05 Graph (1)

College of Computer Science, CQU

Outline

- Basic Concept
- Graph ADT
- Graph Representation
- Adjacency Matrix
- Adjacency List

Graph Applications

- Modeling connectivity in computer and communications networks.
- Representing a map as a set of locations with distances between locations; used to compute shortest routes between locations.
- Modeling flow capacities in transportation networks.
- Finding a path from a starting condition to a goal condition; for example, in artificial intelligence problem solving.
- Modeling computer algorithms, showing transitions from one program state to another.
- Finding an acceptable order for finishing subtasks in a complex activity, such as constructing large buildings.
- Modeling relationships such as family trees, business or military organizations, and scientific taxonomies

Basic Concept

- Graphs are a formalism useful for representing relationships between things
- A **graph G** consists of a set of **vertices** and a set of connections linking pairs of vertices. These pairs of vertices are called **edges.**
- A **graph** `G` is represented as `G = (V, E)`
  - `V` is a set of **vertices**: $\{v_1, \dots, v_n\}$
  - `E` is a set of **edges**: $\{e_1, \dots, e_m\}$ where each e_i connects two vertices (v_{i1}, v_{i2})
- Operations include:
  - iterating over vertices
  - iterating over edges
  - iterating over vertices adjacent to a specific vertex
  - asking whether an edge exists connects two vertices

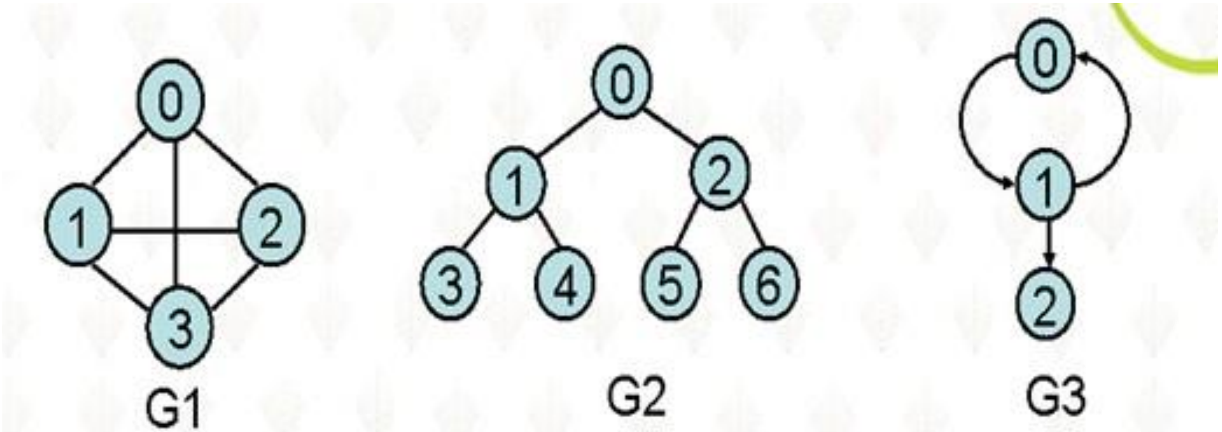
Basic Concept

- If each $<v_i,v_j>$ in the E is undirected, that is $<v_i,v_j>$ is same as $<v_j,v_i>$, G is called an undirected graph. In undirected graph, the edge $<v_i,v_j>$ can be written as is (v_i,v_j) .

- If each $<v_i,v_j>$ in the E is directed, G is called a directed graph. In directed graph, the edge $<v_i,v_j>$ is also called arcs.

- Complete graph: a graph that has the maximum number of edges
  - Undirected graph (n vertices)----n(n-1)/2
  - Directed graph (n vertices)-------n(n-1)

Example



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> }

- G1 and G2 are undirected graphs, and G3 is a directed graph.
- G2 is a tree -→tree is a special case of graphs

Basic Concept

- (v_i,v_j) : vertices v_i and v_j are adjacent (相邻的)

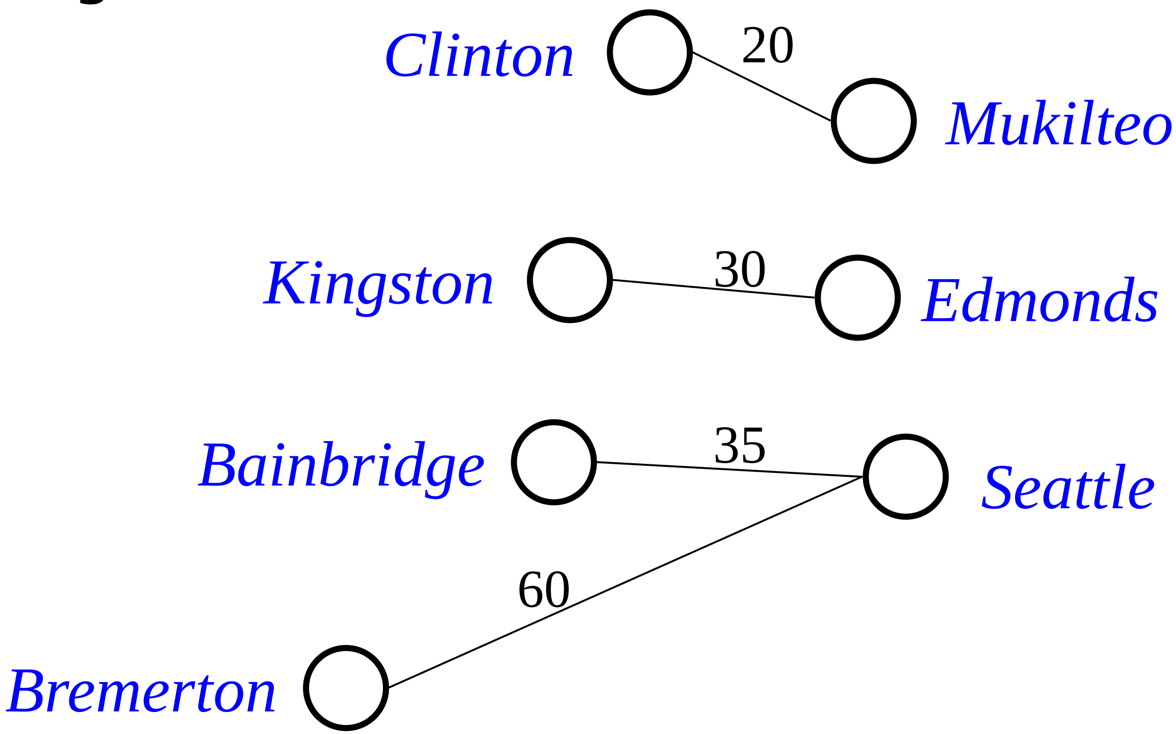
  edge (v_i,v_j) is incident on e_i and e_j (相关联)

