國立清華大學 電機工程學系
EE2410 Data Structure

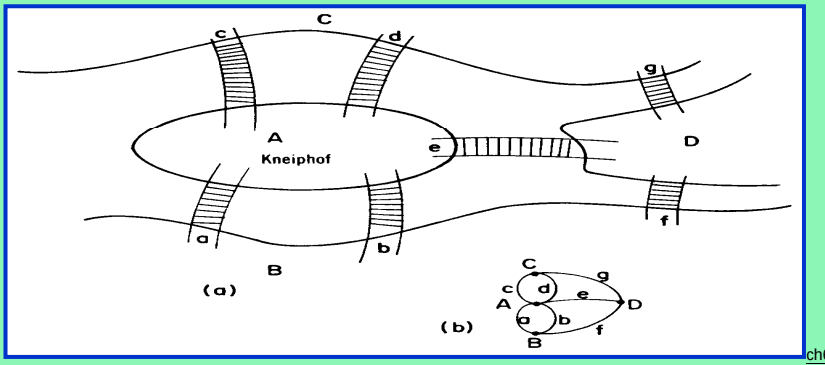
Chapter 6
Graph (Part I)

Outline

- **The Graph Abstract Data Type**
  - **Introduction**
  - **Definitions**
  - **Graph Representations**
- **Elementary Graph Operations**
- **Minimum Cost Spanning Trees**

River Pregel in Konigsberg

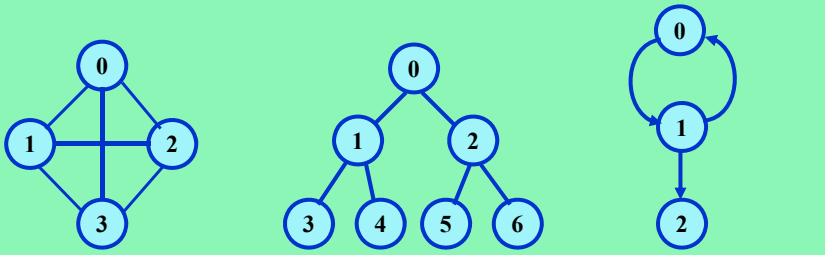
- **Problem**
  - Is there a cyclic walk that traverses every bridge only once (1736)
- **For an Euler's path to exist**
  - The degree of each vertex is even
  - The degree is the number of edges incident to a vertex



Definition and Notations of Graph

- **Definition**
  - A graph, G, consists of two sets, V and E
  - V is a finite nonempty set of vertices → V(G)
  - E is a set of pairs of vertices, called edges → E(G)
- **Terminology**
  - Undirected graph: edges are not directed
  - Directed graph (Digraph): edges are directed
    - directed pair <u, v>, u is the tail and v is the head

Sample Graphs



G1 G2 G3

V(G1) = { 0, 1, 2, 3 }; E(G1) = { (0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3) }

V(G2) = { 0, 1, 2, 3, 4, 5, 6 }; E(G2) = { (0, 1), (0, 2), (1, 3), (1, 4), (2, 5), (2, 6) }

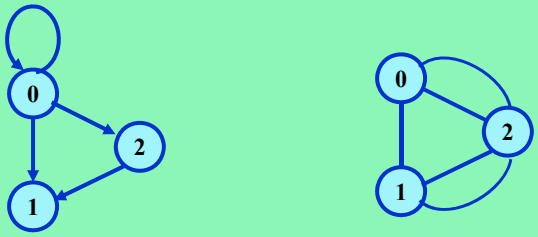
V(G3) = { 0, 1, 2 }; E(G3) = { <0, 1>, <1, 0>, <1, 2> }

Question: What are the maximum number of edges in a graph with n nodes?

