Graphs

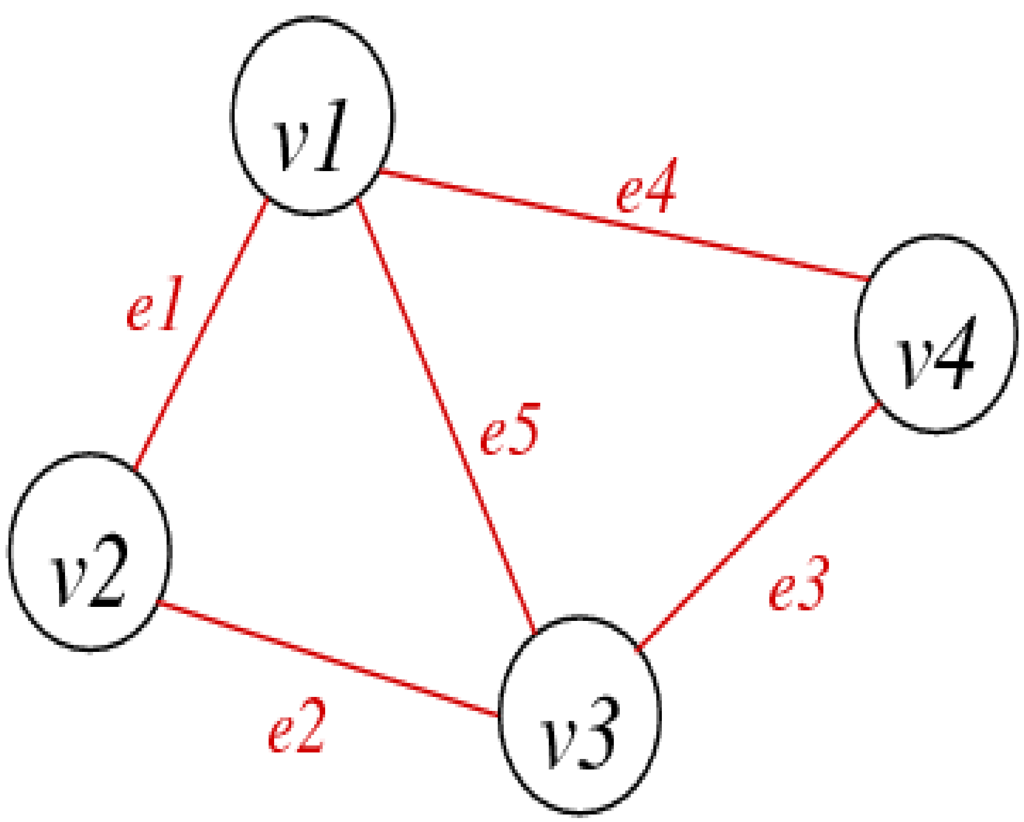
Computing 2 COMP1927 16x1
Sedgewick Part 5: Chapter 17

What are 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 we've seen so far
  - Lists…linear sequence of items
  - trees ... branched hierarchy of items
  - These are both special cases of graphs.
- Graphs are more general ... allow arbitrary connections.

DEFINITION OF A GRAPH

- A graph $G = (V,E)$
  - V is a set of vertices
  - E is a set of edges (subset of $V \times V$)
- Example:



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

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

OTHER GRAPH APPLICATION EXAMPLES

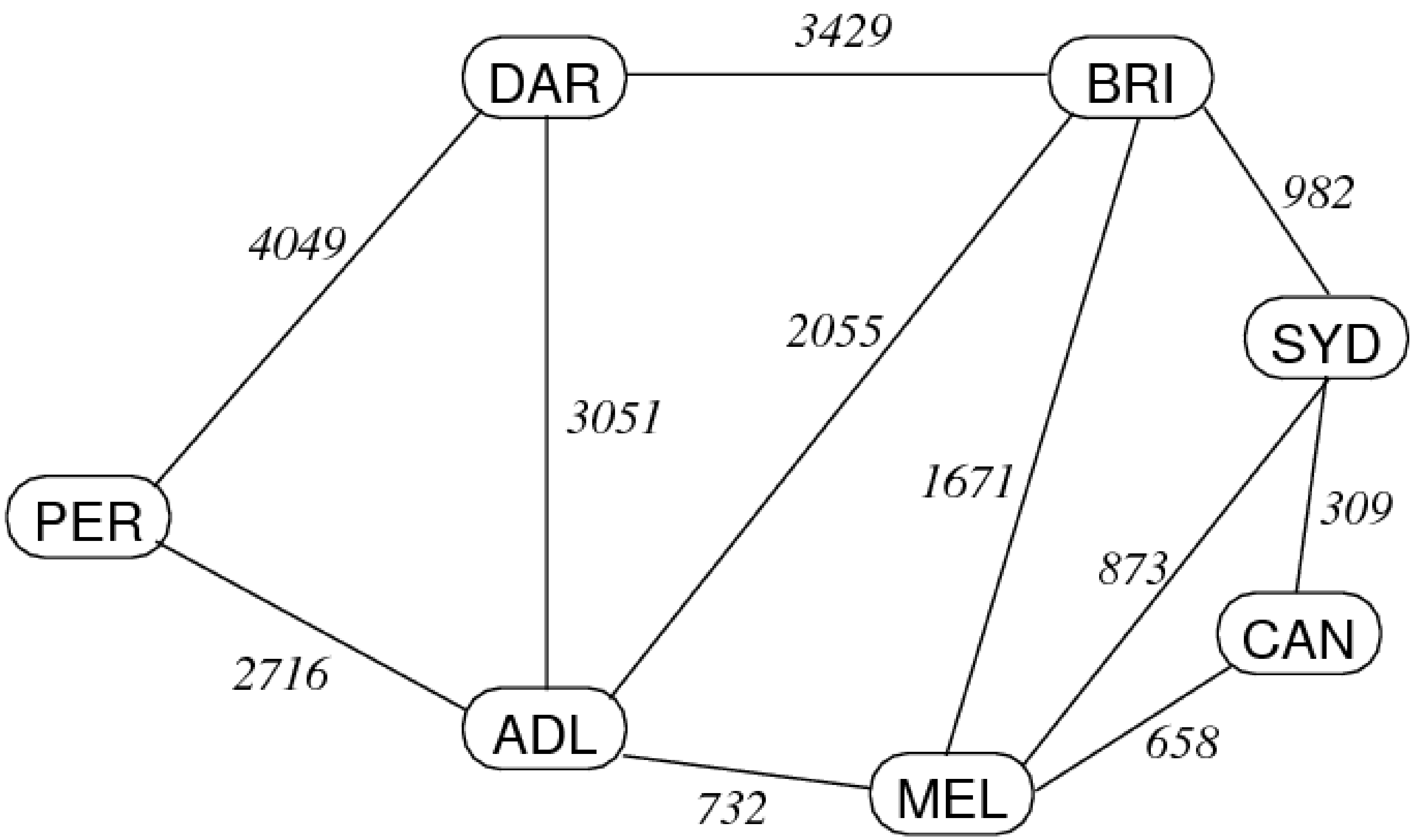
Graph	Vertices	Edges
Communication	Telephones, Computers	cables
Games	Board positions	Legal moves
Social networks	People	Friendships
Scheduling	Tasks	Precedence Constraints
Circuits	Gates,Registers, Processors	Wires
Transport	Intersections/ airports	Roads,flights

A real example: Australian road distances

Dist	Adel	Bris	Can	Dar	Melb	Perth	Syd
Adel	-	2055	1390	3051	732	2716	1605
Bris	2055	-	1291	3429	1671	4771	982
Can	1390	1291	-	4441	658	4106	309
Dar	3051	3429	4441	-	3783	4049	4411
Melb	732	1671	658	3783	-	3448	873
Perth	2716	4771	4106	4049	3448	-	3972
Syd	1605	982	309	4411	873	3972	-

A REAL GRAPH EXAMPLE

- Alternative representation of Australian roads:

