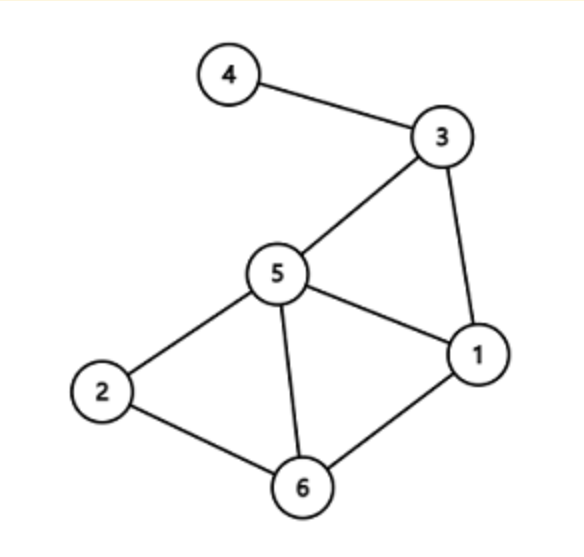
图论基础

——吴兆昕

图（graph）

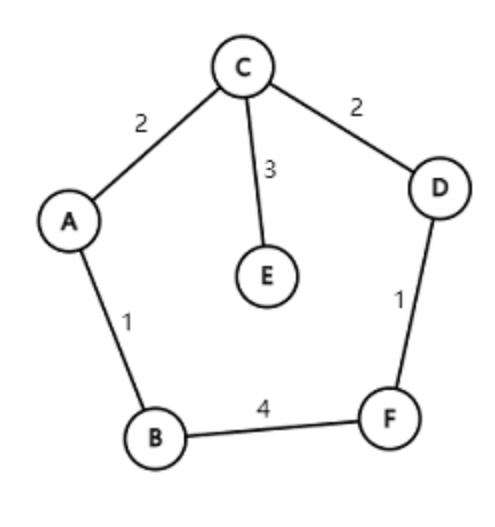
图论（graph theory）是数学的一个分支，图是图论的主要研究对象。

什么是图?
图可以简要理解为一些点和一些连接这些点构成的图形。
绘图网站，该网站可以由给定的边绘制出对应的图。



图（graph）

图由点（vertex）和边（edge）构成，点和边上都可以存储信息，一般来说，边上存储的信息为一个数值（权值），称为边权。
如下图可以表示一个地图，共6个点6条边，每个点分别有自己的名字，代表一个地点，每条边上有一个数值表示两个地点之间的距离，当然点上也能有权值，如每个地点的人数，称为点权（编号不是点权）。



概念

OI-wiki

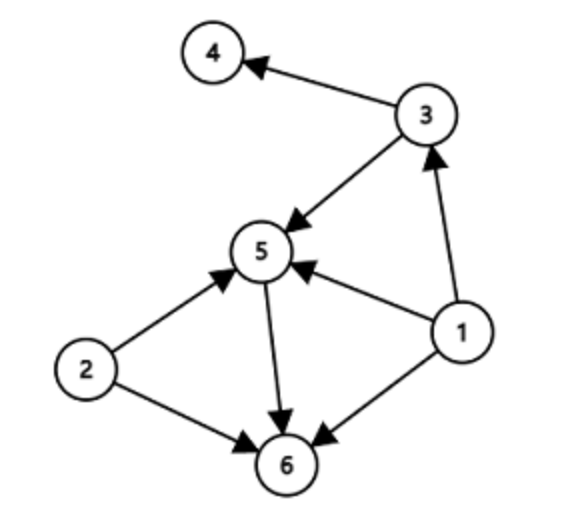
- 度数（degree）：起点（终点）为某个点的边的数量称为这个点的出度（入度），若边没有起终点之分则称度数。
- 路径：同字面意思，从一个点出发，沿着边，到达另外的一些点。
- 简单路径：不重复经过每一个点的路径。
- 回路：起点和终点相同的路径。
- 环：不重复经过点的回路。
- 三元环：共有三个点的环。
- 子图：由一个图中点某些点和某些边构成的新图称为原图的子图。

图的分类

根据图的特点，可以将图分为很多类，这里简要介绍。

有向图与无向图

如果一个图的边是有方向的（有向边），就称为有向图，用道路打比方则可以认为是单行线，反之则为无向图。



图的分类

简单图

无自环和重边的图。
自环，指起点和终点为同一个点的边。
重边，指存在两条边的起点和终点完全相同，若权值不同但起终点相同，通常也可以认为是重边。

连通图

从任何一个点出发，都能到达其他的任何点，则称为连通图。
一个不连通的图可以被分成很多个连通的部分，这些部分叫连通块。
常用来形容无向图。

图的分类

完全图

每个点到任意一个其它点都有一条边

有向无环图（DAG）

同字面意思。
一类非常特殊的图，每个点能到达的点和能被到达的点的交集为空。
如果把一条边看作某个点对另一个点会产生影响，那么有向无环图中的每个点都不会对能直接或间接影响到自己的点产生影响。
此类图可以使用递推（动态规划）解决问题。