- $<v_i,v_j>$ vertex v_i is adjacent to vertex v_j , vertex v_j is adjacent from vertex v_i . edge $<v_i,v_j>$ is incident on v_i and v_j (相关联)

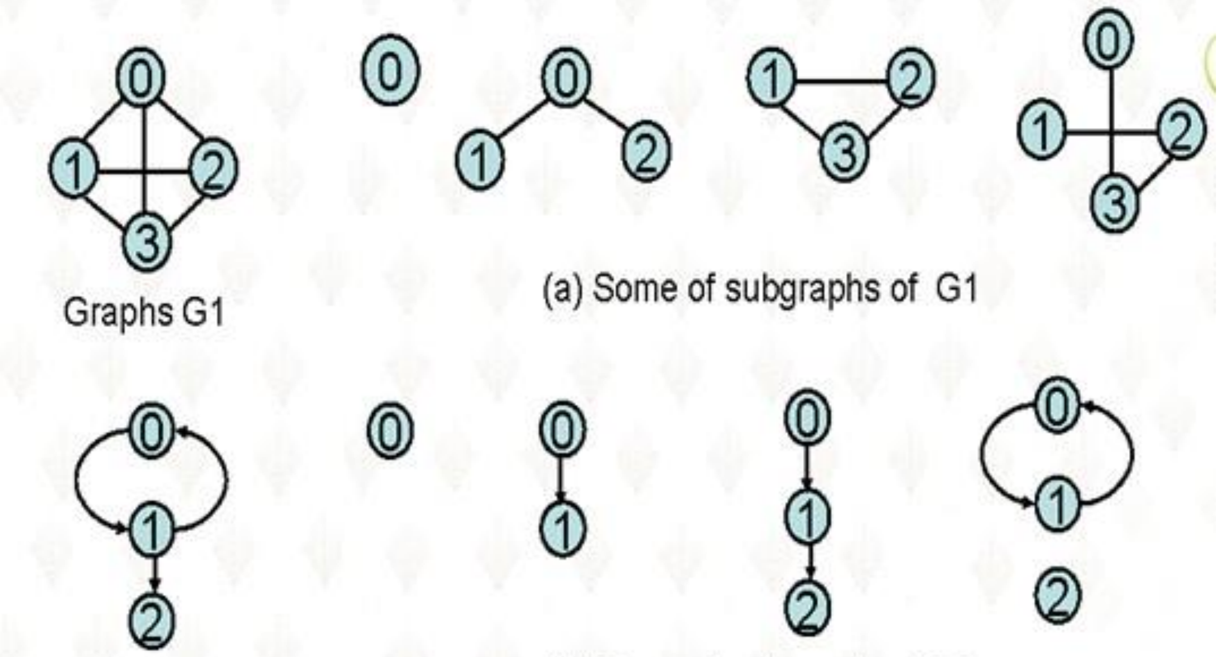
- Weighted graph: graphs whose each edge has a weight.
- Subgraph: Assume there are two graphs G=(V,E) and G'=(V',E'). If V'≤V,and E' ≤E, G' is called subgraph of G.

Weighted Graphs

- In a *weighted graph*, each edge has an associated weight or cost.



Example



Graphs G1

(a) Some of subgraphs of G1

Graphs G3

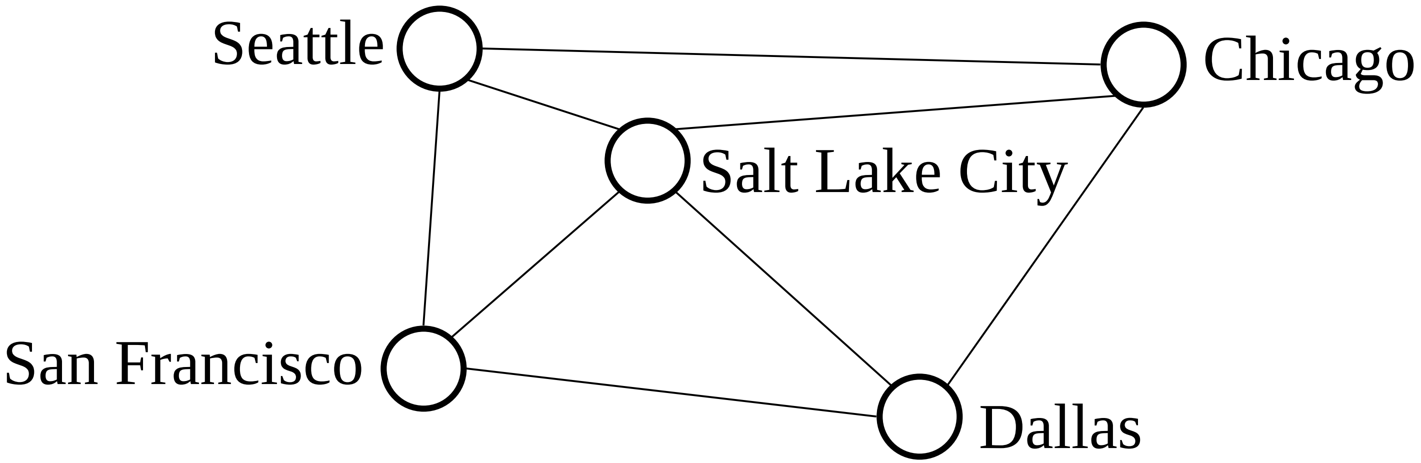
(b) Some of subgraphs of G3

Basic Concept

- **path** : A sequence of vertices $v_1, v_2, \ldots, v_n$ forms a **path** of length $n - 1$ if there exist edges from v_i to v_{i+1} for $1 \le i < n$.
- **Simple path**: if all vertices on the path are distinct.
- ***The length* of a path** : the number of edges it contains.
- **cycle:** a path of length three or more that connects some vertex v_1 to itself.
- **simple cycle** : if the path is simple, except for the first and last vertices being the same.