→ $n \cdot (n-1)/2$ for undirected graph and $n \cdot (n-1)$ for digraph

Restrictions on Graph

- **No self loops**
  - A self loop (or self edge) is an edge from a vertex v back to itself
  - That is, (v, v) and <v, v> are not legal
- **No multiple occurrences of the same edge**



Graph with self edge Multigraph

Terminology

- **Complete graph**
  - An n-vertex, undirected graph with exactly n(n-1)/2 edges is said to be complete
  - An n-vertex, directed graph with exactly n(n-1) edges is said to be complete
- **Adjacent nodes**
  - If (u, v) is an edge in E(G), then u and v are adjacent, and edge (u, v) is incident on vertices u and v
  - If <u, v> is a directed edge, then u is adjacent to v, and v is adjacent from u
- **A subgraph of G**
  - is a graph G' such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$

Path in A Graph

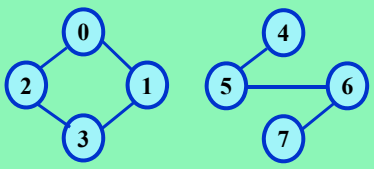
- **A path from vertex u to vertex v**
  - is a sequence of vertices u, i_1, i_2, ..., i_k, v such that (u, i_1), (i_1, i_2), ..., (i_k, v) are all edges in E(G)
  - A path (0, 1), (1, 3), (3, 2) is also written as 0, 1, 3, 2
- **A simple path**
  - is a path in which all vertices except possibly the first and last are distinct
- **The length of a path**
  - is the number of edges on a path
- **A cycle**
  - is a simple path in which the first and last vertices are the same

Connected Component

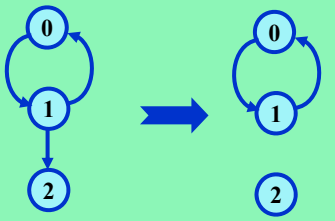
- **Connected vertices**
  - In a undirected graph, two vertices u and v are said to be connected iff there is a path in G from u to v
- **Connected graph**
  - An undirected graph is said to be connected iff for every pair of distinct vertices u and v in V(G) there is a path from u to v in G
- **A connected component**
  - is a maximal connected subgraph
- **A tree is a connected acyclic graph**

Strongly Connected Component

- **Strongly connected graph**
  - 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
- **Strongly connected component (SCC)**
  - A SCC is a maximal subgraph that is strongly connected



A graph with two connected components



G3 Two SCC's of G3

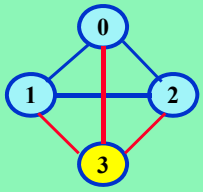
Abstract Data Type Graph

```
class  Graph
{
//  objects: A nonempty set of vertices and a set of undirected edges
//  where each edge is a pair of vertices
public:
    Graph ();  // Create an empty graph

    void   InsertVertex(Vertex  v);  //  Insert v into graph; v has no incident edges

    void   InsertEdge(Vertex u, Vertex v);  // Insert edge (u, v) into graph

    void   DeleteVertex(Vertex v);   //  Delete v and all edges incident to it

    void   DeleteEdge(Vertex u, Vertex v);  //  Delete edge (u, v) from the graph

    Boolean  IsEmpty ();
          //  if graph has no vertices return TRUE(1); else return FALSE(0);

    List<Vertex>  Adjacent( Vertex v);
          //  return a list of all vertices that are adjacent to v
}
```


Graph Representations

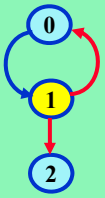
- **Adjacency matrices**
- **Adjacency lists**
- **Adjacency multi-lists**

Adjacency Matrices



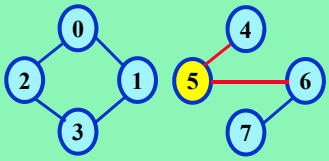
G1

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



G3

	0	1	2
0	0	1	0
1	1	0	1
2	0	0	0



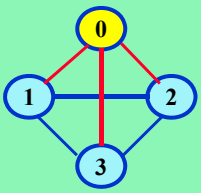
	0	1	2	3	4	5	6	7
0	0	1	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0
2	0	0	0	1	0	0	0	0
3	0	1	1	0	0	0	0	0
4	0	0	0	0	0	1	0	0
5	0	0	0	0	1	0	1	0
6	0	0	0	0	0	1	0	1
7	0	0	0	0	0	0	1	0

Questions: How many edges? Is G connected ?

