Graph Theory

What's a Graph?

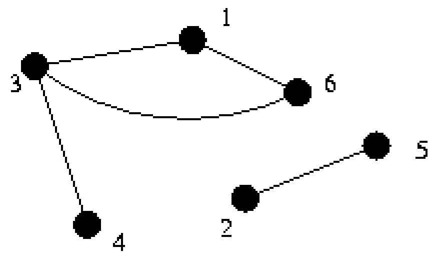
Formally, a *graph* is the following:

- a collection of *vertices* V, and
- a collection of *edges* E consisting of pairs of vertices.

Think of vertices as ``locations''. The set of vertices is the set of all the possible locations. In this analogy, edges represent paths between pairs of those locations; the set E contains all the paths between the locations.

Representation

The graph is normally represented using that analogy. Vertices are points or circles; edges are lines between them.



In this example graph, V = {1, 2, 3, 4, 5, 6} and E = {(1,3), (1,6), (2,5), (3,4), (3,6)}.

Each *vertex* is a member of the set V. A vertex is sometimes called a *node*.

Each *edge* is a member of the set E. Note that some vertices might not be the end point of any edge. Such vertices are termed `isolated'.

Sometimes, numerical values are associated with edges, specifying lengths or costs; such graphs are called *edge-weighted* graphs (or weighted graphs). The value associated with an edge is called the *weight* of the edge. A similar definition holds for node-weighted graphs,

Examples of Graphs

Telecowmunication (USACO Championship 1996)

Given a set of computers and a set of wires running between pairs of computers, what is the minimum number of machines whose crash causes two given machines to be unable to communicate? (The two given machines will not crash.)

Graph: The vertices of the graph are the computers. The edges are the wires between the computers.

Sample Problem: Riding The Fences

Farmer John owns a large number of fences, which he must periodically check for integrity. He keeps track of his fences by maintaining a list of points at which fences intersect. He records the name of the point and the one or two fence names that touch that point. Every fence has two end points, each at some intersection point, although the intersection point may be the end point of only one fence.

Given a fence layout, calculate if there is a way for Farmer John to ride his horse to all of his fences without riding along a fence more than once. Farmer John can start and finish anywhere, but cannot cut across his fields (i.e., the only way he can travel between intersection points is along a fence). If there is a way, find one way.

Graph: Farmer John starts at intersection points and travels between the points along fences. Thus, the vertices of the underlying graph are the intersection points, and the fences represent edges.

Knight moves

Given: Two squares on an 8x8 chessboard. Determine the shortest sequence of knight moves from one square to the other.

Graph: The graph here is harder to see. Each location on the chessboard represents a vertex. There is an edge between two positions if it is a legal knight move.

Overfencing [Kolstad & Schrijvers, Spring 1999 USACO Open]

Farmer John created a huge maze of fences in a field. He omitted two fence segments on the edges, thus creating two ``exits'' for the maze. The maze is a `perfect' maze; you can find a way out of the maze from any point inside it.

Given the layout of the maze, calculate the number of steps required to exit the maze from the `worst' point in the maze (the point that is `farther' from either exit when walking optimally to the closest exit).

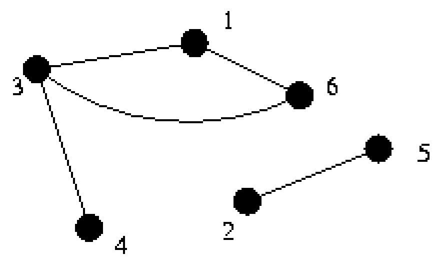
Here's what one particular W=5, H=3 maze looks like:

```
+-+-+-+-+-+
|         |
+-+ +-+ + +
|     | | |
+ +-+-+ + +
| |     |
+-+ +-+-+-+
```

Graph: The vertices of the graph are positions in the grid. There is an edge between two vertices if they represent adjacent positions that are not separated by a wall.

Terminology

Let's look again at the first example graph:



An edge is a *self-loop* if it is of the form (u,u). The sample graph contains no self-loops.

A graph is *simple* if it neither contains self-loops nor contains an edge that is repeated in E. A graph is called a *multigraph* if it contains a given edge more than once or contain self-loops. For our discussions, graphs are assumed to be simple. The example graph is a simple graph.

An edge (u,v) is *incident* to both vertex u and vertex v. For example, the edge (1,3) is incident to vertex 3.

