目录

Graphs and their representations

```
Graphs are sets of vertices that are connected by edges.
```

Many problems require

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

Graph applications

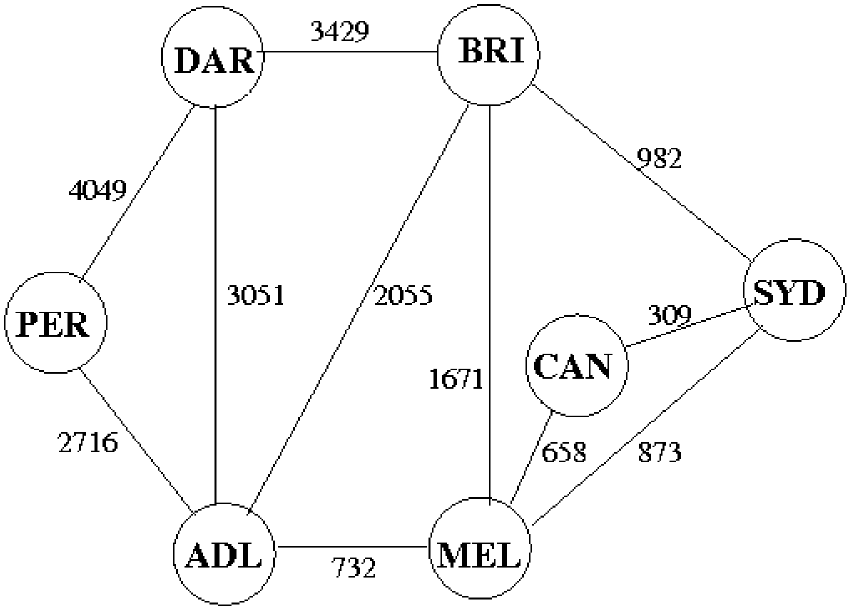
graph	vertices	edges
communication	telephones, computers	fiber optic cables
circuits	gates, registers, processors	wires
mechanical	joints	rods, beams, springs
hydraulic	reservoirs, pumping stations	pipelines
financial	stocks, currency	transactions
transportation	street intersections, airports	highways, airway routes
scheduling	tasks	precedence constraints
software systems	functions	function calls
internet	web pages	hyperlinks
games	board positions	legal moves
social relationship	people, actors	friendships, movie casts
neural networks	neurons	synapses
protein networks	proteins	protein-protein interactions
chemical compounds	molecules	bonds

3

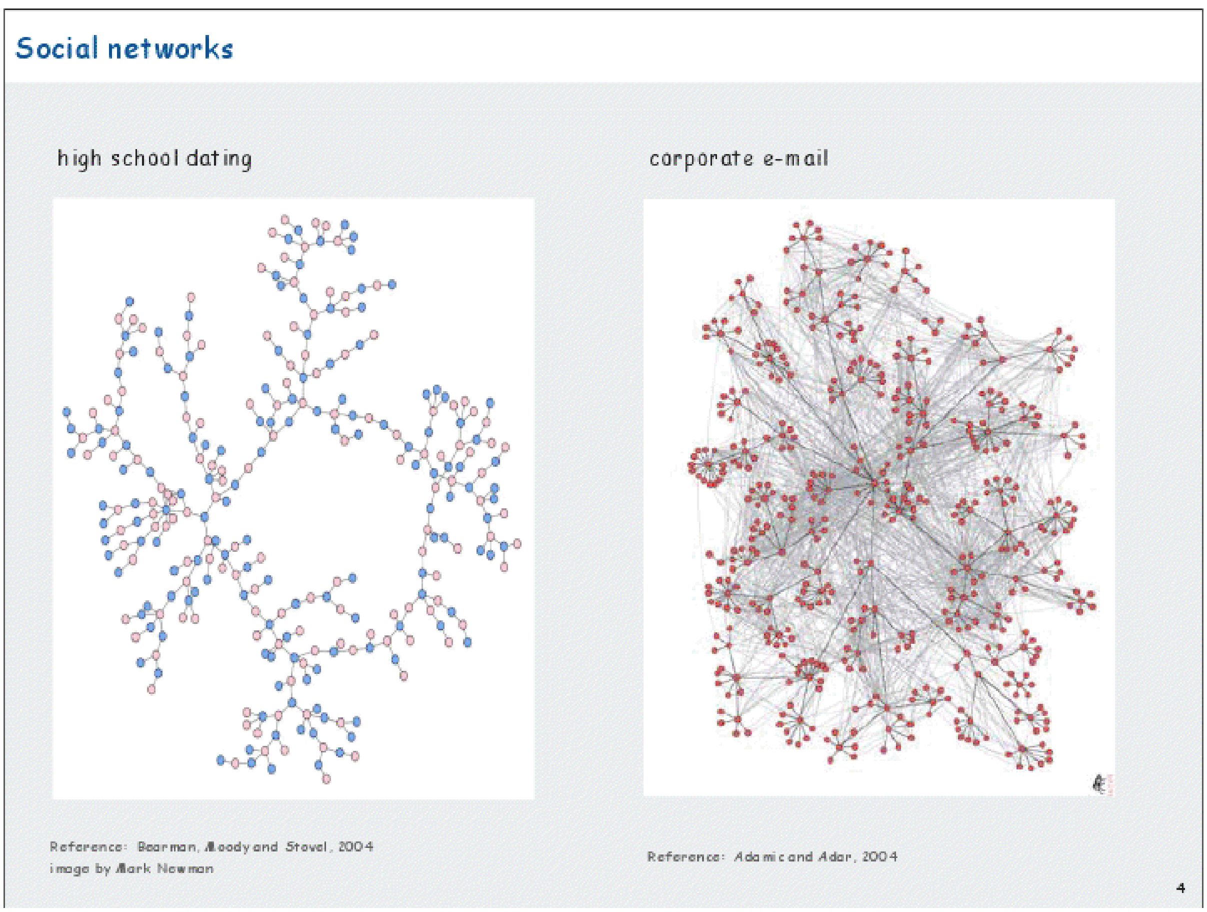
An example: road distances between Australian cities:

