

图论

常见概念

oriented graph: 有向图

bidirectional edges: 双向边

平面图: 若能将无向图 $G = (V, E)$ 画在平面上使得任意两条无重合顶点的边不相交, 则称 G 是平面图。

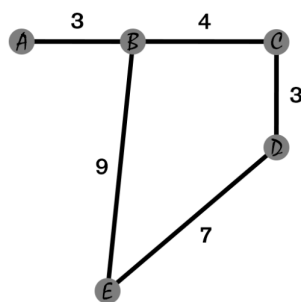
无向正权图上某一点的偏心距: 记为 $ecc(u) = \max \{dist(u, v)\}$, 即以这个点为源, 到其他点的**所有最短路的最大值**。如下图 A 点, $ecc(A)$ 即为 12。

图的直径: 定义为 $d = \max \{ecc(u)\}$, 即**最大的偏心距**, 亦可以简化为图中最远的一对点的距离。

图的中心: 定义为 $arg = \min \{ecc(u)\}$, 即**偏心距最小的点**。如下图, 图的中心即为 B 点。

图的绝对中心: 可以定义在边上的图的中心。

图的半径: 图的半径不同于圆的半径, 其不等于直径的一半 (但对于绝对中心定义上的直径而言是一半)。定义为 $r = \min \{ecc(u)\}$, 即**中心的偏心距**。计算方式: 使用全源最短路, 计算出所有点的偏心距, 再加以计算。



平面图性质

一、定义

$G = (V, E)$ 是一个无向图。

1. **图G可嵌入平面**: 如果可以把图G的所有结点和边都画在平面上, 同时除断点外连线之间没有交点, 就称图G可嵌入平面。画出的无边相交的G'称G的平面嵌入。
2. **可平面化**: 如果图G可以嵌入平面, 就称图G可平面化。
3. **面**: G中边所包含的区域称作一个面。有界区域称为内部面, 无界区域称为外部面, 常记作 R_0 , 包围面的长度最短的闭链称为该面的边界, 面 R 的边界的长度称为该面的度数, 记作 $\deg(R)$ 。
4. **面的度数计算**: 含有割边和桥的度数为2, 其余为1。

二、性质

1. 性质 1. $K_1, K_2, K_3, K_4, K_5 - e$ 均为极大可平面图。
2. 性质 2. 极大平面图必是连通图。
3. 性质 3. 当图阶数 $n \geq 3$ 时, 有割点或者桥的平面图不是极大平面图。

三、定理

1. **定理 1:** 图 G 可嵌入球面当且仅当图 G 可嵌入平面。
2. **定理 2:** G 中各面的度数之和等于图 G 边数的两倍。
证明: 设 e 为图 G 的两个面的公共边,再计算两个面的度数时候边数各提供1,当 e 不是公共边时候,也就是 e 为桥或者割边时候提供度数为2。因此,面的度数之和为边的两倍。
3. **定理 3:** 设 R 是图 G 的某个平面嵌入的一个内部面,则存在图 G 的一个平面嵌入使 R 为外部面。
4. **定理 4:** 设图 G 是简单的可平面图,如果 G 中任意两个不相邻的结点加边后所得到的为非可平面图。则称 G 是极大可平面图,极大可平面图的任何平面嵌入都称为极大平面图。极大平面图必是连通图。
5. **定理 5:** 图 G 为 n 阶简单的连通的平面图, G 为极大平面图当且仅当 G 的每一个面的度数为3。
定理说明: 结点数大于等于3的极大平面图的任何面都是由三角形组成。
6. **定理 6: 欧拉公式:** 设图 G 是有 n 个结点、 m 条边和 r 个面的连通平面图,则它们满足:

$$n - m + r = 2$$

四、结论

1. **结论 1:** $K_1, K_2, K_3, K_4, K_5 - e$ (K_5 任意删去一条边)均为极大可平面图,它们的任何平面嵌入都是极大平面图;当阶数等于3时候,有割边或桥的平面图不可能是极大平面图。
2. **结论 2:** 无向完全图 K_5 和无向完全二部图 $K_{3,3}$ 都是极小非可平面图(去掉一条边就成为可平面图)。
3. **结论 3:** 一个图是可平面图,那么它的子图也是可平面图;一个图的子图是非可平面图,那么图本身也是非可平面图。
4. **结论 4:** 同一个图的平面嵌入中,外部面和内部面的度数可以不同。

五、推论

1. **推论 1:** 设图 G 是有 n 个结点、 m 条边的连通平面简单图,其中 $n \geq 3$,则有:

$$m \leq 3n - 6$$
证明: 由图 G 的面度数之和为边数的二倍,即 $2m$ 。又因为 G 是平面简单图每一个面的度数至少为3,则

$$2m \geq 3r$$
由欧拉公式有: $m \leq 3n - 6$
2. **推论 2:** 设图 G 是有 n 个结点、 m 条边的连通平面简单图,其中 $n \geq 3$ 且没有长度为3的圈,则有:

$$m \leq 2n - 4$$
证明: G 没有长度为3的圈也就没有度为3的面, G 的每一个面的度数至少为4。所以 $2m \geq 4r$,由欧拉公式有: $m \leq 2n - 4$
提示: 对于推论1和推论2我们可以用定理进行判定它不是平面图。
例 1: 证明 K_5 和 $K_{3,3}$ 是非平面图。
证明:
 - 在 K_5 中, m 应该小于等于 $3n - 6$,即 $m \leq 9$ 。而完全图 K_5 具有10条边。所以是非平面图。
 - 在 $K_{3,3}$ 中,没有长度大于3的圈,根据推论2可知, $m \leq 2n - 4$,也就是 $m \leq 8$,而 $K_{3,3}$ 含有9条边,所以是非平面图。
3. **推论 3:** 设 G 是连通的平面图,且每个面的度数至少为 l ($l \geq 3$),则

$$m \leq \frac{l}{l-2}(n-2).$$
证明: 同理,有 $2m \geq r \times l$,根据欧拉公式化简得:

$$2m \geq l(m - n + 2)$$
4. **推论 4:** 设 G 是平面图,有 ω 个连通分支, n 个结点, m 条边, r 个面,则公式

$$n - m + r = \omega + 1$$
成立。

5. **推论 5:** 设 G 是有 n 个结点、 m 条边和 r 个面、 ω 个连通分支的平面图, 且 G 的各个面的度数至少为 l , ($l \geq 4$), 则

$$m \leq \frac{(n-\omega-1)l}{l-2}.$$

证明: 证明过程与推论3类似, 用到推论4的结论。

6. **推论 6:** 设 G 是任意平面简单图, 则 $\delta(G) \leq 5$.

证明: 设 G 有 n 个顶点 m 条边. 若 $m \leq 6$, 结论显然成立; 若 $m > 6$, 假设 G 的每个顶点的度数 > 6 , 则由推论 1, 有

$$6n \leq \sum d(v) = 2m \leq 2(3n - 6) = 6n - 12$$

与定理矛盾, 故 $\delta(G) \leq 5$.

