Graphs

Computing 2 COMP1927 17x1
Sedgewick Part 5: Chapter 17

WHAT ARE GRAPHS

Many applications require

- a collection of **items** (i.e. a set)
- and **relationships/connections** between items
- and these relationships lead to natural questions – is there a way to reach from one item to another using these connections ?, how many other items can be reached from a given item?

Examples include:

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

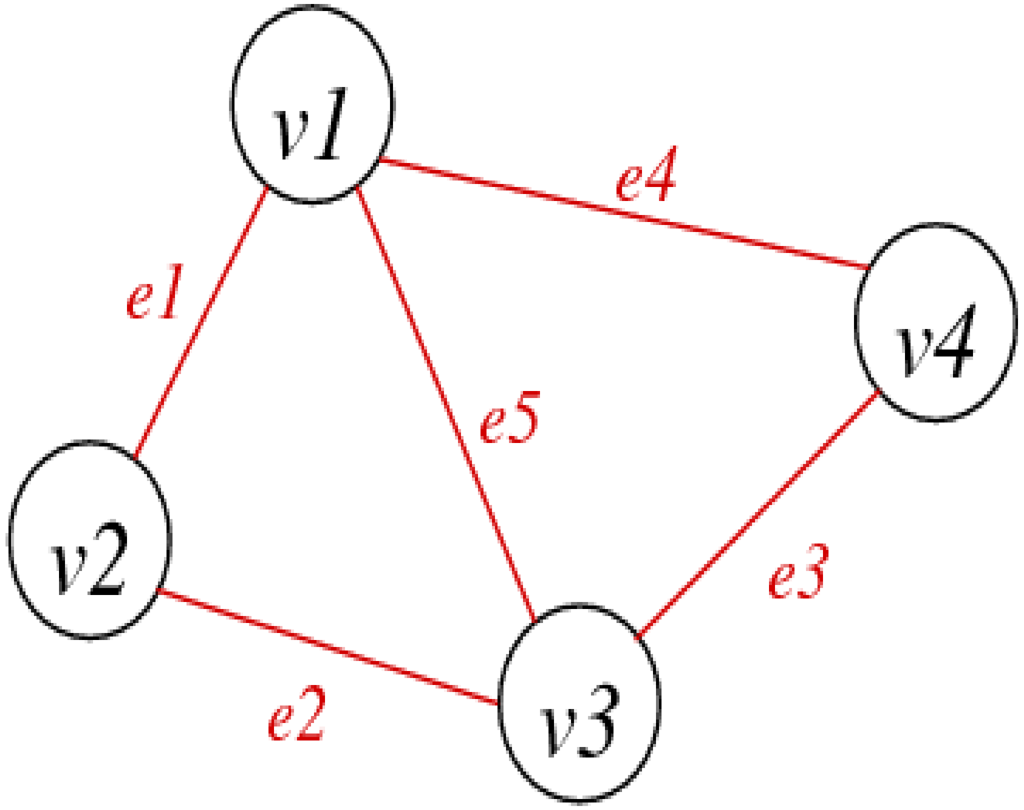
Collection types we've seen so far

- lists…linear sequence of items (stack, queue)
- trees … branched hierarchy of items

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

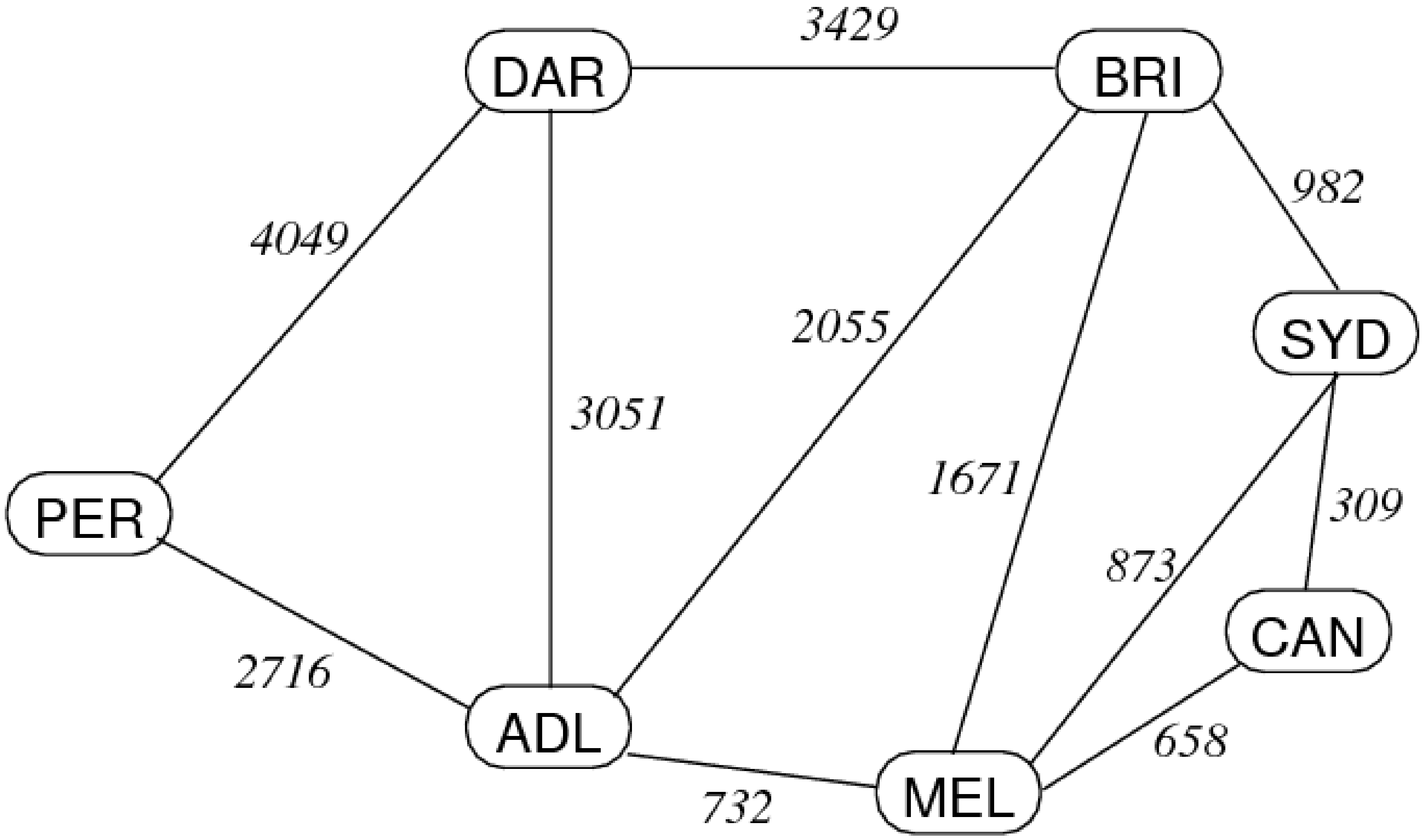
Graph	Vertices	Edges
Communication	Telephones, Computers	Cables
Games	Board positions	Legal moves
Social networks	People	Friendships
Scheduling	Tasks	Precedence Constraints
Circuits	Gates, Registers, Processors	Wires
Transport	Intersections/ airports	Roads, flights

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

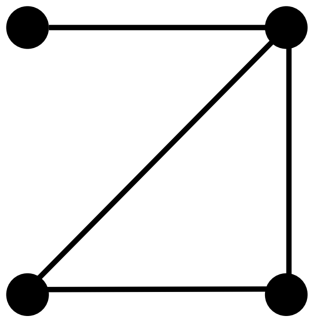
- is there a way to get from item A to item B?
- what is the best way to get from A to B?
- which vertices are connected?

Graph algorithms are in general significantly more difficult than list or tree processing

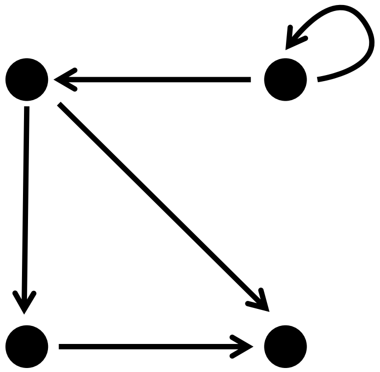
- no implicit order of the items
- graphs can contain cycles
- concrete representation is less obvious
- complexity of algorithms depend connection complexity

GRAPH TYPES

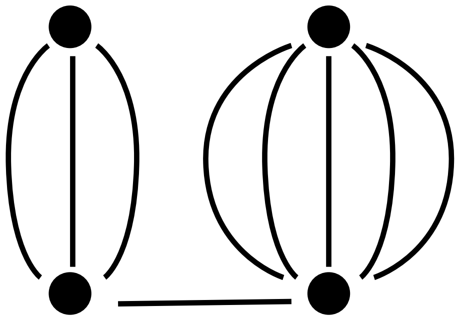
Depending on the application, graphs can have different properties:



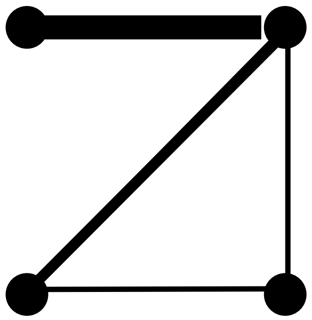
undirected



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

PROPERTIES OF GRAPHS

Terminology: |V| and |E| normally written as V and E

- a graph with V vertices has at most V(V-1)/2 edges

The ratio V:E has at most V(V-1)/2 edges

- if E is closer to $V^2/2$, the graph is dense
- If E is closer to V, the graph is sparse

