Programowanie dynamiczne

Nadaje się do rozwiązywania problemów, które można podzielić na podproblemy, ale różne podproblemy mogą wymagać rozwiązania tych samych podpodproblemów. (Metoda *dziel i zwyciężaj* rozwiązuje wtedy te same podpodproblemy wielokrotnie.)

Projektowanie algorytmu dynamicznego:

1. Scharakteryzowanie struktury optymalnego rozwiązania
2. Rekurencyjne zdefiniowanie kosztu optymalnego rozwiązania
3. Obliczenie rozwiązania metodą wstępującą (*bottom-up*)
4. Konstruowanie optymalnego rozwiązania na podstawie wyników wcześniejszych obliczeń

Mnożenie ciągu macierzy

Dane: ciąg n macierzy A_1,... A_n różnych rozmiarów (takich, że można wykonać mnożenie $A_1 \cdot A_2 \ldots \cdot A_n$, t.j. macierz A_i ma wymiatry $p_{i-1} \times p_i$)

Wynik: nawiasowanie minimalizujące łączny koszt mnożeń

```
Matrix-Multiply(A,B)
if columns[A] ≠ rows[B] then
  error “niezgodne rozmiary”
else
  for i ← 1 to rows[A] do
    for j ← 1 to columns[B] do
      C[i,j] ← 0
      for k ← 1 to columns[A] do
        C[i,j] ← C[i,j] + A[i,k] · B[k,j]
  returnc C
```

Czas: $\Theta(p \cdot q \cdot r)$, gdzie $p \times q$, $q \times r$ – wymiary macierzy

liczba możliwych nawiasowań

$$P(n) = \begin{cases} 1 & \text{dla } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{dla } n \geq 2 \end{cases}$$

$P(n) = C(n-1)$, gdzie $C(i)$ – i-ta liczba Catalana.

$$C(n) = \frac{1}{n+1}\binom{2n}{n} = \Omega(4^n/n^{3/2})$$

Rośnie wykładniczo ze względu na n.

Struktura optymalnego nawiasowania

- $A_1 \ldots A_n$ jest dzielony na $(A_1 \ldots A_k)$ oraz $(A_{k+1} \ldots A_n)$, dla pewnego k, $1 \le k < n$.
- Umieszczenie nawiasów w $A_1 \ldots A_k$ (odp. $A_{k+1} \ldots A_n$) jest optymalnym nawiasowaniem dla $A_1 \ldots A_k$ (odp. $A_{k+1} \ldots A_n$)

Rozwiązanie rekurencyjne:
$m[i,j]$ – optymalny koszt mnożenia podciągu $A_i \ldots A_j$.

$$m[i,j] = \begin{cases} 0 & \text{dla } i = j \\ \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & \text{dla } i < j \end{cases}$$

UWAGA: liczba możliwych podproblemów wynosi: $\binom{n}{2} + n = \Theta(n^2)$ (dokładnie jeden podproblem dla każdej pary i,j, $1 \le i \le j \le n$).

Matrix-Chain-Order

Matrix-Chain-Order(p)

$\triangleright$ $p = \langle p_0, ..., p_n \rangle$ - wymiary

$n \leftarrow$ *length*$[p] - 1$

for $i \leftarrow 1$ to n do

    $m[i,i] \leftarrow 0$

for $l \leftarrow 2$ to n do $\triangleright$ l - dł. podciągu

    for $i \leftarrow 1$ to $n - l + 1$ do

        $j \leftarrow i + l - 1$

        $m[i,j] \leftarrow \infty$

        for $k \leftarrow i$ to $j - 1$ do

            $q \leftarrow m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$

            if $q < m[i,j]$ then

                $m[i,j] \leftarrow q$

                $s[i,j] \leftarrow k$ $\triangleright$ podział $A_i \ldots A_j$

return m i s $\triangleright$ czas $\Theta(n^3)$, pam. $\Theta(n^2)$

Wykonanie optymalnego mnożenia

s – tablica wyznaczona przez `Matrix-Chain-Order`.
Mnożenie podciągu $A_i \ldots A_j$.

```
Matrix-Chain-Multiply(A, s, i, j)
1 if j > i then
2    X ← Matrix-Chain-Multiply(A, s, i, s[i, j])
3    Y ← Matrix-Chain-Multiply(A, s, s[i, j] + 1, j)
4    return Matrix-Multiply(X, Y)
5 else return A_i
```

