图 论

清华大学计算机系　胡泽聪

目录

- 概念介绍
- 图的存储结构
- 最短路
- 最小生成树
- 有向无环图

概念介绍

基本概念

- 节点（点）　vertex
- 边　edge
- 点集　V
- 边集　E
- 图　$G = (V, E)$

图的分类

- 有向图　directed graph
- 无向图　undirected graph
- 混合图　mixed graph

- 树　tree
- 有向无环图　directed acyclic graph (DAG)
- 环套树/外向树
- 二分图　bipartite graph
- 仙人掌图　cacti

其他的东西

- 重边　multiple edges
- 自环　loop
- 路径　route
- 环　ring
- 生成树　spanning tree
- 度数　degree
  - 入度　in-degree
  - 出度　out-degree
- 拓扑排序　topological sort
- 连通性　connectivity

复杂一点的东西

- 独立集 independent set
- 团 clique
- 点覆盖 dominating set
- 边覆盖

- 这些现在不讲……

一些记号与约定

- $u \to v$代表u到v的有向边
- $u - v$代表u到v的无向边（如果会引起歧义，则用$i \leftrightarrow j$）
- 默认情况下，节点编号为$1 \sim n$。

图的存储结构

最基础的想法

- 直接以读入形式存储：边表
- 邻接矩阵

邻接表

- 很多时候题目里边数和点数同阶（比如都为100000）。
- 有时候也会遇到特殊的图，如树。

- 通常我们需要快速知道一个点引出的所有边。
- 直观的想法就是，把边按照一个端点排序，然后记录每个点对应在列表中的起始位置。
- 对于双向边，存正反两条。
- 这就是邻接表。

前向星

- 如果我们需要动态加边呢?

```
struct edge {
    int next, node, w;
} e[M * 2 + 1];
int head[N + 1], tot = 0;

inline void addedge(int a, int b, int w) {
    e[++tot].next = head[a];
    head[a] = tot, e[tot].node = b, e[tot].w = w;
}
```

- 本质上就是链表。
- 注意初始化。

前向星

- 如果要遍历x发出的所有边：

```
for (int i = head[u]; i; i = e[i].next) {
    // node = e[i].node, w = e[i].w
}
```

最短路

何谓最短路

- 给定图G，每条边有边权。
- 求从一点到另一点的边权和最小的路径。
- 要求图中不能有负回路（否则为NP问题）。

宽度优先搜索

- 如果所有边权均为1，宽度优先搜索可以求出最短路。
- 边权不为1？拆点。

Dijkstra 算 法

- 用于求从图中一点出发到其它所有点的最短路。即SSSP（single source shortest path，单源最短路）。
- 时间复杂度$O(n^2)$，空间复杂度$O(n)$（不含图结构）。
- 要求所有权值非负。称这样的图为正权图。

- 在下面的分析中，视1号点为源点，记d_u为1号点到点u的最短路距离。

Dijkstra算法

- 我们首先来看一些正权图的性质：

1. 如果1到u的最短路上包含v，那么这条路径上1到v的部分为1到v的最短路。
2. 任意一条最短路的一部分的长度一定不大于整条路径的长度。

Dijkstra算法

- 由这两个性质，我们有：

1. 假设1到u的最短路路径依次经过$p_1, p_2, \dots, p_m$（其中$p_1 = 1$，$p_m = u$），那么
$$0 = d_1 = d_{p_1} \le d_{p_2} \le \cdots \le d_{p_{m-1}} \le d_{p_m} = d_u$$
2. 假设已知1号点到其它所有点的最短路，且有$d_1 \le d_2 \le \cdots \le d_n$，那么
$$d_i = \min\{d_j + w(j,i)\}$$

- 由上面的分析可以得到Dijkstra算法。

Dijkstra算法

- 算法流程大致如下：

1. 记S为起点，置$d_S = 0$；对于其它点x，置$d_x = \infty$；
2. 找到当前尚未处理过的点中距离最小的，记为u；
3. 枚举所有与u相邻的节点v，用$d_u + w(u, v)$更新d_v；
4. 标记u为已处理；
5. 如果还有尚未处理过的点，返回2；否则算法结束。

