图论基础

什么是图

图作为一种数据结构，表现的是若干对象的集合，以及这些对象间关系的集合。

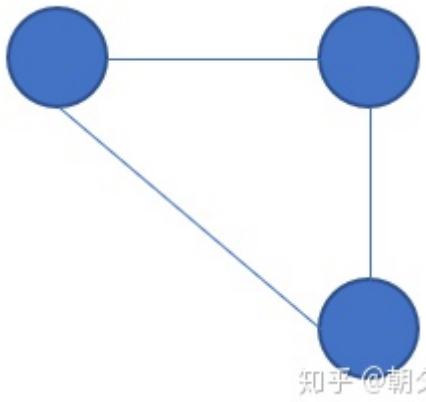
比如用图表现一个班级里各个同学的关系，则每个同学就是一个对象，所有的同学以及他们之间所有的关系的集合就是图。

图中的对象称为“结点”或“顶点”，一般用圆来表示，顶点间的关系称为“边”，用连线或箭头来表示。

图一般分为四种：

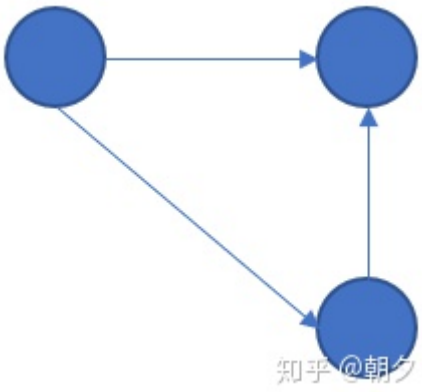
- **无向图：**边没有方向，以连线表示。

> 示例：A和B是朋友，则B和A也是朋友。调转初始与末尾结点后，关系不变。



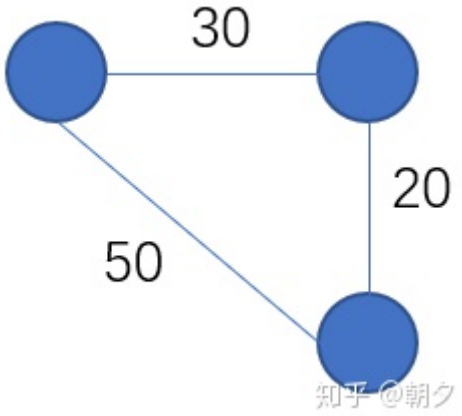
- **有向图：**边有方向，以箭头表示。

> 示例：要先学习A知识后才能学B知识。调转初始与末尾结点后，关系不同。



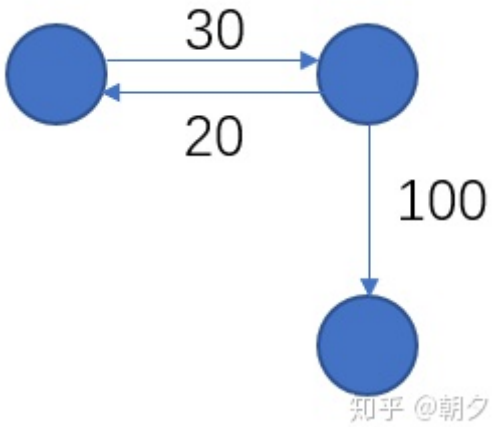
- **加权无向图：**边无方向，有权值。

> A与B直线距离50公里，B与A直线距离也是50公里。



- **加权有向图**：边有方向，有权值。

> 从A到B只能坐车，需要花费50分钟。从B到A只能步行，需要花费100分钟。



图的术语

边、点、权值

顶点集合为 V，边集合为 E 的图记为 $G=(V,\ E)$，其顶点数为 $|V|$，边数为 $|E|$。

连接顶点 a、b 的边记为 $e=(a,b)$。在有向图中 (a,b) 和 (b,a) 是不同的边。

边的**权值**记为 $w(a,b)$。

环路

无向图中若两顶点有边，则称两顶点**相邻**，相邻顶点构成的顶点序列称**路径**。起点若与终点相同，则称为**环**。

自环

若一条边的两个顶点为同一顶点,则此边称作**自环**。

简单图

既不含平行边也不含自环的图。

有向无环图

无环有向图的简称为“DAG”。

与顶点相连的边数称为**度（Degree）**。有向图的度又分为**入度**（以该顶点为终点）和**出度**（以该顶点为起点）。

连通图

任意两个顶点都是连通的，即都可从第一个顶点到第二个顶点，也可以从第二个顶点到第一个顶点。

完全图

任意两个顶点间都有一条边，无向完全图共有 $n \times (n+1)/2$ 条边，有向完全图共有 $n \times (n+1)$ 条边。

稀疏图

边数远小于完全图的图。

稠密图

边数接近于完全图的图。

图的存储方式

邻接矩阵

若有 n 个点，则开一个 $n \times n$ 的二维数组，其中一维表示起点，另一维则表示终点，其储存的值则可用于表示边的存在与否或者边的权值大小。若是无权图，一般边的权值默认为1。

基本模板

```
int e[maxn][maxn];
void add_edge(int start,int end,int v)
{
    //这里表示的是有向图
    //若是无向图则edge[start][end]=edge[end][start]=value
    //若无权值，则可用1表示有边，0表示无边
    edge[start][end]=value;
}
```

需要注意的是，邻接矩阵的空间复杂度为 $O(n^2)$，对于点较多的图论题，大概率会超出内存的限制，所以基本没什么用QWQ。

邻接表

std::vector

为了便利，我们邻接表使用C++标准库的vector容器来进行演示。

若有 n 个点，则开一个大小为 n 的vector数组，其中数组下标表示起点，vector中储存的为终点。若需要表示权值，可以使用结构体。

无向无权图模板：

```
vector<int> e[maxn];
int main(void) {
    int n, m;
    cin >> n >> m;
    init();
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        e[u].push_back(v);
        e[v].push_back(u);
    }
}
```