六、判别定理

1. **极大平面图的判别定理:** $n(n \geq 3)$ 阶连通的简单平面图 G . 则以下四个条件等价:

1. G 是极大平面图;
2. G 中每个面的度数都是3;
3. G 中有 m 条边 r 个面, 则
 $3r = 2m$;
4. 设 G 带有 n 个顶点, m 条边, r 个面则
 $m = 3n - 6$;

单源最短路径 (SSSP问题)

(正权稀疏图) 动态数组存图+Dijkstra算法

使用优先队列优化, 以 $\mathcal{O}(M \log N)$ 的复杂度计算。

```

1  vector<int> dis(n + 1, 1E18);
2  auto dijkstra = [&](int s = 1) -> void {
3      using PII = pair<int, int>;
4      priority_queue<PII, vector<PII>, greater<PII>> q;
5      q.emplace(0, s);
6      dis[s] = 0;
7      vector<int> vis(n + 1);
8      while (!q.empty()) {
9          int x = q.top().second;
10         q.pop();
11         if (vis[x]) continue;
12         vis[x] = 1;
13         for (auto [y, w] : ver[x]) {
14             if (dis[y] > dis[x] + w) {
15                 dis[y] = dis[x] + w;
16                 q.emplace(dis[y], y);
17             }
18         }
19     }
20 };

```

(负权图) Bellman ford 算法

使用结构体存边（该算法无需存图），以 $\mathcal{O}(NM)$ 的复杂度计算，注意，当所求点的路径上存在负环时，所求点的答案无法得到，但是会比 INF 小（因为负环之后到所求点之间的边权会将 $d[\text{end}]$ 的值更新），该性质可以用于判断路径上是否存在负环：在 $N - 1$ 轮后仍无法得到答案（一般与 $\text{INF}/2$ 进行比较）的点，到达其的路径上存在负环。

下方代码例题：求解从 1 到 n 号节点的、最多经过 k 条边的最短距离。

```

1  const int N = 550, M = 1e5 + 7;
2  int n, m, k;
3  struct node { int x, y, w; } ver[M];
4  int d[N], backup[N];
5
6  void bf() {
7      memset(d, 0x3f, sizeof d); d[1] = 0;
8      for (int i = 1; i <= k; ++ i) {
9          memcpy(backup, d, sizeof d);
10         for (int j = 1; j <= m; ++ j) {
11             int x = ver[j].x, y = ver[j].y, w = ver[j].w;
12             d[y] = min(d[y], backup[x] + w);
13         }
14     }
15 }
16 int main() {
17     cin >> n >> m >> k;
18     for (int i = 1; i <= m; ++ i) {
19         int x, y, w; cin >> x >> y >> w;
20         ver[i] = {x, y, w};
21     }
22     bf();
23     for (int i = 1; i <= n; ++ i) {
24         if (d[i] > INF / 2) cout << "N" << endl;
25         else cout << d[n] << endl;
26     }
27 }
```

(负权图) SPFA 算法

以 $\mathcal{O}(KM)$ 的复杂度计算，其中 K 虽然为常数，但是可以通过特殊的构造退化接近 N ，需要注意被卡。

```

1  const int N = 1e5 + 7, M = 1e6 + 7;
2  int n, m;
3  int ver[M], ne[M], h[N], edge[M], tot;
4  int d[N], v[N];
5
6  void add(int x, int y, int w) {
7      ver[++ tot] = y, ne[tot] = h[x], h[x] = tot;
8      edge[tot] = w;
9  }
10 void spfa() {
11     ms(d, 0x3f); d[1] = 0;
12     queue<int> q; q.push(1);
```

```

13     v[1] = 1;
14     while(!q.empty()) {
15         int x = q.front(); q.pop(); v[x] = 0;
16         for (int i = h[x]; i; i = ne[i]) {
17             int y = ver[i];
18             if(d[y] > d[x] + edge[i]) {
19                 d[y] = d[x] + edge[i];
20                 if(v[y] == 0) q.push(y), v[y] = 1;
21             }
22         }
23     }
24 }
25 int main() {
26     cin >> n >> m;
27     for (int i = 1; i <= m; ++ i) {
28         int x, y, w; cin >> x >> y >> w;
29         add(x, y, w);
30     }
31     spfa();
32     for (int i = 1; i <= n; ++ i) {
33         if (d[i] == INF) cout << "N" << endl;
34         else cout << d[n] << endl;
35     }
36 }

```

(正权稠密图) 邻接矩阵存图+Dijkstra算法

很少使用，以 $\mathcal{O}(N^2)$ 的复杂度计算。

```

1  const int N = 3010;
2  int n, m, a[N][N];
3  int d[N], v[N];
4
5  void dji() {
6      ms(d, 0x3f); d[1] = 0;
7      for (int i = 1; i <= n; ++ i) {
8          int x = 0;
9          for (int j = 1; j <= n; ++ j) {
10             if(v[j]) continue;
11             if(x == 0 || d[x] > d[j]) x = j;
12         }
13         v[x] = 1;
14         for (int j = 1; j <= n; ++ j) d[j] = min(d[j], d[x] + a[x][j]);
15     }
16 }
17 int main() {
18     cin >> n >> m;
19     ms(a, 0x3f);
20     for (int i = 1; i <= m; ++ i) {
21         int x, y, w; cin >> x >> y >> w;
22         a[x][y] = min(a[x][y], w); //注意需要考虑重边问题
23         a[y][x] = min(a[y][x], w); //无向图建双向边
24     }

```

```

25     dji();
26     for (int i = 1; i <= n; ++ i) {
27         if (d[i] == INF) cout << "N" << endl;
28         else cout << d[n] << endl;
29     }
30 }

```

多源汇最短路（APSP问题）

使用邻接矩阵存图，可以处理负权边，以 $\mathcal{O}(N^3)$ 的复杂度计算。注意，这里建立的是单向边，计算双向边需要额外加边。

```

1  const int N = 210;
2  int n, m, d[N][N];
3
4  void floyd() {
5      for (int k = 1; k <= n; k++)
6          for (int i = 1; i <= n; i++)
7              for (int j = 1; j <= n; j++)
8                  d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
9  }
10 int main() {
11     cin >> n >> m;
12     for (int i = 1; i <= n; i++)
13         for (int j = 1; j <= n; j++)
14             if (i == j) d[i][j] = 0;
15             else d[i][j] = INF;
16     while (m--) {
17         int x, y, w; cin >> x >> y >> w;
18         d[x][y] = min(d[x][y], w);
19     }
20     floyd();
21     for (int i = 1; i <= n; ++ i) {
22         for (int j = 1; j <= n; ++ j) {
23             if (d[i][j] > INF / 2) cout << "N" << endl;
24             else cout << d[i][j] << endl;
25         }
26     }
27 }

```

平面图最短路（对偶图）

对于矩阵图，建立对偶图的过程如下（注释部分为建立原图），其中数据的给出顺序依次为：各 $n(n+1)$ 个数字分别代表从左向右、从上向下、从右向左、从下向上的边。

```

1  for (int i = 1; i <= n + 1; i++) {
2      for (int j = 1, w; j <= n; j++) {
3          cin >> w;
4          int pre = Hash(i - 1, j), now = Hash(i, j);
5          if (i == 1) {
6              add(s, now, w);

```

```

7         } else if (i == n + 1) {
8             add(pre, t, w);
9         } else {
10            add(pre, now, w);
11        }
12        // flow.add(Hash(i, j), Hash(i, j + 1), w);
13    }
14 }
15 for (int i = 1; i <= n; i++) {
16     for (int j = 1, w; j <= n + 1; j++) {
17         cin >> w;
18         int now = Hash(i, j), net = Hash(i, j - 1);
19         if (j == 1) {
20             add(now, t, w);
21         } else if (j == n + 1) {
22             add(s, net, w);
23         } else {
24             add(now, net, w);
25         }
26         // flow.add(Hash(i, j), Hash(i + 1, j), w);
27     }
28 }
29 for (int i = 1; i <= n + 1; i++) {
30     for (int j = 1, w; j <= n; j++) {
31         cin >> w;
32         int now = Hash(i, j), net = Hash(i - 1, j);
33         if (i == 1) {
34             add(now, s, w);
35         } else if (i == n + 1) {
36             add(t, net, w);
37         } else {
38             add(now, net, w);
39         }
40         // flow.add(Hash(i, j), Hash(i, j - 1), w);
41     }
42 }
43 for (int i = 1; i <= n; i++) {
44     for (int j = 1, w; j <= n + 1; j++) {
45         cin >> w;
46         int pre = Hash(i, j - 1), now = Hash(i, j);
47         if (j == 1) {
48             add(t, now, w);
49         } else if (j == n + 1) {
50             add(pre, s, w);
51         } else {
52             add(pre, now, w);
53         }
54         // flow.add(Hash(i, j), Hash(i - 1, j), w);
55     }
56 }