图的存储

邻接矩阵

储存图最简单的方法为邻接矩阵，用一个二维数组`v[x][y]=z`，来直接表示存在一条从`x`到`y`且权值为`z`的边，可以用将`z`设为-1或无穷大等不会出现的数值来表示边不存在，也可以单独再开一个二维数组来存每条边是否存在。
代码略。

缺点：空间消耗较大，时间消耗较大，且无法表示重边。
开一个`5000*5000`的`int`数组需要消耗大约100Mb空间，所以一般只有点数小于`5000`，且边的数量非常多时才使用此方法。

图的存储

邻接表

对于每个点，把以这个点为起点的边都用某种方法存起来，就是邻接表。
这里同样可以使用二维数组，但注意对每个点要记录一共有多少条邻边（相邻的边）。
如果有权值再单独用同样的方法记录权值，或者可以用结构体。
可以使用动态开二维数组或指针分配内存的方法减少空间消耗（不如stl魔法）。

stl魔法-vector

`vector`，向量，在C++中常被用做动态数组，注意常数较大，能用普通数组的地方最好用普通数组。**动态数组小心越界！**

```cpp
#include<vector>
// 包含在万能头内
std::vector<int> a;
// 使用using namespace std可以省略前面的std::
// 定义一个int类型的vector变量a
a.push_back(1);
// 向a的末尾插入一个1
a[0]=2; // 字面意思，注意从0开始并且不要越界
int n = a.size(); // a的长度，类型为unsigned，注意0-1=inf
for(auto &i : a) {}
// 遍历a，i会依次变为a中的每一个数，c++11以上有效
```

图的存储

链式前向星

邻接表的一种，与前一种方法不同的是使用链式存储（链表）。

```
int head[100001] = {0}; // 表头
int nxt[200001], ver[200001], val[200001], tot = 0;
// 分别代表 链 终点 权值 当前边的总数
void add(int x, int y, int v) {
    ver[++ tot] = y, val[tot] = v, nxt[tot] = head[x], head[x] = tot;
    //ver[++ tot] = x, val[tot] = v, nxt[tot] = head[y], head[y] = tot;
}
for(int i = head[x]; i; i = nxt[i]) {
    int y = ver[i], v = val[i];
}
```

图的存储

链式前向星

若链式前向星的`tot`的初值设为1，并且加边的时候加双向边，那么编号为`i`的边对应的反向边为`i^1`（`^`：异或），该性质会在某些情况下用到。链式前向星也有结构体写法（略）。

直接存边

用三个一维数组或一个结构体数组或一个`pair<int, pair<int, int>>`数组直接存边。
直接存边的后两种方法可以便于将边按照权值排序。

图的遍历

DFS & BFS

就跟普通的搜索一样，对每个节点记录一个标记表示这个节点是否被访问过，然后从一个节点开始遍历。
一般来说遍历一个图使用DFS较多，图的遍历常用于无向图找连通块，有向图判断一个点是否能到达某个点。

BFS做一定修改后通常用于遍历DAG，被称为**拓扑排序**，每次将剩余点中所有入度为0的点入队，然后将队首的点相邻的所有边删除，然后队首出队。

DFS

以下代码记录每个点所属的连通块

```
int n;
bool vis[100001] = {0};
int bl[100001], cnt = 0; // 属于的连通块 连通块总数
void dfs(int x, int b) {
    vis[x] = 1, bl[x] = b;
    for(int i = head[x]; i; i = nxt[i])
        if(! vis[ver[i]]) dfs(ver[i], b);
}
void work() {
    for(int i = 1; i <= n; i ++) if(! vis[i]) dfs(i, ++ cnt);
}
```

拓扑排序

```
int n;
queue<int> q;
int dg[100001] = {0}; // 在加边时统计每个点的度数
void bfs() {
    for(int i = 1; i <= n; i ++)
        if(! dg[i]) q.push(i);
    while(! q.empty()) {
        int x = q.front();
        q.pop();
        for(int i = head[x]; i; i = nxt[i]) {
            dg[ver[i]] --;
            if(! dg[ver[i]]) q.push(ver[i]);
        }
    }
}
```