The *degree* of a vertex is the number of edges which are incident to it. For example, vertex 3 has degree 3, while vertex 4 has degree 1.

Vertex u is *adjacent* to vertex v if there is some edge to which both are incident (that is, there is an edge between them). For example, vertex 2 is adjacent to vertex 5.

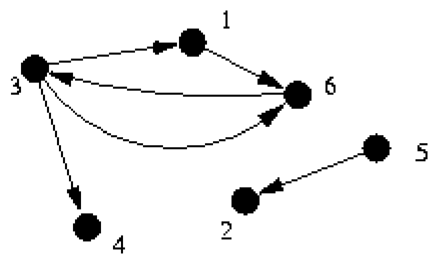
A graph is said to be *sparse* if the total number of edges is small compared to the total number possible ((*N x (N-1))/2*) and *dense* otherwise. For a given graph, whether it is dense or sparse is not well-defined.

Directed Graph

Graphs described thus far are called *undirected*, as the edges go `both ways'. So far, the graphs have connoted that if one can travel from vertex 1 to vertex 3, one can also travel from vertex 3 to vertex 1. In other words, (1,3) being in the edge set implies (3,1) is in the edge set.

Sometimes, however, a graph is *directed*, in which case the edges have a direction. In this case, the edges are called *arcs*.

Directed graphs are drawn with arrows to show direction.



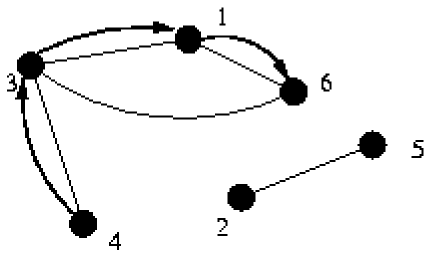
The *out-degree* of a vertex is the number of arcs which *begin* at that vertex. The *in-degree* of a vertex is the number of arcs which *end* at that vertex. For example, vertex 6 has in-degree 2 and out-degree 1.

A graph is assumed to be undirected unless specifically called a directed graph.

Paths

A *path* from vertex *u* to vertex *x* is a sequence of vertices (v_0, v_1, ..., v_k) such that $v_0 = u$ and $v_k = x$ and (v_0, v_1) is an edge in the graph, as is (v_1, v_2), (v_2, v_3), etc. The length of such a path is *k*.

For example, in the undirected graph above, (4, 3, 1, 6) is a path.



This path is said to *contain* the vertices v_0, v_1, etc., as well as the edges (v_0, v_1), (v_1, v_2), etc.

Vertex *x* is said to be *reachable* from vertex *u* if a path exists from *u* to *x*.

A path is *simple* if it contains no vertex more than once.

A path is a *cycle* if it is a path from some vertex to that same vertex. A cycle is *simple* if it contains no vertex more than once, except the start (and end) vertex, which only appears as the first and last vertex in the path.

These definitions extend similarly to directed graphs (e.g., (v_0, v_1), (v_1, v_2), etc. must be arcs).

Graph Representation

The choice of representation of a graph is important, as different representations have very different time and space costs.

The vertices are generally tracked by numbering them, so that one can index them just by their number. Thus, the representations focus on how to store the *edges*.

Edge List

The most obvious way to keep track of the edges is to keep a list of the pairs of vertices representing the edges in the graph.

This representation is easy to code, fairly easy to debug, and fairly space efficient. However, determining the edges incident to a given vertex is expensive, as is determining if two vertices are adjacent. Adding an edge is quick, but deleting one is difficult if its location in the list is not known.

For weighted graphs, this representation also keeps one more number for each edge, the edge weight. Extending this data structure to handle directed graphs is straightforward. Representing multigraphs is also trivial.

Example

The sample undirected graph might be represented as the following list of edges:

	V_1	V_2
e_1	4	3
e_2	1	3
e_3	2	5
e_4	6	1
e_5	3	6

Adjacency Matrix

A second way to represent a graph utilizes an *adjacency matrix*. This is a N by N array (N is the number of vertices). The i,j entry contains a 1 if the edge (i,j) is in the graph; otherwise it contains a 0. For an undirected graph, this matrix is symmetric.

This representation is easy to code. It's much less space efficient, especially for large, sparse graphs. Debugging is harder, as the matrix is large. Finding all the edges incident to a given vertex is fairly expensive (linear in the number of vertices), but checking if two vertices are adjacent is very quick. Adding and removing edges are also very inexpensive operations.