Wywołanie: `Matrix-Chain-Multiply(`A`,`s`,`1`,`n`)`

Recursive-Matrix-Chain

```
Recursive-Matrix-Chain(p, i, j)
1 if i = j
2   then return 0
3 m[i, j] ← ∞
4 for k ← i to j − 1 do
5   q ← Recursive-Matrix-Chain(p, i, k)
    +Recursive-Matrix-Chain(p, k + 1, j)+p_{i−1}p_k p_j
6   if q < m[i, j]
7     then m[i, j] ← q
8 return m[i, j]
```

Czas: $T(n)$

$$\begin{aligned} T(1) &\geq 1 \\ T(n) &\geq 1 + \sum_{k=1}^{n-1}(T(k) + T(n-k) + 1) \quad \text{dla } n > 1 \end{aligned}$$

(koszt wierszy 1-2 i 6-1 ≥ 1 jednostka czasu)

Oszacowanie $T(n)$

$$
\begin{aligned}
T(n) &\geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \\
T(n) &\geq 2 \sum_{i=1}^{n-1} T(i) + n
\end{aligned}
$$

Indukcja: Podstawa: $T(1) \geq 2^0 = 1$. Zał. ind.: dla $i < n$, $T(i) \geq 2^{i-1}$.

$$
\begin{aligned}
T(n) &\geq 2 \sum_{i=1}^{n-1} 2^{i-1} + n \\
&= 2 \sum_{i=0}^{n-2} 2^i + n \\
&= 2(2^{n-1} - 1) + n \\
&= (2^n - 2) + n \\
&\geq 2^{n-1}
\end{aligned}
$$

$T(n) = \Omega(2^n)$ – czas wykładniczy (wielokrotnie rozwiązywany każdy z $O(n^2)$ podproblemów).

Spamiętywanie

Dla każdego podproblemu zapamiętujemy rozwiązanie i zaznaczamy, że jest już rozwiązany. Przy następnym napotkaniu tego samego podproblemu wykorzystujemy wcześniej wyznaczone rozwiązanie. (Wtedy niektóre z podproblemów rozwiązywanych w programowaniu metodą wstępującą mogą zostać pominięte.)

Przykład dla ciągu macierzy:

Memoized-Matrix-Chain(p)

$n \leftarrow$ *length*$[p] - 1$

for $i \leftarrow 1$ to n do

    for $j \leftarrow i$ to n do

        $m[i, j] \leftarrow \infty$

return Lookup-Chain(p, 1, n)

Lookup-Chain

Lookup-Chain(p, i, j)
1 if $m[i,j] < \infty$
2   then return $m[i,j]$
3 if $i = j$ then
4    $m[i,j] \leftarrow 0$
5 else
5   for $k \leftarrow i$ to $j-1$ do
6       $q \leftarrow$Lookup-Chain(p, i, k)
        $+$Lookup-Chain(p, $k+1$, j)$+p_{i-1}p_kp_j$
7   if $q < m[i,j]$
8   then $m[i,j] \leftarrow q$
9 return $m[i,j]$

Czas: $O(n^3)$ – każde z $\Theta(n^2)$ pól $m[i,j]$ jest ≤ 1 raz wyznaczone w wierszach 4–8, co zajmuje $O(n)$ (nie licząc czasu na wyznaczenie innych pól).

Najdłuższy Wspólny Podciąg

$Z = \langle z_1 \ldots z_k \rangle$ jest *podciągiem* ciągu $X = \langle x_1 \ldots , x_m \rangle$, jeśli istnieje rosnący ciąg indeksów $\langle i_1, \ldots , i_k \rangle$, taki że dla każdego $j = 1 \ldots k$ zachodzi $x_{i_j} = z_j$.

Z jest *wspólnym podciągiem* X i Y jeśli jest podciągiem X oraz jest podciągiem Y.

NWP – najdłuższy wspólny podciąg.

Prefix $X_i = \langle x_1 \ldots x_i \rangle$.

Najdłuższy Wspólny Podciąg

$Z = \langle z_1 \dots z_k \rangle$ jest *podciągiem* ciągu $X = \langle x_1 \dots, x_m \rangle$, jeśli istnieje rosnący ciąg indeksów $\langle i_1, \dots, i_k \rangle$, taki że dla każdego $j = 1 \dots k$ zachodzi $x_{i_j} = z_j$.

