Graph Basics

- Graphs
- Properties of Graphs
- Graph Terminology

❖ Graphs

Many applications require

- a collection of items (i.e. a set)
- relationships/connections between items

Examples:

- maps: items are cities, connections are roads
- web: items are pages, connections are hyperlinks

Collection types you're familiar with

- lists ... linear sequence of items (COMP1511)
- trees ... branched hierachy of items (Weeks 02/03)

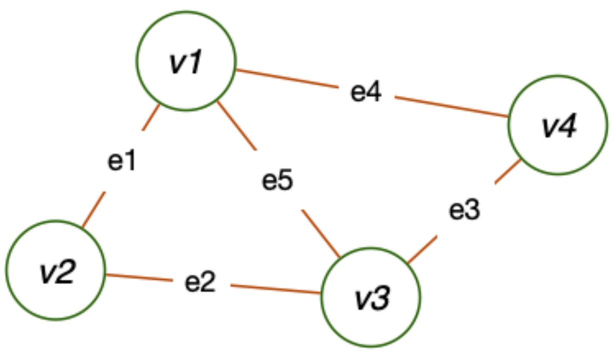
Graphs are more general ... allow arbitrary connections

❖ ... Graphs

A graph $G = (V,E)$

- V is a set of vertices 点
- E is a set of edges (subset of $V \times V$)

Example:



$V = \{ v1, v2, v3, v4 \}$

$E = \{ e1, e2, e3, e4, e5 \}$

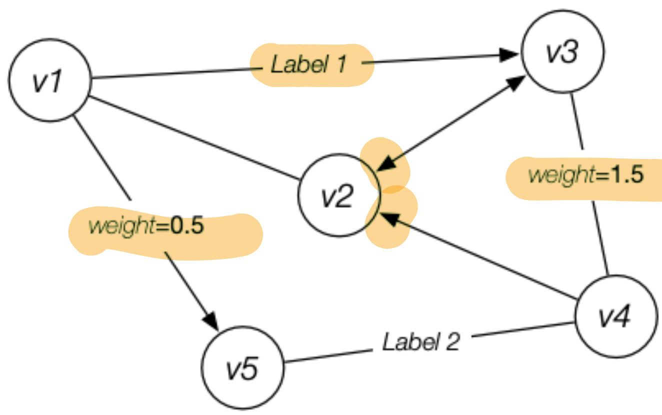
or

$E = \{ (v1,v2), (v2,v3), (v3,v4), (v1,v4), (v1,v3) \}$

❖ ... Graphs

Nodes are distinguished by a unique identifier

Edges may be (optionally) directed, labelled and/or weighted



❖ ... Graphs

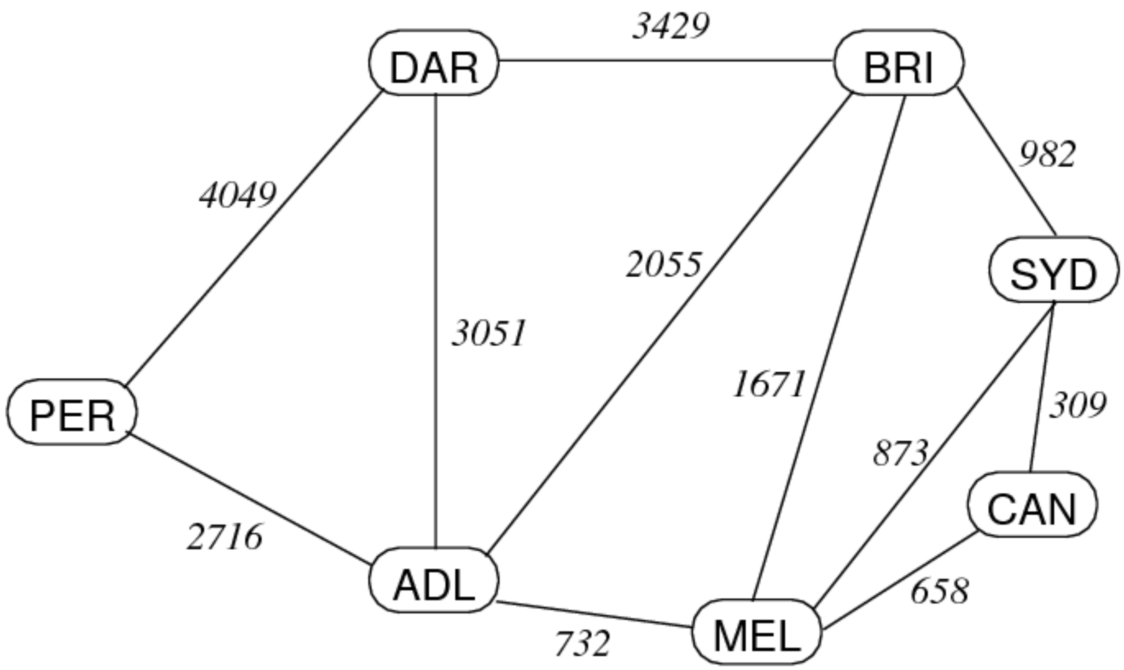
A real example: Australian road distances

Distance	Adelaide	Brisbane	Canberra	Darwin	Melbourne	Perth	Sydney
Adelaide	-	2055	1390	3051	732	2716	1605
Brisbane	2055	-	1291	3429	1671	4771	982
Canberra	1390	1291	-	4441	658	4106	309
Darwin	3051	3429	4441	-	3783	4049	4411
Melbourne	732	1671	658	3783	-	3448	873
Perth	2716	4771	4106	4049	3448	-	3972
Sydney	1605	982	309	4411	873	3972	-

Notes: vertices are cities, edges are distance between cities, symmetric

❖ ... Graphs

Alternative representation of above:



❖ ... Graphs

Questions we might ask about a graph:

- is there a way to get from item A to item B? Yes 经过其他点
- what is the best way to get from A to B?
- which items are directly connected (A↔B)?

