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

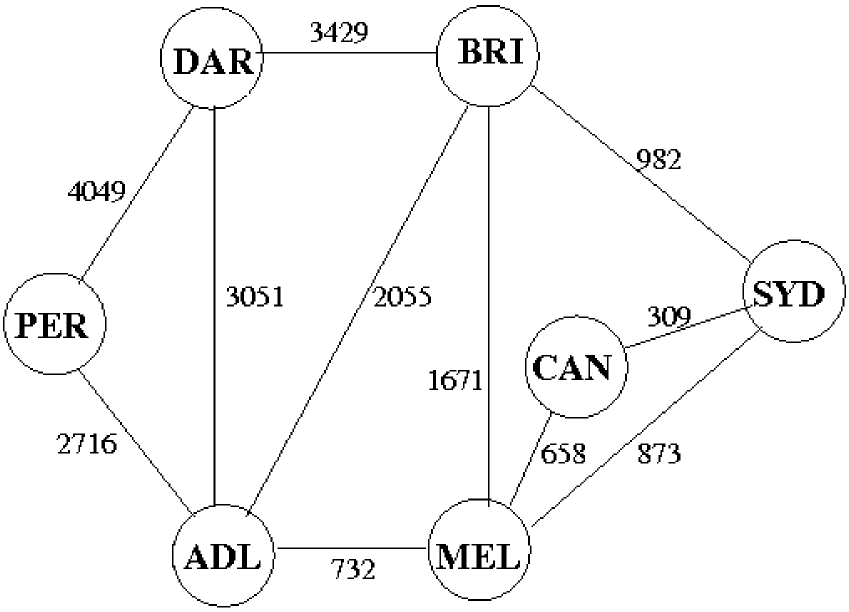
graph	vertices	edges
communication	telephones, computers	fiber optic cables
circuits	gates, registers, processors	wires
mechanical	joints	rods, beams, springs
hydraulic	reservoirs, pumping stations	pipelines
financial	stocks, currency	transactions
transportation	street intersections, airports	highways, 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	-

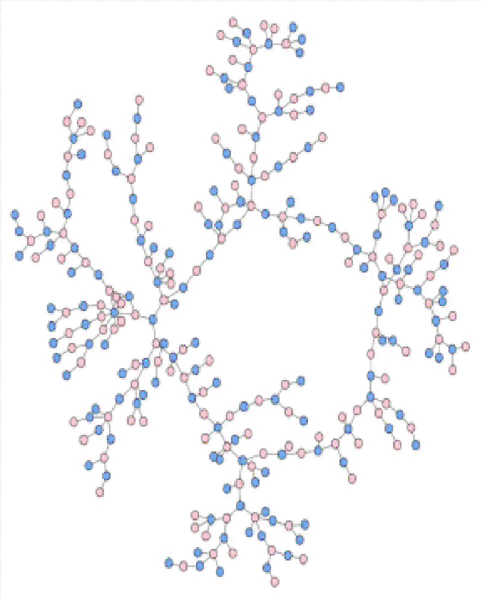
can be expressed (partially) as :



Many more interesting examples.