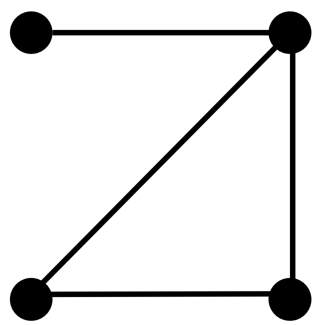


GRAPHS

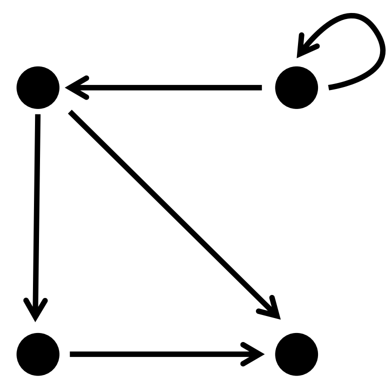
- Questions we might ask about a graph
  - is there a way to get from item A to item B?
  - what's the best way?
  - which items are connected?
- Graph algorithms are in general significantly more difficult than list or tree processing
  - no implicit order of the items
  - graphs can contain cycles
  - concrete representation is less obvious
  - complexity of algorithms depend connection complexity

SIMPLE GRAPHS

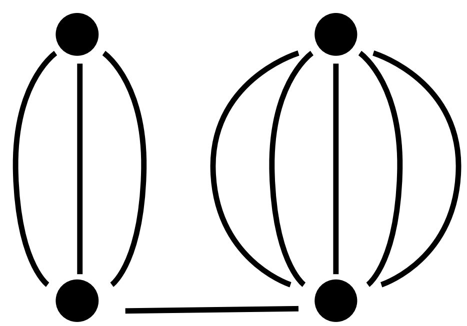
Depending on the application, graphs can have different properties:



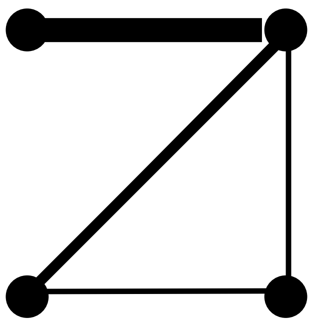
undirected



directed



multigraph



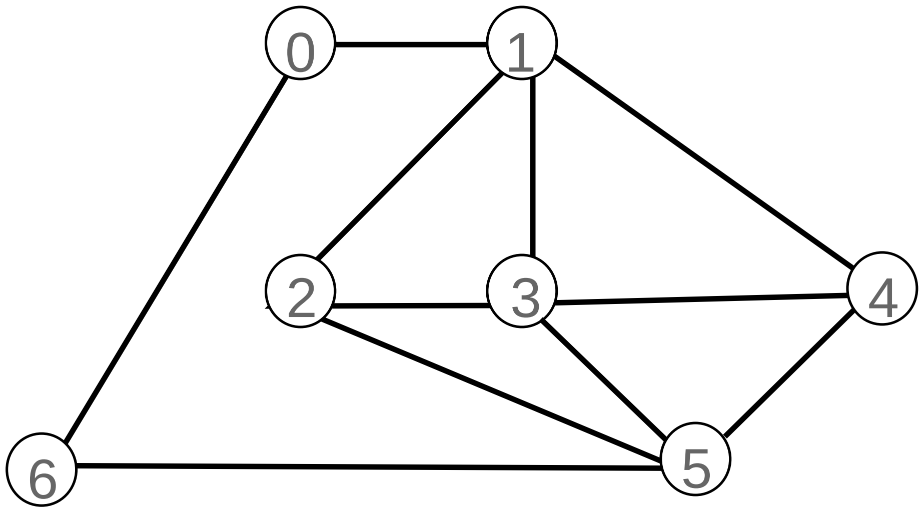
weighted

At this point, we will only consider **simple graphs** which are characterised by:

- a set of vertices, and
- a set of undirected edges that connect pairs of vertices
  - no self loops
  - no parallel edges

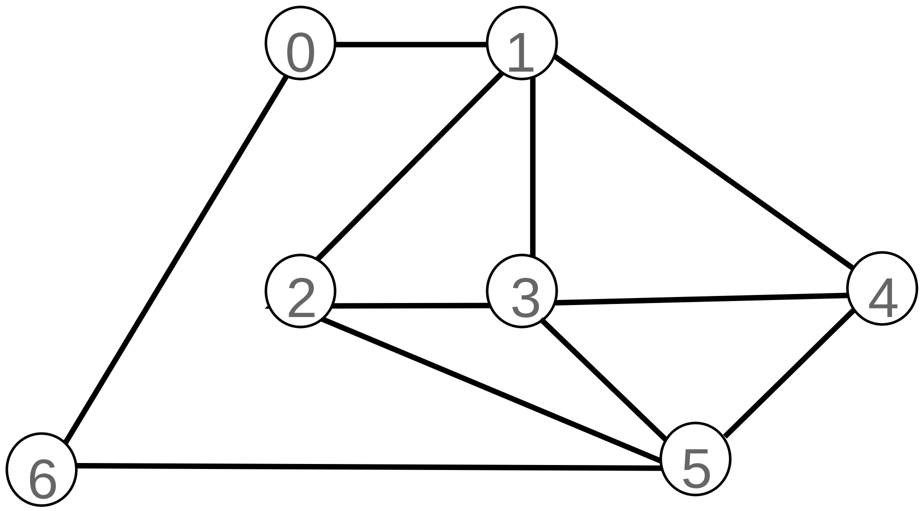
Simple Graph: Vertices and Edges

- *In our example graph:*
  - *V* (number of vertices): *7*
    - *From 0* to *6*
    - A *7*-vertex graph
  - *E* (number of edges): *11*
- How many edges can a *7*-vertex simple graph have?
  - $7*(7-1)/2 = 21$



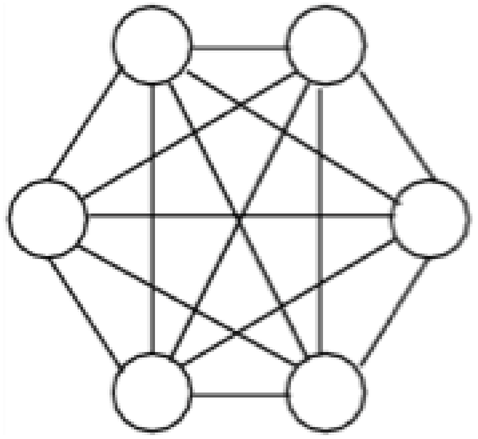
Simple Graph: Vertices and Edges

- $E <= V*(V-1)/2$
  - If E is closer to V^2 the graph is dense
  - If E is closer to V the graph is sparse
  - If E is 0 we have a set
- These properties may affect
  - choice of data structures to represent the graph and
  - the algorithms used



GRAPHS: TERMINOLOGY

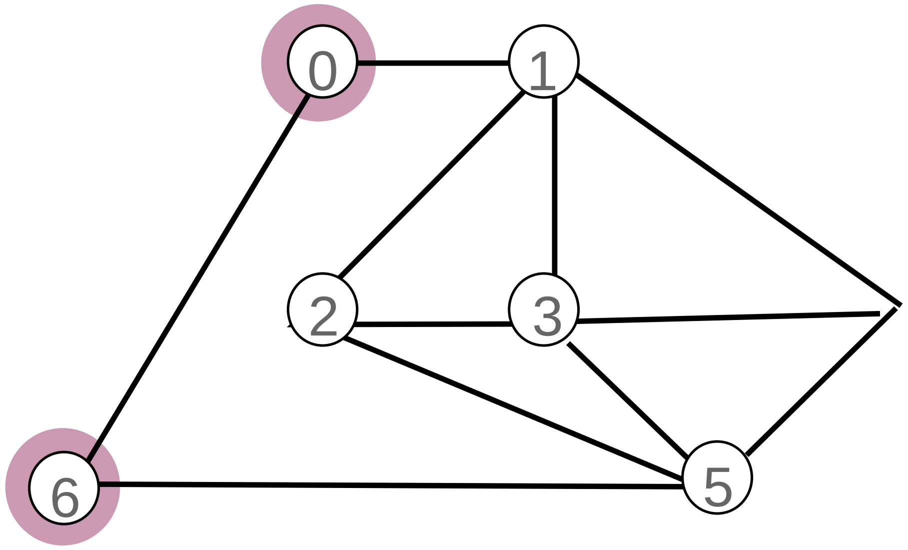
- The degree of a vertex is the number of edges from the vertex
- A complete graph is a graph where every vertex is connected to all the other vertices
  - E = V(V-1)/2
  - The degree of every vertex is
    - V-1



