

# COMP9024: Data Structures and Algorithms

## Graphs (I)

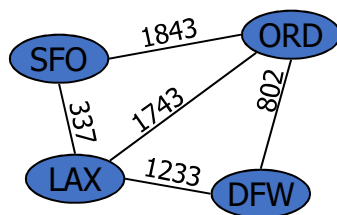
1

### Contents

- Graph terminology
- Adjacency matrix representation
- Adjacency list representation

2

# Graphs

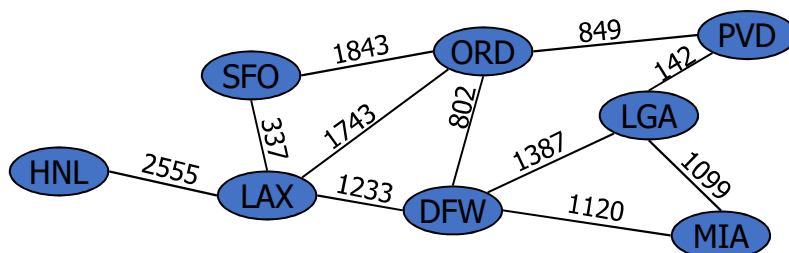


3

3

# Graphs

- A graph is a pair  $(V, E)$ , where
  - $V$  is a set of nodes, called **vertices**
  - $E$  is a collection of pairs of vertices, called **edges**
  - Vertices and edges are positions and store elements
- Example:
  - A vertex represents an airport and stores the three-letter airport code
  - An edge represents a flight route between two airports and stores the mileage of the route

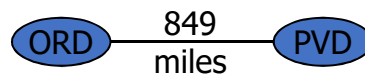


4

4

## Edge Types

- Directed edge
  - ordered pair of vertices  $(u,v)$
  - first vertex  $u$  is the origin
  - second vertex  $v$  is the destination
  - e.g., a flight
- Undirected edge
  - unordered pair of vertices  $(u,v)$
  - e.g., a flight route
- Directed graph
  - all the edges are directed
  - e.g., route network
- Undirected graph
  - all the edges are undirected
  - e.g., flight network

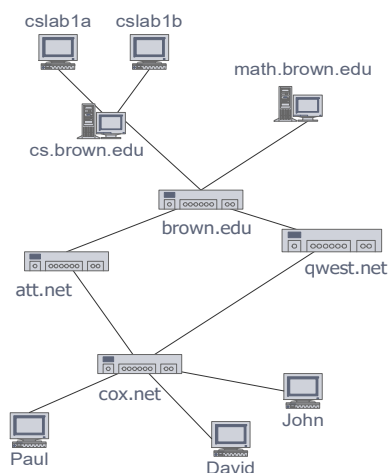


5

5

## Applications

- Electronic circuits
  - Printed circuit board
  - Integrated circuit
- Transportation networks
  - Highway network
  - Flight network
- Computer networks
  - Local area network
  - Internet
  - Web
- Databases
  - Entity-relationship diagram

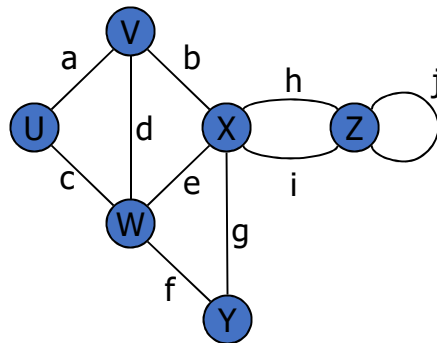


6

6

## Terminology (1/5)

- End vertices (or endpoints) of an edge
  - U and V are the endpoints of a
- Edges incident on a vertex
  - a, d, and b are incident on V
- Adjacent vertices
  - U and V are adjacent
- Parallel edges
  - h and i are parallel edges
- Self-loop
  - j is a self-loop

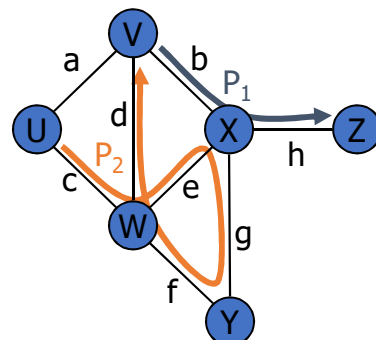


7

7

## Terminology (2/5)

- Path
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
  - each edge is preceded and followed by its endpoints
- Simple path
  - path such that all its vertices and edges are distinct
- Examples
  - $P_1 = (V, b, X, h, Z)$  is a simple path
  - $P_2 = (U, c, W, e, X, g, Y, f, W, d, V)$  is a path that is not simple

