

# Obsah

<b>I Společné povinné okruhy</b>	<b>4</b>
<b>1 Základy složitosti a vyčíslitelnosti</b>	<b>5</b>
1.1 Výpočetní modely (Turingovy stroje, RAM) . . . . .	5
1.1.1 Turingův stroj . . . . .	5
1.1.2 RAM (Random Access Machine) . . . . .	7
1.1.3 Gödelovo číslo . . . . .	9
1.1.4 Univerzální Turingův stroj . . . . .	10
1.2 Rekurzivní a rekurzivně spočetné množiny. . . . .	10
1.2.1 Formální definice . . . . .	10
1.3 Algoritmicky nerozhodnutelné problémy (halting problem). . . . .	10
1.3.1 (Částečně) rozhodnutelný jazyk . . . . .	11
1.3.2 Nerozhodnutelné jazyky . . . . .	11
1.3.3 Vlastnosti částečně rozhodnutelných jazyků . . . . .	12
1.3.4 Enumerátory . . . . .	13
1.3.5 Převoditelnost, úplné jazyky . . . . .	13
1.3.6 Riceova věta . . . . .	14
1.4 Nedeterministický výpočetní model. . . . .	15
1.4.1 Časová a prostorová složitost NTS . . . . .	15
1.5 Základní třídy složitosti a jejich vztahy. . . . .	15
1.5.1 Základní třídy složitosti . . . . .	16
1.5.2 Vztahy mezi třídami . . . . .	18
1.6 Věty o hierarchii. . . . .	20
1.6.1 Věta o deterministické prostorové hierarchii . . . . .	20
1.6.2 Věta o časové prostorové hierarchii . . . . .	21
1.7 Úplné problémy pro třídu NP, Cook-Levinova věta. . . . .	22
1.7.1 Polynomiální převoditelnost, NP-úplnost . . . . .	22
1.7.2 Cook-Levinova věta . . . . .	22
1.7.3 Další NP-úplné problémy . . . . .	26
1.8 Pseudopolynomiální algoritmy, silná NP-úplnost. . . . .	30
1.8.1 Pseudopolynomiální algoritmy . . . . .	30
1.8.2 Silná NP-úplnost . . . . .	31
1.9 Aproximační algoritmy a schémata. . . . .	32
1.9.1 Optimalizační úloha . . . . .	32
1.9.2 Aproximační algoritmus . . . . .	33
1.9.3 Aproximační schémata . . . . .	35
1.9.4 Neapproximovatelnost . . . . .	38
<b>2 Datové struktury</b>	<b>40</b>
2.1 Vyhledávací stromy ((a,b)-stromy, Splay stromy) . . . . .	40
2.1.1 Binární vyhledávací strom (BVS) . . . . .	40
2.1.2 (a,b)-stromy . . . . .	40
2.1.3 Červeno-černé stromy . . . . .	43
2.1.4 Splay stromy . . . . .	44
2.1.5 AVL stromy . . . . .	47

2.2	Haldy (regulární, binomiální) . . . . .	48
2.2.1	Regulérní halda . . . . .	48
2.2.2	Binomiální halda . . . . .	49
2.2.3	Fibonacciho halda . . . . .	50
2.2.4	Další haldy . . . . .	51
2.3	Hašování, řešení kolizí, univerzální hašování, výběr hašovací funkce. . . . .	51
2.3.1	Základní pojmy . . . . .	51
2.3.2	Řešení kolizí . . . . .	52
2.3.3	Univerzální hashování . . . . .	57
2.3.4	Perfektní hashování . . . . .	59
2.4	Analýza nejhoršího, amortizovaného a očekávaného chování datových struktur. . . . .	61
2.4.1	Asymptotická notace . . . . .	62
2.4.2	Prostorová (paměťová) složitost reprezentované struktury . . . . .	62
2.4.3	Časová složitost operací na datové struktuře . . . . .	62
2.5	Chování a analýza datových struktur na systémech s paměťovou hierarchií. . . . .	65
2.5.1	Paměťová hierarchie . . . . .	65
2.5.2	Cache-aware a cache-oblivious algoritmy . . . . .	66
2.5.3	Cache-aware a cache-oblivious datové struktury . . . . .	68
2.5.4	Strategie pro správu cache . . . . .	69
<b>II</b>	<b>Inteligentní agenti</b>	<b>70</b>
<b>3</b>	<b>Přírodou inspirované počítání</b>	<b>71</b>
3.1	Genetické algoritmy, genetické a evoluční programování. . . . .	71
3.1.1	Evoluční algoritmy . . . . .	71
3.1.2	Genetické algoritmy (GA) . . . . .	72
3.1.3	Genetické programování (GP) . . . . .	72
3.1.4	Evoluční programování (EP) . . . . .	72
3.2	Teorie schémat, pravděpodobnostní modely jednoduchého genetického algoritmu. . . . .	73
3.2.1	Schéma . . . . .	73
3.2.2	Pravděpodobnostní modely . . . . .	75
3.3	Evoluční strategie, diferenciální evoluce, koevoluce, otevřená evoluce. . . . .	76
3.3.1	Evoluční strategie . . . . .	76
3.3.2	Diferenciální evoluce . . . . .	78
3.3.3	Koevoluce . . . . .	78
3.3.4	Otevřená evoluce . . . . .	79
3.4	Rojové optimalizační algoritmy. . . . .	80
3.4.1	Optimalizace hejnym částic . . . . .	80
3.4.2	Další rojové optimalizační algoritmy . . . . .	80
3.5	Memetické algoritmy, hill climbing, simulované žlhání. . . . .	81
3.6	Aplikace evolučních algoritmů (evoluce expertních systémů, neuroevoluce, řešení kombinatorických úloh, vícekriteriální optimalizace). . . . .	82
3.6.1	Evoluce expertních systémů . . . . .	82
3.6.2	Neuroevoluce . . . . .	84
3.6.3	Řešení kombinatorických úloh . . . . .	86
3.6.4	Vícekriteriální optimalizace (MOEA – Multi-objective EA) . . . . .	89
<b>III</b>	<b>Strojové učení</b>	<b>92</b>
<b>4</b>	<b>Strojové učení a jeho aplikace</b>	<b>93</b>
4.1	Strojové učení; prohledávání prostoru verzí, učení s učitelem a bez učitele, pravděpodobnostní přístupy, teoretické aspekty strojového učení. . . . .	93

4.2	Evoluční algoritmy; základní pojmy a teoretické poznatky, hypotéza o stavebních blocích, koevoluce, aplikace evolučních algoritmů. . . . .	93
4.3	Strojové učení v počítačové lingvistice a algoritmy pro statistický parsing. . . . .	93
4.3.1	Základy teorie informace . . . . .	94
4.3.2	Modelování jazyka, zašuměný kanál . . . . .	94
4.3.3	Vyhlažování jazykového modelu . . . . .	96
4.3.4	Třídy slov . . . . .	97
4.3.5	Markovovy modely . . . . .	97
4.3.6	Tagging . . . . .	99
4.3.7	Statistický parsing . . . . .	101
4.3.8	Statistický strojový překlad . . . . .	101
4.4	Pravděpodobnostní algoritmy pro analýzu biologických sekvencí; hledávání motivů v DNA, strategie pro detekci genů a predikci struktury proteinů. . . . .	103
4.4.1	Restrikční mapování . . . . .	103
4.4.2	Hledání motivů . . . . .	105
4.4.3	Sequence alignment . . . . .	106
4.4.4	Genome rearrangement . . . . .	106
4.4.5	Predikce genů . . . . .	107
<b>5</b>	<b>Neuronové sítě</b>	<b>108</b>
5.1	Neurofyziologické minimum. . . . .	108
5.1.1	Mozek . . . . .	108
5.1.2	Neuron . . . . .	108
5.1.3	Přenos signálu . . . . .	109
5.1.4	Paměť . . . . .	110
5.2	Modeły pro učení s učitelem, algoritmus zpětného šíření, strategie pro urychlení učení, regularizační techniky a generalizace. . . . .	111
5.2.1	Modeły pro učení s učitelem . . . . .	111
5.2.2	Algoritmus zpětného šíření (Backpropagation) . . . . .	113
5.2.3	Strategie pro urychlení učení . . . . .	116
5.3	Asociativní paměti, Hebbovské učení a hledání suboptimálních řešení, stochastické modely. . . . .	120
5.3.1	Asociativní paměti . . . . .	120
5.3.2	Hebbovské učení a hledání suboptimálních řešení . . . . .	121
5.3.3	Stochastické modely . . . . .	130
5.4	Umělé neuronové sítě založené na principu učení bez učitele. . . . .	131
5.4.1	Kompetiční učení . . . . .	131
5.4.2	PCA (Principal Component Analysis) . . . . .	133
5.4.3	Kohonenovy mapy . . . . .	135
5.4.4	Učení s učitelem . . . . .	138
5.5	Modulární, hierarchické a hybridní modely neuronových sítí. . . . .	139
5.5.1	Modulární sítě . . . . .	139
5.5.2	Hybridní sítě . . . . .	140
5.5.3	Vícevrstvé Kohonenovy mapy . . . . .	142
5.6	Genetické algoritmy a jejich využití při učení umělých neuronových sítí. . . . .	142

# Část I

## Společné povinné okruhy

# Kapitola 1

## Základy složitosti a vyčíslitelnosti

### 1.1 Výpočetní modely (Turingovy stroje, RAM).

#### 1.1.1 Turingův stroj

(Jednopáskový deterministický) **Turingův stroj (TS)**  $M$  je pětice

$$M = (Q, \Sigma, \delta, q_0, F)$$

- $Q$  je konečná **množina stavů**.
- $\Sigma$  je konečná **pásková abeceda**, která obsahuje znak  $\lambda$  pro prázdné políčko. (Často budeme rozlišovat **páskovou (vnitřní)** a **vstupní (vnější) abecedu**.)
- $\delta : Q \times \Sigma \mapsto Q \times \Sigma \times \{R, N, L\} \cup \perp$  je **přechodová funkce**, kde  $\perp$  označuje nedefinovaný přechod.
- $q_0 \in Q$  je **počáteční stav**.
- $F \subseteq Q$  je **množina přijímajících stavů**.

Turingův stroj sestává z **řídící jednotky**, **pásky**, která je potenciálně nekonečná v obou směrech, a **hlavy** pro čtení a zápis, která se pohybuje oběma směry.

**Displej** je dvojice  $(q, a)$ , kde  $q \in Q$  je aktuální stav Turingova stroje a  $a \in \Sigma$  je symbol pod hlavou. Na základě displeje TS rozhoduje, jaký další krok má vykonat.

**Konfigurace** zachycuje stav výpočtu Turingova stroje a skládá se ze

- stavu řídící jednotky
- slova na pásmu (od nejlevějšího do nejpravějšího neprázdného políčka)
- pozici hlavy na pásmu (v rámci slova na této pásmu)

##### 1.1.1.1 Výpočet TS

Výpočet zahajuje TS  $M$  v **počáteční konfiguraci**, tedy v počátečním stavu se vstupním slovem zapsaným na pásmu a hlavou nad nejlevějším symbolem vstupního slova. Vstupní slovo nesmí obsahovat prázdné políčko. Pokud se  $M$  nachází ve stavu  $q \in Q$  a pod hlavou je symbol  $a \in \Sigma$ , pak krok výpočtu probíhá následovně:

1. Je-li  $\delta(q, a) = \perp$ , výpočet  $M$  končí,
2. Je-li  $\delta(q, a) = (q', a', Z)$ , kde  $q' \in Q$ ,  $a' \in \Sigma$  a  $Z \in \{L, N, R\}$ , přejde  $M$  do stavu  $q'$ , zapíše na pozici hlavy symbol  $a'$  a pohne hlavou doleva (pokud  $Z = L$ ), doprava (pokud  $Z = R$ ), nebo hlava zůstane stát (pokud  $Z = N$ ).

TS  $M$  **přijímá slovo**  $w$ , pokud výpočet  $M$  se vstupem  $w$  skončí a  $M$  se po ukončení výpočtu nachází v přijímajícím stavu.

TS  $M$  **odmítá slovo**  $w$ , pokud výpočet  $M$  nad vstupem  $w$  skončí a  $M$  se po ukončení výpočtu nenachází v přijímajícím stavu.

Fakt, že výpočet  $M$  nad vstupním slovem  $w$  skončí, označíme pomocí  $M(w)\downarrow$  a řekneme, že výpočet **konverguje**.

Fakt, že výpočet  $M$  nad vstupním slovem  $w$  nikdy neskončí, označíme pomocí  $M(w)\uparrow$  a řekneme, že výpočet **diverguje**.

### 1.1.1.2 Turingovsky rozhodnutelné jazyky

Jazyk slov přijímaných TS  $M$  označíme pomocí  $L(M)$ .

Řekneme, že jazyk  $L$  je **částečně (Turingovsky) rozhodnutelný** (též **rekurzivně spočetný**), pokud existuje Turingův stroj  $M$ , pro který  $L = L(M)$ .

Řekneme, že jazyk  $L$  je **(Turingovsky) rozhodnutelný** (též **rekurzivní**), pokud existuje Turingův stroj  $M$ , který se *vždy zastaví* a  $L = L(M)$ .

### 1.1.1.3 Turingovsky vyčíslitelné funkce

Turingův stroj  $M$  s páskovou abecedou  $\Sigma$  **počítá** nějakou částečnou funkci  $f_M : \Sigma^* \mapsto \Sigma^*$  (částečná = pro některé vstupy není definovaná). Pokud  $M(w)\downarrow$  pro daný vstup  $w \in \Sigma^*$ , je hodnota funkce  $f_M(w)$  **definovaná**, což označíme pomocí  $f_M(w)\downarrow$ . **Hodnotou funkce**  $f_M(w)$  je potom slovo na (výstupní) pásce  $M$  po ukončení výpočtu nad  $w$ . Pokud  $M(w)\uparrow$ , pak je hodnota  $f_M(w)$  **nedefinovaná**, což označíme pomocí  $f_M(w)\uparrow$ .

Funkce  $f : \Sigma^* \mapsto \Sigma^*$  je **turingovsky vyčíslitelná**, pokud existuje Turingův stroj  $M$ , který ji počítá. **Každá turingovsky vyčíslitelná funkce má nekonečně mnoho různých Turingových strojů, které ji počítají!**

### 1.1.1.4 Varianty TS

Turingovy stroje mají řadu variant, například:

- TS s jednosměrně nekonečnou páskou
- TS s více páskami (vstupní/výstupní/pracovní)
- TS s více hlavami na páskách
- TS s pouze binární abecedou
- nedeterministické TS

Zmíněné varianty jsou ekvivalentní „našemu“ modelu v tom smyslu, že všechny přijímají touž třídu jazyků a vyčíslují touž třídu funkcí.

**$k$ -páskový Turingův stroj** se od jednopáskového Turingova stroje líší následujícím způsobem:

- Má  $k$  pásek, na každém je zvláštní hlava.
  - Vstupní pánska na počátku obsahuje vstupní řetězec. Často je určena jen pro čtení.
  - Pracovní pásky jsou určeny pro čtení i zápis.
  - Výstupní pánska na konci obsahuje výstupní řetězec. Často je určena jen pro zápis s pohybem hlavy jen vpravo.
- Hlavy na páskách se pohybují nezávisle na sobě.
- Přechodová funkce je typu

$$\delta : Q \times \Sigma^k \mapsto Q \times \Sigma^k \times \{R, N, L\}^k \cup \perp$$

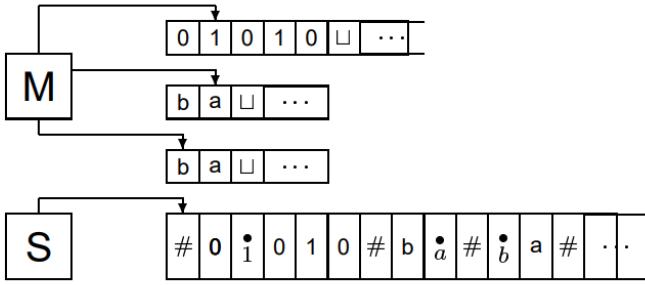
Neplést *multi-tape* (vícepáskové) a *multi-track* TS. Druhé jmenované mají také více pásek, ale *jedinou hlavu společnou pro všechny*.

**Věta 1.1.1.** Ke každému  $k$ -páskovému Turingovu stroji  $M$  existuje jednopáskový Turingův stroj  $M'$ , který simuluje práci  $M$ , přijímá týž jazyk jako  $M$  a počítá touž funkci jako  $M$ .

*Důkaz.* Pouze idea. Chceme simulovat  $k$ -páskový na jednopáskovém. Dvě varianty:

1. Obsahy pásek zapíšeme na jedinou, oddělíme je novým symbolem  $\#$ . Pozice hlav označuje speciální značku, např. je-li hlava nad symbolem 0, označíme to pomocí  $\bar{0}$  – v praxi to znamená pro každý symbol přidat do abecedy jeho označovanou variantu, tj.  $|\Sigma|$  nových symbolů. Dojde-li nějaké pásce místo, musí se všechny symboly vpravo od ní posunout.
2. Vytvoříme novou abecedu  $\Sigma'$  jako kartézský součin  $k$  abeced, tj.  $\Sigma' = \Sigma^k$ . Jeden symbol tedy reprezentuje  $k$ -tici původních symbolů. Je taky potřeba ošetřit pozici hlav.

□



### 1.1.2 RAM (Random Access Machine)

RAM je stroj s náhodným přístupem do paměti. Jde o model, který se často používá jako základní výpočetní model při měření časové i prostorové složitosti algoritmů. Cílem bylo vytvořit model, který by se co nejvíce blížil reálným počítačům. Podobně jako Turingovy stroje, i RAM je strojem s **oddělenou pamětí pro data a pro instrukce**, nejedná se tedy o stroje Von Neumannovy architektury.

**Program** pro RAM je konečnou posloupností instrukcí  $P = I_0, I_1, I_2, \dots, I_l$

Paměť pro data se skládá z neomezené posloupnosti registrů  $r_i, i \in \mathbb{N}$ . Obsahem registru může být libovolně velké přirozené číslo. Při popisu instrukcí budeme dodržovat následující konvence:

- Obsah registru  $r_i$  budeme označovat pomocí  $[r_i]$ .
- Nepřímá adresace (tj. obsahem jiného registru) pomocí  $[[r_i]] = [r_{[r_i]}]$ .
- Přiřazení hodnoty  $c$  do registru  $r_i$  označíme  $r_i \leftarrow c$

Seznam instrukcí pro RAM viz tabulka 1.1.

LOAD( $C, r_i$ )	$r_i \leftarrow C$
ADD( $r_i, r_j, r_k$ )	$r_k \leftarrow [r_i] + [r_j]$
SUB( $r_i, r_j, r_k$ )	$r_k \leftarrow [r_i] \dot{-} [r_j] \quad (x \dot{-} y = \max(x - y, 0))$
COPY( $[r_p], r_d$ )	$r_d \leftarrow [[r_p]]$
COPY( $r_s, [r_d]$ )	$r_{[r_d]} \leftarrow [r_s]$
JNZ( $r_i, I_z$ )	if $[r_i] > 0$ then goto instruction $I_z$
READ( $r_i$ )	$r_i \leftarrow \text{input}$
PRINT( $r_i$ )	$\text{output} \leftarrow [r_i]$

Tabulka 1.1: Seznam instrukcí RAM

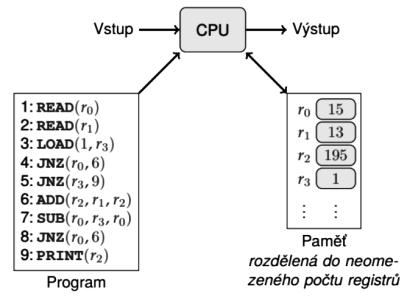
#### 1.1.2.1 Jazyky rozhodnutelné RAM

Uvažme abecedu  $\Sigma = \{\sigma_1, \sigma_2, \dots, \sigma_k\}$ . Slovo  $w = \sigma_{i_1}, \sigma_{i_2} \dots \sigma_{i_n}$  předáme RAMu  $R$  jako posloupnost čísel  $i_1, \dots, i_n$ . Konec slova pozná  $R$  díky tomu, že READ načte 0, není-li už k dispozici vstup.

RAM  $R$  **přijme** slovo  $w$ , pokud  $R(w) \downarrow$  a první číslo, které  $R$  zapíše na výstup je 1.

RAM  $R$  **odmítne** slovo  $w$ , pokud  $R(w) \downarrow$  a  $R$  bud' na výstup nezapíše nic, nebo první zapsané číslo je jiné než 1.

**Jazyk** slov přijímaných RAMem  $R$  označíme pomocí  $L(R)$ .



Pokud pro jazyk  $L$  platí, že  $L = L(R)$  pro nějaký RAM, pak řekneme, že je **částečně rozhodnutelný (RAMem)**. Pokud se navíc výpočet  $R$  nad každým vstupem zastaví, řekneme, že je  $L = L(R)$  **rozhodnutelný (RAMem)**.

#### 1.1.2.2 Funkce vyčíslitelné na RAMu

O RAMu  $R$  řekneme, že **počítá** částečnou aritmetickou funkci  $f : \mathbb{N}^n \mapsto \mathbb{N}, n \geq 0$ , pokud za předpokladu, že  $R$  dostane na vstup  $n$ -tici  $(x_1, \dots, x_n)$ , platí následující:

- Je-li  $f(x_1, \dots, x_n) \downarrow$ , pak  $R(x_1, \dots, x_n) \downarrow$  a  $R$  vypíše na výstup hodnotu  $f(x_1, \dots, x_n)$ .
- Je-li  $f(x_1, \dots, x_n) \uparrow$ , pak  $R(x_1, \dots, x_n) \uparrow$ .

O funkci  $f$ , pro niž existuje RAM, který ji počítá, řekneme, že je **vyčíslitelná na RAMu**.

#### 1.1.2.3 Řetězcové (string) funkce vyčíslitelné na RAMu

RAM  $R$  **počítá** částečnou (řetězcovou) funkci  $f : \Sigma^* \mapsto \Sigma^*$ , kde  $\Sigma = \{\sigma_1, \sigma_2, \dots, \sigma_k\}$ , pokud platí:

- Vstupní řetězec  $w = \sigma_{i_1}\sigma_{i_2} \dots \sigma_{i_n}$  je předaný jako posloupnost čísel  $i_1, \dots, i_n$ .
- Konec slova pozná  $R$  díky tomu, že READ načte 0, není-li už k dispozici vstup.
- Pokud je  $f(w) \downarrow = \sigma_{j_1}\sigma_{j_2} \dots \sigma_{j_m}$ , pak  $R(w) \downarrow$  a na výstup je zapsaná posloupnost čísel  $j_1, j_2, \dots, j_m, 0$ .
- Pokud  $f(w) \uparrow$ , pak  $R(w) \uparrow$ .

O funkci  $f$ , pro niž existuje RAM  $R$ , který ji počítá, říkáme, že je **vyčíslitelná na RAMu**.

#### 1.1.2.4 Programování na RAMu

Programy pro RAM odpovídají procedurálnímu jazyku:

- Máme k dispozici **proměnné (skalární i neomezená pole)**:  
Předpokládejme, že v programu používáme pole  $A_1, \dots, A_p$  a skalární proměnné  $x_0, \dots, x_s$ .
  - Pole indexujeme od 0.
  - Prvek  $A_i[j]$ , kde  $i \in \{1, \dots, p\}$ ,  $j \in \mathbb{N}$ , umístíme do registru  $r_{i+j \cdot (p+1)}$ .
  - Prvky pole  $A_i, i = 1, \dots, p$  jsou tedy vregistrech  $r_i, r_{i+p+1}, r_{i+2(p+1)}, \dots$
  - Proměnnou  $x_i$ , kde  $i \in \{0, \dots, s\}$  umístíme do registru  $r_{i \cdot (p+1)}$ .
  - Skalární proměnné jsou tedy postupně vregistrech  $r_0, r_{p+1}, r_{2(p+1)}, \dots$
- Cykly (**for** i **while**) – s pomocí podmíněného skoku, případně čítače v proměnné.
- Nepodmíněný skok (**goto**) – s použitím pomocného registru, kam uložíme 1 a použijeme podmíněný skok.
- **Podmíněný příkaz** – s pomocí podmíněného skoku.
- **Funkce a procedury** – do místa použití funkce rovnou v programu napíšeme tělo funkce (inline).
- *Nemáme rekurzivní volání funkcí* – ta se však dají vždy nahradit pomocí cyklu while a zásobníku.

#### 1.1.2.5 Ekvivalence RAM a TS

**Věta 1.1.2.** Ke každému Turingovu stroji  $M$  existuje ekvivalentní RAM  $R$ .

*Důkaz.* Odpovídající RAM sestrojíme takto:

- Obsah pásky uložen ve dvou polích:  $T_r$  obsahuje pravou část pásky a  $T_l$  obsahuje levou část pásky.
- Poloha hlavy – pamatujeme si index v proměnné  $h$  a stranu pásky (pravá/levá) v proměnné  $s$ .
- Stav – v proměnné  $q$ .

- Výběr instrukce – podmíněný příkaz podle  $h, s$  a  $q$

□

**Věta 1.1.3.** Ke každému RAMu  $R$  existuje ekvivalentní Turingův stroj  $M$ .

*Důkaz.* K RAMu  $R$  sestrojíme TS  $M$  jako 4-páskový:

**Vstupní páska** Posloupnost čísel, která má dostat  $R$  na vstup. Jsou zakódovaná binárně a oddělená znakem  $\#$ . Z této pásky  $M$  jen čte.

**Výstupní páska** Sem zapisuje  $M$  čísla, která  $R$  zapisuje na výstup. Jsou zakódovaná binárně a oddělená znakem  $\#$ . Na tuto pásku  $M$  jen zapisuje.

**Paměť RAM** Obsah paměti stroje  $R$  reprezentujeme na pásmu  $M$  takto:

Jsou-li aktuálně využité registry  $r_{i_1}, r_{i_2}, \dots, r_{i_m}$ , kde  $i_1 < i_2 < \dots < i_m$ , pak je na pásmu reprezentující paměť RAM  $R$  řetězec:

$$(i_1)_B | ([r_{i_1}])_B \# (i_2)_B | ([r_{i_2}])_B \# \dots \# (i_m)_B | ([r_{i_m}])_B$$

Index  $B$  značí binární zápis daného čísla.

**Pomocná páska** Pro výpočty součtu, rozdílu, neprímých adres, posunu části paměťové pásky a podobně.

□

**Churchova-Turingova teze:** Ke každému algoritmu v intuitivním smyslu existuje ekvivalentní Turingův stroj.

### 1.1.3 Gödelovo číslo

Chceme každému TS přiřadit číslo. Postup: zakódování TS v malé abecedě, převod do binární abecedy, převod binárního řetězce na přirozené číslo.

Mějme TS  $M = (Q, \Sigma, \delta, q_0, F)$  s **jediným přijímacím stavem a binární vstupní abecedou** (každý TS lze upravit do této podoby). Stačí nám vlastně zakódovat přechodovou funkci. K tomu použijeme nejprve abecedu

$$\Gamma = \{0, 1, L, N, R, |, \#, ;\}$$

Nechť  $Q = \{q_0, q_1, \dots, q_r\}$ , kde  $r \geq 1$ ,  $q_0$  je počáteční stav a  $q_1$  je jediný přijímací stav. Dále nechť  $\Sigma = \{X_0, X_1, \dots, X_s\}$ , kde  $s \geq 2$ ,  $X_0 = '0'$ ,  $X_1 = '1'$ ,  $X_2 = \lambda$ . Instrukci  $\delta(q_i, X_j) = (q_k, X_l, Z)$ , kde  $Z \in \{L, N, R\}$  zakódujeme jako

$$(i)_B | (j)_B | (k)_B | (l)_B | Z$$

kde index  $B$  značí zápis v binární soustavě. Nechť  $C_1, C_2, \dots, C_n$  jsou kódy jednotlivých instrukcí, pak celou funkci  $\delta$  zakódujeme jako

$$C_1 \# C_2 \# \dots \# C_n$$

Celý tento řetězec pak převedeme do binární soustavy podle této tabulky:

$\Gamma$	0	1	L	N	R		#	;
kód	000	001	010	011	100	101	110	111

Binárnímu řetězci  $w \in \{0, 1\}^*$  přiřadíme číslo  $i$ , jehož binární zápis je  $1w$ , tedy  $(i)_B = 1w$ . Tento řetězec pak označíme jako  $w_i$ . Dodefinujeme ještě, že 0 odpovídá prázdnému řetězci (tj.  $w_0 = w_1 = \epsilon$ ).

Každému TS  $M$  můžeme přiřadit **Gödelovo číslo**  $e$ , pro které platí, že řetězec  $w_e$  je kódem TS  $M$ . Turingův stroj s Gödelovým číslem  $e$  označíme pomocí  $M_e$ . Jazyk přijímaný Turingovým strojem  $M_e$  označíme  $L_e = L(M_e)$ .

Pokud řetězec  $w_e$  není syntakticky správným kódem Turingova stroje, pak  $M_e$  je prázdným Turingovým strojem, který každý vstup okamžitě odmítne a  $L_e = \emptyset$ . Z toho plyne, že ke každému číslu  $e$  jsme naopak schopni přiřadit nějaký Turingův stroj  $M_e$ .

Kód TS **není jednoznačný**, protože nezáleží na pořadí instrukcí, na očíslování stavů kromě počátečního a přijímajícího, znaků páskové abecedy kromě  $0, 1, \lambda$ , a binární zápis čísla stavu nebo znaku může být uvozen libovolným počtem 0. Každý TS má nekonečně mnoho různých kódů a potažmo nekonečně mnoho Gödelových čísel.

Značení  $\langle X \rangle$  označuje **kód objektu**  $X$  pomocí binárního řetězce. Značení  $\langle X_1, \dots, X_n \rangle$  označuje kód  $n$ -tice objektů  $X_1, \dots, X_n$ . Jednotlivé objekty jsou od sebe odděleny znakem ';' (respektive jeho binárním kódem).

### 1.1.4 Univerzální Turingův stroj

Vstupem univerzálního Turingova stroje  $\mathcal{U}$  je kód dvojice  $\langle M, x \rangle$ , kde  $M$  je Turingův stroj a  $x$  je řetězec.  $\mathcal{U}$  simuluje práci stroje  $M$  nad vstupem  $x$ . Výsledek práce  $\mathcal{U}(\langle M, x \rangle)$  (tj. zastavení/přijetí/zamítnutí vstupu a obsah výstupní pásky) je dán výsledkem  $M(x)$ .

Jazyku univerzálního Turingova stroje  $\mathcal{U}$  budeme říkat **univerzální jazyk** a budeme jej značit  $L_{\mathcal{U}}$ , tedy

$$L_{\mathcal{U}} = L(\mathcal{U}) = \{\langle M, x \rangle \mid x \in L(M)\}$$

$\mathcal{U}$  popíšeme jako 3-páskový, protože je to technicky jednodušší. První páška obsahuje vstup  $\mathcal{U}$ , tedy kód  $\langle M, x \rangle$ . Na druhé pášce je uložen obsah pracovní pásky  $M$ . Symboly  $X_i$  jsou zapsány jako  $(i)_B$  v blocích téže délky oddělených znakem '|'. Třetí páška obsahuje číslo aktuálního stavu  $q_i$  stroje  $M$ .

## 1.2 Rekurzivní a rekurzivně spočetné množiny.

### 1.2.1 Formální definice

**Predikát** (nebo **relace**) je množina  $n$ -tic  $R \subseteq \mathbb{N}^n, n \geq 1$ . Fakt, že  $(x_1, \dots, x_n) \in R$  budeme též označovat pomocí  $R(x_1, \dots, x_n)$ .

**Charakteristickou funkcí** predikátu  $R$  je funkce  $\lambda_R(x_1, \dots, x_n)$ , pro kterou platí, že

$$\lambda_R(x_1, \dots, x_n) \simeq \begin{cases} 1 & R(x_1, \dots, x_n) \\ 0 & \text{jinak} \end{cases}$$

**Množina** je *unární* predikát, tj.  $R \subseteq \mathbb{N}$ .

Predikát (nebo relace)  $R \subseteq \mathbb{N}^n, n \geq 1$  je **primitivně** (resp. **obecně**) **rekurzivní predikát** (**PRP**, **ORP**), pokud je jeho charakteristická funkce primitivně (resp. obecně) rekurzivní. Obecně rekurzivním predikátům a relacím budeme též říkat **rekurzivní**.

Predikát (nebo relace)  $R \subseteq \mathbb{N}^n, n \geq 1$  je **rekurzivně spočetný** (**RSP**), pokud existuje funkce  $n$  proměnných  $f_R$ , pro kterou platí, že

$$(\forall (x_1, \dots, x_n) \in \mathbb{N}^n) [f_R(x_1, \dots, x_n) \downarrow \Leftrightarrow R(x_1, \dots, x_n)]$$

to jest, hodnota funkce  $f_R$  je pro danou  $n$ -tici definovaná právě když je na ní predikát  $R$  splněný. Hodnota funkce v tomto případě není důležitá. Takové funkci budeme také říkat *částečná charakteristická funkce*.

Unární rekurzivní (resp. rekurzivně spočetný) predikát  $A \subseteq N$  budeme též nazývat **rekurzivní množinou** (resp. **rekurzivně spočetnou množinou**).

Množiny jsou ekvivalentní jazykům, protože TS je stejně silný výpošetní model jako ČRF (částečně rekurzivní funkce), pomocí kterých jsou rek. a rek. spoč. množiny definovány.

## 1.3 Algoritmicky nerozhodnutelné problémy (halting problem).

Rozhodovací problém plně popisuje jazyk *kladných instancí*. Budeme proto volně zaměňovat rozhodnutelnost problému a rozhodnutelnost jazyka.

### 1.3.1 (Částečně) rozhodnutelný jazyk

Jazyk  $L$  je **částečně rozhodnutelný**, pokud existuje Turingův stroj  $M$ , který jej přijímá (tj.  $L = L(M)$ ).

Jazyk  $L$  je **rozhodnutelný**, pokud existuje Turingův stroj  $M$ , který jej přijímá (tj.  $L = L(M)$ ) a navíc se výpočet  $M$  zastaví s každým vstupem  $x$  (tj.  $M(x)\downarrow$ ).

Pomocí  $L_e$  označíme částečně rozhodnutelný jazyk přijímaný Turingovým strojem  $M_e$ , tedy TS s Gödelovým číslem  $e$ .

*Ekvivalence definic:*

**Částečně rozhodnutelný jazyk = rekurzivně spočetný jazyk.**

**Rozhodnutelný jazyk = rekurzivní jazyk.**

#### 1.3.1.1 Základní vlastnosti rozhodnutelných jazyků

**Věta 1.3.1.** Jsou-li  $L_1$  a  $L_2$  (částečně) rozhodnutelné jazyky, pak  $L_1 \cup L_2$ ,  $L_1 \cap L_2$ ,  $L_1 \cdot L_2$ ,  $L_1^*$  jsou (částečně) rozhodnutelné jazyky.

*Důkaz.* Nechť TS  $M_1$  přijímá  $L_1$  a TS  $M_2$  přijímá  $L_2$ .

1.  $L_1 \cup L_2$ : paralelní běh  $M_1, M_2$ , pokud jeden z nich přijme, přijmu.
2.  $L_1 \cap L_2$ : paralelní běh  $M_1, M_2$ , pokud oba přijmou, přijmu.
3.  $L_1 \cdot L_2$  (konkatenace): zkusím rozdělit slovo na 2 podslova všemi způsoby a paralelním během přjmout.
4.  $L_1^*$ : zkusím rozdělit slovo na podslova všemi způsoby a zkusím přjmout podslova původním strojem  $M_1$ .

□

**Věta 1.3.2** (Postova). Jazyk  $L$  je rozhodnutelný, právě když  $L$  i  $\bar{L}$  jsou částečně rozhodnutelné jazyky.

*Důkaz.*

, $\Rightarrow$ “ Pro  $\bar{L}$  použijeme  $M_L$  s prohozenými přijímacími a nepřijímacími stavami.

, $\Leftarrow$ “ Paralelní běh  $M_L$  a  $M_{\bar{L}}$ , jeden z nich určitě někdy přijme (slovo patří buď do jazyku nebo jeho doplňku). Pokud přijme  $M_L$ , zastavíme a přijmeme; pokud přijme  $M_{\bar{L}}$ , zastavíme a odmítneme.

□

### 1.3.2 Nerozhodnutelné jazyky

Různých jazyků, např. v abecedě  $\Sigma = \{0, 1\}$ , je **nespočetně mnoho**, neboť slova ze  $\Sigma^*$  odpovídají binárnímu kódování přirozených čísel, těch je spočetně. Jazyk je nějaká podmnožina  $\Sigma^*$ , počet různých podmnožin  $\mathbb{N}$  je  $2^{|\mathbb{N}|}$ , což je nespočetně mnoho.

Částečně rozhodnutelných jazyků je **spočetně mnoho**, neboť pro č. rozhodnutelný jazyk existuje TS, tomu lze přiřadit Gödelovo číslo  $\in \mathbb{N}$ .

**Důsledek:** Existuje nekonečně mnoho jazyků (nad abecedou  $\Sigma = \{0, 1\}$ ), které nejsou ani částečně rozhodnutelné.

**Věta 1.3.3.** Diagonalační jazyk

$$L_{DIAG} = \{\langle M \rangle \mid \langle M \rangle \notin L(M)\}$$

není částečně rozhodnutelný.  $\langle M \rangle$  označuje kódování  $M$  do (binárního) řetězce.

*Důkaz.* Sporem. Nechť  $M_L$  přijímá  $L_{DIAG}$ . Pak

$$\langle M_L \rangle \in L_{DIAG} \leftrightarrow \langle M_L \rangle \notin L_{DIAG}$$

□

Jazyk  $\overline{L_{DIAG}}$  je částečně rozhodnutelný, neboť máme k dispozici univerzální Turingův stroj  $U$  a  $\langle M \rangle \in \overline{L_{DIAG}}$  právě když  $\langle M, \langle M \rangle \rangle \in L(U)$ . Není ale rozhodnutelný díky Postově větě.

**Věta 1.3.4.** *Univerzální jazyk (resp. problém)*

$$L_U = \{\langle M, x \rangle \mid x \in L(M)\}$$

je částečně rozhodnutelný, ale není rozhodnutelný.

*Důkaz.*  $L_U$  je částečně rozhodnutelný, neboť existuje univerzální Turingův stroj.

Kdyby byl rozhodnutelný, tak podle Postovy věty je  $\overline{L_U}$  částečně rozhodnutelný, tj. existuje TS  $M$  tak, že

$$\overline{L_U} = L(M) = \{\langle M, x \rangle \mid x \notin L(M)\}$$

Pomocí tohoto stroje  $M$  sestrojíme  $M'$ , který bude přijímat  $L_{DIAG}$ .  $M'$  bude se vstupem  $e$  pracovat následovně: Pokud  $e$  není syntakticky správný zápis TS, tak  $e$  odpovídá TS, který vždy odmítne, a  $M'(e)$  tedy přijme. Pokud je  $e$  syntakticky správný zápis, tak  $M'$  spustí  $M$  pro  $\langle e, e \rangle$ .  $M'$  následně přijme, odmítne, nebo se nezastaví, podle toho, jak se zachová  $M$ . Tedy máme  $M'$  který přijímá  $L_{DIAG}$ , který ale není částečně rozhodnutelný. Spor.  $\square$

**Věta 1.3.5** (Halting problem). *Problém zastavení (resp. jeho jazyk)*

$$L_{HALT} = \{\langle M, x \rangle \mid M(x) \downarrow\}$$

je částečně rozhodnutelný, ale není rozhodnutelný.

*Důkaz.* Částečná rozhodnutelnost plyne z existence univerzálního TS.

Nerozhodnutelnost ukážeme sporem. Nechť  $L_{HALT}$  je rozhodnutelný, tedy  $\overline{L_{HALT}}$  je částečně rozhodnutelný a tedy existuje  $M$  takový, že  $L(M) = \overline{L_{HALT}}$

Nyní definujme jazyk

$$L' = \{\langle M \rangle \mid M(\langle M \rangle) \uparrow\}$$

jde vlastně o diagonálou  $\overline{L_{HALT}}$ .

S pomocí stroje  $M$  sestrojíme nyní stroj  $M'$  přijímající jazyk  $L'$ .  $M'$  bude se vstupem  $e$  pracovat následovně: Pokud  $e$  není syntakticky správný zápis TS, zacykli se. V opačném případě spustí  $M(e, e)$ . Pokud  $M(e, e)$  přijme, pak přijmi, v opačném případě se  $M'$  zacyklí a nezastaví se, tj.  $M'(e) \uparrow$ . Takto zkonstruovaný stroj  $M'$  má tu vlastnost, že přijme svůj vstup právě když se zastaví. Zřejmě platí, že  $L(M') = L'$ .

Nyní se podívejme, jestli  $\langle M' \rangle$  patří do  $L'$ , nebo ne.

1. Pokud  $\langle M' \rangle \in L$ , znamená to, že se  $M'(\langle M' \rangle)$  zastaví a přijme, protože  $L' = L(M')$ , to ale současně znamená, že  $M'(\langle M' \rangle) \uparrow$  podle definice  $L'$ , což je spor.
2. Pokud  $\langle M' \rangle \notin L$ , pak podle definice  $L'$  to znamená, že  $M'(\langle M' \rangle) \downarrow$ . Podle toho, jak jsme si popsali  $M'$ , znamená to, že přijme slovo  $\langle M' \rangle$ , z toho dostaneme  $\langle M' \rangle \in L(M') = L'$ , což je však ve sporu s předpokladem.

$\square$

---

Následující sekce jsou nejspíš již nad rámec otázky. Ale souvisí s (ne)rozhodnutelností.

### 1.3.3 Vlastnosti částečně rozhodnutelných jazyků

**Věta 1.3.6.** *Pro jazyk  $L \subseteq \Sigma^*$  jsou následující tvrzení ekvivalentní:*

1.  $L$  je částečně rozhodnutelný.
2. Existuje Turingův stroj  $M_e$  splňující

$$L = \{x \in \Sigma^* \mid M_e(x) \downarrow\} (= \text{dom } \varphi_e)$$

3. Existuje rozhodnutelný jazyk  $B$  splňující

$$L = \{x \in \Sigma^* \mid (\exists y \in \Sigma^*)[\langle x, y \rangle \in B]\}$$

*Důkaz.*

1.  $\leftrightarrow$  2. triviálně.

1.  $\leftrightarrow$  3.: Stručně: Hodnotu  $y$  lze chápat jako počet kroků, v němž nějaký  $M_L$  přijímá  $x$ . Jazyk  $B$  je pak rozhodnutelný, protože simuluje běh  $M_L$  max po  $y$  kroků (nezacyklí se). Technikality vynechány.  $\square$

**Věta 1.3.7.** *Jazyk  $L \subseteq \Sigma^*$  je rozhodnutelný, právě když jeho charakteristická funkce*

$$\chi_L(x) = \begin{cases} 1 & x \in L \\ 0 & x \notin L \end{cases}$$

*je algoritmicky vyčíslitelná.*

*Důkaz.* Je-li  $L$  rozhodnutelný, upravíme příslušný  $M_L$  tak, že před přijetím/odmítnutím ještě smaže pásku a napiše na ni 1/0.

Opačně analogicky.  $\square$

### 1.3.4 Enumerátory

Nechť  $\Sigma$  je abeceda, předpokládejme, že  $<$  je ostré uspořádání na znacích. Nechť  $u, v \in \Sigma^*$  jsou dva různé řetězce. Řekneme, že  $u$  je **lexikograficky menší** než  $v$ , pokud

1. je  $u$  kratší
2. mají oba řetězce touž délku a je-li  $u[i] < v[i]$  pro  $i$  první index s  $u[i] \notin v[i]$ .

Tento fakt označíme pomocí  $u \prec v$ . Obvyklým způsobem rozšiřujeme značení i na  $u \preceq v$ ,  $u \succ v$  a  $u \succeq v$ .

**Enumerátorem** pro jazyk  $L$  je Turingův stroj  $E$ , který

- ignoruje svůj vstup,
- během výpočtu vypisuje řetězce  $w \in L$  (oddělené znakem '#') na vyhrazenou výstupní pásku
- každý řetězec  $w \in L$  je někdy vypsán TS  $E$ .
- Je-li  $L$  nekonečný,  $E$  svou činnost nikdy neskončí.

**Věta 1.3.8.** *Jazyk  $L$  je částečně rozhodnutelný, právě když pro něj existuje enumerátor  $E$ .*

*Jazyk  $L$  je rozhodnutelný, právě když pro něj existuje enumerátor  $E$ , který navíc vypisuje prvky  $L$  v lexikografickém pořadí.*

*Důkaz.* Existuje-li enumerátor, pak zodpovíme  $x \in L$  tak, že rozbehneme enumerátor a vypíše-li někdy  $x$ , přijmeme.

Existuje-li enumerátor vypisující v lexikografickém pořadí, pak v okamžiku překročení pozice, kde by mělo být  $x$ , zastavíme a odmítнемe ( $\rightarrow$  rozhodnutelnost).

Máme-li rozhodnutelný jazyk, sestrojíme enumerátor takto: generujeme slova v dané abecedě v lexikografickém pořadí, pro každé spustíme TS příslušného jazyka, pokud přijme, vypíšeme slovo, pokud odmítne, nevypisujeme.

Máme-li částečně rozhodnutelný jazyk, sestrojíme enumerátor takto: máme proměnné  $d =$  délka slova a  $k =$  počet kroků TS. Generujeme slova v dané abecedě v lexikografickém pořadí do určité délky  $d$ , pro každé spustíme TS příslušného jazyka, *jehož chod omezíme na  $k$  kroků*, pokud přijme, vypíšeme slovo. Iterativně zvyšujeme  $k, d$ .  $\square$

### 1.3.5 Převoditelnost, úplné jazyky

Jazyk  $A$  je  **$m$ -převoditelný** na jazyk  $B$  (což označíme pomocí  $A \leq_m B$ ), pokud existuje totální vyčíslitelná funkce  $f$  splňující

$$(\forall x \in \Sigma^*)[x \in A \Leftrightarrow f(x) \in B]$$

Jazyk  $A$  je  **$m$ -úplný**, pokud je  $A$  částečně rozhodnutelný a každý částečně rozhodnutelný jazyk  $B$  je na něj  $m$ -převoditelný.

Podobně definujeme i **1-převoditelnost** a **1-úplnost**, avšak zde navíc chceme, aby funkce  $f$  byla prostá.

Převoditelnost  $\leq_m$  je **reflexivní** a **tranzitivní** relace (**kvaziusporádání**).

Pokud  $A \leq_m B$  a  $B$  je (částečně) rozhodnutelný jazyk, pak totéž lze říct o  $A$ .

Pokud  $A \leq_m B$ ,  $B$  je částečně rozhodnutelný jazyk a  $A$  je  $m$ -úplný jazyk, pak  $B$  je též  $m$ -úplný.

### Věta 1.3.9. Jazyky

$$\begin{aligned} L_{\mathcal{U}} &= \{\langle M, x \rangle \mid x \in L(M)\} \\ K_0 &= \{\langle M, x \rangle \mid M(x) \downarrow\} (= L_{HALT}) \\ K &= \{\langle M \rangle \mid M(\langle M \rangle) \downarrow\} \end{aligned}$$

jsou  $m$ -úplné. Zvláště pak jede o jazyky částečně rozhodnutelné, které nejsou rozhodnutelné.

*Důkaz.* Převedeme  $L_{\mathcal{U}} \leq_m K_0 \leq_m K$ .

$L_{\mathcal{U}}$  je  $m$ -úplný: pro libovolný částečně rozhodnutelný jazyk  $L$  vezmu  $M_L$  a odsimuluji na  $\mathcal{U}$  s  $\langle M, x \rangle$ ,  $\mathcal{U}$  přijme, pokud  $M$  přijme.

$L_{\mathcal{U}} \leq_m K_0$ : pokud  $M$  přijímá  $x$ , pak přijmu a zastavím, jinak cyklím (pokud  $\mathcal{U}$  odmítne nebo cyklí).

$K_0 \leq_m K$ : z  $\langle M, x \rangle$  vytvoříme nový TS  $M_{(M,x)}$ , který pro jakýkoliv vstup tento vstup smaže, přepíše ho na  $x$  a spustí na něm  $M$ . ( $M_{(M,x)}$  prostě za každých okolností spustí  $M$  nad  $x$ ), pak

$$\langle M_{(M,x)} \rangle \in K \Leftrightarrow \langle M, x \rangle \in K_0$$

□

### 1.3.6 Riceova věta

**Věta 1.3.10** (Riceova (jazyky)). Nechť  $C$  je třída částečně rozhodnutelných jazyků a položme  $L_C = \{\langle M \rangle \mid L(M) \in C\}$ . Potom je jazyk  $L_C$  rozhodnutelný, právě když je třída  $C$  buď prázdná nebo obsahuje všechny částečně rozhodnutelné jazyky.

*Důkaz.* Sporem. Mějme netriviální třídu  $C$  (neobsahuje všechny částečně rozhodnutelné jazyky ani není prázdná). Nechť  $M_C$  je TS,  $L_C = L(M_C)$ . Vzhledem k tomu, že  $L_C$  je rozhodnutelný, tak o libovolném TS  $M$  (zadaného kódem) můžeme pomocí  $M_C$  říci, zda jazyk jím přijímaný do třídy  $C$  patří. Zvolme si nyní nějaký jazyk z  $C$ , přijímaný TS  $N$  (předpokládáme, že  $C$  neobsahuje prázdný jazyk, pokud ano, zvolíme  $C = \text{dopl}\check{n}\text{k}C$ ). Vytvořme si nyní takový stroj  $M$ , který na vstup pustí svůj vlastní kód  $\langle M \rangle$  a pokud se zastaví, tak spustí stroj  $N$  s původním vstupem (čili, pokud  $M(\langle M \rangle) \downarrow$ , tak  $L(M) = L(N)$ ). Pokud se tedy  $M$  zastaví, tak  $\langle M \rangle \in L_C$ , pokud se  $M$  nezastaví, tak  $\langle M \rangle \notin L_C$ , čili pomocí  $M_C$  bychom mohli rozhodovat jazyk  $K = \{\langle M \rangle \mid M(\langle M \rangle) \downarrow\}$ , který je nerozhodnutelný. □

Z Riceovy věty plyne, že následující jazyky nejsou rozhodnutelné:

$$\begin{aligned} K_1 &= \{\langle M \rangle \mid L(M) \notin \emptyset\} \\ \text{Fin} &= \{\langle M \rangle \mid L(M) \text{ je konečný jazyk}\} \\ \text{Cof} &= \{\langle M \rangle \mid \overline{L(M)} \text{ je konečný jazyk}\} \\ \text{Inf} &= \{\langle M \rangle \mid L(M) \text{ je nekonečný jazyk}\} \\ \text{Dec} &= \{\langle M \rangle \mid L(M) \text{ je rozhodnutelný jazyk}\} \\ \text{Tot} &= \{\langle M \rangle \mid L(M) = \Sigma^*\} \\ \text{Reg} &= \{\langle M \rangle \mid L(M) \text{ je regulární jazyk}\} \end{aligned}$$

## 1.4 Nedeterministický výpočetní model.

Nedeterministický Turingův stroj (NTS) je pětice  $M = (Q, \Sigma, \delta, q_0, F)$ , kde  $Q, \Sigma, q_0, F$  mají týž význam jako u „obyčejného“ deterministického Turingova stroje (DTS). Rozdíl oproti DTS je v přechodové funkci, nyní

$$\delta : Q \times \Sigma \mapsto \mathcal{P}(Q \times \Sigma \times \{L, N, R\})$$

Možné představy:

- NTS  $M$  v každém kroku „uhodne“ nebo „vybere“ správnou instrukci.
- NTS  $M$  vykonává všechny možné instrukce současně a nachází se během výpočtu ve více konfiguracích současně.

*Nedeterministický Turingův stroj není reálný výpočetní model ve smyslu silnější Churchovy-Turingovy teze.*

Výpočet NTS  $M$  nad slovem  $x$  je posloupnost konfigurací  $C_0, C_1, C_2, \dots$ , kde  $C_0$  je počáteční konfigurace a z  $C_i$  do  $C_{i+1}$  lze přejít pomocí přechodové funkce  $\delta$ . Výpočet je **přijímající**, pokud je konečný a v poslední konfiguraci výpočtu se  $M$  nachází v přijímajícím stavu.

Slovo  $x$  je **přijato** NTS  $M$  pokud existuje přijímající výpočet  $M$  nad  $x$ . **Jazyk** slov přijímaných NTS  $M$  označíme pomocí  $L(M)$ .

### 1.4.1 Časová a prostorová složitost NTS

Nechť  $M$  je nedeterministický Turingův stroj a nechť  $f : \mathbb{N} \mapsto \mathbb{N}$  je funkce.

Řekneme, že  $M$  **pracuje v čase**  $f(n)$ , pokud *každý* výpočet  $M$  nad *libovolným* vstupem  $x$  délky  $|x| = n$  skončí po provedení nejvýše  $f(n)$  kroků.

Řekneme, že  $M$  **pracuje v prostoru**  $f(n)$ , pokud *každý* výpočet  $M$  nad *libovolným* vstupem  $x$  délky  $|x| = n$  využije nejvýše  $f(n)$  buněk pracovní pásky.

Nechť  $f : \mathbb{N} \mapsto \mathbb{N}$  je funkce, potom definujeme třídy:

**NTIME**( $f(n)$ ) – třída jazyků přijímaných nedeterministickými TS, které pracují v čase  $O(f(n))$ .

**NSPACE**( $f(n)$ ) – třída jazyků přijímaných nedeterministickými TS, které pracují v prostoru  $O(f(n))$ .

Třída **NP** je třída jazyků přijímaných nedeterministickými Turingovými stroji v polynomiálním čase, tj.

$$NP = \bigcup_{k \in \mathbb{N}} NTIME(n^k)$$

*Pozn.: Třída NP se definuje ještě jinak, viz další sekce. Ekvivalence definic se dokazuje, důkaz v další sekci, ale mohl by se hodit i sem.*

## 1.5 Základní třídy složitosti a jejich vztahy.

Zatímco vyčíslitelnost řeší, zda vůbec lze nějaký problém řešit, *složitost* se zabývá tím, jak efektivně jej lze řešit, a to především z hlediska času a prostoru. Formálně nyní odlišíme 2 typy řešených úkolů: *rozhodovací problémy* a *(optimalizační) úlohy*.

V **rozhodovacím problému** se ptáme, zda daná **instance**  $x$  splňuje danou podmínu. Odpověď je **typu ano/ne**. Rozhodovací problém formalizujeme jako **jazyk kladných instancí**  $L \subseteq \Sigma^*$  a otázku, zda  $x \in L$ .

V **úloze** pro danou **instanci**  $x$  hledáme  $y$ , které splňuje určitou podmínu. Odpověď je zde **y nebo informace o tom, že žádné vhodné y neexistuje**. Úlohu formalizujeme jako **relaci**  $R \subseteq \Sigma^* \times \Sigma^*$ .

V **optimalizační úloze** navíc požadujeme, aby hodnota  $y$  byla maximální nebo minimální vzhledem k nějaké míře.

### 1.5.1 Základní třídy složitosti

Nechť  $M$  je (deterministický) Turingův stroj, který se zastaví na každém vstupu a nechť  $f : \mathbb{N} \rightarrow \mathbb{N}$  je funkce.

Řekneme, že  $M$  pracuje v čase  $f(n)$ , pokud výpočet  $M$  nad libovolným vstupem  $x$  délky  $|x| = n$  skončí po provedení nejvýše  $f(n)$  kroků.

Řekneme, že  $M$  pracuje v prostoru  $f(n)$ , pokud výpočet  $M$  nad libovolným vstupem  $x$  délky  $|x| = n$  využije nejvýše  $f(n)$  buněk pracovní pásky.

#### 1.5.1.1 Základní deterministické třídy složitosti

Nechť  $f : \mathbb{N} \rightarrow \mathbb{N}$  je funkce, potom definujeme třídy:

**TIME( $f(n)$ )** – třída jazyků přijímaných Turingovými stroji, které pracují v čase  $O(f(n))$ .

**SPACE( $f(n)$ )** – třída jazyků přijímaných Turingovými stroji, které pracují v prostoru  $O(f(n))$ .

Často se místo TIME používá DTIME a místo SPACE se používá DSPACE, aby se zdůraznilo, že jde o deterministické TS.

#### 1.5.1.2 Význačné deterministické třídy složitosti

Třída problémů řešitelných v polynomiálním čase:

$$P = \bigcup_{k \in \mathbb{N}} \text{TIME}(n^k)$$

Třída problémů řešitelných v polynomiálním prostoru:

$$\text{PSPACE} = \bigcup_{k \in \mathbb{N}} \text{SPACE}(n^k)$$

Třída problémů řešitelných v exponenciálním čase:

$$\text{EXPTIME} = \bigcup_{k \in \mathbb{N}} \text{TIME}(2^{n^k})$$

Polynomiální třídy jsou význačné díky několika hezkým vlastnostem polynomů:

- nerostou příliš rychle
- jsou uzavřeny na skládání
- Silnější verze *Churchovy-Turingovy teze* tvrdí, že každý „rozumný a obecný“ výpočetní model lze na Turingově stroji simulovat s polynomiálním zpomalením či polynomiálním zvětšením potřebného prostoru.

Z toho plyne, že třídy P a PSPACE jsou nezávislé na použitém výpočetním modelu (pokud lze tento simulovat na TS s polynomiálním zpomalení/náruštem prostoru). Rozhodně jsou nezávislé na tom, v jakém běžném programovacím jazyce algoritmus implementujeme.

Třída P tedy zhruba odpovídá třídě problémů, které lze řešit na počítači v rozumném čase. Opatrně však na big-O notaci, do které se můžou schovat i velké konstanty a polynomiální algoritmus pak může být v reálu pomalý.

#### 1.5.1.3 Třída NP

Abychom mohli definovat třídu NP, potřebujeme nejprve následující definici:

**Verifikátorem** pro jazyk  $L$  je algoritmus  $V$ , pro který platí, že

$$L = \{x \mid (\exists y)[V \text{ přijme } (x, y)]\}$$

Řetězec  $y$  zveme také **certifikátem**  $x$ .

Časovou složitost verifikátoru měříme vzhledem k  $|x|$ . **Polynomiální verifikátor** je takový, který pracuje v polynomiálním čase vzhledem k  $|x|$ . Pokud polynomiální verifikátor  $V$  přijímá  $(x, y)$ , pak  $y$  má nutně délku polynomiální vzhledem k  $x$ . Řetězec  $y$  je pak zván **polynomiálním certifikátem**  $x$ .

**Třída NP** je třídou jazyků, které mají *polynomiální verifikátory*. Odpovídá třídě úloh, u nichž jsme schopni v polynomiálním čase ověřit, že daný řetězec  $y$  je řešením, i když jej *nejsme nutně schopni v polynomiálním čase najít*.

Třídu NP je možno také definovat jako třídu jazyků přijímaných nedeterministickými Turingovými stroji v polynomiálním čase, tj.

$$\bigcup_{k \in \mathbb{N}} \text{NTIME}(n^k)$$

Nedeterminismus zde odpovídá „hádání“ správného certifikátu  $y$  vstupu  $x$ .

**Věta 1.5.1.** *Obě výše uvedené definice třídy NP jsou ekvivalentní, tj.*

$$NP = \bigcup_{k \in \mathbb{N}} \text{NTIME}(n^k)$$

*Důkaz.* Ukážeme, že  $L \in NP \Leftrightarrow L \in \bigcup_{k \in \mathbb{N}} \text{NTIME}(n^k)$ :

„ $\Rightarrow$ “ Předpokládejme nejprve, že jazyk  $L \in NP$ , to znamená, že existuje polynom  $p$  a polynomiální verifikátor  $B \in P$ , pro které platí, že  $x \in L$  právě když existuje  $y$ ,  $|y| \leq p(|x|)$ , pro které  $(x, y) \in B$ . NTS  $M_L$ , který bude přijímat  $L$ , bude pracovat ve dvou fázích. V první fázi zapíše na vstupní pásku za slovo  $x$  slovo  $y$ , tato fáze je nedeterministická a pro každé slovo  $y$ , takové že  $|y| \leq p(|x|)$  existuje výpočet  $M_L$ , který jej napíše. Na zápis  $y$  stačí čas  $p(|x|)$ . Ve druhé fázi bude  $M_L$  simulovat práci TS  $M_B$ , který rozpoznává jazyk  $B$ , na vstupu  $(x, y)$ , přičemž přijme, pokud  $(x, y) \in B$ . Zřejmě  $L = L(M_B)$  a  $M_B$  pracuje v polynomiálním čase (neboť  $B$  je *polynomiální verifikátor*).

„ $\Leftarrow$ “ Nyní předpokládejme, že  $L \in \bigcup_{k \in \mathbb{N}} \text{NTIME}(n^k)$ . To znamená, že  $x \in L$  právě když existuje polynomiálně dlouhý výpočet nedeterministického Turingova stroje  $M$ , kde  $L = L(M)$ , jenž  $x$  přijme. V každém kroku tohoto výpočtu vybírá  $M$  z několika možných instrukcí, nechť řetězec  $y$  kóduje právě to, které instrukce byly v každém kroku vybrány. Řetězec  $y$  má délku nejvýš  $p(|x|)$  pro nějaký polynom  $p$ , protože  $M$  pracuje v polynomiálním čase a možností, jak pokračovat z dané konfigurace podle přechodové funkce je jen konstantně mnoho (protože máme konečnou abecedu). Simulací  $M$  s použitím instrukcí daných dle  $y$  můžeme deterministicky ověřit, zda  $y$  kóduje přijímající výpočet. Řetězec  $y$  tedy může sloužit jako polynomiálně dlouhý certifikát kladné odpovědi a DTS simulující  $M$  s pomocí instrukcí daných  $y$  je polynomiální verifikátor.  $\square$

#### 1.5.1.4 Modely TS s menším než lineárním prostorem

Ač se zdá na první pohled nesmyslné uvažovat TS pracující v prostoru menším než  $O(n)$ , tj. menším než je samotná délka vstupu, po menších úpravách je to možné. Model TS s menším než lineárním prostorem vypadá takto:

- uvažujeme vícepáskový TS: vstupní pánska je pouze pro čtení, pracovní pásky jsou pro čtení i zápis, výstupní pánska je pouze pro zápis a pohybuje se jen vpravo
- *do prostoru se počítá pouze obsah pracovních pásek*
- součástí konfigurace je stav, poloha hlavy na vstupní pánsce, polohy hlav na pracovních páskách a obsah pracovních pásek
- konfigurace *neobsahuje* vstupní slovo

S pomocí tohoto modelu TS můžeme definovat následující třídy jazyků:

$$L = \text{SPACE}(\log_2 n)$$

$$NL = \text{NSPACE}(\log_2 n)$$

$$\text{NPSPACE} = \bigcup_{k \in \mathbb{N}} \text{NSPACE}(n^k)$$

### 1.5.1.5 Přehled všech zmíněných tříd jazyků

(D)TIME( $f(n)$ )	jazyky přijímané DTS v čase $f(n)$
(D)SPACE( $f(n)$ )	jazyky přijímané DTS v prostoru $f(n)$
NTIME( $f(n)$ )	jazyky přijímané NTS v čase $f(n)$
NSPACE( $f(n)$ )	jazyky přijímané NTS v prostoru $f(n)$
P	$\bigcup_{k \in \mathbb{N}} \text{TIME}(n^k)$
NP	$\bigcup_{k \in \mathbb{N}} \text{NTIME}(n^k)$ / jazyky s polynomiálními verifikátory
PSPACE	$\bigcup_{k \in \mathbb{N}} \text{SPACE}(n^k)$
NPSPACE	$\bigcup_{k \in \mathbb{N}} \text{NSPACE}(n^k)$
EXPTIME	$\bigcup_{k \in \mathbb{N}} \text{TIME}(2^{n^k})$
L	SPACE( $\log_2 n$ )
NL	NSPACE( $\log_2 n$ )

Tabulka 1.2: Přehled tříd jazyků

### 1.5.2 Vztahy mezi třídami

**Věta 1.5.2.** Pro každou funkci  $f : \mathbb{N} \mapsto \mathbb{N}$  platí, že  $\text{TIME}(f(n)) \subseteq \text{SPACE}(f(n))$ .

*Důkaz.* Během své práce nad vstupem  $x$  nestihne TS  $M$  popsat víc buněk, než kolik na to má času, pracuje-li tedy  $M$  v čase  $f(n)$ , nestihne popsat víc než  $f(n)$  buněk. Tvrzení platí i v případě, kdy  $f(n) < n$ , i když v tom případě musíme uvažovat jiný model TS, který umožňuje nepočítat do prostoru velikost vstupu.  $\square$

**Věta 1.5.3.** Pro každou funkci  $f : \mathbb{N} \mapsto \mathbb{N}$  platí

$$\text{TIME}(f(n)) \subseteq \text{NTIME}(f(n)) \subseteq \text{SPACE}(f(n)) \subseteq \text{NSPACE}(f(n))$$

*Důkaz.* První a třetí inkluze platí triviálně (DTS je speciální případ NTS).

Druhá inkluze, tj.  $\text{NTIME}(f(n)) \subseteq \text{SPACE}(f(n))$ : Chceme simulovat NTS  $M$  pracující v čase  $f(n)$  pomocí DTS  $M'$ .  $M$  se v každém kroku nedeterministicky rozhoduje, má však jen konstatně mnoho možností  $c_M$ .  $M'$  bude mít string kódující všechny možné volby, ten zabere prostor  $c_M \cdot f(n)$ . Tento string se bude používat jako look-up tabulka kdykoliv  $M$  provádí nedeterministickou volbu. Pro každou volbu tedy  $M'$  simuluje  $M$  a pokud  $M$  přijme, tak přijme i  $M'$ . Celkově  $M'$  potřebuje  $c_M f(n) + f(n)$  místa, tedy  $M' \in \text{SPACE}(f(n))$ .  $\square$

**Věta 1.5.4.** Nechť  $f(n)$  je funkce, pro kterou platí  $f(n) \geq \log_2 n$ . Pro každý jazyk  $L \in \text{NSPACE}(f(n))$  platí, že  $L \in \text{TIME}(2^{c_L f(n)})$ , kde  $c_L$  je konstanta závislá na jazyku  $L$ .

*Důkaz.* Nechť  $M = (Q, \Sigma, \delta, q_0, F)$ . Konfigurace se skládá ze slova na pásce, polohy hlavy v rámci tohoto slova a stavu, v němž se stroj  $M$  nachází. Délka slova na pásce je omezená  $f(n)$ , počet různých poloh hlavy v rámci tohoto slova je  $f(n)$  a počet stavů je  $|Q|$ . Počet konfigurací je tedy shora omezen pomocí

$$|\Sigma|^{f(n)} \cdot f(n) \cdot |Q| = 2^{f(n) \log_2 |\Sigma|} 2^{\log_2 f(n)} 2^{\log_2 |Q|} = 2^{f(n) \log_2 |\Sigma| + \log_2 f(n) + \log_2 |Q|} \leq 2^{f(n)(\log_2 |\Sigma| + 1 + \log_2 |Q|)}$$

Horní odhad na počet konfigurací je současně i horním odhadem na časovou složitost, neboť přijímací výpočet se v každé konfiguraci octne nejvýše jednou. Pokud bychom se totiž do nějaké dostali vícekrát, pak jsme se nutně octli v cyklu a výpočet tedy nikdy neskončí, což je pro slova z  $L$  spor.

