二分图

二分图

基础概念

二分图：又称为二部图，如果一个图G=(V,E)的顶点可以分为两个部分，对所有边，其两端点都处于不同的部分，那么这个图就是二分图。

$$\text{若对于图}G=(V,E)\text{，存在}V\text{的一个划分}(A,B)\text{, 使}\forall(u,v)\in E,$$
$$\text{都有}u\in A,v\in B\text{或者}u\in B\text{，}v\in A\text{，则称图}G\text{为二分图}$$

$$\text{在图}G=(V,E)\text{中，边集}M\subset E\text{被称为}G\text{的一个匹配当且仅当}$$
$$\text{对于}V\text{中的每个点，}M\text{中与其关联的边都不超过一条。}$$

完美匹配：对于任意v，M中有且仅有一条边与其关联。

最大匹配：包含边数最多的匹配。

边覆盖：边集F满足G中任意顶点都至少是F中某条边的端点。

点覆盖：点集S满足G中任意边都至少又一个端点属于S。

独立集：点集S任意两点都没有边相连。

最大独立集：在图中选出最多的点使得任意两点之间没有边相连。

可行顶标：给每个节点i分配一个权值l(i),对于所有边(u,v)满足w(u,v)<=l(u)+l(v)。

相等子图：在一组可行顶标下原图的生成子图，包含所有点但只包含满足w(u,v)=l(u)+l(v)的边(u,v)。

最佳匹配：带权二分图的权值最大的完备匹配称为最佳匹配。

交错路：始于非匹配点且由匹配边与非匹配边交错而成。

增广路：是始于非匹配点且终于非匹配点的交错路。

二分图中的性质

最小点覆盖=最大匹配

最小边覆盖=|V|-最大匹配

最大独立集=|V|-最大匹配

二分图不存在长度为奇数的环

二分图判断

二染色：将所有顶点染成黑白两色，要求每条边的两个端点颜色不同，那么必然不存在奇环。因此，我们可以使用DFS或BFS遍历整张图，如果发现了奇环，那么就不是二分图。

二分图最大匹配

二分图是特殊的网络流，最佳匹配相当于求最大（小）费用最大流，用Ford-Fulkerson算法也能实现。二分图最大匹配也可以转化为网络流模型求解。

匈牙利算法

用于解决二分图最大匹配问题，该算法的核心就是寻找增广路径。

基本步骤

（1）首先从任意的一个未配对的点u开始，从点u的边中任意选一条边（假设这条边是从u->v）开始配对。如果点v未配对，则配对成功，这是便找到了一条增广路。如果点v已经被配对，就去尝试“连锁反应”，如果这时尝试成功，就更新原来的配对关系。

（2）如果刚才所选的边配对失败，那就要从点u的边中重新选一条边重新去试。直到点u 配对成功，或尝试过点u的所有边为止。

（3）接下来就继续对剩下的未配对过的点一一进行配对，直到所有的点都已经尝试完毕，找不到新的增广路为止。

（4）输出配对数。

```
//luoguP3386 【模板】二分图最大匹配
#include <bits/stdc++.h>

using namespace std;

int tot = 0, head[50005], cnt = 0, vis[501], fst[501];
//vis[i]表示i点最终的匹配，fst[i]表示每一轮i点有无被访问
struct X
{
    int next, to;
} edge[50005];

void addedge(int u, int v)
{ //链式前向星加边
```

```
    edge[++tot].next = head[u];
    edge[tot].to = v;
    head[u] = tot;
}

bool check(int x)
{
    for (int i = head[x]; i; i = edge[i].next)
    { //遍历每一条边
        int to = edge[i].to;
        if (!fst[to])
        { //如果右边的点没有被遍历
            fst[to] = 1;
            if (!vis[to] || check(vis[to]))
            { //如果右边的点没有匹配或者有其他的匹配
                vis[to] = x; //匹配两点然后返回
                return true;
            }
        }
    }
    return false; //找不到可匹配的点
}

int main()
{
    int n, m, e, u, v;
    scanf("%d%d%d", &n, &m, &e);
    for (int i = 1; i <= e; i++)
    {
        scanf("%d%d", &u, &v);
        addedge(u, v);
    }
    for (int i = 1; i <= n; i++)
    {  //对于每一个左边的点
        memset(fst, 0, sizeof(fst));
        if (check(i))
        {  //如果这个点能找到他的匹配
            cnt++;
        }
    }
    cout << cnt << endl;
    return 0;
}
```

