

一般图最大权（最大）匹配

建议首先搞懂二分图最大匹配、二分图最大权（最大）匹配、一般图最大匹配这几个 special case，另外可以先阅读参考文献，其中 [1] 讲的是最好的。

问题描述

给定一个带权无向图 $\langle V, E \rangle$ 不失一般性，可以认为算法竞赛中遇到的题目里，权值均为非负整数。一般图最大权匹配，是指使得边权和最大的匹配；一般图最大权最大匹配，是指使得边权和最大的最大匹配。

对偶问题

对每个点 u 定义一个对偶变量 y_u 对每个奇数大小的点集 B 定义一个对偶变量 z_B 对于一条边 uv 定义它的松弛变量 $yz_{uv} = y_u + y_v + \sum_{u, v \in B} z_B - w_{uv}$ 可以看到，和 KM 算法相比，唯一的区别是 z 变量。那么为什么要引入 z 变量呢？由于一般图中存在奇环，或者说匹配的过程中存在花，那么在松弛的时候，不论如何给环中的点赋予类型，奇环中总有一条边要同时加上或者减去一个值，这将使得这条边的紧性遭到破坏 Edmond 提出的方法是，缩花后，将花中的所有点看作同一类型，同时加上 δ 后，给 z_B 减去 2δ 补足即可。

这一对偶问题需要满足几个条件：

1. $z_B, yz_{uv} \geq 0$
2. 若 uv 是匹配边或是形成花的边（花环中的其它弦是不算的） $yz_{uv} = 0$
3. 仅当 B 中恰有 $\frac{|B|-1}{2}$ 条匹配边时 z_B 可以非零（注意到花满足这一条件）

算法流程

将所有 y_u 的初始赋为 $\frac{\max w}{2}$ 所有 z_B 的初值赋为 0，没有匹配边和花。可以看到，初始时条件是满足的。定义所有 $yz = 0$ 的边为紧边，注意到这显然包括了所有的匹配边和花边。称所有紧边组成的生成子图为相等子图。

与 KM 算法类似，逐步在相等子图中寻找增广路，如果找不到，就对对偶变量进行松弛，以得到更多的紧边。与 KM 算法有一些区别的是，在一般图最大权匹配时，将所有未盖点设为初始点，并标记为 S 点。算法维护一个队列，每次取出队列中的一个点，考虑它所有紧的邻边，分情况讨论：

1. 若它的 peer 是 free 点（即尚未打标记的点，由于未打标记，一定是匹配点），那么将 peer 标记为 T 点，将 peer 匹配的边标记为 S 点，并加入队列
2. 若它的 peer 是 T 点，那么忽略
3. 若它的 peer 是 S 点，若它们在交错森林的同一棵树中，那么找到了一朵花；若在同一朵花中，忽略；否则也找到了一条增广路，即从根走到当前点，然后到 peer 然后到 peer 的根

考虑发现花的情况，显然新花应当被标记为 S 点，另外新花的 z 设为 0。且花也可能发生嵌套，形成一棵花的森林，但是这个花森林至多只有 $\mathcal{O}(n)$ 个点（因为一朵花至少由 3 个点缩成 1 个点）。需要证明缩点时上述三个条件仍然满足：

1. 由于缩点不修改对偶变量，因此 1 显然满足
2. 注意到搜索是在相等子图中进行的，因此所有新产生的花边显然都是紧边

3. 显然

这个过程构建出了一个交错森林。注意到每个点要么是单点，要么是（最外层的）花。对于一朵花中的所有点，我们将它的类型 S/T 规定为最外层花的类型。

假如找到了增广路，需要验证增广后上述三个条件仍然满足：

1. 由于增广不修改对偶变量，因此 1 显然满足
2. 由于是在相等子图中增广，因此所有匹配边仍然是相等子图中的边。增广时虽然可能改变花托的位置，但是花的结构不变，因此 2 也满足
3. 与 2 同理

假如找不到增广路，需要对对偶变量进行调整，也就是进行松弛操作。不妨将所有的 S 点减去 δ ，那么为了保证所有的匹配边和花边仍是紧边，需要给所有 T 点加上 δ ，所有的 S 花的 z 加上 2δ ，所有的 T 花的 z 减去 2δ 。注意到所有的新花都是 S 花，为了使得 $z \geq 0$ ，这就要求 $\delta > 0$ 。考虑到 y 的初值是 $\frac{1}{2}$ 的倍数，不妨将 δ 定为 $\frac{1}{2}$ 。修改后，显然条件 3 是满足的，但是其它条件则都需要仔细分析。