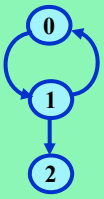
→ requires $O(n^2)$

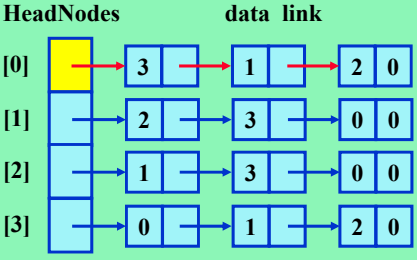
Adjacency Lists

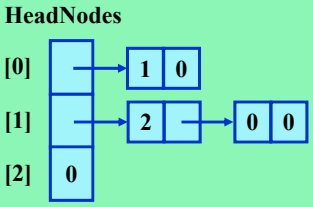
G1



G3







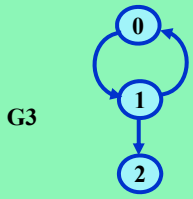
Graph Using Adjacency Lists

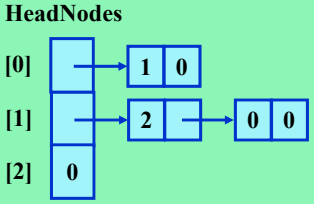
```
class  Graph
{
private:
      List<int>  *HeadNodes;
      int   n;
public:
      Graph( const  int  vertices = 0) : n ( vertices )
      {  HeadNodes = new List<int>[n]; } ;
};
```

Complexities of simple operations:

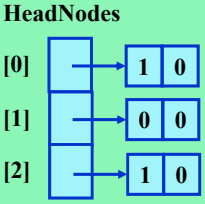
1. Determine the total number of edges of a graph: O(n+e)
2. Determine the out-degree of a node: O(out-degree of the node)
3. Determine the in-degree of a node: needs inverse adjacency lists

