rangle$.

Často $F: \{0, 1\}^N \rightarrow \langle 0, 1 \rangle$, případně „normalizovaně“ $\bar{F}(x) = F(x) / \sum_{j=1}^N F(x_j)$

Schéma křížení:

$$\left. \begin{array}{l} (\alpha_1, \dots, \alpha_r, \alpha_{r+1}, \dots, \alpha_K) \\ (\beta_1, \dots, \beta_r, \beta_{r+1}, \dots, \beta_K) \end{array} \right\} \begin{array}{l} \rightarrow (\alpha_1, \dots, \alpha_r, \beta_{r+1}, \dots, \beta_K) \\ \rightarrow (\beta_1, \dots, \beta_r, \alpha_{r+1}, \dots, \alpha_K) \end{array}$$

pro náhodně zvolené $r, 1 \leq r \leq K$, případně podle obecnějšího schématu

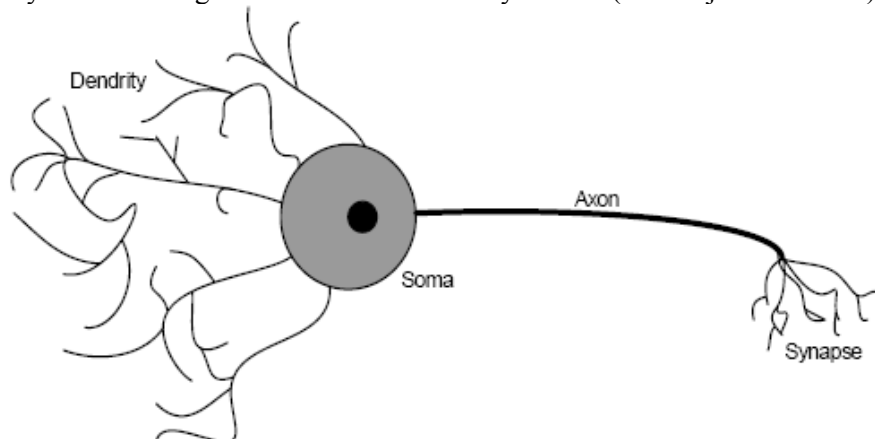
$$\left. \begin{array}{l} (\alpha_1, \dots, \alpha_r, \alpha_{r+1}, \dots, \alpha_s, \alpha_{s+1}, \dots, \alpha_K) \\ (\beta_1, \dots, \beta_r, \beta_{r+1}, \dots, \beta_s, \beta_{s+1}, \dots, \beta_K) \end{array} \right\} \begin{array}{l} \rightarrow (\alpha_1, \dots, \alpha_r, \beta_{r+1}, \dots, \beta_s, \alpha_{s+1}, \dots, \alpha_K) \\ \rightarrow (\beta_1, \dots, \beta_r, \alpha_{r+1}, \dots, \alpha_s, \beta_{s+1}, \dots, \beta_K) \end{array}$$

pro náhodně zvolená r a s .

Mutace je obvykle realizována jako náhodná změna chromozómu na jiný (opačný).

Neuronové sítě

Vychází z analogie s funkcí mozku. Lidský neuron (velmi zjednodušeně!):

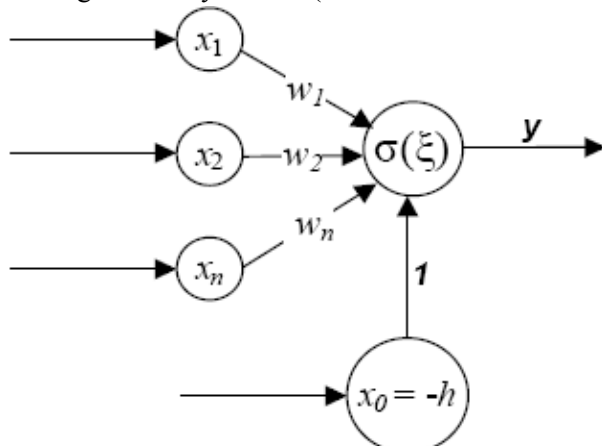


Krátké dendrity tvoří vstup binárních signálů. Dlouhý axon výstup binárního signálu.

Synapse jsou navázány na dendrity jiných neuronů.

Signály jsou binární (neuron je buď v klidovém nebo vybuzeném stavu). Učení probíhá tak, že se mění váhy jednotlivých spojů. Co se ukáže důležité, to se posílí, co důležité není oslabí.

