# Vypracované otázky k MSZ pro rok 2022

Specializace NNET

11. června 2022

Vladimír Dušek, xdusek27

Aktuální PDF a zdrojové kódy: `https://github.com/vdusek/msz`

## Specializace Počítačové sítě – NNET

1. Architektura superskalárních procesorů a algoritmy zpracování instrukcí mimo pořadí, predikce skoků.
2. Paměťová konzistence a předbíhání operací čtení a zápisu, podpora virtuálního adresového prostoru.
3. Datový paralelismus SIMD, HW implementace a SW podpora.
4. Architektury se sdílenou pamětí UMA a NUMA, zajištění lokality dat.
5. Problém koherence pamětí cache na systémech se sdílenou pamětí, protokol MSI.
6. Paralelní zpracování v OpenMP: Smyčky, sekce a tasky a synchronizační prostředky.
7. Pravděpodobnost, podmíněná pravděpodobnost, nezávislost.
8. Náhodná proměnná, typy náhodné proměnná, funkční a číselné charakteristiky, významná rozdělení pravděpodobnosti.
9. Bodové a intervalové odhady parametrů, testování hypotéz o parametrech.
10. Vícevýběrové testy, testy o rozdělení, testy dobré shody.
11. Regresní analýza.
12. Markovské řetězce a základní techniky pro jejich analýzu.
13. Randomizované algoritmy (Monte Carlo a Las Vegas algoritmy).
14. Problém generalizace strojového učení a přístup k jeho řešení (trénovací, validační a testovací sada, regularizace, předtrénování, multi-task learning, augmentace dat, dropout, ...)
15. Generativní modely a diskriminativní přístup ke klasifikaci (gaussovský klasifikátor, logistická regrese, ...)
16. Neuronové sítě a jejich trénování (metoda gradientního sestupu, účelová (loss) funkce, výpočetní graf, aktivační funkce, zápis pomocí maticového násobení, ...)
17. Neuronové sítě pro strukturovaná data (konvoluční a rekurentní sítě, motivace, základní vlastnosti, použití)
18. Prohledávání stavového prostoru (informované a neinformované metody, lokální prohledávání, prohledávání v nejistém prostředí, hraní her, CSP úlohy)
19. Klasifikace formálních jazyků (Chomského hierarchie), vlastnosti formálních jazyků a jejich rozhodnutelnost.
20. Konečné automaty (jazyky přijímané KA, varianty KA, minimalizace KA, Mihill-Nerodova věta).
21. Regulární množiny, regulární výrazy a rovnice nad regulárními výrazy.
22. Zásobníkové automaty (jazyky přijímané ZA, varianty ZA).
23. Turingovy stroje (jazyky přijímané TS, varianty TS, lineárně omezené automaty, vyčíslitelné funkce).
24. Nerozhodnutelnost (problém zastavení TS, princip diagonalizace a redukce, Postův korespondenční problém).
25. Časová a paměťová složitost (třídy složitosti, úplnost, SAT problém).
26. Postrelační a rozšířené relační databáze (objektový a objektově relační databázový model – struktura a operace; podpora práce s XML a JSON dokumenty v databázích).
27. NoSQL databáze (porovnání relačních a NoSQL; CAP věta a ACID/BASE principy; typy NoSQL databází; dotazování v NoSQL databázích; agregace dat pomocí Map-Reduce a agregační pipeline).
28. Získávání znalostí z dat (pojem znalost; typické zdroje dat; základní úlohy získávání znalostí; analytické projekty a proces získávání znalostí z dat).

29. Porozumění datům (důvod a cíl; popisné charakteristiky dat a vizualizační techniky; korelační analýza).
30. Prostorové DB (problematika mapování prostoru, ukládání, indexace; využití).
31. Indexace (nejen) v prostorových DB (kD-Tree a Grid File (a jejich varianty), R-Tree).
32. Lambda kalkul (definice všech pojmů, operací...).
33. Práce v lambda kalkulu (demonstrace reprezentace čísel a pravdivostních hodnot a operací nad nimi).
34. Haskell – lazy evaluation (typy v jazyce včetně akcí, uživatelské typy, význam typových tříd, demonstrace lazy evaluation).
35. Prolog – způsob vyhodnocení (základní princip, unifikace, chování vestavěných predikátů, operátor řezu – vhodné a nevhodné užití).
36. Prolog – změna DB/programu za běhu (demonstrace na prohledávání stavového prostoru, práce se seznamy).
37. Model PRAM, suma prefixů a její aplikace.
38. Distribuované a paralelní algoritmy – algoritmy nad seznamy, stromy a grafy.
39. Interakce mezi procesy a typické problémy paralelismu (synchronizační a komunikační mechanismy).
40. Distribuované a paralelní algoritmy – předávání zpráv a knihovny pro paralelní zpracování (MPI).
41. Distribuovaný broadcast, synchronizace v distribuovaných systémech.
42. Klasifikace a vlastnosti paralelních a distribuovaných architektur, základní typy jejich topologií.
43. Distribuované a paralelní algoritmy – algoritmy řazení, select, algoritmy vyhledávání.
44. Bezdrátové lokální sítě (Wifi, Bluetooth).
45. Hledání minimální kostry obyčejného grafu (pojmy, stromy a kostry, Kruskalův algoritmus, Primův algoritmus).
46. Hledání nejkratších cest ze zdrojového uzlu do všech ostatních uzlů grafu (Bellman-Fordův algoritmus, Dijkstrův algoritmus).
47. Klasifikace algoritmů volby koordinátora, algoritmus Bully a jeho složitost.
48. Podmínky konsistentního globálního stavu distribuovaného systému.
49. Principy distribuovaného zpracování MapReduce, průběh a jednotlivé operace distribuovaného výpočtu pomocí MapReduce, jeho implementace v Apache Hadoop a Apache Spark.
50. Symetrická kryptografie. Vlastnosti, vlastnosti bezpečného algoritmu, délka klíče, útok silou, příklady symetrických algoritmů, Feistelovy šifry, DES, režimy činnosti, proudové šifry.
51. Asymetrická kryptografie, vlastnosti, způsoby použití, poskytované bezpečnostní funkce, elektronický podpis a jeho vlastnosti, hybridní kryptografie, algoritmus RSA, generování klíčů, šifrování, dešifrování.
52. Hašovací funkce, klíčovaný haš a MAC a jejich použití a vlastnosti.
53. Správa klíčů v asymetrické kryptografii (certifikáty X.509).
54. Základní architektury přepínačů, algoritmy pro plánování, řešení blokování, vícestupňové přepínací sítě.
55. Základní funkce směrovače, zpracování paketů ve směrovači, typy přepínání a architektur.
56. Metody pro výpočet směrování v sítích (Bellman-Ford, Dijkstra, Path vector, DUAL).
57. Řízení toku dat (flow-control) a prevence zahlcení (congestion-control) na transportní vrstvě (MP-TCP, QUIC, SCTP, DCCP).
58. Metody detekce síťových incidentů (signatury, statistické metody) a nástroje (IDS/IPS).
59. Sítě Peer-to-Peer: vlastnosti, chování, způsoby směrování. Strukturované a nestrukturované sítě.

60. Události v JavaScriptu (smyčka událostí, asynchronní programování, klientské události, obsluha událostí)
61. Přenos a distribuce webových dat (URI, protokol HTTP, proudy HTTP, CDN, XHR)
62. Bezpečnost webových aplikací (SOP, XSS, CSRF, bezpečnostní hlavičky HTTP)

# Obsah


**1 AVS – Architektura superskalárních procesorů a algoritmy zpracování instrukcí mimo pořadí, predikce skoků. 5**

**2 AVS – Paměťová konzistence a předbíhání operací čtení a zápisu, podpora virtuálního adresového prostoru. 26**

**3 AVS – Datový paralelismus SIMD, HW implementace a SW podpora. 35**

**4 AVS – Architektury se sdílenou pamětí UMA a NUMA, zajištění lokality dat. 42**

**5 AVS – Problém koherence pamětí cache na systémech se sdílenou pamětí, protokol MSI. 45**

**6 AVS – Paralelní zpracování v OpenMP: Smyčky, sekce a tasky a synchronizační prostředky. 54**

**7 MSP – Pravděpodobnost, podmíněná pravděpodobnost, nezávislost. 85**

**8 MSP – Náhodná proměnná, typy náhodné proměnná, funkční a číselné charakteristiky, významná rozdělení pravděpodobnosti. 92**

**9 MSP – Bodové a intervalové odhady parametrů, testování hypotéz o parametrech. 103**

**10 MSP – Vícevýběrové testy, testy o rozdělení, testy dobré shody. 108**

**11 MSP – Regresní analýza. 120**

**12 MSP – Markovské řetězce a základní techniky pro jejich analýzu. 128**

**13 MSP – Randomizované algoritmy (Monte Carlo a Las Vegas algoritmy). 136**

**14 SUI – Problém generalizace strojového učení a přístup k jeho řešení (trénovací, validační a testovací sada, regularizace, předtrénování, multi-task learning, augmentace dat, dropout, ...). 142**

**15 SUI – Generativní modely a diskriminativní přístup ke klasifikaci (gaussovský klasifikátor, logistická regrese, ...). 147**

**16 SUI – Neuronové sítě a jejich trénování (metoda gradientního sestupu, účelová (loss) funkce, výpočetní graf, aktivační funkce, zápis pomocí maticového násobení, ...). 154**

**17 SUI – Neuronové sítě pro strukturovaná data (konvoluční a rekurentní sítě, motivace, základní vlastnosti, použití). 162**

**18 SUI – Prohledávání stavového prostoru (informované a neinformované metody, lokální prohledávání, prohledávání v nejistém prostředí, hraní her, CSP úlohy). 169**

**19 TIN – Klasifikace formálních jazyků (Chomského hierarchie), vlastnosti formálních jazyků a jejich rozhodnutelnost. 186**

**20 TIN – Konečné automaty (jazyky přijímané KA, varianty KA, minimalizace KA, Mihill-Nerodova věta). 201**

**21 TIN – Regulární množiny, regulární výrazy a rovnice nad regulárními výrazy. 209**

**22 TIN – Zásobníkové automaty (jazyky přijímané ZA, varianty ZA). 212**

**23 TIN – Turingovy stroje (jazyky přijímané TS, varianty TS, lineárně omezené automaty, vyčíslitelné funkce). 216**

**24 TIN – Nerozhodnutelnost (problém zastavení TS, princip diagonalizace a redukce, Postův korespondenční problém). 225**

**25 TIN – Časová a paměťová složitost (třídy složitosti, úplnost, SAT problém). 237**

**26 UPA – Postrelační a rozšířené relační databáze (objektový a objektově relační databázový model – struktura a operace; podpora práce s XML a JSON dokumenty v databázích). 244**

**27 UPA – NoSQL databáze (porovnání relačních a NoSQL; CAP věta a ACID/BASE principy; typy NoSQL databází; dotazování v NoSQL databázích; agregace dat pomocí Map-Reduce a agregační pipeline). 250**

**28 UPA – Získávání znalostí z dat (pojem znalost; typické zdroje dat; základní úlohy získávání znalostí; analytické projekty a proces získávání znalostí z dat). 256**

**29 UPA – Porozumění datům (důvod a cíl; popisné charakteristiky dat a vizualizační techniky; korelační analýza). 262**

**30 UPA – Prostorové DB (problematika mapování prostoru, ukládání, indexace; využití). 270**

**31 UPA – Indexace (nejen) v prostorových DB (kD-Tree a Grid File (a jejich varianty), R-Tree). 275**

**32 FLP – Lambda kalkul (definice všech pojmů, operací, ...). 284**

**33 FLP – Práce v lambda kalkulu (demonstrace reprezentace čísel a pravdivostních hodnot a operací nad nimi). 288**

**34 FLP – Haskell – lazy evaluation (typy v jazyce včetně akcí, uživatelské typy, význam typových tříd, demonstrace lazy evaluation). 293**

**35 FLP – Prolog – způsob vyhodnocení (základní princip, unifikace, chování vestavěných predikátů, operátor řezu – vhodné a nevhodné užití). 294**

**36 FLP – Prolog – změna DB/programu za běhu (demonstrace na prohledávání stavového prostoru, práce se seznamy). 295**

**37 PRL – Model PRAM, suma prefixů a její aplikace. 296**

**38 PRL – Distribuované a paralelní algoritmy - algoritmy nad seznamy, stromy a grafy. 298**

**39 PRL – Interakce mezi procesy a typické problémy paralelismu (synchronizační a komunikační mechanismy). 299**

**40 PRL – Distribuované a paralelní algoritmy – předávání zpráv a knihovny pro paralelní zpracování (MPI). 305**

**41 PRL – Distribuovaný broadcast, synchronizace v distribuovaných systémech. 306**

**42 PRL – Klasifikace a vlastnosti paralelních a distribuovaných architektur, základní typy jejich topologií. 316**

**43 PRL – Distribuované a paralelní algoritmy – algoritmy řazení, select, algoritmy vyhledávání. 326**

**44 BMS – Bezdrátové lokální sítě (Wifi, Bluetooth). 349**

**45 GAL – Hledání minimální kostry obyčejného grafu (pojmy, stromy a kostry, Kruskalův algoritmus, Primův algoritmus). 362**

**46 GAL – Hledání nejkratších cest ze zdrojového uzlu do všech ostatních uzlů grafu (Bellman-Fordův algoritmus, Dijkstrův algoritmus). 372**

**47 PDI – Klasifikace algoritmů volby koordinátora, algoritmus Bully a jeho složitost. 381**

**48 PDI – Podmínky konsistentního globálního stavu distribuovaného systému. 387**

**49 PDI – Principy distribuovaného zpracování MapReduce, průběh a jednotlivé operace distribuovaného výpočtu pomocí MapReduce, jeho implementace v Apache Hadoop a Apache Spark. 392**

**50 KRY – Symetrická kryptografie. Vlastnosti, vlastnosti bezpečného algoritmu, délka klíče, útok silou, příklady symetrických algoritmů, Feistelovy šifry, DES, režimy činnosti, proudové šifry. 399**

**51 KRY – Asymetrická kryptografie, vlastnosti, způsoby použití, poskytované bezpečnostní funkce, elektronický podpis a jeho vlastnosti, hybridní kryptografie, algoritmus RSA, generování klíčů, šifrování, dešifrování. 413**

**52 KRY – Hašovací funkce, klíčovaný haš a MAC a jejich použití a vlastnosti. 418**

**53 KRY – Správa klíčů v asymetrické kryptografii (certifikáty X.509). 425**

**54 PDS – Prerekvizity k ostatním otázkám. 431**

**55 PDS – Základní architektury přepínačů, algoritmy pro plánování, řešení blokování, vícestupňové přepínací sítě. 435**

**56 PDS – Základní funkce směrovače, zpracování paketů ve směrovači, typy přepínání a architektur. 449**

**57 PDS – Metody pro výpočet směrování v sítích (Bellman-Ford, Dijkstra, Path vector, DUAL). 465**

**58 PDS – Řízení toku dat (flow-control) a prevence zahlcení (congestion-control) na transportní vrstvě (MP-TCP, QUIC, SCTP, DCCP). 471**

**59 PDS – Metody detekce síťových incidentů (signatury, statistické metody) a nástroje (IDS/IPS). 486**

**60 PDS – Sítě Peer-to-Peer: vlastnosti, chování, způsoby směrování. Strukturované a nestrukturované sítě. 493**

**61 WAP – Události v JavaScriptu (smyčka událostí, asynchronní programování, klientské události, obsluha událostí). 500**

**62 WAP – Přenos a distribuce webových dat (URI, protokol HTTP, proudy HTTP, CDN, XHR). 503**

**63 WAP – Bezpečnost webových aplikací (SOP, XSS, CSRF, bezpečnostní hlavičky HTTP). 510**

# Kapitola 1

# AVS – Architektura superskalárních procesorů a algoritmy zpracování instrukcí mimo pořadí, predikce skoků.

## 1.1 Zdroje

- `AVS_2019-09-23.mp4`
- `AVS_2019-09-30.mp4`
- `AVS_2019-10-07.mp4`
- `AVS_2019-10-14.mp4`
- `AVS-01.pdf`
- `AVS-02.pdf`
- `AVS-03.pdf`
- `AVS-04.pdf`

## 1.2 Úvod a kontext

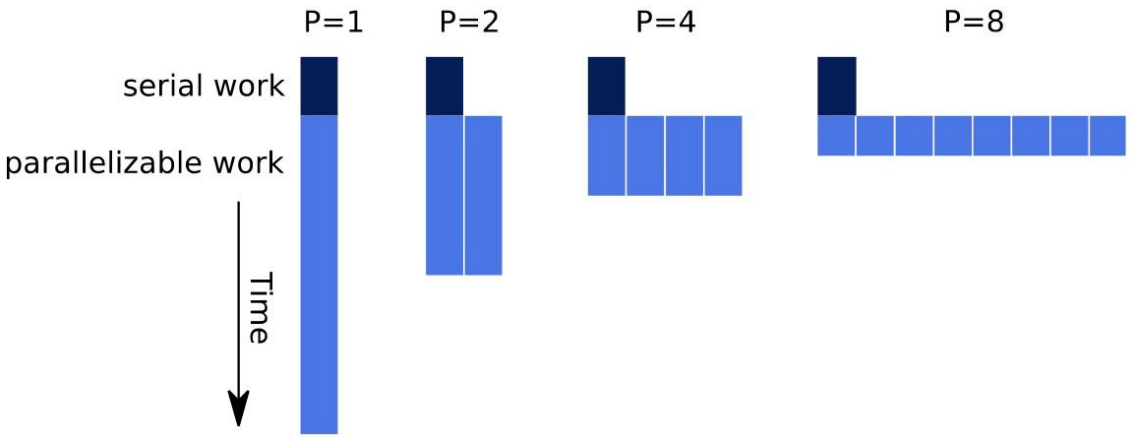
- Chceme zvyšovat výpočetní výkon. Jak na to?
  - Zvyšovat počet tranzistorů, tam brzy narazíme na fyzikální limity.
  - Počítat efektivněji (bez čekání) a paralelizovat.
- **Moorův zákon**
  - Počet tranzistorů na čipu se zdvojnásobuje každé dva roky při zachování stejné ceny.
  - Se snižováním rozměrů tranzistorů se rychlost zvyšuje, příkon snižuje.
  - Nejedná se o fyzikální zákon, spíše jde o empirický vztah vypozorovaný z historických dat.
- **Úrovně paralelismu**
  - Paralelní provádění instrukcí (ILP, *instruction level parallelism*).
  - Střídání vláken na CPU (TLP, *thread level parallelism*).
  - Paralelní zpracování dat (DLP, *data level parallelism*).

- Rozdělení úloh na vlákna/procesy na více jader.

- **Amdahlův zákon**
  - Vyjadřuje kdy mý smysl něco optimalizovat (zrychlovat) a kdy ne, tj. které části má smysl paralelizovat.

$$\lim_{P \to \infty} S(P) = \frac{1}{\alpha}$$

  - kde
    * $P$ je počet procesorů;
    * $S(P)$ funkce dosáhnutého zrychlení v závislosti na $P$;
    * $\alpha$ sekvenční část úlohy (ze své podstaty).
  - Myšlenka: Pro nekonečný počet procesorů bude maximální možné zrychlení převrácená hodnota poměru z principu sekvenční části ku paralilizovatelné.
  - Kdy má smysl paralelizovat? Když z principu sekvenční část je malá.




Obrázek 1.1: Amdahlův zákon.

## 1.3 Vývoj procesorů

- Metriky:
  - CPI (*clocks per instruction*) – Kolik taktů trvá jedna instrukce v průměru.
  - IPC (*instructions per clock*) – Kolik instrukcí se vykoná za jeden takt v průměru.

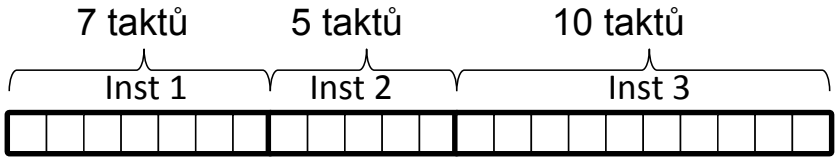
$$\text{IPC} = \frac{1}{\text{CPI}}$$

  - R, IPS (*instructions per second*) – Výkon; kolik instrukcí se vykoná za sekundu.
  - f [Hz] – Frekvence; počet taktů procesoru za sekundu.

### 1.3.1 Subskalární procesor

- Sekvenční zpracování instrukcí (Von Neumannova technika).
- Každá instrukce může trvat jiný počet taktů.

- Příklad:




Obrázek 1.2: Vykonávání instrukcí subskalárním procesorem; 3 instrukce, 22 taktů.

  - CPI, IPC:

$$\mathrm{CPI} = \frac{22}{3} \approx 7,33$$

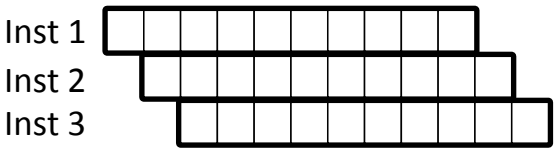
$$\mathrm{IPC} \approx \frac{1}{7,33} \approx 0,14$$

  - IPS při frekvenci 5 GHz:

$$\mathrm{IPS} = 5 \times 10^9 \times \frac{3}{22} \approx 682 \times 10^6$$

### 1.3.2 Skalární procesor

- Řetězená linka („pipeline“).
- Všechny instrukce musí trvat stejný počet taktů.
- Cíl: jedna instrukce každý takt (v limitě $N \to \infty$), reálně je to horší, kvůli tzv. pokutám (viz dále).
- Příklad:




Obrázek 1.3: Vykonávání instrukcí skalárním procesorem; 3 instrukce každá trvá 10 taktů.

  - CPI, IPC:

$$\mathrm{CPI} = \frac{10 + 3 - 1}{3} = 4$$

$$\mathrm{IPC} = \frac{1}{4} = 0,25$$

  - IPS při frekvenci 5 GHz:

$$\mathrm{IPS} = 5 \times 10^9 \times 0,25 \approx 1250 \times 10^6$$

### 1.3.3 Superskalární procesor

- Superřetězená linka – několik řetězených linek.
- Možnost vykonávat několik instrukcí za takt.
- Jak to funguje?

- Mějme $N$ linek (dnes běžné 6-9) a příslušné dekodéry instrukcí.
- Dekodéry vybírají instrukce z linek tak, aby je bylo možné nezávisle zpracovat.
- Dokáží vydávat až $N$ instrukcí za takt.

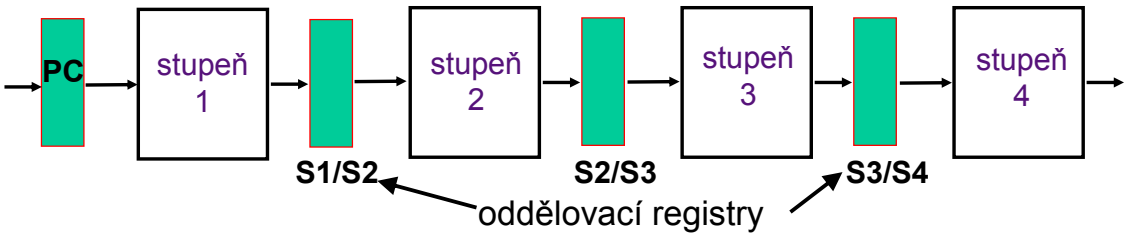
- Platí:

$$\text{IPC} < N$$

- Využívá přeskládání pořadí instrukcí tak, jak procesoru vyhovuje, ale nemění sémantiku programu.

## 1.4 Architektura skalárních procesorů

- Jak se realizuje?
  - Programová logika je rozdělena na části a mezi ně jsou vloženy oddělovací registry.




Obrázek 1.4: Ideální řetězená linka.

- Jaké jsou předpoklady pro zřetězené zpracování?
  - Nepřetržitý přísun dat.
  - Instrukce musí být možná rozdělit na sekvenci nezávislých kroků.
  - Trvání kroků by mělo být přibližně stejné.
- Co má vliv na celkové dosažitelné zrychlení?
  - Přestávky a pokuty způsobené zpracováním závislostí (např. před sčítáním musím načíst data z paměti).
  - Náběh a doběh („naplnění“ a „vypláchnutí“ linky).
  - Zpoždění oddělovacích registrů.
- Zrychlení skalární linky oproti subskalární.
  - Každá instrukce probíhá v několika krocích, tzv. mikroinstrukcích.
  - Výkonnost ($R$):

$$R = \text{IPS} = \frac{\text{počet instrukcí}}{\text{čas}}$$

  - Zrychlení linky pro $N$ instrukcí ($S_N$):

$$S_N = \frac{\text{skalární výkon}}{\text{subskalární výkon}} = \frac{\text{subskalární čas}}{\text{skalární čas}}$$

- Potom:

$$\text{subskalární čas} = N \times t_i = N \times \tau \times k$$

$$\text{skalární čas} = (N - 1 + k) \times (\tau + t_d)$$

  * $N$ je počet instrukcí;
  * $t_i$ je průměrná doba trvání instrukce;
  * $k$ je počet mikroinstrukcí (počet stupňů);
  * $\tau$ je doba trvání mikroinstrukce;
  * $t_d$ je doba zpoždění registru.

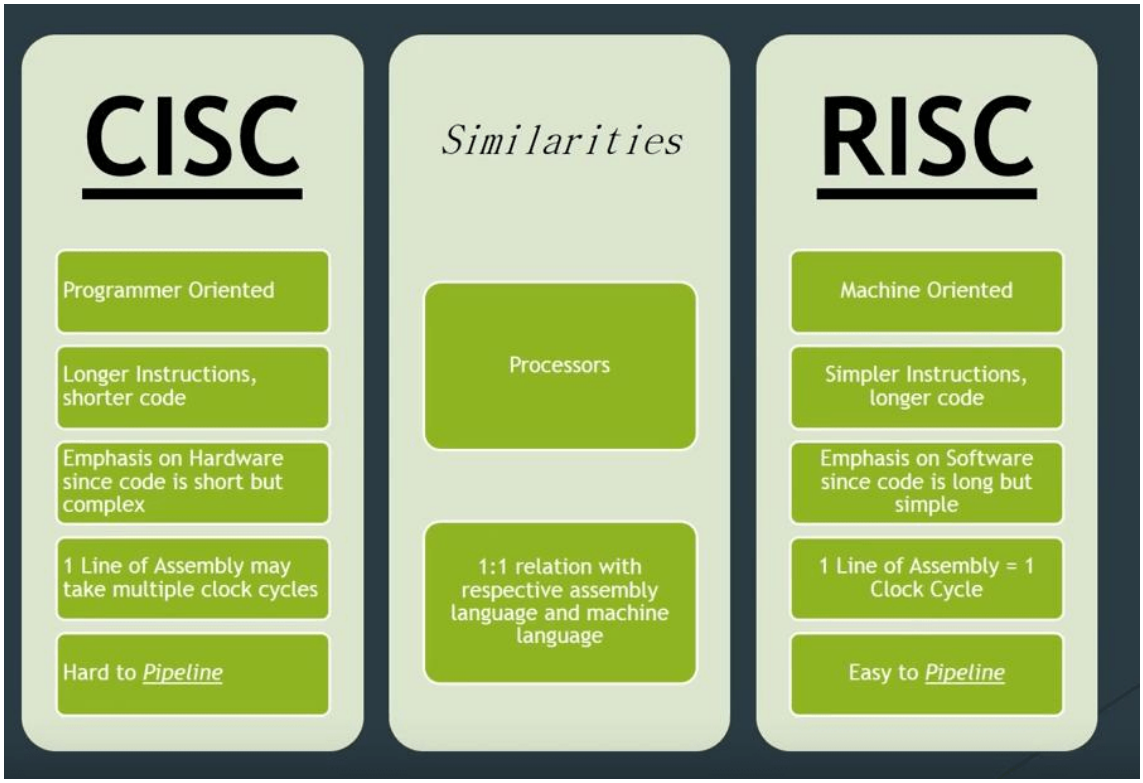
- Zrychlení při pozastavování linky.
  - V realitě se maximálnímu dosažitelnému zrychlení budeme chtít pouze přiblížit, jsou případy, kdy je nutné linky zastavit (různé kolize, viz dále).
  - Dobu zastavování linky můžeme zprůměrovat na pokutu $q$ taktů vztaženou na každou instrukci.
  - Počet taktů na 1 instrukci je pak CPI $= 1 + q$.

$$\text{skalární čas} = (N - 1 + k) \times (\tau + t_d) \times (1 + q)$$

  - Zpomalení registrů často nemusíme řešit – je příliš malé, proto lze zanedbat.

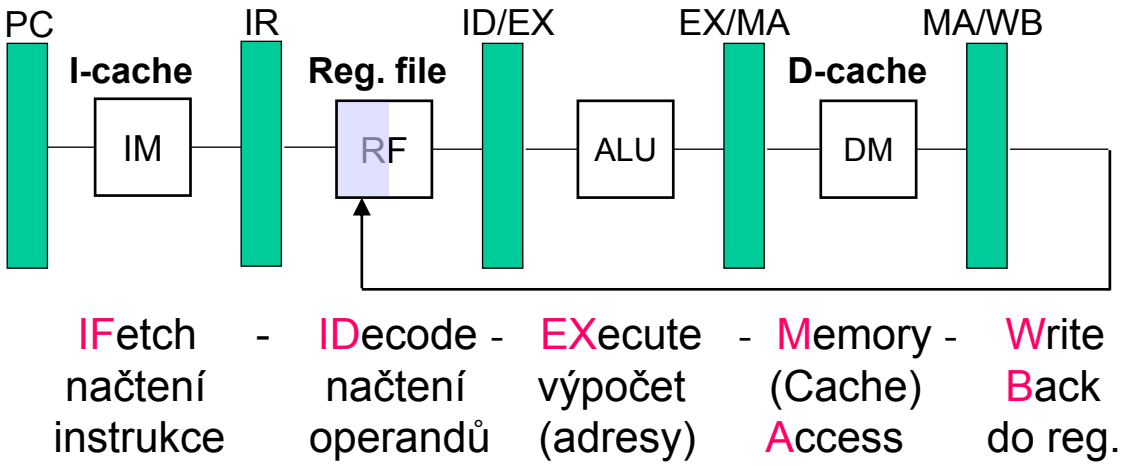
## 1.5 Architektura skalárních procesorů RISC

- RISC (*reduced instruction set computer*) – Redukovaná (minimální) instrukční sada.
- CISC (*complex instruction set computing*) – Komplexní instrukční sada.
- Dnešní moderní procesory jsou CISC, ale „jádro“ mají RISC.
- CISC funguje pouze jako nadstavba ve fázi decode, kdy je CISC instrukce dekódována na několik RISC instrukcí.
- RISC má velké množství registrů, v jednom taktu umí číst i zapisovat (brány zápisu, brány čtení).

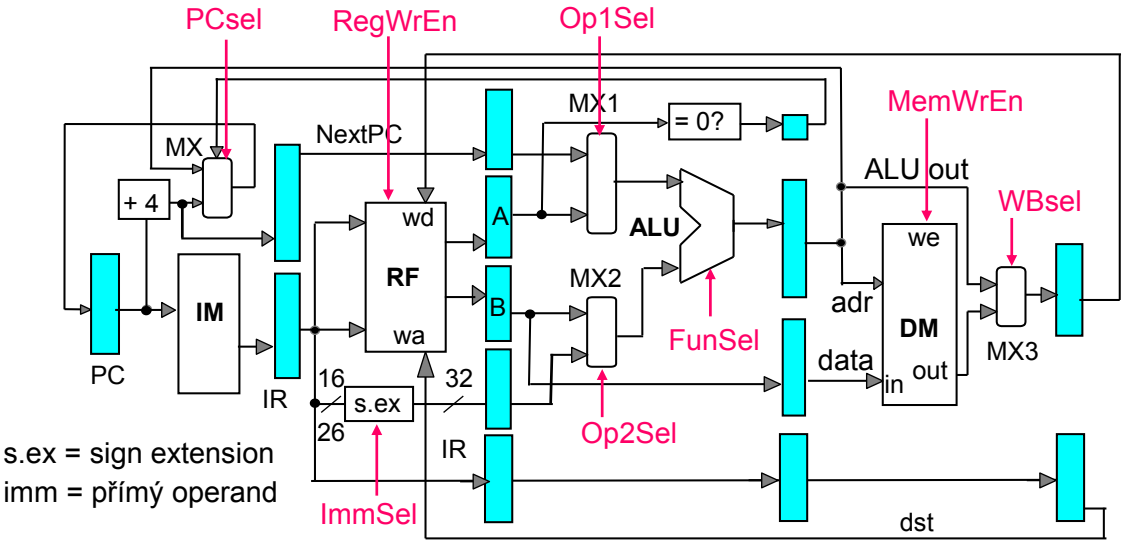



Obrázek 1.5: CISC vs RISC.

- Charakteristika RISC instrukcí:
  - Všechny instrukce mají stejnou velikost (32/64 bitů).
  - Málo formátů instrukcí a formáty jsou pravidelné.
  - Přístup do paměti mají pouze instrukce LOAD a STORE.
  - Hodně registrů.
- Formáty instrukcí:
  - Register type: `op src1 src2 dst` (např. sčítání, odčítání).
  - Imm type: `op src/dst address` (např. načítání, ukládání).
  - Jump type: `target_address` (např. skok).
- Stupně řetězeného zpracování:
  - IF (*instruction fetch*) – Načtení instrukce z instrukční cache na základě hodnoty v PC registru.
  - ID (*instruction decode*) – Dekódování instrukce, získání opcode, získání dat z registrů do procesoru.
  - EX (*execute*) – Počítání, ADD sčítá, LOAD/STORE počítá adresu, JUMP počítá podmínku.
  - MA (*memory access*) – ADD nic, LOAD čte z datové cache, STORE zapisuje do datové cache, JUMP zapisuje do PC registru.
  - WB (*write back*) – ADD zapisuje výsledek do registru DST, LOAD zapisuje do registru, STORE nedělá nic.




Obrázek 1.6: Stupně řetězeného zpracování RISC. Zeleně jsou oddělovací registry, dále instrukční cache, registrové pole, ALU, datová cache.




Obrázek 1.7: Klasická RISC pipeline (fáze IF, ID, EX, MA, WB). To co se při fázi WB zapíše do registrového pole lze hned v tom samém taktu použít!

## 1.6 Architektura superskalárních procesorů

- Jak zrychlit dobu výpočtu programu?

$$\text{doba výpočtu} = IC \times CPI \times T$$

  - IC (*instruction count*) – Počet provedených instrukcí. Jak snížit?
    - Optimalizací kódu, napsat program líp.
  - CPI – Počet taktů na instrukci. Jak snížit?
    - Více instrukcí v jednom stupni – $m$-cestný superskalární procesor.
    - Reálně dosažitelná hodnota IPC (kolik instrukcí končí v taktu) je vždy značně nižší než $m$.

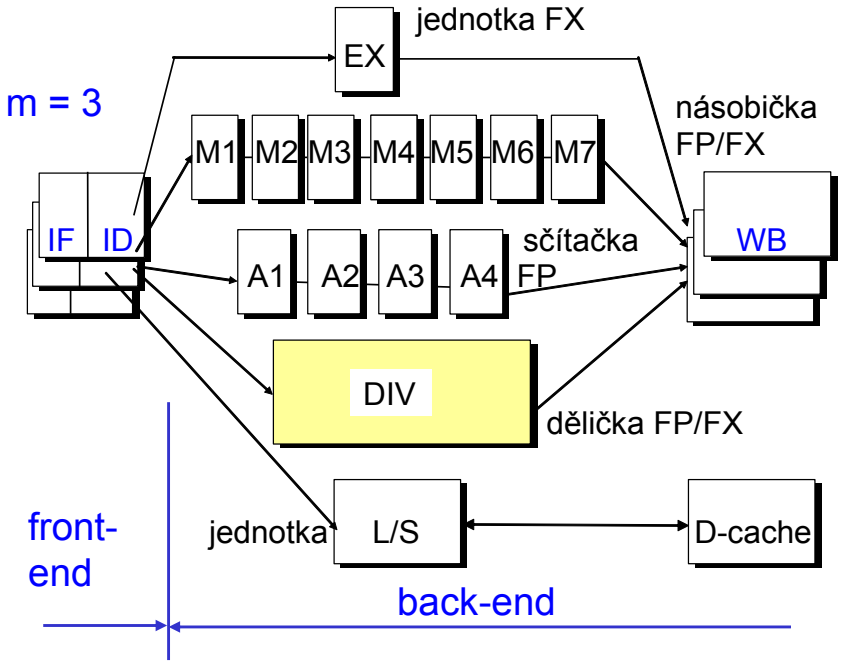
- T – doba vykonání jednoho taktu. Jak snížit?
  * Větší počet stupňů linky (až 30).
  * S tím jsou spojené velké pokuty, vyšší příkon, doba zpoždění registrů.

### 1.6.1 Fáze superskalárního zpracování

- **Front end** – IF, ID.
  - Načítá a dekóduje několik instrukcí najednou, počet se mění dynamicky.
  - $M$-cestný superskalár znamená, že vydává až $m$ instrukcí do funkčních jednotek v 1 taktu.
- **Back end** – EX, MA, WB.
  - Provádí a ukládá výsledky několika instrukcí souběžně.
  - Některé stupně jsou rozděleny na podstupně.

### 1.6.2 Dělení superskalárních procesorů

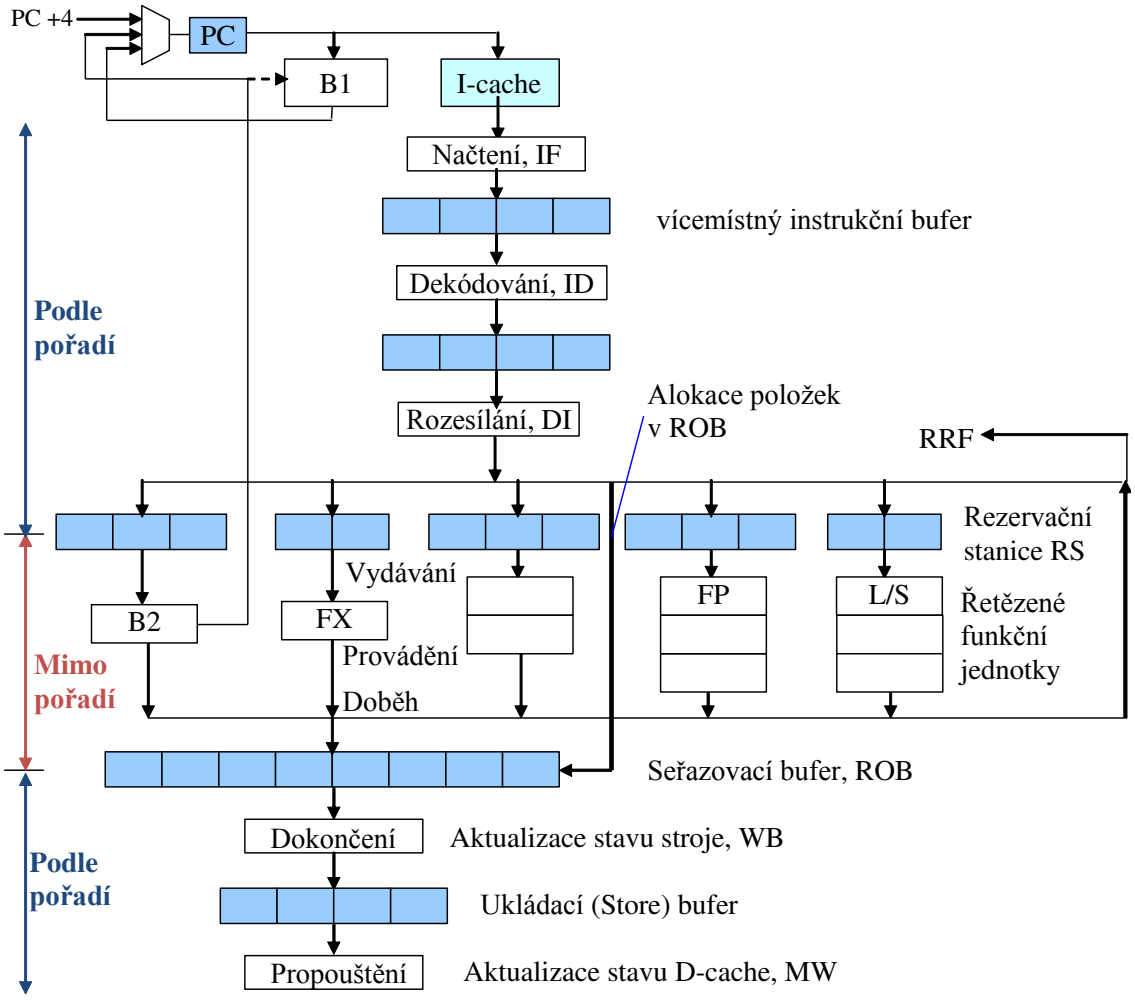
- Dle způsobu jakým instrukce opouštějí front-end.
- **INO – in-order**
  - Instrukce jsou vykonávány podle pořadí v programu, po vyřešení konfliktů.
  - Jednotlivé stupně mohou mít různé zpoždění nebo propustnost instrukcí za takt.
  - Front end opouští až $m$ instrukcí v jednom taktu.




Obrázek 1.8: Příklad skalárního in-order procesoru.

- **OOO – out-of-order**

- Mohou vykonávat instrukce mimo pořadí v programu, nepravé konflikty vyřešeny přejmenováním v HW, RAW řešeny čekáním rozpracovaných instrukcí.
- Back end – Vykonává se out-of-order.
- Front end – Vykonává se in-order (jinak by to pochopitelně byl nesmysl).
- První prediktor skoku se nechází už ve fázi IF.
- Reorder Buffer (ROB) – Seřazovací paměť.
- Rename Register Field (RRF) – Registry pro přejmenování.
- Dispatch (DI) – Nová fáze instrukce pro rozeslání instrukcí do rezervačních stanic a ROB.




Obrázek 1.9: Generická OOO superskalární architektura. Po dekódování se instrukce mohou zpracovat v libovolném pořadí, ale na konci se musí opět seřadit.

### 1.6.3 Rysy superskalárních procesorů

- Paralelní řetězené linky (INO i OOO).
  - Časový i prostorový paralelismus (paralelní načítání, dekódování, vydávání instrukcí do FJ, jejich paralelní provádění a dokončování).
- Přejmenování registrů v HW (OOO).

  - Odstraní konflikty WAR a WAW (viz dále).
- Dynamické plánování instrukcí out-of-order (OOO).
  - Po dekódování čekají instrukce na své operandy, které se tvoří. Jakmile jsou operandy připraveny, spustí se operace.
  - Instrukce, včetně přístupů do paměti, jsou zpracovávány v jiném pořadí oproti pořadí v programu (OOO).
- Seřazovací paměť (OOO).
  - Stupeň WB pomocí ní zajišťuje ukládání výsledků v pořadí určeném zdrojovým kódem.
- Spekulativní zpracování instrukcí (OOO).
  - Spekulace, že skok dopadne podle predikce nebo že dopředu načtená data se již nezmění.

## 1.7 Konflikty při řetězeném zpracování instrukcí

- Instrukce může záviset na něčem co produkuje dřívější instrukce.
  - Datová závislost – závislost se může týkat hodnot dat.
  - Řídící závislost – závislost se může týkat adresy příští instrukce.
- Instrukce v řetězené lince může potřebovat prostředek, který právě používá jiná instrukce.
  - Strukturní závislost.
- Jak konflikty řešíme?
  - Přejmování registrů.
  - Vyplněný prázdných taktů užitečnými instrukcemi.
  - Přehození pořadí instrukcí bez změny sémantiky programu s hlídáním zpoždění mezi operacemi.
  - Rozbalení smyček.
  - SW řetězení smyček v programu, případně v kombinaci s rozbalením.
  - Překladač zná kolik má jaká instrukce zpoždění.

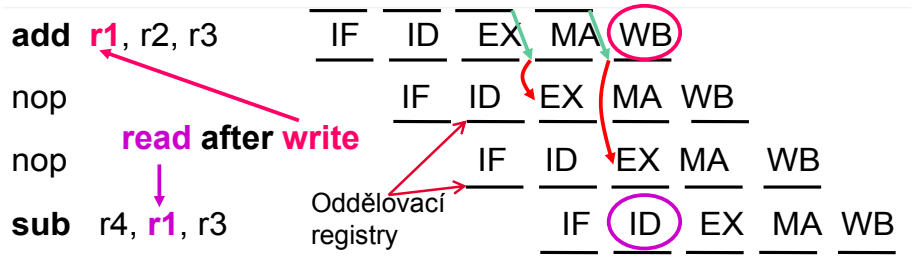
### 1.7.1 Datová závislost

- Dále dělíme na:
  - **Pravé** (*true dependencies*) – Nejde s nimi „nic moc dělat“, definují sémantiku programu.
  - **Nepravé** (*false dependencies*) – Mohou vzniknout pouze u superskalárních procesorů při vykonávání instrukcí mimo pořadí v programu.

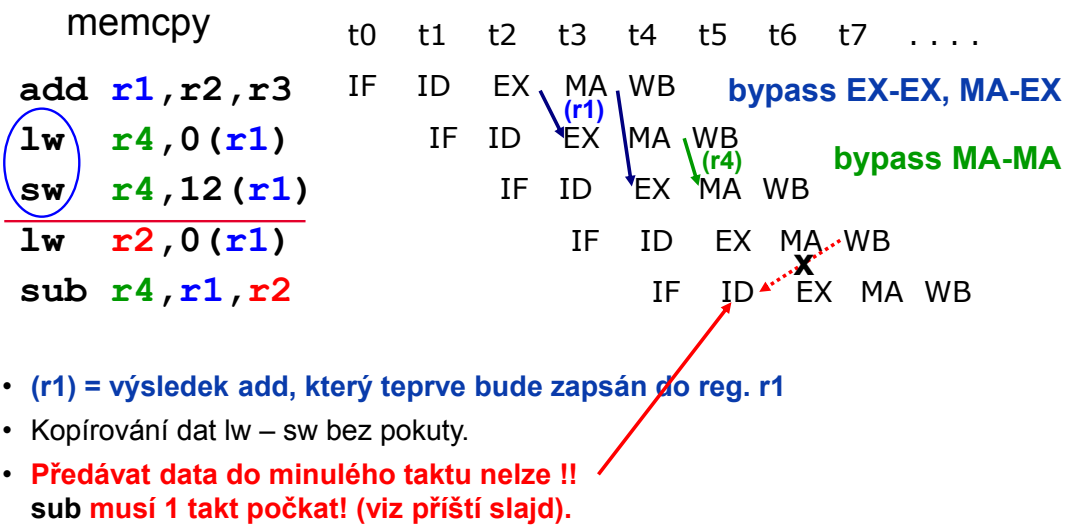
#### RAW (*read after write*)

- Pravý konflikt.

- Instrukce $I_2$ pracuje s výsledkem instrukce $I_1$, který ještě nebyl vypočítán nebo načten. I když je instrukce $I_2$ provedena po instrukci $I_1$, tak $I_1$ instrukce byla zpracována pouze částečně v rámci pipeline.
- Řešení: Nová datová cesta (zkratka), tzv. **bypass**. Např. můžeme dostat data z výstupu ALU na vstup ALU v dalším taktu.



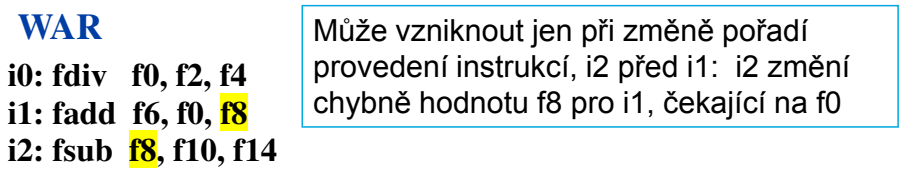

Obrázek 1.10: Příklad RAW (nop značí čekání), instrukce sub čte výsledek zapsaný instrukcí add. Lze řešit bypassem.




Obrázek 1.11: Příklad RAW, instrukce sub čte výsledek zapsaný instrukcí lw. Nelze (kompletně) řešit bypassem, stále bude nutné čekat.

**WAR** (*write after read*)

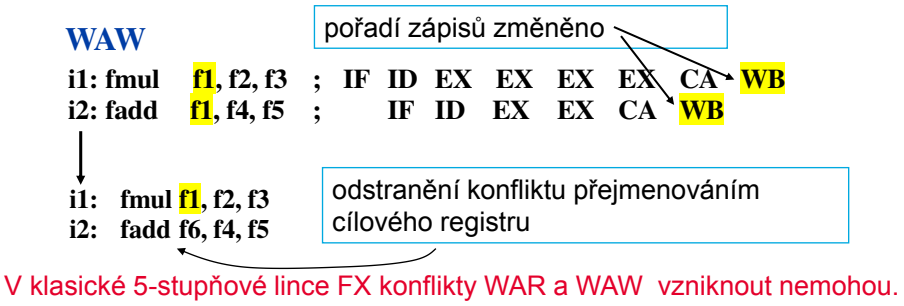
- Nepravý konflikt.
- Přepsání dat, která ještě někdo měl číst.




Obrázek 1.12: Příklad WAR.

**WAW** (*write after write*)

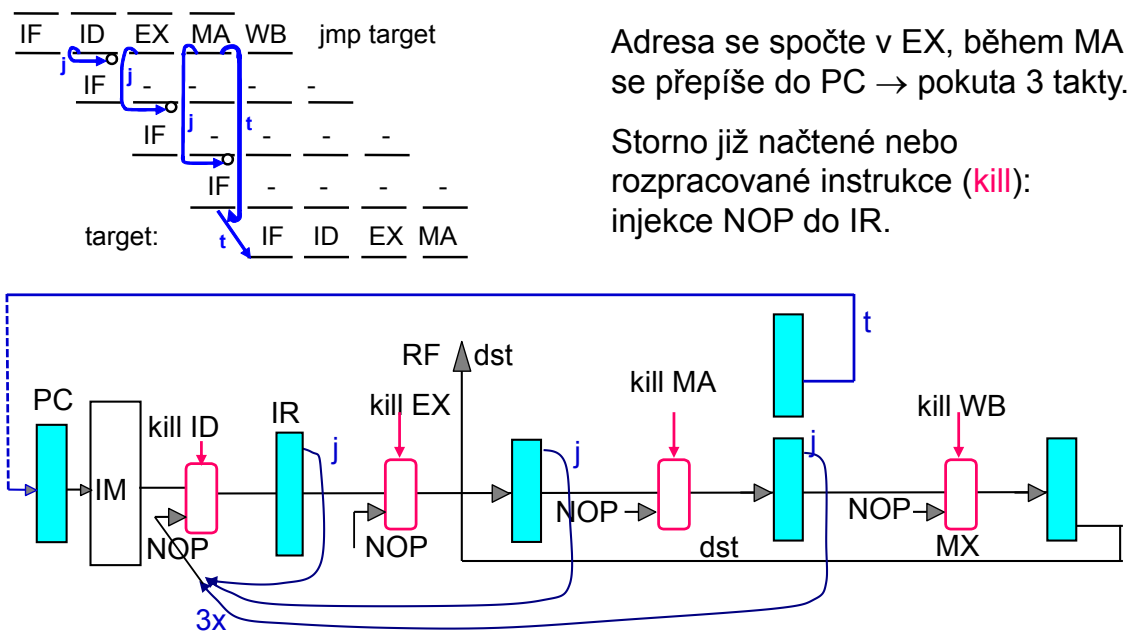
- Nepravý konflikt.
- Instrukce, která byla naplánována později se přesune dopředu a přepíše hodnotu registru, kterou chtěla zapsat nějaká jiná instrukce.




Obrázek 1.13: Příklad WAW.

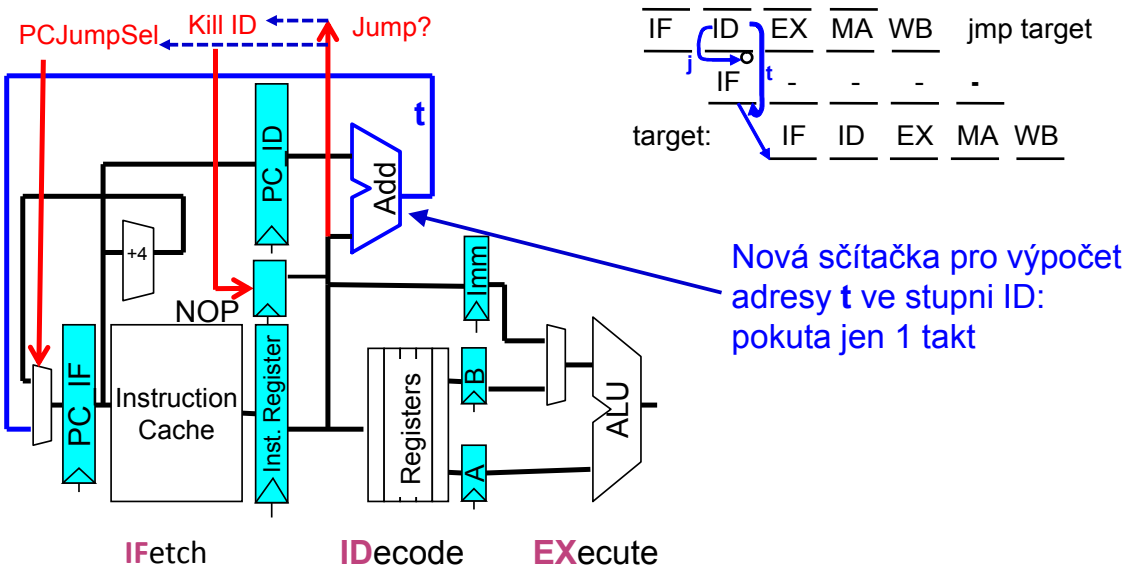
### 1.7.2 Řídící závislost

- Týká se podmíněných/nepodmíněných skoků.
- Ve fázi EX se počítá jestli a kam skočit.



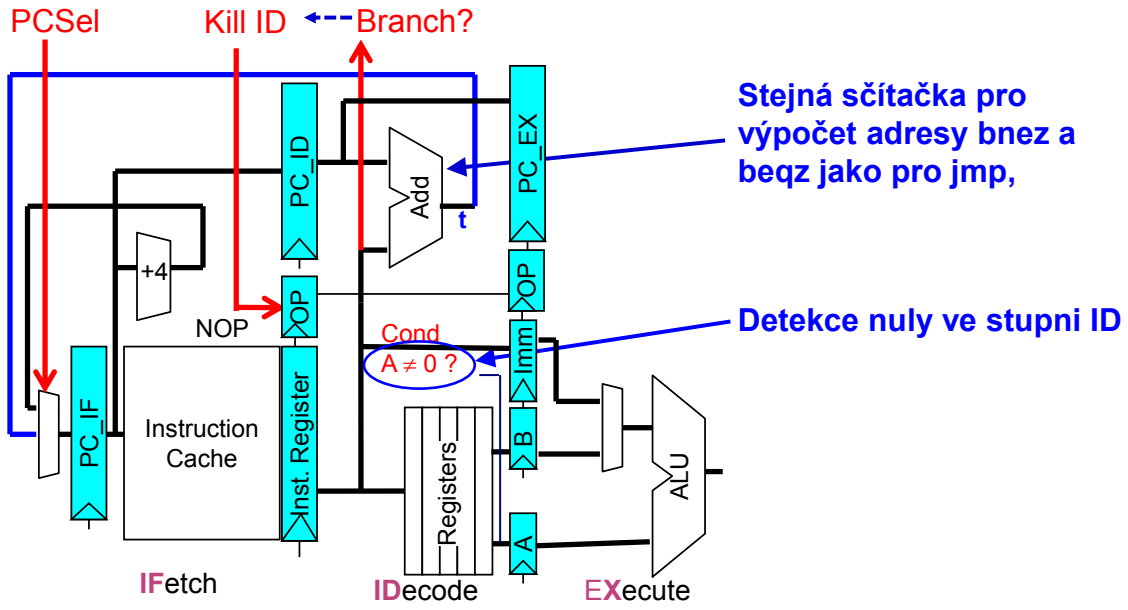

Obrázek 1.14: Řídící konflikt.

- Co dělat abychom nemuseli čekat 3 takty na zjištění adresy další instrukce při skoku?
  - Pro **nepodmíněný skok** přidáme další hardware (sčítačku) ve stupni ID, zjistíme, že se jedná o skok a ihned můžeme připočíst / odečíst offset a víme odkud brát další instrukci. Výsledná pokuta bude 1 takt.




Obrázek 1.15: Úprava HW pro nepodmíněné skoky (jump).

– Pro **podmíněný skok** přidáme také další hardware (sčítačku, komparátor pro test na nulu) ve stupni ID. Pokud skočím, je pokuta 1 takt, pokud ne, žádný.




Obrázek 1.16: Úprava HW pro podmíněné skoky (bnez, beqz).

– U složitějších podmínek můžu vyhodnotit až po EX fázi, nutně tedy pokuta 1 takt vždy, pokud skočím tak 2 takty.

### 1.7.3 Strukturní závislost

- WB a ID – Oba přistupují do instrukčního registru, ale zápisové a čtecí brány jsou dostupné současně, lze tedy vykonávat zároveň.
- MA a IF – Načítání instrukcí a dat, řeší se oddělením cache pro instrukce a data.

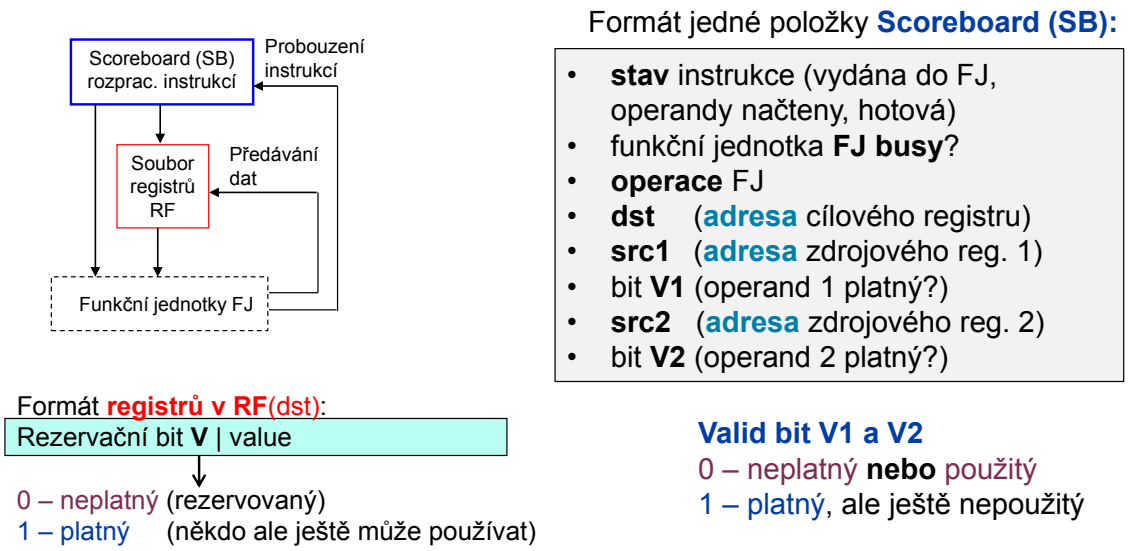
- Nic horšího nastat nemůže.

## 1.8 Algoritmy zpracování instrukcí mimo pořadí

- Jak určit kdy se má která instrukce vykonat? Máme algoritmy dynamického plánování instrukcí.
- Myšlenka: spočítá se graf závislostí instrukcí, na základě kterého se bude vybírat v každém taktu $m$ nezávislých instrukcí, které je možné vykonat.
- Dynamické plánování instrukcí – Instrukce jsou vydávány do FJ a prováděny mimo pořadí v programu, pokud mezi nimi nejsou konflikty a FJ jsou volné.
- Zabýváme se pouze datovými konflikty.
  - Pravé datové konflikty se řeší čekáním / přejmováním.
  - Nepravé datové konflikty se řeší dynamickým plánováním, změním pořadí instrukcí tak, abych stále mohl něco dělat.
- **Seřazovací paměť** (ROB, *re-order buffer*)
  - Kruhová vyrovnávací paměť rozpracovaných instrukcí, které jsou uloženy ve frontě FIFO.
  - Instrukce jsou vloženy do ROB při vydání do rezervační stanice (fáze decode).
  - Do datové cache musí být výsledky zapsány v pořadí.
  - Propouštění instrukcí pouze z čela ROB.
  - Formát:
    * Typ instrukce – aritmetická, LOAD / STORE, branch.
    * Cílový registr – adresa.
    * Flag – stav instrukce (například jestli instrukce doběhla ve FJ).
    * Hodnota – spočtená instrukcí, zatím nezávazná.

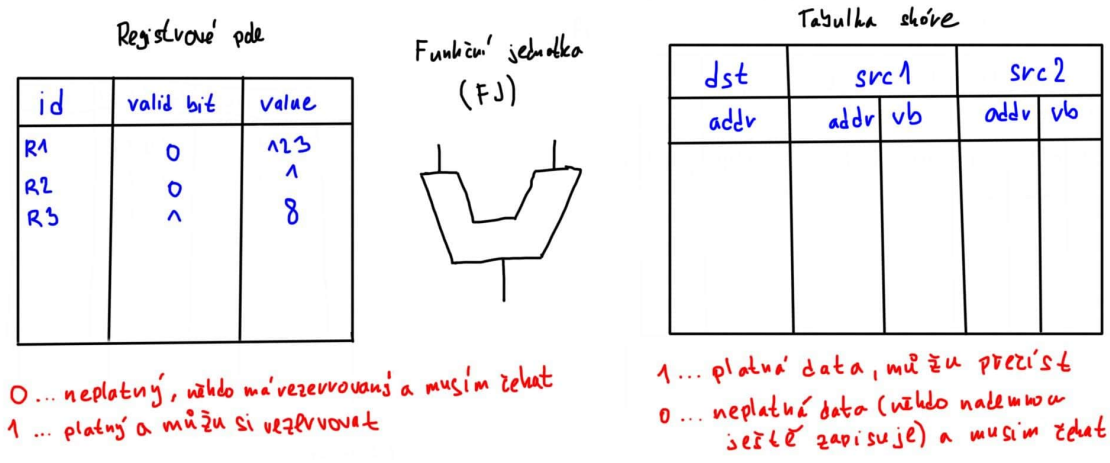
### 1.8.1 Scoreboarding (Thorntonův algoritmus)

- Registruje všechny konflikty (RAW, WAW, WAR) v tabulce rozpracovaných instrukcí a udržuje jejich skóre (SB).
- SB vydá instrukce dál jen když nejsou v konfliktu s ostatními instrukcemi v SB.
- Přejmenování registrů neprobíhá.
- Konflikty RAW, WAR a WAW se řeší čekáním.




Obrázek 1.17: Princip algoritmu.

- Formát:
  - Registrové pole (*register field*) – Identifikátor registru, Valid bit, Hodnota.
  - Tabulka skóre (*score board*) – Stav instrukce, Funkční jednotka, Operace, DST (adresa), SRC1 (adresa), Valid bit1, SRC2 (adresa), Valid bit2.
- **Postup algoritmu:**
  1. Rezervace cílového registru v poli registrů (kvůli WAW konfliktu).
     (a) Registr má valid bit na 1. Hodnota v něm je platná a žádná instrukce do něho nezapisuje. Můžu nastavit na 0 a tím si registr zarezervovat.
     (b) Registr má valid bit na 0. Do registru právě generuje výsledek jiná instrukce, musím čekat.
  2. Při rezervování cílového registru je přidán daný záznam do tabulky skóre a je vyplněn příslušnými hodnotami z registrového pole (kvůli RAW konfliktu).
     (a) Pokud jsou valid bity zdrojových registrů nastaveny na 1, znamená to, že všechny předcházející instrukce, které s nimi pracovali už skončili. Můžu z nich natáhnout data do funkční jednotky a jejich valid bity v tabulce skóre nastavim na 0, aby do registrů mohli zapisovat další instrukce.
     (b) Pokud je aspoň jeden valid bit zdrojových registrů na 0, znamená to, že data v nich ještě nejsou aktuální. Jiná instrukce do nich ještě zapisuje a já musím čekat.
  3. Mám výsledek z funkční jednotky (kvůli WAR konfliktu).
     (a) Pokud se v tabulce skóre můj cílový registr vůbec nenachází a nebo se nachází jako zdrojový a má valid bit na 0, tak můžu výsledek zapsat do registrového pole.
     (b) Pokud se v tabulce skóre můj cílový registr nachází jako zdrojový a má valid bit na 1, tak jiná instrukce s jeho starou hodnotou ještě bude pracovat a já musím čekat.
  4. Smažu záznam z tabulky skóre.

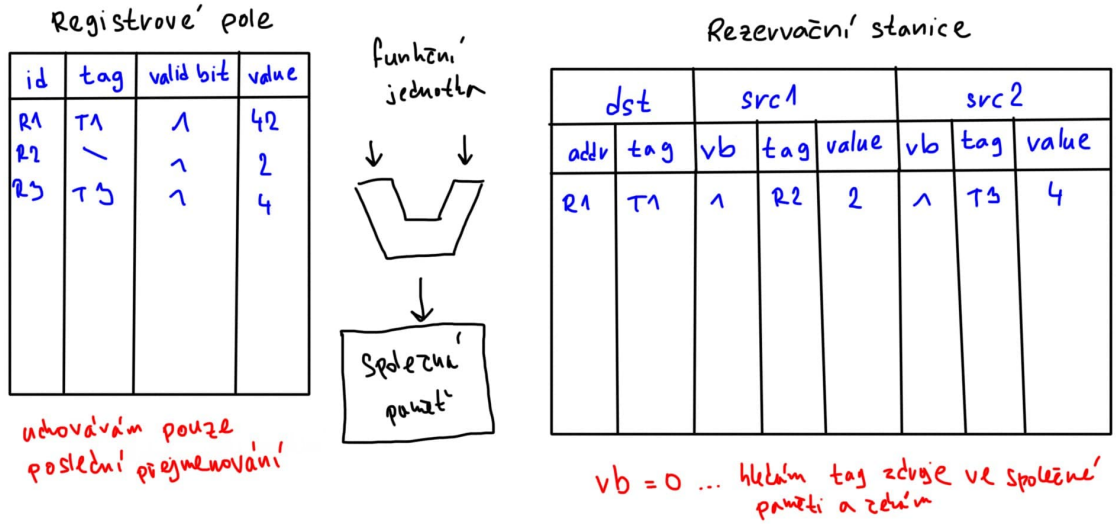



Obrázek 1.18: Příklad algoritmu Scoreboarding.

### 1.8.2 Rezervační stanice (Tomasolův algoritmus)

- Buď individuální nebo společná rezervační stanice, konflikty se řeší přejmenováním, tj. uložením do jiného registru.
- Registr může být přejmenován několikrát, ale do původního registru nemusí být uložena každá hodnota, která vznikne prácí s jeho přejmenovanou verzí.
- Myšlenka: přejmenovávám cílové registry, tak řeším nepravé konflikty a výsledky ukládám jinám, tj. nečekám.
- Shrnutí konfliktů: RAW se řeší čekáním, nelze jinak, ale mezitím můžeme vykonávat jinou instrukci. WAR a WAW se řeší pomocí přejmování a příznaků.
- Formát:
  - Registrové pole (register field): Jméno, Tag, Valid bit, Hodnota.
  - Rezervační stanice: DST adresa, DST tag, SRC1 valid bit, SRC1 tag, SRC1 hodnota, SRC2 valid bit, SRC2 tag, SRC2 hodnota.
- **Postup algoritmu:**
  1. Cílovému registru v poli registrů dám nové jméno, tzv. tag, v rezervační stanici vytvořím nový záznam a vyplním DST.
     (a) V poli registrů udržuji pouze poslední přejmování.
  2. Natáhnutí zdrojových registrů do rezervační stanice.
     (a) Pokud jsou valid bity zdrojových registrů v poli registrů na 1, vyplním v rezervační stanici celý SRC (valid bit, tag a hodnotu).
     (b) Pokud nějaký z valid bitů zdrojových registrů je na 0, vyplním v rezervační stanici valid bit a tag (poslední přejmování), na hodnotu čekám – dodá mi ji FJ.
  3. Pokud mají oba zdrojové operandy valid bit na 1, tak je oprace spuštěna a funkční jednotka vypočítá výsledek, který dá na společnou sběrnici (id, tag, value).

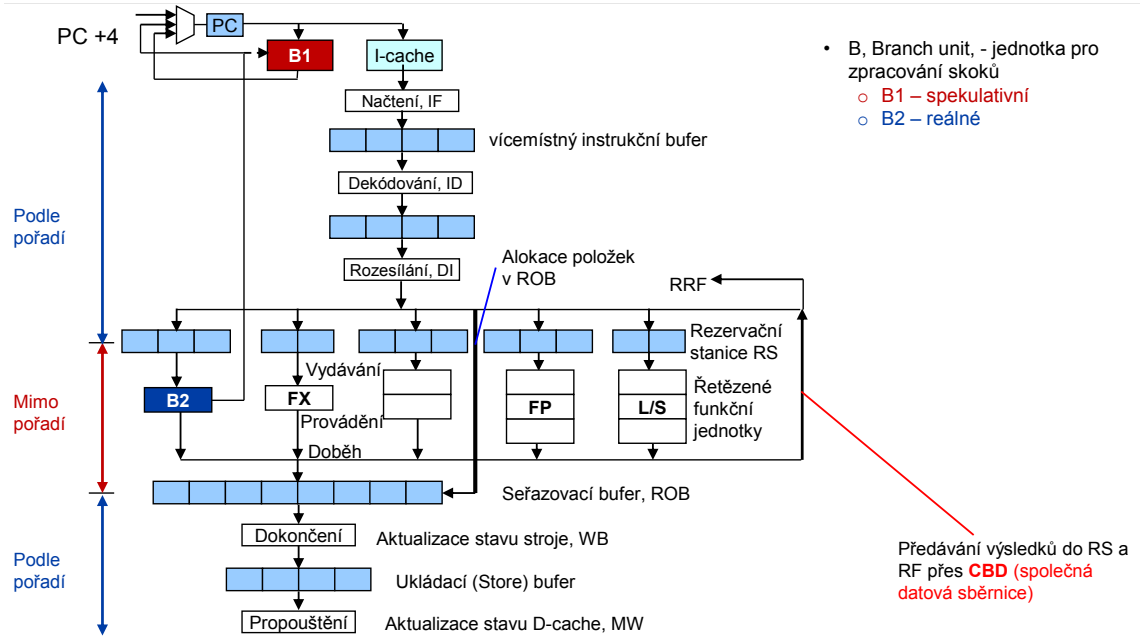
4. Nahrání dat ze společné sběrnice.
   (a) Rezervační stanice monitoruje společnou sběrnici. Pokud se nějaký tag shoduje s tagem zdroje v rezervační stanici, který má valid bit na 0, tak si hodnotu vezme valid bit je nastaven na 1.
   (b) Registrové pole taktéž monitoruje společnou sbernici. Hledá stejnou dvojici registr id, tag, když najde tak si vezme výsledek. (bere až poslední přejmenování)




Obrázek 1.19: Příklad algoritmu Rezervační stanice.

## 1.9 Predikce skoků

- Pro superskalární procesor čekat 1 takt při skoku je špatné, jsou třeba prediktory skoku a čekání kompletně eliminovat.
- Skoky jsou vysoce předvídatelné, můžeme je predikovat.
- Prediktor se nachází hned ve fázi IF.
  - Pamatuje si, kam se v minulosti skákalo (má uložené adresy).
  - Pokud ji najde, ví že se jedná o skok
- Pokud narazíme na skokovou instrukci, tak spekulujeme, ale nikdy si nejsme jistí. K instrukci je přidám příznak (spekulativní bit), který je později v případě korektní spekulace odstraněn, případně ponechám a příznaky se použijí k odstranění těchto instrukcí z ROB (*re-order buffer*) a RS (rezervační stanice).



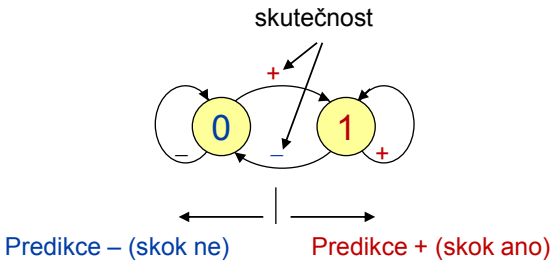

Obrázek 1.20: Umístění prediktoru skoků.

### 1.9.1 Predikce podmínky skoku

- Predikujeme, jestli se bude skákat, nebo nikoliv (týká se pouze podmíněných skoků).

#### 1-bitový prediktor

- Tabulka BHT (*branch history table*), která obsahuje adresu instrukce a u toho informaci, jestli se z ní posledně skákalo (1 bit).
  - 0 – spekuluju, že se nebude skákat.
  - 1 – spekuluju, že se bude skákat.
- Hned po IF můžu porovnavat.
- Pokud narazím na novou instrukci skoku, zjistim to až v ID, přidám ji do tabulky a po fázi EX k ní vložím / aktualizuji záznam, jestli jsem skočil.



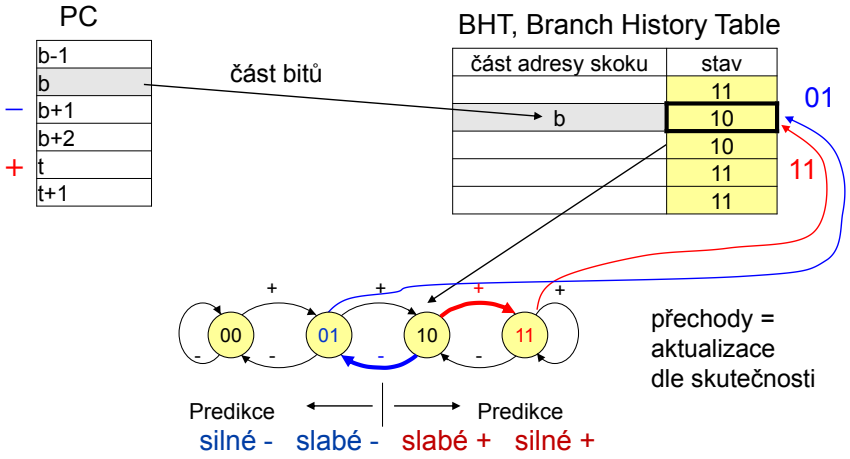

Obrázek 1.21: 1-bitový prediktor.

#### 2-bitový prediktor

- Stejný princip, pouze máme 2 bity.
- Vyšší bit říká, jestli máme skočit:

  - 00, 01 – spekuluju, že se nebude skákat.
  - 10, 11 – spekuluju, že se bude skákat.

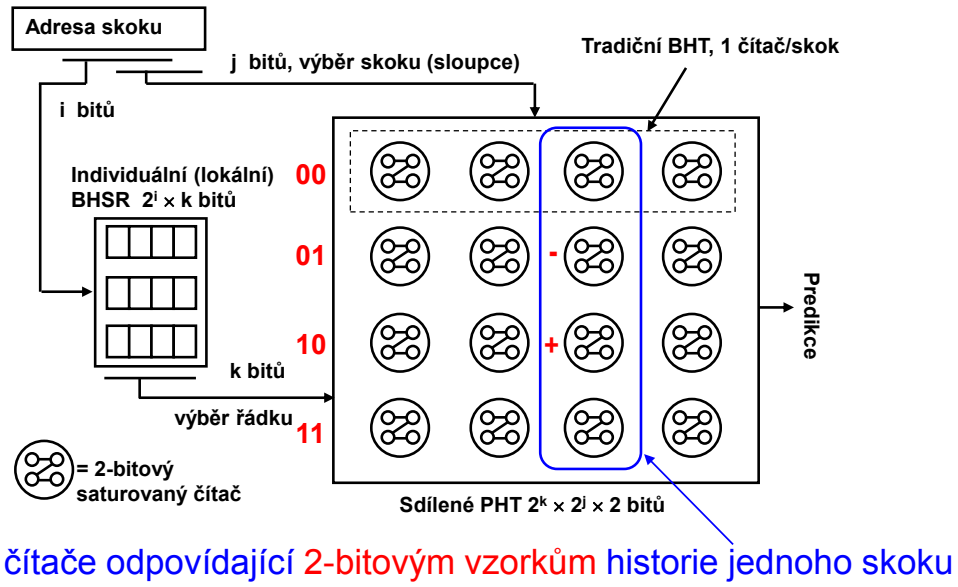
- Je schopen velice dobře rozeznat vnořené smyčky




Obrázek 1.22: 2-bitový prediktor. Příklad: adresu b jsem našel v tabulce, tj. je to skoková instrukce. Je ve stavu 10, tedy budu spekuluji, že budu skákat. Později se dozvím, že jsem skočil správně, přejdu do stavu 11 a vše je v pořádku. Nebo se dozvím, že jsem se spletl a skákat jsem neměl, přejdu do stavu 01 a je třeba větev výpočtu zahodit.

**Adaptivní prediktor s lokálními BHSR (*branch history shift register*)**

- Stejný jako 2-bitový prediktor, pouze mám k dispozici několik dvoubitových prediktorů a posuvný registr.
- Posuvný registr mi říká, který prediktor použít, na základě historie skoků, které si pamatuje.
- Je se schopný naučit nějaké vzorce chování, čím větší posuvný registr, tím složitější patterny.
- Konkrétněji:
  - Máme 4 čítače, každá má jiný výchozí stav
  - Všechny se aktualizují vždy
  - Podle historie vybírám ten, který se mi nejvíc hodí




Obrázek 1.23: Adaptivní prediktor s lokálními BHSR.

### 1.9.2 Predikce cílovej adresy

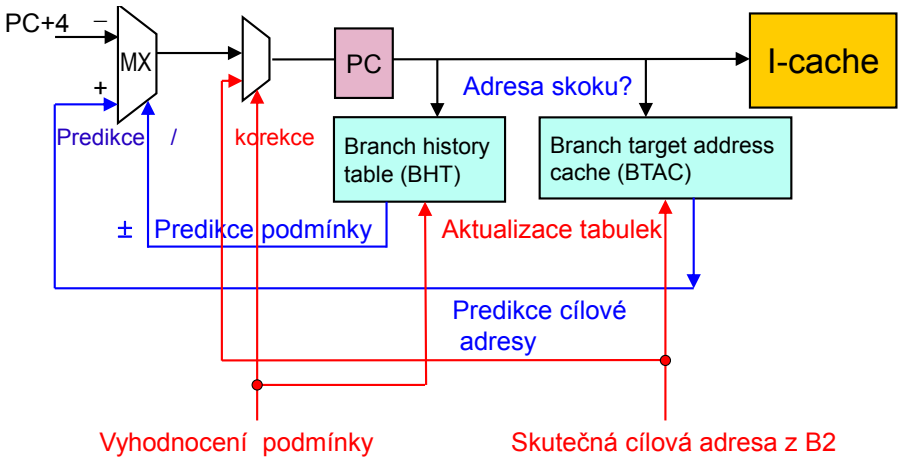
- Predikujeme, kam se bude skákat.
- Tabulka kde je uložené na jaké adresy se skákalo.
- Intuice:
  - Když prediktor podmínky skokové instrukce (BHT) říká „bude se skákat“:
    - a prediktor cílové adresy zná adresu – skočí se,
    - a prediktor cílové adresy nezná adresu – neskočí se.
  - Když prediktor podmínky skokové instrukce (BHT) říká „nebude se skákat“, tak se skákat nebude ať má předikto cílové adresy cokoliv. Prediktor skokové instrukce má vyšší prioritu.

#### Branch Target Address Cache (BTAC)

- Tabulka, která obsahuje adresu instrukce a informaci jestli se skočilo a kam se skočilo.

#### Return Stack Buffer

- Zásobník návratových adres.
- Pro instrukce return.




Obrázek 1.24: Predikce podmínky a cílové adresy.

# Kapitola 2

# AVS – Paměťová konzistence a předbíhání operací čtení a zápisu, podpora virtuálního adresového prostoru.

## 2.1 Zdroje

- `AVS-03.pdf`
- `AVS-04.pdf`
- `AVS_2019-10-07.mp4`
- `AVS_2019-10-14.mp4`

## 2.2 Paměti cache (rychlé vyrovnávací paměti)

- K čemu je?
  - Snížení objemu komunikace s hlavní pamětí.
  - Snížení průměrné doby přístupu.
- Jelikož malé paměti jsou rychlejší a dražší, velké paměti pomalejší a levnější, je optimální paměťový systém hierarchický – Úrovně cache: L1C, L2C, L3C.
- Datová cache a instrukční cache.
- Metriky:
  - Cache miss – data v cache nejsou.

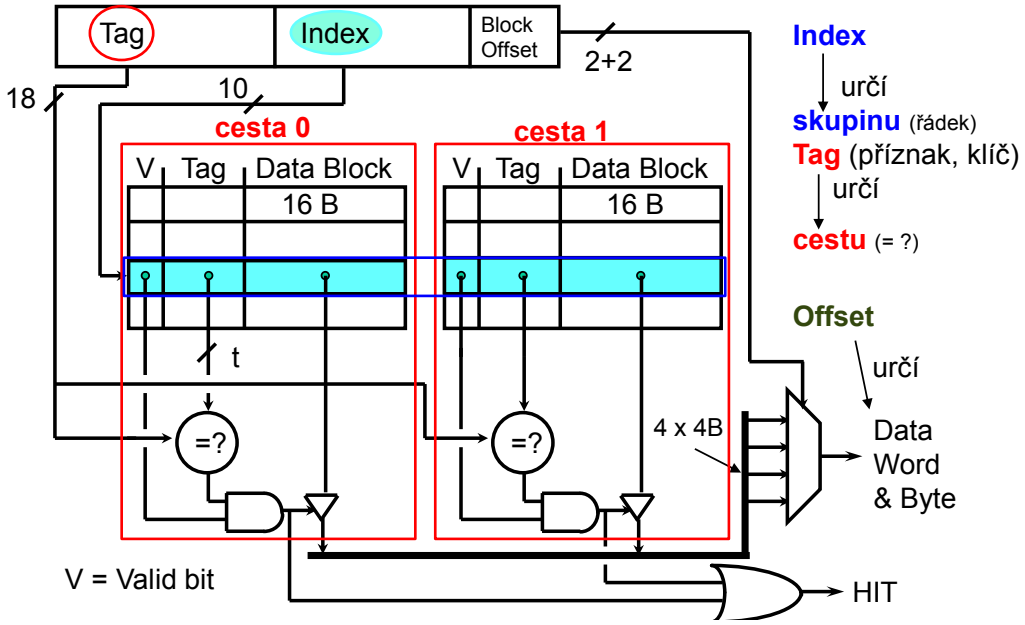
$$\text{miss rate} = \frac{\text{cache miss}}{\text{celkový počet přístupů}}$$

  - Cache hit – data v cache jsou (cache hit / celkový počet = hit rate).

$$\text{hit rate} = \frac{\text{cache hit}}{\text{celkový počet přístupů}}$$

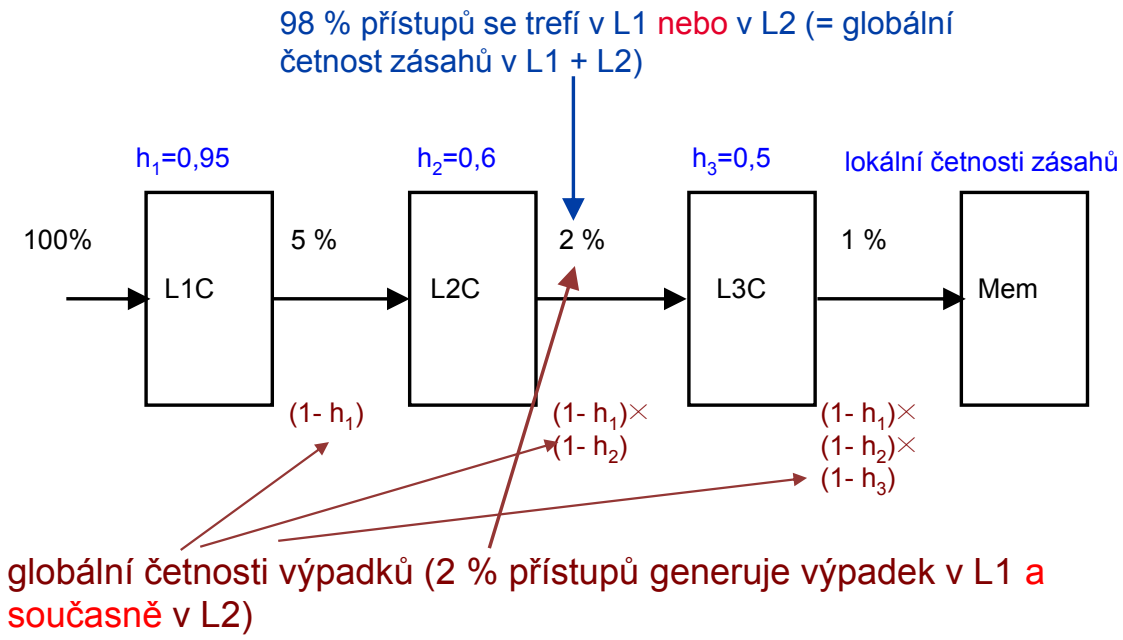
- Velikost jednoho bloku cache je typicky 64 B (celková velikost je např. 16 KB).

  - Z paměti do cache vždy bereme celý blok.
- Platí princip časové a prostorové lokality.
- Vzhledem ke klesající velikosti paměti se vzrůstající úrovní hierarchie musí existovat mapování:
  - přímé mapování;
  - plně asociativní;
  - skupinově asociativní (nejpoužívanější).




Obrázek 2.1: Skupinově asociativní cache.

- Strategie výběru bloku pro přemístění z cache do paměti:
  - LRU (*last recently used*) – blok nejdéle nepoužitý, zaznamenává se počet přístupů;
  - FIFO – nejstarší blok;
  - náhodný.




Obrázek 2.2: Lokální a globální četnost výpadků a zásahů.

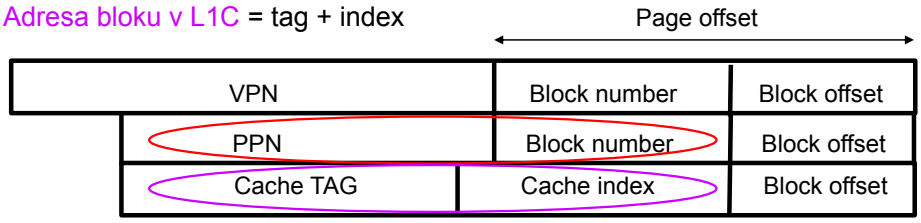
## 2.3 Podpora virtuální paměti

- Každý proces „vidí“ pamět’ po svým (od `000`... až po `fff`...). – Procesy jsou od sebe „odstíněny“.
- Pamět dělíme na tzv. stránky (*page*).
  - Velikost stránky je typicky 4 KB (lze změnit).
  - Mapování mezi fyzickým a virtuálním adresovým prostorem probíhá po stránkách.
- Máme dva adresované prostory:
  - Virtuální adresa (VA) = adresa virtuální stránky (VPN, *virtual page number*) + číslo bloku na stránce (*page offset*).
  - Fyzická adresa (PA) = (PPN, *physical page number*) + číslo bloku na stránce (*page offset*).
- Page offset – číslo bloku na stránce.
- Block offset – které slovo/byte.
- MMU (*memory management unit*) – Jednotka správy paměti v procesoru, která provádí překlad adres.
- Překlad:
  - Překládání na úrovni stránek.
  - Více-úrovňová tabulka stránek.

**Pg.offset** = Block number (číslo bloku na stránce) + **block offset** (které slovo, bajt)

Adresa bloku v paměti = PPN + Block number

Adresa bloku v L1C = tag + index




Velikost stránky: 4, 8 kB, ale i 64 kB, 2MB, 4MB

Cache index: 16kB, 32kB, 512kB, 16MB

Obrázek 2.3: Podpora virtuální paměti.

### 2.3.1 Lineární tabulka stránek (PT, *page table*)

- Byla by příliš velká, takhle to udělat nejde.

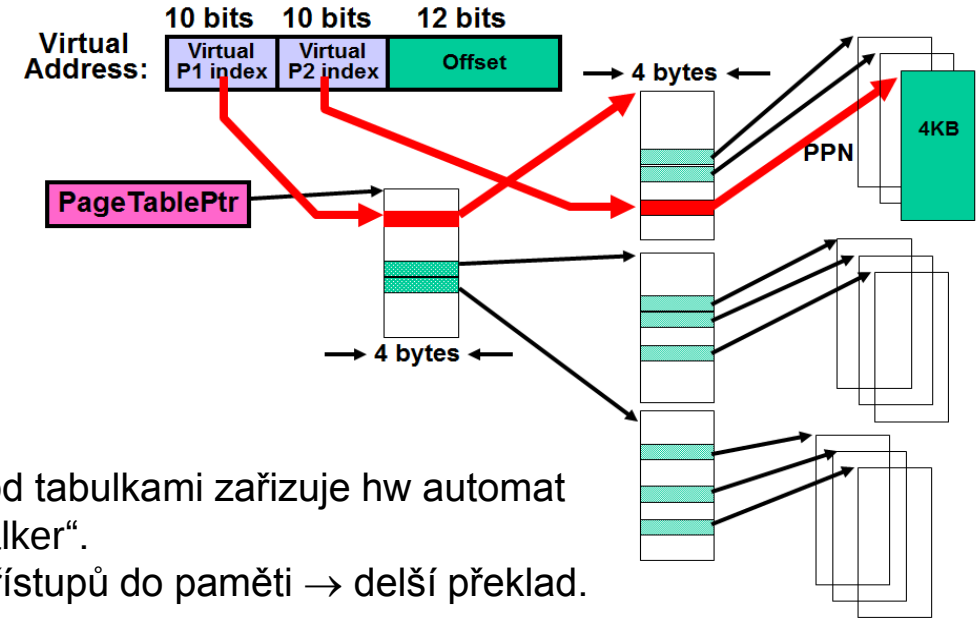
- Položka PT (PTE, Page Table Entry) mapuje číslo virtuální stránky na číslo fyzické stránky, VPN → PPN; Vyhledání je snadné!
- **PT je v paměti.**
- Pro 32-bitové adresy, stránky 4-KB a PTEs 4-byte:
  - $2^{32}/2^{12} = 2^{20}$ PTEs, tj. **tabulka stránek 4 MB** na 1 proces
  - až $2^{32}$ = 4 GB dat v celém virtuálním prostoru na 1 proces
- **Větší stránky?**
  - Vnitřní fragmentace (celá stránka se neužije) ☹
  - Větší pokuta při výpadku stránky (delší čas čtení z disku) ☹
  - Méně překladu při zpracování velkých dat (matice) ☺
- **A co teprve 64-bit virtuální adresový prostor???**
  - Dokonce při velikosti stránek 1MB bychom potřebovali $2^{64}/2^{20} = 2^{44}$ PTEs 8-byte ($2^7$ TB!)

- **Naštěstí je obsazení virtuálního adresového prostoru řídké**

Obrázek 2.4: Velikost lineární tabulky stránek.

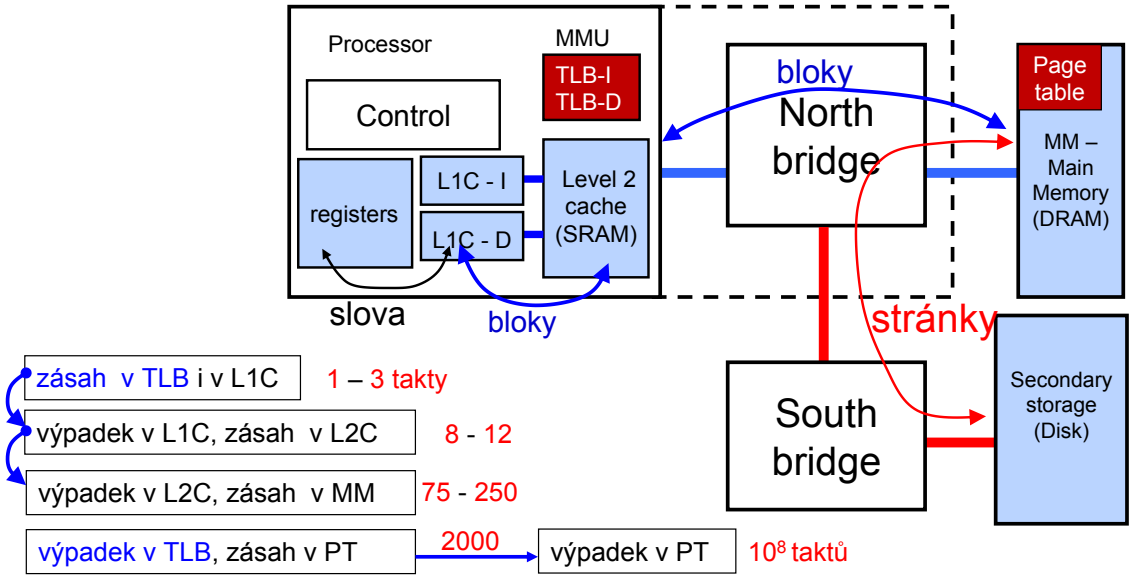
### 2.3.2 Více-úrovňová tabulka stránek

- Hierarchická struktura (n-ární strom).
- Výhody:
  - ušetříme paměť;
  - ušetříme rychlost vyhledávání.
- Nevýhoda:
  - Tabulky stránek jsou uloženy v paměti, přístup do paměti je drahý. Řešíme pomocí hardwaru, tzv. TLB (*translation lookaside buffer*).

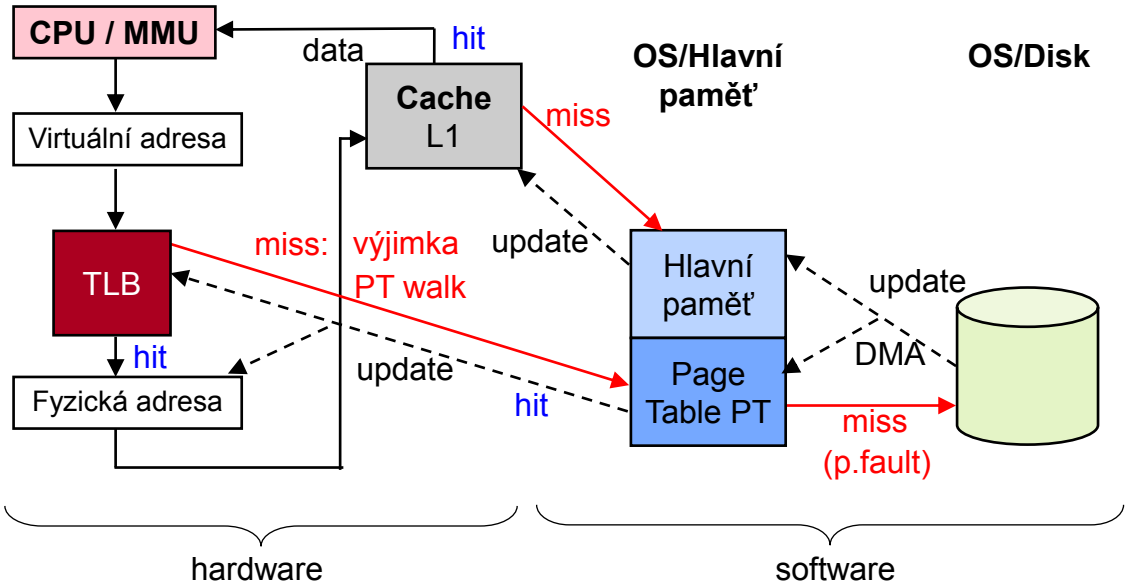



Průchod tabulkami zařizuje hw automat „PT walker“.
Více přístupů do paměti → delší překlad.

Obrázek 2.5: Více-úrovňová tabulka stránek.



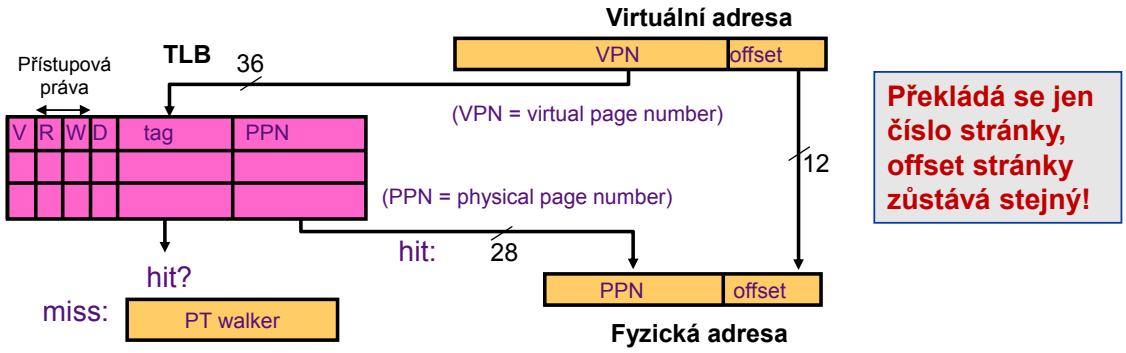

Obrázek 2.6: Paměti cache v systému virtuální paměti.




Obrázek 2.7: Schéma přístupu do paměti s L1C.

### 2.3.3 TLB (*translation lookaside buffer*)

- Cache pro překlad stránek, která je umístěna na procesoru a obsahuje sadu aktuálně používaných dvojic (VPN, PPN).
- Plně asociativní.




Obrázek 2.8: TLB.

### 2.3.4 Virtuální paměť a cache

- Spolupráce s CPU a L1C vyžaduje co nejrychlejší přístup do cache.
- Jak dospět od virtualní adresy k adrese bloku (index, tag) v L1C?
  - virtuální adresa je k dispozici ihned;
  - fyzická adresa je k dispozici až po překladu VA;
- Která adresa by se mohla použít pro přístup? Tři možnosti:
  - P/P cache: fyzický index, fyzický tag
  - V/V cache: virtutální index, virtutální tag

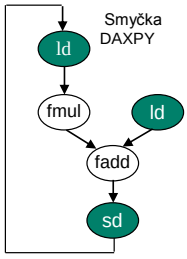
- V/P cache: virtutální index, fyzický tag
- P/V cache: fyzický index, virtuální tag – nepoužívá se, index by se musel získat překladem a čas se neuspoří.

## 2.4 Optimalizace toku dat přes paměť (předbíhání)

- Stupně L/S jednotky:
  - generování (výpočet) adresy;
  - překlad z virtuální na fyzickou adresu;
  - přístup do cache (adresace).
- Mezi instrukcemi Load a Store se stejnou adresou existují datové závislosti, podobně jako u registrů.
  - RAW, WAR, WAW.
  - Není je možné odhalit dříve než jsou spočteny adresy paměťových operandů.
  - Závislosti musí být respektovány, aby se zachovala sémantika programu, jinak vzniká paměťová nekonzistence.
- Instrukce Load a Store je možné vykonávat podle pořadí v programu, což může být pomalé. A nebo za jistách okolností můžou být vykonávány mimo pořadí:
  - RPR (read can pass read);
  - RPW (read can pass write);
  - WPR (write can pass read);
  - WPW (write can pass write) – Když jdou na jiné adresy.

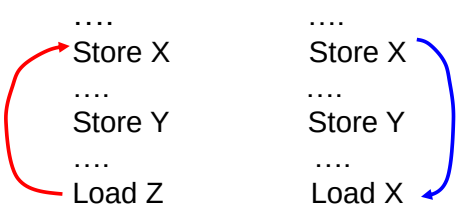
### 2.4.1 Read can pass write (RPW)

- Tzv. bypassing (přeskočení dopředu) a forwarding (zkratka).




- OOO procesor v podstatě provádí HW rozbalení smyčky.
- Nové ld může předběhnout aktuální sd, iterace smyčky se mohou částečně překrýt!
- RPW je hlavním zdrojem lepší výkonnosti, načítání bývá totiž na začátku těla smyček se závislými instrukcemi 2 způsoby:

Load z adresy ***Z*** předběhne Store s jinou adresou ***X***, které ještě nezačalo (*bypassing*)




Load načte data z ještě nedokončené instrukce Store se stejnou adresou ***X*** (load from store *forwarding*, předávání).

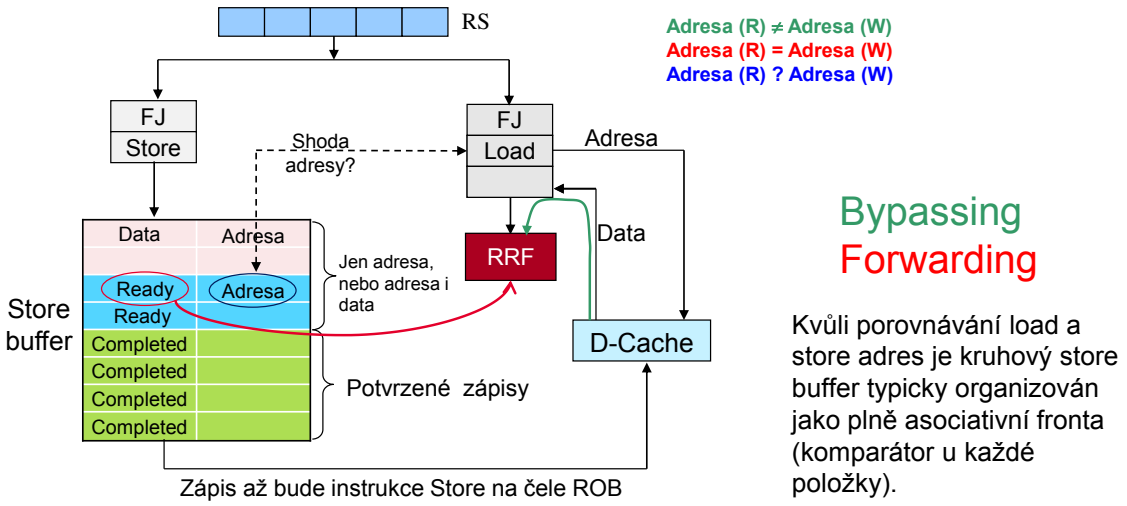
Obrázek 2.9: RPW, příklad smyčka DAXPY.

- Potřebujeme techniku dynamického rozlišování adres: adresa(R) $\neq$ adresa(W)?

- neshoda – není konflikt RAW, R nečeká na W a předbíhá;
- shoda – R načte data do dst reg z čekajícího nedokončeného zápisu (store bufferu);
- neví - čeká nebo spekuluje (předpokládá, že se liší).

### 2.4.2 Store buffer

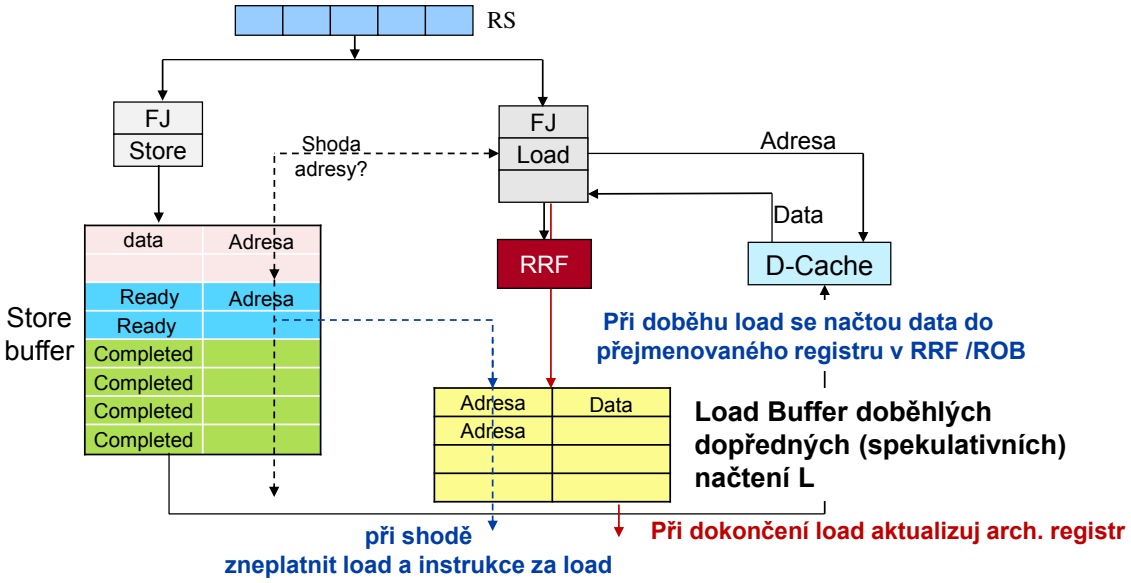
- Položka je ve store bufferu alokována v době dekódování (DI). Pokud je store buffer plný, musí se čekat.
- Adresy instrukcí Store jsou ve bufferu v programovém pořadí.
- Nové adresy Load se kontrolují s čekajícími adresami Store na shodu/neshodu.
- Když je instrukce Store na čele ROB i store buferu, dojde k propuštění Store, tj. k zápisu do Dcache. Položka přejde do stavu Available.




Obrázek 2.10: Organizace jednotky L/S podporující RPW.

### 2.4.3 Load buffer

- Tabulka pro spekulativní načítání.
- Pokud některé adresy Store nejsou známy, je třeba spekulovat, že se budou lišit od adresy Load.
- Pro ověření spekulace jsou adresy doběhlých spekulativních načítání uloženy do nového L-bufferu.
- Každá potvrzená instrukce Store pak musí ověřit, že nemá adresu shodnou s nějakou položkou L-bufferu, tj. s nějakým spekulativním čtením, které Store předběhlo.
  - Při shodě je potřeba postiženou instrukci Load a další za ní následující instrukce zrušit a opakovat.
  - Pokud k žádné shodě nedošlo až do dokončení Store, je proveden přepis Load dat z RRF do ARF.




Obrázek 2.11: Out-of-order Load / Store jednotka.

### 2.4.4 Relaxovaná paměťová konzistence

- Sekvenční konsistence, která zachovává pořadí přístupů do paměti, není v současnosti zajímavá. Je totiž překážkou modernímu hardware a optimalizujícím kompilátorům.
- Předbíhání RPW je jen jedna možnost uvolnění pořadí přístupů do paměti, i když pro výkonnost nejvýznamnější.
- Existuje však řada ještě volnějších modelů se zcela volným pořadím čtení a zápisů.
- Moderní procesory vykazují relaxovanou (uvolněnou) paměťovou konsistenci, kdy se čtení a zápisy mohou vzájemně předbíhat, pokud to nemění správnost programu.
- Relaxovaná paměťová konzistence:
  - Lepší výkonnost a jednodušší HW implementace.
  - Je ponecháno na programátorovi, aby identifikoval a označil spec. instrukcemi (např. paměťovými bariérami) ty instrukce L/S, které musí být uspořádány.
  - Všechny ostatní instrukce se mohou provádět mimo pořadí.
- Na vyšší úrovni musí programátor použít synchronizační příkazy pro vymezení oblastí předepsaného pořadí L/S:
  - direktivou flush v OpenM;
  - proměnnými volatile.
- Nevýhodou relaxovaných modelů je břímě navíc pro programátora, možnost záludných chyb.

# Kapitola 3

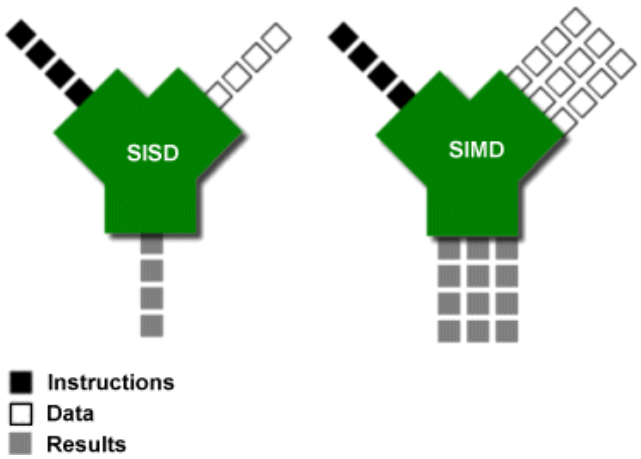
# AVS – Datový paralelismus SIMD, HW implementace a SW podpora.

## 3.1 Zdroje

- `AVS-05.pdf`
- `AVS_2019-10-21.mp4`

## 3.2 Paralelismus v procesorech

- Flynnova taxonomie:
  - SISD (*single instruction stream, single data stream* – Skalární procesor.
  - SIMD (*single instruction stream, multiple data streams* – Vektorové procesory.
  - MISD (*multiple instruction streams, single data stream* – Používá se pouze ve velmi speciálních případech.
  - MIMD (*multiple instruction streams, multiple data streams* – Vícevláknové procesory.




Obrázek 3.1: SISD vs SIMD.

- Instrukční paralelismus:

- Superskalární procesory – instrukční paralelismus řízen pomocí hardware ( OOO, pipeline).
- Procesory s dlouhým instrukčním slovem VLIW – instrukční paralelismus řízen software (kompilátor) nebo hrubé granularitě (vlákna - TLP, procesy).

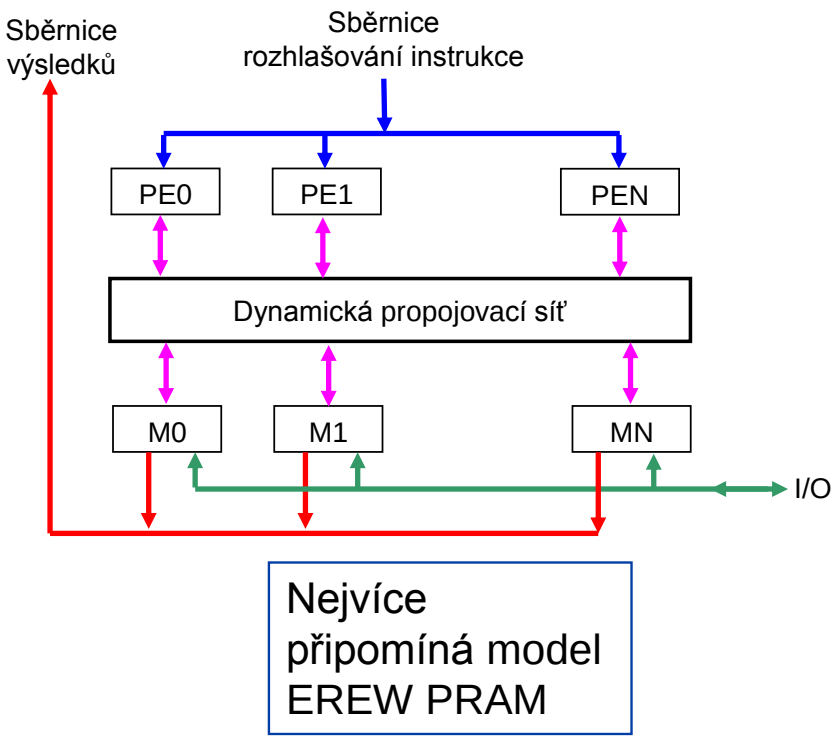
- Doposud jsme se zabývali instrukčním paralelismem, tedy jak provádět více instrukcí zároveň. Nyní, máme jednu instrukci a ta se aplikuje na více dat (SIMD) ⇒ zjednodušení řízení procesoru!

## 3.3 Datový paralelismus v procesorech (SIMD)

- SIMD architektury obsahují velký počet výpočetních jednotek (PE, processing units) a řídící procesor.
  - Soubor procesorů řízený centrální jednotkou pracuje synchronně po instrukcích, všechny procesory dělají totéž. Některé procesory mohou stát (NOP). Je nezbytné změnit způsob uvažování, při mapování algoritmů na tyto architektury. V současnosti
  - Implementováno pomocí AVX/SSE registrů.
- Nejsou považovány za univerzální výpočetní systémy; pro některé testy špičková výkonnost, pro jiné jen nízká.
  - Používají se jako HW akcelerátory a koprocesory (GPU, AVX).
  - Silná stránka: práce s vektory, maticemi (zvuk, video, diferenciální rovnice, neuronové sítě, . . . )
  - Slabá stránka: podmíněné příkazy, switch.
- Výpočet jednotlivých prvků musí probíhat nezávisle na ostatních.
- Kdy má smysl přemýšlet nad SIMD (vektorizací)
  - Propustnost paměti – stíhám procesoru předávat data?
  - Výkon – mám dost výkonu abych spočítal data, která mám v procesoru?

### 3.3.1 Architektury se sdílenou pamětí

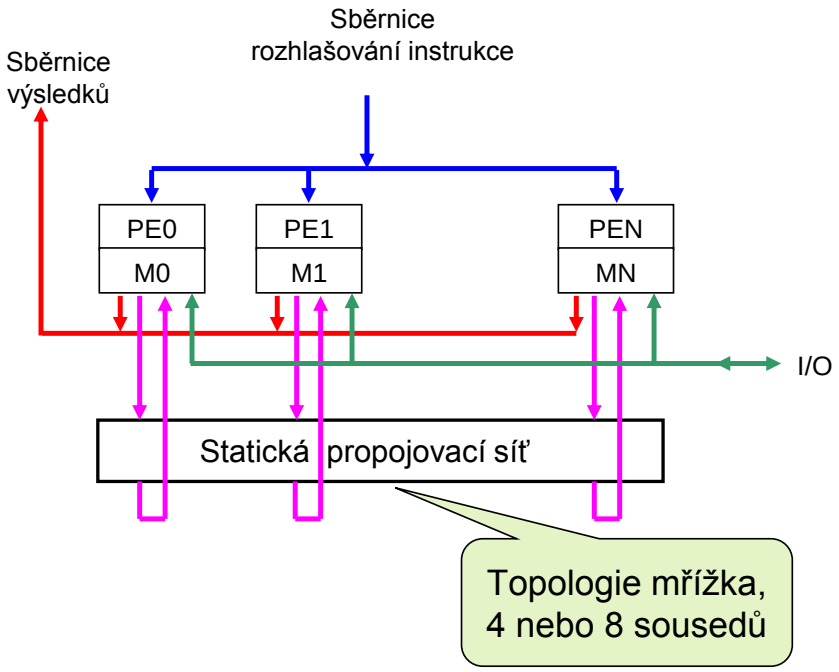
- Paměťové moduly sdíleny všemi PE elementy.
- Čtení i zápis dat probíhá skrze dynamickou propojovací síť (např. křížový přepínač).
- Jednotlivé PE elementy komunikují pouze přes sdílenou paměť.
- Současné architektury GPU se nejvíce podobají tomuto modelu.




Obrázek 3.2: SIMD se sdílenou pamětí.

### 3.3.2 Architektury s distribuovanou pamětí

- Každý PE má svůj vlastní paměťový modul – většinu výpočtu provádí nad lokálními daty.
- Komunikace resp. výměna dat s ostatními PE elementy probíhá skrze statickou propojovací síť (např. mřížka, torus, ...).




Obrázek 3.3: SIMD se distribuovanou pamětí.

## 3.4 HW implementace

- Typicky SWAR (SIMD within a register).
- 16 nebo 32 tzv. vektorových registrů o velikosti typicky 512 bitů.
- Každý registr je rozdělen do několika nezávislých částí.
- Daná instrukce je vykonána na každé části registru v jednom kroku.
- Nejznámější SWAR rozšíření dnešních CPU: AVX, SSE.

## 3.5 SW podpora, vektorizace

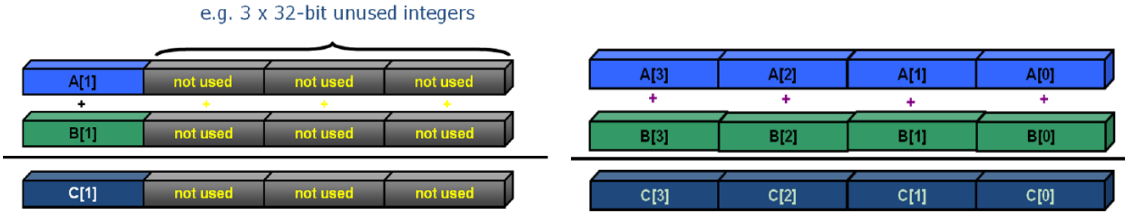
- Jak vektorizovat?
  - vektorizované knihovny (Intel MKL, Atlas, Numpy, Scipy);
  - automatické vektorizace kompilátorem (`-o3`, `-vec`);
  - pragma hints pro kompilátor do zdrojového kódu (OpenMP);
  - vektorové intrinsic funkce.

Bez vektorizace

```
for (i = 0; i < MAX; i++)
{
  c[i] = a[i] + b[i];
}
```

Vektorizovováno

```
for (i = 0; i < MAX; i+=4)
{
  c[i:4] = a[i:4] + b[i:4];
}
```




Obrázek 3.4: Příklad vektorizace. Načítám naráz 4 prvky, počítám naráz 4 prvky, ukládám naráz 4 prvky.

- Co dokáže kompilátor auto vektorizovat?
  - Cyklus kde známe počet iterací už v době, kdy do něho vstoupíme (v těle cyklu se nemůže měnit podmínka).
  - Cyklus který má jeden vstup a jeden výstup (žádný breaky a continue).
  - Cyklus který nemá podmínky a volání funkcí.
  - V případě vnořených cyklů lze vektorizovat pouze ten nejvnitřnější.

- **Počitatelné smyčky**

```
typedef struct{ float* data; size_t size;} vec_t;

void vec_elwise_product(vec_t* a, vec_t* b, const vec_t* c)
{
  for (auto i = 0; i < a->size; i++}
    c->data[i] = a->data[i] * b->data[i];
}
```

- **Smyčky s jedním vstupem a výstupem**

```
while (i < 100) {
   a[i] = b[i] * c[i];
   if (a[i] < 0.0) break; // data-dependent exit condition:
   i++;
} // loop not vectorized
```

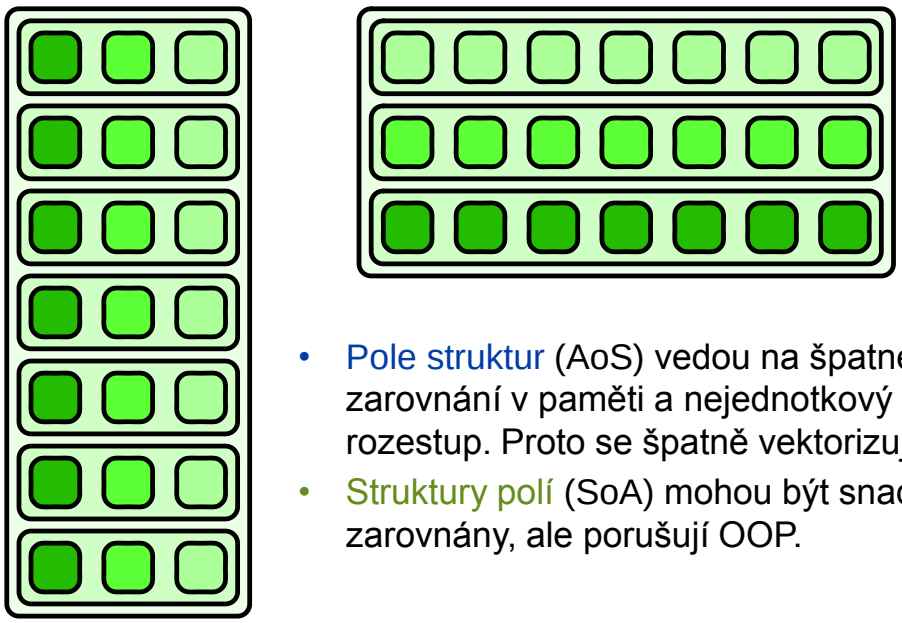
Obrázek 3.5: Příklad toho, co dokáže kompilátor auto vektorizovat.

- **Přímý kód** (žádné **switch**, **if** pouze s maskováním)

```
for (int i = 0; i < length; i++) {
   float s = b[i] * b[i] - 4 * a[i] * c[i];
   if (s >= 0) x[i] = sqrt(s);
   else        x[i] = 0.;
} // loop vectorized (because of masking)
```

Obrázek 3.6: Příklad toho, co kompilátor nedokáže auto vektorizovat.

- Na čem může selhat?
  - Nejednotkový rozestup: $b[i] + a[i + 5]$.
  - Nezarovnané datové struktury.
  - Datové závislosti mezi iteracemi – OpenMP nekontroluje, programátor musí hlídat.
  - Pointer aliasing (překryté paměťové prostory).
  - Data musí být zarovnaná na velikosti SIMD registru (16, 32, 64 bytů pro SSE, AVX, AVX-512).
- Datové struktury jsou problém.
  - Pole struktur vs struktura polí, co je „hezké“ z hlediska OOP, může být z hlediska vektorizace problém.



- Pole struktur (AoS) vedou na špatné zarovnání v paměti a nejednotkový rozestup. Proto se špatně vektorizují.
- Struktury polí (SoA) mohou být snadno zarovnány, ale porušují OOP.

Obrázek 3.7: Datové struktury – AoS nebo SoA.

## 3.6 Vektorizace pomocí OpenMP

```
void add(float* a, float* b, float* c, float* d, float* e, int n)
{
    #pragma omp simd
    for (int i = 0; i < n; i++) {
        a[i] = a[i] + b[i] + c[i] + d[i] + e[i];
    }
}
```

Výpis 3.1: Obyčejná smyčka.

```
#pragma omp simd reduction(+:sum)
for (i = 0; i < *p; i++) {
    A[i] = B[i] * C[i];
    sum = sum + A[i];
}
```

Výpis 3.2: Redukce, po skončení vektorovýho výpočtu posčítá mezivýsledky.

```
#pragma omp simd aligned(sum:64)
```

Výpis 3.3: Aligned, dané proměnné jsou zarovnány na počet bytů.

```
#pragma omp simd safelen (10)
for (i = 0; i < MAX; i++) {
    a[i] += a[i - 10];
}
```

Výpis 3.4: Safelen, maximální počet iterací, které se mohou vykonávat současně bez porušení závislostí.

```
#pragma omp simd linear (x)
for (i = 0; i < MAX; i++) {
    x[i] = x_orig + i * linear_step;
}
```

Výpis 3.5: Linear, hodnota proměnné je ve vztahu k číslu iterace.

# Kapitola 4

# AVS – Architektury se sdílenou pamětí UMA a NUMA, zajištění lokality dat.

## 4.1 Zdroje

- `AVS-11.pdf`
- `AVS-12.pdf`
- `AVS_2019-12-02.mp4`
- `AVS_2019-12-09.mp4`
- *Otázka je propletená s otázkou AVS 5.*

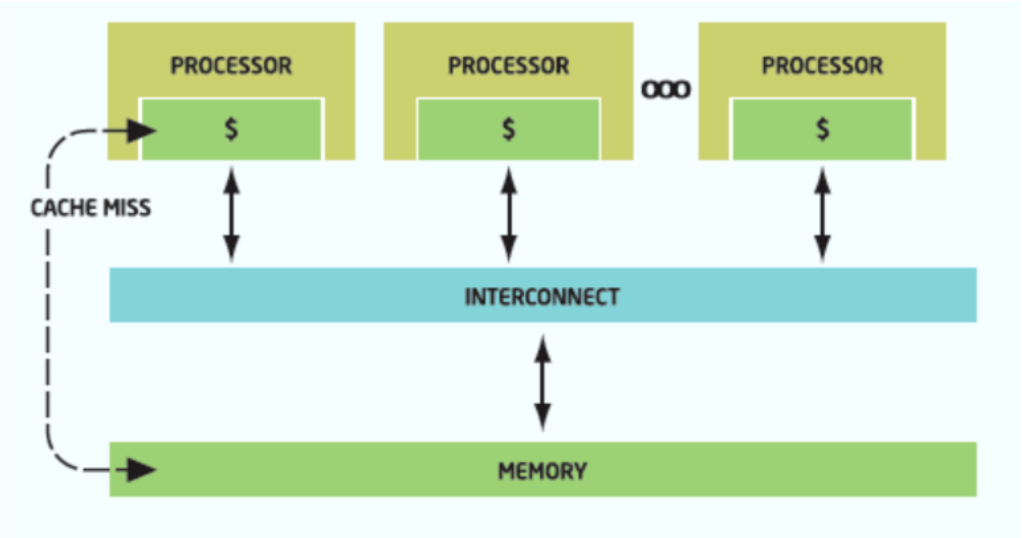
## 4.2 Architektury se sdíleným adresovým prostorem

- Jaký potřebujeme hardware aby nám fungovala správně sdílená paměť, která je potřebná pro paralelizaci.
- Sdílený adresový prostor (SAS, *shared address space*).
- Sekvenční počítač:
  - Vidí vlastní čtení a zápisy v pořadí, jak je vydává.
  - Při čtení vidí poslední hodnotu zapsanou na danou adresu.
  - U paralelního počítače ale různé CPU nemusí vidět paměťové akce ve stejném pořadí.
- U paralelního počítače musíme zajistit:
  - **Šíření zápisů** – každý zápis se stane viditelný všem CPU, tj. všem jádrům.
  - **Serializaci zápisů** – všechny CPU vidí zápisy na adresu x v témže pořadí (u sběrnice splněno triviálně).
- Jakmile do sdílené paměti může přistupovat více vláken, je nutné zajistit, aby na jedné adrese našla při čtení vždy stejná data, ať jsou data v kterékoliv úrovni cache nebo v hlavní paměti (tj. koherentní kopie).
  - **Koherence** se týká jednoznačnosti zápisů/čtení na 1 adresu.
  - **Konzistence** se týká pořadí přístupů na různé adresy. Oproti koherenci specifikuje v jakém pořadí jednotlivé procesy spouštějí své paměťové operace, či jak se toto pořadí jeví ostatním procesům.

- Proto je kritická:
  - Aktualizace (zápis) dat na jednu adresu 1 vláknem (správný postup je dán **protokolem koherence cache**).
  - Aktualizace dat na různých adresách více vlákny (pořadí je určeno **modelem paměťové konzistence**).

### 4.2.1 UMA (*uniform memory access*)

- Uniformní doba přístupu do hlavní paměti.
  - Každý procesor/jádro platí stejnou pokutu za přístup do libovolné části paměti.
- Centrální sdílená paměť → UMA.
- Implementace:
  - Plné propojení – extrémně drahé.
  - Sběrnice – jedna sběrnice o kterou procesory soupeří, neefektivní.
  - Křížový přepínač (X-bar) – umožňuje paralelní přenosy.
  - Kruhové propojení – lepší než sběrnice, levnější než křížový přepínač.
  - Propojovací síť.

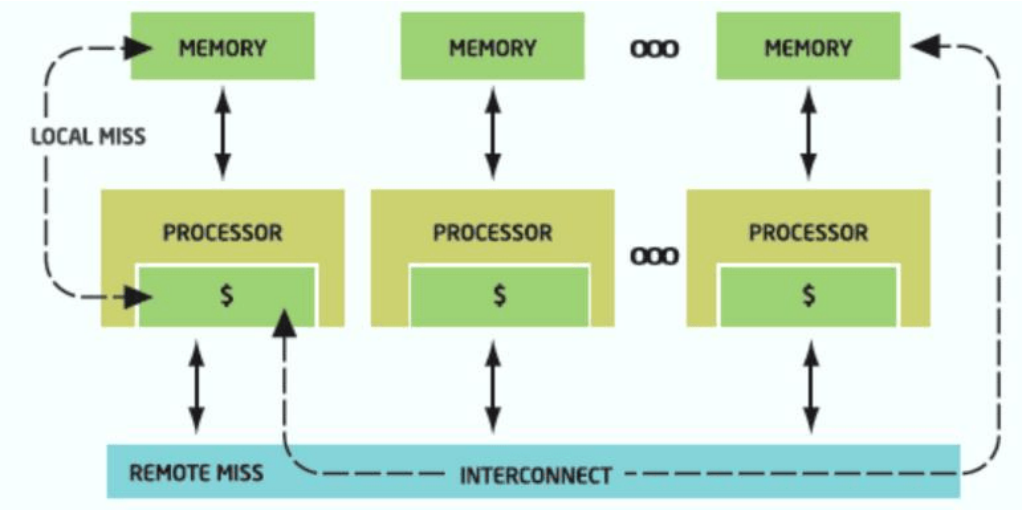



Obrázek 4.1: Architektura UMA.

### 4.2.2 NUMA (*non-uniform memory access*)

- Neuniformní doba přístupu do hlavní paměti (např. lokální a vzdálený výpadek v cache).
- Vzdálený přístup – automatické zaslání zprávy tam a zpět.
- Sdílená paměť fyzicky distribuovaná → NUMA.
- Implementace:
  - Propojovací síť, např. 2D-torus, tlustý strom, hyperkostka.
  - Někdy oddělená adresová a datová síť.

– Rozšiřitelnost až do 2048 jader a 64 terabajtů (TB).




Obrázek 4.2: Architektura NUMA.

## 4.3 Zajištění lokality dat

- **Prostorová lokalita**
  - Prostorová lokalita (*spatial locality*) se týká používání datových prvků v rámci relativně blízkých míst uložení.
  - Pokud procesor pracuje s daným datovým blokem, pravděpodobně bude pracovat i s následujícím datovým blokem. – Do cache se načte z paměti požadovaný blok i blok následující.
- **Časová lokalita**
  - Časová lokalita (*temporal locality*) označuje opětovné použití konkrétních dat v relativně krátkém časovém období.
  - Pokud procesor pracoval nedávno s daným datovým blokem, pravděpodobně s ním bude pracovat i v budoucnu. – Z cache odstraňujeme datové bloky, se kterými procesor dlouho nepracoval.
- **Jak zajistit**
  - Hlavní příčinou pomalého běhu aplikací v systému SAS je fyzická vzdálenost paměti od procesoru, který s ní pracuje.
  - Pro uspokojivý provoz aplikace je třeba klást zvláštní důraz na zajištění lokality dat zpracovávaných dat.
  - Případně omezení současného přepisování konkrétní stránky paměti v cache.
  - Příklad: místo toho, aby procesor přistupoval k rychlé systémové sběrnici, aby mohl do paměti RAM, musí jít do karty PCI a pak přes dráty do přepínače, odtud do sousedního uzlu, do paměti a zpět.

# Kapitola 5

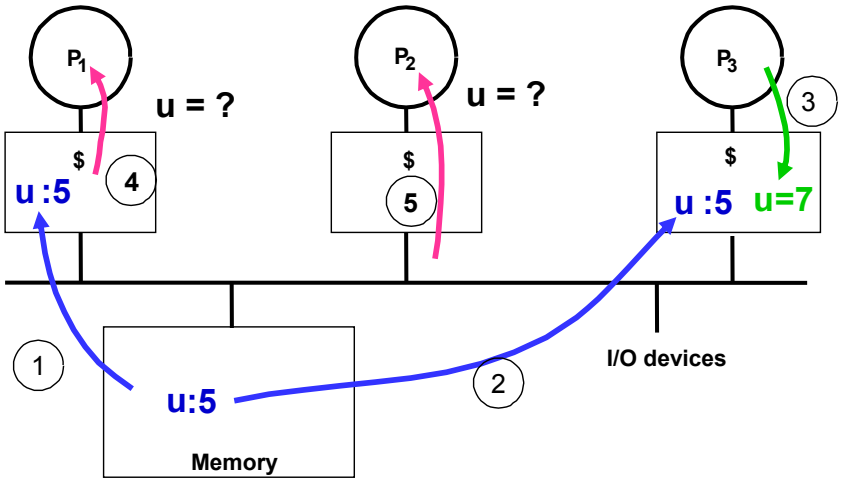
# AVS – Problém koherence pamětí cache na systémech se sdílenou pamětí, protokol MSI.

## 5.1 Zdroje

- `AVS-11.pdf`
- `AVS-12.pdf`
- `AVS_2019-12-02.mp4`
- `AVS_2019-12-09.mp4`
- *Otázka je propletená s otázkou AVS 4.*

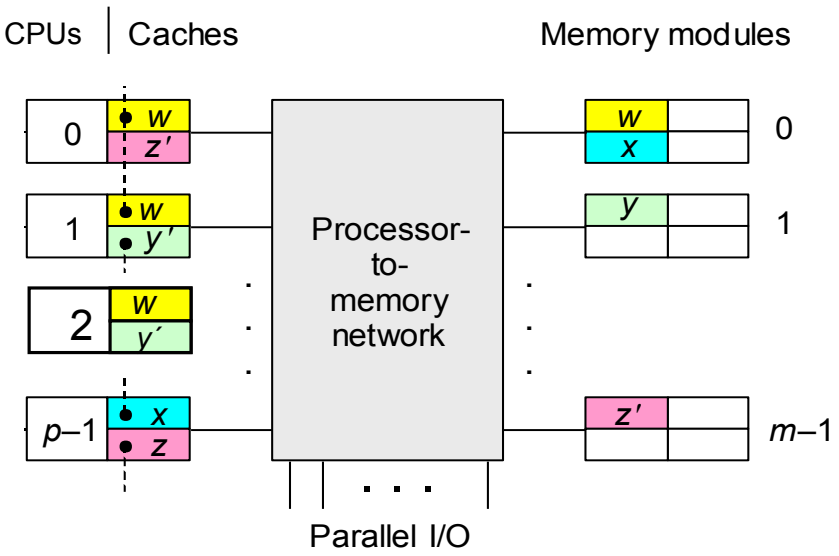
## 5.2 Koherence paměti cache (CC)

- Proč máme paměti cache? Kvůli rychlosti, přístup do paměti je příliš drahý.
- **Koherence pamětí cache** (CC) znamená, že pro danou adresu existuje jediná (koherentní) verze sdílených dat v jedné, několika nebo i ve všech pamětech cache. Na kopii v paměti nezáleží, ta může být zastaralá (neplatná).
  - Jedna adresa má stejnou hodnotu ve všech pamětích cache (nikoliv hlavní paměť)!
  - Potřebujeme mechanismus (protokol), který toto zajišťuje.
- GPU jsou nekoherentní, synchronizace mezi tolika jádry by byla příliš náročná.




Obrázek 5.1: Koherence paměti cache.

- Jaké mohou být stavy bloku paměti (*cache line*) v paměti cache?
  - **Čistý a špinavý**
    - Čistý – blok se zatím pouze četl, nebylo do něho zapsáno.
    - Špinavý – do bloku někdo zapsal.
  - **Jediný a sdílený**
    - Jediný – existuje jediná kopie bloku v pamětích cache.
    - Sdílený – existuje několik kopií bloku v pamětích cache.
  - **Platný a neplatný**
    - Platný – pokud ho chce procesor číst, cache vrátí cache hit (je tam aktuální hodnota).
    - Sdílený – pokud ho chce procesor číst, cache vrátí cache miss (je tam neaktuální hodnota).

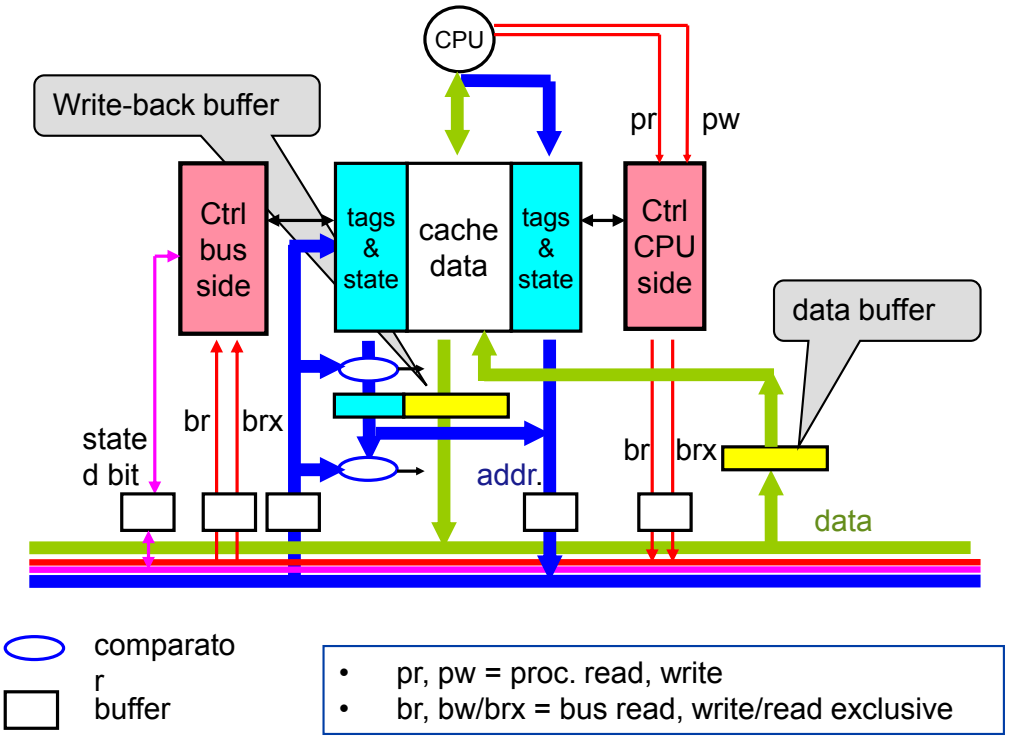



Obrázek 5.2: Každé jádro procesoru má svoji cache, memory modules je hlavní paměť. Paměťové bloky s čarou jsou modifikované. Stav bloků: $x$ je čistý, jediný a platný; $w$ je čistý, sdílený a platný; $y$ je špinavý, sdílený a platný; $z$ je špinavý, sdílený a neplatný (nekohorence).

- Pro zajištění koherence pamětí cache máme 2 strategie: protokoly CC založené na monitorování komunikace a protokoly CC založené na distribuovaném adresáři.

## 5.3 Protokoly CC založené na monitorování komunikace (naslouchání, *snooping*)

- Pro systémy, které mají sběrnici a které mají broadcast. Stav bloku je připojen ke všem jeho kopiím v místních cache. Žádost s adresou bloku jde od žadatele všem (broadcast), reagují a odpovídají jen ti, kterých se to týká.
- Není možné modifikovat více než jednu proměnnou v jednom taktu!
- Většinou používají UMA systémy.



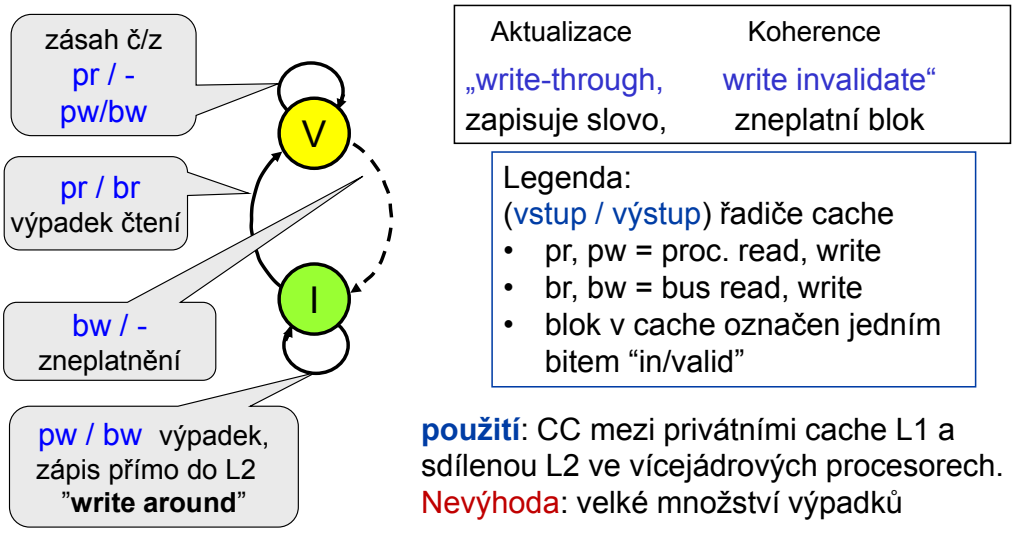

Obrázek 5.3: Řadič paměti cache s nasloucháním.

- Aktualizace paměti – procesor zapíše do cache, kdy data zapsat do hlavní paměti?
  - Při cache hit
    - **Write-through** – Průpis slova do L1 a hned výš.
    - **Write-back** – Zpětný zápis bloku z L1 a výš až když je to nutné (musím danou cache line vyhodit).
  - Při cache miss
    - **Write-allocate** – Chci aktualizovat data, ale v L1 nejsou, musíme najít místo, natáhnout data z vyšší úrovně a pak je přepsat.
    - **Write-around** – Chci aktualizovat data, ale v L1 nejsou, tak chodím o úroveň výš, dokud data nenajdu a přepíšu je tam. Pro data, která už nikdy později nepoužiju, nemá smysl je tahat tedy do cache.
- Udržování koherence – jak dát ostatním vědět, že data jsou neplatná?
  - **Write-update**
    - Jakmile se hodnota změní, pošle se broadcastem ostatním.
  - **Write-invalidate**
    - Jakmile zjistím, že hodnotu někdo někde jinde změnil, tak svoji hodnotu zneplatním (valid bit na 0).

### 5.3.1 Dvou-stavový protokol na sběrnici

- Write-through, write-invalidate.
- Triviální dvoustavový protokol.

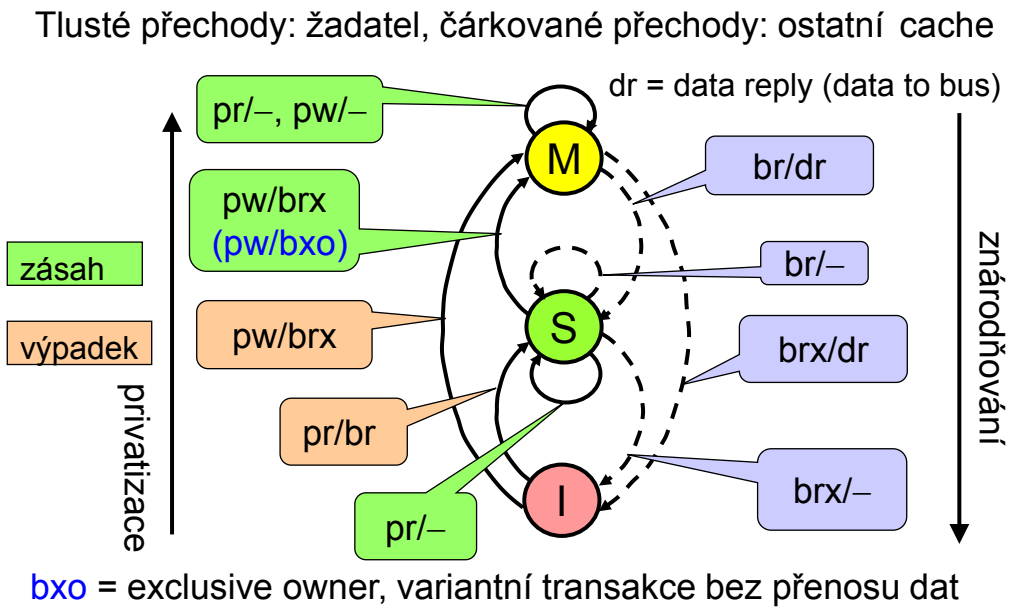
- Používá se mezi L1 a L2, protože tyto cache jsou privátní pro jádro.
- Stavy bloku: platný (*valid*) / neplatný (*invalid*).




Obrázek 5.4: Dvoustavový protokol na sběrnici. Plná čára – co dělám já v lokální cache; čárkovaná – co dělají cache jiných jader.

### 5.3.2 Tří-stavový protokol MSI

- Write-back, write-invalidate.
- Používá se všude jinde než mezi mezi L1 a L2.
- Aby se nemusely opakované zápisy do bloku cache neustále propisovat výš do SM, je možné špinavou kopii v L1 cache označit dalším stavem M a nechat ji v L1 co nejdéle (dokud nebude blok vybrán k výměně za jiný blok).
- Stavy bloku:
  - **Modifikovaný** (M, *modified*) – Jen 1 cache má platnou špinavou kopii, kopie ve sdílené paměti SM je zastaralá (dirty bit je 1).
  - **Sdílený** (S, *shared*) – 1 nebo více pamětí cache má kopii shodnou se sdílenou pamětí SM (čistá kopie).
  - **Neplatný** (I, *invalid*) – V této cache je pouze neplatná kopie (zastaralá).
- Blok S lze jen číst. Před zápisem musí CPU změnit jeho stav na M a ostatní kopie S musí být zneplatněny.
- CPU je pak jediný vlastník bloku M, má k němu exkluzivní přístup a může do něj zapisovat.




Obrázek 5.5: Třístavový protokol MSI.

### 5.3.3 Čtyř-stavový protokol MESI

- Další stav **E** bloku cache značí, že existuje čistá kopie pouze v jediné cache. Stačí 2-bitový kód jako pro MSI.
- Výhody:
  - Ve stavu **E** (i **M**) je možný zápis (E→M, M→M) **bez oznamování** brx nebo bxo na **sběrnici**
  - Blok **E** dodá cache dalšímu žadateli (E →S) rychleji místo Mem!

- Na sběrnici jsou ale teď místo signálu **dirty bit d** nově dva wired-OR signály:
  - owner bit e (exkluzivní vlastník, signalizuje **M** i **E**) a
  - shared bit f (kopie jen v paměti a v žádné cache: f = false;
    existuje platná kopie alespoň v jedné cache: f = true).

### 5.3.4 Pěti-stavový protokol MESIF

- Intel

- Kolektivní cíl všech cache je minimalizovat přístupy do sdílené hlavní paměti mimo čip, využít data na čipu.
- Nový stav F, read-only forwarding, umožňuje přenos čisté kopie z jedné cache do druhé. **Je to rychlejší (např. u vícejádrových procesorů) než přenos z paměti.**
- Ve sdílených kopiích je vždy jen jedna ve stavu F a ta se kopíruje. Ostatní kopie jsou ve stavu S.
- Když je blok ve stavu F kopírován, stav F migruje do nové kopie zatímco zdrojová kopie přejde do stavu S.
- Blok M se musí před jeho sdílením nebo výměnou (kvůli místu) nahrát zpět do paměti.

### 5.3.5 Pěti-stavový protokol MOESI

- AMD
  - MOESI má jinou optimalizaci:
    1. **eliminuje kopírování bloku M do paměti před jeho sdílením**; původní blok M přejde do stavu O (owned),
    2. **další sdílené špinavé kopie S** vznikají kopírováním dat z bloku O. Kopie v paměti je zastaralá.
  - Stavy E a O a M jsou unikátní (blok jen v jedné cache)
  - Sdílené kopie jsou buď špinavé M $\rightarrow$ [O, S, S, S,… ] nebo čisté E $\rightarrow$ [ S, S, S, …]
  - Stavy M, O, E dovolují zápis (O a E přejdou do M) a nahrazují paměť jako zdroj dat (přenos cache-to-cache).

## 5.4 Protokoly CC založené na distribuovaném adresáři

- Stavy bloku paměti v jednotlivých cache jsou uloženy ještě odděleně v některém z adresářů. Žádost jde od žadatele přes domovský adresář bloku jen majitelům kopií, jen ti také odpovídají.
- Většinou používají NUMA systémy.

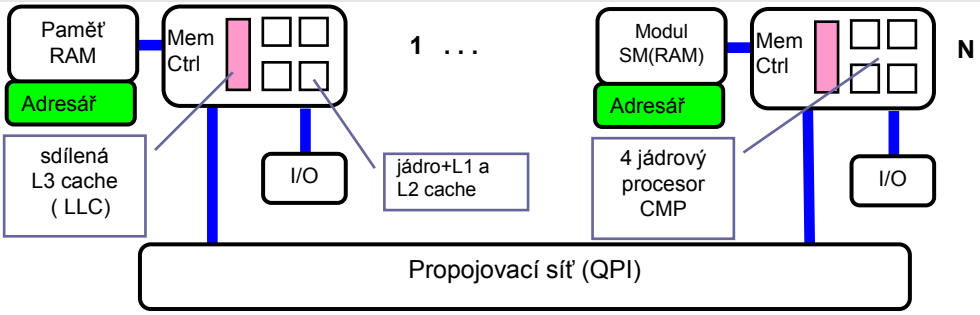
- U CMP se sběrnicí je snadný broadcast signálů *br*, *brx*, *bxo* a komunikace bitů *d*, *e*, *f*. Arbitr sběrnice také žádosti seřazuje.
- U multiprocesorů NUMA s větším počtem CMP by byl broadcast složitý. Proto žadatel komunikuje **přes prostředníka** (zástupce). Tímto prostředníkem jsou **distribuované adresáře** (directories) a jejich řadiče (DirCtrl) u jednotlivých modulů DSM.
- **Domovský adresář** (Home Directory, *H*) pro určitý modul SM udržuje informace o blocích (cache line) v modulu (RAM):
  - jestli je blok platný, čistý nebo špinavý
  - kde se nachází, ve které (**sdílené last level**) cache (LLC).
- Řadič domovského adresáře **DirCtrl:**
  - Seřazuje příchozí žádosti a reaguje na ně.
  - Příkazy (zprávy) pro koherenci, ale místo broadcastu zasílá **pouze do relevantních uzlů** (multicast).
  - To u velkých systémů redukuje enormně komunikaci.

- **Záznam v adresáři o bloku na adrese *X* je *N*+1 bitový vektor:**
  - nultý clean/dirty bit $V_0$
  - *N* presenčních bitů $V_i$, i =1,2, …, *N* (bit-mapa):

```
bit    0 1 2 3 4 5 …    N
X  [  0 0 1 1 0 1 … 0 1 ] sdílený blok S je ve více cache
X  [  0 0 0 0 0 0 … 0 0 ] samé nuly: blok je jen v RAM
X  [  1 0 0 1 0 0 … 0 0 ] špinavý blok M je v cache 3
```

- **Bloky v cache** jsou označeny duplicitně dirty/clean bitem (kromě valid bitu).
- Stavové bity jsou čteny a modifikovány dvěma řadiči: CacheCtrl a DirCtrl. (V názvosloví Intel: „Cache agent“ a „Home agent“ nebo jen „Home“). Zpracovávají zprávy CC a datové odpovědi.

## Malý multiprocesor DSM CC-NUMA CC založená na adresářích




Sekvence komunikací u protokolu CC s adresáři:

1. žádost R/W → Home
2. Home kontaktuje jen relevantní agenty CacheCtrl
3. CacheCtrl: odpověď (potvrzení)→ Home, datová odpověď žadateli, případně i do Home
4. Home: povolení R/W → žadateli

- *Řadiče (agenti) DirCtrl a CacheCtrl modifikují podle potřeby stavové bity bloků.*
- Všechna data procházejí přes cache L1. Mohou být při nedostatku místa vyhozeny do vyšší úrovně cache nebo až do SM RAM.

# Kapitola 6

# AVS – Paralelní zpracování v OpenMP: Smyčky, sekce a tasky a synchronizační prostředky.

## 6.1 Zdroje

- Moje materiály ke zkoušce z AVS.
- `AVS-07.pdf`
- `AVS-08.pdf`
- `AVS-10.pdf`
- *Otázka není vysázená, pouze vloženo exportované PDF z google docs.*

# 7) OpenMP — Paralelizace smyček

- Doposud jsme se věnovali pouze jádru procesoru a co tam udělat pro zvýšení výkonu — zpracování instrukcí mimo pořadí, přeuspořádávání, přednačítání, spekulace, datový paralelismus, ...
- Nyní abstrahujeme od samotného jádra procesoru a budeme uvažovat paralelizaci na vyšší úrovni, tedy na více jádrech procesoru.
- Přednášky 7, 8, 10 jsou jak se to řeší softwarově a 11, 12, 13 jak je to udělané v hardware.

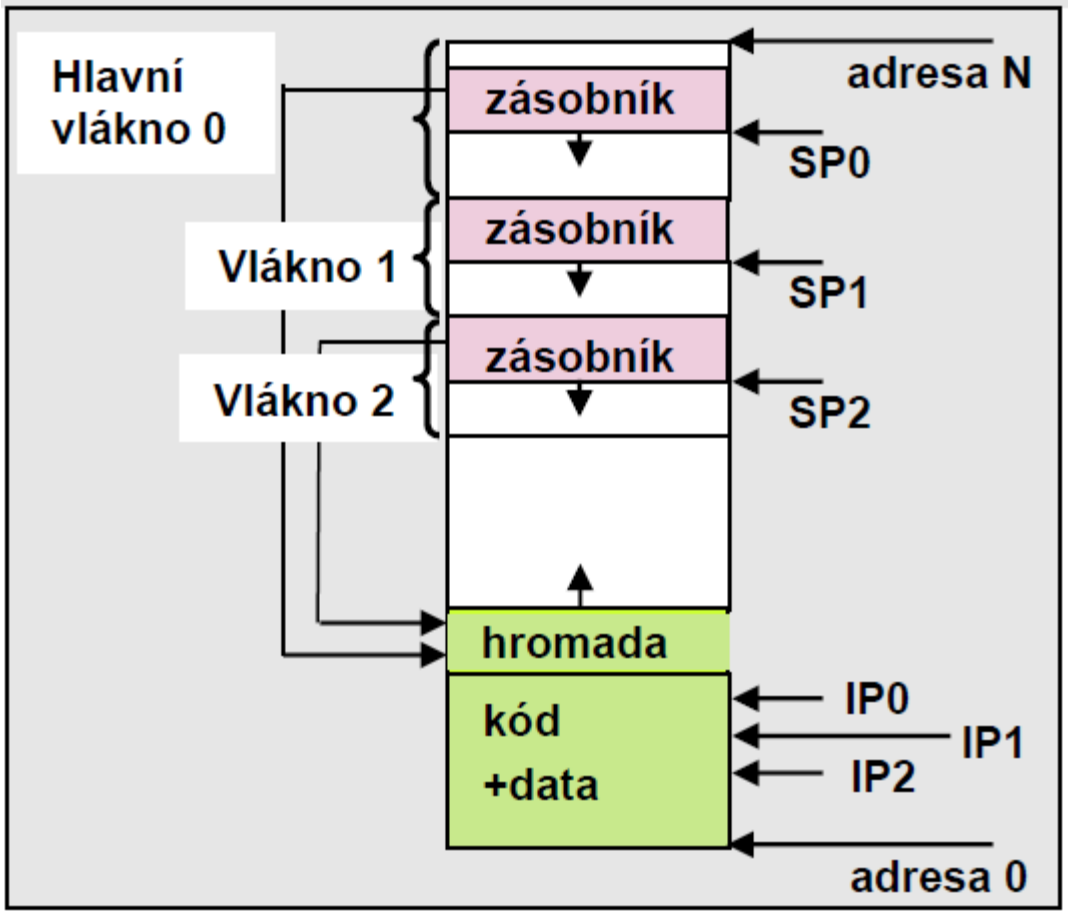
## Úvod

### Paralelní programovací modely

- SISD — Single Instruction Single Data
  - Skalární procesor
- SIMD — Single Instruction Multiple Data
  - Vektorové procesory
- MISD — Multiple Instruction Single Data
  - Používá se pouze ve velmi speciálních případech
- MIMD — Multiple Instruction Multiple Data
  - Vícevláknové procesory

### Sdílený adresný prostor

- Vlákna fungují na systémech se sdílenou pamětí




- Každé vlákno má svůj zásobník, tedy cokoliv naalokované staticky, je privátní pro vlákno
- Halda je společná pro všechna vlákna, tedy cokoliv naalokované dynamicky (malloc, new, ...) je sdílené

## Jak paralelizovat

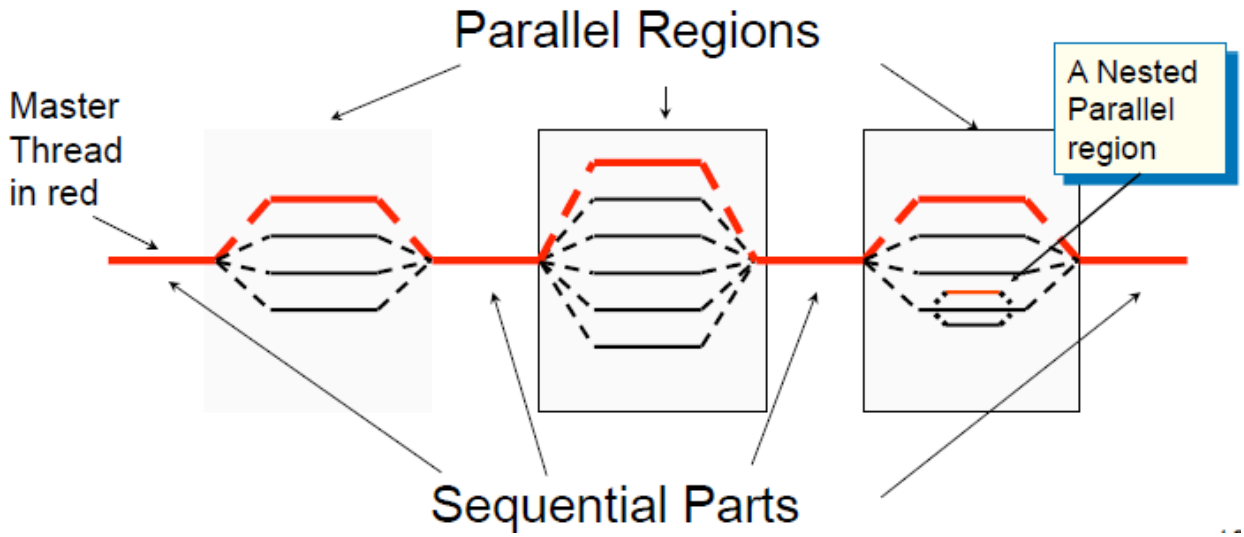
- Sekvenční jazyk + příkazy paralelního zpracování, komunikace a synchronizace
- Sekvenční jazyk + dynamické knihovny
- **Sekvenční jazyk + direktivy pro kompilátor (pragma)**
- OpenMP API — kombinace direktiv a knihovních programů

## Výpočetní model vláken

- Vlákna jsou ovládaná a použitá kernelem OS
- Hardware vlákno

## Model zpracování vláken

- Strukturovaný blok




- Na konci bloku bariéra — sloučení vláken (zabití potomků)

## Vytvoření týmu vláken

### Direktiva

#### pragma omp parallel

- Vytvoří paralelních sekci — vlákna

```
omp_set_num_threads(4);
double a[1000];

#pragma omp parallel [clause[clause]... ]
{
    int id = omp_get_thread_num();
    work(id, a);
}
```

### Dovětky

- Upravují chování direktivy parallel

#### private(list)

- Udělat privátní kopii sdílené proměnné
- Deklaruje novou privátní proměnnou, ale neinicializuje!

```
int a = 5;
#pragma omp parallel private(a)
```

```
{
    printf("a = %d\n", a);  // vypise smeti
}
```

### firstprivate(list)

- Stejný význam jako private, pouze s inicializací na hodnotu, ze které se tvoří kopie.

```
int a = 5;
#pragma omp parallel firstprivate(a)
{
    printf("a = %d\n", a);  // vypise 5
}
```

- Privatizovat lze pouze uplné objekty, ne např. pouze nějaké prvky pole
- Privatizovat lze pouze objekty, které mají copy construktor nebo primitivní datové typy
    - C++ Vector lze, C struktura lze, ...
    - C array nelze (nevíme kolik má prvků), pokud to uděláme, dostaneme neinicializovaný pointer — segfault

```
float a[10] = {0};
#pragma omp parallel private(a)
{
    a[3] = 123;   // segfault
}
```

### shared(list)

### default(shared|none)

- Výchozí stav, vše nad paralelní sekcí je sdílené, můžu změnit, každé proměnné můžu nastavit jestli má být sdílená nebo privátní

### reduction(operator:list)

- Dílčí výsledky do finální hodnoty

### if(logical expression)

- Kdy vytvořit paralelní kód

### copyin(list)

### num_threads(thread_count)

- Kolik má mít sekce vláken

## Paralelizace smyček

- Paralelizuje se nejvnější smyčka
- Platí stejné podmínky jako u vektorizace

```
#pragma omp parallel
#pragma omp for
for (int i = 0; i < N; i++) {
    a[i] += b[i];
}
```

Tyto zápisy jsou ekvivalentní:

```
double res[MAX]; int i;
#pragma omp parallel
{
  #pragma omp for
  for (i = 0; i < MAX; i++)
     res[i]  = huge();
  }
}
```

```
double res[MAX]; int i;
#pragma omp parallel for
{
  for (i = 0; i < MAX; i++)
     res[i]  = huge();
  }
}
```

když paralelní oblast obsahuje jen #pragma omp for

## Direktiva

### pragma omp for

- Iterační paralelismus
- Vlákna si nějakým způsobem rozeberou iterace
- Zaručuje že je to vhodná smyčka (stejné podmínky jako pro vektorizaci)

## Dovětky

### private(list)

- Stejný význam

### firstprivate(list)

- Stejný význam

### lastprivate(list)

- Vezme se hodnota, která vznikne v poslední iteraci a uloží se zpět do původní proměnné mimo paralelní sekci

### reduction(operator:list)

- Stejný význam
- Operátory:
  - +, -, *, min, max, bitové operace, nebo vlastní redukce

```
double A[MAX];
double ave = 0.0;

#pragma omp parallel for reduction (+: ave)
for (int i = 0; i < MAX; i++) {
    ave += A[i];
}
ave = ave / MAX
```

### schedule(kind[, chunk_size]

- Jak rozložit zátěž mezi vlákna
- Kolik iterací dostane jaké vlákno?

### nowait

- Na konci smyčky není jinak implicitní bariéra

# Dovětek nowait

```
#pragma omp parallel default node \
        shared(n, a, b, c, d) private (i)
{
  #pragma omp for nowait
  for (int i=0; i < n-1; x++)
  {
    b[i] = (a[i] + [a[i+1]) / 2;
  }

  #pragma omp for nowait
  for (int i = 0; i < n; y++)
  {
     d[i] = 1.0 / c[i];
  }
}
```

Zrušena implicitní bariera

## collapse

- Vytvoří jednu smyčku ze dvou vnořených a tu paralelizuje

```
#pragma omp parallel for collapse(2)
for (int y = 0; y < 25; ++y) {
    for (int x = 0; x < 80; ++x) {
        tick(x, y);
    }
}
```

## Paralelizace smyček se závislostmi

- Mohu zaměnit vnitřní / vnější smyčky tak, aby vnější byla bez závilostí a mohla se paralelizovat

Jak udělat iterace smyčky nezávislé, aby mohly být prováděny v libovolném pořadí bez závislostí?

Příklad:

```
for (i=2; i<=m; i++)
  for (j=1; j<=n; j++)
    a[i][j] = 2 * a[i-1][j];
```

Závislost v indexu `i` přemístíme do sekvenční smyčky, tam nevadí:

```
int i,j;
#pragma parallel for private(i)
for (j=1; j<=n; j++)    ← paralelní, j je impl. privátní
 for (i=2; i<=m; i++)   ← sekvenční, i je privátní
  a[i][j] = 2 * a[i-1][j];
```

Pozor ale na lokalitu dat!