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	-

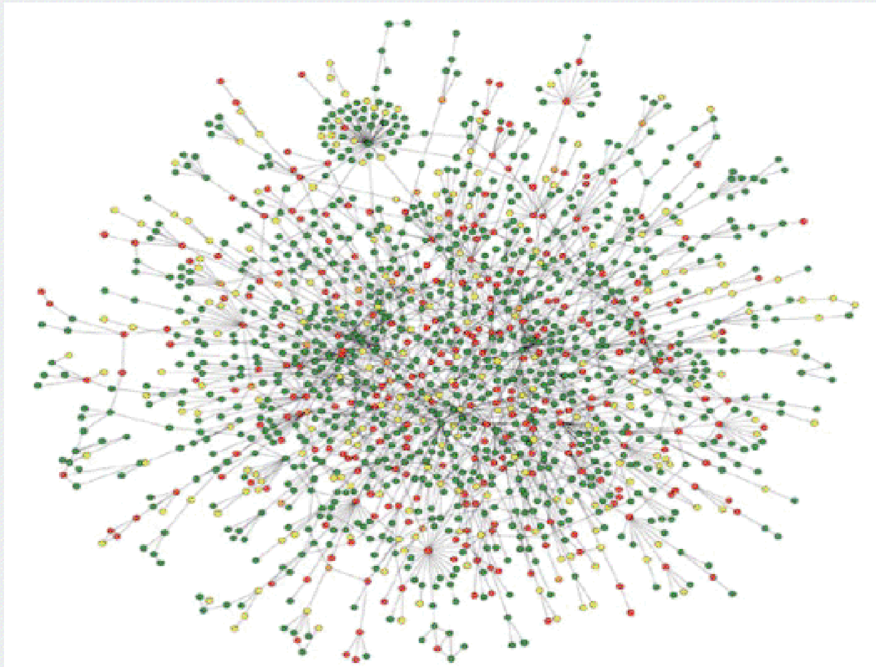
can be expressed (partially) as :



Many more interesting examples.

