國立清華大學 電機工程學系
EE2410 Data Structure

Chapter 6
Graph (Part II)

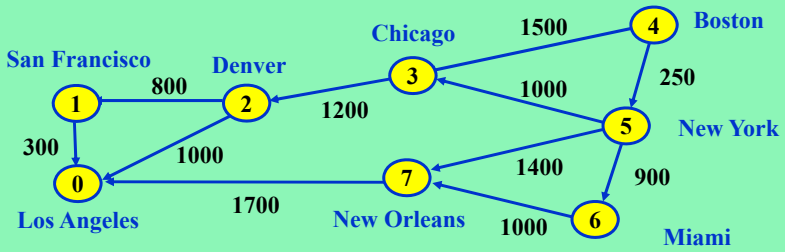
Outline

- **Shortest Path and Transitive Closure**
  - **Single Source / All Destinations**
  - **All-Pairs Shortest Paths**
  - **Transitive Closure**
- **Activity Network**
  - **Activity on Vertex (AOV) Networks**
  - **Activity on Edge (AOE) Networks**

Shortest Path Problem

- **Application**
  - Graph can be used to represent the highway structure
  - Vertices represent cities
  - Edges represent sections of highway
  - Edge weight is the distance of an edge
- **Questions**
  - (1) Is there a path from city A to city B?
  - (2) If there is more than one path from A to B, which is the shortest path?

Example: A Weighted Digraph



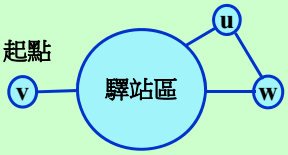
Length-adjacency matrix

	0	1	2	3	4	5	6	7
0	0							
1	300	0						
2	1000	800	0					
3			1200	0				
4				1500	0	250		
5				1000		0	900	1400
6							0	1000
7	1700							0

Edsger Dijkstra's Algorithm

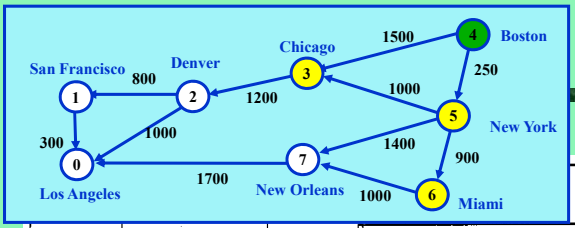
```
void  Graph::ShortestPath( const int n, const int v)
// dist[j], 0≦j<n, is set to the length of the shortest path from vertex v to vertex j
// in a graph G with n vertex and edge lengths given by length[i][j]
{
  for ( int  i=0; i<n; i++ ) {  s[i] = FALSE;  dist[i] = length[v][i]; }  // initialize
  s[v] = TRUE;
  dist[v] = 0;

   for ( i=0; i<n-2; i++) {
        int  u = choose(n);  // routine 'choose' returns a value u such that
                             // dist[u] = minimum dist[w], where s[w] = FALSE
        s[u] = TRUE;
        for ( int  w=0; w<n; w++) {
              if ( ! s[w] )
                 if ( dist[u] + length[u][w] < dist[w] )
                     dist[w] = dist[u] + length[u][w];
        }
   }
}
```



驛站區由近而遠, 從 0 個擴大成 n-1 個

Example

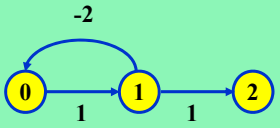


Source is Boston

Iteration	S	Vertex	Distance							
	en-route set	selected	LA [0]	SF [1]	DEN [2]	CHI [3]	BOST [4]	NY [5]	MIA [6]	NO [7]
Initial	--	----	+∞	+∞	+∞	1500	0	250	+∞	+∞
1	{4}	5	+∞	+∞	+∞	1250	0	250	1150	1650
2	{4,5}	6	+∞	+∞	+∞	1250	0	250	1150	1650
3	{4,5,6}	3	+∞	+∞	2450	1250	0	250	1150	1650
4	{4,5,6,3}	7	3350	+∞	2450	1250	0	250	1150	1650
5	{4,5,6,3,7}	2	3350	3250	2450	1250	0	250	1150	1650
6	{4,5,6,3,7,2}	1	3350	3250	2450	1250	0	250	1150	1650
	{4,5,6,3,7,2,1}									