## Plánování iterací

- Jak rozdělit iterace mezi vlákna?
- Lze specifikovat pomocí dovětku
  - schedule({static, dynamic, guided}, [chunk_size])

src: http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

## Chunksize

- Po kolika iteracích rozděluji
- Výchozí velikost = 1

## Schedule type

### Static

- Každému vláknu přidělím stejný počet iterací
- Vhodné pokud jsou iterace stejně náročné
- Nejmenší režie, rozdělení spočítá předem

```
schedule(static):
****************
                ****************
                                ****************
                                                ****************
```

```
schedule(static, 4):
****            ****            ****            ****
    ****            ****            ****            ****
        ****            ****            ****            ****
            ****            ****            ****            ****
```

```
schedule(static, 8):
********                        ********
        ********                        ********
                ********                        ********
                        ********                        ********
```

## Dynamic

- Vhodné pokud je každá iterace jinak náročná a já nemám způsob jak to spočítat
- Čím menší chunk size, tím větší režie (synchronizovaný přístup k indexu smyčky — zamykání), ale zase horší vyvážení zátěže — je třeba experimentálně najít něco vhodného

```
schedule(dynamic):
*   ** **  * * *  *      *  *    **   *  *  * *       *  *   *
  *       *     *    * *     * *   *    *        * *   *    *
 *       *    *     * *   *   *     *  *       *  *  *  *  *   *
   *  *     *    * *    *  *    *    *    ** *  *   *     *   *
```

```
schedule(dynamic, 1):
    *    *     *        *   *    * *  *  *         *  * *  * *
*  *  *   * *     *  * * *    * *      *   ***  *   *         *
 *   *  *  *  *    ** *    *      *  *  * *   *  *   *   *
  *    *     *  **        *  * *    *         *  *    *  * *  *
```

```
schedule(dynamic, 4):
            ****                    ****                    ****
****            ****    ****            ****        ****
    ****            ****    ****            ****        ****
        ****                    ****            ****
```

```
schedule(dynamic, 8):
                ********                                ********
                        ********        ********
********                        ********        ********
        ********
```

## Guided

- Na počátku větší chunk size a progresivně snižuju
- Tedy nějaké vyvažování zátěže
- Jak?
  - První přidělení chunku:
    - q_0 = number_of_iterations / number_of_threads
  - Každé další přidělení:
    - q_i = q_{i-1} * (1 - 1 / number_of_threads)
- Příklad:
  - Smyčka s 1000 iteracemi
  - Paralelizace na 4 vláknech
  - První chunk: 1000 / 4 = 250
  - Druhý chunk: 250 * 3 / 4 = 188
  - Třetí chunk: 188 * 3 / 4 = 141
  - ...

```
schedule(guided):
                            *********                       *
                ************                    *******  ***
                                      *******                  *
****************                            *****       **    *
```

```
schedule(guided, 2):
                ************                    ****     **
                                      *******         ***    **
                            *********
****************                            *****       **    **
```

```
schedule(guided, 4):
                                      *******
                ************                    ****    ****
                            *********
****************                            *****    ****    ***
```

```
schedule(guided, 8):
                ************                 ********        ***
****************
                                      ********
                            *********                ********
```

# Příklady

## Příklad 1

### Příklad: Privátní proměnné

```
int x = 5, y = 6, z = 7;
float a[10], b[10], c[10];
#pragma omp parallel num_threads(5) \
        private(x, a)               \
        firstprivate(y, b)          \
        shared(z, c)                \
{
  int thread_id = omp_get_thread_num();
  x++; y++; z++;

  a[thread_id] = 0;
  b[thread_id] = 1;
  c[thread_id] = 2;

  a += thread_id; *a = 5;
  b += thread_id; *b = 5;
  c += thread_id; *c = 5;
}
```

Co bude v proměnných?

Kam zapíši?

Kam zapíši?

```
int x = 5, y = 6, z = 7;
float a[10], b[10], c[10];

#pragma omp parallel
    num_threads(5)
    private(x, a)
    firstprivate(y, b)
    shared(z, c)
{
    int thread_id = omp_get_thread_num();  // privatni

    x++;  // x = smeti
    y++;  // y = 7 v kazdem vlaknu
    z++;  // race condition, zde by byla treba kriticka sekce
          // a nejake atomicke pricteni, potom by byl vysledek
          // z = 7, 8, 9, 10, 11 postupne pro dana vlakna

    a[thread_id] = 0;  // pristup pres neinicializovany pointer,
                       // segfault
```

```
    b[thread_id] = 1;  // vsechna vlakna zapisuji na svuj index
                       // dvojku, kazde ma svoji privatni kopii
                       // pointeru

    c[thread_id] = 2;  // vsechna vlakna zapisuji na svuj index
                       // dvojku, jeden sdileny pointer

    a += thread_id;  // posunuti neinicializovaneho pointeru,
                     // segfault

    b += thread_id;  // kazde vlakno si posune svuj pointer o svoje
                     // id

    c += thread_id;  // race condition, blbost

    *a = 5;  // pristup pres neinicializovany pointer, segfault
    *b = 5;  // kazde vlakno zapise 5 na svuj index
    *c = 5;  // neni garantovano kam se to zapise, kvuli race
             // condition
}
```

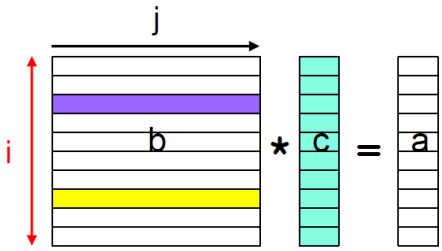
## Příklad 2

### Příklad: matice b(m×n) krát vektor c(n×1) = vektor a(m×1)

```
void mxv(int m, int n, double* a, double* b[],
            double* c)
{
 #pragma omp parallel for default(none)\
          shared(m,n,a,b,c)
 for (int i=0; i < m; i++)
 {
  double sum = 0.0;
  for (int j=0; j < n; j++)
     sum += b[i][j]*c[j];
  a[i] = sum;
 }
}
```

paralelní smyčka

sekvenční smyčka




Bude-li *m* = 10 řádků a 2 vlákna,
pak thread 0 si vezme řádky *i* = <0, 4>
a thread 1 řádky *i* = <5, 9>.

```
void mxv(int m, int n, double* a, double* b[], double* c)
{
```

```
    #pragma omp parallel for \
        default(none) \
        shared(m, n, a, b, c)
    for (int i = 0; i < m; i++) {
        double sum = 0.0;
        for (int j = 0; j < n; j++) {
            sum += b[i][j] * c[j];
            a[i] = sum;
        }
    }
}
```

Příklad 3

## Příklad redukce

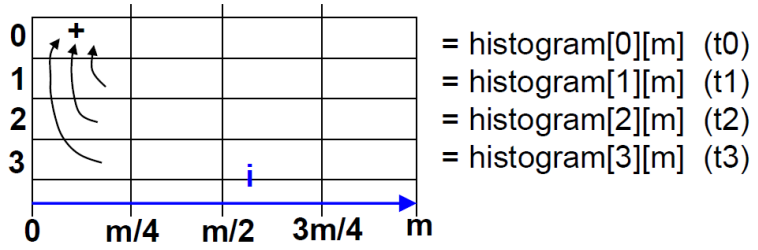
```
int mini = a[0];
int maxi = a[0];
for (i=1; i<n; i++)
{
  if (a[i]< mini)
    mini = a[i];
  if (a[i]> maxi)
    maxi = a[i];
}
```

```
int mini, maxi;
# pragma omp parallel for reduction \
          (min:mini, max:maxi)
```

Příklad 4

## Příklad výpočtu histogram

- Vektor a[n] má prvky integer v intervalu <0, m-1>. Četnost výskytu různých hodnot se má znázornit histogramem histogram[m].
  - Nejdřív každé z *P* vláken vytvoří vlastní histogram histogram[myid][m], myid = 0, …, *nt*-1 ze svého segmentu vektoru a[n]
  - pak každé vlákno posčítá jednu část všech *P* histogramů do části jednoho globálního histogramu histogram [0][m].




## Příklad: histogram hodnot prvků vektoru a[n]

```
int histogram [10][m];                    // až 10 vláken
#pragma omp parallel shared (a, m, n)
{
  int nt, myid, i, k;
  nt = omp_get_num_threads();
  myid = omp_get_thread_num();
  #pragma omp for schedule(static)
  for (i = 0; i < n; i++)                 //segmenty vektoru
     histogram[myid][a[i]]+=1;
  #pragma omp for schedule(static)
  for (i = 0; i < m; i++) {               //segmenty histogramů
    for (k = 1; k < nt; k++)              //vláken k > 0 do k = 0
      histogram[0][i]+=histogram[k][i];
  }
}
```

Příklad 5

## Řízené plánování, příklad 4 vlákna = 4 barvy

**Příklad:** 200 iterací. Postupně odebrané porce:

**50 =** $\lceil 200/4 \rceil$
**38 =** $\lceil 50(1 - 1/4) \rceil$,
**29 =** $\lceil 38(1 - 1/4) \rceil$,
**22 =** $\lceil 29(1 - 1/4) \rceil$, **vlákno skončilo 1. porci iterací,**
**17, … dostane 2. porci**
**13, …. vlákno přichází pro 2. porci**
**10, … vlákno přichází pro 2. porci**
**8, … vlákno přichází pro 3. porci,**
**6, … atd.**
**5,**
**2.**

Celkem iterací na vlákna: 50, 50, 48, 52 a **11 synchronizovaných přístupů** k indexu smyčky

Příklad 6

## Příklad: rozvržení iterací na vlákna

- Mějme 35 stejně náročných iterací 0 až 34, 3 vlákna 0 až 2. Najděte všechny dávky iterací přidělené vláknům 0-2.

| | **vlákno 0** | **vlákno 1** | **vlákno 2** |
|---|---|---|---|
| static, no chunk | 12 | 11 | 11 |
| static, chunk = 5 | 5 + 5 + 4 | 5 + 5 | 5 + 5 |
| dynamic, chunk = 7 | 7 + 7 | 7 +6 | 7 |
| guided, no chunk | 12 | 8 + 3 | 6 + 4 + 1 |

Příklad 7

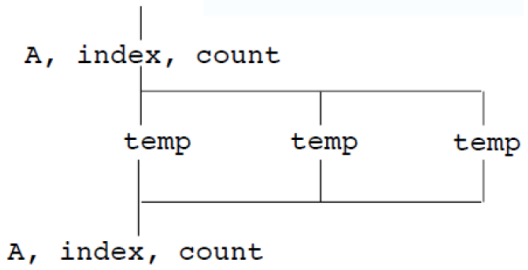
## Příklad sdílených a privátních dat

```
extern double A[10];
void work(int * index)
{
  double temp[10];
  static int count;
}
```

```
double A[10];
int main(){
  int index[10];
  #pragma omp parallel
     work(index);
   printf("%d\n", index[0]);
}
```

**A, index** a **count** jsou **sdílené** všemi vlákny.

**temp** je lokální (**privátní**) v každém vláknu.




Příklad 8

## Příklad

```
main(){
int C, B; int A = 20, n = 100, idx, *data;
...   // alokace a načtení vektoru data

#pragma omp parallel
 {
  #pragma omp for firstprivate(A)\
                  lastprivate(B,idx)
  for (int i = 0; i < n; i++) {
     B = A + i;                  /* A: je-li jen private, není def. */
     if (data[i] == 0) idx = i;
  }
  C = B;    /* B: je-li jen private, není hodnota B a tedy C def. */
            /* sdílené C = lastprivate B = 20+n-1 */
 }  /* konec paralelní oblasti: idx je náhodně nastaveno některými vlákny v některých
       iteracích. Lastprivate je zde nesmysl, lépe je použít redukci (např. když hledáme nejvyšší
       index nulového prvku ve vektoru data). */
```

# 8) OpenMP — Sekce a tasky

- S paralelníma oblastma je třeba šetřit, respektive jejich režie je poměrně náročná.

# Paralelní sekce

- Pro funkční paralelismus
    - Řada funkcí, které je možné vykonávat nezávisle

## Direktivy

### pragma omp sections

- Do této oblasti přijdou vlákna a začnou si rozebírat jednotlivé sekce
- Mohu používat stejné dovětky jako při for a simd
- Má na konci bariéru (stejně jako parallel)

### pragma omp section

- Jednotlivé sekce pro rozebrání
- Nevím pořadí vykonání ani jaké vlákno vykoná co
- Každou sekci jedno vlákno
- Počet sekcí není možné tvořit za běhu programu
- V OpenMP neexistuje žádný způsob jak komunikovat mezi sekcemi, musel bych napsat vlastní komunikaci přes signálování

```
#pragma omp parallel
{
    #pragma omp sections [clause[ clause] ...]
    {
        #pragma omp section
        {
            work1();
        }
        #pragma omp section
        {
            work2();
            work3();
        }
        #pragma omp section
        {
            work4();
        }
    }  // implicitni bariera pokud se nepouzije nowait
}
```

- Mám 3 nezávislé množiny funkcí, vytvořím 3 sekce

## pragma omp single

- Blok příkazů, která se vykoná pouze jednou a to prvním vláknen, které k němu dojde
- Na konci bloku je bariéra
- Dovětek copyprivate
    - Vlákno single zpřístupní svoje privátní proměnné ostatním vláknům

```
#pragma omp single
```

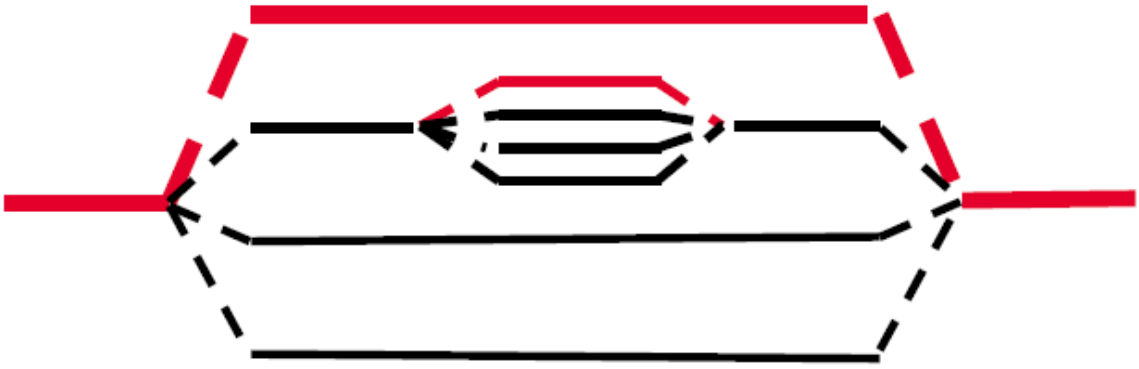
## pragma omp master

- Blok příkazů, která se vykoná hlavním vláknem
- Na konci bloku není bariéra

```
#pragma omp master
```

## Vnořený paralelismus

- Uvnitř sekce nový pragma omp parallel



## Pravidla používání direktiv sdílení práce (for, sections, single)

- K příkazu musí dojít všechna vlákna nebo žádné.
- Každé vlákno musí projít sérii příkazů sdílení práce ve stejném pořadí.
- Není dovoleno skočit dovnitř nebo ven z bloku spojeného s tímto příkazem (jedině exit()).
- Vnořování příkazů sdílení práce je nelegální.
- Příkazy sdílení práce nesmí uvnitř obsahovat bariéru.

- Mají implicitní bariéru nebo nowait na konci.
- Direktivy sdílení práce mohou být sirotci.

## Tasky

- Blok příkazů, který je přidám do fronty, odkud si jej berou jednotlivý vlákna, co zrovna nemají co na práci.
- Jakmile nějaké vlákno narazí na task, přidá ho do fronty a jede dál.
- Oproti sekcím, můžeme generovat za chodu kolik a jak chceme!

## Direktivy

### pragma omp task

- Změna oproti zbytku OpenMP, všechny proměnné nad taskem nejsou public, ale firstprivate!
- Generování tasků probíhá vždy když někdo narazí na pragma omp task, tedy je třeba použít single (to má bariéru a čeká se na vygenerování tasků) nebo master

```
#pragma omp parallel
{
    #pragma omp master  // Thread 0 prida tasky do fronty
    {
        #pragma omp task
        {
            fred();
        }
        #pragma omp task
        {
            daisy();
        }
        #pragma omp task
        {
            billy();
        }
    }
}  // Zde vsechna vlakna cekaji a kontroluji frontu tasku,
   // vsechny tasky jsou pred touto barierou
```

## pragma omp taskwait

- Všechny tasky definované v aktuálním tasku, musí být dokončeny, než pokračuju.
- Pro rekurzi

```
#pragma omp parallel
{
    #pragma omp master
    {
        #pragma omp task
        {
            fred();
        }
        #pragma omp task
        {
            daisy();
        }
        #pragma omp taskwait
        #pragma omp task  // fred() a daisy() musi byt
        // dokonceny drive, nez se billy vubec prida do fronty
        {
            billy();
        }
    }
}
```

## pragma omp taskgroup

- Na konci skupiny jsou všechny tasky dokončeny

## pragma omp cancel

- Konec výpočtu, pro vyhledávání

## Příklad na OpenMP cancel – Je v matici nulový prvek?

```
bool has_zero = false;
#pragma omp parallel default(none) shared(matrix, has_zero)
{
   #pragma omp for
   for (int row = 0; row < rows; row++)
   {
     for (int col = 0; col < cols; col++)
     {
       if (matrix(row, col) == 0)
       {
         #pragma omp critical
         {
           has_zero = true;
         }
         #pragma omp cancel for
       }
     }
    #pragma omp cancellation point for
  }
}
```

Zde přerušíme výpočet

Zde se ostatní vlákna dozví, že mají ukončit výpočet

Probírali se ještě další věci, ale spíš takové detaily z OpenMP, hádám nebude na zkoušce.

# Příklady

## Příklad 1)

## Příklad na tasky: Průchod vázaným seznamem

```
my_pointer = listhead;
#pragma omp parallel
{
  #pragma omp single nowait
  {
    while (my_pointer)
    {
      #pragma omp task firstprivate(my_pointer)
      {
        do_independent_work (my_pointer);
      }
      my_pointer = my_pointer->next;
    }
  } // end of single - bariéra potlačena (nowait)
} // end of parallel region - implicitní bariéra
```

Jedno vlákno bude řídit smyčku while, generovat tasky pro další vlákna týmu

my_pointer musí být firstprivate, aby každý task měl def. svou hodnotu

blok 1

blok 2

blok 3

Všechny tasky dokončí zde

Příklad 2)

## Fibonacciova čísla sekvenčně - rekurzivně

| n | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| fib(n) | 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 |

```
if (n<2) return n;
```

**Sekvenčně:**

```
int seqfib(const int n)
{
  int x, y;
  if (n < 2) return n;

  x = seqfib(n - 1);
  y = seqfib(n - 2);
  return x + y;
}
```

Jakmile je dosaženo jisté hodnoty *n*, je lépe počítat fib(n) sekvenčně a režii tasků v OpenMP vynechat:

if (n≤ 30) return seqfib(n);

## Fibonacciova čísla paralelně s tasky

```
int main (int argc,
          char **argv)
{
  int n, result;
  n = atoi (argv[1]);
  #pragma omp parallel
  {
    #pragma omp single
    {
      result = fib(n);
    }
  }
   printf ("fib(%d)=%d\n",
            n, result);
}
```

```
int fib (int n)
{
  int x, y;
  if (n< 2 ) return n;
  if (n≤ 30) return seqfib(n);

#pragma omp task shared(x)
      x = fib(n-1);
#pragma omp task shared(y)
      y = fib(n-2);
#pragma omp taskwait
      return x+y
}
```

pozastav rodič. task, až dokončí dceřiné tasky

x+y musí být přístupné – shared, default je firstprivate

```
int fib_seq(int n)
{
    if (n < 2) {
        return n;
    }

    int a = fib_seq(n - 1);
    int b = fib_seq(n - 2);
```

```
    return a + b;
}

int fibb(int n)
{
    if (n < 30) {  // pro snadne vypocty nema smysl tvorit
                   // tasky, rezie by prevazovala
        return fib_seq(n);
    }
    else {

        int a, b;

        #pragma omp task shared(a)
        {
            a = fibb(n - 1);
        }

        #pragma omp task shared(b)
        {
            b = fibb(n - 2);
        }

        // zde je task pozastaven a vracen do fronty,
        // vlakno je volne a muze pracovat na necem jinem
        #pragma omp taskwait

        return a + b;
    }
}

int main()
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            fibb(100);
        }
    }  // zde je bariera a vlakna si rozebiraji tasky
```

```
    return 0;
}
```

# 10) OpenMP — synchronizace

## Úvod

- Synchronizace slouží
  - K ochraně přístupů ke sdílených datům
  - K čekání na nějakou událost (producent — konzument)
  - K vynucení pořadí akcí
  - ...

## Direktivy

### Vysoká úroveň

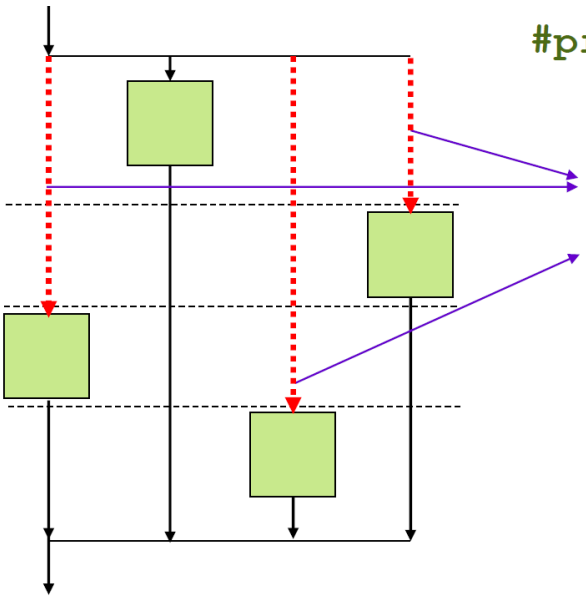
- critical
- atomic
- barrier
- master
- ordered

### Nízká úroveň

- Jsou součástí direktiv vyšší úrovně
- flush
- locks

# Vzájemné vyloučení

## Kritická sekce



```
#pragma omp critical[(name)]
    blok
```

Čekání na vstup do kritické sekce

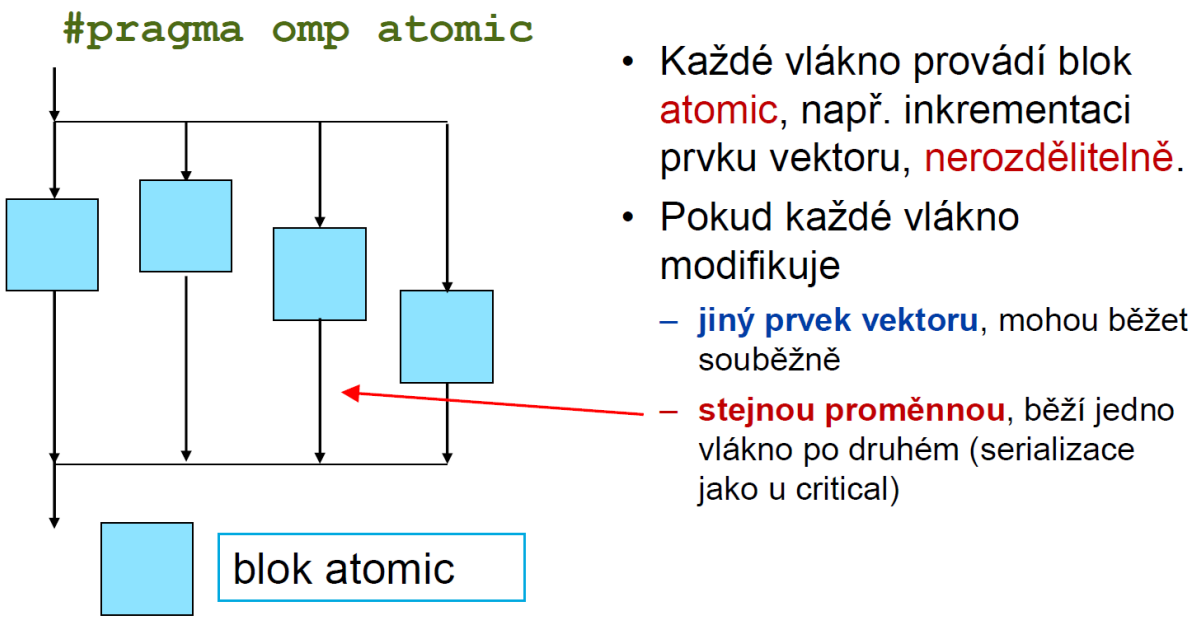
Každé vlákno provádí kritickou sekci, ale v určitém okamžiku pouze jediné.
Pořadí není zaručeno!

Kritická sekce

```
// hleda se index prvniho nuloveho prvku
int first = n;
#pragma omp parallel for
for (int i = 0; i < n; i++) {
    if (a[i] == 0 && i < first) {
        #pragma omp critical
        // podminka znovu, protoze nekdo pri mym cekani na
kritickou
        // sekci mohl zapsat
        if (i < first) {
            first = i;
        }
    }
}
```

- Kritická sekce je jedna a pouze globální! Pokud chci více kritických sekcí, je nutné použít identifikátory.
- Poměrně velká režie, když to jde, používat atomic.

## Atomická modifikace




- Chrání aktualizaci jednoho paměťového místa
- Pouze pro
  - Triviální datové typy, které načtu jednou instrukcí! (ne struktury)
    - int, float, double, char, ...
  - Triviální operace
    - +,*, , /, bitwise AND, XOR, OR (&, ^, |), <<, >>

## Dovětky

- read
- write
- update
  - Vychozi stav
- capture
  - Ctu hodnotu a zapisuju novou
- seq_cst
  - Vnucení konzistence, všechno co je předtím, se musí dokončit, žádné předbíhání

```
// ctu hodnotu a zapisuju novou
#pragma omp atomic capture
{
    old_value = *p;
    (*p)++;
```

```
}
```

```
// histogram
#pragma omp parallel
{
    #pragma omp for shared(histogram, a, n)
    for (i = 0; i < n; i++) {
        #pragma omp atomic
        histogram[a[i]] += 1;
    }
}
```

```
// v podstate implementace redukce se scitanim
s = 0;
#pragma omp parallel
{
    int mysum = 0;
    #pragma omp for nowait
    for (int i = 0; i < n; i++) {
        mysum += a[i];
    }
    #pragma omp atomic
    s += mysum;
}
```

## Synchronizace událostmi

- Jde o to, zařídit pořadí nějakých událostí

### Direktivy

#### Bariéra

- Výchozí u každé direktivy parallel, for, section, taskgroup
- Čeka se až dojdou všechna vlákna, poté se pokračuje
- Bariéry nejsou dovoleny v dynamickém rozsahu — nedává smysl
    - for, ordered, sections, single, master

```
#pragma omp barrier
```

### Ordered

- Oblast, která se provádí v sekvenčním pořadí
- Například pro logování a debugování, jinak moc nedává smysl

```
void work(int k)
{
    #pragma omp ordered
    printf("%d ", k);  // zde budou vlakna cekat
}

#pragma omp for ordered schedule(dynamic)
for (i = low; i < high; i += step) {
    work(i);
}
```

### Master

- Označuje blok kódu v rámci paralelní oblasti, který je vykonáván hlavním vláknem, ostatní vláka přeskočí
- Neobsahuje implicitní bariéru na konci
- Použití:
    - Pro čtení parametrů a generování tasků

```
#pragma omp master
```

## Synchronizace programovaná uživatelem

### Direktivy

### Flush

- Zábrana, která specifikuje, kde se v kódu povoluje předbíhání
- Nad flushem nic nesmí předbíhat to pod ním
- Vnutí sekvenční konzistenci
- Volitelný dovětek seznam proměnných, který se musí “spláchnout”

### Locks

- …

```c
// tato funkce je volala paralelne ruznymi vlakny
// implementace bariery pro N vlaken
void bariera(int N)
{
    static int counter = 0;

    #pragma omp flush

    #pragma omp atomic
    counter++;

    #pragma omp flush

    while (counter != N) {
        usleep(1);
        #pragma omp flush
    }

    #pragma omp master
    counter = 0;
}
```

# Kapitola 7

# MSP – Pravděpodobnost, podmíněná pravděpodobnost, nezávislost.

## 7.1 Zdroje

- `MSP_pred_01_Opakovani_Pravd-NP-NV.pdf`
- `SUR_2020-02-11.mp4`
- Wikipedia

## 7.2 Základy kombinatoriky

### 7.2.1 Variace

- Variace $k$-té třídy z $n$ prvků je každá uspořádaná $k$-tice vytvořená z celkového počtu $n$ prvků, přičemž při výběru záleží na pořadí jednotlivých prvků.
- **Variace bez opakování** je $k$-členná skupina utvořená z daných $n$ prvků tak, že v nich záleží na pořadí a žádný z daných prvků se v ní neopakuje.

$$V(k,n) = \frac{n!}{(n-k)!} \ , \ k \leq n$$

  - Např. Mějme fotbalové družstvo o celkem 20ti hráčích. Kolik možných základních 11 je možné sestavit? Posty jsou respektovány.

- **Variace s opakováním** je uspořádaná $k$-tice z $n$ prvků sestavená tak, že každý se v ní vyskytuje nejvýše $k$-krát.

$$V'(k,n) = n^k \ , \ k < n$$

  - Např. Kolik možných hesel, je možné vytvořit, ze znaků a-z, A-Z, 0-9?

### 7.2.2 Permutace

- Permutace $n$-prvkové množiny je uspořádaná $n$-tice obsahující každý prvek právě jednou, takže jednoznačně určuje jedno z možných uspořádání těchto prvků. Odtud (řídce užívané) české synonymum pro permutaci pořadí. Ekvivalentní definice je, že se jedná o $n$-prvkovou variaci z $n$ prvků.

- **Permutace bez opakování** – Pokud se prvky ve výběru nemohou opakovat, pak počet všech možných pořadí je určen vztahem.

$$P(n) = n!$$

  - Např. na konferenci vystoupí 5 řečníků, jaký je počet všech možných pořadí, ve kterých vystoupí?

- **Permutace s opakováním** – Pokud se prvky ve výběru mohou opakovat, pak počet permutací s opakováním z $n$ prvků je určen jako.

$$P'(k_1, k_2, \ldots, k_n) = \frac{(k_1 + k_2 + \ldots + k_n)!}{k_1! \cdot k_2! \cdot \ldots \cdot k_n!}$$

### 7.2.3 Kombinace

- Kombinace je základní pojem z kombinatoriky, $k$-členná kombinace z $n$ prvků je skupina $k$ prvků, vybraná z $n$ různých prvků, u níž nezáleží na jejich pořadí. Od variace se liší tím, že je neuspořádaná.
- **Kombinace bez opakování** – Počet kombinací $k$-té třídy z $n$-prvků bez opakování, neuspořádaných $k$-tic vybraných z těchto prvků tak, že se v ní každý vyskytuje nejvýše jednou, je

$$C_k(n) = \binom{n}{k} = \frac{n!}{k!(n-k)!}$$

  - Např. V rovině je 6 různých bodů (žádné 3 neleží na jedné přímce). Kolik různých úseček dostaneme pospojováním všech těchto bodů navzájem?
  - Např. Mějme fotbalové družstvo o celkem 20ti hráčích. Kolik může být kombinací základních 11? Když neřešíme posty.

- **Kombinace s opakováním** – Počet kombinací $k$-té třídy z $n$ prvků s opakováním, tzn. každý prvek se ve výběru může objevit vícekrát, je určen vztahem.

$$C'_k(n) = \binom{(n+k-1)}{n-k} = \binom{(n+k-1)}{k} = \frac{(n+k-1)!}{k!(n-1)!}$$

- Kombinační čísla
  - Platí:

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!} \tag{7.1}$$

$$1 = \binom{0}{0} = \binom{n}{0} = \binom{n}{n} \tag{7.2}$$

$$0 \leq k \leq n \tag{7.3}$$

$$\binom{n}{1} = n \tag{7.4}$$

$$0 \leq k \leq n \tag{7.5}$$

## 7.3 Základy pravděpodobnosti

- Experiment (také vědecký pokus) je soubor jednání a pozorování, jehož účelem je ověřit (verifikovat) nebo vyvrátit (falzifikovat) hypotézu nebo poznatek, které něco tvrdí o příčinných vztazích určitých fenoménů.
  - Deterministický pokus – Vede vždy k jedinému možnému výsledku.
  - Náhodný (stochastický) pokus – Vede k jednomu z více možných výsledků.
- **Základní prostor** – Množina všech možných výsledků pokusu, značíme $\Omega$.
- **Náhodný jev** – Tvrzení o výsledku náhodného pokusu, o kterém lze po uskutečnění pokusu jednoznačně rozhodnout, zda je čí není pravdivé. Formálněji; nechť $A$ je náhodný jev, platí $A \subseteq \Omega$, tj. náhodným jevem rozumíme libovolnou podmnožinu základního prostoru.
- **Elementární náhodný jev** – Náhodný jev $A$ se nazývá elementární náhodný jev, pokud neexistují různé náhodné jevy $B$, $C$, takové, že $A = B \cup C$.

### 7.3.1 Jevové pole

- Mejmě dvojici $(\Sigma, \Omega)$. $\Omega$ je základní prostor a $\Sigma$ je množina náhodných jevů, resp. systém podmnožin základního prostoru $\Sigma \subseteq 2^{\Omega}$. $\Sigma$ nazýváme jevové pole.
- Platí:
  1. $A \in \Sigma \Rightarrow \bar{A} \in \Sigma$
  2. $A, B \in \Sigma \Rightarrow A \cup B, A \cap B, A \setminus B \in \Sigma$
  3. $\emptyset \in \Sigma,\ \Omega \in \Sigma$
- Dále:
  - $\omega \in \Omega$ je elementární náhodný jev,
  - $\Omega$ je jistý jev,
  - $\emptyset$ je nemožný jev.

### 7.3.2 Kolmogorova axiomatická definice pravděpodobnosti

- Nechť $(\Sigma, \Omega)$ je jevové pole, potom zobrazení $P : \Sigma \to \langle 0, 1 \rangle$ nazveme pravděpodobností na jevovém poli $(\Sigma, \Omega)$, pokud splňuje:
  - nezápornost,
  $$\forall A \in \Sigma \ : \ P(A) \geq 0$$
  - normovanost,
  $$P(\Omega) = 1$$
  - pravděpodobnost sjednocení disjunktních náhodných jevů je stejná jako součet jejich pravděpodobností.
  $$\forall A_1, \ldots, A_n \in \Sigma : A_i \cap A_j = \emptyset \Rightarrow P\left(\bigcup_{i=1}^{n} A_i\right) = \sum_{i=1}^{n} P(A_i)$$
- Pro $A \in \Sigma$ nazýváme hodnotu $P(A)$ pravděpodobností jevu $A$.
- Trojici $(\Omega, \Sigma, P)$ nazýváme pravděpodobnostní prostor.

- Dále platí, nechť $A, B \in \Sigma$:

$$A \cap B \neq \emptyset \Rightarrow P(A \cup B) = P(A) + P(B) - P(A \cap B)$$

### 7.3.3 Klasická (Laplaceova) pravděpodobnost

- Předpokládejme:
  - Základní prostor $\Omega$ je konečná množina.
  - Pro elementární jevy platí:

$$\forall \omega \in \Omega : P(\omega) = \frac{1}{|\Omega|}$$

- Pak pravděpodobnost definujeme jako poměr počtu příznivých jevů ku počtu všech jevů.

$$\forall A \in 2^{\Omega} : P(A) = \frac{|A|}{|\Omega|}$$

**Příklad: házení šestistrannou kostkou**

- Základní prostor: $\Omega = \{1, 2, 3, 4, 5, 6\}$
- Jevové pole: $(2^{\Omega}, \Omega)$.
- Elementární náhodné jevy: $\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}$
- Náhodný jev „sudé číslo“: $A = \{2, 4, 6\}$

$$P(A) = \frac{|A|}{|\Omega|} = \frac{3}{6} = 0,5$$

- Náhodný jev „číslo větší než 4“: $B = \{5, 6\}$

$$P(B) = \frac{|B|}{|\Omega|} = \frac{2}{6} = 0,\bar{3}$$

### 7.3.4 Geometrická pravděpodobnost

- Rozšíření klasické pravděpodobnosti pro nekonečný základní prostor $\Omega$.
- Zde je definice pravděpodobnosti založena na porovnání objemů, ploch či délek geometrických útvarů.
- Uvažujme dvojrozměrný prostor, výpočet pravděpodobnosti pak vypadá následovně:

$$P(A) = \frac{A_S}{\Omega_S}$$

  Kde $A_S$ je obsah plochy reprezentující jev $A$ a $\Omega_S$ je obsah plochy reprezentující všechny možné výsledky.

## 7.4 Podmíněná pravděpodobnost

### 7.4.1 Podmíněná pravděpodobnost

- Nechť $(\Sigma, \Omega)$ je jevové pole, $A, B \in \Sigma$ jsou náhodné jevy a platí $P(B) > 0$.

- Podmíněná pravděpodobnost jevu $A$ vzhledem k jevu $B$ pak je:

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}$$

- Podmíněná pravděpodobnost udává s jakou pravděpodobností nastane jev $A$, když už nastal jev $B$.

### 7.4.2 Věta o úplné pravděpodobnosti

- Nechť $B_i \in 2^{\Omega}$ , $i \in \{1, 2, \ldots, n\}$ je rozklad základního prostoru $\Omega$, platí

$$\bigcup_{i=1}^{n} B_i = \Omega \ \wedge \ i, j \in \{1, 2, \ldots, n\} : B_i \neq B_j \Rightarrow B_i \cap B_j = \emptyset$$

- Nechť $A \in 2^{\Omega}$ je náhodný jev, platí:

$$P(A) = \sum_{i=1}^{n} P(B_i) \cdot P(A \mid B_i)$$

### 7.4.3 Bayesova věta

- Nechť $B_i \in 2^{\Omega}$ , $i \in \{1, 2, \ldots, n\}$ je rozklad základního prostoru $\Omega$, platí

$$\bigcup_{i=1}^{n} B_i = \Omega \ \wedge \ i, j \in \{1, 2, \ldots, n\} : B_i \neq B_j \Rightarrow B_i \cap B_j = \emptyset$$

- Nechť $A \in 2^{\Omega}$ je náhodný jev, o němž víme, že již nastal.

$$P(B_i \mid A) = \frac{P(B_i) \cdot P(A \mid B_i)}{P(A)}$$

- Pravděpodobnost $P(A)$ lze spočítat na základě věty o úplné pravděpodobnosti.

**Příklad: Jak odhadnu pravděpodobnost, že dostanu infarkt, když budu tlustý, pokud mám statistiky pořízené na velkém vzorku populace, kde je u každého člověka záznam o tom, zda byl tlustý a zda prodělal infarkt.**

- Na základě věty o úplně pravděpodobnosti:

$$P(tlusty) = P(infarkt) \cdot P(tlusty \mid infarkt) + P(neInfarkt) \cdot P(tlusty \mid neInfarkt)$$

- Na základě bayesovy věty:

$$P(infarkt \mid tlusty) = \frac{P(infarkt) \cdot P(tlusty \mid infarkt)}{P(tlusty)}$$

**Příklad: V populaci užívá drogy $2\,\%$ lidí. Test na drogy má $3\,\%$ šanci na falešnou pozitivitu $4\,\%$ na falešnou negativitu. Pokud vyjde člověku test pozitivní, jaká je pravděpodobnost, že skutečně bere drogy?**

- Nechť $A$ jsou užívalé drog a $B$ jsou neuživatelé drog, pak platí:

$$P(A) = \frac{2}{100} = 0,02 \qquad , \qquad P(B) = \frac{98}{100} = 0,98$$

- Pozitivní výsledek testu na drogy označme jako $P$ a negativní jako $N$, pak platí:

$$P(P \mid A) = \frac{96}{100} = 0,96 \quad , \quad P(P \mid B) = \frac{3}{100} = 0,03$$

$$P(N \mid A) = \frac{4}{100} = 0,04 \quad , \quad P(N \mid B) = \frac{97}{100} = 0,97$$

- Na základě věty o úplně pravděpodobnosti:

$$P(P) = P(A) \cdot P(P \mid A) + P(B) \cdot P(P \mid B) = 0,02 \cdot 0,96 + 0,98 \cdot 0,03 = 0,0486$$

- Na základě bayesovy věty:

$$P(A \mid P) = \frac{P(A) \cdot P(P \mid A)}{P(P)} = \frac{0,02 \cdot 0,96}{0,0486} \approx 39,51\,\% \tag{7.6}$$

### 7.4.4 Nezávislost

- Řekneme, že jevy $A$ a $B$ jsou nezávislé, pokud situace, kdy nastal jev $A$, neovlivní pravděpodobnost, že nastane jev $B$.
- Formálně: nechť $A, B \in 2^{\Omega}$ jsou náhodné jevy, tyto jevy nazveme nezávislé, pokud platí:

$$P(A \mid B) = P(A) \ \vee \ P(B \mid A) = P(B)$$

- Lze upravit:

$$P(A \mid B) = P(A)$$

$$\frac{P(A \cap B)}{P(B)} = P(A)$$

$$P(A \cap B) = P(A) \cdot P(B)$$

- A tedy platí:

$$A, B \in 2^{\Omega} : A, B \text{ jsou nezávislé} \Leftrightarrow P(A \cap B) = P(A) \cdot P(B)$$

- Pozn. nezávislé jevy a disjunktní jevy jsou něco jiného.

**Příklad: Dva hody kostkou a (ne)závislé jevy.**

- **Zadání**
  - Nechť $\Omega$ je základní prostor dvou hodů kostkou, pak $|\Omega| = 36$.
  - Nechť $A \in 2^{\Omega}$ je jev, kdy první hod padla 4.

$$A = \{(4,1), (4,2), \ldots, (4,6)\}$$

  - Nechť $B \in 2^{\Omega}$ je jev, kdy druhý hod padla 2.

$$B = \{(1,2), (2,2), \ldots, (6,2)\}$$

  - Nechť $C \in 2^{\Omega}$ je jev, kdy součet hodů je 6.

$$C = \{(1,5), (2,4), (3,3), (4,2), (5,1)\}$$

- **a) Jsou jevy A a B nezávislé?**

$$P(A) = \frac{1}{6}\ ,\ P(B) = \frac{1}{6}$$

$$P(A) \cdot P(B) = \frac{1}{36}$$

$$P(A \cap B) = \frac{|\{(4,2)\}|}{36} = \frac{1}{36}$$

$$P(A \cap B) = P(A) \cdot P(B) \ \Rightarrow \ \text{jevy A, B jsou nezávislé}$$

- **b) Jsou jevy A a C nezávislé?**

$$P(A) = \frac{1}{6}\ ,\ P(C) = \frac{5}{36}$$

$$P(A) \cdot P(B) = \frac{5}{36}$$

$$P(A \cap B) = \frac{|\{(4,2)\}|}{36} = \frac{1}{36}$$

$$P(A \cap B) \neq P(A) \cdot P(B) \ \Rightarrow \ \text{jevy A, B jsou závislé}$$

### 7.4.5 Sum Rule

- *Učilo se v SUI, platí pro nezávislé jevy.*
- Nechť $\Omega$ je základní prostor a $A \in 2^{\Omega}$ je náhodný jev, pak platí tzv. *sum rule*:

$$P(A) = \sum_{B \in 2^{\Omega}} P(A) \cdot P(B)$$

### 7.4.6 Product rule

- *Učilo se v SUI, platí pro nezávislé jevy.*
- Nechť $\Omega$ je základní prostor a $A, B \in 2^{\Omega}$ jsou náhodné jevy, pak platí tzv. *product rule*:

$$P(A, B) = P(A \mid B) \cdot P(B) = P(B \mid A) \cdot P(A)$$

- Konvence pro zároveň:

$$P(A, B) = P(A) \cdot P(B)$$

# Kapitola 8

# MSP – Náhodná proměnná, typy náhodné proměnná, funkční a číselné charakteristiky, významná rozdělení pravděpodobnosti.

## 8.1 Zdroje

- `MSP_pred_01_Opakovani_Pravd-NP-NV.pdf`
- `MSP_pred_02_Opakovani_Statistika_Regrese.pdf`
- Wikipedia

## 8.2 Náhodná proměnná

- Náhodná proměnná (také náhodná veličina) je funkce, reprezentuje nějaký náhodný proces. Přiřazuje každému elementárnímu náhodnému jevu nějakou (zpravidla číselnou) hodnotu.
  - Příklad náhodné proměnné reprezentující náhodný proces hod mincí:

    $$X = \begin{cases} 1 & \text{pokud padne „hlava“} \\ 0 & \text{pokud padne „orel“} \end{cases}$$

    Zkoumání pravděpodobnosti:

    $$P(X = 1) = \dots$$

- **Formálně** – Nechť $\Omega$ je základní prostor a $(\Sigma, \Omega)$ je jevové pole. Pak zobrazení $X : \Omega \to \mathbb{R}$ se nazývá náhodná proměnná, pokud je měřitelné, tj.

  $$\forall x \in \mathbb{R} : \{\omega \in \Omega \mid X(\omega) < x\} \in \Sigma$$

- Realizaci náhodné veličiny, tj. $X(\omega),\ \omega \in \Omega$ označíme $x$, pak
  - množinu $\{\omega \in \Omega \mid X(\omega) < x\}$ zapisujeme jako $\{X < x\}$,
  - množinu $\{\omega \in \Omega \mid X(\omega) = x\}$ zapisujeme jako $\{X = x\}$.

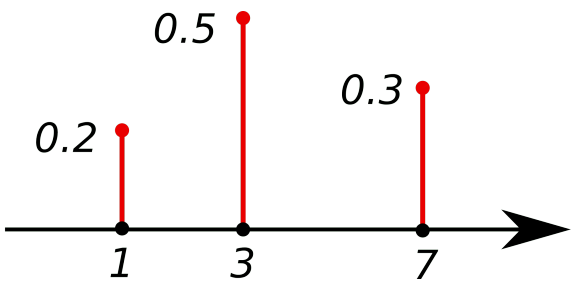
- Obor hodnot náhodné proměnné značíme $Z$.
  - Obor hodnot náhodné proměnné je definiční obor pravděpodobnostní funkce, resp. funkce hustoty pravděpodobnostni (viz dále).

### 8.2.1 Distribuční funkce

- Hodnota $P(\{\omega \in \Omega \mid X(\omega) < x\})$ se nazývá distribuční funkce náhodné veličiny $X$ a značí se $F(x)$.
- Zkráceně zapisujeme jako $F(x) = P(X < x)\ ,\ \forall x \in \mathbb{R}$.
- Distribuční funkce udává, že hodnota náhodné proměnné je menší než zadaná hodnota.
- Vlastnosti:
  - DF je zprava spojitá,
  $$\lim_{x \to \alpha^+} F(x) = F(\alpha)$$
  - DF je neklesající,
  $$\alpha < \beta \Rightarrow F(\alpha) \leq F(\beta)$$
  - asymptotické vlastnosti:
  $$\lim_{x \to -\infty} F(x) = 0$$
  $$\lim_{x \to +\infty} F(x) = 1$$

## 8.3 Diskrétní náhodná proměnná

- Náhodná proměnná se nazývá **diskrétní**, pokud obor hodnot $Z \subset \mathbb{R}$ je konečná nebo nejvýše spočetně nekonečná množina.
- **Pravděpodobnostní funkce**
  - Udává pravděpodobnost, že diskrétní náhodná veličina se přesně rovná nějaké hodnotě.
  - Funkci $p(x) = P(X = x)\ ,\ x \in \mathbb{R}$ nazýváme pravděpodobností funkcí diskrétní náhodné veličiny $X$.
  - Platí:
  $$\sum_{x \in Z} p(x) = 1$$
  $$\forall x \in Z\ :\ 0 \leq p(x) \leq 1$$



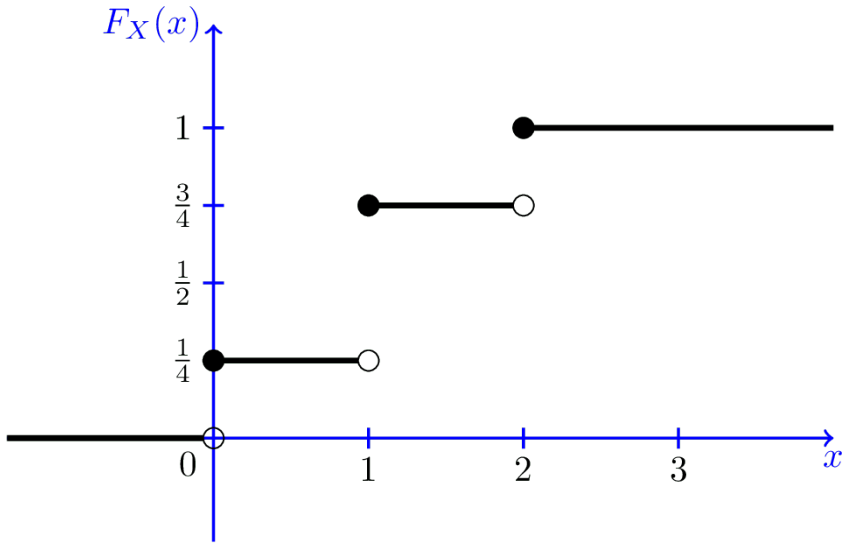

Obrázek 8.1: Příklad pravděpodobnostní funkce pro DNP.

- **Distribuční funkce**
  - Distribuční funkce má tvar:

$$F(x) = \sum_{t<x} p(t) \ , \ \forall x \in \mathbb{R}$$

  - Platí:

$$p(x) = \lim_{t \to x^+} F(t) - F(x)$$




Obrázek 8.2: Příklad distribuční funkce pro DNP.

- **Příklad** – náhodná proměnná reprezentující součet hodnot po 5 hodech šestistrannou kostkou.

$$X = \text{suma 5 hodů šestistrannou kostkou}$$

  - Zkoumání pravděpodobnosti, že součet bude větší než 15:

$$P(X > 15) = \dots$$

## 8.4 Spojitá náhodná proměnná

- Náhodná proměnná se nazývá **spojitá**, pokud obor hodnot $Z \subseteq \mathbb{R}$ je nekonečná nespočetná množina a existuje nezáporná, po částech spojitá funkce $f(x)$, taková, že

$$F(x) = \int_{-\infty}^{x} f(t) \ dt \ , \ x \in \mathbb{R}$$

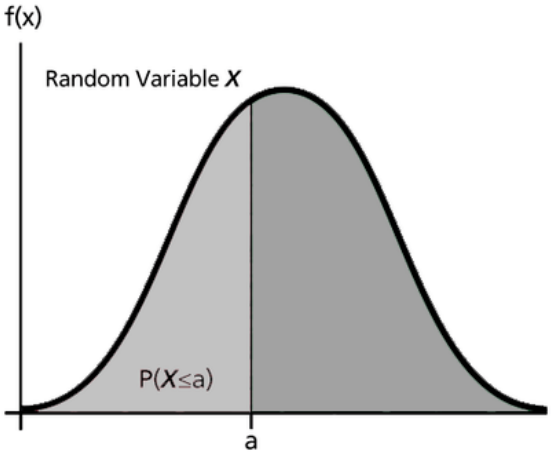
- Pravděpodobnost konkrétní hodnoty je 0, formálně:

$$\forall c \in \mathbb{R} : P(X = c) = 0$$

- **Funkce fustoty pravděpodobnosti**
  - Funkci $f : \mathbb{R} \to \mathbb{R}^+$ nazýváme funkci hustoty pravděpodobnosti náhodné veličiny $X$.

– Platí:

$$\int_{-\infty}^{+\infty} f(x)\ dx = 1$$




Obrázek 8.3: Příklad funkce hustoty pravděpodobnosti pro SNP.

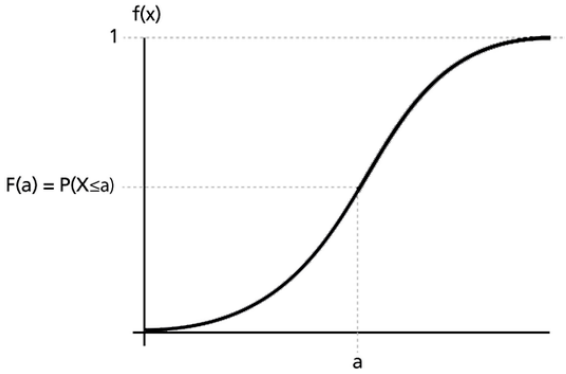
- **Distribuční funkce**
  – Má tvar:

$$F(x) = \int_{-\infty}^{x} f(t)\ dt\ ,\ x \in \mathbb{R}$$

  – Platí, nechť $a, b \in \mathbb{R}$:

$$P(x < a) = P(x \leq a) = F(a)$$

$$P(a < x < b) = P(a \leq x \leq b) = \int_{a}^{b} f(x)\ dx = F(b) - F(a)$$




Obrázek 8.4: Příklad distribuční funkce pro SNP.

- **Příklad** – náhodná proměnná reprezentující vítězný čas závodu v běhu na 100 m.

$$X = \text{vítězný čas v běhu na } 100\,\text{m}$$

  – Zkoumání pravděpodobnosti, že vítězný čas je pod 10:

$$P(X < 10) = \ldots$$

## 8.5 Číselné charakteristiky náhodné proměnné

- Dělíme na:
  - charakteristiky polohy – střední hodnota, medián, modus;
  - charakteristiky variability – rozptyl, směrodatná odchylka.

- **Medián** – Nechť $p \in \langle 0, 1 \rangle$ je kvantil náhodné proměnné. Pokud $p = 0,5$, tak se $p$ kvantil nazývá medián a značí se $\tilde{x}$.
  - Kvantil je charakteristika, kterou stanovená část $p$ (uváděná jako číslo z intervalu $\langle 0, 1 \rangle$) hodnot nepřesahuje. Je také možné říct, že kvantily jsou hodnoty, které dělí soubor seřazených (například naměřených) hodnot na několik zhruba stejně velkých částí. (Příklad: výrok, že 90 % účastníků závodu mělo čas pod 2 hodiny, vlastně konstatuje, že 90. percentil dosažených časů je 2 hodiny.)

- **Modus** – Modus náhodné veličiny $X$ je reálné číslo $\hat{x}$, které je maximem pravděpodobnostní funkce, resp. funkce hustoty pravděpodobnosti.

- **Střední hodnota** – Střední hodnota (také očekávaná hodnota) diskrétní náhodné proměnné je pravděpodobnostně vážený průměr všech jejích možných hodnot, pro spojitou náhodnou proměnnou je součet nahrazen integrálem proměnné vzhledem k její hustotě pravděpodobnosti.
  - Značí se:
  $$E(X) \ , \ \mu(X)$$
  - Pro DNP:
  $$E(X) = \sum_{i=1}^{n} x_i \cdot p(x_i)$$
  - Pro SNP:
  $$E(X) = \int_{-\infty}^{+\infty} x \cdot f(x) \ dx$$

- **Rozptyl** – Rozptyl (také střední kvadratická odchylka, variance) je druhý centrální moment náhodné veličiny. Jedná se o charakteristiku variability rozdělení pravděpodobnosti náhodné veličiny, která vyjadřuje variabilitu rozdělení souboru náhodných hodnot kolem její střední hodnoty.
  - Značí se:
  $$D(X) \ , \ S^2(X) \ , \ \sigma^2(X)$$
  - Pro DNP:
  $$D(X) = \sum_{i=1}^{n} (x_i - E(X))^2 \cdot p_i$$
  - Pro SNP:
  $$D(X) = \int_{-\infty}^{\infty} x^2 \cdot p(x) \ dx - E^2(X)$$

- **Směrodatná odchylka** – Je odmocnina rozptylu náhodné veličiny.
  - Značí se:
  $$S(X) \ , \ \sigma(X)$$

– Pro DNP, SNP:

$$\sigma(X) = \sqrt{D(X)}$$

- **Šikmost, špičatost, čebyšova nerovnost**
  - *Nepředpokládám, že by se zkoušelo.*
- **Kovariance, korelace** – Charakteristiky pro popis vzájemného vztahu dvou veličin.
  - *Nepředpokládám, že by se zkoušelo.*

## 8.6 Vybraná rozdělení diskrétní náhodné proměnné

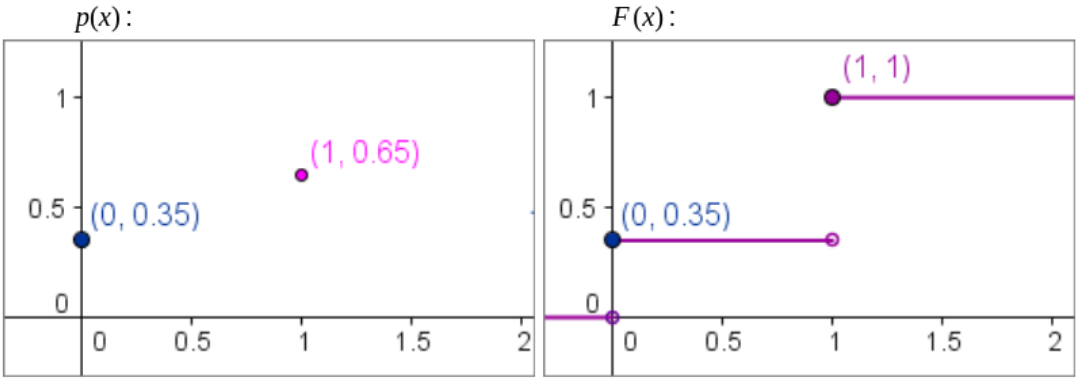
### 8.6.1 Alternativní (Bernoulliho) rozdělení

- Dvě alternativy (mohou být různě pravděpodobné).
- Parametr $p$ udává pravděpodobnost jevu.

$$X \sim A(p)\ ,\ p \in \langle 0, 1 \rangle$$

$$Z = \{0, 1\}$$

$$p(x) = \begin{cases} 1 - p & x = 0 \\ p & x = 1 \end{cases}\ ,\ x \in Z$$




Obrázek 8.5: Alternativní rozdělení – příklad pravděpodobnostní a distribuční funkce.

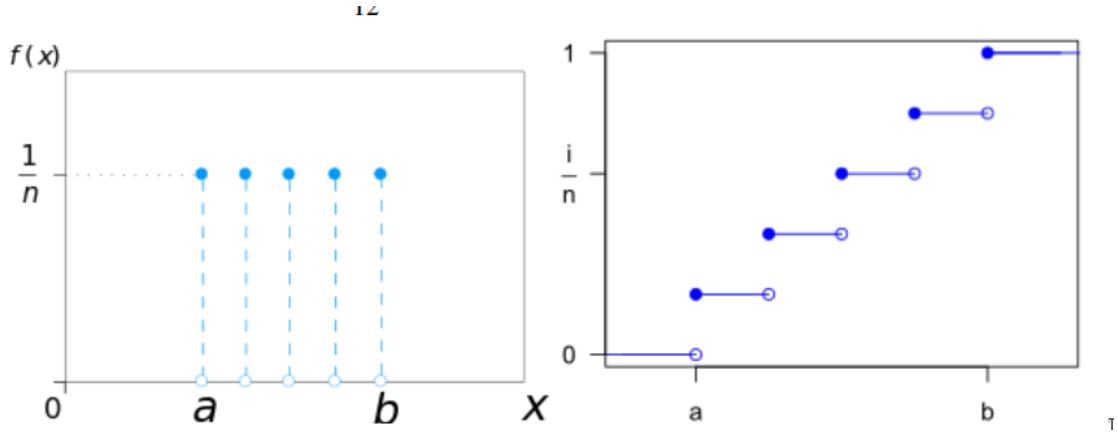
### 8.6.2 Klasické rozdělení

- Všechny náhodné jevy jsou stejně pravděpodobné.
- Parametr $n$ udává počet alternativ.

$$X \sim C(n)\ ,\ n \in \mathbb{N}$$

$$Z = \{0, 1, \ldots, n\}$$

$$p(x) = \frac{1}{n}\ ,\ x \in Z$$




Obrázek 8.6: Klasické rozdělení – příklad pravděpodobnostní a distribuční funkce.

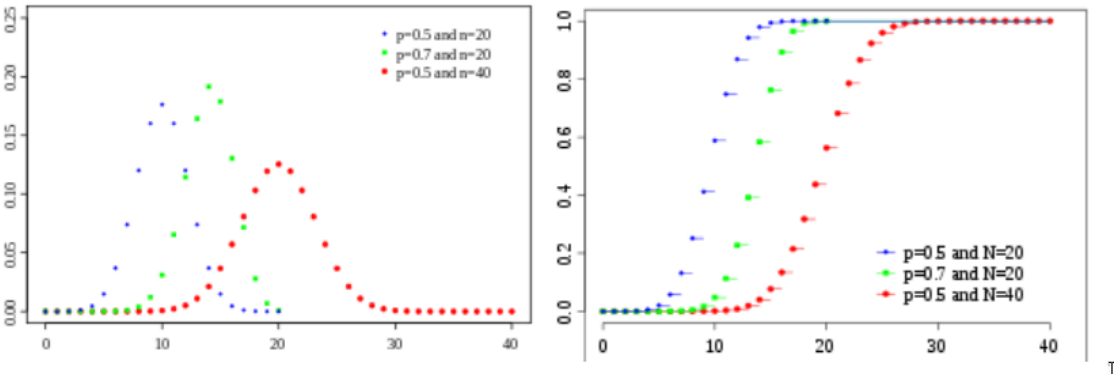
### 8.6.3 Binomické rozdělení

- Popisuje četnost výskytu náhodného jevu v $n$ nezávislých pokusech, v nichž má jev stále stejnou pravděpodobnost. Pokud speciálně $n = 1$, jde o alternativní rozdělení.
- Parametr $n$ počet opakování a parametr $p$ pravděpodobnost výskytu jevu.
- Například, jaká je pravděpodobnost, že při 5 vrzích kostkou padne právě 2× číslo 1?

$$X \sim Bi(n, p) \ , \ n \in \mathbb{N} \ , \ p \in \langle 0, 1 \rangle$$

$$Z = \{0, 1, \ldots, n\}$$

$$p(x) = \binom{n}{x} \cdot p^x \cdot (1-p)^{n-x} \ , \ x \in Z$$




Obrázek 8.7: Binomické rozdělení – příklad pravděpodobnostní a distribuční funkce.

### 8.6.4 Poissonovo rozdělení

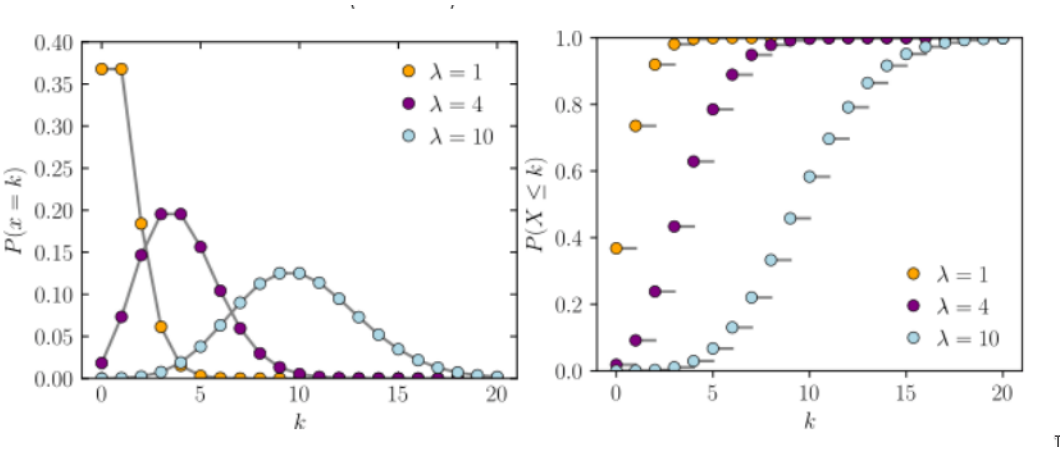
- Popisuje náhodnou veličinu, která vyjadřuje počet výskytů jevů v určitém intervalu (času, délky, objemu), když jevy nastávají nezávisle na sobě.
- Parametr $\lambda$ udává počet výskytů v intervalu (průměr).
- Například, občas nám přijde dopis (to je náš jev, událost). Během roku dostaneme 1460 dopisů, t.j. v průměru 4 za den. Počet příchozích dopisů během jednoho dne (to

je náš časový interval) se řídí Poissonovým rozdělením. Nejvyšší je pravděpodobnost, že přijdou 4 dopisy. Pravděpodobnost dvou dopisů je o něco menší. Pravděpodobnost, že jich přijde 100, je téměř nulová.

$$X \sim Po(\lambda) \ , \ \lambda \in \mathbb{R}^+$$

$$Z = \{0, 1, \ldots\}$$

$$p(x) = e^{-\lambda} \cdot \frac{\lambda^x}{x!} \ , \ x \in Z$$




Obrázek 8.8: Poissonovo rozdělení – příklad pravděpodobnostní a distribuční funkce.

## 8.7 Vybraná rozdělení spojité náhodné proměnné

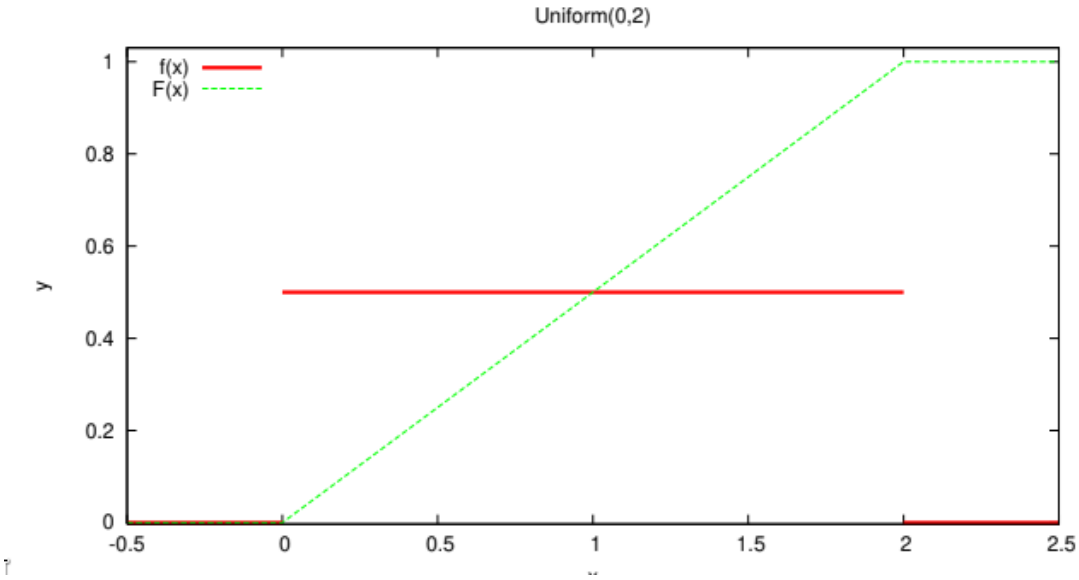
### 8.7.1 Rovnoměrné rozdělení

- Všechny náhodné jevy jsou stejně pravděpodobné (stejné jako klasické rozdělení, akorát pro SNP).
- Parametry $a, b$ udávají dolní, resp. horní hranici intervalu.

$$X \sim R(a, b) \ , \ a, b \in \mathbb{R} \wedge a < b$$

$$Z = \mathbb{R}$$

$$f(x) = \begin{cases} \frac{1}{b-a} & x \in \langle a, b \rangle \\ 0 & \text{jinak} \end{cases} \ , \ x \in Z$$




Obrázek 8.9: Rovnoměrné rozdělení – příklad funkce hustoty pravděpodobnosti a distribuční funkce.

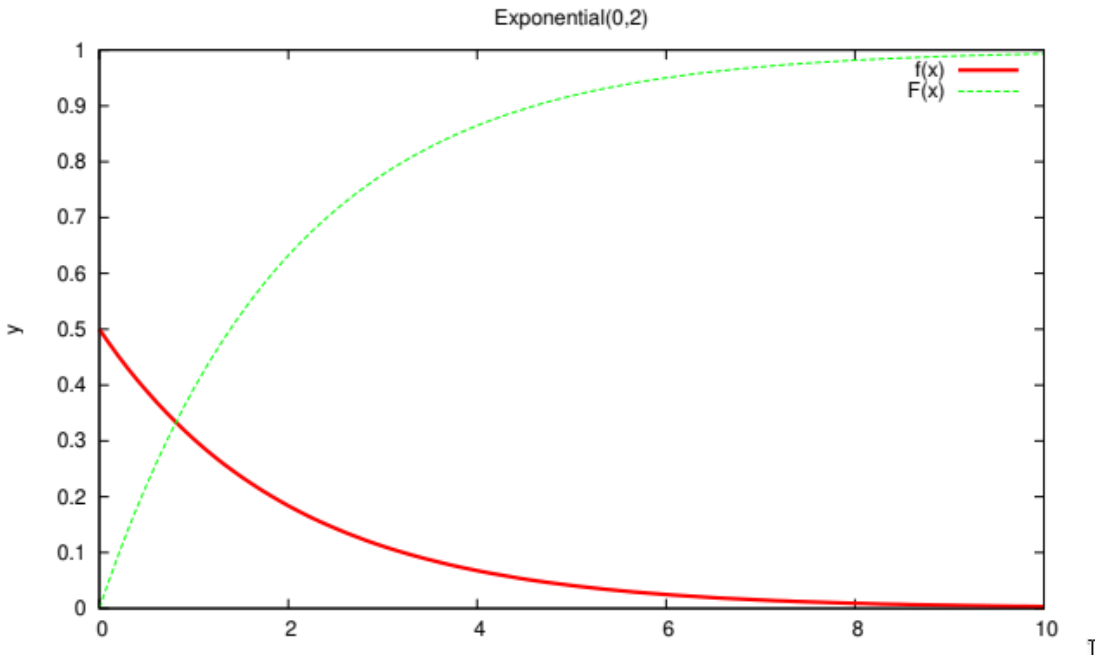
### 8.7.2 Exponenciální rozdělení

- Vyjadřuje rozdělení délky intervalu mezi náhodně se vyskytujícími událostmi, jejichž pravděpodobnost výskytu má Poissonovo rozdělení.
- Parametr $\lambda$ udává počet výskytů v intervalu (průměr).
- Například, využívá se v pojistné matematice při určování (pravděpodobnostního) rozdělení výše pojistného plnění nebo času mezi nastalé pojistné události, dále ve fyzice při modelování času radioaktivního rozpadu a v systémech hromadné obsluhy.

$$X \sim Ex(a, \lambda) \ , \ a, \lambda \in \mathbb{R} \wedge \lambda > 0$$

$$Z = \langle a, +\infty \rangle$$

$$f(x) = \begin{cases} \lambda \cdot e^{-\lambda(x-a)} & x > a \\ 0 & \text{jinak} \end{cases} \ , \ x \in Z$$




Obrázek 8.10: Exponenciální rozdělení – příklad funkce hustoty pravděpodobnosti a distribuční funkce.

### 8.7.3 Normální (Gaussovo) rozdělení

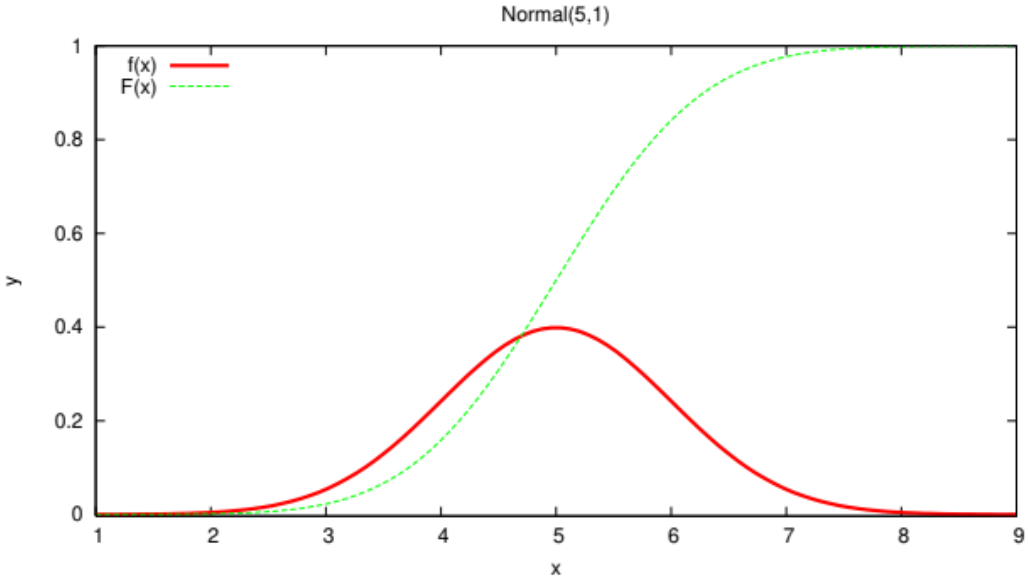
- Jeho důležitost ukazuje centrální limitní věta (CLV), jež zhruba řečeno tvrdí, že součet či aritmetický průměr velkého počtu libovolných vzájemně nezávislých a nepříliš „divokých“ náhodných veličin se vždy podobá normálně rozdělené náhodné veličině. Normální rozdělení proto za určitých podmínek dobře aproximuje řadu jiných pravděpodobnostních rozdělení (spojitých i diskrétních).
- Parametr $\mu$ udává střední hodnotu, parametr $\sigma^2$ rozptyl.
- Například, náhodné chyby (chyby měření, ...) způsobené velkým počtem malých, neznámých a vzájemně nezávislých příčin, jsou v důsledku CLV rovněž rozděleny přibližně normálně. Proto bývá normální rozdělení také označováno jako zákon chyb. Podle tohoto zákona se také teoreticky řídí rozdělení některých fyzikálních a technických veličin.

$$X \sim N(\mu, \sigma^2)\ ,\ \mu, \sigma^2 \in \mathbb{R}$$

$$Z = \mathbb{R}$$

$$f(x) = \frac{1}{\sigma \cdot \sqrt{2\pi}} \cdot e^{-\frac{(x-\mu)^2}{2\sigma^2}}\ ,\ x \in Z$$

- **Centrální limitní věta** označuje tvrzení, podle něhož se (za určitých podmínek) rozdělení výběrového průměru blíží k normálnímu rozdělení, a to bez ohledu na to, jaké je rozdělení průměrované náhodné veličiny. Jinak řečeno pokud platí předpoklady centrální limitní věty, tak výběrový průměr má jakožto náhodná veličina asymptoticky normální rozdělení.




Obrázek 8.11: Normální rozdělení – příklad funkce hustoty pravděpodobnosti a distribuční funkce.

# Kapitola 9

# MSP – Bodové a intervalové odhady parametrů, testování hypotéz o parametrech.

## 9.1 Zdroje

- `MSP_pred_02_Opakovani_Statistika_Regrese`.pdf
- `MSP_pred_03_Norm-Bi_Odhady-Testy.pdf`
- Wikipedia

## 9.2 Úvod a kontext

- **Náhodný výběr** – Z celého stavového prostoru jsou náhodně vybrány vzorky (je proveden náhodný výběr).

  Např. princip volebních průzkumů.

- Na statistický soubor $(x_1, x_2, \ldots, x_n)$ můžeme nahlížet jako na výběrový soubor získaný náhodným výběrem z náhodné proměnné $X$.
  - Stejným způsobem pro vícerozměrné statistické soubory. Soubor $((x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n))$ může být získán náhodným výběrem ze dvou náhodných proměnných $X, Y$.
- Cílem je, na základě statistického souboru $(x_1, x_2, \ldots, x_n)$ odhadnout parametry náhodné proměnné $X$.
  - Typicky jde o číselné charakteristiky jako jsou střední hodnota a rozptyl.
- Jako **statistickou hypotézu** chápeme určitý předpoklad o rozdělení náhodných veličin. Jestliže se tyto předpoklady týkají hodnot parametrů rozdělení náhodné veličiny, pak hovoříme o parametrických hypotézách. V opačném případě se jedná o hypotézy neparametrické.
- **Parametrické testy** – Předpokládají nějaké rozdělení pravděpodobnosti, hypotéza je o parametrech rozdělení základního souboru.
  - Hypotézy o parametru jednoho základního souboru – střední hodnota, mediána, rozptyl, ...

  - Hypotézy o parametrech dvou základních souborů (srovnávací testy) – rovnost středních hodnot, rovnost rozptylů, ...

- **Neparametrické testy** – Nepředpokládají žádné rozdělení pravděpodobnosti, hypotéza je o jiných vlastnostech základního souboru.
  - Hypotéza o tvaru rozdělení, závislosti proměnných, ...

## 9.3 Bodový odhad

- Bodový odhad aproximuje hledaný parametr jednou číselnou hodnotou (jde o nejlepší odhad).
  - To se hodí zejména pokud je parametr potřeba pro další výpočty.
- Formálně; nechť $(X_1, X_2, \ldots, X_n)$ je náhodný výběr z rozdělení s distribuční funkcí $F(x, \theta)$. Statistika $T(X_1, X_2, \ldots, X_n)$ se nazývá bodovým odhadem parametru $\theta$, pokud nabývá hodnot blízkých parametru $\theta$.
- Vlasnoti:
  - Bodový odhad T se nazývá nestranný (nevychýlený), pokud platí:

$$E(T) = \theta$$

  - Bodový odhad T se nazývá stranný (vychýlený), pokud platí:

$$E(T) \neq \theta$$

  - Bodový odhad T se nazývá konzistentní, pokud platí:

$$\lim_{n \to \infty} P(|T(X_1, X_2, \ldots, X_n) - \theta| < \epsilon) = 1$$

- Máme metody pro bodové odhady parametrů pro jednotlivá pravděpodobnostní rozdělení (Normální, Binomické, ...).

## 9.4 Intervalový odhad

- Intervalový odhad aproximuje hledaný parametr intervalem. Tj. hledaný parametr se s předem stanovenou spolehlivostí nachází uvnitř výsledného intervalu.
  - To se hodí, pokud potřebujeme znát přesnost odhadu parametru.
  - Bodový odhad je intervalový odhad se spolehlivostí 0.
- Formálně; nechť $X$ je náhodná proměnná, která má distribuční funkci $F(x, \theta)$. Interval spolehlivosti pro parametr $\theta$ na hranici spolehlivosti $1 - \alpha$ , $\alpha \in \langle 0, 1 \rangle$ je dvojice statistik $T_1$, $T_2$, pro které platí:

$$P(T_1 \leq \theta \leq T_2) = 1 - \alpha$$

  Intervalový odhad parametru $\theta$ se spolehlivostí $1 - \alpha$ , $\alpha \in \langle 0, 1 \rangle$ je interval $\langle t_1, t_2 \rangle$, kde $t_1$, resp. $t_2$ je realizací statistiky $T_1$, resp. $T_2$.
- Požadavky na intervalový odhad:
  - aby pravděpodobnost $1 - \alpha$ byla co největší;

- interval byl co nejmenší.
- Typicky volíme $\alpha \in \{0,1, 0,05, 0,01\}$.

- Máme metody pro intervalové odhady parametrů pro jednotlivá pravděpodobnostní rozdělení (Normální, Binomické, ...).

## 9.5 Testování hypotéz o parametrech

- Testování statistických hypotéz umožňuje posoudit, zda experimentálně získaná data vyhovují předpokladu, který jsme před provedením testování učinili. Můžeme například posuzovat, zda platí předpoklad, že určitý lék je účinnější než jiný; nebo zda platí, že úroveň matematických dovedností žáků 9. tříd je nezávislá na pohlaví a na regionu.
- **Statistická hypotéza** $H$ je tvrzení o vlastnostech pravděpodobnostního rozdělení zkoumané náhodné proměnné $X$ s distribuční funkcí $F(x, \theta)$.
  - Postup kterým hypotézu ověřujeme, se nazývá test statistické hypotézy.
- Jako alternativu vůči hypotéze postavíme tzv. **alternativní hypotézu** $H_A$ (také nulová hypotéza), kterou volíme v kontextu dané úlohy.
  - $H_0 : \theta = \theta_0$ je hypotéza, že parametr $\theta$ má hodnotu $\theta_0$
  - $H_A : \theta > \theta_0$ je jednostranná alternativní hypotéza.
  - $H_A : \theta \neq \theta_0$ je oboustranná alternativní hypotéza.
- Pro testování hypotézy $H$ proti nějaké zvolené alternativní hypotéze $H_A$ se konstruuje vhodná statistika $T(X_1, X_2, \ldots, X_n)$, tzv. **testové kritérium**.
- Při hledání testového kritéria $T$ se vychází z požadavků na zamítnutí hypotézy $H$. Zajímá nás, za jakých podmínek lze hypotézu zamítnout. K tomu se konstruuje množina možných hodnot realizace statistiky $T$. Tato množina se nazývá kritický obor a označuje se $W_\alpha$. Velikost této množiny závisí na spolehlivosti našeho tvrzení (parametr $\alpha$ – hladina významnosti).
  - Pokud realizace zvolené statistiky $t = T(x_1, x_2, \ldots, x_n)$ padne do kritického oboru $W_\alpha (t \in W_\alpha)$ říkáme, že **hypotézu zamítáme** na hladině významnosti $\alpha$.
- U většiny testů se místo kritického oboru udává doplněk kritického oboru $\overline{W}_\alpha$.
  - $\overline{W}_\alpha = \mathbb{R} \setminus W_\alpha$.
  - Pokud realizace zvolené statistiky $T$ padne do doplňku kritického oboru $t \in \overline{W}_\alpha$ říkáme, že **hypotézu nezamítáme** na hladině významnosti $\alpha$.
- Nezamítnutí hypotézy $H$, resp. $H_A$, neznamená ještě prokázání její platnosti, neboť jsme na základě realizace náhodného výběru získali pouze informace, které nestačí na její zamítnutí. Je-li to možné, je vhodné před přijetím dané hypotézy zvětšit rozsah statistického souboru a znovu hypotézu $H$ testovat.

| $H$ | PLATÍ | NEPLATÍ |
|---|---|---|
| ZAMÍTÁME | CHYBA 1. DRUHU ($\alpha$) | — |
| NEZAMÍTÁME | — | CHYBA 2. DRUHU ($\beta$) |

Obrázek 9.1: Testování hypotézy.

- Chybu 1. druhu ($\alpha$) si volíme, chyba 2. druhu závisí na typu testu a na parametrech. Hodnota $1 - \beta$ se nazývá síla testu.
- Pro konkrétní rozdělení budeme u testů uvádět realizaci testovacího kritéria $t = T(x_1, x_2, \ldots, x_n)$ a doplněk kritického oboru $\overline{W}_\alpha$ vzhledem k příslušné alternativní hypotéze.
- Existují různé druhy testů pro různé typy hypotéz v závislosti na rozdělení.

## 9.6 Základní testy hypotéz pro vybraná rozdělení

### 9.6.1 Jeden výběr z Normálního rozdělení

- Máme realizi náhodného výběru $(x_1, x_2, \ldots, x_n)$ pro náhodnou proměnnou $X$ a předpokládáme, že $X \sim N(\mu, \sigma^2)$.
- Testujeme zda daná realizace náhodného výběru odpovídá danému normálnímu rozdělení.
- Testujeme:
  - střední hodnotu, např. Studentův jednovýběrový test;
  $$H_0 : \mu = \mu_0$$
  - rozptyl, např. Test na rozptyl.
  $$H_0 : \sigma^2 = \sigma_0^2$$
- Pro výběr z 2-rozměrného Normálního rozdělení jsou jiné testy (např. Studentův párový test).

### 9.6.2 Dva výběry z Normálního rozdělení

- Máme realizi náhodného výběru $(x_1, x_2, \ldots, x_n)$ pro náhodnou proměnnou $X$ a $(y_1, y_2, \ldots, y_m)$ pro náhodnou proměnnou $Y$. Předpokládáme, že $X \sim N(\mu_X, \sigma_X^2)$ a $Y \sim N(\mu_Y, \sigma_Y^2)$.
- Testujeme zda realizace náhodného výběru $(x_1, x_2, \ldots, x_n)$ a $(y_1, y_2, \ldots, y_n)$ odpovídá stejnému normálnímu rozdělení.
- Testujeme:
  - rovnost rozptylů, např. F-test;
  $$H_0 : \sigma_X^2 = \sigma_Y^2$$
  - střední hodnotu, např. Studentův dvouvýběrový test za podmínky $\sigma_X^2 = \sigma_Y^2$;
  $$H_0 : \mu_X - \mu_Y = \mu_0$$

### 9.6.3 Jeden výběr z Binomického rozdělení

- Předpokládáme, že $X \sim Bi(1, p)$, neznámý parametr je $p$, provedeme $n$ měření / pokusů a získáme realizi náhodného výběru $(x_1, x_2, \ldots, x_n)$ pro náhodnou proměnnou $X$.
- Testujeme zda realizace náhodného výběru $(x_1, x_2, \ldots, x_n)$ odpovídá danému binomickému rozdělení.
- Testujeme:
  - pravděpodobnost $p$

$$H_0 : p = p_0$$

### 9.6.4 Dva výběry z Binomického rozdělení

- Předpokládáme, že $X \sim Bi(1, p_X)$ a $Y \sim Bi(1, p_Y)$. Z každého provedeme několik měření / pokusů. Získáme realizaci náhodných výběrů $X : (x_1, x_2, \ldots, x_n)$ a $Y : (y_1, y_2, \ldots, y_m)$.
- Testujeme zda realizace náhodného výběru $X : (x_1, x_2, \ldots, x_n)$ a $Y : (y_1, y_2, \ldots, y_m)$ odpovídá stejnému binomickému rozdělení.
- Testujeme:
  - shodnost pravděpodobností $p_X$ a $p_Y$:

$$H_0 : p_X = p_Y$$

# Kapitola 10

# MSP – Vícevýběrové testy, testy o rozdělení, testy dobré shody.

## 10.1 Zdroje

- `MSP_pred_02_Opakovani_Statistika_Regrese`.pdf
- `MSP_pred_04_ANOVA.pdf`
- `MSP_pred_08_Testy_DS_Testy_rozdeleni.pdf`
- Wikipedia

## 10.2 ANOVA

- ANOVA (*analysis of variance*, analýza rozptylu) jsou statistické metody, které umožňují provádět vícenásobné porovnávání středních hodnot (resp. rozptylů).
- V čem je rozdíl oproti základním testům hypotéz pro vybraná rozdělení?
  - ANOVA jsou souhrnné testy pro více něž dva výběry (proto spadá pod vícevýběrové testy).
  - Proč neudělat více dvouvýběrových testů? Museli bychom v případě více jak 2 hodnot faktoru provést dvouvýběrový test pro všechny dvojice hodnot, nebo současně porovnat naměřené hodnoty s předem danou hodnotou (efektivita).
- ANOVA je založena na hodnocení vztahů mezi rozptyly porovnávaných výběrových souborů (testování shody středních hodnot se převádí na testování shody dvou rozptylů).
- **Faktor** – Statistický znak (znaky), který ovlivňuje měřenou veličinu. U každého faktoru uvažujeme o konečném počtu jeho hodnot.
  - Např. chov králíků, zajímá nás velikost (měřená veličina) v závislosti na typu krmiva (faktor).
  - Např. Dva termíny pro písemku ze cvičení MSP. Zajímá nás počet bodů z písemky (měřená veličina), v závislosti na skupině (jeden faktor) a na termínu (druhý faktor).

### 10.2.1 Postup

**1. Test středních hodnot**

- Testování hypotézy:
$$H : \mu_1 = \mu_2 = \ldots = \mu_n$$
- Dílčí kroky:
  - variabilita mezi skupinami,
  - celkovou variabilita,
  - součet variability uvnitř skupin.

**2.) Post host analýza**

- Pokud analýza rozptylu zamítne nulovou hypotézu $H : \mu_1 = \mu_2 = \ldots = \mu_n$ o vlivu působícího faktoru, je nutno doplnit rozbor ještě dalšími metodami následného zkoumání existujících rozdílů. Tyto tzv. multikomparativní testy (testy pro mnohonásobné porovnávání) pak dávají výsledkem statistickou významnost jednotlivých rozdílů středních hodnot u všech možných párů porovnávaných skupin.
- Obvykle testujeme tzv. kontrasty, tj. hledáme dvojice $A_i$ a $A_j$, které vliv třídícího znaku způsobují.
- Pro každou dvojici testujeme hypotézu $H : \alpha_i = \alpha_j$ vzhledem k $H : \alpha_i \neq \alpha_j$.
- Testy:
  - Scheffeho metoda (F-test)

**3.) Test rovnosti rozptylů**

- Test rovnosti rozptylů (test homoskedasticity).
- Testujeme hypotézu
$$H : \sigma_1^2 = \sigma_2^2 = \ldots = \sigma_n^2$$
- Proti alternativní
$$H_A : \exists i, j : \sigma_i^2 \neq \sigma_j^2$$
- Testy:
  - Barlettův test

### 10.2.2 Jednofaktorová ANOVA

- Náhodná proměnná je ovlivněna pouze jedním faktorem.
- U náhodné veličiny $X$ uvažujeme jeden faktor $A$, který nabývá $I$ různých kvalitativních hodnot $A_1, A_2, \ldots, A_I$, kde $I > 2$.
  - Každá kvalitativní hodnota $A_i$ je popsána náhodnou veličinou $X_i$.
  - Náhodné veličiny $X_1, X_2, \ldots, X_I$ jsou nezávislé.
- Testujeme hypotézu
$$H : \mu_1 = \mu_2 = \ldots = \mu_n$$
- Proti alternativní
$$H_A : \exists i, j : \mu_i \neq \mu_j$$

### 10.2.3 Dvoufaktorová ANOVA

- V praxi se často setkáváme s pokusy, kdy sledujeme více působících faktorů, např. vliv krmení a plemene, vliv léku v různých stádiích onemocnění, vliv živné půdy a způsobu kultivace na růst zárodků, vliv různých druhů antibiotik a jejich dávky apod. Pokud zkoumáme vliv dvou a více faktorů působících na závisle proměnnou, hovoříme o vícefaktorové analýze rozptylu.
- Náhodná proměnná je ovlivněna dvěma (nebo více) faktory.
- Dvoufaktorová bez interakce – náhodná proměnná je ovlivněna dvěma nezávislými faktory.
- Dvoufaktorová s interakcí – náhodná proměnná je ovlivněna dvěma závislými faktory.

## 10.3 Testy dobré shody a testy o rozdělení

**Princip testů dobré shody**

- Testy dobré shody vycházejí z porovnání teoretické pravděpodobnosti a odhadnuté pravděpodobnosti pomocí relativních četností u náhodné veličiny, která může nabývat konečného počtu možností.
  - Odpovídají empirická data nějakému teoretickému rozdělení?
- Vychází se z Multinomického rozdělení, které definuje pravděpodobnost při výběru (s opakováním) z konečného počtu možností.

**Testy o rozdělení**

- Nechť $X$ je náhodná proměnná, která má distribuční funkci $F(x, \theta)$. Předpokládejme, že neznáme tvar distribuční funkce (nevíme jaké má rozdělení) a neznáme parametr $\theta$. Na základě měření (pokusů) chceme odhadnou typ rozdělení a neznámý parametr $\theta$. Provedeme $n$ pokusů (měření). Výsledky těchto pokusů jsou popsány náhodným výběrem $X = (X_1, \ldots, X_n)$ a jeho realizací $x = (x_1, \ldots, x_n)$.
- Opět předpokládáme, že složky náhodného vektoru jsou nezávislé a mají stejné rozdělení jako náhodná proměnná $X$.
- Při testování hypotéz o rozdělení si zvolíme rozdělení, s kterým chceme porovnat naměřená data. Pokud rozdělení obsahuje neznámý parametr, pro jeho odhad použijeme statistiky pro bodový odhad.
- Hypotéza je ve tvaru:

$$H : X \sim F(x, \theta) \ , \ H_A : X \neq F(x, \theta)$$

- Testy:
  - Pearsonův chí-kvadrát test ($\chi^2$),
  - Kolmogorov-Smirnov,
  - Anderson-Darling,
  - grafické metody.

## 10.4 Příklad: Při kontrole výrobků byla sledována odchylka X [mm] jejich rozměru od požadované velikosti. Naměřené hodnoty tvoří statistický soubor.

| Statistický soubor (př. 1, v. 1) | | | |
|---|---|---|---|
| i | x | i | x |
| 1 | 0.72 | 26 | -0.13 |
| 2 | -0.55 | 27 | 0.55 |
| 3 | 0.3 | 28 | -0.24 |
| 4 | 0.12 | 29 | 0.07 |
| 5 | 0.12 | 30 | 0.44 |
| 6 | 0.8 | 31 | -0.2 |
| 7 | 0.63 | 32 | -0.29 |
| 8 | -0.71 | 33 | 1 |
| 9 | -0.42 | 34 | 0.21 |
| 10 | -0.06 | 35 | -0.52 |
| 11 | 0.46 | 36 | -0.72 |
| 12 | 0.66 | 37 | 0.59 |
| 13 | 0.63 | 38 | -0.7 |
| 14 | 0.28 | 39 | -0.42 |
| 15 | -0.08 | 40 | -0.3 |
| 16 | 0.08 | 41 | -0.23 |
| 17 | -0.02 | 42 | -0.04 |
| 18 | 0.44 | 43 | 0.32 |
| 19 | 0.4 | 44 | -0.16 |
| 20 | 0.41 | 45 | -0.27 |
| 21 | -0.59 | 46 | 0.49 |
| 22 | -0.13 | 47 | 0.44 |
| 23 | 0.33 | 48 | 0.24 |
| 24 | 0.38 | 49 | -0.39 |
| 25 | -0.83 | 50 | -0.38 |

| Uspořádaný statistický soubor (př. 1, v. 1) | | | |
|---|---|---|---|
| i | x | i | x |
| 1 | -0.83 | 26 | 0.08 |
| 2 | -0.72 | 27 | 0.12 |
| 3 | -0.71 | 28 | 0.12 |
| 4 | -0.7 | 29 | 0.21 |
| 5 | -0.59 | 30 | 0.24 |
| 6 | -0.55 | 31 | 0.28 |
| 7 | -0.52 | 32 | 0.3 |
| 8 | -0.42 | 33 | 0.32 |
| 9 | -0.42 | 34 | 0.33 |
| 10 | -0.39 | 35 | 0.38 |
| 11 | -0.38 | 36 | 0.4 |
| 12 | -0.3 | 37 | 0.41 |
| 13 | -0.29 | 38 | 0.44 |
| 14 | -0.27 | 39 | 0.44 |
| 15 | -0.24 | 40 | 0.44 |
| 16 | -0.23 | 41 | 0.46 |
| 17 | -0.2 | 42 | 0.49 |
| 18 | -0.16 | 43 | 0.55 |
| 19 | -0.13 | 44 | 0.59 |
| 20 | -0.13 | 45 | 0.63 |
| 21 | -0.08 | 46 | 0.63 |
| 22 | -0.06 | 47 | 0.66 |
| 23 | -0.04 | 48 | 0.72 |
| 24 | -0.02 | 49 | 0.8 |
| 25 | 0.07 | 50 | 1 |

### 10.4.1 Proveďte roztřídění statistického souboru, vytvořte tabulku četností a nakreslete histogramy pro relativní četnosti a relativní kumulativní četnosti.

- Variační obor:

$$\langle x_{(1)}\ ,\ x_{(n)}\rangle = \langle \min_i x_i\ ,\ \max_i x_i\rangle = \langle -0.83\ ,\ 1\rangle$$

- Rozpětí:

$$x_{(n)} - x_{(1)} = 1.83$$

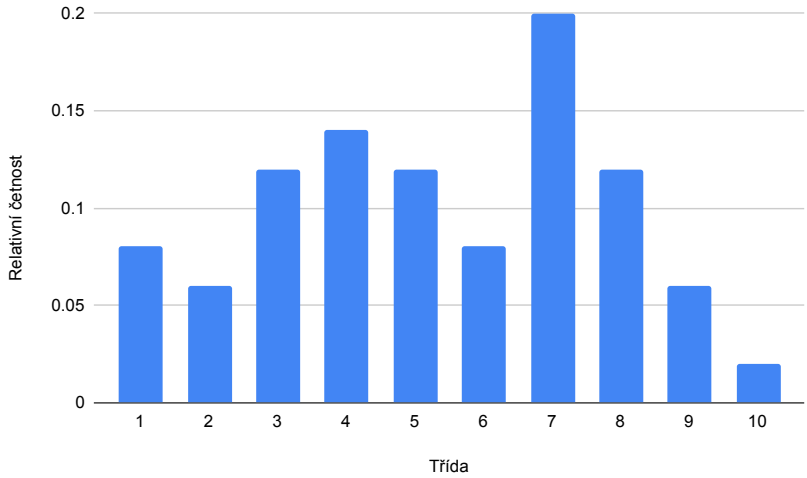
- Počet tříd:

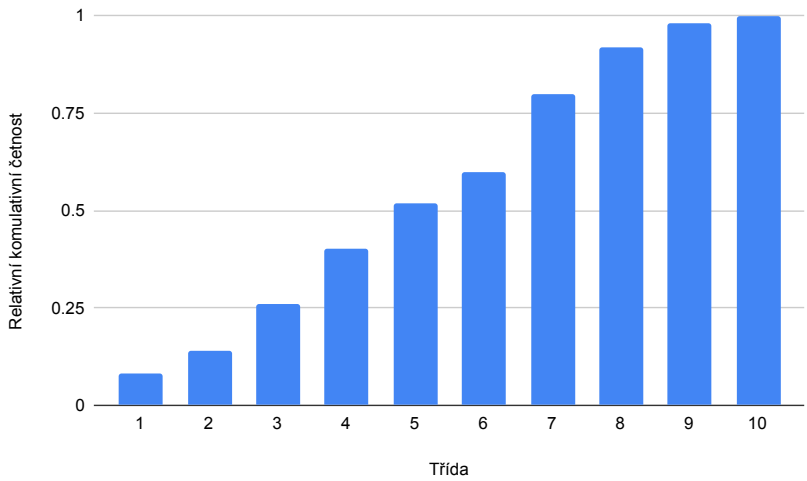
$$m = 10$$

- Délka třídy:

$$\frac{x_{(n)} - x_{(1)}}{m} = 0.183$$

| Třída | ( | ) | Střed třídy | Četnost | Komulativní četnost | Relativní četnost | Relativní komulativní četnost |
|---|---|---|---|---|---|---|---|
| 1 | -0.83 | -0.647 | -0.7385 | 4 | 4 | 0.08 | 0.08 |
| 2 | -0.647 | -0.464 | -0.5555 | 3 | 7 | 0.06 | 0.14 |
| 3 | -0.464 | -0.281 | -0.3725 | 6 | 13 | 0.12 | 0.26 |
| 4 | -0.281 | -0.098 | -0.1895 | 7 | 20 | 0.14 | 0.4 |
| 5 | -0.098 | 0.085 | -0.0065 | 6 | 26 | 0.12 | 0.52 |
| 6 | 0.085 | 0.268 | 0.1765 | 4 | 30 | 0.08 | 0.6 |
| 7 | 0.268 | 0.451 | 0.3595 | 10 | 40 | 0.2 | 0.8 |
| 8 | 0.451 | 0.634 | 0.5425 | 6 | 46 | 0.12 | 0.92 |
| 9 | 0.634 | 0.817 | 0.7255 | 3 | 49 | 0.06 | 0.98 |
| 10 | 0.817 | 1 | 0.9085 | 1 | 50 | 0.02 | 1 |



### 10.4.2 Vypočtěte aritmetický průměr, medián, modus, rozptyl a směrodatnou odchylku.

- Aritmetický průměr:

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i = 0.0546$$

- Medián:

$$\tilde{x} = 0.075$$

- Modus:

$$\hat{x} = 0.44$$

- Rozptyl:

$$s^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2 \approx 0.2031$$

- Směrodatná odchylka:

$$s = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2} \approx 0.4507$$

### 10.4.3 Vypočtěte bodové odhady střední hodnoty, rozptylu a směrodatné odchylky.

- Bodový odhad střední hodnoty:

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i = 0.0546$$

- Bodový odhad rozptylu:

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2 \approx 0.2073$$

- Bodový odhad směrodatné odchylky:

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2} \approx 0.4553$$

### 10.4.4 Testujte předpoklad o výběru z normálního rozdělení Pearsonovým (chí-kvadrát) testem na hladině významnosti $0.05$.

| Třída | ( | ) | Empirická četnost | Teoretická četnost | t |
|---|---|---|---|---|---|
| 1 | -0.83 | -0.647 | 4 | 1.782228068 | 2.75975473 |
| 2 | -0.647 | -0.464 | 3 | 3.283980097 | 0.02455699878 |
| 3 | -0.464 | -0.281 | 6 | 5.159357099 | 0.1369706484 |
| 4 | -0.281 | -0.098 | 7 | 6.911228534 | 0.001140227553 |
| 5 | -0.098 | 0.085 | 6 | 7.893760222 | 0.4543243877 |
| 6 | 0.085 | 0.268 | 4 | 7.687473576 | 1.768781543 |
| 7 | 0.268 | 0.451 | 10 | 6.383428059 | 2.048991933 |
| 8 | 0.451 | 0.634 | 6 | 4.519517866 | 0.4849692852 |
| 9 | 0.634 | 0.817 | 3 | 2.728304577 | 0.02705651101 |
| 10 | 0.817 | 1 | 1 | 1.404263731 | 0.1163806772 |
| Suma | | | 50 | 47.75354183 | 7.822926943 |

- Aby celkový počet teoretických četností odpovídal reálným, byly krajní intervaly rozšířeny. Aby všechny teoretické četnosti byly větší jako 1 a aspoň 80 % z nich bylo větších než 5 byly hranice tříd upraveny.

| Třída | ( | ) | Empirická četnost | Teoretická četnost | t |
|---|---|---|---|---|---|
| 1 | -10000 | -0.5 | 7 | 5.579005509 | 0.3619328461 |
| 2 | -0.5 | -0.3 | 4 | 5.322375553 | 0.3285519943 |
| 3 | -0.3 | -0.1 | 9 | 7.453019054 | 0.3210980718 |
| 4 | -0.1 | 0.1 | 6 | 8.631438102 | 0.8022378661 |
| 5 | 0.1 | 0.3 | 5 | 8.267269529 | 1.291242548 |
| 6 | 0.3 | 0.5 | 11 | 6.548893446 | 3.025297285 |
| 7 | 0.5 | 10000 | 8 | 8.197998807 | 0.004782085073 |
| Suma | | | 50 | 50 | 6.135142697 |

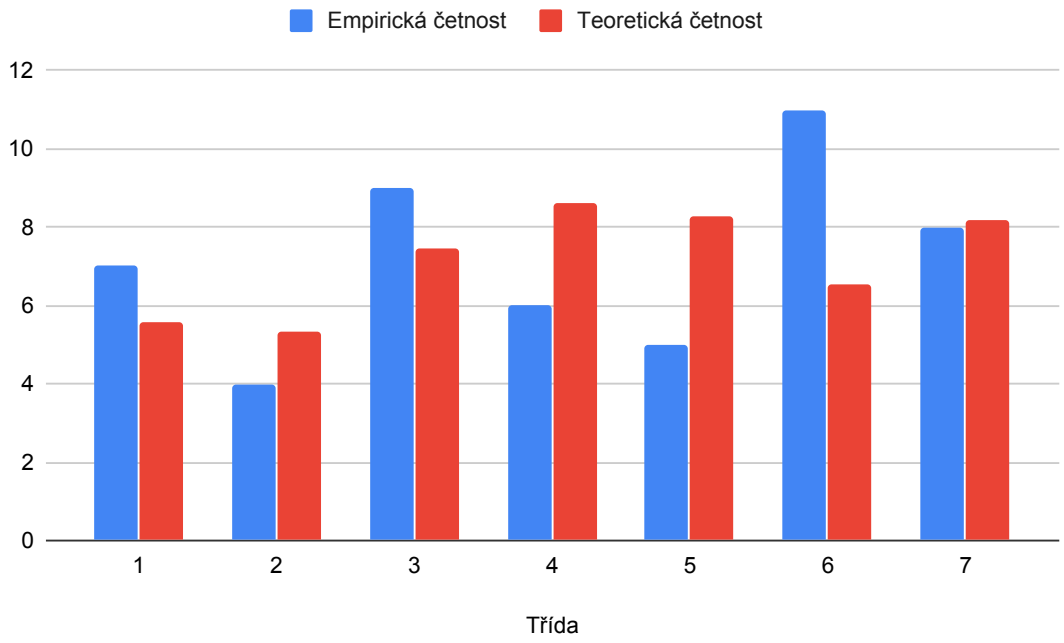
- Testovací kritérium: $t = \sum_{i=1}^{m} = \frac{(f_i - \hat{f}_i)^2}{\hat{f}_i} \approx 6.135$, kde $f$ je empirická četnost a $\hat{f}$ je teoretická četnost.
- Stupeň volnosti: $k = m - q - 1 = 4$, kde $m$ je počet tříd a $q$ je počet odhadů parametrů.
- Kvantil Pearsonova rozdělení pro hladinu významnosti $\alpha = 0.05$:

$$\chi^2_{1-\alpha}(k) = \chi^2_{0.95}(4) \approx 9.4877$$

- Doplněk kritického oboru:

$$\overline{W_\alpha} = \langle 0\ ,\ \chi^2_{1-\alpha}(k) \rangle \approx \langle 0\ ,\ 9.4877 \rangle$$

- Jelikož $t \in \overline{W_\alpha}$, tak hypotéza $X \sim N(0.0546, 0.2073)$ se **nezamítá**.




**10.4.5 Za předpokladu (bez ohledu na výsledek části d), že statistický soubor byl získán náhodným výběrem z normálního rozdělení, určete intervalové odhady střední hodnoty, rozptylu a směrodatné odchylky se spolehlivostí** 0.95 **a** 0.99**.**

- Předpokládáme $X \sim N(\mu, \sigma^2)$
- Bodový odhad střední hodnoty: $\overline{x} = 0.0546$
- Bodový odhad rozptylu: $s^2 \approx 0.2073$
- Bodový odhad směrodatné odchylky: $s \approx 0.4553$

**Intervalový odhad střední hodnoty**

- Stupeň volnosti: $k = n - 1 = 49$, kde $n$ je počet vzorků.
- Kvantil Studentova rozdělení pro hladinu významnosti $\alpha = 0.05$:

$$t_{1-\frac{\alpha}{2}}(k) = t_{0.975}(49) \approx 2.0096$$

- Kvantil Studentova rozdělení pro hladinu významnosti $\alpha = 0.01$:

$$t_{1-\frac{\alpha}{2}}(k) = t_{0.995}(49) \approx 2.6799$$

- Střední hodnota pro $\alpha = 0.05$ :

$$\mu \in \left\langle \overline{x} - t_{1-\frac{\alpha}{2}}(k) \cdot \frac{s}{\sqrt{n}} \ , \ \overline{x} + t_{1-\frac{\alpha}{2}}(k) \cdot \frac{s}{\sqrt{n}} \right\rangle \approx \left\langle -0.0761 \ , \ 0.1853 \right\rangle$$

- Střední hodnota pro $\alpha = 0.01$ :

$$\mu \in \left\langle \overline{x} - t_{1-\frac{\alpha}{2}}(k) \cdot \frac{s}{\sqrt{n}} \ , \ \overline{x} + t_{1-\frac{\alpha}{2}}(k) \cdot \frac{s}{\sqrt{n}} \right\rangle \approx \left\langle -0.1197 \ , \ 0.2289 \right\rangle$$

**Intervalový odhad rozptylu**

- Kvantil Pearsonova rozdělení pro hladinu významnosti $\alpha = 0.05$:

$$\chi^2_{\frac{\alpha}{2}}(k) = \chi^2_{0.025}(49) \approx 31.555$$

$$\chi^2_{1-\frac{\alpha}{2}}(k) = \chi^2_{0.975}(49) \approx 70.222$$

- Kvantil Pearsonova rozdělení pro hladinu významnosti $\alpha = 0.01$:

$$\chi^2_{\frac{\alpha}{2}}(k) = \chi^2_{0.005}(49) \approx 27.249$$

$$\chi^2_{1-\frac{\alpha}{2}}(k) = \chi^2_{0.995}(49) \approx 78.231$$

- Rozptyl pro $\alpha = 0.05$ :

$$\sigma^2 \in \left\langle \frac{(n-1)\cdot s^2}{\chi^2_{1-\frac{\alpha}{2}}(k)} \ , \ \frac{(n-1)\cdot s^2}{\chi^2_{\frac{\alpha}{2}}(k)} \right\rangle \approx \left\langle 0.1447 \ , \ 0.3219 \right\rangle$$

- Rozptyl pro $\alpha = 0.01$ :

$$\sigma^2 \in \left\langle \frac{(n-1)\cdot s^2}{\chi^2_{1-\frac{\alpha}{2}}(k)} \ , \ \frac{(n-1)\cdot s^2}{\chi^2_{\frac{\alpha}{2}}(k)} \right\rangle \approx \left\langle 0.1298 \ , \ 0.3727 \right\rangle$$

**Intervalový odhad směrodatné odchylky**

- Směrodatná odchylka pro $\alpha = 0.05$ :

$$\sigma \in \left\langle \sqrt{\frac{(n-1)\cdot s^2}{\chi^2_{1-\frac{\alpha}{2}}(k)}} \ , \ \sqrt{\frac{(n-1)\cdot s^2}{\chi^2_{\frac{\alpha}{2}}(k)}} \right\rangle \approx \left\langle 0.3803 \ , \ 0.5674 \right\rangle$$

- Směrodatná odchylka pro $\alpha = 0.01$ :

$$\sigma \in \left\langle \sqrt{\frac{(n-1)\cdot s^2}{\chi^2_{1-\frac{\alpha}{2}}(k)}} \ , \ \sqrt{\frac{(n-1)\cdot s^2}{\chi^2_{\frac{\alpha}{2}}(k)}} \right\rangle \approx \left\langle 0.3603 \ , \ 0.6106 \right\rangle$$

### 10.4.6 Testujte hypotézu optimálního seřízení stroje, tj. že střední hodnota odchylky je nulová, proti dvoustranné alternativní hypotéze, že střední hodnota odchylky je různá od nuly, a to na hladině významnosti $0.05$.

- Hypotéza: $H_0 : \mu = 0$
- Alternativní hypotéza: $H_A : \mu \neq 0$
- Bodový odhad střední hodnoty: $\overline{x} = 0.0546$
- Bodový odhad směrodatné odchylky: $s \approx 0.4553$
- Počet vzorků: $n = 50$

**Testujeme pomocí Studentova jednovýběrového testu**

- Testovací kritérium: $t = \dfrac{\overline{x} - \mu}{s} \cdot \sqrt{n} \approx 0.848$
- Stupeň volnosti: $k = n - 1 = 49$
- Kvantil Studentova rozdělení pro hladinu významnosti $\alpha = 0.05$:

$$t_{1-\frac{\alpha}{2}}(k) = t_{0.975}(49) \approx 2.0096$$

- Doplněk kritického oboru pro alternativní hypotézu $H_A$:

$$\overline{W_\alpha} = \big\langle -t_{1-\frac{\alpha}{2}}(k)\ ,\ t_{1-\frac{\alpha}{2}}(k) \big\rangle \approx \big\langle -2.0096\ ,\ 2.0096 \big\rangle$$

- Jelikož $t \in \overline{W_\alpha}$, tak hypotéza $H_0$ se **nezamítá** a alternativní hypotéza $H_A$ se **zamítá**.

**10.4.7 Ověřte statistickým testem na hladině významnosti $0.05$, zda seřízení stroje ovlivnilo kvalitu výroby, víte-li, že výše uvedený statistický soubor $50$ hodnot vznikl spojením dvou dílčích statistických souborů tak, že po naměření prvních $20$ hodnot bylo provedeno nové seřízení stroje a pak bylo naměřeno zbývajících $30$ hodnot.**

| X = x1 : x20 | |
|---|---|
| i | X |
| 1 | 0.72 |
| 2 | -0.55 |
| 3 | 0.3 |
| 4 | 0.12 |
| 5 | 0.12 |
| 6 | 0.8 |
| 7 | 0.63 |
| 8 | -0.71 |
| 9 | -0.42 |
| 10 | -0.06 |
| 11 | 0.46 |
| 12 | 0.66 |
| 13 | 0.63 |
| 14 | 0.28 |
| 15 | -0.08 |
| 16 | 0.08 |
| 17 | -0.02 |
| 18 | 0.44 |
| 19 | 0.4 |
| 20 | 0.41 |

| Y = x21 : x50 | |
|---|---|
| i | Y |
| 1 | -0.59 |
| 2 | -0.13 |
| 3 | 0.33 |
| 4 | 0.38 |
| 5 | -0.83 |
| 6 | -0.13 |
| 7 | 0.55 |
| 8 | -0.24 |
| 9 | 0.07 |
| 10 | 0.44 |
| 11 | -0.2 |
| 12 | -0.29 |
| 13 | 1 |
| 14 | 0.21 |
| 15 | -0.52 |
| 16 | -0.72 |
| 17 | 0.59 |
| 18 | -0.7 |
| 19 | -0.42 |
| 20 | -0.3 |
| 21 | -0.23 |
| 22 | -0.04 |
| 23 | 0.32 |
| 24 | -0.16 |
| 25 | -0.27 |
| 26 | 0.49 |
| 27 | 0.44 |
| 28 | 0.24 |
| 29 | -0.39 |
| 30 | -0.38 |

$$n_x = 20 \qquad n_y = 30$$

$$\overline{x} = 0.2105 \qquad \overline{y} \approx -0.0493$$

$$s_x^2 \approx 0.1806 \qquad s_y^2 \approx 0.2039$$

$$s_x \approx 0.425 \qquad s_y \approx 0.4516$$

**Test rovnosti rozptylů pomocí F-testu**

- Hypotéza
$$H_0 : \sigma_x^2 = \sigma_y^2$$
- Alternativní hypotéza
$$H_A : \sigma_x^2 \neq \sigma_y^2$$
- Testovací kritérium:
$$t = \frac{s_x^2}{s_y^2} \approx 0.7844$$
- Stupně volnosti:
$$k_x = n_x - 1 = 19$$
$$k_y = n_y - 1 = 29$$
- Kvantily Fisher-Snedecorova rozdělení pro hladinu významnosti $\alpha = 0.05$:
$$F_{\frac{\alpha}{2}}(k_x, k_y) = F_{0.025}(19, 29) \approx 0.4163$$
$$F_{1-\frac{\alpha}{2}}(k_x, k_y) = F_{0.975}(19, 29) \approx 2.2313$$
- Doplněk kritického oboru pro alternativní hypotézu $H_A$:
$$\overline{W_\alpha} = \left\langle F_{\frac{\alpha}{2}}(k_x, k_y)\ ,\ F_{1-\frac{\alpha}{2}}(k_x, k_y)\right\rangle \approx \left\langle 0.4163\ ,\ 2.2313\right\rangle$$
- Jelikož $t \in \overline{W_\alpha}$, tak hypotéza $H_0$ se **nezamítá**.

**Test rovnosti středních hodnot pomocí Studentova dvouvýběrového testu**

- Hypotéza (pro $\mu_0 = 0$ za podmínky $\sigma_x^2 = \sigma_y^2$)
$$H_0 : \mu_x - \mu_y = \mu_0$$
- Alternativní hypotéza
$$H_A : \mu_x - \mu_y \neq 0$$
- Stupeň volnosti:
$$k = n_x + n_y - 2 = 48$$
- Testovací kritérium:
$$t = \frac{\overline{x} - \overline{y} - \mu_0}{\sqrt{k_x \cdot s_x^2 + k_y \cdot s_y^2}} \cdot \sqrt{\frac{n_x \cdot n_y \cdot k}{n_x + n_y}} \approx 2.04$$
- Kvantil Studentova rozdělení pro hladinu významnosti $\alpha = 0.05$:
$$t_{1-\frac{\alpha}{2}}(k) = t_{0.975}(48) \approx 2.0106$$
- Doplněk kritického oboru pro alternativní hypotézu $H_A$:
$$\overline{W_\alpha} = \left\langle -t_{1-\frac{\alpha}{2}}(k)\ ,\ t_{1-\frac{\alpha}{2}}(k)\right\rangle \approx \left\langle -2.0106\ ,\ 2.0106\right\rangle$$
- Jelikož $t \notin \overline{W_\alpha}$, tak hypotéza $H_0$ se **zamítá**.

# Kapitola 11

# MSP – Regresní analýza.

## 11.1 Zdroje

- `MSP_pred_02_Opakovani_Statistika_Regrese.pdf`
- `MSP_pred_06_Regresni-analyza_Uvod.pdf`
- `MSP_pred_07_Reg-analyza_Testy_Spec-modely_Diagnostika.pdf`
- Wikipedia

## 11.2 Úvod a kontext

- Základní úlohou regresní analýzy je nalezení vhodného modelu studované závislosti.
- **Korelační analýza** se zabývá vzájemnými (většinou lineárními) závislostmi, kdy se klade důraz především na intenzitu (sílu) vzájemného vztahu než na zkoumání veličin ve směru příčina – následek.
- **Regresní analýza** se zabývá jednostrannými závislostmi. Jedná se o situaci, kdy proti sobě stojí vysvětlující (nezávislá) proměnná v úloze příčin a vysvětlovaná (závislá) proměnná v úloze následků (hledání závislostí mezi atributy). V podstatě jde o aproximaci souboru dat vhodnou funkcí (tzv. regresní funkce).
  - Na začátku regresní analýzy je třeba odhadnout typ funkce. K tomu slouží explorativní analýza, která se používá ke zjištění, jak cílový atribut závisí na ostatních atributech (na kterých a jak).
  - Poté je třeba určit parametry regresní funkce, například pomocí metody nejmenších čtverců.
  - V závěru je třeba model verifikovat, zda funguje i na datech, na kterých nebyl přímo trénován.
- Rozlišujeme různé typy:
  - Jednoduchá lineární regrese – Cílový atribut závisí na jednom dalším atributu lineárně.
  - Vícenásobná lineární regrese – Cílový atribut závisí na několika dalších atributech lineárně.
  - Nelineární regrese – Cílový atribut závisí na dalších atributech nelineárně.

## 11.3 Polynomiální regrese

- Polynomiální regrese představuje proložení (aproximaci) zadaných hodnot polynomem.
- Postup:
  - Mějme datový soubor $Y$ reprezentovaný uspořádanou n-ticí:

$$Y = (y_1, y_2, \ldots, y_n)$$

  - cílem je najít takový polynom k-tého stupně:

$$P_k(x) = p_0 + p_1 x + \ldots + p_k x^k$$

  - pro který platí

$$y_i = P_k(x_i) + e_i$$

    pro $i \in 1 \ldots n$, kde $e_i$ je odchylka (nebo také chyba). Koeficienty $p_0, p_1, \ldots, p_k$ jsou přitom voleny tak, aby součet druhých mocnin odchylek, resp. suma

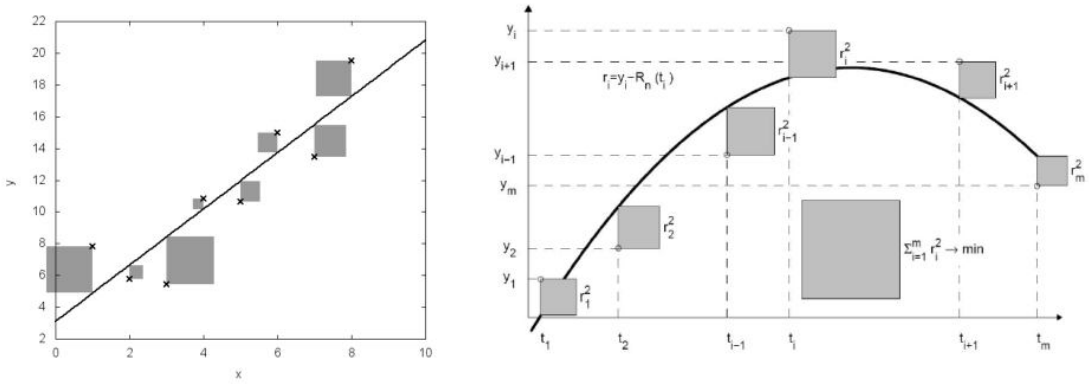
$$\sum_{i=1}^{n} e_i^2$$

    , byla co nejmenší.

### 11.3.1 Metoda nejmenších čtverců

Předpokládejme, že známe tvar regresní funkce $y = \varphi(\boldsymbol{x}, \boldsymbol{\beta})$ a neznámé $\boldsymbol{\beta} = (\beta_1, ..., \beta_m)^T$.
K odhadu regresních koeficientů $\boldsymbol{\beta} = (\beta_1, ..., \beta_m)^T$ využijeme metodu nejmenších čtverců – minimalizujeme tzv. **reziduální součet čtverců**:

$$S^* = S_e = \sum_{i=1}^{n} \left(y_i - \varphi(\boldsymbol{x}_i, \boldsymbol{\beta})\right)^2$$




Tento součet lze brát jako funkci proměnných $\boldsymbol{\beta} = (\beta_1, ..., \beta_m)^T$: $S^* \equiv S^*(\beta_1, \cdots, \beta_m)$

A pomocí parciálních derivací hledáme její minimum: $\dfrac{\partial S^*}{\partial \beta_j} = 0$.

Dostaneme $m$ rovnic pro $m$ neznámých parametrů $\boldsymbol{\beta} = (\beta_1, ..., \beta_m)^T$.

Obrázek 11.1: Odhad parametrů regresní funkce pomocí metody nejmenších čtverců.

### 11.3.2 Střední kvadratická chyba

- Jedna z chybových metrik je tzv. střední kvadratická chyba (MSE, *Mean Squared Error*).
- Mějme trénovací datový soubor $(X, Y)$, kde $X = (x_1, x_2, \ldots, x_n)$ jsou hodnoty ovlivňující proměnné a $Y = (y_1, y_2, \ldots, y_n)$ jsou hodnoty cílové proměnné, a regresní funkci $f$, která aproximuje datovou sadu $(X, Y)$.
- Výpočet chyby MSE regresní funkce $f$ na datovém souboru $(X, Y)$:

$$\mathrm{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - f(x_i))^2$$

## 11.4 Příklad: Měřením dvojice (Výška[cm], Váha[kg]) u vybraných studentů z FIT byl získán dvourozměrný statistický soubor

| Dvourozměrný statistický soubor (př. 2, v. 23) | |
|---|---|
| X - výška | Y - váha |
| 161 | 81 |
| 180 | 92 |
| 179 | 87 |
| 196 | 119 |
| 198 | 114 |
| 187 | 103 |
| 169 | 88 |
| 169 | 92 |
| 180 | 94 |
| 175 | 90 |
| 151 | 59 |
| 182 | 112 |
| 193 | 110 |
| 154 | 64 |
| 182 | 106 |
| 190 | 108 |
| 158 | 78 |
| 200 | 126 |
| 197 | 116 |
| 176 | 106 |

Obrázek 11.2: Statistický soubor pro tento a všechny další příklady.

### 11.4.1 Vypočtěte bodový odhad koeficientu korelace.

$$n = 20 \qquad \sum_{i=1}^{n} x_i^2 = 643\,961$$

$$\overline{x} = 178.85 \qquad \sum_{i=1}^{n} y_i^2 = 195\,277$$

$$\overline{y} = 97.25 \qquad \sum_{i=1}^{n} x_i \cdot y_i = 352\,644$$

- Odhad koeficientu korelace:

$$r = \frac{\sum\limits_{i=1}^{n} x_i \cdot y_i - n \cdot \overline{x} \cdot \overline{y}}{\sqrt{\left(\sum\limits_{i=1}^{n} x_i^2 - n \cdot \overline{x}^2\right) \cdot \left(\sum\limits_{i=1}^{n} y_i^2 - n \cdot \overline{y}^2\right)}} \approx 0.9409$$

### 11.4.2 Na hladině významnosti 0.05 testujte hypotézu, že náhodné veličiny Výška a Váha jsou lineárně nezávislé.

- Hypotéza
$$H_0 : \rho = 0$$
- Alternativní hypotéza
$$H_A : \rho \neq 0$$
- Testovací kritérium
$$t = \frac{|r| \cdot \sqrt{n-2}}{\sqrt{1-r^2}} \approx 11.7859$$
- Stupeň volnosti
$$k = n - 2$$
- Kvantil Studentova rozdělení pro hladinu významnosti $\alpha = 0.05$:
$$t_{1-\frac{\alpha}{2}}(k) = t_{0.975}(18) \approx 2.101$$
- Doplněk kritického oboru pro alternativní hypotézu $H_A$:
$$\overline{W_\alpha} = \left\langle 0\ ,\ t_{1-\frac{\alpha}{2}}(k) \right\rangle \approx \left\langle 0\ ,\ 2.101 \right\rangle$$
- Jelikož $t \notin \overline{W_\alpha}$, tak hypotéza $H_0$ se **zamítá**.

### 11.4.3 Regresní analýza, data proložte přímkou

- Tvar přímky
$$Vaha = \beta_0 + \beta_1 \cdot Vyska$$
- Pomocné výpočty

$$n = 20 \qquad \sum_{i=1}^{n} x_i^2 = 643\,961$$

$$\sum_{i=1}^{n} x_i = 3\,577 \qquad \sum_{i=1}^{n} y_i^2 = 195\,277$$

$$\sum_{i=1}^{n} y_i = 1\,945 \qquad \sum_{i=1}^{n} x_i \cdot y_i = 352\,644$$

$$H = \begin{pmatrix} n & \sum\limits_{i=1}^{n} x_i \\ \sum\limits_{i=1}^{n} x_i & \sum\limits_{i=1}^{n} x_i^2 \end{pmatrix}$$

$$det(H) = n \cdot \sum_{i=1}^{n} x_i^2 - \left( \sum_{i=1}^{n} x_i \right)^2 = 84\,291$$

**Bodový odhad koeficientů $\beta_0$, $\beta_1$ a rozptylu $s^2$**

- Hledáme lineární funkci $y = \beta_0 + \beta_1 \cdot x$, která bude nejlépe aproximovat naše naměřená data. Bodové odhady koeficientů $\beta_0$ a $\beta_1$ budeme značit $b_0$ a $b_1$.
- Bodový odhad koeficientů pomocí metody nejmenších čtverců

$$b_1 = \frac{1}{det(H)} \cdot \left( n \cdot \sum_{i=1}^{n} x_i \cdot y_i - \sum_{i=1}^{n} x_i \cdot \sum_{i=1}^{n} y_i \right) \approx 1.1343$$

$$b_0 = \overline{y} - b_1 \cdot \overline{x} \approx -105.6274$$

  - Regresní funkce:

$$y = 1.1343 \cdot x - 105.6274$$

- Bodový odhad rozptylu pomocí metody nejmenších čtverců
  - Minimální hodnota reziduálního součtu čtverců

$$S^*_{min} = \sum_{i=1}^{n} y_i^2 - b_0 \cdot \sum_{i=1}^{n} y_i - b_1 \cdot \sum_{i=1}^{n} x_i \cdot y_i \approx 702.7344$$

  - Rozptyl

$$s^2 = \frac{S^*_{min}}{n-2} \approx 39.0408$$

**Testování hypotézy $H_1 : \beta_0 = -100$**

- Alternativní hypotéza

$$H_{1A} : \beta_0 \neq -100$$

$$h_{11} = \frac{\sum\limits_{i=1}^{n} x_i^2}{det(H)} \approx 7.6397$$

- Testovací kritérium

$$t_1 = \frac{b_0 - \beta_0}{s \cdot \sqrt{h_{11}}} \approx -0.3258$$

- Stupeň volnosti
$$k = n - 2$$
- Kvantil Studentova rozdělení pro hladinu významnosti $\alpha = 0.05$:
$$t_{1-\frac{\alpha}{2}}(k) = t_{0.975}(18) \approx 2.101$$
- Doplněk kritického oboru pro alternativní hypotézu $H_{1A}$:
$$\overline{W_\alpha} = \big\langle -t_{1-\frac{\alpha}{2}}(k)\ ,\ t_{1-\frac{\alpha}{2}}(k)\big\rangle \approx \big\langle -2.101\ ,\ 2.101\big\rangle$$
- Jelikož $t_1 \in \overline{W_\alpha}$, tak hypotéza $H_1$ se **nezamítá**.

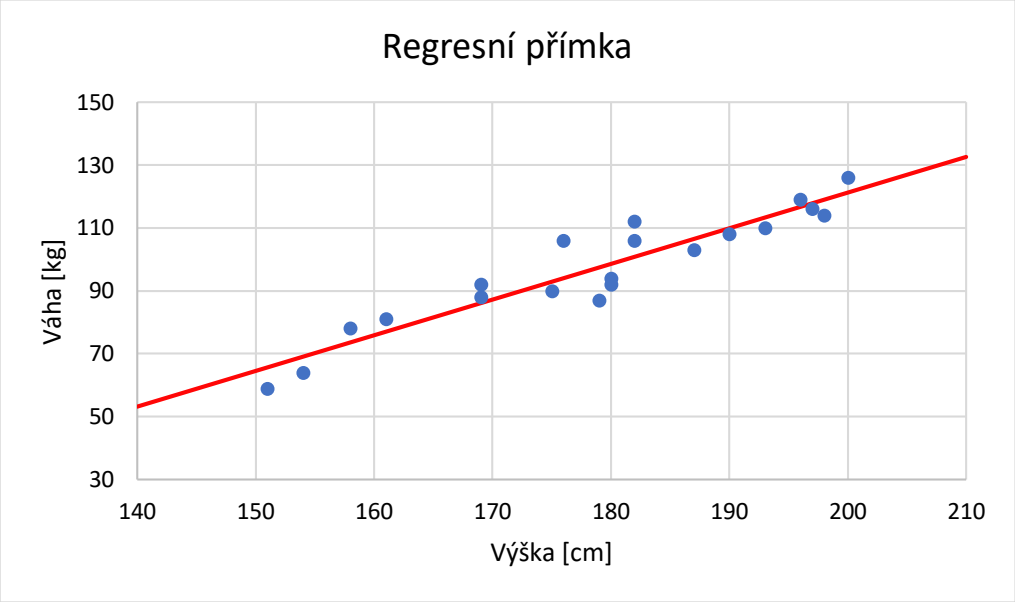
**Testování hypotézy** $H_2 : \beta_1 = 1$

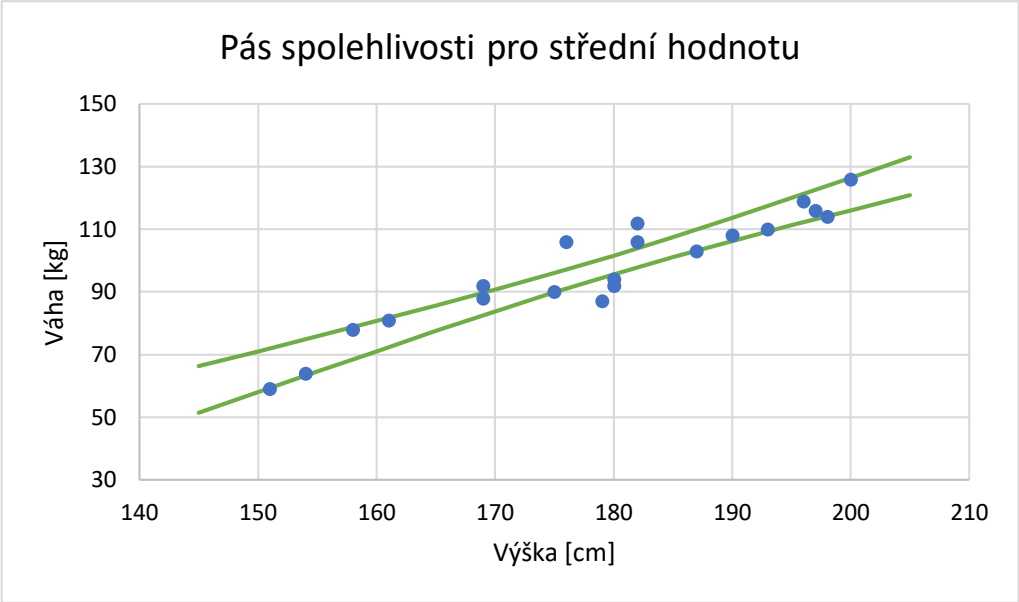
- Alternativní hypotéza
$$H_{2A} : \beta_1 \neq 1$$
$$h_{22} = \frac{n}{det(H)} \approx -0.9207$$
- Testovací kritérium
$$t_2 = \frac{b_1 - \beta_1}{s \cdot \sqrt{h_{22}}} \approx -0.0234$$
- Doplněk kritického oboru je stejný jako u testování hypotézy $H_1$.
- Jelikož $t_2 \in \overline{W_\alpha}$, tak hypotéza $H_2$ se **nezamítá**.

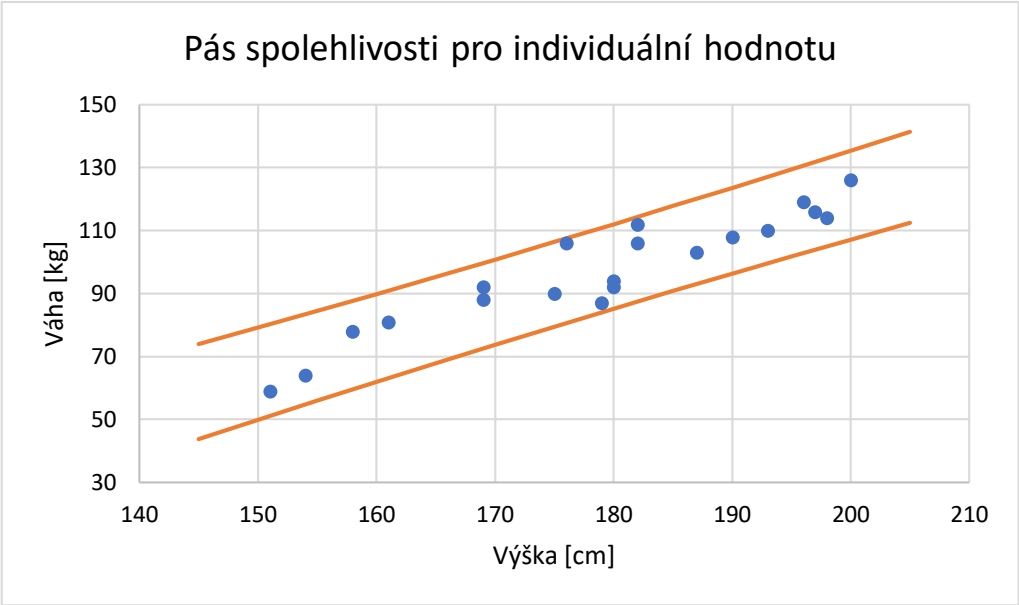
**Graf bodů s regresní přímkou a pásem spolehlivosti pro individuální hodnotu výšky**

- Intervalový odhad střední hodnoty $y$
$$\Big\langle (b_0 + b_1 \cdot x) - t_{1-\frac{\alpha}{2}}(k) \cdot s \cdot \sqrt{h^*}\ ,\ (b_0 + b_1 \cdot x) + t_{1-\frac{\alpha}{2}}(k) \cdot s \cdot \sqrt{h^*}\Big\rangle$$
- Intervalový odhad individuální hodnoty $y$
$$\Big\langle (b_0 + b_1 \cdot x) - t_{1-\frac{\alpha}{2}}(k) \cdot s \cdot \sqrt{h^* + 1}\ ,\ (b_0 + b_1 \cdot x) + t_{1-\frac{\alpha}{2}}(k) \cdot s \cdot \sqrt{h^* + 1}\Big\rangle$$
  kde
$$h^* = \frac{1}{n} + \frac{n \cdot (x - \overline{x})^2}{det(H)}$$

| Výpočet pásu spolehlivosti | | | | | | |
|---|---|---|---|---|---|---|
| | | Střední hodnota y | | Individuální hodnota y | | |
| x | y | ( | ) | ( | ) | h* |
| 145 | 58.8461 | 51.40493336 | 66.29996989 | 43.75984854 | 73.94505472 | 0.3218730351 |
| 150 | 64.5176 | 57.99367516 | 71.05466926 | 49.86236702 | 79.1859774 | 0.2474878694 |
| 155 | 70.1891 | 64.55022056 | 75.84156504 | 55.90621979 | 84.48556581 | 0.1849663665 |
| 160 | 75.8606 | 71.05677164 | 80.67845514 | 61.88672149 | 89.84850528 | 0.1343085264 |
| 165 | 81.5321 | 77.48234722 | 85.59632073 | 67.79960031 | 95.27906764 | 0.0955143491 |
| 170 | 87.2036 | 83.77325809 | 90.64885103 | 73.64125042 | 100.7808587 | 0.06858383457 |
| 175 | 92.8751 | 89.84598285 | 95.91956745 | 79.4089766 | 106.3565737 | 0.05351698283 |
| 180 | 98.5466 | 95.60998723 | 101.4990042 | 85.10119614 | 112.0077953 | 0.05031379388 |
| 185 | 104.2181 | 101.0383467 | 107.414086 | 90.71756529 | 117.7348674 | 0.05897426772 |
| 190 | 109.8896 | 106.1966864 | 113.5991875 | 96.2590075 | 123.5368663 | 0.07949840434 |
| 195 | 115.5611 | 111.178717 | 119.960598 | 101.7276382 | 129.4116768 | 0.1118862037 |
| 200 | 121.2326 | 116.054296 | 126.4284602 | 107.1266 | 135.3561562 | 0.1561376659 |
| 205 | 126.9041 | 120.8653187 | 132.9608787 | 112.4598367 | 141.3663607 | 0.2122527909 |




Pás spolehlivosti pro střední hodnotu
150
130
110
90
70
50
30
Váha [kg]
140
150
160
170
180
190
200
210
Výška [cm]


Pás spolehlivosti pro individuální hodnotu
150
130
110
90
70
50
30
Váha [kg]
140
150
160
170
180
190
200
210
Výška [cm]

# Kapitola 12

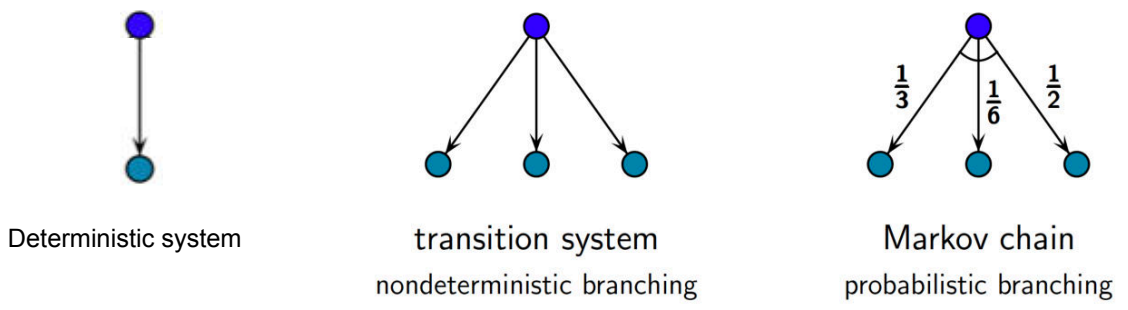
# MSP – Markovské řetězce a základní techniky pro jejich analýzu.

## 12.1 Zdroje

- `MSP_11_Markov_chains.pdf`
- `MSP_12_Markov_decision_processes.pdf`
- `MSP_2021-11-30_1080p.mp4`
- `MSP_2021-12-07_1080p.mp4`

## 12.2 Úvod a kontext

- Markovské řetězce a další pravděpodobnostní modely slouží pro modelování náhodných (stochastických) systémů.
- Stochastické systémy můžeme dělit podle:
  - času (spojitý / diskrétní),
  - způsobu rozhodování (plně pravděpodobnostní / nedeterministické),
  - velikosti stavového prostoru (konečný / nekonečný),
  - pozorovatelnosti prostředí (plně pozorovatelné prostředí / částečně pozorovatelné prostředí).




Obrázek 12.1: Deterministické, nedeterministické a pravděpodobnostní větvení.

| | Fully probabilistic | Nondeterministic |
|---|---|---|
| Discrete time | Discrete-time Markov chains (DTMCs) | Markov decision processes (MDPs) (probabilistic automata) |
| Continuous time | Continuous-time Markov chains (CTMCs) | CTMDPs/IMCs<br>Probabilistic timed automata (PTAs) |

Obrázek 12.2: Dělení pravděpodobnostních modelů na základě času a způsobu rozhodování.

## 12.3 Markovské řetězce

- Markovské řetězce (DTMC, *Discrete-Time Markov Chains*) jsou pravděpodobnostní modely, které májí:
  - diskrétní čas,
  - plně pravděpodobnostní způsob rozhodování,
  - konečný stavový prostor,
  - plně pozorovatelné prostředí.
- Formálně, markovský řetězec je čtveřice $D = (S, s_0, P, L)$, kde
  - $S$ je konečná množina stavů;
  - $s_0 \in S$ je výchozí stav (může být zobecněn jako výchozí pravděpodobnostní distribuce);
  - $P : S \times S \to \langle 0, 1 \rangle$ je pravděpodobnostní přechodová matice, pro kterou platí:

$$\forall s \in S \sum_{s' \in S} P(s, s') = 1$$

  - $L : S \to 2^{AP}$ je funkce označující stavy.
- Pravděpodobnost přechodu závisí pouze na aktuálním stavu (*memorylessness*).

**Cesta** Cesta je posloupnost stavů $\langle s_0, s_1, s_2, \dots \rangle$ pro kterou platí $\forall i \; P(s_i, s_{i+1}) > 0$.

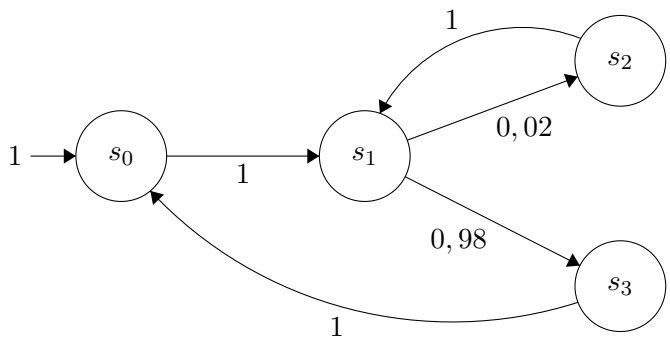
**Pravděpodobnost cesty** Pravděpodobnost ($Pr$, *probability measure*) konečné cesty $\omega = \langle s_0, s_1, s_2, \dots, s_n \rangle$ je:

- $Pr(s_0) = 1$
- $Pr(s_0, s_1, s_2, \dots, s_n) = P(s_0, s_1) \cdot P(s_1, s_2) \cdot \dots \cdot P(s_{n-1}, s_n)$.

**Příklad 1** Pravděpodobnostní systém doručování zpráv. Význam stavů:

- start – $s_0$;

- zpráva byla doručena – $s_3$;
- zpráva byla ztracena – $s_2$;
- odesílání – $s_1$.




Obrázek 12.3: Pravděpodobnostní systém doručování zpráv modelpván pomocí markovského řetězce.

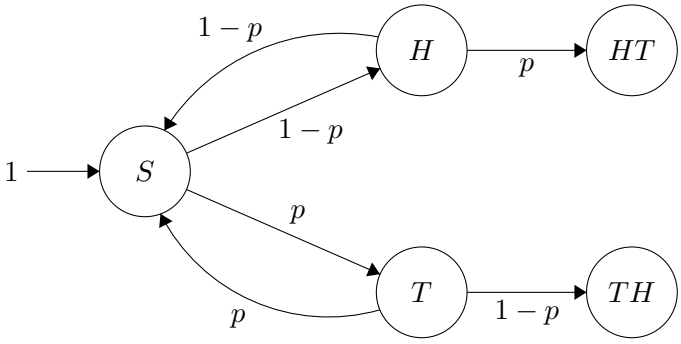
- Jaká je pravděpodobnost, že zpráva byla úspěšně přijata do 5ti kroků?

$$Pr(s_0, s_1, s_3) + Pr(s_0, s_1, s_2, s_1, s_3) = 0,98 + 0,02 \cdot 0,98 = 0,9996$$

- Jaká je pravděpodobnost, že zpráva bude někdy doručena?

$$\sum_{n=0}^{\infty} Pr(s_0, (s_1, s_2)^n, s_3) = \sum_{n=0}^{\infty} 0,02^n \cdot 0,98 = 1$$

**Příklad 2** Pravděpodobnostní systém pro namodelování férové mince pomocí „cinklé“ mince (nevíme jak).




Obrázek 12.4: Pravděpodobnostní systém pro namodelování férové mince pomocí „cinklé“ mince.

## 12.4 Analýza přechodů (*transient analysis*)

- Vysvětlení:
  - $t_k(s)$ vyjadřuje pravděpodobnost, že po spuštění procesu z počátečního stavu $s_0$, se nacházím ve stavu $s \in S$ v čase $k \geq 0$.

- Formálně:

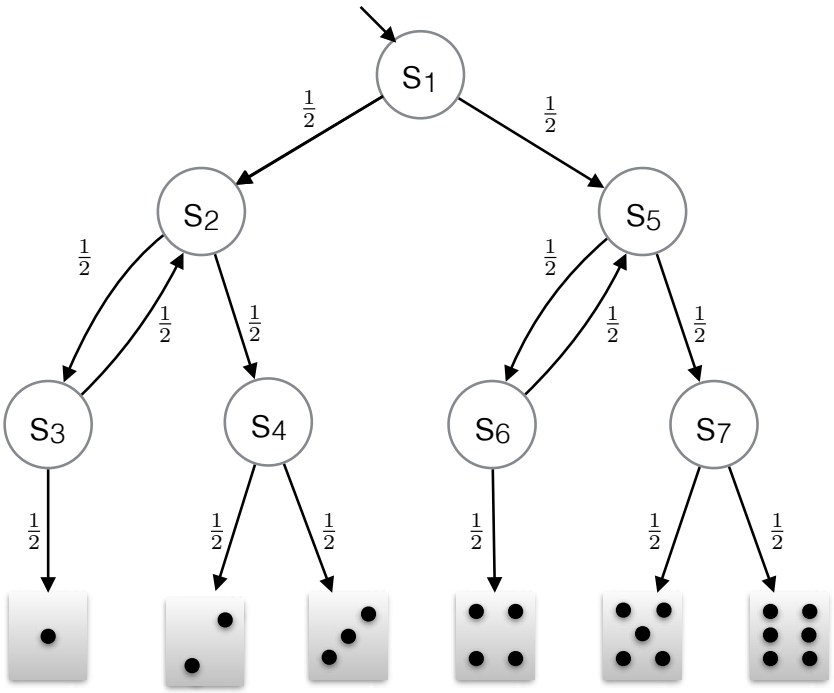
$$t_k(s) = P[X(k) = s \mid X(0) = s_0]$$

- Můžeme použít pravděpodobnost $Pr$ a vypsat všechny cesty délky $k$, které vedou do $s$ – exponenciální složitost $\mathcal{O}(e^k)$.
- Nebo lépe, můžeme využít vlastnost *memorylessness* a dostat se na lineární složitost $\mathcal{O}(k)$ (pravděpodobnost přechodu v čase $k-1$ z nějakého předchůdce $s$ do $s$).

$$t_0(s_0) = 1 \ , \ t_0(s) = 0 \ , \ s \neq s_0$$

$$t_k(s) = \sum_{s' \in S} t_{k-1}(s') \cdot P(s', s)$$

**Příklad 3** Modelování protokolu házení férovou šestistrannou kostkou pomocí férové mince (*Knuth-Yao dice*).




Obrázek 12.5: Modelování protokolu házení férovou šestistrannou kostkou pomocí férové mince. Stavy reprezentující výsledek hodu kostkou mají *self-loop* s pravděpodobnostní 1 a značme je $r_i$ pro $i \in \{1, 2, \ldots, 6\}$.

- Tranzientní analýza ze stavu $s_1$:
  - pro $k = 1$ : $s_2 = s_5 = 0,5$
  - pro $k = 2$ : $s_3 = s_4 = s_6 = s_7 = 0,25$
  - pro $k = 3$ : $s_2 = s_5 = r_1 = r_2 = \ldots = r_6 = 0,125$
  - pro $k = 5$ : $i \in \{1, 2, \ldots, 6\}$ : $r_i = (0,5)^3 + (0,5)^5 = 0,156$

## 12.5 Analýza ustáleného stavu (*steady state analysis*)

- Vysvětlení:

- $t_\infty(s)$ vyjadřuje pravděpodobnost, že po spuštění procesu z počátečního stavu $s_0$, se nacházím ve stavu $s \in S$ v čase $k = \infty$.
- Přesněji:

$$t_\infty(s) = \lim_{k \to \infty} t_k(s)$$

- Zkoumání chování systému po uplynutí „nekonečno“ kroků.
- Jde o ustálené rozdělení pravděpodobnosti napříč stavy – pokud bychom udělali ještě jeden krok navíc, tak už nezmění.

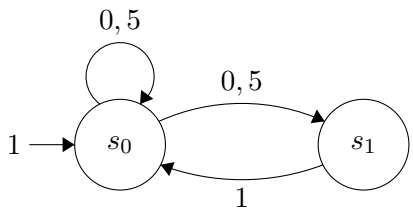
### 12.5.1 Neredukovatelné markovské řetězce

- Celý model je z hlediska teorie grafů silně souvislá komponenta – Z každého stavu se lze dostat do každého dalšího stavu.
- Výsledné pravděpodobnostní rozdělení nezávisí na počátečním stavu.
- Lze spočítat pomocí tzv. balančních rovnic:

$$\forall s \in S : t_\infty(s) = \sum_{s' \in S} t_\infty(s') \cdot P(s', s)$$

$$\sum_{s \in S} t_\infty(s) = 1$$

**Příklad 4** Uvažme následující model.




Obrázek 12.6: Markovův řetězec.

- Sestavíme následující soustavu rovnic:

$$x_0 = 0,5x_0 + x_1$$

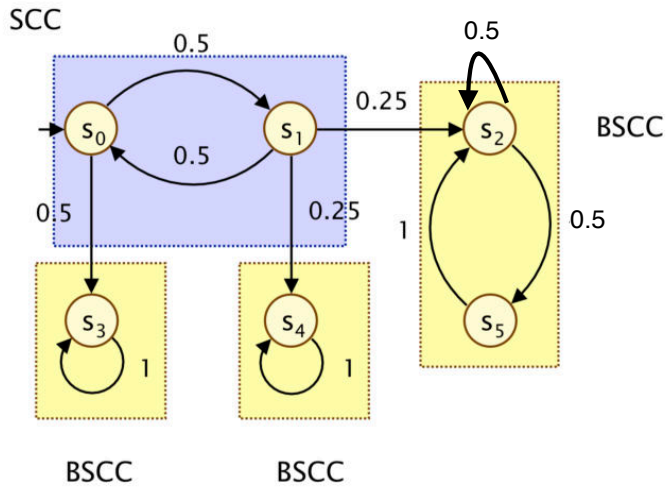
$$x_1 = 0,5x_0$$

- Řešení:

$$x_0 = \frac{2}{3} \; ; \; x_1 = \frac{1}{3}$$

### 12.5.2 Obecné (aperiodické) markovské řetězce

- Model je tvořen několika souvislýma komponentama (SCC).
  - Dále rozlišujeme ještě tzv. *bottom* silně souvislé komponenty (BSCC), to jsou takové, ze kterých už se není možné dostat.
- Výsledné pravděpodobnostní rozdělení závisí na počátečním stavu.




Obrázek 12.7: Příklad markovského řetězce, který je tvořem 4 silně souvislýma komponentama, z čehož 3 jsou *bottom* silně souvislé.

- Stavy dělíme na:
  - Přechodové stavy – stavy mimo BSCC ($s_t$):

$$t_\infty(s_t) = 0$$

  - Rekurentní stavy – stavy uvnitř BSCC ($s_r$):

$$t_\infty(s_r) > 0$$

- Výpočet: Součin pravděpodobnosti, že se dostanu do dané BSCC a pravděpodobnostni, že se dostanu do daného stavu uvnitř BSCC.

## 12.6 Problém dosažitelnosti stavu (*reachability problem*)

- Nechť $T \subseteq S$ je nějaká cílová množina.
- Vysvětlení:
  - $x(s)$ vyjadřuje pravděpodobnost, že se dostanu do stavu $s' \in T$, pokud začínám ve stavu $s \in S$.
- Proč to dělat? Zjistím s jakou pravděpodobností se dostanu do cílových stavů (viz další příklad).
- Postup:
  1. Všechny cílové stavy nastavíme jako tzv. absorbující (pokud ho dosáhneme, tak už ho neopustíme).

$$\forall s \in T : P(s, s) = 1$$

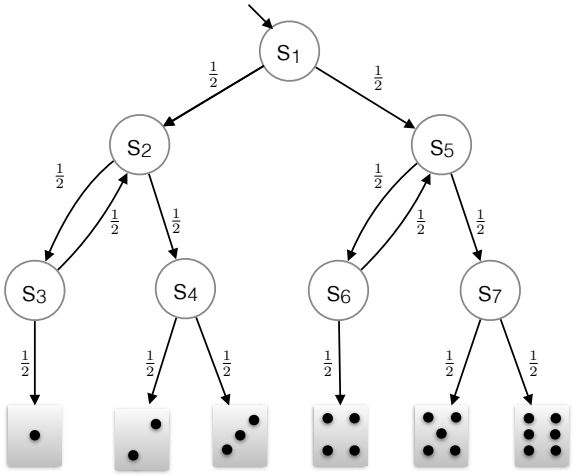
  2. Spočítáme množinu stavů $S_{no} \subset S$, která obsahuje stavy, ze kterých nevede žádná cesta do nějakého stavu z množiny $T$.
  3. Vyřešíme soustavu rovnic:

$$x(s) = 1 \ , \ \forall s \in T$$

$$x(s) = 0 \ , \ \forall s \in S_{no}$$

$$x(s) = \sum_{s' \in S} P(s, s') \cdot x(s') \ , \ \forall s \notin (T \cup S_{no})$$

**Příklad 5** Problém dosažitelnosti stavu pro protokol „Modelování protokolu házení férovou šestistrannou kostkou pomocí férové mince“.




Unbounded reachability to $r_1$

$x(r_1) = 1$

$x(s_3) = 1/2 \cdot x(r_1) + 1/2 \cdot x(s_2)$

$x(s_2) = 1/2 \cdot x(s_3) + 1/2 \cdot x(s_4)$ $= 0$

$x(s_1) = 1/2 \cdot x(s_2) + 1/2 \cdot x(s_5)$ $= 0$

---

$x(s_1) = 1/6$

Obrázek 12.8: Příklad na problém dosažitelnosti. $T = \{r_1\}$.

## 12.7 Očekávaný počet kroků (*expected time to reach a state*)

- Nechť $T \subseteq S$ je nějaká cílová množina.
- Vysvětlení:
  - $e(s)$ vyjadřuje očekávaný (průměrný) počet kroků, že se dostanu do stavu $s' \in T$, pokud začínám ve stavu $s \in S$.
  - Podmínka: pravděpodobnost dosáhnutí $s' \in T$ z $s$ musí být 1.
- Postup:
  - Všechny cílové stavy nastavíme jako tzv. absorbující (pokud ho dosáhneme, tak už ho neopustíme).

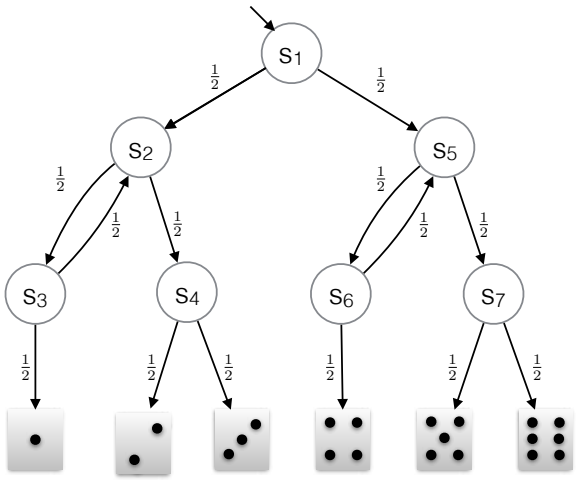
    $$\forall s \in T : P(s, s) = 1$$

  - Vyřešíme soustavu rovnic:

    $$e(s) = 0 \ , \ \forall s \in T$$

    $$e(s) = 1 + \sum_{s' \in S} P(s, s') \cdot e(s') \ , \ \forall s \notin T$$

**Příklad 6** Problém očekávaného počtu kroků pro protokol „Modelování protokolu házení férovou šestistrannou kostkou pomocí férové mince“.




Expected time to reach $r_1 \ldots r_6$

$e(r_i) = 0$ for all $1 \leq i \leq 6$

$$e(s_1) = 1 + 1/2 \cdot e(s_2) + 1/2 \cdot e(s_5)$$
$$e(s_2) = 1 + 1/2 \cdot e(s_3) + 1/2 \cdot e(s_4)$$
$$e(s_3) = 1 + 1/2 \cdot e(r_1) + 1/2 \cdot e(s_2)$$
$$e(s_4) = 1 + 1/2 \cdot e(r_2) + 1/2 \cdot e(r_3)$$
$$e(s_5) = 1 + 1/2 \cdot e(s_6) + 1/2 \cdot e(s_7)$$
$$e(s_6) = 1 + 1/2 \cdot e(r_4) + 1/2 \cdot e(s_5)$$
$$e(s_7) = 1 + 1/2 \cdot e(r_5) + 1/2 \cdot e(r_6)$$

$$e(s_1) = 11/3 \approx 3.67$$

Obrázek 12.9: Příklad na očekávaný počet kroků. $T = \{r_1, r_2, \ldots, r_6\}$.

# Kapitola 13

# MSP – Randomizované algoritmy (Monte Carlo a Las Vegas algoritmy).

## 13.1 Zdroje

- `MSP_13_Randomized_Algorithms.pdf`
- `MSP_2021-12-14_1080p.mp4`

## 13.2 Úvod a kontext

- K čemu jsou randomizované algoritmy?
  - Analyzujeme průměrné chování algoritmů (nikoliv nejhorší případ).
  - Randomizací můžeme dosáhnout snížení očekávané ceny algoritmu.
- Přesahuje rámec analýzy nejlepšího nebo nejhoršího případu.
  - Analýza všech případů (vstupů) pomocí jejich pravděpodobnostního rozdělení.
  - V mnoha případech poskytuje lepší vhled do praktické složitosti.
- Existují dva přístupy k randomizaci:
  - randomizace pořadí vstupů (např. Hiring Problem),
  - randomizace volby provedené v rámci algoritmu (např. Quicksort).

## 13.3 Indikátorová náhodná proměnná (*indicator random variable*)

- Mějme prostor jevů $S$ a událost $A \in S$.
- Indikátorová náhodná proměnná pro $A$:

$$I\{A\} = \begin{cases} 1 & \text{pokud A nastane} \\ 0 & \text{jinak} \end{cases}$$

- Nechť $X_A = I\{A\}$, pak platí, že $E[X_A] = Pr\{A\}$ (pravděpodobnost výskytu události A).
- Hlavní myšlenka: Vyjádřit očekávání náhodné proměnné ($X$) jako očekávání sum komponent, které se snáze počítají (indikátorové proměnné $X_i$).

### 13.3.1 Příklad: Jaký je očekávaný počet padnutí orla při $n$ hodů mincí?

$$X_i = I\{\text{i-tý hod mincí je událost padne orel}\}$$

$$X = \sum_{i=1}^{n} X_i$$

$$E[X] = E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E\left[X_i\right] = \sum_{i=1}^{n} x_i \cdot p(x_i) = \sum_{i=1}^{n} 1 \cdot \frac{1}{2} = \frac{n}{2}$$

## 13.4 Hiring Problem

- Firma chce nabrat nejlepšího zaměstnance.
- Můžeme randomizovat seznam kandidátů.
- Složitost ($c_h$ je cena najmutí nového kandidáta):
  - nejhorší případ: $\mathcal{O}(n \cdot c_h)$
  - nejlepší případ: $\mathcal{O}(c_h)$
- Počet možných uspořádání kandidátů je $n!$, pak řešíme kolik permutací bude mít cenu $1, 2, \ldots, n$.

```
HIRE-ASSISTANT(n)
best = 0            // candidate 0 is a least-qualified dummy candidate
for i = 1 to n
    interview candidate i
    if candidate i is better than candidate best
        best = i
        hire candidate i
```

Obrázek 13.1: Hiring Problem v pseudokódu.

```
RANDOMIZED-HIRE-ASSISTANT(n)
randomly permute the list of candidates
best = 0            // candidate 0 is a least-qualified dummy candidate
for i = 1 to n
    interview candidate i
    if candidate i is better than candidate best
        best = i
        hire candidate i
```

Obrázek 13.2: Hiring Problem randomizovaný v pseudokódu.

### 13.4.1 Příklad: Analýza Hiring Problem pomocí indikátorové proměnné

- Indikátorová proměnná.

$$X_i = I\{\text{i-tý kandidát je přijat}\}$$

- Celkový počet přijatých kandidátů (cena).

$$X = \sum_{i=1}^{n} X_i$$

- Očekávaná cena algoritmu.

$$E[X]$$

- Pravděpodobnost, že i-tý kandidát je přijat (kandidát i je přijat, pokud je lepší, než všichni předchozí).

$$E[X_i] = \frac{1}{i}$$

$$E[X] = E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E\left[X_i\right] = \sum_{i=1}^{n} x_i \cdot p(x_i) = \sum_{i=1}^{n} 1 \cdot \frac{1}{i} = \ln n + \mathcal{O}(1)$$

- Pravděpodobnostní analýza nám dává asymptotický lepší ohraničení ceny

$$\mathcal{O}(\log n \cdot c_h) \quad \text{vs} \quad \mathcal{O}(n \cdot c_h)$$

## 13.5 Las Vegas

- Pro každý vstup dává správné výsledky (korektnost je zaručena).
- Pro každý vstup existuje pravděpodobnost, že doba běhu bude delší než je žádoucí nebo očekávané.
- Analyzujeme čas běhu algoritmu, který odpovídá nějaké náhodné proměnné.

### 13.5.1 Příklad: Je dáno pole $A[1, 2, \ldots, n]$ s $n$ prvky (kde $n$ je sudé). Polovina z prvků obsahuje nuly, druhá polovina obsahuje jedničky. Cíl: Najděte index který obsahuje jedničku. Zkonstruujte Las Vegas algoritmus.

```
def las_vegas(A, n):
    while True:
        i = random_int(0, n)
        if A[i] == 1:
            return i
```

Výpis 13.1: Las Vegas algoritmus v Pythonu.

- Průměrná složitost je $\mathcal{O}(1)$, to můžeme ukázat pomocí očekávané doby běhu s využitím indikátorové proměnné:

$$X_i = \begin{cases} 1 & \text{pokud algoritmus provede i-té porovnání} \\ 0 & \text{jinak} \end{cases}$$

$$X = \sum_{i=1}^{\infty} X_i$$

$$E[X] = E\left[\sum_{i=1}^{\infty} X_i\right] = \sum_{i=1}^{\infty} E\left[X_i\right] = \sum_{i=1}^{\infty} Pr\{X_i\} = \sum_{i=1}^{\infty} \left(\frac{1}{2}\right)^{i-1} = 2 = \mathcal{O}(1)$$

### 13.5.2 Příklad: Analýza Quicksortu.

- Myšlenka randomizace: výběr pivota.

```python
def quicksort(numbers: list[int]) -> list[int]:
    if not numbers:
        return []
    pivot = get_random_elem(numbers)
    less = []
    greater = []
    for number in numbers:
        if number < pivot:
            less.append(number)
        elif number > pivot:
            greater.append(number)
    return [*quicksort(less), pivot, *quicksort(greater)]
```

Výpis 13.2: Quicksort v Pythonu.

