2023香农先修班第八次课

图

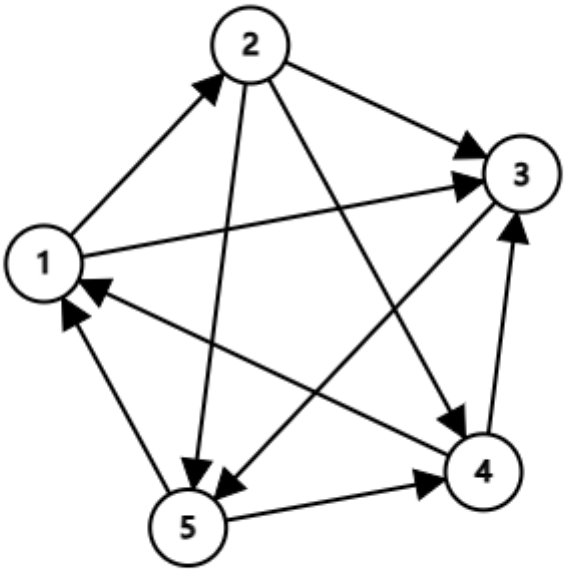
基础概念

图(Graph) 是由若干给定的顶点及连接两顶点的边所构成的图形，这种图形通常用来描述某些事物之间的某种特定关系。顶点用于代表事物，连接两顶点的边则用于表示两个事物间具有这种关系。

- **有向图(Directed Graph)**：边均为单向的有向边的图。
- **无向图(Undirected Graph)**：边均不具有方向性的图。
- **前驱(Tail)**、**后继(Head)**、入边、出边：对于有向边 $e = u \to v$，u 称为该有向边 e 的**前驱**（或**起点**），v 则为**后继**（**终点**），而 e 视为 u 的出边、v 的入边。
- 图的阶：即图的顶点数。
- **度(Degree)**、**入度(In-degree)**和**出度(Out-degree)**：无向图上某个顶点的邻边数称为这个顶点的**度数**；在有向图中，顶点的入边数为该顶点的**入度**、出边数为该顶点的**出度**。
- **自环(Loop)**：对于边 $e = u \to v$，若 $u = v$，则称 e 为一个**自环**。
- **重边(Multiple Edge)**：对于 $e_i = u_i \to v_i, e_j = u_j \to v_j \in E(i \neq j)$，若 $u_i = u_j$ 且 $v_i = v_j$，则称 e_i, e_j 为一组**重边**（**平行边**）。
- **途径(Walk)**：**途径**指图上的一个边集 E，$\forall e_i = u_i \to v_i \in E, u_i = v_{i-1}, v_i = u_{i+1}$。
- **轨迹(Trail)**：对于一个途径 W，若 $\forall e_i, e_j \in W(i \neq j), e_i \neq e_j$，则可称这条途径为轨迹。
- **简单路径(Path)**：对于一个轨迹，若其经过的顶点没有重复，则可称为一条**简单路径**。
- **回路(Circuit)**：对于一个轨迹，若其起始点和终点相同，则称为一个**回路**。
- **环(Cycle)**：对于一个回路，若其经过的顶点只有起始点出现了两次，则称为一个**环**（或**简单回路**、**简单环**）。
- **简单图(Simple Graph)**：不含重边和自环的图。
- **有向无环图(Directed Acyclic Graph)**：不含环的有向图，简称为 **DAG**。

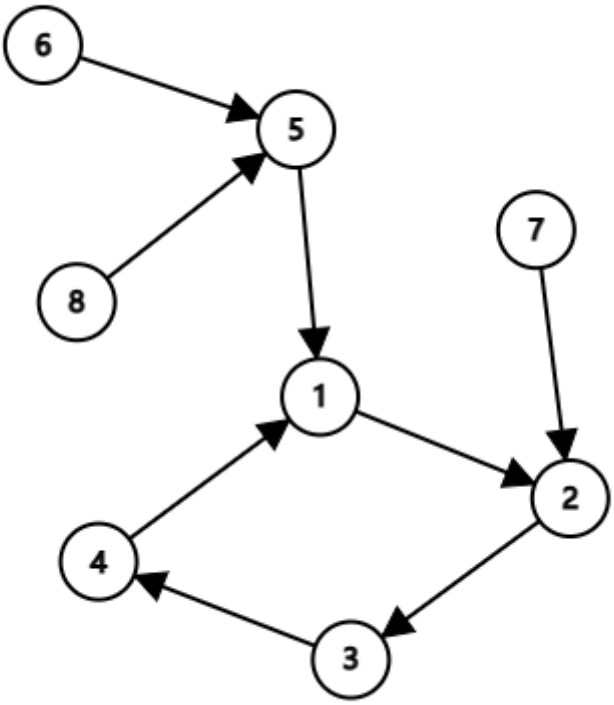
特殊的图

- **无向完全图(Complete Graph)**：任意两个顶点间均有连边的完全图。
- **有向完全图(Complete Digraph)**：任意不同两点间都有两条方向不同的边的有向图。
- **竞赛图(Tournament Graph)**：任意不同两点间都恰有一条有向边的有向图。



