Rozhledy matematicko-fyzikální

Jan Kára; Daniel Kráľ; Martin Mareš

Recepty z programátorské kuchařky Korespondenčního semináře z programování, 1. část

Rozhledy matematicko-fyzikální, Vol. 80 (2005), No. 1, 26–33

Persistent URL: http://dml.cz/dmlcz/146084

Recepty z programátorské kuchařky Korespondenčního semináře z programování, 1. část

Jan Kára, Daniel Král', Martin Mareš, MFF UK Praha

Matematicko-fyzikální fakulta Univerzity Karlovy pořádá pro studenty středních škol *Korespondenční seminář z programování*. Letošní ročník je již sedmnáctý. Od školního roku 2003/04 dostávají účastníci semináře společně se zadáním úloh i krátké povídání o algoritmech: *Recepty z programátorské kuchařky*. Protože se domníváme, že takovéto povídání by mohlo být užitečné i pro čtenáře *Rozhledů matematicko-fyzikálních*, nabídneme v tomto a v několika následujících číslech upravenou verzi *Receptů*. Pokud budete mít zájem dovědět se o *Korespondenčním semináři z programování* více, můžete navštívit jeho webové stránky na adrese `http://ksp.mff.cuni.cz/` nebo zaslat e-mail jeho organizátorům na adresu `ksp@mff.cuni.cz`.

V úvodním dílu o algoritmech a datových strukturách začneme jedním z nejznámějších algoritmů, Dijkstrovým algoritmem pro hledání nejkratších cest v grafech. Než vysvětlíme, co jsou grafy, a popíšeme Dijkstrův algoritmus, zastavíme se u datové struktury zvané halda, kterou budeme v tomto algoritmu používat.

Rychlost algoritmů

Nejdříve vysvětlíme, jak budeme měřit rychlost algoritmů. Nebude to se stopkami v ruce, spíše se zaměříme na to, jak se daný algoritmus zpomalí, pokud se vstupní data zvětší dvakrát, desetkrát apod.

Představme si, že máme za úkol setřídit N čísel a máme dva algoritmy, z nichž jeden běží v čase $\left(5N^2 + 100N\right)$ ns a druhý v čase $(1000N \log_2 N + 250)$ ns. Pokud je N malé, je první algoritmus rychlejší, naopak pro velké vstupy je rychlejší druhý algoritmus. Abychom se

vyhnuli práci s multiplikativními konstantami, které výrazně závisí na konkrétním programovacím jazyce, kompilátoru, počítači atd., zavedeme notaci „velké O“.

Funkce f je $O(g)$ pro funkci g, pokud existují reálná konstanta c a přirozené číslo n_0 takové, že platí $f(n) \leqq c \cdot g(n)$ pro všechna přirozená čísla $n \geqq n_0$.

Použitím této notace pak můžeme říci, že první algoritmus běží v čase $O(N^2)$ a druhý v čase $O(N \log N)$ (všimněte si, že zde nezáleží na základu logaritmu, neboť ten se „schová“ do konstanty c). Protože nás bude zajímat, jak se navržené algoritmy chovají pro velká data, je druhý algoritmus z našeho pohledu lepší než první. Podobně, jako měříme časovou složitost, budeme měřit i paměťovou složitost.

Halda

Teď už můžeme začít s povídáním o algoritmech a datových strukturách. *Halda* je datová struktura pro uchovávání množiny čísel (či jakýchkoliv jiných objektů, na kterých máme definováno uspořádání, tj. umíme je porovnávat). Tato datová struktura obvykle podporuje následující operace: přidání nového prvku do haldy, odebrání nejmenšího prvku a dotaz na nejmenší prvek. Ukážeme takovou implementaci haldy, že pokud halda obsahuje N čísel (prvků), potom na přidání jednoho prvku, či odebrání nejmenšího prvku potřebujeme čas $O(\log N)$ a na zjištění hodnoty nejmenšího prvku v haldě konstantní čas $O(1)$.

Implementace haldy v programu, která má výše uvedené parametry, může být například tato: Pokud halda obsahuje N prvků, pak máme její prvky uloženy v poli na pozicích 0 až $N-1$. Prvek na pozici s indexem k má dva *následníky*, a to prvky na pozicích $2k+1$ a $2k+2$. Pokud je k příliš velké, $2k+2 > N-1$, pak má prvek na k-té pozici jednoho či dokonce žádného následníka. Prvek na pozici $\lfloor (k-1)/2 \rfloor$ (celá část čísla $(k-1)/2$) pak nazýváme *předchůdcem* prvku na pozici k. Ti z vás, kteří znají binární stromy, jistě v tom, co bylo řečeno, rozpoznali možnost, jak v poli uchovávat vyvážené binární stromy (následníci jsou potomci a předchůdci jsou rodiče v obvyklé stromové terminologii).