- V nejhorším případě: $\mathcal{O}(n^2)$.
  - Pivotem je zvoleno vždy největší nebo nejmenší číslo.
- V nejlepším případě: $\mathcal{O}(n \cdot \log n)$.
  - Pivotem je zvoleno vždy číslo uprostřed.
- Průměrný případ pomocí pravděpodobnostní analýzy:
  - Sestrojení indikátorové proměnné. Vyjadřuje, zda dvojice prvků spolu byla porovnána.
  $$X_{ij} = \begin{cases} 1 & \text{pokud } x_i \text{ a } x_j \text{spolu byly porovnány} \\ 0 & \text{jinak} \end{cases}$$
  - Celkový počet porovnání:
  $$X = \sum_{1 \le i \le j \le n} X_{ij}$$
  $$E[X] = \sum_{1 \le i \le j \le n} Pr[x_i \text{ a } x_j \text{ jsou porovnány}]$$
  - Nechť $y_1, y_2, \ldots, y_n$ jsou vstupní prvky v seřazeném pořadí (suma je komutativní, takže lze provést).
  $$E[X] = \sum_{1 \le i \le j \le n} Pr[y_i \text{ a } y_j \text{ jsou porovnány}]$$
  - *Pro výpočet viz přednášku, je to nad rámec a nebude se zkoušet.*
  $$Pr[y_i \text{ a } y_j \text{ jsou porovnány}] = \frac{2}{j - i + 1}$$
  $$E[X] = \sum_{1 \le i \le j \le n} \frac{2}{j - i + 1} = \ldots = 2n \cdot \ln n = \mathcal{O}(n \cdot \log n)$$

## 13.6 Monte Carlo

- Pro každý vstup existuje pravděpodobnost výskytu chyby (nesprávného výsledku).
- Je zaručena doba běhu algoritmu.
- Pro maximalizaci korektnosti využívá tzv. aplifikaci – pokud algoritmus neskončí korektně, tak se opakuje, opakování je zastropováno konstantou.
  - Myšlenka: mám algoritmus kterej je v podstatě k ničemu (je velmi malá šance, že je korektní), ale hodněkrát ho opakuju.
- Analyzujeme čas běhu a pravděpodobnost korektnosti algoritmu.

### 13.6.1 Příklad: Je dáno pole $A[1, 2, \ldots, n]$ s $n$ prvky (kde $n$ je sudé). Polovina z prvků obsahuje nuly, druhá polovina obsahuje jedničky. Cíl: Najděte index který obsahuje jedničku. Zkonstruujte Monte Carlo algoritmus.

```
def monte_carlo(A, n):
    limit = 1000
    for _ in range(0, limit):
        i = random_int(0, n)
        if A[i] == 1:
            return i
    return None
```

Výpis 13.3: Monte Carlo algoritmus.

- Složitost: $\mathcal{O}(1)$.
- Pravděpodobnost korektnosti: $1 - 0,5^{1000}$

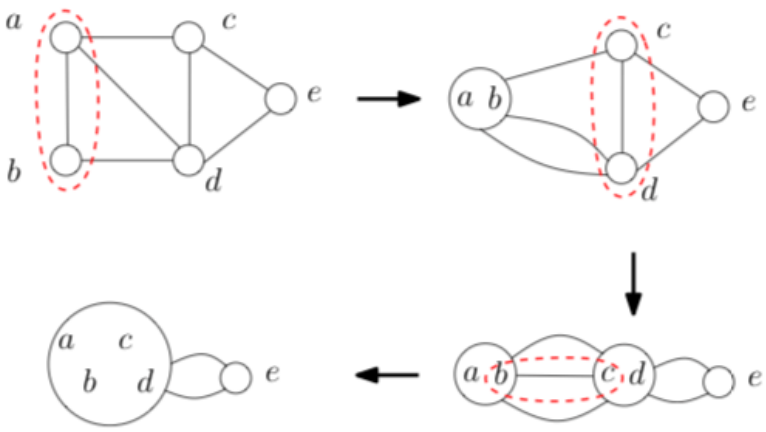
### 13.6.2 Příklad: Problém minimálního řezu v grafu

- Problém minimálního řezu v grafu (*min cut problem*) spočívá v rozdělení množiny uzlů na dvě neprázdné podmnožiny takovým způsobem, že počet hran, které vedou z jedné podmnožiny do druhé je minimální.
- Myšlenka randomizace: náhodně slučujeme vrcholy.

**Algorithm 1:** IntuitiveKarger($G$)

**while** *there are more than 2 supernodes:* **do**
  Pick an edge $(u, v) \in E(G)$ uniformly at random;
  Merge $u$ and $v$;
Output edges between the remaining two supernodes

Obrázek 13.3: Algoritmus v pseudokódu.




Obrázek 13.4: Algoritmus vizualizace.

- Algoritmus má garantovanou složitost $\mathcal{O}(n^2)$ – každé sloučení stojí $\mathcal{O}(n)$.
- Šance, že algoritmus je v tomto stavu korektní, je velmi malá $\Theta(\frac{1}{n^2})$, proto je nutné zavést amplifikaci.
  - Zavedeme konstantu $c$ a algoritmus poběží:

$$c \cdot n^2 \cdot \ln n$$

  - Složitost:

$$\mathcal{O}(n^4 \log n)$$

  - Pravděpodobnost korektního běhu pak je:

$$1 - \frac{1}{n^c}$$

- Reálné použití spočívá typicky v kombinaci randomizace a *brute force* přístupu. Na začátku randomizujeme, protože je menší šance, že bude porušena korektnost. Tím se problém redukujeme na menší (v případě min-cut se graf zmenšuje). Čím menší instance problému, tím je vyšší pravděpodobnost porušení korektnosti, ale zároveň se cena *brute force* řešení snižuje. Tedy, až je instance problému dostatečně malá (graf je dostatečně malý), tak provedeme jeho vyřešení *brute force* algoritmem (který je korektní).

# Kapitola 14

# SUI – Problém generalizace strojového učení a přístup k jeho řešení (trénovací, validační a testovací sada, regularizace, předtrénování, multi-task learning, augmentace dat, dropout, ...).

## 14.1 Zdroje

- `08-basics_in_ml.pdf`
- `SUI_2019-10-21.mp4`
- `SUI_2019-11-04.mp4`
- `Wikipedia`

## 14.2 Strojové učení

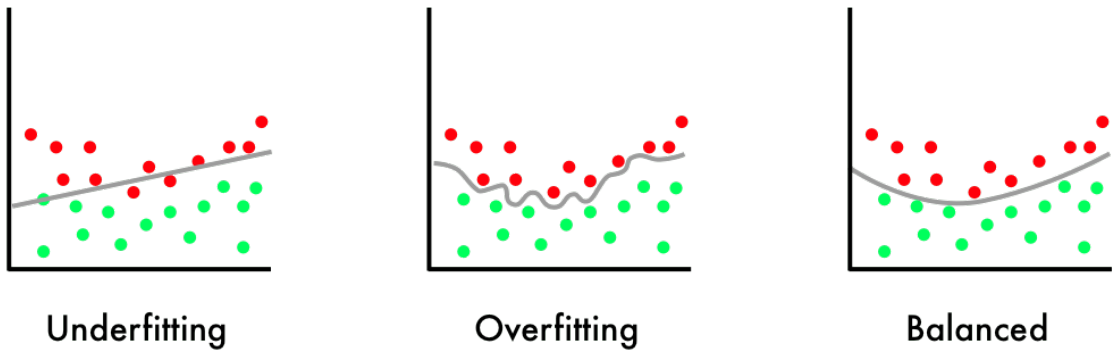
- Strojové učení je podoblastí umělé inteligence, zabývající se takovými algoritmy, které se dokáží samy učit – zlepšují se na základě svých zkušeností a využívání dat, aby dosáhly vytyčeného cíle.
- Typicky je vytvářen (trénován, učen) model na základě trénovacích dat, který je poté schopen provádět předpovědi nebo dělat rozhodnutí, aniž by k tomu byl explicitně naprogramován.
- Hlavní přístupy k vytváření modelů jsou:
  - **Učení s učitelem** (*supervised learning*)
    - Model je učen na označených trénovacích datech, která se skládají z dvojic reprezentace vstupního objektu (vektoru příznaků) a požadovaného výstupu.
    - Např. klasifikace, regrese.
  - **Učení bez učitele** (*unsupervised learning*)
    - Model je učen na neoznačených datech.
    - Algoritmy se snaží vytvořit vhodnou a kompaktní reprezentaci vstupních

dat (snížení dimenzionality, vyhlazování, diskretizace), ve které poté hledají charakteristické rysy a podobnosti.
    * Např. shlukování, hledání anomálií (detekce odlehlých hodnot).
  - **Zpětnovazebné učení** (*reinforcement learning*)
    * Model je učen na neoznačených datech ale dostává zpětnou vazbu, typicky jednou za nějaký delší časový úsek.
    * Např. hraní her (po odehrání jedné šachové partie model dostane zpětnou vazbu, jak se mu dařilo).

- Trénovací algoritmy se obvykle snaží minimalizovat nějakou **chybovou funkci** (resp. objektivní funkci, *objective function*), např.
  - MLE (*Maximum likelihood estimation*) – pro učení se pravděpodobnostních rozložení.
  - Metoda nejmenších čtverců (*Least squares*) – pro regresi.
  - Křížová entropie (*Cross-entropy*) – pro klasifikaci.
- Často objektivní funkce vrací chybu pro celý dataset, a chybová funkce chyby pro konkrétní datový vzorek.
- Komplikovanější modely (např. neuronové sítě) využívají jiné, složitější optimalizační strategie (např. gradient descent).

## 14.3 Problém generalizace

- **Generalizace** je vlastnost modelu strojového učení správně fungovat i na datech, na kterých nebyl trénován.
  - V případě klasifikace to znamená rozpoznávat co nejlépe i doposud neviděná data (data, na kterých model nebyl trénován).
- **Přetrénování** (*overfitting*) – Model se příliš specializuje na trénovací data, negeneralizuje. Trénování chceme ukončit včas, předtím, než se model začne přetrénovávat.
- **Podtrénování** (*underfitting*) – Model je málo natrénovaný, trénování by ještě mělo pokračovat.




Obrázek 14.1: Podtrénování, přetrování a vhodné natrénování modelu pro klasifikaci.

- Jak dosáhnout toho, aby model generalizoval?

### 14.3.1 Rozdělení trénovací sady

- **Trénovací sada**
  - Slouží k procesu trénování modelu.
  - Typicky proces trénování opakujeme v iteracích, sledujeme hodnotu chybové funkce a upravujeme parametry modelu.
- **Validační sada**
  - Používá se v průběhu trénování k odhalení přetrénování modelu na trénovacích datech.
  - Sledujeme hodnotu chybové funkce na trénovací a validační sadě. Pokud se chyba na trénovací sadě zmenšuje, ale na validační roste, model začíná být přetrénovaný.
  - S trénovací datovou sadou je obvykle v poměru 1 : 4.
- **Testovací sada**
  - Testovací sada slouží pro otestování chování modelu v reálných podmínkách, není vůbec součástí procesu trénování.
  - Po veškerých experimentech s trénovací a validační datovou sadou je provedeno jediné vyhodnocení modelu na těchto datech.
  - Někdy může být validační a testovací sada sjednocena a neděláme mezi něma rozdíl.

### 14.3.2 Regularizace

- Metody jak zabránit přetrénování, a to přidáním náhodného šumu, změnou hodnot parametrů modelu (penalizace extrémních hodnot), přidání regularizačních členů do chybové funkce funkcí, ...
- **Regularizace vah**
  - Myšlenka: můžu omezit váhy, nedopustím příliš velké hodnoty.
  - K chybové funkci přičítám složku, která mi říká, jak moc velké jsou váhy (např. velikost vah na druhou).
- **Regularizace aktivací**
  - Myšlenka: můžu omezit aktivaci vrstvu.
  - K chybové funkci přičítám složku, která mi říká, jak moc velké jsou hodnoty které vstupují do aktivační funkce.
- Formálně; $J$ je objektivní funkce, $\theta$ jsou parametry modelu, $D$ je dataset (s označenými daty – učení s učitelem), $loss$ je chybová funkce, $f$ je model, $\lambda$ je funkce, která vrátí „penalizaci“ na základě velikosti vah a $\beta$ je funkce, která vrátí „penalizaci“ na základě velikosti aktivací.

$$J(D, \theta) = \sum_{x,y}^{D} loss(f(x), y) + \lambda(|\theta|)^2 + \beta(|activation|)^2$$

### 14.3.3 Předtrénování

- Způsob iniciálního nastavení parametrů modelu učení bez učitele, kdy nastavujeme parametry modelu pouze na základě vstupů na obecné datové sadě.
- Dotrénování následně proběhne na specifické datové sadě pomocí učení s učitelem.
- *Nikde jsem o tomto nenašel více informací, možná to souvisí s finetuningem.*

### 14.3.4 Finetuning

- Využití modelu, který byl natrénován pro podobnou úlohu a jeho dotrénování pro náš specifický problém.
  - Může být přetrénována pouze část modelu, jeho konec např.
- Využívá se typicky v neuronových sítí, např. podchytávání nějakých vzorů v obrázcích je stejné a liší se až v poslední vrstvě.
- Výhody: stačí menší dataset, méně výpočetního výkonu, model bude i dobře generalizovat.

### 14.3.5 Multi-task learning

- Taková úloha strojového učení, ve které se řeší více učebních úloh současně, přičemž se využívají společné rysy a rozdíly mezi úlohami (model se naráz učí na několik úloh).
- To může vést ke zvýšení efektivity učení a přesnosti předpovědí pro modely specifické pro danou úlohu ve srovnání s tím, pokud by modely byly trénovaný odděleně.
- MTL zlepšuje generalizaci modelu tím, že využívá informace o doméně obsažené v tréninkových signálech příbuzných úloh jako induktivní zkreslení. Děje se tak paralelním učením úloh při použití sdílené reprezentace.
- V kontextu klasifikace je cílem MTL zlepšit přenost více klasifikačních úloh jejich společným učením.
  - Konkrétní příklad může být spamový filtr, který lze považovat za mnoho různých klasifikačních úloh pro každého uživatele. Různí lidé sledují různé příznaky, na základě kterých rozlišují nevyžádané e-maily od legitimních. Například anglicky mluvící člověk může zjistit, že všechny e-maily v ruštině jsou spam, u rusky mluvících lidí tomu tak není. Přesto existuje v této klasifikační úloze určitá podobnost mezi uživateli, například jedním společným rysem může být text týkající se převodu peněz. Společné řešení úlohy klasifikace spamu každého uživatele prostřednictvím MTL může umožnit, aby se řešení vzájemně informovala a zlepšila výkonnost.

### 14.3.6 Augmentace dat

- Většina modelů strojového učení by dosáhla lepších výsledků, pokud by bylo k dispozici více trénovacích dat. Ovšem jejich získání je často poměrně nákladné.
- Augmentace dat je technika používaná ke zvýšení množství trénovacích dat přidáním mírně upravených kopií již existujících dat, nebo nově vytvořených syntetických vzorků z existujících dat.

- Typicky jde o operace: rotace, změna rozlišení, změna barev, změna kontrastu a jasu, přidání šumu, provádění výřezů, ...

### 14.3.7 Dropout vrstva

- Jde o vrstvu v neuronových sítí.
- Cílem je snížit náchylnost sítě k přetrénování.
- Spočívá v nastavení výstupu každého neuronu ve skryté vrstvě na 0 s pravděpodobností $p$ (často 50 %).
- K deaktivaci neuronů dochází u každého vzorku dat při trénování, do učení je tak uměle vnášen šum, díky kterému je síť schopna více generalizovat.
- Po ukončení trénování se síť navrátí do původního stavu a všechny váhy vedoucí z výstupu každého neuronu, resp. vstupu, se vynásobí pravděpodobností $p$, tj. pravděpodobností s jakou byl neuron ponecháván pro učení v jednotlivých minidávkách.
- Dropout vrstva zvýšuje počet potřebných iterací k dosažení konvergence.

# Kapitola 15

# SUI – Generativní modely a diskriminativní přístup ke klasifikaci (gaussovský klasifikátor, logistická regrese, ...).

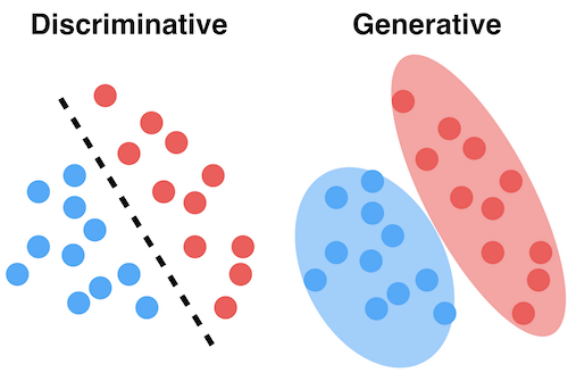
## 15.1 Zdroje

- `08-basics_in_ml.pdf`
- `SUI_2019-11-11.mp4`
- `SUI_2019-11-18.mp4`

## 15.2 Úvod a kontext

- Klasifikace je druh problému, kde je cílem zařadit nový vzorek do jedné nebo více kategorií na základě množiny trénovacích dat, která obsahuje vzorky, jejichž kategorie je známa.
- Rozdělení klasifikátorů dle typu modelu:
  - **Generativní modely**
    - Modelují přímo rozložení hustoty pravděpodobnosti.
    - Většinou nemají takový problém s přetrénováním.
    - Modulární – sestavíme jednoduché modely, které popisují nějaký konkrétní fenomény v datech a ty poté můžeme skládat dohromady.
  - **Diskriminativní modely**
    - Modelují přímo rozhodovací hranici.
    - Menší plýtvání parametry – učíme se přímo rozhodovací hranici.
    - Většinou fungují dobře, pokud máme hodně trénovacích dat.
    - Umožňuje end-to-end řešení.
- Rozdělení klasifikátorů dle popisu:
  - **Parametrický klasifikátor**
    - Klasifikátor lze popsat parametry.

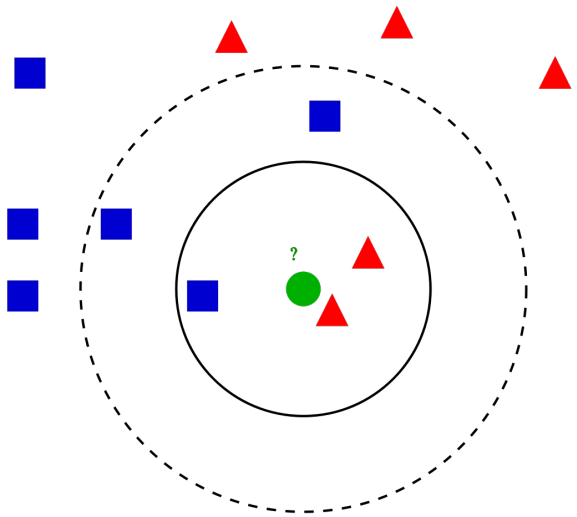
    - Např. polynom, pravděpodobností rozdělení, ...
  - **Neparametrický klasifikátor**
    - Klasifikátor nelze popsat parametry, repektive všechna trénovací data jsou parametrem.
    - Např. K-nejbližších sousedů.




Obrázek 15.1: Podstata diskriminativních a generativních modelů.

### 15.2.1 K-nejbližších sousedů

- V n-dimenzionálním prostoru máme trénovací data, o kterých víme jaké třídě náleží (jedná se o učení s učitelem).
- Pokud chceme klasifikovat nový vzorek tak pomocí některé distanční metriky (Euklidovská, Hammingova, ...) spočítáme jeho K nejbližších sousedů.
- Na základě toho jaké třídě náleží nejvíce sousedů, rozhodneme do které třídy náš vzorek přiřadíme.
- Výhody:
  - Dokážeme využít tzv. měkkého rozhodování, tedy rozlišovat kolik ze sousedů náleží jaké třídě a tím neurčovat pouze binárně ano/ne, ale můžeme říct s jakou pravděpodobností náleží najé třídě.
- Nevýhody:
  - Výpočetní náročnost (není výpočetně efektivní).
  - Musíme mít k dispozici všechny trénovací data, abychom mohli klasifikovat.
  - Veličiny je nutné normalizovat (záleží na distanční metrice) – převést do stejného dynamického rozsahu (např. vydělit maximem, pomocí variace a standardní odchylky).




Obrázek 15.2: Příklad klasifikace pomocí metody k-nejbližších sousedů.

## 15.3 Generativní modely klasifikátorů

- Z testovacích dat (z pozorování) spočítáme parametry pro každou třídu do které chceme klasifikovat – $p(features \mid class)$
- Poté spočítáme apriorní pravděpodobnost třídy – $p(class)$.
- Pomocí bayesova vzorce odvodíme posteriorní pravděpodobnost každé třídy pro daný příznak – $p(class \mid features)$.
- Výhody:
  - Jsou robustnější, odolnější vůči přetrénování.
  - Stačí méně trénovacích dat.
- Nevýhody:
  - Snažíme se přesně modelovat model (funkci hustoty pravděpodobnosti) – $p(features \mid class)$. Ale i tam, kde to nepotřebujeme, tedy tam kde nesousedí s žádnou jinou třídou. Nás primárně zajímá hranice těch tříd.
  - Což je nevyužívání potenciálu parametrů (např. stavíme komplexní systém nad jednoduchým)
  - Další problém nastává, pokud data nejsou Gaussovsky rozložená.
- Modely:
  - Maximum a-posteriori klasifikátor (MAP) – Gaussovský klasifikátor
- Aplikace bayesovy věty:

Posterior probability | Likelihood | Prior probability

$$P(grenade|heavy) = \frac{P(heavy|grenade)P(grenade)}{P(heavy)}$$

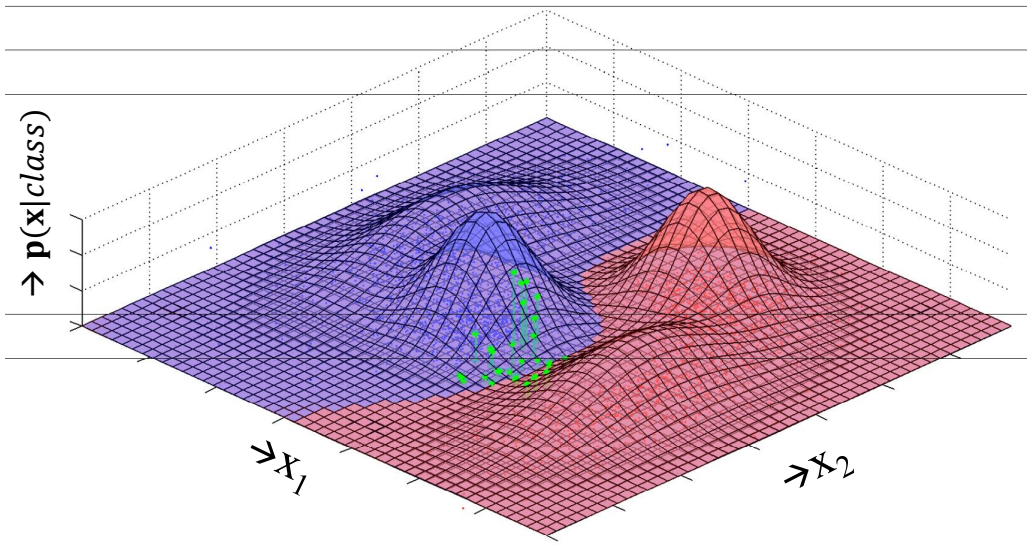
Evidence

Obrázek 15.3: Bayesovský teorém o podmíněné pravděpodobnosti.

- Apriorní pravděpodobnost (prior probability) – Spočítáme jako marginální pravděpodobnost třídy na trénovacích datech.
- Věrohodnost (likelihood) – Spočítáme jako podmíněnou pravděpodobnost.
- Evidence – Spočítáme pomocí sum rule.
- Posterirní pravděpodobnost (posterior probability) – Výsledek, s jakou pravděpodobností vzorek náleží jaké třídě,

### 15.3.1 Gaussovský klasifikátor

- Gaussovský klasifikátor, resp. MAP (*maximum aposteriori probability*) klasifikátor.




Obrázek 15.4: Příklad gaussovského klasifikátoru pro dvourozměrná data. Data jsou klasifikována do dvou tříd. Každá třída je modelována dvěma normálníma rozděleníma.

- Při trénování modelujeme normální rozložení pravděpodobnosti v datech pro jednotlivé třídy.
  - Rozložení může být až n-rozměrné v závilosti na množství příznaků – tzv. vícerozměrné gaussovo rozložení.
  - Třída může být modelována více než jedním normálním rozložením – tzv. multivariantní gaussovo rozložení.
- Pro klasifikaci je využívána teorie pravděpodobnosti, resp. bayesův teorém a je počítána pravděpodobnost náležitosti do určité třídy.

- Cílem trénování je získat pravděpodobností funkci (parametry normálních rozdělení) pro jednotlivé třídy.

$$p(x) = \mathcal{N}(x, \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \cdot e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

- Při trénování provádíme tzv. odhad s maximální věrohodností (*maximum likelihood estimation*) pro střední hodnotu a rozptyl. Objektivní funkce vypadá následovně:

$$\arg\max_{\eta} \; p(X \mid \eta) = \arg\max_{\eta} \prod_{i=1}^{n} p(x_i \mid \eta) = \arg\max_{\mu,\sigma^2} \prod_{i=1}^{n} \mathcal{N}(x_i, \mu, \sigma^2)$$

- Z objektivní funkce lze odvolit, že nejlepší odhady parametrů střední hodnotu a rozptylu jsou bodové odhady:

$$\mu = \frac{1}{n} \sum_{i=1}^{n} x_i$$

$$\sigma^2 = \frac{1}{n} \sum_{i=1}^{n} (x_i - \mu)^2$$

- Jak probíhá klasifikace?
  - Mějme dvě třídy $\omega_1$ a $\omega_2$.
  - Pro daný datový vzorek a jeho příznak (vektor příznaků) $x$, vyber třídu $\omega_i$ s větší posteriorní pravděpodobností $P(\omega_i \mid x)$.
  - Vyber $\omega_1$ pokud:

$$P(\omega_1 \mid x) > P(\omega_2 \mid x) \;\Rightarrow$$

$$\Rightarrow \frac{P(\omega_1) \cdot P(x \mid \omega_1)}{P(x)} > \frac{P(\omega_2) \cdot P(x \mid \omega_2)}{P(x)} \;\Rightarrow \tag{15.1}$$

$$\Rightarrow P(\omega_1 \;,\; x) > P(\omega_2 \;,\; x)$$

## 15.4 Diskriminativní modely klasifikátorů

- Snažíme se z trénovacích dat neodhadovat celé funkce hustoty pravděpodobnosti, ale pouze hranice mezi třídami.
- Data nemusejí být Gaussovsky rozdělená (resp. každá třída odpovídat nějakému pravděpodobnostímu rozdělení).
- Učíme se rovnou $p(class \mid features)$, případně s žádnou pravděpodobností vůbec nepočítáme.
- Výhody/nevýhody:
  - Menší plýtvání parametry.
  - Obvykle vyšší výkon s dostatečným množstvím dat.
  - Umožňuje „end-to-end“ řešení.
- Modely:

- Lineární logistická regrese
- Support Vector Machine
- Neuronové sítě

### 15.4.1 Lineární logistická regrese

- Rozhodovací hranici mezi třídami modelujeme pomocí regresní funkce (lineární) a její výstup dáme na vstup jiné funkce, která vrátí pravděpodobnost náležitosti dané třídě (hodnotu z intervalu $\langle 0, 1 \rangle$).
  - Takovou funkcí může být např. sigmoid:

$$f(x) = \frac{e^x}{e^x + 1}$$

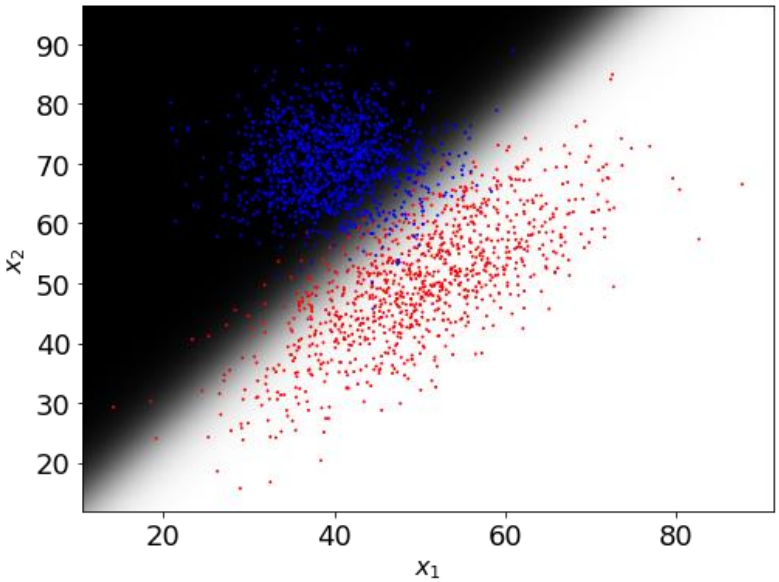
- Jak probíhá klasifikace?
  - Pravděpodobnost, že datový vzorek $\mathbf{x}$ (reprezentovaný vektorem 2 příznaků) náleží třídě $\omega_1$.
  - Spočítáme standardní lineární regresi a výstup dáme na vstup funkci sigmoid, která už vrátí hodnoty z intervalu $\langle 0, 1 \rangle$.
  - Výpočet probíhá jako lineární kombinace dvou lineárních funkci (máme 2 příznaky, ale klidně bychom mohli použít polynom vyššího řádu).
  - Vektor $\mathbf{w}$ reprezentuje natrénované váhy.

$$P(\omega_1 \mid \mathbf{x}) = \sigma(\hat{\mathbf{x}}^{\mathbf{T}} \cdot \mathbf{w}) = \sigma(w_1 \cdot x_1 + w_2 \cdot x_2 + w_0)$$

- Jak probíhá trénování?
  - Jak naučit vektor $\mathbf{w}$? Opět lze použít odhad s maximální věrohodností (MLE, *maximum likelihood estimation*).
  - Akorát nemáme Gaussovo, ale Bernoulliho rozdělení.
  - $c_i$ kóduje správnou třídu, tj. má hodnotu 1 nebo 0.
  - Tuto funkci chceme zmaximalizovat pro trénovací data, tedy najít maximální hodnoty vektoru $\mathbf{w}$.

$$P(\mathbf{c} \mid \mathbf{X}) = \prod_{i=1}^{n} P(c_i \mid \mathbf{x}_i) = \prod_{i=1}^{n} \sigma(\hat{\mathbf{x}}_{\mathbf{i}}^{\mathbf{T}} \cdot \mathbf{w})^{c_i} \cdot (1 - \sigma(\hat{\mathbf{x}}_{\mathbf{i}}^{\mathbf{T}} \cdot \mathbf{w}))^{(1-c_i)}$$

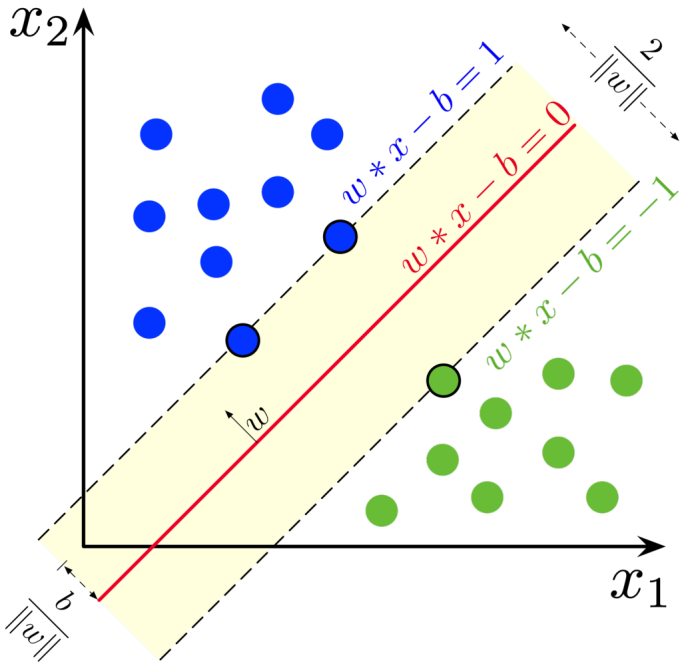
  - Objektivní funkci MLE lze transformovat na chybovou funkci Cross Entropy (CE).




Obrázek 15.5: Příklad natrénované hranici pro klasifikaci s využitím logistické regrese.

### 15.4.2 Support Vector Machine

- Myšlenka spočívá v tom, že najdeme dvě rovnoběžky mezi datovými třídami a uprostřed rovnoběžek vedeme přímku, která tvoří rozhodovací hranici.
- Provádí tvrdé rozhodování (žádná pravděpodobnost náležitosti dané třídě).
- Pokud jsou data prolnutý skrz nějaký gaussovský šum, tak SVM může mít problémy.




Obrázek 15.6: Příklad SVM klasifikátoru.

# Kapitola 16

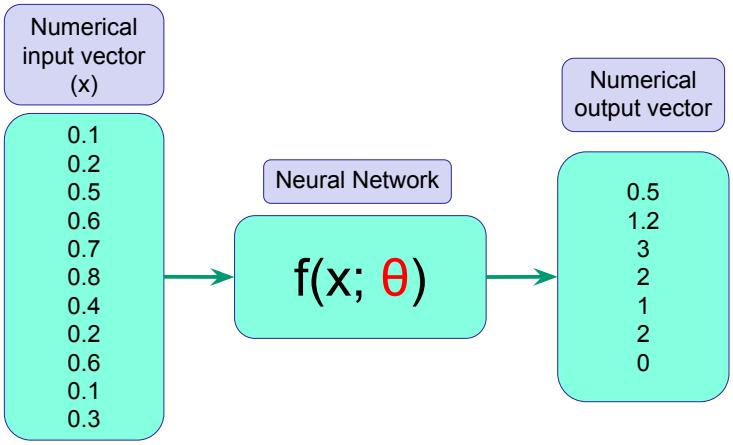
# SUI – Neuronové sítě a jejich trénování (metoda gradientního sestupu, účelová (loss) funkce, výpočetní graf, aktivační funkce, zápis pomocí maticového násobení, ...).

## 16.1 Zdroje

- `09-neural_networks.pdf`
- `SUI_2019-11-25.mp4`
- `SUI_2019-12-02.mp4`

## 16.2 Neuronové sítě

- Neuronové sítě jsou jedním z výpočetních modelů používaných v umělé inteligenci, resp. strojovém učení.
- Na nejvyšší abstrakci lze na neuronovou síť pohlížet jako na funkci, která dostane na vstup vektor (matici) reálných čísel a vrátí jiný vektor reálných čísel.

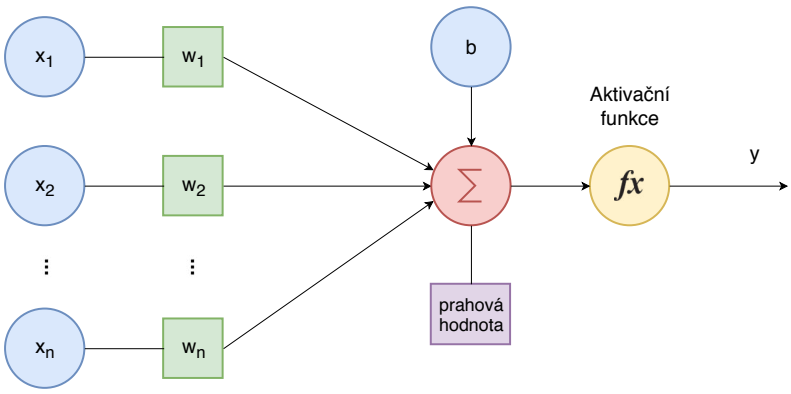



Obrázek 16.1: Neuronová síť jako funkce, $x$ je vektor vstupů, $\theta$ jsou parametry, které je třeba se naučit.

### 16.2.1 Perceptron

- Základní jednotka neuronové sítě se nazývána jako perceptron (také neuron).
- Vstup perceptronu je získáván z výstupu jiných neuronů, či externího zdroje, pokud se jedná o perceptron ve vstupní vrstvě.
- Každému vstupu je přiřazen váhový koeficient, který určuje jeho relativní důležitost v porovnání s ostatními.
- Celkový výstup je vypočítán pomocí váženého součtu všech vstupů, aplikací aktivační funkce a přičtení zarovnávací hodnoty (*bias*).

$$y = f(b + \sum_{i=1}^{n} w_i x_i) \tag{16.1}$$

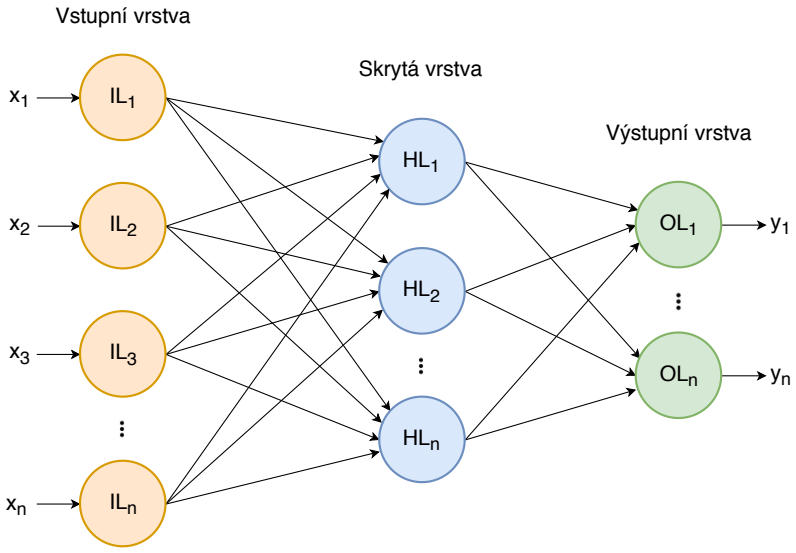



Obrázek 16.2: Perceptron.

### 16.2.2 Vícevrstvé neuronové sítě

- Použití pouze jednoho perceptronu není příliš efektivní, v praxi se setkáváme s vícevrstvými neuronovými sítěmi (*multi layer perceptron*).

- Jsou tvořeny opakováním perceptronu, které jsou na sebe napojeny ve vrstvách (počet vrstev a způsob propojení určuje architekturu sítě).
- Počet vstupních neuronů je dán počtem vstupů matematického modelu.
- Počet neuronů ve skryté vrstvě je volen s ohledem na složitost úlohy. Čím složitější úloha, tím více neuronů je potřeba pro naučení se potřebných příznaků a pochopení struktury vzoru.
- Počet výstupních neuronů obvykle odpovídá počtu klasifikačních tříd (v případě klasifikační úlohy).
- Pokud má neuronová síť více skrytých vrstev, pak o ní hovoříme jako o hluboké neuronové síti (*deep neural network*).
- Jednotlivé skryté vrstvy mohou mít různé formy specializace (konvoluční, pooling, dropout, rekurentní, ...)




Obrázek 16.3: Vícevrstvé neuronové sítě.

### 16.2.3 Aktivační funkce

- Aktivační funkce (také přenosová) slouží pro výpočet výstupní hodnoty neuronu v závislosti na jeho vektoru vstupních hodnot.
- Proč jsou aktivační funkce potřeba:
  - Přidávají do výpočtu nelinearitu – bez nich, by bylo možné libovolně velkou neuronovou síť vyjádřit jako jednu lineární fukci!
  - Bez aktivačních funkcí by výstup neuronu mohla být jakákoliv hodnota. Aktivační funkce poskytuje tedy jisté hranice, mezi kterými může neuron produkovat výstup.
- Volba aktivační funkce má výrazný vliv na dobu učení (trénování) neuronové sítě.
- Např. sigmoid, ReLU (*rectified linear unit*), jednotkový skok, hyperbolický tangens, ...

## 16.3 Trénování neuronové sítě

- Cílem trénování (učení) neuronových sítí je nastavit parametry (váhy $w$) tak, aby poskytovaly co nejpřesnější výsledky.
- V průběhu trénování se váhové koeficienty postupně mění a to takovým způsobem, aby nakonec poskytovaly správné hodnoty výstupního signálu na dané vstupní signály.
- Po procesu naučení neuronové sítě lze na síť pohlížet jako na *black box*, který je vhodná k nasazení ve zvolených aplikačních rovinách.
- Neuronové sítě mohou být učeny s učitelem i bez, v závislosti na dané úloze strojového učení (klasifikace, regrese, shlukování, hledání anomálií, ...)

### 16.3.1 Inicializace vah

- Za inicializaci vah se považuje proces před samotným učením neuronové sítě, kdy se váhám a zarovnáním přiřadí výchozí hodnoty.
- Vhodné počáteční hodnoty mohou proces trénování sítě velmi urychlit.
- Váhy mohou být inicializovány na základě nějakého normálního rozdělení a následně vynásobeny nějakým výrazem (záleží na úloze).

### 16.3.2 Chybové funkce

- Chybová funkce, také objektivní funkce, anglicky *loss function*.
- Pro proces učení neuronové sítě je nutný algoritmus, který provádí změnu vah ($w$) a výši zkreslení (*bias*, $b$). Obecně až do takové fáze, kdy při vstupu dat z datasetu výstup odpovídá očekávanému výsledku.
- Ke zjištění toho, do jaké míry se výstup sítě liší od očekávaného výsledku, se využívá chybových funkcí.
- Z jejich výstupu je určeno, jakým směrem a zhruba o kolik je nutné změnit váhové koeficienty jednotlivých vstupů.
- Správný průběh učení neuronové sítě lze pozorovat z hodnot chybové funkce. Ty by se měly každou iteraci snižovat a síť by tak měla lépe aproximovat funkci specifickou pro daný úkol.
- **Mean squared error** (MSE) je základní funkce pro výpočet chyby, $\hat{Y}$ je vektor predikcí, $Y$ je vektor očekávaných výsledků, $n$ velikost těchto vektorů. Díky umocnění se eliminují záporné hodnoty a dojde ke zvýraznění větších chyb.

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 \tag{16.2}$$

- **Cross entropy loss** (CE) je chybová funkce, $x$ značí vstupní vektor o velikosti $n$, $y(x_i, w)$ je výstup z neuronové sítě za použití vah $w$ a $t_j$ je očekávaný výstup. Funkce se využívá v případech, kdy výstup může nabývat hodnot z několika tříd, které nejsou vzájemně výlučné.

$$\mathrm{CE} = \sum_{i=1}^{n}\sum_{j=1}^{m} t_j \cdot \ln y(x_n, w) \tag{16.3}$$

### 16.3.3 Zpětné šíření chyby

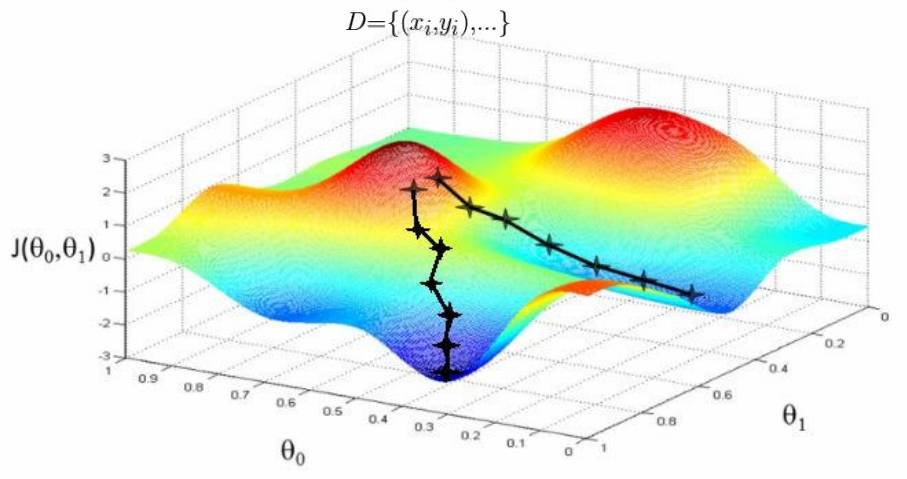
- Zpětné šíření chyby (*backpropagation*) je adaptační algoritmus, pomocí kterého lze vypočítat podíl jednotlivých neuronů na celkové chybě sítě a upravit váhy jednotlivých neuronů tak, aby byla minimalizována celková chyba.
- Jedná se o nejrozšířenější adaptační algoritmus vícevrtsvých neuronových sítí.
- Samotný algoritmus se skládá ze tří etap.
  - Nejprve je vstupní signál šířen sítí dopředu (*feedforward*), ze vstupní vrstvy do vrstev skrytých, kde je spočítán váhový součet všech vstupů a dle aktivační funkce je výstup neuronů směrován do dalších vrstev. Tímto způsobem je signál propagován až do vrstvy výstupní, kde je spočítána celková chyba dle chybové funkce.
  - Dále je chyba šířena neuronovou sítí zpětně od vrstev vyšších k vrstvám nižším. Každému neuronu je spočítána korekce vah.
  - Ty jsou nakonec aktualizovány a celý proces se opakuje s dalšími vstupy.

### 16.3.4 Gradientní sestup

- Gradientní sestup (*gradient descent*) je iterační optimalizační algoritmus prvního řádu pro nalezení lokálního minima diferencovatelné funkce.
  - V případě trénování neuronové sítě jde o hledání (lokálního) minima chybové funkce.
- Jeho podstatou je opakované provádění kroků v opačném směru, než je gradient (nebo přibližný gradient) funkce v aktuálním bodě, protože to je směr nejstrmějšího sestupu.
  - Naopak kroky ve směru gradientu povedou k lokálnímu maximu této funkce; postup se pak nazývá gradientový výstup.
- Tato technika se používá jako rozšíření algoritmů zpětného šíření používaných k trénování umělých neuronových sítí.

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial J(D,\theta)}{\partial \theta_j}$$

$$J(D,\theta) = \sum_{D=\{(x_i,y_i),\ldots\}} loss\left(f\left(x_i,\theta\right), y_i\right)$$

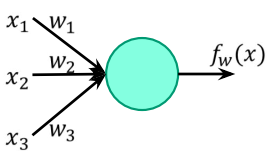



Obrázek 16.4: Gradientní sestup, $J$ značí objektivní funkci (chybu pro celý dataset), $D$ je dataset, $\theta$ jsou parametry.

## 16.4 Vyjádření neuronové sítě

- Při práci s neuronovými sítěmi abstrahujeme od jednotlivým neuronů, vrstev, aktivačních funkcí, ...
- Celá neuronová síť lze vyjádřit maticovým násobením a nebo grafem (orientovaný, acyklický).

### 16.4.1 Maticové násobení

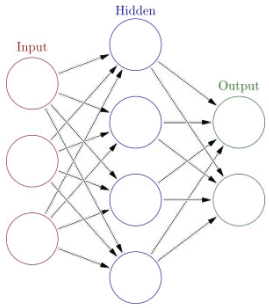



$$f(\mathrm{x};\mathrm{w}) = \mathrm{w}_1\mathrm{x}_1 + \mathrm{w}_2\mathrm{x}_2 + \mathrm{w}_3\mathrm{x}_3 = \sum_i \mathrm{w}_i\mathrm{x}_i$$

$$\rightarrow \begin{bmatrix}1 & 0 & 3\end{bmatrix} \times \begin{bmatrix}2\\-1\\-1\end{bmatrix} = -1$$

Obrázek 16.5: Vyjádření perceptronu pomocí násobení vektorů.

$$\begin{bmatrix}1 & 0 & 3\end{bmatrix} \times \begin{bmatrix}2\\-1\\-1\end{bmatrix} = -1$$




$$\begin{bmatrix}1 & 0 & 3\end{bmatrix} \times \begin{bmatrix}2\\-1\\-1\end{bmatrix} \begin{bmatrix}0\\2\\0\end{bmatrix} \begin{bmatrix}-1\\0\\0\end{bmatrix} \begin{bmatrix}1\\1\\0\end{bmatrix} = -1, 0, -1, -1$$

Obrázek 16.6: Vyjádření třívrstvé neuronové sítě pomocí násobení vektorů.

$$\begin{bmatrix}1 & 0 & 3\end{bmatrix} \times \begin{bmatrix}2\\-1\\-1\end{bmatrix} \begin{bmatrix}0\\2\\0\end{bmatrix} \begin{bmatrix}-1\\0\\0\end{bmatrix} \begin{bmatrix}1\\1\\0\end{bmatrix} = -1, 0, -1, -1$$

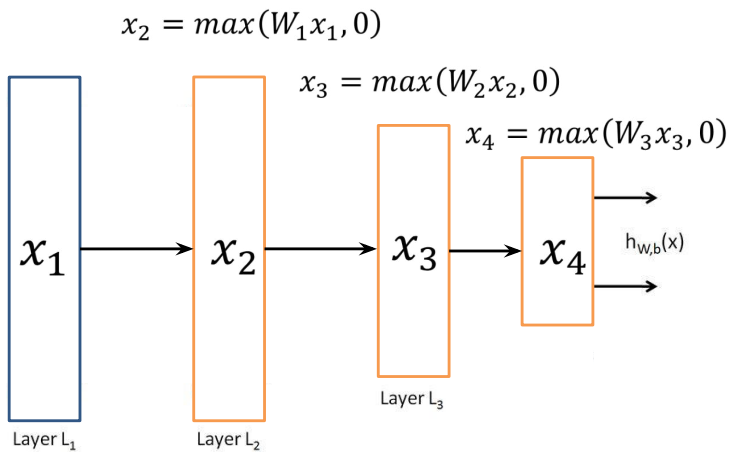
$$\downarrow$$

$$\begin{bmatrix}1 & 0 & 3\end{bmatrix} \times \begin{bmatrix}2 & 0 & -1 & 1\\-1 & 2 & 0 & 1\\-1 & 0 & 0 & 0\end{bmatrix} = \begin{bmatrix}-1 & 0 & -1 & -1\end{bmatrix}$$

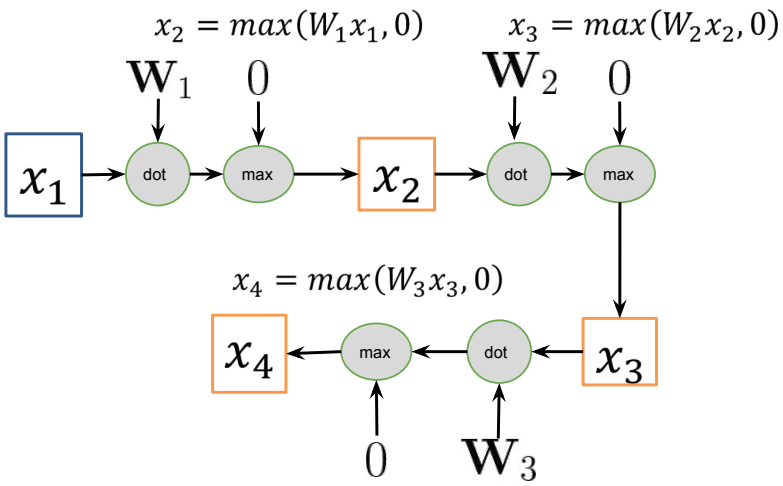
$$f(\mathrm{x}; \mathbf{W}) = \mathbf{W}\mathrm{x}$$

Obrázek 16.7: Vyjádření třívrstvé neuronové sítě pomocí násobení matic, $\mathbf{x}$ je vstupní vektor, $\mathbf{W}$ je vektor (matice) vah.

### 16.4.2 Výpočetní graf




Obrázek 16.8: Vyjádření pomocí násobení matic lze transformovat velmi jednoduše do kódu, $x_i$ jsou mezivýsledky, $max$ je aktivační funkce, $W_j$ jsou váhy v jednotlivých vrstvách.




Obrázek 16.9: Vyjádření v kódu, lze zapsat jako orientovaný, acyklický graf, *dot* značí násobení.

# Kapitola 17

# SUI – Neuronové sítě pro strukturovaná data (konvoluční a rekurentní sítě, motivace, základní vlastnosti, použití).

## 17.1 Zdroje

- `09-neural_networks.pdf`
- `10-sequences_and_language.pdf`
- `SUI_2019-12-02.mp4`
- `SUI_2019-12-09.mp4`

## 17.2 Strukturovaná data

- Za strukturuovaná data považujeme obrázky, zvuk, text, ...
  - Mají nějakou strukturu, nejsou to neuspořádaná data.
- Standardní vícevrstvé neuronové sítě by fungovali i pro tyto data, avšak lze využít vlastností strukturovaných dat, pro efektivnější práci neuronových sítí.

  O jaké vlastnosti jde:
  - **Lokalita** – V případě obrázku, pixely, které se vyskytují blízko sebe, pravděpodobně patří stejnému objektu. Naopak pixely daleko od sebe spíše nepatří.
  - **Invariance zpracování vzhledem k pozici** – V případě obrázku, vychází myšlenky, že objekty v obraze se mohou pohybovat. Pokud se určitý objekt detekuje na jednom místě, je možné ho stejným způsobem detekovat i na jiném místě.
- Další specifické vlastnosti mají sekvence (video, zvuk) – časovou souvislost.

## 17.3 Konvoluční neuronové sítě

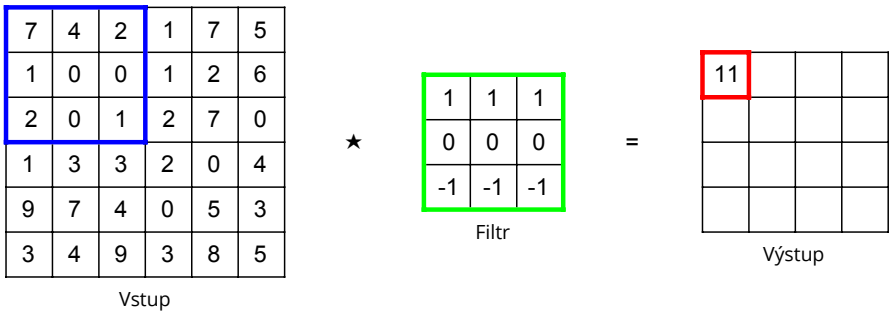
- Konvoluční neuronové sítě (CNN, *convolutional neural networks*) jsou hluboké neuronové sítě, ve kterých jednotlivé skryté vrstvy mají formu zaměření pro konkrétní činnost (konvoluce, pooling,dropout, ...).

### 17.3.1 Konvoluční vrstvy

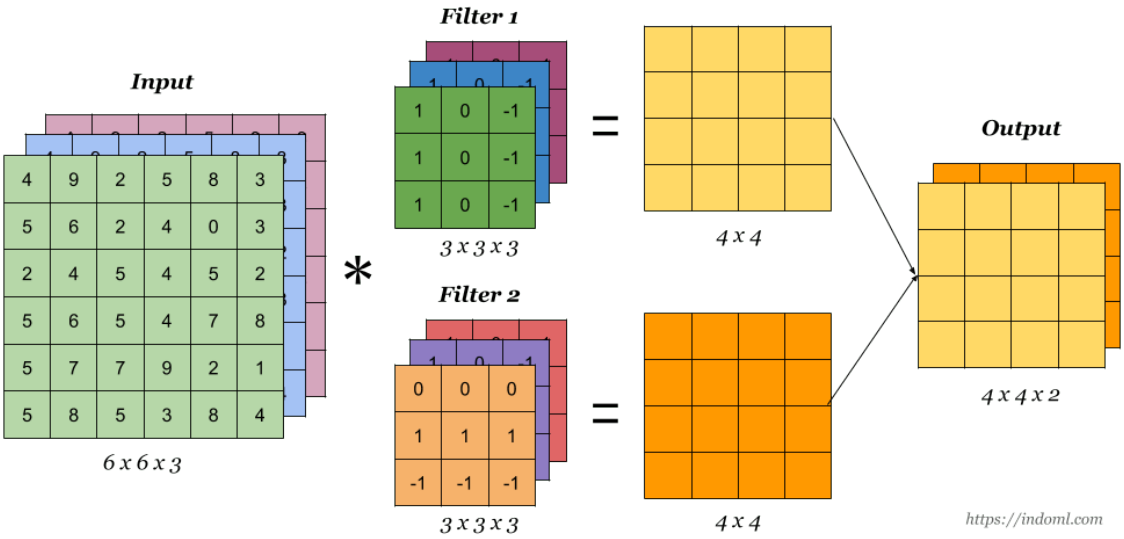
- Konvoluční vrstvy (*convolutional layers*) jsou hlavní stavební kameny konvolučních neuronových sítí. Jejich název je převzat od jména matematické operace, která se ve vrstvě provádí – konvoluce ($\star$).
  - Jde o operaci, která dokáže dobře využít vlastnosti strukturovaných dat (lokalita, invariance zpracování vzhledem k pozici).
  - Obraz je reprezentován diskrétními hodnotami pixelů, konkrétně maticí těchto hodnot, proto nás zajímá diskrétní 2D konvoluce.

$$f(x,y) \star h(x,y) = \sum_{i=-k}^{k} \sum_{j=-k}^{k} f(x-i, y-j) \cdot h(i,j) \tag{17.1}$$

  - Symbol $\star$ je operátor konvoluce nad funkcemi $f(x,y)$ a $h(x,y)$. První operand, v našem případě funkce $f$, obvykle značí vstupní obraz a druhý operand, funkce $h$, značí konvoluční jádro (filtr).
- Intuice, která stojí za aplikací konvoluce, je hledání podobnosti tzv. konvolučního jádra (konvoluční filtr) a daného segmentu obrazu. Výsledkem je, jak velký má být signál daného segmentu v rámci celého výsledku.
- Obrázek je typicky reprezentován maticí hodnot pixelů pro každý barevný kanál (pokud není v odstínu šedi). Konvoluce se aplikuje na každý barevný kanál.
- Parametry konvoluční vrstvy se skládají z množiny **konvolučních filtrů**. Samotná síť se učí optimalizací jejich jednotlivých hodnot. Filtry jsou po vstupu posouvány po určitém kroku a je počítána výsledná dvoudimenzionální mapa. Konvoluční filtry mohou mít různá uplatnění. Například pro rozmazaní či doostření obrazu, ale také pro detekci hran. Hrana je rozpoznána prudkou změnou světelných podmínek. Podstata filtru je nalézt gradient, tj. směr maximální změny, který hranu charakterizuje.
- Výstupní matice je menší než vstupní. To je důsledkem toho, že pro výpočet jednoho výsledného bodu potřebujeme na vstupu matici o velikosti filtru. Pokud chceme zmenšení rozlišení předejít, dá se původní matice rozšířit. A to buď nulovými body (*zero padding*), případně jinými, vhodně vybranými hodnotami. Avšak většinou je zmenšení žádané, výstup je jiný obrázek, resp. jeho jiná (jednodušší) reprezentace.
- Vlivem výpočtu v konvolučních vrstvách může dojít k tomu, že některé hodnoty v matici nabudou záporných hodnot. Barvy jsou ale reprezentovány třemi kanály v rozmezí $0-255$. Proto je často používána přenosová funkce ReLU, která odstraní negativní hodnoty z aktivační mapy.



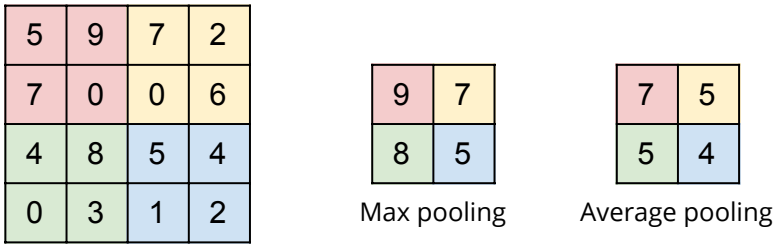

Obrázek 17.1: Příklad aplikace konvoluce (*výsledek je chybný, má být 10*, $7 + 4 + 2 - 2 - 1 = 10$).




Obrázek 17.2: Příklad aplikace konvoluce, vstup má 3 barevné kanaly, vrstva obsahuje 3 konvoluční filtry.

### 17.3.2 Pooling vrstvy

- Sdružování (*pooling*), slouží pro podvzorkování – zmenšení rozměrů dvoudimenzionální mapy.
- Díky tomu, se zmenší počet parametrů sítě v dalších vrstvách a dojde ke snížení výpočetních nároků.
- Realizuje se agregací několika sousedních buněk do jedné. Nejběžnější způsoby jsou *average pooling*, kdy se počítá průměrná hodnota z agregovaných buněk a *max pooling*, kdy se vybírá maximální hodnota.
- Pooling vrstvy nemají parametry (neučí se), jsou natvrdo naprogramované.
- Za sdružováním stojí myšlenka, že úplně přesná lokace příznaku není pro úlohu důležitá. Naopak může být na škodu a podpořit problém přetrénování sítě. Důležitější je tedy relativní lokace příznaku vzhledem k ostatním. Tím se dosáhne větší obecnosti modelu (generalizace).
- Pooling vrstva se běžně vkládá mezi jednotlivé konvoluční vrstvy. V současnosti se nejvíce používá přístup *max pooling* a *average pooling*.




Obrázek 17.3: Příklad aplikace max a average poolingu.

### 17.3.3 Plně propojená vrstva

- Plně propojená vrstva (*fully connected layer* nebo *dense layer*) spojuje každý neuron z dané vrstvy s každým neuronem vrstvy následující.
- Jedná se o stejný princip jako u klasických vícevrstvých neuronových sítí.
- Ačkoliv jsou sítě s plně propojenými vrstvami schopny dobře aproximovat libovolnou funkci, jejich vysoký počet spojení výrazně zvyšuje nároky na výpočetní kapacity. Proto se vyskytují spíše až v koncových vrstvách konvolučních sítí, kde mají na starosti klasifikaci objektů.

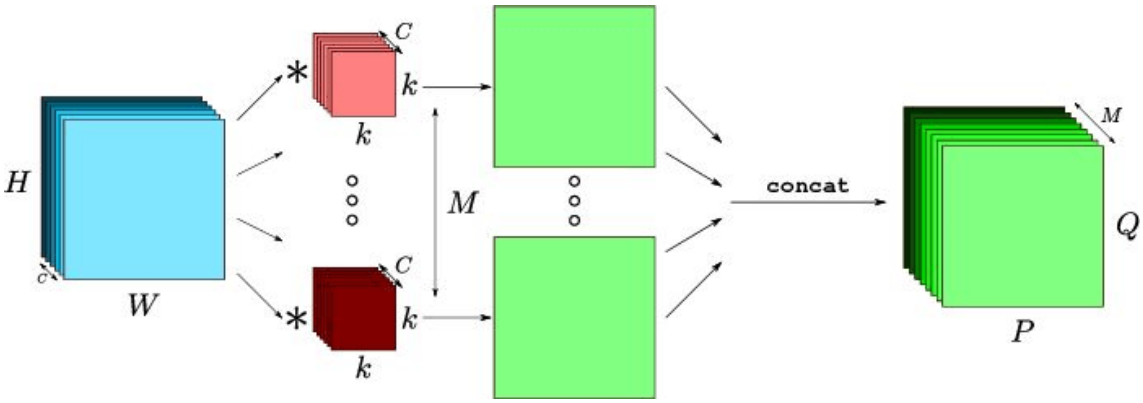
### 17.3.4 Dropout vrstva

- Dropout vrstva má za úkol snížit náchylnost sítě k přetrénování.
- Spočívá v nastavení výstupu každého neuronu ve skryté vrstvě na 0 s pravděpodobností 50 % (tzv. deaktivace neuronu).
- K deaktivaci neuronů dochází u každého vzorku dat při trénování, do učení je tak uměle přidáván šum, díky kterému je síť schopna více generalizovat.
- Dropout vrstva zvýšuje počet potřebných iterací k dosažení konvergence.

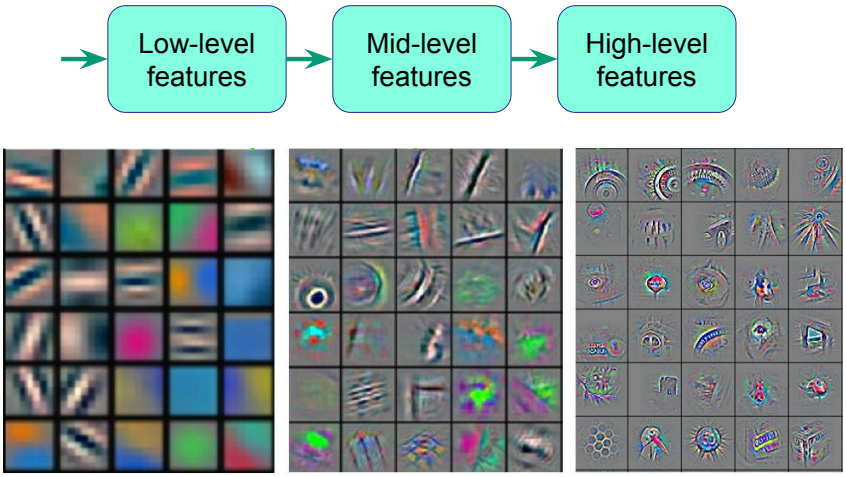
### 17.3.5 Loss vrstva

- Za poslední typ vrstvy lze považovat *loss* vrstvu, která specifikuje jak proces trénování penalizuje rozdíl mezi predikcí sítě (výstupem) a skutečnými hodnotami (označená trénovací data) – chybové funkce (objektivní funkce).
- Jedná se o finální vrstvu neuronové sítě.
- Existuje mnoho chybových funkcí určené pro různé typy úloh.

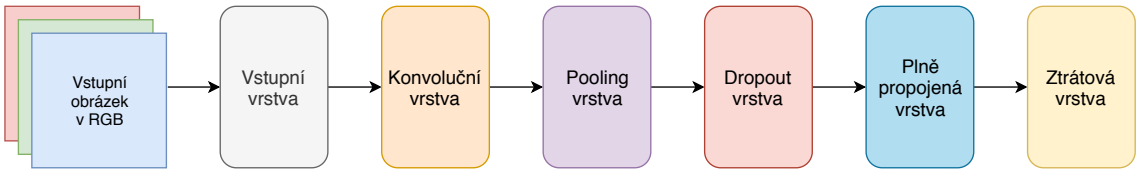
### 17.3.6 Příklady architektur



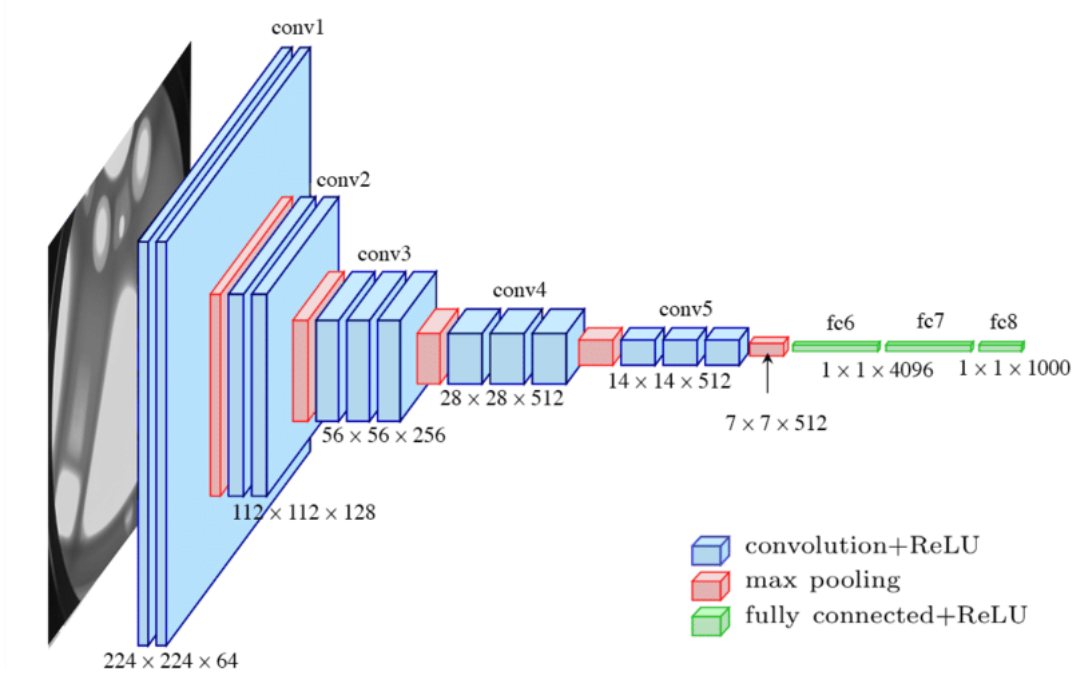

Obrázek 17.4: Příklad konvoluční vrstvy. Vstupem je obrázek o velikosti $H \times W$, který má $c$ kanálů. Konvoluční vrstva má $M$ konvolučních jader. Každé konvoluční jádro má velikost $k \times k$ a je zopakováno pro všechny kanály ($c$-krát). Výstup každého konvolučního jádra je agregován (sečten) do jednoho obrázku. Výstup konvoluční vrstvy je obrázek o velikosti $P \times Q$, který má $M$ kanálů (počet kanálů výstupního obrázku odpovídá počtu konvolučních jader ve vrstvě).




Obrázek 17.5: Příklad konvolučních jader.



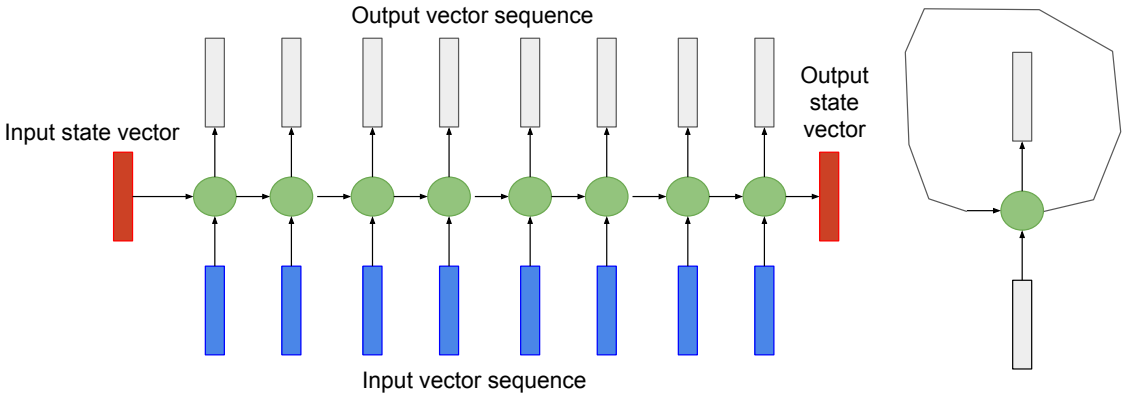

Obrázek 17.6: Příklad uspořádání jednotlivých typů vrstev za sebe. Typicky konvoluční a pooling (potažmo dropout) vrstvy se několikrát opakují.




Obrázek 17.7: Příklad konvoluční neuronové sítě VGG pro klasifikaci obrazu (vstupní a ztrátová vrstva nejsou ve schématu zobrazeny). Čím hlubší vrstva, tím menší matice, ale o to více konvolučních filtrů.

## 17.4 Rekurentní neuronové sítě

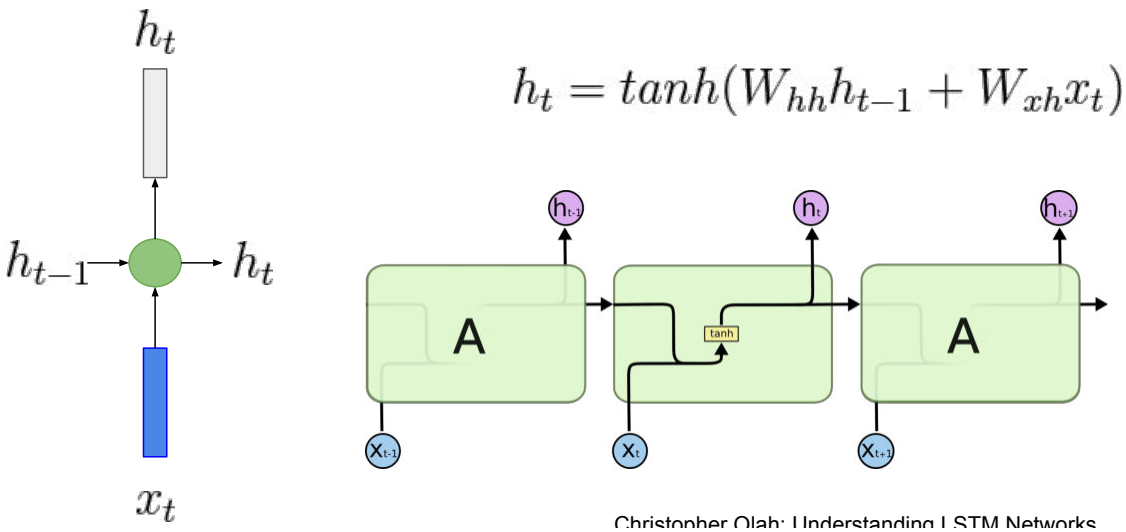
- Základní vícevrstvé NN i konvoluční NN patří mezi tzv. dopředné neuronové sítě.
- Rekurentní NN jsou další kategorií, která narozdíl od dopředných NN nepropaguje signál výhradně do dalších vrstev.
  - Resp. rekurentní vrstvy nejsou acyklické, ale obsahují jakýsi *self loop*.
- U výpočtu odezvy perceptronu je využito kromě aktuálního vstupu i vnitřní stav perceptronu.
  - Resp. jsou využity informace z předchozích vstupů, které jsou zakódovány ve vnitřním stavu.
- Rekurentní NN dokáží zachytit vlastnosti časové souvislosti v rámci sekvence dat.
- Rekurentní NN mají problém se naučit dlouhodobé závislosti.




Obrázek 17.8: Vizualizace rekurentní vrstvy. Výstup vrstvy jde na vstup stejné vrstvě v další iteraci.

### 17.4.1 Vanilla RNN

- První model rekurentní NN, základní model.




Obrázek 17.9: Schéma Vanilla RNN.

# Kapitola 18

# SUI – Prohledávání stavového prostoru (informované a neinformované metody, lokální prohledávání, prohledávání v nejistém prostředí, hraní her, CSP úlohy).

## 18.1 Zdroje

- `opora_izu-esf-5a.pdf`
- `02-prohledavani.pdf`
- `03-lokalni_prohledavani.pdf`
- `04-prohledavani_nejiste_prostredi.pdf`
- `05-adversarial_search.pdf`
- `06-csp.pdf`
- `Wikipedia`

## 18.2 Úvod a kontext

- **Stavový prostor** je dvojice $(S, O)$, kde
  - $S = \{s_1, s_2, \ldots, s_n\}$ je množina stavů (množina všech možných stavů úlohy).
  - $O = \{o_1, o_2, \ldots, o_m\}$ je množina operátorů (množina všech operátorů, kterými lze stavy úlohy měnit).
- **Úloha** je dvojice $(s_0, G)$, kde
  - $s_0 \in S$ je počáteční stav,
  - $G = \{s_{g1}, s_{g2}, \ldots\}$ je množina cílových stavů.
- **Řešení úlohy** je posloupnost operátorů:

$$s_1 = o_1(s_0), \ldots s_n = o_n(s_{n-1}) \ ; \ s_n \in G$$

  - Řešení úlohy je nalezení sekvence akcí, které dosáhnou cíle.

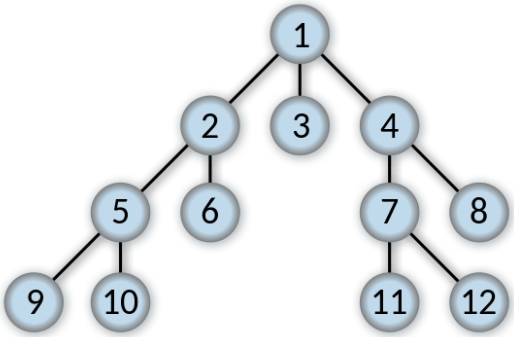
- **Terminologie**
  - Expanze uzlu – určení všech bezprostředních následovníků uzlu.
  - Generace uzlu – proces vytvoření uzlu.
- **Definice problému**
  1. Počáteční stav.
  2. Množina akcí $a$, která lze provést v každém stavu $s$ ($a = getActions(s)$).
  3. Přechodový model, nový stav (deterministický) po provedení akce $a$ ze stavu $s$ – $result(s, a)$.
  4. Test, je-li stav cílový – $isGoal(s)$.
  5. Cena cesty (počet kroků v sekvenci).
- **Kritéria hodnocení algoritmů**
  - Úplnost – Pokud nějaká řešení úlohy existují, tak úplná metoda jedno z nich musí nalézt.
  - Optimálnost – Pokud nějaká řešení úlohy existují, tak optimální metoda musí nalézt nejlepší z těchto řešení. Optimální metoda je vždy úplná.
  - Časová složitost – Jak dlouho trvá nalézt řešení.
  - Paměťová složitost – Kolik paměti je třeba pro nalezení řešení.
- **Známé úlohy**
  - Úloha dvou džbánů
  - Úloha Loydovy osmičky
  - Úloha osmi dam (CSP)
  - Úloha hanojských věží
  - Úloha balančních vah
  - Problém barvení map (CSP)

## 18.3 Neinformované metody pro prohledávání stavového prostoru

- Algoritmy nevyužívají žádné další informace kromě samotné definice úlohy.
- Aby algoritmus pořád neopakoval stejné kroky, musí si svou historii nějak ukládat (seznam CLOSED).

### 18.3.1 Prohledávání do šířky (BFS – Breadth First Search)

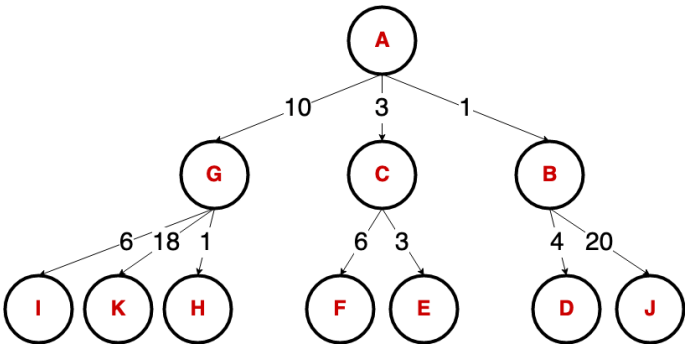
- Prohledává postupně od nejbližších (vzdálenost chápeme jako počet uzlů na cestě).
- Používá FIFO frontu OPEN a seznam CLOSED.
- Hodnocení
  - Úplná, optimální (pokud jsou všechny akce stejně dobré – fixní cena kroku).
  - Časová a prostorová složitost: $\mathcal{O}(b^d)$.




Obrázek 18.1: Ukázka BFS. Pořadí, ve kterém jsou uzly expandovány odpovídá názvům uzlů.

### 18.3.2 Uniform Cost Search (UCS, Best First Search)

- De facto Dijkstrův algoritmus.
- Co když je každá akce jinak dobrá?
  - Stačí nevybrat nejbližší uzel (hloubka stromu), ale uzel s nejmenší cenou cesty.
  - Slepé prohledávání do šířky s respektováním cen přechodů.
- Používá prioritní frontu OPEN a seznam CLOSED.
- Hodnocení
  - Úplná, optimální.
  - Prostorová i časová složitost metody je dána cenou optimálního řešení ($C^*$) dělenou nejmenším přírůstkem ceny mezi dvěma uzly ($\Delta C_{min}$): $\mathcal{O}(b^{C^*/\Delta C_{min}})$.



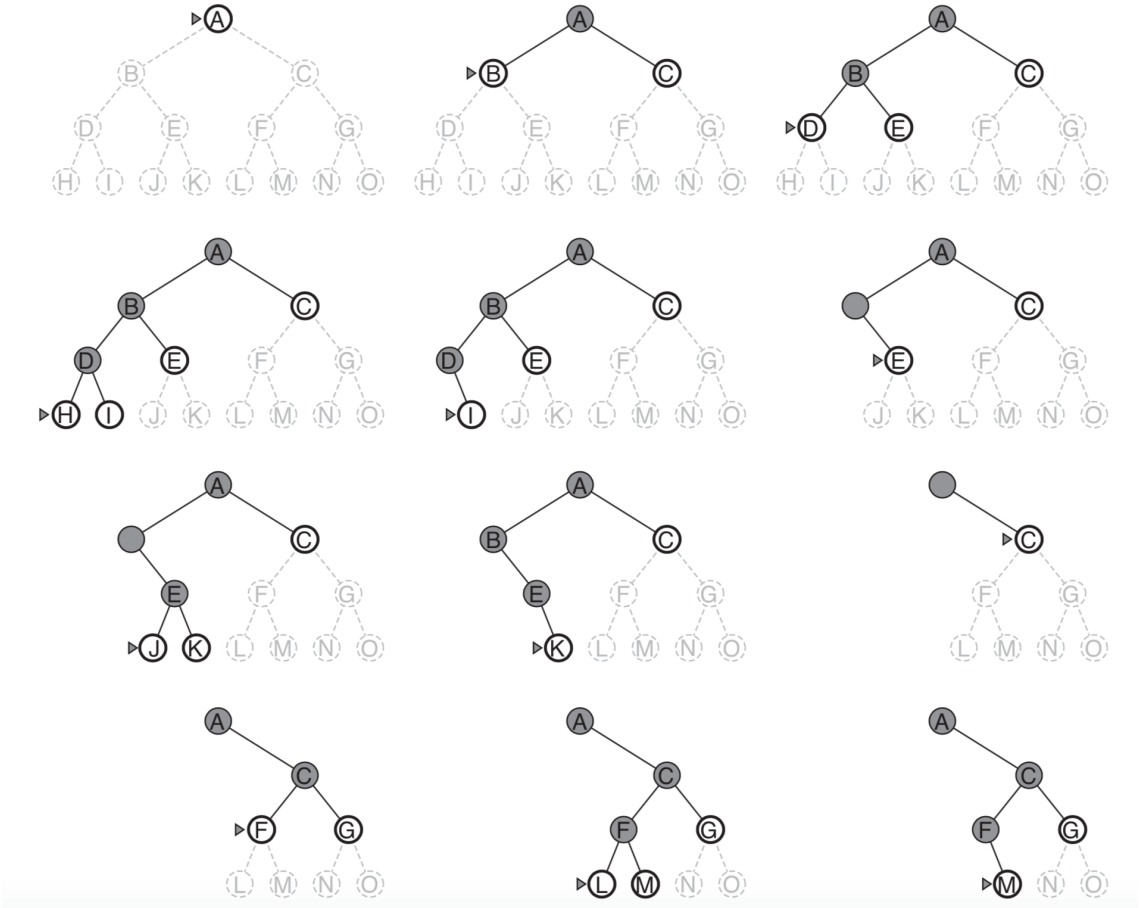

Obrázek 18.2: Ukázka UCS. Pořadí, ve kterém jsou uzly expandovány odpovídá názvům uzlů.

### 18.3.3 Prohledávání do hloubky (DFS – Depth First Search)

- DFS řeší problém s pamětí při prohledávání stromu (v grafu musíme stejně ukládat prozkoumané stavy, takže nám nepomůže).
- Používá LIFO zásobník OPEN a seznam CLOSED.
- Hodnocení
  - Není úplná, není optimální.

- Prostorová složitost: $\mathcal{O}(b \cdot m)$.
- Časová složitost: $\mathcal{O}(b^m)$.

- Vylepšení
  - Slepé prohledávání do omezené hloubky (Depth Limited Search - DLS).
  - Slepé prohledávání do omezené hloubky s postupným zanořováním (Iterative Deeping Search - IDS) – Postupně se volá procedura DLS a postupně se zvyšuje maximální hloubka.



Obrázek 18.3: Ukázka DFS.

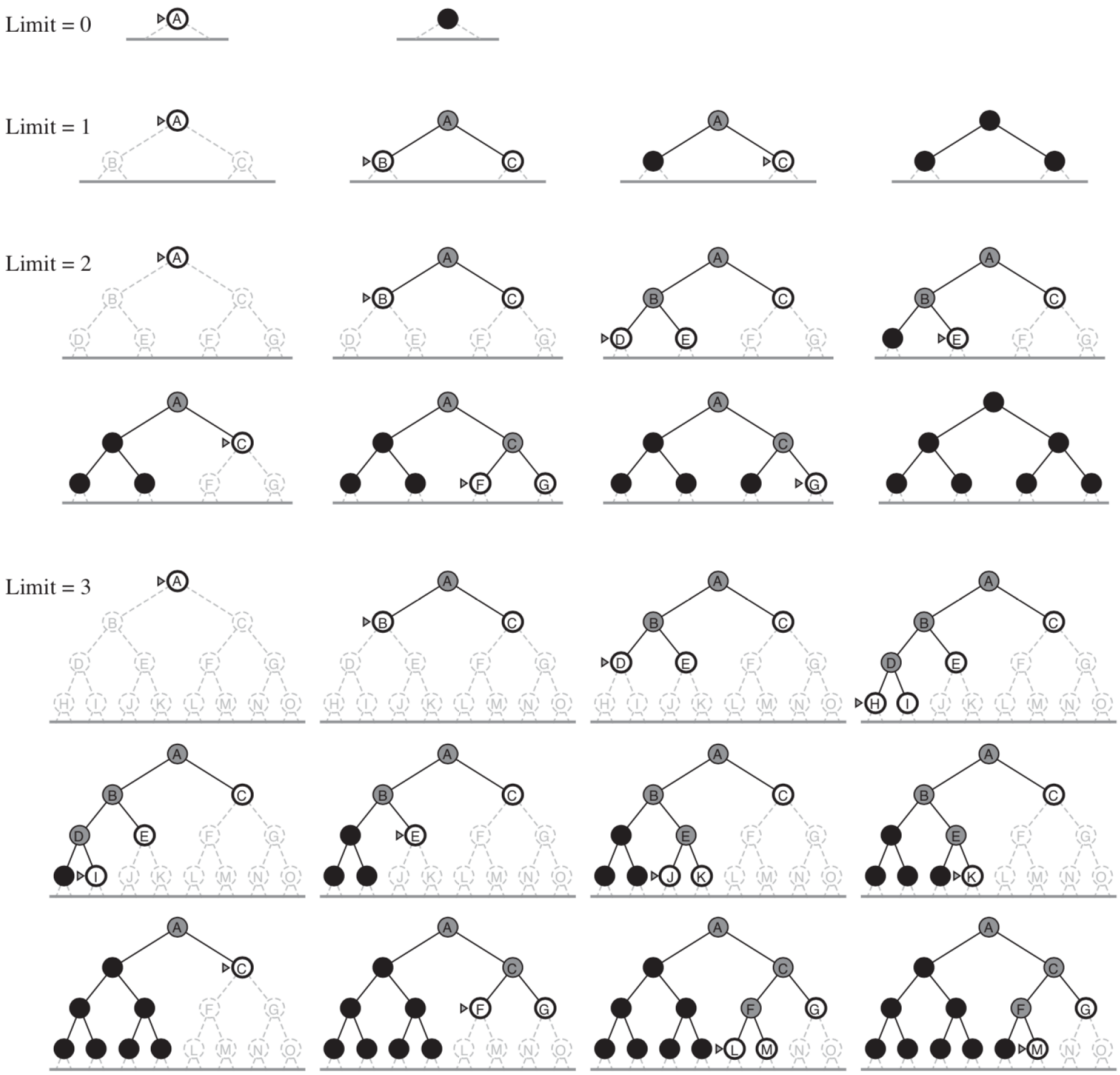
### 18.3.4 Prohledávání do omezené hloubky (DLS – Depth Limited Search)

- Vylepšení DFS, kde se omezí hloubka prohledávání.
  - Proč to dělat? Ochrana proti zacyklení.
- Úplná metoda, pokud není řešení ve větší hloubce, než je nastavený limit.

### 18.3.5 Prohledávání do omezené hloubky s postupným zanořováním (IDS – Iterative Deeping Search)

- Iterativní DLS, začíná se s limitem 0 a v každé iteraci se o 1 zvyšuje.

- Doporučená metoda při využití neinformovaných metod (pokud víme, že máme dostatek paměti, BFS bude rychlejší).
- Optimální metoda s malými paměťovými nároky.
- Optimálnost s ohledem na počet operací (ne cenu řešení).



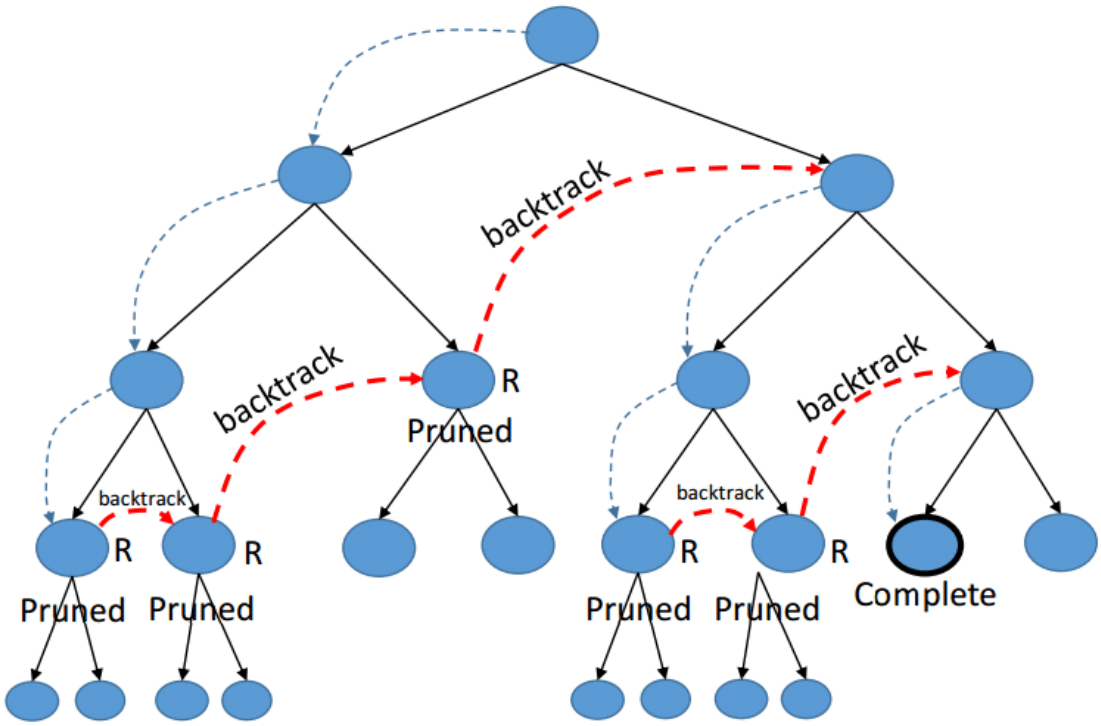

Obrázek 18.4: Ukázka IDS.

### 18.3.6 Backtracking

- Slepé prohledávání se zpětným návrácení
  - Operátory je nutné udržovat seřezené.
  - Velmi podobné DFS, ale místo expanze uzlu (generování všech následníků) generuje pouze jediného následníka (nejprve prvního a při případných návratech další).
  - Jde využít opět omezenou hloubku (jako u DLS).
- Využívá zásobník OPEN.
- Hodnocení

- Není úplný, ani optimální.
- Extrémně nízká prostorová složitost.

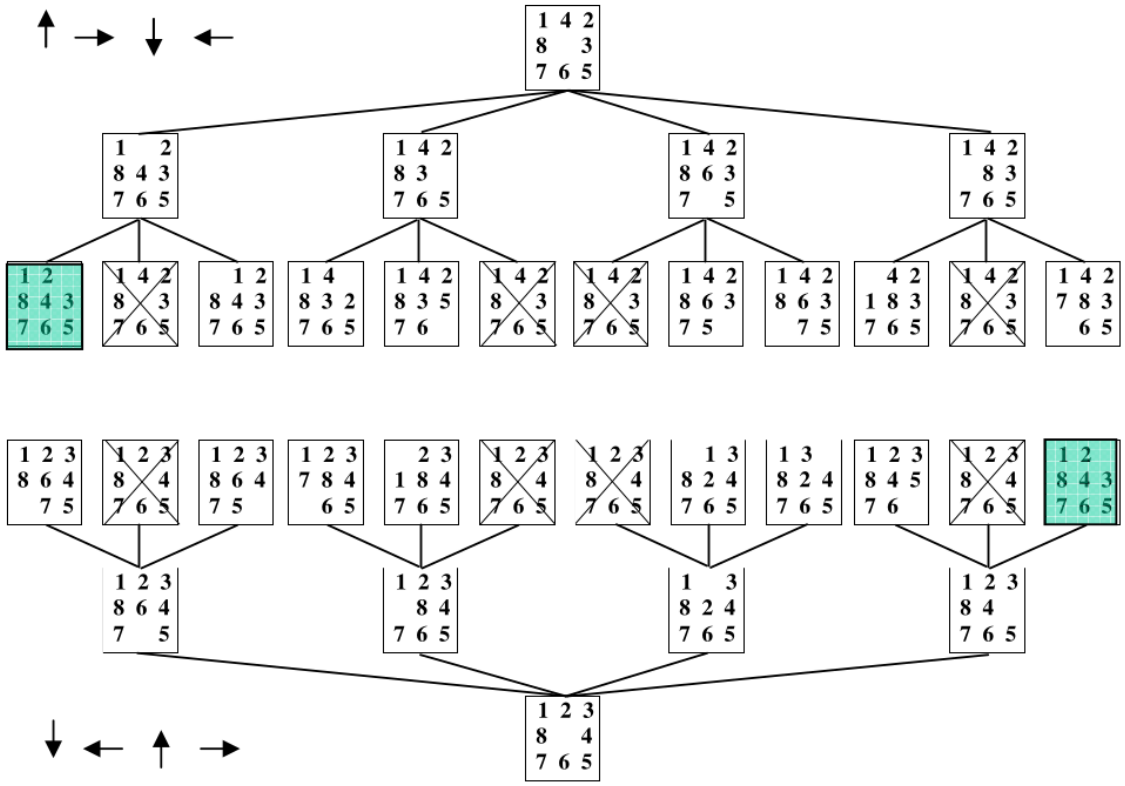
```
procedure EXPLORE(node n)
    if REJECT(n) then return
    if COMPLETE(n) then
        OUTPUT(n)
    for n_i : CHILDREN(n) do EXPLORE(n_i)
```




Obrázek 18.5: Ukázka Backtrackingu.

### 18.3.7 Birectional Search (BS, obousměrné BFS)

- BFS z počátečního a cílového stavu zároveň, hledá se spojení.
- Dá se použít pouze pro řešení úloh s reverzibilními operátory.
  - Např. lze použít pro řešení úlohy Loydovu osmičku, ale nelze použít pro řešení úlohy dvou džbánů.
- Hodnocení
  - Úplná, optimální.
  - Časová a prostorová složitost: $\mathcal{O}(2b^{d/2})$.



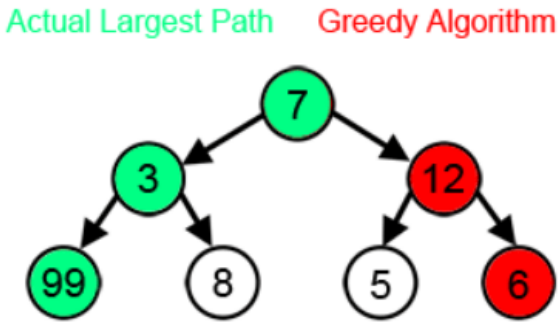
Obrázek 18.6: Ukázka BS, prohledávání stavového prostoru hlavolamu 8 metodou obousměrného prohledávání.

## 18.4 Informované metody pro prohledávání stavového prostoru

- Algoritmy využívají informace o tom, jak je daný stav „nadějný“.
- Implementace metod odpovídá UCS (jediná změna je způsob ohodnocení uzlů).
- UCS využíval pro ohodnocení uzlu $n$ funkci $g(n)$, která odpovídala nejkratší ceně cesty z počátečního uzlu do uzlu $n$.
- Informované metody se řídí podle funkce $f(n)$, která využívá funkci $h(n)$ a může mít tvar:
  - $f(n) = g(n) + h(n)$,
  - $h(n)$ je heuristická funkce a je to odhad ceny cesty z uzlu $n$ do nejbližšího cílového uzlu.

### 18.4.1 Hladový algoritmus (Greedy Search)

- „Hladové/chamtivé prohledávání, bereme nejbližší krok.“
- Využívá se pouze heuristická funkce, $g(n) = 0$, $f(n) = h(n)$.
  - Odhadovanou cenou z daného uzlu do uzlu cílového.
  - K expanzi vybírá uzel, který má toto hodnocení nejnižší.
  - Např. vzdálenost vzdušnou čarou.
- Dobrá heuristika může časovou náročnost výrazně redukovat.



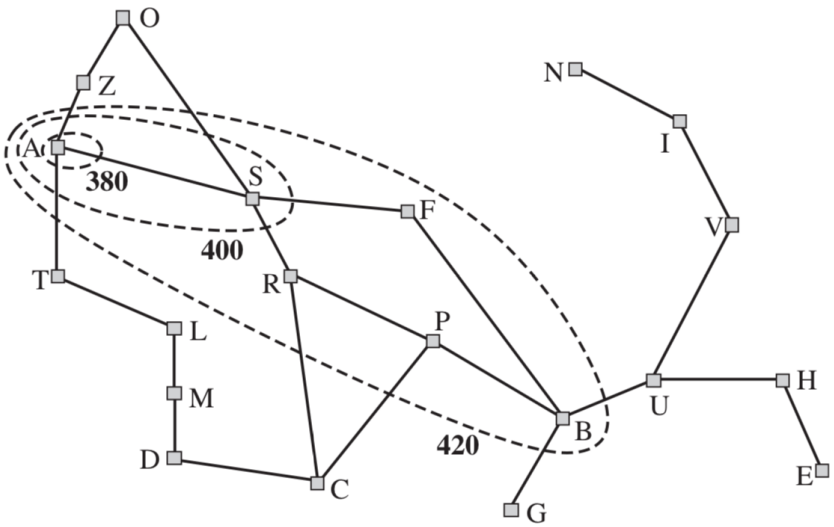

Obrázek 18.7: Ukázka hladového algoritmu, vybírá maximum.

### 18.4.2 A* Search

- Využívají se obě složky funkce $f(n)$:

$$f(n) = g(n) + h(n)$$

- Úplná i optimální
  - Platí pro tzv. přípustné heuristiky, kdy odhad $h(n)$ nebude nikdy větší než reálná cena do cíle.
- Heuristika je v tomto případě chápána jako způsob prořezávání (prunning) – vyloučení některých možností bez jejich prozkoumání.




Obrázek 18.8: Ukázka A* Search.

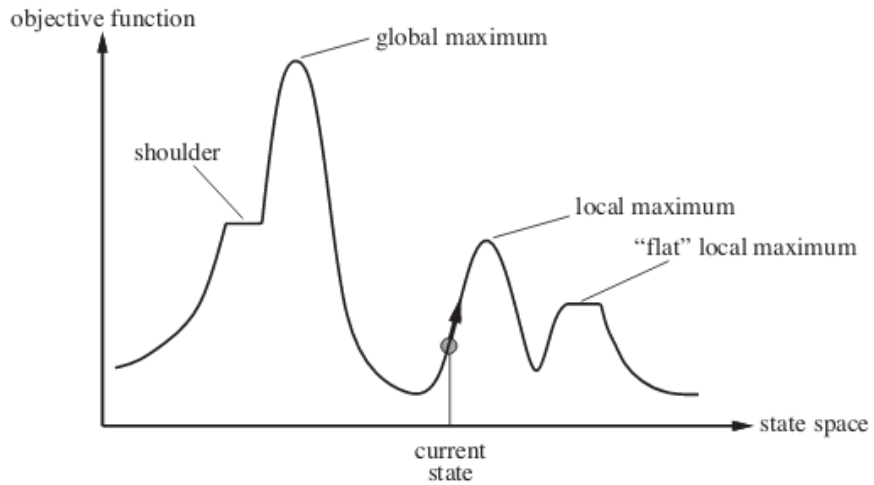
## 18.5 Lokální prohledávání

- Neprohledáváme systematicky celý prostor, jsme pouze v jednom (nebo i více) aktuálním uzlu (stavu), který vyhodnocujeme a modifikujeme.
- Metody využívají málo paměti, jsou použitelné pro řešení úloh ve velkých (i spojitých) prostorech, kde nemáme zdroje pro prozkoumání všech možností.

- Abychom získali nový uzel ze současného, využíváme sousednost uzlů (např. skrze akce, náhodná malá změna, kombinace více uzlů).

### 18.5.1 Hill climbing

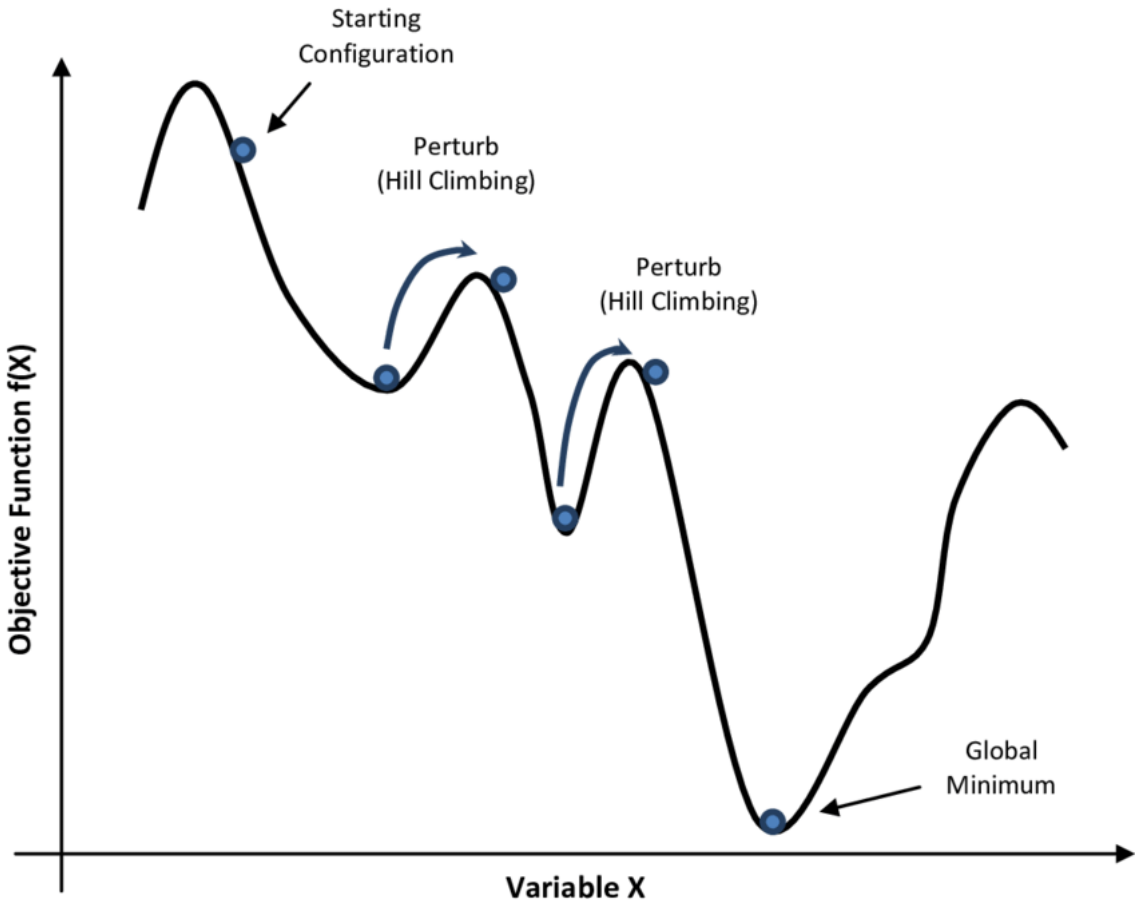
- Také označovaný jako *greedy local search*.
- Jako nový uzel se zvolí nejlepší ze sousedních uzlů (steepest-ascent varianta), který je zároveň lepší než současný uzel; pokud takový není, výsledným řešením (výstupem algoritmu) je současný uzel.
- Problematické jsou lokální extrémy a části, na kterých je funkce konstantní (plateau).
- Gradient descent u trénování neuronových sítí by sem spadal (resp. varianta tohoto algoritmu pro spojité prostředí).
- Vylepšení:
  - Stochastic hill climbing – pokud je více sousedních uzlů lepších než aktuální, nemusí se volit ten nejlepší.
  - Random-restart hill climbing – jeden běh algoritmu nalezne lokální extrém, pokud iterativně spouštíme algoritmus s (náhodnou) inicializací, zvyšujeme pravděpodobnost nalezení globálního extrému.




Obrázek 18.9: Ukázka Hill climbing.

### 18.5.2 Simulated annealing

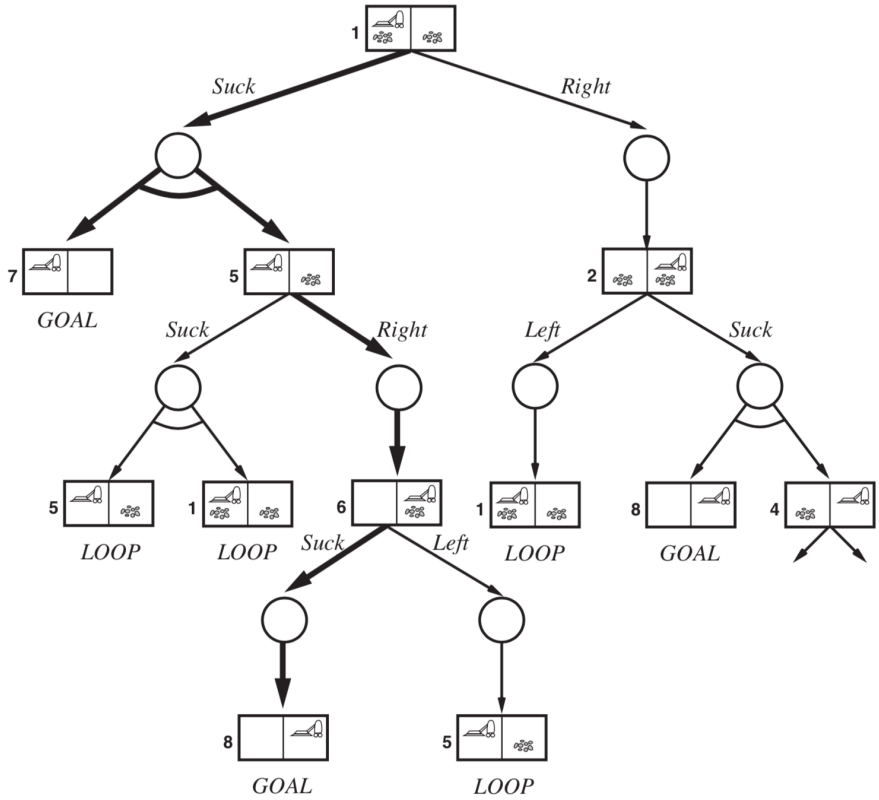
- Kombinace hill-climbing a náhodné procházky (volíme náhodného souseda).
- Náhodně vybíráme sousední uzel, pokud je lepší než současný, nahradíme, pokud je horší, spočítáme, o kolik je horší $\Delta E$ (záporné číslo).
- Myšlenka snižování teploty $T$ v každé iteraci (čím nižší je teplota, tím menší pravděpodobnost nahrazení horším uzlem).
- Pravděpodobnost nahrazení horším uzlem se spočítá podle vztahu $e^{\Delta E/T}$.




Obrázek 18.10: Ukázka Simulated annealing.

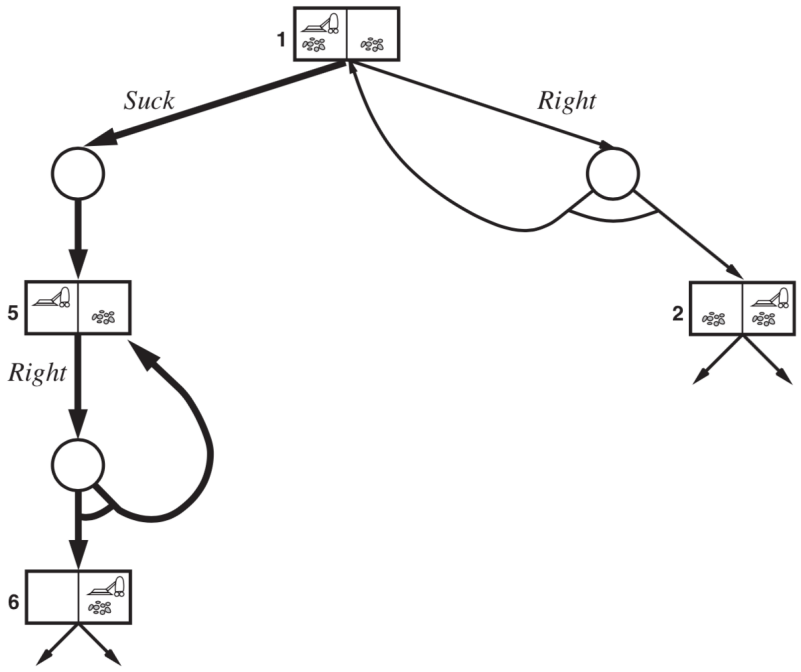
## 18.6 Prohledávání v nejistém prostředí

- Máme nedeterministické akce (nejisté prostředí, částečně pozorovatelné prostředí).
- Zadaný problém již nelze řešit pouze pevně danou sekvencí akcí, ale potřebujeme vytvořit nějaký plán / strategii.
- Výsledkem $RESULT(s, a)$ je nyní množina akcí a cena $COST(s, a, s')$ provedení akce $a$ ve stavu $s$ závisí také na novém stavu $s'$, ve kterém se ocitneme po provedení akce.
- Dříve jsme věděli, co se stane po provedení akce, plán = sekvence akcí (nezávisle na vjemech) Nyní se rozhodujeme i na základě vjemů.
- Řešení pomocí AND / OR prohledávacích stromů (operátor and se používá pro výsledek nedeterministické akce).
  - OR – můžu si vybrat, kterou akci provedu.
  - AND – výsledkem akce může být více stavů, řešení pro všechny možné nové stavy.




Obrázek 18.11: AND-OR prohledávací stromy.

- Bavíme se o belief state – reflektuje agentovu důvěru o aktuálním stavu (v jakých stavech se může nacházet). V nepozorovatelném prostředí se vyskytuje celkem $2^n$ belief states.
- Incremental belief-state search – postupně hledáme řešení pro každý stav; výhoda: pokud neexistuje řešení, zjistí se rychle; jiná řešení později. Udržování si Belief state je základem inteligentního agenta v ČPP (většina reálných prostředí).



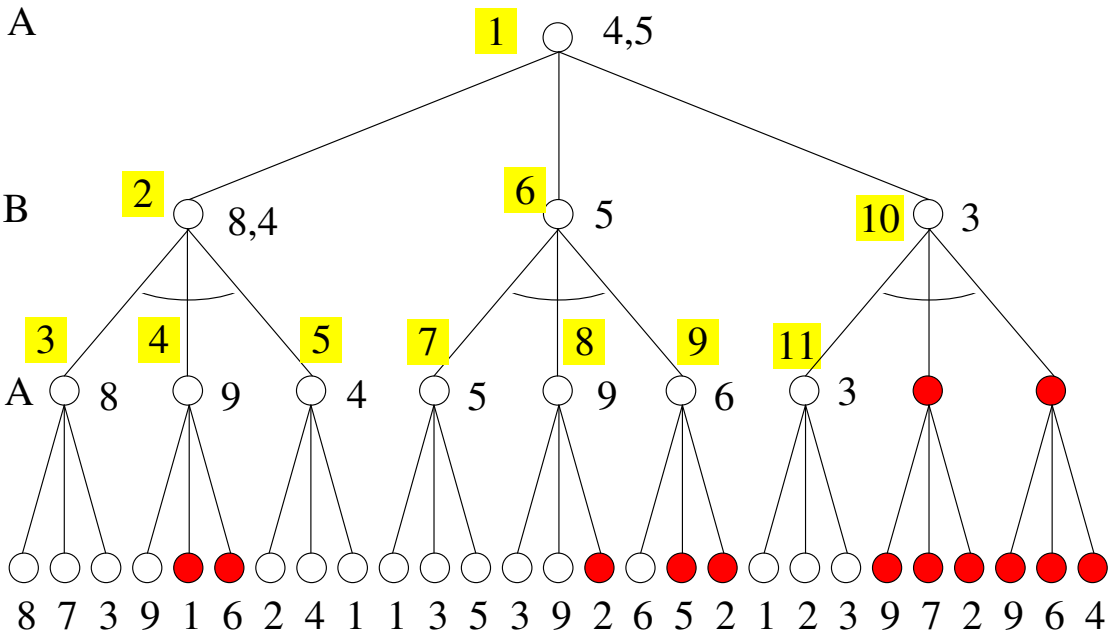

Obrázek 18.12: Nedeterministické akce: pohyb může selhat.

## 18.7 Hraní her (adversial search)

- Typ prohledávání v soutěživém prostředí, často ho označujeme jako hry teorie her (game theory).
- Nejčastěji se jedná o speciální druh her:
  - Hry s nulovým součtem (zero-sum game) – Součet zisku/užitku všech hráčů je konstantní – typ her, ve kterých není možné, aby všichni prosperovali (lepší výsledek je na úkor druhých).
  - Tahové
  - Dvou hráčů
  - Perfect information (plně pozorovatelné)
  - Deterministické
- Multiagentní prostředí, pokud se v prostředí vyskytují dva agenti, chápeme to tak, že nám druhý agent (hráč) chce škodit.
- Prozkoumat všechny možnosti je u většiny her (stejně jako rozhodování v reálném světě) nemožné a přesto nutné zvolit nějakou akci (tah).
- Hra má svůj strom (game tree), my ovšem prohledáváme pouze některé části tohoto stromu, což je náš prohledávací strom (search tree).
- Každý stav hry je ohodnocen hodnotící funkcí. Kladné hodnoty této funkce musí znamenat příznivý stav pro hráče na tahu (hráče A). Záporné pak příznivý stav pro protihráče (hráče B).

### 18.7.1 Algoritmus MiniMax

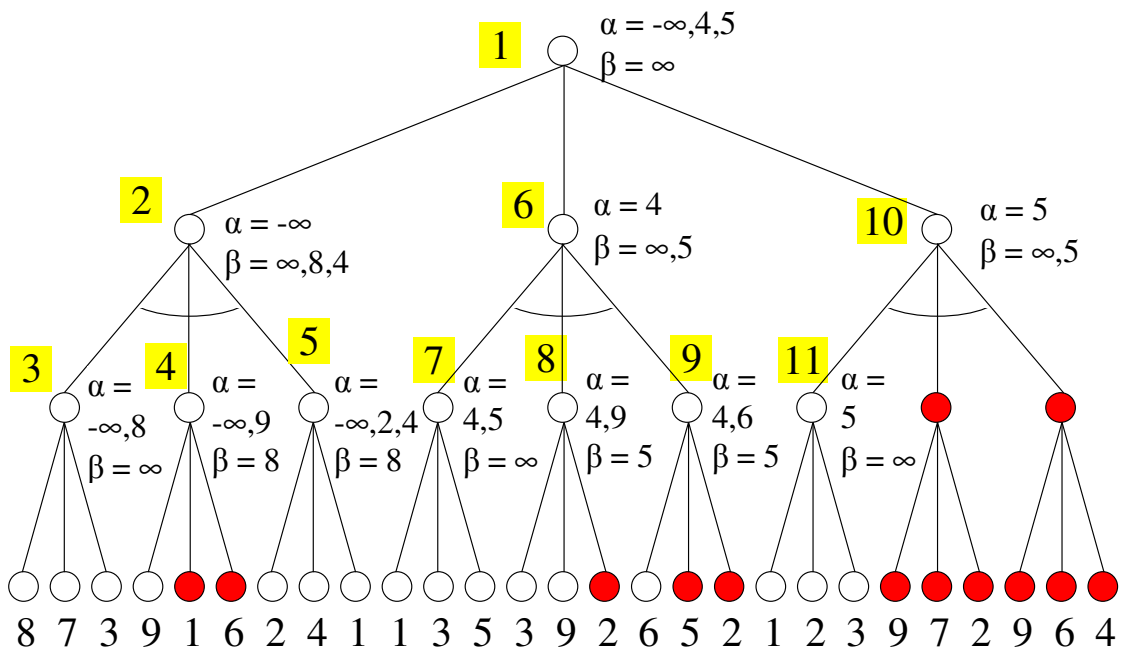
- Hry dvou hráčů.
- Racionální chování, protivník vždy zvolí svůj nejlepší tah (pro nás nejhorší).




Obrázek 18.13: Příklad na zbytečná vyšetřování některých uzlů AND/OR grafu metodou MiniMax.

### 18.7.2 Alpha-beta pruning

- Hry dvou hráčů.
- Stejné jako MiniMax ale používá pruning.
- Alpha – dosavadní nejlepší tah pro MAX
- Beta – dosavadní nejlepší tah pro MIN
- Neprohledávat zbytečně, pokud by MIN nebo MAX tento tah stejně neprovedli.




Obrázek 18.14: Příklad práce procedury AlfaBeta (alfa a beta řezů).

## 18.8 Úlohy s omezujícíma podmínkama

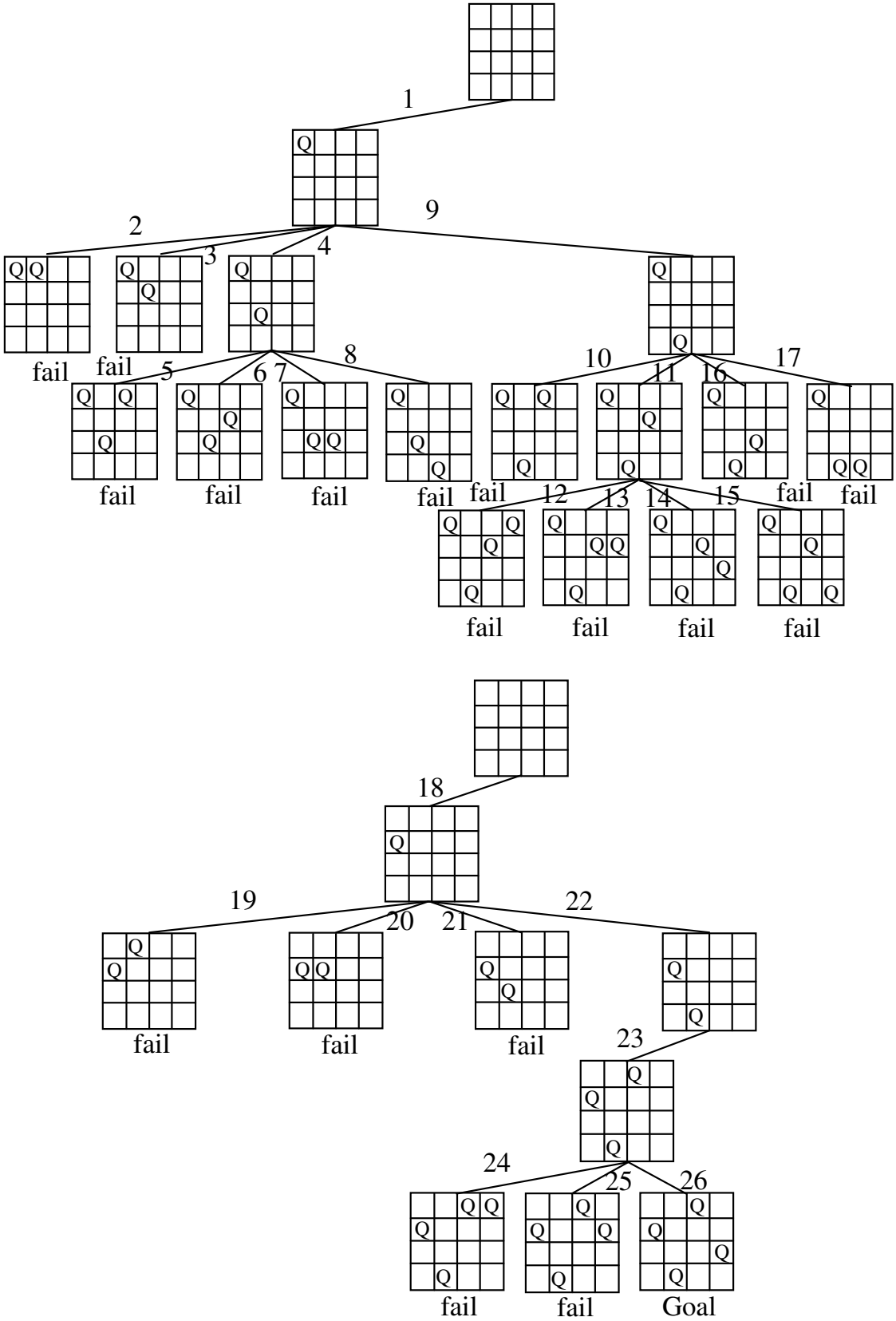
- Úlohy s omezujícíma podmínkama (CSP, *constraint satisfaction problem*).
- Záleží pouze na nalezení cílového stavu při splnění předem daných omezujících podmínek (není důležitý postup!).
- Stavy v CSPs jsou obvykle definovány množinou proměnných, kterým se přiřazují hodnoty z množin přípustných hodnot pro tyto proměnné.
- Opět nehledáme sekvenci akcí/tahů, nyní se ani nesnažíme najít nejlepší řešení. Hledáme pouze nějaké řešení, které splňuje předem stanovené podmínky pro kombinaci hodnot proměnných.
- Formální definice CSP úlohy:
  - $X$ je množina proměnných,
  - $D$ je množina domén,
  - $D_i$ je domána pro $X_i$,
  - $C$ je množina omezení, která definují povolené kombinace hodnot.
- **Naivní prohledávání** v každé úrovni stromu přiřadí hodnotu jedné proměnné, po přiřazení všech se zkoumá, jestli je řešení konzistentní.
- **Inference je alternativa** k prohledávání. Uvažujeme inferenci nazývanou constraint propagation CP. Snahou je snížit počet hodnot v doménách, což zrychlí prohledávání. Existuje několik algoritmů využívajících tento přístup.

### 18.8.1 Backtracking pro CSP

- Metodu zpětného navracení (backtracking) lze k řešení CSP použít velmi snadno – pokud aplikace operátoru vede na stav porušující omezující podmínky, pak je tento

operátor považován za neaplikovatelný a vrací se o krok zpět.

- Metoda je úplná (a každá úplná metoda je pro CSP optimální). Označíme-li symbolem $n$ počet proměnných a symbolem $m$ maximální počet přiřaditelných hodnot, pak platí:
  - Pro prostorovou složitost: $\mathcal{O}(n)$
  - Pro časovou složitost: $\mathcal{O}(m^n)$




Obrázek 18.15: Řešení úlohy čtyř dam metodou Backtracking for CSP (fail značí neúspěch, Goal znamená cíl).

### 18.8.2 Forward-checking

- Princip algoritmu Forward checking spočívá v tom, že po každém přiřazení hodnoty nějaké proměnné vyřazuje z množin přípustných hodnot dosud volných proměnných ty hodnoty, které jsou s právě přiřazenou hodnotou v konfliktu.

| | |
|---|---|
| | $S_1 = \{1,2,3,4\}$, $S_2 = \{1,2,3,4\}$, $S_3 = \{1,2,3,4\}$, $S_4 = \{1,2,3,4\}$ |
| Q | $i_1 = 1$,<br>$S_1 = \{2,3,4\}$, $S_2 = \{3,4\}$, $S_3 = \{2,4\}$, $S_4 = \{2,3\}$ |
| Q<br>Q | $i_1 = 1$, $i_2 = 3$,<br>$S_1 = \{2,3,4\}$, $S_2 = \{4\}$, $S_3 = \{\ \}$, $S_4 = \{2\}$ **fail !**<br>$S_3 = \{2,4\}$, $S_4 = \{2,3\}$ |
| Q<br>Q | $i_1 = 1$, $i_2 = 4$,<br>$S_1 = \{2,3,4\}$, $S_2 = \{\}$, $S_3 = \{2\}$, $S_4 = \{3\}$ |
| Q<br>Q | $i_1 = 1$, $i_2 = 4$, $i_3 = 2$,<br>$S_1 = \{2,3,4\}$, $S_2 = \{\}$, $S_3 = \{\}$, $S_4 = \{\}$ **fail !**<br>$S_2 = \{1,2,3,4\}$, $S_3 = \{1,2,3,4\}$, $S_4 = \{1,2,3,4\}$ |
| Q | $i_1 = 2$,<br>$S_1 = \{3,4\}$, $S_2 = \{4\}$, $S_3 = \{1,3\}$, $S_4 = \{1,3,4\}$ |
| Q<br>Q | i1 = 2, i2 = 4,<br>S1 = {3,4}, S2 = {}, S3 = {1}, S4 = {1,3} |
| Q<br>Q | i1 = 2, i2 = 4, i3 = 1,<br>S1 = {3,4}, S2 = {}, S3 = {}, S4 = {3} |
| Q<br>Q<br>Q | i1 = 2, i2 = 4, i3 = 1, i4 = 3,<br>S1 = {3,4}, S2 = {}, S3 = {}, S4 = {} **Goal !** |

Obrázek 18.16: Řešení úlohy čtyř dam metodou Forward checking (fail značí neúspěch, Goal znamená cíl).

# Kapitola 19

# TIN – Klasifikace formálních jazyků (Chomského hierarchie), vlastnosti formálních jazyků a jejich rozhodnutelnost.

## 19.1 Zdroje

- `tin_2021_merged.pdf`
- `TIN_2020-09-22.mp4`
- `TIN_2020-09-25_demo.mp4`
- `TIN_2020-09-29.mp4`
- `TIN_2020-10-06.mp4`
- `TIN_2020-10-13.mp4`
- `https://petrzemek.net/publications`

## 19.2 Úvod a kontext

- Symbol, znak
- Abeceda – Konečná množina symbolů.
- Řetězec
- Konkatenace řetězců
- Reverze řetězce
- Podřetězec
- Délka řetězce
- Formální jazyk
- Doplněk jazyka
- Konkatenace jazyků
- Mocnina jazyka
- Iterace jazyka

## 19.3 Gramatiky

**Gramatika** Gramatika slouží k formální specifikaci jazyků, zejména pak nekonečných jazyků. Gramatika je čtveřice $G = (N, \Sigma, P, S)$, kde

- $N$ je konečná množina neterminálů (pomocné syntaktické celky);
- $\Sigma$ je konečná množina terminálů (symbolů),
  - $N \cap \Sigma = \emptyset$;
- $P$ je konečná množina pravidel,
  - $P \subseteq (N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$;
  - prvek $(\alpha, \beta) \in P$ zapisujeme jako $\alpha \to \beta$;
- $S$ je výchozí neterminál,
  - $S \in N$.

**Přímá derivace** Nechť $G = (N, \Sigma, P, S)$ je gramatika a nechť $\lambda, \mu \in (N \cup \Sigma)^*$. Mezi řetězci $\lambda$ a $\mu$ platí binární relace $\lambda \Rightarrow \mu$, nazývaná přímá derivace, pokud můžeme vyjádřit:

$$\lambda = \gamma\alpha\delta$$

$$\mu = \gamma\beta\delta$$

kde $\gamma, \delta \in (N \cup \Sigma)^*$ a $\alpha \to \beta \in P$. Říkáme, že řetězec $\mu$ lze přímo derivovat z řetězce $\lambda$ v gramatice $G$.

**Derivace** Nechť $G = (N, \Sigma, P, S)$ je gramatika a nechť $\lambda, \mu \in (N \cup \Sigma)^*$. Mezi řetězci $\lambda$ a $\mu$ platí binární relace $\lambda \Rightarrow^+ \mu$, nazývaná derivace, jestliže existuje posloupnost přímých derivací $v_{i-1} \Rightarrow v_i, \quad i = 1, \ldots, n, \quad n \geq 1$, taková, že platí:

$$\lambda = v_0 \Rightarrow v_1 \Rightarrow \cdots \Rightarrow v_{n-1} \Rightarrow v_n = \mu$$

. Tuto posloupnost nazýváme derivací délky $n$. Říkáme, že řetězec $\mu$ lze derivovat z řetězce $\lambda$ v gramatice $G$. Jestliže v gramatice $G$ platí pro řetězce $\lambda$ a $\mu$ relace $\lambda \Rightarrow^+ \mu$ a nebo identita $\lambda = \mu$, pak píšeme $\lambda \Rightarrow^* \mu$.

**Větná forma, věta, jazyk generovaný gramatikou** Nechť $G = (N, \Sigma, P, S)$ je gramatika. Řetězec $\alpha \in (N \cup \Sigma)^*$ nazýváme větnou formou, jestliže platí $S \Rightarrow^* \alpha$, tj. řetězec $\alpha$ je generovatelný z výchozího symbolu $S$. Větná forma, které obsahuje pouze terminální symboly, se nazývá věta. Jazyk $L(G)$ generovaný gramatikou $G$, je definován množinou všech vět

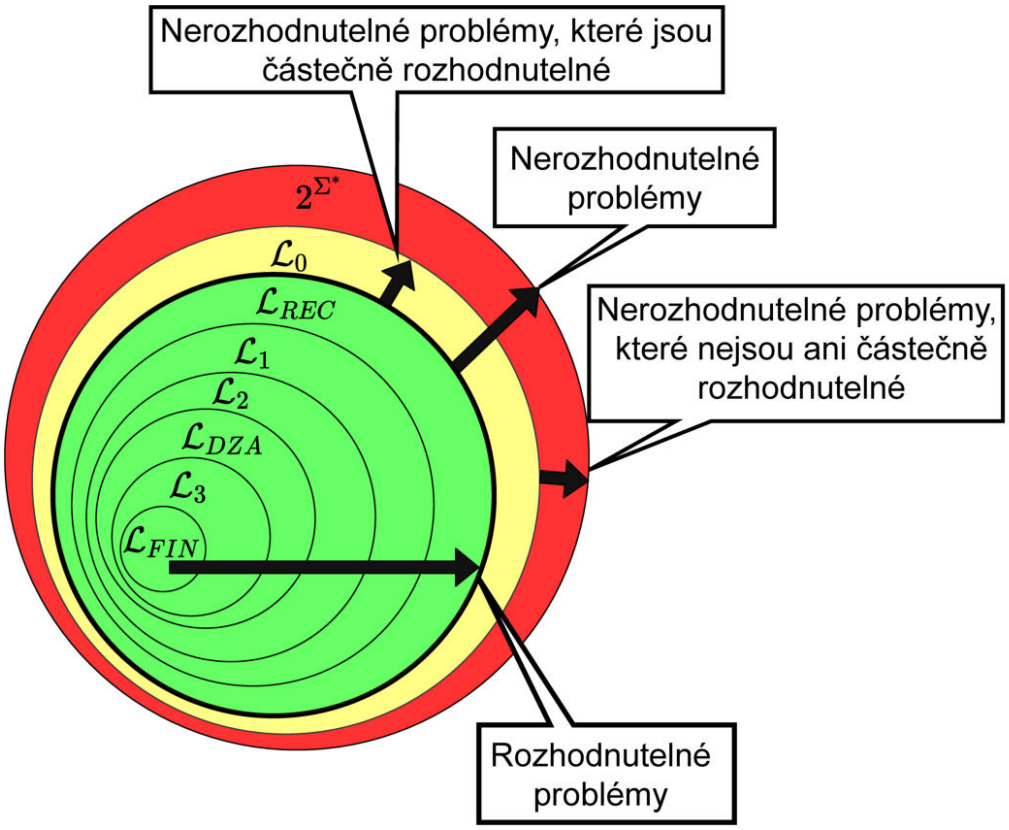
$$L(G) = \{w \mid w \in \Sigma^* \wedge S \Rightarrow^+ w\}$$

.

## 19.4 Chomského hierarchie

Chomského hierarchie je hierarchie tříd formálních gramatik generujících formální jazyky. Gramatiky jsou děleny dle tvaru jejich pravidel do 4 kategorií. Každé kategorii gramatik odpovídá kategorie formálních jazyků, které jsou gramatikami generovány – $\mathcal{L}_0, \mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3$.

Platí: $2^{\Sigma^*} \supset \mathcal{L}_0 \supset \mathcal{L}_1 \supset \mathcal{L}_2 \supset \mathcal{L}_3$. Existují i jazyky mimo třídu $\mathcal{L}_0$, které neni možné gramatikami vyjádřit. Chomského hierarchie vymezuje popisnou a rozhodovací sílu dané třídy.




Obrázek 19.1: Chomského hierarchie doplněna o další třídy a podtřídy jazyků. Jednotlivé třídy jsou vysvětleny dále. Teorii nerozhodnutelnosti je věnována jiná otázka.

**Třída jazyků $2^{\Sigma^*} \setminus \mathcal{L}_0$**

- Jazyky v této třídě není možné vyjádřit žádnou gramatikou.
- Příklad jazyků ($\langle M \rangle$ je kód TS $M$):
  - $\{\langle M \rangle \mid L(M) = \emptyset\} \in 2^{\Sigma^*} \setminus \mathcal{L}_0$
  - $\{\langle M \rangle \mid |L(M)| \leq 42\} \in 2^{\Sigma^*} \setminus \mathcal{L}_0$
  - $\{\langle M \rangle \mid L(M) \in \mathcal{L}_{Rec}\} \in 2^{\Sigma^*} \setminus \mathcal{L}_0$

**Typ 0 (obecné / neomezené / rekurzivně vyčíslitelné gramatiky)**

- Pravidla v nejobecnějším tvaru:
  - $\alpha \to \beta, \quad \alpha \in (N \cup \Sigma)^* N (N \cup \Sigma)^*, \quad \beta \in (N \cup \Sigma)^*$
- Jazyky $\mathcal{L}_0$ jsou příjímány turingovými stroji (TS) s nekonečnou páskou.
  - Varianty: deterministický TS, nedeterministický TS a vícepáskový TS disponují stejnou vyjadřovací silou (jsou mezi sebou převoditelné).

- Mezi třídami jazyků $\mathcal{L}_0$ a $\mathcal{L}_1$ existuje třída tzv. rekurzivních jazyků ($\mathcal{L}_{Rec}$, *Recursive Languages*).
  - Platí $\mathcal{L}_0 \supset \mathcal{L}_{Rec} \supset \mathcal{L}_1$.
  - Rekurzivně vyčíslitelné jazyky:
    * Nechť $M$ je turingův stroj, který přijímá nějaký jazyk z $\mathcal{L}_0$. Pokud $M$ dostane na vstup $w \in L(M)$, tak pro něj zastaví a přijme ho. Avšak, pokud $M$ dostane na vstup $w \notin L(M)$, tak buď zastaví a odmítne ho a nebo se zacyklí (nezastaví).
  - Rekurzivní jazyky:
    * Nechť $M$ je turingův stroj, který přijímá nějaký jazyk z $\mathcal{L}_{Rec}$. Pokud $M$ dostane na vstup $w \in L(M)$, tak pro něj zastaví a přijme ho. Avšak, pokud $M$ dostane na vstup $w \notin L(M)$, tak zastaví a odmítne ho. Turingův stroj, který zastaví pro jakýkoliv vstup se nazývá úplný turingův stroj.
- Příklad jazyků ($\langle M \rangle$ je kód TS $M$):
  - $\{\langle M \rangle \mid$ TS M má aspoň 42 stavů $\} \in \mathcal{L}_{Rec} \setminus \mathcal{L}_1$
  - $\{\langle M \rangle \mid$ TS M učiní více než 42 kroků na vstupu $\epsilon\} \in \mathcal{L}_{Rec} \setminus \mathcal{L}_1$
  - $\{\langle M \rangle \mid$ TS M učiní více než 42 kroků na nějakém vstupu $\} \in \mathcal{L}_{Rec} \setminus \mathcal{L}_1$
  - $\{\langle M \rangle \mid L(M) \neq \emptyset\} \in \mathcal{L}_0 \setminus \mathcal{L}_{Rec}$
  - $\{\langle M \rangle \mid |L(M)| \geq 42\} \in \mathcal{L}_0 \setminus \mathcal{L}_{Rec}$

**Typ 1 (kontextové gramatiky)**

- Pravidla ve tvaru:
  - $\alpha A \beta \to \alpha \gamma \beta, \quad A \in N, \quad \alpha, \beta \in (N \cup \Sigma)^*, \quad \gamma \in (N \cup \Sigma)^+$
  - A nebo $S \to \epsilon$, pokud se $S$ nevyskytuje na pravé straně žádného pravidla (neobsahují tzv. epsilon pravidla).
  - Z tvaru pravidel vyplývá, že větné formy se v průběhu derivace nemohou zkracovat, ledaže se derivuje $S \Rightarrow \epsilon$, ale v takovém případě se S nemůže vyskytovat na pravé straně žádného pravidla.
  - Formálně: pokud $\alpha \Rightarrow \beta$, pak $|\alpha| \leq |\beta|$ (s vyjímkou $S \Rightarrow \epsilon$).
- Jazyky $\mathcal{L}_1$ jsou příjímány nedeterministickým lineárně omezeným automatem (LOA).
  - Nedeterministický LOA je NTS s lineárně omezenou páskou v závislosti na délce vstupního řetězce.
  - Není známo, zda deterministický LOA disponuje stejnou vyjadřovací sílou jako nedeterministický LOA.
- Příklad jazyků:
  - $\{a^n b^n c^n \mid n \geq 10\} \in \mathcal{L}_1 \setminus \mathcal{L}_2$
  - $\{ww \mid w \in \Sigma^*\} \in \mathcal{L}_1 \setminus \mathcal{L}_2$

**Typ 2 (bezkontextové gramatiky)**

- Pravidla ve tvaru:

- $A \to \alpha, \quad A \in N, \quad \alpha \in (N \cup \Sigma)^*$
- Mohou obsahovat i tzv. epsilon pravidla (narozdíl od kontextových).
- Jak je to možné, když $\mathcal{L}_1 \supset \mathcal{L}_2$?
- Bezkontextové gramatiky s epsilon pravidly jsou převoditelné na kontextové gramatiky bez epsilon pravidel.

- Jazyky $\mathcal{L}_2$ jsou příjímány nedeterministickým zásobníkovým automatem (ZA).
  - Varianty: nedeterministický ZA a rozšířený nedeterministický ZA disponují stejnou vyjadřovací silou.
- Mezi třídami jazyků $\mathcal{L}_2$ a $\mathcal{L}_3$ existuje třída tzv. deterministických bezkontextových jazyků ($\mathcal{L}_{DCF}$, *Deterministic Context Free Languages*).
  - Platí $\mathcal{L}_2 \supset \mathcal{L}_{DCF} \supset \mathcal{L}_3$.
  - Jsou přijímány deterministickým ZA a deterministickým rozšířeným ZA. [[**todo: ověřit**]]
- Příklad jazyků:
  - $\{a^n b^n \mid n \geq 1\} \in \mathcal{L}_{DCF} \setminus \mathcal{L}_3$
  - $\{w \mid w \in \{a,b\}^* \wedge \#_a(w) = \#_b(w)\} \in \mathcal{L}_{DCF} \setminus \mathcal{L}_3$
  - $\{ww^R \mid w \in \{a,b\}^+\} \in \mathcal{L}_2 \setminus \mathcal{L}_{DCF}$
  - $\{a^i b^j c^k \mid i = j \vee j = k\} \in \mathcal{L}_2 \setminus \mathcal{L}_{DCF}$

**Typ 3 (regulární gramatiky)**

- Pravidla ve tvaru:
  - Pravolineární: $A \to wB \mid w, \quad A, B \in N, \quad w \in \Sigma^*$
  - Levolineární: $A \to Bw \mid w, \quad A, B \in N, \quad w \in \Sigma^*$
  - Pravoregulární: $A \to aB \mid a, \quad A, B \in N, \quad a \in \Sigma$
  - Levoregulární: $A \to Ba \mid a, \quad A, B \in N, \quad a \in \Sigma$
- Gramatiky se všemi tvary pravidel jsou ekvivalentní.
- Jazyky $\mathcal{L}_3$ jsou příjímány nedeterministickým konečným automatem (KA).
  - Varianty: nedeterministický KA, deterministický KA a rozšířený KA disponují stejnou vyjadřovací silou.
- Třída konečných jazyků $\mathcal{L}_{Fin}$ je podmnožinou regulárních jazyků, platí $\mathcal{L}_{Fin} \subset \mathcal{L}_3$.
- Příklad jazyků:
  - $\{a^n b^m \mid n \geq 10 \wedge m \leq 20\} \in \mathcal{L}_3 \setminus \mathcal{L}_{Fin}$
  - $\{abc\} \in \mathcal{L}_{Fin}$

## 19.5 Vlastnosti formálních jazyků a jejich rozhodnutelnost

- Které vlastnosti jazyků, resp. tříd jazyků, zkoumáme?
  - **Uzavřenost** vůči nějaké operaci – zda výsledek operace patří do stejné třídy.
  - **Rozhodnutelnost** problému – zda je problém rozhodnutelný, částečně rozhodnutelný a nebo nerozhodnutelný.

### Operace nad jazyky

- Základní množinové operace – sjednocení, průnik, doplněk.
- Základní jazykové operace – konkatenace, iterace, reverzace.
- Substituce
  - Nechť $\Sigma$ a $\Gamma$ jsou abecedy. Funce $\sigma : \Sigma^* \to 2^{\Gamma^*}$ se nazývá substituce právě tehdy, když
    * $\sigma(\epsilon) = \epsilon$
    * $\forall x, y \in \Sigma^* \ : \ \sigma(xy) = \sigma(x)\sigma(y)$
  - Neformálně řečeno, každý symbol vstupního řetězce je nahrazen množinou řetězců.
  - Příklad:
    * Substituce: $\sigma(a) = \{0, 00\}$, $\sigma(b) = \{1, 111, 1110\}$
    * Aplikace: $\sigma(ab) = \{0, 00\}\{1, 111, 1110\} = \{01, 0111, 01110, 001, 00111, 001110\}$
- Morfismus
  - Nechť $\Sigma$ a $\Gamma$ jsou abecedy. Funce $\phi : \Sigma^* \to \Gamma^*$ se nazývá morfismus právě tehdy, když
    * $\phi(\epsilon) = \epsilon$
    * $\forall x, y \in \Sigma^* \ : \ \phi(xy) = \phi(x)\phi(y)$
  - Neformálně řečeno, morfismus je zvláštní případ substitutce, kde každý substituovaný jazyk má právě jedno slovo.
  - Příklad:
    * Morfismus: $\phi(a) = 00$, $\phi(b) = 01$, $\phi(c) = 10$
    * Aplikace: $\phi(abac) = \phi(a)$

| | L3 | L2-D | L2 | L1 | L0-Rec | L0 |
|---|---|---|---|---|---|---|
| Sjednoceni | Ano | Ne | Ano | Ano | Ano | Ano |
| Prunik | Ano | Ne | Ne | Ano | Ano | Ano |
| Prunik s L3 | Ano | Ano | Ano | Ano | Ano | Ano |
| Konkatenace | Ano | Ne | Ano | Ano | Ano | Ano |
| Iterace | Ano | Ne | Ano | Ano | Ano | Ano |
| Doplnek | Ano | Ano | Ne | Ano | Ano | Ne |
| Reverzace | Ano | Ne | Ano | Ano | Ano | Ano |
| Substituce | Ano | Ne | Ano | Ano | Ne | Ano |
| Morfismus | Ano | Ne | Ano | Ano | Ne | Ano |
| Inv. morfismus | Ano | Ano | Ano | Ano | Ano | Ano |

Obrázek 19.2: Uzavřenost jednotlivých tříd jazyků vůči operacím.

- Vzhledem k de Morganovým zákonům:

$$L_1 \cap L_2 = \overline{\overline{L_1 \cap L_2}} = \overline{\overline{L_1} \cup \overline{L_2}}$$

  pro operace průnik, sjednoceni a doplněk platí, že libovolná jazyková třída je uzavřena buď na
  - doplněk
  - a nebo na všechny tři.
- Proč? Neformálně: ze dvou těch operací jsem schopen udělat tu třetí.

### Problémy nad jazyky

- Náležitost / nenáležitost jazyka do třídy jazyků.
- Neprázdnost jazyka – Jazyk je neprázdný, pokud obsahuje aspoň 1 řetězec.
- Prázdnost jazyka – Jazyk je prázdný, pokud neobsahuje žádný řetězec.
- Konečnost jazyka – Jazyk je konečný, pokud obsahuje konečný počet řetězců.
- Náležitost řetězce – Řetězec náleží jazyku, pokud je jeho součástí.
- Univerzalita – Jazyk obsahuje všechny řetězce nad abecedou ($L = \Sigma^*$).

- Inkluze jazyků – Jazyk $L_1$ je inkluzí jazyka $L_2$, pokud je jazyk $L_1$ podmnožinou jazyka $L_2$.
- Ekvivalence gramatik – Gramatiky $G_1$ a $G_2$ jsou ekvivalentní, pokud platí, že $L(G_1) = L(G_2)$ ($G_1$ a $G_2$ mohou být různé).

| | L3 | L2-D | L2 | L1 | L0-Rec | L0 |
|---|---|---|---|---|---|---|
| Neprazdnost | Ano | Ano | Ano | Castecne | Castecne | Castecne |
| Prazdnost | Ano | Ano | Ano | Ne | Ne | Ne |
| Konecnost | Ano | Ano | Ano | Ne | Ne | Ne |
| Nalezitost | Ano | Ano | Ano | Ano | Ano | Castecne |
| Inkluze | Ano | Ne | Ne | Ne | Ne | Ne |
| Ekvivalence g. | Ano | Ano | Ne | Ne | Ne | Ne |

Obrázek 19.3: Rozhodnutelnost problémů v jednotlivých třídách jazyků.

## 19.6 Vlastnosti regulárních jazyků

Uzávěrové vlastnosti a rozhodnutelnost problémů regulárních jazyků.

### 19.6.1 Sjednocení, průnik, doplněk, iterace, konkatenace

- Uzavřenost regulárních jazyků vůči operacím sjednocení, průnik, doplněk, iterace a konkatenace vyplývá z definice regulárních výrazů, resp. regulárních množin a ekvivalence regulárních množin a regulárních jazyků (*viz otázka regulární výrazy*).
  - Alternativně např. pro sjednocení (pro ostatní operace je to analogické). Mějme regulární jazyky $L_1$ a $L_2$. Pak musí existovat KA $M_1$ a $M_2$, které je přijímají. Pokud jazyk $L_3 = L_1 \cup L_2$ je regulární, pak musí existovat KA $M_3 = M_1 \cup M_2$, který ho přijímá.

### 19.6.2 Doplněk (*co*)

Nechť $L \subseteq \Sigma^*$ je regulární jazyk. Nechť $M = (Q, \Sigma, \delta, q_0, F)$ je úplně definovaný KA, pro který platí $L = M(L)$. Pak KA $M' = (Q, \Sigma, \delta, q_0, Q - F)$ zřejmě přijímá jazyk $L_{co} = \Sigma^* - L$, tj. doplněk jazyka $L$.

### 19.6.3 Průnik

Uzavřenost vzhledem k průniku plyne z de Morganových zákonů:

$$L_1 \cap L_2 = \overline{\overline{L_1 \cap L_2}} = \overline{\overline{L_1} \cup \overline{L_2}}$$

a tedy $L_1, L_2 \in \mathcal{L}_3 \Leftrightarrow L_1 \cap L_2 \in \mathcal{L}_3$

### 19.6.4 Regulárnost

- Nechť $L \subseteq \Sigma^*$, jak řešit problém: $L \in \mathcal{L}_3$?
  - Dokážeme existenci takového KA $M$, pro který platí $L(M) = L$.
  - S využitím Myhill-Nerodovi věty. Najdeme relaci pravé kongruence s konečným indexem na $\Sigma^*$ takovou, že jazyk $L$ je sloučením některých tříd rozkladu.

### 19.6.5 Neregulárnost

- Nechť $L \subseteq \Sigma^*$, jak řešit problém: $L \notin \mathcal{L}_3$?
  - S využitím Pumping Lemma pro regulární jazyky.
  - S využitím Myhill-Nerodovi ukážeme, že neexistuje taková relace pravé kongruence s konečným indexem na $\Sigma^*$ taková, že jazyk $L$ je sloučením některých tříd rozkladu.

### 19.6.6 Prázdnost/neprázdnost

Problém je rozhodnutelný. Můžu sestrojit algoritmus, který vyzkouší, zda je možné, se nějak dostat z výchozího stavu do nějakého konečného (problém dosažitelnosti v grafu). Formálně: K jazyku $L \in \mathcal{L}_3$ sestrojíme úplně definovaný DKA $M$, takový, že $L = L(M)$, pak

$$L(M) \neq \emptyset \Leftrightarrow \exists q \in Q \ : \ (q \in F \wedge \text{q je dosažitelný z } q_0)$$

### 19.6.7 Univerzalita

Problém je rozhodnutelný. K jazyku $L \in \mathcal{L}_3$ sestrojíme úplně definovaný DKA $M$, takový, že $L = L(M)$, pak

$$L(M) = \Sigma^* \Leftrightarrow \forall q \in Q \ : \ (q \in F \ \vee \ q \text{ je nedosažitelný z } q_0)$$

### 19.6.8 Náležitost

Problém je rozhodnutelný. K jazyku $L \in \mathcal{L}_3$ sestrojíme úplně definovaný DKA $M$, takový, že $L = L(M)$, pak

$$w \in L \Leftrightarrow (q_0, w) \vdash^* (q, \epsilon) \wedge q \in F$$

### 19.6.9 Ekvivalence jazyků

Problém je rozhodnutelný. K jazyku $L_1, L_2 \in \mathcal{L}_3$ sestrojíme úplně definovaný DKA $M_1, M_2$, takový, že $L_1 = L(M_1)$ a $L_2 = L(M_2)$. Automaty $M_1, M_2$ minimalizujeme a porovnáme jejich struktury (ignorujeme identifikátory stavů).

### 19.6.10 Konečnost

- **[[todo]]**

## 19.7 Vlastnosti bezkontextových jazyků

Uzávěrové vlastnosti a rozhodnutelnost problémů bezkontextových jazyků.

### 19.7.1 Substituce

**Věta 6.2** Třída bezkontextových jazyků je uzavřena vůči substituci.

*Důkaz.*

- Ve shodě s definicí substituce nechť $\Sigma = \{a_1, a_2, ..., a_n\}$ je abeceda bezkontextového jazyka $L$ a $L_a$ pro $a \in \Sigma$ libovolné bezkontextové jazyky. Nechť $G = (N, \Sigma, P, S)$ a $G_a = (N_a, \Sigma_a, P_a, S_a)$ pro $a \in \Sigma$ jsou gramatiky, pro které $L = L(G)$ a $L_a = L(G_a)$ pro $a \in \Sigma$.
- Předpokládejme, že $N \cap N_a = \emptyset$ a $N_a \cap N_b = \emptyset$ pro každé $a, b \in \Sigma$, $a \neq b$. Sestrojme gramatiku $G' = (N', \Sigma', P', S)$ takto:
  1. $N' = N \cup \bigcup_{a \in \Sigma} N_a$.
  2. $\Sigma' = \bigcup_{a \in \Sigma} \Sigma_a$.
  3. Nechť $h$ je morfismus na $N \cup \Sigma$ takový, že
     - $h(A) = A$ pro $A \in N$ a
     - $h(a) = S_a$ pro $a \in \Sigma$

     a nechť $P' = \{A \to h(\alpha) \mid (A \to \alpha) \in P\} \cup \bigcup_{a \in \Sigma} P_a$.
- Uvažujme libovolnou větu $a_{i_1} a_{i_2} ... a_{i_m} \in L$ a věty $x_j \in L_{a_j}$, $1 \leq j \leq m$. Pak $S \overset{*}{\underset{G'}{\Rightarrow}} S_{a_{i_1}} S_{a_{i_2}} .... S_{a_{i_m}} \overset{*}{\underset{G'}{\Rightarrow}} x_1 S_{a_{i_2}} .... S_{a_{i_m}} \overset{*}{\underset{G'}{\Rightarrow}} ... \overset{*}{\underset{G'}{\Rightarrow}} x_1 x_2 ... x_m$ a tedy $L' \subseteq L(G')$. Podobně $L(G') \subseteq L'$. □

### 19.7.2 Sjednocení, konkatenace, iterace, pozitivní iterace, morfismus

Nechť $L_a$ a $L_b$ jsou bezkontextové jazyky.

1. Uzavřenost vůči $\cup$ plyne ze substituce $L_a$, $L_b$ do jazyka $\{a, b\}$.
2. Uzavřenost vůči . plyne ze substituce $L_a$, $L_b$ do jazyka $\{ab\}$.
3. Uzavřenost vůči $*$ plyne ze substituce $L_a$ do jazyka $\{a\}^*$.
4. Uzavřenost vůči $+$ plyne ze substituce $L_a$ do jazyka $\{a\}^+$.
5. Nechť $h$ je daný morfismus a $L'_a = \{h(a)\}$ pro $a \in \Sigma$. Substitucí jazyků $L'_a$ do jazyka $L$ získáme jazyk $h(L)$.

### 19.7.3 Průnik

- Bezkontextové jazyky nejsou uzavřeny vůči průniku, protipříklad:

$$L_1 = \{a^n, b^n, c^m \mid n, m \geq 1\} \in \mathcal{L}_2$$

$$L_2 = \{a^m, b^n, c^n \mid n, m \geq 1\} \in \mathcal{L}_2$$

$$L_1 \cap L_2 = \{a^n, b^n, c^n \mid n \geq 1\} \notin \mathcal{L}_2$$

### 19.7.4 Doplněk

- Bezkontextové jazyky nejsou uzavřeny vůči doplňku.

- Předpokládejme, že bezkontextové jazyky jsou uzavřeny vůči doplňku. Potom z uzavřenosti vůči sjednocení a de Morganových zákonů:

$$L_1 \cap L_2 = \overline{\overline{L_1 \cap L_2}} = \overline{\overline{L_1} \cup \overline{L_2}}$$

  vyplývá uzavřenost vůči průniku. Což je spor.

### 19.7.5 Bezkontextovost

- Nechť $L \subseteq \Sigma^*$, jak řešit problém: $L \in \mathcal{L}_2$?
  - Dokážeme existenci takového NZA $M$, pro který platí $L(M) = L$.

### 19.7.6 Nebezkontextovost

- Nechť $L \subseteq \Sigma^*$, jak řešit problém: $L \notin \mathcal{L}_3$?
  - S využitím Pumping Lemma pro bezkontextové jazyky.

### 19.7.7 Prázdnost/neprázdnost

Zásobníkový automat je příliš komplexní model pro řešení takovýchto problémů, radši budeme vycházet z bezkontextových gramatik. K rozhodování neprázdnosti lze využít algoritmus iterativně určující množinu $N_t$ nonterminálů generujících terminální řetězce (viz přednášky). Pak $L(G) \neq \emptyset \Leftrightarrow S \in N_t$.

### 19.7.8 Náležitost

U problému příslušnosti řetězce můžeme např. určit průnik NZA s KA přijímajícím právě řetězec $w$ a pak ověřit neprázdnost (převod na gramatiku, problém prázdnosti gramatiky umíme řešit).

### 19.7.9 Konečnost

Problém konečnosti můžeme rozhodovat na základě platnosti Pumping lemma pro CFL:

- Dle Pumping lemma pro bezkontextové jazyky existuje pro každý bezkontextový jazyk $L$ konstanta $k \in \mathbb{N}$ taková, že každou větu $w \in L$, $|w| \geq k$, můžeme rozepsat jako $uvwxy$, kde $vx \neq \varepsilon$ a $|vwx| \leq k$, a $\forall i \in \mathbb{N} : uv^iwx^iy \in L$.
- Pro testování konečnosti tedy postačí ověřit, že žádný řetězec ze $\Sigma^*$ o délce mezi $k$ a $2k-1$ nepatří do daného jazyka:
  - Pokud takový řetězec existuje, může být „napumpován“ a dostáváme nekonečně mnoho řetězců patřících do daného jazyka.
  - Jestliže takový řetězec neexistuje, $k-1$ je horní limit délky řetězců $L$.
  - Pokud by existoval řetězec délky $2k$ nebo větší patřící do $L$, můžeme v něm podle Pumping lemma najít $vwx$ a vypustit $vx$. Vzhledem k tomu, že $0 < |vx| \leq k$, postupným opakováním vypouštění bychom se dostali k nutné existenci řetězce z $L$ o délce mezi $k$ a $2k-1$.
- K určení konstanty $k$ postačí reprezentovat $L$ pomocí bezkontextové gramatiky v CNF s $n$ nonterminály a zvolit $k = 2^n$ (viz důkaz Pumping lemma).

□

### 19.7.10 Ekvivalence jazyků

- Nerozhodnutelný problém, dokonce i sama inkluze je nerozhodnutelná.
- Důkaz redukcí, komplikovanější.

## 19.8 Vlastnosti kontextových jazyků

Uzávěrové vlastnosti a rozhodnutelnost problémů kontextových jazyků.

### 19.8.1 Sjednocení, průnik, konkatenace, iterace, komplement

**Věta 8.10** Třída kontextových jazyků je uzavřena vůči operacím $\cup$, $\cap$, ., $*$ a komplementu.

*Důkaz.*

- Uzavřenost vůči $\cup$, $\cap$, . a $*$ lze ukázat stejně jako u rekurzívně spočetných jazyků.
- Důkaz uzavřenosti vůči komplementu je značně komplikovaný (všimněme si, že LOA je *nedeterministický* a nelze tudíž užít konstrukce použité u rekurzívních jazyků) – zájemci naleznou důkaz v doporučené literatuře.

□

❖ Poznamenejme, že již víme, že u kontextových jazyků

- lze rozhodovat členství věty do jazyka (rekurzívnost) a
- nelze rozhodovat inkluzi jazyků (neplatí ani pro bezkontextové jazyky).

❖ Dále lze ukázat, že pro kontextové jazyky nelze rozhodovat prázdnost jazyka (užije se redukce z Postova problému přiřazení – viz další přednášky).

## 19.9 Vlastnosti obecných jazyků

Uzávěrové vlastnosti a rozhodnutelnost problémů obecných jazyků.

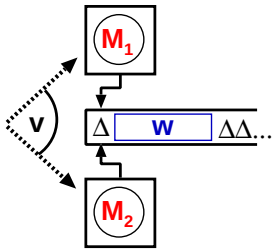
- Zkoumáme pro rekurzivně vyčíslitelné $\mathcal{L}_{RE}$ a rekurzivní $\mathcal{L}_{Rec}$ zároveň.

### 19.9.1 Sjednocení, průnik, konkatenace, iterace

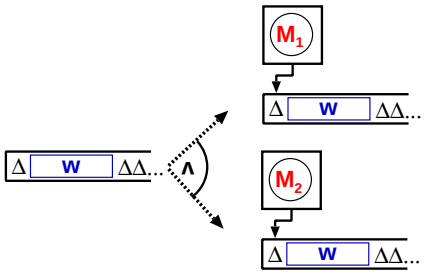
**Věta 8.4** Třídy rekurzívních a rekurzívně vyčíslitelných jazyků jsou uzavřeny vůči operacím $\cup$, $\cap$, . a $*$.

*Důkaz.* Nechť $L_1$, $L_2$ jsou jazyky přijímané TS $M_1$, $M_2$. Zřejmě můžeme předpokládat, že množiny stavů TS $M_1$, $M_2$ jsou disjunktní.

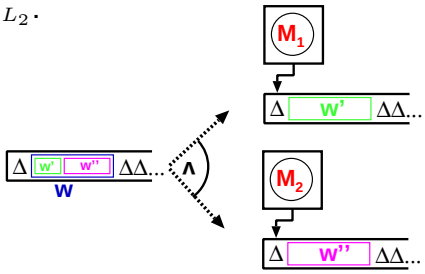
- NTS $M_{L_1 \cup L_2}$, $L(M_{L_1 \cup L_2}) = L_1 \cup L_2$, sestrojíme tak, že sjednotíme po složkách stroje $M_1$ a $M_2$, zavedeme nový počáteční stav, z něj nedeterministické přechody přes $\Delta/\Delta$ do obou původních počátečních stavů a sloučíme původní koncové stavy do jediného nového koncového stavu.



- Třípáskový TS $M_{L_1 \cap L_2}$, $L(M_{L_1 \cap L_2}) = L_1 \cap L_2$, okopíruje vstup z první pásky na druhou, na ní simuluje stroj $M_1$, pokud ten přijme, okopíruje vstup z první pásky na třetí, na ní simuluje stroj $M_2$, pokud i ten přijme, přijme i stroj $M_{L_1 \cap L_2}$.




- Třípáskový NTS $M_{L_1.L_2}$, $L(M_{L_1.L_2}) = L_1.L_2$, okopíruje nedeterministicky zvolený prefix vstupu z první pásky na druhou, na ní simuluje stroj $M_1$, pokud ten přijme, okopíruje zbytek vstupu z první pásky na třetí, na ní simuluje stroj $M_2$, pokud i ten přijme, přijme i stroj $M_{L_1.L_2}$.




*Důkaz pokračuje dále.*

- Dvoupáskový NTS $M_{L_1^*}$, $L(M_{L_1^*}) = L_1^*$, je zobecněním předchozího stroje: po částech kopíruje vstup z první pásky na druhou a na ní simuluje opakovaně stroj $M_1$. Obsah druhé pásky má ohraničený speciálními značkami a po každé simulaci stroje $M_1$ ho smaže. Umožňuje samozřejmě posuv pravé značky dále doprava při nedostatku místa.

Jsou-li stroje $M_1$ a $M_2$ úplné, je možné vybudovat stroje podle výše uvedených pravidel také jako úplné (u $M_{L_1 \cup L_2}$, $M_{L_1 \cap L_2}$, $M_{L_1.L_2}$ je to okamžité, u $M_{L_1^*}$ nepřipustíme načítání prázdného podřetězce vstupu z 1. na 2. pásku – pouze umožníme jednorázově přijmout prázdný vstup). To dokazuje uzavřenost vůči uvedeným operacím také u rekurzívních jazyků.

□

### 19.9.2 Doplněk

**Věta 8.5** Třída rekurzívních jazyků je uzavřena vůči komplementu.

*Důkaz.* TS $M$ přijímající rekurzívní jazyk $L$ vždy zastaví. Snadno upravíme $M$ na $M'$, který při nepřijetí řetězce vždy přejde do unikátního stavu $q_{reject}$. TS $\overline{M}$, $L(\overline{M}) = \overline{L}$, snadno dostaneme z $M'$ záměnou $q_F$ a $q_{reject}$. □

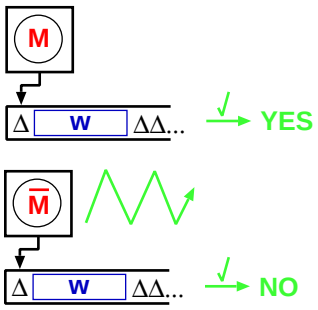
❖ Třída rekurzívně vyčíslitelných jazyků není uzavřena vůči komplementu!

- Výše uvedené konstrukce nelze užít – cyklení zůstane zachováno.
- Důkaz neuzavřenosti bude uveden v dalších přednáškách.

**Věta 8.6** Jsou-li $L$ i $\overline{L}$ rekurzívně vyčíslitelné, pak jsou oba rekurzívní.

*Důkaz.*
Mějme $M$, $L(M) = L$, a $\overline{M}$, $L(\overline{M}) = \overline{L}$. Úplný TS přijímající $L$ sestrojíme takto:

- Použijeme dvě pásky. Na jedné budeme simulovat $M$, na druhé $\overline{M}$. Simulace se bude provádět proloženě krok po kroku: krok $M$, krok $\overline{M}$, krok $M$, ...
- Přijmeme, právě když by přijal $M$, zamítneme abnormálním zastavením, právě když by přijal $\overline{M}$. Jedna z těchto situací určitě nastane v konečném počtu kroků.