For weighted graphs, the value of the (i,j) entry is used to store the weight of the edge. For an unweighted multigraph, the (i,j) entry can maintain the number of edges between the vertices. For a weighted multigraph, it's harder to extend this.

Example

The sample undirected graph would be represented by the following adjacency matrix:

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	0	1	0	0	1

V_2	0	0	0	0	1	0
V_3	1	0	0	1	0	1
V_4	0	0	1	0	0	0
V_5	0	1	0	0	0	0
V_6	1	0	1	0	0	0

It is sometimes helpful to use the fact that the (i,j) entry of the adjacency matrix raised to the k-th power gives the number of paths from vertex i to vertex j consisting of exactly k edges.

Adjacency List

The third representation of graph is to keep track of all the edges incident to a given vertex. This can be done by using an array of length N, where N is the number of vertices. The i^{th} entry in this array is a list of the edges incident to i'th vertex (edges are represented by the index of the other vertex incident to that edge).

This representation is much more difficult to code, especially if the number of edges incident to each vertex is not bounded, so the lists must be linked lists (or dynamically allocated). Debugging this is difficult, as following linked lists is more difficult. However, this representation uses about as much memory as the edge list. Finding the vertices adjacent to each node is very cheap in this structure, but checking if two vertices are adjacent requires checking all the edges adjacent to one of the vertices. Adding an edge is easy, but deleting an edge is difficult, if the locations of the edge in the appropriate lists are not known.

Extend this representation to handle weighted graphs by maintaining both the weight and the other incident vertex for each edge instead of just the other incident vertex. Multigraphs are already representable. Directed graphs are also easily handled by this representation, in one of several ways: store only the edges in one direction, keep a seperate list of incoming and outgoing arcs, or denote the direction of each arc in the list.

Example

The adjacency list representation of the example undirected graph is as follows:

	Adjacent
Vertex	Vertices
1	3, 6
2	5
3	6, 4, 1
4	3
5	2
6	3, 1

Implicit Representation

For some graphs, the graph itself does not have to be stored at all. For example, for the Knight moves and Overfencing problems, it is easy to calculate the neighbors of a vertex, check adjacency, and determine all the edges without actually storing that information, thus, there is no reason to actually store that information; the graph is implicit in the data itself.

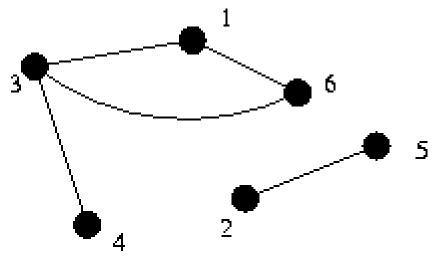
If it is possible to store the graph in this format, it is generally the correct thing to do, as it saves a lot on storage and reduces the complexity of your code, making it easy to both write and debug.

If N is the number of vertices, M the number of edges, and d_{max} the maximum degree of a node, the following table summarizes the differences between the representations:

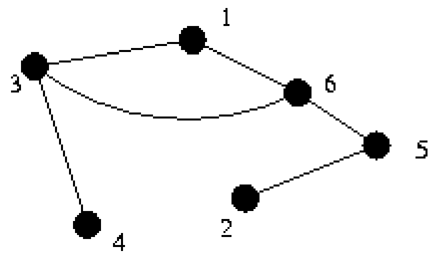
Efficiency	Edge List	Adj Matrix	Adj List
Space	2xM	N^2	2xM
Adjacency Check	M	1	d_{max}
List of Adj Vertices	M	N	d_{max}
Add Edge	1	1	1
Delete Edge	M	2	2xd_{max}

Connectedness

An undirected graph is said to be *connected* if there is a path from every vertex to every other vertex. The example graph is *not* connected, as there is no path from vertex 2 to vertex 4.



However, if you add an edge between vertex 5 and vertex 6, then the graph becomes connected.



A *component* of a graph is a maximal subset of the vertices such that every vertex is reachable from each other vertex in the component. The original example graph has two components: {1, 3, 4, 6} and {2, 5}. Note that {1, 3, 4} is not a component, as it is not maximal.

A directed graph is said to be *strongly connected* if there is a path from every vertex to every other vertex.

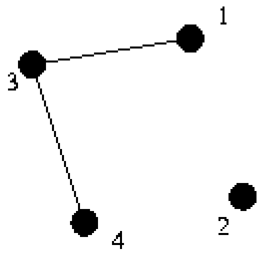
A *strongly connected component* of a directed graph is a vertex u and the collection of all vertices v such that there is a path from u to v and a path from v to u.