Graph algorithms are generally more complex than tree/list ones:

- no implicit order of items
- graphs may contain cycles
- concrete representation is less obvious
- algorithm complexity depends on connection complexity

❖ Properties of Graphs

Terminology: $|V|$ and $|E|$ (cardinality) normally written just as V and E.

A graph with V vertices has at most $V(V-1)/2$ edges.

ab ba 没区别 so /2

每个点除自己都有线

The ratio $E:V$ can vary considerably.

- if E is closer to V^2, the graph is dense 密集
- if E is closer to V, the graph is sparse 稀疏
  - Example: web pages and hyperlinks

Knowing whether a graph is sparse or dense is important

- may affect choice of data structures to represent graph
- may affect choice of algorithms to process graph

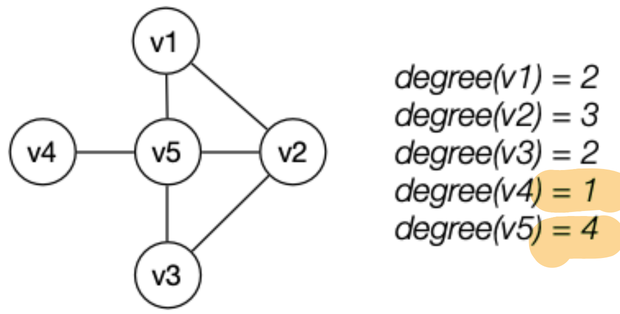
❖ Graph Terminology

For an edge *e* that connects vertices *v* and *w*

- *v* and *w* are adjacent (neighbours)
- *e* is incident on both *v* and *w*

Degree of a vertex *v*

- number of edges incident on *e*



Synonyms:

- vertex = node
- edge = arc = link (Note: some people use arc for *directed* edges)

❖ ... Graph Terminology

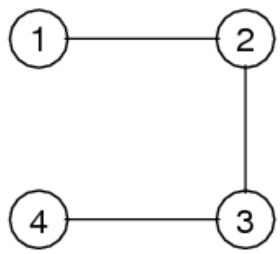
Path: a sequence of vertices where

- each vertex has an edge to its predecessor 前任

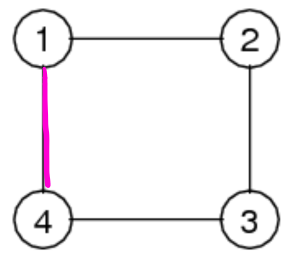
Cycle: a path where

- last vertex in path is same as first vertex in path

Length of path or cycle = #edges



Path: 1–2, 2–3, 3–4



Cycle: 1–2, 2–3, 3–4, 4–1

❖ ... Graph Terminology

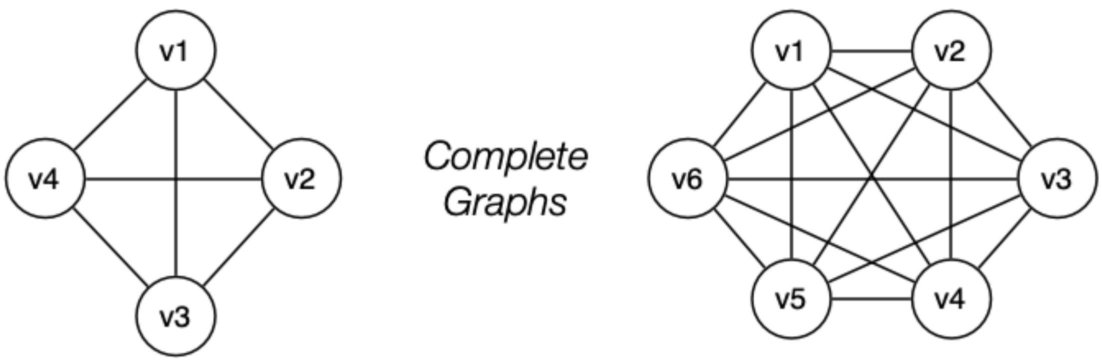
Connected graph

一个通所有

- there is a *path* from each vertex to every other vertex
- if a graph is not connected, it has ≥2 connected components

Complete graph K_V

- there is an *edge* from each vertex to every other vertex
- in a complete graph, $E = V(V-1)/2$



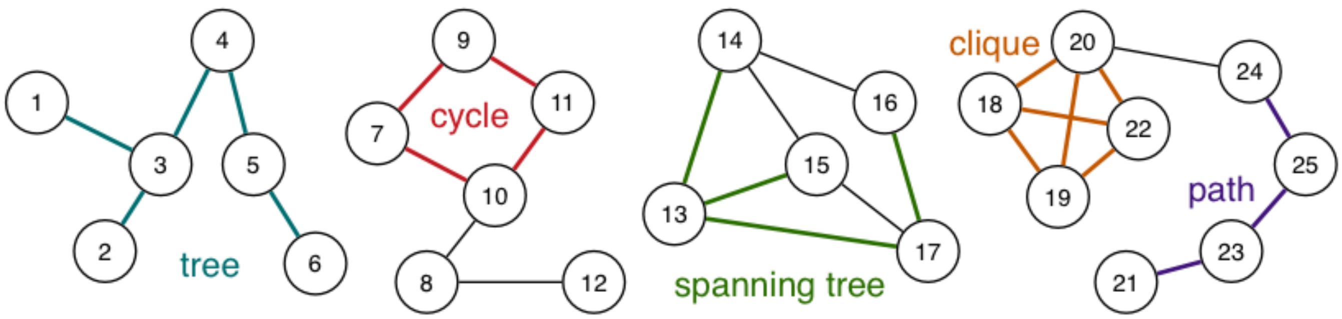
❖ ... Graph Terminology

Tree: connected (sub)graph with no cycles

Spanning tree: tree containing all vertices

Clique: complete subgraph

Consider the following *single graph*:



This graph has 25 vertices, 32 edges, and 4 connected components

Note: The entire graph has no spanning tree; what is shown in green is a spanning tree of the third connected component

❖ ... Graph Terminology

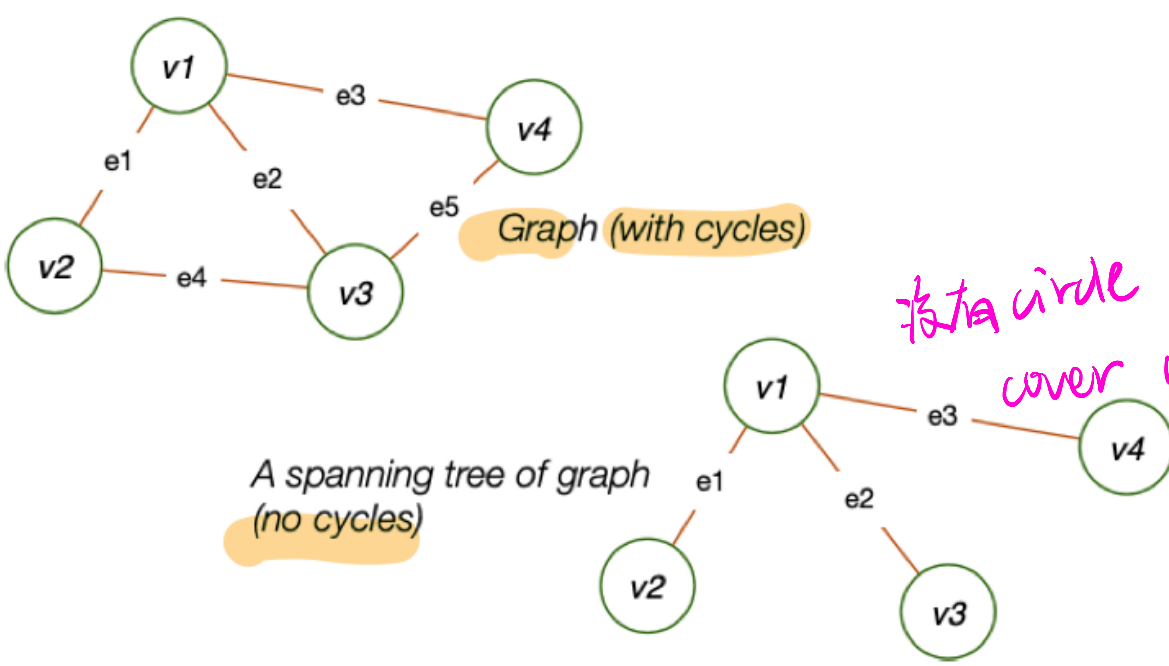
A spanning tree of connected graph $G = (V,E)$

- is a subgraph of G containing all of V
- and is a single tree (connected, no cycles)

A spanning forest of non-connected graph $G = (V,E)$

- is a subgraph of G containing all of V
- and is a set of trees (not connected, no cycles),
    - with one tree for each *connected component*

Can form spanning tree from graph by removing edges



没有 circle 并 cover all V.

Many possible spanning trees can be formed. Which is "best"?

❖ ... Graph Terminology

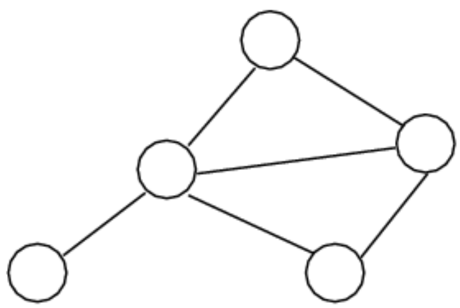
Undirected graph

- *edge(u,v) = edge(v,u)*, no self-loops (i.e. no *edge(v,v)*)

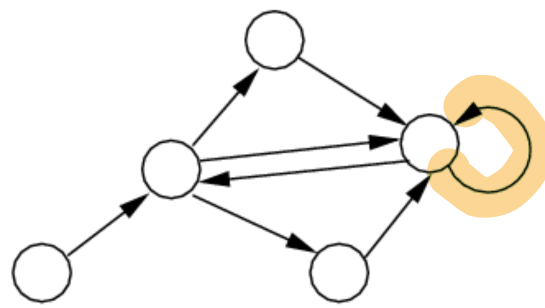
Directed graph

- *edge(u,v) ≠ edge(v,u)*, can have self-loops (i.e. *edge(v,v)*)

Examples:



Undirected graph



Directed graph

❖ ... Graph Terminology

Other types of graphs ...