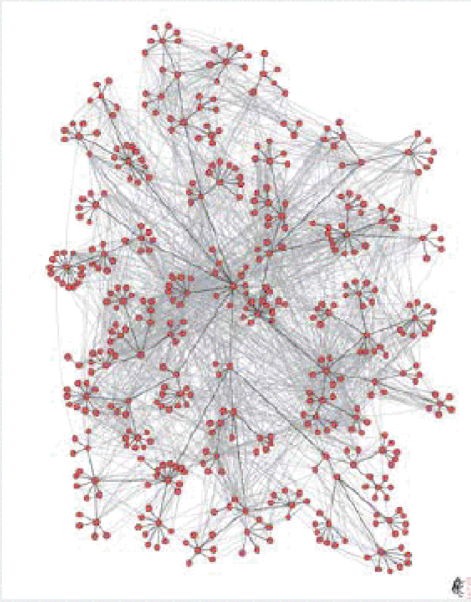
Social networks

high school dating



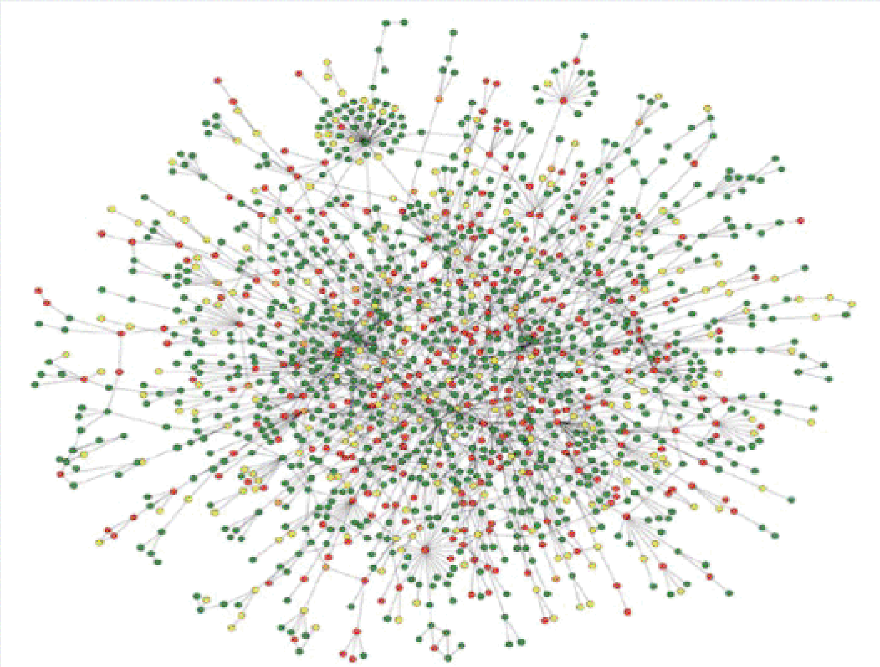
Reference: Bearman, Moody and Stovel, 2004
image by Mark Newman

