



南京大學

NANJING UNIVERSITY

Introduction to

# *Algorithm Design and Analysis*

[15] Path in Graph



*Yu Huang*

<http://cs.nju.edu.cn/yuhuang>  
Institute of Computer Software  
Nanjing University



# In the last class...

- **Optimization Problem**
  - Greedy strategy
- **MST Problem**
  - Prim's Algorithm
  - Kruskal's Algorithm
- **Single-Source Shortest Path Problem**
  - Dijkstra's Algorithm



# Path in Graphs

- **Single-source shortest path**
  - Dijkstra's algorithm by example
  - Priority queue-based implementation
  - Proof of correctness
- **All-pair shortest path**
  - Shortest Path and Transitive Closure
  - Washall's Algorithm for Transitive Closure
  - All-Pair Shortest Paths
  - Matrix for Transitive Closure
  - Multiplying Bit Matrices - Kronrod's Algorithm



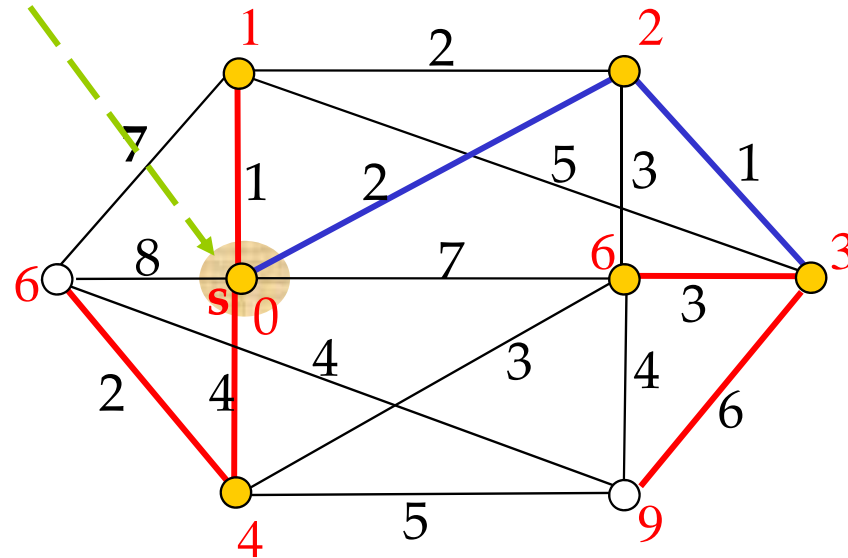
# Single Source Shortest Paths

The single source

Red labels on each vertex is the length of the shortest path from  $s$  to the vertex.

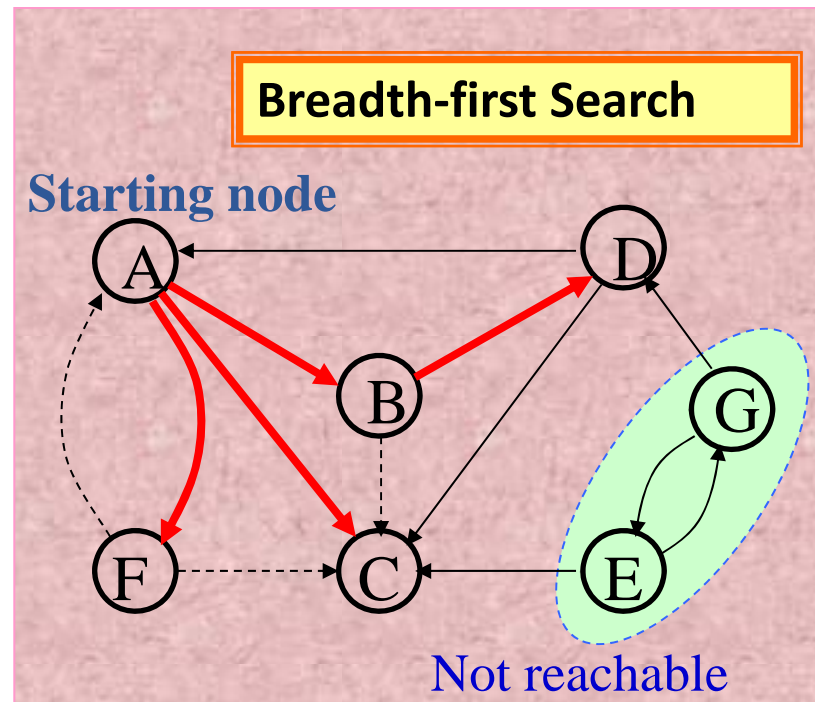
Note:

The shortest  $[0, 3]$ -path doesn't contain the shortest edge leaving  $s$ , the edge  $[0, 1]$

