

目录

1. Graphs and their representations
 1. General terminology (for undirected, unweighted graphs)
 1. Complete graph
 2. Sparseness/denseness of a graph
 3. Tree
 4. Spanning tree
 5. Examples
 6. Graph Terminology
 7. Hamiltonian paths and cycles
 8. Eulerian path and cycles
 9. Examples of Eulerian and Hamiltonian cycles and paths
 10. Undirected vs Directed Graphs
 11. Other types of graphs
2. Implementing Graphs
 1. Graph ADT
 2. Adjacency Matrix Representation
 3. Adjacency List Representation
 4. Implementation Comparison

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

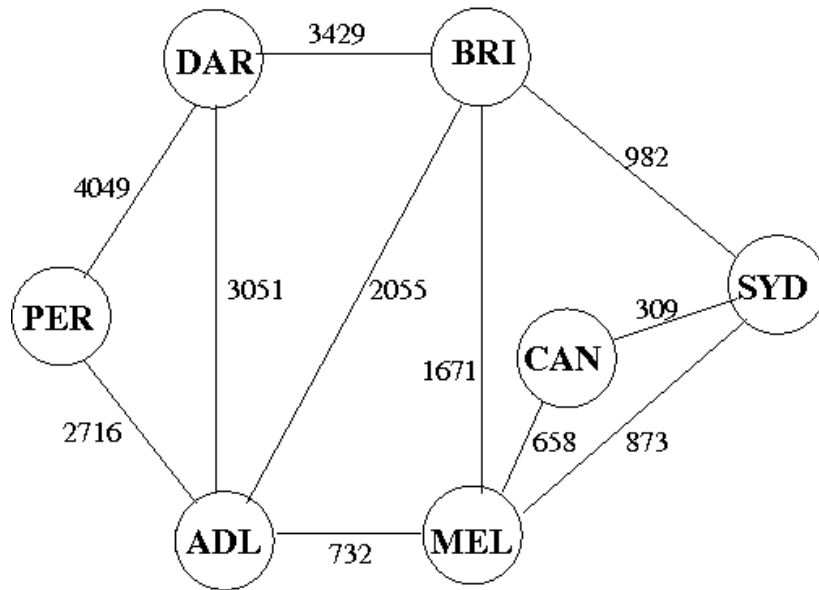
graph	vertices	edges
communication	telephones, computers	fiber optic cables
circuits	gates, registers, processors	wires
mechanical	joints	rods, beams, springs
hydraulic	reservoirs, pumping stations	pipelines
financial	stocks, currency	transactions
transportation	street intersections, airports	highways,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

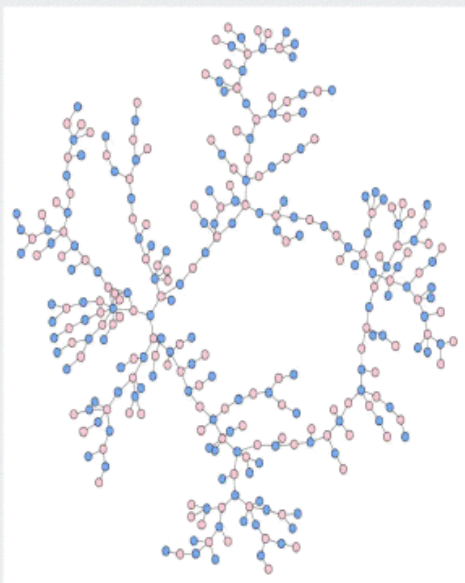
can be expressed (partially) as :



Many more interesting examples.

