

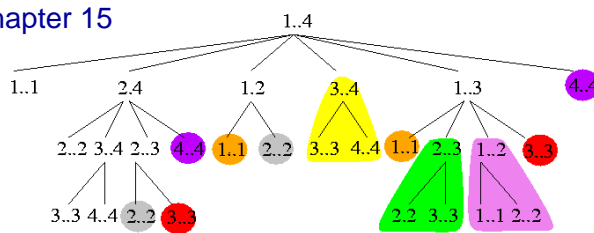
Unit 4: Dynamic Programming

- **Course contents:**

- Assembly-line scheduling
- Matrix-chain multiplication
- Longest common subsequence
- Optimal binary search trees
- Applications: Optimal polygon triangulation, flip-chip routing, technology mapping for logic synthesis

- **Reading:**

- Chapter 15



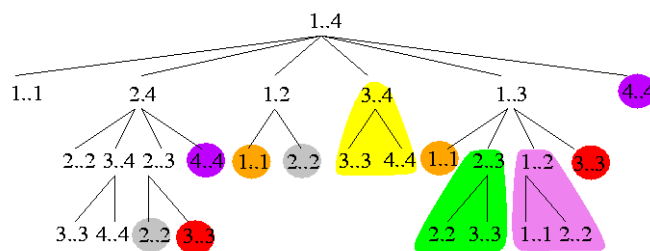
Unit 4

Y.-W. Chang

1

Dynamic Programming (DP) v.s. Divide-and-Conquer

- Both solve problems by combining the solutions to subproblems.
- Divide-and-conquer algorithms
 - Partition a problem into **independent** subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem.
 - Inefficient if they solve the same subproblem more than once.
- Dynamic programming (DP)
 - Applicable when the subproblems are **not independent**.
 - DP solves each subproblem just once.

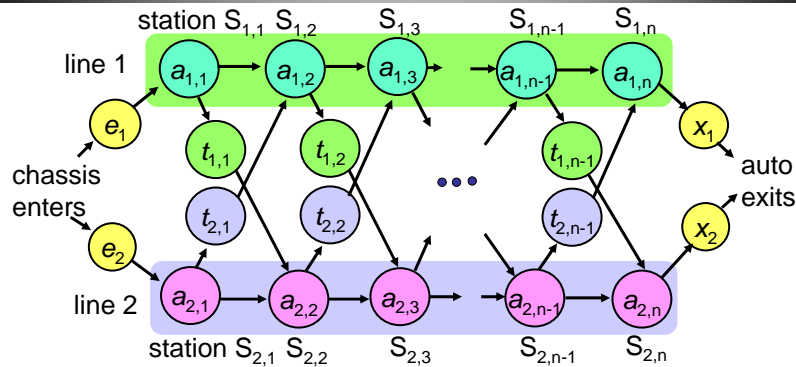


Unit 4

Y.-W. Chang

2

Assembly-line Scheduling



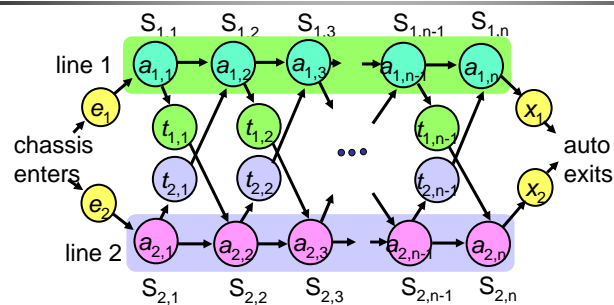
- An auto chassis enters each assembly line, has parts added at stations, and a finished auto exits at the end of the line.
 - $S_{i,j}$: the j th station on line i
 - $a_{i,j}$: the assembly time required at station $S_{i,j}$
 - $t_{i,j}$: transfer time from station $S_{i,j}$ to the $j+1$ station of the other line.
 - $e_i (x_i)$: time to enter (exit) line i

Unit 4

Y.-W. Chang

3

Optimal Substructure



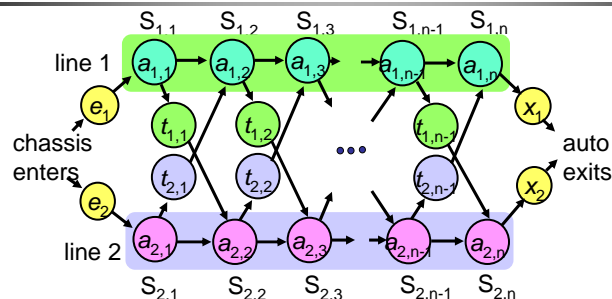
- Objective: Determine the stations to choose to minimize the total manufacturing time for one auto.
 - Brute force: $\Omega(2^n)$, why?
 - The problem is linearly ordered and cannot be rearranged => Dynamic programming?
- **Optimal substructure:** If the fastest way through station $S_{i,j}$ is through $S_{i,j-1}$, then the chassis must have taken a fastest way from the starting point through $S_{i,j-1}$.

Unit 4

Y.-W. Chang

4

Overlapping Subproblem: Recurrence



- **Overlapping subproblem:** The fastest way through station $S_{1,j}$ is either through $S_{1,j-1}$ and then $S_{1,j}$, or through $S_{2,j-1}$ and then transfer to line 1 and through $S_{1,j}$.

