

DD2354, Algoritmer och komplexitet, 2007

Uppgifter till övning 4

Dynamisk programmering

Talföljder Givet är två följder av positiva heltal $a_1, a_2, \dots, a_n$ och $b_1, b_2, \dots, b_n$ där alla tal är mindre än n^2 samt ett positivt heltal B där $B \leq n^3$. Problemet är att avgöra om det finns någon talföljd $c_1, c_2, \dots, c_n$ så att $\sum_{i=1}^n c_i = B$ och $c_i = a_i$ eller $c_i = b_i$ för $1 \leq i \leq n$.

Beskriv och analysera en algoritm som löser detta problem med hjälp av dynamisk programmering.

Tala också om hur man ska utvidga algoritmen så den också talar om hur lösningen ser ut, alltså om $c_i = a_i$ eller $c_i = b_i$ för $1 \leq i \leq n$.

Matriskedjemultiplikation Följande algoritm beräknar det minimala antalet talmultiplikationer som krävs för att multiplicera ihop ett antal matriser av varierande storlek med varandra. 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:

$$m[i, j] = \begin{cases} 0 & \text{om } i = j \\ \min_{i \leq k < j} (m[i, k] + m[k+1, j] + r[i-1]r[k]r[j]) & \text{om } i < j \end{cases}$$

```
1. procedure OptimalMatmul(var m: matris; r: vektor; n: integer);
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 $M_1 M_2 \dots M_n$ 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. Bygg 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 $M_i M_{i+1} \dots M_j$.

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 $(M_7 M_8)M_9$.

Skriv 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 .

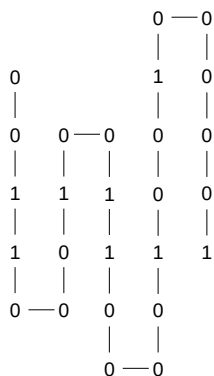
Exempel: (((MM)M)(MM)).

Proteinvikning Ett protein är en lång kedja av aminosyror. Proteinkedjan är inte rak som en pinne utan hopvikt på ett intrikat sätt som minimerar den potentiella energin. Man vill väldigt gärna kunna räkna ut hur ett protein kommer att vika sig. I denna uppgift ska vi därför studera en enkel modell av proteinvikning där aminosyrorna är antingen *hydrofoba* eller *hydrofila*. Hydrofoba aminosyror tenderar att klumpa ihop sig.

För enkelhets skull ser vi proteinet som en binär sträng där ettor motsvarar hydrofoba aminosyror och nollor hydrofila aminosyror. Strängen (proteinet) ska sedan vikas i ett tvådimensionellt kvadratisk gitter. Målet är att få dom hydrofoba aminosyrorna att klumpa ihop sig, det vill säga att få så många ettor som möjligt att ligga nära varandra. Vi har alltså ett optimeringsproblem där målfunktionen är antalet par av ettor som ligger intill varandra i gittret (lodrätt eller vågrätt) utan att vara intill varandra i strängen.

Du ska konstruera en algoritm som med hjälp av dynamisk programmering konstruerar en optimal *dragspelsvikning* av en given proteinsträng av längd n . En dragspelsvikning är en vikning där strängen först går en sträcka rakt nedåt, sedan en sträcka rakt uppåt, sedan en sträcka rakt nedåt, och så vidare. I en sådan vikning kan man notera att lodräta par av intilliggande ettor alltid kommer i följd i strängen, så det är bara vågräta par av ettor som bidrar till målfunktionen.

I följande figur är strängen 00110001001100001001000001 dragspelsvikt på ett sådant sätt att målfunktionen blir 4.



Not: Proteinvikningsproblemet är ett viktigt algoritmiskt problem som studeras i bioinformatiken. Det behandlas tillsammans med många andra problem med biologisk anknytning i den valfria kursen *Algoritmisk bioinformatik* som går i period 4 varje år.

Analysator för kontextfri grammatik En *kontextfri grammatik* brukar användas för att beskriva syntax för bland annat programspråk. En kontextfri grammatik i *Chomskynormalform* beskrivs av

- en mängd slutsymboler T (som brukar skrivas med små bokstäver),
- en mängd ickeslutsymboler N (som brukar skrivas med stora bokstäver),
- startsymbolen S (en av ickeslutsymbolerna i mängden N),
- en mängd omskrivningsregler som antingen är på formen $A \rightarrow BC$ eller $A \rightarrow a$, där $A, B, C \in N$ och $a \in T$.

Om $A \in N$ så definieras $\mathcal{L}(A)$ genom

$$\mathcal{L}(A) = \{bc : b \in \mathcal{L}(B) \text{ och } c \in \mathcal{L}(C) \text{ där } A \rightarrow BC\} \cup \{a : A \rightarrow a\}.$$

Språket som genereras av grammatiken definieras nu som $\mathcal{L}(S)$, vilket alltså är alla strängar av slutsymboler som kan bildas med omskrivningskedjor som börjar med startsymbolen S .

Exempel: Betrakta grammatiken med $T = \{a, b\}$, $N = \{S, A, B, R\}$, startsymbolen S och reglerna $S \rightarrow AR$, $S \rightarrow AB$, $A \rightarrow a$, $B \rightarrow b$, $R \rightarrow SB$. Vi kan se att strängen $aabb$ tillhör språket som genereras av grammatiken med hjälp av följande kedja av omskrivningar:

$$S \rightarrow AR \rightarrow aR \rightarrow aSB \rightarrow aSb \rightarrow aABb \rightarrow aaBb \rightarrow aabb.$$

I själva verket kan man visa att det språk som genereras av grammatiken är precis alla strängar som består av k stycken a följt av k stycken b där k är ett positivt heltal.

Din uppgift är att *konstruera* och *analysera* en effektiv algoritm som avgör ifall en sträng tillhör det språk som genereras av en grammatik. Indata är alltså en kontextfri grammatik på Chomskynormalform samt en sträng av slutsymboler. Utdata är sant eller falskt beroende på om strängen kunde genereras av grammatiken eller inte. Ange tidskomplexiteten för din algoritm uttryckt i antalet regler m i grammatiken och längden n av strängen.

Mer om grammatiker kan man läsa i kursen *Artificiella språk och syntaxanalys*.

Lösningar

Lösning

Talföljder Vi skapar en boolesk $n \times B$ -matris M som från början är fylld med nollor. En etta i $M[k, s]$ ska betyda att det finns något val av $c_1, \dots, c_k$ så att $\sum_{i=1}^k c_i = s$. Rekursionsekvationen för $M[k, s]$ blir:

$$M[k, s] = \begin{cases} 1 & \text{om } k = 1 \text{ och } (s = a_1 \text{ eller } s = b_1), \\ 1 & \text{om } k > 1 \text{ och } (M[k-1, s-a_k] = 1 \text{ eller } M[k-1, s-b_k] = 1), \\ 0 & \text{annars.} \end{cases}$$

Beräkningen börjar med att $M[1, a_1]$ och $M[1, b_1]$ sätts till 1. Därefter sätter man ettor i $M[2, a_1 + a_2]$, $M[2, a_1 + b_2]$, $M[2, b_1 + a_2]$ och $M[2, b_1 + b_2]$ i den andra raden av matrisen, sedan sätter man ettor i den tredje raden och så vidare. Om det hamnar en etta i $M[n, B]$ så är svaret på problemet ja.

Algoritmen kan se ut så här i C:

