Dynamic Programming

Bin Wang

School of Software
Tsinghua University

March 29, 2022

Outline

Overview

Dynamic Programming vs Divide & Conquer

1. Similarities
   - partition the problem into subproblems
   - combining the solutions from subproblems
2. Differences
   - overlapping subproblems vs no overlapping subproblems
   - Dynamic programming is typically applied to optimization problems.

Overview

Dynamic Programming vs Divide & Conquer

1. Similarities
   - partition the problem into subproblems
   - combining the solutions from subproblems
2. Differences
   - overlapping subproblems vs no overlapping subproblems
   - Dynamic programming is typically applied to **optimization problems**.

Overview

Steps of Dynamic Programming

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Overview

Steps of Dynamic Programming

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Overview

Steps of Dynamic Programming

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Overview

Steps of Dynamic Programming

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Where to cut steel rods?

Rod-cutting problem

Given a rod of length n inches and a table of prices p_i for $i = 1, 2, \ldots, n$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

Example

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Where to cut steel rods?

If an optimal solution cuts the rod into k pieces, for some $1 \leq k \leq n$, then an optimal decompositon

$$n = i_1 + i_2 + \cdots + i_k$$

of the rod into pieces of lengths $i_1, i_2, \ldots, i_k$ provides maximum corresponding revenue

$$r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$$

Where to cut steel rods?

Sample inspection

$$
\begin{aligned}
r_1 &= 1 & \text{from solution } 1 &= 1 \text{ (no cuts)}\\
r_2 &= 5 & \text{from solution } 2 &= 2 \text{ (no cuts)}\\
r_3 &= 8 & \text{from solution } 3 &= 3 \text{ (no cuts)}\\
r_4 &= 10 & \text{from solution } 4 &= 2+2\\
r_5 &= 13 & \text{from solution } 5 &= 2+3
\end{aligned}
$$

Generally

$$r_n = max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \cdots, r_{n-1} + r_1)$$

Where to cut steel rods?

Sample inspection

$$\begin{aligned} r_1 &= 1 && \text{from solution 1} &&= 1 \text{ (no cuts)} \\ r_2 &= 5 && \text{from solution 2} &&= 2 \text{ (no cuts)} \\ r_3 &= 8 && \text{from solution 3} &&= 3 \text{ (no cuts)} \\ r_4 &= 10 && \text{from solution 4} &&= 2+2 \\ r_5 &= 13 && \text{from solution 5} &&= 2+3 \end{aligned}$$

Generally

$$r_n = max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \cdots, r_{n-1} + r_1)$$

Where to cut steel rods?

Optimal substructure

Optimal solutions to a problem incorporate optimal solutions to related subproblems, which we may solve independently.

A simpler equation

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

Where to cut steel rods?

Optimal substructure

Optimal solutions to a problem incorporate optimal solutions to related subproblems, which we may solve independently.

A simpler equation

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})$$

Recursive implementation

Recursive top-down implementation

$\textsc{Cut-Rod}(p, n)$

```
if n == 0
        return 0
q = −∞
for i = 1 to n
        q = max(q, p[i] + CUT-ROD(p, n − i))
return q
```

$$T(n) = 2^n$$

Recursive implementation

Recursive top-down implementation

CUT-ROD(p, n)

if $n == 0$
  return 0
$q = -\infty$
for $i = 1$ **to** n
  $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$
return q

$$T(n) = 2^n$$

Dynamic programming

Top-down with memoization

MEMOIZED-CUT-ROD(p, n)

```
let r[0..n] be a new array
for i = 0 to n
        r[i] = −∞
return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

Dynamic programming

Top-down with memoization

MEMOIZED-CUT-ROD-AUX(p, n, r)

if $r[n] \geq 0$
    return $r[n]$
if $n == 0$
    $q = 0$
else $q = -\infty$
    for $i = 1$ **to** n
        $q = \max(q, p[i] +$ MEMOIZED-CUT-ROD-AUX$(p, n - i, r))$
$r[n] = q$
return q

Dynamic programming

Bottom-up solution

BOTTOM-UP-CUT-ROD(p, n)

```
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
        q = -∞
        for i = 1 to j
                q = max(q, p[i] + r[j - i])
        r[j] = q
return r[n]
```

Reconstructing a solution

EXTENDED-BOTTOM-UP-CUT-ROD(p, n)

