Viggo Kann 2021-08-26

DD2350 Algoritmer, datastrukturer och komplexitet

Uppgifter till övning 4

Dynamisk programmering

På denna övning är det också **inlämning av skriftliga lösningar av teoriuppgifterna till labb 2** och muntlig redovisning av teoriuppgifterna.

Träskvandring Tina ska gå genom ett träsk som representeras av ett $n \times n$-rutmönster från vänsterkanten till högerkanten. I varje steg kan hon gå ett steg rakt till höger, snett uppåt höger eller snett nedåt höger. Det är olika jobbigt att gå på olika ställen i träsket. Att hamna på ruta (i, j) kostar arbetet $A[i, j]$, som är ett positivt heltal.

Beskriv en algoritm som hittar det minimala arbetet att ta sig igenom träsket i följande olika fall.

a) Anta att Tina får starta var som helst på vänsterkanten och får sluta var som helst på högerkanten.

b) Anta att Tina alltid startar i ruta $(n/2, 1)$.

c) Anta att Tina dessutom måste sluta i ruta $(n/2, n)$.

d) Skriv ut den minst jobbiga väg Tina kan ta genom träsket i c).

Kombinera mynt och sedlar Givet en (konstant) uppsättning mynt- och sedelslag med valörerna $v_1, \ldots, v_k$ kronor och ett maximalt belopp n. (Alla tal i indata är positiva heltal.) Formulera en rekursion för på hur många sätt man kan bilda varje summa mellan 0 och n kronor med hjälp av denna uppsättning mynt och sedlar.

Längsta gemensamma delsträng Strängarna `ALGORITM` och `PLÅGORIS` har den gemensamma delsträngen `GORI`. Den *längsta gemensamma delsträngen* hos dessa strängar har alltså längd 4. I en delsträng måste tecknen ligga i en sammanhängande följd.

Konstruera en effektiv algoritm som givet två strängar $a_1a_2 \ldots a_m$ och $b_1b_2 \ldots b_n$ beräknar och returnerar längden hos den längsta gemensamma delsträngen. Algoritmen ska bygga på dynamisk programmering och gå i tid $O(nm)$.

Här följer två roliga men mer komplicerade exempel som vi nog inte hinner med på övningen:

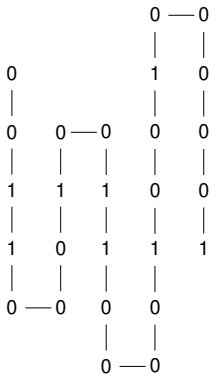
Proteinvikning Ett protein är en lång kedja av aminosyror. Proteinkedjan är inte rak som en pinne utan hopvikt på ett intrikat sätt som minimerar den potentiella energin. Man vill väldigt gärna kunna räkna ut hur ett protein kommer att vika sig. I denna uppgift ska vi därför studera en enkel modell av 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 kvadratiskt 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.



Definition av problemet PROTEINDRAGSPELSVIKNING:

> INMATNING: En binär sträng med n tecken.
>
> PROBLEM: Hitta den dragspelsvikning av indatasträngen som ger det största värdet på målfunktionen, alltså det största antalet par av ettor som ligger bredvid varandra men inte direkt efter varandra i strängen.

Konstruera och analysera tidskomplexiteten för en algoritm som löser proteindragspelsvikningsproblemet med dynamisk programmering.

Du får gärna använda dig av nedanstående algoritm, som beräknar antalet par ettor i ett varv (dvs mellan två sträckor) i en dragspelsvikning som ligger bredvid varandra (men inte direkt efter varandra i strängen). Anta att proteinet lagras i en array `p[1..n]`. Parametrarna `a` och `b` anger index i arrayen för den första sträckans ändpunkter. Parametern `c` anger index för den andra sträckans slutpunkt. Se figuren nedanför till höger.

```
profit(a,b,c) =
   shortest←min(b-a,c-(b+1));
   s←0;
   for i←1 to shortest do
      if p[b-i]=1 and p[b+1+i]=1 then
         s←s+1;
   return s;
```

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 $A \to BC$ eller $A \to a$, där $A, B, C \in N$ och $a \in T$.