Digraph With Negative Edges

- **When negative edge lengths are permitted**
  - The digraph should have no cycle of negative length
  - This is to ensure that the shortest path consists of a finite number of edges
- **Example**
  - The shortest path from vertex 0 to 2 is -∞
    (0, 1, 0, 1, 0, 1, ..., 0, 1, 2)
  - Because there is a cycle (1, 0, 1) of length -1



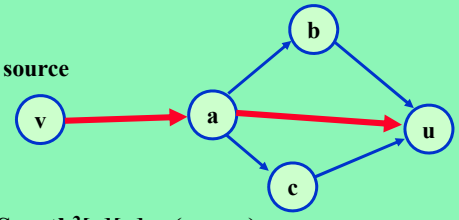
Possible Search Space

- **Let $dist^k[u]$**
  - be the length of a shortest path from the source vertex v to vertex u that contains at most *k* edges
- **Then, $dist^1[u] = length[v][u]$, $0 \leqq u < n$**
- **Under the no-negative-cycle constraint**
  - We can limit our search to shortest paths with at most n-1 edges
  - Hence, $dist^{n-1}[u]$ is the length of an unrestricted shortest path from v to u

Recurrence Relation

- Given $dist^m[u]$ for every $0 \leqq u < n$
- How to compute $dist^{m+1}[u]$?
- Recurrence Relation

$$dist^{m+1}[u] = \min\{ dist^m[u], \min_i \{dist^m[i] + length[i][u]\} \}$$



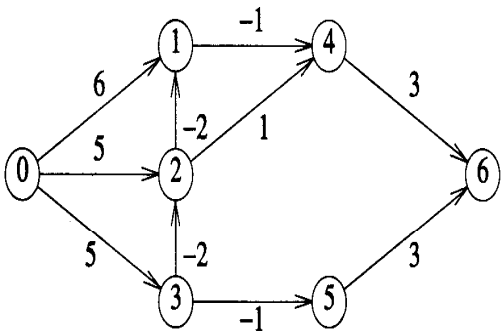
Assume that $S\text{-}path^2[v][u] = (v, a, u)$

What are the possible $S\text{-}path^3[v][u] = \{ (v,a,u), (v,a,b,u), (v,a,c,u)\}$

Example of Shortest Paths With Negative Edge Lengths

Source vertex: 0

Each shortest path consists of at most 6 edges



(a) A directed graph

	$dist^k[7]$						
k	0	1	2	3	4	5	6
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

(b) $dist^k$

Bellman and Ford Algorithm For computing Shortest Paths

```
1 void Graph::BellmanFord(const int n, const int v)
2 // Single source all destination shortest paths with negative edge lengths
3 {
4   for (int i = 0; i < n; i++) dist[i] = length[v][i]; // initialize dist
5   for (int k = 2; k <= n-1; k++)
6     for (each u such that u != v and u has at least one incoming edge)
7       for (each <i, u> in the graph)
8         if (dist[u] > dist[i] + length[i][u]) dist[u] = dist[i] + length[i][u];
9 }
```

Complexity:

$O(n^3)$ when adjacency matrix is used

$O(n \cdot e)$ when adjacency lists are used

Outline

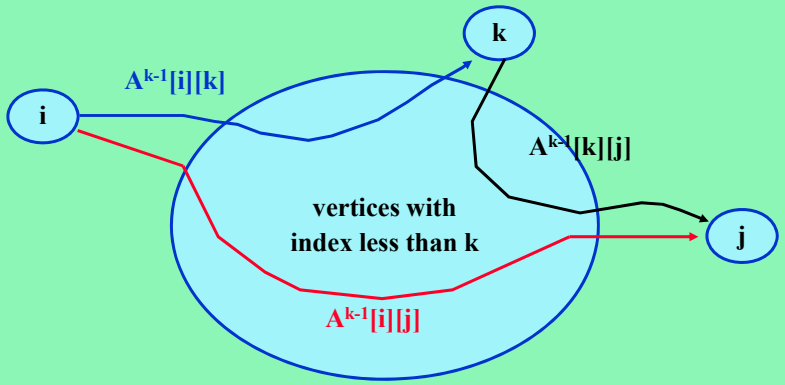
- **Shortest Path and Transitive Closure**
  - **Single Source / All Destinations**
  - **All-Pairs Shortest Paths**
  - **Transitive Closure**
- **Activity Network**
  - **Activity on Vertex (AOV) Networks**
  - **Activity on Edge (AOE) Networks**