```
let r[0..n] and s[0..n] be new arrays
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        if q < p[i] + r[j - i]
            q = p[i] + r[j - i]
            s[j] = i
    r[j] = q
return r and s
```

Reconstructing a solution

PRINT-CUT-ROD-SOLUTION(p, n)

```
(r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
while n > 0
        print s[n]
        n = n − s[n]
```

Example

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

Reconstructing a solution

PRINT-CUT-ROD-SOLUTION(p, n)

$(r, s) =$ EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
while $n > 0$
    print $s[n]$
    $n = n - s[n]$

Example

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

Overview

Purpose

Give a sequence $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices to be multiplied, and we wish to compute the product: $A_1 A_2 \ldots A_n$.

Fully parenthesized

A product of matrices is **fully parenthesized** if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses.

Overview

Purpose

Give a sequence $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices to be multiplied, and we wish to compute the product: $A_1 A_2 \ldots A_n$.

Fully parenthesized

A product of matrices is **fully parenthesized** if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses.

Overview

Example

$(A_1(A_2(A_3A_4)))$, $(A_1((A_2A_3)A_4))$, $((A_1A_2)(A_3A_4))$, $((A_1(A_2A_3))A_4)$, $(((A_1A_2)A_3)A_4)$.

Overview

MATRIX-MULTIPLY(A, B)

if $A.columns \neq B.rows$
  error "incompatible dimensions"
else Let C be a new $A.rows \times B.columns$ matrix
  for $i = 1$ **to** $A.rows$
    for $j = 1$ **to** $B.columns$
      $c_{ij} = 0$
      for $k = 1$ **to** $A.columns$
        $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
  return C

Overview

Why parenthesized?

Different costs incurred by different parenthesizations!

Example

$A_{10\times 100}B_{100\times 5}C_{5\times 50}$

$A_{10\times 100}(B_{100\times 5}C_{5\times 50})$:

$10 \times 100 \times 50 + 100 \times 5 \times 50 = 75,000$

$(A_{10\times 100}B_{100\times 5})C_{5\times 50}$:

$10 \times 100 \times 5 + 10 \times 5 \times 50 = 7,500$

Overview

Why parenthesized?

Different costs incurred by different parenthesizations!

Example

$A_{10\times100}B_{100\times5}C_{5\times50}$

$A_{10\times100}(B_{100\times5}C_{5\times50})$:

$10\times100\times50+100\times5\times50=75,000$

$(A_{10\times100}B_{100\times5})C_{5\times50}$:

$10\times100\times5+10\times5\times50=7,500$

Brute force

Counting the number of parenthesization

Denote the number of alternative parenthesizations of a sequence of n matrices by $P(n)$, then:

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

$P(n)$ grows as $\Omega(4^n/n^{\frac{3}{2}})$.

Brute force

Counting the number of parenthesization

Denote the number of alternative parenthesizations of a sequence of n matrices by $P(n)$, then:

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

$P(n)$ grows as $\Omega(4^n/n^{\frac{3}{2}})$.

Step1: The structure of an optimal solution

- Suppose that an optimal parenzhesization of $A_i A_{i+1} \ldots A_j$ split the product between A_k and A_{k+1}.
- Then the parenthesization of "prefix" subchain $A_i A_{i+1} \ldots A_k$ within this optimal parenthesization of $A_i A_{i+1} \ldots A_j$ must be an optimal parenthesization of $A_i A_{i+1} \ldots A_k$.
- Similarly, subchain $A_{k+1} A_{k+2} \ldots A_j$ in the optimal parenthesization of must be an optimal parenthesization of $A_{k+1} A_{k+2} \ldots A_j$.

Step1: The structure of an optimal solution

- Suppose that an optimal parenzhesization of $A_iA_{i+1}\ldots A_j$ split the product between A_k and A_{k+1}.
- Then the parenthesization of "prefix" subchain $A_iA_{i+1}\ldots A_k$ within this optimal parenthesization of $A_iA_{i+1}\ldots A_j$ must be an optimal parenthesization of $A_iA_{i+1}\ldots A_k$.
- Similarly, subchain $A_{k+1}A_{k+2}\ldots A_j$ in the optimal parenthesization of must be an optimal parenthesization of $A_{k+1}A_{k+2}\ldots A_j$.

Step1: The structure of an optimal solution

- Suppose that an optimal parenzhesization of $A_iA_{i+1}\ldots A_j$ split the product between A_k and A_{k+1}.
- Then the parenthesization of "prefix" subchain $A_iA_{i+1}\ldots A_k$ within this optimal parenthesization of $A_iA_{i+1}\ldots A_j$ must be an optimal parenthesization of $A_iA_{i+1}\ldots A_k$.
- Similarly, subchain $A_{k+1}A_{k+2}\ldots A_j$ in the optimal parenthesization of must be an optimal parenthesization of $A_{k+1}A_{k+2}\ldots A_j$.

Step2: A recursive solution

- Let $m[i,j]$ be the minimum number of scalar multiplications needed to computer the matrix $A_{i...j}$; for the full problem, a cheapest way would thus be $m[1,n]$
- Let us assume that the optimal parenthesization splits the product $A_iA_{i+1}\dots A_j$ between A_k and A_{k+1}, where $i \leq k < j$.

Step2: A recursive solution

- Let $m[i,j]$ be the minimum number of scalar multiplications needed to computer the matrix $A_{i \ldots j}$; for the full problem, a cheapest way would thus be $m[1,n]$
- Let us assume that the optimal parenthesization splits the product $A_i A_{i+1} \ldots A_j$ between A_k and A_{k+1}, where $i \leq k < j$.

Step2: A recursive solution

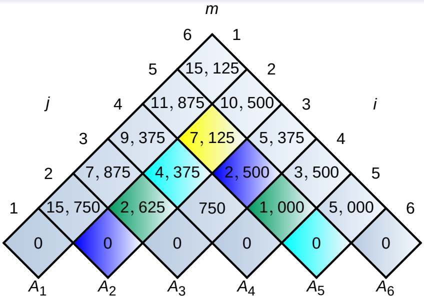
$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] \\ \qquad + p_{i-1} p_k p_j\} & \text{if } i < j. \end{cases}$$

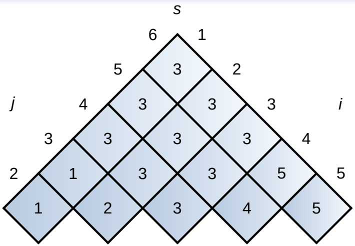
Step3: Computing the optimal costs

MATRIX-CHAIN-ORDER(p)