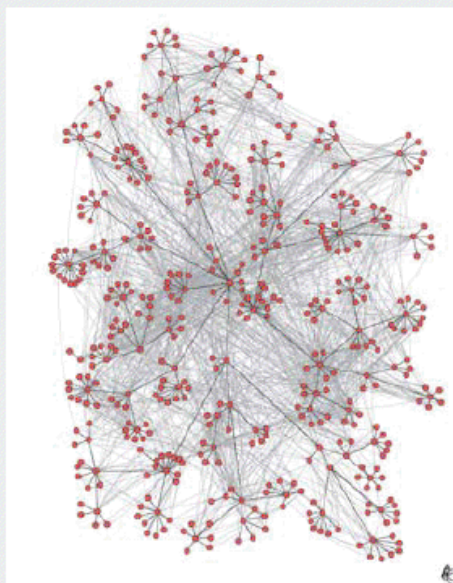
Social networks

high school dating



Reference: Barabási, Albert-László and Strogatz, 2004
image by Mark Newman

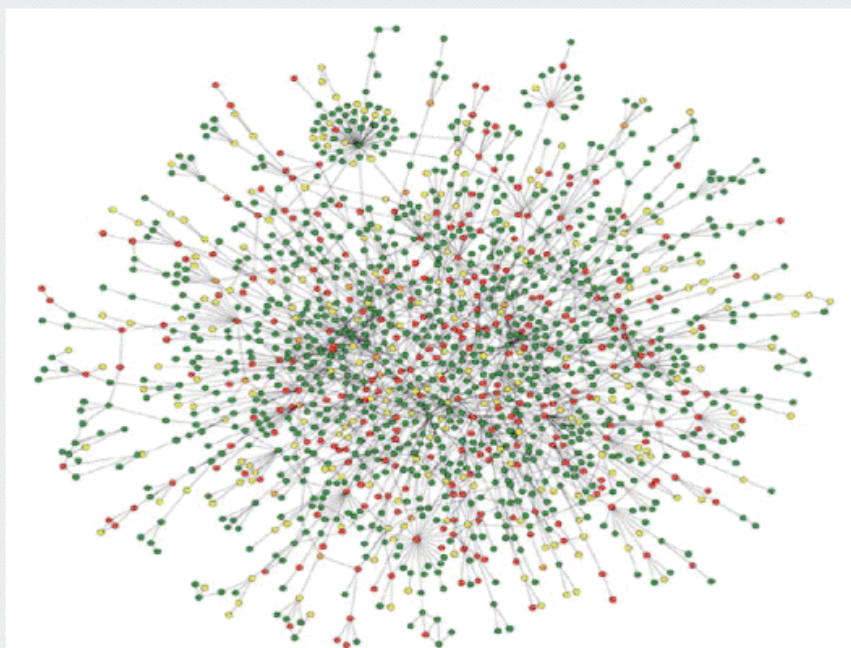
corporate e-mail



Reference: Adamic and Adar, 2004

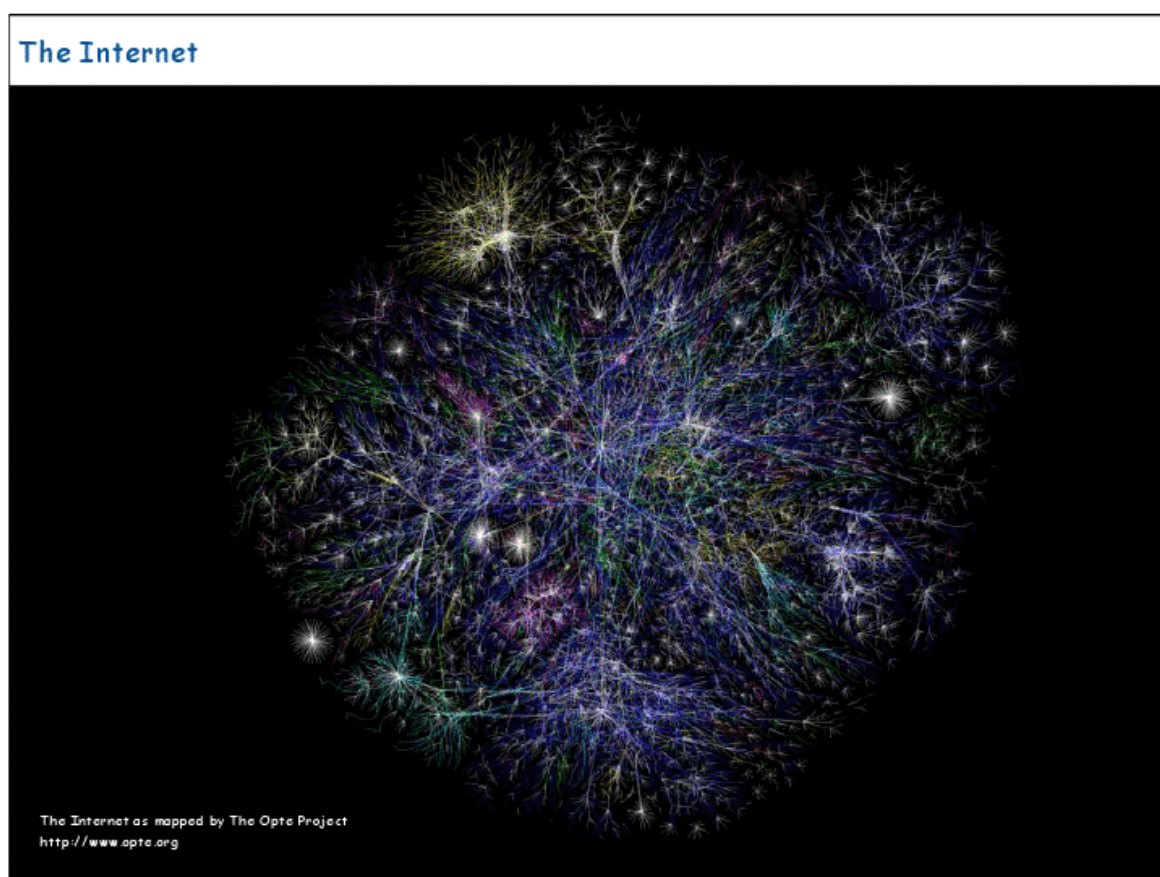
4

Protein interaction network



Reference: Jeong et al., Nature Reviews | Genetics

6

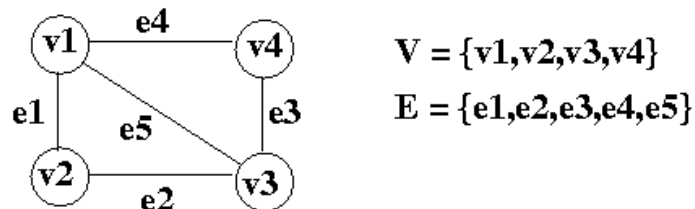


General terminology (for undirected, unweighted graphs)

- 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 $V \times V$)

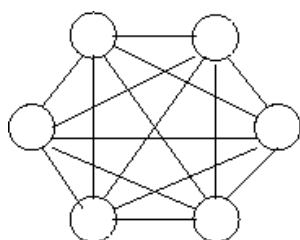


Complete graph

在图论的数学领域，完全图是一个简单的无向图，其中每对不同的顶点之间都恰连有一条边相连。

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
 - $|E| = V(V-1)/2$



一个局部的完全图

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

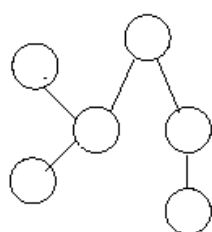
Spanning tree

- a tree that contains all vertices in the graph

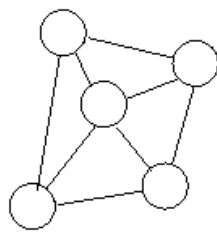
Tree: 可以是子树
Spanning tree: 整个图是个 tree

Examples

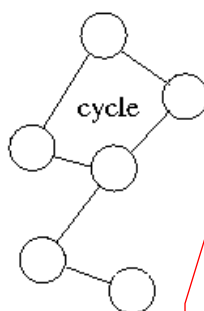
The following figures are graphs:



sparse graph
spanning tree

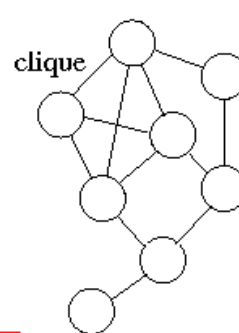


dense graph



cycle

个人理解:
cycle: 就是一圈
clique: 既有圈, 又有互相
的关联



clique

Graph 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

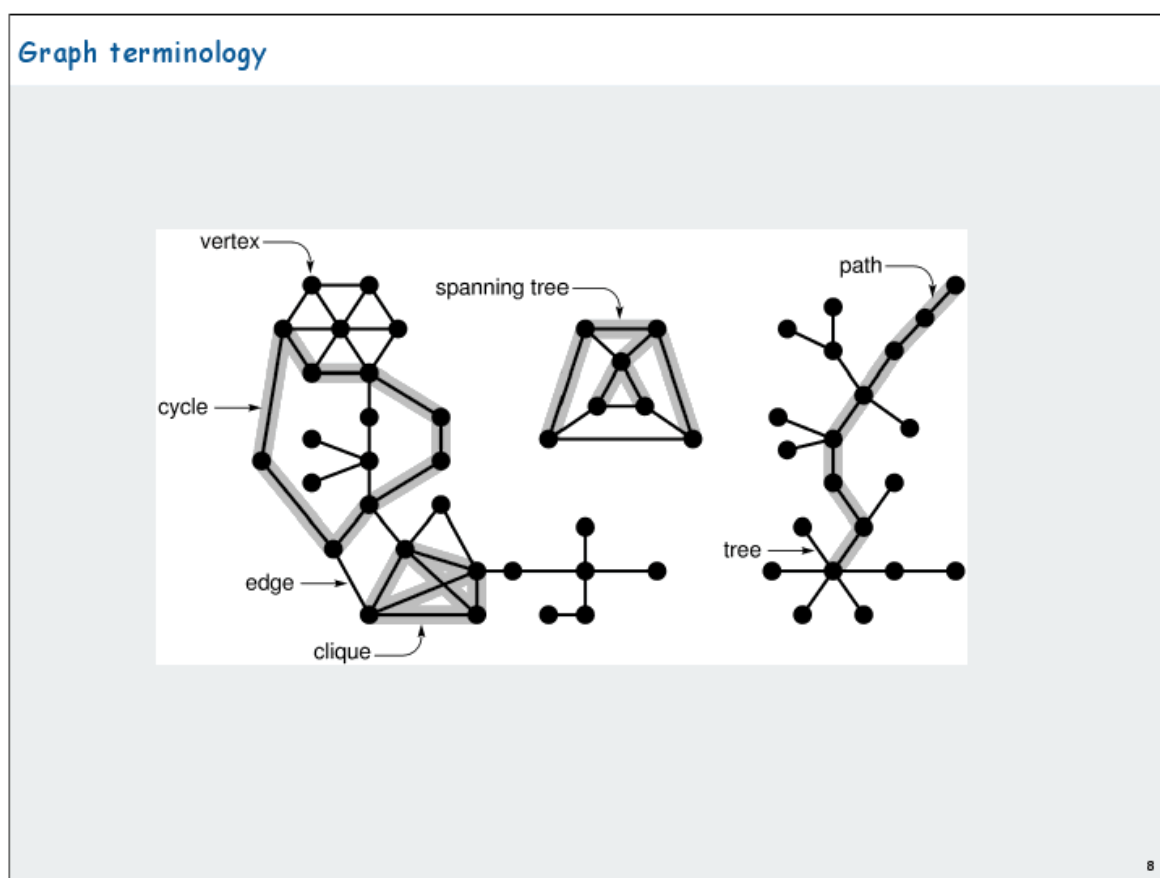
- there is a path from each vertex to every other vertex
- if a graph is not connected, it consists of a set of subgraphs each of which is connected
- if $|E| = 0$, then we have a set of vertices

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

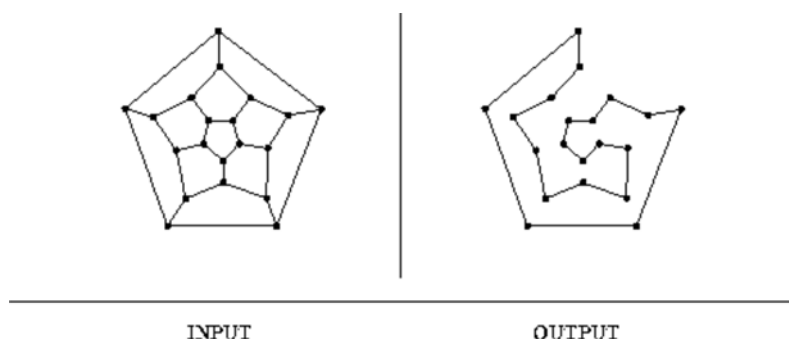
a sequence of edges joined at vertices

A **Hamiltonian path**:

- 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)*

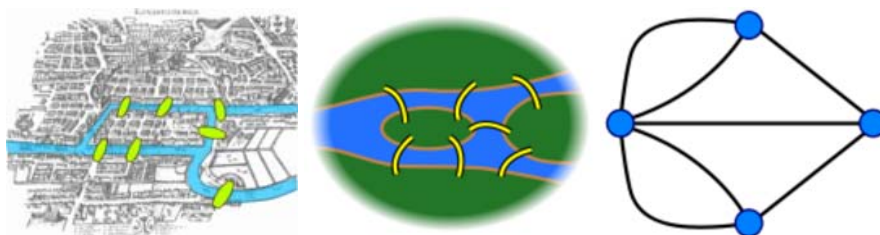
Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

- 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 $\geq n/2$
- [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.