Stačí tedy zvolit  $c_M = (\log_2 |\Sigma| + \log_2 |Q| + 1)$ .  $\square$

**Věta 1.5.5.** Platí následující inkluze:

$$L \subseteq P \subseteq NP \subseteq PSPACE \subseteq NPSPACE \subseteq EXPTIME$$

*Důkaz.* Jednotlivé inkluze:

**P ⊆ NP, PSPACE ⊆ NPSPACE** : Přímo z definice.

**NP ⊆ PSPACE** : Plyne z 1.5.3.

**L ⊆ P, NPSPACE ⊆ EXPTIME** : Plyne z 1.5.4.

□

**Věta 1.5.6** (Savičova). Pro každou funkci  $f(n) \geq \log_2 n$  vyčíslitelnou v prostoru  $O(f(n))$  platí, že:

$$NSPACE(f(n)) \subseteq SPACE(f^2(n))$$

*Důkaz.* Díky 1.5.4 víme, že každý  $M$  pracující v prostoru  $f(n)$  má až  $2^{c_M f(n)}$  různých konfigurací. Chceme simulovat NTS  $M$  nějakým DTS, ale standardní techniky procházení stromu konfigurací (do šírky či do hloubky) jistě zaberou příliš mnoho prostoru. Půjdeme na to jinak.

*Konfigurační strom* NTS (strom všech výpočtů) zredukujeme na *konfigurační graf* tím způsobem, že každá konfigurace zde bude právě jednou. Vrcholy tedy tvoří jednotlivé konfigurace a hrana se mezi dvěma vrcholy nachází právě tehdy, pokud lze přejít z jedné příslušné konfigurace do druhé podle přechodové funkce  $M$ . Na uložení celého grafu nemáme prostor, máme ho zadaný jen implisitně pomocí funkce  $hrana(i, j)$ , která nám řekne, zda lze přejít z  $i$  do  $j$

Hledáme cestu z  $K_0^x$  (poč. konfigurace nad slovem  $x$ ) do  $K_F^x$  (přijímající konfigurace, BÚNO každá je jen jedna (jediný přijímací stav, smazaná páska, hlava na začátku)).  $M$  přijme  $x$  pokud existuje orientovaná cesta z  $K_0^x$  do  $K_F^x$ . Chceme tedy algoritmus, který existenci této cesty ověří.

Zavedeme funkci *Dosažitelná*( $i, K_1, K_2$ ), která zjistí, zda z konfigurace  $K_1$  je konfigurace  $K_2$  dosažitelná v nejvýše  $2^i$  krocích Turingova stroje  $M$ .

```

1   Dosazitelna(i, K1, K2):
2       if i = 0:
3           if (K1, K2) in E or K1 == K2:
4               return true
5           else
6               return false
7       for K in V:
8           if Dosazitelna(i-1, K1, K) and Dosazitelna(i-1, K, K2):
9               return true
10      return false

```

Víme, že počet vrcholů je nejvýše  $2^{c_M f(n)}$  a tedy voláním *Dosažitelná*( $c_M f(n), K_0^x, K_F^x$ ) na grafu  $G_{M,x}$  zjistíme, zda existuje přijímací výpočet.

Zbývá spočítat prostorovou složitost funkce *Dosažitelná*. Pro otestování existence hrany na řádku 3 nám stačí přechodová funkce  $\delta$  stroje  $M$ , která je konstatně velká a nezávislá na  $n$ . V cyklu na řádcích 7-9 potřebujeme generovat všechny konfigurace. Na uložení jedné konfigurace stačí  $c_M f(n)$  bitů, stačí tedy generovat všechny binární řetězce této délky a vybírat si ty, které kódují konfigurace.

Jedna instance funkce *Dosažitelná* tedy vyžaduje prostor pro uložení  $K_1, K_2, K$  a  $i$ , na všechny tyto proměnné stačí prostor  $c_M f(n)$ . Navíc potřebujeme jen bity pro uložení odpovědí z rekursivních volání, které už prostorovou složitost nenaruší.

Abychom mohli v cyklu generovat konfigurace  $K$ , musíme však vědět, jak velký prostor rezervovat pro jednu konfiguraci, potřebujeme tedy umět označit  $c_M f(n)$  buněk, máme-li vstup  $x$  délky  $n$ , a při tom si musíme vystačit s prostorem  $O(f(n))$ . To platí, je-li funkce  $f$  vyčíslitelná<sup>1</sup>

<sup>1</sup>Funkce je  $f$  vyčíslitelná v prostoru  $O(f(n))$ , pokud k ní existuje Turingův stroj  $M_f$ , který pracuje v prostoru  $O(f(n))$  a při výpočtu nad vstupem  $x = 1^n$  je po ukončení jeho činnosti na výstupu zapsán řetězec  $y = 1^f(n)$ .

Unární kódování přirozeně odpovídá tomu, co chceme: rezervovat buňky pro výsledek funkce; tj. zapíšeme jedničky na tolik polí, kolik chceme rezervovat. Ne každá funkce  $f$  je vyčíslitelná v prostoru  $O(f(n))$ , nicméně pro běžné funkce to platí.

Hloubka rekurze je omezena pomocí  $c_M f(n) = O(f(n))$ , protože to je počáteční hodnota  $i$ , které předáváme funkci jako parametr a v každém dalším voláním toto  $i$  snížíme o jednu.

Dohromady tedy dostaneme, že celkový prostor, který potřebujeme, je velký  $O(f(n) \cdot f(n)) = O(f^2(n))$ . Protože funkce Dosažitelná je deterministická, vyžaduje její volání *deterministický* prostor  $O(f^2(n))$ . Nebylo by také obtížné na základě této funkce vytvořit DTS  $M'$ , který by vyžadoval týž prostor a přijímal jazyk  $L$ . Z toho plyne, že  $L \in \text{SPACE}(O(f^2(n)))$ .  $\square$

**Důsledek:**  $\text{PSPACE} = \text{NPSPACE}$ .

## 1.6 Věty o hierarchii.

### 1.6.1 Věta o deterministické prostorové hierarchii

Funkci  $f : \mathbb{N} \mapsto \mathbb{N}$ , kde  $f(n) \geq \log n$ , nazveme **prostorově konstruovatelnou**, je-li funkce, která zobrazuje  $1^n$  na binární reprezentaci  $f(n)$  vyčíslitelná v prostoru  $O(f(n))$ .

**Věta 1.6.1** (o deterministické prostorové hierarchii). *Pro každou prostorově konstruovatelnou funkci  $f : \mathbb{N} \mapsto \mathbb{N}$  existuje jazyk  $L$ , který je rozhodnutelný v prostoru  $O(f(n))$ , nikoli však v prostoru  $o(f(n))$ .*

*Důkaz.* Připomenutí big O a little O notace:

$$\begin{aligned} f(n) \in O(g(n)) &\Leftrightarrow \exists c > 0, \exists n_0 > 0, \forall n > n_0 : 0 \leq f(n) \leq c \cdot g(n) \\ f(n) \in o(g(n)) &\Leftrightarrow \forall c > 0, \exists n_0 > 0, \forall n > n_0 : 0 \leq f(n) \leq c \cdot g(n) \end{aligned}$$

Chceme najít jazyk  $L$ , který je rozhodnutelný v prostoru  $O(f(n))$ , ale ne v prostoru  $o(f(n))$ . Definujeme

$$L = \{(\langle M \rangle, 10^k) \mid M \text{ nepřijímá (odmítá)} (\langle M \rangle, 10^k) \text{ v prostoru } \leq f(|(\langle M \rangle, 10^k)|)$$

Zápis  $10^k$  reprezentuje slovo skládající se z jedničky a  $k$  nul, tj. 100...0.

#### 1. $L$ je rozhodnutelný v prostoru $O(f(n))$ :

Algoritmus pro rozhodnutí  $L$  je tento:

- (a) Pro vstup  $x, |x| = n$  spočti  $f(n)$  (díky **prostorové konstruovatelnosti**) a označ  $f(n)$  buněk na pásce. Pokud se algoritmus pokusí označit více než  $f(n)$  buněk, *odmítни*.
- (b) Pokud  $x$  není tvaru  $(\langle M \rangle, 10^k)$  pro nějaký validní  $M$ , *odmítни*.
- (c) Simuluj běh  $M$  na vstupu  $x$ . Pokud  $M$  přesáhne  $f(n)$  prostoru nebo  $2^{f(n)}$  času, *odmítni*.
- (d) Pokud  $M$  přijme, *odmítni*. Jinak *přijmi*.

Časový limit v bodě (c) je pro ošetření situace, kdy sice  $M$  nepřekročí povolený prostor, ale zacyklí se a nezastaví. Pro odpočítávání  $2^{f(n)}$  času nám stačí  $f(n)$  prostoru, čili nepřekročíme povolený prostor.

#### 2. $L$ je nerozhodnutelný v prostoru $o(f(n))$ :

Dokážeme sporem. Nechť  $L$  je *rozhodnutelný* v  $o(f(n))$ , pak existuje TS  $M$ , který jej rozhoduje. Uvažme slovo  $w = (\langle M \rangle, 10^k)$ , pro nějaké dostatečně velké  $k$ . Platí:

$$w \in L \Leftrightarrow M \text{ přijme } w \Leftrightarrow w \notin L \text{ (spor)}$$

Zde se uplatní slovo  $10^k$  uměle přidané ke vstupu. Kdyby tam nebylo, tak je délka vstupu pouze  $|M|$  a tedy i práce  $M$  by se musela vejít do prostoru  $O(|M|)$ . Což by mohlo být komplikované, jelikož  $M$  si potřebuje v paměti držet celé vstupní slovo zapsané v nějaké své interní abecedě plus další složky určující konfiguraci  $M$ . Umělé prodloužení vstupu poskytne  $M$  více prostoru a přitom samo nároky nijak moc nezvýší. Když tedy řekneme,

že použijeme slovo  $w = (\langle \overline{M} \rangle, 10^k)$  pro nějaké dostatečně velké  $k$ , tak tím myslíme, že  $M$  zajistíme dostatek prostoru pro práci a tedy odmítnutí nebude způsobeno překročením prostoru.

Celkově je princip důkazu založen na odlišnosti při zpracování slova  $(\langle M \rangle, 10^k)$ . Uměle přidané slovo nám zajistí, že  $M$  při výpočtu nepřekročí povolený prostor.  $\square$

### Důsledek 1.6.1.

1. Jsou-li  $f_1, f_2 : \mathbb{N} \mapsto \mathbb{N}$  funkce, pro které platí, že  $f_1(n) \in o(f_2(n))$  a  $f_2$  je prostorově konstruovatelná, potom

$$\text{SPACE}(f_1(n)) \subsetneq \text{SPACE}(f_2(n))$$

2. Pro každá dvě reálná čísla  $0 \leq \epsilon_1 < \epsilon_2$  platí, že

$$\text{SPACE}(n^{\epsilon_1}) \subsetneq \text{SPACE}(n^{\epsilon_2})$$

3.  $NL \subsetneq PSPACE \subsetneq EXPSPACE = \bigcup_{k \in \mathbb{N}} \text{SPACE}(2^{n^k})$

### 1.6.2 Věta o časové prostorové hierarchii

Funkci  $f : \mathbb{N} \mapsto \mathbb{N}$ , kde  $f(n) \in \Omega(n \log n)$ , nazveme **časově konstruovatelnou**, je-li funkce, která zobrazuje  $1^n$  na binární reprezentaci  $f(n)$  vyčíslitelná v čase  $O(f(n))$ .

**Věta 1.6.2** (o deterministické časové hierarchii). *Pro každou časově konstruovatelnou funkci  $f : \mathbb{N} \mapsto \mathbb{N}$  existuje jazyk  $A$ , který je rozhodnutelný v čase  $O(f(n))$ , nikoli však v čase  $o(\frac{f(n)}{\log f(n)})$ .*

*Důkaz.*<sup>2</sup>

Oproti prostorovém případu je zde navíc faktor  $\log f(n)$ . To je způsobeno tím, že TS provádějící simulaci v důkazu této věty musí provádět práci navíc, aby si pamatoval konfiguraci stroje, který simuluje. Tato práce nepotřebuje prostor navíc, nýbrž čas navíc, konkrétně řádu  $\log f(n)$ .

Jinak je důkaz podobný. Chceme najít jazyk  $L$ , který je rozhodnutelný v čase  $O(f(n))$ , ale ne v čase  $o(f(n)/\log f(n))$ . Definujeme

$$L = \{(\langle M \rangle, 10^k) \mid M \text{ nepřijímá (odmítá)} (\langle M \rangle, 10^k) \text{ v čase } \leq \frac{f(n)}{\log f(n)}, \text{ kde } n = |(\langle M \rangle, 10^k)|\}$$

Zápis  $10^k$  reprezentuje slovo skládající se z jedničky a  $k$  nul, tj.  $100\dots0$ .

1.  **$L$  je rozhodnutelný v čase  $O(f(n))$ :**

Algoritmus pro rozhodnutí  $L$  je tento:

- (a) Pro vstup  $x, |x| = n$  spočti  $f(n)$  (díky **časové konstruovatelnosti**) a označ  $f(n)$  buněk na pásce. Pokud se algoritmus pokusí označit více než  $f(n)$  buněk, *odmítni*.
- (b) Pokud  $x$  není tvaru  $(\langle M \rangle, 10^k)$  pro nějaký validní  $M$ , *odmítni*.
- (c) Simuluj běh  $M$  na vstupu  $x$ . **Pokud  $M$  nedoběhne v  $\frac{f(n)}{\log f(n)}$  krocích, odmítni.**
- (d) Pokud  $M$  přijme, *odmítni*. Jinak *prijmi*.

Pro každý krok simulace  $M$  musí algoritmus snížit interní čítač kroků o jedna. Tento čítač je inicializován na hodnotě  $\frac{f(n)}{\log f(n)}$  a jeho binární zakódování má tedy délku  $O(\log f(n))$ . Jeho snížení v každém kroku simulace zabere dalších  $O(\log f(n))$ . Takže celková časová náročnost simulace stroje s časovou složitostí  $O(\frac{f(n)}{\log f(n)})$  je skutečně  $O(f(n))$ .

2.  **$L$  je nerozhodnutelný v  $o(\frac{f(n)}{\log f(n)})$ :**

Dokážeme sporem. Nechť  $L$  je rozhodnutelný v  $o(\frac{f(n)}{\log f(n)})$ , pak existuje TS  $M$ , který jej rozhoduje. Uvažme slovo  $w = (\langle M \rangle, 10^k)$ , pro nějaké dostatečně velké  $k$ . Platí:

$$w \in L \Leftrightarrow M \text{ přijme } w \Leftrightarrow w \notin L \text{ (spor)}$$

---

<sup>2</sup> Zdroj důkazu: <http://idav.ucdavis.edu/~okreylos/TAShip/Spring2001/LectureNotes27.pdf>

□

### Důsledek 1.6.2.

1. Jsou-li  $f_1, f_2 : \mathbb{N} \mapsto \mathbb{N}$  funkce, pro které platí, že  $f_1(n) \in o(f_2(n)/\log f_2(n))$  a  $f_2$  je časově konstruovatelná, potom

$$TIME(f_1(n)) \subsetneq TIME(f_2(n))$$

2. Pro každá dvě reálná čísla  $0 \leq \epsilon_1 < \epsilon_2$

$$TIME(n^{\epsilon_1}) \subsetneq TIME(n^{\epsilon_2})$$

3.  $P \subsetneq EXPTIME$

## 1.7 Úplné problémy pro třídu NP, Cook-Levinova věta.

### 1.7.1 Polynomiální převoditelnost, NP-úplnost

Jazyk  $A$  je **převoditelný v polynomiálním čase (polynomiálně převoditelný)** na jazyk  $B$ , psáno  $A \leq_m^P B$ , pokud existuje funkce  $f : \Sigma^* \mapsto \Sigma^*$  vyčíslitelná v polynomiálním čase, pro kterou platí

$$(\forall w \in \Sigma^*)(w \in A \Leftrightarrow f(w) \in B)$$

**Vlastnosti:**

1.  $\leq_m^P$  je reflexivní a tranzitivní relace (*kvaziusporádání*).
2. Pokud  $A \leq_m^P B$  a  $B \in P$ , pak  $A \in P$ .
3. Pokud  $A \leq_m^P B$  a  $B \in NP$ , pak  $A \in NP$ .

*Důkaz.*

1. Reflexivita plyne z toho, že identita je funkce spočitatelná v polynomiálním čase. Tranzitivita plyne z toho, že složením dvou polynomů vznikne opět polynom.
2. Je-li  $B \in P$ , pak existuje Turingův stroj  $M$ , který přijímá  $B$  v polynomiálním čase. Je-li  $f$  funkce, která převádí  $A$  na  $B$ , a je-li  $f$  spočitatelná v polynomiálním čase, pak TS  $M'$ , který pro vstup  $x$  spočítá  $f(x)$  a poté pustí  $M$  k rozhodnutí, zda  $f(x) \in B$ , přijímá  $A$  v polynomiálním čase.
3. Platí z téhož důvodu jako předchozí bod, protože tytéž argumenty lze použít i pro nedeterministický TS.

□

Jazyk  $B$  je **NP-těžký**, pokud je na něj převoditelný kterýkoli problém  $A \in NP$ .

Jazyk  $B$  je **NP-úplný**, je-li NP-těžký a navíc  $B \in NP$ .

Pokud chceme ukázat, že nějaký problém  $B$  je NP-úplný, pak stačí

1. ukázat  $B \in NP$
2. najít jiný NP-úplný problém  $A$  a převést jej na  $B$  (tj. ukázat  $A \leq_m^P B$ ).

*Za předpokladu  $P \neq NP$  platí, že pokud  $B$  je NP-úplný problém, pak  $B \notin P$ .*

### 1.7.2 Cook-Levinova věta

Původní znění Cook-Levinovy věty je následující:

**Věta 1.7.1.** *Pokud by byl problém splnitelnosti booleovských formulí řešitelný v polynomiálním čase, pak by se  $P = NP$ . Přesněji, splnitelnost booleovských formulí je NP-úplný problém.*

Obecněji lze označení *Cook-Levinova věta* použít pro libovolnou větu, která *ukazuje NP-úplnost praktického problému přímo z definice* třídy NP. Jako první praktický NP-úplný problém se obvykle uvažuje splnitelnost formule v konjunktivně normální formě. U nás se však jako výchozí problém vžilo KACHLÍKOVÁNÍ.

### KACHLÍKOVÁNÍ (anglicky *Tiling*)

**Instance:** Množina barev  $B$ , přirozené číslo  $s$ , čtvercová síť  $S$  velikosti  $s \times s$ , hrany jejichž krajních políček jsou obarveny barvami z  $B$ . Dále je součástí instance množina  $K \subseteq B \times B \times B \times B$  s typy kachlíků, které odpovídají čtverci, jehož hrany jsou obarveny barvami z  $B$ . Tyto kachlíky mají přesně definovaný horní, dolní, levý i pravý okraj a není možné je otáčet.

**Oázka:** Existuje přípustné vykachlíkování čtvercové sítě  $S$  kachlíky, jejichž typy jsou v množině  $K$ ? Přípustné vykachlíkování je takové přiřazení typů kachlíků jednotlivým polím čtvercové sítě  $S$ , v němž kachlíky, které spolu sousedí mají touž barvu na vzájemně dotýkajících se hranách a kachlíky, které se dotýkají strany  $S$ , mají shodnou barvu s okrajem. Jednotlivé typy kachlíků lze použít víckrát.

Dokážeme, že KACHLÍKOVÁNÍ (KACHL) je NP-úplný problém:

*Důkaz.* Všimněme si nejprve, že  $\text{KACHL} \in \text{NP}$ . To plyne z toho, že dostaneme-li vykachlíkování sítě  $S$ , tedy přiřazení typů kachlíků jednotlivým políčkům, dokážeme ověřit v polynomiálním čase, jde-li o přípustné vykachlíkování.

Nechť  $A \subseteq \{0,1\}^*$ ,  $A \in \text{NP}$ . Ukážeme, že  $A \leq_m^P \text{KACHL}$ . Jelikož  $A \in \text{NP}$ , existuje NTS  $M$ ,  $L(M) = A$ , a počet kroků každého přijímajícího výpočtu je omezen polynomem  $p(n)$ . BÚNO  $p(n) > n$ , jinak vezmeme  $\max(p(n), n)$ .

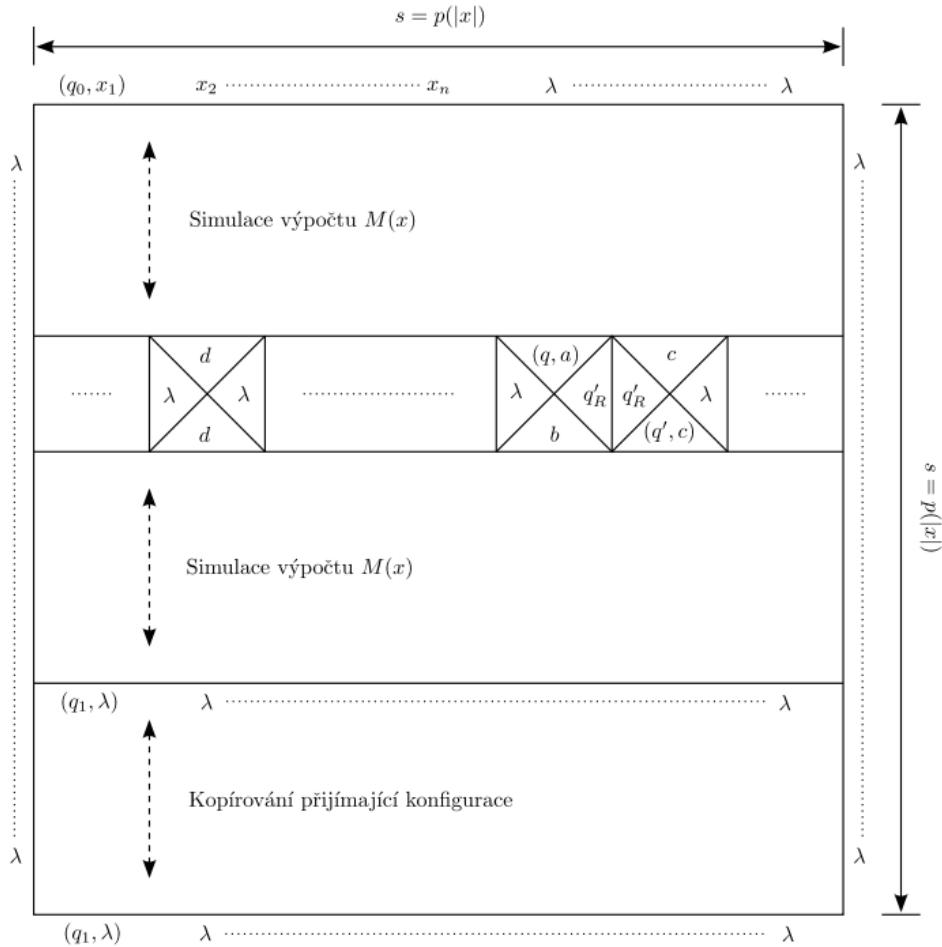
Připomeňme si, že podle definice  $x \in A$ , právě když existuje přijímající výpočet NTS  $M$  nad vstupem  $x$  délky nejvýše  $p(|x|)$ . Nechť  $M = (Q, \Sigma, \delta, q_0, F)$ , kde  $Q$  obsahuje stavy  $q_0$  a  $q_1$  a  $\{0, 1, \lambda\} \subseteq \Sigma$ . BÚNO platí:

1.  $F = q_1$ , tj.  $M$  má jediný přijímající stav  $q_1$  různý od  $q_0$ .
2.  $\forall a \in \Sigma : \delta(q_1, a) = \emptyset$ , tj. z přijímajícího stavu neexistuje definovaný přechod.
3. Počáteční konfigurace vypadá tak, že hlava stojí na nejlevějším symbolu vstupního slova  $x$ , které je zapsáno počínaje od levého okraje vymezeného prostoru délky  $p(|x|)$ . Zbytek pásky je prázdný.
4. Během výpočtu se hlava  $M$  nepohně nalevo od místa, kde byla v počáteční konfiguraci, tj. mimo vymezený prostor.
5. Přijímající konfigurace: že páška je prázdná a hlava stojí na nejlevější pozici vymezeného prostoru. To odpovídá tomu, že než se  $M$  rozhodne přijmout, smaže nejprve obsah pásky a přesune hlavu k levému okraji vymezeného prostoru.

Není těžké ukázat, že ke každému NTS  $M_1$  lze zkonstruovat NTS  $M_2$ , který přijímá týž jazyk jako  $M_1$  a splňuje uvedené podmínky.

Nechť  $x$  je instance problému  $A$ , popíšeme, jak z  $M$ , polynomu  $p$  a instance  $x$  vytvořit instanci KACHLÍKOVÁNÍ, pro kterou bude platit, že v ní existuje přípustné vykachlíkování, právě když existuje přijímající výpočet  $M(x)$ .

Idea důkazu je taková, že hrany barev mezi dvěma řádky kachlíků budou kódovat konfigurace výpočtu NTS  $M$  nad vstupem  $x$ . Vhodným výběrem kachlíků zabezpečíme, že v přípustném vykachlíkování bude řada kachlíků simulovat změnu konfigurace na následující pomocí přechodové funkce. Horní a dolní okraje sítě  $S$  obarvíme tak, aby barvy určovaly počáteční a přijímající konfiguraci, obě jsou dané jednoznačně, přijímající zcela jednoznačně, počáteční je sice závislá na  $x$ , ale pro dané  $x$  je již jednoznačná. Konstrukce je ilustrována na obrázku ??.



Barvy kachlíků tedy budou odpovídat symbolům, které potřebujeme pro zakódování konfigurace, ale budeme potřebovat i pomocné barvy pro přenos informace o stavu o kachlík vlevo nebo vpravo. Položíme tedy

$$B = \Sigma \cup Q \times \Sigma \cup \{q_L, q_R \mid q \in Q\}$$

- Barva  $(q, a)$  v této konfiguraci bude kódovat políčko na páscce, nad kterým se vyskytuje hlava, přičemž  $q$  označuje stav, ve kterém se  $M$  nachází. Vždy je pouze jedna barva tohoto typu na řádku.
- Ostatní políčka konfigurace budou obarvena barvou odpovídající symbolu na daném místě.
- Velikost jedné strany čtvercové sítě položíme  $s = p(|x|)$ .
- Boční strany  $S$  budou obarveny barvou  $\lambda$  (prázdný symbol).
- Horní strana  $S$  bude obarvena počáteční konfigurací:  $[(q_0, x_1), x_2, x_3, \dots, x_n, \lambda, \dots, \lambda]$ , pro vstup  $x = x_1 x_2 x_3 \dots x_n$ .
- Spodní hrana  $S$  bude obarvena jednoznačnou přijímající konfigurací:  $[(q_1, \lambda), \lambda, \dots, \lambda]$

Do typů kachlíků zakódujeme přechodovou funkci Turingova stroje  $M$ , čímž dosáhneme toho, že správné vykachlíkování řádků 2 až  $s$  bude odpovídat výpočtu  $M$  nad  $x$ . Navíc přidáme možnost kopírování přijímající konfigurace tak, abychom osetřili i případ, kdy výpočet  $M$  nad  $x$  skončí po méně než  $p(|x|)$  krocích.

Použité kachlíky jsou vypsány v tabulce 1.3.

Tabulka 1.3: Kachlíky

(I)		$\forall a \in \Sigma \text{ kopírování buněk, nad nimiž není hlava}$
(II)		$\forall q \forall a \forall q' \forall a' : (q', a', N) \in \delta(q, a)$
(III), (IV)		$\forall q \forall a \forall q' \forall a' : (q', a', R) \in \delta(q, a)$
(V), (VI)		$\forall q \forall a \forall q' \forall a' : (q', a', L) \in \delta(q, a)$
(VII)		

Funkce  $f$ , která bude převádět instanci problému  $A$  na instanci problému KACHL provede právě popsanou konstrukci, tj. z popisu  $M$  a instance  $x$  vytvoří instanci  $(B, K, s, S)$ , kde množina  $K$  obsahuje popsané typy kachlíků a pod  $S$  mínime obarvení okrajů sítě. Tuto konstrukci je zřejmě možné provést v polynomiálním čase. Jediné co je v konstrukci závislé na velikosti vstupu je tedy rozměr sítě  $s = p(|x|)$  a obarvení horního okraje  $S$  počáteční konfigurací.

Zbývá ukázat, že  $x \in A \Leftrightarrow$  takto zkonztruovaná instance KACHL má přípustné vykachlíkování:

$\Rightarrow$ : Předpokládejme nejprve, že  $x \in A$  a tedy existuje přijímající výpočet  $M(x)$  daný posloupností konfigurací  $K_0^x = K_0, \dots, K_t = K_F$ . Podle předpokladu platí, že  $t \leq p(|x|)$  a délka slova na pásmu v každé konfiguraci je rovněž nejvýš  $p(|x|)$ . Popíšeme, jak s pomocí konfigurací  $K_0, \dots, K_t$  obarvit síť  $S$ . Pro řadu  $i = 0$ , tj. horní okraj  $S$  je vybarvení správně díky konstrukci. Dále indukcí: máme-li validně vybarvenou  $i$ -tou řadu, vykachlíkujeme  $(i+1)$ -ní řadu s odšimováním příslušné instrukce, která vedla k přechodu z  $K_i$  do  $K_{i+1}$ . Takto se dostaneme k  $K_t = K_F$  a poté dokopírujeme  $K_F$  až na poslední řádek čtvercové sítě  $S$ . Instrukci lze vždy odšimulovat díky použitým typům kachlíků.

$\Leftarrow$  Nechť  $\exists$  přípustné vykachlíkování čtvercové sítě  $S$ . Ukážeme, že řádky barev mezi jednotlivými řádky kachlíků určují posloupnost konfigurací v přijímajícím výpočtu  $M$  nad  $x$ .

Indukcí dle  $i$ : nechť  $b_{i,1}, \dots, b_{i,s} \in B$  je posloupnost barev mezi  $i$ -tým a  $(i+1)$ -ním řádkem. Musíme ukázat tyto dvě vlastnosti pro každé  $i = 0, \dots, s$ :

1. V posloupnosti  $b_{i,1}, \dots, b_{i,s}$  je **právě jedna barva** typu  $(q, a) \in Q \times \Sigma$ , ostatní jsou typu  $a \in \Sigma$ , tj. posloupnost  $b_{i,1}, \dots, b_{i,s}$  **určuje validní konfiguraci**  $K_i$ .
2. Platí, že  $K_i$  lze z  $K_{i-1}$  **vytvořit pomocí přechodové funkce**  $\delta$  pro  $i > 0$  a  $K_0 = K_0^x$ .

Obě vlastnosti jsou jistě splněné pro  $i = 0$ . Předpokládejme nyní, že jsou splněny pro  $i \in [0, \dots, s]$  a ukažme, že platí pro  $i + 1$ . Obě vlastnosti opět jednoduše plynou z toho, jaké kachlíky máme k dispozici. Podle indukčního předpokladu se na řádku barev vyskytuje právě jedna pozice, řekněme  $k$ , pro kterou platí, že  $b_{i,k} = (q, a)$  pro nějaký stav  $q \in Q$  a znak  $a \in \Sigma$ . To znamená, že v  $(i+1)$ -ním řádku kachlíků je právě jeden kachlík s horní barvou  $(q, a)$ , a to na pozici  $S[i+1, k]$ . Kachlíky nám neumožňují vytvořit barvu  $(q, a)$  z ničeho, musí jít o barvu tohoto typu převedenou z předchozího řádku, a to buď kachlíkem (II) a nebo kachlíky (III) a (IV) a nebo kachlíky (V) a (VI).

Spodní hrana  $S$  je obarvena přijímající konfigurací, takže poslední řada barev kachlíků musí odpovídat přijímající konfiguraci.

Dostáváme tedy, že posloupnost konfigurací daných barvami na hranách mezi řádky kachlíků v čtvercové síti  $S$  odpovídají přijímajícímu výpočtu  $M$  nad vstupem  $x$  a jde dokonce o vzájemnou korespondenci, a tedy máme-li přípustné vykachlíkování, znamená to, že  $M(x)$  přijme. Z toho plyne, že  $x \in A$ .

□

### 1.7.3 Další NP-úplné problémy

#### 1.7.3.1 SAT – Splnitelnost formule v KNF

##### SPLNITELNOST FORMULE V KNF (SAT)

**Instance:** Formule  $\varphi$  na  $n$  proměnných v konjunktivně normální formě.

**Oázka:** Existuje ohodnocení proměnných  $t \in \{0, 1\}^n$ , pro které platí  $\varphi(t) = 1$ ? Jinými slovy, existuje ohodnocení, pro které je  $\varphi$  splněná?

**Věta 1.7.2.** SAT je NP-úplný.

*Důkaz.*

1. SAT  $\in$  NP, ohodnocení  $t$  umíme polynomiálně rychle ověřit.
2. KACHL  $\rightarrow$  SAT Sestrojíme  $\varphi$  s  $s^2 \cdot |K|$  proměnnými ( $s^2$  velikost pole,  $K$  počet typů kachlíků) tvaru  $x_{i,j,k}$ , kde  $i, j \in \{1 \dots s\}$ ,  $k \in \{1 \dots |K|\}$  a hodnota  $x_{i,j,k}$  vyjadřuje „na poli  $S[i, j]$  je kachlík typu  $T_k$ “.

Definujeme:

$$V = \{(p, q) \mid \text{spodní barva kachlíku typu } T_p \text{ se neshoduje s horní barvou } T_q\}$$

$$H = \{(p, q) \mid \text{prava barva kachlíku typu } T_p \text{ se neshoduje s levou barvou } T_q\}$$

a pro  $i \in \{1, \dots, s\}$  definujeme

$$U_i = \{k \mid \text{horní barva } T_k \text{ se shoduje s barvou horního okraje } S \text{ v } i\text{-tém sloupci}\}$$

$$B_i = \{k \mid \text{spodní barva } T_k \text{ se shoduje s barvou spodního okraje } S \text{ v } i\text{-tém sloupci}\}$$

$$L_i = \{k \mid \text{levá barva } T_k \text{ se shoduje s barvou levé okraje } S \text{ v } i\text{-tém řádku}\}$$

$$R_i = \{k \mid \text{pravá barva } T_k \text{ se shoduje s barvou pravého okraje } S \text{ v } i\text{-tém řádku}\}$$

Konstrukce  $\varphi$ :

- (a)  $C_{i,j} = \bigvee_{k=1}^{|K|} x_{i,j,k} \quad \dots \quad \text{„každé pole má kachlík“}$
- (b)  $\alpha_{i,j} = \bigwedge_{k=1}^{|K|} \bigwedge_{l=k+1}^{|K|} (\overline{x_{i,j,k}} \vee x_{i,j,l}) \quad \dots \quad \text{„každé pole má nejvýše 1 kachlík“}$
- (c)  $\beta_{i,j} = C_{i,j} \wedge \alpha_{i,j} \quad \dots \quad \text{„každé pole má právě 1 kachlík“}$

- (d)  $\gamma_{i,j} = \bigwedge_{(p,q) \in V} (\overline{x_{i,j,p}} \vee \overline{x_{i+1,j,q}})$  ... „pole jsou **vertikálně** kompatibilní“  
(e)  $\delta_{i,j} = \bigwedge_{(p,q) \in H} (\overline{x_{i,j,p}} \vee \overline{x_{i,j+1,q}})$  ... „pole jsou **horizontálně** kompatibilní“  
(f)  $\varepsilon = \bigwedge_{j=1}^s \left( \bigvee_{k \in U_j} x_{1,j,k} \right) \wedge \bigwedge_{j=1}^s \left( \bigvee_{k \in B_j} x_{s,j,k} \right) \wedge \bigwedge_{j=1}^s \left( \bigvee_{k \in L_j} x_{j,1,k} \right) \wedge \bigwedge_{j=1}^s \left( \bigvee_{k \in R_j} x_{j,s,k} \right)$   
... „okraje jsou kompatibilní“

Nyní

$$\varphi = \bigwedge_{i=1}^s \bigwedge_{j=1}^s \beta_{i,j} \wedge \bigwedge_{i=1}^{s-1} \bigwedge_{j=1}^s \gamma_{i,j} \wedge \bigwedge_{i=1}^s \bigwedge_{j=1}^{s-1} \delta_{i,j} \wedge \varepsilon$$

**Ekvivalence:** pokud existuje vykachlíkování, tak existuje splňující ohodnocení  $\varphi$ :  $x_{i,j,k} = 1$  pokud na  $S[i, j]$  je kachlík typu  $T_k$ ; 0 jinak. Pokud existuje ohodnocení, tak vykachlíkujeme takto: díky  $\beta$  existuje pro  $\forall i, j$  právě 1  $k$  takové, že  $x_{i,j,k} = 1 \rightarrow$  na  $S[i, j]$  umístíme  $T_k$ .  $\square$

### 1.7.3.2 3SAT – Splnitelnost formule ve 3KNF

#### SPLNITELNOST FORMULE V 3KNF (3SAT)

**Instance:** Formule  $\varphi$  na  $n$  proměnných v 3KNF (každá klauzule obsahuje **právě 3** literály).

**Oázka:** Existuje ohodnocení proměnných  $t \in \{0, 1\}^n$ , pro které platí  $\varphi(t) = 1$ ? Jinými slovy, existuje ohodnocení, pro které je  $\varphi$  splněná?

**Věta 1.7.3.** 3SAT je NP-úplný.

*Důkaz.* (idea)

1. 3SAT  $\in$  NP, ohodnocení  $t$  umíme polynomiálně rychle ověřit.
2. SAT  $\rightarrow$  3SAT Z formule  $\psi = C_1 \wedge C_2 \wedge \dots \wedge C_m$  v KNF sestrojíme  $\varphi$  v 3KNF.

$$\forall C_i = (e_1 \vee e_2 \vee \dots \vee e_k) \rightarrow \alpha_i$$

- $k = 1$ :  $\alpha_i = (e_1 \vee y_1^j \vee y_2^j) \wedge (e_1 \vee y_1^j \vee \bar{y}_2^j) \wedge (e_1 \vee \bar{y}_1^j \vee y_2^j) \wedge (e_1 \vee \bar{y}_1^j \vee \bar{y}_2^j)$
- $k = 2$ :  $\alpha_i = (e_1 \vee e_2 \vee y^j) \wedge (e_1 \vee e_2 \vee \bar{y}^j)$
- $k = 2$ :  $\alpha_i = C_i$
- $k > 3$ :  $\alpha_i = (e_1 \vee e_2 \vee y_1^j) \wedge (\bar{y}_1^j \vee e_3 \vee e_2 \vee y_2^j) \wedge \dots \wedge (\bar{y}_{k-3}^j \vee e_{k-1} \vee e_k)$

Kde všechna  $y$  jsou nové proměnné. Pak  $\varphi = \bigwedge_{j=1}^m$ . Konstrukce je polynomiální. Ekvivalence rozborem případů.  $\square$

### 1.7.3.3 VP – Vrcholové pokrytí

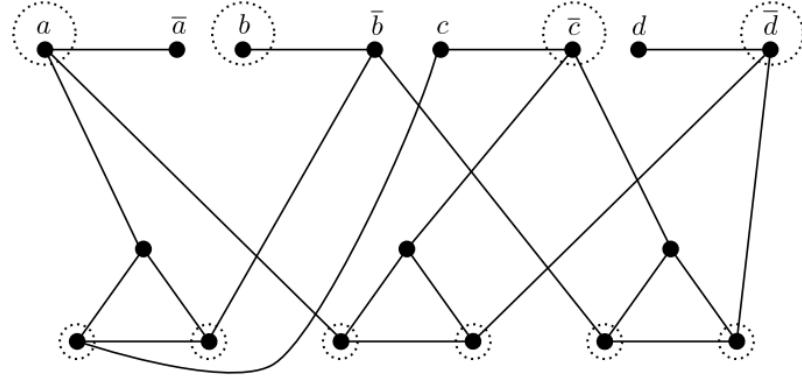
#### VRCHOLOVÉ POKRYTÍ (VP) (ANGL. VERTEX COVER)

**Instance:** Graf  $G = (V, E)$  a přirozené číslo  $k$ .

**Oázka:** Existuje množina  $S \subseteq V$ , která obsahuje nejvýš  $k$  vrcholů a z každé hrany obsahuje alespoň jeden koncový vrchol? (Tj.  $|S| \leq k$  a  $(\forall e \in E)[S \cap e \neq \emptyset]$ .)

**Věta 1.7.4.** VP je NP-úplný.

Důkaz. (idea) Převodem 3SAT → VP. Příklad:



Obrázek 1.1: Příklad převodu formule  $\varphi = (a \vee b \vee c) \wedge (a \vee c \vee d) \wedge (b \vee c \vee d)$  na instanci VP. Jednotlivé trojúhelníky dole odpovídají klauzulím, zakroužkované vrcholy určují vrcholové pokrytí velikosti k, které odpovídá splňujícímu ohodnocení a = b = 1, c = d = 0.

□

#### 1.7.3.4 HK – Hamiltonovská kružnice

##### HAMILTONOVSKÁ KRUŽNICE (HK)

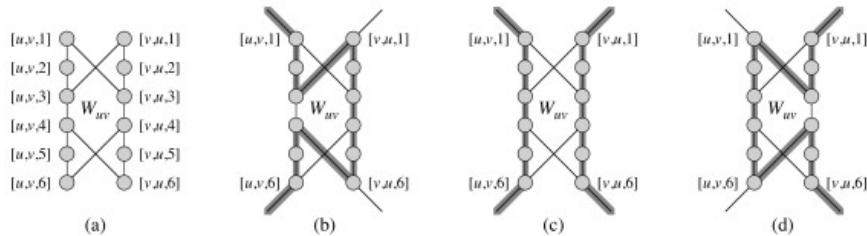
**Instance:** Graf  $G = (V, E)$ .

**Otzáka:** Existuje v  $G$  cyklus vedoucí přes všechny vrcholy?

**Věta 1.7.5.** HK je NP-úplný.

Důkaz. (idea) Převodem z VP. Pro graf  $G = (V, E)$  chceme najít VP velikosti k, zkonstruujeme  $G' = (V', E')$  tak, že:

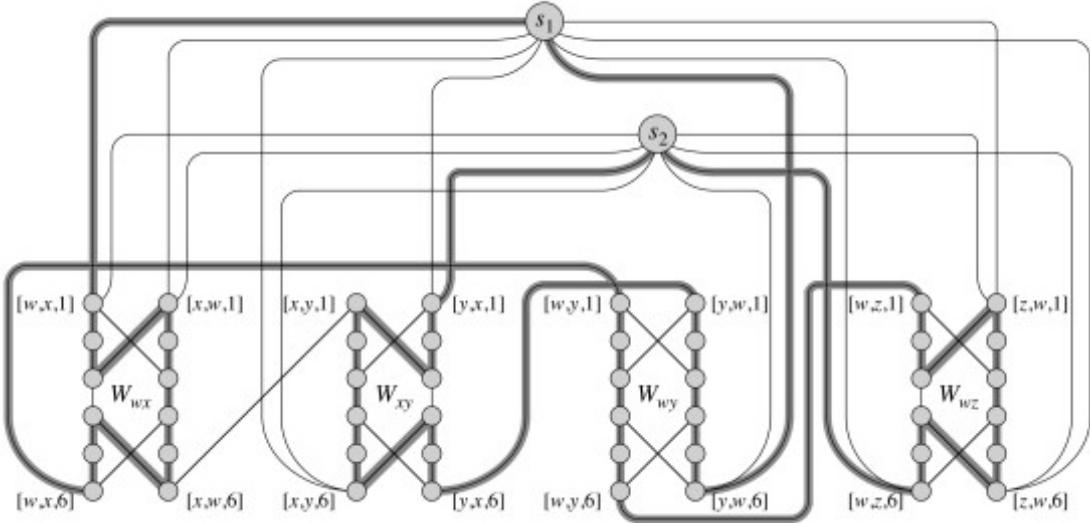
- pro každou hranu  $(u, v) \in E$  vytvoříme podgraf jako na obrázku ??(a).



Obrázek 1.2: Podgraf pro hranu  $(u, v)$

Hamiltonovská kružnice může projít tímto podgrafem pouze 3 způsoby naznačenými na obrázku ??(b,c,d):

1. (b) hranu je pokrytá vrcholem  $u$
  2. (c) hranu je pokrytá oběma vrcholy  $u, v$
  3. (d) hranu je pokrytá vrcholem  $v$
- do  $V'$  přidáme k „výběrových“ vrcholů  $s_1, \dots, s_k$
  - pro každý vrchol  $v$  spojíme všechny podgrafe odpovídající jemu incidentním hranám do řetízku pod sebe (tj.  $(v, x, 6)$  napojíme na  $(v, y, 1)$  atd.). Konce tohoto řetízku spojíme hranou se všemi výběrovými vrcholy  $s_1, \dots, s_k$



Obrázek 1.3: Výsledné zapojení.

□

#### 1.7.3.5 3DM – Trojrozměrné párování

##### TROJROZMĚRNÉ PÁROVÁNÍ (3DM)

**Instance:** Množina  $M \subseteq W \times X \times Y$ , kde  $W, X, Y$  jsou po dvou disjunktní množiny, z nichž každá obsahuje právě  $q$  prvků.

**Otázka:** Obsahuje  $M$  perfektní párování? Jinými slovy, existuje množina  $M' \subseteq M$ ,  $|M'| = q$ , trojice v níž obsažené jsou po dvou disjunktní?

Převodem ze SAT.

#### 1.7.3.6 LOUP – Loupežníci

##### LOUPEŽNÍCI (LOUP) (angl. Partition)

**Instance:** Množina prvků  $A$  a s každým prvkem  $a \in A$  asociovaná cena (váha, velikost, ...)  $s(a) \in \mathbb{N}$ .

**Otázka:** Lze rozdělit prvky z  $A$  na dvě poloviny s toutéž celkovou cenou? Přesněji, existuje množina  $A' \subseteq A$  taková, že

$$\sum_{a \in A'} s(a) = \sum_{a \in A \setminus A'} s(a) ?$$

Převodem ze 3DM.

#### 1.7.3.7 Další

Klika, Nezávislá množina, Orientovaná HK, Hamiltonovská cesta z  $s$  do  $t$ , Hamiltonovská cesta, Obchodní cestující, Batoh, Rozvrhování, Dominující množina, Hitting set, Čtyřlistá kostra, Největší společný podgraf, Množinové pokrytí.

## 1.8 Pseudopolynomiální algoritmy, silná NP-úplnost.

I mezi různými NP-úplnými problémy a úlohami může být co do složitosti jejich řešitelnosti rozdíl.

### 1.8.1 Pseudopolynomiální algoritmy

Nechť  $A$  je libovolný rozhodovací problém a  $I$  nechť je instance tohoto problému. Potom

$\text{len}(I)$  označuje **délku zakódování** instance  $I$  při standardním *binárním* kódování.

$\text{max}(I)$  označuje hodnotu **největšího číselného parametru**, který se vyskytuje v  $I$ .

Řekneme, že  $A$  je **číselný problém**, pokud pro každý polynom  $p$  existuje instance  $I$  tohoto problému taková, že  $\text{max}(I) > p(\text{len}(I))$ . *Intuitivně: pokud je v problému nějaký parametr, kterého fakt záleží, jestli je to 10 nebo 1000000000 a ovlivní to rychlosť běhu algoritmu, pak je to číselný problém.*

Například BATOH nebo LOUPEŽNÍCI jsou číselné problémy. SPLNITELNOST nebo KACHLÍKOVÁNÍ číselné nejsou.

Řekneme, že algoritmus, který řeší problém  $A$ , je **pseudopolynomiální**, pokud je jeho časová složitost omezena polynomem dvou proměnných  $\text{len}(I)$  a  $\text{max}(I)$ . *Intuitivně: algoritmus je v podstatě polynomiální, ale visí na tom nějakém parametru, který může být třeba i 1000000000000.*

Druhý pohled na věc je, že pseudopolynomiální algoritmus by byl polynomiální, pokud předáme vstup zakódovaný **unárně**. *Intuitivně: pak totiž musíme těch 1000000000000 kódovat jako 1000000000000 jedniček, takže pak velikost vstupu reflektuje ten astronomicky velký parametr.*

Je-li problém  $A$  řešitelný pseudopolynomiálním algoritmem a není-li  $A$  číselný problém, pak je zřejmě tento pseudopolynomiální algoritmus ve skutečnosti polynomiální.

#### 1.8.1.1 Příklad: Batoh

##### BATOH (KNAPSACK)

**Instance:** Množina  $n$  předmětů  $A = \{a_1, \dots, a_n\}$ , s každým předmětem asociovaná velikost  $s(a_i) \in \mathbb{N}$  a cena či hodnota  $v(a_i) \in \mathbb{N}$ . Přirozené číslo  $B \geq 0$  udávající velikost batohu.

**Cíl:** Nalézt množinu předmětů  $A' \subseteq A$ , která dosahuje maximální souhrnné hodnoty předmětů v ní obsažených, a přitom se předměty z  $A'$  vejdu do batohu velikosti  $B$ . Tj. chceme, aby platilo:

$$\sum_{a \in A'} s(a) \leq B$$

$$\sum_{a \in A'} v(a) = \max_{A'' \subseteq A, \sum_{a \in A'} s(a) \leq B} \sum_{a \in A''} v(a)$$

Následující pseudopolynomiální algoritmus je založen na dynamickém programování. Na vstupu očekává jen předměty, které se do batohu vejdu a současně jejich velikost je nenulová ( $\forall i \in \{1, \dots, n\} : 0 < s[i] \leq B$ ).

```

1 Batoh(s , v , B):
2     # s ... pole velikosti predmetu
3     # v ... pole cen predmetu
4     # B ... velikost batohu
5     V = sum(v[i] for i in range(1,n))
6     n = len(s)
7     T = [ , ]
8     S = [ , ]
9     for c in range (1, V):
10        T[0 , c] = None
11        S[0 , c] = B + 1
12        for j in range(0, n):
13            T[j , 0] = None
14            S[j , 0] = 0
15            for c in range(1, V):
16                T[j , c] = T[j - 1 , c]
17                S[j , c] = S[j - 1 , c]
18                if v[j] <= c and S[j , c] > S[j - 1 , c - v[j]] + s[j]:
19                    T[j , c] = union(T[j - 1 , c - v[j]] , {j})
20                    S[j , c] = S[j - 1 , c - v[j]] + s[j]
21        cc = max{c for (n,c) in S if S[n , c] <= B}
22    return T[n , cc]

```

$T$  je tabulka typu  $(n+1) \times (V+1)$ , na pozici  $T[j, c]$ ,  $0 \leq j \leq n$ ,  $0 \leq c \leq V$ , bude podmnožina indexů  $\{1, \dots, j\}$  prvků celkové ceny přesně  $c$  s minimální velikostí.

$S$  je tabulka typu  $(n+1) \times (V+1)$ , na pozici  $S[j, c]$  je celková velikost předmětů v množině  $T[j, c]$ . Pokud neexistuje množina s předměty ceny přesně  $c$ , je na pozici  $S[j, c]$  číslo  $B+1$ .

Popsaný algoritmus pracuje v čase  $\Theta(nV)$  (počítáme-li aritmetické operace jako konstantní). Zdá se, tedy, že jde o polynomiální algoritmus, ale musíme si uvědomit, že velikost vstupu je při standardním binárním kódování pouze  $O(n \log_2(B+V))$  a  $V$  tedy může být exponenciálně větší než vstup ( $\Rightarrow$  je to číselný problém). Formálně tedy  $\text{len}(I) = O(n \log_2(B+V))$ , zatímco  $\max(I) = \max(\{B\} \cup \{v[i], s[i] \mid 1 \leq i \leq n\})$  a dle definice jde tedy skutečně o pseudopolynomiální algoritmus.

### 1.8.2 Silná NP-úplnost

Nechť  $A$  je NP-úplný problém a  $p$  je polynom. Pomocí  $A(p)$  označíme restrikci problému  $A$  na instance  $I$ , pro něž platí  $\max(I) \leq p(\text{len}(I))$ .

Řekneme, že problém  $A$  je **silně NP-úplný**, pokud existuje polynom  $p$ , pro který  $A(p)$  je NP-úplný problém. *Intuitivně: omezení  $A(p)$  nám dá pouze „nečíselné instance“, tj. takové, kde jsou číselné parametry rozumně malé (jdou omezit polynomem).* Pokud i tyhle „rozumné“ instance jsou NP-úplné, tak je to fakt silně NP-úplný problém

Řekneme, že problém  $A$  je **slabě NP-úplný**, pokud pro něj existuje pseudopolynomiální algoritmus.

*Pozn.: Jsou NP-úplné problémy, které nejsou ani silně, ani slabě NP-úplné.*

**Každý nečíselný NP-úplný problém je silně NP-úplný.** Neboť pro nečíselný problém existuje polynom  $p$  takový, že pro *každou* instanci  $I$  platí  $\max(I) \leq p(\text{len}(I))$ , a pak tedy  $A(p) = A$ , který je NP-úplný (tj. restrikce podle tohoto polynomu  $p$  zachová všechny instance původního problému)).

Pokud by existoval silně NP-úplný problém, který lze vyřešit pseudopolynomiálním algoritmem, znamenalo by to, že  $P = NP$ .

Silně NP-úplný problém je NP-úplný i při unárním kódování.

I číselné NP-úplné problémy mohou být silně NP-úplné, například ROZVRHOVÁNÍ nebo BIN PACKING. Ukážeme že to platí i pro TSP:

### TRAVELLING SALESMAN PROBLEM (TSP) - ROZHODOVACÍ VERZE

**Instance:** Množina měst  $C = \{c_1, \dots, c_n\}$ , hodnoty  $d(c_i, c_j) \in \mathbb{N}$  přiřazující každé dvojici měst vzdálenost a  $D \in \mathbb{N}$ .

**Otázka:** Existuje permutace měst  $c_{\pi(1)}, c_{\pi(2)}, \dots, c_{\pi(n)}$ , pro kterou platí, že

$$\left( \sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)}) \right) + d(c_{\pi(n)}, c_{\pi(1)}) \leq D \quad ?$$

**Věta 1.8.1.** *Problém obchodního cestujícího (rozhodovací verze) je silně NP-úplný.*

*Důkaz.* Problém TSP jistě patří do třídy NP, neboť jsme schopni ověřit v polynomiálním čase, zda daná permutace měst splňuje naše požadavky, a zřejmě i  $TSP(p)$  patří do NP pro každý polynom  $p$ . Těžkost problému  $TSP(p)$  pro vhodně zvolený polynom  $p$  si ukážeme převodem z problému *Hamiltonovské kružnice v neorientovaném grafu*.

Uvažme neorientovaný graf  $G = (V, E)$  a sestavme na jeho základě instanci obchodního cestujícího následujícím způsobem:

- Množinu měst  $C$  položíme rovnou množině vrcholů  $V$ , tedy  $C = V = \{v_1, \dots, v_n\}$ .
- Vzdálenost mezi městy či vrcholy  $v_i$  a  $v_j$  pro  $i \neq j$  a  $i, j \in \{1, \dots, n\}$  určíme následujícím předpisem:

$$d(v_i, v_j) = \begin{cases} 0 & \text{pokud } \{v_i, v_j\} \in E \\ 1 & \text{jinak} \end{cases}$$

- Hodnotu D položíme rovnou 0.

Není těžké ukázat, že v grafu  $G$  existuje hamiltonovská kružnice, právě když v sestrojené instanci obchodního cestujícího existuje cyklus nulové délky, který obejde všechna města, přičemž navštíví každé z nich právě jednou. Z jedné strany hamiltonovská kružnice využívá jen hrany nulové délky, z druhé strany cesta nulové délky využívá jen hrany grafu  $G$ . Hodnota maximálního číselného parametru v sestrojené instanci je přitom rovna 1, tedy omezená verze problému  $TSP(1)$  je stále NP-úplná, což znamená, že problém TSP je silně NP-úplný.  $\square$

## 1.9 Aproximační algoritmy a schémata.

Pokud nejsme schopni rychle získat optimální řešení NP-úplné úlohy, můžeme slevit ze svých požadavků a pokusit se najít řešení, jež není od toho optimálního příliš vzdáleno. Nejprve si upřesníme pojem optimalizační úlohy:

### 1.9.1 Optimalizační úloha

**Optimalizační úlohu** definujeme jako trojici  $A = (D_A, S_A, \mu_A)$ , kde

- $D_A \subseteq \Sigma^*$  je **množina instancí**
- $S_A(I)$  je **množina přípustných řešení** pro instanci  $I \in D_A$
- $\mu_A(I, \sigma)$  přiřazuje instanci  $I \in D_A$  a přípustnému řešení  $\sigma \in S_A(I)$  kladné racionální číslo, tzv. **hodnotu řešení**.

Úloha může být **A maximalizační** resp. **minimalizační**, pak optimálním řešením instance  $I$  je to přípustné řešení  $\sigma \in S_A(I)$ , jež má **maximální** resp. **minimální** hodnotu  $\mu_A(I, \sigma)$ .

**Hodnotu optimálního řešení** označíme pomocí  $opt(I)$ .

Příklad: Minimalizační úloha *Vrcholového pokrytí (VP)*:

- množina instancí  $D_{VP} =$  všechny řetězce kódující nějaký graf  $G = (V, E)$

- množina přípustných řešení pro instanci  $G$ , tj.  $S_{VP}(G) =$  všechny množiny vrcholů  $S \subseteq V$  pokrývající všechny hrany
- hodnota řešení  $S$ , tj.  $\mu_{VP}(G, S) = |S|$ , tedy počet vrcholů v množině

### 1.9.2 Aproximační algoritmus

Algoritmus  $R$  nazveme **aproximačním algoritmem** pro optimalizační úlohu  $A$ , pokud pro každou instanci  $I \in D_A$  je výstupem  $R(I)$  přípustné řešení  $\sigma \in S_A(I)$  (pokud nějaké existuje). Označení  $R(I)$  budeme často používat přímo pro hodnotu řešení  $\sigma$  vráceného algoritmem  $R$  na instanci  $I$ , tj.  $R(I) = \mu_A(I, \sigma)$ .

Je-li  $A$  maximalizační úloha, pak  $\varepsilon \geq 1$  je **aproximačním poměrem** algoritmu  $R$ , pokud pro každou instanci  $I \in D_A$  platí, že

$$opt(I) \leq \varepsilon \cdot R(I)$$

Je-li  $A$  minimalizační úloha, pak  $\varepsilon \geq 1$  je **aproximačním poměrem** algoritmu  $R$ , pokud pro každou instanci  $I \in D_A$  platí, že

$$R(I) \leq \varepsilon \cdot opt(I)$$

#### 1.9.2.1 Příklad: Bin Packing

##### BIN PACKING (BP)

**Instance:** Konečná množina předmětů  $U = \{u_1, \dots, u_n\}$ , s každým předmětem asociovaná velikost  $s(u)$ , což je racionální číslo, pro které platí  $0 \leq s(u) \leq 1$ .

**Cíl:** Najít rozdělení všech předmětů do co nejmenšího počtu po dvou disjunktních množin  $U_1, \dots, U_m$  takové, že

$$(\forall i \in \{1, \dots, m\}) \left( \sum_{u \in U_i} s(u) \leq 1 \right)$$

Naším cílem je minimalizovat  $m$ .

Rozhodovací varianta je ekvivalentní problému ROZVRHOVÁNÍ, a tedy i úloha BIN PACKING je NP-těžká. Zkusíme navrhnout aproximační algoritmus:

**Algoritmus 1: First Fit (FF):**

1. Ber jeden předmět po druhém, pro každý předmět  $u$  najdi první koš, do nějž se tento předmět ještě vejde.
2. Pokud takový koš neexistuje, přidej nový koš obsahující jen předmět  $u$ .

**Věta 1.9.1.** Pro každou instanci  $I \in D_{BP}$  platí, že  $FF(I) < 2 \cdot opt(I)$ .

**Důkaz.** V řešení, které vrátí FF, je nejvýš jeden koš, který je zaplněn nejvýš z poloviny. Kdyby totiž existovaly dva koše  $U_i, U_j$  pro  $i \neq j$ , které jsou zaplněny nejvýš z poloviny, tak by FF nepotřeboval zakládat nový koš pro předměty z  $U_j$ , všechny by se vešly do  $U_i$ . Pokud  $FF(I) > 1$ , pak z toho plyne, že

$$FF(I) < \lceil 2 \sum_{i=1}^n s(u_i) \rceil \leq 2 \lceil \sum_{i=1}^n s(u_i) \rceil$$

kde první nerovnost plyne z toho, že po zdvojnásobení obsahu jsou všechny koše plné až na jeden, který může být zaplněn jen částečně. Rovnosti bychom přitom dosáhli jedině ve chvíli, kdy by byly všechny koše zaplněné právě z poloviny, což není podle našeho předpokladu možné. Druhá nerovnost plyne z vlastností zaokrouhlování.

Na druhou stranu musí platit, že

$$opt(I) \geq \lceil \sum_{i=1}^n s(u_i) \rceil$$

Dohromady tedy dostaneme, že  $FF(I) < 2 \cdot opt(I)$ . Pokud  $FF(I) = 1$ , pak i  $opt(I) = 1$  a i v tomto případě platí ostrá nerovnost.  $\square$

**Věta 1.9.2.** Pro libovolné  $k$  existuje instance  $I \in D_{BP}$ , pro niž je  $opt(I) \geq k$  a FF vytvoří pro  $I$  aspoň  $\frac{5}{3}opt(I)$  košů.

*Důkaz.* Instance bude mít  $U = \{u_1, u_2, \dots, u_{18k}\}$ , s těmito prvky asociujeme váhy takto:

$$s(u_i) = \begin{cases} \frac{1}{7} + \varepsilon & 1 \leq i \leq 6k \\ \frac{1}{3} + \varepsilon & 6k < i \leq 12k \\ \frac{1}{2} + \varepsilon & 12k < i \leq 18k \end{cases}$$

kde  $\varepsilon > 0$  je dostatečně malé kladné racionální číslo takové, že  $(\frac{1}{7} + \varepsilon + \frac{1}{3} + \varepsilon + \frac{1}{2} + \varepsilon) \leq 1$  a zároveň  $6 \cdot (\frac{1}{7} + \varepsilon) \leq 1$  a zároveň  $2 \cdot (\frac{1}{3} + \varepsilon) \leq 1$  a zároveň  $(\frac{1}{2} + \varepsilon) \leq 1$ .

Optimální rozdělení rozdělí prvky do  $6k$  košů. Algoritmus FF bude brát prvky jeden po druhém a vytvoří nejprve  $k$  košů, každý s 6 prvky velikosti  $\frac{1}{7} + \varepsilon$ , poté  $3k$  košů, každý se 2 prvky velikosti  $\frac{1}{3} + \varepsilon$  a nakonec  $6k$  košů, každý s jedním prvkem velikosti  $\frac{1}{2} + \varepsilon$ . Dohromady je tedy  $10k$  košů, a tedy

$$\frac{FF(I)}{opt(I)} = \frac{5}{3}$$

$\square$

**Algoritmus 2: First Fit Decreasing (FFD):**

1. **Setříd' předměty sestupně podle velikosti.**
2. Ber jeden předmět po druhém (**od největšího po nejmenší**), pro každý předmět  $u$  najdi první koš, do nějž se tento předmět ještě vejde.
3. Pokud takový koš neexistuje, přidej nový koš obsahující jen předmět  $u$ .

**Věta 1.9.3 (BD).** Pro každou instanci  $I \in D_{BP}$  platí, že  $FFD(I) < \frac{11}{9} \cdot opt(I) + 4$ .

**Věta 1.9.4.** Pro libovolné  $k$  existuje instance  $I \in D_{BP}$ , pro niž je  $opt(I) \geq k$  a FF vytvoří pro  $I$  aspoň  $\frac{11}{9}opt(I)$  košů.

*Důkaz.* Instance bude mít  $U = \{u_1, u_2, \dots, u_{30k}\}$ , s těmito prvky asociujeme váhy takto:

$$s(u_i) = \begin{cases} \frac{1}{2} + \varepsilon & 1 \leq i \leq 6k \\ \frac{1}{4} + 2\varepsilon & 6k < i \leq 12k \\ \frac{1}{4} + \varepsilon & 12k < i \leq 18k \\ \frac{1}{4} - 2\varepsilon & 18k < i \leq 30k \end{cases}$$

kde  $\varepsilon > 0$  je dostatečně malé kladné racionální číslo (prostě aby to všechno zase vyšlo).

Optimální rozdělení rozdělí prvky do  $9k$  košů (zcela zaplněných). Algoritmus FFD vytvoří  $11m$  košů.  $\square$

### 1.9.3 Aproximační schémata

Lze dosáhnout „libovolně malého“ aproximačního poměru? Co kdybychom si mohli aproximační poměr předepsat?

Algoritmus  $ALG$  je **aproximačním schématem** pro optimalizační úlohu  $A$ , pokud na vstupu očekává instanci  $I \in D_A$  a racionální číslo  $\varepsilon > 0$  a na výstupu vydá řešení  $\sigma \in S_A(I)$  s aproximačním poměrem  $(1 + \varepsilon)$ . Tj. pro maximalizační úlohu platí, že

$$opt(I) \leq (1 + \varepsilon) \cdot ALG(I, \varepsilon)$$

a pro minimalizační úlohu platí, že

$$ALG(I, \varepsilon) \leq (1 + \varepsilon) \cdot OPT(I)$$

Označíme  $ALG_\varepsilon$  instanci algoritmu  $ALG$  se zafixovaným  $\varepsilon$ .

Řekneme, že  $ALG$  je **polynomiální aproximační schéma (PAS)**, pokud je pro každé  $\varepsilon$  časová složitost algoritmu  $ALG_\varepsilon$  polynomiální v  $len(I)$ .

Řekneme, že  $ALG$  je **úplně polynomiální aproximační schéma (ÚPAS)**, pokud  $ALG$  pracuje v čase polynomiálním v  $len(I)$  a  $\frac{1}{\varepsilon}$ .

Rozdíl mezi PAS a ÚPAS je v tom, že v případě PAS se může ve funkci odhadující složitost algoritmu  $ALG_\varepsilon$  objevit hodnota  $\frac{1}{\varepsilon}$  i v exponentu, což není možné v případě ÚPAS. Například složitost  $O(n^{\frac{1}{\varepsilon}})$  by byla naprosto v pořádku pro PAS, ale nikoli pro ÚPAS.

#### 1.9.3.1 Úplně polynomiální aproximační schéma pro Batoh

Pro konstrukci aproximačního schématu pro Batoh využijeme pseudopolynomiálního algoritmu ze sekce 1.8.1.1. Ten pracuje v čase  $O(nV)$ , kde  $n$  je počet předmětů a  $V$  je jejich celková cena. Víme, že tento čas by byl polynomiální, kdyby hodnota  $V$  byla omezena nějakým polynomem.

Hodnota  $V$  se do odhadu složitosti dostala jako horní odhad potenciálních cen předmětů v batohu. Pokud provedeme zaokrouhlení cen předmětů a poté jejich přeskálování, můžeme zmenšit součet nových cen předmětů a tím snížit časové nároky algoritmu. Zaokrouhlením však ztratíme optimálnitu výsledku. Čím hrubější zaokrouhlení vstupních cen provedeme, tím rychlejší běh algoritmu dostaneme, ale tím horšího výsledku dosáhneme. Této myšlenky využívá následující algoritmus, kde parametr  $\varepsilon$  určuje míru zaokrouhlení vstupních cen.

```

1   BatohAPX(s , v , B, epsilon ):
2     # s ... pole velikosti predmetu (0 < s[i] <= B pro kazde i )
3     # v ... pole cen predmetu
4     # B ... velikost batohu
5     # ε ... racionalni cislo (apr. pomer)
6
7     n = len(s)
8     m = argmax(v[ i ] for i in range(0 ,n))
9     if ε ≥ n - 1:
10        return {m}
11     t = ⌊log₂(ε·v[m]/n)⌋ - 1
12
13     c = []
14     for i in range(1 ,n):
15         c [ i ] = ⌊v[i]/2^t⌋
16
17     return Batoh(s ,c ,B)  # viz 1.8.1.1

```

Hodnota  $t$  určuje zaokrouhlení a je samozřejmě zvolena tak, aby správně vyšel aproximační poměr

a časová složitost algoritmu. Pokud si odmyslíme celé části, můžeme nahlédnout, že

$$c[i] \approx \frac{v[i]}{2^t} \approx \frac{v[i]}{\frac{\varepsilon \cdot v[m]}{n} \cdot \frac{1}{2}} = \frac{2 \cdot n \cdot v[i]}{\varepsilon \cdot v[m]}$$

Poměr mezi  $v[i]$  a  $v[m]$  určuje, kolik procent zabírá  $v[i]$  vzhledem k  $v[m]$ . Základním cílem přepočtu je pochopitelně zmenšit hodnotu  $v[m]$ , odpovídajícím způsobem se proporcionalně musí zmenšit i  $v[i]$ . Dalším důležitým faktorem je podíl  $\varepsilon/n$ . To vychází z toho, že chceme celkově dosáhnout chybu  $\varepsilon$ , můžeme si na jednom prvku dovolit chybu  $\varepsilon/n$ . Ve výpočtu se uvažuje opačná hodnota podílu, protože chybu chceme dosáhnout v přepočtené hodnotě  $c[i]$  a ne ve  $v[i]$ .