Complete Graph

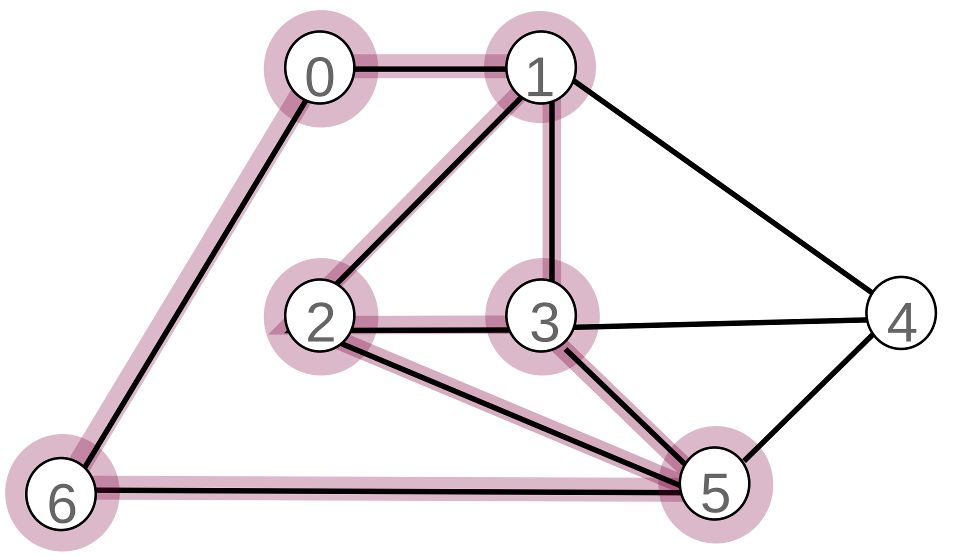
GRAPH TERMINOLOGY

- adjacent: two vertices, v and w are adjacent if there is an edge, e, between them
- e is incident on both v and w



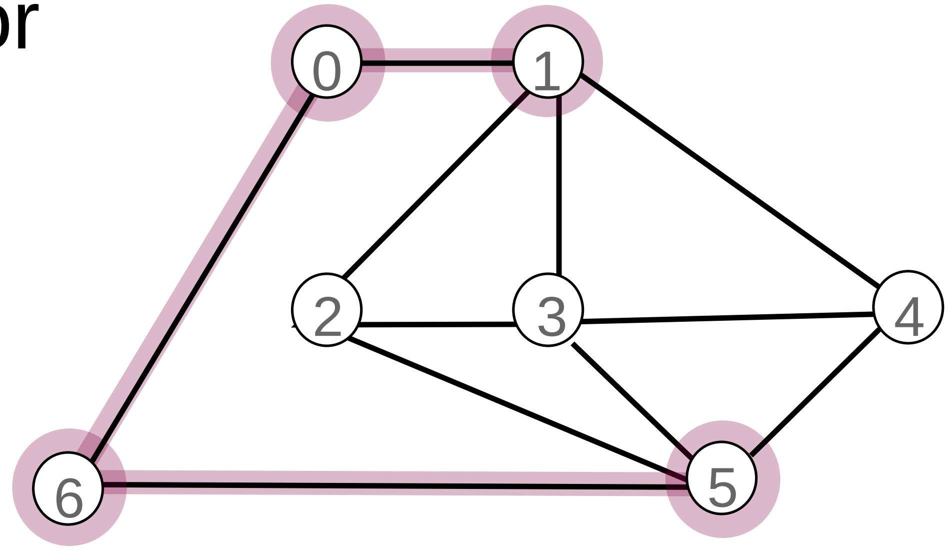
GRAPH TERMINOLOGY

subgraph: a subset of vertices with their associated edges



GRAPH TERMINOLOGY: PATHS

a path: a sequence of vertices where each one is connected to its predecessor
1,0,6,5



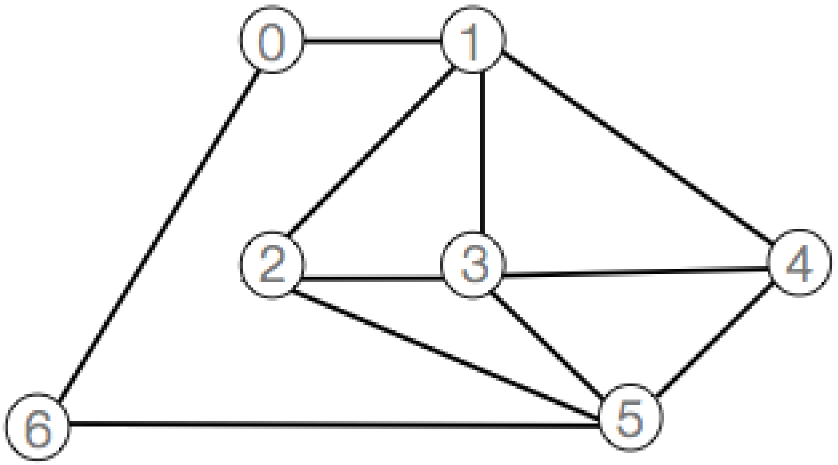
a graph is a tree if there is exactly one path between each pair of vertices

a path is simple if it doesn't have any repeating vertices

a path is a cycle if it is simple apart from its first and last vertex

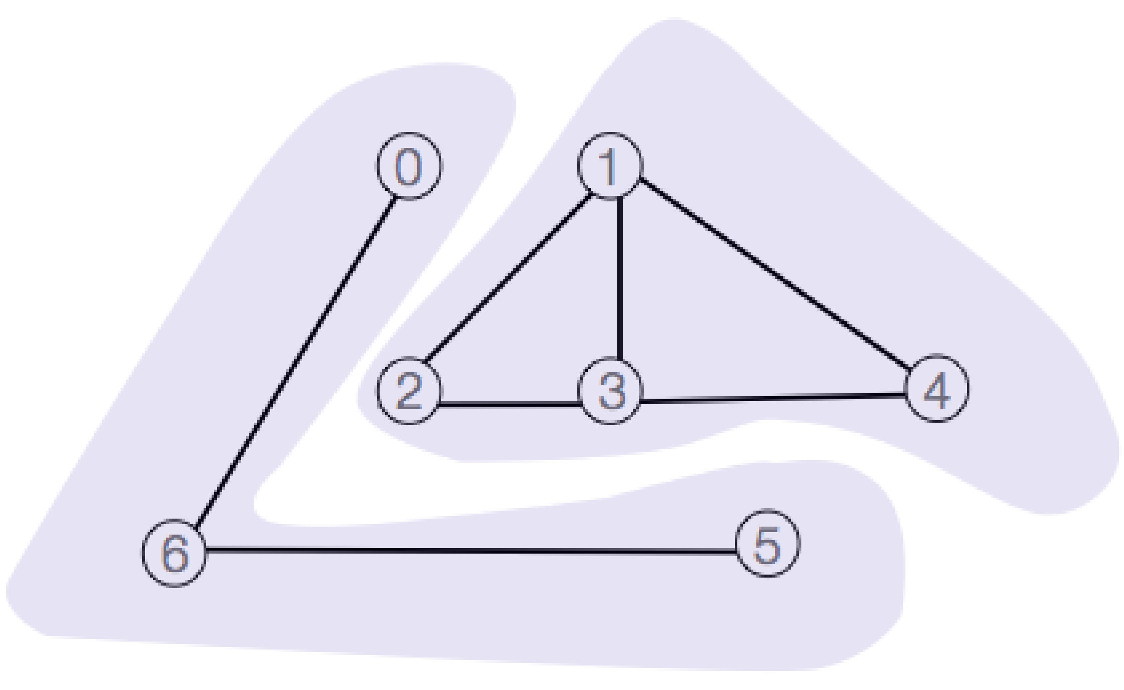
GRAPH TERMINOLOGY

- A graph is a connected graph, if there is a path from every vertex to every other vertex in the graph