- $f_1[j]$: fastest time from the starting point through $S_{1,j}$

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{if } j = 1 \\ \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{if } j \geq 2 \end{cases}$$

- The fastest time all the way through the factory

$$f^* = \min(f_1[n] + x_1, f_2[n] + x_2)$$

Unit 4

Y.-W. Chang

5

Computing the Fastest Time

Fastest-Way(a, t, e, x, n)

```

1.  $f_1[1] \leftarrow e_1 + a_{1,1}$ 
2.  $f_2[1] \leftarrow e_2 + a_{2,1}$ 
3. for  $j \leftarrow 2$  to  $n$ 
4.   do if  $f_1[j-1] + a_{1,j} \leq f_2[j-1] + t_{2,j-1} + a_{1,j}$ 
5.     then  $f_1[j] \leftarrow f_1[j-1] + a_{1,j}$ 
6.         $l_1[j] \leftarrow 1$ 
7.   else  $f_1[j] \leftarrow f_2[j-1] + t_{2,j-1} + a_{1,j}$ 
8.         $l_1[j] \leftarrow 2$ 
9.   do if  $f_2[j-1] + a_{2,j} \leq f_1[j-1] + t_{1,j-1} + a_{2,j}$ 
10.    then  $f_2[j] \leftarrow f_2[j-1] + a_{2,j}$ 
11.        $l_2[j] \leftarrow 2$ 
12.    else  $f_2[j] \leftarrow f_1[j-1] + t_{1,j-1} + a_{2,j}$ 
13.        $l_2[j] \leftarrow 1$ 
14. if  $f_1[n] + x_1 \leq f_2[n] + x_2$ 
15. then  $f^* = f_1[n] + x_1$ 
16.      $l^* = 1$ 
17. else  $f^* = f_2[n] + x_2$ 
18.      $l^* = 2$ 

```

Running time?

- $l_1[j]$: The line number whose station $j-1$ is used in a fastest way through $S_{1,j}$

Unit 4

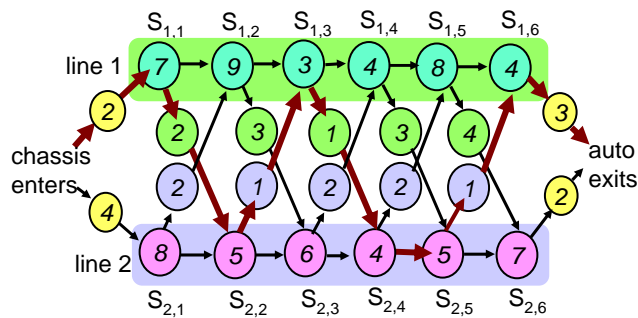
Y.-W. Chang

6

Constructing the Fastest Way

Print-Station(l, n)

- | | |
|--|---|
| <ol style="list-style-type: none"> 1. $i \leftarrow l^*$ 2. Print "line" i ", station " n 3. for $j \leftarrow n$ downto 2 4. do $i \leftarrow l[j]$ 5. Print "line" i ", station " $j-1$ | <p>line 1, station 6
 line 2, station 5
 line 2, station 4
 line 1, station 3
 line 2, station 2
 line 1, station 1</p> |
|--|---|

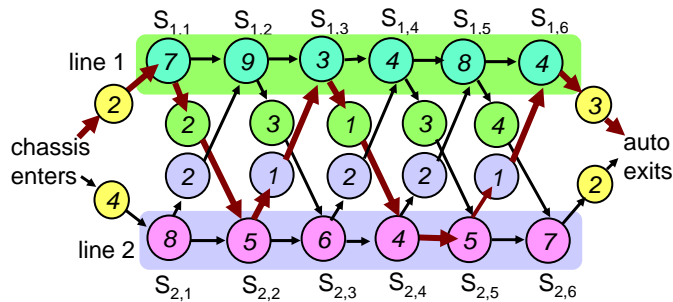


Unit 4

Y.-W. Chang

7

An Example



$f_1[j]$	9	18	20	24	32	35	$f^* = 38$
$f_2[j]$	12	16	22	25	30	37	
j	1	2	3	4	5	6	

$l_1[j]$	1	2	1	1	2	$f^* = 1$
$l_2[j]$	1	2	1	2	2	

Unit 4

Y.-W. Chang

8

Dynamic Programming (DP)

- Typically apply to **optimization problem**.
- Generic approach
 - Calculate the solutions to all subproblems.
 - Proceed computation from the small subproblems to the larger subproblems.
 - Compute a subproblem based on previously computed results for smaller subproblems.
 - Store the solution to a subproblem in a table and never recompute.
- Development of a DP
 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 bottom-up.
 4. Construct an optimal solution from computed information (omitted if only the optimal value is required).

Unit 4

Y.-W. Chang

9

When to Use Dynamic Programming (DP)

- DP computes recurrence efficiently by storing partial results \Rightarrow efficient only when the number of partial results is small.
- **Hopeless configurations:** $n!$ permutations of an n -element set, 2^n subsets of an n -element set, etc.
- **Promising configurations:** $\sum_{i=1}^n i = n(n+1)/2$ contiguous substrings of an n -character string, $n(n+1)/2$ possible subtrees of a binary search tree, etc.
- **DP works best on objects that are linearly ordered and cannot be rearranged!!**
 - Matrices in a chain, characters in a string, points around the boundary of a polygon, points on a circle, the left-to-right order of leaves in a search tree, etc.
 - Objects are ordered left-to-right \Rightarrow Smell DP?

Unit 4

Y.-W. Chang

10

DP Example: Matrix-Chain Multiplication

- If A is a $p \times q$ matrix and B a $q \times r$ matrix, then $C = AB$ is a $p \times r$ matrix

$$C[i, j] = \sum_{k=1}^q A[i, k] B[k, j]$$

time complexity: $O(pqr)$.

