第六讲

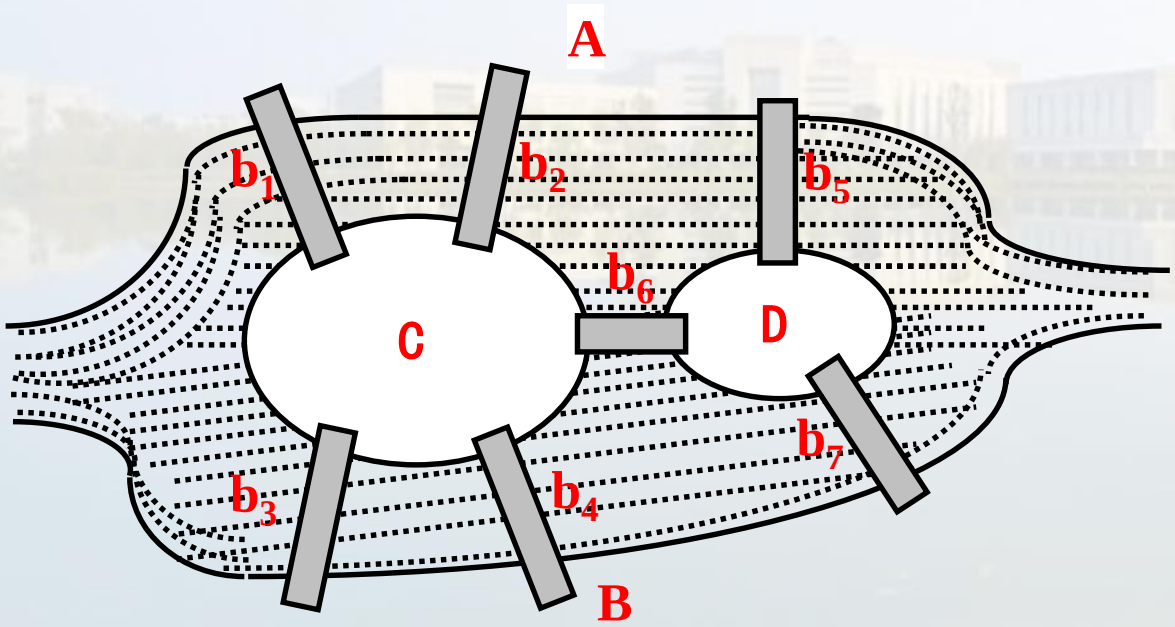
图论初步

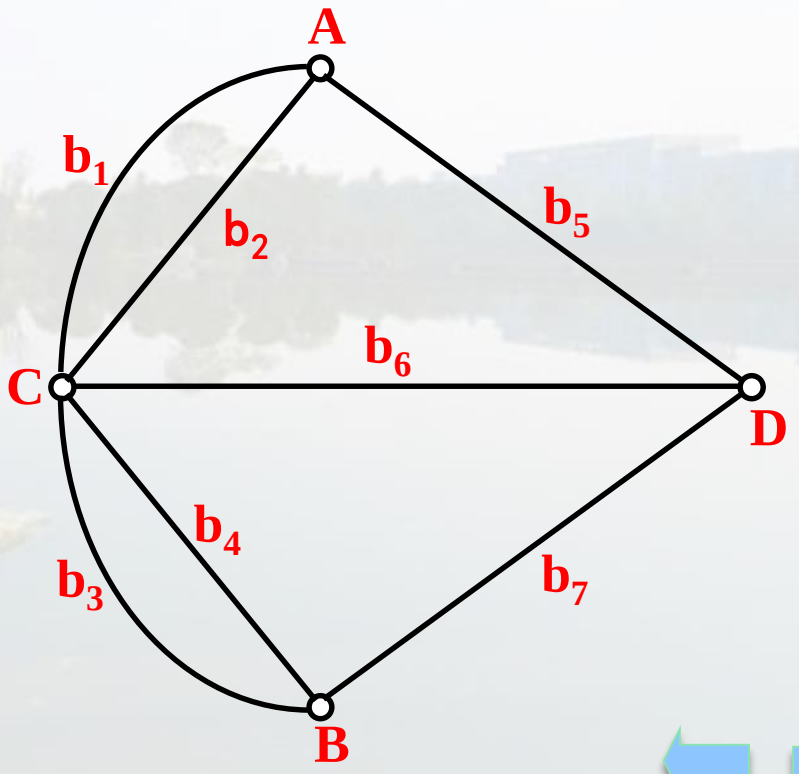
数学科学学院 汪小平

一、图论概述

图论起源于18世纪“**哥尼斯堡七桥问题**”. 在哥尼斯堡的一个公园里, 有七座桥将普雷格尔河中两个岛及岛与河岸连接起来(如图). 问是否可能从这四块陆地中任一块出发, 恰好通过每座桥一次, 再回到起点?

