Algoritmy a datové struktury

Prezenční studium VŠFS

Ondřej Čepek

Sylabus

- Asymptotické odhady a třídění
- Algoritmy typu „Rozděl a panuj“
- Dolní odhady
- Grafové algoritmy
- Vyhledávání v textu
- Grafové algoritmy
- Třídy P a NP

Algoritmy pro třídění pomocí porovnávání (dvojic prvků)

Úloha: setřídit n čísel (klíčů), které jsou (nesetříděné) v poli A

```
Bubble-Sort (A);
begin    for j := 1 to n - 1 do
         for i := 1 to n - j do
         if (A[i] > A[i+1]) then begin x := A[i]; A[i] := A[i+1]; A[i+1] := x end
end.
```

Časová složitost

v nejhorším případě (reverzně setříděný vstup)	$\Theta(n^2)$
v nejlepším případě (setříděný vstup)	$\Theta(n^2)$
v průměrném případě (přes všechny permutace vstupu)	$\Theta(n^2)$

Bubble-Sort lze modifikovat tak, že po každé fázi (průchod vnitřním cyklem), zkontroluje zda došlo k nějaké změně (alespoň jednomu prohození) a pokud ne tak ukončí běh. Tato modifikace zlepší časovou složitost nejlepšího případu na $\Theta(n)$, ovšem nejhorší i průměrný případ zůstanou $\Theta(n^2)$.

```
Insertion-Sort (A);
begin    for j := 2 to n do
         begin    x := A[j]; i := j - 1;  {vlož A[j] do již setříděné části pole A[1] – A[j-1]}
                  while (i > 0) and (A[i] > x) do
                  begin A[i+1] := A[i]; i := i - 1 end;
                  A[i+1] := x
         end
end.
```

Na rozdíl od Bubble-Sortu porovná každou dvojici vstupních čísel nejvýše jednou (je tedy o něco „chytřejší“), ale i tak jde o algoritmus kvadratický.

Časová složitost

v nejhorším případě (reverzně setříděný vstup) $\Theta(n^2)$

v nejlepším případě (setříděný vstup) $\Theta(n)$

v průměrném případě (přes všechny permutace vstupu) $\Theta(n^2)$

Cíl: třídící algoritmus, který i v nejhorším případě třídí rychleji než v kvadratickém čase

Jak porovnávat algoritmy?

časová složitost algoritmu
prostorová složitost algoritmu

} oboje závisí na „velikosti“ vstupních dat

Jak měřit velikost vstupních dat?

rigorózně: počet bitů nutných k zapsání vstupních dat

Příklad: vstupem jsou (přirozená) čísla $a_1, a_2, \dots, a_n$ která je třeba setřídit

velikost dat D v binárním zápisu je $|D| = \lceil \log_2 a_1 \rceil + \dots + \lceil \log_2 a_n \rceil$

časová složitost = funkce $f(|D|)$ udávající počet kroků algoritmu v závislosti na velikosti vstupních dat

intuitivně: není podstatný přesný tvar funkce f (multiplikativní a aditivní konstanty), ale pouze to, do jaké „třídy“ funkce f patří (lineární, kvadratická, exponenciální, …)

Příklad: $f(|D|) = a\,|D| + b$ lineární algoritmus

$f(|D|) = a\,|D|^2 + b\,|D| + c$ kvadratický algoritmus

$f(|D|) = k\,2^{`D`}$ exponenciální algoritmus

Co je to krok algoritmu?

rigorózně: operace daného abstraktního stroje (Turingův stroj, stroj RAM)

zjednodušení (budeme používat): krok algoritmu = operace proveditelná v konstantním (tj. na velikosti dat nezávislém) čase

- aritmetické operace (sčítání, odčítání, násobení, …)
- porovnání dvou hodnot (typicky čísel)
- přiřazení (pouze pro jednoduché datové typy, ne pro pole …)

→ tím se zjednoduší i měření velikosti dat (čísla mají pevnou maximální velikost)

Příklad: setřídit čísla $a_1, a_2, \ldots, a_n$ → velikost dat je $|D| = n$

Toto zjednodušení nevadí při porovnávání algoritmů, ale může vést k chybě při zařazování algoritmů do tříd složitosti

Proč měřit časovou složitost algoritmů?

stačí přeci mít dostatečně rychlý počítač …

Doba provádění f(n) operací (délka běhu algoritmu) pro vstupní data velikosti n za předpokladu že použitý hardware je schopen vykonat 1 milion operací za sekundu

	n						
f(n)	20	40	60	80	100	500	1000
n	20μs	40μs	60μs	80μs	0.1ms	0.5ms	1ms
n log n	86μs	0.2ms	0.35ms	0.5ms	0.7ms	4.5ms	10ms
n^2	0.4ms	1.6ms	3.6ms	6.4ms	10ms	0.25s	1s
n^3	8ms	64ms	0.22s	0.5s	1s	125s	17min
2^n	1s	11.7dní	36tis.let				
n!	77tis.let						

Růst rozsahu zpracovatelných úloh, tj. „zvládnutelné“ velikosti vstupních dat, díky zrychlení výpočtu (lepší hardware) za předpokladu, že na „stávajícím“ hardware lze řešit úlohy velikosti x

	Zrychlení výpočtu			
f(n)	původní	10 krát	100 krát	1000 krát
n	x	10x	100x	1000x
n log n	x	7.02x	53.56x	431.5x
n^2	x	3.16x	10x	31.62x
n^3	x	2.15x	4.64x	10x
2^n	x	x+3	x+6	x+9

Asymptotická (časová) složitost

Intuitivně: zkoumá „chování“ algoritmu na „velkých“ datech, tj. nebere v úvahu multiplikativní a aditivní konstanty, pouze zařazuje algoritmy do „kategorií“ podle jejich skutečné časové složitosti

Rigorózně:

$f(n)$ je asymptoticky menší nebo rovno $g(n)$, značíme $f(n) \in O(g(n))$, pokud

$$\exists c>0\ \exists n_0>0\ \forall n \geq n_0 : 0 \leq f(n) \leq c\, g(n)$$

$f(n)$ je asymptoticky větší nebo rovno $g(n)$, značíme $f(n) \in \Omega(g(n))$, pokud

$$\exists c>0\ \exists n_0>0\ \forall n \geq n_0 : 0 \leq c\, g(n) \leq f(n)$$

$f(n)$ je asymptoticky stejné jako $g(n)$, značíme $f(n) \in \Theta(g(n))$, pokud

$$\exists c>0\ \exists d>0\ \exists n_0>0\ \forall n \geq n_0 : 0 \leq c\, g(n) \leq f(n) \leq d\, g(n)$$

$f(n)$ je asymptoticky ostře menší nebo rovno $g(n)$, značíme $f(n) \in o(g(n))$, pokud

$$\forall c>0\ \exists n_0>0\ \forall n \geq n_0 : 0 \leq f(n) \leq c\, g(n)$$

$f(n)$ je asymptoticky ostře větší nebo rovno $g(n)$, značíme $f(n) \in \omega(g(n))$, pokud

$$\forall c>0\ \exists n_0>0\ \forall n \geq n_0 : 0 \leq c\, g(n) \leq f(n)$$

Heap-Sort

Základem algoritmu je použití datové struktury „binární halda“, což je binární strom s následujícími dvěma vlastnostmi:

1. Je to úplný strom, tzn. že všechna patra stromu kromě posledního jsou zcela zaplněna a poslední patro je zaplněno zleva.
2. Každá dvojice otec – syn splňuje nerovnost (klíč otce ≥ klíč syna)

Haldu lze implementovat polem, indexy synů i otce daného uzlu lze snadno spočítat.

Algoritmus má 2 fáze: (nesetříděná posloup. → halda) a (halda → setříděná posloup.)

Základem obou fází je následující rekurzivní procedura:

```
Heapify (A, i, n);  {nechá vrchol s indexem i propadnout na správné místo}
begin    l := 2*i; r := 2*i + 1;          {indexy synů, pokud vrchol i syny má}
         if (l <= n) and (A[l] > A[i]) then max := l else max := i;
         if (r <= n) and (A[r] > A[max]) then max := r;
         if (max <> i) then  begin     Prehod (A[i], A[max]);
                                       Heapify (A, max, n)
                             end
end.
```

Vytváření haldy z nesetříděné posloupnosti:

```
Build-Heap (A);
begin for i := n div 2 downto 1 do Heapify (A, i, n)
end.
```

Poznámka: vrcholy s indexy od n div 2 + 1 do n jsou listy, tj. tvoří podhaldy velikosti jedna, které není potřeba „opravovat“ pomocí procedury Heapify.

Vlastní třídící algoritmus:

```
Heap-Sort (A);
begin     Build-Heap (A);
                    for delka := n-1 downto 1 do begin   Prehod (A[1], A[delka+1]);
                                                         Heapify (A, 1, delka)
                                                end
end.
```

Poznámka: při každém průchodu cyklem (po provedení příkazu Přehoď) je aktuální halda v A[1] - A[delka], položky za A[delka] jsou již správně setříděné

Časová složitost: v nejhorším, nejlepším i průměrném případě $\Theta(n \log n)$

Algoritmy typu „Rozděl a panuj (Divide et impera)“

- metoda pro návrh algoritmů (ne dělení programu na samostatné celky)
- algoritmus typu „Rozděl a panuj“ má typicky 3 kroky

1. ROZDĚL úlohu na několik podúloh stejného typu ale menšího rozsahu
2. VYŘEŠ podúlohy, a to:
   a) rekurzivně dalším dělením pro podúlohy dostatečně velkého rozsahu
   b) přímo pro podúlohy malého rozsahu (často triviální)
3. SJEDNOŤ řešení podúloh do řešení původní úlohy

Příklady: MergeSort, Binární vyhledávání

Analýza časové složitosti

$T(n)$ doba zpracování úlohy velikosti n (předpoklad: pokud $n < c$ tak $T(n) = \Theta(1)$)

$D(n)$ doba na rozdělení úlohy velikosti n na a podúloh stejné velikosti n/b (většinou $a=b$)

$S(n)$ doba na sjednocení řešení podúloh do řešení původní úlohy velikosti n

$\Rightarrow$ rekurentní rovnice: $T(n) = D(n) + aT(n/b) + S(n)$ pro $n \geq c$

$T(n) = \Theta(1)$ pro $n < c$

Binární vyhledávání

Vyhledávání prvku `x` v setříděném poli `A` o `n` prvcích.

1. ROZDĚL pole `A` na poloviny.
2. VYŘEŠ menší podúlohu
   - vrácením indexu `n/2`, pokud `x=A[n/2]`,
   - hledáním prvku `x` v `A[1..n/2 - 1]`, pokud `x<A[n/2]`,
   - hledáním prvku `x` v `A[n/1 + 1]`, pokud `x>A[n/2]`,
   - odpovědí NENALEZEN, pokud `n<=0`.
3. SJEDNOŤ řešení podúloh vrácením výsledku hledání ve správné části pole.

