Rozhledy matematicko-fyzikální

Daniel Kráľ; Martin Mareš; Milan Straka

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

Rozhledy matematicko-fyzikální, Vol. 82 (2007), No. 3, 18–23

Persistent URL: http://dml.cz/dmlcz/146093

Terms of use:

© Jednota českých matematiků a fyziků, 2007

Institute of Mathematics of the Czech Academy of Sciences provides access to digitized documents strictly for personal use. Each copy of any part of this document must contain these *Terms of use*.

This document has been digitized, optimized for electronic delivery and stamped with digital signature within the project *DML-CZ: The Czech Digital Mathematics Library* http://dml.cz

Recepty z programátorské kuchařky Korespondenčního semináře z programování, V. část*)

Daniel Král', Martin Mareš a Milan Straka, MFF UK Praha

V pátém dílu našeho povídání o zajímavých algoritmech a datových strukturách si vysvětlíme návrh algoritmů pomocí techniky *dynamického programování*. Tato technika je založena na jednoduché myšlence: vyřešme danou úlohu nejprve pro zadání menší velikosti a poté tato řešení zkombinujme dohromady. Dynamické programování si předvedeme na dvou příkladech. Na konci kuchařky pak naleznete několik otázek a problémů, na kterých si můžete vyzkoušet, jak jste našemu povídání porozuměli.

Prvním z našich dvou příkladů je úloha známá jako *problém batohu.* Je dáno N předmětů o celočíselných hmotnostech $m_1, \ldots, m_N$ a celé číslo M (nosnost batohu) a úkolem je vybrat některé z předmětů tak, aby součet jejich hmotností byl co největší, ale zároveň nepřekročil M. Předvedeme si algoritmus, který tento problém řeší a jehož časová složitost je $O(MN)$.

Během výpočtu budeme používat pomocné pole $A[i]$, $i = 0, \ldots, M$, a celý výpočet bude rozdělen do N kroků. Na konci k-tého kroku bude prvek $A[i]$ nenulový (což odpovídá hodnotě `true`), jestliže z prvních k předmětů lze vybrat předměty, jejichž součet hmotností je přesně i. Před prvním krokem (po nultém kroku) jsou všechny hodnoty $A[i]$ pro $i > 0$ nulové a $A[0]$ má nějakou nenulovou hodnotu, řekněme -1. Všimněme si, jak kroky algoritmu odpovídají podúlohám, které řešíme: v prvním kroku vyřešíme podúlohu odpovídající úloze, kdyby byl k dispozici jen první předmět, v druhém kroku vyřešíme podúlohu s prvním a druhým předmětem, v třetím s prvními třemi předměty atd.

*) Informace o Korespondečním semináři z programování pořádaného Matematicko-fyzikální fakultou Univerzity Karlovy v Praze lze nalézt na webové stránce `http://ksp.mff.cuni.cz`.

Popišme si nyní k-tý krok algoritmu. Pole A budeme procházet od konce, tj. od $i = M$. Pokud je hodnota $A[i]$ stále nulová, ale hodnota $A[i - m_k]$ je nenulová, změníme hodnotu uloženou v $A[i]$ na k (později si vysvětlíme, proč hodnotu $A[i]$ volíme tímto způsobem). Nyní si rozmyslíme, že po provedení k-tého kroku odpovídají nenulové hodnoty v poli A hmotnostem podmnožin z prvních k předmětů. Pokud je hodnota $A[i]$ nenulová, pak buď byla nenulová před k-tým krokem (a v tom případě odpovídá hmotnosti nějaké podmnožiny prvních $k-1$ předmětů), anebo se stala nenulovou v k-tém kroku. Potom ale hodnota $A[i - m_k]$ byla před k-tým krokem nenulová, a tedy existuje podmnožina prvních $k - 1$ předmětů, jejíž hmotnost je $i - m_k$. Přidáním k-tého předmětu k této podmnožině vytvoříme podmnožinu předmětů hmotnosti přesně i. Naopak pokud lze vytvořit podmnožinu X hmotnosti i z prvních k předmětů, pak takovou podmnožinu X lze buď vytvořit jen z prvních $k - 1$ předmětů, a tedy hodnota $A[i]$ je nenulová již před k-tým krokem, anebo k-tý předmět je obsažen v takové množině X. Potom ale hodnota $A[i - m_k]$ je nenulová před k-tým krokem (hmotnost podmnožiny X bez k-tého prvku je $i - m_k$) a hodnota $A[i]$ se stane nenulovou v k-tém kroku.

Po provedení všech N kroků odpovídají nenulové hodnoty $A[i]$ přesně hmotnostem podmnožin ze všech předmětů, co máme k dispozici. Speciálně největší index i_0 takový, že hodnota $A[i_0]$ je nenulová, odpovídá hmotnosti „nejtěžší“ podmnožiny předmětů, která nepřekročí hmotnost M. Nalézt jednu množinu této hmotnosti také není obtížné: protože v k-tém kroku jsme měnili nulové hodnoty v poli A na hodnotu k, tak v $A[i_0]$ je uloženo číslo jednoho z předmětů nějaké takové množiny, v $A[i_0 - m_{A[i_0]}]$ číslo dalšího předmětu atd. Zdrojový kód tohoto algoritmu lze nalézt níže.