```
n = p.length - 1
for i = 1 to n
    m[i, i] = 0
for l = 2 to n          // l is the chain length.
    for i = 1 to n - l + 1
        j = i + l - 1
        m[i, j] = ∞
        for k = i to j - 1
            q = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j
            if q < m[i, j]
                m[i, j] = q
                s[i, j] = k
return m and s
```

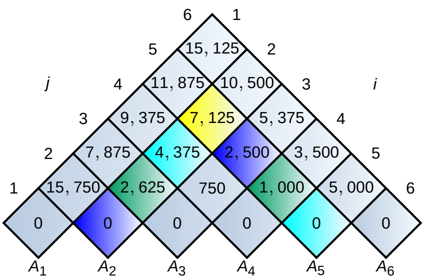
Step3: Computing the optimal costs

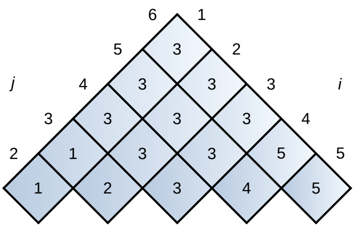




matrix	dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

Step3: Computing the optimal costs





$$m[2,5] = \min \left\{ \begin{array}{lll} m[2,2] + m[3,5] + p_1 p_2 p_5 & & \\ = 0 + 2500 + 35 \cdot 15 \cdot 20 & = & 13000 \\ m[2,3] + m[4,5] + p_1 p_3 p_5 & & \\ = 2625 + 1000 + 35 \cdot 5 \cdot 20 & = & 7125 \\ m[2,4] + m[5,5] + p_1 p_4 p_5 & & \\ = 4375 + 0 + 35 \cdot 10 \cdot 20 & = & 11375 \end{array} \right\}$$

Constructing an optimal solution

```
PRINT-OPTIMAL-PARENS(s, i, j)
if i = j
        print "A"_i
else print "("
        PRINT-OPTIMAL-PARENS(s, i, s[i, j])
        PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
        print ")"
```

Example result

$((A_1(A_2A_3))((A_4A_5)A_6))$

Constructing an optimal solution

PRINT-OPTIMAL-PARENS(s, i, j)