欧拉于1736年研究并解决了此问题，他把问题归结为如下图的"一笔画"问题，证明上述走法是不可能的。

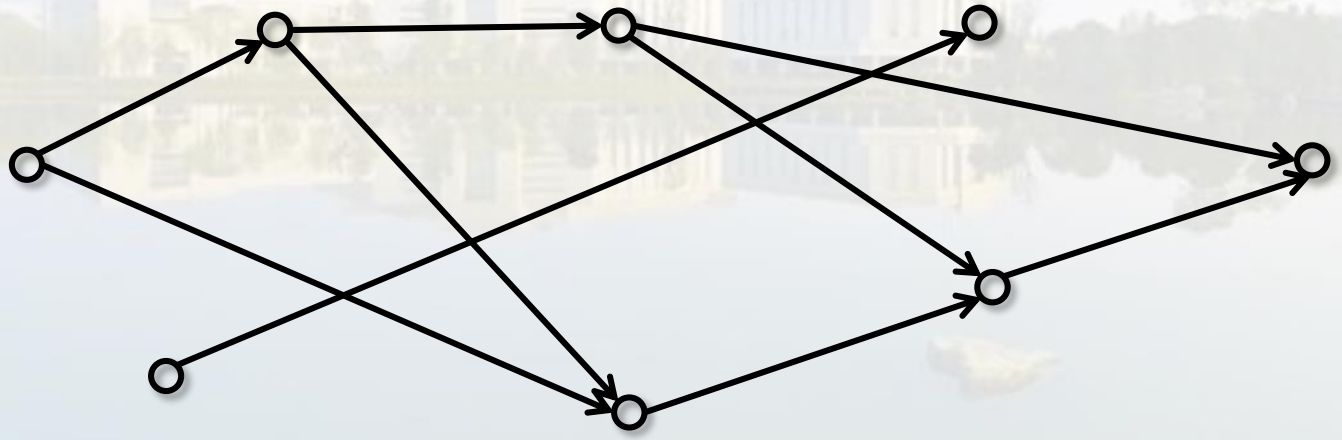




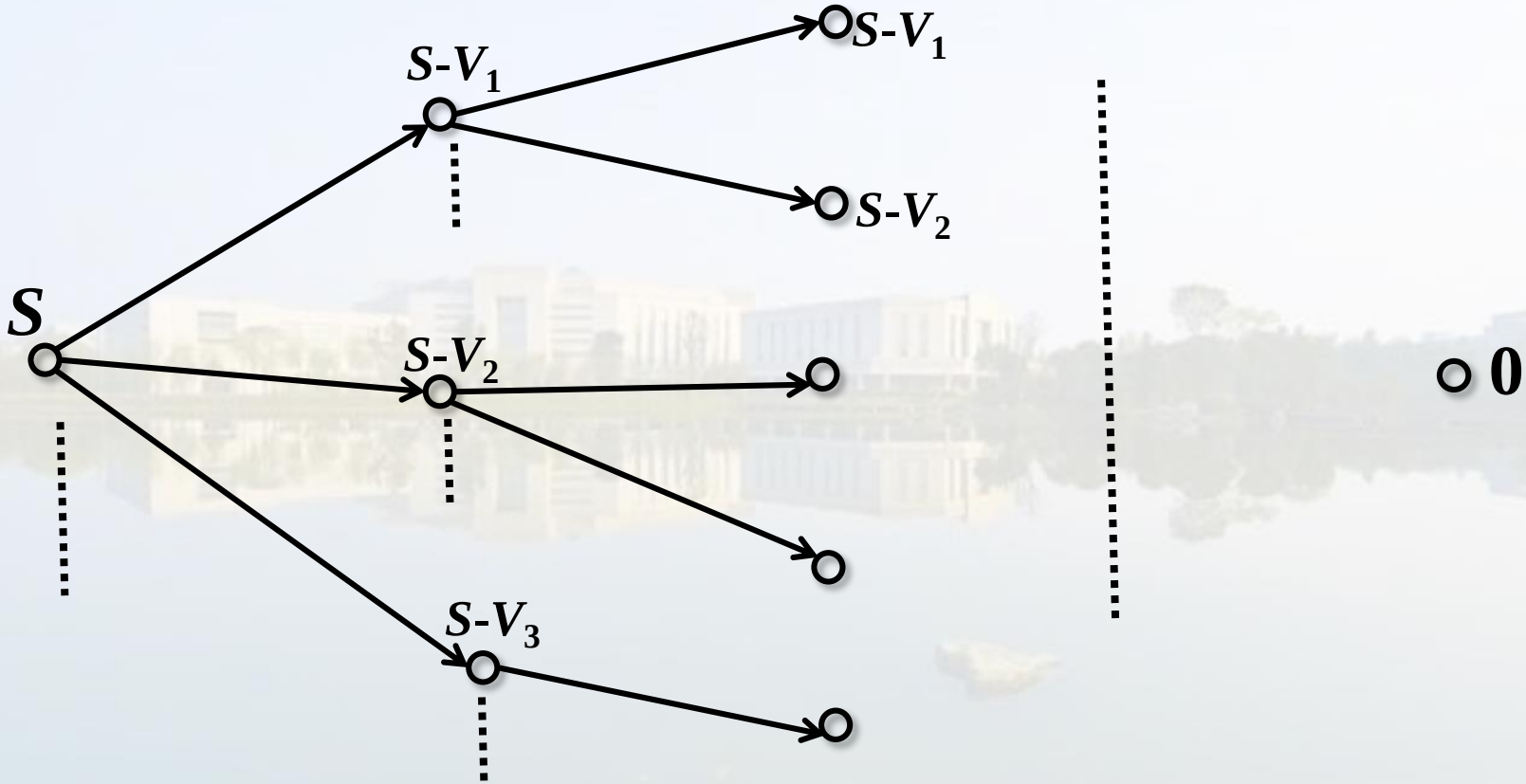
嵌套矩形问题：有n个矩形，每个矩形可以用两个整数a、b描述，表示它的长和宽。矩形X(a,b)可以嵌套在Y(c,d)中，当且仅当a<c,b<d，或者b<c,a<d（相当于把矩形X旋转90º)。

例如，(1,5)可以嵌套在(6,2)内，但不能嵌套在(3,4)内。

请选出尽量多的矩形排成一行，使得除了最后一个之外，每一个矩形都可以嵌套在下一个矩形内。



硬币问题：有n种硬币，面值分别为$V_1,V_2,\ldots,V_n$，每种都有无限多。给定非负整数S，可以选用多少个硬币，使得面值之和恰好为S？输出硬币数目的最小值和最大值。$1<=n<=100,0<=S<=10000,1<=V_i<=S$。