Inverse Adjacency Lists



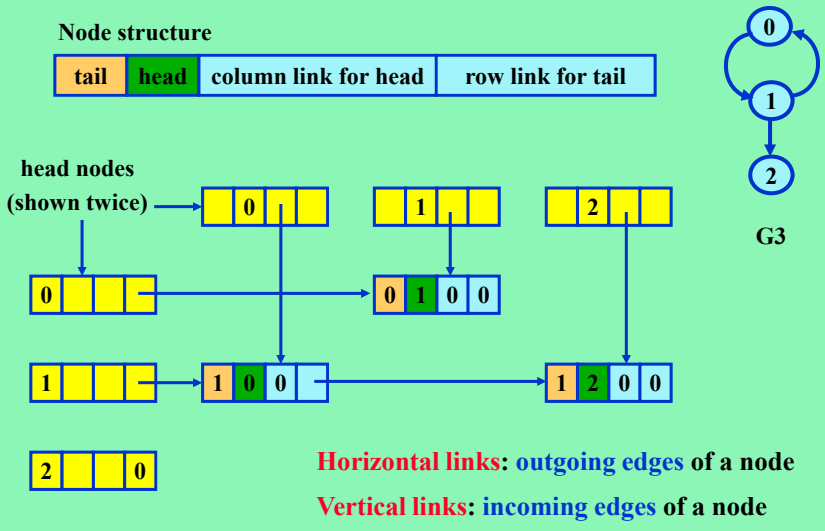


adjacency lists



inverse adjacency lists

Orthogonal List Representation

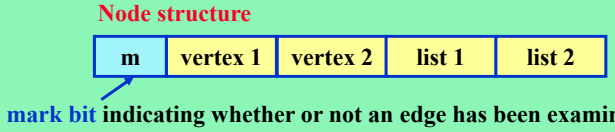


Horizontal links: outgoing edges of a node

Vertical links: incoming edges of a node

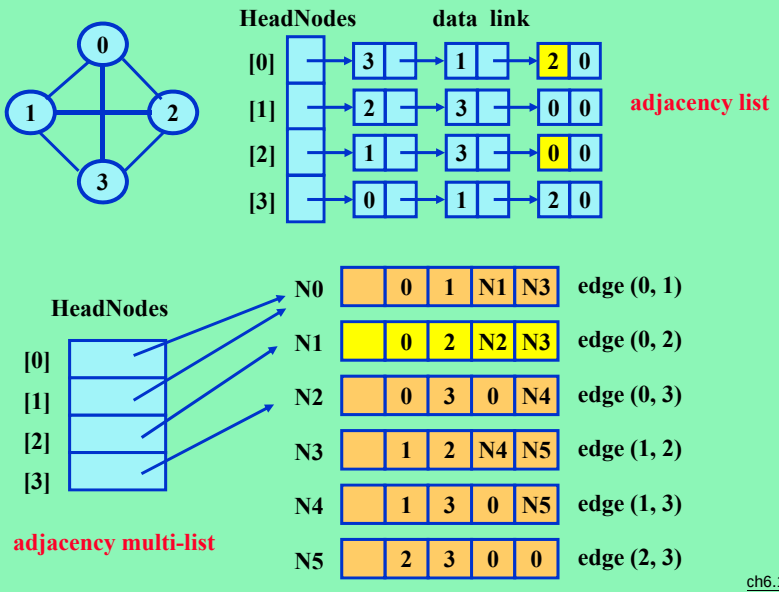
Adjacency Multi-Lists

- **Motivations**
  - An edge (u,v) in adjacency lists is represented by two entries, one in list for u, the other in list for v
  - During graph traversal, we need to mark an edge as visited → need a better representation
- **Adjacency Multi-Lists**
  - There is one node for each edge
  - A node may be shared among several lists



mark bit indicating whether or not an edge has been examined

Example: Adjacency Multi-Lists



ADT of Adjacency Multi-Lists

```
enum  Boolean  { FALSE, TRUE }
class   Graph;
class  GraphEdge {
friend  Graph;
private:
   Boolean  m;
   int   vertex1, vertex2;
   GraphEdge  *path1, *path2;
};
```

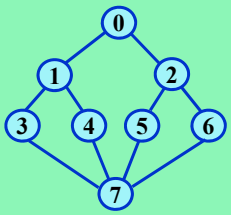
```
typedef  GraphEdge  *EdgePtr;
class Graph {
private:
   EdgePtr *HeadNodes;
   int  n;
public:
   Graph(const int);
};
```

```
Graph::Graph(int vertices=0) : n (vertices)
{
//  Set up the array of head nodes
   HeadNodes = new EdgePtr[n];
   for(i=0; i<n; i++)  HeadNodes[i] = 0;
}
```

Outline

- The Graph Abstract Data Type
- Elementary Graph Operations
  - Depth First Search
  - Breadth First Search
  - Connected Components
  - Spanning Trees
  - Bi-connected Components
- Minimum Cost Spanning Trees

Depth First Search

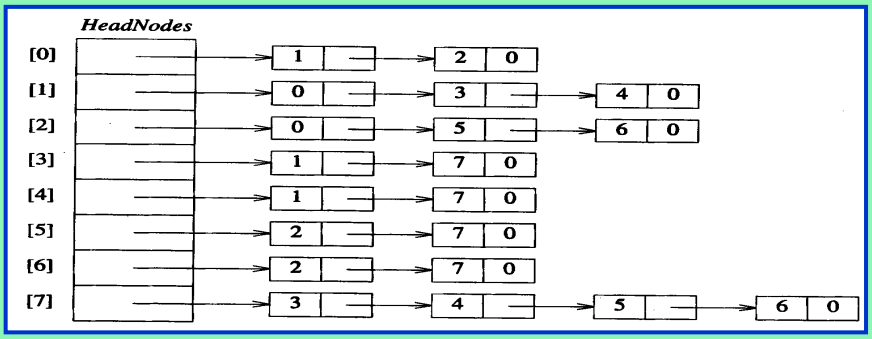


Depth First Search (DFS) orders: (for example)

0, 1, 3, 7, 4, 5, 2, 6

0, 1, 4, 7, 3, 5, 2, 6

etc.

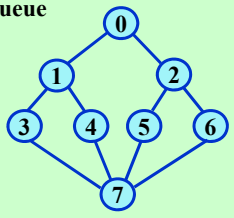


Depth First Search Algorithm

```
void  Graph::DFS()   //  Driver
{
   visited = new  Boolean[n];
   for ( int  i=0; i<n; i++ )  visited[i] = FALSE;

   DFS(0);  // start search at vertex 0

   delete[ ]  visited;
}

void  Graph::DFS(const int v) // Workhorse
// visit all previously unvisited vertices that are reachable from vertex v
{
   visited[v] = TRUE;
   for ( each vertex w adjacent to v )
          if ( ! visited[w] )  DFS(w);
   }
}
```


Breadth First Search

```
void  Graph::BFS( int  v )
// A breadth first search of the graph is carried out beginning at vertex v
// visited[i] is set to TRUE when v is visited. The algorithm uses a queue
{
   visited = new Boolean[n];
   for ( int i=0; i<n; i++)  visited[i] = FALSE;
   visited[v] = TRUE;
   Queue<int>  q;
   q.Insert(v);  //  add vertex v to the queue
   while ( ! q.IsEmpty() )  {
          v = *q.Delete(v);  //  remove vertex v from the queue
          for ( all vertices w adjacent to v ) {
                if ( ! visited[w] ) {
                     q.Insert(w);
                     visited[w] = TRUE;
                }
          }
    } // end of while loop
    delete [] visited;
}
```