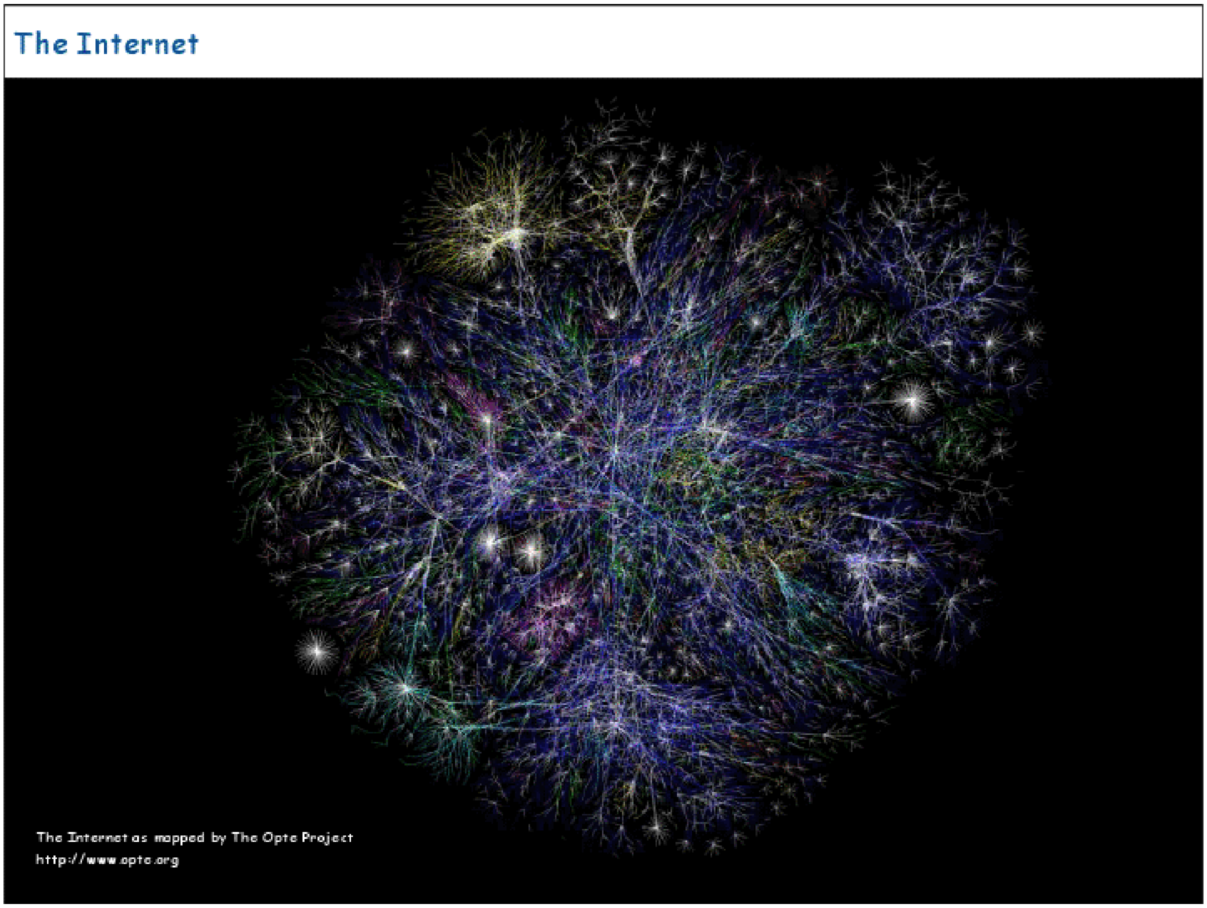


Protein interaction network



Reference: Jeong et al, Nature Review | Genetics

6

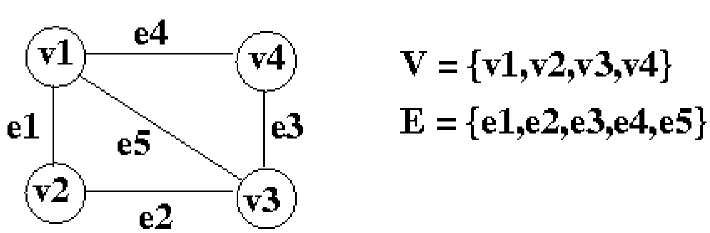


General terminology (for undirected, unweighted graphs)

- we assume graphs have **no parallel edges**
  - i.e. at most one edge connecting any two vertices
- we assume graphs have **no self loops**
  - i.e. no edges from a vertex to itself

A graph is represented by $G = (V,E)$ where:

- V is a set of vertices and
- E is a set of edges (equal to a subset of $V \times V$)

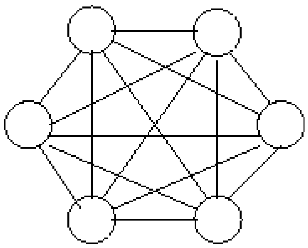


Complete graph

A graph is *complete* if:

- there is an edge from each vertex to the other V-1 vertices

  - but you double count so there are $V*(V-1)/2$ edges
  - $|E| = V(V-1)/2$



A **clique** is a complete subgraph

- a subset of vertices that form a complete graph

Sparseness/denseness of a graph

A graph with $|V|$ vertices has at most $V(V-1)/2$ edges, i.e. $|E| \leq V(V-1)/2$

- the ratio $|V|$ to $|E|$ can vary considerably
  - **dense** graphs
    - $|E|$ is closer to $V(V-1)/2$
  - **sparse** graphs
    - $|E|$ is closer to V
  - of course, a graph may be sparse but contain *cliques*

Knowing whether a graph is sparse or dense is important

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

Tree

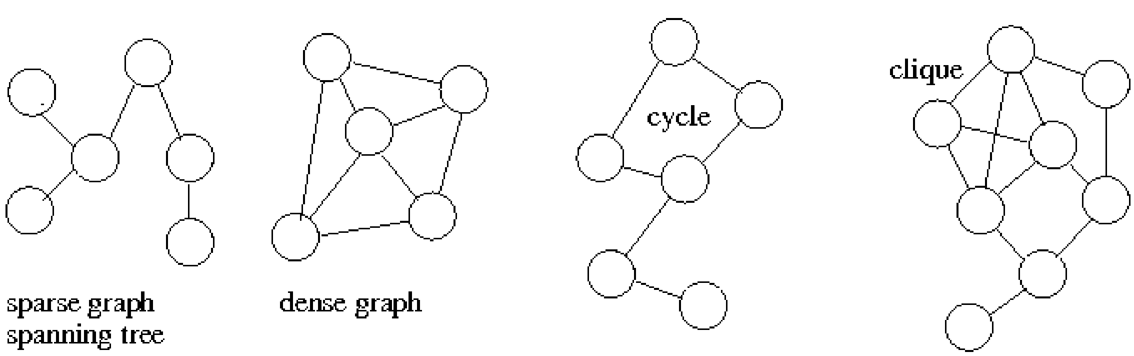
- a connected (sub)graph with no cycles

Spanning tree

- a tree that contains all vertices in the graph

Examples

The following figures are graphs:



Terminology

A **graph** consists of a:

- set of vertices V (e.g. {1, 2, 3, 4, 5})
- set of edges E, involving all vertices in V (e.g. {1-2, 2-3, 2-4, 3-5})

subgraph

- subset of edges (e.g. {1-2, 2-4}) together with
- subset of vertices involved (e.g. {1, 2, 4})

induced subgraph

- subset of vertices (e.g. {1, 2, 3, 5})
- all edges involving a pair from (e.g. {1-2, 2-3, 3-5})

cycle

- a path where the last vertex in a path is the same as the first vertex in the path