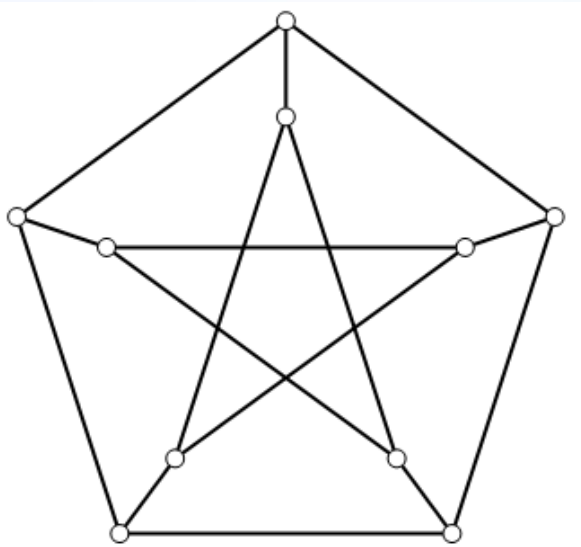


二、图的基本概念

1. 数学表示

图G是一个三元组：G=<V(G),E(G),φ(G)>，其中：

- V(G): 图G的结点集
- E(G): 图G的边集
- φ(G): E→V×V的关联函数



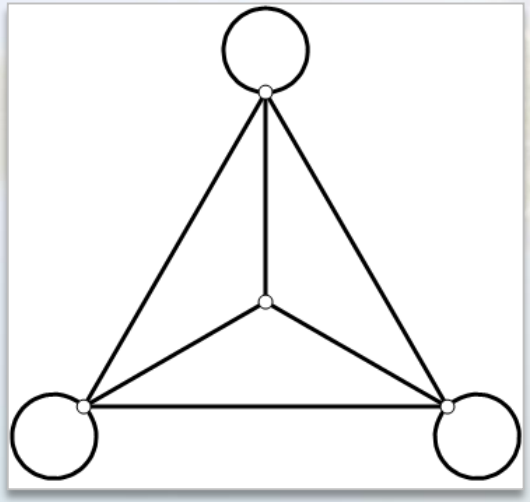
2. 边的概念

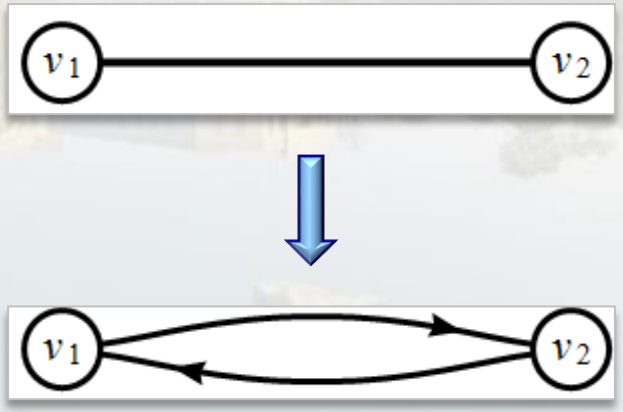
- 有向边：<i,j>表示以i为起点，j为终点的边。

- 无向边：(i,j)表示既能从i到j，又能从j到i的边。

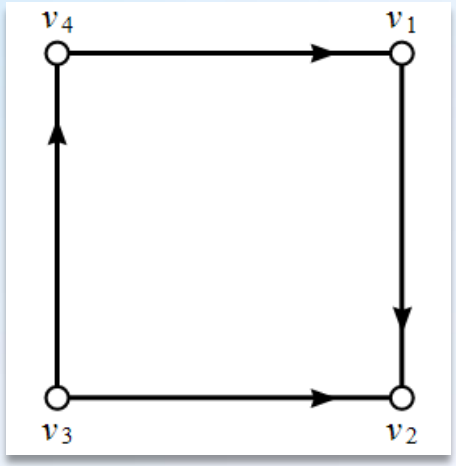
- 重边（平行边）：两个节点间方向相同的若干条边。

- 自环：自己连向自己的边。
- 对称边：两端点间方向相反的两条边，一条无向边可以拆成两条有向边（对称边）

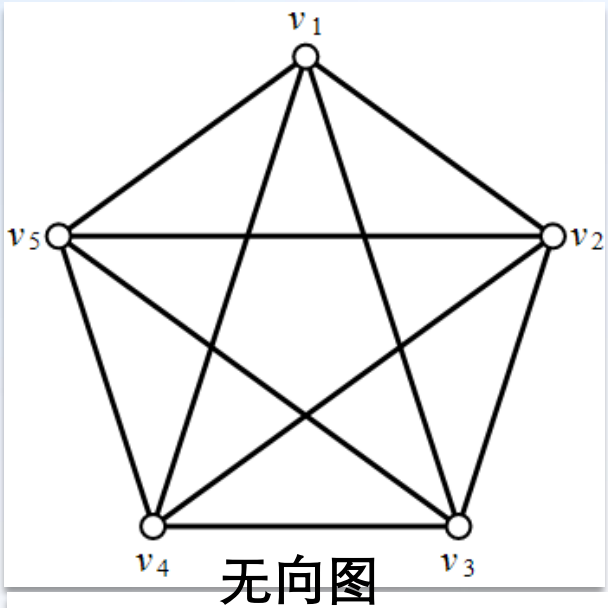




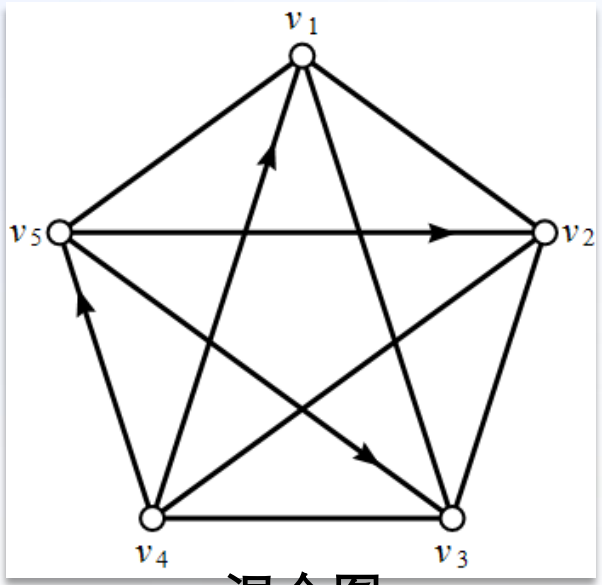
3. 图的分类



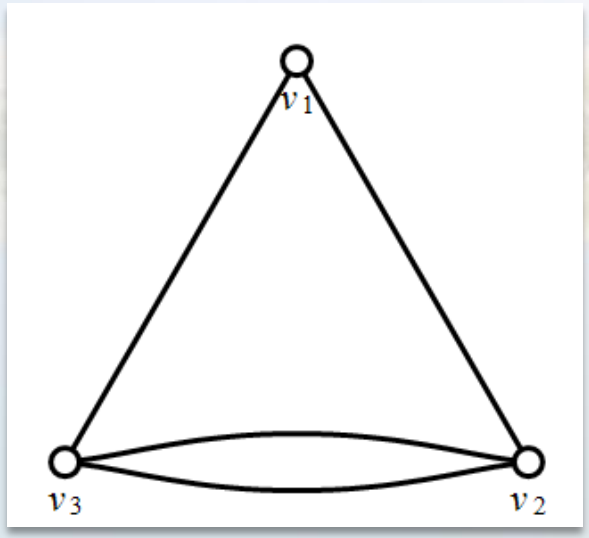
有向图



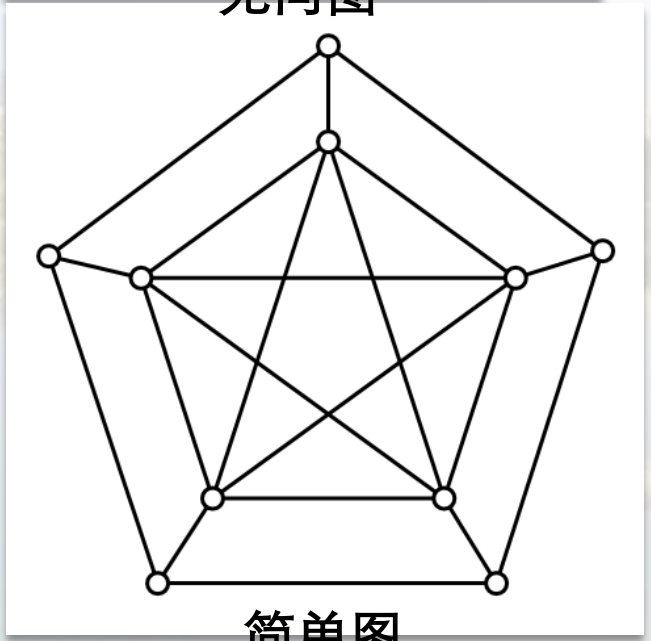
无向图



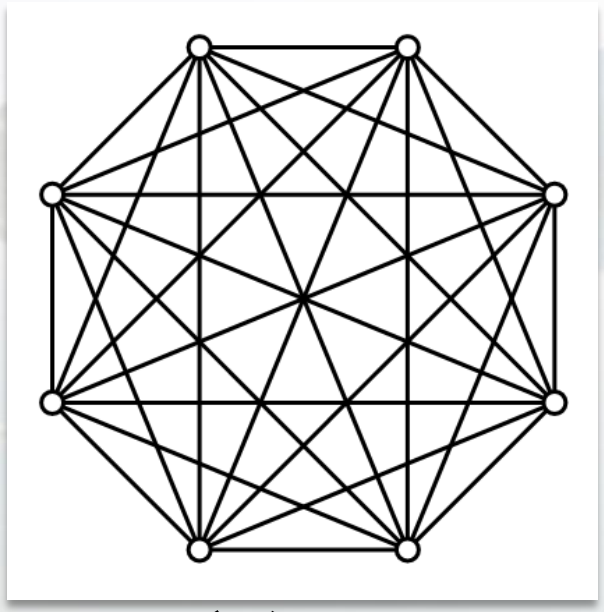
混合图