Basics of All-Pairs Shortest-Paths

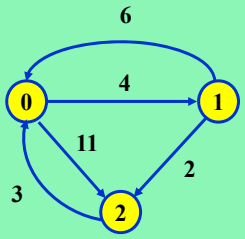
- **Assume that**
  - The digraph has n vertices with index of {0, ..., n-1}
- **Let $A^k[i][j]$**
  - be the length of the shortest path from i to j going through no intermediate vertex of index greater than k
- **$A^{n-1}[i][j]$**
  - will be the length of the shortest i-to-j path in G
- **The basic idea in all-pair algorithm**
  - is to successively generate the matrices A^{-1}, A^0, ..., A^{n-1}

Recurrence Relation

$A^k[i][j] = \min\{ A^{k-1}[i][j], A^{k-1}[i][k] + A^{k-1}[k][j] \}, \ k \geqq 0$
where $A^{-1}[i][j] = length[i][j]$



Example for All-Pairs Shortest-Paths Problem



A^{-1}	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

A^{-1}

A^{0}	0	1	2
0	0	4	11
1	6	0	2
2	3	7	0

A^{0}

A^{1}	0	1	2
0	0	4	6
1	6	0	2
2	3	7	0

A^{1}

A^{2}	0	1	2
0	0	4	6
1	5	0	2
2	3	7	0

A^{2}

All-Pairs Shortest Paths

$A^{k}[i][j] = \min\{ A^{k-1}[i][j], A^{k-1}[i][k] + A^{k-1}[k][j] \}, \; k \geqq 0$
where $A^{-1}[i][j] = length[i][j]$

```
1 void Graph::AllLengths(const int n)
2 // length [n ][n ] is the adjacency matrix of a graph with n vertices.
3 // a [i ][j ] is the length of the shortest path between i and j
4 {
5   for (int i = 0; i < n; i++)
6     for (int j = 0; j < n; j++)
7       a [i ][j ] = length [i ][j ];  // copy length into a
8   for (int k = 0; k < n; k++)  // for a path with highest vertex index k
9     for (i = 0; i < n; i++)  // for all possible pairs of vertices
10      for (int j = 0; j < n; j++)
11        if ((a [i ][k ] + a [k ][j ]) < a [i ][j ]) a [i ][j ] = a [i ][k ] + a [k ][j ];
12 }
```

Complexity:$O(n^3)$

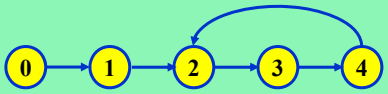
Outline

- **Shortest Path and Transitive Closure**
  - **Single Source / All Destinations**
  - **All-Pairs Shortest Paths**
  - **Transitive Closure**
- **Activity Network**
  - **Activity on Vertex (AOV) Networks**
  - **Activity on Edge (AOE) Networks**

Transitive Closure

- **Problem**
  - **Given a digraph with unweighted edges**
  - **We want to determine if there is a path from i to j for all values of i and j**
- **Transitive closure matrix of graph G, denoted as A^+**
  - **$A^+[i][j] = 1$ if there is a path of length >0 from i to j;**
  - **Otherwise, $A^+[i][j] = 0$;**
- **Reflexive transitive closure matrix, denoted as A^***
  - **$A^*[i][j] = 1$ if there is a path of length $\geqq 0$ from i to j;**
  - **Otherwise, $A^*[i][j] = 0$;**

