Algoritmy a datové struktury II

TIN061

Ondřej Čepek

Sylabus

Vyhledávání v textu
Toky v sítích
Aritmetické algoritmy (FFT)
Paralelní algoritmy (třídění a sčítání)
Převoditelnost problémů a třídy časové složitosti
Aproximační algoritmy
Základní geometrické algoritmy v rovině
(Pravděpodobnostní algoritmy a kryptografie)

Vyhledávání řetězců v textu

Σ konečná abeceda (množina znaků)

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

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

skládání (konkatenace) slov u a v :

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

(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 $x = x_1x_2 \ldots x_n \in \Sigma^*$ a hledané vzorky
$K = \{y_1, y_2, \ldots, y_k\}$, kde $y_p = y_{p,1} \ldots y_{p,length(p)} \in \Sigma^*$ pro $p = 1, \ldots, k$

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

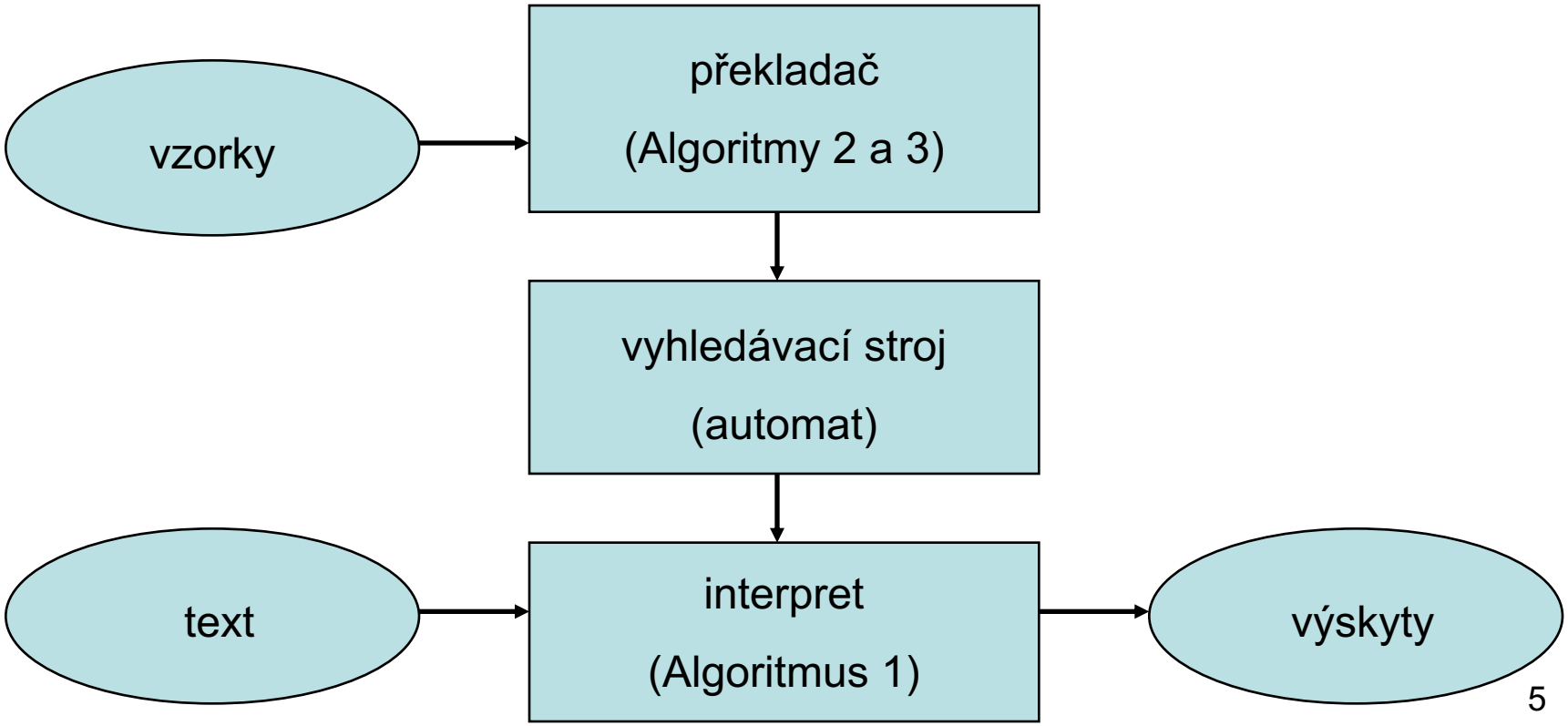
Algoritmus na míru

```
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 = {y1 … yk}                 {množina vzorků}
výstup:   Q = {0, … ,q}                 {množina stavů vyhledávacího stroje}
          g : Q x Σ → Q ∪ {⊥}            {výstupní funkce splňující Vlastnost 1}
          o : Q → P(K)                  {„polotovar“ výstupní funkce out}
procedure Enter(yp,1 … yp,m);           {připojení slova yp ke grafu funkce g}
begin     stav := 0; i:= 1;
          while (i <= m) and (g(stav,yp,i) <> ⊥) do
            begin   stav := g(stav,yp,i);        {pohyb po již hotové větvi}
                    i := i + 1                   {posun ve slově yp}
            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,yp,i) := q;           {prodloužení větve}
                    stav := q;                   {pokročení do nového stavu}
                    i := i + 1                   {posun ve slově yp}
            end;
          o(stav) := {yp}
end;      {of Enter}
begin     Q := {0}; for all x ∈ Σ do g(0,x) := ⊥;        {hlavní program}
          for p := 1 to k do Enter(yp);
          for all x ∈ Σ do if g(0,x) = ⊥ then g(0,x) = 0
end.
```

Algoritmus 3 (konstrukce vyhledávacího stroje – 2.fáze)

vstup: $Q = \{0, \ldots, q\}$ {množina stavů vyhledávacího stroje}
$g : Q \times \Sigma \rightarrow Q \cup \{\perp\}$ {výstupní funkce splňující Vlastnost 1}
$o : Q \rightarrow P(K)$ {„polotovar“ výstupní funkce out}

