Динамичко програмирање 1

Динамичкото програмирање обично се применува во проблемите на оптимизација: проблемот може да има многу решенија, секое решение има вредност, а се бара решение кое што има оптимална (најголема или најмала) вредност. Во случај да постојат повеќе решенија кои што имаат оптимална вредност, обично се бара кое било од нив. Во една широка класа на проблеми на оптимизација, едно од оптималните решенија може да се најде користејќи го динамичкото програмирање.

Поим и историјат

Динамичко програмирање претставува начин на програмирање односно тип на алгоритми, со помош на кои се доаѓа до **оптимални вредности** на широка класа на проблеми од определен тип, така што алгоритмот во повеќето случаи претставува **оптимално решение на проблемот**.

Самиот збор "програмирање" слично како и кај линеарното програмирање се однесува на пополнување на табели при решавање на проблемот, а не на употребата на компјутери и програмски јазици. Техниките на оптимизација кои имаат елементи на динамичко програмирање биле познати и порано, но денеска за автор на методот се смета професор *Richard E. Belman.* Во средина на 50-те години *Belman* го проучувал динамичкото програмирање и дал цврста математичка основа за овој начин на решавање на проблемите.

Воопштено зборувајќи, проблемот се решава така што се воочува хиерархијата на проблемите од ист тип, содржани во главниот проблем и решавањето започнува од наједноставните проблеми. Вредностите и деловите на решенијата на сите решени потпроблеми се паметат во табела, па потоа со нивните комбинации се добиваат решенија на поголемите потпрограми се до решение на главниот проблем.

Идеја за реализација

Да ја илустрираме идејата на еден од најпознатите проблеми на динамичкото програмирање - проблемот на ранец *(knapsack problem)*.

1) Проблем на ранец

За овој проблем постојат неколку познати варијанти и секоја може да се реши со повеќе формулации. Во продолжение ја даваме онаа формулација (една од варијантите) по која проблемот го добил името: крадец со ранец во кој може да се сместат N волуменски единици, влегол во просторија во која што се чуваат скапоцени предмети.

Во просторијата има вкупно M типови на предмети, секој во многу големи количини (повеќе отколку што може да се собере во ранецот). За секој тип на предмет позната е неговата вредност $V(k)$ и неговиот волумен $Z(k)$, $k=1,M$. Сите големини се целобројни. Крадецот сака да го наполни ранецот со најскапоцена содржина. Потребно е да се одредат предметите кои треба да ги стави во ранецот и нивната вкупна вредност.

Решение:

Најнапред да направиме неколку важни забелешки за подобро да се објасни суштината на проблемот.

- Ранецот кој што е оптимално пополнет не мора да биде пополнет до врвот, или попрецизно, збирот на волуменот на предметите ставени во ранецот не мора да биде еднаков на волуменот на ранецот. Важно е тој збир да не е поголем од волуменот на ранецот, а во исто време збирот на вредностите на тие предмети да биде максимален. На пример, нека ранецот има капацитет од $N=7$ волуменски единици и нека постојат $M=3$ предмети такашто $V=(3,4,8)$ и $Z=(3,4,5)$ односно првиот предмет има вредност 3 и волумен 3, вториот вредност 4 и волумен 4, а третиот предмет има вредност 8 и волумен 5. Една варијанта за пополнување на ранецот е со полнење на ранецот до врвот така што во него ги ставаме првиот и вториот предмет бидејќи $z_1+z_2=3+4=7=N$. Сепак, таквото пополнување не е оптимално бидејќи неговата вредност е $v_1+v_2=3+4=7$, додека со ставање на третиот предмет во ранецот се добива ранец со повредна содржина $v_3=8$, а при тоа ранецот не е пополнет до врвот $(z_3=5<7=N)$.
- Врз основа на претходниот пример, се добива впечаток дека до решението се доаѓа така што се одредува k за кое $V(k)/Z(k)$ е најголемо, па ранецот се полни само со предметот k. Овој начин не претставува решение што исто така ќе го покажеме на поедноставен пример: нека е $N=7$, $M=3$, $V=(3,4,6)$, $Z=(3,4,5)$. Наведената идеја (алчен избор) наложува да се избере третиот предмет, како највреден по единица волумен. Според тоа во ранецот би се ставил само еден од предметите од третиот тип (во иднина: предмет број 3), а вредноста на ранецот би била еднаква на 6. Лесно може да се согледа дека изборот на првите два предмети дава вредност на рабецот 7, што е подобро (како и оптимално) решение. Идејата за алчен избор води кон решение во случај да не мора да се земаат цели предмети и вредноста на дел од некој предмет да е сразмерна со големината на тој дел.