多重图



简单图
(无自环,无重边)



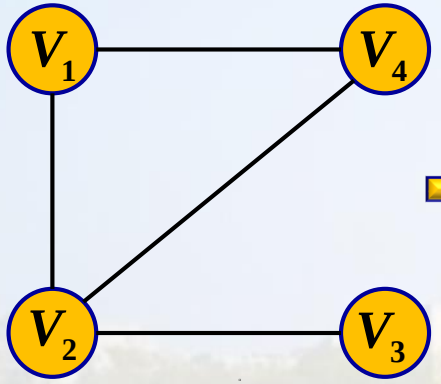
完全图K_8

稀疏图、稠密图

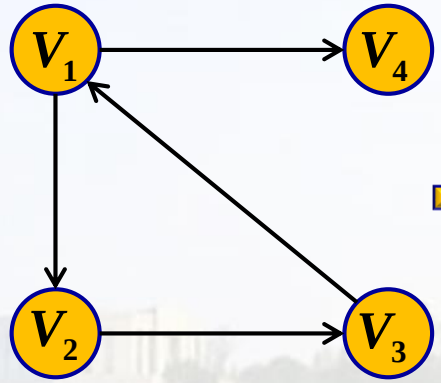
三、图的存储结构

1. 邻接矩阵

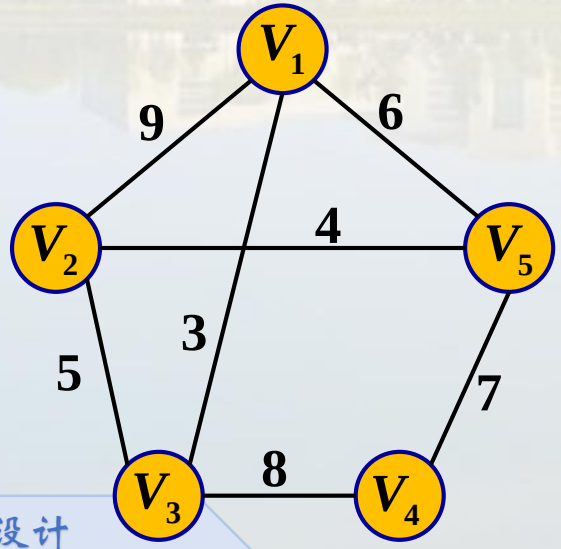
用一个一维数组存储图中顶点的信息, 用一个二维数组(矩阵)表示图中各顶点之间的邻接关系.



$$A=\begin{array}{c} \begin{matrix} V_1 & V_2 & V_3 & V_4 \end{matrix} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix} \end{array} \begin{matrix} V_1 \\ V_2 \\ V_3 \\ V_4 \end{matrix}$$

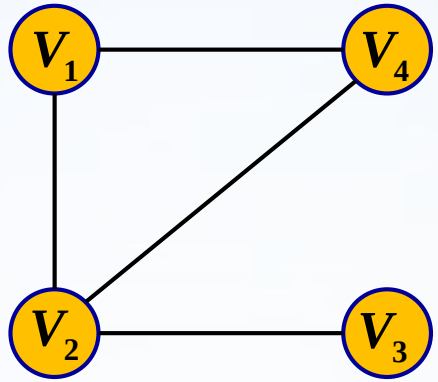


$$A=\begin{array}{c} \begin{matrix} V_1 & V_2 & V_3 & V_4 \end{matrix} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{array} \begin{matrix} V_1 \\ V_2 \\ V_3 \\ V_4 \end{matrix}$$