Subgraphs

Graph G' = (V', E') is a subgraph of G = (V, E) if V' is a subset of V and E' is a subset of E.

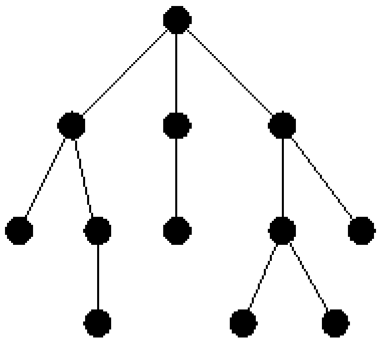
The subgraph of G *induced* by V' is the graph (V', E'), where E' consists of all the edges of E that are between members of V'.

For example, for V' = {1, 3, 4, 2}, the subgraph induced is:



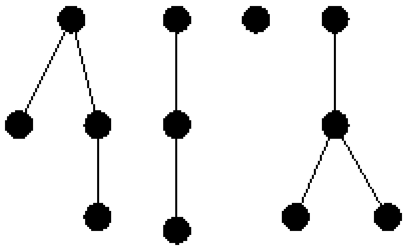
Special Graphs

An undirected graph is said to be a *tree* if it contains no cycles and is connected.



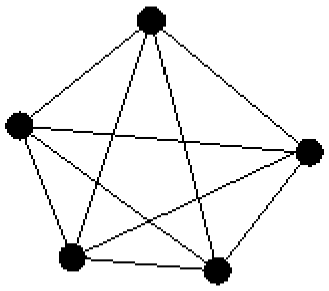
Many trees are what is called *rooted*, where there is a notion of the "top" node, which is called the *root*. Thus, each node has one *parent*, which is the adjacent node which is closer to the root, and may have any number of *children*, which are the rest of the nodes adjacent to it. The tree above was drawn as a rooted tree.

An undirected graph which contains no cycles is called a *forest*.

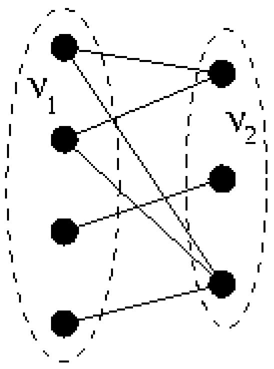


A directed acyclic graph is often referred to as a *dag*.

A graph is said to be *complete* if there is an edge between every pair of vertices.



A graph is said to be *bipartite* if the vertices can be split into two sets V_1 and V_2 such there are no edges between two vertices of V_1 or two vertices of V_2.



Graph Theory

图论知识

译 by Lucky Crazy

何为*图*?

正式地说，图 G 是由：

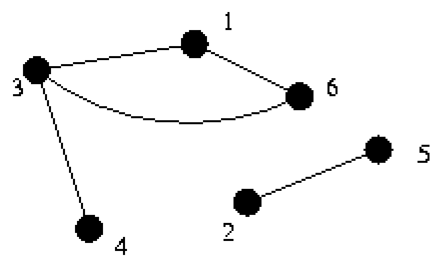
- 所有***结点***的集合 V
- 所有***边***的集合 E

构成的，简写成 G(V，E)。

我们可以把结点假设成一个“地点”，而结点集合就是一个所有地点的集合。同样，边可以被假设成连接两个“地点”的一条“路”；那么边的集合就可以认为是所有这样的“路”的集合。

表示方法：

图通常用如下方法表示；结点是点或者圈，而边是直线或者曲线。



在上图中，结点 V = {1, 2, 3, 4, 5, 6} ，边 E = {(1, 3), (1, 6), (2, 5), (3, 4), (3, 6)}.