1. $z_B \geq 0$ 这就要求所有的奇花原先的 z 值大于 0。因此，在每一轮结束后，应当把所有 z 为 0 的 T 花展开，展开后的子 T 花如果仍为 0，还需要继续展开。发生一次增广后，需要将所有 z 为 0 的花展开，以防在下一轮产生 $z=0$ 的 T 花。
2. $S-T$ 边， $T-T$ 边， $T-\text{free}$ 边显然都满足 $yz \geq 0$ 。对于所有的 $S-\text{free}$ 边，由于它之前一定不是紧的，因此可以减掉 $\frac{1}{2}$ 。对于所有的 $S-S$ 边，首先它一定不是紧的，否则已经找到了增广路或花。对于当前所有未盖点，注意到它们历史上也一直是未盖点，从而一直是 S 点，因此它们的值是相同的。考虑所有已经标记 S/T 的点，由于它们通过紧边与未盖点连通，而 z 又是 1 的倍数，因此所有已标记点乘 2 模 2 同余。因此这条 $S-S$ 边至少有 1 可以减掉，从而也是合法的。
3. 注意到交错森林中的所有边，以及所有 S/T 花边仍然保持是紧的。匹配边还可能是 $\text{free}-\text{free}$ 边，显然也是紧的，花还可能是 free 花，显然也是紧的。那么条件 2 也是成立的。

当然，只改变 $\frac{1}{2}$ 可能使得相等子图根本没有发生变化。容易发现，需要考虑的是以下 3 种情况：

1. $\min yz_{uv}$ 其中 u 是 S 点， v 是 free 点
2. $\min \frac{yz_{uv}}{2}$ 其中 u, v 是不同花中的 S 点
3. $\min z_B$ 其中 B 是 T 花

当 δ 取以上 3 种情况的最小值时，显然相等子图将发生改变。

对于一般图最大权最大匹配，当找不到增广路，也找不到上述 3 种情况中的任何一种时，算法停止。

对于一般图最大权匹配，需要增加 $y_u \geq 0$ 作为条件。当找不到增广路，也找不到上述 3 种情况中的任何一种时，取 $\delta = \min y_u$ 。当 $\min y_u \leq \delta$ 时，算法也停止。这里需要注意，未盖点的 y 总是最小的，原因同上，因此这个 $\min$ 即为未盖点的 y 。

正确性证明

一般图最大权匹配的情况比较简单。对于找到的匹配，它的权值和为

$$\sum_{i=1}^n y_i + \sum_B z_B \cdot \frac{|B|-1}{2}$$

而对于任意一个匹配，它的权值和都小于等于这个值。

对于一般图最大权最大匹配的情况，首先需要证明找到的是最大匹配。算法结束时，图中非交错森林边的只有 $T-T$ 边， $T-S$ 边和 $T-\text{free}$ 边，且没有 T 花。可以发现，即使加入这些不紧的边，交错森林仍保持不变。由于没有增广路，因此找到的已经是最大匹配。

设所有匹配点为 M 它的权值和为

$$\sum_{i \in M} y_i + \sum_{B} z_B \cdot \frac{|B|-1}{2}$$

而对于任意一个最大匹配，它的权值和

$$\leq \sum_{i \in \text{any set with size } |M|} y_i + \sum_{B} z_B \cdot \frac{|B|-1}{2}$$

由于匹配点的权总大于任何未盖点的权，因此找到的是最大权最大匹配。

复杂度分析

显然增广只会有 $\mathcal{O}(n)$ 次。一次增广中会有若干次松弛，每次松弛可能发生

1. 松弛一条 $S-S$ 边，出现增广路
2. 松弛一条 $S-S$ 边，出现一朵新花
3. 松弛一条 $S-\text{free}$ 边，交错树中新增一个点
4. 松弛一朵 T 花，并将它展开

1 显然只会出现 1 次。2 只会出现 $\mathcal{O}(n)$ 次（本轮增广的过程中产生的花全是 S 花，也不会在本轮增广中变成 T 花，是不会被删除的）。3 只会出现 $\mathcal{O}(n)$ 次，因为交错森林中的边是不会变成不紧的，故点也不会离开交错森林。4 只会出现 $\mathcal{O}(n)$ 次，因为 T 花必然是上一轮增广中留下来的花，本轮增广不可能产生 T 花。

一次松弛显然可以通过 $\mathcal{O}(m)$ 遍历整个图计算得出，因而时间复杂度是 $\mathcal{O}(n^2 m)$ 的。

优化

利用 KM 算法中的 slack 变量，可以将复杂度优化到 $\mathcal{O}(n^3)$ 。上述算法复杂度瓶颈在于 $S-\text{free}$ 边和 $S-S$ 边的计算。对于 $S-\text{free}$ 边，维护方法与 KM 中完全相同。