corporate e-mail



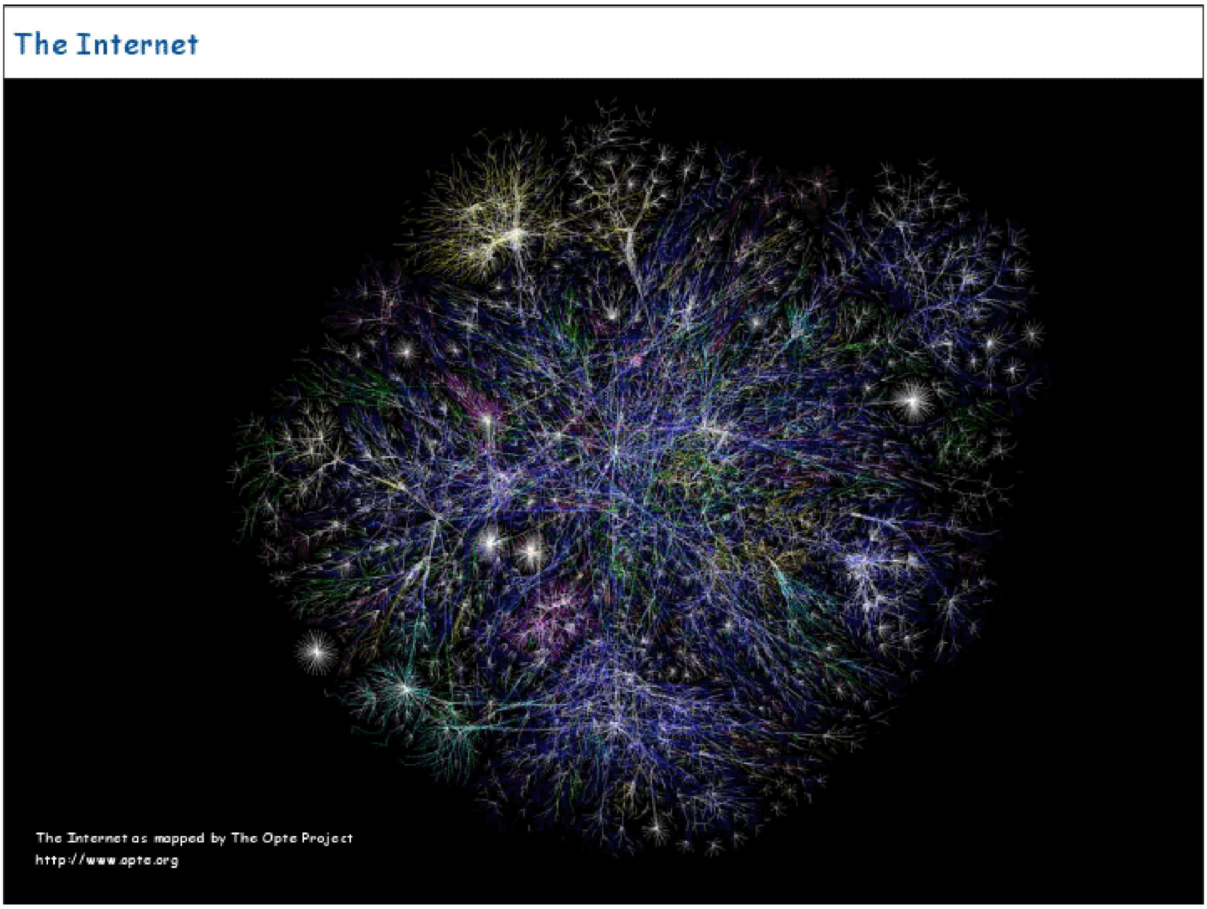
Reference: Adamic and Adar, 2004

Protein interaction network



Reference: Jeong et al, Nature Review | Genetics

6

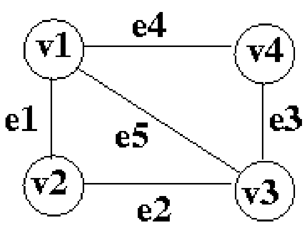


General terminology (for undirected, unweighted graphs)

- 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 $V \times V$)

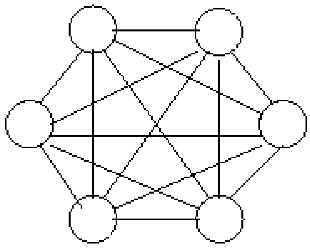


V = {v1,v2,v3,v4}
E = {e1,e2,e3,e4,e5}

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*| ≤ *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