Example of Transitive Closure Matrix



	0	1	2	3	4
0	0	1	0	0	0
1	0	0	1	0	0
2	0	0	0	1	0
3	0	0	0	0	1
4	0	0	1	0	0

adjacency matrix

	0	1	2	3	4
0	0	1	1	1	1
1	0	0	1	1	1
2	0	0	1	1	1
3	0	0	1	1	1
4	0	0	1	1	1

A^+

A[i][i]=1 when a cycle containing i exists

	0	1	2	3	4
0	1	1	1	1	1
1	0	1	1	1	1
2	0	0	1	1	1
3	0	0	1	1	1
4	0	0	1	1	1

A^*

A[i][i]=1 is always 1

Finding Transitive Closure

- **For Directed Graph**
  - The transitive closure can be computed using the all-pairs shortest-path algorithm
  - But the recurrence relation is modified as follows:

    a[i][j] = a[i][j] || (a[i][k] && a[k][j]);
  - The final matrix obtained is A^+
- **For Undirected Graph**
  - Transitive closure can be found more easily through the identification of connected components, $O(n^2)$
  - For a vertex pair (i, j), $A^+[i][j]=1$ if vertices i and j are in the same connected component
  - $A^+[i][i] = 1$ iff the component containing i has at least two vertices

Outline

- Shortest Path and Transitive Closure
- Activity Network
  - Activity on Vertex (AOV) Networks
  - Activity on Edge (AOE) Networks

Activities-On-Vertex (AOV) Networks

- Activity
  - A project can be subdivided into several subprojects called activities
- Definition of AOV network
  - Activity-on-vertex network is a directed graph G
  - The vertices represent tasks or activities
  - The edges represent precedence relations between tasks

Activities For Completing a Degree

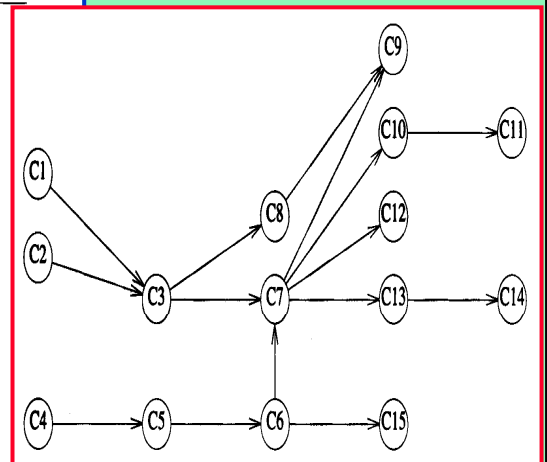
Course number	Course name	Prerequisites
C1	Programming I	None
C2	Discrete Mathematics	None
C3	Data Structures	C1, C2
C4	Calculus I	None
C5	Calculus II	C4
C6	Linear Algebra	C5
C7	Analysis of Algorithms	C3, C6
C8	Assembly Language	C3
C9	Operating Systems	C7, C8
C10	Programming Languages	C7
C11	Compiler Design	C10
C12	Artificial Intelligence	C7
C13	Computational Theory	C7
C14	Parallel Algorithms	C13
C15	Numerical Analysis	C5

(a) Courses needed for a computer science degree at a hypothetical university

Example: AOV Network

Course number	Course name	Prerequisites
C1	Programming I	None
C2	Discrete Mathematics	None
C3	Data Structures	C1, C2
C4	Calculus I	None
C5	Calculus II	C4
C6	Linear Algebra	C5
C7	Analysis of Algorithms	C3, C6
C8	Assembly Language	C3
C9	Operating Systems	C7, C8
C10	Programming Languages	C7
C11	Compiler Design	C10
C12	Artificial Intelligence	C7
C13	Computational Theory	C7
C14	Parallel Algorithms	C13
C15	Numerical Analysis	C5

(a) Courses needed for a computer science degree at a hypothetical university



(b) AOV network representing courses as vertices and prerequisites as edges

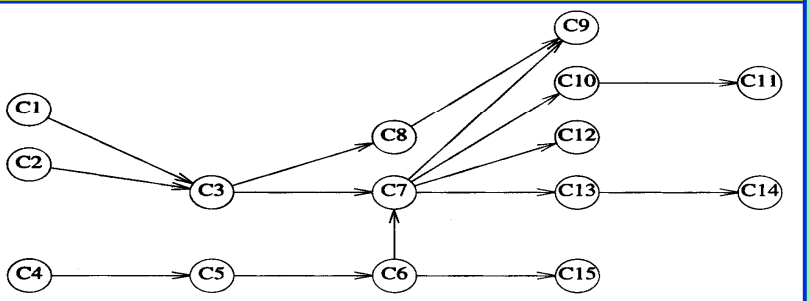
Terminology

- **Predecessor**
  - Vertex i is a predecessor of vertex j and j is a successor of i iff there is a directed path from vertex i to vertex j
  - i is an immediate predecessor of j iff <i, j> is an edge
- **Transitive relation**
  - A relation · is transitive iff it is the case that for all triples i, j, k, i · j and j · k ➔ i · k
- **Partial order**
  - A precedence relation that is both transitive and irreflexive is a partial order