一个五阶竞赛图

- **零图(Null Graph)**：0个顶点的图。
- **平凡图**：1个顶点的图。
- **基环树(Pseudo Tree)**：恰好包含一个环的无向图。若一棵基环树每个点入度都为 1，则是一棵**基环外向树**；若每个点出度都为 1 则是一棵**基环内向树。**



一棵基内向环树

拓扑排序

拓扑排序是将一个 DAG 中的顶点以线性方式进行排序，使得对于任何的顶点 u 到 v 的有向边 $e = u \to v$，都可以有 u 在 v 的前面，将所有节点排序后，若 u 到 v 间存在一条有向边 $e = u \to v$，则 v 必定排在 u 的后面。

拓扑排序算法选择所有入度为 0 的节点为起点（顺序随机，按题意要求可能有一定顺序，如字典序）进行 BFS，当当前边指向的节点的所有前驱节点都已被访问，则可以将该节点加入队列等待访问。

示例代码

这里及下面其他示例用的都是链式前向星存图，大家可试试自行实现下邻接表的拓扑排序及其他算法。

```
vector<int> fa(maxn, 0);
queue<int> q;

void init()
{
    for (int j = 1; j <= n; j++)
        if (!fa[j])
            q.push(j);
}

void bfs(int p)
{
    cout << p << " ";

    for (int j = head[p]; j; j = edges[j].next)
        if (!(--fa[edges[j].to]))
            q.push(edges[j].to);

    q.pop();
    if (!q.empty())
        bfs(q.front());
}
```

最短路

不同方法的比较 ¶

最短路算法	Floyd	Bellman-Ford	Dijkstra	Johnson
最短路类型	每对结点之间的最短路	单源最短路	单源最短路	每对结点之间的最短路
作用于	任意图	任意图	非负权图	任意图
能否检测负环?	能	能	不能	能
推荐作用图的大小	小	中/小	大/中	大/中
时间复杂度	$O(N^3)$	$O(NM)$	$O(M \log M)$	$O(NM \log M)$

注：表中的 Dijkstra 算法在计算复杂度时均用 `priority_queue` 实现。

单源最短路

~~关于 SPFA，它已经死了~~

Dijkstra 算法

将结点分成两个集合：已确定最短路长度的点集（记为 S 集合）的和未确定最短路长度的点集（记为 T 集合）。一开始所有的点都属于 T 集合。

初始化 $dis(root) = 0$，其它点设为 INF。

然后重复下面两个操作直到 T 为空：

1. 从 T 集合中，选取一个最短路长度最小的结点，移到 S 集合中。
2. 对那些刚刚被加入 S 集合的结点的所有出边 $e = u \to v$ 执行松弛操作，即 $dis(v_i) = \min(dis(v_i), dis(u) + w(e))$。

或者说，遍历到某个点时，就更新该点所有出边的直接后驱的最短路长度，并选择最短路长度最小的未被遍历的点作为下一个被遍历的点。

Dijkstra 算法本身复杂度其实是 $O(n^2)$，有多种优化能大幅降低此复杂度，其中常用的是用优先队列 `priority_queue` 优化（复杂度为 $O(m \log m)$），此外有二叉堆、斐波那契堆、线段树优化等，大家可自行了解。

示例代码

```
int cnt = 1;
vector<int> vis(maxn, 0), dis(maxn, inf);
priority_queue<pll> q; //用pair<LL, LL>方便记录节点。

void init(int p) //初始化需要把源点的距离设为0，并将其入队。
{
    dis[p] = 0;
    q.push({ 0, p });
}

void dijkstra(int p)
```

```
{
    vis[p] = 1;
    cnt++;
    if (cnt >= n)
        return;

    for (int j = head[p]; j; j = edges[j].next)
        if (!vis[edges[j].to] && dis[edges[j].to] > dis[p] + edges[j].w)
        {
            dis[edges[j].to] = dis[p] + edges[j].w;
            q.push({ -dis[edges[j].to], edges[j].to }); //优先队列默认为大根堆，将数
取相反数以当成小根堆使用。
        }

    while (!q.empty() && vis[q.top().second])
        q.pop();
    if (!q.empty())
        dijkstra(q.top().second);
}
```