Weighted graph

- each edge has an associated value (weight)
- e.g. road map (weights on edges are distances between cities)

Multi-graph

- allow multiple edges between two vertices
- e.g. function call graph (`f()` calls `g()` in several places)

Labelled graph

- edges have associated labels
- can be used to add semantic information

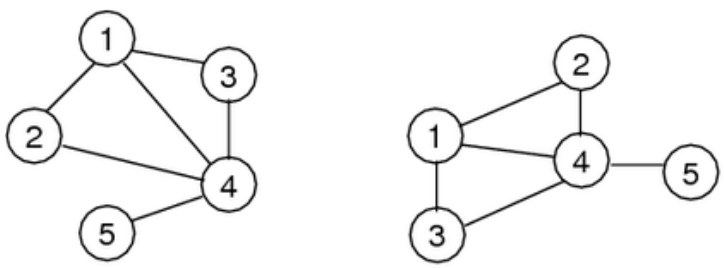
Graph Representations

❖ Graph Representations

Describing graphs:

- could describe via a diagram showing edges and vertices
- could describe by giving a list of edges
- assume we identify vertices by distinct integers

E.g. four representations of the same graph:



(c)

1–2 1–3 1–4
2–4
3–4
4–5

(d)

1–3
2–1 2–4
4–1 4–3
5–4

(a) (b) (c) (d)

❖ ... Graph Representations

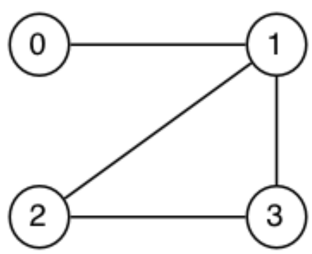
We discuss three different graph data structures:

1. Array of edges
   - explicit representation of edges as (v,w) pairs
2. Adjacency matrix
   - edges defined by presence value in VxV matrix
3. Adjacency list
   - edges defined by entries in array of V lists

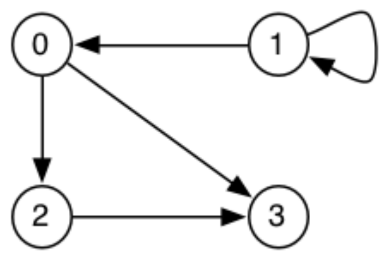
❖ Array-of-edges Representation

Edges are represented as an array of **`Edge`** values (= pairs of vertices)

- space efficient representation
- adding and deleting edges is slightly complex
- undirected: order of vertices in an **`Edge`** doesn't matter
- directed: order of vertices in an **`Edge`** encodes direction



[(0,1), (1,2), (1,3), (2,3)]



[(1,0), (1,1), (0.2), (0,3), (2,3)]

For simplicity, we always assume vertices to be numbered **`0..V-1`**

❖ ... Array-of-edges Representation

Graph initialisation

```
newGraph(V):
|  Input   number of nodes V
|  Output  new empty graph (no edges)
|  g->nV = V
|  g.nV = V    // #vertices (numbered 0..V-1)
|  g.nE = 0    // #edges
|  allocate enough memory for g.edges[]
|  return g
```

Assumes ≅ **`struct Graph { int nV; int nE; Edge edges[]; }`**

❖ ... Array-of-edges Representation

Edge insertion

```
insertEdge(g,(v,w)):
|  Input  graph g, edge (v,w)
|  Output graph g containing (v,w)
|
|  i=0     i=g.nE边数
|  while i < g.nE ∧ g.edges[i] ≠ (v,w) do
|     i=i+1        边数.
|  end while
|  if i=g.nE then         // (v,w) not found
|     g.edges[i]=(v,w)
|     g.nE=g.nE+1
|  end if
```

We "normalise" edges so that e.g (v < w) in all (v,w)

❖ ... Array-of-edges Representation

Edge removal

```
removeEdge(g,(v,w)):
|  Input   graph g, edge (v,w)
|  Output  graph g without (v,w)
|
|  i=0          i=g.nE 个边
|  while i < g.nE ∧ g.edges[i] ≠ (v,w) do
|     i=i+1
|  end while
|  if i < g.nE then   // (v,w) found
|     g.edges[i]=g.edges[g.nE-1]
|           // replace by last edge in array
|     g.nE=g.nE-1
|  end if
```

❖ ... Array-of-edges Representation

Print a list of edges

```
showEdges(g):
|   Input graph g
|
|   for all i=0 to g.nE-1 do
|   |   (v,w)=g.edges[i]
|   |   print v"—"w
|   end for
```

❖ Array-of-edges Cost Analysis

Storage cost: *O(E)*

Cost of operations:

- initialisation: *O(1)*
- insert edge: *O(E)* (need to check for all edge in array)
- delete edge: *O(E)* (need to find edge in edge array)

If array is full on insert

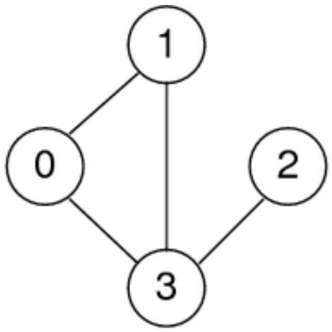
- allocate space for a bigger array, realloc copy 复制 edges across ⇒ still *O(E)*

If we maintain edges in order

- use binary search to find edge ⇒ *O(log E)*

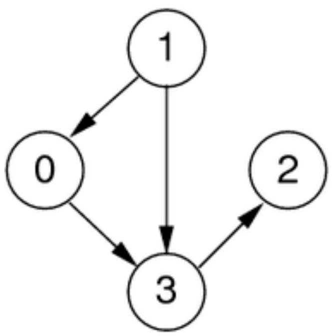
❖ Adjacency Matrix Representation

Edges represented by a $V \times V$ 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