- **The matrix-chain multiplication problem:** Given a chain $\langle A_1, A_2, \dots, A_n \rangle$ of n matrices, matrix A_i has dimension $p_{i-1} \times p_i$, parenthesize the product $A_1 A_2 \dots A_n$ to minimize the number of scalar multiplications.
- **Exp:** dimensions: $A_1: 4 \times 2$; $A_2: 2 \times 5$; $A_3: 5 \times 1$
 $(A_1 A_2) A_3$: total multiplications = $4 \times 2 \times 5 + 4 \times 5 \times 1 = 60$.
 $A_1 (A_2 A_3)$: total multiplications = $2 \times 5 \times 1 + 4 \times 2 \times 1 = 18$.
- **So the order of multiplications can make a big difference!**

Matrix-Chain Multiplication: Brute Force

- $A = A_1 A_2 \dots A_n$: How to evaluate A using the minimum number of multiplications?
- Brute force: check all possible orders?
 - $P(n)$: number of ways to multiply n matrices.

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

- $P(n) = \Omega\left(\frac{4^n}{n^{3/2}}\right)$, **exponential in n** .
- Any efficient solution?
 - The matrix chain **is linearly ordered and cannot be rearranged!!**
 - Smell Dynamic programming?

Matrix-Chain Multiplication

- $m[i, j]$: minimum number of multiplications to compute matrix $A_{i..j} = A_i A_{i+1} \dots A_j$, $1 \leq i \leq j \leq n$.
 - $m[1, n]$: the cheapest cost to compute $A_{1..n}$.

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

- Applicability of dynamic programming
 - **Optimal substructure**: an optimal solution contains within its optimal solutions to subproblems.
 - **Overlapping subproblem**: a recursive algorithm revisits the same problem over and over again; only $\theta(n^2)$ subproblems.

Unit 4

Y.-W. Chang

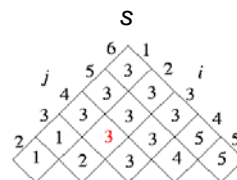
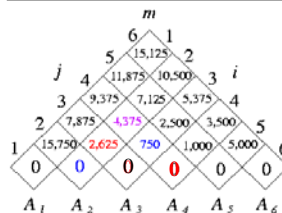
13

Bottom-Up DP Matrix-Chain Order

```

Matrix-Chain-Order(p)
1. n ← length[p]-1;
2. for i ← 1 to n
3.   m[i, i] ← 0;
4. for l ← 2 to n
5.   for i ← 1 to n-l+1
6.     j ← i+l-1;
7.     m[i, j] ← ∞;
8.     for k ← i to j-1
9.       q ← m[i, k] + m[k+1, j] + pi-1pkpj;
10.      if q < m[i, j]
11.        m[i, j] ← q;
12.        s[i, j] ← k;
13. return m and s
    
```

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



$$m[2, 4] = \min \begin{cases} m[2, 2] + m[3, 4] + p_1 p_2 p_4 = 0 + 750 + 35 \times 15 \times 10 = 6000. \\ m[2, 3] + m[4, 4] + p_1 p_3 p_4 = 2625 + 0 + 35 \times 5 \times 10 = 4375. \end{cases}$$

Unit 4

Y.-W. Chang

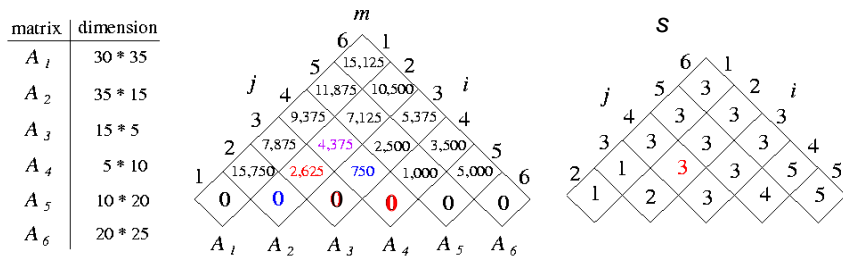
14

Constructing an Optimal Solution

- $s[i, j]$: value of k such that the optimal parenthesization of $A_i A_{i+1} \dots A_j$ splits between A_k and A_{k+1} .
- Optimal matrix $A_{1..n}$ multiplication: $A_{1..s[1, n]} A_{s[1, n]+1..n}$.
- **Exp:** call Matrix-Chain-Multiply($A, s, 1, 6$): $((A_1 (A_2 A_3))((A_4 A_5) A_6))$.

```

Matrix-Chain-Multiply( $A, s, i, j$ )
1. if  $j > i$ 
2.  $X \leftarrow$  Matrix-Chain-Multiply( $A, s, i, s[i, j]$ );
3.  $Y \leftarrow$  Matrix-Chain-Multiply( $A, s, s[i, j]+1, j$ );
4. return Matrix-Multiply( $X, Y$ );
5. else return  $A_i$ ;
    
```



Unit 4

Y.-W. Chang

15

Two Approaches to DP

1. Bottom-up iterative approach
 - Start with recursive divide-and-conquer algorithm.
 - Find the dependencies between the subproblems (whose solutions are needed for computing a subproblem).
 - Solve the subproblems in the correct order.
 2. Top-down recursive approach (**memoization**)
 - Start with recursive divide-and-conquer algorithm.
 - Keep top-down approach of original algorithms.
 - Save solutions to subproblems in a table (possibly a lot of storage).
 - Recurse only on a subproblem if the solution is not already available in the table.
- If all subproblems must be solved at least once, bottom-up DP is better due to less overhead for recursion and for maintaining tables.
 - If many subproblems need not be solved, top-down DP is better since it computes only those required.