```
1  if i = j
2      print "A"_i
3  else print "("
           PRINT-OPTIMAL-PARENS(s, i, s[i, j])
           PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
           print ")"
```

Example result

$((A_1(A_2A_3))((A_4A_5)A_6))$

History

- In 1973, S.Godbole presented the $O(n^3)$ algorithm.
- In 1975, A. K. Chandra from IBM developed an $O(n)$ heuristic algorithm.
- In 1978, F. Y. Chin improved Chandra's algorithm, but the algorithm is also in $O(n)$.
- In 1982, T. C. Hu and M.T. Shing gave an $O(n \lg n)$-time algorithm by solving the equivalent problem of finding the optimal triangulation of a convex polygon.

History

- In 1994, P. Ramanan presented a simpler algorithm and obtained the tight lower bound of $\Omega(n \lg n)$ for a related problem.
- In 2003, H.Lee and S.J. Hong found an optimal product schedule for evaluating a chain of matrix products on a parallel computer.

Elements of dynamic programming

When should we look for a dynamic programming solution to a problem?

1. Optimal substructure
2. Overlapping subproblems

Optimal Substructure

How to discover optimal substructure?

1. Make a choice to split the problem into one or more subproblems;
2. Just assume you are given the choice that leads to an optimal solution;
3. Given this choice, try to best characterize the resulting space of subproblems;
4. Show the subproblems chosen are optimal by using a **"cut-and-paste"** technique.

Optimal Substructure

How to discover optimal substructure?

1. Make a choice to split the problem into one or more subproblems;
2. Just assume you are given the choice that leads to an optimal solution;
3. Given this choice, try to best characterize the resulting space of subproblems;
4. Show the subproblems chosen are optimal by using a **"cut-and-paste"** technique.

Optimal Substructure

How to discover optimal substructure?

1. Make a choice to split the problem into one or more subproblems;
2. Just assume you are given the choice that leads to an optimal solution;
3. Given this choice, try to best characterize the resulting space of subproblems;
4. Show the subproblems chosen are optimal by using a **"cut-and-paste"** technique.

Optimal Substructure

How to discover optimal substructure?

1. Make a choice to split the problem into one or more subproblems;
2. Just assume you are given the choice that leads to an optimal solution;
3. Given this choice, try to best characterize the resulting space of subproblems;
4. Show the subproblems chosen are optimal by using a **"cut-and-paste"** technique.

Optimal Substructure

Rule of thumb

Keep the space of subproblems as simple as possible.

Optimal Substructure

Optimal substructure varies in two ways

1. How many subproblems are used in an optimal solution to the original problem, and
2. How many choices we have in determining which subproblem(s) to use in an optimal solution.

Two factors decide the running time

1. the number of subproblems overall.
2. the number of choices for each subproblem.

Optimal Substructure

Optimal substructure varies in two ways

1. How many subproblems are used in an optimal solution to the original problem, and
2. How many choices we have in determining which subproblem(s) to use in an optimal solution.

Two factors decide the running time

1. the number of subproblems overall.
2. the number of choices for each subproblem.

Optimal Substructure

Example

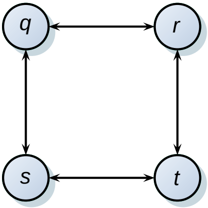
- In rod cutting, we had $\Theta(n)$ subproblems overall, and at most n choices to examine for each yielding a $\Theta(n^2)$ running time.
- For matrix-chain multiplication, there were $\Theta(n^2)$ subproblems overall, and in each we had at most $n - 1$ choices, giving an $O(n^3)$ running time.

Independence of subproblems

Unweighted longest simple path

- Given a directed graph $G = (V, E)$ and vertices $u, v \in V$. The unweighted longest simple path consists the most edges from u to v.
- Supposed we decompose a longest simple path $u \overset{p}{\leadsto} v$ into path $u \overset{p_1}{\leadsto} w \overset{p_2}{\leadsto} v$.
- Mustn't p_1 be a longest simple path from u to w, and mustn't p_2 be a longest simple path from w to v?

Independence of subproblems



Consider $q \to r \to t$, which is the longest simple path from q to t.

$$q \text{ to } r : q \to s \to t \to r$$

$$r \text{ to } t : r \to q \to s \to t.$$

Combining :

$$q \to s \to t \to r \to q \to s \to t$$

Independence of subproblems

Unweighted longest simple path

NO!

Why?

- The suproblems in finding the longest simple path are not **independent**.
- **independent**: The solution to one subproblem does not affect the solution to another subproblem of the same problem.

Overlapping subproblems

Overlapping subproblems

When a recursive algorithm revisits the same problem over and over again, we say that the optimization problem has **overlapping subproblems**.

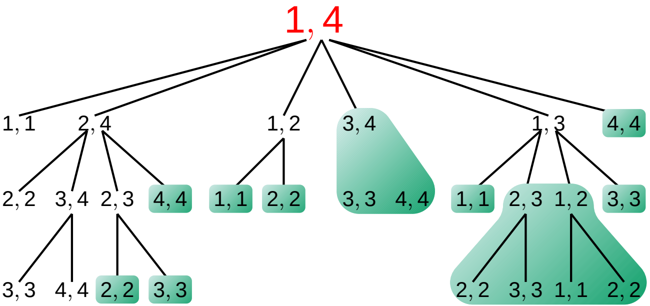
Overlapping subproblems

Example

RECURSIVE-MATRIX-CHAIN(p, i, j)

if $i = j$
    return 0
$m[i, j] = \infty$
for $k = i$ **to** $j - 1$
    $q =$ RECURSIVE-MATRIX-CHAIN(p, i, k)
        $+$RECURSIVE-MATRIX-CHAIN$(p, k + 1, j)$
        $+p_{i-1} p_k p_j$
    if $q < m[i, j]$
        $m[i, j] = q$
return $m[i, j]$

Overlapping subproblems



RECURSIVE-MATRIX-CHAIN$(p, 1, 4)$

$$T(n) \geq 2\sum_{i=1}^{n-1} T(i) + n \geq 2^{n-1}$$

Memoization

Memoization

- A memoized recursive algorithm maintains an entry in a table for the solution to each subproblem.
- When the subproblem is first encountered, its solution is computed and then stored in the table.
- Each subsequent time that the subproblem is encountered, just return the stored value in the table.

Memoization

Example

MEMOIZED-MATRIX-CHAIN(p)