BFS order: 0,1,2,3,4,5,6,7

Connected Components

- **For an undirected graph**
  - The connected components can be computed by either DFS or BFS search
  - All nodes visited during a traversal along with their edges form a connected components

```
void  Graph::Components()
// Determine the connected components of the graph
{
   visited = new Boolean[n];
   for ( int  i=0; i<n; i++)  visited[i] = FALSE;
   for ( i=0; i<n; i++) {
         if ( ! visited[i] ) {    // pick one node that is not visited yet
              DFS(i);  // Find a component
              OutputNewComponent();
         }
    delete [] visited;
}
```

Complexity = O(n+e)
for adjacency lists

Spanning Trees

- **Definition**

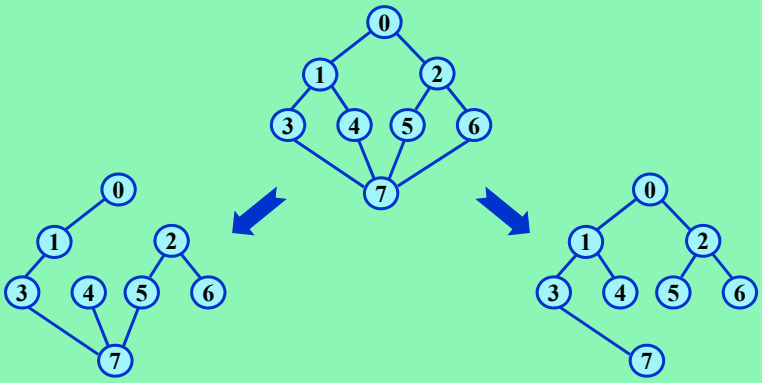
  Any tree is a spanning tree of G if

  (1) The tree consists solely of edges in G

  (2) The tree includes all vertices in G

- **For a connected graph G**
  - Depth-first or breadth-first search partitions the edges into two sets, T and N
  - T is the set of tree edges
  - N is the set of non-tree edges
- **The tree edges of a traversal**
  - and every vertex forms a spanning tree

Examples: Spanning Trees



Depth-first spanning tree

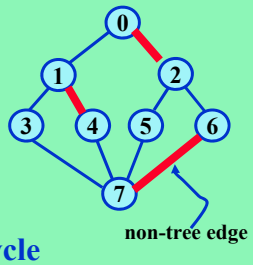
(starting from vertex 0)

Breadth-first spanning tree

(starting from vertex 0)

Creation of Circuit Equations

- **In a spanning tree of a connected graph**
  - **Each non-tree edge added to the tree forms a cycle**
  - **Each cycle is unique**
- **Application to circuit analysis**
  - **Represent a circuit as a graph**
  - **Find a spanning tree**
  - **Each non-tree edge corresponds to a cycle**
  - **Generate a current equation using Kirchhoff's law**
  - **A set of independent current equations are obtained**

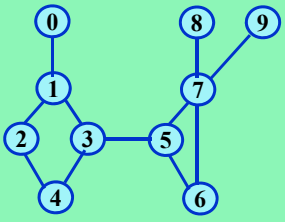


Minimal Connected Subgraph

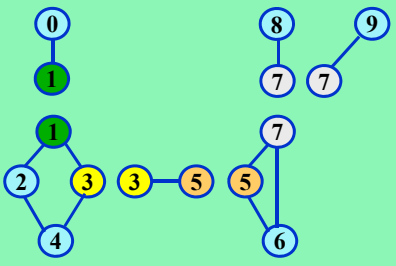
- **Property**
  - A spanning tree is a minimal sub-graph G' of G such that V(G') = V(G), and G' is connected
- **Reasons**
  - Any connected graph with n vertices must have n-1 edges
  - All connected graphs with n-1 edges are trees
  - Therefore, a spanning tree is a minimal sub-graph
- **Application to communication**
  - Vertices represent cities, while edges represents communication links
  - The minimum number of links connecting n cities is n-1
  - The cost of each link is different, represented as weight
  - Finding minimum-cost spanning tree is desired !