Unit 4

Y.-W. Chang

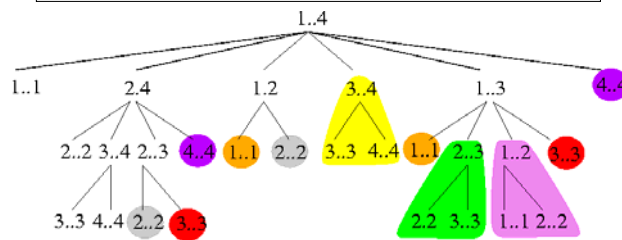
16

Top-Down, Recursive Matrix-Chain Order

- Time complexity: $\Omega(2^n)$ ($T(n) > \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1)$).

```

Recursive-Matrix-Chain(p, i, j)
1. if i = j
2.   return 0;
3. m[i, j] ← ∞;
4. for k ← i to j-1
5.   q ← Recursive-Matrix-Chain(p, i, k)
      + Recursive-Matrix-Chain(p, k+1, j) + pi-1pkpj;
6.   if q < m[i, j]
7.     m[i, j] ← q;
8. return m[i, j];
    
```



Unit 4

Y.-W. Chang

17

Top-Down DP Matrix-Chain Order (Memoization)

- Complexity: $O(n^2)$ space for $m[]$ matrix and $O(n^3)$ time to fill in $O(n^2)$ entries (each takes $O(n)$ time)

```

Memoized-Matrix-Chain(p)
1. n ← length[p]-1;
2. for i ← 1 to n
3.   for j ← i to n
4.     m[i, j] ← ∞;
5. return Lookup-Chain(p, 1, n);
    
```

```

Lookup-Chain(p, i, j)
1. If m[i, j] < ∞
2.   return m[i, j];
3. if i = j
4.   m[i, j] ← 0;
5. else for k ← i to j-1
6.   q ← Lookup-Chain(p, i, k) + Lookup-Chain(p, k+1, j) + pi-1pkpj;
7.   if q < m[i, j]
8.     m[i, j] ← q;
9. return m[i, j];
    
```

Unit 4

Y.-W. Chang

18

Longest Common Subsequence

- **Problem:** Given $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$, find the **longest common subsequence (LCS)** of X and Y .
- **Exp:** $X = \langle a, b, c, b, d, a, b \rangle$ and $Y = \langle b, d, c, a, b, a \rangle$
LCS = $\langle b, c, b, a \rangle$ (also, LCS = $\langle b, d, a, b \rangle$).
- Exp: DNA sequencing:
 - $S1 = \text{ACCGGTCGAGATGCAG};$
 - $S2 = \text{GTCGTTCGGAATGCAT};$
 - $\text{LCS } S3 = \text{CGTCGGATGCA}$
- Brute-force method:
 - Enumerate all subsequences of X and check if they appear in Y .
 - Each subsequence of X corresponds to a subset of the indices $\{1, 2, \dots, m\}$ of the elements of X .
 - There are 2^m subsequences of X . Why?

Unit 4

Y.-W. Chang

19

Optimal Substructure for LCS

- $c[i, j]$: length of the LCS of X_i and Y_j , where
 $X_i = \langle x_1, x_2, \dots, x_i \rangle$ and
 $Y_j = \langle y_1, y_2, \dots, y_j \rangle$.
- $c[m, n]$: length of LCS of X and Y .
- Basis: $c[0, j] = 0$ and $c[i, 0] = 0$.

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

Unit 4

Y.-W. Chang

20

Top-Down DP for LCS

- $c[i, j]$: length of the LCS of X_i and Y_j , where $X_i = \langle x_1, x_2, \dots, x_i \rangle$ and $Y_j = \langle y_1, y_2, \dots, y_j \rangle$.
- $c[m, n]$: LCS of X and Y .
- Basis: $c[0, j] = 0$ and $c[i, 0] = 0$.

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

- The top-down dynamic programming: initialize $c[i, 0] = c[0, j] = 0$, $c[i, j] = \text{NIL}$

TD-LCS(i, j)

1. if $c[i, j] = \text{NIL}$
2. if $x_i = y_j$
3. $c[i, j] = \text{TD-LCS}(i-1, j-1) + 1$
4. else $c[i, j] = \max(\text{TD-LCS}(i, j-1), \text{TD-LCS}(i-1, j))$
5. return $c[i, j]$

Unit 4

Y.-W. Chang

21

Bottom-Up DP for LCS

- Find the right order to solve the subproblems.
- To compute $c[i, j]$, we need $c[i-1, j-1]$, $c[i-1, j]$, and $c[i, j-1]$.
- $b[i, j]$: points to the table entry w.r.t. the optimal subproblem solution chosen when computing $c[i, j]$.