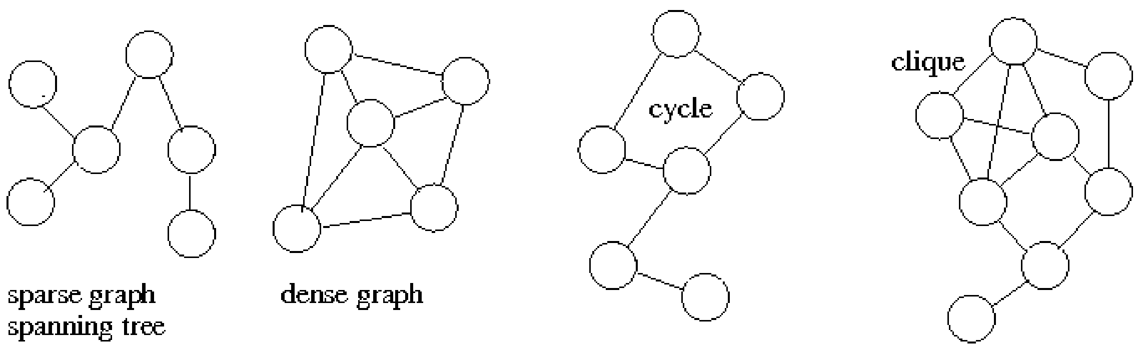
- a connected (sub)graph with no cycles

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

disconnected graph

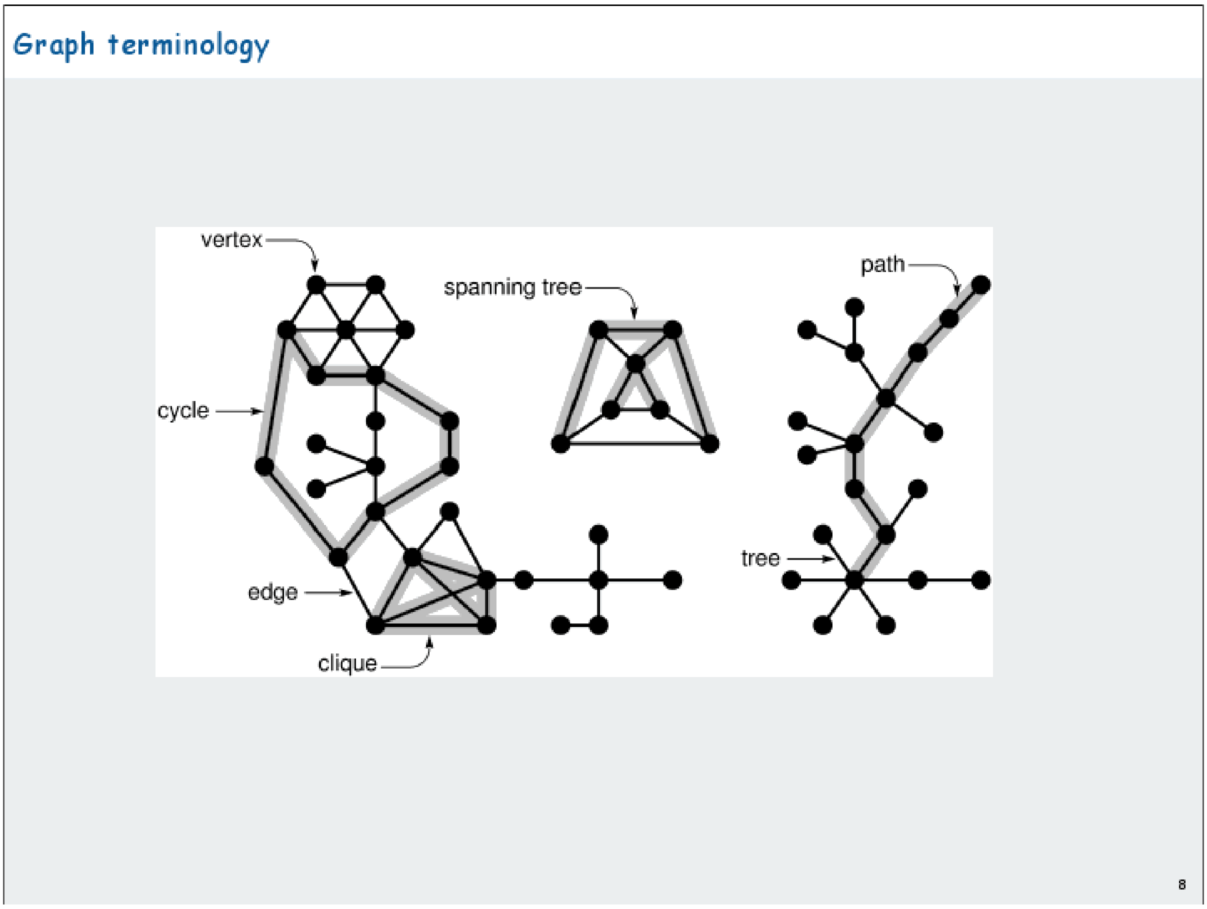
- consists of a set of subgraphs each of which is connected
- if there are no edges, i.e. $|E| = 0$, then the disconnected graph is a set of vertices

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

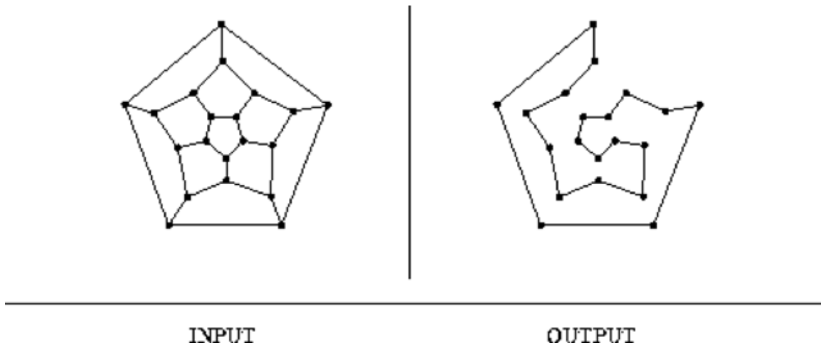
```
a sequence of edges joined at vertices
```