Topological Order

- **Topological order**
  - A topological order is a linear order of the vertices
  - For any two vertices i and j, if i is a predecessor of j in the network, then i precedes j in the linear ordering

Topological order: C1 C2 C4 C5 C3 C6 C8 C7 C10 C13 C12 C14 C15 C11 C9
C4 C5 C2 C1 C6 C3 C8 C15 C7 C9 C10 C11 C12 C13 C14

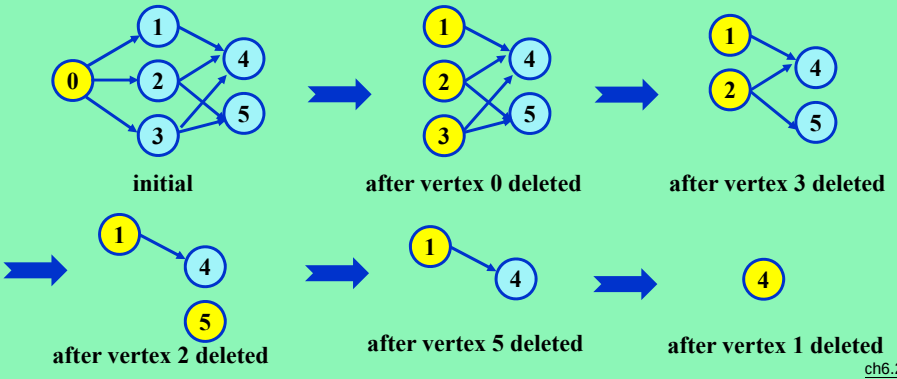


(b) AOV network representing courses as vertices and prerequisites as edges

A Topological Sorting Algorithm

```
Input the AOV network. Let n be the number of vertices
for ( int  i=0; i<n; i++){  //  output the vertices
{
   if ( every vertex has a predecessor )  return;  // network has a cycle
   pick a vertex v that has no predecessors;
   cout << v;
   delete v and all edges leading out of v from the network;
}
```

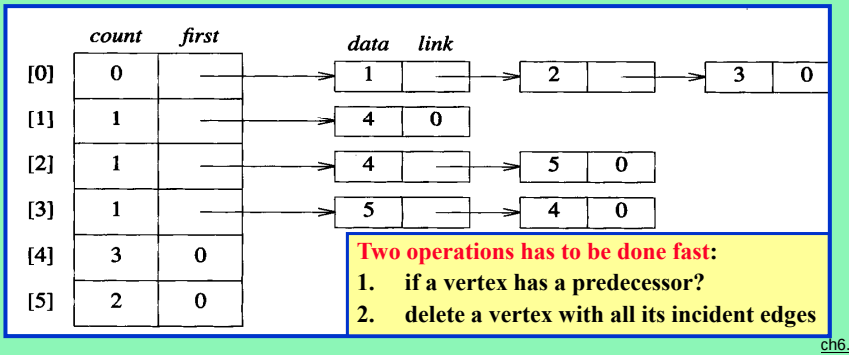
Complexity: O(e+n)



Internal Representations Used By Topological Sorting

```
class  Graph {
private:
  List<int>  *HeadNodes;
  int  *count;  // keep in-degree
  int   n;
```

```
public:
  Graph( const int  vertices=0) : n (vertices) {
  HeadNodes = new List<int>[n];
  count = new int[n];
  };
  void  TopologicalOrder();
};
```



Two operations has to be done fast:

