

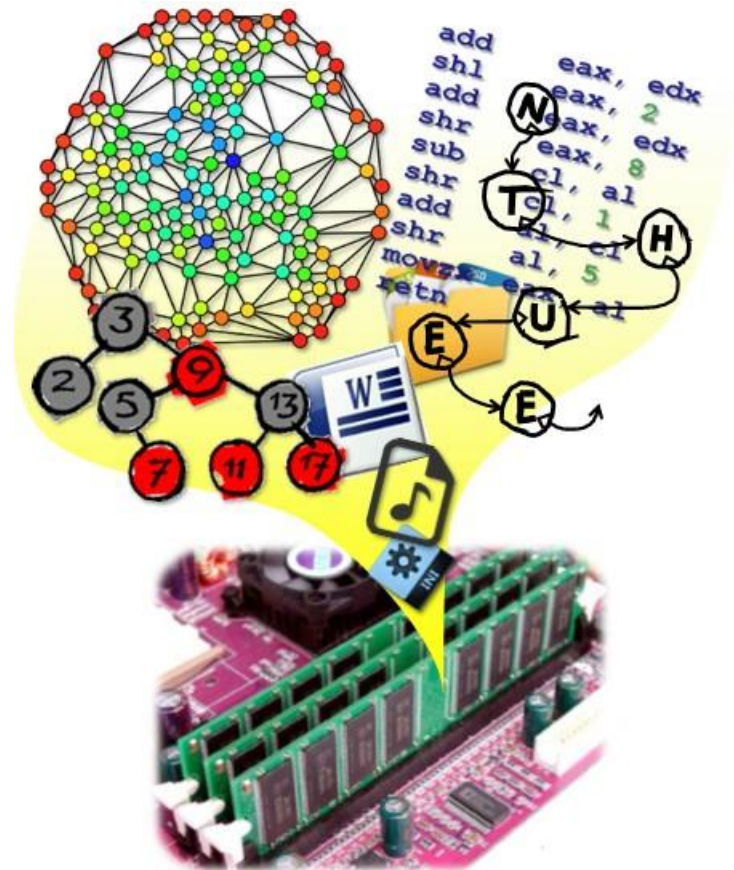
Data Structures

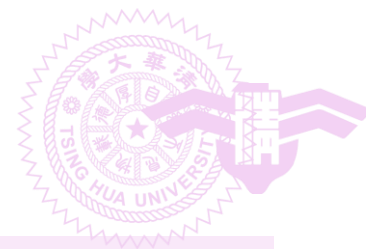
CH6 Graphs

Prof. Ren-Shuo Liu

NTHU EE

Spring 2017



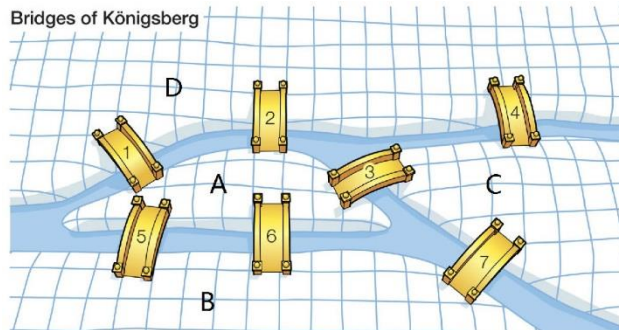


Outline

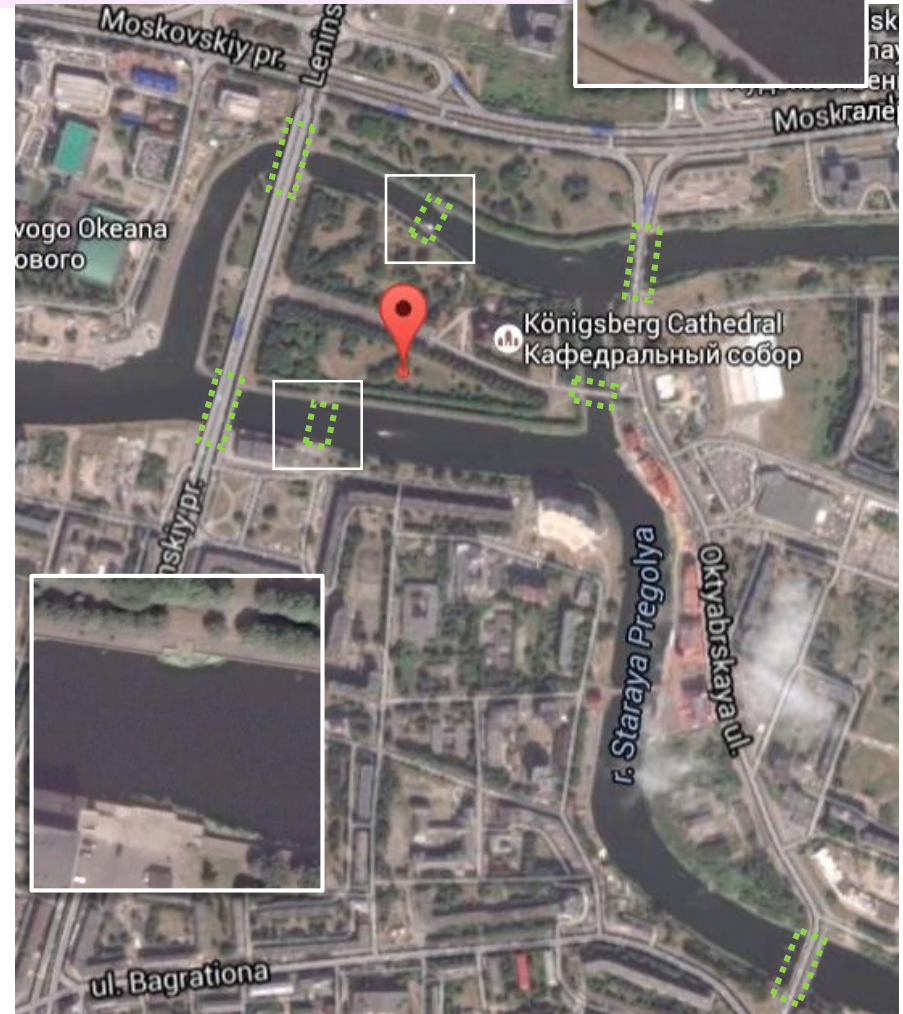
- **6.1 Introduction and the graph abstract data type**
- 6.2 Elementary graph operations
- 6.3 Minimum-cost spanning trees
- 6.4 Shortest paths and transitive closure
- 6.5 Activity networks

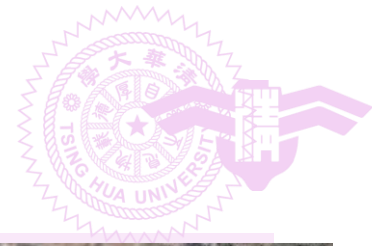
Konigsberg Bridge Problem

- Also known as "一筆畫問題"
 - Four land areas are interconnected by seven bridges



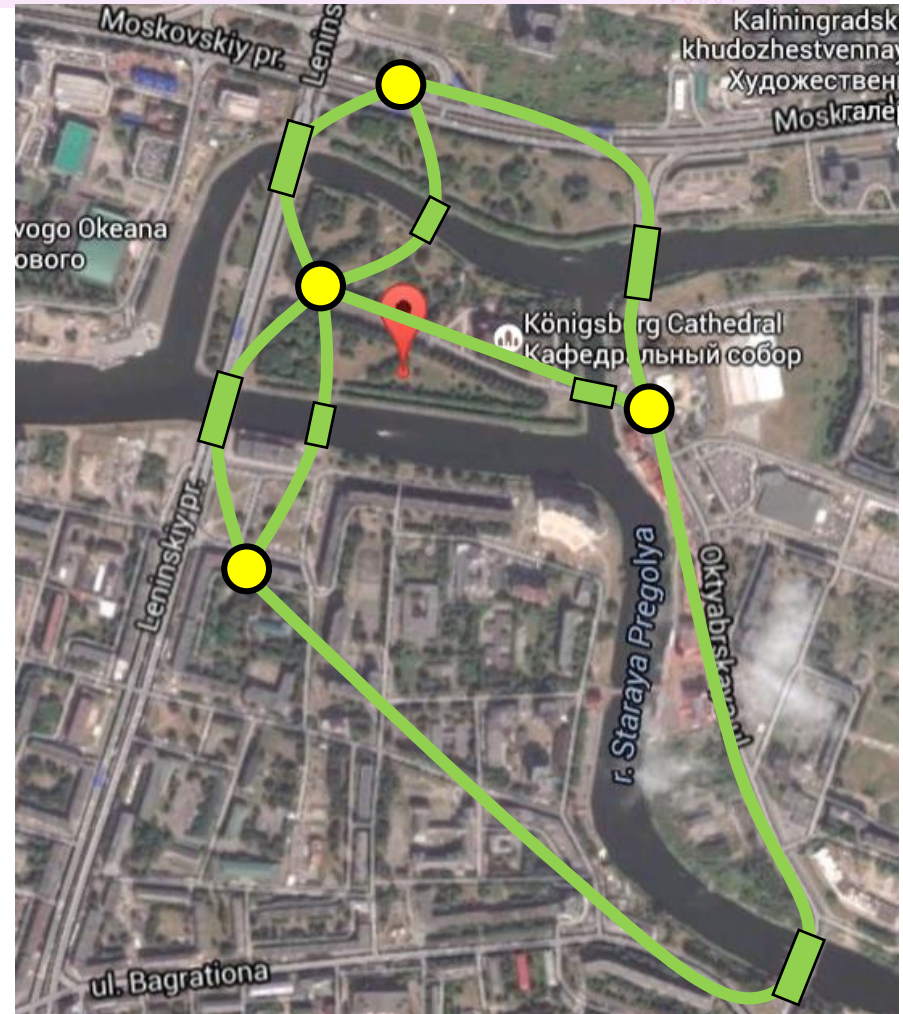
- Is it possible to walk across seven bridges exactly once in returning to the starting place?

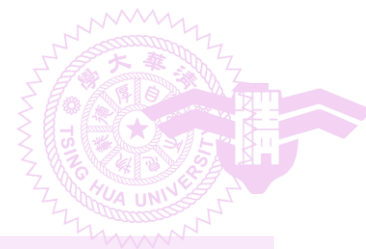




Konigsberg Bridge Problem

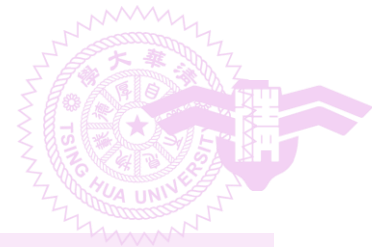
- Euler solved the problem by representing the land areas as **vertices** and the bridges as **edges** (1736)
 - First recorded evidence of the use of graphs
- Since then, graphs have been used in a wide variety of applications
 - Analysis of circuits, genetics, social networks...



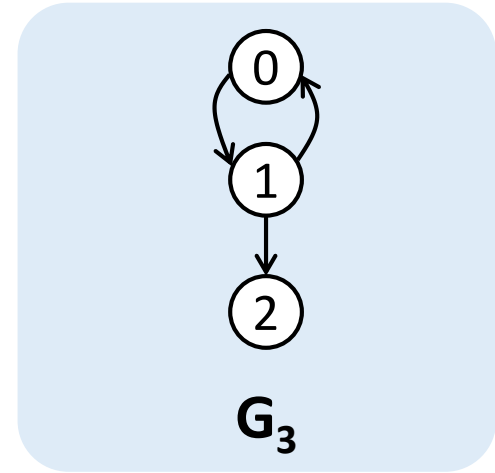
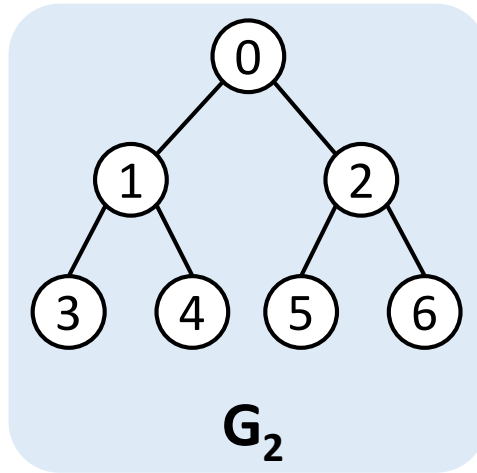
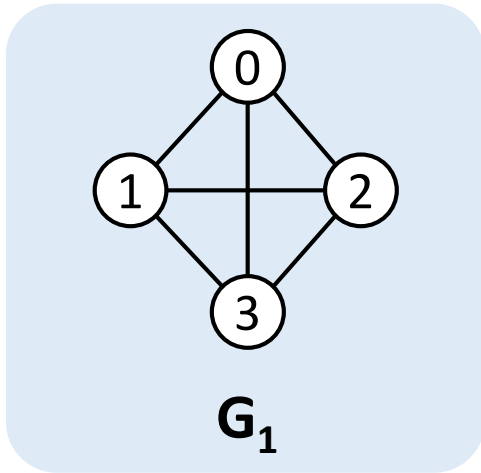


Graphs

- Definition : A graph, G , consists of two sets, V and E
 - $G = (V, E)$
- V is a finite, nonempty set of **vertices**
- E is a set of pairs of vertices, called **edges**
 - **Undirected** graphs (無向圖)
 - Pair of vertices representing any edge is unordered
 - (u, v) and (v, u) represent the same edge
 - **Directed** graphs (**digraphs**) (有向圖)
 - Each edge is represented by a directed pair $\langle u, v \rangle$
 - u is the tail and v the head of the edge



Graphs



$$V(G_1) = \{0, 1, 2, 3\}$$

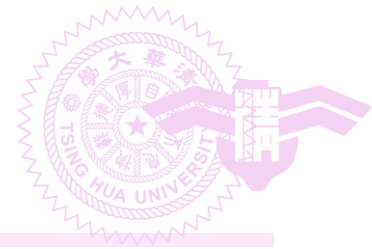
$$E(G_1) = \{(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)\}$$

$$V(G_2) = \{0, 1, 2, 3, 4, 5, 6\}$$

$$E(G_2) = \{(0, 1), (0, 2), (1, 3), (1, 4), (2, 5), (2, 6)\}$$

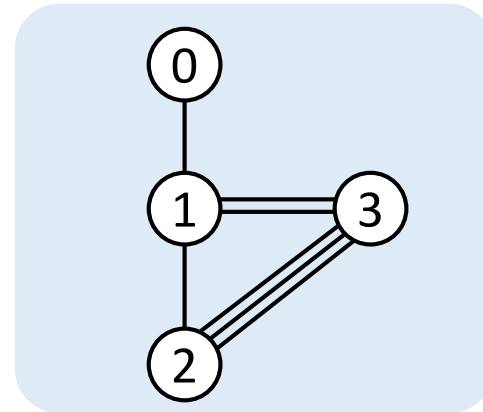
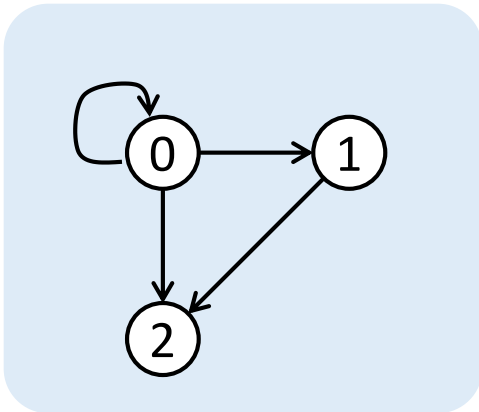
$$V(G_3) = \{0, 1, 2\}$$

$$E(G_3) = \{\langle 0, 1 \rangle, \langle 1, 0 \rangle, \langle 1, 2 \rangle\}$$



Simple Graphs (Strict Graphs)

- This book only considers **simple graphs** (strict graphs)
- The followings are not allowed in simple graphs
 - Self edges / **self loops**
 - (v, v)
 - $\langle v, v \rangle$
 - Multiple occurrences of the same edge

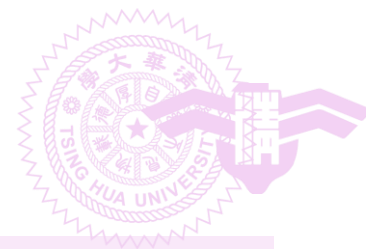


- Therefore, the max number of edges of an n -vertex simple graph
 - $n(n-1)/2$ for an undirected graph
 - $n(n-1)$ for an directed graph



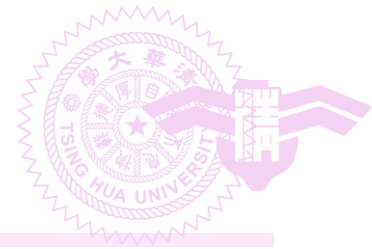
Terminologies

- **Complete graphs** (also called as **cliques** (團))
 - A graph having the max possible number of edges
 - $n(n-1)/2$ for an undirected graph
 - $n(n-1)$ for an directed graph
- **Adjacency and incidence**
 - u and v are **adjacent** if $(u, v) \in G$
 - (u, v) is **incident on** (關聯) u and also v
- A **subgraph** of G is a graph G' such that
 - $V(G') \subseteq V(G)$
 - $E(G') \subseteq E(G)$



Terminologies

- A **path** from u to v in a graph G is
 - a sequence of vertices: $u, i_1, i_2, \dots, i_k, v$
 - $(u, i_1), (i_1, i_2), \dots, (i_k, v) \in E(G)$, G is undirected
 - $\langle u, i_1 \rangle, \langle i_1, i_2 \rangle, \dots, \langle i_k, v \rangle \in E(G)$, G is directed
- A **simple path** is
 - a path in which all vertices except possibly the first and last are distinct
- A **cycle** is
 - a simple path in which the first and last are the same



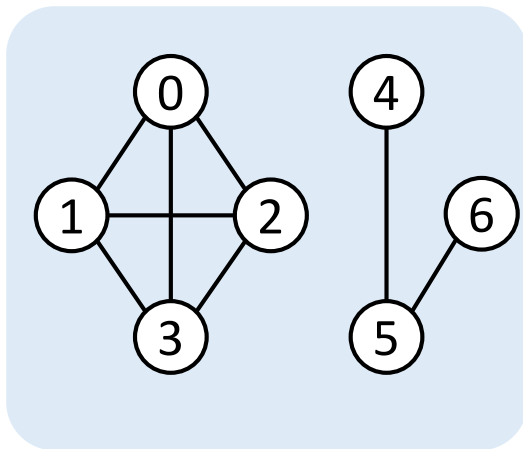
Connectivity

- In an undirected graph, vertices u and v are **connected** iff there is a path from u to v
- An undirected graph is **connected** iff every pair of distinct vertices u and v in $V(G)$ is connected
 - A tree is a connected acyclic graph

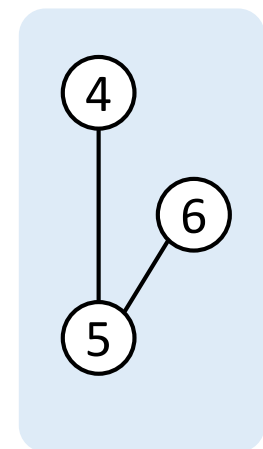
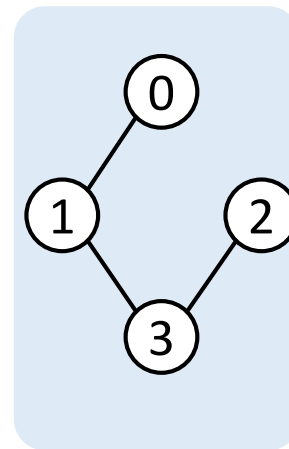
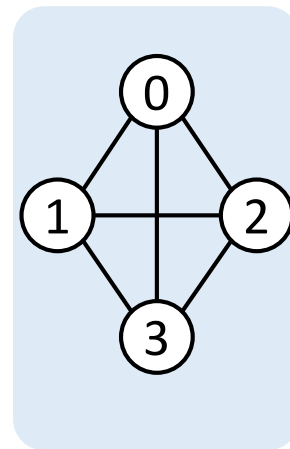


Connected Components

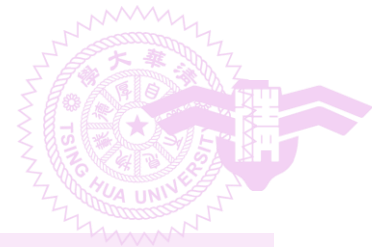
- A **connected component** (or **component** for short), H , of an undirected graph is
 - the maximal connected subgraph
 - By maximal, we mean that G contains no other subgraph that is both connected and properly contains H



G_1

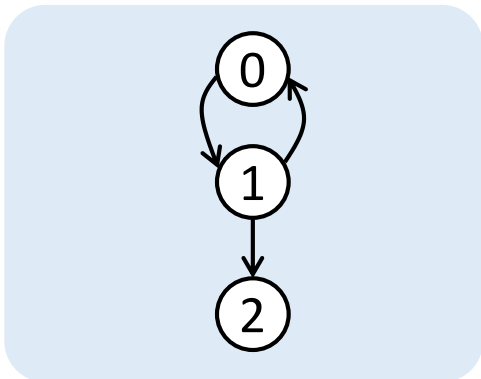


Some connected components of G_1

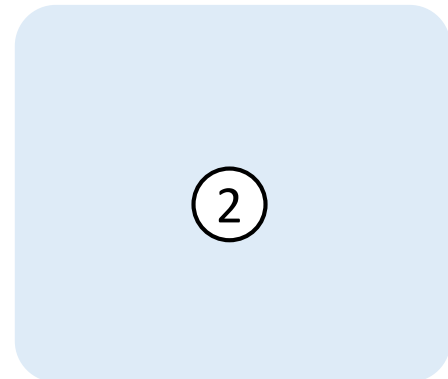
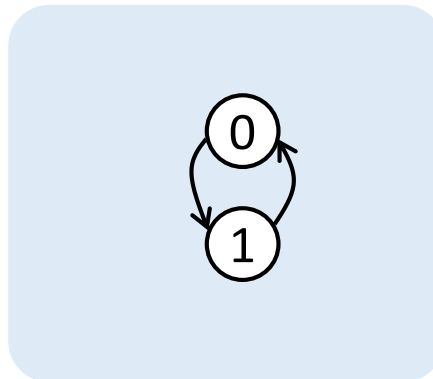


Strongly Connected Graphs

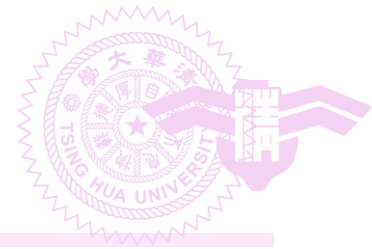
- A **digraph** G is said to be **strongly connected** iff for every pair of distinct vertices u and v in $V(G)$ there is a directed path from u to v and also from v to u
- A **strongly connected component** is a maximal subgraph that is strongly connected



G_3

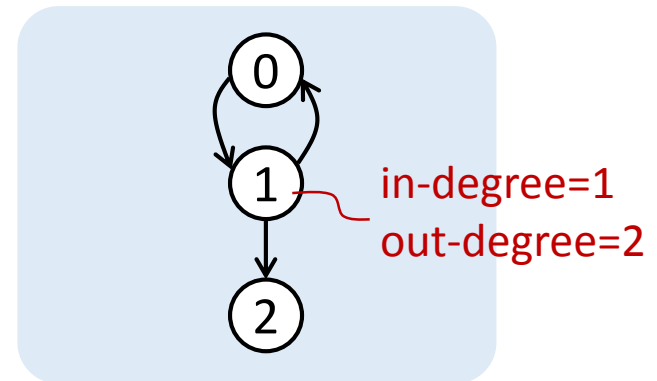
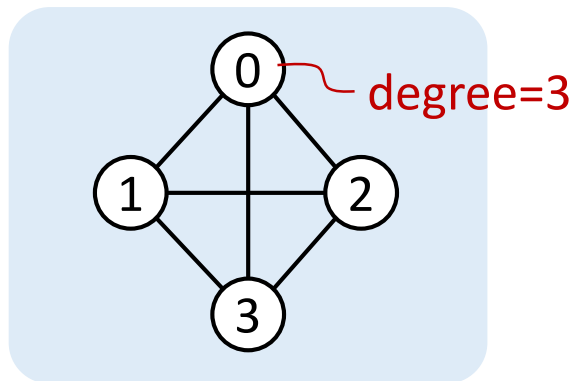