A **Hamiltonian path**:

- 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.
```

- 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

- 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 often 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.
```

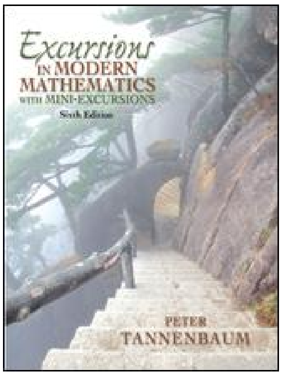
- ... so if any vertex has odd degree, it cannot contain a Euler cycle

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

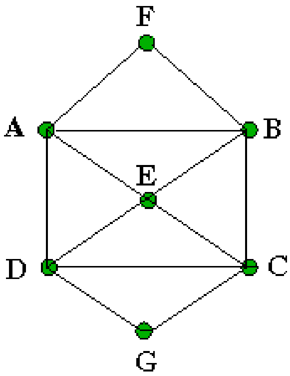
- ... so if there are more than 2 vertices of odd degree then it cannot contain a Euler path
- ... and if there is just one vertex of odd degree or more than 2 vertices then it has no Eulerian cycle or path

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

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

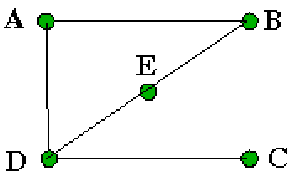


Examples of Eulerian and Hamiltonian cycles and paths



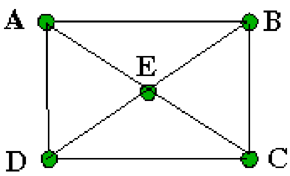
Has both Eulerian and Hamiltonian cycles

- 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**



Has both paths

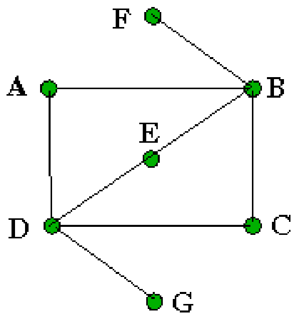
- contains
  - Eulerian path (and hence no Eulerian cycle)
  - Hamiltonian path (e.g. ABEDC)
    - but no Hamiltonian cycle (C is a deadend)



Hamiltonians but no Eulers

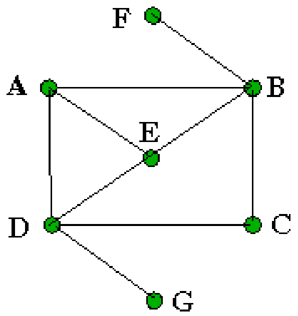
- contains no Eulerian path or cycle

    - 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

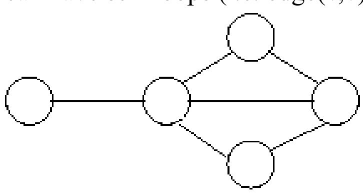
Undirected vs Directed Graphs

Undirected graph:

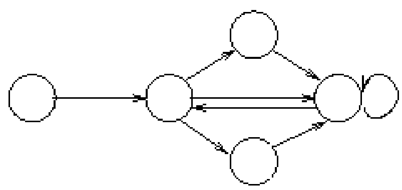
- edge(u,v) = edge(v,u)
- no self-loops (i.e. no edge(v,v))

Directed graph:

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



undirected graph



directed graph

```
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…

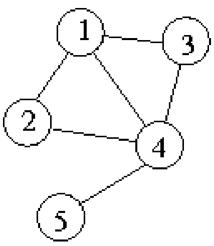
Implementing Graphs

Need some way of identifying vertices

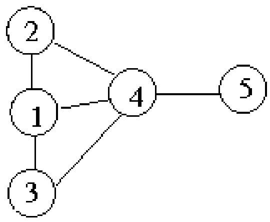
- could give diagram showing edges and vertices
- could give a list of edges

Here are 4 representations of the same graph

- *are they the same?*



(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)

... 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

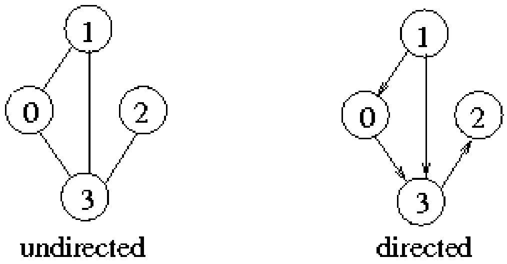
```
// Graph.h: ADT interface for undirected/unweighted graphs