**Věta 1.9.5.** Algoritmus *BatohAPX* pracuje v čase  $O(\frac{1}{\varepsilon}n^3)$ . Pro libovolnou instanci  $I = (s, v, B)$  úlohy *Batoh* a libovolné  $\varepsilon > 0$  platí, že

$$\text{opt}(I) \leq (1 + \varepsilon) \cdot \text{BatohAPX}(I, \varepsilon)$$

Důkaz.

1. **aproximační poměr:**

(a) **Nechť**  $\varepsilon \geq n - 1$ .

Algoritmus vrátí  $\{m\}$ , což je přípustné řešení, neboť předpokládáme  $s[m] \leq B$ . Zjevně platí  $\text{opt}(I) \leq n \cdot [m]$ . Dostáváme:

$$\frac{\text{opt}(I)}{\text{BatohAPX}(I, \varepsilon)} = \frac{\text{opt}(I)}{\frac{v[m]}{2^t}} \leq \frac{n \cdot v[m]}{\frac{v[m]}{2^t}} = n \leq 1 + \varepsilon$$

(b) **Nechť**  $\varepsilon < n - 1$ .

Budeme používat následující vlastnost:

$$x - 1 \leq \lfloor x \rfloor \leq x \quad (1.9.1)$$

i. **Horní a dolní odhad**  $c[i]$ :

$$\frac{v[i]}{2^t} - 1 \leq \left\lfloor \frac{v[i]}{2^t} \right\rfloor \leq \frac{v[i]}{2^t}$$

Dle definice  $c[i]$  a po vynásobení  $2^t$ :

$$v[i] - 2^t \leq c[i] \leq v[i] \quad (1.9.2)$$

ii. **Horní a dolní odhad**  $2^t$ :

Dle definice  $t$  a díky 1.9.1 dostaneme dolní odhad:

$$2^t \geq 2^{\log_2(\frac{\varepsilon \cdot v[m]}{n}) - 2} = \frac{1}{4} \frac{\varepsilon \cdot v[m]}{n} \quad (1.9.3)$$

obdobně pro odhad shora společně s faktrem, že  $\varepsilon < n - 1$

$$2^t \leq 2^{\log_2(\frac{\varepsilon \cdot v[m]}{n}) - 1} = \frac{1}{2} \frac{\varepsilon \cdot v[m]}{n} < \frac{1}{2} \cdot v[m] \quad (1.9.4)$$

iii. **Výsledná množina**  $M$ :

Pro výslednou množinu  $M$  vrácenou algoritmem *BatohAPX* platí díky 1.9.2:

$$M = \text{BatohAPX}(I, \varepsilon) = \sum_{i \in M} v[i] \geq \sum_{i \in M} c[i] \cdot 2^t$$

Nechť  $M^*$  je optimální řešení instance  $I$ . Víme, že  $M$  je optimálním řešením upravené instance  $I' = (s, c, B)$  (se zredukovanými cenami). Pro tuto upravenou instanci je  $M^*$  pouze přípustným řešením (jelikož má nezredukované

ceny). Proto dále platí:

$$\begin{aligned}
\sum_{i \in M} c[i] \cdot 2^t &\geq \sum_{i \in M^*} c[i] \cdot 2^t \\
&\geq \sum_{i \in M^*} (v[i] - 2^t) \\
&= \sum_{i \in M^*} v[i] - \sum_{i \in M^*} 2^t \\
&\geq \text{opt}(I) - n \cdot 2^t
\end{aligned}$$

První nerovnost platí, neboť pro  $I'$  je  $M$  je optimální a  $M^*$  pouze přípustná.  
Druhá nerovnost plyne z 1.9.2. Dohromady tedy dostaváme

$$\text{BatohAPX}(I, \varepsilon) \geq \text{opt}(I) - n \cdot 2^t$$

a jednoduchou úpravou

$$\frac{\text{opt}(I)}{\text{BatohAPX}(I, \varepsilon)} \leq 1 + \frac{n \cdot 2^t}{\text{BatohAPX}(I, \varepsilon)}$$

#### iv. Odhad aproximačního poměru:

Nyní již stačí jen ukázat, že

$$\frac{n \cdot 2^t}{\text{BatohAPX}(I, \varepsilon)} \leq \varepsilon$$

Protože  $M$  je optimálním řešením  $I'$  a  $m$  je přípustným řešením  $I'$ , dostaneme, že

$$\begin{aligned}
\text{BatohAPX}(I, \varepsilon) &= \sum_{i \in M} c[i] \cdot 2^t \geq c[m] \cdot 2^t \\
&\geq v[m] - 2^t \\
&\geq v[m] - \frac{1}{2}v[m] = \frac{1}{2}v[m]
\end{aligned}$$

Druhá nerovnost plyne z 1.9.2, třetí z 1.9.4.

Dohromady s 1.9.4 tedy dostaneme

$$\frac{n \cdot 2^t}{\text{BatohAPX}(I, \varepsilon)} \leq \frac{n \cdot \frac{1}{2} \cdot \frac{\varepsilon \cdot v[m]}{n}}{\frac{1}{2} \cdot v[m]} = \varepsilon$$

#### 2. časová složitost:

Kromě posledního kroku (řádek 17) lze všechny kroky provést v  $O(n)$ . Volání funkce  $\text{Batoh}(s, c, B)$  zabere  $O(nC)$  viz. 1.8.1.1. Stačí tedy odhadnout  $C$ :

$$C = \sum_{i=1}^n c[i] \leq n \cdot c[m] \leq \frac{n \cdot v[m]}{2^t} \leq \frac{n \cdot v[m]}{\frac{1}{4} \cdot \frac{\varepsilon \cdot v[m]}{n}} = \frac{4n^2}{\varepsilon}$$

kde předposlední nerovnost plyne z 1.9.3. Tedy  $C \in O(\frac{1}{\varepsilon}n^2)$ , dohromady tedy  $O(\frac{1}{\varepsilon}n^3)$

□

Algoritmus  $\text{BatohAPX}$  tedy tvoří **úplně polynomiální aproximační schéma** pro úlohu BATOH.

Dá se říci, že úplně polynomiální aproximační schéma je maximum, čeho lze s aproximačními algoritmy dokázat v případě, že nemáme přímo polynomiální algoritmus pro danou úlohu. Techniku použitou v  $\text{BatohAPX}$  lze použít i v mnoha jiných případech, kdy máme k dispozici pseudopolynomiální algoritmus, před jehož použitím jde jen o to vhodně zaokrouhlit vstupní hodnoty.

### 1.9.3.2 Aproximační schémata a NP-úplnost

Postup z předchozí sekce lze i obrátit: pokud se nám pro úlohu A, která splňuje jistá omezení, podaří popsat ÚPAS, můžeme z něj zpětně zkonstruovat i pseudopolynomiální algoritmus pro tuto úlohu.

**Věta 1.9.6.** *Nechť A je optimalizační úloha, jejíž přípustná řešení mají nezápornou celočíselnou hodnotu a nechť existuje polynom q dvou proměnných takový, že pro každou instanci I úlohy A platí, že*

$$opt(I) < q(len(I), max(I))$$

*Pokud existuje úplně polynomiální aproximační schéma pro A, pak existuje i pseudopolynomiální algoritmus pro A.*

*Důkaz.* BÚNO A je maximalizační úloha (minimalizační analogicky). Nechť ALG je ÚPAS pro úlohu A. Pseudopolynomiální algoritmus ALG' řešíci úlohu A postupuje následovně:

1. Pro danou instanci I položíme  $\varepsilon = \frac{1}{q(len(I), max(I))}$
2. Pustíme  $ALG_\varepsilon(I)$ .

Čas výpočtu je polynomiální v  $len(I)$  a  $1/\varepsilon = q(len(I), max(I))$  (z definice ÚPAS), dohromady se tedy jedná o polynomiální algoritmus v  $len(I)$  a  $max(I)$ , tedy o pseudopolynomiální algoritmus. Protože ALG je ÚPAS a protože A je maximalizační úloha, platí

$$opt(I) \leq (1 + \varepsilon) \cdot ALG_\varepsilon(I)$$

tedy

$$opt(I) - ALG_\varepsilon(I) \leq \varepsilon \cdot ALG_\varepsilon(I) \leq \varepsilon \cdot OPT(I) < \frac{q(len(I), max(I))}{q(len(I), max(I))} = 1$$

, kde druhá nerovnost plyne z toho, že A je maximalizační úloha, a třetí nerovnost plyne z definice  $\varepsilon$  a toho, že  $q$  omezuje velikost  $opt(I)$ .

Protože hodnoty přípustných řešení úlohy A jsou *nezáporná celá čísla*, musí ve skutečnosti platit, že

$$opt(I) = ALG_\varepsilon(I)$$

□

**Důsledek 1.9.1.** *Nechť A je silně NP-úplná optimalizační úloha, která splňuje předpoklady věty 1.9.6. Pokud  $P \neq NP$ , pak neexistuje úplně polynomiální aproximační schéma pro úlohu A.*

### 1.9.4 Neapproximovatelnost

V některých případech nemůžeme (pokud  $\mathbf{P} \neq \mathbf{NP}$ ) najít ani polynomiální aproximační algoritmus s konstantním poměrem.

#### TRAVELLING SALESMAN PROBLEM (TSP)

**Instance:** Množina měst  $C = \{c_1, \dots, c_n\}$ , hodnoty  $d(c_i, c_j) \in \mathbb{N}$  přiřazující každé dvojici měst vzdálenost.

**Přípustné řešení:** Permutace měst  $c_{\pi(1)}, c_{\pi(2)}, \dots, c_{\pi(n)}$

**Cíl:** Minimalizovat

$$\left( \sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)}) \right) + d(c_{\pi(n)}, c_{\pi(1)})$$

**Věta 1.9.7.** *Pokud existuje polynomiální aproximační algoritmus ALG pro úlohu obchodního cestujícího s konstantním aproximačním poměrem  $\varepsilon > 1$ , potom  $P = NP$ .*

*Důkaz.* Předpokládejme, že  $ALG$  je polynomiální aproximační algoritmus pro TSP s konstantním aproximačním poměrem  $\varepsilon > 1$ , tzn.  $\forall I \in D_{TSP} : ALG(I) \leq \varepsilon \cdot opt(I)$ . Ukážeme, jak tohoto algoritmu využít k vyřešení NP-úplného problému Hamiltonovské kružnice.

Nechť  $G = (V, E)$  je libovolný graf, v němž chceme najít hamiltonovskou kružnici. Zkonstruujeme instanci  $I$  obchodního cestujícího následujícím způsobem. Množina měst bude  $V$ , vzdálenost  $d(u, v)$  mezi městy  $u$  a  $v$  určíme jako

$$d(u, v) = \begin{cases} 1 & \text{pokud } (u, v) \in E \\ \varepsilon \cdot |V| & \text{jinak} \end{cases}$$

Konstrukce je zřejmě polynomiální a polynomiální je i běh algoritmu  $ALG$  na instanci  $I$ . Pokud v  $G$  existuje hamiltonovská kružnice, potom  $opt(I) = |V|$ , protože k tomu, abychom prošli všechna města, si vystačíme s přechody délky 1. Pokud v  $G$  neexistuje hamiltonovská kružnice, pak i nejlepší řešení v  $I$  musí využít některý přechod délky  $\varepsilon \cdot |V|$ , a proto je v tomto případě  $opt(I) > \varepsilon \cdot |V|$ . Tato nerovnost je ostrá proto, že kružnice má alespoň dvě hrany a všechny mají nenulovou délku. Platí tedy, že  $opt(I) = |V|$  právě když v  $G$  existuje hamiltonovská kružnice, přičemž pokud v  $G$  hamiltonovská kružnice neexistuje, platí  $opt(I) > \varepsilon \cdot |V|$ .

Vzhledem k tomu, že  $opt(I) \leq ALG(I) \leq \varepsilon \cdot opt(I)$  (z definice aproximačního algoritmu), znamená to, že  $ALG(I) \leq \varepsilon \cdot |V|$ , právě když v  $G$  existuje hamiltonovská kružnice (pokud neexistuje, pak  $ALG(I) \geq opt(I) > \varepsilon \cdot |V|$ ). Protože  $ALG$  je polynomiální algoritmus, který takto řeší NP-úplný problém hamiltonovské kružnice, tak  $P = NP$ .  $\square$

**Poznámka:** Pro  $TSP$  s trojúhelníkovou nerovností existuje  $\frac{3}{2}$ -aproximační algoritmus (a dosud není znám žádný lepší). Pro  $TSP$  v euklidovské rovině existuje dokonce polynomiální *aproximační schéma*.

Pokud se začneme zabývat approximací, jsou tedy ve složitosti NP-úplných úloh značné rozdíly (ačkoli z hlediska polynomiální převoditelnosti vypadají jako stejně těžké):

- úlohy, které jsou rychle libovolně dobře approximovatelné (mají ÚPAS) jako například BATOH
- úlohy, které mají aspoň PAS
- úlohy, pro něž máme aproximační algoritmus s konstantním poměrem, ale ne už s libovolným konstantním poměrem, například VRCHOLOVÉ POKRYTÍ, nemají tedy ani PAS
- úlohy, pro něž dokonce nemůžeme najít ani aproximační algoritmus s konstantním poměrem, jako je třeba KLIKA nebo OBCHODNÍ CESTUJÍCÍ, tyto úlohy jsou tedy řešitelné aproximačními algoritmy jen velmi obtížně, pokud vůbec.

Na příkladu Obchodního cestujícího navíc vidíme, že úloha se stává tím jednodušší, čím více víme o jejích instancích (zda splňují trojúhelníkovou nerovnost, zda metrika vzdáleností je euklidovská).

Samozřejmě všechny tyto úvahy a výsledky platí jen za předpokladu, že  $P \neq NP$ .

# Kapitola 2

## Datové struktury

### 2.1 Vyhledávací stromy ((a,b)-stromy, Splay stromy).

#### 2.1.1 Binární vyhledávací strom (BVS)

Binární vyhledávací strom (**BVS**) je *dynamická datová struktura*, v níž platí:

- (*binární strom*) každý uzel má nanejvýš dva potomky – levého a pravého.
- Každému uzlu je přiřazen určitý klíč. Podle hodnot těchto klíčů jsou uzly uspořádány.
- Levý podstrom uzlu obsahuje pouze klíče menší než je klíč tohoto uzlu.
- Pravý podstrom uzlu obsahuje pouze klíče větší než je klíč tohoto uzlu.

##### 2.1.1.1 Operace

Vyhledávací (nemodifikující) operace: FIND(x), MIN(), MAX(), PREC(x), SUCC(x), ...

Modifikující operace: INSERT(x), DELETE(x). Jediný zajímavý případ je DELETE(x) když má  $x$  dva potomky: pak se najde maximum v levém podstromě  $x$  (=PREC(x)) nebo minimum v pravém podstromě  $x$  (=SUCC(x)), jeho obsah se přesune do  $x$  a pak se tento prvek smaže.

Složitost operací obecně závisí na výšce stromu  $h$ . Ta může zdegenerovat až na  $O(n)$ . Proto existují **vyvažovací** stromy, které různými mechanismy udržují výšku na  $O(\log n)$ .

#### 2.1.2 (a,b)-stromy

Nechť  $a, b \in \mathbb{N}$ ,  $a \geq 2$ ,  $b \geq 2a - 1$ . Strom je **(a,b) strom**, když platí:

1. Každý vnitřní vrchol kromě kořene má alespoň  $a$  a nejvýše  $b$  synů.
2. Kořen má alespoň 2 a nejvýše  $b$  synů, nebo je listem.
3. Všechny cesty z kořene do listu jsou stejně dlouhé.

Hodnoty jsou pouze v listech. Vnitřní vrcholy obsahují ukazatele na své podstromy jako pole/spojový seznam klíčů a dále seznam maxim svých podstromů.

**Pozorování:** Bud'  $T$  (a,b) strom s hloubkou  $h$ . Platí

$$2a^{h-1} \leq \text{počet listů } T \leq b^h$$

tedy pro libovolné  $n$  má každý (a,b) strom  $T$  s  $n$  listy hloubku  $h = \Theta(\log n)$ , přesněji

$$\log_b n \leq d \leq 1 + \log_a n$$

Volba parametrů  $a, b$  závisí na použití, např. (2,3) stromy nebo B-stromy, kde  $a, b \approx$  velikost bloku na disku.

### 2.1.2.1 Operace

- **FIND(x)**: Prohledávej od kořene, vyber potomka pro zanoření podle maxim.
- **INSERT(x)**: Najdi vrchol  $v$ , pod který patří  $x$ . Pokud  $v$  má  $b$  potomků, přidej nový list s  $x$  pod  $v$ , hotovo. Pokud má  $v$  právě  $b$  potomků, rozštěp  $v$ : vytvoř nový vrchol  $v'$ , pod který přemístí prvních  $\frac{b+1}{2}$  potomků  $v$  (operace SPLIT). Rekurzivně vlož  $v'$  do rodiče  $v$ .
- **DELETE(x)**: Najdi  $x$ . Nechť  $v$  je rodič  $x$ .
  1. Pokud  $v$  má  $a > b$  potomků, smaž list s  $x$ , hotovo.
  2. Pokud má  $v$  společně se svým levým nebo pravým sousedem  $> 2a$  potomků, přesuň jednoho potomka ze souseda do  $v$  a smaž list s  $x$ . Aktualizuj záznamy u rodiče  $v$ .
  3. Pokud nemá  $v$  ani s jedním sousedem  $> 2a$  potomků, pak smaž  $x$  a přesuň všechny ostatní potomky  $v$  do jednoho ze sousedů (operace JOIN). Rekurzivně smaž  $v$  a přitom aktualizuj informaci o sousedech  $v$  u rodiče  $v$ .
- **JOIN( $T_1, T_2$ )**: Spojí stromy  $T_1$  a  $T_2$  za předpokladu, že  $\max(T_1) < \min(T_2)$ . Nechť  $h(T_1)$  a  $h(T_2)$  označují výšky stromů  $T_1$  resp.  $T_2$ . BÚNO  $h(T_1) \geq h(T_2)$ , jinak analogicky. Nechť  $r = h(T_1) - h(T_2)$ . Připoj syny kořene  $T_2$  k synům posledního vrcholu na hladině  $r$  ve stromě  $T_1$ . Pokud bude mít tento vrchol po sliti více potomků než  $b$ , rozštěp a rekurzivně opravuj stejně jako v případě INSERTu. Časová složitost je úměrná rozdílu výšek.
- **SPLIT( $T, k$ )**: Rozštěpí strom  $T$  na  $T_1, T_2$ , jeden obsahující klíče  $< k$ , druhý klíče  $\geq k$ . Udržujeme si 2 zásobníky  $L$  a  $P$ . Postupujeme jako při FIND( $k$ ). Při průchodu vrcholem  $v$  jej rozštěpíme na 3 podstromy:  $T_L, T_k$  a  $T_P$ , kde  $T_L$  obsahuje podstromy  $v$  (zavěšené pod nově vytvořeným kořenem) s hodnotami  $< k$ ,  $T_k$  je podstrom, kam pokračujeme při hledání a  $T_P$  obsahuje podstromy s hodnotami  $> k$ .  $T_L$  dáme na zásobník  $L$ ,  $T_P$  na zásobník  $P$  a pokračujeme v hledání. Až dojdeme do listu, přidáme jej na zásobník  $P$  a poté z vrcholů na zásobníku  $L$  postupně pospojujeme pomocí operací JOIN nový strom  $T_1$  a z vrcholů na zásobníku  $P$  vytvoříme strom  $T_2$ . Časová složitost  $O(\log n)$ . Nalezení listu je  $O(\log n)$ , JOINování podstromů také: jsou jelikož slučujeme odshora zásobníku, tak suma rozdílů výšek slučovaných dvojic stromů je  $O(\log n)$ .
- **ORD(i)**: Vrátí  $i$ -tý nejmenší prvek ve stromě. Pokud v každém vrcholu udržují aktuální počet listů v daném podstromě, pak lze ORD( $i$ ) provést v  $O(\log n)$ .

### 2.1.2.2 Paralelní verze

Chceme-li (a,b) stromy použít v paralelním prostředí, tj. s více procesy přistupujícími ke stromu, je třeba provést následující úpravy:

- omezit  $b \geq 2a$
- při INSERT a DELETE provádět už cestou dolů preventivní štěpení, tj. pokud při INSERT potkám vrchol s  $b$  potomky, rozštěpím; a pokud při DELETE potkám vrchol s  $a$  potomky, tak bud' přidán potomka od souseda nebo sloučím se sousedem.

Výhodou je, že lze rovnou odemykat navštívené vrcholy, neboť se k nim již nebudu vracet.

### 2.1.2.3 Amortizovaná analýza

Při výstavbě stromu posloupností operací INSERT dojde k  $O(n)$  vyvažovacím operacím (ty jsou  $O(1)$ )

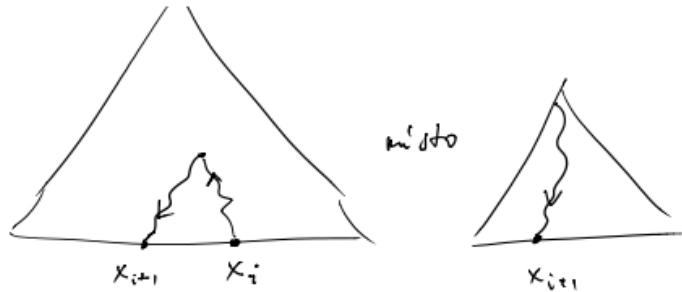
Počet štěpení a slučování vrcholů při  $m$  insertech a  $l$  deletech je  $O(m + l + \log n)$ , tzn. amortizovaně  $O(1)$  štěpení/slučování za operaci. (bez důkazu)

#### 2.1.2.4 A-sort

Třídící algoritmus, který nejprve z prvků sestaví (a,b) strom a poté je vypíše od nejlevějšího listu po nejpravější.

Při vkládání hledáme pozici pro aktuální klíč nikoliv od kořene, ale od předchozího vloženého uzlu (strom s „prstem“ – ukazatelem na posledně použitý list). Zajímá nás počet kroků nutný k nalezení správného místa pro  $x_{i+1}$ :

- $x_{i+1} < x_i$ : cesta nahoru maximálně  $\log_a(|\{j \mid j \leq i; x_{i+1} < x_j\}|)$
- $x_{i+1} > x_i$ : cesta nahoru maximálně  $\log_a(|\{j \mid j \leq i; x_{i+1} < x_j\}|)$  (fakt to vyjde stejně)



Předpokládáme, že při vkládání vrcholů je časově nejnáročnější nalezení místa pro něj, tedy že štěpení při opakovaném vkládání vychází amortizovaně  $O(1)$  na operaci.

Celkový čas je tedy

$$T \leq 2 \cdot \sum_{i=1}^n \log_a(|\{j \mid j \leq i; x_{i+1} < x_j\}|) + O(n)$$

kde první člen odpovídá hledání míst pro vkládané prvky a druhý výpisu posloupnosti (předpokládáme listy propojené spojovým seznamem). To lze díky konkávnosti logaritmu upravit na

$$\begin{aligned} T &\leq 2 \cdot n \cdot \log_a \left( \frac{\sum_{i=1}^n |\{j \mid j \leq i; x_{i+1} < x_j\}|}{n} \right) + O(n) \\ &= O(n \log \frac{F}{n}) \end{aligned}$$

kde  $F = \sum_{i=1}^n |\{j \mid j \leq i; x_{i+1} < x_j\}|$  vyjadřuje celkový počet „inverzí“ v původní posloupnosti.

Jistě platí  $0 \leq F \leq n^2$ , tedy máme zajistěnou i v nejhorším případě složitost  $O(n \log n)$ , v případě částečně setříděné posloupnosti je složitost lepší.

Existuje alternativní varianta algoritmu, v níž se místo pro nový prvek hledá vždy od nejpravějšího/nejlevějšího listu. Složitost vychází stejná.

Často se používají (2,3) stromy, protože mají nejmenší prostorové nároky a vzhledem k tomu, že nepoužíváme DELETE, nejsou ani tolik náročné na vyvažování (při obecném používání si ale právě z důvodu vyvažování vedou lépe (2,4) stromy).

#### 2.1.2.5 B-stromy

Varianta používaná pro velká data uložená na disku, velikost uzlu odpovídá bloku paměti, aby se minimalizoval počet přístupů.

**Rozdíly:**

- hodnoty uloženy i ve vnitřních vrcholech (namísto maxim z podstromů).
- místo  $a, b$  se volí pouze jediná hodnota, **řád stromu**: B-strom řádu  $t$  je  $\approx (t-1, 2t-1)$  strom (nebo  $(t, 2t)$  strom nebo  $(\lfloor t/2 \rfloor, t)$  strom – záleží na konkrétní definici).

## Varianty

**B+ stromy** Data **pouze** v listech (jako u obecných (a,b) stromů), sousední listy spojeny ukazateli. Používá např NTFS.

**B\* stromy** Zaplněnější vrcholy: dolní limit na zaplnění není  $1/2$  maxima, nýbrž  $2/3$  maxima.  
Při plném zaplnění se vrchol hned nerozpadá, nýbrž sdílí klíče se svým sousedem, teprve po zaplnění obou uzlů se tyto dva rozštěpí na 3.

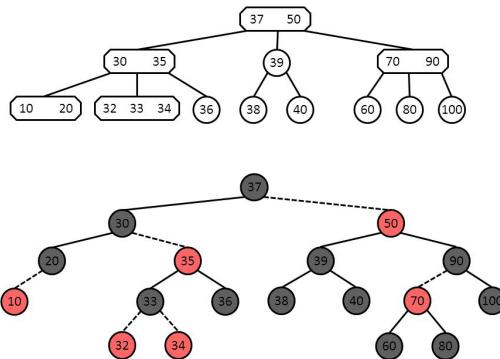
### 2.1.3 Červeno-černé stromy

Binární vyhledávací strom, kde každý vrchol jeobarven buď červeně nebo černě, a platí následující:

1. listy jsou černé
2. červené vrcholy mohou být syny pouze černých vrcholů
3. všechny cesty z libovolně (pevně) zvoleného vrcholu  $x$  do listu v podstromě  $x$  mají stejný počet černých vrcholů

Hodnoty jsou uloženy pouze ve vnitřních vrcholech, listy jsou ve skutečnosti pouze NULL pointery.

Dá se ukázat ekvivalence s (2,4) stromy, často se ČČ používají pro jejich implementaci. Nicméně ekvivalence platí zejména co se týče složitosti, samotná implementace je odlišná, zejména aktualizační operace pro ČČ a reprezentace vrcholů.



Obrázek 2.1: Ekvivalence ČČ a (2,4) stromů na příkladu.

Používají se v knihovnách, např. STL v C++.

**Věta 2.1.1.** Červeno-černý strom obsahující  $n$  prvků má hloubku  $O(\log(n))$ .

*Důkaz.* Nechť  $b$  je počet černých vrcholů na cestě z kořene do listu. Nejmenší možný strom obsahuje pouze černé vrcholy a má tedy  $1 + 2 + 4 + \dots + 2^{b-1} = 2^b - 1$  vrcholů. Největší možný strom sestává z cest, kde se střídají černé a červené vrcholy a má tedy  $1 + 2 + 4 + \dots + 2^{2b-1} = 2^{2b} - 1$  vrcholů. Tedy

$$2^b - 1 \leq n \leq 2^{2b} - 1$$

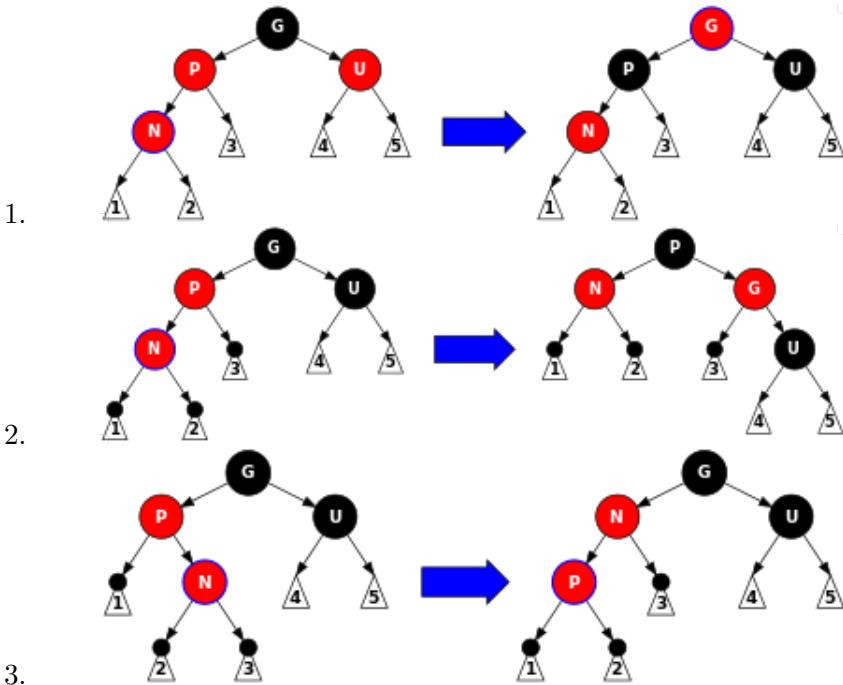
$$b - 1 \leq \log_2 n + 1 \leq 2b$$

Hloubka stromu je nejvýše  $2b \in O(\log_2 n)$

□

#### 2.1.3.1 Operace INSERT(x)

Najdeme místo pro nový vrchol jako v BVS. Přítomný černý NULLový list vymažeme, vložíme nový červený vrchol  $N$  obsahující  $x$  a přidáme mu 2 černé NULLové syny. Podmínka 3 je v pořádku, mohla se ale porušit podmínka 2 (rodič  $N$  je červený). Opravíme:



### 2.1.3.2 Operace DELETE(x)

Odstraňovaný vrchol  $x$  má nejvýše jednoho neNULLového syna (má-li dva, prohodí se s maxinem svého levého podstromu, ten má jistě nejvýše jednoho potomka). Je-li  $x$  červený, OK. Je-li  $x$  černý a jeho potomek červený, OK – odstraníme, přebarvíme syna. Je-li  $x$  černý a potomek také, je třeba opravovat. Odstraníme  $x$  a potomka označíme jako „dvojitě černého“. Tuto barvu navíc propagujeme nahoru, v nejhorším až do kořene, kde lze jednoduše odstranit. Rozbor případů viz třeba wikipedie.

### 2.1.4 Splay stromy

Splay stromy jsou variantou *binárních vyhledávacích stromů* a jedná se o *adaptivní* datové struktury. Základní myšlenkou je **udržovat často používané prvko blízko u kořene**.

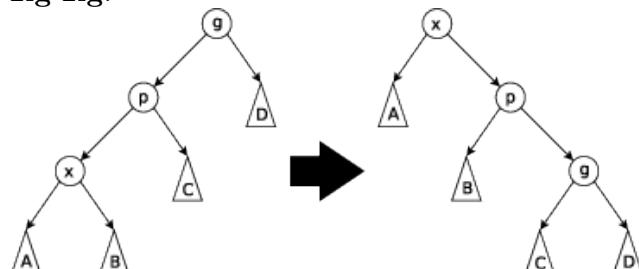
Cena za operaci může být až  $O(n)$ , ale amortizovaná cena za  $k$  operací je  $O(k \log n + n \log n)$  pro  $n$  prvků uložených ve stromě.

#### 2.1.4.1 Operace SPLAY(x)

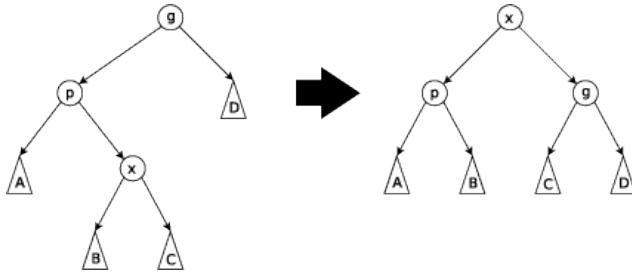
Tato operace přesune prvek  $x$  do kořene pomocí níže uvedených rotací. Aplikuje se po každé operaci FIND, INSERT, ...

Používají se 3 typy rotací:

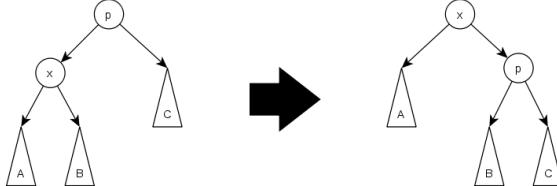
1. zig-zig:



2. zig-zag:



3. zig:



Pokud to lze, používají se dvojrotace (zig-zig nebo zig-zag), jednoduchý zig slouží pro rotaci u kořene.

Časovou složitost rozebereme níže.

#### 2.1.4.2 Operace FIND(x)

Funguje stejně, jako u klasického BVS, jen se na závěr ještě zavolá SPLAY(x).

#### 2.1.4.3 Operace INSERT(x)

Je několik variant:

1. Proved' INSERT(x) stejně, jako u klasického BVS, poté proved' SPLAY(x).
2. Proved' SPLIT(T,x)  $\rightarrow T_1, T_2$ , pak vytvoř nový strom s x jako kořenem a  $T_1, T_2$  jako levým resp. pravým synem.
3. Varianta v Kouckého poznámkách: najdi x, nechť u je poslední vrchol podél cesty k chybějícímu x. Proved' SPLAY(u) a vlož x doleva/doprava pod kořen. Lze rozmyslet, že u je buď bezprostředně menší nebo bezprostředně větší než x.

#### 2.1.4.4 Operace DELETE(x)

Opět více variant:

1. Proved' DELETE(x) stejně, jako u klasického BVS, poté proved' SPLAY(parent(x)).
2. Proved' SPLAY(X), smaž kořen a spoj dva vzniklé podstromy L a R: najdi největší prvek a v L, SPLAY(a), připoj R pod a

#### 2.1.4.5 Amortizovaná analýza

Použijeme *potenciálovou metodu* amortizované analýzy. Amortizovaný čas jedné operace je

$$t = a + \Phi(T') - \Phi(T)$$

kde  $a$  je skutečný čas operace,  $T$  je strom před operací,  $T'$  strom po operaci a  $\Phi(T)$  je potenciál stromu  $T$ , který definujeme takto:

$$\Phi(T) = \sum_{x \in T} r(x), \quad \text{kde } r(x) \text{ je } \log_2(s(x)) \text{ a } s(x) \text{ je počet vrcholů pod } x$$

Hodnota  $r(x)$  vyjadřuje *rank* uzlu  $x$ . Maximální možná hodnota je  $\log_2(n)$ , tj.  $\Phi(T)$  je nejvýše  $n \log_2(n)$ .  $\Phi(T)$  je tím nižší, čím je strom vyváženější.

Amortizovaný čas na  $m$  operací je tedy:

$$\begin{aligned} t_1 + t_2 + \dots + t_m &= \sum_{i=1}^m (a_i + \Phi(T_i) - \Phi(T_{i-1})) \\ &= \sum_{i=1}^m a_i + \Phi(T_m) - \Phi(T_0) \end{aligned}$$

Tedy *reálný čas* na všechny operace  $(\sum_{i=1}^m a_i)$  se od amortizovaného  $(\sum_{i=1}^m t_i)$  liší nejvýše v rozdílu potenciálů před a po sekvenci operací, tj.  $\Phi(T_m) - \Phi(T_0)$ . Tento rozdíl může být nejvýše  $n \log_2(n)$ .

**Věta 2.1.2.** Operace  $SPLAY(x)$  na libovolném vrcholu  $x$  stromu  $T$  zabere amortizovaný čas  $\leq 3(r'(x) - r(x)) + 1$ . Značení  $r'(x)$  vyjadřuje rank vrcholu  $x$  po operaci,  $r(x)$  před operací.

*Důkaz.* Předpokládáme, že provedení rotace samotné zabere čas  $t = 1$ .

Vždy se mění pouze rank zúčastnených vrcholů  $x, p$  (parent) a  $g$  (grandparent). Rozebereme jednotlivé případy:

1. „**zig-zig**“: Jde o dvojrotaci, předpokládáme tedy cenu 2. Cena za operaci je

$$t = 2 + r'(x) - r(x) + r'(p) - r(p) + r'(g) - r(g)$$

Zjevně  $r'(x) = r(g)$  (kořen před a po), tedy

$$t = 2 - r(x) + r'(p) - r(p) + r'(g)$$

Dále si všimneme, že  $r(p) \geq r(x)$  a  $r'(p) \leq r'(x)$ , z čehož plyne

$$\begin{aligned} t &\leq 2 - r(x) + r'(x) - r(x) + r'(g) \\ &= 2 + r'(x) - 2r(x) + r'(g) \end{aligned}$$

Nyní přichází klíčové pozorování. Uvažme výraz  $2r'(x) - r(x) - r'(g)$ . Ten musí být alespoň 2, neboť:

$$\begin{aligned} 2r'(x) - r(x) - r'(g) &= \log_2 \left( \frac{s'(x)}{s(x)} \right) + \log_2 \left( \frac{s'(x)}{s'(g)} \right) \\ &\geq \log_2 \left( \frac{s'(x)}{\frac{s'(x)}{2}} \right) + \log_2 \left( \frac{s'(x)}{\frac{s'(x)}{2}} \right) \\ &= 1 + 1 = 2 \end{aligned}$$

Druhá nerovnost je dána díky  $s'(x) \geq s(x) + s'(g)$  (viz obrázek) a protože díky vlastnostem logaritmu je nejnižší možná hodnota dosažena právě při volbě  $s(x) = s'(g) = \frac{s'(x)}{2}$ . Aplikujeme-li toto pozorování na předchozí nerovnici, dostaneme

$$\begin{aligned} t &\leq 2 + r'(x) - 2r(x) + r'(g) \\ &\leq (2r'(x) - r(x) - r'(g)) + r'(x) - 2r(x) + r'(g) \\ &= 3(r'(x) - r(x)) \end{aligned}$$

2. „**zig-zag**“: Postupujeme obdobně. Cena za operaci je

$$t = 2 + r'(x) - r(x) + r'(p) - r(p) + r'(g) - r(g)$$

Zjevně  $r'(x) = r(g)$  (kořen před a po) a  $r(x) \leq r(p)$  a tedy

$$t \leq 2 - 2r(x) + r'(p) + r'(g)$$

Zopakujeme pozorování z předchozího případu, tentokrát však pro  $2r'(x) - r'(p) - r'(g)$ . Stejně jako v předchozím případě plyne  $2r'(x) - r'(p) - r'(g) \geq 2$  a tedy

$$\begin{aligned} t &\leq 2 - 2r(x) + r'(p) + r'(g) \\ &\leq (2r'(x) - r'(p) - r'(g)) - 2r(x) + r'(p) + r'(g) \\ &= 2(r'(x) - r(x)) \end{aligned}$$

### 3. „zig“:

Zde máme pouze  $x$  a  $p$ . Cena za operaci je tedy

$$t = 1 + r'(x) - r(x) + r'(p) - r(p)$$

Jelikož rank  $p$  jistě klesne ( $r'(p) - r(p) < 0$ ) a rank  $x$  naopak stoupne ( $r'(x) - r(x) > 0$ ), platí

$$\begin{aligned} t &= 1 + r'(x) - r(x) + r'(p) - r(p) \\ &\leq 1 + r'(x) - r(x) \\ &\leq 1 + 3(r'(x) - r(x)) \end{aligned}$$

□

**Důsledek 2.1.1.** *Posloupnost  $m$  operací zabere amortizovaný čas*

$$m \cdot O(\log_2 n) + O(n \log_2 n)$$

*Důkaz.* Díky předchozí větě a odhadu  $\forall x : r(x) \leq \log_2(n)$  máme amortizovanou složitost na jednu operaci  $O(\log_2 n)$ , pro  $m$  operací tedy  $m \cdot O(\log_2 n)$ . Druhá složka  $O(n \log_2 n)$  je horním odhadem na rozdíl potenciálů před a po sekvenči operací.

Operace FIND, INSERT i DELETE lze provést tak, že sestávají z operace SPLAY a poté již jen několika akcí složitosti  $O(1)$ . □

## 2.1.5 AVL stromy

Binární vyhledávací strom je **AVL strom**, když pro každý uzel  $x$  ve stromě platí

$$|(\text{výška levého podstromu uzlu } x) - (\text{výška pravého podstromu uzlu } x)| \leq 1$$

**Věta 2.1.3.** *Výška AVL stromu s  $n$  vrcholy je  $O(\log n)$ .*

*Důkaz.* Nechť  $N(h)$  je minimální počet vrcholů stromu výšky  $h$ . Pak platí:

$$\begin{aligned} N(0) &= 1 \\ N(1) &= 2 \\ N(h) &= 1 + N(h-1) + N(h-2) \end{aligned}$$

Zřejmě  $N(h) \geq 2 \cdot N(h-2)$  (protože  $N(h-2) \leq N(h-1)$ ).

Dále platí:  $N(h) \geq 2^{\frac{h}{2}}$ . To lze dokázat indukcí díky předchozímu pozorování:

$$N(h) \geq 2 \cdot N(h-2) \geq 2 \cdot 2^{\frac{h-2}{2}} = 2 \cdot 2^{\frac{h}{2}-1} = 2^{\frac{h}{2}}$$

Z toho dostaneme  $h \leq 2 \log N(h)$  □

Dotazovací operace mají tedy složitost  $O(\log n)$ .

Modifikující operace Insert a Delete pracují stejně jako na normálním BVS s případným doatečným vyvážením pomocí rotací. Při Insert je třeba  $O(1)$  rotací, při Delete to může být až  $O(\log n)$ . Celková složitost obou operací je  $O(\log n)$ .

## 2.2 Haldy (regulární, binomiální).

### 2.2.1 Regulérní halda

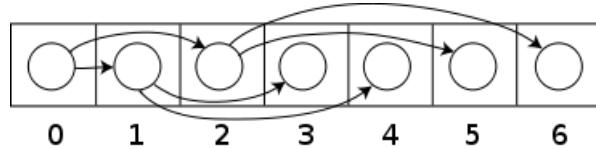
Halda je **stromová** datová struktura splňující tzv. **vlastnost haldy**: pokud je  $B$  potomek  $A$ , pak  $x(A) \leq x(B)$ . To znamená, že v kořenu je vždy prvek s nejnižším klíčem. Taková halda se pak označuje jako *min heap*, opakem je *maxheap*.

Regulérní halda je co nejefektivněji zaplněná, přesněji všechny patra haldy jsou zcela zaplněné kromě nejspodnějšího, to je zaplněno zleva.

Typicky uvažujeme *binární* neboli **2-regulární** haldu, ale obecně může být halda  $k$ -regulární.

#### 2.2.1.1 Reprezentace

Haldy lze díky jejich zaplněnosti dobře implementovat pomocí pole, kde potomci uzlu na  $n$ -té pozici se nachází na pozici  $2n$  a  $(2n + 1)$  (v případě binární haldy).



Libovolné pole prvků lze rychle převést na reprezentaci haldy – lze ukázat, že to jde v  $O(n)$  (operace HEAPIFY()):

*Důkaz.* Nechť  $n = 2^{h+1} - 1$  pro nějaké  $h$  (tj. stavíme kompletní strom hloubky  $h$ ). Haldu stavíme od spodních pater. Budeme zkoumat, o kolik nejvýše pater musí prvky „probublat“, aby se dostaly na správnou pozici

- poslední patro (hloubka  $h$ ) má  $2^h$  prvků, žádný z nich ale nikam neprobublává
- patro  $h - 1$  má  $2^{h-1}$  prvků, každý probublá nejvýše o 1 patro (do listu)
- patro  $h - 2$  má  $2^{h-2}$  prvků, každý probublá nejvýše o 2 patro (do listu)
- ...

Celkový počet operací je tedy:

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

Jelikož řada  $\sum_{i=0}^{\infty} \frac{i}{2^i}$  konverguje, lze ji shora omezit nějakou konstantou  $C$  (lze ukázat  $C = 2$ ) a tedy počet operací je  $C \cdot 2^h$ , tedy  $O(n)$ .  $\square$

#### 2.2.1.2 Operace

- **INSERT(x)**:  $O(\log n)$

Prvek je přidán na konec posledního patra a poté se opraví halda, tj. prvek probublává směrem ke kořeni, dokud není opravena „haldovitost“. Složitost je daná výškou haldy, která je  $O(\log n)$

- **MIN()**:  $O(1)$

Vrátí vrchol haldy (bez mazání).

- **DELETE-MIN()**:  $O(\log n)$

Smaže vrchol haldy. Na jeho místo se přesune poslední prvek (nejpravější v posledním patře; poslední prvek pole) a opraví se halda: přesunutý prvek probublává z kořene směrem dolů, dokud není napravena „haldovitost“.

## 2.2.2 Binomiální halda

Motivace: chceme rychlejší INSERT(x).

**Binomiální halda** je ve skutečnosti množina více jednotlivých haldově uspořádaných stromů (obecných, ne binárních), každý velikosti  $2^k$  pro nějaké  $k$ . Vždy si pamatujeme, ve kterém stromě aktuálně leží nejmenší prvek.

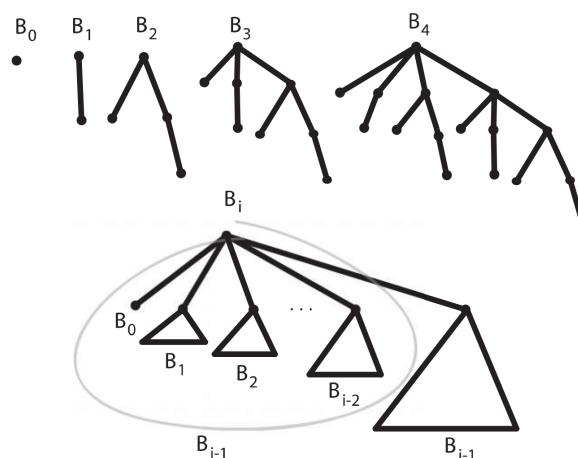
Existují 2 varianty:

1. **zbrklá varianta** Pro každé  $k$  existuje v haldě vždy nejvýše 1 strom velikosti  $2^k$ . Halda je implementována jako *pole* odkazů na jednotlivé stromy, kde  $i$ -tý odkaz ukazuje na strom s velikostí  $2^i$ . Operace:

- **INSERT(x)**:  $O(\log n)$ , amortizovaně  $O(1)$  (za jistých podmínek)

Přidán nový haldový strom velikosti  $1 = 2^0$  obsahující  $x$ . Aktualizuje se ukazatel na strom s nejnižším prvkem, je-li třeba.

Opraví se struktura: dokud existují nějaké 2 stromy stejné velikosti  $2^i$ , spoj je do jednoho velikosti  $2^{i+1}$ . Vždy se přivésí jeden strom pod kořen toho druhého jako nejpravější potomek. To tedy implikuje strukturu hald jednotlivých velikostí:



Slévání funguje podobně jako *binární sčítání*.

- **MIN()**:  $O(1)$

Vrátí vrchol stromu s minimálním prvkem (na který si pamatujeme ukazatel).

- **DELETE-MIN()**:  $O(\log n)$

Smaže kořen stromu s minimálním prvkem. Pokud má tento strom  $2^i$  prvků, rozpadne se na stromy velikosti  $2^0, 2^1, \dots, 2^{i-1}$ . Ty přidáme do haldy a podobně jako při INSERT opravíme haldu slučováním stromů stejné velikosti

INSERT trvá stále  $O(\log n)$ , nicméně vložení  $n$  prvků do prázdné haldy trvá pouze  $O(n)$  – tedy amortizovaně máme INSERT v  $O(1)$ , pokud neprovádíme DELETE-MIN. Stačí rozmyslet, že slití stromů velikosti  $2^i$  proběhne právě  $\frac{n}{2^i}$ -krát, dohromady

$$\sum_{i=0}^{\log n} \frac{n}{2^i} \leq n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

2. **líná varianta** Bez omezení na počet stromů stejné velikosti. Halda implementována jako *spojovalý seznam* odkazů na stromy.

Operace:

- **INSERT(x)**:  $O(1)$

Přidán nový haldový strom velikosti  $1 = 2^0$  obsahující  $x$ . Aktualizuje se ukazatel na strom s nejnižším prvkem, je-li třeba. (Struktura se neopravuje.)

- **MIN()**:  $O(1)$

Vrátí vrchol stromu s minimálním prvkem (na který si pamatujeme ukazatel).

- **DELETE-MIN()**:  $O(n)$ , amortizovaně  $O(\log n)$

Smaže kořen stromu s minimálním prvkem. Pokud má tento strom  $2^i$  prvků, rozpadne se na stromy velikosti  $2^0, 2^1, \dots, 2^{i-1}$ . Ty přidáme do haldy a opravíme haldu slučováním stromů stejné velikosti, dokud to lze.

V praxi: vytvoříme prázdné pole délky  $\log n$  (což je maximální počet různých velikostí stromů), poté postupně bereme stromy ze spojového seznamu a vkládáme na správnou pozici. Je-li obsazena, sloučíme a vložíme o pozici výše, zde případně pokračujeme ve slučování. Jakmile vložíme všechny stromy, vytvoříme z pole opět spojový seznam.

**DELETE-MIN** je v nejhorším případě  $O(n)$ , amortizovaně však  $O(\log n)$ , což ukážeme potenciálovou metodou:

Nechť  $|T|$  značí počet stromů v haldě  $T$ , pak zvolme  $\Phi(T) = C \cdot |T|$ , kde  $C$  je vhodná konstanta. Amortizovaná složitost operací:

- **INSERT**:  $C + \Phi(T') - \Phi(T) = C + C = O(1)$  (přibyl 1 nový strom)
- **MIN**:  $C + 0 = O(1)$  (žádný strom nepřibyl ani neubyl)
- **DELETE-MIN**:  $C \cdot |T| + \Phi(T') - \Phi(T) = C \cdot |T| + O(\log n) - C \cdot |T| = O(\log n)$

Počet stromů po slučování je nejvýše  $\log n$ .

### 2.2.3 Fibonacci halda

Chceme přidat operaci **DECREASE-KEY**. Motivace: *Dijkstrův algoritmus*. Šlo by prostě v případě porušení haldovitosti probublat ke kořeni v  $O(\log n)$ , my ale chceme  $O(1)$ .

Ostatní operace budou fungovat stejně, jako u líné binomiální haldy.

#### 2.2.3.1 Implementace

Každý vrchol si udržuje své potomky ve spojovém seznamu, jejich počet a zda-li již o nějakého potomka přišel (označení). Celkově tedy z každého uzlu vedou ukazatele na: rodiče, sourozence a potomky.

#### 2.2.3.2 Operace **DECREASE-KEY(v, k)**

Sníží hodnotu vrcholu  $v$  o  $k$ .

1. Pokud je zachována haldovitost (tj. hodnota  $v$  je stále větší než hodnota rodiče), hotovo.
  2. Pokud je porušena haldovitost, odtrhní podstrom  $v$  a zařadí ho do seznamu stromů. Byl-li označen, ozdzač.
- (a) Pokud rodič *není* označen, ozdzač jej. Hotovo.
  - (b) Pokud rodič *je* označen (tj. již o nějakého potomka přišel), rekurzivně odtrhní rodiče.

Každý rodič může přijít nejvýše o jednoho potomka, pak je sám odtržen. Kořen může přijít o libovolně mnoho synů.

Časová složitost amortizovaně  $O(1)$  (i přestože může nastat kaskáda řezů). Ukážeme potenciálovou metodou: V předchozím případě jsme uvažovali potenciál  $\Phi(T) = C \cdot |T|$ . Upravíme jej na

$$\Phi(T) = C \cdot (|T| + 2 \cdot \#\text{označených vrcholů})$$

Jelikož pouze **DECREASE-KEY** operuje s označenými vrcholy, analýza ostatních operací zůstává stejná.

Nechť **DECREASE-KEY** provede  $k$  řezů. V takovém případě dojde k ozdzačení  $k$  vrcholů ( $\Delta\Phi_- = 2Ck$ ), přidání  $k$  stromů do haldy ( $\Delta\Phi_+ = Ck$ ) a označení jednoho nového vrcholu ( $\Delta\Phi_+ = 2C$ ). Amortizovaná cena operace je tedy

$$C \cdot k + \Delta\Phi = C \cdot k - 2C \cdot k + C \cdot k + 2C = 2C = O(1)$$

### 2.2.3.3 Analýza DELETE-MIN

Jako řád stromu označíme počet potomků kořene. Tato definice funguje i u binomiální haldy, tam dokonce platila korespondence, že strom řádu  $k$  má velikost  $2^k$ . To u Fibonacciho haldy platit nebude, protože stromy budeme prořezávat.

Složitost  $O(\log n)$  u binomiálních stromů vycházela z předpokladu, že maximální řád stromu je  $O(\log n)$ . To u Fibonacciho haldy není tak zjevné.

**Věta 2.2.1.** Strom ve Fibonacciho haldě řádu  $k$  má nejméně  $F_{k+2}$  vrcholů, kde  $F_i$  je  $i$ -té Fibonacciho číslo.

*Důkaz.* Indukcí. Stačí si nakreslit maximálně „poškozený“ strom, jeho podstromy jsou maximálně poškozené stromy nižších řádů.

Využíváme faktu  $\sum_{i=1}^n F_i = F_{n+2} - 1$  (indukcí). □

**Věta 2.2.2.** Pro každé  $n \geq 3$  platí  $F_n \geq (\frac{5}{4})^n$ .

*Důkaz.* Indukcí. □

Lze dokoncě dokázat  $F_n \geq \varphi^{(n-2)}$ , kde  $\varphi$  je zlatý řez.

**Věta 2.2.3.** Maximální řád stromu ve Fibonacciho haldě je  $O(\log n)$ .

*Důkaz.* Plyne z předchozích 2 vět. □

## 2.2.4 Další haldy

### 2.2.4.1 Pairing heap

Fibonacciho haldy se považují za pomalé, kvůli vysokým konstantním faktorům u všech operací. **Pairing heap** je „zjednodušení“, které udržuje téměř stejně dobrou asymptotickou složitost operací, ale v praxi je rychlejší.

Operace:

**MIN**  $O(1)$  : vrat kořen haldy

**MERGE**  $O(1)$  : porovnej kořeny slučovaných stromů, zavěs větší pod menší

**INSERT**  $O(1)$  : vytvoř nový strom pro vložený element a proved' MERGE s původní haldou

**DECREASE-KEY (optional)**  $O(\log n)$  : odstraň strom zakořeněný v daném prvku, sniž klíč, proved' MERGE s původní haldou

**DELETE-MIN:**  $O(\log n)$  odstraň kořen a sluč vzniklé podstromy. Různé strategie slučování.

### 2.2.4.2 Brodal heap, strict Fibonacci heap

Obě struktury mají stejnou složitost operací jako Fibonacci, ale zaručují je i v **nejhorším případě**. Jejich konstrukce je ale složitá a v praxi jsou nepoužitelné (podle slov autorů)

## 2.3 Hašování, řešení kolizí, univerzální hašování, výběr hašovací funkce.

Převzato z matfyz wiki (konvertováno pandocem).

### 2.3.1 Základní pojmy

Základní motivací pro hashování je *slovníkový problém*, kdy máme za úkol reprezentovat množinu  $S$  prvků z nějakého univerza  $U$  a provádět na ní následující operace:

- **MEMBER (FIND)** (je třeba, aby tato operace probíhala velmi rychle)
- **INSERT**

- **DELETE**

Triviální řešení: pole o velikosti  $U$ . To je však nepoužitelné v případě, že  $|S| \ll |U|$  (a navíc  $U$  může být neúnosně velké).

Použijeme *hashovací funkci*  $h : U \rightarrow \{0, \dots, m - 1\}$  a množinu  $S$  reprezentují **hašovací tabulkou** = polem s  $m$  políčky tak, že  $x \in S$  je uložen na indexu  $h(x)$ . Předpokládejme, že funkce  $h$  se dá spočítat v čase  $O(1)$  – jiné funkce nemají smysl.

Hlavní problém hašování jsou **kolize**:  $x \neq y, h(x) = h(y)$ . Existují různé strategie **řešení kolizí** a **předcházení kolizím** vhodnou volbou hašovací funkce (univerzální hašování, …)

Pro následující analýzy si označíme:

- $n = |S|$
- $N = |U|$
- $m = \text{velikost hašovací tabulky}$
- $\alpha = \frac{n}{m} = \text{faktor zaplnění (load factor)}$

### 2.3.2 Řešení kolizí

Obecně jsou dvě strategie: *zřetězení záznamů*, kdy je každý *bucket* nezávislý a obsahuje ukazatel na nějaký spojový seznam; a *otevřené adresování*, kdy jsou všechny prvky přímo v tabulce (ne nutně na nařízeném poli). Mnoho variant je nějakou hybridní kombinací.

#### 2.3.2.1 Hashování se separovanými řetězci

V tomto typu hashování se kolize řeší *řetězením ve spojových seznamech*: pro každé políčko založíme zvlášť spoják všech prvků, které se do něj hashují. V nejhorském případě mají všechny prvky stejný hash a máme jen jeden spojový seznam.

Paměťová náročnost je pro každý seznam  $O(1 + l(i))$ , kde  $l(i)$  je délka toho seznamu.

Existují dvě varianty – **neuspořádaná** a **s uspořádanými prvky** v řetězích. Liší se jedině v očekávaném počtu testů pro neúspěšné hledání (když dojdu v řetězci za místo, kde by byl hledaný prvek, můžu skončit).

Pro odhad složitosti algoritmu předpokládáme, že:

- Hashovací funkce  $h$  rozděluje data rovnoměrně
- Sama reprezentovaná množina  $S$  je náhodný výběr z  $U$  s rovnoměrným rozdělením

Tyto předpoklady v praxi ale splněny být nemusí.

##### 2.3.2.1.1 Očekávaná průměrná délka řetězců

Pro odhad složitosti se počítá **očekávaná délka řetězců**. Označme délku  $i$ -tého řetězce jako  $l(i)$ . To je náhodná veličina s *binomickým rozdělením*, tj.  $l(i) \sim Bi(n, \frac{1}{m})$  a pravděpodobnost, že tento řetězec má délku  $l$ , odpovídá:

$$P(l(i) = l) = \binom{n}{l} \left(\frac{1}{m}\right)^l \left(1 - \frac{1}{m}\right)^{n-l}$$

Toto je jen approximace (pro nekonečnou velikost univerza i seznamů), pro případ, že  $N \gg n^2m$ , ale lze použít. Očekávaná délka řetězce pak odpovídá střední hodnotě binomického rozdělení, tj.  $n \frac{1}{m} = \alpha$ . Jinými slovy, délka řetězce na libovolné pozici odpovídá celkovém faktoru zaplnění tabulky.

Rozptyl vyjde  $\frac{n}{m} \left(1 - \frac{1}{m}\right)$ , opět ze vzorce pro binomické rozdělení.

Očekávaná délka *nejdelšího* řetězce je  $\Theta(\frac{\log n}{\log \log n})$  (odvození je hodně technické, lze nalézt ve Finkových slajdech).

### 2.3.2.1.2 Očekávaný počet testů (= porovnání) při hledání

- **úspěšné hledání:** odpovídá průměrnému počtu testů provedených při *vložení* každého z prvků, tj.  $(1 + \text{očekávaná délka řetězce při každém vkládání})$ :

$$\frac{1}{n} \sum_{i=0}^{n-1} \left( 1 + \frac{i}{m} \right) = 1 + \frac{1}{n} \frac{n(n-1)}{2m} = 1 + \frac{\alpha}{2} - \frac{1}{2m} \approx 1 + \frac{\alpha}{2}$$

- **neúspěšné hledání:** v podstatě odpovídá průměrné délce řetězce, až na jeden člen navíc, který odpovídá situaci, kdy je řetězec prázdný – i v tom případě totiž musím jeden test provést.

$$E(T) = P(\mathbf{l}(i) = 0) + \sum_l l P(\mathbf{l}(i) = l) = \left( 1 - \frac{1}{m} \right)^n + \frac{n}{m} \approx e^{-\alpha} + \alpha$$

S uspořádanými řetězci končím dřív:  $e^{-\alpha} + 1 + \frac{\alpha}{2} - \frac{1}{\alpha}(1 - e^{-\alpha})$ .

### 2.3.2.2 Hashování s přemísťováním/přesuny

Nevýhodou separovaných řetězců je nutnost alokovat další paměť, to je neefektivní. Proto zavedeme do hashovací tabulky pomocné ukazatele a celé řetězce nacpeme přímo do ní (a zřetězené prvky prostě rozházíme na jiné adresy). Pro *hashování s přemísťováním* se v tabulce uchovává navíc jednoduše odkaz na *předchozí a následující prvek* řetězce.

Při **INSERTu** na místo, kde už je nějaký prvek z *jiného* řetězce, přehodíme tento cizí prvek jinam (přitom přesměrujeme příslušné ukazatele).

Při **DELETE** prvního prvku řetězce je nutné na jeho místo přesunout druhý (pokud existuje).

Očekávaný počet testů je stejný jako pro hashování se separovanými řetězci. Přemísťování v tabulce je ale náročnější než 1 test, proto jsou **INSERT** a **DELETE** pomalejší.

### 2.3.2.3 Hashování se dvěma ukazateli

Od předchozího se liší tím, že používá pouze dopředné odkazy a na poli  $h(i)$  je odkaz BEGIN na pole tabulky, kde skutečně začíná řetězec prvků s hashem  $h(i)$ . Řetězec tak už nemusí začínat na indexu svého hashe.

Místo přesouvání prvků algoritmy mění BEGIN (ten je na  $j$ -tém políčku vyplněn, právě když existuje řetězec prvků s hashem  $j$ ).

- **INSERT** všechno vkládá na konec řetězce, zakládá-li nový, do BEGIN (na místě určené hashem) píše, kde se ve skutečnosti nachází.
- **DELETE** jen upravuje odkazy na následující, nebo BEGIN (pokud smazáním zanikne řetězec).

Kvůli tomu, že řetězce začínají jinde než na svém místě, je počet testů o něco větší:

- Úspěšné hledání:  $1 + \frac{(n-1)(n-2)}{6m^2} + \frac{n-1}{2m} \approx 1 + \frac{\alpha^2}{6} + \frac{\alpha}{2}$
- Neúspěšné hledání přibližně  $1 + \frac{\alpha^2}{2} + \alpha + e^{-\alpha}(2 + \alpha) - 2$ .

### 2.3.2.4 Srůstající (coalesced) hashování

Srůstající hashování používá jen jeden ukazatel v hashovací tabulce navíc – odkaz na další prvek NEXT. Řetězce tak mohou obsahovat hodnoty s různými hashy. Prvek  $s$  vkládáme vždy do řetězce, který prochází  $h(s)$ -tým políčkem v tabulce.

Existují různé varianty:

- standardní (bez pomocné paměti) – LISCH, EISCH
- s pomocnou pamětí – LICH, VICH, EICH.

#### 2.3.2.4.1 Bez pomocné paměti – LISCH a EISCH

*LISCH* je „late insertion“, tedy přidává se za poslední prvek řetězce. *EISCH* („early insertion“) přidává za první prvek řetězce.

- **MEMBER** je stejný pro oba (jen projití řetězce po odkazech NEXT).
- **INSERT**: projití celého řetězce a pokud vkládaný prvek v řetězci není, vložení na libovolné volné místo v tabulce a připojení
  - LISCH: na konec řetězce.
  - EISCH: za první prvek (pokud je řetězec neprázdný).

Algoritmy **DELETE** nejsou známy, kromě primativních. Problémem je u nich zachování náhodného uspořádání prvků v řetězcích, které se předpokládá pro dodržení očekávaných časů operací. Je ale možné také prvky jen označit jako odstraněné a jejich místa použít při vkládání dalších (to ale zpomaluje hledání).

*EISCH* je kupodivu o něco rychlejší na úspěšné vyhledání (je větší pravděpodobnost práce s novým prvkem), očekávaný počet testů je stejný.

#### 2.3.2.4.2 S pomocnou pamětí – LICH, VICH, EICH

V této variantě rozdělíme paměť na dvě části:

- (hash-funkcí) přímo adresovatelná
- pomocná část (bez přístupu hash-funkcí) – tzv. „sklep“

Při kolizích nejdříve ukládáme do řádků z pomocné části, pak teprve do přímo adresovatelné, tedy oddalujeme srůstání řetězců. Chování se tak až do určitého okamžiku podobá separovaným.

Existují tři varianty podle chování algoritmu **INSERT**:

- LICH vždy přidává na konec řetězce
- EICH v případě neprázdného řetězce vždy za 1. prvek
- VICH na konec, pokud řetězec končí v pomocné paměti, jinak na místo, kde řetězec pomocnou paměť opustil (tj. chová se na pomocné paměti jako LICH a v přímo adresovatelné části jako EICH).

Algoritmy až na VICH se chovají stejně jako ve standardním srůstajícím hashování, rozhodující je výběr volného řádku pro vložení: např. „vždy vyber z nejvyšší adresy“ může zaručit používání pomocné paměti.

Také tu není přirozené efektivní **DELETE**.

Na hledání volného řádku se v praxi hodí např. spojový seznam volných řádků.

#### 2.3.2.5 Lineární přidávání

Nepoužívá ukazatele v hašovací tabulce.

V případě kolize při **INSERTu** nalezneme nejbližší vyšší volné políčko a vloží nový prvek tam. Předpokládáme „cyklickou“ paměť, tj. když dojdeme na konec, vkládáme od začátku.

Problémem je tvoření *shluků* – při velkém zaplnění se operace dost zpomalují. Také nepodporuje efektivní **DELETE** – bud' označit místo jako smazané nebo přehašovat.

*Očekávaný počet testů* je  $\frac{1}{2} \left( 1 + \left( \frac{1}{1-\alpha} \right)^2 \right)$  v neúspěšném a  $\frac{1}{2} \left( 1 + \left( \frac{1}{1-\alpha} \right) \right)$  v úspěšném případě (bez důkazu).

Variantou je přičítat konstantu vyšší než 1, tj. obecně  $h(x) + ai$  (částečně zabraňuje shlukům) nebo použít **kvadratické přidávání**:  $h(x) + ai + bi^2$ .

### 2.3.2.6 Dvojité hashování

*Dvojité hashování* je vylepšení předchozí metody tak, aby nevznikaly shluky. Výběr následujícího řádku bude závislý na předchozím, ale s rovnoměrným rozložením. Na to použijí *druhou hashovací funkci*  $h_2$ .

Při operacích **INSERT** pak hledám nejmenší  $i$  od 0, že  $(h_1(x) + i \cdot h_2(x)) \bmod m$  je volné políčko, tj. postupně přičítám  $h_2(x)$  a modulím. Stejný postup je i pro operaci **MEMBER**.

Je nutné, aby  $h_2(x) \not\equiv m$ , tj. abych měl prosté posloupnosti (a z každého políčka tak mohl vést řetězec po celé tabulce). Idea, že iterace  $h_2$  tvorí pro každé  $x$  náhodnou permutaci paměťových míst, není úplně přesná, ale v praxi stačí, aby z  $h_1(x) = h_1(y)$  plynulo, že  $h_2(x)$  a  $h_2(y)$  budou odlišné.