Paths

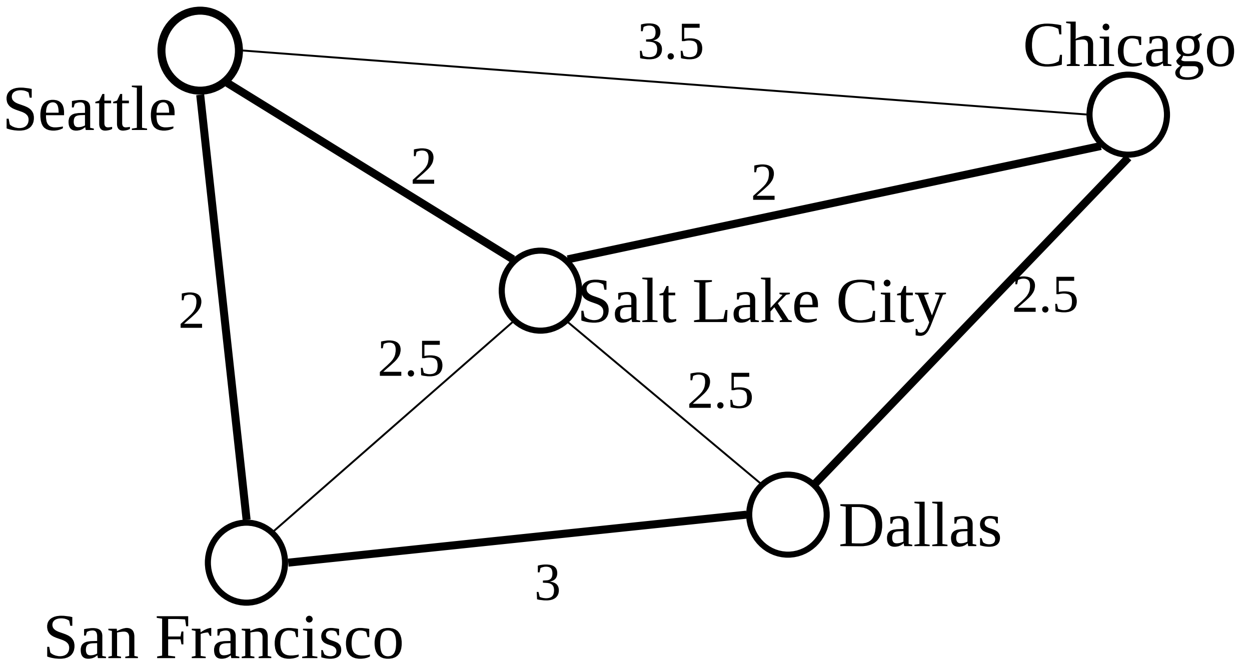
- A *path* is a list of vertices $\{v_1, v_2, \ldots, v_n\}$ such that $(v_i, v_{i+1}) \in E$ for all $0 \le i < n$.



p = {SEA, SLC, CHI, DAL, SFO, SEA}

Path Length and Cost

- **Path length**: the number of edges in the path
- **Path cost**: the sum of the costs of each edge



length(p) = 5

cost(p) = 11.5

p = {SEA, SLC, CHI, DAL, SFO, SEA}

Simple Paths and Cycles

- A *simple path* repeats no vertices (except that the first can be the last):
  - p = {Seattle, Salt Lake City, San Francisco, Dallas}
  - p = {Seattle, Salt Lake City, Dallas, San Francisco, Seattle}

- A *cycle* is a path that starts and ends at the same node:
  - p = {Seattle, Salt Lake City, Dallas, San Francisco, Seattle}

- A *simple cycle* is a cycle that repeats no vertices except that the first vertex is also the last (in undirected graphs, no edge can be repeated)

Example

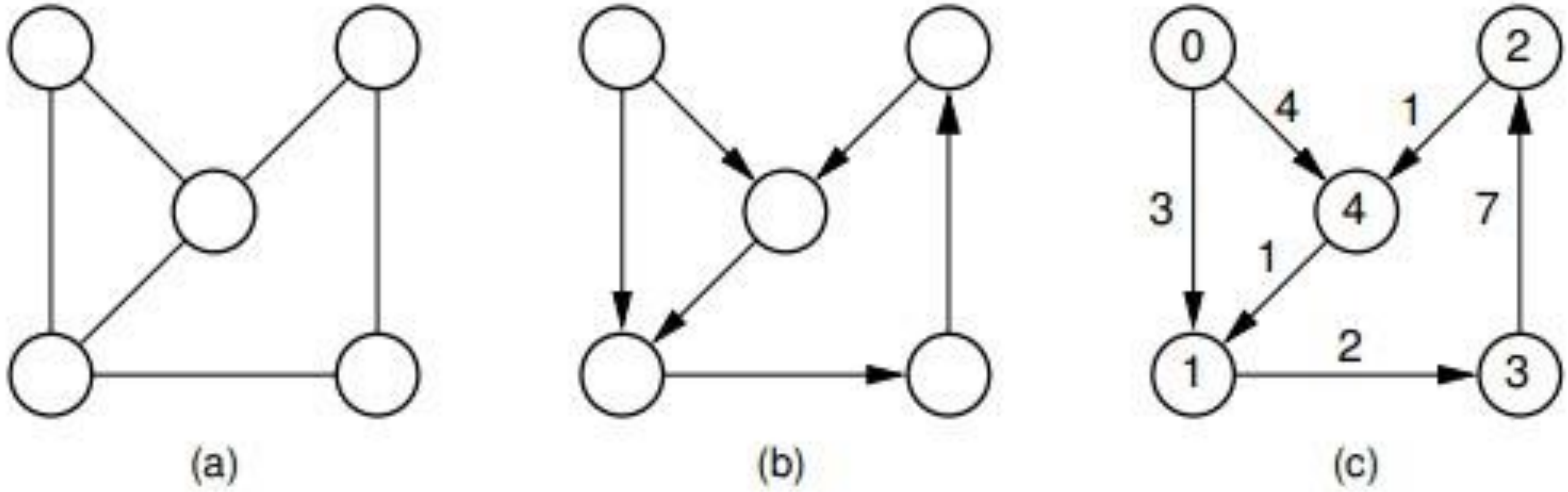


Figure 11.1 Examples of graphs and terminology. (a) A graph. (b) A directed graph (digraph). (c) A labeled (directed) graph with weights associated with the edges. In this example, there is a simple path from Vertex 0 to Vertex 3 containing Vertices 0, 1, and 3. Vertices 0, 1, 3, 2, 4, and 1 also form a path, but not a simple path because Vertex 1 appears twice. Vertices 1, 3, 2, 4, and 1 form a simple cycle.

Basic Concept

- **An undirected graph is connected** **:** if there is at least one path from any vertex to any other.
- **connected components** **:**The maximally connected subgraphs of an undirected graph. For example, Figure 11.2 shows an undirected graph with three connected components.