Strongly connected components of G_3



Degree

- The **degree** of a vertex is the number of edges incident to that vertex
- For digraph
 - The **out-degree** of a vertex v is the number of edges for which v is the tail
 - The **in-degree** of a vertex v is the number of edges for which v is the head



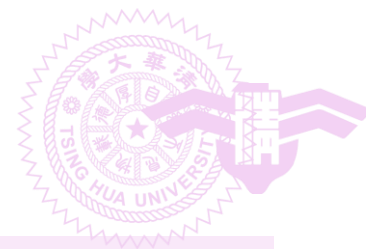


Graph ADT

```
class Graph
{
public:
    virtual ~Graph() {} // virtual destructor
    bool IsEmpty() const           {return n == 0};
    int NumberOfVertices() const   {return n};
    int NumberOfEdges() const      {return e};

    virtual int Degree(int u) const           = 0;
    virtual bool ExistsEdge(int u, int v) const = 0;
    virtual void InsertVertex(int v)          = 0;
    virtual void InsertEdge(int u, int v)     = 0;
    virtual void DeleteVertex(int v)          = 0;
    virtual void DeleteEdge(int u, int v)     = 0;

private:
    int n; // number of vertices
    int e; // number of edges
};
```



Inheritance vs. Template

- Key question: do **types** affect the **behaviors** of a class **according to your expectation**?
- Inheritance: types may affect behaviors
 - **Rectangle** and **Circle** can calculate their areas but have different calculating mechanisms
 - According to this expectation, we design a base class, Shape, with a virtual GetArea() method and let specific shape classes to inherit
- Template: types do not affect behaviors
 - Stack exhibits a **last-in-first-out** behavior
 - Both **Stack of Rectangle** and **Stack of Circle** do so
 - According to this expectation, we design a template stack instead of a base stack and different inherited classes



Non-Virtual vs Virtual Functions

- Non-virtual
 - **Static-binding** (at compile time) according to the type of a object pointer or reference
- Virtual
 - **Dynamic-binding** (resolved at run time) according to hidden information in each object
 - Polymorphism: derived classes exhibit their specific behavior even if they are referred to using the base class pointer/reference

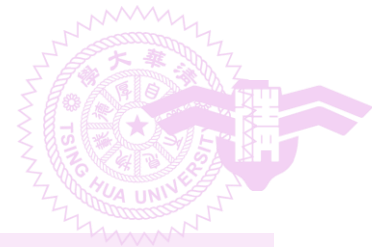
```
int main()
{
    Rectangle r;
    Circle c;
    cout << AreaRatio(r, c);
    cout << AreaRatio(c, r);
}

float AreaRatio(Shape& s1, Shape& s2)
{
    return s1.GetArea() / s2.GetArea();
}
```



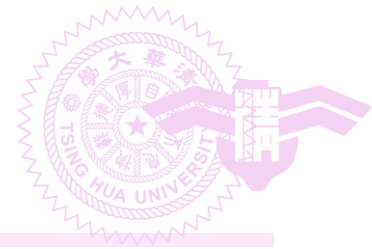

Pure Virtual Functions

- Sometimes we want **derived classes NOT to inherit the implementation of a virtual function by default**
 - Implementation of GetArea()
 - Maybe impractical to have one method to calculate the area of both Rectangle and Circle
 - Maybe error-prone if someone inherits from Shape another specific shape, say Star, without redefining Star's GetArea



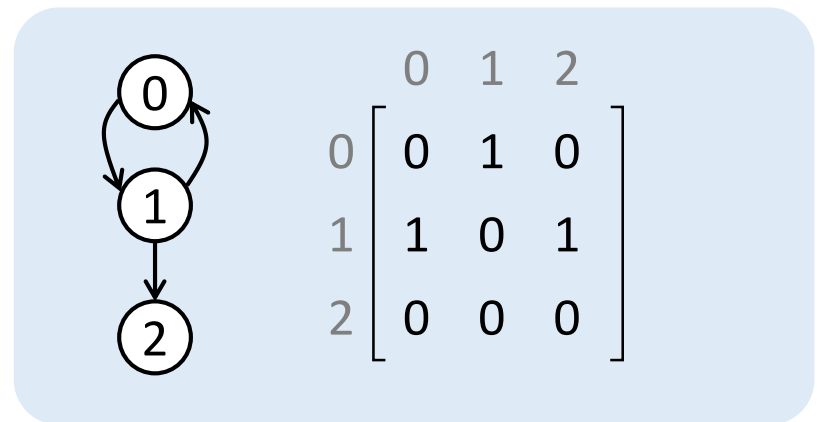
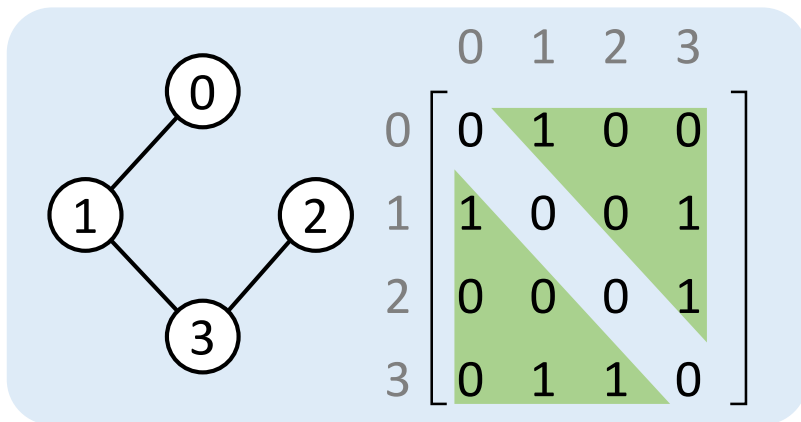
Graph Representations

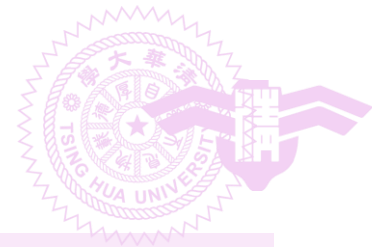
- Three categories most commonly used representations
 - Adjacency **matrices**
 - Adjacency **lists**
 - Adjacency **multi-lists**
- The choice of a particular representation will depend upon the **application** one has **in mind** and the **functions** one **expects** to perform on the graph



Adjacency Matrices

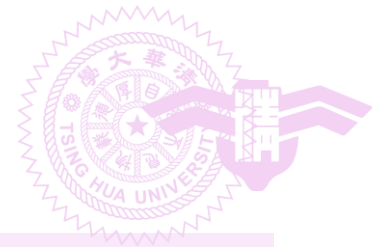
- The adjacency matrix of an n -vertex graph, G , is a 2D $n \times n$ array, say array A
 - $A[u][v] = 1$ iff (u, v) (or $\langle u, v \rangle$) is in $E(G)$
 - $A[u][v] = 0$ otherwise
- Adjacency matrices of **undirected** graphs are always **symmetric**
 - This allows optimization that halves the space requirement
- Adjacency matrices **are wasteful of space for sparse graphs** (i.e., graphs with only few edges)





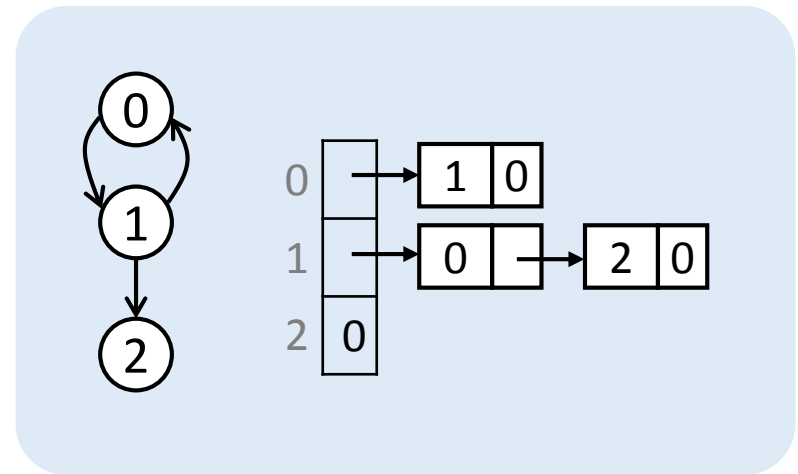
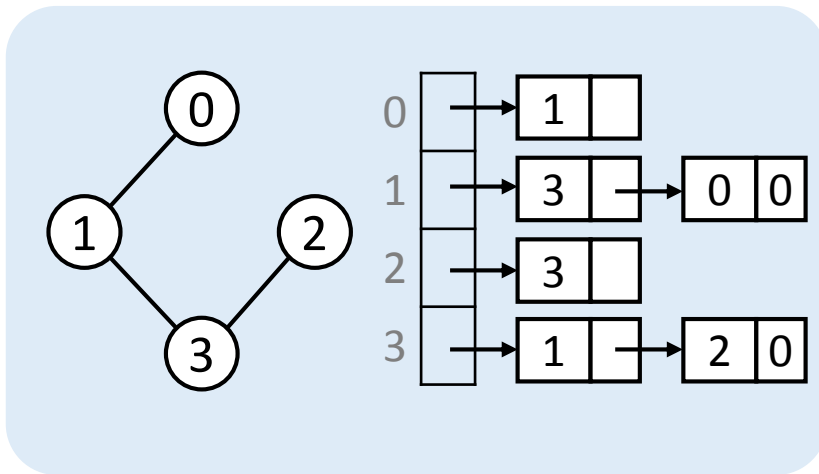
Adjacency Matrix Operations

Degree (int u)	Return $\sum a[u][i]$;	$O(n)$
Out-degree(int u)	Return $\sum a[u][i]$;	$O(n)$
In-degree(int u)	Return $\sum a[i][u]$;	$O(n)$
ExistsEdge(int u, int v)	Return $a[u][v]$;	$O(1)$
InsertEdge(int u, int v)	Set $a[u][v] = 1$;	$O(1)$
DeleteEdge(int u, int v)	Set $a[u][v] = 0$;	$O(1)$
IsConnectedGraph()		$O(n^2)$



(Linked) Adjacency Lists

- The adjacency list of an n -vertex, e -edge graph, G
 - Contains an n -element array, n chains, and $2e$ chain nodes
 - Nodes in chain i represent the vertices adjacent from vertex i
 - Nodes in each chain are not required to be ordered
- (Recall the equivalence class problem)

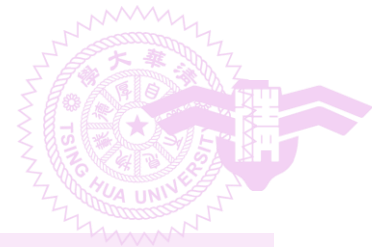


(Linked) Adjacency List Operations



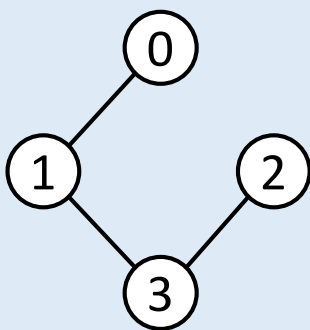
Degree (int u)	Count the # of nodes in chain u;	$O(e)$
Out-degree(int u)	Count the # of nodes in chain u;	$O(e)$
In-degree(int u)	Count the # of u's in all the chains;	$O(n+e)$
ExistsEdge(int u, int v)	Look for v in chain u;	$O(e)$
InsertEdge(int u, int v)	• Check the existence of v in chain u (and u in v);	$O(e)$
	• Push v onto chain u (and u onto v);	$O(1)$
DeleteEdge(int u, int v)	• Find v in chain u (and u in v);	$O(e)$
	• Remove v from chain u (and u from v);	$O(1)$
IsConnectedGraph()		$O(n+e)$

Facebook would need to scan its billions of users to calculate how many people [follows your page](#) if Facebook uses the simple Adjacency List representation



Sequential Adjacency Lists

- Sequential adjacency list of an n -vertex, e -edge graph, G
 - Contains an $(n+2e+1)$ -element array
 - $n+1$ for indexing the list of each vertex
 - $2e$ for adjacency information

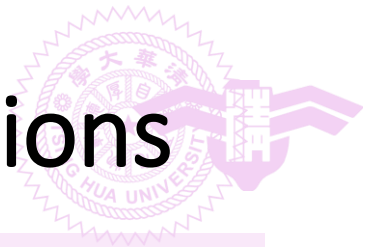


starting/ending indices
of the lists of n vertices

set to $(n+2e+1)$

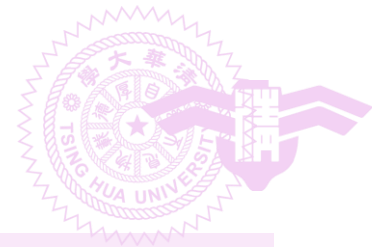
0	1	2	3	4	5	6	7	8	9	10
5	6	8	9	11	1	3	0	3	1	2





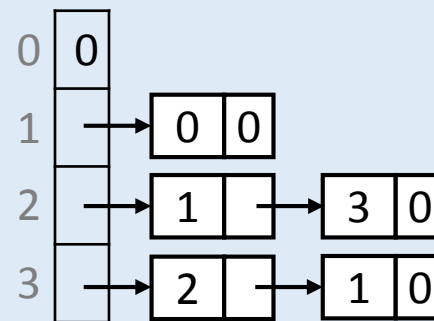
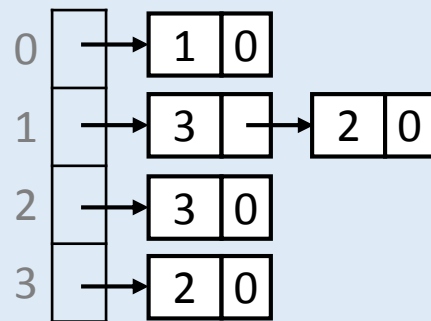
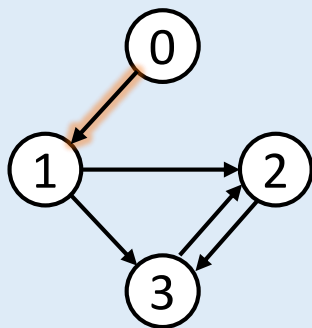
Sequential Adjacency List Operations

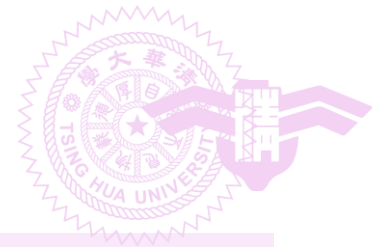
Degree (int u)	Return the index difference between u and u+1	$O(1)$
Out-degree(int u)	Return the index difference between u and u+1	$O(1)$
In-degree(int u)	Count the # of u's in the entire graph;	$O(n+e)$
ExistsEdge(int u, int v)	Look for v in list u;	$O(e)$
InsertEdge(int u, int v)	Make space and insert the edge	$O(n+e)$
DeleteEdge(int u, int v)	Delete the edge and compact the array	$O(n+e)$
IsConnectedGraph()		$O(n+e)$



Inverse Adjacency Lists

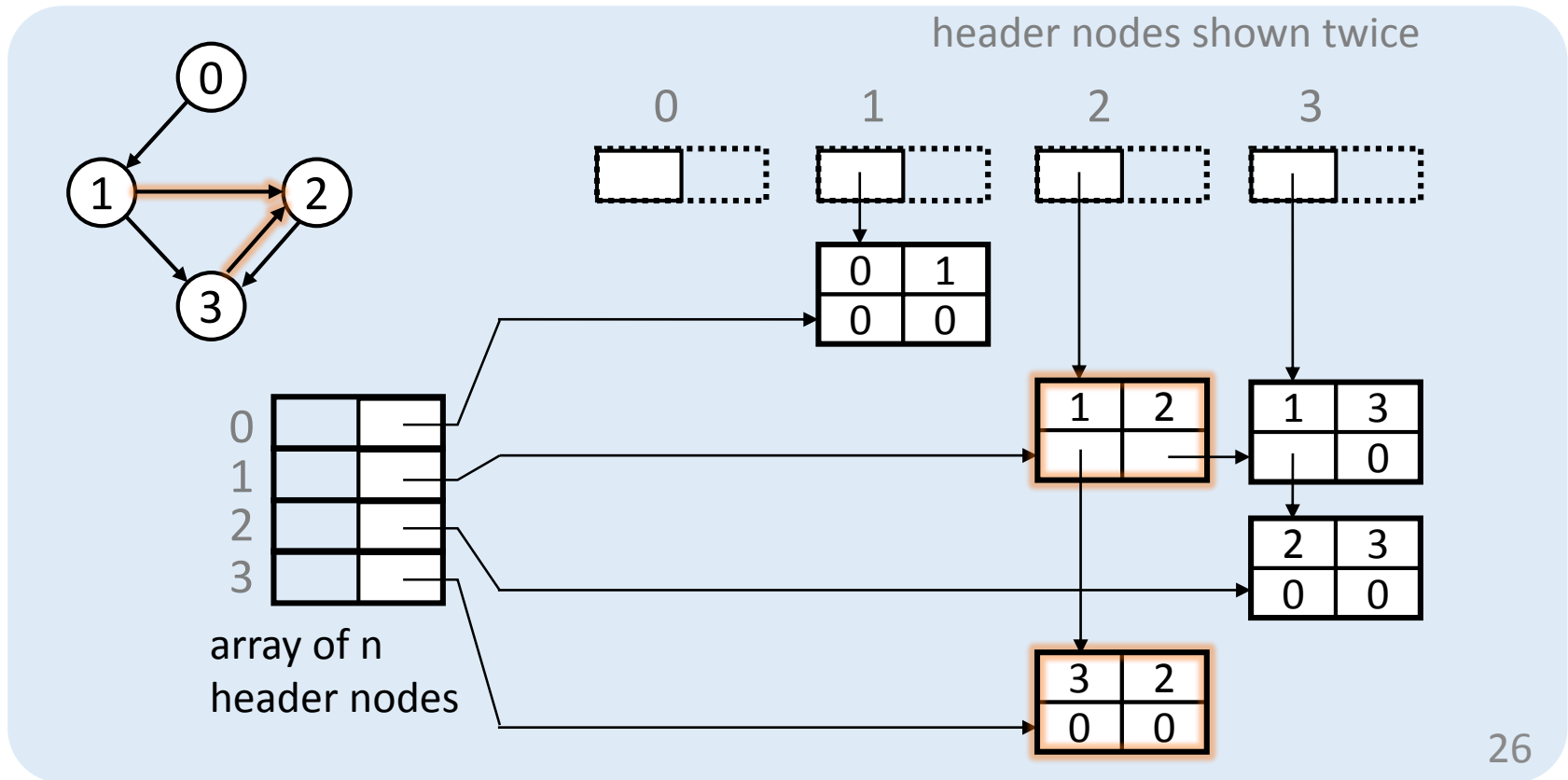
- Ease repeatedly accessing all vertices adjacent to and from another vertex in a digraph
 - E.g., in-degree and out-degree
 - Keep an additional inverse adjacency list
 - List i stores edges of the form $\langle x, i \rangle$
- An alternative is to use orthogonal adjacency lists





Orthogonal Adjacency List

- Use an $n \times n$ **orthogonal (正交) list** to store the adjacency information
 - Terms correspond to edges, (u, v) or $\langle u, v \rangle$
- (Recall p.218 **sparse matrices**)





Orthogonal Adjacency List Operations

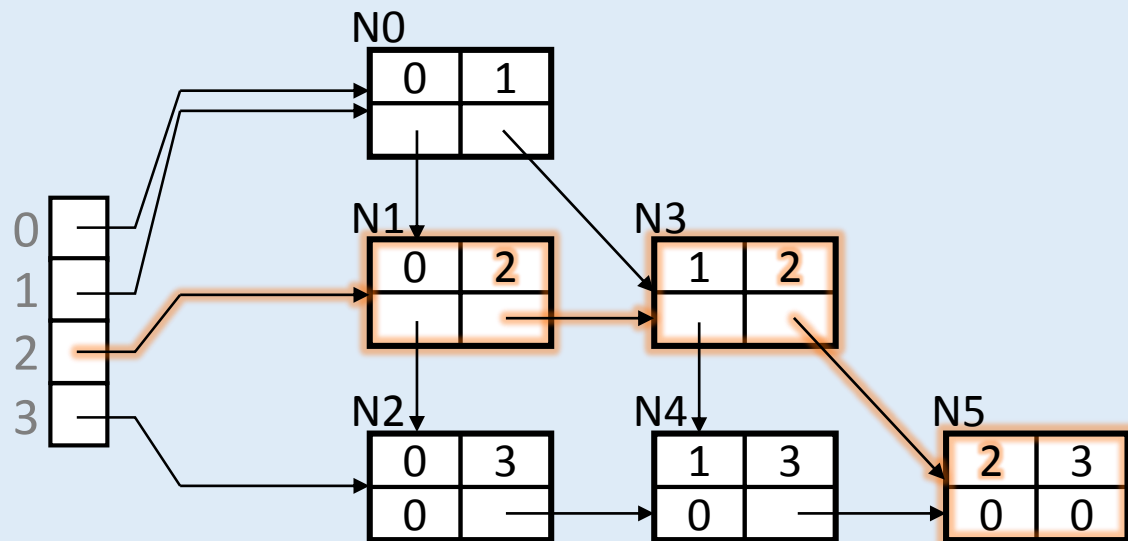
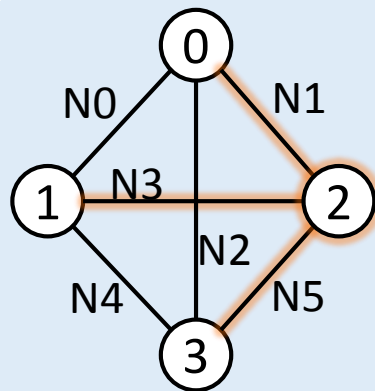
Out-degree(int u)	Count the # of nodes in chain u	O(e)
In-degree(int u)	Count the # of nodes in chain u	O(e)
ExistsEdge(int u, int v)	Look for v in chain u ;	O(e)
InsertEdge(int u, int v)	<ul style="list-style-type: none">• Check the existence of v in chain u (and u in v);• Insert (instead of push or append) v into chain u (and u into v);	O(e) O(e)
DeleteEdge(int u, int v)	<ul style="list-style-type: none">• Locate v in chain u (and u in v);• Remove v from chain u (and u from v);	O(e) O(1)
IsConnectedGraph()		O(n+e)





Adjacency Multi-Lists

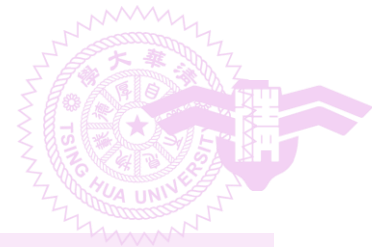
- Multi-lists
 - One node can be shared among multiple lists
- Adjacency multi-lists
 - An edge is represented by a node
 - Support accessing all edges incident on a vertex in undirected graphs





Discussion: Adjacency Matrices vs Adjacency Lists

Operation	Which performs better
Determining if (u, v) is an edge in G	
Degree of vertex u	
Determining if there is a path from u to v	
Adding an edge to G	
Space to store a dense graph	
Space to store a sparse graph	
Konigsberg Bridge Problem	