1. if a vertex has a predecessor?
2. delete a vertex with all its incident edges

Topological Sorting Algorithm

```
1 void Graph::TopologicalOrder ()
2 // The n vertices of a network are listed in topological order
3 {
4    int top = -1;
5    for (int i = 0; i < n; i++) // create a linked stack of vertices with
6       if (count [i ] == 0) { count [i ] = top; top = i;} // no predecessors

7    for (i = 0; i < n; i++)
8       if (top == -1) { cout << " network has a cycle" << endl; return;}
9       else {
10         int j = top; top = count [top ]; // unstack a vertex
11         cout << j << endl;
12         ListIterator<int> li (HeadNodes [j ]);
13         if (! li.NotNull ()) continue;
14         int k = *li.First ();
15         while (1) { // decrease the count of the successor vertices of j
16            count [k ]--;
17            if (count [k ] == 0) { count [k ] = top; top = k;} // add vertex k to stack
18            if (li.NextNotNull ()) k = *li.Next (); // k is a successor of j
19            else break;
20         }
21      } // end of else
22 }
```

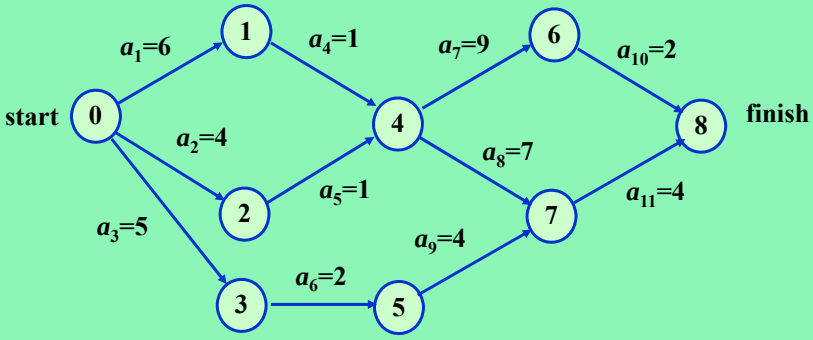
Complexity is: O(e+n)

Outline

- **Shortest Path and Transitive Closure**
- **Activity Network**
  - **Activity on Vertex (AOV) Networks**
  - **Activity on Edge (AOE) Networks**

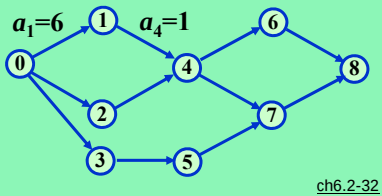
Activity-On-Edge (AOE) Network

- **AOE Network**
  - Is a weighted directed graph in which
  - The vertices represent events
  - The edges represent activities or tasks



Terminology

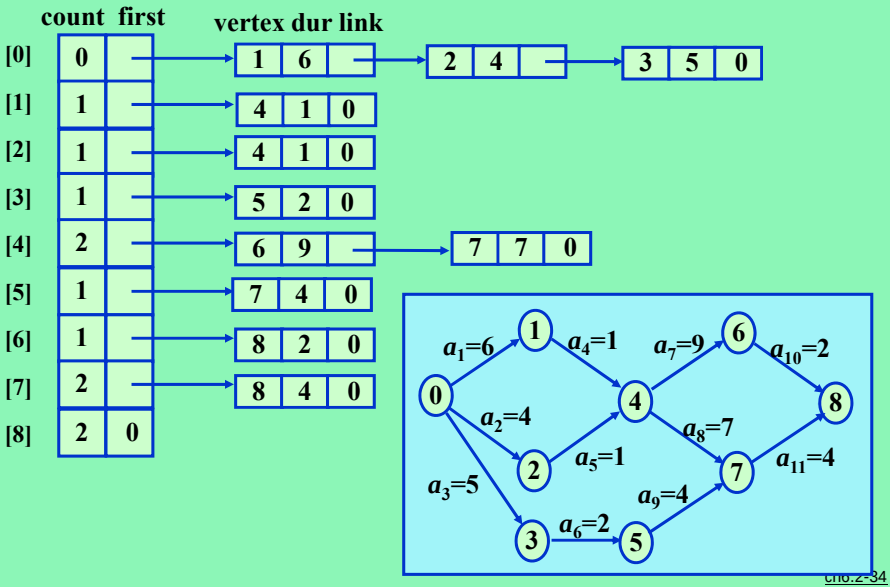
- **Critical Path**
  - Is the longest path from start vertex to finish vertex
  - determines the minimum amount of time to finish the project
- **Earliest Time of an activity a_i, denoted as $e(a_i)$**
  - is the length of the longest path from start to the source vertex of a_i
- **Latest Time of an activity a_i, denoted as $l(a_i)$**
  - indicates latest time an activity may start without increasing the project duration
- **Critical activity**
  - is an activity for which $e(a_i) = l(a_i)$