Articulation Point

- **Definition of Articulation Point**
  - A vertex v of G is an articulation point iff the deletion of v, together with the deletion of all edges incident to v, leaves behind a graph that has at least two connected components
- **Definition of Bi-connected Graph**
  - A bi-connected graph is a connected graph that has no articulation points



A connected graph



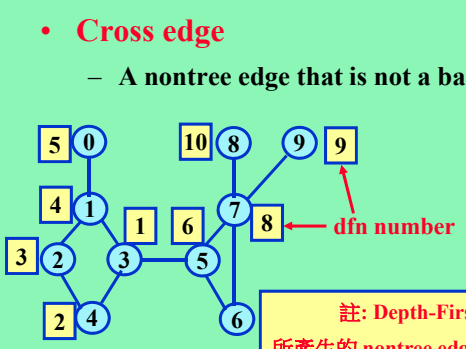
6 bi-connected components

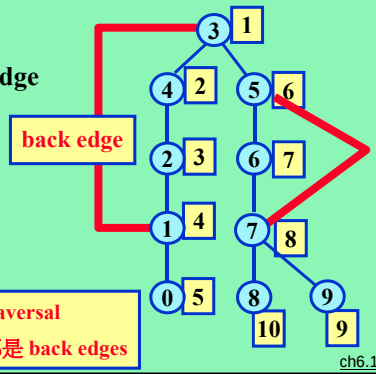
Bi-Connected Components

- **Definition**
  - A biconnected component of a connected graph G is a maximal biconnected subgraph H of G
  - By maximal, it means that G contains no other subgraph that is both biconnected and properly contains H
- **Properties**
  - A biconnected graph has just one biconnected component – the whole graph
  - Two biconnected components can have at most one vertex in common
  - No edge can be in two biconnected components
  - Hence, biconnected components of G partition the edges of G

Back Edge and Cross Edge

- **Depth first number (dfn)**
  - The order of a node visited during a depth first search
- **Back edge**
  - A nontree edge (u, v) is a back edge iff u is an ancestor of v or v is an ancestor of u
- **Cross edge**
  - A nontree edge that is not a back edge

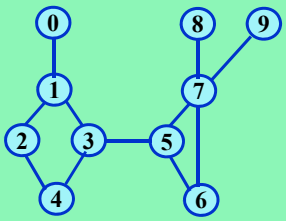


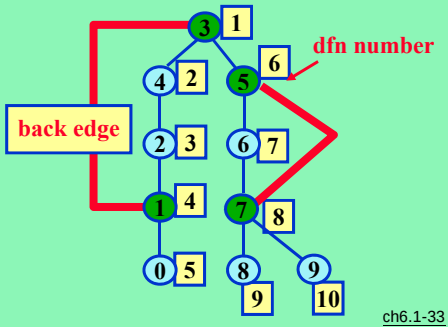


註: Depth-First Traversal
所產生的 nontree edges 都是 back edges

Where Are The Articulation Points ?

- **Root is an articulation point**
  - iff it has at least two children
- **Back path is a path starting from a vertex *u***
  - reaches an ancestor of u through u's descendants and single back edge
- **A Non-root vertex u is an articulation point iff**
  - (1) u has at least one child
  - (2) u has NO such child w that there exist a back path starting from w

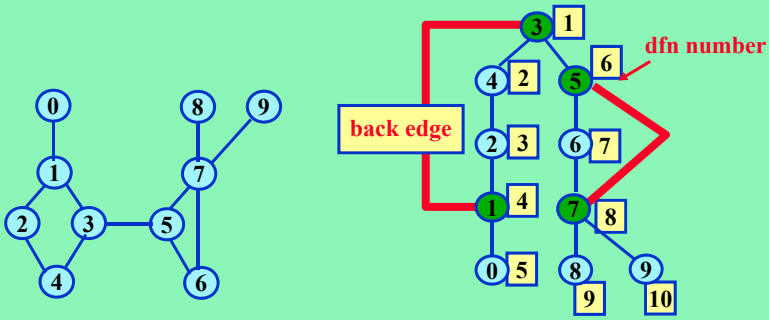





```
void  Graph::DfnLow(const int x)  //  begin DFS at vertex x
{
  num = 1;
  dfn = new int[n];
  low = new int[n];
  for ( int i=0; i<n; i++) {  dfn[i] = low[i] = 0; }
  DfnLow(x, -1);  // start at vertex x
  delete [ ]  dfn;
  delete [ ]  low;
}
void  Graph::DfnLow ( const int u, const int v)
//  Compute dfn and low while performing a depth first search beginning
//  at vertex u. vertex v is the parent (if any) of u in the resulting spanning tree
{
   dfn[u] = low[u] = num++;
   for ( each vertex w adjacent from u )
        if ( dfn[w]==0) {  w is an unvisited vertex
             DfnLow(w, u);
             low[u] = min2( low[u], low[w]);
        }
        else if ( w != v)  low[u] = min2( low[u], dfn[w] );  // back edge
    }
}
```

