

Föreläsning 5: Dynamisk programmering

Vi betraktar en typ av problem vi tidigare sett:

Indata: En uppsättning intervall $[s_i, f_i]$ med vikt w_i .

Mål: Att hitta en uppsättning icke överlappande intervall med maximal viktsumma.

Tänk så här: Låt O vara ett optimalt val. Vi tänker baklänges ut hur det måste se ut. Antag att intervallen är sorterade efter växande sluttid. Finns intervall n med eller inte?

1. Intervall n finns inte med i O . Då är O ett optimalt val av intervallen $1, \dots, n-1$.

2. Intervallet n finns med. Låt k vara det största index så att intervall k går att förena med intervall n . Då måste O innehålla ett optimalt urval O' av intervallen $1, \dots, k$.

Def: Låt $M[i]$ = vikten av det bästa urvalet bland intervallen $1, \dots, i$.

Rekursionsformel:

$$M[k] = \begin{cases} w_1 & k = 1 \\ \max(M[k-1], M[j] + w_k) & \text{annars} \end{cases}$$

I rekursionen är j maximalt så att intervall j går att förena med intervall k .

Vi vill beräkna $M[n]$. Värdet kan beräknas i tid $O(n)$.

Principen bakom DynP är att bara lösa samma problem en gång. Genom att spara redan uträknade resultat i en tabell och slå upp i den för varje delproblem som ska lösas kan detta åstadkommas.

Flödesschema:

- (1) Bestäm utseendet hos en optimal lösning.
- (2) Sätt upp en rekursionsekvation för optimalvärdet.
- (3) Tabulera optimalvärdena för olika delproblem (rekursivt eller från små problem till större).
- (4) Konstruera optimallösningen.

Vid konstruktion av DynP-algoritmer fokuserar man vanligen på *värdet* av lösningen, inte så mycket på lösningens utseende.

Exempel på DynP

Problem: Givet talföljd $x_1, x_2, \dots, x_n$ vill vi beräkna längden av längsta konsekutiva delföljden av växande tal.

Låt $v(i)$ = längden av längsta sekvens som slutar i x_i .

Algoritm:

```

(1)       $v(1) \leftarrow 1$ 
(2)      for  $i = 2$  to  $n$ 
(3)          if  $x(i-1) \leq x(i)$ 
(4)               $v(i) \leftarrow v(i-1) + 1$ 
(5)          else
(6)               $v(i) \leftarrow 1$ 
(7)      return  $v$ 

```

Sedan beräknas $\max_i v(i)$.

Problem: Samma problem men utan att följderna behöver vara konsekutiva.

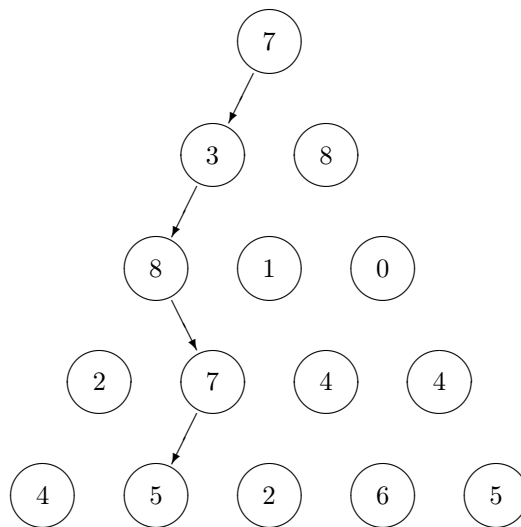
Metod: Definiera $v(i)$ som ovan. Vi har nu rekursionssambandet

$$v[1] = 1c[i, j] = \begin{cases} 1 & x[i] \leq x[j] \\ 0 & \text{annars.} \end{cases}$$

$$v[i] = \max_{1 \leq k \leq i} \{v[k]c[k, i] + 1\}$$

Beräkna först alla $v[i]$. Bestäm sedan $\max_i v[i]$.

Problem: Hitta stigen från toppen till botten som maximerar summan av de ingående talen.



Låt a_{ij} vara elementet på rad i , kolumn j .