výstup: $f : Q \rightarrow Q$ {zpětná funkce splňující Vlastnost 2}
$out : Q \rightarrow P(K)$ {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
```

Toky v sítích

Síť: orientovaný graf $G = (V,E)$ se dvěma vybranými vrcholy s,t (zdroj a stok) a kladnou kapacitou $c(u,v)$ na každé hraně $(u,v) \in E$. Kapacita je dodefinována i pro ostatní dvojice vrcholů: $c(u,v) = 0$ pokud $(u,v) \notin E$.

Tok: je funkce $f : V \times V \rightarrow R$ splňující následující tři vlastnosti:

1. (Symetrie) $\forall\ u,v \in V :\ f(u,v) = -f(v,u)$
2. (Kapacita) $\forall\ u,v \in V :\ f(u,v) \leq c(u,v)$
3. (Zachování toku) $\forall\ u \in V - \{s,t\} :\ \Sigma_{v \in V} f(u,v) = 0$ (jako Kirkhoffův zákon pro el.proud)

Pokud $f(u,v) > 0$ tak říkáme, že máme (nenulový) tok z u do v. Pokud $f(u,v) = c(u,v)$ tak říkáme, že je hrana (u,v) saturovaná. Velikost toku f značíme $|f|$ a je to celkový čistý tok ze zdroje, tj. $|f| = \Sigma_{v \in V} f(s,v)$.

Problém: hledání maximálního toku, tj. toku maximální velikosti.

Řez: v kontextu toků je řez dvojice množin vrcholů X,Y taková, že $X \cup Y = V$, $s \in X$, $t \in Y$. Kapacita řezu X,Y je součet kapacit hran jdoucích přes řez, tj. $c(X,Y) = \Sigma_{u \in X, v \in Y} c(u,v)$. Minimální řez je řez s minimální kapacitou. Tok přes řez X,Y je součet toků po hranách jdoucích přes řez, tj. $f(X,Y) = \Sigma_{u \in X, v \in Y} f(u,v)$.

Lemma 1: Pro každý tok f a řez X,Y platí, že tok přes řez X,Y je roven velikosti toku, tj. $f(X,Y) = |f|$.

Díky Lemma 1 a triviálnímu faktu, že $f(X,Y) \leq c(X,Y)$ pro každý řez X,Y (díky kapacitnímu omezení) platí, že velikost maximálního toku je nejvýše rovna kapacitě minimálního řezu. Ukážeme, že zde vždy platí rovnost.

Reziduální kapacita toku f je funkce $r : V \times V \rightarrow R$ definovaná předpisem

$$r(u,v) = c(u,v) - f(u,v)$$

Reziduální graf pro f: orientovaný graf $R = (V,E')$, kde $(u,v) \in E'$ tehdy a jen tehdy, když platí $r(u,v) > 0$ a hodnotu $r(u,v)$ pak nazýváme kapacitou hrany (u,v) v reziduálním grafu.

Zlepšující cesta pro f: libovolná cesta p z s do t v grafu R. Reziduální kapacita $r(p)$ zlepšující cesty p je rovna minimálnímu $r(u,v)$ z hran (u,v) na cestě p. Velikost toku můžeme zvýšit až o $r(p)$ zvýšením toku (o stejné množství) na všech hranách cesty p. (Poznámka: při zvýšení f(u,v) je nutné proporčně snížit f(v,u) pro zachování symetrie.)

Věta (o max.toku a min.řezu): Následující podmínky jsou ekvivalentní

1. tok f je maximální tok
2. pro f neexistuje zlepšující cesta
3. platí $|f| = c(X,Y)$ pro nějaký řez X,Y

Věta o max.toku a min.řezu dává návod jak zkonstruovat maximální tok postupným zlepšováním.

Metoda zlepšující cesty: Začni s nulovým tokem a opakuj následující krok, dokud není dosaženo toku, pro který neexistuje zlepšující cesta.

Zlepšující krok: Pro aktuální tok f najdi zlepšující cestu p a zvyš velikost toku pomocí zvýšení toku na cestě p o r(p).

Jednoduchá implementace (Ford a Fulkerson): Najdi libovolnou zlepšující cestu pomocí prohledání grafu R (libovolným algoritmem na prohledávání orientovaných grafů).

Poznámky:

• ze znalosti maximálního toku můžeme v čase 0(m) zkonstruovat minimální řez (viz důkaz Věty o max.toku a min.řezu)

• pokud jsou kapacity iracionální čísla, tak nemusí jednoduchá implementace metody zlepšující cesty skončit po konečném počtu kroků, velikost toku sice vždy konverguje ale nemusí konvergovat k velikosti maximálního toku

• pokud jsou kapacity racionální čísla, tak lze úlohu převést na ekvivalentní úlohu s celočíselnými kapacitami

• pokud jsou kapacity celočíselné, tak každý zlepšující krok zvýší velikost toku alespoň o jedna, a tudíž metoda končí po nejvýše |f*| zlepšujících krocích, navíc je zkonstruovaný maximální tok celočíselný (na každé hraně)

„Ideální verze“ metody zlepšující cesty:

Lemma 2: Vždy existuje posloupnost nejvýše m zlepšujících cest, pomocí nichž lze zkonstruovat maximální tok.

Implementace s maximálním zlepšením (Edmonds a Karp): V každém kroku najdi mezi všemi zlepšujícími cestami tu s maximální reziduální kapacitou.

Lemma 3: Nechť f je libovolný tok a nechť f^* je maximální tok na G. Potom velikost maximálního toku na reziduálním grafu R pro tok f je $|f^*| - |f|$.

Věta: Počet zlepšujících kroků při implementaci s maximálním zlepšením je $0(m \log c)$ kde c je maximální kapacita nějaké hrany.

Poznámky:

• toto už je polynomiální počet kroků vzhledem k velikosti dat

• odhad platí jen když jsou kapacity celočíselné (a metoda v tom případě samozřejmě opět konverguje k maximálnímu toku)

• zlepšující cestu s maximální reziduální kapacitou lze najít v polynomiálním čase pomocí modifikovaného Dijkstrova algoritmu (pro tzv. bottleneck problém), kde délku cesty neměříme součtem délek hran ale délkou nejkratší hrany (detaily nebudeme zkoumat, další probírané algoritmy jsou lepší)

• nyní bude cílem implementace jejíž časová složitost bude záviset jen na n a m

Implementace s nejkratším zlepšením (Edmonds a Karp): V každém kroku najdi mezi všemi zlepšujícími cestami tu nejkratší, tj. zlepšující cestu s minimálním počtem hran.

Věta: Počet zlepšujících kroků při implementaci s nejkratším zlepšením je $0(nm)$ a metoda zlepšující cesty běží v tomto případě v čase $0(nm^2)$.

Definice: Nechť f je tok a R je reziduální graf pro f. Úroveň u(x) vrcholu x v R je délka nejkratší cesty z s do x v R. Úrovňový graf U pro tok f je podgraf grafu R, který obsahuje pouze vrcholy dosažitelné z s a pouze hrany (x,y) pro které $u(y) = u(x) + 1$.

Pozorování: Graf U lze zkonstruovat v čase O(m) pomocí BFS a pokud existuje zlepšující cesta, tak U obsahuje všechny nejkratší zlepšující cesty.

Dalšího zlepšení lze dosáhnout tím, že místo zvyšování toku po jednotlivých nejkratších zlepšujících cestách použijeme všechny nejkratší zlepšující cesty „najednou“.

Definice: Tok f na grafu U se nazývá blokující tok, pokud každá cesta z s do t v grafu U (v původním U, tj. ne pozměněném tokem f) obsahuje saturovanou hranu.

Algoritmus (Dinic): Začni s nulovým tokem a opakuj následující (blokující) krok dokud existuje zlepšující cesta, tj. dokud je vrchol t dosažitelný v aktuálním úrovňovém grafu:

(Blokující) krok: Najdi blokující tok f’ na úrovňovém grafu U definovaném pomocí aktuálního toku f. Nahraď tok f tokem f +f’ který je definován předpisem

$$(f + f')(x,y) = f(x,y) + f'(x,y)$$

Lemma 4: Dinicův algoritmus zastaví po nejvýše $n - 1$ blokujících krocích.

Jak hledat blokující tok: Existuje několik metod, my probereme tu původní Dinicovu, která je nejprimitivnější.

Idea: Najdi cestu z s do t v U (pomocí DFS), zvyš po ní tok tak, aby nějaká hrana na nalezené cestě byla saturována. Z U vyhoď všechny nově saturované hrany a postup opakuj dokud je t dosažitelné z s v grafu U.

Formální popis: Začni s nulovým tokem a jdi na Inicializaci. Aktuálně prohlížený vrchol budeme označovat x a p bude zlepšující cesta z s do x.

Inicializace: Definuj p = [s] a x = s. Jdi na Vpřed.

Vpřed: Pokud z x nevede v U žádná hrana jdi na Vzad. Jinak vezmi libovolnou hranu (x,y), prodluž p o vrchol y a do x dosaď y (posuň aktuální vrchol). Pokud platí $y \neq t$ tak opakuj Vpřed, pokud $y = t$ jdi na Zlepši.

Vzad: Pokud $x = s$ tak zastav (neexistuje zlepšující cesta do t). Pokud $x \neq s$ tak nechť (v,x) je poslední hrana na p. Zkraťˇ p o vrchol x a hranu (v,x) odstraň z U. Do x dosaď y (posuň aktuální vrchol) a jdi na Vpřed.

Zlepši: Nechť $d = \min\{c(x,y) - f(x,y) \mid (x,y)$ je hrana v p$\}$. Přidej d k toku na všech hranách cesty p, odstraň z U všechny nově saturované hrany a jdi na Inicializace.

Věta: Dinicův algoritmus nalezne blokující tok v úrovňovém grafu U v čase $0(nm)$ a maximální tok ve vstupním grafu G v čase $0(n^2m)$.

Metoda „preflow-push“