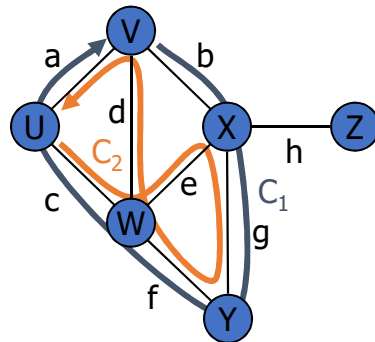


8

8

## Terminology (3/5)

- Cycle
  - circular sequence of alternating vertices and edges
  - each edge is preceded and followed by its endpoints
- Simple cycle
  - cycle such that all its vertices and edges are distinct
- Examples
  - $C_1 = (V, b, X, g, Y, f, W, c, U, a, \rightarrow)$  is a simple cycle
  - $C_2 = (U, c, W, e, X, g, Y, f, W, d, V, a, \rightarrow)$  is a cycle that is not simple

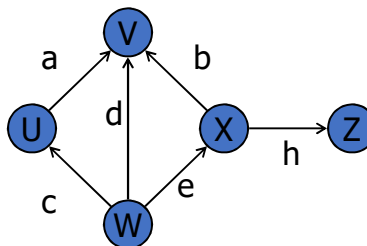
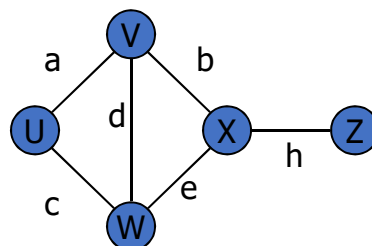


9

9

## Terminology (4/5)

- Degree of a vertex in a undirected graph
  - The number of edges
  - for example, the degree of V is 3
- Indegree (outdegree) of a vertex (directed graph)
  - The number of incoming (outgoing) edges
  - For example, the indegree of V is 3 and its out degree is 0

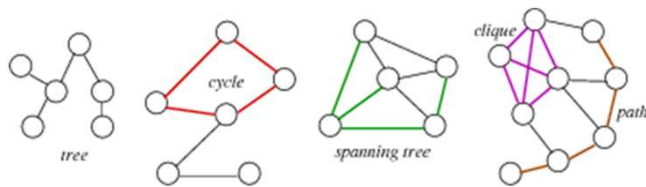


10

10

## Terminology (5/5)

- Tree: connected graph with no cycles
- Spanning tree: tree containing all vertices
- Clique: complete subgraph



11

## Properties

### Property 1

$$\sum_v \deg(v) = 2m$$

Proof: each edge is counted twice

### Property 2

In an undirected graph with no self-loops and no multiple edges

$$m \leq n(n-1)/2$$

Proof: each vertex has degree at most  $(n-1)$

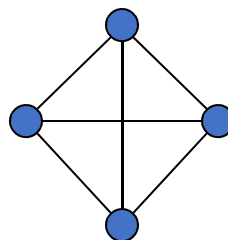
What is the bound for a directed graph?

### Notation

$n$	number of vertices
$m$	number of edges
$\deg(v)$	degree of vertex $v$

### Example

- $n = 4$
- $m = 6$
- $\deg(v) = 3$



12

12

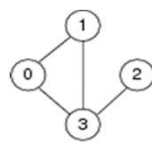
# Graph Representations

- Adjacency lists
- Adjacency matrix
- Both representations map vertices into integers in  $[0, n-1]$ , where  $n$  is the number of vertices.