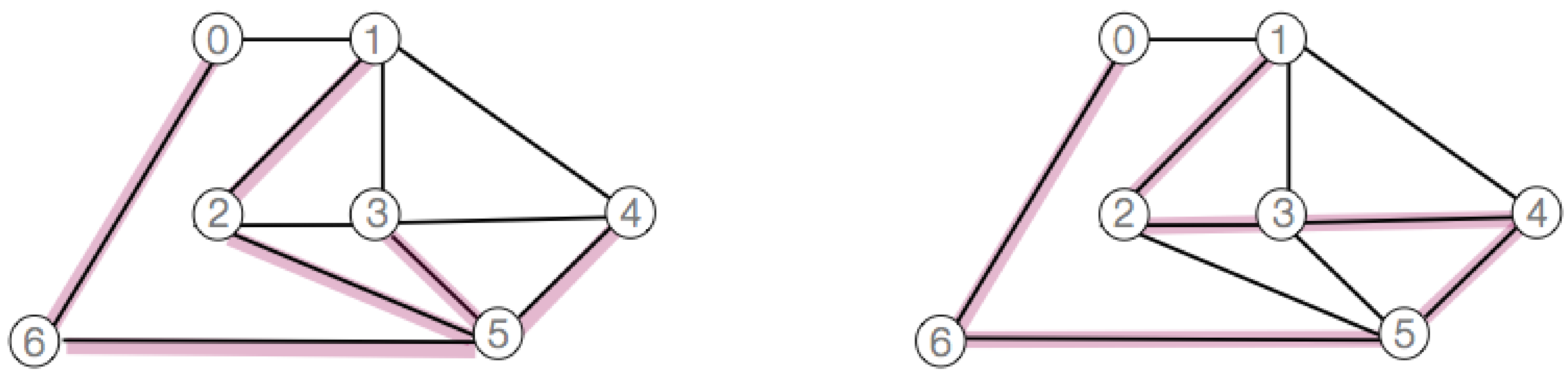
GRAPH TERMINOLOGY

- A graph that is not connected consists of a set of connected components, which are maximally connected subgraphs



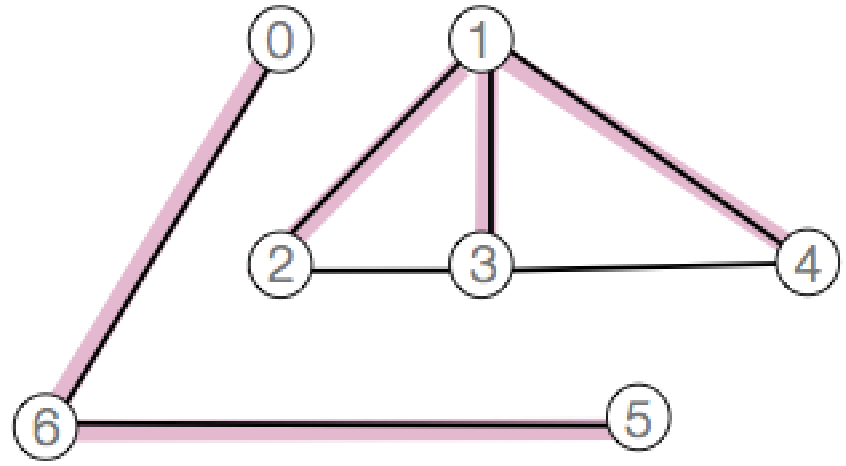
Graph Terminology

- A spanning tree of a graph is a subgraph that contains all the vertices and is a single tree



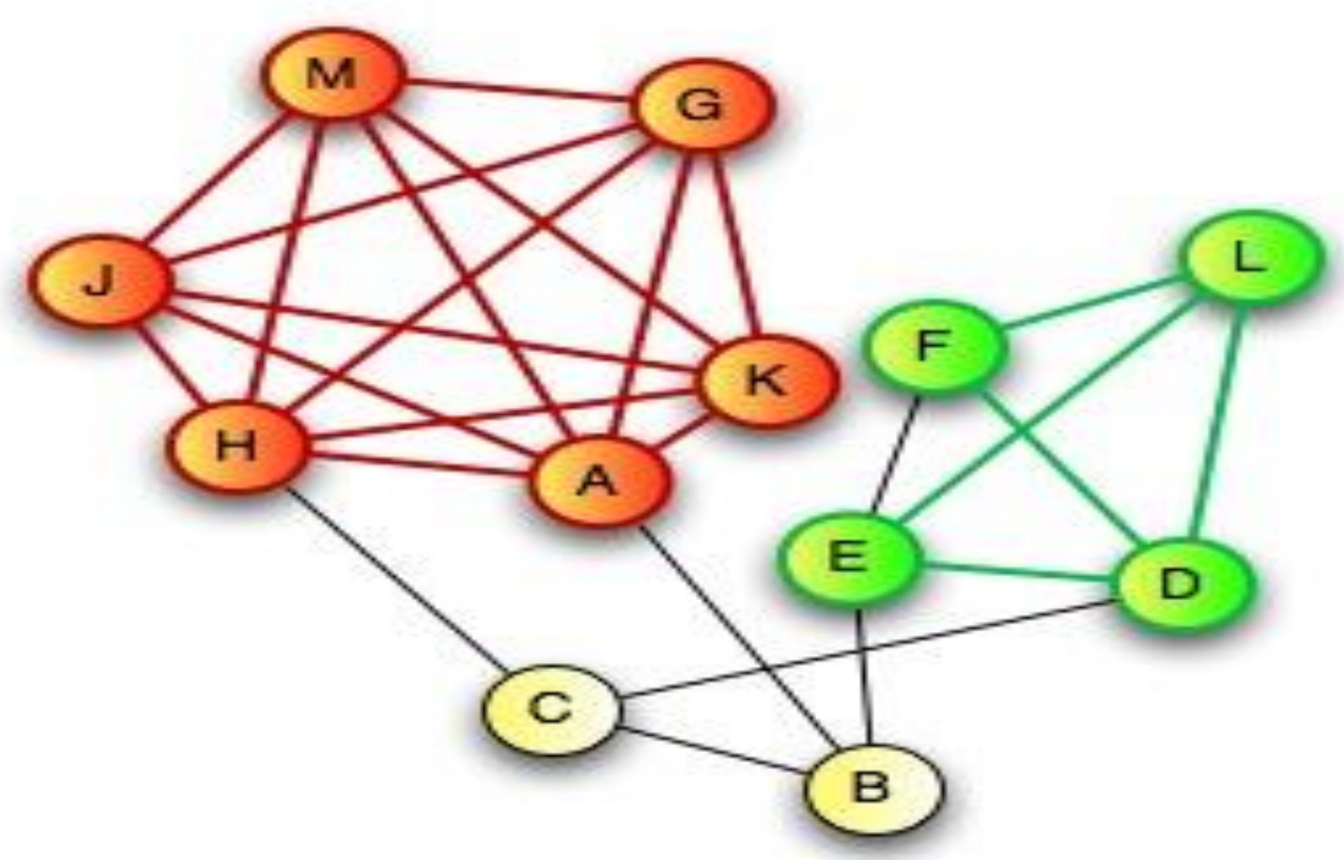
Graph Terminology

- A spanning forest of a graph is a subgraph that contains all its vertices and is a set of trees