Outline

- 6.1 The graph abstract data type
- **6.2 Elementary graph operations**
- 6.3 Minimum-cost spanning trees
- 6.4 Shortest paths and transitive closure
- 6.5 Activity networks

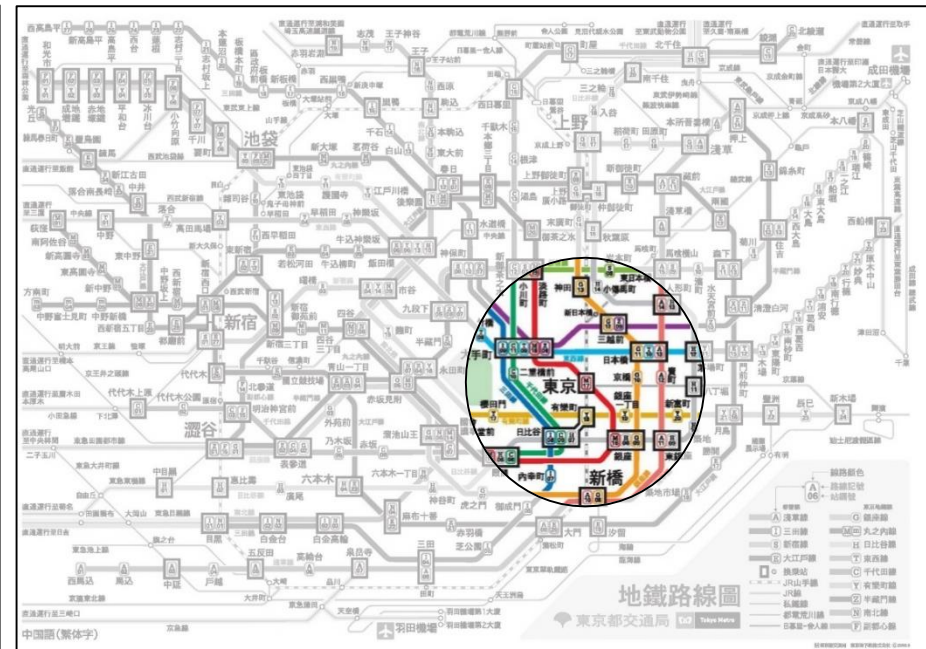
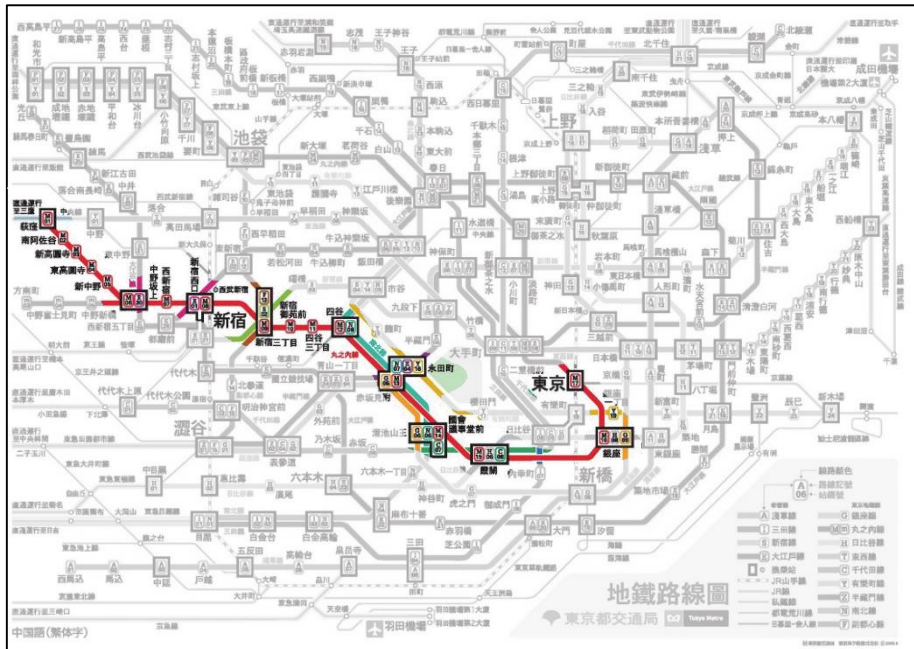


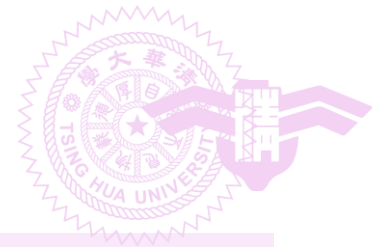
Elementary Graph Operations

- Depth-first search (DFS)
- Breadth-first search (BFS)
- Connected components
- Spanning trees
- Biconnected components

Concept of Search

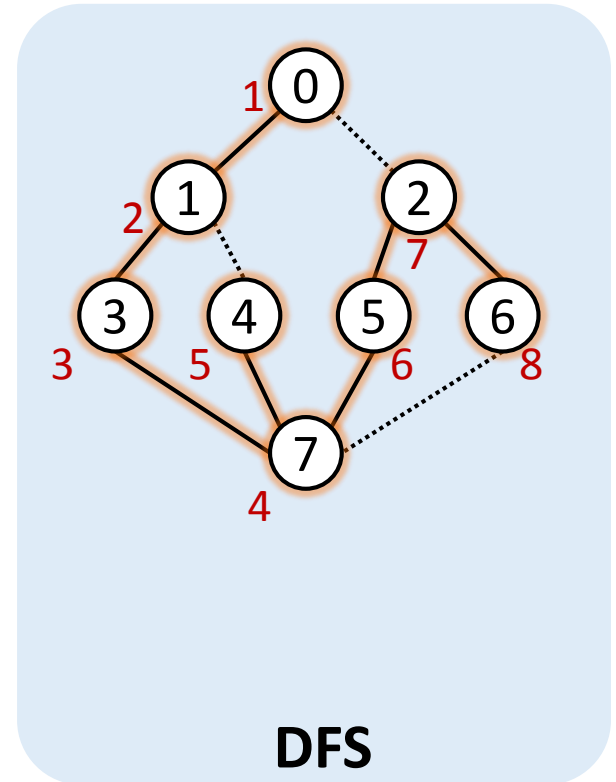
- Suppose we want to systematically traverse a city with a subway map in hand
 - Depth-first style
 - Following a subway path and visiting the places one after one
 - Breadth-first style
 - Visiting all places within a certain traveling distance

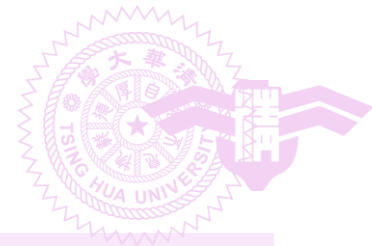




Depth-First Search (DFS)

- Begin by visiting the **start vertex** v
- Next an unvisited vertex w adjacent to v is selected
- A depth-first search from w is initiated
 - Recursion
- Backtrack if no unvisited vertices are reachable

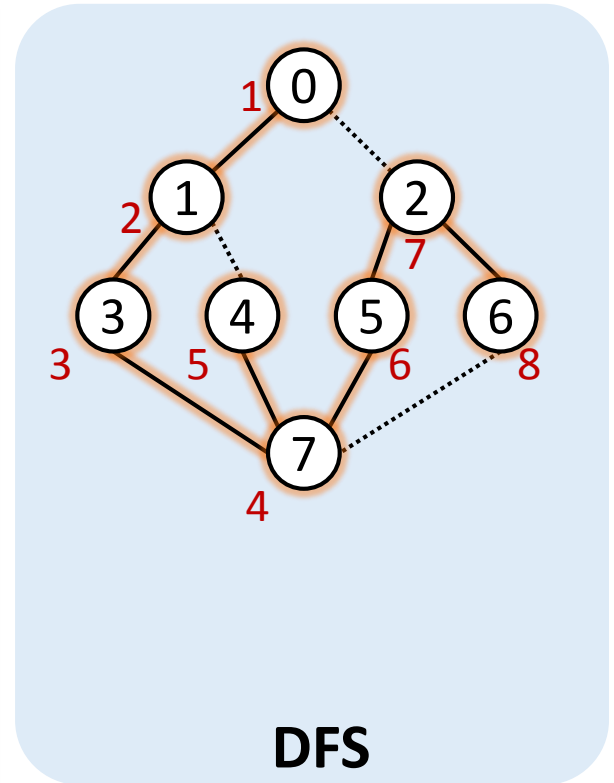


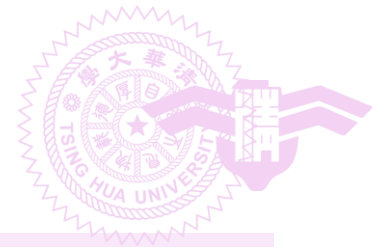


Depth-First Search (DFS)

```
virtual void Graph::DFS() // Driver
{
    visited = new bool[n];
    fill (visited, visited + n, false);
    DFS(0); // start search at vertex 0
    delete [] visited;
}
```

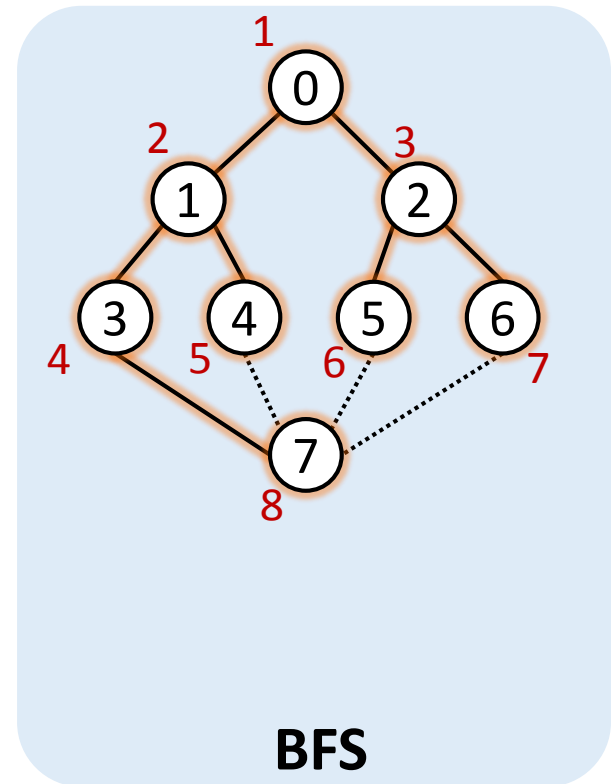
```
virtual void Graph::DFS(const int v)
{
    visited[v] = true;
    for (each vertex w adjacent to v)
        if (!visited[w])
            DFS(w);
}
```

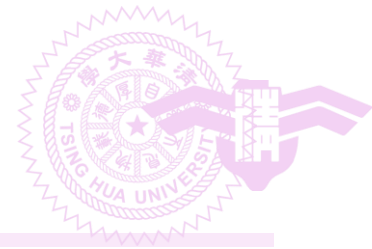




Breadth-First Search

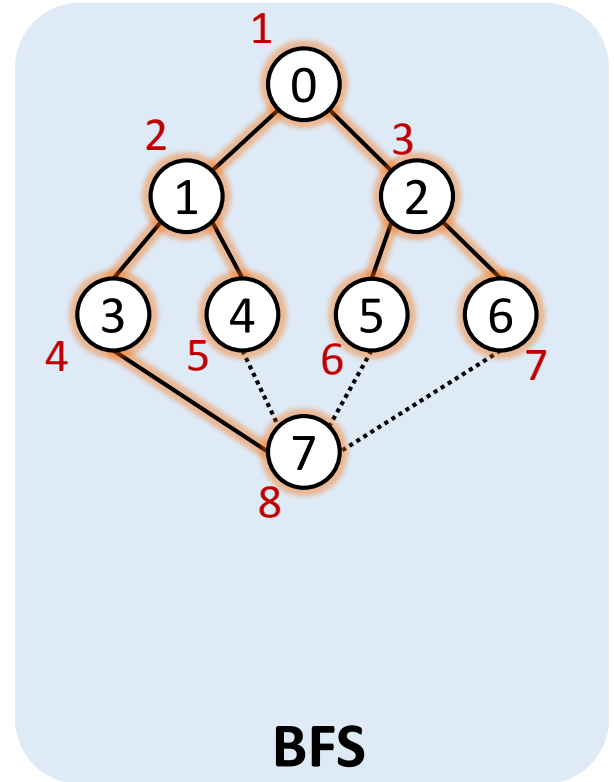
- Begin by visiting the **start vertex** v
- All unvisited vertices adjacent to v are visited
- Unvisited vertices adjacent to these newly visited vertices are then visited, and so on

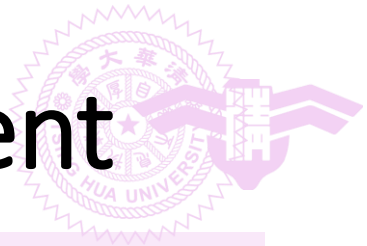




Breadth-First Search

```
virtual void Graph::BFS(int v)
{
    visited = new bool [n];
    fill (visited, visited + n, false);
    visited[v] = true;
    Queue<int> q;
    q.Push (v);
    while (!q.IsEmpty ()) {
        v = q.Front ();
        q.Pop ();
        for (all vertices w adjacent to v)
            if (!visited [w]) {
                q.Push (w);
                visited[w] = true;
            }
    }
    delete [] visited;
}
```





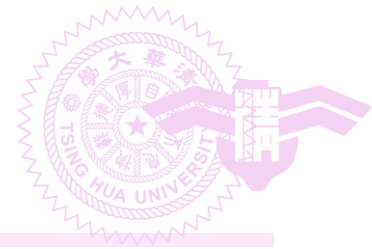
Concept of Connect Component

- Determine whether a graph is **connected**
 - Call DFS or BFS and then determine if there is any unvisited vertex
- Find connected components in a graph
 - Make repeated calls to either DFS(v) or BFS(v)
 - where v is a vertex that has not yet been visited



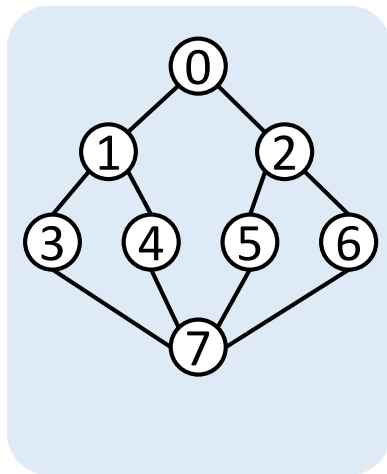
Connect Components

```
virtual void Graph::Components()
{
    visited = new bool [n];
    fill (visited, visited + n, false);
    for (i = 0 ; i < n ; i++){
        if (!visited[i]) {
            DFS(i); // find the component containing i
            OutputNewComponent ();
        }
    }
    delete [] visited;
}
```

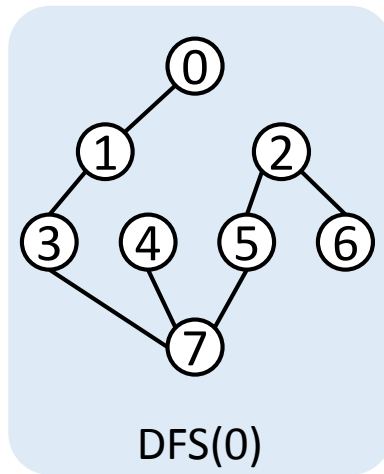


Concept of Spanning Tree

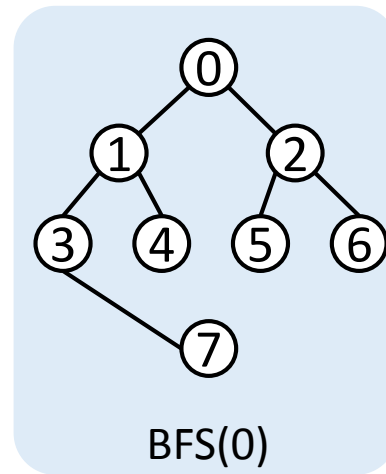
- Any tree consisting solely of edges in G and including all vertices in G is called a **spanning tree**
 - Tree is a **connected** graph **without loops**
 - Graph has multiple spanning trees
 - **Traversing** a graph can produce a spanning tree
 - **Depth-first spanning trees** or **breadth-first spanning trees**



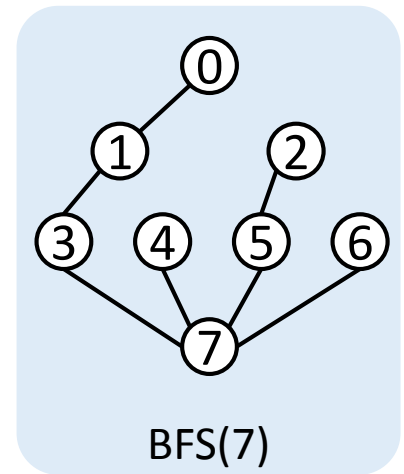
G



DFS(0)

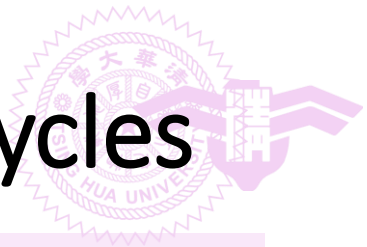


BFS(0)



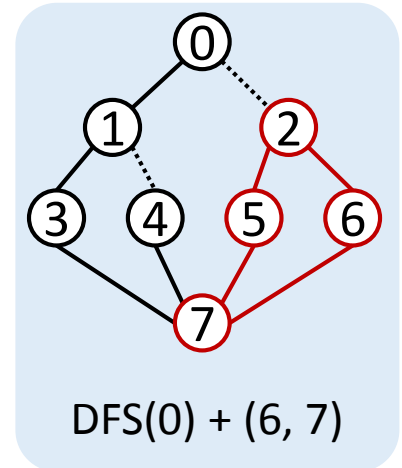
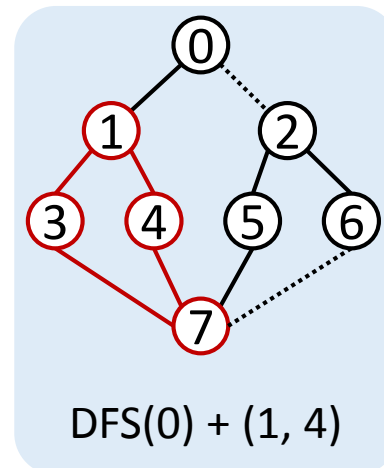
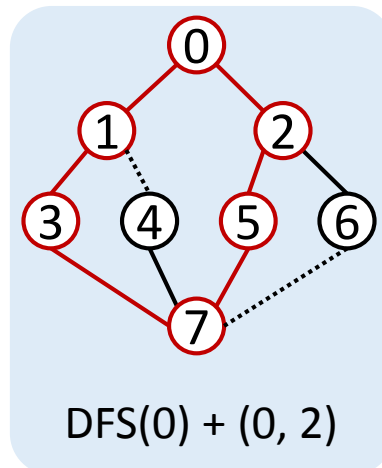
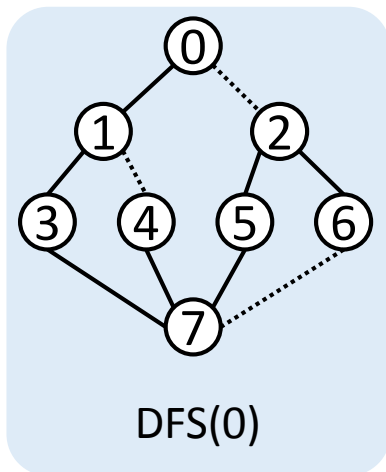
BFS(7)

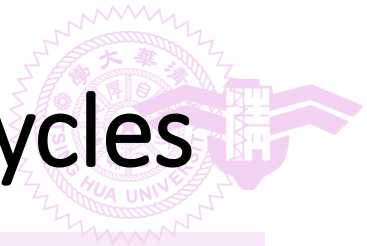
Some spanning trees of G



Spanning Tree \rightarrow Independent Cycles

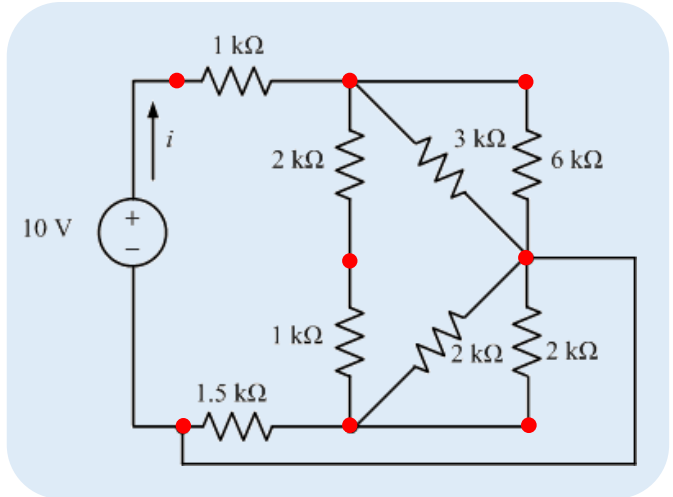
- Introducing a nontree edge (v, w) into a spanning tree produces a cycle
- These cycles are independent
 - Each introduced nontree edge is not contained in any other cycle
 - We cannot obtain any of these cycles by taking a linear combination of the remaining cycles
 - $(\# \text{ of independent cycles}) = (\# \text{ edges}) - (\# \text{ vertices} - 1)$



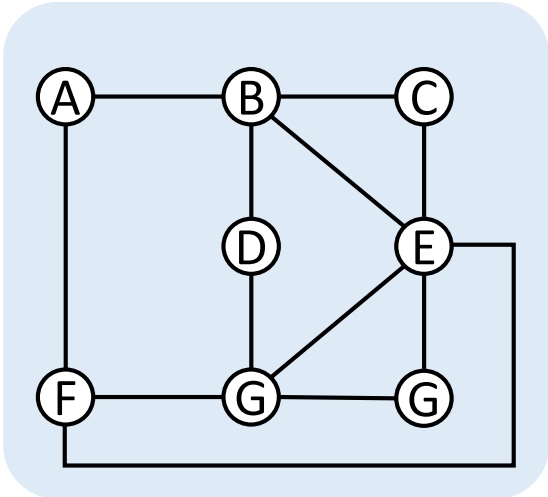


Spanning Tree \rightarrow Independent Cycles

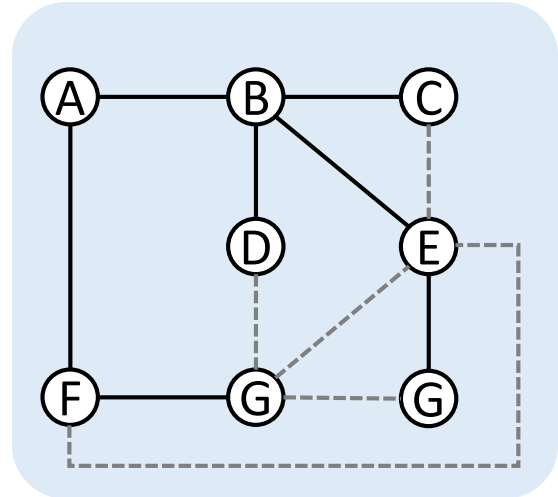
- Producing independent Kirchhoff's voltage equations of an electrical circuit network



Circuit

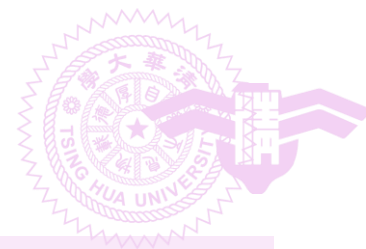


Graph



BFS(A)

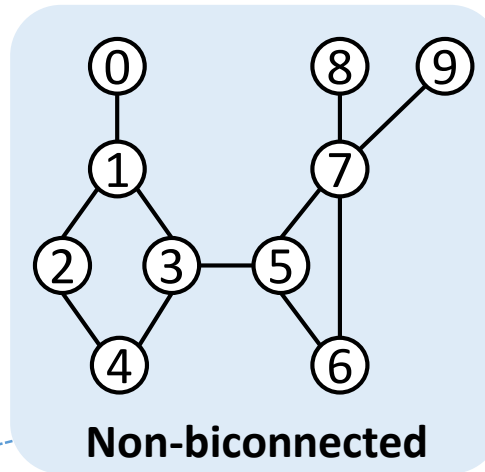
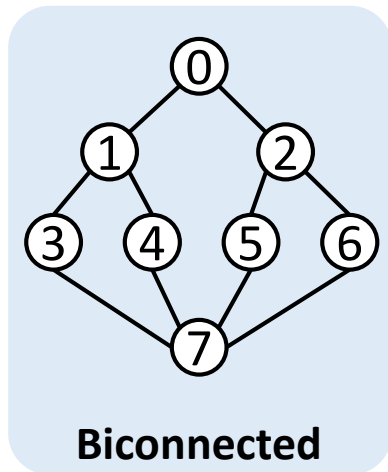
- Equations:
- $V_{BC} + V_{CE} + V_{EB} = 0$
 - $V_{AB} + V_{BD} + V_{DG} + V_{GF} + V_{FA} = 0$
 - ...