```

最小生成树（MST问题）

（稀疏图）Prim算法

使用邻接矩阵存图，以 $\mathcal{O}(N^2 + M)$ 的复杂度计算，思想与 `dijkstra` 基本一致。

```

1  const int N = 550, INF = 0x3f3f3f3f;
2  int n, m, g[N][N];
3  int d[N], v[N];
4  int prim() {
5      ms(d, 0x3f); //这里的d表示到“最小生成树集合”的距离
6      int ans = 0;
7      for (int i = 0; i < n; ++ i) { //遍历 n 轮
8          int t = -1;
9          for (int j = 1; j <= n; ++ j)
10             if (v[j] == 0 && (t == -1 || d[j] < d[t])) //如果这个点不在集合内且当前距离集合最
近
11                 t = j;
12             v[t] = 1; //将t加入“最小生成树集合”
13             if (i && d[t] == INF) return INF; //如果发现不连通，直接返回
14             if (i) ans += d[t];
15             for (int j = 1; j <= n; ++ j) d[j] = min(d[j], g[t][j]); //用t更新其他点到集合的距
离
16         }
17         return ans;
18     }
19 int main() {
20     ms(g, 0x3f); cin >> n >> m;
21     while (m -- ) {
22         int x, y, w; cin >> x >> y >> w;
23         g[x][y] = g[y][x] = min(g[x][y], w);
24     }
25     int t = prim();
26     if (t == INF) cout << "impossible" << endl;
27     else cout << t << endl;
28 } //22.03.19已测试

```

（稠密图）Kruskal算法

平均时间复杂度为 $\mathcal{O}(M \log M)$ ，简化了并查集。

```

1  struct DSU {
2      vector<int> fa;
3      DSU(int n) : fa(n + 1) {
4          iota(fa.begin(), fa.end(), 0);
5      }
6      int get(int x) {
7          while (x != fa[x]) {
8              x = fa[x] = fa[fa[x]];
9          }
10         return x;
11     }

```



```

12     bool merge(int x, int y) { // 设x是y的祖先
13         x = get(x), y = get(y);
14         if (x == y) return false;
15         fa[y] = x;
16         return true;
17     }
18     bool same(int x, int y) {
19         return get(x) == get(y);
20     }
21 };
22 struct Tree {
23     using TII = tuple<int, int, int>;
24     int n;
25     priority_queue<TII, vector<TII>, greater<TII>> ver;
26
27     Tree(int n) {
28         this->n = n;
29     }
30     void add(int x, int y, int w) {
31         ver.emplace(w, x, y); // 注意顺序
32     }
33     int kruskal() {
34         DSU dsu(n);
35         int ans = 0, cnt = 0;
36         while (ver.size()) {
37             auto [w, x, y] = ver.top();
38             ver.pop();
39             if (dsu.same(x, y)) continue;
40             dsu.merge(x, y);
41             ans += w;
42             cnt++;
43         }
44         assert(cnt < n - 1); // 输入有误，建树失败
45         return ans;
46     }
47 };

```

缩点（Tarjan 算法）

（有向图）强连通分量缩点

强连通分量缩点后的图称为 SCC。以 $\mathcal{O}(N + M)$ 的复杂度完成上述全部操作。

性质：缩点后的图拥有拓扑序 $color_{cnt}, color_{cnt-1}, \dots, 1$ ，可以不需再另跑一遍 topsort；缩点后的图是一张有向无环图（DAG、拓扑图）。

```

1 struct SCC {
2     int n, now, cnt;
3     vector<vector<int>> ver;
4     vector<int> dfn, low, col, s;
5
6     SCC(int n) : n(n), ver(n + 1), low(n + 1) {
7         dfn.resize(n + 1, -1);

```

```

8     col.resize(n + 1, -1);
9     now = cnt = 0;
10 }
11 void add(int x, int y) {
12     ver[x].push_back(y);
13 }
14 void tarjan(int x) {
15     dfn[x] = low[x] = now++;
16     S.push_back(x);
17     for (auto y : ver[x]) {
18         if (dfn[y] == -1) {
19             tarjan(y);
20             low[x] = min(low[x], low[y]);
21         } else if (col[y] == -1) {
22             low[x] = min(low[x], dfn[y]);
23         }
24     }
25     if (dfn[x] == low[x]) {
26         int pre;
27         cnt++;
28         do {
29             pre = S.back();
30             col[pre] = cnt;
31             S.pop_back();
32         } while (pre != x);
33     }
34 }
35 auto work() { // [cnt 新图的顶点数量]
36     for (int i = 1; i <= n; i++) { // 避免图不连通
37         if (dfn[i] == -1) {
38             tarjan(i);
39         }
40     }
41
42     vector<int> siz(cnt + 1); // siz 每个 scc 中点的数量
43     vector<vector<int>> adj(cnt + 1);
44     for (int i = 1; i <= n; i++) {
45         siz[col[i]]++;
46         for (auto j : ver[i]) {
47             int x = col[i], y = col[j];
48             if (x != y) {
49                 adj[x].push_back(y);
50             }
51         }
52     }
53     return {cnt, adj, col, siz};
54 }
55 };

```

(无向图) 割边缩点

割边缩点后的图称为边双连通图 (E-DCC)，该模板可以在 $\mathcal{O}(N + M)$ 复杂度内求解图中全部割边、划分边双（颜色相同的点位于同一个边双连通分量中）。

割边（桥）：将某边 e 删去后，原图分成两个以上不相连的子图，称 e 为图的割边。

边双连通：在一张连通的无向图中，对于两个点 u 和 v ，删去任何一条边（只能删去一条）它们依旧连通，则称 u 和 v 边双连通。一个图如果不存在割边，则它是一个边双连通图。

性质补充：对于一个边双，删去任意边后依旧联通；对于边双中的任意两点，一定存在两条不相交的路径连接这两个点（路径上可以有公共点，但是没有公共边）。

```

1  struct EDCC {
2      int n, m, now, cnt;
3      vector<vector<array<int, 2>>> ver;
4      vector<int> dfn, low, col, s;
5      set<array<int, 2>> bridge, direct; // 如果不需要，删除这一部分可以得到一些时间上的优化
6
7      EDCC(int n) : n(n), low(n + 1), ver(n + 1), dfn(n + 1), col(n + 1) {
8          m = now = cnt = 0;
9      }
10     void add(int x, int y) { // 和 scc 相比多了一条连边
11         ver[x].push_back({y, m});
12         ver[y].push_back({x, m++});
13     }
14     void tarjan(int x, int fa) {
15         dfn[x] = low[x] = ++now;
16         s.push_back(x);
17         for (auto &[y, id] : ver[x]) {
18             if (!dfn[y]) {
19                 direct.insert({x, y});
20                 tarjan(y, id);
21                 low[x] = min(low[x], low[y]);
22                 if (dfn[x] < low[y]) {
23                     bridge.insert({x, y});
24                 }
25             } else if (id != fa && dfn[y] < dfn[x]) {
26                 direct.insert({x, y});
27                 low[x] = min(low[x], dfn[y]);
28             }
29         }
30         if (dfn[x] == low[x]) {
31             int pre;
32             cnt++;
33             do {
34                 pre = s.back();
35                 col[pre] = cnt;
36                 s.pop_back();
37             } while (pre != x);
38         }
39     }
40     auto work() {
41         for (int i = 1; i <= n; i++) { // 避免图不连通

```

```

42         if (!dfn[i]) {
43             tarjan(i, 0);
44         }
45     }
46     /**
47      * @param cnt 新图的顶点数量, adj 新图, col 旧图节点对应的新图节点
48      * @param siz 旧图每一个边双中点的数量
49      * @param bridge 全部割边, direct 非割边定向
50      */
51     vector<int> siz(cnt + 1);
52     vector<vector<int>> adj(cnt + 1);
53     for (int i = 1; i <= n; i++) {
54         siz[col[i]]++;
55         for (auto &[j, id] : ver[i]) {
56             int x = col[i], y = col[j];
57             if (x != y) {
58                 adj[x].push_back(y);
59             }
60         }
61     }
62     return tuple{cnt, adj, col, siz};
63 }
64 };

```

(无向图) 割点缩点

割点缩点后的图称为点双连通图 (V-DCC), 该模板可以在 $\mathcal{O}(N + M)$ 复杂度内求解图中全部割点、划分点双 (颜色相同的点位于同一个点双连通分量中)。

割点 (割顶): 将与某点 i 连接的所有边删去后, 原图分成两个以上不相连的子图, 称 i 为图的割点。

点双连通: 在一张连通的无向图中, 对于两个点 u 和 v , 删去任何一个点 (只能删去一个, 且不能删 u 和 v 自己) 它们依旧连通, 则称 u 和 v 边双连通。如果一个图不存在割点, 那么它是一个点双连通图。