Карактеристики на решението

Од овој пример се заклучува дека проблемот на ранец не е тривијален и до решението не може да се дојде директно. Потребна е повнимателна анализа својствена на проблемот и решението, на основа која што подоцна ќе го конструира решението.

Анализирајќи го проблемот можеме да се согледа следното: ако при оптимално пополнување на ранецот последен избран предмет е x, тогаш претходно избраниот предмет на оптимален начин го пополнува ранецот со капацитет N-z_x**.** Овој заклучок лесно се докажува со сведување на контрадикција. Имено да претпоставиме дека постои друг начин подобро да се пополни ранецот со капацитет N-z_x. Нека на потполно ист начин се пополнат и првите N-z_x единици волумен на ранецот со големина N и потоа нека го додадеме x-тиот предмет. Со тоа се добива пополнување на целиот ранец, кој е подобар од почетниот, што е невозможно бидејќи почетното пополнување е оптимално.

Според тоа, **оптималното решение на проблемот содржи во себе оптимални решенија на проблемите од истиот тип содржани во главниот проблем**. Вообичаено е во ваков случај да се каже дека проблемот има оптимална подструктура. Наведеното својство се нарекува и својство на *Belman*, по авторот на методите на динамичкото програмирање. Ова е клучна карактеристика за примена на динамичко програмирање. Благодарејќи на оваа карактеристика, можеме да дојдеме до оптималното решение на проблемот, комбинирајќи со оптималните решенија на проблемите.

Ќе докажеме, дека користејќи ја математичката индукција, за секое целобројно N (и дадените типови на предмети, кои не се менуваат) умееме да најдеме најдобро пополнување на ранецот со капацитет N. Оптималното решение за $N=0$ е празен ранец. Нека се познати решенијата за сите ранци со капацитет q и нека е $B(q)$ збир на сите вредности, а $S(q)$ низа на редни броеви на предметите ставени во ранец со капацитет q при оптимален избор.

Секој од M предметите кои што можат да се соберат во празен ранец со капацитет N, да го испробаме како последен избран за ранец со капацитет N. Остатокот на ранецот во секој од овие случаи да ги пополниме на оптимален начин што можеме на основа на индуктивната хипотеза. Најголемата добиена вредност од сите ранци на капацитетот N ќе биде оптимална. Оваа претпоставка следува директно на основа на оптималноста на подструктурите, бидејќи еден предмет мора да биде последен, а ние го испробуваме секој како последен.

Според досега кажаното, решението за ранецот со капацитет N е:

$$B(N) = \max_{\substack{1 \le x \le M \\ Z(x) \le N}} \{V(x) + B(N - Z(x))\}, \quad S(N) = S(N - Z(x_N)) \cup \{x_N\}$$

каде што x_N е она x кое остварува максимум во $B(N)$. Да забележиме дека нема потреба да го паметиме целиот збир $S(q)$ за секој капацитет на ранецот q, односно доволно е како член на низата $C(q)$ да се запамти последниот додаден елемент x_N. Сите елементи тогаш може да се најдат со редот (од последниот кон првиот) и тоа се:

$$a = C(N),\ b = C(N\text{-}Z(a)),\ c = C(N\text{-}Z(a)\text{-}Z(b)),\ d = C(N\text{-}Z(a)\text{-}Z(b)\text{-}Z(c)), \ldots$$

или додека не се добијат сите предмети од ранецот.