每一个结点都是集合 V 中的一个数字，每条边都是集合 E 中的成员，注意并不是每个节点都有边与其他结点相连，这样没有边与其他结点相连的结点被称作***孤立结点***。

有时，边会与一些数值关联，类似表示边的长度或花费。我们把这些数字称作边的***权***，这样边有权的图被称为***边权图***。类似的，我们还定义了***点权图***，即每个结点都有一个权。

图论的几个例子

奶牛的电信 (USACO 锦标赛 1996)

农夫约翰的奶牛们喜欢通过电邮保持联系，于是她们建立了一个奶牛电脑网络，以便互相交流。这些机器用如下的方式发送电邮：如果存在一个由 c 台电脑组成的序列 a1,a2,...,a(c)，且 a1 与 a2 相连，a2 与 a3 相连，等等，那么电脑 a1 和 a(c)就可以互发电邮。很不幸，有时候奶牛会不小心踩到电脑上，农夫约翰的车也可能碾过电脑，这台倒霉的电脑就会坏掉。这意味着这台电脑不能再发送电邮了，于是与这台电脑相关的连接也就不可用了。有两头奶牛就想：如果我们两个不能互发电邮，至少需要坏掉多少台电脑呢？请编写一个程序为她们计算这个最小值和与之对应的坏掉的电脑集合。

图: 每个结点表示一台电脑，而边就相对应的成了连接各台电脑的缆线。

骑马修栅栏

农民 John 每年有很多栅栏要修理。他总是骑着马穿过每一个栅栏并修复它破损的地方。John 是一个与其他农民一样懒的人。他讨厌骑马，因此从来不两次经过一个一个栅栏。你必须编一个程序，读入栅栏网络的描述，并计算出一条修栅栏的路径，使每个栅栏都恰好被经过一次。John 能从任何一个顶点(即两个栅栏的交点)开始骑马，在任意一个顶点结束。每一个栅栏连接两个顶点，顶点用 1 到 500 标号(虽然有的农场并没有 500 个顶点)。一个顶点上可连接任意多(>=1)个栅栏。所有栅栏都是连通的(也就是你可以从任意一个栅栏到达另外的所有栅栏)。你的程序必须输出骑马的路径(用路上依次经过的顶点号码表示)。我们如果把输出的路径看成是一个 500 进制的数，那么当存在多组解的情况下，输出 500 进制表示法中最小的一个 (也就是输出第一个数较小的，如果还有多组解，输出第二个数较小的，等等)。输入数据保证至少有一个解。

图: 农民 John 从一个栅栏交叉点开始，经过所有栅栏一次。因而，图的结点就是栅栏交叉点，边就是栅栏。

骑士覆盖问题

在一个 *n x n* 的棋盘中摆放尽量少的骑士，使得棋盘的每一格都会被至少一个骑士攻击到。但骑士无法攻击到它自己站的位置.

图: 这里的图较为特殊，棋盘的每一格都是一个结点，如果骑士能从这一格跳到另一格，这就说在这两个格相对应的结点之间有一条边。

穿越栅栏 [1999 USACO 春季公开赛]

农夫 John 在外面的田野上搭建了一个巨大的用栅栏围成的迷宫。幸运的是，他在迷宫的边界上留出了两段栅栏作为迷宫的出口。更幸运的是，他所建造的迷宫是一个“完美的”迷宫：即你能从迷宫中的任意一点找到一条走出迷宫的路。
给定迷宫的宽 W(1<=W<=38)及长 H(1<=H<=100)。
2*H+1 行，每行 2*W+1 的字符以下面给出的格式表示一个迷宫。然后计算从迷宫中最“糟糕”的那一个点走出迷宫所需的步数。（即使从这一点以最优的方式走向最靠近的出口，它仍然需要最多的步数）当然了，牛们只会水平或垂直地在 X 或 Y 轴上移动，他们从来不走对角线。每移动到一个新的方格算作一步（包括移出迷宫的那一步）

这是一个 W=5,H=3 的迷宫：

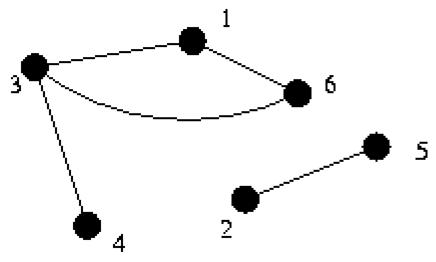
```
+-+-+-+-+-+
|         |
+-+ +-+ + +
|     | | |
+ +-+-+ + +
| |     |
+-+ +-+-+-+
```

如上图的例子，栅栏的柱子只出现在奇数行或奇数列。每个迷宫只有两个出口。

图: 如上图，图的每一个位置都是一个结点，如果两个相邻的位置之间没有被栅栏分开，则说在这两个位置相对应的结点之间有一条边。

用语：

我们再看刚才的图：



如果有一条边起点与终点都是同一个结点，我们就称它为***环边***，表示为(v, v)，上图中没有环边。

简单图是指一张没有环边且边在边集 E 中不重复出现的图。与简单图相对的是***复杂图***。在我们的讨论中不涉及复杂图，所有的图都是简单图。

边 (u,v) ***连接***了结点 u 和结点 v 。例如，边 (1,3) 连接了结点 1 与 3。结点的***度***是指连接该结点所有边的个数。例如，结点 3 的度数是 3，结点 4 的度数是 1。