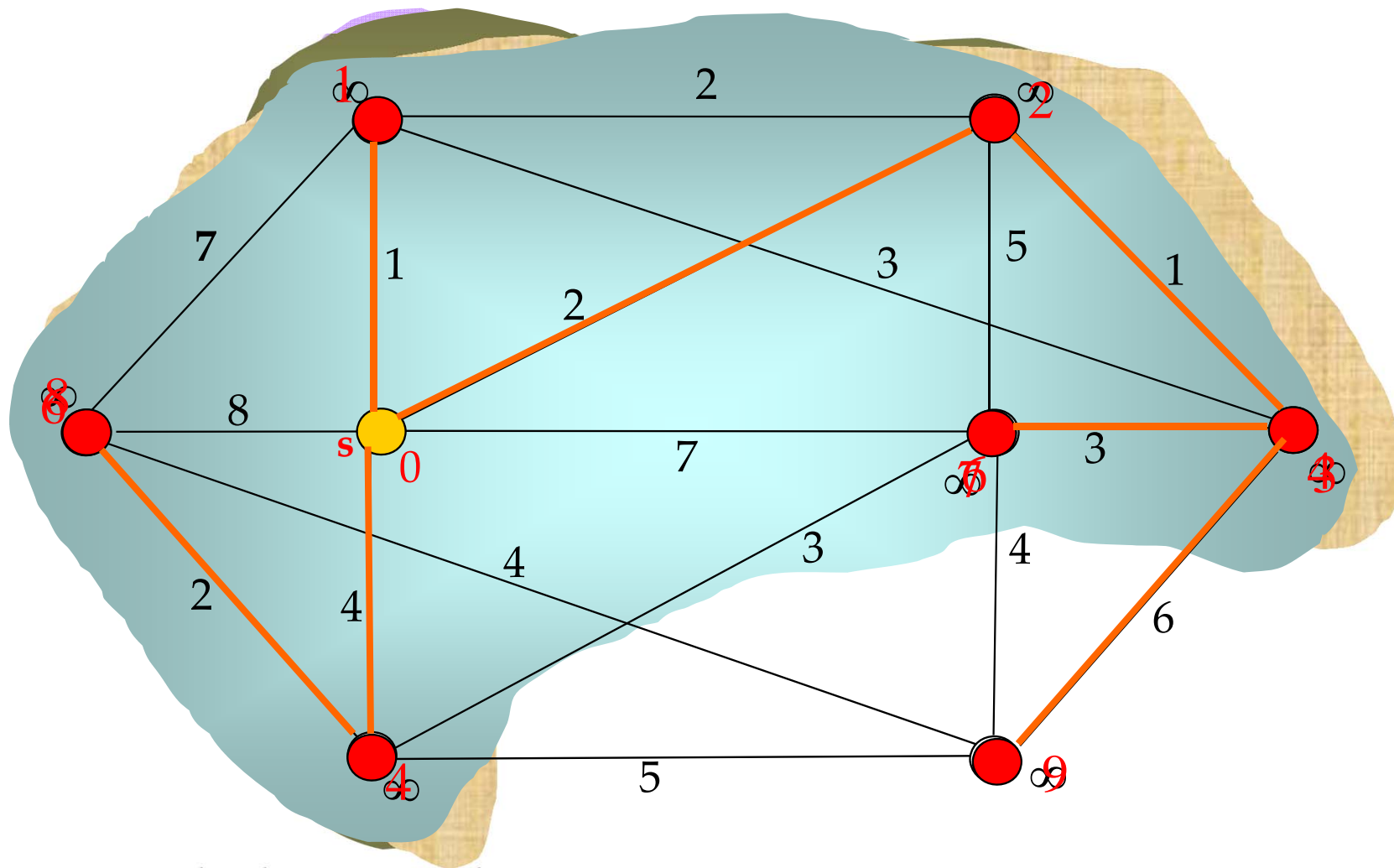


# Warm Up

- Single-source shortest path over **uniformly weighted graph**
  - Just BFS



# Dijkstra's Algorithm



# Priority Queue-based Implementation

## Shortest Paths

```
Void shortestPaths(EdgeList[] adjInfo, int n, int s,  
int[] parent, float[] fringeWgt)
```

```
int[] status = new int[n+1];  
MinPQ pq = create(n, status, parent, fringeWgt);
```

```
insert(pq, s, -1, 0);  
while(!isEmpty(pq))  
    int v = getMin(pq);  
    deleteMin(pq);  
    updateFringe(pq, adjInfo[v], v);
```



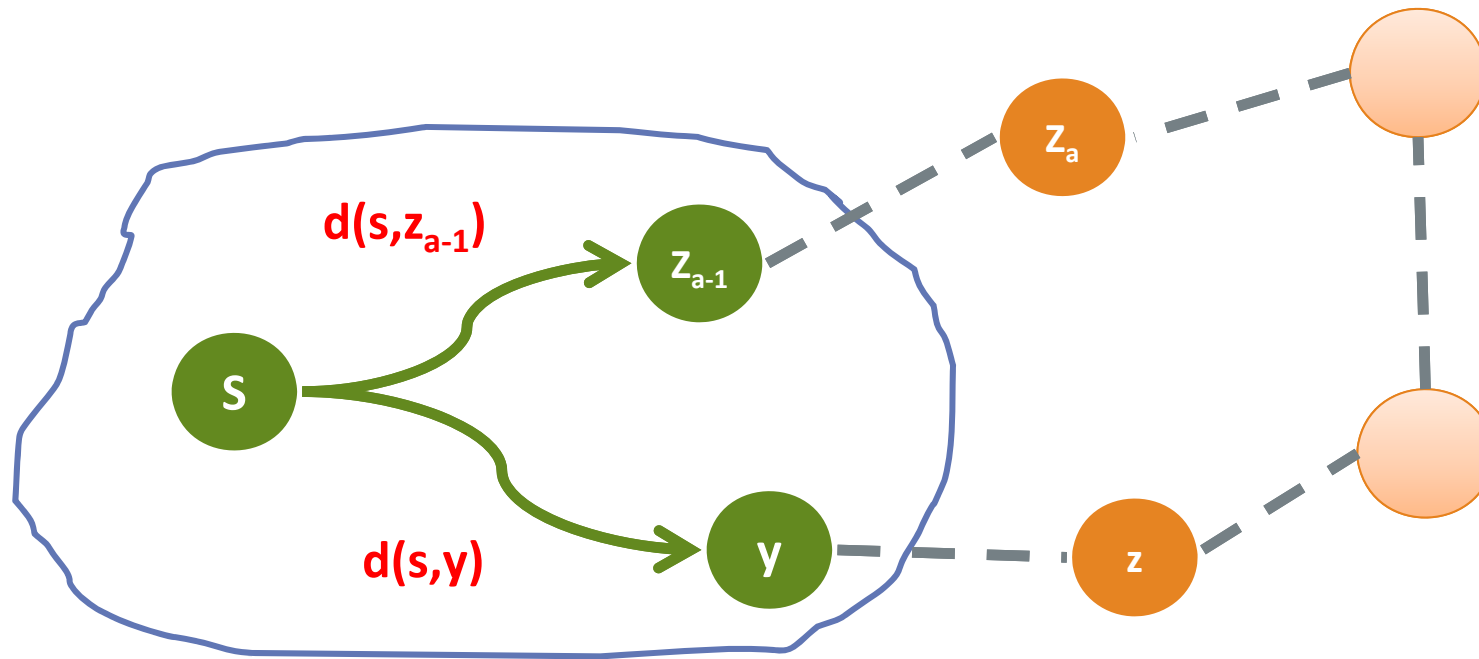
```
void updateFringe(MinPQ pq, EdgeList  
adjInfoOfv, int v)
```

```
float myDist = pq.fringeWgt[v];  
EdgeList remAdj;  
remAdj = adjInfoOfv;  
while(remAdj != nil)  
    EdgeInfo wInfo = first(remAdj);  
    int w = wInfo.to;  
    float newDist = myDist + wInfo.weight;  
    if(pq.status[w] == unseen)  
        insert(pq, w, v, newDist);  
    else if(pq.status[w] == fringe)  
        if(newDist < getPriority(pq, w))  
            decreaseKey(pq, w, v, newDist);  
    remAdj = rest(remAdj);  
return;
```



# Correctness of the Dijkstra's Algorithm

- $W(s \rightarrow y \rightarrow z) < W(s \rightarrow z_{a-1} \rightarrow z_a \rightarrow z)$





# From Algorithm to Skeleton

- Single-source shortest path (SSSP)

- SSSP + node weight constraint

- E.g. in routing

- Each router has its cost (node cost)
    - Each route has its cost (edge cost)

- SSSP + capacity constraint

- The “pipe problem”

- The “electric vehicle problem”

- Give capacity, whether there exists a path?
    - Give the path, what is the min capacity?