Critical Path Analysis

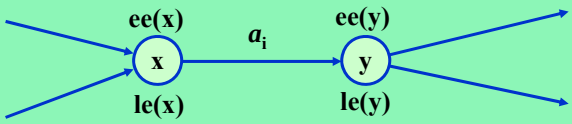
- **Purpose of critical path analysis**
  - is to identify the critical activities so that resources may be concentrated on these activities in an attempt to reduce project finish time
  - That is, it is useful to identify project bottlenecks
- **Finding all critical paths**
  - (1) compute every activity's $e(a_i)$ and $l(a_i)$
  - (2) identify critical activities, i.e., a_i for which $e(a_i)=l(a_i)$
  - (3) remove all non-critical activities
  - (4) Generate all paths from start to finish

Example: Data Representation



Overall Procedure of Critical Path Analysis

- **Procedure**
  - **Step 1: compute the earliest event time for each vertex i, denoted as ee(i)**
  - **Step 2: compute the latest event time for each vertex i, denoted as le(i)**
  - **Step 3: compute the earliest time for an activity a_i , <x, y>, using formula: $e(a_i) = ee(x)$**
  - **Step 4: compute the latest time for an activity a_i , <x, y>, using formula $l(a_i) = le(y) - $ duration of activity a_i**



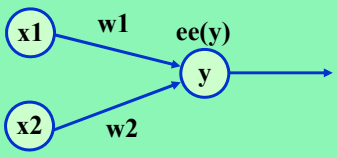
Calculation of Earliest Activity Times

- **The earliest event time of each vertex**
  - **can be computed by a forward stage**
- **Step 1: Sort vertices in the topological order**
- **Step 2: Evaluate earliest event time of each vertex by**

$$ee(y) = \max_{x_i \in P(y)} \{ ee(x_i) + \text{duration of } \langle x_i, y \rangle \}$$

P(y) is the set of y's immediate predecessors

- **Step 3: $e(a_i) = ee($ source vertex of $a_i)$**



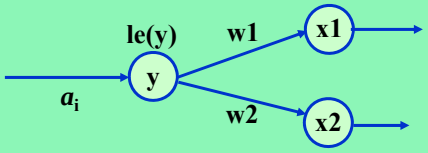
Calculation of Latest Activity Times

- **The latest event time of each vertex**
  - **can be computed by a backward stage**
- **Step 1: Sort vertices in reverse topological order**
- **Step 2: Evaluate latest event time of each vertex by**

$$le(y) = \min_{x_i \in S(y)} \{ le(x_i) - \text{duration of } \langle y, x_i \rangle \}$$

S(y) is the set of y's immediate successors

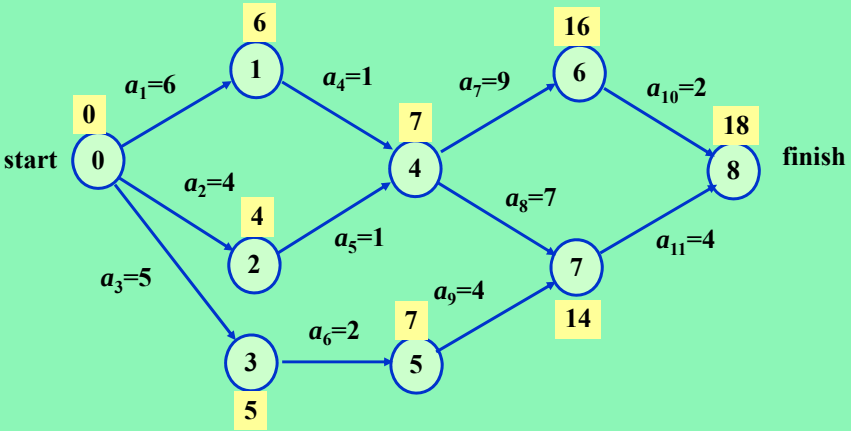
- **Step 3: $l(a_i)$ = le(destination vertex of a_i) – duration of activity a_i**