$D(n) = O(1)$, $S(n)=O(1)$, $a=1$, $b=2$

$T(n) = D(n) + aT(n/b) + S(n) = T(n/2)+O(1)$ pro $n \geq 2$

$T(n) = \Theta(1)$ pro $n<1$

Protože si v každé instanci vystačíme s konstantním časem a jedním rekurzivním voláním na úlohu poloviční velikosti, složitost je $T(n)=O(\log n)$, což je i řešením dané rekurzivní rovnice, později uvidíme, jak se dají takové rovnice řešit.

Binární vyhledávání

```
function BinarniVyhledavani(A: array [1..N] of integer; x:
integer): integer;
{Ve vzestupně uspořádaném poli A hledá hodnotu x, je-li x
obsaženo v A, vrací index prvku x v A, jinak vrací 0.}
var i, j, k: integer;
begin
  i:=1;
  j:=N;
  repeat
    k:=(i+j) div 2; {index prostředního prvku}
                    {rozdělení pole na poloviny}
    if x>A[k] then i:=k+1  {výběr správné poloviny}
              else j:=k-1
  until (A[k] = x) or (i > j); {řešení podúlohy}
  if A[k]=x then BinarniVyhledavani := k {sjednocení}
            else BinarniVyhledavani := 0
end;
```

Merge-sort

Třídění pole A o n prvcích sléváním (vzestupně).

1. ROZDĚL pole A na dvě poloviny.
2. VYŘEŠ podúlohu na obou polovinách, tj. setřiď obě poloviny vzestupně za pomoci rekurzivního volání.
3. SJEDNOŤ (SLIJ) obě setříděné poloviny do jednoho setříděného pole (to je výrazně jednodušší, než celé třídění), sloučení lze provést v čase O(n).

$D(n) = O(1), S(n)=O(n), a=2, b=2$

$T(n) = D(n) + aT(n/b) + S(n) = 2T(n/2)+O(n)$ pro $n \geq 2$

$T(n) = \Theta(1)$ pro $n<1$

Řešení rovnice je $T(n)=O(n \log n)$, jak uvidíme později.

Mergesort

```
procedure Mergesort (var P, Q: array[1..N] of integer;
Zac, Kon: integer);
{Setřídí v poli P úsek od Zac do Kon, Q je pomocné pole
pro slévání.}
var Stred; {prostředek tříděného úseku}
    i, j, k: integer; {pomocné indexy}
begin
  Stred:=(Zac+Kon) div 2;
  if Zac < Stred then Mergesort(P, Q, Zac, Stred);
    {setřídění levého úseku}
  if Stred+1 < Kon then Mergesort(P, Q, Stred+1, Kon);
    {setřídění pravého úseku}




    {následuje slévání - viz další slajd...}
```

Mergesort (slévání)

```
i:=Zac; {levý úsek}
j:=Stred+1; {pravý úsek}
k:=Zac; {výsledek}
while (i<=Stred) and (j<=Kon) do {slévání do pole Q}
begin
  if P[i]<=P[j] then begin Q[k]:=P[i]; i:=i+1; end
                else begin Q[k]:=P[j]; j:=j+1; end;
  k:=k+1;
end;
while i<=Stred do {dokopírování zbytku levého úseku}
  begin Q[k]:=P[i]; i:=i+1; k:=k+1 end;
while j<=Kon do   {dokopírování zbytku pravého úseku}
  begin Q[k]:=P[j]; j:=j+1; k:=k+1 end;
{zbývá přenést setříděný úsek <Zac, Kon> zpět z Q do P:}
for k:=Zac to Kon do
  P[k]:=Q[k];
end; {procedure Mergesort}
```

Metody řešení rekurentních rovnic

1. substituční metoda
2. master theorem (řešení pomocí „kuchařky“)

V obou případech používáme následující zjednodušení:

- předpoklad $T(n) = \Theta(1)$ pro dostatečně malá n nepíšeme explicitně do rovnice
- zanedbáváme celočíselnost, tj. např. píšeme $n/2$ místo $\lceil n/2 \rceil$ nebo $\lfloor n/2 \rfloor$
- řešení nás většinou zajímá pouze asymptoticky (nehledíme na konkrétní hodnoty konstant) $\Rightarrow$ asymptotická notace používána už v zápisu rekurentní rovnice

Příklad: MergeSort $T(n) = 2T(n/2) + \Theta(n)$

Substituční metoda

- uhodnout asymptoticky správné řešení
- indukcí ověřit správnost odhadu (zvlášť pro dolní a horní odhad)

Příklad: opět MergeSort

Master theorem

Nechť $a \geq 1$, $c > 1$, $d \geq 0$ jsou reálná čísla a nechť $T : \mathbf{N} \rightarrow \mathbf{N}$ je neklesající funkce taková, že pro všechna n ve tvaru c^k (kde $k \in \mathbf{N}$) platí

$$T(n) = aT(n/c) + F(n)$$

kde pro funkci $F : \mathbf{N} \rightarrow \mathbf{N}$ platí $F(n) = \Theta(n^d)$. Označme $x = \log_c a$. Potom

a) je-li $x < d$, potom platí $T(n) = \Theta(n^d)$,

b) je-li $x = d$, potom platí $T(n) = \Theta(n^d \log n) = \Theta(n^x \log n)$,

c) je-li $x > d$, potom platí $T(n) = \Theta(n^x)$.

Příklady:

- MergeSort $T(n) = 2T(n/2) + \Theta(n)$
- Binární vyhledávání $T(n) = T(n/2) + \Theta(1)$
- Rovnice $T(n) = 9T(n/3) + \Theta(n)$
- Rovnice $T(n) = 3T(n/4) + \Theta(n^2)$
- Rovnice $T(n) = 2T(n/2) + \Theta(n \log n)$

Násobení čtvercových matic

Vstup: matice A a B řádu n x n

Výstup: matice $C = A \otimes B$ (také řádu n x n)

Klasický algoritmus

```
begin for i:=1 to n do
    for j:=1 to n do
    begin C[i,j] := 0;
          for k:=1 to n do C[i,j] := C[i,j] + A[i,k] * B[k,j]
    end
end
```

Časová složitost: $T(n) = \Theta(n^3)$ (n^2 skalárních součinů délky n)

Nyní předpokládejme že n je mocnina čísla 2 ($n=2^k$), což umožňuje opakované dělení matice na 4 matice polovičního řádu až do matic řádu 1 x 1 a zkusme „rozděl a panuj“ (předpoklad $n=2^k$ později odstraníme)

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

$$C_{11} = (A_{11} \otimes B_{11}) \oplus (A_{12} \otimes B_{21})$$

$$C_{12} = (A_{11} \otimes B_{12}) \oplus (A_{12} \otimes B_{22})$$

$$C_{21} = (A_{21} \otimes B_{11}) \oplus (A_{22} \otimes B_{21})$$

$$C_{22} = (A_{21} \otimes B_{12}) \oplus (A_{22} \otimes B_{22})$$

Každý skalární součin je „roztržen“ na dvě poloviny a „dokončen“ maticovým sčítáním.

Počet maticových operací na maticích řádu $n/2$: 8 násobení $\otimes$ a 4 sčítání $\oplus$

Počet sčítání (reálných čísel) v maticovém sčítání: $4(n/2)^2 = n^2$

Časová složitost: $T(n) = 8T(n/2) + \Theta(n^2)$

Master theorem: $a=8, c=2, \log_c a=3, d=2$ $T(n) = \Theta(n^3)$

(asymptoticky stejné jako klasický algoritmus)

Ke snížení složitosti je potřeba snížit $a=8$ a zachovat nebo jen mírně zvýšit $d=2$.

Strassenův algoritmus (1969)

Používá pouze 7 násobení submatic řádu n/2 (místo původních 8)

$$M_1 = (A_{12} \ominus A_{22}) \otimes (B_{21} \oplus B_{22})$$

$$M_2 = (A_{11} \oplus A_{22}) \otimes (B_{11} \oplus B_{22})$$

$$M_3 = (A_{11} \ominus A_{21}) \otimes (B_{11} \oplus B_{12})$$

$$M_4 = (A_{11} \oplus A_{12}) \otimes B_{22}$$

$$M_5 = A_{11} \otimes (B_{12} \ominus B_{22})$$

$$M_6 = A_{22} \otimes (B_{21} \ominus B_{11})$$

$$M_7 = (A_{21} \oplus A_{22}) \otimes B_{11}$$

Počet maticových operací řádu n/2: 7 násobení $\otimes$ a 10 sčítání $\oplus$ a odčítání $\ominus$

$$C_{11} = M_1 \oplus M_2 \ominus M_4 \oplus M_6$$

$$C_{12} = M_4 \oplus M_5$$

$$C_{21} = M_6 \oplus M_7$$

$$C_{22} = M_2 \ominus M_3 \oplus M_5 \ominus M_7$$

Počet maticových operací řádu n/2: 8 sčítání $\oplus$ a odčítání $\ominus$

Časová složitost: $T(n) = 7T(n/2) + \Theta(n^2)$

Master theorem: $a=7, c=2, \log_c a = \log_2 7 = x, d=2$ $\quad T(n) = \Theta(n^x) = \Theta(n^{2.81})$

Hledání k-tého z n prvků.

Vstup: neuspořádaná posloupnost n (různých) čísel

Výstup: k-té nejmenší číslo

Časovou složitost budeme pro jednoduchost měřit počtem porovnání.

Pro k = 1 (minimum) a k = n (maximum) jde triviálně pomocí n - 1 porovnání.

Pro k = n/2 (medián) ?????? (ukážeme, že je to stejně těžké jako pro obecné k)

První nápad: setřídit posloupnost, potom vybrat k-tý ⇒ časová složitost Ω(n log n)

Jde to lépe? ⇒ zkusíme „Rozděl a panuj“

- z posloupnosti vybereme prvek (pivot) a podle něj roztřídíme posloupnost na tři části a to na m prvků menších než pivot, vybraného pivota a (n-m-1) prvků větších než pivot
- na to je potřeba n-1 porovnání
- pokud k>m+1 tak — zahodíme m+1 malých prvků a hledáme (k-m-1)-ní prvek mezi (n-m-1) prvky většími než pivot
- pokud k=m+1 tak — pivot je hledaný prvek a končíme
- pokud k<m+1 tak — zahodíme n-m velkých prvků a hledáme k-tý prvek mezi m prvky menšími než pivot

Tohle ovšem může špatně dopadnout … pokud nezajistíme „dobrý“ výběr pivota.