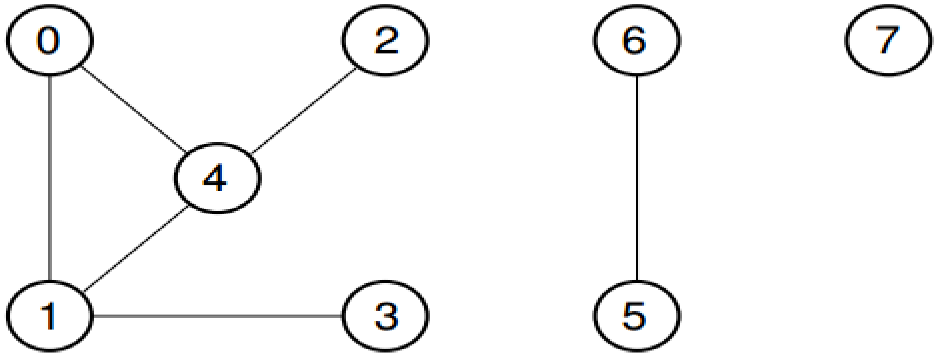


Figure 11.2 An undirected graph with three connected components. Vertices 0, 1, 2, 3, and 4 form one connected component. Vertices 5 and 6 form a second connected component. Vertex 7 by itself forms a third connected component.

Connectivity

- If there is a path from vertex v_i to v_j, v_i and v_j are connected.
- An undirected graph are connected-----if, for every pair of distinct vertices v_i , v_j, there is a path from v_i to v_j in G.
- Connected component(undirected graph)----a maximal connected subgraph.(a tree is graph that is connected and acycle(无环))
- Strongly connected(directed graph)--- if, for every pair of distinct vertices v_i , v_j, there is a path from v_i to v_j ,and also from v_j to v_i in G.

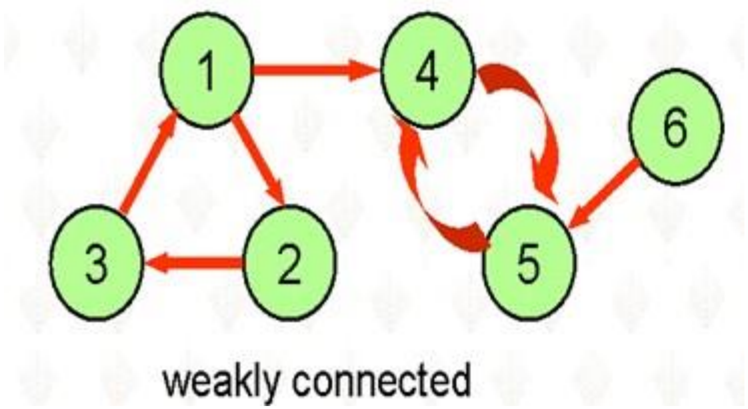
Connectivity

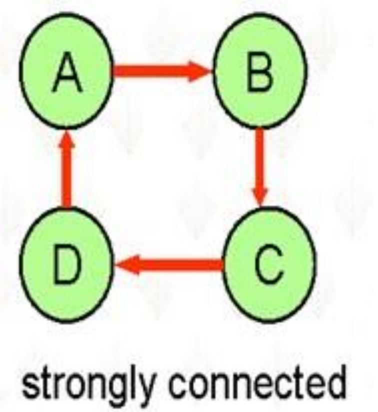
- Weakly connected(directed graph)---For a directed graph G, if the undirected graph obtained by suppressing the directions on the edges of G is connected.
- Strongly connected Component---a maximal subgraph that is strongly connected.
- Degree of v_i-----the number of edges incident to that vertex.
- In-degree---the number of edges that have v_i as the head.
- out-degree---the number of edges that have v_i as the tail.

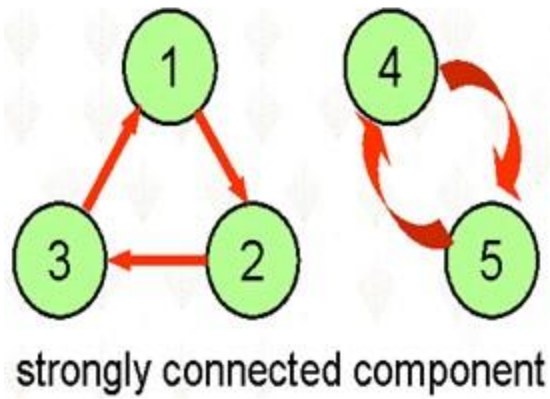
  the number of edges:

$$e = \frac{1}{2} \sum_{i=0}^{n-1} d_i$$

  Where d_i is the degree of vertex v_i


1
4
6
3
2
5
weakly connected


A
B
D
C
strongly connected

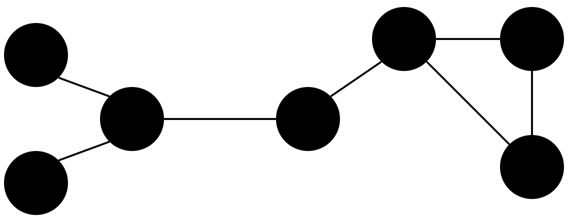

1
4
3
2
5
strongly connected component

Connectivity

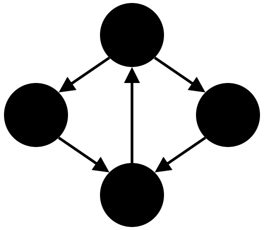
- Acycle –a graph without cycles.
- directed acyclic graph (DAG)--a directed graph without cycles.
- free tree—a connected, undirected graph with acycles.
- spanning tree—a subgraph of undirected graph G, which is connected and without cycles.

Connectivity

- Undirected graphs are *connected* if there is a path between any two vertices

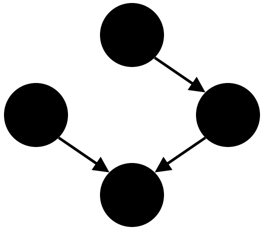


- Directed graphs are *strongly connected* if there is a path from any one vertex to any other

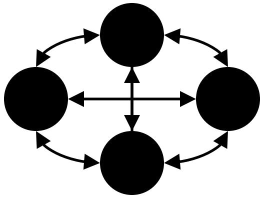


Connectivity

- Digraphs are *weakly connected* if there is a path between any two vertices, *ignoring direction*



- A *complete* graph has an edge between every pair of vertices



Basic Concept

- **Acyclic graph**: A graph without cycles is called **acyclic**.
- **directed acyclic graph (DAG):** a directed graph without cycles
- A **free tree** is a connected, undirected graph with no simple cycles. An equivalent definition is that a free tree is connected and has $|\mathbf{V}| - 1$ edges.