Knowing whether a graph is sparse or dense is important

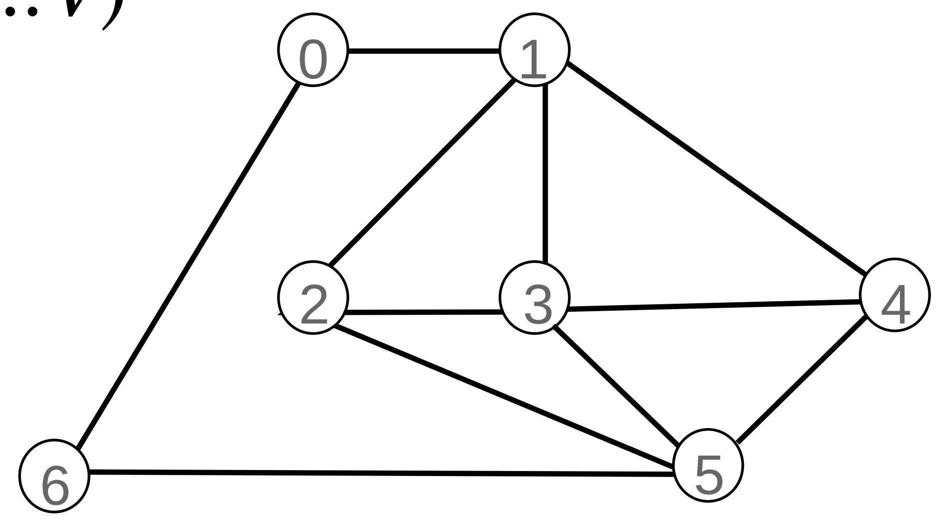
- may affect choice of data structures to represent graph
- may affect choice of algorithms to process graph

Describing Graphs

Defining graphs

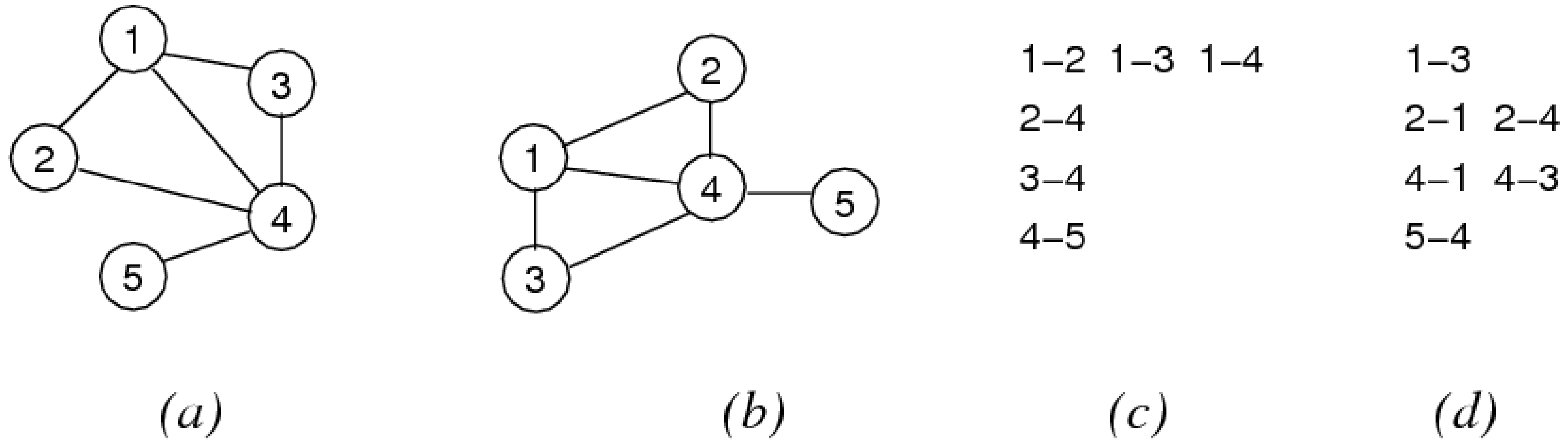
- V need to be identified (e.g. number $1..V$)
- E need to be drawn or enumerated

E.g.: In our 7 vertex graph:



- V (number of vertices): 7
- E (number of edges): 11
- Maximum number of edges : $7*(7-1)/2 = 21$

E.g. four representations of the same graph



(a)

(b)

1–2 1–3 1–4
2–4
3–4
4–5

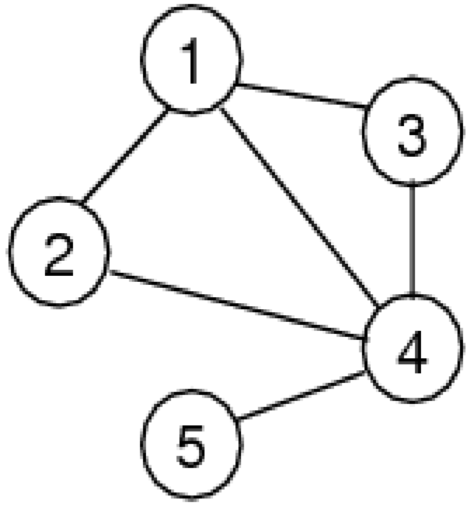
(c)

1–3
2–1 2–4
4–1 4–3
5–4

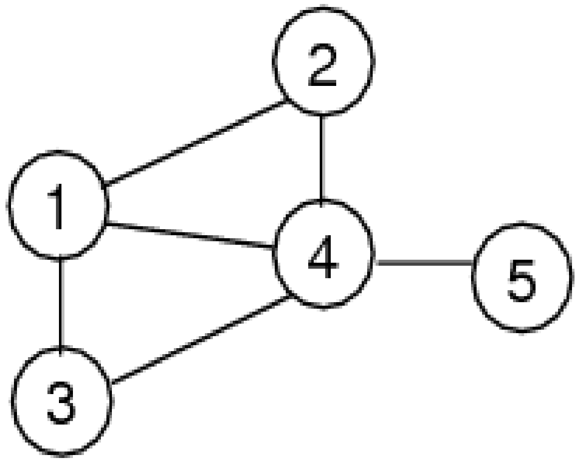
(d)

DEFINING GRAPHS

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



(a)



(b)

1–2 1–3 1–4
2–4
3–4
4–5

(c)

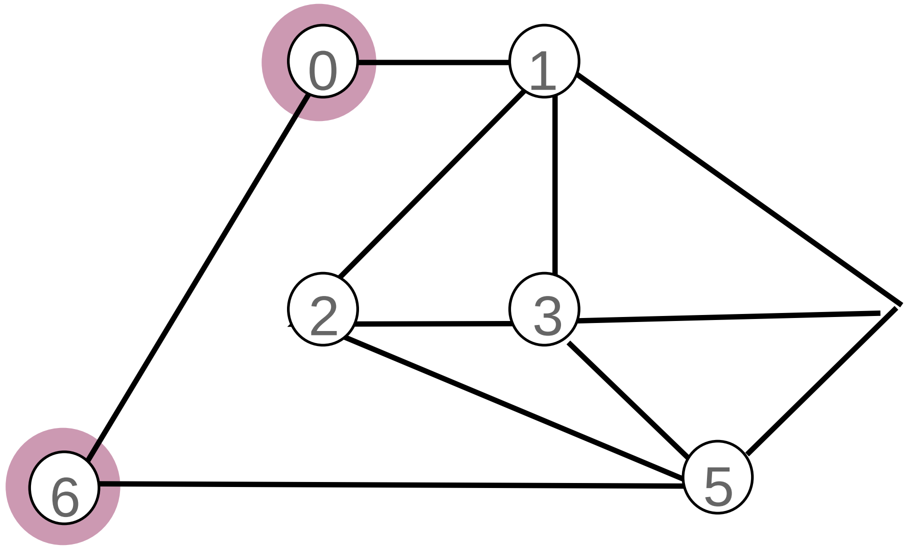
1–3
2–1 2–4
4–1 4–3
5–4

(d)

GRAPH TERMINOLOGY

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

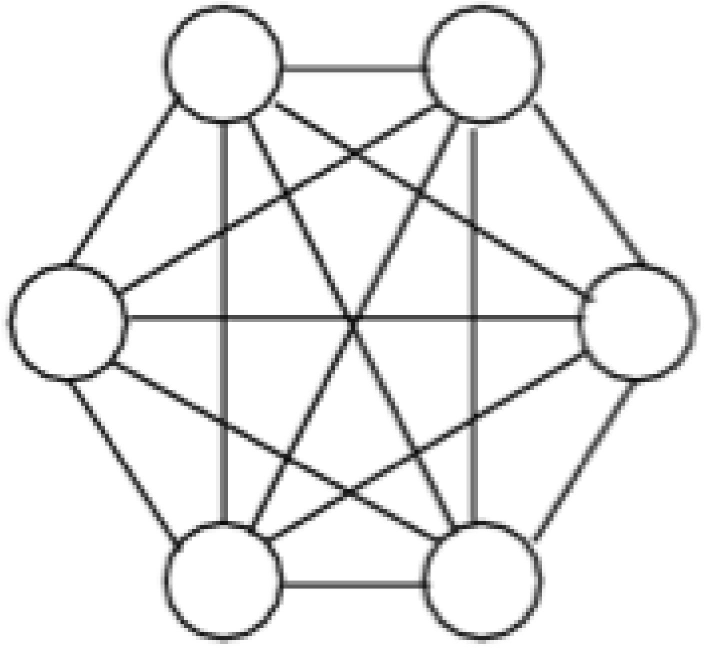
- *v* and *w* are adjacent
- *e* is incident on both *v* and *w*