**“Dijkstra’s  
Skeleton”**



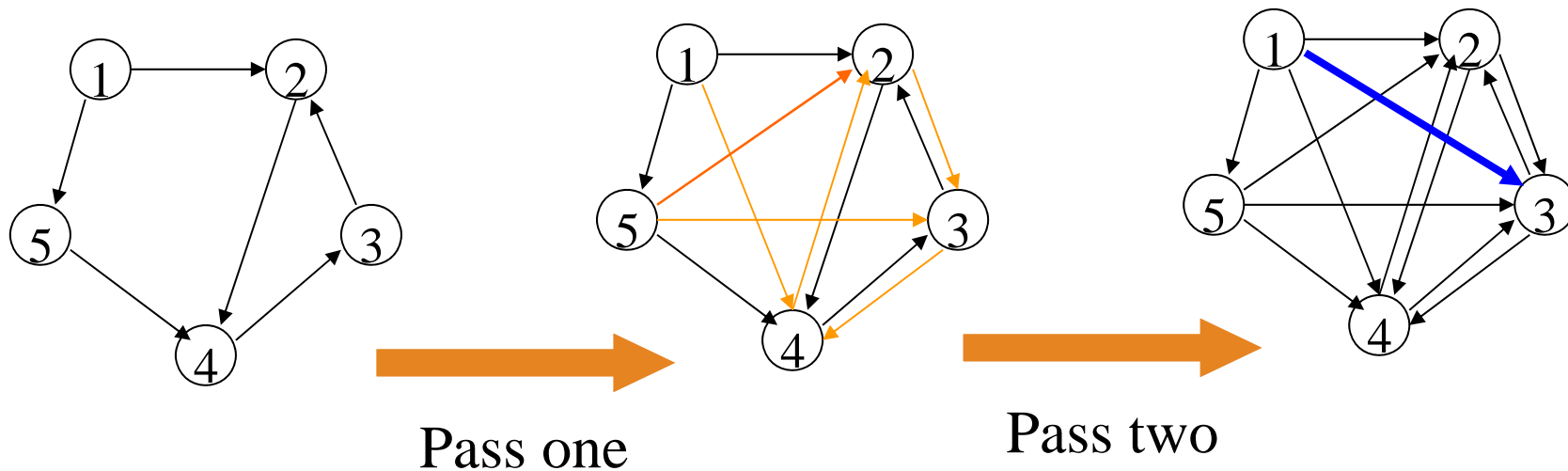
# Fundamental Questions

- For **all** pair of vertices in a graph, say,  $u, v$ :
  - Is there a path from  $u$  to  $v$ ?
  - What is the **shortest** path from  $u$  to  $v$ ?
- Reachability as a (reflexive) **transitive closure** of the adjacency relation
  - which can be represented as a bit matrix



# Transitive Closure by Shortcuts

- The idea: if there are edges  $s_i s_k$ ,  $s_k s_j$ , then an edge  $s_i s_j$ , the “shortcut” is inserted.



# Shortcut Algorithm

- **Input:**  $A$ , an  $n \times n$  boolean matrix that represents a binary relation
- **Output:**  $R$ , the boolean matrix for the transitive closure of  $A$
- **Procedure**
  - `void simpleTransitiveClosure(boolean[ ][ ] A, int n, boolean[ ][ ] R)`
  - `int i,j,k;`
  - *Copy  $A$  to  $R$ ;*
  - *Set all main diagonal entries,  $r_{ii}$ , to *true*;*
  - *while (any entry of  $R$  changed during one complete pass)*
  - *for ( $i=1; i \leq n; i++$ )*
  - *for ( $j=1; j \leq n; j++$ )*
  - *for ( $k=1; k \leq n; k++$ )*
  - $$r_{ij} = r_{ij} \vee (r_{ik} \wedge r_{kj})$$

**$O(n^4)$**

← The order of  
( $i,j,k$ ) matters



# Another Way to Add Shortcuts

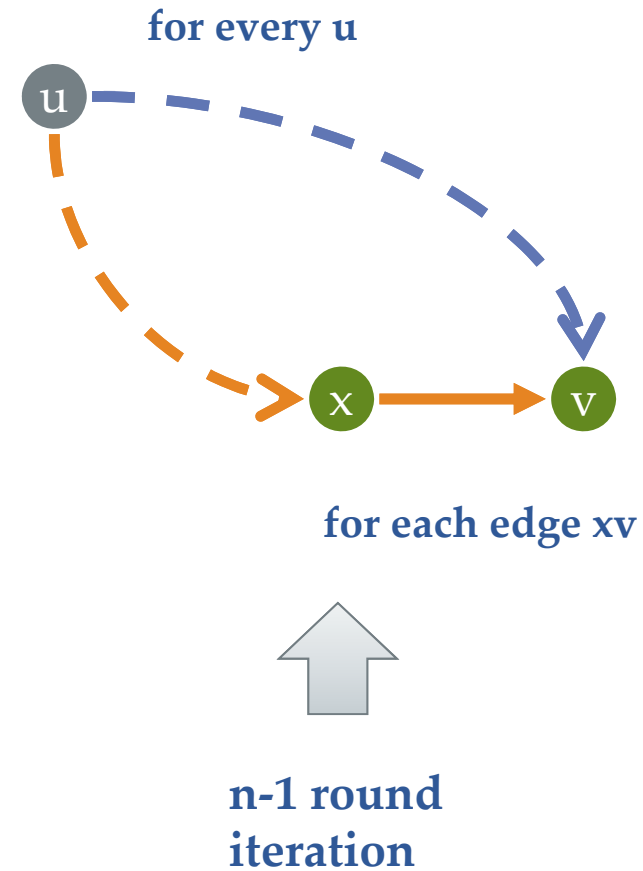
- Enumerate all edges  $(x,v)$ 
  - $v$  as the destination
  - Enumerate all possible sources  $u$

While any entry of  $R$  changed  
for all vertices  $u$

for every edge  $(x,v)$

$$r_{uv} = r_{uv} \vee (r_{ux} \wedge r_{xv})$$

**$O(n^2m)$**



# Length of the Path

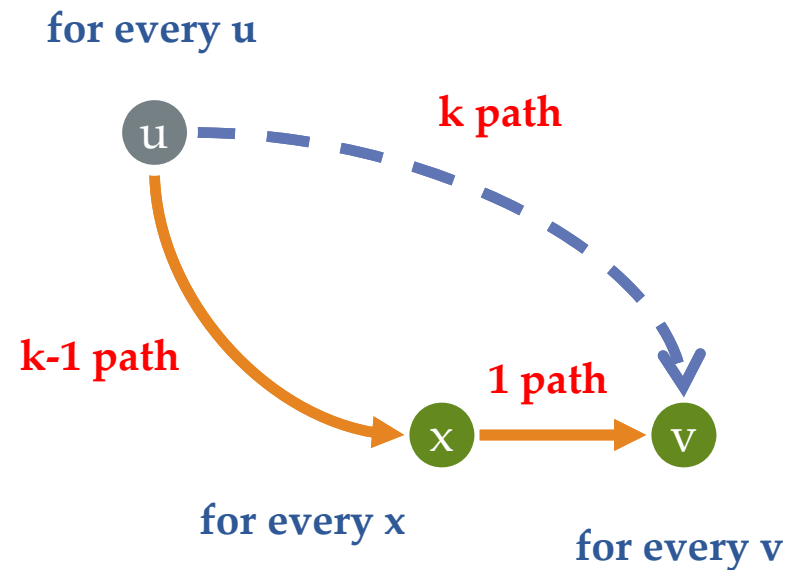
- **Recursion**
  - Reachable via at most k edges
- **Enumeration**
  - Enumerate all path length
  - Enumerate all sources and destinations

for k=1 to n-1  
 for all vertices u  
 for all vertices v  
 for all vertices x pointing to x