最短路

最短路即固定起点终点后，找到一条路径，使得这条路径上的边的权值和最小（即最短的路径）。

- Floyd（弗洛伊德）：求所有点两两之间的最短路，时间复杂度$O(n^3)$，不常用
- SPFA：求单点到所有点的最短路，时间复杂度最坏$O(nm)$，实际飞快，但近年来某些毒瘤出题人喜欢卡SPFA
- Dijkstra：求单点到所有点最短路，**边权必须非负**，堆优化后时间复杂度$O(nlogn)$，具体复杂度与使用的堆相关

Floyd

主要思想为每次合并两条最短路，合成的结果为新的最短路的待选。

记录每两个点之间的最短路`d[x][y]`。
现在有两条路径 x->...->k，k->...->y，合并之后为 x->...->k->...->y，如果这条路径比之前从x到y的路径短，那么从x到y的路径就更新为新找到的路径。

具体实现枚举`x,y,k`即可，注意`k`的枚举需要放在最外层循环，否则无法得出正确的结果。

```
for for for { d[x][y] = min(d[x][y], d[x][k], d[k][y]);}
```

Floyd

应用

- 求多源多汇（多起点多终点）最短路，复杂度较高，不常用。
- 求两两之间是否能到达，做法同最短路，用`d[x][y]`表示从x能到y，让`d[x][y] |= d[x][k] | d[k][y]`即可。 不常用，不如对每个点都dfs一遍。
- **求最小权值环**，一个最小权值环可以拆成一条最短路和一条边 x->y + y->...->x，求出最短路后，枚举每条边，和当前找到的最小环比较，如果权值和小于当前环，则将当前环设为新找到的环。

SPFA

SPFA：Shortest Path Faster Algorithm
本质为队列优化Bellman-Ford算法，可用于求单点到其它点的最短路，可以处理负边，并可以判断负环（如果出现负环那么就不存在最短路）。

考虑用当前图中的每一个点更新起点到别的点的最短路，用`dis[x]`表示从起点到x点的最短路长度，对于一条边 x->y，如果`dis[x]+val<dis[y]`，那么`dis[y]`就可以被更新为`dis[x]+val`，y的最短路被更新后，又可以用y的最短路去更新其它点的最短路，重复以上过程，直到没有点能再被更新为止（详细过程）。

SPFA

具体实现

相当于一个可以多次入队的广搜。

先将所有点的距离设为正无穷，然后将起点的距离设为0，然后采用队列，先将起点塞入队列。

然后每次用队首更新其它点，如果其它点被更新了，就把它加入队列，直到队列为空为止，如果一个点入队超过n次则说明有负环。

记录一下哪些点在队列中，可以避免队列中出现重复的元素。

```cpp
int dis[100001];
bool iq[100001] = {0}; // 是否在队列中
queue<int> q;
void spfa(int s) {
    memset(dis, 0x3f, sizeof(dis));
    dis[s] = 0, q.push(s);
    while(! q.empty()) {
        int x = q.front();
        q.pop();
        iq[x] = 0;
        for(int i = head[x]; i; i = nxt[i])
            if(dis[x] + val[i] < dis[ver[i]]) {
                dis[ver[i]] = dis[x] + val[i];
                if(! iq[ver[i]]) iq[ver[i]] = 1, q.push(ver[i]);
            }
    }
}
```

Dijkstra

Dijkstra算法用于解决非负权图上的单源（单起点）最短路问题，主要思想类似贪心。

考虑将点分为两个集合，一个集合为已经确定最短路的点，一个集合为未确定最短路的点，如果我们用前一个集合中的点去更新后一个集合中的点的最短路，我们就能确定后一个集合中某一个点的最短路，这个点就是当前距离最小的点。

使用反证法等多种方法都可以证明此方法的正确性。
求最小可以使用堆来完成。

Dijkstra

用一个数组记录每个点是否已经确定最短路，然后记录每个点当前的最短路长度（初始为inf）。

用一个堆，存每个点的当前最短路和编号，每次取出路径最短的点，然后用这个点更新其它点的最短路，如果成功更新，则将被更新的点塞入堆中，循环直到堆为空。

在取点的时候，如果遇到已经确定最短路的点，直接跳过即可，这样可以避免在堆中删除一个点再插入（因为一个点被更新前，在堆中会有一个节点，更新后又会有一个节点）。

```cpp
int dis[100001];
bool vis[100001] = {0};
priority_queue<pair<int, int> > q;
void dij(int s) {
    memset(dis, 0x3f, sizeof(dis));
    dis[s] = 0;
    q.push(make_pair(0, s));
    while(! q.empty()) {
        int x = q.top().second;
        q.pop();
        if(vis[x]) continue;
        vis[x] = 1;
        for(int i = head[x]; i; i = nxt[i])
            if(dis[x] + val[i] < dis[ver[i]]) {
                dis[ver[i]] = dis[x] + val[i];
                q.push(make_pair(-dis[ver[i]], x));
                // 此处取负来实现小根堆
            }
    }
}
```

end