Z jest *wspólnym podciągiem* X i Y jeśli jest podciągiem X oraz jest podciągiem Y.

NWP – najdłuższy wspólny podciąg.

Prefix $X_i = \langle x_1 \dots x_i \rangle$.

Tw. (O optymalnej podstrukturze NWP) Niech $X = \langle x_1 \dots x_m \rangle$ i $Y = \langle y_1 \dots y_n \rangle$. Niech $Z = \langle z_1 \dots z_k \rangle$ – NWP X i Y.

1. Jeśli $x_m = y_n$, to $z_k = x_m = y_n$ i Z_{k-1} jest NWP X_{m-1} i Y_{n-1}
2. Jeśli $x_m \neq y_n$ i $z_k \neq x_m$, to Z jest NWP X_{m-1} i Y.
3. Jeśli $x_m \neq y_n$ i $z_k \neq y_n$, to Z jest NWP X i Y_{n-1}.

Dowód Tw. o opt. podstrukturze NWP

D-d

1. Jeśli $z_k \neq x_m$ to możnaby dołączyć $x_m = y_n$ do Z uzyskując dłuższy wspólny podciąg X i Y.
2. $z_k \neq x_m$, czyli Z jest wspólnym podciągiem X_{m-1} i Y. Gdyby istniał dłuższy wspólny podciąg W ciągów X_{m-1} i Y, to W byłby też wspólnym podciągiem X i Y dłuższym niż Z.
3. Analogicznie do poprzedniego przypadku.

□

Zależność rekurencyjna i algorytm

$c[i, j]$ – długość NWP prefiksów X_i i Y_j.

$$c[i,j] = \begin{cases} 0, & \text{jeśli } i = 0 \text{ lub } j = 0 \\ c[i-1, j-1] + 1 & \text{jeśli } i, j > 0 \text{ i } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{jeśli } i, j > 0 \text{ i } x_i \neq y_j \end{cases}$$

```
LCS-Length(X,Y)
  m ← length[X]
  n ← length[Y]
  for i ← 1 to m do
      c[i,0] = 0
  for j ← 1 to n do
      c[0,j] = 0
...
```

LCS-Length (c.d.)

...

for $i \leftarrow 1$ **to** m **do**
    for $j \leftarrow 1$ **to** n **do**
        if $x_i = y_j$ **then**
            $c[i,j] \leftarrow c[i-1,j-1]+1$
            $b[i,j] \leftarrow$ "$\nwarrow$"
        else
        if $c[i-1,j] > c[i,j-1]$ **then**
            $c[i,j] \leftarrow c[i-1,j]$
            $b[i,j] \leftarrow$ "$\uparrow$"
        else
            $c[i,j] \leftarrow c[i,j-1]$
            $b[i,j] \leftarrow$ "$\leftarrow$"
return c i b

Czas: $\Theta(nm)$, pamięć: $\Theta(nm)$.

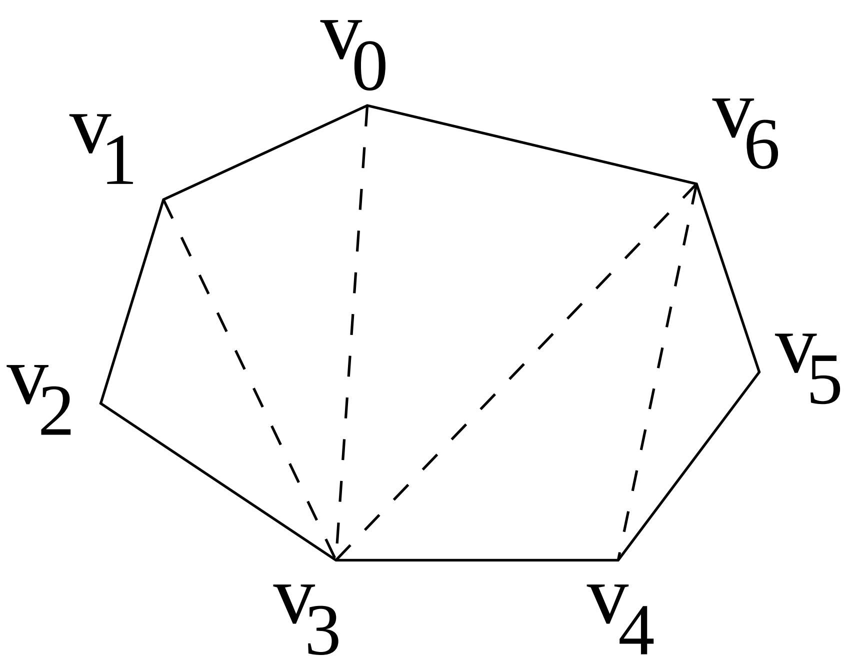
Print-LCS

```
Print-LCS(b, X, i, j)
if i = 0 or j = 0 then
    return
if b[i, j] = "↖" then
    Print-LCS(b, X, i − 1, j − 1)
    wypisz x_i
else
if b[i, j] = "↑" then
    Print-LCS(b, X, i − 1, j)
else Print-LCS(b, X, i, j − 1)
```