Degree of a vertex *v* = number of edges incident on *v*

GRAPHS: TERMINOLOGY

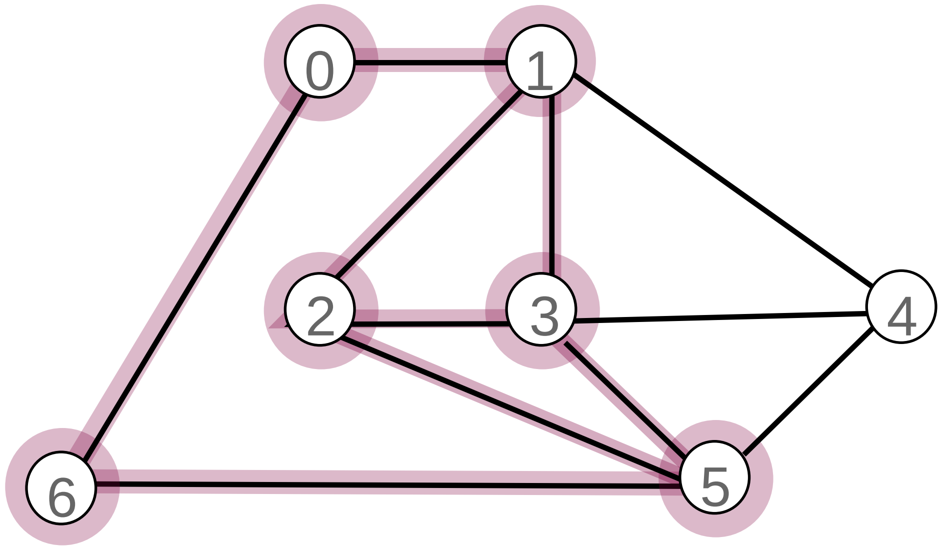
- The degree of a vertex is the number of edges from the vertex
- A complete graph is a graph where every vertex is connected to all the other vertices
  - $E = V(V-1)/2$
  - The degree of every vertex is *V-1*



Complete Graph

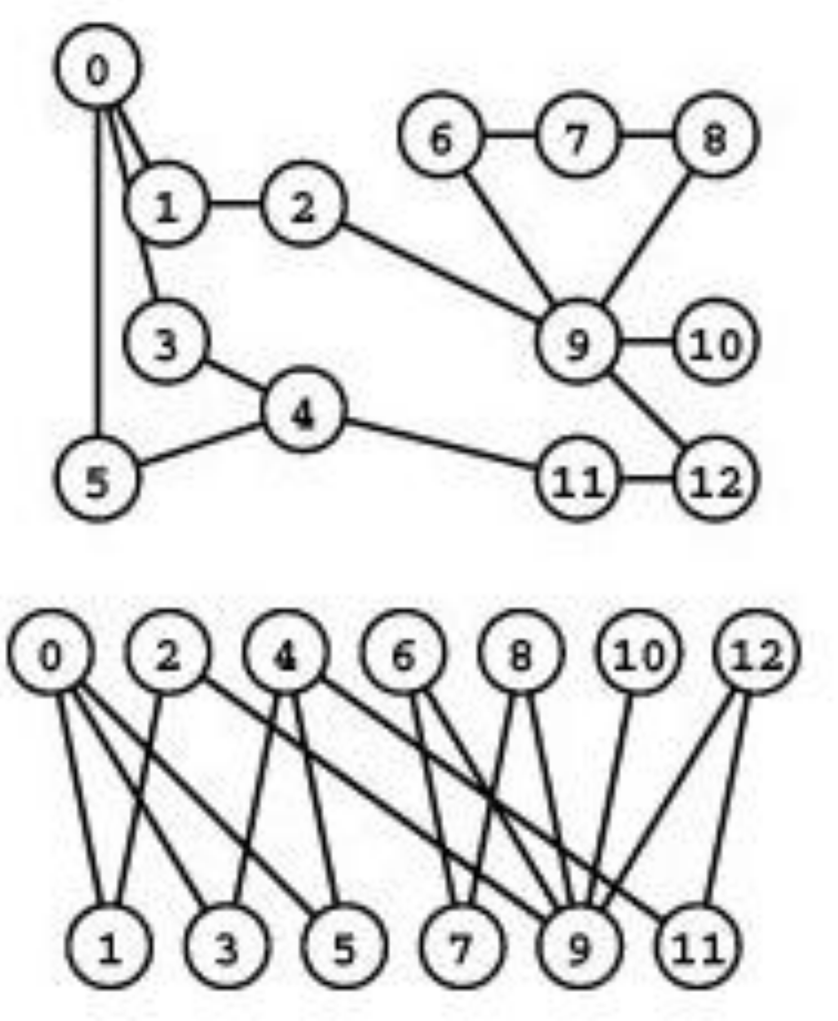
GRAPH TERMINOLOGY

Subgraph: a subset of vertices with their associated edges

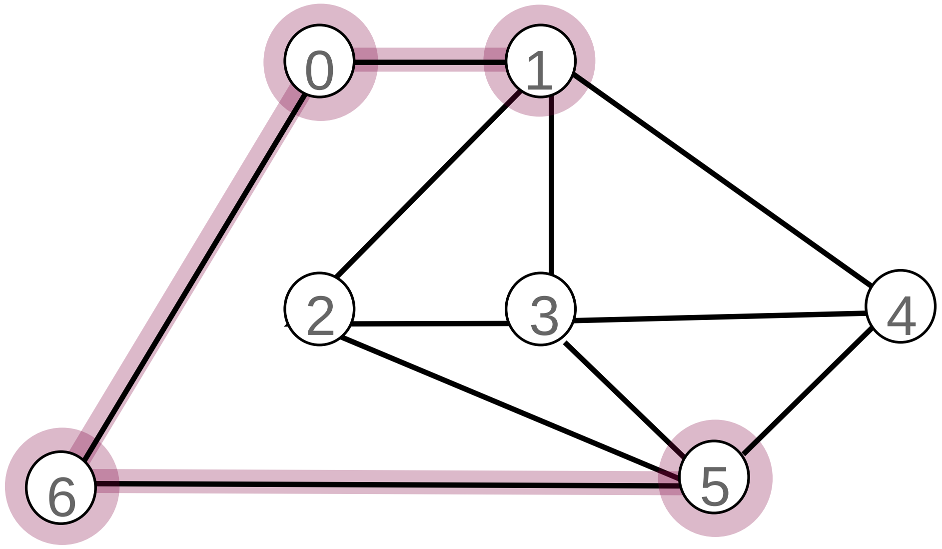


GRAPH TERMINOLOGY

Bipartite graph: a graph whose vertices can be divided into two sets such that all edges connect a vertex in one set with a vertex in the other set. F



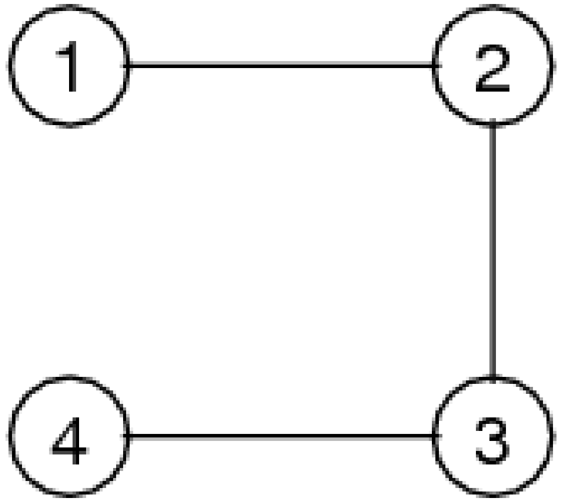
GRAPH TERMINOLOGY: PATHS



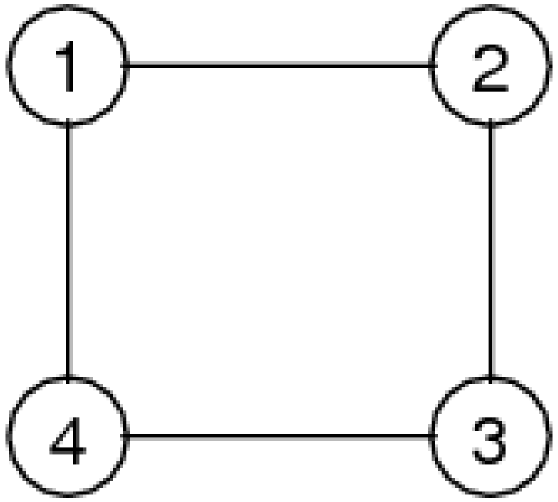
Path: a sequence of vertices where each successive vertex is adjacent (connected) to its predecessor - e.g., 1,0,6,5