**$O(n^4)$**

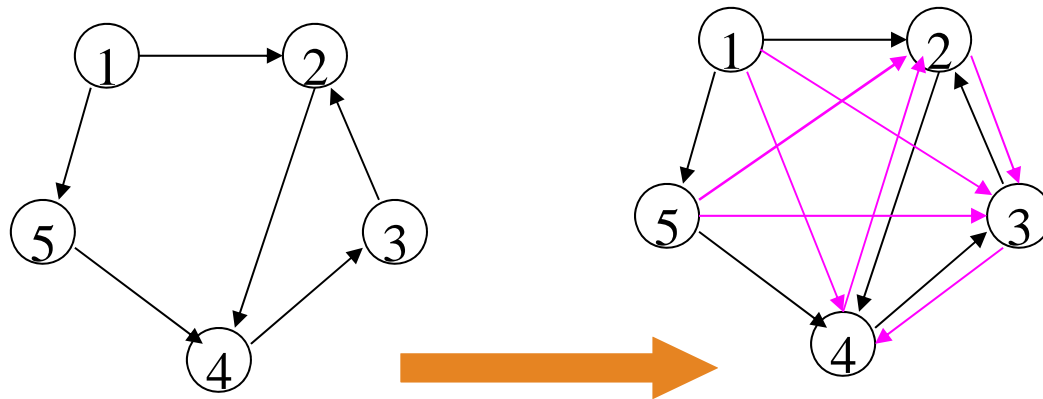
$$r_{uv}^k = r_{uv}^{k-1} \vee (r_{ux}^{k-1} \wedge r_{xv})$$

$r_{xv} = 1$



# Shortcuts in Different Order

- Duplicated checking may be deleted by changing the order of the vertices.



Pass one

Check the vertices in decreasing order.

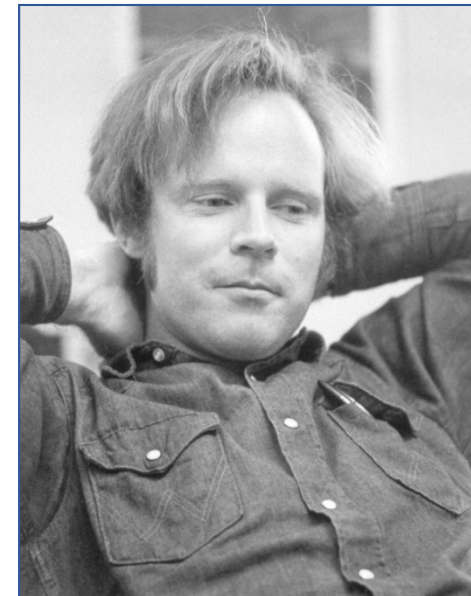
No edge is added in Pass two. End.

# Floyd's Lemma

组合问题的优良算法具有巨大回报，这个事实激励了技术水平的突飞猛进。... 大约从1970年起，计算机科学家们经历了所谓的‘**Floyd引理**’现象：看似需用 $n^3$ 次运算的问题实际上可能用 $O(n^2)$ 次运算就能求解，看似需用 $n^2$ 次运算的问题实际上可能用 $O(n \log n)$ 次运算就能处理，而且 $n \log n$ 通常还可以减少到 $O(n)$ 。一些更难的问题的运行时间也从 $O(2^n)$ 减少到 $O(1.5^n)$ ，再减少到 $O(1.3^n)$ ，等等。

- Knuth, Volumn4A, TAOCP

Robert W Floyd, In Memoriam  
by Donald E. Knuth, Stanford University





# Change the Order: Floyd-Warshall's Algorithm

- **void** simpleTransitiveClosure(**boolean**[][] A, **int**  $n$ , **boolean**[][]  $R$ )
- **int**  $i, j, k$ ;
- Copy  $A$  to  $R$ ;
- Set all main diagonal entries,  $r_{ii}$ , to *true*;
- ~~**while** (any entry of  $R$  changed during one complete pass)~~
- **for** ( $k=1$ ;  $k \leq n$ ;  $k++$ )
- **for** ( $i=1$ ;  $i \leq n$ ;  $i++$ )
- **for** ( $j=1$ ;  $j \leq n$ ;  $j++$ )
- $r_{ij} = r_{ij} \vee (r_{ik} \wedge r_{kj})$

$k$  varies in the  
outmost loop

Note: “false to true” can  
not be reversed



# Why the Floyd-Warshall Algorithm Works

- $\langle k, i, j \rangle$  or  $\langle i, j, k \rangle$ 
  - The order matters
  - *That's why Dijkstra fails*



算法青年

2014-8-4 23:18 来自 微博 weibo.com

算法没学好还真看不懂的笑话啊，赞~ //@ant\_hengxin: 信息量很大啊

@geelaw ★

再发一次我创造的一个笑话。问：为什么 Dijkstra 没有提出 Floyd 算法？答：因为他是 ijk 而不是 kij。

2014-8-3 22:58 来自 微博Win8客户端

120 | 12 | 7

阅读 5230 推广

5

评论

5



# Correctness of Warshall's Algorithm

- **Notation:**

- The value of  $r_{ij}$  changes during the execution of the body of the “**for**  $k...$ ” loop