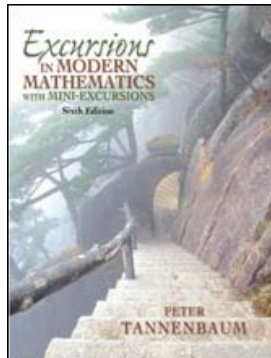
- ... so if any vertex has odd degree, it cannot contain a Euler cycle

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

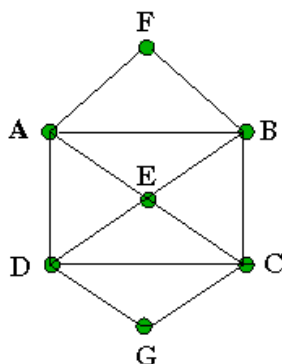
- ... so if there are more than 2 vertices of odd degree then it cannot contain a Euler path
- ... and if there is just one vertex of odd degree or more than 2 vertices then it has no Eulerian cycle or path

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

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

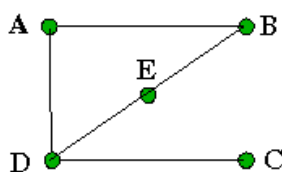


Examples of Eulerian and Hamiltonian cycles and paths



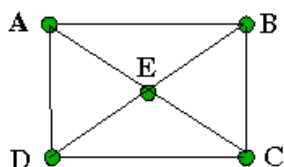
Has both Eulerian and Hamiltonian cycles

- 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**



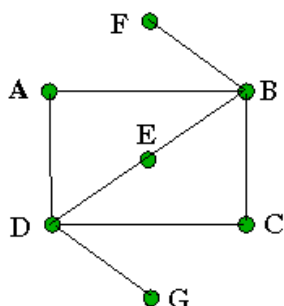
Has both paths

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



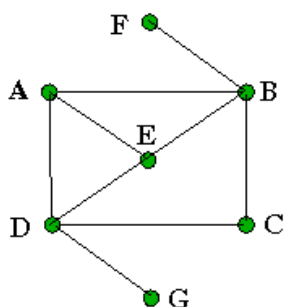
Hamiltonians but no Eulers

- 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 Eulers
- There is a theorem that states whether an arbitrary graph has an Eulerian path, or cycle, or neither
- There is *no* theorem for Hamiltonians

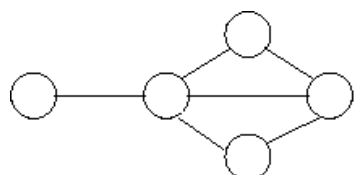
Undirected vs Directed Graphs

Undirected graph:

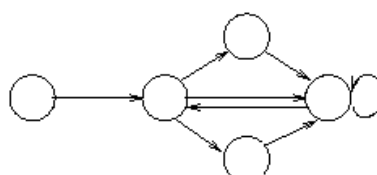
- $\text{edge}(u,v) = \text{edge}(v,u)$
- no self-loops (i.e. no $\text{edge}(v,v)$)

Directed graph:

- $\text{edge}(u,v) \neq \text{edge}(v,u)$,
- can have self-loops (i.e. $\text{edge}(v,v)$)



undirected graph



directed graph

Unless stated otherwise, we assume graphs are undirected.

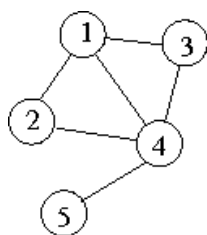
Other types of graphs

- **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...