二分图最大权匹配

最大完备匹配是最佳完美匹配（最大权匹配）特殊情况(所有边的权值都为1)。

若相等子图有**完美匹配**，则其必然为原图的**最大权完美匹配**。

KM算法

KM(Kuhn and Munkers)算法是匈牙利算法的一种贪心扩展，可以在$O(n^3)$时间内解决二分图最佳匹配问题。

通过给每个顶点一个标号（顶标）来把最大权匹配问题转化为求最大完备的问题。

对于某组可行顶标，如果其相等子图存在完美匹配，那么，该匹配就是原二分图的最大权完美匹配。

我们的目标就算通过不断地调整可行顶标，使得相等子图是完美匹配。

基本步骤

```
(1)初始化可行标杆
(2)用匈牙利算法寻找完备匹配
(3)若未找到完备匹配则修改可行标杆
(4)重复(2)、(3)直到找到相等子图的完备匹配
```

板子

```
//luoguP1559运动员最佳匹配问题
#include<bits/stdc++.h>
using namespace std;
const int inf = 0x3f3f3f3f;
int n, r, ans, a[25][25];//边权
int lx[25], ly[25]; //顶标
int visx[25], visy[25]; //标记
int match[25];//记录匹配对象
int minz;//记录最小的改变量
bool dfs(int s)  //寻找增广路
{
    visx[s] = 1; //s点参与了匹配
    for (int i = 1; i <= n; i++)
    {
        if (!visy[i])
        {  //如果没参与匹配
            int t = lx[s] + ly[i] - a[s][i]; //匹配原则是两边顶标和=边权
            if (!t)
            {
                visy[i] = 1; //本轮参与了匹配
                if (!match[i] || dfs(match[i])) //匈牙利算法
                { //如果i没被标记或者可以找到更优
                    match[i] = s; //匹配成功
                    return true;
                }
            }
            else
            {
                if (t > 0) //如果不符合匹配要求
                { //minz为参与匹配的所有男生换人所需要降低的最低期望的最小值。
                    minz = min(minz, t);
                }
            }
        }
    }
    return false;
}
```

```
void km()  //km算法本体
{
    for (int i = 1; i <= n; i++)
    {
        while (1)
        {
            minz = inf; //初始化变量
            memset(visx, 0, sizeof(visx));
            memset(visy, 0, sizeof(visy));
            if (dfs(i))  //如果找到匹配(增广路)
            {
                break; //找下一个匹配
            }
            for (int j = 1; j <= n; j++)
            {
                if (visx[j]) //这一轮j点参与了匹配
                {
                    lx[j] -= minz; //匹配那一方下降顶标
                }
            }
            for (int j = 1; j <= n; j++)
            {
                if (visy[j])
                {
                    ly[j] += minz; //被匹配那一方上升期望
                }
            }
        }
    }
}

signed main()
{
    ios::sync_with_stdio(0); //读入优化
    cin.tie(0);
    cout.tie(0);
    cin >> n;
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            cin >> a[i][j]; //这里是P[i][j]
        }
    }
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= n; j++)
        { //将图转化为无向图
            cin >> r;
            a[j][i] *= r; //P[i][j]*Q[j][i],不要反了
        }
    }
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            lx[i] = max(lx[i], a[i][j]); //顶标预处理
        }
```

```
    }
    km();
    for (int i = 1; i <= n; i++)
    {
        ans += a[match[i]][i];     //累加答案
    }
    cout << ans;
    return 0;
}
```