- After initializations:  $r_{ij}^{(0)}$

- After the  $k^{\text{th}}$  time of execution:  $r_{ij}^{(k)}$



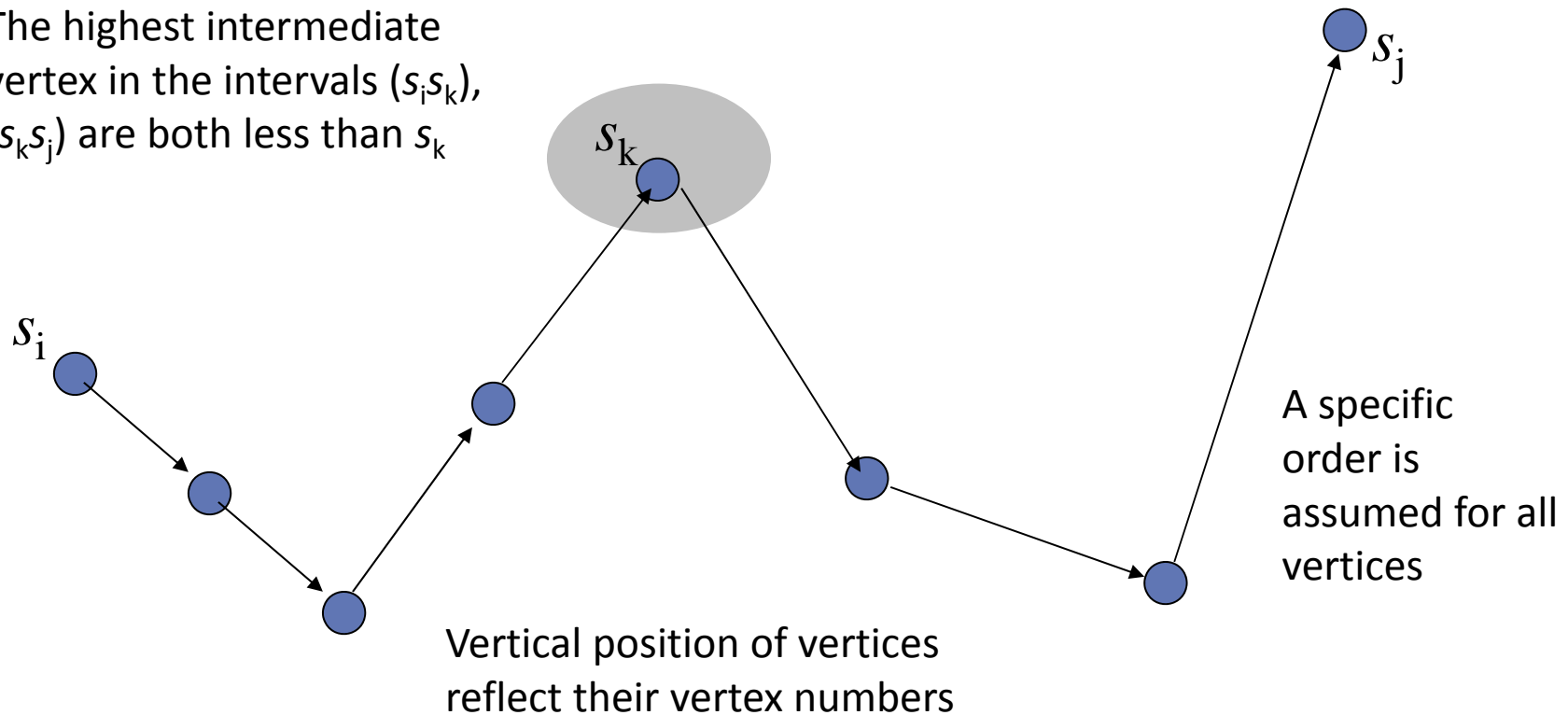
# Correctness of Warshall's Algorithm

- If there is a simple path from  $s_i$  to  $s_j$  ( $i \neq j$ ) for which the highest-numbered intermediate vertex is  $s_k$ , then  $r_{ij}^{(k)} = \text{true}$ .
- Proof by induction:
  - Base case:  $r_{ij}^{(0)} = \text{true}$  if and only if  $s_i s_j \in E$
  - Hypothesis: the conclusion holds for  $h < k$  ( $h \geq 0$ )
  - Induction: the simple  $s_i s_j$ -path can be looked as  $s_i s_k$ -path +  $s_k s_j$ -path, with the indices  $h_1, h_2$  of the highest-numbered intermediate vertices of both segment **strictly (simple path)** less than  $k$ . So,  $r_{ik}^{(h_1)} = \text{true}$ ,  $r_{kj}^{(h_2)} = \text{true}$ , then  $r_{ik}^{(k-1)} = \text{true}$ ,  $r_{kj}^{(k-1)} = \text{true}$  (Remember, false to true can not be reversed). So,  $r_{ij}^{(k)} = \text{true}$



# Highest-numbered Intermediate Vertex

The highest intermediate vertex in the intervals  $(s_i s_k)$ ,  $(s_k s_j)$  are both less than  $s_k$



$$r_{ij} = r_{ij} \vee (r_{ik} \wedge r_{kj})$$



# Correctness of Warshall's Algorithm

- If there is **no** path from  $s_i$  to  $s_j$ , then  $r_{ij}=false$ .
  - i.e., if  $r_{ij}=true$ , then there is a  $(s_i, s_j)$  path
- **Proof**
  - If  $r_{ij}=true$ , then only two cases:
    - $r_{ij}$  is set by initialization, then  $s_i s_j \in E$
    - Otherwise,  $r_{ij}$  is set during the  $k^{th}$  execution of (for  $k=1,2,\dots$ ) when  $r_{ik}^{(k-1)}=true$ ,  $r_{kj}^{(k-1)}=true$ , which, recursively, leading to the conclusion of the existence of a  $s_i s_j$ -path.  
(Note: if a  $s_i s_j$ -path exists, there exists a simple  $s_i s_j$ -path)



# All-pairs Shortest Path

- **Shortest path property**
  - If a shortest path from  $x$  to  $z$  consisting of path  $P$  from  $x$  to  $y$  followed by path  $Q$  from  $y$  to  $z$ . Then  $P$  is a shortest  $xy$ -path, and  $Q$ , a shortest  $yz$ -path.
- **The regular matrix representing a graph can easily be transformed into a (minimum) distance matrix  $D$** 

(just replacing 1 by edge weight, 0 by infinity, and setting main diagonal elements as 0)



# Computing the Distance Matrix

- **Basic formula:**

- $D^{(0)}[i][j] = w_{ij}$
- $D^{(k)}[i][j] = \min(D^{(k-1)}[i][j], D^{(k-1)}[i][k] + D^{(k-1)}[k][j])$

- **Basic property:**

- $D^{(k)}[i][j] \leq d_{ij}^{(k)}$

where  $d_{ij}^{(k)}$  is the weight of a shortest path from  $v_i$  to  $v_j$  with highest numbered intermediate vertex  $v_k$ .





# All-Pairs Shortest Paths

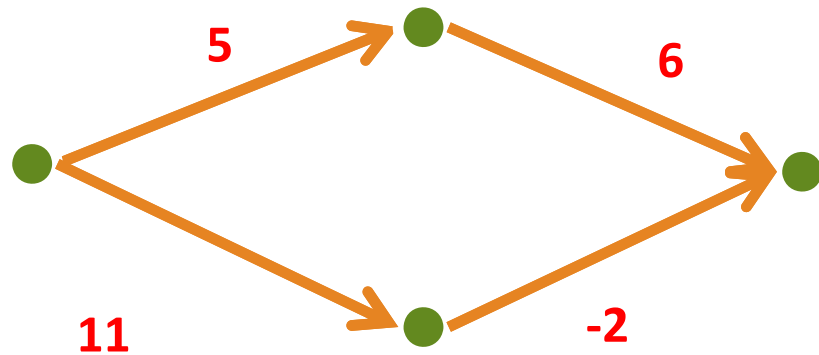
- **Floyd algorithm**
  - Only slight changes on Washall's algorithm.

```
Void allPairsShortestPaths(float [][] W, int n, float [][] D)  
    int i, j, k;  
    Copy W into D;  
    for (k=1; k≤n; k++)  
        for (i=1; i≤n; i++)  
            for (j=1; j≤n; j++)  
                D[i][j] = min (D[i][j], D[i][k]+D[k][j]);
```