Algoritmus (Blum et al. 1972)

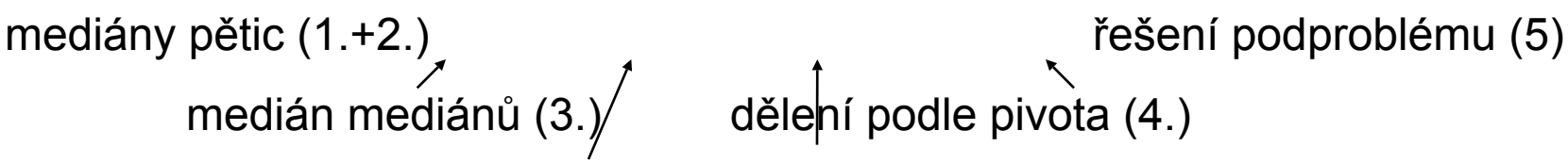
1. Rozděl posloupnost délky n na ⌈n/5⌉ pětic (poslední může být neúplná).
2. V každé pětici najdi její medián.
3. Rekurzivně najdi medián ze získané množiny ⌈n/5⌉ mediánů.
4. Použij medián mediánů jako pivot k rozdělení vstupní posloupnosti.
5. Pokud medián mediánů není hledaným prvkem, tak rekurzivně hledej v množině prvků menších než je on nebo v množině prvků větších než je on.

Jak „dobré“ je rozdělení podle pivota nelezeného výše uvedeným algoritmem?

Platí: Množina prvků menších než pivot i množina prvků větších než pivot každá obsahuje alespoň 3n/10 prvků ⇒ v bodě 5 iteruji s nejvýše 7n/10 prvků

Nechť: T(n) = počet porovnání použitý k nalezení k-tého z n prvků v nehorším případě

$$T(n) = 7n/5 + T(n/5) + (n-1) + T(7n/10)$$



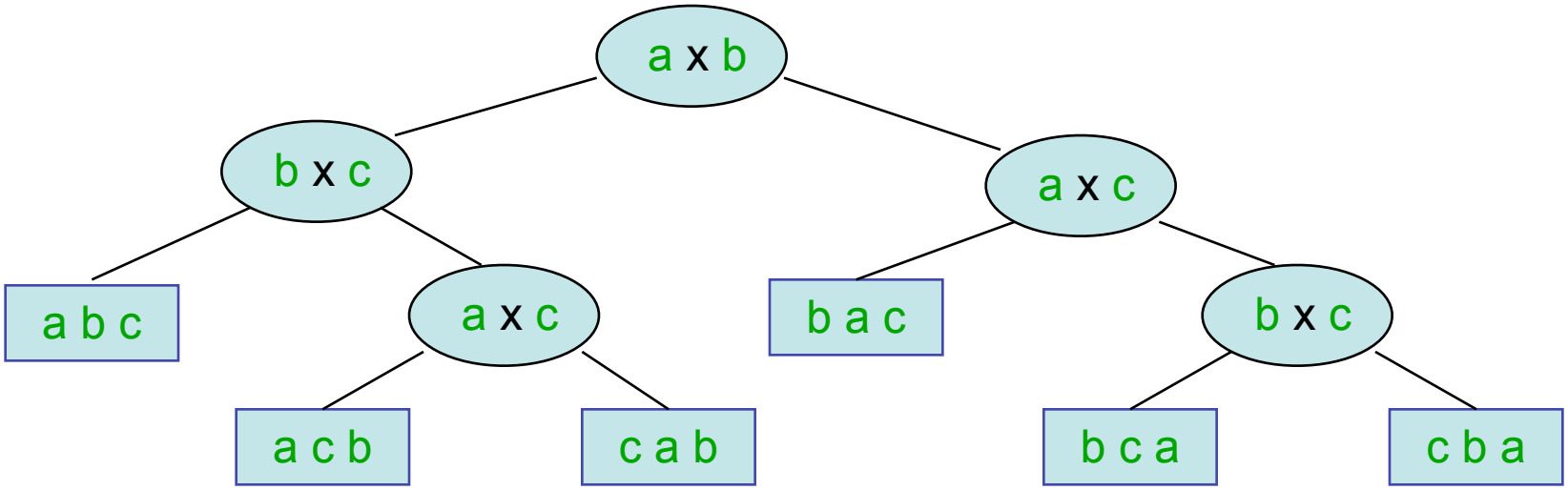
Tvrzení: T(n) = O(n)

Důkaz: substituční metodou (klíčový fakt: 1/5+7/10<1, což při dělení na trojice nevyjde)

Dolní odhad složitosti porovnávacích třídících algoritmů

Pozorování: každý (deterministický) třídící algoritmus založený na porovnávání (dvojic prvků) lze jednoznačně modelovat rozhodovacím stromem, což je binární strom, jehož vnitřní uzly odpovídají porovnáním a listy permutacím vstupní posloupnosti.

Příklad: rozhodovací strom pro Insertion-Sort a n=3 (označme vstup a,b,c)



levá větev vždy odpovídá výsledku "<" a pravá větev ">" (BÚNO vstupy po dvou různé)

Rozhodovací strom modeluje korektní třídící algoritmus ⇒ musí obsahovat listy se všemi n! možnými pořadími (permutacemi) vstupní posloupnosti.

Počet porovnání v nejhorším případě = nejdelší větev od kořene k listu = výška stromu.

Věta: Binární strom s n! listy má výšku Ω(n log n).

Základní grafové algoritmy

Značení: graf G=(V,E), V vrcholy, |V|=n, E hrany, |E|=m

neorientovaný graf: hrana = neuspořádaná dvojice vrcholů

orientovaný graf: hrana = uspořádaná dvojice vrcholů

Reprezentace grafů: matice sousednosti $\Theta(n^2)$

seznamy sousedů $\Theta(n+m)$

Prohledávání grafů

Prohledávání do šířky (BFS – breadth first search)

```
BFS(G,s)
for each u∈V do begin barva[u]:=bílá; d[u]:=Maxint; p[u]:=NIL end;
barva[s]:=šedá; d[s]:=0; Fronta:={s};
while Fronta neprázdná do
        u:=první ve Frontě;
        for each v je soused u do if barva[v]=bílá then
        begin barva[v]:=šedá; d[v]:=d[u]+1; p[v]:=u; v zařaď na konec Fronty end;
        barva[u]:=černá; vyhoď u z Fronty
```

Poznámky k BFS (opakování z přednášky Programování):

1. Prohledává graf po vrstvách podle vzdálenosti (měřeno počtem hran) od vrcholu s
2. Postupně navštíví všechny vrcholy dostupné z s a vytvoří strom nejkratších cest
3. Je základem složitějších algoritmů, např. Dijkstrova algoritmu (nejkratší cesty v grafu s nezápornými váhami na hranách) nebo Primova (Jarníkova) algoritmu (minimální kostra váženého grafu)
4. Funguje i na orientovaném grafu (beze změny)
5. Při zadání pomocí seznamů sousedů běží v čase $\Theta(n+m)$

Použití BFS: testování souvislosti neorientovaného grafu

- vybereme náhodně vrchol s a spustíme BFS z s
- pokud po ukončení BFS zůstane nějaký vrchol bílý: graf není souvislý
- spočítání počtu komponent souvislosti: opakované spouštění BFS z náhodně vybraného bílého vrcholu dokud nějaký bílý vrchol existuje
- opět běží v čase $\Theta(n+m)$

Prohledávání do hloubky (DFS – depth first search)

• neorientovaná verze viz přednáška z Programování – hlavní rozdíl proti BFS spočívá v tom, že aktivní (šedé) vrcholy nejsou ukládány do fronty ale do zásobníku, který je buď explicitně vytvářen algoritmem nebo implicitně rekurzivním voláním

• orientovaná verze: probereme podrobně, předpokládáme že graf je reprezentován pomocí seznamu sousedů

```
DFS(G)
begin    for i:=1 to n do barva[i]:=bílá;
         čas:=0;
         for i:=1 to n do if barva[i]=bílá then NAVŠTIV(i)
end;

NAVŠTIV(i) {jednoduchá verze}
begin    barva[i]:=šedá;
         čas:=čas+1;
         d[i]:=čas;
         for each j je soused i do if barva[j]=bílá then NAVŠTIV(j);
         barva[i]:=černá;
         čas:=čas+1;
         f[i]:=čas
end;
```

Klasifikace hran pro DFS na orientovaném grafu:

(i,j) je stromová	j byl objeven z i	při prohlížení (i,j) je j bílý
(i,j) je zpáteční	j je předchůdce i v DFS stromě	při prohlížení (i,j) je j šedý
(i,j) je dopředná	i je předchůdce j v DFS stromě (ale ne přímý rodič)	při prohlížení (i,j) je j černý a navíc $d(i) < d(j)$
(i,j) je příčná	nenastal ani jeden z předchozích tří případů	při prohlížení (i,j) je j černý a navíc $d(i) > d(j)$

Vlastnosti DFS

1. Stromové hrany tvoří orientovaný les (DFS les = množina DFS stromů)
2. Vrchol j je následníkem vrcholu i v DFS stromě $\Leftrightarrow$ v čase d(i) existovala z i do j cesta sestávající výlučně z bílých vrcholů
3. Intervaly [d(i), f(i)] tvoří „dobré uzávorkování“ tj. pro každé $i \neq j$ platí
   - buď $[d(j), f(j)] \cap [d(i), f(i)] = \emptyset$
   - nebo $[d(i), f(i)] \subset [d(j), f(j)]$ a i je následníkem j v DFS stromě
   - nebo $[d(j), f(j)] \subset [d(i), f(i)]$ a j je následníkem i v DFS stromě

   Důsledek: j je následníkem i v DFS stromě $\Leftrightarrow [d(j), f(j)] \subset [d(i), f(i)]$

```
NAVŠTIV(i) {plná verze}
begin   barva[i]:=šedá;
        čas:=čas+1;
        d[i]:=čas;
        for each j je soused i do
        if barva[j]=bílá
                then    begin   NAVŠTIV(j);
                                označ (i,j) jako stromovou
                        end
                else if barva[j]=šedá
                        then    begin   ohlas nalezení cyklu;
                                        označ (i,j) jako zpětnou
                                end
                        else if d(i) < d(j)
                                then označ (i,j) jako dopřednou
                                else označ (i,j) jako příčnou
        barva[i]:=černá;
        čas:=čas+1;
        f[i]:=čas
end;
```

Složitost: stále lineární $\Theta(n+m)$

Topologické číslování vrcholů orientovaného grafu

Definice: Funkce $t : V \rightarrow \{1,2, \ldots ,n\}$ je topologickým očíslováním množiny V pokud pro každou hranu $(i,j) \in E$ platí $t(i) < t(j)$.

