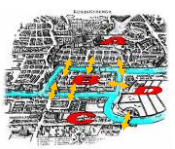
EECS 204002
Data Structures 資料結構
Prof. REN-SONG TSAY 蔡仁松 教授
NTHU

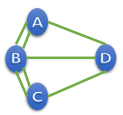
CH. 6 GRAPHS

6.1 The Graph Abstract Data Type

The Use of Graphs

- The first record (1736)
  - Konigsberg Bridge Problem: Walk across all the bridges exactly once
  - Solved by Euler
  - Formulate as a graph
- Prove: possible
  - Iff the degree of each vertex is even





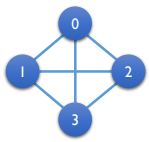
6.1.2

Graph Definition

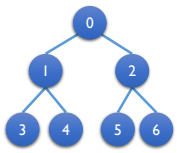
- A graph, ***G***, consists of two sets, ***V*** and ***E***.
  - ***G = (V, E)***
  - ***V***: a set of **vertices**.
  - ***E***: a set of *pairs of vertices* called ***edges***.
- **Undirected graph** (simply **graph**)
  - (u, v) and (v, u) represent the same edge.
- **Directed graph** (**digraph**)
  - $< u, v > \neq < v, u >$
  - $< u, v > \Rightarrow u$ is head and v is tail of edge.

4

Graph Examples



Undirected Graph
V(G)={0, 1, 2, 3}
E(G)={(0,1), (0,2), (0,3), (1,2), (1,3), (2,3)}



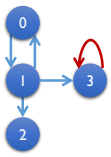
Undirected Graph
V(G)={0, 1, 2, 3, 4, 5, 6}
E(G)={(0,1), (0,2), (1,3), (1,4), (2,5), (2,6)}

Directed Graph
V(G)={0, 1, 2}
E(G)={<0,1>, <1,0>, <1,2>)}

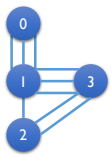
5

Restrictions

- ***Self edges*** and ***self loops*** are not permitted!
  - Edges of the form (v, v) and $<v, v>$ are not legal.
- A graph may not have multiple occurrences of the same edge (***multigraph***).



Graph with self edge



Multigraph

6

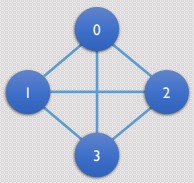
Graph, Vertex and Edge

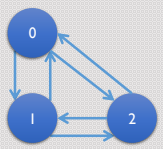
- For a graph with n vertices, the maximum # of edges is:
  - $n(n-1)/2$ for undirected graph
  - $n(n-1)$ for directed graph
- Vertices u and v are **adjacent** if $(u, v) \in E$ and edge (u, v) is **incident** on vertices u and v.
- For a directed edge $< u, v >$, we say u is **adjacent to** v and v is **adjacent from** u, and edge (u, v) is **incident** on vertices u and v.

7

Complete Graph

Complete undirected graph	Complete directed graph
• Graph with n vertices has exactly $n(n-1)/2$ edges.	• Graph with n vertices has exactly $n(n-1)$ edges.

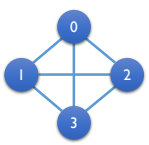




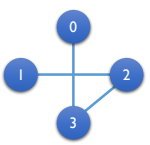
8

Subgraph

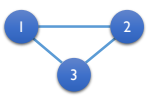
- G' is a subgraph of G such that $V(G')$? $V(G)$ and $E(G')$? $E(G)$.



Graph



Subgraph

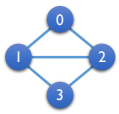


Subgraph

9

Path and Simple Path

- **Path**:
  - A path from u to v represents a sequence of vertices $u, i_1, i_2, \ldots, i_k, v$ such that $(u, i_1), (i_1, i_2), \ldots, (i_k, v)$ are edges in graph.
- **Simple path:**
  - A simple path is a path in which all vertices except possibly the first and the last are distinct.



Sequence	Path?	Simple path?
0,1,3,2	Yes	Yes
0,2,0,1	Yes	No
0,3,2,1	No	No

10

Cycle

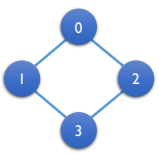
A cycle is a simple path which the first and the last vertices are the same.

- Notes: if the graph is a directed graph, we usually add the prefix "directed" to above terms:
  - Directed path
  - Directed simple path
  - Directed cycle

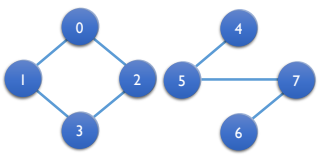
11

Connected Graph

- Undirected graph G is said to be **connected** iff for **every pair of distinct vertices u and v**, there is a **path from u to v** in G.



Connected graph

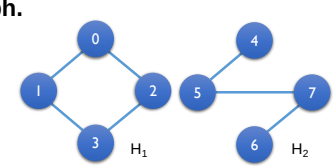


Not a connected graph

12

Connected Component & Tree

- A **connected component**, *H*, of an undirected graph is a **maximal connected subgraph.**



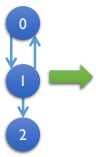
Graph with two connected components

- **Tree:**
  - A connected acyclic graph.

13

Strongly Connected

- Directed graph G is said to be **strongly connected** iff for **every pair of distinct vertices u and v**, there is a **directed path from u to v** and **also from v to u** in G.