Рекурзивно програмско решение

Задачата може да се реши со употреба на рекурзивна потпрограма *napolni(q,B,C)*, која за ранецот со капацитет q ја наоѓа неговата оптимална вредност B и редниот број на последниот додаден предмет C. Ќе сметаме дека низите V и Z како и бројот на типови на предмети M се глобални големини.

```
procedure napolni(q:integer; var B:integer; var C:integer);
var BB, CC : integer;
begin
            B:=0;
            C:=0;
            if q>0 then
            begin
                 for i:=1 to M do
                      if Z[i]<=q then
                      begin
                            napolni(q-Z[i],BB,CC);
                            if BB+V[i]>B then
                            begin
                                  B:=BB+V[i];
                                  C:=i;
                            end;
                      end;
            end;
end.
```

Од главната програма би се повикала потпрограмата *napolni(N,B,C)*, а секое повикување на ова потпрограма може да предизвика M нови повикувања поради решавање на генерализираните потпроблеми. На пример за $N=1000$, $M=10$ и предметите со волумен од $Z_{min}=5$ до $Z_{max}=10$, би требало помеѓу $M^{N/Zmax} = 10^{100}$ и $M^{N/Zmin} = 10^{200}$ рекурзивни повикувања на потпрограмата *napolni* и тоа само на последното ниво на длабочината на рекурзијата (листови во дрвото на рекурзивните повикувања). Кога секое повикување би траело една наносекунда (10^{-9}s), би требало повеќе од $10^{91}s > 10^{83}$ години, а староста на комплетната вселена се проценува на помалку од 10^{12} години!

Решение со динамичко програмирање

Наместо рекурзивно, проблемот можеме да го решиме и со редови од $q=1$ до $q=N$, пополнувајќи табели за B и C. За секое q потребно е M преминувањена низ циклусот за да се определи последниот избран елемент и најголемата вредност, што значи дека вкупниот број операции е пропорционален со

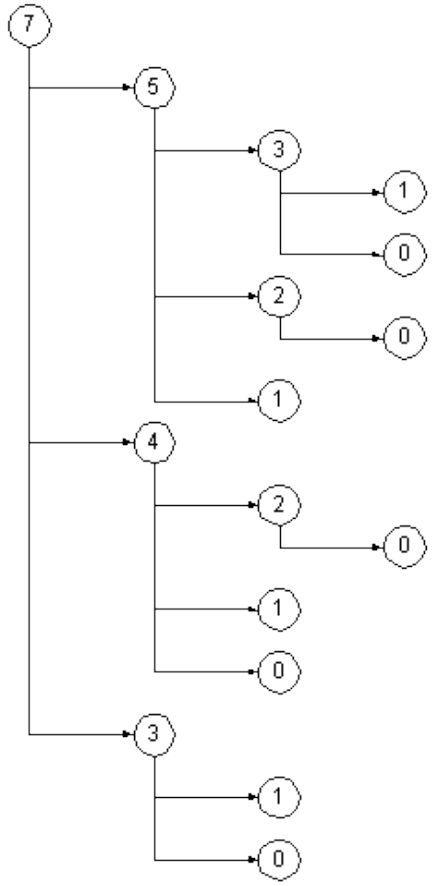
MN. Во горниот пример тоа би било десет илјади циклуси од неколку пресметувачки чекори, што на денешните компјутери се извршува за значително помалку од една секунда.

```
var
            B, C:array[0..1000] of longint;
            V, Z:array[1..10] of integer;
            N, M, q, i : integer;
begin
            readln(N,M);
            for i:=1 to M do
                  readln(V[i], Z[i]);
                  B[0]:=0; C[0]:=0;
                  for q:=1 to N do
                  begin
                       B[q]:=0; C[q]:=0;
                       for i:=1 to M do
                            if Z[i]<=q then
                                  if B[q-Z[i]]+V[i]>B[q] then
                                  begin
                                        B[q]:=B[q-Z[i]]+V[i];
                                         C[q]:=1;
                                  end;
                 end;
            writeln('Optimalna  vrednost  na  sodrzinata  na  ranecot  e',
B[N]);
            writeln('Izbranite  predmeti  se:');  {se  ispisuvaat  nanazad,
sto ne e bitno}
            q:=N;
            while c[q]>0 do
            begin
                  writeln(c[q]);
                   q:=q-z[c[q]];
            end;
end.
```