connected graph

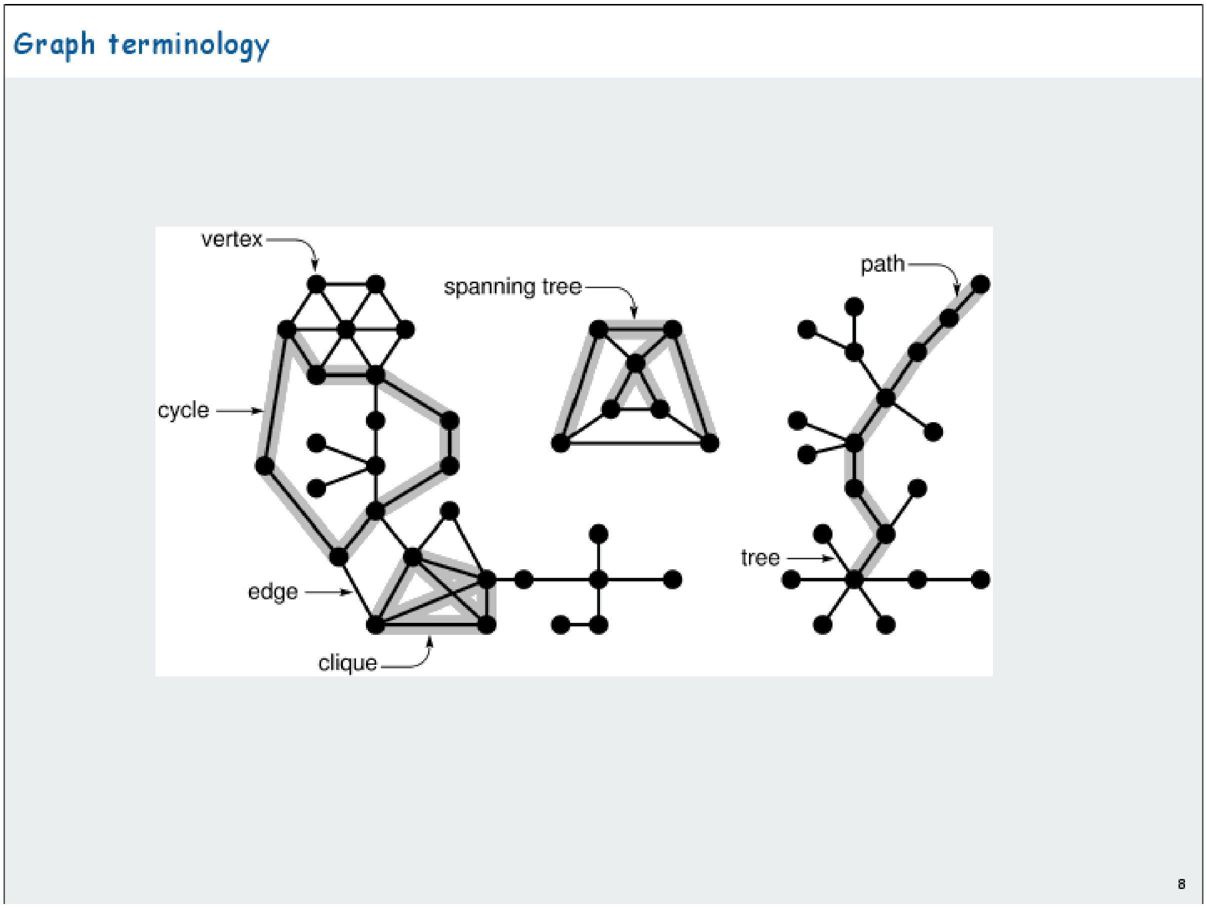
- there is a path from each vertex to every other vertex
- if a graph is not connected, it consists of a set of subgraphs each of which is connected
- if $|E| = 0$, then we have a set of vertices

adjacency

- A vertex is adjacent to a second vertex if there is an edge that connects them

degree

- The degree of a vertex is the number of edges at that vertex



Hamiltonian paths and cycles

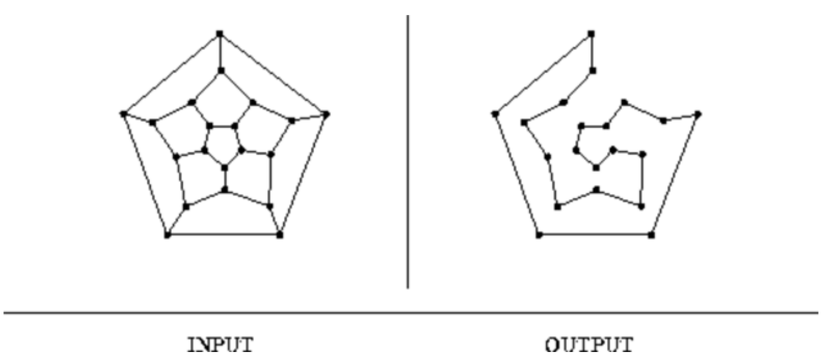
A *path* is

```
a sequence of edges joined at vertices
```

A **Hamiltonian path**: 所有顶点只用一次

- visits every vertex in the graph exactly once
- if the path starts and ends at the same vertex it is called a **Hamiltonian cycle**

Example:



- The problem of finding a Hamiltonian cycle in a graph is a special case of the *Traveling Salesman Problem*

(TSP)

```
 Given a set of cities and distance between every pair of cities, the
problem is to find the shortest possible route that visits every city
exactly once and returns to the starting point.
```

所有顶点的度都>=n/2

- So, in the TSP, we know that there are potentially many Hamiltonian tours
  - we search for the Hamiltonian cycle with minimum weight.
- In essence the difference:
  - TSP deals with weighted graphs (the route with minimum distance)
  - Hamiltonian cycles deals with unweighted graphs (is there a route?)
- For sufficiently dense graphs, there (almost) always exists at least one Hamiltonian cycle
  - there is an efficient algorithm for finding a Hamiltonian cycle if all vertices have degree >=n/2
    - More on Hamiltonian paths and tours

Eulerian path and cycles

遍历每条边一次

- an **Eulerian path** traverses each edge exactly once
- if the route starts and ends at the same vertex it is called an **Eulerian cycle**