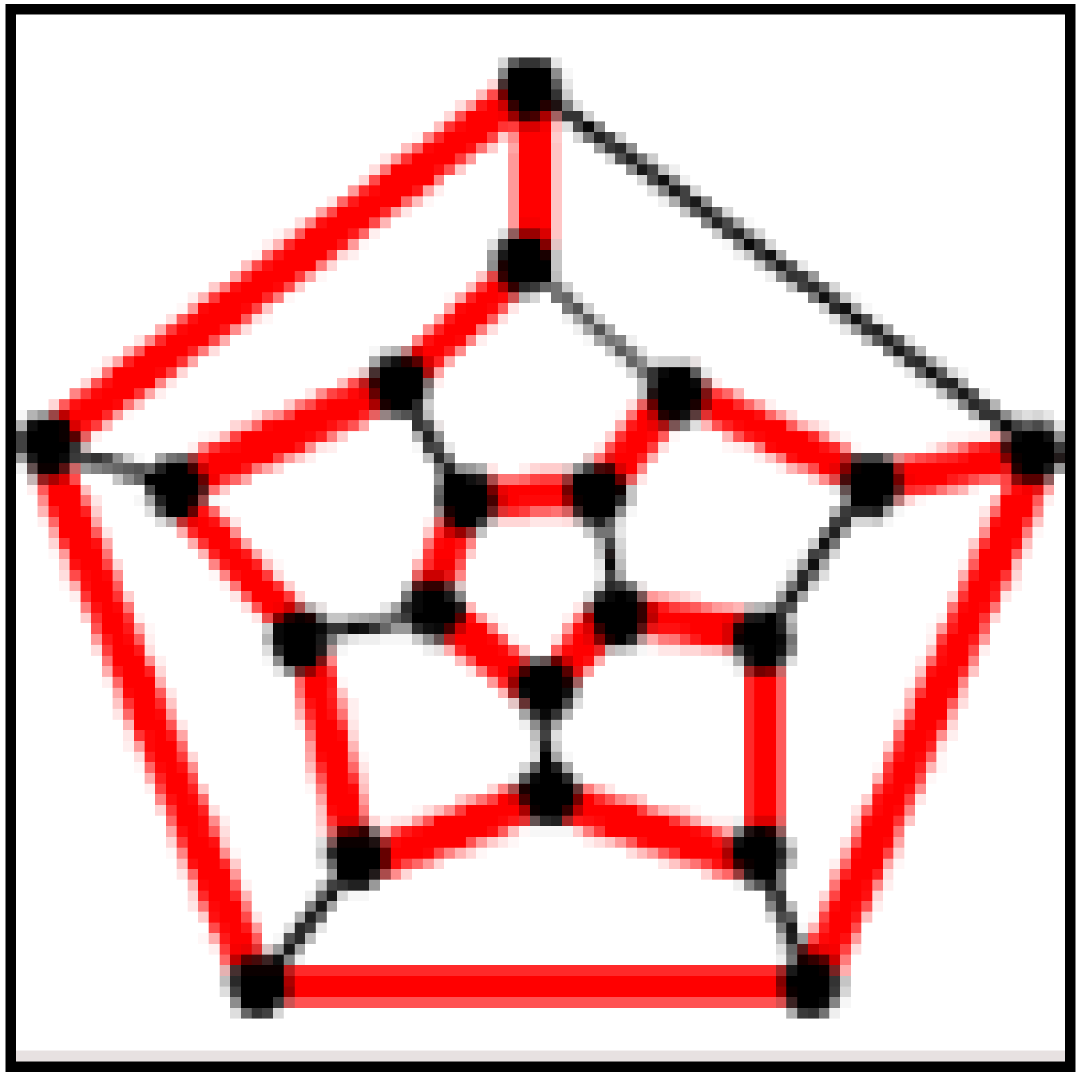
CLIQUES

- Clique: complete subgraph
  - Clique containing vertices{A, G, H, J, K, M}
  - Another clique containing vertics {D,E,F,L}

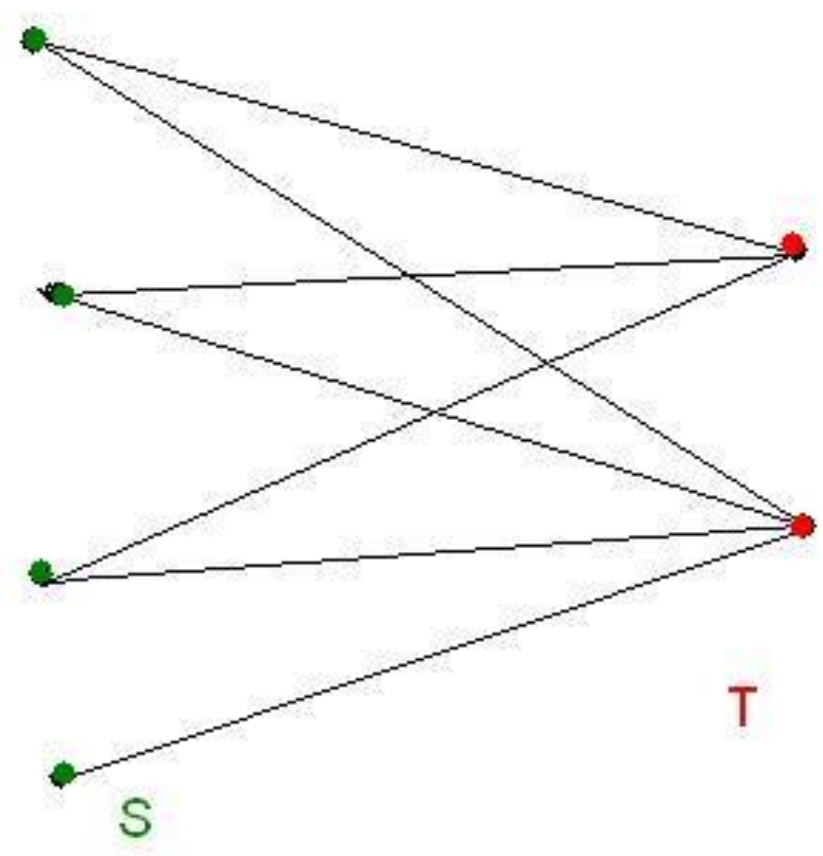


...Graph Terminology

- Hamilton path
  - A simple path that connects two vertices that visits every **vertex** in the graph exactly once
  - If the path is from a vertex back to itself it is called a hamilton tour

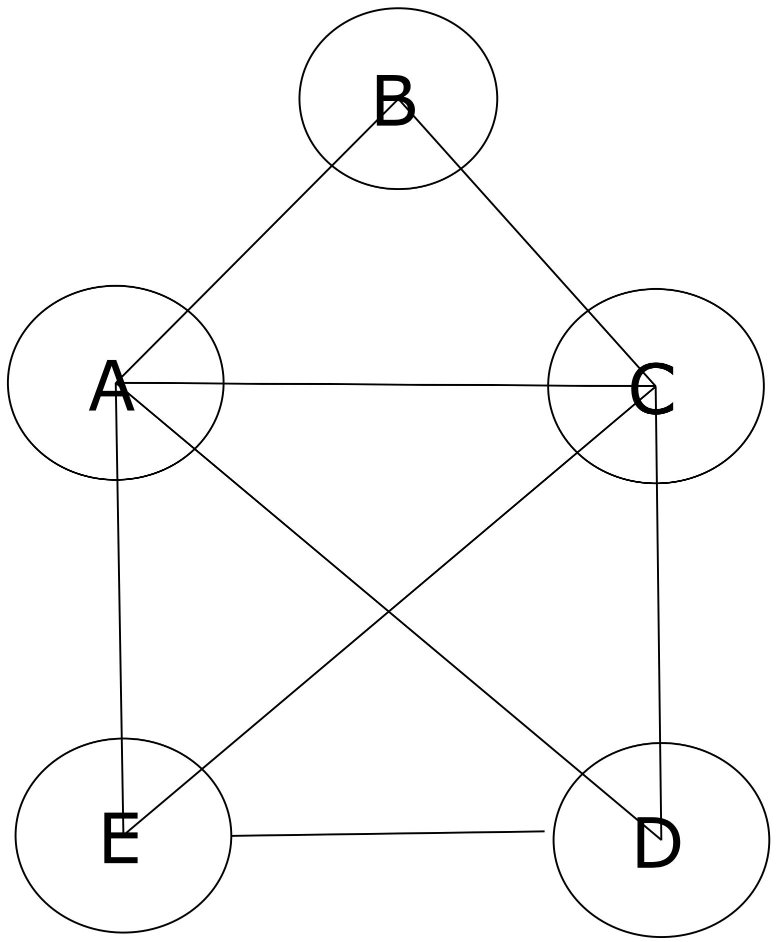


Exercise:
Does this have a hamilton path?