```
n = p.length − 1
let m[1..n, 1..n] be a new table
for i = 1 to n
        for j = i to n
                m[i, j] = ∞
return LOOKUP-CHAIN(p, 1, n)
```

Memoization

Example

LOOKUP-CHAIN(p, i, j)

if $m[i,j] < \infty$
    return m[i,j]
if $i = j$
    $m[i,j] = 0$
else for $k = i$ **to** $j - 1$
        $q =$ LOOKUP-CHAIN(p, i, k)
            $+$LOOKUP-CHAIN$(p, k+1, j) + p_{i-1} p_k p_j$
        if $q < m[i,j]$
            $m[i,j] = q$
return $m[i,j]$

Memoization

When to use?

- If some subproblems in the subproblem space need not be solved at all, the memoized solution has the advantage of solving only subproblems that are definitely required.

Longest common subsequence

Definition

- Given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, another sequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a **subsequence** of X if there exists a strictly increasing sequence $\langle i_1, i_2, \ldots, i_k \rangle$ of indices of X such that for all $j = 1, 2, \ldots, k$, we have $x_{i_j} = z_j$.

Longest common subsequence

Definition

- Given two sequences X and Y, we say that Z is a **common subsequence** of X and Y if Z is a subsequence of both X and Y.
- In the **longest-common-subsequence problem**, we are given two sequences X and Y, and wish to find a maximum-length common subsequence of X and Y.

Longest common subsequence

Definition

- Given two sequences X and Y, we say that Z is a **common subsequence** of X and Y if Z is a subsequence of both X and Y.
- In the **longest-common-subsequence problem**, we are given two sequences X and Y, and wish to find a maximum-length common subsequence of X and Y.

Longest common subsequence

Example

- If $X = \langle A, B, C, B, D, A, B \rangle$, $Y = \langle B, D, C, A, B, A \rangle$, the sequence $\langle B, C, A \rangle$ is a common subsequence of both X and Y, but not a longest common subsequence of both X and Y. The sequence $\langle B, C, B, A \rangle$ is an LCS of X and Y.

Longest common subsequence

Example

- In biological applications, the DNA of one organism may be $S_1 =$ $ACCGGTCGAGTGCGCGGAAGCCG$, while the DNA of another organism may be $S_2 =$$GTCGTTCGGAATGCCGTT$. One goal of comparing two strands of DNA is to determine how "similar" the two strands are. In our example, an LCS of S_1 and S_2 is $S_3 =$$GTCGTCGGAAGCCG$.

Longest common subsequence

Example

- In biological applications, the DNA of one organism may be $S_1 = ACCGGTCGAGTGCGCGGAAGCCG$, while the DNA of another organism may be $S_2 = GTCGTTCGGAATGCCGTT$. One goal of comparing two strands of DNA is to determine how "similar" the two strands are. In our example, an LCS of S_1 and S_2 is $S_3 = GTCGTCGGAAGCCG$.

Longest common subsequence

longest-common-subsequence problem

Find out an LCS of two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$.

Step1: Characterizing an LCS

Theorem 15.1(Optimal substructure of an LCS)

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Step1: Characterizing an LCS

Theorem 15.1(Optimal substructure of an LCS)

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Step1: Characterizing an LCS

Theorem 15.1(Optimal substructure of an LCS)

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Step1: Characterizing an LCS

Theorem 15.1(Optimal substructure of an LCS)

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Step2: A recursive solution

Define $c[i, j]$ to be the length of an LCS of the sequences X_i and Y_j, then

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

Step3: Computing the length of an LCS

LCS-LENGTH(X, Y)

$m = X.length$
$n = Y.length$
for $i = 1$ **to** m
    $c[i, 0] = 0$
for $j = 0$ **to** n
    $c[0, j] = 0$

Step3: Computing the length of an LCS

Start