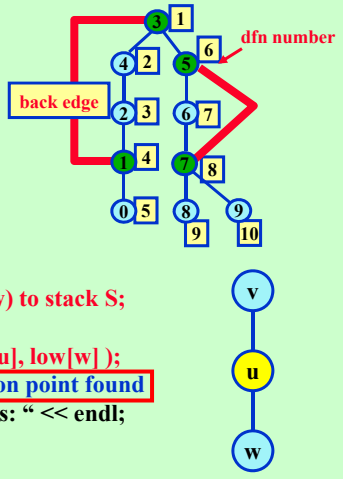
low(u) is the lowest depth first number reachable by back path starting from u

Example: Values of *dfn* and *low*



Vertex ID	0	1	2	3	4	5	6	7	8	9
dfn	5	4	3	1	2	6	7	8	9	10
low	5	1	1	1	1	6	6	6	9	10


```
void  Graph::Biconnected()
{
   num = 1;  dfn = new int[n];  low = new int[n];
   for ( int  i=0; i<n; i++) {  dfn[i] = low[i] = 0; }
   Biconnected(0, -1);  // start at vertex 0
   delete [] dfn;  delete [] low;
}
void  Graph::Biconnected ( const int u, const int v)
{
   dfn[u] = low[u] = num++;
   for ( each vertex w adjacent from u )
        if ( (w !=v) && (dfn[w] < dfn[u]) )  add (u, w) to stack S;
        if ( dfn[w] == 0) {  // w is an unvisited vertex
            Biconnected(w, u);    low[u] = min2( low[u], low[w] );
            if ( low[w] >= dfn[u])  {  // u an articulation point found
                cout << "New biconnected components: " << 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] = min2( low[u], dfn[w] ) ;  // back edge
}
```



Outline

- The Graph Abstract Data Type
- Elementary Graph Operations
- Minimum Cost Spanning Trees
  - Kruskal's Algorithm
  - Prim's Algorithm
  - Sollin's Algorithm

Minimum-Cost Spanning Tree

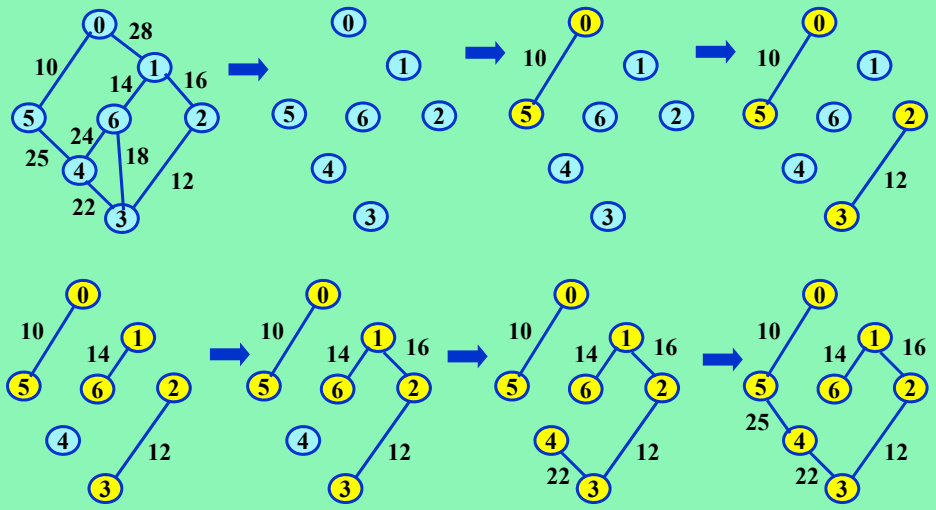
- Cost of a spanning tree
  - is the sum of the costs (weights) of the edges in the spanning tree
- A minimum-cost spanning tree
  - is a spanning tree of least cost
- Greedy method
  - The solution is constructed in stages
  - At each stage, the best decision (using some criterion) is picked
  - No decision, once made, can be reversed
- Selection criterion in forming a min-cost spanning tree