Definice: Předtok (preflow) je funkce $f : V \times V \rightarrow R$ splňující stejně jako tok podmínku symetrie a kapacity na každé hraně, ale která místo podmínky zachování toku má pro každý vrchol u (kromě zdroje s) podmínku $e(u) = \Sigma_{v \in V} f(u,v) \geq 0$, kde e(u) je přebytek (exces) ve vrcholu u. Vrchol u různý od s a t se nazývá aktivní pokud má kladný přebytek ($e(u) > 0$).

Definice: Nechť f je předtok a R reziduální graf pro f. Pak funkce $h : V \rightarrow N$ je výšková funkce vzhledem k f pokud

- $h(s) = |V|$
- $h(t) = 0$
- $\forall (u,v) \in R : h(u) \leq h(v) + 1$

Pokud pro hranu $(u,v) \in R$ platí $h(u) = h(v) + 1$, tak se (u,v) nazývá přípustná.

Inicializace (pro generický preflow-push algoritmus):

1. Vynuluj všechna h(u), e(u) a f(u,v).
2. Dosaď $h(s) := |V|$.
3. Pro všechny sousedy u zdroje s dosaď
   - $f(s,u) := c(s,u)$ a $f(u,s) := - c(s,u)$
   - $e(u) := c(s,u)$

Platí po inicializaci: f je předtok a h je výšková funkce vzhledem k f.

Algoritmus používá dvě základní akce :

ZATLAČ(u) (neboli PUSH(u))

- POUŽITÍ : pokud je u aktivní (e(u) > 0) a existuje přípustná hrana (u,v) v R, tj. hrana splňující r(u,v) > 0 a h(u) = h(v) + 1.
- AKCE : pošli d(u,v) = min {e(u), r(u,v)} jednotek toku z u do v a příslušně aktualizuj f(u,v), f(v,u), e(u) a e(v).

Definice: ZATLAČ(u) je saturující, pokud se hrana (u,v), po které je posílán tok, stane saturovanou (což nastane pokud d(u,v) = r(u,v)) a nesaturující v opačném případě (tj. pokud d(u,v) = e(u) < r(u,v)).

ZVEDNI(u) (neboli LIFT(u))

- POUŽITÍ : pokud je u aktivní (e(u) > 0) a neexistuje přípustná hrana (u,v) v R.
- AKCE : h(u) := 1 + min {h(v) | (u,v) v R}

Algoritmus (generický preflow-push algoritmus):

- Inicializace
- Dokud existuje aktivní vrchol tak nějaký vyber (nechť je to u) a uplatni buď ZATLAČ(u) nebo ZVEDNI(u).

Lemma 1: Pro každý vrchol u během celého běhu algoritmu platí:

- h(u) nikdy neklesne
- každé ZVEDNI(u) zvýší h(u) alespoň o jedna

Lemma 2: Funkce h zůstává během celého běhu algoritmu výškovou funkcí vzhledem k aktuálnímu (před)toku f.

Lemma 3: Během celého běhu algoritmu platí, že v R neexistuje cesta z s do t.

Věta (korektnost algoritmu):

Pokud algoritmus skončí, tak f (v tom okamžiku) reprezentuje maximální tok.

Časová složitost: Za celou dobu běhu algoritmus udělá:

- $O(|V|^2)$ zvednutí
- $O(|V||E|)$ saturujících zatlačení
- $O(|V|^2|E|)$ nesaturujících zatlačení

  což při implementaci potřebující $O(|V|)$ na každé zvednutí a $O(1)$ na každé zatlačení dává celkovou složitost algoritmu $O(|V|^2|E|)$.

Poznámka: „chytrým“ výběrem aktivních vrcholů lze časovou složitost snížit až na $O(|V||E| \log(|V|^2 / |E|))$, což je asymptoticky nejrychlejší známý algoritmus [Goldberg, Tarjan 88].

Násobení polynomů

Dva způsoby reprezentace polynomů:

1. Pomocí vektoru koeficientů (zde vektor komplexních čísel)

$$A(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_1x^1 + a_0x^0$$

A(x) je polynom s mezí stupně n, a_0 … a_{n-1} jsou jeho koeficienty a A(x) je polynom stupně k pokud a_k je jeho nejvyšší nenulový koeficient

- Součet dvou polynomů: $\Theta(n)$
- Výpočet hodnoty v bodě: $\Theta(n)$ pomocí Hornerova schématu
- Součin dvou polynomů: $\Theta(n^2)$ (konvoluce příslušných vektorů koeficientů)

2. Pomocí funkčních hodnot v bodech

$$\{ (x_0,y_0), (x_1,y_1), \ldots , (x_{n-1},y_{n-1}) \}$$

Takto lze jednoznačně reprezentovat libovolný polynom A(x) s mezí stupně n, a to libovolnou n-ticí ve které jsou všechny x_i po dvou různé a $y_i = A(x_i)$ pro každé i.

- Výpočet hodnoty v bodě: mimo zadané body nelze bez konverze k reprezentaci pomocí vektoru koeficientů
- Součet dvou polynomů: $\Theta(n)$
- Součin dvou polynomů: $\Theta(n)$ ale POZOR potřebujeme 2n bodů

Konverze mezi oběma reprezentacemi

1. Koeficienty → Dvojice (evaluace): $\Theta(n^2)$ pomocí Hornerova schématu (i pro $2n$ bodů), ale lze v $\Theta(n \log n)$ když body pro výpočet funkčních hodnot vybrány „chytře“
2. Dvojice → Koeficienty (interpolace): obecně $\Theta(n^3)$ Gaussovou eliminací nebo $\Theta(n^2)$ pomocí Lagrangeovy formule (nebudeme probírat), ale lze v $\Theta(n \log n)$ když známe funkční hodnoty v „chytře“ vybraných bodech

Násobení polynomů v $\Theta(n \log n)$

Chytře vybranými body budou $2n$-té komplexní odmocniny čísla 1, které budeme značit

$$w_{2n}^0, w_{2n}^1, \ldots, w_{2n}^{2n-1}$$

Násobení pro vstup $a_0 \ldots a_{n-1}$ a $b_0 \ldots b_{n-1}$ (dva vektory koeficientů polynomů s mezí stupně n) pak bude probíhat následovně:

1. Doplníme oba vektory n nulami na posloupnosti délky $2n-1$ (to je nová mez stupně)
2. Evaluace: spočítáme funkční hodnoty $A(x)$ i $B(x)$ ve všech $2n$ odmocninách čísla 1
3. Násobení: bod po bodu $C(x) = A(x)B(x)$ ve všech $2n$ odmocninách čísla 1
4. Interpolace: spočítáme vektor koeficientů polynomu $C(x)$ zadaného $2n$ funkčními hodnotami (ve všech odmocninách čísla 1)

Kroky 1 a 3 trvají $\Theta(n)$ a kroky 2 a 4 trvají $\Theta(n \log n)$ pokud jsou implementovány Rychlou Fourierovou Transformací (FFT)

Vlastnosti n-tých odmocnin čísla 1 v komplexním oboru

Krátící lemma: Pokud $n \geq 0$, $k \geq 0$ a $d > 0$ potom $w_{dn}^{dk} = w_n^k$.

Důsledek: Pro $n > 0$ sudé platí $w_n^{n/2} = w_2^1 = -1$.

Půlící lemma: Pro $n > 0$ sudé platí, že druhé mocniny n komplexních n-tých odmocnin čísla 1 jsou rovny $n/2$ komplexním $n/2$-tým odmocninám čísla 1 (každá zastoupena 2x).

Součtové lemma: Pokud $n \geq 0$ a $k > 0$ a k není dělitelné n potom $\Sigma_{j=0}^{n-1} (w_n^k)^j = 0$.

Diskrétní Fourierova Transformace (DFT)

Polynom $A(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_1x^1 + a_0x^0$ s mezí stupně n zadaný vektorem koeficientů chceme reprezentovat pomocí funkčních hodnot v n komplexních n-tých odmocninách čísla 1, tj. chceme spočítat hodnoty

$$y_k = A(w_n^k) \text{ pro } k = 0, 1, \ldots, n-1.$$

Vektor y (funkčních hodnot) se nazývá DFT vektoru a (koeficientů), píšeme $y = DFT_n(a)$.

Transformace opačným směrem se nazývá inverzní DFT a píšeme $a = DFT_n^{-1}(y)$.

Poznámka: pokud n je mezí stupně obou vstupních polynomů (které chceme vynásobit), tak zde vlastně pracujeme s $n' = 2n$, ale pro jednoduchost značení budeme používat n (navíc na asymptotickou složitost nemá vliv zda používáme n nebo n').

Rychlá Fourierova Transformace (FFT)

Algoritmus využívající strategii „rozděl a panuj“ pro spočítání $DFT_n(a)$, rozdělením $A(x)$ pomocí koeficientů lichého a sudého stupně na dva polynomy s mezí stupně $n/2$:

$$A^s(x) = a_0 + a_2x + a_4x^2 + \dots + a_{n-2}x^{n/2-1}$$
$$A^l(x) = a_1 + a_3x + a_5x^2 + \dots + a_{n-1}x^{n/2-1}$$

Nyní zjevně platí $A(x) = A^s(x^2) + x\,A^l(x^2)$ (vztah označme (¤)), takže úloha spočítat funkční hodnoty polynomu $A(x)$ v bodech $w_n^0, w_n^1, \dots, w_n^{n-1}$ je převedena na úlohu

- spočítat funkční hodnoty polynomů $A^s(x)$ a $A^l(x)$ (s mezí stupně $n/2$) v bodech $(w_n^0)^2, (w_n^1)^2, \dots, (w_n^{n-1})^2$ (což je ale jen $n/2$ různých bodů díky půlícímu lemmatu), tj. místo původní úlohy vyřešit dvakrát zcela stejnou úlohu na polovičních datech

- zkombinovat výsledky podle vztahu (¤) což zabere čas $\Theta(n)$

Celkový čas $T(n)$ pro FFT délky n je $T(n) = 2T(n/2) + \Theta(n)$ a tedy $T(n) = \Theta(n \log n)$.

Lemma Nechť V_n je Vandermondeho matice pro $w_n^0, w_n^1, \dots, w_n^{n-1}$. Pak prvek na pozici (j,k) v inverzní matici V_n^{-1} je roven w_n^{-kj} / n.

Tím pádem lze i inverzní DFT počítat pomocí FFT, protože spočítání koeficientů polynomu zadaného funkčními hodnotami $y_0, y_1, \dots, y_{n-1}$ je to samé jako spočítání funkčních hodnot polynomu $Y(x)/n$ s koeficienty $y_0/n, y_1/n, \dots, y_{n-1}/n$ a sice v bodech $w_n^0, w_n^{-1}, \dots, w_n^{-(n-1)}$

Implementace FFT

1. Rekurzivní implementace (přímý přepis algoritmu)