Funkce navíc musíme volit "chytré" (i lineární přidávání je spec. příp. dvojitého hashování, kdy  $h_2 \equiv 1$ ). Pak tato metoda je znatelně rychlejší než lin. přidávání. Předpoklad náhodnosti použitý v teoretické analýze sice splnit nelze, ale přiblížit se mu ano.

#### 2.3.2.6.1 Očekávaný počet testů

Předpokládáme, že iterování funkce  $h_2$  tvoří náhodné permutace (což, jak bylo řečeno, není úplně přesné). Počet testů označíme  $C(m, n)$ .

- **neúspěšný případ:** Označme  $q_i(n, m)$  pravděpodobnost, že při zaplnění  $\frac{n}{m}$  je při **INSERT(x)** pro nějaké  $x$  obsazeno prvních  $i - 1$  políček, kam bych ho mohl vložit. Potom

$$q_i(n, m) = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+1}{m-i+1} = \frac{\prod_{j=0}^{i-1} (n-j)}{\prod_{j=0}^{i-1} (m-j)}$$

a tedy

$$q_i(n, m) = \frac{n}{m} q_{i-1}(n-1, m-1)$$

Očekávaný počet testů je (předposlední rovnost plyne z rekurentního vztahu pro  $q_j$ , poslední krok dokázat indukcí):

$$\begin{aligned} C(n, m) &= \sum_{j=0}^n (j+1)(q_j(n, m) - q_{j+1}(n, m)) \\ &= \sum_{j=0}^n (q_j(n, m)) = 1 + \frac{n}{m} C(n-1, m-1) \\ &= \frac{m+1}{m-n+1} \approx \frac{1}{1-\alpha} \end{aligned}$$

- **úspěšný případ** – stejná metoda jako u dřívějších analýz, takže vychází:

$$\frac{1}{n} \sum_{i=0}^{n-1} C(i, m) = \frac{1}{n} \sum_{i=0}^{n-1} \frac{m+1}{m-i+1} \approx \frac{1}{\alpha} \ln \left( \frac{m+1}{m-n+1} \right) \approx \frac{1}{\alpha} \ln \left( \frac{1}{1-\alpha} \right)$$

### 2.3.2.7 Kukaččí hašování

Používáme 2 hašovací funkce  $h_1$  a  $h_2$ , prvek  $x$  musí být uložen buď v příhrádce  $h_1(x)$ , nebo  $h_2(x)$ . V každé příhrádce je nejvýše jeden prvek.

Operace FIND a DELETE jsou triviálně  $O(1)$ .

Při operaci **INSERT** postupuj takto:

---

**Algorithm 1**


---

```

1: if  $T[h_1(x)] = x$  or  $T[h_1(x)] = x$  then
2:   return
3:   pos  $\leftarrow h_1(x)$ 
4:   for  $n$  times do
5:     if  $T[pos]$  empty then
6:        $T[pos] \leftarrow x$ 
7:       return
8:     swap( $x, T[pos]$ )
9:     if pos =  $h_1(x)$  then
10:      pos  $\leftarrow h_2(x)$ 
11:    else
12:      pos  $\leftarrow h_1(x)$ 
13: rehash(); insert( $x$ )

```

---

Zaříznout je možno i dříve, cca po  $\Omega(\log n)$  krocích.

Při rehashování náhodně vygenerujeme nové funkce  $h_1, h_2$  a můžeme zvětšit velikost tabulky. Vložíme všechny prvky (pozor, tady může dojít k vnořenému rehashování – pozor na ztrátu prvků!)

#### 2.3.2.7.1 Analýza

Kukačí graf funguje dobře pro  $m \approx 2.3n$ . My budeme pracovat s grafy, pro něž platí  $m \leq 2cn, c > 1$ . (Pozn.: Koucký používal  $m = 6n$ , tj.  $c = 3$ ).

Při analýze budeme pracovat s **kukačcím grafem**, kde

- vrcholy jsou příhrádky tabulky
- hrany jsou dvojice  $(h_1(x), h_2(x)), \forall x \in S$

**Věta 2.3.1.** Nechť  $S \subseteq U, |S| = n, m \leq 2cn, c > 1$ . Pravděpodobnost, že pro náhodně zvolené hašovací funkce  $h_1, h_2$  jsou pozice  $i, j$  spojeny cestou délky  $k$ , je nejvýše  $\frac{1}{mc^k}$ .

*Důkaz.* Induction on  $k$ :

- $k = 1$  For one element, the probability that it forms an edge  $ij$  is  $2/m^2$ . So, the probability that there is an edge  $ij$  is at most  $\frac{2n}{m^2} \leq \frac{1}{mc}$
- $k > 1$  There exists a path between  $i$  and  $j$  of length  $k$  if there exists a path from  $i$  to  $u$  of length  $k - 1$  and an edge  $uj$ . For one position  $u$ , the  $i - u$  path exists with probability  $\frac{1}{mc^{k-1}}$ . The conditional probability that there exists the edge  $uj$  if there exists  $i - u$  path is at most  $\frac{1}{mc}$  because some elements are used for the  $i - u$  path. By summing over all positions  $u$ , the probability that there exists  $i - j$  path is at most

$$m \frac{1}{mc^{k-1}} \frac{1}{mc} = \frac{1}{mc^k}$$

□

**Věta 2.3.2.** Nechť  $S \subseteq U, |S| = n, m \leq 2cn, c > 1$ . Pravděpodobnost, že pro náhodně zvolené hašovací funkce  $h_1, h_2$  obsahuje kukačkový graf cyklus je nejvýše  $\frac{1}{2}$ .

*Důkaz.* Pravděpodobnost cyklu délky  $k$  vedoucího přes  $i$  je dle předchozí věty  $\frac{1}{mc^k}$ .

Pravděpodobnost cyklu libovolné délky je  $\sum_{k>1} \frac{1}{mc^k} \leq \frac{1}{m} \frac{1}{c-1}$ .

Pravděpodobnost, že nějaký vrchol  $i$  je obsažen v cyklu je  $m \frac{1}{m} \frac{1}{c-1} = \frac{1}{c-1}$

□

**Důsledky:** Složitost operace Insert bez přehašování je  $O(1)$ . Pro  $c > 2$  je počet přehašování při vkládání  $n$  prvků  $O(1)$ . Amortizovaná složitost Insert včetně přehašování pro  $c > 2$  je rovněž  $O(1)$ .

### 2.3.2.8 Srovnání

Podle počtu testů:

	neúspěšné	úspěšné
1.	separované uspořádané řetězce	separované (usp. i neusp.) řetězce, přemístování
2.	separované řetězce, přemístování	dva ukazatele
3.	dva ukazatele	VICH
4.	VICH, LICH	LICH
5.	EICH	EICH
6.	LISCH, EISCH	EISCH
7.	dvojité hashování	LISCH
8.	lineární přidávání	dvojité hashování
9.		lineární přidávání

- VICH je při vhodném  $\alpha$  lepší než hashování se dvěma ukazateli.
- Lineární přidávání se nedá použít pro  $\alpha > 0.7$ , dvojité hashování pro  $\alpha > 0.9$ .
- Separované řetězce a obecné srůstající hashování používají víc paměti, přemístování a dvojité hashování zas víc času, tj. nelze říct, které je jednoznačně lepší.

### 2.3.2.9 Implementační dodatky

- Pro hledání volných řádků se většinou používá seznam (zá sobník).
- Přeplnění se většinou řeší držením  $\alpha$  v rozumném intervalu ( $\langle 1/4, 1 \rangle$ ) a přehashováním do jinak velké tabulky ( $2^i \cdot m$ ) při pře- nebo podtečení
- V praxi se doporučuje přehashování odkládat (např. pomocnými tabulkami) a provádět při nečinnosti systému.

**DELETE** se ve strukturách, které ho nepodporují, řeší označením políčka jako smazaného s možností využití při vkládání. V případě, že polovina polí je blokována tímto způsobem, se vše přehashuje. Pro srůstající hashování se toto používat nemusí, máme metody na zachování náhodnosti rozdelení dat.

### 2.3.2.10 Přehašování

Při velkém zaplnění tabulky je u některých metod nutné přehašovat. V praxi se to dělá tak, že máme sadu hašovacích funkcí  $h_0, h_1, \dots$ , kde  $h_0$  hašuje do tabulky velikosti  $2^0 m = m$ , funkce  $h_1$  do tabulky velikosti  $2^1 m$ , obecně  $h_i$  hašuje do tabulky velikosti  $2^i m$ .

Spolu s množinou  $S$  si pak udržujeme i aktuální  $i$  takové, že

$$2^{i-2}m < |S| < 2^i m$$

a  $S$  je aktuálně hašována funkcí  $h_i$ .

Při operacích **INSERT** a **DELETE** vždy kontrolujeme přetečení či podtečení této podmínky a případně zvětšíme/zmenšíme  $i$  a přehašujeme.

Amortizovaná složitost přehašování je  $O(1)$  (je ale vhodné jej dělat mimo dobu aktivního používání).

### 2.3.3 Univerzální hashování

Problém pevné hašovací funkce: pokud  $N > mn$ , tak pro každou hašovací funkci  $h$  existuje  $S \subseteq U$  velikosti  $n$  taková, že  $h$  hašuje všechny prvky  $S$  do jedné příhrádky.

Budeme chtít sestrojit množinu funkcí  $H : U \rightarrow M$ , takový, že náhodně zvolená funkce  $f \in H$  hešuje libovolnou množinu  $S$  „většinou dobře“.

*Definice:* Systém funkcí  $H = \{h_i; i \in I\} : U \rightarrow \{0, \dots, m-1\}$  je univerzální, pokud:

$$\forall x, y \in U, x \neq y : |\{i \in I; h_i(x) = h_i(y)\}| \leq \frac{|I|}{m}$$

. Tj. zaručuje se, že pro každé dva různé prvky má maximálně 1 funkce kolizi.

Jiná formulace: pro náhodnou  $h$  z  $H$  a pro  $\forall x, y \in U, x \neq y$  platí

$$P[h(x) = h(y)] \leq \frac{1}{m}$$

*Definice:* Systém funkcí  $H = \{h_i; i \in I\} : U \rightarrow \{0, \dots, m-1\}$  je  $c$ -univerzální, pokud:

$$\forall x, y \in U, x \neq y : |\{i \in I; h_i(x) = h_i(y)\}| \leq \frac{c|I|}{m}$$

Tj. zaručuje se, že pro každé dva různé prvky má maximálně  $c$  funkci kolizi.

Jiná formulace: pro náhodnou  $h$  z  $H$  a pro  $\forall x, y \in U, x \neq y$  platí

$$P[h(x) = h(y)] \leq \frac{c}{m}$$

*Definice:* Systém funkcí  $H = \{h_i; i \in I\} : U \rightarrow \{0, \dots, m-1\}$  je  $k$ -nezávislý, pokud pro náhodně zvolenou  $h \in H$ :

$$P[\bigwedge_{i=1}^k h(x_i) = z_i] = O\left(\frac{1}{m^k}\right)$$

pro všechna po dvou různá  $x_1, \dots, x_k \in U$  a všechna  $z_1, \dots, z_k \in M$ .

*V tomhle je bordel. Koucký používá  $k$ -univerzální záměnně s  $k$ -nezávislý, wiki taky, nicméně Fink a ještě nějaký jiný zdvoj to má takhle.*

### 2.3.3.0.1 Existence $c$ -univerzálních systémů

Předpokládejme, že universum má tvar  $U = \{0, 1, \dots, N-1\}$  kde  $N$  je nějaké prvočíslo a vezmeme funkce typu

$$h_{a,b}(x) = ((ax + b) \mod N) \mod m$$

Jsou dobré použitelné, protože se dají počítat rychle. Protože  $N$  je prvočíslo, můžeme pracovat v  $\mathbb{Z}_N$ , což je těleso. Rovnice  $h_{a,b}(x) = h_{a,b}(y)$  je ekvivalentní s:

$$\exists i \in \{0, \dots, m-1\} \wedge \exists r, s \in \{0, \dots, \lceil \frac{N}{m} \rceil - 1\} : (ax + b \equiv i + rm) \mod N \wedge (ay + b \equiv i + sm) \mod N$$

Z Frobeniových vět o jednoznačnosti řešení lineárních rovnic plyne, že pro každé  $r, s, i$  existuje jen jedna dvojice  $a, b$ , které vyhovuje. Počet řešení soustavy je tedy omezený číslem  $m \cdot \lceil \frac{N}{m} \rceil^2$  ( $i$ :  $m$  hodnot,  $r, s$ :  $\lceil \frac{N}{m} \rceil$  hodnot pro daná  $x, y$ ).

Pak je systém  $c$ -univerzální pro  $c = (\lceil \frac{N}{m} \rceil)^2 / (\frac{N}{m})^2$  a jeho velikost odpovídá  $N^2$ .

### 2.3.3.0.2 Vlastnosti

Vyrobíme si pomocnou funkci

$$\delta_i(x, y) = \begin{cases} 1 & \text{pro } h_i(x) = h_i(y), x \neq y \\ 0 & \text{jinak} \end{cases}$$

Chceme potom spočítat součet  $\delta_i(x, S) = \sum_{y \in S} \delta_i(x, y)$ . Z výsledku vidíme očekávanou délku řetězce pro libovolnou (jednu) množinu dat. Tohle pak sečtu přes všechny mé hash-funkce a z  $c$ -univerzality dostanu

$$\sum_{i \in I} \delta_i(x, S) = \sum_{y \in S} \sum_{i \in I} \delta_i(x, y) \leq \sum_{y \in S, x \neq y} c \frac{|I|}{m} = \begin{cases} (|S| - 1)c \frac{|I|}{m} & \text{pro } x \in S \\ |S|c \frac{|I|}{m} & \text{jinak} \end{cases}$$

Z toho dopočítám (podělením  $|I|$ ) horní odhad očekávaného  $\delta_i(x, S)$ .

Výsledek: očekávaný čas operací **MEMBER**, **INSERT** a **DELETE** v  $c$ -univerzálním hashování je  $O(1 + c\alpha)$  (kde faktor naplnění  $\alpha = \frac{|S|}{m}$ ). Čas  $n$  po sobě jdoucích operací na původně prázdné tabulce je  $O(n(1 + \frac{c}{2}\alpha))$ . To není lepší hodnota než mají separované řetězce ( $O(1 + \alpha)$ ), ale u nich předpokládám rovnoramenné rozdělení dat.

Výběr vhodné funkce není úplně jednoduchý, protože funkcí může celkem být např až  $N^2$ , tj. nelze ho provést jednoduchým zavoláním generátoru náhodných čísel, nýbrž např. náhodným vybráním každého bitu indexu funkce. Proto je výhodné najít co nejmenší  $c$ -univerzální systémy (viz dále).

### 2.3.3.0.3 Dolní odhady velikosti univ. systémů

Očíslujme hash-funkce z  $I$  a induktivně definujme množiny  $U_i$  jako největší podmnožiny  $U_{i-1}$  takové, že  $h_{i-1}(U_i)$  je jednoprvková. Platí  $|U_i| \geq \lceil \frac{U_{i-1}}{m} \rceil$ , tedy  $|U_i| \geq \lceil \frac{N}{m^i} \rceil$  – velikost těchto množin klesá s logaritmem a  $|I| \geq \frac{m}{c}(\lceil \log_m N \rceil - 1)$ . Takže velikost univ. systému roste alespoň úměrně logaritmu velikosti univerza.

### 2.3.3.0.4 Dolní odhad c

*5-univerzální systém:* Zvolme  $t \in \mathbb{N}$  a k němu vezměme  $t$ -té prvočíslo  $p_t$  tak, že  $t \ln p_t \geq m \ln N$ . Definujme systém funkcí  $H = \{g_{c,d,l}(x) | t < l \leq 2t, c, d \in \{0, 1, \dots, p_{2t}-1\}\}$  kde  $((c(x \bmod p_l) + d) \bmod p_{2t}) \bmod m$ . Zřejmě  $|H| = tp_{2t}^2$ .

Odhadem  $|G| = \{(c, d, l); h_{c,d,l}(x) = h_{c,d,l}(y)\}$ , když si množinu rozdělíme na  $G_1 = \{(c, d, l) \in G; x \bmod p_l \neq y \bmod p_l\}$  a  $G_2 = \{(c, d, l) \in G; x \bmod p_l = y \bmod p_l\}$ , se dá dokázat, že systém je 5-univerzální, za dalších podmínek i 3.25-univerzální.

*Dolní odhad c:* Platí:  $c > 1 - \frac{m}{N}$ . Spočítáme  $\sum_{h \in H} \sum_{x,y \in U} \delta_h(x, y)$  – pro pevnou  $h$  máme (z Cauchy-Schwarzovy nerovnosti,  $u_{i,h} = |\{x \in U, h(x) = i\}|$ ):

$$\sum_{x,y \in U} \delta_h(x, y) = \sum_{i=0}^{m-1} u_{i,h}(u_{i,h} - 1) \geq \frac{(\sum_{i=0}^{m-1} u_{i,h})^2}{m} - N = \frac{N^2}{m} - N$$

Tedy  $\sum_{h \in H} \sum_{x,y \in U} \delta(x, y) \geq \frac{|H|N(N-m)}{m}$ . Zároveň platí  $\sum_{h \in H} \sum_{x,y \in U} \delta(x, y) \leq \sum_{x,y \in U} c \frac{|H|}{m} = N^2 c \frac{|H|}{m}$ , což mi dává výsledek.

### 2.3.4 Perfektní hashování

Základní ideou *perfektního hashování* je nalézt hash-funkci, která pro danou množinu  $S$  nedělá žádné kolize, takže operace **MEMBER** bude velice rychlá. Potom je nevýhoda, že nelze přirozeným způsobem realizovat operaci **INSERT**, tj. v praxi se nesmí moc často vyskytovat.

Tabulka by neměla být o mnoho větší než množina  $S$  a funkce  $h$  rychle spočitatelná a její realizace nezabírat moc paměti (tedy žádná zadávání tabulkou).

#### 2.3.4.0.1 Definice

- Hashovací funkce  $h$  je perfektní pro množinu  $S$ , pokud pro  $\forall x, y \in S, x \neq y : h(x) \neq h(y)$
- Soubor funkcí  $H : U \rightarrow \{0, \dots, m-1\}$  je  $(N, m, n)$ -perfektní, pokud  $\forall S \subseteq U$  takové, že  $|S| = n$  existuje  $h \in H$  perfektní pro  $S$ .

#### 2.3.4.1 Odhadování velikosti

Každá funkce  $h$  je perfektní pro  $\sum \{\prod_{j=0}^{n-1} |h^{-1}(i_j)| ; 0 \leq i_0 < \dots < i_{n-1} < m\}$  množin (sčítáme přes všechny množiny hashů  $h(S)$  a pro každou z nich uvažujeme všechny možnosti, jak mohla vzniknout). Z Cauchy-Schwarzovy nerovnosti plyne, že tento výraz nabývá maxima, když  $\forall i : |h^{-1}(i)| = \frac{N}{m}$ . Každá funkce je tedy perfektní pro max.  $\binom{m}{n} (\frac{N}{m})^n$  množin. Z toho plyne:

$$|H| \geq \frac{\binom{N}{n}}{\binom{m}{n} \left(\frac{N}{m}\right)^n}$$

Jiný odhad lze provést jako u  $c$ -univ. systémů s očíslovanými funkcemi  $|H| = \{h_1, \dots, h_t\}$ . Používám induktivně definované množiny  $U_i$ , kde  $U_0 = U$  a  $U_i$  je největší podmnožina  $U_{i-1}$ , kde je zrovna funkce  $h_i$  konstantní. Dostáváme  $|U_i| \geq \frac{|U_{i-1}|}{m}$ , tj.  $|U_t| \geq \frac{N}{m^t}$ , ale z perfektnosti plyne  $|U_t| \leq 1$ . Dostáváme  $t \geq \frac{\log N}{\log m}$ .

### 2.3.4.2 Existence

Reprezentujme soubor funkcí  $H = \{h_1, \dots, h_t\}$  na univerzu velikosti  $N$  pomocí matice  $M(H)$  typu  $N \times t$ , takže  $M(H)_{x,i} = h_i(x)$ , tj. v jednom sloupci jsou výsledky jedné hashovací funkce pro všechny prvky univerza.

Pak žádná funkce z  $H$  není perfektní pro množinu  $S \subset U$ , když podmatice  $M(H)$  tvořená řádky příslušejícími prvkům  $S$  nemá prostý sloupec. Takových matic je maximálně (počet všech funkcí minus počet prostých, to celé krát libovolné doplnění na  $N$  řádek):

$$\left( m^n - \prod_{i=0}^{n-1} (m-i) \right)^t \cdot m^{(N-n)t}$$

Podmnožin  $U$  velikosti  $n$  je pak  $\binom{N}{n}$ , čímž vynásobeno mám počet matic neodpovídajících  $(N, m, n)$ -perfektnímu systému. Všech matic je  $m^{Nt}$ . Potom existuje  $(N, m, n)$ -perfektní systém, když:

$$\binom{N}{n} \left( m^n - \prod_{i=0}^{n-1} (m-i) \right)^t \cdot m^{(N-n)t} < m^{Nt}$$

Příšernými kejklemi dostaneme podmínu existence  $t \geq n(\ln N)e^{\frac{n^2}{m}}$ .

### 2.3.4.3 Konstrukce funkce

Chceme splnit rychlou spočitatelnost a paměťovou nenáročnost. Předpokládáme univerzum prvočíselné velikosti a funkce typu:

$$h_k(x) = (kx \mod N) \mod m$$

Označme  $b_i^k = |\{x \in S; (kx \mod N) \mod m = i\}|$ . Potom pokud  $h_k(x)$  není perfektní, pak nějaké  $b_i^k = 2$  a mám  $\sum_{i=0}^{m-1} (b_i^k)^2 \geq n+2$ .

Odhadnu výraz  $\sum_{k=1}^{N-1} (\sum_{i=0}^{m-1} (b_i^k)^2) - n = \sum_{x \neq y \in S} \{k; 1 \leq k < N, h_k(x) = h_k(y)\}$ . Z vlastností modula mám takových  $k$  pro daná  $x, y$  nejvýše  $2\lfloor \frac{N}{m} \rfloor = 2\lfloor \frac{N-1}{m} \rfloor$ . Dostávám tedy odhad  $2(N-1)\frac{n(n-1)}{m}$  a z něj vidím, že existuje takové  $k$ , že  $\sum_{i=0}^{m-1} (b_i^k)^2 \leq \frac{2n(n-1)}{m} + n$ , tedy pro tabulku velikosti  $m > n(n-1)$  mám perfektní funkci.

Dá se dokázat trochu slabší předpoklad, že  $P(k; \sum_{i=0}^{m-1} (b_i^k)^2 < \frac{3n(n-1)}{m} + n) \geq 1/4$ , který je základem pravděpodobnostního algoritmu.

Pak mám deterministický algoritmus, který pro  $m = n(n-1) + 1$  nalezne perfektní  $h_k$  v čase  $O(nN)$  a pravděpodobnostní, který pro  $m = 2n(n-1)$  najde perfektní  $h_k$  v čase  $O(n)$ . Mám tedy konstrukci perfektní hash-funkce, ta ale nesplňuje požadavek na malou tabulkou ( $m = \Theta(n^2)$ ).

#### 2.3.4.3.1 Menší tabulka

Zmenšíme-li velikost tabulky na  $m = n$ , bude výše uvedený algoritmus schopný nalézt funkci, pro kterou platí  $\sum (b_i^k)^2 < 3n$  ( $\sum (b_i^k)^2 < 4n$  v pravděpodobnostní variantě). Každou kolizi pak můžeme "rozstrkat" perfektní funkcí nad miniaturní tabulkou a celková velikost všech tabulek bude mnohem menší:

- Vezmeme nalezenou funkci a najdeme všechny neprázdné množiny  $S_i = \{s \in S; h_k(s) = i\}$
- Pro jím odpovídající  $c_i = |S_i|(|S_i| - 1) + 1$  (dvojnásobek v pravděp. metodě) najdeme  $k_i$  takové, že  $h_{k_i}$  je perfektní funkce pro  $S_i$  do tabulky velikosti  $c_i$ .
- Definujme  $d_i = \sum_{j=0}^{i-1} c_j$ , potom pokud  $h_k(x) = l$ , pak  $g(x) = d_l + h_{kl}(x)$  je perfektní, její hodnota spočitelná v čase  $O(1)$  a hashuje do tabulky velikosti  $O(3n)$  ( $O(6n)$  s pravděp. případě), je naleznutelná v čase  $O(nN)$  ( $O(n)$ ) a pro její uložení do paměti jsou potřeba hodnoty  $k$  a  $k_i$ , vyžadující  $O(n \log N)$  paměti.

Pro výpočet  $g(x)$  potřebuji 2 násobení, 2 modulo a 1 sčítání (pro  $d_i$  v paměti), tabulka má velikost  $\sum c_i \leq \sum (b_i^k)^2 < 3n$ . Taková funkce ale stále nesplňuje požadavek na málo paměti pro uložení.

#### 2.3.4.3.2 “Malá” funkce

Víme, že pro  $m \in \mathbb{N}$  je počet prvočísel, která ho dělí  $O(\frac{\log m}{\log \log m})$ . Z toho úvahou o dělitelích čísla  $D = \prod_{1 \leq i < j \leq n} (s_j - s_i) \leq N^{n^2}$  na  $n$ -prvkové  $S$  a vzorce hustoty prvočísel  $p_t \leq 2t \ln t$  dostanu, že existuje  $p$  o velikosti  $O(\ln D) = O(n^2 \ln N)$  takové, že  $\phi_p(x) = x \pmod p$  je perfektní pro  $S$ .

Deterministické nalezení trvá  $O(n^3 \log n \log N)$  (test perfektnosti každého systému je  $O(n \log n)$ ). Proto použijeme pravděpodobnostní algoritmus (mezi  $4cn^2 \ln N$  přír. číslu je aspoň  $1/2$  prvočísel, která vyhovuje): nejdřív najde prvočíslo a pak testuje perfektnost. Očekávaný počet testů je  $O(\ln(4cn^2 \ln N))$ , celková složitost algoritmu je pak  $O(n \log n (\log n + \log \log N))$ . Najde zhruba až  $2\times$  větší prvočíslo než deterministický.

Tuto funkci použijeme ke konstrukci výsledné hash-funkce:

1. Nalezneme prvočíslo  $q_0$ , aby  $\phi_{q_0}(x) = x \pmod {q_0}$  byla perfektní pro  $S$
2. Položíme  $S_1 = \{\phi_{q_0}(s) | s \in S\}$ , pak najdeme prvočíslo  $q_1 \in \langle n(n-1)+1, 2n(n-1)+2 \rangle$
3. K němu existuje  $l \in \langle 1, q_0-1 \rangle$  takové, že  $h_l(x) = ((lx \pmod {q_0}) \pmod {q_1})$  je perfektní pro  $S_1$
4. Položíme  $S_2 = \{h_l(s) | s \in S_1\}$  a najdeme perfektní  $g$  pro  $S_2$  do tabulky s méně než  $3n$  řádky (viz výše, počítá se ale do univerza o velikosti  $q_1$ ).
5. Pak  $f(x) = g(h_l(\phi_{q_0}(x)))$  je perfektní.

Funkce  $q_0$  je určena 1 číslem o velikosti  $O(n^2 \log N)$ ,  $h_l$  2 číslu o velikosti  $O(n^2)$  a  $O(q_0)$ ,  $g$  je určená  $n+1$  číslu o  $O(q_1)$ , tj. celkem zadání vyžaduje  $O(n \log n + \log n + \log \log N)$  paměti.

## 2.4 Analýza nejhoršího, amortizovaného a očekávaného chování datových struktur.

Uvažujeme dva „zdroje“: **prostor** a **čas**. Využití obou zdrojů závisí na *velikosti vstupních dat*. To je

- *formálně*: počet bitů nutných k zapsání vstupních dat
- *neformálně*: počet nějakých elementárních entit na vstupu, např. počet vrcholů a hran grafu, počet měst pro TSP, počet prvků vstupní posloupnosti, kterou chceme setřídit atd.

Většinou nám stačí neformální chápání velikosti vstupu, občas se ale hodí rigoróznější pohled – například při definici číselných problémů a pseudopolynomiálních algoritmů. Velikost vstupu značíme typicky  $n$ .

**Časová složitost** je pak nějaká funkce  $f$  na  $n$ , která udává *počet kroků algoritmu* v závislosti na  $n$ . Za *krok algoritmu* považujeme triviální operace proveditelné v konstantním čase, např. sčítání, násobení, porovnání nebo přiřazení. Opět by šlo chápat definici rigorózněji a počítat třeba operace nějakého abstraktního stroje (Turingův, RAM).

**Prostorová složitost** je opět nějaká funkce  $f$  na  $n$ , která udává *velikost použité paměti* v závislosti na  $n$ . Jednotkou paměti může být třeba byte.

Typicky nás nezajímá přesný tvar funkce  $f$  (tj. multiplikativní a aditivní konstanty), ale pouze do jaké „třídy“ funkce patří (lineární, kvadratická, exponenciální, ...).

### 2.4.1 Asymptotická notace

Intuitivně: zkoumá „chování“ algoritmu na „velkých“ datech, tj. nebene v úvahu multiplikativní a aditivní konstanty, pouze zařazuje algoritmy do „kategorií“ podle jejich skutečné časové složitosti.

Značení:

- $f(n) \in O(g(n)) \dots f(n)$  je **asymptoticky menší nebo rovna**  $g(n)$   
 $\exists c > 0 \exists n_0 > 0 \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)$
- $f(n) \in \Omega(g(n)) \dots f(n)$  je **asymptoticky větší nebo rovna**  $g(n)$   
 $\exists c > 0 \exists n_0 > 0 \forall n \geq n_0 : 0 \leq c \cdot g(n) \leq f(n)$
- $f(n) \in \Theta(g(n)) \dots f(n)$  je **asymptoticky stejná jako**  $g(n)$   
 $\exists c > 0 \exists d > 0 \exists n_0 > 0 \forall n \geq n_0 : 0 \leq c \cdot g(n) \leq f(n) \leq d \cdot g(n)$
- $f(n) \in o(g(n)) \dots f(n)$  je **asymptoticky ostře menší**  $g(n)$   
 $\forall c > 0 \exists n_0 > 0 \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)$
- $f(n) \in \omega(g(n)) \dots f(n)$  je **asymptoticky ostře větší**  $g(n)$   
 $\forall c > 0 \exists n_0 > 0 \forall n \geq n_0 : 0 \leq c \cdot g(n) \leq f(n)$

### 2.4.2 Prostorová (paměťová) složitost reprezentované struktury

Je důležitá, ale obvykle jednoduchá na spočítání a není šance ji vylepšit – jedině použít úplně jinou strukturu.

### 2.4.3 Časová složitost operací na datové struktuře

#### 2.4.3.1 Časová složitost v nejlepším případě

Chování v optimálních podmínkách. Analýza nejlepšího případu sama o sobě moc nevypovídá, ale může přinést odhad spodní hranice možného zlepšení.

#### 2.4.3.2 Časová složitost v nejhorším případě

Její znalost nám zaručí, že nemůžeme být nepřijemně překvapeni (dobou běhu algoritmu). Hodí se pro interaktivní režim – uživatel sedící u databáze průměrně dobrou odezvu neocení, ale jediný pomalý případ si zapamatuje a bude si stěžovat.

Občas je těžké určit „nejhorší“ případ. Často se spíš uvažuje nejhorší možný chod algoritmu (nejvíce cyklů, ...), aniž bychom znali konkrétní vstup, pro nějž by se takto choval (ani nemusí existovat).

Bezpečná analýza, ale pesimistická.

Za vylepšení nejhoršího případu obvykle platíme zhoršením průměrného případu.

Mnohdy může dobrá implementace vést k tomu, že k nejhorším případům nedochází (hašovací tabulky).

#### 2.4.3.3 Očekávaná časová složitost

Je to vlastně vážený průměr – složitost každého případu vstupních dat násobíme pravděpodobností jeho výskytu a sečteme. Je zajímavá pro dávkový režim zpracování. Například Quicksort patří mezi nejrychlejší známé třídící algoritmy, ale v nejhorším případě má složitost kvadratickou.

Mnoho problémů se špatnou složitostí nejhoršího případu se chovají dobře pro průměrné případy. To mnohdy vyhovuje (quicksort), někdy naopak chceme, aby *všechny* instance byly „špatné“ (kryptografie).

#### 2.4.3.4 Amortizovaná složitost

**Amortizovaná analýza** je analýza nejhoršího případu pro **sekvenci operací**. Tím můžeme dostat těsnější odhad na složitost jedné operace (v rámci sekvence operací), než kdybychom analyzovali každou operaci zvlášť. To nás zajímá proto, že některé datové struktury mají takovou vnitřní organizaci, že na ní závisí složitost, a ta organizovanost se během posloupnosti operací mění. Nejhorší případ vlastně uklízí za následující nebo předchozí rychlé případy.

Existují 3 základní techniky pro amortizovanou analýzu:

1. **agregovaná analýza:** Analyzuje celkový čas běhu pro posloupnost operací.
2. **účetní (bankéřova) metoda:** Za operace musíme něco platit, za ty jednodušší méně, ušetřený kredit se použije na ty náročnější.
3. **potenciálová metoda:** Definujeme vhodnou nezápornou *potenciálovou funkci* (nebo prostě *potenciál*), která charakterizuje „uklizenost“ struktury. Operace tuto hodnotu zvyšují nebo snižují.

*Příklady:*

- **Binární čítač**

Máme  $n$ -bitový čítač s operací Inkrement, která přičte jedničku – tzn. nejzazší *nulový bit se změní na 1* a všechny následující jedničkové byty se změní na nula. Počet změněných bitů je v nejhorším případě  $n$ . Zajímá nás, kolik bitů se změní při *posloupnosti k operací Increment* a jaký je tedy celkový čas posloupnosti operací.

1. **Agregovaná analýza:** Poslední bit se změní při každé operaci, tj.  $k$ -krát. předposlední bit se změní při každé druhé operaci, tedy v průměru  $k/2$ -krát.  $i$ -tý bit od konce se změní každých  $2^i$  operací, tedy v průměru  $k/2^i$ -krát. Celkový průměrný počet změn bitů je tedy

$$\sum_{i=0}^n \frac{k}{2^i} \leq k \sum_{i=0}^{\infty} \frac{1}{2^i} = 2k$$

Abychom získali *horní* odhad, stačí vzít vždy horní celou část, tj.  $i$ -tý bit od konce se změní *nejvýše*  $\lceil k/2^i \rceil$ -krát. Celkový počet změn je tedy *nejvýše*

$$\sum_{i=0}^n \left\lceil \frac{k}{2^i} \right\rceil \leq \sum_{i=0}^k \left(1 + \frac{k}{2^i}\right) \leq n + 2k$$

2. **Účetní metoda:** Změna jednoho bitu stojí jeden žeton a na každou operaci dostaneme dva žetony. U každého jedničkového bitu v původní bitové sekvenci si „uschováme“ jeden žeton. Při inkrementu máme vynulování jedničkových bitů předplaceno. Oba žetony využijeme na jedinou změnu nulového bitu na jedničku a předplacení vynulování tohoto bitu.

V každém okamžiku tedy máme na všech jedničkách žetony, které se použijí na jejich případné vynulování. Každá operace stojí 2 žetony, tj.  $2k$  žetonů celkem. Nejvíce  $n$  žetonů jsme utratili na iniciální předplacení jedniček, dohromady tedy spotřebujeme nejvíce  $n + 2k$  žetonů.

3. **Potenciálová metoda:** Vhodnou potenciálovou funkcí je kupříkladu  $\Phi = \text{počet jedničkových bitů}$ . Označíme  $\Phi$  potenciál před operací a  $\Phi'$  potenciál po provedení operace. Zajímá nás

$$\text{amortizovaný čas} = \text{skutečný čas} + (\Phi' - \Phi)$$

Nechť  $j$  je počet jedniček vynulovaných při dané operaci. Pak skutečný čas operace je  $j + 1$  a změna potenciálů  $(\Phi' - \Phi) = \Delta\Phi = (1 - j)$ . Amortizovaný čas je tedy

$$\text{amortizovaný čas} = j + 1 + 1 - j = 2$$

Celkový čas posloupnosti  $k$  operací pak lze spočítat jako

$$T = \sum_{i=1}^k \text{skutečný čas } i\text{-té operace} = k \cdot \text{amortizovaný čas} + (\Phi_0 - \Phi_k) \leq 2k + n$$

Poslední nerovnost plyne z  $0 \leq \Phi \leq n$ . Takovéto sečtení sumy  $\sum_{i=1}^k (\Phi_{i-1} - \Phi_i) = \Phi_0 - \Phi_k$  se nazývá *teleskopické vykrácení*.

- **Dynamické pole**

Máme pole, do kterého přidáváme prvky i je z něj mažeme. Implementace používá statická pole pevné velikosti  $p$ . Je-li v poli již  $n = p$  prvků (tj. je plné) a my chceme přidat prvek, vytvoříme nové pole velikosti  $2p$ , do něj zkopiujeme obsah původního pole a přidám prvek. Je-li  $p = 4n$  a my chceme přidat prvek, zmenšíme pole na polovinu.

1. **Agregovaná analýza:** (neformálně) Zkopírování pole trvá  $O(n)$ . Po zkopirování je  $n = p/2$  a k dalšímu kopírování dojde nejdříve po  $n/2$  operacích Insert nebo Delete. Předpokládáme-li, že operace Insert a Delete jsou konstantní (pokud nedošlo ke kopírování), pak je amortizovaná složitost  $O(1)$ .

(formálně) Nechť  $k$  je celkový počet operací a  $k_i$  je počet operací mezi  $(i-1)$  a  $i$ -tou realokací (platí  $\sum_i k_i = k$ ). Při první realokaci se kopíruje nejvýše  $n_0 + k_1$  prvků, kde  $n_0$  je počáteční počet prvků. Při každé další realokaci se kopíruje nejvýše  $2k_i$  prvků (plyne z toho, že pole se zdvojnásobuje, tj. nezvětšuje se o konstantní hodnotu). Celkem se tedy zkopiuje nejvýše

$$n_0 + k_1 + \sum_{i \geq 2} 2k_i \leq n_0 + 2k$$

prvků. Předpokládáme-li  $O(1)$  na zkopirování jednoho prvku, pak dostáváme požadovanou amortizovanou složitost.

2. **Potenciálová metoda:** Definujeme následující potenciál

$$\Phi = \begin{cases} 0 & \text{pokud } p = 2n \\ n & \text{pokud } p = n \\ n & \text{pokud } p = 4n \end{cases}$$

a zbylé případy lineární interpolací. Explicitně

$$\Phi = \begin{cases} 2n - p & \text{pokud } p \leq 2n \\ \frac{p}{2} - n & \text{pokud } p > 2n \end{cases}$$

Minimální potenciál má tedy pole zaplněné právě z poloviny.

Prozkoumáme 3 případy:

- **Přidáváme či odebíráme prvek a nedochází k realokaci.** Skutečná cena operace je 1, rozdíl potenciálů je  $\leq 2$  (důkaz rozborem případů). Amortizovaná cena je tedy  $\leq 3$
- **Přidáváme prvek a dochází k realokaci.** To nutně znamená, že  $\Phi = n$  a  $\Phi' = 2$ . Skutečná cena je  $n + 1$  (realokace  $n$  prvků + vložení 1 prvku). Amortizovaná cena je tedy

$$A = (n + 1) + (2 - n) = 3$$

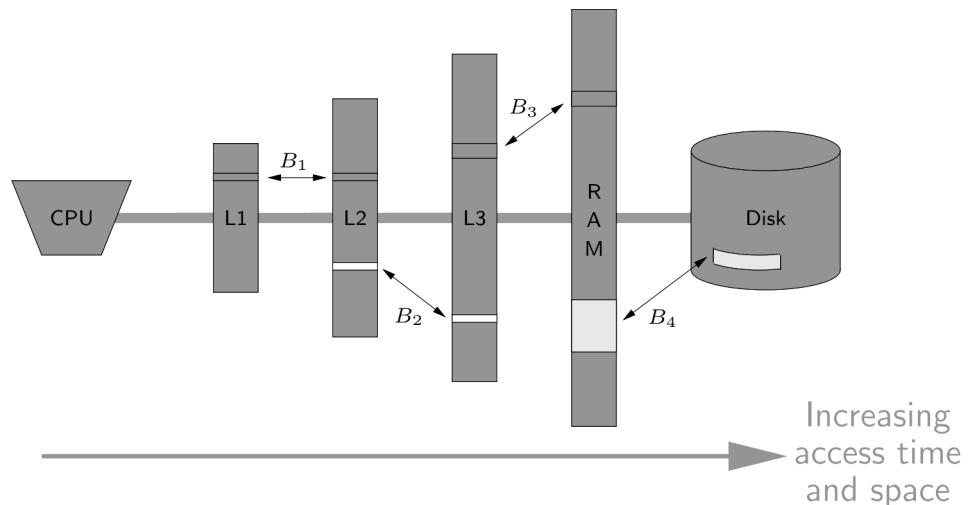
- **Odebíráme prvek a dochází k realokaci.** To nutně znamená, že  $\Phi = n$  a  $\Phi' = 1$ . Skutečná cena je  $n + 1$  (realokace  $n$  prvků + odebrání 1 prvku). Amortizovaná cena je tedy

$$A = (n + 1) + (1 - n) = 2$$

## 2.5 Chování a analýza datových struktur na systémech s paměťovou hierarchií.

### 2.5.1 Paměťová hierarchie

Paměť je v moderních počítačích hierarchická, od registrů v procesoru přes cache, RAM až po pevný disk. S rostoucí velikostí však roste i doba odezvy.



Obrázek 2.2: Paměťová hierarchie

	size	speed
L1 cache	32 KB	223 GB/s
L2 cache	256 KB	96 GB/s
L3 cache	8 MB	62 GB/s
RAM	32 GB	23 GB/s
SDD	112 GB	448 MB/s
HDD	2 TB	112 MB/s
Internet	$\infty$	10 MB/s

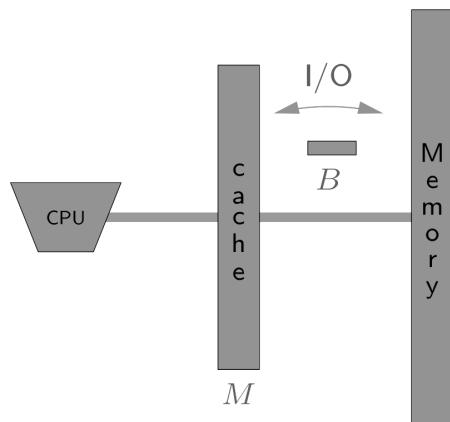
Tabulka 2.1: Příklad velikostí a rychlostí různých typů pamětí.

#### 2.5.1.1 Zjednodušený model paměti

Pro analýzu chování algoritmů a datových struktur v systémech s paměťovou hierarchií budeme uvažovat zjednodušený model paměti. Budeme mít pouze 2 úrovně: *pomalý disk* a *rychlou cache*.

Paměť je rozdělená na bloky velikosti  $B$  (pro jednoduchost budeme předpokládat, že jeden prvek (integer, ...) zabírá přesně jednotkový prostor, takže do jednoho bloku se vejde  $B$  prvků). Velikost cache označíme  $M$ . Cache tedy obsahuje  $P = M/B$  bloků.

Procesor může přistupovat pouze k datům v cache. Paměť je plně asociativní, přičemž předpokládáme, že každý blok z disku může být uložený na libovolné pozici v cache. Data se mezi diskem a cache přesouvají po celých blocích. Naším



cílem bude určit **celkový počet přenesených bloků**.

### 2.5.2 Cache-aware a cache-oblivious algoritmy

*Pozn.: Otázka se ptá na datové struktury na systémech s paměťovou hierarchií, nikoliv algoritmy. Ke strukturám se taky dostaneme, ale v DS bylo tohle téma převážně o algoritmech. Proto je sem dáme taky.*

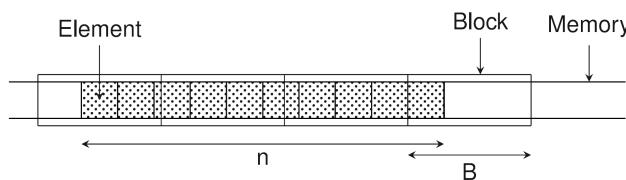
**Cache-aware** algoritmus je takový algoritmus, který **zná hodnoty  $M$  a  $B$**  a podle nich nastavuje své parametry (např. velikost vrcholu B-stromu).

**Cache-oblivious** algoritmus je takový algoritmus, který efektivně funguje **bez znalosti  $B$  a  $M$** .

Budeme zkoumat různé algoritmy prostřednictvím **cache-oblivious analýzy** – tj. hodnoty  $M$  a  $B$  považujeme za neznáme a zkoumáme chování algoritmu, konkrétně počet přenesených bloků.

- **přečtení souvislého pole:** Máme pole zapsané v paměti v kuse, chceme přečíst úsek délky  $n$ . Počet přenesených bloků je nejvýše

$$\left\lceil \frac{n}{B} \right\rceil + 1$$

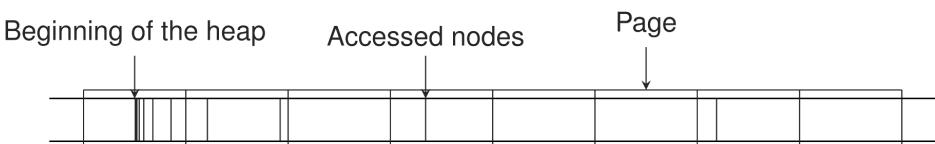


- **binární vyhledávání v poli:** Celkem provedeme  $\Theta(\log n)$  porovnání s hledaným prvkem (pro jednoduchost předpokládáme neúspěšné vyhledávání). Posledních  $\Theta(\log B)$  prvků je uloženo nejvýše ve 2 blocích. Ostatní prvky jsou uloženy v po dvou různých blocích. Celkem je tedy přenesených bloků

$$\Theta(\log n - \log B) = \Theta(\log \frac{n}{B})$$

- **průchod binární haldou uloženou v poli:** Analogicky jako předchozí případ, opět vyjde

$$\Theta(\log n - \log B) = \Theta(\log \frac{n}{B})$$



- **transpozice matice  $n \times n$ :** Předpokládejme nejhorší možný případ, tj.  $\frac{M}{B} < n$ . Pokud je matice uložena po řádcích a do jednoho bloku se vejde nejvýše jeden řádek, tak to znamená, že se do cache nevejde celý sloupec matice.

1. **naivní algoritmus:** Naivní implementace prochází matici po řádcích.

Rekněme, že se řádek matice  $A$  vejde do jednoho bloku. První řádek matice  $A$  odpovídá prvnímu sloupci matice  $A^T$ , přičemž každý prvek tohoto sloupce je v jiném bloku paměti. Pro transpozici prvního řádku tedy potřebujeme načít do cache  $1 + n$  bloků. Z předpokladu  $\frac{M}{B} < n$  se však také bloků do cache nevejde, a tedy nutně musí nějaký blok z cache vypadnout. Pokud cache postupuje podle strategie LRU („least recently used“), pak tento vypadnutý blok odpovídá prvnímu

---

**Algorithm 2**

---

```

1: for  $i = 1 \dots n$  do
2:   for  $j = 1 \dots n$  do
3:     swap( $A[i, j], A[j, i]$ )

```

---

řádku  $A^T$ . To znamená, že jakmile se dostaneme k druhému řádku  $A$  a chceme umístit jeho první element  $A_{2,1}$  do  $A_{1,2}^T$ , je již první řádek  $A^T$  mimo cache a musí se znova načíst. Toto se opakuje pro každý další element.

Pokud by matice byla uložena po sloupcích (nebo pokud by transpozice postupovala po sloupcích), je situace podobná, avšak výpadky nenastávají při zápisu do  $A^T$ , nýbrž při čtení z  $A$ .

Celkový počet přístupů do paměti je  $\Omega(n^2)$ .

2. **cache-aware algoritmus** Funguje za předpokladu, že  $M > B^2$  (tzv. „tall cache“). Matici rozdělíme na podmatice  $B \times B$  a použijeme následující algoritmus:

---

**Algorithm 3**

---

```

1: for  $i = 1; i < n; i += B$  do
2:   for  $j = i; j < n; j += B$  do
3:     for  $ii = i; ii < \min(k, i + B); ii + +$  do
4:       for  $jj = \max(j, ii + 1); jj < \min(k, j + B); jj + +$  do
5:         swap( $A[ii, jj], A[jj, ii]$ )

```

---

Postupujeme tedy nikoliv po řádcích, ale po podmaticích  $B \times B$ . To nám zaručuje, že v rámci této podmatice nedochází k výpadkům paměti, jako v naivním algoritmu. První řádek této podmatice je zapsán nejvýše ve 2 blocích, zapisujeme do  $B$  bloků  $A^T$ , tedy  $B+2$  bloků, což se do cache vejde. Přejdeme na druhý řádek, potřebujeme zapisovat do těch samých  $B$  bloků, ty jsou stále v cache – žádný výpadek. První řádek může již v klidu z cache vypadnout, bude-li třeba. I sloupce cílové podmatice mohou být uloženy až ve 2 blocích, celkem tedy načteme  $4B$  bloků (bez výpadků). Podmatic je  $(n/B)^2$ , celkem tedy

$$4B \cdot \left(\frac{n}{B}\right)^2 = 4\frac{n^2}{B} = O\left(\frac{n^2}{B}\right)$$

Algoritmus pracuje s hodnotou  $B$ , je tedy cache-aware.

3. **cache-oblivious algoritmus** Rekurzivní algoritmus. Matici  $A$  rozdělíme na submatice

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \quad A^T = \begin{pmatrix} A_{11}^T & A_{21}^T \\ A_{12}^T & A_{22}^T \end{pmatrix}$$

Rekurzivně transponujeme menší a menší podmatice, až se v nějakém  $i$ -tém kroku dostaneme na velikost  $n_i$ , která se již vejde do paměti, tj.  $n_i \leq B \leq 2n_i$ . Transponování v rámci této matice má stejné vlastnosti, jako v případě  $B \times B$  matice u cache-aware algoritmu – tj. nedochází k výpadkům a celkem je třeba načíst  $4n_i \leq 4B$  bloků. Celkem máme  $n/n_i$  těchto sumbatic, takže počet přenesených bloků je nejvýše

$$\left(\frac{n}{n_i}\right)^2 \cdot 4n_i \leq \frac{8n^2}{n_i} = O\left(\frac{n^2}{B}\right)$$

To je stejný výsledek, jako v předchozím případě. Tento algoritmus však již nezávisí na  $B$  a je tedy cache-oblivious.

- **násobení matic**
- **merge-sort**
- **funnel-sort**

---

**Algorithm 4**


---

```

1: function TRANSPOSEONDIAGONAL( $A$ )
2:   if matice je „malá“ then
3:     Transponuj  $A$  triviálně
4:   else
5:      $A_{11}, A_{12}, A_{21}, A_{22} \leftarrow$  souřadnice podmatic
6:     transposeOnDiagonal( $A_{11}$ )
7:     transposeOnDiagonal( $A_{22}$ )
8:     transposeAndSwap( $A_{12}, A_{21}$ )
9: function TRANSPOSEANDSWAP( $A, B$ )
10:  if matice je „malá“ then
11:    Prohod'  $A$  a  $B$  a transponuj triviálně
12:  else
13:     $A_{11}, A_{12}, A_{21}, A_{22}, B_{11}, B_{12}, B_{21}, B_{22} \leftarrow$  souřadnice pod-
        matic
14:    transposeAndSwap( $A_{11}, B_{11}$ )
15:    transposeAndSwap( $A_{12}, B_{21}$ )
16:    transposeAndSwap( $A_{21}, B_{12}$ )
17:    transposeAndSwap( $A_{22}, B_{22}$ )

```

---

### 2.5.3 Cache-aware a cache-oblivious datové struktury

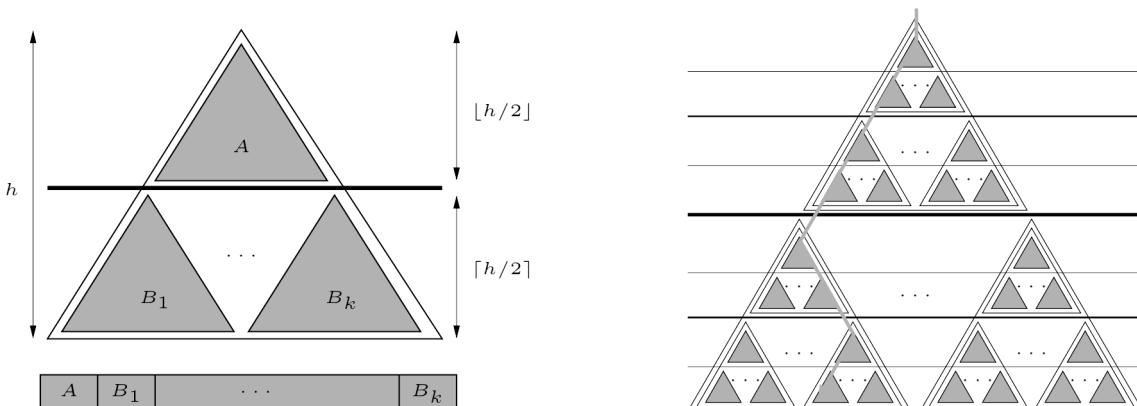
Budeme chtít sestrojit efektivní reprezentaci **binárního stromu**.

- **Klasický binární strom (pomocí ukazatelů)**: co uzel, to jiný blok  $\Rightarrow \log(n)$  přístupů do paměti.
- **Binární halda**:  $\Theta(\log(n) - \log(B))$
- **B-strom**: uzly velikosti  $B$ , výška stromu je  $\Theta(\log_B(n))$ , při průchodu k listu se načte  $\Theta(\log_B(n))$  bloků. Nicméně je to cache-aware struktura (a navíc jsme chtěli binární strom).

**van Emde Boas rozložení**: Rozsekneme strom v polovině výšky (tj.  $h/2 = \log_2(n)/2$ ), vzniklé podstromy mají velikost  $\sqrt{n}$ . Do paměti uložíme v pořadí shora a zleva (viz obrázek). Rekurzivně rozdělujeme dále až do jisté úrovně, kdy podstrom výšky  $z$  již vejde do cache. Platí  $z \leq \log_2 B \leq 2z$ . Nyní počet podstromů výšky  $z$  na cestě z kořene do listu je

$$\frac{h}{z} \leq \frac{2 \log_2 n}{\log_2 B} = 2 \log_B(n)$$

Počet načtených bloků je tedy  $\Theta(\log_B n)$ .



### 2.5.4 Strategie pro správu cache

**OPT:** Optimální off-line algoritmus předpokladající znalost všech přístupů do paměti

**FIFO:** Z cache smažeme stránku, která je ze všech stránek v cache nejdelší dobu

**LRU:** Z cache smažeme stránku, která je ze všech stránek v cache nejdéle nepoužitá

Srovnání LRU a OPT:

**Věta 2.5.1** (Sleator, Tarjan). *Nechť  $s_1, s_2, \dots, s_k$  je posloupnost přístupů do paměti, nechť LRU má k dispozici  $n_{LRU}$  bloků v cache, OPT má k dispozici  $n_{OPT}$  bloků v cache a  $n_{LRU} > n_{OPT}$ . Označme  $F_{LRU}$  a  $F_{OPT}$  počet přenesených bloků jednotlivých strategií (= počet výpadků cache). Pak*

$$F_{LRU} \leq \frac{n_{LRU}}{n_{LRU} - n_{OPT}} \cdot F_{OPT} + n_{OPT}$$

**Důsledek 2.5.1.** *Pokud LRU může uložit dvojnásobný počet bloků než OPT, pak má LRU nejvýše dvojnásobný počet přenesených bloků oproti OPT (plus  $n_{OPT}$ ).*

Zdvojnásobení velikosti cache většinou nemá vliv na asymptotický počet přenesených bloků, viz např.:

- průchod polem:  $O(\frac{n}{B})$
- Mergesort:  $O(\frac{n}{B} \log(\frac{n}{M}))$
- Funnelsort:  $O(\frac{n}{B} \log \frac{M}{B} (\frac{n}{B}))$
- van Emde Boas:  $O(\log_B n)$

*Důkaz.* Rozsekáme  $s_1, s_2, \dots, s_k$  na kusy, kde v každém kusu až na poslední přenese LRU přesně  $n_{LRU}$  bloků (tj. dojde k  $n_{LRU}$  výpadkům). To znamená, že v každém kusu se přistupuje aspoň k  $n_{LRU}$  různým blokům (plyne z definice LRU). OPT má v daném kusu nejméně  $(n_{LRU} - n_{OPT})$  výpadků, neboť i při sebelepším managementu má k dispozici pouze  $n_{OPT}$  bloků v cache, tedy k výpadkům nutně musí dojít.

Označíme  $F'_{LRU}$  resp.  $F'_{OPT}$  počet výpadků pro LRU resp. OPT v daném kusu. Platí  $F'_{LRU} = n_{LRU}$  a  $F'_{OPT} \geq (n_{LRU} - n_{OPT})$  a tedy

$$\begin{aligned} \frac{F'_{LRU}}{F'_{OPT}} &\leq \frac{n_{LRU}}{n_{LRU} - n_{OPT}} \\ F'_{LRU} &\leq \frac{n_{LRU}}{n_{LRU} - n_{OPT}} F'_{OPT} \end{aligned}$$

V posledním kusu dat tato nerovnost neplatí, neboť počet výpadků LRU může být menší než  $n_{LRU}$ . Označme  $F''_{LRU}$  resp.  $F''_{OPT}$  počet výpadků pro LRU resp. OPT v posledním kusu. Stejnou logikou jako v obecném případě můžeme odvodit, že  $F''_{OPT} \geq F''_{LRU} - n_{OPT}$ , a tedy jednoduchou úpravou

$$F''_{LRU} \leq F''_{OPT} + n_{OPT}$$

Díky  $n_{LRU} > n_{OPT}$  také platí  $\frac{n_{LRU}}{n_{LRU} - n_{OPT}} > 1$  a tedy

$$F''_{LRU} \leq \frac{n_{LRU}}{n_{LRU} - n_{OPT}} F''_{OPT} + n_{OPT}$$

□

# Část II

## Inteligentní agenti

# Kapitola 3

## Přírodou inspirované počítání

### 3.1 Genetické algoritmy, genetické a evoluční programování.

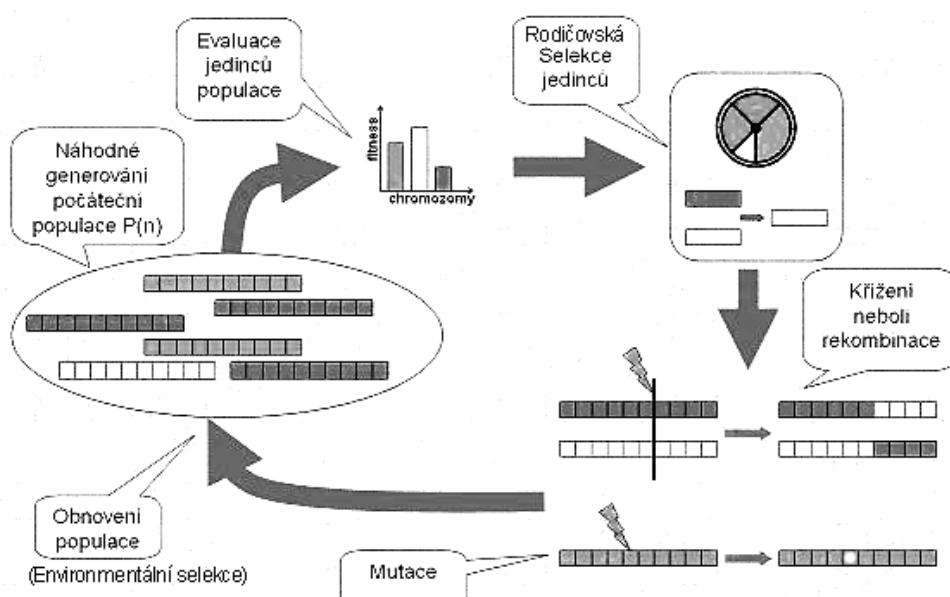
#### 3.1.1 Evoluční algoritmy

**Evoluční algoritmy (EA)** je souhrnné označení pro všechny výpočetní metody inspirované evoulcí. Historicky se nejdříve paralelně vyvíjely 3 metodiky: genetické algoritmy, evoluční strategie a evoluční programování. Až později došlo k jejich seskupení pod společný zastřešující název: evoluční algoritmy (někdy také evoluční počítání).

EA jsou populační stochastické prohledávací algoritmy. Obecně se hodí spíše označení meta-algoritmus, jelikož pro specifické problémy se vytváří doménově specifické varianty EA s doménově danou reprezentací a vlastními implementacemi evolučních operátorů. Obecně platí „no free lunch theorem“ – tj. neexistuje jeden ultimátní nejlepší algoritmus pro všechny typy problémů.

Základní struktura obecného EA je následující:

1. vytvoř náhodnou iniciální populaci  $P(t = 0)$ , ohodnoť  $P(0)$
2. dokud není splněna ukončovací podmínka:
  - (a) rodičovská selekce podle fitness
  - (b) křížení s pravděpodobností  $p_c$
  - (c) mutace s pravděpodobností  $p_m$
  - (d) ohodnocení nové populace, environmentální selekce do  $P(t + 1)$



### 3.1.2 Genetické algoritmy (GA)

Původní Hollandův GA (dnes označován jako Simple GA, SGA) je nejstarší a nejjednodušší varianta EA, jejímž klíčovým rysem je to, že jedinci jsou **binární řetězce**. Používá ruletovou selekci (rodičovskou, environmentální se nepoužívá), 1-bodové křížení, bitové mutace (viz níže). V původním Hollandově návrhu je i operátor *inverze*, který obrátí část řetězce, ovšem se zachováním významu bitů. Inverze se ukázala být nevýhodnou.

### 3.1.3 Genetické programování (GP)

Tento pojem označuje genetické algoritmy, v nichž je jedincem v populaci počítačový **program**. Myšlenkou tedy je, že budeme evolvovat počítačový kód, který za nás bude řešit problémy. Duchovním otcem GP je John Koza. V jeho původní práci jsou programy reprezentovány jako syntaktické stromy, kde listy jsou proměnné a konstanty, vnitřní uzly jsou operace. **Křížení** je výměna podstromu, **mutací** se používá několik typů: mutace konstant, výměna uzlu stejné arity, permutace podstromů, záměna neterminálu za terminál apod. **Fitness** je poměrně jednoduchá: spustit vygenerovaný program na testovacích datech a ohodnotit, nakolik řeší daný problém.

Vylepšení základního principu je použití **automaticky definovaných funkcí (ADF)**. To jsou samostatné stromy, které mají omezenou množinu terminálů i neterminálů a jsou charakterizovány svou aritou. Volání nějaké ADF je pak novým terminálem, který se může vyskytnout v listech. Celý program pak sestává z hlavního stromu a několika vedlejších stromů ADF. Genetické operace GP pracují buď jen v rámci hlavního programu nebo v rámci ADF.

GP se často potýká s problémem bobtnání (**bloat**), tj. programy mají tendenci narůstat. V praxi se to řeší restrikcí velikosti či hloubky stromu, a to buď jako penalizující člen ve fitness a nebo jako kontrola (a případná eliminace) nově vznikajících jedinců. Zavádí se taky *anti-bloat* operátory, tedy křížení a mutace, které jedince nezvětšují, případně jej rovnou zmenšují.

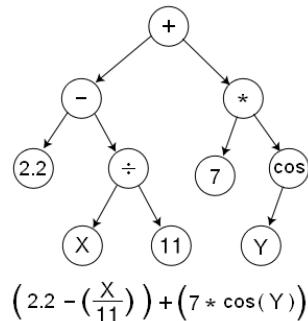
Kromě stromů lze použít i jiné struktury. „Cestou dolů“ jsou **lineární GP**, tj. program je lineární sekvence příkazů. To vede k jednodušším operátorům i celkové reprezentaci, rovněž rychlejší emulaci běhu; nicméně je tu velké riziko toho, že vznikne nesmyslný program. „Cestou nahoru“ jsou **grafové GP**, kde program je (často acyklický) graf. Vznikly nejprve jako rozšíření na paralelní programy, později získaly obecnější tvar a dnes se používají např. pro evoluci obvodů či neuronových sítí. Kvůli větší komplexitě reprezentace jsou komplikovanější i genetické operátory.

### 3.1.4 Evoluční programování (EP)

Evoluční programování je jeden z nejstarších směrů vývoje evolučních algoritmů, dokonce ještě o něco starší než Hollandovy GA (60. léta). Průkopníkem je Lawrence Fogel. Zásadní myšlenkou je vyvinout „umělou inteligence“, konkrétně agenta, který bude umět předpovídat stav prostředí – jinými slovy bude umět odvodit patřičnou akci v reakci na prostředí, v němž se nachází. Agent je zde reprezentován **konečným automatem**, prostředí je pak **sekvence symbolů konečné abecedy**.

Samotná evoluce pak vypadá takto: jedinci v populaci jsou konečné automaty, těm jsou předkládány vstupy. Jejich výstup je vždy porovnán s následujícím vstupem v posloupnosti. Fitness je úspěšnost predikce, přičemž se může měřit různě: vše nebo nic, absolutní chyba, mean square error. Často je do fitness započítána i velikost automatu.

*Každý* automat podstoupí mutaci, která může mít různou „intenzitu“ – od malých, „kosmetických“ změn po radikální. Příklady mutací jsou: změna výstupního symbolu, změna následného stavu, přidání stavu, odebrání stavu, změna počátečního stavu, ... Noví jedinci jsou ohodnoceni a poté je vybráno (z nových i starých najednou)  $N$  jedinců do nové populace. Výběr probíhá buď turnajem, nebo deterministicky (tj. vezme se  $N$  nejlepších jedinců. (Pozn.: V původním algo-



ritmu není nijak omezeno, kolik potomků může být vygenerováno z jednoho rodiče, ani že velikost populace musí zůstávat konstantní.)

#### Křížení neexistuje (!).

V dnešní době se EP vyvinulo do obecnější podoby. Zakódování jedinců je velmi benevolentní a mělo by být přirozené pro daný problém (často genotyp = fenotyp). Používá se více „chytrých mutací“, které jsou těsně přizpůsobeny problému a dané reprezentaci. Křížení se typicky nepoužívá. Rodičovská selekce je prostá: každý jedinec se vybere právě jednou. Environmentální selekce je turnajová na spojené populaci rodičů a potomků.

Jelikož se EP vyvíjelo paralelně s GA i ES, jsou mezi nimi nevyhnutelně mnohé podobnosti. Pojdeme si tyto metody porovnat.

##### 3.1.4.1 EP vs. GA

V GA jsou jedinci sekvence symbolů abecedy, původně pouze binární, dnes už vcelku libovolné. V EP není žádné omezení na podobu reprezentace a typicky je přizpůsobena problému (jedinec může být např. neuronová síť).

Mutace funguje rovněž trochu jinak. V EP máme škálu mutací od malých změn až po radikální zásahy, přičemž pravděpodobnost aplikace mutace je nepřímo úměrná její závažnosti. Celková závažnost mutací se rovněž může snižovat v průběhu konvergence k optimálnímu řešení.

Zjevným rozdílem je rovněž absence křížení u EP.

##### 3.1.4.2 EP vs. ES

Pokud je jedincem v EP posloupnost reálných čísel, začínají být velice podobné evolučním strategiím. EP dokonce také mohou být *self-adaptive*, tj. jedinec nese kromě genotypu i parametry ovlivňující evoluci, konkrétně míru závažnosti mutací (rozptyl gaussovské mutace).

Existují však rozdíly: EP používá turnajovou selekci, ES deterministicky odstraňuje dané množství nejhorších řešení. Druhým rozdílem je absence křížení u EP, zatímco ES je často implementuje.