Existence úplného TS pro $\overline{L}$ plyne z uzavřenosti rekurzívních jazyků vůči komplementu. □

❖ Důsledkem výše uvedených vět je mj. to, že pro $L$ a $\overline{L}$ musí vždy nastat jedna z následujících situací:

- L i $\overline{L}$ jsou rekurzívní,
- L ani $\overline{L}$ nejsou rekurzívně vyčíslitelné,
- jeden z těchto jazyků je rekurzívně vyčíslitelný, ale ne rekurzívní, druhý není rekurzívně vyčíslitelný.

# Kapitola 20

# TIN – Konečné automaty (jazyky přijímané KA, varianty KA, minimalizace KA, Mihill-Nerodova věta).

## 20.1 Zdroje

- `tin_2021_merged.pdf`
- `TIN_2020-09-22.mp4`
- `TIN_2020-09-29.mp4`
- `TIN_2020-10-02_demo.mp4`
- `TIN_2020-10-06.mp4`
- `TIN_2020-10-16_demo.mp4`

## 20.2 Konečný automat

Konečné automaty dokáží přijímat regulární jazyky.

**Definice** Konečný automat (KA) je pětice $M = (Q, \Sigma, \delta, q_0, F)$, kde

- $Q$ je konečná množina stavů;
- $\Sigma$ je vstupní abeceda;
- $\delta$ je přechodová funkce (parciální funkce),
  - $\delta : Q \times \Sigma \to 2^Q$;
- $q_0$ je výchozí stav,
  - $q_0 \in Q$;
- $F$ je množina koncových stavů,
  - $F \subseteq Q$;

**Konfigurace** Konfigurace KA je dvojice $(q, w) \in Q \times \Sigma^*$, kde

- $q$ je aktuální stav;
- $w$ je nezpracovaná část vstupního řetězce.

**Počáteční konfigurace** Počáteční konfigurace je taková konfigurace $(q, w)$, kde $w \in \Sigma^*$ je vstupní řetězec a $q \in Q$ je výchozí stav.

**Finální konfigurace** Finální konfigurace je taková konfigurace $(q, w)$, kde $w = \epsilon$ je $q \in F$.

**Přechod** Přechod (krok výpočtu) KA je binární relace (značíme $\vdash$) na množině konfigurací $\vdash \subseteq (Q \times \Sigma^*)^2$, taková, že

$$(q, w) \vdash (q', w') \Leftrightarrow \exists a \in \Sigma \ : \ w = aw' \wedge q' \in \delta(q, a)$$

**Jazyk přijímaný** Mějme KA $M = (Q, \Sigma, \delta, q_0, F)$ a jazyk $L(M)$, který je přijímaný KA $M$.

$$L(M) = \{w \mid w \in \Sigma^* \wedge (q_0, w) \vdash^* (q_f, \epsilon) \wedge q_f \in F\}$$

**Dosažitelné a nedosažitelné stavy** Mějme KA $M = (Q, \Sigma, \delta, q_0, F)$. Stav $q \in Q$ je dosažitelný, pokud platí $\exists w \in \Sigma^* : (q_0, w) \vdash^* (q, \epsilon)$. Stav je nedosažitelný, pokud není dosažitelný.

**Relace nerozlišitelnosti** Relace nerozlišitelnosti je binární relace nad množinou stavů. Neformálně říká, že z jednoho ze stavů, automat dokáže akceptovat rozlišovací řetězec, z druhého nikoliv. Formálně: nechť $p, q \in Q$ jsou rozlišitelné, pak platí

$$\forall w \in \Sigma^* : (p, w) \vdash^* (p', \epsilon) \wedge (q, w) \vdash^* (q', \epsilon) \wedge ((p' \in F \wedge q' \notin F) \vee (p' \notin F \wedge q' \in F))$$

Dva stavy jsou nerozlišitelné, pokud nejsou rozlišitelné.

**k-nerozlišitelnost** Říkáme, že stavy $q_1, q_2 \in Q$ jsou k-nerozlišitelné a píšeme $q_1 \equiv^k q_2$, právě když neexistuje $w \in \Sigma^*, |w| \leq k$, který rozlišuje $q_1$ a $q_2$. Lze dokázat, že $\equiv$ je relací akvivalence na $Q$ (reflexivní, symetrická, tranzitivní). Neformálně relace říká, jak dlouhé slovo dané stavy rozlišuje.

## 20.3 Varianty konečného automatu

Všechny varianty konečného automatu mají stejnou vyjadřovací sílu (jsou mezi sebou převoditelné).

**Nedeterministický konečný automat** Nedeterministický konečný automat (NKA) je výchozí konečný automat (NKA = KA).

**Deterministický konečný automat** Deterministický konečný automat (DKA) se od NKA liší pouze tvarem přechodové funkce. Formálně:

- $\delta$ je přechodová funkce (parciální funkce),
  - $\delta : Q \times \Sigma \to Q$

**Rozšířený konečný automat** Rozšířený (nedeterministický) konečný automat (RKA) se od NKA liší pouze tvarem přechodové funkce. Rozšiřuje ji o tzv. epsilon přechody. Formálně:

- $\delta$ je přechodová funkce (parciální funkce),
  - $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$

**Úplně definovaný konečný automat** Úplně definovaný konečný automat se od NKA liší pouze tvarem přechodové funkce. Formálně:

- $\delta$ je přechodová funkce (totální funkce),
  - $\delta : Q \times \Sigma \to 2^Q$
  - $\forall q \in Q \ \forall a \in \Sigma : \delta(q, a) \in Q$

**Redukovaný úplně definovaný deterministický konečný automat** Úplně definovaný DKA nazýváme redukovaný (také minimální), jestliže žádný $q \in Q$ není nedosažitelný a žádná dvojice $p, q \in Q$ není nerozlišitelná.

## 20.4 Determinizace NKA

**Algoritmus**

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

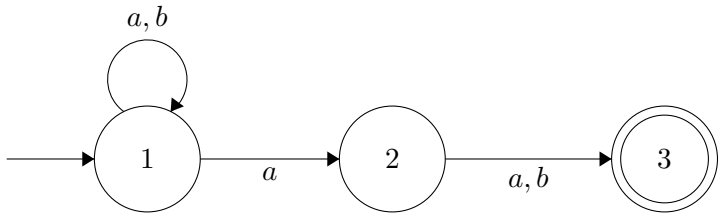
Výstup: DKA $M' = (Q', \Sigma, \delta', q_0', F')$, $L(M) = L(M')$

Metoda:

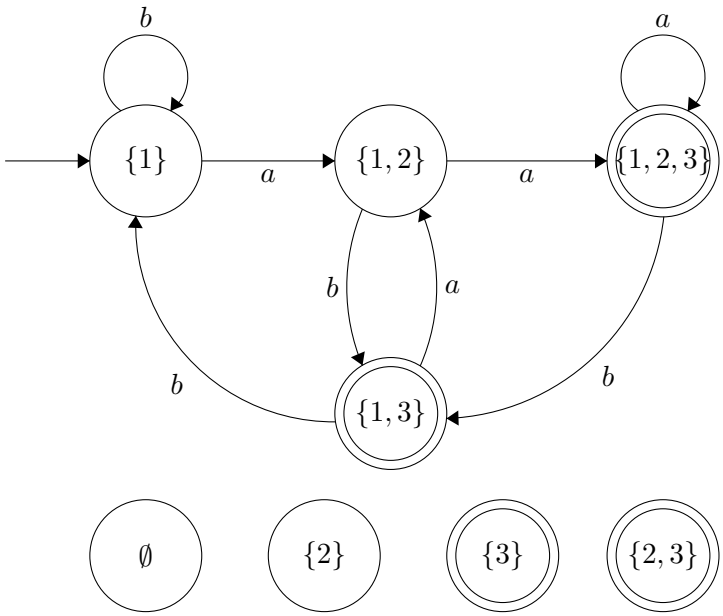
1. Polož $Q' = 2^Q$.
2. Polož $q_0' = \{q_0\}$.
3. Polož $F' = \{S \mid S \in 2^Q \wedge S \cap F \neq \emptyset\}$.
4. Pro všechna $S \in 2^Q$ a pro všechna $a \in \Sigma$ polož:
   - $\delta'(S, a) = \bigcup_{q \in S} \delta(q, a)$.

Obrázek 20.1: Algoritmus determinizace NKA.

**Příklad**



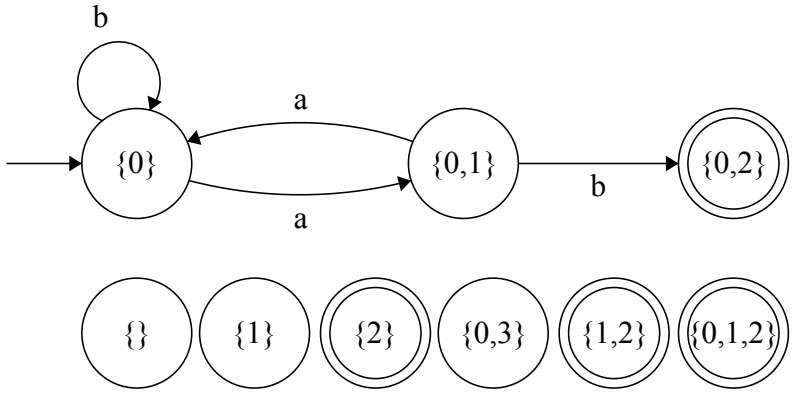

Obrázek 20.2: Nedeterministický konečný automat $M$.




Obrázek 20.3: Deterministický konečný automat $M'$, $L(M) = L(M')$.

## 20.5 Minimalizace konečného automatu

- Proč? Zrychlení vykonávání konečného automatu.
- Jak? Odstraněním nepotřebných přechodů a zmenšením počtu stavů.
- Postup (předpokládá DKA):
  1. Odstranění nedosažitelných stavů.
  2. Převod DKA na úplně definovaný DKA.
  3. Převod úplně definovaný DKA na redukovaný DKA (odstranění nerozlišitelných stavů).
  4. Odstranění *sink* stavu.




Obrázek 20.4: Deterministický konečný automat.

### 20.5.1 Eliminace nedosažitelných stavů

**Algoritmus 3.4** Eliminace nedosažitelných stavů

**Vstup:** Deterministický konečný automat $M = (Q, \Sigma, \delta, q_0, F)$.

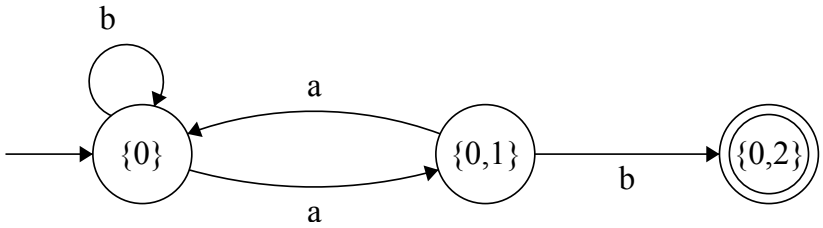
**Výstup:** Deterministický konečný automat $M'$ bez nedosažitelných stavů, $L(M) = L(M')$.

**Metoda:**

1. $i := 0$
2. $S_i := \{q_0\}$
3. `repeat`
4. $\quad S_{i+1} := S_i \cup \{q \mid \exists p \in S_i \ \exists a \in \Sigma : \delta(p, a) = q\}$
5. $\quad i := i + 1$
6. `until` $S_i = S_{i-1}$
7. $M' := (S_i, \Sigma, \delta_{|S_i}, q_0, F \cap S_i)$

Základem algoritmu minimalizace deterministického konečného automatu je koncept nerozlišitelných stavů.

Obrázek 20.5: Algoritmus eliminace nedosažitelných stavů.

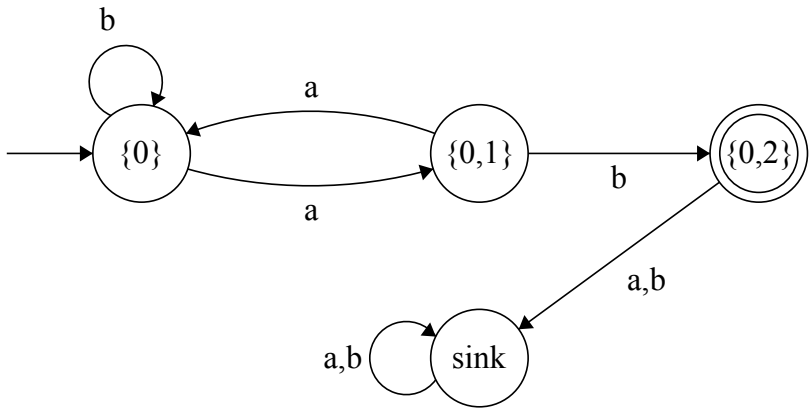



Obrázek 20.6: Deterministický konečný automat bez nedosažitelných stavů.

### 20.5.2 Převod DKA na úplně definovaný

- Vstup: DKA $M = (Q, \Sigma, \delta, q_0, F)$

- Výstup: úplně definovaný DKA $M'$
- Metoda:
  1. $\forall q \in Q\ \forall a \in \Sigma : \delta(q,a) = \emptyset \Rightarrow \delta'(q,a) = q_s$
  2. $\forall a \in \Sigma : \delta'(q_s,a) = q_s$
  3. $M' = (Q \cup \{q_s\}, \Sigma, \delta \cup \delta', q_o, F)$
- $q_s$ je tzv. *sink* stav.




Obrázek 20.7: Úplně definovaný deterministický konečný automat.

### 20.5.3 Převod úplně definovaného DKA na redukovaný

**Algoritmus 3.5** Převod na redukovaný deterministický konečný automat.
**Vstup:** Úplně definovaný DKA $M = (Q, \Sigma, \delta, q_0, F)$.
**Výstup:** Redukovaný DKA $M' = (Q', \Sigma, \delta', q_0', F')$, $L(M) = L(M')$.
**Metoda:**

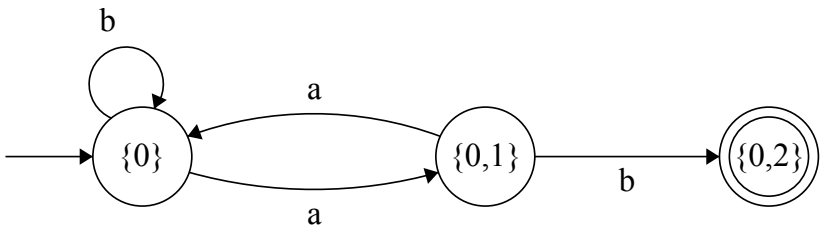
1. Odstraň nedosažitelné stavy s využitím algoritmu 3.4.
2. $i := 0$
3. $\overset{0}{\equiv} := \{(p,q) \mid p \in F \Longleftrightarrow q \in F\}$
4. `repeat`
5. $\quad \overset{i+1}{\equiv} := \{(p,q) \mid p \overset{i}{\equiv} q \wedge \forall a \in \Sigma : \delta(p,a) \overset{i}{\equiv} \delta(q,a)\}$
6. $\quad i := i + 1$
7. `until` $\overset{i}{\equiv} = \overset{i-1}{\equiv}$
8. $Q' := Q/\overset{i}{\equiv}$
9. $\forall p,q \in Q\ \forall a \in \Sigma : \delta'([p],a) = [q] \Leftrightarrow \delta(p,a) = q$
10. $q_0' = [q_0]$
11. $F' = \{[q] \mid q \in F\}$

**Poznámka 3.2** Výraz $[x]$ značí ekvivalenční třídu určenou prvkem $x$.

Obrázek 20.8: Algoritmus převodu úplně definovaného DKA na redukovaný.

### 20.5.4 Odstranění *sink* stavu

- Vstup: DKA $M = (Q, \Sigma, \delta, q_0, F)$
- Výstup: DKA $M'$ bez *sink* stavu
- Metoda:
  1. $Q' = \{q \mid q \in Q \wedge w \in \Sigma^* \wedge q_f \in F \wedge (q, w) \vdash^* (q_f, \epsilon)\}$
  2. $\delta' = \{(p, a, q) \mid \delta(p, a) = q \wedge q \notin Q - Q'\}$
  3. $M' = \{Q', \Sigma, \delta', q_0, F\}$




Obrázek 20.9: Úplně definovaný deterministický konečný automat bez *sink* stavu.

## 20.6 Myhill-Nerodova věta

**Pumping Lemma** Pumping Lemma poskytuje nutnou podmínku, ale nikoliv dostačující pro regulární jazyky, tj. když jazyk je regulární, tak podmínka musí platit. To znamená, že pomocí PL můžeme dokázat, že jazyk není regulární. Ale nemůžeme použít PL k důkazu regularity jazyky. Myšlenka Pumping Lemma spočívá v tom, že pokud mám $w \in L$, jehož délka je větší, než je počet stavů KA, který daný jazyk přijímá, tak tam musí být někde smyčka. A tím pádem, můžu tu smyčku buď úplně vypustit a nebo zopakovat vícekrát.

**Mihill-Nerodova věta – kontext** Mihill-Nerodova věta charakterizuje jazyky nad $\Sigma^*$ a jejich vztah ke konečným automatům. Je silnější než Pumping Lemma, protože poskytuje nutné a dosačující podmínky pro to, aby jazyk byl regulární. Pomocí ní můžeme dokázat, že jazyk je a nebo není regulární.

**Relace ekvivalence** Relace ekvivalence je binární relace (značíme $\sim$), která je reflexivní, symetrická a tranzitivní. Rozkládá množinu nad kterou je definovaná na tzv. **třídy ekvivalence**. Pro každé dvojice tříd ekvivalence platí, že jsou vzájemně disjunktní. Sloučením všech tříd ekvivalence dostaneme původní množinu. Počet tříd rozkladu je tzv. **index ekvivalence**. Tříd může být i nekonečno, pak index definujeme jako $\infty$. Relaci ekvivalence nad množinou $\Sigma^*$ zapisujeme jako $\Sigma^* / \sim$.

**Pravá kongruence** Pravá kongruence (pravá invariance) je speciální typ relace ekvivalence, která splňuje požadovanou vlatnost. Nechť $\Sigma$ je abeceda a $\sim$ je relace ekvivalence nad $\Sigma^*$. Relace ekvivalence $\sim$ je pravou kongruencí pokud platí:

$$\forall u, v, w \in \Sigma^* \ : \ u \sim v \Rightarrow uw \sim vw$$

Myšlenka: Pro všechny slova $u, v$ z jazyka platí, že pokud jsem se přes ně dostal do nějakého (stejného) stavu, tak když k oboum připojím slovo $w$, tak se zase dostanu do stejného stavu.

**Prefixová ekvivalence** Nechť $L$ je libovolný (ne nutně regulární) jazyk nad abecedou $\Sigma$. Na množině $\Sigma^*$ definujeme relaci $\sim_L$ zvanou prefixová ekvivalence pro $L$ takto:

$$u \sim_L v \Leftrightarrow (\forall w \in \Sigma^* \ : \ uw \in L \Leftrightarrow vw \in L)$$

Myšlenka: slova $u$ a $v$ dovedou konečný automat do stejného stavu.

**Mihill-Nerodova věta** Jazyk $L \subseteq \Sigma^*$ je regulární právě když:

- $\exists$KA $M : L(M) = L$
- Existuje pravá kongruence $\sim$ konečného indexu na $\Sigma^*$ taková, že $L$ je sjednocením vybraných tříd rozkladu $\Sigma^*/\sim$.
- Relace prefixové ekvivalence $\sim_L$ má konečný index (existuje minimální KA, který odpovídá).

Pokud dokážu, že nad daným jazykem neexistuje relace pravé kongruence s konečným indexem, tak jazyk není regulární.

# Kapitola 21

# TIN – Regulární množiny, regulární výrazy a rovnice nad regulárními výrazy.

## 21.1 Zdroje

- `tin_2021_merged.pdf`
- `TIN_2020-09-29.mp4`

## 21.2 Regulární množiny

Regulární jazyky lze definovat pomocí regulárních gramatik, konečných automatů a regulárních množin.

**Definice** Nechť $\Sigma$ je konečná abeceda. Regulární množiny nad $\Sigma$ definujeme rekurzivně takto:

- $\emptyset$ je regulární množina nad $\Sigma$;
- $\{\epsilon\}$ je regulární množina nad $\Sigma$;
- $\{a\}$ je regulární množina nad $\Sigma$ pro $\forall a \in \Sigma$;
- jsou-li $P$ a $Q$ regulární množiny nad $\Sigma$, pak také
  - $P \cup Q$,
  - $P \,.\, Q$,
  - $P^*$

  jsou regulární množiny nad $\Sigma$.
- Žádné jiné množiny, než ty, které lze získat pomocí výše uvedených pravidel, nejsou regulárními množinami.

**Příklad (a)** Jazyk definovaný regulární množinou.

$$L_{RM} = (\{a\} \cup \{d\}) \,.\, (\{b\}^*) \,.\, \{c\}$$

## 21.3 Regulární výrazy

Regulární výrazy jsou pouze zkrácený zápis regulárních množin.

**Definice** Regulární výrazy nad $\Sigma$ a regulární množiny, které označují, jsou rekurzivně definovány takto:

- $\emptyset$ je regulární výraz označují regulární množinu $\emptyset$;
- $\epsilon$ je regulární výraz označují regulární množinu $\{\epsilon\}$;
- $a$ je regulární výraz označují regulární množinu $\{a\}$ pro $\forall a \in \Sigma$;
- jsou-li $p$ a $q$ regulární výrazy označují regulární množiny $P$ a $Q$, pak
  - $(p + q)$ je regulární výraz označující regulární množinu $P \cup Q$,
  - $(pq)$ je regulární výraz označující regulární množinu $P \,.\, Q$,
  - $(p^*)$ je regulární výraz označující regulární množinu $P^*$.

**Příklad (b)** Jazyk definovaný regulárním výrazem, platí $L_{RM} = L_{RV}$.

$$L_{RV} = (a + d)b^*c$$

**Příklad (c)** Jazyk definovaný regulární gramatikou $G$, platí $L_{RM} = L_{RV} = L(G)$.

$$G = (\{S, A\}, \{a, b, c, d\}, P, S)$$

$$P = \{S \to aA \mid dA, A \to bA \mid c\}$$

**Příklad (d)** Jazyk definovaný konečným automatem $M$, platí $L_{RM} = L_{RV} = L(G) = L(M)$.

$$M = (\{q_0, q_1, q_2\}, \{a, b, c, d\}, \delta, q_0, \{q_2\})$$

$$\delta(q_0, a) = q_1 \ , \ \delta(q_0, d) = q_1 \ , \ \delta(q_1, b) = q_1 \ , \ \delta(q_1, c) = q_2$$

## 21.4 Rovnice nad regulárními výrazy

**Definice** Rovnice, jejímiž složkami jsou koeficienty a neznámé, které reprezentují (dané a hledané) regulární výrazy, nazýváme rovnicemi nad regulárními výrazy.

**Příklad** Uvažujme rovnici nad regulárními výrazy nad abecedou $\Sigma = \{a, b\}$

$$X = aX + b$$

- Řešením je regulární výraz $X = a^*b$
- Důkaz:
  - LS $= a^*b$
  - PS $= a(a^*b) + b = a^+b + b = (a^+ + \epsilon)b = a^*b$

**Shrnutí**

- Ne vždy existuje jediné řešení rovnice nad regulárními výrazy.
- Regulární gramatiky, nedeterministické/deterministické/rozšířené konečné automaty a regulární výrazy mají ekvivalentní vyjadřovací sílu.

# Kapitola 22

# TIN – Zásobníkové automaty (jazyky přijímané ZA, varianty ZA).

## 22.1 Zdroje

- `tin_2021_merged.pdf`
- `TIN_2020-10-06.mp4`
- `TIN_2020-10-13.mp4`
- `TIN_2020-10-16_demo.mp4`

## 22.2 Zásobníkový automat

Zásobníkové automaty dokáží přijímat bezkontextové jazyky.

**Definice** Zásobníkový automat (ZA) je sedmice $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, kde

- $Q$ je konečná množina stavů;
- $\Sigma$ je vstupní abeceda;
- $\Gamma$ je zásobníková abeceda;
- $\delta$ je přechodová funkce (parciální funkce),
  - $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to 2^{Q \times \Gamma^*}$;
- $q_0$ je výchozí stav,
  - $q_0 \in Q$;
- $Z_0$ je výchozí symbol na zásobníku,
  - $Z_0 \in \Gamma$;
- $F$ je množina koncových stavů,
  - $F \subseteq Q$;

**Konfigurace** Konfigurace ZA je trojice $(q, w, \alpha) \in Q \times \Sigma^* \times \Gamma^*$, kde

- $q$ je aktuální stav;
- $w$ je nezpracovaná část vstupního řetězce;
- $\alpha$ je obsah zásobníku.

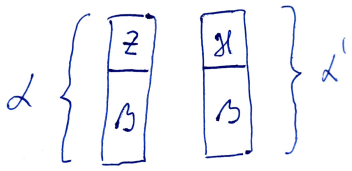
**Počáteční konfigurace** Počáteční konfigurace ZA je taková konfigurace $(q, w, \alpha)$, kde $w \in \Sigma^*$ je vstupní řetězec, $q \in Q$ je výchozí stav a $\alpha = Z \in \Gamma$ je výchozí symbol na zásobníku.

**Koncová konfigurace** Koncová konfigurace ZA je taková konfigurace $(q, w, \alpha)$, kde $q \in F$, $w = \epsilon$ a $\alpha \in \Gamma^*$.

**Přechod** Přechod (krok výpočtu) ZA je binární relace (značíme $\vdash$) na množině konfigurací $\vdash \subseteq (Q \times \Sigma^* \times \Gamma^*)^2$, taková, že

$$(q, w, \alpha) \vdash (q', w', \alpha') \Leftrightarrow \exists a \in (\Sigma \cup \epsilon),\ \exists Z \in \Gamma,\ \exists \beta, \gamma \in \Gamma^* \ :$$

$$:\ w = aw' \wedge \alpha = Z\beta \wedge \alpha' = \gamma\beta \wedge (q', \gamma) \in \delta(q, a, Z)$$




Obrázek 22.1: Stav zásobníku během přechodu ZA. Platí $Z \in \Gamma,\ \alpha, \alpha', \beta, \gamma \in \Gamma^*$.

**Jazyk přijímaný** Mějme ZA $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ a jazyk $L(M)$, který je přijímaný ZA $M$.

$$L(M) = \{w \mid w \in \Sigma^* \wedge (q_0, w, Z_0) \vdash^* (q_f, \epsilon, \epsilon) \wedge q_f \in F\}$$

Avšak existují různé varianty. ZA může přijímat i v $q_f \in F$: pouze při vyprázdnění vstupu, nebo pouze při vyprázdnění zásobníku a nebo při vyprázdnění vstupu a vyprázdnění zásobníku.

## 22.3 Varianty zásobníkového automatu

Zde je změna oproti konečným automatům. Všechny varianty zásobníkového automatu nemají stejnou vyjadřovací sílu (nejsou mezi sebou převoditelné).

### 22.3.1 Nedeterministický zásobníkový automat

Nedeterministický zásobníkový automat (NZA) je výchozí zásobníkový automat (NZA = ZA).

### 22.3.2 Rozšířený zásobníkový automat

Rozšířený (nedeterministický) zásobníkový automat (RNZA) disponuje stejnou vyjadřovací sílou jako NZA (jsou mezi sebou převoditelné).

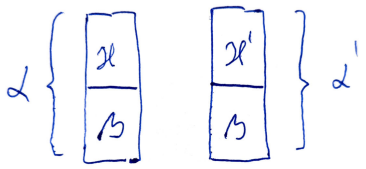
**Definice** RNZA se od NZA liší tvarem přechodové funkce, umožňuje ze zásobníku konzumovat více než 1 symbol. Formálně:

- $\delta$ je přechodová funkce (parciální funkce),
  - $\delta : Q \times (\Sigma \cup \{\epsilon\} \times \Gamma^*) \to 2^{Q \times \Gamma^*}$

**Konfigurace, přechod** Konfigurace RNZA a NZA jsou shodné. Přechod se liší díky jinému tvaru přechodové funkce. Formálně: Přechod RNZA je binární relace (značíme $\vdash$) na množině konfigurací $\vdash \subseteq (Q \times \Sigma^* \times \Gamma^*)^2$, taková, že

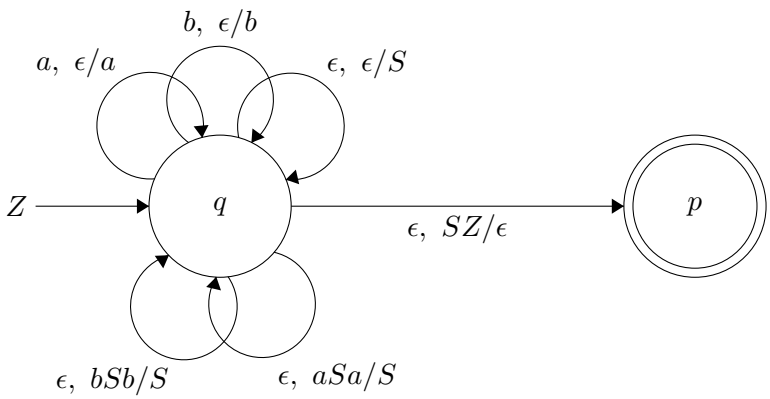
$$(q, w, \alpha) \vdash (q', w', \alpha') \Leftrightarrow \exists a \in (\Sigma \cup \epsilon),\ \exists \beta, \gamma, \gamma' \in \Gamma^* \ :$$

$$:\ w = aw' \wedge \alpha = \gamma\beta \wedge \alpha' = \gamma'\beta \wedge (q', \gamma') \in \delta(q, a, \gamma)$$




Obrázek 22.2: Stav zásobníku během přechodu RNZA. Platí $\alpha, \alpha', \beta, \gamma, \gamma' \in \Gamma^*$.

**Příklad** Příklad RNZA, který přijímá jazyk $L = \{ww^R \mid w \in \{a, b\}^+\}$. Platí $L \in \mathcal{L}_2 \setminus \mathcal{L}_{DCF}$.




Obrázek 22.3: Příklad rozšířeného zásobníkového automatu.

### 22.3.3 Deterministický zásobníkový automat

Deterministický zásobníkový automat (DZA) disponuje menší vyjadřovací sílou než NZA (resp. RNZA). Jazyky přijímané DZA označujeme jako deterministické bezkontextové jazyky ($\mathcal{L}_{DCF}$). Formálně: $\mathcal{L}_{DCF} \subset \mathcal{L}_2$ a tedy $\exists L \in \mathcal{L}_2 : L \notin \mathcal{L}_{DCF}$.

**Zdroje nedeterminismu NZA**

- volba dalšího stavu z množiny stavů ($q \in 2^{Q \times \Gamma^*}$),
- rozhodnutí, zda číst další znak ze vstupu ($a \in (\Sigma \cup \{\epsilon\})$).

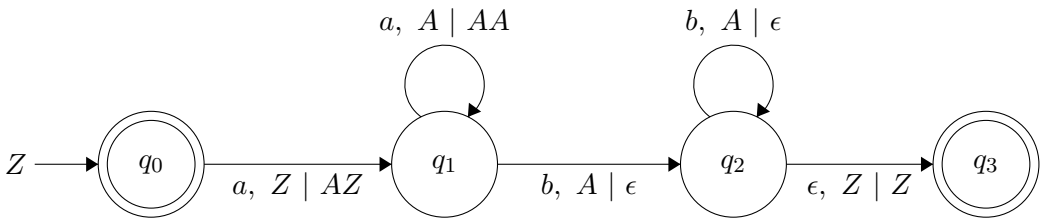
**Definice** Formálně:

$$\forall q \in Q\ \forall a \in \Sigma\ \forall Z \in \Gamma\ :$$

$$: \ ( \ |\delta(q, a, Z)| \leq 1 \ \wedge \ |\delta(q, \epsilon, Z)| = 0) \ \vee$$

$$\vee \ ( \ |\delta(q, a, Z)| = 0 \ \wedge \ |\delta(q, \epsilon, Z)| \leq 1)$$

**Příklad** Příklad DZA, který přijímá jazyk $L = \{a^n b^n \mid n \geq 0\}$. Platí $L \in \mathcal{L}_{DCF} \setminus \mathcal{L}_3$.




Obrázek 22.4: Příklad deterministického zásobníkového automatu.

### 22.3.4 Deterministický rozšířený zásobníkový automat

**Zdroje nedeterminismu RNZA**

- volba dalšího stavu z množiny stavů ($q \in 2^{Q \times \Gamma^*}$),
- rozhodnutí, zda číst další znak ze vstupu ($a \in (\Sigma \cup \{\epsilon\})$),
- rozhodnutí, kolik znaků číst ze zásobníku ($\gamma \in \Gamma^*$).

# Kapitola 23

# TIN – Turingovy stroje (jazyky přijímané TS, varianty TS, lineárně omezené automaty, vyčíslitelné funkce).

## 23.1 Zdroje

- `tin_2021_merged.pdf`
- `TIN_2020-10-20.mp4`
- `TIN_2020-10-27.mp4`
- `TIN_2020-11-03.mp4`
- `TIN_2020-11-24_p1.mp4`

## 23.2 Turingův stroj

- Turingovy stroje jsou velmi robustní a jejich různé úpravy (determinismus, nedeterminismus, počet pásek, ...) mají ekvivalentní vyjadřovací sílu z hlediska rozhodnutelnosti, ale z hlediska složitosti mají různou vyjadřovací sílu.
- Po pásce se můžeme pohybovat oběma směry. Páska je z prava nekonečná.
- Na TS lze nahlížet také jako na funkci, která má vstup počáteční pásku, pak nějaký výpočet (činnost TS) a jako výstup vrací stav pásky.

**Definice** Turingův stroj (TS) je šestice $M = (Q, \Sigma, \Gamma, \delta, q_0, g_f)$, kde

- $Q$ je konečná množina stavů;
- $\Sigma$ je vstupní abeceda (symboly, které se mohou vyskytovat na výchozím stavu pásky),
  - $\Delta \notin \Sigma$;
- $\Gamma$ je pásková abeceda (symboly, které je možné zapisovat na pásku),
  - $\Sigma \subset \Gamma \ \wedge \ \Delta \in \Gamma \ \wedge \ L, R \notin \Gamma$;
  - Symbol $\Delta$ značí tzv. blank (prázdný symbol), který se vyskytuje na místech pásky, která nebyla ještě použita (může ale být na pásku zapsán i později).

- $\delta$ je přechodová funkce (parciální funkce),
  - $\delta : (Q - \{q_f\}) \times \Gamma \to Q \times (\Gamma \cup \{L, R\})$;
- $q_0$ je výchozí stav,
  - $q_0 \in Q$;
- $q_f$ je koncový stav,
  - $q_f \in Q$;

**Konfigurace** Konfigurace TS je trojice $(q, \alpha, n) \in Q \times \{\gamma\Delta^\omega \mid \gamma \in \Gamma^*\} \times \mathbb{N}$, kde

- $q$ je aktuální stav;
- $\alpha$ značí stav pásky;
- $n$ značí pozici hlavy.

**Počáteční konfigurace** Počáteční konfigurace je taková konfigurace $(q_0, \gamma\Delta^\omega, 0)$, kde $\gamma\Delta^\omega$ je výchozí stav pásky a $q_0$ je výchozí stav.

**Koncová konfigurace** Koncová konfigurace je taková konfigurace $(q_f, \gamma\Delta^\omega, n)$, kde $\gamma\Delta^\omega$ je stav pásky a $q_f$ je koncový stav.

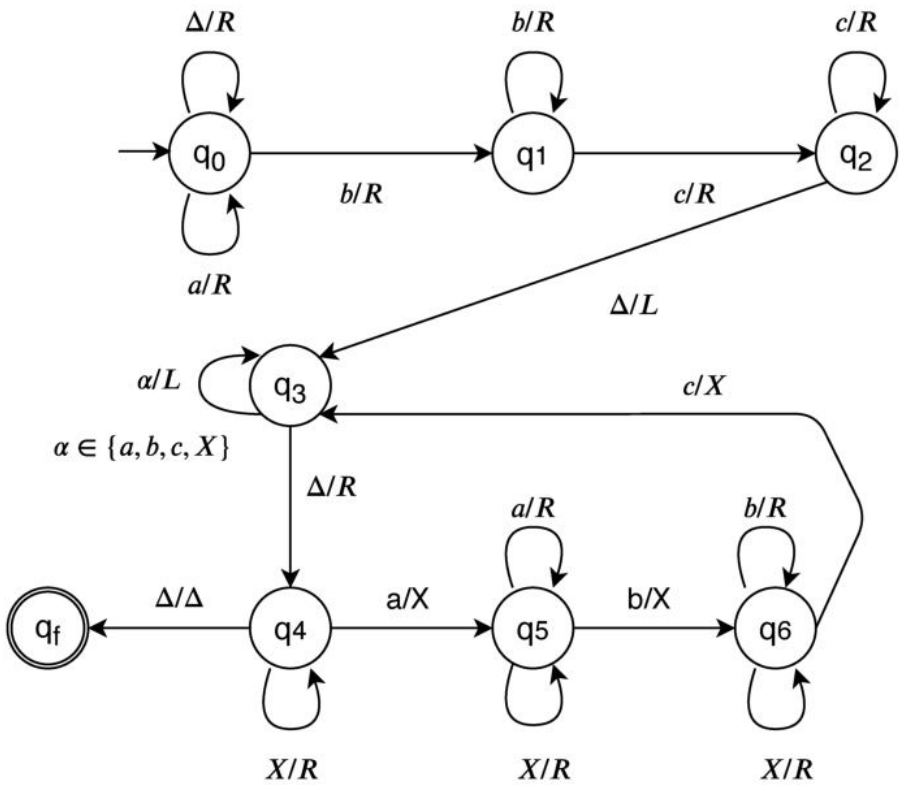
**Přechod** Přechod (krok výpočtu) TS $M$ definujeme jako nejmenší binární relaci $\vdash$ takovou, že $\forall q_1, q_2 \in Q\ \forall \gamma \in \Gamma^\omega\ \forall n \in \mathbb{N}\ \forall b \in \Gamma$:

- $(q_1, \gamma, n) \vdash (q_2, \gamma, n+1)$ pro $\delta(q_1, \gamma_n) = (q_2, R)$ – operace posunu hlavy doprava při $\gamma_n$ po hlavou;
- $(q_1, \gamma, n) \vdash (q_2, \gamma, n-1)$ pro $\delta(q_1, \gamma_n) = (q_2, L)$ – operace posunu hlavy doleva při $\gamma_n$ po hlavou;
- $(q_1, \gamma, n) \vdash (q_2, s_b^n(\gamma), n)$ pro $\delta(q_1, \gamma_n) = (q_2, b)$ – operace zápisu symbolu $b$ při $\gamma_n$ po hlavou.

Pro libovolný řetězec $\gamma \in \Gamma^\omega$ a číslo $n \in \mathbb{N}$ označme $\gamma_n$ n-tý symbol daného řetězce a označme $s_n^b(\gamma)$ řetězec, který vznikne z $\gamma$ záměnou $\gamma_n$ za $b$.

**Jazyk přijímaný**

- Řetězec $w \in \Sigma^*$ je přijat TS $M = (Q, \Sigma, \Gamma, \delta, q_0, g_f)$, jesliže pro $M$ platí: $(q_0, \Delta\gamma\Delta^\omega, 0) \vdash^* (q_f, \gamma', n)$ pro nějaké $\gamma, \gamma' \in \Gamma^*$ a $n \in \mathbb{N}$.
- Množinu $L(M) = \{w \mid w \text{ je přijat TS } M\} \subseteq \Sigma^*$ nazýváme jazyk přijímaný TS $M$.




Obrázek 23.1: Příklad TS, který přijímá jazyk $L = \{a^n b^n c^n \mid n > 0\}$.

**Univerzální turingův stroj** Je možné sestrojit takový turingův stroj, který má vlastnost, že dokáže simulovat chování jiného Turingova stroje. Může fungovat jako interpret. Na vstupu přijme zakódovaný Turingův stroj (program) a vstup, který vykoná.

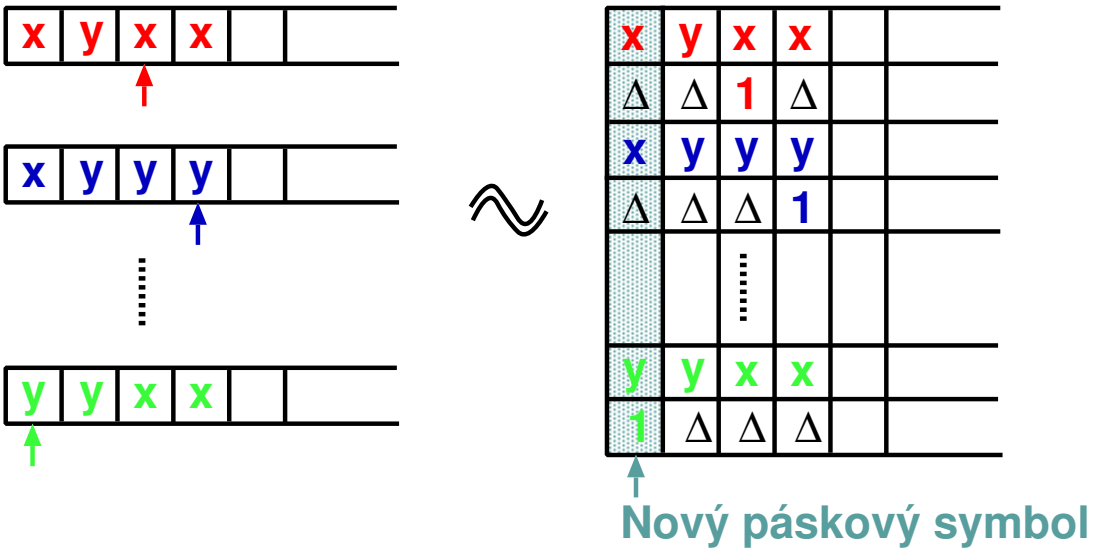
## 23.3 Varianty turingova stroje

### 23.3.1 Deterministický turingův stroj

Deterministický turingův stroj (DTS) je výchozí turingův stroj (DTS = TS).

### 23.3.2 Vícepáskový turingův stroj

Vícepáskový TS má stejnou (rozhodovací) vyjadřovací sílu jako jednopáskový TS (jsou mezi sebou převoditelné). Formálně to znamená, že pro každý $k$-páskový TS $M$ existuje jednopáskový TS $M'$ takový, že $L(M) = L(M')$. Myšlenka důkazu spočívá v tom, že $k$ pásek můžeme simulovat jednou. Jeden symbol na pásce potom bude mít formu $2k$-tice. Pro každou pásku jsou potřeba 2 prvky v n-tici, protože v jedné je třeba si udržovat pozici hlavy. Myšlenka je na obrázku.



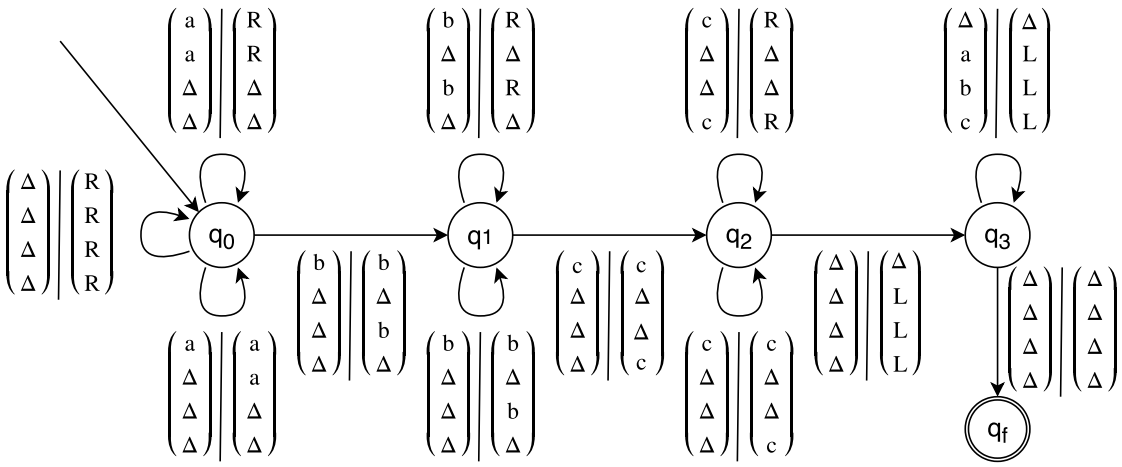

Obrázek 23.2: Myšlenka převodu vícepáskového TS na jednopáskový.

**Definice** Vícepáskový TS se od TS liší tvarem přechodové funkce, disponuje více páskami a s každou může pracovat zároveň. Formálně:

- $\delta$ je přechodová funkce (parciální funkce).

$$\delta : (Q - \{q_f\}) \times \Gamma_1 \times \ldots \times \Gamma_n \to Q \times (\Gamma_1 \cup \{L, R\}) \times \ldots \times (\Gamma_n \cup \{L, R\})$$

**Příklad** Příklad vícepáskového TS.




Obrázek 23.3: Příklad 4-páskový TS, který přijímá jazyk $L = \{a^n b^n c^n \mid n > 0\}$.

### 23.3.3 Nedeterministický turingův stroj

Nedeterministický TS (NTS) má stejnou (rozhodovací) vyjadřovací sílu jako DTS (jsou mezi sebou převoditelné). Formálně to znamená, že pro každý NTS $M$ existuje DTS $M'$ takový, že $L(M) = L(M')$. Myšlenka důkazu spočívá v tom, že strom běhů NTS prohledáváme do šířky (jelikož běh může obsahovat smyčky). Konkrétně, NTS $M$ budeme simulovat třípáskovým DTS. Význam jednotlivých pásek tohoto stroje je následující:

- Páska 1 obsahuje vstupní řetězec.
- Páska 2 je pracovní páska. Obsahuje kopii pásky 1 ohraničenou vhodnými speciálními značkami. Po neúspěšném pokusu o přijetí je její obsah smazán a obnoven z první pásky.
- Páska 3 obsahuje kódovanou volbu posloupností přechodů; při neúspěchu bude její obsah nahrazen jinou posloupností.

**Definice** NTS se od TS liší tvarem přechodové funkce. Formálně:

- $\delta$ je přechodová funkce (parciální funkce).

$$\delta : (Q - \{q_f\}) \times \Gamma \to 2^{Q \times (\Gamma \cup \{L,R\})}$$

**Příklad** Příklad nedeterministického TS, kde nedeterminismus může přinést značné zjednodušení, je TS, který přijímá jazyk $L = \{ww \mid w \in \Sigma^*\}$.

### 23.3.4 Úplný turingův stroj

Úplný turingův stroj je takový TS, který pro libovolný vstup zastaví, a buď přijme nebo odmítne (nemůže se zacyklit). Úplné turingovy stroje mají menší (rozhodovací) vyjadřovací sílu než obecný turingův stroj. Nedeterministický Turingův stroj je úplný, právě když pro každý vstup je každá výpočetní větev konečná (tj. pro každý vstup vždy zastaví).

- Jazyk $L \subseteq \Sigma^*$ se nazývá:
  - Rekurzivně vyčíslitelný $(REL, \mathcal{L}_0, \mathcal{L}_{RE})$, jestliže $L = L(M)$ pro nějaký TS $M$.
  - Rekurzivní $(RL, \mathcal{L}_{Rec})$, jestliže $L = L(M)$ pro nějaký úplný TS $M$.
- Je-li $M$ úplný TS, pak říkáme, že $M$ rozhoduje jazyk $L(M)$.
- Ke každému rekurzívnímu jazyku existuje TS, který ho rozhoduje, tj. zastaví pro každé vstupní slovo (např. na pásku zapíše $YES$ nebo $NO$).
- TS přijímající rekurzívně vyčíslitelný jazyk $L$ zastaví pro každé $w \in L$, ovšem pro $w \notin L$ může zastavit, ale také může donekonečna cyklit.
- Platí $2^{\Sigma^*} \subset \mathcal{L}_{RE} \subset \mathcal{L}_{Rec} \subset \mathcal{L}_1$.

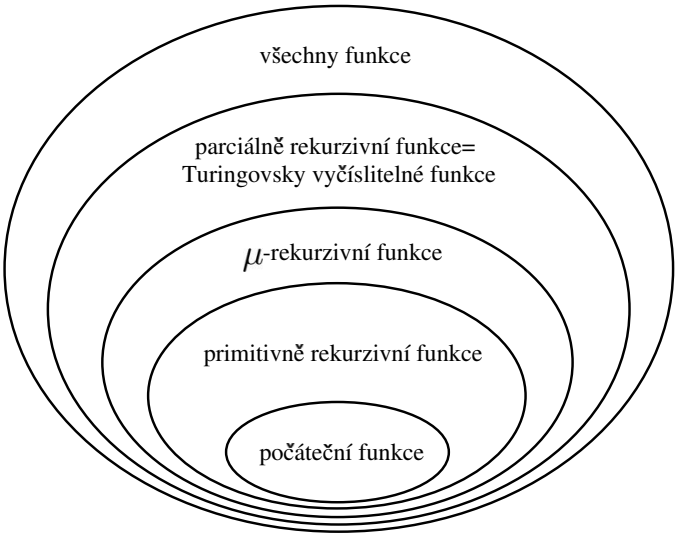
## 23.4 Lineárně omezený automat

- Lineárně omezený automat (LOA) je nedeterministický TS, který nikdy neopustí tu část pásky, na níž je zapsán jeho vstup.
  - Takový turingův stroj, který má lineárně omezenou velikost pásky vůči délce vstupu (*Linear Bounded Automata*). Avšak, díky možnosti zápisu n-tic na pásku (viz důkaz u vícepaskových TS), je možné pásku zkomprimovat tak, že stačí pouze tolik místa, kolik je délka vstupního slova.
- Deterministický LOA můžeme definovat jako DTS, který nikdy neopustí část pásky se zapsaným vstupem.
  - Není známo, zda deterministický LOA je či není striktně slabší než LOA.
- Třída jazyků, kterou lze generovat kontextovými gramatikami, odpovídá třídě jazyků, které lze přijímat LOA.

- Z konečné délky pásky vyplývá, že pro každý LOA existuje pouze konečné množství konfigurací. To znamená, že pokud LOA cyklí, jsme to schopni detekovat. A tedy, problémy pro LOA jsou rozhodnutelné.

## 23.5 Vyčíslitelné funkce

- Tzv. rekurzivní / vyčíslitelné funkce jsou $f : \mathbb{N}^n \to \mathbb{N}^m$, kde $n, m \in \mathbb{N}$.
- Klasifikace parciálních funkcí:
  - Totální funkce (např. funkce sčítání) – funkce je definovaná pro celý definiční obor, tj. pro každý vstup existuje výstup.
  - Striktně parciální funkce (např. funkce dělení) – funkce není definovaná pro celý definiční obor, tj. existuje vstup jemuž není definovaný výstup.
- Funkce totální, nemusí být vyčíslitelná (není možné je vypočítat počítačem, ačkoliv matematicky je lze zadefinovat).
- Konvence: $n$-tici $(x_1, x_2, \ldots, x_n) \in \mathbb{N}^n$ budeme označovat jako $\bar{x}$.




Obrázek 23.4: Hierarchie vyčíslitelných funkcí.

### 23.5.1 Počáteční funkce

Základní funkce, nacházejí se nejníže v hierarchii vyčíslitelných funkcí.

- Nulová funkce [$\xi$] (*zero function*)

$$\xi : () \to 0$$

$$\xi() = 0$$

- Funkce následníka [$\sigma$] (*successor function*)

$$\sigma : \mathbb{N} \to \mathbb{N}$$

$$\sigma(x) = x + 1$$

- Projekce [$\pi$] (*projection*) – Vybírá z $n$-tice $k$-tý prvek.

$$\pi_k^n : \mathbb{N}^n \to \mathbb{N}$$

$$\pi_k^n(\bar{x}) = x_k$$

### 23.5.2 Primitivně rekurzivní funkce

Jde o funkce vytvořené z počátečních funkcí.

- Kombinace [$\times$] – Kombinace dvou funkcí $f : \mathbb{N}^k \to \mathbb{N}^n$ a $g : \mathbb{N}^k \to \mathbb{N}^m$ vznikne funkce:

$$f \times g : \mathbb{N}^k \to \mathbb{N}^{n+m}$$

$$f \times g(\bar{x}) = (f(\bar{x}), g(\bar{x}))\ ,\ \bar{x} \in \mathbb{N}^k$$

- Kompozice [$\circ$] – Kompozice dvou funkcí $f : \mathbb{N}^k \to \mathbb{N}^n$ a $g : \mathbb{N}^n \to \mathbb{N}^m$ vznikne funkce:

$$g \circ f : \mathbb{N}^k \to \mathbb{N}^m$$

$$g \circ f(\bar{x}) = g(f(\bar{x}))\ ,\ \bar{x} \in \mathbb{N}^k$$

- Primitivní rekurze – Primitivní rekurze je technika, která umožňuje vytvořit funkci $f : \mathbb{N}^{k+1} \to \mathbb{N}^m$ na základě jiných dvou funkcí $g : \mathbb{N}^k \to \mathbb{N}^m$ a $h : \mathbb{N}^{k+m+1} \to \mathbb{N}^m$ rovnicemi:

$$f(\bar{x}, 0) = g(\bar{x})$$

$$f(\bar{x}, y+1) = h(\bar{x}, y, f(\bar{x}, y))\ ,\ \bar{x} \in \mathbb{N}^k$$

Následují příklady definic funkcí, které spadají do třídy primitivně rekurzivních funkcí.

**Konstantní funkce** Funkce $\kappa_m^n$ libovolné $n$-tici $x \in \mathbb{N}^n$ přiřadí konstantní hodnotu $m \in \mathbb{N}$.

$$\kappa_m^0 \equiv \sigma \circ \sigma \circ \ldots \circ \sigma \circ \xi$$

($\sigma$ je aplikována $m$-krát)

**Funkce sčítání**

$$plus : \mathbb{N}^2 \to \mathbb{N}$$

$$plus(x, 0) = \pi_1^1(x)$$

$$plus(x, y+1) = \sigma \circ plus(x, y)$$

**Funkce násobení**

$$mult : \mathbb{N}^2 \to \mathbb{N}$$

$$mult(x, 0) = \xi()$$

$$mult(x, y+1) = plus(x, mult(x, y))$$

**Funkce umocňování**

$$exp : \mathbb{N}^2 \to \mathbb{N}$$

$$exp(x, 0) = \sigma \circ \xi()$$

$$exp(x, y+1) = mult(x, exp(x, y))$$

**Funkce předchůdce**

$$pred : \mathbb{N} \to \mathbb{N}$$

$$pred(0) = \xi()$$

$$pred(x+1) = \pi_1^2(x, pred(x))$$

**Funkce monus** Funkce odčítání, ale nejmenší číslo je 0.

$$monus : \mathbb{N}^2 \to \mathbb{N}$$

$$monus(x, 0) = \pi_1^1(x)$$

$$monus(x, y+1) = pred \circ monus(x, y)$$

**Funkce negace** Logická negace, 0 je *false*, ostatní čísla jsou *true*.

$$neg : \mathbb{N} \to \mathbb{N}$$

$$neg(x) = monus(\sigma \circ \xi(), x)$$

**Funkce rovnost** Vrací 0 – *false* a 1 – *true*.

$$eq : \mathbb{N}^2 \to \mathbb{N}$$

$$eq \equiv monus \circ (\kappa_1^2 \times (plus \circ ((monus \circ (\pi_2^2 \times \pi_1^2)) \times monus \circ (\pi_1^2 \times \pi_2^2))))$$

**Funkce je větší** Vrací 0 – *false* a 1 – *true*.

$$isGreater : \mathbb{N}^2 \to \mathbb{N}$$

$$isGreater(x, y) = neg \circ eq(monus(x, y), \xi())$$

**Funkce odmocnina** Funkce $foo$ je pomocná.

$$foo : \mathbb{N}^2 \to \mathbb{N}$$

$$foo(0, y) = \xi()$$

$$foo(x+1, y) = plus(isGreater(mult(\sigma \times \sigma(x)), y), foo(x, y))$$

$$sqrt : \mathbb{N} \to \mathbb{N}$$

$$sqrt(x) = monus(x, foo(x, x))$$

### 23.5.3 $\mu$-rekurzivní funkce

- Třída totálních vyčíslitelných funkcí se nazývá $\mu$-rekurzivní funkce.

### 23.5.4 Parciálně rekurzivní funkce

- Zavádí koncept tzv. *minimalizace*.
- Umožňuje realizaci *while* cyklu.
- Parciálně rekurzivní funkce jsou ekvivalentní Turingovým strojům.
- **[[todo: dokončit]]**

### 23.5.5 Turingovsky vyčíslitelné funkce

- Parciální funkce, kterou může počítat nějaký turingův stroj se nazývá turingovsky vyčíslitelná funkce.
- Každá parciálně rekurzivní funkce je turingovsky vyčíslitelná.
- Každý výpočetní proces prováděný turingovým strojem je procesem vyčíslení nějaké parciálně rekurzivní funkce.

**Definice** Turingův stroj $M = (Q, \Sigma, \Gamma, \delta, q_0, g_f)$ vyčísluje (počítá) parciální funkci $f : (\Sigma^*)^m \to (\Sigma_1^*)^n$, $\Sigma_1 \subseteq \Gamma$, $\Delta \notin \Sigma_1$, jestliže pro každé $(w_1, w_2, \ldots, w_m) \in (\Sigma^*)^m$ a odpovídající počáteční konfiguraci $\Delta w_1 \Delta w_2 \Delta \ldots \Delta w_m \Delta^\omega$ TS $M$:

1. V případě, že $f(w_1, w_2, \ldots, w_m)$ je definována, pak M zastaví a páska obsahuje $\Delta v_1 \Delta v_2 \Delta \ldots \Delta v_n \Delta^\omega$, kde $(v_1, v_2, \ldots, v_n) = f(w_1, w_2, \ldots, w_m)$.
2. V případě, že $f(w_1, w_2, \ldots, w_m)$ není definována, $M$ cyklí nebo zastaví abnormálně.

# Kapitola 24

# TIN – Nerozhodnutelnost (problém zastavení TS, princip diagonalizace a redukce, Postův korespondenční problém).

## 24.1 Zdroje

- `tin_2021_merged.pdf`
- `TIN_2020-11-03.mp4`
- `TIN_2020-11-10.mp4`
- `TIN_2020-11-20_demo.mp4`

## 24.2 Rozhodovací problém

**Rozhodovací problém**

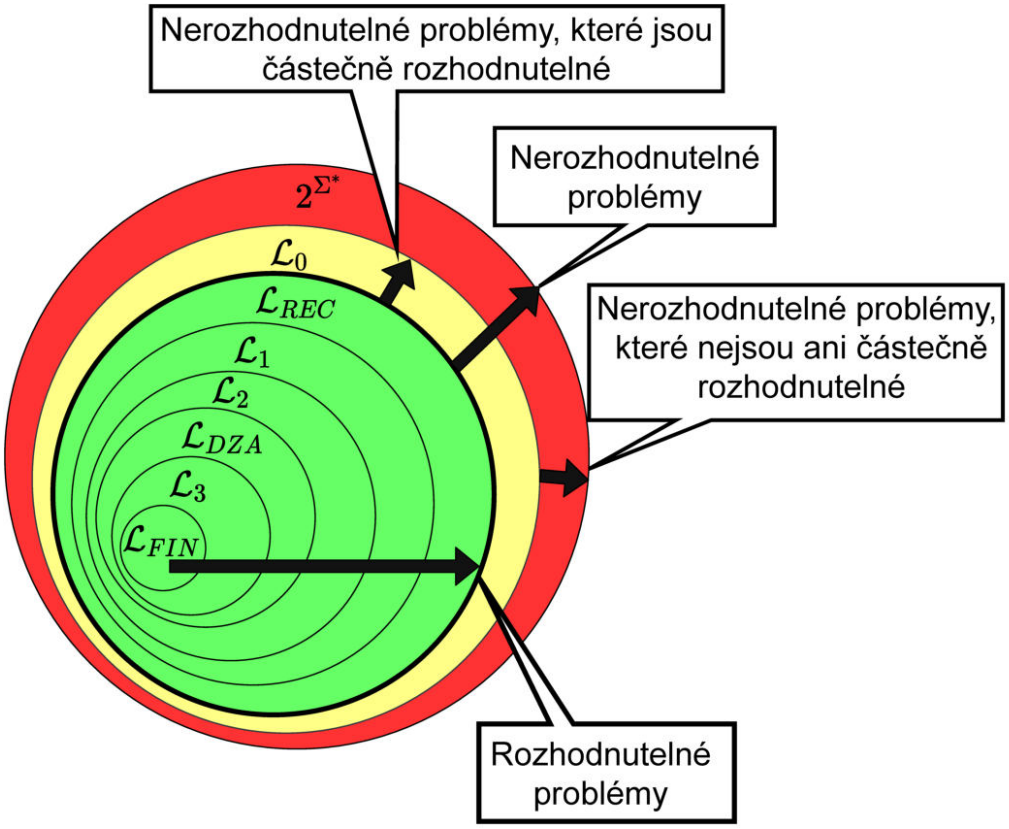
- Rozhodovací problém (*decision problem*) $P$ může být chápán jako funkce $f_P$ s oborem hodnot $\{true, false\}$.
- Rozhodovací problém je obvykle specifikován:
  - definičním oborem $A_P$ reprezentujícím množinu možných instancí problému,
  - podmnožinou $B_P \subseteq A_P$, $B_P = \{p \mid f_P(p) = true\}$ instancí, pro které je hodnota $f_P$ rovna $true$.

**Kódování problémů**

- V teorii formálních jazyků používáme ke kódování jednotlivých instancí problémů řetězce nad vhodnou abecedou $\Sigma$.
- Pak je rozhodovací problém $P$ přirozeně specifikován jazykem $L_P = \{w \in \Sigma^* \mid w = code(p),\ p \in B_P\}$, kde $code : A_P \to \Sigma^*$ je injektivní funkce, která přiřazuje instancím problému příslušný řetězec (nezávisle na $f_P$).

**Rozhodování problémů TS**

- Nechť $P$ je problém specifikovaný jazykem $L_P$ nad abecedou $\Sigma$. Problém P nazveme:
  - **Rozhodnutelný**, pokud $L_P$ je rekurzívní jazyk, tj. existuje TS, který $L_P$ rozhoduje (přijme každý řetězec $w \in L_P$, a zamítne každý řetězec $w \in \Sigma^* - L_P$).
  - **Nerozhodnutelný**, když není rozhodnutelný.
  - **Částečně rozhodnutelný**, jestliže LP je rekurzívně vyčíslitelný jazyk.
- Z toho plyne, že každý rozhodnutelný problém je současně částečně rozhodnutelný, ale některé nerozhodnutelné problémy nejsou ani částečně rozhodnutelné.
  - Problém je nerozhodnutelný, pokud pro něj neexistuje úplný turingův stroj.




Obrázek 24.1: Hierarchie jazyků/problémů a jejich (ne)rozhodnutelnost.

## 24.3 Známé problémy

### Problém zastavení TS

- Problém zastavení TS $M$ na daném vstupu $w$ (HP, *Halting Problem*).
- Neformálně: Znáte-li zdrojový kód programu (TS) a jeho vstup, rozhodněte, zda program zastaví, nebo zda poběží navždy bez zastavení.
- Problém není rozhodnutelný, ale je částečně rozhodnutelný.
  - $HP \in \mathcal{L}_{RE}$
  - Důkaz: s využitím Cantorovy diagonalizace.

$$HP = \{\langle M \rangle \# \langle w \rangle \mid \text{TS M na vstupu w zastaví}\}$$

### Problém nezastavení TS

- Problém nezastavení TS $M$ na daném vstupu $w$ (co-HP, *co-Halting Problem*).
- Neformálně: Znáte-li zdrojový kód programu (TS) a jeho vstup, rozhodněte, zda program nezastaví (poběží navždy bez zastavení), nebo zda zastaví.
- Problém není ani částečně rozhodnutelný.
  - $coHP \notin \mathcal{L}_{RE}$
  - Důkaz: vyplývá z definice komplementu.

$$coHP = \{\langle M \rangle \# \langle w \rangle \mid \text{TS M na vstupu w nezastaví}\}$$

### Problém náležitosti

- Problém náležitosti řetězce $w$ do jazyka $L$, $L \in \mathcal{L}_0$ (MP, *Membership Problem*).
- Neformálně: Znáte-li TS a jeho vstup, rozhodněte, zda TS daný vstup přijme (zastaví a akceptuje) a nebo nepřijme (zastaví a neakceptuje, zastaví abnormálně, nezastaví).
  - Jde o podobný případ jako problém zastavení TS, pouze obsahuje krok navíc.
- Problém není rozhodnutelný, ale je částečně rozhodnutelný.
  - $MP \in \mathcal{L}_{RE}$
  - Důkaz: s využitím redukce z HP.

$$MP = \{\langle M \rangle \# \langle w \rangle \mid w \in L(M)\}$$

### Problém nenáležitosti

- Problém nenáležitosti řetězce $w$ do jazyka $L$, $L \in \mathcal{L}_0$ (coMP, *Non-Membership Problem*).
- Neformálně: Znáte-li TS a jeho vstup, rozhodněte, zda TS daný vstup nepřijme (zastaví a neakceptuje, zastaví abnormálně, nezastaví) a nebo přijme (zastaví a akceptuje).
- Problém není ani částečně rozhodnutelný.
  - $coMP \notin \mathcal{L}_{RE}$
  - Důkaz: s využitím redukce z coHP.

$$coMP = \{\langle M \rangle \# \langle w \rangle \mid w \notin L(M)\}$$

### Postův korespondenční problém

- Postův systém $S$ nad abecedou $\Sigma$ je dán neprázdným seznamem dvojic neprázdných řetězců nad $\Sigma$, formálně:

$$S = \langle (\alpha_1, \beta_1), \ldots, (\alpha_k, \beta_k) \rangle, \text{ pro } k \geq 1, \forall i : 1 \leq i \leq k : \alpha_i, \beta_i \in \Sigma^+$$

- Řešením Postova systému $S$ je každá neprázdná posloupnost přirozených čísel $I = \langle i_1, i_2, \ldots, i_m \rangle$, kde $m \geq 1$ a $1 \leq j \leq m : 1 \leq i_j \leq k$, tak, že:

$$\alpha_{i_1}, \alpha_{i_2}, \ldots, \alpha_{i_m} = \beta_{i_1}, \beta_{i_2}, \ldots, \beta_{i_m}$$

  - $m$ není omezené a indexy se mohou opakovat.
- Postův korespondenční problém (PCP) zní: existuje pro daný Postův systém řešení?
- Problém není ani částečně rozhodnutelný.
  - $PCP \notin \mathcal{L}_{RE}$
  - Důkaz: *nezkouší se*
- Příklad: PS $S$ nad $\Sigma = \{a, b\}$:
  - $S = \langle (ab, a), (aa, baaab), (aa, a) \rangle$
  - $S$ má řešení, např. $I = \langle 1, 2, 1, 3 \rangle$
    - $ab.aa.ab.aa = a.baaab.a.a$

**Další problémy**

- $\{\langle M \rangle \mid$ TS M má aspoň 2020 stavů $\} \in \mathcal{L}_{Rec}$
- $\{\langle M \rangle \mid$ TS M učiní více než 2020 kroků na vstupu $\epsilon\} \in \mathcal{L}_{Rec}$
- $\{\langle M \rangle \mid$ TS M učiní více než 2020 kroků na nějakém vstupu $\} \in \mathcal{L}_{Rec}$
- $\{\langle M \rangle \mid L(M) \neq \emptyset\} \in \mathcal{L}_{RE}$
- $\{\langle M \rangle \mid |L(M)| \geq 42\} \in \mathcal{L}_{RE}$
- $\{\langle M \rangle \mid L(M) = \emptyset\} \notin \mathcal{L}_{RE}$
- $\{\langle M \rangle \mid |L(M)| \leq 42\} \notin \mathcal{L}_{RE}$
- $\{\langle M \rangle \mid L(M) \in \mathcal{L}_{Rec}\} \notin \mathcal{L}_{RE}$

## 24.4 Diagonalizace

- Diagonalizace (Cantorova diagonální metoda) je důkázová technika sporem, která slouží k porovnání mohutnosti dvou množin (zejména dvou nekonečných množin).
- Spočetné nekonečno:
  - Dokážeme „spočítat“ a uspořádat.
  - Existuje bijekce mezi množinou $\mathbb{N}$ a každou spočetně nekonečnou množinou.
- Nespočetné nekonečno:
  - Nedokážeme „spočítat“ a uspořádat.
  - Neexistuje bijekce mezi množinou $\mathbb{N}$ a žádnou nespočetně nekonečnou množinou.

**Příklad 1: dokažte, že $\mathbb{R}$ je větší než $\mathbb{N}$**

- Zřejmě platí, že pokud podmnožiny množiny je nespočetná, tak i množina je nespočetná.
  - Z toho vyplývá, že stačí dokázat že libovolná podmnožina $M \subseteq \mathbb{R}$ je nespočetná.

- Vybereme množinu $M = (0, 1)$.

- Předpokládejme, že $M$ je spočetná množina, pak existuje bijekce $f : \mathbb{N} \leftrightarrow M$.
- Uspořádejme $M$ do libovolné posloupnosti a zobrazme $f$ jako nekonečnou matici.

| | 1. desetinná pozice | 2. desetinná pozice | 3. desetinná pozice | ... |
|---|---|---|---|---|
| $f(0)$ | $d_{11}$ | $d_{12}$ | $d_{13}$ | ... |
| $f(1)$ | $d_{21}$ | $d_{22}$ | $d_{23}$ | ... |
| $f(2)$ | $d_{31}$ | $d_{32}$ | $d_{33}$ | ... |
| ... | ... | ... | ... | ... |

Kde $\forall i, j : d_{ij} \in \{0, 1, \ldots, 9\}$ a $\forall n \in \mathbb{N} : f(n) = 0, d_{n1}d_{n2}\ldots$

- Uvažme číslo $x = 0, e_1e_2e_3\ldots$ takové, že $e_i = 9 - d_i i$ pro $i = 1, 2, 3, \ldots$.
- Zřejmě platí, že $x \in M$ a tedy, musí existovat $n \in \mathbb{N} : f(n) = x$.
- Ale současně $\forall n \in \mathbb{N} : f(n) \neq x$ protože dojde k neshodě minimálně na diagonále.
  - Spor.

**Příklad 2: dokažte, že $2^{\Sigma^*}$ je větší než $\mathcal{L}_0$**

**Věta 9.1** Pro každou abecedu $\Sigma$ existuje jazyk nad $\Sigma$, který není typu 0 (tj. rekurzívně vyčíslitelný).

*Důkaz.*

1. Libovolný jazyk typu 0 nad $\Sigma$ může být přijat TS s $\Gamma = \Sigma \cup \{\Delta\}$: Pokud $M$ používá více symbolů, můžeme je zakódovat jako jisté posloupnosti symbolů ze $\Sigma \cup \{\Delta\}$ a sestrojit TS $M'$, který simuluje $M$.
2. Nyní můžeme snadno systematicky vypisovat všechny TS s $\Gamma = \Sigma \cup \{\Delta\}$. Začneme stroji se dvěma stavy, pak se třemi stavy, ...
   Závěr: Množina všech takových strojů a tedy i jazyků typu 0 je spočetná.
3. Množina $\Sigma^*$ ale obsahuje nekonečně mnoho řetězců a proto je množina $2^{\Sigma^*}$ zahrnující všechny jazyky nespočetná – důkaz viz další strana.
4. Z rozdílnosti mohutností spočetných a nespočetných množin plyne platnost uvedené věty.

□

**Lemma 9.1** Pro neprázdné, konečné $\Sigma$ je množina $2^{\Sigma^*}$ nespočetná.

*Důkaz.* Důkaz provedeme tzv. diagonalizací (poprvé použitou Cantorem při důkazu rozdílné mohutnosti $\mathbb{N}$ a $\mathbb{R}$).

- Předpokládejme, že $2^{\Sigma^*}$ je spočetná. Pak dle definice spočetnosti existuje bijekce $f : \mathbb{N} \longleftrightarrow 2^{\Sigma^*}$.
- Uspořádejme $\Sigma^*$ do nějaké posloupnosti $w_1, w_2, w_3, \ldots$, např. $\varepsilon$, $x$, $y$, $xx$, $xy$, $yx$, $yy$, $xxx$, ... pro $\Sigma = \{x, y\}$. Nyní můžeme $f$ zobrazit nekonečnou maticí:

$$\begin{array}{lcccccc} & w_0 & w_1 & w_2 & \ldots & w_i & \ldots \\ L_0 = f(0) & a_{00} & a_{01} & a_{02} & \ldots & a_{0i} & \ldots \\ L_1 = f(1) & a_{10} & a_{11} & a_{12} & \ldots & a_{1i} & \ldots \\ L_2 = f(2) & a_{20} & a_{21} & a_{22} & \ldots & a_{2i} & \ldots \\ \ldots & & & & & & \end{array}, \text{ kde } a_{ij} = \begin{cases} 0, \text{ jestliže } w_j \notin L_i, \\ 1, \text{ jestliže } w_j \in L_i. \end{cases}$$

- Uvažujme jazyk $\overline{L} = \{w_i \mid a_{ii} = 0\}$. $\overline{L}$ se liší od každého jazyka $L_i = f(i)$, $i \in \mathbb{N}$:
  - je-li $a_{ii} = 0$, pak $w_i$ patří do jazyka,
  - je-li $a_{ii} = 1$, pak $w_i$ nepatří do jazyka.
- Současně ale $\overline{L} \in 2^{\Sigma^*}$, $f$ tudíž není surjektivní, což je spor.

□

### Příklad 3: dokažte, že HP není rozhodnutelný, ale je částečně rozhodnutelný

**Věta 9.2** **Problém zastavení TS (Halting Problem)**, kdy nás zajímá, zda daný TS $M$ pro danou vstupní větu $w$ zastaví, **není rozhodnutelný**, ale je **částečně rozhodnutelný**.

*Důkaz.*

- Problému zastavení odpovídá rozhodování jazyka $HP = \{\langle M \rangle \# \langle w \rangle \mid M \text{ zastaví při } w\}$, kde $\langle M \rangle$ je kód TS $M$ a $\langle w \rangle$ je kód $w$.
- Částečnou rozhodnutelnost ukážeme snadno použitím modifikovaného univerzálního TS $T_U$, který zastaví přijetím vstupu $\langle M \rangle \# \langle w \rangle$ právě tehdy, když $M$ zastaví při $w$ – modifikace spočívá v převedení abnormálního zastavení při simulaci na zastavení přechodem do $q_F$.
- Nerozhodnutelnost ukážeme pomocí diagonalizace:
  1. Pro $x \in \{0, 1\}^*$, nechť $M_x$ je TS s kódem $x$, je-li $x$ legální kód TS. Jinak ztotožníme $M_x$ s pevně zvoleným TS, např. TS, který pro libovolný vstup okamžitě zastaví.
  2. Můžeme nyní sestavit posloupnost $M_\varepsilon, M_0, M_1, M_{00}, M_{01}, M_{10}, M_{11}, M_{000}, \ldots$ zahrnující všechny TS nad $\Sigma = \{0, 1\}$ indexované řetězci z $\{0, 1\}^*$.

3. Uvažme nekonečnou matici

| | $\varepsilon$ | $0$ | $1$ | $00$ | $01$ | $10$ | ... |
|---|---|---|---|---|---|---|---|
| $M_\varepsilon$ | $H_{M_\varepsilon,\varepsilon}$ | $H_{M_\varepsilon,0}$ | $H_{M_\varepsilon,1}$ | $H_{M_\varepsilon,00}$ | $H_{M_\varepsilon,01}$ | ... | |
| $M_0$ | $H_{M_0,\varepsilon}$ | $H_{M_0,0}$ | $H_{M_0,1}$ | $H_{M_0,00}$ | $H_{M_0,01}$ | ... | |
| $M_1$ | $H_{M_1,\varepsilon}$ | $H_{M_1,0}$ | $H_{M_1,1}$ | $H_{M_1,00}$ | $H_{M_1,01}$ | ... | |
| $M_{00}$ | $H_{M_{00},\varepsilon}$ | $H_{M_{00},0}$ | $H_{M_{00},1}$ | $H_{M_{00},00}$ | $H_{M_{00},01}$ | ... | |
| $M_{01}$ | $H_{M_{01},\varepsilon}$ | $H_{M_{01},0}$ | $H_{M_{01},1}$ | $H_{M_{01},00}$ | $H_{M_{01},01}$ | ... | |
| ... | | | | | | | |

kde $H_{M_x,y} = \begin{cases} \textbf{C}, \text{ jestliže } M_x \text{ cyklí na } y, \\ \textbf{Z}, \text{ jestliže } M_x \text{ zastaví na } y. \end{cases}$

4. Předpokládejme, že existuje *úplný* TS $K$ přijímající jazyk $HP$, tj. $K$ pro vstup $\langle M\rangle\#\langle w\rangle$
   - zastaví normálně (přijme) právě tehdy, když $M$ zastaví na $w$,
   - zastaví abnormálně (odmítne) právě tehdy, když $M$ cyklí na $w$.
5. Sestavíme TS $N$, který pro vstup $x \in \{0,1\}^*$:
   - Sestaví $M_x$ z $x$ a zapíše $\langle M_x\rangle\#x$ na svou pásku.
   - Simuluje $K$ na $\langle M_x\rangle\#x$, přijme, pokud $K$ odmítne, a přejde do nekonečného cyklu, pokud $K$ přijme.

Všimněme si, že $N$ v podstatě komplementuje diagonálu uvedené matice:

| | $\varepsilon$ | $0$ | $1$ | $00$ | $01$ | $10$ | ... |
|---|---|---|---|---|---|---|---|
| $M_\varepsilon$ | $H_{M_\varepsilon,\varepsilon}$ | $H_{M_\varepsilon,0}$ | $H_{M_\varepsilon,1}$ | $H_{M_\varepsilon,00}$ | $H_{M_\varepsilon,01}$ | ... | |
| $M_0$ | $H_{M_0,\varepsilon}$ | $H_{M_0,0}$ | $H_{M_0,1}$ | $H_{M_0,00}$ | $H_{M_0,01}$ | ... | |
| $M_1$ | $H_{M_1,\varepsilon}$ | $H_{M_1,0}$ | $H_{M_1,1}$ | $H_{M_1,00}$ | $H_{M_1,01}$ | ... | |
| $M_{00}$ | $H_{M_{00},\varepsilon}$ | $H_{M_{00},0}$ | $H_{M_{00},1}$ | $H_{M_{00},00}$ | $H_{M_{00},01}$ | ... | |
| $M_{01}$ | $H_{M_{01},\varepsilon}$ | $H_{M_{01},0}$ | $H_{M_{01},1}$ | $H_{M_{01},00}$ | $H_{M_{01},01}$ | ... | |
| ... | | | | | | | |

6. Dostáváme, že

$$\begin{aligned} N \text{ zastaví na } x \quad &\Leftrightarrow \quad K \text{ odmítne } \langle M_x\rangle\#\langle x\rangle && \text{(definice } N\text{)} \\ &\Leftrightarrow \quad M_x \text{ cyklí na } x && \text{(předpoklad o } K\text{).} \end{aligned}$$

7. To ale znamená, že $N$ se liší od každého $M_x$ alespoň na jednom řetězci – konkrétně $x$. Což je ovšem spor s tím, že posloupnost $M_\varepsilon, M_0, M_1, M_{00}, M_{01}, M_{10}, M_{11}, M_{000}, \ldots$ zahrnuje všechny TS nad $\Sigma = \{0,1\}$. Tento spor plyne z předpokladu, že existuje TS $K$, který pro daný TS $M$ a daný vstup $x$ určí (rozhodne), zda $M$ zastaví na $x$, či nikoliv.

$\square$

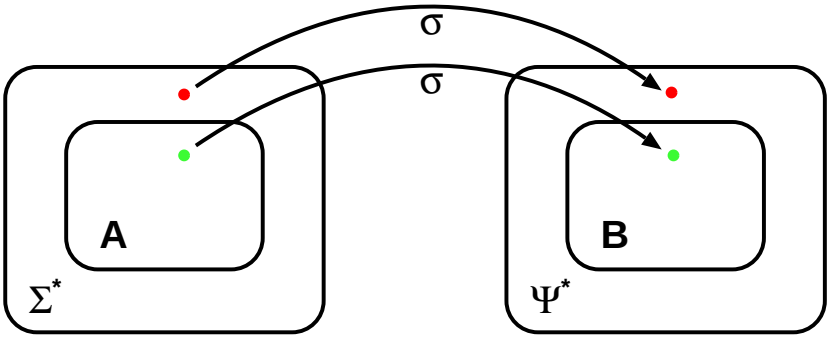
❖ Ukázali jsme, že problém zastavení TS je částečně rozhodnutelný a tedy jazyk $HP$ rekurzívně vyčíslitelný. Z věty 8.6 pak plyne, že **komplement problému zastavení** není ani částečně rozhodnutelný a jazyk co-$HP = \{\langle M\rangle\#\langle w\rangle \mid M$ nezastaví při $w\}$ je příkladem jazyka, jenž není ani rekurzívně vyčíslitelný.

## 24.5 Redukce

- Redukce je důkazová technika, která slouží k dokázání, že nějaký problém není rozhodnutelný (částečně rozhodnutelný) – neboli, že určitý jazyk není rekurzívní (rekurzívně vyčíslitelný).
  - Víme, že jazyk $A$ není rekurzívní (rekurzívně vyčíslitelný) a zkoumáme jazyk $B$. Ukážeme, že $A$ lze úplným TS převést (redukovat) na $B$, to ale znamená, že $B$ rovněž není rekurzivní (rekurzívně vyčíslitelný) – jinak by šlo použít úplný TS (ne-úplný TS) přijímající $B$ a příslušné redukce k sestavení úplného TS (ne-úplného TS) přijímajícího $A$, což by byl spor.
- Redukce vytváří uspořádání na problémech (nějaký problém je aspoň tak těžký jako jiný problém).

**Definice** Mějme jazyky $L_1 \subseteq \Sigma_1^*$ a $L_2 \subseteq \Sigma_2^*$. Redukce jazyka $L_1$ na jazyk $L_2$ je funkce $\sigma : \Sigma_1^* \to \Sigma_2^*$ taková, že

- $\sigma$ je implementovatelná úplným TS (totální rekurzivně vyčíslitelná funkce),
- $\forall w \in \Sigma_1^* \ : \ w \in L_1 \Leftrightarrow \sigma(w) \in L_2$ (zachovává členství v jazyce).

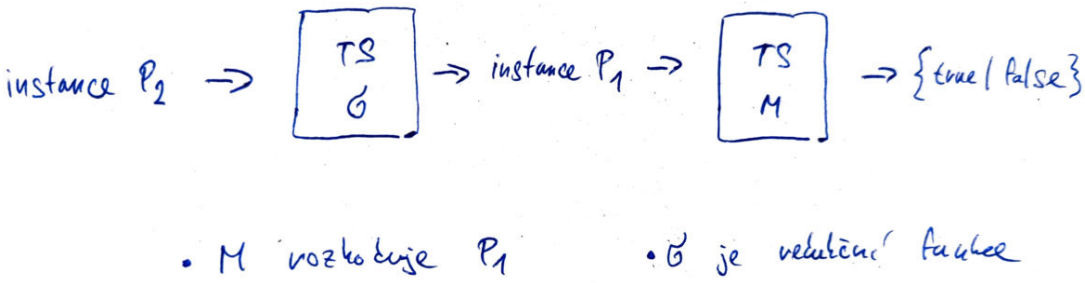



Redukce $\sigma : \Sigma^* \to \Psi^*$.

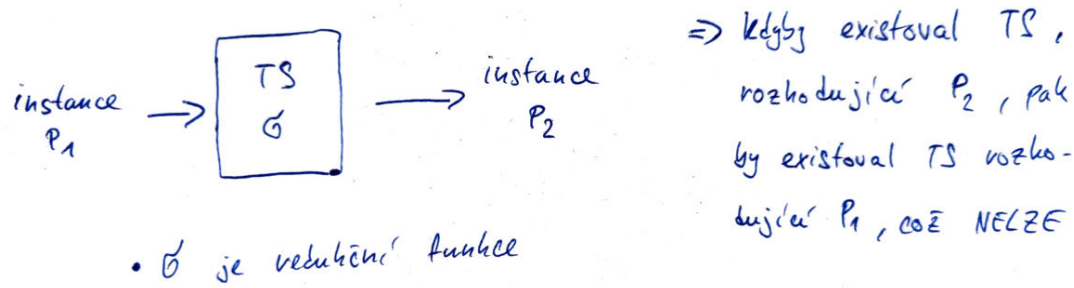
**Použití**

- 1. směr:
  - Máme známý problém $P_1$, o kterém víme, že je (částečně) rozhodnutelný. To znamená, že existuje TS, který ho (částečně) rozhoduje.
  - Máme nový problém $P_2$, o kterém chceme dokázat, že je (částečně) rozhodnutelný. Redukujeme $P_2$ na $P_1$.
  - Cílem je najít takovou funkci $\sigma$ (překladač), která namapuje instance problému $P_2$ na instance problému $P_1$, takovým způsobem, že není porušena definice redukce.

– Z toho vyplývá, že je-li $P_1$ (částečně) rozhodnutelný, tak i $P_2$ je (částečně) rozhodnutelný.




- 2. směr:
  - Máme známý problém $P_1$, o kterém víme, že je nerozhodnutelný (ani částečně).
  - Máme nový problém $P_2$, o kterém chceme dokázat, že je nerozhodnutelný (ani částečně).
  - Cílem je najít takovou funkci $\sigma$ (překladač), která namapuje instance problému $P_1$ na instance problému $P_2$, takovým způsobem, že není poručena definice redukce.
  - Z toho vyplývá, že je-li $P_1$ nerozhodnutelný (ani částečně), tak i $P_2$ je nerozhodnutelný (ani částečně).




**Obecný postup důkazu redukcí**

1. Navrhnout redukci.
2. Ukázat, že redukce lze implementovat úplným TS.
3. Ukázat, zachování členství v jazyce.

### Příklad 1: dokažte, že problém neprázdnosti jazyka TS je nerozhodnutelný

- Dokážeme redukcí z problému zastavení TS (HP).
- Jazyk charakterizující problém zastavení (obsahuje takové instance, pro které TS $M$ na $w$ zastaví):

$$HP = \{\langle M\rangle\#\langle w\rangle \mid \text{ TS } M \text{ na vstupu } w \text{ zastaví } \}$$

- Jazyk charakterizující problém neprázdnosti (obsahuje takové instance, pro které jazyk TS $M$ není prázdný):

$$NEMP = \{\langle M\rangle \mid \text{M je TS takový, že } L(M) \neq \emptyset\}$$

- Navrhneme redukci (redukující HP na NEMP, viz směr 2, „NEMP je aspoň tak těžký jako HP“):

$$HP \leq NEMP$$

$$\sigma : \{0,1,\#\}^* \to \{0,1\}^*$$

- Funkce $\sigma$ přiřadí řetězci $x \in \{0,1,\#\}^*$ řetězec $\langle M_\sigma\rangle$, kde $M_\sigma$ je TS implementující $\sigma$, pracující následovně:
  - $M_\sigma$ smaže svůj vstup (pozn. jazyk $L(M_\sigma)$ tedy bude buď $\emptyset$ a nebo $\Sigma^*$).
  - $M_\sigma$ zapíše na svůj vstup řetězez $x$.
  - $M_\sigma$ ověří, zda $x = \{\langle M\rangle\#\langle w\rangle\}$ má validní strukturu pro TS $M$ a jeho vstup $w$. Pokud nemá, tak odmítne.
  - TS $M_\sigma$ odsimiluje běh TS $M$ na $w$. Pokud $M$ zastaví, tak přijme, jinak cyklí.
- Funkce $\sigma$ lze evidentně realizovat úplným TS $M_\sigma$. Skládá se, ze 4 komponent:
  1. Smazání vstupu – konstantní operace, pro kterou $M_\sigma$ pouze vypíše daný kód.
  2. Zapsání $x$ na vstup – snadná operace, $M_\sigma$ vypíše kód, který provede přesun doprava a zápis $a_i$ pro $x = a_1a_2\ldots a_n$ pro $i = 1,2,\ldots,n$.
  3. Ověření správného strukturování vstupu – konstantní operace, pro kterou $M_\sigma$ pouze vypíše daný kód.
  4. Předání řízení univerzálnímu TS – konstantní operace, pro kterou $M_\sigma$ pouze vypíše daný kód.
- Ukázání zachování členství v jazyce.
  - Studujme jazyk $M_\sigma$:
    * $L(M_\sigma) = \emptyset \Leftrightarrow x$ nemá validní strukturu $\langle M\rangle\#\langle w\rangle$ a nebo $M$ na $w$ nezastaví.
    * $L(M_\sigma) = \Sigma^* \Leftrightarrow x$ má validní strukturu $\langle M\rangle\#\langle w\rangle$ a $M$ na $w$ zastaví.
  - Zachování členství:

$$\forall x \in \{0,1,\#\}^* : \sigma(x) = \langle M_\sigma\rangle \in NEMP \Leftrightarrow$$

$$\Leftrightarrow L(M_\sigma) = \Sigma^* \Leftrightarrow$$

$$\Leftrightarrow x = \langle M\rangle\#\langle w\rangle \text{ a } M \text{ na } w \text{ zastaví } \Leftrightarrow$$

$$\Leftrightarrow x \in HP$$

**Příklad 2: dokažte, že problém prázdnosti jazyka TS není ani částečně rozhodnutelný**

- Dokážeme redukcí z problému nezastavení TS (coHP).
- Jazyk charakterizující problém nezastavení (obsahuje takové instance, pro které TS $M$ na $w$ nezastaví):

$$coHP = \{\langle M\rangle\#\langle w\rangle \mid \text{ TS } M \text{ na vstupu } w \text{ nezastaví }\}$$

- Jazyk charakterizující problém prázdnosti (obsahuje takové instance, pro které jazyk TS $M$ je prázdný):

$$EMP = \{\langle M \rangle \mid \text{M je TS takový, že } L(M) = \emptyset\}$$

- Navrhneme redukci (redukující coHP na EMP, viz směr 2, „EMP je aspoň tak těžký jako coHP“):

$$coHP \leq EMP$$

$$\sigma : \{0, 1, \#\}^* \to \{0, 1\}^*$$

- Funkce $\sigma$ přiřadí řetězci $x \in \{0, 1, \#\}^*$ řetězec $\langle M_\sigma \rangle$, kde $M_\sigma$ je TS implementující $\sigma$, pracující následovně:
  - $M_\sigma$ smaže svůj vstup (pozn. jazyk $L(M_\sigma)$ tedy bude buď $\emptyset$ a nebo $\Sigma^*$).
  - $M_\sigma$ zapíše na svůj vstup řetězez $x$.
  - $M_\sigma$ ověří, zda $x = \{\langle M \rangle \# \langle w \rangle\}$ má validní strukturu pro TS $M$ a jeho vstup $w$. Pokud nemá, tak přijme.
  - TS $M_\sigma$ odsimiluje běh TS $M$ na $w$. Pokud $M$ zastaví, tak přijme, jinak cyklí.
- *Zbytek na stejný princip jako v příkladu 1.*

**Příklad 3: dokažte, že problém náležitosti řetězce jazyku $L(M)$, kde M je TS, je nerozhodnutelný**

- Dokážeme redukcí z problému zastavení TS (HP).
- Jazyk charakterizující problém zastavení (obsahuje takové instance, pro které TS $M$ na $w$ zastaví):

$$HP = \{\langle M \rangle \# \langle w \rangle \mid \text{ TS } M \text{ na vstupu } w \text{ zastaví } \}$$

- Jazyk charakterizující problém náležitosti řetězce jazyku $L(M)$, kde M je TS (obsahuje takové instance, pro které jazyk TS $M$ na $w$ zastaví a přijme):

$$MP = \{\langle M \rangle \# \langle w \rangle \mid \text{M je TS takový, že } w \in L(M)\}$$

- Navrhneme redukci (redukující HP na MP, viz směr 2, „MP je aspoň tak těžký jako HP“):

$$HP \leq MP$$

$$\sigma : \{0, 1, \#\}^* \to \{0, 1, \#\}^*$$

$$\sigma(\langle M \rangle \# \langle w \rangle) = \langle M' \rangle \# \langle w' \rangle$$

- Funkce $\sigma$ přiřadí řetězci $x \in \{0, 1, \#\}^*$ řetězec $\langle M \rangle \# \langle w \rangle$, kde $M_\sigma$ je TS implementující $\sigma$, pracující následovně:
  - $M_\sigma$ smaže svůj vstup (pozn. jazyk $L(M_\sigma)$ tedy bude buď $\emptyset$ a nebo $\Sigma^*$).
  - $M_\sigma$ zapíše na svůj vstup řetězez $x$.
  - $M_\sigma$ ověří, zda $x = \{\langle M \rangle \# \langle w \rangle\}$ má validní strukturu pro TS $M$ a jeho vstup $w$. Pokud nemá, tak **[[todo: přijme?]]**.
  - TS $M_\sigma$ odsimiluje běh TS $M$ na $w$. Pokud $M$ na $w$ zastaví přijme, tak $M_\sigma$ přijme. Jinak, $M_\sigma$ cyklí a nebo zamítá.
- *Zbytek na stejný princip jako v příkladu 1.*

**Příklad 4: dokažte, že problém víceznačnosti bezkontextových gramatik, není ani částečně rozhodnutelný**

**[[note: Příklad je na redukci z PCP, jedná se o těžší redukci, která se nebude zkoušet.]]**

# Kapitola 25

# TIN – Časová a paměťová složitost (třídy složitosti, úplnost, SAT problém).

## 25.1 Zdroje

- `tin_2021_merged.pdf`
- `TIN_2020-12-01.mp4`
- `TIN_2020-12-04_demo.mp4`
- `TIN_2020-12-08.mp4`
- `TIN_2020-12-11_demo.mp4`

## 25.2 Úvod a kontext

- Časová složitost – počet kroků (přechodů) TS provedený od počátku do konce výpočtu.
- Prostorová (paměťová) složitost – počet buněk pásky TS požadovaný pro daný výpočet.
- Je-li časová složitost výpočtu prováděného TS rovna $n$, pak prostorová složitost tohoto výpočtu není větší než $n + 1$.
  - Tvrzení je jednoduchou implikací plynoucí z definice časové a prostorové složitosti.
- Při popisu složitosti algoritmů (výpočtů TS), chceme často vyloučit vliv aditivních a multiplikativních konstant:
  - Různé aditivní a multiplikativní konstanty vzniknou velmi snadno „drobnými“ úpravami uvažovaných algoritmů.
  - Primárně nás zajímá, jak složitost roste v závislosti na délce vstupu. Zejména pro dlouhé vstupy.
- Různé případy při analýze složitosti:
  - analýza složitosti nejhoršího případu,
  - analýza složitosti nejlepšího případu,

– analýza složitosti průměrného případu,

– amortizovaná analýza – Studuje posloupnost operací jako celek. Tato technika umožňuje, na rozdíl od klasického přístupu mnohem přesnější určení časové složitosti algoritmu.

## 25.3 Asymptotická složitost

- Složitost algoritmů, resp. turingových strojů.
- Asymptotická složitost – neřešíme aditivní a multiplikativní konstanty a bereme v potaz pouze nejvyšší polynom.
- Nechť $\mathcal{F}$ je množina funkcí $f : \mathbb{N} \to \mathbb{N}$. Pro danou funkci $f \in \mathcal{F}$ definujeme množiny funkcí $\mathcal{O}(f(n))$, $\Omega(f(n))$ a $\Theta(f(n))$ následovně:
  - **Asymptotické horní omezení** funkce $f(n)$ je množina

$$\mathcal{O}(f(n)) = \{g(n) \in \mathcal{F} \mid \exists c \in \mathbb{R}^+ \; \exists n_0 \in \mathbb{N} \; \forall n \in \mathbb{N} : n \geq n_0 \Rightarrow$$

$$\Rightarrow 0 \leq g(n) \leq c \cdot f(n)\}$$

  - **Asymptotické dolní omezení** funkce $f(n)$ je množina

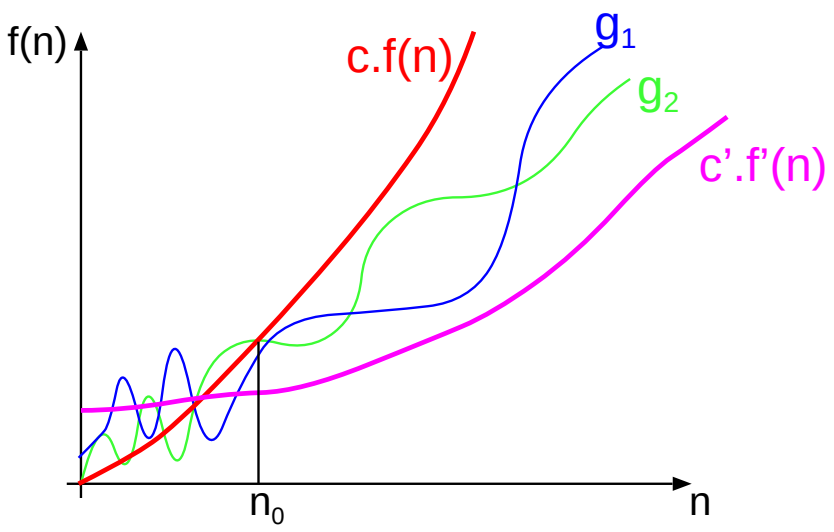
$$\Omega(f(n)) = \{g(n) \in \mathcal{F} \mid \exists c \in \mathbb{R}^+ \; \exists n_0 \in \mathbb{N} \; \forall n \in \mathbb{N} : n \geq n_0 \Rightarrow$$

$$\Rightarrow 0 \leq c \cdot f(n) \leq g(n)\}$$

  - **Asymptotické oboustranné omezení** funkce $f(n)$ je množina

$$\Theta(f(n)) = \{g(n) \in \mathcal{F} \mid \exists c_1, c_2 \in \mathbb{R}^+ \; \exists n_0 \in \mathbb{N} \; \forall n \in \mathbb{N} : n \geq n_0 \Rightarrow$$

$$\Rightarrow 0 \leq c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)\}$$




Obrázek 25.1: Asymptotická složitost.

## 25.4 Třídy složitosti

- Složitosti problémů.
- Třídy složitosti zavádíme jako prostředek ke kategorizaci (vytvoření hierarchie) problémů dle jejich složitosti, tedy dle toho, jak dalece efektivní algoritmy můžeme navrhnout pro jejich rozhodování.
- Mějme dány funkce $t, s : \mathbb{N} \to \mathbb{N}$ a nechť $T_M$, resp. $S_M$, značí časovou, resp. prostorovou, složitost TS $M$. Definujeme následující časové a prostorové třídy složitosti deterministických a nedeterministických TS:

$$DTime[t(n)] = \{L \mid \exists k\text{-páskový DTS } M : L = L(M) \ \wedge \ T_M \in \mathcal{O}(t(n))\}$$

$$NTime[t(n)] = \{L \mid \exists k\text{-páskový NTS } M : L = L(M) \ \wedge \ T_M \in \mathcal{O}(t(n))\}$$

$$DSpace[s(n)] = \{L \mid \exists k\text{-páskový DTS } M : L = L(M) \ \wedge \ S_M \in \mathcal{O}(s(n))\}$$

$$NSpace[s(n)] = \{L \mid \exists k\text{-páskový NTS } M : L = L(M) \ \wedge \ S_M \in \mathcal{O}(s(n))\}$$

- Definici tříd složitosti pak přímočaře zobecňujeme tak, aby mohly být založeny na množině funkcí, nejen na jedné konkrétní funkci.

### 25.4.1 Konkrétní třídy složitosti

**Deterministický/nedeterministický polynomiální čas** Pro všechny třídy platí, že $n$ je délka vstupu a $k$ je konstanta.

$$\mathbf{P} = \bigcup_{k=0}^{\infty} DTime(n^k) \qquad \equiv^? \qquad \mathbf{NP} = \bigcup_{k=0}^{\infty} NTime(n^k)$$

**Deterministický/nedeterministický polynomiální prostor**

$$\mathbf{PSPACE} = \bigcup_{k=0}^{\infty} DSpace(n^k) \qquad \equiv \qquad \mathbf{NPSPACE} = \bigcup_{k=0}^{\infty} NSpace(n^k)$$

**Deterministický/nedeterministický logaritmický prostor**

$$\mathbf{LOGSPACE} = \bigcup_{k=0}^{\infty} DSpace(k{\cdot}\log(n)) \qquad \equiv^? \qquad \mathbf{NLOGSPACE} = \bigcup_{k=0}^{\infty} NSpace(k{\cdot}\log(n))$$

**Deterministický/nedeterministický exponenciální čas**

$$\mathbf{EXP} = \bigcup_{k=0}^{\infty} DTime(2^{n^k}) \qquad \equiv^? \qquad \mathbf{NEXP} = \bigcup_{k=0}^{\infty} NTime(2^{n^k})$$

**Deterministický/nedeterministický exponenciální prostor**

$$\mathbf{EXPSPACE} = \bigcup_{k=0}^{\infty} DSpace(2^{n^k}) \qquad \equiv \qquad \mathbf{NEXPSPACE} = \bigcup_{k=0}^{\infty} NSpace(2^{n^k})$$

Neformálně: třídy složitosti jsou množiny problémů, pro které existuje k-páskový TS takový, že je dokáže řešit v nějaké časové/prostorové složitosti.

### 25.4.2 Příklady

**1) Dokažte, že** $2n + 5 \in \mathcal{O}(n)$

- Z definice $\mathcal{O}(n)$:
$$c \cdot n \geq 2n + 5 \text{ pro } \forall n \geq n_0$$
- Nerovnice platí pro $c = 3$ a $n_0 = 5$.

**2) Dokažte, že** $n^2 \notin \mathcal{O}(n)$

- Dokážeme sporem.
- Z definice $\mathcal{O}(n)$, musí existovat $c$ a $n_0$ takové, že
$$c \cdot n \geq n^2 \text{ pro } \forall n \geq n_0$$
- Nechť $m \geq n_0$, pak platí:
$$m^2 > c \cdot m \;\Rightarrow\; m > c$$
- Potom:
$$m \geq n_0 \;\wedge\; m > c$$
- A tedy
$$m = max(c, n_0) + 1$$
- Dosadíme zpět $m$:
$$max(c, n_0) + 1 \geq n_0 \;\wedge\; max(c, n_0) + 1 > c$$
- Což je spor.

**3) Dokažte, že** $L = \{w \mid w \in \Sigma^* \wedge w = w^R\}$ **patří do** $DTime(n)$

- Dokáže se, sestrojením algoritmu.
- Využijeme 2 páskový TS, postup:
  1. TS se přesune na konec první pásky – $\mathcal{O}(n)$.
  2. TS kopíruje obsah první pásky od konce na druhou pásku – $\mathcal{O}(n)$.
  3. TS se přesune na začátek druhé pásky – $\mathcal{O}(n)$.
  4. TS postupně znak po znaku porovnává obsah obou pásek – $\mathcal{O}(n)$.
- Celkově $4\mathcal{O}(n) = \mathcal{O}(n)$ a tady $L \in DTime(n)$.

## 25.5 Vlastnosti tříd složitosti

**Vícepáskový a jednopáskový TS**

- Je-li jazyk $L$ přijímán nějakým $k$-páskovým DTS $M_k$ v čase $t(n)$, pak je také přijímán nějakým 1-páskovým DTS $M_1$ v čase $\mathcal{O}(t(n)^2)$.
  - Více pásek nepřináší nic z pohledu vyčíslitelnosti.
  - Z pohledu časové složitosti přináší více pásek polynomiální zrychlení (resp. kvadratické).

**Deterministický a nedeterministický TS**

- Je-li jazyk $L$ přijímán nějakým NTS $M_n$ v čase $t(n)$, pak je také přijímán nějakým DTS $M_d$ v čase $2^{\mathcal{O}(t(n))}$.
  - Nedeterminismus nepřináší nic z pohledu vyčíslitelnosti.
  - Z pohledu časové složitosti, lze NTS implementovat deterministickým TS, ale za cenu exponenciálního nárůstu času.
- (Savitchův teorém) $NSpace[s(n)] \subseteq DSpace[s^2(n)]$ pro každou prostorově zkonstruovatelnou funkci $s(n) \geq \log n$.
  - Z pohledu prostorové složitosti, lze NTS implementovat deterministickým TS, ale za cenu kvadratického nárůstu prostoru.
  - Proto **PSPACE** $\equiv$ **NPSPACE**.

**Vztah prostoru a času** Intuitivně můžeme říci, že zatímco prostor může růste relativně pomalu, čas může růst výrazně rychleji, neboť můžeme opakovaně procházet týmiž buňkami pásky – opačně tomu být zřejmě nemůže (nemá smysl mít nevyužitý prostor).

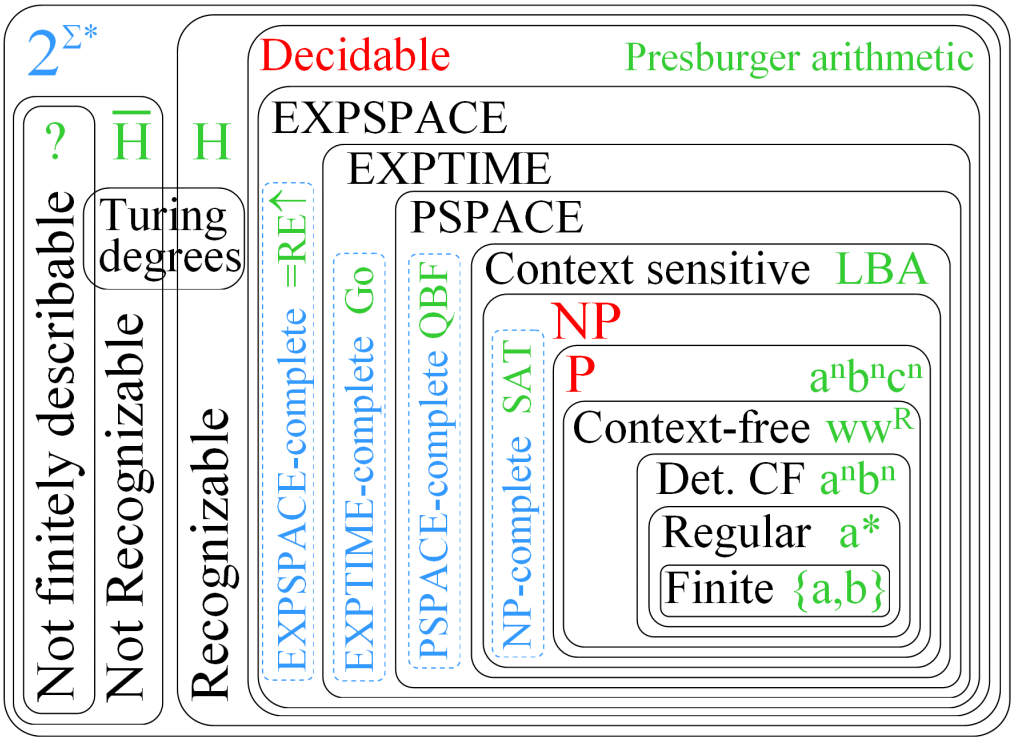
**Uzavřenost vůči doplňku** Doplňkem třídy rozumíme třídu jazyků, které jsou doplňkem jazyků dané třídy. Tedy označíme-li doplněk třídy $\mathcal{C}$ jako *co-*$\mathcal{C}$, pak $L \in \mathcal{C} \Leftrightarrow \bar{L} \in$ *co-*$\mathcal{C}$. U rozhodování problémů toto znamená rozhodování komplementárního problému (prázdnost $\times$ neprázdnost apod).

- Prostorové třídy jsou obvykle uzavřeny vůči doplňku.
- Pro časové třídy je situace jiná:
  - Některé třídy jako **P** či **EXP** jsou uzavřeny vůči doplňku.
  - U jiných významných tříd zůstává otázka uzavřenosti vůči doplňku otevřená. Proto má smysl hovořit např. i o třídách jako **co-NP**, **co-NEXP**, ...

❖ Z dosavadního můžeme shrnout, že platí následující:

- **LOGSPACE** $\subseteq$ **NLOGSPACE** $\subseteq$ **P** $\subseteq$ **NP**
- **NP** $\subseteq$ **PSPACE** $=$ **NPSPACE** $\subseteq$ **EXP** $\subseteq$ **NEXP**
- **NEXP** $\subseteq$ **EXPSPACE** $=$ **NEXPSPACE** $\subseteq$ **2-EXP** $\subseteq$ **2-NEXP** $\subseteq$ ...
- ... $\subset$ **ELEMENTARY** $\subset$ **PR** $\subset$ **R** $\subset$ **RE**

Obrázek 25.2: Hierarchie složitostních tříd. **ELEMENTARY** je „věž“ exponenciál jejíž výška je závislá na délce vstupu. **PR** jsou primitivně rekurzivní, **R** rekurzivní a **RE** rekurzivně vyčíslitelné třídy.




Obrázek 25.3: Rozšířená Chomského hierarchie.

## 25.6 Redukce, úplnost, těžkost

- Až doposud jsme třídy používali jako horní omezení složitosti problémů. Všimněme si nyní omezení dolního – to zavedeme pomocí redukovatelnosti třídy problémů na daný problém.

**Redukovatelnost** Nechť $\mathcal{R}$ je třída funkcí. Jazyk $L_1 \subseteq \Sigma_1^*$ je $\mathcal{R}$-redukovatelný (přesněji $\mathcal{R}$ *many-to-one reducible*) na jazyk $L_2 \subseteq \Sigma_2^*$, což zapisujeme $L_1 \leq_{\mathcal{R}}^m L_2$, jestliže existuje funkce $f \in \mathcal{R}$ taková, že $\forall w \in \Sigma_1^* : w \in L_1 \Leftrightarrow f(w) \in L_2$.

- Jazyk je redukovatelný na jiný jazyk jesliže existuje funkce, která tuto redukci realizuje.

**Těžkost** Nechť $\mathcal{R}$ je třída funkcí a $\mathcal{C}$ třída jazyků. Jazyk $L_0$ je $\mathcal{C}$-těžký ($\mathcal{C}$-*hard*) vzhledem k $\mathcal{R}$ redukovatelnosti, jestliže $\forall L \in \mathcal{C} : L \leq_{\mathcal{R}}^{m} L_0$.

- Jazyk je $\mathcal{C}$-těžký, jesliže lze všechny jazyky z třídy $\mathcal{C}$ na něj $\mathcal{R}$-redukovat.

**Úplnost** Nechť $\mathcal{R}$ je třída funkcí a $\mathcal{C}$ třída jazyků. Jazyk $L_0$ nazveme $\mathcal{C}$-úplný ($\mathcal{C}$-*complete*) vzhledem k $\mathcal{R}$ redukovatelnosti, jestliže $L_0 \in \mathcal{C}$ a $L_0$ je $\mathcal{C}$-těžký ($\mathcal{C}$-*hard*) vzhledem k $\mathcal{R}$ redukovatelnosti.

- Jazyk je $\mathcal{C}$-úplný, jestliže je $\mathcal{C}$-těžký a zároveň do třídy $\mathcal{C}$ sám patří.

**Polynomiální redukce** Polynomiální redukce jazyka $L_1$ nad abecedou $\Sigma_1$ na jazyk $L_2$ nad abecedou $\Sigma_2$ je funkce $f : \Sigma_1^* \to \Sigma_2^*$, pro kterou platí:

- $\forall w \in \Sigma_1^* : w \in L_1 \Leftrightarrow f(w) \in L_2$
- $f$ je vyčíslitelná DTS v polynomiálním čase.

Existuje-li polynomiální redukce jazyka $L_1$ na $L_2$, říkáme, že $L_1$ se (polynomiálně) redukuje na $L_2$ a píšeme $L_1 \leq_P^m L_2$.

## 25.7 Příklady NP-úplných problémů

- Existence kliky dané velikosti v grafu (*clique problem*)
- Existence Hamiltonovské kružnice v grafu (*hamiltonian cycle problem*)
- Barvení Rgrafů (*graph coloring*)
- Problém obchodního cestujícího (*travelling salesman problem*)
- Problém batohu (*knapsack problem*)
- Prvočíselný rozklad čísel (*prime factorization*) – Neví se, jestli je NP-úplný.
- Problém splnitelnosti booleovské formule (*SAT problem*)

### SAT problém

- Nechť $V = \{v_1, v_2, \ldots, v_m\}$ je konečná množina Booleovských proměnných (prvotních formulí výrokového počtu). Literálem nazveme každou proměnnou $v_i$ nebo její negaci $\bar{v}_i$. Klausulí nazveme výrokovou formuli obsahující pouze literály spojené výrokovou spojkou $\vee$ (nebo, disjunkce).
- SAT problém (*boolean satisfiability problem*): Je daná množina výrokových proměnných $V$ a množina klauzulí nad $V$. Je tato množina klauzulí splnitelná?
- Příklad 3-SAT:

$$(x \vee \neg x \vee y) \wedge (\neg x \vee \neg y \vee \neg y) \wedge (\neg x \vee y \vee y)$$

- Rozhodovací problém: množina splnitelných booleovských formulí (tedy jazyk takovýchto zakódovaných formulí).

# Kapitola 26

# UPA – Postrelační a rozšířené relační databáze (objektový a objektově relační databázový model – struktura a operace; podpora práce s XML a JSON dokumenty v databázích).

## 26.1 Zdroje

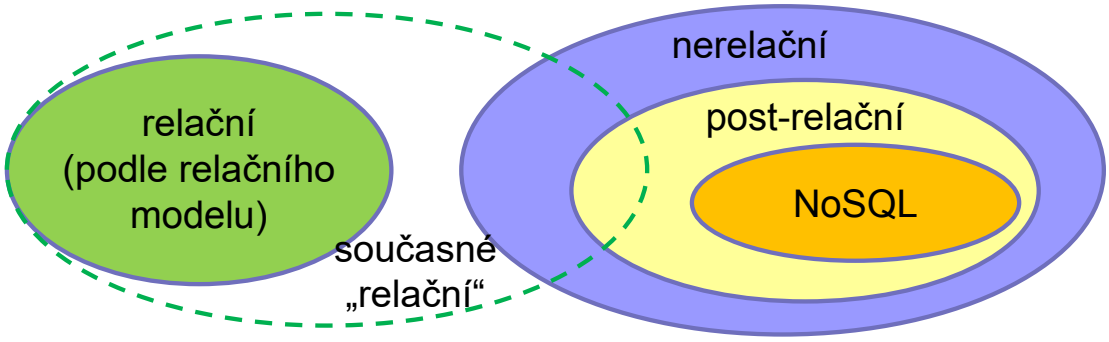
- `01-uvod.pdf`
- `02-ordbms.pdf`
- `szz-discord-bazi.pdf`
- `szz-kastak.pdf`
- `Wikipedia`
- Mé materiály k UPA.

## 26.2 Úvod a kontext

- Postrelační databázový systém je relační databázový systém rozšířený o určitou specifikaci na databázové úrovni, protože aplikační řešení problému by bylo nedostačující (neefektivní).
  - Na konci 80. let roste vliv objektové orientace. To má za důsledek vznik objektových databází a rozširování schopností relačních databází (SŘBD) nad rámec relačního modelu.
- Vývoj databází
  - Relační model, SŘBD, standard SQL.
  - Objektově orientované SŘBD.
  - Objektově relační DB, datové sklady, získávání znalostí z dat.
  - NoSQL databáze, podpora XML a JSON, grafové databáze, BigData, Data Science.

### 26.2.1 Relační databáze

- Datový model:
  - relační model dat;
  - kolekce (normalizovaných) tabulek (relací);
  - omezená množina datových typů (jednoduché datové typy, typicky pevné délky);
  - vztahy mezi řádky tabulek ($1 : N$) vyjádřeny pomocí cizích a kandidátních klíčů;
  - pro vztahy $M : N$ je nutná vazební tabulka.
- Dotazovací jazyk:
  - deklarativní – SQL (matematická logika).
- Výpočetní model:
  - založený na hodnotách ve sloupcích tabulek;
  - žádné reference ani ukazatele;
  - jednoduchá navigace pomocí kurzoru.
- Standardizováno (SQL 1986).




Obrázek 26.1: Schéma databází.

## 26.3 Objektové databáze

- Objektová databáze je databázový řízený systém, ve kterém je informace reprezentována ve formě objektů a používá se v objektově orientovaném programování.
- Umožňují modelování a vytváření perzistentních dat jako objektů.
- OSŘBD (Objektový systém řízení báze dat) musí splňovat kritéria ŠŘBD a musí to být OO systém.
- Objektové databáze nenahrazují, nýbrž doplňují relační (dnes spíše objektově-relační).
- Datový model:
  - Neexisuje jednoznačná definice jako u relačního modelu.
  - Objektový (třídy, zapouzdření, dědičnost, polymorfismus, jednoznačný OID, komunikace zasílání zpráv).

  - Databáze je tvořena kolekcí objektů, které mezi sebou komunikují zasíláním zpráv.
  - Vlastnosti objektu jsou definovány třídou, ta zapouzdřuje data i chování.
  - Data jsou reprezentovány atributy, chování je definováno metodami třídy.
  - Atributy objektů mohou být jiné objekty (abstraktní datové typy).
  - Vztahy objektů vytvářené pomocí referencí prostřednictvím OID.
  - Vztahy $M : M$ lze vytvářet přímo.
- Dotazovací jazyk:
  - Object Query Langauge (OQL) – snaha o vytvoření standardu jako obdoba SQL;
  - Objektově orientované jazyky (typicky Java).
- Výpočetní model:
  - Navigace po objektové struktuře pomocí referencí prostřednictvím OID.
- Neexistuje standardizace.

## 26.4 Objektově relační databáze

- Objektově relační databáze je hybridem objektové a relační databáze.
  - Cílem je spojit výhody relačního a objektového modelu.
- Objektové rysy ve standardu SQL.
- První OSŘBD: PostreSQL (open source).
- Podpora objektových rozšíření u některých SŘBD (Oracle, IBM, ...).
- Datový model:
  - Relační základ (tabulka) zůstává, ale jde o obecnější (vnořené) relace. Hodnoty mohou mít bohatší strukturu definovanou jako abstraktní datový typ (ADT).
  - ADT zapouzdřuje data a operace.
  - Podpora dědičnosti a polymorfismu.
  - Zavedena obdoba OID, která umožňuje vytvářet nový typ vazeb mezi tabulkami.
  - Nesplňuje normalizaci relační databáze.
- Dotazovací jazyk:
  - dialekt SQL (objektové rozšíření).
- Výpočetní model:
  - navigace po tabulkách pomocí kurzoru i pomocí reference.
- Standardizováno (SQL 1999).
- Objektově relační rysy:
  - Objektový model je založený na konceptech.
    * Strukturovaný datový typ.

    - Typová tabulka.
  - Uživatelem definované typy (UDT).
    - Odpovídají abstraktním datovým typům.

## 26.5 Objektově relační mapování

- Objektově relační mapování slouží k odstranění sémantické „mezery“ mezi OO modelem aplikací v OO jazyce a relačním databázovým modelem.
  - Zajišťuje automatickou konverzi dat mezi relační databází a objektově orientovaným programovacím jazykem.
- Zajišťuje:
  - Konverzi mezi relační databází a objekty v aplikaci.
  - Perzistenci objektů (synchronizaci s DB).
  - Poskytnutí základních CRUD operací bez nutnosti programování v SQL.
  - Automatickou konverzi datových typů.
- Příklad:
  - SQLAlchemy pro Python.
- Schéma komunikace:

$$\text{Aplikace} \Leftrightarrow \text{ORM} \Leftrightarrow \text{DB}$$

## 26.6 Práce s XML

- **XML** (*Extensible Markup Language*) je obecný značkovací jazyk, který umožňuje snadné vytváření konkrétních značkovacích jazyků (tzv. aplikací) pro různé účely a různé typy dat. Používá se pro serializaci dat (soupeří s JSON či YAML).
- Datové XML soubory:
  - Soubor XML obvykle použit pro zápis dat (typicky k přenosu).
  - Typický životní cyklus: DB $\rightarrow$ XML $\rightarrow$ DB.
  - Pravidelná struktura, jemná granularita, nezáleží na pořadí dat (rozhodující je název elementu / atributu).
  - Příklady: faktury, objednávky; dokumenty vytvořené podle šablony; export dat z relační DB.
- Dokumentově zaměřené XML soubory:
  - Často určen pro čtení či zpracování lidmi.
  - Méně pravidelná struktura, pořadí elementů zásadní.
  - Data typicky nepochází z relačních databází, nejvýše z dokumentových.
  - Příklady: e-knihy, e-maily, WWW stránky v XHTML, ...
- Nativní XML databázové systémy – Datový model vychází z modelu XML dokumentu (zpravidla varianta DOM mapovaná na příslušné úložiště) $\rightarrow$ rychlost.
- Wrapper – programy umožňující relační pohled nad XML dokumentem nebo dokumenty.

### 26.6.1 Databázové systémy s podporou XML

- DB → XML, XML → DB (*middleware*).
- Většina komerčních a částečně i otevřených rozšířených SŘBD pro datově zaměřené aplikace.
- Navázání XML dat – Z dokumentu XML vytvoří objekty zapouzdřující data, resp. z XML schématu třídy pro práci s dokumentem.
- Standard SQL – Datový typ XML, mapování SQL a XML, specializované funkce, podpora dotazování (XQuery).

### 26.6.2 Ukládání XML dokumentů do databáze

- Uložení do relační tabulky:
  - Používané dříve, ztrácí výhody XML.
  - Rozklad / sestavení XML dokumentu na aplikační úrovni standardní uložení / načtení do / z DB.
- Uložení do CLOB (*character large object*):
  - Flexibilní s ohledem na změnu schématu.
  - Rychlé získání dokumentu, operace nad částmi pomalé a paměťově náročné.
  - Pomalé aktualizace.
- Objektově-relační uložení:
  - Použití typu XMLType pro vytvoření sloupcových objektů a objektových tabulek.
  - Mapování a validace na základě XML schématu.
  - Efektivní práce s částí dokumentu pomocí XPath.
  - Možnost využití podpory pro omezení v SQL (např. PRIMARY KEY).

## 26.7 Práce s JSON

- **JSON** (*JavaScript Object Notation*) je způsob zápisu dat (datový formát) nezávislý na počítačové platformě, určený pro přenos dat, která mohou být organizována v polích nebo agregována v objektech.
- Výhody oproti XML:
  - úspornější,
  - čitelnější člověkem,
  - méně přísný (snadněji programově zpracovatelný),
  - mnohem snadnější práce s poli.
- Podpora v SQL:
  - Nemá vlastní datový typ, ale je implementován jako varianta řetězce (VARCHAR2, CLOB).
  - V jednotlivých dialektech existují funkce, které berou jako parametr JSON v řetězci, dokáží ho naparsovat a dále s ním pracovat (`is_json()`, `json_value()`).

- Podpora v NoSQL:
  - Volné schéma dvojic klíč a hodnota.
  - Např. MongoDB.

# Kapitola 27

# UPA – NoSQL databáze (porovnání relačních a NoSQL; CAP věta a ACID/BASE principy; typy NoSQL databází; dotazování v NoSQL databázích; agregace dat pomocí Map-Reduce a agregační pipeline).

## 27.1 Zdroje

- `09-nosql_databases.pdf`
- `UPA_2019-11-19.mp4`
- `szz-kastak.pdf`
- `https://youtu.be/W2Z7fbCLSTw`

## 27.2 Relační databáze

- Data organizována do tabulek, řádek reprezentuje záznam (koncept matematické relace, řádek prvkem relace nad doménami sloupcu tabulky).
- Každý sloupec má přesně daný (jednoduchý) datový typ (tj. množina/doména odpovídající části relace).
- Záznam v tabulce se může odkazovat na záznam (jiné) tabulky (hodnota cizího klíče odpovídá hodnotě primárního klíče odkazovaného záznamu).
- Organizace dat musí splňovat normální formy (1NF, 2NF, 3NF, EKNF, BCNF, 4NF, 5NF, DKNF, 6NF1; jinak hrozí redundance/chyby).
- Dotazy a úpravy nad daty pomocí SQL (dotazování pomocí SELECT vychází z relační algebry).
- Databázový systém zaručuje **ACID** změnu uložených dat (Atomicity, Consistency, Isolation, Durability).
  - Jeho zaručení není jednoduché, je třeba provádět a plánovat transakce, řídít

souběžný přístup (zamykání), obnovení (rollback), ...

### 27.2.1 ACID

- **Atomicity** – Atomičnost transakcí, žádný rozpracovaný stav a to i ve vztahu k možné chybě OS či HW (proběhne celá transakce, tj. všechny její změny, nebo nic).
- **Consistency** – V DB jsou pouze platná data dle daných pravidel (integritní omezení). Transakce se neuskuteční, pokud to nelze dodržet, jinak platí, že původní i nový stav je platný.
- **Isolation** – Souběžné transakce se neovlivňují. Serializace. Pořadí však není zajištěno.
- **Durability** – Uskutečněná transakce nebude ztracena (její projev). Podpora obnovy dat (*rollback*) po pádu HW/SW.

## 27.3 Moderní databáze

### 27.3.1 Kladené požadavky

- Cloudová řešení, distribuované databáze:
  - Decentralizace uložiště dat.
  - Úmyslná redundance proti výpadkům a pro zajištění rychlosti.
  - Velké objemy dat a velké množství operací (big data).
- Problematické datové typy:
  - Údaje klíč-hodnota, objekty, nestrukturované dokumenty, RDF grafy.
- Iterativní vývoj:
  - Časté změny schématu databáze.
  - Různé způsoby použití databáze.
- Vysoké požadavky na škálovatelnost:
  - Mobilní zařízení jako klienti.
  - Nerovnoměrné rozložení zátěže (prostorové i časové).
  - Specifické požadavky na dostupnost.
  - Předem neznámé dotazy, nelze optimalizovat indexy.

### 27.3.2 Moderní relační databáze

- Pro uspokojení požadavků na moderní databáze, se zbavujeme relačního modelu a opouštíme předchozí zásady (ACID).
- Některé požadavky na moderní databáze není možné uspokojit pouze pomocí rozšíření relačních databází, proto vznikají nová paradigmata.
- Dodržování ACID někdy nevhodně omezuje práci s databází, což vede k jeho úmyslnému zanedbání nebo opuštění pro zisk rychlosti a dostupnosti dat.
- To má za důsledek vznik specializovaných nerelačních (postrelačních) databází pro strukturovaná data nebo pro specificky přistupovaná data.

### 27.3.3 Co bychom chtěli místo ACID

- CAP
  - **Consistency** – Každý uzel (každý klient) vidí ve stejný čas stejná data (data jsou konzistentní nezávisle na běžících operacích či jejich umístění).
  - **Availability** – Každý požadavek obsloužen, úspěšně nebo neúspěšně, nepřetržitý provoz (nepřetržitý provoz, vždy možnost číst a zapisovat).
  - **Partition tolerance** – Funkční navzdory chybám sítě nebo výpadkům uzlů.
- Cap věta: „U sdílených (distribuovaných) systémů je možné uspokojit maximálně 2 ze 3 výše uvedených požadavků."
- Proč?
  - **Konzistence a dostupnost** – Databáze není distribuovaná, bude vždy dostupná a konzistentní. To znamená vysokou latenci u vzdálených klientů a také podstatně nižší bezpečnost, riziko výpadku apod.
  - **Konzistentní a odolná** – Databáze distribuovaná na spoustě místech. Aby byla konzistentní, musí se při jakékoliv změně na nějaký okamžik zamknout a počkat až se změny synchronizují. Tím ztratí dočasnou dostupnost.
  - **Dostupná a odolná** – Databáze distribuovaná na spoustě místech, kterou při změně nezamkneme. Synchronizace nějakou dobu trvá. Klient může někdy dostat neaktuální (zastaralá) data.
- Co s tím?
  - Něco musíme obětovat (řešit jinak), různá řešení pro různé požadavky.
  - Konzistence a dostupnost – clusterové databáze, LDAP.
  - Konzistence a odolnost – distribuované databáze.
  - Dostupnost a odolnost – web caching, DNS.

## 27.4 NoSQL databáze

- Nerelační datový model (klíč-hodnota, dokumenty, grafy, ...) a distribuovaná architektura.
- Kdy jsou vhodné?
  - Pro distribuované uložení či zpracování nerelačních dat.
  - Pokud potřebujeme vysokou rychlost na úkor ACID.
- Kdy nejsou vhodné?
  - NoSQL nenahrazují klasické relační databáze, mají jiné využití, pro klasické informační systémy je nejlepší relační databáze.
- NoSQL dodržují tzv. BASE, resp. BaSsEc (CAP s kompromisy), znamená následující:
  - **Basically available** – Většinu času je dostupná. Data nejsou uložena na všech uzlech, jsou rozloženy na některých uzlech s jistým replikačním faktorem (např.: uchováváme tři kopie na nejvíc potřebných místech).

  - **Soft-state** – Databáze může obsahovat neaktuální data a klient s tím musí počítat.
  - **Eventual consistency** – Databáze bude konzistentní po nějaké době (konečný čas).
- Tedy omezuje se konzistence a databáze je vysoce dostupná a odolná proti výpadkům.
- Princip NoSQL paradigmat:
  - Několik typů databází, většina založena na principu klíč-hodnota.
  - Implementováno pomocí hashování (extrémně rychlé).
  - Hodnota je BLOB (*binary large object*) – databáze se je často vůbec nesnaží chápat.
  - Nemá předdefinované schéma – flexibilita.
  - Vysoká škálovatelnost (přidávání, ubírání dat).
  - Pokud nás zajímá pouze část hodnoty, ať pro dotazy nebo zápis, tak je poměrně neefektivní.
- Podpora *partitioning* (*sharding*) – Rozdělení dat na různé uzly v clusteru.

ACID:

- silná konzistence
- izolovanost
- orientace na komit
- vnořené transakce
- dostupnost?
- konzervativní (pesimistické)
- složitá evoluce (schématu, . . . )

BASE:

- slabá konzistence (stará data)
- dostupnost na prvním místě
- přibližné odpovědi jsou OK
- jednodušší, rychlejší
- dodávka dat „jak to jen půjde“
- agresivní (optimistické)
- jednodušší evoluce

Obrázek 27.1: Porovnání ACID vs BASE.

### 27.4.1 Jednotlivá NoSQL paradigmata

**Key-value databáze**

- Databáze key-value je jednoduché paradigma ukládání, načítání a správu dat ve formátu key-value (podobná struktura jako Python slovník, nebo JSON).
- Nevyhodnocují se žádné *queries* ani *joins*.
- Extrémně rychlé, jelikož všechna data jsou držena v paměti.
- Využití:
  - jako cache nad jinou datovou vrstvou,

  - jako message broker (např. Celery, systém pro správu distribuované fronty úloh).
- Např. Redis, Memcached, Oracle NoSQL, ...

**Sloupcové databáze (*wide column*)**

- Řádky jako u relační databáze, narozdíl od relačních databází nemá schéma (ačkoliv data jsou svým způsobem strukturovaná).
- Klíč-hodnota, kde hodnota je řádek v tabulce.
- Optimalizované pro čtení dat po sloupcích, místo po řádcích (vhodné pro analýzu).
- Dotazovací jazyk podobný SQL (nemá *joins*).
- Využití:
  - škálování velké množství time-series dat (záznamy z IoT zařízení, senzorů, ...),
  - pro historická data,
  - high-write, low-read.
- Např. Apache Cassandra, Apache HBase (Google Bigtable), ...

**Dokumentové databáze (*document oriented*)**

- Nestrukturované dokumenty (bez schéma), každý dokument ve formátu key-value.
- Dokumenty mohou být seskupovány do kolekcí, kolekce mají indexy a mohou být organizovány do hierarchie.
- Nepodporují joiny.
- Využití:
  - mobilní hry,
  - IoT,
  - obecně aplikace,
  - celkově široké využití.
- Např. MongoDB, Firestore, ...

**Grafové databáze**

- Alternativa k relačním databázím.
- Uchováváme uzly a hrany, které tyto uzly spojují.
- Využití:
  - RDF dokumenty,
  - Reprezentace sítí (dopravní, počítačové) a jejich topologie.
- Např. Neo4j, Dgraph, ...

### 27.4.2 Dotazování v NoSQL

- Klíč se využívá na:

  - Nalezení správného uzlu v clusteru pro daný klíč (pomocí directory service a nebo distribuované hashovací tabulky).
  - Následně nalezení diskového bloku, kde je hodnota uložena v rámci uzlu (pomocí stromových struktur, hashovací tabulky, ...).
- Je těžké vykonávat nad dotazi nějaké optimalizace (jako se děje v relačních databází).
- Typicky není podporována *join* operace.
- Filtrování podle klíče:
  - Vždy podporováno.
  - Klíč je primární index, implementovaný pomocí distribuované hashovací tabulky.
  - Je možné zavést druhořadé indexy, pouze pro dotazování (ne pro *sharding*).
- Filtrování podle hodnoty:
  - Podporované pouze ve specializovaných NoSQL.

### 27.4.3 Agregace dat

- Agregace $\sim$ *grouping*.
- Je omezené hranicemi daného shardu (je snažší agregovat data v rámci shardu).
- Map-Reduce se používá pro složité nebo vlastní agregace.
- **Agregace pomocí Map-Reduce**
  - Ve fázi *map* se odděleně vykoná daná operace *map* na dataset na každém uzlu (transformace vstupní kolekce na výstupní).
  - Výsledky operace *map* ze všech uzlů, se podle klíče spojí dohromady (tzv. *handshake* fáze pro výměnu dat mezi uzly).
  - Nakonec se podle jednotlivých hodnot klíče aplikuje zadaná *reduce* operace na každý seznam hodnot.
- **Agregační pipeline**
  - V MongoDB.
  - Dokumenty vstupují do více úrovňové pipeline, která je transformuje na výsledek.
  - Jde o pipeline pro zpracování a hlavní agregaci a filtrování dat (agregační vzory).
  - Všechny výsledky, průběžné i finální, jsou kolekce dokumentů.

# Kapitola 28

# UPA – Získávání znalostí z dat (pojem znalost; typické zdroje dat; základní úlohy získávání znalostí; analytické projekty a proces získávání znalostí z dat).

## 28.1 Zdroje

- Má DP.
- `10-ziskavani_znalosti.pdf`
- `UPA_2019-11-26.mp4`
- `UPA_2019-12-03.mp4`

## 28.2 Úvod a kontext

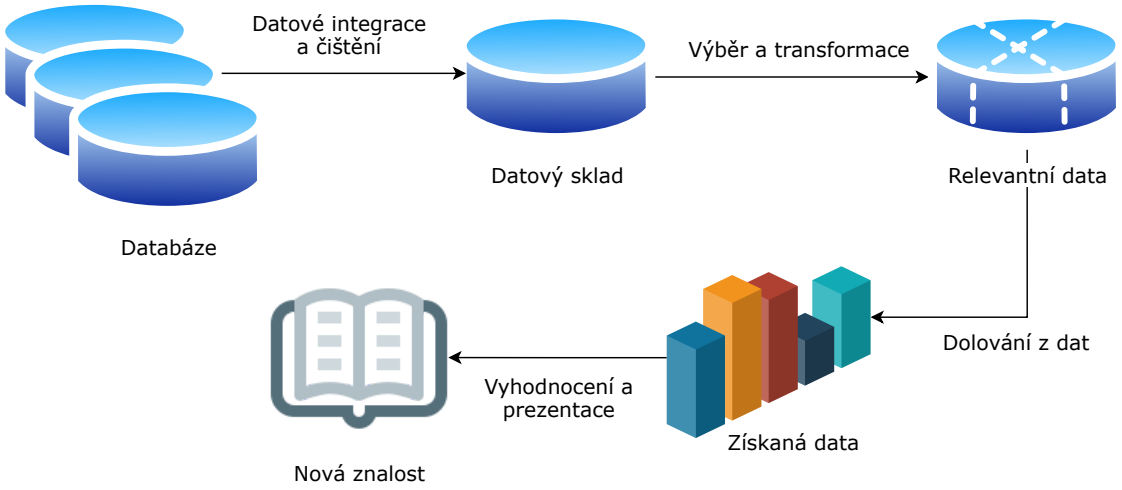
- Získávání znalostí z databází (dat) je proces extrahování a objevování vzorů ve velkých datových souborech na pomezí strojového učení, statistiky a databázových systémů.
- Cílem je získat informace dopředu neznámé a potenciálně užitečné (**znalost**) a transformovat je do srozumitelné struktury pro další použití.

## 28.3 Proces získávání znalostí z dat

- Celý proces získávání znalostí je zobrazen na obrázku dále. Jedná se o iterativní sekvenci kroků, které jsou představeny v následujícím výčtu.

1. **Datové čištění** – Odstranění šumu, nekonzistencí, neúplností a dalších znaků dat nízké kvality.
2. **Datové integrace** – Kombinace několika různých datových zdrojů. Proces čištění a integrace bývá typicky prováděn v jednom kroku jako forma předzpracování (příprava dat). Výsledky tohoto kroku pak jsou ukládány do datového skladu.

3. **Výběr** – Selekce relevantních dat z datového skladu pro konkrétní dolovací úlohu.
4. **Datové transformace** – Transformace dat do potřebné podoby pro konkrétní dolovací algoritmus.
5. **Dolování z dat** – Hlavní krok ve kterém jsou použity statistické metody či algoritmy strojového učení pro získávání nových dat (datových vzorů).
6. **Vyhodnocení** – Vyhodnocení a identifikování datových vzorů, které mohou představovat novou znalost.
7. **Prezentace** – Předávání a prezentování získaných znalostí uživatelům pomocí vhodných vizualizačních technik.

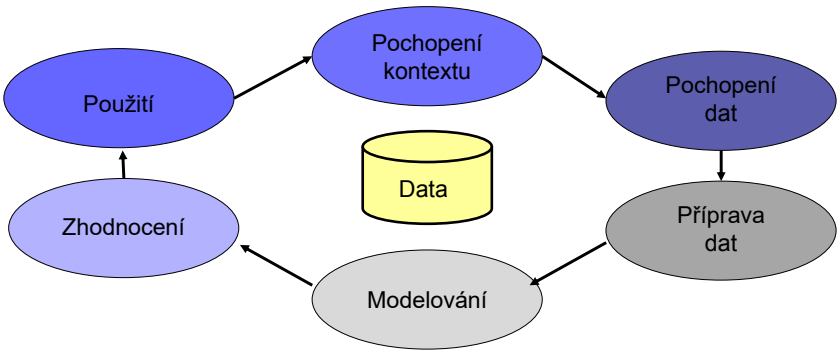



Obrázek 28.1: Fáze procesu získávání znalostí z databází. V diagramu je možné se vracet zpět a jednotlivé fáze libovolně revidovat a opakovat (např. zvolit jiné parametry dolovacího algoritmu pro získání lepších výsledků).

### 28.3.1 Model CRISP

- Model CRISP (*CRoss-Industry Standard Process for Data Mining*) je formální, teoretický model, popisující proces získávání znalostí z databází.
- Časté jsou návraty mezi fázemi modelu.

1. **Pochopení kontextu** – Cílem je dozvědět se co nejvíce o kontextu řešeného problému (pochopení dané problematiky).
2. **Pochopení dat** – Cílem je dozvědět se o dostupných/potřebných datech co nejvíce (popis, prozkoumání a zhodnocení kvality dat).
3. **Příprava dat** – Cílem je připravit data pro aplikaci dolovacího algoritmu (výběr, čištění, integrace, transformace).
4. **Modelování** – Cílem je vytvořit model připravených dat.
5. **Zhodnocení** – Cílem je zhodnotit dosažené výsledky a další postup.
6. **Použití** – Cílem je stanovit strategie použití výsledků a naplánování kroků.




Obrázek 28.2: Model CRISP.

## 28.4 Analytické (data mining) projekty

- Analogie bežných softwarových projektů.
- Zadavatel: vlastní organizace / jiná organizace.
- Řešitel: datoví analytici (+ programátoři), doménoví experti.
- Vstupy:
  - Vymezení problému k řešení pro rozhodování na základě dat (*Data-Driven Decision making*).
  - Dostupná data (interní, externí).
- Výstup: využitelná znalost – model dat, vzory v datech, profily atd.
- Technologie:
  - Programování s využitím knihoven (např. Python + Scikit Learn).
  - Prostředí nějakého dolovacího nástroje (např. Rapid Miner).

## 28.5 Typické zdroje dat

- Relační databáze
- Datové sklady – Modelován jako multidimenzionální datová kostka. Zpřístupnění pomocí OLAP operací.
- Transakční databáze – Soubor / tabulka jehož každý záznam reprezentuje obchodní transakci.
- Dokumenty (textové / multimediální) – Datový soubor.
- Proudy dat (senzory, kamery, ...).
- Velmi rozsáhlá data (**Big Data**)
  - vysoký objem (*volume*),
  - vysoká rychlost vzniku (*velocity*),
  - rozličnost zdrojů, typů, formátů (*variety*),
  - věrohodnost, nejistá vlivem nekonzistence, neúplnosti (*veracity*).

## 28.6 Úlohy získávání znalostí

- Typy dolovacích úloh můžeme rozdělit do dvou základních skupin:
  - **Deskriptivní** – Deskriptivní úlohy charakterizují obecné vlastnosti a vztahy v analyzovaných datech, které mohou pomoci učinit lepší rozhodnutí (korelační analýza, dolování frekventovaných vzorů, asociační analýza, shluková analýza, analýza odlehlých hodnot). Např. časté společné nákupy zboží při analýze nákupního košíku.
  - **Prediktivní** – Prediktivní úlohy provádí předpověď budoucího chování na základě analýzy současných dat (klasifikace, regrese). Např. předpověď chování nového zákazníka na základě dosavadního chování zákazníků.

### 28.6.1 Frekventované vzory

- Frekventované vzory (*frequent patterns*) jsou vzory, které se v datech vyskytují často. Jejich dolování vede objevování zajímavým asociací a korelací v datech.
- Existuje mnoho druhů frekventovaných vzorů, včetně frekventovaných množin, frekventovaných podsekvencí (sekvenční vzory) a frekventovaných podstruktur.
- Frekventovaná množina obvykle označuje množinu položek, které se často vyskytují společně v souboru transakčních dat (např. mléko a chléb, které mnoho zákazníků mnohdy kupuje společně).
- Frekventované podsekvence značí události, které často nastávají po sobě (např. zákazníci mají tendenci kupovat digitální fotoaparát a poté paměťovou kartu).
- Substruktura se může vyskytovat v datech, která mají strukturní formu (např. grafy, stromy nebo mřížky).

### 28.6.2 Asociační analýza

- Asociační analýza (*association rule learning*) je metoda nalézání asociačních pravidel spojujících zároveň se vyskytující atributy (např. události, položky atd.) ve zkoumaných datech.
- Typickým použitím je analýza nákupního košíku, tj. zkoumání, které položky jsou často nakupovány společně.
- Příkladem takového asociačního pravidla může být

$$kupuje(X, pocitac) \rightarrow kupuje(X, monitor) \tag{28.1}$$

  kde $X$ je proměnná reprezentující zákazníka. Pravidlo říká, že pokud si zákazník koupí počítač, pravděpodobně si koupí i monitor.

### 28.6.3 Klasifikace

- Klasifikace (*classification*) je forma prediktivní úlohy, kdy je novým datovým vzorům přiřazováno předem známé kategoriální označení tříd (zařazování dat do skupin na základě různých aspektů).
- Taková analýza pomáhá lépe porozumět datům jako celku.
- Z hlediska strojového učení se jedná o úlohu učení s učitelem.

- Klasifikátor je nejprve třeba naučit na označené trénovací datové sadě. Až poté je možné, aby klasifikátor predikoval přiřazení třídy novým datům.
- Možné využití klasifikace:
  - Klasifikace potenciálních zákazníků banky na základě rizikovosti na ty, kterým je vhodné poskytnout úvěr a na ty kterým ne.
  - Klasifikace e-mailů – Rozhodnout, zda se jedná o spam či nikoliv.
  - Stanovení lékařské diagnózy na základě výsledků různých vyšetření.
  - Cílení marketingu – Vyhodnotit kterému uživateli zobrazit jakou reklamu.
- Mezi běžné klasifikační metody patří:
  - klasifikátor k-nejbližších sousedů,
  - Bayesovská (naivní) klasifikace a Bayesovské sítě,
  - rozhodovací stromy,
  - SVM (*Support Vector Machine*),
  - klasifikace pomocí neuronových sítí.

### 28.6.4 Regrese

- Regrese (*regression*) je prediktivní úloha, ve které se na základě znalosti jistých atributů predikuje hodnota jiného (cílového) atributu.
- Narozdíl od klasifikace, není cílový atribut kategorický, nýbrž numerický (má spojitý charakter).
- Možné využití regrese:
  - Určení výše platu pracovníka na základě znalostí jeho dalších vlastností.
  - Predikce očekávané pooperační délky života pacientů trpících rakovinou.
  - Predikce výšky dítěte.
  - Predikce vývoje ceny komodity.
- Pro účely regrese lze využít některé klasifikační algoritmy (neuronové sítě, k-nejbližších sousedů, SVM) rozšířené o predikování spojité veličiny a nebo regresní analýzu. Regresní analýza je matematická metoda hledání závislostí mezi atributy. V podstatě jde o aproximaci trénovacích dat vhodnou funkcí (regresní funkce).
- Jsou rozlišovány různé typy:
  - **Jednoduchá lineární regrese** – Cílový atribut závisí na jednom dalším atributu lineárně.
  - **Vícenásobná lineární regrese** – Cílový atribut závisí na několika dalších atributech lineárně.
  - **Nelineární regrese** – Cílový atribut závisí na dalších atributech nelineárně.

### 28.6.5 Shluková analýza

- Na rozdíl od klasifikace nejsou při shlukové analýze (*clustering*) předem známy třídy rozdělení dat.
- Proces shlukování se sám snaží data rozdělit do přirozených skupin (shluků).

- Vytvořené shluky pravděpodobně odrážejí nějaký fenomén v datech, který způsobuje, že některé instance jsou si navzájem podobnější než jiné.
- Vytvořené shluky pravděpodobně odrážejí nějaký fenomén v datech, který způsobuje, že některé instance jsou si navzájem podobnější než jiné.
- Existují různé způsoby, jak lze vyjádřit výsledek shlukování:
  - Identifikované shluky tvoří disjunktní množiny. Každá instance patří pouze do jednoho shluku a každý shluk obsahuje alespoň jednu instanci.
  - Identifikované shluky se mohou překrývat. Každá instance může patřit do několika shluků.
  - Identifikované shluky se překrývají a každá instance patří do každého shluku s určitou pravděpodobností.
  - Identifikované shluky tvoří hierarchickou strukturu. Na nejvyšší úrovni je hrubé rozdělení instancí do shluků a každý shluk je dále upřesněn, v krajním případě až na jednotlivé instance.
- Za základní algoritmus shlukování je považován algoritmus **k-means** (*k-means clustering*). Data rozděluje do shluků, které tvoří disjunktní množiny a počet shluků musí být předem zadán ($k$). Na začátku je náhodně vybráno $k$ počátečních bodů, které představují počáteční středy shluků. Všechny datové body jsou přiřazeny do shluků podle vzdálenosti k nejbližšímu středu. Poté je vypočtena střední hodnota bodů v každém shluku, aby se vytvořil nový střed shluku. Takto algoritmus iterativně pokračuje, dokud dochází ke změnám ve shlucích.

### 28.6.6 Analýza odlehlých hodnot

- Cílem analýzy odlehlých hodnot (*outlier detection*) je nalezení extrémních hodnot, které se výrazně liší od ostatních.
- Takové datové objekty se nazývají odlehlé (anomálie). Možné využití analýzy odlehlých hodnot:
  - Odhalení podezřelého chování na záběrech z bezpečnostních kamer.
  - Detekce vadných kusů při výrobě.
  - Odhalení platby odcizenou platební kartou.
  - Detekce chyb měření.
- Pro tyto účely je možné využít statistické metody. Nejprve jsou v datech hledány pravděpodobností rozdělení, které stojí za generováním dat. Datové vzory, pro které je velmi nepravděpodobné, že byly vygenerovány tímto vybraným rozdělením, jsou prohlášeny za odlehlé hodnoty.
- Dále je možné využít některé upravené klasifikační metody, jako jsou algoritmus k-nejbližších sousedů, neuronové sítě a SVM. Také je možné využít některé shlukovací algoritmy, které dokáží přirozeně detekovat hodnoty, které se výrazně liší od ostatních a nepřiřadit je do žádného shluku. Jedná se o tzv. metody shlukování založené na hustotě.

# Kapitola 29

# UPA – Porozumění datům (důvod a cíl; popisné charakteristiky dat a vizualizační techniky; korelační analýza).

## 29.1 Zdroje

- `11-porozumeni_datum.pdf`
- `UPA_2019-12-03.mp4`
- `UPA_2019-12-10.mp4`

## 29.2 Důvod a cíl porozumění datům

- Druhý krok modelu procesu CRISP.
- Cílem je získat detailní informace o datech, jejich vlastnostech a vytvořit podklady pro datovou sadu. Dozvědět se o datech co nejvíce.
- Předpoklady (z pohledu dat):
  - chápeme problematiku (máme kontext),
  - máme stanovený cíl projektu a kritéria úspěšnosti,
  - víme, jaká data jsou / budou k dispozici,
  - víme, o jakou DM úlohu půjde.
- Vstup:
  - Dostupné datové zdroje a dokumentace k nim.
- Výstup:
  - Podklady pro vytvoření počáteční datové sady nebo přímo vytvoření.
  - Detailní informace o vlastnostech dat (popisné charakteristiky + grafy).
- Dosažením cíle zahrnuje kroky:
  - Rozpracovaní informace – Jaké data máme, význam, dostupnost, cena, věrohodnost dat.

- Popis dat – struktura, význam, formát a množství dat.
- Prozkoumání dat (explorační analýza) – popisné charakteristiky, grafy, korelace, ...
- Zhodnocení kvality dat – Úplnost, chybějíci hodnoty, šum, ...

## 29.3 Popisné charakteristiky z hlediska distribuce výpočtu

- **Distributivní**
  - Lze počítat distribuovaně.
  - Např.: počet prvků.
- **Algebratické**
  - Výsledek je algebraická operace nad mezivýsledky, které lze počítat distribuovaně.
  - Např.: průměr.
- **Holistické**
  - Lze spočítat pouze výpočtem nad celým souborem.
  - Např.: medián.

## 29.4 Statistické popisné charakteristiky

- Různými statistickými metodami se snažíme datům lépe porozumnět a získat o nich více informací.
- **Míry polohy** – Určují střed dat, případně další body z hlediska hodnot dat.
  - Aritmetický průměr (vážený průměr, citlivost na odlehlé hodnoty)
  - Geometrický průměr (n-tá odmocina ze součinu hodnot, kde n je počet vzorků). Typicky pro výpočet tempa růsta.
  - Harmonický průměr – Počet vzorků / suma převrácených hodnot.
  - Medián
  - Modus
  - Kvantily
    * Má 2 parametry $(k, q)$, $q$-kvantilů rozděluje uspořádaný datový soubor na $q$ přibližně stejně pravděpodobných intervalů (stejný počet výskytu).
    * Hodnoty kvantilů jsou hodnoty náhodné veličiny, tedy průměr v každém $q$ intervalu.
    * Pro $q = 100$ percentily, $q = 10$, decily, $q = 5$ kvintily, $q = 4$ kvartily.
- **Míry variability** – Určují rozptýlenost hodnot kolem středu.
  - Rozpětí – max-min, ovlivněno extrémy
  - Mezikvartilové rozpětí – rozpětí 50 % středních hodnot, např. pro kvartil, $q_3$ – $q_2$
  - Rozptyl

  - Směrodatná odchylka
- **Další**
  - šikmost, špičatost, ...

## 29.5 Vizualizační techniky

### 29.5.1 Vizualizační technika pro jeden atribut

- Cíl – ukázat rozložení hodnot atributu.
- Pro kvantitativní atributy – krabicový graf, graf hustoty pravděpodobnosti, houslový graf, kvantilový graf, ...
- Pro kategorické atributy – histogram.

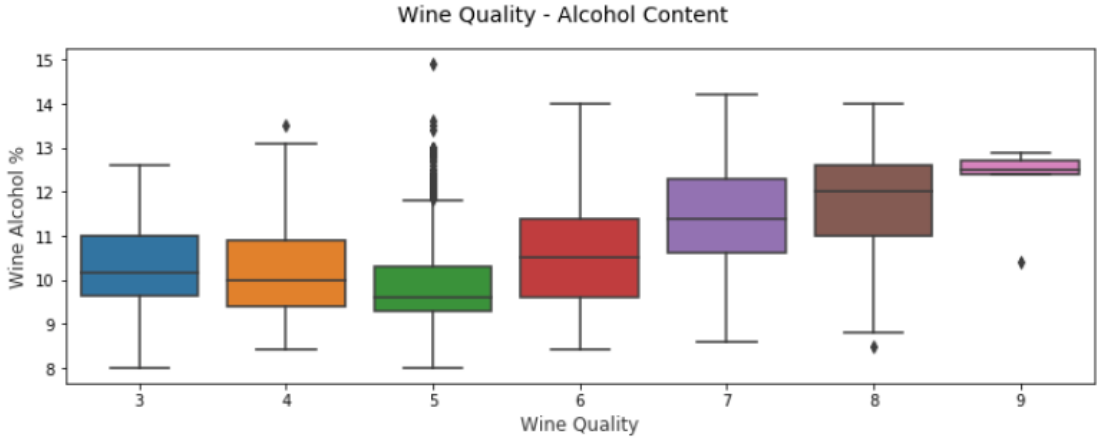
### 29.5.2 Vizualizační technika pro více atributů

- Cíl:
  - Porovnání rozložení hodnot.
  - Odhalení potenciálních vztahů mezi atributy (korelace).
- Porovnání rozložení hodnot dvou atributů:
  - krabicový / houslový graf,
  - histogram,
  - bodový graf, lokální regrese.
- Porovnání hodnot více atributů:
  - krabicový / houslový graf,
  - matice grafů,
  - rozšíření použití hloubky, barvy, velikosti, tvaru, ...,
  - systém paralelních souřadnic.

### 29.5.3 Konkrétní typy vizualizací

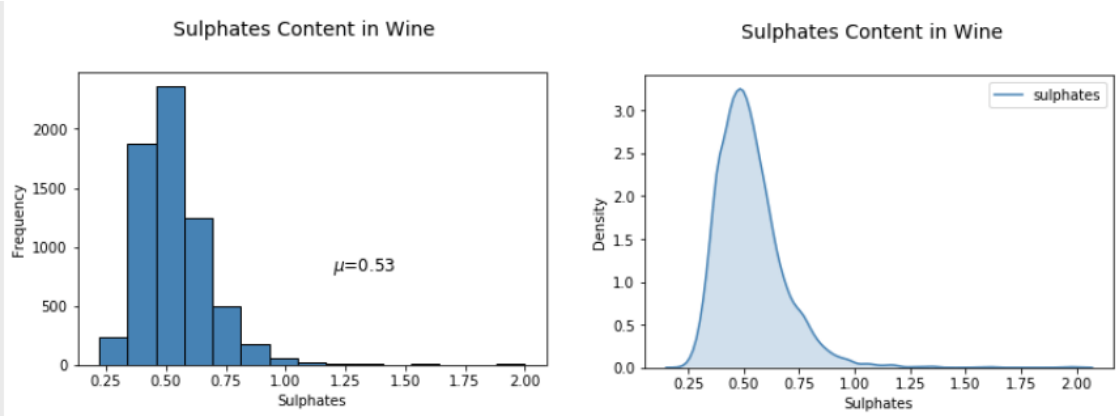
**Krabicový graf (box plot)**

- Rozdělení podle kvartilů.
- Zobrazen medián, odlehlé hodnoty (1. a 4. kvartil) a maximum i minimum.
- Na ose x máme kategorický atribut, na ose y kvantitativní atribut.




Obrázek 29.1: Krabicový graf.

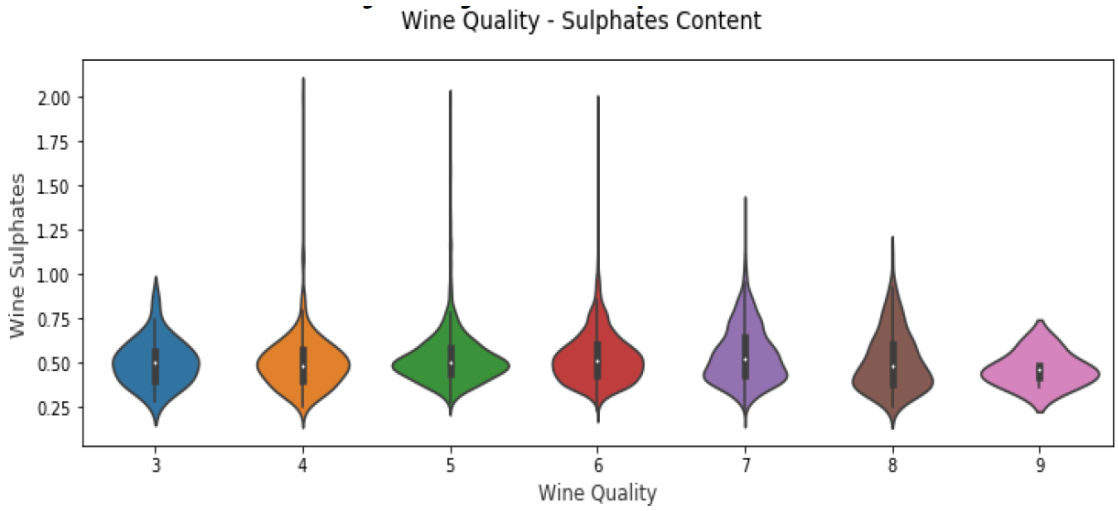
**Histogram, graf hustoty pravděpodobnosti**



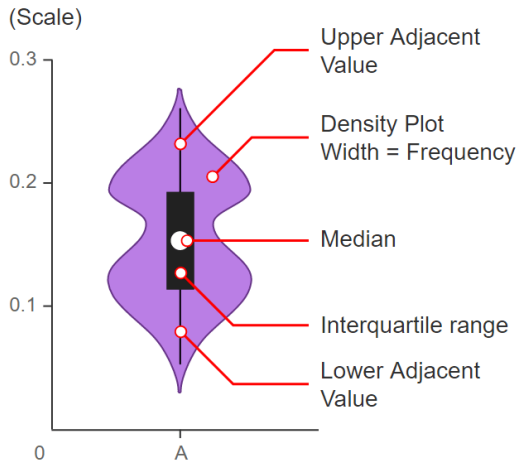

Obrázek 29.2: Histogram a graf hustoty pravděpodobnosti.

**Houslový graf**

- Kombinace krabicového grafu a grafu hustoty.
- Dokáže odhalit vícemodální data! (viz obrázek, více než jeden vrchol)



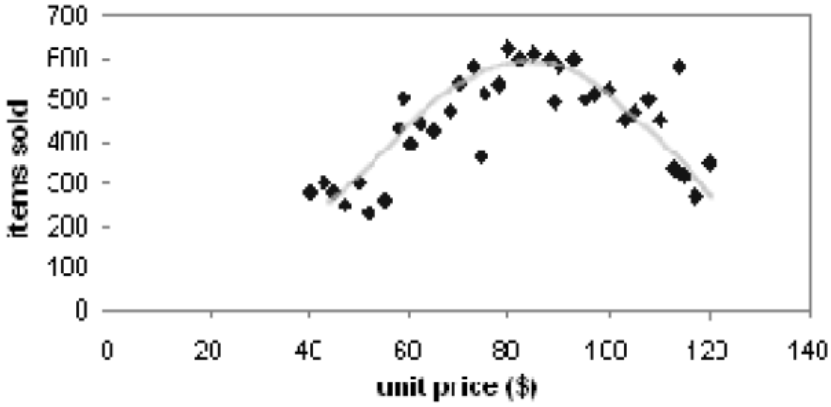

Obrázek 29.3: Houslový graf 1.




Obrázek 29.4: Houslový graf 2.

**Bodový graf**

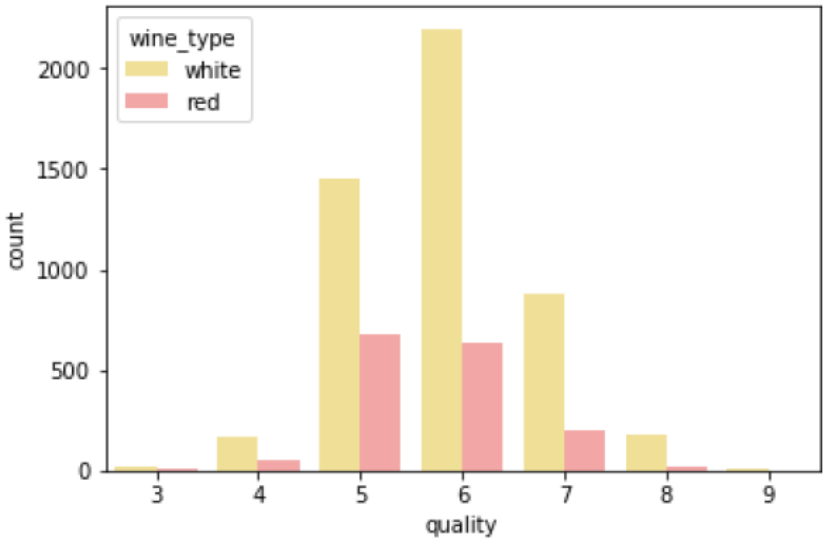
- Klasický, jednoduchý.
- Pro prvotní náhled na data, odhalí nejvýraznější rysy.
- Lze kombinovat s regresí.




Obrázek 29.5: Bodový graf.

**Sloupcový graf**

- Pro kategorické atributy.




Obrázek 29.6: Sloupcový graf.

## 29.6 Korelační analýza

- Korelace (*correlation*) popisuje vzájemný vztah mezi dvěma atributy (jak se vzájemně ovlivňují). Pokud se mezi dvěma atributy potvrdí korelace, je pravděpodobné, že jsou atributy na sobě závislé. Na základě toho však ještě nelze rozhodnout, zda jeden z nich je příčinou a druhý následkem (korelace neimplikuje kauzalitu).
- Míra korelace je vyjadřována pomocí korelačních koeficientů, které nabývají hodnot z intervalu $\langle -1, 1 \rangle$. Hodnota korelačního koeficientu $-1$ značí zcela nepřímou závislost (antikorelaci), tedy čím vyšších hodnot nabývá jeden atribut, tím nižších nabývá druhý. Hodnota korelačního koeficientu $+1$ značí zcela přímou závislost. Pokud je korelační koeficient roven 0 (nekorelovanost), pak mezi znaky není žádná statisticky

zjistitelná lineární závislost. Avšak i při nulovém korelačním koeficientu na sobě veličiny mohou záviset, pouze tento vztah nelze vyjádřit lineární funkcí, a to ani přibližně.