```
REC-FFT(a);
n := length(a);                    {n je mocnina dvojky}
if n=1 then return a;              {dno rekurze}
p := e^(2πi / n);                  {principiální kořen w_n^1 = generátor ostatních kořenů}
w := 1;                            {aktuálně zpracovávaný kořen, začíná se s w = w_n^0}
a^s := (a_0, a_2, … , a_(n-2));
a^l := (a_1, a_3, … , a_(n-1));    {příprava vstupních dat pro rekurzi}
y^s := REC-FFT(a^s);
y^l := REC-FFT(a^l);               {vlastní rekurze}
for k := 0 to (n/2 – 1) do
        y_k := y_k^s + w y_k^l;            {spočítání hodnoty A(w) v aktuálním kořeni w = w_n^k}
        y_(k+n/2) := y_k^s – w y_k^l;      {spočítání hodnoty A(w) v protilehlém kořeni w_n^(k+n/2)}
        w := w p;                          {posun na další kořen, tj. na w_n^(k+1)}
return y
```

Časová složitost: je vidět, že $T(n) = 2T(n/2) + \Theta(n)$ a tedy $T(n) = \Theta(n \log n)$, protože práce mimo rekurzi je zjevně $\Theta(n)$.

2. Iterativní implementace (hlavní idea)

vznikne následujícími úpravami rekurzívní implementace

- tělo for cyklu z rekurzivní implementace nahradíme „butterfly“ operací
- strom rekurze procházíme po patrech odspodu
- iniciální pořadí listů stromu pro start algoritmu získáme bitovou reverzí

Časová složitost: na každém patře stromu je voláno právě $n/2$ butterfly operací, tj. na každém patře je celková práce $\Theta(n)$, což dohromady s faktem, že hloubka stromu je přesně $\log n$ dává časovou složitost algoritmu $\Theta(n \log n)$.

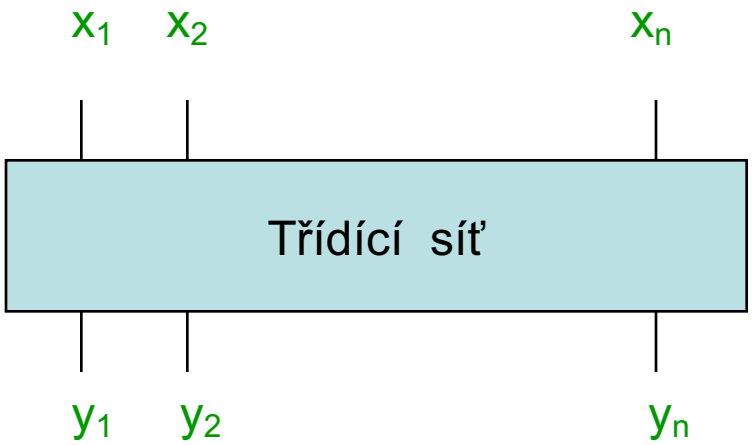
3. Paralelní implementace (hlavní idea)

- butterfly operace na stejném patře stromu se provádějí paralelně
- algoritmus lze realizovat hardwarově vhodným zřetězením butterfly obvodů do sítě, která má „šířku“ n a „hloubku“ $\log n$

Časová složitost: na každém patře nyní trvá (s použitím $n/2$ „procesorů“) veškerá práce $O(1)$, takže celková časová složitost je $\Theta(\log n)$.

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 $(K_u,K_v) \in E$ 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)

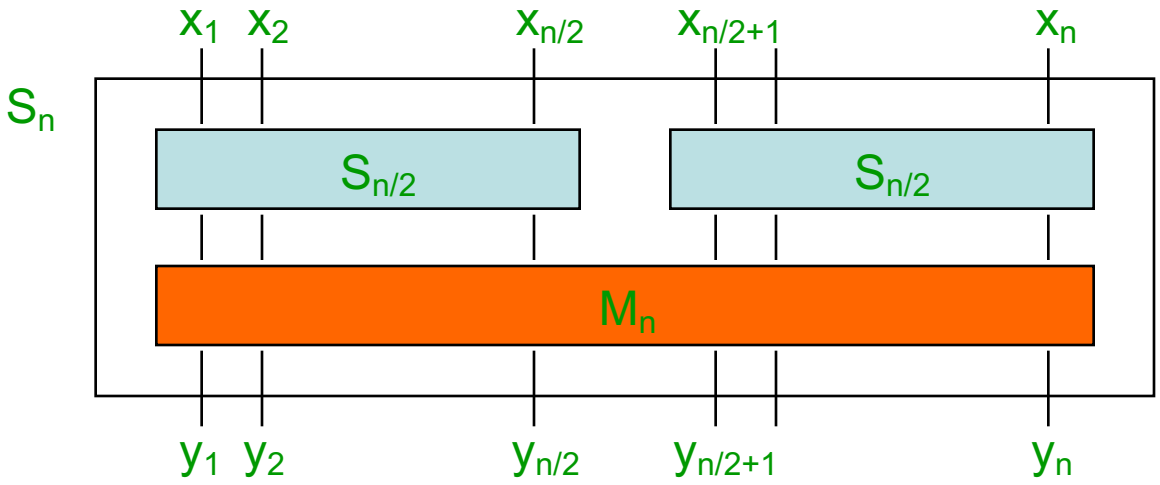
Topologicky jiná reprezentace sítě:

- „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

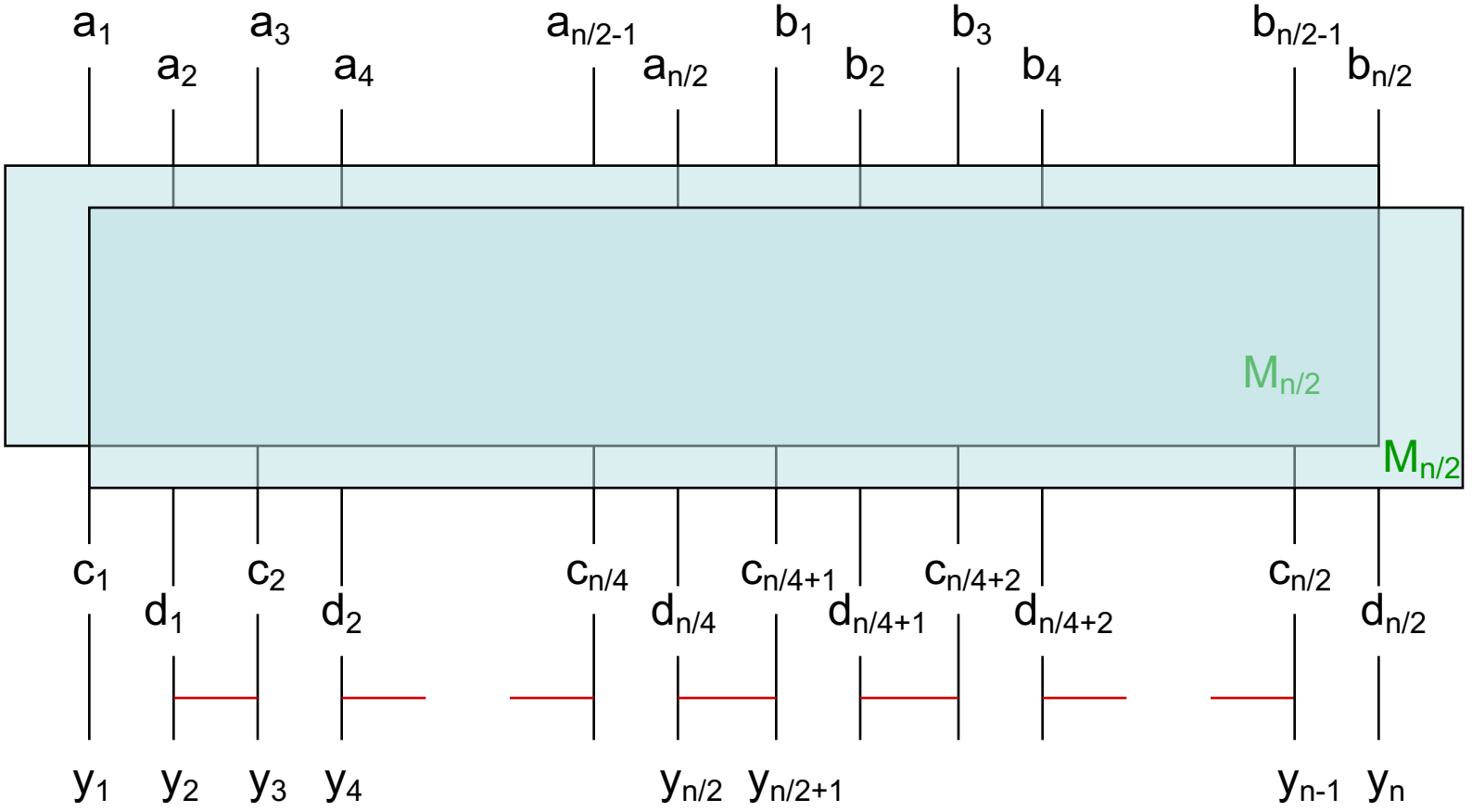
Chceme setřídit $x_1, x_2, \ldots, x_n$ (předpokládáme že n je mocnina dvojky)

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) :

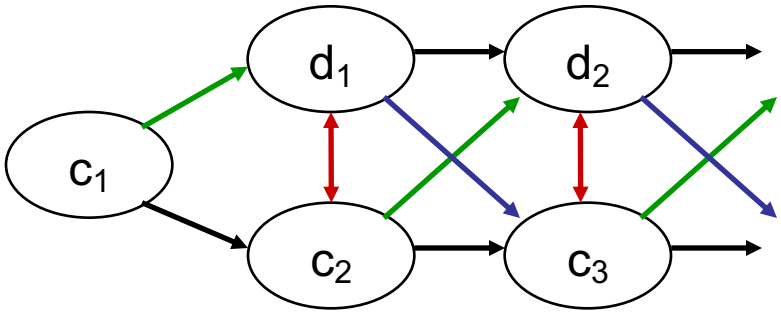


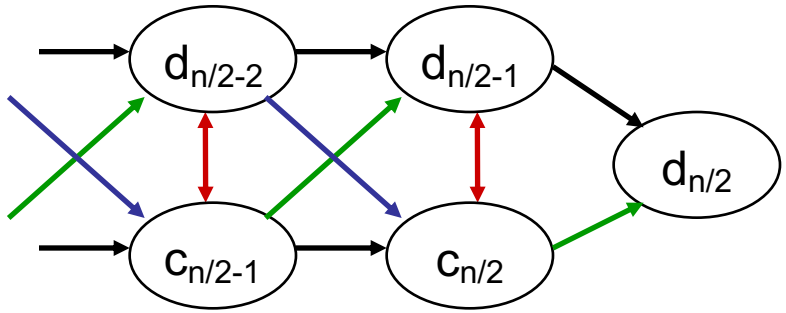
Liché členy obou setříděných posloupností jsou vstupem jedné kopie $M_{n/2}$ a sudé členy jsou vstupem druhé kopie $M_{n/2}$. Navíc jsou výstupy obou sítí propojeny jednou hladinou komparátorů dle obrázku (červené komparátory). Rekurze se opět zastaví pro $n=2$.

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

Transpoziční síť = síť složená pouze z komparátorů (zcela libovolně umístěných)

$\Rightarrow$ třídící síť je tedy speciálním případem transpoziční sítě

Lemma 1 Každá transpoziční síť dává pro vstup $x_1, x_2, \ldots, x_n$ na výstupu nějakou permutaci Π vstupních hodnot, tedy dává na výstupu $x_{\Pi(1)}, x_{\Pi(2)}, \ldots, x_{\Pi(n)}$.

Důkaz Zcela zřejmý, síť nemůže udělat nic jiného než nějak zpřeházet vstupy.

Definice Nechť C je transpoziční síť šířky n. Permutace $\Pi : \{1, 2, \ldots, n\} \rightarrow \{1, 2, \ldots, n\}$ je dosažitelná pro C, pokud existuje vstupní posloupnost $x_1, x_2, \ldots, x_n$ taková, že C vydá na výstupu posloupnost $x_{\Pi(1)}, x_{\Pi(2)}, \ldots, x_{\Pi(n)}$.

Lemma 2 Nechť C je třídící síť šířky n. Pak je pro C dosažitelných všech n! permutací.

Lemma 3 Nechť C je transpoziční síť šířky n a nechť p je počet dosažitelných permutací pro C. Potom platí

$$p \leq 2^{s(C)}.$$

Důsledek Pokud je C třídící síť tak

$$n! \leq 2^{s(C)}$$

a tím pádem má C velikost $s(C) = \Omega(n \log n)$ a hloubku $d(C) = \Omega(\log n)$.

Aritmetické sítě

• definice je podobná jako u třídících sítí, zcela stejně je definována šířka (počet vstupů, tj. vstupních drátů), velikost (počet hradel) a hloubka (počet hladin) sítě

• časová složitost výpočtu opět odpovídá hloubce sítě

• dráty zde přenášejí jen hodnoty 0 a 1 (tj. jeden bit informace, ne více bitů jako v případě třídících sítí)

• hradla jsou logická hradla, typicky unární hradlo NOT a binární hradla AND, OR a XOR

• drát z jednoho výstupu může jít do více vstupů („rozvětvit se“), musí být ovšem stále zachována acyklicita sítě

Sčítač (full adder)

Sčítač je obvod mající tři vstupy x,y,z a dva výstupy c,s, jehož funkci lze chápat tak, že x a y jsou bity dvou binárních čísel ve stejném řádu, z je přenos z nižšího řádu, s příslušný bit součtu x+y a c je přenos do vyššího řádu

z tabulky pro výpočet s a c plyne, že s = parita(x,y,z) a c = majorita(x,y,z)

Sčítání dvou n-bitových binárních čísel

• n bude mocnina dvojky, sčítat budeme čísla $a = (a_{n-1}, \dots, a_0)$ a $b = (b_{n-1}, \dots, b_0)$

• ukážeme dva obvody: pro klasické sčítání a pro carry-lookahead sčítání

Obvod pro klasické sčítání

• sčítání je realizováno kaskádou n sčítačů, kde sčítač v řádu i čeká na přenos (carry bit) od sčítače řádu i –1 a posílá svůj carry bit do sčítače řádu i +1

• síť má velikost $\Theta(n)$ a hloubku také $\Theta(n)$

Obvod pro carry-lookahead sčítání

zrychlení je založeno na myšlence, že v některých případech lze přenos z i-tého řádu určit jen na základě a_i a b_i (které jsou známy od začátku) a není potřeba čekat na přenos z nižšího řádu (jehož spočítání může trvat dlouho)

• v případě, že $a_i = b_i = 0$ tak $c_{i+1} = 0$ (bez ohledu na hodnotu c_i) → kill carry bit c_{i+1}

• v případě, že $a_i = b_i = 1$ tak $c_{i+1} = 1$ (bez ohledu na hodnotu c_i) → generate carry bit c_{i+1}

• v případě $a_i \neq b_i$ platí, že $c_{i+1} = c_i$ → propagate carry bit ci do carry bitu c_{i+1}

tím je definován carry status každého řádu $i \in \{1, \ldots, n\}$, který označíme proměnnou x_i, jejíž hodnotu z množiny {k,p,g} lze snadno spočítat z a_{i-1} a b_{i-1} podle uvedených pravidel

uvažujme dva za sebou napojené sčítače jako jeden obvod s jedním vstupním a jedním výstupním carry bitem: carry status kombinovaného obvodu lze spočítat z (k,p,g) obou zúčastněných sčítačů

tím je definován binární carry status operátor $\otimes$ na množině {k,p,g}, který je asociativní (bez důkazu, stačí rutinní rozbor všech 27 případů)

pokud dodefinujeme $x_0 = y_0 = k$, pak pro každý řád $i \in \{1, \ldots, n\}$ lze definovat proměnnou

$$y_i = y_{i-1} \otimes x_i = x_0 \otimes x_1 \otimes \ldots \otimes x_i$$

kterou lze chápat jako i-tý prefix součinu $x_0 \otimes x_1 \otimes \ldots \otimes x_n$

Lemma Pro $i = 0, 1, \ldots, n$ platí

1. pokud $y_i = k$ tak $c_i = 0$
2. pokud $y_i = g$ tak $c_i = 1$
3. případ $y_i = p$ nenastane

Důsledek Pokud jsou známa všechna y_i tak už lze součet spočítat v $O(1)$ pomocí n paralelně běžících sčítačů (pokud ztotožníme $k = 0$ a $g = 1$, což lze protože p nemůže nastat), takže problém sčítání se tímto převádí na problém výpočtu všech prefixů.

Sčítací obvod sestává z n KPG obvodů a prefixového obvodu. Každý KPG obvod je využit dvakrát. Při prvním průchodu má i-tý KPG obvod na vstupu a_i a b_i a na výstupu x_i, které pošle do prefixového obvodu. Při druhém průchodu funguje i-tý KPG obvod jako sčítač (respektive jako jeho část počítající paritu), na vstupu má a_i, b_i a y_i, které přijde z prefixového obvodu (s tím že hodnoty k a g jsou interpretovány jako 0 a1) a na výstupu má $s_i = \text{parita}(a_i, b_i, y_i)$.

Čili zbývá dodělat paralelní prefixový obvod …

Paralelní prefixový obvod

jako vstupy dostane carry statusy jednotlivých řádů, tj. $x_0, x_1, \ldots, x_n$ a spočítá z nich hodnoty všech prefixů $y_0, y_1, \ldots, y_n$

označme $[i, j] = x_i \otimes x_{i+1} \otimes \ldots \otimes x_j$ (a tedy $[i, i] = x_i$)

díky asociativitě platí $[i, k] = [i, j-1] \otimes [j,k]$ pro libovolné j takové, že $i < j \leq k$

chceme spočítat $y_i = [0, i]$ pro všechna $i \in \{0, 1, \ldots, n\}$

paralelní prefixový obvod sestává pouze z obvodů realizujících carry status operátor $\otimes$

obvod má topologii úplného binárního stromu (zde potřebujeme aby n byla mocnina 2):

- v listech stromu jsou vstupy x_1 až x_n a v kořeni je vstup x_0
- při průchodu od listů ke kořeni se počítá pro každý vnitřní uzel (reprezentující interval od i do k), jehož levý syn reprezentuje interval od i do $j-1$ (a tedy má hodnotu $[i, j-1]$) a pravý syn interval od j do k (a tedy má hodnotu $[j, k]$) hodnota $[i, k] = [i, j-1] \otimes [j,k]$
- při průchodu od kořene k listům přijde do každého vnitřního uzlu (reprezentujícího interval od i do k) shora hodnota $[0, i-1]$, kterou uzel pošle beze změny levému synovi (reprezentujícímu interval od i do $j-1$) a pravému synovi (reprezentujícímu interval od j do k) spočítá hodnotu $[0, j-1] = [0, i-1] \otimes [i,j-1]$
- výstupy y_0 až y_{n-1} jsou spočítány v listech a výstup y_n v kořeni
- velikost obvodu je $\Theta(n)$ a hloubka $\Theta(\log n)$

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 (NEZÁVISLÁ MNOŽINA): Je dán neorientovaný graf G a číslo k.

  Otázka: Existuje v G úplný (prázdný) 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 F na 0-1 proměnných v KNF.

  Otázka: Existuje ohodnocení proměnných pro které má F hodnotu 1?

Ukážeme, že HK → TSP, SAT → 3SAT → KLIKA → NM → SAT a NM → SP → ROZ, kde A → B znamená, že A je polynomiálně redukovatelné na B, tj. 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.

Pseudopolynomiální algoritmy

Algoritmus pro SP: předpokládejme, že platí $a_1 \geq \ldots \geq a_n$, a že A je pole délky b.