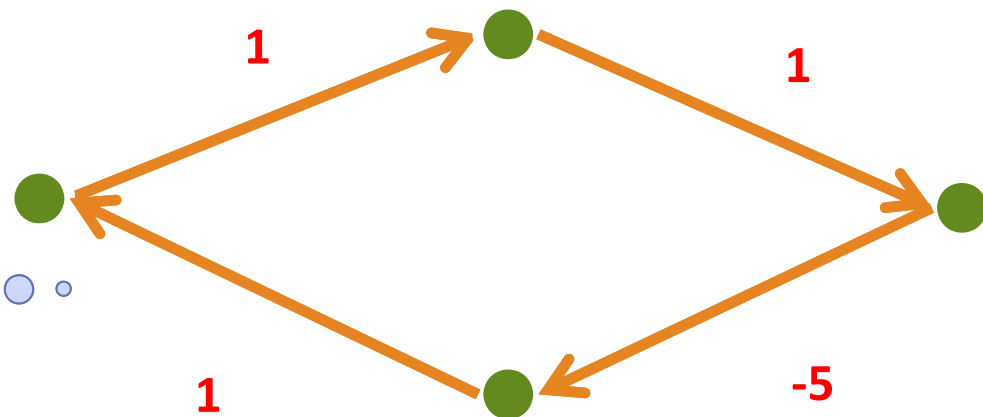
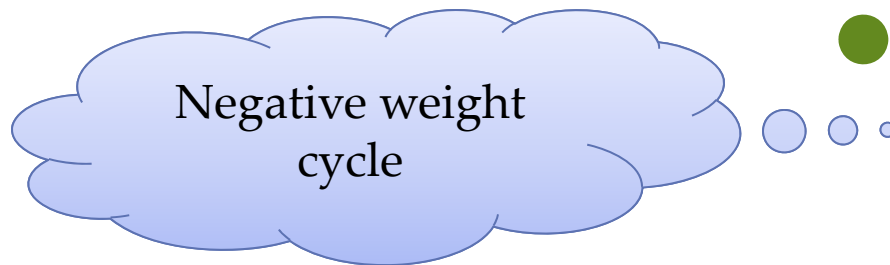
- **Routing table tracking the path**



# Negative Weight



“shortest path” is not well defined

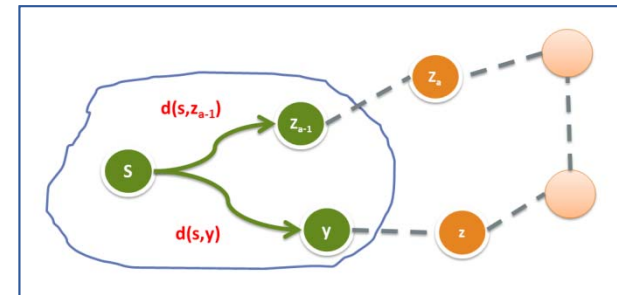


# Negative Weight

- Can the shortest path algorithm work correctly?

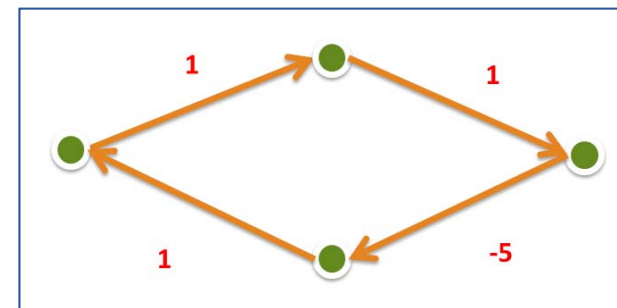
- Dijkstra's algorithm

- No negative weight edge



- Floyd's algorithm

- No negative weight cycle



# Matrix Representation

- Define family of matrix  $A^{(p)}$ :
  - $a_{ij}^{(p)} = \text{true}$  if and only if there is a path of length  $p$  from  $s_i$  to  $s_j$ .
- $A^{(0)}$  is specified as identity matrix.  $A^{(1)}$  is exactly the adjacency matrix.
- Note that  $a_{ij}^{(2)} = \text{true}$  if and only if exists some  $s_k$ , such that both  $a_{ik}^{(1)}$  and  $a_{kj}^{(1)}$  are *true*. So,  $a_{ij}^{(2)} = \bigvee_{k=1,2,\dots,n} (a_{ik}^{(1)} \wedge a_{kj}^{(1)})$ , which is an entry in the *Boolean matrix product*.



# Boolean Matrix Operations

- **Boolean matrix product  $C=AB$  as:**
  - $c_{ij} = \bigvee_{k=1,2,\dots,n} (a_{ik} \wedge b_{kj})$
- **Boolean matrix sum  $D=A+B$  as:**
  - $d_{ij} = a_{ij} \vee b_{ij}$
- **$R$ , the transitive closure matrix of  $A$ , is the sum of all  $A^p$ ,  $p$  is a non-negative integer.**
- **For a digraph with  $n$  vertices, the length of the longest simple path is no larger than  $n-1$ .**



# Bit Matrix

- A **bit string** of length  $n$  is a sequence of  $n$  bits occupying contiguous storage(word boundary) (usually,  $n$  is larger than the word length of a computer)
- If  $A$  is a **bit matrix** of  $n \times n$ , then  $A[i]$  denotes the  $i$ th row of  $A$  which is a bit string of length  $n$ .  $a_{ij}$  is the  $j$ th bit of  $A[i]$ .
- The **procedure bitwiseOR(a,b,n)** compute  $a \vee b$  bitwise for  $n$  bits, leaving the result in  $a$ .



# Straightforward Multiplication of Bit Matrix

- Computing  $C=AB$

- <Initialize  $C$  to the zero matrix>
- **for** ( $i=1; i \leq n, i++$ )
- **for** ( $k=1; k \leq n, k++$ )
- **if** ( $a_{ik} == \text{true}$ ) **bitwiseOR**( $C[i], B[k], n$ )

Thought as a union of sets (row union),  $n^2$  unions are done at most

In the case of  $a_{ik}$  is *true*,  $c_{ij} = a_{ik} b_{kj}$  is true iff.  $b_{kj}$  is true. As a result:  $C[i] = \cup_{k \in A[i]} B[k]$ ,  
( $A[i] = \{k / a_{ik} = \text{true}\}$ )

Union for  $B[k]$  is **repeated each time** when the  $k$ th bit is *true* in a different row of  $A$  is encountered.

# Reducing the Duplicates by Grouping

- Multiplication of  $A, B$ , two  $12 \times 12$  matrices

$A_1$	1	0	1	1	0	1	0	1	0	0	0	1
$A_2$												
$A_3$	1	0	1	1	1	0	0	1	1	0	1	1
$A_4$												
$A_5$												
$A_6$					0	1	0	1				
$A_7$	1	0	1	1	1	0	0	1	1	1	1	0
$A_8$												
$A_9$												
$A_{10}$												
$A_{11}$												
$A_{12}$												