LCS-Length(X, Y)

1. $m \leftarrow \text{length}[X]$;
2. $n \leftarrow \text{length}[Y]$;
3. for $i \leftarrow 1$ to m
4. $c[i, 0] \leftarrow 0$;
5. for $j \leftarrow 0$ to n
6. $c[0, j] \leftarrow 0$;
7. for $i \leftarrow 1$ to m
8. for $j \leftarrow 1$ to n
9. if $x_i = y_j$
10. $c[i, j] \leftarrow c[i-1, j-1] + 1$
11. $b[i, j] \leftarrow \text{"↖"}$
12. else if $c[i-1, j] \geq c[i, j-1]$
13. $c[i, j] \leftarrow c[i-1, j]$
14. $b[i, j] \leftarrow \text{"↑"}$
15. else $c[i, j] \leftarrow c[i, j-1]$
16. $b[i, j] \leftarrow \text{"←"}$
17. return c and b

Unit 4

Y.-W. Chang

22

Example of LCS

- LCS time and space complexity: $O(mn)$.
- $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle \Rightarrow \text{LCS} = \langle B, C, B, A \rangle$.

j	0	1	2	3	4	5	6
i	y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0
1	A	0	0	0	1	1	1
2	B	1	1	1	2	2	2
3	C	0	1	2	2	2	2
4	B	0	1	2	2	3	3
5	D	0	1	2	2	3	3
6	A	0	1	2	2	3	4
7	B	0	1	2	2	3	4

Unit 4

Y.-W. Chang

23

Constructing an LCS

- Trace back from $b[m, n]$ to $b[1, 1]$, following the arrows: $O(m+n)$ time.

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

1. if $i = 0$ or $j = 0$
2. return;
3. if $b[i, j] = \nwarrow$
4. Print-LCS($b, X, i-1, j-1$);
5. print x_i ;
6. else if $b[i, j] = \uparrow$
7. Print-LCS($b, X, i-1, j$);
8. else Print-LCS($b, X, i, j-1$);

j	0	1	2	3	4	5	6
i	y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0
1	A	0	0	0	1	1	1
2	B	1	1	1	2	2	2
3	C	0	1	2	2	2	2
4	B	0	1	2	2	3	3
5	D	0	1	2	2	3	3
6	A	0	1	2	2	3	4
7	B	0	1	2	2	3	4

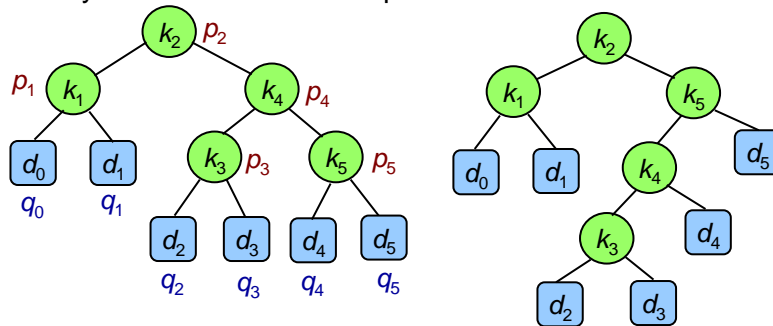
Unit 4

Y.-W. Chang

24

Optimal Binary Search Tree

- Given a sequence $K = \langle k_1, k_2, \dots, k_n \rangle$ of n distinct keys in sorted order ($k_1 < k_2 < \dots < k_n$) and a set of probabilities $P = \langle p_1, p_2, \dots, p_n \rangle$ for searching the keys in K and $Q = \langle q_0, q_1, q_2, \dots, q_n \rangle$ for unsuccessful searches (corresponding to $D = \langle d_0, d_1, d_2, \dots, d_n \rangle$ of $n+1$ distinct dummy keys with d_i representing all values between k_i and k_{i+1}), construct a binary search tree whose expected search cost is smallest.



Unit 4

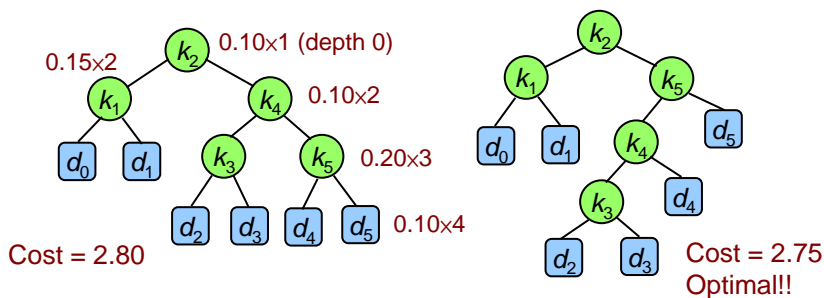
Y.-W. Chang

25

An 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

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



$$E[\text{search cost in } T] = \sum_{i=1}^n (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^n (\text{depth}_T(d_i) + 1) \cdot q_i$$

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

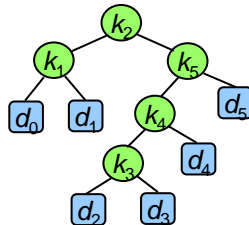
Unit 4

Y.-W. Chang

26

Optimal Substructure

- If an optimal binary search tree T has a subtree T' containing keys k_i, \dots, k_j , then this subtree T' must be optimal as well for the subproblem with keys k_i, \dots, k_j and dummy keys d_{i-1}, \dots, d_j .
 - Given keys k_i, \dots, k_j with k_r ($i \leq r \leq j$) as the root, the left subtree contains the keys k_i, \dots, k_{r-1} (and dummy keys d_{i-1}, \dots, d_{r-1}) and the right subtree contains the keys k_{r+1}, \dots, k_j (and dummy keys d_r, \dots, d_j).
 - For the subtree with keys k_i, \dots, k_j with root k_i , the left subtree contains keys k_i, \dots, k_{i-1} (**no key**) with the dummy key d_{i-1} .