13

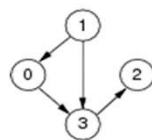
## Adjacency matrix (1/8)

- Edges represented by a  $n \times n$  matrix



*Undirected graph*

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



*Directed graph*

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

14

## Adjacency matrix (2/8)

- Advantages
  - easily implemented as 2-dimensional array
  - can represent graphs, digraphs and weighted graphs
    - ❑ undirected graphs: symmetric boolean matrix
    - ❑ digraphs (directed graphs): non-symmetric boolean matrix
    - ❑ weighted graphs: non-symmetric matrix of weight values
- Disadvantages:
  - if few edges (sparse)  $\Rightarrow$  memory-inefficient

15

## Adjacency matrix (3/8)

Graph initialization

newGraph(n):

**Input:** number of nodes n

**Output:** new empty graph

```
g.nV = n;  
g.nE = 0;  
allocate memory to g.edges[][]  
for all i,j=0...n-1 do  
    g.edges[i][j]=0;  
return g;
```

16



## Adjacency matrix (4/8)

### Edge insertion

insertEdge(g,(v,w))

**Input:** graph g, edge (v,w)

```
if ( g.edges[v][w]= 0 )
{ g.edges[v][w]=1;
  g.edges[w][v]=1;
  g.nE=g.nE+1;
}
```

17

## Adjacency matrix (5/8)

### Edge removal

removeEdge(g,(v,w))

**Input** graph g, edge (v,w)

```
if ( g.edges[v][w]≠0)
{
  g.edges[v][w]=0;
  g.edges[w][v]=0;
  g.nE=g.nE-1;
}
```

18

## Adjacency matrix (6/8)

Write an algorithm to output all edges of a graph (no duplicates!)

show(g)

**Input:** graph g

```
for all i=0 to g.nV-1 do  
  for all j=i+1 to g.nV-1 do  
    if ( g.edges[i][j]≠0 )  
      print i—"j";
```

19

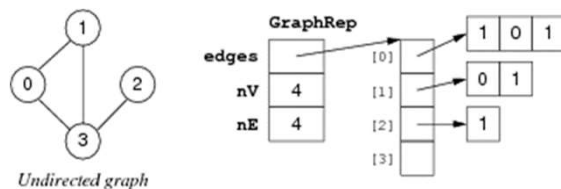
## Adjacency matrix (7/8)

- Space complexity:  $O(n^2)$ 
  - if a graph is sparse, most storage is wasted.
- Time complexity:
  - initialisation:  $O(n^2)$  (initialise  $n \times n$  matrix)
  - insert an edge:  $O(1)$  (set two cells in matrix)
  - delete an edge:  $O(1)$  (unset two cells in matrix)

20

## Adjacency matrix (8/8)

A space optimisation: store only top-right part of matrix.



New space complexity:

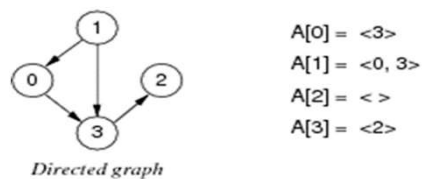
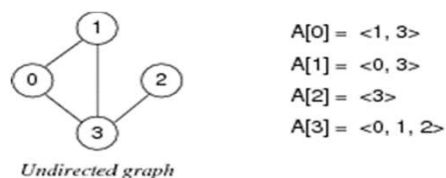
- $n-1$  int ptrs +  $n(n-1)/2$  ints (but still  $O(n^2)$ )

Requires us to always use edges  $(v,w)$  such that  $v < w$ .

21

## Adjacency List (1/6)

- For each vertex, store linked list of adjacent vertices:



22

## Adjacency List (2/6)

- Advantages
  - relatively easy to implement in languages like C
  - memory efficient if E:V relatively small
- Disadvantages:
  - one graph has many possible representations unless lists are ordered by same criterion e.g. ascending

23

## Adjacency List (3/6)

Graph initialization

`newGraph(n)`

**Input:** number of nodes `n`

**Output:** new empty graph

`g.nV = n;`

`g.nE = 0;`

`allocate memory for g.edges[];`

**for** all `i=0..n-1` **do**

`g.edges[i]=NULL;`

**return** `g`

24

## Adjacency List (4/6)

### Edge insertion

insertEdge(g,(v,w))

**Input:** graph g, edge (v,w)

```
if ( inLL(g.edges[v],w) )
{ insertLL(g.edges[v],w);
  insertLL(g.edges[w],v);
  g.nE=g.nE+1;
}
```

25

## Adjacency List (5/6)

### Edge removal

removeEdge(g,(v,w))