```

int ExistsC(int n, int a[], int b[], int B)
{ char M[n + 1][B + 1]; /* Dynamiska matriser är ett tillägg i gcc */
  int i, j;
  for (i = 1; i <= n; i++)
    for (j = 1; j <= B; j++) M[i][j] = 0;
  M[1][a[1]] = M[1][b[1]] = 1;
  for (i = 2; i <= n; i++)
    for (j = 1; j <= B; j++) {
      if (j - a[i] > 0 && M[i - 1][j - a[i]]) M[i][j] = 1;
      if (j - b[i] > 0 && M[i - 1][j - b[i]]) M[i][j] = 1;
    }
  return M[n][B];
}

```

Satserna inuti den dubbla forslingan utförs $(n - 1)B$ gånger och i dom satserna utförs högst 4 jämförelser och 2 tilldelningar, dvs ett konstant antal. Hela algoritmen går alltså i $O(n^2) + O((n - 1)B) = O(n^2 + nB) \subseteq O(n^4)$.

Genom att gå bakifrån (från den n -te raden och uppåt) och undersöka i vilka positioner det finns ettor kan man lista ut en lösning. Ettan i $M[n, B]$ måste ha satts dit på grund av att det finns en etta i antingen $M[n - 1, B - a_n]$ eller $M[n - 1, B - b_n]$. Kom ihåg vilken av dessa det är. Om det står en etta i båda positionerna är det bara att välja en av dom (eftersom bara en enda lösning efterfrågas). Fortsätt på samma sätt från den valda positionen ända tills rad 1 har nåtts. Implementationen i C ser ut på följande sätt.