Pozorování: topologické očíslování existuje pouze pro acyklické grafy

Hloupý algoritmus:

1. Najdi vrchol ze kterého nevede žádná hrana a přiřaď mu poslední volné číslo
2. Odstraň očíslovaný vrchol z grafu a pokud je graf neprázdný tak jdi na bod 1.

Složitost: $\Theta(n(n+m))$

Chytrý algoritmus: mírná modifikace DFS, běží v čase $\Theta(n+m)$

Lemma: G obsahuje cyklus $\Leftrightarrow$ DFS(G) najde zpětnou hranu

Věta: Očíslování vrcholů acyklického grafu G podle klesajících časů jejich opuštění (časy $f(i)$) je topologické.

Tranzitivní uzávěr orientovaného grafu

Definice: Orientovaný graf $G'=(V,E')$ je tranzitivním uzávěrem orientovaného grafu $G=(V,E)$ pokud pro každou dvojici vrcholů $i,j \in V$ takových, že $i \neq j$ platí

$$\text{z } i \text{ do } j \text{ vede v } G \text{ orientovaná cesta} \Rightarrow (i,j) \in E'$$

Tranzitivní uzávěr G' reprezentovaný maticí sousednosti = matice dosažitelnosti grafu G

Matici dosažitelnosti lze získat v čase $\Theta(n(n+m))$ pomocí n použití DFS

Silně souvislé komponenty orientovaného grafu

Definice: Nechť $G=(V,E)$ je orientovaný graf. Množina vrcholů $K \subseteq V$ se nazývá silně souvislá komponenta grafu G pokud

- Pro každou dvojici vrcholů $i,j \in K$ takových, že $i \neq j$ existuje v grafu G orientovaná cesta z i do j a orientovaná cesta z j do i. (1)
- Neexistuje množina vrcholů L která by byla ostrou nadmnožinou K a splňovala (1).

Hloupý algoritmus: vytvoříme tranzitivní uzávěr (matici dosažitelnosti) a z něj v čase $\Theta(n^2)$ „přečteme“ jednotlivé SSK

Chytrý algoritmus:

Vstup: orientovaný graf $G=(V,E)$ zadaný pomocí seznamů sousedů

Fáze 1: DFS(G) doplněné o vytvoření spojového seznamu vrcholů podle klesajících časů jejich opuštění

Fáze 2: vytvoření transponovaného grafu G^T

Fáze 3: DFS(G^T) modifikované tak, že vrcholy jsou v hlavním cyklu zpracovávány v pořadí podle seznamu vytvořeného ve Fázi 1 (místo podle čísel vrcholů)

Výstup: DFS stromy z Fáze 3 = silně souvislé komponenty grafu G

Definice: Nechť $G=(V,E)$ je orientovaný graf. Graf $G^T=(V,E^T)$, kde

$$(i,j) \in E^T \Leftrightarrow (j,i) \in E$$

se nazývá transponovaný graf ke grafu G.

Platí: Transponovaný graf lze zkonstruovat v čase $\Theta(n+m)$ a tím pádem celý algoritmus běží v čase $\Theta(n+m)$.

Lemma: Nechť $G=(V,E)$ je orientovaný graf a K je SSK v G. Po provedení DFS(G) platí:

1. množina K je podmnožinou vrcholů jediného DFS stromu
2. v daném DFS stromě tvoří množina K podstrom

Extremální cesty v (orientovaných) grafech

extremální cesta = nejkratší (nejdelší) cesta (záleží na kontextu)

graf bez vah na hranách: délka cesty = počet hran na cestě (lze nalézt pomocí BFS)

graf s váhami na hranách: označme

$G = (V,E)$ orientovaný graf

$w : E \rightarrow R$ váhová funkce

pokud $p = (v_0, v_1, \ldots, v_k)$ je orientovaná cesta (povolujeme opakování vrcholů), tak

$$w(p) = w(v_0,v_1) + w(v_1,v_2) + \ldots + w(v_{k-1},v_k)$$

Definice (váha nejkratší cesty z u do v)

$$\delta(u,v) = \begin{cases} \min \{ w(p) \mid p \text{ je cesta z } u \text{ do } v \} & \text{pokud } \exists \text{ cesta z } u \text{ do } v \\ \infty & \text{jinak} \end{cases}$$

Definice (nejkratší cesta z u do v)

Nejkratší cesta z u do v je libovolná cesta z u do v pro kterou platí $w(p) = \delta(u,v)$

Negativní cykly: negativní cyklus = orientovaný cyklus s celkovou negativní váhou

- Graf bez negativních cyklů: $\delta(u,v)$ definováno pro všechny dvojice vrcholů u a v a alespoň jedna nejkratší cesta je pro každou dvojici vrcholů prostá (bez cyklů)
- Graf s negativními cykly: pokud z u do v ∃ cesta obsahující negativní cyklus, tak dodefinujeme $\delta(u,v) = -\infty$

Nejkratší cesty z jednoho zdroje

Úloha: pro pevně zvolený vrchol $s \in V$ (zdroj) chceme spočítat $\delta(s,v)$ pro všechna $v \in V \setminus \{s\}$

Co nás čeká:

1. acyklický graf (a jakékoli váhy) → algoritmus DAG (algoritmus kritické cesty)
2. nezáporné váhy (a jakýkoli graf) → Dijkstrův algoritmus
3. *bez omezení (jakýkoli graf i váhy)* → *Bellman-Fordův algoritmus* (nás nečeká)

Triviální pozorování

Vlastnost 1 Pokud $p=(v_0,v_1, \dots ,v_k)$ je nejkratší cesta z v_0 do v_k, pak $\forall i,j : 0 \le i \le j \le k$ platí, že (pod)cesta $p_{ij}=(v_i, \dots ,v_j)$ je nejkratší cestou z v_i do v_j.

Vlastnost 2 Pokud je p nejkratší cestou z s do v a poslední hrana na p je $(u,v)\in E$, pak $\delta(s,v) = \delta(s,u) + w(u,v)$

Vlastnost 3 Pokud je $(u,v)\in E$ hrana, tak $\delta(s,v) \le \delta(s,u) + w(u,v)$.

Zpřesňování horních odhadů pro nejkratší cesty

Pro každý $v\in V$ budeme držet hodnotu $d(v)$, pro kterou bude platit invariant $d(v) \ge \delta(s,v)$.

```
Inicializace (G,s);
  for each v∈V(G) do
          begin     d(v) := ∞ ;
                    p(v) := NIL          {předchůdce na nejkratší cestě}
          end;
  d(s) := 0.
```

Po inicializaci se opakovaně (v nějakém pořadí) provádí přepočítávání odhadů:

```
Relax (u,v,w);
  if d(v) > d(u) + w(u,v) then
  begin   d(v) := d(u) + w(u,v);
          p(v) := u
  end.
```

Vlastnost 4 Pokud je $(u,v) \in E$ hrana, tak v okamžiku po provedení Relax (u,v,w) platí $d(v) \leq d(u) + w(u,v)$.

Vlastnost 5 Pokud byla provedena Inicializace (G,s), tak $\forall v \in V$ platí $d(v) \geq \delta(s,v)$ a tato nerovnost zůstane v platnosti po libovolné posloupnosti relaxačních kroků. Navíc pokud hodnota d(v) klesne až na hodnotu $\delta(s,v)$, tak už se v dalším průběhu nezmění.

Vlastnost 6 Pokud z s do v nevede orientovaná cesta, tak od Inicializace (G,s) dál platí $d(v) = \delta(s,v) = \infty$.

Vlastnost 7 Nechť je p nejkratší cesta z s do v a poslední hrana na p je (u,v). Nechť je provedena Inicializace (G,s) a po ní posloupnost relaxačních kroků, která obsahuje volání Relax (u,v,w). Pak pokud $d(u) = \delta(s,u)$ platí v nějaký okamžik před zavoláním Relax (u,v,w), tak $d(v) = \delta(s,v)$ platí v jakémkoli okamžiku po zavolání Relax (u,v,w).

Algoritmus DAG (directed acyclic graph) = algoritmus kritické cesty

```
DAG (G,w,s);
  topologicky setřiď vrcholy grafu G;
  Inicializace (G,s);
  for each (u∈V(G) v topologickém pořadí) do
        for each (v∈V(G) takové že (u,v) ∈E(G)) do Relax (u,v,w)
```

Věta: Nechť $G=(V,E)$ je acyklický vážený orientovaný graf a $s \in V(G)$ libovolný vrchol. Pak po ukončení procedury DAG (G,w,s) pro každý vrchol $v \in V(G)$ platí $d(v) = \delta(s,v)$.

Časová složitost: Celý algoritmus běží v $\Theta(n+m)$ protože

- topologické očíslování (setřídění) trvá $\Theta(n+m)$
- vlastní algoritmus trvá $\Theta(1)$ na vrchol a $\Theta(1)$ na hranu, tj. celkem také $\Theta(n+m)$

Aplikace: Acyklický graf, kde (hrany = činnosti) a (váhy = doby trvání činnosti). Graf vyjadřuje závislosti mezi činnostmi, každá orientovaná cesta odpovídá činnostem které musí být prováděny jedna po druhé. Snažíme se najít kritickou cestu, tzn. cestu v grafu s největším možným součtem (každé zpoždění činnosti na kritické cestě způsobí zpoždění celého projektu).

Řešení: V algoritmu DAG buď

- všem vahám otočíme znaménka nebo
- v Inicializace (G,s) zaměníme ∞ za $-\infty$ a v Relax (u,v,w) otočíme nerovnost

Dijkstrův algoritmus

- předpoklad: všechny váhy na hranách jsou nezáporné ($\forall (u,v) \in E$ platí $w(u,v) \geq 0$)
- všechny vrcholy jsou během práce algoritmu rozděleny do dvou množin

a) vrchol v patří do S pokud je jeho nejkratší vzdálenost od zdroje s již spočítána, takže platí $d(v) = \delta(s,v)$ – na začátku (po Inicializace (G,s)) platí $S = \emptyset$

b) v opačném případě patří v patří do $Q = V \setminus S$ kde Q je implementována jako datová struktura podporující vyhledávání vrcholu v s minimálním d(v)

```
Dijkstra (G,w,s);
  Inicializace (G,s);
  S := ∅; Q := V(G);
  while (Q ≠ ∅) do
     u := Extract-Min (Q);
     S := S ∪ {u};
     for each (v∈V(G) takové že (u,v) ∈E(G)) do Relax (u,v,w)
```

Věta: Nechť G=(V,E) je vážený orientovaný graf s nezápornými váhami na hranách a nechť $s \in V(G)$ je libovolný vrchol. Pak po ukončení procedury Dijkstra (G,w,s) pro každý vrchol $v \in V(G)$ platí $d(v) = \delta(s,v)$.