Graph Density

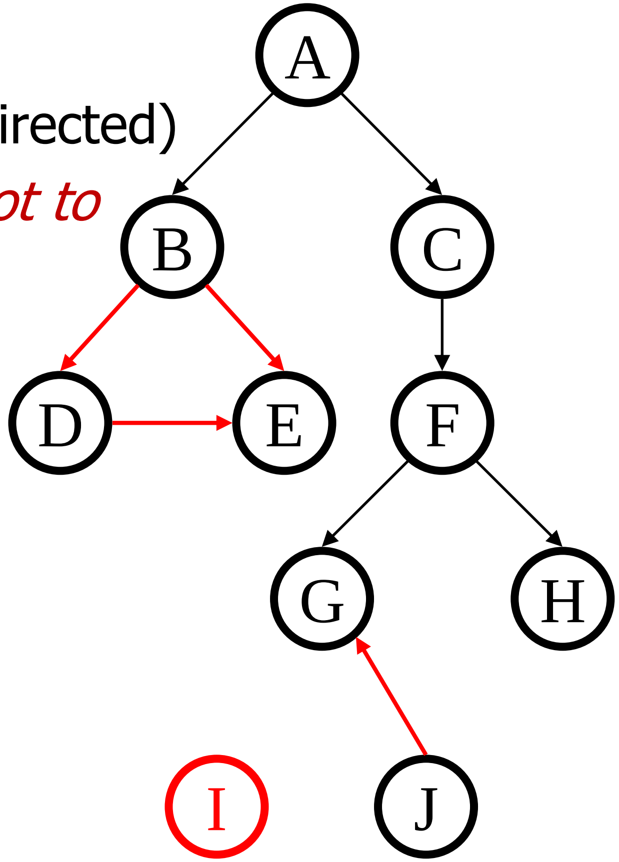
- A *sparse* graph has $O(|V|)$ edges

- A *dense* graph has $\Theta(|V|^2)$ edges

- Anything in between is either *sparsish* or *densy* depending on the context.

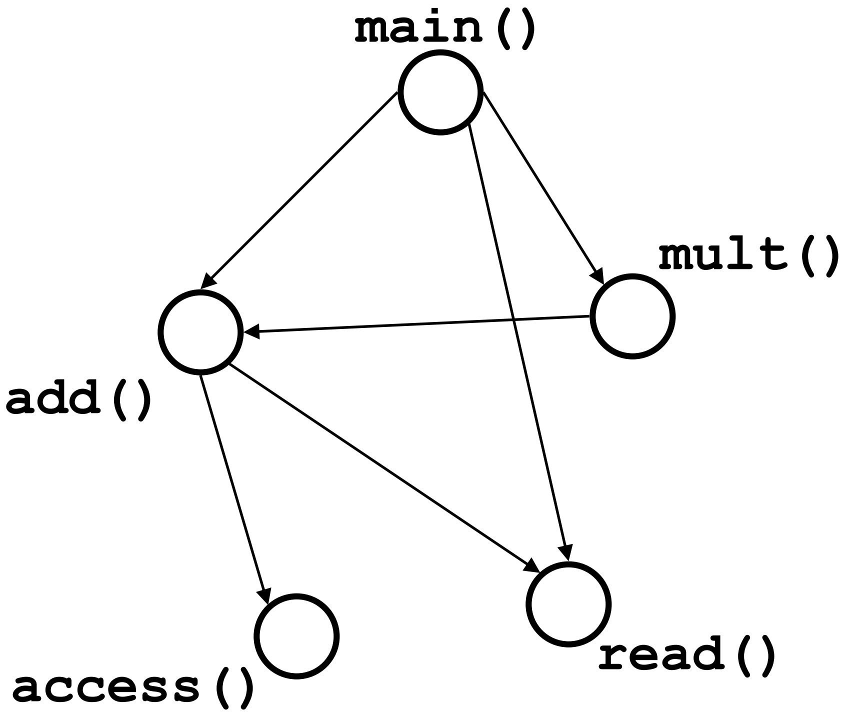
Trees as Graphs

- Every tree is a graph with some restrictions:
  - the tree is *directed*
  - there are *no cycles* (directed or undirected)
  - there is a *directed path from the root to every node*



Directed Acyclic Graphs (DAGs)

- DAGs are directed graphs with no cycles
- Trees $\subset$ DAGs $\subset$ Graphs



Graph ADT

```
// Graph abstract class. This ADT assumes that the number
// of vertices is fixed when the graph is created.
class Graph {
private:
    void operator =(const Graph&) {} // Protect assignment
    Graph(const Graph&) {} // Protect copy constructor
public:
    Graph() {} // Default constructor
    virtual ~Graph() {} // Base destructor

    // Initialize a graph of n vertices
    virtual void Init(int n) =0;

    // Return: the number of vertices and edges
    virtual int n() =0;
    virtual int e() =0;
```

Graph ADT

```
// Return v's first neighbor
virtual int first(int v) =0;

// Return v's next neighbor
virtual int next(int v, int w) =0;

// Set the weight for an edge
// i, j: The vertices
// wgt: Edge weight
virtual void setEdge(int v1, int v2, int wght) =0;

// Delete an edge
// i, j: The vertices
virtual void delEdge(int v1, int v2) =0;
```

Graph ADT

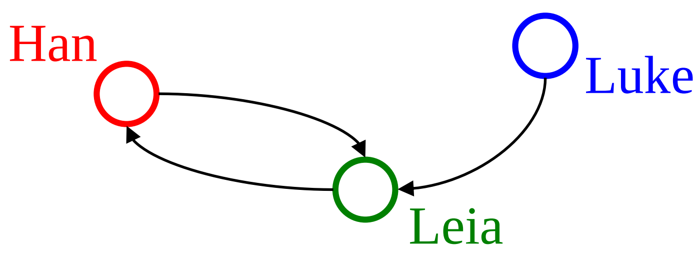
```
// Determine if an edge is in the graph
// i, j: The vertices
// Return: true if edge i,j has non-zero weight
virtual bool isEdge(int i, int j) =0;

// Return an edge’s weight
// i, j: The vertices
// Return: The weight of edge i,j, or zero
virtual int weight(int v1, int v2) =0;

// Get and Set the mark value for a vertex
// v: The vertex
// val: The value to set
virtual int getMark(int v) =0;
virtual void setMark(int v, int val) =0;
};
```

Graph Representations

- List of vertices + list of edges
- 2-D matrix of vertices (marking edges in the cells)
  "adjacency matrix"
- List of vertices each with a list of adjacent vertices
  "adjacency list"

Adjacency Matrix

- A `|V| x |V|` array in which an element `(u, v)` is true if and only if there is an edge from **`u`** to **`v`**



	Han	Luke	Leia
Han			
Luke			
Leia			

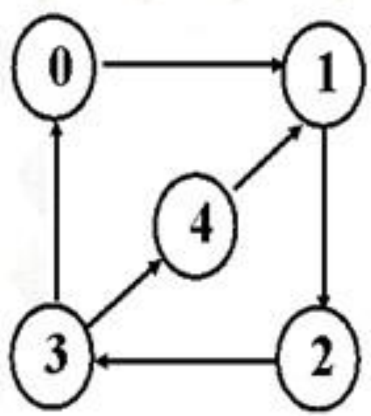
runtime:

space requirements:

Adjacency Matrix

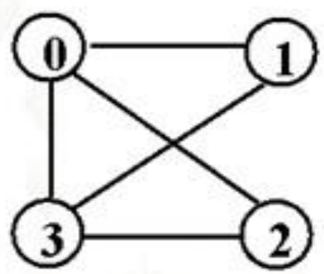
- A graph may be represented with a two-dimensional array. If G has n=|v| vertices, let M be an n×n matrix whose entries are defined by:

$$M_{ij} = \begin{cases} 1 & \text{if } \langle i,j \rangle \text{ is an edge} \\ 0 & \text{otherwise} \end{cases}$$



$$M = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Adjacency Matrix



$$M = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

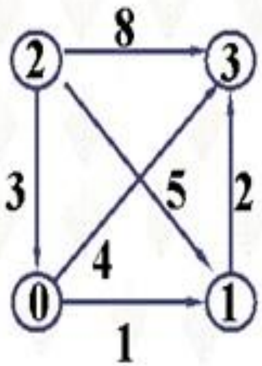
Worst case:

- O(1): to determine existence of a specific edge
- $O(|V|^2)$: storage cost
- O(|V|) : for finding all vertices accessible from a specific vertex
- O(1): to add or delete an edge
- Not easy to add or delete a vertex; better for static graph structure
- Symmetric (对称)：matrix for undirected graph; so half if redundant then.

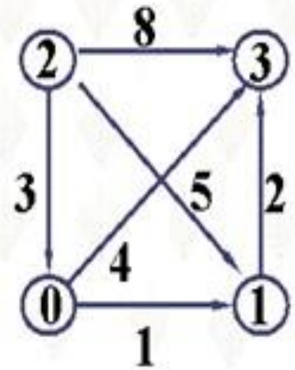
Adjacency Matrix

$$M_{ij}=\begin{cases} w_{ij} & \text{if } \langle i,j\rangle \in E,\ w_{ij} \text{ is the weight with} \\ 0 & \text{otherwise} \end{cases}$$

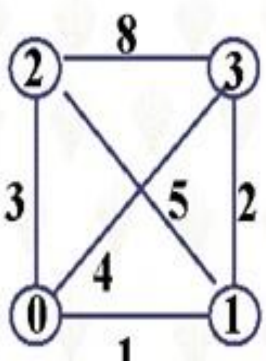
$$M_{ij}=\begin{cases} w_{ij} & \text{if } \langle i,j\rangle \in E,\ w_{ij} \text{ is the weight with} \\ \propto & \text{otherwise} \end{cases}$$



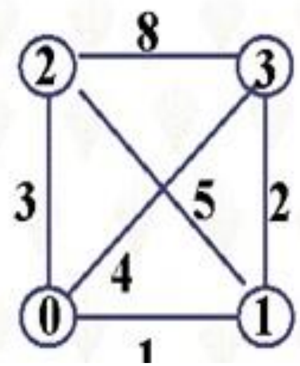
$$\mathbf{M}=\begin{bmatrix} 0 & 1 & 0 & 4 \\ 0 & 0 & 0 & 2 \\ 3 & 5 & 0 & 8 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$



$$\mathbf{M}=\begin{bmatrix} \infty & 1 & \infty & 4 \\ \infty & \infty & \infty & 2 \\ 3 & 5 & \infty & 8 \\ \infty & \infty & \infty & \infty \end{bmatrix}$$



$$\mathbf{M}=\begin{bmatrix} 0 & 1 & 3 & 4 \\ 1 & 0 & 5 & 2 \\ 3 & 5 & 0 & 8 \\ 4 & 2 & 8 & 0 \end{bmatrix}$$



$$\mathbf{M}=\begin{bmatrix} \infty & 1 & 3 & 4 \\ 1 & \infty & 5 & 2 \\ 3 & 5 & \infty & 8 \\ 4 & 2 & 8 & \infty \end{bmatrix}$$

Adjacency Matrix Implementation

```
// Implementation for the adjacency matrix representation
class Graphm : public Graph {
private:
  int numVertex, numEdge; // Store number of vertices, edges
  int **matrix;           // Pointer to adjacency matrix
  int *mark;              // Pointer to mark array
public:
  Graphm(int numVert)     // Constructor
    { Init(numVert); }

  ~Graphm() {         // Destructor
    delete [] mark; // Return dynamically allocated memory
    for (int i=0; i<numVertex; i++)
      delete [] matrix[i];
    delete [] matrix;
  }
```

Adjacency Matrix Implementation

```
void Init(int n) { // Initialize the graph
  int i;
  numVertex = n;
  numEdge = 0;
  mark = new int[n];      // Initialize mark array
  for (i=0; i<numVertex; i++)
    mark[i] = UNVISITED;
  matrix = (int**) new int*[numVertex]; // Make matrix
  for (i=0; i<numVertex; i++)
    matrix[i] = new int[numVertex];
  for (i=0; i< numVertex; i++) // Initialize to 0 weights
    for (int j=0; j<numVertex; j++)
      matrix[i][j] = 0;
}
```

Adjacency Matrix Implementation

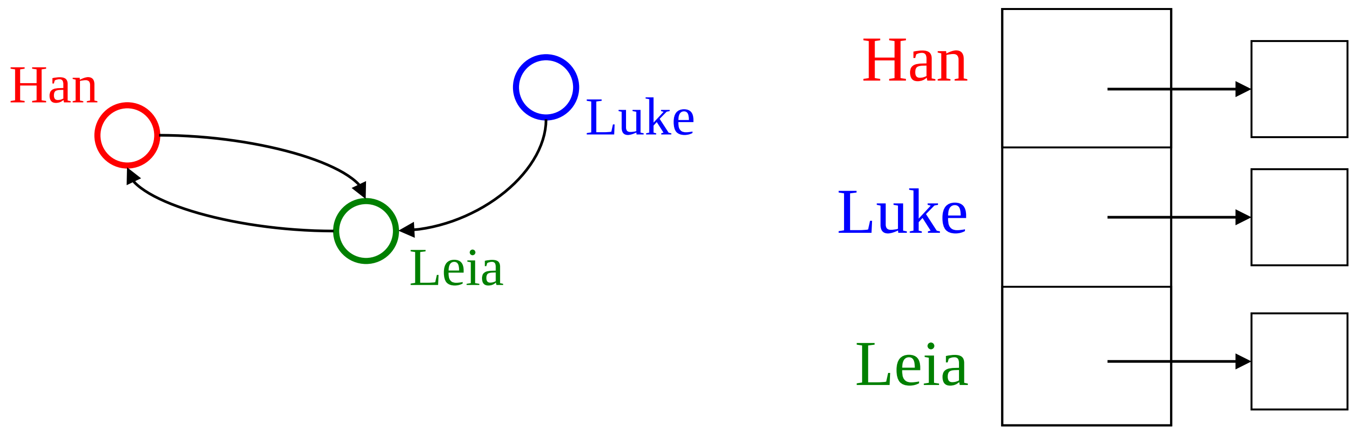
```
int n() { return numVertex; } // Number of vertices
int e() { return numEdge; }   // Number of edges

// Return first neighbor of "v"
int first(int v) {
  for (int i=0; i<numVertex; i++)
    if (matrix[v][i] != 0) return i;
  return numVertex;             // Return n if none
}

// Return v's next neighbor after w
int next(int v, int w) {
  for(int i=w+1; i<numVertex; i++)
    if (matrix[v][i] != 0)
      return i;
  return numVertex;             // Return n if none
}
```