但是 $S-S$ 边额外要求它们不在同一朵花中。考虑维护任意两朵 S 花之间的最小边权，及达到最小边权的边。由于增广时不会展开 S 花，可以发现 S 点只能产生不会消失。其中一部分复杂度是产生新的 S 花时维护与其它 S 花的最小边权，这可以暴力枚举边来实现，复杂度为 $\mathcal{O}(m)$ 。另一部分复杂度是缩花时需要重新计算该花到其它 S 花的最小边权（其它 S 花到它的最小边权显然不变）。注意到组成它的子花中 S 花占了约一半，而一次增广中所有花的子花数量显然是 $\mathcal{O}(n)$ 的。因此，我们可以直接暴力枚举所有 n 个点来进行更新，一次增广的复杂度是 $\mathcal{O}(n^2)$ 的。在这个基础上，就可以维护 slack 变量了。

总复杂度 $\mathcal{O}(n^3)$

实现细节

1. 给一个点赋 `label` 即要给它所在顶层花赋 `label` 这点要时刻注意，很容易出错。
2. 对于 S -`free` 边的 `slack` 变量，由于 T 和 `free` 点会相互转换，因此对 T 和 `free` 点都维护 `slack` 变量会比较方便。具体来说，对每个非 S 点维护它到所有点的 `slack` 考虑到 S 点只会产生而不会消失，因此只需要在新产生 S 点时更新 `slack` 至于一个点是否是 `free` 点，计算前检查一下即可。
3. `slack` 变量只记录在哪条边取到，不记值，会比较方便。
4. 事实上，一次松弛可能导致多种改变，但是我们只需要处理其中一种即可。下次 δ 会变成 0 ，但是没关系，这时图同样会发生改变（是上次残留下来没有做完的改变）。同理每次增广最后也不需要真的展开 $z_B=0$ 的奇花，它们会在下一轮增广时被处理掉。

好像暂时想不到什么了。

碎碎念

证明中好几处用到了 y 的初值完全相等这个性质，但是个人感觉正确性应该不依赖于它。个人能力所限，没法找到更优雅的证明了。

参考代码

鄙人的渣渣代码成功在 UOJ 上跑出了倒数第三的排名，真是太菜了 :(大家不要学我，找份好一点的代码看看吧。

```
#include <bits/stdc++.h>

namespace GMW{
    const int N = 1010;
    const int M = 100100;
    using T = long long;
    const T INF = 1e18;
    const bool max_cardinality = false;

    int n;
    std::vector<int> e[N];
    int color[N * 2], slack[N * 2], slackl[N * 2][N * 2]; T dual[N * 2];
    int queue[N], head, tail; bool inque[N * 2];
    int vis[N], viscnt;
    int nxt[N], pre[N]; // nxt is the matching edge, pre is the parent edge
    in the alternating forest
    int uu[M], vv[M], ecnt; T w[M];

    std::vector<int> children[N * 2], children_e[N * 2], blossom_queue;
    int fa[N * 2], top[N * 2], base[N * 2];

    inline int peer(int u, int edge){ return top[uu[edge]] == top[u] ?
vv[edge] : uu[edge]; }
```

```

inline T get_slack(int edge){ return edge == -1 ? LLONG_MAX :
dual[uu[edge]] + dual[vv[edge]] - 2 * w[edge]; }
inline void update(int &e1, int e2){ if (e1 == -1 || get_slack(e1) >
get_slack(e2)){e1 = e2;} }

int new_blossom(){
    int id = blossom_queue.back();
    blossom_queue.pop_back();
    children[id].clear();
    children_e[id].clear();
    fa[id] = dual[id] = 0;
    slack[id] = -1;
    memset(slackl[id], -1, sizeof(slackl[id]));
    return id;
}

void addedge(int u, int v, T weight){
    uu[ecnt] = u;
    vv[ecnt] = v;
    w[ecnt] = weight;
    e[u].emplace_back(ecnt);
    e[v].emplace_back(ecnt);
    ecnt ++;
}

void settop(int u, int tp){
    top[u] = tp;
    for (auto v : children[u]){
        settop(v, tp);
    }
}

inline void push(int u){
    if (inque[u]) return;
    inque[u] = true;
    queue[tail ++] = u;
    for (auto edge : e[u]){
        int v = uu[edge] ^ vv[edge] ^ u;
        if (color[top[v]] != 0){
            update(slack[v], edge);
        }
    }
}

void slack_type3(int b){
    if (slackl[b][0] != -1) return;
    slack[b] = -1;
    if (b <= n){
        for (auto edge : e[b]){
            int v = b ^ uu[edge] ^ vv[edge];
            if (top[b] != top[v] && color[top[v]] == 0){