Låt $V[i, j]$ vara värdet på bästa stigen från (i, j) ner till rad n . Då gäller

$$V[i, j] = \begin{cases} a_{ij} & i = n, \\ a_{ij} + \max\{V[i+1, j], V[i+1, j+1]\} & \text{annars.} \end{cases}$$

Beräkning av alla $V[i, j]$:

```

(1)          for  $j = 1$  to  $n$ 
(2)             $V[n, j] \leftarrow a_{nj}$ 
(3)           $i = n - 11$  for  $j = 1$  to  $i$ 
(4)             $V[i, j] \leftarrow a_{ij} +$ 
                $\max\{V[i + 1, j], V[i + 1, j + 1]\}$ 

```

Körtiden blir $\Theta(n^2)$ för att hitta $V[1, 1]$, värdet av den bästa stigen. Minnesåtgången är också $\Theta(n^2)$.

När dynamisk programmering fungerar

Dynamisk programmering fungerar vanligen när

1. Problemet kan delas upp i delproblem.
2. Problemet kan lösas genom ett girigt val som leder till en delproblemsuppdelning.
3. Lösningarna på delproblemen på ett naturligt sätt kan lagras i en array.
4. Värdena på arrayen kan beräknas relativt enkelt genom en rekursionsekvation.

Problem: DELSUMMA

Input: En följd av positiva heltal $a_1, a_2, \dots, a_n$ och ett heltal M .

Problem: Avgör om det finns en delföljd av talen vars summa är exakt M .

Går att lösa med dynamisk programmering. Sätt

$$V[i, j] = \begin{cases} 1 & \text{om det går att få summa } j \text{ med de } i \text{ första talen,} \\ 0 & \text{annars.} \end{cases}$$

Initiering:

$$\begin{cases} v[1, a_1] = 1 \\ v[1, 0] = 1 \\ v[1, j] = 0 & \text{annars} \end{cases}$$

Rekursion:

$$V[i, j] = \begin{cases} 1 & v[i-1, j] = 1 \text{ eller } v[i-1, j-a_i] = 1 \\ 0 & \text{annars.} \end{cases}$$

Algoritmen går i $O(Mn)$. Eftersom indatas storlek är n och $\log M$ är detta inte en riktig polynomiell algoritm.

Floyd-Warshalls algoritm

I boken presenteras Bellman-Fords algoritm som ett exempel på en DynP-algoritm för beräkning av kortaste avstånd i grafer med negativa kantvikter.

Givet en graf $G = (V, E)$ med vikter w_{ij} på kanterna. Om $(v_i, v_j) \notin E$ är $w_{ij} = \infty$. Bestäm kortaste avstånden mellan varje par av hörn i G !

Låt $d[i, j, k]$ vara kortaste avståndet mellan v_i och v_j inskränkt till stigar där bara hörnen $v_1, \dots, v_k$ förekommer (utöver v_i och v_j).

Då är $d[i, j, 0] = w_{ij}$ om $i \neq j$ och $d[i, i, 0] = 0$. Sedan gäller följande rekursions samband:

$$d[i, j, k] = \min\{d[i, j, k-1], d[i, k, k-1] + d[k, j, k-1]\}$$

för $k \geq 1$.

De kortaste avstånden vi är ute efter svarar mot $d[i, j, n]$ där n är antalet noder. Följande algoritm beräknar alla d -värden:

```

FLOYD-WARSHALL( $G = \langle V, E \rangle, W$ )
(1)       $n \leftarrow \dim(W)$ 
(2)      for  $i \leftarrow 1$  to  $n$ 
(3)          for  $j \leftarrow 1$  to  $n$ 
(4)               $d[i, j, 0] \leftarrow w_{ij}$ 
(5)      for  $k \leftarrow 1$  to  $n$ 
(6)          for  $i \leftarrow 1$  to  $n$ 
(7)              for  $j \leftarrow 1$  to  $n$ 
(8)                   $d[i, j, k] \leftarrow \min(d[i, j, k-1], d[i, k, k-1] + d[k, j, k-1])$ 
(9)      return  $d$ 

```

Algoritmen arbetar i tid $O(|V|^3)$. Det kan verka som om algoritmen på samma tid som Bellman-Fords algoritm klarar av att beräkna **alla** avstånd. Men lägg märke till att BF arbetar i tid $O(|V||E|)$ vilket för glesa grafer är mindre. Negativa cykler kan hittas genom att man testat om $d[i, i, k] < 0$ för några i, k .