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
Graph freeGraph(Graph);            // free the graph mallocs
void showGraph(Graph);             // print the graph
```

```
Edge newEdge(Vertex, Vertex);      // create a new edge
void insertEdge(Edge, Graph);      // insert an edge
void removeEdge(Edge, Graph);      // remove an edge
void showEdge(Edge);               // print an edge
int isEdge(Edge, Graph);           // check edge exists
```

GraphAM.c: A Graph ADT based on an Adjacency Matrix

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

Example:



are represented as:

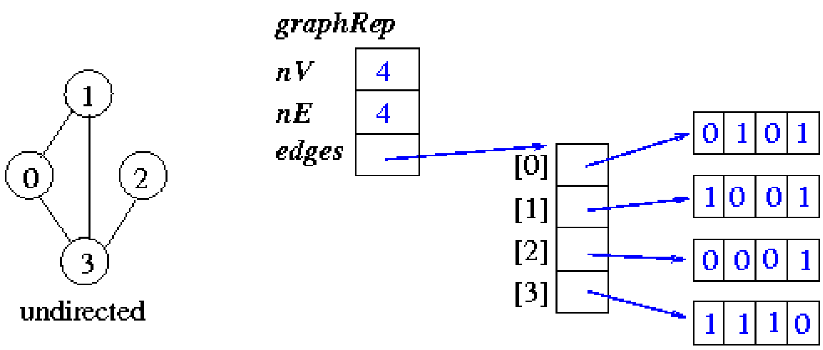
Undirected (note the symmetry)

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

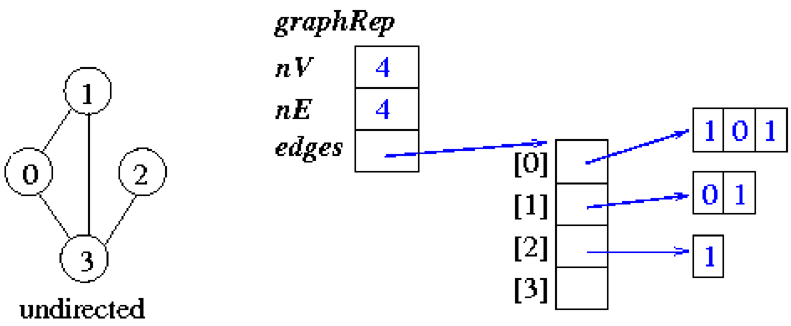
```
// GraphAM.c: an adjacency matrix implementation

#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;
}

Graph freeGraph(Graph g) {

   // code not shown

   return g;
}

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);
        for (int i = 0; i < g->nV; i++) {
            int nshown = 0;
            for (int 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 newEdge(Vertex v, Vertex w) { // create an edge from v to w
    Edge e = {v, w};
    return e;
}
void showEdge(Edge e) { // print an edge
    printf("<%d %d>", e.v, e.w);
    return;
}

int isEdge(Edge e, Graph g) { // return 1 if edge found, otherwise 0
   int found = 0;

   // code not shown
```

```

   return found;
}

void insertEdge(Edge e, Graph g) { // 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, "insertEdge: invalid vertices <%d %d>\n", e.v, e.w);
       }
       else {
          if (isEdge(e, g) == 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 removeEdge(Edge e, Graph g) { // remove an edge from a graph
    if (g == NULL) {
        fprintf(stderr, "removeEdge: graph not initialised\n");
    }
    else {
        if (!validV(g, e.v) || !validV(g, e.w)) {
            fprintf(stderr, "removeE: invalid vertices\n");
        }
        else {
            if (isEdge(e, g) == 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 even $O(V^2)$ just 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 $O(V^2)$)