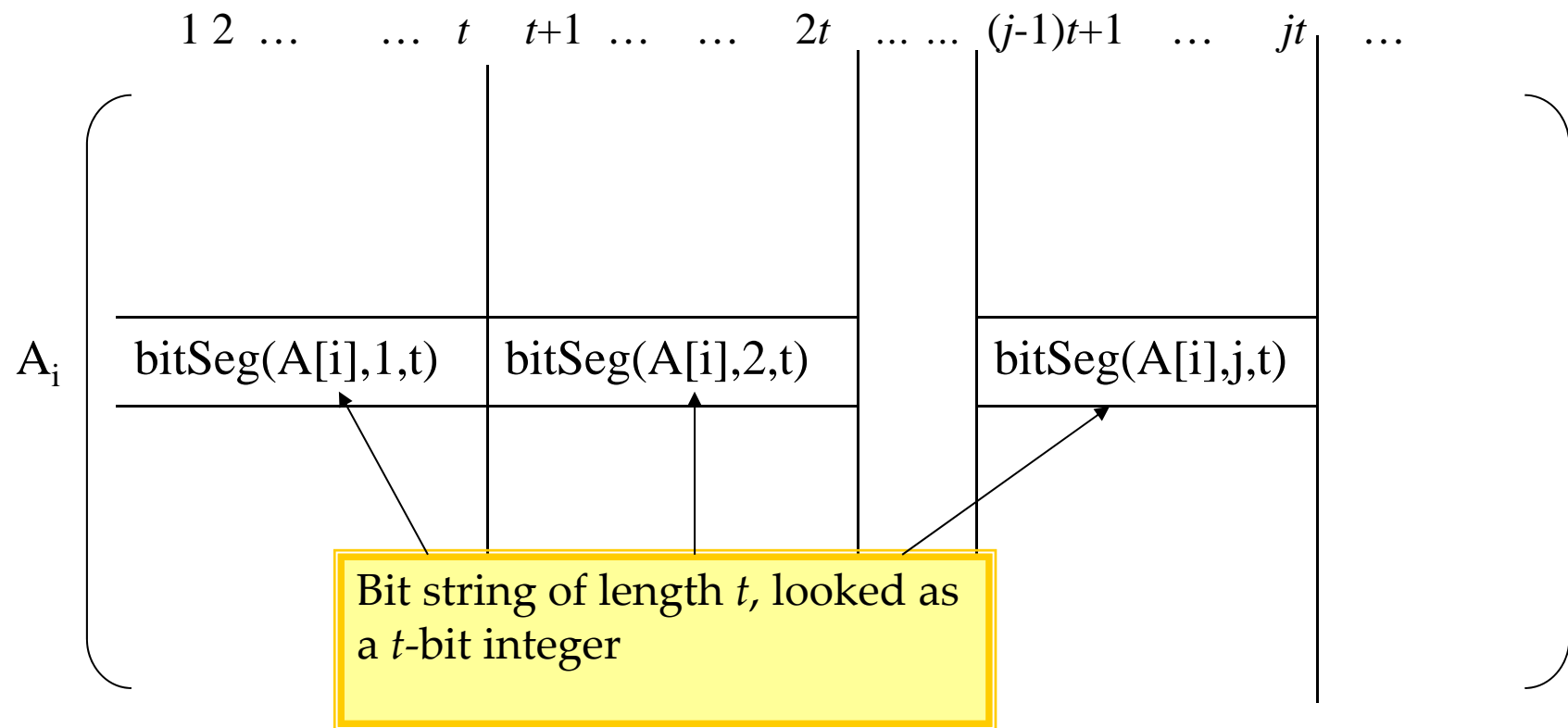
Segment  
Length  $t$

- 12 rows of  $B$  are divided evenly into 3 groups, with rows 1-4 in group 1, etc.
- With each group, all possible unions of different rows are pre-computed. (This can be done with 11 unions if suitable order is assumed.)
- When the first row of  $AB$  is computed,  $(B[1] \cup B[3] \cup B[4])$  is used in stead of 3 different unions, and this combination is used in computing the 3<sup>rd</sup> and 7<sup>th</sup> rows as well.



# The Segmentation for Matrix A

The  $n \times n$  array

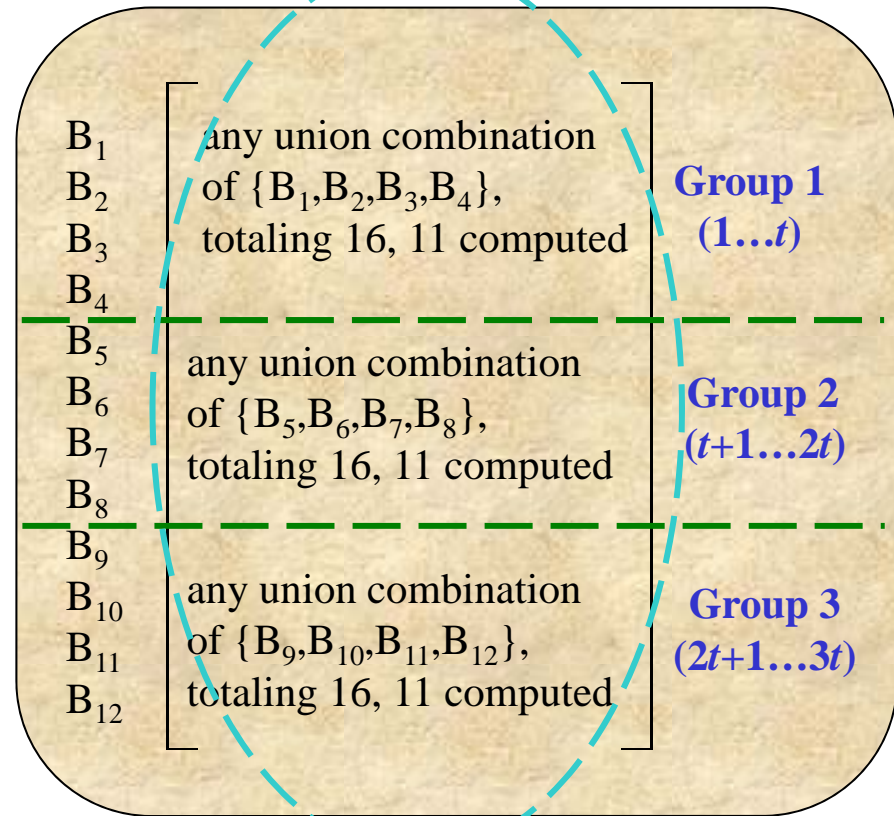


# An Example

	Group 1 (1...t)				Group 2 (t+1...2t)				Group 3 (2t+1...3t)			
A <sub>1</sub>	1	0	1	1	0	1	0	1	0	0	0	1
A <sub>2</sub>	1	0	0	0	1	0	1	1	0	0	1	0
A <sub>3</sub>	1	0	1	1	1	0	0	1	1	0	1	1
A <sub>4</sub>	0	1	1	0	0	0	1	0	1	0	1	0
A <sub>5</sub>	0	1	0	0	1	1	0	1	0	1	0	1
A <sub>6</sub>	1	1	0	1	0	1	0	1	1	0	1	0
A <sub>7</sub>	1	0	1	1	1	0	0	1	1	1	1	0
A <sub>8</sub>	1	1	1	1	0	0	1	1	0	1	1	0
A <sub>9</sub>	0	1	1	0	1	0	1	0	1	1	1	0
A <sub>10</sub>	1	0	0	0	1	0	1	1	0	0	1	1
A <sub>11</sub>	0	1	0	1	0	1	0	1	0	1	0	0
A <sub>12</sub>	1	0	0	1	0	0	1	0	1	0	0	0

t=4

bitSeg(A[7], 1, t)  
= 1011<sub>2</sub> = 11



Where to store?



# Storage of the Row Combinations

- **Using one large 2-dimensional array**
- **Goals**
  - keep all unions generated
  - provide indexing for using
- **Coding within a group**
  - One-to-one correspondence between a bit string of length  $t$  and one union for a subset of a set of  $t$  elements
- **Establishing indexing for union required**
  - When constructing a row of  $AB$ , a segment can be notated as a integer. Use it as index.