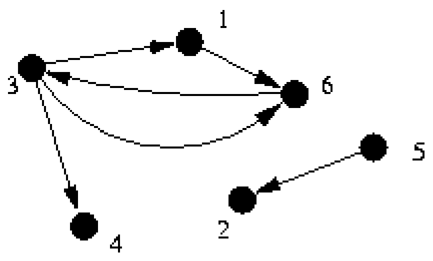
通常，如果结点 u 和 v 被用一条边连接，我们就说结点 u 与 v ***相连，***例如，上图结点 2 与 5 相连。

稀疏图 的定义是图的边的总数小于可能边数((N x $(N-1))/2$)（N为结点数），而***密集图***的定义相反。例如，上图就是一张稀疏图。

有向图：

在以上内容中我们介绍的图都为***无向图，***即每一条边都是“双向”的，如果存在一条边(1,3)，则我们可以从结点 1 到达结点 3，也可以从结点 3 到达结点 1，换句话说，如果存在边(1,3)就必定存在边(3,1)。

但有时，图也必须被定于成有向图，即每条边都有一个方向，我们只能“单向”地根据边的方向遍历图。这样的边又称为***弧***。弧用带箭号的直线或弧线表示。

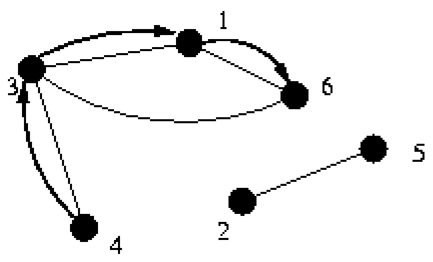


有向图结点的***出度***就是指从该结点“出发”的弧的个数，相反，结点的***入度***就是指在该结点“结束”的弧的个数。例如，上图结点 6 的入度为 2，出度为 1。

路径:

如果我们说结点 u 到结点 x 有路径，就是指存在一个结点的序列（v_0, v_1, . . . , v_k）且 $v_0 = u$ 、$v_k = x$ ， 以及(v_0, v_1) 是边集中的一条边，同样，(v_1, v_2)，(v_2, v_3)……也是。

例如，上面的无向图，（4, 3, 1, 6）就是从 4 到 6 的一条路径。



这条路径包含了边(4,3)、(3,1)、(1,6)。

如果结点 x 到 v 有一条路径，则我们从结点 x 开始通过边必定能访问到结点 v。

在一条路径序列中，如果所经过的结点都只在序列中出现一次，我们就称它为***简单路径***。

环的定义是一条起始结点与中止结点为同一结点的路径，如果一个环除了起始结点（中止结点）以外的所有结点都只在环中出现一次，那么这种环被称为***简单环***。

以上定义同样适用于有向图。

图的表示法

选择一个好的方法来表示一张图是十分重要的，因为不同的图的表示方法有着不同的时间及空间需求。

通常，结点被用数字编号来表示，我们可以通过它们的编号数字来对他们进行索引。 这样，似乎所有问题都集中在如何表示边上了。

边集

边集表示法似乎是最明显的表示法，他将所有边列成一张表，用结点来表示边。

该表示法容易编写，容易调试且所需空间小。但是，它需要花费相当大的代价用于确定一条边是否存在，即确定两个结点是否相连。利用边集添加新边是很快的，但删除旧边就很麻烦了，尤其是当需要删除的边在边集中的所在位置不确定时。

对于带权图，该表示法也只需要在边集中添加一个元素，用于记录边权。同样，该表示法也适用于有向图和稀疏图。

例：

一张无向无权图可以表示成边集如下：

	V_1	V_2
e_1	4	3
e_2	1	3
e_3	2	5
e_4	6	1
e_5	3	6

邻接矩阵

邻接矩阵式表示图的另一种方法，邻接矩阵是一个 N 乘以 N 的数组(N 为结点数)。如果边集 E 中存在边(i,j)，则对应数组的(i,j)元素的值为一。相反，如果数组元素的值为零，就意味着图中结点 i 与 j 之间没有边。无向图的邻接矩阵总是关于左上右下对角线对称。