My však v poli nebudeme uchovávat prvky haldy v libovolném pořadí. V každém okamžiku bude platit následující invariant (podmínka): Každý

prvek pole je menší než kterýkoliv z jeho (nejvýše dvou) následníků. Takže naše halda může vypadat například takto:

0	1	2	3	4	5	6	7	8
5	6	20	25	7	21	22	26	27

Z toho, co jsme právě popsali, je jasné, že nejmenší prvek je uložen na pozici s indexem 0, a tedy můžeme snadno v konstantním čase zjistit jeho hodnotu. V následujícím odstavci popíšeme, jak lze do haldy prvky rychle přidávat a naopak je z haldy odebírat.

Zaměřme se nejprve na to, jak lze prvek do haldy přidat. Jestliže halda obsahuje N prvků, pak nový prvek, říkejme mu třeba X, nejprve umístíme na konec pole, tj. na pozici s indexem N. Nyní prvek X porovnáme s jeho předchůdcem. Pokud je jeho předchůdce menší, pak je vše v pořádku a jsme hotovi. V opačném případě X prohodíme s jeho předchůdcem. Nyní je X zřejmě menší než kterýkoliv z jeho následníků, ale stále by mohl být menší než jeho nový předchůdce. Takže X porovnáme s jeho současným předchůdcem a pokud je X opět menší, pak tyto dva prvky prohodíme. Takto pokračujeme, dokud současný předchůdce X není menší než X, nebo X žádného předchůdce nemá (tj. X je na pozici 0). Protože se v každém kroku index pozice, kde se prvek X právě nachází, zmenší alespoň na polovinu, provedeme celkově nejvýše $O(\log N)$ výměn, a spotřebujeme tedy čas $O(\log N)$.

Odebrání nejmenšího prvku může probíhat takto: Prvek z poslední pozice (tj. z pozice $N-1$) přesuneme na pozici 0. Místo s předchůdcem jej však porovnáme s jeho následníky a v případě, že je větší než některý z nich, prohodíme ho s ním (pokud je větší než oba jeho následníci, pak ho prohodíme s menším z nich). Protože se v každém kroku index „bublajícího“ prvku v poli alespoň zdvojnásobí, opět spotřebujeme čas $O(\log N)$.

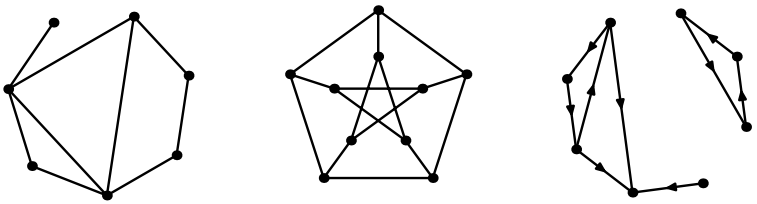
Povšimněme si, že pokud bychom si pro každý prvek pamatovali jeho umístění v poli, pak by bylo možné libovolný prvek odebrat v čase $O(\log N)$ – stačilo by dát na jeho místo prvek s indexem N a výměnami (dopředu či dozadu) zajistit, aby prvky v poli stále splňovaly naši podmínku.

Ukázku implementace haldy si můžete prohlédnout na konci naší kuchařky.

Dijkstrův algoritmus

Dijkstrův algoritmus se používá pro hledání nejkratších cest v grafu. Graf si můžeme představovat jako nějaké body, kterým říkáme *vrcholy*, spojené navzájem čarami, kterým říkáme *hrany*. V našem případě budou mít hrany přiřazeny *váhy*, tedy něco jako délku čáry. *Cestou* nazveme posloupnost vrcholů $v_1 v_2 \dots v_d$ takovou, že každé dva po sobě jdoucí vrcholy jsou spojeny hranou; *délka cesty* bude součet vah hran spojujících dvojice po sobě následujících vrcholů. Graf si můžeme představit například jako města spojená silnicemi, kde váha je délka silnice; délka cesty je vzdálenost, kterou ujedeme mezi městy. Někdy se setkáme s *orientovanými* grafy, ve kterých mají hrany přiřazenu orientaci, tj. směr z jednoho vrcholu do druhého; je po nich možné cestovat pouze v tomto směru (jednosměrné silnice). Příklady několika grafů jsou zakresleny na obr. 1. Všimněte si, že v nákresu grafu se mohou hrany křížit, i když nemají společný vrchol (samozřejmě, že takovéto křížení není dovoleno použít k odbočení na cestě grafu).