Časová složitost: $\Theta(n^2)$ pokud je Q implementováno jako pole

$\Theta((n+m)\log n)$ pokud je Q implementováno jako binární halda

Vyhledávání řetězců v textu

Σ konečná abeceda (množina znaků)

Σ^* množina slov nad abecedou Σ (konečné posloupnosti znaků)

délka slova : $u = x_1x_2 \dots x_m \in \Sigma^* \Rightarrow length(u) = m$ (počet znaků ve slově)

skládání (konkatenace) slov u a v :

$u = x_1x_2 \dots x_m$, $v = y_1y_2 \dots y_n \Rightarrow uv = x_1x_2 \dots x_my_1y_2 \dots y_n$

(a samozřejmě $length(uv) = length(u) + length(v)$)

prázdné slovo ε ($\forall u \in \Sigma^*$ platí $u\varepsilon = \varepsilon u = u$)

předpona (prefix): $u \in \Sigma^*$ je předponou $v \in \Sigma^*$ pokud $\exists w \in \Sigma^* : uw = v$

přípona (sufix): $u \in \Sigma^*$ je příponou $v \in \Sigma^*$ pokud $\exists w \in \Sigma^* : wu = v$

pokud $w \neq \varepsilon$ tak se jedná o vlastní předponu (příponu)

Řešená úloha:

vstup: abeceda Σ, prohledávaný text $x = x_1x_2 \dots x_n \in \Sigma^*$ a hledané vzorky $K = \{y_1, y_2, \dots, y_k\}$, kde $y_p = y_{p,1} \dots y_{p,length(p)} \in \Sigma^*$ pro $p = 1, \dots, k$

výstup: všechny výskyty vzorků v x, tj. všechny dvojice $[i,p]$ takové, že y_p je příponou slova $x_1x_2 \dots x_i$ ($1 \le i \le n$ a $1 \le p \le k$)

Naivní algoritmus

```
for p := 1 to k do
  for i := 1 to (n – length(p) + 1) do
  begin  j := 0;
         while (j < length(p)) and (x_{i+j} = y_{p,1+j}) do j := j + 1;
         if (j = length(p)) then Report(i,p)
  end
```

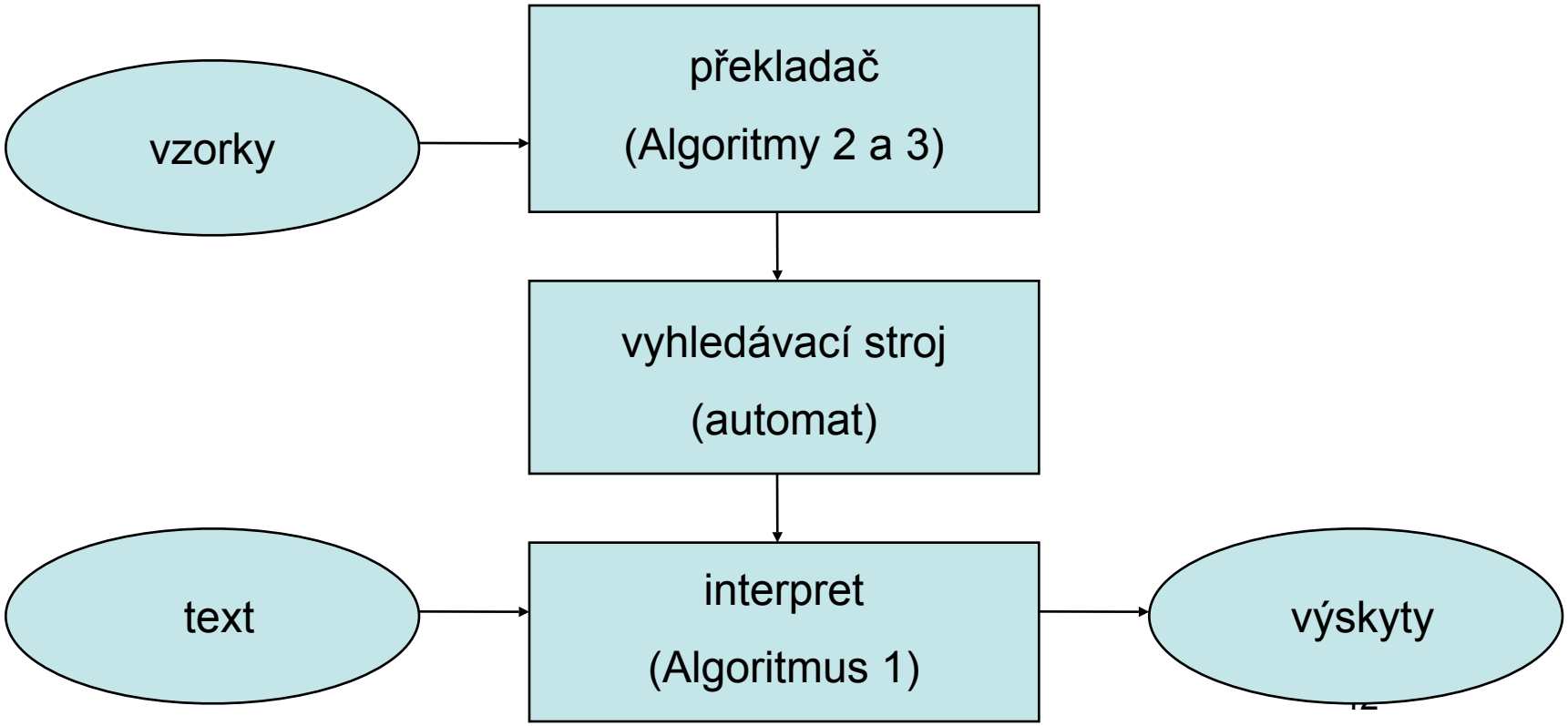
Algoritmus na míru

```
c := 0;
for i := 1 to n do
begin
  if (x_i = a) then c := c + 1
          else begin
                 if (c ≥ h – 1) then Report(i,1) ; c := 0
               end
end
```

Ukážeme, že algoritmus na míru (= konečný automat) lze vyrobit pro libovolný vzorek nebo množinu vzorků, a to tak, že:

- výroba automatu (vyhledávacího stroje) trvá $\Theta(h \cdot |\Sigma|)$
- vyrobený automat prohlédne text za $\Theta(n)$
- celková práce algoritmu je $\Theta(n + h \cdot |\Sigma|)$

Algoritmus Aho-Corasick(ová) (1975)



Vyhledávací stroj pro konečnou abecedu Σ a množinu vzorků K je čtveřice

$M = (Q, g, f, out)$, kde

1. $Q = \{0,1, \ldots ,q\}$ je množina stavů
2. $g : Q \times \Sigma \rightarrow Q \cup \{\perp\}$ je přechodová funkce, pro kterou platí $\forall x \in \Sigma: g(0,x) \in Q$
   (symbol $\perp$ znamená „nedefinováno“, přechod ze stavu 0 je definován $\forall x \in \Sigma$)
3. $f : Q \rightarrow Q$ je zpětná funkce, pro kterou platí $f(0) = 0$ (nastupuje pokud g dá $\perp$)
4. $out : Q \rightarrow P(K)$ je výstupní funkce (pro daný stav vydá podmnožinu vzorků)

Algoritmus 1 (interpret vyhledávacího stroje)

vstup: $x = x_1 \ldots x_n \in \Sigma^*$, $K = \{y_1 \ldots y_k\}$, $M = (Q, g, f, out)$

```
state := 0;
for i := 1 to n do
begin
    (1)  while (g(state,x_i) = ⊥) do state := f(state);
    (2)  state := g(state,x_i);
    (3)  for all y_p ∈ out(state) do Report (i,p)
end
```

Klíčové vlastnosti vyhledávacího stroje (konečného automatu):

1. přechodová funkce g

   graf funkce g (pro definované dvojice bez smyčky ve stavu 0) je ohodnocený strom, pro který

   - stav 0 je kořenem stromu
   - každá cesta z kořene je ohodnocena nějakou předponou nějakého vzorku z K
   - každá předpona každého vzorku z K ohodnocuje cestu z kořene do nějakého (právě jednoho) stavu s $\Rightarrow$ říkáme, že předpona (slovo) u reprezentuje stav s (speciálně prázdné slovo ε reprezentuje stav 0)
   - hloubka stavu s reprezentovaného slovem u je definována jako d(s) = length(u) a pro funkci g (na hranách stromu) platí: $d(g(s,x_i)) = d(s) + 1$

2. zpětná funkce f

   pro každý stav s reprezentovaný slovem u platí, že stav f(s) je reprezentován nejdelší vlastní příponou slova u, která je zároveň předponou nějakého vzorku z K

3. výstupní funkce out

   pro každý stav s reprezentovaný slovem u a pro každý vzorek $y_p \in K$ platí:

   $y_p \in out(s)$ tehdy a jen tehdy když je y příponou slova u

Algoritmus 2 (konstrukce vyhledávacího stroje – 1.fáze)

```
vstup:    K = {y_1 … y_k}                 {množina vzorků}
výstup:   Q = {0, … ,q}                   {množina stavů vyhledávacího stroje}
          g : Q x Σ → Q ∪ {⊥}              {přechodová funkce splňující Vlastnost 1}
          o : Q → P(K)                    {„polotovar“ výstupní funkce out}
procedure Enter(y_p,1 … y_p,m);           {připojení slova y_p ke grafu funkce g}
begin     stav := 0; i:= 1;
          while (i <= m) and (g(stav,y_p,i) <> ⊥) do
            begin   stav := g(stav,y_p,i);        {pohyb po již hotové větvi}
                    i := i + 1                    {posun ve slově y_p}
            end;
          while (i <= m) do
            begin   Q := Q ∪ {q+1}; q := q+1      {vytvoření nového stavu}
                    for all x ∈ Σ do g(q,x) := ⊥;
                    g(stav,y_p,i) := q;           {prodloužení větve}
                    stav := q;                    {pokročení do nového stavu}
                    i := i + 1                    {posun ve slově y_p}
            end;
          o(stav) := {y_p}
end;      {of Enter}
begin     Q := {0}; for all x ∈ Σ do g(0,x) := ⊥;        {hlavní program}
          for p := 1 to k do Enter(y_p);
          for all x ∈ Σ do if g(0,x) = ⊥ then g(0,x) = 0
end.
```