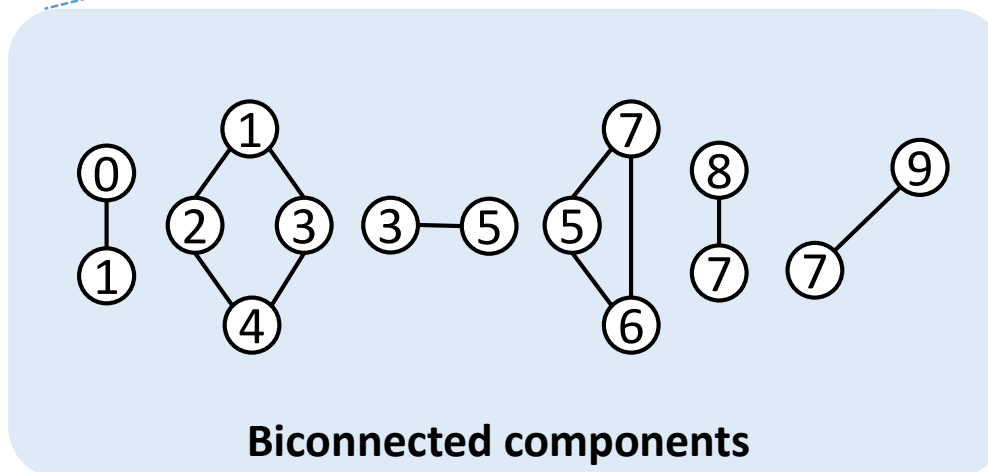
Biconnected Components


- A vertex v of undirected, connected graph G is an **articulation (關節) point** iff
 - Deleting v and all edges incident to v makes G disconnected
- A **biconnected graph** is a connected graph with **no articulation points**
 - No single point of failure
 - A desired property for, say, a communication network
- A **biconnected component** is a **maximal** biconnected subgraph

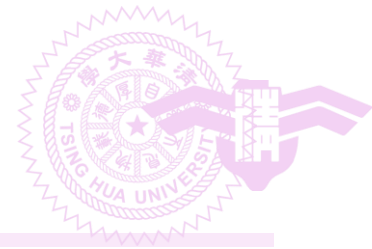
Biconnected Components



Vertices 1, 3, 5, 7 are articulation points

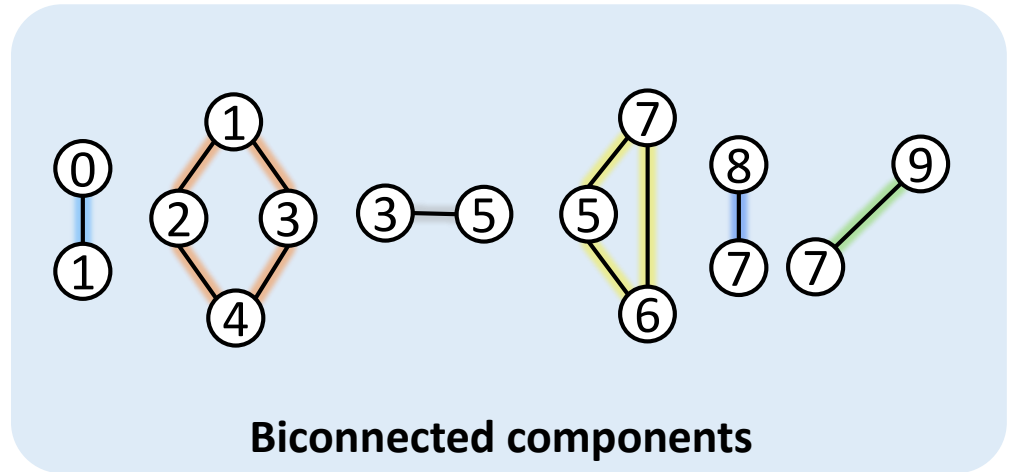
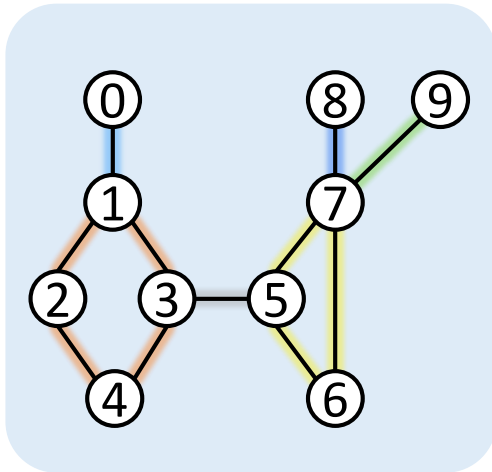


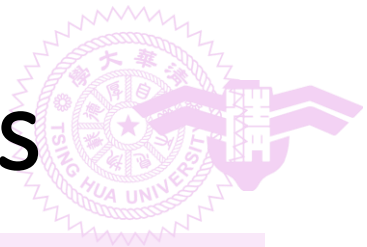

 This is not a biconnected component because it is not **maximal**



Biconnected Components

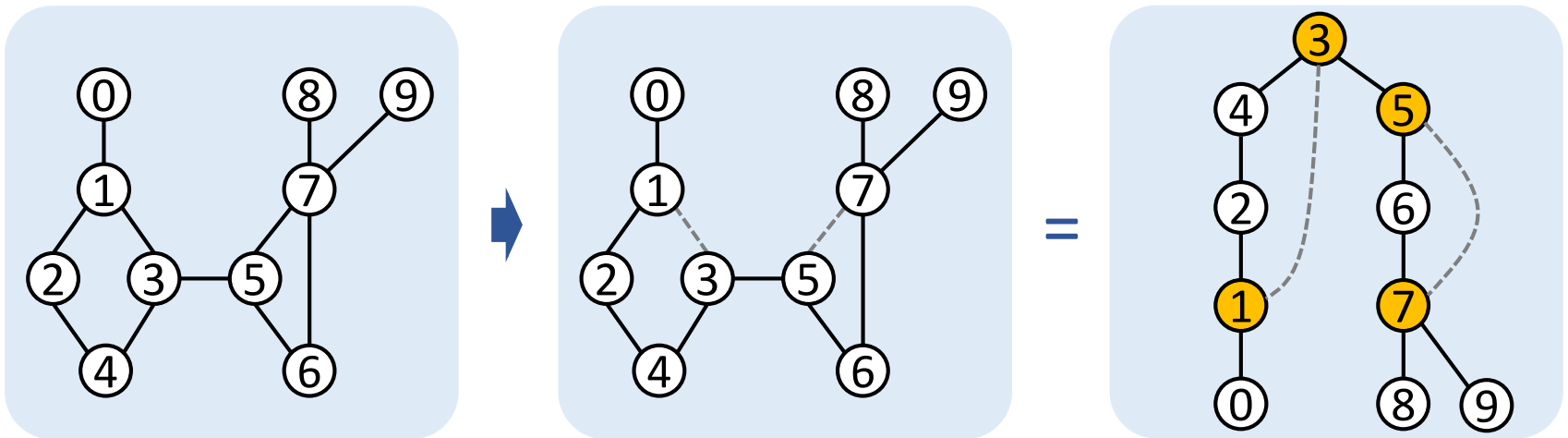
- A binnected graph has just one biconnected component: the whole graph
- Two biconnected components of the same graph can have **at most one common vertex**
 - Therefore, no edges can be in two or more biconnected components
- Biconnected components of G partition the edges of G





Find Biconnected Components

- Depth-first spanning tree can be used to find articulation points, which indicate biconnected components



Graph

DFS(3) spanning tree

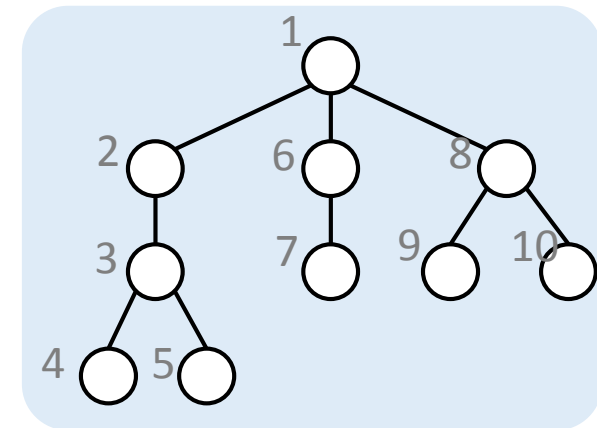
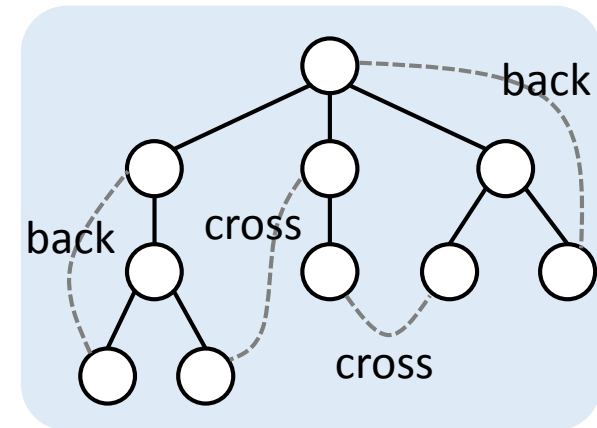
(shaded vertices are articulation points)

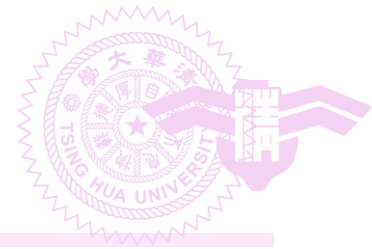
(We can pick any vertex as the root and find the same articulation points)



Depth-First Spanning Trees of Graphs

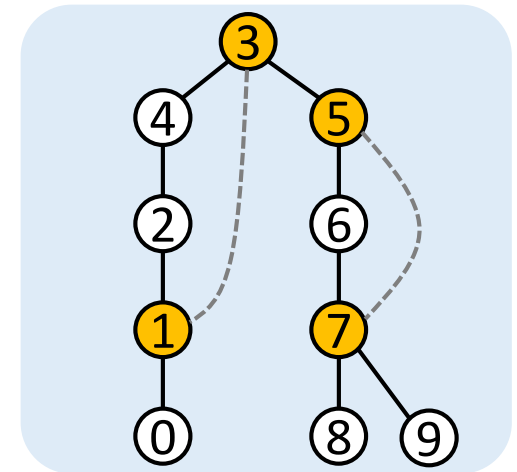
- Nontree edges
 - **Back edge**
 - A nontree edge (u, v) in which either u is an ancestor of v or v an ancestor of u
 - **Cross edge**
 - A nontree edge that is not a back edge
- From the definition of DFS, a graph has **no cross edges** with respect to its depth-first spanning trees
- **Depth-first number, dfn,**
 - The sequence in which the vertices are visited during the DFS

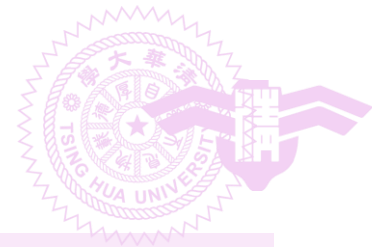




Identifying Articulation Points

- Analyzing any **depth-first spanning tree** of the graph
 - Leaf
 - Never being an articulation point
 - Root
 - Being an articulation point iff it has ≥ 2 children
 - since there are no cross edges among the root's subtrees
 - Other (non-root ,non-leaf) vertex, u
 - Being an articulation point iff u 's **ancestors lacks a non-tree edge to any of u 's subtrees**
 - Without the non-tree edge, u separates the ancestors from the subtrees

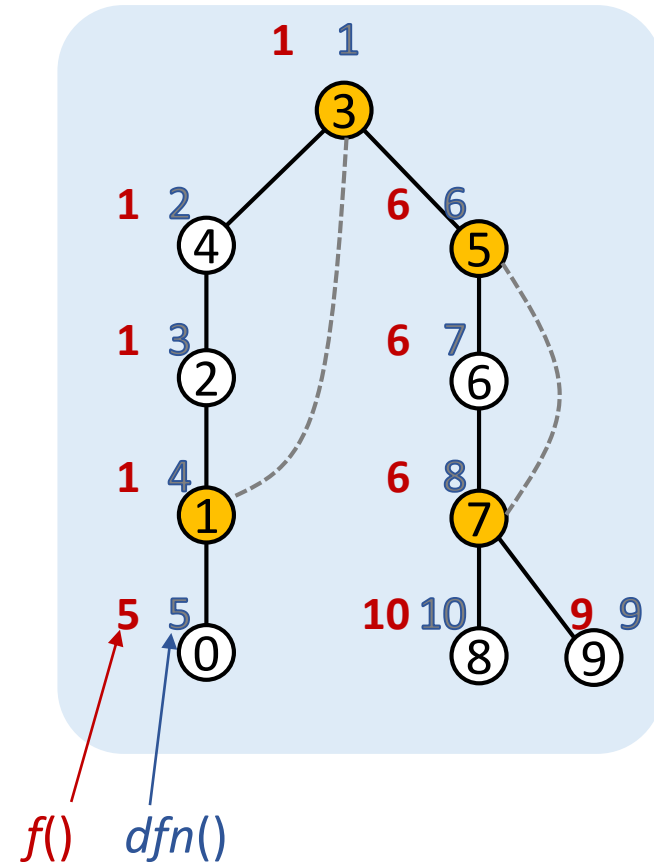




Algorithm

- Technique

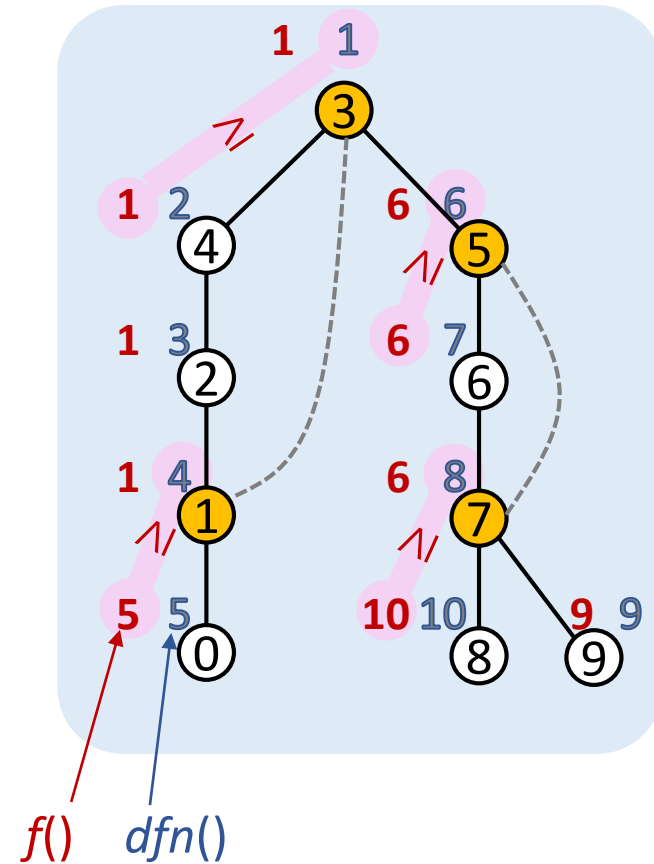
- Find the **lowest** reachable ancestor through descendants and one back edge
- Define $f(w)$ for a vertex w as the minimum of the following values
 - $dfn(w)$
 - $dfn(x \mid (w,x) \text{ is a nontree edge})$
 - $f(w\text{'s children})$





Algorithm

- u has any child w such that $f(w) \geq dfn(u)$
 - \rightarrow u 's ancestors lacks a non-tree edge to any of u 's subtrees
 - \rightarrow u is an **articulation point**
- Conventional name of $f()$ is *low()*



Computing dfn, low, and Outputting Biconnected Components

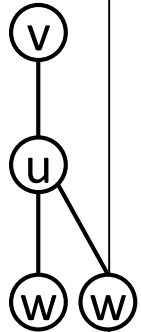


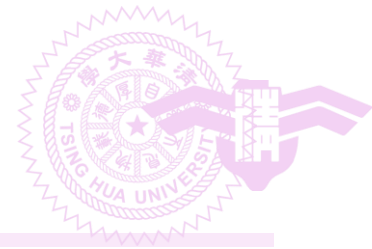
```
virtual void Graph::Biconnected()
{
    num = 1;           // num is an int data member of Graph
    dfn = new int[n]; // dfn is declared as int* in Graph
    low = new int[n]; // low is declared as int* in Graph
    fill(dfn, dfn + n, 0);
    fill(low, low + n, 0);
    rBiconnected(0, -1);
    delete [] dfn;
    delete [] low;
}
```

Computing dfn, low, and Outputting Biconnected Components



```
void Graph::rBiconnected (const int u, const int v)
{
    dfn[u] = low[u] = num++;
    for (each vertex w adjacent from u){ // (u, w)
        if ((v != w) && (dfn[w] < dfn[u]))
            add (u, w) into stack s;
        if (dfn[w] == 0) { // w is an unvisited vertex, a child
            rBiconnected(w, u);
            low[u] = min(low[u], low[w]);
            if (low[w] >= dfn[u]) { // u is an articulation point
                cout << "New Biconnected Component:" << endl;
                do {
                    delete an edge from the stack s;
                    let this edge be (x, y);
                    cout << x << "," << y << endl;
                } while ( (x, y) and (u, w) are not the same edge)
            }
        }
        else if (w != v)
            low[u] = min(low[u], dfn[w]); // back edge
    }
}
```





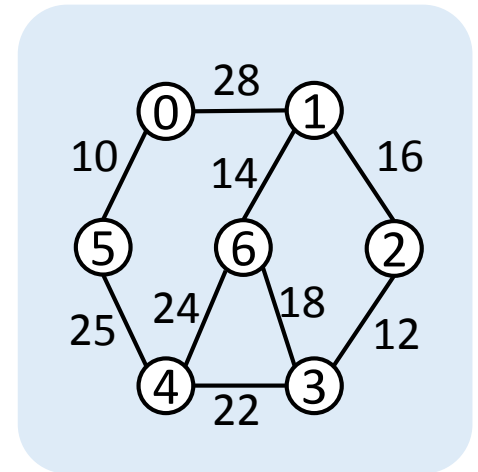
Outline

- 6.1 The graph abstract data type
- 6.2 Elementary graph operations
- **6.3 Minimum-cost spanning trees (MSTs)**
- 6.4 Shortest paths and transitive closure
- 6.5 Activity networks



Minimum-Cost Spanning Trees (MSTs)

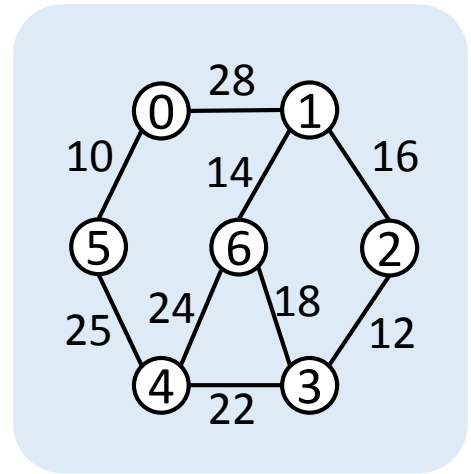
- An graph can have many spanning trees
- For a weighted, connected, and undirected graph
 - We define the **cost** of a spanning tree is the sum of the weights of the edges in the spanning tree
- We may want to **minimize the cost**
 - Possible applications: road construction, circuit layout, internet routing

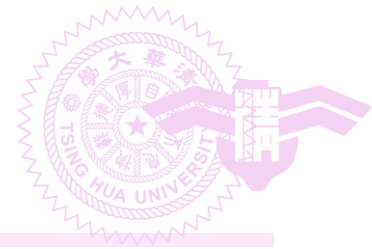




Minimum-Cost Spanning Trees

- Three **greedy** methods
 - Kruskal's
 - Prim's
 - Sollin's
- In a greedy method, we construct an optimal solution in stages
 - At each stage, we make the best decision possible at the time
 - (e.g., the least-cost edge is chosen for building a minimum-cost spanning tree)
 - We do not change this decision later
- Greedy strategy can lead to MST construction