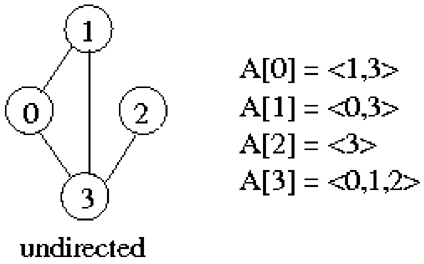
  - the implementation above does not do this

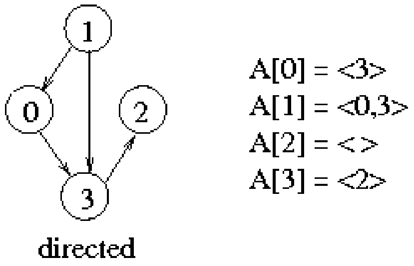
GraphAL.c: A Graph ADT based on an Adjacency Linked List

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

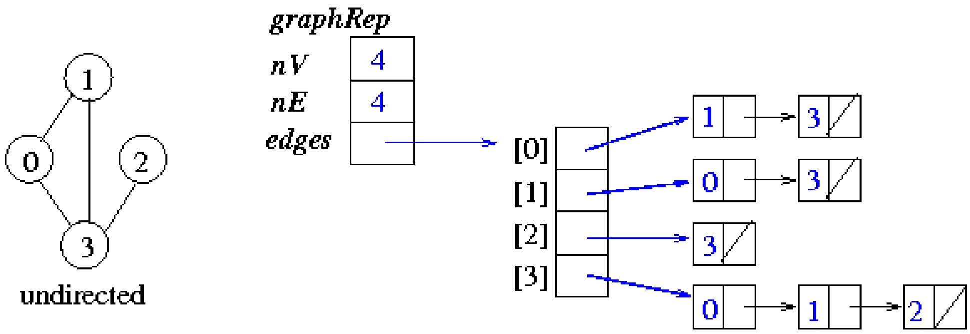
- space will be proportional to the number of vertices plus number of edges
- used if a graph is sparse.

Example:





Implementation



切换行号显示

```
// 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(List));
     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;
```

```
}

Graph freeGraph(Graph g) {

   // code not shown

   return g;
}

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 vx = g->edges[i];
            while (vx != NULL) {
               printf("<%d %d> ", i, vx->name);
               nshown++;
               vx = vx->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 newEdge(Vertex v, Vertex w) {
  Edge e = {v, w};
  return e;
}

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

int isEdge(Edge e, Graph g) {
// a linear search for edge 'e': return 1 if edge found, 0 otherwise
   int found = 0;

   // code not shown

   return found;
}

void insertEdge(Edge e, Graph g){ // edge is e.v---e.w
  if (g == NULL) {
     fprintf(stderr, "insertE: graph not initialised\n");
  }
  else {
     if (!validV(g, e.v) || !validV(g, e.w)) {
        fprintf(stderr, "insertEdge: invalid vertices <%d %d>\n", e.v, e.w);
     }
     else {
        if (isEdge(e, g) == 0) {
           List n1 = malloc(sizeof(struct node));
           List n2 = malloc(sizeof(struct node));
           if (n1 == NULL || n2 == NULL) {
              fprintf(stderr, "Out of memory\n");
              exit(1);
           }
           n1->name = e.w;           // node contains w
           n1->next = g->edges[e.v]; // node's next is v's linked list
           g->edges[e.v] = n1;       // node is new head for v

           n2->name = e.v;
           n2->next = g->edges[e.w];
           g->edges[e.w] = n2;

           g->nE++;
        }
    }
  }
```

```
125   return;
126 }
127
128 static int removeV(Graph g, Vertex v, Vertex w) { // return 1 if found&removed
129    int success = 0;
130
131    // code not shown
132
133    return success;
134 }
135
136 void removeEdge(Edge e, Graph g) {
137   if (g == NULL) {
138      fprintf(stderr, "removeEdge: graph not initialised\n");
139   }
140   else {
141      if (!validV(g, e.v) || !validV(g, e.w)) {
142         fprintf(stderr, "removeEdge: invalid vertices %d-%d\n", e.v, e.w);
143      }
144      else {
145         if (removeV(g, e.w, e.v) == 1) { // remove v from w's list
146            g->nE--;                      // decrement nE if an edge is removed
147         }
148         removeV(g, e.v, e.w);            // remove w from v's list
149      }
150   }
151   return;
152 }
```