Analogie – umělý neuron (realizován hardwarově nebo softwarově):



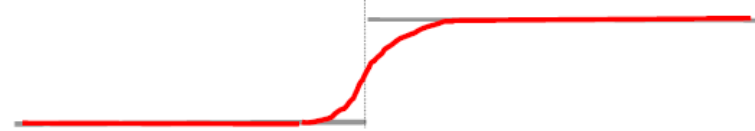
Neuron se aktivuje, pokud vstupy (0 nebo 1) násobené váhami w_j v součtu překročí prahovou hodnotu h .

Výstup umělého neuronu určuje **přenosová funkce** neuronu, zvaná též **aktivační funkce**. Nejčastěji se užívá **Heavisidova funkce** definovaná takto:

$$\sigma(\xi) = \begin{cases} 0 & \text{jestliže } \xi < 0 \\ 1 & \text{jestliže } \xi \geq 0 \end{cases}, \text{ kde } \xi = \sum_{j=0}^n w_j \cdot x_j = \sum_{j=1}^n w_j \cdot x_j - h.$$

Je-li $\sigma(\xi) = 0$, říkáme, že neuron je v klidovém stavu, je-li $\sigma(\xi) = 1$, říkáme, že je ve vybuzeném stavu.

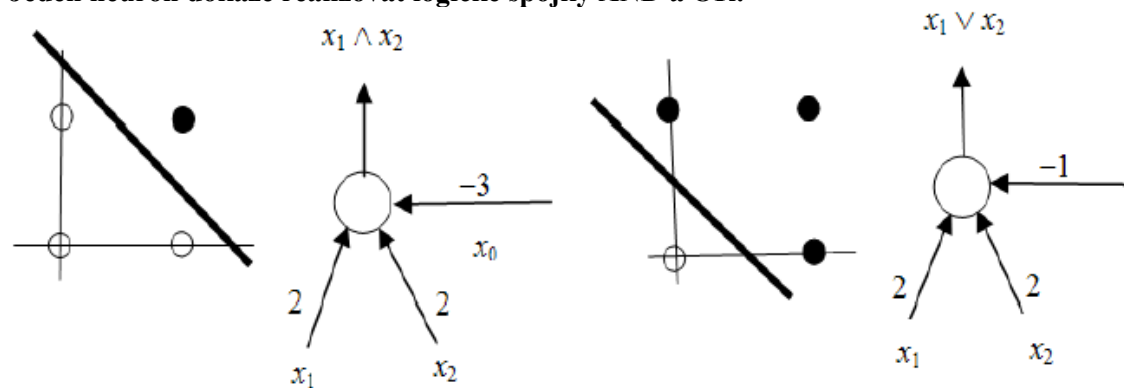
Nespojitá Heavisidova funkce bývá aproximována hladkými **sigmoidami**



Neurony jsou propojeny do sítí:

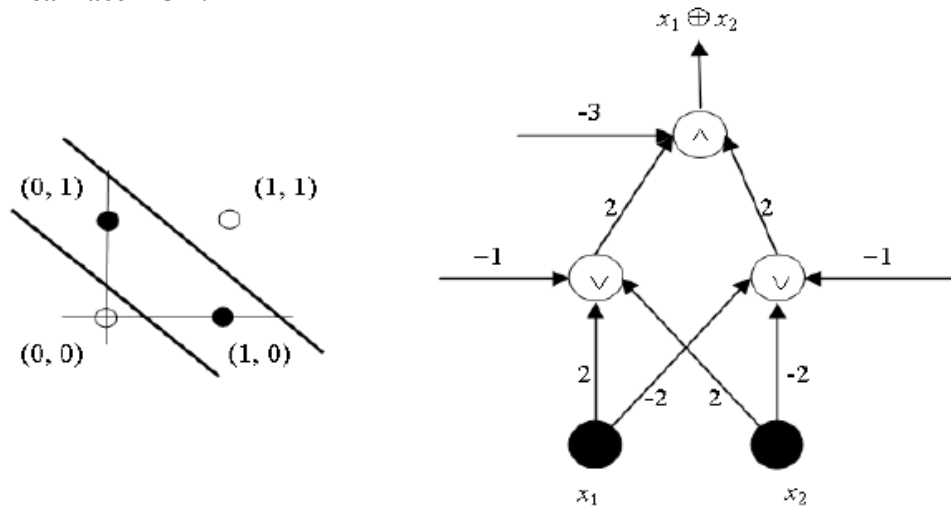
- Vstupní vrstva
- Výstupní neuron nebo neurony
- Skryté neurony - různé architektury sítí v závislosti na grafu propojení neuronů. (obvykle organizovány hierarchicky do několika vrstev (1, 2, ... skryté vrstvy))

Jeden neuron dokáže realizovat logické spojky AND a OR:



Na realizaci operací, které nejsou lineárně separabilní je třeba síť se skrytou vrstvou.