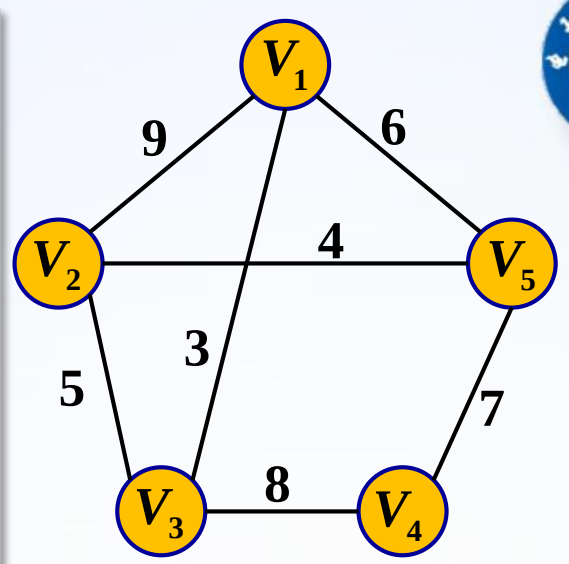
$$A=\begin{array}{c} \begin{matrix} V_1 & V_2 & V_3 & V_4 & V_5 \end{matrix} \\ \begin{bmatrix} 0 & 9 & 3 & \infty & 6 \\ 9 & 0 & 5 & \infty & 4 \\ 3 & 5 & 0 & 8 & \infty \\ \infty & \infty & 8 & 0 & 7 \\ 6 & 4 & \infty & 7 & 0 \end{bmatrix} \end{array} \begin{matrix} V_1 \\ V_2 \\ V_3 \\ V_4 \\ V_5 \end{matrix}$$

```
#define MAXN 100
int main()
{
    int n, m, u, v, map[MAXN][MAXN];
    memset(map, 0, sizeof(map));//初始化，元素清零
    scanf("%d%d", &n, &m);
    for(int i = 0; i < m; i++)
    {
        scanf("%d%d", &u, &v);
        map[u-1][v-1] = 1;   map[v-1][u-1] = 1;//无向图需要
    }
    //其它操作
    return 0;
}
```




```
4 4
1 2
1 4
2 4
2 3
```

```
#include <bits/stdc++.h>
using namespace std;
#define MAXN 100
#define INF 99999999
int main() {
    //考虑边带权的邻接矩阵(比如距离)
    int n, m, u, v, w, map[MAXN][MAXN];
    scanf("%d%d", &n, &m);
    for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
            if(i == j)    map[i][j] = 0;
            else          map[i][j] = INF;
    for(int i = 0; i < m; i++) {
        scanf("%d%d%d", &u, &v, &w);
        map[u-1][v-1] = w;
        map[v-1][u-1] = w;//无向图需要
    }
    //其它操作
    return 0;
}
```




```
5 7
1 2 9
1 3 3
1 5 6
2 3 5
2 5 4
3 4 8
4 5 7
```

```
//假设要遍历结点u的所有邻接边
for(int i = 0; i < n; i++)
{
    if(i != u && map[u][i] < INF)
        printf("%d ", map[u][i]);
}
```

图的深度优先遍历

```
#define MAXN 100
int n, m, visited[MAXN], graph[MAXN][MAXN];
void dfs(int k)
{
    visited [k] = 1; //表示已访问
    for(int i = 0; i < n; i++)
        if(! visited[i] && graph[k][i])
            dfs(i);
}
```

是很多算法的基础，比如可以实现连通块的搜索与计数。

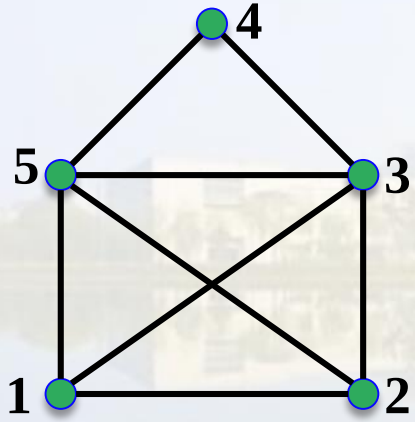
邻接矩阵的特点：

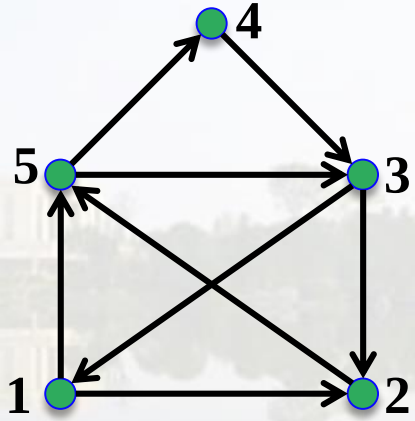
- 空间复杂度：$O(n^2)$
- 可以O(1)查询点对间的边数（或相邻情况）

邻接矩阵的缺点：空间复杂度大，处理稀疏图效率低，不便于处理多重图边上的附加信息。

The House Of Santa Claus

在我们的童年时代, 我们通常要解答圣诞老人的家谜(the riddle of the house of Santa Clasu). 你还记得吗? 要点就在于一笔把家画完, 而且一条边不能画两次. 圣诞老人的家如下图.

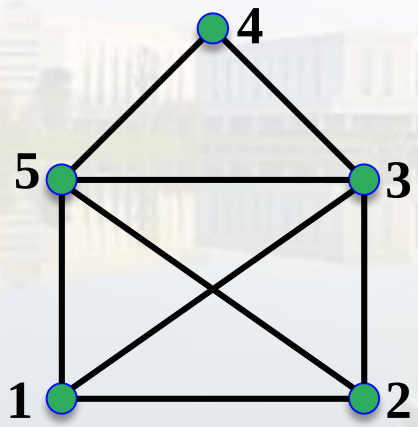




请你在计算机上“画出”这个房子. 因为不一定只有一种可能, 要求给出从左下方开始的所有可能, 并按递增顺序排列, 上面右图是一种可能画法(153125432).

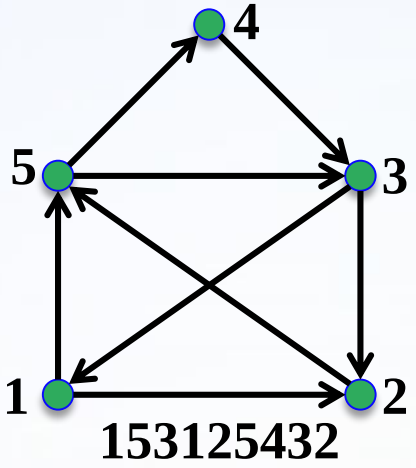
- **Samples**

Input	Output
无	123153452 123154352 154352312



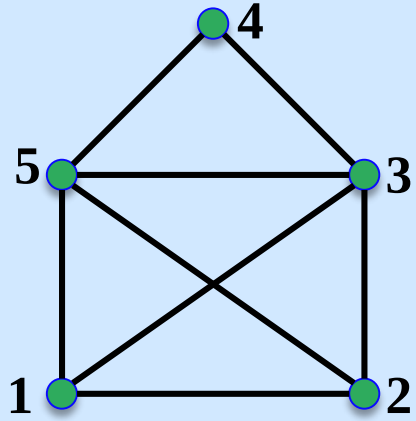
题目分析

- 圣诞老人的家是一个含8条边的无向图.
- 输出结果为9位数字.
- 从1按深度优先搜索(回溯法)即可得所有可能画法.
- 为保证结果递增, 在深度优先搜索访问邻结点时, 应按结点序号的递增顺序访问.