Časová složitost algoritmu je $O(NM)$, neboť se skládá z N kroků, z nichž každý vyžaduje čas $O(M)$. Paměťová složitost činí $O(N+M)$, což představuje paměť potřebnou pro uložení pomocného pole A a hmotností daných předmětů.

```
var N: word; { počet předmětů }
  M: word; { hmotnostní omezení }
  hmotnost: array[1..N] of word;
    { hmotnosti daných předmětů }
  A: array[0..M] of integer;
  i, k: word;
```

```
begin
  A[0]:=-1;
  for i:=1 to M do A[i]:=0;
  for k:=1 to N do
    for i:=M downto hmotnost[k] do
      if (A[i-hmotnost[k]]<>0) and (A[i]=0) then A[i]:=k;
  i:=M;
  while A[i]=0 do i:=i-1;
  writeln('Maximální hmotnost: ',i);
  write('Předměty v množině:');
  while A[i]<>-1 do
    begin
      write(' ',A[i]);
      i:=i-hmotnost[A[i]];
    end;
  writeln;
end.
```

Náš druhý příklad bude z oblasti grafových algoritmů. Floyd-Warshallův algoritmus pro nalezení nejkratších cest mezi všemi vrcholy grafu — my se ale pokusíme bez definice grafu jak v zadání, tak v řešení tohoto příkladu obejít. Připomeňme však, že v prvním dílu kuchařek [1] jsme si ukázali algoritmus, Dijkstrův algoritmus, pro nalezení nejkratších cest v grafu z jednoho vrcholu do všech ostatních vrcholů. Tento algoritmus pracuje v čase $O(N^2)$, kde N je počet vrcholů grafu. Nejkratší cesty mezi všemi vrcholy grafu bychom pomocí tohoto algoritmu, kdybychom jej spustili pro každý vrchol zvlášť, nalezli v čase $O(N^3)$.

Vstupem algoritmu je N měst. Mezi některými dvojicemi měst vedou (obousměrné) silnice, jejichž (nezáporné) délky jsou též dány na vstupu. Předpokládáme, že silnice se jinde než ve městech nepotkávají (pokud se kříží, tak mimoúrovňově). Úkolem je spočítat nejkratší vzdálenosti mezi všemi dvojicemi měst, tj. délky nejkratších cest mezi všemi dvojicemi měst. *Cestou* rozumíme posloupnost měst takovou, že každá dvě po sobě následující města jsou spojená silnicí, a délka cesty je součet délek silnic, které tato města spojují.

Vzdálenosti mezi městy jsou na začátku algoritmu uloženy ve dvourozměrném poli D, tj. $D[i][j]$ je vzdálenost z města i do města j. Pokud mezi městy i a j nevede žádná silnice, bude $D[i][j] = \infty$ (v programu bude tato hodnota rovna nějakému dostatečně velkému číslu). V průběhu

výpočtu si budeme na pozici $D[i][j]$ udržovat délku nejkratší dosud nalezené cesty mezi městy i a j.

Algoritmus se skládá z N fází. Na konci k-té fáze bude v $D[i][j]$ uložena délka nejkratší cesty mezi městy i a j, která prochází přes některá z měst 1 až k. V průběhu k-té fáze tedy stačí vyzkoušet, zda pro města i a j je kratší stávající cesta přes města $1, \ldots, k-1$, jejíž délka je uložena v $D[i][j]$, anebo existuje kratší cesta přes město k. Pokud taková cesta existuje, pak její část do města k je nejkratší cesta z i do k přes města $1, \ldots, k-1$ a její část z města k je nejkratší cesta z k do j přes města $1, \ldots, k-1$. Délka takové cesty je tedy rovna $D[i][k] + D[k][j]$. V k-té fázi tedy stačí pro všechny dvojice i a j vyzkoušet, zda je součet $D[i][k] + D[k][j]$ menší než stávající hodnota $D[i][j]$, a pokud ano, nahradit hodnotu $D[i][j]$ tímto součtem.