❖ ... Adjacency Matrix Representation

Advantages

- easily implemented as 2-dimensional array
- can represent graphs, digraphs and weighted graphs
  - graphs: symmetric boolean matrix
  - digraphs: non-symmetric boolean matrix
  - weighted: non-symmetric matrix of weight values

Disadvantages:

- if few edges (sparse) ⇒ memory-inefficient   ($O(V^2)$ space)

内存占用太多

❖ ... Adjacency Matrix Representation

Graph initialisation

```
newGraph(V):
|   Input  number of nodes V
|   Output new empty graph
|
|   g.nV = V    // #vertices (numbered 0..V-1)
|   g.nE = 0    // #edges
|   allocate memory for g.edges[][]
|   for all i,j=0..V-1 do
|      g.edges[i][j]=0     // false
|   end for
|   return g
```

❖ ... Adjacency Matrix Representation

Edge insertion

```
insertEdge(g,(v,w)):
|  Input  graph g, edge (v,w)
|  Output graph g containing (v,w)
|
|  if g.edges[v][w] = 0 then   // (v,w) not in graph
|     g.edges[v][w]=1     // set to true
|     g.edges[w][v]=1
|     g.nE=g.nE+1
|  end if
```

❖ ... Adjacency Matrix Representation

Edge removal

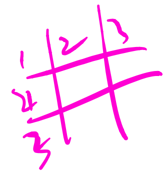
```
removeEdge(g,(v,w)):
|  Input  graph g, edge (v,w)
|  Output graph g without (v,w)
|
|  if g.edges[v][w] ≠ 0 then   // (v,w) in graph
|     g.edges[v][w]=0         // set to false
|     g.edges[w][v]=0
|     g.nE=g.nE-1
|  end if
```

❖ ... Adjacency Matrix Representation

Print a list of edges

```
showEdges(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 then
|   |           print i"—"j
|   |       end if
|   |   end for
|   end for
```

❖ Adjacency Matrix Cost Analysis



3×3

Storage cost: $O(V^2)$

If the graph is sparse, most storage is wasted.

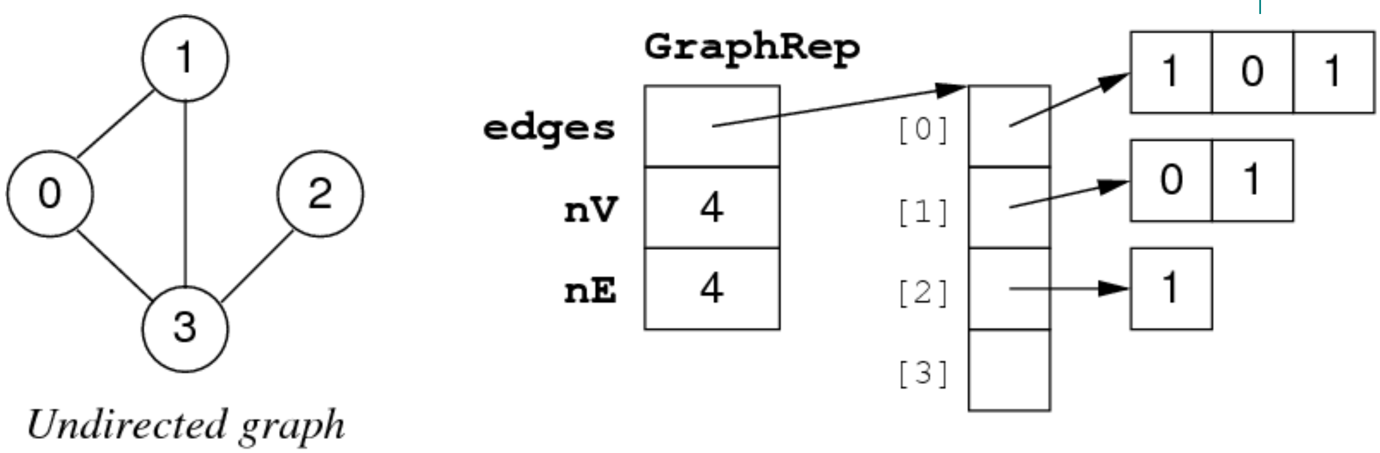
稀疏

Cost of operations:

- initialisation: $O(V^2)$ (initialise $V \times V$ matrix)
- insert edge: $O(1)$ (set two cells in matrix)
- delete edge: $O(1)$ (unset two cells in matrix)

❖ ... Adjacency Matrix Cost Analysis

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

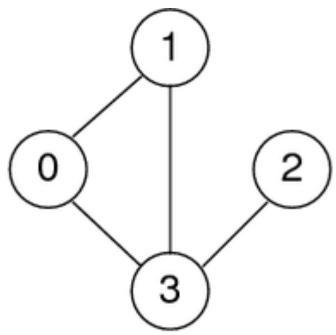


New storage cost: *V-1* int ptrs + *V(V+1)/2* ints (but still $O(V^2)$)

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

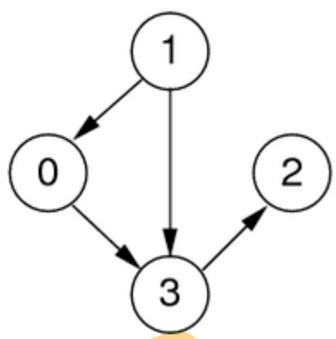
❖ Adjacency List Representation

For each vertex, store linked list of adjacent vertices:



Undirected graph

A[0] = <1, 3>
A[1] = <0, 3>
A[2] = <3>
A[3] = <0, 1, 2>



Directed graph

A[0] = <3>
A[1] = <0, 3>
A[2] = < >
A[3] = <2>

❖ ... Adjacency List Representation

Advantages

- relatively easy to implement in languages like C
- can represent graphs and digraphs
- memory efficient if *E:V* relatively small

Disadvantages:

- one graph has many possible representations
  (unless lists are ordered by same criterion e.g. ascending)

❖ ... Adjacency List Representation

Graph initialisation

```
newGraph(V):
|  Input  number of nodes V
|  Output new empty graph
|
|  g.nV = V    // #vertices (numbered 0..V-1)
|  g.nE = 0    // #edges
|  allocate memory for g.edges[]
|  for all i=0..V-1 do
|     g.edges[i]=newList()   // empty list
|  end for
|  return g
```

❖ ... Adjacency List Representation

Edge insertion:

```
insertEdge(g,(v,w)):
|  Input  graph g, edge (v,w)
|  Output graph g containing (v,w)
|
|  if not ListMember(g.edges[v],w) then
|     // (v,w) not in graph
|     ListInsert(g.edges[v],w)
|     ListInsert(g.edges[w],v)
|     g.nE=g.nE+1
|  end if
```

❖ ... Adjacency List Representation

Edge removal:

```
removeEdge(g,(v,w)):
|  Input  graph g, edge (v,w)
|  Output graph g without (v,w)
|
|  if ListMember(g.edges[v],w) then
|     // (v,w) in graph
|     ListDelete(g.edges[v],w)
|     ListDelete(g.edges[w],v)
|     g.nE=g.nE-1
|  end if
```

❖ ... Adjacency List Representation

Print a list of edges

```
showEdges(g):
|   Input graph g
|
|   for all i=0 to g.nV-1 do
|   |   for all v in g.edges[i] do
|   |       if i < v then
|   |           print i"—"v
|   |       end if
|   |   end for
|   end for
```

❖ Adjacency List Cost Analysis

Storage cost: *O(V+E)*

Cost of operations:

- initialisation: *O(V)* (initialise *V* lists)
- insert edge: *O(E)* (need to check if vertex in list)
- delete edge: *O(E)* (need to find vertex in list)

Could sort vertex lists, but no benefit (although no extra cost)

❖ Comparison of Graph Representations

Summary of operations above:

	array of edges	adjacency matrix	adjacency list
Storage space usage	E	V^2	$V+E$
initialise	1	V^2	V
insert edge	E	1	E
remove edge	E	1	E

Other operations:

	array of edges	adjacency matrix	adjacency list
disconnected(v)?	E	V	1
isPath(x,y)?	$E \cdot \log V$	V^2	$V+E$
copy graph	E	V^2	$V+E$
destroy graph	1	V	$V+E$

Graph ADT

❖ Graph ADT

Data: set of edges, set of vertices

Operations:

- building: create graph, add edge
- deleting: remove edge, drop whole graph
- scanning: check if graph contains a given edge

Things to note:

- set of vertices is fixed when graph initialised
- we treat vertices as `int`s, but could be arbitrary `Item`s

Will use this ADT as a basis for building more complex operations later.

❖ ... Graph ADT

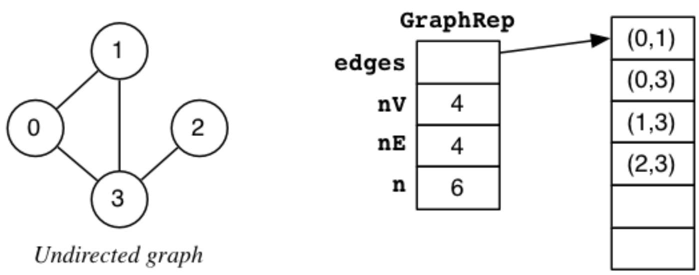
Graph ADT interface **`Graph.h`**

```
// graph representation is hidden
typedef struct GraphRep *Graph;

// vertices denoted by integers 0..N-1
typedef int Vertex;

// edges are pairs of vertices (end-points)
typedef struct Edge { Vertex v; Vertex w; } Edge;

// operations on graphs
Graph newGraph(int V);  // new graph with V vertices
void  insertEdge(Graph, Edge);
void  removeEdge(Graph, Edge);
bool  adjacent(Graph, Vertex, Vertex);
      // is there an edge between two vertices?
void  freeGraph(Graph);
```

❖ Graph ADT (Array of Edges)

Implementation of **`GraphRep`** (array-of-edges representation)

```
typedef struct GraphRep {
   Edge *edges; // array of edges
   int   nV;    // #vertices (numbered 0..nV-1)
   int   nE;    // #edges
   int   n;     // size of edge array
} GraphRep;
```



❖ ... Graph ADT (Array of Edges)

Implementation of graph initialisation (array-of-edges)

```
Graph newGraph(int V) {
   assert(V >= 0);
   Graph g = malloc(sizeof(GraphRep));
   assert(g != NULL);
   g->nV = V; g->nE = 0;
   // allocate enough memory for edges
   g->n = Enough;
   g->edges = malloc(g->n*sizeof(Edge));
   assert(g->edges != NULL);
   return g;
}
```

How much is enough? ... No more than *V(V-1)/2* ... Much less in practice (sparse graph)

$\frac{4\times3}{2}=2\times3=6$

❖ ... Graph ADT (Array of Edges)