Slack+BFS优化

每次扩大相等子图最少只能加入一条相等边，也就是最多会进行$O(n^2)$次扩大相等子图。

每次扩大相等子图后都需要进行dfs增广，单次复杂度可达$O(n^2)$。

也就是说，km+dfs的复杂度完全可以卡到n^4$n4$。，在数据量大时是不可接受的。

使用slack优化KM算法，可以稳定在$O(n^3)$复杂度下完成程序。

在每次扩大子图后，都记录一下新加入的相等边所为我们提供的新增广方向，然后从此处继续寻找增广路即可。

```
void bfs(int u)
{
    int x, y = 0, yy = 1;
    memset(pre, 0, sizeof(pre)); //清空前驱
    for (int i = 1; i <= n; i++) slack[i] = inf;
    match[y] = u; //出发点的配对设为u
    while (1)
    {
        x = match[y]; //获取y当前匹配的左部点
        minz = inf; //松弛量
        vis[y] = idx; //本轮已经访问过y点
        for (int i = 1; i <= n; i++)
        {
            if (vis[i]==idx) continue; //已经访问过了
            if (slack[i] > lx[x] + ly[i] - mp[x][i])
            {
                slack[i] = lx[x] + ly[i] - mp[x][i];
                pre[i] = y; //记录i的前驱
            }
            if (slack[i] < minz)
            {
                minz = slack[i]; //记录最小松弛量
                yy = i; //记录来源
            }
        }
        for (int i = 0; i <= n; i++)
        {
            if (vis[i]==idx) //本轮参与匹配
            {       //更新顶标
                lx[match[i]] -= minz, ly[i] += minz;
            }
            else
            {
                slack[i] -= minz; //更新其他点的松弛量
            }
```

```
        }
        y = yy; //替换出发点
        if (!match[y]) break; //如果无法匹配了，跳出
    }
    while (y)
    {  //根据增广路记录配对
        match[y] = match[pre[y]];
        y = pre[y];
    }
}

ll KM()  //EK求最大权匹配
{
    for (int i = 1; i <= n; i++)
    {
        idx++; //时间戳代替memset
        bfs(i);
    }
    ll res = 0; //记录答案
    for (int i = 1; i <= n; i++)
    {
        if (match[i])
        {
            res += mp[match[i]][i];
        }
    }
    return res;
}
```

增广路

因为增广路长度为奇数，路径起始点非左即右，所以我们先考虑从左边的未匹配点找增广路。 注意到因为交错路的关系，增广路上的第奇数条边都是非匹配边，第偶数条边都是匹配边，于是左到右都是非匹配边，右到左都是匹配边。 于是我们给二分图 **定向**，问题转换成，有向图中从给定起点找一条简单路径走到某个未匹配点，此问题等价给定起始点 能否走到终点 。 那么只要从起始点开始 DFS 遍历直到找到某个未匹配点，。 未找到增广路时，我们拓展的路也称为 **交错树**。