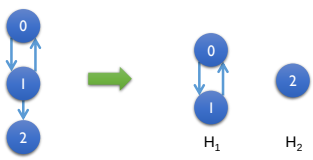


Not a strongly connected digraph!
There is no directed path from 2 to 0

14

Strongly Connected Component

- A **strongly connected component** is a maximal subgraph that is strongly connected.

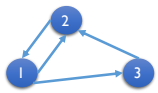


Two strongly connected components

15

Degree of a Vertex

- **Degree** of a vertex v:
  - The # of edges incident to v.
- In a directed graph:
  - **In-degree** of v
    - The # of edges for which v is the tail (incoming edges).
  - **Out-degree** of v
    - The # of edges for which v is the head (outgoing edges).
  - **Degree of v = in-degree + out-degree**



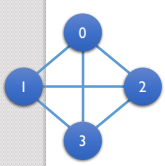
16

6.1.3 GRAPH REPRESENTATION

17

Adjacency Matrix

- A two dimensional array with the property that $a[i][j] = 1$ iff the edge (i, j) or $< i, j >$ is in $E(G)$.



$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 2 & 1 & 1 & 0 & 1 \\ 3 & 1 & 1 & 1 & 0 \end{array}$$

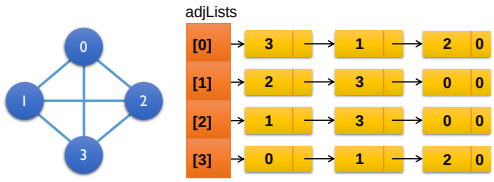
$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 2 & 0 & 0 & 0 \end{array}$$

- Waste of memory when a graph is sparse
  - Storage $O(n^2)$

18

Adjacency Lists

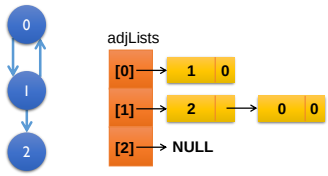
- Undirected graph: Use a chain to represent each vertex and its **adjacent** vertices.



19

Adjacency Lists

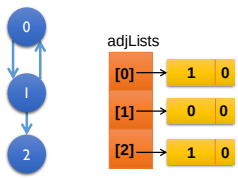
- Directed graph: Use a chain to represent each vertex and its **adjacent** vertices.
  - Length of list = out-degree of v



20

Inverse Adjacency Lists

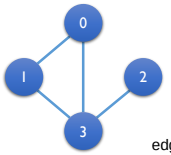
- Directed graph: Use a chain to represent each vertex and its **adjacent from** vertices
  - Length of list = in-degree of v



21

Sequential Representation

- Example: $n = 4, e = 4$
- Int nodes[$n + 2e + 1$] => nodes[13]



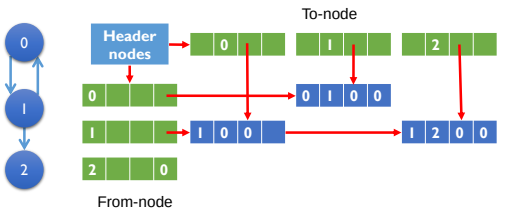
nodes edges

0	1	2	3	4	5	6	7	8	9	10	11	12
5	7	9	10	13	1	3	0	3	3	0	1	2

Adjacency Multilists

- Multilists: lists in which nodes may be shared among several lists

Veretex 1	Vertext 2	Link1	Link2



Weighted Edges

- Edges of a graph sometimes have weights associated with them.
  - **Distance** from one vertex to another.
  - **Cost** of going from one vertex to an adjacent vertex.
- We use additional field in each edge to store the weight.
- A graph with weighted edges is called a **network**.

How Many Kinds of Graphs ?

- 2 types:
  - Directed,
  - Undirected
- 2 edge types:
  - Weighted,
  - Unweighted
- 4 representations:
  - Adjacency matrix,
  - Adjacency lists,
  - Sequential lists,
  - Adjacency multilists

25

ADT: Graph

```
class Graph
{// object: A nonempty set of vertices and a set of undirected edges.
public:
  virtual ~Graph() {}                          // virtual destructor
  bool IsEmpty() const{return n == 0};         // return true iff graph has no
vertices
  int NumberOfVertices() const{return n}; // return the # of vertices
  int NumberOfEdges() const{return e};     // return the # of edges
  virtual int Degree(int u) const = 0;     // return the degree of a vertex
  virtual bool ExistsEdge(int u, int v) const = 0; // check the existence
of edge
  virtual void InsertVertex(int v) = 0;            // insert a vertex v
  virtual void InsertEdge(int u, int v) = 0;       // insert an edge (u, v)
  virtual void DeleteVertex(int v) = 0;            // delete a vertex v
  virtual void DeleteEdge(int u, int v) = 0;       // delete an edge (u, v)
  // More graph operations…
protected:
  int n; // number of vertices
  int e; // number of edges
};
```

26

Implementation Notes

To accommodate various graph types, we make the following assumptions:

- Data type of edge weight is **double** (or represented as a template parameter).
- We define operations which are **independent** of specific graph representation in the Graph.
- We assume the **iterator** is used to visit adjacent vertices.

27

Example: LinkedGraph

```
void Graph::foo(void){
  // use iterator to visit adjacent vertices of v
  for (each vertex w adjacent to v)…
}
```

```
class LinkedGraph: public Graph
{
public:
  // constructor
  LinkedGraph(const int vertices = 0) : n(vertices), e(0){
     adjLists = new Chain<int>[n];
  }
  // more customized operations…
private:
  Chain<int> *adjLists    // adjacency lists
};
```

28