Од каде ваква драматична разлика во ефикасноста на понудените решенија? Да испитаме подетално на помал пример, како работи рекурзивниот алгоритам. Нека е даден пример за ранец со волумен *N=7* и нека постојат *M=3* предмети со волумен *Z=(2,3,4)*. Вредностите на предметите не се од значење за следење на текот на рекурзивните повикувања. На сликата 1 е прикажана хиерархија на рекурзивните повикувања на потпрограмата *napolni* во облик на дрво. Јазлите на дрвото се наведени по редослед на настанување, т.е. по редослед на повикувања на примерокот на потпрограмата *napolni*, претставен со тие јазли. Ознаките на јазлите претставуваат вредности на аргументот *q*, т.е. на големината на ранецот кој треба да се пополни.

Како што гледаме, при преминувањето по дрвото на рекурзивните повикувања, еден ист проблем (пополнување на ранецот со иста големина *q*) се среќава повеќе пати и секој пат (непотребно) се решава повторно. При тоа бројот на повторените решавања по правило расте експоненцијално со зголемување на димензиите на почетниот проблем. Со други зборови, **потпроблемите имаат заеднички потпотпроблеми, односно делумно се преклопуваат**.

Преклопувањето на потпроблемот е втора особина која е значајна за примена на динамичкото програмирање. Оваа особина не е неопходна за да се примени динамичкото програмирање, но без преклопувања на потпроблемите динамичкото програмирање ја губи својата голема предност во брзина во однос на рекурзијата, бидејќи тогаш со рекурзијата секој потпроблем се креира и решава само еднаш. Кога нема преклопување на потпроблемите, во некои проблеми се случува рекурзивното решение да работи побргу и/или да троши значително малку од меморискиот простор (види глава “Мемоизација”).



Слика 1: Хиерархија на рекурзивни повици на потпрограмата *napolni*.

Варијанти на проблемот на ранец

Веќе е споменато дека проблемот на ранец може да се појави во неколку различни варијанти. Проблемот може да се постави така што за секој предмет да се зададе број на расположливи примероци или така што од секоја врста на предмети да е на располагање точно по еден примерок.

2) Проблем на ранец со само едно појавување на даден предмет

Ќе го разгледаме детално и случајот кога секој предмет може да се земе најмногу еднаш.

Решение:

Нека е познато оптималното пополнување на ранецот со големина N. Ако во тоа пополнување не учествува M-тиот предмет, тогаш истото пополнување е оптимално и за ранецот со големина N и првите $M-1$ предмети. Ако во оптималното пополнување учествува и M-ти предмет, тогаш останатите предмети од ранецот прават оптимално пополнување за ранецот со големина $N-Z(M)$ и првите $M-1$ предмети. Заклучуваме дека проблемот и понатаму има оптимална подструктура, но сега големината на проблемот се задава со два пареметри, а тоа се капацитетот на ранецот и бројот на предметите. Да ја означиме со $B(X,Y)$ најголемата вредност на ранецот со капацитет X, пополнуван со некои од првите Y предмети. Тогаш е:

$$B(X,0) = 0$$

$$B(X,Y) = \begin{cases} B(X,Y-1), & \text{ako e } Z(Y) > X \\ \max \begin{cases} B(X,Y-1) \\ V(Y) + B(X - Z(Y)), Y-1) \end{cases} & \text{ako e } Z(Y) \le X \end{cases}$$

Според тоа, потпроблемите може да се решаваат по следниот редослед: прво за сите ранци и еден предмет, па за сите ранци и два предмети, итн. По решавањето на *B(N,M)* имаме решение на почетниот проблем.

И тука како и во претходната варијанта, поради реконструкција на решението доволна е дополнителна низа *C*, каде што во *C(X)* ќе го памтиме последниот ставен предмет при оптимално пополнување на ранецот со капацитет *X*.