```
for j := 1 to b do {A[j] := 0; a_0 := b+1};
for i := 1 to n do
        A[a_i] := 1;
        for j := b downto a_{i-1} do if (A[j] = 1) and (j+a_i ≤ b) then A[j+a_i] := 1;
SP := (A[b] = 1).
```

Platí: Po i-tém průchodu hlavním cyklem obsahuje pole A jedničky právě u těch indexů, které odpovídají součtům všech neprázdných podmnožin množiny $\{a_1, \ldots, a_i\}$, které jsou nejvýše rovny b.

Časová složitost: O(nb), což je

- exponenciální časová složitost vzhledem k binárně (ale také ternárně, dekadicky, …) kódovanému vstupu, ale
- polynomiální časová složitost vzhledem k unárně kódovanému vstupu.

Algoritmy s těmito vlastnostmi se nazývají pseudopolynomiální. Formální definice na dalším slajdu.

Nechť je dán rozhodovací problém Q a jeho instance X. Definujme:

kód(X) = délka zápisu (počet bitů) instance X v binárním (či „vyšším“) kódováním

max(X) = největší číslo v X (velikost čísla, NE délka jeho binárního zápisu)

Definice: Algoritmus řešící Q se nazývá pseudopolynomiální, pokud je jeho časová složitost při spuštění na vstupu X omezena polynomem v proměnných kód(X) a max(X).

Poznámka: každý polynomiální algoritmus je samozřejmě také pseudopolynomiální.

Pozorování: Pokud je Q takový, že pro každou jeho instanci X platí $max(X) \leq p(kód(X))$ pro nějaký (pevný) polynom p, tak pro Q pojem polynomiálního a pseudopolynomiálního algoritmu splývá. Problémy, kde toto nenastává budeme nazývat číselné problémy.

Definice: Rozhodovací problém Q se nazývá číselný, pokud neexistuje polynom p takový, že pro každou instanci X problému Q platí $max(X) \leq p(kód(X))$.

Věta: Nechť Q je NP-úplný problém, který není číselný. Potom pokud $P \neq NP$, tak Q nemůže být řešen pseudopolynomiálním algoritmem.

Otázka: Je každý číslený problém řešitelný nějakým pseudopolynomiálním algoritmem?

Odpověď: NE (a typickým představitelům takových problémů se říká silně NP-těžké)

Silně NP- úplné problémy

Nechť je Q rozhodovací problém a p polynom. Symbolem Q_p označíme množinu instancí problému Q (tj. podproblém problému Q), pro které platí $max(X) \leq p(kód(X))$, tj.

$$Q_p = \{X \in Q \mid max(X) \leq p(kód(X))\}$$

Věta: Nechť je A pseudopolynomiální algoritmus řešící Q. Potom pro každý polynom p je A polynomiálním algoritmem řešícím Q_p.

Definice: Rozhodovací problém Q se nazývá silně NP-úplný, pokud $Q \in$ ***NP*** a existuje polynom p takový, že podproblém Q_p je NP-úplný.

Věta: Nechť Q je silně NP-úplný problém. Potom pokud ***P ≠ NP***, tak Q nemůže být řešen pseudopolynomiálním algoritmem.

Příklady číselných silně NP-úplných problémů:

Obchodní Cestující (TSP) :

- je to číselný problém (váhy na hranách mohou být libovolně velké)
- je silně NP-úplný neboť zůstává NP-úplný i když váhy omezíme (malou) konstantou

3-partition (3-P) – toto je „čistě“ číselný problém :

Zadání: $a_1, \ldots, a_{3m}, b \in N$, taková že $\forall j : ¼\, b < a_j < ½\, b$ a platí $\Sigma_{j=1}^{3m} a_j = mb$.

Otázka: $\exists S_1, \ldots, S_m$ disjunktní rozdělení množiny $\{1, \ldots, 3m\}$ takové, že $\forall i : \Sigma_{j \in Si} a_j = b$?

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 $x_i > 1/3o$ 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.

Úloha vrcholového pokrytí (optimalizační verze):

Vstup: Neorientovaný graf $G = (V,E)$.

Úloha: Najít vrcholové pokrytí minimální velikosti, tj. najít $V' \subseteq V$ takové, že pro každé $(u,v) \in E$ platí $u \in V'$ nebo $v \in V'$ (nebo oboje), a navíc V' má minimální možnou kardinalitu.

Algoritmus VP: opakovaně vyber v grafu libovolnou hranu (u,v) dej jak u tak v do postupně konstruovaného vrcholového pokrytí a odstraň jak u tak v z grafu spolu se všemi incidentními (a tedy pokrytými) hranami dokud nezbývá v grafu žádná hrana.

Věta: Algoritmus VP má poměrovou chybu 2.

Úloha obchodního cestujícího (optimalizační verze):

Vstup: Úplný vážený neorientovaný graf $G = (V,E)$ a váhová funkce $c : E \rightarrow Z^+ \cup \{0\}$

Úloha: Najít v G Hamiltonovskou kružnici nejmenší celkové váhy (délky).

1.Obchodní cestující s trojúhelníkovou nerovností

Platí: $\forall\ u,v,w \in V : c(u,w) \leq c(u,v) + c(v,w)$

Napřed nutno zjistit: Je tento podproblém vůbec NP-těžký (a má tedy vůbec cenu uvažovat o aproximačních algoritmech)??

Algoritmus TSP:

a) Najdi minimální kostru grafu G.

b) Vyber libovolný vrchol grafu G a spusť z něj na nalezené kostře DFS, které očísluje vrcholy v preorder pořadí

c) Výsledná Hamiltonovská kružnice je dána pořadím (permutací) z bodu b)

Poznámka: Pokud je v bodě a) použit Primův (Jarníkův) algoritmus, tak celý algoritmus běží v čase $O(|E|) = O(|V|^2)$.

Věta: Algoritmus TSP má konstantní poměrovou chybu 2.

2. Obchodní cestující bez trojúhelníkové nerovnosti

Věta: Nechť $R \geq 1$ je libovolná konstanta. Potom pokud $P \neq NP$, tak neexistuje polynomiální aproximační algoritmus řešící obecný případ obchodního cestujícího s poměrovou chybou nejvýše R.

Důsledek (o existenci neaproximovatelných úloh): Existují NP-těžké optimalizační úlohy, pro které neexistují polynomiální aproximační algoritmy s konstantní poměrovou chybou (pokud $P \neq NP$).

Opačný případ: Existují NP-těžké optimalizační úkoly, které lze aproximovat s libovolně malou relativní chybou (poměrovou chybou libovolně blízko 1) s tím, že čím menší je požadovaná chyba tím vyšší je časová složitost aproximačního algoritmu.

Aproximační schémata

Definice: Aproximační schéma (AS) pro optimalizační úlohu X je algoritmus, jehož vstupem je zadání Y úlohy X a (racionální) číslo $e>0$, který pro libovolné pevné e pracuje jako aproximační algoritmus pro úlohu X s relativní chybou e.

Poznámka: Doba běhu může být exponenciální jak ve velikosti zadání Y tak v $1/e$.

Definice: Polynomiální aproximační schéma (PAS) pro optimalizační úlohu X je AS, jehož časová složitost je polynomiální vzhledem k velikosti zadání Y úlohy X.

Poznámka: Doba běhu může být stále ještě exponenciální vzhledem k $1/e$.

Definice: Úplně polynomiální aproximační schéma (ÚPAS) pro optimalizační úlohu X je PAS, jehož časová složitost je polynomiální také vzhledem k k $1/e$.

Úloha součtu podmnožiny (optimalizační verze):

Vstup: Množina přirozených čísel $A = \{x_1, \ldots, x_n\}$ a přirozené číslo t.

Úloha: Najít množinu indexů $S \subseteq \{1, \ldots, n\}$ takovou, že $sum = \sum_{i \in S} x_i$ je co největší při platnosti podmínky $sum \leq t$.

1. Pseudopolynomiální algoritmus pro SP

Značení: Nechť L je uspořádaný seznam přirozených čísel $a_1, \ldots, a_n$. Pak L+x, kde x je přirozené číslo je uspořádaný seznam přirozených čísel $a_1+x, \ldots, a_n+x$.