Z popisu algoritmu přímo plyne, že po N-té fázi je na pozici $D[i][j]$ uložena délka nejkratší cesty z města i do města j. Protože v každé z N fází algoritmu musíme vyzkoušet všechny dvojice i a j, vyžaduje každá fáze čas $O(N^2)$. Celková časová složitost našeho algoritmu tedy je $O(N^3)$. Paměťová složitost algoritmu je $O(N^2)$. Popišme si ještě, jak bychom postupovali, kdybychom kromě vzdáleností mezi městy chtěli nalézt i nejkratší cesty mezi nimi. To lze jednoduše vyřešit například tak, že si budeme udržovat další pomocné pole $E[i][j]$, do kterého při změně hodnoty $D[i][j]$ uložíme nejvyšší číslo města na cestě z i do j délky $D[i][j]$ (při změně v k-té fázi je to číslo k). Máme-li pak vypsat nejkratší cestu z i do j, vypíšeme nejprve cestu z i do $E[i][j]$ a pak cestu z $E[i][j]$ do j. Tyto cesty nalezneme stejným (rekurzivním) postupem.

```
var N:word;  počet měst
  D:array[1..N] of array[1..N] of longint;
{ délky silnic mezi městy, D[i][i]=0,
místo neexistujících je "nekonečno" }
  E:array[1..N] of array[1..N] of word;
{ nejvyšší číslo města na nejkratší cestě
z i do j; E[i][j]=0 znamená, že nejkratší
cesta používá pouze silnici z i do j }
  i,j,k:word;
begin
  for i:=1 to N do
    for j:=1 to N do E[i][j]:=0;
  for k:=1 to N do
```

```
    for i:=1 to N do
      for j:=1 to N do
        if D[i][k]+D[k][j] < D[i][j] then
          begin
            D[i][j]:=D[i][k] + D[k][j];
            E[i][j]:=k;
          end;
end.
```

Pro vypsání nejkratší cesty z města i do města j bychom pak použili následující proceduru:

```
procedure cesta(i: word; j: word);
begin
  if (E[i][j]=0) then
    begin
      writeln('Jeďte po silnici
                z města ',i,' do města ',j,'.');
      exit
    end;
  cesta(i,E[i][j]);
  cesta(E[i][j],j);
end;
```

Na závěr si uveďme několik otázek a problémů souvisejících s technikou dynamického programování a dnes předvedených algoritmů.

1. Jak byste řešili problém batohu, ve kterém by každý předmět kromě hmotnosti měl i (nezápornou) cenu a úkolem by bylo nalézt množinu předmětů, jejichž hmotnost je nejvýše M a jejichž součet cen je co nejvyšší? Snažíme se tedy dosáhnout co nejvyšší ceny naložených předmětů (ne hmotnosti jako v našem povídání).
2. Jak je třeba změnit Floyd-Warshallův algoritmus, jestliže silnice mezi městy budou jednosměrné?
3. Proč není dobře, když „nekonečno“ ve Floyd-Warshallově algoritmu bude `maxint`? Zamyslete se, co by pak bylo součtem dvou takových „nekonečen“ v podmínce ve vnitřním cyklu algoritmu.
4. Proč musíme ve Floyd-Warshallově algoritmu předpokládat, že délky silnic jsou nezáporné?
5. V minulé kuchařce [2] jsme si vysvětlili, co jsou binární vyhledávací stromy. Připomeňme, že doba vyhledání prvku v binárním vyhledávacím stromě je přímo úměrná jeho vzdálenosti od kořene. Předpo-

kládejte, že máte dáno N uspořádaných prvků a u každého z nich je dána četnost a_i, $i = 1, \ldots, N$, jak často bude tento prvek potřeba vyhledávat. Navrhněte algoritmus, který vytvoří binární vyhledávací strom, že *průměrná* doba vyhledání prvku ve stromě bude minimální, tj. součet $\sum_{i=1}^{n} a_i \ell_i$ bude nejmenší možný, kde ℓ_i je délka cesty ve stromě od kořene k i-tému prvku.

6. Je dáno N měst, z nichž některá jsou spojena silnicí. Délky silnic mezi městy jsou též dány. Vaším úkolem je navrhnout okružní cestu přes města tak, aby každé město bylo navštíveno alespoň jednou a celková délka cesty byla co nejkratší.
   Nejjednodušším řešením je samozřejmě vyzkoušet všech $N!$ možných pořadí návštěv měst a mezi jednotlivými po sobě následujícími městy se přesunout po nejkratší cestě mezi těmito městy. Tento algoritmus má časovou složitost $O(N!) = 2^{O(N \log N)}$. Pomocí dynamického programování lze však navrhnout algoritmus, jehož časová složitost je jen $2^{O(N)}$.

Literatura

[1] Kára, J., Kráľ, D., Mareš, M.: Recepty z programátorské kuchařky Korespondenčního semináře z programování – I. část. *Rozhledy matematicko-fyzikální* **80**, 1 (2005), 26–33.

[2] Kráľ, D., Mareš, M., Valla, T.: Recepty z programátorské kuchařky Korespondenčního semináře z programování – IV. část. *Rozhledy matematicko-fyzikální* **82**, 1 (2007), 22–35.

* * * * *

I když nezdobí mě vlasy žádné,
nosit hřeben důvody mám pádné:
Užívám ho k učesání myšlenek,
aby byly hezké aspoň navenek.
Občas v hlavě mi však nápad vzejde,
který vůbec učesati nejde.
Pro mé přátele to není novina:
Napadla ho zas nějaká konina!

Emil Calda[*)]

*) *Úvod do obecné teorie prostoru,* Karolinum, Praha, 2003