Simple path - the path doesn't have any repeating vertices

Cycle – A path where last vertex in path is same as first vertex in path



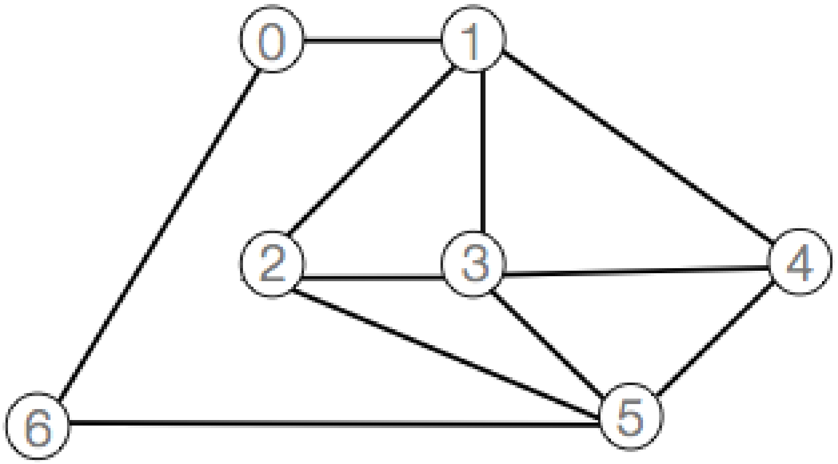
Path: 1–2, 2–3, 3–4



Cycle: 1–2, 2–3, 3–4, 4–1

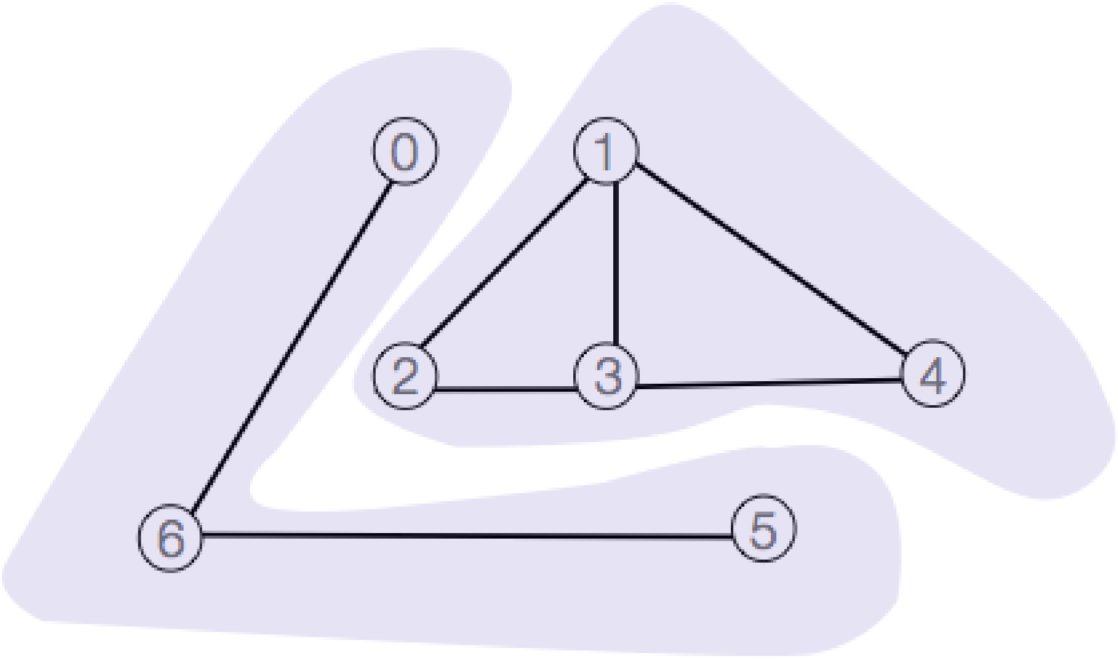
GRAPH TERMINOLOGY

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



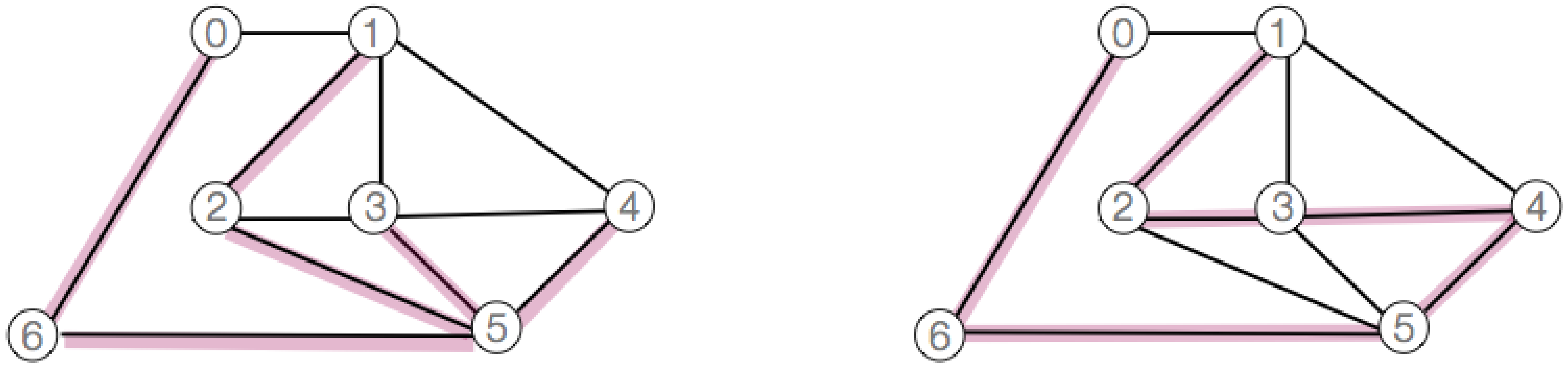
GRAPH TERMINOLOGY

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

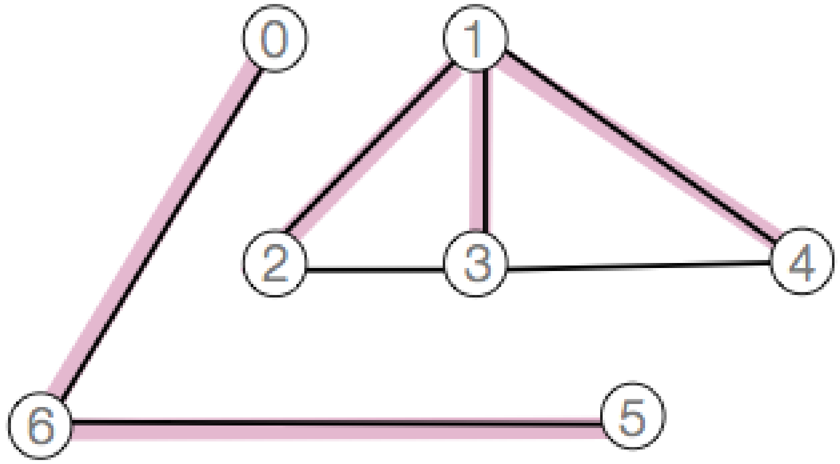


GRAPH TERMINOLOGY

- A graph is a tree if there is exactly one path between each pair of vertices
- A spanning tree of a connected graph is a sub-graph (a sub set of graph G) that contains all of the graph's vertices and is a single tree

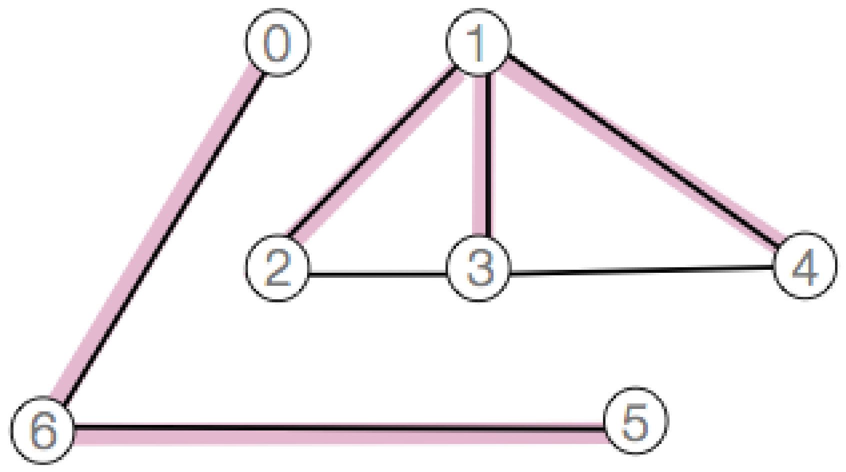


- A spanning forest of a graph is a sub-graph that contains all its vertices and is a forest (a set of trees)