性质补充: 每一个割点至少属于两个点双。

```

1  struct V_DCC {
2      int n;
3      vector<vector<int>> ver, col;
4      vector<int> dfn, low, S;
5      int now, cnt;
6      vector<bool> point; // 记录是否为割点
7
8      V_DCC(int n) : n(n) {
9          ver.resize(n + 1);
10         dfn.resize(n + 1);
11         low.resize(n + 1);
12         col.resize(2 * n + 1);
13         point.resize(n + 1);
14         S.clear();
15         cnt = now = 0;
16     }
17     void add(int x, int y) {

```

```

18     if (x == y) return; // 手动去除重边
19     ver[x].push_back(y);
20     ver[y].push_back(x);
21 }
22 void tarjan(int x, int root) {
23     low[x] = dfn[x] = ++now;
24     S.push_back(x);
25     if (x == root && !ver[x].size()) { // 特判孤立点
26         ++cnt;
27         col[cnt].push_back(x);
28         return;
29     }
30
31     int flag = 0;
32     for (auto y : ver[x]) {
33         if (!dfn[y]) {
34             tarjan(y, root);
35             low[x] = min(low[x], low[y]);
36             if (dfn[x] <= low[y]) {
37                 flag++;
38                 if (x != root || flag > 1) {
39                     point[x] = true; // 标记为割点
40                 }
41                 int pre = 0;
42                 cnt++;
43                 do {
44                     pre = S.back();
45                     col[cnt].push_back(pre);
46                     S.pop_back();
47                 } while (pre != y);
48                 col[cnt].push_back(x);
49             }
50         } else {
51             low[x] = min(low[x], dfn[y]);
52         }
53     }
54 }
55 pair<int, vector<vector<int>>> rebuild() { // [新图的顶点数量, 新图]
56     work();
57     vector<vector<int>> adj(cnt + 1);
58     for (int i = 1; i <= cnt; i++) {
59         if (!col[i].size()) { // 注意, 孤立点也是 V-DCC
60             continue;
61         }
62         for (auto j : col[i]) {
63             if (point[j]) { // 如果 j 是割点
64                 adj[i].push_back(point[j]);
65                 adj[point[j]].push_back(i);
66             }
67         }
68     }
69     return {cnt, adj};
70 }

```

```

71     void work() {
72         for (int i = 1; i <= n; ++i) { // 避免图不连通
73             if (!dfn[i]) {
74                 tarjan(i, i);
75             }
76         }
77     }
78 };

```

染色法判定二分图 (dfs算法)

判断一张图能否被二分染色。

```

1  vector<int> vis(n + 1);
2  auto dfs = [&](auto self, int x, int type) -> void {
3      vis[x] = type;
4      for (auto y : ver[x]) {
5          if (vis[y] == type) {
6              cout << "NO\n";
7              exit(0);
8          }
9          if (vis[y]) continue;
10         self(self, y, 3 - type);
11     }
12 };
13 for (int i = 1; i <= n; ++i) {
14     if (vis[i]) {
15         dfs(dfs, i, 1);
16     }
17 }
18 cout << "Yes\n";

```

链式前向星建图与搜索

很少使用这种建图法。**dfs**：标准复杂度为 $\mathcal{O}(N + M)$ 。节点子节点的数量包含它自己（至少为 1），深度从 0 开始（根节点深度为 0）。**bfs**：深度从 1 开始（根节点深度为 1）。**topsort**：有向无环图（包括非联通）才拥有完整的拓扑序列（故该算法也可用于判断图中是否存在环）。每次找到入度为 0 的点并将其放入待查找队列。

```

1  namespace Graph {
2      const int N = 1e5 + 7;
3      const int M = 1e6 + 7;
4      int tot, h[N], ver[M], ne[M];
5      int deg[N], vis[M];
6
7      void clear(int n) {
8          tot = 0; //多组样例清空
9          for (int i = 1; i <= n; ++i) {
10             h[i] = 0;
11             deg[i] = vis[i] = 0;
12         }
13     }

```

```

14 void add(int x, int y) {
15     ver[++tot] = y, ne[tot] = h[x], h[x] = tot;
16     ++deg[y];
17 }
18 void dfs(int x) {
19     a.push_back(x); // DFS序
20     siz[x] = vis[x] = 1;
21     for (int i = h[x]; i; i = ne[i]) {
22         int y = ver[i];
23         if (vis[y]) continue;
24         dis[y] = dis[x] + 1;
25         dfs(y);
26         siz[x] += siz[y];
27     }
28     a.push_back(x);
29 }
30 void bfs(int s) {
31     queue<int> q;
32     q.push(s);
33     dis[s] = 1;
34     while (!q.empty()) {
35         int x = q.front();
36         q.pop();
37         for (int i = h[x]; i; i = ne[i]) {
38             int y = ver[i];
39             if (dis[y]) continue;
40             d[y] = d[x] + 1;
41             q.push(y);
42         }
43     }
44 }
45 bool topsort() {
46     queue<int> q;
47     vector<int> ans;
48     for (int i = 1; i <= n; ++i)
49         if (deg[i] == 0) q.push(i);
50     while (!q.empty()) {
51         int x = q.front();
52         q.pop();
53         ans.push_back(x);
54         for (int i = h[x]; i; i = ne[i]) {
55             int y = ver[i];
56             --deg[y];
57             if (deg[y] == 0) q.push(y);
58         }
59     }
60     return ans.size() == n; //判断是否存在拓扑排序
61 }
62 } // namespace Graph

```

一般图最大匹配（带花树算法）

与二分图匹配的差别在于图中可能存在奇环，时间复杂度与边的数量无关，为 $\mathcal{O}(N^3)$ 。下方模板编号从 0 开始，例题为 [UOJ #79. 一般图最大匹配](#)。

```

1  struct Graph {
2      int n;
3      vector<vector<int>>> e;
4      Graph(int n) : n(n), e(n) {}
5      void add(int u, int v) {
6          e[u].push_back(v);
7          e[v].push_back(u);
8      }
9      pair<int, vector<int>> work() {
10         vector<int> match(n, -1), vis(n), link(n), f(n), dep(n);
11         auto find = [&](int u) {
12             while (f[u] != u) u = f[u] = f[f[u]];
13             return u;
14         };
15         auto lca = [&](int u, int v) {
16             u = find(u), v = find(v);
17             while (u != v) {
18                 if (dep[u] < dep[v]) swap(u, v);
19                 u = find(link[match[u]]);
20             }
21             return u;
22         };
23         queue<int> q;
24         auto blossom = [&](int u, int v, int p) {
25             while (find(u) != p) {
26                 link[u] = v;
27                 v = match[u];
28                 if (vis[v] == 0) {
29                     vis[v] = 1;
30                     q.push(v);
31                 }
32                 f[u] = f[v] = p;
33                 u = link[v];
34             }
35         };
36         auto augment = [&](int u) {
37             while (!q.empty()) q.pop();
38             iota(f.begin(), f.end(), 0);
39             fill(vis.begin(), vis.end(), -1);
40             q.push(u);
41             vis[u] = 1;
42             dep[u] = 0;
43             while (!q.empty()) {
44                 int u = q.front();
45                 q.pop();
46                 for (auto v : e[u]) {
47                     if (vis[v] == -1) {

```



```

48         vis[v] = 0;
49         link[v] = u;
50         dep[v] = dep[u] + 1;
51         if (match[v] == -1) {
52             for (int x = v, y = u, temp; y != -1;
53                 x = temp, y = x == -1 ? -1 : link[x]) {
54                 temp = match[y];
55                 match[x] = y;
56                 match[y] = x;
57             }
58             return;
59         }
60         vis[match[v]] = 1;
61         dep[match[v]] = dep[u] + 2;
62         q.push(match[v]);
63     } else if (vis[v] == 1 && find(v) != find(u)) {
64         int p = lca(u, v);
65         blossom(u, v, p);
66         blossom(v, u, p);
67     }
68 }
69 }
70 };
71 auto greedy = [&]() {
72     for (int u = 0; u < n; ++u) {
73         if (match[u] != -1) continue;
74         for (auto v : e[u]) {
75             if (match[v] == -1) {
76                 match[u] = v;
77                 match[v] = u;
78                 break;
79             }
80         }
81     }
82 };
83 greedy();
84 for (int u = 0; u < n; u++) {
85     if (match[u] == -1) {
86         augment(u);
87     }
88 }
89 int ans = 0;
90 for (int u = 0; u < n; u++) {
91     if (match[u] != -1) {
92         ans++;
93     }
94 }
95 return {ans / 2, match};
96 }
97 };
98
99 signed main() {
100     int n, m;