**Input:** graph g, edge (v,w)

```
if ( inLL(g.edges[v],w) )
{
  deleteLL(g.edges[v],w);
  deleteLL(g.edges[w],v);
  g.nE=g.nE-1;
}
```

26

## Adjacency List (6/6)

Analyse space complexity and time complexity of adjacency list representation:

- Space complexity:  $O(n+m)$ , where  $m$  is the number of edges
- Time complexity:
  - initialisation:  $O(n)$  (initialise  $n$  lists)
  - insert an edge:  $O(1)$  (insert one vertex into one list (digraph) or two lists (undirected graph)) if don't check for duplicates
  - delete edge:  $O(m)$  (need to find vertex in list(s))
  - If vertex lists are sorted
    - ❑ insertion requires search of list  $\Rightarrow O(m)$
    - ❑ deletion always requires a search, regardless of list order

27

## Comparison of Graph Representations

	adjacency matrix	adjacency list
space usage	$n^2$	$n+m$
initialise	$n^2$	$n$
insert edge	$1$	$1$
remove edge	$1$	$m$

	adjacency matrix	adjacency list
disconnected( $v$ )?	$n$	$1$
isPath( $x,y$ )?	$n^2$	$n+m$
copy graph	$n^2$	$n+m$
destroy graph	$n$	$n+m$

28

## Graph Abstract Data Type (1/2)

Data:

- set of edges, set of vertices

Operations:

- insertion: create graph, add edge
- deletion: remove edge, delete whole graph
- search: check if graph contains a given edge

Things to note:

- the set of vertices is fixed when a graph is initialised
- we treat vertices as ints, but could be arbitrary Items

29

## Graph Abstract Data Type (2/2)

Graph ADT interface graph.h

```
typedef struct GraphRep *Graph;
```

```
typedef int Vertex;
```

```
typedef struct Edge { Vertex v; Vertex w; } Edge;
```

```
Graph newGraph(int V);
```

```
void insertEdge(Graph, Edge);
```

```
void removeEdge(Graph, Edge);
```

```
bool adjacent(Graph, Vertex, Vertex);
```

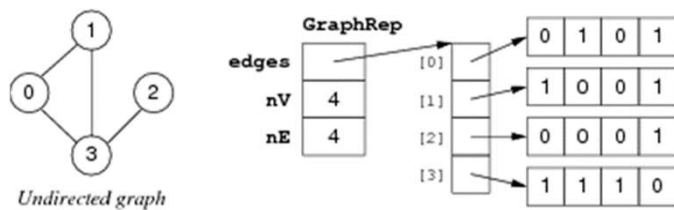
```
void freeGraph(Graph);
```

30

## Graph Implementation with Adjacency Matrix (1/4)

Implementation of GraphRep (adjacency-matrix representation)

```
typedef struct GraphRep {  
    int **edges;  
    int  nV;  
    int  nE;  
} GraphRep;
```



31

## Graph Implementation with Adjacency Matrix (2/4)

Implementation of graph initialisation (adjacency-matrix representation)

```
Graph newGraph(int n) {  
    assert(n >= 0);  
    int i;  
    Graph g = malloc(sizeof(GraphRep));  
    assert(g != NULL); g->nV = n; g->nE = 0;  
    g->edges = malloc(n * sizeof(int *));  
    assert(g->edges != NULL);  
    for (i = 0; i < n; i++) {  
        g->edges[i] = calloc(n, sizeof(int)); assert(g->edges[i] != NULL);  
    }  
    return g;  
}
```

32



## Graph Implementation with Adjacency Matrix (3/4)

Implementation of edge insertion/removal (adjacency-matrix representation)

```
bool validV(Graph g, Vertex v)
{ return (g != NULL && v >= 0 && v < g->nV);}

void insertEdge(Graph g, Edge e) {
    assert(g != NULL && validV(g,e.v) && validV(g,e.w));
    if (!g->edges[e.v][e.w]) {
        g->edges[e.v][e.w] = 1; g->edges[e.w][e.v] = 1; g->nE++; }

void removeEdge(Graph g, Edge e) {
    assert(g != NULL && validV(g,e.v) && validV(g,e.w));
    if (g->edges[e.v][e.w]) {
        g->edges[e.v][e.w] = 0; g->edges[e.w][e.v] = 0; g->nE--; }}
```