Algoritmus 3 (konstrukce vyhledávacího stroje – 2.fáze)

```
vstup:   Q = {0, … ,q}                 {množina stavů vyhledávacího stroje}
         g : Q x Σ → Q ∪ {⊥}            {přechodová funkce splňující Vlastnost 1}
         o : Q → P(K)                   {„polotovar“ výstupní funkce out}
výstup:  f : Q → Q                      {zpětná funkce splňující Vlastnost 2}
         out : Q → P(K)                 {výstupní funkce splňující Vlastnost 3}
vytvoř prázdnou frontu stavů;
f(0) := 0; out(0) := ∅;
for all x ∈ Σ do begin                  {zpracuje potomky kořene}
                    s := g(0,x);
                    if s <> 0 then
                    begin    f(s) := 0; out(s) := o(s);
                             zařaď s na konec fronty
                    end
                 end;
while fronta není prázdná do
begin    r := první prvek z fronty (a vyřaď r z fronty);
         for all x ∈ Σ do if g(r,x) <> ⊥ then    {zpracuje potomky uzlu r}
         begin    s := g(r,x); t := f(r);
                  while g(t,x) = ⊥ then t := f(t);
                  f(s) := g(t,x); out(s) := o(s) ∪ out(f(s));
                  zařaď s na konec fronty
         end
end
```

Algoritmus Knuth-Morris-Pratt

- zjednodušená verze algoritmu Aho-Corasick(ová) pro vyhledávání jediného vzorku
- kratší a snadněji pochopitelný popis
- (mírně) lepší asymptotická složitost ($\Theta(n + h)$ místo $\Theta(n + h \cdot |\Sigma|)$)
- graf přechodové funkce g není strom ale řetězec, což umožňuje g explicitně vůbec nepoužívat (zde je ta úspora ve složitosti preprocessingu, protože g má $h \cdot |\Sigma|$ přechodů), funkce g je používána pouze implicitně
- zpětná funkce f se zde nazývá prefixová funkce a protože v případě jediného vzorku odpovídá číslo stavu délce prefixu daného vzorku, který je daným stavem reprezentován, tak má f jednoduší definici:

  f(s) je délka nejdelší vlastní přípony slova reprezentovaného stavem s (toto slovo je prostě předpona délky s daného vzorku), která je zároveň předponou (daného vzorku)

- výstupní funkce je triviální, ve stavu h hlásí výskyt (jediného) vzorku, jinde nic

procedura Prefix (nahrazuje Algoritmy 2 a 3)

vstup: $K = \{y\}$ {jediný vzorek}
výstup: $f : Q \rightarrow Q$ {prefixová funkce}

```
f(1) := 0;
t := 0;
for q := 2 to h do
begin   while (t > 0) and (y_{t+1} <> y_q) do t := f(t);
        if (y_{t+1} = y_q) then t := t +1;
        f(q) := t
end
```

Algoritmus KMP (nahrazuje Algoritmus 1)

vstup: $x = x_1 \ldots x_n \in \Sigma^*$, $K = \{y\}$, prefixová funkce f

```
state := 0;
for i := 1 to n do
begin
        (1)  while (state > 0) and (y_{state+1} <> x_i) do state := f(state);
        (2)  if y_{state+1} = x_i then state := state + 1;
        (3)  if (state = h) then begin Report (i);
                                       state := f(state)
                                 end
end
```

Dosažitelnost v grafu násobením matic

Vstup: Orientovaný graf $G=(V,E)$, $|V|=n$, $|E|=m$

Úloha: Výpočet tranzitivního uzávěru.

Víme: Matici dosažitelnosti lze získat v čase $\Theta(n(n+m))$ pomocí n použití DFS.

Mějme graf reprezentovaný maticí sousednosti A_G (s 1 na diagonále), potom:

- $(A_G \otimes A_G)$ reprezentuje matici „dosažitelnosti (nejvýš) na 2 kroky"
- $(A_G \otimes A_G \otimes A_G)$ reprezentuje matici „dosažitelnosti (nejvýš) na 3 kroky"
- $(A_G)^n$ reprezentuje matici „dosažitelnosti (nejvýš) na n kroků", tedy matici sousednosti tranzitivního uzávěru.

To vše za předpokladu, že

- místo násobení celých čísel použijeme logickou konjunkci (tedy and), a
- místo sčítání celých čísel použijeme logickou disjunkci (tedy or).

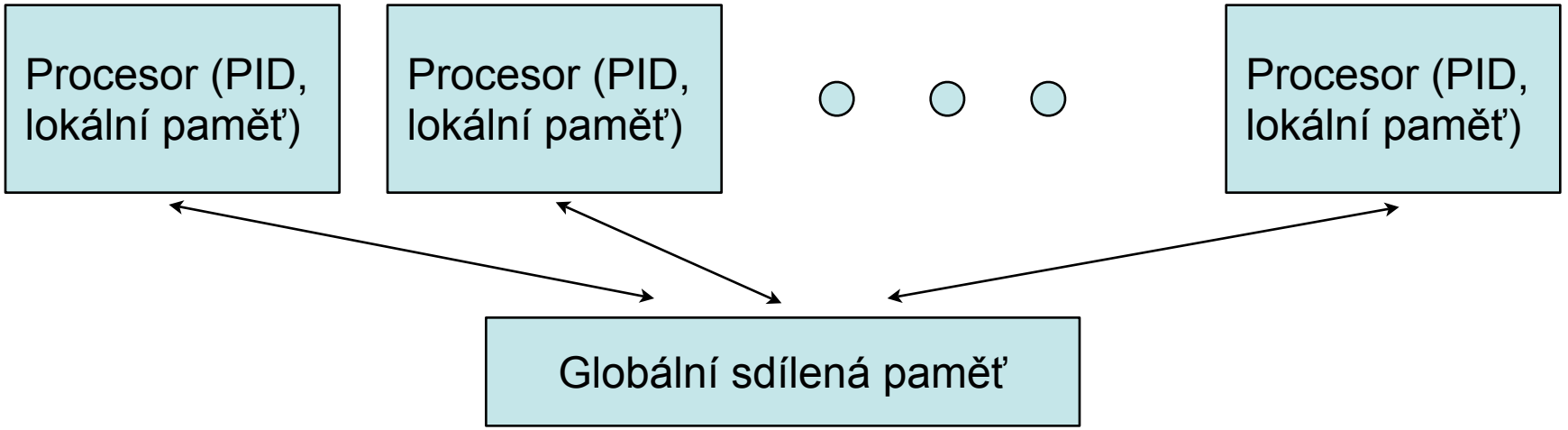
Násobení matic je asociativní, proto stačí $\log n$ násobení, dohromady dostaneme čas $\Theta(n^3 \log n)$, což je horší, ale lze dobře paralelizovat, protože

1. všechny prvky výsledné matice lze vyhodnocovat současně a

2. u skalárního součinu délky n lze vyhodnocovat současně všech n součinů.

=> Pomocí n^3 procesorů lze vynásobit dvě matice řádu n v konstantním čase.

Model paralelního stroje PRAM



- Každý procesor má své PID, které zná.
- Jednotlivé procesory spolu komunikují přes sdílenou paměť.
- Všechny procesory se řídí týmž algoritmem.
- Procesory pracují paralelně a synchronizovaně, tj. 1 krok PRAMU = 1 instrukce každého procesoru.
- V 1 kroku PRAMu může z téže buňky globální paměti číst libovolný počet procesorů (CR = concurrent read).
- V 1 kroku PRAMu může na tutéž buňku globální paměti zapisovat libovolný počet procesorů, pokud zapisují touž hodnotu (CW = concurrent write).

Paralelní výpočet dosažitelnosti tranzitivního uzávěru

Myšlenka: Každý z n^3 procesorů interpretuje svůj PID jako trojici $0 \le i, j, k \le n-1$.
Tj. $PID = in^2 + jn + k$.

Realizace: Procesory se shodnými souřadnicemi i a j počítají prvek $A[i, j]$, přičemž procesor se souřadnicemi i, j, k vyhodnocuje výraz $A[i, k]*A[k, j]$.

Algoritmus (paralelní mocnění booleovských matic)

vstup: A: array [1..n, 1..n] of boolean; {matice sousednosti G ve sdílené paměti}

výstup: v A je uložena mocnina A^l pro $l \ge n$. {matice dosažitelnosti tranz. uzávěru}

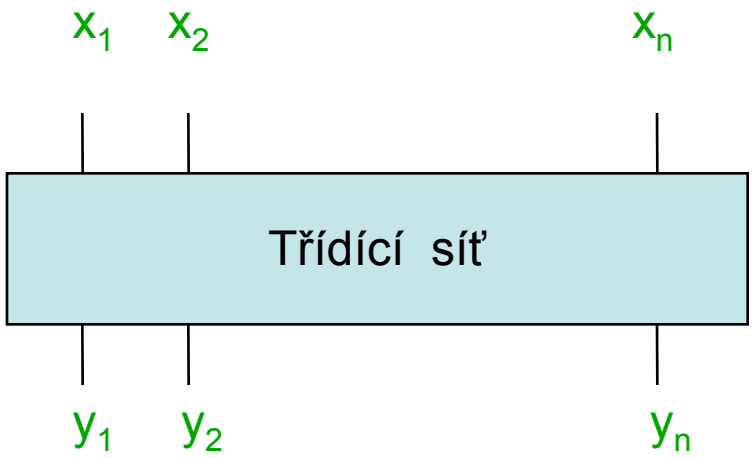
```
begin
   i:=PID div n²; j:=(PID mod n²) div n; k:=(PID mod n);
   l:=1; {současná mocnina}
   while l<n-1 do
   begin
      if A[i, k]&A[k, j] then A[i, j]:=true;
      l:=l*2;
   end;
end
```

Paralelní výpočet matice dosažitelnosti

- Je třeba zabezpečit stejný počet instrukcí i při nesplnění podmínky if, aby byla práce synchronizovaná.
- Zapisuje se pouze true, tedy vždy totéž.
- Cyklus se opakuje $\Theta(\log n)$ krát, celkový parelelní čas je tedy týž.
- Počet procesorů je $\Theta(n^3)$.
- Paralelizovat lze i DFS, ale jen pomocí n procesorů, z nichž každý realizuje jedno DFS, jež nemohou pracovat na týchž datech, tj. vyžadují větší lokální paměť a celkový čas je lineární.

Třídící sítě

Třídící síť je obvod který má n vstupů z hodnotami z nějakého lineárně uspořádaného typu (tj.každé dvě hodnoty jsou porovnatelné) a n výstupů, na kterém jsou vstupní hodnoty setříděné (bez ohledu na to v jakém pořadí přišly na vstup).



Tento obvod obsahuje jediný typ hradla a sice komparátor, což je hradlo se dvěma vstupy x_1 a x_2 a dvěma výstupy y_1 a y_2, pro které platí $y_1=\min\{x_1,x_2\}$ a $y_2=\max\{x_1,x_2\}$.

Formální definice třídící sítě:

- $K = \{K_1, K_2, \dots, K_s\}$ je množina komparátorů, s se pak nazývá velikost sítě
- $O = \{ (k,i) \mid 1 \le k \le s, 1 \le i \le 2 \}$ je množina výstupů (k je číslo komparátoru a i výstupu)
- $I = \{ (k,i) \mid 1 \le k \le s, 1 \le i \le 2 \}$ je množina vstupů
- $C = (K, f)$ je třídící síť, kde $f : O \rightarrow I$ je částečné prosté zobrazení

Podmínka acyklicity sítě:

Požadujeme aby orientovaný graf $G = (K,E)$ kde $(Ku,Kv) \in E$ pokud existují i a j takové, že $f(u,i) = (v,j)$, byl acyklický.