```

```

101     cin >> n >> m;
102
103     Graph graph(n);
104     for (int i = 1; i <= m; i++) {
105         int x, y;
106         cin >> x >> y;
107         graph.add(x - 1, y - 1);
108     }
109     auto [ans, match] = graph.work();
110     cout << ans << endl;
111     for (auto it : match) {
112         cout << it + 1 << " ";
113     }
114 }

```

一般图最大权匹配（带权带花树算法）

下方模板编号从 1 开始，复杂度为 $\mathcal{O}(N^3)$ 。

```

1  namespace Graph {
2      const int N = 403 * 2; //两倍点数
3      typedef int T; //权值大小
4      const T inf = numeric_limits<int>::max() >> 1;
5      struct Q { int u, v; T w; } e[N][N];
6      T lab[N];
7      int n, m = 0, id, h, t, lk[N], sl[N], st[N], f[N], b[N][N], s[N], ed[N], q[N];
8      vector<int> p[N];
9      #define dvd(x) (lab[x.u] + lab[x.v] - e[x.u][x.v].w * 2)
10     #define FOR(i, b) for (int i = 1; i <= (int)(b); i++)
11     #define ALL(x) (x).begin(), (x).end()
12     #define ms(x, i) memset(x + 1, i, m * sizeof x[0])
13     void upd(int u, int v) {
14         if (!sl[v] || dvd(e[u][v]) < dvd(e[sl[v]][v])) {
15             sl[v] = u;
16         }
17     }
18     void ss(int v) {
19         sl[v] = 0;
20         FOR(u, n) {
21             if (e[u][v].w > 0 && st[u] != v && !s[st[u]]) {
22                 upd(u, v);
23             }
24         }
25     }
26     void ins(int u) {
27         if (u <= n) { q[++t] = u; }
28         else {
29             for (int v : p[u]) ins(v);
30         }
31     }
32     void mdf(int u, int w) {
33         st[u] = w;

```

```

34     if (u > n) {
35         for (int v : p[u]) mdf(v, w);
36     }
37 }
38 int gr(int u, int v) {
39     v = find(ALL(p[u]), v) - p[u].begin();
40     if (v & 1) {
41         reverse(1 + ALL(p[u]));
42         return (int)p[u].size() - v;
43     }
44     return v;
45 }
46 void stm(int u, int v) {
47     lk[u] = e[u][v].v;
48     if (u <= n) return;
49     Q w = e[u][v];
50     int x = b[u][w.u], y = gr(u, x);
51     for (int i = 0; i < y; i++) {
52         stm(p[u][i], p[u][i ^ 1]);
53     }
54     stm(x, v);
55     rotate(p[u].begin(), y + ALL(p[u]));
56 }
57 void aug(int u, int v) {
58     int w = st[lk[u]];
59     stm(u, v);
60     if (!w) return;
61     stm(w, st[f[w]]), aug(st[f[w]], w);
62 }
63 int lca(int u, int v) {
64     for (++id; u | v; swap(u, v)) {
65         if (!u) continue;
66         if (ed[u] == id) return u;
67         ed[u] = id;
68         if (u = st[lk[u]]) u = st[f[u]];
69     }
70     return 0;
71 }
72 void add(int u, int a, int v) {
73     int x = n + 1, i, j;
74     while (x <= m && st[x]) ++x;
75     if (x > m) ++m;
76     lab[x] = s[x] = st[x] = 0;
77     lk[x] = lk[a];
78     p[x].clear();
79     p[x].push_back(a);
80     for (i = u; i != a; i = st[f[j]]) {
81         p[x].push_back(i);
82         p[x].push_back(j = st[lk[i]]);
83         ins(j);
84     }
85     reverse(1 + ALL(p[x]));
86     for (i = v; i != a; i = st[f[j]]) { // 复制，只需改循环

```

```

87         p[x].push_back(i);
88         p[x].push_back(j = st[lk[i]]);
89         ins(j);
90     }
91     mdf(x, x);
92     FOR(i, m) {
93         e[x][i].w = e[i][x].w = 0;
94     }
95     memset(b[x] + 1, 0, n * sizeof b[0][0]);
96     for (int u : p[x]) {
97         FOR(v, m) {
98             if (!e[x][v].w || dvd(e[u][v]) < dvd(e[x][v])) {
99                 e[x][v] = e[u][v], e[v][x] = e[v][u];
100             }
101         }
102         FOR(v, n) {
103             if (b[u][v]) { b[x][v] = u; }
104         }
105     }
106     ss(x);
107 }
108 void ex(int u) {
109     for (int x : p[u]) mdf(x, x);
110     int a = b[u][e[u][f[u]].u], r = gr(u, a);
111     for (int i = 0; i < r; i += 2) {
112         int x = p[u][i], y = p[u][i + 1];
113         f[x] = e[y][x].u;
114         s[x] = 1;
115         s[y] = sl[x] = 0;
116         ss(y), ins(y);
117     }
118     s[a] = 1, f[a] = f[u];
119     for (int i = r + 1; i < p[u].size(); i++) {
120         s[p[u][i]] = -1;
121         ss(p[u][i]);
122     }
123     st[u] = 0;
124 }
125 bool on(const Q &e) {
126     int u = st[e.u], v = st[e.v];
127     if (s[v] == -1) {
128         f[v] = e.u, s[v] = 1;
129         int a = st[lk[v]];
130         sl[v] = sl[a] = s[a] = 0;
131         ins(a);
132     } else if (!s[v]) {
133         int a = lca(u, v);
134         if (!a) {
135             return aug(u, v), aug(v, u), 1;
136         } else {
137             add(u, a, v);
138         }
139     }

```

```

140     return 0;
141 }
142 bool bfs() {
143     ms(s, -1), ms(sl, 0);
144     h = 1, t = 0;
145     FOR(i, m) {
146         if (st[i] == i && !lk[i]) {
147             f[i] = s[i] = 0;
148             ins(i);
149         }
150     }
151     if (h > t) return 0;
152     while (1) {
153         while (h <= t) {
154             int u = q[h++];
155             if (s[st[u]] == 1) continue;
156             FOR(v, n) {
157                 if (e[u][v].w > 0 && st[u] != st[v]) {
158                     if (dvd(e[u][v])) upd(u, st[v]);
159                     else if (on(e[u][v])) return 1;
160                 }
161             }
162         }
163         T x = inf;
164         for (int i = n + 1; i <= m; i++) {
165             if (st[i] == i && s[i] == 1) {
166                 x = min(x, lab[i] >> 1);
167             }
168         }
169         FOR(i, m) {
170             if (st[i] == i && sl[i] && s[i] != 1) {
171                 x = min(x, dvd(e[sl[i]][i]) >> s[i] + 1);
172             }
173         }
174         FOR(i, n) {
175             if (~s[st[i]]) {
176                 if ((lab[i] += (s[st[i]] * 2 - 1) * x) <= 0) return 0;
177             }
178         }
179         for (int i = n + 1; i <= m; i++) {
180             if (st[i] == i && ~s[st[i]]) {
181                 lab[i] += (2 - s[st[i]] * 4) * x;
182             }
183         }
184         h = 1, t = 0;
185         FOR(i, m) {
186             if (st[i] == i && sl[i] && st[sl[i]] != i && !dvd(e[sl[i]][i]) &&
on(e[sl[i]][i])) {
187                 return 1;
188             }
189         }
190         for (int i = n + 1; i <= m; i++) {
191             if (st[i] == i && s[i] == 1 && !lab[i]) ex(i);