增广路性质

（1）有奇数条边

（2）起点在二分图的X边，终点在二分图的Y边

（3）路径上的点一定是一个在X边，一个在Y边，交错出现。

（4）整条路径上没有重复的点

（5）起点和终点都是目前还没有匹配的点，其他的点都已经出现在匹配子图中。

（6）路径上的所有第奇数条边都是还没有进入目前的匹配子图的边，而所有第偶数条边都已经进入目前的匹配子图。奇数边比偶数边多一条边。

（7）于是当我们把所有第奇数条边都加到匹配子图并把偶数条边都删除，匹配数增加了1

```
//代码转载OI-wiki
```

```cpp
struct augment_path {
  vector<vector<int> > g;
  vector<int> pa;  // 匹配
  vector<int> pb;
  vector<int> vis;  // 访问
  int n, m;         // 两个点集中的顶点数量
  int dfn;          // 时间戳记
  int res;          // 匹配数

  augment_path(int _n, int _m) : n(_n), m(_m) {
    assert(0 <= n && 0 <= m);
    pa = vector<int>(n, -1);
    pb = vector<int>(m, -1);
    vis = vector<int>(n);
    g.resize(n);
    res = 0;
    dfn = 0;
  }

  void add(int from, int to) {
    assert(0 <= from && from < n && 0 <= to && to < m);
    g[from].push_back(to);
  }

  bool dfs(int v) {
    vis[v] = dfn;
    for (int u : g[v]) {
      if (pb[u] == -1) {
        pb[u] = v;
        pa[v] = u;
        return true;
      }
    }
    for (int u : g[v]) {
      if (vis[pb[u]] != dfn && dfs(pb[u])) {
        pa[v] = u;
        pb[u] = v;
        return true;
      }
    }
    return false;
  }

  int solve() {
    while (true) {
      dfn++;
      int cnt = 0;
      for (int i = 0; i < n; i++) {
        if (pa[i] == -1 && dfs(i)) {
          cnt++;
        }
      }
      if (cnt == 0) {
        break;
      }
      res += cnt;
    }
    return res;
```

```
  }
};
```

一般图最大匹配

开花算法

开花算法也叫带花树算法，可以用来解决一般图最大匹配问题。此算法是第一个给出证明说最大匹配有多项式复杂度。

一般图匹配和二分图的匹配的区别：**一般图可能存在奇环。**

```
// 转载自OI-WIKI
template <typename T>
class graph {
 public:
  struct edge {
    int from;
    int to;
    T cost;
  };
  vector<edge> edges;
  vector<vector<int> > g;
  int n;
  graph(int _n) : n(_n) { g.resize(n); }
  virtual int add(int from, int to, T cost) = 0;
};

// undirectedgraph
template <typename T>
class undirectedgraph : public graph<T> {
 public:
  using graph<T>::edges;
  using graph<T>::g;
  using graph<T>::n;

  undirectedgraph(int _n) : graph<T>(_n) {}
  int add(int from, int to, T cost = 1) {
    assert(0 <= from && from < n && 0 <= to && to < n);
    int id = (int)edges.size();
    g[from].push_back(id);
    g[to].push_back(id);
    edges.push_back({from, to, cost});
    return id;
  }
};

// blossom / find_max_unweighted_matching
template <typename T>
vector<int> find_max_unweighted_matching(const undirectedgraph<T> &g) {
  std::mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
  vector<int> match(g.n, -1);   // 匹配
  vector<int> aux(g.n, -1);     // 时间戳记
```

```
  vector<int> label(g.n);        // "o" or "i"
  vector<int> orig(g.n);         // 花根
  vector<int> parent(g.n, -1);   // 父节点
  queue<int> q;
  int aux_time = -1;

  auto lca = [&](int v, int u) {
    aux_time++;
    while (true) {
      if (v != -1) {
        if (aux[v] == aux_time) {  // 找到拜访过的点 也就是LCA
          return v;
        }
        aux[v] = aux_time;
        if (match[v] == -1) {
          v = -1;
        } else {
          v = orig[parent[match[v]]];  // 以匹配点的父节点继续寻找
        }
      }
      swap(v, u);
    }
  };  // lca

  auto blossom = [&](int v, int u, int a) {
    while (orig[v] != a) {
      parent[v] = u;
      u = match[v];
      if (label[u] == 1) {  // 初始点设为"o" 找增广路
        label[u] = 0;
        q.push(u);
      }
      orig[v] = orig[u] = a;  // 缩花
      v = parent[u];
    }
  };  // blossom

  auto augment = [&](int v) {
    while (v != -1) {
      int pv = parent[v];
      int next_v = match[pv];
      match[v] = pv;
      match[pv] = v;
      v = next_v;
    }
  };  // augment

  auto bfs = [&](int root) {
    fill(label.begin(), label.end(), -1);
    iota(orig.begin(), orig.end(), 0);
    while (!q.empty()) {
      q.pop();
    }
    q.push(root);
    // 初始点设为 "o", 这里以"0"代替"o", "1"代替"i"
    label[root] = 0;
    while (!q.empty()) {
      int v = q.front();
```

```
    q.pop();
    for (int id : g.g[v]) {
      auto &e = g.edges[id];
      int u = e.from ^ e.to ^ v;
      if (label[u] == -1) {  // 找到未拜访点
        label[u] = 1;        // 标记 "i"
        parent[u] = v;
        if (match[u] == -1) {  // 找到未匹配点
          augment(u);          // 寻找增广路径
          return true;
        }
        // 找到已匹配点 将与她匹配的点丢入queue 延伸交错树
        label[match[u]] = 0;
        q.push(match[u]);
        continue;
      } else if (label[u] == 0 && orig[v] != orig[u]) {
        // 找到已拜访点 且标记同为"o" 代表找到"花"
        int a = lca(orig[v], orig[u]);
        // 找LCA 然后缩花
        blossom(u, v, a);
        blossom(v, u, a);
      }
    }
  }
  return false;
};  // bfs

auto greedy = [&]() {
  vector<int> order(g.n);
  // 随机打乱 order
  iota(order.begin(), order.end(), 0);
  shuffle(order.begin(), order.end(), rng);

  // 将可以匹配的点匹配
  for (int i : order) {
    if (match[i] == -1) {
      for (auto id : g.g[i]) {
        auto &e = g.edges[id];
        int to = e.from ^ e.to ^ i;
        if (match[to] == -1) {
          match[i] = to;
          match[to] = i;
          break;
        }
      }
    }
  }
};  // greedy

// 一开始先随机匹配
greedy();
// 对未匹配点找增广路
for (int i = 0; i < g.n; i++) {
  if (match[i] == -1) {
    bfs(i);
  }
}
return match;
```

```
}
```