Wywołanie: `Print-Lcs(`b`,`X`,`*length*$[X]$`,`*length*$[Y]$`)`

Czas: $O(m+n)$.

triangulacja wielokąta wypukłego

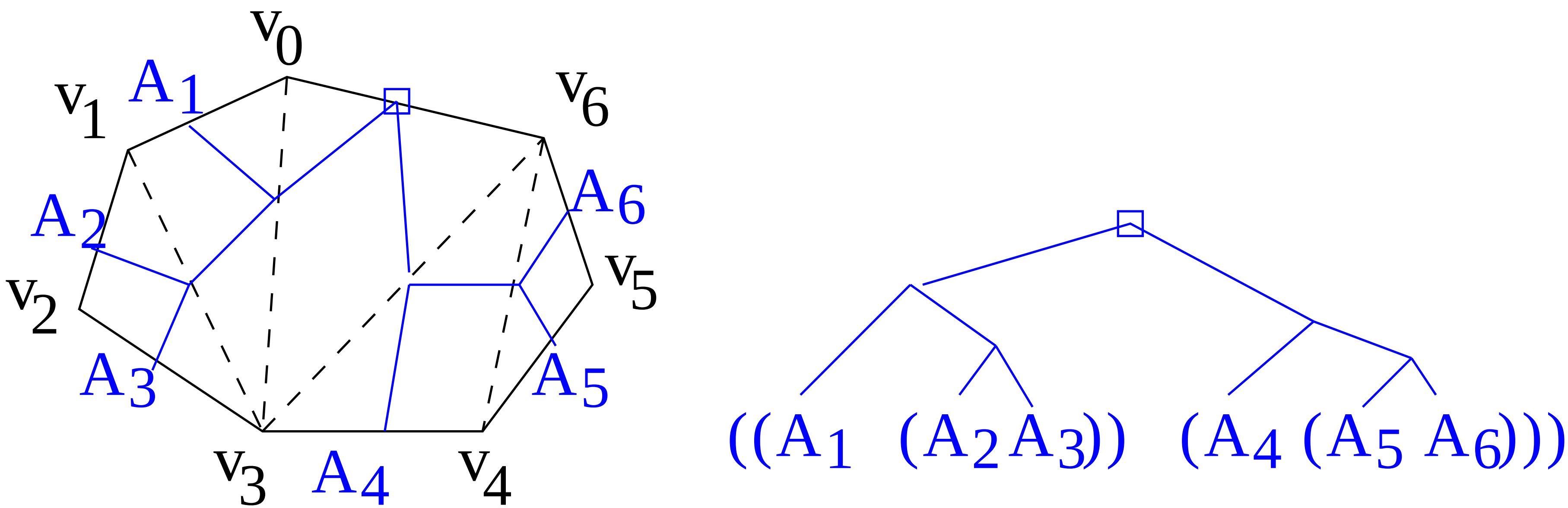


Waga trójkąta: $w(\triangle v_i v_j v_k)$

(n.p. $w(\triangle v_i v_j v_k) = |v_i v_j| + |v_j v_k| + |v_k v_i|$

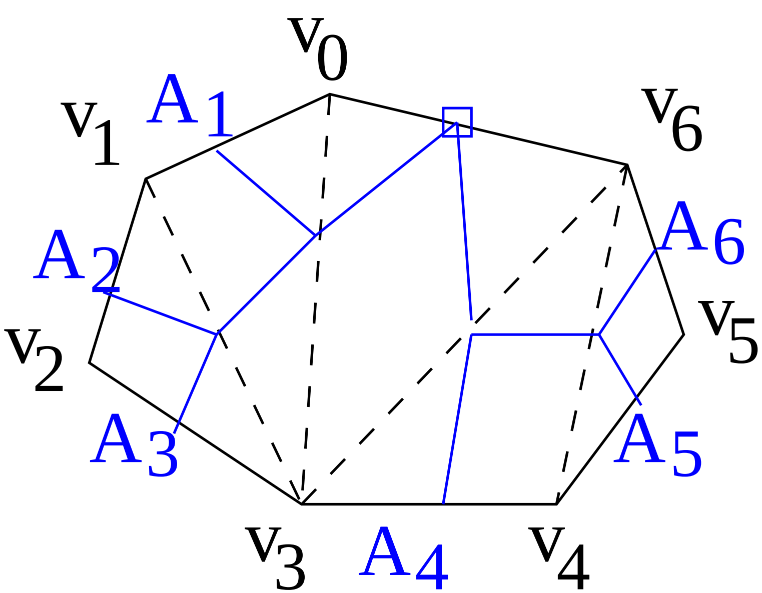
Koszt triangulacji = suma wag jej trójkątów.

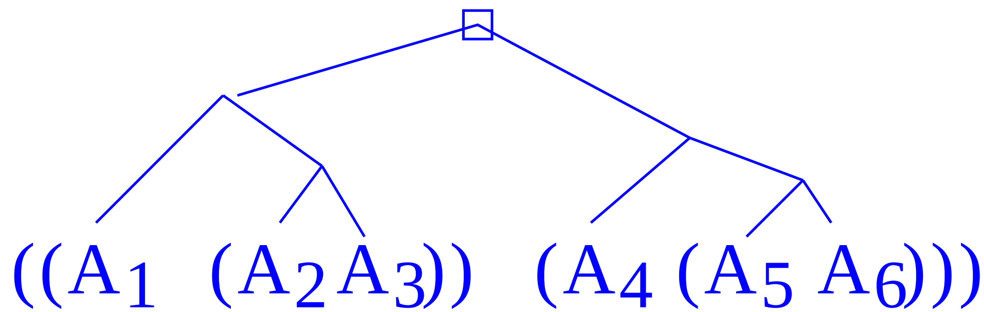
związek z problemem nawiasowania



Jeśli A_i ma wymiary $p_{i-1} \times p_i$, to problem nawiasowania ciągu mnożeń $A_1 \ldots A_n$ redukujemy do problemu triangulacji przyjmując: $w(\triangle v_i v_j v_k) = p_i p_j p_k$.

związek z problemem nawiasowania





Jeśli A_i ma wymiary $p_{i-1} \times p_i$, to problem nawiasowania ciągu mnożeń $A_1 \ldots A_n$ redukujemy do problemu triangulacji przyjmując: $w(\triangle v_i v_j v_k) = p_i p_j p_k$. Algorytm `Matrix-Chain-Order` można przerobić aby rozwiązywał problem triangulacji: zamiast $p = \langle p_0, \ldots, p_n \rangle$ – parametr $v = \langle v_0, \ldots, v_n \rangle$, oraz wiersz 9 zastępujemy:

9 $\quad q \leftarrow m[i, k] + m[k + 1, j] + w(\triangle v_i v_j v_k)$

uzasadnienie

Niech T – *optymalna* triangulacja wielokąta wypukłego o wierzchołkach $\langle v_0 \ldots v_n \rangle$, zawierająca $\triangle v_0 v_k v_n$. Waga T = suma $w(\triangle v_0 v_k v_n)$ oraz wag trangulacji dwu wielokątów $\langle v_0 \ldots v_k \rangle$ i $\langle v_k \ldots v_n \rangle$. Obie muszą być optymalne.

Równanie rekurencyjne:

Niech $t[i,j]$ koszt optymalnej triangulacji $\langle v_{i-1} \ldots v_j \rangle$.

$$t[i,j] = \begin{cases} 0 & \text{dla } i = j \\ \min_{i \le k \le j-1} \{t[i,k] + t[k+1,j] + \\ \qquad + w(\triangle v_{i-1} v_k v_j)\} & \text{dla } i < j \end{cases}$$

Poza funkcją wagi – wzór identyczny ze wzorem na $m[i,j]$.

Wniosek. Optymalną triangulację można wyznaczyć w czasie $\Theta(n^3)$ i pamięci $\Theta(n^2)$.