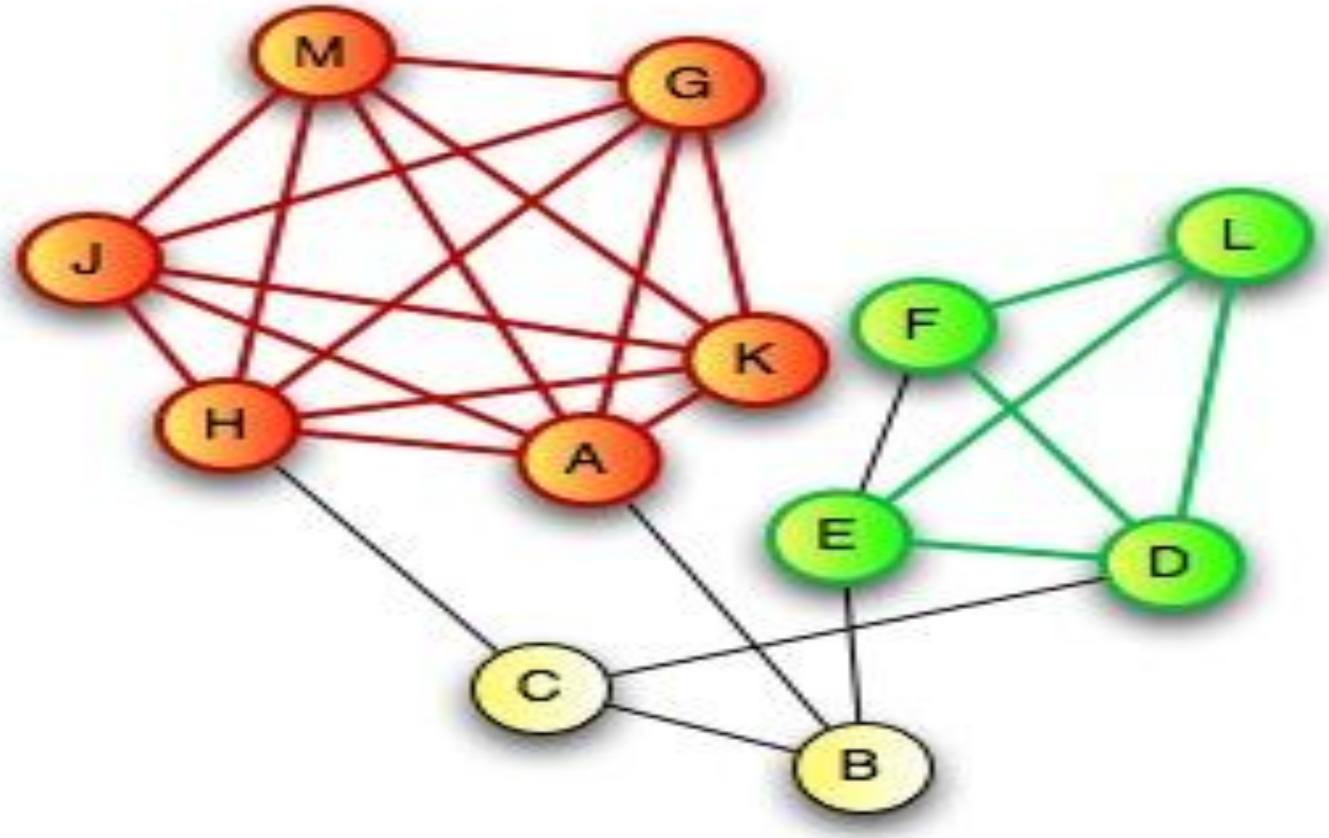
GRAPH TERMINOLOGY

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



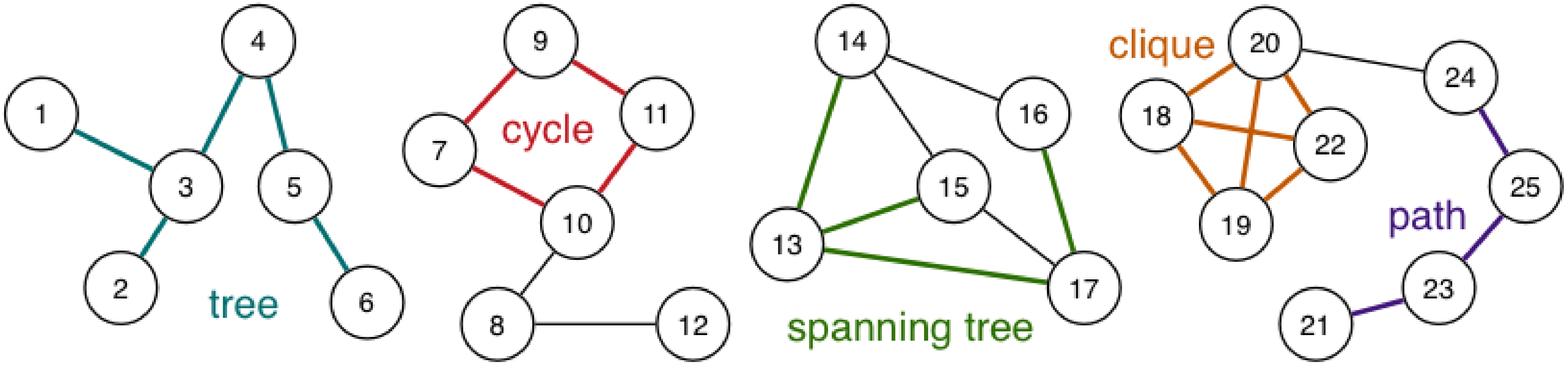
CLIQUES

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



CLIQUES

- Consider the following *single graph*:

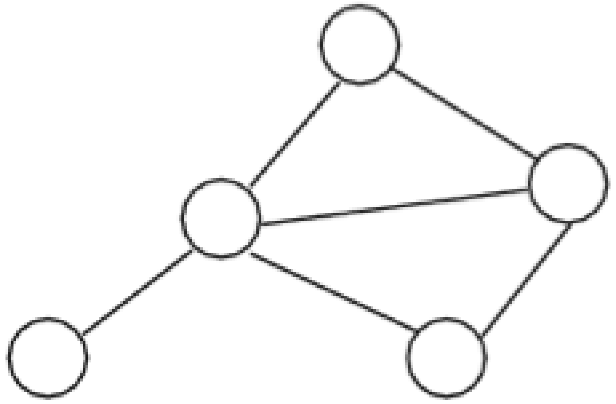


- This graph has 25 vertices, 28 edges and 4 connected components

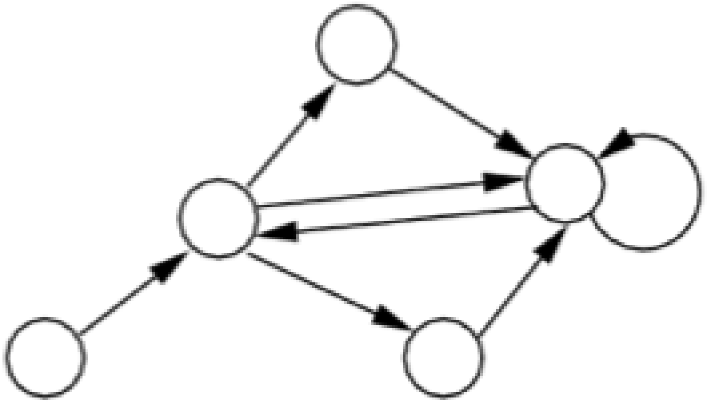
OTHER TYPES OF GRAPHS

- Directed graph (di-graph): each edge has an associated direction (e.g. hyperlinks)
  - 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
- edges in directed graph are known as *directed edges*
- first vertex in a diagraph is the *source*; the second vertex is the *destination* (directed edge points from source to destination*)*
- *indegree* (number of edges where it is the destination)
- *outdegree* (number of edges where it is the source)

Undirected vs Directed Graphs



Undirected graph

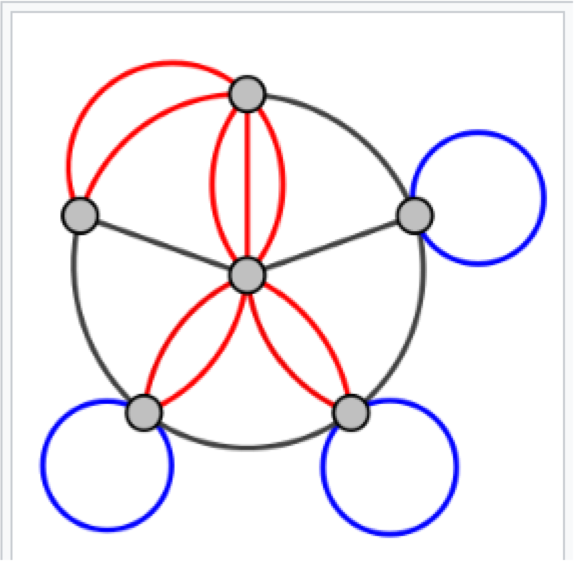


Directed graph

OTHER TYPES OF GRAPHS

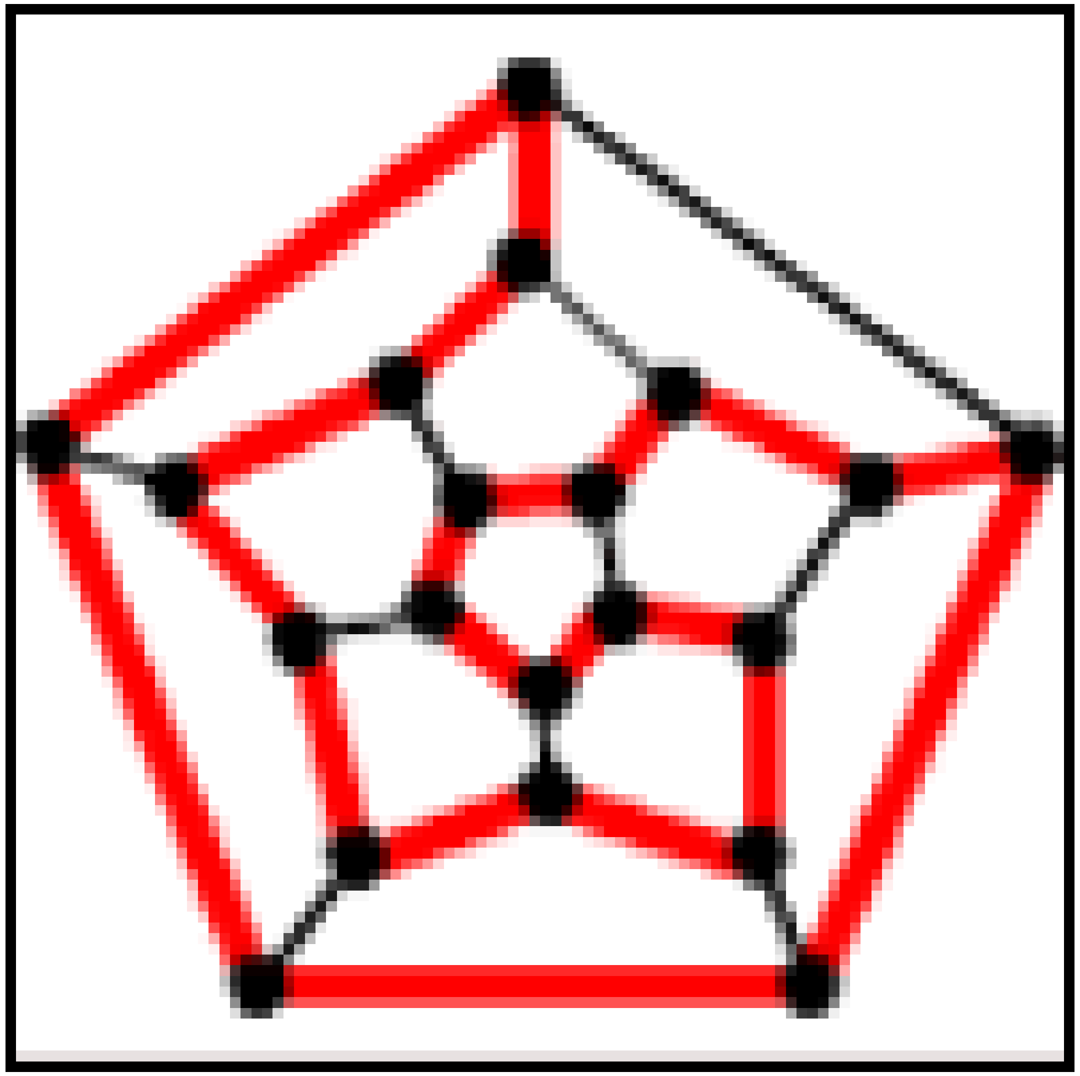
- Weighted graph
  - each edge has an associated value (*weight*)
  - e.g. road map (weights on edges are distances between cities)