Dijkstra算法 – 代码

```
for (int i = 1; i <= n; ++i)
    d[i] = INT_MAX, visit[i] = false;
d[S] = 0;

for (int i = 1; i <= n; ++i) {
    int minD = INT_MAX, minP = -1;
    for (int j = 1; j <= n; ++j)
        if (!visit[j] && d[j] < minD)
            minD = d[j], minP = j;

    visit[minP] = true;
    for (int j = 1; j <= n; ++j)
        d[j] = min(d[j], minD + w[minP][j]);
}
```

- 定义d[]为源点到每个点的最短路距离。
- 初始化。
- 找到尚未处理的最短路距离最小的点。
- 标记为已处理。
- 用这个点更新其他点的d[]。

Dijkstra算法 – 优化

- 首先得改变存储图的方式。

- 瓶颈在于哪里?
- 外层循环无法优化。
- 内层循环:
  - 找最小值
  - 更新答案

- 使用堆优化。

Dijkstra算法 – 堆

- 堆是一个支持插入元素、查询或删除最小值的数据结构。
  - 插入、删除复杂度为$O(\log n)$；
  - 查询复杂度为$O(1)$；
  - 还可以支持把一个元素改小，复杂度为$O(\log n)$。

- 不需要明白具体实现，当成一个黑箱就好。
- `std::priority_queue<int, vector<int>, greater<int> >`

Dijkstra算法 – 堆优化

- 为了优化查询最小值部分，我们需要存储所有尚未处理过的点。
- 在更新时候，如果距离变小，则在堆中更新。

- 而在更新时，也只在前向星中遍历从当前点发出的边。

- 复杂度$O(n\log n)$。

Dijkstra算法 – 堆优化

- 另一种写法是直接往堆里加，而不是改值。
- 适用于STL中的优先队列。

SPFA算法

- Shortest Path Faster Algorithm
- 队列优化的Bellman-Ford算法
- 同样用于求单源最短路。
- 可以用于带负权的图。

SPFA算法

- 算法流程大致如下：

1. 记S为起点，置$d_S = 0$；对于其它点x，置$d_x = \infty$；
2. 维护一个队列，初始为空；加入S；
3. 取出队首节点，记为u；
4. 枚举所有与u相邻的节点v，用$d_u + w(u, v)$更新d_v；如果d_v被更新且v不在队列中，则将其加入队列；
5. 如果队列不为空，返回3；否则算法结束。

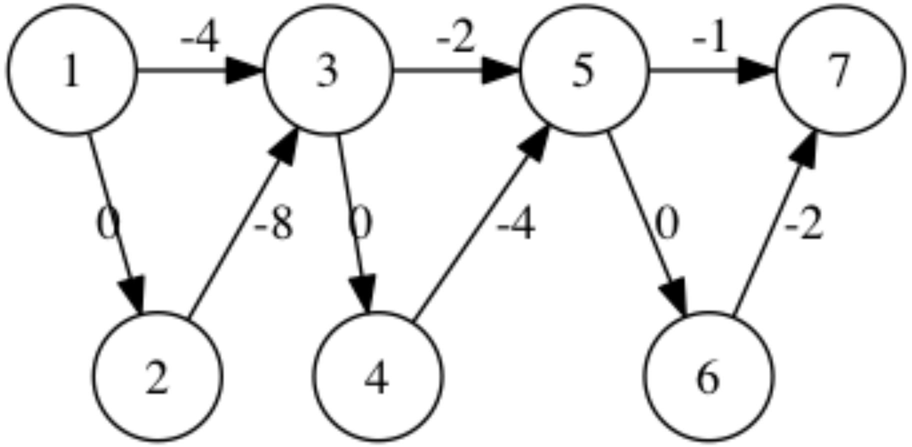
SPFA算法 – 代码

```
queue<int> q;
for (int i = 1; i <= n; ++i)
    d[i] = INT_MAX, inQueue[i] = false;
d[S] = 0, inQueue[S] = true;

while (!q.empty()) {
    int u = q.front();
    inQueue[u] = false, q.pop();

    for (int i = head[u]; i; i = e[i].next) {
        int v = e[i].node;
        if (d[v] > d[u] + e[i].w) {
            d[v] = d[u] + e[i].w;
            if (!inQueue[v])
                inQueue[v] = true, q.push(v);
        }
    }
}
```