Example: the *Konigsberg Bridge* problem

- interesting that this is the most shown example used to illustrate an Eulerian path/cycle
  - but its claim to fame is that it contains neither!
    - *crossing each bridge exactly once cannot be done*, which is why its called a problem
  - More on Eulerian paths and tours

```
A connected graph has an Eulerian cycle if all its vertices have even
degree.
```

Eulerian cycle: 所有的顶点都是偶数的度

- ... so if any vertex has odd degree, it cannot contain a Euler cycle

```
A connected graph has an Eulerian path if it has exactly 2 vertices of odd
degree, and all the rest have even degree.
```

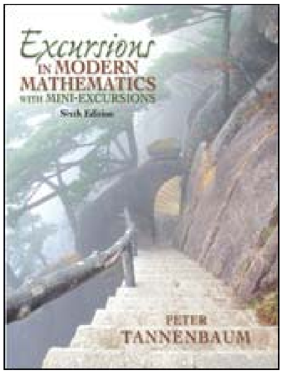
Eurlerian path：恰好有2个顶点有奇数的度

- ... so if there are more than 2 vertices of odd degree then it cannot contain a Euler path
- ... and if there is just one vertex of odd degree or more than 2 vertices then it has no Eulerian cycle or path

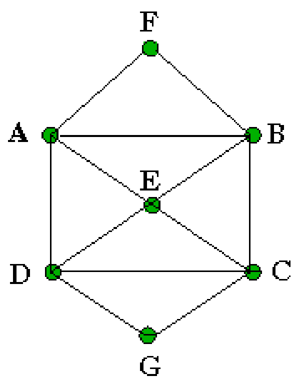
Look at the Konigsberg Bridge problem again and count the vertex degrees ...

Following figures courtesy of *Excursions in Modern Mathematics* by Peter Tannenbaum.

No Euler Path：超过2个顶点为奇数的度
一个顶点为奇数的度或者超过2个顶点为奇数的度。。。



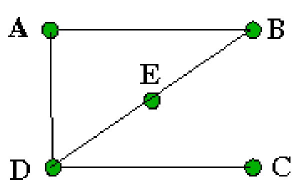
Examples of Eulerian and Hamiltonian cycles and paths



Has both Eulerian and Hamiltonian cycles

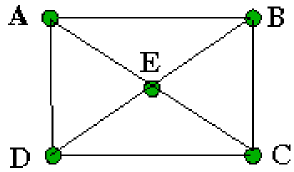
- Eulerian cycle (e.g. AFBCGDABEDCEA)
  - **not** a Euler path
- Hamiltonian cycle (e.g. AFBCGDEA)
  - **definitely** a Hamiltonian path (truncate off the last vertex!)
- note the difference:
  - Euler cycles and paths are mutually exclusive
    - a graph **cannot have both an Eulerian path and cycle**
  - in contrast, a **Hamiltonian cycle means a graph must have a Hamiltonian path**
    - generate a Ham. path from a Ham. cycle by truncating the last node
    - but **a Hamiltonian cycle cannot always be generated from a Hamiltonian path**

总结：
1. 不可能同时有欧拉path或者cycle
2. 有Hamiltonian cycle一定有path
3. 判断H 回路 的方法就是所有顶点的度都≥n/2
4. 判断欧拉path：只有两个是奇数的度
5. 判断欧拉cycle：所有都为偶数的度



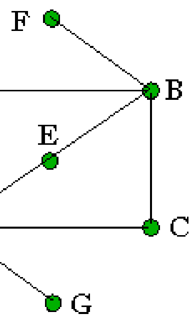
Has both paths

- contains
  - Eulerian path (and hence no Eulerian cycle)
  - Hamiltonian path (e.g. ABEDC)
    - but no Hamiltonian cycle (C is a deadend)



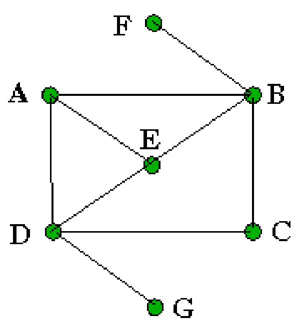
Hamiltonians but no Eulers

- contains no Eulerian path or cycle
  - look at the degrees of the vertices
- does contain a Hamiltonian cycle (e.g. ABCDEA)
  - and Hamiltonian path (e.g. ABCDE)



Euler (path) but no Hamiltonians

- contains a Eulerian path (e.g. GDABCDEBF) (and hence no Eulerian cycle)
- it does not contain anything Hamiltonian



Neither Eulers nor Hamiltonians

- contains no Eulerian path or cycle
- contains no Hamiltonian path or cycle

Conclusions:

- Knowing whether a graph contains an Eulerian path/cycle tells us nothing about its Hamiltonians
- Knowing whether a graph contains a Hamiltonian path/cycle tells us nothing about its Eulerians
- There is a theorem that states whether an arbitrary graph has an Eulerian path, or cycle, or neither
- There is *no* theorem for Hamiltonians

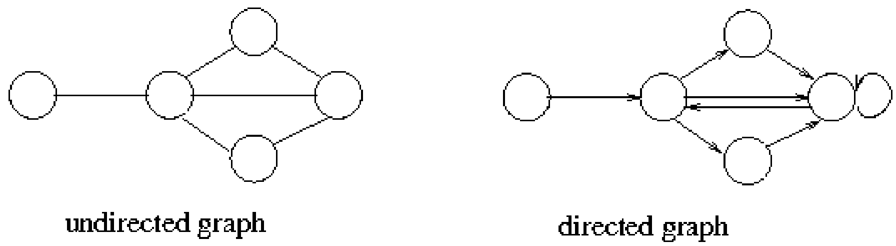
Undirected vs Directed Graphs

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))




```
Unless stated otherwise, we assume graphs are undirected.
```