##### 3.1.4.3 Význam křížení

Malá odbočka: trpí EP tím, že nemá křížení? Pomáhá vlastně křížení, nebo stačí mutace? Jones (1995) provedl tzv. „headless chicken experiment“: porovnával klasické GA a GA, kde je klasické křížení nahrazeno „náhodným křížením“. To funguje tak, že se dva vybraní jedinci nezkříží mezi sebou, namísto toho se pro každého z nich vygeneruje partner jako náhodná sekvence z domény (tj. například náhodný binární vektor správné délky). Pak se provede jednobodové křížení a od každého rodiče se vybere jeden potomek. Jones označil tento operátor jako *headless chicken* – stejně jako bezhlavé kuře pobíhající po dvorku není doopravdy kuře (protože mu něco důležitého chybí), tak ani toto „náhodné křížení“ není vlastně křížení, protože při něm vůbec nedochází ke kontaktu mezi jedinci. Jedná se ve skutečnosti o jakousi makromutaci.

Pointou experimentu bylo porovnat výkonnost obou variant a určit, nakolik opravdové křížení skutečně pomáhá. Závěrem bylo konstatování, že pokud má problém jasně definované stavební bloky, pak křížení skutečně pomáhá (především na začátku evoluce), zatímco pokud nejsou bloky zřejmé, pak křížení příliš nepomáhá (výkon obou variant byl srovnatelný).

## 3.2 Teorie schémat, pravděpodobnostní modely jednoduchého genetického algoritmu.

### 3.2.1 Schémata

**Schéma** v kontextu genetických algoritmů (tj. jedinec je slovo v abecedě {0,1}) je reprezentace množiny jedinců jako slova v abecedě {0,1,\*}, kde \* označuje libovolný symbol („žolík“). Například schéma 0\*11\* reprezentuje množinu jedinců {00110, 00111, 01110, 01111}. Platí následující:

- schéma s  $r$  \* reprezentuje  $2^r$  jedinců

- jedinec délky  $m$  je reprezentován  $2^m$  schématy
- je  $3^m$  schémat délky  $m$
- v populaci velikosti  $n$  je  $2^m$  až  $n \cdot 2^m$  schémat

Pro schéma  $S$  lze definovat následující vlastnosti:

**Řád schématu  $o(S)$ :** počet pevných pozic (0 a 1)

**Definující délka  $d(S)$ :** vzdálenost mezi první a poslední pevnou pozicí

**Fitness  $F(S)$ :** průměrná fitness odpovídajících jedinců v populaci

**Věta o schématech (VoS):** Krátká schémata s nadprůměrnou fitness a malým řádem se v populaci během GA exponenciálně množí.

*Důkaz.* Budeme zkoumat, co se děje s konkrétním schématem  $S$  během selekce, křížení a mutace.

Nechť  $P(t)$  je populace v čase  $t$ ,  $n$  je velikost populace,  $m$  je délka jedinců v populaci,  $C(S, t)$  je četnost schématu  $S$  v populaci  $P(t)$  (tj. počet jedinců v  $P(t)$  reprezentovaných  $S$ ). Budeme odhadovat  $P(S, t + 1)$ .

*Selekce:* Schéma  $S$  má pravděpodobnost vybrání

$$p_s(S) = \frac{F(S)}{\sum_{u \in P(t)} F(u)}$$

Tedy

$$C(S, t + 1) = C(S, t) \cdot n \cdot p_s(S)$$

neboť provádíme  $n$  výběrů do nové populace pomocí ruletové selekce, přičemž na ruletě zabírají jedinci odpovídající schématu  $S$  právě  $C(S, t) \cdot p_s(S)$  procent místa. Ekvivalentně lze psát

$$C(S, t + 1) = C(S, t) \cdot \frac{F(S)}{\frac{\sum_{u \in P(t)} F(u)}{n}} = C(S, t) \cdot \frac{F(S)}{F_{avg}(t)}$$

kde  $F_{avg}(t)$  označuje průměrnou fitness v populaci  $P(t)$ .

Představme si, že schéma  $S$  je nadprůměrné o  $\varepsilon\%$ , tj.  $F(S) = F_{avg}(t) + \varepsilon \cdot F_{avg}(t)$ . Pak

$$C(S, t + 1) = C(S, t)(1 + \varepsilon)$$

$$C(S, t + 1) = C(S, 0)(1 + \varepsilon)^t$$

Tedy četnost nadprůměrných schémat roste geometrickou řadou.

*Křížení:* Předpokládáme klasické jednobodové křížení. Pravděpodobnost, že schéma  $S$  křížení nepřežije, je

$$p_{death}(S) = \frac{d(S)}{m - 1}$$

a tedy pravděpodobnost přežití je

$$p_{surv}(S) \geq 1 - p_c \cdot \frac{d(S)}{m - 1}$$

kde  $p_c$  je pravděpodobnost křížení. Všimněte si nerovnosti: je totiž šance, že i když se bod křížení „treffí“ dovnitř schématu, tak část doplněná s druhého rodiče bude pořád odpovídat schématu a celkově tedy schéma přežije. Zjevné je to například při křížení dvou totožných jedinců (tam schéma přežije vždy).

*Mutace:* Pravděpodobnost změny jednoho bitu je  $p_m$ , pravděpodobnost přežití schématu je tedy

$$p_{surv}(S) = (1 - p_m)^{o(S)}$$

$$p_{surv}(S) \approx 1 - p_m \cdot o(S)$$

Poslední approximace funguje pro  $p_m \ll 1$ .

Dohromady tedy dostáváme:

$$C(S, t + 1) \geq C(S, t) \cdot \frac{F(S)}{F_{avg}(t)} \cdot \left(1 - p_c \cdot \frac{d(S)}{m - 1} - p_m \cdot o(S)\right)$$

□

**Hypotéza o stavebních blocích (HoSB)** GA hledá suboptimální řešení problému rekombinací krátkých, nadprůměrných s malým řádem schémat („building blocks“).

### 3.2.1.1 Důsledky VoS

GA pracuje s  $n$  jedinci, ale implicitně vyvíjí mnohem více schémat:  $2^m$  až  $n 2^m$  (implicitní paralelismus). Holland tvrdí, že za splnění jistých předpokladů ( $n = 2^m$ , schémata zůstávají nadprůměrná, ...) platí, že počet schémat, kterým se v GA dostává exponenciálního růstu je úměrný  $n^3$ .

GA také optimálně řeší problém explorace a exploatace, což lze ukázat na analogii s tzv. **2-rukým banditou**: *Mám  $N$  mincí, ruce bandity (= hrací automat) vyplácí se střední hodnotou  $m_1$  resp.  $m_2$  a rozptylem  $s_1$  resp.  $s_2$ , cílem je maximalizovat zisk. Analytickým řešením je alokovat exponenciálně více pokusů pro právě vyhrávající ruku.* GA rovněž alokuje exponenciálně mnoho nadějným schématům. Nejdříve se myslelo, že GA hraje  $3^m$ -rukého banditu, tj. všechna schémata jsou konkurenční ruce. Ve skutečnosti se hraje více her, v nichž schémata soutěží o konfliktní pevné pozice: schémata řádu  $k$  soutěží právě o těch  $k$  pozicích – hrají spolu  $2^k$ -rukého banditu. To stále není úplně přesné, protože na rozdíl od bandity, kde jsou ruce nezávislé, nesampluje GA schémata nezávisle.

Problémem také je, že GA často neodhadne správně skutečnou fitness schémat. Např. je-li  $F(111\ldots*) = 2$ ,  $F(0*\ldots*) = 1$  a jinak  $F(x) = 0$ , pak platí  $F(1*\ldots*) = \frac{1}{2}$ ,  $F(0*\ldots*) = 1$ . Jenže GA odhadne  $F(1*\ldots*) \approx 2$ , protože  $111\ldots*$  v populaci převáží. Tomuto se říká *kolaterální konverence*, tj. jakmile se někam začne konvergovat, už nejsou schémata samplována rovnoměrně a fitness se neodhadne správně. Podobným problémem je velký rozptyl fitness (např.  $1*\ldots*$  z předchozího příkladu). GA bude nejspíše konvergovat tam, kde je fitness větší → chybný odhad statické fitness.

### 3.2.2 Pravděpodobnostní modely

Od 90. let se objevují snahy přesně modelovat chování GA, konkrétně: jak přesně vypadají populace, zobrazení přechodu k další populaci, vlastnosti tohoto zobrazení, asymptotické chování jednoduchého GA. Existují modely pro konečné i nekonečné populace.

#### 3.2.2.1 Ještě jednodušší jednoduchý GA (JJGGA)

Pro jednodušší tvorbu modelů byl původní jednoduchý GA ještě o něco zjednodušen a formalizován takto:

- inicializuj náhodnou počáteční populaci binárních řetězců  $x$  délky  $l$
- dokud nenajdeš dost dobré  $x$ :
  - dokud nenaplníš novou populaci:
    - \* vyber selekcí 2 jedince, zkřiž je s pravděpodobností  $p_c$ , jednoho potomka zahod'
    - \* mutuj každý bit nového řetězce s pravděpodobností  $p_m$
    - \* vlož do nové populace

#### 3.2.2.2 Analýza

Každý jedinec je tedy binární řetězec délky  $l$ , což lze chápat jako binární zápis nějakého čísla  $[0, \dots, 2^l - 1]$  (např. 00000111 je 7). Populace v čase  $t$  je reprezentována dvěma vektory  $p(t)$  a  $s(t)$ , oba délky  $2^l$ . Číslo  $p_i(t)$  vyjadřuje, kolik procent populace zabírá řetězec  $i$ ; číslo  $s_i(t)$  vyjadřuje pravděpodobnost selekce řetězce  $i$ .

Dále definujeme matici  $F$  takto:  $F(i, i) = f(i)$ ;  $F(i, j) = 0$  pro  $i \neq j$ . Hodnota  $f(i)$  je fitness jedince  $i$ . Pak můžeme definovat vztah vektorů  $s(t)$  a  $p(t)$  jako

$$s(t) = \frac{F \cdot p(t)}{\sum_{j=0}^{2^l-1} F(j, j) \cdot p_j(t)}$$

Naším ideálem je definovat matici  $G$ , která realizuje jeden krok JJGA, tj.  $p(t+1) = G \cdot p(t)$ , nebo obdobně  $s(t+1) = G \cdot s(t)$ . Toto zobrazení bude fungovat jako složení dvou dílčích zobrazení: matice  $F$  (fitness) a matice  $M$  (mutace a křížení).

Nejprve uvažme  $G = F$  (tj. nemáme žádné genetické operátory). Platí  $E(p(t+1)) = s(t)$ , kde  $E$  je střední hodnota. Pak díky  $s(t+1) \sim F \cdot p(t+1)$  (liší se jen o multiplikativní odchylku) můžeme psát  $E(s(t+1)) \sim F \cdot s(t)$ . V případě konečné populace nám výběrové chyby mohou způsobit odchylku od  $E()$ , obecně čím větší populace, tím menší odchylka. U nekonečné populace by to bylo přesné.

Nyní se podívejme na situaci  $G = M$ , tedy neřešíme selekci. Zde využijeme pomocnou veličinu  $r(i, j, k)$ , která udává pravděpodobnost, že z  $i$  a  $j$  vznikne  $k$ . Když tuto veličinu známe, pak lze vyjádřit

$$E(p_k(t+1)) = \sum_{i=0}^{2^l-1} \sum_{j=0}^{2^l-1} s_i(t) \cdot s_j(t) \cdot r(i, j, k)$$

Funkci  $r(i, j, k)$  lze analyticky spočítat.<sup>1</sup>

Závěr: JJGA pracující prostřednictvím  $G$  je dynamický systém,  $p(t)$  či  $s(t)$  jsou body (trajektorie). Neznáme jeho pevné body, můžeme však vyjádřit pevné body jednotlivých zobrazení: Pevné body  $F$  jsou populace se stejnou fitness, stabilní pevný bod  $F$  je maximální stejná fitness. (Jediný pevný bod  $M$ : stejné pravděpodobnosti  $s$  (resp. stejné zastoupení jedinců  $p$ ). Kombinace míchání  $M$  a ostření  $F$  odpovídá fenoménu *punctuated equilibria* (známe z biologie)).

### 3.2.2.3 Konečné populace

Pro konečné populace lze definovat GA jako *markovovský řetězec*, tj. stochastický proces v systému se stavů a přechody mezi stavů s definovanými pravděpodobnostmi. Pravděpodobnost přechodu závisí vždy pouze na předchozím stavu. Jeden stav systému je jedna konkrétní populace. Populací o  $n$  jedincích délky  $l$  je  $N = \binom{n+2^l-1}{2^l-1}$ , matice pravděpodobností přechodů je  $N \times N$ , podrobnější rozbor opět v Nerudových slajdech.

Dokázala se korespondence ideálního JJGA s nekonečnou populací a modelu s konečnou populací (pro  $n \rightarrow \infty$ ).

## 3.3 Evoluční strategie, diferenciální evoluce, koevoluce, otevřená evoluce.

### 3.3.1 Evoluční strategie

**Evoluční strategie (ES)** je metoda pro optimalizaci reálných funkcí, která je výjimečná tím, že jako první používala *self-adaptation*, tj. parametry evoluce byly součástí jedince – šlo tedy o „evoluci evoluce“. Evolvovaný jedinec se tedy skládá z **genetických parametrů** ovlivňujících chování a **strategických parametrů** ovlivňující evoluci.

Vznikly v 60. letech, autoři **Rechenberg a Schwefel**. ES jsou jedna z větví evolučních algoritmů (anglicky se jako „umbrella term“ používá buď *Evolutionary Algorithms* nebo *Evolutionary Computation*), nicméně nevyčlenily se v průběhu, jak by se dalo čekat. ES se vyvíjely paralelně s GA a EP (evolučním programováním), až později došlo k seskupení těchto tří metod pod jeden společný obor.

#### 3.3.1.1 Notace

**$M$**  počet jedinců v populaci

**$L$**  počet vznikajících potomků

**$R$**  počet rodičů každého nového jedince

Doporučení:  $M > 1$ ,  $L > M$  aby se vytvořily různé selekce (např.  $L \approx 7 \cdot M$ )

---

<sup>1</sup><http://ktiml.mff.cuni.cz/~neruda/eva2-13.pdf>

### 3.3.1.2 Varianty

**(M+L) ES** – M jedinců do nové populace je vybráno z M+L starých i nových jedinců

**(M,L) ES** – M nových jedinců je vybráno jen z L nových potomků (tato varianta se ukázala být robustnější vůči uvíznutí v lokálních extrémech)

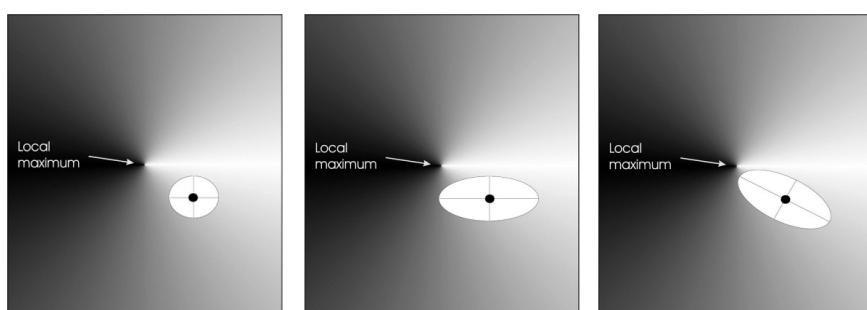
### 3.3.1.3 Kódování, genetické operátory

Jedinec je tvaru  $C(i) = [G_n(i), S_k(i)]$ ,  $k \in \{1, n, 2n\}$ , tedy první část je samotný genotyp, druhá část jsou parametry evoluce, konkrétně **hodnoty standardních odchylek floating point mutací**. Podle hodnoty  $k$  rozlišujeme 3 varianty:

**k=1** : jedna společná odchylka pro všechny parametry

**k=n** : nekorelované mutace; geometricky se mutuje po elipse rovnoběžné s osami

**k=2n** : korelované mutace, odpovídají mutování z n-rozměrného normálního rozdělení. Pro uložení korelační matice stačí n-rozměrný vektor (???). Tato matice odpovídá matici rotace (tj. elipsa se natočí v optimálním směru).



**Populační cyklus** vypadá takto:

```

1 n=0, randomly init population P[n] of size M
2 evaluate fitness of individuals in P[n]
3 while not good enough:
4     repeat L times:
5         choose R parents
6         crossover , mutate , evaluate new individual
7         choose M new individuals (by the type of ES)
8         n++

```

Nový jedinec je akceptován pouze tehdy, je-li lepší než rodič – pak rodiče nahradí.

**Mutace** jedince probíhá takto: *genetické* parametry (genotyp) jsou změněny přičtením náhodného čísla s norm. rozdělením s příslušnou odchylkou (a případně i rotací). *Strategické* parametry = odchylky jsou zvětšeny nebo zmenšeny podle „pravidla 1/5“, což heuristika, která říká, že nejlepší je, když má mutace úspěšnost 20 % (tzn. při menší zvyš odchylku, při větší sníž). V případě varianty s rotacemi jsou mezi strategickými parametry také hodnoty rotací – ty se mutují jednoduše přičtením náhodného čísla z  $N(0,1)$ .

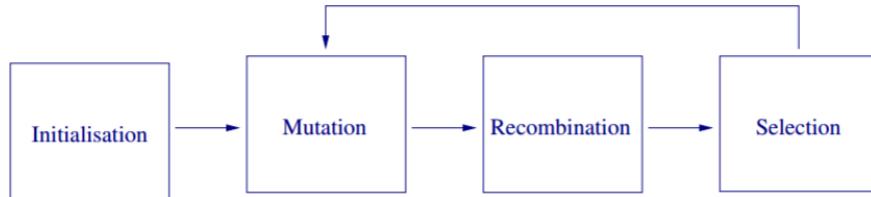
**Křížení** je uniformní, typicky více rodičů. Podle velikosti R mluvíme o lokálním ( $R=2$ ) nebo globálním ( $R=M$ ). Může být diskrétní (použij hodnotu náhodně vybraného rodiče) nebo aritmetické (průměr).

### 3.3.1.4 CMA-ES

Dnes nejkomplexnější verzí je **CMA-ES** (Correlation Matrix Adaptation ES). Udržuje se korelační matice proměnných, která se inkrementálně aktualizuje po každém kroku metodou maximum-likelihood.

### 3.3.2 Diferenciální evoluce

Alternativní algoritmus spojité optimalizace, který funguje dobře i pro funkce které jsou nediferencovatelné, nespojitě, zašuměné, neseparabilní nebo vysoce podmíněné. Každý jedinec prochází opakovaně cyklem **mutace** → **křížení** → **selekce**. Během mutace dochází k posunu „podle ostatních“, tj. jedinec se posouvá ve směru populace. DE je také invariantní vůči rotaci či škálování prohledávaného prostoru.



#### 3.3.2.1 Mutace

V populaci  $p$  zvolíme pro jedince  $x_{i,p}$  tři různé jedince:  $x_{a,p}$ ,  $x_{b,p}$  a  $x_{c,p}$ . Definujeme donora:

$$v_{i,p+1} = x_{a,p} + F \cdot (x_{b,p} - x_{c,p})$$

kde  $F \in \langle 0; 2 \rangle$  je parametr mutace.

#### 3.3.2.2 Křížení

Uniformní křížení původního jedince s donorem. Parametr  $C$  určuje pravděpodobnost změny,  $rand_{ij} \in \langle 0, 1 \rangle$  je pseudonáhodné číslo. Číslo  $I_{rand}$  je náhodný index z  $\langle 1, 2, \dots, D \rangle$ , kde  $D$  je délka jedince. Pak vytvoříme pokusný vektor  $u_{i,p+1}$  jako:

$$u_{i,p+1}[j] = v_{i,p+1}[j] \leftrightarrow rand_{ji} \leq C \vee j = I_{rand}$$

$$u_{i,p+1}[j] = x_{i,p}[j] \leftrightarrow rand_{ji} > C \wedge j \neq I_{rand}$$

Díky  $I_{rand}$  je zajištěno, že ve výsledku bude aspoň jeden prvek z donora.

#### 3.3.2.3 Selekce

Jako nový jedinec  $x_{i,p+1}$  se vezme lepší (dle fitness) z dvojice  $x_{i,p}$ ,  $u_{i,p+1}$ .

### 3.3.3 Koevoluce

Situace, kdy se dva či více druhů vyvíjí závisle na sobě, tj. fitness jedince není závislá pouze na prostředí, ale především na interakci s jinými jedinci (tzv. *kontextová fitness*). Rozlišujeme koevoluci **kooperativní** a **kompetitivní**. Typickým modelem kompetitivní koevoluce je tzv. **dravec-korist** (predator-prey). Kompetitivně lze rovněž vyvíjet strategie pro nejrůznější hry - např. dáma apod.

#### 3.3.3.1 Kompetitivní evoluce: dravec-korist

Na začátku jednoduché strategie obou hráčů se postupně zdokonalují, a to i bez změny prostředí („závody ve zbrojení“). Jedná se o *inkrementální učení*. Řeší se i problém s nastartováním učení tj. kdybychom pustili dravce na dokonale se chovající korist, tak ji nikdy nechyti a selekce nebude fungovat.

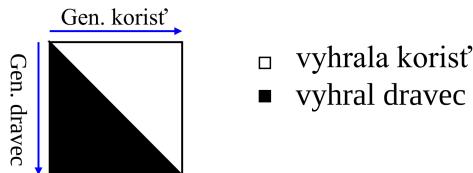
**Příklady:**

- hráči Tic-Tac-Toe
- strategie pro věžovo dilema
- třídící vs. „parazitní“ programy
- roboti „dravec“ a „korist“ v aréně (korist rychlejší, dravec lépe vidí)

### 3.3.3.1.1 Sledování fitness

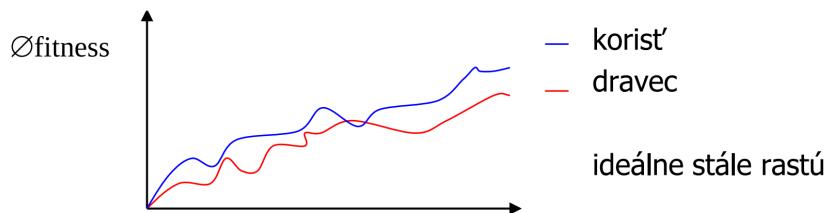
V případě koevoluce nestačí jednoduše sledovat fitness, ta totiž závisí na měnící se strategii soupeře. Tj. změna chování hráče vede ke změně *krajiny fitness* pro soupeře. Používají se dvě strategie:

**CIAO (Current Individual vs. Ancestral Opponents)** V každé generaci necháme nejlepšího jedince soupeřit se všemi nejlepšími jedinci z předchozích generací. Výsledek můžeme zaznamenat do čtvercové bitmapy, kde bílý pixel znamená vítězství kořisti a černý vítězství dravce. Získáme tak grafickou reprezentaci evoluce, kterou můžeme sledovat v průběhu.



Obrázek 3.1: CIAO obrázek ukazující „ideální“ výsledek (nejlepší z nové generace vždy porazí nejlepší z předchozích)

**Mistrovství** Vyhodnocení po skončení evoluce. Necháme soupeřit nejlepšího z každé generace s nejlepšími soupeři ze všech generací (tj. nejen z předchozích. „Fitness“ jedince je pak množství vítězství.



### 3.3.3.2 Zacyklení strategií

Typickým problémem kompetitivní koevoluce je **zacyklení** strategií. Tj. existuje množina efektivních chování jako reakce na chování soupeře, jakmile však změní strategii, změní ji i soupeř, na což reaguje svou změnou strategie, ... Toto chování by ideálně mělo najít optimální strategii pro všechny strategie soupeře, často se ale stává, že se zacyklíme.

Příklad pro roboty dravec-kořist:

- **Strategie dravce**
  - D1: pronásleduj kořist
  - D2: „číhej“, tj. zůstaň na místě, sleduj kořist a zaútoč pouze pokud se kořist přiblíží
- **Strategie kořisti**
  - K1: stůj na místě (u stěny)
  - K2: rychle se pohybuj a vyhýbej se dravci (i stěnám)

Na každou strategii dravce funguje nějaká strategie kořisti a naopak: D1 > K1 > D2 > K2 > D1 > ...

TODO koevoluce

### 3.3.4 Otevřená evoluce

Evoluce bez konce, tj. neexistuje ukončovací podmínka ani optimální řešení, neboť fitness funkce je *dynamická*. Často je dynamické i kódování jedince – např. pokud u neuroevoluce mohou sítě ubývat a přibývat neurony či synapse.

Míra změn fitness může být různá:

**malé změny** robot se opotřebuje, v chemické továrně se trochu změní suroviny apod.  
**výrazné morfologické změny** vrcholy zanikají, nové vznikají, „emerging markets“  
**cyklické změny** roční období, spotřeba elektřiny, ...  
**katastrofické změny** bouchla elektrárna, nehoda na dálnici, ...

Byly navrženy různé postupy pro vyrovnání se s dynamickou fitness. Například *vyvolaná hypermutace* (při snížení fitness), *náhodná migrace* (při snížení fitness se generují noví náhodní jedinci) a obecně metody vedoucí k udržení diverzity populace: zmenšení selekčního tlaku, crowding, niching, subpopulace, dynamické druhy (které se množí jen mezi sebou).

Dynamická fitness je vyvolaná i kompetitivní koevolucí, a tedy evoluční souboje typ lovec-kořist lze označit za otevřenou evoluci.

## 3.4 Rojové optimalizační algoritmy.

### 3.4.1 Optimalizace hejnem částic

Anglicky *particle swarm optimization* – jedná se o populační prohledávací algoritmus inspirovaný hejny ptáků/hmyzu/ryb. Jedinec se nazývá **částice** a jde typicky o vektor reálných čísel. Neexistují křížení ani mutace, jak je známe.

#### 3.4.1.1 Algorimus

```
1 init each particle
2 while not (max iterations or minimum error)
3     foreach particle p:
4         f = fitness(p)
5         if f > p.pBest
6             p.pBest = f
7     gBest = max(p.pBest for each particle p)
8     foreach particle p:
9         p.v = p.v +
10            + c1 * rand(0,1) * (pBest - p.position)
11            + c2 * rand(0,1) * (gBest - p.position)
12         p.position += p.v
```

Každá částice má rychlostní vektor  $v$ , který se aktualizuje podle nejlepší pozice částice v historii (řádek 10) a nejlepší globální pozice v historii (řádek 11). Učící konstanty  $c1, c2$  se často nastavují na  $c1 = c2 = 2$ .

Pozn. celé to asi může být v jednom for cyklu s tím, že  $gBest$  se aktualizuje průběžně a ne po "generacích".

PSO jsou podobné GA – stochastický model, fitness pro ohodnocení, náhodná iniciální konfigurace atd. – ale současně se v mnohem liší: neexistují genetické operace, částice mají paměť a výměna informací probíhá pouze od nejlepších částic ostatním.

PSO nepoužívají gradient pro optimalizaci, což znamená, že optimalizovaná funkce nemusí být diferencovatelná.

### 3.4.2 Další rojové optimalizační algoritmy

Obecně se definuje **Swarm Intelligence (SI)** jako „*The emergent collective intelligence of groups of simple agents*“. Dvě základní vlastnosti jsou sebeorganizace a dělba práce. Kromě PSO sem lze zařadit ještě Ant Colony Optimization (ACO), Artificial Bee Colony (ABC), Glowworm Swarm Optimization, Cuckoo Search Algorithms a v jistém směru sem spadá i diferenciální evoluce (jelikož je population-based). Kromě DE se tyto algoritmy (a mnohé další, třeba i simulated annealing) označují také nálepkou *metaphor-based metaheuristics*, a to díky tomu, že jsou typicky inspirovány nějakou reálnou skutečností (chování roje, chování mravenců, zpracování kovů apod.). Tyto

se staly terčem kritiky pro nedostatek inovace a efektivity, kterou schovávají právě za poutavou metaforou ([https://en.wikipedia.org/wiki/List\\_of\\_metaphor-based\\_metaheuristics](https://en.wikipedia.org/wiki/List_of_metaphor-based_metaheuristics)).

### 3.4.2.1 Ant Colony Optimization (ACO)

Systém inspirovaný chování mravenců při hledání potravy, konkrétně použitím feromonů. V podstatě jde o techniku nalezení nejlepší cesty v grafu. Agent (mravenec) se pohybuje v grafu a zanechává za sebou feromonovou stopu. Pohyb se řídí rovnicí:

$$p_{ij}^k(t) = \frac{([\tau_{ij}(t)]^\alpha \cdot [\eta_{ij}]^\beta)}{\left( \sum_{n \in J_k} [\tau_{in}(t)]^\alpha \cdot [\eta_{in}]^\beta \right)}$$

$p_{ij}^k(t)$  je pravděpodobnost přechodu mravence  $k$  z uzlu  $i$  do uzlu  $j$  v čase  $t$ ,  $J_k$  jsou uzly, kam má mravenec  $k$  povoleno jít,  $\eta_{ij}$  symbolizuje „viditelnost“ mezi  $i$  a  $j$  a  $\tau_{ij}(t)$  reprezentuje množství nevypařeného feromonu mezi  $i$  a  $j$  v čase  $t$ . Parametry  $\alpha$  a  $\beta$  kontrolují, nakolik prohledávání závisí na feromonech. Každý mravenec  $k$  má také tabu list navštívených uzelů, aby nenavštívil nějaký dvakrát.

Umístění feromonů se řídí rovnicí:

$$\Delta\tau_{ij}^k = \begin{cases} \frac{Q}{L_k}(t) & \text{leží-li hrana } ij \text{ na cestě mravence } k \\ 0 & \text{jinak} \end{cases}$$

$Q$  je konstanta,  $L_k$  je cena cesty (její délka),  $k$  je mravenec,  $t$  je čas (iterace).

Feromony se časem vypařují. Celková změna feromonu na hraně  $ij$  je tedy:

$$\tau_{ij}(t+1) = (1-p)\tau_{ij}(t) + \sum_{k=1}^m [\Delta\tau_{ij}^k(t)]$$

kde  $m$  je počet mravenců a  $p$  je míra vypařování. To je důležitý parametr algoritmu, který ovlivňuje poměr mezi explorací a exploatací.

Mírně vyložená varianta ACO je Ant Colony System (ACS).

Dobrý přehled dalších algoritmů (např. Artificial Bee Colony) lze nalézt na <http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0122827>

## 3.5 Memetické algoritmy, hill climbing, simulované žíhání.

**Memetické algoritmy** jsou EA obohacené o lokální prohledávání uvnitř cyklu. Základní princip je ten, že evolucí vyvinutý jedinec ještě před ohodnocením podstoupí proceduru lokálního prohledávání – to může být např. hill climbing, simulované žíhání nebo backpropagation u ANN).

Inspirováno **mémym** (angl. *meme* (čti /mi:m/)), což jsou ideje/myšlenky šířící se ve společnosti, které mohou podstoupit mutaci či selekci, stejně jedinci v klasické evoluci.

Alternativně jsou MA označovány jako *Baldwinian EA*, *Lamarckian EA* nebo jako *genetic local search*.

### 3.5.0.1 Algoritmus

```

1 t = 0, initialize P[0]
2 P[0].localSearch()
3 evaluate(P[0])
4 while notTerminated do
5     P[t] = selectIndividuals()
6     mutate(P[t])
7     P[t].localSearch()
8     evaluate(P[t])
9     P[t+1] = buildNextGen(P[t])
10    t++

```

### 3.5.0.2 Existují dva přístupy, co dělat s výsledkem:

**Lamarckismus** Když prohledáváním najdu lepšího jedince, vezmu ho (to je ovšem darwinicky nekorektní, změnil jsem genotyp na základě fenotypu).

**Baldwinismus** Když lokálním prohledáváním najdu lepšího jedince, vezmu jen jeho fitness (ale neměním genotyp).

### 3.5.0.3 Hill climbing

Nejjednodušší informovaná metoda lokálního prohledávání stavového prostoru. Vstupem je uzel, ze kterého se má prohledávání zahájit. Nejprve je uzel expandován - jsou vygenerovány jeho sousední uzly a tyto uzly jsou ohodnoceny. Algoritmus vybere z nich nejlépe ohodnocený uzel a ten je nadále expandován.

Algoritmus tak expanduje uzly se stále vyšším ohodnocením, dokud nenarazí na uzel po jehož expanzi mají všechny jeho sousední uzly horší ohodnocení. Algoritmus nemá paměť, navštívené uzly okamžitě zapomíná.

Varianty:

**first choice HC vs. steepest ascent HC** First choice HC vezme prvního lepšího souseda, steepest zkusi všechny a vezme nejlepšího.

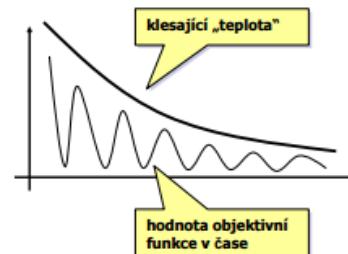
**stochastic HC** Ze zlepšujících sousedů vybere náhodně podle míry zlepšení.

**random restart HC (or shotgun HC)** Náhodné restarty v různých místech, částečně řeší problém uvíznutí v lokálním optimu.

### 3.5.0.4 Simulované žíhání

Lokální prohledávání kombinující HC a náhodnou procházku. Algoritmus náhodně volí stav okolí a vydá se do něj pokud:

- je lepší než aktuální stav
- je horší, ale zhoršení je povoleno s určitou mírou pravděpodobnosti odvozenou od aktuální **teploty**; teplota se přitom postupně snižuje podle „ochlazovacího“ schématu



### 3.5.0.5 Algoritmus

```
1 current = init()
2 for t = 1 to ∞
3     T = schedule(t)
4     if T = 0 then
5         return current
6     next = randomNeighbor(current)
7     ΔE = f(next) - f(current)
8     if (ΔE > 0) or (eΔE/T < rand(0,1))
9         current = next
```

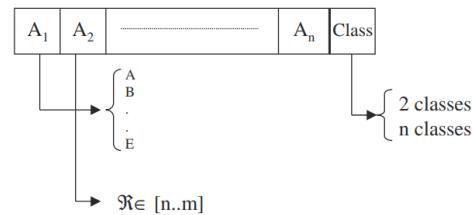
## 3.6 Aplikace evolučních algoritmů (evoluce expertních systémů, neuroevoluce, řešení kombinatorických úloh, vícekriteriální optimalizace).

### 3.6.1 Evoluce expertních systémů

Expertní systémy jsou modely z oblasti *strojového učení*, jejichž základní charakteristikou je, že pro výpočet svého výstupu používají pravidla IF: THEN. Zastupují tedy větev strojového učení

založenou na *formální logice* (druhá, protikladná větev, reprezentovaná především neuronovými sítěmi, je založena na *konekcionismu*, tj. modeluje biologické fungování mozku, nikoli logické). Expertní systémy, které se učí pomocí simulované evoluce, se nazývají **learning classifier systems (LCS)**. Existují dvě základní paradigmata: *michiganský přístup* (jedinec je pravidlo) a *pittsburghský přístup* (jedinec je množina pravidel).

Nejprve ještě ke klasifikaci obecně. **Klasifikační úloha** spočívá v označování předkládaných instancí z nějaké domény správnou „nálepkou“, jejich přiřazení do nějaké **třídy** (kterých je omezená množina, často pouze dvouprvková). **Instance** mají pevnou strukturu danou konkrétní problémovou doménou, která se dá vyjádřit pevným počtem **atributů** (nebo *příznaků*, angl. *features*), které charakterizují tuto instanci; a v případě anotovaného datasetu má instance jako atribut i správnou třídu. Atributy obecně mohou být nominální (prvek nejaké konečné množiny), diskrétní nebo spojité.



### 3.6.1.1 Pittsburghský přístup

Pittsburghský přístup je nejbližší klasickému GA. Jedinec v populaci je kompletní množina pravidel řešící daný problém. Tato množina může být různě velká, máme tedy jedince variabilní délky. Při klasifikaci může dojít ke *konfliktu pravidel*, tj. více pravidel může být použito pro klasifikaci dané instance – to lze řešit např. utříděním pravidel do seznamu, pak se z pravidel „matchujících“ danou instanci aplikuje to s nejnižším pořadovým číslem. **Genetické operátory** jsou poměrně komplikované, typicky jich je více a některé fungují na úrovni celých množin, některé na úrovni jednotlivých pravidel a některé na úrovni termů v pravidlech.

**Fitness** jedince se počítá takto: pro každou instanci v trénovací množině se vezme první matchující pravidlo a přiřadí se třída daná tímto pravidlem – ta se porovná s očekávanou třídou (trénovací vzorky jsou anotované). Fitness je pak dána procentem úspěšně klasifikovaných instancí.

Konkrétním reprezentantem pittsburghského přístupu je například systém GABIL. Ten je určen pro řešení *binární klasifikace* (tj. pouze 3 třídy) a funguje pouze pro domény s *nominálními* atributy. Jednotlivá pravidla (tzv. *komplexy*) jsou ve formátu *predikát  $\rightarrow$  třída*, kde predikát je výraz v konjunktivní normální formě (CNF) ve tvaru

$$(A_1 = V_1^1 \vee \dots \vee A_1 = V_1^m) \wedge \dots \wedge (A_n = V_n^1 \vee \dots \vee A_n = V_n^m)$$

kde  $A_i$  je  $i$ -tý atribut a  $V_i^j$  je  $j$ -tá hodnota  $i$ -tého atributu. Taková pravidla lze pak zapsat pomocí binárního řetězce, například

1100|0010|1001|1

tedy vyjadřuje pravidlo: pokud první atribut nabývá první či druhé hodnoty, druhý atribut třetí hodnoty a třetí atribut první nebo čtvrté hodnoty, pak instance patří do třídy 1.

Na těchto binárních řetězcích lze pak snadno aplikovat genetické operátory. Ty fungují na různých úrovních:

- na **úrovni jedince (množiny pravidel)** výměna pravidel, kopírování pravidel, generalizace pravidla, smazání pravidla, specializace pravidla, zahrnutí jednoho pozitivního příkladu
- na **úrovni komplexů** rozdělení komplexu na 1 selektoru, generalizace selektoru (nahrazení 11...1), specializace generalizovaného selektoru, zahrnutí negativního příkladu
- na **selektorech** mutace  $0 \leftrightarrow 1$ , rozšíření  $0 \rightarrow 1$ , zúžení  $1 \rightarrow 0$ ,

### 3.6.1.2 Michiganský přístup

Zde je jedincem v populaci pravidlo. V klasickém GA typicky dojde ke konvergenci populace k jedinému řešení – my ale chceme získat množinu pravidel, ideálně docela ortogonálních. Jednou možností je **Iterative Rule Learning** přístup, který postupně generuje pravidla iterativním spouštěním GA. Tj. spustí se GA, najde se nejlepší pravidlo, to se uloží a z trénovací množiny se

odstraní všechny správně klasifikované instance. Toto se opakuje, dokud není trénovací množina prázdná.

Michiganský přístup funguje trochu jinak. Na klasifikaci se podílí *celá populace*. Evoluce neprobíhá po generacích, nýbrž jen občas a jen na části populace. Konkrétně lze rozlišit fázi *objevování nových pravidel* (pomocí GA) a fázi *učení*, během které dochází k odměňování/penalizaci jedinců, jak si vedou při klasifikaci – od toho je pak odvozena jejich fitness.

Pravidla mohla mít na pravé straně kromě klasifikačního výstupu i další příznaky, tzv „zprávy“. Na levé straně pravidel se pak mohly vyskytovat „receptory“ pro jejich příjem. Celý systém má frontu zpráv.

Problémem při klasifikaci celou populací realizace algoritmu odměn pro pravidla. Původní Hollandův návrh obsahoval tzv. *bucket brigade*: pravidla musejí dát část svého „kreditu“, když chtějí soupeřit o možnost být v cestě k řešení. Tento systém byl ale poměrně těžkopádný.

Zjednodušením tohoto systému byl **ZCS (Zero Classifier System)**. Neměl vnitřní zprávy ani žádný složitý systém alokace odměn. Síla pravidel se upravovala podle jednoduchého algoritmu:

- pravidlům, která odpovídají vstupu, ale mají jiný výstup, se síla zmenší násobením konstantou  $0 < T < 1$
- všem pravidlům se zmenší síla o malou část B
- tahle síla se rozdělí rovnoměrně mezi pravidla, která *minule* odpověděla správně, zmenšená o faktor  $0 < G < 1$
- nakonec, odpověď od systému se zmenší o B a rozdělí rovnoměrně mezi pravidla, která *ted'* odpověděla správně

Inovací bylo přidání **cover** operátoru. Ten řešil případy, kdy nebylo žádné pravidlo pro daný vstup. Pak bylo vygenerováno ad hoc. Pravidla jsou ve formátu  $01\#|0$ , kde 1 znamená „příznak nastal“, 0 opak a # je „žolík“. Pro nebinární nominální atributy lze samozřejmě použít rozšířenou abecedu. Ad hoc pravidlo bylo tedy vygenerováno tak, že se nějaké náhodné vstupní atributy nahradily # a zvolil se náhodný výstup.

Dalším vylepšením byl systém **XCS**, který zakládá fitness pravidla na jeho přesnosti, nikoli „výdělku“, tj. dává šanci prosadit se i pravidlům, která vedou k akcím s malou odměnou. Tímto je podpořeno vyvíjení komplexního systému pravidel pokrývajícího co nejvíce případů.

### 3.6.2 Neuroevoluce

Neuronové sítě se klasicky učí pomocí nějaké varianty backpropagation, ale jsou situace, kdy může být lepší pokusit se vyvinout optimální síť (řešící daný problém) pomocí EA. Mnohdy nelze *učení s učitelem* použít vůbec – například v robotice – a je nutno spoléhat na *zpětnovazební učení* (reinforced learning).

Předmětem evoluce mohou být synaptické váhy, struktura sítě, nebo obojí najednou.

Existují i hybridní přístupy, kdy jsou EA zkombinovány s lokálním prohledáváním. Je například možno pomocí GA najít počáteční vah a poté síť doučit pomocí klasické BP → výsledná fitness (BP je citlivé na počáteční nastavení vah).

#### 3.6.2.1 Učení vah

Nejjednodušší je zakódovat váhy do vektoru a pak použít klasický floating point GA se standardními genetickými operátory. Problémy: mnoho parametrů optimalizováno naráz (stovky až tisíce hodnot synaptických vah); různé vzájemně si konkurenční kódy reprezentují funkčně totožné sítě; předčasná konvergence. (Jednoduché) krížení navíc nefunguje příliš dobře, neboť výsledné vektory často nekódují validní síť.

#### 3.6.2.2 Učení struktur

Prostor možných architektur je nekonečný a především nediferencovatelný, takže gradientní metody jsou k ničemu. Evoluce však může fungovat bez problémů. Strukturu sítě lze zakódovat několika způsoby:

**přímé kódování** – binární matice synapsí, před evolucí se typicky převedou na jeden dlouhý linearizovaný vektor

**gramatické kódování** 2D formální gramatiky, které jsou „programem“ pro vytvoření matice růst sítě ve 2D rané pokusy v evoluční robotice, velmi neefektivní

**celulární kódování** v podstatě Genetické programování – program obsahuje instrukce, jak zkonztruovat síť (operace typu: přidej neuron, rozděl neuron sériově/paralelně, přepoj synapsi apod.)

### 3.6.2.3 Evoluce částečných sítí

**SANE** (Symbiotic Adaptive Neuroevolution) je variantou neuroevoluce, kdy je struktura pevně daná: jediná skrytá vrstva, pevný počet skrytých neuronů. Jedinci v evoluci jsou neurony (včetně vstupních a výstupních vah). Při ohodnocování se náhodně vybere pevný počet neuronů z populace, postaví se síť a ta se vyhodnotí. Toto se opakuje tak, aby každý neuron z populace byl vybrán aspoň 10×. Fitness neuronu je pak průměrná fitness všech sítí, jichž byl součástí.

**ESP** (Enforced Sub-Populations) funguje na podobném principu, tj. architektura je opět předem daná s pevně daným počtem neuronů (na rozdíl od SANE může být i rekurentní). Rozdíl je v tom, že pro každý z těchto „slotů“ se vyvíjejí neurony zvlášť, v oddělených subpopulacích. Takto navržená evoluce podporuje diverzitu (dobrá síť potřebuje různé druhy neuronů), potlačuje redundanci (neurony se vyvíjí pro kompatibilní role) a implicitně rozděluje prohledávaný prostor na podúlohy.

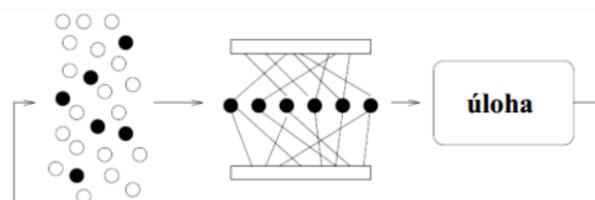
### 3.6.2.4 NEAT (Neuroevolution of Augmenting Topologies)

Základními charakteristikami algoritmu NEAT jsou:

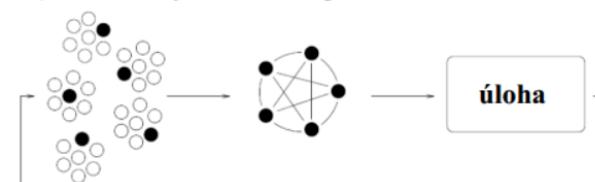
1. evoluce topologie společně s vahami
2. smysluplné křížení pomocí historických značek
3. ochrana inovací pomocí druhů

Evoluce topologie zahrnuje přidání/odebrání synapse, přidání/odebrání vrcholu. Typicky se začíná od menší sítě, která postupně roste.

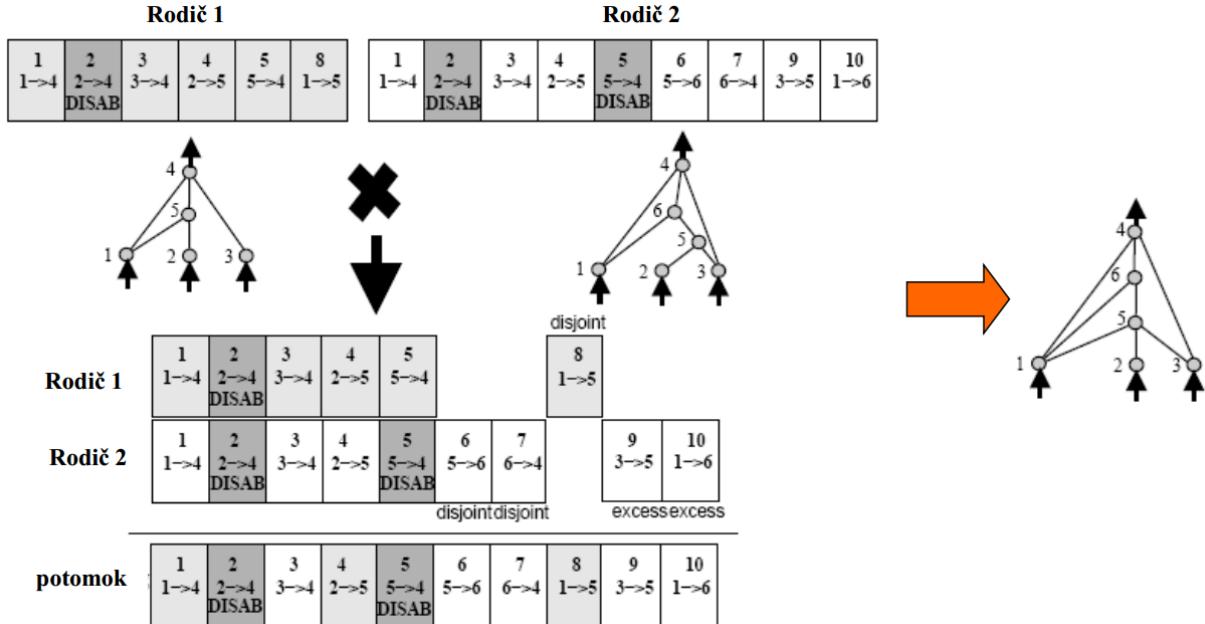
Každý neuron a každý synaptický spoj dostanou při svém vzniku svou *historickou značku* (nebo „rodné číslo“). Tato značka se dědí, takže i u různých potomků lze poznat, zda neurony/spoje mají stejný evoluční původ. Při křížení dvou jedinců se potom postupuje takto: pokud se uzel (neuron či spoj) vyskytuje pouze v jednom rodiči, je automaticky zkopirován; pokud se vyskytuje v obou, je náhodně vybrán jeden z nich.



a) SANE – Symbiotic Adaptive Neuro-Evolution



a) ESP – Enforced Sub-Populations

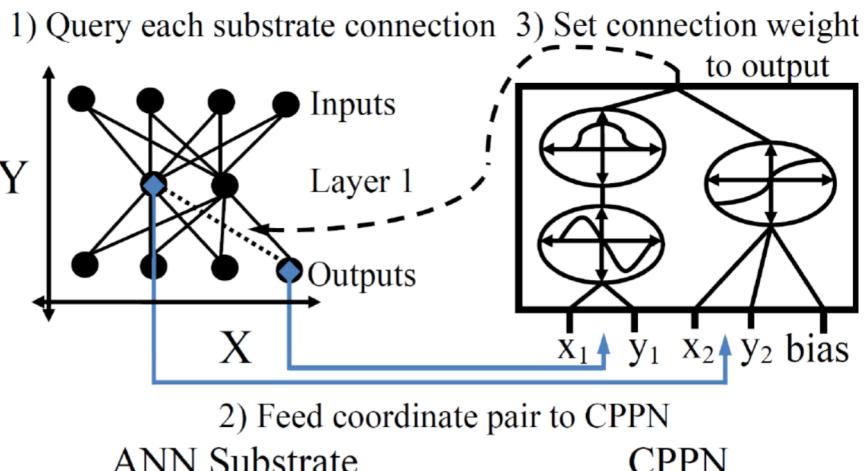


Když během evoluce vznikne jedinec, který se výrazně liší od ostatních (na to musí být definovaná nějaká metrika), založí se nový **druh**. Ten je několik generací chráněný – jedinci nového druhu soutěží pouze mezi sebou.

### 3.6.2.5 HyperNEAT (Hyper-cube NEAT)

Varianta NEAT, kdy nevyvíjíme síť samotnou, ale jinou síť, která nám naši síť postaví. Tato pomocná síť se nazývá CPPN (Compositional Pattern Producing Network) a liší se od klasických sítí především tím, že v neuronech se místo sigmoidy používají nejrůznější geometrické funkce, např. absolutní hodnota, bipolární sigmoida, Gaussovská funkce, lineární funkce, sinusoida nebo kroková funkce. Na vstupu CPPN je dvojice souřadnic dvou neuronů (pro 2D síť tedy 4 souřadnice:  $x_1, y_1, x_2, y_2$ ), na výstupu je váha, která se má přidělit synapsi mezi neurony na těchto souřadnicích.

Pomocí CPPN tedy můžeme doplnit vhodné váhy do ANN sítě nazývané *substrát*, ta se poté použije pro řešení daného problému. Tato konstrukce umožňuje využít geometrické pravidelnosti daného problému. Příklady použití: šachy, dáma, dravec-kořist, Go, atd. Neurony jsou uspořádány v prostoru daném doménou problému, lze je adresovat souřadnicemi, které využívá CPPN. Síť může být i 3D, kde jednotlivé vrstvy sítě jsou umístěny v různých  $z$  souřadnicích.



Pro evoluci CPPN se používá klasický NEAT s 2 odlišnostmi: nově vygenerovaný neuron dostane náhodně jednu z geometrických funkcí (namísto sigmoidy); a fitness CPPN se spočte podle správnosti výstupu ANN substrátu vygenerovaného pomocí CPPN.

Substrát lze libovolně škálovat. Můžeme třeba použít dvojnásobný počet neuronů, kterým pak ale přidělíme souřadnice ze stejného intervalu, jako u menšího substrátu.

### 3.6.3 Řešení kombinatorických úloh

EA jsou velmi vhodné pro řešení NP-těžkých úloh. Rozebereme několik příkladů:

#### 3.6.3.1 Batoh (0-1 knapsack problem)

*Problém:* máme batoh o kapacitě  $CMAX$ , množinu  $n$  věcí, z nichž každá má cenu  $v(i)$  a objem  $c(i)$ . Úkolem je naplnit batoh tak, abychom maximalizovali sumu cen obsažených věcí a současně nepřekročili kapacitu batohu.

**Kódování:** Binární jedinci délky  $n$ , přímočará bitová mapa. Např. 0110010 znamená, že bereme věci s indexy 2,3 a 6. Pozor, jedinci nemusí splňovat podmínu o  $CMAX$ .

**Operátory:** Lze použít přímočaré křížení, mutaci a klasickou selekci.

**Fitness:** Bude mít 2 členy:

$$\max_{B \subset \{1..n\}} \sum_{i \in B} v(i) \text{ a současně } \min_{B \subset \{1..n\}} CMAX - \sum_{i \in B} c(i)$$

Problém s víceúčelovou optimalizací lze řešit různě:

- Oba členy fitness zvážit a sečíst
- Použít jeden z obecných EA pro víceúčelové funkce
- Anebo chytře změnit zakódování:
  - 1 znamená: DEJ věc do batohu POKUD se nepřeplní
  - takto dosáhneme i toho, že všechny řetězce jsou přípustná řešení

#### 3.6.3.2 Problém obchodního cestujícího (TSP – travelling salesman problem)

*Problém:* Máme  $n$  měst, mezi nimi existují cesty definované délky (úplný graf, ohodnocené hrany). Cílem je navštívit všechna města s co nejmenšími náklady – tj. najít co nejkratší hamiltonovskou kružnicí.

**Fitness:** Délka cesty.

Existuje několik možných reprezentací a na nich závislých genetických operátorů:

**reprezentace sousednosti:** Cesta je seznam měst, město  $j$  je na pozici  $i$ , vede-li cesta z  $i$  do  $j$ . Např. (248397156) odpovídá cestě 1-2-4-3-8-5-9-6-7. Některé seznamy negenerují přípustnou cestu. Klasické křížení nefunguje. Schémata fungují, např (\*3\*...) značí všechny cesty s hranou 2-3.

**reprezentace bufferem** Máme buffer vrcholů, třeba uspořádaný. Reprezentace představuje pořadí města v bufferu, města se z bufferu postupně mažou. Např. buffer (123456789) a reprezentace (112141311) kóduje cestu 1-2-4-3-8-5-9-6-7. V této reprezentaci lze použít jednobodové křížení.

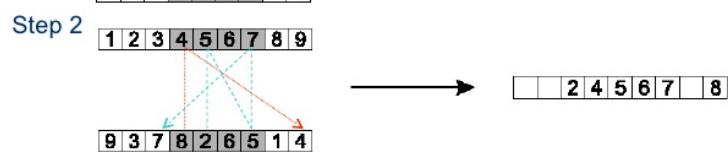
**reprezentace permutací** Nejvíce přímočaré: řetězec (517894623) kóduje cestu 5-1-7-8-9-4-6-2-3. Klasické křížení nefunguje, byly navrženy speciální operátory (dobrý přehled viz <sup>2</sup>):

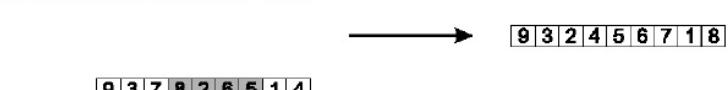
**PMX (partially mapped crossover** Náhodně vybereme podřetězec z rodiče 1, ten zkopiujeme do nového jedince. Pak se podíváme na tytéž pozice v rodiči 2. Pro každou hodnotu, která se ještě nevyskytuje v budovaném potomkovi, budeme hledat vhodné umístění: nechť chceme umístit hodnotu  $h$ , která se nachází v rodiči 2 na pozici  $i$ . Vezmeme hodnotu  $h_2$  na téže pozici  $i$ , ale v rodiči 1. Pak se podíváme, kde se  $h_2$  nachází v rodiči 2 – označme tuto pozici jako  $i_2$ . Je-li to mimo zkopiovanou oblast, pak jsme našli vhodné umístění a zkopiujeme původní hodnotu  $h$  do nového potomka na  $i_2$ . Je-li  $i_2$

<sup>2</sup><http://www.rubicite.com/Tutorials/GeneticAlgorithms/CrossoverOperators/PMXCrossoverOperator.aspx>

uvnitř kopírovaného intervalu, pak pokračujeme v hledání – do  $h_2$  dosadíme hodnotu na  $i_2$  v rodiči 1. Viz obr. 3.5.

**Step 1** 

**Step 2** 

**Step 3** 

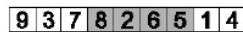
Obrázek 3.2: Partially mapped crossover (PMX)

**OX (order crossover)** Nejrychlejší křížení. Stejně jako v PMX nejprve vezmeme náhodný podinterval rodiče 1 a bez změny jej zkopírujeme do potomka. Poté vezmeme nepoužité hodnoty z rodiče 2 a počínaje pravým koncem zkopiovaného intervalu je dokopírujeme do potomka.

- Copy randomly selected set from first parent

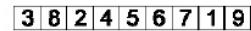


→ 



- Copy rest from second parent in order 1,9,3,8,2



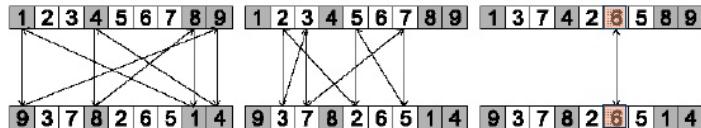
→ 



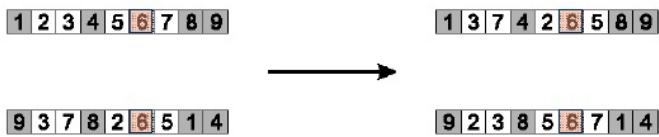
Obrázek 3.3: Order crossover (OX)

**CX (cyclic crossover)** Nejprve jsou identifikovány „cykly“ mezi dvěma rodiči: začneme první hodnotou v prvním rodiči, podíváme se na hodnotu na tomtéž indexu v rodiči 2, tuto hodnotu najdeme v rodiči 1, podíváme se na hodnotu na též indexu v rodiči 2 atd. dokud neužavřeme cyklus. Takto najdeme všechny cykly. Nového jedince pak vytvoříme kopírováním hodnot po cyklech: hodnoty z prvního cyklu zkopiujeme z prvního rodiče, hodnoty z druhého cyklu z druhého rodiče, hodnoty z třetího cyklu opět z prvního rodiče atd.

- Step 1: identify cycles



- Step 2: copy alternate cycles into offspring



Obrázek 3.4: Cyclic crossover (CX)

**ER (edge recombination)** Pro každé město si udělám seznam hran, které do/z něj vedou a jsou použity v některém z rodičů. Pak začnu náhodným městem a postupně napojuji další, přičemž vždy ze sousedů vybírám město s nejméně hranami (v případě shody náhodně). Může se stát, že nelze hranu vybrat, ale to se v praxi stává zřídka.

**ER2** Vylepšení: označím si hrany, které jsou dvakrát (tj. v obou rodičích) a při výběru jim dávám přednost. To zachovává ještě více hran z obou rodičů.

Element	Edges	Element	Edges
1	2,5,4,9	6	2,5+,7
2	1,3,6,8	7	3,6,8+
3	2,4,7,9	8	2,7+, 9
4	1,3,5,9	9	1,3,4,8
5	1,4,6+		

Choices	Element selected	Reason	Partial result
All	1	Random	[1]
2,5,4,9	5	Shortest list	[1 5]
4,6	6	Common edge	[1 5 6]
2,7	2	Random choice (both have two items in list)	[1 5 6 2]
3,8	8	Shortest list	[1 5 6 2 8]
7,9	7	Common edge	[1 5 6 2 8 7]
3	3	Only item in list	[1 5 6 2 8 7 3]
4,9	9	Random choice	[1 5 6 2 8 7 3 9]
4	4	Last element	[1 5 6 2 8 7 3 9 4]

Obrázek 3.5: Edge recombination 2 (ER2): příklad pro rodiče (123456789) a (937826514)

Mutace jsou obecně jednodušší (pro všechny reprezentace), používá se:

- inverze podcesty
- posun podcesty
- záměna 2 měst
- záměna podcest
- složitější heuristiky: např. **2-OPT**. Tento „chytrý operátor“ funguje tak, že se vyberou dvě hrany, ty se odeberou a nahradí jinými dvěma hranami (je jen jedna možnost). Nakonec se vybere kratší z obou vzniklých cest. Vlastně se jedná o mutaci, která vezme část cesty a tu vloží do jedince v opačném pořadí. 2-OPT se navíc ještě podívá, jestli je výsledek kratší a přijme ho jen pokud je.
- 2-OPT se přirozeně dá zobecnit na **k-OPT**, odstraní se  $k$  hran a vzniklé fragmenty se pospojují tak, aby výsledná cesta byla co možná nejkratší.

Úspěšnost evoluce závisí také na inicializaci. Nejpoužívanější jsou dvě varianty:

**nejbližší sousedi** Začni s náhodným městem, postupně vybírej jako další to nejbližší z dosud nevybraných.

**vkládání hran** Budujeme cestu  $T$ , na začátku náhodná hrana. K cestě  $T$  vyber nejbližší město  $c$  mimo  $T$ . Najdi hranu  $k-j$  v  $T$  tak, že minimalizuje rozdíl mezi  $k-c-j$  a  $k-j$ . Vyhod'  $k-j$ , vlož  $k-c-j$  do  $T$ .

### 3.6.4 Vícekriteriální optimalizace (MOEA – Multi-objective EA)

Velmi často potřebujeme optimalizovat více funkcí současně, tj. místo jedné fitness jich máme celý vektor  $\vec{f} = (f_1, f_2, \dots, f_n)$ . Ty si často odporují, nelze tedy najít řešení optimální pro všechny zároveň.

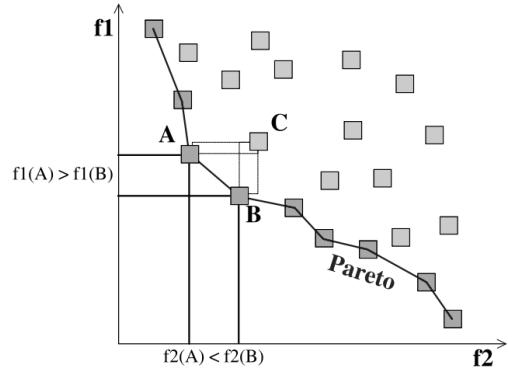
Řekneme, že řešení  $x$  **slabě dominuje** řešení  $y$ , jestliže  $x$  je ve všech kritériích aspoň tak dobré jako  $y$ , tj.  $f_i(x) \leq f_i(y) \quad \forall i \in \{1, \dots, n\}$ .

Řekneme, že  $x$  **dominuje**  $y$ , když jej slabě dominuje a navíc je aspoň v jedno kritériu lepší, tj.  $\exists j : f_j(x) < f_j(y)$ .

Řešení  $x$  a  $y$  jsou **nesrovnatelná**, jestliže  $x$  nedominuje  $y$  a  $y$  nedominuje  $x$ .

**Pareto-optimální řešení** (Paretova fronta, Pareto-optimální fronta) je množina jedinců, kterí nejsou dominováni žádným jiným jedincem. Je často nekonečná, hledáme konečnou approximaci.

V případě MOEA není jeden nejlepší jedinec v populaci, používá se celá populace a považuje se za aktuální approximaci Pareto-optimální množiny řešení.



#### 3.6.4.1 Algoritmy

Hlavní rozdíl mezi jednokriteriálními a vícekriteriálními algoritmy je v selekci (operátory mohou klidně zůstat stejné). S čímž samozřejmě souvisí způsob, jakým se počítá fitness. Celkově je potřeba zajistit konvergenci k Pareto optimální frontě a zároveň zajistit i dobré pokrytí této fronty.

**Agregace fitness** Nejjednodušší přístup. Definujeme agregovanou fitness  $f$  jako váženou sumu všech  $f_i$  a problém řešíme jako klasickou jednokriteriální optimalizaci. Problémem je nastavení vah u jednotlivých  $f_i$ .

**VEGA (Vector Evaluated GA)** Populaci  $N$  jedinců utřídíme podle každé z  $n$  fitness funkcí, poté pro každé  $i$  vyberu  $\frac{N}{n}$  nejlepších jedinců dle  $f_i$ . Na ty poté aplikují křížení, mutaci a provedu selekci do další generace.

Nevýhodou tohoto přístupu je obtížné udržování diverzity populace a často dochází ke konvergenci k extrémům jednotlivých  $f_i$ .

**NSGA (Non-dominated sorting GA)** Fitness se počítá pomocí dominance, diverzitu populace zaručuje **niching**.

Populace  $P$  je rozdělena na postupně konstruované fronty  $F_1, F_2, \dots$ , a to takto:

- $F_1$  je množina všech nedominovaných jedinců z  $P$
- $F_2$  je množina všech nedominovaných jedinců z  $P - F_1$
- $F_3$  je množina všech nedominovaných jedinců z  $P - (F_1 \cup F_2)$
- ...

Pro každého jedince  $i$  ve frontě  $F_k$  se spočte jeho *niching faktor* jako

$$sh(i) = \sum_{j \in F_k} sh(i, j)$$

kde

$$sh(i, j) = \begin{cases} 1 - \left( \frac{d(i, j)}{dshare} \right)^2 & \text{pro } d(i, j) < dshare \\ 0 & \text{jinak} \end{cases}$$

kde  $d(i, j)$  je vzdálenost  $i$  a  $j$ ,  $dshare$  je předem daný parametr. Niching faktor tedy vyjadřuje, jak blízko jsou další jedinci v populaci. Čím blíže 0, tím osamocenější jedinec je, naopak hodnota blízká 1 znamená mnoho blízkých sousedů.

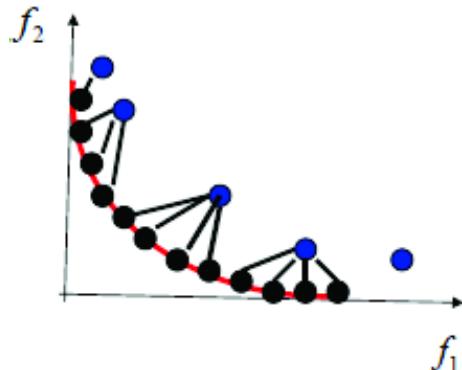
Jedinci z  $F_1$  dostanou nějakou „dummy fitness“ dělenou jejich niching faktorem. Jedinci z  $F_2$  dostanou jinou dummy fitness, která je menší než fitness nejhoršího jedince z  $F_1$  a opět se dělí jejich niching faktorem; atd.

**NSGA II** Odstraňuje nutnost určení *dshare* – nahrazuje ji **crowding distance**, což je součet přes všechna kritéria, vzdálenosti nejbližšího horšího (v daném kritériu) a nejbližšího lepšího jedince. Fitness se pak ve skutečnosti nepočítá, namísto toho se v turnajové selekci porovnávají jedinci nejprve podle čísla fronty (menší je lepší), v případě remízy podle crowding distance (větší je lepší – jedinci jsou v řídce osídlené oblasti a tedy lépe přispívají k pokrytí fronty). Dále zavádí elitismus: stará a nová generace jsou spojeny, utříděny a z nich se pak vybírá lepší část do další populace. Postupně se kopírují fronty (od nejlepší) dokud se vejdu do populace, z fronty, která se již nevezde celá, se vybírá podle turnajové metody popsáne výše.

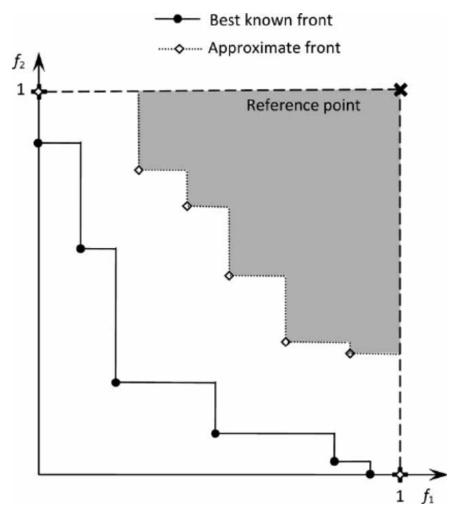
### 3.6.4.2 Porovnání algoritmů

Pro porovnání kvality řešení dvou různých algoritmů je třeba vyhodnotit, která z výsledných populací lépe approximuje Pareto optimální množinu.