- 初始化。
- 判断队列是否为空。
- 取出队首节点。
- 用这个点更新其他点的$d[]$，并判断是否要加入队列。

SPFA算法 – 复杂度

- 实际上算法在最坏情况下的复杂度是$O(nm)$。
- 但一般来说，除非是故意要卡SPFA，否则在效率上是和Dijkstra差不多的。



SPFA算法 – 优化

- 有三类常见优化：

1. SLF（smallest label first）：如果加入的元素距离小于队首元素，则置于队首；
2. LLL（largest label last）：记录队列中元素距离平均值，从队首取元素时，如果距离大于平均值则移至队尾，重复直到找到小于等于平均值的元素为止；
3. 随机化：从队首取元素时，有p的概率直接将其移至队尾；p应当是于图规模有关的一个小概率。

- 其实这些优化不能优化复杂度，甚至可能使算法变慢，但能在一定程度上化解针对性数据。

Floyd算法

- 用于求图中任意两点之间的最短路。即APSP (all pairs shortest path)。
- 时间复杂度$O(n^3)$，空间复杂度$O(n^2)$。
- 原理是DP。

Floyd算法

- 令$f[i][j][k]$为从i到j，中途只经过编号$1\sim k$的节点的最短路。
- 考虑是否经过k：
  - 如果经过，那么$f[i][j][k]=f[i][k][k-1]+f[k][j][k-1]$；
  - 否则，$f[i][j][k]=f[i][j][k-1]$。
- 直接这样做的空间复杂度是$O(n^3)$，我们可以进一步优化。
- 首先，$f[..][..][k]$只从$f[..][..][k-1]$转移，可以用滚动数组优化。
- 进一步地，如果经过k，那么只会从$f[..][k][k-1]$和$f[k][..][k-1]$转移，而$f[k][..][k]$和$f[..][k][k]$必然不会这样转移，故可以直接在原数组上迭代。

Floyd算法 – 代码

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

- $f[i][j]$初始值为无穷大。如果i和j之间有边那么$f[i][j]$为最小的边权。

Floyd算法 – 拓展应用

- 求图中的最小环。先考虑无向图。
- Floyd算法保证了最外层循环到k的时候所有点对之间的最短路只经过$1\sim k-1$号节点。
- 环至少有3个节点，设编号最大的为x，与之直接相连的两个节点为u和v。
- 环的长度应为$f[u][v][x-1]+w[v][x]+w[x][u]$。其中$w$为边权，如不存在边则为无穷大。
- 我们只要在进行第x次迭代之前枚举所有编号小于k的点对更新答案即可。

Floyd算法 – 代码2

```
for (int k = 1; k <= n; ++k) {
    for (int i = 1; i < k; ++i)
        for (int j = 1; j < i; ++j)
            ring = min(ring, f[i][j] + w[j][k] + w[k][i]);
    for (int i = 1; i <= n; ++i)
        for (int j = 1; j <= n; ++j)
            f[i][j] = min(f[i][j], f[i][k] + f[k][j]);
}
```

- j只循环到$i-1$，因为无向图中$i \to j$和$j \to i$是等价的。
- 有向图则类似，大家自己思考一下。另外注意两个点的环的情况。

最小生成树

何谓最小生成树

- 最小生成树　minimum spanning tree (MST)
- 给定无向连通图G，每条边有边权。
- 求G的生成树，使得树上边的边权和最小。

Prim 算法

- 类似于Dijkstra算法。
- 贪心思想。
- 算法流程大致如下：

1. 先任意找一个点加入初始的点集；
2. 求出所有尚未加入点集的点到点集内的点的最小边权，如果不存在则为无穷大；
3. 将得到边权最小的点加入点集，边加入最小生成树；
4. 如果点集$\neq V$，重复第2步。

Prim算法 – 伪代码

```
int ans = 0, S = 1;
visit[S] = true;
for (int i = 1; i <= n; ++i)
    d[i] = w[S][i];

for (int i = 1; i <= n; ++i) {
    int minD = INT_MAX, minP = -1;
    for (int j = 1; j <= n; ++j)
        if (!visit[j] && d[j] < minD)
            minD = d[j], minP = j;

    ans += minD;
    visit[minP] = true;
    for (int j = 1; j <= n; ++j)
        d[j] = min(d[j], w[minP][j]);
}
```