Realizace XOR:



Program pro neuronovou síť spočívá ve vhodném nastavení vah všech spojů.

Na rozdíl od číslicových počítačů jsou programy **stabilní vzhledem k malým změnám kódu**. Malá změna kódu číslicového počítače \Rightarrow zcela jiné chování programu.

Malá změna vah změní chování neuronové sítě jen málo.

Fáze práce sítě:

Organizační etapa: Stanoví se rozměr sítě a její organizace (topologie)

Adaptační etapa: Na základě trénovací množiny se upravují váhy tak aby práce sítě odpovídala požadavkům.

Aktivační etapa: Síť se využívá k výpočtu již „naučené“ sítě

V adaptační etapě:

Program se nestanovuje se „ad hoc“ programátorem ale **učením sítě**.

\Rightarrow Učení s učitelem

Učení s učitelem:

Existuje **algoritmus zpětného šíření chyby**, který na základě odchylky skutečného výstupu od požadovaného **upraví váhy** tak, aby výsledek byl „příště lepší“. Užívá se gradientní princip. Je minimalizována hodnota, která udává součet druhých mocnin odchylek výsledků, které dává síť od výsledků, které byly očekávány.

Učení bez učitele:

Podporuje se vytváření vhodných shluků při rozdělování vzorů do disjunktních množin. Uplatňují se analogie zákonů psychologie (Hebb: Shoda v aktivitě neuronů podporuje váhu spoje).

Užití Neuronových sítí:

Především tam, **kde neznáme zákonitost** (a tedy ani algoritmus řešení), **ale máme pouze zkušenost s kterými reakcemi na který vstup jsme spokojeni**.

Tam, kde je třeba rozhodovat na **základě kontextu**.



- Rozpoznávání písma, řeči, ...
- Predikce časových řad.
- Komprese obrazových dat.
- Expertní rozhodování („tuším jak se správně rozhodnout, ale zdůvodnit proč neumím“).
- ...

Realizace neuronových sítí je dnes především softwarová.

Perspektivně lze realizovat přímo hardware.

Jiné netradiční architektury výpočtu**Paralelní systémy:**

Tradiční paralelizmus (výpočetní systémy na principu SIMD, MIMD, multicube, ...) pro řešení úloh, kde není znám polynomiálně složitý algoritmus **příliš nepomohou**.

Je-li **K** procesorů, zvýší se propustnost systému nejvýše **K-krát**. Třídou složitosti Θ to neovlivní.

Jiné fyzikální, chemické, biologické, ... modely výpočtu:

Kvantové počítače: Foton se může nacházet „současně na více místech“ (s různou pravděpodobností). Nemá deterministicky určenou polohu. To dává šanci elementární částice užít přímo pro modelování nedeterministického Turingova stroje či jiného modelu nedeterministického výpočtu. Ve stádiu předběžných úvah a neurčitých záměrů

DNA počítače: Myšlenka založena se schopnosti řetězců aminokyselin DNA vytvářet masivně vlastní kopie paralelně. Výpočet by byl realizován jako biologický experiment. Pokud se aminokyseliny spojí do vhodného řetězce, lze jej považovat za řešení úlohy.

Lepší perspektivu skýtají možná peptidy (12 bází místo 4 bází u DNA).

Ve stádiu předběžných úvah a neurčitých záměrů

Chemické počítače (reakčně difusní modely výpočtu): Data jsou reprezentována různými koncentracemi chemikálií na vstupu. Výpočet je modelován průběhem chemické reakce. Ve stádiu předběžných úvah a neurčitých záměrů

Analogové počítače: Jsou starší než u číslicových. Ke škodě věci se na ně poněkud pozapomenulo. Vytvoří se fyzikální, obvykle spojitě pracující model děje (mechanický, hydraulický, elektromagnetický, ...), který se řídí stejnými nebo podobnými zákony jako řešený problém. Nechá se proběhnout vývoj na tomto modelu. Výsledek poskytne informaci o řešení původního problému. Dávno známé, dnes neprávem poněkud opomíjené