При пополнување на матрицата *B* по колоните се користат само вредности од претходната колона (т.е. на колоната *Y-1)*. Благодарејќи на тоа, наместо матрицата *B* со *M* колони можеме да ја користиме матрицата само со две колони, што е значителна заштеда на просторот, која што може пресудно да делува на применливоста на постапката (за некои вредности *M* и *N*). Разгледувајќи ја повнимателно релацијата по која се пресметува *B(X,Y)*, гледаме дека потребните елементи од претходната колона, сите имаат реден број на врста помал или еднаков на *X*. Со тоа при пополнување на *Y*-та колона на матрицата *B* наназад (од последната врста кон првата), можеме сите операции да ги изведеме во иста колона, па за чување на потребните податоци за доволна низа *B* (ако се користи на опишаниот начин). Во продолжение следува програмата:

```
var
            b, c:array[0..1000] of longint;
            v, z:array[0..10] of integer;
            N, M, x, y: integer;
begin
            write ('n,m='); readln(N,M);
            for y:=1 to M do readln(v[y], z[y]);
            for x:=1 to N do begin
                        b[x]:=0;
                        c[x]:=0;
            end;
            for y:=1 to M do
                        for x:=N downto 1 do
                                    if z[y]<=x then
                                                if b[x] < v[y]+b[x-z[y]]
then begin
                                                     b[x]:=v[y]+b[x-
z[y]];
                                                           c[x]:=y;
                                    end;
             writeln('Optimalna vrednost na sodrzinata na ranecot e',
b[N]);
            writeln('Izbranite predmeti se:'); {se ispisuvaat nanazad,
sto ne e bitno)                          w:=N;
            while c[x]>0 do begin
                        writeln(c[x]);
                        x:=x-z[c[x]];
            end;
end.
```

При решавање на проблемот со оваа варијанта (секој предмет најмногу еднаш), треба да имаме на ум уште една важна претпоставка: решавањето на проблемот со динамичко програмирање се исплаќа единствено ако бројот на предметот *M* е доволно голем, а капацитетот на ранецот *N* релативно мал, така што *MN* операциијата (колку приближно е потребно за овој начин на решавање) да се изврши побргу отколку да се најдат сите 2^M можни поднизи и нивните збирови на вредностите, односно волуменот.

Да речеме за *N=10000, M=100*, динамичкото програмирање на решението го добиваме без чекање, додека 2^{100} поднизи практично не можеме да формираме. Меѓутоа, во случај на мала вредност на *M*

(на пример десет) и голема вредност на N (на пример сто милиони) подобро е да се испробат сите поднизи.

Проблемот на ранец е од исклучително значење во практиката и неговите решенија често се користат, на пример при транспортот на стока. На натпреварите исто така можат многу често да се сретнат овие или други варијанти на проблемот на ранец со можни усложнувања или поедноставувања. Да ги спомнеме познатите примери:

3) Подниза со даден збир

Од дадената низа на природни броеви A, да се одвои подниза чиј збир на елементите е даден број M или да се испише порака дека таква низа не постои.

Решение:

Со S да ја означиме сумата, а со N бројот на елементите во низата. Можеме да сметаме дека со низата A се зададени такви предмети со кои е $v_1=z_1=a;\ i=1,N$. Сега е јасно проблемот да се сведи на оптимално пополнување на ранецот со капацитет M, при што решенија има ако и само ако вредноста на оптималната содржина на ранецот е еднаква на неговиот волумен M, т.е. ако и само ако ранецот е наполнет до врв.

4) Поднизи со минимална разлика на збирови

Броевите од даената низа на природни броеви A, да се поделат во две групи така што разликата на збировите на елементите во поединте групи да биде минимална.

Решение:

Во овој пример низата A повторно има улога како и низата Z и низата V. Нека S и N имаат исти значења како и во претходниот пример. Решението на проблемот се состои во следново: предметите кои сочинуваат оптимално пополнување на ранецот со капацитет $S/2$ (делењето е целобројно), треба да се спои во една а сите останати предмети во друга група. Ако ранецот е пополнет до врв групите за парно S ќе имаат еднакви збирови. Во спротивно првата група има помал збир, а другата поголем, но тие збирови се најблиску до вредноста $S/2$ а со тоа и еден до друг.