```

```

192     }
193 }
194 return 0;
195 }
196 template<typename TT> i64 work(int N, const vector<tuple<int, int, TT>> &edges) {
197     ms(ed, 0), ms(lk, 0);
198     n = m = N; id = 0;
199     iota(st + 1, st + n + 1, 1);
200     T wm = 0; i64 r = 0;
201     FOR(i, n) FOR(j, n) {
202         e[i][j] = {i, j, 0};
203     }
204     for (auto [u, v, w] : edges) {
205         wm = max(wm, e[v][u].w = e[u][v].w = max(e[u][v].w, (T)w));
206     }
207     FOR(i, n) { p[i].clear(); }
208     FOR(i, n) FOR(j, n) {
209         b[i][j] = i * (i == j);
210     }
211     fill_n(lab + 1, n, wm);
212     while (bfs()) {};
213     FOR(i, n) if (lk[i]) {
214         r += e[i][lk[i]].w;
215     }
216     return r / 2;
217 }
218 auto match() {
219     vector<array<int, 2>> ans;
220     FOR(i, n) if (lk[i]) {
221         ans.push_back({i, lk[i]});
222     }
223     return ans;
224 }
225 } // namespace Graph
226 using Graph::work, Graph::match;
227
228 signed main() {
229     int n, m;
230     cin >> n >> m;
231     vector<tuple<int, int, i64>> ver(m);
232     for (auto &[u, v, w] : ver) {
233         cin >> u >> v >> w;
234     }
235     cout << work(n, ver) << "\n";
236     auto ans = match();
237 }

```

二分图最大匹配

二分图：一个图能被分为左右两部分，任何一条边的两个端点都不在同一部分中。

匹配（独立边集）：一个边的集合，这些边没有公共顶点。

二分图最大匹配即找到边的数量最多的那个匹配。

一般我们规定，左半部包含 n_1 个点（编号 $1 - n_1$ ），右半部包含 n_2 个点（编号 $1 - n_2$ ），保证任意一条边的两个端点都不可能在同一部分中。

匈牙利算法（KM算法）解

$\mathcal{O}(NM)$ 。

```

1  signed main() {
2      int n1, n2, m;
3      cin >> n1 >> n2 >> m;
4
5      vector<vector<int>> ver(n1 + 1);
6      for (int i = 1; i <= m; ++i) {
7          int x, y;
8          cin >> x >> y;
9          ver[x].push_back(y); //只需要建立单向边
10     }
11
12     int ans = 0;
13     vector<int> match(n2 + 1);
14     for (int i = 1; i <= n1; ++i) {
15         vector<int> vis(n2 + 1);
16         auto dfs = [&](auto self, int x) -> bool {
17             for (auto y : ver[x]) {
18                 if (vis[y]) continue;
19                 vis[y] = 1;
20                 if (!match[y] || self(self, match[y])) {
21                     match[y] = x;
22                     return true;
23                 }
24             }
25             return false;
26         };
27         if (dfs(dfs, i)) {
28             ans++;
29         }
30     }
31     cout << ans << endl;
32 }
```

HopcroftKarp算法（基于最大流）解

该算法基于最大流，常数极小，且引入随机化，几乎卡不掉。最坏时间复杂度为 $\mathcal{O}(\sqrt{NM})$ ，经[测试](#)，在 N, M 均为 2×10^5 的情况下能在 60ms 内跑完。

```

1 struct HopcroftKarp {
2     int n, m;
3     vector<array<int, 2>> ver;
4     vector<int> l, r;
5
6     HopcroftKarp(int n, int m) : n(n), m(m) { // 左右半部
7         l.assign(n, -1);
8         r.assign(m, -1);
9     }
10    void add(int x, int y) {
11        x--, y--; // 这个板子是 0-idx 的
12        ver.push_back({x, y});
13    }
14    int work() {
15        vector<int> adj(ver.size());
16
17        mt19937 rgen(chrono::steady_clock::now().time_since_epoch().count());
18        shuffle(ver.begin(), ver.end(), rgen); // 随机化防卡
19
20        vector<int> deg(n + 1);
21        for (auto &[u, v] : ver) {
22            deg[u]++;
23        }
24        for (int i = 1; i <= n; i++) {
25            deg[i] += deg[i - 1];
26        }
27        for (auto &[u, v] : ver) {
28            adj[--deg[u]] = v;
29        }
30
31        int ans = 0;
32        vector<int> a, p, q(n);
33        while (true) {
34            a.assign(n, -1), p.assign(n, -1);
35
36            int t = 0;
37            for (int i = 0; i < n; i++) {
38                if (l[i] == -1) {
39                    q[t++] = a[i] = p[i] = i;
40                }
41            }
42
43            bool match = false;
44            for (int i = 0; i < t; i++) {
45                int x = q[i];
46                if (~l[a[x]]) continue;
47
48                for (int j = deg[x]; j < deg[x + 1]; j++) {

```



```

49         int y = adj[j];
50         if (r[y] == -1) {
51             while (~y) {
52                 r[y] = x;
53                 swap(l[x], y);
54                 x = p[x];
55             }
56             match = true;
57             ++ans;
58             break;
59         }
60         if (p[r[y]] == -1) {
61             q[t++] = y = r[y];
62             p[y] = x;
63             a[y] = a[x];
64         }
65     }
66 }
67 if (!match) break;
68 }
69 return ans;
70 }
71 vector<array<int, 2>> answer() {
72     vector<array<int, 2>> ans;
73     for (int i = 0; i < n; i++) {
74         if (~l[i]) {
75             ans.push_back({i, l[i]});
76         }
77     }
78     return ans;
79 }
80 };
81
82 signed main() {
83     int n1, n2, m;
84     cin >> n1 >> n2 >> m;
85     HopcroftKarp flow(n1, n2);
86     while (m--) {
87         int x, y;
88         cin >> x >> y;
89         flow.add(x, y);
90     }
91
92     cout << flow.work() << "\n";
93
94     auto match = flow.answer();
95     for (auto [u, v] : match) {
96         cout << u << " " << v << "\n";
97     }
98 }

```

二分图最大权匹配（二分图完美匹配）

定义：找到边权和最大的那个匹配。

一般我们规定，左半部包含 n_1 个点（编号 $1 - n_1$ ），右半部包含 n_2 个点（编号 $1 - n_2$ ）。

使用匈牙利算法（KM算法）解，时间复杂度为 $\mathcal{O}(N^3)$ 。下方模板用于求解最大权值、且可以输出其中一种可行方案，例题为 [UOJ #80. 二分图最大权匹配](#)。

```

1  struct MaxCostMatch {
2      vector<int> ans1, ansr, pre;
3      vector<int> lx, ly;
4      vector<vector<int>> ver;
5      int n;
6
7      MaxCostMatch(int n) : n(n) {
8          ver.resize(n + 1, vector<int>(n + 1));
9          ans1.resize(n + 1, -1);
10         ansr.resize(n + 1, -1);
11         lx.resize(n + 1);
12         ly.resize(n + 1, -1E18);
13         pre.resize(n + 1);
14     }
15     void add(int x, int y, int w) {
16         ver[x][y] = w;
17     }
18     void bfs(int x) {
19         vector<bool> visl(n + 1), visr(n + 1);
20         vector<int> slack(n + 1, 1E18);
21         queue<int> q;
22         function<bool(int)> check = [&](int x) {
23             visr[x] = 1;
24             if (~ansr[x]) {
25                 q.push(ansr[x]);
26                 visl[ansr[x]] = 1;
27                 return false;
28             }
29             while (~x) {
30                 ansr[x] = pre[x];
31                 swap(x, ans1[pre[x]]);
32             }
33             return true;
34         };
35         q.push(x);
36         visl[x] = 1;
37         while (1) {
38             while (!q.empty()) {
39                 int x = q.front();
40                 q.pop();
41                 for (int y = 1; y <= n; ++y) {
42                     if (visr[y]) continue;
43                     int del = lx[x] + ly[y] - ver[x][y];
44                     if (del < slack[y]) {