- 定义$d[]$为节点到已选择点集的最小边权。
- 初始化。
- 找到尚未加入的边权最小的点。
- 加入答案。
- 用这个点更新其他点的$d[]$。

Prim算法 – 正确性

- 令Prim算法构造出来的树为Y，假设存在一棵最小生成树Y_1且$Y_1 \neq Y$。
- 令e为在算法过程中加入的第一条不在Y_1中的边，设当前的点集为V_{now}，设e的两端分别为u和v，其中u在V_{now}中。
- 由于Y_1是一棵树，那么存在一条$u \to v$且不经过e的路径。找到第一条连接一个V_{now}中的点和一个不在V_{now}中的点的边。令这条边为f。
- 由于f没有在e被加入的这次迭代中被加入，所以一定有$w(f) \geq w(e)$。
- 那么在Y_1中加入e，此时构成了一个环，从环上删去f，得到了一棵树Y_2，而这棵树的边权和不劣于Y_1，因此Y_2也是最小生成树。
- 不断重复以上操作，可以进而将属于Y_2且不属于Y的所有边替换。故Prim算法构造出来的树Y是原图的一棵最小生成树。

Prim算法 – MST的环切性质

- 从上述证明中可以得到一个结论，即最小生成树的“环切性质”。
- 对于图中任意一条不属于MST的边$e: (u, v)$，对于树上在$u \rightarrow v$路径上的任意一条边f，必定有$w(e) \geq w(f)$。
- 证明也很简单：若非如此，可以e替换f，易知替换后仍为一棵树，且边权和更小。

Prim算法 – 复杂度

- 迭代n次。
  - 找d最大的尚未加入节点，$O(n)$；
  - 修改剩余节点的d值，$O(n)$。
- 总复杂度$O(n^2)$。

- 形式上是不是和Dijkstra很像?
- 如果用前向星存储图，并用堆优化，可以达到 $O((n+m)\log n)$ 的复杂度。

Kruskal算法

- 基于和Prim同样的贪心思想。
- 算法流程大致如下：

1. 将所有边按权值排序；
2. 按权值从小到大枚举每条边，如果这条边的两端不在同一个连通块内，则加入这条边，否则舍弃。

Kruskal 算 法

- 那么重点就在于如何维护“在同一连通块内”的信息。
- 有一种十分牛逼的数据结构叫做并查集。

Kruskal 算 法 – 并 查 集

- 并查集用于维护有根树森林的连通性。
- 定义f[]表示每个节点的父亲节点。特别地，如果一个节点是根节点，那么其父亲节点是自身。
- 如果两个节点拥有相同的父亲节点，那么这两个节点在同一棵树，即同一连通块内。
- 一个朴素的做法是：

```
int find(int x) {
    while (f[x] != x)
        x = f[x];
    return x;
}
```

Kruskal算法 – 并查集

- 但是这样最坏的复杂度也可以达到单次查询$O(n)$。我们需要优化。
- 一个优化叫做“路径压缩”。
- 即，每次往上查自己父亲的同时，将已经查到的父亲的父亲赋为自己的父亲。
- 这里$f[]$的含义就变成了节点的某个祖先。
- 递归版的实现也很简单：

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

Kruskal算法 – 并查集

- 有时候，比如这里，我们需要合并两个集合。
- 方法也很简单：我们找到两个集合的根，然后令其中一个的父亲为另外一个。
- 也即：

```
void union(int x, int y) {
    f[find(x)] = find(y);
}
```

Kruskal算法 – 并查集

- 并查集的复杂度我们可以认为是接近线性的。
- 具体的复杂度分析比较复杂，而且一般都涉及到了合并操作的另一个优化。
- 有兴趣的同学可以查询了解。

Kruskal算法 – 伪代码

- 回到Kruskal算法。使用了并查集的Kruskal算法可以在$O(m \log m)$的时间复杂度内求解最小生成树问题。
- 伪代码如下：

```
int ans = 0;
sort(&e[1], &e[n] + 1);
for (int i = 1; i <= m; ++i)
    if (find(e[i].a) != find(e[i].b)) {
        ans += e[i].w;
        union(e[i].a, e[i].b);
    }
```