(1) Use only edges within the graph

(2) Use exactly n-1 edges

(3) Should not use edges that produce a cycle

Example of Forming A Min-Cost Spanning Tree – Kruskal's Algorithm



Kruskal's Algorithm

```
T = ∅;
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, w) does not create a cycle in T )  add (v, w) to T;
        else  discard (v, w);
}
if ( T contains fewer than n-1 edges )  {
     cout << "No spanning tree" << endl;
}
```

T is the set of collected edges in the spanning tree

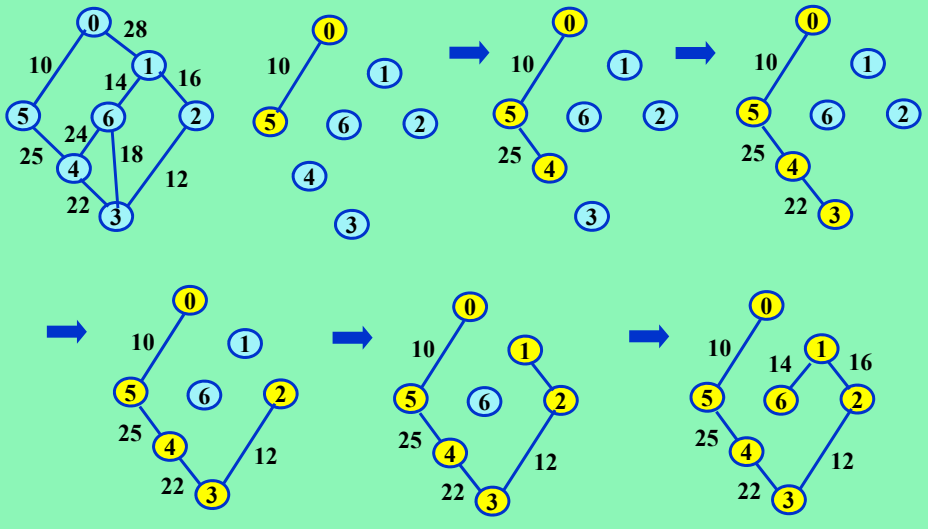
O(e·log e) if min heap is used, and set is used for cycle checking

It can be proved that: Kruskal's algorithm is optimal

(1) If there is a spanning tree, → Kruskal will find it

(2) If there is a min-cost spanning tree U, then there exists a cost-preserving transformation that maps U to the one Kruskal finds

Example of Prim's Algorithm

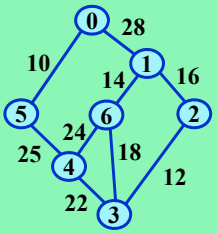


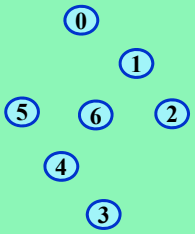
Prim's Algorithm

Notations:
(1) *TV* is the set of collected vertices in the spanning tree
(2) *T* is the set of collected edges in the spanning tree

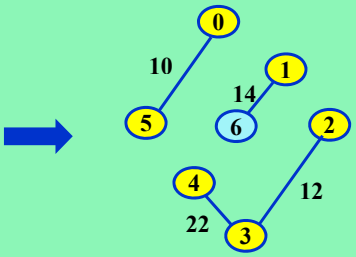
```
// Assume that G has at least one vertex
TV = {0};  // Start with vertex 0 and no edges
for ( T= ∅; T contains fewer than n-1 edges;  add (u,v) to T )
{
    Let (u,v) be a least-cost edge such that u ∈ TV and v∉TV;
    if ( there is no such edge )  break;
    add v to TV;
}
if ( T contains fewer than n-1 edges)  {
      cout << "No spanning tree" << endl;
}
```

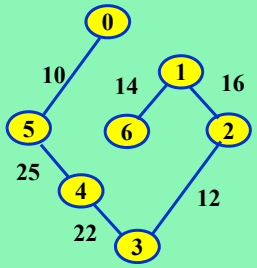
Stages in Sollin's Algorithm





initial





Total weight = 99