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

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

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 $S \to AR$, $S \to AB$, $A \to a$, $B \to b$, $R \to SB$. Vi kan se att strängen *aabb* tillhör språket som genereras av grammatiken med hjälp av följande kedja av omskrivningar:

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

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 *Automater och språk*.

Lösningar

Lösning till Träskvandring
a) Vi låter $W[i, j]$ vara det minimala arbetet som krävs för att komma till ruta (i, j). Tina får starta var som helst på vänsterkanten. Basfallen blir att hamna på en första ruta (från någonstans utanför träsket). Det kostar endast $A[i, 1]$ för varje i. Vi kommer att vilja uttrycka hur jobbigt det är att komma till ruta i, j från ruta $i, j-1$, vilket är arbetet $A[i, j]$ plus det det kostade att komma till ruta $i, j-1$. Vi vill hitta det *minst jobbiga* sättet att komma till varje ruta i, j, så i allmänhet finns det tre ställen Tina kan ha kommit ifrån, alla i kolumnen $j-1$, och vi måste jämföra hur jobbigt det är att komma från vart och ett och välja det minst jobbiga. Nu verkar det rimligt att börja beräkna kolumnvis hur jobbigt det är att komma till varje ruta. Att komma till en ruta i ovankanten uppifrån går inte, så det värdet sätts till oändligheten för att vår jämförelse inte ska premiera den riktningen. På samma sätt hanterar vi underkanten av matrisen. Därför kommer vi att ha följande rekursion: $W[i, j] = \min(W[i-1, j-1], W[i, j-1], W[i+1, j-1]) + A[i, j]$. Vi beräknar alla värdena i den nya matrisen W.

Basfallen:
for $j \leftarrow 1$ **to** n
  $W[j, 1] = A[j, 1]$

Fyll i matrisen:
for $j \leftarrow 2$ **to** n
  for $i \leftarrow 1$ **to** n
    $fromabove = W[i-1, j-1]$ **if** $i > 1$, **else** ∞
    $fromsame = W[i, j-1]$
    $frombelow = W[i+1, j-1]$ **if** $i < n$, **else** ∞
    $W[i, j] = \min(fromabove, fromsame, frombelow) + A[i, j]$

Tina fick lämna träsket från vilken ruta som helst på högerkanten. Därför kan vi gå igenom sista kolumnen i vår matris och leta efter det minsta värdet där.

Hitta svaret:
opt $\leftarrow \infty$
for $i \leftarrow 1$ **to** n
  if $W[i, n] < opt$ **then**
    $opt \leftarrow W[i, n]$
return *opt*

Om vi sätter ihop dessa tre steg får vi en dynamisk programmering-algoritm som beräknar det minst jobbiga sättet att ta sig från vänster sida av träsket till höger sida.

b) Om Tina alltid startar i ruta $(n/2, 1)$ har vi lite annorlunda basfall. Eftersom Tina aldrig får förflytta sig rakt uppåt eller rakt nedåt, är alla andra rutor i den vänstra kolumnen omöjliga att nå, dvs arbetet för att ta sig dit är oändligt. Rekursionen blir likadan.

c) Om Tina dessutom alltid slutar i rutan $(n/2, n)$, kommer vi när vi är ute efter svaret bara att vara intresserade av $W[n/2, n]$, oavsett vad som står i de andra rutorna i sista kolumnen. Det enda som ändras är vad vi ska returnera, där sökandet efter *opt* ersätts med att returnera $W[n/2, n]$. Rekursionen fungerar likadant.

d) Nu vill vi konstruera en bästa väg. Vi kan antingen spara varifrån vi kom i varje steg, dvs låta W innehålla både ett jobbighetsvärde och en pekare till rutan man kommit ifrån, eller ta vår slutruta, och subtrahera dess värde från A-matrisen, och se vilken av rutorna som vi kan ha kommit ifrån som hade detta värde, och sedan upprepa processen tills vi kommer till startrutan. (Om det är mer än en väg till en ruta som var lika jobbig, gör det inget. Vi vill bara ha *en* minst jobbig väg genom träsket, och det kan finnas flera som är lika jobbiga.)