```
SOUČET(A,t)
begin    L0 := (0);          { seznam délky 1 obsahující číslo 0 }
         for i := 1 to n do Li := MERGE(Li-1, Li-1 + xi);
                             { MERGE slije oba seznamy, čísla větší než t zahodí }
         řešení := největší prvek v Ln
end.
```

Věta: Seznam L_i pro $1 \le i \le n$ je uspořádaný seznam obsahující součty všech podmnožin množiny $A = \{x_1, \ldots, x_n\}$, které jsou menší nebo rovny číslu t.

Důkaz: indukcí podle i

Časová složitost:

- v každém případě $O(|L_1| + \ldots + |L_n|)$
- pokud jsou v seznamech drženy duplicitní hodnoty tak (v nejhorším případě) $\Omega(2^n)$
- ale pokud jsou duplicity v MERGE vyházeny, tak $O(n \cdot t)$
- algoritmus je polynomiální pokud $t \le p(n)$ nebo $\forall i: x_i \le p(n)$ pro nějaký polynom p

2. Prořezávání seznamů

Nechť je dáno $0 < d < 1$. Prořezat seznam L parametrem d znamená odebrat z L co nejvíce prvků tak, že pro každý odstraněný prvek y existuje v prořezaném seznamu L' prvek z takový, že $(1 - d)\, y \le z \le y$.