Some useful utility functions:

```
// check if two edges are equal
bool eq(Edge e1, Edge e2) {
   return ( (e1.v == e2.v && e1.w == e2.w)
            || (e1.v == e2.w && e1.w == e2.v) );
}

// check if vertex is valid in a graph
bool validV(Graph g, Vertex v) {
   return (g != NULL && v >= 0 && v < g->nV);
}

// check if an edge is valid in a graph
bool validE(Graph g, Edge e) {
   return (g != NULL && validV(e.v) && validV(e.w));
}
```

❖ ... Graph ADT (Array of Edges)

Implementation of edge insertion (array-of-edges)

```
void insertEdge(Graph g, Edge e) {
   // ensure that g exists and array of edges isn't full
   assert(g != NULL && g->nE < g->n && isValidE(g,e));
   int i = 0;  // can't define in for (...)
   for (i = 0; i < g->nE; i++)
       if (eq(e,g->edges[i])) break;
   if (i == g->nE)  // edge e not found
      g->edges[g->nE++] = e;
}
```

已经有了

number of edges

❖ ... Graph ADT (Array of Edges)

Implementation of edge removal (array-of-edges)

```
void removeEdge(Graph g, Edge e) {
    // ensure that g exists
    assert(g != NULL && validE(g,e));
    int i = 0;
    while (i < g->nE && !eq(e,g->edges[i]))
       i++;
    if (i < g->nE)   // edge e found
       g->edges[i] = g->edges[--g->nE];
}
```

❖ ... Graph ADT (Array of Edges)

Implementation of edge check (array-of-edges)

```
bool adjacent(Graph g, Vertex x, Vertex y) {
   assert(g != NULL && validV(g,x) && validV(g,y));
   Edge e;
   e.v = x; e.w = y;
   for (int i = 0; i < g->nE; i++) {
      if (eq(e,g->edges[i]))  // edge found
         return true;
   }
   return false;  // edge not found
}
```

❖ ... Graph ADT (Array of Edges)

Re-implementation of edge insertion (array-of-edges)

```
void insertEdge(Graph g, Edge e) {
   // ensure that g exists
   assert(g != NULL && validE(g,e));
   int i = 0;
   for (i = 0; i < g->nE; i++)
       if (eq(e,g->edges[i])) break; 已有
   if (i == g->nE) { // edge e not found
      if (g->n == g->nE) {  // array full; expand
         g->edges = realloc(g->edges, 2*g->n);
         assert(g->edges != NULL);
         g->n = 2*g->n;
      }
      g->edges[g->nE++] = e;
   }
}
```

❖ ... Graph ADT (Array of Edges)

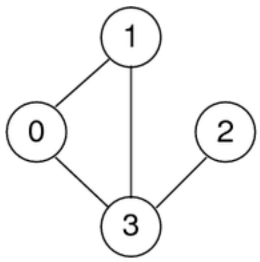
Implementation of graph removal (array-of-edges)

```
void freeGraph(Graph g) {
   assert(g != NULL);
   free(g->edges);   // free array of edges
   free(g);          // remove Graph object
}
```

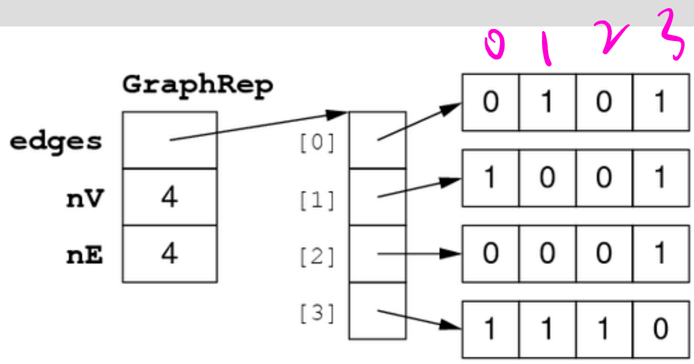
❖ Graph ADT (Adjacency Matrix)

Implementation of **`GraphRep`** (adjacency-matrix representation)

```
typedef struct GraphRep {
   int  **edges; // adjacency matrix
   int    nV;    // #vertices
   int    nE;    // #edges
} GraphRep;
```

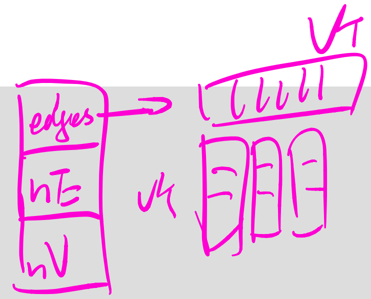


Undirected graph



❖ ... Graph ADT (Adjacency Matrix)

Implementation of graph initialisation (adjacency-matrix)




```
Graph newGraph(int V) {
   assert(V >= 0);
   Graph g = malloc(sizeof(GraphRep));
   assert(g != NULL);
   g->nV = V;  g->nE = 0;
   // allocate array of pointers to rows
   g->edges = malloc(V * sizeof(int *));
   assert(g->edges != NULL);
   // allocate memory for each column and initialise with 0
   for (int i = 0; i < V; i++) {
      g->edges[i] = calloc(V, sizeof(int));
      assert(g->edges[i] != NULL);
   }
   return g;
}
```

Standard library function **`calloc(size_t nelems, size_t nbytes)`**

- allocates a memory block of size **`nelems*nbytes`**
- and sets all bytes in that block to zero

❖ ... Graph ADT (Adjacency Matrix)