### 29.6.1 Číselné vyjádření

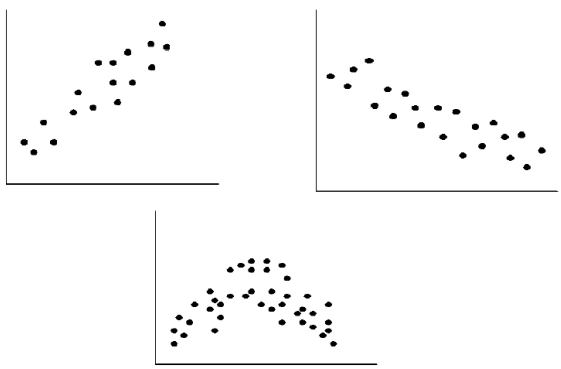
- Pro kvantitativní atributy:
  - **Pearsonův korelační koeficient**
    - Výpočet pomocí kovariace – průměrná hodnota součinu vzdáleností jednotlivých atributů od průměrné hodnoty.
    - A celé je ještě normalizované pomocí součinu směrodatných odchylek.
    - Výsledek je z intervalu $(-1, 1)$ a vyjadřují pozitivní (přímá úměra) a negativní korelaci (nepřímá úměra).

$$\rho_{X,Y} = \frac{\mathrm{cov}(X, Y)}{\sigma_X \sigma_Y} \tag{29.1}$$

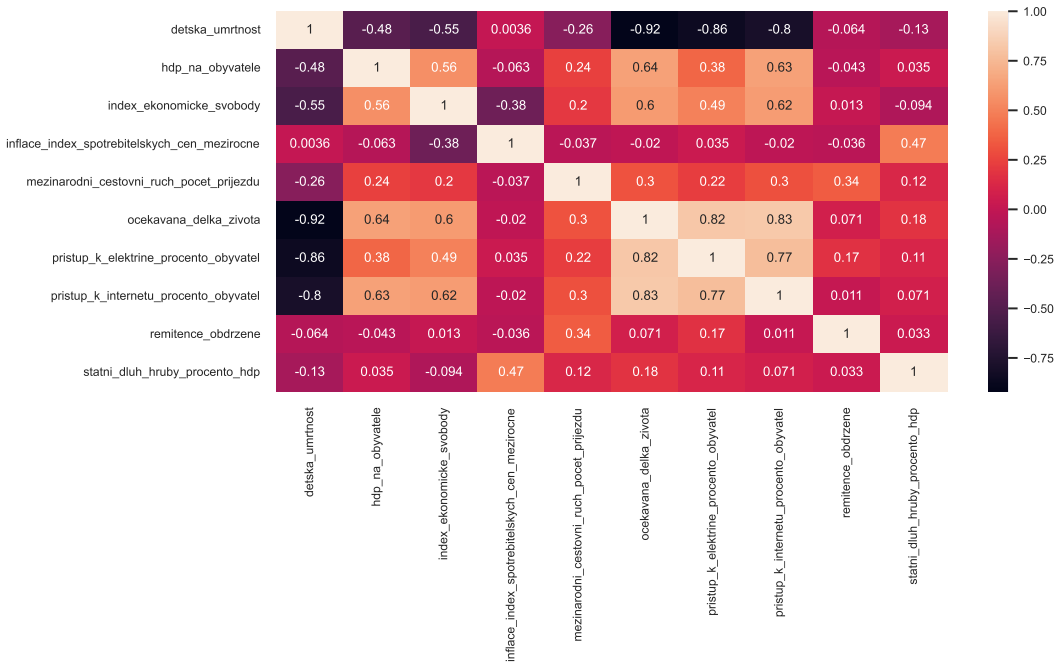
- Pro kategorické atributy:
  - **Test dobré shody**
    - Je stanovena nějaká nulová (bázová) hypotéza a testem dobré shody se buďto vyvrací nebo nevyvrací.
    - Rozdíl četností naměřených a očekávaných hodnot.
    - Očekávané četnosti spočítáme dle nějakého rozdělení.
    - Čím vyšší hodnota, tím vyšší pravděpodobnost, že jsou data korelovaná.

### 29.6.2 Vizuální techniky

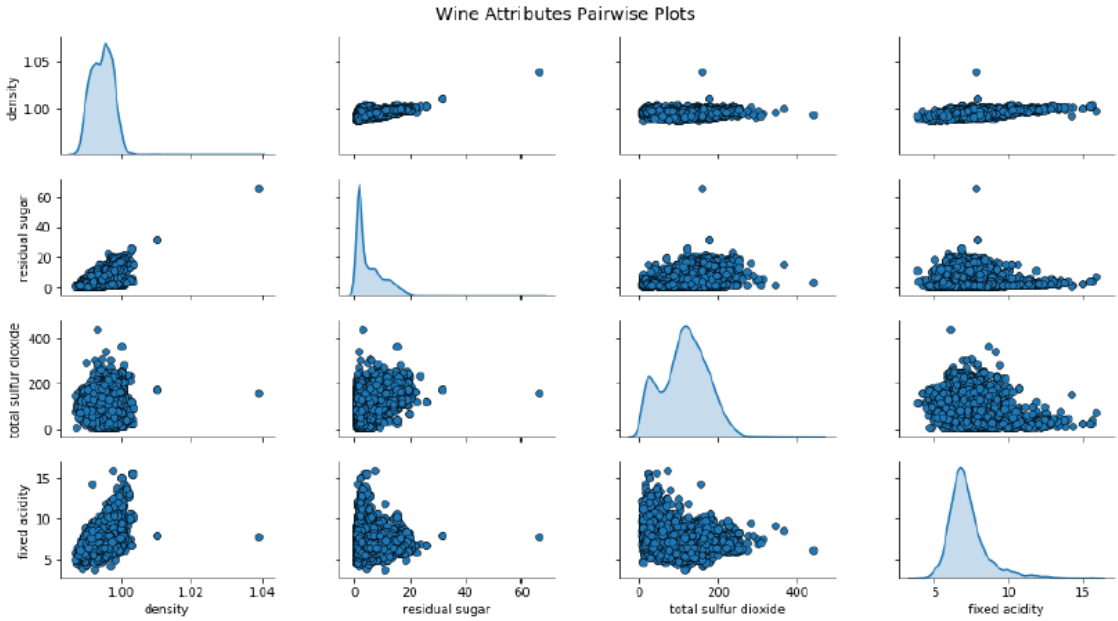
- Korelaci atributů vyhodnotíme na základě vizualizace, k tomu může sloužit např.
  - bodový graf,
  - matice korelačních koeficientů s tepelnou mapou,
  - matice bodových grafů.



Obrázek 29.7: Korelace v bodovém grafu.




Obrázek 29.8: Korelace v matici korelačních koeficientů s teplotní mapou.




Obrázek 29.9: Korelace v matici bodových grafů.

# Kapitola 30

# UPA – Prostorové DB (problematika mapování prostoru, ukládání, indexace; využití).

## 30.1 Zdroje

- `PDB-Spatial-CZ.pdf`
- `03-spatial_databases.pdf`
- `szz-kastak.pdf`
- `szz-discord-bazi.pdf`

## 30.2 Úvod a kontext

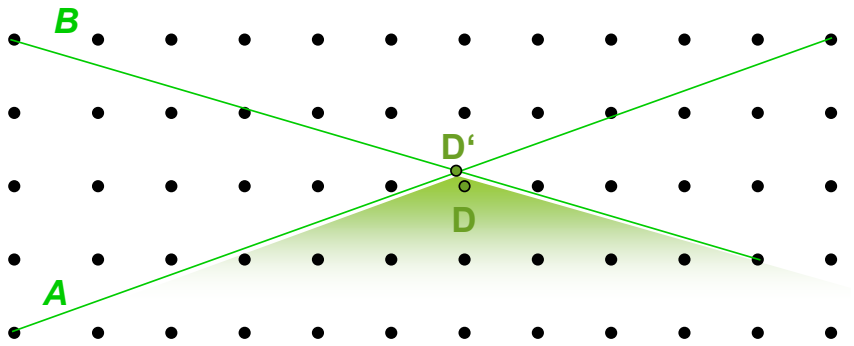
- Prostorové databáze podporují ukládání, zpracování a manipulace s prostorovými daty.
- Jsou případem pokročilého databázového systému – postrelačního databázového systému.
- Jsou schopné spravovat data, která se váží k určitému (libovolně velikému) prostoru. Mají schopnost spravovat velké množství relativně jednoduchých geometrických objektů.
- Jejich DDL (*data definition language*) a DML (*data manipulation language*) zahrnují prostorové datové typy. Ty jsou podpořeny i na implementační úrovni, takže je možné efektivně provádět operace indexace, vyhledávání, spojování (join), ...
- Jako uživatelé ukládáme geometrické (základní matematicko-geometrické tvary reprezentující např. města, řeky, domy, atomy, planety) a topologické údaje (vzájemné vztahy mezi ukládanými objekty, jako např. vzdálenost).
  - Popisujeme entity v prostoru a prostor jako takový.
  - V rámci geometrických údajů mluvíme o entitách jako bod, lomená čára, (vyplněný) polygon.

## 30.3 Problémy a jejich řešení

- Obvykle uvažujeme dvou až tří rozměrný euklidovský prostor, který je spojitý ve svých dimenzích a pozice každého bodu je definována uspořádanou uspořádanou $n$-ticí $p \in \mathbb{R}^n$.
- Pro vzdálenost bodů v $n$ rozměrném prostoru platí euklidovská metrika:

$$d(A, B) = \sqrt{(a_1 - b_1)^2 + \ldots + (a_n - b_n)^2}$$

- Problém, který nastává, je nemožnost počítače reprezentovat přesně reálná čísla – dochází k diskretizaci prostoru.
  - Tento problém vyvstává např. při výpočtu průsečíku, či sousednosti.

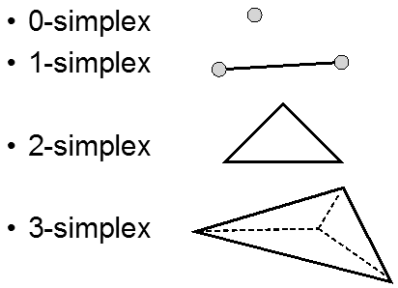



Obrázek 30.1: Problém diskretizace (spojitého) prostoru při reprezentaci průsečíku dvou úseček.

- Řešením je oddělení typů a operací nad prostorovými daty, tak aby byl korektně ošetřen tento číselný problém vzhledem ke geometrickému systému. Lze použít simplexy nebo úplné deskriptory (realms).

### 30.3.1 Simplexy

- Simplexy jsou nejmenší nevyplněné objekty daného rozměru.
- Použijeme jednoduché geometrické entity ke skládání složitějších celků.




Obrázek 30.2: Simplexy.

- Lze vypozorovat, že $d$-simplex sestává z $d + 1$ simplexů rozměru $d - 1$.
- Takové tvořící elementy se potom nazývají styky (faces).

- Kombinace simplexů do složitějších struktur je povolena jen tehdy, pokud průnik libovolných dvou simplexů je styk.
- Wikipedie: Simplex (či n-simplex) je n-rozměrným zobecněním trojúhelníku. Jedná se o konvexní obal množiny $n+1$ afinně nezávislých bodů umístěný v euklidovském prostoru dimenze $n$ či vyšší.

### 30.3.2 Úplné popisy (deskriptory, *realms*)

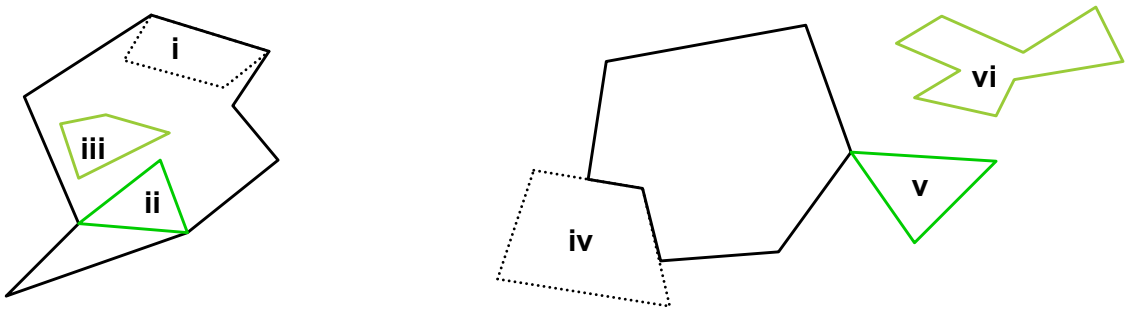
- Souhrnný popis všech objektů v databázi.
- Formálněji je to množina bodů, úseček, případně vyšších celků, nad sítí bodů, které mají tyto vlastnosti:
  - každý (koncový) bod je bodem sítě;
  - každý koncový bod usečky (složitějšího útvaru) je bodem sítě;
  - žádný vnitřní bod úsečky (složitějšího útvaru) není zaznamenán v síti;
  - žádné dvě úsečky (složitější útvary) nemají ani průsečík ani se nepřekrývají.

## 30.4 Mapování reality

- Při popisu reality musíme být schopni definovat vnořené objekty (např. plochu uvnitř plochy). To realizujeme pomocí R-cyklů a R-ploch.

### 30.4.1 R-cyklus

- R-cyklus je polygon (mnohoúhelník) uložený v diskrétní prostorové reprezentaci.
- Tj. uzavřená lomená úsečka, která je vytvořena podle pravidel ukládání deskriptorů.
  - Skládá se z posloupností $n$ úseček $s_1, \ldots, s_n$ a pro každou úsečku musí platit, že koncový bod úsečky $s_i$ je roven počátečnímu bodu úsečky $s_{(i+1) \bmod n}$.
  - Navíc žádné dvě úsečky $s_i$, $s_j$, kde $i, j \in 1, \ldots, n$ se nemohou protínat.




Obrázek 30.3: Různé míry vnoření polygonů (plošně – i, ii, iii, hranově – ii, iii, zcela/vrcholově – iii) a disjunkce polygonů (plošně – iv, v, vi, hranově – vi, v, zcela/vrcholově – vi).

### 30.4.2 R-plocha

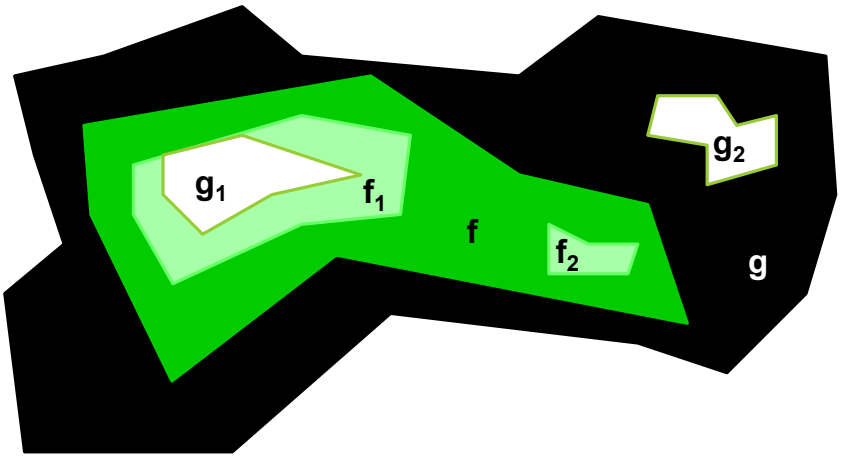
- Jak vložit několik poligonů do jednoho?

- Máme r-cyklus a uvnitř něho další r-cykly.

- R-plocha $f$ je dvojice $f = (c, H)$, kde je $c$ je R-cyklus a $H$ je množina R-cyklů, s jistými vlastnostmi:
  - $\forall h \in H$ je hranově vnořený v $c$;
  - $\forall h_i, h_j \in H : h_i \neq h_j$ platí, že $h_i$ a $h_j$ jsou hranově disjunktní;
  - není možné vytvořit žádný další R-cyklus ze segmentů popisujících R-plochu $f$.

### 30.4.3 Vnořená R-plocha

- Nechť $f = (f_0, F)$ a $g = (g_0, G)$ jsou dvě R-plochy. Řekneme, že $f$ je plošně vnořená v $g$ právě tehdy když:
  - $f_0$ je plošně vnořená v $g_0$ a zároveň;
  - $\forall g \in G : g$ je plošně disjunktní s $f_0$ a nebo $\exists f \in F : g$ je plošně vnořené v $f$.




Obrázek 30.4: R-plochy $f$ je vnořená v $g$.

### 30.4.4 Region

- Množina hranově disjunktní R-ploch.

### 30.4.5 Vnořený region

- Nechť $R, G$ jsou regiony, potom F je plošně vnořený v G právě tehdy když

$$\forall f \in F : \exists g \in G : f \text{ je plošně vnořená v } g$$

## 30.5 Operace nad prostorovými datami

- Číselné konstanty/charakteristiky (vypočítané při ukládání, tedy při operacích): délka, velikost, objem, plocha, průměr, obvod, ...
- Prostorové konstanty/charakteristiky (počítají se během ukládání, vytvářejí nové prostorové konstanty/charakteristiky) hodnotu: střed, těžiště, ...

- Metriky (číslo jako výsledek, nelze vypočítat předem): vzdálenost, průměr pro množinu prvků, ...
- Predikáty: stejné, uvnitř, protíná, sousedí, ...
- Vytvářejí se/vybírají se nové objekty: průsečík, rozdíl, voronoi, konvexní trup, ...

## 30.6 Ukládání dat

- Prostorové databáze si musí umět poradit s různorodou velikostí dat, kde datové položky se mohou pro hodnoty jednoho typu významně lišit a nabývat vysokých hodnot.
- Způsob uložení je konzistentní pro různou velikost dat.
  - Pro každá prostorová data tak dojde k vyčlenění dat, které se nemění s charakterem vkládaného objektu.
  - Ostatní data se ukládají mimo, do zvláštních datových stránek, které tak leží mimo a nebrání rychlému zpracování.
- Pokud prostorová data nepřekročí určitou velikost, tak jsou uložena stejně, jako ostatní. V rámci jedné stránky na disku/v paměti je možné mít více datových záznamů, jakmile však délka přeroste jistou mez, tak je uložen odkaz na souvislou oblast na disku, kde jsou velká data uložena za sebou.
- Velké data (cykly, plochy, regiony, ...) se neukládají jako běžná data – v tabulce je pouze ukazatel.
- V některých případech se volí předzpracování prostorových dat – ukládají se např. výsledky agregačních funkcí (průměr, suma, střed) nebo prvotní aproximace.
- Bývají uložené předpočítané hodnoty (délky, obsahy, objemy, ...) a aproximace. Např. průsečíky se nepočítají při dotazu, ale jsou předpočítané a uložené v prostorových objektech.

# Kapitola 31

# UPA – Indexace (nejen) v prostorových DB (kD-Tree a Grid File (a jejich varianty), R-Tree).

## 31.1 Zdroje

- `PDB-Spatial-CZ.pdf`
- `03-spatial_databases.pdf`
- `szz-kastak.pdf`
- `szz-discord-bazi.pdf`

## 31.2 Úvod a kontext

- Ve vícedimenzionálním prostoru nelze z reprezentace hodnot jednoznačně určit uspořádání – předchůdce a následníka. Nelze tedy použít obecně užívané indexační algoritmy pro 1D.
- Řešení v podobě mapování do 1D nám většinou nestačí, proto se v prostorových databázích využívají specializované indexační algoritmy.

## 31.3 Mapování do 1D

- Nejjednodušší řešení je namapovat data do jednorozměrného prostoru.
- Touto transformací však zcela jistě ztratíme sousednost.
- Krom toho, taková transformace nemusí být vůbec realizovatelná. A pokud ano, tak výsledek dotazů nemusí být transformovatelný zpět, takže lze výsledek jen těžko interpretovat.

## 31.4 Indexace bodů

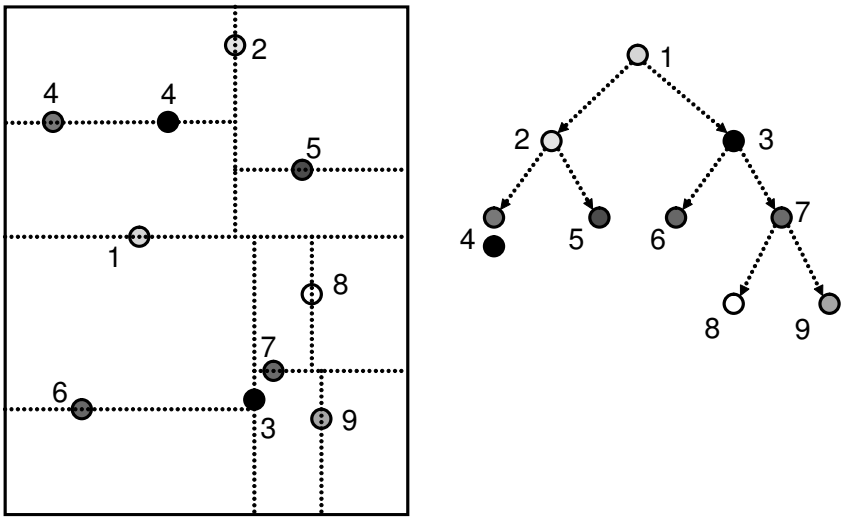
- Pro indexaci bodů využíváme 2 přístupy: stromy dělící prostor a hashování.

### 31.4.1 Stromy dělící prostor

- Myšlenka: vytvoříme uspořádání na množině bodů v prostoru (podobně jako ho máme přirozeně na množině čísel) a díky tomu, budeme moct uspořádat body do stromu.

**k-D Tree**

- k-D Tree je datová struktura (binární strom) dělící prostor hyperplochami na nejvyšší možné úrovni, a to vždy na dvě části.
  - Hyperplocha v rovině je přímka.
  - Dělící hyperplocha musí být rovnoběžná s osovým (souřadným) systémem.
  - Při dělení, musí ve výsledných plochách být obsažen vždy aspoň jeden bod, který pak už není součástí jakékoli jiné hyperplochy.
- Implementace stromu:
  - vytvoření stromu – pomocí rekurze;
  - vyhledávání – bez problému, cyklus;
  - vkládání – pomocí rekurze (pořadí vkládání bodů výrazně ovlivňuje vyváženost výsledného stromu);
  - mazání – velký problém, vyžaduje znovuvložené celého podstromu.
- Závěr: díky problematica mazání uzlů se tato struktura používá pouze pro pro statická data.



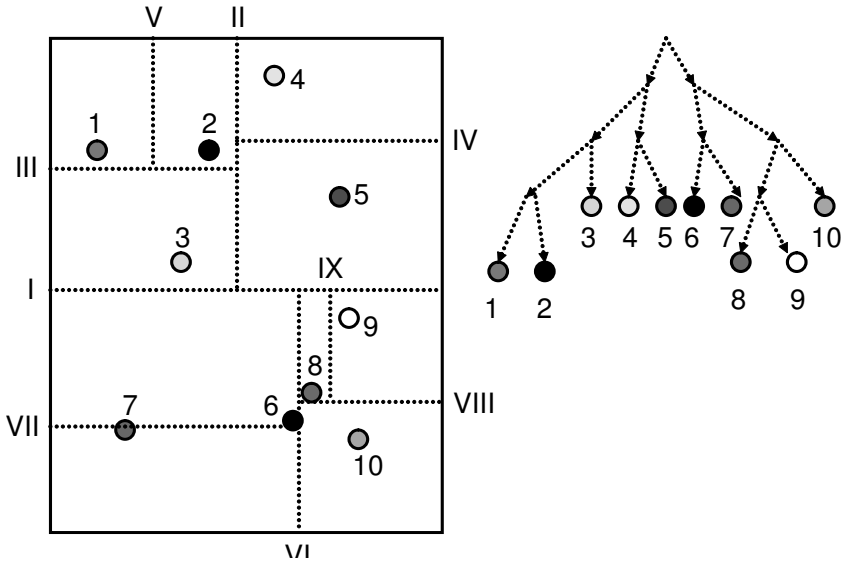

Obrázek 31.1: Příklad k-D Tree. Pořadí vkládání uzlů do indexačního stromu probíhalo podle číslovek u uzlu.

**Adaptivní k-D Tree**

- Snaží se eliminovat problémy spojené s pořadím vkládání bodů.
- Shodný s k-D Tree, až na to, že dělící hyperplochy
  - už nemusí obsahovat nějaký bod, stačí, že dělí prostor;

– jsou vybírány tak, aby výsledný prostor obsahoval přibližně stejný počet bodů.

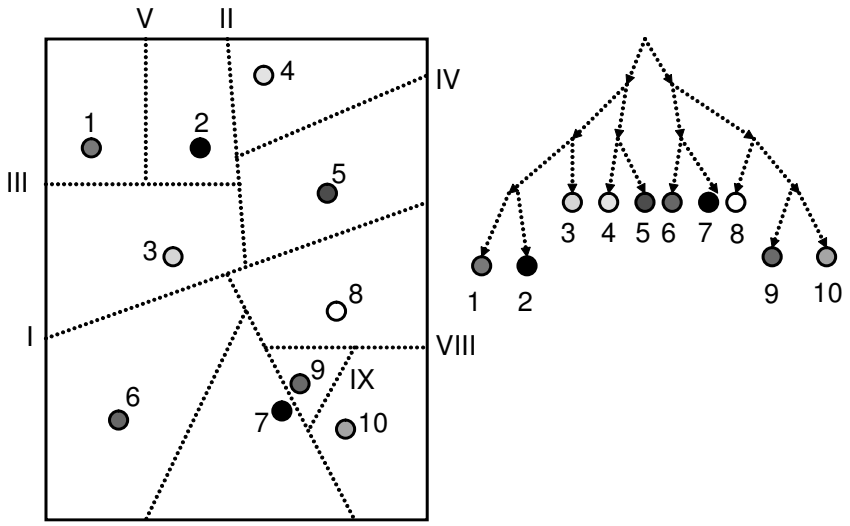
- Ze změn vyplývá, že data jsou uložena až v listových uzlech.
- Závěr: vylepšeno vkládání, ale mazání je stále velký problém, proto vhodnost stále spíše pro statická data.
  - *Proč je mazání problém, když body jsou uloženy v listech? Smažu bod a předpokládám, list může být i prázdný.*




Obrázek 31.2: Příklad Adaptivního k-D Tree. Římská čísla označují pořadí dělení prostoru.

**BSP (*binary space partitioning*) Tree**

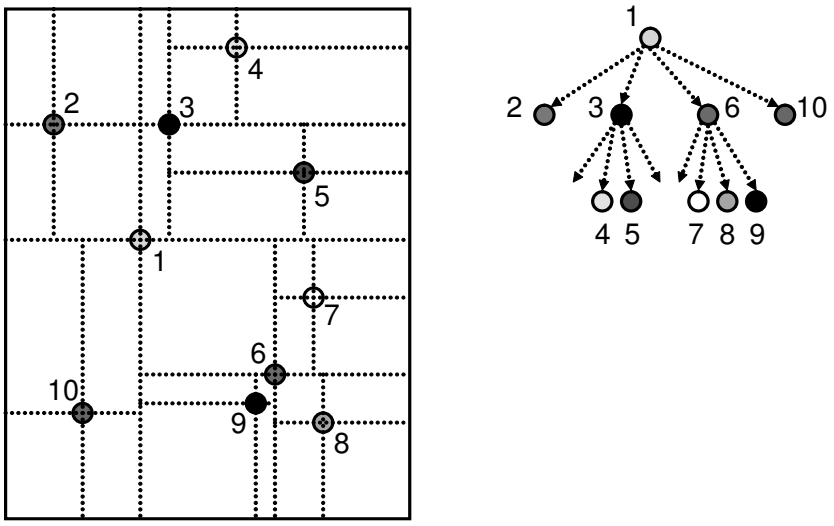
- Stejný jako Adaptivní k-D Tree až na to, že dělící hyperplochy nemusí být rovnoběžné s osovým (souřadným) systémem.
- Prostor se dělí do doby, než počet bodů v ploše neklesne pod určitou hodnotu (na obrázku 2).
- Algoritmus není adaptivní (odolný proti změně dat).
- Má vyšší nároky na paměť než k-D Tree, u kterého šlo díky pravidelnému střídání dělení ignorovat jednu souřadnici hyperplochy.




Obrázek 31.3: Příklad BSP Tree. Římská čísla označují pořadí dělení prostoru.

**Quad Tree**

- Algoritmus, který se plně shoduje s K-D Tree, avšak s rozdílem, že nedělí prostor na polovinu, ale na $2^n$ podstromů ($n$ je stupeň dimenze prostoru).
  - Díky tomu nemusejí být podstromy shodné, neboť některé listy nemusejí vést do míst, kde je obsažen nějaký bod.
- Prostor se dělí do doby, než počet bodů v ploše neklesne pod určitou hodnotu (na obrázku 2).
- Některé prostory neobsahují žádný bod.



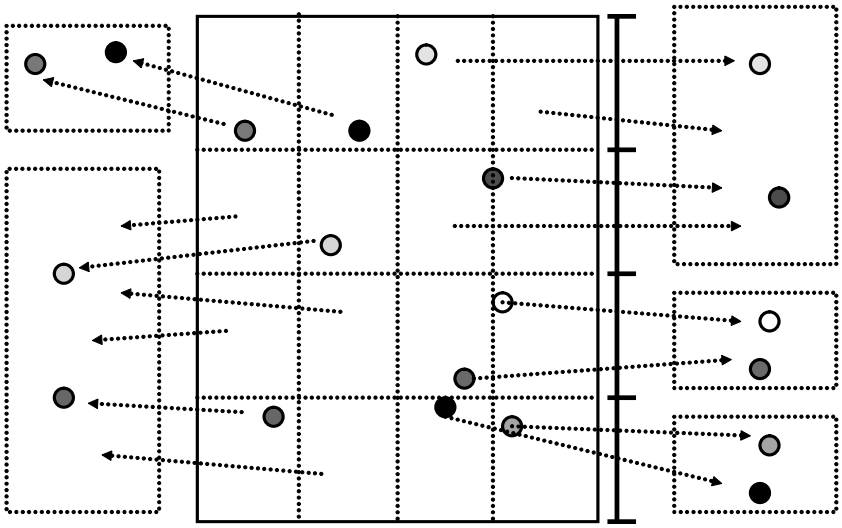

Obrázek 31.4: Příklad Quad Tree. Čísla u bodů označují pořadí, v jakém je prostor dělen hyperplochami na $2^n = 4$ podprostory a jak jsou vkládány údaje do stromu. Některé větve nemusejí obsahovat data, i když sousední je obsahují.

### 31.4.2 Hashování

- Tzv. adaptivní a lineární hashování [[?]].

#### Grid File

- Sledovaný úsek prostoru je rozdělen $n$-rozměrnou mřížkou (ne nutně pravidelnou) na buňky.
  - Buňky obsahují různý počet bodů, klidně 0.
- K tomuto základnímu rozdělení existuje adresář, který každou buňku přiřazuje k datové jednotce. (*bucket*).
  - Adresář je poměrně velký, proto je vždy ukládán na disk.
  - Mřížka je také uložena na disku, nicméne pro vyhledání datové jednotky stačí 2 přístupy na disk, což je snesitelné.
- Při vkládání může dojít k přetečení (datová jednotka není schopna pojmout další údaj), které není lokální.
  - Je nutné vložit rozdělující hyperplochu a tak zvětšit adresář.
  - Podobně se chová i mazání – jelikož není lokální, tak odstranění hyperplochy je třeba prověřit. Pokud je dat málo, tak může zaniknout i datová jednotka (data buďto zanikla s ní, nebo jsou přesunuta do jiné datové jednotky).

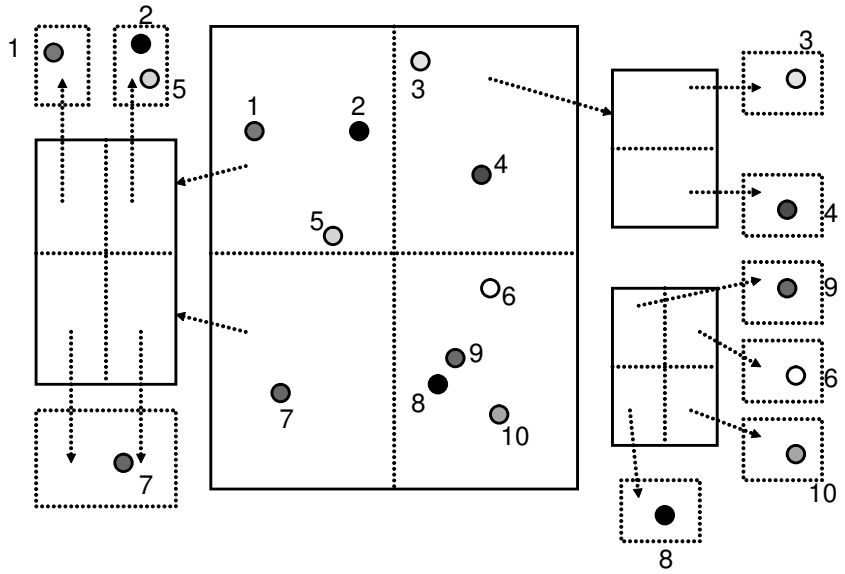


Obrázek 31.5: Příklad Grid File.

#### Two-level Grid File

- Je zavedena mřížka druhé úrovně.
  - Vzniká vztah kořenový adresář a podadresář.
  - První úroveň slouží pouze jako ukazatel do základní struktury Grid filu.
- V takovém systému jsou změny při vkládání či mazání často lokální, nicméně i tak není problematika přetečení úplně vyřešena.

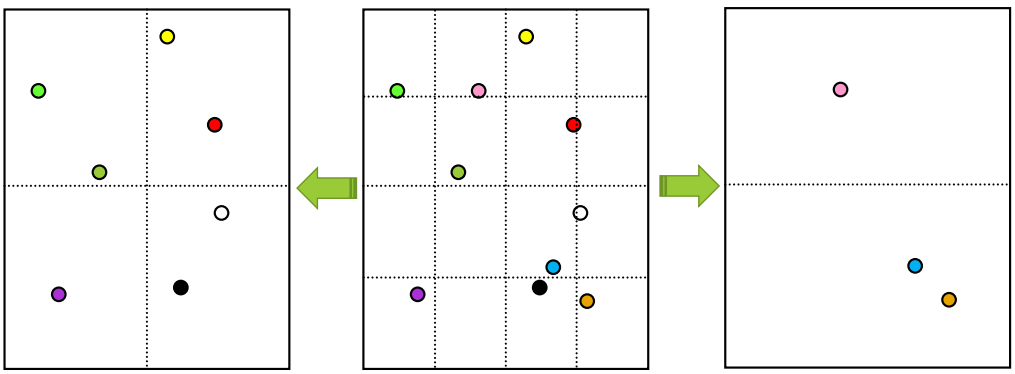
- Na nejvyšší úrovni dělí mřížka oblast na poměrně velké celky. Tyto velké celky jsou poté předmětem delení autonomní struktury Grid File. Jedná se tedy o dvě úrovně, kde ta první odkazuje pouze na základní strukturu Grid File a až ta odkazuje na datové jednotky.



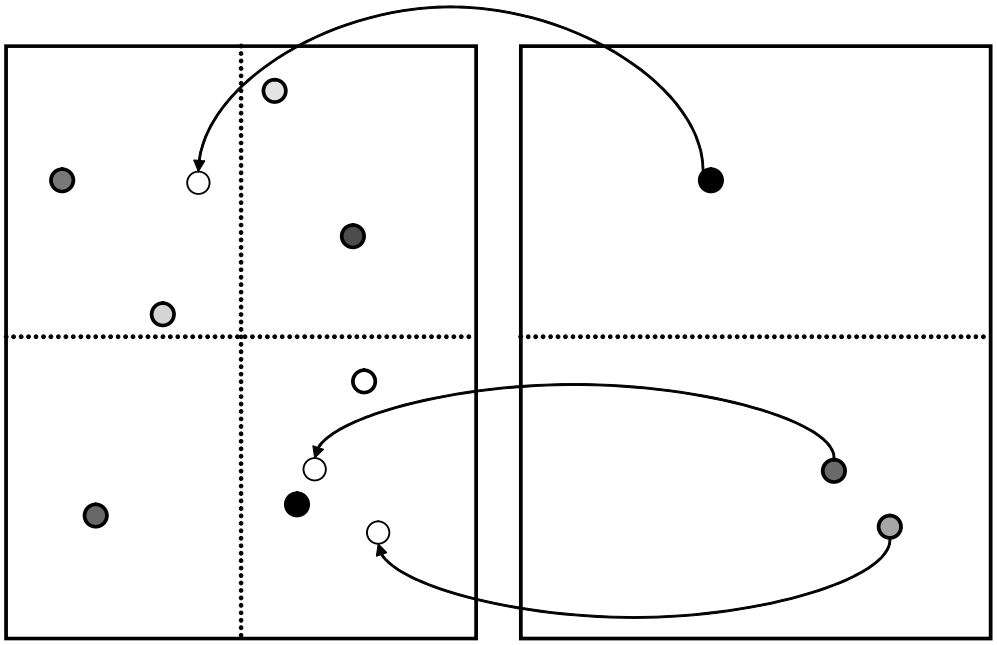

Obrázek 31.6: Příklad Two-level Grid File.

**Twin Grid File**

- Celá struktura Grid File se vyskytuje dvakrát.
  - Není však v hierarchickém (vertikálním) vztahu (jako u Two Level) ale v horizontálním.
  - I když je jedna ze struktur nadřazená, tak je jedno, která to bude.
- Cílem je maximálně využít prostor pro indexovací strukturu.
  - Data se mezi obě části dělí prakticky rovnoměrně.
  - Jedna struktura je však primární a druhá je sekundární (přetoková).
- Algoritmus umožňuje úplnou paralelizaci vyhledávání a částečnou pro vkládání.



Obrázek 31.7: Příklad Twin Grid File (1).



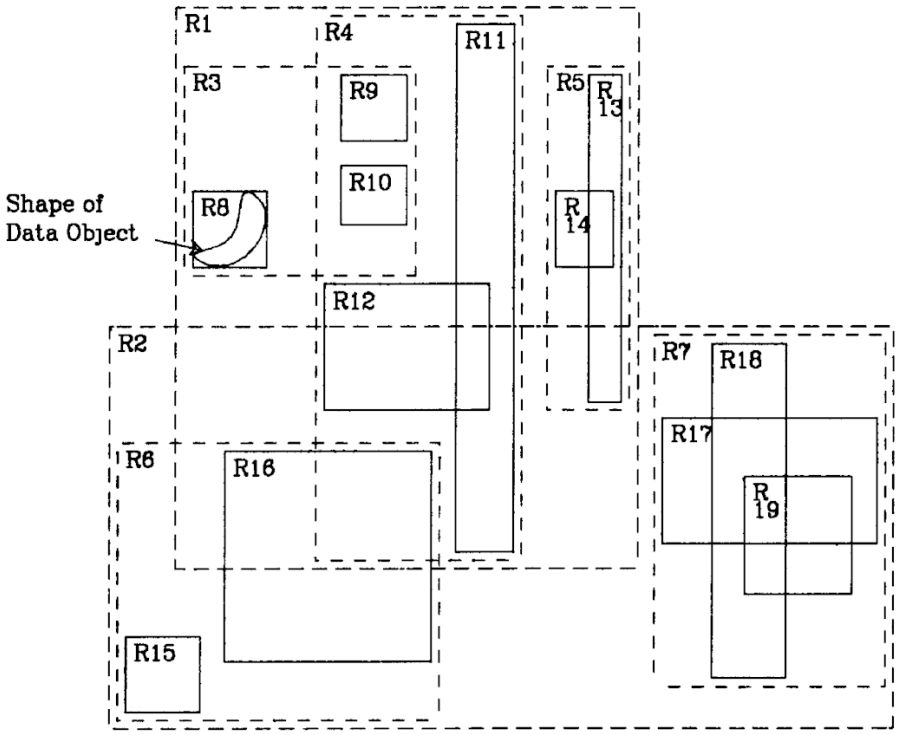
Obrázek 31.8: Příklad Twin Grid File (2). Na obrázku je ukázáno jak jsou do přetokové struktury odkládány body tak, aby nedošlo k přetečení ve struktuře primární. Šipky naznačují umístění bodů v prostoru. Do sekundární struktury (pravé) se body přesouvají tehdy, pokud by mělo dojít k rozdělení primární struktury a přitom úroveň sekundární struktury je menší, nebo je schopna bod pojmout bez dělení.

## 31.5 Indexace vícerozměrných objektů

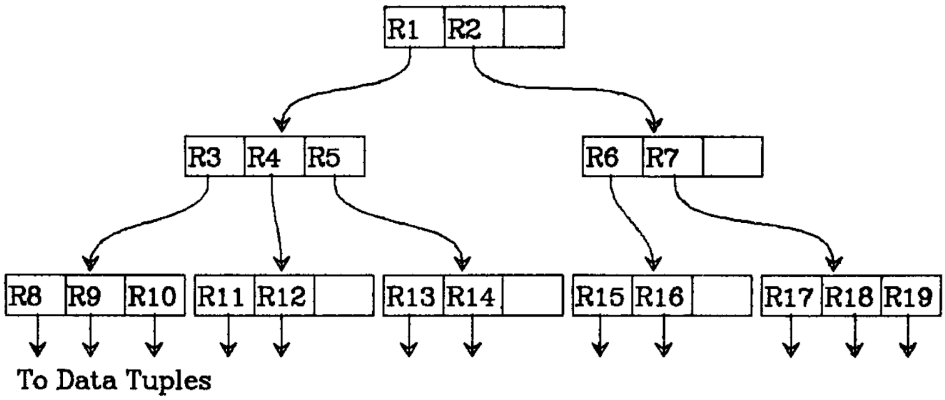
- Uložení bodů je sice primární, ale v obecné realitě s ním nelze vystačit, proto vznikly algoritmy pro indexaci vícerozměrných útvarů, které používají buďto překrývání (*overlapping*) a nebo ořezávání (*clipping*).

### 31.5.1 Překrývání – R-Trees

- Indexační struktury se překrývají, takže vzniká více vyhledávacích cest, implementačně se buňky překrývají svými hranicemi.
- V praxi tak dochází k tomu, že algoritmus je stejný, jen počet prohledávacích cest se zvětšuje, protože dopředu není jasné, ve které buňce je nakonec objekt uložen.



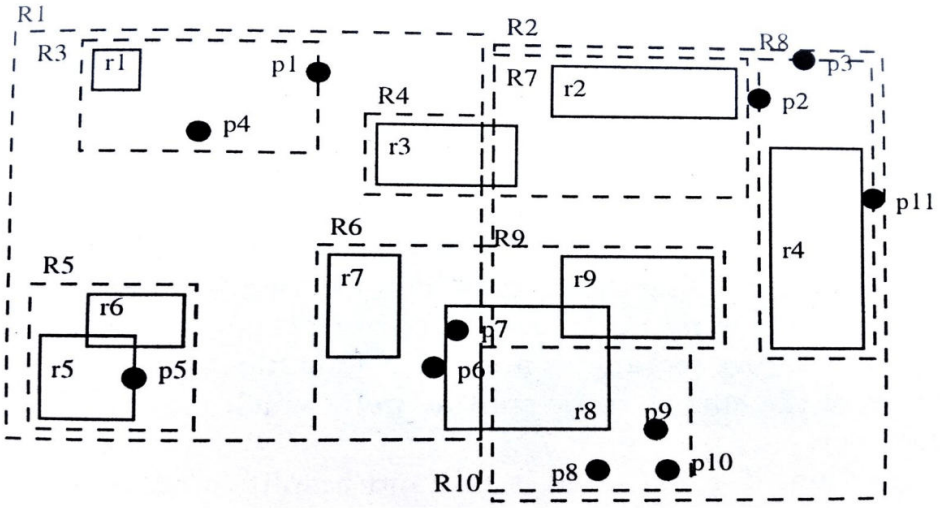

Obrázek 31.9: R-Tree příklad.



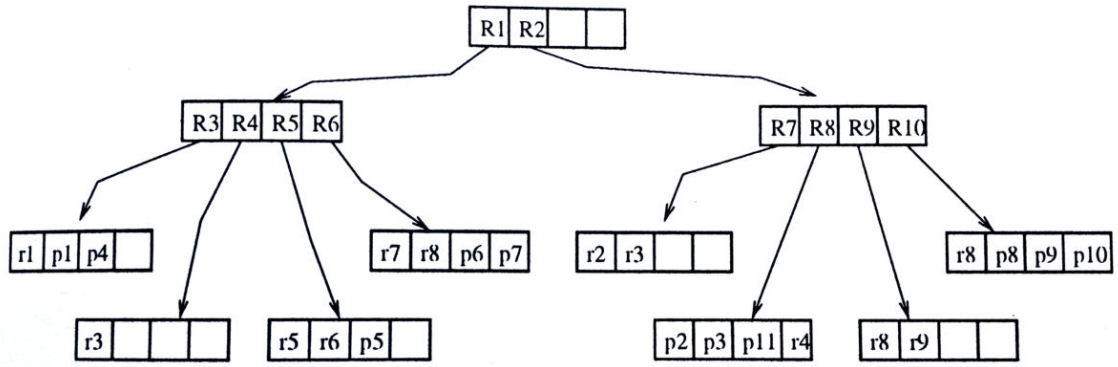

Obrázek 31.10: R-Tree příklad – index.

### 31.5.2 Ořezávání – R$^+$-Trees

- U metod založených na ořezávání není povoleno, aby se MBB (*minimal bounding box*), buňky překrývaly.
- Jediným řešením je rozsekat objekty tak, aby sledovaly hranice dělicích MBB. Jeden objekt tak může být rozdělen na více.
- Při vyhledávání pak tedy nestačí vyhledat objekt, ale je třeba se vrátit k jeho původní, nedělené reprezentaci.




Obrázek 31.11: $R^+$-Tree příklad.




Obrázek 31.12: $R^+$-Tree příklad – index.

# Kapitola 32

# FLP – Lambda kalkul (definice všech pojmů, operací, ...).

## 32.1 Zdroje

- `FLP-FP.pdf`
- `flp_lambda_kalkul.pdf`
- `FLP_2020-02-03.mp4`
- `https://petrzemek.net/publications`

## 32.2 Úvod a kontext

- Lambda kalkul je formální systém (jazyk) a výpočetní model používaný v teoretické informace a matematice pro studium funkcí a rekurze.
- Lambda kalkul je teoretickým základem funkcionálního programování a příslušných programovacích jazyků (Lisp, Haskell).
- Lambda kalkul je výpočetně ekvivalentní turingovým strojům (a parciálně rekurzivním funkcím) (Church-Turingova teze) – Jakýkoliv program může být vyjádřen v Lambda kalkulu.
- Funkce má ve funkcionálním programování (a tedy i v lambda kalkulu) význam matematický, tj. jedná se o čísté funkce (*pure functions*). Výsledek závisí pouze na vstupních parametrech, funkce nemá žádný interní stav.

## 32.3 Základy

- Intuitivně, identita vs rovnost.

**Proměnná** Proměnná jako z jiných programovacích jazyků.

**Abstrakce** Definice funkce.

$$\begin{aligned} \lambda\, x\, .\, x+2 &\quad \text{odpovídá} \quad f(x) = x+2 \\ \lambda\, x\, .\, \lambda\, y\, .\, x+y &\quad \text{odpovídá} \quad f(x,y) = x+y \\ \lambda\, x\, y\, .\, x-y &\quad \text{odpovídá} \quad f(x,y) = x-y \end{aligned} \tag{32.1}$$

**Aplikace** Volání funkce.

$$\begin{array}{rcl} (\lambda\ x\ .\ x+2)\ 5 & \text{odpovídá} & f(5) \\ (\lambda\ x\ y\ .\ x-y)\ 10\ 5 & \text{odpovídá} & f(10,5) \end{array} \tag{32.2}$$

**Výraz** Jako výraz, resp. $\lambda$-výraz, se označuje každý z výše uvedených třech prvků (proměnná, abstrakce, aplikace). Typicky se značí velkými písmeny.

**Volná a vázaná proměnná** Rozlišujeme volnou a vázanou proměnnou. Vázanná proměnná je proměnná, která je parametr funkce. Ostatní proměnné jsou volné.

$$\lambda\ vazana\ .\ vazana + volna \tag{32.3}$$

**Zjednodušování zápisů**

$$\begin{array}{rcl} (A\ B) & = & A\ B \\ (\lambda\ E\ .(((A)\ B)\ C))) & = & \lambda\ E\ .\ A\ B\ C \\ (\lambda\ A(\lambda\ B(\lambda\ C\ .\ x))) & = & \lambda\ A\ B\ C\ .\ x \end{array} \tag{32.4}$$

**Pojmenování výrazů**

LET $K = \lambda\ x\ .\ x+2$ poté K můžeme používat v jiných výrazech

## 32.4 Konverze (redukce)

- Upravování výrazů na jiné, často jednodušší, resp. jejich konverze (redukování).
- Konverze zapisujeme šipkou.

### 32.4.1 $\alpha$-konverze [alfa]

- Přejmenování proměnných
- Provádění má formu substituce: $[W/V]$ znamená, že $V$ je nahrazeno za $W$.
- Lze uplatnit pouze na vázané proměnné.
- Nesmí se změnit význam výrazu – vázaná proměnná se nesmí stát volnou a volná proměnná se nesmí stát vázanou.
- Příklad správného použití (substituce se bude muset dále vyhodnotit i ve výrazu $E$):

$$\lambda\ V\ .\ E \Rightarrow_\alpha \lambda\ W\ .\ E\ [W/V]$$

- Příklad chybného použití ($y$ se z volné stalo vázanou):

$$\lambda\ x\ .\ x\ y \Rightarrow_\alpha \lambda\ y\ .\ y\ y$$

### 32.4.2 $\beta$-konverze [beta]

- Aplikace funkce na argument.
- Provedení má opět formu substituce.
- Příklad správného použití (substituce se bude muset dále vyhodnotit i ve výrazu $E$):

$$(\lambda\ V\ .\ E)M \Rightarrow_\beta E\ [M/V]$$

- Příklad chybného použití ($y$ se z volné stalo vázanou):

$$(\lambda\ x\ y\ .\ x\ y)(x\ y) \Rightarrow_\beta \lambda\ y\ .(x\ y)\ y$$

### 32.4.3 $\eta$-konverze [eta]

- Vyjadřuje ekvivalenci výrazů a jejich převoditelnost.
- Příklad správného použití (podmínka platnosti: $V$ není volné v $E$):

$$\lambda\ V\ .\ E\ V \Rightarrow_\eta E$$

- Příklad chybného použití ($x$ je volné):

$$\lambda\ x\ .(x\ y)\ x \Rightarrow_\eta (x\ y)$$

## 32.5 Rekurze

### 32.5.1 Operátor pevného bodu

- Operátor pevného bodu značíme $Y$.

$$\text{LET}\ \ Y = \lambda\ f\ .(\lambda\ x\ .\ f\ (x\ x))(\lambda\ x\ .\ f\ (x\ x))$$

- V matematice jako pevný bod označujeme bod, který se v daném zobrazení zobrazí sám na sebe (např. $sin(0) = 0$).
- Pevný bod výrazu $E$: $E\ Y$.
- Nechť pro výraz $E$ je pevný bod $k_E = Y\ E$.
- Z definice vlastnosti: $E\ k_E = k_E = Y\ E = E\ (Y\ E)$.

### 32.5.2 Bottom

- Bottom značíme $\perp$ , definuje se pomocí operátoru pevného bodu.

$$\text{LET}\ \ \perp = Y\ (\lambda\ f\ x\ .\ f)$$

- Výraz, který bude neustále na výstup produkovat sebe (a zkonzumuje všechny parametry.)
- Tímto modelujeme do jisté míry nekonečnou smyčku v programu.
  - Lze využít pro signalizaci chybné hodnoty v programu, například dělení nulou.

### 32.5.3 Vytvoření rekurze

**Příklad: vytvoření rekurze**

$$
\begin{aligned}
Y\ E = \lambda\ f\ .(\lambda\ x\ .\ f\ (x\ x))(\lambda\ x\ .\ f\ (x\ x))\ E \Rightarrow_\beta \\
\Rightarrow_\beta (\lambda\ x\ .\ E\ (x\ x))(\lambda\ x\ .\ E\ (x\ x)) \Rightarrow_\beta \\
\Rightarrow_\beta E\ ((\lambda\ x\ .\ E\ (x\ x))\ (\lambda\ x\ .\ E\ (x\ x))) = \\
= E\ (Y\ E)
\end{aligned}
\tag{32.5}
$$

# Kapitola 33

# FLP – Práce v lambda kalkulu (demonstrace reprezentace čísel a pravdivostních hodnot a operací nad nimi).

## 33.1 Zdroje

- `FLP-FP.pdf`
- `flp_lambda_kalkul.pdf`
- `FLP_2020-02-03.mp4`

## 33.2 Logické operace

### 33.2.1 True, False, Not

$$\text{LET } True = \lambda\ x\ y\ .\ x$$

$$\text{LET } False = \lambda\ x\ y\ .\ y$$

$$\text{LET } Not = \lambda\ x\ .\ x\ False\ True$$

**Příklad:** $Not\ False$

$$\begin{aligned} Not\ False &= (\lambda\ x\ .\ x\ False\ True)\ False \Rightarrow_\beta \\ &\Rightarrow_\beta False\ False\ True = \\ &= (\lambda\ x\ y\ .\ y)\ False\ True \Rightarrow_\beta \\ &\Rightarrow_\beta True \end{aligned} \tag{33.1}$$

### 33.2.2 And

$$\text{LET } And = \lambda\ x\ y\ .\ x\ y\ False$$

**Příklad:** $And\ True\ True$

$$\begin{aligned} And\ True\ True &= (\lambda\ x\ y\ .\ x\ y\ False)\ True\ True \Rightarrow_\beta \\ &\Rightarrow_\beta True\ True\ False = \\ &= (\lambda\ x\ y\ .\ x)\ True\ False \Rightarrow_\beta \\ &\Rightarrow_\beta True \end{aligned} \tag{33.2}$$

**Příklad:** $And\ True\ False$

$$\begin{aligned} And\ True\ False &= (\lambda\ x\ y\ .\ x\ y\ False)\ True\ False \Rightarrow_\beta \\ &\Rightarrow_\beta True\ False\ False = \\ &= (\lambda\ x\ y\ .\ x)\ False\ False \Rightarrow_\beta \\ &\Rightarrow_\beta False \end{aligned} \tag{33.3}$$

**Příklad:** $And\ False\ True$

$$\begin{aligned} And\ False\ True &= (\lambda\ x\ y\ .\ x\ y\ False)\ False\ True \Rightarrow_\beta \\ &\Rightarrow_\beta False\ True\ False = \\ &= (\lambda\ x\ y\ .\ y)\ True\ False \Rightarrow_\beta \\ &\Rightarrow_\beta False \end{aligned} \tag{33.4}$$

**Příklad:** $And\ False\ False$

$$\begin{aligned} And\ False\ False &= (\lambda\ x\ y\ .\ x\ y\ False)\ False\ False \Rightarrow_\beta \\ &\Rightarrow_\beta False\ False\ False = \\ &= (\lambda\ x\ y\ .\ y)\ False\ False \Rightarrow_\beta \\ &\Rightarrow_\beta False \end{aligned} \tag{33.5}$$

### 33.2.3 Or

$$\text{LET}\ \ Or = \lambda\ x\ y\ .\ x\ True\ y$$

**Příklad:** $Or\ True\ True$

$$\begin{aligned} Or\ True\ True &= (\lambda\ x\ y\ .\ x\ True\ y)\ True\ True \Rightarrow_\beta \\ &\Rightarrow_\beta True\ True\ True = \\ &= (\lambda\ x\ y\ .\ x)\ True\ True \Rightarrow_\beta \\ &\Rightarrow_\beta True \end{aligned} \tag{33.6}$$

**Příklad:** $Or\ True\ False$

$$\begin{aligned} Or\ True\ False &= (\lambda\ x\ y\ .\ x\ True\ y)\ True\ False \Rightarrow_\beta \\ &\Rightarrow_\beta True\ True\ False = \\ &= (\lambda\ x\ y\ .\ x) True\ False \Rightarrow_\beta \\ &\Rightarrow_\beta True \end{aligned} \tag{33.7}$$

**Příklad:** $Or\ False\ False$

$$\begin{aligned} Or\ False\ False &= (\lambda\ x\ y\ .\ x\ True\ y)\ False\ False \Rightarrow_\beta \\ &\Rightarrow_\beta False\ True\ False = \\ &= (\lambda\ x\ y\ .\ y) False\ False \Rightarrow_\beta \\ &\Rightarrow_\beta False \end{aligned} \tag{33.8}$$

### 33.2.4 Xor

$$\text{LET}\ \ Xor = \lambda\ x\ y\ .\ x\ (Not\ y)\ y$$

### 33.2.5 Nand

$$\text{LET}\ \ Nand = \lambda\ x\ y\ .\ x\ (Not\ y)\ True$$

### 33.2.6 Implikace

$$\text{LET}\ \ Implication = \lambda\ x\ y\ .\ x\ y\ True$$

### 33.2.7 Ekvivalence

$$\text{LET}\ \ Equivalence = \lambda\ x\ y\ .\ x\ y\ (Not\ y)$$

## 33.3 Aritmetika

### 33.3.1 Reprezentace čísel

$$\begin{aligned} &\text{LET}\ \ 0 = \lambda\ x\ y\ .\ y \\ &\text{LET}\ \ 1 = \lambda\ x\ y\ .\ x\ y \\ &\text{LET}\ \ 2 = \lambda\ x\ y\ .\ x\ (x\ y) = \lambda\ x\ y\ .\ x^2\ y \\ &\text{LET}\ \ 3 = \lambda\ x\ y\ .\ x\ (x\ (x\ y)) = \lambda\ x\ y\ .\ x^3\ y \\ &\text{LET}\ \ n = \lambda\ x\ y\ .\ x\ (\ldots\ (x\ y) \ldots) = \lambda\ x\ y\ .\ x^n\ y \end{aligned} \tag{33.9}$$

### 33.3.2 Následník

$$\text{LET}\ \ Succ = \lambda\ a\ b\ c\ .\ a\ b\ (b\ c)$$

**Příklad:** $Succ\ 0$

$$\begin{aligned} Succ\ 0 &= (\lambda\ a\ b\ c\ .\ a\ b\ (b\ c))\ 0 \Rightarrow_\beta \\ &\Rightarrow_\beta \lambda\ b\ c\ .\ 0\ b\ (b\ c) = \\ &= \lambda\ b\ c\ .(\lambda\ x\ y\ .\ y)\ b\ (b\ c) \Rightarrow_\beta \\ &\Rightarrow_\beta \lambda\ b\ c\ .(\lambda\ y\ .\ y)\ (b\ c) \Rightarrow_\beta \\ &\Rightarrow_\beta \lambda\ b\ c\ .\ b\ c \Rightarrow_\alpha \\ &\Rightarrow_\alpha \lambda\ x\ y\ .\ x\ y = \\ &= 0 \end{aligned} \tag{33.10}$$

**Příklad:** $Succ\ 2$

$$\begin{aligned}
Succ\ 2 &= (\lambda\ a\ b\ c\ .\ a\ b\ (b\ c))\ 2 \Rightarrow_\beta \\
&\Rightarrow_\beta \lambda\ b\ c\ .\ 2\ b\ (b\ c) = \\
&= \lambda\ b\ c\ .(\lambda\ x\ y\ .\ x\ (x\ y))\ b\ (b\ c) \Rightarrow_\beta \\
&\Rightarrow_\beta \lambda\ b\ c\ .(\lambda\ y\ .\ b\ (b\ y))\ (b\ c) \Rightarrow_\beta \\
&\Rightarrow_\beta \lambda\ b\ c\ .(b\ (b\ (b\ c))) = \\
&= \lambda\ b\ c\ .\ b^3\ c \Rightarrow_\alpha \\
&= \lambda\ x\ y\ .\ x^3\ y = \\
&= 3
\end{aligned} \tag{33.11}$$

### 33.3.3 Test na 0

$$\text{LET}\ \ IsZero = \lambda\ a\ .\ a\ (\lambda\ b\ .\ False)\ True$$

**Příklad:** $IsZero\ 0$

$$\begin{aligned}
IsZero\ 0 &= (\lambda\ a\ .\ a\ (\lambda\ b\ .\ False)\ True)\ 0 \Rightarrow_\beta \\
&\Rightarrow_\beta 0\ (\lambda\ b\ .\ False)\ True = \\
&= (\lambda\ x\ y\ .\ y)\ (\lambda\ b\ .\ False)\ True \Rightarrow_\beta \\
&\Rightarrow_\beta True
\end{aligned} \tag{33.12}$$

**Příklad:** $IsZero\ 3$

$$\begin{aligned}
IsZero\ 3 &= (\lambda\ a\ .\ a\ (\lambda\ b\ .\ False)\ True)\ 3 \Rightarrow_\beta \\
&\Rightarrow_\beta 3\ (\lambda\ b\ .\ False)\ True = \\
&= (\lambda\ x\ y\ .\ x\ (x\ (x\ y)))\ (\lambda\ b\ .\ False)\ True \Rightarrow_\beta \\
&\Rightarrow_\beta (\lambda\ b\ .\ False)\ ((\lambda\ b\ .\ False)\ ((\lambda\ b\ .\ False)\ True)) \Rightarrow_\beta \\
&\Rightarrow_\beta False
\end{aligned} \tag{33.13}$$

### 33.3.4 Sčítání

- Myšlenka: $f$-krát se provede následník $g$.

$$\text{LET}\ \ Add = \lambda\ f\ g\ .\ f\ Succ\ g$$

**Příklad:** $Add\ 3\ 2$

$$\begin{aligned}
Add\ 3\ 2 &= (\lambda\ f\ g\ .\ f\ Succ\ g)\ 3\ 2 \Rightarrow_\beta \\
&\Rightarrow_\beta 3\ Succ\ 2 = \\
&= \lambda\ x\ y\ .\ x\ (x\ (x\ y))\ Succ\ 2 \Rightarrow_\beta \\
&\Rightarrow_\beta Succ\ (Succ\ (Succ\ 2)) \Rightarrow_\beta \\
&\Rightarrow_\beta Succ\ (Succ\ 3) \Rightarrow_\beta \\
&\Rightarrow_\beta Succ\ 4 \Rightarrow_\beta \\
&\Rightarrow_\beta 5
\end{aligned} \tag{33.14}$$

### 33.3.5 Předchůdce

- *Implementace předchůdce je komplikovaná, předpokládám, že se nebude zkoušet.*

$$\text{LET } \; Prev = \lambda \ldots$$

### 33.3.6 Odečítání

- Myšlenka: $g$-krát se provede předchůdce $f$.

$$\text{LET } \; Sub = \lambda \; f \; g \; . \; g \; Prev \; f$$

**Příklad:** $Sub \; 3 \; 2$

$$\begin{aligned} Sub \; 3 \; 2 &= (\lambda \; f \; g \; . \; g \; Prev \; f) \; 3 \; 2 \Rightarrow_\beta \\ &\Rightarrow_\beta 2 \; Prev \; 3 = \\ &= (\lambda \; x \; y \; . \; x \; (x \; y)) \; Prev \; 3 \Rightarrow_\beta \\ &\Rightarrow_\beta Prev \; (Prev \; 3) \Rightarrow_\beta \\ &\Rightarrow_\beta = Prev \; 2 \Rightarrow_\beta \\ &\Rightarrow_\beta = 1 \end{aligned} \tag{33.15}$$

# Kapitola 34

# FLP – Haskell – lazy evaluation (typy v jazyce včetně akcí, uživatelské typy, význam typových tříd, demonstrace lazy evaluation).

## 34.1 Zdroje

- `FLP-FP.pdf`
- `flp_lambda_kalkul.pdf`
- `FLP_2020-02-03.mp4`

## 34.2 Úvod a kontext

- [[**todo**]]

# Kapitola 35

# FLP – Prolog – způsob vyhodnocení (základní princip, unifikace, chování vestavěných predikátů, operátor řezu – vhodné a nevhodné užití).

## 35.1 Zdroje

- `todo`

## 35.2 Úvod a kontext

- Prolog je logický, deklarativní programovací jazyk.
- Každý program se skládá z množiny Hornových klauzulí.
- Spuštění programu a předání vstupů odpovídá dotazu, které spočívají v predikátech, které chceme ověřit (zda jsou pravdivé).
- Hornova klauzule je speciální druh klauzilí ve formě disjunkce literálů, která obsahuje maximálně jeden pozitivní literál, ostatní jsou negované. Tento zápis lze převést na implikaci.
- Vyhodnocení.
- Unifikace.
- Operátor řezu.

# Kapitola 36

# FLP – Prolog – změna DB/programu za běhu (demonstrace na prohledávání stavového prostoru, práce se seznamy).

## 36.1 Zdroje

- `todo`

## 36.2 Úvod a kontext

- [[todo]]

# Kapitola 37

# PRL – Model PRAM, suma prefixů a její aplikace.

## 37.1 Zdroje

- `PRL_06_PRAM_MNG.pdf`

## 37.2 Analýza algoritmů v paralelním prostředí

- Počet procesorů potřebných k řešení úlohy v závislosti na velikosti instance $n$.

$$p(n)$$

- Čas potřebný k vyřešení úlohy v krocích.

$$t(n)$$

- Cena paralelního řešení – Jaké množství práce je potřebné na vyřešení problému.

$$c(n) = p(n) \cdot t(n)$$

- Optimální cena – Taková cena, která je rovna optimálnímu sekvenčnímu algoritmu.

$$c(n) = t_{seq}(n)$$

- Zrychlení oproti sekvenčnímu algoritmu.

$$\frac{t_{seq}(n)}{t(n)}$$

- Efektivnost.

$$\frac{t_{seq}(n)}{c(n)}$$

## 37.3 PRAM

- Parallel Random Access Machine
- Broadcast
- **[[todo]]**

## 37.4 Suma prefixů

- Varianty: Pre-scan, Scan (all-sums), Reduce
- **[[todo]]**

## 37.5 Aplikace sumy prefixů

- Packing problem
- Problém viditelnosti
- CLA (Carry Look Ahead Parallel Binary Adder)
- Radix Sort
- Quick Sort
- **[[todo]]**

# Kapitola 38

# PRL – Distribuované a paralelní algoritmy - algoritmy nad seznamy, stromy a grafy.

## 38.1 Zdroje

- `PRL_07_List_MNG.pdf`

## 38.2 Algoritmy nad seznamy

- Predecessor computing
- List ranking
- Parallel suffix sum
- List ranking revisited
- Random mating
- List ranking optimal

## 38.3 Algoritmy nad stromy a grafy

- Euler tour traversal
- Spočítání pozice každé hrany
- Nalezení rodičů
- Přiřazení pořadí preorder vrcholům
- Počet následníků vrcholů
- Výpočet úrovně vrcholu
- Barvení

# Kapitola 39

# PRL – Interakce mezi procesy a typické problémy paralelismu (synchronizační a komunikační mechanismy).

## 39.1 Zdroje

- `PRL_09_SynchSareMem_slidyA.pdf`
- `PRL_09_SynchSareMem_slidyB.pdf`
- `PRL_10_Modely_slidy.pdf`

## 39.2 Úvod a kontext

- Interakce mezi procesy
  - **Soupeření** (*competition*)
    * Dva a více procesů se snaží v jeden okamžik přistoupit k nějakému sdílenému zdroji.
    * Je třeba synchronizace, ke zdroji může přistoupit pouze jeden proces, ostatní musí čekat.
    * Problémy: čtenáři a písaři, problém večeřících filosofů
  - **Kooperace** (*cooperation*)
    * Dva a více procesů na něčem spolupracují a nebo se na něčem musí dohodnout.
    * Problémy: producenti a konzumenti
- **Kritická sekce**
  - Část programu, ve které se pracuje se sdíleným prostředkem.
  - Je nutné zaručit, pokud procesy přistupují ke sdíleným zdrojům (místo ve sdílené paměti), musí být tento přístup výlučný.
- **Uváznutí** (*deadlock*)
  - Uváznutí je situace, kdy proces(y) čekají na událost, která nemůže nastat.

- Příklad: nechť $p_1$ a $p_2$ jsou procesy a $r_1$ a $r_2$ sdílené prostředky. Proces $p_1$ disponuje $r_1$ a zároveň požaduje (čeká na) $r_2$. Proces $p_2$ disponuje $r_2$ a zároveň požaduje (čeká na) $r_1$.
- Může existovat uzavřená smyčka takovýchto procesů.

- **Vyhladovění** (*starvation*)
  - Proces(y) se nemůže dostat ke sdílenému zdroji.
  - Příklad: nechť $p_1, p_2$ a $p_3$ jsou procesy a $r$ je sdílené prostředek. Všechny procesy pracují s $r$. Procesy $p_1$ a $p_2$ se vzájemně střídají při práci s $r$, ale $p_3$ se k $r$ nikdy nedostane.
- **Souběh** (*race condition*)
  - Procesy soupeří o sdílený zdroj, výsledek jejich operací je nepředvídatelný, jelikož může nastat nesprávné pořadí nebo načasování.
  - Příklad: nechť $p_1$ a $p_2$ jsou procesy a $r$ je sdílený prostředek. Oba procesy chtějí hodnotu ve sdílené proměnné inkrementovat. Po zápisu obou procesů, může proměnná $r$ nabývat dvou různých hodnot $r_i \in \{r_{i-1} + 1, r_{i-1} + 2\}$.

## 39.3 Problémy

- Typické problémy práce se sdílenou pamětí.

### 39.3.1 Producenti a konzumenti

- Mějme konečnou vyrovnávací paměť (buffer) a 2 procesy, které vykonávají jisté operace.
  - Producent, který vykonává operaci zápisu do bufferu (produkuje informace).
  - Konzument, který vykonává operace čtení z bufferu (konzumuje informace).
- Operace posouvají ukazatele (`in`, `out`), které ukazují na místa, kde se bude číst, resp. zapisovat.
- Problémy:
  - Soubežná činnost producenta a konzumenta.
  - Co se stane, když konzument nemá co konzumovat?
  - Co se stane, když producent chce produkovat a buffer je plný?

### Implementace s aktivním čekáním

```
int n = SIZE, in = 0, out = 0, counter = 0;
char buffer[n];

void producent() {
    while (1) {
        char item = produce(); // produce new item
        while (counter == n); // wait if the buffer is full
        buffer[in] = item;
        in = (in + 1) % n;
        counter += 1;
    }
}

void consument() {
    while (1) {
        while (counter == 0); // wait until the buffer is not empty
        char item = buffer[out]; // take item
        out = (out + 1) % n;
        counter -= 1;
        consume(item); // consume item
    }
}
```

Výpis 39.1: Implementace s aktivním čekáním.

### Implementace s využitím semaforů

- Semafor S – zajištění výhradního přístupu k bufferu.
- Semafor N – určený pro synchronizování procesů na čísle $N = in - out$, což vyjadřuje počet prvků v bufferu.

```
semaphore S, N;
int n = SIZE, in = 0, out = 0;
char buffer[n];
N.count = 0;
S.count = 1;

void producent() {
    while (1) {
        char item = produce(); // produce new item
        S.lock();
        buffer[in] = item;
        in += 1;
        S.unlock();
        N.unlock();
    }
}

void consument() {
    while (1) {
        N.lock();
        S.lock();
        char item = buffer[out]; // take item
        out += 1;
        S.unlock()
        consume(item); // consume item
    }
}
```

Výpis 39.2: Implementace s využitím semaforů.

**Implementace s využitím semaforů a cyklického bufferu**

- Semafor S – zajištění výhradního přístupu k bufferu.
- Semafor N – určený pro synchronizování procesů na čísle $N = in - out$, což vyjadřuje počet prvků v bufferu.
- Semafor E – určený pro synchronizování procesů na počtu volných míst v bufferu.

```
semaphore S, N, E;
int n = SIZE, in = 0, out = 0;
char buffer[n];
N.count = 0;
S.count = 1;
E.count = n;

void producent() {
    while (1) {
        char item = produce(); // produce new item
        S.lock();
        buffer[in] = item;
        in += 1;
        S.unlock();
        N.unlock();
    }
}

void consument() {
    while (1) {
        N.lock();
        S.lock();
        char item = buffer[out]; // take item
        out += 1;
        S.unlock()
        consume(item); // consume item
    }
}
```

Výpis 39.3: Implementace s využitím semaforů a cyklického bufferu.

### 39.3.2 Problém večeřících filosofů

- [[todo]]

### 39.3.3 Čtenáři a písaři

- [[todo]]

## 39.4 Softwarové řešení

- Softwarové (algoritmické, pomocí zasílání zpráv) řešení práce se sdílenou pamětí.

### 39.4.1 Dekkerův algoritmus

- [[todo]]

### 39.4.2 Pettersonův algoritmus

- [[todo]]

## 39.5 Hardwarové řešení

- Hardwarové řešení práce se sdílenou pamětí.

### 39.5.1 Test and set

- [[todo]]

### 39.5.2 Swap

- [[todo]]

### 39.5.3 Bounded wait Test and set

- [[todo]]

## 39.6 OS řešení

- Řešení práce se sdílenou pamětí na úrovni operačního systému.

### 39.6.1 Semafory

- [[todo]]

### 39.6.2 Monitory

- [[todo]]

### 39.6.3 Kritické sekce

- [[todo]]

## 39.7 Zasílání zpráv

- Primitiva
  - send
  - receive
- Druhy kanálů
  - Synchronní
  - Asynchronní
- Modely komunikujících procesů
  - CSP (Communication Sequential Processes)
  - Occam
  - PI-Kalkul
  - ADA
  - Linda

# Kapitola 40

# PRL – Distribuované a paralelní algoritmy – předávání zpráv a knihovny pro paralelní zpracování (MPI).

## 40.1 Zdroje

- `PRL_13_MPI.pdf`

## 40.2 Úvod a kontext

- [[todo]]

# Kapitola 41

# PRL – Distribuovaný broadcast, synchronizace v distribuovaných systémech.

## 41.1 Zdroje

- `PRL_12_bcast_slidy.pdf`
- *Otázka souvisí s prvními 2 otázkami z PDI.*

## 41.2 Úvod a kontext

- *Stejný úvodní text jako pro otázku 47.*
- Mějme množinu procesů v rámci distribuovaného systému. Řešíme problém nalezení shody na nějaké věci (synchronizační problém). Problém můžeme rozdělit na dvě situace:
  - **Problém volby koordinátora** – Výběr jednoho z procesů, který bude vedoucím procesem (koordinátor). Tento proces pak může vykonat určitou činnost nebo může sloužit ostatním procesům k realizaci význačné role v systému.
  - **Problém vzájemného vyloučení** – Předpokládejme, že konkrétní zdroj může v daném okamžiku používat pouze jeden proces. Tento problém se běžně vyskytuje ve víceprocesorových systémech, ale také v distribuovaných systémech.
- Synchronizační problémy lze v rámci operačních systémů nebo multiprocesorových systémů řešit pomocí provádění atomických operací, sdílené paměti apod. – je pro ně podpora v rámci operačního systému nebo hardwaru. V distribuovaných systémech takovéto prostředky nejsou často k dispozici, proto se synchronizační problémy řeší pomocí zasílání zprav, resp. algoritmicky.

## 41.3 Synchronizace

- Synchronizace zaručuje (částečné) uspořádání mezi událostmi – zajištění uspořádání některých událostí v systému v nějakém pořadí.
- V distribuovaných systémech se sdílenou pamětí můžeme použít semafory nebo monitory.

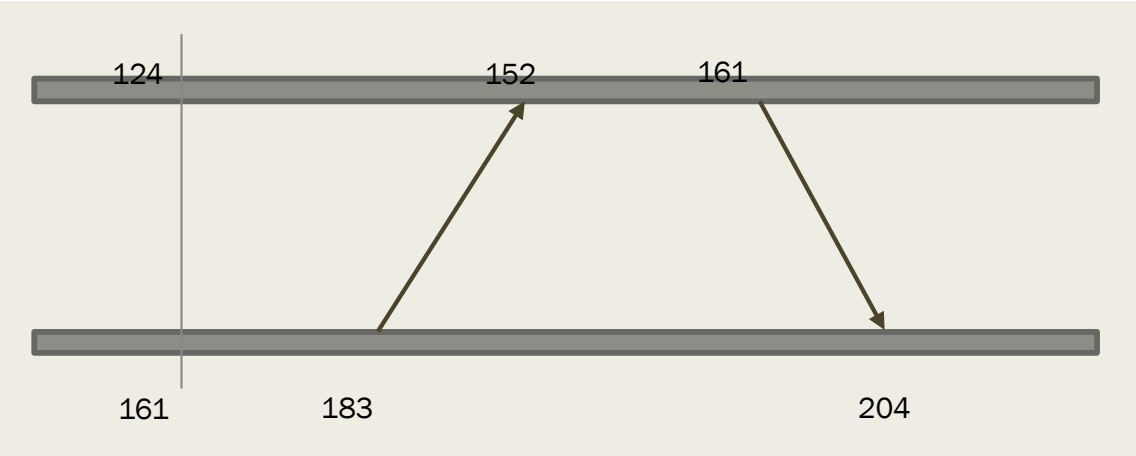
- V distribuovaných systémech se sdíleným prostředkem, jako je například nástěnka, můžeme synchronizaci realizovat například pomocí systému ADA nebo LINDA.
- Ale pokud chceme zajistit synchronizaci pouze pomocí předáváním zpráv potřebujeme algoritmy.
- Požadavky:
  - **Kauzalita** – uspořádání v reálném čase $\sim$ uspořádání dle logických hodin nebo časových razítek („korektní chování z pohledu uživatele“).
  - Všechny procesy uspořádávají události v tom samém pořadí.

## 41.4 Synchronizace globálním (reálným, fyzickým) časem

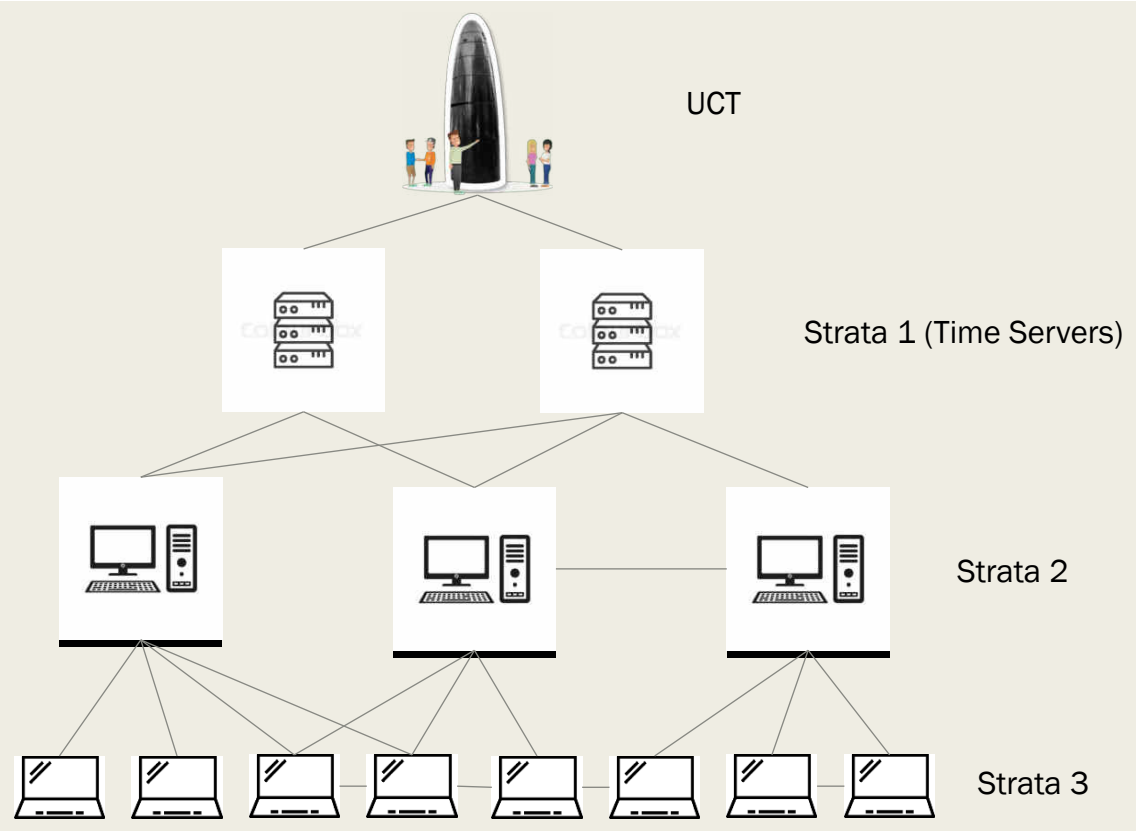
- Využívá se fyzický globální čas.
- Předpokládá se, že komunikace dotaz-odpověď (návratový čas) je dostatečně krátká.
- Hlavní uzel si vyžádá od všech hodnotu posunu vůči svému aktuálnímu času.
- Následně vypočte průměrnou hodnotu posunu a z té posuny pro jednotlivé uzly.
- Seznam algoritmů:
  - Berkley algoritmus
  - Marzullo-ův algoritmus

### 41.4.1 Network Time Protocol (NTP)

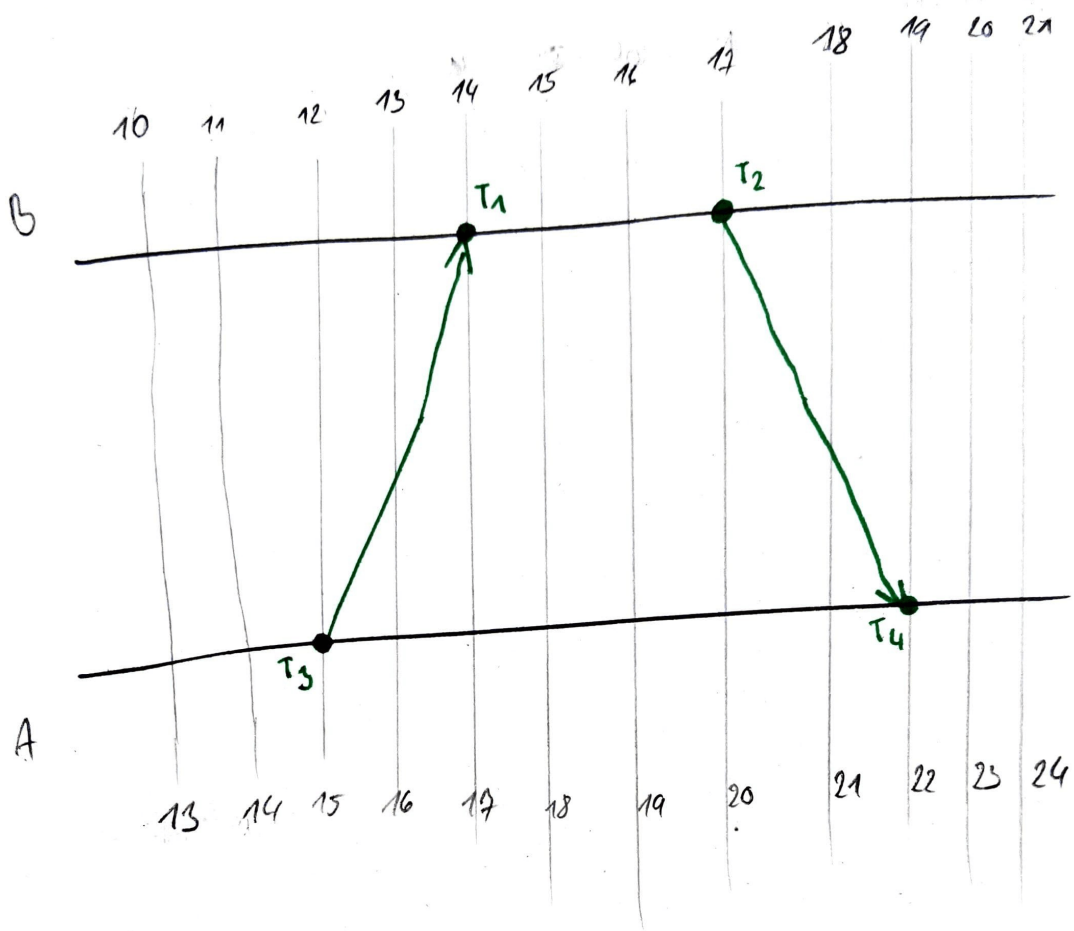
- Synchronizace v síti na různých úrovních (stratech).
- Komunikace přes UDP.
- Universal Coordinated Time (UCT) – Spolehlivý a přesný zdroj času, založen na atomových hodinách (nezávislý rotaci Země teoreticky).
- Řešíme:
  - Zpoždění (*delay*, RTT, *round trip time*) mezi dvěma uzly.
  - Posunutí (*offset*) dvou uzlů vzájemně od sebe.




Obrázek 41.1: Příklad uzlů s fyzickým časem. Zpoždění: $RTT = 152 - 183 + 204 - 161 = -31 + 43 = 12$, posunutí: $o = \frac{1}{2}(152 - 183 + 161 - 204) = -37$.




Obrázek 41.2: Úrovně NTP.




$a = 14 - 15 = -1$

$b = 17 - 22 = -5$

$\theta = \frac{-1-5}{2} = -3$ offset

$\delta = a - b = 4$ round trip time (RTT)

Obrázek 41.3: Další příklad uzlů s fyzickým časem.

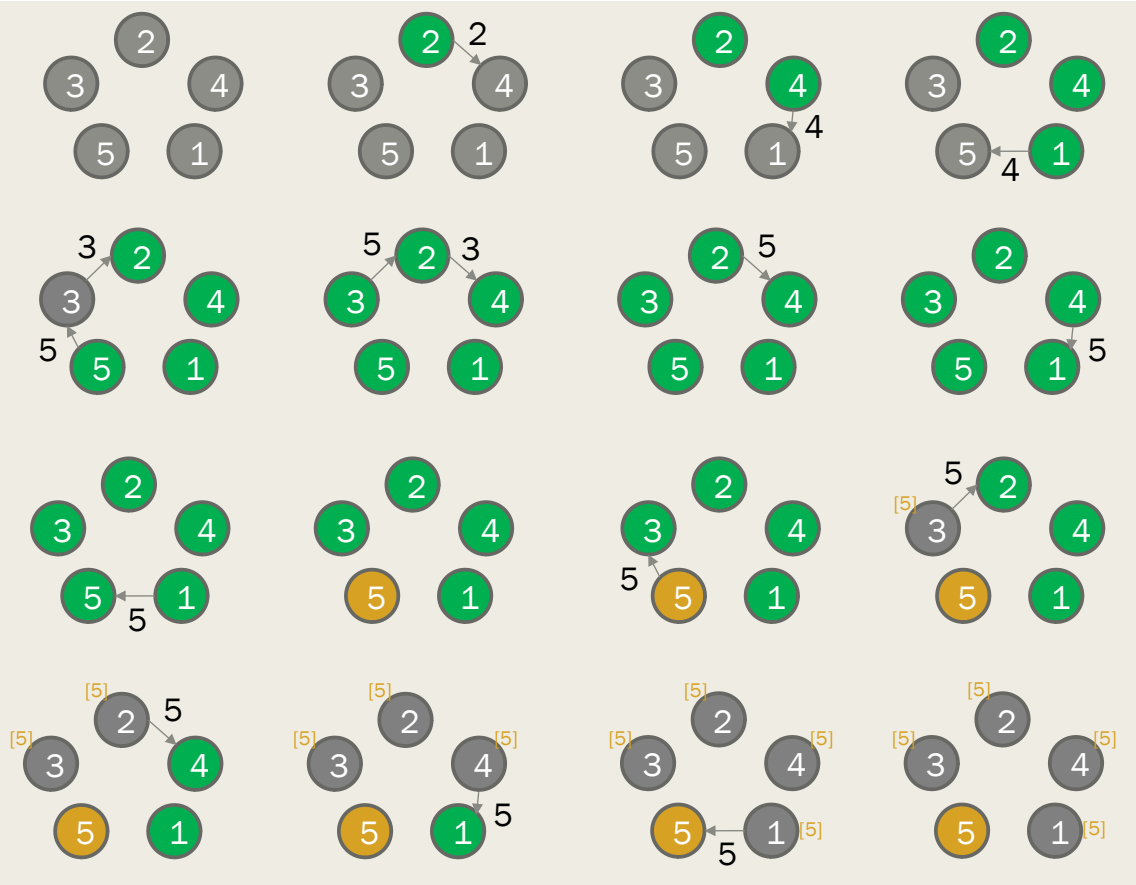
## 41.5 Synchronizace řízená koordinátorem

- Chceme vybrat koordinátora (*master*, hlavní uzel).
  - Koordinátora lze využít pro centralizovanou správu přístupu ke kritické sekci. Koordinátor monitoruje stav sdíleného zdroje a udržuje frontu pro přístup.

- Předpokládáme:
  - Každý proces má unikátní ID.
  - Procesy neznají stav (běžící, neběžící) dalších procesů.
  - Každý proces zná ID dalších procesů (záleží na topologii).
- Cíl:
  - Dosáhnutí shody mezi všemi procesy na procesu, který je koordinátor.
  - Kritérium výběru koordinátora může být různé. Např. na základě proces ID (proces s největším ID se stane koordinátorem).

### 41.5.1 Chang Roberts algoritmus

- *Toto je v podstatě Ring algoritmus z PDI, viz otázku 47. Ring algoritmus používá seznam a díky tomu ušetří jedno kolo zpráv, tj. vyšší prostorová složitost výměnou za lepší časovou.*
- Předpokládá kruhovou topologii.
- Hledá maximální hodnotu PID.
- Zprávy jsou posílány po směru hodinových ručiček.
- **Postup**
  1. Uzel zahájí komunikaci, označí se za účastníka a pošle zprávu se svým ID následujícímu.
  2. Pokud uzel přeposílá zprávu, označí se za účastníka.
  3. Každý uzel po obdržení zprávy:
     (a) Pokud ID ve zprávě je větší než ID tohoto uzlu, je zpráva přeposlána dále tak jak je.
     (b) Pokud je číslo ve zprávě menší než má uzel ID, potom:
        i. Pokud již je účastník, zahodí zprávu.
        ii. Pokud není účastník, nahradí hodnotu původní za svoje ID a přepošle zprávu dále.
     (c) Pokud je číslo ve zprávě stejné jako má uzel ID, potom tento uzel volbu vyhrál. Potom zahájí druhou část algoritmu.
  4. Vítěz volby se odznačí jako účastník a pošle svoje číslo dále.
  5. Každý, kdo obdrží zprávu a je stále účastníkem, si číslo poznačí, odznačí se jako účastník a pošle zprávu dále.
  6. Pokud zprávu obdrží vítěz volby, zprávu zahodí.
- **Složitost**
  - V nejhorším případě se musí přeposlat $3(n-1)$ zpráv.
  - Nejhorší případ – uzel s maximálním indexem je první za iniciátorem hlasování.



Obrázek 41.4: Příklad činnosti Chang-Roberts algoritmu.

## 41.6 Synchronizace logickým časem

- Procesy hledají shodu navzájem na tom, který z nich získá kritickou sekci.
- Známe 2 mechanismy (třídy algoritmů) vzájemného vyloučení v distribuovaných systémech.
- Algoritmy založené na časových razítcích:
  - Lamportův algoritmus
  - Algoritmus Ricart-Agrawala
  - Meakawův algoritmus
- Algoritmy založené na tokenech:
  - Raymondův algoritmus
  - Suzuki-Kasami vysílací algoritmus

### 41.6.1 Logické (Lamportovy) hodiny

- Vytvoření tzv. logického času pomocí relace *happened before*.
  - Relace $R_{HB}(e_1, e_2)$ značí:
    - událost $e_1$ předchází $e_2$ v rámci jednoho procesu;

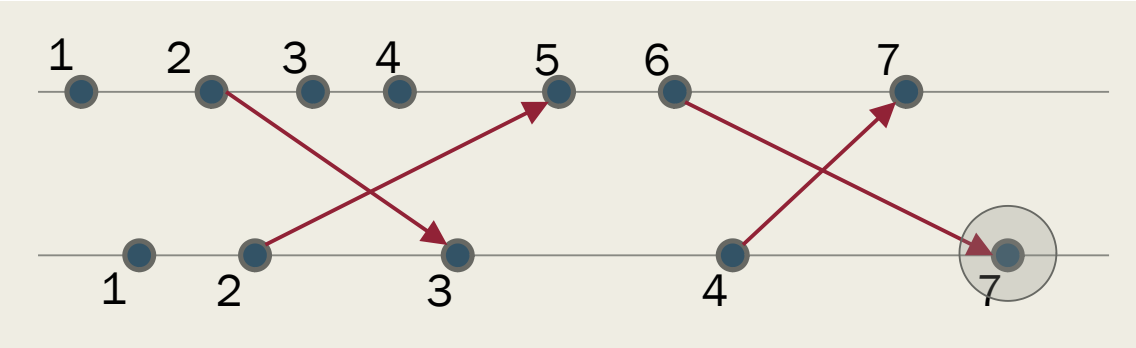
    - událost $e_1$ je $send(p_2, m)$ v rámci procesu $p_1$ a $recv(p_1, m)$ je událost $e_2$ v rámci procesu $p_2$.
  - $R_{HB}$ je tranzitivní, nereflexivní relace částečného uspořádání.
  - Abychom dosáhli úplného uspořádání je třeba dodat další informaci, zde lze použít ID procesů, pak procesy s nižším číslem „mají přednost“ a jejich události, pro které nemůžeme uspořádání původně určit, předcházejí událostem procesů s vyšším číslem.
  - Nerozlišujeme zvýšení logického času na základě toho, jestli k němu došlo vnitřní událostí, nebo na základě komunikace.

- **Definice**
  - Logické hodiny $C$ jsou funkce $C : E \rightarrow T$, kde
    - $E$ je konečná množina událostí;
    - $T$ je časová doména (např. $\mathbb{N}$) nad kterou je definována relace částečného uspořádání ($<$).
  - Každý proces $P_i$ má své logické hodiny $C_i$.
  - Logické hodiny před každou událostí $e$ inkrementují čítač.
  - $C(e)$ vrací pro každou událost odpovídající logický čas.
  - Pro konzistentní hodiny požadujeme:

$$\forall (e_1, e_2) \in E : R_{HB}(e_1, e_2) \Rightarrow C(e_1) < C(e_2)$$

- **Implementace**
  - Spolu se zprávou se posílá i časové razítko dle logického času vysílacího procesu.
  - Při přijetí zprávy $recv(p, m, t_p)$ si příjemce aktualizuje svůj logický čas:

$$C_i = max(C_i + 1, t_p + 1)$$

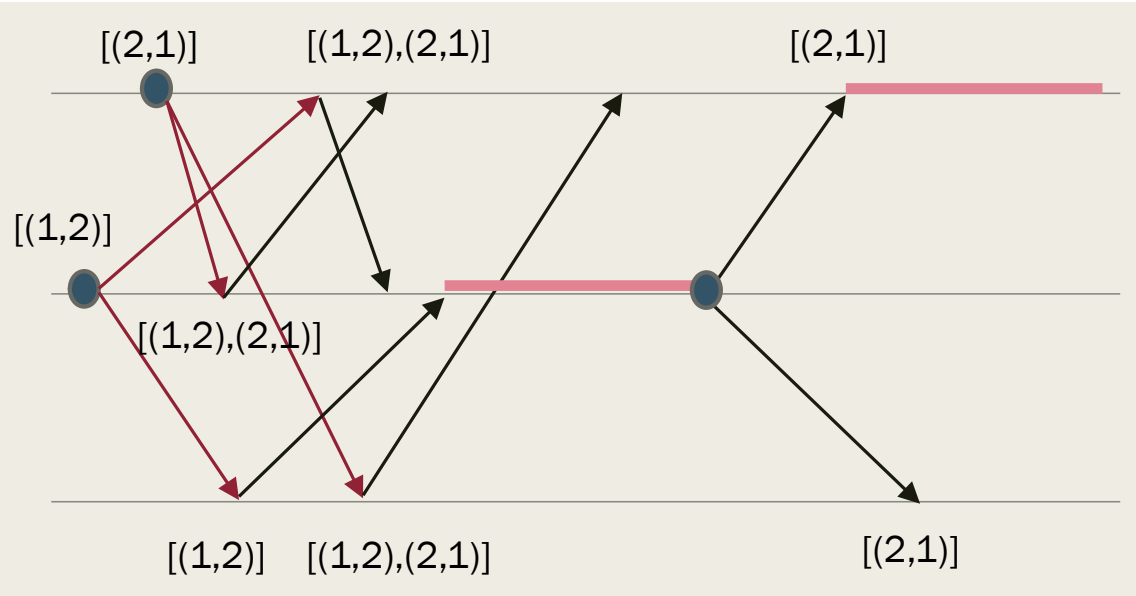



Obrázek 41.5: Příklad komunikace dvou procesů a logických hodin.

### 41.6.2 Lamportův algoritmus

- Založeno na FIFO doručování zpráv.
- Udržuje lokální prioritní frontu zpráv ve které priority jsou úplně uspořádány podle předávaných časových razítek.

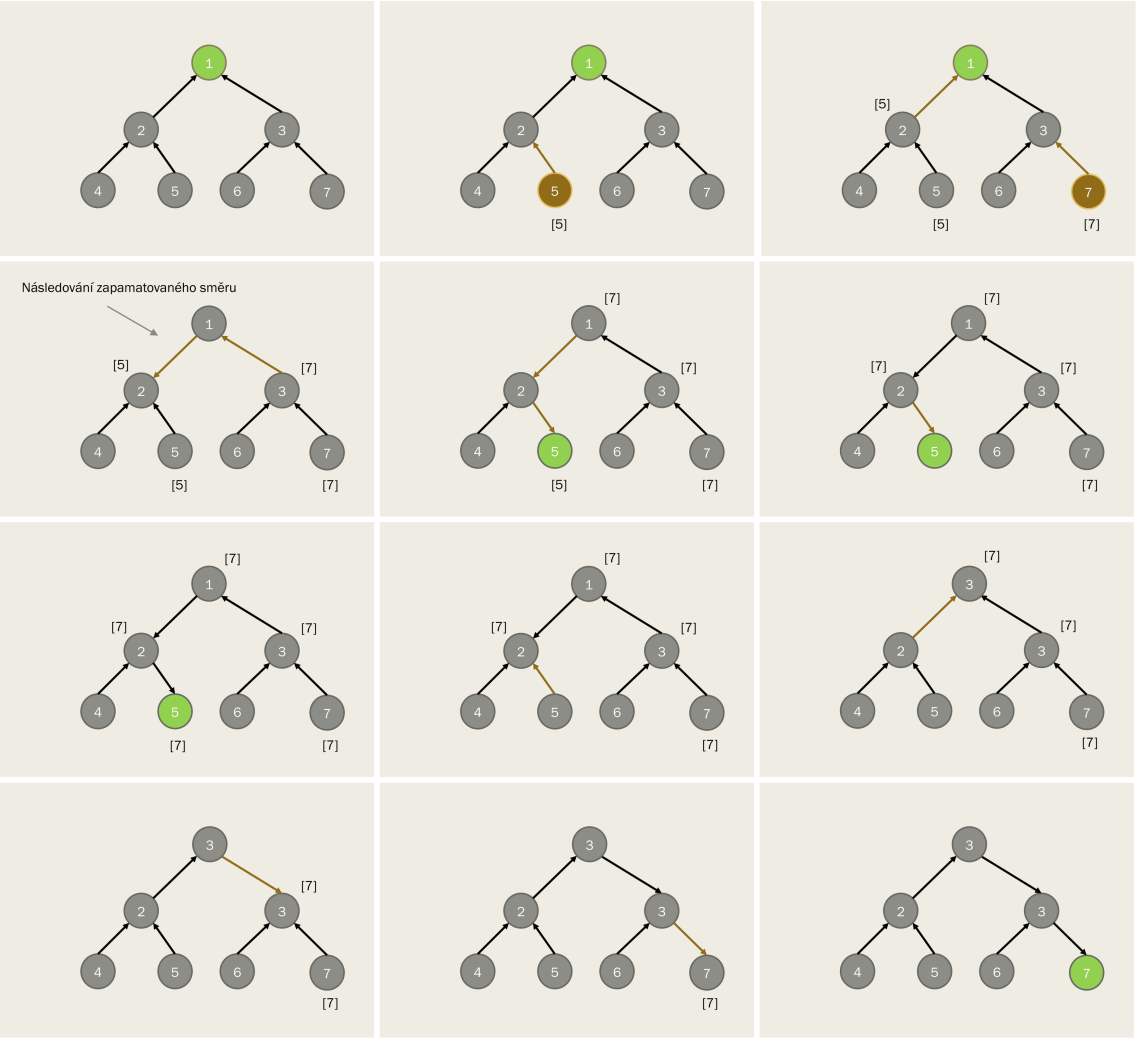
- Složitost: Lamportův algoritmus vyžaduje $3(n-1)$ zpráv:
  - $(n-1)$ požadavků;
  - $(n-1)$ odpovědí;
  - $(n-1)$ uvolnění.




Obrázek 41.6: Příklad Lamportova algoritmu. Časová razítka obsahují navíc i ID procesu a díky tomu jsou úplně uspořádána. Červenou čarou je zvýrazněn přístup do kritické sekce.

### 41.6.3 Raymondův algoritmus

- Jeden token reprezentuje v systému jednu kritickou sekci.
- Proces může vstoupit do kritické sekce, pokud obdrží token.
- Důkaz o vzájemném vyloučení procesu je triviální (token může držet jen jeden proces a ten jej předává, pokud není v kritické sekci).
- Předpokládá stromovou uspořádanou strukturu.
- Analýza:
  - Počet zaslaných zpráv – třída složitosti $\mathcal{O}(\log N)$.
  - Průměrné synchronizační zpoždění je $\log \frac{N}{2}$.
  - Přenosnost se snižuje při zahlcení sítě zprávami.
  - Může použít greedy strategii – možnost hladovění.




Obrázek 41.7: Příklad Raymondova algoritmu. Zeleně je označen proces s přístupem do kritické sekce. Hnědě proces, který o kritickou sekci žádá.

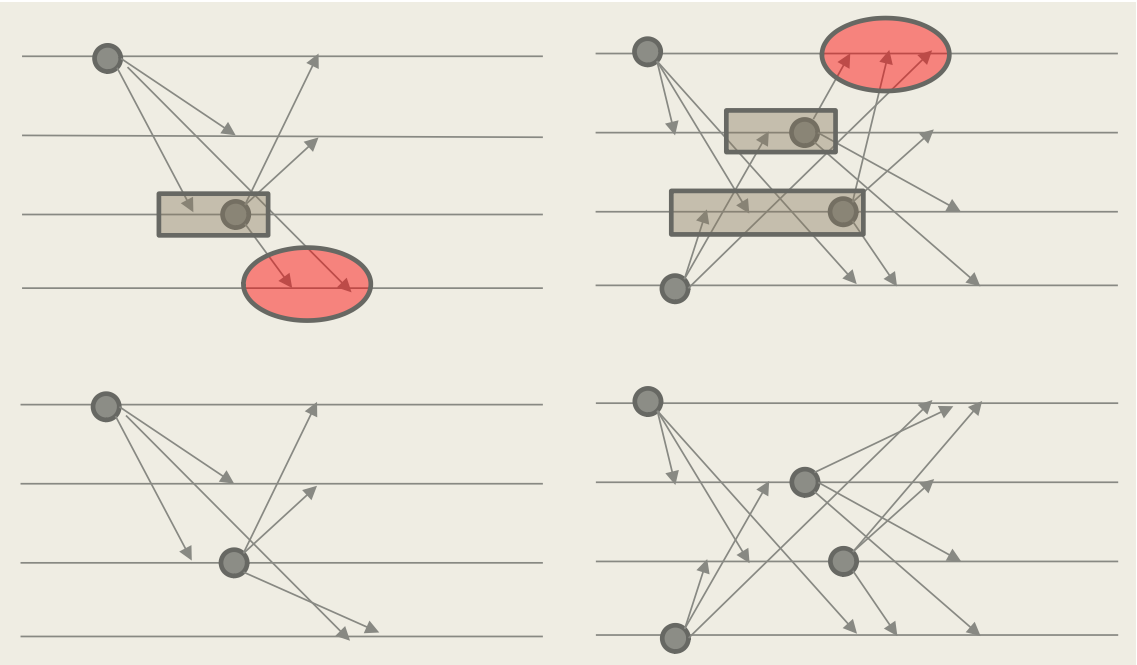
## 41.7 Distribuovaný broadcast

- Distribuovaný broadcast ∼ všesměrové vysílání.
- Předpoklady pro broadcast:
  - Známé největší možné zpoždění při doručování zprávy.
  - Lokální hodiny pro každý z procesů.
  - Známý nejvyšší časový limit pro vykonání interní akce.
  - Topologie sítě je obecná.
- Vlastnosti
  - **Platnost** (*validity*) – Pokud zpráva byla odeslána korektním procesem, tak v konečném čase ji každý další proces obdrží.
  - **Shoda** (*agreement*) – Pokud zpráva byla obdržena korektním procesem, pak v konečném čase ji obdrží všechny procesy.

- **Integrita** (*integrity, no duplication*) – Žádná zpráva není doručena více než jednou.
- **Opravdovost** (*no creation*) – Pokud proces obdržel zprávu $m$ od procesu $p$, pak tento proces zprávu opravdu odeslal.

- Vlastnosti z hlediska pořadí doručování
  - **FIFO uspořádání** – Pokud proces broadcastne zprávu $m$ a poté zprávu $n$, pak každý proces obdrží nejprve $m$ a poté až $n$ (FIFO uspořádání od jednoho procesu).
  - **Kauzální uspořádání** – Pokud broadcast zprávy $m$ předchází broadcastu zprávy $n$, pak každý proces obdrží nejprve $m$ a poté až $n$ (FIFO uspořádání od všech procesů).
  - **Úplné uspořádání** – Pokud procesy $p$ a $q$ oba obdrží zprávu $m$ a $n$, pak $p$ obdrží $m$ před $n$ pokud $q$ obdrží zprávu $m$ před $n$ (přijímají z pohledu příjemců ve stejném pořadí všechny procesy).

- Klasifikace
  - Best effort broadcast – platnost (shoda nikoliv, protože nekorektní proces odešle pouze podmnožině korektních).
  - Spolehlivý (reliable) broadcast – platnost, shoda a integrita.
  - FIFO broadcast – spolehlivý a FIFO uspořádání.
  - Kauzální broadcast – spolehlivý a kauzální uspořádání.
  - Atomický broadcast – spolehlivý, FIFO uspořádání a kauzální uspořádání.
  - Kauzálně atomický broadcast – spolehlivý, kauzální uspořádání a úplné uspořádání.



Obrázek 41.8: Příklad broadcastu.

# Kapitola 42

# PRL – Klasifikace a vlastnosti paralelních a distribuovaných architektur, základní typy jejich topologií.

## 42.1 Zdroje

- `PRL_01_02_Arch_MNG.pdf`
- *Otázka není vysázená, pouze vloženo exportované PDF z google docs.*

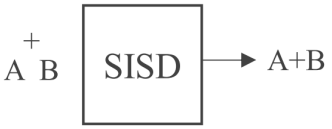
# 1) Architektury

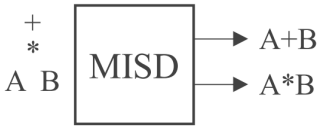
## Klasifikace paralelních systémů

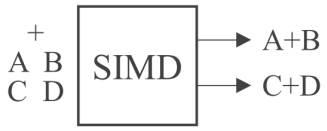
### Von Neumannova / Harvadská architektura

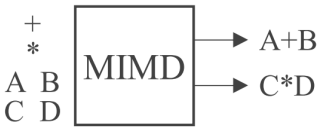
- Má instrukční čítač, je známo, kde se nacházíme v posloupnosti instrukcí

### Flynnova klasifikace



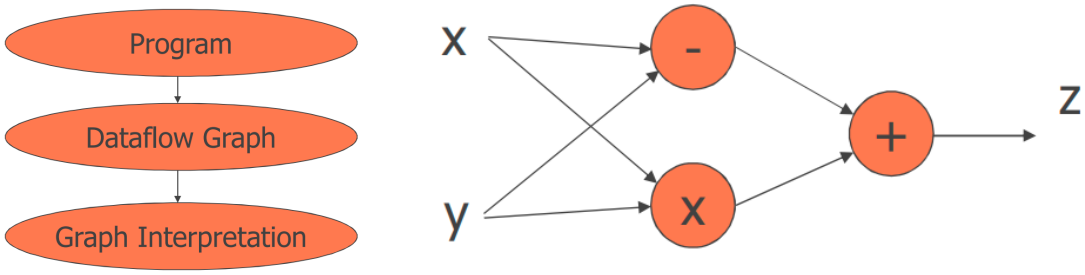




- **SISD**
  - V paralelním prostředí: **VLIW** (Very long instruction word)
  - Instrukce až tisíce bitů, je rozdělena na pole, které provádějí samostatné operace (paralelně)
- **SIMD**
  - Vektorové / maticové procesory
- **MISD**
  - Zřetězené procesory (překrývání instrukcí)
- **MIMD**
  - Multiprocesory (sdílená paměť)
  - Multicomputery (předávání zpráv) – Distribuované systémy

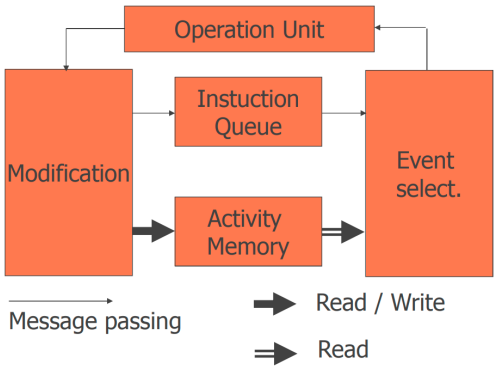
### Data Flow architektura

- Nemá instrukční čítač
- Vyhodnocuje výrazy, které jsou v paměti reprezentovány pomocí grafu
- Výpočet provádí tehdy, je-li pro operaci dostatečný počet vstupních dat




- Typicky funkcionální jazyky
- Při psaní programu nemusíme přemýšlet paralelně, architektura se sama snaží najít všechno co lze vypočítat paralelně
- Realizovatelná architektura, používá se pro konkrétní věci (např. simulace)

**Dataflow procesor**




## Redukční architektura

- Program je napsán jako řetězec, který se počítač snaží postupně redukovat, až dojde k jednomu výsledku.
- Mám pole ve kterém jsou řetězce, toto pole prohledávám a snažím se najít takové podřetězce, které se dají zredukovat na něco jednoduššího.
- Redukce se provádí paralelně.
- Zatím nemá využití – hledá se obtížně hardware realizace.

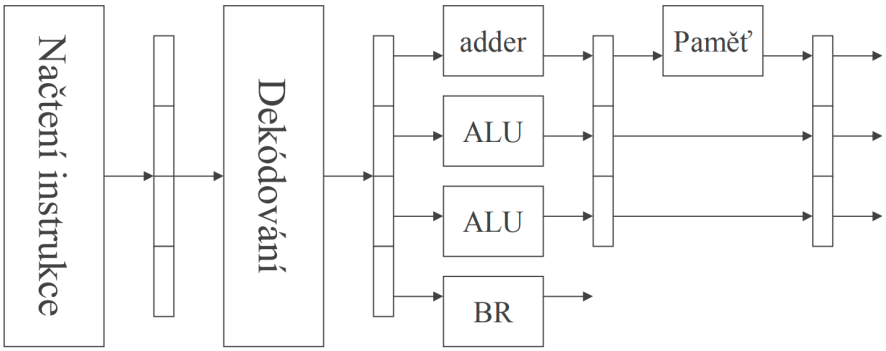
### Redukce pro x=3 y=2

- +<(*<3 2>)(-<3 2>)>
- +< 6 1 >
- 7

# Konkrétní Von Neumannovy (Harvardské) architektury

## VLIW – Very Long Instruction Word

- Jediný tok řízení, který řídí všechny procesory (SISD)
- Přesto lze paralelismus
- Operační kód je velmi dlouhý a je rozdělen na části, které provádí různé operace (ale jsou řízeny jedním čítačem).

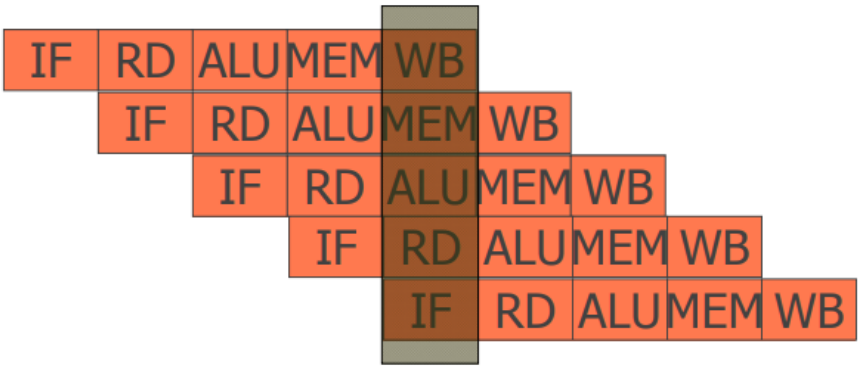



- Překladač musí přeložit dlouhou instrukci na menší a poskládat je tak, aby šly řešit paralelně.
- Oproti tomu Super Skalární procesory rozložení instrukcí řeší až na úrovni hardware
- Výhody
    - Jednoduchý hardware (instrukce překládá překladač)
    - Dobře škálovatelný
- Nevýhody
    - Podmíněné skoky jsou problém
    - Problém toku dat – instrukce zpracované v jednom kroku, nemohou navzájem používat své výsledky
    - Velikost programu – synchronizační NOPy

## Zřetězené procesory

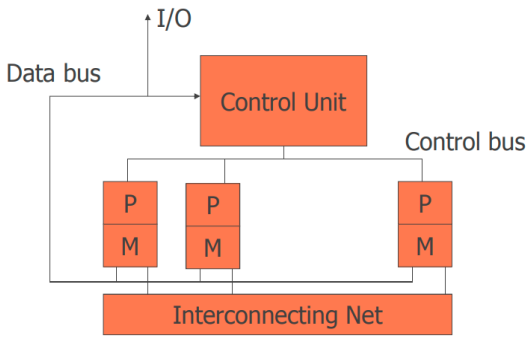
- MISD
- Lineárně propojené procesory
- Řešení úloh s proudovým charakterem
- Data procházejí postupně jednotlivými procesory

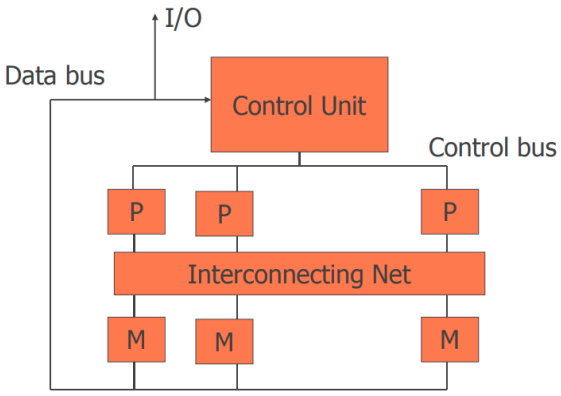
- “Fáze” instrukcí



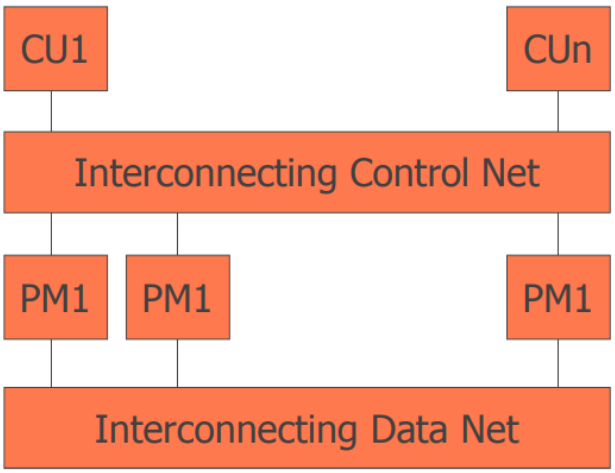

## Vektorové procesory

- SIMD
- Jedna řídící jednotka
- Propojení více výpočetních (datových) jednotek, které mají každá segment paměti.
- Všechny procesory provádějí stejnou instrukci, ale s různými daty.
- Propojovací síť propujuje jednotlivé paměťové moduly.
- Rozdílný čas přístupu do paměti.
- Dva modelu (obrázky)




- Reálně se používá Multiple SIMD (MSIMD)
  - Několik řídících jednotek, každá si může zabrat několik procesorů
  - Z důvodu běhu OS apod.




- Výhody
  - Jednodušší než MIMD (menší počet tranzistorů na procesor)
  - Menší nároky na paměť.
  - Žádná synchronizace mezi procesy
- Nevýhody
  - Ne všechny problémy jsou datově paralelizovatelné
  - Pokles výkonnosti u programů s mnoha podmíněnými skoky
  - Nejsou vhodné při malém počtu procesorů

# Granularita paralelismu

## Uvnitř instrukcí

- Uvnitř jedné instrukce několik jednotek, které paralelně provádí činnost té instrukce
- Nejjemnější paralelismus
- Pro programátora neviditelný
- Např.:
  - Načíst operační kód, paralelně se spočítá jeho hodnota a zároveň se inkrementuje programový čítač

## Mezi instrukcemi

- Některé instrukce se provádějí paralelně
- Není vidět na úrovní programovacího jazyka (na úrovni assembleru může, záleží na implementaci)

### Mezi příkazy

- Vektorové počítače
- Specializované koprocesory (FPU)
- C, pragma překladačům, knihovny, apod.
- Např.:
    - Vektorizace smyček

### Mezi bloky procesu

- Vlákna (threads)
- Obvykle je mezi vlákny sdílen adresový prostor, ale každé má oddělený programový čítač
- Vyšší programovací jazyky

### Mezi procesy

- Žádná sdílená paměť
- Nutné zajistit komunikaci a synchronizaci
- Vyšší programovací jazyky

# 2) Komunikace

### Synchronizace

- Zajištění požadovaných vztahů mezi událostmi.
- Nepřenáší se data

### Prostředky

- Zasílání zpráv
- Semafor
- Monitor
- Bariéra

### Typické synchronizační úlohy

- Soupeření – vzájemné vyloučení (čtenáři – písaři)
- Kooperace – dohoda (producent - konzument)

## Komunikace

- Přenáší data
  - Předávání informací mezi procesy (vlákny)
- Prostředky pro komunikaci
  - Sdílená paměť
  - Zasílání zpráv

## Sdílená paměť

- Skutečná – HW
- Simulovaná – na úrovni OS
- Všechny procesy mají přístup do společného paměťového prostoru
- Je třeba řešit současný přístup k jedné buňce paměti

### EREW – Exclusive read exclusive write

- Snadný pro HW, složité pro algoritmy

### ERCW – Exclusive read concurrent write

- Nelogická varianta, souběžné čtení je daleko snažší zajistit než souběžný zápis, v reálu neexistuje

### CREW – Concurrent read exclusive write

- Něco mezi

### CRCW – Concurrent read concurrent write

- Složitý pro HW, snadný pro algoritmy
- Několik způsobů řešení zápisových konfliktů
  - **COMMON** – všechny zapisované hodnoty musí být shodné (logický OR)
  - **ARBITRATY** – zapisované hodnoty můžou být různé a zapíše se libovolná z nich (vyhledávání, index)
  - **PRIORITY** – zapisované hodnoty mohou být různé a zapíše se ta s nejvyšší prioritou (vyhledávání, nejlevější index)

## Zasílání zpráv

- Každý procesor má svůj adresový prostor
- Komunikace probíhá pomocí zpráv

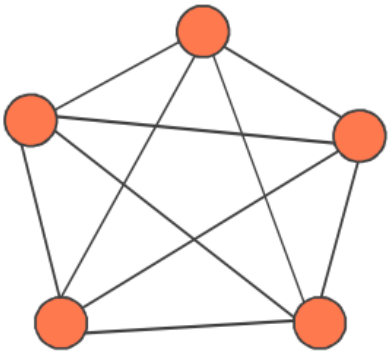
## Propojovací sítě

- Propojit procesory se sdílenou pamětí
- Propojit procesory spolu

## Statická

- Během výpočtu se propojení nemění
- Používají se pro architektury bez sdílené paměti
- Různé architektury

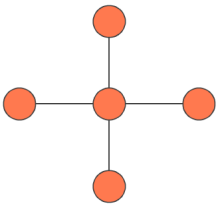
**Úplné propojení**



- Diameter = 1
- Arc connectivity = $p$-1
- Bisection width = $p^2/4$

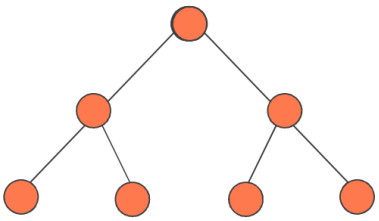
**Hvězda**

- pro architektury slave – master



- Diameter = 2
- Arc connectivity = 1
- Bisection width = ($p$-1)/2

**Stromy**

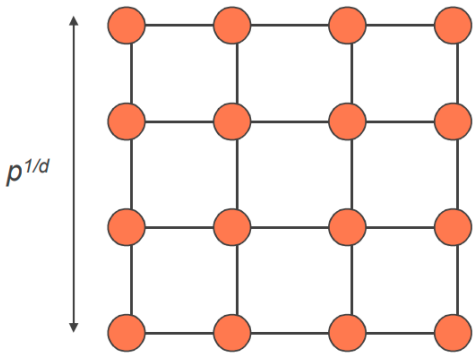


- Diameter = 2 $\log_d$ ((p+1)/2)
- Arc connectivity = 1
- Bisection width = 1
- Obvykle binární

**Lineární pole**

- Diameter = $p$-1
- Arc connectivity = 1
- Bisection width = 1

**Mřížka**




- Kartézský součin $d$ lineárních polí, z nichž každé má $p^{1/d}$ uzlů
- Diameter = $dp^{1/d}$
- Arc connectivity = $d$
- Bisection width = $2p^{(1-1/d)}$
- <u>Obvykle d=2</u>

## Dynamická

- Během výpočtu se propojení mění (uvnitř běhu algoritmu)
- Více se do toho pouštět nebudeme – složité

# Kapitola 43

# PRL – Distribuované a paralelní algoritmy – algoritmy řazení, select, algoritmy vyhledávání.

## 43.1 Zdroje

- `PRL_03_04_Sort_MNG.pdf`
- `PRL_05_Search_MNG.pdf`
- *Otázka není vysázená, pouze vloženo exportované PDF z google docs.*

# 4) Algoritmy řazení

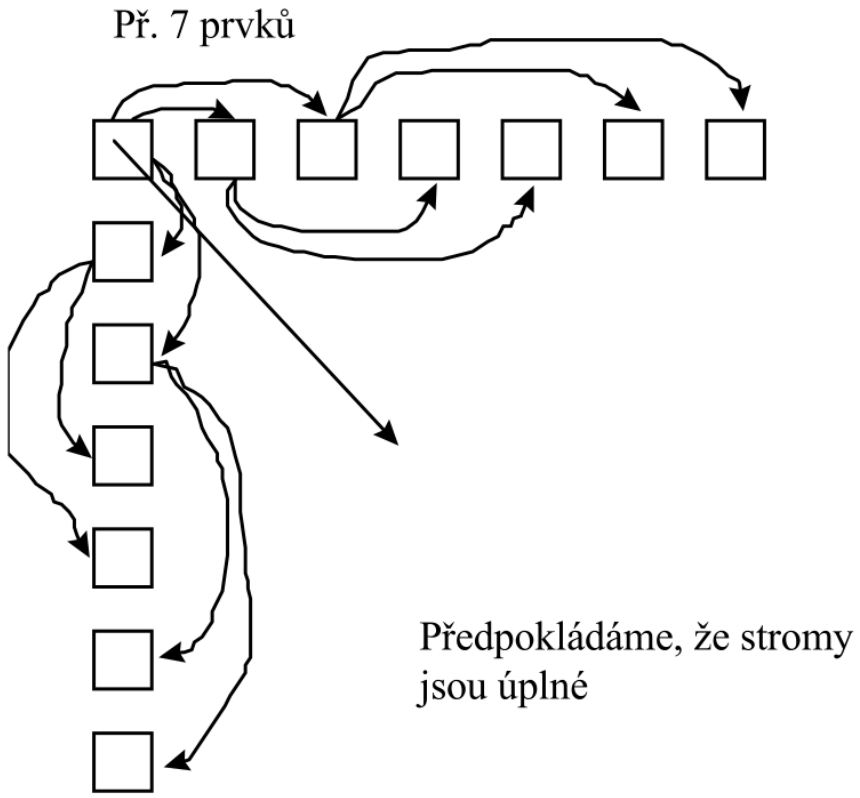
- Optimální sekvenční:
  - p(n) = 1
  - t(n) = O(n * log(n))
  - c(n) = O(n * log(n))

## Enumeration sort (mřížka)

- Správná pozice každého prvku ve výstupní seřazené posloupnosti je dána počtem prvků, které jsou menší než tento prvek.
- Ideální algoritmus pro paralelní zpracování.

### Topologie

- Na mřízce
- p(n) = n^2
- Extrémní případ, dosáhneme nejlepší časové složitosti
- Mřížka, kde procesory jsou vodorovně propojeny do stromu a svisle taky



## Vlastnosti procesoru

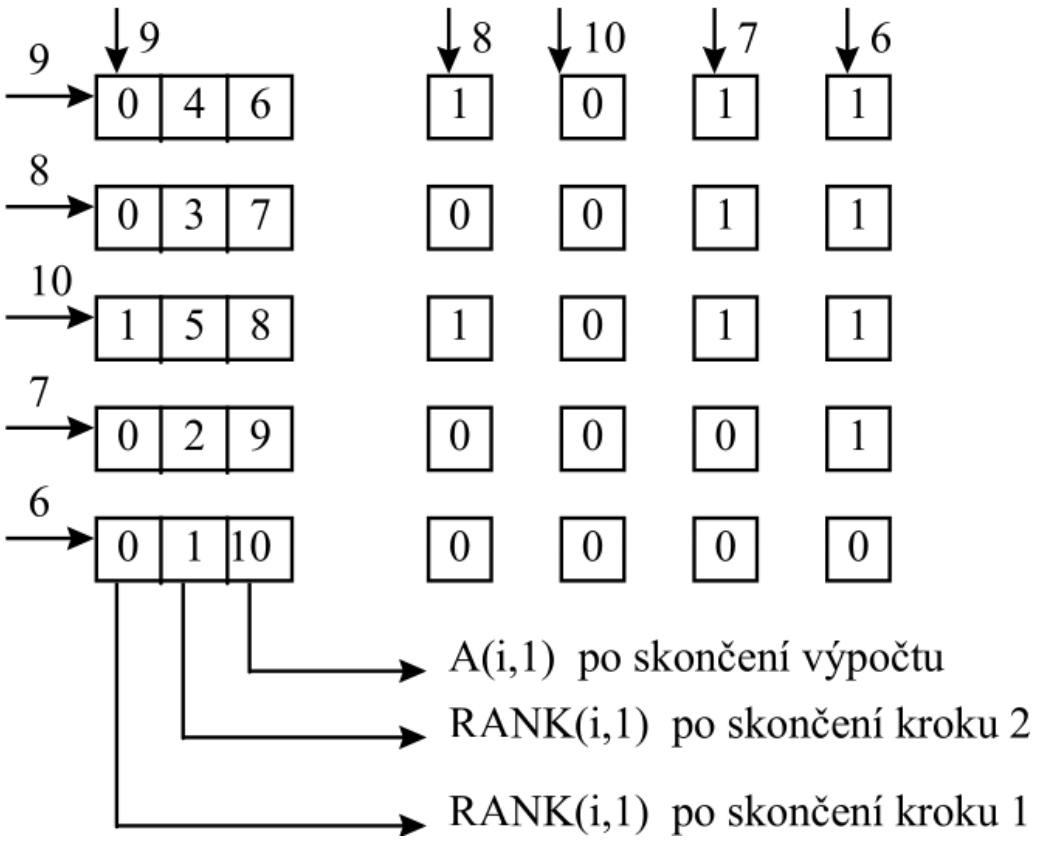
- Může uložit dva prvky do svých registrů A, B
- Může porovnat A a B a uložit výsledek do registru RANK
- Pomocí stromového propojení může předat obsah kteréhokoliv registru jinému procesoru
- Může přičítat k registru RANK

## Algoritmus

1. Každý prvek je porovnán se všemi ostatními pomocí jedné řady procesorů
2. Správná pozice prvku je RANK(x i ) = 1 + počet menších prvků
3. Každý prvek je zadán na správné místo

## Příklad

Př. x = {9, 8, 10, 7, 6}, u prvního sloupce pro všechny registry, u ostatních jen RANK



### Analýza

- t(n) = O( log(n) )
- c(n) = O( n^2 * log(n) )
- p(n) = n^2
- Není optimální

### Diskuze

- Extrémně rychlý algoritmus O(log(n)), což znamená zrychlení O(n) oproti optimálnímu sekvenčnímu algoritmu.
    - Nic není rychlejší
- Spotřebovává příliš mnoho procesorů, na hranici přijatelnosti

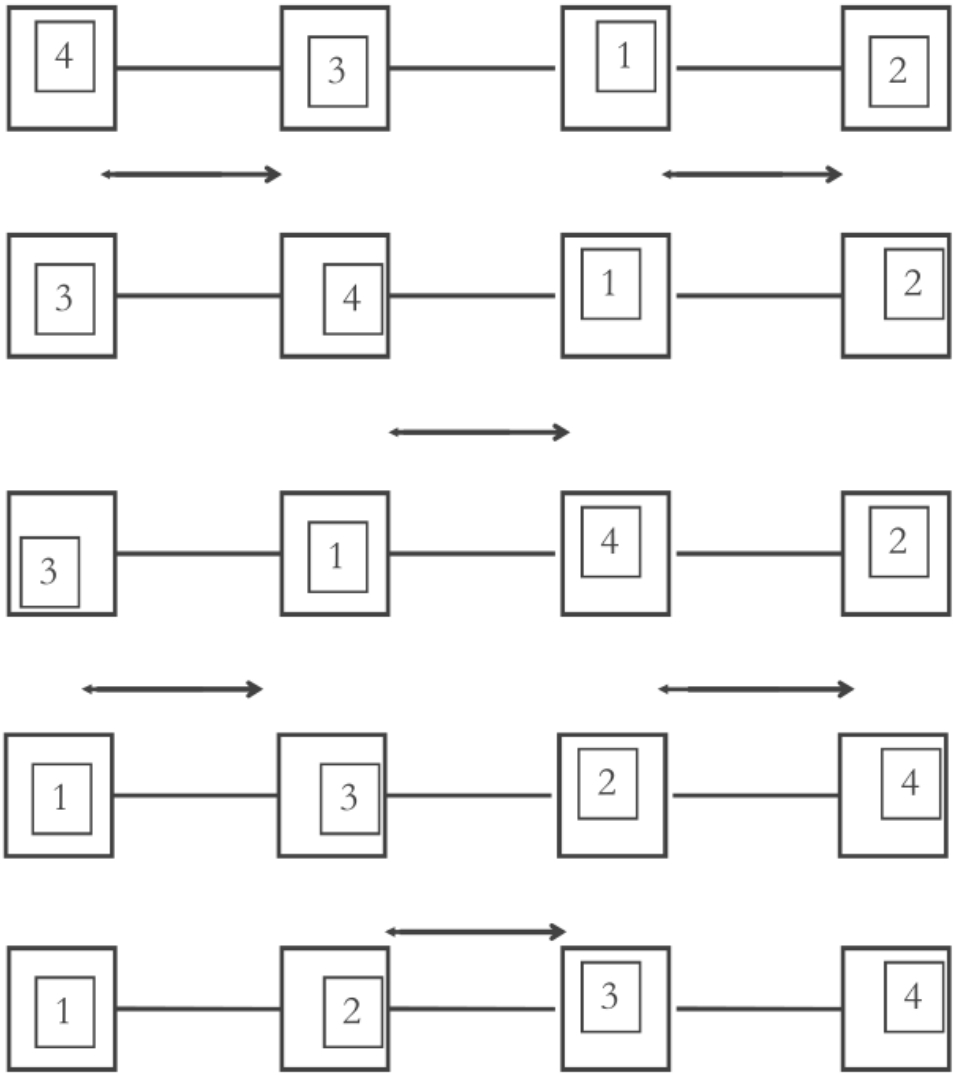
## Odd-even transposition sort

- Paralelní bubble sort
- Lineární pole n procesorů p(n) = n

### Algoritmus

- Na počátku každý procesor p_i obsahuje jednu z řazených hodnot y_i
- V prvním kroku se každý lichý procesor p_i spojí se svým sousedem p_i+1 a porovnají své hodnoty je-li y i >y i+1, procesory vymění své hodnoty
- V druhém kroku každý sudý procesor udělá totéž
- Po n krocích (maximálně) jsou hodnoty seřazeny

## Příklad



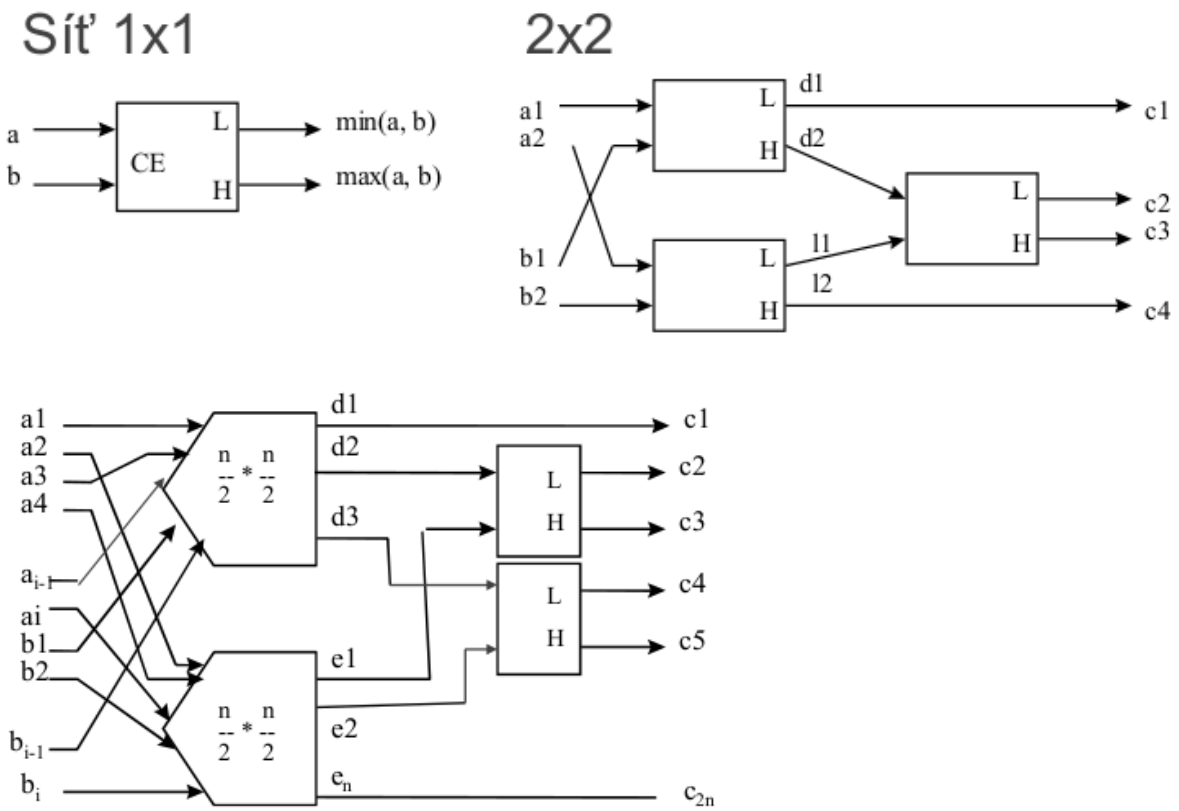

## Analýza

- Každý z kroků (1) a (2) provádí jedno porovnání a dva přenosy – konstantní čas
- Složitost: t(n) = O(n)
- Cena: c(n) = t(n) * p(n) = O(n) * n = O(n^2)
    - což není optimální
- Algoritmus má lineární časovou složitost, což je to nejlepší, čeho lze při lineární topologii dosáhnout

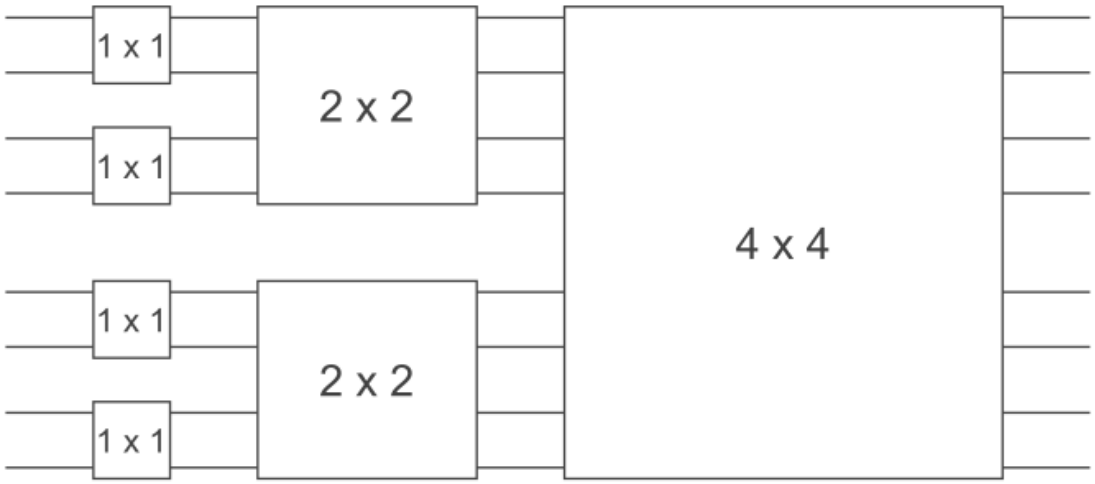
## Odd even merge sort

- Řadí se speciální sítí procesorů
- Každý procesor má dva vstupní a dva výstupní kanály
- Každý procesor umí porovnat hodnoty na svých vstupech, menší dá na výstup L (low), a větší dá výstup H (high).

## Topologie



## Příklad




## Analýza

- Řadíme posloupnost o délce n = 2^m
- 1.fáze potřebuje 2^m-1 CE
- 2.fáze potřebuje 2^m-2 sítí 2x2 po 3 procesorech
- 3.fáze 2^m-3 sítí 4x4 po 9 procesorech
- 4.fáze 2^m-4 sítí po 25 procesorech
- atd.

## Časová složitost

- t(n) = O(m^2 ) = O(log^2 (n))

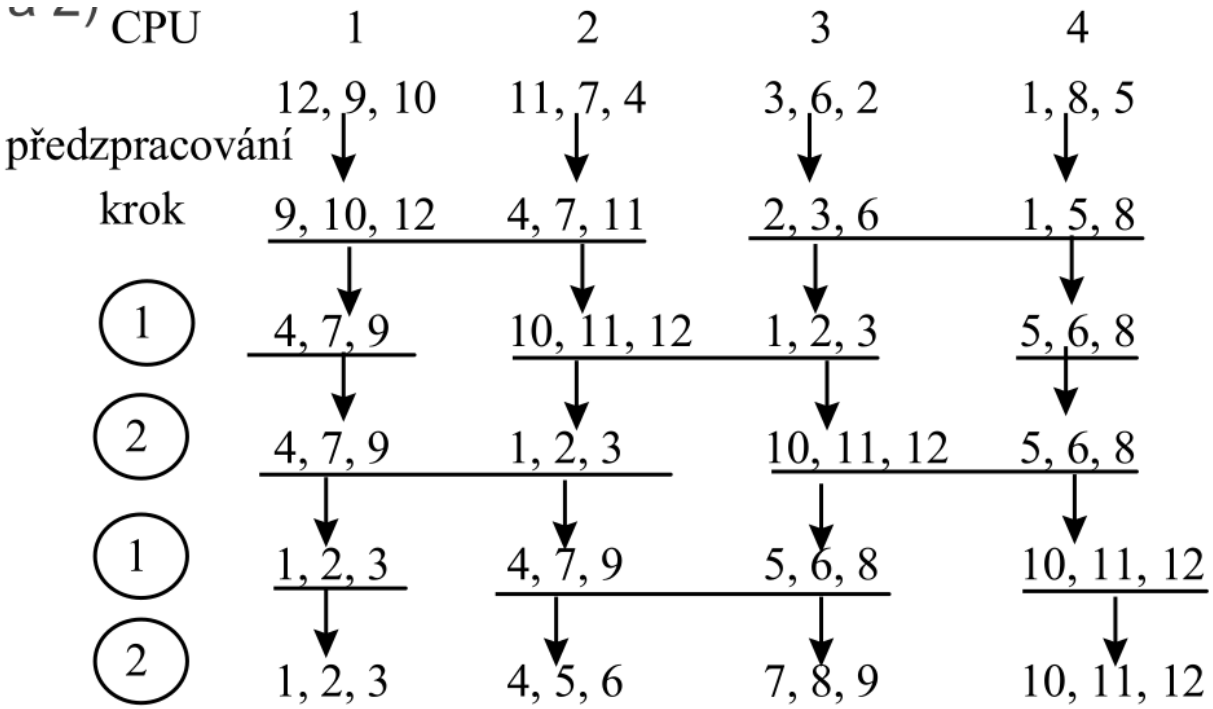
## Cena

- c(n) = O(n * log^4 (n))
- Což není optimální

# Merge-splitting sort

- Lineární pole procesorů p(n) < n
- Je variantou algoritmů lichý-sudý, kde každý procesor obsahuje několik čísel
- Porovnání a výměna je nahrazena operacemi merge-split
- Každý CPU se stará o více prvků
- Pomocí nastavení počtu procesorů jsme schopni nastavovat vlastnosti algoritmu

Příklad




Analýza

- Předzpracování optimálním alg.
  - O((n/p) * log (n/p))
- Přenos S_i+1 do P_i
  - O(n/p)
- Spojení S_i a S_i+1 do S'_i optimálním alg.
  - 2*n/p
- Přenos S_i+1 do P_i+1
  - O(n/p)
- Krok 1 nebo 2
  - O(n/p)

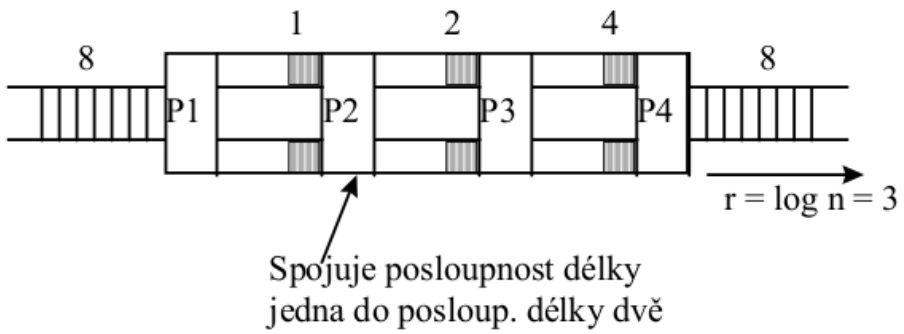
t(n) = O[(n/p) * log (n/p)] + O(n) = O((n * log(n))/p) + O(n)
c(n) = t(n) * p = O(n * log n) + O(n * p)
což je optimální pro p <= log n

## Pipeline Merge sort

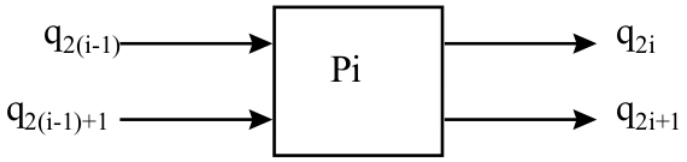
- Lineární pole procesorů p(n) = log n + 1

- Data nejsou uložena v procesorech, ale postupně do nich vstupují
- Každý procesor spojuje dvě seřazené posloupnosti délky 2i-2

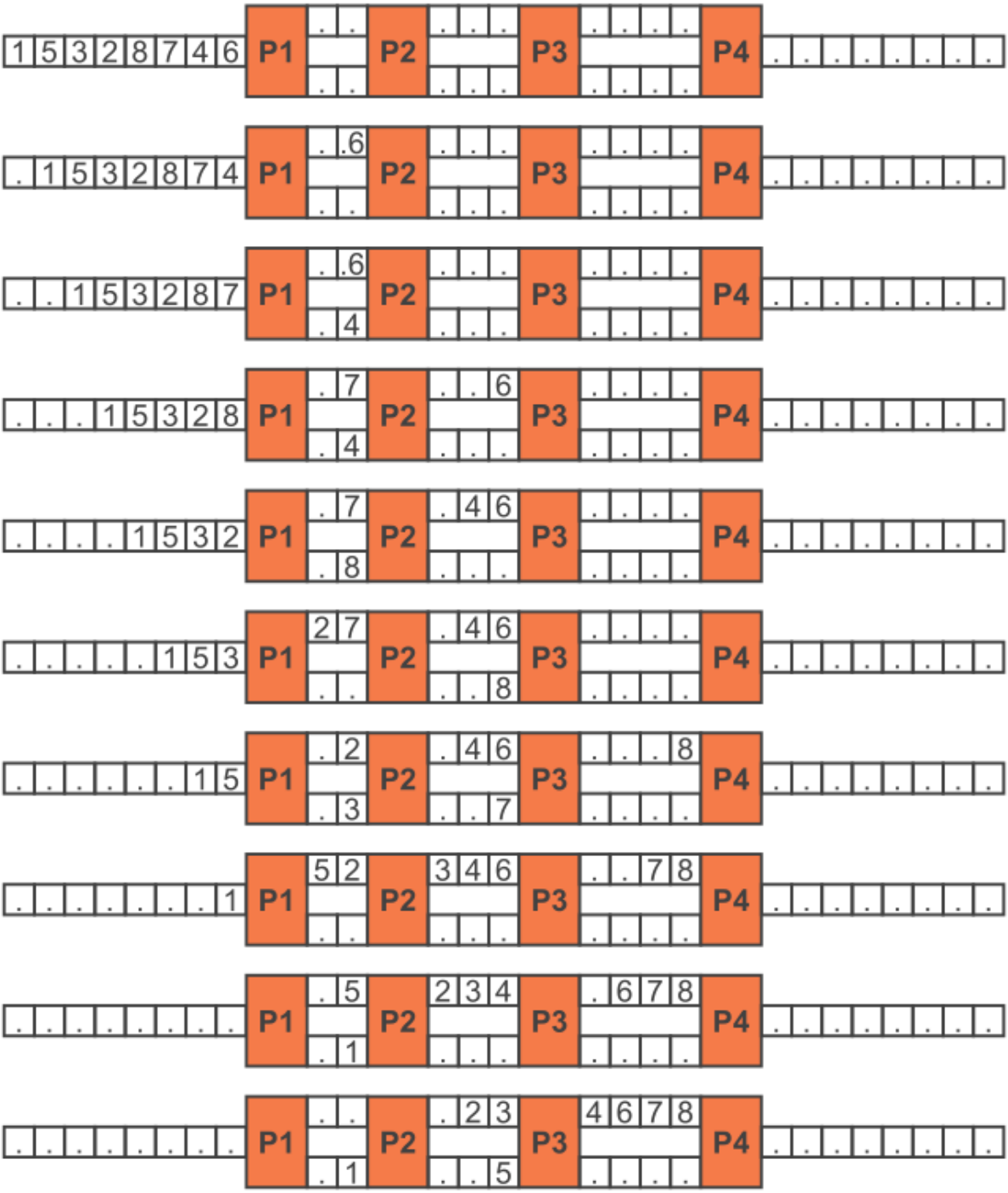
## Topologie




Označení front :



## Příklad




## Analýza

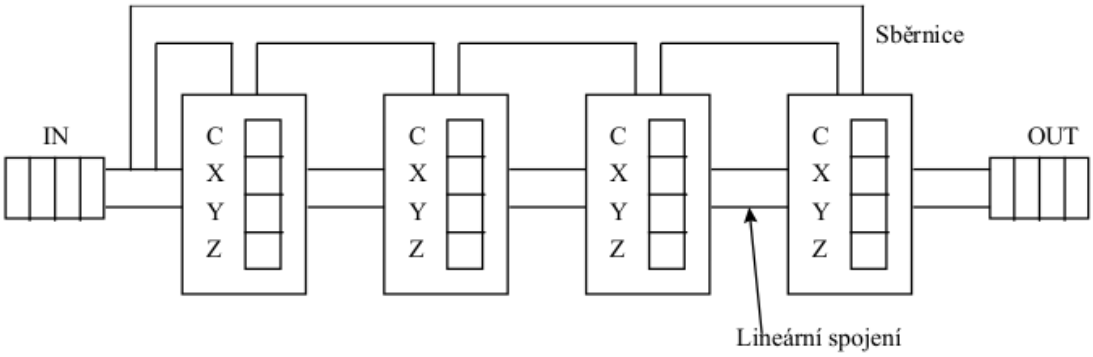
- c(n) = t(n) * p(n) = O(n) * (log n + 1) = O(n * log n)

- což je optimální

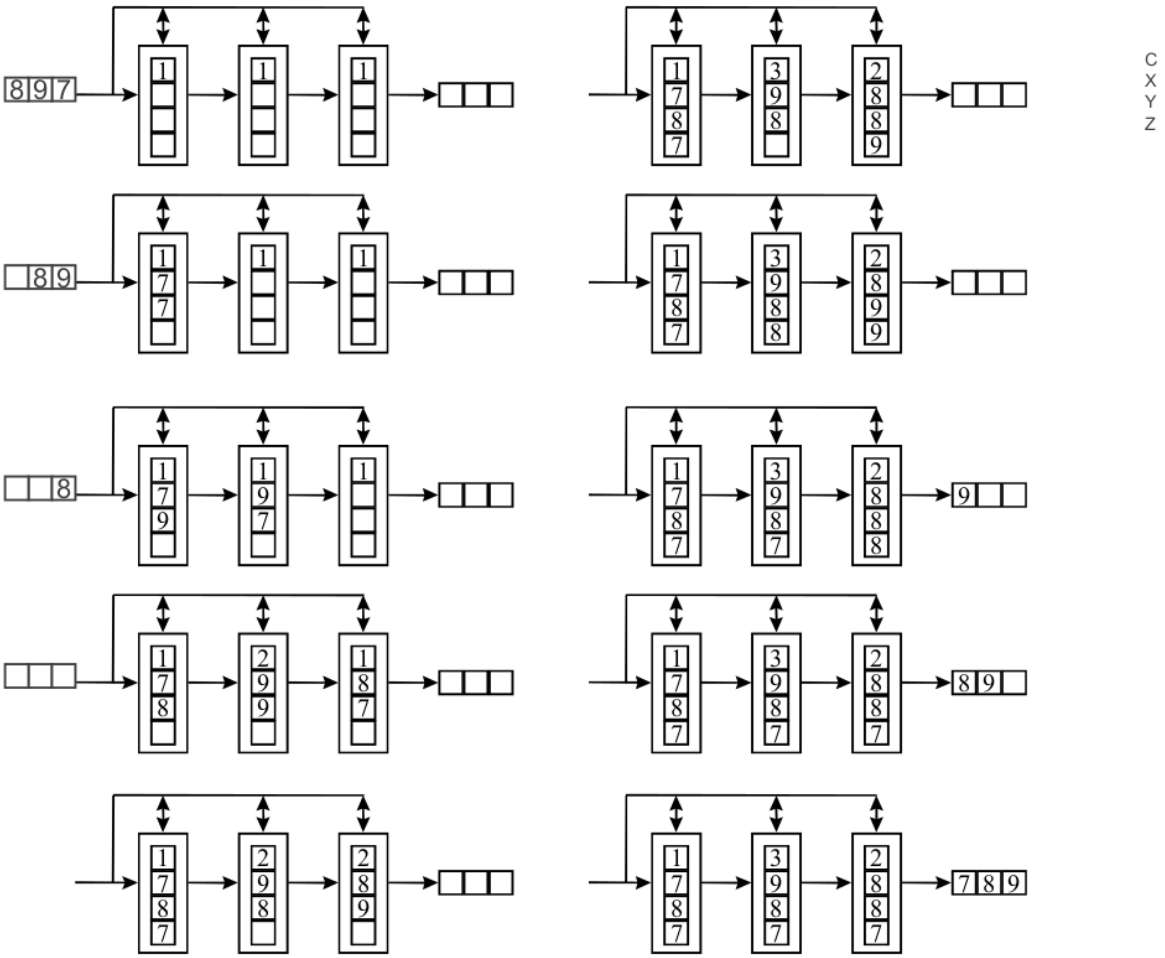
## Enumeration sort (lineární pole)

- Lineární pole n procesorů, doplněných společnou sběrnicí, schopnou přenést v každém kroku jednu hodnotu.

## Topologie



## Příklad




## Příklad

- Krok 1) je v konstantním čase, krok 2) trvá 2n cyklů, krok 3) trvá n cyklů
- t(n) = O(n)
- c(n) = t(n) * p(n) = O(n) * n = O(n2)
- což není optimální

# Minimum Extraction Sort

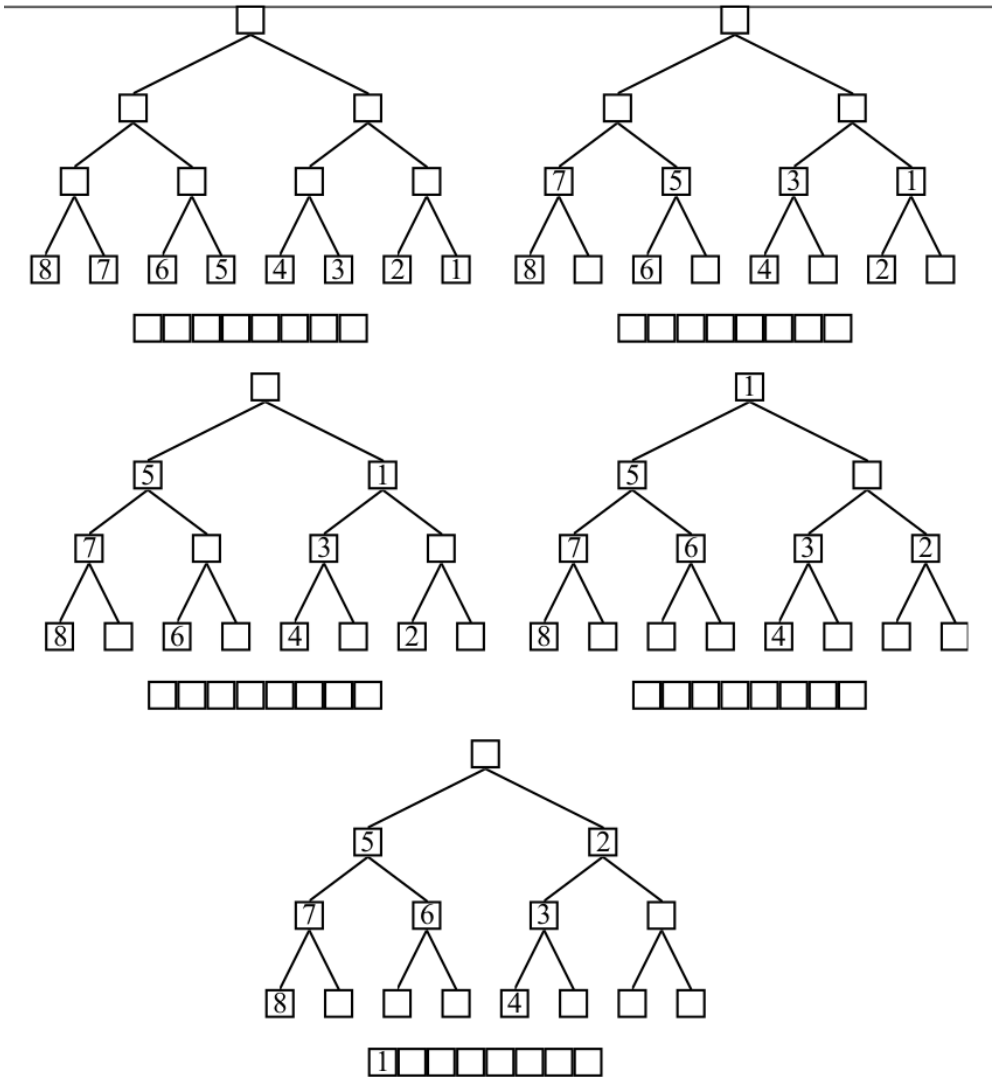
- Stromová architektura
- Každý procesor se stará o jeden prvek

## Algoritmus

- Každý list obsahuje jeden prvek

- Každý nelistový procesor porovná hodnoty svých dvou synů a menší z nich pošle svému otci po (log n) + 1 krocích se minimální prvek dostane do kořenového procesoru
- Každým dalším krokem se získá další nejmenší prvek

## Příklad



## Analýza

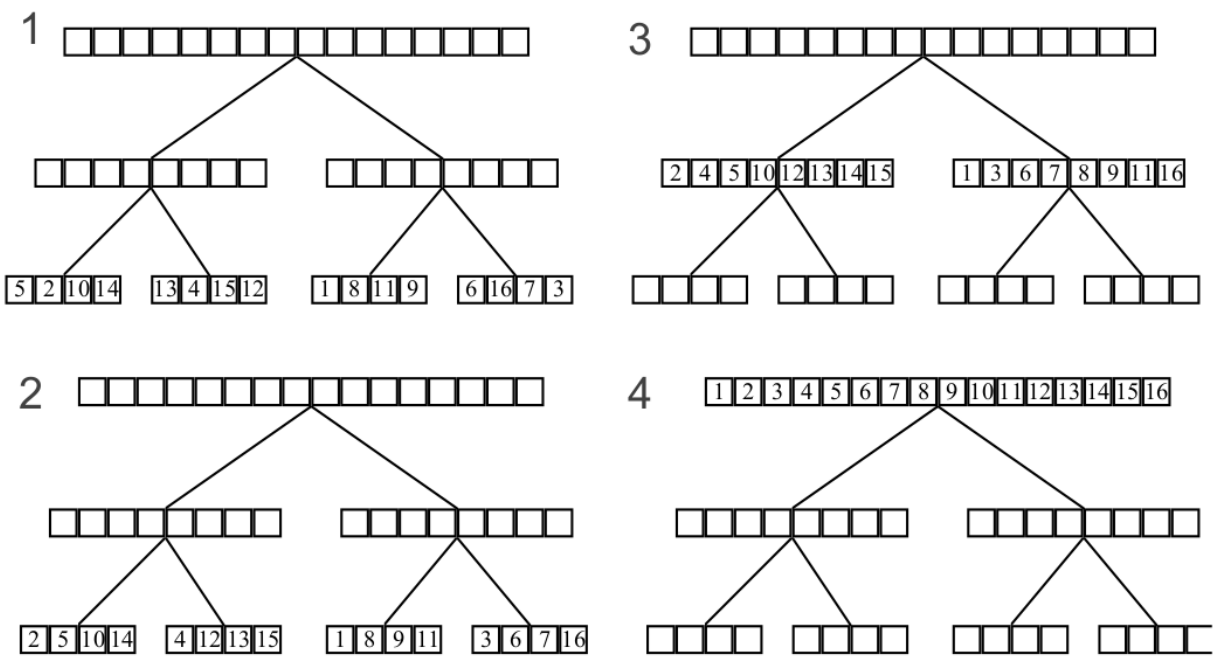
- Jelikož strom má (log n)+1 úrovní, první prvek se získá po (log n)+1 krocích. Kořenový procesor potřebuje jeden krok na porovnání a jeden na uložení výsledku do paměti. Každý ze zbylých n-1 prvků spotřebuje 2 kroky.
- t(n) = 2n + (log n) - 1 = O(n)

- p(n) = 2n - 1
- c(n) = t(n)*p(n) = O(n^2)
- což není optimální

## Bucket Sort

- Stromová architektura
- Jeden procesor se stará o více než jeden prvek
- Fáze předpřípravy, seřazení optimálním sekvenčním algoritmem
    - viz merge sort

### Příklad




### Analýza

- t(n) = O(n)
- p(n) = O(log n)
- c(n) = O(n*log n)
- optimální

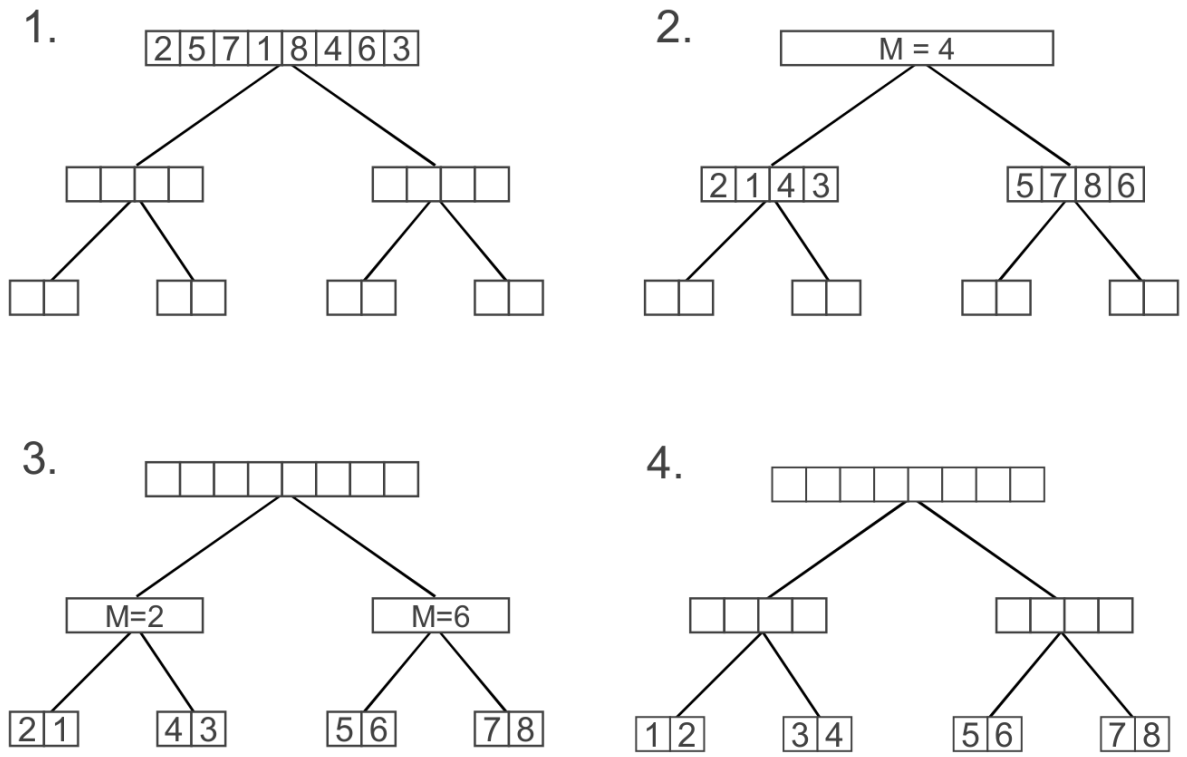
## Median Finding and Splitting

- Stromová architektura
- Jeden procesor se stará o více než jeden prvek

- Jdeme opačným směrem, od kořene k listům

## Algoritmus

- Uzel najde medián a všechny menší prvky pošle levému synovi a všechny větší pravému.
- Pokračuju do nějakého počtu, poté všichni seřadí zbytek optimálním sekvenčním algoritmem




Algoritmus nalezení mediánu není úplně triviální.