有向无环图

何谓有向无环图

- 有向图，没有环。

- 一些有向无环图：
  - 正常的依赖关系
  - 公平组合游戏的局面转移

拓扑序列

- 拓扑序列是图中节点的一个排列，满足：
  如果序列中a出现在b前面，那么图中不存在b到a的路径。
- 有向无环图可拓扑，即存在拓扑序列。
  - 按照拓扑序列解决依赖关系。

拓扑排序

- 算法流程大致如下：

1. 统计每个点的入度；
2. 维护一个队列，将所有入度为0的点加入；
3. 取出队首节点u，枚举所有与其相邻的节点v；
4. 令v的入度减1，如果变为了0，则加入队列；
5. 如果队列不为空，返回2；
6. 出队顺序即为拓扑序列。

Tarjan强连通分量算法

连通性

- 首先我们要具体了解连通性的概念。
- 对于有向图而言：
  - 强连通
  - 弱联通
- 对于无向图而言：
  - 联通
  - 双联通
  - 拓展：k-联通

强连通分量

- 对于一张有向图，如果整张图强连通，那么它是强连通图。
- 如果它的某个子图强连通，那么这个子图是一个强连通分量。
- Tarjan算法可以求出有向图的所有极大强连通分量。

Tarjan算法

- Tarjan算法的实质是将图转成树，然后在树上求解问题。
- 首先任意构造一棵原图的生成树，并任选一个点为根。
- 深度优先遍历这棵树。对于每个节点，我们维护两个值：
  - $dfn[x]$，代表访问这个节点的时间戳；
  - $low[x]$，代表这个节点的子树中所有节点，通过“向上的”非树边能访问到的最早的时间戳。
- 同时维护一个队列，代表当前遍历过的且尚未确定属于哪个强连通分量的节点，按照遍历顺序排列。

Tarjan算法 – 代码

- 打上时间戳，加入队列；
- 枚举每条从x发出的边；
- 第一次访问：是树边；

- 已在队列中：是向上的非树边；

- 找到了一个强连通分量（x是分量中第一次访问的节点）；
- 将队列中的元素标记为属于此强连通分量。

```
void tarjan(int x) {
    ++cnt;
    low[x] = dfn[x] = x, visit[x] = true;
    q.push(x), inqueue[x] = true;
    for (edge e : edges_from[x]) {
        int y = e.node;
        if (!visit[y]) {
            tarjan(y);
            low[x] = min(low[x], low[y]);
        } else if (inqueue[y]) {
            low[x] = min(low[x], dfn[y]);
        }
    }
    if (low[x] == dfn[x]) {
        ++scc, belong[x] = scc;
        inqueue[x] = false;
        while (q.top() != x) {
            belong[q.top()] = scc;
            inqueue[q.top()] = false;
            q.pop();
        }
    }
}
```

Tarjan算法

- 很明显复杂度为$O(n+m)$，即遍历图的复杂度。
- Tarjan算法也可以用于求无向图中的边双连通分量，只需要稍作改动：不允许从一个点走回其父亲。
- 最重要的问题是：求出强连通分量之后可以干什么?

例题 – APIO2009 ATM

- 给定一张n个点m条边的有向图，每个点有点权，有一些点被标记为终止节点。从1号点出发，沿着图中的边走，最后走到一个终止节点，途中经过的点的点权和为你的收益。如果经过了一个点多次，收益也只能计算一次。求最大的收益。保证能从1号点走到某个终止节点。

- $n, m \leq 500{,}000$。

Tarjan算法 – 应用

- 如果我们走到了某个强连通分量，那么我们就可以走遍这个强连通分量中的每个节点，然后从任意一条边走出这个强连通分量。
- 那么我们把每个强连通分量缩成一个点，并把这些点的点权和作为新的点的点权。
- 这样我们得到了一个DAG，似乎可以DP了。
- 令$f[i]$表示走到i号节点的最大收益，有：
- $f[i] = w[i] + \max\{f[j], (j, i) \in E\}$
- 直接按照拓扑序DP。
- 当然由于是DAG，直接做最长路也是可以的。