Multiplikation av flera matriser

Problem: Om vi har en följd av matriser av olika dimension som skall multipliceras ihop kan vi göra det på olika sätt beroende på vilka matriser vi multiplicerar först.

Exempelvis: $A_1 A_2 A_3 A_4 A_5$ kan multipliceras $(A_1 A_2)((A_3 A_4) A_5)$ eller $A_1((A_2 A_3)(A_4 A_5))$.

Antalet operationer påverkas faktiskt av vilken metod man använder.

Följande algoritm beräknar det minimala antalet talmultiplikationer som krävs. Algoritmen har komplexiteten $O(n^3)$ och bygger på dynamisk programmering. $m[i, j]$ kommer att innehålla det minimala antalet multiplikationer som behövs för att räkna ut matrisprodukten $M_i M_{i+1} \dots M_j$.

Rekursionsekvationen för $m[i, j]$ är:

$m[i, j] = \min_{i \leq k < j} (m[i, k] + m[k+1, j] + r[i-1]r[k]r[j])$ om $i < j$. och $m[i, j] = 0$ om $i = j$.

```

1. procedure OptimalMatmul (m, r, n);
2.   var i, j, k, t, len: integer;
3.   begin
4.     for i:= 1 to n do m[i,i]:= 0;
5.     for len:= 1 to n-1 do
6.       for i:= 1 to n-len do begin
7.         j:= i+len;
8.         m[i,j]:= m[i+1,j]+r[i-1]*r[i]*r[j];
9.         for k:= i+1 to j-1 do begin
10.          t:= m[i,k]+m[k+1,j]+r[i-1]*r[k]*r[j];
11.          if t < m[i,j] then begin
12.            m[i,j]:= t;
13.          end;
14.        end;

```

```

15.         end;
16. end /* OptimalMatmul */;

```

Om vi har n stycken matriser $M_1 M_2 \cdots M_n$ där matris M_i har dimension $r[i - 1] \times r[i]$ så beräknas det minimala antalet multiplikationer av anropet `OptimalMatmul(m, r, n)`. Svaret finns då i $m[1, n]$.

Man vill egentligen inte bara veta det minimala antalet multiplikationer utan också i vilken ordning man ska multiplicera ihop matriserna för att uppnå det minimala antalet. Vi kan bygga ut `OptimalMatmul` så att man kan få reda på ordningen. Lägg till en hjälpmatris $bryt[i, j]$ som innehåller var gränsen går mellan dom två matriserna som bildar produkten $M_i M_{i+1} \cdots M_j$.

Exempel: $bryt[7, 9] = 7$ betyder att man ska multiplicera $M_7(M_8 M_9)$ och $bryt[7, 9] = 8$ betyder att man ska multiplicera $(M_7 M_8)M_9$.

Vi skriver sedan en procedur som skriver ut hur multiplikationen ska ske, dvs tolkar värdena av $bryt$ och skriver ut en parentetisering. Utskriften ska inte innehålla matrisindexen (som är onödiga) utan bara symbolisera varje matris med ett M .

Exempel: $((MM)M)(MM)$.

Stoppa in satsen `bryt[i, j] := i`; efter rad 8 och satsen `bryt[i, j] := k`; efter rad 12 i proceduren ovan.

Utskrift av parentetiseringen görs sedan med anropet `Skriv(1, n, bryt)`.

```

procedure Skriv(a, b: integer; bryt: matris);
begin
  if a = b then Write('M')
  else begin
    Write('(');
    Skriv(a, bryt[a, b], bryt);
    Skriv(bryt[a, b] + 1, b, bryt);
    Write(')');
  end
end;

```