## Analýza

- t(n) = O(n)
- p(n) = O(log n)
- c(n) = O(n.log n)
- což je optimální

Shrnutí

# Řazení na SIMD bez společné paměti

| | t(n) | cena optimální? |
|---|---|---|
| **Speciální topologie** | | |
| Enumeration Sort | O(log n) | N |
| Odd-even Marge Sort | $\log^2 n$ | N |
| **Lineární pole procesorů** | | |
| Odd-Even Trasposition | n | N |
| Merge-splitting Sort (agregovaná verze předchozího) | | A |
| Pipeline Merge Sort | n | A |
| Enumeration Sort | n | N |
| **Mřížka (mesh)** | | |
| Mesh Sort | $n^{1/2}$ | N |
| Agregable Mesh Sort | | A |
| **Strom** | | |
| Minimum Extraction | n | N |
| Bucket Sort agregovaná verze předchozího | | A |
| Median Finding and Splitting | n | A |
| **Hyperkostka** | | |
| Cube Sort | | |
| $t(n) = O(\log n) .. O(\log^2 n)$ | | |
| $p(n) = n^2 .. 2n$ | | |

# 4) Algoritmy mediánu (select)

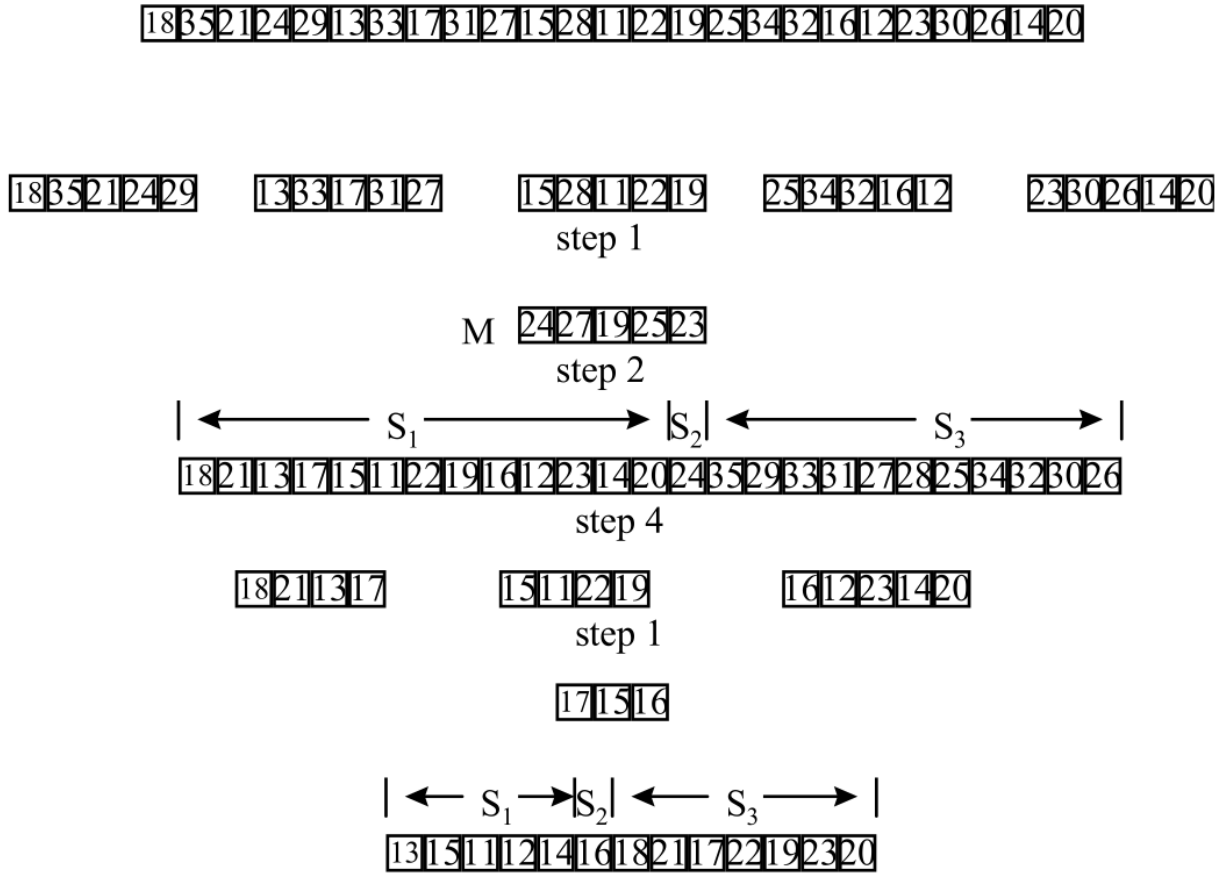
## Sequential select

- Hledá k-tý nejmenší prvek v posloupnosti S
- Je-li k = |S| / 2, jde o medián

```
Algoritmus
procedure SEQUENTIAL_SELECT(S, k)
(1) if |S| ≤ Q then seřaď S a odpočítej
       else rozděl S na |S|/Q posloupností Si o délce Q prvků
(2) // Seřaď každou posloupnost Si a nalezni její medián M[i]
     for i=1 to |S|/Q do
        M[i] = SEQUENTIAL_SELECT(Si, |Si|/2)
     end for
(3) // Nalezni "medián mediánů" m
    m = SEQUENTIAL_SELECT(M, |M|/2)
(4) L =  {si ∈ S: si < m}
    E =  {si ∈ S: si = m}
    G = { si ∈ S: si > m}
(5) if |L| > k then SEQUENTIAL_SELECT(L,k) // prvek musí být v L
    else if |L| + |E| > k then return m    // prvek musí být v E
    else SEQUENTIAL_SELECT(G, k-|L|-|E|)   // prvek musí být v G
```

- Pro Q>=5 t(n) = O(n)

Příklad


18 35 21 24 29 13 33 17 31 27 15 28 11 22 19 25 34 32 16 12 23 30 26 14 20
18 35 21 24 29
13 33 17 31 27
15 28 11 22 19
25 34 32 16 12
23 30 26 14 20
step 1
M
24 27 19 25 23
step 2
S1
S2
S3
18 21 13 17 15 11 22 19 16 12 23 14 20 24 35 29 33 31 27 28 25 34 32 30 26
step 4
18 21 13 17
15 11 22 19
16 12 23 14 20
step 1
17 15 16
S1
S2
S3
13 15 11 12 14 16 18 21 17 22 19 23 20

## Parallel select

- Hledá k-tý nejmenší prvek v posloupnosti S
- EREW PRAM s N procesory $P_1 \ldots P_N$
- Používá sdílené pole M o N prvcích

```
Algoritmus
procedure PARALLEL_SELECT(S, k)
(1) if |S| ≤ 4 then přímo nalezni k-tý prvek
    else rozděl S na N posloupností Si o délce n/N a každou přiřaď
          jednomu procesoru Pi
(2) for i=1 to N do in parallel
       M[i] = SEQUENTIAL_SELECT(Si, |Si|/2)
     end for
(3) m = PARALLEL_SELECT(M, |M|/2)  ← s menším počtem procesorů
(4) L =  {si ∈ S: si < m}
    E =  {si ∈ S: si = m }
    G = { si ∈ S: si > m}
(5) if |L| > k then PARALLEL_SELECT(L,k)
       else if |L| + |E| > k then return m
              else PARALLEL_SELECT(G, k-|L|-|E|)
```

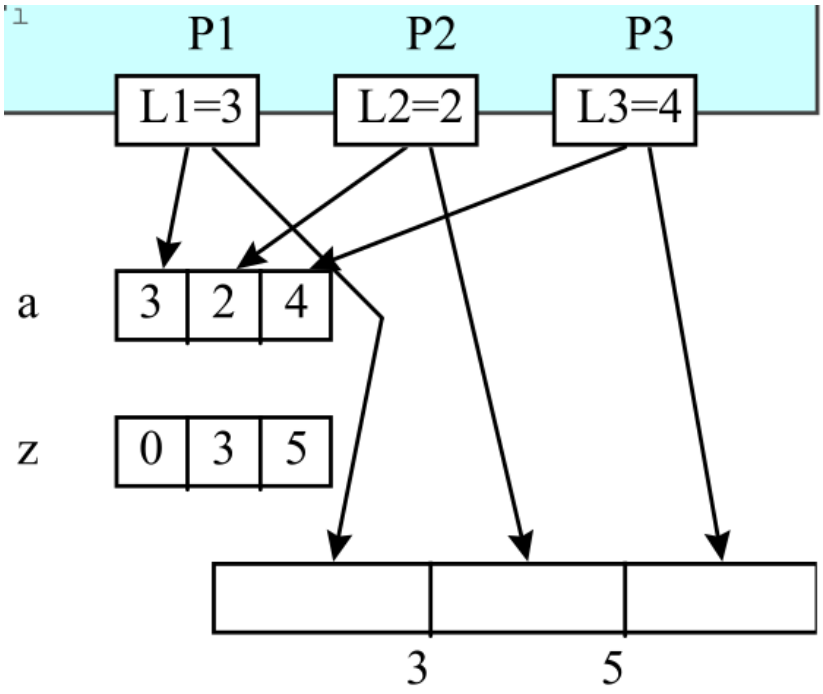
## Analýza

- t(n) = O(n/N) pro n > 4, N < n/log n
- p(n) = N
- c(n) = t(n) * p(n) = O(n)
- což je optimální

## Parallel splitting

- Krok 4 algoritmu Parallel select
- Úloha: Je dána posloupnost S a číslo m
  Mají se vytvořit tři posloupnosti:
  $L = \{s_i \in S: s_i < m\}$
  $E = \{s_i \in S: s_i = m\}$
  $G = \{s_i \in S: s_i > m\}$
- Složitost sekvenčního algoritmu je O(n)
- Paralelní řešení - máme N procesorů, které si sekvenci S rozdělí na podposloupnosti $S_i$ o délce n/N

## Příklad




## Analýza

- t(n) = O(log N + n/N) = O(n/N) pro dostatečně malé N
- Cena: c(n) = O(n/N) * N = O(n)
- což je optimální

# 5) Algoritmy vyhledávání

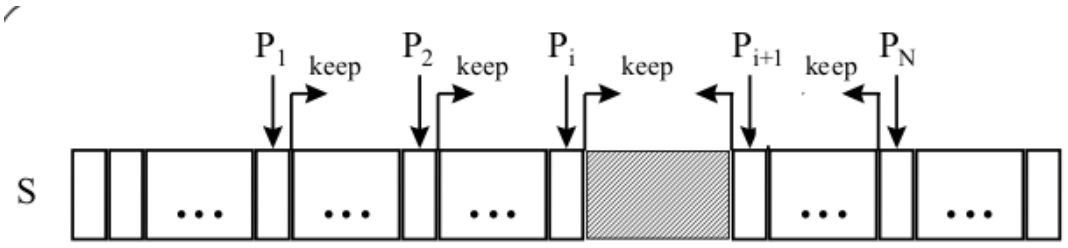
- Zjišťujeme zda prvek je v posloupnosti, případně na jakém indexu.
- Varianty
  - Posloupnost je seřazená
  - Posloupnost není seřazená

## N-ární search

- Vyhledává se seřazené posloupnosti

### Princip

- Při binárním vyhledávání zjistíme, ve které polovině se prvek nachází
- Při n-árním vyhledávání zjistíme, ve které z N+1 částí, se prvek nachází, kde N je počet procesorů




- Každý procesor se podívá na svoji “vlaječku” a na “vlaječku” svého pravého souseda
- Ten který detekuje změnu, zapíše svoji pozici a pozici svého pravého souseda – získáme blok je kterém je prvek
- Celý proces opakuju, dokud vyhledávací část není “akorát” malá, pak sekvenčně najdu

### Analýza

- t(n) = O(log(n+1) / log(N+1)) = O(log_N+1 (n+1))
- c(n) = O(N * log_N+1 (n+1))
- což není optimální

## Unsorted search

- Vyžaduje architekturu se sdílenou pamětí (PRAM)

- Vyhledává prvek v neseřazené posloupnosti

## Princip

- Mám N procesorů, n prvků a hledaný prvek x
- Každý procesor si přečte hledaný prvek
- n prvků je rozděleno mezi N procesorů a provede se sekvenční search
- Do sdílené proměnné pak ten kdo našel zapíše index prvku, případně nedefinovaná hodnota

## Analýza

### Architektura EREW – Exclusive read exclusive write

- 1. krok = O(log n)
- 2. krok = O(n/N)
- 3.krok = O(log N)
- t(n) = O(log N + n/N)
- c(n) = O(N * log N + n)

### Architektura CREW – Concurrent read exclusive write

- 1. krok = O(1)
- 2. krok = O(n/N)
- 3.krok = O(log N)
- t(n) = O(log N + n/N)
- c(n) = O(N * log N + n)

### Architektura CRCW – Concurrent read concurrent write

- 1. krok = O(1)
- 2. krok = O(n/N)
- 3.krok = O(1)
- t(n) = O(n/N)
- c(n) = O(n)  což je optimální

## Tree search

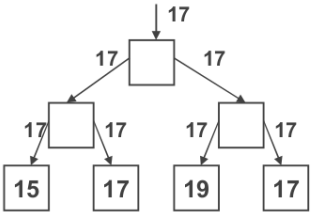
- Vyhledává v neseřazené posloupnosti
- Stromová architektura s 2n-1 procesorů

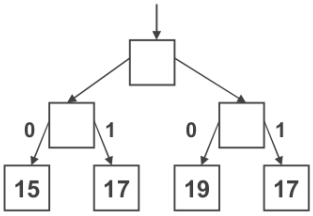
## Algoritmus

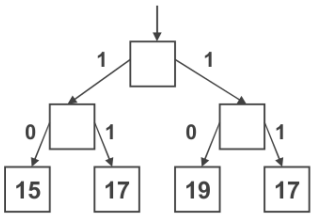
1. Kořen načte hledanou hodnotu x a předá ji synům ... až se dostane ke všem listům
2. Listy obsahují seznam prvků, ve kterých se vyhledává (každý list jeden). Všechny listy paralelně porovnávají x a x i , výsledek je 0 nebo 1.
3. Hodnoty všech listů se předají kořenu každý ne list spočte logické or svých synů a výsledek zašle otci. Kořen dostane 0 - nenalezeno, 1- nalezeno

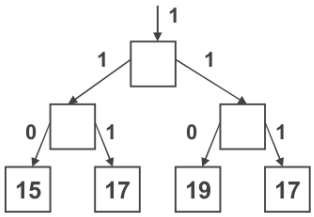
## Analýza

- Krok (1) má složitost O(log n), krok (2) má konstantní složitost, krok (3) má O(log n).
- t(n) = O(log n)
- p(n) = 2.n-1
- c(n) = t(n).p(n) = O(n.log n)
    - což není optimální



# Kapitola 44

# BMS – Bezdrátové lokální sítě (Wifi, Bluetooth).

## 44.1 Zdroje

- `08-bezdratove_lan.pdf`
- `BMS_2020-12-02.mp4`
- `BMS_2020-12-09.mp4`

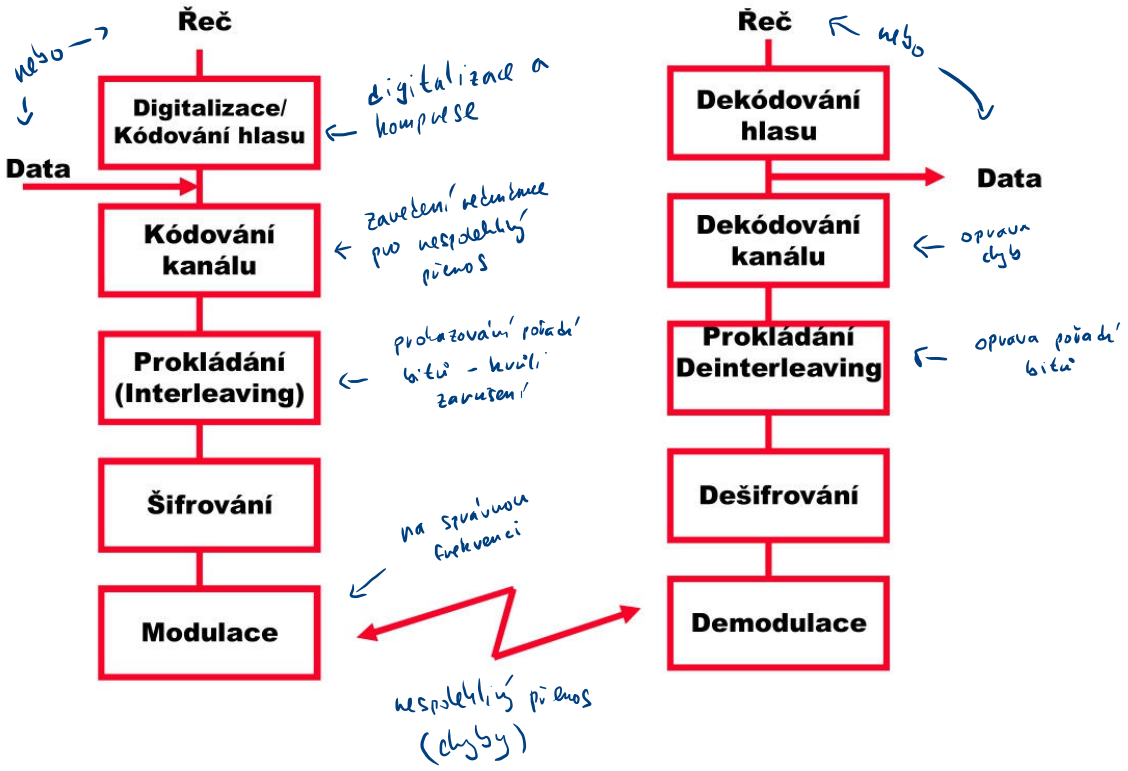
## 44.2 Bezdrátová komunikace

- Radiové vlny je elektromagnetické vlnění ve frekvenčním rozsahu 300 GHz – 30 Hz.
- Bitová rychlost – Počet přenesených bitů za časovou jednotku.
- Baudová rychlost – Počet přenesených signálových jednotek za časovou jednotku (baud rate $\leq$ bit rate).
- Princip bezdrátové komunikace a jednotlivé fáze přenosu:

1. **Kódování hlasu a digitalizace**
   - Z analogového hlasu uděláme digitální signál.
   - Velká redundance, provádíme kompresy.
   - Dojde ke ztrátě nějakých informací a přidání šumu (kvantizační šum) při dekódování.
   - Např. pulzně kódová modulace (PCM, *pulse-code modulation*).
2. **Kódování kanálu**
   - Data jsou přenášena přes rádiové kanály, které jsou nespolehlivé (data se mohou ztratit, bit může „přeskočit“, ...).
   - Počítáme s tím, zavádíme redundanci tak, aby bylo možné většinu chyb opravit.
     - Chceme minimalizovat opětovné přenášení (to je problém).
     - Detekce chyb pomocí BER – Bit Error Rate, FER – Frame Error Rate.
   - Zvyšuje se tím výrazně objem dat.
   - Metody:

  - Blokové korekční kódy (Hammingův kód) – vstup je rámec, výstup je původní rámec + zabezpečovací bity.
  - Konvoluční korekční kódy – nezná pojem rámec, vstupují bity a vystupují bity rychleji.
  - Turbo korekční kódy.

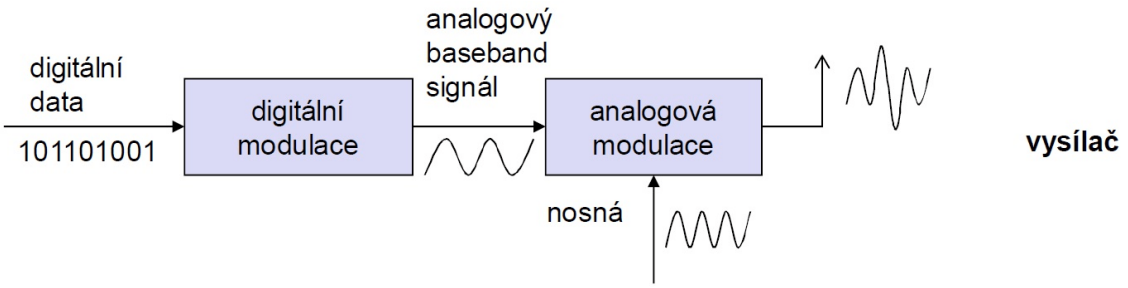
3. **Prokládání (*interleaving*)**
   - Pokud se na kanálu objevuje nějaké rušení, tak je většinou impulsní a zaruší se několik bitů vedle sebe.
   - To se nelíbí mechanismům kódování kanálu, ty mají rády pokud je zarušen „sem tam“ nějaký bit, ale ne shluk vedle sebe.
   - Zabraňujeme tomu tak, že prohazujeme pořadí jednotlivých bitů.
4. **Šifrování dat**
   - Může být v různé fázi, není shoda.
   - Umístění šifrování tak jak ve schématu je problém, jelikož se mohou objevit chyby díky přenosu, což se většine šifrovacích algoritmů nelíbí, proto by dávalo smysl to umístit až za (de)kódování kanálu.
5. **Modulace**
   - Dostat náš signál do analogové podoby pro přenos a na potřebnou frekvenci (úprava frekvencí, amplitud, fází).
   - Také prováděn multiplexing (přenášení více signálů přes jednu anténu).
6. **Přenos signálu přes telekomunikační síť**
   - Jak bylo řečeno výše, jde o nespolehlivý přenos přes rádiové (nebo historicky infračervené) záření.
7. **Demodulace**
   - Dostat signál z podoby vhodné pro přenos na podobu vhodnou pro dekódování (úprava frekvencí, amplitud, fází).
   - Prováděn demultiplexing.
8. **Dešifrování**
   - Dešifrování, platí to stejné jako u šifrování.
9. **Prokládání (*deinterleaving*)**
   - Překládání bitů analogicky k prokládání na druhé straně.
10. **Dekódování kanálu**
    - Opravení chyb.
11. **Dekódování hlasu**
    - Pokud chceme řeč, tak převod do analogové podoby.




Obrázek 44.1: Schéma bezdrátové komunikace.

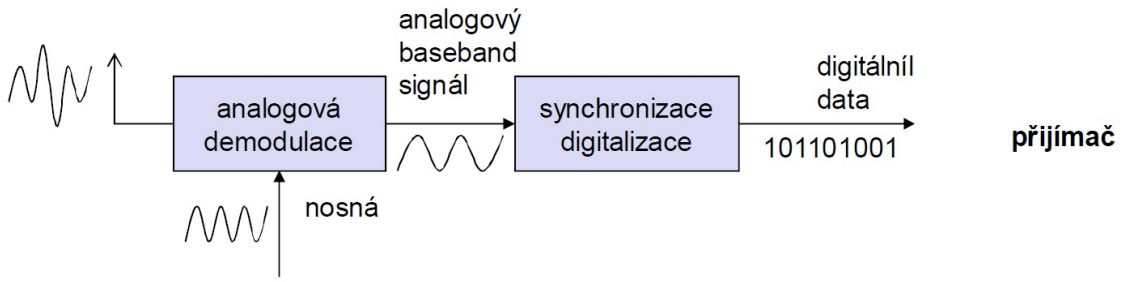
## 44.3 Modulace

- Na straně vysílače:
  - **Digitální modulace** (klíčování) – Vstupní (digitální) signál, je třeba převést na analogový.
  - **Analogová modulace** – Analogový signál je třeba dostat na správnou frekvenci pro přenos (kombinace s nosným signálem).




Obrázek 44.2: Digitální a analogová modulace.

- Na straně přijímače:
  - **Analogová demodulace** – Oddělení nosného signálu.
  - **Synchronizace a digitalizace** – Převod analogového signálu na digitální.




Obrázek 44.3: Analogová demodulace a digitalizace.

### 44.3.1 Analogová (de)modulace

- Vstup: analog, výstup: analog.
- Amplitudová modulace (AM, *amplitude modulation*).

$$f(x) = C \cdot \sin(x) \ , \ C \in \mathbb{R}$$

- Frekvenční modulace (FM, *frequency modulation*).

$$f(x) = \sin(C \cdot x) \ , \ C \in \mathbb{R}$$

- Fázová modulace (PM, *phase modulation*).

$$f(x) = \sin(x + C) \ , \ C \in \mathbb{R}$$

### 44.3.2 Digitální (de)modulace

- Vstup: digitál, výstup: analog.
- Cíl: v jedné sinusovce (signálová jednotka) přenést co nejvíce bitů.
- Amplitudová modulace (ASK, *amplitude shift keying*).
- Frekvenční modulace (FSK, *frequency shift keying*).
- Fázová modulace (PSK, *phase shift keying*).
  - 2-PSK, 4-PSK, 8-PSK
- Kvadraturní amplitudová modulace (QAM, *quadrature amplitude modulation (shift keying)*).
  - 8-QAM, 16-QAM, 32-QAM, 64-QAM
- Pro ASK, FSK, 2-PSK: baud rate = bit rate.

## 44.4 Sdílení spektra

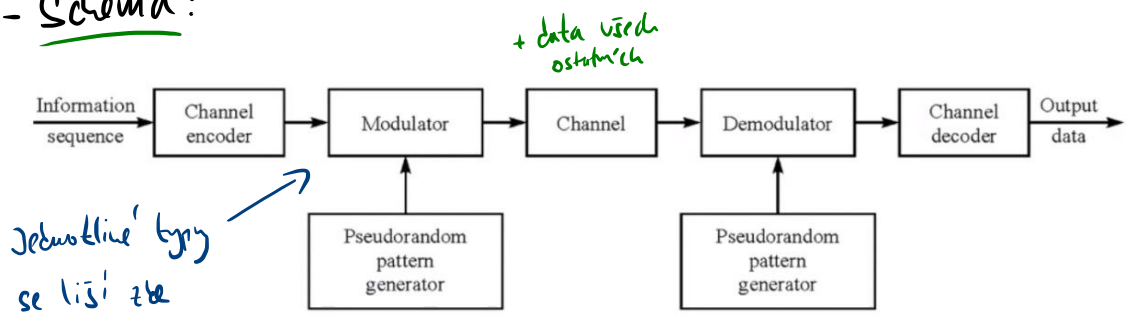
- Jak umožnit více než dvěma koncovým zařízením připojeným ke stejnému přenosovému médiu komunikovat tak, aby se vzájemně nerušili?
  - Metoda přístupu ke kanálu (*channel access method*).
- Metoda přístupu ke kanálu je založena na sdílení spektra (multiplexing), které umožňuje několika datovým tokům nebo signálům sdílet stejný komunikační kanál nebo přenosové médium. V této souvislosti multiplexování zajišťuje fyzická vrstva.

- Mnoho přístupů:
  1. **Prostorový multiplex** (SDMA, *space-division multiple access*)
     - Jsme schopni využít 1 frekvenci pro více účastníků, jsou-li fyzicky daleko od sebe.
     - Je potřeba ochranný interval.
  2. **Frekvenční multiplex** (FDMA, *frequency-division multiple access*)
     - Rozdělení spektra na menší úseky (kanály), které poté přidělujeme účastníkům.
     - Kanál má 2 atrbitu: šířku a střední frekvenci.
     - Nevýhoda: plýtvání pásmem, nutnost ochranných intervalů.
  3. **Časový multiplex** (TDMA, *time-division multiple access*)
     - Kanál dostane celé spektrum po určitý čas.
     - Nelze použít v analogovém vysílání.
     - Nutná synchronizace.
     - Jedna nosná, vysoká propustnost.
  4. **Časový a frekvenční multiplex** (FTDMA, *frequency and time division multiple access*)
     - Kombinace časového a frekvenčního multiplexingu.
     - Každý účastník dostane kanál a časový slot (kde a kdy komunikuje).
     - Využívá se v GSM.
     - Nutná časová i frekvenční koordinace.
  5. **Kódový multiplex** (CDMA) – viz dále.
  6. **Ortogonální multiplex s frekvenčním dělením** (OFDM)– viz dále.

### 44.4.1 Kódový multiplex (CDMA)

- Všichni vysílají na jednom kanálu po celou dobu.
  - Nesnažíme se bránit vzájemnému zarušení, to bereme jako součást systému.
- Všichni dostávají signál všech a musí si extrahovat informace pro sebe.
  - signál všech → algoritmus → signál účastníka
  - Každá stanice má své unikátní číslo, pokud přijímač toto číslo zná, může se „naladit“ (z čísla derivujeme „náhodný“ signál).
- Nevýhoda:
  - složitost přijímače (potřeba počítač pro kódování / dekódování),
  - nižší rychlost přenosu.
- Výhoda:
  - efektivní využití pásma (žádné ochranné intervaly),
  - žádná koordinace a synchronizace,
  - odolnost proti rušení a odposlouchávání.

- Schéma:

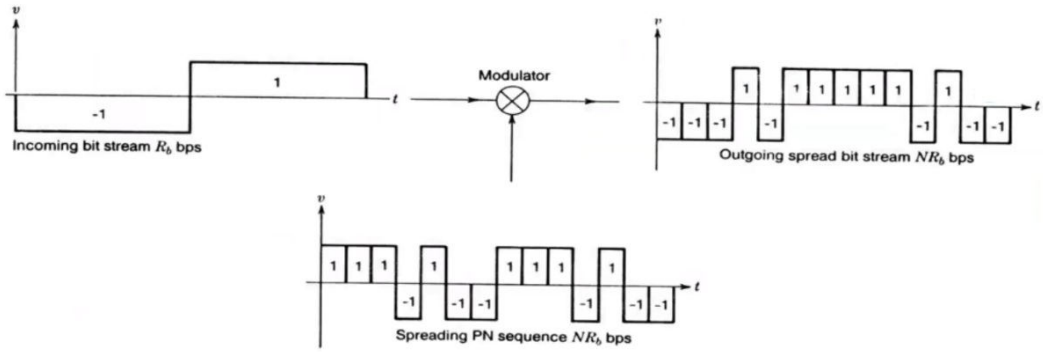



Obrázek 44.4: CDMA schéma.

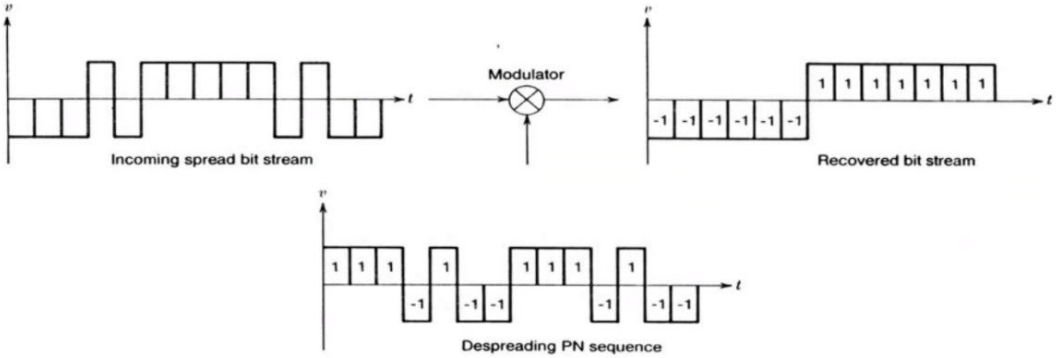
**DSSS** (***direct sequence spread spectrum***)

- „Rozprostřené spektrum“.
- Užitečný signál $\oplus$ (XOR) vygenerovaný náhodný signál.
- Výsledek se namoduluje a posílá.

Na straně odesílatele:




Na straně příjemce:




Obrázek 44.5: DSSS.

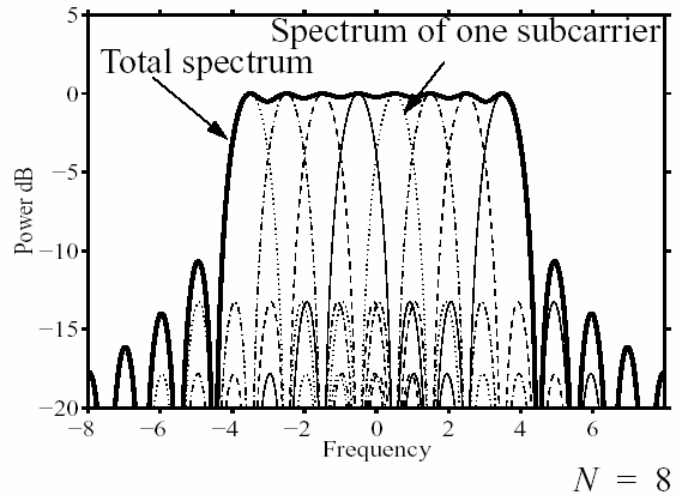
**FHSS** (***frequency hopping spread spectrum***)

- Uživatel má k dispozici několik frekvenčních kanálů.

- Náhodně vygenerovaná sekvence definuje, kdy se komunikuje přes jaký frekvenční kanál. – Frekvenční kanály se v průběhu komunikace střídají.

### 44.4.2 Ortogonální multiplex s frekvenčním dělením (OFDM)

- Ortogonální multiplex s frekvenčním dělením (OFDM, *orthogonal frequency division multiplexing*)
- Kombinace a FDMA multiplexingu a QAM modulace.
  - Datový tok celého kanálu se tak dělí na stovky dílčích datových toků jednotlivých subnosných.
  - Vzniká mnoho úzkých subkanálů, které jsou přenášeny paralelně.
- Subnosné frekvence jsou dále modulovány dle potřeby různě robustními modulacemi (QPSK, 16-QAM, 64-QAM).
- Jednotlivé subnosné jsou vzájemně ortogonální, takže maximum každé nosné by se mělo překrývat s minimy ostatních.
- Umožňuje, aby se postranní pásma subkanálů překrývala, což šetří pásmo.
- Přijímač je schopen oddělit překrývající se subkanály, protože jsou ortogonální.




Obrázek 44.6: Subkanály OFDM, pouze krajní se nemohou překrývat.

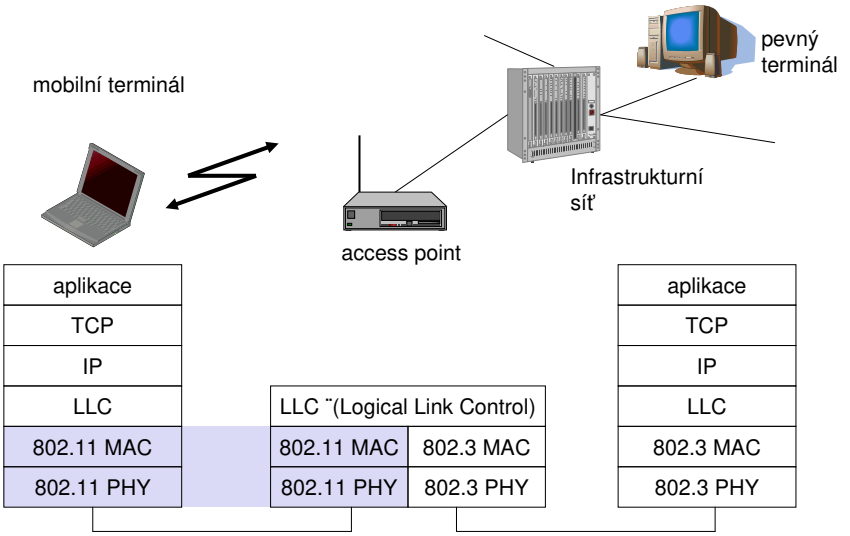
## 44.5 Bezdrátové lokální sítě

- **Výhody**
  - Velmi flexibilní – nemusíme natahovat dráty.
  - Možnost ad-hoc sítí bez předchozího plánování.
  - Téměř žádné problémy s vedením (historické budovy, požární přepážky).
  - Robustnější vůči nehodám (nehrozí přerušení vodičů).
- **Nevýhody**
  - Nižší přenosová rychlost oproti drátovým sítím (1-10 Mbit/s).

    - Máme pouze omezené frekvence, na kterých můžeme přenášet data.
  - Mnoho proprietárních řešení, standardizace trvá nějakou dobu (např. IEEE 802.11).
  - Produkty musí dodržovat mnoho různých omezení pro bezdrátové přístroje (frekvenční plánování, výkony, homologace, ...), vytvořit globální řešení trvá delší dobu.
- **Cíle návrhu**
  - Globální provoz.
    - Historicky problémy vzhledem k různým legislativám ohledně rádiového vysílání v jednotlivých státech.
  - Nízká spotřeba pro provoz z baterií.
  - Bezpečnost – Bezdrátové sítě se snáze odposlouchávají, než drátové, proto je šifrování nutnost.
  - Transparentnost k vyšším protokolům.
    - Např. TCP/IP neví jestli běží po drátové/bezdrátové technologii.
- Jako fyzické médium se používá **rádiový přenos** (typicky bezlicenční pásmo 2,4 GHz), ačkoliv historicky byl ve hře i infračervený přenos.
- **Adhoc sítě**
  - Decentralizovaný typ bezdrátové sítě, kde nezávisí na předem existující infrastruktuře, jako jsou směrovače v kabelových sítích nebo *access pointy* v bezdrátových sítích.
  - Jako ad hoc síť se typicky označuje množina sítí, kde jsou všechna zařízení rovnocenná a volně spojitelná s jakoukoliv jinou ad hoc sítí v dosahu.
  - Určité uzly podílí na předávání dat jiným uzlům a jejich určení je prováděno dynamicky na základě síťové konektivity.
- **Infrastrukturní síť**
  - Máme předem danou infrastrukturu (směrovače, *access pointy*), ke které se připojují klienti.

## 44.6 WiFi

- WiFi (*wireless local area network*), několik verzí standardu IEEE 802.11.
- Specifikuje fyzickou vrstvu ($L_0$) a vrstvu síťového rozhraní ($L_1$) v TCP/IP stacku, resp. ISO/OSI.
- Fyzická vrstva – fyzické médium (rádiové, infračervené vysílání) + vrstva konvergence, která zapouzdřuje médium.
- Vrstva síťového rozhraní – přístup k fyzickému médiu, fragmentace, šifrování, detekce a korekce chyb.




Obrázek 44.7: IEEE 802.11 WiFi architektura, PHY značí fyzickou vrstvu, MAC síťového rozhraní.

### 44.6.1 IEEE 802.11

- Pásmo: 2,4 GHz.
- Maximální rychlost: 2 Mbitps.
- Fyzická vrstva:
  - rádiové kanály, multimplexing CDMA (*code division multiple access*), konkrétně DSSS (*direct sequence spread spectrum*) – přežilo,
  - rádiové kanály, CDMA, konkrétně FHSS – zahynulo.
  - infračervené kanály – zahynulo.
- Fyzická vrstva, která přežila, má 11 kanálů o šířce 22 MHz, které se překrývají (netypické).

### 44.6.2 IEEE 802.11a

- Americký nástupce (použití pouze USA) 802.11.
- Frekvenční pásmo: 5,0 GHz.
- Maximální rychlost: 54 Mbitps
- Maximální dosah: 100 m mimo budovy, 10 m v budově.
- Cíl:
  - přidělení nového frekvenčního pásma, které není tak vytížené (není zpětná kompatibilita),
  - dosáhnutí vyšší rychlosti,
  - odstranění infračervených a rádiových FHSS kanálů.
- Jak dosahuje zrychlení?
  - Díky tomu, že není zpětně kompatibilní, může používat nové sdílení spektra OFDM.

- Modulace: 2-PSK, 4-PSK, 16-QAM, 64-QAM.
- Zhodnocení:
  - není zpětně kompatibilní,
  - je rychlé,
  - má menší dosah (kvůli vyšší frekvenci, která má vyšší útlum).

### 44.6.3 IEEE 802.11b

- Evropský nástupce (použití celosvětově) 802.11.
- Frekvenční pásmo: 2,4 GHz.
- Maximální rychlost: 11 Mbitps.
- Maximální dosah: 300 m mimo budovy, 30 m v budově.
- Cíl:
  - původní frekvenční pásmo,
  - dosáhnutí vyšší rychlosti,
  - odstranění infračervené a rádiové FHSS fyzické vrstvy,
  - zpětná kompatibilita (ale pouze 1 fyzická vrstva).
- Sdílení spektra DSSS (rozprostírací sekvence je tzv. Barkerův kód).
- Jak dosahuje zrychlení?
  - Původní modulační schémata DBPSK, DQPSK pro rychlost 2 Mbitps.
  - Nové modulační schéma CCK pro 11 Mbitps.
- Zhodnocení:
  - je zpětně kompatibilní,
  - není tak rychlé jako 802.11a,
  - ale má větší dosah než 802.11a.

### 44.6.4 IEEE 802.11g

- Frekvenční pásmo: 2,4 GHz.
- Maximální rychlost: 54 Mbitps
- Rychlé a má velký dosah.
- Cíl:
  - všechno dobré z 802.11a importovat do 802.11b, tj. zachovat frekvenci 2,4 GHz,
  - sdílení spektra OFDM,
  - zpětná kompatibilita s 802.11b.

| Standard | 802.11g | 802.11b | 802.11a |
|---|---|---|---|
| **Podporované rychlosti** | 54, 48, 36, 24, 18, 12, 9, 6,11, 5.5, 2, 1 Mbps | 11, 5.5, 2, 1 Mbps | 54, 48, 36, 24, 18, 12, 9, 6 Mbps |
| **Maximální rychlost** | 54 Mbps | 11 Mbps | 54 Mbps |
| **Frekvence** | 2.4 GHz (2.4 GHz až 2.4835 GHz) | 2.4 GHz (2.4 GHz až 2.4835 GHz) | 5 GHz (5.725 GHz až 5.850 GHz) |
| **Kanály** | 3 nepřekrývající se, 13 překrývajících se | 3 nepřekrývající se, 13 překrývajících se | 12 nepřekrývajících se |
| **Modulace** | OFDM/CCK (6,9,12,18,24,36,48,54)<br>OFDM (6,9,12,18,24,36,48,54)<br>DQPSK/CCK (22, 33, 11, 5.5 Mbps)<br>DQPSK (2 Mbps)<br>DBPSK (1 Mbps) | DQPSK/CCK (11, 5.5 Mbps)<br>DQPSK (2 Mbps)<br>DBPSK (1 Mbps) | BPSK (6, 9 Mbps)<br>QPSK (12, 18 Mbps)<br>16-QAM (24, 36 Mbps)<br>64-QAM (48, 54 Mbps) |
| **Maximální dosah** | Do 300 m | Do 300 m | Do 150 m |
| **Zpětně kompatibilní s:** | 802.11b | N/A | N/A |
| **Charakteristika** | Náhrada 802.11b s vyšší rychlostí a lepší bezpečností | Dnes nejrozšířenější | Ideální pro vysoké kapacity a rychlosti |

Obrázek 44.8: Srovnání standardů IEEE 802.11 a, b a g.

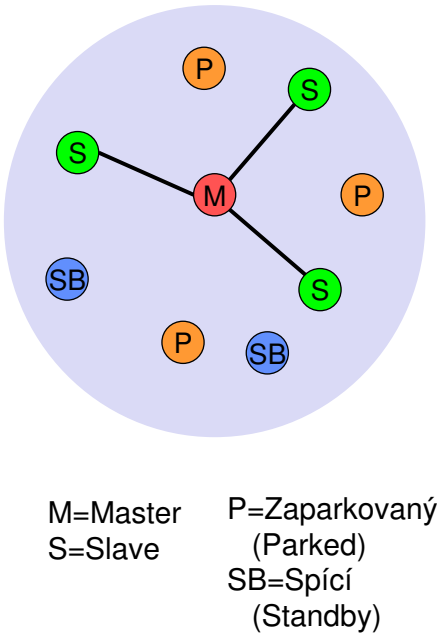
## 44.7 Bluetooth

- Bluetooth, *wireless personal area network*.
- Typický případ použití: komunikace mezi různými zařízeními stejného uživatele.
  - Propojení počítačů, periferie, mobilní telefon (náhrada kabelů).
- Univerzální rádiový prostředek pro bezdrátové propojení v ad-hoc sítích.
- Krátký dosah (10 m) a nízká spotřeba (bezlicenční pásmo 2,4 GHz).
- Požadavky: přenos hlasu a dat (datová rychlost cca. 1 Mbitps).
- Vlastnosti:
  - Frekvenční pásmo: 2,4 GHz (79 rádiových kanálů, vzdálenost 1 MHz).
  - Modulace: G-FSK (*gaussian frequency-shift keying*).
  - Multiplexing: FHSS (*frequency hopping spread spectrum*).
  - Datové přenosy – asynchronní, bez navázání spojení.
  - Hlasové přenosy – synchronní, navazování spojení, forward error correction, vadné pakety se neposílají znovu.
- Topologie: hvězdicová, „pikonet“ / „scatternet“.
- Zabezpečení:
  - PIN → šifrovací klíč (128 bit) → šifrovaná komunikace.
- Historicky problémy s kompatibilitou a párováním.

### 44.7.1 Pikonet

- Sbírka zařízení, které se nacházejí ve vzájemné blízkosti podle principu ad-hoc.
- Jedno z nich pracuje jako master a ostatní jako slave po celou dobu života pikonetu.

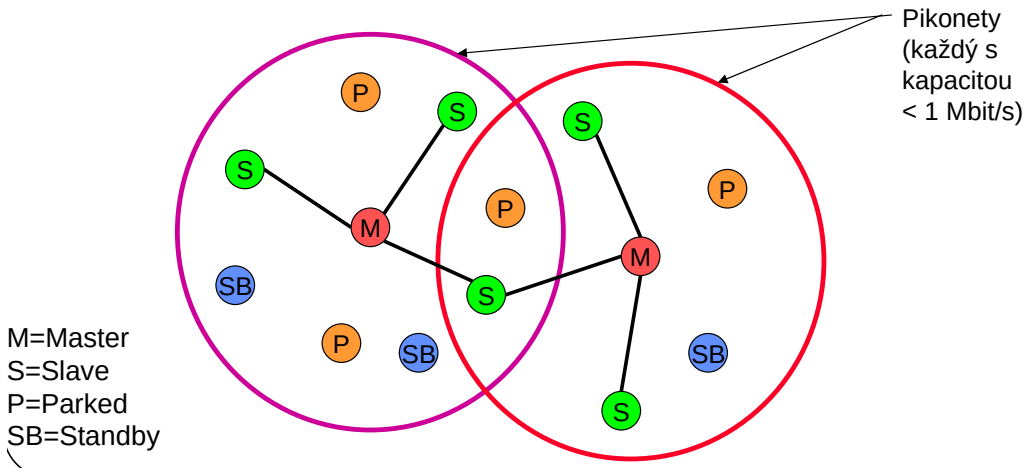
- Master určuje hopping sekvenci, slave se musí zasynchronizovat.
- Každý pikonet má svou hopping sekvenci.
- Uzel se stane účastníkem pikonetu tím, že se zasychnchronizuje na hopping sekvenci.
- Každý pikonet má jeden master uzel a v jednom okamžiku až 7 aktivních slave uzlů (ale více než 200 může být ve stavu „zaparkovaný“).
- Všechna zařízení v pikonetu mají stejnou hopping sekvenci.




Obrázek 44.9: Příklad pikonetu.

### 44.7.2 Scatternet

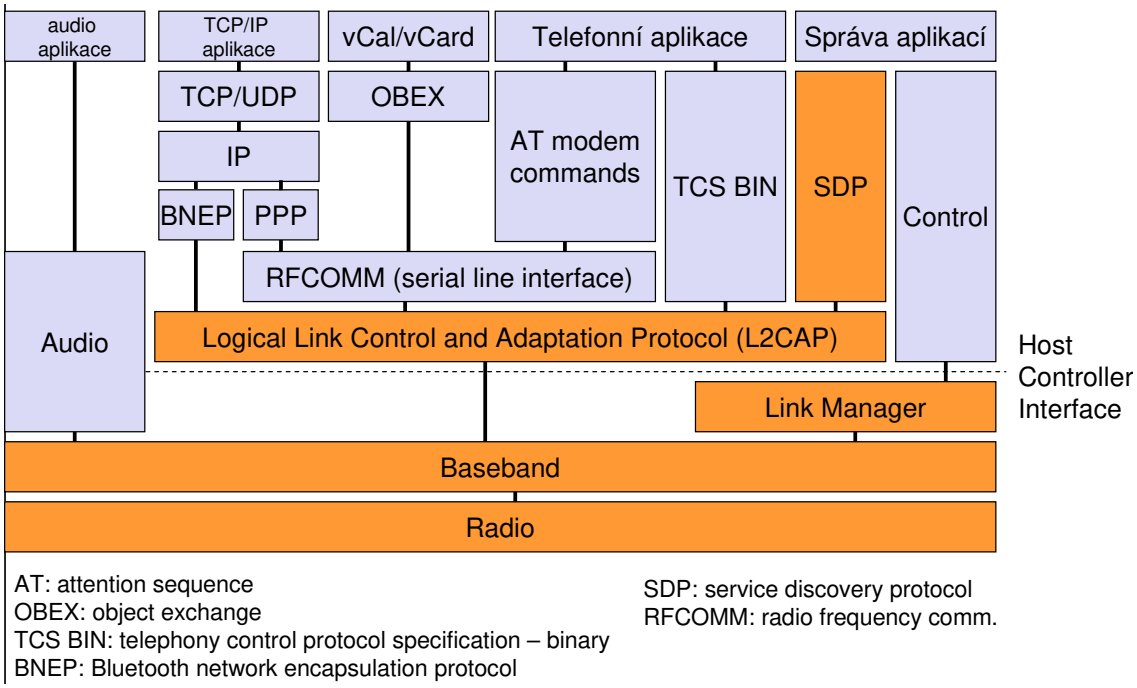
- Propojení několika blízkých pikonetů prostřednictvím sdílení uzlů master nebo slave.
  - Uzel může být slave v jednom pikonetu a master ve druhém.
- Komunikace mezi pikonety.
  - Uzel přeskakuje tam a zpátky mezi pikonety.
- Ne každý uzel se dokáže připojit do více pikonetů, většinou pouze laptop nebo smartphone.




Obrázek 44.10: Příklad scatternetu.

### 44.7.3 Protokol stack

- Service discovery protokol – Protokol pro zjištění nabízených služeb, hledá uzly v rádiovém dosahu.




Obrázek 44.11: Bluetooth protokol stack.

# Kapitola 45

# GAL – Hledání minimální kostry obyčejného grafu (pojmy, stromy a kostry, Kruskalův algoritmus, Primův algoritmus).

## 45.1 Zdroje

- `gal-handouts.pdf`
- `GAL_2020-10-22.mp4`
- `GAL_2020-10-29.mp4`

## 45.2 Úvod a kontext

**Orientovaný graf** Orientovaný graf je dvojice $G = (V, E)$, kde $V$ je konečná množina uzlů a $E \subseteq V \times V$ je množina hran.

**Neorientovaný graf** Neorientovaný graf je dvojice $G = (V, E)$, kde $V$ je konečná množina uzlů a $E \subseteq \binom{V}{2}$ je množina hran. (Hrana je tedy dvouprvková množina, avšak běžně se držíme stejného značení jako u orientovaných grafů a používáme dvojici.)

**Ohodnocený graf** Ohodnocený graf je takový graf, jehož každá hrana má přiřazenou nějakou hodnotu, typicky definovanou pomocí váhové funkce $w : E \mapsto \mathbb{R}$.

**Podgraf** Graf $G' = (V', E')$ je podgraf grafu $G = (V, E)$ jestliže $V' \subseteq V$ a $E' \subseteq E$.

**Sled** Posloupnost uzlů $\langle v_0, v_1, \ldots, v_k \rangle$, kde $(v_{i-1}, v_i) \in E$ pro $i = 1, \ldots, k$ se nazývá sled délky $k$ z $v_0$ do $v_k$.

**Uzavřený sled** Sled $\langle v_0, v_1, \ldots, v_k \rangle$ se nazývá uzavřený, pokud existuje hrana $(v_0, v_k)$.

**Dosažitelnost** Pokud existuje sled $s$ z uzlu $u$ do uzlu $v$, říkáme, že $v$ je dosažitelný z $u$ sledem $s$, značeno $u \overset{s}{\Rightarrow} v$.

**Tah** Tah je sled ve kterém se neopakují hrany.

**Cesta** Cesta je sled ve kterém se neopakují uzly.

**Souvislý graf** Neorientovaný graf se nazývá souvislý, pokud mezi libovolnými dvěma uzly existuje cesta.

**Kružnice** Uzavřená cesta se nazývá kružnice.

**Cyklus** Orientovaná kružnice se nazývá cyklus (první a poslední uzel je shodný).

**Prostý graf** Orientovaný graf bez cyklů se nazývá prostý.

**Acyklický graf** Graf je bez cyklů, resp. kružnic, se nazývá acyklický.

**Strom** Graf, který je souvislý a acyklický, se nazývá strom.

**Kostra** Strom, který tvoří podgraf souvislého grafu na množině všech jeho vrcholů, se nazývá kostra (*spanning tree*).

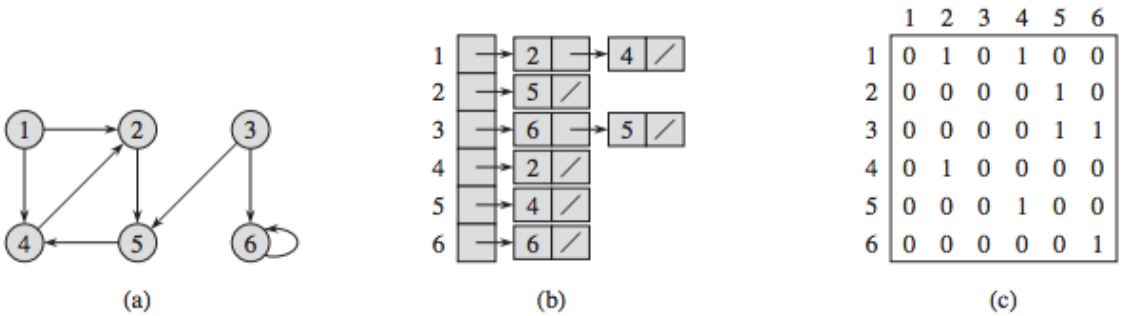
**Minimální kostra** Nechť $G = (V, E)$ je souvislý neorientovaný graf s váhovou funkcí $w : E \mapsto \mathbb{R}$. Minimální kostra (*MST, minimum spanning tree*) je strom $G' = (V, E')$, kde $E' \subseteq E$ a

$$w(E') = \sum_{(u,v)\in T} w(u, v)$$

je minimální ze všech možných alternativních koster.

**Seznam sousedů** Seznam sousedů (*Adj*, *adjacency list*) je reprezentace grafu v paměti. Jde o preferovanou variantu pro řídké grafy – kde $m << n^2$. Pro každý uzel máme definovaný seznam jeho sousedů.

**Matice sousednosti** Matice sousednosti (*adjacency matrix*) je reprezentace grafu v paměti. Jde o preferovanou variantu pro husté grafy – kde $m$ je skoro $n^2$.




Obrázek 45.1: Příklad reprezentace grafu pomocí seznamu sousedů a matice sousednosti.

## 45.3 Generický algoritmus

Hledání minimální kostry je problém, který lze řešit algoritmy, které spadají do kategorie tzv. hladových (*greedy*) deterministických algoritmů. Spočívají v tom, že průběžně odhadují kostru přidáváním dalších hran a nikdy se nemusejí vracet (neprovádí se *backtracking*). Generický algoritmus tvoří jakousi základní kostru pro další, už konkrétní, algoritmy.

**Řez** Nechť $G = (V, E)$ je graf. Řez grafu $G$ je dvojice $(S, V - S)$, kde $\emptyset \subseteq S \subseteq V$.

**Křížení** Hrana $(u, v) \in E$ kříží řez $(S, V - S)$, pokud jeden její konec je v $S$ a druhý v $V - S$.

**Respektování** Nechť $A \subseteq E$ je množina hran. Řez $(S, V - S)$ respektuje množinu hran $A$, pokud žádná hrana v $A$ nekříží řez $(S, V - S)$.

**Lehkost** Nechť $(S, V - S)$ je řez a $B$ je množina hran, která ho kříží. Hrana z množiny $B$ s nejmenší hodnotou se nazývá lehká.

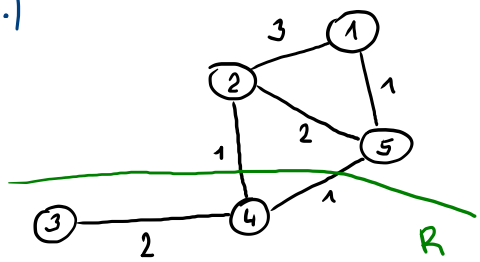
**Bezpečnost** Nechť $G = (V, E)$ je souvislý neorientovaný graf s reálnou váhovou funkcí $w$. Nechť $A \subseteq E$ je součástí nějaké minimální kostry $G$. Nechť $(S, V - S)$ je řez, který respektuje $A$. Nechť $(u, v)$ je lehká hrana křížící $(S, V - S)$. Pak hrana $(u, v)$ je bezpečná pro $A$.

```
1 def generic_mst(G):
2     # G je graf
3     A = {}# A je mnozina hran rozpracovane minimalni kostry
4     while netvori_kostru(A, G):
5         for hrana in G.E:
6             if je_bezpecna(A, hrana):
7                 A += {hrana}
8     return A
```

Výpis 45.1: Generický algoritmus. Před každou iterací algoritmu je množina $A$ podmnožinou nějaké minimální kostry. Hrana $(u, v) \in E$ je bezpečná pro $A$, pokud $A \cup \{(u, v)\}$ je podmnožinou nějaké minimální kostry.

Př. $G = (V, E)$ $\quad W: E \to \mathbb{R}$ (definované váhami)

1.)



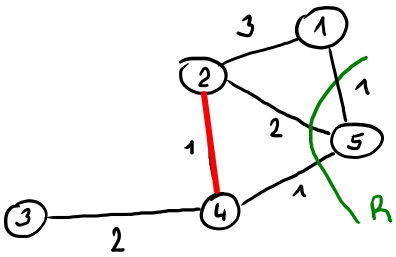

$A = \{\}$

$R = (\{1,2,5\}, \{3,4\})$

$LH = \{(2,4), (4,5)\}$

$A \leftarrow A \cup \{(2,4)\}$

2.)




$A = \{(2,4)\}$

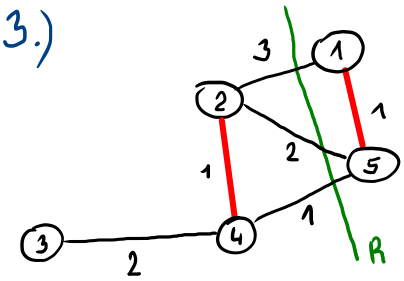
$R = (\{1,2,3,4\}, \{5\})$

$LH = \{(1,5), (4,5)\}$

$A \leftarrow A \cup \{(1,5)\}$

Obrázek 45.2: Příklad, část 1.

3.)



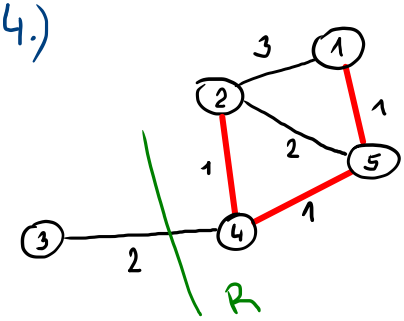

$A = \{(1,5), (2,4)\}$

$R = (\{1,5\}, \{2,3,4\})$

$LH = \{(4,5)\}$

$A \leftarrow A \cup \{(4,5)\}$

4.)



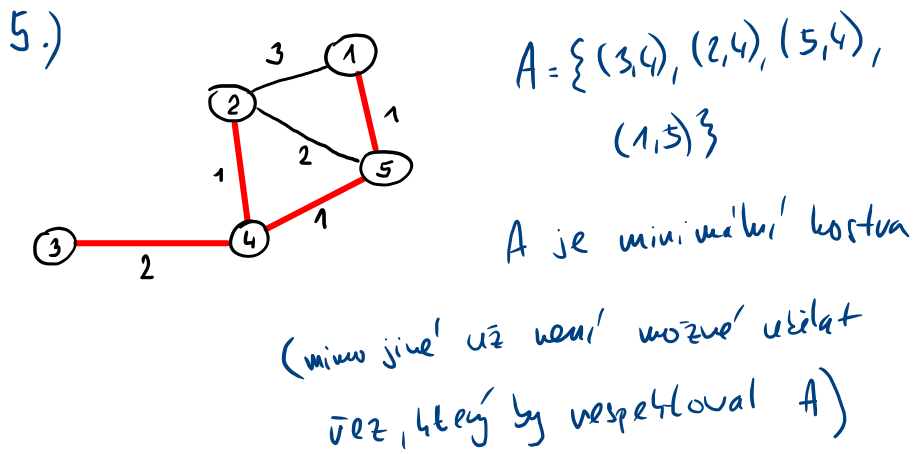

$A = \{(1,5), (2,4), (4,5)\}$

$R = (\{1,2,4,5\}, \{3\})$

$LH = \{(3,4)\}$

$A \leftarrow A \cup \{(3,4)\}$

Obrázek 45.3: Příklad, část 2.




Obrázek 45.4: Příklad, část 3.

## 45.4 Kruskalův algoritmus

Kruskalův a Primův algoritmus se liší v tom, jakým způsobem vybírají bezpečnou hranu. Kruskalův algoritmus nahlíží na $A$ jako na les a hledá hranu s nejmenším ohodnocením, která spojuje stromy v lese. Na konci je $A$ jeden strom.

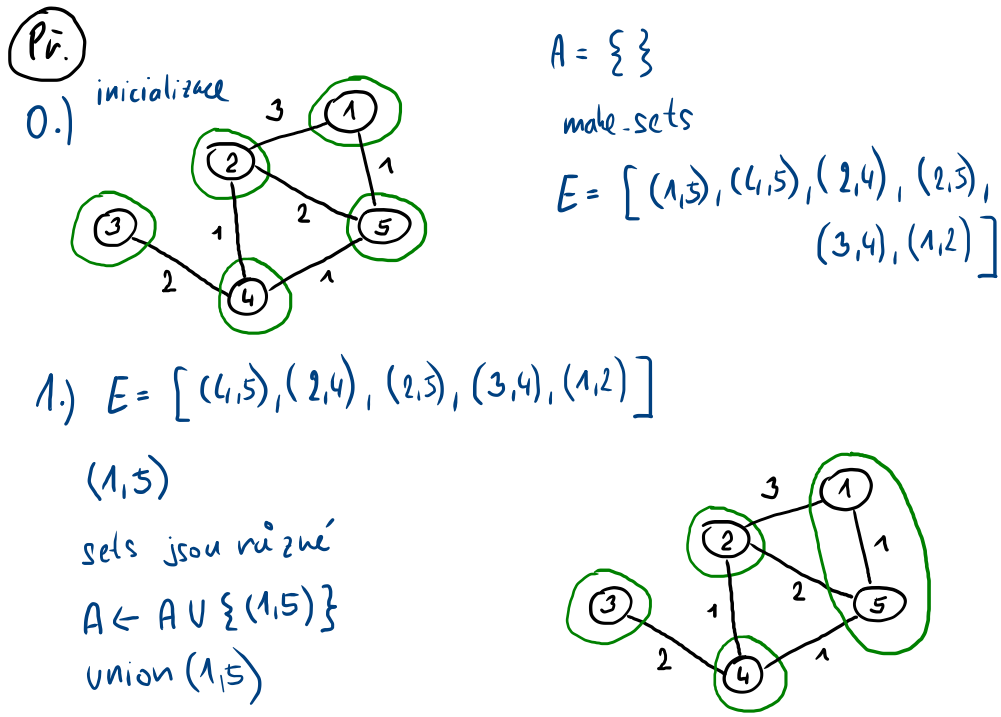
```
def kruskal_mst(G):
    # G je graf

    # inicializace, kazdy uzel je ve sve mnozine
    A = {}# A je mnozina hran rozpracovane minimalni kostry
    for v in G.V:
        make_set(v)

    # seradit vzestupne podle w
    E = sort(G.E, G.w)

    for (u, v) in E:
        if find_set(u) != find_set(v):
            A += {(u, v)}
            union(u, v)

    return A
```

Výpis 45.2: Kruskalův algoritmus. Funkce `make_set(v)` vytvoří množinu obsahující $v$, `find_set(v)` vrátí reprezentanta množiny ve které se nachází $v$, `union(u, v)` sjednotí dvě množiny obsahující $u$ a $v$.

### 45.4.1 Složitost

- Řádek 5 – $O(1)$
- Řádek 6-7 – $n$-krát složitost *make_set* ($n$ je počet uzlů).
- Řádek 10 – $O(m \cdot \log(m))$ ($m$ je počet hran).
- Řádky 12-15 – Závisí na implementaci *find_set* a *union*.
  - Při implementaci seznamem s heuristickou celkem: $O(m + n \cdot \log(n))$.
  - Při stromové implementaci s váhami a zkratkami celkem: $O((m + n) \cdot \alpha(n))$. Kde $\alpha$ je velmi pomalu rostoucí funkce ($\alpha \leq 4$).
- Pro souvislý graf platí $m > n$. Proto množinové operace stojí $O(m \cdot \alpha(n))$. Jelikož $\alpha(n) = O(\log(n)) = O(\log(m))$, tak celková složitost je $O(m \cdot \log(m))$.
- Dále platí $m < n^2$, pak $\log(m) = O(\log(n))$, proto celkem: $O(m \cdot \log(n))$.

### 45.4.2 Příklad




Obrázek 45.5: Příklad, část 1.

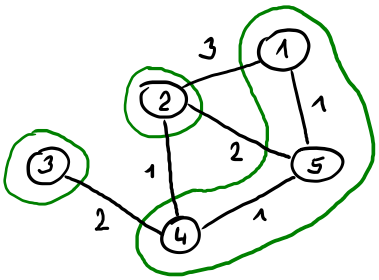
2.) $E = [(2,4), (2,5), (3,4), (1,2)]$

$(4,5)$

sets jsou různé

$A \leftarrow A \cup \{(4,5)\}$

union $(4,5)$



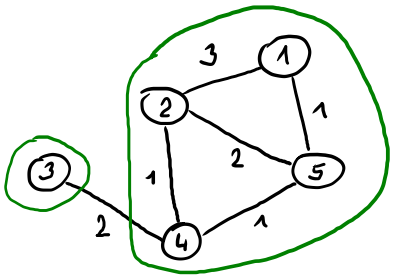

3.) $E = [(2,5), (3,4), (1,2)]$

$(2,4)$

sets jsou různé

$A \leftarrow A \cup \{(2,4)\}$

union $(2,4)$

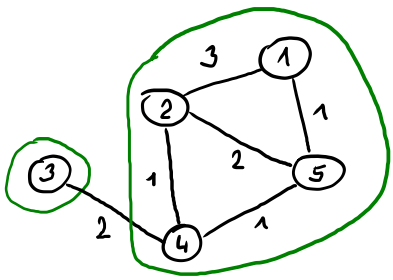



Obrázek 45.6: Příklad, část 2.

4.) $E = [(3,4), (1,2)]$

$(2,5)$

sets nejsou různé



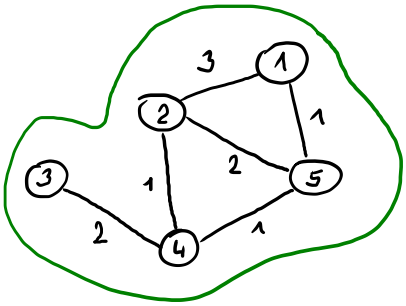

4.) $E = [(1,2)]$

$(3,4)$

sets jsou různé

$A \leftarrow A \cup \{(3,4)\}$

union $(3,4)$




5.) $E = [\ ]$, $(1,2)$, sets nejsou různé

Obrázek 45.7: Příklad, část 3.

## 45.5 Primův-Jarníkův algoritmus

Primův algoritmus buduje tzv. *A* strom. Má zadaný určitý uzel, ze kterého hledá nejbližší další uzel, který by připojil. A pak další a další.

```
1  def prim_mst(G, r):
2      # G je graf
3      # r je vychozi uzel
4
5      for u in G.V:
6          key[u] = INF # pole cen prechodu, kolik stoji prechod do vrcholu na indexu
7          pi[u] = NULL # pole predchudcu, kdo je predchudce vrcholu na indexu
8
9      key[r] = 0
10     Q = Queue(G.V) # prioritni fronta uzlu
11
12     while not Q.empty():
13         u = Q.extract_min(key) # vrati prvek z Q s nejmensi hodnotou v key
14
15         # pro vsechny sousedy uzlu u (Adj je seznam sousedu)
16         for v in Adj[u]:
17             # pokud je levnejsi cesta a jeste to neni prozkoumany uzel
18             if v in Q and w(u, v) < key(v):
19                 pi[v] = u key[v] = w(u, v)
20                 Q.decrease_key(key) # aktualizace prioritni fronty
21
22     return pi
```

Výpis 45.3: Primův algoritmus.

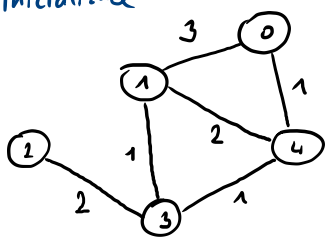
### 45.5.1 Složitost

- Řádky 5-10 – $O(n)$ za použití binární haldy ($n$ je počet uzlů).
- Řádky 12-13 – While cyklus se provede n-krát a protože *extract_min* stojí $O(\log(n))$, tak je celková složitost $O(n \cdot \log(n))$.
- Řádek 16 – For cyklus se provede $O(m)$ krát, protože dálka všech seznamů sousedů je dohromady $2m$ ($m$ je počet hran).
- Řádek 18-20 – $O(1)$.
- Řádek 21 – $O(\log(n))$.
- Jelikož $m > n$, tak celkem $O(n \cdot \log(n) + m \cdot \log(n)) = O(m \cdot \log(n))$.

### 45.5.2 Příklad

Př.

0.) inicializace




$r = 1$

$key = [\infty, 0, \infty, \infty, \infty]$

$\pi = [NULL, NULL, NULL, NULL, NULL]$

$Q = [0, 1, 2, 3, 4]$

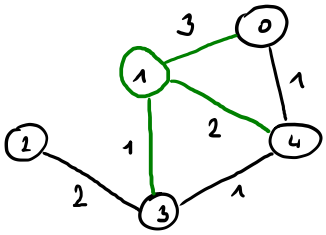
1.) $u = 1$

$Q = [0, 2, 3, 4]$

$v \in \{0, 3, 4\}$

$key = [3, 0, \infty, 1, 2]$

$\pi = [1, NULL, NULL, 1, 1]$




Obrázek 45.8: Příklad, část 1.

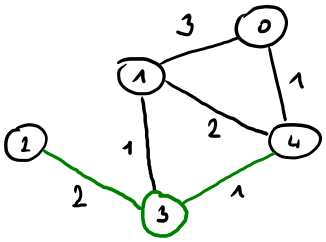
2.) $u = 3$

$Q = [0, 2, 4]$

$v \in \{\cancel{1}, 2, 4\}$

$key = [3, 0, 2, 1, 1]$

$\pi = [1, NULL, 3, 1, 3]$



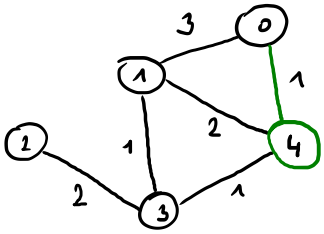

3.) $u = 4$

$Q = [0, 2]$

$v \in \{0, \cancel{1}, \cancel{3}\}$

$key = [1, 0, 2, 1, 1]$

$\pi = [4, NULL, 3, 1, 3]$




Obrázek 45.9: Příklad, část 2.

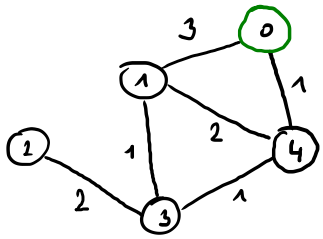
4.) $u = 0$

$Q = [2]$

$v \in \{\cancel{1}, \cancel{4}\}$

$key = [1, 0, 2, 1, 1]$

$\pi = [4, NULL, 3, 1, 3]$



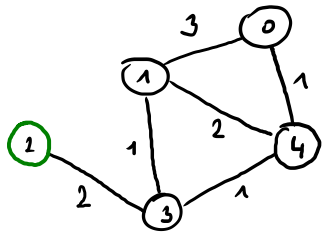

5.) $u = 2$

$Q = [\ ]$

$v \in \{\cancel{3}\}$

$key = [1, 0, 2, 1, 1]$

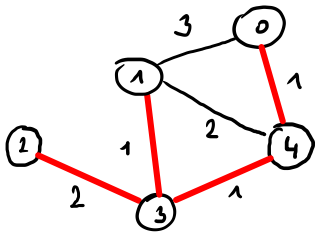
$\pi = [4, NULL, 3, 1, 3]$




Obrázek 45.10: Příklad, část 3.

6.) $key = [1, 0, 2, 1, 1]$

$\pi = [4, NULL, 3, 1, 3]$




Obrázek 45.11: Příklad, část 4.

# Kapitola 46

# GAL – Hledání nejkratších cest ze zdrojového uzlu do všech ostatních uzlů grafu (Bellman-Fordův algoritmus, Dijkstrův algoritmus).

## 46.1 Zdroje

- `gal-handouts.pdf`
- `GAL_2020-11-05.mp4`

## 46.2 Úvod a kontext

**Cena cesty** Nechť $G = (V, E)$ je ohodnocený graf s váhovou funkcí $w : E \mapsto \mathbb{R}$. Cena cesty $p = \langle v_o, v_1, \ldots, v_k \rangle$ je suma

$$w(p) = \sum_{i=0}^{k} w(v_i, v_{i+1})$$

.

**Cena nejkratší cesty** Cena nejkratší cesty z $u$ do $v$ je

$$\delta(u, v) = \begin{cases} min(\{w(p) : u \overset{\text{p}}{\Rightarrow} v\}) \\ \infty \text{pokud cesta neexistuje} \end{cases}$$

.

**Nejkratší cesta** Nejkratší cesta z $u$ do $v$ je pak libovolná cesta $p$ taková, že $w(p) = \delta(u, v)$.

**Cena cesty se záporným cyklem** Pokud na cestě z $u$ do $v$ existuje záporný cyklus (cyklus jehož celková cena je záporná), pak $\delta(u, v) = -\infty$.

**Záporné ohodnocení hran** Pokud na cestě z $u$ do $v$ neexistuje záporný cyklus, tak algoritmy pracují dobře i se záporným ohodnocením hran.

**Reprezentace cesty** Cestu reprezentujeme pomocí pole předchůdců $\pi$.

**Hledání nejkratších cest ze všech uzlů do jednoho** Tento problém lze řešit stejnými algoritmy. Graf se transponuje (převrácení orientace hran), provede se algoritmus pro problém „hledání nejkratších cest ze jednoho uzlu do všech ostatních uzlů“ a poté se transponuje zpět.

**Reprezentace nejkratší cesty** Nejkratší cestu grafu $G = (V, E)$ reprezentujeme pomocí pole předchůdců $\pi$, kde $\pi[v]$ označuje předchůdce uzlu $v \in V$ na nejkratší cestě. Podgraf předchůdců pak je $G_\pi = (V_\pi, E_\pi)$, $V_\pi = \{v \in V : \pi[v] \neq \text{NULL}\} \cup \{s\}$, $E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}$. V okamžiku dokončení algoritmu výpočtu nejkratších cest je $G_\pi$ strom nejkratších cest. Tj. kořenový strom obsahující nejkratší cesty ze zdroje $s$ do všech ostatních uzlů.

## 46.3 Pomocné funkce

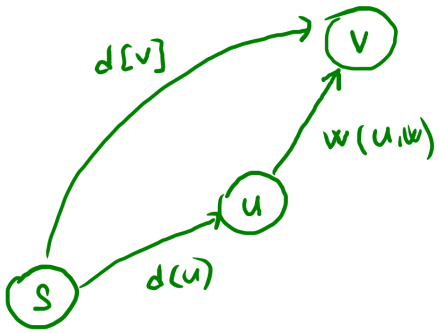
Představené algoritmy pracují z důvodu efektivity se sledy a nikoliv s cestami (bylo by nutné stále kontrolovat, zda nebyla porušena podmínka cesty), ačkoliv je problém nazývá hledání nejkratší cesty.

```
def initialize_single_source(G, s):
    # G je graf
    # s je vychozi uzel
    for v in G.V:
        d[v] = INF # d je pole vzdalenosti
        pi[v] = NULL # pi je pole predchudcu
    d[s] = 0
```

Výpis 46.1: Pomocná inicializační funkce. Složitost je $\Theta(n)$, kde $n$ je počet uzlů.

```
def relax(u, v, w):
    # u a v jsou uzly grafu
    # w je vahova funkce
    if d[v] > d[u] + w(u, v):
        d[v] = d[u] + w(u, v)
        pi[v] = u
```

Výpis 46.2: Pomocná funkce *relax*. Složitost je $O(1)$.




Obrázek 46.1: Ukázka činnosti funkce *relax*.

## 46.4 Bellman-Fordův algoritmus

Slouží pro řešení v obecných grafech, mohou obsahovat cykly a záporné hrany. Záporné cykly je však nutné detekovat a vrátit specifickou hodnotu. V podstatě se jedná o *brute force* algoritmus, provede se relaxace $n-1$-krát pro každou hranu.

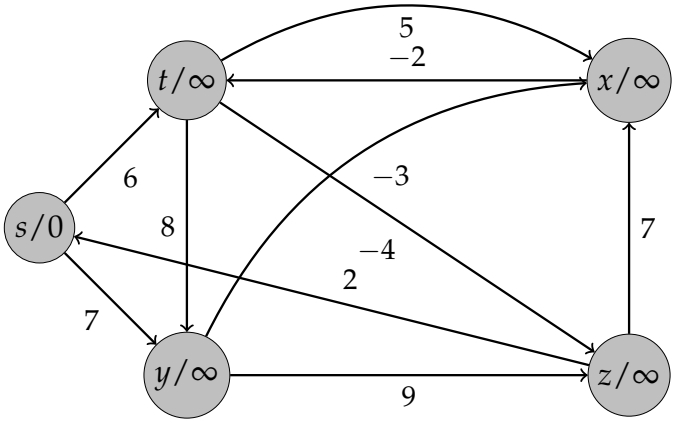
```
1  def bellman_ford(G, s, w):
2      # G je graf
3      # s je vychozi uzel
4      # w je vahova funkce
5
6      # faze inicializace
7      initialize_single_source(G, s)
8      n = len(G.V) # pocet uzlu
9
10     # faze relaxace: provedeni (n-1) * m relaxaci (m je pocet hran)
11     for _ in range(0, n-1):
12         for u, v in G.E:
13             relax(u, v, w)
14
15     # faze detekce zaporneho cyklu
16     for u, v in G.E:
17         if d[u] > d[v] + w(u, v):
18             return NULL
19
20     return pi
```

Výpis 46.3: Algoritmus Bellman-Ford. Proč $n-1$ iterací? Protože mezi libovolnými dvěma uzly v grafu, existuje cesta o maximálním počtu hran $n-1$.

### 46.4.1 Složitost

- Řádek 7, 8 – $\Theta(1)$.
- Řádky 11, 12, 13 – $(n-1) \cdot \Theta(m) = \Theta(n \cdot m)$, kde $n$ je počet uzlů a $m$ je počet hran grafu.
- Řádek 16, 17, 18 – $\Theta(m)$.
- Celkem $\Theta(n \cdot m)$.

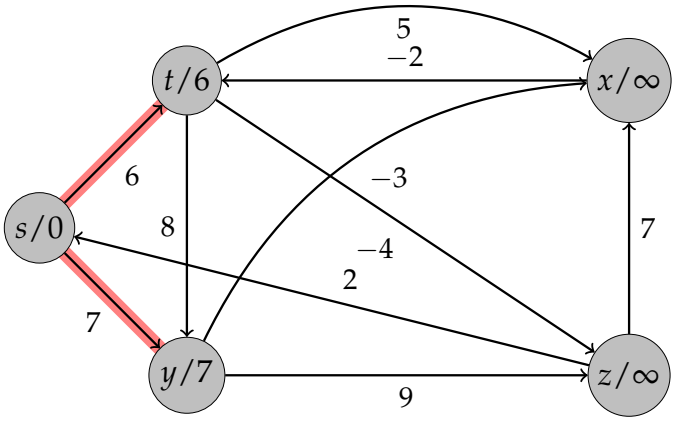
### 46.4.2 Příklad




Obrázek: Práce algoritmu Bellman-Ford.

- Pokud $(u,v) \in E$ je označená, pak $\pi[v] = u$
- Hrany se relaxují v tomto pořadí:
  $(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$.

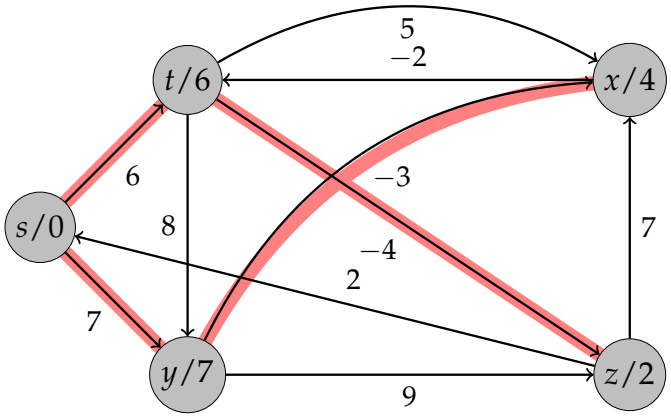
Obrázek 46.2: Příklad, část 1.




Obrázek: Práce algoritmu Bellman-Ford.

- Pokud $(u,v) \in E$ je označená, pak $\pi[v] = u$
- Hrany se relaxují v tomto pořadí:
  $(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$.

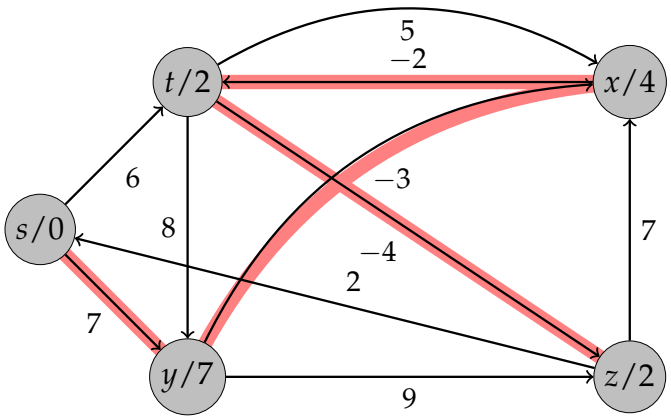
Obrázek 46.3: Příklad, část 2.




Obrázek: Práce algoritmu Bellman-Ford.

- Pokud $(u,v) \in E$ je označená, pak $\pi[v] = u$
- Hrany se relaxují v tomto pořadí:
  $(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$.

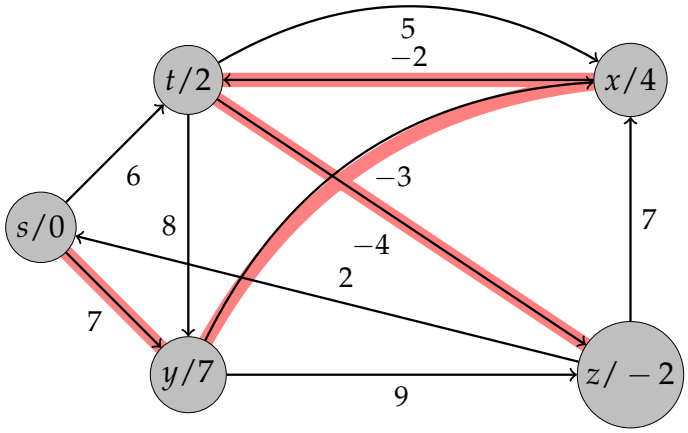
Obrázek 46.4: Příklad, část 3.




Obrázek: Práce algoritmu Bellman-Ford.

- Pokud $(u,v) \in E$ je označená, pak $\pi[v] = u$
- Hrany se relaxují v tomto pořadí:
  $(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$.

Obrázek 46.5: Příklad, část 4.




Obrázek: Práce algoritmu Bellman-Ford.

- Pokud $(u,v) \in E$ je označená, pak $\pi[v] = u$
- Hrany se relaxují v tomto pořadí:
  $(t,x),(t,y),(t,z),(x,t),(y,x),(y,z),(z,x),(z,s),(s,t),(s,y)$.

Obrázek 46.6: Příklad, část 5.

## 46.5 Dijkstrův algoritmus

Slouží pro řešení v acyklických grafech bez záporných hran. Pro takto omezený problém existují rychlejší algoritmy než pro problém v obecných grafech.

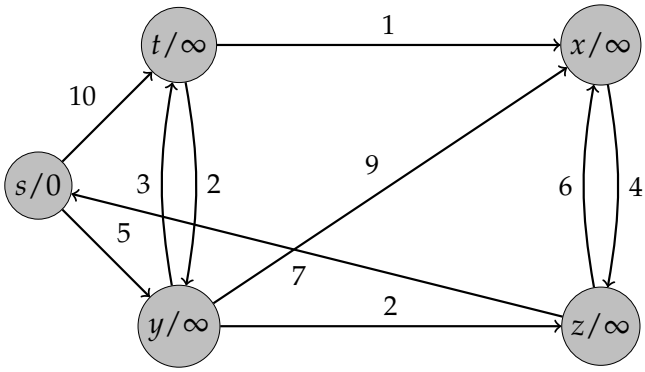
```
def dijkstra(G, s, w):
    # G je graf
    # s je vychozi uzel
    # w je vahova funkce

    # faze inicializace
    initialize_single_source(G, s)
    Q = Queue(G.V) # prioritni fronta uzlu
    S = {}# mnozina uzlu, ktera uz byla prozkoumana

    # faze relaxace
    while not Q.empty():
        u = Q.extract_min(d) # vrati prvek z Q s nejmensi hodnotou v d
        S += {u}
        # pro vsechny sousedy uzlu u (Adj je seznam sousedu)
        for v in Adj[u]:
            relax(u, v, w)

        Q.decrease_key(d) # aktualizace prioritni fronty

    return d, pi
```

Výpis 46.4: Algoritmus Dijkstra.

### 46.5.1 Složitost

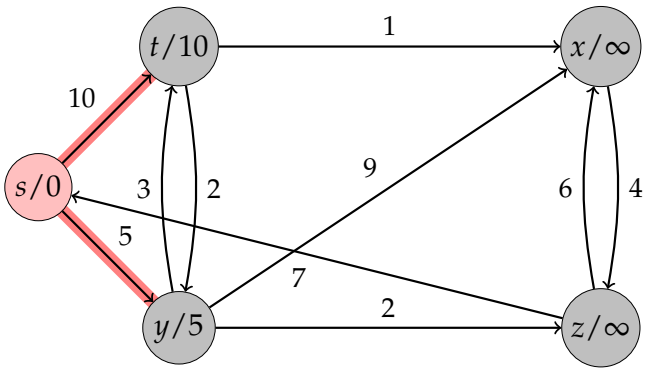
- Předpokládejme implementaci prioritní fronty pomocí pole.
- Řádek 8, 18 – $O(1)$.
- Řádek 11 – While cyklus se provede $n$-krát, kde $n$ je počet uzlů.
- Řádek 12 – $O(n)$, najítí minima v poli uzlů. Celkově (s cyklem) $O(n^2)$.
- Řádek 16 – $O(m)$, pro všechny hrany. Celkově (s cyklem) $O(m \cdot n)$.
- Celkem $O(n^2 + m) = O(n^2)$.
- Pro řídké grafy lze využít implementaci fronty pomocí binární haldy a získat tak $O(m \cdot \log(n))$.
- Při implementaci fronty pomocí Fibonacciho haldy dostaneme časovou složitost $O(n \cdot \log(n) + m)$.

### 46.5.2 Příklad



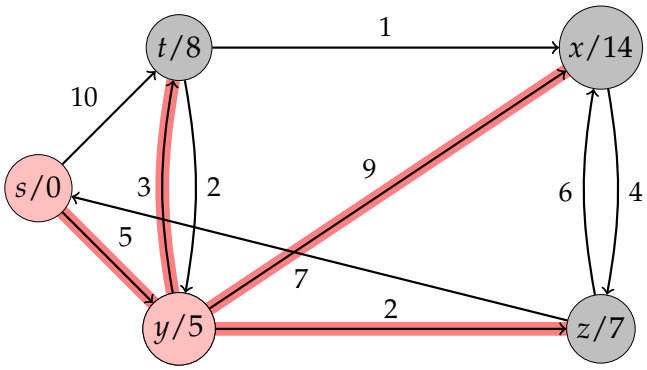

Obrázek: Práce Dijkstrova algoritmu. Označené uzly značí uzly z množiny $S$.

Obrázek 46.7: Příklad, část 1.



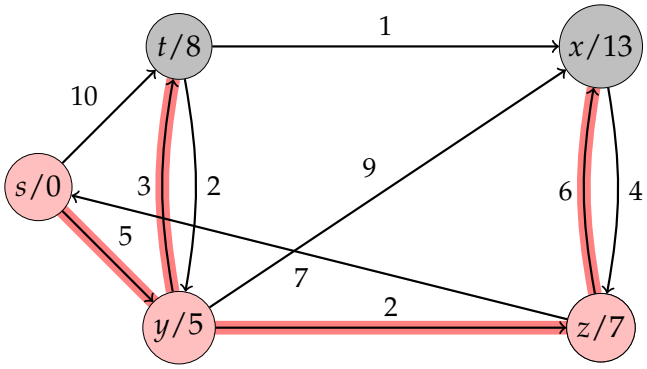

Obrázek: Práce Dijkstrova algoritmu. Označené uzly značí uzly z množiny $S$.

Obrázek 46.8: Příklad, část 2.



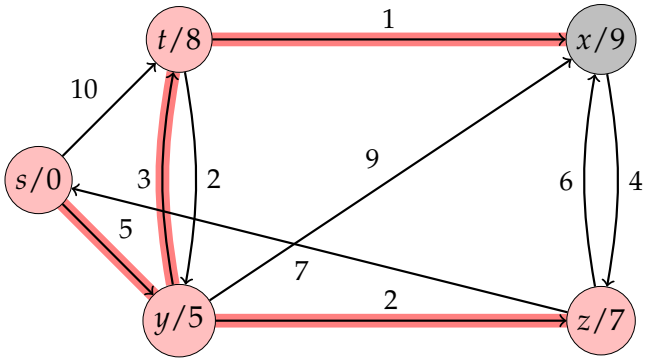

Obrázek: Práce Dijkstrova algoritmu. Označené uzly značí uzly z množiny $S$.

Obrázek 46.9: Příklad, část 3.



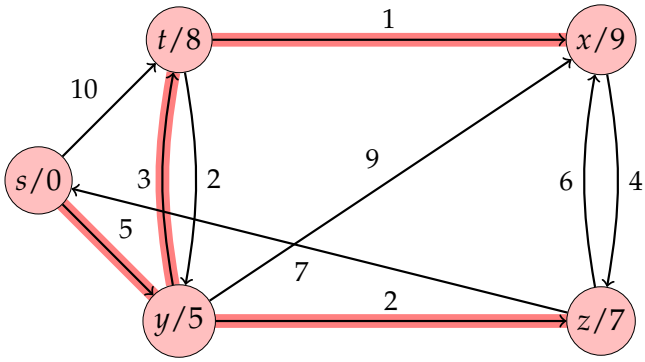

Obrázek: Práce Dijkstrova algoritmu. Označené uzly značí uzly z množiny $S$.

Obrázek 46.10: Příklad, část 4.




Obrázek: Práce Dijkstrova algoritmu. Označené uzly značí uzly z množiny $S$.

Obrázek 46.11: Příklad, část 5.




Obrázek: Práce Dijkstrova algoritmu. Označené uzly značí uzly z množiny $S$.

Obrázek 46.12: Příklad, část 6.

# Kapitola 47

# PDI – Klasifikace algoritmů volby koordinátora, algoritmus Bully a jeho složitost.

## 47.1 Zdroje

- `07_Synchronization.pdf`
- `PDI_2020-11-02.mp4`

## 47.2 Úvod a kontext

- Mějme množinu procesů v rámci distribuovaného systému. Řešíme problém nalezení shody na nějaké věci (synchronizační problém). Problém můžeme rozdělit na dvě situace:
  - **Problém volby koordinátora** – Výběr jednoho z procesů, který bude vedoucím procesem (koordinátor). Tento proces pak může vykonat určitou činnost nebo může sloužit ostatním procesům k realizaci význačné role v systému.
  - **Problém vzájemného vyloučení** – Předpokládejme, že konkrétní zdroj může v daném okamžiku používat pouze jeden proces. Tento problém se běžně vyskytuje ve víceprocesorových systémech, ale také v distribuovaných systémech.
- Synchronizační problémy lze v rámci operačních systémů nebo multiprocesorových systémů řešit pomocí provádění atomických operací, sdílené paměti apod. – je pro ně podpora v rámci operačního systému nebo hardwaru. V distribuovaných systémech takovéto prostředky nejsou často k dispozici, proto se synchronizační problémy řeší pomocí zasílání zprav, resp. algoritmicky.

## 47.3 Problém volby koordinátora

- Předpokládáme:
  - Každý proces má unikátní ID.
  - Procesy neznají stav (běžící, neběžící) dalších procesů.
  - Každý proces zná ID dalších procesů (záleží na topologii).

- Cíl:
  - Dosáhnutí shody mezi všemi procesy na procesu, který je koordinátor.
  - Kritérium výběru koordinátora může být různé. Např. na základě proces ID (proces s největším ID se stane koordinátorem).

## 47.4 Bully algoritmus

- Pro topologii každý s každým – každý proces může komunikovat s každým dalším procesem.
- Používá tři druhy zpráv: ELECTION, OK, COORDINATOR.

### 47.4.1 Postup

- Proces P, který má podezření, že chybí koordinátor, může zahájit volby.
  1. Proces P odešle zprávu ELECTION všem procesům s větším ID.
  2. Pokud nikdo neodpoví, P vyhrává volby a stává se koordinátorem.
  3. Pokud některý z procesů s větším ID odpoví (zpráva OK), tak přebírá řízení a práce P je ukončena.
  4. Pokud P obdrží zprávu ELECTION od procesů s menším ID, pošle jim odpověď OK na zablokování procesů.
- Nakonec zůstane pouze P (nový koordinátor), který o tom informuje ostatní zasláním zprávy COORDINATOR.
- Pokud se proces probudí nebo je restartován, první akcí je vyvolání voleb.

### 47.4.2 Složitost

- Složitost z hlediska počtu zpráv.
- Nejhorší případ (iniciátor s nejmenším ID):
  - $(n-1)$ iterací
  - $2(n-1)$ zpráv ELECTION a OK pro každou iteraci
  - $(n-1)$ zpráv COORDINATOR
  - Celkem: $(n-1) \times 2(n-1) + (n-1) \approx n^2$
- Nejlepší případ (iniciátor s největším ID):
  - $(n-1)$ zpráv COORDINATOR
  - Celkem: $(n-1)$

### 47.4.3 Příklad

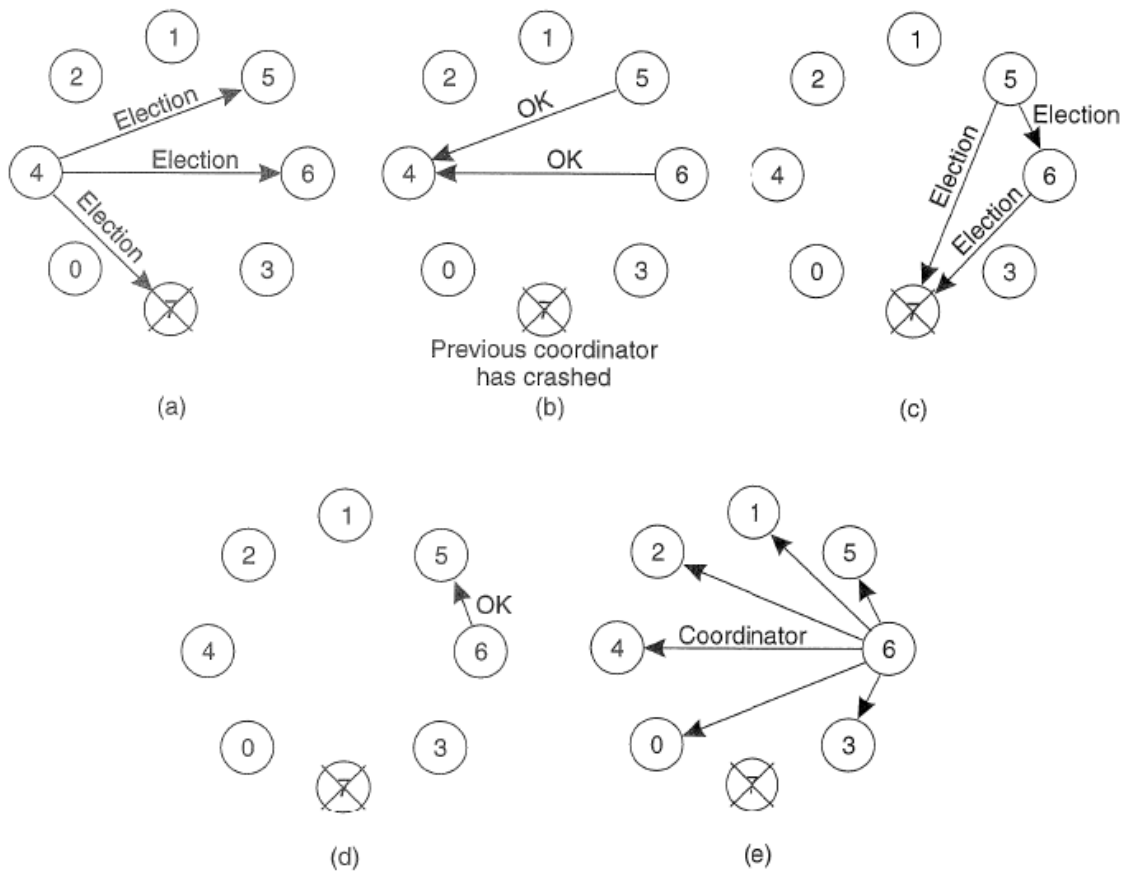



**Figure 5-11.** The bully election algorithm. (a) Process 4 holds an election. (b) Processes 5 and 6 respond, telling 4 to stop. (c) Now 5 and 6 each hold an election. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone.

Obrázek 47.1: Příklad činnosti Bully algoritmu.

## 47.5 Ring Algoritmus

- Pro kruhovou topologii – procesy jsou uspořádané do kruhu podle svého proces ID.
- Každý proces musí vědět nejenom o svém následovníkovi, ale také o jeho následníkovi, který funguje jako „záloha“, v případě že by se přímý následník stal nedostupný.
- Používá dva druhy zpráv: ELECTION, COORDINATOR.

### 47.5.1 Postup

- Proces P, který má podezření, že chybí koordinátor, může zahájit volby.
  1. Zašle zprávu ELECTION obsahující jeho ID dalšímu procesu (pokud další proces nereaguje, proces P zašle stejnou zprávu dalšímu v kruhu).
  2. Každý člen topologie přijme zprávu ELECTION, přidá do ní své ID a přepošle zprávu dalšímu procesu.

- Když se zpráva vrátí k procesu P, je zpráva převedena na zprávu COORDINATOR a poslána následujícímu procesu v topologii, aby bylo možné nahlásit:
  1. Novým koordinátorem se stává proces s nejvyšším ID.
  2. Členové sítě jsou stále aktivní.
- Po síti může obíhat více zpráv zároveň.

### 47.5.2 Složitost

- Složitost z hlediska počtu zpráv.
- Vždy $2n \approx n$ zpráv. Jedno kolečko „oběhne“ zpráva ELECTION a druhé zpráva COORDINATOR.

### 47.5.3 Příklad

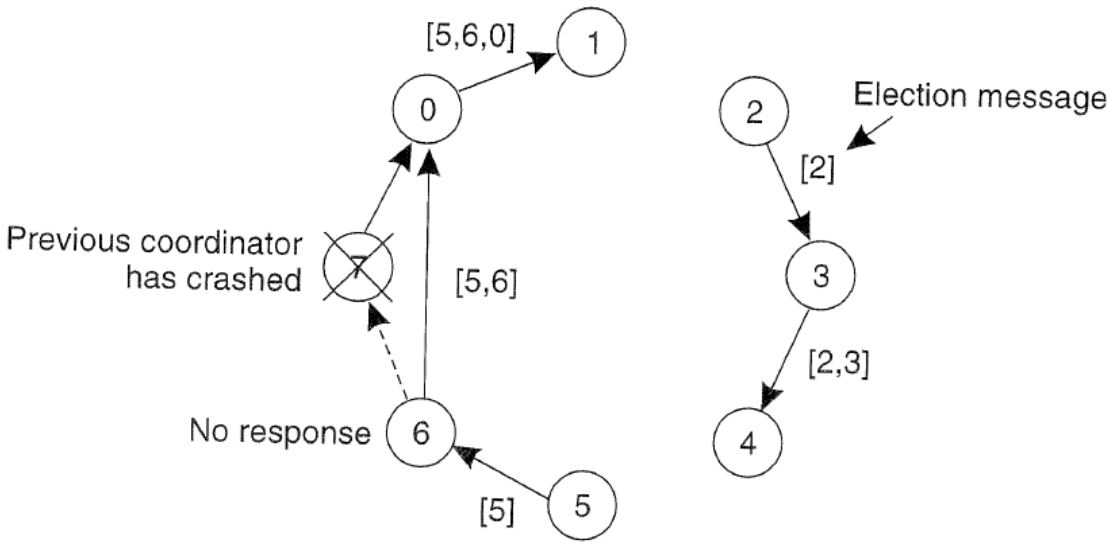



**Figure 5-12.** Election algorithm using a ring.

Obrázek 47.2: Příklad činnosti Ring algoritmu.

## 47.6 Algoritmus pro obecnou topologii

- Předpokládáme, že nemáme ani kruhovou topologii ani spojení každý s každým. Např.: peer-to-peer sítě, sensorové sítě, ...

### 47.6.1 Postup

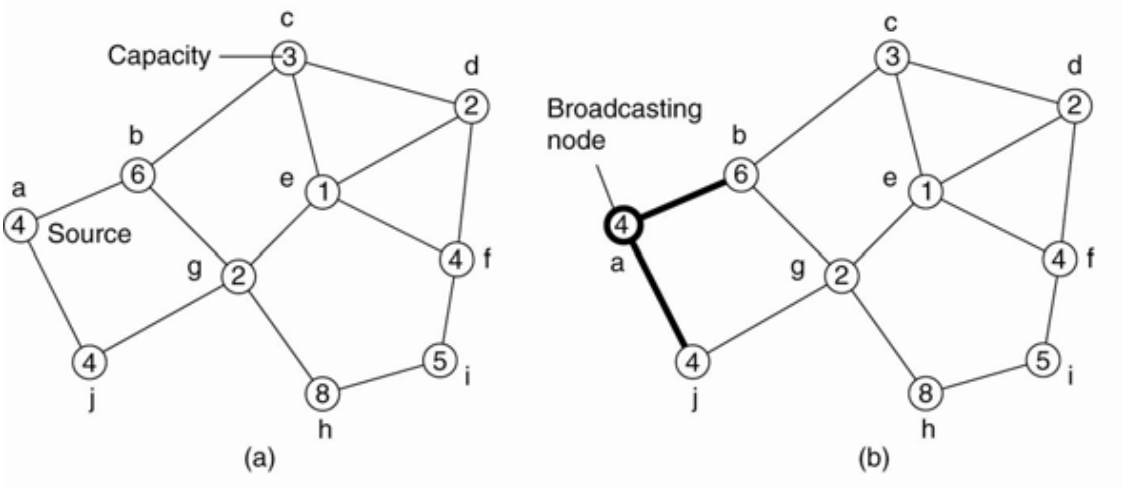
- V první iteraci se broadcastem posílá zpráva ELECTION.
- Každý uzel si uloží od kterého souseda dostal zprávu ELECTION jako první. Tím vzniká kostra grafu (*spanning tree*).

- Uložený soused je poté využijí pro zpětnou komunikaci. To znamená, že další komunikace už probíhá přes strom, nikoliv přes broadcast. Tím je ušetřena některé komunikace.

### 47.6.2 Složitost

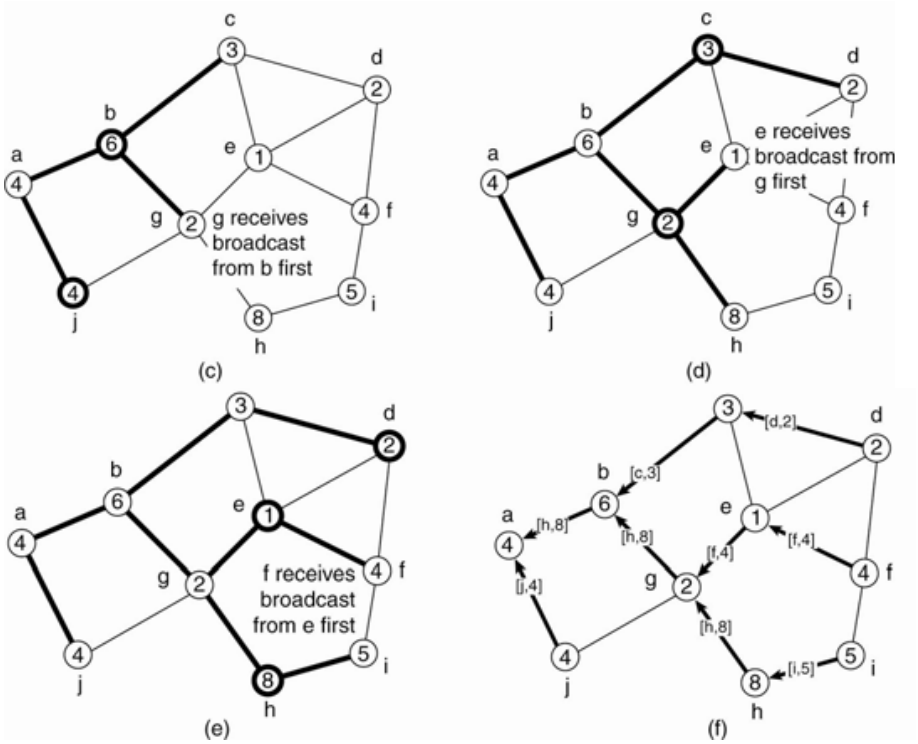
- Složitost z hlediska počtu zpráv.
- Inicializační broadcast: počet hran grafu.
- Odpověď: počet hran kostry grafu.
- Result broadcast: počet hran kostry grafu.

### 47.6.3 Příklad




Node *a* initiates an election.

Obrázek 47.3: Příklad činnosti algoritmu pro obecnou topologii, část 1.




In the end, source *a* notes that *h* is the best leader and broadcasts this info to all nodes.

Obrázek 47.4: Příklad činnosti algoritmu pro obecnou topologii, část 2.

# Kapitola 48

# PDI – Podmínky konsistentního globálního stavu distribuovaného systému.

### 48.0.1 Zdroje

- `04_Global_State_and_Snapshot.pdf`
- `PDI_2020-10-12.mp4`

## 48.1 Úvod a kontext

**Distribuovaný systém** Distribovaný systém je množina procesů $p_1, p_2, \ldots, p_n$, které jsou propojeny komunikačními kanály. V systému neexistuje žádná globální paměť ani globální hodiny. Procesy spolu komunikují pouze zasíláním zpráv skrze komunikačními kanály.

**Komunikační kanál** Komunikační kanál mezi procesy $p_i$ a $p_j$ značíme $C_{ij}$.

**Událost** Rozlišujeme tři typy událostí: interní událost procesu, zaslání zprávy a přijetí zprávy.

**Zpráva** Zpráva $m_{ij}$ značí zprávu zaslanou procesem $p_i$ procesu $p_j$. $send(m_{ij})$ značí odeslání zprávy a $recv(m_{ij})$ přijetí.

**Stav procesu** Lokální stav procesu $p_i$ značíme $LS_i$. Lokální stav je definován jako sekvence všech událostí, o kterých proces $p_i$ ví. Nechť $e$ je libovolná událost, $e \in LS_i$ značí, že událost $e$ patří do lokálního stavu procesu $p_i$, $e \notin LS_i$ značí, že událost $e$ nepatří do lokálního stavu procesu $p_i$.

**Stav komunikačního kanálu** Stav komunikačního kanálu $C_{ij}$ značíme $SC_{ij}$ a je definován množinou zpráv, které obsahuje. Pro kanál $C_{ij}$ můžeme definovat jeho stav na základě lokálních stavů procesů $LS_i$ a $LS_j$:

$$SC_{ij} = transit(LS_i, LS_j) = \{m_{ij} \mid send(m_{ij}) \in LS_i \wedge rec(m_{ij}) \notin LS_j\}$$

## 48.2 Model komunikace

- FIFO – Komukační kanál funguje jako fronta zpráv *first in*, *first out*. Kanál tedy zachovává pořadí zpráv sám o sobě.
- non-FIFO – Komunikační kanál se chová jako datová struktura množina, do které odesílatel vkládá zprávy a příjemce je odebírá v náhodném pořadí.
- Causal ordering (kauzální uspořádání) – Systém, který podporuje kauzální doručení zpráv splňuje následující vlastnost. Pro jakékoliv dvě zprávy $m_{ij}$ a $m_{kj}$ platí, pokud $send(m_{ij}) \rightarrow send(m_{kj})$, pak i $recv(m_{ij}) \rightarrow recv(m_{kj})$.

## 48.3 Konzistentní globální stav

**Globální stav** Globální stav distribuovaného systému je kolekce lokálních stavů procesů a komunikačních kanálů.

$$GS = \left\{ \bigcup_i LS_i \, , \bigcup_{i,j} SC_{ij} \right\}$$

**Časoprostorový diagram** Diagram pro vizualizaci komunikace procesů v distribuovaném systému. Viz obrázek 48.1 a 48.2.

**Konzistentní globální stav** Konzistentní globální stav (*snapshot*) je stav systému v určitém časovém okamžiku. Lze si jej představit jako řez v časoprostorovém diagramu, který rozděluje diagram na dvě části: minulost a budoucnost. Aby byl řez (globální stav) konzistentní, tak pokud je doručení nějaké zprávy v minulosti, musí být v minulosti i její odeslání. Formálně jde o globální stav, který splňuje nálsedující podmínky:

$$send(m_{ij}) \in LS_i \Rightarrow m_{ij} \in SC_{ij} \oplus recv(m_{ij}) \in LS_j$$

$$send(m_{ij}) \notin LS_i \Rightarrow m_{ij} \notin SC_{ij} \wedge recv(m_{ij}) \notin LS_j$$

**K čemu je *snapshot*** *Snapshot* lze využít např. pro tvorbu záloh systému nebo při zotavování systému po chybách.

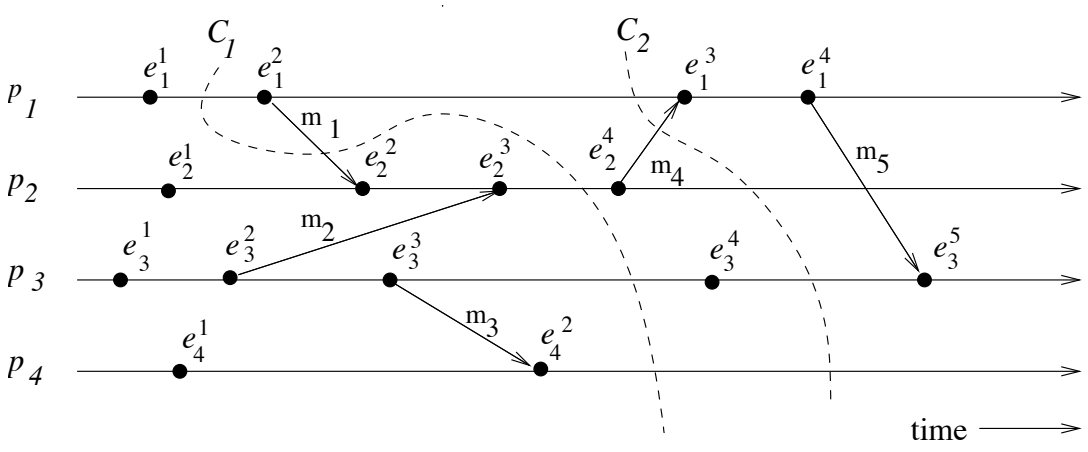
**Jak lze *snapshot* vytvořit** Absence globální sdílené paměti, globálních hodin a nepředvídatelná délka zpoždění v odesílání zpráv v distribuovaném systému činí problém vytváření snapshotů netriviálním. Způsob vytváření lze rozdělit do dvou kategorií: na základě algoritmů a na základě checkpointů.

**Problémy při zaznamenávání snapshotu** Jak rozlišit mezi zprávami, které mají být součástí snapshotu a které nikoliv?

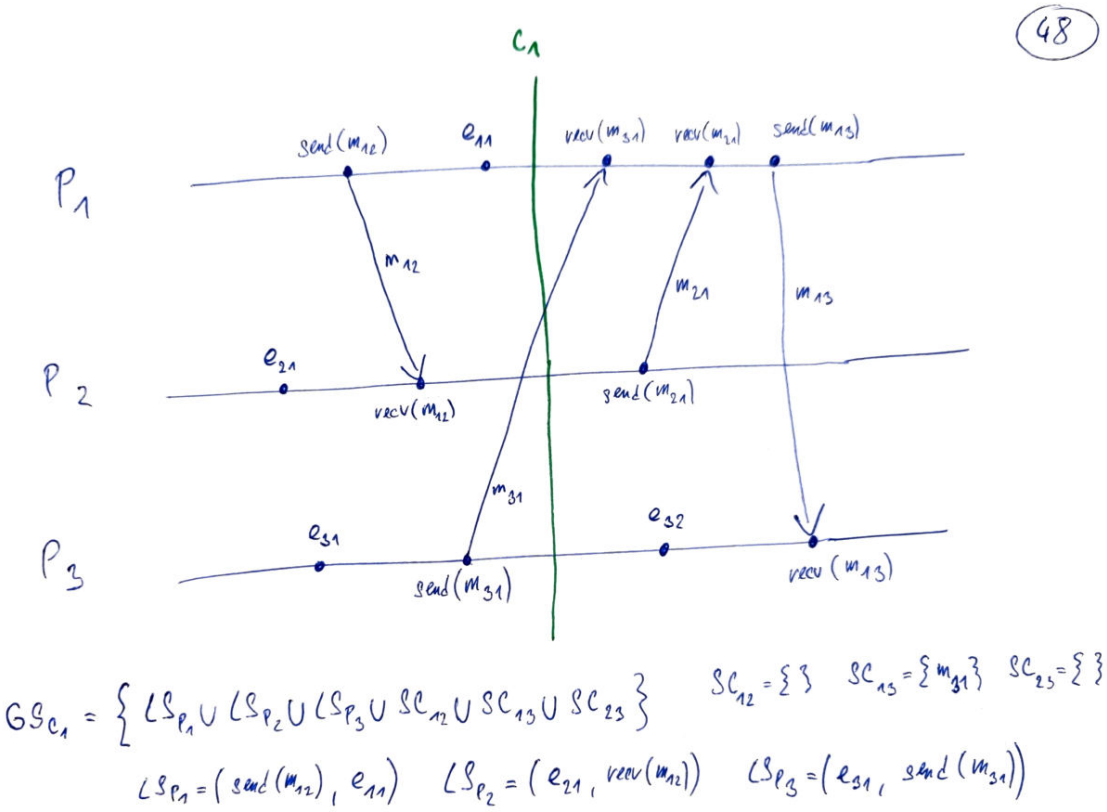
- Zprávy, které jsou odeslány procesem před zaznamenáním svého snapshotu, jsou zaznamenány do stavu.
- Zprávy, které jsou odeslány procesem po zaznamenání svého snapshotu, nejsou zaznamenány do stavu.

**Jak rozpoznat okamžik, ve kterém má proces zaznamenat *snapshot*?**

- Proces $p_j$ musí zaznamenat svůj *snapshot* před zpracováním zprávy $m_{ij}$, která byla poslána procesem $p_i$ po zaznamenání jeho snapshotu.




Obrázek 48.1: Příklad řezu v časoprostorovém diagramu. Řez $C_1$ je nekonzistentní, kvůli zprávě $m_1$. Řez $C_2$ je konzistentní a zpráva $m_4$ je zachycena ve stavu kanálu $SC_{21}$.




Obrázek 48.2: Příklad konzistentního globální stavu formálně.

## 48.4 Chandy-Lamport algoritmus

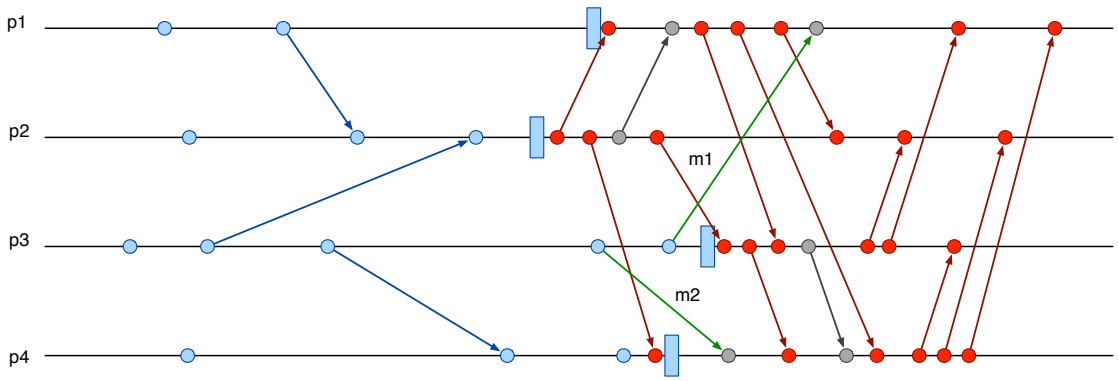
- Pro FIFO komunikaci.
- Používá tzv. marker (kontrolní zpráva) pro oddělení zpráv před zaznamenáním snapshotu a po.
  - Poté, co proces zaznamená svůj lokální stav, odešle marker do všech (odchozích) kanálů, předtím než bude posílat další zprávy.
  - Marker odděluje zprávy v kanálu na ty, které mají být zahrnuty do snapshotu, a na ty, které ne.
  - Proces musí zaznamenat svůj lokální stav nejpozději v okamžiku, kdy obdrží marker skrze některý svůj (příchozí) kanál.

**Marker Sending Rule** for process *i*

1. Process *i* records its state.
2. For each outgoing channel C on which a marker has not been sent, *i* sends a marker along C before *i* sends further messages along C.

**Marker Receiving Rule** for process *j*
On receiving a marker along channel C:
  **if** *j* has not recorded its state **then**
    Record the state of C as the empty set
    Follow the "Marker Sending Rule"
  **else**
    Record the state of C as the set of messages received along C after *j*'s state was recorded and before *j* received the marker along C

Obrázek 48.3: Pseudokód algoritmu Chandy Lamport.




Obrázek 48.4: Příklad zaznamenání globální stavu pomocí algoritmu Chandy Lamport. Modrý obdelník značí zaznamenání lokálního stavu. Červené zprávy souvisí s odesíláním markeru. Zeleně jsou označeny zprávy, které jsou zaznamenány ve stavu kanálu. Modré jsou zprávy v minulosti, zaznamenány ve stavu kanálu. Šedé jsou zprávy v budoucnosti, nejsou zaznamenány vůbec.

- Složitost:

– Z pohledu jedné instance vyžaduje algoritmus $\mathcal{O}(e)$ zpráv, kde $e$ je počet hran v síti, resp. počet komunikačních kanálů.

$$n \cdot (n-1) = \mathcal{O}(n^2)$$

# Kapitola 49

# PDI – Principy distribuovaného zpracování MapReduce, průběh a jednotlivé operace distribuovaného výpočtu pomocí MapReduce, jeho implementace v Apache Hadoop a Apache Spark.

## 49.1 Zdroje

- `09_Hadoop.pdf`
- `10_Spark.pdf`
- `PDI_2020-11-16.mp4`
- `PDI_2020-11-23.mp4`

## 49.2 Úvod a kontext

**OLTP** OLTP (*Online Transactional Processing*, provozní databáze, systémy pro online zpracování transakcí) jsou standardní databázové systémy s pevnou strukturou dat definovou pomocí databázového schématu. Jsou navrženy a optimalizovány pro chod provozních aplikací s primárním cílem zajistit rychlý a souběžný přístup k datům. To vyžaduje transakční zpracování, řízení souběžnosti a techniky obnovy (rollback), které zaručují konzistenci dat. Díky těmto vlastnostem mají OLTP databáze špatný výkon při provádění složitých dotazů, které potřebují spojit mnoho relačních tabulek dohromady nebo agregovat velké objemy dat. Kromě toho obsahují typicky podrobná data a neobsahují historická data, která jsou při datové analýze potřeba.

**OLAP** OLAP (*Online Analytical Processing*, online analytické zpracování) je databázové paradigma specificky zaměřené na dotazy, zejména na analytické dotazy. Používají se zde jiné techniky indexování a optimalizace dotazů. Normalizace není pro toto paradigma žádoucí, protože rozděluje databázi na mnoho tabulek. Složité dotazy v takovém případě vyžadují rekonstrukci dat a s tím spojený vysoký počet spojování tabulek. Pracuje se s tzv.

multidimenzionálními kostkami, avšak v pozadí jsou stále relační databáze.

**NoSQL** Potřeba ukládat proudy dat (zpracovávané v reálném čase bez možnosti pozastavení), obrázky, multimédia, velké JSON soubory, ..., vedla ke vzniku NoSQL databází. NoSQL databáze používají jiné prostředky než tabulková schémata tradiční relační databáze. Často jde o „hloupé“, nestrukturované uložiště klíč-hodnota.

**BigData** Velká, nestrukturovaná (různorodá), rychle rostoucí data, která není možné uložit ani zpracovávat běžnými přístupy (na jednom uzlu, jedním uzlem). Produkují je např.: IoT senzory, sociální sítě, chatovací aplikace, webové vyhledávače, ... Pro jejich zpracování je nutné využít distribuované systémy (pro uložení i zpracování).

**Distribuované zpracování dat** Distribuované zpracování dat je zpracování velkých dat (*big data*) pomocí distribuovaných systémů. To s sebou přináší problémy. Jak zaručit vhodnou distribuci dat a výpočtu mezi uzly? Jak řešit nespolehlivost a výpadky uzlů? Jak a kam zajistit doručení výsledků výpočtu? ...

## 49.3 MapReduce

Algoritmy pro indexování webových stránek (Page Rank) přestávaly být udržitelné, bylo potřeba zvýšit jejich škálovatelnost. Google vydal příspěvek „MapReduce: Simplified Data Processing on Large Clusters“, kde bylo představeno paradigma MapReduce. Jde o paradigma distribuovaného výpočtu založené na funkcích *map* a *reduce* z funcionálního programování.

**Map** Funkce *map* má ve funkcionálním programování 2 vstupní parametry a vrací seznam hodnot. První parametr je unární operátor (nebo funkce fungující jako unární operátor) a druhý je seznam hodnot. Výstupní seznam je spočítán jako aplikace unárního operátoru na vstupní seznam. Příklad:

$$map(square, [1, 2, 3, 4]) = [1, 4, 9, 16]$$

. V paradigmata MapReduce *map* vrací data jako seznam dvojic klíč-hodnota, přesněji:

$$map((key, value)) \rightarrow [(key, value)]$$

.

**Reduce** Funkce *reduce* má ve funkcionálním programování 2 vstupní parametry a vrací jednu hodnotu. První parametr je binární operátor (nebo funkce fungující jako binární operátor) a druhý je seznam hodnot. Výstupní hodnota je spočítána jako postupná aplikace binárního operátoru na všechny hodnoty ve vstupním seznamu. Příklad:

$$reduce(+, [1, 4, 9, 16]) = 30$$

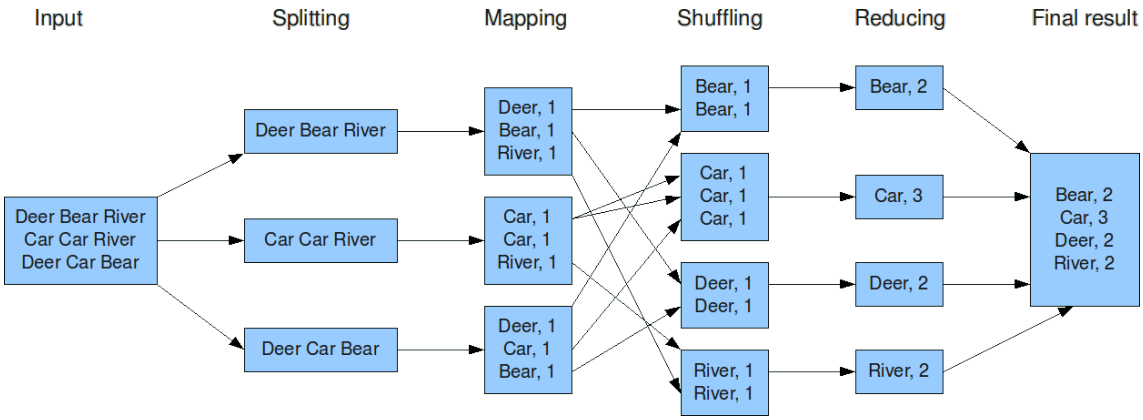
. V paradigmata MapReduce *reduce* bere na vstupu klíč a seznam hodnot a vrací opět seznam dvojic klíč-hodnota, přesněji:

$$reduce(key, [value]) \rightarrow [(key, value)]$$

.

```python
def map(input_key: str, input_value: str) -> list[tuple[str, int]]:
    # input_key - document name
    # input_value - document content (etc. line)
    result = []
    for word in input_value.split(' '):
        result.append((word, 1))
    return result

def reduce(input_key: str, input_value: list[int]) -> tuple[str, int]:
    result = 0
    for val in input_value:
        result += value
    return (input_key, result)
```

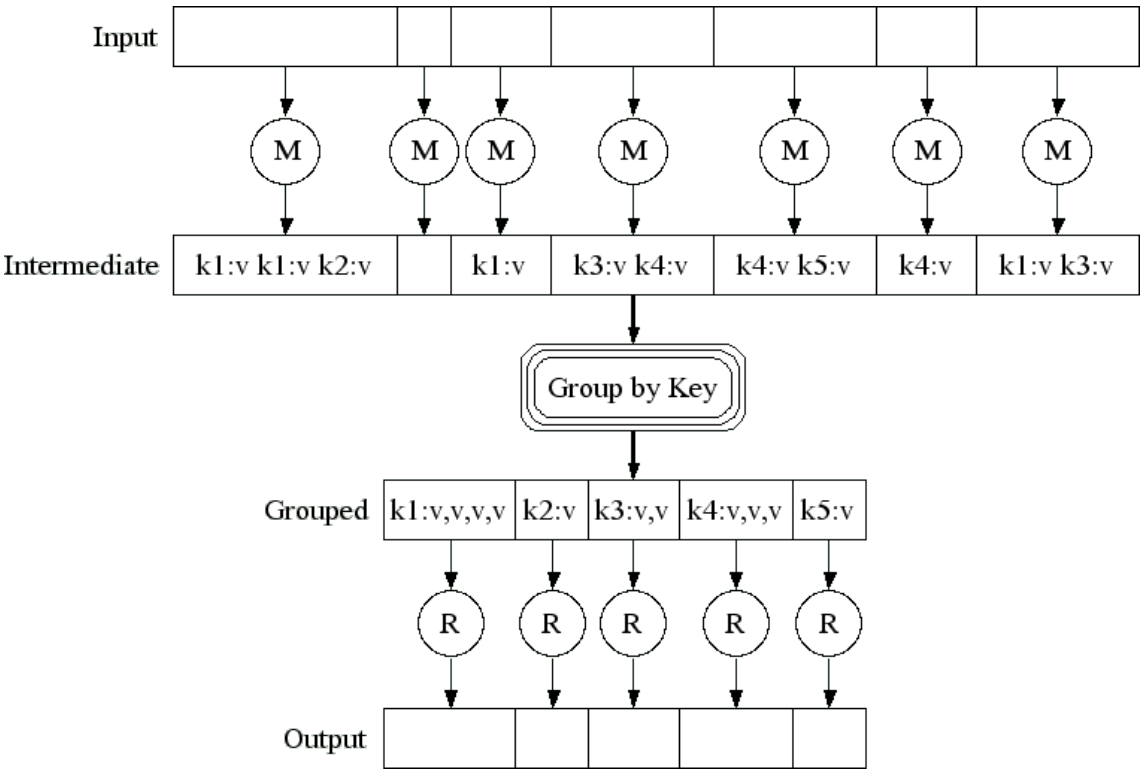
Výpis 49.1: Příklad implementace funkcí *map* a *reduce* v paradigmatu MapReduce pro počítání četnosti slov ve vstupu v Pythonu.




Obrázek 49.1: Úloha počítání četnosti slov v paradigmatu MapReduce v diagramu.

**Průběh MapReduce** Celý MapReduce probíhá v několika krocích, viz obrázek 49.1.

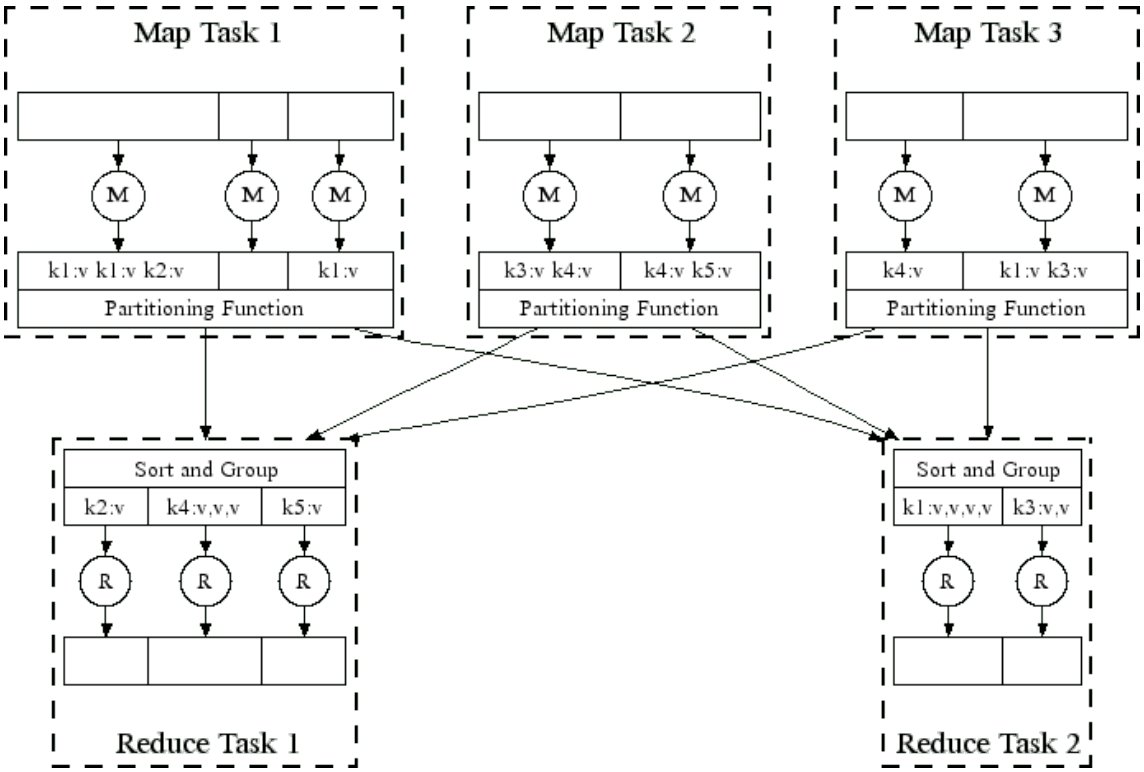
1. Input – Přípravený vstup pro distribuovaný výpočet (např. soubory ve virtuálním distribuovaném souborovém systému, viz dále HDFS).
2. Splitting – Rozdělení vstupu na části, které budou přiděleny jednotlivým uzlům. Může být výchozí (např. rozdělení textového souboru po řádcích) nebo definováno uživatelem.
3. Mapping – Každý uzel aplikuje funkci *map* na svoji přidělenou část. Uživatel definuje jak má funkce *map* vypadat.
4. Shuffling (také Grouping, Partitioning, Comparing)– Výpočetní uzly si mezi sebou vymění hodnoty, které spočítaly, na základě klíče. Tento krok zařižuje platforma pro distribuovaný výpočet sama o sobě, typicky na základě hashů klíčů. Tento krok je většinou *bottleneck*.
5. Reducing – Každý uzel zapojený do tohoto kroku (často je v tomto kroku potřeba méně uzlů, než v kroku mapping) aplikuje funkci *reduce* na svoji přidělenou část. Uživatel definuje jak má funkce *reduce* vypadat.
6. Final Result – Finální výsledek (např. zapsán do do virtuálního distribuovaného souborového systému, viz dále HDFS).




Obrázek 49.2: Výpočet MapReduce v obecném schématu.

**Combiner** Optimalizační krok, jde o „jakési“ provedení operace *reduce* už ve fázi *map* (každým uzlem). Tím je snížen počet mezivýsledků ve fázi Shuffling. Typicky funkce *combine* je stejná jako *reduce*.

**Virtuální distribuovaný souborový systém** Pro realizaci distribuovaného výpočtu je rovněž potřeba distribuovaný souborový systém (DFS). Ten je typicky realizován jako virtuální souborový systém nad jednotlivými souborovými systémy uzlů. Např.: GFS – Google File System, HDFS – Hadoop File System (viz dále). DFS obsahuje data samotná (*data nodes*) a metadata o tom, která data jsou na jakých uzlech (*name nodes*).




Obrázek 49.3: Výpočet MapReduce v obecném schématu a rozdělením práce na jednotlivé uzly (uzel je typicky víceprocesorový).

## 49.4 Apache Hadoop

Apache Hadoop je *open-source* implementace MapReduce paradigmatu vyvíjená Apache Software Foundation. Jde o implementaci v Jave, ta je vhodná, jelikož díky JVM (Java Virtual Machine) je spouštění uživatel definovaných funkcí *map* a *reduce* snadné.

**Hadoop MapReduce** – Implementace MapReduce paradigma. Data jsou čtena a ukládána na HDFS (včetně mezivýsledků). To znamená, můžeme pracovat v podstatě neomezenými daty, ale ukládání a načítání výpočet zpomalují.[1].

**HDFS** HDFS (*Hadoop Distribute File System*) je virtuální distribuovaný souborový systém. Standardní soubor je rozdělen na datové bloky které jsou distribuovány na různé datové uzly. Architektura HDFS se skládá ze dvou typů uzlů – Name Node a Data Node. Name Node obsahuje alokační tabulku pro souborový systém. Ví které datové bloky patří kterému souboru a kde jsou uloženy. Obsahuje další metadata jako názvy souborů, cesty, ... Data Node obsahuje datové bloky. Typicky redundance a replikace, počítá se s možným selháním uzlů. Pro **čtení dat** se klient zeptá Name Nodu na konkrétní soubor v HDFS. Name Node vrátí metadata o souboru, na jakých Data Nodech se vyskytuje. Klient požádá příslušné Data Nody, ty mu pošlou data, která se na klientovi „poskládají“ do výsledného souboru. Pro **zápis dat** se klient zeptá Name Nodu, kam by měl zapisovat. Klient zapíše na příslušný Data Node. Data Node poté vyřeší replikace s dalšími uzly.

[1] *Nebylo přednášeno podrobněji, pravděpodobně stačí princip obecného MapReduce, který byl vysvětlen v předchozí sekci.*

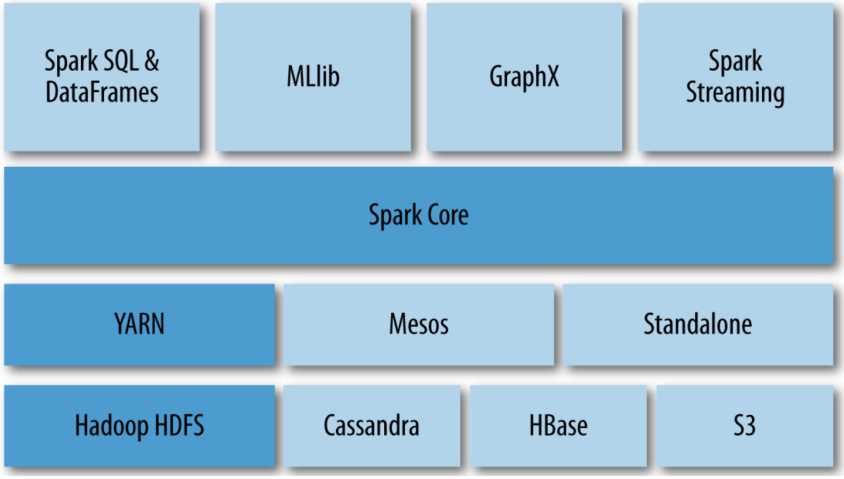
**Hadoop YARN** Hadoop YARN je plánovač (*scheduler*). Plánuje výpočet tak, aby proběhl co nejlepším způsobem na konkrétnbí distribuované architektuře. Plánovač má obecné obecné rozhraní a Hadoop YARN lze nahradit za jiný.

**Hadoop Common** Hadoop Common jsou další knihovny a ovladače pro klienty.

**Další nástroje** Nad Apache Hadoop existuje mnoho dalších nástrojů. Apache Pig pro *high level* programování map-reduce úloh. Apache Hive pro pro dolování dat nad Apache Hadoop. Apache HBase jako distribuovaná databáze nad Apache Hadoop, ...

## 49.5 Apache Spark

Apache Spark je *open-source* nástroj pro distribuované zpracování rozsáhlých dat vyvíjený Apache Software Foundation. Hlavní cíl je zvýšení rychlosti. Spark na to jde přesunutím co nejvíce výpočtů do operační paměti jednotlivých uzlů a tím pádem zminimalizovat počet zápisů a čtení z DFS (snaha odstranit *bottleneck* v kroku shuffling u Hadoopu). Tím ale vzniká jiný problém, a sice výpadek uzlu znamená, že data jsou ztraceny.




Obrázek 49.4: Architektura Apache Spark. Hlavní je Spark Core, zbytek funguje na systému pluginů a může používat HDFS, Hadoop YARN a Hadoop Common.

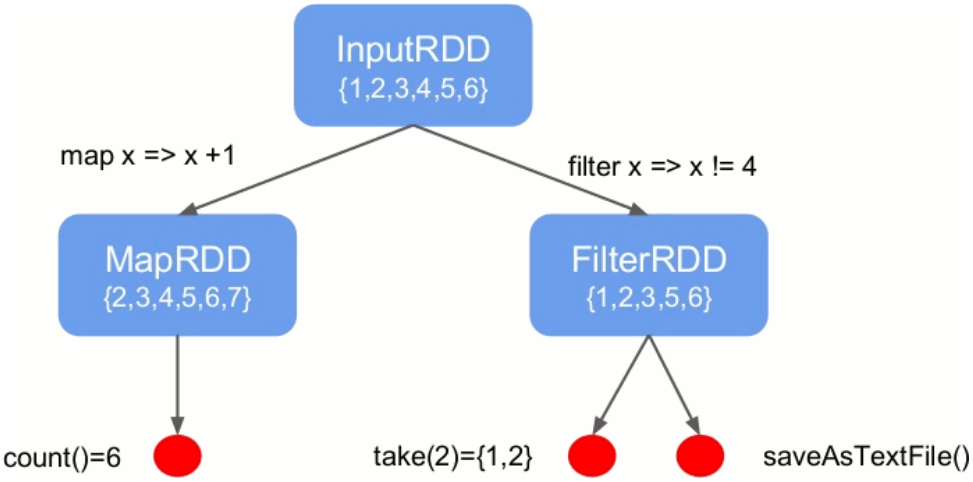
**Resilient Distributed Dataset** Resilient Distributed Dataset (RDD) je základní datová struktura Sparku. Jedná se o typované kolekce n-tic, které jsou neměnné (*read only*). Vstup je transformován na RDD a každá operace je pak transformace jednoho RDD na jiné.

$$RDD_1 \rightarrow map() \rightarrow RDD_2 \rightarrow reduce() \rightarrow RDD_3$$

**Strategie vyhodnocování** Spark uplatňuje strategii vyhodnocování *lazy evaluation*. Vyhodnocování výrazu je odkládáno až do doby, dokud není potřeba jeho hodnota. Zabraňuje opakovanému vyhodnocování. Je vyhodnocována pouze ta část, která je potřeba. RDD funguje jako abstraktní datová struktura, nemusí obsahovat data uvnitř, ale pouze předpis jak data získat a získá je, až když jsou potřeba.

**Struktura výpočtu** Struktura výpočtu odpovídá orientovanému acyklickému grafu (DAG, *Directed Acyclic Graph*). Uzly jsou RDD a hrany jsou transformace. DAG je znám i dalším uzlům, takže pokud nastane výpadek uzlu a výpočet je ztracen, může být uzel snadno zastoupen.

**Klíčové vlastnosti** Klíčové vlastnosti Sparku jsou *lazy evaluation*, *in-memmory* a *parallel computing*.




Obrázek 49.5: Příklad výpočtu v Apache Spark.

# Kapitola 50

# KRY – Symetrická kryptografie. Vlastnosti, vlastnosti bezpečného algoritmu, délka klíče, útok silou, příklady symetrických algoritmů, Feistelovy šifry, DES, režimy činnosti, proudové šifry.

## 50.1 Zdroje

- `KRY03_Sym_MNG.pdf`
- `KRY_2021-02-22.mp4`
- `KRY_2021-03-01.mp4`
- `KRY_2021-03-08.mp4`

## 50.2 Úvod a kontext

**Kryptografie** Kryptografie (šifrování) je věda o metodách utajování smyslu zpráv převodem do podoby, která je čitelná jen se speciální znalostí.

**Kryptoanalýza** Kryptoanalýza je věda zabývající se metodami získávání obsahu šifrovaných informací bez přístupu k tajným informacím, které jsou za normálních okolností potřeba, tzn. především k tajnému klíči.

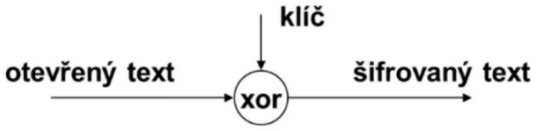
**Kryptologie** Jeden výraz pro kryptografii a kryptoanalýzu.

**Caesarova šifra** Princip Caesarovy šifry je založen na tom, že všechna písmena zprávy jsou během šifrování zaměněna za písmeno, které se abecedně nachází o pevně určený počet míst dále (tj. posun je pevně zvolen). Caesarova šifra spadá do kategorie substitučních šifer (stejný znak je při více vyskytech vždy zašifrován na stejný znak).

**Vigenerova šifra** Rozšíření Caesarovy šifry, klíč je delší než 1 znak. Klíč je řetězec, který reprezentuje posuny. V případě že vstup je delší než klíč, je klíč perioricky opakován. Vigenerova šifra spadá do kategorie polyalfabetických substitučních šifer (stejný znak může být při více výskytech zašifrován na jiný znak).

**Vernamova šifra (*One Time Pad*)** Vernamova šifra spadá do kategorie polyalfabetických substitučních šifer a je i dnes nerozluštitelná pokud:

- klíč je delší než vstupní text,
- klíč se nepoužije opakovaně,
- klíč je náhodný.




Obrázek 50.1: Vernamova šifra.

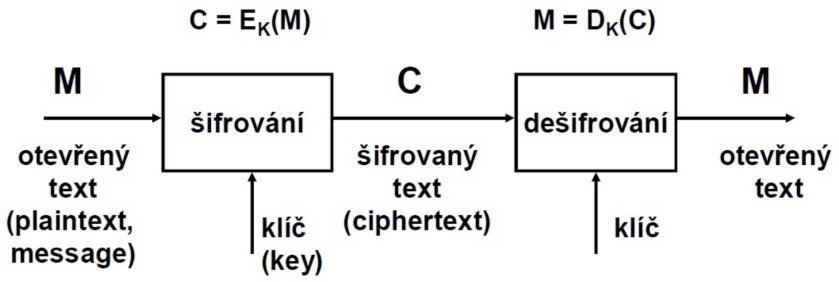
**Autoklíč (*autokey*)** Šifrování klíčem a když vstupní text je delší než klíč, tak se pokračuje šifrováním otevřeným nebo šifrovaným textem. Lze použít u Vigenerovy nebo Vernamovy šifry.

**Symetrická kryptografie** Algoritmy používají k šifrování i dešifrování stejný klíč. Výhodou symetrických šifer je jejich nízká výpočetní náročnost. Asymetrické šifry mohou být i stotisíckrát pomalejší. Nevýhodou je nutnost sdílení tajného klíče, takže jedna strana musí klíč vygenerovat a potom ho bezpečným způsobem předat druhé straně.

**Typy útoků**

- Ciphertext Only Attack (COA) – Útočník zná pouze zašifrovaný text a snaží se zjistit klíč nebo otevřený text. Nejčastěší případ.
- Known Plaintext Attack (KPA) – Útočník zná zašifrovaný text a otevřený text a snaží se zjistit klíč.
- Chosen Plaintext Attack (CPA) – Útočník zná to co v KPA a navíc si text může zvolit.

**Útok silou** Pří útoku silou (*brute force*) zkouší útočník všechny teoreticky možné klíče, dokud nenajde ten správný.




Obrázek 50.2: Princip kryptografie, podle typu klíčů dělíme na symetrickou (tajný klíč) a asymetrickou (veřejný klíč, soukromý klíč).

**Bezpečný algoritmus** V moderní kryptografii je nepřijatelné utajování algoritmů (*security by obscurity*) – předpokládáme, že útočník zná šifrovací algoritmus. Bezpečnost musí záviset pouze na utajení klíče (Kerckhoffuv princip, *security by design*). Symetrický algoritmus je považován za bezpečný, pokud neexistuje rychlejší útok než útok silou.

**Délka klíče** Dnes je považováno 80 bitů a více za dostatečné. Typicky se délka zaokrouhluje na mocninu 2 (typicky 128b). Klíče symetrických algoritmů jsou kratší než asymetrických. Konkrétně: DES – 56b, 3DES – 112, AES – variabilní.

**Využití** Symetrická kryptografie je vhodná pro šifrování většího objemu dat. Narozdíl od asymetrické, která je pro tento účel příliš pomalá. Proto např. HTTPS využívá asymetrickou kryptografii pro výměnu symetrických klíčů a poté symetrickou kryptografii pro šifrování provozu.

**Vlastnosti moderní kryptografie** Symetrická kryptografie zaručuje všechny následující, kromě nepopiratelnosti – více entit má k dispozici klíč.

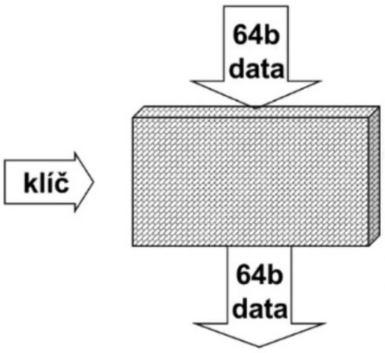
- Důvernost – Utajení informace. Bez znalosti klíče, není možné data číst.
- Autentizace – Prokázání, že zprávu skutečně poslal odesílatel a nikoliv útočník, který se za odesílatele vydává.
- Integrita – Prokázání, že nikdo nemohl data po cestě od odesílatele k příjemci změnit. Ochrana proti neoprávněné, neodhalené modifikaci zprávy.
- Nepopiratelnost – Pokud odesílatel data poslal, nemůže tuto skutečnost popřít.

## 50.3 Blokové šifry

Blokové šifry šifrují data po blocích pevně stanovené délky (64b, 128b, 256b, ...). Pokud je dat více, rozdělí se na více bloků, přičemž do zbylého místa v posledním je umístěno zarovnání *padding* (informace o délce zarovnání může být obsažena v posledním bytu). Příklady blokových šifer:

- Feistelova šifra (spíše princip)
- Data Encryption Standard (DES)
- Triple Data Encryption Algorithm (3DES)
- International Data Encryption Algorithm (IDEA)

- Blowfish
- Tiny Encryption Algorithm (TEA)
- Advanced Encryption Standard (AES)




Obrázek 50.3: Princip blokových šifer.

### 50.3.1 Feistelova šifra

Feistelova šifra (Feistelův princip) je koncept šifrování, který konkrétní algoritmy využívají. Jedná se o substituční-permutační síť. Vstupní blok je rozdělen na dvě poloviny $L$ a $R$, výpočet výstupu pak vypadá následovně.

$$L_i = R_{i-1} \tag{50.1}$$

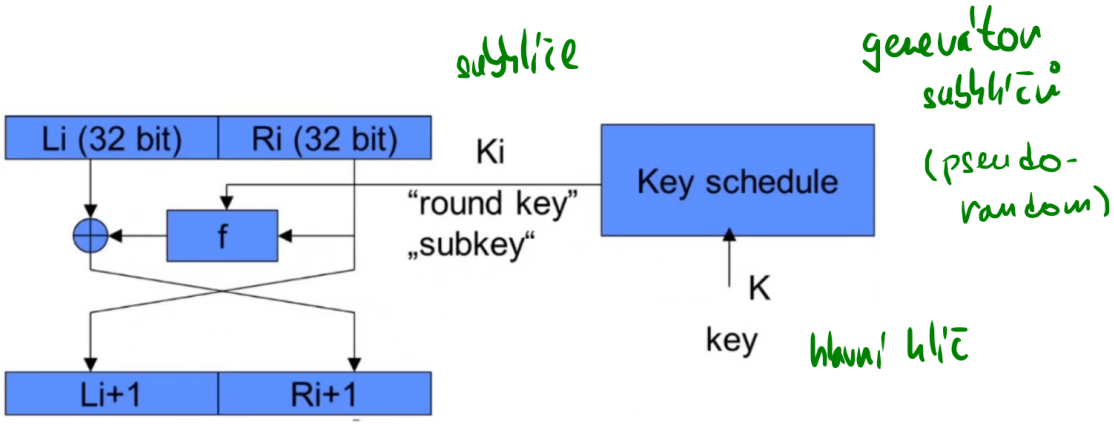
$$R_i = L_{i-1} \oplus F(R_{i-1}, K_i) \tag{50.2}$$

**Funkce F** $F$ je funkce, na kterou Feistelova šifra neklade žádné požadavky. Jednotlivé algoritmy, využívající Festelovu šifru, funkci samy definují. Požadavky na funkci $F$, aby algoritmus byl bezpečný:

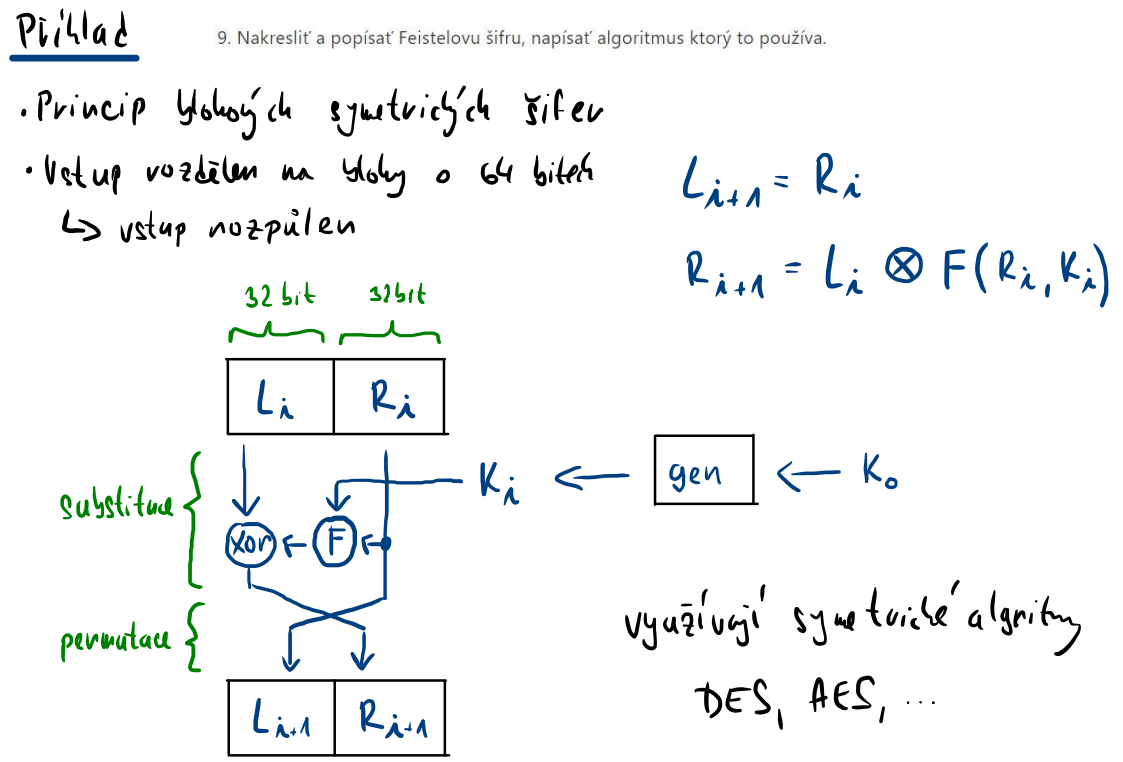
- skrytí vlastností zprávy;
- skrytí vlastností zprávy.

**Subklíč** K je tzv. subklíč, který je generován typicky nějakým pseudonáhodným generátorem na základě inicializačního klíče (hlavní).

**Dešifrování** Dešifrování se provádí stejným způsobem, pouze pořadí subklíčů je opačné.




Obrázek 50.4: Jeden krok opakování (Feistelův krok) vizuálně.




Obrázek 50.5: Feistelova šifra – příklad a rekapitulace.

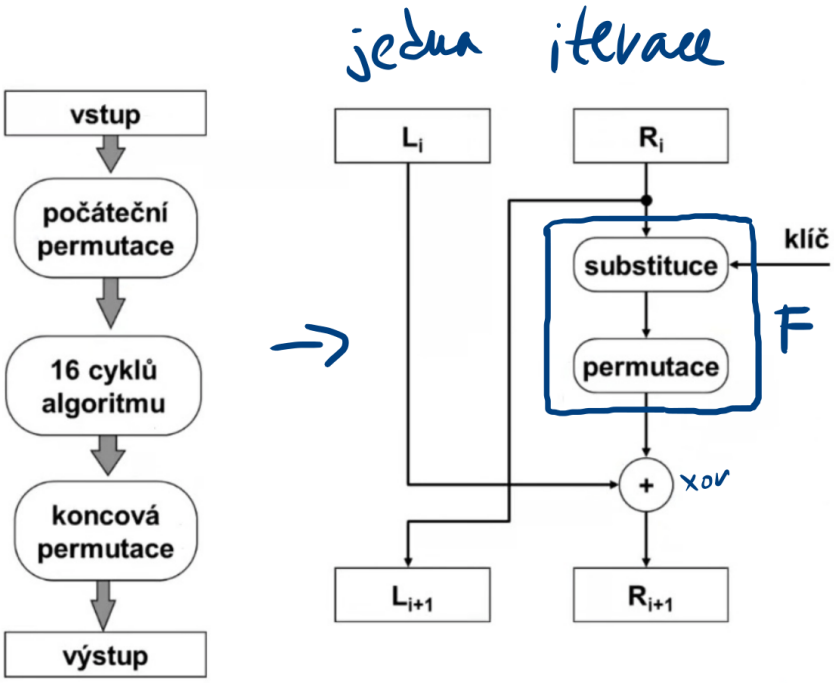
### 50.3.2 Data Encryption Standard (DES)

DES byl první algoritmus s veřejnou specifikací (*security by design*). Využívá princip Feistelovy šifry – 16 kol. Dodatečně přidává na začátek a konec permutaci navíc. Klíč je dlouhý 64b (resp. 56 významových bitů a 8 paritních).

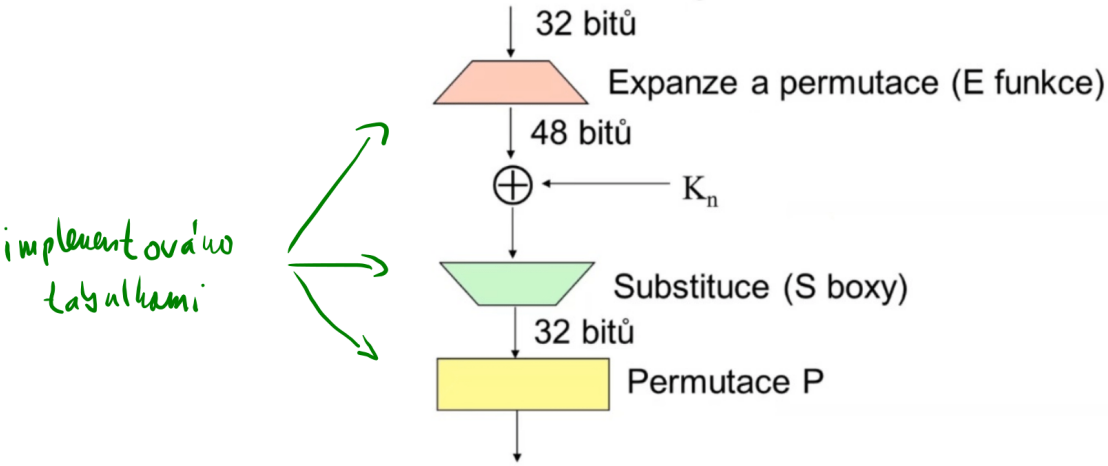
**Slabiny**

- 56 bitový klíč je příliš krátky a je možný útok silou.

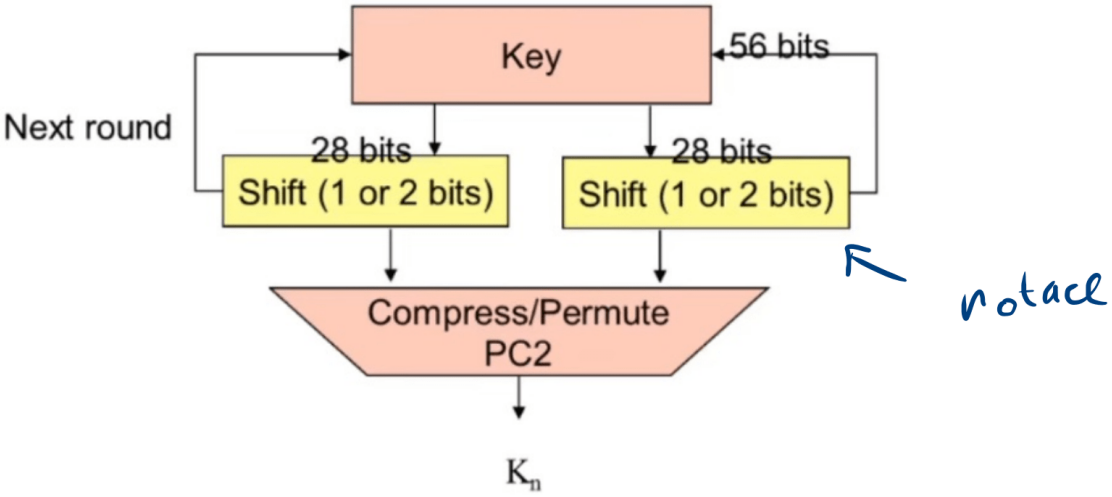
- Rozdílná velikost bloku a klíče (zvláštnost).
- Existence slabých a poloslabých klíčů.
- Není jasné proč zrovna 16 iterací a zda je to dostatečné.




Obrázek 50.6: DES – Schéma fungování algoritmu.




Obrázek 50.7: DES – funkce F.




Obrázek 50.8: DES – generování subklíčů.

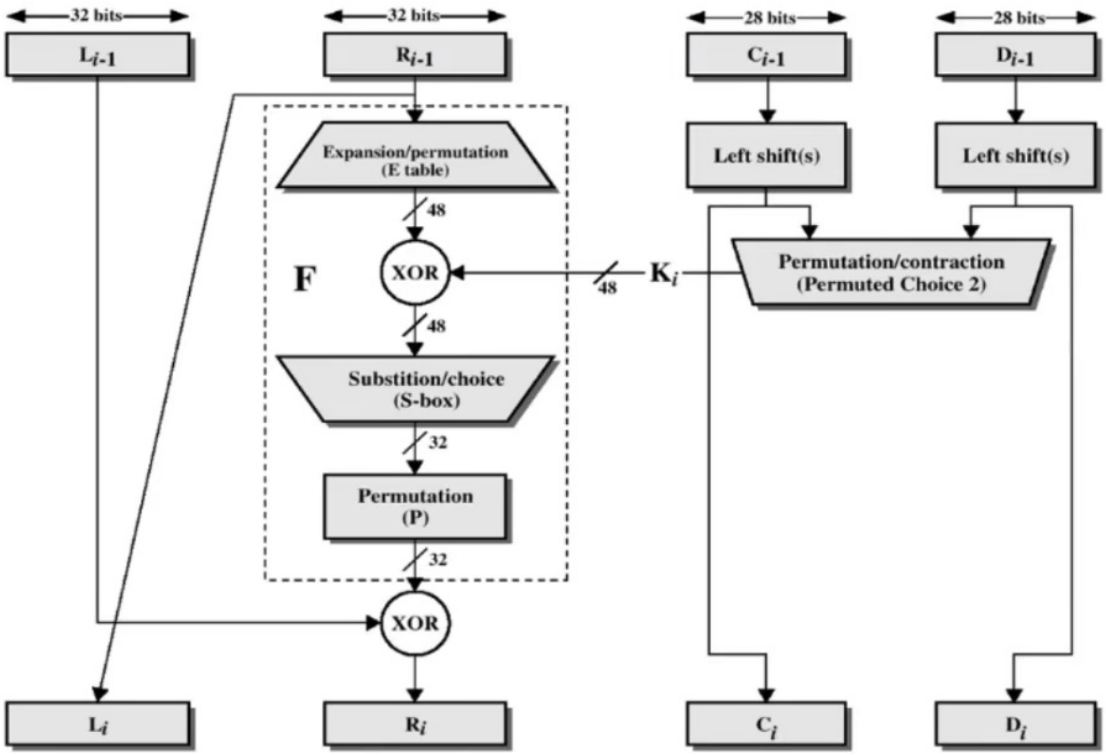
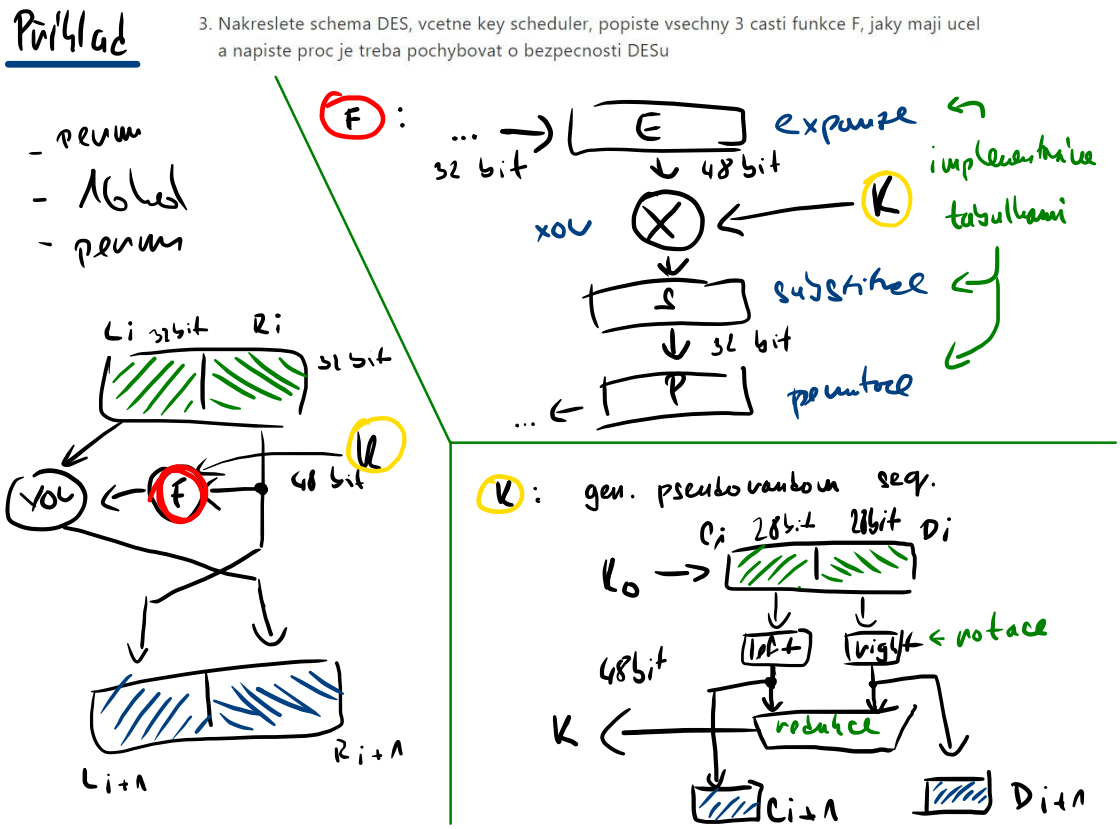



**Figure 2.4 Single Round of DES Algorithm**

Obrázek 50.9: DES – jedno kolo algoritmu. *Left shift* je ve skutečnosti bitová rotace.