- Multi-graph
  - allow *multiple edges* (also called parallel 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…

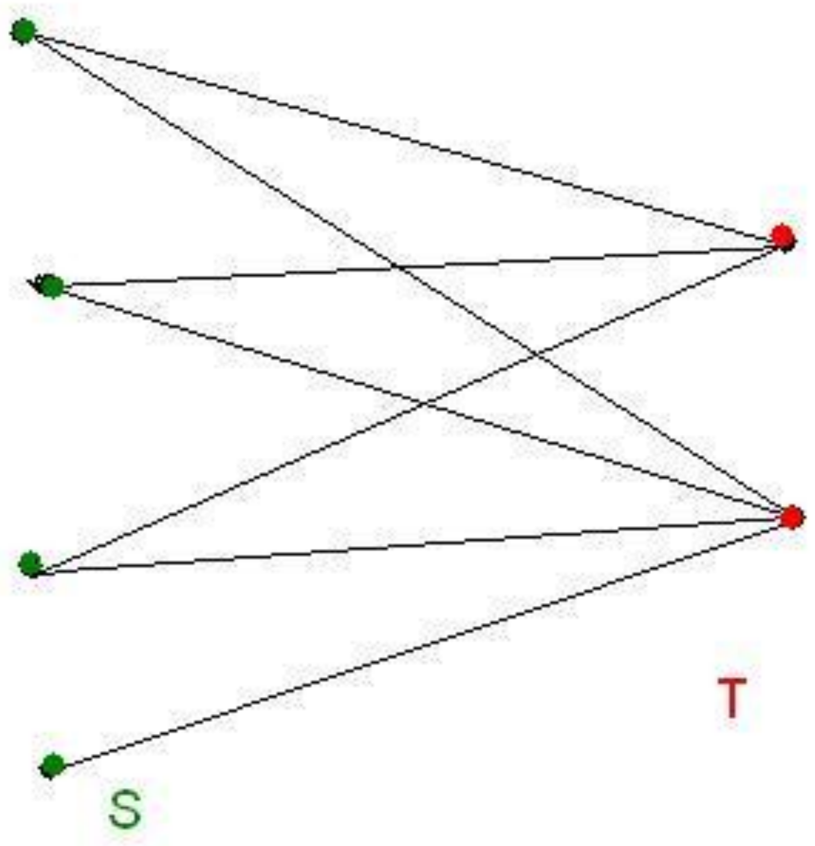


…GRAPH TERMINOLOGY

- 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 cycle

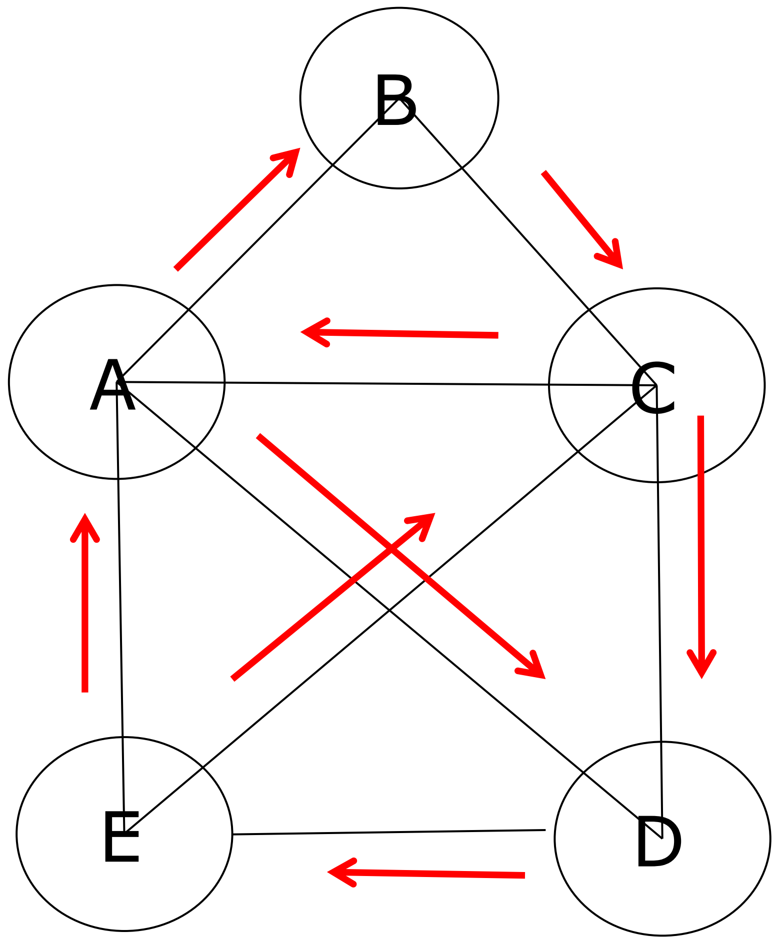


EXERCISE:
DOES THIS HAVE A HAMILTON PATH?



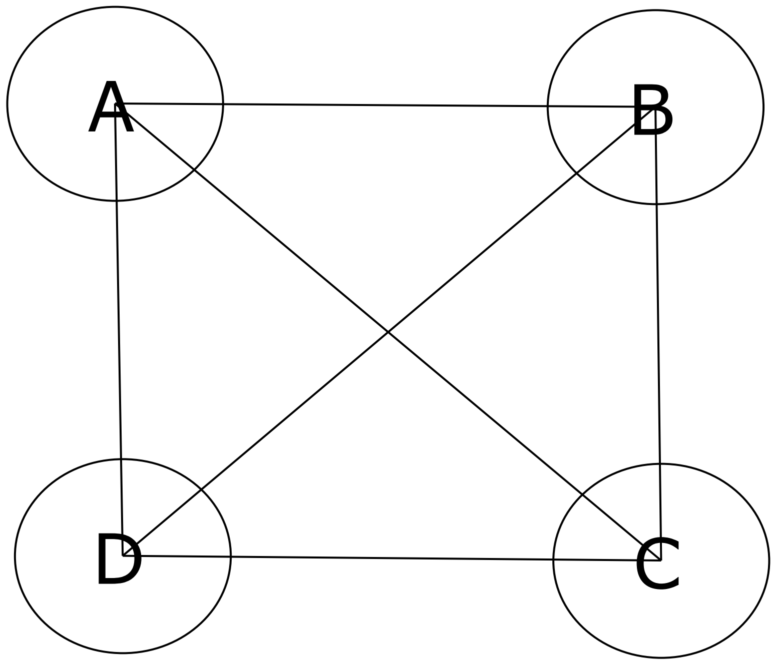
…GRAPH TERMINOLOGY

- 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?

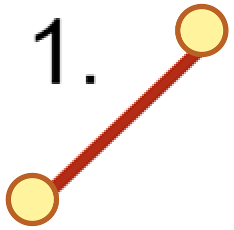
- A graph has an Euler tour if and only if it is connected and all vertices are of even degree
- A graph has an Euler path if and only if it is connected and exactly 2 vertices are of odd degree



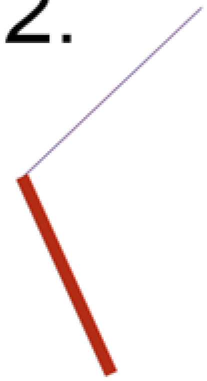
AN EULER PATH/CIRCUIT

- An Euler path starts and ends at different vertices.
- An Euler circuit starts and ends at the same vertex.

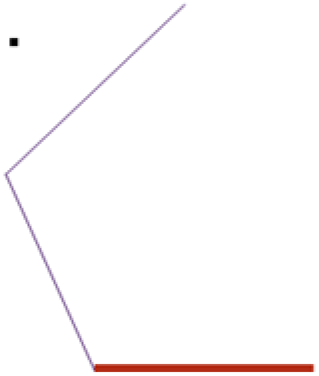
1.



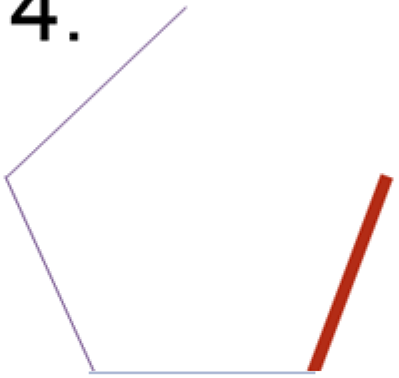
2.