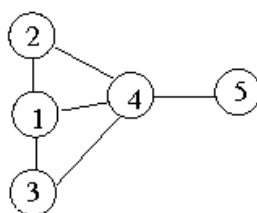
Implementing Graphs

Need some way of identifying vertices

- could give diagram showing edges and vertices
- could give a list of edges
- e.g. 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 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

```
1 // graph.h: generic interface for undirected/unweighted graphs
```

```

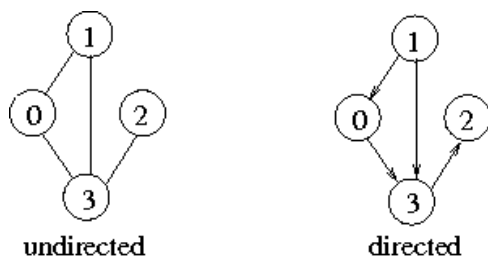
2
3 #define UNVISITED -1
4
5 typedef int Vertex;           // define a VERTEX
6
7 typedef struct {              // define an EDGE
8     Vertex v;
9     Vertex w;
10 } Edge;
11
12 typedef struct graphRep *Graph; // define a GRAPH
13
14 Graph newGraph(int);          // create a new graph
15 void freeGraph(Graph);        // free the graph mallocs
16 void showGraph(Graph);        // print the graph
17
18 Edge newE(Vertex, Vertex);    // create a new edge
19 void insertE(Graph, Edge);     // insert an edge
20 void removeE(Graph, Edge);    // remove an edge
21 void showE(Edge);             // print an edge
22 int isEdge(Graph, Edge);      // check edge exists
23

```

Adjacency Matrix Representation

Edges are represented by a $V \times V$ Boolean matrix, where V is the number of vertices.

Example:



are represented as:

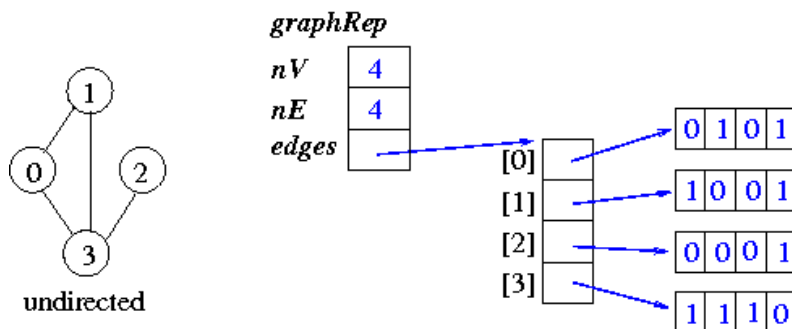
Undirected (note the symmetry)

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

```

1 // graphAM.c: an adjacency matrix implementation
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include "graph.h"
5
6 struct graphRep {
7     int nV;           // #vertices
8     int nE;           // #edges
9     int **edges;      // matrix of Booleans ... THIS IS THE ADJACENCY
MATRIX
10 };
11
12 Graph newGraph(int numVertices) {
13     Graph g = NULL;
14     if (numVertices < 0) {
15         fprintf(stderr, "newgraph: invalid number of vertices\n");
16     }
17     else {
18         g = malloc(sizeof(struct graphRep));
19         if (g == NULL) {
20             fprintf(stderr, "newGraph: out of memory\n");
21             exit(1);
22         }
23         g->edges = malloc(numVertices * sizeof(int *));
24         if (g->edges == NULL) {
25             fprintf(stderr, "newGraph: out of memory\n");
26             exit(1);
27         }
28         int v;
29         for (v = 0; v < numVertices; v++) {
30             g->edges[v] = malloc(numVertices * sizeof(int));
31             if (g->edges[v] == NULL) {
32                 fprintf(stderr, "newGraph: out of memory\n");
33                 exit(1);
34             }
35             for (int j = 0; j < numVertices; j++) {
36                 g->edges[v][j] = 0;
37             }
38         }
39         g->nV = numVertices;
40         g->nE = 0;
41     }
42     return g;
43 }
44
45 void freeGraph(Graph g) {
46     // not shown
47 }
48
49 void showGraph(Graph g) { // print a graph
50     if (g == NULL) {
51         printf("NULL graph\n");
52     }
53     else {