**IGD (Inverted Generational Distance)** IGD měří průměrnou vzdálenost každého řešení na *skutečné* Pareto optimální frontě k nejbližšímu řešení ve výsledné množině. Díky tomu indikátor vyjadřuje jak blízkost řešení k Pareto-optimální frontě (convergence), tak to, jestli jsou řešení po frontě rozmístěna rovnoměrně (spread). Menší hodnoty tohoto indikátoru jsou lepší.



**Hyperobjem (HV – hypervolume)** Hyperobjem vyjadřuje objem prostoru dominovaného danou množinou (a omezeného zvoleným referenčním bodem). V případě, že máme jen dvě funkce, které obě najednou minimalizujeme, tak se jedná o plochu „nad“ body, které jsou obrazy jedinců v populaci. Plocha je shora omezena referenčním bodem (jinak by byla nekonečná). Tento indikátor opět spojuje jak konvergenci, tak pokrytí Pareto-optimální fronty. Větší hodnoty tohoto indikátoru jsou lepší. Někdy se také jako indikátor používá rozdíl hyperobjemu *skutečné* Pareto optimální fronty a approximace nalezené algoritmem. Potom jsou samozřejmě lepší menší hodnoty.



# Část III

## Strojové učení

# Kapitola 4

## Strojové učení a jeho aplikace

- 4.1 Strojové učení; prohledávání prostoru verzí, učení s učitelem a bez učitele, pravděpodobnostní přístupy, teoretické aspekty strojového učení.
- 4.2 Evoluční algoritmy; základní pojmy a teoretické poznatky, hypotéza o stavebních blocích, koevoluce, aplikace evolučních algoritmů.

Skip.

- 4.3 Strojové učení v počítačové lingvistice a algoritmy pro statistický parsing.

Od 80. a 90. let se ve **zpracování přirozených jazyků (NLP, Natural Language Processing)** začaly silně uplatňovat metody strojového učení. Dříve se při úkolech spojených se zpracováním jazyků spoléhalo na ručně zakódovaná pravidla. Ta se nyní získávají pomocí **statistické inference** z velkých korpusů textů, ideálně získaných z reálných zdrojů (tj. ne umělé).

Mnoho metod strojového učení již bylo adaptováno pro NLP. Rozhodovací stromy generují systém rozhodovacích if-then pravidel, na která byla tehdy komunita zvyklá. Postupně se však na výsluní dostaly **statistické modely**, které pracují s pravděpodobnostmi, váhami na vstupních příznacích a s mírou nejistoty.

Využití metod strojového učení má mnoho výhod:

- Statistické učení se automaticky zaměřuje na **nejčastější** (a tedy důležité) případy.
- Budované modely jsou typicky **robustní** vůči novým **neznámémým datům** a **zašuměnému či chybnému vstupu**. Toho je u ručně psaných pravidel velmi obtížné dosáhnout.
- Modely lze zpřesňovat poskytnutím dodatečných trénovacích dat. U ručně psaných pravidel to lze pouze zvyšováním komplexity modelu.

Strojové učení a NLP se vlastně dobře doplňují. NLP samotné nedokáže text zpracovat o moc lépe, než jako „bag-of-words“, kdežto NLP má prostředky pro extrakci struktury (morphologie, syntax, sémantika) z textu. Naopak NLP nedokáže s těchto poznatků získat nějakou hlubší informaci, kdežto strojové učení na to má spoustu nástrojů. Nabízí se tedy výhodná spolupráce: NLP extrahuje sofistikované informace o struktuře, které pak strojové učení použije jako příznaky pro učení nejakého modelu.

---

*Následující sekce jsou výtahem z přednášek NPFL067 a NPFL068*

### 4.3.1 Základy teorie informace

Nechť  $p_X(x)$  je pravděpodobnostní rozdělení náhodné proměnné  $X$  a nechť  $\Omega$  je množina základních jevů. Pak **entropie** tohoto rozdělení je

$$H(X) = - \sum_{x \in \Omega} p(x) \cdot \log_2(p(x))$$

Entropie vyjadřuje *míru nejistoty*. Vyjadřuje se v bitech. Nejnižší možná hodnota je 0 (pokud je výsledek experimentu dán předem), horní limit obecně není – nicméně největší nejistota odpovídá uniformnímu rozdělení, tj. pro  $|\Omega| = n : H(X) \leq \log_2(n)$ . Entropii lze taktéž interpretovat jako minimální počet bitů nutných pro zakódování nějaké informace: čím předvídatelnější jsou data, tím úspornější kódování.

**Podmíněnou entropii** náhodných proměnných  $X$  (prostor  $\Omega$ ) a  $Y$  (prostor  $\Psi$ ) definujeme jako:

$$H(Y|X) = - \sum_{x \in \Omega} \sum_{y \in \Psi} p(x, y) \cdot \log_2 p(y|x)$$

Důležité: první pravděpodobnost **není** podmíněná! Jedná se z jistého pohledu o vážený průměr, kde  $p(x, y)$  je váha a ta není podmíněná.

Podmíněná entropie je vždy lepší, tj.  $H(Y|X) \leq H(Y)$ . Tedy vždy se nám hodí mít nějakou znalost podmiňující odhadovanou proměnnou (příklad: chceme-li odhadnout další slovo ve větě, nás odhad jistě zlepší znalost slovníku daného jazyka včetně distribuce slov).

Abychom změřili, nakolik naše znalost  $X$  pomůže odhadu  $Y$ , můžeme použít míru **vzájemné informace (MI, Mutual Information)**:

$$I(X, Y) = \sum_{x \in \Omega} \sum_{y \in \Psi} p(x, y) \cdot \log_2 \frac{p(x, y)}{p(x)p(y)}$$

Ta se měří rovněž v bitech. Vyjadřuje, o kolik bitů znalost  $X$  sníží entropii  $Y$ .

Posledním důležitým nástrojem z teorie informace je **cross-entropie**. Ta vyjadřuje, nakolik naše pozorování odpovídají očekávanému pravděpodobnostnímu rozdělení. Nechť  $p$  označuje očekávané rozdělení (získané například analýzou slovníku) a  $p'$  označuje odhad pravděpodobností na základě pozorovaných dat (typicky  $p'(x) =$ počet pozorování  $x$  děleno celkový počet pozorování):

$$H_p(p') = - \sum_{x \in \Omega} p(x) \cdot \log_2(p'(x))$$

Cross-entropie slouží nikoliv k ohodnocení či porovnání pozorovaných dat, ale k **porovnání různých distribucí**. Máme-li dvě distribuce  $p$  a  $q$ , pak porovnáním jejich cross-entropií na reálných datech (tj.  $H_p(p')$  vs.  $H_q(p')$ ) zjistíme, které z nich lépe vystihuje reálná data – tj. má nižší cross-entropii (např. máme 2 slovníky pro nějaký jazyk včetně pravděpodobností výskytů jednotlivých slov; na základě reálného vzorku nějakého textu můžeme rozhodnout, který slovník jej odhaduje lépe).

I cross-entropie může být **podmíněná**:

$$H_p(p') = - \sum_{x \in \Omega} \sum_{y \in \Psi} p(x, y) \cdot \log_2 p'(y|x)$$

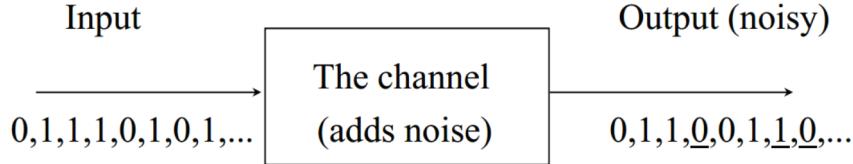
Často se vyplácí sčítat nikoliv přes celý prostor jevů, ale pouze přes **slovníková data**  $T$ :

$$H_p(p') = - \frac{1}{|T|} \sum_{i=1 \dots |T|} \log_2 p'(y_i|x_i)$$

### 4.3.2 Modelování jazyka, zašuměný kanál

**Zašuměný kanál** je označení pro modela média, které k přenášeným slovům přidá nějaký šum. Formálně: do kanálu vstupuje nějaká sekvence symbolů, z něj pak vystupuje pozměněná,

zašuměná sekvence, kterou my přijmeme. Cílem je tedy na základě přijaté zašuměné sekvence uhodnout původní zprávu.



Mnoho problémů NLP lze chápat jako případ zašuměného kanálu, např. rozpoznávání mluvené řeči, OCR, strojový překlad („šum“ kvůli překladu), ...

Formálně nás tedy zajímá podmíněná pravděpodobnost původní zprávy  $A$  za předpokladu přijaté zprávy  $B$ , tedy  $p(A|B)$ . Díky Bayesově větě:

$$p(A|B) = p(B|A)p(A)/p(B)$$

Zajímá nás nejpravděpodobnější možnost, tj.

$$A_{best} = \operatorname{argmax}_A p(B|A)p(A)/p(B) = \operatorname{argmax}_A p(B|A)p(A)$$

kde  $p(B)$  lze ignorovat, protože je pro všechny možnosti  $A$  stejná. Náš výpočet má tedy nyní dvě složky:

- $p(B|A)$ : akustický/překladový/... model (záleží na konkrétním problému)
- $p(A)$ : **jazykový model**

Z hlediska ručně kódovaných pravidel je přirozeným popisem jazyka nějaká *gramatika*. Z hlediska statistického zpracování je základem statistický **jazykový model** (LM, Language Model). Ten tvorí **pravděpodobnostní distribuce přes sekvence slov**, tj. každé sekvenci  $W = (w_1, w_2, \dots, w_k)$  přiřadí pravděpodobnost  $p(W) = p(w_1, w_2, \dots, w_k)$ . To můžeme jednoduše přeformulovat díky Bayesovu pravidlu a řetízkovému pravidlu na

$$p(W) = p(w_1, w_2, \dots, w_k) = p(w_1) \times p(w_2|w_1) \times p(w_3|w_1, w_2) \times \dots \times p(w_k|w_1, w_2, \dots, w_{k-1})$$

Tato přímočará formulace je ale nepraktická kvůli vysokému počtu parametrů i pro relativně krátká  $W$ . Nejčastější v praxi používanou variantou je tzv. **n-gram Language Model**, kdy je „paměť“ omezena na posledních  $n$  slov, tj.

$$p(W) = p(w_1, w_2, \dots, w_k) = \prod_{i=1}^k p(w_i|w_{i-n+1}, w_{i-n+2}, \dots, w_{i-1})$$

Pro každé slovo tedy spočteme jeho pravděpodobnost za **předpokladu předchozích  $n$  slov**, celková pravděpodobnost sekvence je pak **součinem pravděpodobností jednotlivých slov**.

Příklady pro malá  $n$ :

- 0-gram LM: uniformní model**  $p(w) = \frac{1}{|V|}$  (pro slovník velikosti  $|V|$ )
- 1-gram LM: unigram model**  $p(w)$
- 2-gram LM: bigram model**  $p(w_i|w_{i-1})$
- 3-gram LM: trigram model**  $p(w_i|w_{i-2}, w_{i-1})$

Čím vyšší  $n$ , tím přesnější model, nicméně za cenu nárůstu složitosti. Používá se  $n = 3$  (**trigram model**).

Tento model odpovídá **Markovovu procesu (řetězci)** rádu  $n-1$ . Markovův řetězec (1. rádu) je jakákoli posloupnost (stavů, hodnot, ...), která splňuje 2 vlastnosti:

1. **omezená historie:**  $\forall i \in 1 \dots T : P(X_i|X_1, X_2, \dots, X_{i-1}) = P(X_i|X_{i-1})$  Tj. aktuální hodnota je podmíněna pouze předchozí hodnotou.
2. **stacionarita:**  $\forall i \in 1 \dots T, \forall x, y \in S : P(X_i = x|Y_i = y) = p(x|y)$  Tj. pravděpodobnost přechodu z  $x$  do  $y$  je stejná nezávisle na pozici v sekvenci.

Díky stacionaritě se Markovovy řetězce jednoduše reprezentují pomocí *stavových diagramů*. Definici lze také jednoduše rozšířit na vyšší řády. Pak jsou stavy v diagramu odpovídající  $n$ -tice.

Jazykový model se získává z trénovacích dat (preprocessing: sjednocení velikosti písmen, anotace konců vět, ...), a to strategií **Maximum Likelihood Estimate (MLE)** (maximálně věrohodný odhad), který je založen na počtech výskytů jednotlivých  $n$ -gramů v trénovacích datech. Trigramový model pak získáme jako

$$p(w_i|w_{i-2}, w_{i-1}) = \frac{c_3(w_{i-2}, w_{i-1}, w_i)}{c_2(w_{i-1}, w_i)}$$

kde  $c_2$  resp.  $c_3$  označuje počet výskytů daného bigramu resp. trigramu v trénovacích datech. Tato metoda nefunguje na začátku trénovacích dat, pro se často přidávají 2 „dummy“ slova před začátek trénovacích dat.

### 4.3.3 Vyhazování jazykového modelu

Problémem popsaného modelu jsou nulové pravděpodobnosti  $n$ -gramů, které se v trénovacích datech vůbec nevyskytují. To je samozřejmě nezádoucí (na nových datech pak může vyjít nekonečná cross-entropie) a řeší se to tzv. **vyhlazováním** (angl. *LM smoothing*). Existuje více strategií, ale obecná idea je stejná: ubrat nějaký nenulový  $p(w)$  a distribuovat mezi nulové  $p(w)$ , čímž získáme nové, vyhlazené rozdělení  $p'(w)$ . Vždy však musíme dbát na to, aby součet pravděpodobností byl 1.

Strategie:

**přidání jedničky:** K počtu výskytů každého ze všech *možných* slov ze slovníku  $V$  přičteme jedničku:

$$p'(w) = \frac{c(w) + 1}{|T| + |V|}$$

kde  $T$  je trénovací množina (a původní výpočet by tedy vypadal  $p(w) = c(w)/|T|$ )

V případě  $n$ -gramů pro  $n > 1$  vypadá vzorec takto:  $p'(w) = \frac{c(h,w)+1}{c(h)+|V|}$ , kde  $h$  označuje „historii“, tj. předchozích  $n$  slov.

Jednoduché, ale v praxi nepříliš použitelné.

**přidání  $\lambda < 1$ :** Tentýž princip, pouze místo 1 přidáváme nějakou menší konstantu. Myšlenkou je opravit ten nedostatek, že přidání byť jen 1 výskytu může pro malá trénovací data velmi silně pokřivit rozdělení.

$$p'(w) = \frac{c(w) + \lambda}{|T| + \lambda|V|} \quad \text{resp.} \quad p'(w) = \frac{c(h,w) + \lambda}{c(h) + \lambda|V|}$$

**Good-Turingovo vyhlazování:**

$$p'(w) = \frac{(c(w) + 1) \cdot N(c(w) + 1)}{|T| \cdot N(c(w))}$$

kde  $N(c)$  je počet slov s počtem výskytů rovným  $c$  („počet počtů“). Speciálně pro  $c(w) = 0$  je  $p'(w) = \frac{N(1)}{|T| \cdot N(0)}$ . Tedy pro slova s nulovým výskytem si „vypůjčíme“ od slov s 1 výskytem. Pro vyšší hodnoty se pak „masa pravděpodobnosti“ rozdistribuuje vždy směrem k „potřebnějším“.

**lineární interpolace:** *Pozorování:* čím nižší řad  $n$ -gramů, tím méně detailní distribuce a pro 0-gramy dokonce dostáváme uniformní rozdělení. To má sice nízkou informaci, ale na druhou stranu nemá problém s nulovými hodnotami.

*Idea:* zkombinovat různé řady  $n$ -gramů:

$$p'(w_i|w_{i-2}, w_{i-1}) = \lambda_3 p_3(w_i|w_{i-2}, w_{i-1}) + \lambda_2 p_2(w_i|w_{i-1}) + \lambda_1 p_1(w_i) + \lambda_0 \frac{1}{|V|}$$

kde  $\lambda = (\lambda_0, \lambda_1, \lambda_2, \lambda_3)$  jsou parametry ovlivňující detailnost vs. vyhlazenost výsledné distribuce. Platí  $\forall i \lambda_i > 0, \sum_i \lambda_i = 1$ . Tyto parametry se odhadnou EM algoritmem

(detaily vynecháme) pomocí speciálně vyčleněného datasetu (*heldout data*), na němž se budeme snažit minimalizovat cross-entropii pro fixní  $p_1, p_2, p_3$ . Je nutné použít jiná data, než trénovací (ze kterých jsme získali  $p_1, p_2, p_3$ ), jinak by došlo k degeneraci modelu k  $\lambda = (0, 0, 0, 1)$ . Obecně se v NLP rozdělují dostupná data na 3 části: *trénovací*, *testovací* a *heldout data*.

#### 4.3.4 Třídy slov

Další strategií pro korekci chyby distribuce  $p(w)$  způsobené příliš malým trénovacím datasetem (který nikdy nebude dost velký :)) je užití **slovních tříd**.

Uvažme následující situaci: v trénovacích datech se setkáváme s frázemi *short homework*, *simple homework*, *short assignment*; ale nikoliv se *simple assignment*. Pak tedy  $p(\text{homework}|\text{simple})$  je vysoká, ale  $p(\text{assignment}|\text{simple})$  je velmi malá (není nulová vlastně jen díky vyhlazování).

Pokud však seskupíme slova s podobným významem do společné třídy, tj. *(simple, short)(homework, assignment)*, můžeme náš  $n$ -gramový model vybudovat na úrovni tříd a tím dosáhnout lepšího odhadu pro  $n$ -gramy, s nimiž jsme se nesetkali, ale které významově dávají smysl. Cenou za práci na úrovni tříd namísto slov je jistá ztráta informace.

Nová podoba  $n$ -gramového jazykového modelu je tedy následující:

$$p_n(w_i|h_i) = \frac{p(w_i|c_i)}{p_n(c_i|h_i)}$$

kde  $c_i$  označuje **třídu** slova  $w_i$ ,  $p(w_i|c_i)$  popisuje pravděpodobnostní distribuci slov v této třídě a  $h_i$  je historie slova  $w_i$ , avšak tentokrát jako **předchozích  $n$  tříd** (resp. **třídy předchozích  $n$  slov**)! Např. pro trigramový model je  $h_i = c_{i-2}, c_{i-1}$ , nikoliv  $w_{i-2}, w_{i-1}$ !

I pro třídy lze použít lineárně vyhlazený model.

Rozdělení slov do tříd můžeme získat třeba z nějakého slovníku, nicméně jak je ve strojovém učení zvykem, raději ho sestrojíme z dostupných dat. Algoritmus je následující:

Nechť  $V$  je množina slov (=slovník),  $C$  je množina tříd a  $r : V \mapsto C$  je mapování slov do tříd.

1. Na začátku je každé slovo ve vlastní třídě, tj.  $V = C, r = \text{id}$
2. Vyber dvě třídy  $k, l$  takové, že

$$(k, l) = \operatorname{argmax}_{k,l} I_{\text{merge}(r,k,l)}(D, E)$$

kde  $I(D, E)$  vyjadřuje *vzájemnou informaci* pro sousední třídy, tj.  $D, E$  jsou náhodné proměnné takové, že  $d \in D, e \in E$  jsou třídy z  $C$ , které se v trénovacích datech vyskytují vedle sebe.  $I_{\text{merge}(r,k,l)}(D, E)$  vyjadřuje tuto hodnotu po hypotetickém sloučení tříd  $k, l$ .

3. sluč třídy  $k, l$ , přitom uprav  $C$  a  $r$
4. opakuj kroky 2 a 3 dokud  $|C|$  nedosáhne předem dané velikosti

Pozor,  $k, l$  typicky nejsou sousední třídy, kterých se týká  $I(D, E)$ ! Optimálně to jsou třídy slov, která se vyskytují v podobném kontextu a proto jejich sloučení zhorší globální  $I(D, E)$  co nejméně. Optimální  $I(D, E)$  samozřejmě nastává v případě, že je každé slovo ve vlastní třídě, proto je důležité slučování vynucovat v každém kroku.

Důležitá je podmínka na ukončení při dosažení určitého počtu tříd (jinak bychom se dostali na jedinou třídu).

Pro sestrojení tříd se používají *trénovací* data.

#### 4.3.5 Markovovy modely

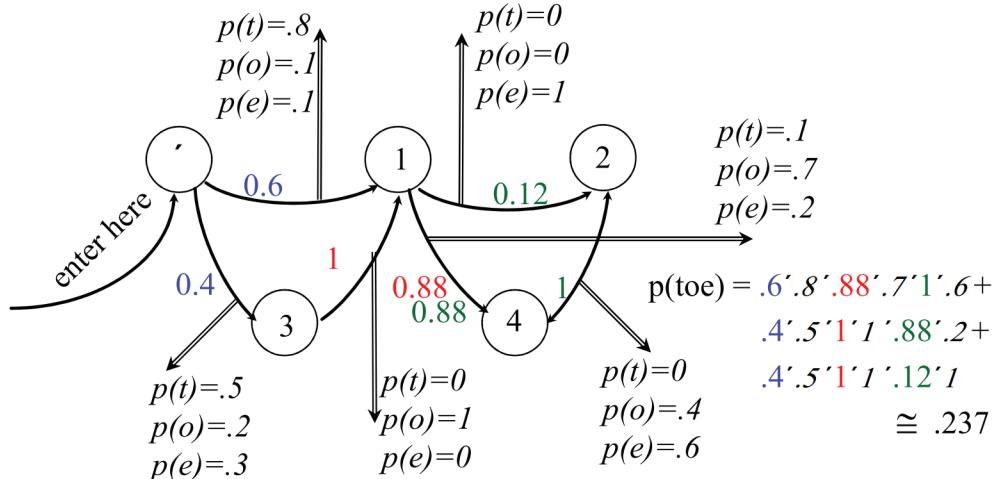
O jednoduchých Markovových procesech jsme se již zmínili u  $n$ -gramového jazykového modelu. Nyní definujeme složitější model, který se používá hned v několika odvětvích NLP.

**Skrytý Markovův model (HMM, Hidden Markov Model)** je pětice  $(S, s_0, Y, P_S, P_Y)$ , kde

- $S = \{s_0, s_1, \dots, s_T\}$  je **množina stavů**

- $s_0$  je počáteční stav
- $Y = \{y_1, y_2, \dots, y_V\}$  je výstupní abeceda
- $P_S(s_j|s_i)$  jsou pravděpodobnosti přechodu mezi stavů. Velikost  $|P_S| = |S|^2$
- $P_Y(y_k|s_i, s_j)$  jsou výstupní pravděpodobnosti. Velikost  $|P_Y| = |S|^2 \times |Y|$

Výstup se v této nejobecnější verzi generuje na přechodových hranách, časté je ale také generování výstupu ze stavů. Výstupní pravděpodobnosti udávají pro každou hranu pravděpodobnost emise daného symbolu.



### Úkoly pro HMM

- Generuj výstup. (Nepříliš hodnotné...)

Začni v  $s_0$ , pohybuj se náhodně podle  $P_S$ , vypisuj náhodně podle  $P_Y$ , opakuj dokud tě někdo nezastaví.

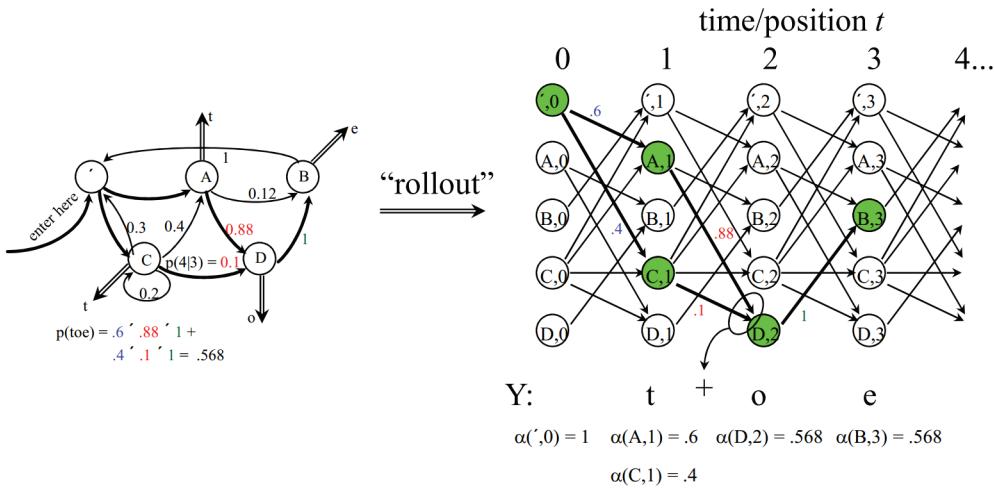
- Dána sekvence  $Y = y_1 y_2 \dots y_k$ , spočti její pravděpodobnost.<sup>1</sup>

Udržujeme si *trellis stav*, tj. stav automatu v čase  $t$ , každý stav drží proměnnou  $\alpha$ . Pravděpodobnost  $Y$  je pak dána jako  $\sum \alpha$  předchozích stavů.

1. iniciální trellis sloupec  $t_0$ : obsahuje jenom  $s_0$ , nastav  $\alpha(s_0, 0) = 1$
2. na základě sloupce  $t_{i-1}$  a aktuálního vstupního symbolu  $y_i$  vytvoř následující sloupec trellis grafu, ale vyber pouze stavů, do nichž se dá dostat z aktuálního sloupce a které generují  $y_1$  (resp. takové, že přechodová hrana generuje  $y_1$ )
3. nastav  $\alpha(state, i) = \sum_{s \in t_{i-1}, \exists \text{hrana } s \rightarrow state} P_S(state|s) \cdot \alpha(s, i-1) \cdot P_Y(y_i|s, state)$
4. jakmile  $i = |Y|$  (vstup je vyčerpán), nastav  $P(Y) = \sum_{s \in t_{|Y|}} \alpha(s, |Y|)$

---

<sup>1</sup>Občas nesprávně označován jako Trellisův algoritmus, ale „trellis“ je pouze označení jistého typu grafu, kde jsou vrcholy rozděleny do sloupců odpovídajících časům  $t, t+1, \dots$



Obrázek 4.1: Příklad „trellis“ grafu pro vstup  $Y = \text{„toe“}$  (varianta s deterministickými výstupy z vrcholů)

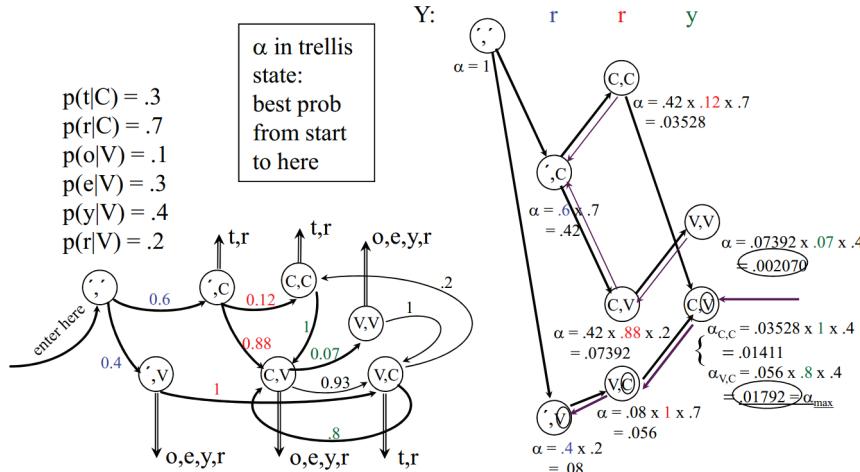
- Dána sekvence  $Y = y_1 y_2 \dots y_k$ , spočti nejpravděpodobnější posloupnost stavů, která ji vygenerovala (popř.  $n$  nejpravděpodobnějších posloupností).

Požívá se **Viterbiho algoritmus**. Princip je podobný jako v předchozím úkolu, pouze v průchodu trellis grafem nebereme součet  $\alpha$  pro různé hrany vedoucí do téhož vrcholu, nýbrž **bereme pouze přechod s maximální  $\alpha$  hodnotou**. Současně si pamatujeme zpětné hrany, abychom na konci mohli cestu zrekonstruovat.

Ve třetím kroku původního algoritmu bychom tedy použili výpočet

$$\alpha(state, i) = \max_{s \in t_{i-1}, \exists \text{hrana } s \rightarrow state} (P_S(state|s) \cdot \alpha(s, i-1) \cdot P_Y(y_i|s, state))$$

Pokud chceme  $n$  nejlepších cest, budeme si vždy pamatovat  $n$  nejlepších zpětných uka-zatelů a cesty poté zrekonstruujeme pomocí backtracking algoritmu.

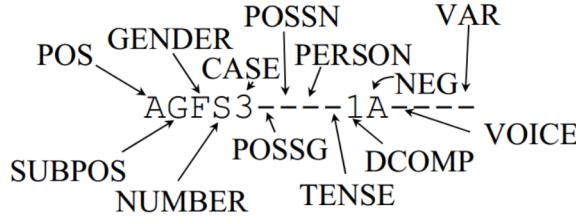


Obrázek 4.2: Příklad výpočtu Viterbiho algoritmu.

- Dána trénovací data  $Y$ , odhadni (tj. nauč se) parametry HMM (tj.  $P_S$  a  $P_Y$ ). **Baum-Welch algoritmus**. Založen na EM.

#### 4.3.6 Tagging

**Tagování** je proces, kdy se snažíme slovo rozložit na *lemma* (=základní tvar slova) a **tag**, tj. „slovňkovou značku“, která obsahuje mluvnické kategorie, jako například číslo, pád, rod, druh, čas apod. Různé korpusy a treebanky používají různé formáty, české jsou výrazně složitější.



Obrázek 4.3: Ukázka standardní české 15-písmenné značky (tagu) včetně vysvětlivek

Formálně je tagování zobrazení  $A^+ \rightarrow T$ , kde  $A$  je abeceda fonémů a  $T$  je množina tagů, tzv. **tagset**. S tagováním úzce souvisí *morfologická analýza*, což je formálně zobrazení  $A^+ \rightarrow 2^{L, C_1, C_2, \dots, C_n}$ , kde  $L$  je lemma a  $C_i$  jsou mluvnické kategorie. Morfologická analýza tedy přiřadí neprázdné sekvenči fonémů (=slovu) množinu možných lemmat spolu s mluvnickými kategoriemi (např. pro slovo „kolem“ naleze morfologická analýza několik možností). Tagování lze pak chápat jako *disambiguation*, tj. výběr jedné z možností (typicky na základě kontextu). Tag  $t \in T$  je pak typicky přímo množina mluvnických kategorií  $t = C_1 C_2 \dots C_m$ .

Různé přístupy k tagování:

1. **ručně psaná pravidla** Mějme následující výstup z morfologické analýzy (Penn tagset):

I	<b>watch</b>	a	<b>fly</b>	.
	NN	NN	DT NN	.
	PRP	VB	NN VB	
	VBP		VBP	

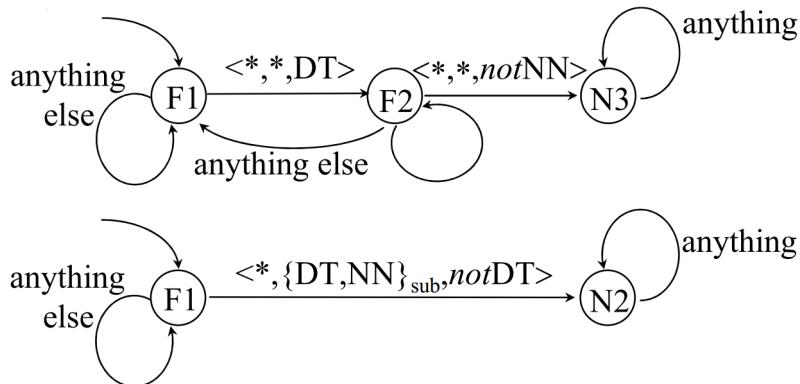
Cílem je vybrat správnou sekvenči značek. Používáme **předdefinovaná pravidla if-then**, například:

```

if ( $t_{i-1} = DT$ ) then ( $t_i = NN$ )
if ( $((t_{i-1} = PRP) \wedge (t_{i+1} = DT))$ ) then ( $t_i = NN$ )
if ( $(t_i \in \{DT, NN\})$ ) then ( $t_i = DT$ )
if ( $(t_{i+1} \in \{VB, VBZ, VBP, VBD, VBG\})$ ) then ( $t_i \neq DT$ )
...

```

Pro každé pravidlo sestrojíme vlastní **konečný automat**, který se po skončení výpočtu nachází v přijímacím stavu, právě tehdy, pokud není pravidlo porušeno. Automaty spouštíme paralelně a zkoušíme všechny možné sekvenči tagů. Pokud všechny přijímají, jde o validní otagování.



## 2. statistické metody

- (a) „pravděpodobnostní“
  - i. HMM
  - ii. maximum entropy
- (b) pravidlové

- i. TBEDL
- ii. Example based
- (c) feature-based
- (d) kombinace klasifikátorů

#### 4.3.7 Statistický parsing

**Parsing (parsování)** je označení pro proces, kdy je pro vstupní větu, o níž předpokládáme, že byla vytvořena nějakou gramatikou, nalezena posloupnost pravidel této gramatiky, kterou byla věta nejpravděpodobněji vytvořena. Typicky se pro větu rovnou sestrojí *parsovací strom*.

#### 4.3.8 Statistický strojový překlad

Klasickým přístupem je chápout strojový překlad jako problém **zašuměného kanálu**, kdy například při překladu z francouzštiny do angličtiny je nezkresleným vstupem anglická věta  $E$  a „zašuměným výstupem“ je francouzská věta  $F$ . Naším cílem je pak nalézt nejpravděpodobnější původní řetězec, tj. nejpravděpodobnější překlad do EN. Formálně  $P(E|F) = P(F|E) \cdot P(E)/P(F)$ , nejpravděpodobnější překlad je

$$\text{argmax}_E = \text{argmax}_E P(E|F) = P(F|E) \cdot P(E)$$

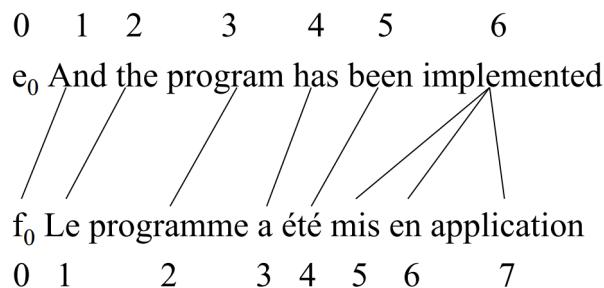
Opět vynecháme člen  $P(F)$ , jelikož ten je pro všechna  $E$  stejný.

Potřebujeme tedy **language model (LM)**  $P(E)$ , **translation model (TM)**  $P(F|E)$  a vyhledávací proceduru, která k  $E$  našle nejlepší  $F$  pomocí LM a TM.

**Language Model** Lze použít v podstatě jakýkoliv, např. (smoothed) 3-gram LM, 3-gram s třídami, ... Jazykový model nemusí nutně operovat přímo se slovy/slovními tvary, může se hodit přidat úroveň abstrakce (viz níže).

**Translation Model** Nestará se o korektnost anglických vět, to je starost LM. To nám dává větší volnost.

Typicky nemáme mapování 1:1, věty bývají i jinak dlouhé. Proto se používá **Alignment (zarovnání)**:



Označíme  $m = |F|, l = |E|$ , pak máme celkem  $m \cdot l$  možných spojů mezi větami  $E$  a  $F$ ; jedno konkrétní zarovnání je pak nějaká podmnožina těchto spojů, tj. celkem  $2^{ml}$  různých zarovnání. V praxi to však bývá o něco lepší, např. při překladu EN → FR má každé anglické slovo 1...n spojů a každé francouzské *právě* jeden spoj. To je tedy celkem „pouze“  $(1+l)^m$  zarovnání.

Nechť celková pravděpodobnost anglické věty  $E$ , francouzské věty  $F$  a zarovnání  $A$  je  $P(F, A, E)$ . Pak můžeme vyjádřit  $P(F|E)$  jako

$$P(F|E) = \frac{P(F, E)}{P(E)} = \frac{\sum_A P(F, A, E)}{\sum_{A,F} P(F, A, E)} = \sum_A P(F, A|E)$$

Jedna z možných **dekompozic** pravděpodobnosti  $P(F, A|E)$ , kterou budeme používat, je tato:

$$P(F, A|E) = P(m|E) \cdot \prod_{j=1}^m P(a_j | a_l^{j-1}, f_l^{j-1}, m, E) \cdot P(f_j | a_l^j, f_l^{j-1}, m, E)$$

kde

- $m$  je délka francouzské věty
- $f_j$  je  $j$ -té francouzské slovo, tj.  $F[j]$
- $a_j$  je alignment  $j$ -tého francouzského slova  $f_j$  (tj. jeden spoj vedoucí z  $f_j$ )
- $a_l^j$  je posloupnost alignmentů  $a_i$  od začátku věty po  $f_j$  (včetně)
- $f_l^j$  je posloupnost francouzských slov od začátku věty po  $f_j$  (včetně)

Pomocí této dekompozice můžeme jednoduše generovat  $F$  a  $A$ :

1. nejprve vygenerujeme délku věty  $m$  na základě  $E$ ,
2. poté spoj  $a_1$  vedoucí z první pozice  $F$  (aniž bychom v tuto chvíli znali  $f_1$ )
3. teď díky spoji  $a_1$  (a anglickému slovu, do něž link vede) vygenerujeme francouzské slovo na první pozici,  $f_1$
4. pokračujeme další pozicí, dokud nezaplníme  $m$  pozic

Předchozí model má však stále příliš mnoho parametrů (podobně jako  $n$ -gramy s „neomezeným“  $n$ ), proto je potřeba approximace. Používá se 5 modelů rostoucí úrovně komplexity, přičemž parametry modelu 1 se použijí jako iniciální odhad parametrů model 2 atd.

**Model 1:** Zjednodušení:

- hodnota  $P(m|E)$  je konstantní (malé  $\varepsilon$ )
- distribuce zarovnání spojů závisí pouze na délce anglické věty  $l$ , tj. namísto  $P(a_j|a_l^{j-1}, f_l^{j-1}, m, E)$  použijeme  $1/(l+1)$
- distribuce francouzských slov  $f_j$  závisí pouze na odpovídajícím anglickém slově, tj.  $P(f_j|a_l^j, f_l^{j-1}, m, E)$  nahradíme  $p(f_j|e_{a_j})$

Distribuce Modelu 1 je tedy:

$$P(F, A|E) = \frac{\varepsilon}{(1+l)^m} \prod_{j=1}^m p(f_j|e_{a_j})$$

**Model 2:** Komplexnější  $P(a_j|\dots)$ , preference více „vertikálních“ spojů.

**Model 3:** Přidána „fertilita“, tj. počet spojů pro dané anglické slovo je explicitně modelován:  $P(n|e_i)$ . Pravděpodobnosti zarovnání z Modelu 2 nahradil model „distortion“.

**Model 4:** Myšlenka „distortion“ rozšířena na „word chunks“.

**Model 5:** Vylepšení Modelu 4.

**vyhledávací procedura** Chceme **dekodér**, který na základě „výstupu“ ( $F$ ) odhalí „vstup“ ( $E$ ). Překladový model ale funguje v opačném směru, tj.  $P(F|E)$ . Řešení (idea): generuj anglická slova po jednom pomocí LM, uchovej  $n$  nejlepších kandidátů podle TM. Také započítej délku kandidátů (anglických vět).

#### 4.3.8.1 Model ATG (Analysis, Translation, Generation)

Doposud jsme operovali na slovech (resp. slovních tvarech – *word forms*), nicméně těch je příliš mnoho a jakákoliv trénovací data budou příliš „řídká“ (sparse). Budeme tedy překlad provádět v těchto fázích:

1. **Analýza:** použijeme 4 kroky lingvistické analýzy:
  - tagging
  - lemmatizace
  - word-sense disambiguation: víceznačná slova jsou ve slovníku vícekrát s příponou, např. kolem-1, kolem-2, ... Správný význam se odvodí automaticky pomocí TM, který pracuje přímo na těchto příponových slovech, tj.  $p(kolem-1|\dots)$  vs.  $p(kolem-2|\dots)$
  - noun-phrase chunks
2. **Překlad:** Oba jazyky analyzujeme prostředky popsanými v předchozím kroku. Jak trénovací fáze, tak samotný ostrý překlad (test) bude pak probíhat na nikoliv na slovech (resp. slovních tvarech), nýbrž na výsledku analýzy (tj. na „chunks“, tags, ...). Jazykový model (EN) bude rovněž vyvinut na analyzovaném zdroji.

3. **Generování:** Opak analýzy. Z přeložených symbolů (chunks, tags, ...) vygenerujeme smysluplnou anglickou větu:
- chunks → lemmata spolu s významy (triviální)
  - lemmata spolu s významy → lemmata (triviální)
  - lemmata + tagy → slovní tvary

## 4.4 Pravděpodobnostní algoritmy pro analýzu biologických sekvencí; hledávání motivů v DNA, strategie pro detekci genů a predikci struktury proteinů.

**Bioinformatika** je vědní disciplína, která využívá výpočetní metody pro studium biologických dat (typicky velkého objemu), zejména dat molekulárně-biologických (tj. DNA a proteiny).

**Typické úkoly:** vyhledávání subsekvencí v datech, srovnávání podobných sekvencí, analýza sekvencí, snaha předpovídat strukturu a funkci proteinů, vizualizace, dynamické modelování

*Následující sekce jsou výtahem z přednášky Bioinformatické algoritmy - NTIN084*

### 4.4.1 Restrikční mapování

**Restrikční enzymy** sekají DNA na specifických místech, tzv. **restriction sites**. Vznikají segmenty různé délky, kterou lze změřit pomocí **gelové elektroforézy**. Naším cílem bude získat **restrikční mapu**, tj. mapu pozic restrikčních míst v DNA.

Formálně: Nechť  $S$  je DNA sekvence. **Fyzická (restrikční) mapa** je množina značek  $M$  a funkce  $p : M \mapsto \mathbb{N}$ , která přiřadí každé značce pozici v  $S$ .

Dále nechť  $X = \{x_1, \dots, x_n\}$  je množina pozic v  $S$ ,  $0 = x_1 < x_2 < \dots < x_n$ . Pak

- **partial digest** je multi-množina  $\delta X = \{x_j - x_i \mid 1 \leq i < j \leq n\}$  všech vzájemných vzdáleností všech dvojic v  $X$ .
- **full digest** je multi-množina  $\Delta X = \{(x_2 - x_1), (x_3 - x_2), \dots, (x_n - x_{n-1})\}$  vzdáleností všech dvojic sousedních prvků.

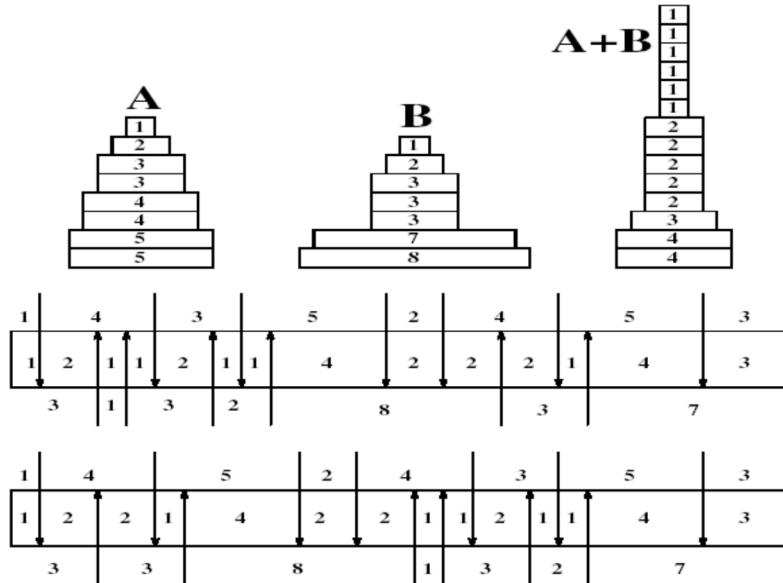
**Problém restrikčního mapování** je definován takto: máme (experimentálně získanou) podmnožinu  $E \subseteq \delta X$ , úkolem je zrekonstruovat  $X$ .

Jeden full digest nemusí stačit, můžeme dostat nejednoznačné řešení. V praxi se používají 3 různé přístupy a tedy definujeme 3 odpovídající problémy:

**DDP (Double Digest Problem)** Použijeme 2 enzymy  $A, B$  a získáme 3 full digesty

- $\Delta A$  – full digest using  $A$
- $\Delta B$  – full digest using  $B$
- $\Delta AB$  – full digest using both  $A$  and  $B$  at the same time

Bez  $\Delta AB$  je nemožné získat jednoznačné řešení. Nicméně i když máme  $\Delta AB$ , může řešení být nejednoznačné.

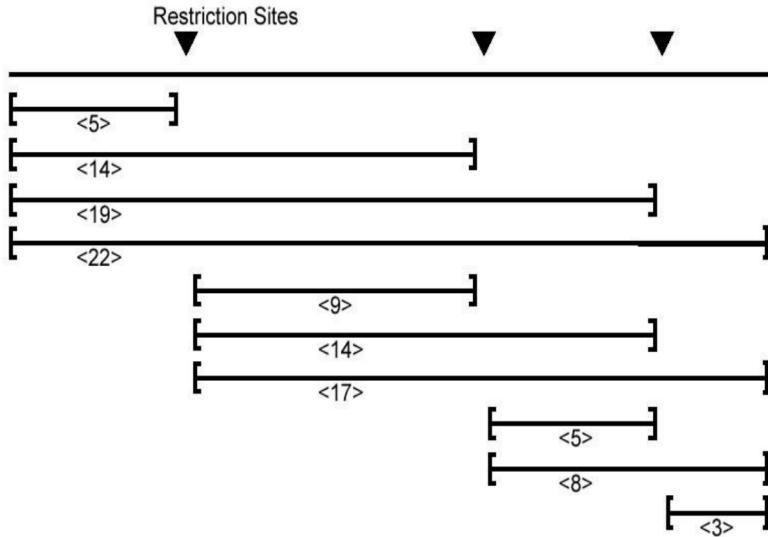


Obrázek 4.4: Příklad DDP s dvěma možnými řešeními.

DDP je oblíbená metoda, protože je experimentálně jednoduchá. Nicméně řešení nemusí být jednoznačné a lze ukázat, že DDP je NP-úplný problém.

*Důkaz.* (idea) Na DDP lze převést *problém 2 loupežníků*, tj. rozdělení množiny  $X$  na dvě stejně velké části. Převod funguje tak, že dosadíme  $\Delta A = \Delta AB = X$  a  $\Delta B = \{\frac{K}{2}, \frac{K}{2}\}$ , kde  $K = \sum_{i=1}^n x_i$ .  $\square$

**PDP (Partial Digest Problem)** Máme k dispozici kompletní  $\delta A$  získané opakováním vyštavením sekvence restrikčnímu enzymu, avšak jen po omezenou dobu.



Obrázek 4.5:  $X = \{0, 5, 14, 19, 22\}$ ,  $\delta X = \{3, 5, 5, 8, 9, 14, 14, 14, 17, 19, 22\}$

Množinu  $\delta X$  lze také reprezentovat pomocí horní trojúhelníkové matice  $M : M[i, j] = (x_j - x_i) : 1 \leq i < j \leq n$ . Z toho je zřejmé, že pro  $n$  řezů získáme celkem  $n(n - 1)/2$  fragmentů.

V podstatě vždy existuje více řešení, např. pro  $X = \{0, 2, 5\}$  a  $X' = \{10, 12, 15\}$  platí  $\delta X = \delta X' = \{2, 3, 5\}$ . Existují i netriviálně nejednoznačná řešení – tzv. *homometrické sety*.

Algoritmy:

### BruteForcePDP

1.  $L$  = experimentálně získaný partial digest

2.  $M =$  délka nejdelšího fragmentu ( $= |X|$ )
  3. pro každou možnou množinu  $X = \{0, x_2, \dots, x_{n-1, M}\}$  spočti  $\delta X$
  4. pokud  $\delta X = L$ , return  $X$
- Časová složitost  $O(M^{n-2})$ .

### AnotherBruteForcePDP

V kroku 3. používej pouze  $x_i$  z  $L$ . Časová složitost  $O(n^{2n-4})$ .

### Branch and Bound

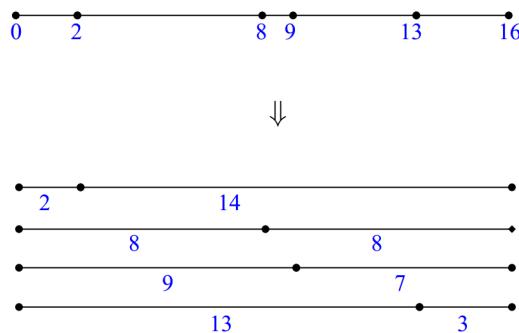
1. Begin with  $X = 0$
2. Remove the largest element in  $L$  and place it in  $X$
3. Pick the largest element in  $X$
4. If the element fits on the right
  - (a) Find all lengths it creates and remove those from  $L$
  - (b) Recursively continue by Step 3, until  $L$  is empty
  - (c) Put back all length removed in Step 1
5. If the element fits on the left
  - (a) Find all lengths it creates and remove those from  $L$
  - (b) Recursively continue by Step 3, until  $L$  is empty
  - (c) Put back all length removed in Step 1

Popsaný algoritmus je backtrackovací. Je tedy exponenciální, ale v praxi docela rychlý. Polynomiální algoritmus není znám (složitost PDP je otevřený problém).

V praxi se moc nepoužívá, je těžké získat experimentálně *všechny* vzájemné délky.

**SPDP (Simplified Partial Digest Problem)** Máme sekvenci  $S$  a enzym  $A$ , provedeme 2 experimenty:

- **krátký:** Každá kopie  $S$  je rozřezána právě na jednom místě (tj. na dvě části). Získáme multiset  $\Gamma = \{\gamma_1, \dots, \gamma_{2n}\}$
- **dlouhý:** Full digest. Získáme fragmenty  $\Lambda = \{\lambda_1, \dots, \lambda_{n+1}\}$



Obrázek 4.6:  $X = \{0, 2, 8, 9, 13, 16\}$ ,  $\Lambda = \{2, 6, 1, 4, 3\}$ ,  $\Gamma = \{2, 14, 8, 8, 9, 7, 13, 3\}$

Nechť  $\Gamma$  je neklesající, pak  $(\gamma_i, \gamma_{2n-i+1})$  tvoří **komplementární délky** (jejich součet je  $|S|$ ).

**Algoritmus:** Každý komplementární pár lze použít jako  $(\gamma_i, \gamma_{2n-i+1})$  nebo jako  $(\gamma_{2n-i+1}, \gamma_i)$ . Pomocí backtrackování nalezni takové konfigurace všech párů, že odvozené fragmenty odpovídají  $\Lambda$ .

Složitost je exponenciální. Problémy: nepřesné měření (lze řešit intervalovou aritmetikou), chybějící fragmenty (nicméně lze detektovat a doplnit chybějící komplement).

#### 4.4.2 Hledání motivů

Jen asi 1 % z DNA kóduje nějaké geny. Speciální protein se naváže na DNA těsně před začátkem genu a umožní čtení a transkripci. Tento protein pozná gen podle speciální značky v DNA kousek před začátkem genu, v tzv. *promoting region*, tyto značky obecně označujeme jako **motivy**. Motiv se může nacházet **kdekoli** v daném regionu a může být mírně **zmutovaný**.

**Motiv** je nějaká hledaná sekvence nukleotidů (A, C, T, G).

Problém: Máme množinu  $t$  DNA sekvencí délky  $n$  (reprezentovány maticí  $DNA$  o rozměrech  $t \times n$ ), chceme v nich najít společný motiv délky  $l$ . Na výstupu chceme pole  $t$  startovních pozic  $s = (s_1, \dots, s_t)$ , na nichž začíná motiv v daných sekvencích.

Jelikož původní motiv může být v různých sekvencích různě zmutovaný a navíc typicky původní motiv vůbec neznáme, chceme nějaké ohodnocení daného řešení (=pole startovních pozic) – to pak budeme chtít optimalizovat.

Předpokládejme nyní, že máme dány nějaké startovní pozice  $s = (s_1, \dots, s_t)$ . Pokud zarovnáme DNA sekvence podle těchto startovních pozic a „vystrihneme“  $l$ -mery začínající na těchto pozicích, dostaneme **alignment** o velikosti  $t \times l$ .

**Profil** pro dané sekvence a dané startovní pozice je pak matice  $4 \times l$ , kde řádky odpovídají jednotlivým nukleotidům, sloupce pozicím v alignmentu a hodnoty matice jsou počty výskytů daného nukleotidu na dané pozici.

**Konsenzus** je  $l$ -mer, který vznikne, když z profilu vybereme vždy nejčastější nukleotid na dané pozici.

**Skóre** je součet počtů výskytů těchto nejčastějších nukleotidů pro každý sloupec.

**Totální vzdálenost**  $l$ -merů začínajících na daných startovních pozicích daných sekvencí je součet Hammingovských vzdáleností těchto  $L$ -merů od konsenzu.

Zjevně platí, že **skóre + totální vzdálenost** =  $l \cdot t$ .

Alignment	<table border="0"> <tr><td>A</td><td>G</td><td>G</td><td>T</td><td>A</td><td>C</td><td>T</td><td>T</td></tr> <tr><td>C</td><td>C</td><td>A</td><td>T</td><td>A</td><td>C</td><td>G</td><td>T</td></tr> <tr><td>A</td><td>C</td><td>G</td><td>T</td><td>T</td><td>A</td><td>G</td><td>T</td></tr> <tr><td>A</td><td>C</td><td>G</td><td>T</td><td>C</td><td>C</td><td>A</td><td>T</td></tr> <tr><td>C</td><td>C</td><td>G</td><td>T</td><td>A</td><td>C</td><td>G</td><td>G</td></tr> </table>	A	G	G	T	A	C	T	T	C	C	A	T	A	C	G	T	A	C	G	T	T	A	G	T	A	C	G	T	C	C	A	T	C	C	G	T	A	C	G	G
A	G	G	T	A	C	T	T																																		
C	C	A	T	A	C	G	T																																		
A	C	G	T	T	A	G	T																																		
A	C	G	T	C	C	A	T																																		
C	C	G	T	A	C	G	G																																		

Profile	<table border="0"> <tr><td>A</td><td>3</td><td>0</td><td>1</td><td>0</td><td>3</td><td>1</td><td>1</td><td>0</td></tr> <tr><td>C</td><td>2</td><td>4</td><td>0</td><td>0</td><td>1</td><td>4</td><td>0</td><td>0</td></tr> <tr><td>G</td><td>0</td><td>1</td><td>4</td><td>0</td><td>0</td><td>0</td><td>3</td><td>1</td></tr> <tr><td>T</td><td>0</td><td>0</td><td>0</td><td>5</td><td>1</td><td>0</td><td>1</td><td>4</td></tr> </table>	A	3	0	1	0	3	1	1	0	C	2	4	0	0	1	4	0	0	G	0	1	4	0	0	0	3	1	T	0	0	0	5	1	0	1	4
A	3	0	1	0	3	1	1	0																													
C	2	4	0	0	1	4	0	0																													
G	0	1	4	0	0	0	3	1																													
T	0	0	0	5	1	0	1	4																													

Consensus      A C G T A C G T

Obrázek 4.7: Skóre pro tento profil je  $3+4+4+5+3+4+3+4 = 20$ . Totální vzdálenost od konsenzu je 10.

#### 4.4.2.1 Dvě ekvivalentní formulace problému

**Problém hledání motivu:** Pro  $t$  zadaných DNA sekvencí délky  $n$  (reprezentovány maticí  $DNA$  o rozměrech  $t \times n$ ) nalezní množinu  $l$ -merů, jeden z každé posloupnosti, které **maximalizují skóre**.

„**Median string**“ problém Pro  $t$  zadaných DNA sekvencí délky  $n$  (reprezentovány maticí  $DNA$  o rozměrech  $t \times n$ ) nalezní  **$l$ -mer  $v$ , který minimalizuje součet vzdáleností  $v$  vůči každé sekvenci**. Vzdálenost  $v$  k dané sekvenci je minimum z vzdáleností  $v$  ke všem možným  $l$ -merům z dané sekvence.

#### 4.4.2.2 Algoritmy

**BruteForceMotifSearch** Spočti skóre pro všechny možné startovní pozice  $s$ , vrát maximum.

Složitost  $O(l \cdot t \cdot n^t)$ , tj. nepoužitelné.

**MedianStringSearch** Pro všechny možné řetězce délky  $l$  (tj. AA...A, ..., TTT...T) spočti totální vzdálenost k daným sekvencím, vrát minimum.

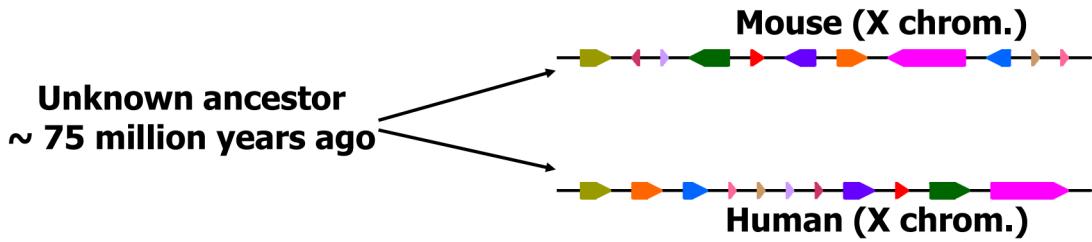
Složitost  $O(l \cdot t \cdot n \cdot 4^l)$ , tj. o něco lepší.

#### 4.4.3 Sequence alignment

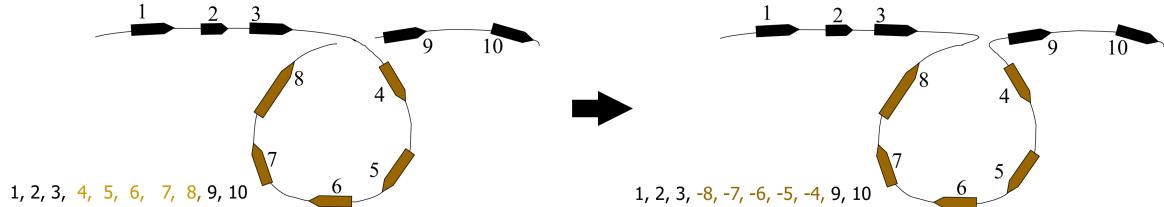
Sequence alignment; dotplot, scoring matrices (PAM, BLOSUM), pairwise and multiple alignment, Needleman wunsch , smith waterman, KBand (BLAST and FASTA heuristics)

#### 4.4.4 Genome rearrangement

**Genom** je kompletní genetický materiál organismu. Může nás zajímat srovnání genomů v rámci druhu i napříč druhy.



Příbuzné organismy mohou mít stejné geny, ale v jiném pořadí a případně opačně orientované (např. kapusta a tuřín mají 99 % stejných genů, jen v jiném pořadí).



Můžeme se setkat s těmito změnami: reversal, translokace, fusion a fission. My se budeme zabývat reversaly.

**Pořadí genů** v sekvenci budeme reprezentovat permutací

$$\pi = \pi_1 \dots \pi_{i-1} \pi_i \pi_{i+1} \dots \pi_{j-1} \pi_j \pi_{j+1} \dots \pi_n$$

**Reveresal**  $\rho(i, j)$  přetocí prvky  $i$  až  $j$  v  $\pi$ , tj. vznikne

$$\pi = \pi_1 \dots \pi_{i-1} \pi_j \pi_{j-1} \dots \pi_{i+1} \pi_i \pi_{j+1} \dots \pi_n$$

**Reversal distance problem:** Máme dvě permutace  $\pi$  a  $\sigma$ , chceme nalézt nejkratší posloupnost reversalů  $\rho_1, \dots, \rho_t$ , která transformuje  $\pi$  na  $\sigma$ . Označíme  $d(\pi, \sigma) = \min t$  jako **reversal distance permutací  $\pi$  a  $\sigma$** .

**Sorting by reversals problem:** Máme permutaci  $\pi$ , chceme nalézt nejkratší posloupnost reversalů  $\rho_1, \dots, \rho_t$ , která transformuje  $\pi$  na  $(1, 2, \dots, n)$ . Označíme  $d(\pi) = \min t$  jako **reversal distance permutace  $\pi$** .

#### 4.4.5 Predikce genů

Gene prediction, codon preference, HMM, CG islands.

z ML in BI: 2: unsupervised: density estimation, data visualisation

# Kapitola 5

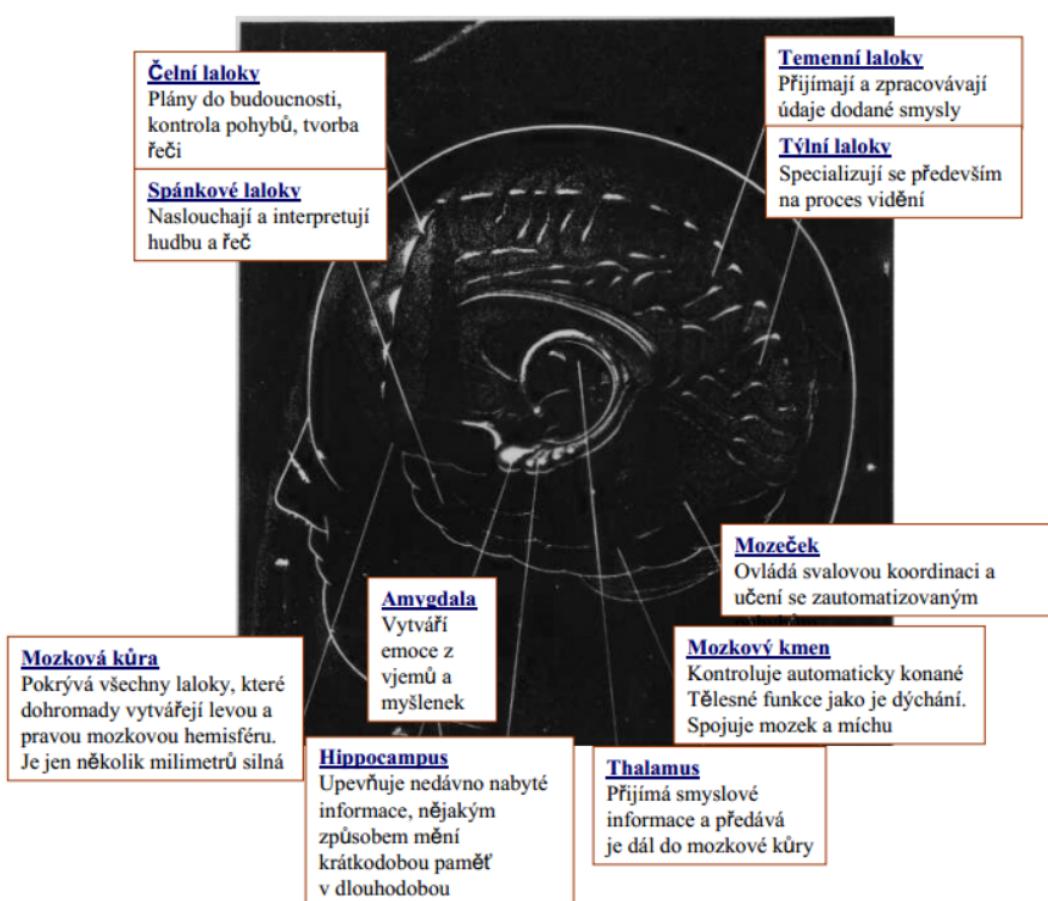
## Neuronové sítě

Zdroje: slajdy Mrázové, knížka R. Rojas: Neural Networks: A Systematic Introduction (podle které jsou ty slajdy udělané – pokud nejde něco ze slajdů pochopit, v knížce jde najít širší kontext)

### 5.1 Neurofyziologické minimum.

Neuronové sítě jsou výpočetní model inspirovaný lidským mozkem a jeho fungováním, proto rozebereme základní biologické poznatky.

#### 5.1.1 Mozek



#### 5.1.2 Neuron

Biologický neuron se skládá z **těla (somatu)**, **dendritů** a **axonu**. Jednotlivé neurony jsou spojeny prostřednictvím **synapsí**.

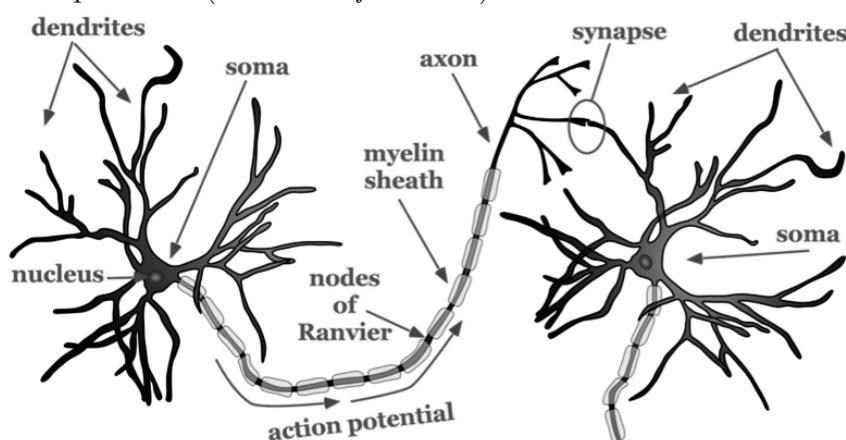
**tělo (soma)** Obsahuje organely neuronu včetně jádra (nukleus). Na základě signálů z dendritů může dojít k **excitaci**.

**dendrity** „Antény“ neuronu, místa vstupu signálů z ostatních neuronů (přijímací strana synapse). Místa synapsí jsou pokryta speciální membránou plnou tzv. *receptorů*, které detekují neurotransmittery v synaptické mezeře (*synaptic cleft*). Délka cca 2-3 mm.

**axon** Jediný výstup neuronu, který však bývá bohatě (typicky pravoúhle) rozvětvený. Přenáší signál k synapsím a dále do ostatních napojených neuronů. Délka může být i přes 1 m.

**synapse** Místo kontaktu z jiným neuronem. Dochází původně elektrický signál přivedený axonem se zde mění na chemický, který překlene mezeru (*synaptic cleft*) mezi presynaptickou a postsynaptickou plochou (z axonu jednoho neuronu do dendritu jiného neuronu). Na 1 neuron připadá až  $10^6$  spojů s jinými neuronami.

Kromě neuronů se v mozku nachází ještě *glie*, které mají především podpůrnou funkci. Prvním typem jsou **astrocyty**, které vyplňují prostor mezi neurony a obalují místa synapsí, čímž brání šíření neurotransmitterů mimo *synaptic cleft*. Druhým typem jsou **myelinating glia**, které obalují axony (celý obal se nazývá *myelin*). Obálka není zcela souvislá, v pravidelných intervalech je membrána axonu exponována (tzv. *node of Ranvier*).