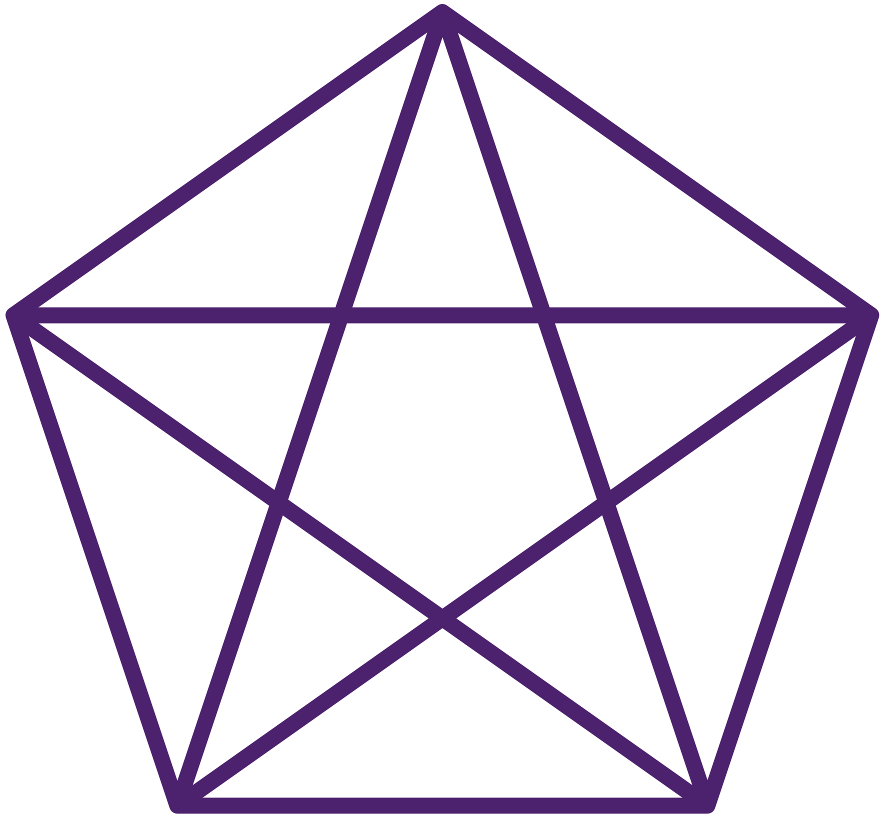


3.

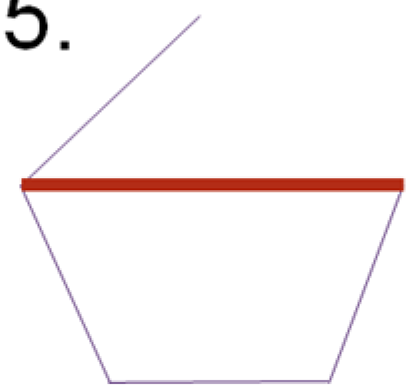


4.

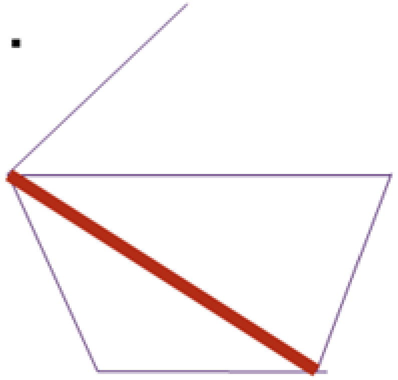




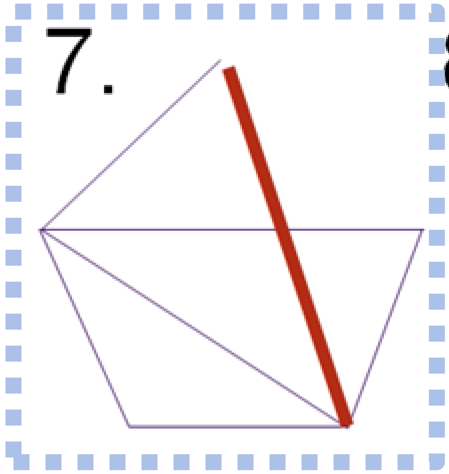
5.



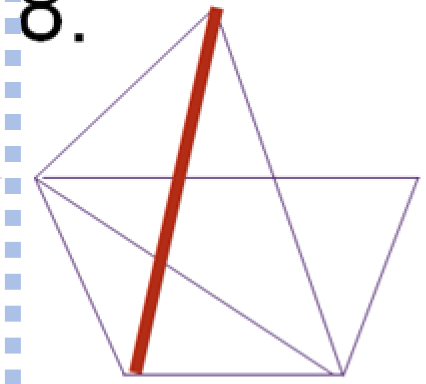
6.



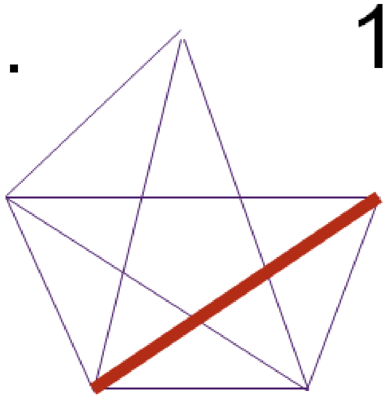
7.



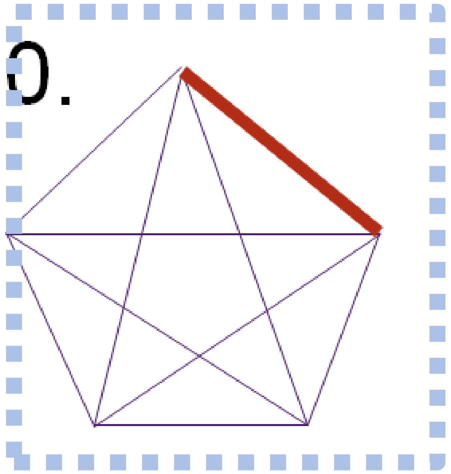
8.



9.



10.



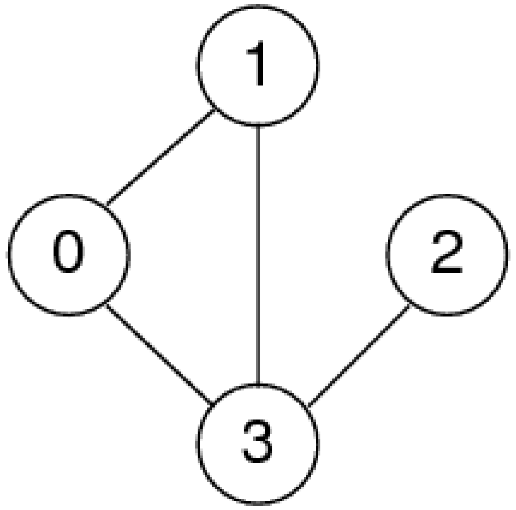
GRAPH ADT

- 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

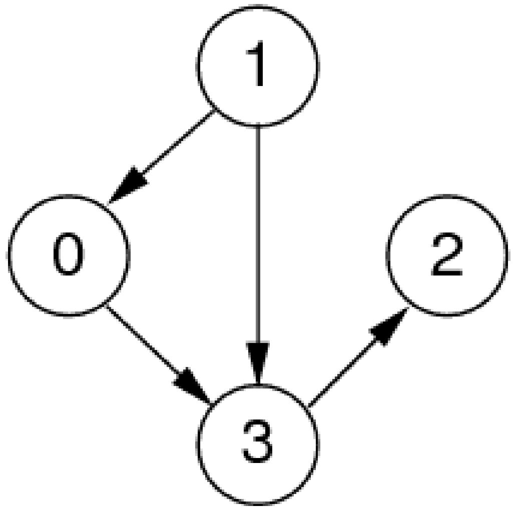
ADJACENCY MATRIX REPRESENTATION

- 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

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);
```

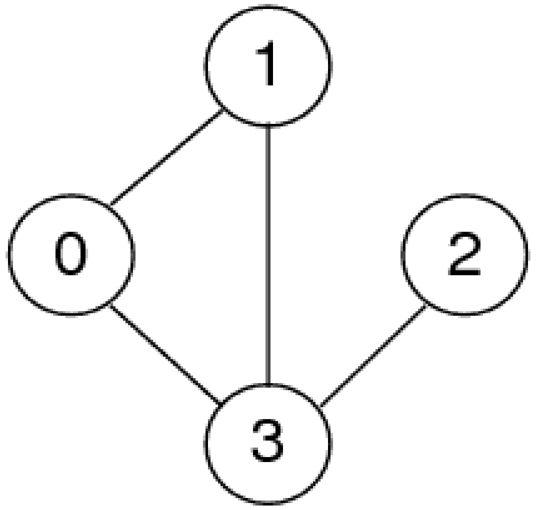
ADT INTERFACE OR GRAPHS

- Graph basics:

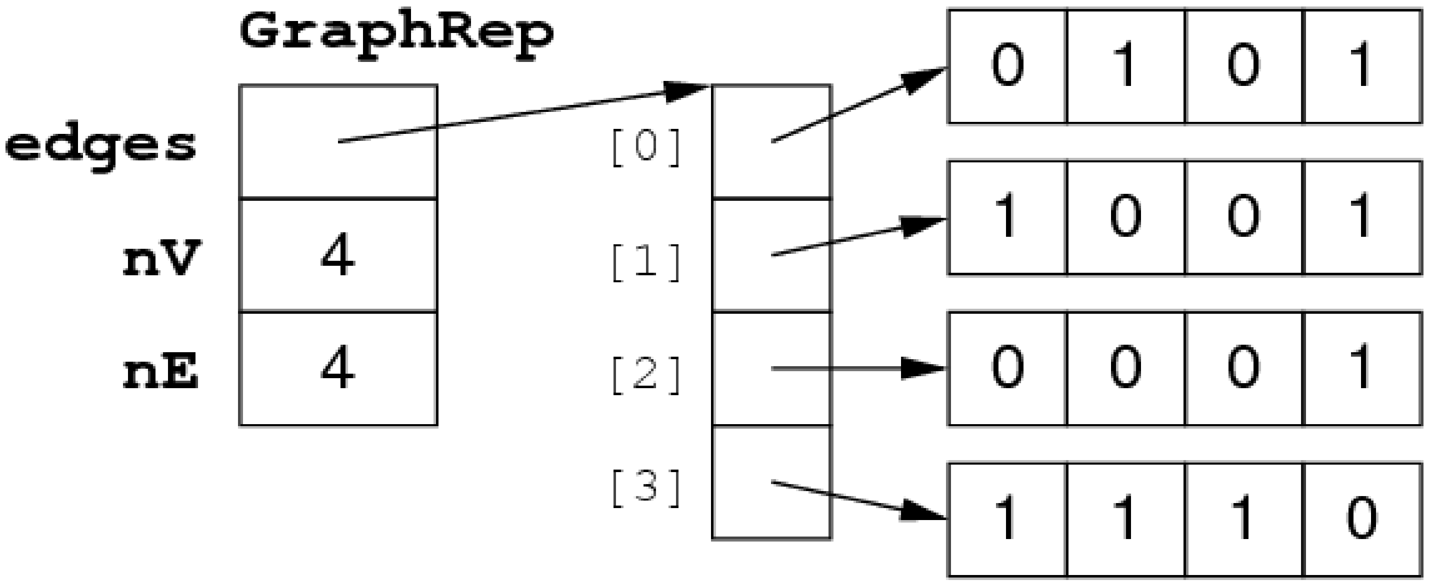
```
// graph handle
typedef struct GraphRep *Graph;