Dijkstrův algoritmus nalezne v grafu nejkratší cestu mezi dvěma zadanými vrcholy za předpokladu, že váhy všech hran jsou nezáporné. Ve skutečnosti tento algoritmus dělá o malinko více: Najde nejkratší cesty z jednoho zadaného vrcholu do všech ostatních.



Obr. 1. Ukázky grafů

Nechť v_0 je vrchol grafu, ze kterého chceme určit délky nejkratších cest. Budeme si udržovat pole délek zatím nalezených cest z vrcholu v_0 do všech ostatních vrcholů grafu. Navíc u některých vrcholů budeme mít poznamenáno, že cesta nalezená do nich je už ta nejkratší možná. Takovým vrcholům budeme říkat *trvale ohodnocené*. Na začátku inicializujeme v poli všechny hodnoty na ∞, kromě hodnoty odpovídající vrcholu v_0, kterou inicializujeme na 0 (délka nejkratší cesty z v_0 do v_0 je zřejmě rovna 0). V každém *kroku* algoritmu pak provedeme toto: Ze všech vrcholů, které nejsou trvale ohodnocené, vybereme takový vrchol w, pro

který je délka zatím nalezené cesty nejkratší. Vrchol w prohlásíme za trvale ohodnocený. Dále pro každý vrchol v, který není trvale ohodnocený, otestujeme, zda cesta z vrcholu v_0 do w a pak po hraně z w do v není kratší než zatím nalezená cesta z v_0 do v. Je-li tomu tak, změníme délku zatím nalezené cesty do v.

Algoritmus skončí, jsou-li všechny vrcholy trvale ohodnocené, nebo pokud všechny vrcholy, které nejsou trvale ohodnocené, mají délku cesty rovnou ∞ (v takovém případě se graf skládá z více nesouvislých částí).

Než dokážeme, že popsaný algoritmus opravdu nalezne délky nejkratších cest z vrcholu v_0, zamysleme se nad jeho časovou složitostí: K uchovávání délek dosud nalezených cest použijeme haldu. Halda bude na začátku obsahovat N prvků a v každém kroku se počet jejích prvků sníží o jeden. Celý algoritmus má nejvýše N kroků, kde N je počet vrcholů vstupního grafu. V každému kroku musíme zkontrolovat tolik vrcholů v, kolik hran vede z vrcholu w. Každá taková kontrola může vyústit ve vyjmutí a přidání prvku v haldě, tj. můžeme na ni potřebovat čas $O(\log N)$. Počet takových změn pro všechny kroky dohromady je nejvýše $O(M)$, kde M je počet hran vstupního grafu. Celková časová složitost našeho algoritmu je tedy $O\big((N+M)\log N\big)$.

Může se samozřejmě stát, že graf má hodně hran, až kvadraticky mnoho v počtu vrcholů N. V takovém případě je lepší haldu vůbec nepoužít, ohodnocení vrcholů si uchovávat v normálním poli a v každém kroku určit vrchol w v čase $O(N)$ prostým výběrem nejmenší hodnoty z trvale neohodnocených vrcholů. Časová složitost této implementace Dijkstrova algoritmu, jejíž kód lze najít na konci kuchařky, je $O\big(N^2\big)$. Poznamenejme, že použitím jiného druhu haldy, tzv. Fibonacciho haldy, lze zlepšit časovou složitost Dijkstrova algoritmu až na $O(M + N\log N)$.

Vraťme se k důkazu správnosti Dijkstrova algoritmu. Dokážeme, že algoritmus skutečně nalezne nejkratší cesty z vrcholu v_0.

Ukážeme, že po každém kroku algoritmu platí následující tvrzení: Nechť A je množina trvale ohodnocených vrcholů. Pak délka dosud nalezené cesty z v_0 do v (v je libovolný vrchol grafu, ne nutně z množiny A) je délka nejkratší cesty $v_0v_1\ldots v_kv$ takové, že všechny vrcholy v_0, v_1, …, v_k jsou v množině A. Tvrzení dokážeme indukcí podle počtu kroků algoritmu, které již proběhly.

Tvrzení zřejmě platí před a po prvním kroku algoritmu. Nechť w je vrchol, který byl v předchozím kroku prohlášen za trvale ohodnocený.