```
PROŘEŽ(L,d)
begin     L' := (y1);
    poslední := y1;
    for i := 2 to |L| do if poslední < (1-d)yi  then
    begin L' := L' ∪ { yi };
            poslední := yi
    end;
    return L'
end.
```

Časová složitost: $\Theta(|L|)$

3. ÚPAS pro SP

Vstup: Množina přiroz. čísel $A = \{x_1, \ldots, x_n\}$, přirozené číslo t a aproximační parametr e.

```
APPROX-SP(A,t,e)
begin     L0 := (0);          { seznam délky 1 obsahující číslo 0 }
    for i := 1 to n do
    beginLi := MERGE(Li-1, Li-1 + xi);
                    { MERGE slije oba seznamy, čísla větší než t zahodí }
        Li := PROŘEŽ (Li, e/n);
    end;
    řešení := největší prvek v Ln
end.
```

Časová složitost: $\Theta(|L_1| + \ldots + |L_n|)$

Myšlenka: Opakovaným prořezáváním se chyba může postupně zvětšovat, ale e/n je dostatečně malý „prořezávací parametr“, aby celková relativní chyba „nasčítaná“ přes n iterací byla nejvýše e.

Věta: Algoritmus APPROX-SP je ÚPAS pro optimalizační úlohu SP.

Značení: y^* = optimální hodnota

z = hodnota vrácená algoritmem APPROX-SP

Cíl 1: Chceme ukázat, že $(1 - e)\, y^* \leq z \leq y^*$.

Lemma: Nechť $y \leq t$ je součet nějaké podmnožiny množiny $\{x_1, \ldots, x_i\}$. Pak na konci i-té iterace algoritmu APPROX-SP existuje $w \in L_i$ (tj. w je v prořezaném seznamu L_i) takové, že platí $(1 - e/n)^i\, y \leq w \leq y$.

Důsledek: Existuje $w \in L_n$ takové, že $(1 - e/n)^n\, y^* \leq w \leq y^*$ a číslo z vrácené algoritmem APPROX-SP je největší takové w.

Lemma: $\forall n > 1$ platí $(1 - e) < (1 - e/n)^n$ a tudíž $(1 - e)\, y^* \leq z \leq y^*$ (Cíl 1 splněn).

Cíl 2: Víme, že časová složitost APPROX-SP je $\Theta(|L_1| + \ldots + |L_n|)$ a chceme ukázat, že je také $O(p(n, \log t, 1/e))$ pro nějaký polynom p ve třech proměnných.

Lemma: $\forall i : |L_i| \leq (n \cdot \ln t) / e$

Důsledek: Algoritmus APPROX-SP má časovou složitost $O((n^2 \cdot \log t) / e)$ (Cíl 2 splněn).

Pravděpodobnostní (randomizované) algoritmy

pravděpodobnostní algoritmus dělá (na rozdíl od deterministického algoritmu) náhodné kroky, např. k některým krokům používá hodnoty získané z generátoru náhodných čísel

tím pádem dvě různá spuštění téhož pravděpodobnostního algoritmu na stejných datech mají (s velkou pravděpodobností) různý průběh

pravděpodobnostních algoritmů je mnoho typů, zde zmíníme jen dva a to algoritmy typu Las Vegas a typu Monte Carlo

Algoritmy typu Las Vegas

výsledek je vždy správný, náhodnost ovlivňuje pouze dobu běhu algoritmu, tj. po jaké cestě se algoritmus ke správnému výsledku dobere

Příklad: randomizovaný QuickSort – od deterministické verze se liší náhodnými výběry pivota při každém dělení posloupnosti, což poskytuje následující výhody

• dává dobrý průměrný čas (tj. O(n log n)) i v případě, že data na vstupu nejsou náhodné permutace – žádný vstup není apriori špatný (pro každý deterministický výběr pivota existují apriori špatné vstupy)

• může být spuštěn paralelně v několika kopiích, výsledek je získán z kopie, kde výpočet skončí nejdříve (pro deterministickou verzi nemá takový postup žádný smysl)

Algoritmy typu Monte Carlo

náhodnost ovlivňuje jak dobu běhu, tak správnost výsledku: algoritmus může udělat chybu, ale pouze jednostranně (u odpovědí ANO/NE) a s omezenou pravděpodobností

Příklad: Rabin-Millerův algoritmus na testování prvočíselnosti

Úloha: pro zadané přirozené číslo n (rychle) rozhodnout zda je n prvočíslo

Trocha teorie (Malá) Fermatova věta (bez důkazu):

Nechť p je prvočíslo. Potom $\forall\ k \in \{1,2, \dots ,p-1\}$ platí $k^{p-1} \equiv 1 \pmod p$

Myšlenka: pokud n není prvočíslo, tak zkusíme (náhodně) najít „svědka“ k, porušujícího $k^{n-1} \equiv 1 \pmod n$, který „dosvědčuje“, že n je opravdu číslo složené (není to prvočíslo)

Problém: pro některá složená čísla je svědků příliš málo, takže je ¨příliš malá pravděpodobnost, že nějakého svědka (náhodně) vybereme.

Definice: Nechť T je množina dvojic přirozených čísel, kde $(k,n) \in T$ pokud $0 < k < n$ a je splněna alespoň jedna z následujících dvou podmínek:

1. neplatí $k^{n-1} \equiv 1 \pmod n$,

2. existuje i takové, že $m = (n-1) / 2^i$ je přirozené číslo a platí $1 < NSD(k^{m-1}-1, n) < n$

Věta 1: Číslo n je složené tehdy a jen tehdy, když existuje k takové, že $(k,n) \in T$.

Věta 2: Nechť n je složené číslo. Pak existuje alespoň $(n-1)/2$ takových čísel k, pro které platí $(k,n) \in T$.

```
Rabin-Miller(n);
for i:=1 to počet do
   k_i := náhodné přirozené číslo z intervalu [1,n-1];
   if (k_i,n) ∈ T then Report (n je složené);
                       Abort;
Report (n je prvočíslo)
```

Pokud Rabin-Miller(n) rozhodne, že n je složené, tak je to zaručeně správný výsledek (byl nalezen „svědek“), pokud Rabin-Miller(n) rozhodne, že n je prvočíslo, tak se může jednat o chybu, ale pouze v případě, že všechna vybraná k_i byli „ne-svědci“ pro složené číslo n, což ale může (díky Větě 2) nastat nejvýše s pravděpodobností

$$P(\text{chyba}) \leq (1/2)^{\text{počet}}$$

pokud jsou výběry jednotlivých k_i vzájemně nezávislé

Vlastnosti algoritmu:

• zvyšováním počtu iterací (počtu testovaných k_i) lze dostat libovolně malou (předem zvolenou) pravděpodobnost chyby

• jednotlivé iterace (testy pro různá k_i) lze provádět paralelně

Časová složitost:

každá iterace trvá jen polynomiálně vzhledem k délce zápisu čísla n (tj. k délce vstupu), k tomu je ovšem potřeba ukázat, že test zda $(k_i,n) \in T$ je možno provést v čase polynomiálním v log n, což není triviální (je nutné mít další znalosti z teorie čísel)

Kryptografie s veřejným klíčem (asymetrickou šifrou)

- každý účastník X má svůj veřejný klíč PX a soukromý klíč SX
- SX je znám pouze X, veřejný klíč PX může X sdělit všem s kterými komunikuje, nebo může být dokonce zveřejněn ve veřejně dostupném seznamu klíčů (třeba na webu)
- oba klíče specifikují funkce, které lze aplikovat na jakoukoli zprávu: tedy pokud D je množina všech konečných posloupností bitů (množina všech možných zpráv), tak obě funkce musí být prosté funkce zobrazující D na D (tj. jsou to permutace množiny D)
- funkci specifikovanou soukromým klíčem SX značíme SX() a funkci specifikovanou veřejným klíčem PX značíme PX(), přičemž předpokládáme, že každá z těchto funkcí je efektivně vyčíslitelná pokud známe příslušný klíč
- funkce SX() a PX() musí tvořit vzájemně inverzní pár funkcí – pro každou zprávu (konečnou posloupnost bitů) M tedy musí platit SX(PX(M)) = M a PX(SX(M)) = M.
- bezpečnost šifry stojí a padá s tím, že nikdo kromě účastníka X není schopen v „rozumném“ čase spočítat SX(M) pro jakoukoli zprávu M, což znamená, že
  1. účastník X musí držet klíč SX v absolutním bezpečí před vyzrazením
  2. funkce SX() nesmí být efektivně vyčíslitelná na základě znalosti PX (a schopnosti efektivně vyčíslit funkci PX()), což je hlavní obtíž při návrhu systému šifrování s veřejným klíčem

Předpokládejme, že máme 2 účastnice: A (Alici) a B (Barboru) s klíči SA, PA, SB a PB

Posílání zašifrované zprávy a její rozšifrování

Barbora chce poslat Alici zašifrovanou zprávu M:

• Barbora si opatří Alicin veřejný klíč PA (přímo od Alice či z veřejného seznamu klíčů)

• Barbora spočítá zašifrovaný text C = PA(M) a pošle ho Alici

• Alice na C aplikuje svůj soukromý klíč SA, tedy spočítá SA(C) = SA(PA(M)) = M

• Pokud C zachytí někdo jiný než Alice, nemá šanci získat M, protože neumí efektivně spočítat SA(C).

Posílání autentizované a podepsané (nešifrované) zprávy

Alice chce odpovědět Barboře tak, aby Barbora měla jistotu, že odpověď Q přichází od Alice a že text odpovědi nebyl pozměněn:

• Alice spočítá svůj digitální podpis q pro zprávu Q pomocí svého soukromého klíče, tj. spočítá q = SA(Q) a pošle Barboře dvojici (Q,q) – tj. zpráva Q odchází nešifrovaně

• Barbora spočítá PA(q) = PA(SA(Q)) = Q a porovná to s došlou zprávou Q

• Pokud se obě zprávy zcela shodují, má Barbora jistotu, že zpráva přichází od Alice a nebyla cestou pozměněna

• Pokud se zprávy liší, tak buď je podpis q falešný (nebyl vytvořen funkcí SA()) nebo je podpis pravý ale nezašifrovaná zpráva Q byla cestou pozměněna

Posílání autentizované a podepsané zašifrované zprávy

Alice chce poslat Barboře zprávu M tak, aby Barbora měla jistotu, že M přichází od Alice a že text M nebyl pozměněn. Navíc Alice chce, aby si M mohla přečíst pouze Barbora a nikdo jiný.

• Alice spočítá svůj digitální podpis pro M, tedy spočítá m = SA(M)

• Alice zašifruje dvojici (M,m) pomocí Barbořina veřejného klíče, tedy spočítá zašifrovaný text C = PB(M,m) a pošle C Barboře

• Barbora rozšifruje C pomocí svého soukromého klíče, tedy spočítá SB(C) = (M,m)

• Barbora ověří platnost Alicina podpisu a autenticitu M pomocí Alicina veřejného klíče, tj. spočítá PA(m) a porovná to s M – při neshodě Barbora ví, že buď bylo C cestou změněno (úmyslně či přenosovou chybou) nebo C nepřichází od Alice.

Hybridní šifrování

Pokud je zpráva M, kterou chce Barbora poslat Alici, velmi dlouhá a výpočet C = PA(M) a následně M = SA(C) by trval příliš dlouho, je možné použít šifrování s veřejným klíčem v kombinaci s nějakou symetrickou šifrou K, která šifruje zprávy rychle:

• Barbora spočítá C = K(M), což je opět dlouhá posloupnost bitů, k tomu spočítá PA(K), což je krátká posloupnost bitů (ve srovnání s M a PA(M)) a pošle (C,PA(K)) Alici

• Alice rozšifruje PA(K) pomocí svého SA, takže dostane K, pomocí kterého rozšifruje C a tak získá M

Hybridní autentizace a podepisování

Pro dlouhou zprávu M je také časově náročné počítat digitální podpis m = SA(M). Zde si vypomůžeme (veřejně známou) hashovací funkcí h, která má následující dvě vlastnosti:

1. I pro dlouhé M lze h(M) spočítat velmi rychle, typicky je h(M) krátký (např. 128 bitový) otisk (fingerprint) zprávy M.

2. Je výpočetně velmi obtížné (v rozumném čase nemožné) najít k M jinou zprávu Q takovou, aby platilo h(M) = h(Q)

Pokud chce Alice podepsat dlouhou zprávu posílanou Barboře, může postupovat takto:

• Alice spočítá otisk h(M) zprávy M, udělá z něj digitální podpis m = SA(h(M)) a pošle Barboře dvojici (M,m)

• Barbora obdrží M a také spočítá otisk h(M) který poté porovná s rozšifrovaným Aliciným digitálním podpisem PA(m) = PA(SA(h(M))). Pokud byla M cestou změněna, tak dojde k neshodě, protože díky vlastnosti 2 je těžké pozměnit M tak, aby se její otisk nezměnil.

Certifikační autority

Pokud si Alice pořizuje Barbořin veřejný klíč z veřejně dostupného seznamu (nebo jí ho Barbora posílá po síti), jak může mít jistotu, že nejde o podvrh? Pokud by byl klíč podvržen a následné zprávy modifikovány nebo podvrhovány stejným člověkem, který podvrhl svůj klíč jako Barbořin, tak jejich nepravost nelze zjistit (protože daný člověk bude mít k podvrženému veřejnému klíči i odpovídající soukromý klíč). Řešení:

• Existuje certifikační autorita Z, jejíž veřejný klíč PZ má každý účastník (tedy i Alice) nainstalován u sebe (například přišel na CD s šifrovacím softwarem).

• Barbora pak má od autority Z vydán certifikát ve tvaru C = „Barbořin klíč je PB“ podepsaný autoritou Z, tedy dvojici (C, SZ(C)) – toto může mít Barbora také již od koupě šifrovacího softwaru, nebo certifikát získá jinou bezpečnou cestou

• Tuto dvojici (C, SZ(C)) připojí Barbora ke každé podepisované zprávě, takže Alice (i kdokoli jiný) zjistí pomocí veřejného klíče PZ, že C bylo opravdu vydáno autoritou Z, a že tedy PB opravdu je Barbořin veřejný klíč

RSA (Rivest, Shamir, Adelman) šifra

pro vysvětlení RSA potřebujeme řadu pojmů a tvrzení z teorie čísel

Věta: Nechť a,b jsou přirozená čísla. Pak NSD(a,b) je nejmenší kladný prvek množiny

$$L = \{ax + by \mid x,y \in \mathbf{Z}\}$$

Důsledek: Nechť a,b jsou přirozená čísla. Pokud d je přirozené číslo, které dělí a i b, tak d dělí také NSD(a,b).

Věta: Nechť a,b jsou přirozená čísla, kde b>0. Pak NSD(a,b) = NSD(b, a mod b).