Crucial in understanding how this list version works is to consider the function *insertE()*

- after some checking (lines 98 - 105) ...
- in line 106 we do a check ... *WHY?*
    - two nodes are mallocd: *n1* and *n2* ... *WHY TWO??*

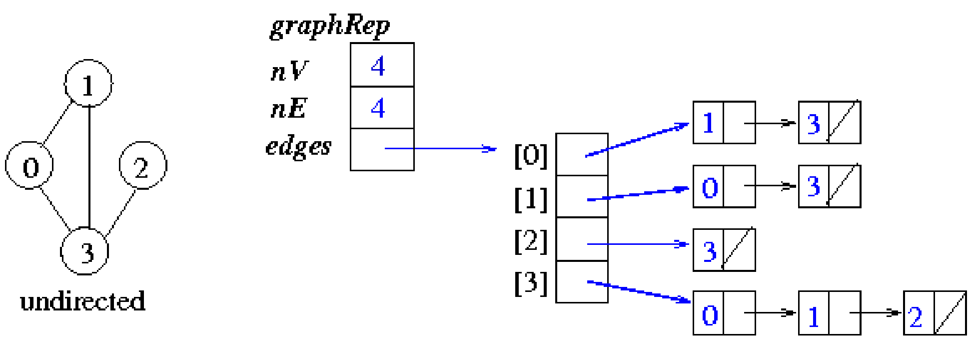
Lines 113 - 119 are the 'pumping heart' of the adjacency list approach

- *n1*: *name* and *next* fields are assigned values *NAME IS E.W ?? NEXT IS E.V ??*
- *n1* is then added into the linked list *WHAT DOES 'ADDED' MEAN?*
- *n2* reverses the roles of *v* and *w*

Note that the array *g->edges[e.w]* consists of pointers only (no data, elements are *next* fields)

- ... unlike the list nodes, which have *name* and *next* fields

Helpful is to show the list structure shown above again:



Implementation Comparison

Adjacency-list implementation:

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

Property	Adjacency	Adjacency
	Matrix	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 in the file *graph1.inp*:

```
#5
0 1
0 2
1 2 1 3 1 4
2 3 2 4
```

- the first line starts with a '#' character and states the number of vertices, called *numV*
  - the vertices are numbered from *0* .. *numV-1*
- the subsequent lines list pairs of vertices, which represent edges in the graph
  - they can be written along the line or down the page

We can read this data, build a graph and print its contents using the following client.

切换行号显示

```c
// readGraph.c  read a graph from stdin and print it
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "Graph.h"

#define WHITESPACE 100

int readNumV(void) { // returns the number of vertices numV or -1
   int numV;
   char w[WHITESPACE];
   scanf("%[ \t\n]s", w);  // skip leading whitespace
   if ((getchar() != '#') ||
       (scanf("%d", &numV) != 1)) {
       fprintf(stderr, "missing number (of vertices)\n");
       return -1;
   }
   return numV;
}

int readGraph(int numV, Graph g) { // reads number-number pairs until EOF
   int success = true;             // returns true if no error
   int v1, v2;
   while (scanf("%d %d", &v1, &v2) != EOF && success) {
       if (v1 < 0 || v1 >= numV || v2 < 0 || v2 >= numV) {
          fprintf(stderr, "unable to read edge\n");
          success = false;
       }
       else {
          insertEdge(newEdge(v1, v2), g);
       }
   }
   return success;
}

int main (void) {
    int numV;
    if ((numV = readNumV()) >= 0) {
        Graph g = newGraph(numV);
        if (readGraph(numV, g)) {
            showGraph(g);
        }
```

```
        g = freeGraph(g);
    }
    else {
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

Compile and execute:

```
prompt$ dcc readGraph.c GraphAM.c
prompt$ ./a.out < graph1.inp
V=5, E=7
<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>
```

Notice that *showGraph()* prints the number of vertices and edges (*numV* and *numE*), and that each edge is actually listed twice, once for each endpoint.

- for example, both <0 1> and <1 0> are shown

There are just 7 edges, but 14 are shown

Graphs 1 (2019-08-24 12:19:02由AlbertNymeyer编辑)