Rozdělení komparátorů do hladin:

- Definujme $L_1 = \{ K_i \mid K_i$ má v G vstupní stupeň nula$\}$ (L_1 je neprázdná díky acyklicitě)
- Nechť jsou definovány $L_1, L_2, \ldots, L_h$, kde $L = L_1 \cup L_2 \cup \ldots \cup L_h \not\subset K$. Pak definujme $L_{h+1} = \{ K_i \mid K_i$ má v $G \setminus L$ vstupní stupeň nula$\}$ (L_{h+1} je neprázdná díky acyklicitě)
- Počet hladin značíme d a nazýváme hloubkou sítě

Práce sítě:

- čas 0 : definovány vstupy sítě (kam patří vstupy všech komparátorů v L_1)

  pracují komparátory v L_1
- čas 1 : definovány vstupy všech komparátorů v L_2

  pracují komparátory v L_2

  …
- čas d-1 : definovány vstupy všech komparátorů v L_d

  pracují komparátory v L_d
- čas d : definovány všechny výstupy sítě

Pozorování: časová složitost třídění odpovídá hloubce sítě (to je tedy klíčový parametr)

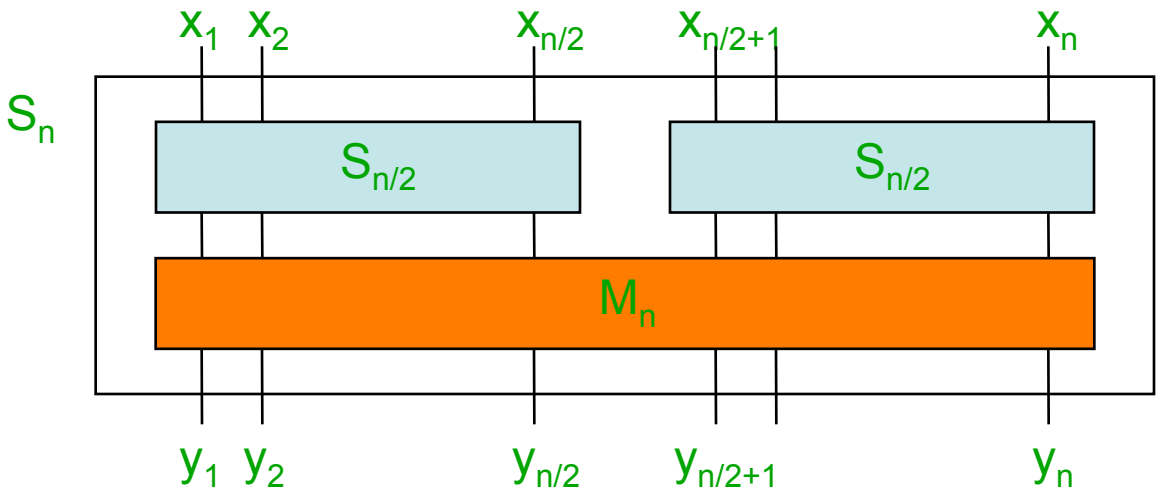
Topologicky jiná reprezentace sítě:

- „dráty“ ze vstupu x_i do výstupu y_i nakresleny jako přímky
- jednotlivé komparátory „roztaženy“ mezi příslušné „dráty“
- každá síť jde takto překreslit
- počtu vstupů/výstupů (drátů) říkáme šířka sítě

Merge-Sort implementovaný třídící sítí

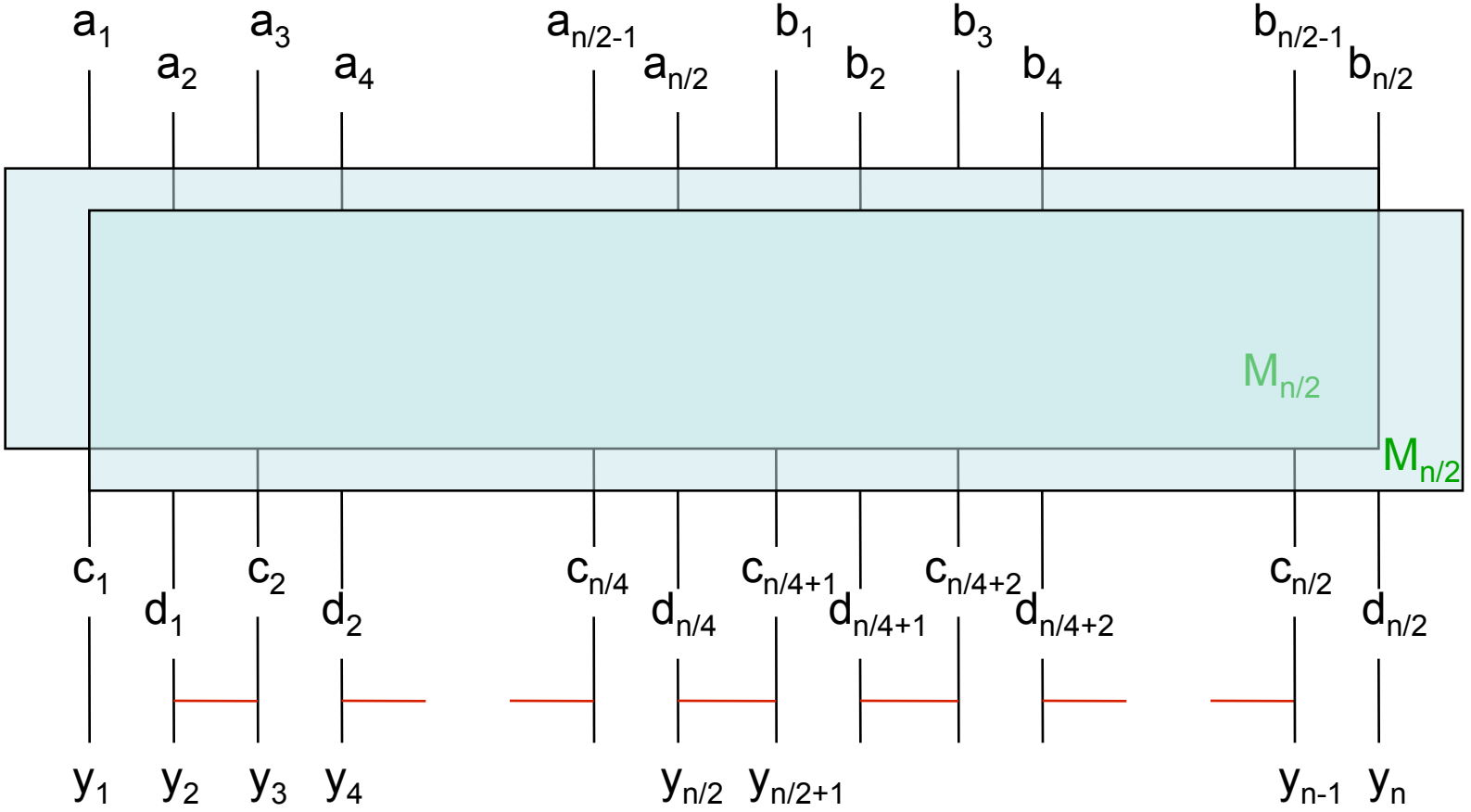
Chceme setřídit $x_1, x_2, \ldots, x_n$ (předpokládáme že n je mocnina dvojky)

Realizujeme to sítí S_n, která je rekurzivně definována následujícím obrázkem



kde M_n je slučovací (slévací) síť šířky n (rekurze se zastaví pro $n=2$)

Zbývá ukázat jak zkonstruovat slučovací síť M_n (opět jde o rekurzivní konstrukci) :

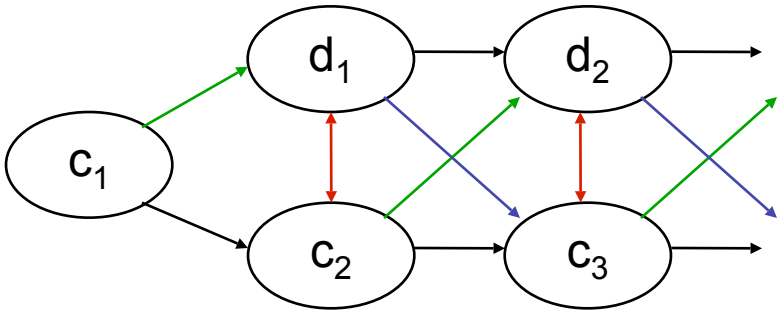


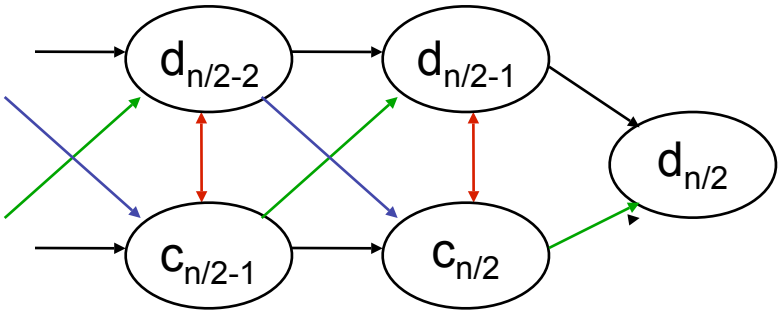
Liché členy obou setříděných posloupností jsou vstupem jedné kopie $M_{n/2}$ a sudé členy jsou vstupem druhé kopie $M_{n/2}$. Navíc jsou výstupy obou sítí propojeny jednou hladinou komparátorů dle obrázku (červené komparátory). Rekurze se opět zastaví pro $n=2$.

Pro vstup platí: $a_1 \le a_2 \le \ldots \le a_{n/2}$ a $b_1 \le b_2 \le \ldots \le b_{n/2}$

Indukční předpoklad: $c_1 \le c_2 \le \ldots \le c_{n/2}$ a $d_1 \le d_2 \le \ldots \le d_{n/2}$

Dokážeme, že: $y_1 \le y_2 \le \ldots \le y_n$





Černé nerovnosti (šipky) víme, zelené nerovnosti a modré nerovnosti (šipky) dokážeme. Bez ohledu to, jak dopadne porovnání jednotlivými červenými komparátory, budou šipky generovat lineární uspořádání, které bude správným uspořádáním výstupních hodnot.