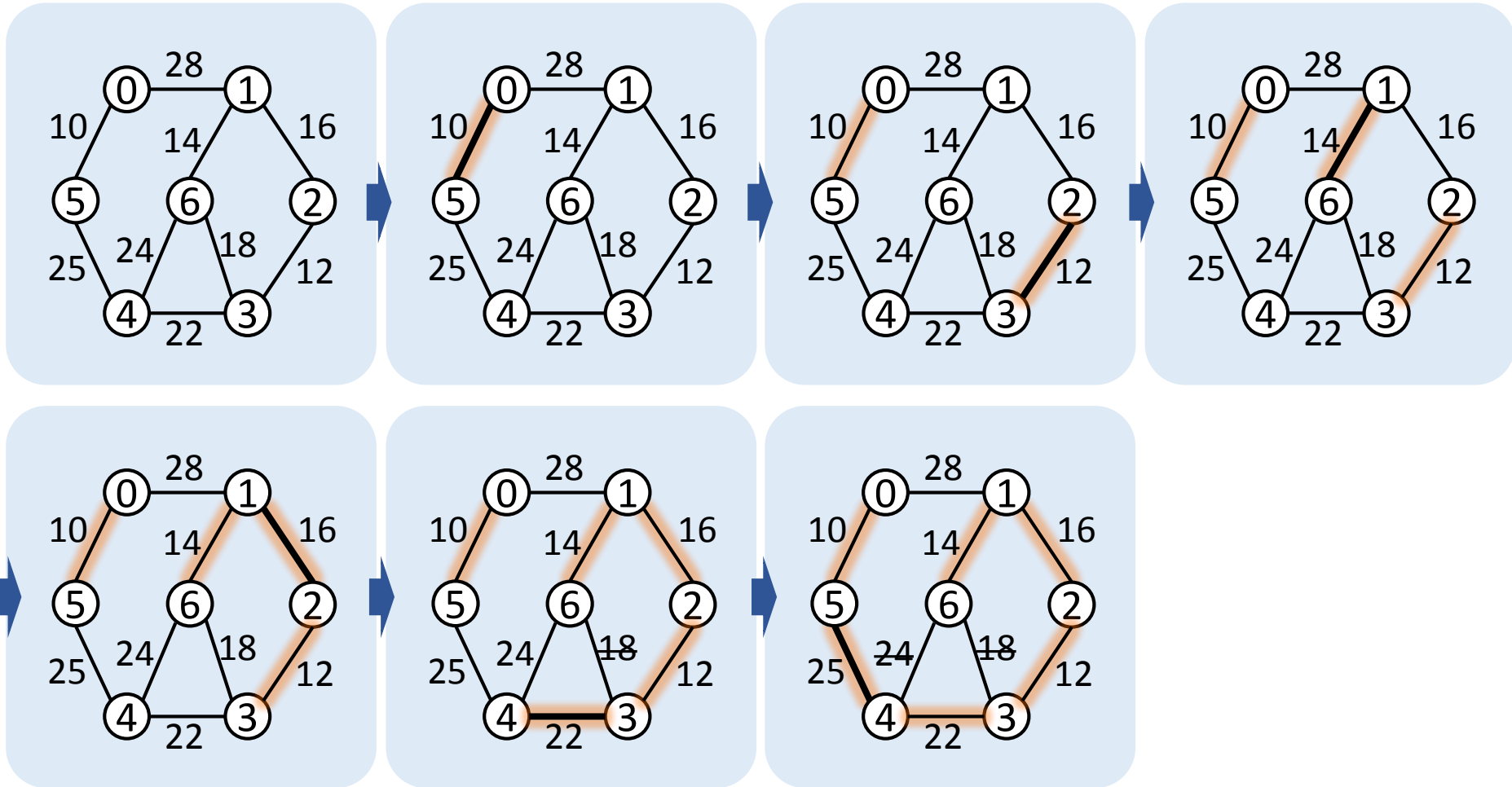


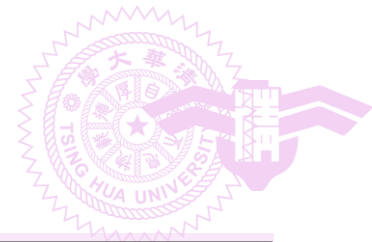
Kruskal's Algorithm

- Create an empty graph, T
- Sort edges according to weights
- Add edges to T one at a time
 - The **least-cost** edge that **does not form a cycle** with T 's edges
- Exactly $n-1$ edges are added, where n is the number of vertices



Kruskal's Example





Kruskal's Algorithm

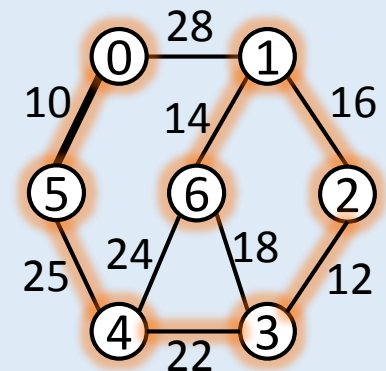
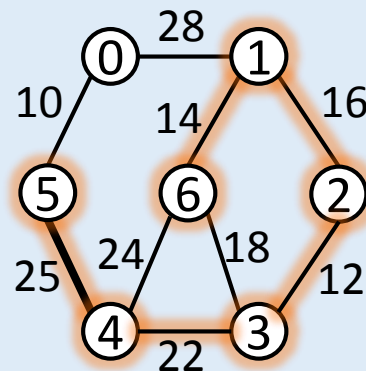
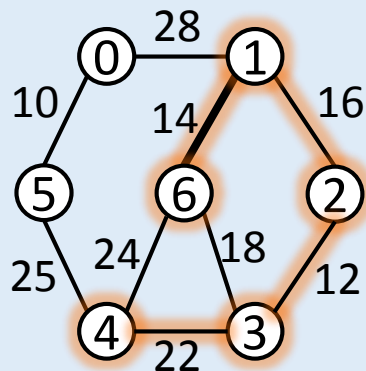
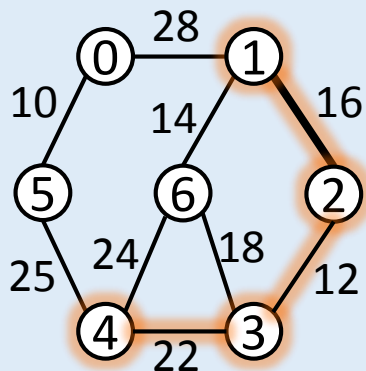
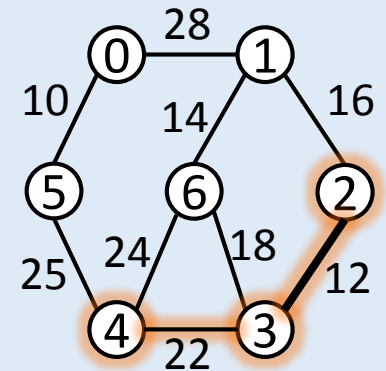
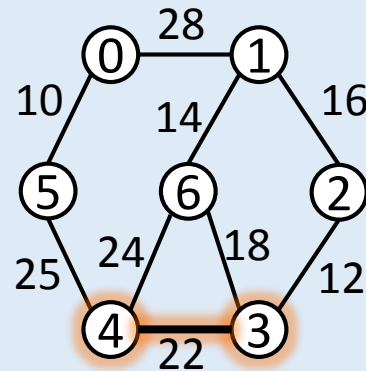
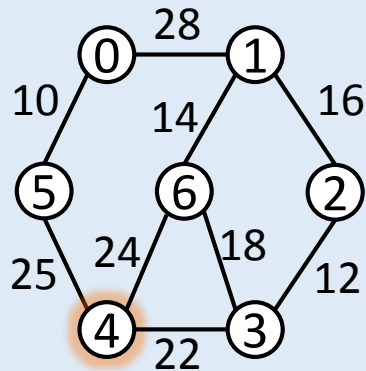
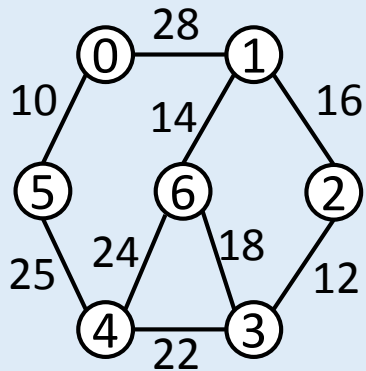
```
T =  $\emptyset$ ;
while ( ( T contains less than n-1 edges) &&
        (E is not empty) ) {
    choose an edge (v, w) from E of lowest cost;
    delete (v, w) from E;
    if ( v and w belong to diff. sets){ // no loop
        add (v, w) to T;
        merge v's and w's sets;
    }else{
        discard (v, w);
    }
}
if ( T contains fewer than n-1 edges)
    cout << "no spanning tree" << endl;
```

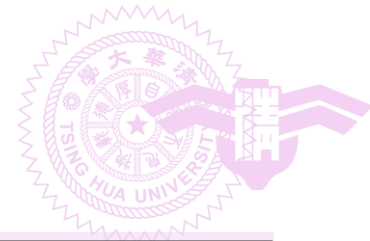


Prim's Algorithm


- Begin with an empty tree, T
- Sort edges according to weights
- Add to T any **vertex** of the graph
- Add **vertices** to T one at a time
 - The vertex is adjacent to a vertex in T
 - The vertex corresponds to the **least-cost** edge

Prim's Example





Prim's Algorithm

```
if (G has at least one vertex) 
    cout << "no spanning tree" << endl;

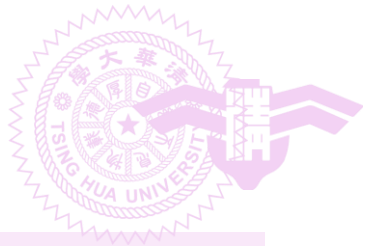
TV = {0}; // start with vertex 0 and no edges
for (T =  $\emptyset$ ; T contains less than n-1 edges; add (u, v) to T)
{
    Let (u, v) be a least-cost edge with u in TV && v not in TV;
    if (there is no such edge)
        break;
    add v to TV;
}

if ( T contains fewer than n-1 edges)
    cout << "no spanning tree" << endl;
```

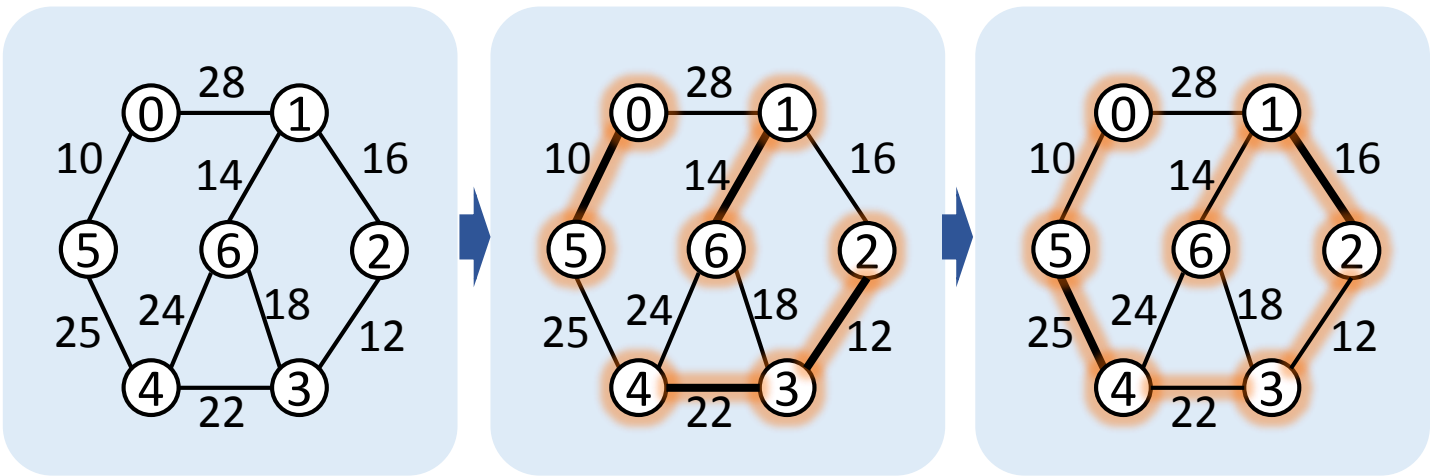


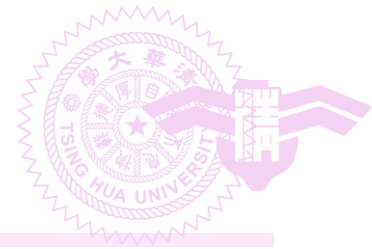

Sollin's Algorithm

- Create n subgraphs, each subgraph having a single vertex
- Sort edges according to weights
- Add edges to each subgraph one at a time
 - The **least-cost** edge that **does not form a cycle** with each subgraph 's edges
 - Duplicate edges are discarded
- A total of exactly $n-1$ edges are added, where n is the number of vertices



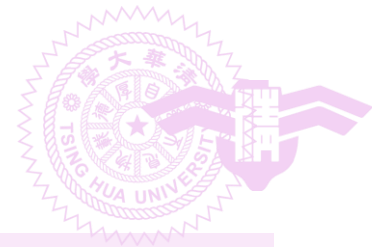
Sollin's Example





Outline

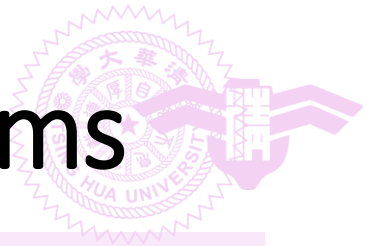
- 6.1 The graph abstract data type
- 6.2 Elementary graph operations
- 6.3 Minimum-cost spanning trees
- **6.4 Shortest paths and transitive closure**
- 6.5 Activity networks



Shortest Path Problem

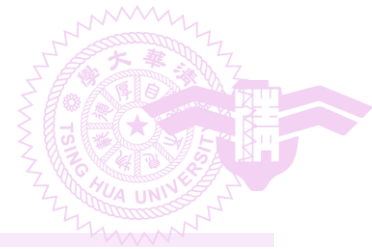
- Let's consider a GPS device using **graph** data structures to represent the highway structures of a state
 - **Vertices** representing **cities**
 - **Edges** representing **sections of highway**
 - **Edge weights** representing **lengths** of the highway sections
- Important questions
 - Is there a path from A to B
 - What is the **shortest path** from A to B





Various Flavors of Path Problems

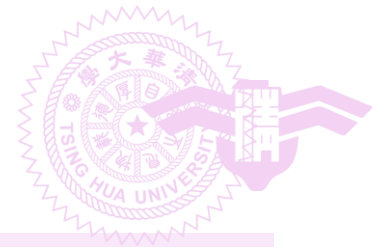
- Edge costs
 - **Non-negative** costs (e.g., traveling distance, spent time)
 - General costs (e.g., spent/obtained fuels)
 - Number of sources and destinations
 - Single source single destination
 - Single source all destinations
 - All sources single destination
 - All pairs
 - Textbook covers
 - Single source all destinations
 - All pairs
- $\left. \begin{array}{l} \text{Single source all destinations} \\ \text{All pairs} \end{array} \right\} \times \left\{ \begin{array}{l} \text{Non-negative costs} \\ \text{General costs} \end{array} \right.$



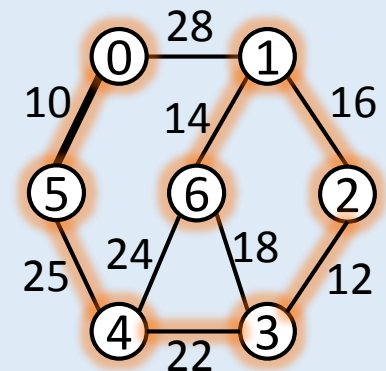
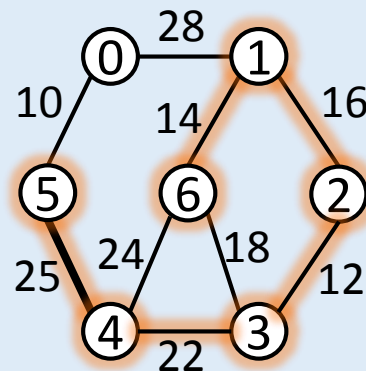
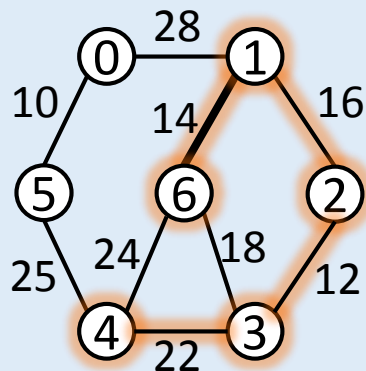
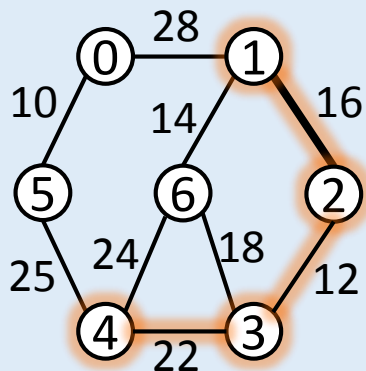
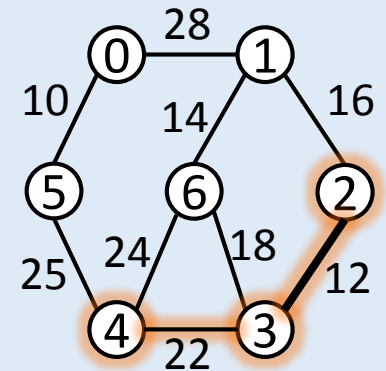
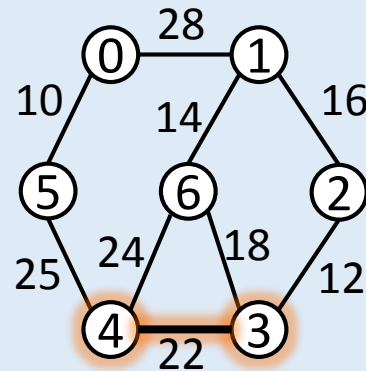
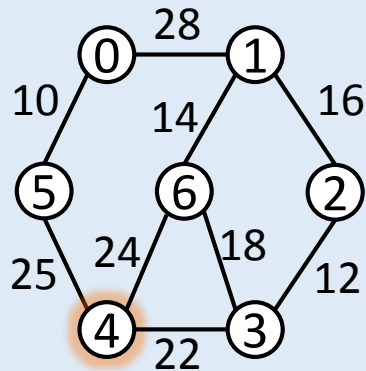
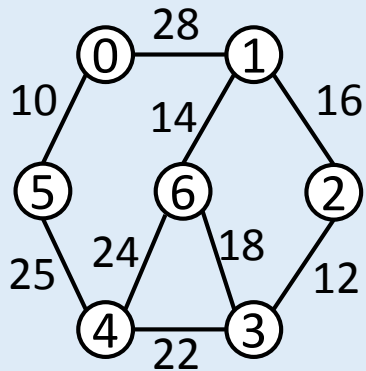
Comparisons

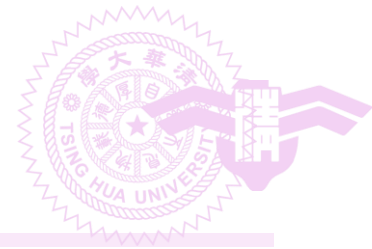
- Prim's (for MST)
 - Dijkstra's (for shortest paths)
- } Strategy: greedy

- Bellman-Ford (for shortest paths)
 - All pairs (for shortest paths)
- } Strategy: table

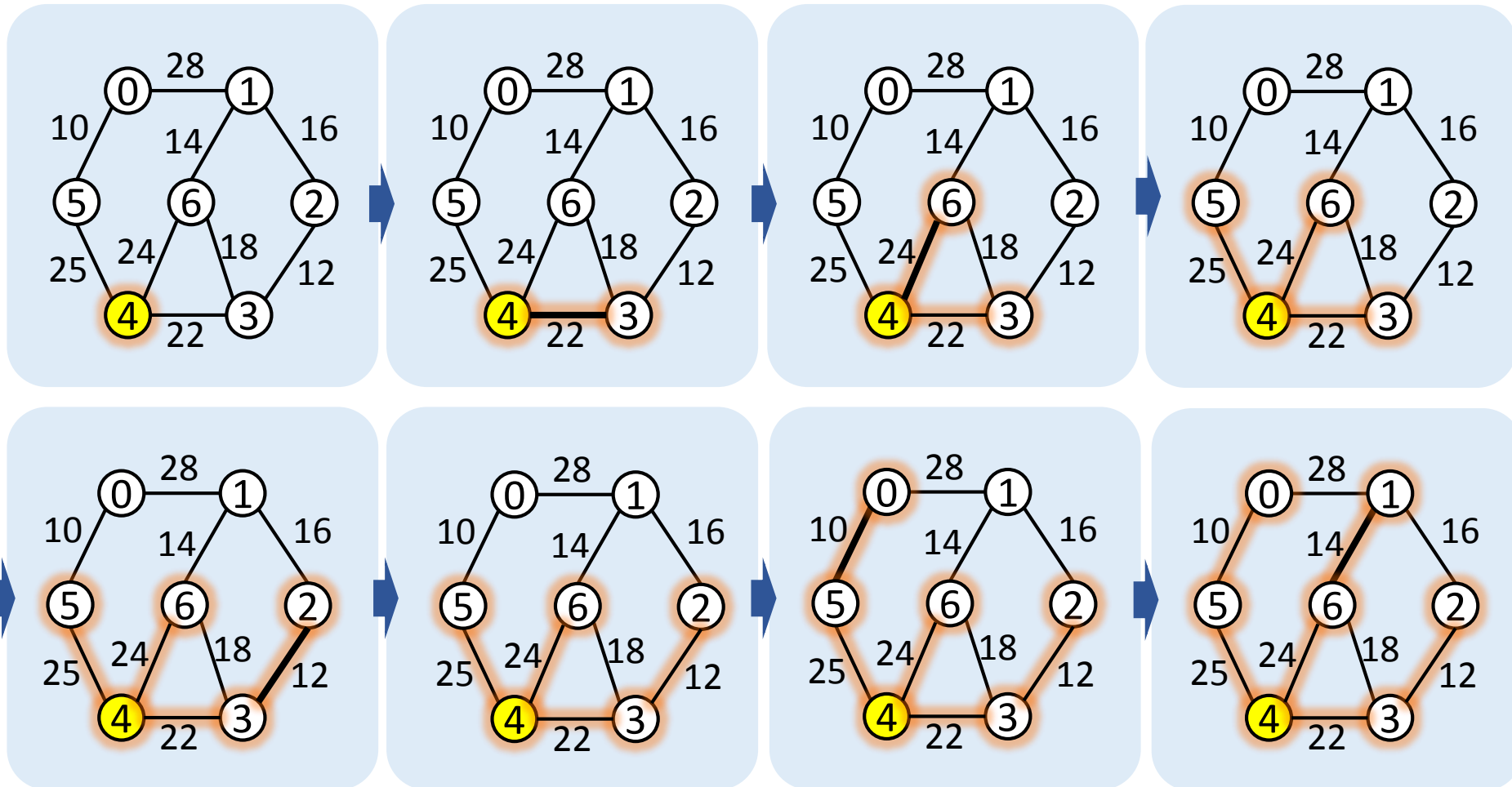


Prim's Example





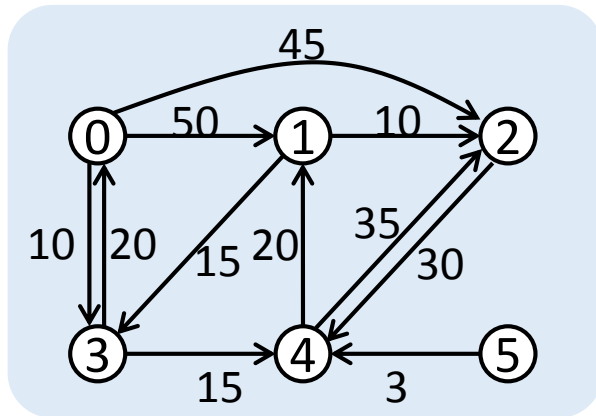
Dijkstra's Example





Single Source, All Destinations, and Nonnegative Costs

- Input
 - A directed graph $G = (V, E)$
 - $\text{Length}(i, j)$ for the edges of G
 - A source vertex v
- Output
 - Determine a shortest path from v to each of the remaining vertices of G **in non-decreasing length order**



Graph



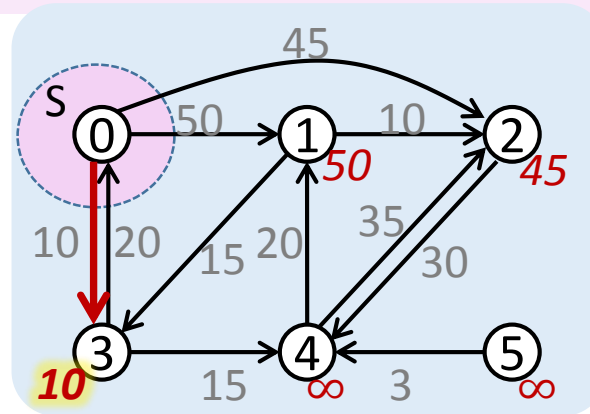
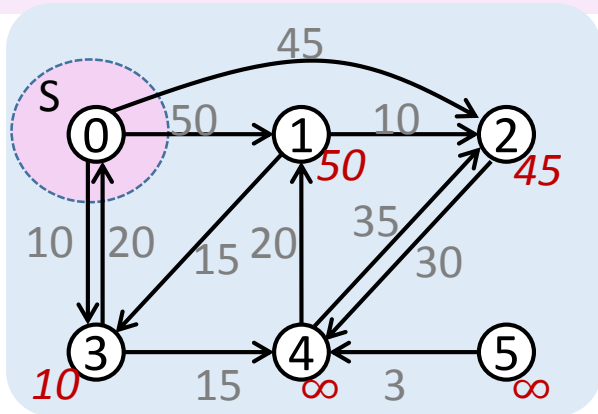
Path	Length
0, 3	10
0, 3, 4	25
0, 3, 4, 1	45
0, 2	45

All-destinations shortest path from 0

Non-decreasing

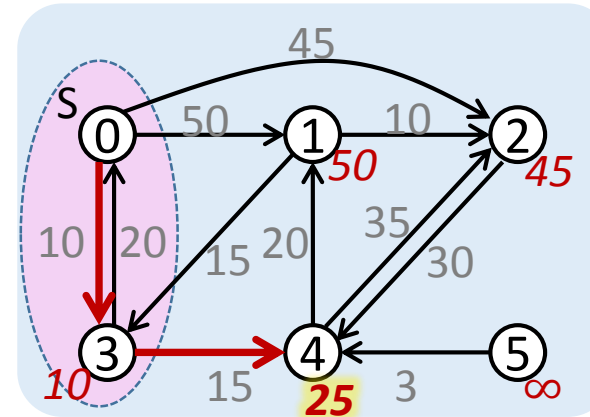
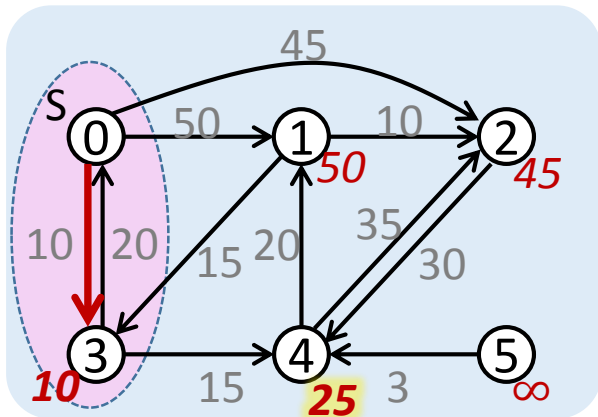


Dijkstra's Algorithm Example



Path	Length
0, 3	10

Vertex 3 has the least-cost dist , i.e., 10.
So, output the 0-3 path.