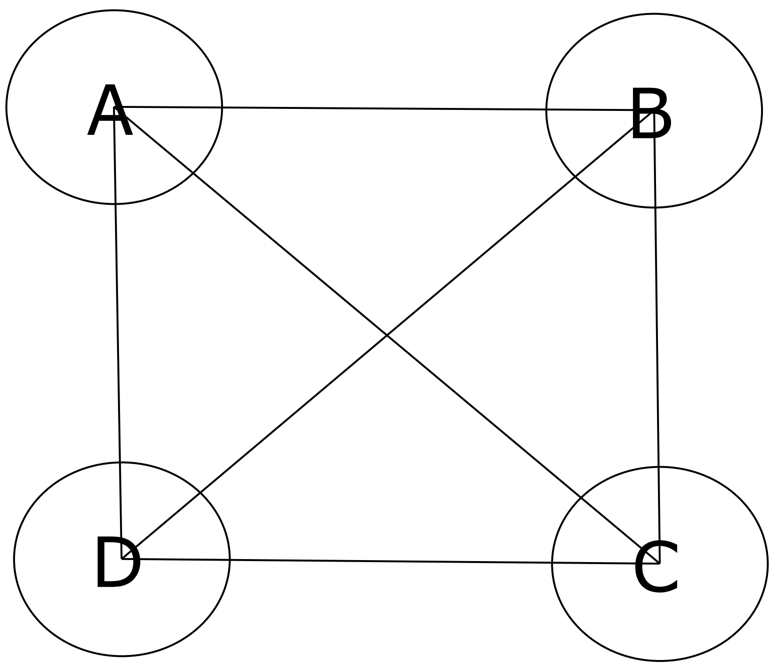
…GRAPH TERMINOLOGY

- Euler path
  - A path the connects two given vertices using each **edge** in the path exactly once.
  - If the path is from a vertex back to itself it is an euler tour



Exercise:
Does this have an Euler path?

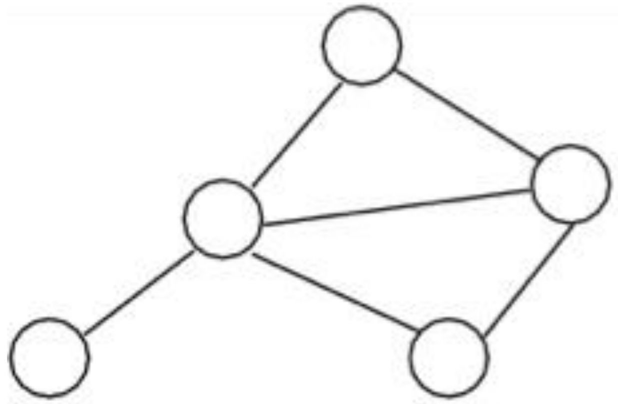
- A graph has an Euler tour if and only if it is connected and all vertices are of even degree
- A graph has an Euler path if and only if it is connected and exactly 2 vertices are of odd degree



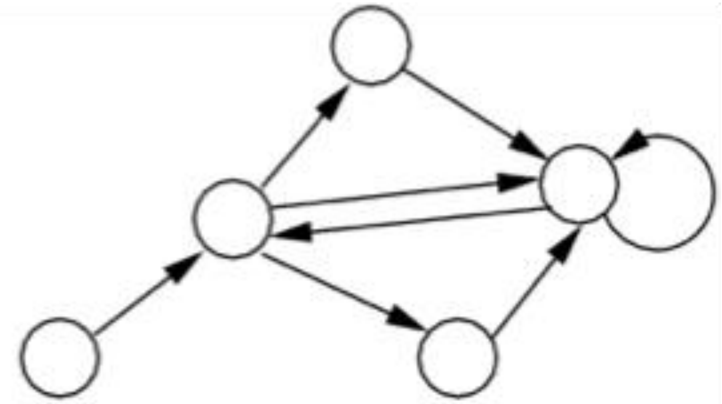
DIRECTED GRAPHS

- If the edges in a graph are directed, the graph is called a directed graph or digraph
  - a digraph with V vertices can have at most V^2 edges
    - Can have self loops
    - edge(u,v) != edge(v,u)
  - a digraph is a tree if there is one vertex which is connected to all other vertices, and there is at most one path between any two vertices
- Unless specified, we assume graphs are undirected in this course.

UNDIRECTED VS DIRECTED GRAPHS



Undirected graph



Directed graph

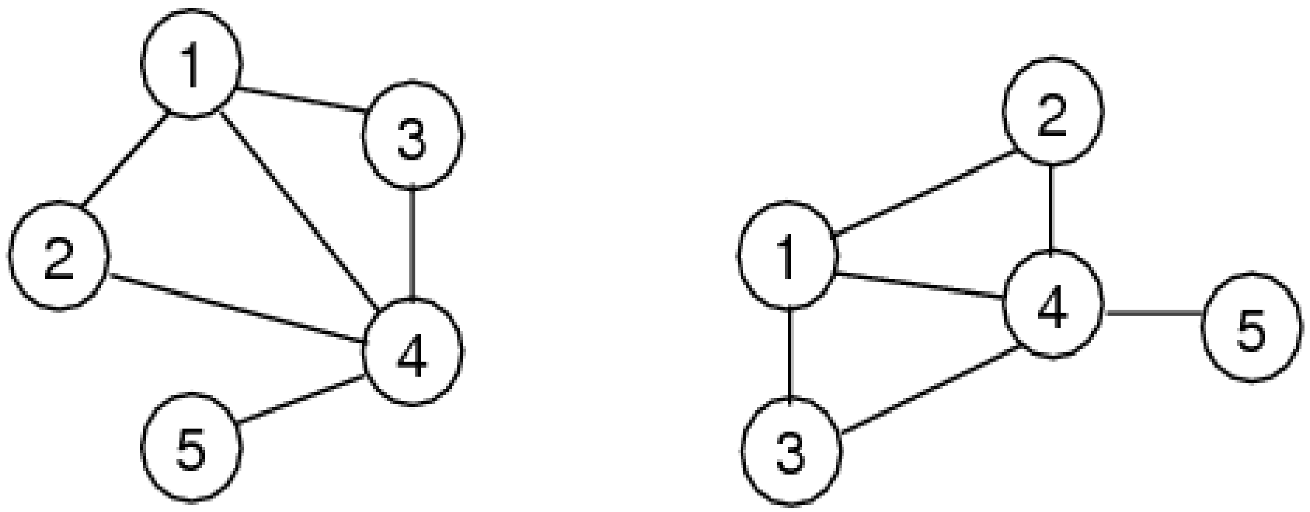
Other types of Graphs

- 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)
  - eg. Transport – may be able to get to new location by bus or train or ferry etc…

DEFINING GRAPHS

- need some way of identifying vertices and their connections
- Below are 4 representations of the **same** graph



1–2 1–3 1–4
2–4
3–4
4–5

1–3
2–1 2–4
4–1 4–3
5–4

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

Graph ADT

- Data:
  - set of edges,
  - set of vertices
- Operations:
  - building: create graph, create edge, add edge
  - deleting: remove edge, drop whole graph
  - scanning: get edges, copy, show
- Notes: In our graphs
  - set of vertices is fixed when graph initialised
  - we treat vertices as ints, but could be Items

ADT Interface For Graphs

- Vertices and Edges

```
typedef int Vertex;

// edge representation
typedef struct edge {
    Vertex v;
    Vertex w;
} Edge;

// edge construction
Edge mkEdge (Vertex v, Vertex w);
```

ADT Interface or Graphs

- Graph basics:

```
// graph handle
typedef struct GraphRep *Graph;

// create a new graph
Graph  graphInit (int noOfVertices);
int validV(Graph g,Vertex v); //validity check
```

- Graph inspection and manipulation:

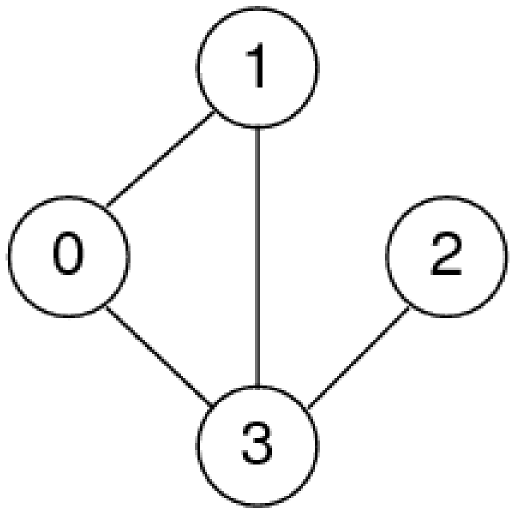
```
void  insertEdge (Graph g, Edge e);
void  removeEdge(Graph g, Edge e);
 Edge *  edges (Graph g, int * nE);
int isAdjacent(Graph g, Vertex v, Vertex w);
int numV(Graph g);
int numE(Graph g);
```

