>>

Graph Representations

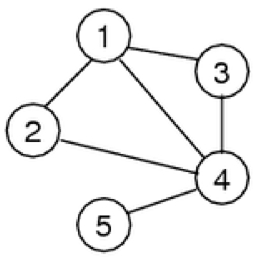
^ >>

❖ Graph Representations

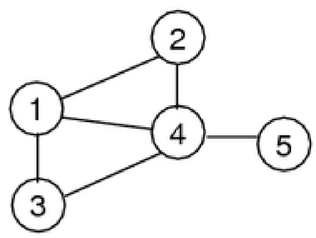
Describing graphs:

- 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:



(a)



(b)

1–2 1–3 1–4
2–4
3–4
4–5

(c)

1–3
2–1 2–4
4–1 4–3
5–4

(d)

❖ ... Graph Representations

We discuss three different graph data structures:

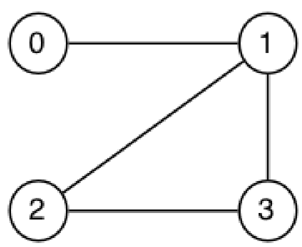
1. Array of edges
   - explicit representation of edges as (v,w) pairs
2. Adjacency matrix
   - edges defined by presence value in VxV matrix
3. Adjacency list
   - edges defined by entries in array of V lists

<< ∧ >>

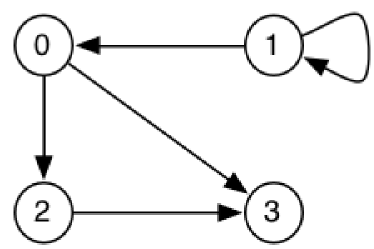
❖ Array-of-edges Representation

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   // #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
|  while i < g.nE ∧ g.edges[i] ≠ (v,w) do
|     i=i+1
|  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
|  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 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, copy edges across ⇒ still *O(E)*

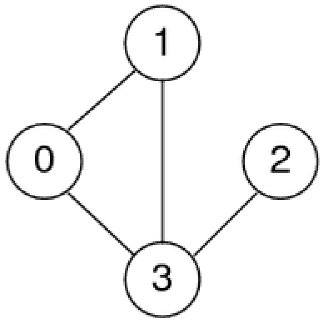
If we maintain edges in order

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

<< ∧ >>

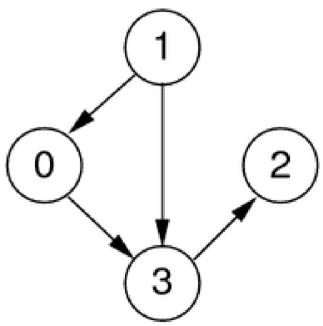
❖ Adjacency Matrix Representation

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)

❖ ... 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

Storage cost: $O(V^2)$

If the graph is sparse, most storage is wasted.

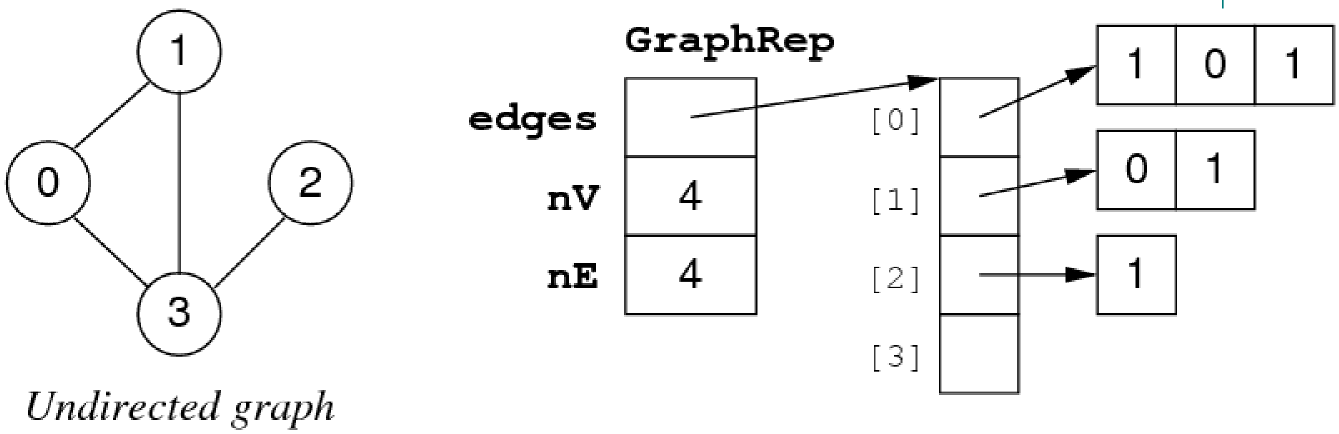
Cost of operations:

- 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.

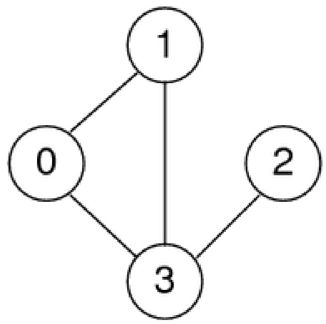


New storage cost: *V-1* int ptrs + *V(V+1)/2* ints (but still $O(V^2)$)

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

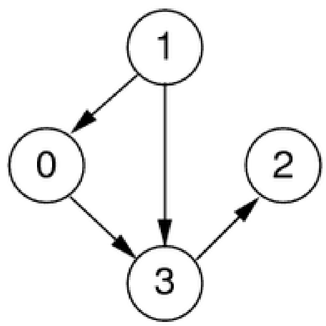
❖ Adjacency List Representation

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
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	$V+E$

Produced: 22 Jun 2020