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. may be able to get to new location by bus or train or ferry etc…

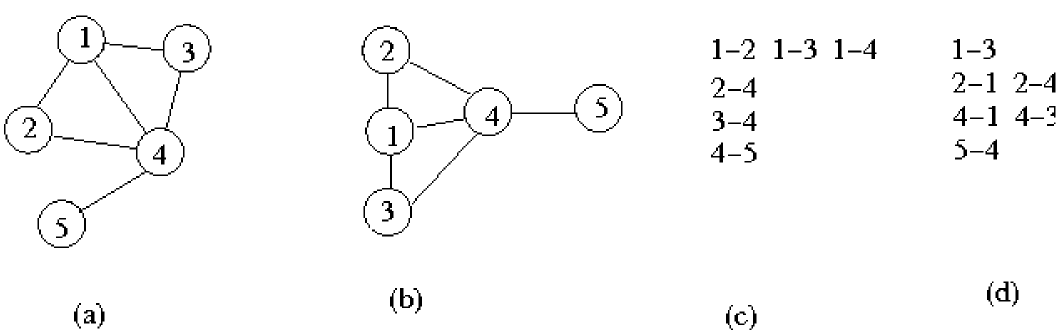
Implementing Graphs

Need some way of identifying vertices

- could give diagram showing edges and vertices
- could give a list of edges

Here are 4 representations of the same graph

- *are they the same?*



... that depends on the implementation

Graph ADT

Graphs consist of vertices and edges. These are represented by 3 data structures:

- **Vertex** that is represented by an *int*
- **Edge** that is represented by 2 vertices
- **Graph** that is represented by an **Adjacency matrix**, or as an **Adjacency list**.

The operations we will define are:

- building:

    - create a graph
    - create an edge
    - add an edge to a graph
- deleting
    - remove an edge from a graph
    - remove and free a graph
- printing
    - 'show' a graph

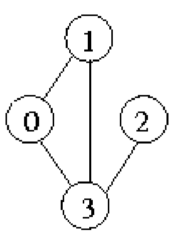
切换行号显示

```
// Graph.h: ADT interface for undirected/unweighted graphs

typedef int Vertex;                // define a VERTEX

typedef struct {                   // define an EDGE
  Vertex v;
  Vertex w;
} Edge;

typedef struct graphRep *Graph;    // define a GRAPH

Graph newGraph(int);               // create a new graph
void freeGraph(Graph);             // free the graph mallocs
void showGraph(Graph);             // print the graph

Edge newE(Vertex, Vertex);         // create a new edge
void insertE(Graph, Edge);         // insert an edge
void removeE(Graph, Edge);         // remove an edge
void showE(Edge);                  // print an edge
int isEdge(Graph, Edge);           // check edge exists

```

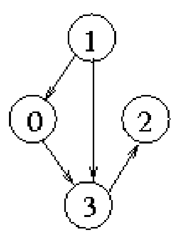
Adjacency Matrix Representation

Edges are represented by a *VxV* Boolean matrix, where *V* is the number of vertices.

Example:



undirected



directed

are represented as:

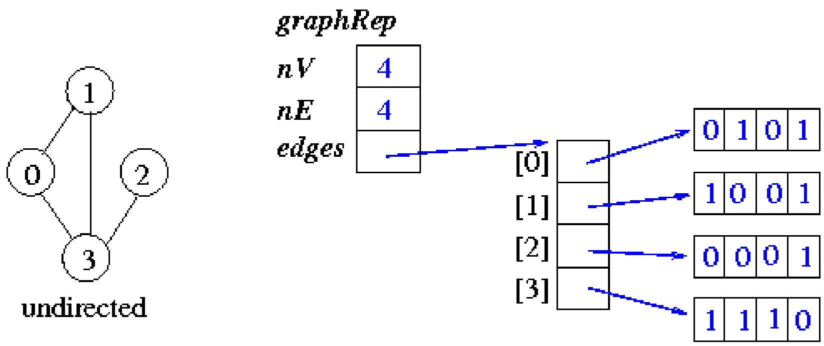
Undirected (note the symmetry)

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 (note the asymmetry)