例题

稳定婚姻问题(Gale-Shapley算法)

这种算法会有男方最优化和女方最优化。

（1）所有男生均向自己最心仪的女生求婚，允许那个女生已经结婚，但不允许向同一个人求婚两次，”好马不吃回头草“；

（2）女生选择其中她最心仪的男生，如果已经结婚，则需要判断当前的男生是否比现任更好，如果是，则更换并让现任变回单身。

（3）循环步骤（1）（2）直到所有男生都不能求婚。

```
#include<stdio.h>
#include<queue>
#include<cstring>
#include<algorithm>
using namespace std;
#define N 35
#define inf 1<<29
#define MOD 2007
#define LL long long
using namespace std;

int couple;
int malelike[N][N],femalelike[N][N];
int malechoice[N],femalechoice[N];
int malename[N],femalename[N];
char str[N];
queue<int>freemale;

signed main(){
    int t;
    scanf("%d",&t);
    while(t--){
        scanf("%d",&couple);
        //情况队列
        while(!freemale.empty()){
            freemale.pop();
        }
        //将男士的名字存下，初始都没有匹配
        for(int i=0;i<couple;i++){
            scanf("%s",str);
            malename[i]=str[0]-'a';
            freemale.push(malename[i]);
        }
        //将名字排序，便于字典序
        sort(malename,malename+couple);
        for(int i=0;i<couple;i++){
            scanf("%s",str);
            femalename[i]=str[0]-'A';
        }
        //男士对女士的印象，按降序排列
```

```
        for(int i=0;i<couple;i++){
            scanf("%s",str);
            for(int j=0;j<couple;j++){
                malelike[i][j]=str[j+2]-'A';
            }
        }
        //女士对男士的打分，添加虚拟人物，编号couple，为女士的初始对象
        for(int i=0;i<couple;i++){
            scanf("%s",str);
            for(int j=0;j<couple;j++) femalelike[i][str[j+2]-'a']=couple-j;
            femalelike[i][couple]=0;
        }
        //一开始男士的期望都是最喜欢的女士
        memset(malechoice,0,sizeof(malechoice));
        //女士先初始一个对象
        for(int i=0;i<couple;i++) femalechoice[i]=couple;
        while(!freemale.empty()){
            //找出一个未配对的男士，注意不要习惯性的POP
            int male=freemale.front();
            //男士心仪的女士
            int female=malelike[male][malechoice[male]];
            //如果当前男士比原来的男友好
            if(femalelike[female][male]>femalelike[female]
[femalechoice[female]]){
                //成功脱光
                freemale.pop();
                //如果右前男友，则打回光棍，并且考虑下一个对象
                //不要把虚拟人物加入队列，否则会死循环或者错误
                if(femalechoice[female]!=couple){
                    freemale.push(femalechoice[female]);
                    malechoice[femalechoice[female]]++;;
                }
                //当前男友为这位男士
                femalechoice[female]=male;
            }else malechoice[male]++;//如果被女士拒绝，则要考虑下一个对象
        }
        for(int i=0;i<couple;i++){
            printf("%c %c\n",malename[i]+'a',malelike[malename[i]]
[malechoice[malename[i]]]+'A');
        }
        if(t) puts("");
    }
}
```