Unit 4

Y.-W. Chang

27

Overlapping Subproblem: Recurrence

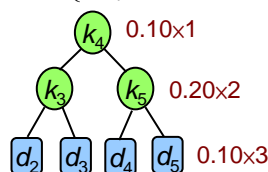
- $e[i, j]$: expected cost of searching an optimal binary search tree containing the keys k_i, \dots, k_j .
 - Want to find $e[1, n]$.
 - $e[i, i-1] = q_{i-1}$ (only the dummy key d_{i-1}).
- If k_r ($i \leq r \leq j$) is the root of an optimal subtree containing keys k_i, \dots, k_j and let $w(i, j) = \sum_{l=i}^j p_l + \sum_{l=i-1}^j q_l$ then

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

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

$$e[i, j] = \begin{cases} q_{i-1} & \text{if } j = i-1 \\ \min_{i \leq r \leq j} \{e[i, r-1] + e[r+1, j] + w(i, j)\} & \text{if } i \leq j \end{cases}$$

Node depths increase by 1 after merging two subtrees, and so do the costs



Unit 4

Y.-W. Chang

28

Computing the Optimal Cost

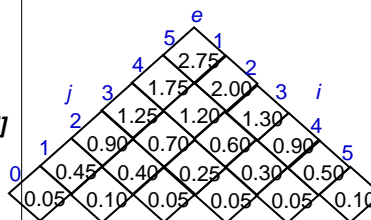
- Need a table $e[1..n+1, 0..n]$ for $e[i, j]$ (why $e[1, 0]$ and $e[n+1, n]$?)
- Apply the recurrence to compute $w(i, j)$ (why?)

$$w[i, j] = \begin{cases} q_{i-1} & \text{if } j = i - 1 \\ w[i, j-1] + p_j + q_i & \text{if } i \leq j \end{cases}$$

Optimal-BST(p, q, n)

1. for $i \leftarrow 1$ to $n + 1$
2. $e[i, i-1] \leftarrow q_{i-1}$
3. $w[i, i-1] \leftarrow q_{i-1}$
4. for $l \leftarrow 1$ to n
5. for $i \leftarrow 1$ to $n - l + 1$
6. $j \leftarrow i + l - 1$
7. $e[i, j] \leftarrow \infty$
8. $w[i, j] \leftarrow w[i, j-1] + p_j + q_i$
9. for $r \leftarrow i$ to j
10. $t \leftarrow e[i, r-1] + e[r+1, j] + w[i, j]$
11. if $t < e[i, j]$ then
12. $e[i, j] \leftarrow t$
13. $root[i, j] \leftarrow r$
14. return e and $root$

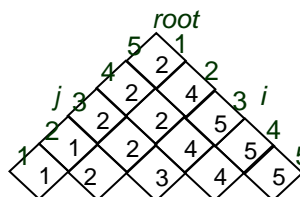
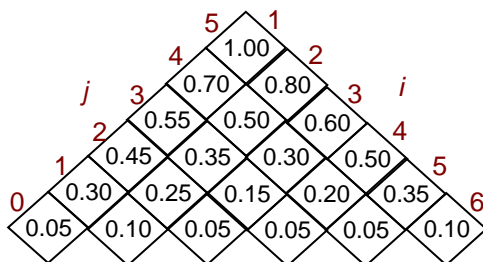
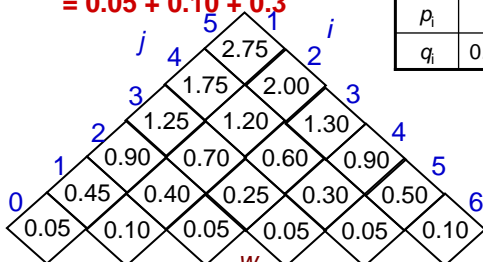
- $root[i, j]$: index r for which k_r is the root of an optimal search tree containing keys k_i, \dots, k_j .



Example

$$e[1, 1] = e[1, 0] + e[2, 1] + w(1, 1) = 0.05 + 0.10 + 0.3$$

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



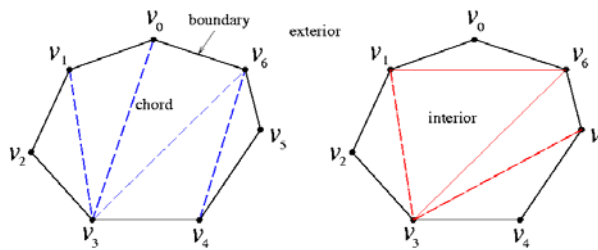
Optimal Polygon Triangulation

- Terminology: polygon, interior, exterior, boundary, convex polygon, triangulation?
- **The Optimal Polygon Triangulation Problem:** Given a convex polygon $P = \langle v_0, v_1, \dots, v_{n-1} \rangle$ and a weight function w defined on triangles, find a triangulation that minimizes $\sum_{\Delta} w(\Delta)$.

- One possible weight function on triangle:

$$w(\Delta v_i v_j v_k) = |v_i v_j| + |v_j v_k| + |v_k v_i|,$$

where $|v_i v_j|$ is the Euclidean distance from v_i to v_j .



Every triangulation of an n -vertex convex polygon has $n-3$ chords and divides it into $n-2$ triangles.