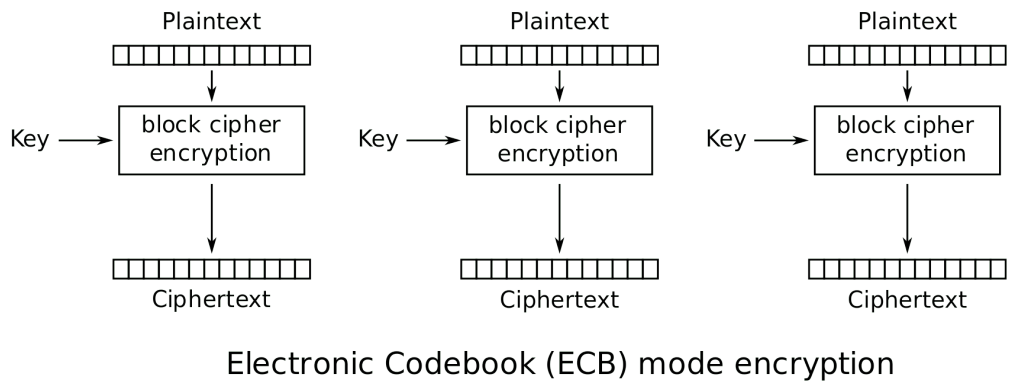

Obrázek 50.10: DES – příklad a rekapitulace.

## 50.4 Provozní režimy činnosti blokových šifer

Jak použít blokové šifry abychom byli schopni šifrovat data delší než jeden blok?

### 50.4.1 Electronic Code Book (ECB)

ECB („kódová kniha“) je výchozí *naivní* režim. Bloková šifra se při něm přímo aplikuje nezávisle na jednotlivé bloky, tedy při daném klíči odpovídá stejnému bloku otevřeného textu stejný blok šifrového textu. To má nežádoucí důsledky z hlediska bezpečnosti, v datech zůstane původní struktura, např. šifrovaný obrázek je rozpoznatelný.

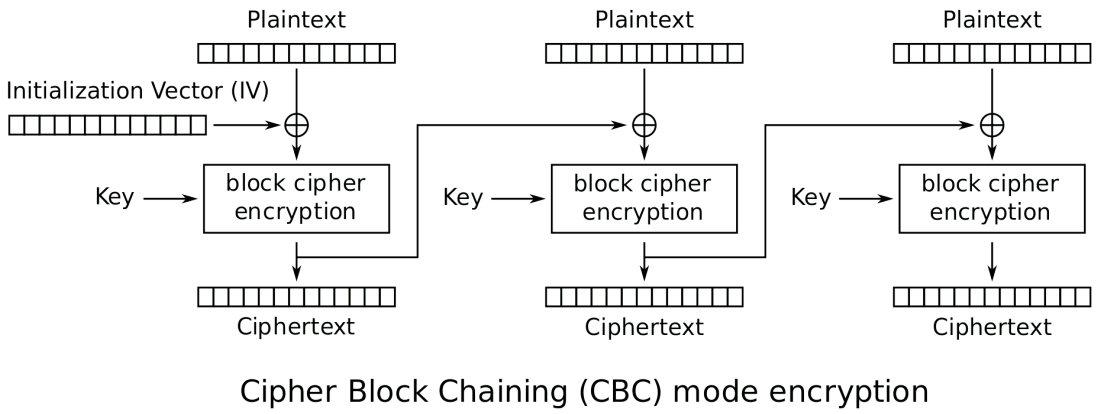



Obrázek 50.11: Ukázka režimu ECB.

### 50.4.2 Cipher Block Chaining (CBC)

V režimu CBC („řetězení šifrových bloků“) je každý blok před šifrováním xorován zašifrovaným předchozím blokem a první blok je xorován inicializačním vektorem. Tento režim je široce používán. Nevýhody plynou ze zřetězené závislosti (šifrovaný blok závisí na všech předcházejících): Šifrování nelze paralelizovat a při poškození šifrového bloku nelze dešifrovat ani blok přímo následující. Dešifrování paralelizovat lze.

$$\begin{aligned} C_i &= E_K(P_i \oplus C_{i-1}) \\ P_i &= D_K(C_i) \oplus C_{i-1} \\ C_0 &= IV \end{aligned} \tag{50.3}$$

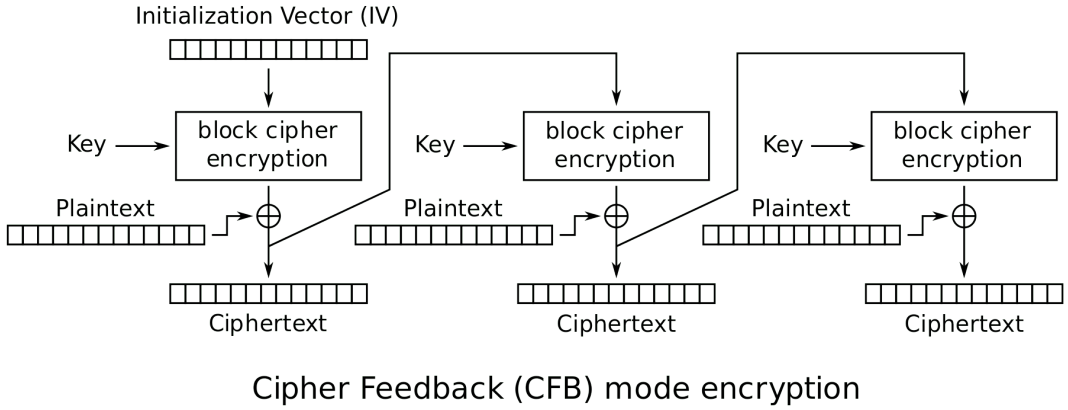



Obrázek 50.12: Ukázka režimu CBC.

### 50.4.3 Cipher Feedback (CFB)

Režim CFB (*šifrová zpětná vazba*) se liší oproti CBC v prohození pořadí operací xor a šifrování – nejprve se zašifruje předchozí šifrovaný blok (resp. inicializační vektor) a výsledek se xoruje s otevřeným blokem. Toto prohození má významné implementační dopady: díky symetrii operace XOR vypadá dešifrovací funkce obdobně jako šifrovací. Šifruje pomaleji než CBC. Vstup není nutné zarovnávat. Plynou stejné nevýhody jako pro CBC.

$$
\begin{aligned}
C_i &= E_K(C_{i-1}) \oplus P_i \\
P_i &= E_K(C_{i-1}) \oplus C_i \\
C_0 &= IV
\end{aligned}
\tag{50.4}
$$



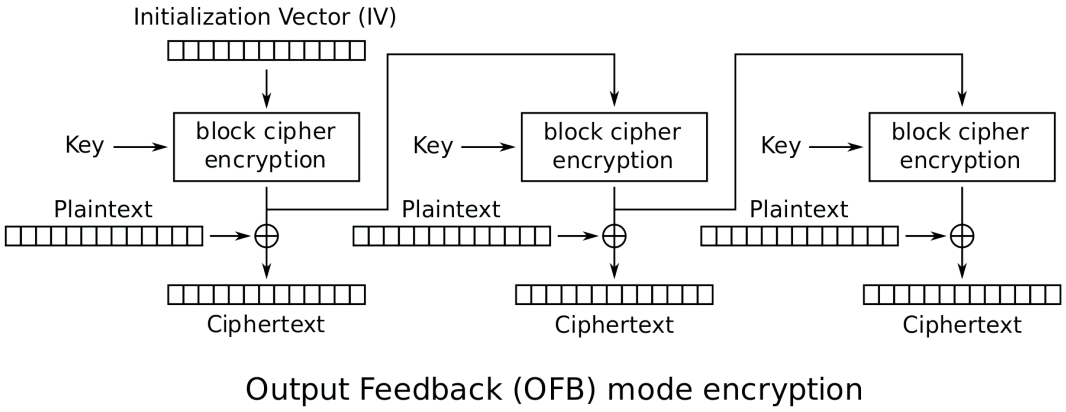

Obrázek 50.13: Ukázka režimu CFB.

### 50.4.4 Output Feedback (OFB)

Režim OFB (*výstupní zpětná vazba*) se liší od CFB pouze v tom, kde bere zpětnou vazbu. Šifrování probíhá pouhým xorováním otevřeného bloku s heslem, které je v každém kroku zašifrováno použitou blokovou šifrou. První blok hesla je získán zašifrováním inicializačního vektoru. Režim převádí blokovou šifru na synchronní proudovou šifru.

**Slabina** Celý blok encryption je pouze generátor pseudonáhdon posloupnosti (je nezávislá na otevřeném nebo šifrovaném textu). To umožňuje Known Plaintext Attack. Z toho plyne, že jedním klíčem není bezpečné šifrovat více než jednu zprávu.

$$
\begin{aligned}
C_i &= E_K(C_{i-1}) \\
P_i &= P_i \oplus C_i \\
C_0 &= IV
\end{aligned}
\tag{50.5}
$$

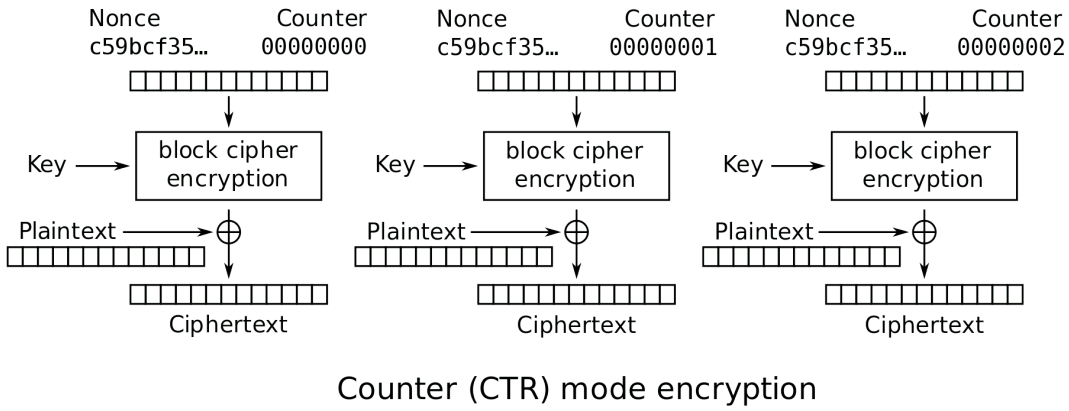



Obrázek 50.14: Ukázka režimu OFB.

### 50.4.5 Counter (CTR)

Režim CTR („čítačový režim“) převádí stejně jako OFB blokovou šifru na synchronní proudovou. Heslo, se kterým se blok otevřeného textu xoruje, je však získáno zašifrováním čítače, který se každou iteraci zvětšuje o pevně danou hodnotu, zpravidla o 1. Obsah čítače je opět před šifrováním nastaven inicializačním vektorem. Každý blok je šifrován nezávisle na ostatních, díky tomu je možné paralelizovat.

$$\begin{aligned} CTR_i &= CTR_{i-1} + 1 \\ P_i &= P_i \oplus E_k(CTR_i) \\ CTR_0 &= IV \end{aligned} \tag{50.6}$$



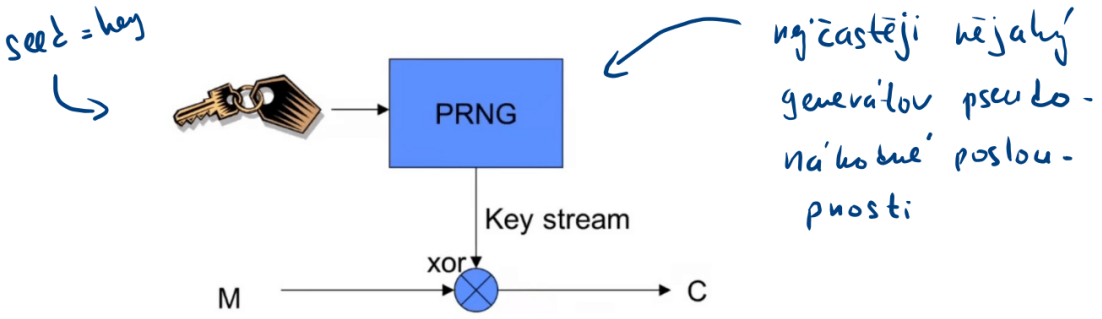

Obrázek 50.15: Ukázka režimu CTR.

## 50.5 Proudové šifry

Proudové šifry šifrují data jako *proud* (stream), nejčastěji po jednotlivých bytech. Dešifrování vždy probíhá stejným způsobem. Proudové šifry jsou rychlejší než blokové šifry a pro implementaci potřebují jednodušší hardware.

**Problémy**

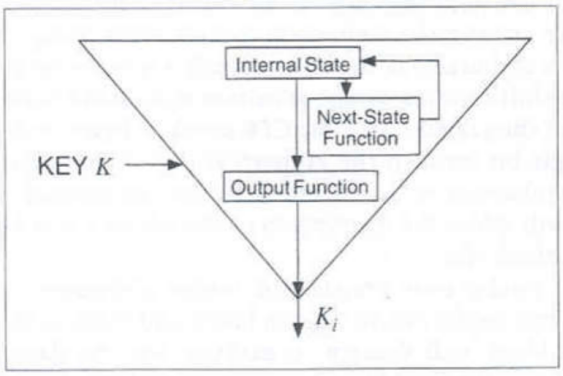
- Nezajišťují samy o sobě integritu.
- Na rozdíl od blokových šifer jsou náchylnější ke kryptoanalytickým útokům, pokud jsou nevhodně implementovány (počáteční stav nesmí být použit opakovaně) – „problém s inicializačním vektorem“.




Obrázek 50.16: Princip proudových šifer.

### 50.5.1 Rozdělení proudových šifer

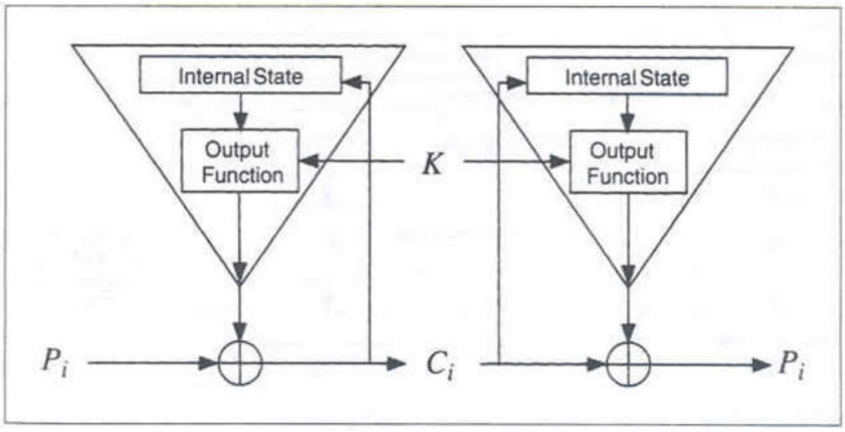
**Synchronní proudové šifry** Proud pseudonáhodných čísel *key stream* je generován nezávisle na vstupním textu nebo zašifrované zprávě. Poté dochází ke kombinaci vygenerovaných čísel se vstupujícím textem (k zakódování) nebo se šifrovaným textem (k dekódování). Nejběžnější formou kombinace keystreamu a vstupního textu je použití operace XOR. Např.: Vernamova šifra, DES v režimu OFB. Pokud se průběhu dešifrování něco ztratí, je konec.




Obrázek 50.17: Princip synchronní proudové šifry.

**Samosynchronizující proudové šifry** Proud pseudonáhodných čísel *key stream* závisí na pevném počtu předcházejících bytů šifrovaného (nebo otevřeného) textu. To znamená, že se šifra dokáže po chybě sama *zotavit* (resynchronizovat)[1]. Např.: Vigenere Autokey, DES v řežimu CFB.

[1]Dnes se nesnážíme dešifrovat poškozená data, pokud nastane chyba v přenosu, vyžádáme si data znovu.




Obrázek 50.18: Princip samosynchronizující proudové šifry.

### 50.5.2 Generátory PRNG

Generátory PRNG (*pseudo-random number generator*) generují pseudo-náhodnou posloupnost (*key stream*) z malého klíče (*seed*).

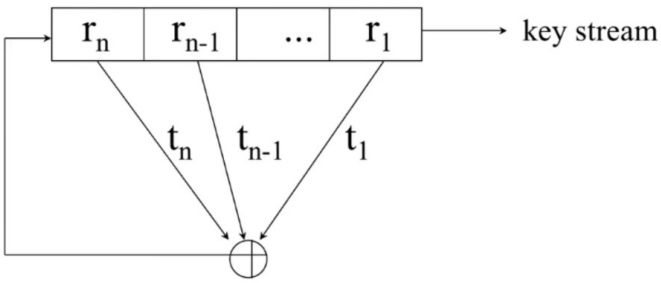
**Blokové šifry v režimu OFB** Blokové šifry v režimu OFB jsou pomalé.

**Linear Feedback Shift Registers (LSFR)** LFSR (posuvný registr s lineární zpětnou vazbou) je posuvný registr, jehož výstup je lineárně závislý na jeho předchozích výstupech a stavu. Mějme

- posuvný registr $R = (r_1, r_2, \ldots, r_n)$,
- sekvenci zpětných vazeb $T = (t_1, t_2, \ldots, t_n)$.

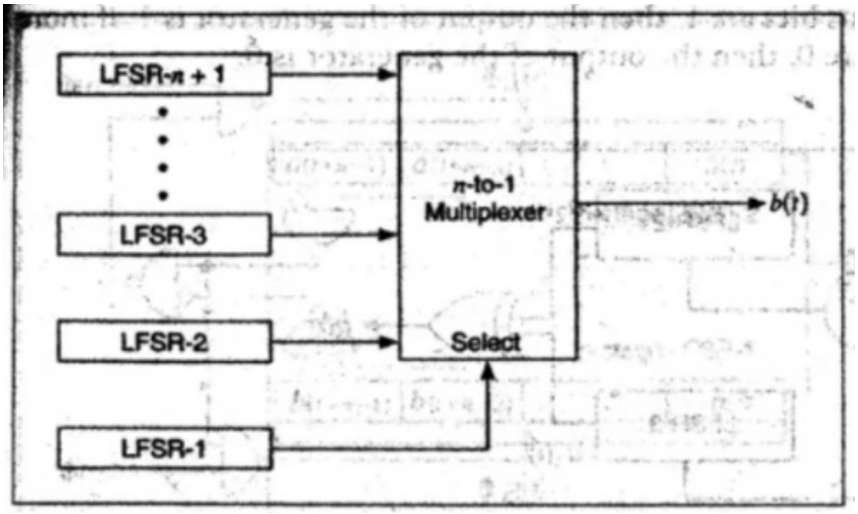
Alternativně lze zapsat polynomem:

$$T(x) = x^n + x^{n-1} + \ldots + x + 1$$



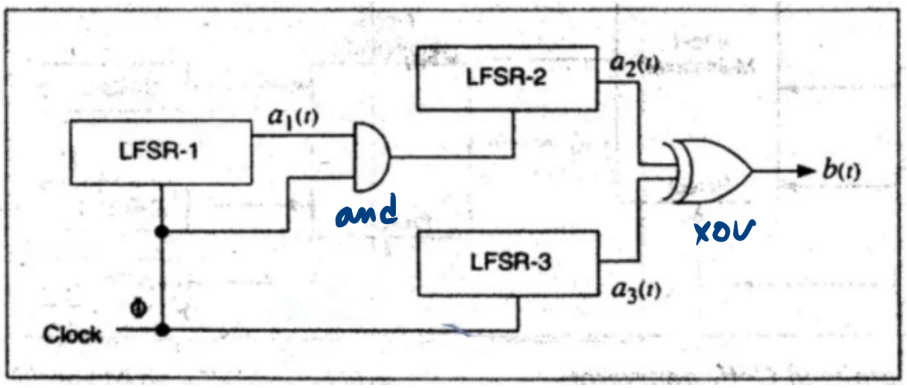

Obrázek 50.19: Příklad LFSR.

**Geffe generátor** Geffe generátor (kombinovaný generátor) je využití 2 a více LFSR propojených multiplexorem.




Obrázek 50.20: Příklad Geffe generátoru.

**Stop and Go generátor** Stop and Go generátor je několik LSFR s různým zdrojem hodin.




Obrázek 50.21: Příklad Stop and Go generátoru.

# Kapitola 51

# KRY – Asymetrická kryptografie, vlastnosti, způsoby použití, poskytované bezpečnostní funkce, elektronický podpis a jeho vlastnosti, hybridní kryptografie, algoritmus RSA, generování klíčů, šifrování, dešifrování.

## 51.1 Zdroje

- `KRY04_Asym_MNG.pdf`
- `KRY_2021-03-08.mp4`
- `KRY_2021-03-22.mp4`
- `https://algoritmy.net/article/4033/RSA`

## 51.2 Úvod a kontext

**Asymetrická kryptografie**

- V asymetrické kryptografii se používají páry klíčů (soukromý a veřejný). Soukromý je používán k dešifrování, resp. vytvoření digitálního podpisu. Veřejný je používán k šifrování, resp. ověření digitálního podpisu.
- Každý uživatel generuje svůj pár klíčů. Veřejný klíč je zveřejněn (znají ho všichni), sourkomý je držen v tajnosti (zná ho pouze vlastník).
- Všechny asymetrické algoritmy jsou blokové.
- Asymetrické algoritmy jsou pomalejší než symetrické.

**Způsoby použití** Asymetrická kryptografie lze využít k:

- šifrování,

- digitálnímu podepisování,
- pro výměnu symetrického klíče (*key exchange*).

**Vlastnosti** Vlastnosti symetrické a asymetrické kryptografie[1].

| | Důvěrnost | Autentizace | Integrita | Nepopiratelnost |
|---|---|---|---|---|
| Symetrická | ano | ? | ? | ne |
| Asymetrická - šifrování | ano | ? | ? | ne |
| Asymetrická - podepisování | ne | ano | ano | ano |
| Asymetrická - kombinace | ano | ano | ano | ano |

```
# Odesilatel (A):
msg = encrypt(msg, SK_A) # necht msg je zprava k odeslani
msg = encrypt(msg, PK_B)
send(msg_2)

# Prijemce (B):
msg = receive()
msg = decrypt(msg, SK_B)
msg = decrypt(msg, PK_A)
```

Výpis 51.1: Kombinace klíčů obou stran u asymetrický kryptografie. Pořadí operací může být i opačné.

**Digitální podpis** Vytvoření digitálního podpisu konkrétních dat pomocí soukromého klíče podepisatele. Každý kdo zná veřejný klíč podepisatele, může pravost podpisu ověřit. Digitální podpis zajišťuje autentizaci, itegritu a nepopiratelnost.

**Algoritmy** Algoritmy asymetrické kryptografie se nedají *vymyslet*, musí se objevit. Jsou založeny na těžkých matematických problémech.

- Problém batohu (*knapsack problem*) – MH (Merkle-Hellman)
- Faktorizace čísel – RSA (Rivest-Shamir-Adleman)
- Diskrétní logaritmus – DSA (Digital Signature Algorithm), DH (Diffie-Hellman)
- Eliptické křivky – ECDSA, ECDH

**Problém batohu** Problém batohu je NP-úplný problém kombinatorické optimalizace. Nechť $x_1, x_2, \ldots, x_n$ je množina objektů, každý objekt má svoji cenu $v_i$ a svoji hmotnost $w_i$, dále mějme batoh, který má kapacitu $W$. Cílem je vybrat takovou množinu objektů, jejichž hmotnost je menší nebo rovna $W$ a má nejvyšší možnou cenu[2]. Formálně chceme maximalizovat sumu

$$\sum_{i=1}^{n} v_i \cdot x_i$$

, při splnění

$$\sum_{i=1}^{n} w_i \cdot x_i \leq W$$

[1] Otazníky – částečně, za předpokladů, ...
[2] Problém má více obdobných variant.

, kde $x_i \in x_1, x_2, \ldots, x_n$.

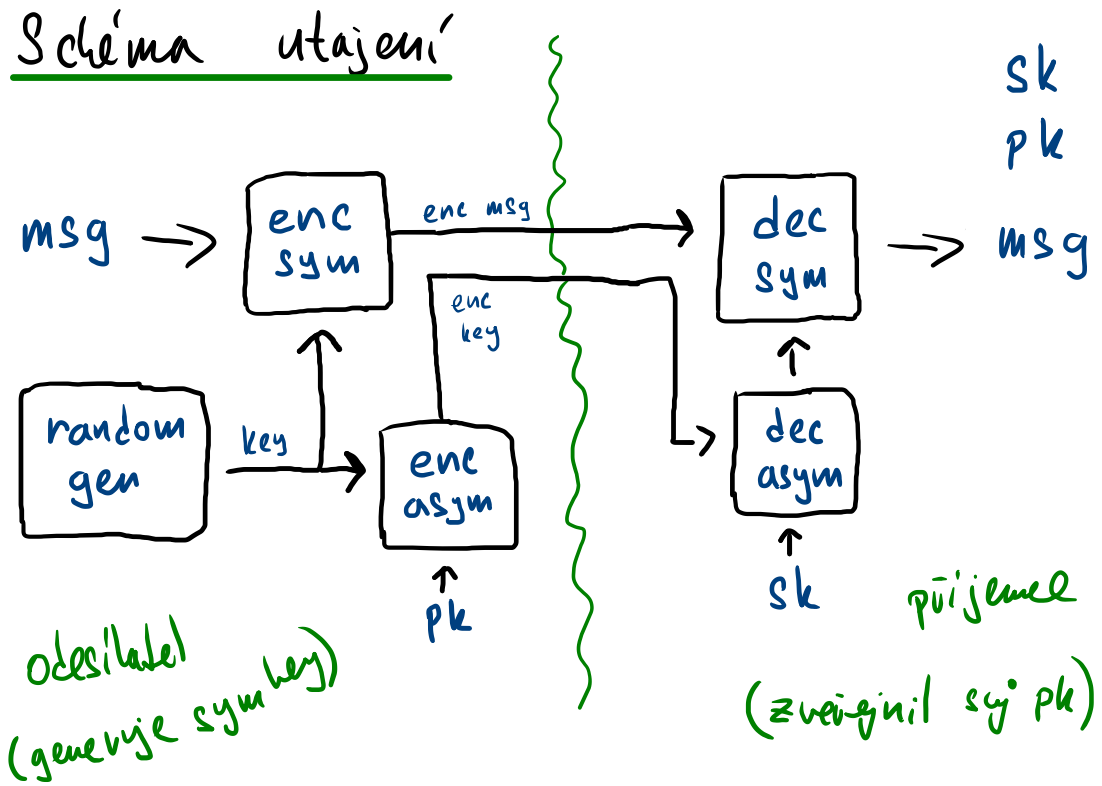
**Faktorizace čísel** Faktorizace čísel označuje problém rozložení čísla na součin menších čísel, v nejčastější podobě pak rozklad celého čísla na součin prvočísel.

**Diskrétní logaritmus** Nechť $p, g, k, Y$ jsou přirozená čísla, pro něž platí $Y \equiv g^k \bmod p$. Potom každé číslo $k$ odpovídající uvedené rovnici nazveme diskrétní logaritmus o základu $g$ z $Y$ vzhledem k modulu $p$. Tato definice nedefinuje číslo $k$ jednoznačně, proto se někdy upravuje tak, že ze všech možných diskrétních logaritmů ve smyslu předchozí definice se vybere ten nejmenší.

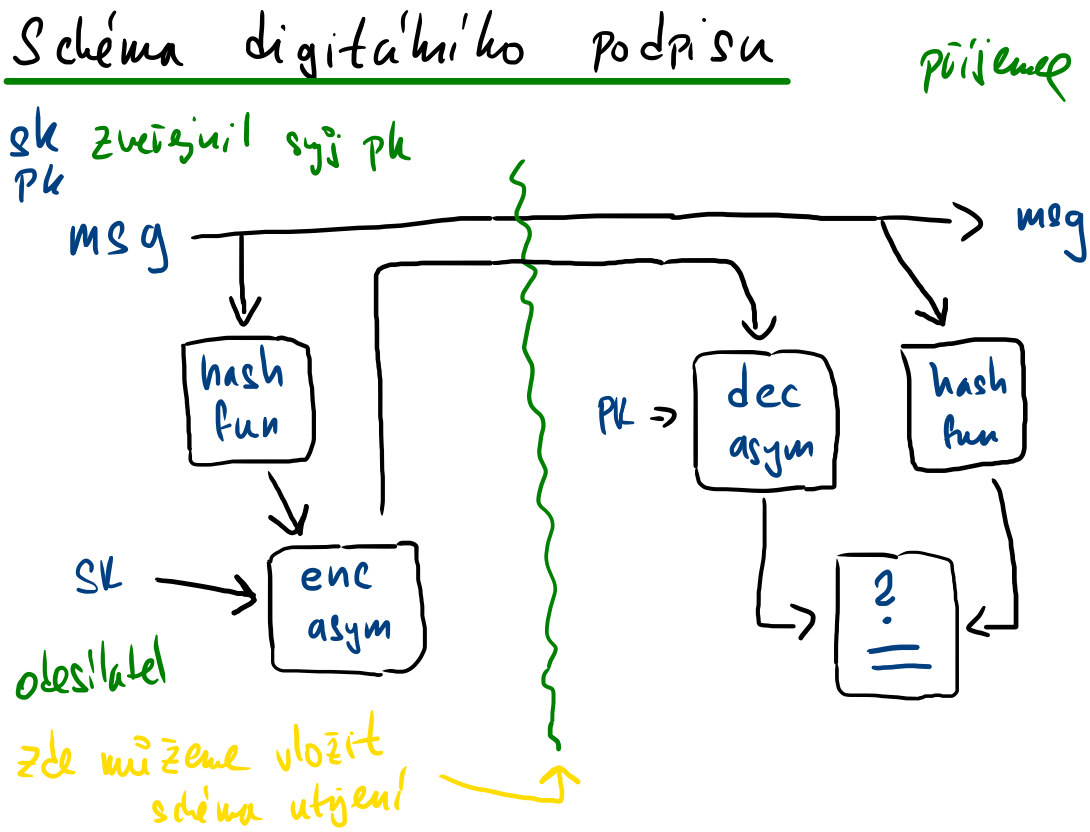
**Eliptické křivky** Jedná se o matematický aparát, na kterém aplikujeme různé algoritmy (DSA, DH).

## 51.3 Hybridní kryptografie

Hybridní kryptografie je kombinace symetrické a asymetrické kryptografie, ve které jsou využity přednosti obou (symetrická – rychlá, ale potřeba stejný klíč; asymetrická – pomalá, ale dva klíče). Asymetrická je využita pro bezpečné zaslání symetrického klíče.




Obrázek 51.1: Schéma utajení hybridní kryptografie.




Obrázek 51.2: Schéma digitálního podpisu hybridní kryptografie.

## 51.4 RSA

Algoritmus RSA (Rivest-Shamir-Adleman) lze použít jak pro šifrování dat pro digitální podepisování. Je založen na problému faktorizace velkých čísel.

**Klíče** Klíče se skládají z:

- $p, q$ – dvě náhodná soukromá prvočísla,
- $n$ – veřejný modul ($n = p \cdot q$),
- $e$ – veřejný exponent ($e < \Phi(n) \ \wedge \ GCD(\Phi(n), e) = 1$), typicky 3 nebo $2^{16} + 1$[3],
- $d$ – soukromý exponent,
- musí platit vztah: $e \cdot d \bmod \Phi(n) = 1$.

Veřejný klíč $PK = (n, e)$, soukromý klíč $SK = (n, d)$.

**Postup generování** Postup generování klíčů:

1. vygenerovat prvočísla $p$ a $q$,
2. spočítat modul $n = p \cdot q$,
3. spočítat $\Phi(n) = (p - 1) \cdot (q - 1)$,

[3] $GCD$ – největší společný dělitel

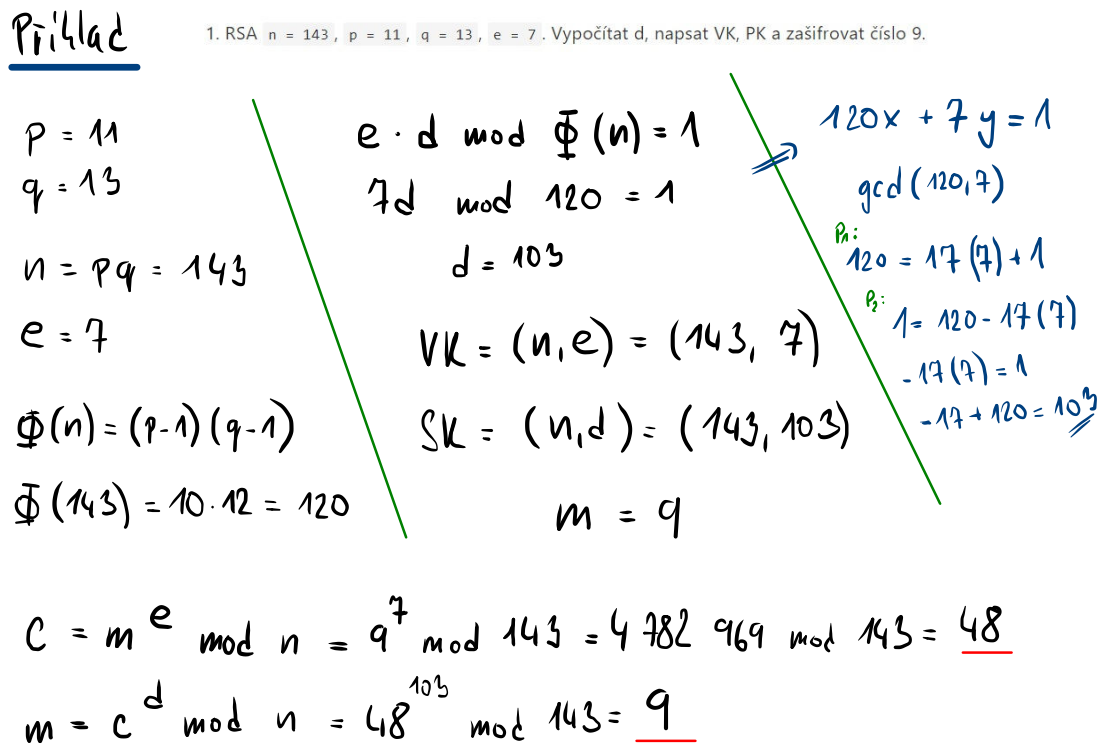
4. zvolit veřejný exponent $e < \Phi(n) \ \wedge \ GCD(\Phi(n), e) = 1$,
5. spočítat soukromý exponent $d$ tak, že platí $e \cdot d \bmod \Phi(n) = 1$.

**Šifrování a dešifrování** Mějme zprávu $m$ reprezentovanou jako celé číslo a zašifrovanou zprávu $c$ reprezentovanou také jako celé číslo. Digitální podpis se vytváří stejným způsobem, pouze se prohodí exponenty.

$$c = m^e \bmod n \tag{51.1}$$

$$m = c^d \bmod n \tag{51.2}$$

**Útoky a slabiny** Pokud útočník rozloží číslo $n$ na činitele $p$ a $q$, tak může dopočítat soukromý klíč. Pokud útočník uhádně hodnotu $(p-1)\cdot(q-1)$, tak může dopočítat soukromý klíč. Šifrování malých čísel je zranitelné, proto se používá „předzpracování“ – zarovnání na $X$ bitů (2048).




Obrázek 51.3: Příklad RSA.

# Kapitola 52

# KRY – Hašovací funkce, klíčovaný haš a MAC a jejich použití a vlastnosti.

## 52.1 Zdroje

- `KRY04_Asym_MNG.pdf`
- `KRY_2021-03-22.mp4`

## 52.2 Úvod a kontext

**Hashovací funkce** Hashovací funkce je funkce (resp. algoritmus) pro převod vstupních dat do (relativně) malého čísla. Výstup hashovací funkce se označuje otisk, *fingerprint*, *digest* či *hash*. Jsou jednosměrné a odolné proti kolizím (viz vlastnosti).

**Obecné vlastnosti** Hashovací funkce by měla:

- Být aplikovatelná na argument o libovolné velikosti.
- Mít výstyp konstantní délky.
- Dokázat spočítat výstup rychle.

**Neklíčované hashovací funkce** Hashovací funkce má pouze jeden argument – data. Např. MD2, MD4, MD5, SHS, SHA1, SHA2, SHA3.

$$f(data) \rightarrow hash$$

**Klíčované hashovací funkce** Hashovací funkce má dva argumenty – data a klíč. Také se jim někdy říká MAC (*message authentication code*).

$$f(data, key) \rightarrow hash$$

**Lavinový efekt** Lavinový efekt *Avalanche Effect* je žádoucí vlastností kryptografických algoritmů, typicky blokových šifer a kryptografických hašovacích funkcí, kdy se při nepatrné změně vstupu (například převrácení jednoho bitu) výrazně změní výstup (např. převrátí se polovina výstupních bitů). V případě kvalitních blokových šifer by taková malá změna klíče nebo otevřeného textu měla způsobit drastickou změnu šifrového textu.

## 52.3 Kryptografická odolnost hashovacích funkcí

**Vlastnosti z hlediska odolnosti** Hashovací funkce by z hlediska kryptografické odolnosti měly splňovat:

- *First preimage resistance* – Pro konkrétní $y$ je výpočetně nezvládnutelné najít takové $x$, aby platilo $h(x) = y$. Útočník má k dispozici konkrétní hash, a snaží se pro něho nalézt zprávu.
- *Second preimage resistance* – Pro konkrétní $x$ je výpočetně nezvládnutelné najít takové $x'$, aby platilo $h(x) = h(x')$. Útočník má k dispozici konkrétní zprávu (nemůže si ji zvolit), ke které se snaží nalézt jinou zprávu, která bude mít stejný hash.
- *Collision resistance* – Je výpočetně nezvládnutelné najít libovolnou dvojici $x, x'$ takovou, aby platilo $x \neq x'$ a $h(x) = h(x')$. Útočník si může zvolit libovolnou zprávu, ke které se snaží nalézt jinou zprávu, která bude mít stejný hash. Pokud platí *collision resistance*, tak platí i *second preimage resistance*.

**Narozeninový problém** V teorii pravděpodobnosti je narozeninový problém úloha vypočítat minimální početnost skupiny lidí, ve které je alespoň 50% pravděpodobnost nalezení dvojice se stejným datem narození. Narozeninovým paradoxem je pak označována skutečnost, že tento počet (23) je mnohem menší než intuitivní odhad. Výsledek je intuitivnější, když uvážíme, že porovnání narozenin bude provedeno mezi všemi možnými dvojicemi jedinců. Při počtu 23 jedinců je třeba uvažovat $(23 \cdot 22)/2 = 253$ dvojic, což je více než polovina počtu dnů v roce $(182,5)$.

Jednodušší je nejprve spočítat jev opačný $\bar{p}(n)$, tedy pravděpodobnost, že všech $n$ narozenin je rozdílných. Pro $n > 365$ je 1, jinak:

$$\begin{aligned}\bar{p}(n) &= 1 \cdot \left(1 - \frac{1}{365}\right) \cdot \left(1 - \frac{2}{365}\right) \cdots \left(1 - \frac{n-1}{365}\right) = \\ &= \frac{365 \cdot 364 \cdots (365 - n + 1)}{365^n} = \\ &= \frac{365!}{365^n (365 - n)!}\end{aligned} \tag{52.1}$$

$$p(n) = 1 - \bar{p}(n) \tag{52.2}$$

**Narozeninový útok** Mějme hashovací funkci, která má $n$ bitový výstup (celkový počet možných hashů je $2^n$). Útočník vytvoří dokument „přátelská dohoda“ a přibližně $2^{n/2}$ sémanticky ekvivalentních verzí (úprava bílých znaků, úprava pořadí celků, jiné formulace, ...). Podobně vytvoří dokument „nepřátelská dohoda“ a přibližně $2^{n/2}$ sémanticky ekvivalentních verzí. S pravděpodobností $0,5$ existuje verze „přátelské dohody“ a „nepřátelské dohody“, které mají stejný hash. Pokud takové verze existují, útočník dá oběti podepsat „přátelskou dohodu“ $\Rightarrow$ existuje validní podpis „nepřátelské dohody“.

**Bezpečnostní cíle OWHF** Bezpečnostní cíle OWHF (*one way hash function*). Některé protokoly nevyžadují bezkoliznost, proto má smysl řešit i tento případ.

- Vyžadované vlastnosti: *first preimage resistance* a *second preimage resistance*
- Cíl útočníka: vytvořit *first preimage* nebo *second preimage* (oba úkoly jsou stejně těžké)

- Složitost: $O(2^n)$ ($n$ je počet bitů hashe)
- Požadovaná délka: $n \geq 80$

**Bezpečnostní cíle CRHF** Bezpečnostní cíle CRHF (*collision resistance hash function*).

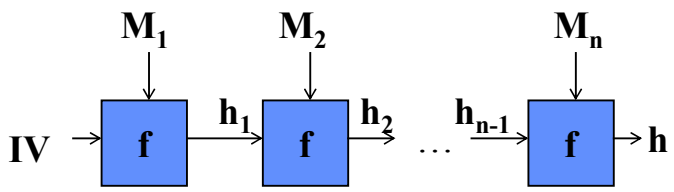
- Vyžadované vlastnosti: *collision resistance*
- Cíl útočníka: vytvořit kolizi
- Složitost: $O(2^{n/2})$ ($n$ je počet bitů hashe) (kvůli narozeninovému útoku)
- Požadovaná délka: $n \geq 160$

**Bezpečnostní cíle MAC** Bezpečnostní cíle MAC (*message authentication code*).

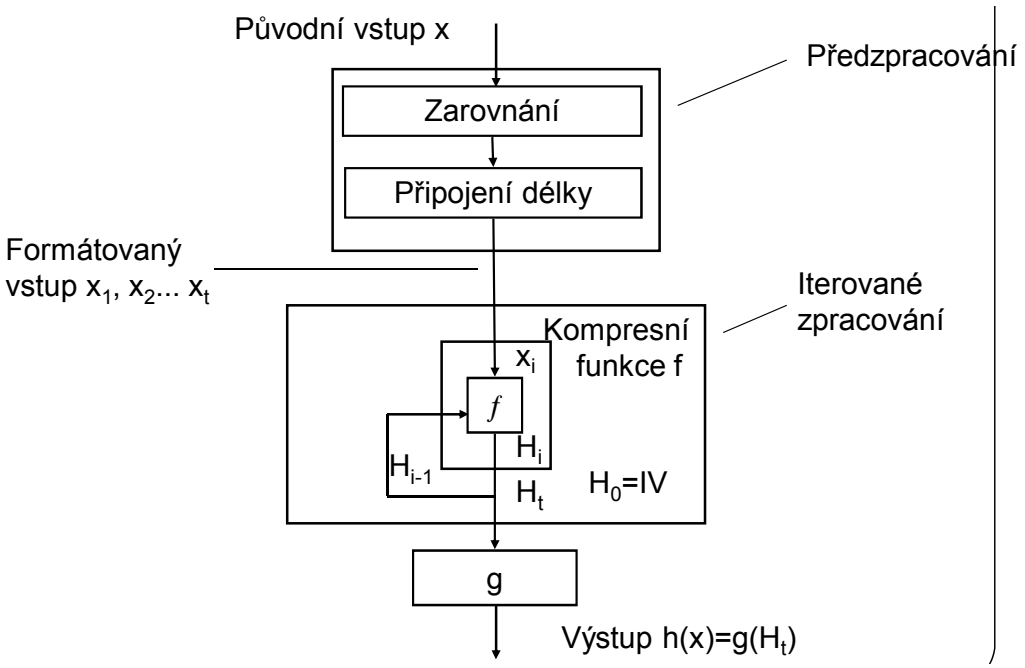
- Vyžadované vlastnosti: *computation resistance*, *key non-recovery*
- Cíl útočníka (útočník si může vybrat):
  - Vytvořit nový hash, který bude odpovídat nové zprávě
  - Nalézt klíč
- Složitost ($n$ je počet bitů hashe, $t$ je počet bitů klíče):
  - Vytvořit nový hash: $O(max(2^{-n}, 2^{-t}))$
  - Nalézt klíč: $O(2^n)$
- Požadovaná délka: $n \geq 64 \ \wedge \ t \geq 64$

## 52.4 Hashovací funkce neklíčované

Nejčastější způsoby sestrojení hashovací funkce neklíčované jsou založené na principu iterace.




Obrázek 52.1: Schéma iterativní neklíčované hashovací funkce. Zpráva je rozdělena na $n$ částí. $f$ je tzv. kompresní funkce. $IV$ je inicializační vektor, resp. konstanta. $h_1$ až $h_{n-1}$ jsou mezivýsledky („mezihashe“) a $h$ je výsledný hash.




Obrázek 52.2: Podrobnější schéma iterativní neklíčované hashovací funkce.

Jednotlivé kroky hashovací funkce:

- Předzpracování – Vstupní data jsou rozdělena na bloky o stejné délce. Je provedeno zarovnání posledního bloku. Je připojena informace o délce zprávy.
- Iterativní zpracování – V iteracích se postupně „přihashovávají“ vstupní bloky. Zpětná vazba pomocí stavové proměnné. Uvnitř kompresní funkce, která z delšího vstupu uděla kratší výstup.
- Postzpracování – Volitelný krok.

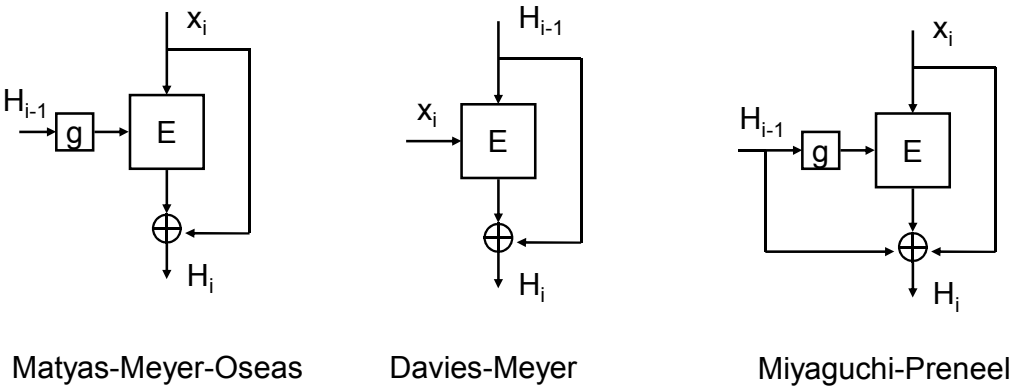
**Merkelova meta-metoda** Nechť $f$ je kompresní funkce odolná proti kolizím. Hashovací funkce $h$ na principu iterace využívající kompresní funkci $f$ je rovněž odolná proti kolizím.

**Merkel-Damgardovo zesílení** Pokud je do vstupu hashovací funkce vložena délka zprávy, tak je zajištěno, že žádná zpráva není prefixem jiné zprávy.

**Zarovnání** Nejednoznačné zarovnání (*ambiguous padding*) – připoj ke zprávě tolik bitů, aby délka zprávy byla násobkem délky bloku. Jednoznačné zarovnání (*unambiguous padding*) – připoj ke zprávě jeden bit a poté proveď nejednoznačné zarovnání.

### 52.4.1 Hashovací funkce s využitím blokových šifer

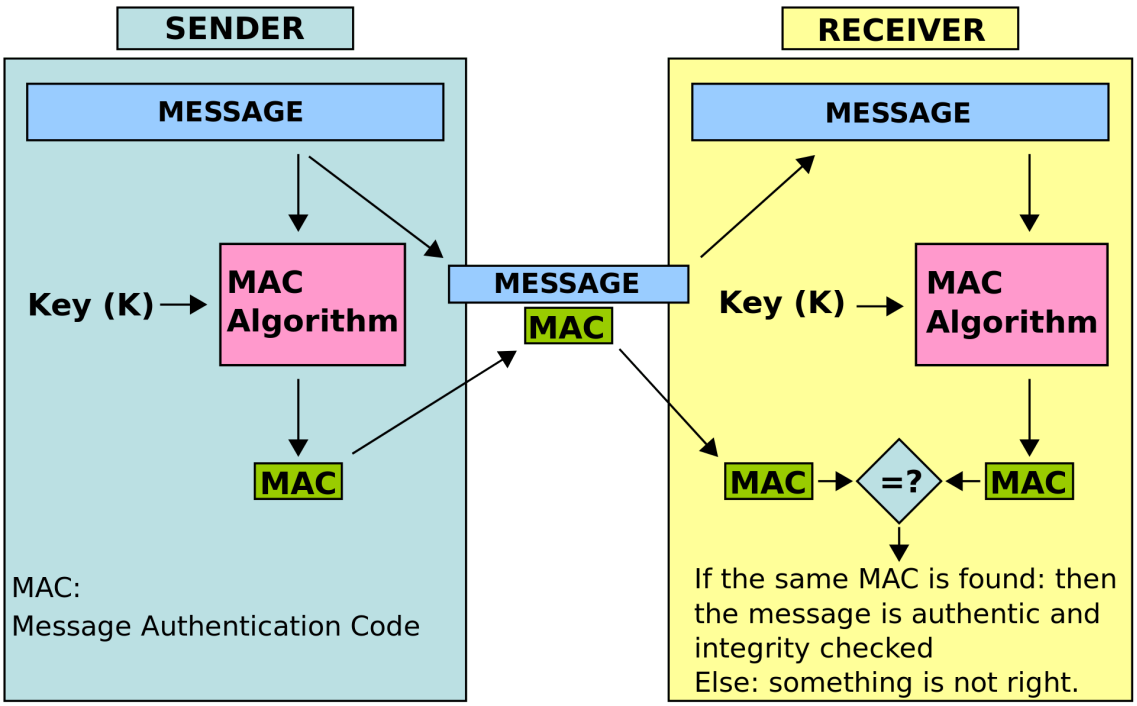
Alternativně lze využít pro konstrukci hashovacích funkcí blokové šifry. Avšak blokové šifry byly navrhovány pro jiný režim činnosti, kterém útočník nezná klíč (a není schopen ho ovlivnit), zná pouze šifrovaný text (ten je schopen ovlivnit). V tomto případě útočník může přimo ovlivňovat hodnoty klíče.




Obrázek 52.3: Ukázka několika možných způsobů využití blokových šifer pro konstrukci kompresní funkce. S využitím iteračního způsobu lze zobecnit pro celou hashovací funkci.

## 52.5 MAC (*message authentication code*)

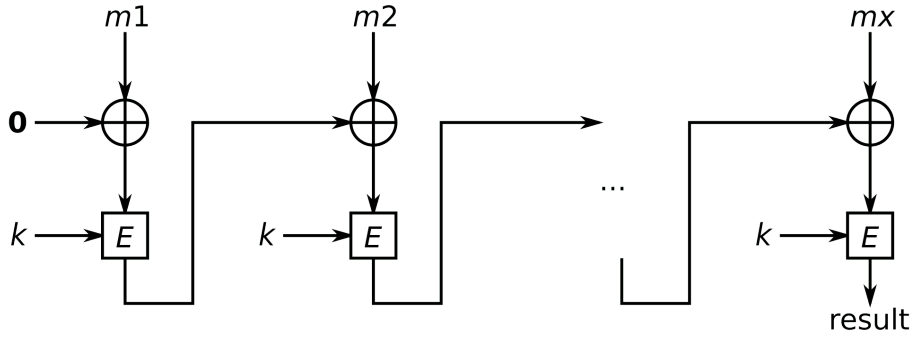
- Rodina hashovacích funkcí $h_k$, které jsou parametrizovatelné klíčem $k$.
- Vlasnosti (stejné jako u obecných hashovacích funkcí, pouze rozšířené o klíč):
  - Výstup $h_k(x)$ lze spočítat rychle, pokud je znám klíč $k$.
  - Jsou výpočetně bezpečné – při znalosti dvojice $(x, h_k(x))$ je výpočetně nemožné spočíst novou dvojici $(x', h_k(x'))$, pro $x \neq x'$, pokud není znám klíč.
- Využití: zajištění autentizace a integrity (nepopiratelnost zajistit nedokáže).




Obrázek 52.4: Schéma použití MAC. Pokud je stejný MAC výpočítán na straně příjemce, tak má jistotu, že zpráva nebyla po cestě změněna a že zprávu poslal skutečně odesílatel.

### 52.5.1 Sestrojení MAC pomocí blokové šifry v CBC

Pro sestrojení MAC hashovací funkce je využita symetrická bloková šifra v režimu CBC (*cipher block chaining*, šifrová zpětná vazba). Rozdíl oproti CBC šifrování spočívá v tom, že mezivýsledky se zahazují a pracuje se až s posledním blokem. Z něho se vezme určitý počet posledních bitů (podle požadované délky hashe – 32, 48, 64) a ten tvoří výsledný hash (MAC).




Obrázek 52.5: Ukázka sestrojení MAC funkce pomocí symetrické blokové šifry v režimu CBC.

Proč stačí výrazně menší délka? Protože klíč. Útočník sice může vyzkoušet všechny možné hashe, ale bez znalostí klíče, nezjistí, který je ten správný.

### 52.5.2 Sestrojení MAC pomocí neklíčované hashovací funkce

Pro sestrojení MAC hashovací funkce je využita neklíčovaná hashovací funkce. Klíč je připojen ke zprávě a je použita standardní hashovací funkce.

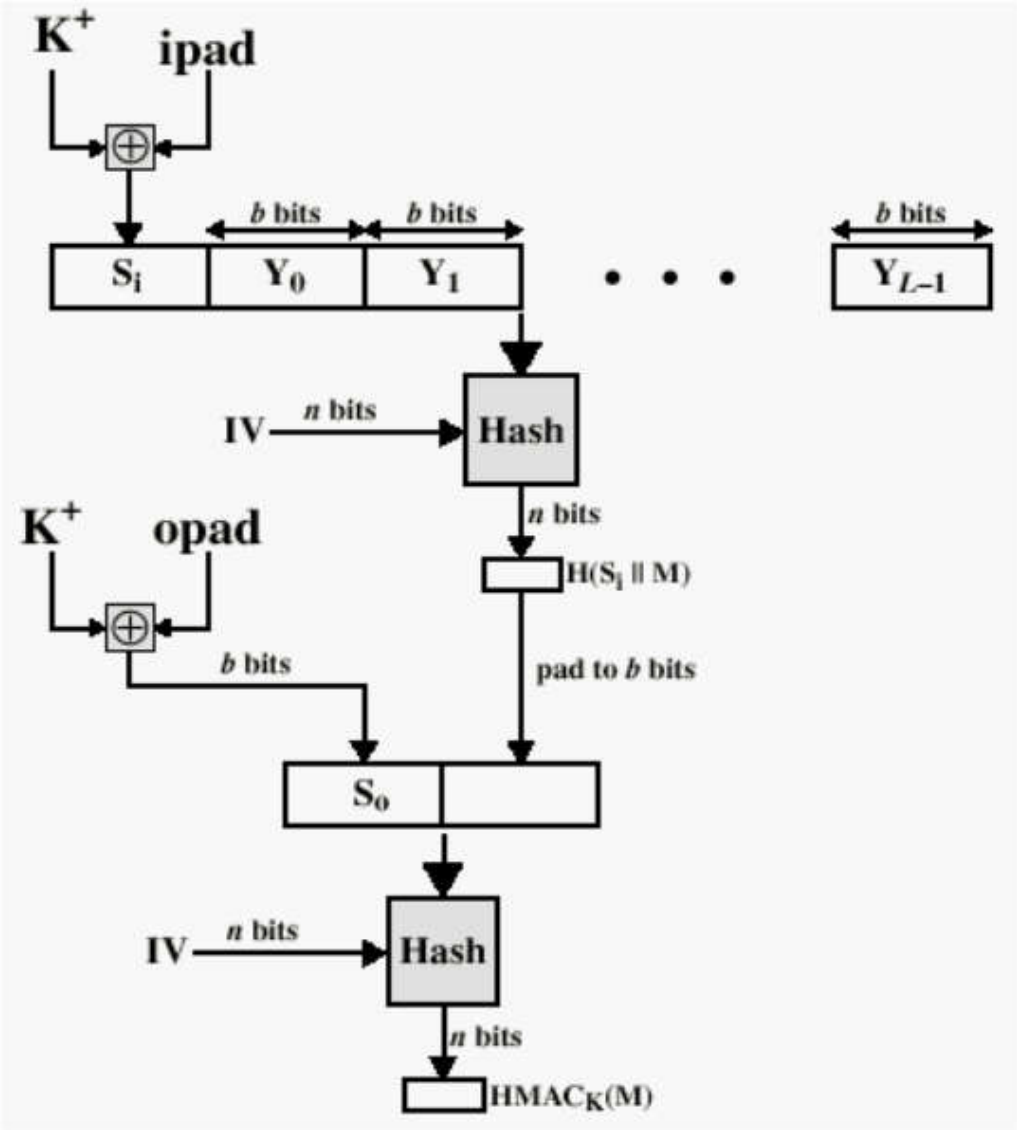
**Secret prefix** Klíč je přidán na začátek zprávy. Formálně: $H = h(k||x)$, kde $H$ je výsledný hash (MAC), $h$ je hashovací funkce, $k$ je klíč a $x$ je zpráva. Útočník může libovolně „přihashovávat“ další bloky bez znalosti klíče a tím vytvářet nové validní hashe – $h(k||x||y)$, kde $y$ je útočníkova zpráva $\Rightarrow$ nepřijatelný způsob.

**Secret suffix** Klíč je přidán na konec zprávy. Formálně: $H = h(x||k)$. Útočník, který může zvolit $x$, může také vytvořit $x'$, pro které $h(x) = h(x')$ se složitostí $O(2^{n/2})$, kde $n$ je délka hashe, bez ohledu na délku klíče $k$ (narozeninový útok) $\Rightarrow$ nepřijatelný způsob (útok který je neovlivnitelný délkou klíče).

**Enveloping** Klíč je přidán na začátek i na konec zprávy. Formálně: $H = h(k||p||x||k)$, kde $p$ je zarovnání. Přijatelný způsob. Základ pro algoritmus HMAC.

### 52.5.3 HMAC (*hash function MAC*)

HMAC (*hash function MAC*) je do dnes používaný algoritmus. Specifikuje použití metody *enveloping*, ale ne, která hashovací funkce se použije.




Obrázek 52.6: Schéma HMAC; `ipad` a `opad` jsou vstupní/výstupní konstanty, které slouží k zarovnání; $Y_i$ jsou bloky vstupní zprávy; $IV$ je inicializační vektor.

# Kapitola 53

# KRY – Správa klíčů v asymetrické kryptografii (certifikáty X.509).

## 53.1 Zdroje

- `KRY05_AsymMgmt_MNG.pdf`
- `KRY_2021-03-29.mp4`

## 53.2 Úvod a kontext

**Problém se zveřejňováním veřejných klíčů** Jak můžu vědět, že publikovaný veřejný klíč patří opravdu entitě, které patřit má? Je potřeba zajistit autenticitu (pravost) veřejných klíčů – Vytvořit spolehlivou vazbu mezi veřejným klíčem a jménem jeho vlastníka.

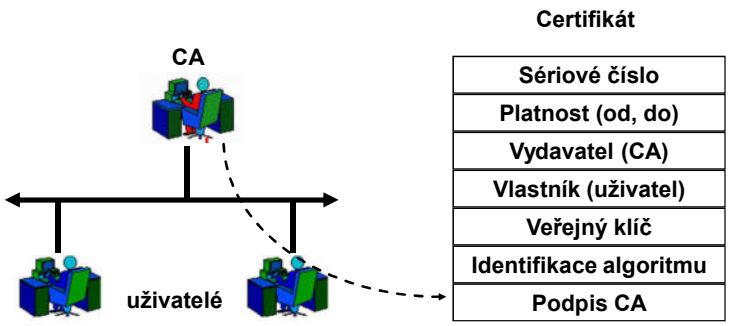
**Systémy založené na veřejném klíči** Systémy založené na veřejném klíči (PKI, *Public Key Infrastructure*) je označení infrastruktury správy a distribuce veřejných klíčů. PKI umožňuje pomocí přenosu důvěry používat cizí veřejné klíče a ověřovat jimi elektronické podpisy bez nutnosti jejich individuální kontroly.

## 53.3 Správa klíčů v asymetrické kryptografii

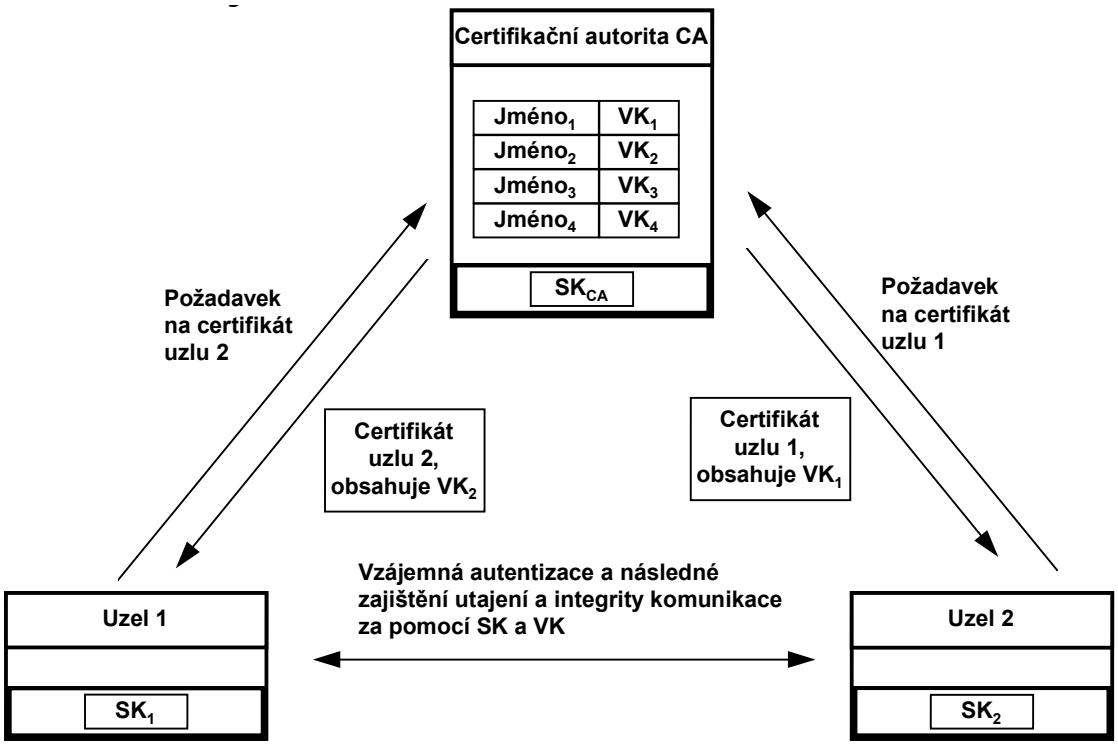
**Certifikát** Certifikace veřejného klíče. Nějaký prostředník (certifikační autorita), kterému důvěřujeme, se zaručuje, že konkrétní veřejný klíč, patří dané entitě.

**Certifikační autorita** Certifikační autorita (CA) je prostředník, který distribuuje certifikáty a které všichni důvěřují. CA negeneruje klíče uživatelům, ty si je generují samy.

**Proces certifikace klíče** CA podepíše veřejný klíč uživatele a jeho další údaje (jméno, doba vydání, doba platnosti, ...) svým soukromým klíčem. Tyto podepsané údaje se nazývají certifikát.




Obrázek 53.1: Příklad certifikátu.




Obrázek 53.2: Příklad navázání bezpečné komunikace mezi dvěma entitami, které mají stejnou certifikační autoritu.

**Navázání bezpečné komunikace** Popis navázání bezpečné komunikace (viz obrázek 53.2):

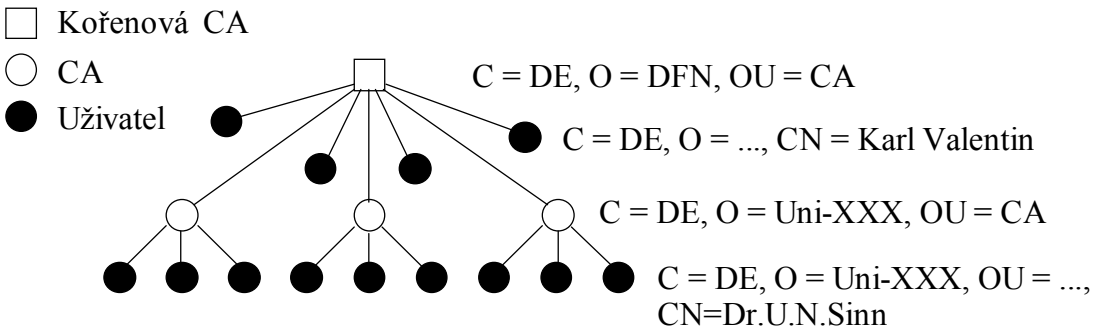
1. Uzel 1 si vygeneruje soukromý a veřejný klíč.
2. Uzel 1 odešle veřejný klíč certifikační autoritě spolu se svým jménem (a dalšíma informacema).
3. CA vytvoří certifikát pro uzel 1 – svým soukromým klíčem podepíše veřejný klíč a jméno uzlu 1. CA odešle certifikát uzlu 1. CA odešle svůj veřejný klíč uzlu 1.
4. Pokud uzel 2 chce také odesílat, provede také kroky 1-3.
5. Uzel 1 podepíše soubor a odešle ho uzlu 2 (soubor a podpis).
6. Uzel 2 si musí sehnat certifikát uzlu 1. Existují 3 způsoby jak to udělat.
   - Odesílatel zašle svůj certifikát společně se zprávou.

- Příjemce si vyžádá certifikát odesílatele od certifikační autority.
- Příjemce si vyžádá certifikát odesílatele od jiné služby (adresářové služby, LDAP).

7. Uzel 2 ověří podpis u certifikátu uzlu 1 veřejným klíčem certifikační autority.
8. Uzel 2 ověří podpis souboru pomocí veřejného klíče odesílatele (který je v certifikátu).

**Strom certifikačních autorit** Model s jednou globální CA je nemožný (příliš mnoho uživatelů, příliš velké vzdálosti, ...). Proto se používá strom certifikačních autorit. Veřejný klíč CA je certifikován jinou CA. CA nejvýše ve stromu se nazývá **kořenová certifikační autorita**.

- Certifikační autorita má svůj vlastní certifikát, který je podepsaný její certifikační autoritou.
- Koncový uživatel důvěřuje stále pouze jedné entitě – kořenové certifikační autoritě, ale přibývá jedna úroveň ověřování navíc.
- Příjemce dostane zprávu s podpisem. Musí znát certifikát odesílatele (podepsaný $CA$), certifikát certifikační autority (podepsaný $CA_{root}$) a veřejný klíč kořenové CA[1].
- Úrovní certifikačních autorit může být více (nejčastěji 1-2).




Obrázek 53.3: Příklad stromu certifikačních autorit. C, O, OU je identifikátor entity.

**Certifikační cesta** Posloupnost certifikátů od certifikátu kořenové CA přes certifikáty dalších CA až k certifikátu komunikující protistrany.

**Zneplatnění certifikátu** Jak zrušit platnost certifikátu? Normálně se zruší sám, až skončí jeho platnost. Pokud je potřeba certifikát zneplatnit před jeho vypršením je třeba využít tzv. revokační seznam (CRL, *certificate revocation list*). Důvody zneplatnění certifikátu:

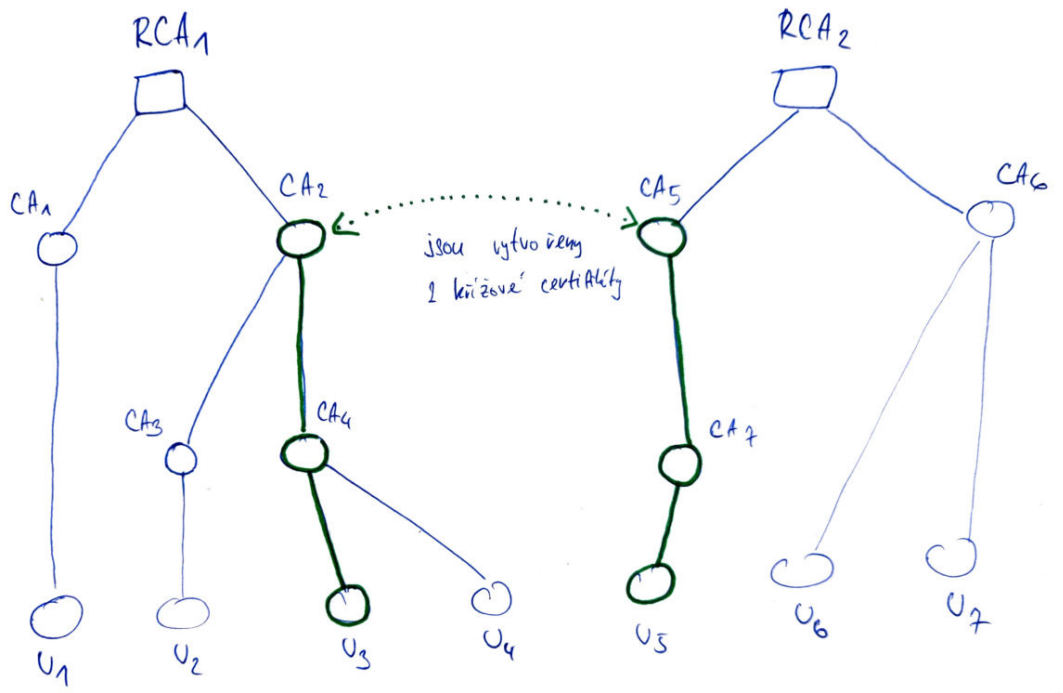
- soukromý klíč uživatele byl kompromitován,
- uživatel ztratil práva, která z certifikátu vyplývají (např. změna zaměstnavatele),
- soukromý klíč CA byl kompromitován (nikdy se nestalo).

**CRL** CRL (*certificate revocation list*) je seznam zneplatněných certifikátů, takových, kterým ještě nevypršela platnost, ale je třeba je zneplatnit. CRL je podepsán CA, které ho spravuje a periodicky aktualizuje (může se zkracovat i růst). Jak se distribuuje:

[1] Veřejný klíč kořenové certifikační autority se z praktických distribuuje ve formě „fiktivního certifikátu“

- *Pull model* – Příjemce certifikátu si dle potřeby stáhne CRL od CA.
- *Push model* – CA pravidelně posílá CRL příjemcům certifikátu.

**Křížový certifikát** V případě, že spolu komunikují uživatelé, kteří nemají společnou kořenovou CA (jejich CA si nedůvěřují), je třeba využít tzv. křížový certifikát. Tím se komplikuje sestavování certifikační cesty, protože je třeba zvážit všechny křížové certifikáty, které jsou k dispozici.




Obrázek 53.4: Příklad křížového certifikátu. Uživatel $U_3$ chce navázat bezpečné spojení s uživatelem $U_5$. Nemají společnou kořenovou CA, proto je třeba využít křížové certifikáty. CA2 vytvoří křížový certifikát pro $CA_5$ a $CA_5$ vytvoří křížový certifikát pro $CA_2$. Příklad: $U_3$ pošle podepsanou zprávu $U_5$, jak bude vypadat certifikační cesta? $U_3 \leftarrow CA_4 \leftarrow CA_2 \leftarrow CA_5 \leftarrow RCA_2$.

## 53.4 Standard X.509

X.509 je standard pro systémy založené na veřejném klíči (PKI). Specifikuje formát certifikátů, formát CRL, parametry certifikátů, metody kontroly platností certifikátů, ...

```
Certificate ::= SIGNED SEQUENCE {
    version [0] Version DEFAULT v1988,
    serialNumber CertificateSerialNumber,
    signature
    AlgorithmIdentifier,
    issuer
    Name,
    validity
    Validity,
    subject
    Name,
    subjectPublicKeyInfo SubjectPublicKeyInfo
}

Version ::= INTEGER {v1988(0) }

CertificateSerialNumber ::= INTEGER

Validity ::= SEQUENCE {notBefore UTCTime, notAfter UTCTime }

SubjectPublicKeyInfo ::= SEQUENCE {
    algorithm
    AlgorithmIdentifier,
    subjectPublicKey
    BIT STRING
}

AlgorithmIdentifier ::= SEQUENCE {
    algorithm
    OBJECT IDENTIFIER,
    parameters
    ANY DEFINED BY algorithm OPTIONAL
}
```

Výpis 53.1: Příklad definice certifikátu ve formátu X.509.

**Význam položek** Význam položek v definici certifikátu ve formátu X.509:

- Version – Standardně 0.
- Serial number – Sériové číslo certifikátu, spolu se jménem vydavatele jednoznačně identifikuje certifikát.
- Issuer – Jméno vydávající CA.
- Subject – Jméno vlastníka certifikátu.
- Validity – Doba platnosti certifikátu (`notBefore`, `notAfter`). Podpis je platný pouze pokud je datum podepsání v intervalu platnosti každého z certifikátů z certifikační cesty.
- SubjectPublicKeyInfo – Veřejný klíč vlastníka certifikátu a algoritmus, pro který je určen.
- Signature – Jakým algoritmem je certifikát podepsaný CA.

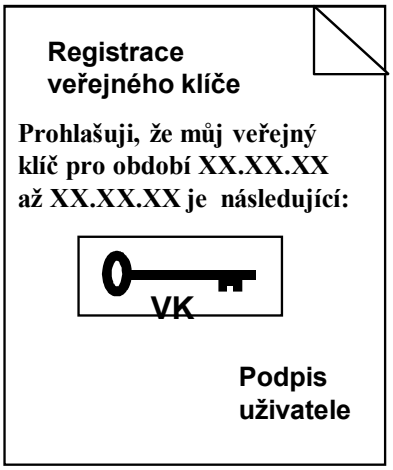
**Prototypový certifikát** Má strukturu certifikátu X.509. Uživatel si vygeneruje tzv. prototypový certifikát, který má standardní strukturu a vyplní informace na nějaké implicitní

hodnoty. Prototyp pošle CA spolu se svým veřejným klíčem, která certifikát dovyplní, podepíše a pošle zpět.

**Registrační autorita** Pokud chce uživatel vydat certifikát, kontaktuje tzv. registrační autoritu (nikoliv přímo CA).

**Míra důvery v certifikát** V praxi chceme více urovní důvěry, než pouze ostrý/žádný (např. chceme vytvořit testovací certifikát). To je řešeno jako rozšíření X.509 přidáním třídy certifikátu (*certification class*). Uživatel chce vydat certifikát od CA jisté třídy.

- Třída 1 – CA vůbec nekontroluje identitu žadatele. Lze jej získat anonymně. Používá se pro testovací certifikáty.
- Třída 2 – Identita žadatele musí být ověřena třetí stranou (notářsky ověřený formulář zaslaný poštou).
- Třída 3 – Standardní certifikát. Žadatel musí osobně navštívit CA (resp. registrační autoritu). Osobní ověření totožnosti.
- Třída 4 – Stejné jako 3 a navíc je nutné prokázat oprávnění žadatele požadovat certifikát.




Obrázek 53.5: Příklad žádosti o certifikát.

# Kapitola 54

# PDS – Prerekvizity k ostatním otázkám.

**ISO/OSI model** Referenční model ISO/OSI se používá jako názorný příklad řešení komunikace v počítačových sítích pomocí vrstevnatého modelu, kde jsou jednotlivé vrstvy nezávislé a snadno nahraditelné.

- **Aplikační vrstva** (L7, *application layer*)
  - Zajišťuje zpracování dat na nejvyšší úrovni (reprezentace dat, kódování, řízení dialogu, ...).
  - Tvořena procesy a aplikacemi, které komunikují po síti.
  - Bývá slučována s prezentační vrstvou (L6, prezentace dat a šifrování) a relační vrstvou (L5, koordinace a komunikace).
  - Příklad protokolů:
    * Uživatelské – vykonávají služby přímo uživateli (Telnet, SSH, FTP, SMTP, HTTP, ...)
    * Systémové – zajišťují síťové funkce (DNS, DHCP, SNMP, BOOTP, ...)
- **Transportní vrstva** (L4, *transport layer*)
  - Rozděluje aplikační data (segmentace) na menší jednotky a zapouzdřuje je do segmentů (TCP) / datagramů (UDP).
  - Vytváří logické spojení mezi procesy (přenáší data konkrétní aplikace ze zdrojového zařízení do aplikace na cílovém zařízení).
  - Adresace: porty.
  - Příklad protokolů: TCP, UDP, DCCP, SCTP, MP-TCP, QUIC
- **Síťová vrstva** (L3, *network layer*)
  - Zapouzdřuje segmenty/datagramy do paketů.
  - Řeší směrování.
  - Adresace: IP adresa (logická adresa).
  - Příklad protokolů: IPv4, IPv6, ARP, RARP, ICMP, IGMP
- **Linková vrstva** (L2, *data link layer*, vrstva síťového rozhraní, *network interface layer*)
  - Zapouzdřuje pakety do rámců.

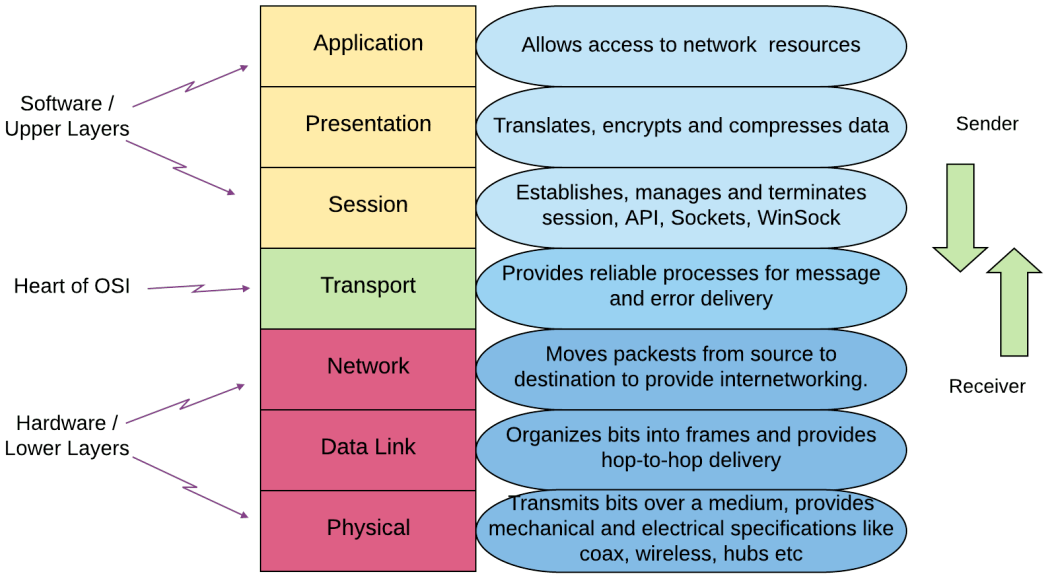
  - Zajišťuje *hop-by-hop* doručení.
  - Adresace: MAC adresa (fyzická adresace).
  - Příklad protokolů: Ethernet, Token Ring, FDDI, X.25, Frame Relay
- **Fyzická vrstva** (L1, *physical layer*)
  - Zajišťuje přenos bitů přes fyzické médium.

**Adresace**

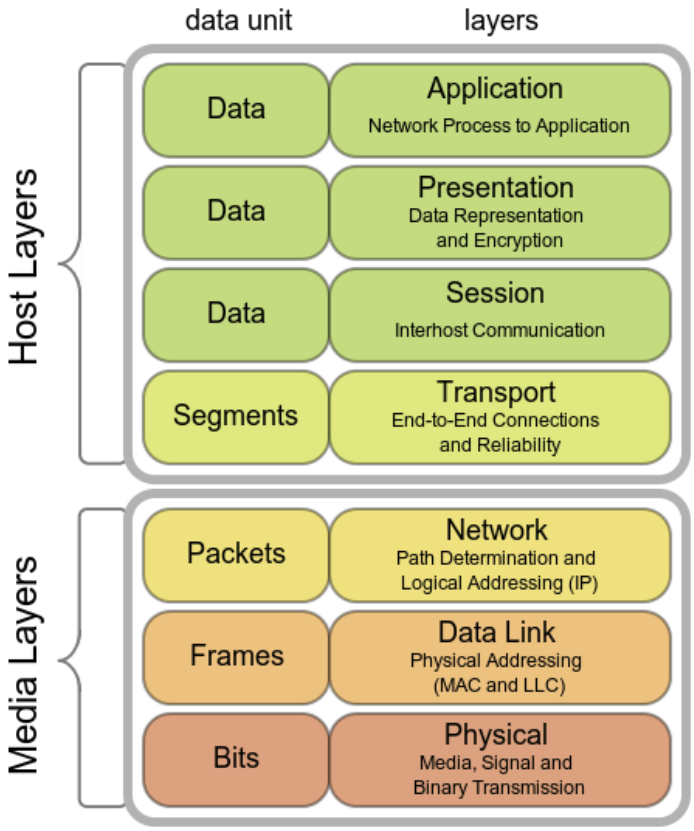
- Port (transportní vrstva, L4)
  - Identifikuje aplikaci v rámci zařízení.
  - Jak se mění při směrování paketu internetem: zůstává stejný s výjimkou zdrojového portu při překladu NAT.
  - Velikost: 16 bit
  - Prostor: plochý
- IPv4, IPv6 (síťová vrstva, L3)
  - Identifikuje uzel v rámci sítě (tzv. logická adresace).
  - Jak se mění při směrování paketu internetem: zůstává stejná s výjimkou zdrojové IP adresy při překladu NAT.
  - Velikost: 32 bit, 128 bit
  - Prostor: pseudohierarchie (A, B, C, D, E), pseudohierarchie (prefix + interface ID)
- MAC (linková vrstva, L2)
  - Identifikuje síťové rozhraní (síťovou kartu).
  - Jak se mění při směrování paketu internetem: mění se *hop-by-hop*.
  - Velikost: 32 bit, 128 bit
  - Prostor: pseudohierarchie (A, B, C, D, E), pseudohierarchie (prefix + interface ID)

**data, segment, datagram, paket, rámec, bit**

- Data – aplikační vrstva (L7)
- Segment – transportní vrstva (L4), TCP
- Datagram – transportní vrstva (L4), UDP
- Paket – síťová vrstva (L3)
- Rámec – linková vrstva (L2)
- Bit – fyzická vrstva (L1)




Obrázek 54.1: Příklad OSI modelu z Linuxhint.




Obrázek 54.2: Příklad OSI modelu z Wiki.

**ACL** ACL (*Access Control List*) je volitelná vrstva zabezpečení, která funguje jako brána firewall pro řízení provozu do jedné nebo více podsítí a z nich.

**NAT** NAT (*Network Address Translation*) je metoda mapování IP adresního prostoru do jiného prostoru (typicky privátní adresy na veřejné adresy). Děje se tak úpravou hlaviček IP paketů během jejich přenosu přes směrovače (úprava zdrojové IP adresy a čísla portu). Směrovač si ukládá čtveřice (WAN_IP : WAN_port, LAN_IP : LAN_port) aby mohl provádět i překlad zpět.

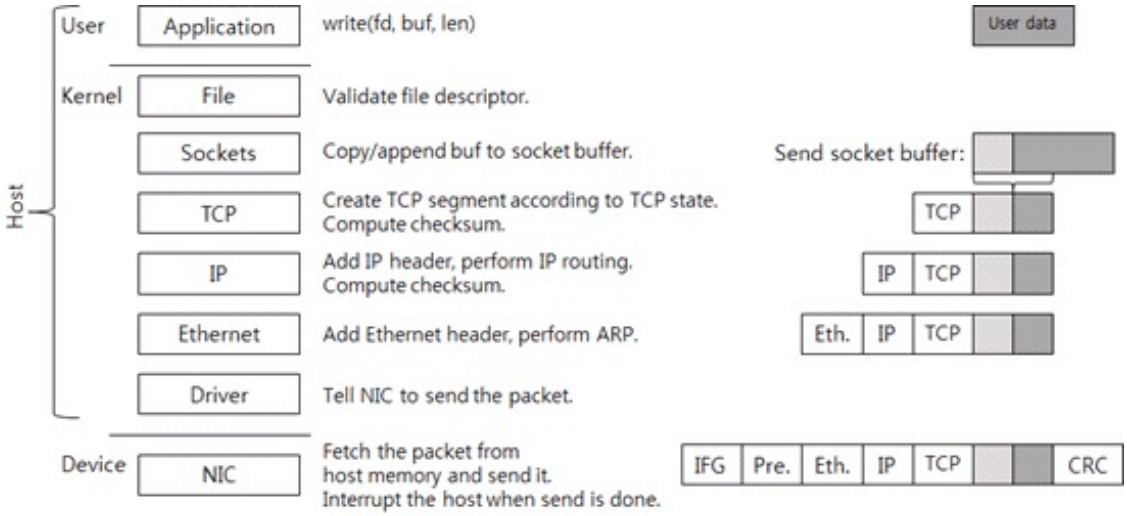
**ARP** ARP (*Address Resolution Protocol*) a RARP (*Reverse ARP*) je protokol, který komunikuje na síťové vrstvě (L3) a zajišťuje „překlad“ IP adres na MAC adresy a obráceně. Pouze pro IPv4, pro IPv6 je pro stejný účel využíván protokol ICMPv6 a zpráva *Neighbor Discovery*. Příklad využití: směrovač potřebuje získat MAC adresu next hopu (zná jeho IP adresu).

**ICMP** ICMP (*Internet Control Message Protocol*) je protokol, který komunikuje na síťové vrstvě (L3) a slouží pro řízení toku a detekce nedosažitelných uzlů.

**MAC adresa** MAC adresa je fyzická adresa zařízení, resp. síťové karty (identifikátor na L2). Na každý síťový port v přepínači/směrovači je jedna síťová karta (a teda i MAC adresa).

**IP adresa** IP adresa je logická adresa zařízení (identifikátor na L3). Pokud má zařízení více síťových karet, má typicky stále pouze jednu IP adresu. Přepínač nemá IP adresu vůbec, pracuje pouze na L2 vrstvě. Směrovač je výjimka a má 2 IP adresy, jednu pro komunikaci v lokální síti (LAN) a druhou pro internet (WAN).

**Síťový tok** Síťový tok je posloupnost paketů (jednosměrná) identifikována čtveřicí (zdrojová IP, zdrojový port, cílová IP, cílový port).




Obrázek 54.3: Operace na jednotlivých vrstvách ISO modelu.

# Kapitola 55

# PDS – Základní architektury přepínačů, algoritmy pro plánování, řešení blokování, vícestupňové přepínací sítě.

## 55.1 Zdroje

- `04-switching.pdf`
- `PDS_2021-03-05.mp4`

## 55.2 Úvod a kontext

**CAM tabulka** CAM tabulka je datová struktura v přepínači, která uchovává informace o tom, za jakým fyzickým portem je zařízení s jakou MAC adresou. Jak se plní? Přepínač si ji plní automaticky. V momentě kdy k němu přijde paket, doplní si MAC adresu a port přes který paket přišel. Pokud přepínač neví za jakým portem je cílová MAC adresa, pošle to všem (*flooding*).

**Propustnost** Kolik je možné přenést dat za časovou jednotku.

## 55.3 Obecná architektura přepínače

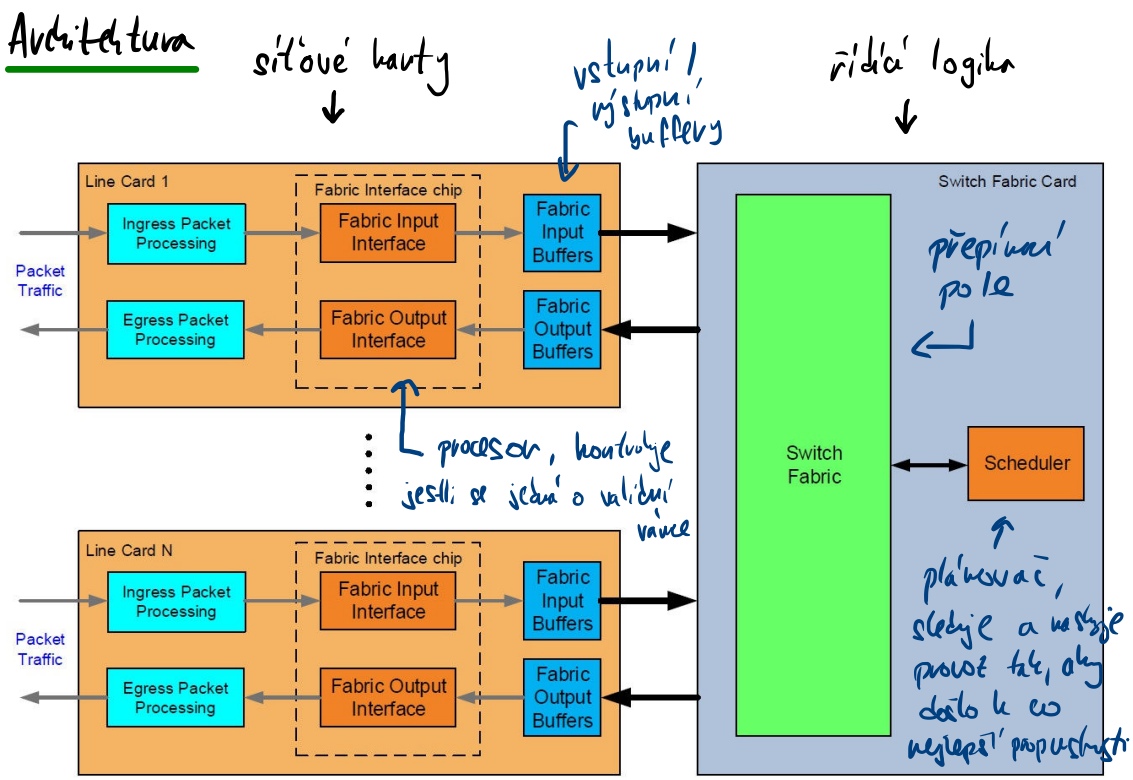
Síťový přepínač (*switch*) je aktivní prvek v počítačové síti, který propojuje jednotlivé prvky do hvězdicové topologie. Přepínač obsahuje síťové porty (až stovky), na něž se připojují síťová zařízení.

- Na jaké vrstvě OSI modelu pracuje? Pracuje s rámci (linková vrstva, L2).
- Na základě čeho provádí přepínání? Na základě cílové MAC adresy.
- Jaké typy přenosů přepíná? Na základě cílové MAC adresy rozslišuje typ přenosu:
  - broadcast (samé 1),
  - multicast (speciální prefix pro IPv4 a IPv6),
  - unicast (cokoliv jiného).

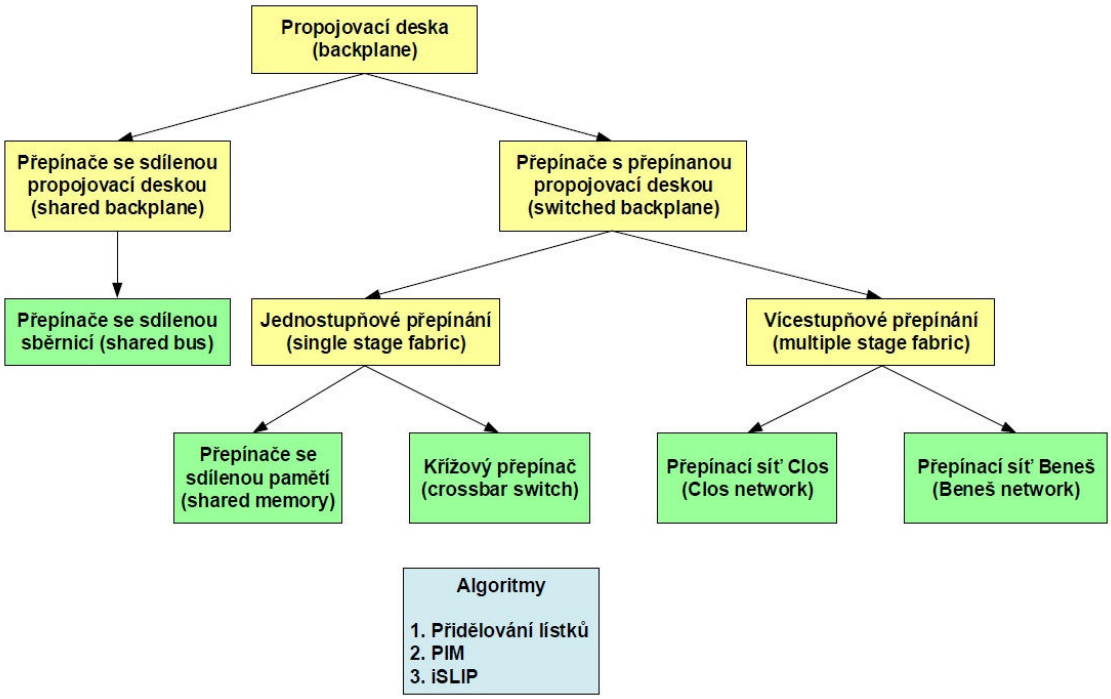
- Co ovlivňuje rychlost přepínání?
  - Hardware (typ paměti, rychlost procesoru)
  - Logika přepínání
  - Šířka pásma cílového rozhraní

**Požadavky na přenos**

- Maximální využití sběrnice (požadavek na plánovač).
  - Aby došlo k maximálnímu přenosu dat přepínací logikou (co nejvíce přenosů v rámci taktu – „paralelizace“).
- Spravedlivé přidělování přenosového pásma (aby byly obsluhovány všechny porty).
- Zachování pořadí rámců (ne vždy je možné).




Obrázek 55.1: Obecná architektura přepínače.




Obrázek 55.2: Dělení přepínačů.

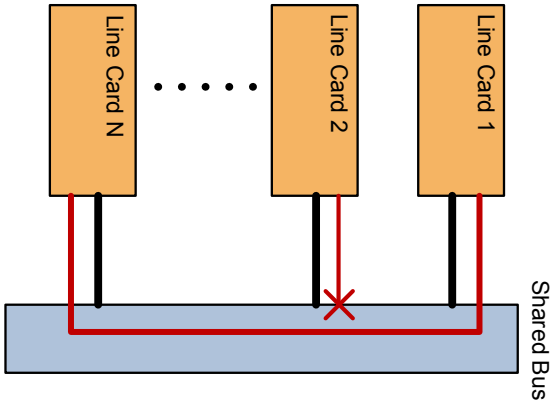
## 55.4 Přepínače se sdílenou propojovací deskou

### 55.4.1 Přepínače se sdílenou sběrnicí

- Může komunikovat pouze jeden port v daný čas, ostatní čekají (řízeno protokolem).
- Lze jednoduše realizovat broadcast a multicast.
- Mějme $N$ portů (karet), rychlost každé karty $R$ bps, taktovací frekvenci $r$. Pak rychlost sběrnice musí být $N \times R$ a šířka sběrnice

$$W = \frac{R \times N}{r}$$

- Pokud se přidají nové porty nebo se na nich zvýší rychlost je potřeba zvýšit propustnost sběrnice („něco za něco“).




Obrázek 55.3: Přepínač se sdílenou sběrnicí.

Úloha č.1:

- Přepínač má 16 portů o rychlosti 100 Mb/s. Jaká je potřebná propustnost sběrnice a jaká musí být šířka sběrnice, aby zvládl požadovaný přenos? Uvažujte taktovací frekvenci sběrnice 40 MHz.

$r = 40$ MHz $\quad N = 16 \quad R = 100$ Mb/s

propustnost – kolik dat za časovou jednotku

$\hookrightarrow N \times R = 1{,}6$ Gb/s

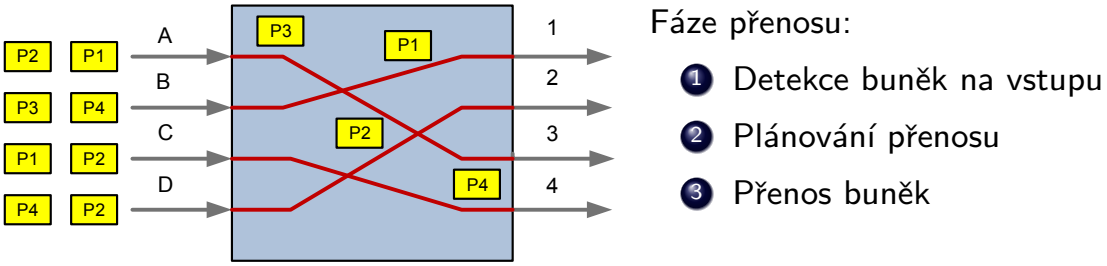
šířka sběrnice – kolik v jeden okamžik maximálně

$\hookrightarrow \frac{N \times R}{r} = \frac{16 \cdot 100 \cdot 10^6}{40 \cdot 10^6} = 40$ b

Obrázek 55.4: Přepínač se sdílenou sběrnicí – příklad.

## 55.5 Přepínače s přepínanou propojovací deskou: jednostupňové přepínání

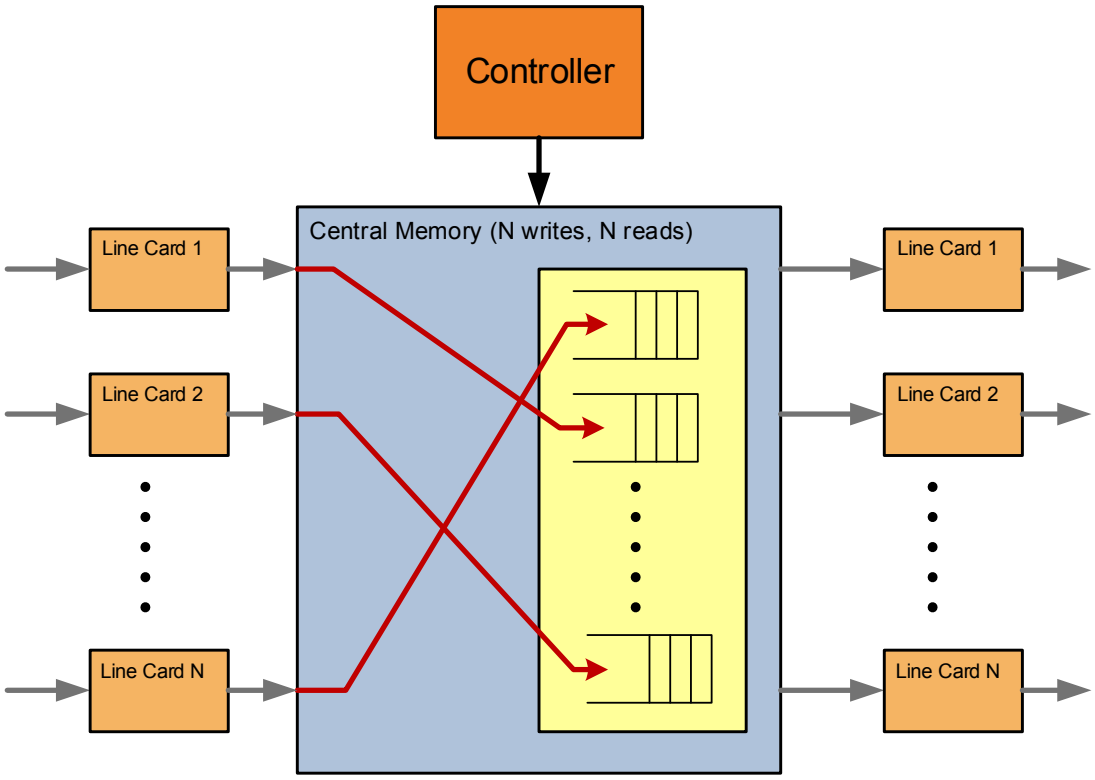
- Umožňuje paralelní přenos rámců.
- Je potřeba plánovač (*scheduler*), který plánuje přenos.
- Provedení přenosu: z bufferů vstupních karet se data přenesou do bufferů výstupních karet.




Obrázek 55.5: Činnost přepínače s přepínanou propojovací deskou.

### 55.5.1 Přepínače se sdílenou pamětí

- Centrální sdílená paměť, která obsahuje frontu pro každý výstupní buffer (každý port).
- Pro jeden přenos se musí 2x přistoupit do paměti (zápis a čtení).
- Zde narážíme na současné technologické limity pamětí (rychlost přístupu).




Obrázek 55.6: Činnost přepínače s přepínanou propojovací deskou a se sdílenou pamětí. Obsahuje: $N$ vstupních karet, řadič, sdílenou paměť s $N$ frontami, $N$ výstupních karet.

- Nechť $R$ je rychlost síťové kart, $N$ je počet síťových karet, pak celková propustnost přepínače je $BW = 2 \times N \times R$ (dva přístupy do paměti).
- Mějme data o velikosti $C$, pak doba přenosu je $t = C \div BW[s]$.

Úloha č.2:

- Jaká je potřebná doba přístupu do paměti u přepínače se sdílenou pamětí, který má 32 portů o rychlosti 1 Gb/s a velikost ukládané buňky je 40 bytů? Jak se tato doba změní pro rychlosti portů 10 Gb/s a 40 Gb/s?

$C = 40\ B \qquad N = 32 \qquad R = 1\ Gb/s$

$$t = \frac{40 \cdot 8}{2 \cdot 32 \cdot 1 \cdot 10^9} = 5 \cdot 10^{-9}\ s = 5\ ns$$

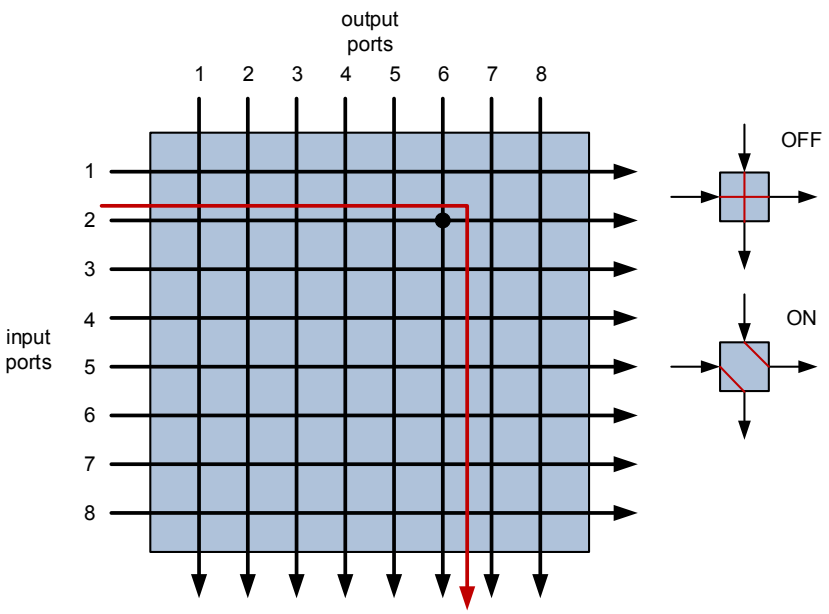
pro 10 -> 0,5 ns

pro 40 -> 0,125 ns

Obrázek 55.7: Přepínač s přepínanou sběrnicí a se sdílenou pamětí – příklad.

### 55.5.2 Křížové přepínače (*crossbar*)

- Založený na jednom velkém propojovacím poli.
- Interně neblokující.
- Nativní podpora multicastu.
- Je potřeba $N^2$ propojení.
- Chybí redundance: pouze jedna cesta mezi vstupem a výstupem.




Obrázek 55.8: Přepínač s přepínanou sběrnicí a křížovými propoji.

## 55.6 Hledání spojů v křížovém přepínači

### 55.6.1 Problém párování

V kontextu přepínání v křížovém přepínači se jedná o problém párování v bipartitním grafu, kde množina vstupních portů je $V_1$, množina výstupních portů je $V_2$ a hrany $E$ jsou propoje vstupů a výstupů. Množina $M$ se nazývá párování, pokud $M \subseteq E$, kde žádné dvě hrany z M, nemají společný vrchol.

**Bipartitní graf** Nechť $G = (V, E)$ je graf. Graf G je bipartitní, pokud platí $V = V_1 \cup V_2 \wedge V_1 \cap V_2 = \emptyset \wedge \forall e = \{u, v\}, e \in E : u \in V_1 \wedge v \in V_2$. Platí li navíc $E = V_1 \times V_2$, pak je graf úplný bipartitní.

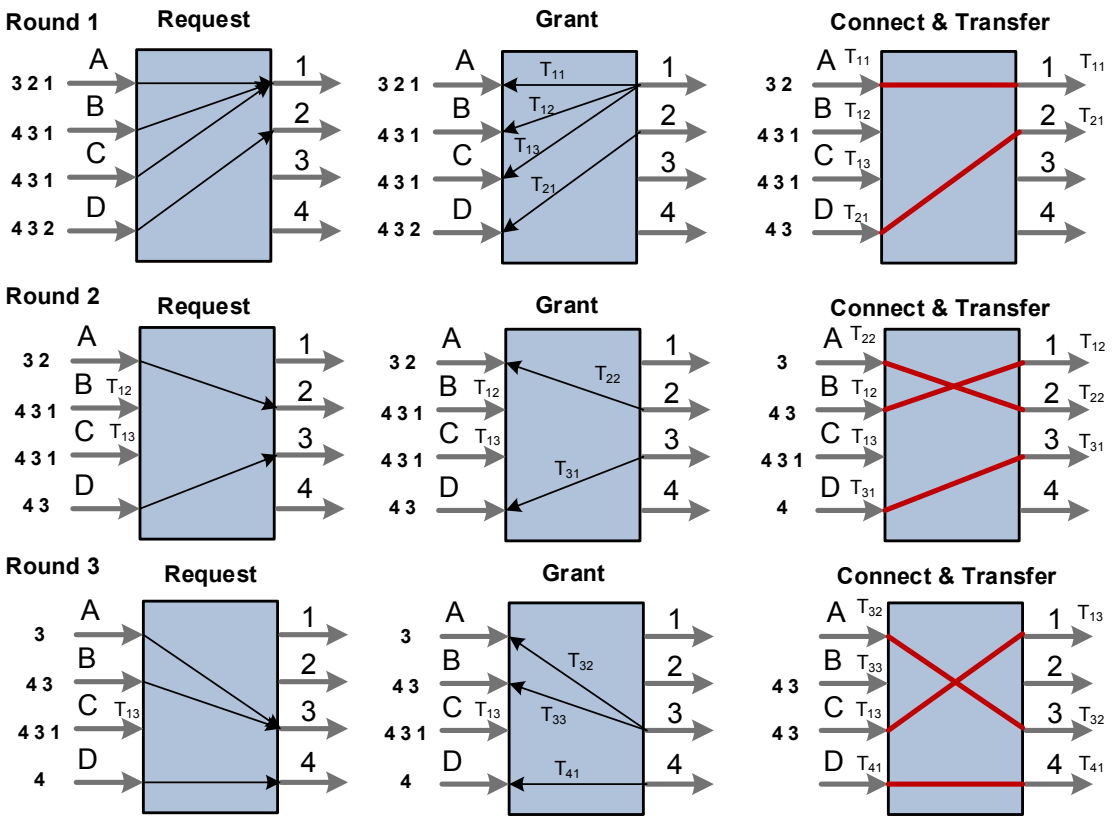
**Největší párování** Největší párování (*maximum matching*) je párování, které má největší počet hran (globální maximum). Složitost: $O(N^{5/2})$.

**Maximální párování** Maximální párování (*maximal matching*) je párování, kdy nelze přidat žádnou další hranu (lokální maximum). Složitost: $O(N + E)$.

Jelikož algoritmy pro hledání maximálního párování mají výrazně menší časovou složitost, volíme pro hledání spojů v křížovém přepínači ty.

### 55.6.2 Algoritmus přidělování lístků

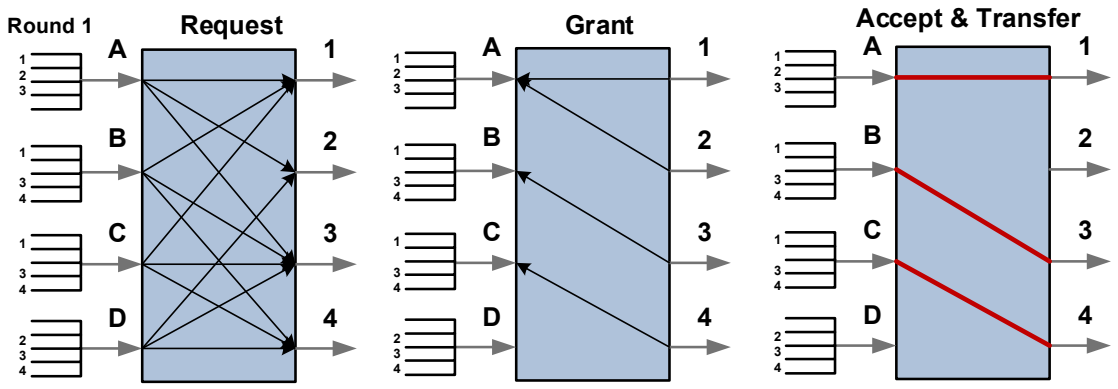
- Princip:
  - Výstupní port $Q$ obsluhuje frontu požadavků na propojení.
  - Požadavek na vstupu $P$ dostane od $Q$ číslo $T_{QX}$ pro obsloužení portu $Q$ s pořadím $X$.
- Fáze:
  1. Žádost o lístek
  2. Přidělení lístku
  3. Propojení vstupů s výstupy a přenos
- Hodnocení:
  - Umožňuje přenos rámců s proměnlivou délkou
  - **Blokování na začátku fronty** (*head of line blocking*) – Situace, kdy více vstupů chce stejný výstup. Musí se zpracovat ve více kolech. Řešení: Virtuální výstupní fronty (každý vstupní port má frontu pro každý výstupní port). Tím může dojít k předbíhání rámců. Viz další algoritmy.



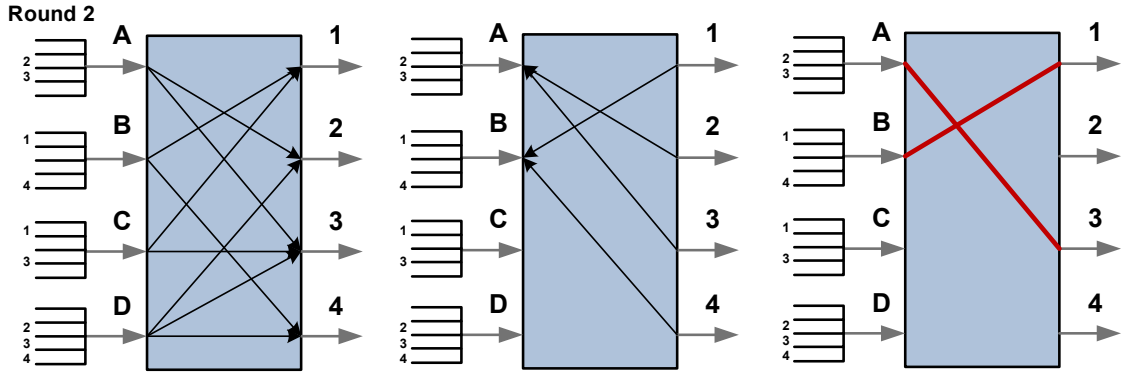

Obrázek 55.9: Příklad činnosti algoritmu přidělování lístků.

### 55.6.3 Algoritmus PIM (*Parallel Iterative Matching*)

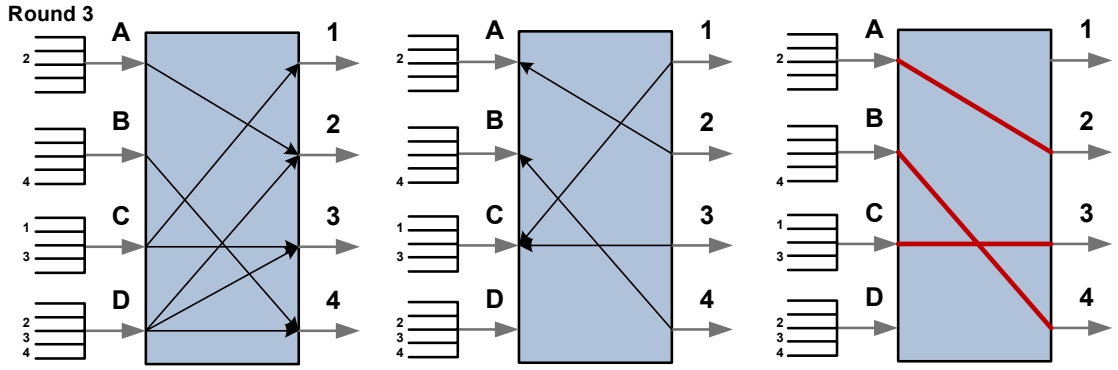
- Algoritmus přidělování lístků + virtuální výstupní fronty + náhodnost.
  - Posílají se požadavky pro všechny pakety ve frontě, nikoliv pouze pro první.
- Hledá maximální párování (pomocí náhodné volby, která zabrání vyhladovění).
- Soutěžení o porty
  - Výstupní port – více žádostí naráz, jednu vyberu náhodně.
  - Vstupní port – povolení na více výstupů naráz, jeden vyberu náhodně.




Obrázek 55.10: Příklad činnosti algoritmu PIM, kolo 1.



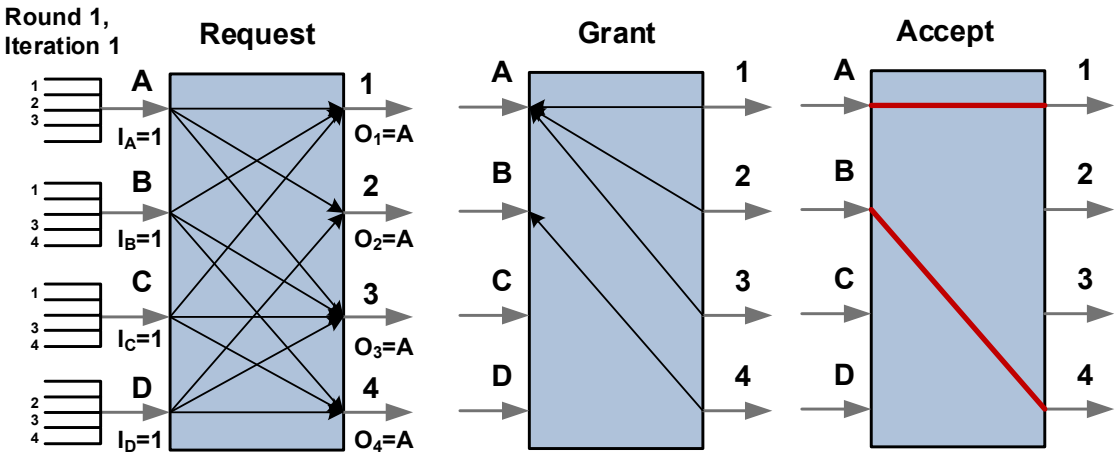

Obrázek 55.11: Příklad činnosti algoritmu PIM, kolo 2.



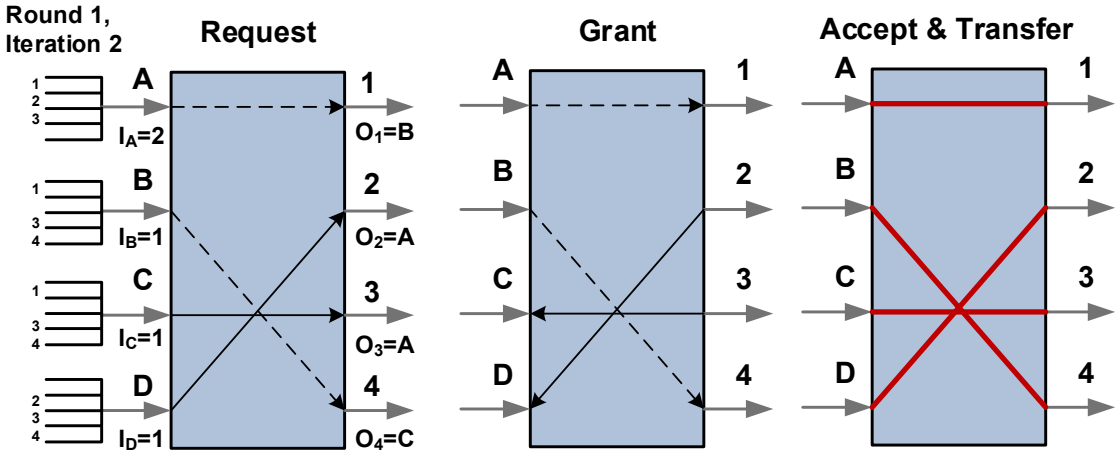

Obrázek 55.12: Příklad činnosti algoritmu PIM, kolo 3.

### 55.6.4 Algoritmus iSLIP

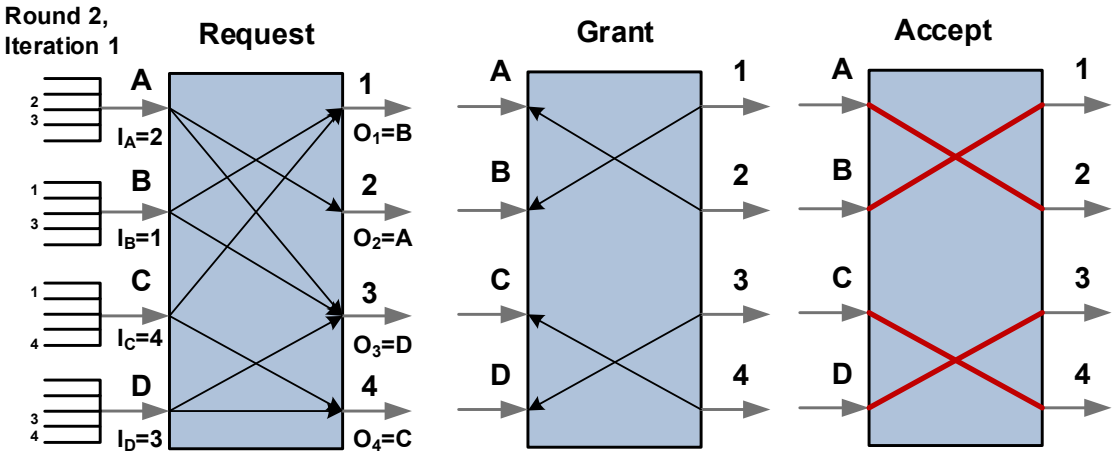
- Algoritmus přidělování lístků + virtuální výstupní fronty + iterace + ukazatele.
- Algoritmus „rotujících ukazatelů“ (vyhneme se náhodnému výběru).
  - Každý vstupní port má ukazatel (počítadlo), podle kterého se určuje priorita výstupu.
  - Každý výstupní port má ukazatel, podle kterého se určuje priorita vstupu.
  - Nová hodnota ukazatele je port na který se posílá / port od kterého se přijímá +1.
- Každé kolo má dvě iterace, resp. 6 fází – *request*, *grant*, *accept*, *request*, *grant*, *accept and transfer*.
  - Proč? Protože fáze *transfer* trvá nejdéle (optimalizace).




Obrázek 55.13: Příklad činnosti algoritmu iSLIP, kolo 1.




Obrázek 55.14: Příklad činnosti algoritmu iSLIP, kolo 2.




Obrázek 55.15: Příklad činnosti algoritmu iSLIP, kolo 3.

## 55.7 Přepínače s přepínanou propojovací deskou: vícestupňové přepínání

Vlastnosti jednostupňového křížového přepínače:

- S počtem portů roste kvadraticky velikost přepínacího pole.
- Vnitřní blokování (soupeření o porty).
- Blokování na začátku fronty (*head of line blocking*).
- Součástí přepínání je hledání maximálního párování.

Je možné zvětšit počet portů, aniž by se dramaticky rozšířila velikost přepínacího pole? Ano, pomocí vícestupňového přepínání (vstup → vnitřní přepínací obvody → výstup).

- Nemají vnitřní blokování – Nejsou potřeba plánovací algoritmy pro hledání maximálního párování.

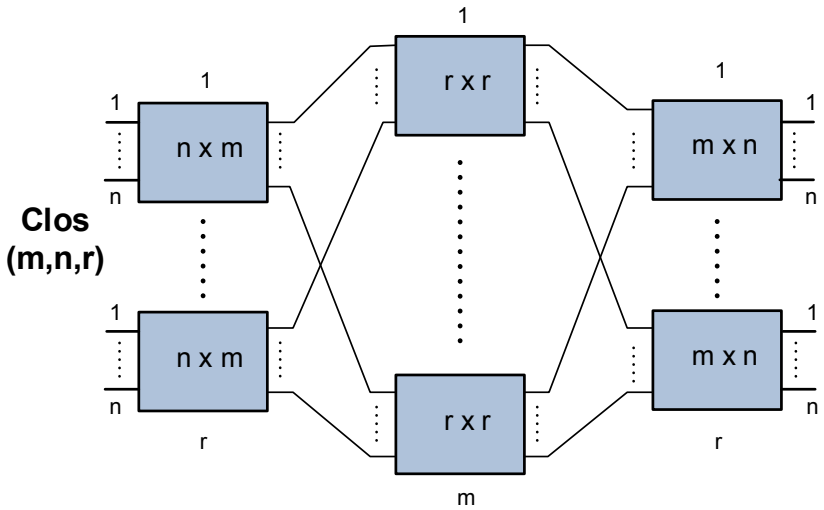
### 55.7.1 Přepínací síť Clos

$Clos(m, n, r)$

- $r$ vstupních bloků s $n$ vstupy (bloků = křížový přepínač)
- $r$ výstupních bloků s $n$ vstupy
- $m$ vnitřních bloků s $r$ vstupy

Closův teorém:

- Pokud $m \geq 2n - 1$, pak lze přidat nové propojení vstupu a výstupu bez přeskládání (síť je neblokující).
  - Pokud všechny požadavky na vstupu jdou na jiné výstupy, pak pro jakoukoliv konfiguraci vstupů a výstupů je síť neblokující (pokud nejdou dva vstupy na jeden stejný výstup).

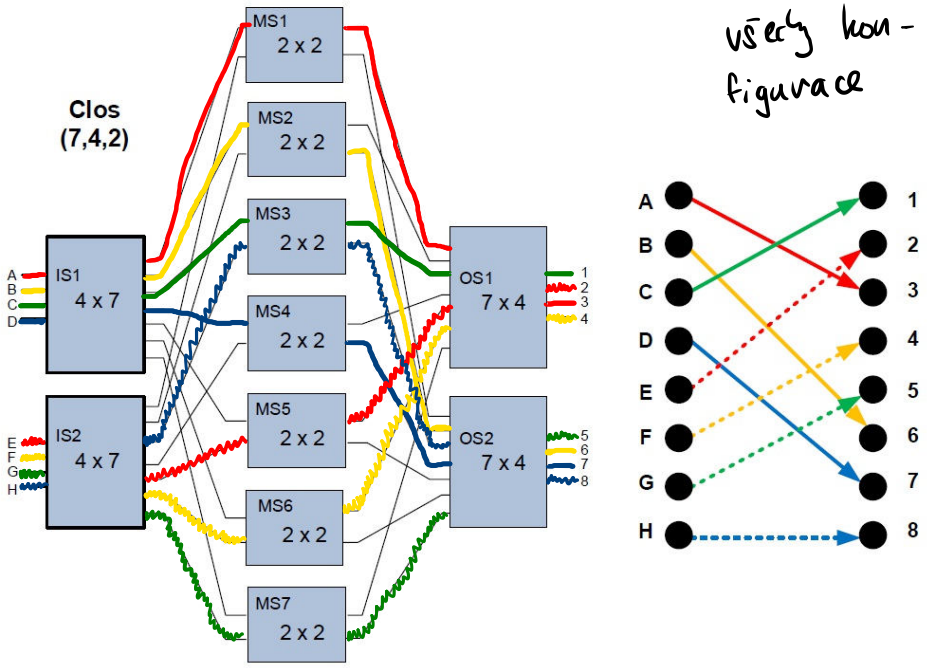



Obrázek 55.16: Obecné schéma Closovy sítě.

Příklad neblokující sítě Clos(7,4,2): osmiportový přepínač

- Platí Closův teorém, tj. $m \geq 2n - 1$

• Vždycky najde propojení přes všechny konfigurace



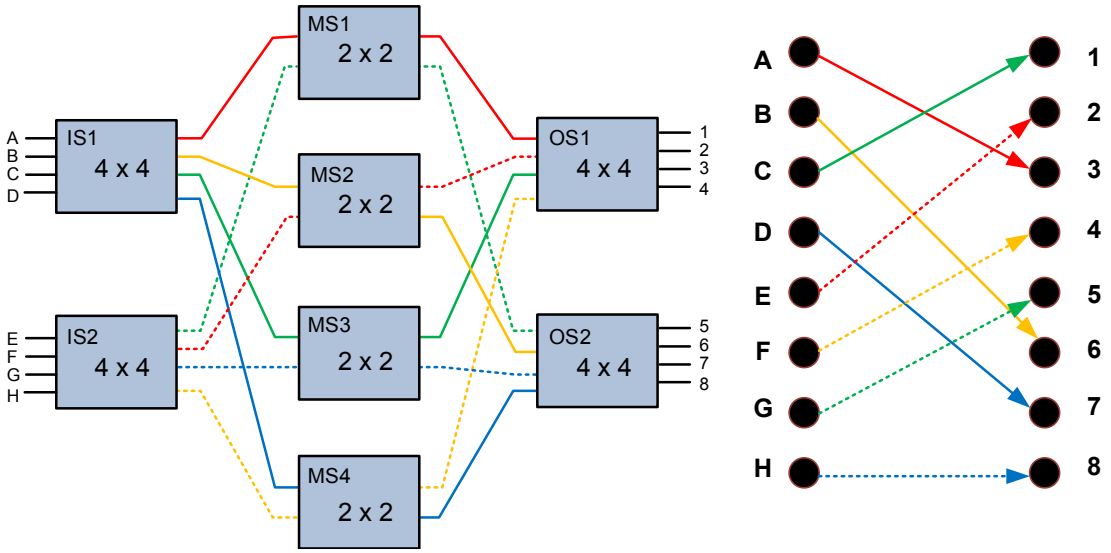

Obrázek 55.17: Příklad Closovy sítě.

### 55.7.2 Modifikovaná přepínací síť Clos

- Pokud $m \geq n$, pak je síť neblokující po přeskládání.
- Výpočetní složitost přeskládání je $O(N \log D)$, kde $D$ je počet barev.
- Proč? Při plnění closova teorému je potřeba hodně bloků. Při této podmínce stačí výrazně méně a složitost výpočtu přeskládání je přijatelná.

Příklad sítě Clos(4,4,2): osmiportový přepínač

- Příklad neblokující konfigurace po přeskládání.

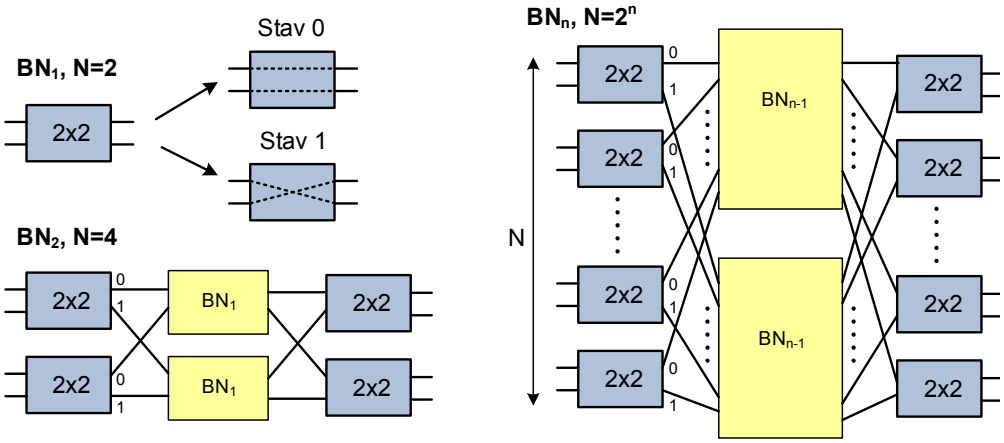



Obrázek 55.18: Příklad modifikované Closovy sítě.

### 55.7.3 Přepínací síť Beneš

$Benes(n)$ také $BN(n)$

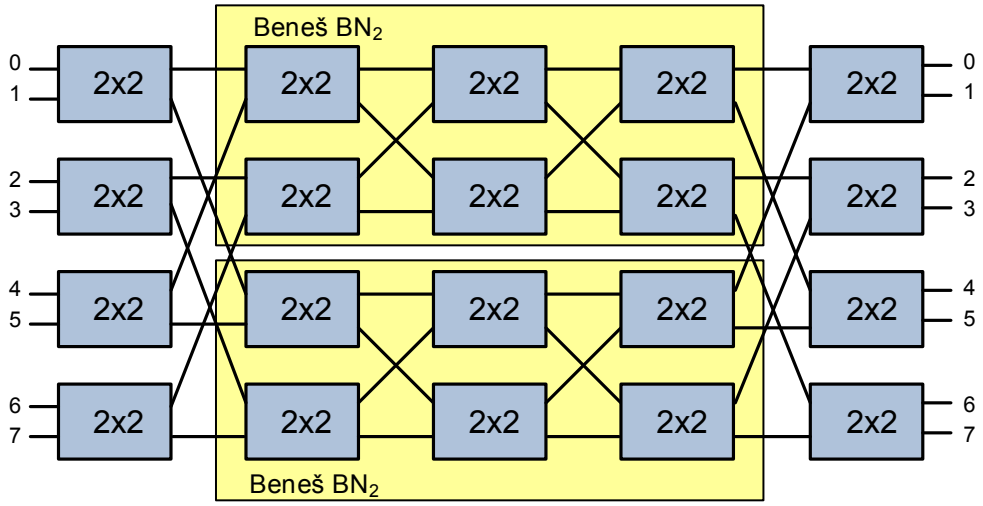
- Základem je modifikovaná síť $Clos(m = 2, n = 2, r = 1)$.
- Rekurzivní konstrukce sítě $Benes(n)$, kde $n$ je stupeň rekurze.
- Vstupní a výstupní přepínače velikosti $2 \times 2$.
- Celkový počet vstupních/výstupních portů $N = 2^n$.
- Prostřední část rekurzivních bloků $BN(n-1)$.
- Neblokující po přeskládání – algoritmus hledání propojení (žádné soupeření o vnitřní uzly nebo výstupní porty).




Obrázek 55.19: Obecné schéma Benešovy sítě.

Příklad: přepínací síť Beneš $BN_3$

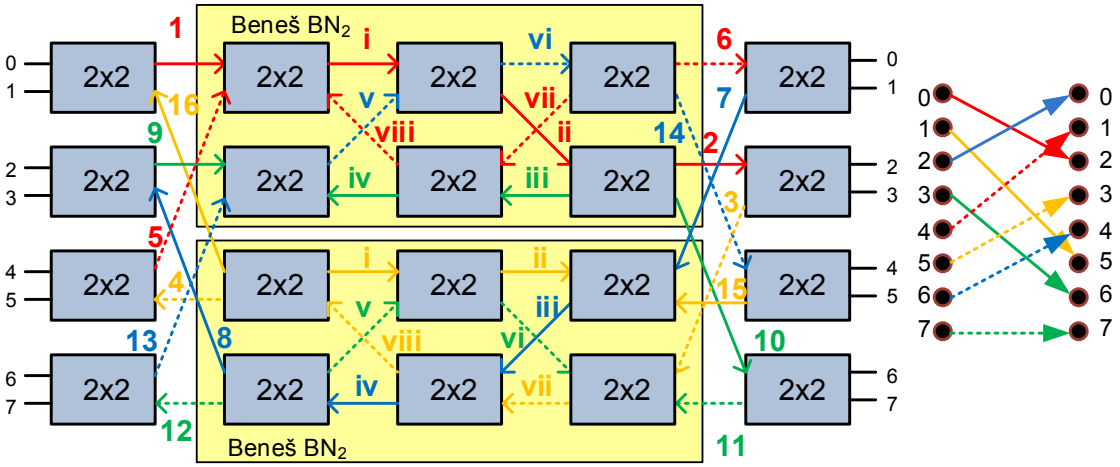
- N = 8 vstupů, 5 stupňů
- Použito např. u směrovače Cisco CSR-1.




Obrázek 55.20: Příklad Benešovy sítě.

Algoritmus hledání propojení

- Horní podsíť $BN_i$ slouží pro dopředné směrování, spodní podsíť $BN_i$ pro zpětné.
- Postupuje se od vstupů k výstupům a zpět, vždy pro sousední porty.
- Nejprve se propojí vstupní a výstupní bloky, poté rekurzivně síť $BN_{i-1}$.




Obrázek 55.21: Příklad Benešovy sítě s přeskládáním.

# Kapitola 56

# PDS – Základní funkce směrovače, zpracování paketů ve směrovači, typy přepínání a architektur.

## 56.1 Zdroje

- `05-routing.pdf`
- `PDS_2021-03-12.mp4`

## 56.2 Úvod a kontext

**Fragmentace paketů** Uzel v síti (směrovač) dostane paket o velikosti $n$. Paket má přeposlat přes výstupní rozhraní do sítě, ve které je $MTU < n$. Aby paket mohl být odeslán, musí být rozdělen (fragmentován) na více menších paketů (fragmenty) a odeslán po částech. Na straně příjemce pak musí nastat opačný proces – defragmentace.

**MTU** MTU (*Maximum Transmission Unit*) je největší velikost paketu, kterou lze v síti odeslat (přes výstupní rozhraní síťového prvku). Závisí na síťové kartě příslušného rozhraní.

## 56.3 Základní popis směrovače

Směrovač je síťové zařízení, které předává datové pakety mezi počítačovými sítěmi. Základní charakteristika:

- Pracuje s pakety (síťová vrstva, L3).
- Tvoří rozhraní mezi počítačovými sítěmi (provádí překlad NAT).
- Klasifikuje a filtruje pakety (firewall, ACL).
- Provádí fragmentaci a defragmentaci.

**Činnost**

1. Vypouzdření paketu z L2 (odebrání L2 hlavičky) a kontrola jestli je v pořádku (pomocí kontrolního součtu).

2. Vyhledání cesty kam se má paket směrovat a překlad adresy NAT (pomocí směrovací tabulky).
3. Určení cílové MAC adresy na základě cílové IP adresy (pošle ARP dotaz).
4. Určení výstupního rozhraní.
5. Sestavení výsledného paketu podle výstupního rozhraní (zapouzdření do příslušné L2 technologie – přidání L2 hlavičky).

**Co ovlivňuje propustnost**

- Rozbalení paketu.
- Vyhledání směrovací cesty.
- Překlad NAT.
- Vyhledání cílové MAC adresy.
- Zabalení paketu do správné technologie.

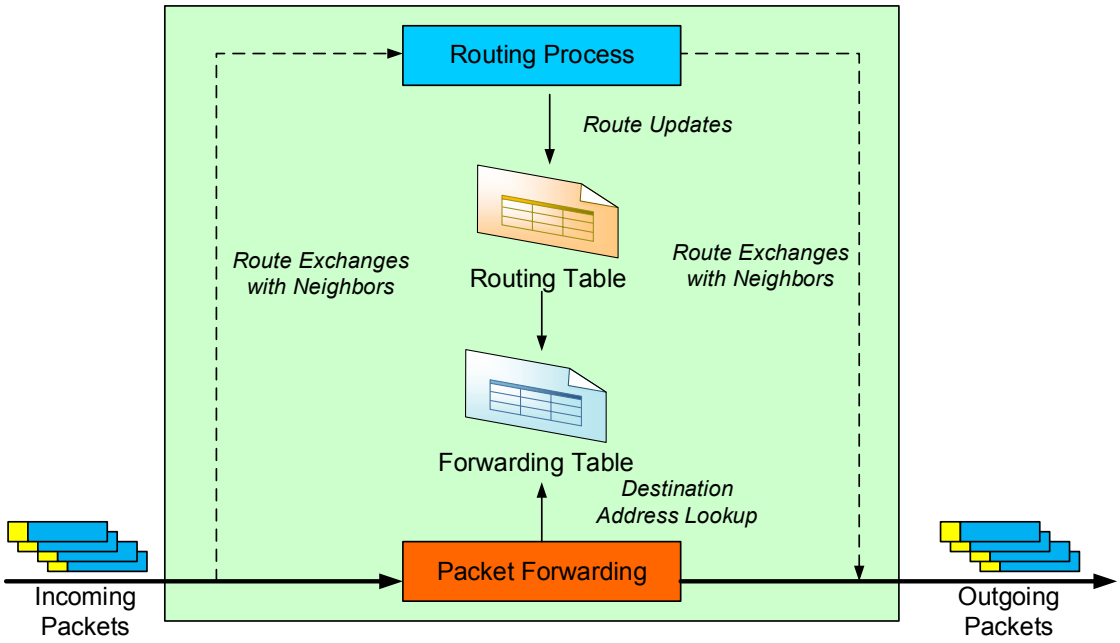
**Typy**

- Páteřní směrovače – součástí pátečních sítí mezi ISP.
- Hraniční směrovače – mezi zákaznickými sítěmi a ISP.
- Podnikové směrovače – propojení koncových systému v podnikových (*enterprise*) sítích.
- Domácí směrovače.

## 56.4 Operace co směrovač vykonává

Pakety mohou být určeny buď přímo pro směrovač (aktualizace směrovací cesty přes směrovací protokoly – aktualizace směrovací tabulky) a nebo jinému uzlu v síti. V takovém případě směrovač paket přeposílá dalším uzlům v síti.

- **Směrování** (*routing*) – Určování cest paketů v prostředí počítačových sítí. Cílem je doručit paket adresátovi, pokud možno co nejefektivnější cestou. Směrování zajišťují nejen směrovače, ale i koncové stanice (při vysílání).
- **Přepínání** (*forwarding, switching*) – Přepnutí paketu ze vstupního rozhraní na výstupní (viz architektura přepínačů).




Obrázek 56.1: Základní činnost směrovače.

**Směrovací tabulka** Směrovací tabulka (*routing table*) obsahuje informace nutné a dostačující pro směrování (`ipprefix` – IP prefix cílové sítě, `nexthop` – IP adresa dalšího uzlu, `metrika` – cena cesty, počet hopů). Informace do ní se získavají pomocí směrovacích protokolů a nebo staticky (administrátor provede manuálně). Směrovací tabulku mají i koncové stanice (počítače), aby mohli posílat pakety do internetu.

| IP prefix | Next hop |
|---|---|
| 10.5.0.0/16 | 192.168.2.254 |
| 10.15.0.0/16 | 104.17.2.1 |
| 88.0.0.0/8 | 129.1.1.1 |
| | |

Obrázek 56.2: Příklad směrovací tabulky (bez metriky).

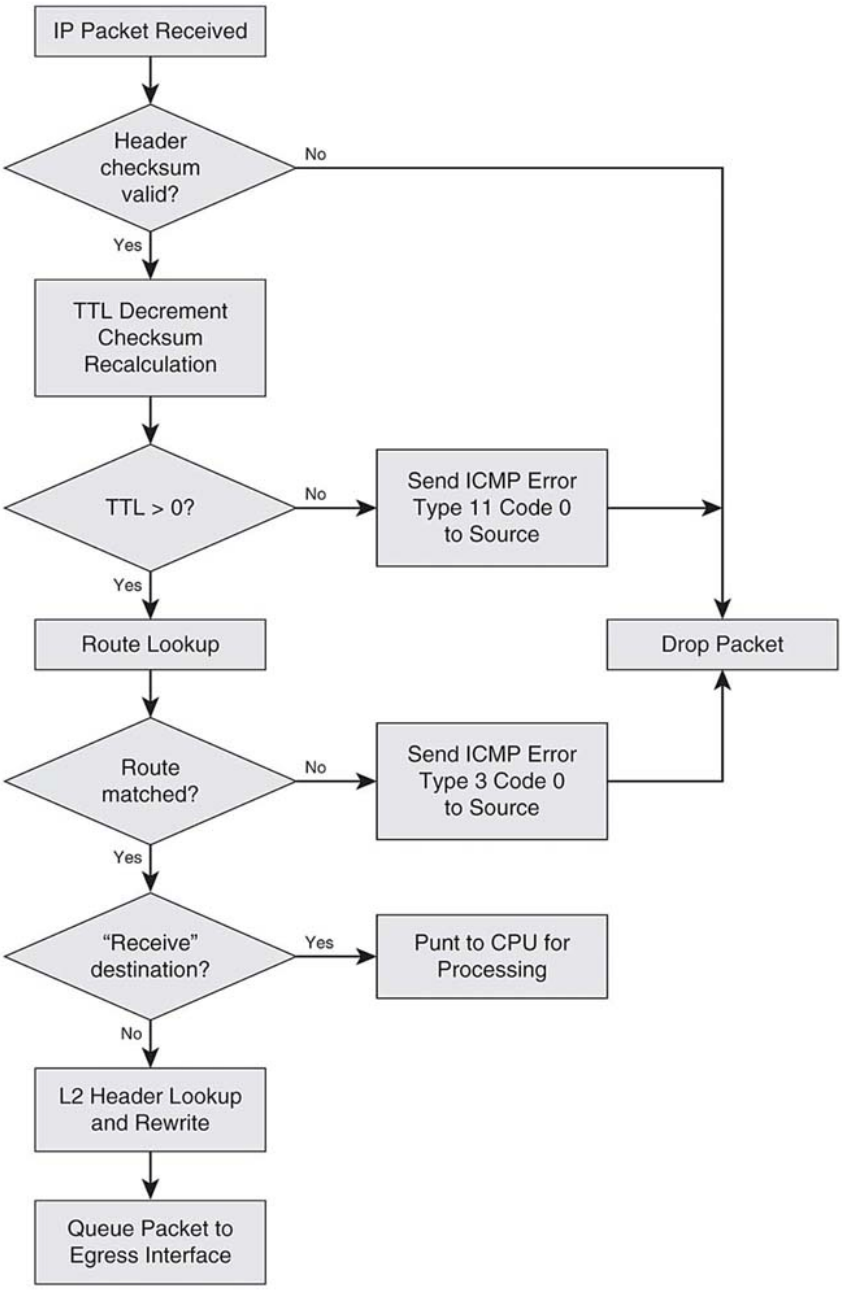
**Přepínací tabulka** Přepínací tabulka (FIB, *forwarding information base*, *forwarding table*) doplňuje směrovací tabulku o další informace, které jsou nutné pro sestavení cílového paketu (výstupní rozhraní, zdrojová MAC adresa, cílová MAC adresa). Informace do ní se získavají pomocí ARP protokolu a doplnění vlastních informací (`dst MAC address`).

| IP prefix | Interface | dst MAC address | src MAC address |
|---|---|---|---|
| 10.5.0.0/16 | eth0 | 00:0F:1F:CC:F3:06 | B8:6B:23:EA:FC:20 |
| 10.15.0.0/16 | eth1 | 01:12:11:A0:17:A0 | 00:16:17:E1:28:5F |
| 88.0.0.0/8 | eth2 | 01:3F:04:10:03:15 | 00:1B:63:8A:DB:1A |
| | | | |

Obrázek 56.3: Příklad přepínací tabulky.

**Zpracování IP paketu (již vypouzdřeno z L2)**

1. Validace hlavičky L3 (formát, verze, délka).
2. Kontrola hodnoty TTL a její dekrementace. Pokud je TLL 0, tak se paket zahodí a pošle ICMP zpráva.
3. Přepočítání kontrolního součtu.
4. Zpracování rozšířených voleb IP protokolu (timestamp, record route, strict source route).
5. Vyhledání cesty (next-hop) na základě cílové adresy (lokální doručení, unicast, multicast). Pokud se nepodaří vyhledat cesta, tak se paket zahodí a pošle ICMP zpráva.
6. Fragmentace a defragmentace IP paketů (pokud $MTU_{in} < MTU_{out}$).
7. Zpracování zpráv ICMP a IGMP.




Obrázek 56.4: Diagram zpracování paketu ve směrovači.

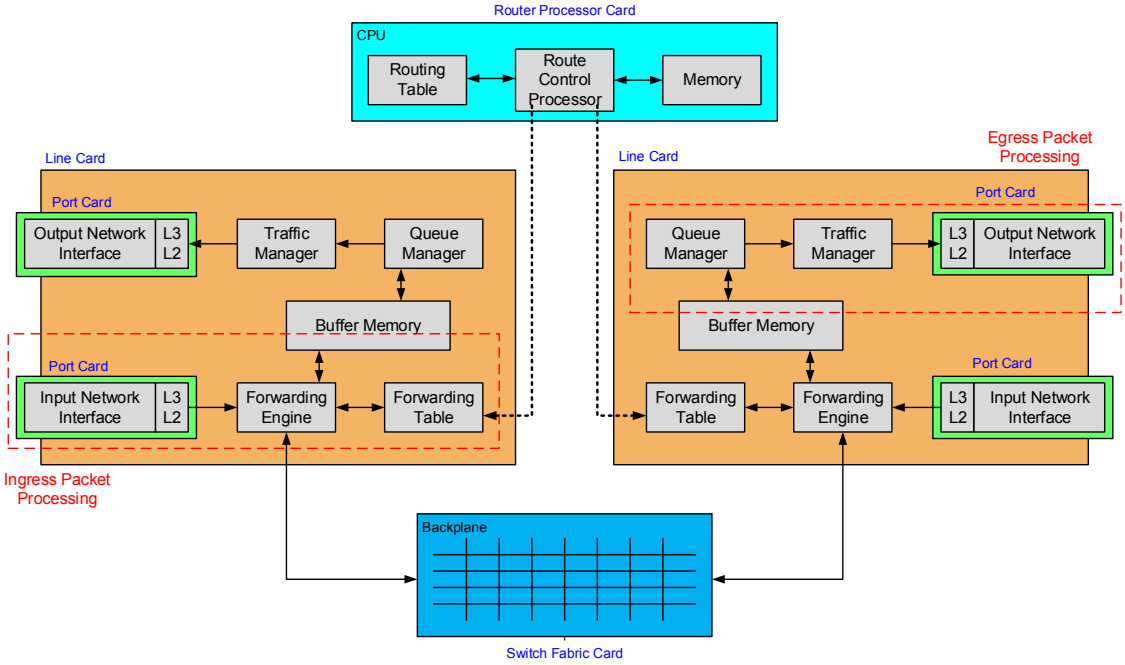
## 56.5 Architektura směrovače

Rozdělíme na fyzické části (z hlediska hardware) a funkční části (z hlediska komponent co něco vykonávají).

### 56.5.1 Fyzické části

- Procesorová část (*Router Processor Card*)
- Přepínací logika (*Switch Fabric Card*)

- Síťové karty (*Line Card*) – každá má vstupní a výstupní rozhraní (*Port Card*)




Obrázek 56.5: Fyzické části (procesory, paměti) směrovače.

### 56.5.2 Funkční části

**Procesorová část** procesor, paměť, směrovací tabulka

- Implementuje směrování na obecném CPU
- Zpracovává směrovací informace: aktualizace, udržuje sousedství
- Obsluhuje směrovací tabulku
- Přenáší data do přepínací tabulky
- Zpracovává pakety, které nelze směrovat pomocí FIB
- Vytváří chybové zprávy ICMP

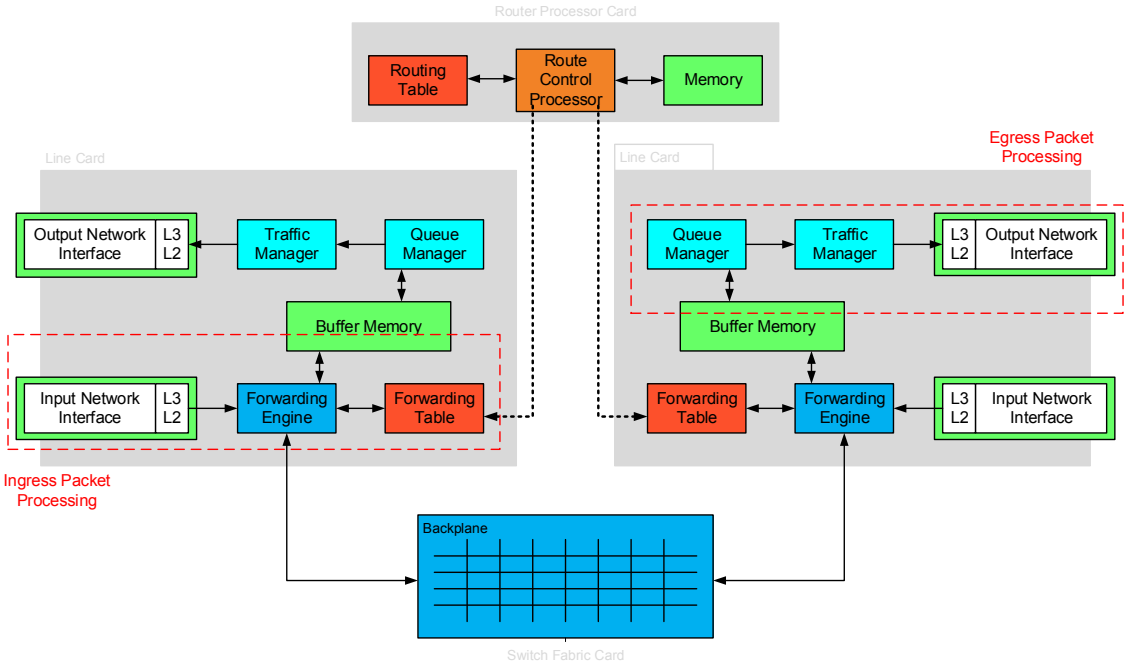
**Přepínací logika** propojovací deska (*Backplane*)

- Propojuje síťové rozhraní
- Vytváří sdílené (shared) či přepínané (switched) propojení
- Rychlost přepínání odpovídá přenosovému pásmu všech rozhraní

**Síťové karty**

- Vstupní/Výstupní síťové rozhraní (*Input/Output Network Interface*)
  - Obsahuje více vstupních/výstupních portů
  - Odstraní zapouzdření L2
  - Předá hlavičku L3 přepínacímu modulu

  - Uloží paket do paměti
  - Zapouzdří odchozí pakety
- Řízení přepínání (*Forwarding Engine*)
  - Dostane hlavičku L3 ze síťového rozhraní
  - Určí výstupní rozhraní podle informace ve FIB
  - Provádí klasifikace paketů pro podporu QoS na výstupu
- Správce front (*Queue Manager*)
  - Ukládá pakety do vyrovnávací paměti, pokud je výstupní port obsazen
  - Spravuje výstupní frontu → různé typy výstupních front
  - Při zaplnění fronty vybírá a zahazuje pakety podle definované politiky
- Správce provozu (*Traffic Manager*)
  - Prioritizuje a reguluje výstupní provoz podle požadavků QoS
  - Omezuje či ořezává výstupní provoz (shaping, policing)

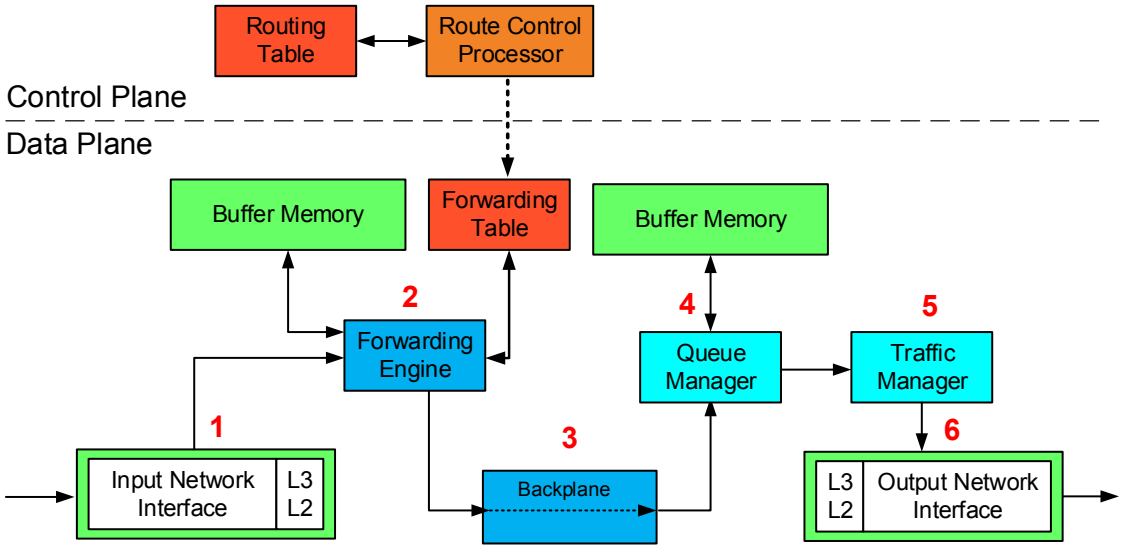



Obrázek 56.6: Funkční částí (procesy).

## 56.6 Zpracování paketu ve směrovači

Směrovač můžeme rozdělit na dvě části:

- *Data Plane* – Specifický hardware pro zpracování paketu (rychlé zpracování) – ASIC.
- *Control Plane* – Obecný procesor pro realizaci složitějších operací (pomalé zpracování).




Obrázek 56.7: Rozdělení přepínače na části *Data Plane* a *Control Plane* a zpracování rozděleno do 6. fází.

**Kontext paketu** Pomocná datová struktura, která reprezentuje paket uvnitř směrovače. Obsahuje vybrané informace z hlaviček. V rámci jednotlivých komponent směrovače se předává (pokud možno) tato struktura, pro větší efektivitu. Na počátku máme pouze částečný kontext, který se průběžně doplňuje. Na základě něho je pak sestaven výstupní paket.

**Fáze zpracování** Pokud některá z fází skončí neúspěšně, paket je zahozen.

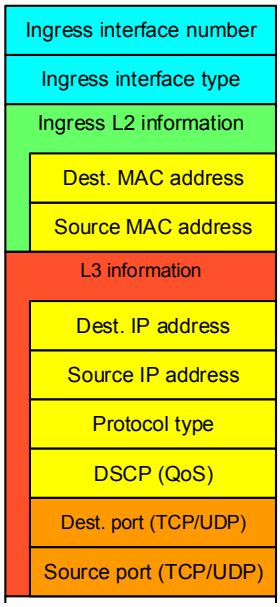
1. Paket přijde na síťové rozhraní
   - Síťová karta zpracuje L2 rámec, zkontroluje FCS
   - Vytvoří kontext paketu: vloží L2 zdrojovou a cílovou adresu
   - Zpracování hlavičky L3: typ protokolu, kontrolní součet, TTL
   - Doplnění kontextu o informace L3: IP adresy, typ protokolu, DSCP + porty
2. Zpracování v přepínacím modulu
   - Vyhledání cesty v přepínací tabulce: next hop + výstupní rozhraní
   - Doplnění dalších informací do kontextu paketu
   - Paket uložen do vyrovnávací paměti → adresa vložena do kontextu
3. Přeposlání kontextu propojovací deskou
   - Paket i kontext jsou přeneseny na výstupní rozhraní
4. Zpracování správcem front
   - Podle priority v kontextu paketu je paket vložen do příslušné výstupní fronty
   - Obsluha fronty probíhá podle daného plánovacího algoritmu
5. Předání kontextu správci provozu
   - Kontrola omezení rychlosti (shaping) dle kontextu

- Překročení rychlost: zahození či zpomalení

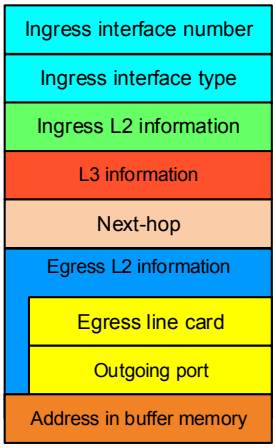
6. Výstupní síťové rozhraní
   - L3: aktualizace TTL, přepočítání kontrolního součtu
   - L2: přidání hlavičky, výpočet CRC
   - Odeslání paketu (zapsání na výstupní médium)

**Plný kontext paketu**

- Kontext doplněn o výstupní informace
- Základní deska přenese paket i kontext na
- výstupní rozhraní
- Kontext obsahuje adresu uložení paketu v paměti
- Zpracování paketu předáno správci front




(a) Kontext paketu – částečný.




(b) Kontext paketu – úplný.

Obrázek 56.8: Kontext paketu (*ingress* – vstup, *engress* – výstup.)

Paket může být zahozen i na výstupní kartě. Proč?

- Vypršelo TTL
- Jsou plné fronty (politika RED, WRED)
- Filtrování na výstupu

Paket může být zpracován dvěma způsoby: rychlou cestou (přes *data plane*) a pomalou cestou (přes *control plane*).

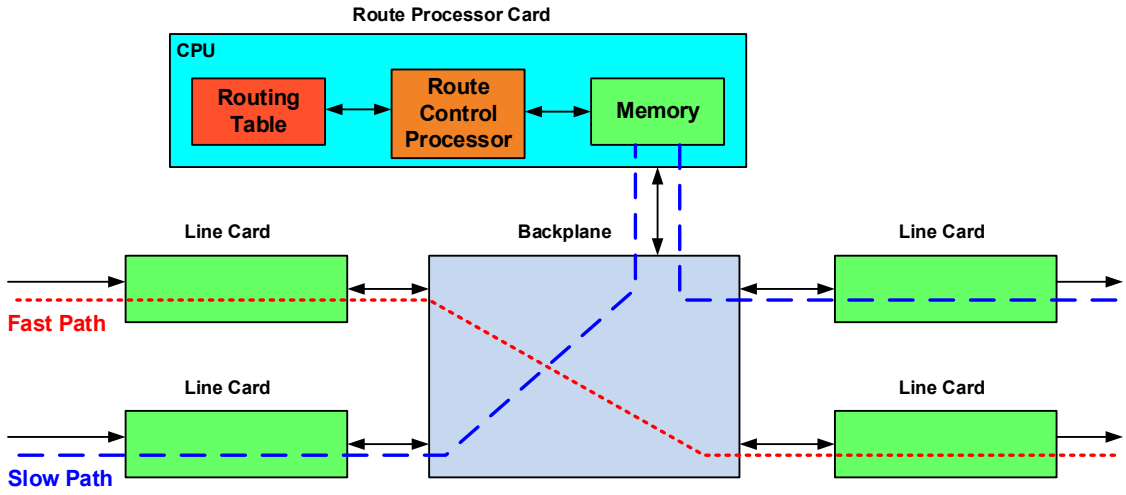
**Rychlá cesta** Paket je možné zpracovat pouze v hardwaru (*data plane*).

- Zpracování hlavičky IP (kontrola verze, délky paketu, snížení TTL, přepočítání kotrolního součtu)

- Přeposlání paketu ze vstupní na výstupní (lokální uložení, přeposlání na jeden port (unicast) či více portů (multicast))
- Klasifikace paketu na základě informací z hlavičky (optimalizované datové struktury pro rychlé uložení a vyhledání)
- Uložení do front, plánování (různé typy front, různé typy obsluhy)

**Pomalá cesta** Paket je částečně zpracován v hardwaru, ale většina zpracování musí proběhnout softwarově (*control plane*). Síťová karta nedokáže zpracovat paket sama a musí se využít centrální procesor.

- Zpracování ARP (zjištění výstupní adresy L2: první paket, ostatní pakety)
- Fragmentace a defragmentace
- Generování zpráv ICMP
- Zpracování směrovacích informací




Obrázek 56.9: Dva způsoby zpracování paketu ve směrovači – pomalou cestou a rychlou cestou.

## 56.7 Typy přepínání paketů

Přepínání paketů z jednoho rozhraní na druhé na základě směrovacích informací je jedna z nejdůležitějších funkcí směrovače. Proces přepínání paketů zahrnuje:

- Zjištění, zda cíl cesty paketu je dosažitelný.
- Vyhledání nejbližšího uzlu na cestě (next-hop) a určení výstupního rozhraní.
- Vyhledání informací pro vytvoření L2 hlavičky paketu na výstupu.

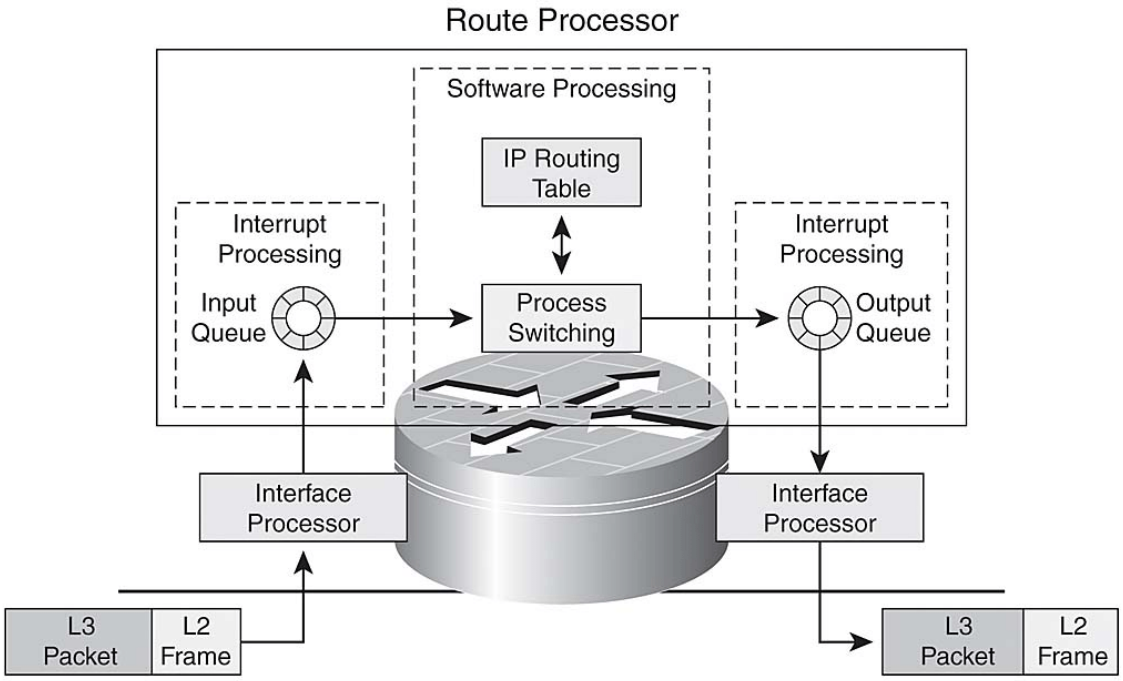
### 56.7.1 Softwarové přepínání (*Process Switching*)

- Přepínání pomocí centrálního procesoru s centrální pamětí.
  - Pomalé – přístup do paměti, obecný procesor (softwarové zpracování).
- Pro každý paket se hledá cesta ve směrovací tabulce a určuje se MAC adresa cíle.