强连通分量 – 性质

- 如果我们把有向图中所有极大强连通分量都视作一个节点，然后重新构图，新的图会是一个DAG。
- 对于无向图，新的图会是一棵树。
- 证明很简单。

- 更进一步地，还可以利用Tarjan算法求出的 $low[]$ 和 $dfn[]$ 来求割边/割点。
  - 若u存在儿子v满足$low[v] \geq dfn[u]$，则u是割点（根需要有两个儿子满足）；
  - 若边(u, v)满足$low[v] > dfn[u]$，则(u, v)为割边（注意判断重边）；

例题

NOIP2013 D1P3
货车运输

- n个点m条边的无向图，每条边有边权。
- q个询问，求两点间的一条路径，满足路径上最大的边权最小。

- $0 < n < 10000$，$0 < m < 50000$，$0 < q < 30000$。

NOIP2014 D2P2
寻找道路

- n个点m条边的有向图，边的权值均为1。
- 求一条起点到终点的最短路径，满足路径上所有点的出边指向的点都可以到达终点。

- $0 < n \leq 10000$，$0 < m \leq 200000$。

USACO 2014 February Gold Roadblock

- n个点m条边的无向连通图，每条边有边权。
- 可以选择一条边将其权值翻倍。求翻倍后从1到n的最短路长度最长可以是多少。

- $n \leq 250$，$m \leq 25000$。

Beijing WC 2012
冻结

- n个点m条边的无向连通图，每条边有边权。
- 可以将最多k条边的边权变为原来的一半。
- 求从1到n的最短路。

- $n, k \leq 50$，$m \leq 1000$。

NOIP2009 P3
最优贸易

- n个点m条边的混合图。
- 一种物品在每个点有一个售价。
- 你需要从起点走到终点，沿途可以进行一次交易（购入一次物品并卖出一次），赚取差价作为收益。可以重复经过一个点。
- 求最大收益。也可以不进行交易。

- $1 \le n \le 100000$，$1 \le m \le 500000$。

BZOJ2208 – JSOI2010 连通数

- n个点的有向无环图。
- 求每个点可以到多少个点。

- $n \leq 2000$。此处为原题的削弱。

模拟题1 P2
删边

- n个点m条边的无向图，依次删去其中k条边。
- 求每次删去一条边后，图中连通块的个数。

- $1 \leq n \leq 100000$，$0 \leq k \leq m \leq 100000$。

模拟题1 P4
旅行路线

- n个点m条边的无向图，每条边有长度和收费。
- 求从起点到终点的一条路径，要求首先长度最小，在此前提下费用最小。

- $1 \le n, m \le 100000$。

Nescafe II P2
黑暗城堡

- n个点m条边的无向连通图，每条边有边权。
- 求有多少棵生成树满足，树上从1号点到每个点的路径长度等于原图中1号点到该点的最短路长度。
- $1 \le n \le 1000$，$n-1 \le m \le \frac{n(n-1)}{2}$。

模拟题2
公共路径

- n个点m条边的无向图，每条边有边权。
- 有两对起点终点，分别求这两对点间的最短路，并使得其公共部分最长。

- $n \leq 2000$，$m \leq \frac{n(n-1)}{2}$。

POJ3177
Redundant Paths

- 给定一张n个点m条边的无向连通图，要求添加一些边，使得任意两节点间存在至少两条不相交的路径。
- 求最少需要添加多少条边。

- $n, m \leq 100000$。

SDOI2010
所驼门王的宝藏

- $R \times C$的网格，其中N个点有传送门。传送门有三类：

1. 可以到达任意同行的传送门；
2. 可以到达任意同列的传送门；
3. 可以到达八连通相邻的传送门。

- 从任意传送门出发，问最多可以经过多少不同的传送门。

- $N \leq 100000$，$R, C \leq 10^6$。

HAOI2016
旅行

- n个点m条边的无向图，每条边有边权。
- 求一条从S到T的路径，使得路径上最大边权与最小边权的比值最小。

- $n \le 500$，$m \le 5000$。

终