Unit 4

Y.-W. Chang

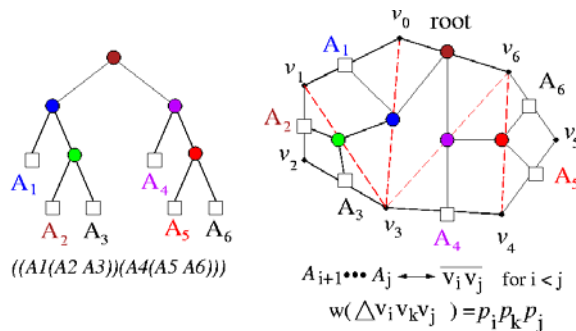
31

Optimal Polygon Triangulation (cont'd)

- Correspondence between **full parenthesization**, **full binary tree (parse tree)**, and **triangulation**
 - full parenthesization \leftrightarrow full binary tree
 - full binary tree ($n-1$ leaves) \leftrightarrow triangulation (n sides)

- $t[i, j]$: weight of an optimal triangulation of polygon $\langle v_{i-1}, v_i, \dots, v_j \rangle$.

$$t[i, j] = \begin{cases} 0 & \text{if } i = j, \\ \min_{i \leq k \leq j-1} \{t[i, k] + t[k+1, j] + w(\Delta v_{i-1} v_k v_j)\} & \text{if } i < j. \end{cases}$$



Unit 4

Y.-W. Chang

32

Pseudocode: Optimal Polygon Triangulation

- Matrix-Chain-Order is a special case of the optimal polygonal triangulation problem.
- Only need to modify Line 9 of Matrix-Chain-Order.
- Complexity: Runs in $\theta(n^3)$ time and uses $\theta(n^2)$ space.

```

Optimal-Polygon-Triangulation(P)
1.  $n \leftarrow \text{num}[P]$ ;
2. for  $i \leftarrow 1$  to  $n$ 
3.    $t[i, i] \leftarrow 0$ ;
4. for  $l \leftarrow 2$  to  $n$ 
5.   for  $i \leftarrow 1$  to  $n-l+1$ 
6.      $j \leftarrow i+l-1$ ;
7.      $t[i, j] \leftarrow \infty$ ;
8.     for  $k \leftarrow i$  to  $j-1$ 
9.        $q \leftarrow t[i, k] + t[k+1, j] + w(\Delta v_{i-1} v_k v_j)$ ;
10.      if  $q < t[i, j]$ 
11.         $t[i, j] \leftarrow q$ ;
12.         $s[i, j] \leftarrow k$ ;
13. return  $t$  and  $s$ 
  
```

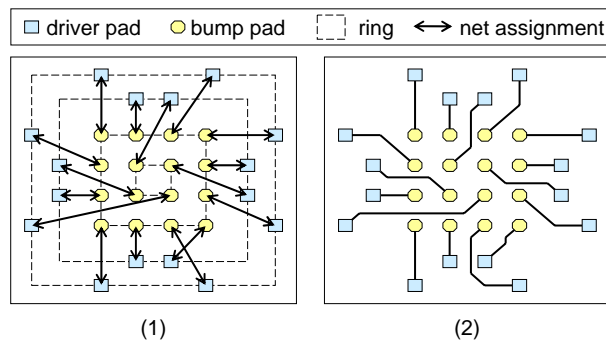
Unit 4

Y.-W. Chang

33

LCS for Flip-Chip Routing

- Lee, Lin, and Chang, ICCAD-09
- Given: A set of driver pads on driver pad rings, a set of bump pads on bump pad rings, a set of nets/connections
- Objective: Connect driver pads and bump pads according to a predefined netlist such that the total wirelength is minimized



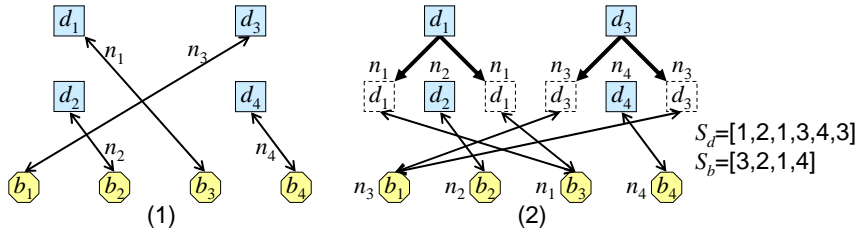
Unit 4

Y.-W. Chang

34

Minimize # of Detoured Nets by LCS

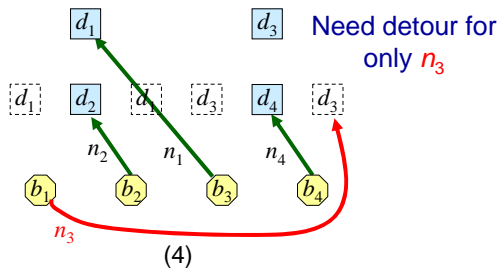
- Cut the rings into lines/segments (form linear orders)



net sequence

	S_d	1	2	1	3	4	3
S_b	0	0	0	0	0	0	0
3	0	0	0	0	1	1	1
2	0	0	1	1	1	1	1
1	0	1	1	2	2	2	2
4	0	1	1	2	2	3	3

(3)



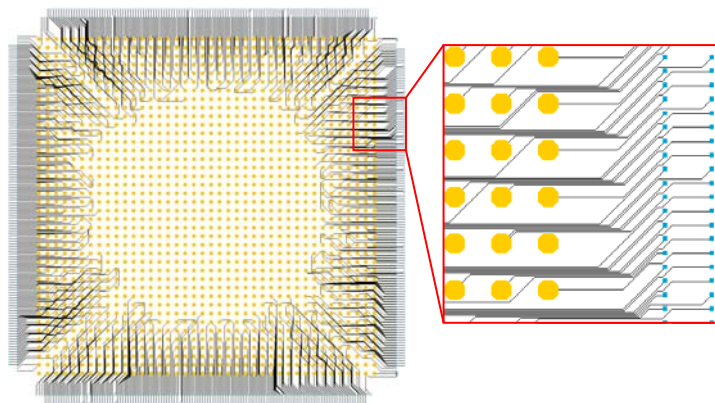
(4)