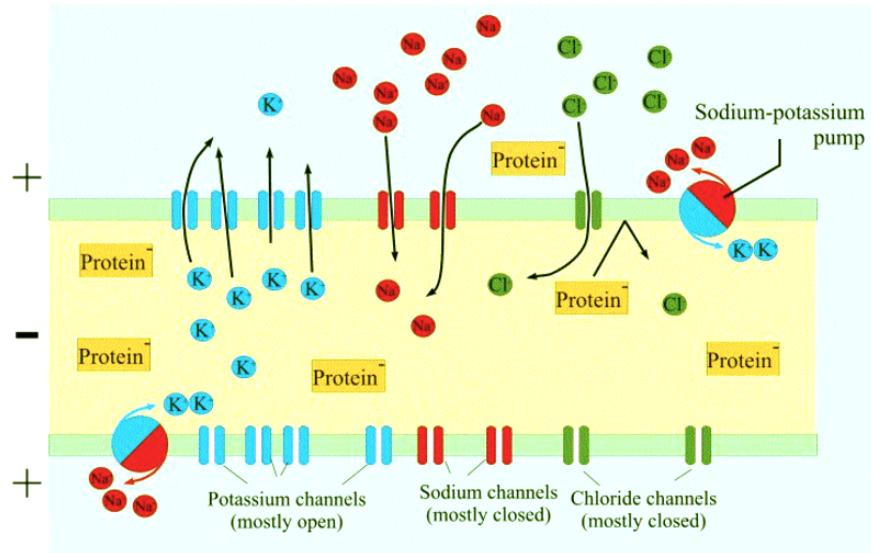


### 5.1.3 Přenos signálu

Obal neuronu tvoří *membrána*, která se skládá ze dvou vrstev molekul **fosfolipidů**. V membráně jsou umístěny **iontové pumpy** a **kanály**. Iontové kanály volně propouští ionty daného typu, iontové pumpy je přenáší aktivně a spotřebovávají přitom energii (ve formě ATP). Každý kanál či pumpa jsou selektivní vůči jednomu typu iontů: především  $K^+$ ,  $Na^+$  a  $Cl^-$ .

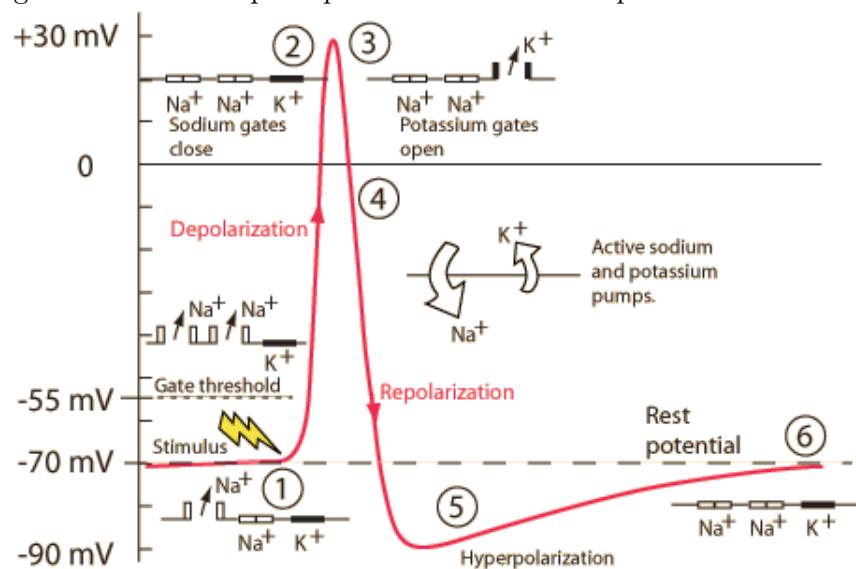
#### 5.1.3.1 Klidový potenciál

Pomocí pump je udržována neustálá **polarizace membrány**. Vně je kladný potenciál, uvnitř záporný (rozdíl se pohybuje kolem  $-70\text{ mV}$ ). Nejvíce prominentní jsou sodíkovo-draslíkové pumpy, které pumpují  $K^+$  dovnitř a současně  $Na^+$  ven. Draslíkové kanály jsou otevřené, takže nepoměr koncentrace draslíkových iontů vně a uvnitř se může vyrovnávat. Sodíkové kanály jsou ovšem uzavřené, takže sodík zůstává více koncentrovaný vně. To vede k zápornému potenciálu uvnitř neuronu.



### 5.1.3.2 Akční potenciál

Některé  $\text{Na}^+$  a  $\text{K}^+$  kanály jsou *voltage-gated*, tj. při dosažení jisté úrovně napětí ( $-55 \text{ mV}$ ) dochází k jejich automatickému uzavření/otevření. Tím je umožněno generování akčního potenciálu. Příliv  $\text{Na}^+$  (například z jiného neuronu, vybuzením receptorů v dendritech) dochází k depolarizaci neuronu. Dosáhne-li depolarizace kritické úrovni (*threshold*), dojde k uzavření  $\text{K}^+$  kanálů a otevření  $\text{Na}^+$  kanálů a rapidnímu přílivu  $\text{Na}^+$  dovnitř neuronu, neboť v něm je stále negativní potenciál. Tzv. *rising phase*. Příliv je natolik veliký, že dojde k tzv. *overshoot*, kdy je vnitřek neuronu nabit kladně na cca  $40 \text{ mV}$ . V tuto chvíli dochází k uzavření  $\text{Na}^+$  kanálů a otevření  $\text{K}^+$  kanálů a potenciál se opět začíná snižovat - tzv. *falling phase*. Jelikož je otevřeno více  $\text{K}^+$  kanálů než obvykle (klasické + voltage-gated), dochází k tzv. *undershoot*, tj. hyperpolarizaci neuronu. Po zavření voltage-gated kanálů se napětí opět ustálí na klidovém potenciálu.



### 5.1.4 Paměť

**Krátkodobý paměťový mechanismus** Založen na cyklickém oběhu vznuků v neuronových sítích. Proběhne-li tato cirkulace cca 300-krát, začne docházet k fixaci informace ve střednědobé paměti – to trvá cca 30 s.

**Střednědobý paměťový mechanismus** Založen na změnách „vah neuronů“. Změna váhových koeficientů v synapsi je vyvolána mnohonásobným působením téhož signálu na příslušných synaptických přechodech. Ve spánku přecházejí některé z takto uchovaných informací do dlouhodobých pamětí. Informace se uchovává několik hodin a případně i dnů.

**Dlouhodobý paměťový mechanismus** Spočívá v kopírování informací ze střednědobé paměti do bílkovin, které jsou uvnitř neuronů – hlavně v jejich jádřech. Některé takto uchovávané informace zůstanou v organismu celý život.

## 5.2 Modely pro učení s učitelem, algoritmus zpětného šíření, strategie pro urychlení učení, regularizační techniky a generalizace.

### 5.2.1 Modely pro učení s učitelem

#### 5.2.1.1 Formální neuron

**Formální neuron** s vahami  $(w_1, w_2, \dots, w_n) \in \mathbb{R}^n$ , prahem  $\theta \in \mathbb{R}$  a přenosovou funkcí  $f : \mathbb{R}^{n+1} \times \mathbb{R}^n \rightarrow \mathbb{R}$  počítá pro libovolný vstup  $\mathbf{x} \in \mathbb{R}^n$  svůj výstup  $y$  jako hodnotu přenosové funkce  $f$  v  $\mathbf{x}$ ,  $f[\mathbf{w}, \theta](\mathbf{z})$ . Mezi nejznámější **přenosové funkce** (také **aktivační funkce**) patří *skoková* (unit step), *sigmoidální* nebo *hyperbolická tangentoida*. **Potenciál neuronu** je vážená suma jeho vstupů:  $\xi = \sum_{i=1}^n x_i w_i + \theta$ . Právě tento potenciál je vstupem přenosové funkce (na přehledu níže je potenciál neuronu označen písmenem  $z$ ).

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	Multi-layer NN	

#### 5.2.1.2 Perceptron

**(Jednoduchý) perceptron** je výpočetní jednotka sestávající z jediného neuronu se skokovou přenosovou funkcí, tj.

$$y = f[\mathbf{w}, \theta](\mathbf{x}) = \begin{cases} 1 & \sum_{i=1}^n w_i x_i \geq \theta \quad \text{tedy pokud } \mathbf{w} \cdot \mathbf{x} \geq \theta \\ 0 & \text{jinak} \end{cases}$$

Často se uvažuje tzv. **rozšířený váhový a vstupní vektor**, kde je navíc umělý vstup s konstantní hodnotou 1 a vahou  $-\theta$ , tj.  $\overrightarrow{w_{ext}} = (w_1, w_2, \dots, w_n, -\theta)$ ,  $\overrightarrow{x_{ext}} = (x_1, x_2, \dots, x_n, 1)$ . Potom lze práh neuronu považovat za váhu speciálního vstupu a přenosovou funkci počítat jako

$$y = f[\mathbf{w}, \theta](\mathbf{x}) = \begin{cases} 1 & \overrightarrow{w_{ext}} \cdot \overrightarrow{x_{ext}} \geq 0 \\ 0 & \text{jinak} \end{cases}$$

Jednoduchý perceptron ve skutečnosti realizuje **dělící nadrovину**, jejíž poloha je dána váhovým vektorem: je to množina všech bodů  $\mathbf{x} \in \mathbb{R}^n$ , pro něž  $\mathbf{w} \cdot \mathbf{x} = 0$ . Ve dvourozměrném případě odpovídá perceptron dělící přímce. Všechny body na jedné straně přímky jsou perceptronem klasifikovány jako 1, všechny body na druhé straně jako 0. Formálně mluvíme o *pozitivním a negativním podprostoru*: **otevřený (uzavřený) pozitivní podprostor** určený váhovým vektorem  $\mathbf{w}$  je množina všech bodů  $\mathbf{x} \in \mathbb{R}^n$ , pro něž  $\mathbf{w} \cdot \mathbf{x} > 0$  ( $\mathbf{w} \cdot \mathbf{x} \geq 0$ ). **Negativní podprostor** je definován analogicky.

Abychom mohli formulovat typy úloh, které perceptron umí řešit, budeme také potřebovat následující definici:

Dvě množiny  $A$  a  $B$  se nazývají **lineárně separabilní** v  $n$ -rozměrném prostoru, pokud existuje  $n + 1$  reálných čísel  $w_1, w_2, \dots, w_n, \theta$  takových, že každý bod  $\mathbf{x} = (x_1, x_2, \dots, x_n) \in A$  splňuje  $\sum_{i=1}^n w_i x_i \geq \theta$  a každý bod  $\mathbf{x} = (x_1, x_2, \dots, x_n) \in B$  splňuje  $\sum_{i=1}^n w_i x_i < \theta$ . Pokud dokonce v obou případech platí ostrá nerovnost, mluvíme o **absolutní lineární separabilitě**.

Lze dokázat, že pokud jsou dvě množiny separabilní, jsou i absolutně separabilní (idea důkazu: posun přímky o  $\varepsilon/2$ ).

Nalezení vhodné dělící nadroviny pro zadané body je ekvivalentní nalezení vhodného vektoru  $\mathbf{w}$ . To probíhá pomocí **perceptronového učení**:

1. inicializace vah náhodnými hodnotami  $w_i(0)$
2. předložení trénovacího vzoru, tj.  $\mathbf{x} = (x_1, x_2, \dots, x_n)$  a očekávaného výstupu  $d(t)$
3. výpočet skutečného výstupu jako  $y(t) = \text{sgn}(\mathbf{w} \cdot \mathbf{x})$
4. adaptace vah:

$w_i(t+1) = w_i(t)$	výstup je správný
$w_i(t+1) = w_i(t) + x_i$	výstup je 0 a měl být 1
$w_i(t+1) = w_i(t) - x_i$	výstup je 1 a měl být 0

5. pokud  $t$  nedosáhl požadované hodnoty, přejdi ke kroku 2

Základním principem je natáčení vektoru  $\mathbf{w}$  (který je kolmý na dělící nadrovinu) ve směru kladných vzorků. Je-li výstup kroku 3 kladný, svírají  $\mathbf{w}$  a  $\mathbf{x}$  úhel menší než  $90^\circ$ , je-li záporný, je úhel větší než  $90^\circ$ .

Pro nastavení počátečních vah může být použita jednoduchá heuristika: vezmi průměr kladných vstupů minus průměr záporných vstupů.

Další heuristikou je použít parametr učení  $\eta \in (0, 1)$ , který ovlivňuje plasticitu modelu. Váhy se pak aktualizují podle  $w_i(t+1) = w_i(t) + \eta \cdot x_i$ , resp.  $w_i(t+1) = w_i(t) - \eta \cdot x_i$ .

**Věta (konvergence perceptronového algoritmu učení):** Nechť  $P$  a  $N$  jsou konečné a lineárně separabilní množiny. Potom provede perceptronový algoritmus učení konečný počet aktualizací váhového vektoru  $\mathbf{w}$ .

*Důkaz.* Nejprve provedeme 3 zjednodušení:

1. Sjednotíme  $P$  a  $N$  jako  $P' = P \cup N^-$ , kde  $N^-$  jsou negované prvky  $N$ .
2. Vektory  $P'$  normalizujeme.
3. Váhový vektor bude také normalizovaný. Předpokládané řešení označíme jako  $\mathbf{w}^*$ .

Nyní uvážíme situaci v kroku  $t$ , kdy došlo k aktualizaci váhového vektoru pomocí nějakého  $\mathbf{p}_i \in P'$ , tedy  $\mathbf{w}_{t+1} = \mathbf{w}_t + \mathbf{p}_i$  (pro přehlednost přesuneme  $t$  do dolního indexu). Nyní budeme zkoumat úhel mezi  $\mathbf{w}^*$  a  $\mathbf{w}_{t+1}$ , konkrétně výraz:

$$\cos \rho = \frac{\mathbf{w}^* \cdot \mathbf{w}_{t+1}}{\|\mathbf{w}_{t+1}\|} \quad (5.2.1)$$

Pro výraz v čitateli víme, že:

$$\mathbf{w}^* \cdot \mathbf{w}_{t+1} = \mathbf{w}^* \cdot (\mathbf{w}_t + \mathbf{p}_i) = \mathbf{w}^* \cdot \mathbf{w}_t + \mathbf{w}^* \cdot \mathbf{p}_i \geq \mathbf{w}^* \cdot \mathbf{w}_t + \delta$$

kde  $\delta = \min\{\mathbf{w}^* \cdot \mathbf{p} \mid \forall \mathbf{p} \in P'\}$ . Protože  $\mathbf{w}^*$  definuje *absolutní* lineární separaci  $P$  a  $N$ , víme, že  $\delta > 0$ . Indukcí dostáváme

$$\mathbf{w}^* \cdot \mathbf{w}_{t+1} \geq \mathbf{w}^* \cdot \mathbf{w}_0 + (t+1)\delta \quad (5.2.2)$$

Pro výraz ve jmenovateli platí

$$\|\mathbf{w}_{t+1}\|^2 = (\mathbf{w}_t + \mathbf{p}_i) \cdot (\mathbf{w}_t + \mathbf{p}_i) = \|\mathbf{w}_t\|^2 + 2\mathbf{w}_t \cdot \mathbf{p}_i + \|\mathbf{p}_i\|^2$$

Všechny vektory v  $P'$  jsou normalizovány, takže poslední člen je roven 1. Navíc  $\mathbf{w}_t \cdot \mathbf{p}_i \leq 0$  (jinak by nebylo třeba aktualizovat váhový vektor), takže dostáváme

$$\|\mathbf{w}_{t+1}\|^2 \leq \|\mathbf{w}_t\|^2 + \|\mathbf{p}_i\|^2 \leq \|\mathbf{w}_t\|^2 + 1$$

a indukcí dostaneme

$$\|\mathbf{w}_{t+1}\|^2 \leq \|\mathbf{w}_0\|^2 + (t+1) \quad (5.2.3)$$

Porovnáním 5.2.2 a 5.2.3 s původní 5.2.1 dostáváme nerovnici

$$\cos \rho \geq \frac{\mathbf{w}^* \cdot \mathbf{w}_0 + (t+1)\delta}{\sqrt{\|\mathbf{w}_0\|^2 + (t+1)}}$$

Pravá strana nerovnice roste proporcionálně k  $\sqrt{t}$ , a protože  $\delta > 0$ , mohla by být libovolně velká. Protože ale  $\cos \rho \leq 1$ , musí existovat horní mez a počet aktualizací váhového vektoru musí být konečný.

□

Zmíníme ještě, že popsany algoritmus není jediný. Existují různé varianty a vylepšení, ve slajdech pí. Mrázové je popsán ještě *přihrádkový algoritmus*.

### 5.2.1.3 Neuronová síť

**Neuronová síť** je uspořádaná 6-tice  $M = (N, C, I, O, w, t)$ , kde:

- $N$  je konečná neprázdná množina neuronů
- $C \subseteq N \times N$  je neprázdná množina orientovaných spojů mezi neurony
- $I \subseteq N$  je neprázdná množina vstupních neuronů
- $O \subseteq N$  je neprázdná množina výstupních neuronů
- $w : C \rightarrow \mathbb{R}$  je váhová funkce
- $t : N \rightarrow \mathbb{R}$  je prahová funkce

Neurony jsou typicky uspořádány do vrstev, pak mluvíme o **vrstevnatých sítích**. První vrstva je **vstupní vrstva** sestávající ze vstupních neuronů, ty nemají v grafu spojů žádné předchůdce a jejich výstup je roven jejich vstupu. Poslední vrstva je **výstupní vrstva** obsahující výstupní neurony. Zbylé vrstvy jsou **skryté vrstvy**.

Typicky uvažujeme síť s acyklickým grafem spojů, tzv. **dopředné sítě** (angl. *feed-forward networks*). Sítě obsahující cykly nazýváme obecně **rekurentní sítě**.

### 5.2.2 Algoritmus zpětného šíření (Backpropagation)

Nejpoužívanější algoritmus učení pro neuronové sítě. Cílem učení je nastavit váhy sítě tak, aby sítě správně počítala výstupy pro předkládané vzory. Přitom však není specifikována ani skutečná, ani očekávaná aktivita skrytých neuronů. Jedná se o tzv. *gradientní metodu*, tzn. snaží se minimizovat nějakou (chybovou) funkci, k čemuž využívá kroky ve směru gradientu funkce v aktuálním bodě.

Pro konečnou množinu trénovacích vzorů  $T$  lze celkovou chybu vyjádřit pomocí rozdílu mezi skutečným a požadovaným výstupem sítě u každého předloženého vzoru. Definujeme tedy **chybovou funkci** jako

$$E = \frac{1}{2} \sum_{p \in T} \sum_{j \in O} (y_{j,p} - d_{j,p})^2$$

kde  $T$  je trénovací množina,  $O$  jsou výstupní neurony,  $y_{j,p}$  je skutečný výstup neuronu  $j$  pro vzor  $p$  a  $d_{j,p}$  je očekávaná odezva neuronu  $j$  na vzor  $p$ . Používá se kvadratická chyba, aby se zanedbal směr chyby. Zlomek  $1/2$  je ve vzorci pouze pro pohodlnější počítání při pozdějším derivování.

Cílem učení je minimalizovat chybu na dané trénovací množině. Úprava vah sítě probíhá iterativně, tj. předloží se vzor, porovná se skutečný a očekávaný výstup, spočte se chyba, adaptují se váhy a pak se pokračuje dalším vzorem. Váhy se upravují od výstupní vrstvy směrem ke vstupní a proti gradientu chybové funkce.

Dlužno poznamenat, že síť je po naučení ještě třeba otestovat na nezávislé testovací množině, aby se stanovila její úspěšnost.

### 5.2.2.1 Adaptační pravidla

V každém kroku aktualizujeme váhy:

$$w_{ij}(t+1) = w_{ij}(t) + \Delta_E w_{ij}(t)$$

kde  $w_{ij}$  je váha spoje mezi neurony  $i$  a  $j$ ; a  $\Delta_E w_{ij}(t)$  je přírůstek váhy přispívající k minimalizaci chyby  $E$ . Ten nalezneme „derivací chyby ve směru této váhy“, kteroužto hodnotu pak odečteme:

$$\Delta_E w_{ij}(t) = -\frac{\partial E}{\partial w_{ij}} = -\frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial \xi_j} \cdot \frac{\partial \xi_j}{\partial w_{ij}}$$

Hodnota  $y_j$  je skutečný výstup neuronu  $j$  a  $\xi_j$  je potenciál neuronu  $j$ , tj. vážená suma jeho vstupů. Tento vzorec ještě dále upravíme, a to zvlášť pro výstupní vrstvu a pro skryté vrstvy.

#### 5.2.2.1.1 Aktualizace synaptických vah pro výstupní vrstvu:

$$\begin{aligned} \Delta_E w_{ij}(t) &\cong -\frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial \xi_j} \cdot \frac{\partial \xi_j}{\partial w_{ij}} \\ &= -\frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial \xi_j} \cdot \frac{\partial}{\partial w_{ij}} \sum_{i'} w_{i'j} y_{i'} \\ &\stackrel{(1)}{=} -\frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial \xi_j} \cdot y_i \\ &\stackrel{(2)}{=} -\frac{\partial E}{\partial y_j} \cdot f'(\xi_j) \cdot y_i \\ &\stackrel{(3)}{=} -(y_j - d_j) \cdot f'(\xi_j) \cdot y_i = \delta_j \cdot y_i \end{aligned}$$

Rovnost (1) platí, neboť v sumě je pouze jediný člen, v němž se vyskytuje „proměnná“  $w_{ij}$ , a to  $w_{ij} y_i$ , který bude po derivaci roven  $y_i$ . Ostatní členy jsou vůči  $w_{ij}$  konstantní a jejich derivace je tedy 0.

Rovnost (2) pracuje pouze s jiným vyjádřením druhého zlomku. Neboť  $y_j = f(\xi_j)$  a tedy druhý zlomek je vlastně pouze jednoduchá derivace přenosové funkce.

Rovnost (3) platí triviálně z definice chybové funkce. Ta je suma rozdílů  $(y_k - d_k)$ , z nichž pouze jeden zůstane po derivaci nenulový. Celkovou derivaci chyby  $E$  podle potenciálu  $\xi_j$  označíme symbolem  $\delta_j$  (bude se nám to hodit později).

### 5.2.2.1.2 Aktualizace synaptických vah pro skryté vrstvy:

$$\begin{aligned}
\Delta_E w_{ij}(t) &\cong -\frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial \xi_j} \cdot \frac{\partial \xi_j}{\partial w_{ij}} \\
&\stackrel{(1)}{=} -\frac{\partial E}{\partial y_j} \cdot f'(\xi_j) \cdot y_i \\
&\stackrel{(2)}{=} -\left( \sum_k \frac{\partial E}{\partial \xi_k} \cdot \frac{\partial \xi_k}{\partial y_j} \right) \cdot f'(\xi_j) \cdot y_i \\
&= -\left( \sum_k \frac{\partial E}{\partial \xi_k} \cdot \frac{\partial}{\partial y_j} \sum_{j'} w_{j'k} y_{j'} \right) \cdot f'(\xi_j) \cdot y_i \\
&\stackrel{(3)}{=} -\left( \sum_k \frac{\partial E}{\partial \xi_k} \cdot w_{jk} \right) \cdot f'(\xi_j) \cdot y_i \\
&\stackrel{(4)}{=} -\left( \sum_k \delta_k \cdot w_{jk} \right) \cdot f'(\xi_j) \cdot y_i = \delta_j \cdot y_i
\end{aligned}$$

Rovnost (1) plyne stejně, jako v předchozím případě (zestručněno).

Rovnost (2) platí, neboť  $y_j$  již není výstup neuronu ve výstupní vrstvě, který by šlo přímo porovnat s očekávaným výstupem, nýbrž výstup nějakého vnitřního neuronu. Budeme tedy iterovat přes všechny neurony  $k$ , do nichž vede spoj z  $j$  (a  $y_j$  tedy přispívá do potenciálu  $\xi_k$ ) a tak vyjádříme vliv  $y_j$  na celkovou chybu takto nepřímo.

Rovnost (3) platí analogicky jako rovnost (1) v předchozím případě.

Rovnost (4) využívá označení  $\delta_j$  zavedeného v předchozím případě. Jak je vidět, pro úpravu vah vedoucích do vnitřního neuronu  $j$  potřebujeme znát  $\delta_k$  pro všechny vrcholy  $k$ , do nichž vede z  $j$  hrana. Toto vynucuje počítat směrem od výstupní vrstvy, neboť pro všechny výstupní neurony umíme  $\delta_k$  jednoduše spočítat. Pak můžeme spočítat  $\delta_k$  (a aktualizace vah) pro předposlední vrstvu, pak pro před-předposlední atd.

Než přejdeme k finálnímu sjednocení vzorců a vyjádření aktualizace vah, vyjádříme si ještě  $f'(\xi_j)$ . Pracujeme se *sigmoidální přenosovou funkcí*, tj.  $f(\xi_j) = \frac{1}{1+e^{-\lambda\xi_j}}$ . Pro tu platí zajímavá a užitečná věc, a to že  $f'(\xi_j) = \lambda f(\xi_j)(1 - f(\xi_j))$  neboli  $f'(\xi_j) = \lambda y_j(1 - y_j)$ .

Nyní tedy umíme vyjádřit přírůstek váhy jako

$$w_{ij}(t+1) = w_{ij}(t) + \alpha \delta_j y_i$$

kde

$$\delta_j = \begin{cases} (d_j - y_j) \lambda y_j (1 - y_j) & \text{pro výstupní neuron} \\ \left( \sum_k \delta_k \cdot w_{jk} \right) \lambda y_j (1 - y_j) & \text{pro skrytý neuron} \end{cases}$$

Parametr  $\alpha \in (0, 1)$  nazýváme **parametrem učení**.

Nyní se pokusíme celý algoritmus zpětného šíření stručně shrnout:

Krok 1: Zvolte náhodné hodnoty synaptických vah.

Krok 2: Předložte nový trénovací vzor ve tvaru:

[ vstup  $\mathbf{x}$ , požadovaný výstup  $\mathbf{d}$  ]

Krok 3: Vypočtěte skutečný výstup. Aktivita neuronů v každé vrstvě je dána pomocí:

$$y_j = f(\xi_j) = \frac{1}{1 + e^{-\lambda\xi_j}}, \quad \text{kde } \xi_j = \sum_i w_{ij}y_i$$

Krok 4: Aktualizujte váhy postupně směrem od výstupní vrstvy ke vstupní podle vzorce

$$w_{ij}(t+1) = w_{ij}(t) + \alpha\delta_j y_i$$

kde

$$\delta_j = \begin{cases} (d_j - y_j)\lambda y_j(1 - y_j) & \text{pro výstupní neuron} \\ \left( \sum_k \delta_k \cdot w_{jk} \right) \lambda y_j(1 - y_j) & \text{pro skrytý neuron} \end{cases}$$

a dále

- $w_{ij}(t)$  je váha spoje z neuronu  $i$  do neuronu  $j$  v čase  $t$
- $\alpha$  je parametr učení
- $\xi_j$  je potenciál neuronu  $j$
- $\delta_j$  je chyba na neuronu  $j$
- $k$  je index pro neurony z vrstvy nad neuronem  $j$
- $\lambda$  je strmost přenosové funkce

Krok 5: Přejdi ke kroku 2.

### 5.2.3 Strategie pro urychlení učení

Popsaný standardní model zpětného učení je poměrně jednoduchý, dává poměrně dobré výsledky, ale má i mnohé nevýhody. V první řadě je docela pomalý. Problém učení neuronových sítí je obecně NP-úplný a výpočetní složitost roste exponenciálně s počtem proměnných. Důležité je taky počáteční nastavení parametrů.

Existují různé postupy a algoritmy pro urychlení učení. Některé zachovávají pevnou topologii sítě, jiné jsou naopak založeny na použití modulárních sítí, některé algoritmy adaptují jak parametry (váhy, prahy, ...) i topologii.

#### 5.2.3.1 Volba počátečních vah

Příliš malé počáteční váhy mohou „paralyzovat učení“, tj. propagovaná chyba je příliš malá. Příliš velké váhy naopak vedou k „saturaci“ neuronů a rovněž pomalému učení. Oba extrémy mají pak za následek ukončení učení v suboptimálním lokálním extrému. Správná volba počátečních vah může toto riziko výrazně snížit.

Chceme zvolit váhy tak, aby potenciál neuronu byla náhodná proměnná se směrodatnou odchylkou  $A$ . Potenciál skrytého neuronu je

$$\xi = w_0 + w_1x_1 + \cdots + w_nx_n$$

Střední hodnota potenciálu je

$$E[\xi_j] = E \left[ \sum_{i=0}^n w_{ij}x_i \right] = \sum_{i=0}^n E[w_{ij}] \cdot E[x_i] = 0$$

což plyne z toho, že váhy jsou nezávislé na vzorech a jsou to náhodné proměnné se střední hodnotou

0. Rozptyl potenciálu je dán pomocí

$$\sigma_{\xi_j}^2 = E[\xi_j^2] - E^2[\xi_j] = E \left[ \left( \sum_{i=0}^n w_{ij} x_i \right)^2 \right] - 0 = \sum_{i,k=0}^n E[w_{ij} w_{kj} x_i x_k] = \sum_{i=0}^n E[w_{ij}^2] E[x_i^2]$$

kde poslední rovnost plyne z vzájemné nezávislosti pro všechna  $j$ . Nyní předpokládejme, že trénovací vzory jsou normalizované a leží v intervalu  $\langle 0, 1 \rangle$ . Pak

$$E[x_i^2] = \int_0^1 x_i^2 dx = \left[ \frac{x^3}{3} \right]_0^1 = \frac{1}{3}$$

Dále nechť váhy jsou náhodné proměnné se střední hodnotou 0 a rovnoměrně rozložené v intervalu  $\langle -a, a \rangle$ . Pak

$$E[w_{ij}^2] = \int_{-a}^a w_{ij}^2 \cdot \frac{1}{2a} dw_{ij} = \left[ \frac{w_{ij}^3}{3 \cdot 2a} \right]_{-a}^a = \frac{a^2}{3}$$

Směrodatná odchylka pro  $N$  vah vedoucích do neuronu  $j$  je pak tedy

$$\sigma_{\xi_j} = \frac{a}{3} \sqrt{N}$$

Chceme-li tedy, aby potenciál neuronu měl danou odchylku  $A$  nezávisle na počtu příchozích vah, pak volme počáteční váhy z intervalu

$$\left( -\frac{3}{\sqrt{N}}A, \frac{3}{\sqrt{N}}A \right)$$

Speciálně pro  $A = 1$  dosáhneme poměrně velkého gradientu a rychlého učení.

### 5.2.3.2 Moment učení

Pokud by se učení řídilo pouze gradientem, může v „úzkých údolích“ chybové funkce docházet k silným oscilacím. Přidání momentu zajistí jistou setrvačnost. Změna váhy  $w_{ij}$  v kroku  $t + 1$  je pak dána jako

$$\Delta w_{ij}(t+1) = -\alpha \frac{\partial E}{\partial w_{ij}(t)} + \alpha_m (w_{ij}(t) - w_{ij}(t-1))$$

kde  $\alpha$  je *parametr učení* a  $\alpha_m$  nazýváme **moment učení**. Vyvážení parametrů  $\alpha$  a  $\alpha_m$  určuje, nakolik se učení řídí gradientem a nakolik setrvačností. Optimální nastavení závisí na konkrétní úloze.

### 5.2.3.3 Adaptivní parametr učení

Myšlenkou je ovládat rychlosť učení pomocí změny parametru učení v průběhu učení. Např. v oblastech vzdálených od lokálního minima může být gradient velmi nízký, pak je vhodné učení urychlit; naopak v oblastech s velkými gradienty jsou vhodnější menší kroky.

Každé váze  $w_i$  přidělíme její vlastní lokální parametr  $\alpha_i$ . Váhy poté adaptujeme podle

$$\Delta w_i = -\alpha_i \frac{\partial E}{\partial w_i}$$

**Algoritmy:**

**Algoritmus Silvy a Almeidy** Základní myšlenka: pokud se poslední dvě iterace nezměnilo znaménko parciální derivace, *urychluj*. Pokud se změnilo, *zpomaluj*.

Formálně:

$$\alpha_i^{(k+1)} = \begin{cases} \alpha_i^{(k)} \cdot u & \text{jestliže } \nabla_i E^{(k)} \cdot \nabla_i E^{(k-1)} > 0 \\ \alpha_i^{(k)} \cdot d & \text{jestliže } \nabla_i E^{(k)} \cdot \nabla_i E^{(k-1)} < 0 \end{cases}$$

kde konstanty  $d < 1, u > 1$  jsou pevně zvolené,  $\nabla_i E^{(k)}$  je parciální derivace chybové funkce podle váhy  $w_i$  v kroku  $k$ . Počáteční hodnoty  $\alpha_i^{(0)}$  se inicializují malými náhodnými hodnotami.

Problémem tohoto přístupu je, že parametr učení roste resp. klesá exponenciálně vůči  $u$  resp.  $d$ . Problémy tedy mohou nastat, pokud následuje mnoho urychlovacích kroků po sobě.

**Delta-bar-delta** Varianta předchozího algoritmu s opatrnějším zrychlováním. Váhy se adaptují podle

$$\alpha_i^{(k+1)} = \begin{cases} \alpha_i^{(k)} + u & \text{jestliže } \nabla_i E^{(k)} \cdot \delta_i^{(k-1)} > 0 \\ \alpha_i^{(k)} \cdot d & \text{jestliže } \nabla_i E^{(k)} \cdot \delta_i^{(k-1)} < 0 \\ \alpha_i^{(k)} & \text{jinak} \end{cases}$$

kde  $\delta_i^{(k)} = (1 - \Phi) \nabla_i E^{(k)} + \Phi \delta_i^{k-1}$ , přičemž  $\Phi$  je konstanta.

**Super SAB** Opět varianta předchozího. Rozdílem je použití momentu a odlišné chování při změně znaménka gradientu.

1. nastav všchna  $\alpha_i^{(0)}$  na  $\alpha_{START} = 1.2$
2. proved' krok algoritmu zpětného šíření s **momentem**
3. pokud se nezměnilo znaménko  $\nabla_i E$ , nastav  $\alpha_i^{(k+1)} = u \cdot \alpha_i^{(k)}$
4. pokud se změnilo znaménko  $\nabla_i E$ , proved':
  - (a) anuluj předchozí změnu vah
  - (b) nastav  $\alpha_i^{(k+1)} = \frac{\alpha_i^{(k)}}{d}$
  - (c) polož  $\Delta w_i^{(t+1)} = 0$
5. přejdi ke Kroku 2

#### 5.2.3.4 Algoritmy 2. řádu

Algoritmy, které berou v úvahu i **druhou derivaci** chybové funkce, tj. nejen gradient, ale i zakřivení. Používá se **kvadratická approximace** chybové funkce pomocí Taylorovy řady:

$$E(\mathbf{w} + \mathbf{h}) \approx E(\mathbf{w}) + \nabla E(\mathbf{w})^T \mathbf{h} + \frac{1}{2} \mathbf{h}^T \nabla^2 E(\mathbf{w}) \mathbf{h}$$

Člen  $\nabla^2 E(\mathbf{w})$  je *Hessovská matice* ( $n \times n$ ) parciálních derivací druhého řádu:

$$\nabla^2 E(\mathbf{w}) = \begin{pmatrix} \frac{\partial^2 E(\mathbf{w})}{\partial w_1^2} & \frac{\partial^2 E(\mathbf{w})}{\partial w_1 \partial w_2} & \dots & \frac{\partial^2 E(\mathbf{w})}{\partial w_1 \partial w_n} \\ \frac{\partial^2 E(\mathbf{w})}{\partial w_2 \partial w_1} & \frac{\partial^2 E(\mathbf{w})}{\partial w_2^2} & \dots & \frac{\partial^2 E(\mathbf{w})}{\partial w_2 \partial w_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 E(\mathbf{w})}{\partial w_n \partial w_1} & \frac{\partial^2 E(\mathbf{w})}{\partial w_n \partial w_2} & \dots & \frac{\partial^2 E(\mathbf{w})}{\partial w_n^2} \end{pmatrix}$$

Zderivováním dostaneme gradient chybové funkce:

$$\nabla E(\mathbf{w} + \mathbf{h})^T \approx \nabla E(\mathbf{w})^T + \mathbf{h}^T \nabla^2 E(\mathbf{w}) \mathbf{h}$$

Jelikož chceme, aby gradient byl nulový (hledáme minimum  $E$ ) můžeme odvodit:

$$\mathbf{h} = -(\nabla^2 E(\mathbf{w}))^{-1} \nabla E(\mathbf{w})$$

**Varianty:**

**newtonovské metody** Využívají přímo předchozího vzorce, iterativně upravují váhy podle

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - (\nabla^2 E(\mathbf{w}))^{-1} \nabla E(\mathbf{w})$$

Tyto metody rychle konvergují, ale problémem bývá výpočet inverzní Hessovské matice.

**pseudonewtonovské metody** Pracují se zjednodušenou approximací Hessovské matice. V úvalu se berou pouze prvky na diagonále, tj.  $\frac{\partial^2 E(\mathbf{w})}{\partial w_i^2}$ , ostatní prvky matice jsou vynulovány. Váhy se pak adaptují podle

$$w_i^{(k+1)} = w_i^{(k)} - \frac{\nabla_i E(\mathbf{w})}{\frac{\partial^2 E(\mathbf{w})}{\partial w_i^2}}$$

Odpadá potřeba inverze Hessovské matice. Tyto metody dobře fungují, má-li chybová funkce kvadratický tvar, jinak mohou nastat problémy.

### Algoritmy:

1. **Quickprop:** Minimalizační kroky se provádí pouze v jedné dimenzi. Váhy se adaptují podle

$$w_i^{(k+1)} = w_i^{(k)} + \Delta^{(k)} w_i$$

kde

$$\Delta^{(k)} w_i = \Delta^{(k-1)} w_i \cdot \frac{\nabla_i E^{(k)}}{\nabla_i E^{(k-1)} - \nabla_i E^{(k)}}$$

Pozorování: předchozí vzorec lze přepsat na

$$\Delta^{(k)} w_i = \frac{\nabla_i E^{(k)}}{\frac{\nabla_i E^{(k-1)} - \nabla_i E^{(k)}}{\Delta^{(k-1)} w_i}}$$

kde jmenovatel je vlastně diskrétní approximací  $\frac{\partial^2 E(\mathbf{w})}{\partial w_i^2}$ . Quickprop tedy používá tzv. **sekantové kroky** (sekant = sečna)

#### 5.2.3.5 Relaxační metody

**perturbace vah** Učení pomocí **perturbace vah** je motivováno tím, že chceme se vyhnout počítání gradientu chybové funkce v každém kroku. Základní myšlenkou je porovnat hodnotu chybové funkce pro aktuální váhy, tj.  $E(\mathbf{w})$  a hodnotu chybové funkce pro mírně změněné váhy (tj. po přičtení malé perturbace  $\beta$ ), tj.  $E(\mathbf{w}')$ . Váhy se aktualizují postupně (vždy náhodně zvolená váha) pomocí

$$\Delta w_i = -\alpha \frac{E(\mathbf{w}') - E(\mathbf{w})}{\beta}$$

**perturbace výstupu neuronu** Alternativním přístupem je perturbace výstupu neuronu. Nechť  $o_i$  je výstup  $i$ -tého neuronu, který chceme perturbovat o  $\Delta o_i$ . Pokud je  $E - E'$  kladné (kde  $E'$  je chyba pro nový, perturbovaný výstup), pak aplikujeme perturbaci – to znamená spočítat nové váhy do neuronu  $i$ . Chceme-li výstup  $(o_i + \Delta o_i)$ , pak potřebujeme potenciál neuronu nastavit na

$$\sum_{k=1}^m w'_k x_k = s^{-1}(o_i + \Delta o_i)$$

kde  $s^{-1}$  je inverze přenosové funkce (pro sigmoidu je  $s^{-1}(y) = \ln \frac{y}{1-y}$ ). Nové váhy jsou tedy dány jako

$$w'_k = w_k \cdot \frac{s^{-1}(o_i + \Delta o_i)}{\sum_{j=1}^m w_j x_j}$$

Pozorování: váhy se adaptují proporcionálně ke své velikosti. Tomu lze zamezit zavedením stochastických faktorů a nebo kombinací s perturbací vah.

### 5.2.3.6 Předzpracování trénovací množiny

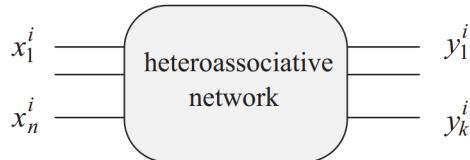
## 5.3 Asociativní paměti, Hebbovské učení a hledání suboptimálních řešení, stochastické modely.

### 5.3.1 Asociativní paměti

Principem asociativních sítí (a pamětí) je mapování vstupních vektorů  $\mathbf{x}$  na dané výstupní vzory  $\mathbf{y}$ , přičemž blízké okolí  $\mathbf{x}$  by se mělo mapovat na týž cílový vzor – ten se tedy chová jako „atraktor“. Asociativní paměti by tedy měly být schopné správně přiřadit vzor i zašuměným vstupům.

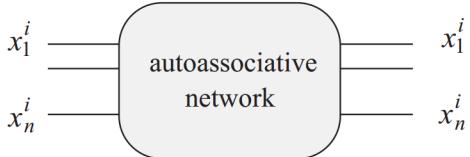
Rozlišujeme 3 základní typy asociativních sítí:

**heteroasociativní síť** Zobrazují  $m$  vstupních vzorů  $\mathbf{x}^1, \mathbf{x}^2, \dots, \mathbf{x}^m$  z  $n$ -rozměrného prostoru na  $m$  výstupních vektorů  $\mathbf{y}^1, \mathbf{y}^2, \dots, \mathbf{y}^m$  v  $k$ -rozměrném prostoru,  $\mathbf{x}^i \mapsto \mathbf{y}^i$ .

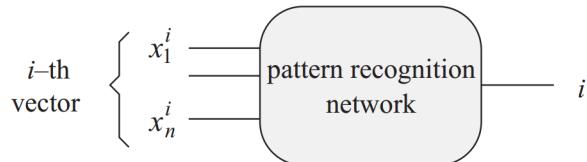


Jestliže pro nějaký vektor  $\mathbf{x}$  platí, že  $\|\mathbf{x} - \mathbf{x}^i\| < \varepsilon$ , potom  $\mathbf{x} \mapsto \mathbf{y}^i$  (pro zvolené  $\varepsilon > 0$ ).

**autoasociativní síť** Podmnožina heteroasociativních sítí: každý vzor se zobrazuje sám na sebe, tj.  $\mathbf{y}^i = \mathbf{x}^i \quad \forall i = 1, \dots, m$ . Funkcí těchto sítí je oprava zašuměných vzorů.



**sítě pro rozpoznávání vzorů** Speciální typ heteroasociativních sítí, kde je každému vektoru  $\mathbf{x}^i$  přiřazena skalární hodnota  $i$ , která reprezentuje třídu daného vzoru. Cílem je identifikace třídy daného vstupního vzoru.



### 5.3.1.1 Struktura asociativních pamětí

Asociativní paměť lze implementovat pomocí jedné vrstvy neuronů (resp. jedné vstupní a jedné výstupní). Síť má  $n$  vstupních a  $k$  výstupních neuronů. Označíme  $w_{ij}$  váhu mezi neurony  $i$  a  $j$ . Pak  $\mathbf{W}$  bude **matice vah** o rozměrech  $n \times k$ . Vektor potenciálů výstupních neuronů označíme jako **excitační vektor**  $\mathbf{e} = \mathbf{x} \cdot \mathbf{W}$ . V případě identické přenosové funkce na výstupních neuronech pak dostáváme *lineární asociátor* a výstup sítě spočteme jako  $\mathbf{y} = \mathbf{x} \cdot \mathbf{W}$ .

Nechť  $\mathbf{X}$  je  $m \times n$  matice vstupních vzorů a  $\mathbf{Y}$  je  $m \times k$  matice výstupních vzorů. Pak lze problém vyjádřit maticově: hledáme  $\mathbf{W}$  tak, aby  $\mathbf{X} \cdot \mathbf{W} = \mathbf{Y}$  (v případě autoasociativní paměti  $\mathbf{X} \cdot \mathbf{W} = \mathbf{X}$ ). Pokud je  $\mathbf{X}$  čtvercová a regulární, lze řešení najít jednoduše jako  $\mathbf{W} = \mathbf{X}^{-1} \cdot \mathbf{Y}$ . Obecně to ale samozřejmě platit nemusí.

### 5.3.1.2 Rekurentní asociativní síť

Než se budeme zabývat učením asociativních sítí, podíváme se ještě podrobněji na problém autoasociativních pamětí a jak jej lze popsát. Autoasociativní sítě jsou rekurentní, tj. z výstupních neuronů vedou spoje zpět do vstupních a výstup sítě v čase  $t$  se použije jako vstup sítě v čase  $t+1$ .

Toto se opakuje, dokud se výstup sítě neustálí, tj. dokud  $\mathbf{x}(t+1) = \mathbf{x}(t)$ . Vzhledem k tomu, že váhová matici  $\mathbf{W}$  je u autoasociativních sítí čtvercová, je problém stabilizace sítě ve skutečnosti otázkou nalezení *vlastního vektoru* matice  $\mathbf{W}$  s vlastním číslem 1, tj.

$$\vec{\xi} \cdot \mathbf{W} = \vec{\xi}$$

Jedná se vlastně o dynamický systém prvního rádu, jelikož stav  $\mathbf{x}(t+1)$  je zcela určen stavem  $\mathbf{x}(t)$ .

### 5.3.1.3 Vlastní automaty (eigenvector automata)

Podívejme se nyní na vlastnosti čtvercové matice  $\mathbf{W}$ . Jak jsme již řekli, zajímají nás pevné body dynamického systému, tedy vlastní vektory matice. Samozřejmě ale ne všechny matice mají netriviální vlastní vektory. Nás budou zajímat pouze takové, které mají dokonce kompletní sadu  $n$  lineárně nezávislých vlastních vektorů. Připomeneme, že pro vlastní vektory  $x^1, x^2, \dots, x^n$  platí

$$\mathbf{x}^i \mathbf{W} = \lambda_i x^i \quad \forall i = 1 \dots n$$

kde  $\lambda_i$  jsou vlastní čísla.

Každá matice s kompletní sadou vlastních vektorů definuje jakýsi „vlastní automat“. Ten funguje tak, že po předložení libovolného vzoru nalezne vlastní vektor s největším vlastním číslem. BÚNO  $\lambda_1$  je největší vlastní číslo. Nechť  $\lambda_1 > 0$  a nechť  $\mathbf{a}_0$  je nějaký nenulový  $n$ -rozměrný vektor. Ten můžeme vyjádřit jako lineární kombinaci  $n$  nezávislých vektorů, konkrétně vlastních vektorů  $\mathbf{W}$ :

$$\mathbf{a}_0 = \alpha_1 \mathbf{x}^1 + \alpha_2 \mathbf{x}^2 + \dots + \alpha_n \mathbf{x}^n$$

Předpokládáme že konstanty  $\alpha_i$  jsou nenulové. Po jedné iteraci s maticí  $\mathbf{W}$  dostáváme:

$$\begin{aligned} \mathbf{a}_1 &= \mathbf{a}_0 \mathbf{W} \\ &= (\alpha_1 \mathbf{x}^1 + \alpha_2 \mathbf{x}^2 + \dots + \alpha_n \mathbf{x}^n) \mathbf{W} \\ &= \alpha_1 \lambda_1 \mathbf{x}^1 + \alpha_2 \lambda_2 \mathbf{x}^2 + \dots + \alpha_n \lambda_n \mathbf{x}^n \end{aligned}$$

Po  $t$  iteracích dostáváme

$$\mathbf{a}_t = \alpha_1 \lambda_1^t \mathbf{x}^1 + \alpha_2 \lambda_2^t \mathbf{x}^2 + \dots + \alpha_n \lambda_n^t \mathbf{x}^n$$

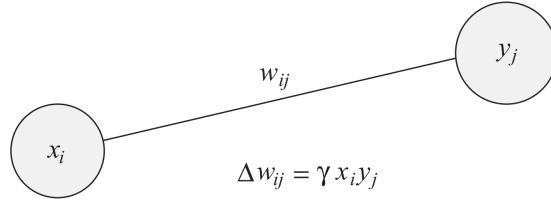
Je zjevné, že pro dost velké  $t$  bude člen s  $\lambda_1$  dominovat a vektor  $\mathbf{a}_t$  můžeme dostat libovolně blízko k  $\mathbf{x}^1$  (ve smyslu směru, ne nutně délky). Vlastní vektor  $\mathbf{x}^1$  s největším vlastním číslem  $\lambda_1$  je *atraktorem* pro všechny vektory  $a_0$ , které mají **nenulovou složku**  $\alpha_1$ .

### 5.3.2 Hebbovské učení a hledání suboptimálních řešení

Automat popsaný v předchozí části vystihuje to, čeho chceme dosáhnout: chceme použít asociativní síť jako dynamický systém, jehož atraktory jsou právě ty vektory, které si chceme uložit. Problémem ovšem je, že případě lineárního vlastního automatu je pouze jeden jediný vektor, který dominuje téměř celý vstupní prostor – to je vlastní vektor odpovídající největšímu vlastnímu číslu. My bychom ale chtěli použít co nejvíce atraktorů, s vlivem rovnoměrně rozprostřeným po vstupním prostoru. Řešením je použít nelineární systém, tj. ve výstupních neuronech použít místo identity binární (0,1) nebo bipolární (-1, 1) skokovou přenosovou funkci. Bipolární je častější, neboť bipolární vektory mají větší šanci být ortogonální, než binární.

#### 5.3.2.1 Hebbovské učení

Mějme 1-vrstvou síť s  $k$  neurony a skokovou přenosovou funkcí *sgn*. Chceme nalézt váhy pro zobrazení  $n$ -rozměrného vektoru  $\mathbf{x}$  na  $k$ -rozměrný vektor  $\mathbf{y}$ . **Hebbovské** je založeno na jednoduchém principu: „Dva neurony, které jsou současně aktivní, by měly mít vyšší stupeň vzájemné interakce než neurony, jejichž aktivita je nekorelovaná – v takovém případě by měla být vzájemná interakce hodně malá nebo nulová“ (*what fires together wires together*).



Hebbovské učení aktualizuje váhy podle pravidla

$$\Delta w_{ij} = \gamma x_i y_j$$

kde  $\gamma$  je parametr učení. Během učení se zapojí jak vstup  $\mathbf{x}$ , tak očekávaný výstup  $\mathbf{y}$ , a podle předchozího pravidla se aktualizuje váhová matice  $\mathbf{W}$  (ta je na začátku učení nulová).  $\mathbf{W}$  je ve skutečnosti **korelační maticí** pro tyto dva vektory a má tvar

$$\mathbf{W} = [w_{ij}]_{n \times k} = [x_i y_j]_{n \times k}$$

Lze ukázat, že  $\mathbf{W}$  zobrazí nenulový vektor  $\mathbf{x}$  právě na  $\mathbf{y}$ :

$$\mathbf{x} \cdot \mathbf{W} = \left( y_1 \sum_{i=1}^n x_i x_i, y_2 \sum_{i=1}^n x_i x_i, \dots, y_k \sum_{i=1}^n x_i x_i \right) = \mathbf{y}(\mathbf{x} \cdot \mathbf{x})$$

Pro  $\mathbf{x} \neq \mathbf{0}$  platí  $\mathbf{x} \cdot \mathbf{x} > 0$  a výstup sítě je

$$\text{sgn}(\mathbf{x} \cdot \mathbf{W}) = (y_1, y_2, \dots, y_k) = \mathbf{y}$$

kde  $\text{sgn}$  je aplikováno na všechny složky excitačního vektoru zvlášť.

Popsali jsme situaci pro jeden vstupní vektor  $\mathbf{x}$  a jeden odpovídající výstupní vektor  $\mathbf{y}$ . Nyní uvažme  $m$   $n$ -rozměrných vektorů  $\mathbf{x}^1, \mathbf{x}^2, \dots, \mathbf{x}^m$ , kterým chceme asociovat  $m$   $k$ -rozměrných výstupních vektorů  $\mathbf{y}^1, \mathbf{y}^2, \dots, \mathbf{y}^m$ . Pro každý pár  $\mathbf{x}^p, \mathbf{y}^p$  spočteme odpovídající korelační matici  $\mathbf{W}^p = [x_i^p y_j^p]_{n \times k}$  a celková matice vah pak bude

$$\mathbf{W} = \mathbf{W}^1 + \mathbf{W}^2 + \dots + \mathbf{W}^m$$

Pro vstup  $\mathbf{x}^p$  pak dostaneme výstup

$$\mathbf{x}^p \cdot \mathbf{W} = \mathbf{x}^p \cdot (\mathbf{W}^1 + \mathbf{W}^2 + \dots + \mathbf{W}^m) = \mathbf{x}^p \cdot \mathbf{W}^p + \sum_{l \neq p} \mathbf{x}^p \cdot \mathbf{W}^l = \mathbf{y}^p \cdot (\mathbf{x}^p \cdot \mathbf{x}^p) + \sum_{l \neq p} \mathbf{y}^l (\mathbf{x}^l \cdot \mathbf{x}^p)$$

Excitační vektor tedy odpovídá  $\mathbf{y}^p$  (vynásobenému kladnou konstantou) s perturbačním členem

$$\sum_{l \neq p} \mathbf{y}^l (\mathbf{x}^l \cdot \mathbf{x}^p)$$

který se označuje jako **crosstalk**. Síť dává na výstupu požadovaný vektor  $\mathbf{y}^p$  v případě, že je crosstalk nulový, tzn. pokud jsou vstupní vzory  $\mathbf{x}^1, \dots, \mathbf{x}^m$  navzájem ortogonální. Avšak i pokud není crosstalk nulový, může síť dávat dobré výsledky, stačí, aby byl crosstalk menší než  $\mathbf{y}^p(\mathbf{x}^p \cdot \mathbf{x}^p)$

Výstupem sítě tedy bude

$$\text{sgn}(\mathbf{x}^p \cdot \mathbf{W}) = \text{sgn} \left( \mathbf{y}^p \cdot (\mathbf{x}^p \cdot \mathbf{x}^p) + \sum_{l \neq p} \mathbf{y}^l (\mathbf{x}^l \cdot \mathbf{x}^p) \right)$$

Jelikož je  $\mathbf{x}^p \cdot \mathbf{x}^p$  kladná konstanta, platí

$$\text{sgn}(\mathbf{x}^p \cdot \mathbf{W}) = \text{sgn} \left( \mathbf{y}^p + \sum_{l \neq p} \mathbf{y}^l \cdot \frac{(\mathbf{x}^l \cdot \mathbf{x}^p)}{(\mathbf{x}^p \cdot \mathbf{x}^p)} \right)$$

Aby byl tento výstup roven  $\mathbf{y}^p$ , musí být absolutní hodnota všech složek perturbačního členu  $\sum_{l \neq p} \mathbf{y}^l \cdot \frac{(\mathbf{x}^l \cdot \mathbf{x}^p)}{(\mathbf{x}^p \cdot \mathbf{x}^p)}$  menší než 1. Pro bipolární vektory to znamená, že skalární součin  $(\mathbf{x}^l \cdot \mathbf{x}^p)$  musí být menší než druhá mocnina délky  $\mathbf{x}^p$ . Pokud jsou náhodně zvoleným bipolárním vektorům přiřazeny (jiné) náhodně zvolené bipolární vektory, je pravděpodobnost, že budou navzájem ortogonální, poměrně vysoká (pokud jich ovšem nebylo zvoleno příliš mnoho). V takovém případě bude crosstalk malý a Hebbovské učení povede k volbě vhodných vah pro asociativní síť.

### 5.3.2.2 Geometrická interpretace Hebbovského učení

V případě autoasociativních sítí se nabízí hezká geometrická interpretace. Pro autoasociativní sítě jsou matice  $\mathbf{W}^i = (\mathbf{x}^i)^T \mathbf{x}^i$ . Pak pro vstup  $\mathbf{z}$  platí

$$\mathbf{z} \cdot \mathbf{W}^i = \mathbf{z}(\mathbf{x}^i)^T \mathbf{x}^i = c_i \cdot \mathbf{x}^i$$

kde  $c_i$  je konstanta. Tedy  $\mathbf{z}$  je zobrazen do lineárního podprostoru  $L_i$  určeného vektorem  $\mathbf{x}^i$ . Matice  $\mathbf{W} = \mathbf{W}^1 + \dots + \mathbf{W}^m$  tedy zobrazí vektor  $\mathbf{z}$  do lineárního podprostoru určeného vektory  $\mathbf{x}^1, \dots, \mathbf{x}^m$ , protože

$$\mathbf{z} \cdot \mathbf{W} = \mathbf{z} \cdot \mathbf{W}^1 + \dots + \mathbf{z} \cdot \mathbf{W}^m = c_1 \mathbf{x}^1 + \dots + c_m \mathbf{x}^m$$

Obecně se jedná o neortogonální projekci.

### 5.3.2.3 Problém kapacity sítě

S rostoucím počtem uložených vzorů se jejich *sféra vlivu* zmenšuje, a to poměrně rychle. Jedním s důvodů je i to, že spolu s vektorem  $\mathbf{x}$  je implicitně „uložen“ i vektor  $-\mathbf{x}$ , což vede k *nepravým stabilním stavům*.

Může se dokonce stát, že dříve uložený vzor bude zapomenut, je-li crosstalk příliš velký. To je samozřejmě nežádoucí. Je proto třeba mít nějaký odhad na počet vzorů  $m$ , které lze „bezpečně“ uložit do autoasociativní sítě s váhovou maticí velikosti  $n \times n$ . Hodnotu  $m$  označíme jako **maximální kapacitu sítě** a její hodnota je přibližně<sup>1</sup>

$$m \sim 0.18n$$

Tedy počet uložených vzorů by neměl přesáhnout  $0.18n$  pro  $n$ -rozměrné vstupní vektory. Pro korelované vzory může dojít k problému i pro nižší hodnoty.

### 5.3.2.4 Pseudoinverzní matice

Hebbovské učení funguje dobře, pokud jsou uložené vzory téměř ortogonální, což pro  $m$  náhodných  $n$ -rozměrných bipolárních vektorů platí, pokud  $m \ll n$ . V praxi jsou však vzory téměř vždy korelované a perturbační člen může ovlivnit kvalitu rozpoznávání, protože skalární součiny  $\mathbf{x}^l \cdot \mathbf{x}^p$ ,  $l \neq p$  nejsou dost malé. Takováto korelace výrazně snižuje kapacitu sítě. Uvážíme-li například problém rozpoznávání rukou psaných číslic, digitalizovaných na mřížky  $16 \times 16$ . Vzory nejsou rovnoměrně rozprostřeny po prostoru, jsou naopak koncentrované v poměrně malé oblasti (protože číslice mají hodně podobné tvary).

Jedním z alternativních přístupů, které se snaží výše popsaný problém řešit, je použití **pseudoinverzní matice** namísto korelační. Nejprve formalizace:

Nechť  $\mathbf{x}^1, \dots, \mathbf{x}^m$  jsou  $n$ -rozměrné vektory, kterým chceme přiřadit  $m$   $k$ -rozměrných vektorů  $\mathbf{y}^1, \dots, \mathbf{y}^m$ . Matice  $\mathbf{X}$  je matice  $m \times n$ , jejíž řádky jsou vstupní vektory, a  $\mathbf{Y}$  je matice  $m \times k$ , jejíž řádky jsou výstupní vektory. Hledáme váhovou matici  $\mathbf{W}$  takovou, že

$$\mathbf{XW} = \mathbf{Y}$$

Jelikož obecně  $m \neq n$  vstupní vektory nejsou nutně lineárně nezávislé, nemusí existovat inverzní matice k  $\mathbf{X}$ . Můžeme nicméně najít takovou matici  $\mathbf{W}$ , která minimalizuje  $\|\mathbf{XW} - \mathbf{Y}\|$ .

**Pseudoinverzní matice** k matici  $\mathbf{X}$  reálných čísel o rozměrech  $m \times n$  je matice  $\tilde{\mathbf{X}}$ , pro níž platí:

1.  $\mathbf{X} \tilde{\mathbf{X}} \mathbf{X} = \mathbf{X}$
2.  $\tilde{\mathbf{X}} \mathbf{X} \tilde{\mathbf{X}} = \tilde{\mathbf{X}}$
3.  $\tilde{\mathbf{X}} \mathbf{X}$  a  $\mathbf{X} \tilde{\mathbf{X}}$  jsou symetrické

<sup>1</sup> Odvození maximální kapacity lze nalézt ve slajdech nebo v Rojasovi. Základní myšlenka je vyjádřit crosstalk pro autoasociativní sítě a odhadnout pravděpodobnost, že nějaký člen bude větší než 1. Jedná se o binomiální rozdělení, které lze approximovat normálním rozdělením a pak lze numericky dojít k výsledku 0.18n.

Lze ukázat, že pseudoinverzní matice vždy existuje a je jednoznačně určena. Pokud existuje  $\mathbf{X}^{-1}$ , pak  $\mathbf{X}^{-1} = \tilde{\mathbf{X}}$ .

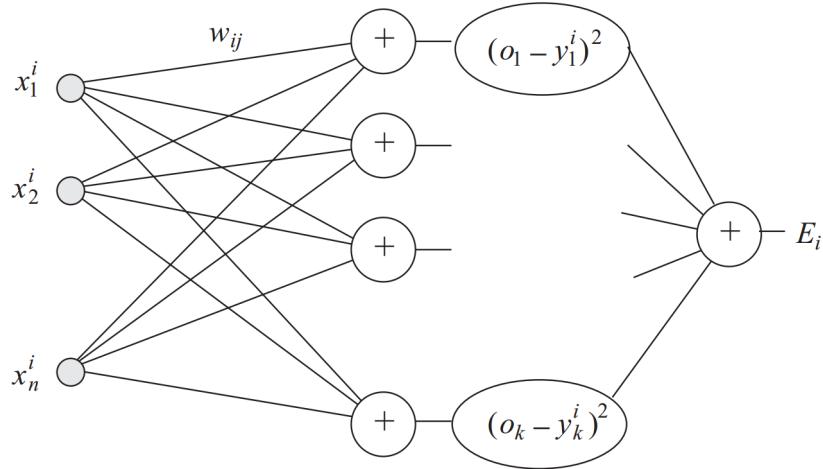
**Věta 5.3.1.** Nechť  $\mathbf{X}$  je reálná matice  $m \times n$  a  $\mathbf{Y}$  je reálná matice  $m \times k$ . Pak reálná matice  $\mathbf{W} = \tilde{\mathbf{X}}\mathbf{Y}$  minimalizuje  $\|\mathbf{X}\mathbf{W} - \mathbf{Y}\|$ .

*Důkaz.* Je docela technický a lze jej najít ve slajdech či v Rojasovi.  $\square$

Důsledkem mj. je, že  $\tilde{\mathbf{X}}$  minimalizuje  $\|\mathbf{X}\tilde{\mathbf{X}} - I\|$ , tedy je „nejlepší aproximací“ inverzní matice.

Experimenty ukazují, že pseudoinverzní matice si vede lépe než Hebbovské učení, jsou-li vzory korelovány.

Pro výpočet pseudoinverzní matice lze využít neuronové sítě se zpětným šířením. Síť je rozšířena o vrstvu, v níž je spočtený výstup porovnán s očekávaným výstupem (po složkách), výstupem je součet chyby na všech složkách. Celková chyba na všech předložených vzorech  $E = E_i$  odpovídá kvadratické normě matice  $\mathbf{X}\mathbf{W} - \mathbf{Y}$ . Algoritmus zpětného šíření (backpropagation) najde matici, která bude minimalizovat  $E$ .



### 5.3.2.5 BAM (Bidirectional Associative Memory)

Model skládající se ze 2 vrstev neuronů, které si mezi sebou rekurzivně posílají informace. Vstupní vrstva posílá výsledky svého výpočtu výstupní vrstvě klasicky prostřednictvím vah sítě. Výstupní vrstva vrací výsledky svých výpočtů zpět vstupní vrstvě **prostřednictvím stejných vah**. Tento model se také označuje jako **rezonanční síť**. Jako u všech asociativních sítí i zde nás zajímá, zda existuje stabilní stav.

Síť zobrazí  $n$ -rozměrný vektor  $\mathbf{x}_0$  na  $k$ -rozměrný vektor  $\mathbf{y}_0$  pomocí váhové matice  $\mathbf{W}$  o rozměrech  $n \times k$ . Přenosovou funkcí je opět  $sgn$ , používají se bipolární vektory. Po prvním průchodu dostaneme

$$\mathbf{y}_0 = \text{sgn}(\mathbf{x}_0 \mathbf{W})$$

Po zpětném průchodu

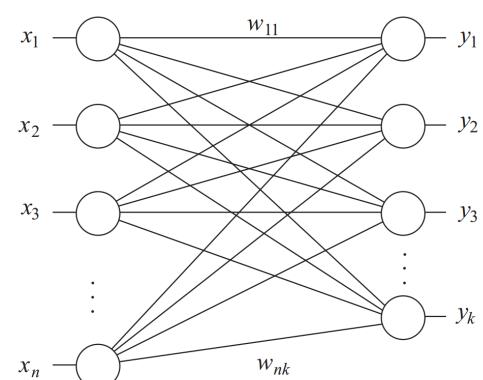
$$\mathbf{x}_1^T = \text{sgn}(\mathbf{W} \mathbf{y}_0^T)$$

Po  $m$  iteracích dostaneme  $(m + 1)$  dvojic vektorů  $(\mathbf{x}_0, \mathbf{y}_0), \dots, (\mathbf{x}_m, \mathbf{y}_m)$ , pro něž platí

$$\mathbf{y}_i = \text{sgn}(\mathbf{x}_i \mathbf{W}) \quad \mathbf{x}_{i+1}^T = \text{sgn}(\mathbf{W} \mathbf{y}_i^T) \quad (5.3.1)$$

Pro pevný bod (stabilní stav)  $(\mathbf{x}, \mathbf{y})$  bude tedy platit

$$\mathbf{y} = \text{sgn}(\mathbf{x} \mathbf{W}) \quad \mathbf{x}^T = \text{sgn}(\mathbf{W} \mathbf{y}^T)$$



Chceme-li nastavit BAM tak, aby  $(\mathbf{x}, \mathbf{y})$  bylo pevným bodem (=atraktorem), můžeme k tomu opět použít Hebbovské učení. To určuje  $\mathbf{W} = \mathbf{x}^T \mathbf{y}$  a pak tedy platí

$$\mathbf{y} = \text{sgn}(\mathbf{x}\mathbf{W}) = \text{sgn}(\mathbf{x}\mathbf{x}^T \mathbf{y}) = \text{sgn}(\|\mathbf{x}\|^2 \mathbf{y}) = \mathbf{y}$$

a zároveň

$$\mathbf{x}^T = \text{sgn}(\mathbf{W}\mathbf{y}^T) = \text{sgn}(\mathbf{x}^T \mathbf{y}\mathbf{y}^T) = \text{sgn}(\mathbf{x}^T \|\mathbf{y}\|^2) = \mathbf{x}^T$$

Chceme-li uložit více vzorů, bude Hebbovské učení fungovat lépe, pokud jsou vstupní vektory navzájem ortogonální a výstupní rovněž. Pro  $m$  dvojic vektorů, které chceme uložit, bude váhová matice mít tvar

$$\mathbf{W} = \mathbf{x}_1^T \mathbf{y}_1 + \mathbf{x}_2^T \mathbf{y}_2 + \cdots + \mathbf{x}_m^T \mathbf{y}_m$$

BAM lze použí pro konstrukci autoasociativních sítí, protože matice vah vytvořené Hebbovským učením jsou symetrické.

### 5.3.2.6 Energetická funkce BAM

Nechť pro danou síť BAM představuje dvojice  $(\mathbf{x}, \mathbf{y})$  stabilní stav. Při inicializaci je síti (zleva) předložen vstupní vektor  $\mathbf{x}_0$  (časem by měla dokoncovať k  $(\mathbf{x}, \mathbf{y})$ ). Spočte se vektor  $\mathbf{y}_0 = \text{sgn}(\mathbf{x}_0 \mathbf{W})$  a ten se použije pro novou iteraci, tentokrát zprava. Nyní můžeme vyjádřit excitaci neuronů v levé (vstupní vrstvy) pomocí **excitačního vektoru e**:

$$\mathbf{e}^T = \mathbf{W}\mathbf{y}_0^T$$

Dvojice  $(\mathbf{x}_0, \mathbf{y}_0)$  by odpovídala stabilnímu stavu, pokud by  $\text{sgn}(\mathbf{e}) = \mathbf{x}_0$ , tj. pokud je  $\mathbf{e}$  dostatečně blízko  $\mathbf{x}_0$ . V takovém případě je skalární součin  $\mathbf{x}_0 \cdot \mathbf{e}$  větší než jiných vektorů, které mají stejnou délku, ale jsou dále (protože  $\mathbf{x}_0$  a  $\mathbf{e}$  svírají menší úhel). Díky tomu bude

$$E = -\mathbf{x}_0 \mathbf{e}^T = -\mathbf{x}_0 \mathbf{W}\mathbf{y}_0^T$$

menší, pokud  $\mathbf{W}\mathbf{y}_0^T$  leží blíže  $\mathbf{x}_0$ . Skalární hodnota  $E$  odpovídá *energetické funkci* sítě a umožňuje nám sledovat konvergenci ke stabilnímu stavu.