Adjacency Matrix Implementation

```
  // Set edge (v1, v2) to "wt"
  void setEdge(int v1, int v2, int wt) {
    Assert(wt>0, "Illegal weight value");
    if (matrix[v1][v2] == 0) numEdge++;
    matrix[v1][v2] = wt;
  }

  void delEdge(int v1, int v2) { // Delete edge (v1, v2)
    if (matrix[v1][v2] != 0) numEdge--;
    matrix[v1][v2] = 0;
  }

  bool isEdge(int i, int j) // Is (i, j) an edge?
  { return matrix[i][j] != 0; }

  int weight(int v1, int v2) { return matrix[v1][v2]; }
  int getMark(int v) { return mark[v]; }
  void setMark(int v, int val) { mark[v] = val; }
};
```

Adjacency List

- A |V|-ary list (array) in which each entry stores a list (linked list) of all adjacent vertices (or edges)



runtime:
space requirements:

Adjacency List

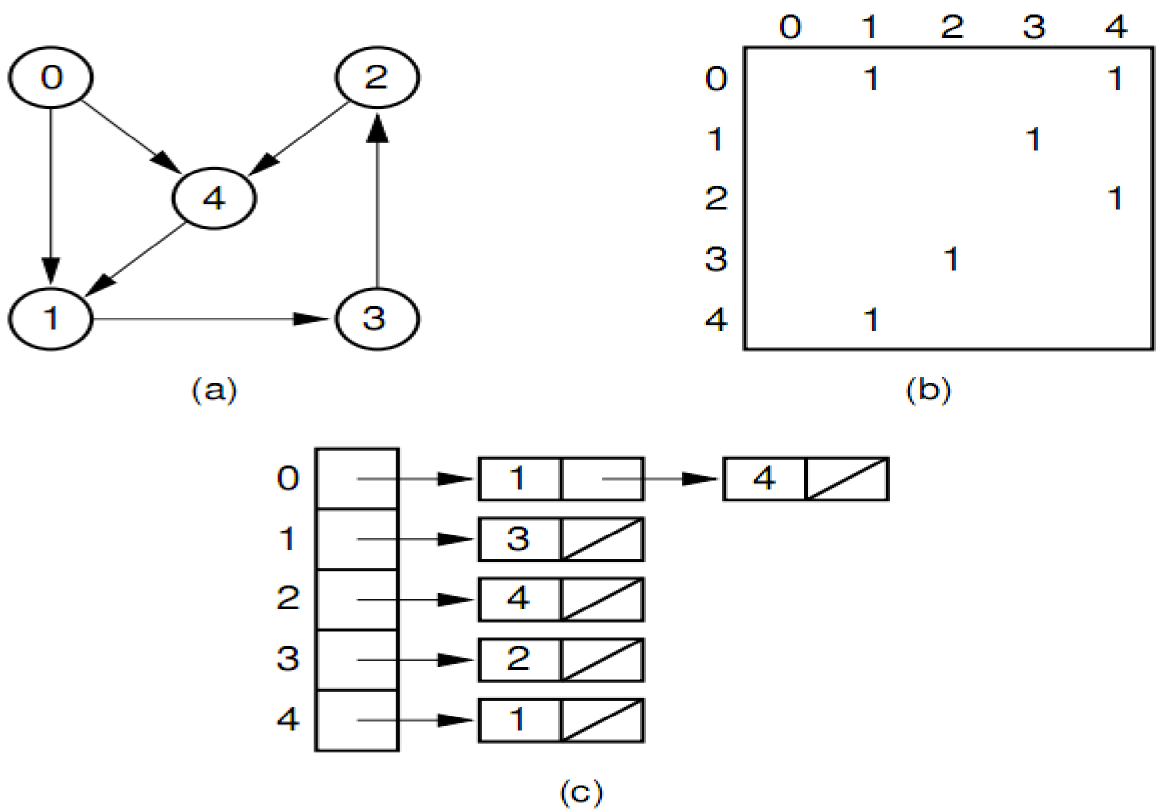


Figure 11.3 Two graph representations. (a) A directed graph. (b) The adjacency matrix for the graph of (a). (c) The adjacency list for the graph of (a).