33

## Graph Implementation with Adjacency Matrix (4/4)

Implement a function to check whether two vertices are directly connected by an edge

```
bool adjacent(Graph g, Vertex x, Vertex y) {
    assert(g != NULL && validV(g,x) && validV(g,y));

    return (g->edges[x][y] != 0);
}
```

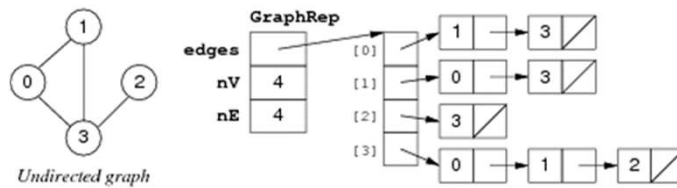
34

## Graph Implementation with Adjacency Lists (1/7)

Implementation of GraphRep (adjacency-list representation)

```
typedef struct GraphRep {
    Node **edges;
    int nV;
    int nE;
} GraphRep;
```

```
typedef struct Node {
    Vertex v;
    struct Node *next;
} Node;
```



35

## Graph Implementation with Adjacency Lists (2/7)

Implementation of graph initialisation (adjacency-list representation)

```
Graph newGraph(int n) {
    int i;
    assert(n >= 0);
    Graph g = malloc(sizeof(GraphRep));
    assert(g != NULL);
    g->nV = n; g->nE = 0;
    g->edges = malloc(nV * sizeof(Node *));
    assert(g->edges != NULL);
    for (i = 0; i < n; i++)
        g->edges[i] = NULL;
    return g;
}
```

36

## Graph Implementation with Adjacency Lists (3/7)

Implementation of edge insertion/removal (adjacency-list representation)

```
void insertEdge(Graph g, Edge e) {
    assert(g != NULL && validV(g,e.v) && validV(g,e.w));

    if (!inLL(g->edges[e.v], e.w)) {
        g->edges[e.v] = insertLL(g->edges[e.v], e.w);
        g->edges[e.w] = insertLL(g->edges[e.w], e.v);
        g->nE++;
    }
}
```

37

## Graph Implementation with Adjacency Lists (4/7)

```
void removeEdge(Graph g, Edge e) {
    assert(g != NULL && validV(g,e.v) && validV(g,e.w));

    if (inLL(g->edges[e.v], e.w)) {
        g->edges[e.v] = deleteLL(g->edges[e.v], e.w);
        g->edges[e.w] = deleteLL(g->edges[e.w], e.v);
        g->nE--;
    }
}
```

inLL, insertLL, deleteLL are standard linked list operations

38

## Graph Implementation with Adjacency Lists (5/7)

Assuming an adjacency list representation, implement a function to check whether two vertices are directly connected by an edge

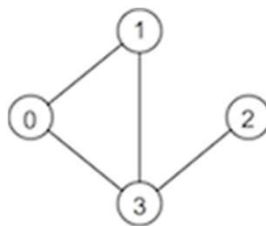
```
bool adjacent(Graph g, Vertex x, Vertex y) {  
    assert(g != NULL && validV(g,x));  
  
    return inLL(g->edges[x], y);  
}
```

39

## Graph Implementation with Adjacency Lists (6/7)

Write a program that uses the graph ADT to

- build the graph depicted below
- print all the nodes that are incident to vertex 1 in ascending order



40

## Graph Implementation with Adjacency Lists (7/7)

```
#include <stdio.h>
#include "Graph.h"
#define NODES 4
#define NODE_OF_INTEREST 1
int main(void) {
    Graph g = newGraph(NODES); Edge e; int i;
    e.v = 0; e.w = 1; insertEdge(g,e);
    e.v = 0; e.w = 3; insertEdge(g,e);
    e.v = 1; e.w = 3; insertEdge(g,e);
    e.v = 3; e.w = 2; insertEdge(g,e);
    for (i = 0; i < NODES; i++) {
        if (adjacent(g, i, NODE_OF_INTEREST))
            printf("%d\n", i);}
    freeGraph(g);
    return 0; }
```

41

## Summary

- Graph terminology
  - vertices, edges, vertex degree, connected graph, tree
  - path, cycle, clique, spanning tree, spanning forest
- Graph representations
  - adjacency matrix
  - adjacency lists
- Suggested reading:
  - Sedgewick, Ch.17.1-17.5

42