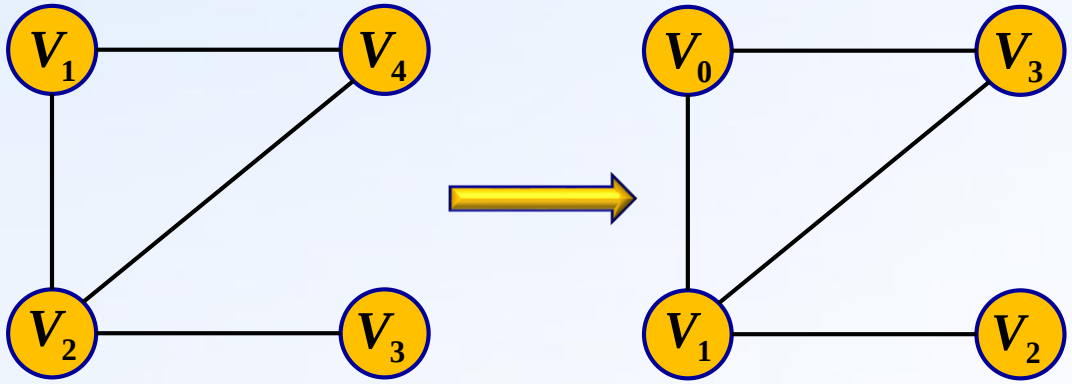


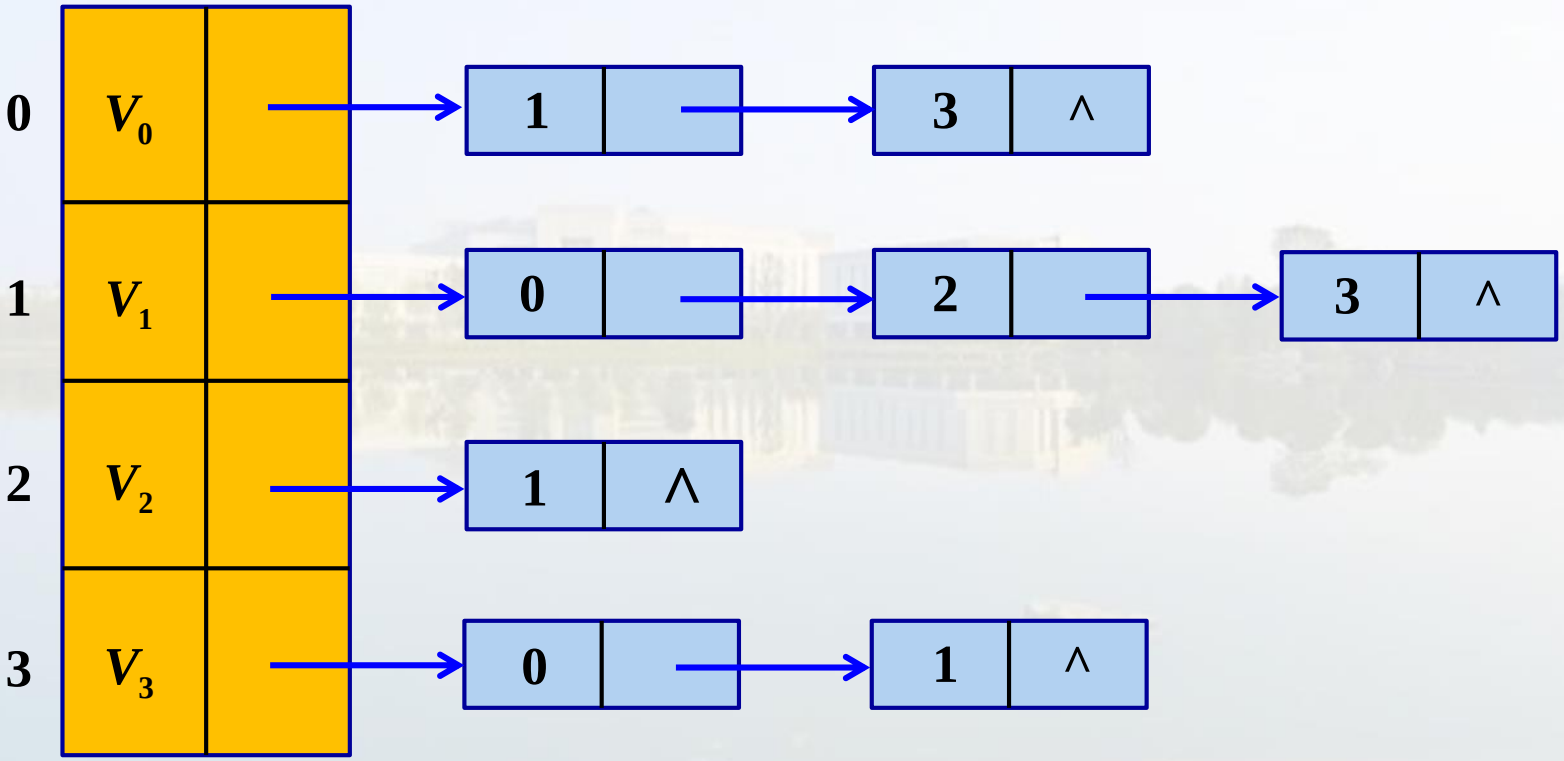
参考程序

```
#include <bits/stdc++.h>
using namespace std;
int house[6][6];
void makehouse()
{
    memset(house, 0, sizeof(house));
    for(int i = 1; i <= 5; i++)
        for(int j = 1; j <= 5; j++)
            if(i != j)  house[i][j] = 1;
    house[4][1] = house[1][4] = 0;
    house[4][2] = house[2][4] = 0;
}
void dfs(int x, int k, string s)
{
    s += char(x + '0');
    if(k == 8)
    {
        cout << s << endl;
        return;
    }
    for(int y = 1; y <= 5; y++)
        if(house[x][y])
        {
            house[x][y] = house[y][x] = 0;
            dfs(y, k+1, s);
            house[x][y] = house[y][x] = 1;
        }
}
int main()
{
    makehouse();
    dfs(1, 0, "");
    return 0;
}
```