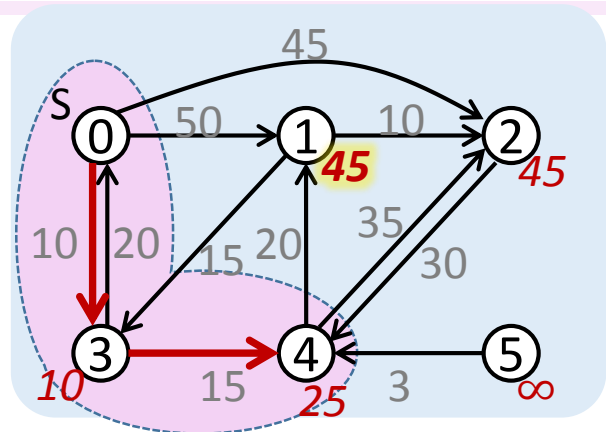
Path	Length
0, 3	10
0, 3, 4	25

Include 3 in to S and update
vertices adjacent from 3.

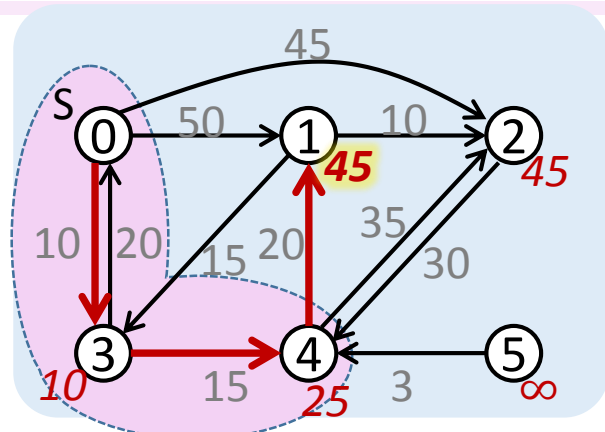
Vertex 4 has least-cost dist , i.e., 25.
Output the 0-4 path.



Dijkstra's Algorithm Example

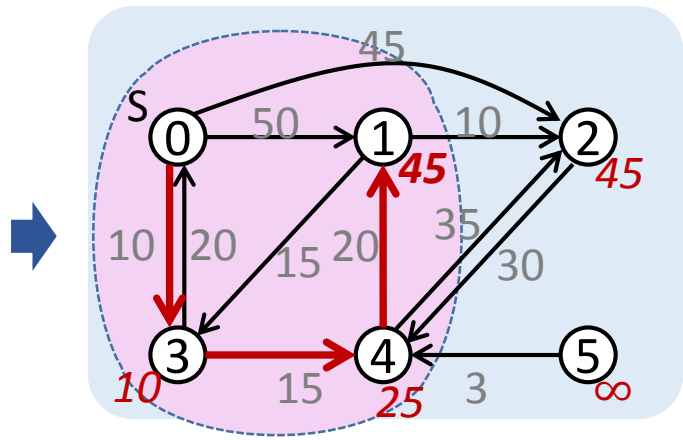


Include 4 in to S and update vertices adjacent from 4.

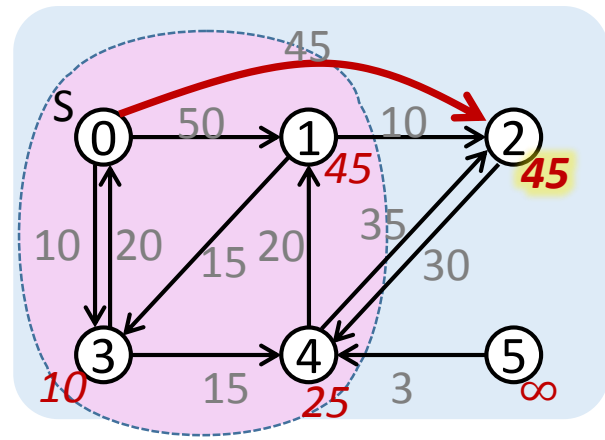


Both vertices 1 and 2 have the least-cost dist. Output one of them, e.g., 1.

Path	Length
0, 3	10
0, 3, 4	25
0, 3, 4, 1	45



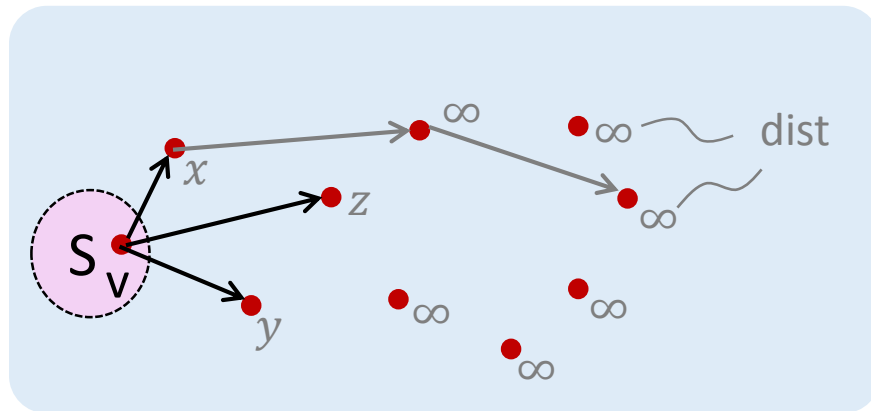
Include 1 in to S and update vertices adjacent from 1.



Include 1 in to S and update vertices adjacent from 1.

Path	Length
0, 3	10
0, 3, 4	25
0, 3, 4, 1	45
0, 2	45

Dijkstra's Algorithm

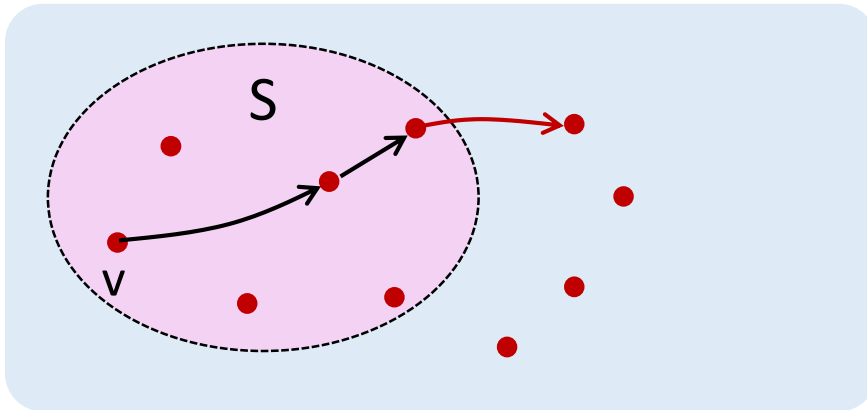


Edsger W. Dijkstra

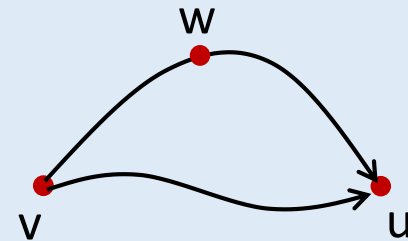
- Dutch computer scientist
- Turing award recipient
- "Dijkstra" pronounces similar to /dye-k-stla/

- **S**: A set of vertices to which the shortest paths have already been found
 - $S = \{v\}$ in the beginning
- **dist[u]**: Shortest distance from v , through vertices in S , to a vertex u not in S
 - $\langle v, u \rangle$ exists \rightarrow $\text{dist}[u] = \text{edge weight}$
 - $\langle v, u \rangle$ doesn't exist $\rightarrow \text{dist}[u] = \infty$
 - $\text{dist}[v]$ is considered as 0

Dijkstra's Algorithm

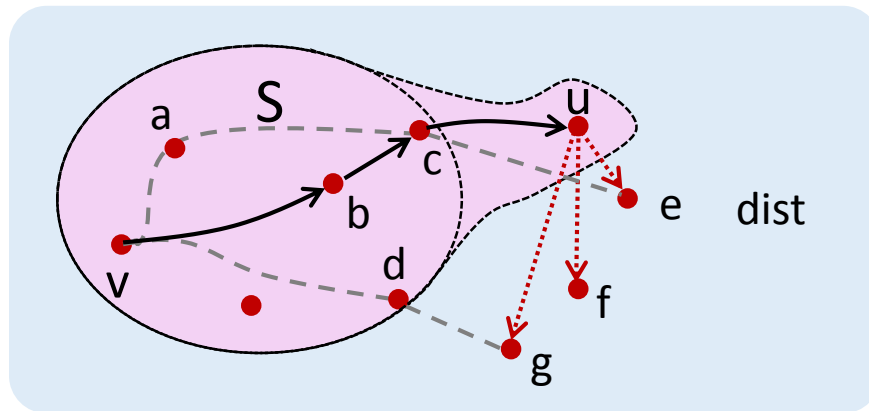


- When S contains $n \geq 1$ vertices
- A next shortest path must contain only vertices in S plus the destination
 - There may be multiple equal-length shortest path. At least one of them is so



- Let u be the destination of a next shortest path
- Assume said path contains an intermediate vertex w not in S
 - The length of the v - w path is no greater than the v - u path
 - u should not be the destination of a next shortest path ($\rightarrow \leftarrow$)

Dijkstra's Algorithm



dist[u]: Shortest distance from v , through vertices in S , to a vertex u not in S

- Greedy: among vertices not in S , find a vertex u with the lowest $\text{dist}[]$
 - u becomes a new member of S
- Keep $\text{dist}[]$ updated
 - u may lower $\text{dist}[]$ of vertices that are not in S and adjacent from u
- The algorithm stops when S contains all n vertices



Dijkstra's Algorithm

```
void MatrixWDigraph::ShortestPath(const int n, const int v)
{
    for (int i = 0; i < n ; i++) { // initialization
        s[i] = false;           // the set, S
        dist[i] = length[v][i]; // dist[]
    }
    s[v] = true;
    dist[v] = 0;
    for (i = 0; i < n-1 ; i++) { // n-1 shortest paths from v
        choose u that is not in S and has smallest dist[u];
        s[u] = true; // u becomes a member of S
        for (each <u, w> in the graph) // update dist[w]
            if (!s[w] && (dist[u] + length[u][w]) < dist[w])
                dist[w] = dist[u] + length[u][w];
    }
}
```

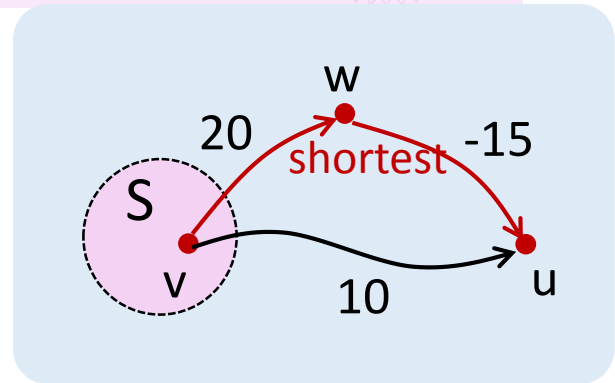
"choosing the smallest dist[u]" is typically in $O(n)$.
So, the overall complexity is $O(n^2)$



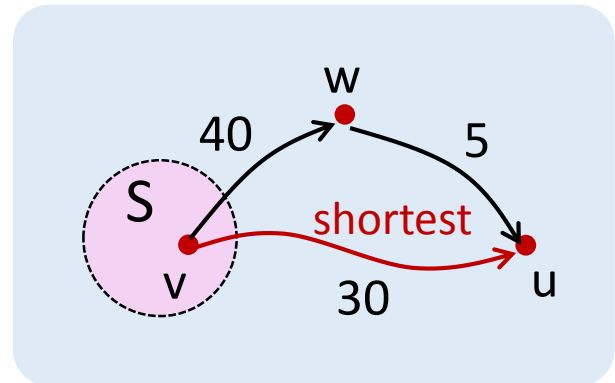
Single Source, All Destinations, and General Costs

- All edge costs (positive, negative, zero) are permitted
 - A more general (also more difficult) problem
 - Dijkstra's greedy strategy does not work here

- Offsetting all edge costs does not help
 - Paths consist of different number of edges
 - Different offset amounts change the length order of paths



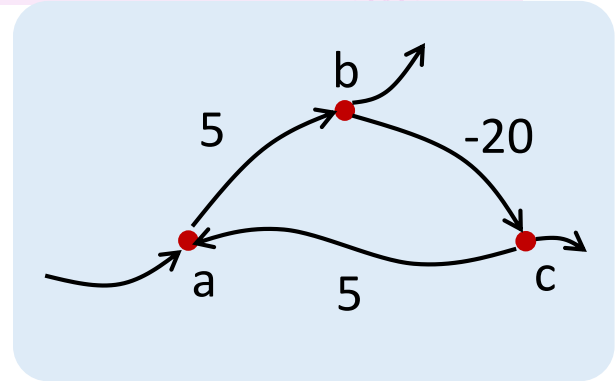
offset by 20





Single Source, All Destinations, and General Costs

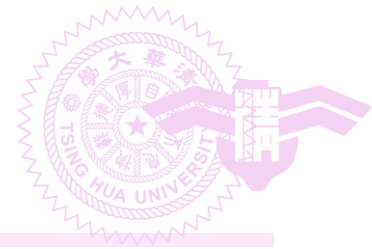
- **Cycles with negative length** are not permitted
 - Otherwise, a cycle produces a path with $-\infty$ cost
 - e.g., ... a-b-c - a-b-c- ...
- A shortest path **must exist** and has **at most $n-1$ edges (i.e., n vertices)**
 - Paths with more than n vertices must contain a cycle
 - Cycles do not lead to shorter paths
 - With at most n vertices, there are a finite number of possible paths
 - The shortest one must exist



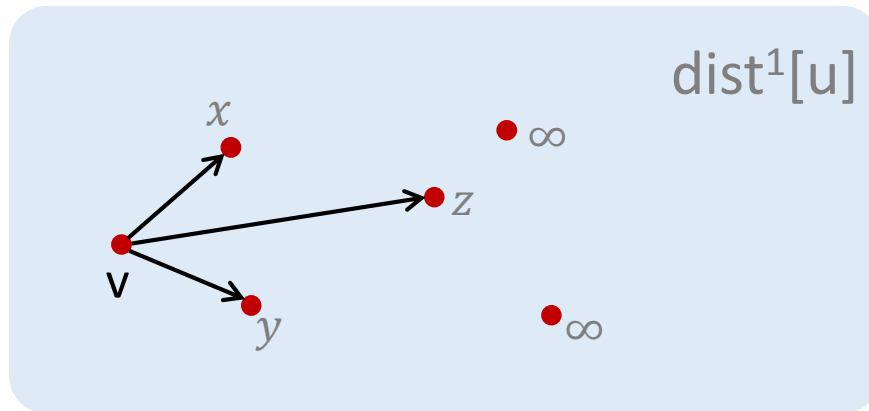


Bellman-Ford Algorithm Concept

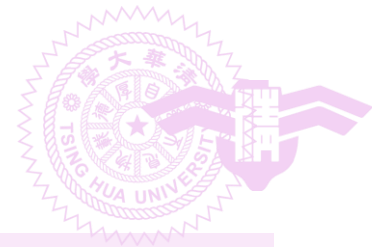
$dist^l[u]$	目標1	目標2	目標3	目標n-1
shortest path with #edge =1	這行可從edge cost直接得知			
shortest path with #edge ≤ 2				↓
shortest path with #edge ≤ 3				
...				
shortest path with #edge $\leq n-1$	這行是我們要的結果			
	#edge >n-1 的 path cost 並不會更低			



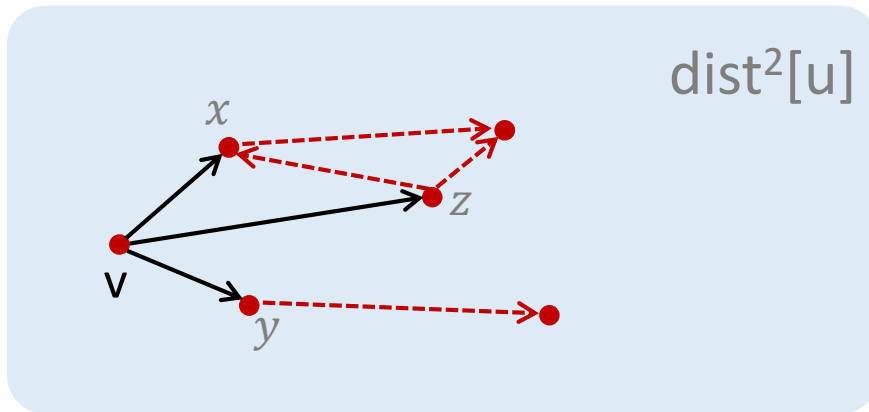
Bellman-Ford Algorithm



- $\text{dist}^l[u]$: Length of a shortest path from v to u with the number of edges $\leq l$
 - $\text{dist}^1[u]$
 - = edge weight if $\langle v, i \rangle$ exists
 - = ∞ if $\langle v, i \rangle$ doesn't exist
 - $\text{dist}^{n-1}[u]$ for all u is our needed results

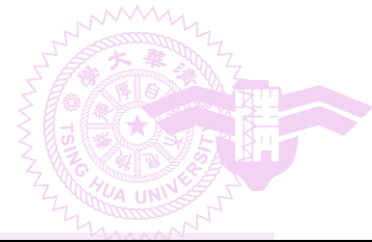


Bellman-Ford Algorithm

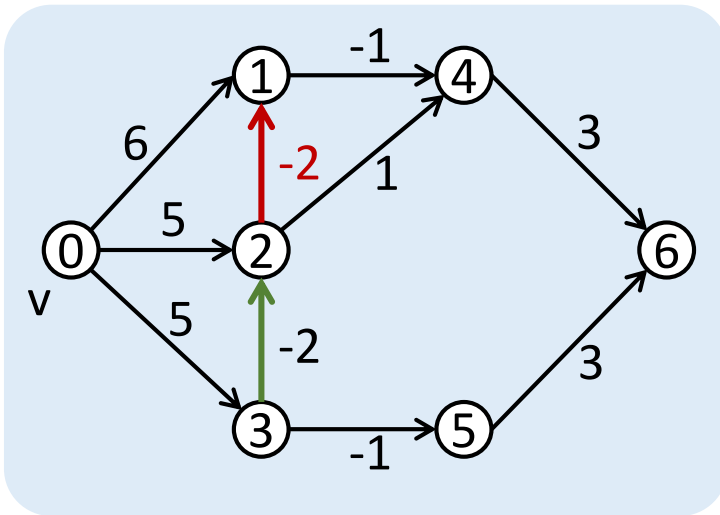


- dist²[x] = min of
- dist¹[x]
 - dist¹[y] + cost(y, x)
 - dist¹[z] + cost(z, x)
 - dist¹[...] + cost(..., x)

- Calculate **dist**^k[u] from **dist**^{k-1}[u], k = 2~(n-1)
 - v-u shortest path with at most k edges, k>1, **dist**^k[u] is the minimum of
 - **dist**^{k-1}[u]
 - (**dist**^{k-1}[i] + length(<i, u>)) for all <i, u>

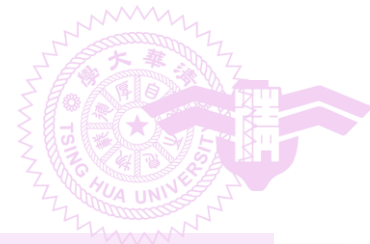


Bellman-Ford Example



k	dist ^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

- Optimization
 - Updating **dist[]** in-place
 - Use only one array for **dist¹[], dist²[]** ...

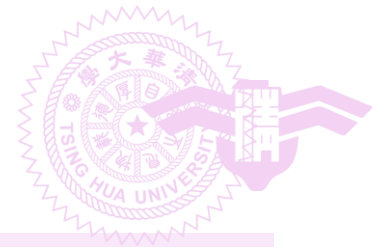


Bellman-Ford Algorithm

```
void MatrixWDigraph::BellmanFord(const int v)
{ // n is the number of vertices
  // in-place update for dist[] is used
  for (int i = 0; i < n ; i++)
    dist[i] = length[v][i]; // dist1[] initialization

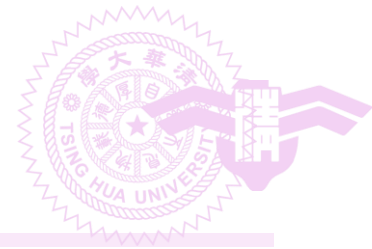
  for (int k = 2; i <= n-1 ; k++) // dist2 ~ dist(n-1)
    for (each u, u != v)
      for (each <i, u> in the graph)
        if (dist[u] > dist[i] + length[i][u])
          dist[u] = dist[i] + length[i][u];
}
```

- "for(each u)" and "for (each <i, u>)" together is $O(n^2)$ for an adjacency matrix and is $O(e)$ for an adjacency list.
- The overall complexity is $O(n^3)$ for an adjacency matrix and is $O(ne)$ for an adjacency list.