Adjacency List

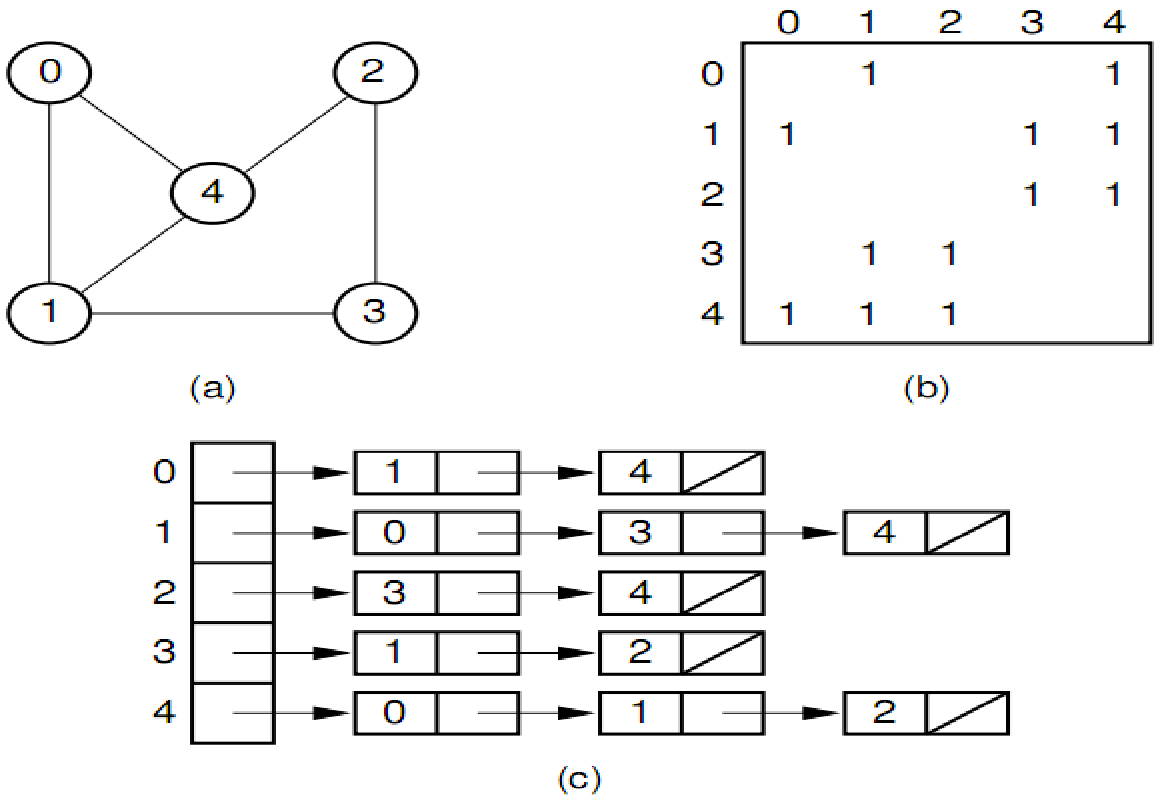
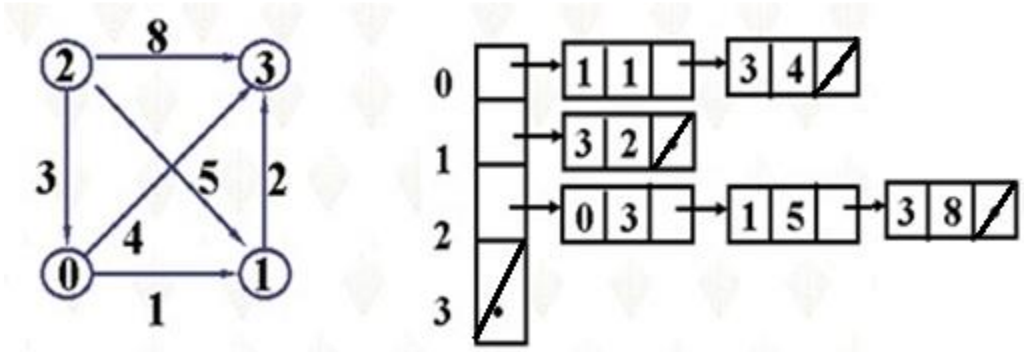


Figure 11.4 Using the graph representations for undirected graphs. (a) An undirected graph. (b) The adjacency matrix for the graph of (a). (c) The adjacency list for the graph of (a).

Adjacency List



Worst case:

- O(|V|): to determine existence of a specific edge
- O(|V|+|E|) : storage cost
- O(|V|) : for finding all neighbors of a specific vertex
- O(|V|): to add or delete an edge
- Still not easy to add or delete a vertex; however, we can use a linked list in place of the array

Adjacency List Implementation

```
// Edge class for Adjacency List graph representation
class Edge {
   int vert, wt;
public:
   Edge() { vert = -1; wt = -1; }
   Edge(int v, int w) { vert = v; wt = w; }
   int vertex() { return vert; }
   int weight() { return wt; }
};
```

Adjacency List Implementation

```
class Graphl : public Graph {
private:
  List<Edge>** vertex;          // List headers
  int numVertex, numEdge;       // Number of vertices, edges
  int *mark;                    // Pointer to mark array
public:
  Graphl(int numVert)
    { Init(numVert); }

  ~Graphl() {          // Destructor
    delete [] mark; // Return dynamically allocated memory
    for (int i=0; i<numVertex; i++) delete [] vertex[i];
    delete [] vertex;
  }
```

Adjacency List Implementation

```
void Init(int n) {
  int i;
  numVertex = n;
  numEdge = 0;
  mark = new int[n];  // Initialize mark array
  for (i=0; i<numVertex; i++) mark[i] = UNVISITED;
  // Create and initialize adjacency lists
  vertex = (List<Edge>**) new List<Edge>*[numVertex];
  for (i=0; i<numVertex; i++)
    vertex[i] = new LList<Edge>();
}

int n() { return numVertex; } // Number of vertices
int e() { return numEdge; }   // Number of edges
```

Adjacency List Implementation

```
int first(int v) { // Return first neighbor of "v"
  if (vertex[v]->length() == 0)
    return numVertex;        // No neighbor
  vertex[v]->moveToStart();
  Edge it = vertex[v]->getValue();
  return it.vertex();
}
```

Adjacency List Implementation

```
// Get v's next neighbor after w
int next(int v, int w) {
  Edge it;
  if (isEdge(v, w)) {
    if ((vertex[v]->currPos()+1) < vertex[v]->length()) {
      vertex[v]->next();
      it = vertex[v]->getValue();
      return it.vertex();
    }
  }
  return n(); // No neighbor
}
```

Adjacency List Implementation

```
bool isEdge(int i, int j) { // Is (i,j) an edge?
  Edge it;
  for (vertex[i]->moveToStart();
       vertex[i]->currPos() < vertex[i]->length();
       vertex[i]->next()) {                // Check whole list
    Edge temp = vertex[i]->getValue();
    if (temp.vertex() == j) return true;
  }
  return false;
}

 void delEdge(int i, int j) {  // Delete edge (i, j)
   if (isEdge(i,j)) {
     vertex[i]->remove();
     numEdge--;
   }
 }
```

Adjacency List Implementation

```
// Set edge (i, j) to "weight"
void setEdge(int i, int j, int weight) {
  Assert(weight>0, "May not set weight to 0");
  Edge currEdge(j, weight);
  if (isEdge(i, j)) { // Edge already exists in graph
    vertex[i]->remove();
    vertex[i]->insert(currEdge);
  }
  else { // Keep neighbors sorted by vertex index
    numEdge++;
    for (vertex[i]->moveToStart();
         vertex[i]->currPos() < vertex[i]->length();
         vertex[i]->next()) {
      Edge temp = vertex[i]->getValue();
      if (temp.vertex() > j) break;
    }
    vertex[i]->insert(currEdge);
  }
}
```

Adjacency List Implementation

```
  int weight(int i, int j) { // Return weight of (i, j)
    Edge curr;
    if (isEdge(i, j)) {
      curr = vertex[i]->getValue();
      return curr.weight();
    }
    else return 0;
  }

  int getMark(int v) { return mark[v]; }
  void setMark(int v, int val) { mark[v] = val; }
};
```

Knowledge Points

- Chapter 11, pp.381-392

Homework

- P410, 11.3

-End-