```

```

45         pre[y] = x;
46         slack[y] = del;
47         if (!slack[y] && check(y)) return;
48     }
49 }
50 }
51 int val = 1E18;
52 for (int i = 1; i <= n; ++i) {
53     if (!visr[i]) {
54         val = min(val, slack[i]);
55     }
56 }
57 for (int i = 1; i <= n; ++i) {
58     if (visl[i]) lx[i] -= val;
59     if (visr[i]) {
60         ly[i] += val;
61     } else {
62         slack[i] -= val;
63     }
64 }
65 for (int i = 1; i <= n; ++i) {
66     if (!visr[i] && !slack[i] && check(i)) {
67         return;
68     }
69 }
70 }
71 }
72 int work() {
73     for (int i = 1; i <= n; ++i) {
74         for (int j = 1; j <= n; ++j) {
75             ly[i] = max(ly[i], ver[j][i]);
76         }
77     }
78     for (int i = 1; i <= n; ++i) bfs(i);
79     int res = 0;
80     for (int i = 1; i <= n; ++i) {
81         res += ver[i][ansl[i]];
82     }
83     return res;
84 }
85 void getMatch(int x, int y) { // 获取方案 (0代表无匹配)
86     for (int i = 1; i <= x; ++i) {
87         cout << (ver[i][ansl[i]] ? ansl[i] : 0) << " ";
88     }
89     cout << endl;
90     for (int i = 1; i <= y; ++i) {
91         cout << (ver[i][ansr[i]] ? ansr[i] : 0) << " ";
92     }
93     cout << endl;
94 }
95 };
96
97 signed main() {

```

```

98     int n1, n2, m;
99     cin >> n1 >> n2 >> m;
100
101     MaxCostMatch match(max(n1, n2));
102     for (int i = 1; i <= m; i++) {
103         int x, y, w;
104         cin >> x >> y >> w;
105         match.add(x, y, w);
106     }
107     cout << match.work() << '\n';
108     match.getMatch(n1, n2);
109 }

```

二分图最大独立点集（Konig 定理）

给出一张二分图，要求选择一些点使得它们两两没有边直接连接。最小点覆盖等价于最大匹配数，转换为最小割模板，答案即为总点数减去最大流得到的值。

```

1 | cout << n - flow.work(s, t) << endl;

```

最长路（topsort+DP算法）

计算一张 DAG 中的最长路径，在执行前可能需要使用 tarjan 重构一张正确的 DAG，复杂度 $\mathcal{O}(N + M)$ 。

```

1 | struct DAG {
2 |     int n;
3 |     vector<vector<pair<int, int>>> ver;
4 |     vector<int> deg, dis;
5 |     DAG(int n) : n(n) {
6 |         ver.resize(n + 1);
7 |         deg.resize(n + 1);
8 |         dis.assign(n + 1, -1E18);
9 |     }
10 | void add(int x, int y, int w) {
11 |     ver[x].push_back({y, w});
12 |     ++deg[y];
13 | }
14 | int topsort(int s, int t) {
15 |     queue<int> q;
16 |     for (int i = 1; i <= n; i++) {
17 |         if (deg[i] == 0) {
18 |             q.push(i);
19 |         }
20 |     }
21 |     dis[s] = 0;
22 |     while (!q.empty()) {
23 |         int x = q.front();
24 |         q.pop();
25 |         for (auto [y, w] : ver[x]) {
26 |             dis[y] = max(dis[y], dis[x] + w);
27 |             --deg[y];
28 |             if (deg[y] == 0) {

```

```

29         q.push(y);
30     }
31 }
32 }
33 return dis[t];
34 }
35 };
36
37 signed main() {
38     int n, m;
39     cin >> n >> m;
40     DAG dag(n);
41     for (int i = 1; i <= m; i++) {
42         int x, y, w;
43         cin >> x >> y >> w;
44         dag.add(x, y, w);
45     }
46
47     int s, t;
48     cin >> s >> t;
49     cout << dag.topsort(s, t) << "\n";
50 }

```

最短路径树（SPT问题）

定义：在一张无向带权联通图中，有这样一棵**生成树**：满足从根节点到任意点的路径都为原图中根到任意点的最短路径。

性质：记根节点 $Root$ 到某一结点 x 的最短距离 $dis_{Root,x}$ ，在 SPT 上这两点之间的距离为 $len_{Root,x}$ ——则两者长度相等。

该算法与最小生成树无关，基于最短路 **Dijkstra** 算法完成（但多了个等于号）。下方代码实现的功能为：读入图后，输出以 1 为根的 SPT 所使用的各条边的编号、边权和。

```

1  map<pair<int, int>, int> id;
2  namespace G {
3      vector<pair<int, int> > ver[N];
4      map<pair<int, int>, int> edge;
5      int v[N], d[N], pre[N], vis[N];
6      int ans = 0;
7
8      void add(int x, int y, int w) {
9          ver[x].push_back({y, w});
10         edge[{x, y}] = edge[{y, x}] = w;
11     }
12     void djikstra(int s) { // ! 注意, 该 djikstra 并非原版, 多了一个等于号
13         priority_queue<PII, vector<PII>, greater<PII> > q; q.push({0, s});
14         memset(d, 0x3f, sizeof d); d[s] = 0;
15         while (!q.empty()) {
16             int x = q.top().second; q.pop();
17             if (v[x]) continue; v[x] = 1;
18             for (auto [y, w] : ver[x]) {
19                 if (d[y] >= d[x] + w) { // ! 注意, SPT 这里修改为>=号

```

```

20         d[y] = d[x] + w;
21         pre[y] = x; // 记录前驱结点
22         q.push({d[y], y});
23     }
24 }
25 }
26 }
27 void dfs(int x) {
28     vis[x] = 1;
29     for (auto [y, w] : ver[x]) {
30         if (vis[y]) continue;
31         if (pre[y] == x) {
32             cout << id[{x, y}] << " "; // 输出SPT所使用的边编号
33             ans += edge[{x, y}];
34             dfs(y);
35         }
36     }
37 }
38 void solve(int n) {
39     djikstra(1); // 以 1 为根
40     dfs(1); // 以 1 为根
41     cout << endl << ans; // 输出SPT的边权和
42 }
43 }
44 bool solve() {
45     int n, m; cin >> n >> m;
46     for (int i = 1; i <= m; ++i) {
47         int x, y, w; cin >> x >> y >> w;
48         G::add(x, y, w), G::add(y, x, w);
49         id[{x, y}] = id[{y, x}] = i;
50     }
51     G::solve(n);
52     return 0;
53 }

```

无源汇点的最小割问题 Stoer-Wagner

也称为全局最小割。定义补充（与《网络流》中的定义不同）：

割：是一个边集，去掉其中所有边能使一张网络流图不再连通（即分成两个子图）。

通过**递归**的方式来解决**无向正权图**上的全局最小割问题，算法复杂度 $\mathcal{O}(VE + V^2 \log V)$ ，一般可近似看作 $\mathcal{O}(V^3)$ 。

```

1  signed main() {
2      int n, m;
3      cin >> n >> m;
4
5      DSU dsu(n); // 这里引入DSU判断图是否联通，如题目有保证，则不需要此步骤
6      vector<vector<int>> edge(n + 1, vector<int>(n + 1));
7      for (int i = 1; i <= m; i++) {
8          int x, y, w;
9          cin >> x >> y >> w;

```

```

10     dsu.merge(x, y);
11     edge[x][y] += w;
12     edge[y][x] += w;
13 }
14
15 if (dsu.Poi(1) != n || m < n - 1) { // 图不联通
16     cout << 0 << endl;
17     return 0;
18 }
19
20 int MinCut = INF, S = 1, T = 1; // 虚拟源汇点
21 vector<int> bin(n + 1);
22 auto contract = [&]() -> int { // 求解S到T的最小割, 定义为 cut of phase
23     vector<int> dis(n + 1), vis(n + 1);
24     int Min = 0;
25     for (int i = 1; i <= n; i++) {
26         int k = -1, maxc = -1;
27         for (int j = 1; j <= n; j++) {
28             if (!bin[j] && !vis[j] && dis[j] > maxc) {
29                 k = j;
30                 maxc = dis[j];
31             }
32         }
33         if (k == -1) return Min;
34         S = T, T = k, Min = maxc;
35         vis[k] = 1;
36         for (int j = 1; j <= n; j++) {
37             if (!bin[j] && !vis[j]) {
38                 dis[j] += edge[k][j];
39             }
40         }
41     }
42     return Min;
43 };
44 for (int i = 1; i < n; i++) { // 这里取不到等号
45     int val = contract();
46     bin[T] = 1;
47     MinCut = min(MinCut, val);
48     if (!MinCut) {
49         cout << 0 << endl;
50         return 0;
51     }
52     for (int j = 1; j <= n; j++) {
53         if (!bin[j]) {
54             edge[S][j] += edge[j][T];
55             edge[j][S] += edge[j][T];
56         }
57     }
58 }
59 cout << MinCut << endl;
60 }