多源最短路

多源最短路问题一般较少，大部分的数据范围都使得能使用 Floyd 算法进行解决。

Floyd 算法

Floyd 算法本质上是 dp，通过枚举断点和边求出结果，其复杂度达到 $O(n^3)$，较少用到，但还是会用到，例如刚过去的周日这场团队训练赛的 F 题就用到了。

使用 Floyd 算法需要用邻接矩阵进行存图。

示例代码

```
for (int k = 1; k <= n; k++)
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            if (f[i][j] > f[i][k] + f[k][j])
                f[i][j] = f[i][k] + f[k][j];
```

最小生成树

最小生成树(Minimum Spanning Tree)是无向连通图上边权和最小的生成树。

求最小生成树

Kruskal 算法

Kruskal 算法是通过重复贪心地选择最小的可用边来构造最小生成树的算法。实现时使用并查集维护节点是否已经连通，若边的两端的两个节点未连通，则可以把当前边加入最小生成树。Kruskal 算法的复杂度主要是边排序的复杂度，为 $O(m \log m)$。

示例代码

```
int dsu[maxn]; //并查集
pll es[maxm];

int find(int x)
{
    return x == dsu[x] ? x : x = find(dsu[x]);
}

void _union(int x, int y)
{
    dsu[find(x)] = find(y);
}

void kruskal()
{
    for (int i = 1; i <= m; i++)
        es[i] = { edges[i * 2 - 1].w, i * 2 - 1 }; //链式前向星存无向图时相邻两个下标位置代表一条无向边。
    sort(es + 1, es + m + 1);

    for (int i = 1; i <= m; i++)
        if (dsu[edges[es[i].second].to] != dsu[edges[es[i].second + 1].to])
        {
            edges[es[i].second].onTree = edges[es[i].second + 1].onTree = 1;
            _union(edges[es[i].second].to, edges[es[i].second + 1].to);
        }
}
```

Prim 算法

Prim 算法也是基于贪心的算法，实现方式类似于单源最短路的 Dijkstra 算法。若用优先队列进行优化，复杂度也大概为 $O(m \log m)$。

示例代码

```
int cnt = 1;
vector<int> vis(maxn, 0);
priority_queue<pll> q;

void prim(int p)
{
    vis[p] = 1;
    cnt++;
    if (cnt >= n)
        return;

    for (int j = head[p]; j; j = edges[j].next)
        if (!vis[edges[j].to])
            q.push({ -edges[j].w, j });

    while (!q.empty() && vis[q.top().second.second])
        q.pop();
    if (!q.empty())
```

```
    {
        edges[q.top().second].onTree = edges[q.top().second - 1 +
((q.top().second & 1) << 1)].onTree = 1;
        prim(edges[q.top().second].to);
    }
}
```

连通性

概念

- **连通(Connected)**和**连通图(Connected Graph)**：对于一个无向图，若存在一条途径使得顶点 u 能到达 v，则称 u 和 v **连通**，所有顶点两两互相连通的无向图为**连通图。**
- **可达**：对于一个有向图，若存在一条途径使 u 能到达 v，则称 u **可达** v。
- **强连通(Strongly Connected)**和**弱连通(Weakly Connected)**：一张所有顶点两两互相可达的有向图为**强连通**，把所有边替换成无向边后能得到一张连通图的有向图为**弱连通。**
- **连通分量(Connected Component)**：一张图无向图的一个子图上的顶点若两两互相连通，这个子图就是这张图的一个**连通分量。**
- **强连通分量(Strongly Connected Component)**和**弱连通分量(Weakly Connected Component)**：即强连通的子图和弱连通的子图。

强连通分量与缩点

求 scc 常常在判环或者在有环图上 dp 时用到。缩点是把一个强连通分量上的边权或其它有关权值计算合并后把整个强连通分量作为一个点以方便求解的做法。

Tarjan 算法

Tarjan 算法基于 dfs 生成树，在访问节点时为节点 p 时打上时间戳 dfn_p，同时维护节点 p 在其 dfs 生成子树中能抵达的最早被访问的节点的时间戳 low_p。low 值相等的节点位于同一个 scc 中。

通过将访问过的节点入栈已快速获得位于同一个 scc 的节点。

当位于节点 u 并访问其相邻节点 v 时：

1. v 未被访问：向 v 继续 dfs 并回溯用 low_v 更新 low_u；
2. v 已入栈：用 dfn_v 更新 low_u；
3. v 已出栈：不进行操作。