# Storage the Unions

allUnion

one row for  
one group

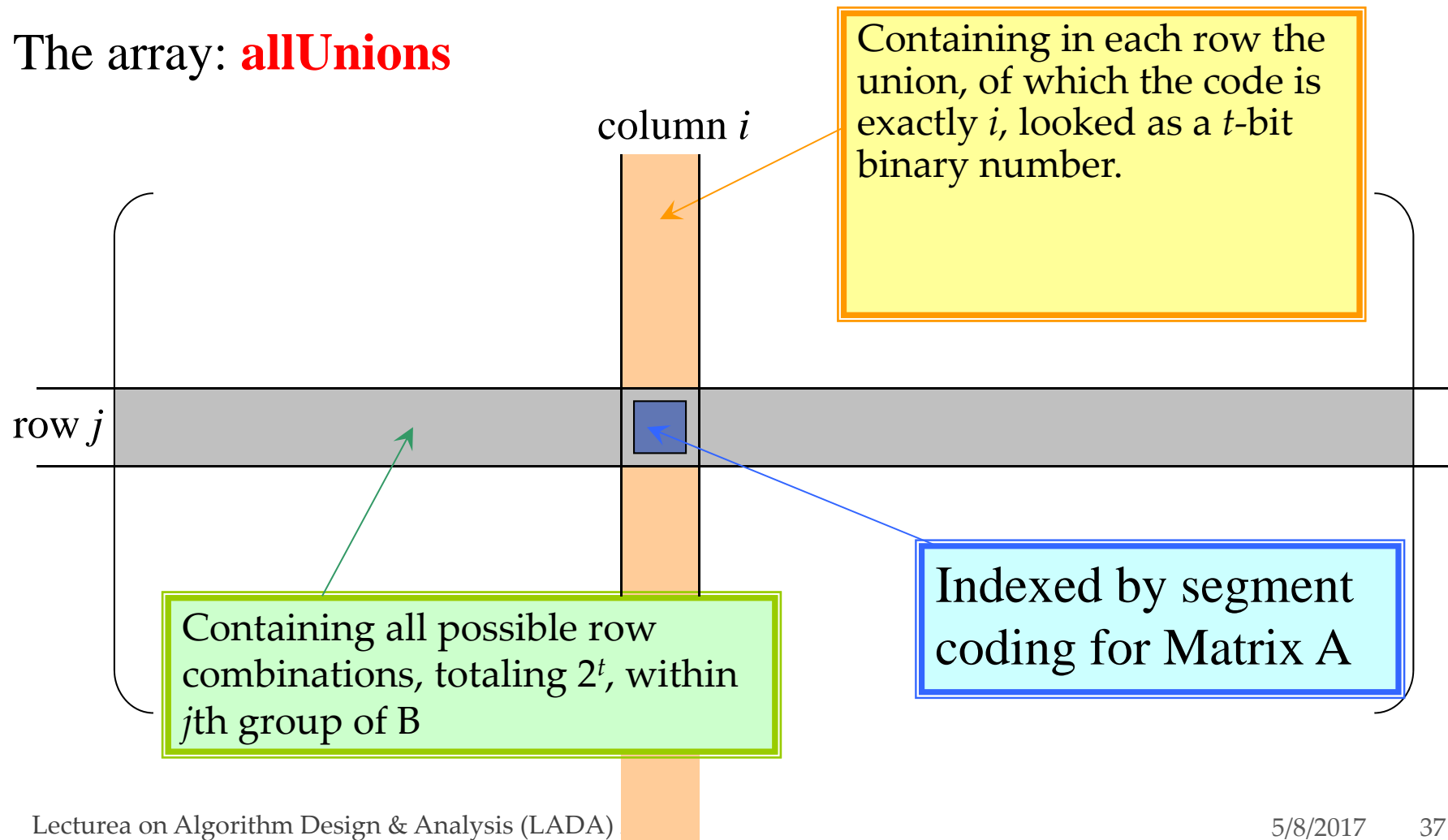
column indexed by  $\text{bitSeg}(A[i],j,t)$

$\phi$	4	3	3,4	2	2,4	2,3	2,3,4	1	1,4	1,3	1,3,4	1,2	1,2,4	1,2,3	1,2,3,4
$\phi$	8	7	7,8	6	6,8										
$\phi$	12	11	11,12	10	10,12										

$i,j,k$  stands for  $B_i \cup B_j \cup B_k$

# Array for Row Combinations

The array: **allUnions**



# Cost as Function of Group Size

- **Cost for the pre-computation**
  - There are  $2^t$  different combination of rows in one group, including an empty and  $t$  singleton. Note, in a suitable order, each combination can be made using only one union. So, the total number of union is  $g[2^t - (t+1)]$ , where  $g = n/t$  is the number of group.
- **Cost for the generation of the product**
  - In computing one of  $n$  rows of  $AB$ , at most one combination from each group is used. So, the total number of union is  $n(g-1)$



# Selecting Best Group Size

- The total number of union done is:

$$g[2^t - (t+1)] + n(g-1) \approx (n2^t)/t + n^2/t \text{ (Note: } g=n/t \text{ )}$$

- Trying to minimize the number of union
  - Assuming that the first term is of higher order:
    - Then  $t \geq \lg n$ , and the least value is reached when  $t = \lg n$ .
  - Assuming that the second term is of higher order:
    - Then  $t \leq \lg n$ , and the least value is reached when  $t = \lg n$ .
- So, when  $t \approx \lg n$ , the number of union is roughly  $2n^2/\lg n$ , which is of lower order than  $n^2$ . We use  $t = \lfloor \lg n \rfloor$

For symplicity, exact power for n is assumed



# Sketch for the Procedure

- $t = \lfloor \lg n \rfloor$ ;  $g = \lceil n/t \rceil$ ;
- **<Compute and store in **allUnions** unions of all combinations of rows of  $B$ >**
- for ( $i=1$ ;  $i \leq n$ ;  $i++$ )
- **<Initialize  $C[i]$  to 0>**
- for ( $j=1$ ;  $j \leq g$ ;  $j++$ )
- $C[i] = C[i] \cup \text{allUnions}[j][\text{bitSeg}(A[i], j, t)]$



# Kronrod Algorithm

- **Input:**  $A, B$  and  $n$ , where  $A$  and  $B$  are  $n \times n$  bit matrices.
- **Output:**  $C$ , the Boolean matrix product.
- **Procedure**
  - The processing order has been changed, from “row by row” to “group by group”, resulting the reduction of storage space for unions.



# Complexity of the Kronrod Algorithm

- For computing all unions within a group,  $2^t - 1$  union operations are done.
- One union is bitwiseOR'ed to  $n$  row of  $C$
- So, altogether,  $(n/t)(2^t - 1 + n)$  row unions are done.
- The cost of row union is  $\lceil n/w \rceil$  bitwise or operations, where  $w$  is word size of bitwise or instruction dependent constant.



*Thank you!*

*Q & A*

*Yu Huang*

<http://cs.nju.edu.cn/yuhuang>