Hloubka a velikost třídící sítě šířky $n = 2^k$

1.Slučovací síť M_n

má hloubku (počet hladin) $d(M_n) = \log_2 n$

a velikost (počet komparátorů) $s(M_n) = n/2 \log_2(n/2) + 1$

2.Třídící síť S_n

má hloubku (počet hladin) $d(S_n) = 1/2 \log_2 n (\log_2 n+1)$

a velikost (počet komparátorů) $s(S_n) = 1/4 \, n \log_2 n (\log_2 n–1) + (n–1)$

Dolní odhad složitosti třídění pomocí transpozičních sítí

Ať C je třídící síť s n vstupy, $s(C)$ komparátory a hloubkou $d(C)$.

Z C můžeme sestrojit sekvenční algoritmus pro třídění, který používá přesně $s(C)$ porovnání, algoritmus prostě simuluje síť C.

Víme, že algoritmus pro třídění pomocí porovnání potřebuje $\Omega(n \log n)$ porovnání.

Z toho plyne, že $s(C) = \Omega(n \log n)$.

Protože v každé hladině může být nejvýš $n/2$ komparátorů, musí platit, že $d(C) = \Omega(\log n)$.

Třídy P a NP, převoditelnost problémů, NP úplnost

Úloha: pro dané zadání najít strukturu s danými vlastnostmi

Příklady:

• v daném orientovaném grafu najdi cyklus

• vynásob dvě dané čtvercové matice

Optimalizační úloha: pro dané zadání najít optimální (většinou nejmenší nebo největší) strukturu s danými vlastnostmi

Příklady:

• v daném neorientovaném grafu najdi největší (počtem vrcholů) úplný podgraf (kliku)

• pro danou množinu úkolů najdi nejkratší rozvrh

Rozhodovací problém: pro dané zadání odpovědět ANO/NE

Příklady:

• existuje v daném neorientovanám grafu Hamiltonovská kružnice?

• je daná čtvercová matice regulární?

My se v následujícím omezíme jen na rozhodovací problémy, což lze (více méně) udělat bez újmy na obecnosti - v tom smyslu, že k většině (optimalizačních) úloh existuje „stejně těžký“ rozhodovací problém.

Definice (vágní): Třída P je třída rozhodovacích problémů, pro které existuje (deterministický sekvenční) algoritmus běžící v polynomiálním čase (vzhledem k velikosti zadání), který správně rozhodne ANO/NE (který řeší daný problém).

- je daný orientovaný graf silně souvislý?
- obsahuje daný neorientovaný graf trojúhelník? (speciální případ „kliky“)
- je daná matice regulární?

Nedeterministický algoritmus = algoritmus, který v každém svém kroku může volit z několika možností

Nedeterministický algoritmus řeší daný rozhodovací problém ⇔ pro každé kladné zadání problému (odpověď ANO) existuje posloupnost voleb vedoucí k tomu, že algoritmus odpoví ANO, pro žádné záporné zadání taková posloupnost voleb neexistuje.

Definice (vágní): Třída NP je třída rozhodovacích problémů, pro které existuje nedeterministický sekvenční algoritmus běžící v polynomiálním čase (vzhledem k velikosti zadání), který řeší daný problém.

Jiný model nedeterministického algoritmu: dopředu provede volby (do paměti zapíše vektor čísel) a pak už provádí jednotlivé kroky původního algoritmu deterministicky.

Alternativní definice (opět vágní): Rozhodovací problém patří do třídy NP, pokud pro každé jeho kladné zadání existuje (polynomiálně velký) certifikát, pomocí něhož lze v polynomiálním čase (deterministicky) ověřit, že zadání je skutečně kladné (že odpověď na dané zadání je skutečně ANO).

Příklady problémů ze třídy NP:

- KLIKA (úplný podgraf): Je dán neorientovaný graf G a číslo k.

  Otázka: Existuje v G úplný podgraf velikosti alespoň k?

- HK (Hamiltonovská kružnice): Je dán neorientovaný graf G.

  Otázka: Existuje v G Hamiltonovská kružnice?

- TSP (obchodní cestující): Je dán ohodnocený úplný neorientovaný graf G a číslo k.

  Otázka: Existuje v G Hamiltonovská kružnice celkové délky nejvýše k?

- SP (součet podmnožiny): Jsou dána přirozená čísla $a_1, a_2, \ldots, a_n, b$.

  Otázka: Existuje podmnožina čísel $a_1, a_2, \ldots, a_n$, jejíž součet je přesně b?

- ROZ (rozvr. na paralel. strojích): Je dán počet úkolů, jejich délky, počet strojů a číslo k.

  Otázka: Existuje přípustný rozvrh délky nejvýše k?

- SAT (splnitelnost Booleovských formulí): Je dána formule na n 0-1 proměnných v KNF.

  Otázka: Existuje (pravdivostní) ohodnocení proměnných pro které má daná formule hodnotu 1?

Ukážeme, že HK → TSP, SP → ROZ a SAT → KLIKA, kde A → B znamená, že pokud existuje polynomiální algoritmus řešící B potom také existuje polynomiální algoritmus řešící A, neboli vyřešit B je alespoň tak „těžké“ jako vyřešit A.

Převody (redukce) mezi rozhodovacími problémy

Nechť A,B jsou dva rozhodovací problémy. Říkáme, že A je polynomiálně redukovatelný na B, pokud existuje zobrazení f z množiny zadání problému A do množiny zadání problému B s následujícími vlastnostmi:

1. Nechť X je zadání problému A a Y zadání problému B takové, že Y = f(X). Potom je X kladné zadání problému A tehdy a jen tehdy, když je Y kladné zadání problému B.

2. Nechť X je zadání problému A. Potom je zadání f(X) problému B (deterministicky sekvenčně) zkonstruovatelné v polynomiálním čase vzhledem k velikosti X.

Poznámka: Z 2. také vyplývá, že velikost f(X) je polynomiální vzhledem k velikosti X.

NP-úplnost

Definice: Problém B je NP-těžký pokud pro libovolný problém A ze třídy NP platí, že A je polynomiálně redukovatelný na B.

Definice: Problém B je NP-úplný pokud 1) patří do třídy NP a 2) je NP-těžký.

Důsledek 1: Pokud je A NP-těžký a navíc je polynomiálně redukovatelný na B, tak je B také NP-těžký.

Důsledek 2: Pokud existuje polynomiální algoritmus pro nějaký NP-těžký problém, pak existují polynomiální algoritmy pro všechny problémy ve třídě NP.

Věta (Cook-Levin 1971): SAT je NP-úplný.

Aproximační algoritmy

Aprox. algoritmy jsou vhodné tam, kde je nalezení optimálního řešení „beznadějné“ (časově příliš náročné), typicky u NP-těžkých optimalizačních úloh (optimalizačních verzí NP-úplných rozhodovacích problémů). Mají následující tři vlastnosti:

1. konstruují suboptimální řešení

2. poskytují odhad kvality zkonstruovaného řešení vzhledem k optimu

3. běží v polynomiálním čase (jinak nejsou zajímavé)

Příklad maximalizační úlohy (optimalizační verze KLIKY):

Pro daný neorientovaný graf najdi největší (počtem vrcholů) kliku (úplný podgraf).

Po aproximačním algoritmu chceme garanci typu $f(APROX) \geq \frac{3}{4} f(OPT)$, kde $f(X)$ je v tomto případě počet vrcholů (tj. velikost kliky) v řešení X, OPT je optimální řešení a $APROX$ je řešení vydané aproximačním algoritmem.

Příklad minimalizační úlohy (optimalizační verze ROZ):

Pro dané úkoly a daný počet strojů najdi nejkratší rozvrh.

Po aproximačním algoritmu chceme garanci typu $f(APROX) \leq 2\, f(OPT)$.

Definice: Poměrová chyba aproximačního algoritmu je definována jako poměr (podíl) $f(APROX) / f(OPT)$ pro minimalizační úlohy a $f(OPT) / f(APROX)$ pro maximalizační úlohy. Relativní chyba je pak definována jako $|f(APROX) - f(OPT)| / f(OPT)$.

Naivní aproximační algoritmus FRONTA pro optimalizační verzi ROZ: bere úkoly postupně podle jejich čísel a každý úkol vždy umístí na stroj, který je volný nejdříve.

Značení: OPT = optimální rozvrh, Q = rozvrh zkonstruovaný algoritmem FRONTA, délka(OPT) = o, délka(Q) = q

Věta: Pokud m je počet strojů, tak $q \leq ((2m - 1) / m)o$ a tento odhad již nelze zlepšit.

Důsledek: Aproximační algoritmus FRONTA má poměrovou chybu 2.

Důkaz:

1. Těsnost odhadu: Pro každé m zkonstruujeme zadání, pro které platí v dokazované nerovnosti rovnost, a to následujícím způsobem

   $x_1 = x_2 = \ldots = x_{m-1} = m-1$ ($m-1$ úkolů délky $m-1$)

   $x_m = x_{m+1} = \ldots = x_{2m-2} = 1$ ($m-1$ úkolů délky 1)

   $x_{2m-1} = m$ (1 úkol délky m)

2. Platnost nerovnosti: Nechť j je úkol končící jako poslední v rozvrhu Q (končící v čase q) a nechť t je okamžik zahájení úkolu j. Potom žádný procesor nemá prostoj před časem t a platí $mq \leq (2m - 1)o$.

Lepší aproximační algoritmus USPOŘÁDANÁ FRONTA pro optimalizační verzi ROZ: pracuje stejně jako FRONTA, ale na začátku úkoly setřídí do nerostoucí posloupnosti podle jejich délek.

Značení: OPT = optimální rozvrh,
U = rozvrh zkonstruovaný algoritmem USPOŘÁDANÁ FRONTA,
délka(OPT) = o, délka(U) = u

Věta: Pokud m je počet strojů, tak $u \leq ((4m - 1) / 3m)o$ a tento odhad již nelze zlepšit.

Důsledek: Aproximační algoritmus USPOŘÁDANÁ FRONTA má poměrovou chybu 4/3.

Důkaz: Těsnost odhadu: Pro každé liché m zkonstruujeme zadání, pro které platí v dokazované nerovnosti rovnost, a to následujícím způsobem

$x_1 = x_2 = 2m-1$ (2 úkoly délky $2m-1$)

$x_3 = x_4 = 2m-2$ (2 úkoly délky $2m-2$)

$x_{2m-3} = x_{2m-2} = m+1$ (2 úkoly délky $m+1$)

$x_{2m-1} = x_{2m} = x_{2m+1} = m$ (3 úkoly délky m)

Lemma: Pokud pro všechny úkoly platí $x_i \geq 1/3o$ pak $u = o$.

Dokončení důkazu: Nechť j je úkol končící jako poslední v rozvrhu U (končící v čase u). Pokud $x_j > 1/3o$ tak použijeme Lemma, v opačném případě je důkaz velmi podobný jako pro algoritmus FRONTA.