- Whole graph operations:

```
Graph GRAPHcopy (Graph g);
void  GRAPHdestroy (Graph g);
```

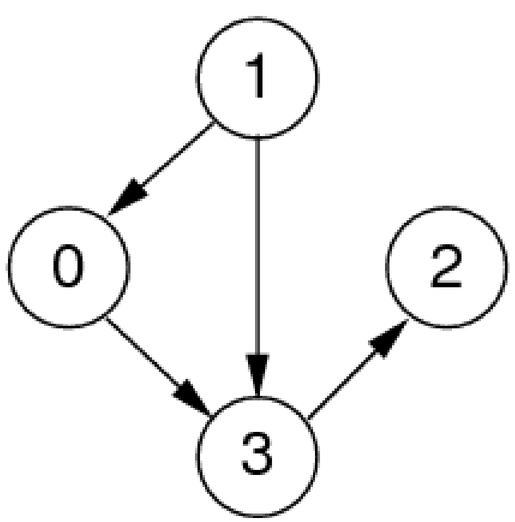
ADJACENCY MATRIX REPRESENTATION

- Edges represented by a VxV 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

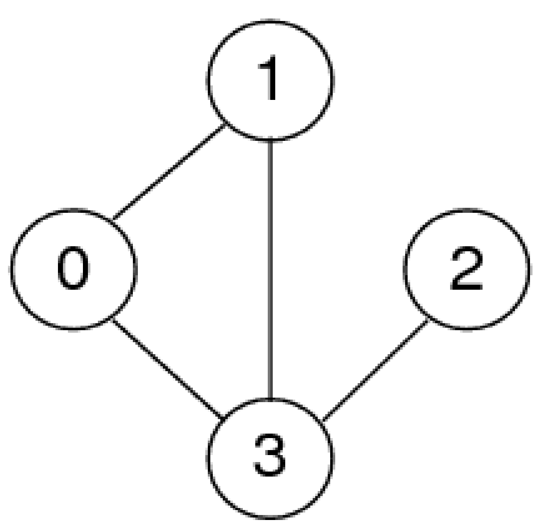
A	0	1	2	3
0	0	0	0	1
1	1	0	0	1
2	0	0	0	0
3	0	0	1	0

ADJACENCY MATRIX REPRESENTATION

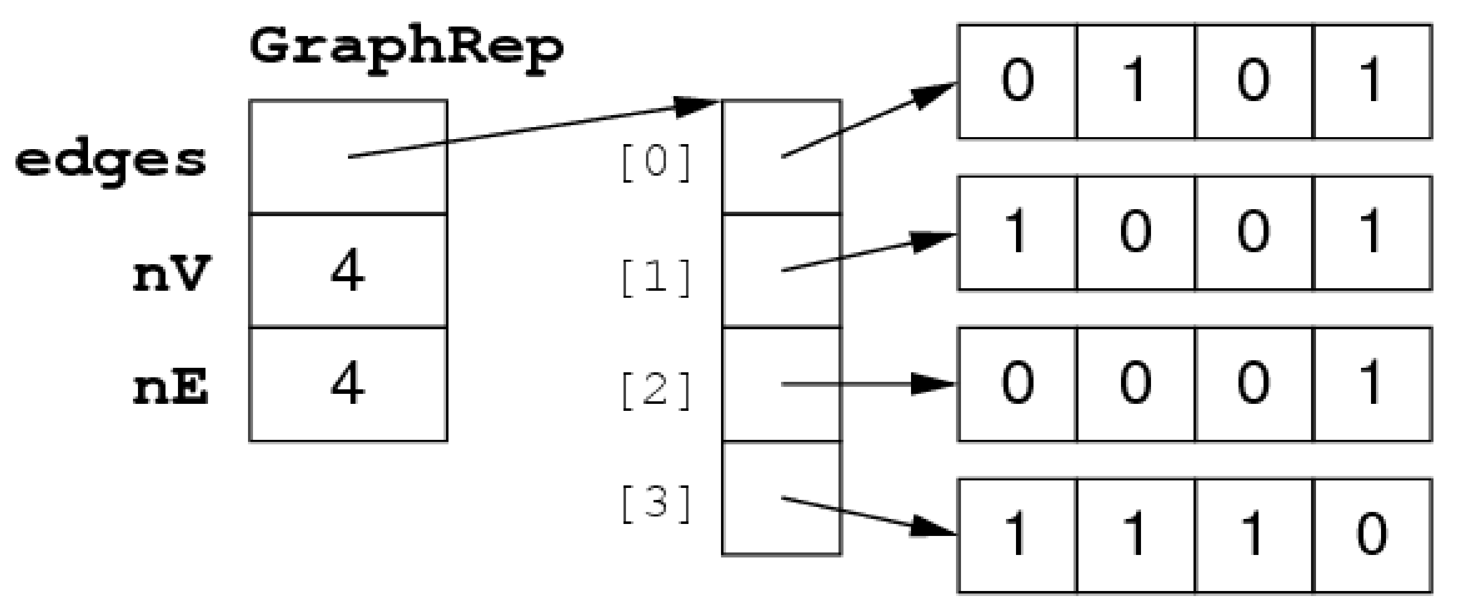
- Advantages
  - easily implemented in C as 2-dimensional array
  - can represent graphs, digraphs and weighted graphs
    - graphs: symmetric boolean matrix
    - digraphs: non-symmetric boolean matrix
    - weighted: non-symmetric matrix of weight values
- Disadvantages:
  - if few edges ⇒ sparse, memory-inefficient