5) Постави загради во алгебарски израз со минуси

Даден е изразот a_1-a_2\-...-a_n, каде што сите броеви во низата се цели. Да се постават загради во овој израз така што вредноста на изразот да биде еднаков со зададен број X.

Решение:

Проблемот може еквивалентно да се постави на следниот начин: броевите од дадената низа на цели броеви A да се поделат во две групи така што a_1 задолжително да припадне во првата, а a_2 во втората група и разликата на збировите на елементите на првата и втората група да биде еднаква на X. Оваа формулација е еквивалентна, бидејќи заградите секогаш може да се постават така што пред броевите од првата група има парен број на минуси, а пред бреовите од втората група непарен број на минуси кои однесуваат на тие броеви. Оваа постапка на поставување загради веројатно е полесно да се изведе, отколку прецизно да се опише, па на читателот му препуштаме после формирањето на двете групи на броеви, заградите да ги постави сам.

Нека S_1 и S_2 се суми на броевите од првата и втората група, а S сума на сите N броеви така што $X=S_1-S_2=2S_1-S$. Сега решавањето на преформулираниот проблем се сведува на избор на некои од броевите $a_3,a_4,\ldots,a_n$ (внимавајте дека a_1 и a_2 се изоставени), така што збирот на избраните броеви е еднаков на $M=(X+S)/2-a_1$, а таа задача е веќе разгледувана.

Резиме

Во сите овие три проблеми, кои се сведуваат на проблем на ранец, варијантата “секој предмет најмногу еднаш”, до решение може да се дојде на потполно ист начин како и во случајот елементите на низата да се цели броеви (а не природни) броеви. Услов елементите на низата да се позитивни, е даден само затоа што во спротивна интерпретација се губи физичката смисла: мора да се воведат негативни волумени и негативни вредности. Особината на низата која е суштински битна во овие три проблеми е да се вредностите на елментите на низата, како и нивниот вкупен број на умерени целобројни големини. За низа од N цели броеви кои се сите по апсолутна вредност помали од G, сумите и/или разликите на елементите на сите поднизи (секој елемент може да се земе директно, со променлив предзнак или да се изостави), можат да имат помалку од $2NG$ различни вреднсоти, било позитивни, било негативни. Типично, за $N=G=100$, со динамичко програмирање лесно се испитува за секој од можните 20000 броеви, дали може да се појави како сума или разлика на некои елементи од низата, позитивни или негативни. Ваква постапка за решавање не би можела да се примени во случајот можните кандидати за збир (или разлика) на некои (или сите) елементи на низата да има за неколку редови големини повеќе, а тоа се случува ако е ограничувањето на елементите на низа многу големо, или ако се допуштат реални броеви. Тогаш мора да се испробат сите поднизи, што значи дека ефикасна постапка во тој случај нема.

На крајот на ова поглавје да сумираме што се беше потребно за решавање на секоја од варијантите на проблемот на ранец (или кој било друг проблем) со динамичко програмирање:

- Ја анализираме структурата на оптималното решение. Треба да се согледа дека извесни делови на едно оптимално решение на проблемот всушност е оптималното решение на помали проблеми од истиот тип. Да ги одредиме параметрите кои што задаваат големина на проблемот и потпроблемите.
- Вредноста на оптималното решение да ја зададеме рекурентно, т.е. користејќи вредности на оптималното решение на некои помали проблеми.
- Да најдеме ефикасен начин за на основа на вредностите на оптималните решенија на потпроблемите да добиеме вредност на оптималното решение на проблемот (типично користејќи еден циклус, понекогаш само неколку наредби).
- Да ги најдеме и табеларно вредностите на решенијата за наједноставните потпроблеми, а потоа да ги комбинираме вредностите на решенијата на помалите потпроблеми, користејќи го пронајдениот начин наоѓаме вредности на решенијата на поголемите потпроблеми, се до решавањето на самиот проблем. При тоа вредностите ги табелираме вредностите на решенија и делумните информации за начинот на добивање на решенија за сите потпроблеми.
- На основ на табеларните информации го конструираме бараното оптимално решение.