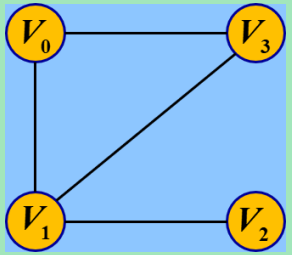
2. 邻接表

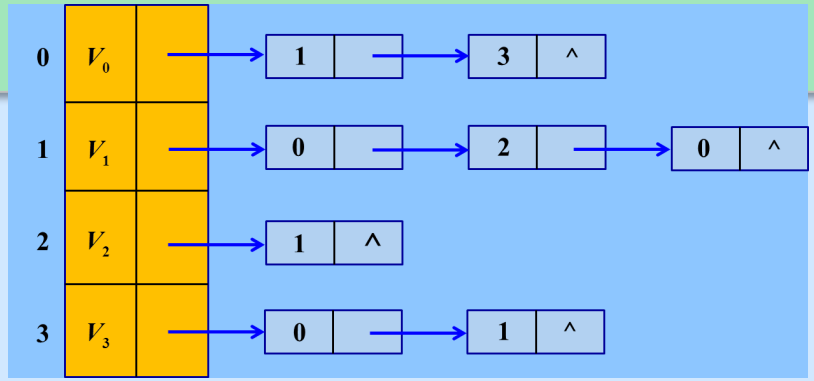




动态链表 - 动态分配结点

```
#include <bits/stdc++.h>
using namespace std;
#define MAXN 100
struct edge { //边结构体
    int nodeid;
    int edge_value;
    struct edge* next;
};
struct node { //结点结构体
    //int node_value;
    struct edge* next;
}mynode[MAXN];
int main() {
    int n, m, u, v, w;
    edge* newedge;
    scanf("%d%d", &n, &m);
    memset(mynode, 0, sizeof(mynode));
    for(int i = 0; i < m; i++) {
        scanf("%d%d%d", &u, &v, &w);
        newedge = new edge; //建新边
        newedge->nodeid = v; //记录边的一个端点
        newedge->edge_value = w; //记录边权
        newedge->next = mynode[u].next; //插入到u的邻接边中
        mynode[u].next = newedge;
        //若是无向图，则有对称边
        newedge = new edge;
        newedge->nodeid = u;
        newedge->edge_value = w;
        newedge->next = mynode[v].next;
        mynode[v].next = newedge;
    }
    //其它操作
    return 0;
}
```

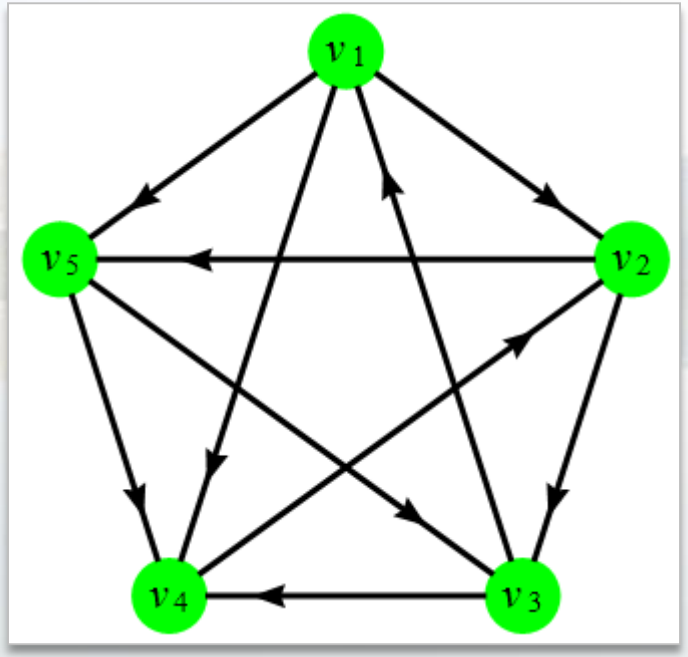




vector实现

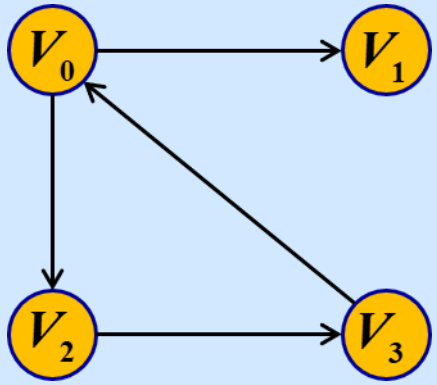
```
vector<vector<int> > mynode(5); // > >
for(int i = 0; i < 10; i++)
{
    scanf("%d%d", &u, &v);
    mynode[u].push_back(v);
}
```

1	2	4	5
2	3	5	
3	1	4	
4	2		
5	3	4	



```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int n, m, u, v;
    scanf("%d%d", &n, &m);
    vector<vector<int> > mynode(n); // > >
    for(int i = 0; i < m; i++) {
        scanf("%d%d", &u, &v);
        mynode[u].push_back(v);
        //mynode[v].push_back(u); //无向图需要加上
    }
    for(int i = 0; i < n; i++) {
        int k = mynode[i].size(); //邻接结点个数
        printf("%d-%d: ", i, k);
        for(int j = 0; j < k; j++)
            printf("%d ", mynode[i][j]); //邻接结点编号
        printf("\n");
    }
    return 0;
}
```

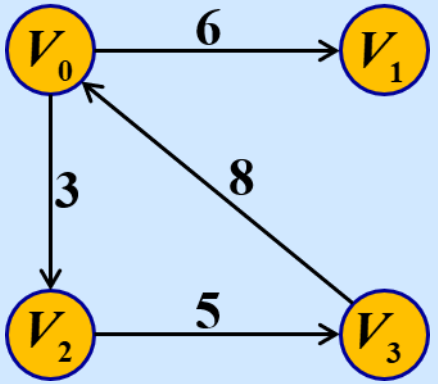
无权图




```
4 4
3 0
2 3
0 2
0 1
0-2: 2 1
1-0:
2-1: 3
3-1: 0
```

```
#include <bits/stdc++.h>
using namespace std;
struct edge {
    int nodeid, value;
};
int main() {
    int n, m, u, v, w;    edge temp;
    scanf("%d%d", &n, &m);
    vector<vector<edge> > mynode(n); // > >
    for(int i = 0; i < m; i++) {
        scanf("%d%d%d", &u, &v, &w);
        temp.nodeid = v;    temp.value = w;
        mynode[u].push_back(temp);
    }
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < mynode[i].size(); j++)
            printf("%d-%d(%d) ", i, mynode[i][j].nodeid, mynode[i][j].value);
        printf("\n");
    }
    return 0;
}
```