无向带权图模板：

```
struct node {
    int v, w;
    node(int const &v = 0, int const &w = 0) : v(v), w(w) { }
};
vector<node> e[maxn];

int main(void) {
    int n, m;
    cin >> n >> m;
    init();
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        e[u].push_back(node(v, w));
        e[v].push_back(node(u, w));
    }
}
```

链式前向星

链式前向星和邻接表类似，也是链表的链式结构和线性结构的结合。

每个结点**u**都有一个链表，链表的所有数据是从**u**出发的所有边的集合（对比邻接表存的是顶点集合），**边的表示为一个三元组(v, w, next)（无权图只有v和next）**，v表示的从节点u到临点v的边**(u, v)**，**w**代表边上的权值，**next**指向下一条边的编号。

带权图模板：

```
int head[maxn], cnt;

struct node {
    int v, w, next;
} e[maxn << 1];

void add(int u, int v, int w) {
```

```
    e[cnt].v = v;
    e[cnt].w = w;
    e[cnt].next = head[u];
    head[u] = cnt++;
}
void init() {
    memset(head, -1, sizeof(head));
    cnt = 0;
}

int main(void) {
    int n, m;
    cin >> n >> m;
    init();
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        add_edge(u, v, w);
        add_edge(v, u, w);
    }
}
```

图的遍历方式

DFS（深度优先搜索）

链式前向星

```
void dfs(int u, int fa){
    if (vis[u]) return;
    vis[u] = 1;
    for (int i = head[u]; ~i ; i = e[i].next){
        int v = e[i].v, w = e[i].w;
        if (fa == x)
            continue;
        dfs(v, u);
    }
}
```

vector

```
void dfs(int u, int fa){
    if (vis[u]) return;
    vis[u] = 1;
    for (int i = 0; i < (int)e[u].size(); i++){
        int v = e[u][i].v, w = e[u][i].w;
        if (fa == x)
            continue;
        dfs(v, u);
    }
}
```

BFS (广度优先搜索)

```
void bfs(int u) {
    queue<int> q;
    q.push(u);
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        if (vis[u]) continue;
        vis[u] = 1;
        for (int i = head[u]; ~i; i = e[i].next) {
            int v = e[i].v;
            if (!vis[v]) {
                q.push(v);
            }
        }
    }
}
```