```
EUCLID(a,b)
if b=0 then Return(a)
        else Return(EUCLID(b, a mod b))
```

Lemma: Nechť $a > b \geq 0$ a EUCLID(a,b) udělá $k \geq 1$ rekurzivních kroků. Pak $a \geq F(k+2)$ a $b \geq F(k+1)$, kde F(i) je i-té Fibonacciho číslo.

Důsledek (Lamého věta): Nechť $a > b \geq 0$ a $F(k) \leq b < F(k+1)$. Pak EUCLID(a,b) udělá nejvýše $k - 1$ rekurzivních kroků.

Věta (bez Dk): $F(k) = \Theta(\varphi^k)$, kde $\varphi = (1+\sqrt{5})/2$ (což je tzv. „zlatý řez“).

Důsledek: Nechť $a > b \geq 0$ a $F(k) \leq b < F(k+1)$. Pak EUCLID(a,b) udělá nejvýše O(log b) rekurzivních kroků.

Pozorování: Pokud a,b jsou dvě nejvýše t-bitová binární čísla, tak EUCLID(a,b) provede O(t) rekurzivních kroků a v každém z nich O(1) aritmetických operací na (nejvýše) t-bitových číslech, tj. $O(t^3)$ bitových operací, pokud předpokládáme, že každá aritmetická operace na t-bitových číslech potřebuje $O(t^2)$ bitových operací (což je snadné ukázat). EUCLID je tedy polynomiální algoritmus vzhledem k velikosti vstupu.

Euklidův algoritmus lze snadno rozšířit tak, aby počítal také koeficienty x,y, pro které NSD(a,b) = ax + by.

```
EXTENDED-EUCLID(a,b)

if b=0 then Return(a,1,0)

        else (d',x',y') := EXTENDED-EUCLID(b, a mod b);

             (d,x,y) := (d',y',x' - ⌊a/b⌋ y');

             Return(d,x,y)
```

Věta (bez Dk): Nechť $n > 1$ a $a < n$ jsou dvě nesoudělná přirozená čísla. Pak má rovnice $ax \equiv 1 \pmod{n}$, právě jedno řešení $0 < x < n$ (a pokud jsou a,n soudělná, tak nemá žádné řešení).

Definice: Řešení rovnice $ax \equiv 1 \pmod{n}$ značíme $(a^{-1} \bmod n)$ a nazýváme multiplikativní inverz čísla a modulo n (aby existoval, tak musí být a,n nesoudělná).

Pozorování: $(a^{-1} \bmod n)$ snadno získáme pomocí rozšířeného Euklidova algoritmu.

Věta (speciální důsledek tzv. „čínské věty o zbytcích“ – bez Dk): Nechť a,b jsou nesoudělná přirozená čísla. Pak pro každá přirozená čísla x,y platí: $x \equiv y \pmod{ab}$ tehdy a jen tehdy, když $x \equiv y \pmod{a}$ a zároveň $x \equiv y \pmod{b}$.

Věta (malá Fermatova): Nechť p je prvočíslo. Pak $\forall k \in \{1,2, \ldots ,p-1\}$ platí $k^{p-1} \equiv 1 \pmod{p}$.

Nyní máme vše co potřebujeme k definici a vysvětlení RSA:

1. Náhodně vyber dvě velká prvočísla p a q (např. každé s 200 binárními ciframi).

2. Spočítej $n = pq$ (v uvedeném případě má n cca 400 binárních cifer).

3. Vyber malé liché číslo e, které je nesoudělné s číslem $(p-1)(q-1)$.

4. Spočítej multiplikativní inverz d čísla e modulo $(p-1)(q-1)$.

5. Zveřejni (e,n) jako veřejný RSA klíč a uschovej (d,n) jako soukromý RSA klíč.

Věta (korektnost RSA): Funkce $P(M) = M^e \bmod n$ a $S(M) = M^d \bmod n$ definují dvojici inverzních funkcí na množině všech zpráv, tj. na množině všech čísel v $Z_n = \{0,1, \ldots ,n-1\}$.

Proč je RSA bezpečná?

Na základě (e,n) není (zatím) nikdo schopen spočítat d aniž by znal rozklad n = pq a tím pádem také číslo (p-1)(q-1). A faktorizace velkých čísel je výpočetně těžký problém.

Jak je RSA rychlá?

To, jak rychle lze spočítat P(M) a S(M) závisí na tom, jak rychle umíme počítat zbytek modulo n při umocňování, tj. jak rychle lze spočítat $a^b \bmod n$.

UMOCNI (a,b,n) {kde binární zápis čísla b je $<b_k, \ldots, b_0>$ }

c := 1; d := a mod n;

for i := k-1 downto 0 do

  c := 2·c;

  d := (d·d) mod n;

  if b_i = 1 then c := c + 1;

                d := (d·a) mod n;

Return(d)

Časová složitost: Pokud a,b jsou nejvýše t-bitová binární čísla, tak UMOCNI provede O(t) aritmetických operací na (nejvýše) t-bitových číslech, tj. $O(t^3)$ bitových operací, pokud předpokládáme, že každá aritmetická operace na t-bitových číslech potřebuje $O(t^2)$ bitových operací (což je snadné ukázat).