Unit 4

Y.-W. Chang

35

Flip-Chip Routing Example

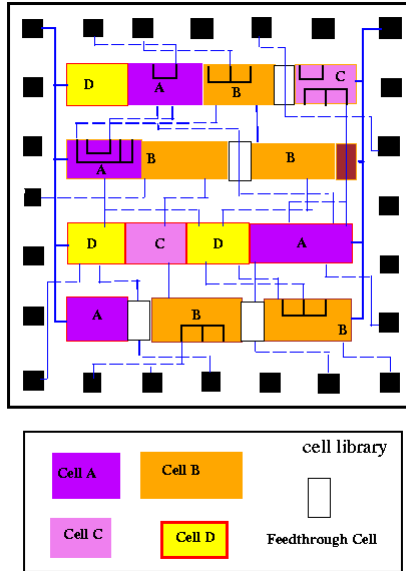


Unit 4

Y.-W. Chang

36

Standard-Cell Based VLSI Design Style

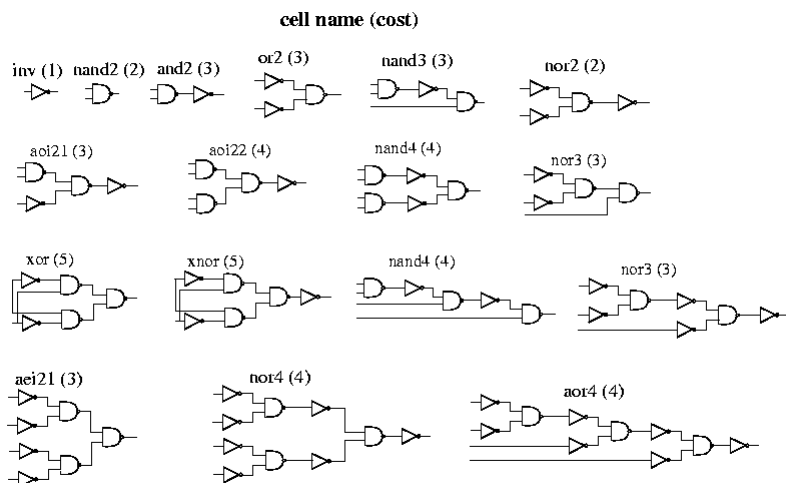


Unit 4

Y.-W. Chang

37

Pattern Graphs for an Example Library



Unit 4

Y.-W. Chang

38

Technology Mapping

- **Technology Mapping:** The optimization problem of finding a minimum cost covering of the subject graph by choosing from the collection of pattern graphs for all gates in the library.
- A **cover** is a collection of pattern graphs such that every node of the subject graph is contained in one (or more) of the pattern graphs.
- The cover is further constrained so that each input required by a pattern graph is actually an output of some other pattern graph.

Unit 4

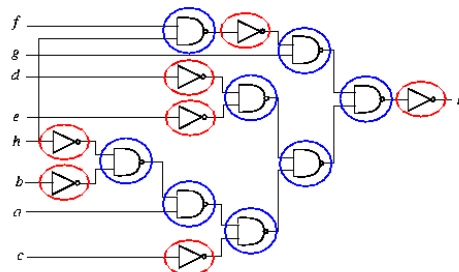
Y.-W. Chang

39

Trivial Covering

- Mapped into 2-input NANDs and 1-input inverters.
- 8 2-input NAND-gates and 7 inverters for an area cost of 23.
- Best covering?

$$\begin{aligned}t1 &= d + e; \\t2 &= b + h; \\t3 &= a \ t2 + c; \\t4 &= t1 \ t3 + f g h;\end{aligned}$$



an example subject graph

Unit 4

Y.-W. Chang

40

Optimal Tree Covering by Dynamic Programming

- If the subject directed acyclic graph (DAG) is a tree, then a polynomial-time algorithm to find the minimum cover exists.
 - Based on dynamic programming: optimal substructure? overlapping subproblems?
- Given: subject trees (networks to be mapped), library cells
- Consider a node n of the subject tree
 - Recursive assumption: For all children of n , a best match which implements the node is known.
 - Cost of a leaf is 0.
 - Consider each pattern tree which matches at n , compute cost as the cost of implementing each node which the pattern requires as an input plus the cost of the pattern.
 - Choose the lowest-cost matching pattern to implement n .

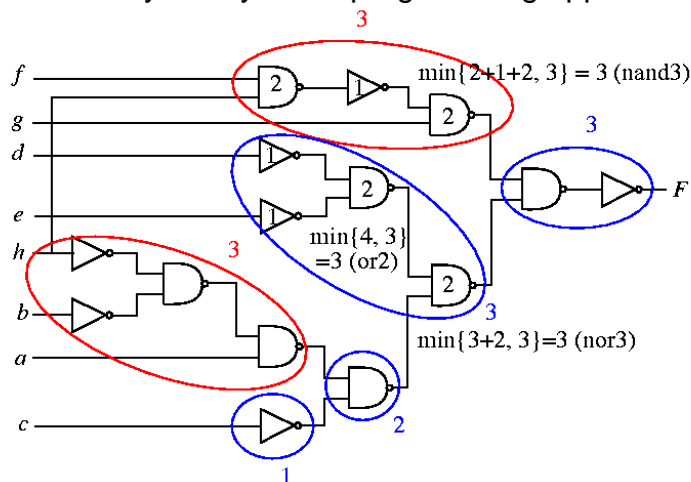
Unit 4

Y.-W. Chang

41

Best Covering

- A best covering with an area of 15.
- Obtained by the dynamic programming approach.



Unit 4

Y.-W. Chang

42