该表示法容易编写。但他对空间的需求和浪费较大，尤其是对于较大的稀疏图，调试起来也比较麻烦，并且寻找与某一结点相连的结点也很慢。不过该表示法在判断两结点是否相连，以及在添加、删除结点方面速度都是极快的。

对于带权图，数组的(i,j)元素的值被表示为边(i,j)的权。对于无权复杂图来说，数组的(i,j)元素的值可以用于表示结点(i,j)之间边的个数。而对于有权复杂图来说，邻接矩阵很难将其表示清楚。

例：

下面是用邻接矩阵表示一幅无权图的例子：

	V_1	V_2	V_3	V_4	V_5	V_6
V_1	0	0	1	0	0	1
V_2	0	0	0	0	1	0
V_3	1	0	0	1	0	1
V_4	0	0	1	0	0	0
V_5	0	1	0	0	0	0
V_6	1	0	1	0	0	0

邻接表

第三种表示图的方法是用一个列表列出所有与现结点之间有边存在的结点名称。该方法可以用一个长度为 N 的数组来实现（N 为结点个数），数组的每一个元素都是一个链表表头。数组的第 i 元素所连接的链表连接了所有与结点 i 之间有边的结点名称。

该表示方法较难编写,特别是对于复杂图,两结点之间边的个数不受限制的时候，链表可能要做成环，或者要动态分配内存。邻接表的调试也同样十分麻烦，特别是使用了环形链表后。但是，这种表示法所需求的空间与边集相同，寻找某结点的相邻结点也十分容易，然而，如果需要判断两结点是否相邻，就需要遍历该结点的所有相邻结点以证明相邻性，这浪费了不少时间。添加边十分容易，但删除边就十分困难了。

将该表示法用于带权图,就需要在链表的每一节上添加一个元素用于储存该节表示的两结点之间的权。有向图与复杂图同样可以用邻接表表示。对于有向图，我们只需存储单向的相邻即可。

例：

邻接表表示无向图如下：

	邻接
结点	结点
1	3，6
2	5
3	6，4，1
4	3
5	2
6	3，1

表示法的选择:

对于某些图来说，我们并不需要考虑所有结点之间的关系，例如上面的骑士覆盖和穿越栅栏两题，我们只需考虑与某结点相邻结点之间的关系，确定相连，我们就不用考虑存储两个不相邻结点之间的信息。当然，这些信息来自于题目本身的暗示。

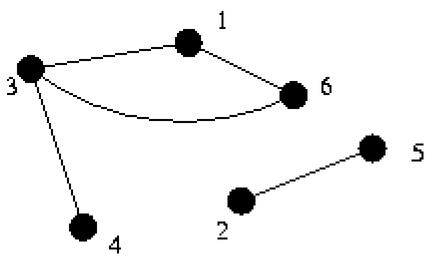
如果你发现一种表示方法可以在规定空间与时间范围内实现你的算法解决问题，并且可以使你的程序变得简洁、容易调试，那它通常就是对的。记住，编写与调试的简单是最重要的，我们不需要为一些毫无意义的加快程序速度，减少使用空间来浪费编写与调试的时间。

我们还要通过题目给我们的信息，确定我们要对图进行哪些操作，权衡后来决定表示方法。下面的表示我们为您提供的选择依据（N 为结点数，M 为边数，d_{max} 为图中度的最大值）：

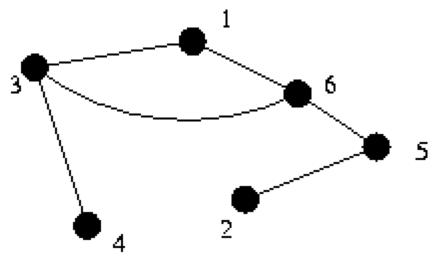
功效表	边集	邻接矩阵	邻接表
消耗空间	2xM	N^2	2xM
连接判断的消耗时间	M	1	d_{max}
检查某结点所有相邻结点的消耗时间	M	N	d_{max}
添加边的消耗时间	1	1	1
删除边的消耗时间	M	2	2xd_{max}