Eftersom vi började med värdet i slutrutan, och tog reda på varifrån vi kunde ha kommit för att få det värdet där, och eftersom vi upprepar detta för varje ruta i stigen, kommer vi att nå startrutan. (Annars kan vi inte ha något värde i slutrutan.) I varje ruta står nämligen hur jobbigt det är att komma dit från startrutan, den enklaste vägen, och om vi subtraherar bort det som hörde till just denna ruta, återstår hur jobbigt det var att komma till förra rutan. Det värdet måste stå i minst en av rutorna som vi kunde ha kommit från, och det betyder att vi kunde nå den rutan till det priset. Vi kommer alltid att nå startrutan utan att passera någon onåbar ruta, eftersom $W[i, j] - A[i, j] \neq \infty$ såvida inte $W[i, j] = \infty$. Eftersom vi tittar på stigen baklänges, kan vi spara värdena i en stack för att skriva ut dem i rätt ordning. □

Lösning till Kombinera mynt och sedlar

Låt $N[b, j]$ vara antalet sätt att bilda beloppet b med valörerna $v_1, \ldots, v_j$ kronor.

N kan till exempel definieras rekursivt på följande sätt:

$$N[b,j] = \begin{cases} 1 & \text{om } b = 0 \text{ eller } (j = 1) \wedge (b \mod v_1 = 0), \\ 0 & \text{om } j = 1 \text{ och } b \mod v_1 \neq 0, \\ \sum_{i=0}^{\lfloor b/v_j \rfloor} N[b - i \cdot v_j, j-1] & \text{om } 0 < b \leq n \end{cases}$$

En annan möjlig rekursiv formulering bygger på tanken att en kombination av valörerna $v_1, \ldots, v_j$ antingen innehåller valören v_j eller bara innehåller valörerna $v_1, \ldots, v_{j-1}$:

$$N[b,j] = \begin{cases} 1 & \text{om } b = 0, \\ 0 & \text{om } j = 0 \text{ och } b > 0, \\ N[b, j-1] & \text{om } 0 < b < v_j \text{ och } j > 0 \\ N[b - v_j, j] + N[b, j-1] & \text{om } v_j < b \leq n \text{ och } j > 0 \end{cases}$$

□

Lösning till Längsta gemensamma delsträng
För varje par av tecken ifrån var sin sträng, låt $M[i,j]$ vara antal bokstäver till vänster om (och inklusive) a_i som överensstämmer med lika många bokstäver till vänster (och inklusive) b_j. Längden av den längsta gemensamma strängen är då det största talet i matrisen M.

M kan definieras rekursivt på följande sätt:

$$M[i,j] = \begin{cases} 0 & \text{om } i = 0 \text{ eller } j = 0, \\ M[i-1, j-1] + 1 & \text{om } a_i = b_j, \\ 0 & \text{annars.} \end{cases}$$

Här kan vi se att den rekursion vi kommit fram till är samma som vi tittade på förra övningen. Då kom vi fram till följande algoritm:

```
for j ← 0 to n
    M[0, j] ← 0
for i ← 1 to m
    M[i, 0] ← 0
    for j ← 1 to n
        if a_i = b_j then
            M[i, j] ← M[i − 1, j − 1] + 1
        else M[i, j] ← 0
return M
```

Det vi behöver göra utöver detta, för att svara på frågan i denna uppgift, är att hitta det största talet i M. Vi kan förstås först beräkna M och sedan gå igenom matrisen och jämföra värden, men det går bra att utöka algoritmen så att den jämför värden under beräkningarnas gång. Då slipper vi få hela matrisen som utdata. Följande algoritm beräknar hela M och returnerar det största talet i M.

```
max ← 0
for j ← 0 to n
    M[0, j] ← 0
for i ← 1 to m
    M[i, 0] ← 0
    for j ← 1 to n
        if a_i = b_j then
            M[i, j] ← M[i − 1, j − 1] + 1
            if M[i, j] > max then max ← M[i, j]
        else M[i, j] ← 0
return max
```

Tiden domineras även nu av den nästlade for-slingan och är alltså $\Theta(nm)$. □

Lösning till 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:

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

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

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

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 →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 → 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). □