```
for i = 1 to m
    for j = 1 to n
        if x_i = y_j
            c[i,j] = c[i-1,j-1] + 1
            b[i,j] = "↖"
        else if c[i-1,j] ≥ c[i,j-1]
                c[i,j] = c[i-1,j]
                b[i,j] = "↑"
            else c[i,j] = c[i,j-1]
                b[i,j] = "←"
return c and b
```

Running time

$$O(mn)$$

Step3: Computing the length of an LCS

Start

```
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i,j] = c[i-1,j-1] + 1
            b[i,j] = "↖"
        else if c[i-1,j] ≥ c[i,j-1]
            c[i,j] = c[i-1,j]
            b[i,j] = "↑"
        else c[i,j] = c[i,j-1]
            b[i,j] = "←"
return c and b
```

Running time

$O(mn)$

An Example of LCS

j		0	1	2	3	4	5	6
i		y_i	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

$$X = \langle A, B, C, B, D, A, B \rangle$$
$$Y = \langle B, D, C, A, B, A \rangle$$

Step4: Constructing an LCS

PRINT-LCS(b, X, i, j)

if $i = 0$ or $j = 0$
    return
if $b[i, j] =$ "$\nwarrow$"
    PRINT-LCS$(b, X, i-1, j-1)$
    print x_i
elseif $b[i, j] =$ "$\uparrow$"
    PRINT-LCS$(b, X, i-1, j)$
else PRINT-LCS$(b, X, i, j-1)$

Overview

Problem

Considering a word-to-word translation system from English to French, we need to look up dictionary as efficient as possible.

Overview

Problem

Considering a word-to-word translation system from English to French, we need to look up dictionary as efficient as possible.

Definition

- Given a sequence $K = \langle k_1, k_2, \ldots, k_n \rangle$ of n distinct keys in sorted order, and $n+1$ "dummy keys", $d_0, d_1, \ldots, d_n$ for values not in K, we wish to build a binary search tree.
- d_i represents all values between k_i and k_{i+1}.

Overview

Problem

Considering a word-to-word translation system from English to French, we need to look up dictionary as efficient as possible.

Definition

- Given a sequence $K = \langle k_1, k_2, \ldots, k_n \rangle$ of n distinct keys in sorted order, and $n + 1$ "dummy keys", $d_0, d_1, \ldots, d_n$ for values not in K, we wish to build a binary search tree.
- d_i represents all values between k_i and k_{i+1}.

Overview

Problem

Considering a word-to-word translation system from English to French, we need to look up dictionary as efficient as possible.

Definition

- For each key k_i, we have a probability p_i that a search will be for k_i; and for each dummy key d_i, we have a probability q_i.

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

Overview

Optimal binary search tree

A binary search tree whose expected search cost is smallest.

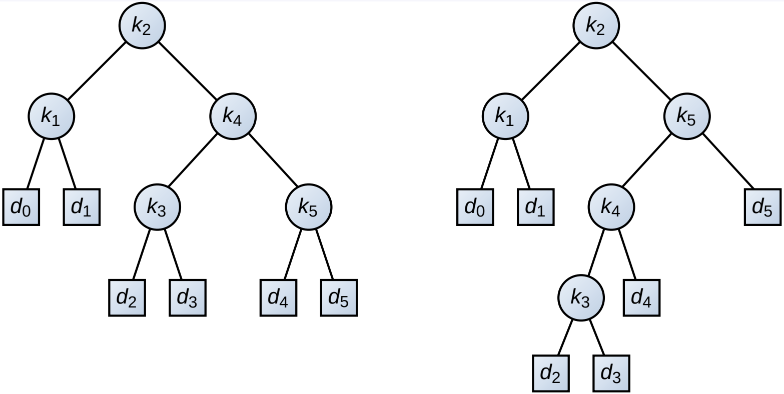
Overview

Expected cost of a search in a binary search tree T

$$E = \sum_{i=1}^{n}(\mathrm{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n}(\mathrm{depth}_T(d_i) + 1) \cdot q_i$$

$$= 1 + \sum_{i=1}^{n} \mathrm{depth}_T(k_i) \cdot p_i + \sum_{i=0}^{n} \mathrm{depth}_T(d_i) \cdot q_i$$