```

定义了
nV: 顶点数量
nE: 边的数量
edges: 二维数组, 邻接矩阵

邻接矩阵的所有值都是0

```

54     printf("V=%d, E=%d\n", g->nV, g->nE);
55     int i;
56     for (i = 0; i < g->nV; i++) {
57         int nshown = 0;
58         int j;
59         for (j = 0; j < g->nV; j++) {
60             if (g->edges[i][j] != 0) {
61                 printf("%d-%d ", i, j);
62                 nshown++;
63             }
64         }
65         if (nshown > 0) {
66             printf("\n");
67         }
68     }
69     return;
70 }
71 }
72
73 static int validV(Graph g, Vertex v) { // checks if v is in graph
74     return (v >= 0 && v < g->nV);
75 }
76
77 Edge newE(Vertex v, Vertex w) { // create an edge from v to w
78     Edge e = {v, w};
79     return e;
80 }
81 void showE(Edge e) { // print an edge
82     printf("%d-%d", e.v, e.w);
83     return;
84 }
85
86 int isEdge(Graph g, Edge e) { // return 1 if edge found, otherwise 0
87 // not shown
88 }
89
90 void insertE(Graph g, Edge e) { // insert an edge into a graph
91     if (g == NULL) {
92         fprintf(stderr, "insertE: graph not initialised\n");
93     }
94     else {
95         if (!validV(g, e.v) || !validV(g, e.w)) {
96             fprintf(stderr, "insertE: invalid vertices %d-%d\n", e.v,
e.w);
97         }
98         else {
99             if (isEdge(g, e) == 0) { // increment nE only if it is new
100                 g->nE++;
101             }
102             g->edges[e.v][e.w] = 1;
103             g->edges[e.w][e.v] = 1;
104         }
105     }
106     return;
107 }
108
109 void removeE(Graph g, Edge e) { // remove an edge from a graph
110     if (g == NULL) {
111         fprintf(stderr, "removeE: graph not initialised\n");
112     }
113     else {
114         if (!validV(g, e.v) || !validV(g, e.w)) {
115             fprintf(stderr, "removeE: invalid vertices\n");
116         }
117         else {
118             if (isEdge(g, e) == 1) { // is edge there?
119                 g->edges[e.v][e.w] = 0;
120                 g->edges[e.w][e.v] = 0;

```

显示邻接矩阵中值不是0的组合

首先判断是否为有效的顶点
其次判断是否已经相连
最后直接为矩阵赋值即可

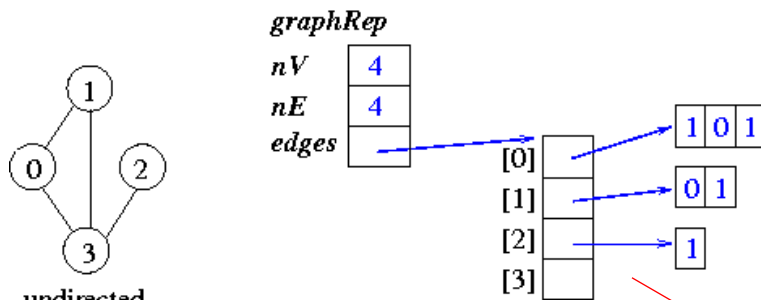
首先判断顶点是否存在
其次判断是否两点连接
最后将其值设置为0

```

121      g->nE--;
122      }
123    }
124  }
125  return;
126 }
    
```

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 $i+1, \dots, nV-1$ only (but still $O(V^2)$)



- the implementation above does not do this

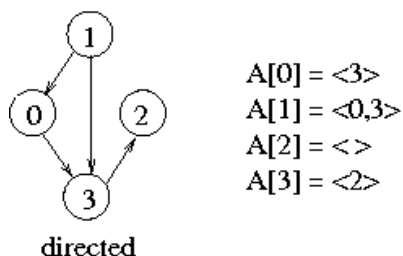
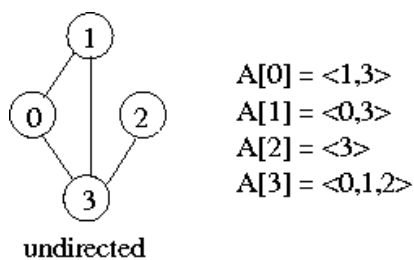
没有头节点，直接就是第一个节点

Adjacency List Representation

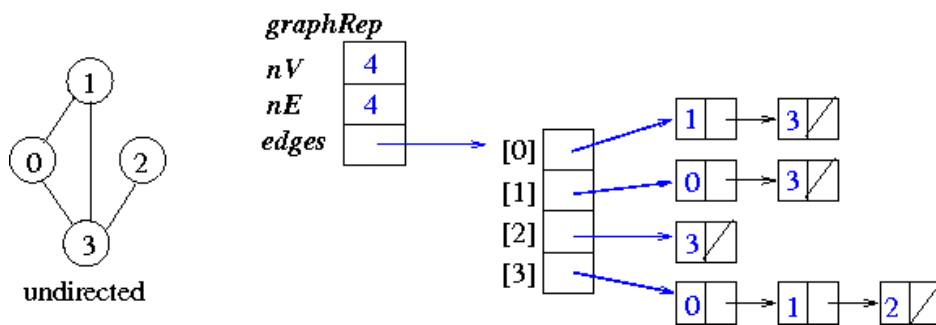
Adjacent vertices are stored in a linked list for each vertex.