Uvažujme nejprve vrchol v, který je trvale ohodnocený. Pokud $v = w$, je tvrzení triviální. V případě, kdy $v \neq w$, ukážeme, že existuje nej-

kratší cesta z v_0 do v přes vrcholy z A, která nepoužívá vrchol w. Označme D délku nejkratší cesty z v_0 do v přes vrcholy z $A \setminus \{w\}$. Protože v každém kroku vybíráme vrchol s nejmenším ohodnocením a ohodnocení vybraných vrcholů v jednotlivých krocích tvoří neklesající posloupnost (váhy hran jsou nezáporné), je délka cesty z v_0 do w přes vrcholy z A alespoň D, a tedy délka libovolné cesty z v_0 do v přes w používající vrcholy z A je alespoň D. Taková cesta proto nemůže být kratší než D. Existuje tedy nejkratší cesta z v_0 do v přes vrcholy z A.

Uvažujme nyní vrchol v, který ještě není trvale ohodnocený. Nechť $v_0 v_1 \dots v_k v$ je nejkratší cesta z v_0 do v taková, že všechny vrcholy v_0, v_1, …, v_k jsou v množině A. Pokud $v_k = w$, pak jsme ohodnocení v změnili na délku této cesty v právě proběhlém kroku. Pokud $v_k \neq w$, pak můžeme předpokládat, že žádný z vrcholů v_1, …, v_k není w (na konci minulého odstavce jsme ukázali, že do každého vrcholu $v' \in A$, $v' \neq w$, existuje nejkratší cesta přes vrcholy z $A \setminus \{w\}$). Vidíme tedy, že délka cesty do v se rovnala správné hodnotě už před právě proběhlým krokem.

Vzhledem k tomu, že po posledním kroku obsahuje množina A právě ty vrcholy, do kterých existuje cesta z vrcholu v_0, dokázali jsme, že algoritmus funguje správně.

Na závěr poznamenejme, že Dijkstrův algoritmus funguje i pro orientované grafy a že jej lze snadno upravit tak, aby kromě určení délky nejkratší cesty takovou cestu také našel: U každého vrcholu si v okamžiku, kdy mu měníme ohodnocení, poznamenáme, ze kterého vrcholu do něj přicházíme. Nejkratší cestu do nějakého vrcholu pak zrekonstruujeme tak, že u posledního vrcholu této cesty zjistíme, který vrchol je předposlední, u předposledního, který je předpředposlední, atd.

Implementace haldy

```
var halda:array[0..MAX] of integer;
    N: word; { počet prvků v haldě }

procedure chyba; { něco je špatně }

function nejmensi:integer;
begin
  if N=0 then chyba;
  nejmensi:=halda[0]
end;
```

```
procedure vloz(prvek: integer);
var i,j:word;
    x:integer;
begin
  if N=MAX then chyba;
  i:=N; N:=N+1;
  halda[i]:=prvek;
  while (i>0) and (halda[(i-1) div 2]>halda[i]) do
    begin
      j:=(i-1) div 2;
      x:=halda[j]; halda[j]:=halda[i]; halda[i]:=x;
      i:=j;
    end
end;
procedure zrus_nejmensi;
var i,j:word;
x:integer;
begin
  if N=0 then chyba;
  halda[0]:=halda[N-1];
  N:=N-1; i:=0;
  while 2*i+1<=N-1 do
    begin
      j:=i;
      if (2*i+1<=N-1) and (halda[j]>halda[2*i+1])
        then j:=2*i+1;
      if (2*i+2<=N-1) and (halda[j]>halda[2*i+2])
        then j:=2*i+2;
      if i=j then break;
      x:=halda[i]; halda[i]:=halda[j]; halda[j]:=x;
      i:=j
    end
end;
```

Implementace Dijkstrova algoritmu

```
var N: word;
          { počet vrcholů }
```

```
    vahy: array[1..MAX,1..MAX] of integer;
          { váhy hran, -1 = hrana neexistuje }
    delky: array[1..MAX] of integer;
          { délky zatím nalezených cest, -1 = nekonečno }
    trvaly: array[1..MAX] of boolean;
          { trvale ohodnocen? }

procedure Dijkstra(odkud: word);
var i: word;
    w,v: word;
begin
  for i:=1 to N do
    begin
      trvaly[i]:=false;
      delky[i]:=-1;
    end;
  delky[odkud]:=0;
  repeat
    w:=-1;
    for i:=1 to N do
      if not(trvaly[i]) and (delky[i]<>-1) then
        if w=-1 then
          w:=i
        else
          if delky[i]<delky[w] then
            w:=i;
    if w<>-1 then
      begin
        trvaly[w]:=true;
        for i:=1 to N do
          if (vahy[w][i]<>-1) and not(trvaly[i]) and
          { podmínku "not(trvaly[i])" lze vypustit }
          (delky[w]+vahy[w][i]<delky[i]) then
            delky[i]:=delky[w]+vahy[w][i]
      end
  until w=-1;
end;
```