```

欧拉路径/欧拉回路 Hierholzers

欧拉路径：一笔画完图中全部边，画的顺序就是一个可行解；当起点终点相同时称欧拉回路。

有向图欧拉路径存在判定

有向图欧拉路径存在：¹ 恰有一个点出度比入度多 1（为起点）；² 恰有一个点入度比出度多 1（为终点）；³ 恰有 $N - 2$ 个点入度均等于出度。如果是欧拉回路，则上方起点与终点的条件不存在，全部点均要满足最后一个条件。

```

1  signed main() {
2      int n, m;
3      cin >> n >> m;
4
5      DSU dsu(n + 1); // 如果保证连通，则不需要 DSU
6      vector<unordered_multiset<int>> ver(n + 1); // 如果对于字典序有要求，则不能使用
unordered
7      vector<int> degI(n + 1), degO(n + 1);
8      for (int i = 1; i <= m; i++) {
9          int x, y;
10         cin >> x >> y;
11         ver[x].insert(y);
12         degI[y]++;
13         degO[x]++;
14         dsu.merge(x, y); // 直接当无向图
15     }
16     int s = 1, t = 1, cnt = 0;
17     for (int i = 1; i <= n; i++) {
18         if (degI[i] == degO[i]) {
19             cnt++;
20         } else if (degI[i] + 1 == degO[i]) {
21             s = i;
22         } else if (degI[i] == degO[i] + 1) {
23             t = i;
24         }
25     }
26     if (dsu.size(1) != n || (cnt != n - 2 && cnt != n)) {
27         cout << "No\n";
28     } else {
29         cout << "Yes\n";
30     }
31 }
```

无向图欧拉路径存在判定

无向图欧拉路径存在：¹ 恰有两个点度数为奇数（为起点与终点）；² 恰有 $N - 2$ 个点度数为偶数。

```

1  signed main() {
2      int n, m;
3      cin >> n >> m;
4
5      DSU dsu(n + 1); // 如果保证连通，则不需要 DSU
```



```

6     vector<unordered_multiset<int>> ver(n + 1); // 如果对于字典序有要求，则不能使用
    unordered
7     vector<int> deg(n + 1);
8     for (int i = 1; i <= m; i++) {
9         int x, y;
10        cin >> x >> y;
11        ver[x].insert(y);
12        ver[y].insert(x);
13        deg[y]++;
14        deg[x]++;
15        dsu.merge(x, y); // 直接当无向图
16    }
17    int s = -1, t = -1, cnt = 0;
18    for (int i = 1; i <= n; i++) {
19        if (deg[i] % 2 == 0) {
20            cnt++;
21        } else if (s == -1) {
22            s = i;
23        } else {
24            t = i;
25        }
26    }
27    if (dsu.size(1) != n || (cnt != n - 2 && cnt != n)) {
28        cout << "No\n";
29    } else {
30        cout << "Yes\n";
31    }
32 }

```

有向图欧拉路径求解（字典序最小）

```

1     vector<int> ans;
2     auto dfs = [&](auto self, int x) -> void {
3         while (ver[x].size()) {
4             int net = *ver[x].begin();
5             ver[x].erase(ver[x].begin());
6             self(self, net);
7         }
8         ans.push_back(x);
9     };
10    dfs(dfs, s);
11    reverse(ans.begin(), ans.end());
12    for (auto it : ans) {
13        cout << it << " ";
14    }

```

无向图欧拉路径求解

```

1  auto dfs = [&](auto self, int x) -> void {
2      while (ver[x].size()) {
3          int net = *ver[x].begin();
4          ver[x].erase(ver[x].find(net));
5          ver[net].erase(ver[net].find(x));
6          cout << x << " " << net << endl;
7          self(self, net);
8      }
9  };
10 dfs(dfs, s);

```

差分约束

给出一组包含 m 个不等式，有 n 个未知数的形如：

$$\begin{cases} x_{u_1} - x_{v_1} \leq w_1 \\ x_{u_2} - x_{v_2} \leq w_2 \\ \dots \\ x_{u_m} - x_{v_m} \leq w_m \end{cases}$$

的不等式组，求任意一组满足这个不等式组的解。SPFA 解， $\mathcal{O}(nm)$ 。[参考](#)

```

1  signed main() {
2      int n, m;
3      cin >> n >> m;
4
5      vector<array<int, 3>> e(m + 1);
6      for (int i = 1; i <= m; i++) {
7          int u, v, w;
8          cin >> u >> v >> w;
9          e[i] = {v, u, w};
10     }
11
12     vector<int> d(n + 1, 1E9);
13     d[1] = 0;
14     for (int i = 1; i < n; i++) {
15         for (int j = 1; j <= m; j++) {
16             auto [u, v, w] = e[j];
17             d[v] = min(d[v], d[u] + w);
18         }
19     }
20     for (int i = 1; i <= m; i++) {
21         auto [u, v, w] = e[i];
22         if (d[v] > d[u] + w) {
23             cout << "NO\n";
24             return 0;
25         }
26     }
27     for (int i = 1; i <= n; i++) {
28         cout << d[i] << " \n"[i == n];
29     }
30     return 0;
31 }

```

2-Sat

基础封装

基于 tarjan 缩点，时间复杂度为 $\mathcal{O}(N + M)$ 。注意下标从 0 开始，答案输出为字典序最小的一个可行解。

```

1  struct TwoSat {
2      int n;
3      vector<vector<int>> e;
4      vector<bool> ans;
5      TwoSat(int n) : n(n), e(2 * n), ans(n) {}
6      void add(int u, bool f, int v, bool g) {
7          e[2 * u + !f].push_back(2 * v + g);
8          e[2 * v + !g].push_back(2 * u + f);
9      }
10     bool work() {
11         vector<int> id(2 * n, -1), dfn(2 * n, -1), low(2 * n, -1);
12         vector<int> stk;
13         int now = 0, cnt = 0;
14         auto tarjan = [&](auto self, int u) -> void {
15             stk.push_back(u);
16             dfn[u] = low[u] = now++;
17             for (auto v : e[u]) {
18                 if (dfn[v] == -1) {
19                     self(self, v);
20                     low[u] = min(low[u], low[v]);
21                 } else if (id[v] == -1) {
22                     low[u] = min(low[u], dfn[v]);
23                 }
24             }
25             if (dfn[u] == low[u]) {
26                 int v;
27                 do {
28                     v = stk.back();
29                     stk.pop_back();
30                     id[v] = cnt;
31                 } while (v != u);
32                 ++cnt;
33             }
34         };
35         for (int i = 0; i < 2 * n; ++i) {
36             if (dfn[i] == -1) {
37                 tarjan(tarjan, i);
38             }
39         }
40         for (int i = 0; i < n; ++i) {
41             if (id[2 * i] == id[2 * i + 1]) return false;
42             ans[i] = id[2 * i] > id[2 * i + 1];
43         }
44         return true;
45     }
46     vector<bool> answer() {
47         return ans;

```

```

48     }
49 };

```

答案不唯一时不输出

在运行后针对每一个点进行一次 dfs，时间复杂度为 $\mathcal{O}(N^2)$ ，当且仅当答案唯一时才输出，否则输出 ? 替代。

2-Sat方案计数为 NPC 问题。

```

1  // 结构体中增加
2  int check(int x, int y) {
3      vector<int> vis(2 * n);
4      auto dfs = [&](auto self, int x) -> void {
5          vis[x] = 1;
6          for (auto y : e[x]) {
7              if (vis[y]) continue;
8              self(self, y);
9          }
10     };
11     dfs(dfs, x);
12     return vis[y];
13 }
14 // 主函数中增加
15 for (int i = 0; i < n; i++) {
16     if (sat.check(2 * i, 2 * i + 1)) {
17         cout << 1 << " ";
18     } else if (sat.check(2 * i + 1, 2 * i)) {
19         cout << 0 << " ";
20     } else {
21         cout << "?" << " ";
22     }
23 }

```