赋权图



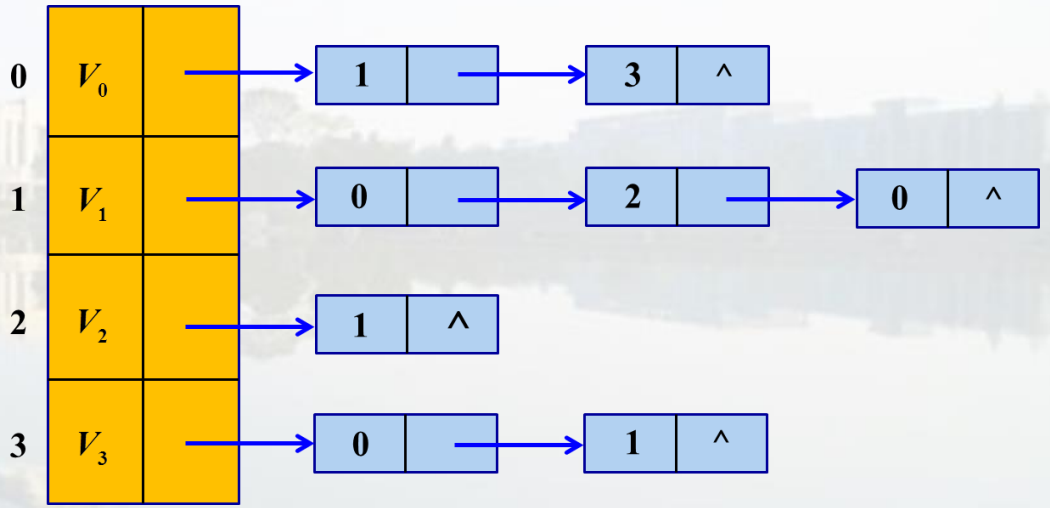

```
4 4
3 0 8
2 3 5
0 2 3
0 1 6
0-2(3) 0-1(6)

2-3(5)
3-0(8)
```

邻接表的特点：

- 空间复杂度： O(n+m)
- 可以高效的访问结点的所有邻接边（结点）
- 可以很好的处理重边
- 无法高效查询任意点对间的信息

邻接表还有一些其它实现方式：



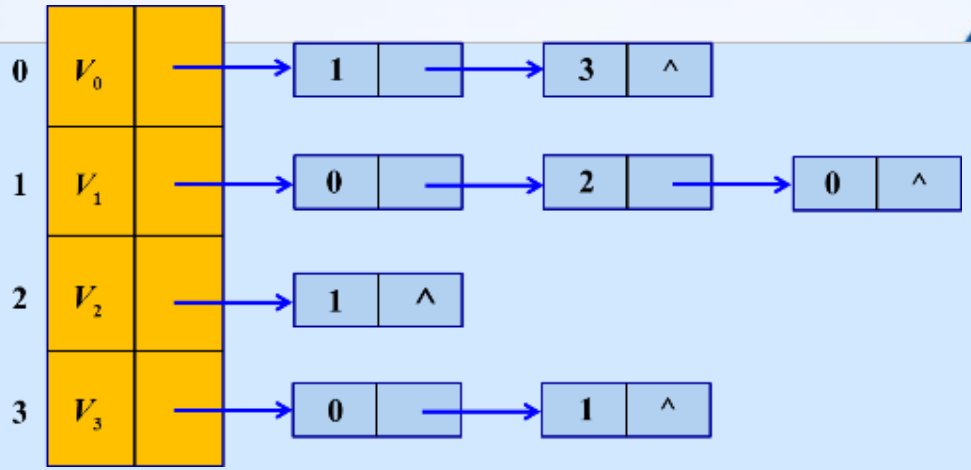
其他实现方法：静态链表

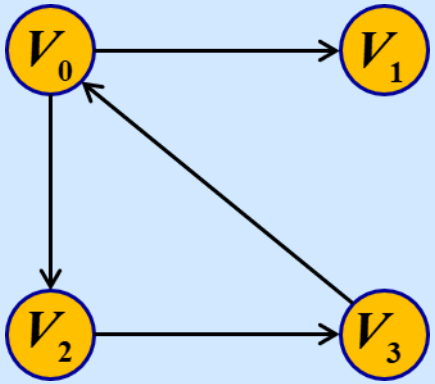
```
#include <bits/stdc++.h>
using namespace std;
#define MAXN 100
#define MAXM 1000
//本例代码针对有向图
int main() {
    int n, m, u, v;
    int node[MAXN], id[MAXM], w[MAXM], next[MAXM];
    memset(node, -1, sizeof(node));
    memset(next, -1, sizeof(next));
    scanf("%d%d", &n, &m);
    for(int i = 0; i < m; i++) {//i是边的编号
        scanf("%d%d%d", &u, &v, &w[i]);
        id[i] = v;
        next[i] = node[u];
        node[u] = i;
    }
    //其它操作
    return 0;
}
```

id[i]	w[i]	next[i]

id[i]	w[i]	next[i]

node[u] → | id[j] | w[j] | next[j] |






```
//假设要遍历结点u的所有邻接边
int e = node[u];
while(e != -1) {
    printf("%d ", w[e]);
    e = next[e];
}
```

```
#include <bits/stdc++.h>
using namespace std;
#define MAXN 100
#define MAXM 1000
struct edge {
    int nodeid;
    int edge_value;
    struct edge* next;
}myedge[MAXM];
struct node {
    //int node_value;
    struct edge* next;
}mynode[MAXN];
int main() {
    int k = 0, n, m, u, v, w;
    scanf("%d%d", &n, &m);
    memset(mynode, 0, sizeof(mynode));
    for(int i = 0; i < m; i++) {
        scanf("%d%d%d", &u, &v, &w);
        newedge[k].nodeid = v;
        newedge[k].edge_value = w;
        newedge[k].next = mynode[u].next;
        mynode[u].next = &newedge[k];
        k++;
        //若是无向图，则
        newedge[k].nodeid = u;
        newedge[k].edge_value = w;
        newedge[k].next = mynode[v].next;
        mynode[v].next = &newedge[k];
        k++;
    }
    //其它操作
    return 0;
}
```

```
//假设要遍历结点u的所有邻接边
struct edge* e = mynode[u].next;
while(e != NULL) {
    printf("%d ", e->edge_value);
    e = e->next;
}
```

动态链表 – 静态分配结点