```

```
        update(slack1[b][top[v]], edge);
        update(slack[b], edge);
    }
}
slack1[b][0] = 0;
return;
}
for (auto u : children[b]){
    slack_type3(u);
    for (int i = 1; i <= 2 * n; ++ i){
        if (top[i] != top[b] && color[top[i]] == 0){
            update(slack1[b][top[i]], slack1[u][i]);
            update(slack[b], slack1[u][i]);
        }
    }
}
slack1[b][0] = 0;
}

void push_to_queue(int u){
    if (u <= n) push(u);
    for (auto v : children[u]){
        push_to_queue(v);
    }
}

void setcolor(int u, int col){
    u = top[u];
    if (color[u] == col){
        return;
    }
    color[u] = col;
    if (col == 0){
        push_to_queue(u);
        slack_type3(u);
    }
    else if (col == 1){
        int edge = nxt[base[u]];
        assert(edge >= 0);
        int v = top[peer(u, edge)];
        setcolor(v, 0);
    }
}

int lca(int u, int v){ // to modify
    ++ viscnt;
    u = top[u], v = top[v];
    while (u || v){
        if (u){
            if (vis[u] == viscnt) return u;
        }
    }
}
```

```

        vis[u] = viscnt;
        u = top[peer(u, nxt[base[u]]]);
        u = top[peer(u, pre[u])];
    }
    std::swap(u, v);
}
return 0;
}

int adjust(int b, int edge){
    auto get_child = [&](int u){
        while (u && fa[u] != b){
            u = fa[u];
        }
        return u;
    };
    int ch = get_child(uu[edge]) | get_child(vv[edge]);
    int pos = std::find(children[b].begin(), children[b].end(), ch) -
children[b].begin();
    if (pos % 2){
        std::reverse(children[b].begin() + 1, children[b].end());
        std::reverse(children_e[b].begin(), children_e[b].end());
        int sz = children[b].size();
        pos = (sz - pos) % sz;
    }
    return pos;
}

void augment_blossom(int b, int edge){
    if (b <= n) return;
    int pos = adjust(b, edge);
    auto &ch = children[b];
    auto &che = children_e[b];
    augment_blossom(ch[pos], edge);
    for (int i = pos - 1; i >= 0; i -= 2){
        int ed = che[i - 1];
        augment_blossom(ch[i - 1], ed);
        augment_blossom(ch[i], ed);
        nxt[uu[ed]] = nxt[vv[ed]] = ed;
    }
    std::rotate(ch.begin(), ch.begin() + pos, ch.end());
    std::rotate(che.begin(), che.begin() + pos, che.end());
    base[b] = base[ch[0]];
}

bool linkSS(int u, int v, int edge){ // return found augment path
    int l = lca(u, v);
    if (l == 0){ // found an augment path
        for (int _ = 0; _ < 2; ++ _){
            int x = u, y = v, e1 = edge;
            while (x){

```

```
        int tmp = nxt[base[top[x]]];
        nxt[x] = e1;
        augment_blossom(top[x], e1);
        if (tmp == -1) break;
        y = peer(x, tmp);
        e1 = pre[top[y]];
        x = peer(y, e1), y = peer(x, e1);
        nxt[y] = e1;
        augment_blossom(top[y], e1);
    }
    std::swap(u, v);
}
return true;
}
int id = new_blossom();
auto &ch = children[id];
auto &che = children_e[id];
for (int _ = 0; _ < 2; ++ _){
    int x = top[u], e1 = edge;
    while (true){
        ch.emplace_back(x);
        che.emplace_back(e1);
        if (x == l) break;
        int e2 = nxt[base[x]];
        int y = top[peer(x, e2)];
        ch.emplace_back(y);
        che.emplace_back(e2);
        e1 = pre[y];
        x = top[peer(y, e1)];
    }
    std::swap(u, v);
    std::reverse(ch.begin(), ch.end());
    std::reverse(che.begin(), che.end());
    if (_ == 0) che.pop_back();
}
ch.pop_back();
base[id] = base[ch[0]];
pre[id] = pre[ch[0]];
for (auto x : ch){
    fa[x] = id;
}
settop(id, id);
setcolor(id, 0);
return false;
}

void extract(int u){
    int edge = pre[u];
    int pos = adjust(u, edge);
    auto &ch = children[u];
```

```

    auto &che = children_e[u];
    for (int i = 0; i < int(ch.size()); ++ i){
        fa[ch[i]] = 0, settop(ch[i], ch[i]);
        setcolor(ch[i], i <= pos ? ((i % 2) ^ 1) : -1);
    }
    pre[ch[pos]] = edge;
    for (int i = 0; i < pos; i += 2){
        pre[ch[i]] = che[i];
    }
    blossom_queue.emplace_back(u);
    top[u] = 0;
}

bool match(){
    memset(color, -1, sizeof(color));
    memset(slack, -1, sizeof(slack));
    memset(slack1, -1, sizeof(slack1));
    memset(pre, -1, sizeof(pre));
    memset(inque, 0, sizeof(inque));
    head = tail = 0;
    for (int i = 1; i <= n; ++ i){
        if (nxt[i] == -1){
            setcolor(i, 0);
        }
    }
    while (true){
        for (; head < tail; ++ head){
            int u = queue[head];
            for (auto edge : e[u]){
                if (get_slack(edge)){
                    continue;
                }
                int v = peer(u, edge);
                if (top[u] == top[v]){
                    continue;
                }
                if (color[top[v]] == -1){ // must be a matched point
                    assert(nxt[v] != -1);
                    pre[top[v]] = edge;
                    setcolor(v, 1);
                }
                else if (color[top[v]] == 0){
                    if (linkSS(u, v, edge)){
                        return true;
                    }
                }
            }
        }
    }
    int type = 0, pos = 0;
    T delta = LLONG_MAX;
    for (int i = 1; i <= n; ++ i){