D - Transfer Window

有$n \leq 300$个球员，给出球员间两两替换关系，询问通过替换现有的 k个球员能否得到目标的 k个球员，并输出替换方案。（二分图匹配+Floyd传递闭包）

```
#include<bits/stdc++.h>
#define F first
#define S second
using namespace std;
const int N=305;
int n,k,a[N],b[N],mch[N];
```

```
char mp[N];
bool f[N][N],f_[N][N],vis[N],aa[N],bb[N];
vector<int> g[N],nd[2];
vector<pair<int,int> >rd,ans;
bool dfs1(int u){
    for(auto v:g[u]){
        if(vis[v])  continue;
        vis[v]=true;
        if(!mch[v]||dfs1(v)){
            mch[v]=u;
            return true;
        }
    }
    return false;
}
inline bool solve(int x){ //判是否能跑二分图
    int res=0;
    for(auto i:nd[0]){
        memset(vis,false,sizeof vis);
        res+=dfs1(i);
    }
    return res==x;
}
bool dfs2(int u,int v){ //存储路径
    vis[u]=1;
    if(u==v) return 1;
    for(int i=1;i<=n;++i){
        if(vis[i]||!f_[u][i]) continue;
        if(dfs2(i,v)){
            rd.push_back(make_pair(u,i));
            if(aa[u]){
                for(int i=rd.size()-1;~i;--i) ans.push_back(rd[i]);
                rd.clear();
            }
            return true;
        }
    }
    return false;
}
signed main(){
    ios::sync_with_stdio(0);
    cin.tie(0);cout.tie(0);
    scanf("%d%d",&n,&k);
    for(int i=1;i<=k;++i) scanf("%d",&a[i]),aa[a[i]]=1;
    for(int i=1;i<=k;++i) scanf("%d",&b[i]),bb[b[i]]=1;
    for(int i=1;i<=n;++i){
        scanf("%s",mp+1);
        for(int j=1;j<=n;++j) f_[i][j]=f[i][j]=mp[j]-'0';
    }
    for(int k=1;k<=n;++k)
        for(int i=1;i<=n;++i)
            for(int j=1;j<=n;++j)
                f[i][j]|=f[i][k]&f[k][j];  //floyd传递闭包
    for(int i=1;i<=n;++i) if(aa[i]^bb[i]) nd[bb[i]].push_back(i);
    for(auto u:nd[0]) for(auto v:nd[1]) if(f[u][v]) g[u].push_back(v);
    bool flag=solve(nd[0].size());
    if(!flag){printf("NO");return 0;}
    printf("YES\n");
```

```
    for(int i=1;i<=n;++i){
        if(bb[i]&&!aa[i]){
            memset(vis,false,sizeof(vis));
            flag=dfs2(mch[i],i);
            aa[mch[i]]=0,aa[i]=1;
        }
    }
    printf("%d\n",ans.size());
    for(auto to:ans) printf("%d %d\n",to.F,to.S);
    return 0;
}
```

参考资料

https://blog.csdn.net/yangss123/article/details/88716680

https://blog.csdn.net/sixdaycoder/article/details/47720471

https://oi-wiki.org/graph/graph-matching/bigraph-match/

https://oi-wiki.org/graph/graph-matching/bigraph-weight-match/

https://zhuanlan.zhihu.com/p/47114226

https://blog.csdn.net/li13168690086/article/details/81531258

https://www.codeleading.com/article/58755901749/