按以上思路可以将算法写成代码。

示例代码

```
vector<int> vis(maxn, 0), dfn(maxn, 0), low(maxn, 0), scc(maxn, 0), sta;

void tarjan(int p, int f)
{
    vis[p] = 1;
    dfn[p] = low[p] = dfn[f] + 1;
    sta.push_back(p);

    for (int j = head[p]; j; j = edges[j].next)
    {
```

```
        if (!vis[edges[j].to])
        {
            tarjan(edges[j].to, p);
            low[p] = min(low[p], low[edges[j].to]);
        }
        else if (vis[edges[j].to] == 1)
            low[p] = min(low[p], dfn[edges[j].to]);
    }

    if (dfn[p] == low[p])
    {
        while (sta.back() != p)
        {
            scc[sta.back()] = p;
            vis[sta.back()] = 2;
            for (int j = head[sta.back()]; j; j = edges[j].next) //缩点
                add(p, scc[edges[j].to]);
            sta.pop_back();
        }
        scc[p] = p;
        vis[p] = 2;
        sta.pop_back();
    }
}
```

割点与桥

对于一个连通无向图，如果把某一个顶点删除后图变的不连通，则该顶点为 **割点**（或**割顶**），若把某条边删除后图变的不连通，则称这条边为**桥**（或**割边**）。**桥**两端的顶点若有其它相邻节点，则该顶点为**割点**。

双连通分量

在一张连通无向图中，对于两个点 u 和 v，如果无论删去哪一条边都不能使它们不连通，则说 u 和 v **边双连通(Edge Biconnected)**；如果无论删去哪一个非 u, v 的节点都不能使 u 和 v 不连通，则说 u 和 v **点双连通(Point Biconnected)**。

一张连通无向图的某个子图内若顶点两两**边/点双连通**，这个子图就是一个**边/点双连通分量(Edge/Point Biconnected Component)**。

利用 Tarjan 算法判割点和桥求 BCC

求 BCC 可以先用 Tarjan 算法求出边双连通分量，并在此基础上对每个边双连通分量求取割点得到双连通分量。

示例代码

```
int cnt = 0;
vector<int> vis(maxn, 0), dfn(maxn, 0), low(maxn, 0), bcc(maxn, 0), cv(maxn, 0),
ce(maxm, 0), sta;

void tarjan(int p, int f) //求边双连通分量同时判割点
{
    vis[p] = 1;
    dfn[p] = low[p] = dfn[f] + 1;
```

```
    sta.push_back(p);

    for (int j = head[p]; j; j = edges[j].next)
        if (edges[j].to != f) //求边双连通分量只需要在求SCC的基础上加上此判定。
        {
            if (!vis[edges[j].to])
            {
                tarjan(edges[j].to, p);
                low[p] = min(low[p], low[edges[j].to]);

                if (!f)
                    cnt++;
                if (f && low[edges[j].to] >= dfn[p]) //判割点
                    cv[p] = 1;
            }
            else if (vis[edges[j].to] == 1)
                low[p] = min(low[p], dfn[edges[j].to]);
        }

    if (dfn[p] == low[p])
    {
        while (sta.back() != p)
        {
            bcc[sta.back()] = p;
            vis[sta.back()] = 2;
            sta.pop_back();
        }
        bcc[p] = p;
        vis[p] = 2;
        sta.pop_back();
    }

    if (!f && cnt > 1) //特判第一个点是否为割点
        cv[p] = 1;
}

void dfs1(int p, int f) //第一次dfs判桥
{
    vis[p] = 3;

    for (int j = head[p]; j; j = edges[j].to)
    {
        if (bcc[p] != bcc[edges[j].to])
            ce[j] = 1;
        if (vis[edges[j].to] == 2)
            dfs1(edges[j].to, p);
    }
}

void dfs2(int p, int f) //再做二次dfs判断点双连通分量
{
    vis[p] = 0;
    for (int j = head[p]; j; j = edges[j].next)
        if (vis[edges[j].to])
        {
```

```
            sta.push_back(j); //将边入栈
            dfs2(edges[j].to, p);

            if (cv[p] && low[edges[j].to] >= dfn[p]) //发现p是割点
            {
                while(edges[sta.back()].to != edges[j].to || edges[sta.back() -
1 + ((sta.back() & 1) << 1)].to != p)
                {
                    bcc[edges[sta.back() - 1 + ((sta.back() & 1) << 1)].to] =
bcc[edges[sta.back()].to] = edges[j].to;
                    sta.pop_back();
                }
                bcc[p] = bcc[edges[sta.back()].to] = edges[j].to;
                sta.pop_back();
            }
        }
}
```