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

Implementation



切换行号显示

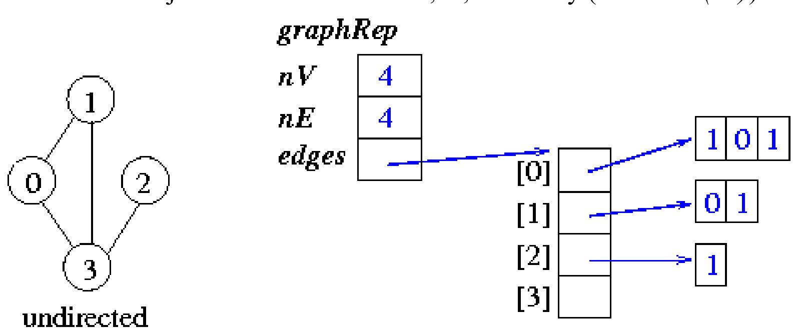
```
// GraphAM.c: an adjacency matrix implementation
#include <stdio.h>
#include <stdlib.h>
#include "Graph.h"

struct graphRep {
    int nV;       // #vertices
    int nE;       // #edges
    int **edges;  // matrix of Booleans ... THIS IS THE ADJACENCY MATRIX
};

Graph newGraph(int numVertices) {
    Graph g = NULL;
    if (numVertices < 0) {
       fprintf(stderr, "newgraph: invalid number of vertices\n");
    }
    else {
        g = malloc(sizeof(struct graphRep));
        if (g == NULL) {
            fprintf(stderr, "newGraph: out of memory\n");
            exit(1);
        }
        g->edges = malloc(numVertices * sizeof(int *));
        if (g->edges == NULL) {
            fprintf(stderr, "newGraph: out of memory\n");
            exit(1);
        }
        int v;
        for (v = 0; v < numVertices; v++) {
            g->edges[v] = malloc(numVertices * sizeof(int));
            if (g->edges[v] == NULL) {
                fprintf(stderr, "newGraph: out of memory\n");
                exit(1);
            }
            for (int j = 0; j < numVertices; j++) {
                g->edges[v][j] = 0;
```

```
            }
        }
        g->nV = numVertices;
        g->nE = 0;
    }
    return g;
}

void freeGraph(Graph g) {
// not shown
}

void showGraph(Graph g) { // print a graph
    if (g == NULL) {
        printf("NULL graph\n");
    }
    else {
        printf("V=%d, E=%d\n", g->nV, g->nE);
        int i;
        for (i = 0; i < g->nV; i++) {
            int nshown = 0;
            int j;
            for (j = 0; j < g->nV; j++) {
                if (g->edges[i][j] != 0) {
                    printf("%d-%d ", i, j);
                    nshown++;
                }
            }
            if (nshown > 0) {
                printf("\n");
            }
        }
    }
    return;
}

static int validV(Graph g, Vertex v) { // checks if v is in graph
    return (v >= 0 && v < g->nV);
}

Edge newE(Vertex v, Vertex w) { // create an edge from v to w
    Edge e = {v, w};
    return e;
}
void showE(Edge e) { // print an edge
    printf("%d-%d", e.v, e.w);
    return;
}

int isEdge(Graph g, Edge e) { // return 1 if edge found, otherwise 0
// not shown
}

void insertE(Graph g, Edge e) { // insert an edge into a graph
   if (g == NULL) {
      fprintf(stderr, "insertE: graph not initialised\n");
   }
   else {
       if (!validV(g, e.v) || !validV(g, e.w)) {
          fprintf(stderr, "insertE: invalid vertices %d-%d\n", e.v, e.w);
       }
```

```
        else {
           if (isEdge(g, e) == 0) { // increment nE only if it is new
              g->nE++;
           }
           g->edges[e.v][e.w] = 1;
           g->edges[e.w][e.v] = 1;
        }
    }
    return;
}

void removeE(Graph g, Edge e) { // remove an edge from a graph
    if (g == NULL) {
        fprintf(stderr, "removeE: graph not initialised\n");
    }
    else {
        if (!validV(g, e.v) || !validV(g, e.w)) {
            fprintf(stderr, "removeE: invalid vertices\n");
        }
        else {
            if (isEdge(g, e) == 1) {   // is edge there?
                g->edges[e.v][e.w] = 0;
                g->edges[e.w][e.v] = 0;
                g->nE--;
            }
        }
    }
    return;
}
```

Adjacency-matrix implementation of an undirected graph:

- to store a graph we need V integer pointers plus V^2 integers
  - if the graph is sparse, most storage is wasted (V^2 array space is reserved no matter what)
  - it is $O(V^2)$ even to initialise!
- could store only top-right part of matrix (remember it is symmetric)
  - vertex i has stored adjacencies to vertices $i+1$, ..., $nV-1$ only (but still $O(V^2)$)



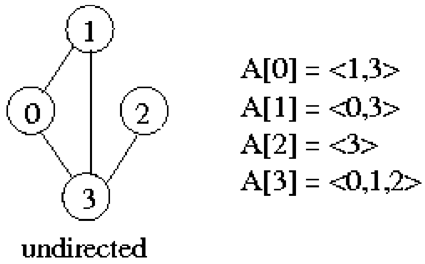
  - the implementation above does not do this

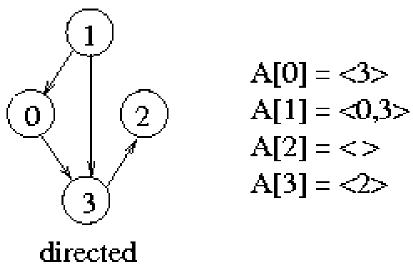
Adjacency List Representation

Adjacent vertices are stored in a linked list for each vertex.

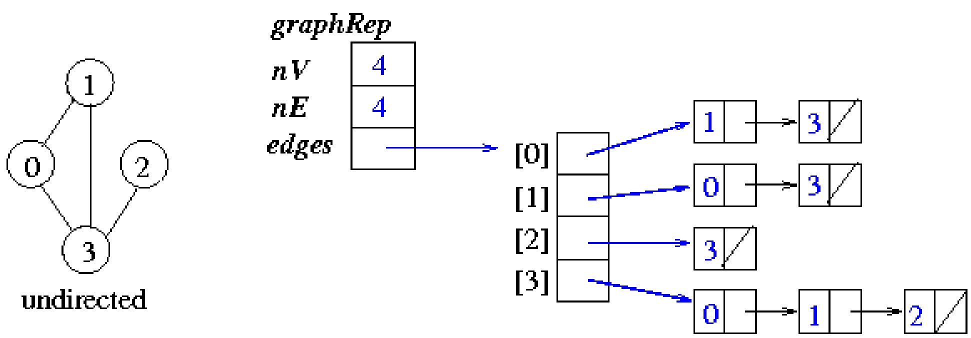
- space will be proportional to the number of vertices plus number of edges
- used if a graph is sparse.