- 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

```

1 // graphAL.c: an adjacency list implementation
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include "graph.h"
5
6 typedef struct node *list;
7 struct node {
8     Vertex name;
9     list next;
10 };
11
12 struct graphRep {
13     int nV;    // #vertices
14     int nE;    // #edges
15     list *edges; // array of linked lists ... THIS IS THE ADJACENCY
LIST
16 };
17
18 Graph newGraph(int numVertices) {
19     Graph g = NULL;
20     if (numVertices < 0) {
21         fprintf(stderr, "newgraph: invalid number of vertices\n");
22     }
23     else {
24         g = malloc(sizeof(struct graphRep));
25         if (g == NULL) {
26             fprintf(stderr, "newGraph: out of memory\n");
27             exit(1);
28         }
29         g->edges = malloc(numVertices * sizeof(int *));
30         if (g->edges == NULL) {
31             fprintf(stderr, "newGraph: out of memory\n");
32             exit(1);
33         }
34         int v;
35         for (v = 0; v < numVertices; v++) {
36             g->edges[v] = NULL;
37         }
38         g->nV = numVertices;
39         g->nE = 0;
40     }
41     return g;
42 }
43
44 void freeGraph(Graph g) {
45     // not shown
46 }
47
48 void showGraph(Graph g) { // print a graph
49     if (g == NULL) {
50         printf("NULL graph\n");
51     }
52     else {
53         printf("V=%d, E=%d\n", g->nV, g->nE);

```

edges : 链表数组

链表数组的首地址都为NULL

```

54     int i;
55     for (i = 0; i < g->nV; i++) {
56         int nshown = 0;
57         list listV = g->edges[i];
58         while (listV != NULL) {
59             printf("%d-%d ", i, listV->name);
60             nshown++;
61             listV = listV->next;
62         }
63         if (nshown > 0) {
64             printf("\n");
65         }
66     }
67 }
68 return;
69 }
70
71 static int validV(Graph g, Vertex v) { // checks if v is in graph
72     return (v >= 0 && v < g->nV);
73 }
74
75 Edge newE(Vertex v, Vertex w) {
76     Edge e = {v, w};
77     return e;
78 }
79
80 void showE(Edge e) { // print an edge
81     printf("%d-%d", e.v, e.w);
82     return;
83 }
84
85 int isEdge(Graph g, Edge e) { // return 1 if edge found, otherwise 0
86     // not shown
87 }
88
89 void insertE(Graph g, Edge e){
90     if (g == NULL) {
91         fprintf(stderr, "insertE: graph not initialised\n");
92     }
93     else {
94         if (!validV(g, e.v) || !validV(g, e.w)) {
95             fprintf(stderr, "insertE: invalid vertices %d-%d\n", e.v,
e.w);
96         }
97         else {
98             if (isEdge(g, e) == 0) {
99                 list newnodev = malloc(sizeof(struct node));
100                 list newnodew = malloc(sizeof(struct node));
101                 if (newnode1 == NULL || newnode2 == NULL) {
102                     fprintf(stderr, "Out of memory\n");
103                     exit(1);
104                 }
105                 newnodev->name = e.w; // put in the data
106                 newnodev->next = g->edges[e.v]; // link to the
existing list attached to e.v
107                 g->edges[e.v] = newnodev; // link e.v to new
node
108                 newnodew->name = e.v;
109                 newnodew->next = g->edges[e.w];
110                 g->edges[e.w] = newnodew;
111                 g->nE++;
112             }
113         }
114     }
115     return;
116 }
117
118 void removeE(Graph g, Edge e) {

```

遍历显示链表内容

直接将需要insert的节点链接到最前面即可

要考虑删除的节点是否为第一个，分别操作

```
119 // not shown
120 }
```

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 !
 - parallel edge detection also requires V complexity!
 - order of edges in the linked lists is not defined: can be a problem in applications if the order is important

Property	Adjacency Matrix	Adjacency List
Space	V^2	$V+E$
Initialise	V^2	V
Insert edge	1	1
Find/remove edge	1	V

Graphs 1 (2019-07-15 17:09:24由AlbertNymeyer编辑)