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

ADJACENCY MATRIX IMPLEMENTATION

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



Undirected graph



0	1	0	1
1	0	0	1
0	0	0	1
1	1	1	0

ADT INTERFACE OR GRAPHS

- Implementation of Graph Initialisation:

```
//Initialise a new graph
Graph newGraph(int nV) {
    int i,j;

    assert(nV >= 0);
    Graph g = malloc(sizeof(struct GraphRep));
    assert(g != NULL);

    g->edges = malloc(nV *sizeof(int *));
    for(i=0; i < nV; i++){
        g->edges[i] = malloc(nV * sizeof(int));
        for(j=0; j < nV; j++){
            g->edges[i][j] = 0;
        }
    }
    g->nV = nV;
    g->nE = 0;
    return g;
}
```

ADT INTERFACE OR GRAPHS

- 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);
```

ADT INTERFACE OR GRAPHS

- Exercise: Implement the following function

//returns the adjacent vertices of a given vertex and sets *nV to the number of adjacent vertices returned.

//O(V)

```
Vertex *  adjacentVertices(Graph g, Vertex v, int *nV);
```

Usage:

```
Graph g; Vertex v; int n;
 …
Vertex *ns = adjacentVertices(g, v, &n);
```

```
Edge *  edges (Graph g, int * nE);
Usage:
```

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

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)
- Exercise: Find the cost of the following functions
  - isAdjacent( );
  - adjacentVertices ( );
  - edges( );

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 $v < w$.

- Exercise:
  - How does the implementation of graphInit( ) change for the optimised solution?

ADT Interface or Graphs

- Exercise: Implement the following function

//return the edges in normalised/canonical form (e.v < e.w), so that each edge appears exactly once in the result array

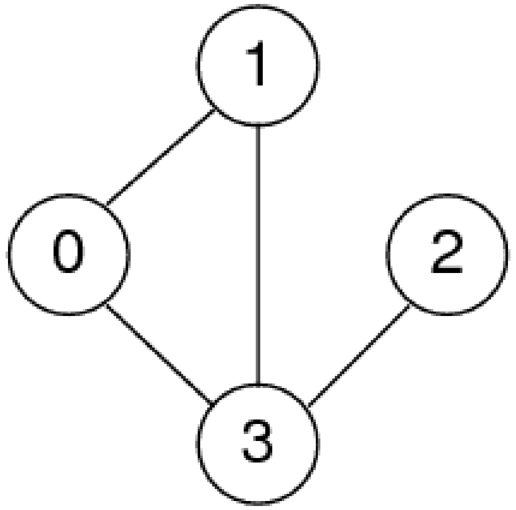
```
Edge *  edges (Graph g, int * nE);
```

Usage:

```
Graph g; int n;
 …
Edge *es = edges(g, &n);
```

ADJACENCY LIST REPRESENTATION

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



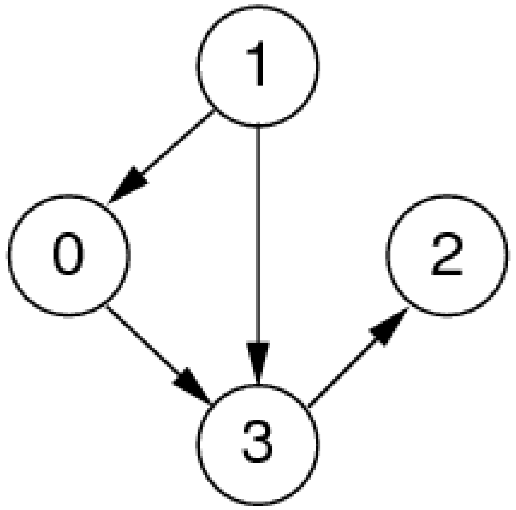
Undirected graph

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] = < >

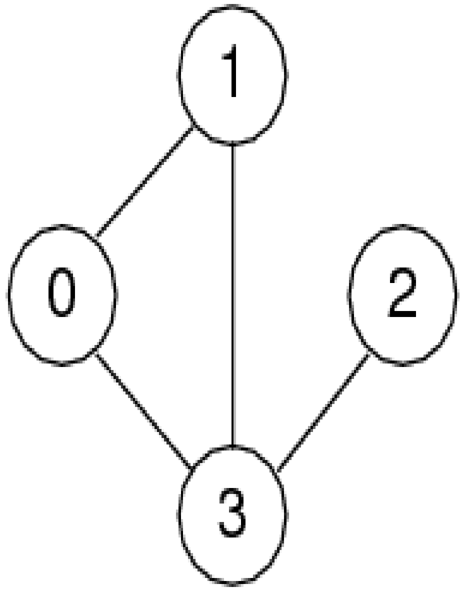
A[3] = <2>

ADJACENCY LIST REPRESENTATION

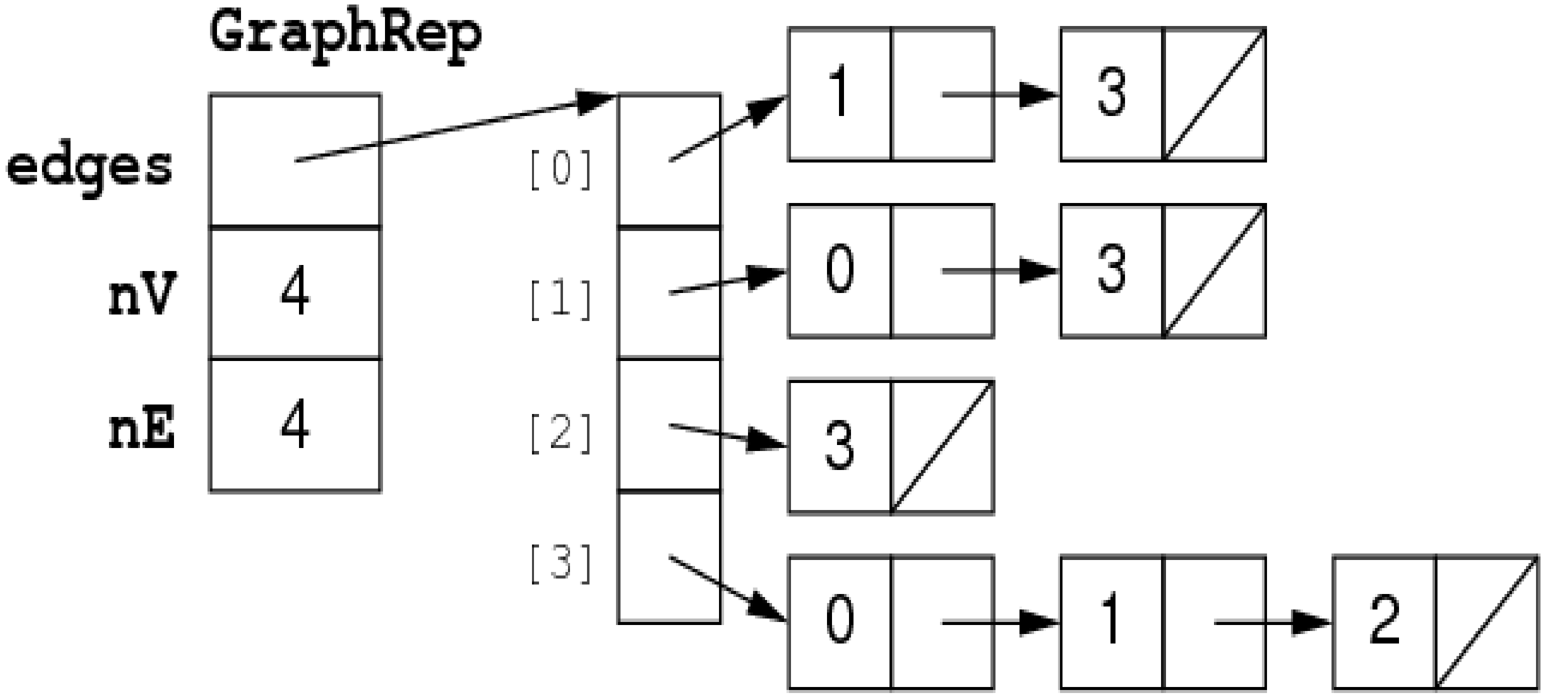
- Advantages
  - relatively easy to implement in C
  - can represent graphs and digraphs
  - memory efficient if E/V relatively small
- Disadvantages:
  - one graph has many possible representations (unless lists are ordered by same criterion e.g. ascending)

ADJACENCY LIST 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;
```



Undirected graph



ADJACENCY LIST REPRESENTATION

Creating a new graph

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

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 ⇒ $O(V)$
- If we do not want to allow parallel edges it is O(V)
- delete always requires a search, regardless of list order

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