Example:





Implementation



切换行号显示

```c
// GraphAL.c: an adjacency list implementation
#include <stdio.h>
#include <stdlib.h>
#include "Graph.h"

typedef struct node *list;
struct node {
  Vertex name;
  list next;
};

struct graphRep {
  int nV;     // #vertices
  int nE;     // #edges
  list *edges; // array of linked lists ... THIS IS THE ADJACENCY LIST
};

Graph newGraph(int numVertices) {
  Graph g = NULL;
  if (numVertices < 0) {
     fprintf(stderr, "newgraph: invalid number of vertices\n");
  }
  else {
     g = malloc(sizeof(struct graphRep));
     if (g == NULL) {
        fprintf(stderr, "newGraph: out of memory\n");
```

```
        exit(1);
     }
     g->edges = malloc(numVertices * sizeof(int *));
     if (g->edges == NULL) {
        fprintf(stderr, "newGraph: out of memory\n");
        exit(1);
     }
     int v;
     for (v = 0; v < numVertices; v++) {
       g->edges[v] = NULL;
     }
     g->nV = numVertices;
     g->nE = 0;
  }
  return g;
}

void freeGraph(Graph g) {
// not shown
}

void showGraph(Graph g) { // print a graph
    if (g == NULL) {
        printf("NULL graph\n");
    }
    else {
        printf("V=%d, E=%d\n", g->nV, g->nE);
        int i;
        for (i = 0; i < g->nV; i++) {
            int nshown = 0;
            list listV = g->edges[i];
            while (listV != NULL) {
               printf("%d-%d ", i, listV->name);
               nshown++;
               listV = listV->next;
            }
            if (nshown > 0) {
                printf("\n");
            }
        }
    }
    return;
}

static int validV(Graph g, Vertex v) { // checks if v is in graph
    return (v >= 0 && v < g->nV);
}

Edge newE(Vertex v, Vertex w) {
  Edge e = {v, w};
  return e;
}

void showE(Edge e) { // print an edge
    printf("%d-%d", e.v, e.w);
    return;
}

int isEdge(Graph g, Edge e) { // return 1 if edge found, otherwise 0
// not shown
}

```

```c
void insertE(Graph g, Edge e){
  if (g == NULL) {
     fprintf(stderr, "insertE: graph not initialised\n");
  }
  else {
     if (!validV(g, e.v) || !validV(g, e.w)) {
        fprintf(stderr, "insertE: invalid vertices %d-%d\n", e.v, e.w);
     }
     else {
        if (isEdge(g, e) == 0) {
           list newnodev = malloc(sizeof(struct node));
           list newnodew = malloc(sizeof(struct node));
           if (newnode1 == NULL || newnode2 == NULL) {
              fprintf(stderr, "Out of memory\n");
              exit(1);
           }
           newnodev->name = e.w;                    // put in the data
           newnodev->next = g->edges[e.v];          // link to the existing list attached to e.v
           g->edges[e.v] = newnodev;                // link e.v to new node
           newnodew->name = e.v;
           newnodew->next = g->edges[e.w];
           g->edges[e.w] = newnodew;
           g->nE++;         两个顶点都要添加，添加到头部位置
        }
     }
  }
  return;
}

void removeE(Graph g, Edge e) {
// not shown
}
```

Implementation Comparison

Adjacency-list implementation:

- efficient storage proportional to $V+E$ instead of V^2 for adjacency matrix
- it comes at a cost:
  - *removeE(Graph, Edge)* is not shown but requires searching the linked lists, with complexity V

Property	Adjacency Matrix	Adjacency List
Space	V^2	V+E
Create	V^2	V
Insert edge (at head)	1	1
Find/remove edge	1	V

Also

- parallel edge detection (weighted graphs) requires V complexity
- order of edges in the linked lists is not defined
  - this can be a problem in applications where order is important

Graph clients

Reading a graph

Assume that a text representation of a graph is the following:

```
5
0-1 0-2 1-2 1-3 1-4 2-3 2-4
```

We can read this 'graph' using the client:

切换行号显示

```
// readGraph.c  read a graph
#include <stdio.h>
#include <stdlib.h>
#include "Graph.h"

int readGraph(Graph g) {
   int readokay = 1;
   int v1;
   int i;
   for (i=0; scanf("%d", &v1) != EOF && readokay; i++) { // read first vertex
      getchar();                          // skip over the separator
      int v2;
      if (scanf("%d", &v2) != 1){         // read second vertex
         printf("Missing vertex in edge\n");
         readokay = 0;
      }
      else {
         insertE(g, newE(v1, v2));
      }
   }
   return readokay;
}

int main (int argc, char *argv[]) {
   Graph g;
   int numV;
   if (scanf("%d", &numV) == 1 && numV>=0) {         // read #vertices
      g = newGraph(numV);
      if (readGraph(g) == 1) {
         showGraph(g);
      }
      freeGraph(g);
   }
   return EXIT_SUCCESS;
}
```

Compile and execute:

```
prompt$ dcc GraphAM.c readGraph.c
prompt$ ./a.out < graph1.txt
V=5, E=0
0-1 0-2
1-0 1-2 1-3 1-4
2-0 2-1 2-3 2-4
3-1 3-2
```

```
4-1 4-2
```

Graphs 1 (2019-07-15 18:07:04由AlbertNymeyer编辑)