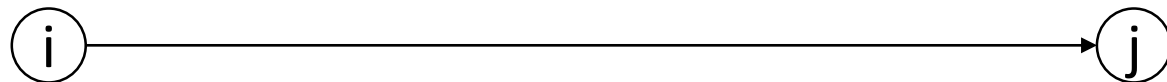
All Pairs and General Costs

- Viable approaches
 - Perform n Bellman-Ford algorithms
 - $O(n^4)$ if an adjacency matrix is used
 - $O(n^2e)$ if an adjacency list is used
- There is an $O(n^3)$ all pair shortest path algorithm
 - Suitable for a dense graph with e being several folds of n

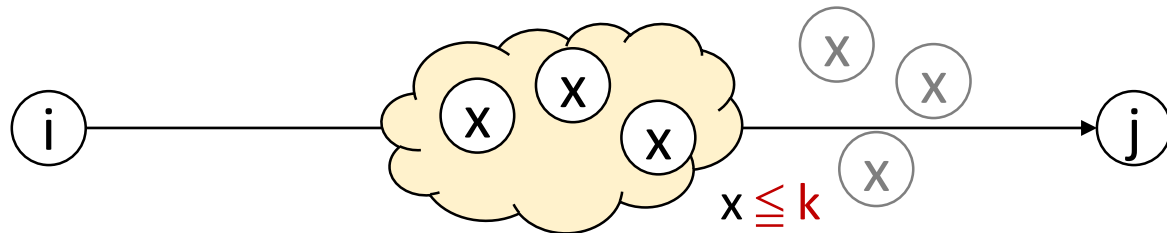


All-Pair Shortest Path Algorithm Concept

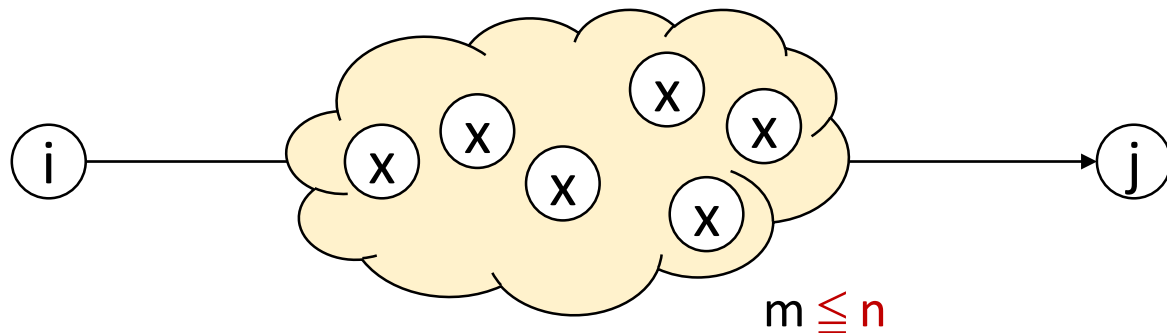
- For each (i, j) pair
 - Shortest path **without an** intermediate vertex

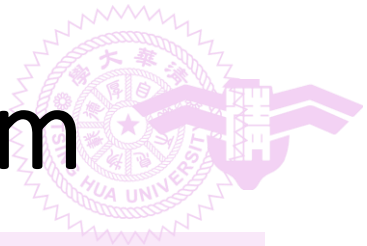


- Shortest path **with some restricted** intermediate vertices



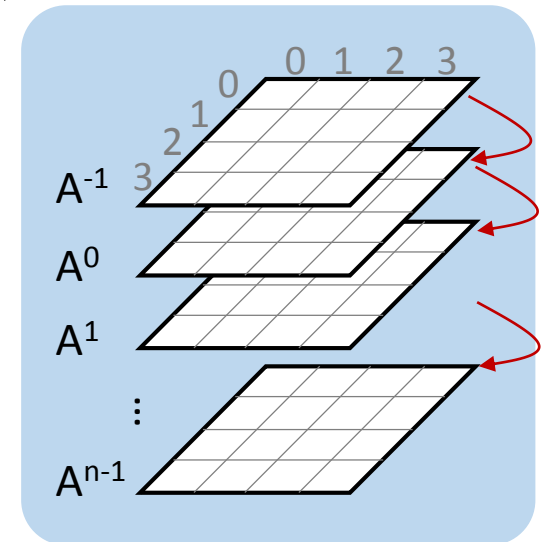
- Shortest path **with any** intermediate vertices

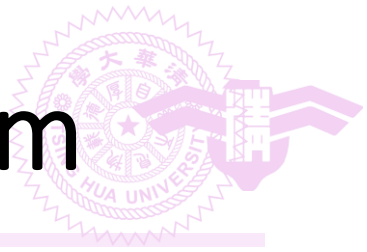




All-Pair Shortest Path Algorithm

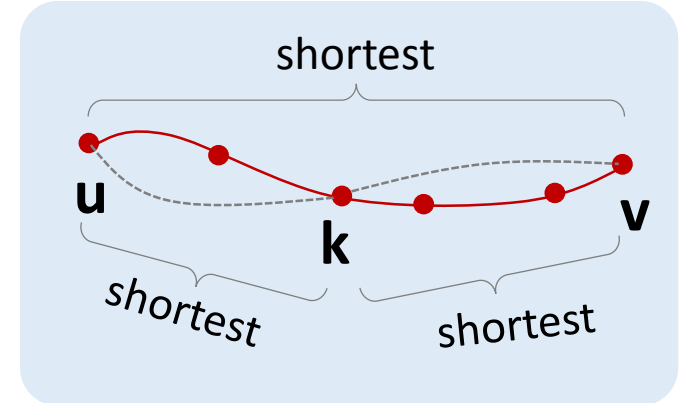
- Define $A^k[i][j]$
 - length of the shortest path from i to j going through no intermediate vertex of index greater than k
 - $A^{-1}[i][j]$ is just the length of the edge $\langle i, j \rangle$
 - $A^{n-1}[i][j]$ is our needed results
- Calculate $A^k[i][j]$ based on $A^{k-1}[i][j]$
 - $A^k[i][j]$ is the minimum of the following
 - $A^{k-1}[i][j]$
 - $A^{k-1}[i][k] + A^{k-1}[k][j]$





All-Pair Shortest Path Algorithm

- Given
 - ShortestPathCost(u, k)
 - ShortestPathCost(k, v)
- If k is on a shortest path from u to v
 - ShortestPathCost(u, v)
= ShortestPathCost(u, k) + ShortestPathCost(k, v)
 - Proof: If there were another path, $(u, k, v)'$, with an even lower cost
 - Either Cost($(u, k)'$) is $<$ ShortestPath(u, k) or Cost($(k, v)'$) is $<$ ShortestPath(k, v), a contradiction



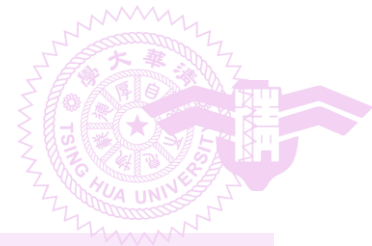


All-Pair Shortest Path Algorithm

```
void MatrixWDigraph::AllLengths(const int n)
{
    for (int i = 0; i<n; i++)
        for (int j = 0; j<n; j++)
            a[i][j]= length[i][j];

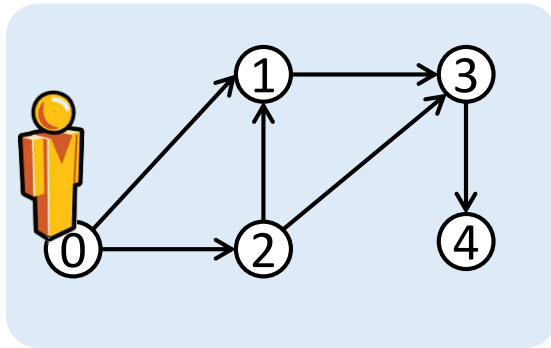
    for (int k= 0; k<n; k++)
        for (int i= 0; i<n; i++)
            for (int j= 0; j<n; j++)
                if(a[i][j] > (a[i][k] + a[k][j]))
                    a[i][j] = a[i][k] + a[k][j];
}
```





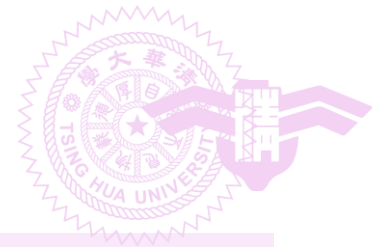
Concept of Transitive Closure

- Transition matrix, **T**
 - State change after a step


$$\mathbf{T} = \begin{matrix} & \begin{matrix} 0 & 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 1 & 1 & 1 & & \\ & 1 & & 1 & \\ & & 1 & 1 & \\ & & & 1 & 1 \\ & & & & 1 \end{bmatrix} \end{matrix}$$

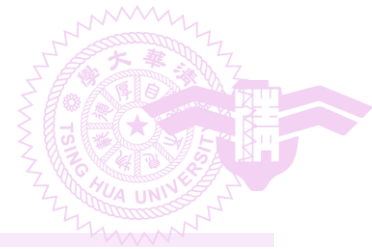
- Closure, **C**
 - The steady state after many steps
 - $(\mathbf{C} \times \mathbf{T}) = \mathbf{C}$
 - (Here we use AND for scalar multiplication and OR for scalar addition in the above example)

$$\mathbf{C} = \begin{matrix} & \begin{matrix} 0 & 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ & 1 & & 1 & 1 \\ & & 1 & 1 & 1 \\ & & & 1 & 1 \\ & & & & 1 \end{bmatrix} \end{matrix}$$



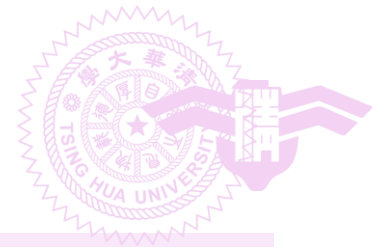
Transitive Closure of a Graph

- Given a graph with **unweighted** edges
 - **Transitive closure** matrix, A^+
 - $A^+[i][j] = 1$ if there is a path of **positive length** from i to j
 - $A^+[i][j] = 0$ otherwise
 - **Reflexive transitive closure** matrix, A^*
 - $A^*[i][j] = 1$ if there is a path of **non-negative length** from i to j
 - $A^*[i][j] = 0$ otherwise
 - The only difference between two (given unweighted edges)
 - $A^+[i][i] = 0$
 - $A^*[i][i] = 1$, as the name "**reflexive**" suggests
- Meanings of '+' and '*'
 - '+' means "**one or more**" in regular expression
 - '*' means "**zero or more**" in regular expression



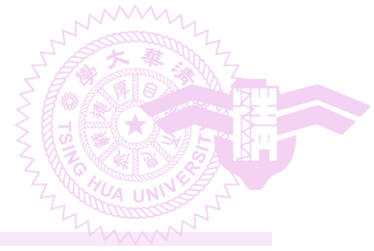
Transitive Closure Algorithm

- For a **directed** graph (with unweighted edges)
 - Perform all-pair shortest path algorithms
 - $O(n^3)$ time complexity
 - Perform n independent Dijkstra algorithms
 - $O(n^2e)$ time complexity
- For an **undirected** graph (with unweighted edges)
 - Perform **connect component algorithm** using searches (e.g., DFS or BFS)
 - $O(n^2)$ time complexity



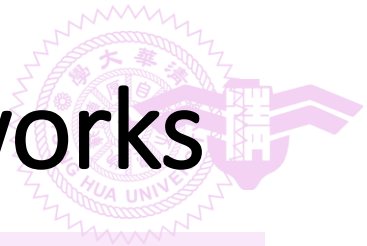
Outline

- 6.1 The graph abstract data type
- 6.2 Elementary graph operations
- 6.3 Minimum-cost spanning trees
- 6.4 Shortest paths and transitive closure
- **6.5 Activity networks**



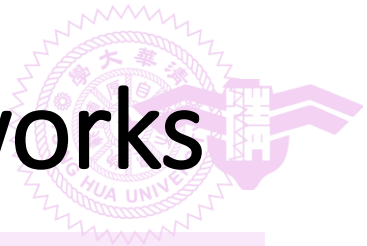
Outline

- 6.1 The graph abstract data type
- 6.2 Elementary graph operations
- 6.3 Minimum-cost spanning trees
- 6.4 Shortest paths and transitive closure
- **6.5 Activity networks**

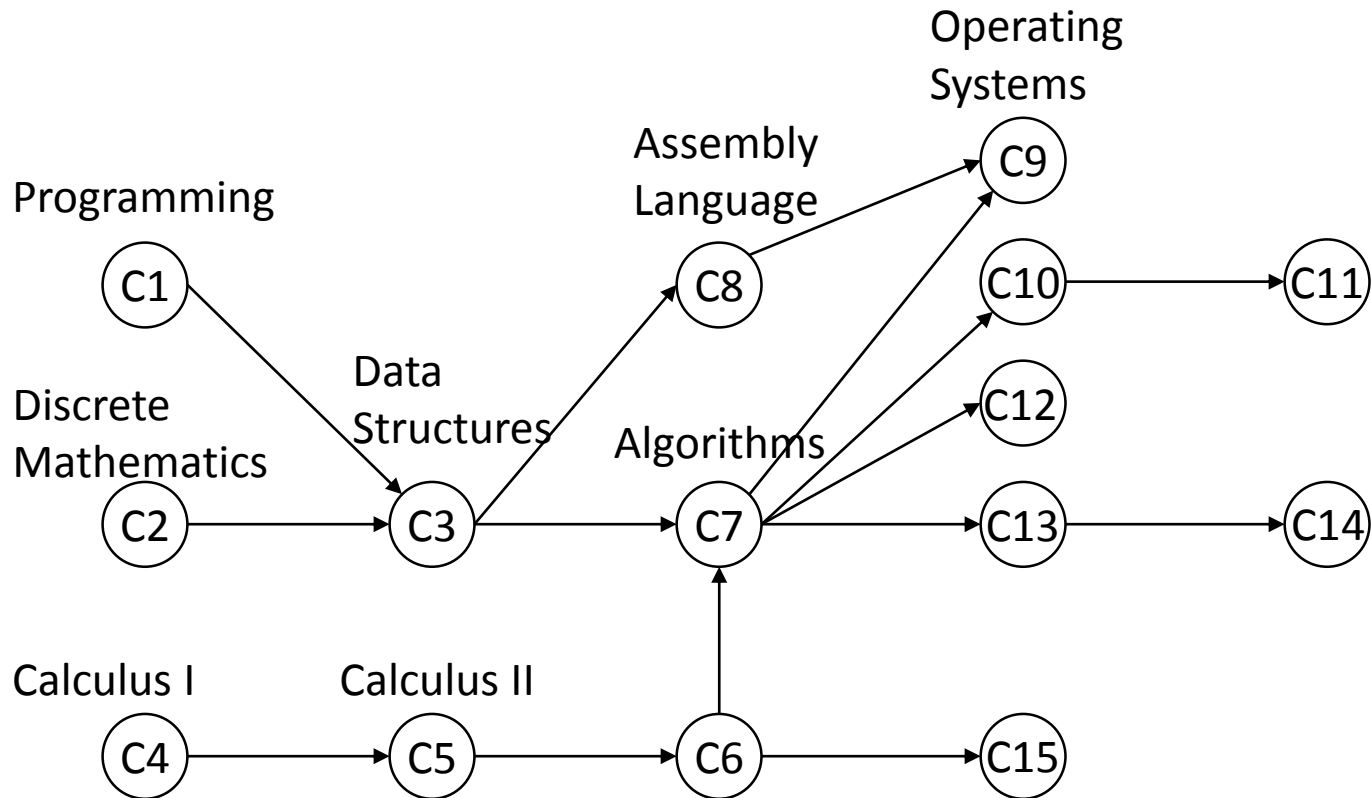


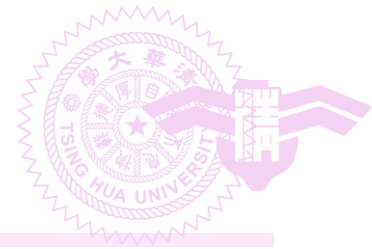
Activity-on-Vertex (AoV) Networks

- Directed graphs
 - Vertices represent tasks (i.e., activities)
 - Edges represent precedence relations
 - Vertex i is a predecessor (successor) of vertex j *iff* there's a path from i to j (j to i)
 - Vertex i is an immediate predecessor (successor) of vertex j *iff* there's an edge from i to j (j to i)



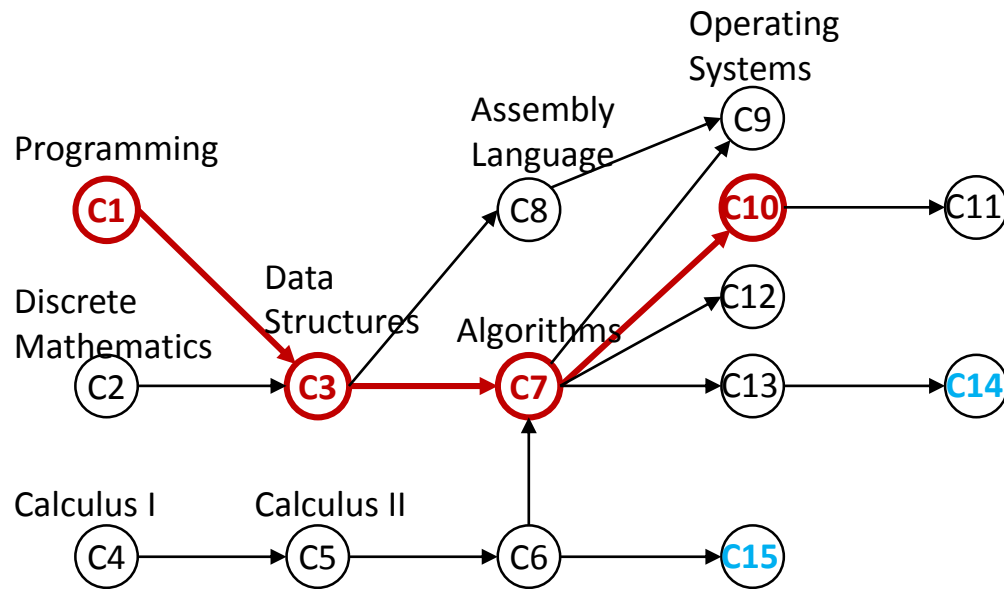
Activity-on-Vortex (AoV) Networks





Topological Order

- A linear order of the vertices of a graph such that
 - for any two vertices i and j , if i is a predecessor of j in the graph, then i precedes j in the linear ordering

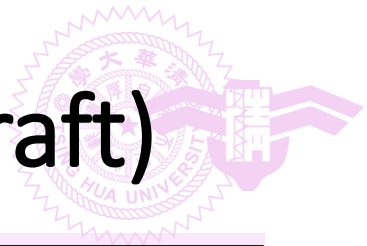


Note:

- Transitivity among >2 vertices
- Topology order between two vertices does not always imply their precedence in the graph

- Two valid topological orderings (there are many of them)

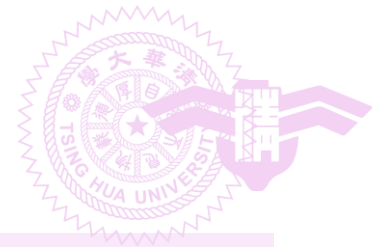
- **C1**, C2, C4, C5, **C3**, C6, C8, **C7**, **C10**, C13, C12, **C14**, **C15**, C11, C9
- C4, C5, C2, **C1**, C6, **C3**, C8, **C15**, **C7**, C9, **C10**, C11, C12, C13, **C14**



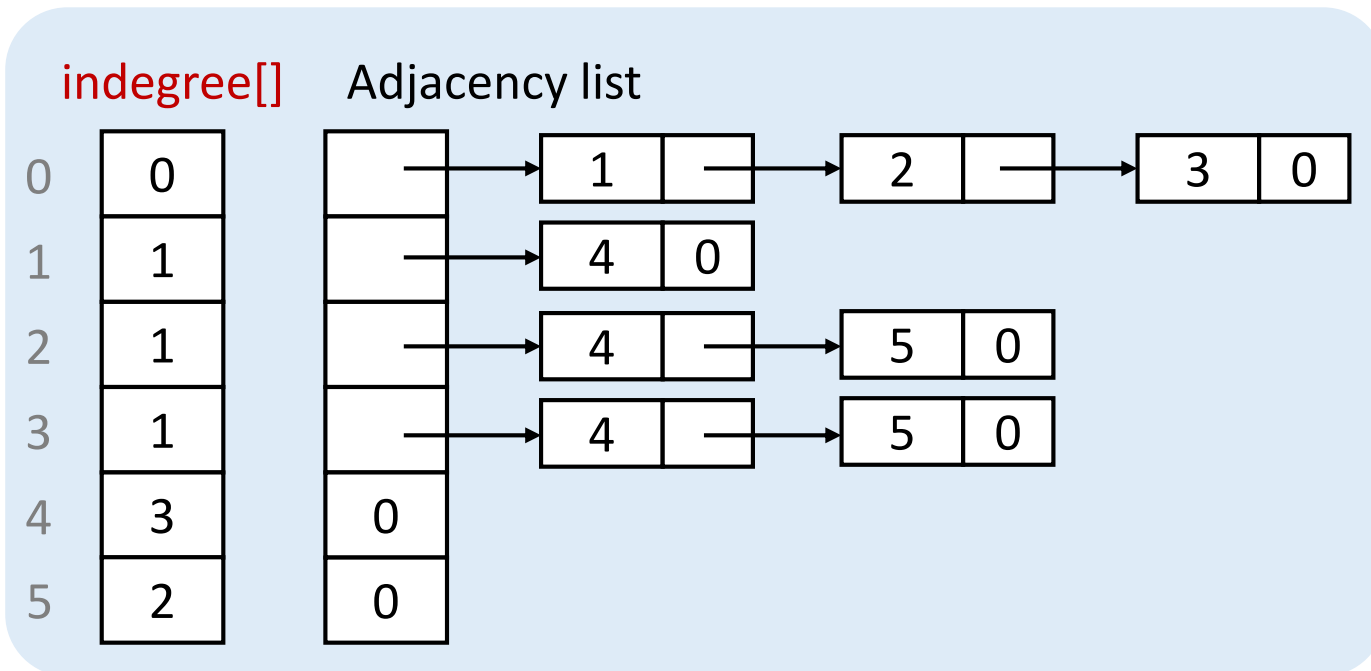
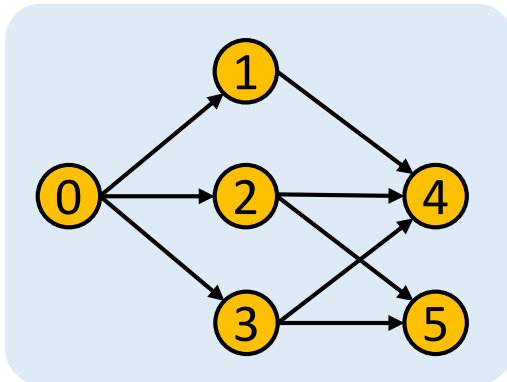
Topological Sorting Algorithm (Draft)

```
for (int i = 0; i<n; i++) {
    if (every vertex has a predecessor){
        // network has a cycle and thus is infeasible
        return;
    }
    if (vertex v has no predecessors) {
        cout << v;
        remove v and all edges leading out of v;
    }
}
```

- Graph representation considerations for the above algorithm
 - How can we remove all edges leading out of a vertex?
 - How can we determine whether a vertex has a predecessor?