Adjacency Matrix Implementation

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

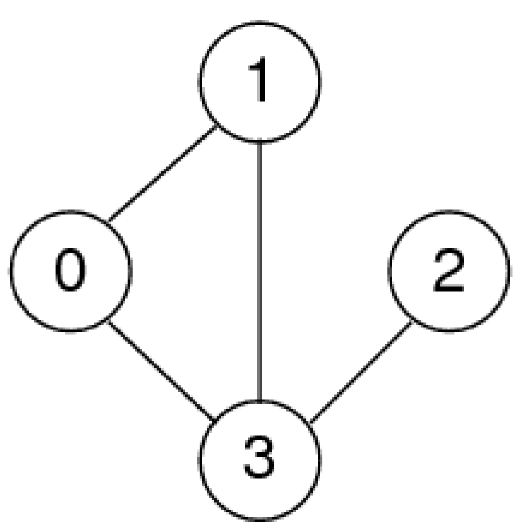


Undirected graph

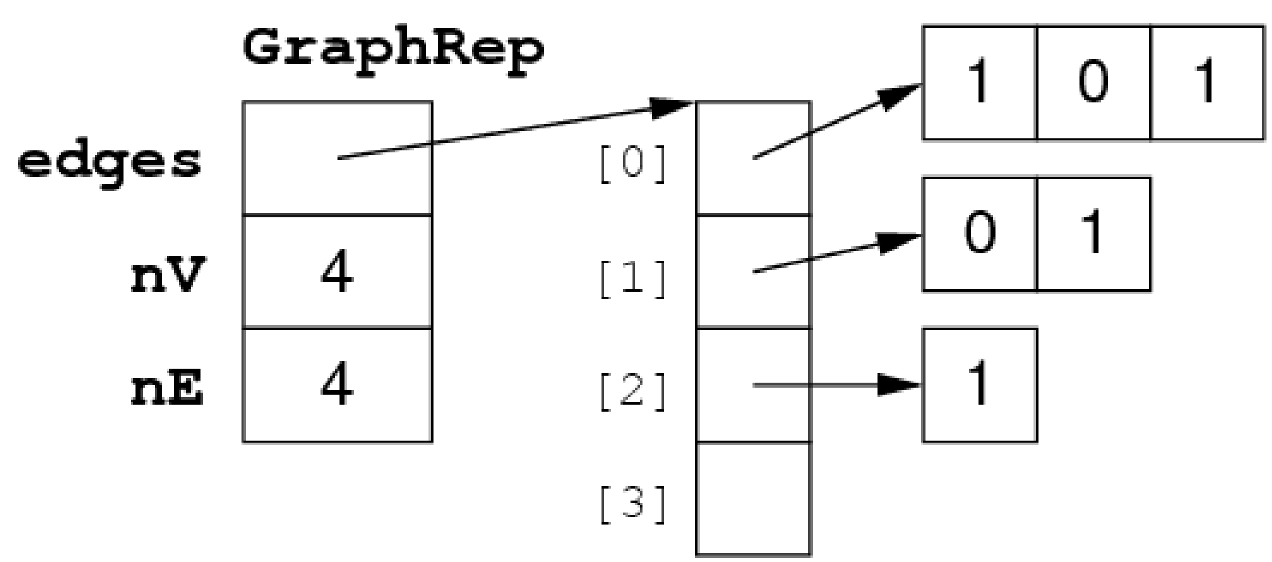


Adjacency Matrix Storage Optimisation

- Storage cost:
  - V int ptrs + V^2 ints If the graph is sparse, most storage is wasted.
- A storage optimisation:
  - If undirected, 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 $v < w$.



Undirected graph



Cost of operations on adjacency matrix

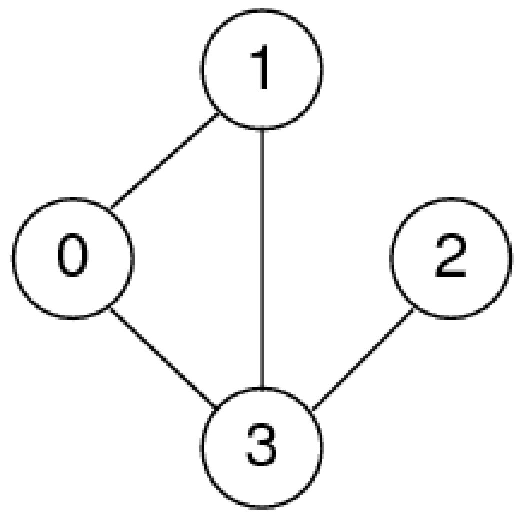
- Cost of operations:
  - initialisation: $O(V^2)$ (initialise $V \times V$ matrix)
  - insert edge: $O(1)$ (set two cells in matrix)
  - delete edge: $O(1)$ (unset two cells in matrix)

- See code for the implementation of these functions and their cost
  - int isAdjacent(Graph g, Vertex v, Vertex w);
  - Vertex * adjacentVertices(Graph g, Vertex v, int * nV);

- Exercise : write the functions and find the cost for
  - Edge * edges (Graph g, int * nE);

ADJACENCY LIST REPRESENTATION

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



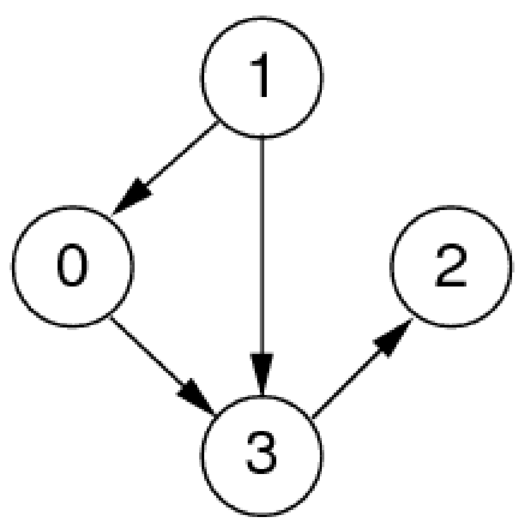
Undirected graph

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] = < >

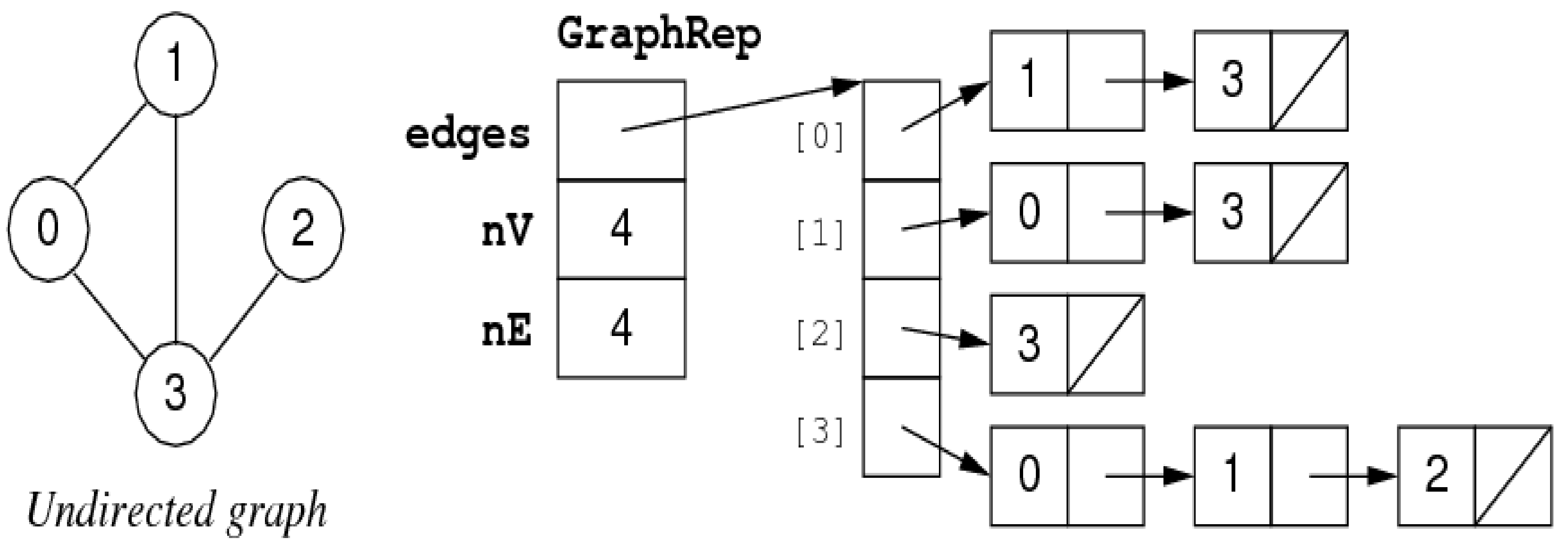
A[3] = <2>

Adjacency List Representation

- Advantages
  - relatively easy to implement in C
  - can represent graphs and digraphs
  - memory efficient if E/V relatively small
- Disavantages:
  - one graph has many possible representations (unless lists are ordered by same criterion e.g. ascending)

ADJACENCY MATRIX IMPLEMENTATION

```
typedef struct vNode *VList;
struct vNode { Vertex v; VList next; };
typedef struct GraphRep {
    int nV;         // #vertices
    int nE;         // #edges
    VList *edges; // array of lists
} GraphRep;
```



COSTS OF OPERATIONS ON ADJACENCY LISTS

- Cost of operations:
  - initialisation: *O(V)*   (initialise *V* lists)
  - insert edge: *O(1)*   (insert one vertex into list)
  - delete edge: *O(V)*   (need to find vertex in list)
- If vertex lists are sorted insert requires search of list $\Rightarrow$ *O(V)*
- If we do not want to allow parallel edges it is O(V)
- delete always requires a search, regardless of list order

Costs of Operations on Adjacency Lists

- See code for the implementation of these functions and their cost
  - int isAdjacent(Graph g, Vertex v, Vertex w);
  - Vertex * adjacentVertices(Graph g, Vertex v, int * nV);

- Exercise : write the functions and find the cost for
  - Edge * edges (Graph g, int * nE);

Comparison of different Graph Representations

	adjacency matrix	adjacency list
space	V^2	V + E
initialise empty	V^2	V
copy	V^2	E
destroy	V	E
insert edge	1	V
find/remove edge	1	V
is v isolated?	V	1
isAdjacent	1	V