Two binary search trees

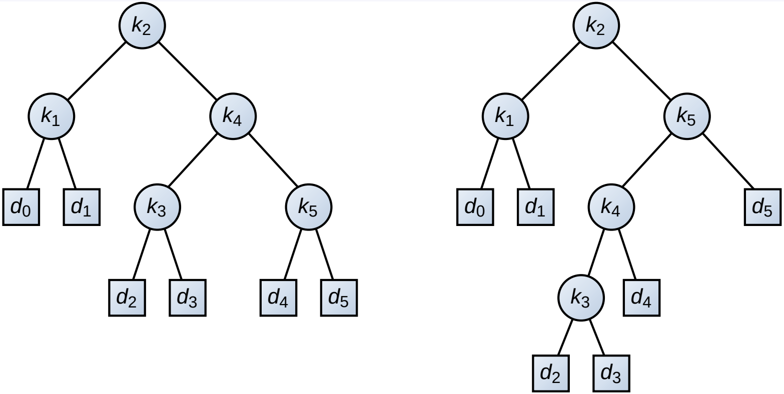


i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

search cost

left tree $= 2.80$,
right tree $= 2.75$

Two binary search trees

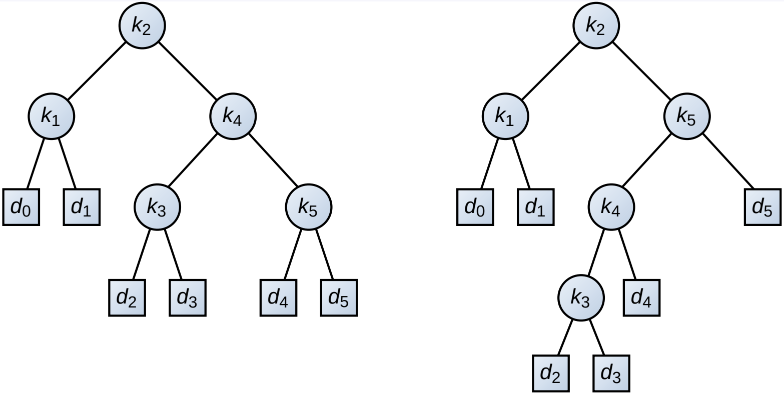


i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

search cost

left tree $= 2.80$,
right tree $= 2.75$

Two binary search trees



i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

search cost

left tree $= 2.80$,
right tree $= 2.75$

Step1:The optimal structure

- If an optimal binary search tree T has a subtree T' containing keys $k_i, \ldots, k_j$, then this subtree T' must be optimal as well for the subproblem with keys $k_i, \ldots, k_j$ and dummy keys $d_{i-1}, \ldots, d_j$.
- Given keys $k_i, \ldots, k_j$, if $k_r (i \leq r \leq j)$ is the root of an optimal subtree, the left subtree of the root k_r will containing $k_i, \ldots, k_{r-1}$, and the right subtree will contain $k_{r+1}, \ldots, k_j$.

Step1:The optimal structure

- If an optimal binary search tree T has a subtree T' containing keys $k_i, \ldots, k_j$, then this subtree T' must be optimal as well for the subproblem with keys $k_i, \ldots, k_j$ and dummy keys $d_{i-1}, \ldots, d_j$.
- Given keys $k_i, \ldots, k_j$, if $k_r (i \leq r \leq j)$ is the root of an optimal subtree, the left subtree of the root k_r will containing $k_i, \ldots, k_{r-1}$, and the right subtree will contain $k_{r+1}, \ldots, k_j$.

Step2: A recursive solution

- Define $e[i,j]$ as the expected cost of searching an optimal binary search tree.
- if k_r is the root of an optimal tree, we have

$$e[i,j] = p_r + (e[i,r-1] + w(i,r-1)) + (e[r+1,j] + w(r+1,j))$$

Step2: A recursive solution

- Define $e[i, j]$ as the expected cost of searching an optimal binary search tree.
- if k_r is the root of an optimal tree, we have

$$
\begin{aligned}
e[i, j] = p_r &+ (e[i, r-1] + w(i, r-1)) \\
&+ (e[r+1, j] + w(r+1, j))
\end{aligned}
$$

Step2: A recursive solution

- For a subtree with keys $k_i, \ldots, k_j$, the sum of probabilities are

$$w(i,j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$$

Step2: A recursive solution

- Noting that

$$w(i,j) = w(i,r-1) + p_r + w(r+1,j).$$

- we rewrite $e[i,j]$ as

$$e(i,j) = e[i,r-1] + e[r+1,j] + w(i,j).$$

Step2: A recursive solution

- Noting that

$$w(i,j) = w(i,r-1) + p_r + w(r+1,j).$$

- we rewrite $e[i,j]$ as

$$e(i,j) = e[i,r-1] + e[r+1,j] + w(i,j).$$

Step2: A recursive solution

- so

$$e[i,j] = \begin{cases} q_{i-1} & \text{if } j = i-1, \\ \min\limits_{i \le r \le j} \{e[i,r-1] \\ \qquad + e[r+1,j] + w(i,j)\} & \text{if } i \le j. \end{cases}$$

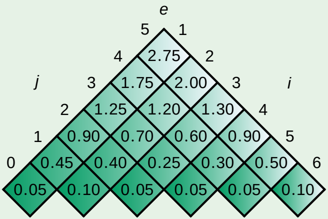
Step3: Computing the search cost

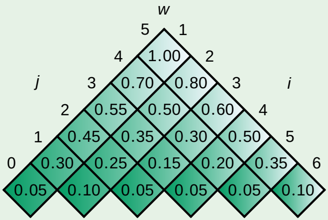
OPTIMAL-BST(p, q, n)