Formálně: Nechť  $\mathbf{W}$  je váhová matice BAM a nechť se v  $i$ -té iteraci spočte výstup  $\mathbf{y}_i$  pravé vrstvy neuronů a výstup  $\mathbf{x}_i$  levé vrstvy neuronů podle 5.3.1. Pak **energetická funkce** sítě BAM se spočte pomocí

$$E(\mathbf{x}_i, \mathbf{y}_i) = -\frac{1}{2} \mathbf{x}_i \mathbf{W} \mathbf{y}_i^T$$

Faktor  $\frac{1}{2}$  je jen škálovací konstanta, která se bude hodit později.

Doposud jsme uvažovali pouze neurony s přenosovou funkcí *signum*, avšak energetickou funkci lze zobecnit i pro neurony s *prahem* a *skokovou přenosovou funkcí*. Prahy neuronů lze simulovat přidáním neuronu s konstantním výstupem 1 do obou vrstev, přičemž váhy z tohoto neuronu odpovídají záporné hodnotě prahu neuronů, do nichž vedou. Formálně to realizujeme rozšířením každého vektoru  $\mathbf{x}$  na  $(x_1, x_2, \dots, x_n, 1)$ , analogicky rozšíříme i vektory  $\mathbf{y}$ . Váhová matice bude rozšířena o jeden řádek a sloupec, kde nový  $(n+1)$ -ní řádek je tvořen zápornými prahy neuronů z pravé vrstvy, nový  $(k+1)$ -ní sloupec tvoří záporné prahy neuronů z levé vrstvy, prvek na pozici  $(n+1, k+1)$  může být 0. Energetická funkce takto modifikované sítě bude

$$E(\mathbf{x}_i, \mathbf{y}_i) = -\frac{1}{2} \mathbf{x}_i \mathbf{W} \mathbf{y}_i^T + \frac{1}{2} \Theta_r \mathbf{y}_i^T + \frac{1}{2} \mathbf{x}_i \Theta_l^T$$

kde  $\Theta_l$  je vektor prahů neuronů v levé vrstvě a  $\Theta_r$  je vektor prahů neuronů v pravé vrstvě.

### 5.3.2.7 Asynchronní sítě BAM

V asynchronní sítí počítá každý neuron svou excitaci v náhodném čase a změní svůj stav na 1 nebo -1 nezávisle na ostatních (ale podle znaménka své excitace). Pravděpodobnost, že budou dva neurony současně měnit svůj stav, je nulová. Zjednodušení: stav neuronu se nemění, je-li celková excitace nulová (tj. ponecháme *signum* nedefinovanou v bode 0). Asynchronní sítě jsou obecně biologicky o něco realističtější než synchronní.

**Věta 5.3.2.** BAM s libovolnou maticí vah  $W$  dosáhne stabilního stavu v konečném počtu iterací – a to jak pomocí synchronní, tak také pomocí asynchronní aktualizace.

*Důkaz.* Pro  $\mathbf{x} = (x_1, \dots, x_n), \mathbf{y} = (y_1, \dots, y_k)$  a váhovou matici  $\mathbf{W} = \{w_{ij}\}$  o rozměrech  $n \times k$  je energetická funkce  $E(\mathbf{x}, \mathbf{y})$  rovna:

$$E(\mathbf{x}, \mathbf{y}) = -\frac{1}{2}(x_1, \dots, x_n) \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1k} \\ w_{21} & w_{22} & \cdots & w_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1} & w_{n2} & \cdots & w_{nk} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_k \end{pmatrix}$$

Součin  $i$ -té řádky  $\mathbf{W}$  a  $\mathbf{y}^T$  udává míru excitace  $i$ -tého neuronu z levé vrstvy, označme ji jako  $g_i$ . Pak lze psát

$$E(\mathbf{x}, \mathbf{y}) = -\frac{1}{2}(x_1, \dots, x_n) \begin{pmatrix} g_1 \\ g_2 \\ \vdots \\ g_k \end{pmatrix}$$

Analogicky pro pravou vrstvu a excitační vektor  $\mathbf{e}$ :

$$E(\mathbf{x}, \mathbf{y}) = -\frac{1}{2}(e_1, \dots, e_n) \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_k \end{pmatrix}$$

Energetickou funkci lze tedy vyjádřit dvěma způsoby:

$$E(\mathbf{x}, \mathbf{y}) = -\frac{1}{2} \sum_{i=1}^k e_i y_i = -\frac{1}{2} \sum_{i=1}^k g_i x_i$$

Stav neuronu  $i$  z levé vrstvy sítě se změní pouze v případě, že má excitace  $g_i$  jiné znaménko, než jeho aktuální stav  $x_i$ . Protože ostatní neurony svůj stav nemění (asynchronní dynamika), bude rozdíl mezi předchozí energií  $E(\mathbf{x}, \mathbf{y})$  a novou energií  $E(\mathbf{x}', \mathbf{y})$  odpovídat

$$E(\mathbf{x}, \mathbf{y}) - E(\mathbf{x}', \mathbf{y}) = -\frac{1}{2} g_i (x_i - x'_i)$$

Protože jak  $x_i$ , tak  $x'_i$  mají různé znaménko od  $g_i$ , má i  $(x_i - x'_i)$  jiné znaménko než  $g_i$  a tedy

$$E(\mathbf{x}, \mathbf{y}) - E(\mathbf{x}', \mathbf{y}) > 0$$

Nový stav sítě má tedy nižší energii. Pro neurony pravé sítě analogicky.

Každá aktualizace stavu sítě vede ke snížení energie. Protože existuje konečně mnoho kombinací bipolárních hodnot stavů, musí proces skončit v nějakém stavu, kdy už nelze energii sítě dále snižovat.

Synchronní sítě lze chápat jako speciální variantu asynchronních a platí pro ně totéž.  $\square$

### 5.3.2.8 Hopfieldův model

Hopfieldův model lze chápat jako speciální případ BAM, kdy jsou levá a pravá vrstva sloučeny do jedné; ačkoliv chronologicky byl vyvinut jako první.

**Hopfieldův model** sestává z  $n$  plně propojených neuronů (každý s každým) s přenosovou funkcí *signum*. Synaptické váhy jsou obousměrné,  $w_{ij} = w_{ji}$  a platí  $\forall i : w_{ii} = 0$  (tj. bez reflexivních spojů). Matice  $\mathbf{W}$  je čtvercová, symetrická a má nulovou diagonálu. Stav neuronů je zachován, dokud nejsou vybrány (náhodně) k aktualizaci.

Během trénovací fáze s  $m$  trénovacími vzory  $\mathbf{x}^1, \dots, \mathbf{x}^m$  se nastaví váhy sítě takto:

$$w_{ij} = \begin{cases} \sum_{s=1}^m x_i^s x_j^s & \text{pro } i \neq j \\ 0 & \text{pro } i = j \end{cases}$$

Při klasifikaci se síti předloží nový vzor, spočtou se excitace a iteruje se:

$$y_j(t+1) = \operatorname{sgn} \left( \sum_{i=1}^n w_{ij} y_i(t) \right) \quad 1 \leq j \leq n$$

Iterace pokračuje dokud se výstupy neuronů neustálí.

Po předložení vzoru  $\mathbf{x}^1$  bude vektor potenciálů sítě

$$\mathbf{y} = \mathbf{x}^1 \mathbf{W} = \mathbf{x}^1 \cdot ((\mathbf{x}^1)^T \mathbf{x}^1 + \dots + (\mathbf{x}^m)^T \mathbf{x}^m - m \mathbf{I}) = (n-m)\mathbf{x}^1 + \sum_{j=2}^m \alpha^{1j} \mathbf{x}^j$$

kde  $\alpha^{1j}$  je skalární součin  $\mathbf{x}^1 \cdot \mathbf{x}^j$ . Stav  $x^1$  je stabilní, jestliže  $m < n$  a perturbační člen  $\sum_{j=2}^m \alpha^{1j} \mathbf{x}^j$  je malý.

Příklad: Váhová matice s *nenulovou diagonálou* nemusí vést ke stabilním stavům:

$$\mathbf{W} = \begin{pmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$$

Pro synchronní sítě vede tato matice k oscilaci mezi  $(-1, -1, -1)$  a  $(1, 1, 1)$ . Pro asynchronní dynamiku se náhodně střídá 8 možných vzorů.

Příklad: *Nesymetrická* matice

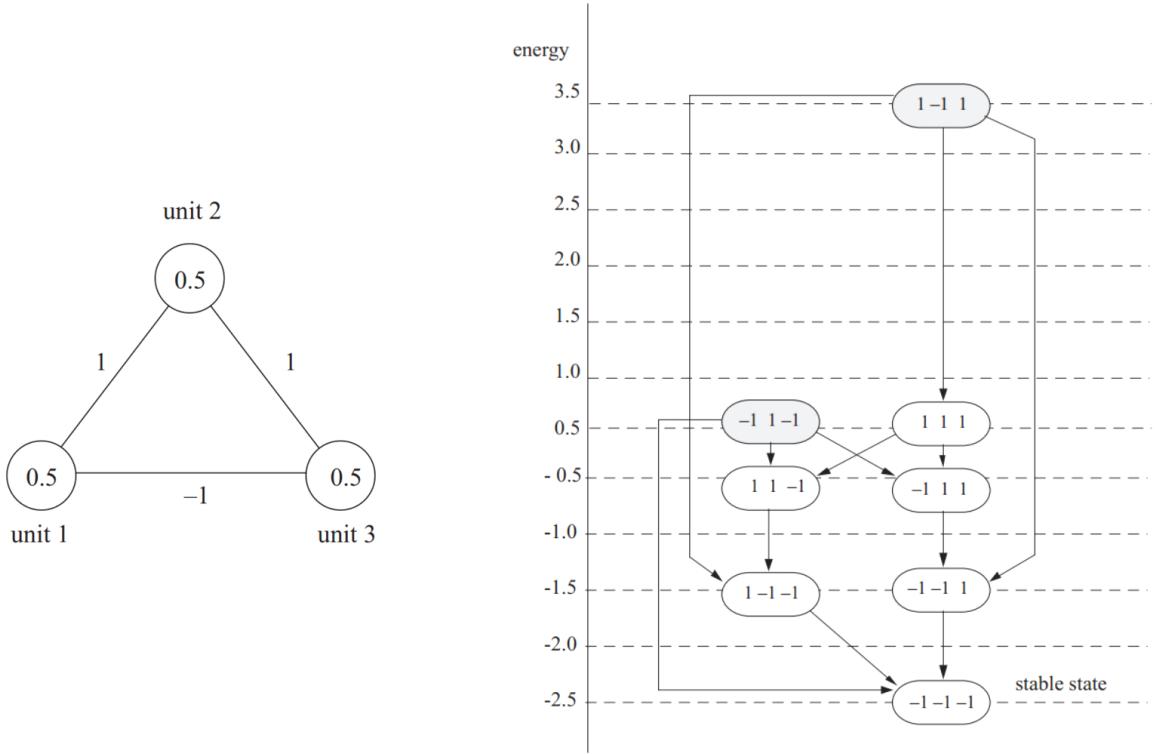
$$\mathbf{W} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$$

vede v případě asynchronní dynamiky k zacyklení.

### 5.3.2.9 Energetická funkce Hopfieldovy sítě

Nechť  $\mathbf{W}$  je váhová matice Hopfieldovy sítě s  $n$  neurony a  $\theta$  je  $n$ -rozměrný řádkový vektor prahů neuronů. **Energie**  $E(\mathbf{x})$  stavu  $\mathbf{x}$  je dána jako

$$E(x) = -\frac{1}{2} \mathbf{x} \mathbf{W} \mathbf{x}^T + \theta \mathbf{x}^T = -\frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n w_{ij} x_i x_j + \sum_{i=1}^n \theta_i x_i$$



Obrázek 5.1: Příklad Hopfieldovy sítě spolu s analýzou stavů, jejich energetických funkcí a konvergencí do stabilního stavu.

**Věta 5.3.3.** *Hopfieldova síť s asynchronní dynamikou dosáhne z libovolného počátečního stavu sítě stabilního stavu v lokálním minimu energetické funkce.*

*Důkaz.* Nechť je síť v (počátečním) stavu  $\mathbf{x} = (x_1, x_2, \dots, x_n)$  a nechť je k aktualizaci náhodně vybrán neuron  $k$ . Jestliže  $k$  nezmění svůj stav, zůstane  $E$  stejná. Jestliže  $k$  změní svůj stav, je nová energetická funkce  $E(\mathbf{x}')$  pro  $\mathbf{x}' = (x_1, \dots, x'_k, \dots, x_n)$ . Rozdíl energetických funkcí  $E(\mathbf{x}') - E(\mathbf{x})$  je dán rozdílem členů  $E$  obsahujících  $x_k$

$$\begin{aligned} E(\mathbf{x}') - E(\mathbf{x}) &= \left( -\frac{1}{2} \sum_{i=1}^n w_{ik} x_i x'_k - \frac{1}{2} \sum_{j=1}^n w_{kj} x'_k x_j + \theta_k x'_k \right) \\ &\quad - \left( -\frac{1}{2} \sum_{i=1}^n w_{ik} x_i x_k - \frac{1}{2} \sum_{j=1}^n w_{kj} x_k x_j + \theta_k x_k \right) \end{aligned}$$

což je díky symetrii vah

$$E(\mathbf{x}') - E(\mathbf{x}) = \left( - \sum_{i=1}^n w_{ik} x_i x'_k + \theta_k x'_k \right) - \left( \sum_{i=1}^n w_{ik} x_i x_k + \theta_k x_k \right)$$

a protože  $w_{kk} = 0$ , lze psát

$$E(\mathbf{x}') - E(\mathbf{x}) = -(x'_k - x_k) \left( \sum_{i=1}^n w_{ik} x_i - \theta_k \right) = -(x'_k - x_k) e_k$$

kde  $e_k$  je excitace neuronu  $k$  (po odečtení prahu). Jelikož došlo k aktualizaci, musí mít  $e_k$  odlišné znaménko od  $x_k$  a rovněž od  $-x'_k$ , a tedy  $E(\mathbf{x}') - E(\mathbf{x}) > 0$ .

Tedy vždy, když dojde k aktualizaci, dojde ke snížení energie. Jelikož máme jen konečný počet stavů, musí síť někdy dosáhnout stavu, z nějž už nelze energii dále snižovat = stabilní stav.  $\square$

Důležitá je nazávislost excitace neuronu na jeho stavu (protože  $w_{kk} = 0$ ).

### 5.3.2.10 Perceptronové učení pro Hopfieldův model

### 5.3.2.11 Hledání suboptimálních řešení (řešení NP-úplných problémů pomocí Hopfieldovy sítě)

Hopfieldovy sítě lze použít pro řešení těžkých optimalizačních úloh. Základní myšlenkou je přeforumulovat úlohu tak, aby její analytický zápis byl isomorfní energetické funkci nějakého Hopfieldova modelu. Ten pak naleze lokální minimum této funkce a tím i nějaké suboptimální řešení daného problému.

#### Příklady problémů a převodů:

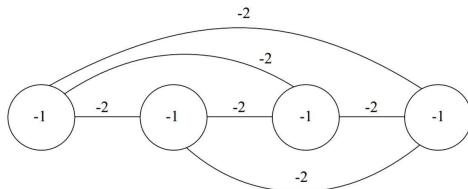
**multiflop** Cíl: nalézt vektor  $\mathbf{x}$ , jehož jediná složka bude rovna 1, ostatní budou nulové. To lze formulovat jako problém nalezení minima funkce

$$\begin{aligned} E(x_1, \dots, x_n) &= \left( \sum_{i=1}^n x_i - 1 \right)^2 \\ &= \sum_{i=1}^n x_i^2 + \sum_{i \neq j} x_i x_j - 2 \sum_{i=1}^n x_i + 1 \end{aligned}$$

Pro binární stavy platí  $x_i^2 = x_i$  a tedy

$$\begin{aligned} E(x_1, \dots, x_n) &= \sum_{i \neq j} x_i x_j - \sum_{i=1}^n x_i + 1 \\ &= -\frac{1}{2} \sum_{i \neq j} (-2) x_i x_j + \sum_{i=1}^n (-1) x_i + 1 \end{aligned}$$

Což je ekvivalentní síti s  $n$  neurony s prahy  $-1$  a se všemi váhami nastavenými na  $-2$ . Jedničku můžeme ignorovat, jelikož je to konstanta a tedy irrelevantní pro optimalizační problém.



**problém  $n$  věží** Cíl: umístit  $n$  věží na šachovnici  $n \times n$  tak, aby se navzájem neohrožovaly.

Definujme proměnné  $x_{ij}$ ,  $1 \leq i, j \leq n$  pro něž platí, že  $x_{ij} = 1$  pokud na poli  $[i, j]$  je věž,  $x_{ij} = 0$  pokud je pole prázdné. Musí platit, že na každém řádku je právě jedna jednička, tj.  $\forall i : \sum_{j=1}^n x_{ij} = 1$ ; a stejně tak v každém sloupci, tj.  $\forall j : \sum_{i=1}^n x_{ij} = 1$ . Budeme tedy chtít minimalizovat

$$E(x_{11}, \dots, x_{nn}) = \sum_{i=1}^n \left( \sum_{j=1}^n x_{ij} - 1 \right)^2 + \sum_{j=1}^n \left( \sum_{i=1}^n x_{ij} - 1 \right)^2$$

Což lze převést na energetickou funkci Hopfieldovy sítě. Lze si také všimnout, že jeden řádek/sloupec odpovídá multiflop problému. Výsledný model má všechny prahy  $-1$  a všechny spoje v rámci daného řádku/sloupce mají váhu  $-2$ .

**problém  $n$  královen** Podobné předchozímu problému, jen s královnami místo věží. Je třeba brát v potaz i diagonály. Lze řešit podobně jako věže s přidanými členy pro diagonály, nicméně se ukazuje, že tato varianta nemusí vždy vést k validnímu řešení. Funkce totiž může dosáhnout lokálního minima i s použitím méně než  $n$  královen. Problémem jsou diagonály, které královnu nemusí mít vůbec – proto již nefunguje řešení pomocí překrytí několika multiflop problémů.

**problém obchodního cestujícího** Cíl: Nalézt cestu přes  $n$  měst  $M_1, \dots, M_n$  tak, aby bylo každé město navštíveno alespoň jednou a délka „okružní jízdy“ byla minimální.

Cestu budeme reprezentovat pomocí matice  $n \times n$ , kde řádek  $i$  je asociován s městem  $M_i$  a sloupec  $j$  znamená  $j$ -té pořadí v cestě. Tj. jednička v buňce  $[i, j]$  znamená, že město  $M_i$  bude navštíveno jako  $j$ -té. Každý sloupec smí obsahovat nejvýše jednu 1, stejně tak každý řádek. Matice tedy odpovídá problému  $n$  věží. Navíc použijeme konvenci, že  $(n+1)$  sloupec je totožný s prvním, čímž dostaneme cyklus.

Budeme chtít minimalizovat následující energetickou funkci:

$$E = \frac{1}{2} \sum_{i,j,k}^n d_{ij} x_{ik} x_{jk+1} + \frac{\gamma}{2} \left( \sum_{i=1}^n \left( \sum_{j=1}^n x_{ij} - 1 \right)^2 + \sum_{j=1}^n \left( \sum_{i=1}^n x_{ij} - 1 \right)^2 \right)$$

První člen odpovídá minimalizaci délky cesty ( $d_{ij}$  je vzdálenost  $M_i, M_j$ ), zbytek jsou omezení na tvar matice totožné problému  $n$  věží. Parametr  $\gamma$  reguluje, jak velký důraz je kladen na minimalizaci cesty a jak velký na validnost cesty. Abychom zajistili validní řešení, musí být parametr  $\gamma$  velmi velký.

Z energetické funkce lze odvodit nastavení parametrů sítě. Prahy neuronů nastavíme na  $\frac{\gamma}{2}$ , váhy mezi neurony nastavíme na

$$w_{ik,j(k+1)} = \begin{cases} -d_{ij} - \gamma & \text{pro neurony ve stejné řadce či sloupci} \\ -d_{ij} & \text{jinak} \end{cases}$$

### 5.3.3 Stochastické modely

Hopfieldův model lze úspěšně použít k řešení optimalizačních problémů, avšak model má tendenci často zůstat v lokálním optimu. Mezi varianty, které se tento problém snaží řešit, patří

- **spojitý model:** Rozšíření na reálné stavy a sigmoidální přenosovou funkci. Vede ke zvětšení počtu možných cest k řešení ve stavovém prostoru.
- **zašumění sítě:** Principem je umožnit únik z lokálního optima zavedením šumu, který umožní aktualizaci sítě i za cenu zvýšení energetické funkce. Tyto stochastické modely probereme blíže.

#### 5.3.3.1 Simulované žíhání

Při minimalizaci energetické funkce  $E$  se tento jev simuluje následujícím způsobem:

- Hodnota proměnné  $x$  se změní vždy, když může aktualizace  $\Delta x$  zmenšit hodnotu energetické funkce  $E$
- Pokud by se při aktualizaci  $x$  naopak hodnota  $E$  zvýšila o  $\Delta E$ , bude nová hodnota  $x$  (tj.  $x + \Delta x$ ) přijata s pravděpodobností  $p_{\Delta E}$ :

$$p_{\Delta E} = \frac{1}{1 + e^{\frac{\Delta E}{T}}}$$

kde  $T$  je tzv. **teplotní konstanta**.

Pro velké  $T$  je  $p_{\Delta E} \approx \frac{1}{2}$ , tj. aktualizace nastane přibližně v polovině zhoršujících případů. Pro  $T = 0$  dojde k aktualizaci jen tehdy, pokud se  $E$  sníží. Postupná změna hodnot  $T$  z velmi vysokých hodnot směrem k nule odpovídá zahrátí a postupnému ochlazování v procesu žíhání

Navíc lze ukázat, že touto strategií lze dosáhnout (asymptoticky) globálního minima energetické funkce.

### 5.3.3.2 Boltzmannův stroj

**Boltzmannův stroj** je Hopfieldova síť, která se skládá z  $n$  neuronů se stavy  $x_1, x_2, \dots, x_n$ . Stav neuronu  $i$  se aktualizuje asynchronně podle pravidla:

$$x_i = \begin{cases} 1 & \text{s pravděpodobností } p_i \\ 0 & \text{s pravděpodobností } 1 - p_i \end{cases} \quad \text{kde} \quad p_i = \frac{1}{1 + \exp\left(-\frac{\left(\sum_{j=1}^n w_{ij} x_j - \theta_i\right)}{T}\right)}$$

kde  $T$  je kladná **teplotní konstanta**.

Stavy mohou být binární nebo bipolární, dále budeme uvažovat binární. Energetická funkce je totožná jako pro Hopfieldův model.

Je-li  $T$  hodně malé, pak  $p_i \approx 1$ , je-li aktivace  $\sum_{j=1}^n w_{ij} x_j - \theta_i > 0$ . Je-li aktivace záporná, pak  $p_i \approx 0$ . Boltzmannův stroj tedy aproxiimuje dynamiku Hopfieldovy sítě a konverguje do *lokálního optimu*.

Je-li  $T > 0$ , pak je pravděpodobnost přechodu do jiného stavu **vždy nenulová**, tj. neexistuje stabilní stav. Pro  $T \gg 0$  projde síť v podstatě celý stavový prostor.

Pokud se teplota snižuje správným způsobem, můžeme očekávat, že systém dosáhne globálního minima s pravděpodobností 1

## 5.4 Umělé neuronové sítě založené na principu učení bez učitele.

Nejprve stručně rozebereme *učení bez učitele* obecně, poté se dostaneme ke konkrétním typům sítí.

Základní principy učení bez učitele jsou **samoorganizace** a **shlukování**. Jinými slovy, nemáme předepsaný očekávaný výstup, síť se musí sama rozhodnout, jaká odezva je nejlepší a podle toho upravit své parametry. Síť se rozhoduje podle rozložení vzorů v příznakovém prostoru – identifikuje shluky vzorů. Problematickým parametrem je počet shluků – ten můžeme buď specifikovat, a nebo nechat síť vybrat i ten např. pomocí krosvalidace.

### 5.4.1 Kompetiční učení

V kompetičním učení soutěží jednotlivé elementy sítě o právo reprezentovat daný vzor. Vždy vítězí právě jeden, soupeři jsou potlačeni.

Vstupní vzor je zpracován takovým počtem neuronů, jaký je očekávaný počet shluků (*clusterů*) v datech. Každý neuron pak reprezentuje jeden shluk. Každý neuron počítá (euklidovskou) vzdálenost mezi svým váhovým neuronem a vstupním vzorem, vítězí ten nejbližší. Lze si to představit jako zobecnění perceptronu pro více váhových vektorů.

Mezi neurony existují „laterální spoje“, pomocí kterých vítězný neuron inhibuje ostatní.

Vítězný neuron adaptuje své váhy podle

$$\mathbf{w} = \mathbf{w} + \Delta \mathbf{w}$$

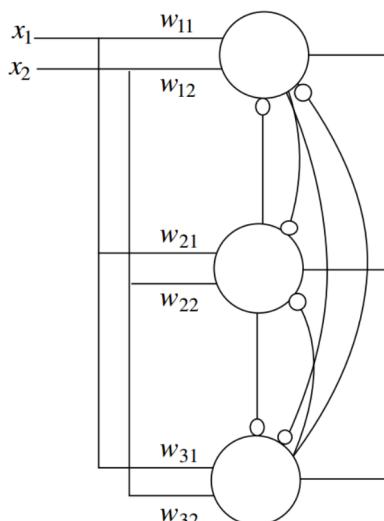
kde  $\Delta \mathbf{w}$  může být

$$\Delta \mathbf{w} = \alpha \mathbf{x}$$

a nebo

$$\Delta \mathbf{w} = \alpha (\mathbf{x} - \mathbf{w})$$

kde  $\alpha$  je **plasticita sítě**, která během učení klesá. Druhý způsob se nazývá *diferenční aktualizace*. Nový váhový vektor se typicky ještě znormuje. Variantou je i *dávková aktualizace*, kdy se korekce vah „kumuluje“ a aplikuje se naráz po několika iteracích.



Proces lze urychlit vhodnou inicializací vah (např. podle náhodně vybraných vzorů). Problémem jsou *mrtvé neurony* (tj. neurony, které nikdy nezvítězí a tedy nereprezentují žádný shluk) – ty řeší např. mrázka v Kohonenově vrstvě (viz dále).

Cílem učení je umístit neurony (resp. jejich váhy) do středu (=težiště) shluku. Formulujeme-li pro 1 neuron s váhovým vektorem  $\mathbf{w}$  a množinu normovaných  $n$ -rozměrných vstupních vektorů  $X = \{\mathbf{x}_1, \dots, \mathbf{x}_m\}$  energetickou funkci jako

$$E_X(\mathbf{w}) = \sum_{i=1}^m \|\mathbf{x}_i - \mathbf{w}\|^2$$

pak lze ukázat, že optimálním případě (tj. v minimu energetické funkce) je vektor vah umístěn právě v težišti:

$$\begin{aligned} E_X(\mathbf{w}) &= m\mathbf{w}^2 - 2 \sum_{i=1}^m \mathbf{x}_i \mathbf{w} + \sum_{i=1}^m \mathbf{x}_i^2 \\ &= m \left( \mathbf{w}^2 - \frac{2}{m} \mathbf{w} \sum_{i=1}^m \mathbf{x}_i \right) + \sum_{i=1}^m \mathbf{x}_i^2 \\ &= m \left( \mathbf{w} - \frac{1}{m} \sum_{i=1}^m \mathbf{x}_i \right)^2 - \frac{1}{m^2} \left( \sum_{i=1}^m \mathbf{x}_i \right)^2 + \sum_{i=1}^m \mathbf{x}_i^2 \\ &= m(\mathbf{w} - \mathbf{x}^*)^2 + K \end{aligned}$$

kde  $\mathbf{x}^*$  je centroid  $X$  a  $K$  je konstanta.

#### 5.4.1.1 Klasické shlukovací metody

**$k$ -NN ( $k$  nejbližších sousedů) – supervised** Trénovací vzory jsou klasifikovány do jedné z tříd a všechny uloženy. Nový vzor se pak zařadí do té třídy, kam patří většina z jeho  $k$  nejbližších sousedů.

**$k$ -means ( $k$  středů) – unsupervised** Na začátku máme  $k$  vektorů, každý ve svém vlastním shluku. Nový vektor je klasifikován k tomu shluku, jehož centroid je mu nejbližší. Cetnroid se pak aktualizuje

$$\mathbf{c}_k = \mathbf{c}_k + \frac{1}{n_k} (\mathbf{x} - \mathbf{c}_k) \quad \text{kde } n_k \text{ je velikost shluku } k$$

Jedná se vlastně o vektorovou kvantizaci.

My se chceme vyhnout nutnosti skladovat všechny vzory jako  $k$ -NN, chtěli bychom pouze uložit strukturu pomocí váhových vektorů.

#### 5.4.1.2 Konvergence kompetičního učení

Při rozboru energetické funkce v předchozí sekci jsme předpokládali, že víme, která množina vzorů náleží danému neuronu. To je ale přesně to, co bychom rádi nechali na starost neuronové sítě. Potřebujeme tedy nějakou metriku, jak měřit „kvalitu“ shlukování pro různé počty a umístění váhových vektorů.

##### 5.4.1.2.1 Stabilita řešení

Intuitivně: stabilní rovnovážný stav vyžaduje jasně ohraničené shluky.

Nechť  $P$  je  $\{\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_m\}$  množina  $n$ -rozměrných ( $n \geq 2$ ) vektorů ležících ve stejném poloprostoru ( $\sim$  formální omezení velikosti shluku).

**Svazek**  $K$  definovaný pomocí  $P$  je množina všech vektorů  $\mathbf{x}$  tvaru

$$\mathbf{x} = \alpha_1 \mathbf{p}_1 + \alpha_2 \mathbf{p}_2 + \dots + \alpha_m \mathbf{p}_m$$

kde  $\alpha_1, \alpha_2, \dots, \alpha_m$  jsou kladná reálná čísla.

Svazek shluku obsahuje všechny vektory „uvnitř“ shluku.

(Úhlový) průměr  $\varphi$  svazku  $K$  definovaného normovanými vektory  $\{\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_m\}$  odpovídá:

$$\varphi = \sup\{\arccos(\mathbf{a} \cdot \mathbf{b}) \mid \forall \mathbf{a}, \mathbf{b} \in K; \|\mathbf{a}\| = \|\mathbf{b}\| = 1\}$$

kde  $0 \leq \arccos(\mathbf{a} \cdot \mathbf{b}) \leq \pi$ .

Průměr svazku definovaného normovanými vektory odpovídá největšímu možnému úhlu mezi vektory ve shluku. Postačující podmínkou pro stabilní řešení je, aby byl úhlový průměr svazků menší než jejich vzájemná vzdálenost.

Nechť  $P = \{\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_m\}$  a  $N = \{\mathbf{n}_1, \mathbf{n}_2, \dots, \mathbf{n}_k\}$  jsou dvě neprázdné množiny normovaných vektorů v  $n$ -rozměrném prostoru ( $n \geq 2$ ), které definují svazky  $K_P$  a  $K_N$ .

- Jestliže je průnik těchto dvou svazků prázdný, je (úhlová) vzdálenost mezi  $K_N$  a  $K_P$  dáná pomocí:

$$\psi_{P,N} = \inf\{\arccos(\mathbf{p} \cdot \mathbf{n}) \mid \mathbf{p} \in K_P, \mathbf{n} \in K_N, \|\mathbf{p}\| = \|\mathbf{n}\| = 1\}$$

kde  $0 \leq \arccos(\mathbf{a} \cdot \mathbf{b}) \leq \pi$ .

- Jestliže se  $K_N$  a  $K_P$  prolínají, je  $\psi_{P,N} = 0$

Jestliže je úhlová vzdálenost mezi shluky větší než úhlový průměr svazků, potom existuje stabilní řešení. Váhové vektory budou ležet ve svazku jím příslušejícího shluku. Jestliže se váhový vektor už „dostal“ dovnitř příslušného svazku, zůstane tam i nadále

Jak ohodnotit různá shlukování: obecně je výhodnější je menší počet kompaktnějších shluků. Definuje se *ohodnocovací funkce* pro penalizaci příliš vysokého počtu shluků

### 5.4.2 PCA (Principal Component Analysis)

PCA je dalším typem učení bez učitele, který se používá pro redukci dimenziality vstupních dat. Pro výpočet lze použít lineární asociátory, tj. typ neuornových sítí.

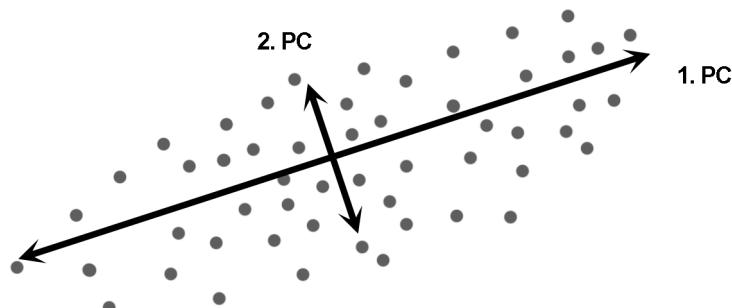
Máme dánu množinu  $m$   $n$ -rozměrných vektorů  $\{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_m\}$ . Prvním nejdůležitějším příznakem (*first principal component*) je vektor  $\mathbf{w}$ , který maximalizuje výraz

$$\frac{1}{m} \sum_{i=1}^m \|\mathbf{w} \cdot \mathbf{x}_i\|^2$$

tedy průměr kvadratických skalárních produktů. Ortogonální projekce vektorů na 1. PC zachovává nejvíce informace.

Druhý nejdůležitější příznak se spočte tak, že od každého  $\mathbf{x}_i$  odečteme jeho ortogonální projekci na 1. PC a opět najdeme  $\mathbf{w}$  maximalizující výraz výše – tentokrát však již pracujeme s „rezidui“ původních vektorů. Výsledkem je vektor kolmý na 1. PC a s největším rozptylem.

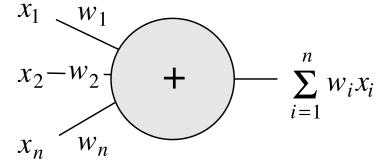
Další PC rekursivně.



#### 5.4.2.1 Ojův algoritmus učení

Algoritmus pro výpočet 1. PC pomocí *lineárních asociátorů*. Lineární asociátor je neuron, jehož výstup je jednoduše vážená suma jeho vstupů – tj. jeho potenciál. Jeho přenosová funkce je tedy identita:  $f(\xi) = \xi$

Předpokládá, že data jsou vycentrována v počátku.




---

#### Algorithm 5 Ojův algoritmus učení

---

```

1:  $X \leftarrow$  množina vstupních vektorů vycentrovaných v počátku
2:  $\mathbf{w} \leftarrow$  náhodně inicializovaný váhový vektor ( $\mathbf{w} \neq 0$ )
3:  $\gamma \leftarrow$  náhodně inicializovaná učící konstanta ( $0 < \gamma \leq 1$ )
4: while not ukončovací podmínka (počet iterací) do
5:    $\mathbf{x} \leftarrow$  náhodně vybraný vektor z  $X$ 
6:    $\phi = \mathbf{x} \cdot \mathbf{w}$ 
7:    $\mathbf{w} = \mathbf{w} + \gamma \cdot \phi \cdot (\mathbf{x} - \phi \mathbf{w})$ 
8:   zmenší  $\gamma$ 

```

---

Parametr učení  $\gamma$  je třeba zvolit dostatečně malý, aby byla zaručena adekvátní adaptace vah (omezení velkých oscilací).

Během učení dochází k „automatická normalizace“ váhového vektoru. Odpadá tedy nutnost explicitní normalizace, kde bychom potřebovali znát globální informaci o všech vzorech. Postačí lokální informace o aktualizované váze, vstupu a skalárním součinu spočteném v asociátoru.

První nejdůležitější příznak odpovídá směru nejdelšího vlastního vektoru korelační matici uvažovaných vstupních vektorů.

#### 5.4.2.2 Konvergence Ojova algoritmu učení

**Věta 5.4.1.** Pokud existuje jednoznačné řešení, Ojův algoritmus konverguje.

*Důkaz.* Začne-li Ojův algoritmus s  $\mathbf{w}$  uvnitř svazku  $K_X$ , bude v něm oscilovat, ale neopustí ho. Pokud je  $\|\mathbf{w}\| = 1$ , pak  $\phi = \mathbf{x} \cdot \mathbf{w}$  odpovídá délce projekce  $\mathbf{x}$  na  $\mathbf{w}$ . Vektor  $\mathbf{x} - \phi \mathbf{w}$  je kolmý na  $\mathbf{w}$ . Takže iterace algoritmu přitahuje  $\mathbf{w}$  k vektorům ze shluku  $X$ . Pokud zůstane  $\|\mathbf{w}\| = 1$ , pak efektem učení je umístění  $\mathbf{w}$  doprostřed shluku.

Stačí tedy ukázat, že  $\mathbf{w}$  je během algoritmu automaticky normován.

1. **Nechť**  $\|\mathbf{w}\| > 1$ : V tomto případě má  $(\mathbf{x} \cdot \mathbf{w})\mathbf{w}$  větší délku, než ortogonální projekce  $\mathbf{x}$  na  $\mathbf{w}$ . Předpokládejme, že  $\mathbf{x} \cdot \mathbf{w} > 0$ , tj. vektory nejsou od sebe příliš daleko. Platí, že vektor  $\mathbf{x} - (\mathbf{x} \cdot \mathbf{w})\mathbf{w}$  má zápornou projekci na  $\mathbf{w}$ , neboť

$$(\mathbf{x} - (\mathbf{x} \cdot \mathbf{w})\mathbf{w})\mathbf{w} = \mathbf{x} \cdot \mathbf{w} - \|\mathbf{w}\|^2 \mathbf{x} \cdot \mathbf{w} < 0$$

Výsledek po větším počtu iterací: vektor  $\mathbf{x} - (\mathbf{x} \cdot \mathbf{w})\mathbf{w}$  má jednu složku kolmou na  $\mathbf{w}$  a druhou složku směřující opačným směrem, než  $\mathbf{w}$ . Opakování iterace, přenesou  $\mathbf{w}$  do centra shluku, takže kolmé složky se časem vyruší. Nicméně složky směřující opačným směrem než  $\mathbf{w}$  se nevyruší a budou postupně  $\mathbf{w}$  zmenšovat.

Je třeba dát pozor, abychom  $\mathbf{w}$  nezměnili příliš nebo dokonce v jednom kroku zcela neobrátili  $\mathbf{w}$  opačným směrem. Tomu se lze vyhnout normalizací vektorů z  $X$  před začátkem učení a vhodnou volbou  $\gamma$ . Pro tu obecně platí, že bychom chtěli, aby pokud  $\phi = \mathbf{x} \cdot \mathbf{w} > 1$ , tak aby i nový váhový vektor měl kladný skalární součin s  $\mathbf{x}$ , tj.

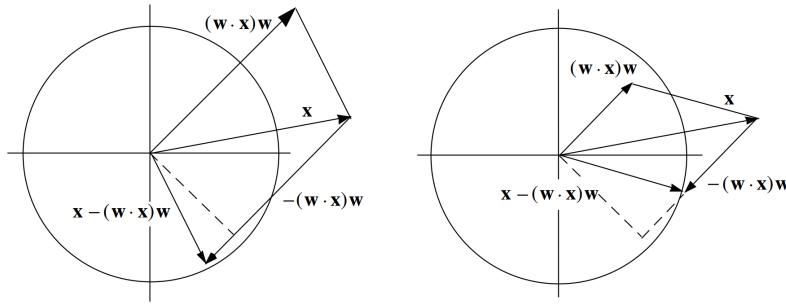
$$\mathbf{x} \cdot (\mathbf{w} + \gamma \phi (\mathbf{x} - \phi \mathbf{w})) > 0$$

což lze upravit na

$$\gamma(\|\mathbf{x}\|^2 - \phi^2) > -1$$

což pro dostatečně malé  $\gamma$  platí vždy.

2. **Nechť**  $\|\mathbf{w}\| < 1$ : (analogicky). V tomto případě má vektor  $\mathbf{x} - (\mathbf{x} \cdot \mathbf{w})\mathbf{w}$  *kladnou* projekci na  $\mathbf{w}$ , což povede po větším počtu iterací ke zvětšování délky  $\mathbf{w}$ .

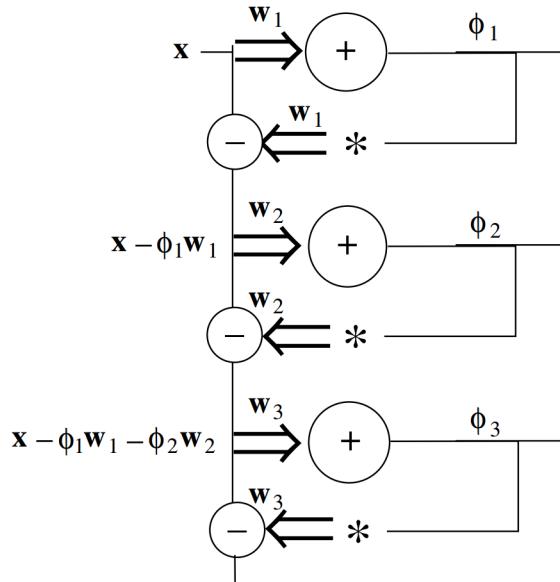


Vektor  $\mathbf{w}$  bude tedy vycentrován do středu shluku a jeho délka bude oscilovat kolem 1 (pro dostatečně malé  $\gamma$ ).  $\square$

Problémy: řídké shluky, příliš velké rozdíly v délce vstupních vektorů.

#### 5.4.2.3 Multiple PC

Pro nalezení prvních  $m$  nejdůležitějších příznaků lze použít následující síť:



#### 5.4.3 Kohonenovy mapy

Model fungující na principu samoorganizace. Důležitou vlastností je, že zachovává topologii mapovaného prostoru.

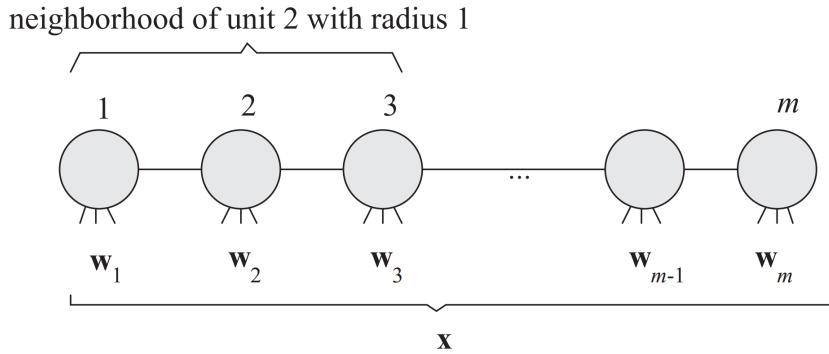
Neurony jsou uspořádány do mřížky (tzv. **Kohonenova mřížka**), která definuje okolí jednotky (nejbližší sousedy) a laterální spoje k neuronům v tomto okolí. V průběhu učení se aktualizují váhy jak daného neuronu, tak jeho sousedů (motivace: blízké neurony odpovídají topologicky blízkým datům, měly by tedy reagovat na podobné signály).

##### 5.4.3.1 Algoritmus učení (1D)

Mapujeme  $n$ -rozměrný prostor pomocí 1D řetězce Kohonenovských neuronů (očíslovaných od 1 do  $m$ ). Každému neuronu náleží  $n$ -dimenzionální váhový vektor  $\mathbf{w}_i$ . Cílem učení je specializovat každý neuron na jinou oblast vstupního prostoru (tuto specializaci charakterizuje maximální excitace příslušného neuronu pro vzory z dané oblasti).

Každý neuron spočte (euklidovskou) vzdálenost mezi vstupem  $\mathbf{x}$  a svým váhovým vektorem  $\mathbf{w}$  (tedy nikoliv skalární součin, jak jsme zvyklí).

**Okolí neuronu  $k$**  o poloměru  $r$  tvoří všechny jednotky umístěné do  $r$  pozic od neuronu  $k$ . Okolí neuronu  $k$  o poloměru 1 pro 1D mřížku jsou jednoduše neuronu  $k-1$  a  $k+1$  (viz. obrázek). Neurony



na okrajích mřížky mají okolí asymetrické. Pro vícerozměrné Kohonenovy mapy a zvolenou metriku je okolí definováno obdobně.

**Funkce laterální vazby**  $\phi(i, k)$  (neighbourhood function) vyjadřuje sílu laterální vazby mezi neurony  $k$  a  $i$ . Příklady:

- $\phi(i, k) = 1$  pro všechny neurony  $i$  z okolí  $k$  o poloměru  $r$ , a  $\phi(i, k) = 0$  pro všechny ostatní.
- funkce „mexického klobouku“
- gaussovské okolí

Algoritmus učení:

---

#### Algorithm 6 Kohonenovské učení

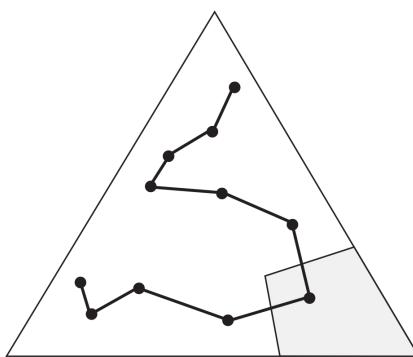
---

```

1: function KOHONENLEARNING( $\phi, r, \eta$ )
2:    $w_1, \dots, w_m \leftarrow$  malé náhodné hodnoty
3:   while not max_iterations do
4:      $x \leftarrow$  předlož nový trénovací vzor
5:      $k \leftarrow$  neuron s minimální hodnotou euklidovské vzdálenosti mezi  $w_k$  a  $x$ 
6:     for  $i$  in  $1 \dots m$  do
7:        $w_i = w_i + \eta \phi(i, k)(x - w_i)$ 
8:     zmenší  $\eta$ , uprav  $\phi$ 
```

---

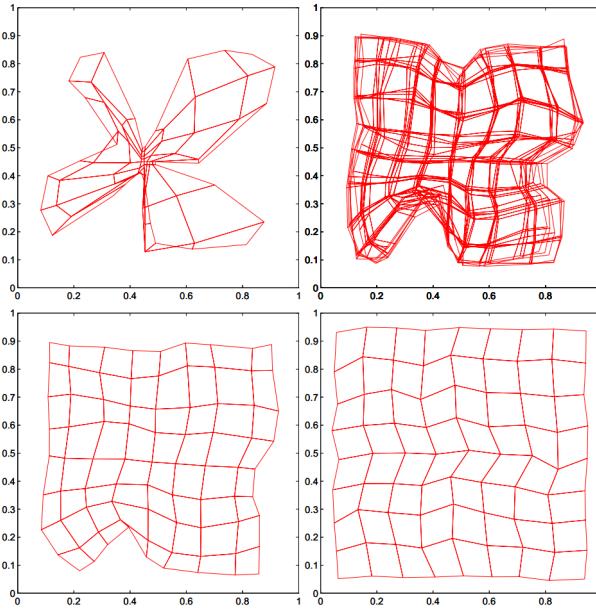
Úprava vah ovlivněných neuronů je přitahuje ve směru  $x$ . Funkce  $\phi$  se v průběhu učení upravuje tak, že se zmenšuje poloměr okolí  $r$ . Učící konstanta  $\eta$  ovlivňuje velikost aktualizace a také se v průběhu algoritmu zmenšuje. Mělo by platit  $0 < \eta < 1$  a označíme-li hodnotu  $\eta$  v kroku  $t$  jako  $\eta(t)$ , pak rovněž  $\sum_{t=1}^{\infty} \eta(t) = \infty \wedge \sum_{t=1}^{\infty} \eta(t)^2 < \infty$ .



Obrázek 5.2: Mapování trojúhelníku pomocí 1D Kohonenova řetězce. Každý neuron (tečka) reprezentuje nějaou část oblasti.

#### 5.4.3.2 Konvergence

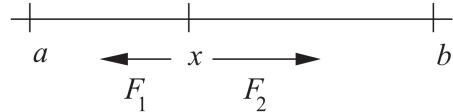
Analýza konvergence je poměrně složitá. Spokojíme se s analýzou podmínek stability řešení za předpokladu, že síť už dospěla do jistého uspořádaného stavu.



Obrázek 5.3: Mapování oblasti pomocí 2D Kohonenovy mapy, proces učení. V pravo nahoře zobrazeno několik iterací přes sebe.

#### 5.4.3.2.1 Jednorozměrný případ

- Uvažme nejjednodušší Kohonenovu mapu: vstupní doménou je interval  $[a, b]$  a máme jediný neuron. Učící algoritmus povede ke konvergenci váhy  $w$  tohoto neuronu doprostřed intervalu.



Adaptační pravidlo zní

$$w_t = w_{t-1} + \eta \phi(i, k)(x - w_{t-1})$$

kde  $w_{t-1}$  resp.  $w_t$  je váhový vektor v kroku  $t-1$  resp.  $t$  a  $x$  je vstup, tj. číslo z intervalu  $[a, b]$ . Pro  $0 < \eta < 1$  nemůže posloupnost  $w_1, w_2, \dots$  opustit interval  $[a, b]$ . Očekávaná hodnota váhy  $w$  je omezená, tudíž i očekávaná hodnota derivace  $w$  vzhledem k  $t$  je nulová, tj.

$$\left\langle \frac{dw}{dt} \right\rangle = 0$$

jinak by očekávaná hodnota  $w$  mohla časem nabýt hodnoty menší než  $a$  nebo větší než  $b$ . Protože

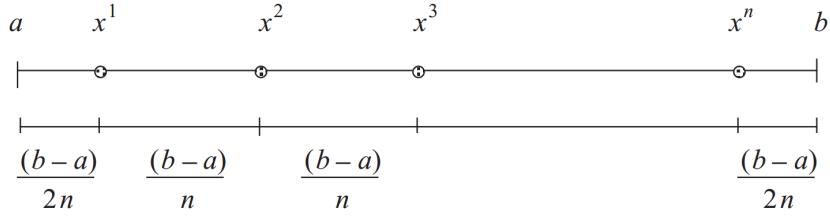
$$\left\langle \frac{dw}{dt} \right\rangle = \eta(\langle x \rangle - \langle w \rangle) = \eta \left( \frac{a+b}{2} - \langle w \rangle \right)$$

z čehož plyne

$$\langle w \rangle = \frac{a+b}{2}$$

- Vstupní doménou je interval  $[a, b]$  a máme  $n$  neuronů s vahami  $w^1, w^2, \dots, w^n$ . Váhy jsou uspořádány monotónně, tj.  $a < w^1 < w^2 < \dots < w^n < b$ . Neuvažujeme okolí. Váhy budou konvergovat k

$$\langle w^i \rangle = a + (2i-1) \frac{b-a}{2n}$$



#### 5.4.3.2.2 Dvourozměrný případ

Vstupní doménou je interval  $[a, b] \times [c, d]$  a máme  $n \times n$  neuronů  $N^{11}, \dots, N^{nn}$  s vahami uspořádanými monotónně, tj.

$$w_1^{ij} < w_1^{ik} \quad \text{pokud } j < k$$

a

$$w_2^{ij} < w_2^{kj} \quad \text{pokud } i < k$$

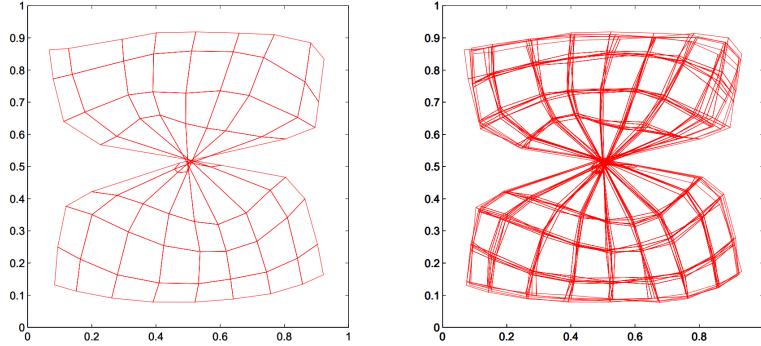
Problém lze zredukovat na dva jednorozměrné. Nechť  $w_1^j = \frac{1}{n} \sum_{i=1}^n w_1^{ij}$  označuje průměr vah neuronu v  $j$ -tém sloupci. Protože  $w_1^{ij} < w_1^{ik}$  pro  $j < k$ , budou  $w_1^j$  monotónně uspořádané, tj.

$$a < w_1^1 < w_1^2 < \dots < w_1^n < b$$

Průměr vah v prvním sloupci bude oscilovat kolem očekávané hodnoty  $\langle w_1^1 \rangle$ . Podobně pro neurony v každém řádku.

#### 5.4.3.3 Problémy

Rozvinutí planární mřížky může být problematické.



Obrázek 5.4: Kohonenova mapa s „uzlem“. Ani další iterace nepomohou uzel rozmotat. „Metastabilní stavy“ a nevhodná volba funkce laterální interakce (příliš rychlý pokles)

#### 5.4.4 Učení s učitelem

Kohonenovy mapy lze učit i pomocí supervised learning. (Pozn.: Ačkoliv je tohle kapitola o učení bez učitele, je vhodné to u Kohonenových map zmínit.)

Používá se LVQ (Learning Vector Quantization).

**LVQ1** Motivace: vektor  $\mathbf{x}$  by měl patřit ke stejné třídě, jako nejbližší  $\mathbf{w}_i$ .

Nechť  $c = \arg \min_i \{\|\mathbf{x} - \mathbf{w}_i\|\}$  je index  $\mathbf{w}_i$  ležícího nejblíže k  $\mathbf{x}$  (tj.  $c$  je „vítězný neuron“).

Váhy se pak adaptují podle:

$$\mathbf{w}_c(t+1) = \begin{cases} \mathbf{w}_c(t) + \alpha(t)(\mathbf{x}(t) - \mathbf{w}_c(t)) & \text{jestliže } \mathbf{x} \text{ a } \mathbf{w}_c \text{ jsou klasifikovány stejně} \\ \mathbf{w}_c(t) - \alpha(t)(\mathbf{x}(t) - \mathbf{w}_c(t)) & \text{jestliže } \mathbf{x} \text{ a } \mathbf{w}_c \text{ jsou klasifikovány jinak} \end{cases}$$

$$\mathbf{w}_i(t+1) = \mathbf{w}_i(t) \quad \text{pro } i \neq c$$

kde  $0 < \alpha(t) < 1$  je učící parametr.

**LVQ2.1** Motivace: Adaptace dvou nejbližších sousedů  $\mathbf{x}$  současně. Jeden z nich musí patřit ke správné třídě, druhý k nesprávné. Navíc musí být  $\mathbf{x}$  z okolí dělicí nadplochy mezi  $\mathbf{w}_i$  a  $\mathbf{w}_j$  ( $\sim$  z „okénka“). Je-li  $d_i$  (resp.  $d_j$ ) Euklidovská vzdálenost mezi  $\mathbf{x}$  a  $\mathbf{w}_i$  (resp. mezi  $\mathbf{x}$  a  $\mathbf{w}_j$ ), lze „okénko“ definovat pomocí vztahu:

$$\min\left(\frac{d_i}{d_j}, \frac{d_j}{d_i}\right) > s, \quad \text{kde} \quad s = \frac{1-w}{1+w}$$

(doporučované hodnoty  $w$  (= šířky „okénka“): 0.2 – 0.3)

Adaptace podle

$$\mathbf{w}_i(t+1) = \mathbf{w}_i(t) - \alpha(t)(\mathbf{x}(t) - \mathbf{w}_i(t)) \mathbf{w}_j(t+1) = \mathbf{w}_j(t) + \alpha(t)(\mathbf{x}(t) - \mathbf{w}_j(t))$$

kde  $\mathbf{w}_i$  a  $\mathbf{w}_j$  leží nejblíže  $\mathbf{x}$ , přitom  $\mathbf{x}$  a  $\mathbf{w}_j$  patří ke stejné třídě a  $\mathbf{x}$  a  $\mathbf{w}_i$  patří k různým třídám a  $\mathbf{x}$  je z „okénka“.

**LVQ3** Stejná pravidla jako LVQ2.1, navíc:

$$\mathbf{w}_k(t+1) = \mathbf{w}_k(t) + \varepsilon\alpha(t)(\mathbf{x}(t) - \mathbf{w}_k(t))$$

pro  $k \in \{i, j\}$  jestliže  $\mathbf{x}$ ,  $\mathbf{w}_i$  i  $\mathbf{w}_j$  patří ke stejné třídě.

## 5.5 Modulární, hierarchické a hybridní modely neuronových sítí.

Kombinace předchozích modelů. Inspirace v biologii.

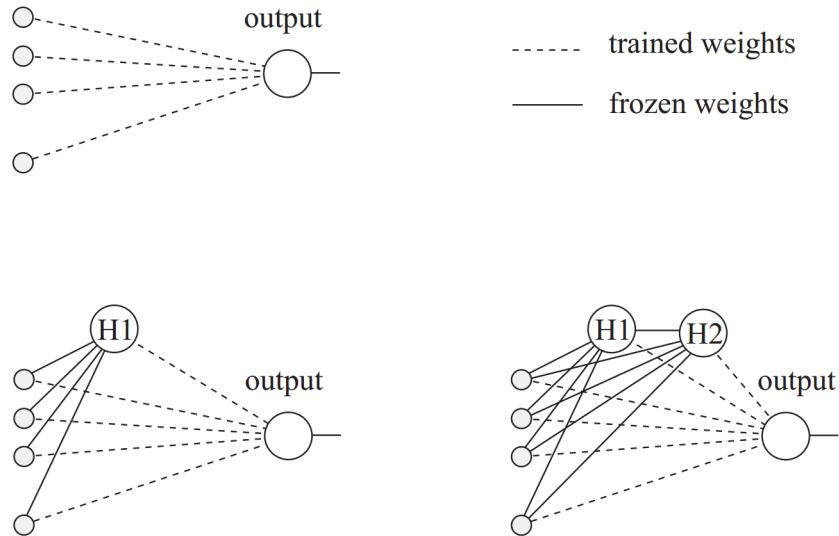
### 5.5.1 Modulární síť

#### 5.5.1.1 Kaskádová korelace

Rostoucí architektura. Začíná bez skrytých neuronů, s přímým napojením vstupů na výstupy. Síť se adaptuje pomocí algoritmu Quickprop. Pokud je po konvergenci učení chyba stále větší než požadovaný práh, **přidá se do sítě nový skrytý neuron**. Tento neuron je natrénován (stranou od sítě samotné) tak, aby maximalizoval korelaci mezi svým výstupem a chybou na výstupu sítě. Pro jediný výstupní neuron tedy formálně maximalizujeme

$$S = \left| \sum_{i=1}^p (V_i - \bar{V})(E_i - \bar{E}) \right|$$

kde  $\bar{E}$  je průměrná chyba přes všechny vzory,  $E_i$  je chyba na vzoru  $i$ ,  $V_i$  je výstup pro  $i$ -tý vzor a  $\bar{V}$  je průměrný výstup. Motivací je, aby se přidávaný neuron naučil nějaký příznak, který vysoce koreluje se zbytkovou chybou sítě.



Obrázek 5.5: Proces přidávání skrytých neuronů (pozn. ve slajdech Mrázové jsou špatně označeny zmrazené hrany)

Po natrénovaní je neuron přidán do sítě a jeho vstupní váhy jsou zmrazeny. Síť se znova učí dokud se nestabilizuje chyba a případně se přidá další neuron. Ten bude mít na vstupu zapojeny výstupy všech vstupních neuronů a všech předchozích skrytých neuronů.

V každém kroku algoritmu se de facto trénuje jediná vrstva vah, tj. jediná sigmoidální jednotka.

### 5.5.1.2 Optimální moduly a mixture of experts

### 5.5.2 Hybridní sítě

Typicky kombinace samoorganizujících a dopředných sítí.

#### 5.5.2.1 ART (Adaptive Resonance Theory)

---

##### Algorithm 7 Kohonenovské učení

---

```

1: function ART1
2:    $\rho \leftarrow$  hodnota z intervalu  $[0, 1]$                                  $\triangleright$  práh bdělosti
3:    $t = 0$ 
4:   for  $i$  in  $0$  to  $N - 1$  do
5:     for  $j$  in  $0$  to  $M - 1$  do
6:        $t_{ij}(0) \leftarrow 1$        $\triangleright$  váha mezi vstupním neuronem  $i$  a výstupním neuronem  $j$  v čase 0
7:        $b_{ij}(0) \leftarrow \frac{1}{1+N}$   $\triangleright$  váha mezi výstupním neuronem  $j$  a vstupním neuronem  $i$  v čase 0
8:     while existuje vstupní vzor do
9:        $\mathbf{x} \leftarrow$  vstupní vzor
10:      for  $j$  in  $0$  to  $M - 1$  do
11:         $\mu_j = \sum_{i=0}^{N-1} b_{ij}(t)x_i$                                  $\triangleright$  výstup výstupního neuronu  $j$ 
12:         $k = \arg \max_j \{\mu_j\}$                                  $\triangleright$  neuron nejlépe odpovídající vstupnímu vzoru
13:         $\|\mathbf{x}\| = \sum_{i=0}^{N-1} x_i$ 
14:         $\|\mathbf{T} \cdot \mathbf{x}\| = \sum_{i=0}^{N-1} t_{ik} \cdot x_i$ 
15:        if  $\|\mathbf{T} \cdot \mathbf{x}\| / \|\mathbf{x}\| > \rho$  then
16:           $t_{ik}(t+1) = t_{ik}(t) \cdot x_i$ 
17:
```

---

#### 5.5.2.2 Sítě se vstřícným šířením (counterpropagation networks)

Síť se skládá ze vstupní, skryté (Kohonenovské) a výstupní (Grossbergovské) vrstvy.

Původně navrženy pro approximaci spojitého zobrazení  $f$  a jeho inverze  $f^{-1}$  pomocí dvou symetrických sekcí.

Skrytá vrstva je typicky Kohonenovská mřížka, výstupem je jediný lineární asociátor (v případě mapování  $f : \mathbb{R}^n \rightarrow \mathbb{R}$ ). Váhy mezi skrytou vrstvou a výstupním neuronem se upravují učením s učitelem.

Trénování probíhá ve 2 fázích:

1. skrytá vrstva je trénována na náhodně vybraných vektorech ze vstupního prostoru. Skrytá vrstva pak reprezentuje nějaké klastrování vstupního prostoru (lze si představit jako Voroného diagram). Lze použít libovolnou trénovací strategii (LVQ). Skrytá vrstva může být Kohonenovská mřížka nebo izolované elementy. Výstup skryté vrstvy je kontrolován tak, že je vždy aktivní pouze jeden neuron.
2. Váhy mezi skrytými neuronami a výstupním neuronem lze chápat jako „výšku“ jednotlivých Voroného segmentů. Učením se tyto výšky nastaví tak, aby co nejlépe approximovaly požadovanou funkci.

Je-li na vstupu vektor  $\mathbf{x}$  a je aktivován skrytý neuron  $i$ , pak chyba approximace odpovídá

$$E = \frac{1}{2}(z_i - f(\mathbf{x}))^2$$

kde  $z_i$  je váha mezi skrytým neuronem  $i$  a výstupním neuronem. Vzorec pro aktualizaci váhy pak dostaneme aplikací gradientní metody, tj.

$$\Delta z_i = \frac{\partial E}{\partial z_i} = \gamma(f(\mathbf{x}) - z_i)$$

kde  $\gamma$  je učící konstanta.

Obě fáze lze provádět buď odděleně za sebou nebo je prolínat. Síť lze přímočaře rozšířit i pro více výstupních jednotek (tj. funkce  $\mathbb{R}^n \rightarrow \mathbb{R}^m$ ).

---

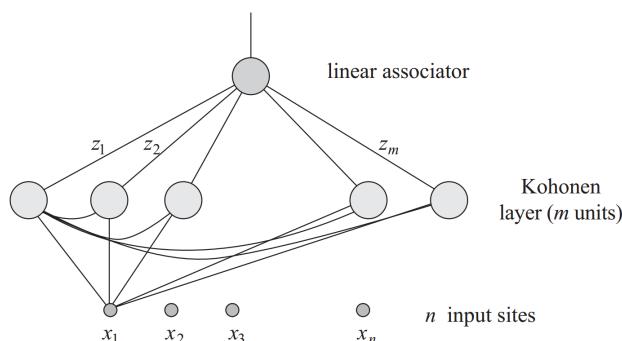
#### Algorithm 8 Kohonenovské učení

---

- 1: **function** COUNTERPROPAGATIONLEARNING
  - 2:      inicializuj náhodné hodnoty synaptických vah
  - 3:
- 

#### 5.5.2.3 Spline networks

#### 5.5.2.4 RBF (Radial Basis Functions)



Další varianta použití Kohonenovské vrstvy, tentokrát však s Gaussovskými aktivačními funkcemi. Výstup každého skrytého neuronu je předán výstupnímu lineárnímu asociátoru, přičemž skrytý neuron jehož váhy leží blíže vstupnímu vzoru by měl být silněji aktivován.

Každý skrytý neuron  $j$  počítá svůj výstup jako

$$g_j(\mathbf{x}) = \frac{\exp\left(\frac{-(\mathbf{x}-\mathbf{w}_j)^2}{2\sigma_j^2}\right)}{\sum_k \exp\left(\frac{-(\mathbf{x}-\mathbf{w}_k)^2}{2\sigma_k^2}\right)}$$

kde  $\mathbf{x}$  je vstup,  $\mathbf{w}_i$  jsou váhové vektory příslušných neuronů a  $\sigma_i$  jsou konstanty (nastavené např. podle vzdálenosti mezi příslušným váhovým vektorem a jeho nejbližším sousedem). Každý výstup je normován sumou výstupů všech skrytých neuronů. To vynucuje propojení všech neuronů ve skryté vrstvě.

Váhy  $z_1, \dots, z_n$  mezi skrytou vrstvou a výstupním asociátorem jsou nastaveny pomocí backpropagation. Chyba na vstupním vzoru  $\mathbf{x}$  je dána jako

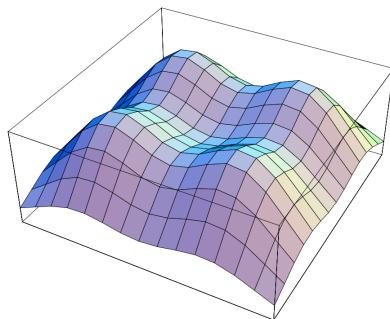
$$E = \frac{1}{2} \left( \sum_{i=1}^n g_i(\mathbf{x}) z_i - f(\mathbf{x}) \right)^2$$

kde  $f(\mathbf{x})$  je očekávaný výstup (tj. výstup modelované funkce  $f$ ).

Aktualizace vah je tedy dána jako

$$\Delta z_i = -\frac{dE}{dz_i} = \gamma g_i(\mathbf{x}) \left( f(\mathbf{x}) - \sum_{i=1}^n g_i(\mathbf{x}) z_i \right)$$

Díky použití Gaussovských funkcí je výsledná approximace spojitá, viz. obrázek ??.



Obrázek 5.6: Aproximace funkce pomocí RBF.

### 5.5.3 Vícevrstvé Kohonenovy mapy

## 5.6 Genetické algoritmy a jejich využití při učení umělých neuronových sítí.

Evoluční a genetické algoritmy viz. 3.1.1, neuroevoluce viz. 3.6.2.