Graph Representation Choice



Topological Sorting Algorithm

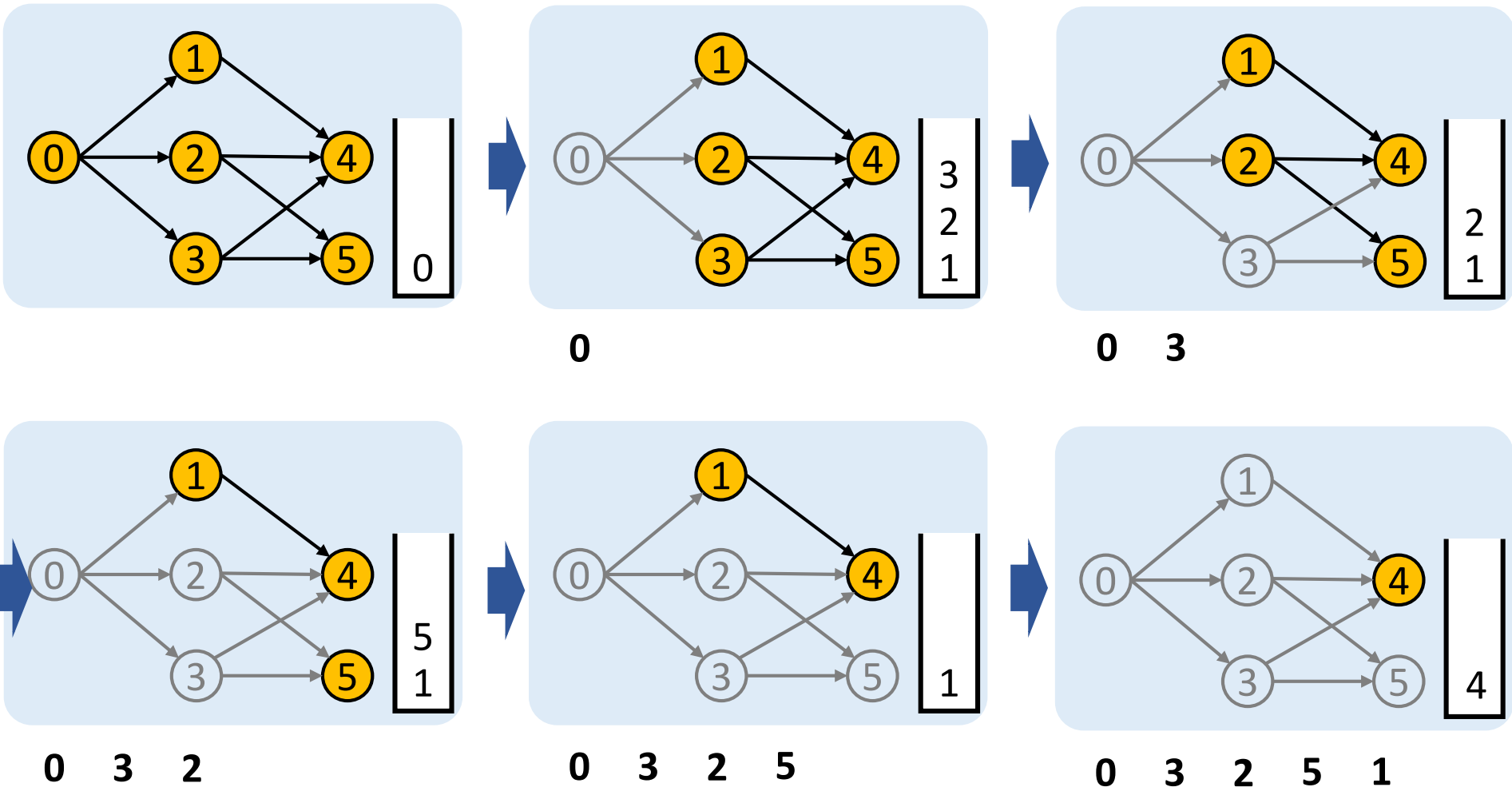


```
void LinkDigraph::TopologicalOrder()
{
    Stack s; // A stack holds 0-indegree vertices
             // Any container is good for this algorithm
    for (int i = 0; i<n; i++)
        if (indegree[i] == 0) s.push(i);

    for (i = 0; i< n; i++) {
        if (s.isEmpty() ) throw "Network has a cycle.";
        int j = s.top(); s.pop();
        cout << j <<endl;
        Chain<int>::ChainIterator ji = adjLists[j].begin();
        while (ji) {
            indegree[*ji]--;
            if (indegree[*ji] == 0) s.push(*ji);
            ji++;
        }
    }
}
```



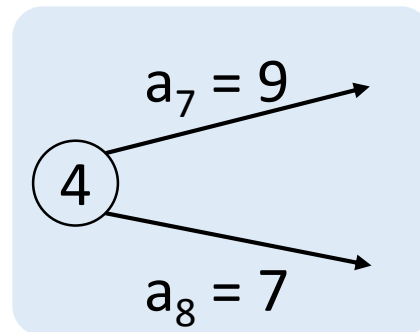
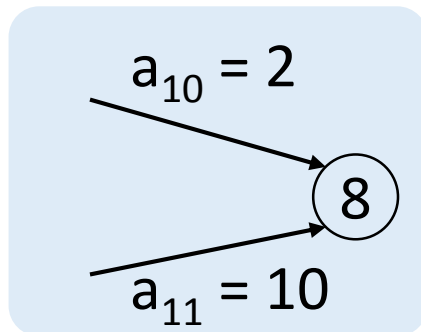
Topological Sorting Example





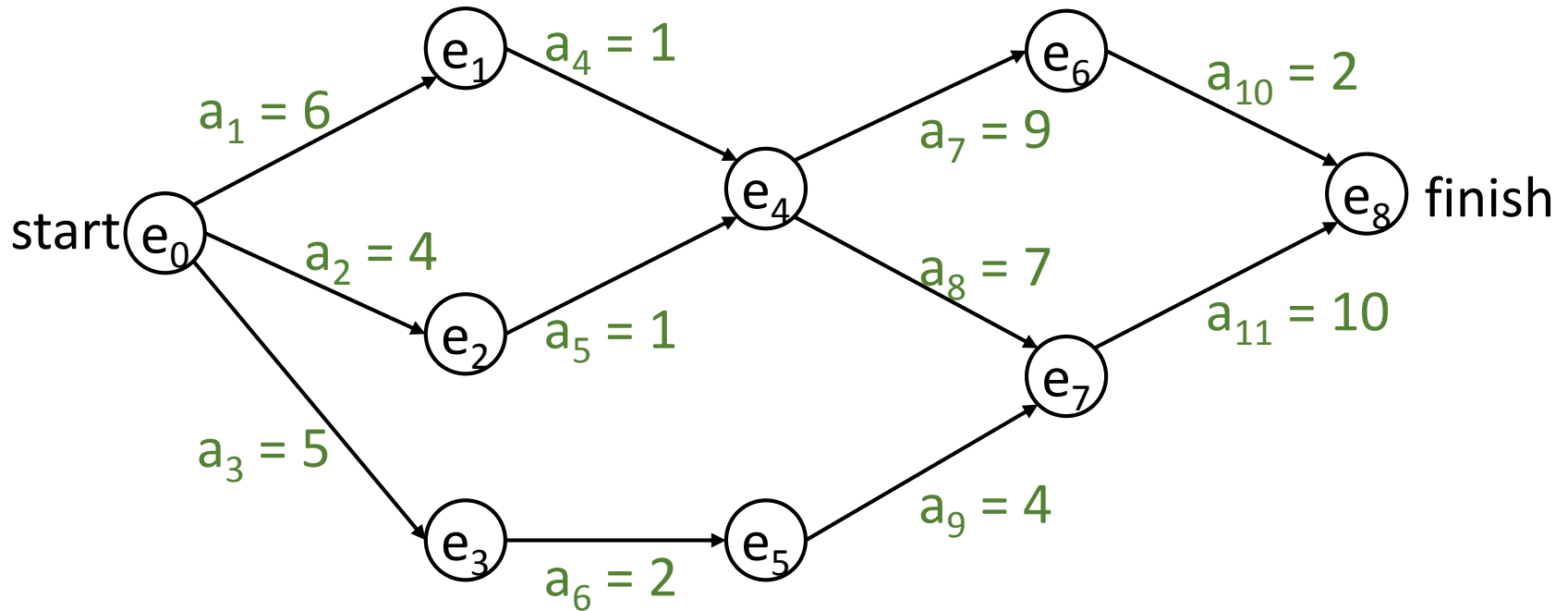
Activity-on-Edge (AoE) Networks

- Directed graph
 - Edges represent tasks (activities) to be performed
 - Vertices represent events
 - Edge cost of each activity is the time needed to perform the activity
 - Event vertex signals the completion of all activities edges entering the vertex
 - Edges leaving a vertex cannot be started until the event at the vertex has





Activity-on-Edge Network



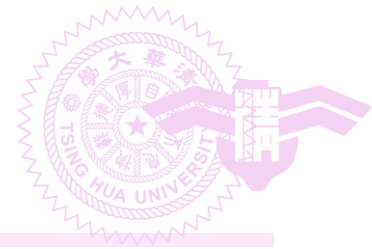
- Events

- e_0 : start of the project
- e_1 : completion of activity a_1
- e_4 : completion of activities a_4 and a_5
- ...
- e_8 : finish of the project



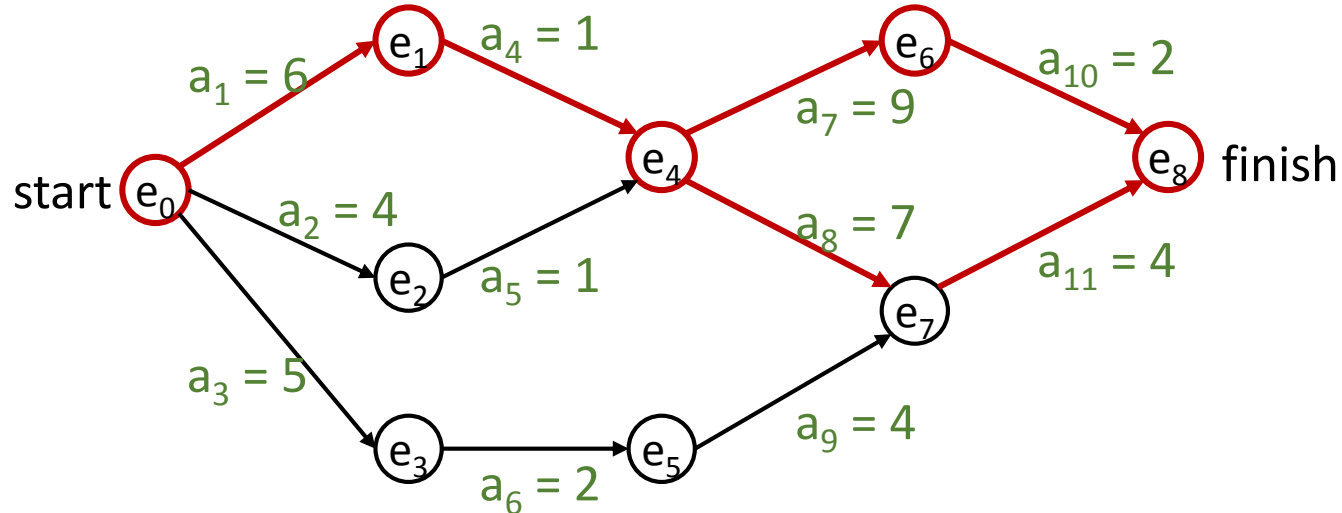
Some Important Concepts

- **Critical path**
 - The longest path from the **start** vertex to the **finish** vertex
- **Earliest time**
 - The earliest time an activity (event) can start (occur)
- **Latest time**
 - The latest time an activity (event) must start (occur) so as not to delay the project
- **Critical activities**
 - All activities for which the earliest time equals the latest time

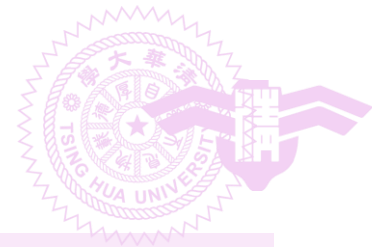


Critical Path

- The longest path from the start vertex to the finish vertex

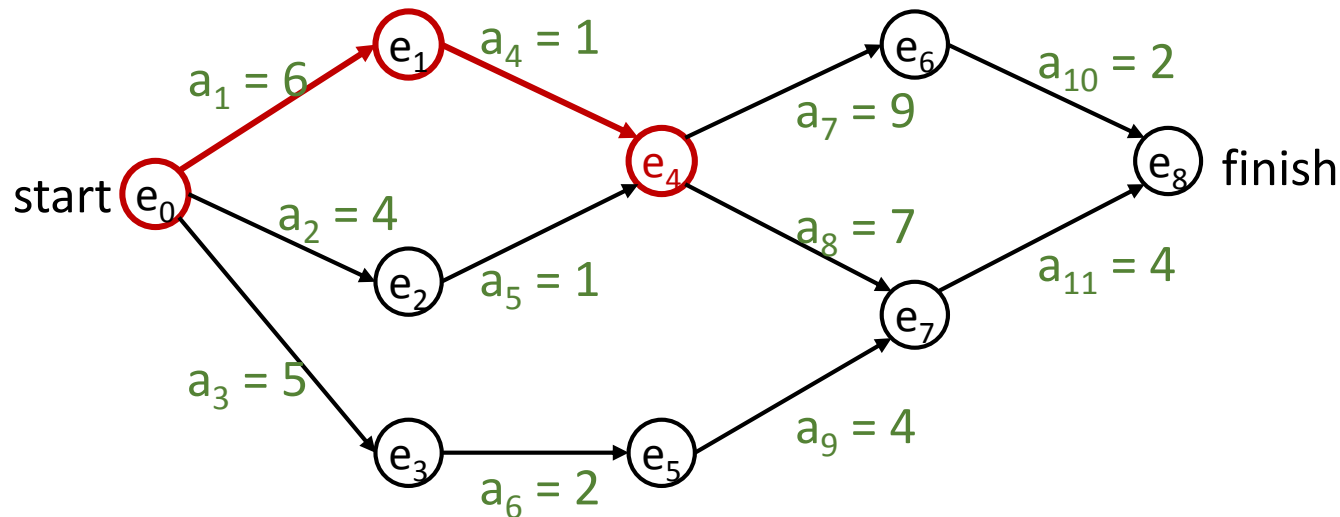


- The above network has two critical paths
- $\text{Length}(0, 1, 4, 6, 8) = 18$
- $\text{Length}(0, 1, 4, 7, 8) = 18$

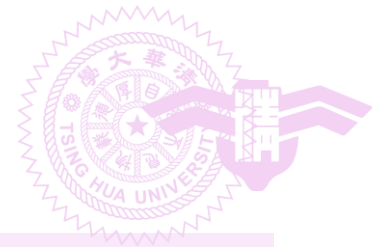


Earliest Event/Activity Time

- The length of the **longest path** from the start vertex to a vertex

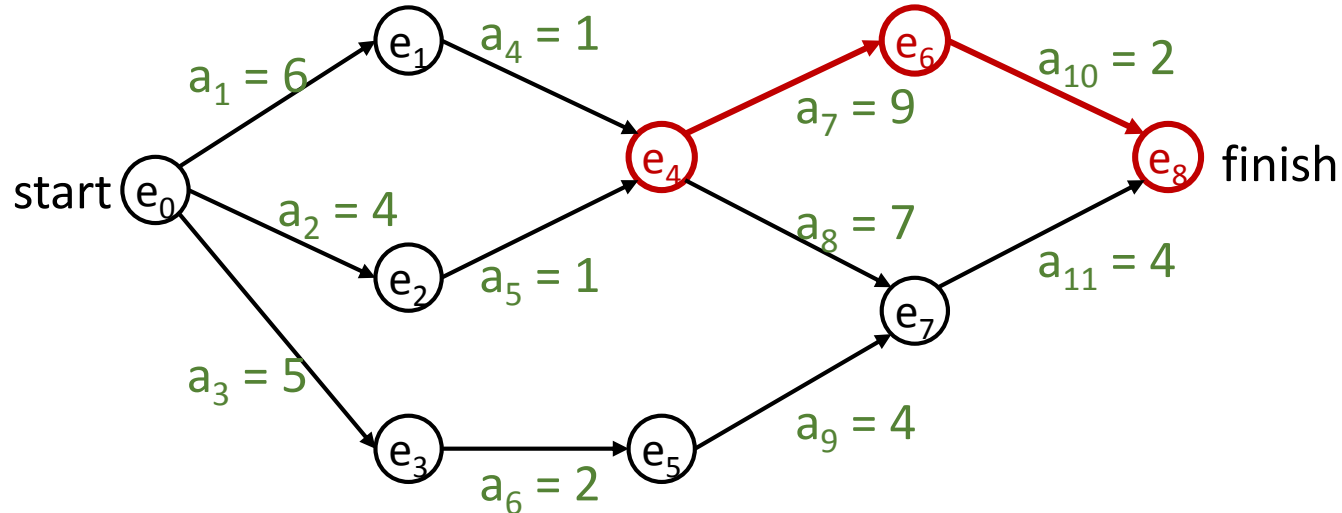


- Earliest event time(e_4) = 7
- Earliest activity time(a_7) = Earliest activity time(a_8) = 7
- Earliest event time(finish) = 18

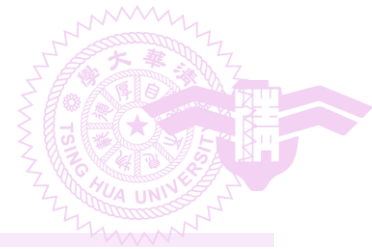


Latest Event/Activity Time

- Earliest time of the finish vertex - the length of the longest path a vertex to the finish



- Earliest event time(finish) = 18
- longest path length(e₄, finish) = 11
- latest event time(e₄) = 7



Critical Activities

- The difference between **the earliest time and the latest time**, i.e., the slack (寬裕), is a measure of the **criticality** of an activity
 - The time by which an activity may be delayed or slowed without delaying the finish of the project
- Activities having no slack are called **critical activities**



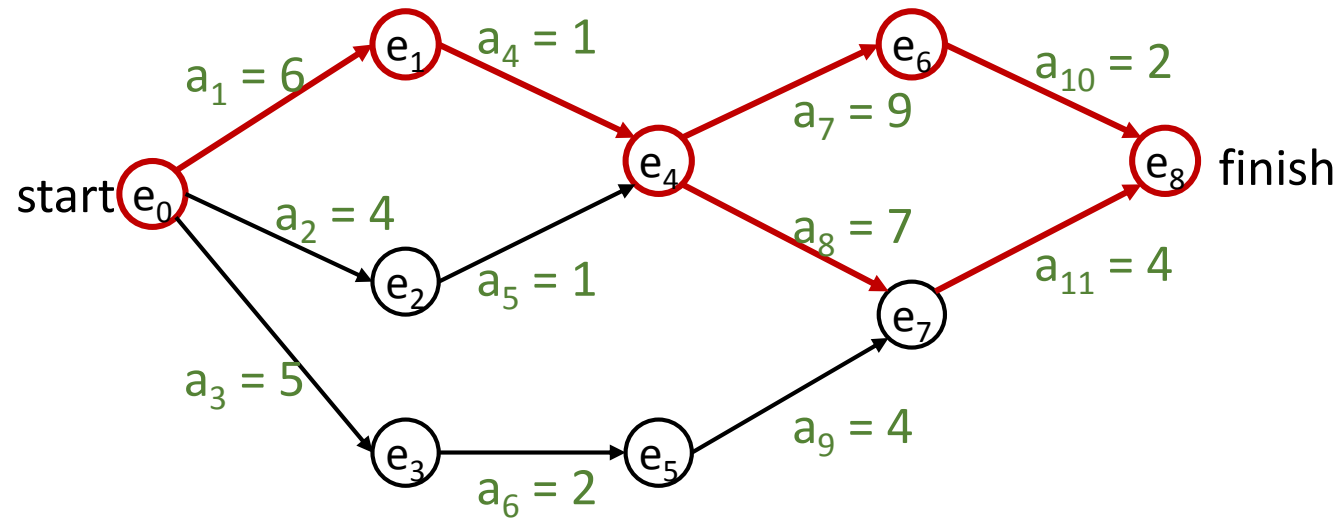
Critical Path Analysis

- Purpose
 - Speed up things, e.g., a project or a circuit
- Steps
 - Compute **earliest time** and **latest time**
 - Identify **critical activities**
 - Find **paths** in the graph with **noncritical activities removed**
- Notes
 - Speeding up **noncritical activities** or **single critical activity not on all critical paths** will not reduce the overall duration
 - **Critical paths can change** after speeding up an activity

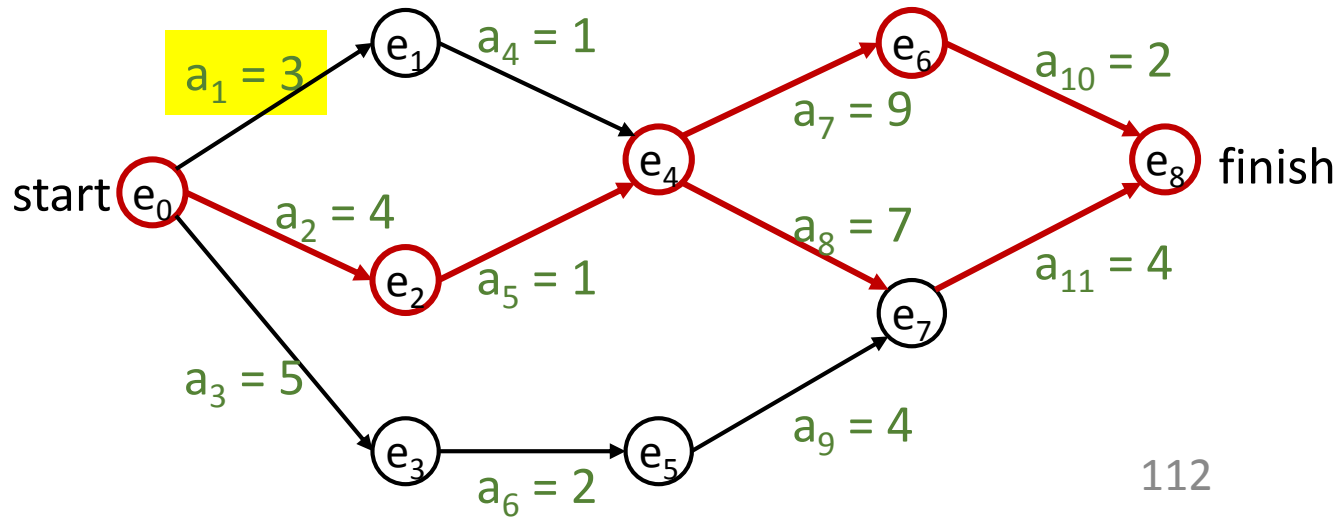


Critical Path Analysis

- a_7, a_8, a_9, a_{10} are critical. However, speeding up one of them cannot reduce overall duration

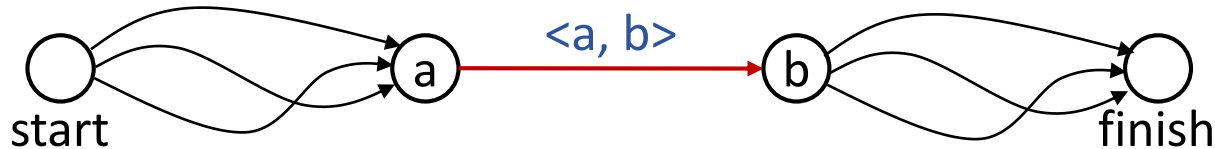


- After a_1 is speeded up from 6 to 3 units, critical paths change

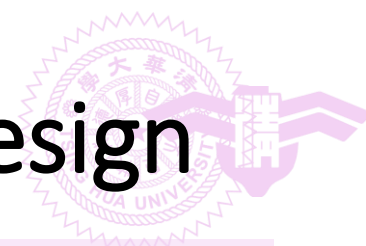




Calculating Earliest and Latest Times



- Earliest activity time($\langle a, b \rangle$)
 - = Earliest event time(a)
 - = Longest path(start, a)
- Latest activity time($\langle a, b \rangle$)
 - = Latest event time(b) - Edge cost($\langle a, b \rangle$)
- Latest event time(b)
 - = Earliest event time(finish) - Longest path(b, finish)
- The above calculation can be performed in two passes based on **topological sorting**
 - Detailed in the textbook



Critical Path Analysis in Circuit Design

- (Supplement materials)
- CAD (computer-aided design) algorithms
 - Identify critical paths in circuits
 - Push the limits for the paths to meet timing constraints
- The following circuit example is an adder
 - Add three bits and produce two resulting bits
 - Red parts are typical critical paths