```

```
T val = get_slack(slack[i]);
if (color[top[i]] == -1 && val < delta){
    delta = val, type = 1, pos = i;
}
}
for (int i = 1; i <= 2 * n; ++ i){
    T val = get_slack(slack[i]) / 2;
    if (top[i] == i && color[i] == 0 && val < delta){
        delta = val, type = 2, pos = i;
    }
}
for (int i = n + 1; i <= 2 * n; ++ i){
    if (top[i] == i && color[i] == 1 && dual[i] / 2 < delta){
        delta = dual[i] / 2, type = 3, pos = i;
    }
}
if (delta >= INF){
    break;
}
if (!max_cardinality){
    T min_value = *std::min_element(dual + 1, dual + 1 + n);
    if (min_value < delta){
        break;
    }
}
for (int i = 1; i <= n; ++ i){
    int col = color[top[i]];
    dual[i] += col == 0 ? -delta : col == 1 ? delta : 0;
}
for (int i = n + 1; i <= 2 * n; ++ i){
    if (top[i] == i){
        int col = color[i];
        dual[i] += col == 0 ? 2 * delta : col == 1 ? -2 * delta
: 0;
    }
}
}
if (type == 1){
    assert(nxt[pos] != -1);
    pre[top[pos]] = slack[pos];
    setcolor(pos, 1);
}
else if (type == 2){
    int edge = slack[pos];
    if (linkSS(uu[edge], vv[edge], edge)){
        return true;
    }
}
else{ // type == 3
    extract(pos);
}
```

```
    }
    return false;
}

// max weight (max cardinality, can control via max_cardinality)
general matching
// remember to set n
T solve(){
    T max_weight = *std::max_element(w, w + ecnt);
    for (int i = 1; i <= n; ++ i){
        dual[i] = max_weight;
        blossom_queue.emplace_back(i + n);
        fa[i] = 0;
        top[i] = base[i] = i;
    }
    while (match())
        ;
    T ans = 0;
    for (int i = 1; i <= n; ++ i){
        if (nxt[i] != -1){
            ans += w[nxt[i]];
        }
    }
    return ans / 2;
}

void init(){
    for (int i = 0; i < N; ++ i){
        e[i].clear();
    }
    ecnt = 0;
    viscnt = 0;
    memset(vis, 0, sizeof(vis));
    memset(nxt, -1, sizeof(nxt));
    memset(top, 0, sizeof(top));
    blossom_queue.clear();
}
}
```

参考文献

- [1]
<https://resources.mpi-inf.mpg.de/departments/d1/teaching/ss12/AdvancedGraphAlgorithms/Slides07.pdf>
- [2] Galil Z. Efficient algorithms for finding maximum matching in graphs[J]. ACM Computing Surveys (CSUR), 1986, 18(1): 23-38.
- [3] 国家集训队2015论文集 , 149-174.

[4] <http://jorisvr.nl/article/maximum-matching>

From:

<https://wiki.cvbbacm.com/> - **CVBB ACM Team**

Permanent link:

https://wiki.cvbbacm.com/doku.php?id=technique:general_matching_weighted



Last update: **2021/07/04 23:03**