- Směrovač využívá standardní mechanismus přepíná procesů v OS (přeřušení).

**Fáze softwarového přepínání**

1. I/O procesor detekuje paket na vstupním médiu. Přenese ho do vstupního bufferu.
2. I/O procesor vygeneruje přerušení. Během přerušení určí centrální procesor typ paketu a zkopíruje ho do centrální paměti.
3. Centrální plánovač zjistí, že ve vstupní frontě je paket. Naplánuje jeho další zpracování procesem.
4. Proces pro zpracování vyhledá ve směrovací tabulce další uzel (next hop) a výstupní rozhraní. V paměti ARP cache vyhledá MAC adresu dalšího uzlu.
5. Přepíše L2 hlavičku paketu a umístí paket do výstupní fronty na výstupním portu.
6. Paket vložen do fronty na výstupním portu.
7. I/O procesor detekuje paket ve vysílací frontě. Zapíše ho na síťové médium.




Obrázek 56.10: Softwarové přepínání.

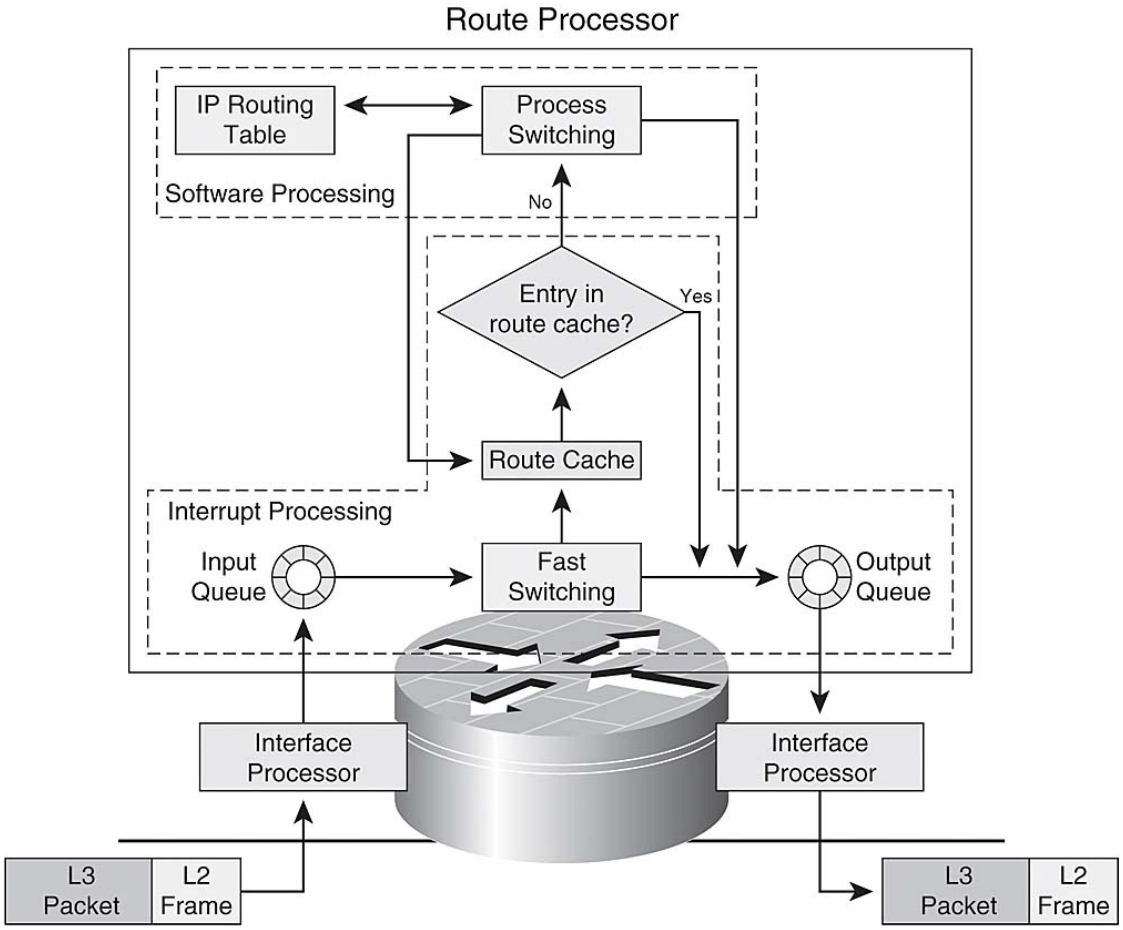
**Optimalizace**

- Nejdéle trvá: na základě cíle vyhledávání informací ve směrovací tabulce, najdu next hop, najdu příslušný záznam v ARP tabulce
- Můžeme využít cache (která si bude pamatovat poslední záznamy.

### 56.7.2 Rychlé přepínání (*Fast Switching*)

- Využívá vyrovnávací paměť *route cache* s předpočítanou L2 hlavičkou.
- První paket toku se přepíná softwarově, další pakety toku jdou rychlou cestou.

- Při přepínání paketu je vložen do paměti *route cache* nový záznam.
- Neexistuje synchronizace mezi směrovací tabulkou, ARP cache a *route cache*.
- Záznam v *route cache* se zneplatní při změně ARP cache či směrovací tabulky.
- Při zaplnění paměti nad určitou mez se začnou záznamy náhodně zahazovat.

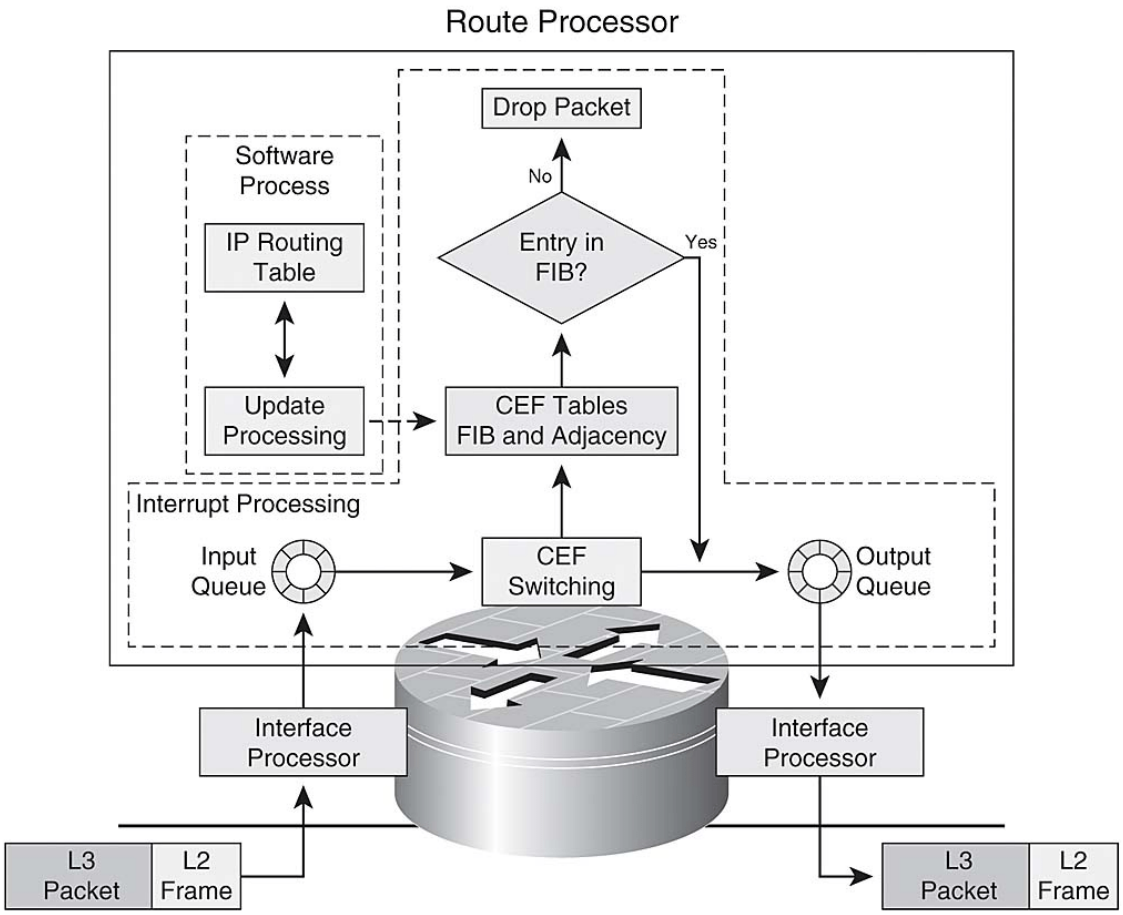



Obrázek 56.11: Rychlé přepínání.

### 56.7.3 Expresní přepínání CEF (Cisco Express Forwarding)

[[todo]]

- Tabulka CEF se předpočítá na základě směrovací tabulky a tabulky sousedů ještě před příchodem paketu → nedochází k softwarovému přepínání.
- Oddělení směrovacích informací od L2 dat → nedochází ke stárnutí záznamů při expiraci záznamu v tabulce ARP.
- Změny ve směrovací tabulce či tabulce ARP se okamžitě propagují do tabulky CEF.
- ARP tabulka se synchronizuje se záznamy v tabulce sousedů.



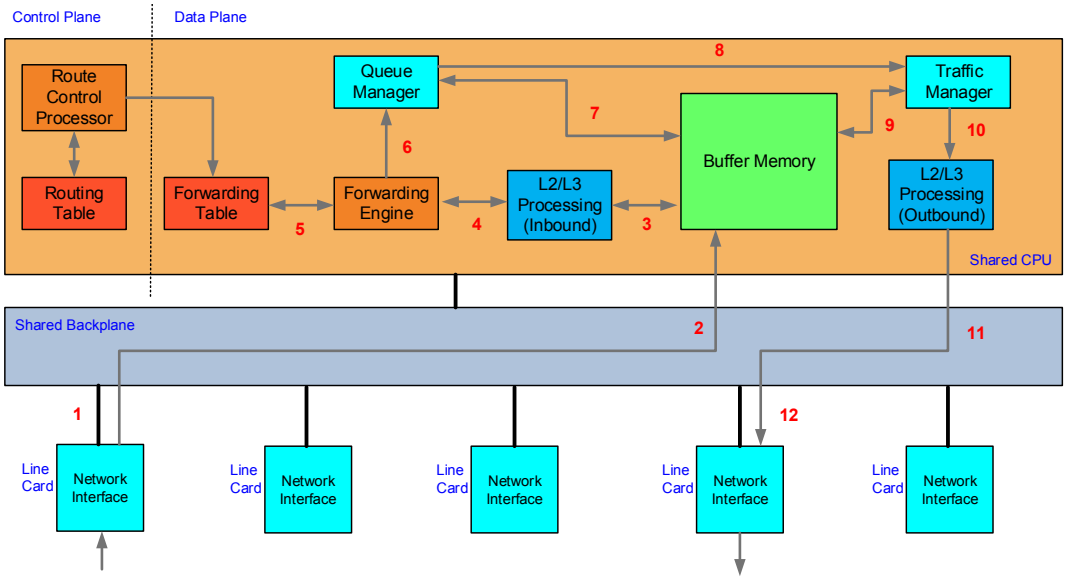

Obrázek 56.12: Expresní přepínání CEF.

## 56.8 Přehled architektur

Architektury směrovačů rozdělmě podle způsobu přepínání paketů.

### 56.8.1 Architektura se sdíleným procesorem

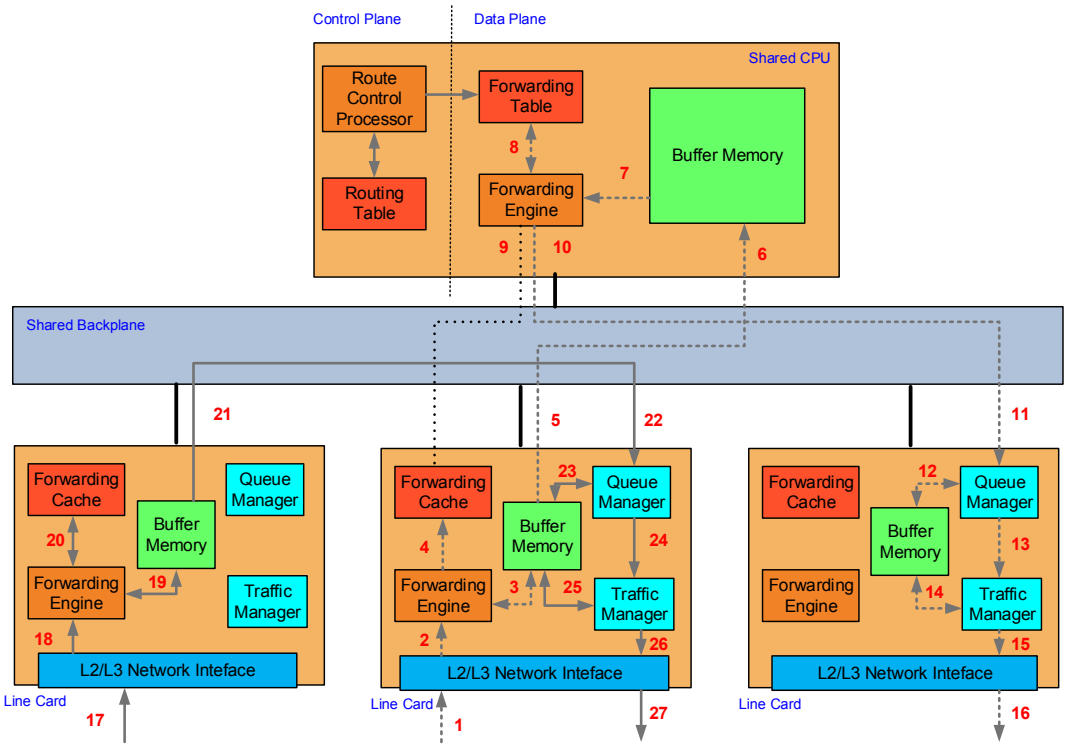
- Využívá softwarové přepínání → každý paket zpracován na CPU.
- Cykly CPU rozděleny mezi přeposílání, směrování a další operace.
- Sdílená sběrnice i procesor ⇒ levné, ale pomalé.




Obrázek 56.13: Architektura se sdíleným procesorem.

### 56.8.2 Architektura se sdíleným procesorem a pamětí cache na kartě

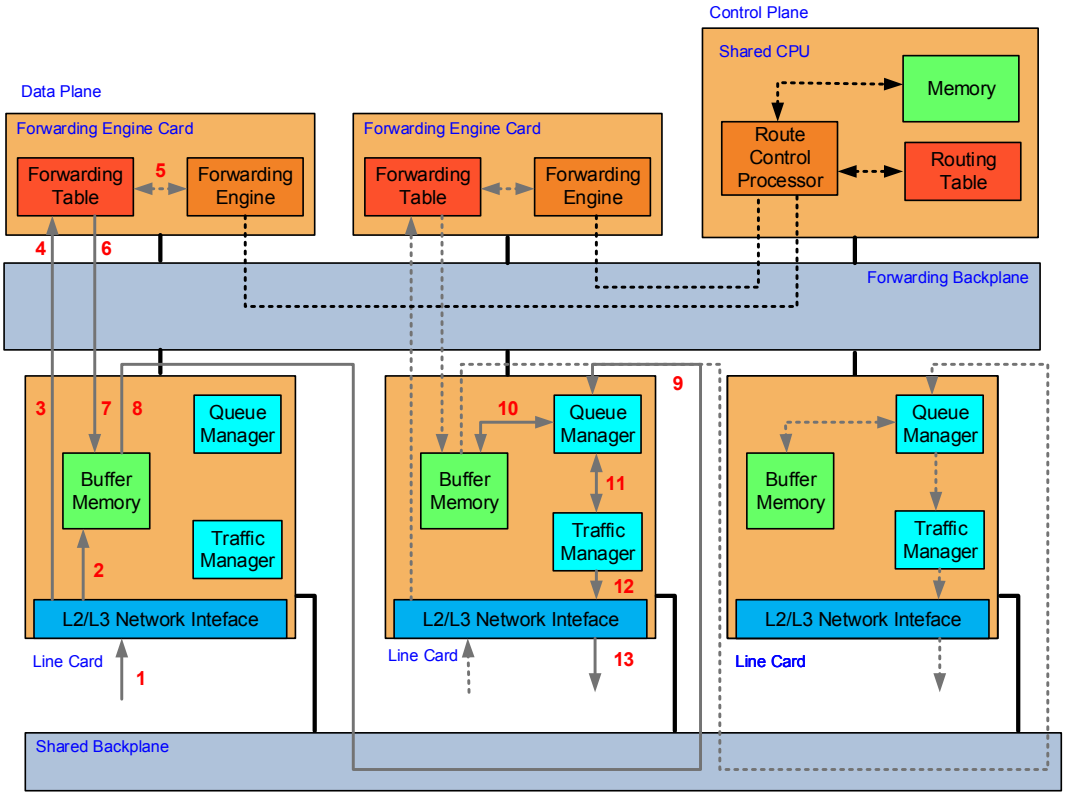
- Varianta s pamětí cache na kartě → synchronizace přepínacích tabulek.
- Síťová karta obsahuje FE pro zpracování hlaviček, paměť a přepínací tabulku.
- Rychlé přepínání (Fast Switching): první paket vs. další pakety.




Obrázek 56.14: Architektura se sdíleným procesorem a pamětí cache na kartě.

### 56.8.3 Architektura s nezávislými moduly FE

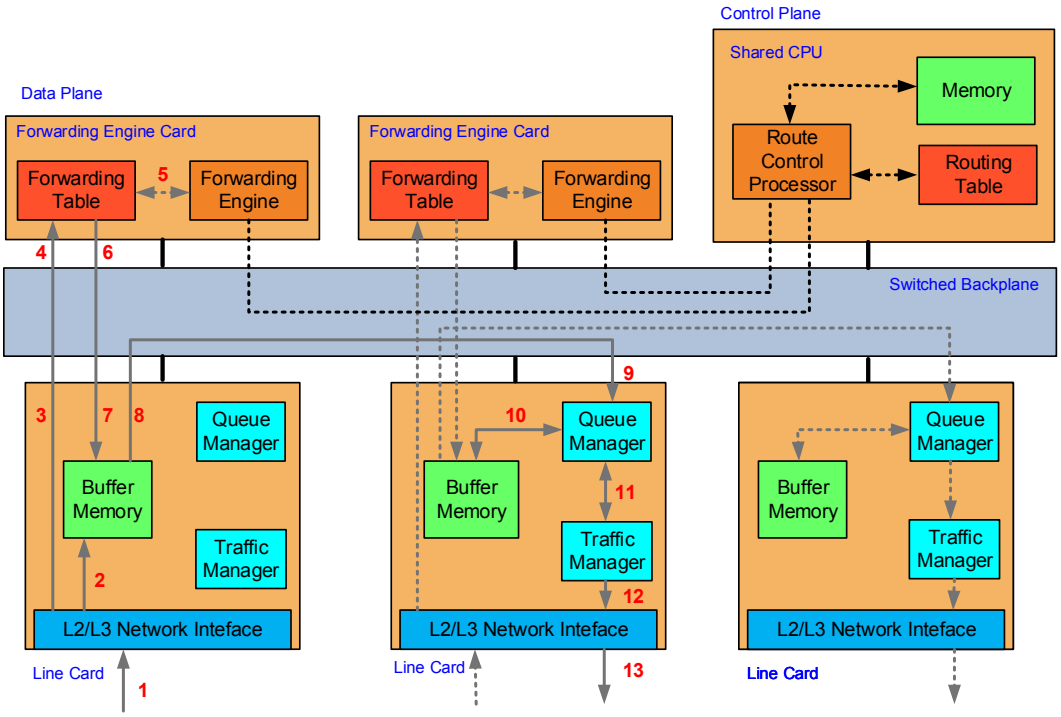
- Přepínací moduly FE implementovány na speciálních kartách.
- Paralelní zpracování paketů, dvě sběrnice (sdílená a přepínaná).




Obrázek 56.15: Architektura s nezávislými moduly FE.

### 56.8.4 Architektura s nezávislými moduly FE a přepínanou sběrnicí

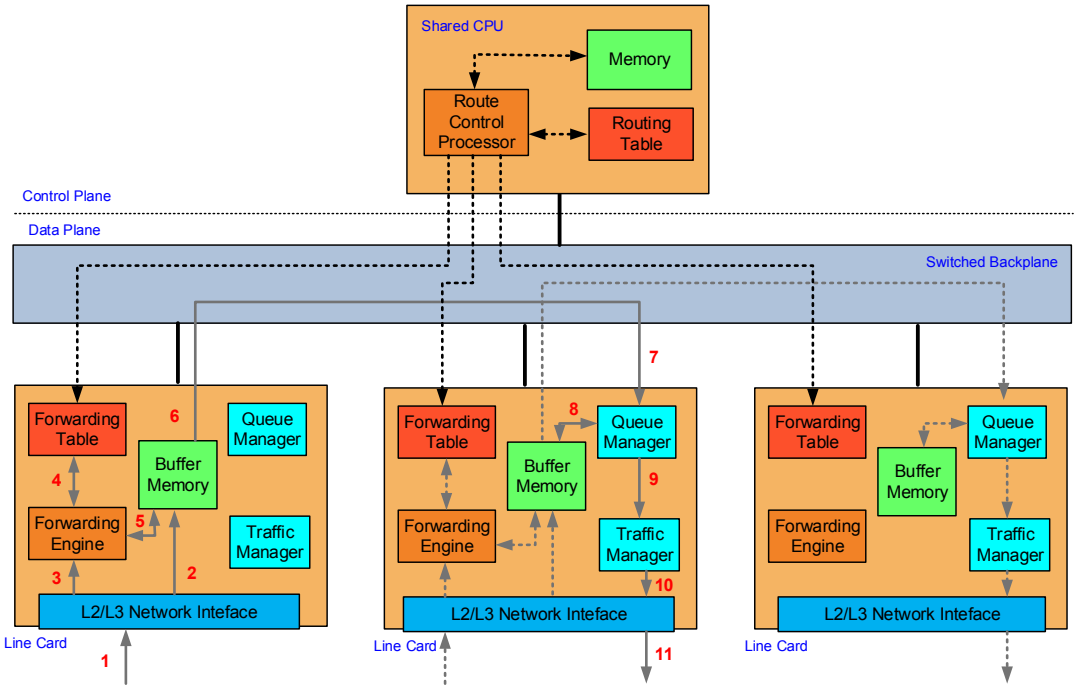
- Varianta s jednou přepínanou sběrnicí ⇒ vyšší propustnost.




Obrázek 56.16: Architektura s nezávislými moduly FE a přepínanou sběrnicí.

### 56.8.5 Distribuovaná architektura (Shared Nothing)

- Veškeré zpracování paketu přeneseno do síťového modulu.
- Oddělení procesu směrování a přepínání → využití technologie CEF.




Obrázek 56.17: Distribuovaná architektura (Shared Nothing).

# Kapitola 57

# PDS – Metody pro výpočet směrování v sítích (Bellman-Ford, Dijkstra, Path vector, DUAL).

## 57.1 Zdroje

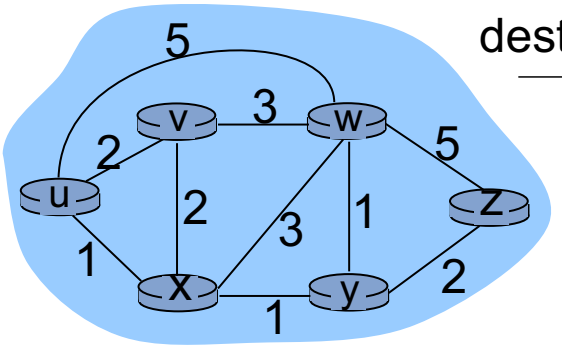
- `09-teorie-smerovani.pdf`
- `PDS_2021-04-16.mp4`

## 57.2 Směrování

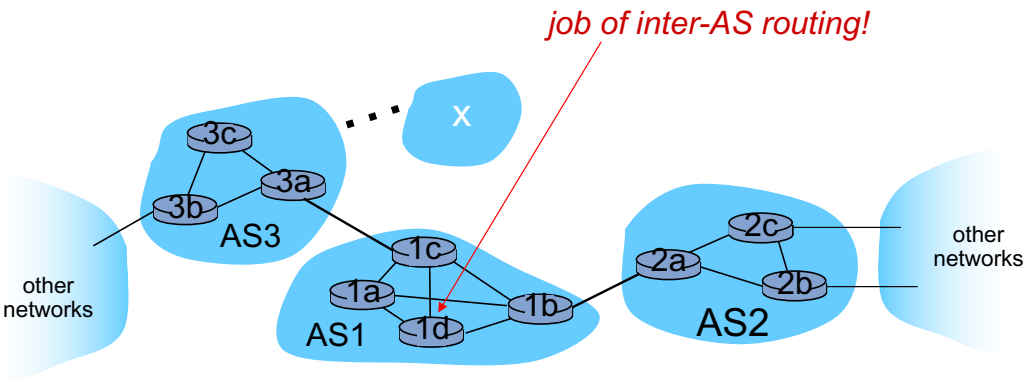
Z pohledu směrování můžeme počítačovou síť zjednodušit na neorientovaný, ohodnocený graf, kde uzly jsou směrovače a hrany jsou propojení mezi něma. Směrování je pak hledání nejkratší cesty mezi dvěma uzly, resp. ze zdrojového do všech.

**Autonomní systém** Není možné směrovat přes celý internet (příliš velký, příliš mnoho uzlů). Využíváme abstrakce, kdy se na podsíť díváme jako na uzel – tzv. autonomní systém. Tímto agregujeme více informací do jedné (přesné adresy uzlů, na vyšší úrovni reprezentujeme pouze adresou sítě). Internet je pak síť sítí. Může být více úrovní (autonomní systém autonomních systémů).

**Hierarchické směrování** Hierarchické směrování znamená, že nesměrujeme mezi každým směrovačem na světě, ale využíváme autonomní systémy. Směrujeme tedy globálně mezi autonomníma systémama a pak uvnitř mezi směrovačema.




Obrázek 57.1: Příklad počítačové sítě zobrazené jako neorientovaný ohodnocený graf $G = (V, E)$, $V = \{u, v, w, x, y, z\}$, $E = \{(u, v), \dots\}$, $w = \{((u, v), 2), \dots\}$.



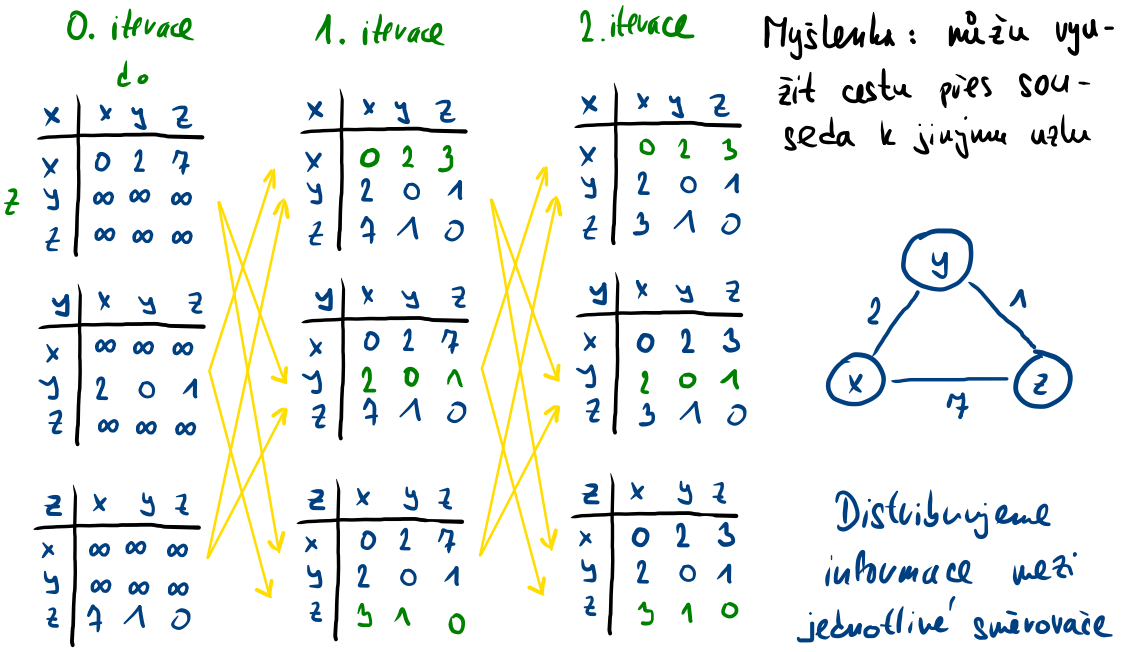

Obrázek 57.2: Příklad autonomních systémů (např. VUT, CESNET, UPC, ...).

## 57.3 Distance Vector přístup

- Distribuované směrovací informace – Každý uzel, zná pouze omezenou část topologie sítě. Konkrétně má informace pouze co sám zná a informace od svých sousedů.
- Používá se pro směrování uvnitř autonomních systémů.
- *Single-metric.*

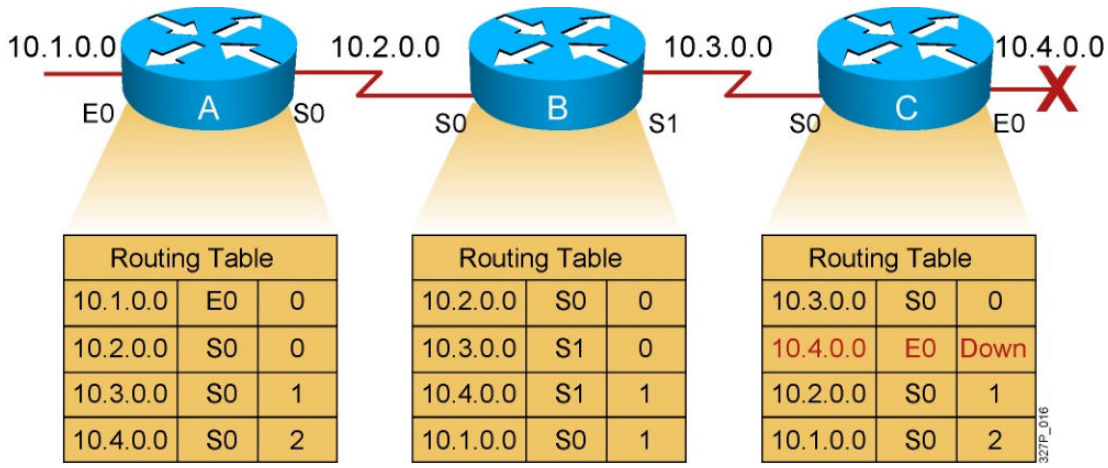
### 57.3.1 Algoritmus Bellman-Ford

- *Pro formální vysvětlení a pseudokód viz otázku „Hledání nejkratších cest ze zdrojového uzlu do všech ostatních uzlů grafu“.*
- Jde o algoritmus hledání nejkratších cest ze zdrojového uzlu do všech ostatních uzlů. To znamená, že ho provádí každý uzel (směrovač).
- Využívá jej např. směrovací protokol RIP (*Routing Protocol*).
  - Komunikuje přes UDP na portu 520.
  - Metrika: *hop count.*
  - Proti *Infinity Count* (viz dále) se brání vylepšením *Split Horizon*: Nikdy se nepošle informace o cestě zpátky rozhraním, ze kterého přišla.




Obrázek 57.3: Příklad výpočtu nejkratší cesty pomocí algoritmu Bellman-Ford.

**Infinity Count** V případě, že vypadne některá ze sítí, může nastat problém počítání ceny cesty do nekonečna. Jelikož si uzly vyměňují pravidelně směrovací informace. Viz následující obrázek.

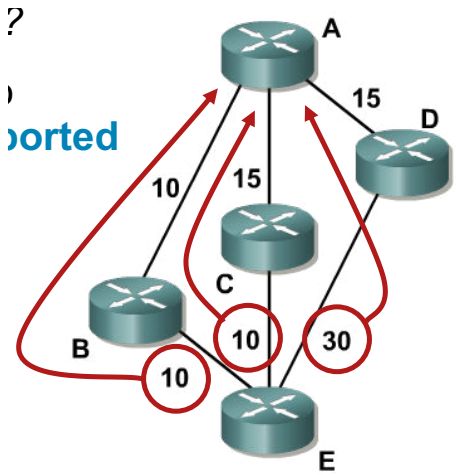



Obrázek 57.4: Příklad *Infinity Count*. Síť 10.4.0.0 vypadne a pro uzel C je najednou nedosažitelná. Uzel B se cestu do sítě 10.4.0.0 naučil dříve od C. Uzel C čerpá informaci o cestě do sítě z uzlu B. Tímto způsobem se cena do sítě zacyklí a jde k $\infty$.

### 57.3.2 Diffusing update algorithm (DUAL)

- Algoritmus DUAL (diffusing update algorithm) je algoritmus společnosti Cisco, který zajišťuje, že daná trasa je přepočítána globálně, kdykoli by mohla způsobit směrovací smyčku.

- Klíčová je tzv. **Feasibility Condition**, která zajišťuje, že jsou vždy vybírány pouze trasy bez smyček. Pokud podmínka platí, nemohou vzniknout žádné smyčky, ale podmínka může za určitých okolností odmítnout všechny trasy k cíli, přestože některé jsou bez smyček.
  - Vzdálenost sousedů k cílové síti nazveme RC (*reported distance*).
  - Nejlepší vzdálenost k danému uzlu nazveme FD (*feasible distance*).
  - Pokud platí $RD < FD$, tak cesta neobsahuje smyčku.
- Využívá jej směrovací protokol EIGRP (*Enhanced Interior Gateway Routing Protocol*) od společnosti Cisco.
  - Pro hledání nejkratší cesty používá algoritmus Bellman-Ford + *Feasibility Condition*.
  - Komunikace: vlastní protokol zabalený do IP paketu (*Reliable Transport Protocol*).
  - Kompozitní metrika – jedno číslo spočítaný na základě několika parametrů (šířka pásma, zpoždění, rychlost, ...).




Obrázek 57.5: Příklad výpočtu nejlepší cesty a uplatnění feasibility condition.

Cesta z A do E:

$$RD(B) = 10 \qquad RD(C) = 10 \qquad RD(D) = 30$$
$$via(B) = 20 \qquad via(C) = 25 \qquad via(D) = 45$$

Potom z pohledu A:

- Cesta $via(B)$ je $FD$ a platí podmínka $FD > RD(B)$, tudíž cesta přes B neobsahuje smyčku.
- Pro cestu $via(C)$ platí podmínka $FD > RD(C)$, tudíž cesta přes C neobsahuje smyčku.
- Pro cestu $via(D)$ neplatí podmínka $FD > RD(D)$, tudíž cesta přes D může obsahovat smyčku a není brána v potaz.

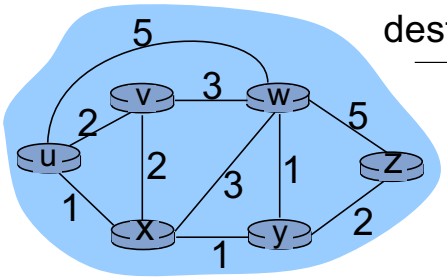
## 57.4 Link State přístup

- Globální směrovací informace – Každý uzel zná celou topologii sítě. Na začátku si uzly vymění informace o topologii sítě.
- Používá se pro směrování uvnitř autonomních systémů.
- *Single-metric.*

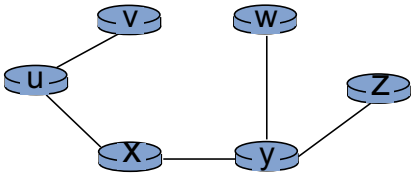
### Dijkstrův algoritmus

- *Pro formální vysvětlení a pseudokód viz otázku „Hledání nejkratších cest ze zdrojového uzlu do všech ostatních uzlů grafu“.*
- Jde o algoritmus hledání nejkratších cest ze zdrojového uzlu do všech ostatních uzlů. To znamená, že ho provádí každý uzel (směrovač).
- Používá ho např. směrovací protokol OSPF (*Open Shortest Path First*).
  - Komunikace: vlastní protokol nad IP.
  - Metrika: odvozena od rychlosti linky.

| Step | $N'$ | $D(v),p(v)$ | $D(w),p(w)$ | $D(x),p(x)$ | $D(y),p(y)$ | $D(z),p(z)$ |
|---|---|---|---|---|---|---|
| 0 | $u$ | $2,u$ | $5,u$ | $1,u$ | $\infty$ | $\infty$ |
| 1 | $ux$ | $2,u$ | $4,x$ | | $2,x$ | $\infty$ |
| 2 | $uxy$ | $2,u$ | $3,y$ | | | $4,y$ |
| 3 | $uxyv$ | | $3,y$ | | | $4,y$ |
| 4 | $uxyvw$ | | | | | $4,y$ |
| 5 | $uxyvwz$ | | | | | |




| destination | link |
|---|---|
| $v$ | $(u,v)$ |
| $x$ | $(u,x)$ |
| $y$ | $(u,x)$ |
| $w$ | $(u,x)$ |
| $z$ | $(u,x)$ |



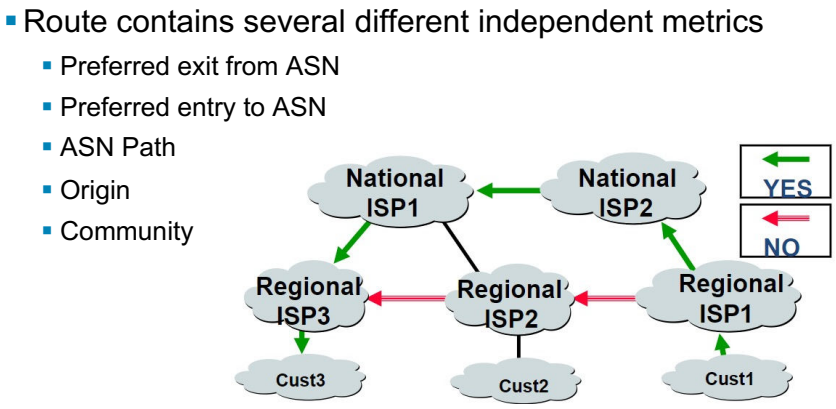

Obrázek 57.6: Příklad výpočtu nejkratší cesty pomocí algoritmu Dijkstra. *Step* značí iteraci, $N'$ je množina již prozkoumaných uzlů, $D$ je pole vzdáleností do uzlu, $p$ je pole předchůdců uzlu.

## 57.5 Path Vector přístup

- Globální směrovací informace.
- Na síť je pohlíženo jako na množinu autonomních systémů.
- Používá se pro směrování mezi autonomními systémy.
- *Multi-metric.*

### Path Vector algoritmus

- Posílá celou cestu (posloupnost uzlů – autonomních systémů), ne jenom vzdálenost.
- Slouží také pro detekci smyček v rámci cesty.
- Cíl je projít přes co nejméně autonomních systémů.
- Umožňuje tzv. *flexible policies*, můžu se rozhodnout jakou cestou budu provoz směrovat. Část provozu můžu přes $AS_1$, jinou část pres $AS_2$, apod.
- Známe celou cestu směrování, víme přes co pakety půjdou, můžeme některý uzel vyřadit.
- Směrovací protokol Border Gateway Protocol (BGP).




Obrázek 57.7: Path vector příklad.

# Kapitola 58

# PDS – Řízení toku dat (flow-control) a prevence zahlcení (congestion-control) na transportní vrstvě (MP-TCP, QUIC, SCTP, DCCP).

## 58.1 Zdroje

- `02-transportni-protokoly.pdf`
- `PDS_2021-02-19.mp4`

## 58.2 Úvod a kontext

**Transportní vrstva** Transportní vrstva je název čtvrté vrstvy modelu vrstvové síťové architektury (ISO/OSI model). Leží mezi vrstvou síťovou (L3) a aplikační (L7).

- Činnost na straně odesílatele: obdrží data z aplikační vrstvy, nasegmentuje je a ke každému segmentu/datagramu přidá L4 hlavičku.
- Činnost na straně příjemce: obdrží segmenty/datagramy, které uspořádá do správného pořadí (*out-of-order* doručení), předá je aplikační vrstvě.
- Komunikace mezi vrstvami L4 a L7 probíhá pomocí socketů.
- Zodpovědnost za *end-to-end* spojení.
- Adresuje aplikace pomocí portových čísel.
- *Quality of Service* – zotavení se z chyb, spolehlivost, řízení toku, řízení zahlcení.

**Multi homing** – Využítí více bodů připojení zároveň (pro jednu komunikaci). Pokud je podporováno a změní se IP adresa, nemusí se navázat nové spojení.

***Connection-oriented*** – Typ spojení, kdy je nejprve nutné, navázat komunikaci. Typicky tři fáze: navázání spojení, přenos dat, ukončení spojení.

***Connection-less*** – Typ spojení, které nerozlišuje jestli spojení existuje, nebo nikoliv. Pouze fáze přenosu dat.

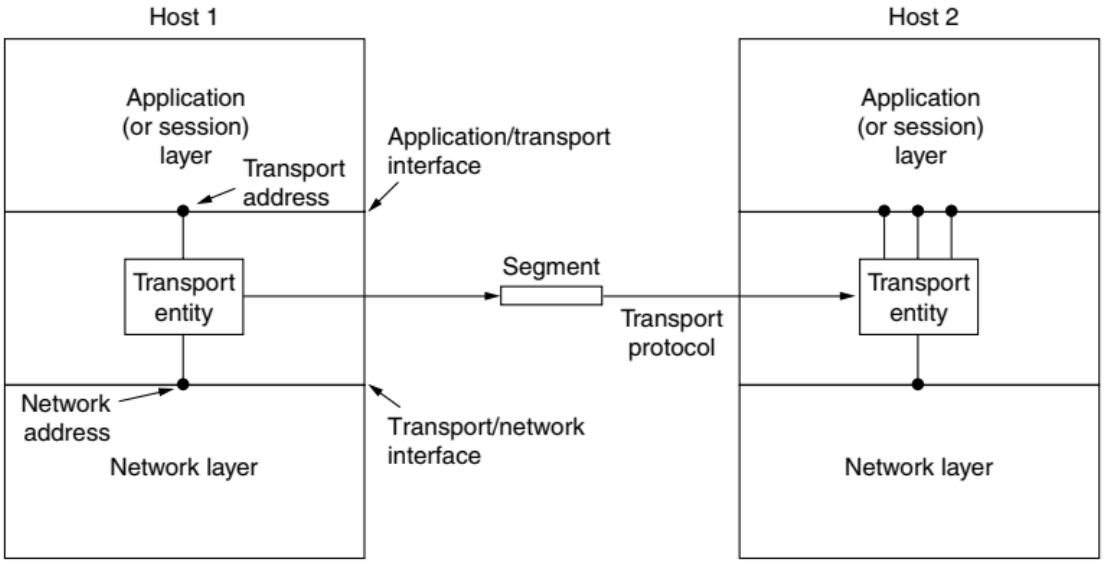
**Round trip time (RTT)** RTT je doba, která uplyne od vyslání paketu z jednoho komunikujícího uzlu k druhému po návrat potvrzení zpátky na první uzel.

**bandwith, throughput, goodput**

- *Bandwith* – šířka pásma (maximální přenos dat danou cestou v jeden moment)
- *Throughput* – propustnost komunikace na úrovni protokolu
- *Goodput* – propustnost komunikace na úrovni aplikace (throughput − režie)

**Bitové chyby**

- „Přeskočení bitu“ (*bit swapping*), jde o chyby na úrovni zpracování informace uzlem v síti (typicky spojené s přenosovým médiem)
- Zabýváme se jimi na úrovni L2
- Řešením je obvykle redundance (Hammingův kód, CRC, Viterbiho algoritmus, ...)




Obrázek 58.1: Transportní vrstva (L4).

## 58.3 Paketové chyby

- Jde o chyby na úrovni přenosu informace mezi dvěma uzly.
- Zabýváme se jimi na transportní vrstvě.
- Řešením je obvykle znovuzaslání paketu.
- Můžeme měřit:
  - PER (*packet error rate*)

$$PER = \frac{\text{pocet chybnych paketu}}{\text{pocet prenesenych paketu}}$$

  - BER (*bit error rate*)

$$BER = \frac{\text{pocet chybnych bitu}}{\text{pocet prenesenych bitu}}$$

- Dále jsou představeny druhy paketových chyb.

**Ztráta (zpoždění) paketu** Jak může dojít ke ztrátě (zpoždění) paketu?

- Neopravitelná bitová chyba (rámec je zahozen na L2)
- Zahlcení linky (směrovače jsou přetíženy a některé pakety zahazují)
- Špatné směrovací tabulky (paket jde špatnou cestou, nebo cyklí a vyprší mu TTL)

**Ztráta fragmentovaných dat** Fragment se může ztratit ze stejného důvodu jako standardní nefragmentovaný paket.

**Duplicita paketu** Příjemce obdržuje stále paket se stejným sekvenčním číslem. Odesílatel si myslí, že se paket k příjemci nikdy nedostal. Proč?

- Ztratí se potvrzení o přijetí paketu.

**Vložení paketu** Příjemce obdrží paket, který do spojení nepatří. Jak k tomu může dojít?

- Přijetí zpožděného paketu, který dorazil až po skončení jednoho a začátku dalšího separátního datového toku.
- Podvrhávání paketu útočníkem snažícím se narušit integritu sítě.
- Paket je chybou „zvláštně zmrzačen“ tak, že chybu nelze rozpoznat (např. je změněm bit v cílové IP adrese) a je směrován jinému příjemci.

**Změna pořadí** Vychází z designu počítačových sítí, kdy paketu mohou proudit různými cestami a s různým zpožděním. Data jsou *out-of-order* a příjemce je musí přeuspořádat.

## 58.4 Řízení toku

Řízení toku (*flow control*) je řízení komunikaci mezi dvěma uzly na straně koncového systému. Řízení zahlcení (*congestion control*) je řízení komunikace v rámci sítě. Alternativně, se za řízení toku považuje koncept detekce a korekce paketových chyb. Často jsou tyto termíny vykládány různě.

### 58.4.1 Detekce paketové chyby

Paketové chyby detekujeme pomocí sekvenčních čísel. Sekvenční číslo je jedinečný identifikátor paketu v rámci datového toku, který identifikuje jeho pořadí. Tímto poznáme ztrátu, duplicitu i změnu pořadí.

Jak rozpoznat ztrátu od zpoždění?

- **Timeout** – Může být fixní, ale v lepším případě se odvozuje od RTT.
- **Negativní potvrzování** – Paket jsem dostal (ACK) vs. paket jsem nedostal (NACK). Pokud je příčina zpoždění zahlcení, tak tento přístup linku ještě více zahltí.

### 58.4.2 Korekce paketových chyb

Pokud paket nedorazil, odesílatel ho pošle znovu. Tzv. *Automatic Repeat/Request* (ARQ) – Čeká se na ACK, když nepřijde do timeoutu, dojde k znovuzaslání. Princip tzv. klouzavého okna (*sliding window*). Existují 3 strategie.

**Stop and wait**

- Klouzavé okno o velikosti 1.
- Odesílatel pošle paket a čeká na potvrzení. Po přijetí potvrzení pošle další paket.
- Špatná efektivita využití pásma.

**Go back n**

- Buffer na straně odesílatele.
- Příjemce potvrzuje naposledy přijatým paketem (např. příjemce pošle ACK2 znamená, že dostal pakety 0, 1, 2).
- Plýtvání při znovuzaslání (např. odesílatel pošle 5 paketů, ztratí se první, musí poslat znovu všech 5).

**Selective repeat**

- Buffery jsou na obou stranách.
- Efektivní využítí šířky pásma, ale složitější implementace.
- Příjemce potvrzuje:
  - Fast Retransmit – Příjemce posílá potvrzení s naposledy přijatým paketem.
  - Bitová maska – V ACK je zavedeno další políčko (bitová maska), které obsahuje informace o tom, co příjemce přijal a co ne.
  - NACK – Příjemce posílá ACK, pokud balík dostal a NACK, pokud nikoliv.

## 58.5 Řízení zahlcení

Zahlcení je detekováno, co se s tím dá dělat?

- Zvýšit kapacitu sítě,
- Snížit množství provozu.

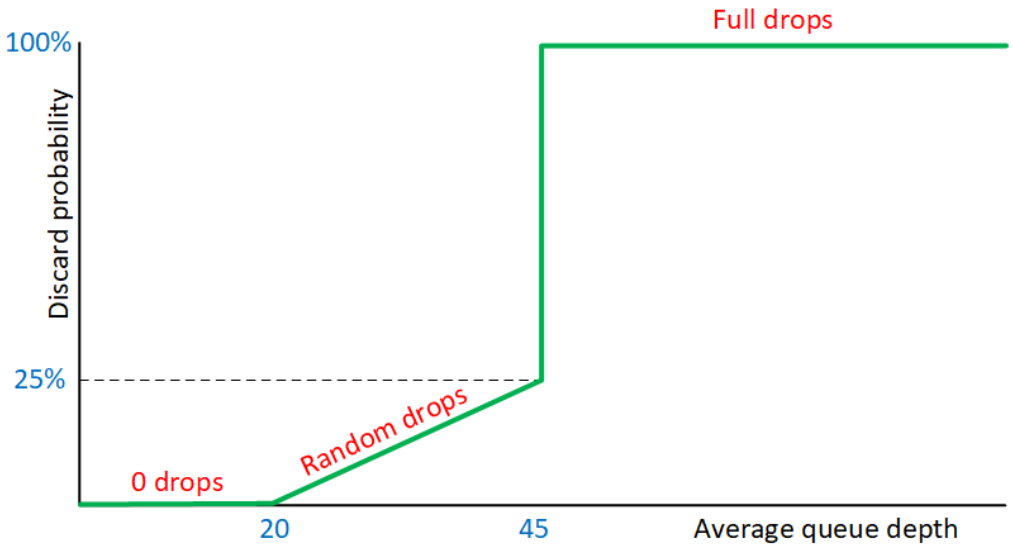
Jak se to dá dělat?

- Opravit zahlcený provoz (*repair*) – Je detekováno zahlcení a řešíme, co budeme dělat.
- Předcházet problému zahlcení (*avoidance*) – Snažíme se zahlcení předcházet (prevence).

### 58.5.1 Zahazování paketů (opravování provozu)

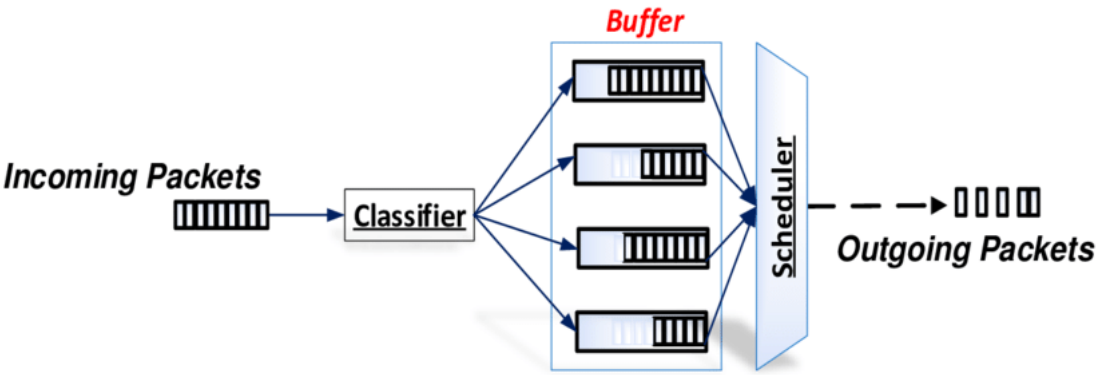
Každý uzel (najčastěji směrovač) v síti kontroluje délku své fronty, pokud příliš narůstá, tak začne preventivně některé pakety zahazovat (to však vede k jejich znovuzaslání).

**RED (*Random Early Detection*)** Algoritmus RED sleduje velikost fronty a zahazuje pakety na základě pravděpodobnosti. Pokud je fronta téměř prázdná, pravděpodobnost zahození je 0. Jak fronta roste, roste i pravděpodobnost zahození příchozího paketu. Když je fronta plná, pravděpodobnost dosáhne hodnoty 1 a všechny příchozí pakety jsou zahazovány.




Obrázek 58.2: Vizualizace RED.

**WRED (*Weighted Random Early Detection*)** Algoritmus WRED je rozšířením RED o přidání váh pro jednotlivé třídy paketů (máme frontu pro každou třídu). WRED nezahazuje všechny pakety se stejnou pravděpodobností, ale rozlišuje jejich důležitost (pomocí hodnot IP precedence nebo DSCP). Některé pakety chceme prioritizovat, např. VoIP (*Voice over Internet Protocol*), RTP (*Real-time Transport Protocol*).



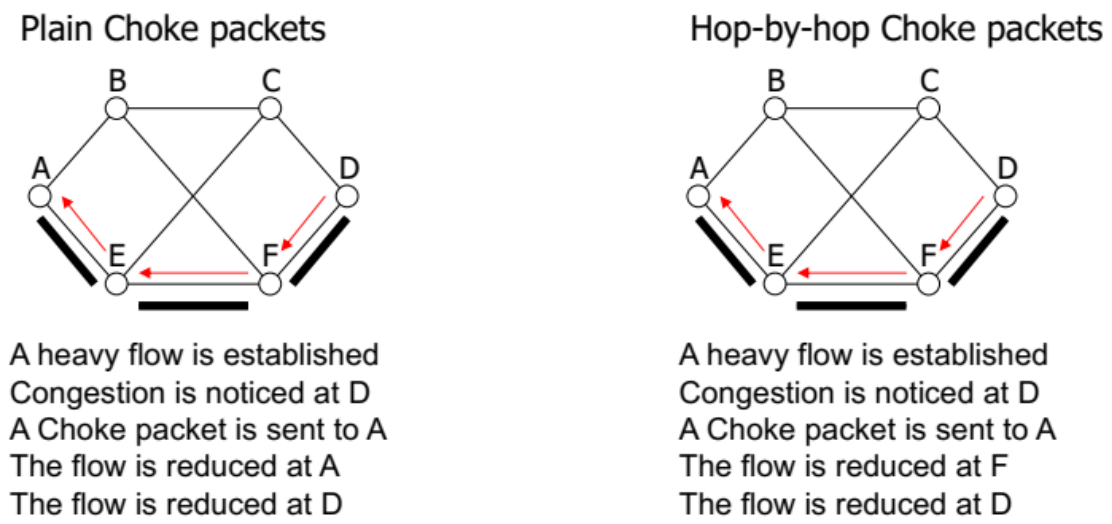

Obrázek 58.3: Vizualizace WRED.

### 58.5.2 Choke pakety (opravování provozu)

Příjemce v síti má problém zpracovávat provoz (nestíhá), tak dá vědět odesílateli (odešle choke paket), aby zpomalil.

**Plain choke packets** Viz obrázek 58.4. Uzel D detekuje zahlcení způsobené provozem od A. Uzel D pošle choke paket A. Proti zahlcení „zabojuje“ pouze odesílatel (A).

**Hop-by-hop choke packets** Viz obrázek 58.4. Uzel D detekuje zahlcení způsobené provozem od A. Uzel D pošle choke paket A. Každý uzel po cestě „bojuje“ proti zahlcení (F, E, A).




Obrázek 58.4: *Plain choke packets* vs. *hop-by-hop choke packets*.

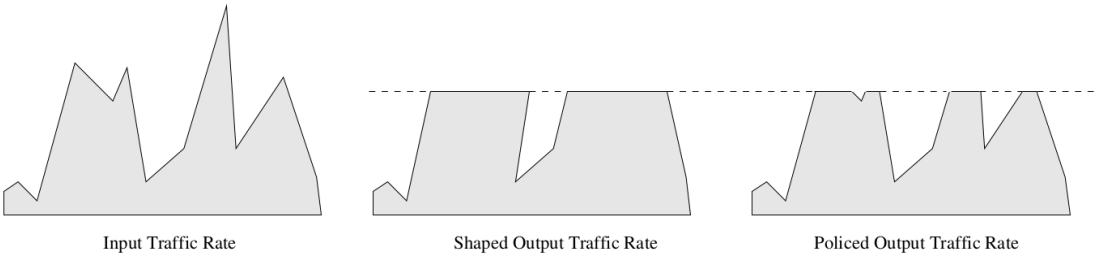
### 58.5.3 Ořezání a rozložení provozu (předcházení zahlcení)

Jedná se o předcházení zahlcení ze strany odesílatele. Odesílatel je „ohleduplný“ a myslí na to, aby příjemce nebyl zahlcen. Odesílatel má pevně stanovenou hranici kolik může maximálně posílat. Implementace pomocí tzv. *token bucket*.

**Ořezání provozu (*policing*)** Cokoliv nad hranici je „oříznuto“ a zahozeno.

**Rozložení provozu (*shaping*)** Cokoliv nad hranici je uloženo do bufferu a odesláno později, až je menší provoz.

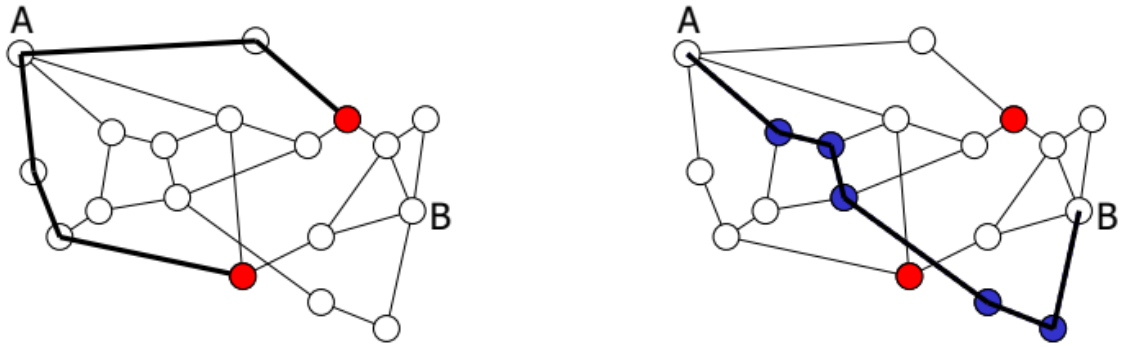
Čím se liší rozložení provozu (policing) a omezení provozu (shaping)?




Obrázek 58.5: Ořezání provozu vs rozložení provozu.

### 58.5.4 Rezervace (předcházení zahlcení)

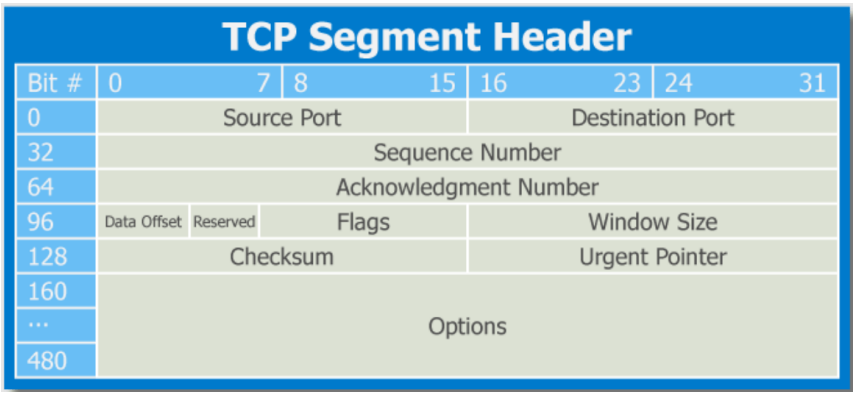
Zařízení v síti (směrovače) jsou schopni se domluvit a vytvořit si tzv. „virtuální okruh“ pro konkrétní komunikaci. Každé zařízení na okruhu ví o provozu a rezervuje potřebné pásmo (předem se deklarují kapacity). Je nutné více signaliace (režie).




Obrázek 58.6: Příklad rezervace.

## 58.6 TCP (*Transmission Control Protocol*)

- Garantuje spolehlivé doručení.
- Garantuje doručení v pořadí.
- Řízení toku a zahlcení.
- *Connection-oriented* – Navázání spojení pomocí *three-way handshake*.
- Pracuje s *byte stream* – nezohledňuje hranice aplikačních dat.
- Sekvenční číslo závisí na tom, kolik bajtů se posílá. Např. $seq = 92, data = 8\,B$, $seq = 100, data = 20\,B$, $seq = 120, data = 13\,B$, ...

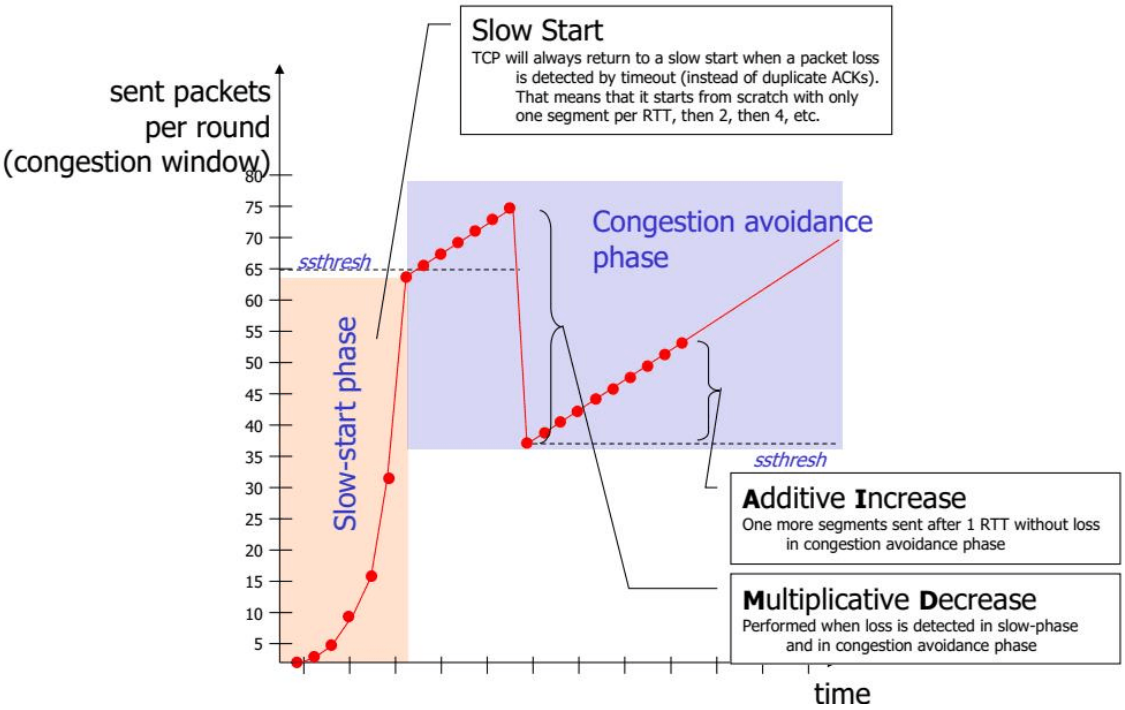



Obrázek 58.7: Hlavička TCP.

**Řízení toku a zahlcení** V TCP jsou rozlišovány 3 fáze přenosu dat.

- *Slow start* – Exponenciální zvyšování rychlosti odesílání až po dosažení nějaké hranice.
- *Congestion avoidance (additive increase)* – Lineární zvyšování počtu odeslaných paketů.
- *Congestion avoidance (multiplicative decrease)* – Skokové snížení rychlosti odesílání. Nastane, pokud je detekována ztráta paketu.

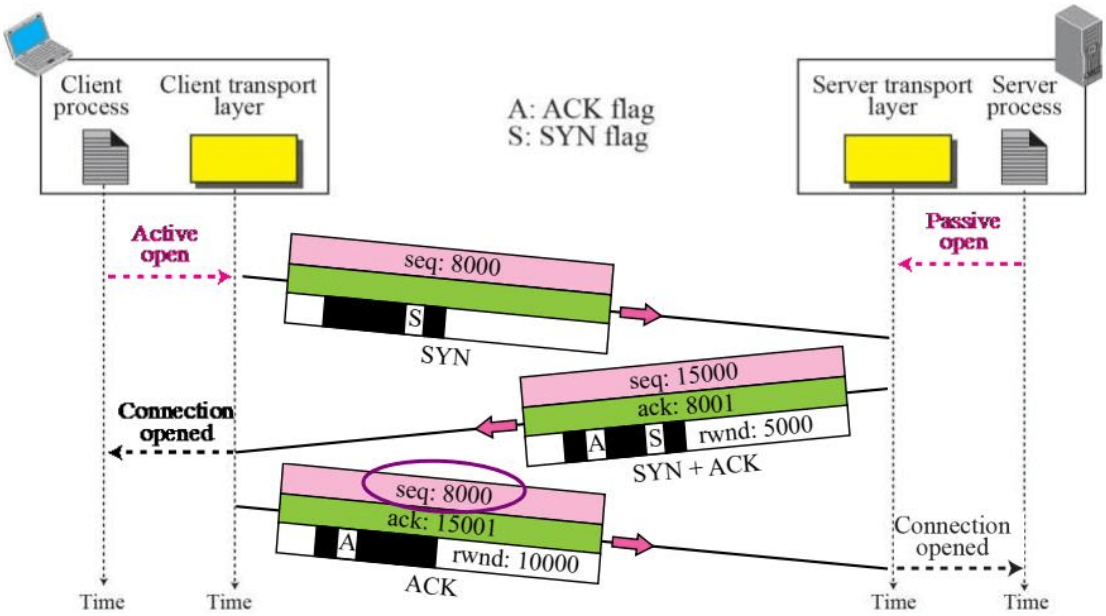
V TCP existuje spoustu algoritmů pro řízení toku a zahlcení, které implementují tento princip (Tahoe, Reno, New Reno, Vegas, CUBIC, Westwood, ...).




Obrázek 58.8: Jednotlivé fáze řízení zahlcení v TCP.

### Nevýhody a problémy

- Nepodporuje *multi homing*.
- *Head of line blocking* – Pokud dojde ke ztrátě paketu, tak vše za ním je pozdrženo
- Velká režie potvrzování (až 35 % veškerého provozu je režie)



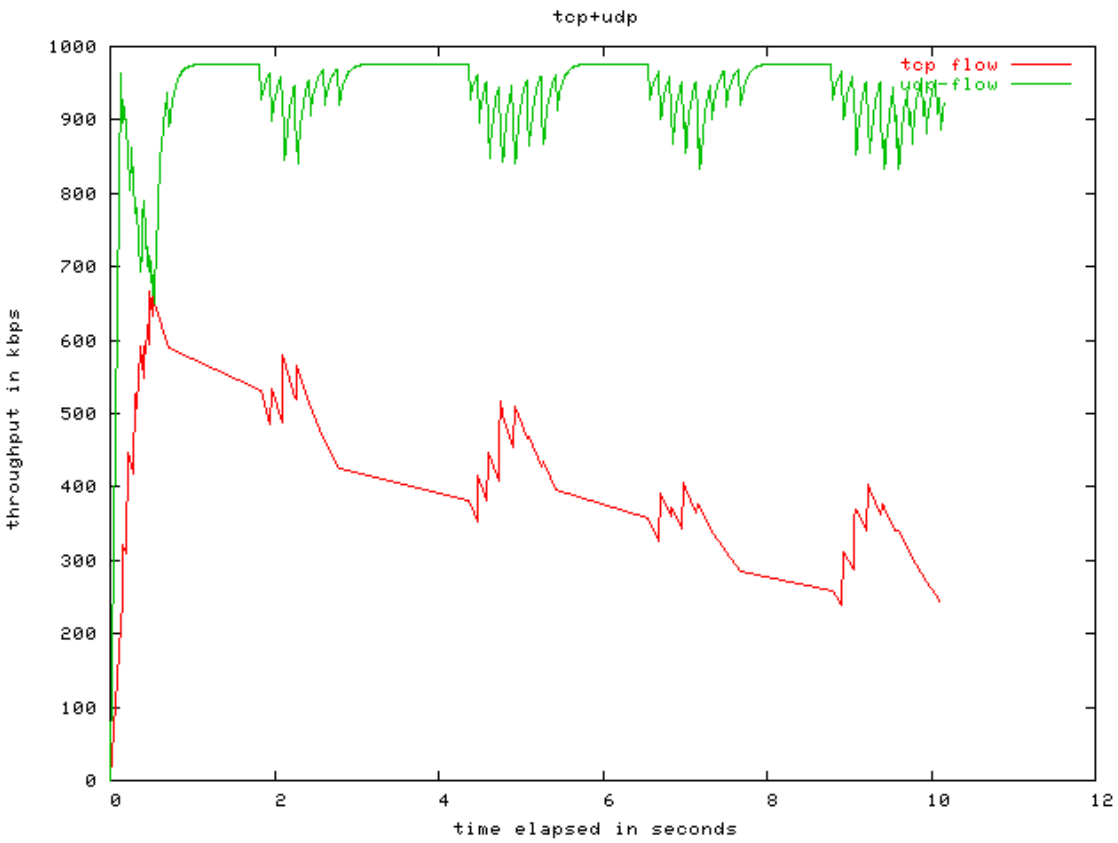

Obrázek 58.9: Zahájení spojení pomocí tzv. *Three-Way Handshake*.

## 58.7 UDP (*User Datagram Protocol*)

- Negarantuje spolehlivé doručení (*best effort*).
- Negarantuje doručení v pořadí.
- *Connection-less*.
- Pracuje s *byte stream* – nezohledňuje hranice aplikačních dat.
- Má minimální režii.

| UDP Datagram Header | | | | |
|---|---|---|---|---|
| Bit # | 0 – 7 | 8 – 15 | 16 – 23 | 24 – 31 |
| 0 | Source Port | | Destination Port | |
| 32 | Length | | Checksum | |

Obrázek 58.10: Hlavička UDP.

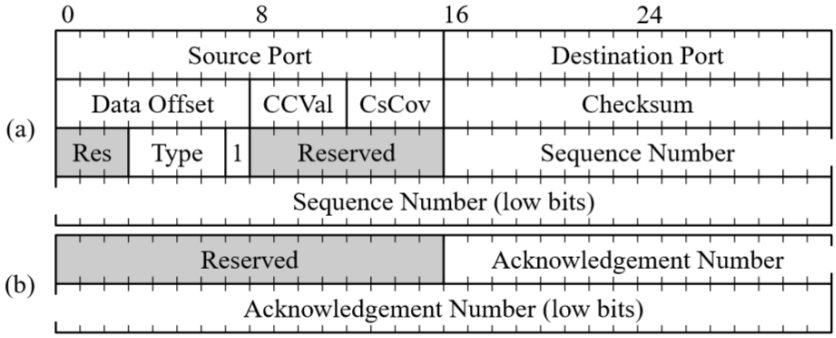



Obrázek 58.11: Problém UDP, kdy si pro sebe zabere celé pásmo, protože TCP se chová „zodpovědně“ a snižuje svůj provoz.

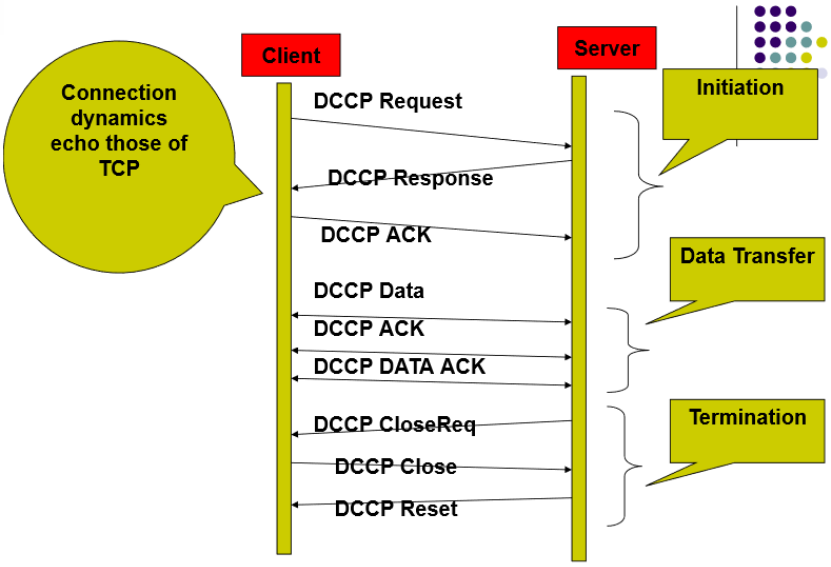
## 58.8 DCCP (Datagram Congestion Control Protocol)

- Cílem je zajistit podporu řízení zahlcení pro UDP.
- Negarantuje spolehlivé doručení (*best effort*).

- Negarantuje doručení v pořadí.
- *Connection-oriented* – Navázání spojení pomocí *three-way handshake*.
- Využití pro audio/video konference.




Obrázek 58.12: Hlavička DCCP. Sekvenční a potvrzovací čísla souvisí pouze s účtováním provozu kvůli řízení zahlcení, nikoliv pro zajištění spolehlivého přenosu.



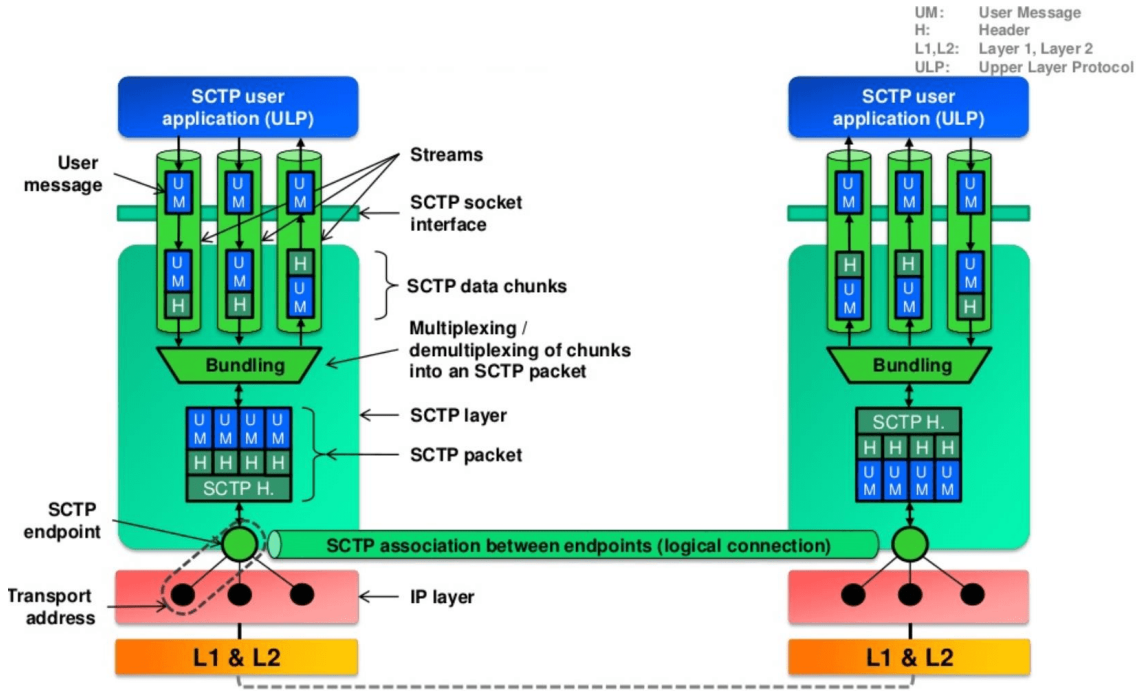

Obrázek 58.13: Komunikace pomocí DCCP. Obsahuje několik typů zpráv: Request, Response, ACK, Data, Data ACK, CloseReq, Close, Reset.

**Jak je zahlcení realizováno** V potvrzeních se nastavují tzv. ECN (*Explicit Congestion Notification*) bity. Pomocí nich dává příjemce odesílateli vědět, jak moc je zahlcen. V průběhu navazování spojení se pomocí CCID (*Congestion Control ID*) dohodne, jaký typ řízení zahlcení se bude používat (algoritmy jako pro TCP, nebo vlastní).

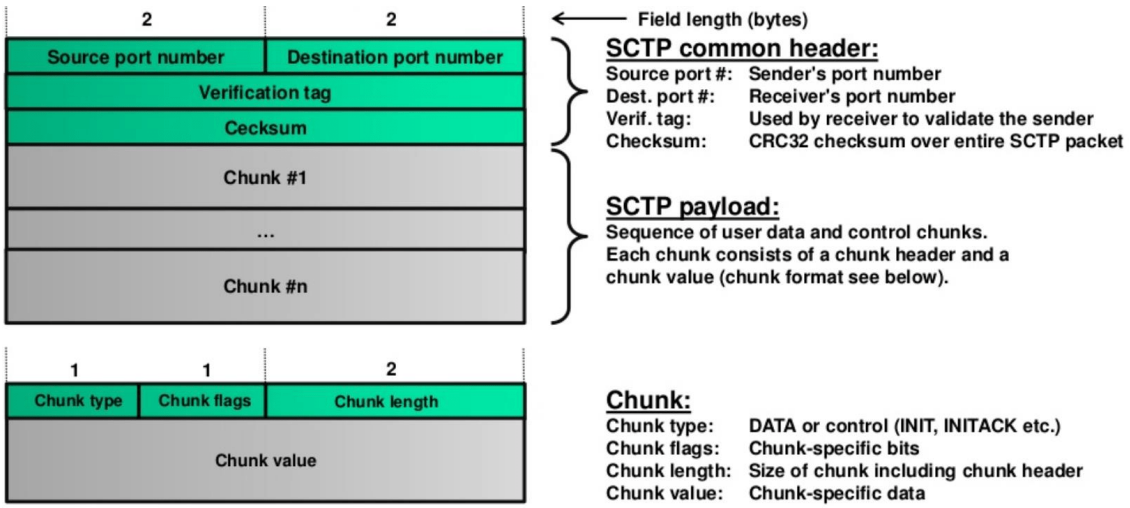
## 58.9 SCTP (*Stream Control Transmission Protocol*)

- Garantuje spolehlivé doručení.
- Garantuje doručení v pořadí.

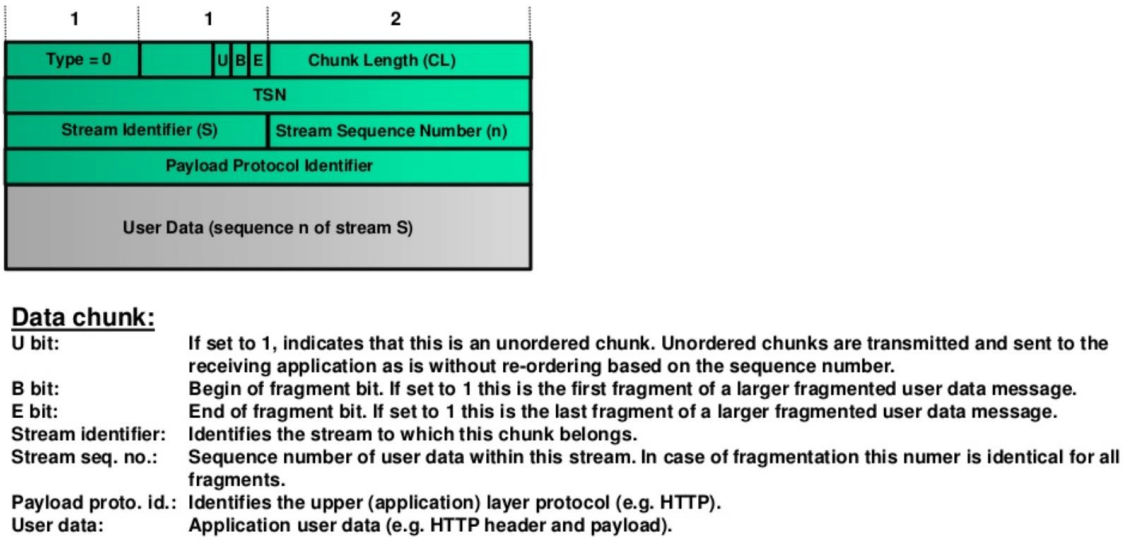
- Pracuje s *message stream* – Respektuje hranice aplikačních dat.
- *Connection-oriented* – Navázání spojení pomocí *four-way handshake*.
- *Path MTU discovery* – Obrana proti fragmentaci (od zdroje k cíli jsou prozkoumány MTU všech uzlů).
- Netrpí problémem *head of line blocking*.
- Podporuje *multi homing*.
- Řízení zahlceni.
- Aplikace je třeba přeprogramovat, aby místo TCP socketů využívali SCTP sockety.



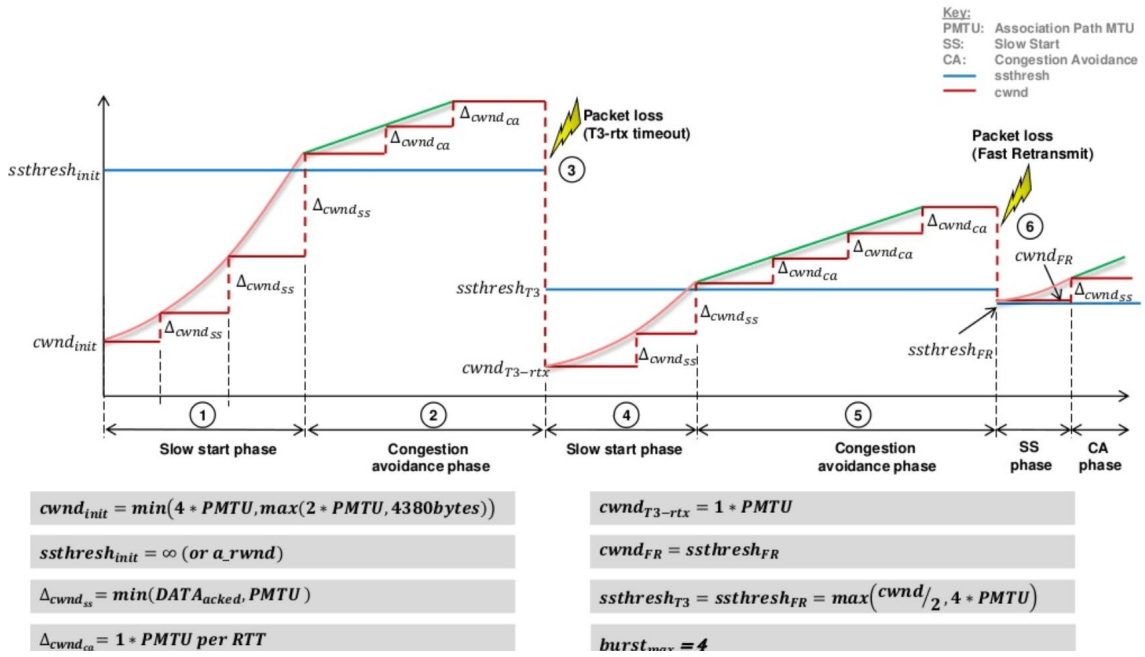

Obrázek 58.14: Komunikace pomocí SCTP. Komunikace s aplikací probíhá přes sockety. Každé aplikační zprávě je přidán základní hlavičkou – vzniká *data chunk*. Několik data chunku je zabaleno dohromady a opatřeno SCTP hlavičkou – vzniká SCTP paket. Ten je poté možno posílat přes více IP spojení.




Obrázek 58.15: Hlavička SCTP. SCTP paket a *data chunk*.



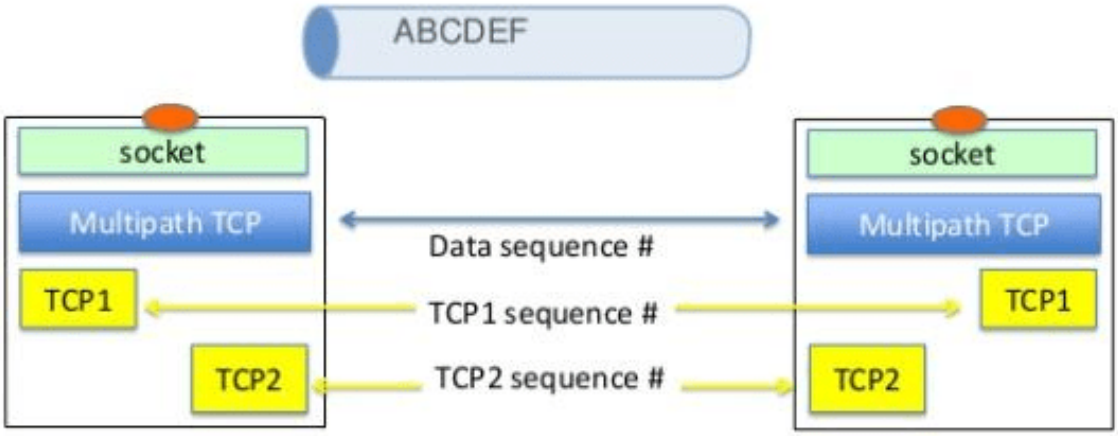

Obrázek 58.16: Detailní hlavička *data chunk*. Identifikátor streamu a identifikátor paketu v rámci streamu.



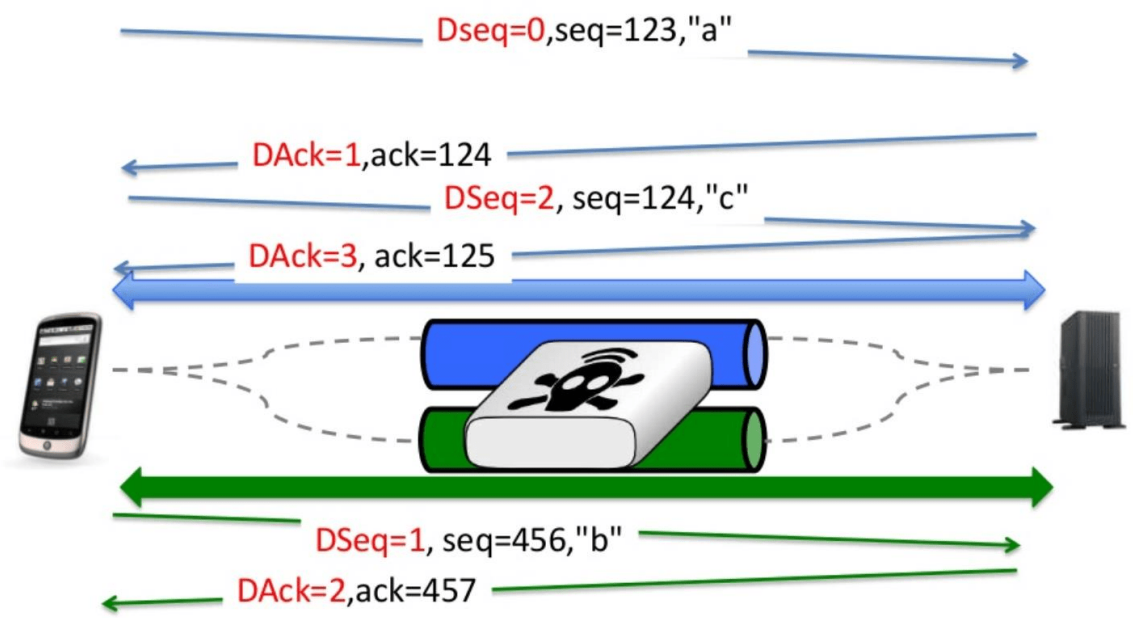

Obrázek 58.17: Řízení zahlcení (*slow start*, *congestion avoidance (additive increase)*, *congestion avoidance (multiplicative decrease)*) je podobné jako TCP Westwood.

## 58.10 MP-TCP (*Multipath TCP*)

- Cílem je vzít co nejvíce vlastností z SCTP a poskytnout je v TCP bez toho, aniž by se musely přeprogramovávat aplikace (má obdobné sockety jako TCP).
- Má menší signalizační náročnost oproti SCTP.
- Chová se jako TCP pouze, podporuje *multi homing* a má vlastní řízení zahlcení.
- Rozšíření TCP hlavičky o tzv. *options*.



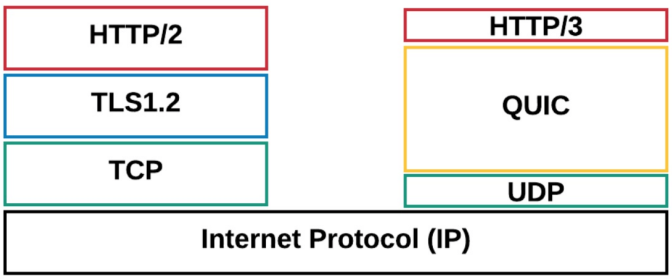

Obrázek 58.18: MPTCP komunikace, rozdělení aplikačních dat na paralelní TCP streamy. Rozlišujeme datová sekvenční čísla a sekvenční čísla TCP.



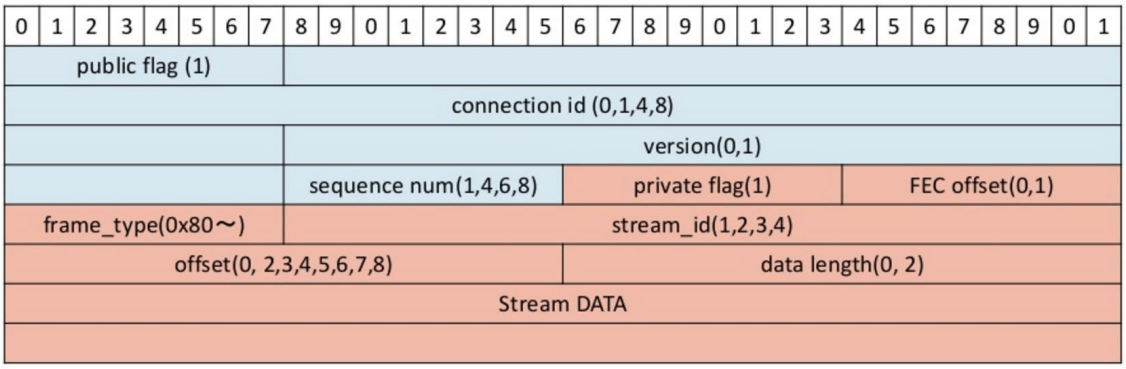

Obrázek 58.19: MPTCP řízení toku. Dvě TCP komunikace a posílání řetězce „abc“. Navázání spojení podobné jako v TCP. Může vypadnout i celý komunikační kanál.

## 58.11 QUIC (*Quick UDP Internet Connections*)

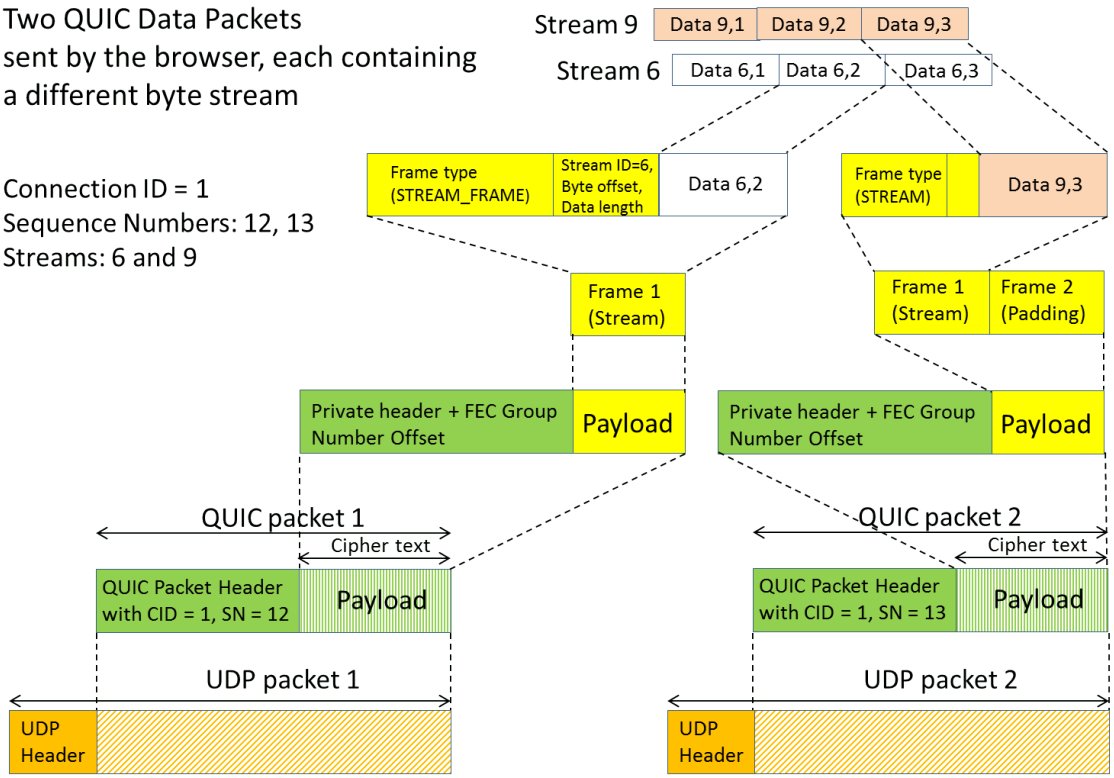
- Motivace pro QUIC: dnes většina provozu probíhá šifrovaně přes HTTPS. Navázání bezpečného HTTPS spojení má velkou režii (HTTPS = TCP + SSL + HTTP). Až $3 \times$ RTT přes započetím komunikace!
  - TCP: syn, syn ack, ack
  - TLS: client hello, server hello, change cipher spec, encrypted handshake message
- Staví nad UDP (aby všechny zařízení v internetu nemuseli implementovat nový transportní protokol).
- Paralelní datové streamy, aby bylo zabréněno *head of line blocking*.
- Má zabudovanou *Forward Error Correction* (FEC), přidává bitovou redundanci, tím umožňuje velké množství ztracených paketů dopočítat. Pokud to není možný, tak „chytrý“ selective repeat.




Obrázek 58.20: Standardní TCP stack vs. QUIC.




Obrázek 58.21: Hlavička QUIC. Spousta políček má proměnlivou velikost.




Obrázek 58.22: Složení QUIC zprávy.

# Kapitola 59

# PDS – Metody detekce síťových incidentů (signatury, statistické metody) a nástroje (IDS/IPS).

## 59.1 Zdroje

- `07-ids.pdf`
- `PDS_2021-03-26.mp4`
- `03-IDS.pdf`
- `NSB_2021-02-23.mp4`

## 59.2 Úvod a kontext

**Identifikace a monitorování síťového provozu**

- Co je cílem?
  - Identifikace protokolu (na různých vrstvách).
  - Identifikace aplikace.
  - Pokud je identifikován nedovolený provoz, tak kontaktovat správce.
- Proč to dělat?
  - Bezpečnost – Detekce útoků a nedovoleného chování.
  - Diagnostika – Správné chování systémových i uživatelských aplikací.
  - Rozlišení služeb – Nastavení kvality služeb, prioritizace kritických přenosů.
  - Vytížení sítě – Sledování rozložení sítě, vytížení síťových prvků a služeb.
  - Účtování služeb – Identifikace provozu VoIP, IPTV, apod.

**Síťové tunelování** Technika, která pro přenos síťového spojení používá jiné síťové spojení (Zapouzdření komunikace do jiného protokolu.). Umožňuje: přenášet data přes nekompatibilní sítě; obcházet administrativní omezení určité sítě; poskytovat zabezpečenou komunikaci přes nezabezpečenou, resp. nedůvěryhodnou síť.

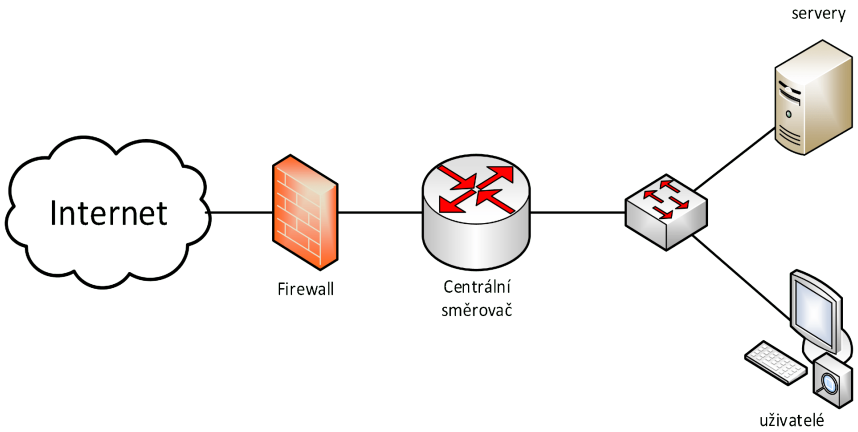
## 59.3 Nástroje IDS/IPS

Umístěno někde v síti, aby se zabezpečili všechny zařízení najednou. Specializované zařízení, např. IDS/IPS sonda nebo jako součást hraničního prvku.

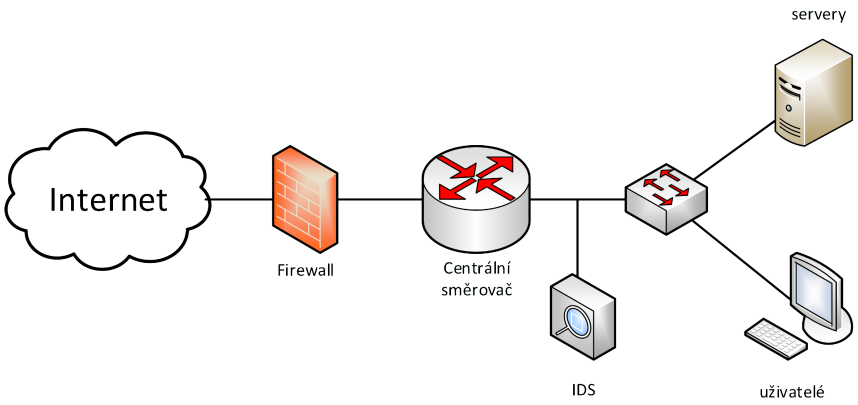
**Intrusion Detection System (IDS)** „Pasivní obraný mechanismus“. Cílem je identifikovat provoz a v případě, že se jedná nedovolený provoz, tak informovat správce. Ten rozhodne, jaké budou kroky (zda se zablokuje, nebo zda se jedná o falešnou pozitivitu).

**Intrusion Prevention System (IPS)** „Aktivní obraný mechanismus“. Cílem je detekovat provoz a v případě, že se jedná nedovolený provoz, tak ho odfiltrovat/zablokovat/zahodit. Nebezpečné v případě falešné pozitivity. Můžeme odříznout provoz, který považujeme za nedovolený, ale ve skutečnosti se jedná o provoz, který jsme nečekali. Většinou se v praxi volí IDS.

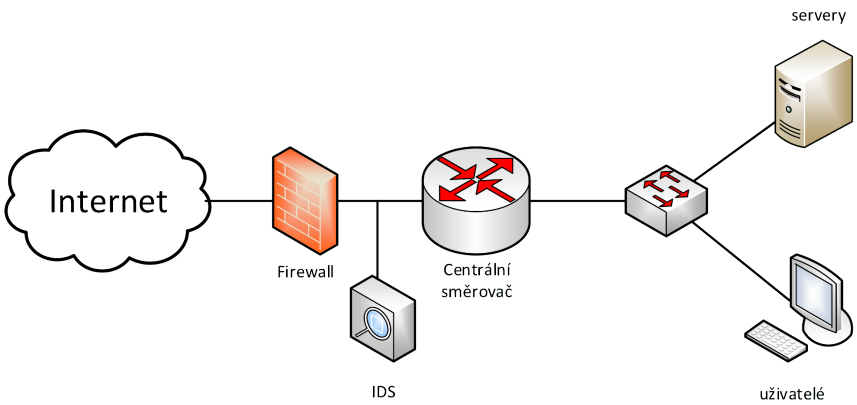
**Síťový incident** Identifikace nedovoleného provozu.




Obrázek 59.1: Způsob zapojení 0. Host based IDS/IPS. Systém nainstalovaný na koncové stanice a reportující informace. „Lepší firewall, lepší antivirus“.




Obrázek 59.2: Způsob zapojení do vniřní sítě. Network based IDS/IPS.




Obrázek 59.3: Způsob zapojení před vnitřní síť. Network based IDS/IPS. Problém s NATem – nerozlišíme jestli komunikace jde na jeden uzel v síti nebo na více. Problém šifrování.

## 59.4 Identifikace provozu pomocí hodnot z hlaviček paketů

- Využití informací z hlaviček jednotlivých protokolů TCP/IP. Jaké informace lze využít?
  - Na linkové vrstvě (L2):
    - Field `EtherType` – IPv4, IPv6, ARP, RARP, autentizace 802.1x, ...
  - Na síťové vrstvě (L3):
    - Field `protocol` – IPv6 nad IPv4, ICMP, IGMP, transportní protokoly (UDP, TCP, SCTP, ...), směrovací protokoly (EIGRP, ESPF, ...), multicast (PIM), ...
  - Na transportní vrstvě (L4):
    - Field `SourcePort`, `DestinationPort`, rozpoznání služby na základě portu – FTP, SSH, Telnet, SMTP, DNS, DHCP, HTTP, POP3, NTP, IMAP, SNMP, LDAP, HTTPS, ...
  - Na aplikační vrstvě vrstvě (L7):
    - Specifické pro aplikace (např: pro HTTP rozpoznávání jednotlivých metod).
- Rychlé, dobře implementovatelné a univerzální řešení. Má své limity:
  - Aplikace nemusí být pevně svázána s portem (dynamické porty).
  - Tunelování, šifrování (spoustu věcí je zapouzdřováno do SSL/TLS), VPN.
  - IPv6 nad IPv4.
  - Skryté kanály – Tunelování skrze protokoly, kde se neočekává nic nelegitimního, běžně se nefiltrují (ICMP, DNS).
- Na tomto konceptu fungují firewally.

## 59.5 Identifikace provozu dle signatury

- Signatura (pattern, signature) je statický řetězec tvořený posloupností znaků z hlavičky či těla protokolu, který slouží k rozpoznání daného protokolu.
  - Je to „reprezentující“ podřetězec paketu komunikace (paket jako řetězec bytů – ASCII).
  - Určujeme až v aplikačních datech (TCP/UDP payload).
- Signatura ale může mít různou podobu:
  - Řetězec, který se vyskytuje na začátku, resp. na nějakém fixním offsetu payloadu.
  - Řetězec, který se vyskytuje na různém místě v payloadu.
  - Posloupnost podřetězců, která se vyskytuje v payloadu.
- Pokud se snažíme identifikovat provoz na základě signatury, tak typicky signaturu nehledáme v celém paketu (příliš náročné), ale pouze v prvních X bytech payloadu.
- Signatura v paketu vs. toku
  - Jednoduchá možnost – Signatura pouze na základě informací v paketu.
  - Složitější možnost – Signatury napříč pakety (v datovém toku).
    - Příklad: signaturu pro identifikaci TLS provozu, by mohla mít podobu dvou řetězců, které se vyskytují každý v jiném paketu (1 – „client hello“, 2 – „server hello“).
- S novou verzí protokolu, je většinou třeba aktualizovat signaturu, protože se změnila struktura komunikace.
- Lze vytvořit manuálně na základě specifikace protokolu a nebo automatizovat (*Longest common subsequence problem*).
- Použití: Někde poslouchám síťový provoz a mám databázi signatur. Snažím se najít v toku signaturu z databáze a tím identifikovat aplikaci.

```
# First thing that happens is that the client sends NICK and USER, in
# either order. This allows MIRC color codes (\x02-\x0d instead of
# \x09-\x0d).
^(nick[\x09-\x0d -~]*user[\x09-\x0d -~]*:|user[\x09-\x0d -~]*:[\x02-\x0d -~]
*nick[\x09-\x0d -~]*\x0d\x0a)
```

Výpis 59.1: Příklad signatury pro detekci IRC. Pomocí regulárního výrazu je specifikována kostra řetězce.

```
^(\x13bittorrent protocol|azver\x01$|get /scrape\?info_hash=get
/announce\?info_hash=|get /client/bitcomet/|GET /data\?fid=)
|d1:ad2:id20:|\x08’7P\)[RP]
```

Výpis 59.2: Příklad signatury pro detekci BitTorrentu. Signatura je založená na klíčových slovech.

```
// SSL Hello with certificate, Client Hello
^(.?.?\x16\x03.*\x16\x03|.?.?\x01\x03\x01?.*\x0b)
```

Výpis 59.3: Příklad signatury pro detekci SSL provozu. Detekce Client Hello paketu, kdy se vyměňují informace o šifrování.

### Automatizace vytváření signatur

- Algoritmus LCS (Longest Common Subsequence).
- Algoritmus hledá „nejbližší“ společný řětězec v toku (algoritmus definuje metriku blízkosti).
- Proces vytvoření signatury:
  - Odchytím si typickou komunikaci aplikace, kterou chci identifikovat v provozu – Několik toků, které jsou dostatečně reprezentativní.
  - Z každého toku vezmeme X prvotních paketů (např. 10 stačí).
  - Poté iterativně hledám „nejbližší“ společný podřetězec vždy pro dvojici toků (kandidát).
  - Cyklus končí, až se kandidát přestane měnit. Výsledek je signatura (minimální možná délka, 2 znaky jsou málo).

## 59.6 Identifikace provozu dle statistického chování

- Vytvoření statistického modelu protokolu na základě trénovacích dat.
  - Popis chování na základě metadat o provozu (odezva, velikost paketu, počet paketů v daném směru, ...).
  - Zkoumáme prvních $n$ bytů toku, které jsou spíše specifické pro protokol (navazování spojení).
  - Na rozdíl od signatur se nezkoumá obsah hlaviček a těla protokolu, ale pouze statistické vlastnosti jeho chování.
- 
- Klasifikace neznámého protokolu: Vybereme model s nejlepší vzdáleností od neznámého protokolu.
- Vytváření se tzv. **otisk protokolu** (*fingerprint*) – soubor vlastností (chování) protokolu, které lze využít k jednoznačné identifikaci protokolu.

### Metoda jednoduchých statistických otisků

- Trénovací data: mějme $n$ síťových toků. Z prvních $m$ paketů každého toku se extrahuje:
  - velikost paketu;
  - časový odstup paketů (závisí na tom co dělá klient a co dělá server);
  - pořadí paketů v toku.
- Učení:

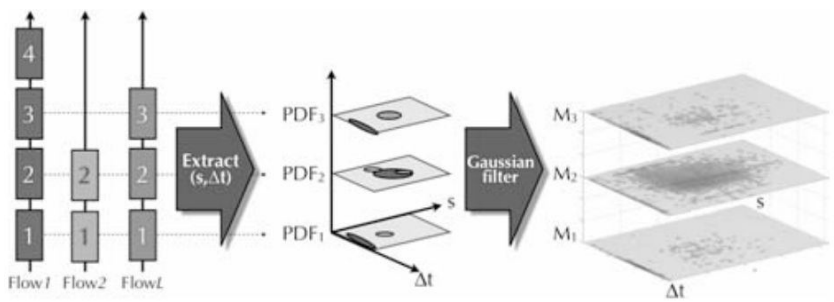
  - Z trénovacích dat se modeluje statistické chování každého paketu v toku. Např. velikost prvního paketu odpovídá tomuhle pravděpodobnostnímu rozdělení, časový odstup druhého paketu od prvnímu odpovídá jinému pravděpodobnostnímu rozdělení. (Typicky normální rozdělení).
  - Poté se aplikují gaussovské filtry pro odstranění šumu.
  - Vzniká tzv. „maska“ a je určena nějaká prahová hodnota (na základě *anomaly score*, která se spočítá z vektoru anomálií).
- Model: protokol je reprezentován otiskem (*fingerprint*) – „maska“, prahová hodnota
- Klasifikace:
  - Máme neznámý tok, který chceme klasifikovat.
  - Sleduju velikosti paketů, odstupy paketů a na základě toho spočítám *anomaly score* pro tok.
  - Získané *anomaly score* porovnám s prahovou hodnotou.

Příklad:

- Tok $F = (P_1, P_2, \dots, P_n)$
- paket $P_i = (S_i, \Delta t_i)$
  $S_i$ - velikost paketu
  $\Delta t_i$ - časový odstup od $P_{i-1}$

- Je sestaven model hustoty pravděpodobnosti

- Trénování:




extrakce

rozložení psti pro jednotlivé pakety

"maska" (~ příznaky)

Obrázek 59.4: Princip metody jednoduchých statistických otisků – fáze učení.

### Statistical Protocol IDentification (SPID)

- Kombinuje statistické chování s atributy.
- Statistické atributy: směr paketů, pořadí paketů, velikost paketů.
- Aplikační atributy: četnost konkrétních bytů, hodnota off-setu.
- Model protokolu používá pro klasifikaci pravděpodobnostní vektory atributů.

- Pro porovnání měří odlišnost dvou pravděpodobnostních rozložení.
- Příklad jaké atributy přenosu lze použít pro rozlišení aplikačního protokolu.
  - Bytová frekvence prvního paketu TCP v každém směru (myšlenka: obsah je typický, pouze pro začátek toho toku – navázání spojení).
  - Počet změn komunikace (klient → server, server → klient).
  - Počet přenesených bytů v daném směru (RTP – balanced, HTTP – download, SMTP – upload).

## 59.7 Detekce anomálie

- Protokol jsme již identifikovali a nyní sledujeme jeho chování.
- Anomálie v provozu – odchylka komunikace od standardního (typického) provozu.
- Jaké anomálie mohou nastat:
  - Neočekávaný nárůst provozu.
  - Snížení provozu, chybějící linky – Výpadek linky.
  - Změna struktury provozu – Jeden uzel se snaží komunikovat se všemi v síti (sledování, detekce malwaru).
- Vytvoření modelu:
  - Trénovací data: typická komunikace.
  - Natrénování modelu.
  - Metrika pro určení odchylky od natrénovaného modelu (např. když se přesáhne práhová hodnota, vyvolá se upozornění).
- Jaké modely lze využít:
  - statistické modely,
  - shlukování (k-means, k-medoids, k-nn clusters, ...),
  - klasifikační modely (bayesovské modely, SVM, neuronové sítě, ...)
  - pravděpodobnostní automaty, markovovy řetězce.
- Sbět trénovacích dat:
  - Netflow – informace o tocích (jaké IP adresy, jaké porty, jak dlouho tok trval, kolik dat se přeneslo, ...).
  - SNMP – informace o provozu (počet přenesených paketů, bytů, ...).
  - Úplné zachytávání paketů.
- Systém může fungovat pouze tehdy, pokud jsme schopni sestavit model běžného chování.

# Kapitola 60

# PDS – Sítě Peer-to-Peer: vlastnosti, chování, způsoby směrování. Strukturované a nestrukturované sítě.

## 60.1 Zdroje

- `08-p2p.pdf`
- `PDS_2021-04-09.mp4`

## 60.2 Architektura peer-to-peer sítí

- Peer-to-Peer (P2P) je alternativní architektura vůči client-server.
- Uzly (uživatelé) spolu komunikují napřímo. Každý uzel v síti má stejnou roli. Není zde žádný centrální bod, žádný uzel není nadřazený ostatním[1].
- Jiný způsob adresování – adresování obsahem.
- Jiný způsob směrování – lokální rozhodování, specifická struktura sítí.
- Příklad: BitTorrent, Napster, Gnutella, Skype (dříve), Bitcoin, Bluetooth, instant messaging služby, ...

**Logická síť** Základem každé P2P sítě je tzv. logická síť (*overlay*), která je vystavěna na aplikační vrstvě TCP/IP. Tedy P2P sítě staví nad exitující síťovou infrastrukturou. Logická síť definuje způsob propojení uzlů, směrování, vyhledávání informací, ...

**Definice P2P sítě** Dynamický soubor nezávislých uzlů (peers), které jsou propojeny a jejichž zdroje (objekty) jsou k dispozici ostatním uzlům v této síti. Zdroje: výpočetní výkon, disková kapacita (soubory), zařízení (tiskárny). Sdílené zdroje jsou přímo přístupné všem uzlům, ty je nabízejí a zároveň využívají. Síť obsahuje prostředky pro připojení uzlu k síti, hledávání a využití zdrojů, ....

**Typy P2P sítí**

- Pravé (pure) – Odebrání libovolného uzlu ze sítě nemá vliv na ztrátu schopnosti sítě poskytovat služby (např. Bitcoin).

[1] Jsou výjimky.

- Hybridní – Pro svou činnost využívají centrální uzel pro poskytování části nabízených síťových služeb. Centrální bod slouží k autentizaci, indexování, inicializaci uzlu, apod.

**Vlastnosti P2P sítí**

- Samo-organizovatelnost:
  - Když se uzel připojí nebo odpojí, tak se síť přeskupí a funguje dále.
  - Decentralizovaná topologie, kde uzly spolupracují na jejím vytvoření a udržování.
  - Každý uzel je zodpovědný za svůj lokální stav a část informací (zdrojů).
  - Uzly mají částečný pohled na topologii sítě (znají své nejbližší sousedy).
- Autonomní chování (samořiditelnost):
  - Uzly se chovají dle svého nejlepšího rozhodování (uzel pouze konzumuje zdroje, ale nechce poskytovat).
  - Rozhodování je lokální a nepredikovatelné ⇒ má vliv na topologii sítě, směrování, rozmístění objektů.
  - Uzly se mohou chovat zlomyslně.
  - Problém s ověřování identity uzlů a důvěryhodností (decentralizované řízení).
- Spolehlivost:
  - Spolehlivost sítě roste s redundancí uzlů a informací.
  - Redundance objektů, kopie jsou umístěny ve více uzlech.
- Životnost uzlu:
  - Doba životnosti uzlu je neodhadnutelná ⇒ problém s garancí služby.
  - Závisí na subjektivním lokální rozhodnutí.

| | Klient – server | Peer-to-Peer | *Výhody/nevýhody P2P* |
|---|---|---|---|
| **Směr provozu** | Asymetrický | Symetrický | *vs. xDSL, kabelový modem* |
| **Topologie sítě** | Stabilní | Dynamická | *Problém spolehlivosti* |
| **Robustnost** | Centrální bod | Distribuce zdrojů | *Kritický počet účastníků* |
| **Rozšiřitelnost** | Náročné | Součást návrhu | *Neomezený růst sítě* |
| **Bezpečnost** | Velký důraz | Problematické | *Chybí odpovědná autorita* |
| **Správa a řízení** | Centralizovaný model | Každý uživatel spravuje vlastní uzel | *Samo-organizovaná síť* |
| **Poskytované zdroje** | Omezené možnosti | Dynamicky rostoucí počet zdrojů | *Sdílení výpočetní prostoru, paměti, apod.* |
| **Kvalita služeb** | Garantovaná | Nelze zajistit | *Dynamicky se měnící* |

Obrázek 60.1: Srovnání vlastností P2P a klient-server architektur.

**Referenční model P2P**

- Mějme množinu uzlů $P$, množinu zdrojů $R$ a množinu identifikátorů $I$.
- Struktura logické sítě:
  - mapování zdrojů: $F_R : R \to I$,
  - mapování uzlů: $F_P : P \to I$.
  - Množina uzlů $P$ zpřístupňuje zdroje $R$ v rámci jmenného prostoru $I$ pomocí $F_R$ a $F_P$.
- Decentralizovaná správa jmenného prostoru:
  - $M : I \to 2^P$
  - Příklad: uzel potřebuje zjistit, kdo má konkrétní zdroj.
- Metrika blízkosti:
  - $d : I \times I \to \mathbb{R}$
  - Příklad: uzel chce konkrétní zdroj, vyhledá všechny uzly, které ho poskytují a na základě metriky blízkosti vybere nejbližšího.

**Geometrie sítě a směrování**

- Geometrie (topologie).
  - Dynamická, uzly se připojují a odpojují.
  - Množina uzlů $P$ zpřístupňuje zdroje $R$ v rámci jmenného prostoru $I$ pomocí $F_R$ a $F_P$.
- Směrování.
  - Každý uzel zná své sousedy (uzly a hrany k nim) pomocí relace sousedství: $N : P \to 2^P$ (uzel $\to$ jeho sousedi).
  - Mějme předávání zprávy $route(p, m, i)$, kde hledáme cestu pro zprávu $m$, k uzlu $p$, který spravuje zdroj $i$. Směrování: kterému sousedovi mám zprávu předat?
  - Distribuovaný proces nalezení cesty v síti P2P na základě lokálních znalostí.
  - Směrovací funkce: $R : P \times I \to 2^P$.
  - Směrovací tabulka: každý uzel má svoji, kterou si postupně plní a aktualizuje.

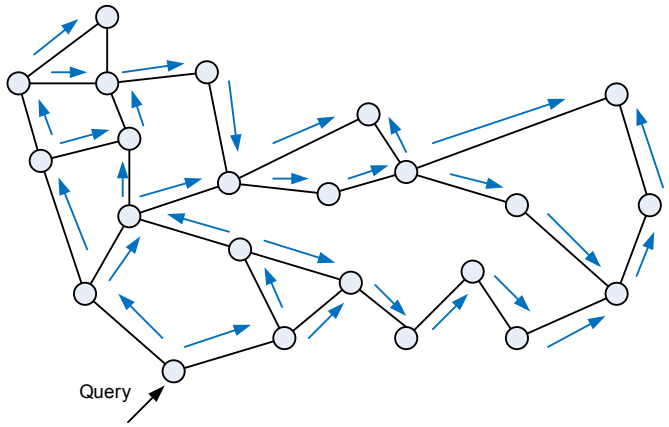
## 60.3 Nestrukturované sítě

- Neexistuje struktura uložení informace (zdroj (objekt) je umístěn na náhodném uzlu).
- Uzel si vyměňuje zprávy se svými sousedy (dotaz na vyhledávání konkrétního objektu).
- Když uzel hledá objekt, neví, jestli se přibližuje, dostává odpovědi pouze ano-ne.
- Jak v takovém systému směrovat (jak najít objekt skrze identifikátor)?

### 60.3.1 Záplava (*flooding*)

- Uzel pošle dotaz na objekt všem svým sousedům.

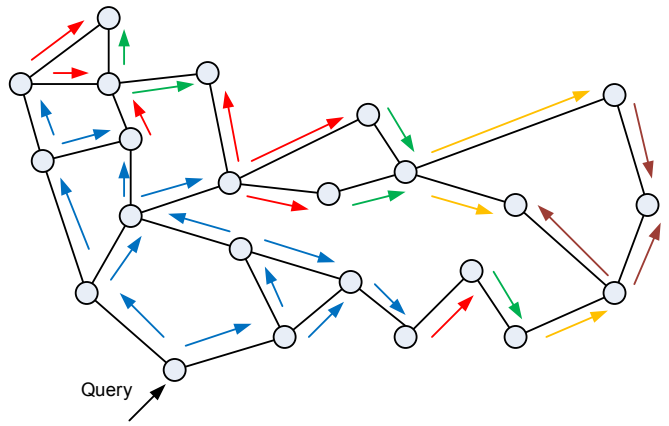
  - Pokud soused má objekt, pošle zpět odpověď.
  - Pokud nemá, pošle zprávu svým sousedům.
- Lze omezit pomocí TTL ve zprávě (zamezuje zacyklení a zahlcení sítě).




Obrázek 60.2: Záplava.

### 60.3.2 Rozšiřující se kruh (*expanding ring*)

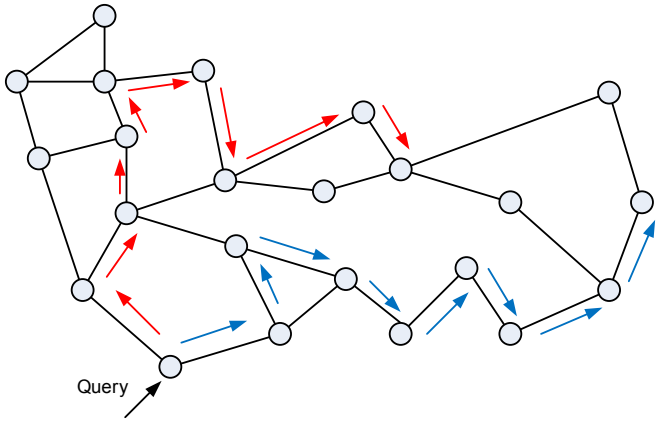
- Uzel pošle dotaz na objekt všem svým sousedům s malým TTL.
  - Pokud objekt je nalezen, hledání končí.
  - Pokud objekt není nalezen, zvýší se TTL a dotaz se pošle znovu.
- Redukuje počet zpráv v síti.




Obrázek 60.3: Rozšiřující se kruh.

### 60.3.3 Náhodný průchod (*random walk*)

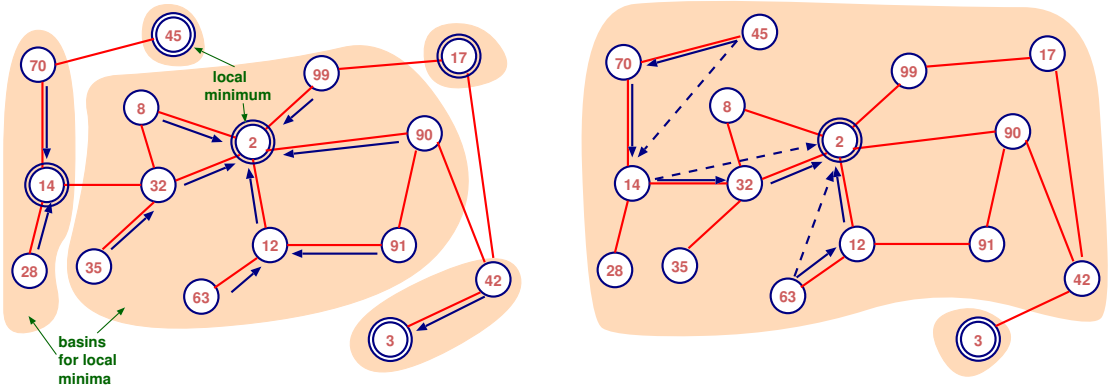
- Uzel pošle dotaz na objekt náhodně vybraným sousedům (i více zároveň).




Obrázek 60.4: Náhodný průchod.

### 60.3.4 Hledání lokálního minima (*local minimum search*)

- Není pro čistě nestrukturované sítě (přidává lokální vzdálenost).
- Máme množinu uzlů identifikovaných hodnotou $x$ a množinu objektů s identifikátorem $w$. Úkolem je umístit objekty do sítě uzlů tak, abychom je mohli najít rychle a spolehlivě, tj. jméno uzlu $x$ by mělo být co nejblíže jménu ukládaného objektu $w$.
- Při hledání používáme metriku vzdálenosti uzlu $x$ od objektu $w$ – $d(x, w)$.
- Uzel $u$ je lokálním minimem pro objekt, pokud je jeho ID nejbližší k ID objektu mezi jeho sousedy do vzdálenosti $h$ kroků.




(a) 1-hop Neighborhoods

(b) 2-hop Neighborhoods

Obrázek 60.5: Příklad hledání lokálního minima. Uzly jsou značeny vzdáleností od objektu. 2 – *hop* – uzel zná sousedy sousedů a zapojuje je do hledání.

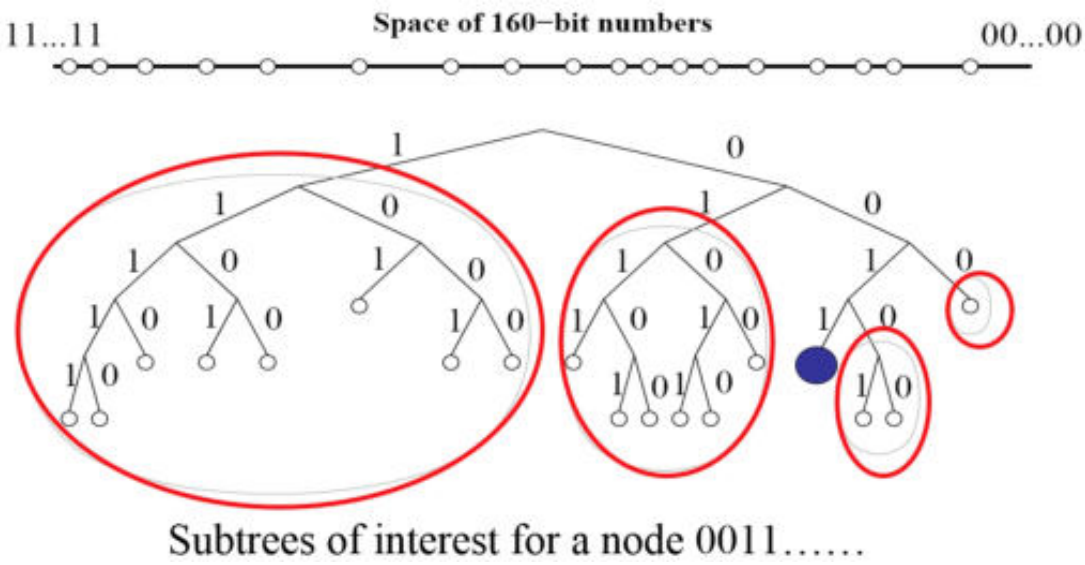
## 60.4 Strukturované sítě

- Kombinují geometrické struktury a směrování.
- Distribuované směrovací algoritmy (metriky: shoda prefixu, euklidovská či lineární vzdálenost, XOR, ...).

- Struktura sítě odpovídá uložení zdrojů.
- Jak směrovat?

### 60.4.1 Kademlia – Distribuovaná hašovací tabulka (DHT)

- Identifikátory uzlů i souborů jsou hash.
- Metrika blízkosti: bitový XOR ($d(a, b) = a \oplus b$).
- Pro směrování používá distribuovanou hašovací tabulku (DHT), která obsahuje informace o dalších uzlech a souborech.
  - Myšlenka: znám dobře svoje okolí, čím dále jsem, tím méně znám. Ale mám tam kontakty, kterých se můžu ptát a které mají lepší lokální informace.
  - Každý uzel má uloženou DHT ze své perspektivy.

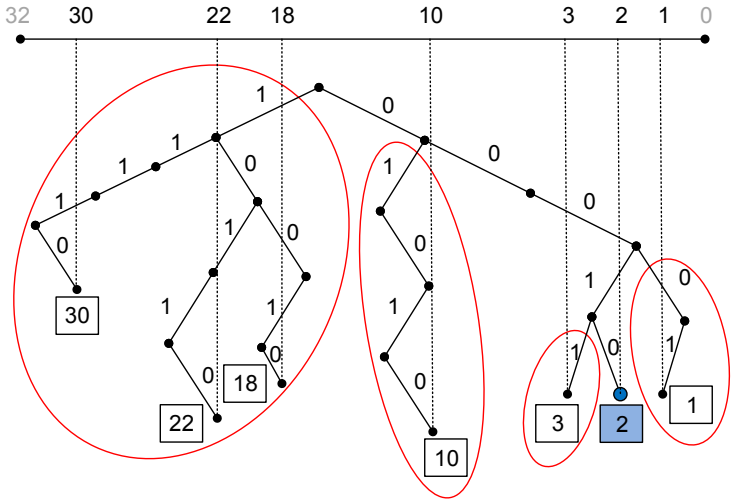



Obrázek 60.6: DHT se 160 bitovým jmenným prostorem. Uzly jsou umístěny ve stromu podle prefixů.

- **Směrovací tabulka** vzniká dynamicky, tím že přijde zpráva od jiného uzlu (spočítá se vzdálenost a uzel se přidá do tabulky).
  - Pokud daný řádek obsahuje méně než k položek (počet replikací), tak se uzel přidá.
  - Pokud je plný, tak zkontroluji, jestli jsou uzly dostupné, případně aktualizuju.

- Tento princip využívá P2P síť **BitTorrent** (buď využívá DHT a nebo koncept *trackers*).
  - *tracker* – Uzel, který má informace o tom, který uzel obsahuje jaké zdroje.
  - *torrent* – Soubor s informaci o sdílených souborech a s odkazem na tracker.
  - BitTorrent je pak aplikační protokol pro distribuci souborů nad TCP.

- Jmenný prostor o velikosti $N = 32$, k=3, délka ID je 5 bitů.
- Uzly ve stejném podstromu mají stejný prefix i stejnou vzdálenost od daného uzlu.
- Např. $d(2,3) = 2 \oplus 3 = 1, d(10 \oplus 2) = 8, d(22 \oplus 2) = 20, d(30 \oplus 2) = 28$, apod.



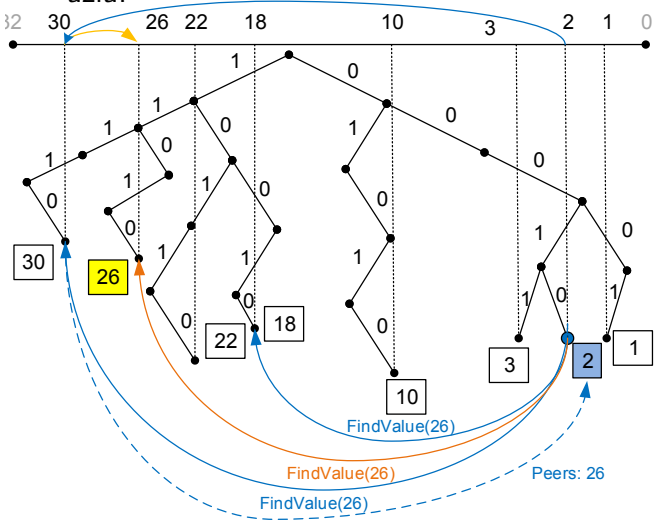

| Node ID=2 | | |
|---|---|---|
| $i$ | Interval | Peers |
| 1 | $[1,2)$ | 3 |
| 2 | $[2,4)$ | 1 |
| 3 | $[4,8)$ | |
| 4 | $[8,16)$ | 10 |
| 5 | $[16,32)$ | 18, 22, 30 |

$k = 3$ je stupeň replikace

- Řádek $i$ obsahuje uzly podstromu v dané vzdálenosti $[2^i, 2^{i+1})$ od daného uzlu.

Obrázek 60.7: Konkrétní příklad DHT pro 5 bitový jmenný prostor. Napravo je směrovací tabulka uzlu 2. *Interval* je interval vzdáleností a *Peers* uzly co se v této vzdálenosti vyskytují.

- `PING(nodeID)`: kontroluje, zda daný uzel je připojen.
- `STORE(fileID,nodeID)`: uloží do daného uzlu `fileID` a `nodeID`, který ho nabízí.
- `FIND_NODE(nodeID)`: daný uzel vrátí nejbližší uzly k uzlu `nodeID`.
- `FIND_VALUE(fileID)`: vrátí adresu uzlu obsahující soubor nebo seznam nejbližších uzlů.




Činnost uzlu ID=2 při hledání obsahu s fileID=26, $\alpha = 2$.

1. Uzel 2 vyhledá *k-bucket*: $d(2,26) = 2 \oplus 26 = 24$.
2. Pošle dotaz na dva ($\alpha = 2$) nejbližší uzly z {18,22,30}.
3. Každý oslovený uzel prohledá svou tabulku.
4. Uzel 30 vrátí odkaz na nejbližší uzel s nodeID=26.
5. Uzel 2 vyhledá soubor s fileID=26.

Obrázek 60.8: Příklad vyhledávání v DHT. Uzel 2 hledá zdroj 26. XOR vzdálenost 2 a 26 je 24. Proto se ptá uzlů, které má ve vzdálenosti $\langle 16, 32)$. Jeden uzlů, podá přesnější informaci (uzel s ID 26). Uzel 2 se následně zeptá uzlu 26.

# Kapitola 61

# WAP – Události v JavaScriptu (smyčka událostí, asynchronní programování, klientské události, obsluha událostí).

## 61.1 Zdroje

- `01-JavaScript.pdf`
- `WAP_2021-02-11.mp4`
- `WAP_2021-02-18.mp4`
- `WAP_2021-02-25.mp4`

## 61.2 Úvod a kontext

- ECMAScript – standardizace Javascriptu.
- Dynamicky typovaný.
- Hoisting – deklarace vsech promennych se presouvaji na zacatek bloku.
- Datové typy – primitivní vs objekty (obecný objekt, pole, regex, funkce).
- Operátory rest a spread.
- Callbacks – Funkce $f$ je předaná jako parametr funkci $g$. Při vyhodnocování funkce $g$ je funkce $f$ zavolána.
- Prototypování, this.

## 61.3 Zpracování události

- Kód se vykonává tak dlouho, dokud se sám nevzdá procesoru (run-to-completition)
- Asynchronní funkce se deklauje pomocí klíčového slova `async` – při vykonávání funkce, se příkazy vykonávají asynchronně.
- Např. čekání na data z API (`fetch`, `axios`).

- Vrácen je `promise` – Slib, že v budoucnu bude něco navráceno. K datům uvnitř `promise` se pristupuje přes `then`.

```
for (let i=1; i<=5; i++) {
    setTimeout( function timer() {
        console.log(i);
    },i*1000 );
}
```

Výpis 61.1: Příklad asynchronního programování, async + await.

- [[todo]]

## 61.4 Smyčka událostí

Hlavní smyčka programu – event loop postupně zpracovává jednu událost za druhou každá událost se zpracuje, dokud je v rámci co vykonávat

- [[todo]]

## 61.5 Asynchronní programování

Klasické synchronní volání funkce způsobí, že vlákno je po celou dobu operace blokované, ať už prováděním kódu samotné funkce, nebo čekáním na dokončení operace probíhající mimo váš program (např. volání API, databázový dotaz). To znamená, že vlákno po dobu prováděné operace nemůže dělat žádnou jinou práci. V praxi je ale často zapotřebí, aby operace probíhala někde na pozadí a vlákno, ze kterého byla operace volána, bylo využito k vykonání jiné části programu. Když je operace dokončena, je o tom volající informován a může na to nějakým způsobem reagovat. Tento přístup k tvorbě programů se nazývá asynchronní programování a řeší blokování vláken.

### 61.5.1 Asynchronní programování v Javascriptu

- Asynchronní funkce se deklauje pomocí klíčového slova `async` – při vykonávání funkce, se příkazy vykonávají asynchronně.
- Např. čekání na data z API (`fetch`, `axios`).
- Vrácen je `promise` – Slib, že v budoucnu bude něco navráceno. K datům uvnitř `promise` se pristupuje přes `then`.

```javascript
function resolveAfter2Seconds() {
    return new Promise(resolve => {
        setTimeout(() => {resolve("resolved"); },2000);
    })
}

async function asyncCall() {
    console.log("calling");
    const result = await resolveAfter2Seconds();
    console.log(result);
}

asyncCall();
```

Výpis 61.2: Příklad asynchronního programování, async + await.

```javascript
const fetchPromise = fetch("https://mdn.github.io/learning-area/javascript/apis/
    fetching-data/can-store/products.json");

console.log(fetchPromise);

fetchPromise.then( response => {
    console.log(`Received response: ${response.status}`);
});

console.log("Started request...");

// Output:
// Promise {<state>: "pending" }
// Started request...
// Received response: 200
```

Výpis 61.3: Příklad asynchronního programování, volání API.

## 61.6 Klientské události

- event listener
- **[[todo]]**

## 61.7 Obsluha událostí

- **[[todo]]**

# Kapitola 62

# WAP – Přenos a distribuce webových dat (URI, protokol HTTP, proudy HTTP, CDN, XHR).

## 62.1 Zdroje

- `WAP04_transport.pdf`
- `WAP05_frontend.pdf`
- `WAP_2021-03-11.mp4`
- `WAP_2021-03-18.mp4`

## 62.2 URI

- URI (*universal resource identifier*) je textový řetězec s definovanou strukturou, který slouží k přesné specifikaci zdroje informací (ve smyslu dokument nebo služba) na internetu.
- URL (*uniform resource locator*) a UNR (*uniform resource name*) jsou podmnožiny URI.

```
scheme:[//[user[:password]@]host[:port]][/path][?query][#fragment]
```

Výpis 62.1: Struktura URI.

- Jednotlivé části:
  - `scheme`
    - Definuje sémantiku zbytku URI.
    - Např. `http`, `ftp`, `file`, `git`, `ldap`, `mailto`, `postgresql`, ...
  - `:[//[user[:password]@]host[:port]]`
    - Autorita cesty – Delegace zodpovědnosti za popis URI.
    - Volitelná autentizace (user, password).
    - Doménové jméno nebo IP adresa.
    - Číslo portu transportní vrstvy (volitelné).

  - `[/path]`
    * Cesta (hierarchická část).
  - `[?query]`
    * Obsahuje segmenty ve formatu `klic=hodnota` oddělené `&`.
    * Např. `?submit_view=t&order=6&form_f1=2017`
  - `[#fragment]`
    * Identifikace sekundárního zdroje v rámci primárního zdroje.
- Příklady:
  - `https://jakjsmenatom.cz`
  - `file://localhost/home/vdusek/Documents/hello.txt`
  - `mailto:josef.novak@gmail.com`
  - `postgresql://user_name:secret_password@localhost/db_name`

## 62.3 Protokol HTTP

- HTTP *hypertext transfer protocol*) je internetový protokol určený pro komunikaci s webovými servery. Slouží pro přenos hypertextových dokumentů zejména ve formátu HTML, ale i jiných typů souborů.
  - Dovoluje přenos dat téměř jakéhokoliv formátu (hlavička Content Type – MIME).
- Původně protokol typu: požadavek → odpověď.
- Metody (stejné pro všechny verze):
  - GET: získání zdroje
  - HEAD: obdobné, ale pouze hlavičky
  - POST: zaslání dat na server, změna stavu
  - PUT: nahrazení (kompletní přepsání) zdroje dodaným obsahem
  - PATCH: Částečná modifikace zdroje
  - DELETE: odstranění zdroje
  - CONNECT: vytvoření tunelu, např. při použití HTTPS a proxy
  - OPTIONS: zjištění podporovaných metod (zdroj, nebo server) a dalších parametrů (např. cache)
  - TRACE: Debugging, adresát vrací původní zprávu (s drobnými změnami)
- Hlavičky: Content-Type, Content-Length, Connection, Keep-Alive, ...
- Stavové kódy.

### 62.3.1 HTTPS

- HTTPS znamená HTTP s využitím TLS (*transport layer security*), což je mezivrstva mezi TCP a HTTP, která řeší šifrování.
- Zajišťuje:
  - Integritu – bez znalosti kryptografického klíče není možnost data změnit tak, aby to nebylo možné detekovat.

  - Utajení – bez znalosti kryptografického klíče není možnost data číst.
- Autentizace jedné/obou komunikujících stran pomocí certifikátů X.509, které obsahují veřejný klíč, a který je podepsaný certifikační autoritou (alternativně *self signed*).

### 62.3.2 HTTP/1.0

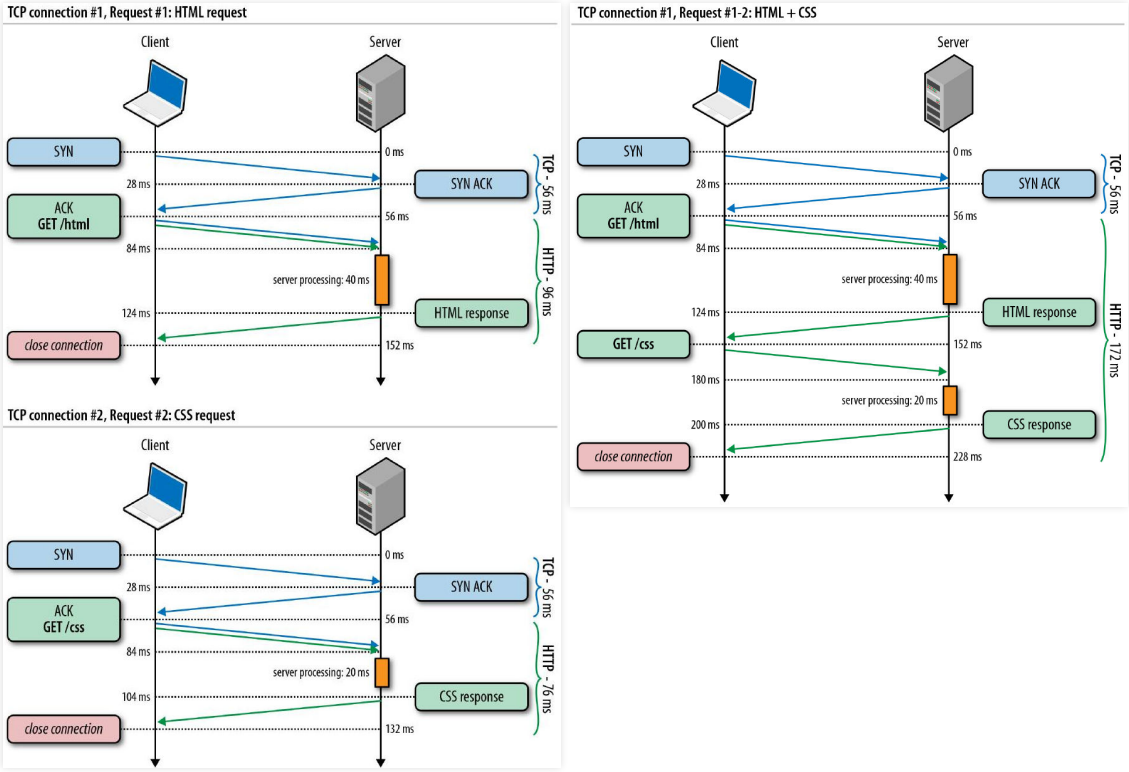
- Pro každý dotaz musí být navázáno TCP spojení (SYN, ACK), to je 1 RTT (*round trip time*).
  - V případě HTTPS ještě TLS, to je 1-2 RTT navíc.
- Spoustu volitelných hlaviček.

```
GET / HTTP/1.0
User-Agent: telnet
Accept: */*

HTTP/1.1 200 OK
Date: Fri, 08 Feb 2019 13:07:30 GMT
Server: Apache
Content-Location: index.php.cz
Vary: negotiate,accept-language
TCN: choice
Pragma: no-cache
X-Frame-Options: deny
Connection: close
Content-Type: text/html; charset=iso-8859-2
Content-Language: cs
```

Výpis 62.2: Příklad HTTP/1.0 komunikace.

### 62.3.3 HTTP/1.1

- Zavádí **Keep-Alive** spojení – Znovupoužití jednoho TCP spojení pro více dotazů.
  - Timeout Keep-Alive – Konec spojení.
- Přenosy dat po částech (chunked).
- Požadavky na konkrétní části dat (byte-range).



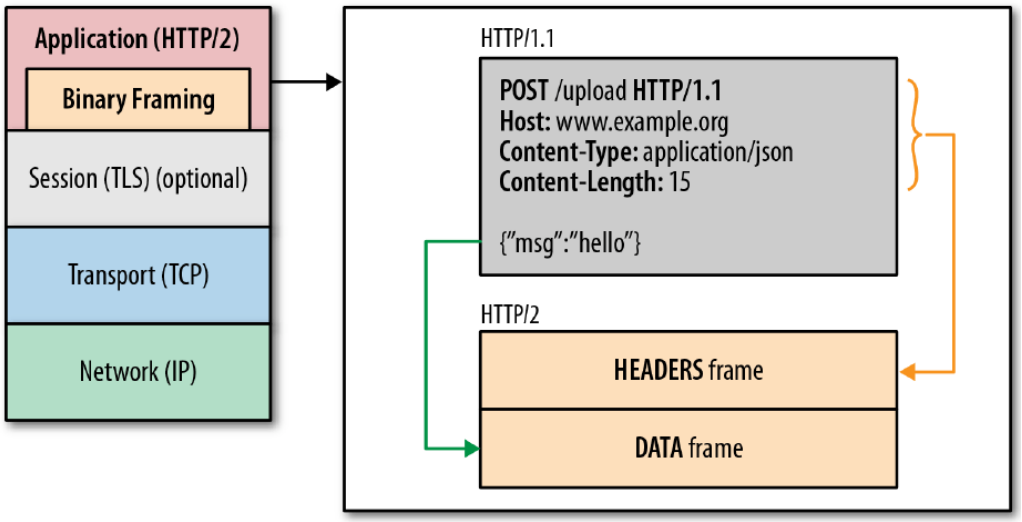

Obrázek 62.1: Komunikace HTTP 1 a HTTP 1.1 s Keep-Alive.

- **Pipelining** – Pošlu více souvisejících požadavků ihned za sebou (např. `GET/html` a `GET/css`).
  - Problém dlouho zpracovávaných dotazů (typ *head of line blocking*).
  - Potenciální útok na server → vyčerpání paměti.
  - Některé servery, či mezilehlé prvky (cache, aplikační firewall) nemusí podporovat pipelineing.
  - Vzniká iniciativou klienta.
- Paralelizace dotazů HTTP – Prohlížeč automaticky ustanoví několik TCP spojení se serverem.

### 62.3.4 HTTP/2

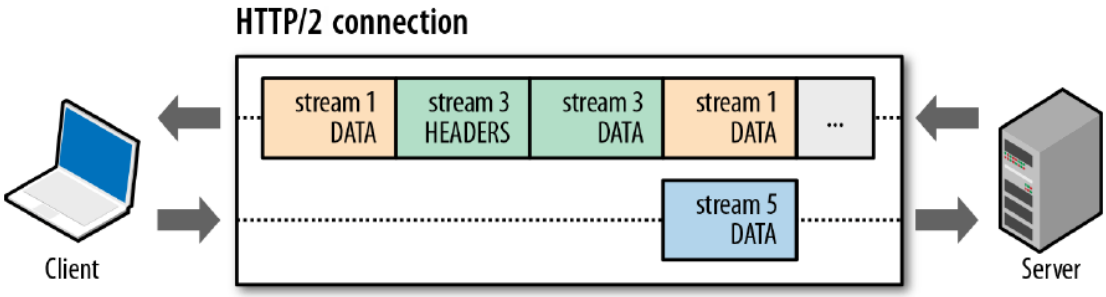
- Cíl: ještě více vylepšit využívání jediného spojení.
  - Paralelní zpracování několika požadavků.
  - Proudy dat.
  - Vylepšení konceptu pipeliningu, aby skutečně fungoval.
  - Multiplexing proudů, části požadavků i odpovědí se mohou míchat.
- Šifrovaná (TLS 1.2 a novější) i nešifrovaná verze
- Data jsou přenášena v **binární podobě** – možnost převodu do textové formy v debuggerech.
  - Zachovává sémantiku metod.

  - Zachovává členění zprávy do hlaviček a těla
  - Zprávy jsou přenášeny v rámcích.




Obrázek 62.2: Binární formát.

- Složky spojení HTTP/2:
  - Rámce – Nejmenší jednotky, nesou hlavičku rámce s identifikátorem proudu, do kterého rámec patří.
  - Zprávy – Kompletní sekvence rámců, které tvoří dotazy a odpovědi.
  - **Proud (stream)** – Obousměrný tok dat, který nese zprávy.



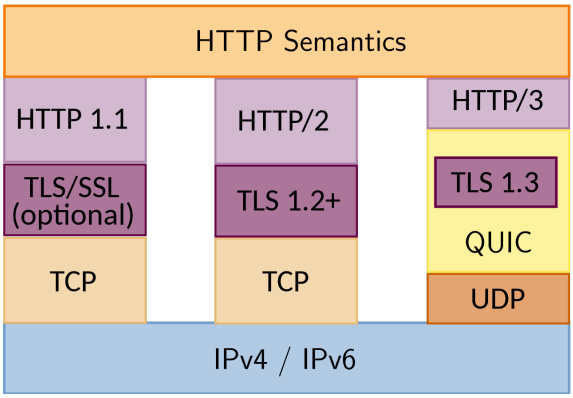

Obrázek 62.3: HTTP 2 spojení, binární streamy.

- **Server push** – zaslání odpovědi aniž by přišel dotaz.
  - HTTP přestává být protokolem typu dotaz → odpověď a stává se dotaz → $1 - n$ odpovědí.
  - Např. Klient si zažádá o index.html. Server mu ho pošle a k tomu metada o dalších souborech, které s ním souvisí, ve formě tzv. push promises.
    * Jde o stejný obsah jako při HTTP HEAD.
- Výhody HTTP/2:
  - Jedno spojení nese několik dotazů a odpovědí.

- Proudům je možné přiřazovat priority (HTML, CSS, JavaScript prioritní. Obrázky, videa a další zdroje méně prioritní).
- Jedno TCP spojení per origin (trojice schema, hostname, port).
- Nabízí kompresi hlaviček.

### 62.3.5 HTTP/3

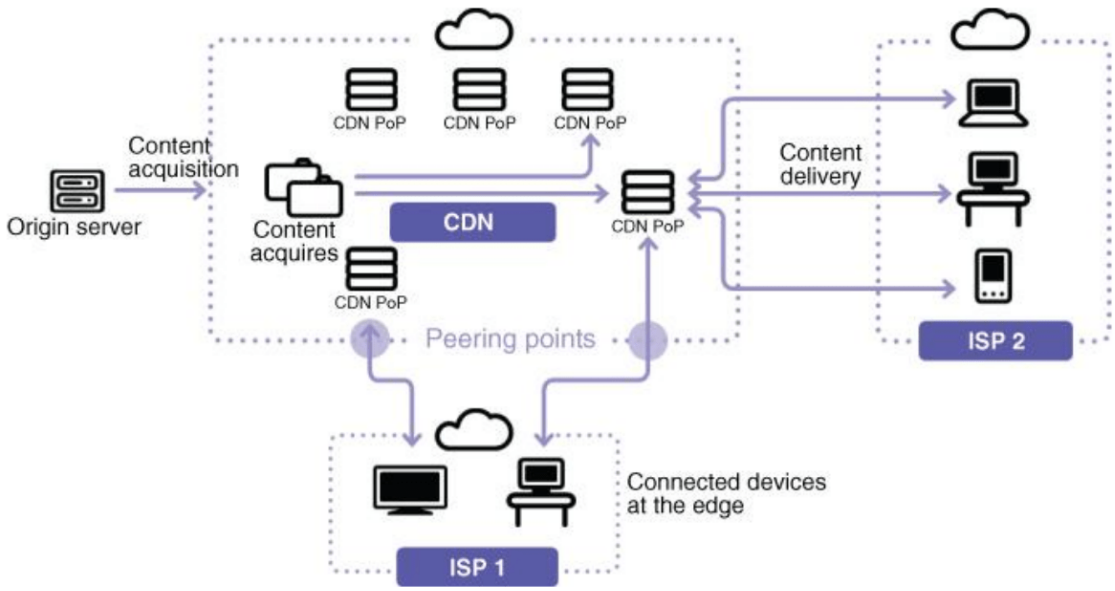
- Cíl, vyřešit problémy:
  - Head of line blocking při TCP – přejít na UDP.
  - Problém dlouhého navázání spojení TCP (1 RTT) + TLS (až 2 RTT).
- Nový protokol QUIC mezi transportní a aplikační vrstvou.




Obrázek 62.4: HTTP evoluce a protokoly.

## 62.4 CDN

- Problém: DNS lookup – 1 RTT, TCP handshake – 1 RTT, TLS handshake – 2 RTT, HTTP request – 1-n RTT.
- Tedy, komunikace může být někdy pomalá, řešíme, jak web zrychlit.
- Řešení je tzv. CDN (*content delivery network*).
- Myšlenka: Velké části obsahu stránky jsou neměnné, málo časté změny. Nakopírujeme je blíže klientům.
  - Distribuovat servery různě po světě – nejbližší uzel blíže uživateli.
  - Distribuce (nejen) statického a objemného obsahu co nejblíže ke klientovi.
- Soukromá (poskytovatel obsahu vlastní CDN), vs. komerční CDN (Akamai, Limelight, Level-3 a další).
- Zároveň slouží jako ochrana před DDoS.




Obrázek 62.5: Příklad CDN.

## 62.5 XHR (AJAX)

- XHR (XMLHttpRequest), resp. AJAX (*Asynchronous JavaScript and XML*).
- API prohlížeče pro dynamické dotazy pomocí HTTP(S).
  - Dynamické úpravy DOM.

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "/images/photo.webp");
xhr.responseType = "blob";

// obsluzna funkce (callback)
xhr.onload = function() {
    if (this.status == 200) {
        var img = document.createElement("img");
        img.src = window.URL.createObjectURL(this.response);
        img.onload = function() {
            ndow.URL.revokeObjectURL(this.src);
        }
        document.body.appendChild(img);
    };
}

xhr.send();
```

Výpis 62.3: Příklad využití XHR.

# Kapitola 63

# WAP – Bezpečnost webových aplikací (SOP, XSS, CSRF, bezpečnostní hlavičky HTTP).

## 63.1 Zdroje

- `WAP10_Bezpecnost.pdf`
- `WAP_2021-04-15.mp4`

## 63.2 Úvod a kontext

- Pouhé načtení webové stránky znamená vykonání cizího (Javascript) kódu v počítači.
- Výrobci prohlížečů vyvažují dva cíle:
  - Tvorba API umožňujících přístup k HW a SW uživatelského počítače (grafická karta, geolokace, mikrofon, kamera, ...).
  - Zamezení/omezení dopadů škodlivého kódu (ochrana soukromí, úprava dat, podvody, ...).

## 63.3 Správa sezení

- Vzhledem k tomu, že protokol HTTP je bezstavový, potřebujeme způsob jak asociovat jeden HTTP request druhému, tedy způsob jak ukládat uživatelská data mezi HTTP requesty.
- Soubory cookie nebo parametry URL (např. jako `example.com?asd=lol&boo=no`) jsou vhodnými způsoby přenosu dat mezi dvěma nebo více požadavky. Nejsou však vhodné v případě, že nechceme, aby tato data byla čitelná/editovatelná na straně klienta.
  - Cookies jsou malé bloky dat vytvořené webovým serverem během prohlížení webových stránek uživatelem a uložené do jeho počítače nebo jiného zařízení jeho webovým prohlížečem.
- Řešením je uložit tato data na straně serveru, dát jim id a nechat klienta znát (a předat zpět při každém http požadavku) pouze toto id. ID je na straně klienta uloženo jako cookie.

- Sezení (session) je tedy ustanovené síťové spojení mezi klientem a serverem.
- Možné útoky:
  - Krádež sezení – získání hodnoty ID sezení útočníkem.
  - Vnucení sezení – vložení útočníkova ID sezení k oběti.
- Jak zabránit odcizení ID sezení?
  - Cookies jsou přenášeny pouze přes HTTPS.
  - Cookies není dostupné přes Javascript, je dostupné pouze přes HTTP.
  - Nastavení délky sezení (automatické odhlášení).
  - Změna ID sezení, pokud se provede nějaká akce (přihlášení/odhlášení, citlivá akce – změna hesla, úprava oprávnění, ...).

## 63.4 Same-origin policy (SOP)

- Origin (původ), definovaný trojicí URI segemntů —- schéma, doména, port.
- SOP je důležitý bezpečnostní mechanismus, který omezuje způsob, jakým může dokument nebo skript načtený jedním origin interagovat se zdrojem z jiného origin.
  - Webový prohlížeč povolí skriptům obsaženým na první webové stránce přístup k datům na druhé webové stránce, ale pouze v případě, že obě webové stránky mají stejný origin.
- SOP zabraňuje tomu, aby škodlivý skript na jedné stránce získal přístup k citlivým datům na jiné webové stránce prostřednictvím jejího DOM.
- Příklad:
  - Webová aplikace je dostupná na url `https://jakjsmenatom.cz`.
  - Její API je dostupné na url `https://api.jakjsmenatom.cz`.
  - Při SOP, API komunikuje pouze se subdoménami `jakjsmenatom.cz`, z jiné domény nelze API dotazovat.
- Útok: **Postraní kanály obcházející SOP**
  - SOP brání přečtení odpovědi → nebrání však odeslání požadavku.
    - XHR na origin mimo SOP.
    - Obrázek s atributem src mimo SOP.
  - XHR generuje událost load při existenci zdroje, i když prohlížeč přístup k obsahu odmítne.
    - Podle událostí error, či timeout lze zjistit otevřené porty.
  - V kombinaci s CSRF lze měnit stav na serveru.

## 63.5 Cross-site scripting (XSS)

- Útočník vloží značky nebo skripty do cílové stránky.
  - Kód se vykoná, když uživatel navštíví stránku.
- Stránka je zranitelná pokud:

– Zobrazuje obsah na základě uživatelských vstupů, které jsou bez dostatečné validace.

```
<script>
var name = decodeURIComponent(window.location.search.substring(1)) || "";
document.write("Hello " + name);
</script>
```

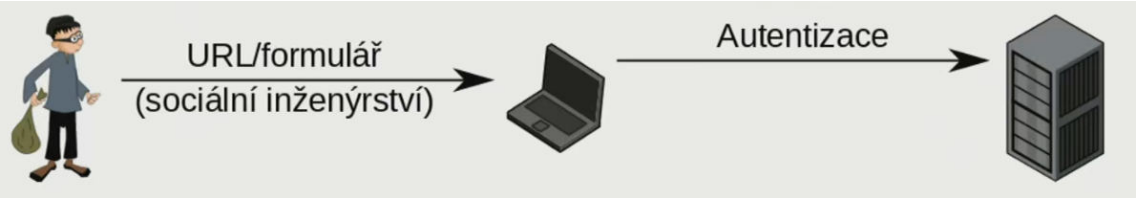
- Očekává se spuštění s URL jako `http://www.example.com/greet.html?David`
  - Co se stane `http://www.example.com/greet.html?%3Cscript%3Ealert('David')%3C/script%3E?`
  - Nebo `http://siteA/greet.html?name=%3Cscript src=siteB/evil.js%3E%3C/script%3E?`

Obrázek 63.1: Příklad XSS.

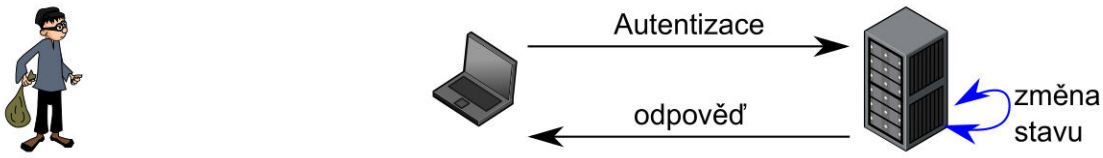
- Varianty XSS:
  - Trvalé (perzistentní), uložené v databázi.
  - Dočasné.
  - Lokální (DOM based)
- Jak realizovat?
  - Zkracovače odkazů, přesměrování (hlavička Location), znaky kódované pomocí reprezentace ASCII, ...
- Nebezpečí XSS: odcizení cookies, obsah web storage, zmáčknuté klávesy, pohyb myši, upravovat formuláře, těžba kryptoměn.
- Řešení:
  - sanitizace vstupů;
  - HTTP hlavička **Content Security Policy**:
    * Zakázání prohlížeči stahování zdrojů mimo povolené origin.
    * CSP poskytuje majitelům webových stránek standardní metodu pro deklarování schváleného původu obsahu, který by měly prohlížeče na dané webové stránce povolit načítat.
  - HTTP hlavička **Feature policy**:
    * Stránka vyžaduje, nebo nevyžaduje určité vlastnosti (např. zobrazení na fullscreen, přístup k poloze ...).
    * Omezení přístupu Javascriptu k browser API (např. zakázání přístupu k poloze).

## 63.6 Cross-site request forgery (CSRF)

- Útočník přinutí uživatele změnit stav ve webové aplikaci, ke které je již přihlášený.
- Změna stavu, ne krádež dat. – Útočník nemá přístup k odpovědi na změnu stavu.
- Není porušená důvěrnost, ale je porušena integrita.




Obrázek 63.2: Princip CSRF, část 1.




Obrázek 63.3: Princip CSRF, část 2.

- Pomocí navštívení URL (GET)
  - http://bank.com/transfer.do?acct=MARIA&amount=100000

```
<a href="http://bank.com/transfer.do?acct=MARIA&amount=100000">View my Pictures!</a>
<img src="http://bank.com/transfer.do?acct=MARIA&amount=100000" width="0" height="0" border="0">
```

  - https://example.com/předmět/12345/termín/123/odregistruj
    https://example.com/předmět/12345/termín/321/registruj
- Vyplnění formuláře (POST)
  - Skryté pole nastavují požadované hodnoty
- XHR (GET, PUT, DELETE, POST atd.)
  - Nutná návštěva úročníkova webu, příp. napadaného webu

Zdroj: https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)

Obrázek 63.4: Jak je možné realizovat CSRF útok.

- Obrana proti CSRF – **Synchronizační token**
  - Každá zmena stavu závisí na náhodném unikátním tokenu (unikátní pro sezení).
  - Token je unikátní pro konkrétní operací, obsahuje zašifrované informace klíčem dostupným jen serveru – typ operace, čas operace, jméno uživatele, nonce, unikátní náhodný řetězec.
  - Token použit
    * Součástí URL GET měnících stav – Lépe: nepoužívat GET definovaný jako bezpečná metoda pro změnu stavu.
    * Součástí požadavků PUT, DELETE aj. prováděných pomocí XHR.
    * Skryté pole ve formulářích.
- Obrana proti CSRF – **HTTP hlavičky**
  - Kontrola hlavičky Origin
    * Origin: `https://fit.vutbr.cz`
    * Obsahuje origin, ze kterého pochází dotaz.
    * Kontrola, zda odpovídá očekávanému (pokud je součásti dotazu).

  - Kontrola hlavičky Referer
    * Referer: `https://fit.vutbr.cz/study/courses/WAP/private/lectures/2019.php?p=bezpečn`
    * URL naposledy zobrazené stránky.
    * Kontrola, zda odpovídá očekávanému (pokud je součásti dotazu).
- Obrana proti CSRF – **Interakce uživatele**
  - Nová autentizace.
  - Jednorázově použitelný token (explictně zadaný uživatelem).
  - CAPTCHA.

## 63.7 Bezpečnostní hlavičky HTTP

- Pomocí bezpečnostních HTTP hlaviček můžeme nastavit různá pravidla pro komunikaci mezi webovým prohlížečem a serverem.
- Typicky umožňují povolit nebo zakázat určité funkce prohlížeče pro vyšší bezpečnost a soukromí.
- Např. Content Security Policy, Feature Policy, Strict-Transport-Security, X-Frame-Options.

### 63.7.1 HTTPS stripping

- Man in the middle útok.
- Postup:
  1. Uživatel vynechá schéma URI, např. www.example.com.
  2. Výchozí schéma je http.
  3. Útočník nepropaguje přesměrování na HTTPS.

Obrázek 63.5: Útočník MITM přiměje oběť k přístupu přes HTTP.

- HTTP hlavička **HTTP Strict-Transport-Security**
  - Na této doméně se vždy šifruje (platí pro všechny porty).
  - Při HTTP server pošle výzvu, že jakékoliv HTTP požadavky předělat na HTTPS.

### 63.7.2 Clickjacking

- Zobrazení několika vrstev prvků přes sebe (neviditelná vrstva).
- Uživatel si myslí, že kliká na něco jiného než ve skutečnosti.
- Při útoku jsou využívány plovoucí rámce (iframe).

- Řešení spočívá v zakázání/omezení zobrazení stránky v iframe.
- HTTP hlavička **X-Frame-Options**
  - Povoluje/zakazuje/omezuje zobrazení stránky v rámci iframe.
- Hlavička **Content Security Policy**.