图的连通性

如果一个图的任意两个结点之间都有一条以上的路径相连，我们就称该图为***连通图***。例如下图就不是连通图，因为结点 2 到结点 4 之间没有路径相通。



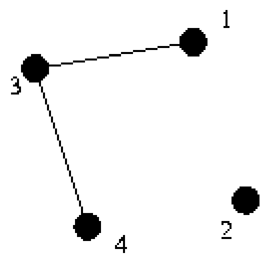
但是如果我们在该图的结点 5 和 6 之间添加一条边，该图就成为了连通图。



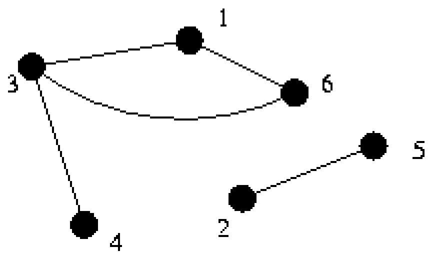
子图:

如果 G = (V, E)成立，且有集合 V’是 V 集合的子集，集合 E’是集合 E 的子集，那么我们就说 图 G’ = (V’, E’) 是图 G = (V, E) 的子图。

同时，在边集合 E’中出现的边必须是连接结点集合 V’中出现的两结点的边。我们考虑下图，它是例图的一个子图，其中 V’={1, 2, 3, 4,}，E’={(1, 3), (1, 4)}。但是如果我们在边集 E’中添加一条边(1, 6)，E’仍然是 E 的子集，但由于在 V’中没有出现结点 6，边(1, 6)就不可能在子图 G’ = (V’, E’)中出现。

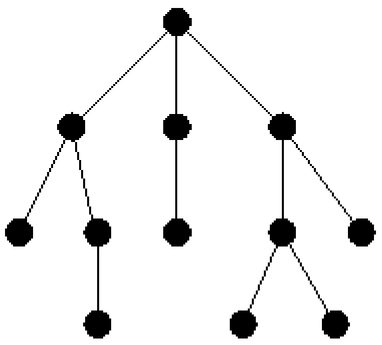


连通子图，如同它的名称一样，是一张图的子图，且该子图就有连通性，如下图，{1, 3, 4, 6}，{2, 5}，{1, 3, 4} ，{1，3}都是该图的连通子图（原文中在此仅标注了结点，我们尚且将边默认为所有这些结点之间的边——译者）。同时，我们还定义了***极大连通子图***，它的定义可以理解为将连通子图扩展的尽量大，在例图中仅有两个极大连通子图：{1, 3, 4, 6}和{2, 5}。非连通图的极大连通子图被称为***连通分量***；有向图中的连通图叫做***强连通图***；非强连通图中的极大连通子图被称为***强连通分量***。一个连通图的***生成树***是该图的*极小连通子图*，在有 N 个结点的情况下，生成树仅有 N-1 条边。



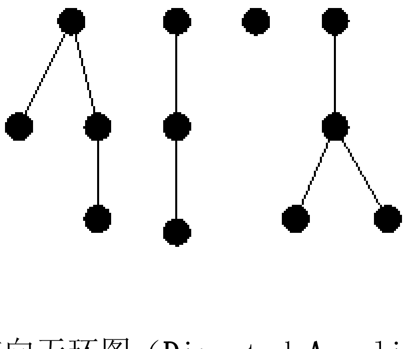
特殊的图：

连通且无环的无向图被称作***树***。



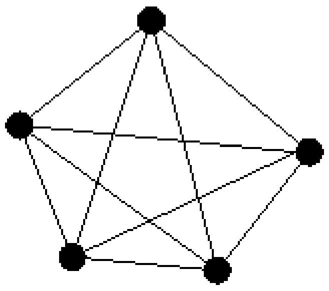
许多树都是“层次化”的。通常,树都有一个“最顶上的”结点,被称作***根结点***;相反，“最底下的”结点们被称作***叶结点***。除了根结点以外，所有结点都**仅**有一个***父结点***，即与它相连的那个上一层结点；除了叶结点以外，所有结点都**至少**有一个***子女结点***，即与它相连的所有下一层结点。同样，类推地，还有***祖先结点***、***后代结点***。经过这样的定义后，树的样式就可以构划得像一棵倒过来生长的树了（如上图）。

不连通且无环的无向图被称为森林（如下图）。他也可以理解为是有许多树构成的图。



有向无环图（Directed Acyclic Graph）被称为*DAG*，是他的英文缩写。

一个图的任意结点与其它所有结点有边相连的图被称为***完全图***。



如果一个图的所有结点可以被分为两个组，同组之间的任意结点都没有边相连，即所有边连接的都是不同组的两个结点，这样的图被称为***二分图***。