Example of Computing Earliest Event Times

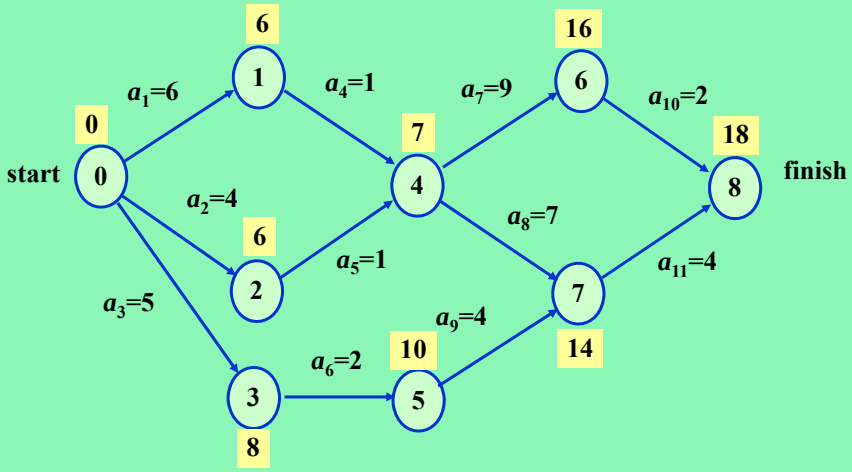
topological order: 0, 1, 2, 3, 4, 5, 6, 7, 8

$e(a_i)$ = ee(source vertex of a_i)



Example of Computing Latest Event Times

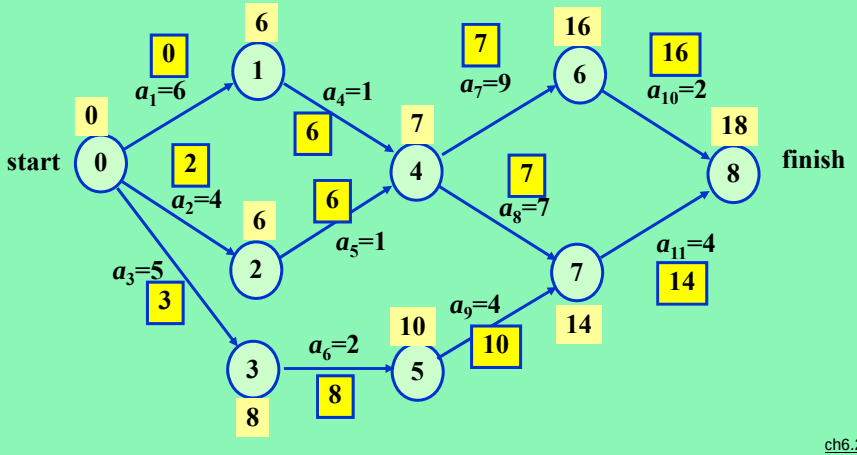
reverse topological order: 8, 7, 6, 5, 4, 3, 2, 1, 0



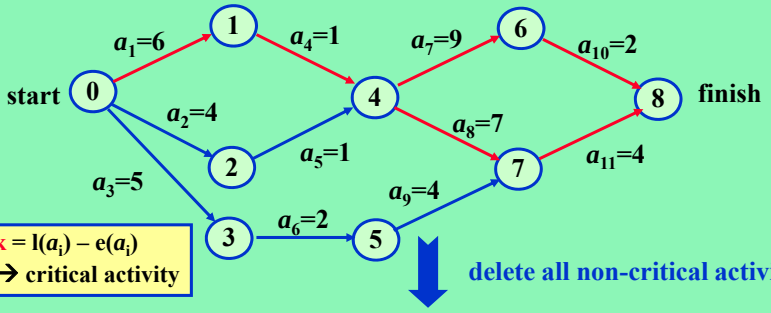
Example of Computing Latest Activity Times

reverse topological order: 8, 7, 6, 5, 4, 3, 2, 1, 0

$l(a_i)$ = le(destination vertex of a_i) – duration of activity a_i



Graph of Critical Paths



slack = $l(a_i) - e(a_i)$
slack=0 → critical activity

delete all non-critical activities