```
for i = 1 to n + 1
    e[i, i - 1] = q_{i-1}
    w[i, i - 1] = q_{i-1}
for l = 1 to n
    for i = 1 to n - l + 1
        j = i + l - 1
        e[i, j] = ∞
        w[i, j] = w[i, j - 1] + p_j + q_j
        for r = i to j
            t = e[i, r - 1] + e[r + 1, j] + w[i, j]
            if t < e[i, j]
                e[i, j] = t
                root[i, j] = r
return e and root
```

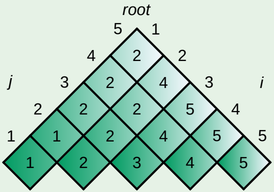
Step3: Computing the search cost

Example





i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10



History

- In 1959, E. N. Gilbert and E. F. Moore from Bell Labs published a paper on constructing optimal binary search trees for the case in which all probabilities p_i are 0; this paper contains an $O(n^3)$-time algorithm.
- In 1971, T. C. Hu and A. C. Tucker devised an algorithm for the case in which all probabilities p_i are 0 that uses $O(n^2)$ time and $O(n)$ space; In 1973, Knuth reduced the time to $O(n \lg n)$.

History

- In 1974, A. V. Aho, J. E. Hopcroft, and J. D. Ullman present the algorithm we just discussed.