Implementation of edge insertion (adjacency-matrix)

```
void insertEdge(Graph g, Edge e) {
   assert(g != NULL && validE(g,e));

   if (!g->edges[e.v][e.w]) {  // edge e not in graph
      g->edges[e.v][e.w] = 1;
      g->edges[e.w][e.v] = 1;
      g->nE++;
   }
}
```

❖ ... Graph ADT (Adjacency Matrix)

Implementation of edge removal (adjacency-matrix)

```
void removeEdge(Graph g, Edge e) {
   assert(g != NULL && validE(g,e));

   if (g->edges[e.v][e.w]) {    // edge e in graph
      g->edges[e.v][e.w] = 0;
      g->edges[e.w][e.v] = 0;
      g->nE--;
   }
}
```

❖ ... Graph ADT (Adjacency Matrix)

Implementation of edge check (adjacency matrix)

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

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

两个

除了

Note: all operations, except creation, are *O(1)*

❖ ... Graph ADT (Adjacency Matrix)

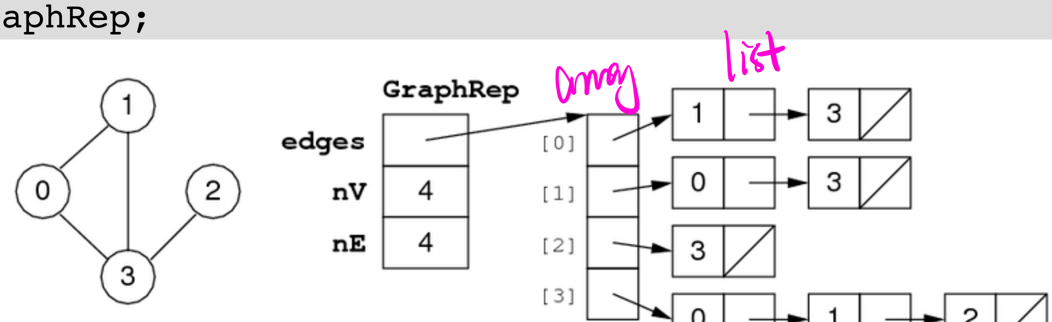
Implementation of graph removal (adjacency matrix)

```
void freeGraph(Graph g) {
   assert(g != NULL);
   for (int i = 0; i < g->nV; i++)
      // free one row of matrix
      free(g->edges[i]);
   free(g->edges);   // free array of row pointers
   free(g);          // remove Graph object
}
```

❖ Graph ADT (Adjacency Lists)

Implementation of **`GraphRep`** (adjacency-lists representation)

```
typedef struct GraphRep {
   Node **edges;   // array of lists
   int    nV;      // #vertices
   int    nE;      // #edges
} GraphRep;
```



Undirected graph

❖ ... Graph ADT (Adjacency Lists)

Assume that we have a linked list implementation

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

with operations like **inLL**, **insertLL**, **deleteLL**, **freeLL**, e.g.

```
bool inLL(Node *L, Vertex v) {
   while (L != NULL) {
      if (L->v == v) return true;
      L = L->next;
   }
   return false;
}
```

check v 是否在 list

❖ ... Graph ADT (Adjacency Lists)

Implementation of graph initialisation (adjacency lists)

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

array

→ Null

→ Null

❖ ... Graph ADT (Adjacency Lists)

Implementation of edge insertion/removal (adjacency lists)

```
void insertEdge(Graph g, Edge e) {
   assert(g != NULL && validE(g,e));
   if (!inLL(g->edges[e.v], e.w)) {  // edge e not in graph
      g->edges[e.v] = insertLL(g->edges[e.v], e.w);
      g->edges[e.w] = insertLL(g->edges[e.w], e.v);
      g->nE++;
   }
}
void removeEdge(Graph g, Edge e) {
   assert(g != NULL && validE(g,e));
   if (inLL(g->edges[e.v], e.w)) {  // edge e in graph
      g->edges[e.v] = deleteLL(g->edges[e.v], e.w);
      g->edges[e.w] = deleteLL(g->edges[e.w], e.v);
      g->nE--;
   }
}
```

❖ ... Graph ADT (Adjacency Lists)

Implementation of edge check (adjacency lists)

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

   return inLL(g->edges[x], y);
}
```

Note: all operations, except creation, are *O(E)*

edges

❖ ... Graph ADT (Adjacency Lists)

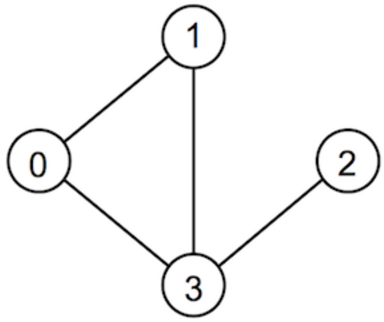
Implementation of graph removal (adjacency lists)

```
void freeGraph(Graph g) {
   assert(g != NULL);
   for (int i = 0; i < g->nV; i++)
      freeLL(g->edges[i]); // free one list
   free(g->edges);   // free array of list pointers
   free(g);          // remove Graph object
}
```

❖ Example: Graph ADT Client

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



❖ ... Example: Graph ADT Client

```
#include <stdio.h>
#include "Graph.h"

#define NODES 4
#define NODE_OF_INTEREST 1

int main(void) {
   Graph g = newGraph(NODES);
   Edge e;

   while (scanf("%d %d", &(e.v), &(e.w)) == 2)
      insertEdge(g,e);

   for (Vertex v = 0; v < NODES; v++) {
      if (adjacent(g, v, NODE_OF_INTEREST))
         printf("%d\n", v);
   }

   freeGraph(g);
   return 0;
}
```