```

void WriteC(int n, int a[], int b[], int B)
{ int c[n + 1], pos;

  /* samma satser som i proceduren ExistsC ovan ska in här*/

  pos = B;
  for (i = n; i > 1; i++) {
    if (pos - a[i] > 0 && M[i - 1][pos - a[i]]) c[i] = a[i];
    else c[i] = b[i];
    pos -= c[i];
  }
  c[1] = pos;
  printf("Lösningen är %d", c[1]);
  for (i = 2; i <= n; i++) printf(" + %d", c[i]);
}

```

□

Lösning

Matriskedjemultiplikation 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;

```

□

Lösning

Proteinvikning Låt $q_{a,b}$ vara det maximala värdet på målfunktionen man kan få för en vikning av delen $p[a..n]$ av proteinet, där den första sträckan i vikningen har ändpunkterna a och b . Vi kan uttrycka $q_{a,b}$ rekursivt på följande sätt:

$$q_{a,b} = \max_{b+1 < c \leq n} (\text{profit}(a, b, c) + q_{b+1, c}).$$

Basfallen är $q_{a,n} = 0$ för $1 \leq a < n$. Svaret hittar vi sedan som $\max_{1 \leq b \leq n} q_{1,b}$.

Nu gäller det bara att beräkna $q_{a,b}$ enligt dessa formler i rätt ordning:

```
for a←1 to n-1 do q[a,n]←0;
for b←n-1 downto 2 do
  for a←1 to b-1 do
    t←-1;
    for c←b+2 to n do
      v←profit(a,b,c)+q[b+1,c];
      if v>t then t←v;
    q[a,b]←t;
  max←0;
for b←2 to n do
  if q[1,b]>max then max←q[1,b];
return max;
```

Eftersom vi som mest har tre nästlade for-slingor och ett anrop till `profit` tar tid $O(n)$ blir tidskomplexiteten uppenbarligen $O(n^4)$. □

Lösning

Analysator för kontextfri grammatik Vi använder dynamisk programmering ungefär som i problemet där man letar efter optimal matriskedjemultiplikationsordning. Här ska vi istället bestämma i vilken ordning och på vilken delsträng reglerna ska tillämpas.

Indata är en uppsättning regler R och en vektor $w[1..n]$ som alltså indexeras från 1 till n . Låt oss bygga upp en matris $M[1..n, 1..n]$ där elementet $M[i, j]$ anger dom ickeslutsymboler från vilka man med hjälp av kedjor av omskrivningar kan härleda delsträngen $w[i..j]$.

Rekursiv definition av $M[i, j]$:

$$M[i, j] = \begin{cases} \{X : (X \rightarrow w[i]) \in R\} & \text{om } i = j \\ \{X : (X \rightarrow AB) \in R \wedge \exists k : A \in M[i, k-1] \wedge B \in M[k, j]\} & \text{om } i < j \end{cases}$$

Eftersom varje position i matrisen är en mängd av ickeslutsymboler så måste vi välja en lämplig datastruktur också för detta. Låt oss representera en mängd av ickeslutsymboler som en bitvektor som indexeras med ickeslutsymboler. 1 betyder att symbolen är med i mängden och 0 att den inte är med i mängden. Exempel: Om $M[i, j][B] = 1$ så är ickeslutsymbolen B med i mängden $M[i, j]$, vilket betyder att det finns en kedja av omskrivningsregler från B till delsträngen $w[i..j]$.

Algoritm som beräknar matrisen $M[i, j]$ och returnerar sant ifall strängen tillhör språket som genereras av grammatiken:

```
for i←1 to n do
  M[i,i]←0; /* alla bitar nollställs */
  för varje regel  $X \rightarrow w[i]$  do
    M[i,i][X]←1;
for len←2 to n do
  for i←1 to n-len+1 do
    j←i+len-1;
```

```

M[i,j] ← 0;
for k ← i+1 to j do
  för varje regel  $X \rightarrow AB$  do
    if M[i,k-1][A]=1 and M[k,j][B]=1 then
      M[i,j][X] ← 1;
return M[1,n][S] = 1;

```

Tid: $O(n^3m)$. Minne: $O(n^2m)$ (eftersom m är en övre gräns för antalet ickeslutsymboler). □
